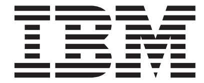
IBM® DB2® ユニバーサル・データベース

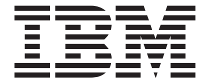


アプリケーション開発の手引き

バージョン 7

SC88-8516-01
(英文原典：SC09-2949-01)

IBM® DB2® ユニバーサル・データベース



アプリケーション開発の手引き

バージョン 7

SC88-8516-01
(英文原典：SC09-2949-01)

ご注意!

本書、および本書がサポートする製品をご使用になる前に、 855ページの『付録G. 特記事項』にある一般的な情報を必ずお読みください。

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

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)

原 典:	SC09-2949-01 IBM® DB2® Universal Database Application Development Guide Version 7
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 2001.8

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

© Copyright International Business Machines Corporation 1993, 2001. All rights reserved.

Translation: © Copyright IBM Japan 2001

目次

第1部 DB2 アプリケーション開発の概念 1

第1章 DB2 アプリケーション開発にあたって 3
本書について 3
本書の対象読者 4
本書の使用法 4
　規則 8
　関連資料 8

第2章 DB2 アプリケーションのコーディング 11
プログラミングの前提条件 12
DB2 アプリケーションのコーディングの概説 13
　変数の宣言および初期化 14
　データベース・サーバーへの接続 19
　トランザクションのコーディング 20
　プログラムの終了 22
　トランザクションの暗黙終了 22
　疑似コードによるアプリケーションの枠組み 23
DB2 アプリケーションの設計 24
　データへのアクセス 25
　データ値の制御 28
　データの関連の制御 30
　サーバーにおけるアプリケーションのロジック. 32
　IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ 33
サポートされる SQL ステートメント 37
許可についての考慮事項 37
　動的 SQL 37
　静的 SQL 38
　API の使用 39
　例 39
組み込み SQL または DB2 CLI プログラムで使用されるデータベース・マネージャー API . 40
テスト環境のセットアップ 40
　テスト・データベースの作成 41
　テスト表の作成 41
　テスト・データの生成. 42
プログラムの実行、テスト、およびデバッグ 44
SQL ステートメントのプロトタイプ化 . . . 44

第2部 アプリケーション内の組み込み SQL 47

第3章 組み込み SQL の概説 49
ホスト言語における組み込み SQL ステートメントの使用 49
ソース・ファイルの作成と準備. 51
組み込み SQL 用のパッケージの作成 . . . 54
　プリコンパイル 54
　コンパイルとリンク 57
　バインド 58
　バインドの実行据え置きの利点. 61
　DB2 バインド・ファイル記述ユーティリティー - db2bfd 62
　アプリケーション、バインド・ファイル、およびパッケージの関係 62
　タイム・スタンプ 63
　再バインド 64

第4章 静的 SQL プログラムの作成 67
静的 SQL を使用する場合の特性とそれを使用する理由 68
静的 SQL の利点 68
例: 静的 SQL プログラム 69
　静的プログラムの動作の仕組み. 70
　C の例: STATIC.SQC 72
　Java の例: Static.sqlj 73
　COBOL の例: STATIC.SQB 75
データの検索および操作のための SQL ステートメントのコーディング 77
　データの検索. 77
ホスト変数の使用 77
　宣言生成プログラム - db2dclgn 80
標識変数の使用 82
　データ・タイプ 84
　STATIC プログラムでの標識変数の使用. . 86
カーソルを用いた複数行の選択. 87
　カーソルの宣言と使用. 87
　カーソルおよび作業単位に関する考慮事項 88

© Copyright IBM Corp. 1993, 2001

例: カーソル・プログラム 90
検索済みデータの更新と削除 98
検索されたデータの更新 98
検索されたデータの削除 98
カーソルのタイプ 98
例: OPENFTCH プログラム 99
高度なスクロール技法 108
すでに検索済みデータのスクロール . . . 108
データのコピーを保持する方法 108
データを 2 度検索する方法 109
表の末尾の位置の確立 111
以前に検索されたデータの更新 111
例: UPDAT プログラム 111
診断処理と SQLCA 構造 122
戻りコード 122
SQLCODE および SQLSTATE 122
SQLCA 構造におけるトークンの切り捨て 123
WHENEVER ステートメントを用いたエラー処理 123
例外、シグナル、割り込みハンドラーについての考慮事項 124
出口リスト・ルーチンに関する考慮事項 125
プログラム例での GET ERROR MESSAGE の使用 125

第5章 動的 SQL プログラムの作成 133
動的 SQL の使用目的 133
動的 SQL サポート・ステートメント . . 133
動的 SQL と静的 SQL との比較 134
PREPARE、DESCRIBE、FETCH、および SQLDA の使用 137
カーソルの宣言および使用 138
例: 動的 SQL プログラム 140
SQLDA の宣言 150
最小の SQLDA 構造を用いたステートメントの準備 151
十分な数の SQLVAR 項目を指定した SQLDA の割り振り 152
SELECT ステートメントの記述 153
行を保持するためのストレージの獲得 . . 154
カーソルの処理 154
SQLDA 構造の割り振り 155
SQLDA 構造を使用するデータの受け渡し 158
対話式 SQL ステートメントの処理 . . . 159
エンド・ユーザーからの SQL 要求の保管 160
例: ADHOC プログラム 161
動的 SQL への変数入力 168
パラメーター・マーカーの使用 168
例: VARINP プログラム 169
DB2 コール・レベル・インターフェース (CLI) 177
組み込み SQL と DB2 CLI との比較 . . 177
DB2 CLI を使用する場合の利点 178
組み込み SQL か DB2 CLI かの決定 . . 180

第6章 一般的な DB2 アプリケーションの技法 183
生成列 184
識別列 184
| 順次値の生成 185
| シーケンスの振る舞いの制御 187
| シーケンス・オブジェクトによるパフォーマンスの向上 188
| シーケンス・オブジェクトと識別列の比較 189
宣言済み一時表 189
保管点によるトランザクションの制御 . . . 191
| アプリケーションの保管点と複合 SQL ブロックの比較 193
保管点の SQL ステートメントのリスト 194
保管点の制約事項 195
保管点およびデータ定義言語 (DDL) . . 195
保管点およびバッファー化挿入 197
カーソル・ブロック化付きの保管点の使用 197
保管点および XA に準拠したトランザクション・マネージャー 198

第3部 ストアード・プロシージャー 199

第7章 ストアード・プロシージャー . . . 201
ストアード・プロシージャーの概説 201
ストアード・プロシージャーの利点 202
ストアード・プロシージャーの作成 205
クライアント・アプリケーション . . . 207
サーバー上でのストアード・プロシージャー 208
OLE 自動化ストアード・プロシージャーの作成 227
OUT パラメーターのストアード・プロシージャーの例 228
コード・ページに関する考慮事項 . . . 241
C++ に関する考慮事項 241

グラフィック・ホスト変数に関する考慮事項 242
マルチサイト更新に関する考慮事項. . . 242
| ストアード・プロシージャーのパフォーマンスの向上. 242
| CHAR パラメーターではなく VARCHAR パラメーターを使用する 243
| DB2 がシステム・カタログ内のストアード・プロシージャーを検索するように強制する 243
| NOT FENCED ストアード・プロシージャー 244
ストアード・プロシージャーからの結果セットの戻り. 246
例: ストアード・プロシージャーからの結果セットを戻す 247
問題の解決 257

第8章 SQL プロシージャーの作成 259
SQL プロシージャーおよび外部プロシージャーの比較. 260
有効な SQL プロシージャー本体ステートメント 261
CREATE PROCEDURE ステートメントの発行 262
SQL プロシージャーでの処理条件 263
条件ハンドラーの宣言 264
SIGNAL および RESIGNAL ステートメント 266
SQL プロシージャーの SQLCODE および SQLSTATE 変数 266
SQL プロシージャーでの動的 SQL の使用 267
ネストされた SQL プロシージャー 269
ネストされた SQL プロシージャー間でのパラメーターの受け渡し 269
ネストされた SQL プロシージャーからの結果セットの戻り 269
ネストされた SQL プロシージャーでの制約事項 269
SQL プロシージャーからの結果セットの戻り 270
呼び出し元またはクライアントへ結果セットを戻す. 271
呼び出し元として結果セットを受け取る 272
SQL プロシージャーのデバッグ 273
SQL プロシージャーのエラー・メッセージを表示する 273
中間ファイルを使用した SQL プロシージャーのデバッグ 275
SQL プロシージャーの例 276

第9章 IBM DB2 ストアード・プロシージャー・ビルダー 283
ストアード・プロシージャー・ビルダーの説明. 283
ストアード・プロシージャー・ビルダーを使用する利点 284
ストアード・プロシージャーの新規作成 284
既存のストアード・プロシージャーでの作業 285
ストアード・プロシージャー・ビルダー・プロジェクトの作成 286
ストアード・プロシージャーのデバッグ 286

第4部 オブジェクト関連プログラミング 287

第10章 オブジェクト関連機能の使用 . . . 289
DB2 オブジェクト拡張を使用する理由. . . 289
DB2 のオブジェクト関連機能. 289

第11章 ユーザー定義特殊タイプ 295
特殊タイプを使用する理由. 295
特殊タイプの定義. 296
修飾されない特殊タイプの分析 296
CREATE DISTINCT TYPE の使用例 . . . 297
例: 通貨 297
例: ジョブ・アプリケーション 297
特殊タイプを使用した、表の定義 297
例: 売上 297
例: 応募の書式. 298
特殊タイプの操作. 298
特殊タイプの操作例 299
例: 特殊タイプと定数の比較 299
例: 特殊タイプ間のキャスト 300
例: 特殊タイプを使用した比較 301
例: 特殊タイプを使用したソース派生 UDF 302
例: 特殊タイプを使用した割り当て . . . 302
例: 動的 SQL での割り当て 302
例: 異なる特殊タイプを使用した割り当て 303
例: UNION 形式での特殊タイプの使用 304

第12章 複合オブジェクトの処理: ユーザー定義の構造型 305
構造型の概説 306
構造型階層の作成. 307
オブジェクトのタイプ付き表への保管 . . 313
オブジェクトの列への保管. 315
構造型の付加的な特性 316
タイプ付き表での構造型の使用 318
タイプ付き表の作成 318
タイプ付き表への挿入 320
参照タイプの使用. 322
参照タイプの比較. 323
タイプ付き視点の作成 325
ユーザー定義タイプ (UDT) またはタイプ・マッピングの除去 327
視点の更新または除去 328
タイプ付き表の照会 328
参照を逆参照する照会 329
付加的な照会仕様技法 331
その他のヒント 333
列タイプとしての構造型の作成と使用 . . . 335
構造型インスタンスの列への挿入 . . . 335
| 構造化タイプ属性を列に挿入する . . . 336
構造型列を持つ表の定義 336
構造型属性を持つタイプの定義 336
構造型値が入っている行の挿入 337
構造型値の検索と変更 338
transform のタイプとの関連付け 340
transform グループを指定する必要のある場合 342
ホスト言語プログラムへのマッピングの作成: transform 関数. 344
構造型ホスト変数の処理 363

第13章 ラージ・オブジェクト (LOB) の使用 365
LOB について 365
ラージ・オブジェクト・データ・タイプ (BLOB、CLOB、DBCLOB) について . . . 366
ラージ・オブジェクト・ロケーターについて 367
例: CLOB 値を使う作業のためのロケーターの使用 369
LOBLOC プログラム例の作動方法 . . . 369
C の例: LOBLOC.SQC 370
COBOL の例: LOBLOC.SQB 372
例: LOB 式の評価の据え置き 375
LOBEVAL プログラム例の作動方法. . . 376
C の例: LOBEVAL.SQC. 377
COBOL の例: LOBEVAL.SQB. 379
標識変数および LOB ロケーター . . . 382
LOB ファイル参照変数 382
例: ファイルへのドキュメントの抽出 . . . 384
LOBFILE プログラム例の作動方法 . . . 384
C の例: LOBFILE.SQC 385
COBOL の例: LOBFILE.SQB 386
例: CLOB 列へのデータの挿入 388

第14章 ユーザー定義関数 (UDF) およびメソッド 389
関数とメソッドについて 389
関数とメソッドを使用する理由 390
UDF とメソッドの概念 393
関数とメソッドのインプリメント 394
関数とメソッドの作成 395
関数とメソッドの登録 395
UDF とメソッドの登録例 395
例: べき乗 396
例: ストリングの検索 396
例: BLOB ストリング検索 397
例: UDT のストリング検索 398
例: UDT パラメーターを指定した外部関数 398
例: UDT での AVG 399
例: 計算 399
例: OLE オートメーション・オブジェクトを使った計算 400
例: 文書 ID を戻す表関数 400
関数とメソッドの使用法 401
関数の参照 401
関数呼び出しの例. 402
関数でのパラメーター・マーカーの使用法 403
修飾された関数参照の使用法 403
修飾されない関数参照の使用法 404
関数参照の要約 405

第15章 ユーザー定義関数 (UDF) とメソッドの作成 409
説明 409
DB2 と UDF 間のインターフェース . . . 411
DB2 から UDF に渡される引き数 . . . 411
UDF 引き数の使用の要約 424

SQL データ・タイプの UDF への受け渡し方法 426
32 ビット・プラットフォームおよび 64 ビット・プラットフォームでのスクラッチパッドの作成 434
UDF インクルード・ファイル: sqludf.h 435
Java ユーザー定義の関数の作成および使用 436
Java UDF のコーディング 437
Java UDF の実行の仕方の変更 438
Java の表関数実行モデル 439
OLE オートメーション UDF の作成 . . . 441
OLE オートメーション UDF の作成と登録 442
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 443
SQL データ・タイプの OLE オートメーション UDF への受け渡し方法 443
BASIC と C++ での OLE オートメーション UDF のインプリメンテーション . . 445
OLE DB 表関数 448
OLE DB 表関数の作成 449
完全修飾行セット名 451
OLE DB Provider のサーバー名の定義 452
ユーザー・マッピングの定義 453
サポートされる OLE DB データ・タイプ 453
スクラッチパッドに関する考慮事項. . . . 456
表関数に関する考慮事項 458
表関数のエラー処理 458
スカラー関数のエラー処理 459
UDF のパラメーターや結果としての LOB ロケーターの使用 459
LOB ロケーター使用のシナリオ 464
コーディングに関するその他の考慮事項 . . 464
ヒントとアドバイス 465
UDF に関する制限と警告 467
UDF コードの例 470
例: 整数除算演算子 470
例: CLOB の折り返し、母音の検出 . . . 474
例: カウンター. 478
例: 天気表関数. 480
例: LOB ロケーターを使用する関数 . . 488
例: BASIC でのカウンター OLE オートメーション UDF 491
例: C++ でのカウンター OLE オートメーション UDF. 492
UDF のデバッグ 497

第16章 活動状態の DBMS でのトリガーの使用 499
トリガーを使う理由 499
トリガーの利点 500
トリガーの概説 501
トリガー・イベント 502
影響される行のセット 503
トリガーの細分性. 503
トリガー活動化時間 503
変位変数. 505
変換表 506
トリガー・アクション 507
トリガー・アクション条件. 508
トリガー SQL ステートメント 508
SQL トリガー・ステートメント内の関数 509
トリガーのカスケード 510
参照制約との対話. 510
複数トリガーの順序付け 510
トリガー、制約、UDT、UDF、および LOB 間の協調. 511
情報の抽出 512
表における操作の妨害 512
業務規則の定義 513
アクションの定義. 513

第5部 DB2 プログラミングに関する考慮事項 515

第17章 複合環境におけるプログラミング 517
各国語サポートに関する考慮 517
照合順序の概説 518
コード・ページ値の導出 523
アプリケーション・プログラム中のロケールの導出. 524
各国語サポート・アプリケーションの開発 525
DBCS 文字セット. 532
拡張 UNIX コード (EUC) 文字セット 533
DBCS 環境での CLI/ODBC/JDBC/SQLJ プログラムの実行 534
日本語および中国語 (繁体字) EUC および UCS-2 コード・セットに関する考慮事項 535
マルチサイト更新に関する考慮事項. . . . 549
リモート作業単位. 549
マルチサイト更新. 549
ホストまたは AS/400 サーバーへのアクセス 556

マルチスレッドのデータベースのアクセス 557
マルチスレッドの使用に際しての推奨事項 558
コード・ページおよび国別/地域別コードを処理するマルチスレッド UNIX アプリケーション 559
複数のスレッドの使用に際して潜んでいる落とし穴. 559
並行トランザクション 561
並行トランザクションを使用する際に気を付けるべき落とし穴 562
X/Open XA インターフェース・プログラミングに関する考慮事項 563
アプリケーション連係 567
ネットワークを通じた大量データの処理 . . 567

第18章 区分データベース環境におけるプログラミング上の考慮事項 569
パフォーマンスの向上 569
FOR READ ONLY カーソルの使用 . . . 569
指示 DSS およびローカル・バイパスの使用 569
バッファー化挿入の使用 571
例: 大量データの抽出 (largevol.c) . . . 577
テスト環境の作成. 583
エラー処理考慮事項 583
重大エラー 583
マージされた複数の SQLCA 構造 . . . 584
エラーを戻した区画を識別する 585
デバッグ. 585
ループ状態または延期状態のアプリケーションの診断 585

第19章 DB2 連合システム用のプログラムの作成 587
DB2 連合システムの紹介 587
データ・ソース表および視点へのアクセス 588
ニックネームによる作業 588
データ保全性を保守するための分離レベルの使用 592
データ・タイプ・マッピングの処理. . . . 593
ローカルに定義するデータ・タイプを DB2 が判別する方法. 593
デフォルトのデータ・タイプ・マッピング 593
デフォルト・タイプ・マッピングを上書きして新しいタイプ・マッピングを作成する方法 594
ラージ・オブジェクト (LOB) サポート . . 595
DB2 による LOB の検索方法. 595
アプリケーションが LOB ロケーターを使用する方法 596
LOB での制約事項 597
LOB と非 LOB データ・タイプの間のマッピング. 597
分散要求を使用したデータ・ソースの照会 597
分散要求のコーディング 597
サーバー・オプションを使用して最適化を容易にする 599
データ・ソース関数の呼び出し 600
DB2 がデータ・ソース関数を呼び出せるようにする 600
関数を呼び出す際のオーバーヘッドの削減 601
CREATE FUNCTION MAPPING ステートメントでの関数名の指定 602
関数マッピングの中断 602
パススルーを使用してデータ・ソースを直接照会する. 603
パススルー・セッションでの SQL 処理 603
考慮事項および制約事項 603

第6部 言語に関する考慮事項 . . . 607

第20章 C および C++ でのプログラミング 609
C および C++ でのプログラミングに関する考慮事項. 609
C および C++ での言語制約事項 609
C および C++ での 3 文字表記 609
C++ のタイプ修飾に関する考慮事項 . . 610
C および C++ の入出力ファイル 610
C および C++ のインクルード・ファイル 611
C および C++ のインクルード・ファイル 614
C および C++ での組み込み SQL ステートメント 615
C および C++ でのホスト変数 616
C および C++ でのホスト変数の命名 . . 616
C および C++ でのホスト変数の宣言 . . 618
C および C++ での標識変数 622
C または C++ でのグラフィック・ホスト変数宣言. 622
C または C++ での LOB データ宣言 . . 625
C または C++ における LOB ロケーター宣言 627
C または C++ におけるファイル参照宣言 628

C および C++ でのホスト変数の初期化 629
C マクロ展開 629
C および C++ でのホスト構造サポート 631
C および C++ での標識表 632
C および C++ での Null 終了ストリング 633
C および C++ でのポインター・データ・タイプ 635
C および C++ でのクラス・データ・メンバーのホスト変数としての使用 636
C および C++ での修飾およびメンバー演算子の使用 638
C および C++ でのグラフィック・ホスト変数の処理 638
C および C++ での日本語または中国語 (繁体字) EUC、および UCS-2 に関する考慮事項 643
C および C++ でのサポートされている SQL データ・タイプ 645
C および C++ における FOR BIT DATA 650
C/C++ ストアード・プロシージャー、関数、およびメソッドのタイプ 650
C および C++ における SQLSTATE および SQLCODE 変数 652

第21章 Java でのプログラミング 655

Java のプログラミングに関する考慮事項 . . 655
SQLJ と JDBC の比較 656
他の言語と比較した Java の利点. . . . 656
Java における SQL セキュリティー. . . 656
Java のソースおよび出力ファイル . . . 656
Java クラス・ライブラリー 657
Java パッケージ 657
Java でサポートされている SQL データ・タイプ 658
Java の SQLSTATE および SQLCODE 値 659
Java のトレース機能 660
Java アプリケーションおよびアプレットの作成 660
JDBC プログラミング 663
DB2Appl プログラムの実行方法 664
JDBC アプリケーションの配布 666
JDBC アプレットの配布および実行 . . . 666
JDBC アプレット・サーバーへの接続 . . 667
JDBC 2.0 668
SQLJ プログラミング 671
DB2 SQLJ サポート 671
Java での SQL ステートメントの組み込み 674
Java のホスト変数 680
SQLJ におけるストアード・プロシージャーおよび関数への呼び出し. 680
SQLJ プログラムのコンパイルと実行 . . 681
SQLJ 変換プログラム・オプション . . . 682
Java のストアード・プロシージャーおよび UDF 683
Java クラスを置く場所 684
Java ルーチン・クラスの更新. 685
Java でのストアード・プロシージャーのデバッグ. 685
Java ストアード・プロシージャーおよび UDF 689
JDBC 1.2 で LOB およびグラフィカル・オブジェクトを使用する 693
JDBC および SQLJ の相互運用性 694
セッション共用 694
Java での接続リソース管理 694

第22章 Perl でのプログラミング 697

Perl でのプログラミングに関する考慮事項 697
Perl の制約事項 697
Perl を使ったデータベースへの接続. . . . 698
Perl での取り出し結果 698
Perl のパラメーター・マーカー 699
Perl の SQLSTATE および SQLCODE 変数 699
Perl DB2 アプリケーションの例 700

第23章 COBOL でのプログラミング . . . 701

COBOL でのプログラミングに関する考慮事項 701
COBOL での言語制限 701
COBOL の入力および出力ファイル 701
COBOL のインクルード・ファイル 702
COBOL での SQL ステートメントの組み込み 705
COBOL のホスト変数 707
COBOL でのホスト変数の命名 707
ホスト変数の宣言. 708
COBOL の標識変数 711
COBOL における LOB 宣言 711
COBOL における LOB ロケーター宣言 713
COBOL におけるファイル参照宣言 . . . 713
COBOL でのホスト構造サポート . . . 714

COBOL のインディケーター表 716
COBOL グループ・データ項目での REDEFINES の使用 717
BINARY/COMP-4 COBOL データ・タイプの使用 717
COBOL でサポートされる SQL データ・タイプ 718
COBOL での FOR BIT DATA 721
COBOL での SQLSTATE および SQLCODE 変数 721
COBOL での日本語または中国語 (繁体字) EUC、および UCS-2 に関する考慮事項 . . 721
オブジェクト指向 COBOL 722

第24章 FORTRAN でのプログラミング 723
FORTRAN でのプログラミングに関する考慮事項 723
FORTRAN の言語制限 723
FORTRAN での参照による呼び出し. . . 723
FORTRAN でのデバッグと注釈行 . . . 724
FORTRAN でのプリコンパイルに関する考慮事項. 724
FORTRAN の入力および出力ファイル . . . 724
FORTRAN のインクルード・ファイル . . . 724
FORTRAN でのファイルの組み込み. . . 727
FORTRAN での SQL ステートメントの組み込み 728
FORTRAN のホスト変数 729
FORTRAN でのホスト変数の命名 . . . 729
ホスト変数の宣言. 729
FORTRAN の標識変数 732
FORTRAN における LOB 宣言 732
FORTRAN における LOB ロケーター宣言 733
FORTRAN におけるファイル参照宣言 733
FORTRAN でサポートされている SQL データ・タイプ 734
FORTRAN の SQLSTATE および SQLCODE 変数 736
FORTRAN でのマルチバイト文字セットに関する考慮事項 737
FORTRAN での日本語または中国語 (繁体字) EUC、および UCS-2 に関する考慮事項 . . 737

第25章 REXX でのプログラミング. . . . 739
REXX でのプログラミングに関する考慮事項 739
REXX の言語制限 740
REXX における SQLEXEC、SQLDBS、および SQLDB2 の登録 740
REXX での SQL ステートメントの組み込み 741
REXX のホスト変数. 743
REXX でのホスト変数の命名. 743
REXX でのホスト変数の参照. 743
REXX の標識変数 744
事前定義 REXX 変数 744
REXX の LOB ホスト変数 746
REXX における LOB ロケーター宣言 746
REXX における LOB ファイル参照宣言 747
REXX での LOB ホスト変数のクリア 748
REXX でサポートされている SQL データ・タイプ 749
REXX でのカーソルの使用 750
REXX の実行要件 751
REXX のバインド・ファイル. 752
REXX の API 構文 752
REXX のストアード・プロシージャー. . . 754
REXX におけるストアード・プロシージャーの呼び出し 754
REXX の日本語または中国語 (繁体字) EUC に関する考慮事項. 756

第7部 付録 757

付録A. サポートされる SQL ステートメント. 759

付録B. サンプル・プログラム 765
組み込み SQL なしの DB2 API サンプル 769
DB2 API 組み込み SQL サンプル 773
DB2 API なしの組み込み SQL サンプル . . 775
ユーザー定義関数のサンプル 777
DB2 コール・レベル・インターフェースのサンプル 777
Java サンプル 779
SQL プロシージャーのサンプル 782
ADO、RDO、および MTS サンプル . . . 784
オブジェクトのリンクと埋め込みのサンプル 785
コマンド行プロセッサーのサンプル. . . . 787
ログ管理ユーザー出口サンプル 787

付録C. DB2DARI および DB2GENERAL ストアード・プロシージャーと UDF . . . 789

DB2DARI ストアード・プロシージャー . . 789
クライアント・アプリケーションでの SQLDA の使用. 789
DB2DARI クライアントにおけるホスト変数の使用 790
ストアード・プロシージャーにおける SQLDA の使用. 790
データ構造使用法の要約 791
入出力 SQLDA および SQLCA 構造 . . 792
DB2DARI ストアード・プロシージャーの戻り値 793
DB2GENERAL UDF およびストアード・プロシージャー 794
サポートされる SQL データ・タイプ . . 794
Java ストアード・プロシージャーと UDF のクラス. 796
NOT FENCED ストアード・プロシージャー 801
入力 SQLDA プログラムの例. 802
入力 SQLDA クライアント・アプリケーション例の動作の仕組み 803
C の例: V5SPCLI.SQC 805
入力 SQLDA ストアード・プロシージャー例の動作の仕組み 808
C の例: V5SPSRV.SQC 809

付録D. ホストまたは AS/400 環境でのプログラミング 811

データ定義言語 (DDL) の使用 812
データ操作言語 (DML) の使用 813
数値データ・タイプ 813
混合バイト・データ 813
長フィールド 813
ラージ・オブジェクト (LOB) データ・タイプ 814
ユーザー定義タイプ (UDT) 814
ROWID データ・タイプ 814
64 ビット整数 (BIGINT) データ・タイプ 814
データ制御言語 (DCL) の使用 814
接続と切断 814
プリコンパイル 815
ブロック化 815
パッケージ属性 816
C ヌル終了ストリング 817
スタンドアロンの SQLCODE および SQLSTATE 817
ソート順序の定義. 818
参照保全の管理 818
ロック 818
SQLCODE と SQLSTATE の相違点. . . . 819
システム・カタログの使用. 819
検索割り当て時の数値変換のオーバーフロー 819
分離レベル 819
ストアード・プロシージャー 820
ストアード・プロシージャー・ビルダー 822
NOT ATOMIC 複合 SQL 823
DB2 コネクトでのマルチサイト更新 . . . 824
DB2 コネクトでサポートされている AS/400 サーバー SQL ステートメント 825
DB2 コネクトで拒否される AS/400 サーバー SQL ステートメント. 825

付録E. EBCDIC バイナリー照合のシミュレート 827

付録F. DB2 ライブラリーの使用法. . . . 833

DB2 PDF ファイルおよびハードコピー版資料 833
DB2 情報 833
PDF 資料の印刷 845
印刷資料の注文方法 845
DB2 オンライン文書. 845
オンライン・ヘルプへのアクセス . . . 845
オンライン情報の表示 848
DB2 ウィザードの使用 850
文書サーバーのセットアップ 852
オンライン情報の検索 853

付録G. 特記事項 855

商標 858

索引 861

IBM と連絡をとる 897

製品情報. 897

第1部 DB2 アプリケーション開発の概念

© Copyright IBM Corp. 1993, 2001

第1章 DB2 アプリケーション開発にあたって

本書について 3
本書の対象読者 4
本書の使用法 4
規則 8
関連資料 8

本書について

本書では、DB2 データベースにアクセスするアプリケーション・プログラムを設計し、コーディングする方法について説明します。また、サポートされるホスト言語プログラムにおける構造化照会言語 (SQL) の使用法についての詳細情報が提供されています。ご使用のオペレーティング・システムでの言語サポートについては、 *アプリケーション構築の手引き* を参照してください。本書では、DB2 アプリケーションの作成に役立つ、いくつかの DB2 ユーティリティーの概要についても示します。そのようなユーティリティーの例として、 33ページの『IBM DB2 Universal Database Project Add-In for Microsoft Visual C++』や 283ページの『第9章 IBM DB2 ストアード・プロシージャー・ビルダー』があります。

以下によってデータにアクセスすることができます。

- ホスト言語に組み込まれた SQL ステートメント。 Java Embedded SQL (SQLJ) を含む
- 動的 API。 JDBC (Java Database Connectivity)、Perl DBI、および DB2 コール・レベル・インターフェース (DB2 CLI) を含む

本書では、DB2 CLI 以外のデータにアクセスする方法をすべて解説します。 DB2 CLI については、*コール・レベル・インターフェースの手引きおよび解説書* で説明されています。 JDBC、 SQLJ、および DB2 CLI では、組み込み SQL にはないデータ・アクセス機能を提供しています。それらの機能には、スクロール可能カーソルや、複数の結果セットを戻すストアード・プロシージャーが含まれます。 25ページの『データへのアクセス』を参照すると、使用するデータ・アクセス方式を決定するために役立ちます。

本書の情報を効果的に使用して、DB2 アプリケーション・プログラムの設計、作成、およびテストを行うには、本書とともに *SQL 解説書* も参照する必要があります。 DB2 コール・レベル・インターフェース (CLI) または Open Database Connectivity (ODBC) インターフェースを使用して、アプリケーションから DB2 データベースにアクセスしている場合は、 *コール・レベル・インターフェースの手引きおよび解説書* を参照してください。アプリケーション・プログラムで DB2 管理 API を使用してデータベース・マネージャー管理機能を実行するには、 *管理 API 解説書* を参照してください。

一部をクライアントで実行し、一部をサーバーで実行するアプリケーションも開発することができます。 DB2 のバージョン 7 では、異なるプラットフォーム間での移行性とスケーラビリティーを拡張したストアード・プロシージャーをサポートするようになり

© Copyright IBM Corp. 1993, 2001

ました。ストアード・プロシージャーについては、
201ページの『第7章 ストアード・プロシージャー』で説明されています。

DB2 のオブジェクト・ベースの拡張機能を使用して、ご使用のアプリケーションを従来の DB2 アプリケーションより強力で、柔軟性に富んだ、アクティブなものにすることができます。拡張には、ラージ・オブジェクト (*LOB*)、特殊タイプ、構造型、ユーザー定義関数 (*UDF*)、およびトリガーがあります。こうした DB2 の機能については、以下で説明しています。

- 289ページの『第10章 オブジェクト関連機能の使用』
- 295ページの『第11章 ユーザー定義特殊タイプ』
- 305ページの『第12章 複合オブジェクトの処理: ユーザー定義の構造型』
- 365ページの『第13章 ラージ・オブジェクト (LOB) の使用』
- 389ページの『第14章 ユーザー定義関数 (UDF) およびメソッド』
- 409ページの『第15章 ユーザー定義関数 (UDF) とメソッドの作成』
- 499ページの『第16章 活動状態の DBMS でのトリガーの使用』

本書で使用している DB2 という表現は、UNIX、Linux、OS/2、および Windows 32 ビット・オペレーティング・システムで稼働する DB2 ユニバーサル・データベース製品を意味しています。他のプラットフォーム上の DB2 についての記述では、特定の製品名 (DB2 ユニバーサル・データベース (AS/400 版) など) が使用されています。

本書の対象読者

本書は、SQL、およびサポートされている 1 つ以上のプログラム言語に精通しているプログラマーを対象としています。

本書の使用法

本書は、タスクごとに分類して、以下の部、章、および付録にまとめられています。

- 第1部 DB2 アプリケーション開発の概念では、本書を使用するために必要な情報、および DB2 ユニバーサル・データベース用のアプリケーションの開発に使用できる方法の概要を記載しています。
  - 第1章 DB2 アプリケーション開発にあたってでは、本書の構成、および本書で使用される表記規則について説明しています。
  - 第2章 DB2 アプリケーションのコーディングでは、 DB2 を使用したアプリケーション開発の全行程を紹介しています。さらに、アプリケーションのコーディング前に考慮すべき、重要なアプリケーション設計の問題についても比較検討しています。この章の最後には、アプリケーション開発を始めるためのテスト環境の設定に役立つ情報を記載しています。

- 第2部 アプリケーション内の組み込み SQL では、アプリケーションに静的 SQL および動的 SQL を組み込む方法について説明しています。この情報には、組み込み SQL アプリケーションの作成に役立つユーティリティーの説明が含まれています。
  - ホスト言語における組み込み SQL ステートメントの使用では、C/C++、Java、COBOL などのホスト言語で書かれた SQL を組み込むことによって、 DB2 アプリケーションを作成するプロセスについて説明しています。ここでは、DB2 プリコンパイラー、アプリケーションのコンパイルとリンク、および組み込み SQL ステートメントのデータベースへのバインドの概要も示します。
  - 第4章 静的 SQL プログラムの作成では、静的 SQL ステートメントを使用して、DB2 組み込み SQL アプリケーションをコーディングするための詳細な情報を説明しています。また、静的 SQL を使用する上での詳細なガイドラインおよび考慮事項を記載しています。
  - 第5章 動的 SQL プログラムの作成では、動的 SQL ステートメントを使用して、DB2 組み込み SQL アプリケーションをコーディングするための詳細情報を説明しています。また、動的 SQL を使用する上での詳細なガイドラインおよび考慮事項を記載しています。
  - 第6章 一般的な DB2 アプリケーションの技法では、共通アプリケーションの開発に関連した問題の対処に役立つ DB2 機能について説明しています。たとえば、固有の行 ID を自動的に作成できる機能、式から動的に派生する列を作成できる機能、宣言された一時表を作成および使用できる機能などがあります。
- 第3部 ストアード・プロシージャー では、ストアード・プロシージャーを使用して、クライアント / サーバー環境で稼働するデータベース・アプリケーションのパフォーマンスを向上させる方法について説明しています。
  - 第7章 ストアード・プロシージャーでは、ストアード・プロシージャーを作成する方法と、ホスト言語を使用してストアード・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作成する方法について説明しています。
  - 第8章 SQL プロシージャーの作成では、 CREATE ステートメントと PROCEDURE ステートメントを使用することによって、 SQL 内にストアード・プロシージャーを作成する方法について説明しています。 SQL プロシージャーのプロシージャーとしてのロジックは、 CREATE PROCEDURE プロシージャーの本体にある SQL によってエンコードされます。
  - 第9章 IBM DB2 ストアード・プロシージャー・ビルダーでは、 DB2 用のストアード・プロシージャーを短時間で開発できるようにサポートするグラフィカル・アプリケーション、 IBM DB2 ストアード・プロシージャー・ビルダーについて説明しています。ストアード・プロシージャー・ビルダーは、 SQL ストアード・プロシージャーの作成にも Java ストアード・プロシージャーの作成にも役立ちます。
- 第4部 オブジェクト関連プログラミングでは、 DB2 が提供するオブジェクト関連サポートの使用方法について説明しています。この情報には、ラージ・オブジェクト、

ユーザー定義関数、ユーザー定義特殊タイプ、およびトリガーの紹介、およびそれらの使用方法に関する詳しい指示が含まれています。

– 第10章 オブジェクト関連機能の使用では、DB2 のオブジェクト指向機能について紹介しています。オブジェクト指向コンテキストで、ラージ・オブジェクト、ユーザー定義関数、およびユーザー定義特殊タイプなどの DB2 機能を利用するように従来のアプリケーションを拡張することについての詳細を記載しています。
– 第11章 ユーザー定義特殊タイプでは、アプリケーションで、独自のデータ・タイプを作成して使用する方法について説明しています。特殊タイプを、組み込みデータ・タイプに対するオブジェクト指向の拡張の基礎として使用する方法について説明しています。
– 第12章 複合オブジェクトの処理: ユーザー定義の構造型では、アプリケーションで、構造型を作成して使用する方法について説明しています。オブジェクトを構造型の階層としてモデル化する方法、表内の行または列という形式で表されている構造型のインスタンスにアクセスする方法、および構造型をアプリケーションの内外にバインドする方法について説明しています。
– 第13章 ラージ・オブジェクト (LOB) の使用では、データ・タイプを定義して使用し、最大 2GB までの 2 進またはテキスト・ストリングのデータ・オブジェクトを保管できるようにする方法について説明しています。また、ネットワーク環境で LOB を効果的に使用する方法についても説明しています。
– 第14章 ユーザー定義関数 (UDF) およびメソッドでは、ユーザー固有の拡張機能を SQL に書き込む方法について説明しています。 UDF を使用して、ユーザー固有のデータ・オブジェクトの動作を表現する方法も説明しています。
– 第15章 ユーザー定義関数 (UDF) とメソッドの作成では、 DB2 アプリケーションを拡張するためのユーザー定義関数を作成する方法について説明しています。ユーザー定義関数の定義過程、ユーザー定義関数に関連するプログラミングについての考慮事項、およびこの重要な機能の使用例を記載しています。さらに、この章ではユーザー定義の表関数、 OLE DB 表関数、および OLE オートメーション UDF について説明しています。
– 第16章 活動状態の DBMS でのトリガーの使用では、トリガーを使用して、使用中のすべてのデータベース・アプリケーションに業務上の規則を要約し、それらを施行する方法について説明しています。

• 第5部 DB2 プログラミングに関する考慮事項では、アプリケーション開発に関する特殊な考慮事項を記載しています。
  – 第17章 複合環境におけるプログラミングでは、次のような拡張機能のあるプログラミング・トピックを説明しています。それは各国語サポート、データベースとアプリケーション用の拡張 UNIX® コード (EUC) のコード・ページの処理、作業単位内にある複数のデータベースへのアクセス、およびマルチスレッド・アプリケーションの作成です。

- 第18章 区分データベース環境におけるプログラミング上の考慮事項では、区分データベース環境で稼働するアプリケーションを開発する場合におけるプログラミング上の考慮事項を説明しています。
- 第19章 DB2 連合システム用のプログラムの作成では、連合サーバーを介して、DB2 ファミリーおよび Oracle データ・ソースから透過的にデータにアクセスするアプリケーションの作成方法について説明しています。

• 第6部 言語に関する考慮事項では、 DB2 がサポートするプログラム言語に関する特殊な情報を記載しています。

- 第20章 C および C++ でのプログラミングでは、 C および C++ で作成されたデータベース・アプリケーションに関する、ホスト言語固有の情報を記載しています。
- 第21章 Java でのプログラミングでは、 JDBC または SQLJ を使用して Java で作成されたデータベース・アプリケーションに関する、ホスト言語固有の情報を記載しています。
- 第22章 Perl でのプログラミングでは、 Perl データベース・インターフェース (DBI) モジュール用の DBD::DB2 データベース・ドライバーを使用して Perl で作成されたデータベース・アプリケーションに関する、ホスト言語固有の情報を記載しています。
- 第23章 COBOL でのプログラミングでは、 COBOL で作成されたデータベース・アプリケーションに関する、ホスト言語固有の情報を記載しています。
- 第24章 FORTRAN でのプログラミングでは、 FORTRAN で作成されたデータベース・アプリケーションに関する、ホスト言語固有の情報を記載しています。
- 第25章 REXX でのプログラミングでは、 REXX で作成されたデータベース・アプリケーションに関する、ホスト言語固有の情報を記載しています。

• 付録には、 DB2 アプリケーションの開発の際に参照できる補足的な情報が含まれています。

- 付録A. サポートされる SQL ステートメントでは、 DB2 ユニバーサル・データベースがサポートしている SQL ステートメントをリストしています。
- 付録B. サンプル・プログラムでは、サポートされているさまざまなホスト言語のプログラム例を示し、プログラムの実動状況を説明しています。
- 付録C. DB2DARI および DB2GENERAL ストアード・プロシージャーと UDF では、ストアード・プロシージャー、および DB2 ユニバーサル・データベースの以前のバージョンと互換性のある UDF を作成する方法について記載しています。
- 付録D. ホストまたは AS/400 環境でのプログラミングでは、分散環境でアプリケーション内のホストまたは AS/400 データベース・サーバーにアクセスする場合の、 DB2 コネクトに関するプログラミングの考慮事項を記載しています。
- 付録E. EBCDIC バイナリー照合のシミュレートでは、 EBCDIC またはユーザー定義の照合順序に従って行われる DB2 の文字ストリングの照合方法について説明しています。

– 付録F. DB2 ライブラリーの使用法では、 DB2 ユニバーサル・データベース製品に関する他の情報源を紹介します。

規則

本書では、以下の表記規則を使用しています。

ディレクトリーおよびパス
本書では、ディレクトリーを区切る際に UNIX 規則を使用します。たとえば、sqllib/samples/java のようになります。 / を ¥ に変え、該当するインストール・ドライブとディレクトリーを追加することによって、パスを Windows 32 ビット・オペレーティング・システムおよび OS/2 のパスに変換することができます。

イタリック
以下のいずれかを示します。
- 新しい用語の紹介
- ユーザーが指定する変数名または値
- 別の情報源への言及。たとえば、書籍または CD-ROM
- 一般的な強調

英大文字
以下のいずれかを示します。
- 省略語
- データベース・マネージャーのデータ・タイプ
- SQL ステートメント

例 以下のいずれかを示します。
- コーディング例、およびコードの一部
- システムにより表示されるものと同様の出力例
- 特定のデータ値の例
- システム・メッセージの例
- ファイルおよびディレクトリー名
- 入力するように指示された情報
- Java メソッド名
- 関数名
- API 名

太字 太字のテキストは強調のために使用されます。

関連資料

下記のマニュアルでは、国際的用途や特定の国でのアプリケーションを開発する方法を記載しています。

資料番号	資料名
SE09-8001-03	*National Language Design Guide, Volume 1*
SE09-8002-03	*NLS Reference Manual, Release 4*

第2章 DB2 アプリケーションのコーディング

プログラミングの前提条件 12
DB2 アプリケーションのコーディングの概説 13
変数の宣言および初期化 14
データベース・マネージャーと対話する変数の宣言 14
エラーおよび警告の処理 17
その他の非実行ステートメントの使用 19
データベース・サーバーへの接続 19
トランザクションのコーディング 20
トランザクションの開始 20
トランザクションの終了 21
プログラムの終了 22
トランザクションの暗黙終了 22
サポートされているほとんどのオペレーティング・システムの場合 22
Windows 32 ビット・オペレーティング・システムの場合 23
DB2 コンテキスト API を使用する場合 23
疑似コードによるアプリケーションの枠組み 23
DB2 アプリケーションの設計 24
データへのアクセス 25
組み込み SQL 26
DB2 コール・レベル・インターフェース (DB2 CLI) および Open Database Connectivity (ODBC) 27
JDBC 27
Microsoft 仕様 28
Perl DBI 28
照会製品 28
データ値の制御 28
データ・タイプ 29
固有制約 29
表検査の制約. 29
参照保全制約. 29
CHECK OPTION を使用した視点 . . . 30
アプリケーションのロジックとプログラム変数のタイプ 30
データの関連の制御 30
参照保全制約. 30
トリガー 31
アプリケーションのロジック 31
サーバーにおけるアプリケーションのロジック. 32
ストアード・プロシージャー 32
ユーザー定義関数 32
トリガー 32
IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ 33
IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ の活動化 35
IBM DB2 Universal Database Tools Add-In for Microsoft Visual C++ の活動化 36
サポートされる SQL ステートメント . . . 37
許可についての考慮事項 37
動的 SQL 37
静的 SQL 38
API の使用 39
例 39
組み込み SQL または DB2 CLI プログラムで使用されるデータベース・マネージャー API . 40
テスト環境のセットアップ 40
テスト・データベースの作成 41
テスト表の作成 41
テスト・データの生成. 42
プログラムの実行、テスト、およびデバッグ 44
SQL ステートメントのプロトタイプ化 . . . 44

© Copyright IBM Corp. 1993, 2001

プログラミングの前提条件

この章は、DB2 アプリケーションの論理的な構成部分を示すモデルを提供し、サポートされている DB2 プログラミング API の個々の機能について説明します。 DB2 アプリケーションを初めて開発されるプログラマーの方は、この章全体を注意深くお読みください。

本書で説明されているアプリケーション開発のプロセスは、適切な操作環境がすでに確立されていることを前提としています。つまり、以下が正しくインストールされ、構成されていることを前提としています。

- アプリケーションを開発するための、サポートされているコンパイラーまたはインタープリター
- DB2 ユニバーサル・データベース (ローカルでもリモートでも可)
- DB2 アプリケーション開発クライアント

上記の作業を首尾よく行う方法の詳細について、 *アプリケーション構築の手引き* およびご使用の環境用の*概説およびインストール* 資料を参照してください。

アプリケーションの*開発* は、 DB2 アプリケーション開発クライアントをインストールしたサーバーまたは任意のクライアントで行うことができます。アプリケーションの*実行* は、サーバー、 DB2 ランタイム・クライアント、DB2 アドミニストレーション・クライアントのいずれかで行うことができます。また、クライアントのインストール時に「Java 対応」コンポーネントをインストールした場合は、それらのクライアントのいずれかで Java JDBC プログラムを開発することもできます。これは、それらのクライアント上で DB2 アプリケーションを実行できることを意味します。しかし、クライアントとともに DB2 アプリケーション開発クライアントもインストールしていなければ、クライアント上では JDBC アプリケーションしか開発できません。

DB2 は、プリコンパイラーを介して、C、C++、Java (SQLJ)、COBOL、および FORTRAN プログラミング言語をサポートします。さらに、動的に解釈される言語である Perl、 REXX、および Java (JDBC) もサポートしています。 DB2 で提供されるプリコンパイラー、およびご使用のプラットフォームでサポートされる言語の詳細については、ご使用のオペレーティング・システム用の*アプリケーション構築の手引き* を参照してください。

注: FORTRAN および REXX のサポートは DB2 バージョン 5 において確立され、今後 FORTRAN または REXX のサポートを拡張する予定はありません。

DB2 には、サンプル・プログラムを実行する際に必要なサンプル・データベースが備えられています。サンプル・データベースとその内容については、 *SQL 解説書* を参照してください。

DB2 アプリケーションのコーディングの概説

DB2 アプリケーション・プログラムは、以下の部分で構成されています。

1. 変数の宣言および初期化
2. データベースへの接続
3. 1 つ以上のトランザクション の実行
4. データベースからの切断
5. プログラムの終了

トランザクション とはデータベース操作の集合で、データベースにコミットされる前に正常に終了しなければなりません。組み込み SQL を使用した場合、トランザクションは暗黙に開始され、アプリケーションが COMMIT または ROLLBACK ステートメントのいずれかを実行すると終了します。トランザクションの例としては、カスタマーの預金の入力、および差し引き残高の更新などがあります。

一定の SQL ステートメントは、ホスト言語から組み込み SQL への変換を処理するために、プログラムの始めと終わりに使用しなければなりません。

プログラムの先頭には、以下のものが必要です。

- データベース・マネージャーがホスト・プログラムと対話するのに使用するすべての変数とデータ構造の宣言。
- SQL 連絡域 (SQLCA) を設定してエラーを処理する SQL ステートメント。

  Java で作成された DB2 アプリケーションは `SQLException` をスローします。これは、SQLCA を使用するのではなく、`catch` ブロックで処理してください。

どのプログラムの本体にも、データにアクセスして管理する SQL ステートメントが含まれています。このようなステートメントが集まってトランザクションを構成します。トランザクションには以下のステートメントが含まれていなければなりません。

- CONNECT ステートメント (データベース・サーバーへの接続を確立する)
- 以下の 1 つ以上:
  - データ操作ステートメント (たとえば SELECT ステートメントなど)
  - データ定義ステートメント (たとえば CREATE ステートメントなど)
  - データ制御ステートメント (たとえば、GRANT ステートメントなど)
- COMMIT または ROLLBACK ステートメント (トランザクションを終了する)

アプリケーション・プログラムの終わりには、通常、次のような働きを持つ SQL ステートメントを置きます。

- プログラムからデータベース・サーバーへの接続を解除する。
- リソースを終結処理する。

変数の宣言および初期化

DB2 アプリケーションをコーディングするには、まず以下を宣言する必要があります。

- データベース・マネージャーと対話する変数
- SQLCA (該当する場合)

データベース・マネージャーと対話する変数の宣言

データベース・マネージャーと対話する変数はすべて、 SQL 宣言セクションで宣言しなければなりません。 SQL 宣言セクションは、以下の構造でコーディングする必要があります。

1. SQL ステートメント BEGIN DECLARE SECTION
2. 1 つ以上の変数宣言のグループ
3. SQL ステートメント END DECLARE SECTION

SQL 宣言セクションで宣言されたホスト・プログラム変数はホスト変数 と呼ばれます。ホスト変数は SQL ステートメント内の*ホスト変数* 参照で使用できます。*ホスト変数* は、 *SQL 解説書* の構文図で使用される 1 つのタグです。 1 つのプログラムに複数の SQL 宣言セクションが含まれる場合もあります。

各ホスト変数の属性は、その変数が SQL ステートメントにおいてどのように使用されるかにより異なります。たとえば、DB2 表からデータを受け取る変数、または DB2 表にデータを保管する変数には、アクセスする列と互換性のあるデータ・タイプ属性と長さ属性がなければなりません。各変数のデータ・タイプを決定するには、 84ページの『データ・タイプ』で説明する DB2 データ・タイプをよく理解していなければなりません。

SQL オブジェクトを表す変数の宣言: DB2 バージョン 7 では、表、別名、視点、および相関の名前の長さは、最高で 128 バイトです。列名の長さは、最高で 30 バイトです。 DB2 バージョン 7 では、スキーマ名の長さは、最高で 30 バイトです。今後のリリースでは、列名および SQL オブジェクトの他の ID の長さが最高 128 バイトまで増える可能性があります。 SQL オブジェクトを表す変数を、128 バイトよりも短い長さで宣言する場合、 SQL オブジェクト ID の長さが今後増えると、アプリケーションの安定度に影響が及ぶ可能性があります。たとえば、 C++ アプリケーションでスキーマ名を保持するために変数 char[9]*schema_name* を宣言する場合、アプリケーションは、DB2 バージョン 6 で許可されているスキーマ名 (最大長が 8 バイト) に対しては正しく機能します。

```
char[9] schema_name; /* holds null-delimited schema name of up to 8 bytes;
 works for DB2 Version 6, but may truncate schema names in future releases */
```

しかし、データベースを DB2 バージョン 7 に移行し、そのバージョンがスキーマ名の最大長として 30 バイトを受け入れている場合、アプリケーションは、スキーマ名 LONGSCHEMA1 と LONGSCHEMA2 を区別することができません。データベース・マネージャーは LONGSCHE の限度である 8 バイトでスキーマ名を切り捨て、アプリケーション内の

ステートメントのうち、スキーマ名を区別しなければならないものはすべて失敗します。アプリケーションを長期間使用するためには、次のように、スキーマ名変数を 128 バイトの長さで宣言してください。

```
char[129] schema_name; /* holds null-delimited schema name of up to 128 bytes
                          good for DB2 Version 7 and beyond */
```

アプリケーションの操作を将来的にさらに向上させるには、アプリケーションで SQL オブジェクト名を表す変数を 128 バイトの長さで宣言することを考慮してください。互換性を向上させることの利点と、変数名を長くすることで必要になるシステム・リソースとを、比較検討する必要があります。

このコーディング練習の使用を簡単なものとし、 C/C++ アプリケーション・コードをわかりやすくするために、 C マクロ展開を使用して、SQL オブジェクト ID の長さを宣言することを考慮してください。インクルード・ファイル sql.h は、SQL_MAX_IDENT が 128 になるように宣言しているため、 SQL_MAX_IDENT マクロを使用すれば、SQL オブジェクト ID を簡単に宣言することができます。以下に例を示します。

```
#include <sql.h>
    char[SQL_MAX_IDENT+1] schema_name;
    char[SQL_MAX_IDENT+1] table_name;
    char[SQL_MAX_IDENT+1] employee_column;
    char[SQL_MAX_IDENT+1] manager_column;
```

C マクロ展開の詳細については、629ページの『C マクロ展開』を参照してください。

ホスト変数と SQL ステートメントの関連付け: ホスト変数を使用して、データベース・マネージャーからデータを受け取ったり、ホスト・プログラムからデータベース・マネージャーにデータを転送したりすることができます。データベース・マネージャーからデータを受け取るホスト変数は*出力ホスト変数*、ホスト・プログラムからデータベース・マネージャーにデータを転送するホスト変数は*入力ホスト変数* です。

次の SELECT INTO ステートメントを例にとります。

```
SELECT HIREDATE, EDLEVEL
  INTO :hdate, :lvl
             FROM EMPLOYEE
  WHERE EMPNO = :idno
```

このステートメントには、hdate と lvl の 2 つの出力ホスト変数と、入力ホスト変数 idno が含まれています。データベース・マネージャーは、ホスト変数 idno に保管されているデータを使用して、 EMPLOYEE 表から検索する行の EMPNO を決めます。検索基準を満たす行が見つかると、 hdate と lvl はそれぞれ HIREDATE 列と EDLEVEL 列に保管されたデータを受け取ります。上記のステートメントは、EMPLOYEE 表の列を使用したホスト・プログラムとデータベース・マネージャー間の対話の例です。

表の各列には、CREATE TABLE ステートメントで定義されたデータ・タイプが割り当てられます。このデータ・タイプをホスト言語データ・タイプに関連付ける必要があります。ホスト言語データ・タイプは、本書の各言語に関する章のサポートされる *SQL データ・タイプ* の節で定義されています。たとえば、INTEGER データ・タイプは 32 ビットの符号付き整数です。これは、各ホスト言語による以下のようなデータ記述項目に対応しています。

C/C++: `sqlint32 variable_name;`

Java: `int variable_name;`

COBOL:
```
01  variable-name  PICTURE S9(9) COMPUTATIONAL-5.
```

FORTRAN:
```
INTEGER*4 variable_name
```

サポートされる SQL データ・タイプのリストと、それに対応するホスト言語データ・タイプについては、以下を参照してください。

- C/C++ の場合、645ページの『C および C++ でのサポートされている SQL データ・タイプ』
- Java の場合、658ページの『Java でサポートされている SQL データ・タイプ』
- COBOL の場合、718ページの『COBOL でサポートされる SQL データ・タイプ』
- FORTRAN の場合、734ページの『FORTRAN でサポートされている SQL データ・タイプ』
- REXX の場合、749ページの『REXX でサポートされている SQL データ・タイプ』

ある列に使用するホスト変数の定義方法を正確に決定するには、その列の SQL データ・タイプを調べなければなりません。データベース内に作成された表すべてに関する情報を含んだ視点の集合である、システム・カタログを照会してこれを調べてください。 *SQL 解説書* では、システム・カタログについて説明しています。

データ・タイプを決定したら、それぞれのホスト言語の章にある変換表を参照して、適切な宣言をコーディングできます。宣言生成プログラム・ユーティリティー (db2dclgn) を使用して、データベース内の指定された表に、適切な宣言を生成することもできます。 db2dclgn の詳細については、 80ページの『宣言生成プログラム - db2dclgn』およびコマンド*解説書* を参照してください。

80ページの表4 は、サポートされるホスト言語による宣言の例を示しています。 REXX アプリケーションは、 LOB ロケーターおよびファイル参照変数以外のホスト変数を宣言する必要はないことに注意してください。他のホスト変数のデータ・タイプとサイズは、変数の内容に基づいて実行時に決定されます。

表4 には、 BEGIN DECLARE SECTION および END DECLARE SECTION ステートメントも示されています。 SQL ステートメントの区切り文字と各言語の区切り文字の

違いに気を付けてください。これらのステートメントの配置、連結、および区切りの正しい規則については、本書の各言語別の章を参照してください。

エラーおよび警告の処理

SQL 連絡域 (SQLCA) については、この章で後ほど詳しく説明します。この節では概要を説明します。 SQLCA を宣言するには、プログラムに INCLUDE SQLCA ステートメントをコーディングします。

各言語でのステートメントを以下に示します。 C または C++ アプリケーションの場合、

```
EXEC SQL INCLUDE SQLCA;
```

Java アプリケーションの場合、Java では明示的に SQLCA を使用することはしません。その代わりに、SQLException インスタンス・メソッドを使用して、 SQLSTATE および SQLCODE 値を入手します。詳細については、659ページの『Java の SQLSTATE および SQLCODE 値』を参照してください。

COBOL アプリケーションの場合、

```
EXEC SQL INCLUDE SQLCA END-EXEC.
```

FORTRAN アプリケーションの場合、

```
EXEC SQL INCLUDE SQLCA
```

プログラムをプリプロセスする際に、データベース・マネージャーは INCLUDE SQLCA ステートメントの代わりにホスト言語変数宣言を挿入します。システムは、警告標識、エラー・コード、および診断情報の変数を使用してプログラムと連絡します。

各 SQL ステートメントを実行すると、システムは SQLCODE および SQLSTATE の両方の戻りコードを戻します。 SQLCODE はステートメントの実行を要約した整数値で、 SQLSTATE は IBM のリレーショナル・データベース製品に共通のエラー・コードを示す文字フィールドです。 SQLSTATE は ISO/ANS SQL92 標準、および FIPS 127-2 標準にも準拠しています。

注: FIPS 127-2 とは、*Federal Information Processing Standards Publication 127-2 for Database Language SQL* のことです。 ISO/ANS SQL92 とは、*American National Standard Database Language SQL X3.135-1992* および *International Standard ISO/IEC 9075:1992, Database Language SQL* を指します。

0 未満の SQLCODE は、エラーが発生してステートメントが処理されなかったことを示していることに注意してください。 1 以上の SQLCODE は、警告が出されたものの、ステートメントの処理は継続していることを示します。 SQLCODE と SQLSTATE のエラー状態のリストについては、 メッセージ*解説書* を参照してください。

各 SQL ステートメントを実行した後のエラー・チェックをシステムで制御したい場合には、 WHENEVER ステートメントを使用します。

注: Java Embedded SQL (SQLJ) アプリケーションは、WHENEVER ステートメントを使用できません。 659ページの『Java の SQLSTATE および SQLCODE 値』で説明されている SQLException メソッドを使用して、 SQL ステートメントが戻したエラーを処理してください。

次の WHENEVER ステートメントは、負の SQLCODE が戻された場合にシステムが行う動作を指定します。

```
WHENEVER SQLERROR GO TO errchk
```

つまり、SQL エラー・コードが発生すると、プログラムの制御が errchk などのラベルの後に続くコードに移ります。このコードには、SQLCA のエラー標識を分析するロジックを組み込んでおきます。 ERRCHK 定義に従って、次の順次プログラム命令を実行したり、特殊関数を実行したり、ほとんどの状況で現行のトランザクション をロールバックしてプログラムを終了するなどの処置がとられます。トランザクションの詳細については 20ページの『トランザクションのコーディング』、アプリケーション・プログラムのエラー・チェックの詳細については 122ページの『診断処理と SQLCA 構造』を参照してください。

WHENEVER SQLERROR ステートメントを使用するときには十分注意してください。アプリケーションのエラー処理コードに SQL ステートメントがあり、それらで元のエラーを処理している間にエラーが発生する場合、アプリケーションで無限のループが実行される可能性があります。この状態での障害追及は困難です。 WHENEVER SQLERROR の宛先を示す最初のステートメントは、必ず WHENEVER SQLERROR CONTINUE にしてください。このステートメントはエラー・ハンドラーをリセットします。このステートメントの後であれば、安心して SQL ステートメントを使用できます。

C または C++ 言語で作成された DB2 アプリケーションの場合、アプリケーションが複数のソース・ファイルで構成されているなら、 SQLCA の多重定義を回避するため、EXEC SQL INCLUDE SQLCA ステートメントを組み込むのはその中の 1 つのファイルだけにしてください。それ以外のソース・ファイルには、次の行を組み込みます。

```
#include "sqlca.h"
extern struct sqlca sqlca;
```

ご使用のアプリケーションが、 ISO/ANS SQL92 または FIPS 127-2 標準に準拠する必要がある場合は、上記のステートメントや INCLUDE SQLCA ステートメントを使用しないでください。 ISO/ANS SQL92 および FIPS 127-2 標準の詳細については、 17ページの『FIPS 127-2 および ISO/ANS SQL92 の定義』を参照してください。上記のステートメントのコーディングに代わる方法については、以下を参照してください。

- C または C++ アプリケーションの場合は、652ページの『C および C++ における SQLSTATE および SQLCODE 変数』
- COBOL アプリケーションの場合は、721ページの『COBOL での SQLSTATE および SQLCODE 変数』
- FORTRAN アプリケーションの場合は、736ページの『FORTRAN の SQLSTATE および SQLCODE 変数』

その他の非実行ステートメントの使用

一般に、その他の*非実行* SQL ステートメントもプログラムのセットアップ部分に含まれます。 *SQL 解説書* および本書のこの後の章では、非実行ステートメントについて説明しています。非実行ステートメントの例を以下に示します。

- INCLUDE `text-file-name`
- INCLUDE SQLDA
- DECLARE CURSOR

データベース・サーバーへの接続

実行可能な SQL ステートメントを実行するには、その前に宛先データベース・サーバーへの接続を確立しておかなければなりません。この接続により、プログラムを実行しているユーザーの許可 ID、およびプログラムが稼働しているデータベース・サーバーの名前を識別します。一般に、アプリケーション・プロセスが一度に接続できるデータベース・サーバーは 1 つだけです。このサーバーを*現行サーバー* といいます。しかし、マルチサイト更新環境内であれば、複数のデータベース・サーバーに接続することができます。この場合、ただ 1 つのサーバーだけが現行サーバーになります。マルチサイト更新の詳細については、549ページの『マルチサイト更新』を参照してください。

プログラムは、以下のいずれかの方法でデータベース・サーバーと接続を確立することができます。

- CONNECT ステートメントを使用して明示的に接続する
- デフォルトのデータベース・サーバーに接続して暗黙的に接続する
- Java アプリケーションの場合は、接続インスタンスを使用する

接続状況および CONNECT ステートメントの使用方法については、 *SQL 解説書* を参照してください。初期化時に、アプリケーション・リクエスターがデフォルトのデータベース・サーバーを確立します。暗黙接続が使用可能になっている場合は、初期化後に開始されるアプリケーション・プロセスにより、デフォルトのデータベース・サーバーへの接続が暗黙的に行われます。アプリケーション・プログラムが最初に実行する SQL ステートメントを CONNECT ステートメントとして使用することは、良い方法です。そうすれば、デフォルトのデータベースに対して不用意に SQL ステートメントを実行しないようにすることができます。

接続が確立されてから、プログラムから次のような SQL ステートメントを発行できます。

- データを操作する
- データベース・オブジェクトを定義し、保守する
- ユーザー権限を授与したり、データベースへの変更をコミットするなどの、制御操作を開始する

接続は、CONNECT RESET、CONNECT TO、または DISCONNECT ステートメントが発行されるまで設定されたままとなります。マルチサイト更新環境では、接続は DB2 RELEASE およびそれに続いて DB2 COMMIT が発行されるまで続きます。マルチサイト更新 (549ページの『マルチサイト更新』参照) を使用している場合、 CONNECT TO ステートメントでは接続は終了しません。

トランザクションのコーディング

トランザクションは一連の SQL ステートメント (途中にホスト言語コードが割り込む) からなり、データベース・マネージャーはこれらのステートメントを一まとまりとして扱います。トランザクションの代わりに、同じ意味として*作業単位* という用語がよく使われます。

トランザクション・レベルでのデータの一貫性を確保するために、システムは、トランザクション内の操作がすべて *完了するか*、*まったく行われなかった* かのいずれかの状態を保ちます。たとえば、ある口座から現金を引き出し、それを別の口座に入れるとします。この 2 つの更新が単一のトランザクションで行われる場合、これらの処理中にシステム障害が発生すると、システムの再起動時にそのデータはトランザクションが開始される前の状態に自動的に復元されます。プログラム・エラーが発生した場合は、エラーになったステートメントによる変更がすべて元の状態に復元されます。トランザクション内でエラーになったステートメントの実行前に行われた作業は、特にロールバックを行わなければ復元されません。

単一のアプリケーション・プログラム内の 1 つまたは複数のトランザクションをコード化することができます。さらに単一のトランザクション内から複数のデータベースにアクセスすることが可能です。複数のデータベースにアクセスするトランザクションは、マルチサイト更新と呼ばれます。これらのトピックについては、 549ページの『リモート作業単位』および 549ページの『マルチサイト更新』を参照してください。

トランザクションの開始

トランザクションは先頭の*実行可能* SQL ステートメントにより暗黙的に開始され、COMMIT ステートメントか ROLLBACK ステートメント、またはプログラムの末尾で終了します。

一方、以下の 6 つのステートメントは実行可能ステートメントではないため、トランザクションを開始**しません**。

```
BEGIN DECLARE SECTION          INCLUDE SQLCA
END DECLARE SECTION            INCLUDE SQLDA
DECLARE CURSOR                 WHENEVER
```

実行可能 SQL ステートメントは常にトランザクション内に現われます。トランザクションが終了した後でプログラムに実行可能 SQL ステートメントが含まれている場合は、新しいトランザクションを自動的に開始します。

トランザクションの終了

トランザクションを終了するには、次のいずれかを行ってください。

- COMMIT ステートメントを使用して、変更を保管する
- ROLLBACK ステートメントを使用して、それらの変更が保管されないようにする

COMMIT ステートメントの使用: このステートメントは、現行トランザクションを終了します。現行トランザクションで行われたデータベースの変更が、他の処理でも認識されるようになります。

アプリケーションの要件に応じて、できるだけ早く変更がコミットされるようにしてください。特に、端末からの入力を待機している間、コミットされていない変更を保留することがないようなプログラムを作成してください。それが原因で、データベース・リソースが長い間保留にされないようにするためです。これらのリソースを保留してしまうと、同じくこれを必要とする他のアプリケーションが実行できなくなります。

COMMIT ステートメントは、ホスト変数の内容には影響を与えません。

アプリケーション・プログラムが、終了処理の前にすべてのトランザクションを明示的に終わらせるようにしてください。トランザクションを明示的に終わらせないと、 DB2 はトランザクションが保留されていた間のすべての変更を、プログラムが正常に終了した時点で自動的にコミットします。ただし、Windows 32 ビット・オペレーティング・システムではコミットは行われません。 DB2 は以下の条件のときに、変更をロールバックします。

- ログが満杯になったとき。
- データベース・マネージャーの処理を終了させるようなその他のシステム条件。

Windows 32 ビット・オペレーティング・システムの場合は、トランザクションを明示的にコミットしないなら、データベース・マネージャーがその変更を必ずロールバックするようにします。

プログラムの終了については、22ページの『プログラムの終了』および 122ページの『診断処理と SQLCA 構造』を参照してください。

ROLLBACK ステートメントの使用: このステートメントは現行トランザクションを終了して、トランザクションが開始される前の状態にデータを復元します。

ROLLBACK ステートメントは、ホスト変数の内容には影響を与えません。

エラーまたは警告があったために実行されることになったルーチン内で ROLLBACK ステートメントと SQL WHENEVER ステートメントを使用している場合は、

ROLLBACK ステートメントの前に WHENEVER SQLERROR CONTINUE および WHENEVER SQLWARNING CONTINUE を指定します。これは、エラーまたは警告により ROLLBACK が失敗した場合にプログラムがループするのを防ぎます。

重大エラーの場合には、ROLLBACK ステートメントでは発行できないというメッセージが表示されます。クライアントとサーバー・アプリケーション間の通信の損失や、データベースの破損などの重大エラーが発生した場合は、 ROLLBACK ステートメントを発行しないでください。重大エラーが発生した後で発行できるステートメントは CONNECT ステートメントだけです。

プログラムの終了

プログラムを正しく終了させるには、以下のステップに従ってください。

1. COMMIT ステートメントまたは ROLLBACK ステートメントを明示的に発行して、現行トランザクション (処理中のものがある場合) を終了する。
2. CONNECT RESET ステートメントを使用して、データベース・サーバーへの接続を解放する。
3. プログラムが使用したリソースの終結処置を行う。たとえば、使用した一時記憶域またはデータ構造をすべて解放します。

注: プログラム終了時に、現行トランザクションがまだ活動中であると、 DB2 は暗黙にそのトランザクションを終了させます。トランザクションを暗黙に終了させる際の DB2 の動作はプラットフォーム固有なので、プログラム終了前に COMMIT か ROLLBACK ステートメントを発行して、すべてのトランザクションを明示的に終了するようにしてください。 DB2 が暗黙的にトランザクションを終了させる方法の詳細については、 『トランザクションの暗黙終了』を参照してください。

トランザクションの暗黙終了

現行トランザクションを終了せずにプログラムを終わらせた場合、 DB2 は暗黙に現行トランザクションを終了します (プログラムに適した終了方法の詳細については、 『プログラムの終了』を参照してください)。 DB2 はアプリケーション終了時に、 COMMIT または ROLLBACK ステートメントのどちらかを発行することにより、現行トランザクションを暗黙に終了します。 DB2 が COMMIT と ROLLBACK のどちらを発行するかは、以下の要因によって決まります。

- アプリケーションが正常に終了したかどうか
- DB2 サーバーが稼働しているプラットフォーム
- アプリケーションがコンテキスト API (557ページの『マルチスレッドのデータベースのアクセス』を参照) を使用しているかどうか

サポートされているほとんどのオペレーティング・システムの場合

正常に終了した場合、DB2 はトランザクションを暗黙にコミットしますが、異常終了の場合は、トランザクションを暗黙にロールバックします。プログラムが異常終了と見なすものでも、データベース・マネージャーは異常終了と見なさない場合があることに注

意してください。たとえば、アプリケーションが予期しないエラーを察知したときに、アプリケーションを即座に終了するように、 exit(-16) をコーディングすることができます。データベース・マネージャーは、この状況を正常終了と見なして、トランザクションをコミットします。一方、データベース・マネージャーは、例外やセグメント化違反などを異常終了とみなします。

Windows 32 ビット・オペレーティング・システムの場合

COMMIT ステートメントを使用して、トランザクションを明示的にコミットしない限り、アプリケーションの終了が正常か異常かに関係なく、DB2 は必ずトランザクションをロールバックします。

DB2 コンテキスト API を使用する場合

アプリケーションは DB2 API を用いてセットアップし、 557ページの『マルチスレッドのデータベースのアクセス』で説明されているようにスレッド間でアプリケーション・コンテキストを渡すことができます。アプリケーションで DB2 API を使用している場合には、アプリケーションの終了が正常か異常かに関係なく、 DB2 は暗黙にトランザクションをロールバックします。 COMMIT ステートメントを使用して、トランザクションを明示的にコミットしない限り、トランザクションはロールバックされます。

疑似コードによるアプリケーションの枠組み

プログラムのコーディングのための疑似コードによる枠組みに、 DB2 アプリケーション・プログラムの一般的な枠組みを疑似コード形式で要約します。もちろん、実際に使用するプログラムに合うようにこの枠組みを調整することが必要です。

```
プログラム開始
EXEC SQL BEGIN DECLARE SECTION                           |
  DECLARE USERID FIXED CHARACTER (8)                     |
  DECLARE PW FIXED CHARACTER (8)                         |
                                                         | アプリケーションの
  (その他のホスト変数宣言)                               | セットアップ
                                                         |
EXEC SQL END DECLARE SECTION                             |
EXEC SQL INCLUDE SQLCA                                   |
EXEC SQL WHENEVER SQLERROR GOTO ERRCHK                   |

  (プログラムのロジック)

EXEC SQL CONNECT TO database A USER :userid USING :pw    |
EXEC SQL SELECT ...                                      |
EXEC SQL INSERT ...                                      | 最初の
  (SQL ステートメントが続く)                             | 作業単位
EXEC SQL COMMIT                                          |

  (プログラムのロジックが続く)

EXEC SQL CONNECT TO database B USER :userid USING :pw    |
EXEC SQL SELECT ...                                      |
EXEC SQL DELETE ...                                      | 2 番目の
  (SQL ステートメントが続く)                             | 作業単位
EXEC SQL COMMIT                                          |

  (プログラムのロジックが続く)
```

```
EXEC SQL CONNECT TO database A
EXEC SQL SELECT ...
EXEC SQL DELETE ...                                    3 番目の
  (SQL ステートメントが続く)                            作業単位
EXEC SQL COMMIT

  (プログラムのロジックが続く)

EXEC SQL CONNECT RESET
ERRCHK
                                                       アプリケーションの
  (SQLCA のエラー情報を検査する)                        終結処置

プログラム終了
```

DB2 アプリケーションの設計

DB2 には、アプリケーションの従来の機能を補足または拡張するためのさまざまなアプリケーション開発機能が備えられています。アプリケーション設計者が決定すべき事柄は、最も基礎的な設計です。つまり、*アプリケーションの設計にどの DB2 機能を使用すればよいのか?* ということです。適切な選択を行うためには、アプリケーションのデータベース設計と目標環境の両方を考慮しなければなりません。たとえば、アプリケーションにロジックを組み込む代わりに、業務上の規則の一部をデータベース設計に組み込むことができます。

使用する機能とその範囲は、さまざまに変更できます。この節は、設計に大きな影響を及ぼす使用可能な機能についての概説であり、特定の目的に対してある機能が他の機能よりもふさわしい理由を示しています。この節に記載された機能の詳細については、より詳しい説明を載せた参照用資料が提供されています。

考慮すべき機能には、次のものがあります。

- 以下の機能を利用したデータへのアクセス:
  - 組み込み SQL。Java Embedded SQL (SQLJ) を含む
  - DB2 コール・レベル・インターフェース (DB2 CLI)、 Open Database Connectivity (ODBC)、および Java Database Connectivity (JDBC)
  - Microsoft 仕様
  - Perl DBI
  - 照会製品
- 以下の機能を利用したデータ値の制御:
  - データ・タイプ (組み込みまたはユーザー定義)
  - 表検査の制約
  - 参照保全制約
  - CHECK OPTION を使用した視点
  - アプリケーションのロジックと変数のタイプ
- 以下の機能を利用したデータ値間の関連の制御:
  - 参照保全制約
  - トリガー

– アプリケーションのロジック

- 以下の機能を利用した、サーバーでのプログラムの実行:
  – ストアード・プロシージャー
  – ユーザー定義関数
  – トリガー

前述のリストには、トリガーなどのように、複数の項目で使用されている機能があります。これは、複数の設計基準に対応するという、これらの機能の柔軟性を反映しています。

最も重要かつ基礎的な決定は、アプリケーションに関連したデータ規則を適用するためのロジックを、データベースに移動させるかどうかです。

データに着目した、ロジックをアプリケーションからデータベースに移すことの重要な利点は、アプリケーションのデータからの独立性が一層高まるということです。データに関連するロジックは、1 箇所 (データベース) に集められます。つまり、データまたはデータ・ロジックの変更を一度に行うことができ、またそれがただちに**すべての**アプリケーションに反映されます。

この後者の利点は非常に強力なものですが、データベース内に置かれるデータ・ロジックがデータのユーザー**すべてに**同じ影響を与えるということに気を付けてください。データに加えられる規則や制約が、そのデータの全ユーザーやアプリケーションのユーザーにも当てはまるかどうかを考慮しなければなりません。

アプリケーションの要件は、データ規則をデータベースで適用するか、アプリケーションで適用するかにも影響を与えます。たとえば、特定の順序でデータが入力されたときに妥当性検査エラーを処理しなければならないことがあります。一般に、これらのタイプのデータの妥当性検査は、アプリケーション・コードで行う必要があります。

アプリケーションを使用するコンピューティング環境も考慮してください。クライアント側のマシンでロジックを実行する場合と、ストアード・プロシージャー、UDF、またはその両方を組み合わせて使用して、通常のより強力なデータベース・サーバー側のマシンでロジックを実行する場合の差について考慮する必要があります。

場合によっては、アプリケーション (アプリケーション固有の理由のため) とデータベース (アプリケーション外の他の対話的な用途のため) の両方を考慮に入れて設計しなければならないということもあります。

データへのアクセス

リレーショナル・データベースでは、データへのアクセスには SQL を使用しなければなりませんが、 SQL をアプリケーションに組み込む方法はユーザーが任意で選択できます。以下のインターフェースおよびサポートされている言語の中から選択してください。

組み込み SQL
C/C++、COBOL、FORTRAN、Java (SQLJ)、REXX

DB2 CLI および ODBC
C/C++、Java (JDBC)

Microsoft 仕様 (ADO、RDO、OLE DB を含む)
Visual Basic、Visual C++

Perl DBI
Perl

照会製品
Lotus Approach、IBM 照会管理機能

組み込み SQL

組み込み SQL には、静的 SQL と動的 SQL のいずれも使用でき、この 2 つのタイプを組み合わせて使用することもできるという利点があります。アプリケーションの使用時に SQL ステートメントの内容や形式が*柔軟性に乏しい* と思われる場合には、アプリケーションで組み込み SQL を使用することを考えてみてください。静的 SQL を使用すると、アプリケーションの実行者は、アプリケーションをデータベースにバインドしたユーザーの特権を一時的に継承することができます。 DYNAMICRULES BIND オプションを指定してアプリケーションをバインドしていない場合は、動的 SQL がアプリケーションの実行者の特権を使用します。一般的には、組み込み動的 SQL を使用してください。これによって実行可能ステートメントは実行時に決定されます。組み込み動的 SQL を使用すると、より安全なアプリケーション・プログラムを作成して、さらに多様な入力を扱うことができます。

注: Java Embedded SQL (SQLJ) アプリケーションは、静的 SQL ステートメントしか組み込むことができません。ただし、JDBC を使用して、SQLJ アプリケーションで動的 SQL 呼び出しを行うことができます。

SQL ステートメントをホスト言語コマンドに変換するには、プログラム言語コンパイラーを使用する前に、組み込み SQL アプリケーションをプリコンパイルする必要があります。さらに、アプリケーションを実行するには、アプリケーション内の SQL をデータベースにバインドする必要もあります。

組み込み SQL の追加情報については、67ページの『第4章 静的 SQL プログラムの作成』を参照してください。

REXX に関する考慮事項: REXX アプリケーションは API を使用することにより、データベース・マネージャー API および SQL により提供される機能の大部分を使用できるようになります。コンパイル言語で作成されたアプリケーションとは異なり、REXX アプリケーションはプリコンパイルされません。その代わりに、動的 SQL ハンドラーがすべての SQL ステートメントを処理します。 REXX と呼び出し可能な API を組み合わせることにより、データベース・マネージャー機能の大部分にアクセスでき

ます。組み込み SQL を使用する API の中には REXX が直接サポートしていないものもありますが、 DB2 コマンド行プロセッサーを使用すれば、 REXX アプリケーションからこれらの API にアクセスできます。

REXX はインタープリター言語なので、コンパイル済みホスト言語に比べてアプリケーションのプロトタイプの開発やデバッグが容易です。 REXX でコーディングされた DB2 アプリケーションは、コンパイル言語を使用した DB2 アプリケーションほどの性能は持ちませんが、プリコンパイル、コンパイル、リンクを行わず、またはさらに別のソフトウェアを使用せずに DB2 アプリケーションを作成する機能を提供できることに注目してください。

REXX を使用した DB2 アプリケーションのコーディングと作成の詳細については、739ページの『第25章 REXX でのプログラミング』を参照してください。

DB2 コール・レベル・インターフェース (DB2 CLI) および Open Database Connectivity (ODBC)

DB2 コール・レベル・インターフェース (DB2 CLI) は、データベース・サーバーの DB2 ファミリーに対する、 IBM の呼び出し可能 SQL インターフェースです。また、これはリレーショナル・データベースのアクセスを目的とした、 C および C++ のアプリケーション・プログラム・インターフェースで、関数呼び出しを使用して動的 SQL ステートメントを関数の引き数として受け渡しします。呼び出し可能 SQL インターフェースとは、データベースをアクセスするためのアプリケーション・プログラム・インターフェース (API) で、関数呼び出しを使用して動的 SQL ステートメントを呼び出します。これは組み込み動的 SQL の代替方法ですが、組み込み SQL とは異なり、プリコンパイルやバインドが必要ありません。

DB2 CLI は Microsoft™ の Open Database Connectivity (ODBC) 仕様と、 X/Open® 仕様に基づいています。業界標準に従うように、およびいずれかのデータベース・インターフェースに精通した DB2 アプリケーション・プログラマーがすぐに覚えられるように、これらの仕様が選択されました。

DB2 での ODBC サポートの詳細については、 *コール・レベル・インターフェースの手引きおよび解説書* を参照してください。

JDBC

DB2 の Java サポートには JDBC が含まれます。これは標準化された Java メソッドによるデータ・アクセスをアプリケーションに提供する、ベンダーに依存しない動的 SQL インターフェースです。 JDBC は、JDBC プログラムをプリコンパイルしたりバインドしたりする必要がないという点で、 DB2 CLI に似ています。ベンダーに依存しない標準なので、JDBC アプリケーションは高い移植性を持っています。

JDBC を用いて作成されるアプリケーションは動的 SQL だけです。 JDBC インターフェースを使用すると、処理に余分なオーバーヘッドがかかります。

JDBC の追加情報については、663ページの『JDBC プログラミング』を参照してください。

Microsoft 仕様

ActiveX Data Object (ADO) に準拠したデータベース・アプリケーションは、 Microsoft Visual Basic™ または Visual C++™ で作成することができます。 ADO アプリケーションは OLE DB ブリッジを使用します。 Remote Data Object (RDO) 仕様に準拠したデータベース・アプリケーションは、 Visual Basic で作成することができます。また、OLE DB Provider からデータを戻す OLE DB 表関数を定義することもできます。 OLE DB 表関数に関する詳細については、448ページの『OLE DB 表関数』を参照してください。

本書では、ADO および RDO 仕様に準拠したアプリケーションを作成するためのチュートリアルは記載していません。 ADO および RDO 仕様を使用する DB2 アプリケーションの完全なサンプルについては、以下のディレクトリーを参照してください。

- Visual Basic で作成されたサンプルについては、`sqllib¥samples¥VB`
- Visual C++ で作成されたサンプルについて、`sqllib¥samples¥VC`
- RDO 仕様を使用するサンプルについては、`sqllib¥samples¥RDO`
- Microsoft Transaction Server™ を使用するサンプルについて、`sqllib¥samples¥MTS`

Perl DBI

DB2 は、DBD::DB2 ドライバーを介してデータ・アクセスを行うために、 Perl データベース・インターフェース (DBI) 仕様をサポートします。 Perl DBI を使用して DB2 データベースにアクセスするアプリケーションを作成することの詳細については、 697 ページの『第22章 Perl でのプログラミング』を参照してください。
`http://www.ibm.com/software/data/db2/perl/` にある DB2 ユニバーサル・データベース Perl DBI Web サイトには、最新の DBD::DB2 ドライバーと、各プラットフォームで得られるサポートについての情報が記載されています。

照会製品

IBM 照会管理機能 (QMF) および Lotus Notes を含む照会製品では、照会の開発およびレポート作成をサポートしています。 SQL ステートメントを開発する方法や、導入できるロジックの程度は、製品ごとに異なります。ユーザーの要求によっては、この方法でデータをアクセスするための要件を満たすことができる場合もあります。本書では、照会製品の詳細については記載していません。

データ値の制御

データベースで許可する値を制御することによって妥当性検査とデータ保全性保護を行うことは、アプリケーション・ロジックの従来の機能の 1 つです。アプリケーションには、データが有効であるかどうかを確認するために、入力されたデータ値を特別にチェックするロジックがあります。 (たとえば、部門番号が有効な番号であるかどうかと、

それが既存の部門の番号であるかをチェックします。) DB2 にはこれらの同一の機能をデータベース内から提供するための方法がいくつかあります。

データ・タイプ

データベース内の各データ・エレメントは表の列に保管されており、各列はデータ・タイプを持つように定義されています。このデータ・タイプにより、列の値のタイプに制限が設けられます。たとえば、整数は、固定された範囲内の数値でなければなりません。 SQL ステートメント内で列を使用する際には、整数は文字ストリングと比較できないなどの、一定の動作に従わなければなりません。 DB2 には、特性および動作を定義した組み込みデータ・タイプのセットが含まれています。 DB2 は、*ユーザー定義特殊タイプ* と呼ばれる、ユーザー固有のデータ・タイプの定義もサポートしています。ユーザー定義特殊タイプは組み込みタイプに基づいていますが、組み込みタイプのすべての動作が自動的にサポートされるわけではありません。データ・タイプを 2 進ラージ・オブジェクト (BLOB) と同様に使用して、データ構造など関連する値の集合から成るデータを保管できます。

データ・タイプの追加情報については、 *SQL 解説書* を参照してください。

固有制約

固有制約により、表の中の 1 つまたは複数の列で同じ値が重複するのを避けられます。固有制約では、固有キーと基本キーがサポートされています。たとえば、DEPARTMENT 表の DEPTNO 列に固有制約を定義することにより、同じ部門番号が 2 つの部門で指定されないようにすることができます。

表の中のデータを使用するすべてのアプリケーションに固有規則を設ける必要がある場合には、固有制約を使用してください。固有制約の追加情報については、*SQL 解説書* を参照してください。

表検査の制約

表検査の制約は、表の列で使用できる値について、データ・タイプによる制限よりもさらに詳しい制限を定義するために使用します。表検査の制約は、範囲の検査という形式と、表の中の同じ列にある他の値の検査という形式で行うことができます。

データを使用するすべてのアプリケーションに対して規則を適用するには、表検査の制約を使用して、表中で使用できるデータを制限することができます。これにより、利用の幅が広がるとともに保守しやすくなります。

表検査の制約の追加情報については、*SQL 解説書* を参照してください。

参照保全制約

参照保全 (RI) 制約は、データを使用するすべてのアプリケーションに対して、値ベースの関係を保持しなければならない場合に使用します。たとえば、RI 制約を使用して、EMPLOYEE 表の DEPTNO 列の値を DEPARTMENT 表の値と一致させることができます。この制約は、挿入、更新、削除が行われないようにします。制約がない場合には

DEPARTMENT に関する情報は失われる可能性があります。 RI 規則をデータベースで集中的に適用すると、一般に適用が確実になり保守しやすくなります。

RI 制約の詳しい使用法については、『データの関連の制御』を参照してください。

参照保全の追加情報に関しては、 *SQL 解説書* を参照してください。

CHECK OPTION を使用した視点

アプリケーションの規則で表検査の制約として定義できないものがある場合や、使用するすべてのデータには当てはまらない規則がある場合には、アプリケーションのロジックへの規則の組み込みに代わる方法があります。データの条件を WHERE 文節または WITH CHECK OPTION 文節の一部として指定した、表の視点を作成することができます。この視点定義は、アプリケーションに有効なデータ・セットに対して、検索できるデータを制限します。さらに、視点が更新可能な場合は、 WITH CHECK OPTION 文節がアプリケーションに適用できる行の更新、挿入、および削除を制限します。

WITH CHECK OPTION の追加情報に関しては、 *SQL 解説書* を参照してください。

アプリケーションのロジックとプログラム変数のタイプ

あるプログラミング言語でアプリケーションのロジックを作成する際には、変数も宣言すると、上記の説明と同じくデータに対して制限を課すことができます。さらに、データベースではなくアプリケーションでの規則を適用させるコードの記述を選択できます。以下のような場合に、アプリケーション・サーバーにロジックを組み込んでください。

- 規則が一般的に適用されるものではない。ただし、『CHECK OPTION を使用した視点』に示された視点の場合は除く。
- データベース内のデータの定義を制御できない。
- アプリケーションのロジックを使用する方が、規則を効果的に扱えると思われる。

たとえば、データが入力される順序について入力データのエラーを処理する必要があり、データベース内の操作の順序では正しく検査できない、という場合があります。

データの関連の制御

アプリケーションのロジックのもう 1 つの主な機能は、システム内の異なる論理エンティティー間の関連の管理です。たとえば、新しい部門を追加する場合には、新しい顧客コードを追加する必要があります。 DB2 は、データベース内の異なるオブジェクト間の関連を管理するために、参照保全制約とトリガーという 2 つの方法を備えています。

参照保全制約

データの関連の制御という点から考えた場合、参照保全 (RI) 制約を使用すると、複数の表のデータ間の関連を制御できるようになります。関連付けられた基本キーに影響を与える操作の振る舞い (DELETE や UPDATE など) について定義するには、 CREATE TABLE または ALTER TABLE ステートメントを使用してください。

RI 制約は、1 つまたは複数の表に対してデータの規則を適用します。データを使用するアプリケーションすべてにその規則が当てはまる場合には、 RI 制約の使用により規則がデータベース内に集められます。これにより、利用の幅が広がるとともに保守しやすくなります。

参照保全の追加情報に関しては、 *SQL 解説書* を参照してください。

トリガー

アプリケーションでも実行できるロジックをサポートするために、更新前または更新後にトリガーを使用することができます。トリガーがサポートする規則や操作が、データを使用するすべてのアプリケーションに当てはまる場合は、トリガーの使用により規則や操作がデータベースに集中し、データベースは利用の幅が広がるとともに保守しやすくなります。

トリガーの追加情報については、499ページの『第16章 活動状態の DBMS でのトリガーの使用』または *SQL 解説書* を参照してください。

更新前のトリガーの使用: 更新または挿入の前に実行されるトリガーを使用すると、修正中または挿入中の値を、データベースが実際に修正される前に修正することができます。これは、アプリケーションからの入力 (ユーザーから見たデータ) を希望の内部データベース形式に変換するために使用できます。また、ユーザー定義の関数を介して他の非データベース操作を活動化させる際にも、*事前トリガー* を使用することができます。

更新後のトリガーの使用: 更新、挿入、削除の後に実行されるトリガーには、いくつかの使用法があります。

- 同じ表または別の表にあるデータを更新、挿入、削除できます。これは、データ間の関連の保守、または監査証跡情報の保持に役立ちます。
- 表の中の残りのデータ、または別の表のデータの値を検査できます。これは、RI 制約を利用できない場合や、表の中の他の行または別の表からデータが参照されているために、検査の制約が使用できない場合に役立ちます。
- ユーザー定義関数を使用して、非データベース操作を開始できます。これは、アラート発行やデータベース外部の情報の更新に役立ちます。

アプリケーションのロジック

データベースではなくアプリケーションで規則を適用させたり、または関連操作を実行させるコードを作成することができます。これは、その規則がデータベースに一般的に適用されるものではない場合に行ってください。データベース内のデータ定義全体に対する制御ができない場合や、アプリケーションのロジックで規則や操作を取り扱うほうがより効果的であると思われる場合にも、ロジックをアプリケーション内に置くことができます。

サーバーにおけるアプリケーションのロジック

DB2 が追加機能を提供しているアプリケーション設計の最終段階は、アプリケーションのロジックの一部をデータベース・サーバーで実行するというものです。通常、この設計はパフォーマンスを向上させるために使用しますが、共通機能をサポートするためにサーバー側でアプリケーションのロジックを実行することもできます。

アプリケーションのロジックのこの面については、以下の節で詳しく説明します。
- ストアード・プロシージャー
- ユーザー定義関数
- トリガー

ストアード・プロシージャー

ストアード・プロシージャーはアプリケーションのルーチンで、クライアントのアプリケーションのロジックから呼び出されますが、データベース・サーバーで実行されます。ストアード・プロシージャーを使用する最も一般的な理由は、結果のデータが少量で済むデータベース集中処理を行うためです。これにより、ストアード・プロシージャーの実行中のネットワーク間の通信量を大幅に減らすことができます。また、複数のアプリケーションで共通の操作の集合に対して、ストアード・プロシージャーを使用することもできます。この方法では、すべてのアプリケーションが同じロジックを使用して操作を実行します。

ストアード・プロシージャーの追加情報については、 201ページの『第7章 ストアード・プロシージャー』を参照してください。

ユーザー定義関数

ユーザー定義関数 (UDF) は、以下のものを戻す SQL ステートメント内での操作に使用するために作成することができます。
- 単一のスカラー値 (スカラー関数)
- 非 DB2 データ・ソースから取り出した表。たとえば、ASCII ファイルまたは Web ページなど (表関数)

UDF には SQL ステートメントを含めることはできません。 UDF は、データ値の変換、1 つまたは複数のデータ値に対する計算の実行、値の部分的な抽出 (ラージ・オブジェクトの部分的な抽出) などに役立ちます。

ユーザー定義関数の定義の追加情報については、 409ページの『第15章 ユーザー定義関数 (UDF) とメソッドの作成』を参照してください。

トリガー

31ページの『トリガー』では、トリガーはユーザー定義関数の呼び出しに使用できると記載されています。これは、特定のステートメントが現れたとき、またはデータ値が変更されたときには必ず一定の非 SQL 操作が行われるようにしたいという場合に役立ちます。特定の状況において電子メールを出したり、ファイルにアラート・タイプの情報を書き込むなどの操作が例に示されています。

トリガーの追加情報については、 499ページの『第16章 活動状態の DBMS でのトリガーの使用』を参照してください。

IBM DB2 Universal Database Project Add-In for Microsoft Visual C++

IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ は、 Visual Studio IDE の Visual C++ コンポーネントをプラグインする管理ツールやウィザードの集合体です。これらのツールやウィザードは、組み込み SQL を使用する DB2 アプリケーションの開発に関連したさまざまなタスクを自動化および単純化します。

IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ を使用すれば、 以下のものを開発、パッケージ化、および展開できます:

- Windows 32 ビット・オペレーティング・システム上で稼働する DB2 ユニバーサル・データベース用の C/C++ で作成されたストアード・プロシージャー
- DB2 ユニバーサル・データベース・サーバーにアクセスする Windows 32 ビット C/C++ 組み込み SQL クライアント・アプリケーション
- C/C++ 関数呼び出しラッパーを使用するストアード・プロシージャーに関連した Windows 32 ビット C/C++ クライアント・アプリケーション

IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ を使用すれば、DB2 アプリケーションの構築や展開という実際の作業ではなく、 DB2 アプリケーションの設計や論理の方に焦点を当てることができます。

IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ が実行するタスクには、次のようなものがあります (これらがすべてではありません)。

- 新しい組み込み SQL モジュールの作成
- SQL ステートメントの組み込み SQL モジュールへの挿入 (SQL Assist を使用)
- インポート済みストアード・プロシージャーの追加
- エクスポート済みストアード・プロシージャーの作成
- DB2 プロジェクトのパッケージ化
- DB2 プロジェクトの Visual C++ 内からの展開

IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ は、ツールバーの形式で表示されます。次のようなツールバー・ボタンがあります。

「DB2 プロジェクト特性 (DB2 Project Properties)」
: プロジェクト特性 (開発データベース・オプションとコード生成オプション) を管理します。

「新規 DB2 オブジェクト (New DB2 Object)」
: 新しい組み込み SQL モジュール、インポート済みストアード・プロシージャー、またはエクスポート済みストアード・プロシージャーを追加します。

「DB2 組み込み SQL モジュール (DB2 Embedded SQL Modules)」
: 組み込み SQL モジュールとそれらに関連したプリコンパイラーのリストを管理します。

「DB2 インポート済みストアード・プロシージャー (DB2 Imported Stored Procedures)」
: インポート済みストアード・プロシージャーのリストを管理します。

「DB2 エクスポート済みストアード・プロシージャー (DB2 Exported Stored Procedures)」
: エクスポート済みストアード・プロシージャーのリストを管理します。

「DB2 プロジェクトのパッケージ化 (Package DB2 Project)」
: DB2 外部プロジェクト・ファイルをパッケージ化します。

「DB2 プロジェクトの展開 (Deploy DB2 Project)」
: パッケージ化されている DB2 外部プロジェクト・ファイルを展開します。

IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ には、以下の 3 つの隠しボタンもあります。これらのボタンは、標準的な Visual C++ ツール・カスタマイズ・オプションを使用すれば見えるようになります。

「新規 DB2 組み込み SQL モジュール (New DB2 Embedded SQL Module)」
: 新しい C/C++ 組み込み SQL モジュールを追加します。

「新規 DB2 インポート済みストアード・プロシージャー (New DB2 Imported Stored Procedure)」
: 新しいデータベース・ストアード・プロシージャーをインポートします。

「新規 DB2 エクスポート済みストアード・プロシージャー (New DB2 Exported Stored Procedure)」
: 新しいデータベース・ストアード・プロシージャーをエクスポートします。

IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ には、以下のコード・エレメントを自動的に生成する機能があります。

- オプションのサンプル SQL ステートメントを含む、骨組みとなる組み込み SQL モジュールのファイル
- 標準データベース接続および組み込み SQL 関数の切断
- インポート済みストアード・プロシージャーの呼び出しラッパー関数
- エクスポート済みストアード・プロシージャーの関数テンプレート
- エクスポート済みストアード・プロシージャーのデータ定義言語 (DDL) ファイル

IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ に関連した用語:

IDE プロジェクト
: 標準的な Visual C++ プロジェクト。

DB2 プロジェクト
: IDE プロジェクトに挿入される DB2 プロジェクト・オブジェクトの集合体。DB2 プロジェクト・オブジェクトは、どの Visual C++ プロジェクトにも挿入できる。 DB2 プロジェクトを使用すれば、組み込み SQL モジュール、インポート済みストアード・プロシージャー、エクスポート済みストアード・プロシージャーなど、さまざまな DB2 オブジェクトを管理できる。これらのオブジェクトとその特性は、追加、削除、変更が可能である。

モジュール
: SQL ステートメントを含んでいる可能性のある C/C++ ソース・コード・ファイル。

開発データベース
: 組み込み SQL モジュールのコンパイルに使用されるデータベース。開発データベースは、インポート可能データベースのストアード・プロシージャー定義のリストを検索するためにも使用される。

組み込み SQL モジュール
: 組み込み静的または動的 SQL を含む C/C++ ソース・コード・ファイル。

インポート済みストアード・プロシージャー
: すでにデータベース内に定義されていて、プロジェクトによって呼び出されるストアード・プロシージャー。

エクスポート済みストアード・プロシージャー
: プロジェクトによって構築および定義される、データベース・ストアード・プロシージャー。

IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ の活動化

IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ を活動化させるには、以下のステップを実行します。

ステップ 1. 現行のログイン ID を使用して、少なくとも 1 回 Visual C++ を開始して、停止させます。 Visual C++ を最初に実行するときに、ユーザー ID 用にプロファイルが作成され、 db2vccmd コマンドによって更新されます。一度も開始せずに db2vccmd を実行しようとすると、以下のようなエラーが出される可能性があります。

```
"Registering DB2 Project add-in ...Failed! (rc = 2)"
```

ステップ 2. まだアドインを登録していない場合は、コマンド行で以下を入力してアドインを登録してください。

```
db2vccmd register
```

ステップ 3. **「ツール —> カスタマイズ (Tools —> Customize)」**を選択します。「カスタマイズ (Customize)」 ノートブックがオープンします。

ステップ 4. **「アドインおよびマクロ・ファイル (Add-ins and Macro Files)」**タブを選択します。「アドインおよびマクロ・ファイル (Add-ins and Macro Files)」ページがオープンします。

ステップ 5. **「IBM DB2 Project Add-In」**チェック・ボックスを選択します。

ステップ 6. **「了解 (OK)」**をクリックします。浮動ツールバーが作成されます。

注: ツールバーを誤ってクローズしてしまった場合は、いったんアドインを非活動化して後に再び活動化させるか、または Microsoft Visual C++ 標準カスタマイズ・オプションを使用すれば、ツールバーを再表示できます。

IBM DB2 Universal Database Tools Add-In for Microsoft Visual C++ の活動化

DB2 Tools Add-In は、一部の DB2 管理ツールおよび開発ツールを Visual C++ 統合開発環境内から立ち上げられるようにするツールバーです。

IBM DB2 Universal Database Tools Add-In for Microsoft Visual C++ を活動化させるには、以下のステップを実行します。

ステップ 1. 現行のログイン ID を使用して、少なくとも 1 回 Visual C++ を開始して、停止させます。 Visual C++ を最初に実行するときに、ユーザー ID 用にプロファイルが作成され、 db2vccmd コマンドによって更新されます。一度も開始せずに db2vccmd を実行しようとすると、以下のようなエラーが出される可能性があります。

```
"Registering DB2 Project add-in ...Failed! (rc = 2)"
```

ステップ 2. まだアドインを登録していない場合は、コマンド行で以下を入力してアドインを登録してください。

```
db2vccmd register
```

ステップ 3. **「ツール —> カスタマイズ (Tools —> Customize)」**を選択します。「カスタマイズ (Customize)」ノートブックがオープンします。

ステップ 4. 「アドインおよびマクロ・ファイル (Add-ins and Macro Files)」タブを選択します。

ステップ 5. **「IBM DB2 Tools Add-In」**チェック・ボックスを選択します。

ステップ 6. **「了解 (OK)」**をクリックします。浮動ツールバーが作成されます。

注: ツールバーを誤ってクローズしてしまった場合は、いったんアドインを非活動化して後に再び活動化させるか、または Visual C++ 標準カスタマイズ・オプションを使用すれば、ツールバーを再表示できます。

IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ の詳細については、以下を参照してください。

- IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ のオンライン・ヘルプ

- http://www.ibm.com/software/data/db2/udb/ide/index.html

サポートされる SQL ステートメント

SQL 言語は、アプリケーション内からのデータ定義、検索、更新、および制御操作を提供します。 759ページの表38 は、DB2 製品がサポートする SQL ステートメントと、そのステートメントが CLP または DB2 CLI によって動的にサポートされているかどうかを示しています。 759ページの表38 を早見表として使用することもできます。構文など、すべてのステートメントの完全な説明については、 *SQL 解説書* を参照してください。

許可についての考慮事項

許可 を与えられたユーザーまたはグループは、データベースへの接続、表の作成、システムの管理などの一般的なタスクを実行できます。*特権* を付与されたユーザーやグループは、特定の方法で特定のデータベースにアクセスできます。 DB2 は、保管された情報を保護するために特権のセットを使用します。さまざまな特権の詳細については、 *管理の手引き: 計画* を参照してください。

大部分の SQL ステートメントは、ステートメントが使用するデータベース・オブジェクトに対する何らかのタイプの特権を必要とします。ほとんどの API 呼び出しは、通常データベース・オブジェクトに対するいかなる特権も必要としませんが、権限を持っていないと呼び出すことができないものが多くあります。 DB2 API を使用すると、アプリケーション・プログラムから DB2 の管理機能を実行できます。たとえば、データベース内にバインド・ファイルの必要ないパッケージを作成するには、 `sqlarbnd` (または REBIND) API を使用できます。各 DB2 API の詳細については、*管理 API 解説書* を参照してください。

各 SQL ステートメントを発行するために必要な特権については、 *SQL 解説書* を参照してください。各 API 呼び出しを発行するために必要な特権および権限については、*管理 API 解説書* を参照してください。

アプリケーションを設計する際に、ユーザーがアプリケーションを実行するために必要な特権を考慮する必要があります。ユーザーが必要とする特権は、以下によって決まります。

- アプリケーションが動的 SQL (JDBC および DB2 CLI を含む) と静的 SQL のどちらを使用するか
- アプリケーションがどの API を使用するか

動的 SQL

DYNAMICRULES RUN (デフォルト) でバインドされたパッケージで動的 SQL を使用するには、動的 SQL アプリケーションを実行するユーザーが、パッケージに対する EXECUTE 特権だけでなく、実行する各 SQL 要求の発行に必要な特権も持っていなけ

ればなりません。ユーザーの許可 ID、ユーザーがメンバーとなっているグループ、または PUBLIC に対して特権を付与することができます。

DYNAMICRULES BIND オプションを指定してアプリケーションをバインドする場合は、許可 ID とアプリケーション・パッケージが関連付けられます。これにより、アプリケーションを実行するどのユーザーでも、その許可 ID に関連付けられた特権を継承することができます。

アプリケーション (組み込み動的 SQL アプリケーションの場合) をバインドする場合は、プログラムに静的 SQL が含まれていなければ、必要になるのはデータベースに対する BINDADD 権限だけです。この特権も、ユーザーの許可 ID、ユーザーがメンバーとなっているグループ、または PUBLIC に対して付与することができます。

DYNAMICRULES BIND オプションを指定して動的 SQL パッケージをバインドする場合、アプリケーションを実行するユーザーにはパッケージに対する EXECUTE 特権だけが必要です。 DYNAMICRULES BIND オプションを指定して動的 SQL アプリケーションをバインドするには、そのアプリケーションですべての動的および静的 SQL ステートメントを実行するのに必要な特権を持っている必要があります。 SYSADM または DBADM 権限を持っており、 DYNAMICRULES BIND オプションを指定してパッケージをバインドする場合は、 OWNER BIND オプションを使って異なる許可 ID を指定することを考慮してください。 OWNER BIND オプションを使用すると、動的 SQL ステートメントに対する SYSADM または DBADM 特権を、パッケージが自動的に継承しないようにすることができます。 DYNAMICRULES BIND および OWNER BIND オプションの詳細については、 *コマンド解説書* の BIND コマンドの節を参照してください。

静的 SQL

静的 SQL は、アプリケーションを実行しているユーザーがパッケージに対する EXECUTE 特権を持っていれば使用できます。パッケージを構成するそれぞれのステートメントに対する特権は必要ありません。 EXECUTE 特権は、ユーザーの許可 ID、ユーザーがメンバーとなっているグループ、または PUBLIC に対して付与することができます。

アプリケーションをバインドするときに VALIDATE RUN オプションを指定しない場合は、アプリケーションのバインドに使用する許可 ID に、アプリケーション内にあるすべてのステートメントを実行できるだけの権限が必要です。 BIND 時に VALIDATE RUN を指定した場合は、このパッケージ内にある静的 SQL に関連してどのような許可障害が発生しても BIND は成功し、その中にあるステートメントの妥当性は実行時に再び検査されます。アプリケーションをバインドするためには、BINDADD 権限が必要です。ステートメントを実行するのに必要な特権は、ユーザーの許可 ID または PUBLIC に付与するようにしてください。静的 SQL ステートメントをバインドする際には、グ

ループ特権は使用しません。動的 SQL の場合と同様に、BINDADD はユーザーの許可 ID、ユーザーがメンバーとなっているグループ、または PUBLIC に対して付与することができます。

上述の静的 SQL の特性により、DB2 にある情報のアクセスを正確に制御できます。このことを実行できるアプリケーションについては、この節の最後の例を参照してください。

API の使用

DB2 が提供する API の大部分は特権を使用する必要がありませんが、呼び出しに何らかの権限を必要とするものが多くあります。特権が必要ない API の場合は、アプリケーションを実行しているユーザーに特権を付与しなければなりません。特権は、ユーザーの許可 ID、ユーザーがメンバーとなっているグループ、または PUBLIC に対して付与することもできます。各 API 呼び出しを発行するために必要な特権および権限については、 *管理 API 解説書* を参照してください。

例

STAFF 表に対する照会を行う必要のある 2 名のユーザー、PAYROLL と BUDGET があるとします。 PAYROLL は会社の従業員の給与支払いを管理しており、給与支払い小切手を出す際に、さまざまな SELECT ステートメントを発行する必要があります。 PAYROLL は各従業員の給与にアクセスできなければなりません。 BUDGET は、給与の支払いにいくら必要であるかの決定を管理しています。しかし、BUDGET が個々の従業員の給与を見ることはできないようにしておかなければなりません。

PAYROLL がさまざまな SELECT ステートメントを発行しているため、 PAYROLL 用に設計されたアプリケーションは、動的 SQL を使用することになるでしょう。そのため、PAYROLL には STAFF 表に対する SELECT 特権が必要になります。 PAYROLL はどうしてもこの表すべてをアクセスする必要があるので、 SELECT 特権を付与することに問題はありません。

一方 BUDGET は、個々の従業員の給与にアクセスさせてはなりません。つまり、 STAFF 表に対する SELECT 特権を BUDGET に付与してはならないということです。 BUDGET は STAFF 表全体の給与の合計にアクセスする必要があるので、 SELECT SUM (SALARY) FROM STAFF を実行するための静的 SQL アプリケーションを作成してバインドし、このアプリケーションのパッケージに対する EXECUTE 特権を BUDGET に付与することができます。これにより BUDGET は、アクセスしてはならない情報を使用せずに必要な情報を取得できます。

組み込み SQL または DB2 CLI プログラムで使用されるデータベース・マネージャー API

アプリケーションは API を使用して、 SQL ステートメントでは使用できないデータベース・マネージャー機能にアクセスすることができます。データベース・マネージャーで使用可能な API の詳細とその呼び出し方法については、 *管理 API 解説書* の例を参照してください。

DB2 API は、以下の目的のために使用できます。

- データベース・マネージャー環境を操作する。これには、データベースとノードのカタログ作成とカタログ解除、データベースとノード・ディレクトリーの走査が含まれます。データベースの作成、削除、および移行を行うための API も使用できます。
- データのエクスポートとインポート、およびデータベースの管理、バックアップ、保管の機能を提供する。
- データベース・マネージャーの構成ファイルとデータベースの構成ファイルを操作する。
- クライアント / サーバー環境固有の操作を提供する。
- プリコンパイル済み SQL ステートメントの実行時インターフェースを提供する。この API は通常、プログラマーが直接呼び出すものではありません。その代わり、プリコンパイラーによる処理の後に、修正済みソース・ファイルに挿入されます。

データベース・マネージャーには、独自のプリコンパイラーを作成することを希望する言語ベンダーのための API や、アプリケーションの開発に有効なその他の API があります。

データベース・マネージャーで使用可能な API の詳細とその呼び出し方法については、 *管理 API 解説書* の例を参照してください。

テスト環境のセットアップ

これ以降の節で説明するタスクの多くを実行するには、テスト環境をセットアップする必要があります。たとえば、アプリケーションの SQL コードをテストするには、データベースが必要です。

テスト環境には、以下のものを含めてください。

- **テスト・データベース**。アプリケーションが表や視点からデータの更新、挿入、または削除を行う場合は、テスト・データを使用してそのアプリケーションの実行を検査しなければなりません。アプリケーションが表や視点からデータの取り出しのみを行う場合は、テストの際に実動レベルのデータを使用することを考慮してください。
- **テスト入力データ**。アプリケーションのテストに使用する入力データは、可能な入力条件をすべて表す有効なデータにするべきです。アプリケーションが入力データが有

効かどうかを検査する場合は、有効なデータと無効なデータの両方を含めるようにして、有効なデータは処理され、無効なデータにはフラグが付けられることを検査してください。

テスト・データベースの作成

テスト・データベースを作成しなければならない場合には、 CREATE DATABASE API を呼び出す小規模なサーバー・アプリケーションを作成するか、またはコマンド行プロセッサーを使用してください。コマンド行プロセッサーについては*コマンド解説書* を、CREATE DATABASE API については*管理 API 解説書* を参照してください。

テスト表の作成

必要なテスト表また視点を設計するには、まずアプリケーションのデータ要求を分析してください。表を作成するには、スキーマで CREATETAB 権限と CREATEIN 特権が必要です。代替権限については、 *SQL 解説書* にある CREATE TABLE に関する情報を参照してください。

アプリケーションがアクセスするデータをリストし、それぞれのデータ項目がどのようにアクセスされるかを記述してください。たとえば、開発中のアプリケーションが TEST.TEMPL、TEST.TDEPT、および TEST.TPROJ 表にアクセスするとします。アクセスのタイプは、表1 のように記録されることになります。

表 1. アプリケーション・データの記述

表または視点名	行の挿入	行の削除	列名	データ・タイプ	アクセスの更新
TEST.TEMPL	なし	なし	EMPNO	CHAR(6)	あり
			LASTNAME	VARCHAR(15)	あり
			WORKDEPT	CHAR(3)	あり
			PHONENO	CHAR(4)	
			JOBCODE	DECIMAL(3)	
TEST.TDEPT	なし	なし	DEPTNO	CHAR(3)	
			MGRNO	CHAR(6)	
TEST.TPROJ	あり	あり	PROJNO	CHAR(6)	あり
			DEPTNO	CHAR(3)	あり
			RESPEMP	CHAR(6)	あり
			PRSTAFF	DECIMAL(5,2)	あり
			PRSTDATE	DECIMAL(6)	あり
			PRENDATE	DECIMAL(6)	

アプリケーション・データ・アクセスの記述が完成したら、そのアプリケーションに必要なテスト表および視点を構成してください。

- アプリケーションが表または視点のデータを修正する場合にテスト表を作成する。CREATE TABLE SQL ステートメントを用いて以下のテスト表を作成します。
  - TEMPL
  - TPROJ
- アプリケーションが実動データベースのデータを修正しない場合にテスト視点を作成する。

  この例では、CREATE VIEW SQL ステートメントを用いて TDEPT 表のテスト視点を作成します。

アプリケーションと一緒にデータベース・スキーマを開発している場合は、テスト表の定義が開発プロセス中に繰り返し詳細化されていきます。 1 次アプリケーションは普通、表を作成することもそれにアクセスすることもできません。それはデータベース・マネージャーが実際にはない表や視点を参照するステートメントをバインドできないからです。時間をかけずに表の作成および変更を処理するには、表を作成するために別個のアプリケーションを開発することを考慮してください。もちろん、いつでもコマンド行プロセッサー (CLP) を使用して対話的にテスト表を作成することができます。

テスト・データの生成

データを表に挿入するには、以下の方法のいずれかを使用してください。

- INSERT...VALUES (SQL ステートメント) は、コマンドが実行されるたびに表に 1 行以上の行を挿入する。
- INSERT...SELECT は、既存表からデータを入手し (SELECT 文節に基づく)、そのデータを INSERT ステートメントで識別された表に挿入する。
- IMPORT または LOAD ユーティリティーは、定義されたソースから大量の新規データまたは既存データを挿入する。
- RESTORE ユーティリティーは、元のデータベースの BACKUP コピーを使用して、既存のデータベースの内容を同一のテスト・データベースにコピーする。

INSERT ステートメントの詳細については、 *SQL 解説書* を参照してください。IMPORT、LOAD、および RESTORE ユーティリティーについては、 *管理の手引き* を参照してください。

ランダムに生成されたテスト・データを表に挿入するための技法は、次の SQL ステートメントに具体的に示されています。以下の CREATE TABLE ステートメントにあるように、 EMP 表には 4 つの列、ENO (従業員番号)、LASTNAME (名字)、HIREDATE (入社日付)、そして SALARY (従業員の給料) があるものとします。

```
CREATE TABLE EMP (ENO INTEGER, LASTNAME VARCHAR(30),
                  HIREDATE DATE, SALARY INTEGER);
```

この表で、従業員番号には 1 からある数値、たとえば 100 までを、残りの列にはランダム・データを挿入するとします。このことを実行するには、次の SQL ステートメントを使用します。

```
INSERT INTO EMP
     -- 100 個のレコードを生成する。
WITH DT(ENO) AS (VALUES(1) UNION ALL
SELECT ENO+1 FROM DT WHERE ENO < 100 )  1

     -- 次に、DT 内に生成されたレコードを使用して、従業員レコードの
     -- 他の列を作成します。
SELECT ENO,  2
    TRANSLATE(CHAR(INTEGER(RAND()*1000000)),  3
              CASE MOD(ENO,4) WHEN 0 THEN 'aeiou' || 'bcdfg'
                              WHEN 1 THEN 'aeiou' || 'hjklm'
                              WHEN 2 THEN 'aeiou' || 'npqrs'
                                     ELSE 'aeiou' || 'twxyz' END,
                                          '1234567890') AS LASTNAME,
    CURRENT DATE - (RAND()*10957) DAYS AS HIREDATE,  4
    INTEGER(10000+RAND()*200000) AS SALARY  5
  FROM DT;

SELECT * FROM EMP;
```

上記のステートメントについて説明します。

1. INSERT ステートメントの最初の部分では、再帰副照会を用いて従業員番号を生成し、最初の 100 人の従業員用の 100 個のレコードを生成します。各レコードには、従業員番号が入ります。従業員数を変更するには、100 ではない数値を入力してください。
2. SELECT ステートメントで LASTNAME 列を生成します。まず RAND 関数を使用して、最大 6 桁の長さのランダム整数を生成します。次に CHAR 関数を使用して、その整数を数字形式に変換します。
3. 数字を英字に変換するために、TRANSLATE 関数を使用して、 10 の異なる数字 (0 から 9) をそれぞれの英字に変換します。 11 個以上の英字があるので、ステートメントは 5 つの異なる方式から変換方式を選択します。その結果、発音できる十分なランダム母音を持つ名前が生成され、その母音はそれぞれの変換で含められます。
4. ステートメントは、ランダムな HIREDATE 値を生成します。 HIREDATE の値は、現在日付から 30 年前までに及んでいますが、現在日付から 0 と 10 957 間のランダム日数を減算して計算されます。 (10 957 は 30 年間の日数です。)
5. 最後に、ステートメントは SALARY をランダムに生成します。給与の最小値は 10 000 で、この値に 0 ～ 200 000 のランダムな数値が加えられます。

ランダムなテスト・データを生成するのに役立つプログラムの実例は、`sqllib/samples/c` サブディレクトリー内の `fillcli.sqc` および `fillsrv.sqc` のサンプル・プログラムを参照してください。

開発中のすべてのユーザー定義関数 (UDF) を、テスト・データのプロトタイプとしたい場合もあります。 UDF を定義する理由と方法の詳細については、

409ページの『第15章 ユーザー定義関数 (UDF) とメソッドの作成』および
389ページの『第14章 ユーザー定義関数 (UDF) およびメソッド』を参照してください。

プログラムの実行、テスト、およびデバッグ

アプリケーション構築の手引き には、ご使用の環境でプログラムを実行する方法が説明されています。以下の方法は、コードのテストおよびデバッグに役立ちます。

- 『SQL ステートメントのプロトタイプ化』で説明する技法と同じ技法を使用する。これには、コマンド行プロセッサー、Explain 機能、プログラムが操作する表およびデータベースに関する情報のシステム・カタログの分析、および製品条件をシミュレートするためのシステム・カタログ統計の更新が含まれます。
- データベース・システム・モニターを使用して、分析のための最適化情報を取得する。 *システム・モニター 手引きおよび解説書* を参照してください。
- 標識機能を使用して、 DB2 ユニバーサル・データベース (OS/390 版) 用に開発されているアプリケーション内の SQL ステートメントの構文を検査したり、 SQL92 基本レベル標準に準拠させる。この機能はプリコンパイル中に呼び出されます。この方法については、54ページの『プリコンパイル』の節の最後の方を参照してください。
- エラー処理 API をすべて利用する。たとえば、エラー処理 API を使用して、テスト段階中のメッセージをすべて印刷できます。エラー処理 API の詳細については、 *管理 API 解説書* を参照してください。

SQL ステートメントのプロトタイプ化

アプリケーションを設計およびコード化するにつれて、データベース・マネージャー機能とユーティリティーを利用して、 SQL コード部分をプロトタイプ化したりパフォーマンスを向上することができます。たとえば、次の事柄が行えます。

- 完成したプログラムをコンパイルし、リンクする前に、コマンド・センターまたはコマンド行プロセッサー (CLP) を使用して多数の SQL ステートメントをテストする。

  これにより、データベースの表、索引、または視点に保管された情報を定義および操作できます。情報を追加、削除、更新するだけでなく、表の内容から報告書を作成できます。組み込み SQL プログラム内のホスト変数を使用するために、最低限いくつかの SQL ステートメントの構文を変更しなければなりません。ホスト変数は、画面に出力されるデータを保管するために使用されます。さらに、組み込み SQL ステートメントの中には、環境に関連していないため、コマンド・センターまたは CLP によってサポートされないもの (BEGIN DECLARE SECTION など) もあります。 CLP によりサポートされていない SQL ステートメントを調べるには、 759ページの表38 を参照してください。

さらに、コマンド行プロセッサー要求の入出力をリダイレクトできます。たとえば、コマンド行プロセッサー要求に入力を行う際に必要となる SQL ステートメントを含んだファイルを 1 または複数作成できます。そうすれば、ステートメントを再入力する必要がなくなります。

コマンド行プロセッサーの詳細については、*コマンド解説書* を参照してください。コマンド・センターの詳細については、*管理の手引き* を参照してください。

- プログラムでの使用を計画している DELETE、INSERT、UPDATE、または SELECT ステートメントの概算コストを調べるために Explain 機能を使用する。 Explain 機能は、対象ステートメントの構造および概算コストについての情報を、ユーザーが提供する表に配置します。 この情報は、Visual Explain または `db2exfmt` ユーティリティーを使用して調べることができます。

  Explain 機能の使用方法については、*管理の手引き: インプリメンテーション* を参照してください。

- システム・カタログ視点を使用して、既存のデータベースについての情報を取り出しやすくする。視点の基準になるシステム・カタログ表は、データベースが作成、変更、更新される通常操作の間、データベース・マネージャーによって作成され、維持されます。これらの視点には、付与された権限、列名、データ・タイプ、索引、パッケージへの依存性、参照制約、表名など、それぞれのデータベースについてのデータが含まれます。システム・カタログ視点内のデータは、通常の SQL 照会機能により使用できます。

  SQL 最適化プログラムが使用する統計情報を含むシステム・カタログ視点の中には、更新が可能なものもあります。これらの視点の列の中には、最適化プログラムに作用するか、または仮定データベースのパフォーマンスを調査できるように変更が可能なものもあります。この方法は、ユーザーの開発システムまたはテスト・システム上の実動システムのシミュレート、および照会方法の分析に使用できます。

  各システム・カタログ視点のすべての説明については、 *SQL 解説書* 内の付録を参照してください。システム・カタログ統計およびその中のユーザーが変更できるものについては、 *管理の手引き: インプリメンテーション* を参照してください。

第2部 アプリケーション内の組み込み SQL

© Copyright IBM Corp. 1993, 2001

第3章 組み込み SQL の概説

ホスト言語における組み込み SQL ステートメントの使用 49
ソース・ファイルの作成と準備. 51
組み込み SQL 用のパッケージの作成 . . . 54
プリコンパイル 54
ソース・ファイル要件. 56
コンパイルとリンク 57
バインド 58
パッケージの名前変更. 59
動的ステートメントのバインド. . . . 59
非修飾表名の解決 60
その他のバインドの考慮事項 60
バインドの実行据え置きの利点. 61
DB2 バインド・ファイル記述ユーティリティー - db2bfd 62
アプリケーション、バインド・ファイル、およびパッケージの関係 62
タイム・スタンプ 63
再バインド 64

ホスト言語における組み込み SQL ステートメントの使用

ホスト言語に組み込まれた SQL ステートメントを使用して、アプリケーションを作成することができます。 SQL ステートメントはデータベース・インターフェースを提供し、一方、ホスト言語はアプリケーションの実行に必要なサポートの残りの部分を提供します。

表2 は、ホスト言語アプリケーションに組み込まれた SQL ステートメントを示しています。この例では、更新が正常に行われたかどうかを判断するために、 SQLCA 構造の SQLCODE フィールドをアプリケーションが検査します。

表 2. ホスト言語における組み込み SQL ステートメントの使用

言語	ソース・コード例

C/C++	`EXEC SQL UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr';`
	`if ( SQLCODE < 0 )`
	`   printf( "Update Error:  SQLCODE = %ld ¥n", SQLCODE );`

Java (SQLJ)	`try {`
	`   #sql { UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr' };`
	`}`
	`catch (SQLException e) {`
	`   println( "Update Error:  SQLCODE = " + e.getErrorCode() );`
	`}`

COBOL	`EXEC SQL UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr' END_EXEC.`
	`IF SQLCODE LESS THAN 0`
	`   DISPLAY 'UPDATE ERROR:  SQLCODE = ', SQLCODE.`

FORTRAN	`EXEC SQL UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr'`
	`if ( sqlcode .lt. 0 ) THEN`
	`   write(*,*) 'Update error:  sqlcode = ', sqlcode`

© Copyright IBM Corp. 1993, 2001

アプリケーションに組み込まれる SQL ステートメントは、ホスト言語固有のものではありません。データベース・マネージャーには、SQL 構文をホスト言語が処理できるように変換する機能があります。

C、C++、COBOL または FORTRAN 言語では、 DB2 プリコンパイラーによって変換されます。 DB2 プリコンパイラーは、PREP コマンドで呼び出します。プリコンパイラーは、組み込み SQL ステートメントを DB2 ランタイム・サービス API 呼び出しに直接変換します。

Java 言語では、SQLJ プログラムは SQLJ 文節を JDBC ステートメントに変換します。 SQLJ 変換プログラムは、SQLJ コマンドで呼び出します。

プリコンパイラーはソース・ファイルを処理する際に、特に SQL ステートメントを探して処理し、非 SQL ホスト言語は無視します。 SQL ステートメントは特殊な区切り文字で囲まれているので、プリコンパイラーはこれを発見することができます。使用している言語の組み込み SQL ステートメントに必要な構文情報については、以下のページを参照してください。
- C/C++ の場合は、615ページの『C および C++ での組み込み SQL ステートメント』
- Java (SQLJ) の場合は、674ページの『Java での SQL ステートメントの組み込み』
- COBOL の場合は、705ページの『COBOL での SQL ステートメントの組み込み』
- FORTRAN の場合は、728ページの『FORTRAN での SQL ステートメントの組み込み』
- REXX の場合は、741ページの『REXX での SQL ステートメントの組み込み』

表3 は、区切り文字と注釈を使用して、サポートされているコンパイル済みホスト言語で有効な組み込み SQL ステートメントを作成する方法を示しています。

表 3. ホスト言語における組み込み SQL ステートメントの使用

言語	ソース・コード例

C/C++	`/* Only C or C++ comments allowed here */`
	`EXEC SQL`
	`   -- SQL comments or`
	`   /* C comments or */`
	`   // C++ comments allowed here`
	`   DECLARE C1 CURSOR FOR sname;`
	`/* Only C or C++ comments allowed here */`

SQLJ	`/* Only Java comments allowed here */`
	`#sql c1 = {`
	`   -- SQL comments or`
	`   /* Java comments or */`
	`   // Java comments allowed here`
	`   SELECT name FROM employee`
	`};`
	`/* Only Java comments allowed here */`

表 3. ホスト言語における組み込み SQL ステートメントの使用 (続き)

<table>
<tr><th>言語</th><th>ソース・コード例</th></tr>
<tr><td>COBOL</td><td><pre>
 * See COBOL documentation for comment rules
 * Only COBOL comments are allowed here
   EXEC SQL
     -- SQL comments or
 *   full-line COBOL comments are allowed here
     DECLARE C1 CURSOR FOR sname END-EXEC.
 * Only COBOL comments are allowed here
</pre></td></tr>
<tr><td>FORTRAN</td><td><pre>
C   Only FORTRAN comments are allowed here
      EXEC SQL
     + -- SQL comments, and
C   full-line FORTRAN comment are allowed here
     + DECLARE C1 CURSOR FOR sname
      I=7 ! End of line FORTRAN comments allowed here
C   Only FORTRAN comments are allowed here
</pre></td></tr>
</table>

ソース・ファイルの作成と準備

ソース・コードは、テキスト・エディターを使用して、ソース・ファイルと呼ばれる ASCII ファイル内に作成されます。ソース・ファイルには、コーディングに使用しているホスト言語に適した拡張子が付いていなければなりません。使用しているホスト言語に必要なファイル拡張子を調べるには、
767ページの表39 を参照してください。

注: すべてのプラットフォームがすべてのホスト言語をサポートするわけではありません。関係する特定の情報については、 *アプリケーション構築の手引き* を参照してください。

ここでは、すでにソース・コードが記述されていることを前提として説明しています。

コンパイル済みホスト言語を使用してアプリケーションを記述した場合は、さらに追加の手順に従ってアプリケーションを作成する必要があります。プログラムのコンパイルとリンクに加えて、*プリコンパイル* および*バインド* を行わなくてはなりません。

簡単に言えば、プリコンパイルによって、組み込み SQL ステートメントをホスト・コンパイラーが処理できる DB2 実行時 API 呼び出しに変換し、バインド・ファイルを作成します。バインド・ファイルには、アプリケーション・プログラム内の SQL ステートメントに関する情報が入ります。 BIND コマンドを実行すると、データベース内に*パッケージ* が作成されます。任意で、プリコンパイラーがプリコンパイル時にバインドを行うようにさせることができます。

バインドとは、バインド・ファイルからパッケージ を作成し、それをデータベースに保管する処理です。アプリケーションが複数のデータベースにアクセスする場合、パッケージはデータベースごとに作成する必要があります。

53ページの図1 は、上記のステップの順序とともに、一般的なコンパイル済み DB2 アプリケーションのさまざまなモジュールを示しています。プログラム作成の各段階で何を行うかについて説明した、これ以降の節を読む際に、この図を参照すると役立ちます。

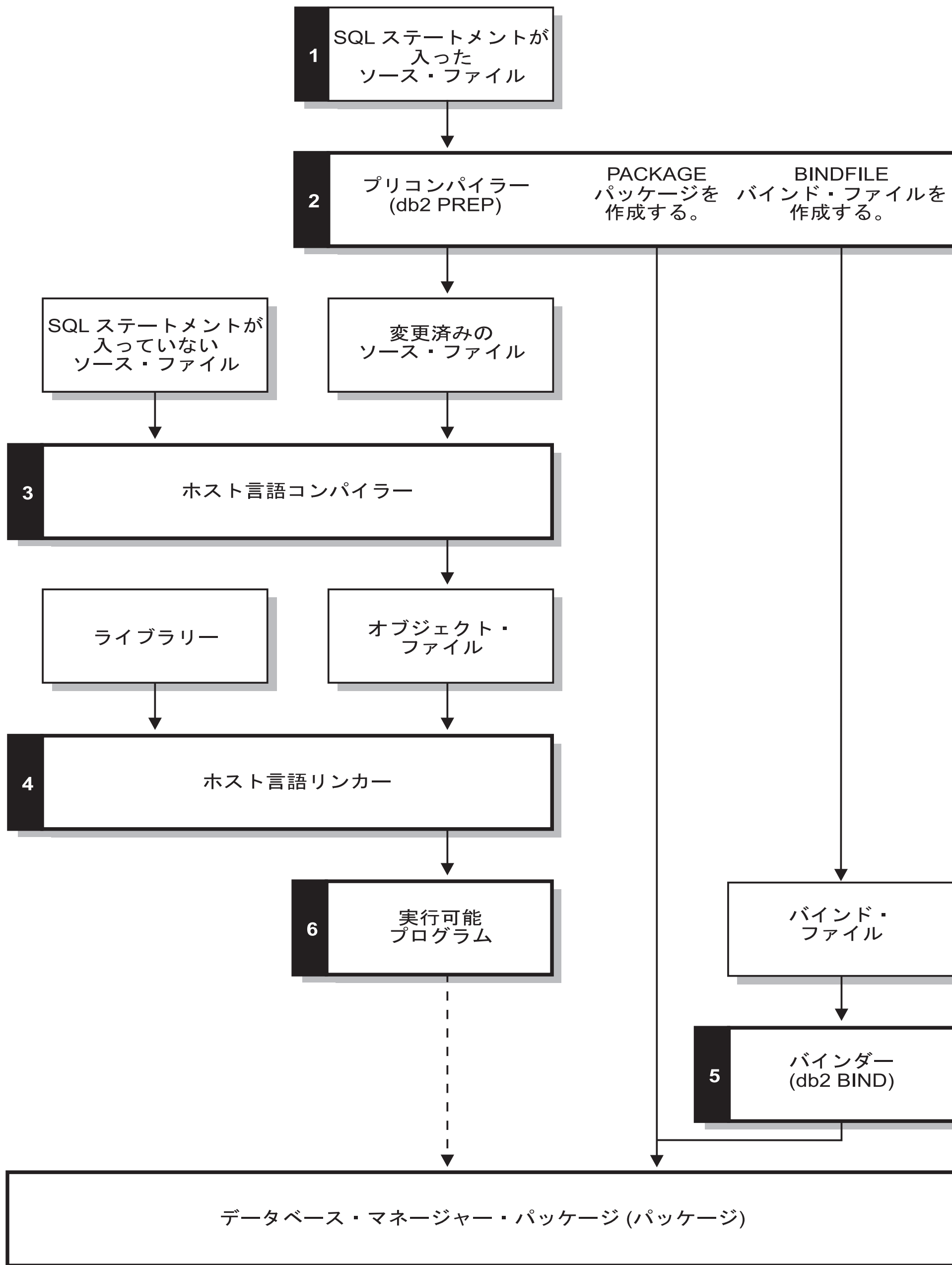


図 1. コンパイル済みホスト言語で記述されたプログラムの作成

組み込み SQL 用のパッケージの作成

コンパイル済みホスト言語で記述されたアプリケーションを実行するには、データベース・マネージャーが実行時に必要とするパッケージを作成しなければなりません。これには、53ページの図1 で示している以下のステップが含まれます。

- プリコンパイル (ステップ 2)。組み込み SQL のソース・ステートメントをデータベース・マネージャーが使用できる形式に変換する作業です。
- コンパイルとリンク (ステップ 3 および 4)。必要なオブジェクト・モジュールを作成する作業です。
- バインド (ステップ 5)。プログラムの実行時にデータベース・マネージャーが使用するパッケージを作成する作業です。

この節では、その他にも以下のトピックについて説明します。

- アプリケーション、バインド・ファイル、およびパッケージの関係。
- 再バインド。パッケージを再バインドする場合とその方法を説明します。

SQLJ アプリケーションが必要とするパッケージを作成するには、 `SQLJ` 変換プログラムと `db2profc` コマンドが必要です。 `SQLJ` 変換プログラムの使用に関する詳細については、 671ページの『SQLJ プログラミング』を参照してください。

プリコンパイル

ソース・ファイルを作成してから、 SQL ステートメントが入っているそれぞれのホスト言語ファイルを、ホスト言語ソース・ファイル用の `PREP` コマンドを使ってプリコンパイルする必要があります。プリコンパイラーはソース・ファイルに含まれている SQL ステートメントを注釈に変換し、そのステートメントについて DB2 実行時 API 呼び出しを生成します。

アプリケーションをプリコンパイルする前に、明示的または暗黙に、サーバーに接続する必要があります。アプリケーション・プログラムをクライアント・ワークステーションでプリコンパイルして、プリコンパイラーが変更されたソースとメッセージをクライアント上で生成しても、プリコンパイラーはサーバー接続を使用していくらかの妥当性検査を実行します。

さらに、プリコンパイラーは、データベース・マネージャーがデータベースに対する SQL ステートメントを処理する上で必要な情報も作成します。この情報は、選択したプリコンパイラー・オプションによって、パッケージ、バインド・ファイル、またはその両方に保管されます。

プリコンパイラーの使用の一般的な例を以下に示します。 *filename.sqc* という C 組み込み SQL ソース・ファイルをプリコンパイルするために、以下のコマンドを実行して、デフォルト名 `filename.c` の C ソース・ファイルと、デフォルト名 `filename.bnd` のバインド・ファイルを作成することができます。

```
DB2 PREP filename.sqc BINDFILE
```

プリコンパイラー構文およびオプションの詳細については、 *コマンド解説書* を参照してください。

プリコンパイラーは、最大で 4 タイプまでの出力を生成します。

- 修正済みソース
- パッケージ
- バインド・ファイル
- メッセージ・ファイル

修正済みソース
このファイルは、プリコンパイラーが SQL ステートメントを DB2 実行時 API 呼び出しに変換した後の、元のソース・ファイルの新バージョンです。適切なホスト言語拡張子が与えられます。

パッケージ
PACKAGE オプション (デフォルト) が使用されている場合、または BINDFILE、SYNTAX、SQLFLAG オプションのいずれも指定されていない場合は、パッケージは接続しているデータベースに保管されます。このパッケージには、このデータベースだけに対して特定のソース・ファイルの静的 SQL ステートメントを実行するために必要なすべての情報が入ります。 PACKAGE USING オプションを使って別の名前を指定した場合以外は、プリコンパイラーはパッケージ名を、ソース・ファイル名の最初の 8 文字から作成します。

PACKAGE オプションを使用する場合、プリコンパイル処理中に使用するデータベースには、ソース・ファイル内の静的 SQL が参照するデータベース・オブジェクトがすべて含まれていなければなりません。たとえば、SELECT ステートメントは、参照する表がデータベースに入っていない限り、プリコンパイルすることはできません。

バインド・ファイル
BINDFILE オプションを使用すると、プリコンパイラーはパッケージの作成に必要なデータを含むバインド・ファイル (拡張子は `.bnd`) を作成します。このファイルは後に `BIND` コマンドとともに使用して、1 つまたは複数のデータベースにバインドすることができます。 BINDFILE を指定しているが PACKAGE オプションを指定していない場合、 `BIND` コマンドを呼び出さない限りバインドは行われません。コマンド行プロセッサー (CLP) の場合は、 `PREP` のデフォルトは BINDFILE オプションを指定しません。したがって、CLP の使用中にバインドを延期したいときは、 BINDFILE オプションを指定する必要があります。

PACKAGE を指定せずに、プリコンパイル時にバインド・ファイルを要求すると、パッケージは作成されず、特定のオブジェクトの存在や許可 SQLCODE はエラーではなく警告として扱われます。これにより、参照するオブジェクトが存在しない場合や、プリコンパイルする SQL ステートメントを実行するための許可を持たない場合でも、プロ

グラムをコンパイルしてバインド・ファイルを作成できます。エラーではなく警告として扱われる特定の SQLCODE のリストについては、 コマンド*解説書* を参照してください。

メッセージ・ファイル

MESSAGES オプションを使用している場合、プリコンパイラーはメッセージを指定のファイルに転送します。このメッセージには、警告およびプリコンパイル中に発生した問題を示しているエラー・メッセージが含まれます。ソース・ファイルが正常にプリコンパイルされない場合は、警告およびエラー・メッセージを使用して問題を判別し、ソース・ファイルを訂正してから、再度プリコンパイルしてみてください。 MESSAGES オプションを使用していない場合は、プリコンパイルのメッセージは標準出力に書き込まれます。

ソース・ファイル要件

ソース・ファイルは必ず、特定のデータベースに対してプリコンパイルしなければなりません。そのデータベースは、最終的にそのアプリケーションとともに使用されることのないデータベースであってもかまいません。実際に、テスト・データベースを開発用に使用することができます。そして、アプリケーションを十分にテストしてから、バインド・ファイルを 1 つまたは複数の製品データベースにバインドすることができます。この機能を使用する別の方法については、61ページの『バインドの実行据え置きの利点』を参照してください。

アプリケーションが使用しているコード・ページがデータベースのコード・ページと異なる場合、プリコンパイル時にどのコード・ページを使用するのかを考慮する必要があります。 529ページの『異なるコード・ページ間での変換』を参照してください。

アプリケーションでユーザー定義関数 (UDF) またはユーザー定義特殊タイプ (UDT) を使用している場合は、アプリケーションをコンパイルする際に FUNCPATH オプションを使用する必要があります。このオプションは、静的 SQL を含むアプリケーションで UDF と UDT を使用できるようにするための関数パスを指定します。 FUNCPATH を指定しない場合、デフォルトの関数パスは *SYSIBM*、*SYSFUN*、*USER* になります (*"USER"* は現行のユーザー ID のことです)。バインド・オプションの詳細については、コマンド*解説書* を参照してください。

複数のサーバーにアクセスするアプリケーション・プログラムをプリコンパイルするには、以下のいずれかを行ってください。

- データベースごとに SQL ステートメントを個別のソース・ファイルに分割する。異なるデータベースのための SQL ステートメントを同じファイルに混合しないでください。それぞれのソース・ファイルを、適切なデータベースに対してプリコンパイルすることができます。この方法で行うことをお勧めします。
- 動的 SQL のみを使用してアプリケーションをコーディングし、プログラムがアクセスするそれぞれのデータベースにバインドする。

- すべてのデータベースが同じであると思われる場合、つまり定義が同じである場合は、 SQL ステートメントを 1 つのソース・ファイルにグループ化することができる。

アプリケーションが DB2 コネクトを介して AS/400 アプリケーション・サーバーへアクセスするとしても、同じプロシージャーが適用されます。サーバーで使用可能な PREP オプションを使用して、接続する予定のサーバーに対してアプリケーションをプリコンパイルしてください。

DB2 ユニバーサル・データベース (OS/390 版) で稼働するアプリケーションをプリコンパイルしている場合、 SQL ステートメントの構文の検査に標識機能を使用することを考慮してください。標識機能は、DB2 ユニバーサル・データベースでサポートされていても、 DB2 ユニバーサル・データベース (OS/390 版) ではサポートされていない SQL 構文を示します。また標識機能を使用して、作成した SQL 構文が SQL92 Entry Level 構文に従っているかも検査できます。 PREP コマンドの SQLFLAG オプションを使用して、標識機能を呼び出し、比較する DB2 ユニバーサル・データベース (OS/390 版) の SQL 構文のバージョンを指定できます。標識機能を使用しても、必ずしも SQL の使用を変更する必要はありません。これは、構文の不適合に関する通知または警告メッセージを出すだけで、プリプロセスが異常終了することはありません。

PREP コマンドの詳細については、 *コマンド解説書* を参照してください。

コンパイルとリンク

修正済みソース・ファイルと、SQL ステートメントが含まれていない追加ソース・ファイルを、適切なホスト言語コンパイラーでコンパイルしてください。言語コンパイラーは、それぞれの修正済みソース・ファイルを*オブジェクト・モジュール* に変換します。

デフォルトのコンパイル・オプションに対するいくつかの例外に関しては、 *アプリケーション構築の手引き*、またはご使用のオペレーティング・プラットフォーム用のプログラミング資料などを参照してください。使用可能なコンパイル・オプションの完全な説明については、コンパイラーの資料を参照してください。

ホスト言語リンカーは実行可能アプリケーションを作成します。以下に例を示します。
- OS/2 および Windows 32 ビット・オペレーティング・システム上では、アプリケーションは実行可能ファイルまたはダイナミック・リンク・ライブラリー (DLL) にすることができます。
- UNIX ベースのシステム上では、アプリケーションは実行可能ロード・モジュールまたは共用ライブラリーにすることができます。

注: Windows 32 ビット・オペレーティング・システム上ではアプリケーションを DLL にすることはできますが、 DLL は DB2 データベース・マネージャーではなく、アプリケーションにより直接ロードされます。 Windows 32 ビット・オペレーティング・システム上で、データベース・マネージャーが DLL をロードすることも可能です。ストアード・プロシージャーは、通常は DLL または共用ライブラリーと

して作成されます。ストアード・プロシージャーの使用については、 201ページの『第7章 ストアード・プロシージャー』を参照してください。

DB2 でサポートされているその他のプラットフォームで実行可能ファイルを作成する方法については、 *アプリケーション構築の手引き* を参照してください。

実行可能ファイルを作成するには、以下のものにリンクします。

- ユーザー・オブジェクト・モジュール。修正済みソース・ファイル、および SQL ステートメントが入っていないその他のファイルから、言語コンパイラーによって生成されます。
- ホスト言語ライブラリー API。言語コンパイラーによって提供されます。
- データベース・マネージャー・ライブラリー。ご使用のオペレーティング環境用のデータベース・マネージャー API が入っています。使用するデータベース・マネージャー API に必要なデータベース・マネージャー・ライブラリーの名前については、 *アプリケーション構築の手引き* またはご使用のオペレーティング・プラットフォーム用のプログラミング資料などを参照してください。

バインド

バインドとは、データベース・マネージャーがアプリケーションの実行時にデータベースをアクセスするために必要とするパッケージを作成する処理です。バインドは、プリコンパイル中に PACKAGE オプションを指定して暗黙に行うことも、プリコンパイル中に作成されたバインド・ファイルに対して `BIND` コマンドを使用することにより明示的に行うこともできます。

`BIND` コマンドの使用の一般的な例を以下に示します。 *filename.bnd* という名前のバインド・ファイルをデータベースにバインドするには、以下のコマンドを実行することができます。

```
DB2 BIND filename.bnd
```

`BIND` コマンドの構文およびオプションの詳細については、 *コマンド解説書* を参照してください。

個別にプリコンパイルされたソース・コード・モジュールごとに 1 つのパッケージが作成されます。アプリケーションに 5 つのソース・ファイルがあって、そのうちの 3 ファイルにプリコンパイルが必要な場合、 3 つのパッケージまたはバインド・ファイルが作成されます。デフォルトの解釈では、それぞれのパッケージには `.bnd` ファイルの作成元となったソース・モジュールの名前と同じ名前が付けられますが、8 文字で切り捨てられます。新しく作成されたパッケージの名前が宛先データベースに現在存在しているパッケージの名前と同じ場合、既存パッケージに代わって新規のパッケージが使用されます。異なるパッケージ名を明示的に指定するには、 `PREP` コマンドの PACKAGE USING オプションを使用しなければなりません。詳細については、*コマンド解説書* を参照してください。

パッケージの名前変更

アプリケーションのバージョンを複数作成する場合、パッケージの名前を変更して、名前が重複するのを避けてください。たとえば、foo というアプリケーション (foo.sqc からコンパイルされたもの) がある場合、これをプリコンパイルして、アプリケーションの全ユーザーに送ります。ユーザーはアプリケーションをデータベースにバインドして、これを実行します。変更が必要な場合には、続けて foo の新しいバージョンを作成し、このアプリケーションとそのバインド・ファイルを、新しいバージョンを必要とするユーザーに送ります。新しいバージョンのユーザーは foo.bnd をバインドし、新しいアプリケーションは問題なく実行されます。ただし、ユーザーが以前のバージョンのアプリケーションを使おうとすると、 FOO パッケージ上のタイム・スタンプに矛盾が生じ (データベース内のパッケージが実行中のアプリケーションと一致しないことになります)、クライアントを再バインドしなければならなくなります。 (パッケージのタイム・スタンプの詳細については、 63ページの『タイム・スタンプ』を参照してください。) 新しいアプリケーションのユーザーは、タイム・スタンプが矛盾していることを通知されます。これは、両方のアプリケーションに同じ名前が使用されているために発生する問題です。

この問題を解決するには、パッケージの名前を変更します。 FOO の最初のバージョンの作成時に、以下のコマンドを使用してこれをプリコンパイルします。

```
DB2 PREP FOO.SQC BINDFILE PACKAGE USING FOO1
```

このアプリケーションが配布されると、ユーザーはこのアプリケーションを問題なくバインドおよび実行できます。新しいバージョンを作成する際には、以下のコマンドを使用してこれをプリコンパイルします。

```
DB2 PREP FOO.SQC BINDFILE PACKAGE USING FOO2
```

このアプリケーションも、配布後に問題なくバインドおよび実行されます。最初のバージョンのパッケージ名は FOO1 で、新しいバージョンは FOO2 になるので名前は重複せず、どのバージョンのアプリケーションも使用できます。

動的ステートメントのバインド

動的に作成されたステートメントについては、特殊レジスターの数によってステートメントのコンパイル環境が決定されます。

- CURRENT QUERY OPTIMIZATION 特殊レジスターは、使用される最適化クラスを決定します。
- CURRENT FUNCTION PATH 特殊レジスターは、 UDF および UDT の解決に使用される関数パスを決定します。
- CURRENT EXPLAIN SNAPSHOT レジスターは、 Explain スナップショット情報を収集するかどうかを決定します。
- CURRENT EXPLAIN MODE レジスターは、適格な動的 SQL ステートメントについての Explain 表情報を収集するかどうかを決定します。これらの特殊レジスターのデ

フォルトは、関連するバインド・オプションで使用されるデフォルトと同じです。特殊レジスターおよび BIND オプションとのレジスターの対話については、 *SQL 解説書* を参照してください。

非修飾表名の解決

以下の方法の一つを使用すると、アプリケーション内で非修飾表名を処理することができます。

- 各ユーザーごとに、以下のコマンドを用いて異なる許可 ID からのさまざまな COLLECTION パラメーターでパッケージをバインドします。

```
CONNECT TO db_name USER user_name
BIND file_name COLLECTION schema_name
```

  上記の例では、*db_name* はデータベース名、*user_name* はユーザー名、そして *file_name* はバインドされるアプリケーション名を表しています。 *user_name* と *schema_name* は普通は同じ値です。その後、SET CURRENT PACKAGESET ステートメントを使用して、使用するパッケージ、すなわち使用する修飾子を指定します。そのデフォルト修飾子が、パッケージをバインドするときに使用する許可 ID になります。 SET CURRENT PACKAGESET ステートメントの使用方法の例は、 *SQL 解説書* を参照してください。
- 各ユーザーに非修飾表名と同じ名前の視点を作成して、非修飾表名が正しく解決されるようにします。 (QUALIFIER オプションはあくまでも DB2 コネクト専用であること、つまりホスト・サーバー使用時に限って使用できることに注意してください。)
- ユーザーごとに一つの別名を作成し、希望する表を指すようにします。

その他のバインドの考慮事項

アプリケーションのコード・ページがデータベースのコード・ページと異なる場合、バインド時にどちらのコード・ページを使用するかを考慮する必要があります。 529ページの『異なるコード・ページ間での変換』を参照してください。

アプリケーションが IMPORT または EXPORT のようなデータベース・マネージャー・ユーティリティー API に呼び出しを発行する場合、提供されるユーティリティー・バインド・ファイルをデータベースにバインドしておくことが必要です。詳細については、ご使用のプラットフォームの*概説およびインストール* を参照してください。

バインド・オプションを使用して、バインド中に発生する一定の操作を制御することができます。

- QUERYOPT バインド・オプションは、バインド時に特定の最適化クラスを利用します。
- EXPLSNAP バインド・オプションは、 Explain 表内の適格な SQL ステートメント用の Explain スナップショット情報を保管します。
- FUNCPATH バインド・オプションは、静的 SQL のユーザー定義特殊タイプおよびユーザー定義関数を正しく解決します。

バインド・オプションについては、 *コマンド解説書* の BIND コマンドの節を参照してください。

バインド処理を開始しても応答がない場合、データベースに接続している他のアプリケーションが、必要なロックを保持している可能性があります。この場合には、どのアプリケーションもデータベースに接続されていないことを確認してください。接続されていることがわかったなら、サーバー上のすべてのアプリケーションを切り離して、バインド処理を継続します。

アプリケーションが DB2 コネクトを使用してサーバーにアクセスする場合、そのサーバーで使用可能な BIND オプションを使用できます。 BIND コマンドとそのオプションに関する詳細については、 *コマンド解説書* を参照してください。

バインド・ファイルは、以前の DB2 ユニバーサル・データベースのバージョンとの下位互換性はありません。レベルが混合した環境では、 DB2 は最下位レベルのデータベース環境で使用できる機能しか使用できません。たとえば、V5.2 クライアントが V5.0 サーバーと接続している場合、そのクライアントは V5.0 の機能しか使用できません。バインド・ファイルはデータベースの機能を伝えるので、それらは混合レベルの制限に従います。

下位レベルのシステムで、上位レベルのバインド・ファイルを再バインドする必要がある場合は、以下のようにすることができます。

- 上位レベルのサーバーに接続するために下位レベルの DB2 アプリケーション開発クライアントを使用して、下位レベルの DB2 ユニバーサル・データベース環境に搬出してバインドできるバインド・ファイルを作成します。
- 上位レベルの DB2 クライアントを下位レベルの実稼働環境で使用して、テスト環境で作成された上位レベルのバインド・ファイルをバインドします。上位レベルのクライアントは、下位レベルのサーバーに適用されるオプションだけを渡します。

バインドの実行据え置きの利点

バインドを実行可能にしてプリコンパイルを行うと、アプリケーションはプリコンパイル処理中に使用されたデータベースだけにアクセスできます。バインドを実行据え置きにしてプリコンパイルすると、 BIND ファイルを各データベースにバインドできるので、多数のデータベースにアクセスできます。このアプリケーション開発方法は、アプリケーションを 1 回だけプリコンパイルするという点で本来柔軟性のあるものですが、アプリケーションはデータベースにいつでもバインドすることができます。

実行中に BIND API を使用すると、アプリケーションはそれ自体をバインドすることができます。これはおそらくインストール手順の一部として、または関連モジュールが実行される前に行われます。たとえば、アプリケーションは複数のタスクを実行することができますが、そのうちで SQL ステートメントを使用する必要があるのは 1 つだけで

す。アプリケーションは、SQL ステートメントを必要とするタスクが呼び出されるとき、および関連パッケージが存在しない場合にだけ、それ自体をデータベースにバインドできるように設計することができます。

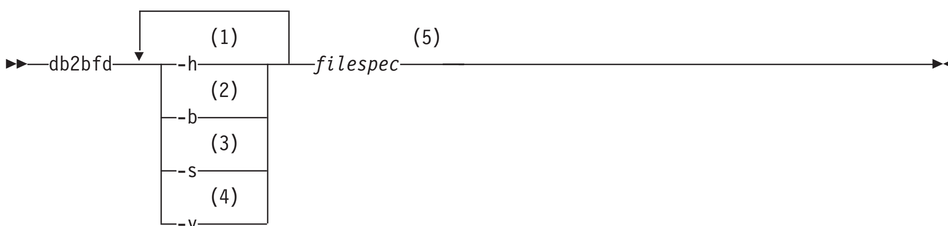
バインドの実行据え置きのもう 1 つの利点は、エンド・ユーザーにソース・コードを提供しなくてもパッケージを作成できるという点です。関連したバインド・ファイルをアプリケーションと共に搬出することができます。

DB2 バインド・ファイル記述ユーティリティー - db2bfd

DB2 バインド・ファイル記述 (db2bfd) ユーティリティーを使用することにより、バインド・ファイルを作成するのに使用するプリコンパイル・オプションを表示するだけでなく、バインド・ファイルの内容を簡単に表示して、ファイル内部の SQL ステートメントを調査および検証することができます。これは、アプリケーションのバインド・ファイルに関連する問題の判別に役立ちます。

db2bfd ユーティリティーは、インスタンスの sqllib ディレクトリー内にある bin サブディレクトリーに入っています。

構文は以下のとおりです。



注:

1 ヘルプ情報を表示

2 バインド・ファイル・ヘッダーを表示

3 SQL ステートメントを表示

4 ホスト変数宣言を表示

5 バインド・ファイルの名前

db2bfd の詳細については、*コマンド解説書* を参照してください。

アプリケーション、バインド・ファイル、およびパッケージの関係

パッケージはデータベースに保管されたオブジェクトで、単一のソース・ファイル内の特定の SQL ステートメントを実行するために必要な情報を含んでいます。データベース・アプリケーションは、そのアプリケーションを作成するのに使用するプリコンパイ

ルされたすべてのソース・ファイルに対して、 1 つのパッケージを用います。それぞれのパッケージは個別のエンティティーであり、同じアプリケーションまたは他のアプリケーションで使用される別のパッケージとは関係ありません。パッケージを作成するには、バインドを実行可能にしてソース・ファイルに対してプリコンパイラーを実行するか、 1 つまたは複数のバインド・ファイルで後からバインダーを実行します。

データベース・アプリケーションはパフォーマンスの向上とサイズを小さくするためにパッケージを使用しますが、これはアプリケーションをコンパイルする理由と同じものです。 SQL ステートメントをプリコンパイルすることによって、このステートメントはアプリケーションの実行時ではなく作成時にコンパイルされてパッケージとなります。それぞれのステートメントが構文解析され、さらに効率的に解釈されたオペランド・ストリングがパッケージに保管されます。実行時に、プリコンパイラーにより生成されるコードにより、入出力データに必要な変数情報を指定したランタイム・サービス・データベース・マネージャー API が呼び出され、パッケージに保管されている情報が実行されます。

プリコンパイルの利点は静的 SQL ステートメントにだけ当てはまります。動的に実行される SQL ステートメント (PREPARE と、EXECUTE か EXECUTE IMMEDIATE を用いる) はプリコンパイルされません。したがって、一連の処理手順全体を実行時に行わなければなりません。

注: SQL ステートメントの静的 SQL は、動的に処理される同じステートメントよりも自動的に速く実行されるとは考えないでください。動的ステートメントの準備にはオーバーヘッドが必要なため、静的 SQL の方が速く実行されます。それ以外の場合は、最適化プログラムがバインド時以前に使用可能なデータベース統計ではなく、現行のデータベース統計を使用できるので、動的に作成された同じステートメントのほうが速く実行されます。トランザクションの完了にかかる時間が 2 秒未満である場合は、一般に静的 SQL の方が速くなります。使用する方式を選択するために、両方のバインド方式をプロトタイプ化してください。静的および動的 SQL の詳細な比較は、134ページの『動的 SQL と静的 SQL との比較』を参照してください。

タイム・スタンプ

パッケージまたはバインド・ファイルを生成すると、プリコンパイラーはタイム・スタンプを生成します。タイム・スタンプはバインド・ファイルまたはパッケージ、および修正済みソース・ファイルに保管されます。

バインドを実行可能にしてアプリケーションをプリコンパイルすると、タイム・スタンプが一致するパッケージと修正済みソース・ファイルが生成されます。アプリケーションを実行する際に、タイム・スタンプが等しいかどうか検査されます。アプリケーションを実行するには、アプリケーションとその関連パッケージのタイム・スタンプが一致していなければなりません。そうしないと、SQL0818N エラーがアプリケーションに戻されます。

アプリケーションをデータベースにバインドする場合、 *PREP コマンドの PACKAGE USING オプションを使用してデフォルトを指定変更しない限り*、アプリケーション名の最初の 8 文字がパッケージ名として使用されることを覚えておいてください。つまり、同じ名前の 2 つのプログラムをプリコンパイルしてバインドすると、 2 番目のプログラムが最初のパッケージを上書きします。最初のプログラムを実行すると、そのプログラムでは修正済みソース・ファイルのタイム・スタンプとデータベースのパッケージのタイム・スタンプとが一致していないため、タイム・スタンプ・エラーになります。

バインドを実行据え置きにしてアプリケーションをプリコンパイルすると、タイム・スタンプが一致する 1 つまたは複数のバインド・ファイルおよび修正済みソース・ファイルが生成されます。このアプリケーションを実行するには、アプリケーション・モジュールにより作成されるバインド・ファイルを実行します。バインド処理は、58ページの『バインド』で説明しているように、バインド・ファイルごとに行わなければなりません。

バインド・ファイルには、プリコンパイル中に修正済みソース・ファイルに保管されたタイム・スタンプと同じものが入っているため、アプリケーションとパッケージのタイム・スタンプは一致します。

再バインド

再バインド は、以前にバインドされたアプリケーション・プログラムのパッケージを作成する処理です。パッケージが無効、または作動不能と示された場合は、これを再バインドしなければなりません。しかし、パッケージが有効であっても再バインドが必要な場合もあります。たとえば、新規に作成された索引を利用する場合、または RUNSTATS コマンドの実行後の更新統計を利用する場合です。

パッケージは、表、視点、別名、索引、トリガー、参照制約、および表検査の制約など、データベース・オブジェクトの一定のタイプに従属させることができます。パッケージがデータベース・オブジェクト (表、視点、トリガーなど) に従属している場合にそのオブジェクトが除去されると、パッケージは*無効* な状態になります。除去されたオブジェクトが UDF である場合、パッケージは*作動不能* 状態になります。詳細については、*管理の手引き: 計画* を参照してください。

無効なパッケージは、実行される際にデータベース・マネージャーによって暗黙に (つまり自動的に) 再バインドされます。作動不能パッケージは、 `BIND` コマンドまたは `REBIND` コマンドのいずれかを実行して、明示的に再バインドしなければなりません。暗黙の再バインドに失敗すると、予期しないエラーが生じる場合があることに気を付けてください。つまり、暗黙の再バインドのエラーは、実際にエラーのあるステートメントではなく実行中のステートメントに戻される場合もあります。作動不能パッケージを実行しようとすると、エラーが発生します。無効なパッケージをシステムで自動的に再バインドするのではなく、これらを明示的に再バインドすることができます。これにより、いつ再バインドを行うかを制御できるようになります。

パッケージを明示的にバインドするために使用するコマンドは、状況により異なります。 SQL ステートメントの数を変更するか、または変更された SQL ステートメントを含めるために修正されたプログラムのパッケージを再バインドするには、 BIND コマンドを使用しなければなりません。パッケージが最初にバインドされた変数からバインド・オプションを変更する必要がある場合にも、 BIND コマンドを使用します。その他の場合には、 BIND コマンドと REBIND コマンドのいずれかを使用してください。パフォーマンスの面では BIND よりも REBIND の方が優れているので、特に BIND を使用する必要がないときは REBIND を使用してください。

REBIND コマンドの詳細については、 *コマンド解説書* を参照してください。

第4章 静的 SQL プログラムの作成

静的 SQL を使用する場合の特性とそれを使用する理由 68
静的 SQL の利点 68
例: 静的 SQL プログラム 69
静的プログラムの動作の仕組み. 70
C の例: STATIC.SQC 72
Java の例: Static.sqlj 73
COBOL の例: STATIC.SQB 75
データの検索および操作のための SQL ステートメントのコーディング 77
データの検索. 77
ホスト変数の使用 77
宣言生成プログラム - db2dclgn 80
標識変数の使用 82
データ・タイプ 84
STATIC プログラムでの標識変数の使用. . 86
カーソルを用いた複数行の選択. 87
カーソルの宣言と使用. 87
カーソルおよび作業単位に関する考慮事項 88
読み取り専用カーソル. 88
WITH HOLD オプション 88
例: カーソル・プログラム 90
カーソル・プログラムの動作の仕組み 90
C の例: CURSOR.SQC 92
Java の例: Cursor.sqlj 94
COBOL の例: CURSOR.SQB 96
検索済みデータの更新と削除 98
検索されたデータの更新 98
検索されたデータの削除 98
カーソルのタイプ 98
例: OPENFTCH プログラム 99
OPENFTCH プログラムの動作の仕組み 99
C の例: OPENFTCH.SQC 101
Java の例: Openftch.sqlj 103
COBOL の例: OPENFTCH.SQB . . . 106
高度なスクロール技法 108
すでに検索済みデータのスクロール . . . 108
データのコピーを保持する方法 108
データを 2 度検索する方法 109
先頭からの検索 109
中間からの検索 109
2 番目の結果表内での行の順序 . . . 109
逆順での検索 110
表の末尾の位置の確立 111
以前に検索されたデータの更新 111
例: UPDAT プログラム 111
UPDAT プログラムの動作の仕組み 112
C の例: UPDAT.SQC. 114
Java の例: Updat.sqlj 116
COBOL の例: UPDAT.SQB. 118
REXX の例: UPDAT.CMD 120
診断処理と SQLCA 構造 122
戻りコード 122
SQLCODE および SQLSTATE 122
SQLCA 構造におけるトークンの切り捨て 123
WHENEVER ステートメントを用いたエラー処理 123
例外、シグナル、割り込みハンドラーについての考慮事項 124
出口リスト・ルーチンに関する考慮事項 125
プログラム例での GET ERROR MESSAGE の使用. 125
C の例: UTILAPI.C 126
Java の例: SQLException のキャッチ 128
COBOL の例: CHECKERR.CBL . . . 129
REXX の例: CHECKERR プロシージャー 131

© Copyright IBM Corp. 1993, 2001

静的 SQL を使用する場合の特性とそれを使用する理由

組み込み SQL ステートメントの構文がプリコンパイル時に完全に認識されている場合、そのステートメントは*静的* SQL と呼ばれます。静的 SQL は、実行時まで構文が認識されない*動的* SQL ステートメントとは対照的です。

注: 静的 SQL は、REXX などのインタープリター言語ではサポートされません。

ステートメントが静的と見なされるためには、 SQL ステートメントの構造を完全に指定しなければなりません。たとえば、ステートメントで参照される列または表の名前は、プリコンパイル時に完全に認識されている必要があります。実行時に指定できる唯一の情報は、ステートメントが参照するホスト変数の値だけです。ただし、データ・タイプなどのホスト変数情報は、プリコンパイルしなければなりません。

静的 SQL ステートメントを準備する際に、ステートメントの実行可能形式が作成され、データベースのパッケージに保管されます。実行可能形式はプリコンパイル時か、後のバインド時に構成することができます。どちらの場合も、実行時より*前に* 準備が行われます。アプリケーションのバインドを行うユーザーの権限が使用されて、データベース統計および構成パラメーター (アプリケーション実行時に最新ではない場合がある) に基づいて最適化が行われます。

静的 SQL の利点

静的 SQL を用いたプログラミングは、組み込み動的 SQL を用いたプログラミングよりも簡単です。静的 SQL ステートメントはホスト言語ソース・ファイルに簡単に組み込まれ、プリコンパイラーは必要に応じて、ホスト言語コンパイラーが処理できるデータベース・マネージャーランタイム・サービス API 呼び出しへの変換を処理します。

アプリケーションのバインドを行うユーザーの許可が使用されているため、エンド・ユーザーはパッケージでステートメントを実行するための直接的な特権は不要です。たとえば、表全体についての更新特権がないユーザーでも、表の一部を更新できるようにすることが可能です。これを行うには、特定の列または値の範囲だけを更新できるように静的 SQL ステートメントを制限します。

静的 SQL ステートメントには*持続性* があります。つまりこれは、ステートメントはパッケージが存在する限り続くという意味です。 動的 SQL ステートメントは、スペース管理の理由で無効にされるか解放されるまで、またはデータベースが遮断されるまでキャッシュされます。必要であれば、動的 SQL ステートメントはキャッシュされたステートメントが無効になった時点で、 DB2 SQL コンパイラーにより暗黙に再コンパイルされます。キャッシュおよびキャッシュされたステートメントが無効となる原因については、 *SQL 解説書* を参照してください。

持続性に関する静的 SQL の重要な利点は、静的ステートメントはある特定のデータベースが遮断した後でも存在するという点です。一方、動的 SQL ステートメントはデー

タベースが遮断すると失われてしまいます。さらに、静的 SQL は実行時に DB2 SQL コンパイラーでコンパイルする必要はありませんが、動的 SQL は実行時に明示的に (PREPARE ステートメントなどを使用して) コンパイルしなければなりません。 DB2 は動的 SQL ステートメントをキャッシュするため、動的ステートメントを頻繁に DB2 でコンパイルする必要はありませんが、アプリケーション実行時に少なくとも 1 回はコンパイルしなければなりません。

静的 SQL にはパフォーマンス上の利点もあります。簡単に言うと、SQL プログラムの実行時間が短いという点です。すなわち、ステートメントの実行可能形式を準備するオーバーヘッドは、実行時ではなくプリコンパイル時に行われるため、静的 SQL ステートメントは動的に処理された同じステートメントよりも迅速に実行されます。

注: 静的 SQL のパフォーマンスは、アプリケーションが最後にバインドされた時のデータベースの統計によって決まります。一方、この統計が変化すると、対応する動的 SQL のパフォーマンスは非常に異なってくる可能性があります。たとえば、後でデータベースに索引を追加する場合、静的 SQL を用いたアプリケーションは、データベースにバインドし直さなければその索引を利用することはできません。また、静的 SQL ステートメントでホスト変数を使用する場合、最適化プログラムは表の分散統計を活用することができません。

例: 静的 SQL プログラム

このサンプル・プログラムでは、静的 SQL ステートメントおよび以下のサポートされる言語でのデータベース・マネージャー API の例を説明しています。

C static.sqc

Java Static.sqlj

COBOL static.sqb

REXX 言語は静的 SQL をサポートしないため、サンプルはありません。

このサンプル・プログラムには、単一行を選択する照会が含まれます。このタイプの照会は SELECT INTO ステートメントを用いて実行することができます。

SELECT INTO ステートメントはデータベース内の複数の表から一行のデータを選択し、そしてその値をステートメントで指定したホスト変数に割り当てます。ホスト変数については、77ページの『ホスト変数の使用』で詳しく説明します。たとえば次のステートメントは、姓が 'HAAS' の従業員の給料をホスト変数 empsal に渡します。

```
SELECT SALARY
  INTO :empsal
  FROM EMPLOYEE
  WHERE LASTNAME='HAAS'
```

SELECT INTO ステートメントには 1 行だけ戻すように、または行を戻さないように指定する必要があります。行が複数検出されると、SQLCODE -811 (SQLSTATE 21000) エラーが起こります。照会の結果として複数行が検出されたら、カーソルを使用してそれらの行を処理する必要があります。詳細については、87ページの『カーソルを用いた複数行の選択』を参照してください。

SELECT INTO ステートメントの詳細については、 *SQL 解説書* を参照してください。

SELECT ステートメントの記述方法の初歩的な説明については、 77ページの『データの検索および操作のための SQL ステートメントのコーディング』を参照してください。

静的プログラムの動作の仕組み

1. **SQLCA を組み込む。** INCLUDE SQLCA ステートメントは SQLCA 構造を定義し宣言します。また、その構造内のエレメントとして `SQLCODE` と `SQLSTATE` を定義します。 SQLCA 構造の SQLCODE フィールドは、 SQL ステートメントまたはデータベース・マネージャー API 呼び出しを実行するたびにデータベース・マネージャーによって診断情報を用いて更新されます。
2. **ホスト変数を宣言する。**ホスト変数は、 SQL BEGIN DECLARE SECTION および END DECLARE SECTION ステートメントによって区切られます。これらの変数は SQL ステートメントで参照できるものです。ホスト変数は、データをデータベース・マネージャーに渡したり、データベース・マネージャーが戻したデータを保持するために使用されます。ホスト変数には、SQL ステートメントで参照される際に、接頭部としてコロン (:) が付けられます。詳細については、77ページの『ホスト変数の使用』を参照してください。
3. **データベースに接続する。**プログラムは `sample` データベースに接続し、そのデータベースへの共用アクセスを要求します。 (START DATABASE MANAGER API 呼び出しまたは `db2start` コマンドが実行されていることを前提としています。) 共用アクセスを用いて同じデータベースに接続する他のプログラムにもアクセスが許可されます。
4. **データを取り出す。** SELECT INTO ステートメントは、照会に基づいて 1 つの値を取り出します。この例では、 `LASTNAME` 列の値が `JOHNSON` である `EMPLOYEE` 表から、 `FIRSTNME` 列を取り出します。 `SYBIL` という値が戻され、ホスト変数 `firstname` に入れられます。 DB2 で用意されているサンプル表は、 *SQL 解説書* の付録にリストされています。
5. **エラーを処理する。** `CHECKERR` マクロ / 関数は、プログラム外部にあるエラー検査ユーティリティーです。エラー検査ユーティリティーの所在は、ご使用のプログラム言語により異なります。

   C DB2 API を呼び出す C プログラムの場合、 utilapi.c 内の `sqlInfoPrint` 関数は、 utilapi.h 内の `API_SQL_CHECK` として再定

義されます。 C 組み込み SQL プログラムの場合、 utilemb.sqc 内の sqlInfoPrint 関数は、 utilemb.h 内の EMB_SQL_CHECK として再定義されます。

Java SQL エラーは SQLException としてスローされ、アプリケーションの catch ブロックで処理されます。

COBOL CHECKERR は checkerr.cbl という名前の外部プログラムです。

このエラー検査ユーティリティーのソース・コードについては、 125ページの『プログラム例での GET ERROR MESSAGE の使用』を参照してください。

6. **データベースから切断する。**プログラムは CONNECT RESET ステートメントを実行して、データベースから切断します。プログラムが戻ると、 SQLJ プログラムはデータベースの接続を自動的にクローズすることに注意してください。

C の例: STATIC.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;   1

int main(int argc, char *argv[])
{   int        rc = 0;

    char dbAlias[15] ;
    char user[15] ;
    char pswd[15] ;

    EXEC SQL BEGIN DECLARE SECTION;  2
        char firstname[13];
    EXEC SQL END DECLARE SECTION;

    /* checks the command line arguments */
    rc = CmdLineArgsCheck1( argc, argv, dbAlias, user, pswd );  3
    if ( rc != 0 ) return( rc ) ;

    printf("¥n¥nSample C program: STATIC¥n");

    /* initialize the embedded application */
    rc = EmbAppInit( dbAlias, user, pswd);
    if ( rc != 0 ) return( rc ) ;

    EXEC SQL SELECT FIRSTNME INTO :firstname   4
            FROM employee
            WHERE LASTNAME = 'JOHNSON';
    EMB_SQL_CHECK("SELECT statement");   5

    printf( "First name = %s¥n", firstname );

    /* terminate the embedded application */
    rc = EmbAppTerm( dbAlias);
    return( rc ) ;
}
/* end of program : STATIC.SQC */
```

Java の例: Static.sqlj

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

class Static
{   static
  {   try
    {   Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
      }
      catch (Exception e)
      {   System.out.println ("¥n  Error loading DB2 Driver...¥n");
        System.out.println (e);
        System.exit(1);
      }
  }

  public static void main(String argv[])
  {   try
    {   System.out.println ("  Java Static Sample");

      String url = "jdbc:db2:sample";       // URL is jdbc:db2:dbname
      Connection con = null;

      // Set the connection                    3
      if (argv.length == 0)
      {   // connect with default id/password
        con = DriverManager.getConnection(url);
      }
      else if (argv.length == 2)
      {   String userid = argv[0];
        String passwd = argv[1];

        // connect with user-provided username and password
        con = DriverManager.getConnection(url, userid, passwd);
      }
      else
      {   throw new Exception("¥nUsage: java Static [username password]¥n");
      }

      // Set the default context
      DefaultContext ctx = new DefaultContext(con);
      DefaultContext.setDefaultContext(ctx);

      String firstname = null;

      #sql { SELECT FIRSTNME INTO :firstname
           FROM employee
           WHERE LASTNAME = 'JOHNSON' } ;    4


      System.out.println ("First name = " + firstname);
    }
```

```
    catch( Exception e )  5
    {
      System.out.println (e);
    }
  }
}
```

COBOL の例: STATIC.SQB

```
Identification Division.
Program-ID. "static".

Data Division.
Working-Storage Section.

    copy "sql.cbl".
    copy "sqlca.cbl".                                                  1

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.                           2
01 firstname         pic x(12).
01 userid            pic x(8).
01 passwd.
  49 passwd-length   pic s9(4) comp-5 value 0.
  49 passwd-name     pic x(18).
    EXEC SQL END DECLARE SECTION END-EXEC.

77 errloc          pic x(80).

Procedure Division.
Main Section.
    display "Sample COBOL program: STATIC".

    display "Enter your user id (default none): "
         with no advancing.
    accept userid.

    if userid = spaces
      EXEC SQL CONNECT TO sample END-EXEC
    else
      display "Enter your password : " with no advancing
      accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR format
* with the length of the input string.
    inspect passwd-name tallying passwd-length for characters
       before initial " ".

    EXEC SQL CONNECT TO sample USER :userid USING :passwd              3
       END-EXEC.
    move "CONNECT TO" to errloc.
    call "checkerr" using SQLCA errloc.

    EXEC SQL SELECT FIRSTNME INTO :firstname                           4
             FROM EMPLOYEE
             WHERE LASTNAME = 'JOHNSON' END-EXEC.
    move "SELECT" to errloc.
    call "checkerr" using SQLCA errloc.                                5

    display "First name = ", firstname.

    EXEC SQL CONNECT RESET END-EXEC.                                   6
    move "CONNECT RESET" to errloc.
```

```
     call "checkerr" using SQLCA errloc.

End-Prog.
     stop run.
```

データの検索および操作のための SQL ステートメントのコーディング

データベース・マネージャーには、データの検索と操作に用いるステートメントが収められており、プログラミングに使用できます。このステートメントをホスト言語コードに組み込むことによってコーディングできます。この節では、コーディング・ステートメントを使って DB2 表内の複数行のデータを検索して操作する方法について説明します。 (さまざまなホスト言語について詳しくは説明しません。) SQL ステートメントの配置、連結、および区切りの正しい規則については、以下を参照してください。

- 609ページの『第20章 C および C++ でのプログラミング』
- 655ページの『第21章 Java でのプログラミング』
- 701ページの『第23章 COBOL でのプログラミング』
- 723ページの『第24章 FORTRAN でのプログラミング』
- 739ページの『第25章 REXX でのプログラミング』

データの検索

SQL アプリケーション・プログラムの最も一般的なタスクの 1 つはデータの検索です。これは、*選択ステートメント* を使用して行います。選択ステートメントは、特定の探索条件を満たすデータベース内の表の行を探索する照会の形式です。探索条件に該当する行が存在すると、そのデータは検索されてホスト・プログラムの特定の変数に入れられ、どのような使用目的にも応えることができます。

選択ステートメント のコーディングが終わったら、アプリケーションに渡される情報を定義する SQL ステートメントをコーディングします。

選択ステートメントの結果は、行と列を持つデータベース内の表であると見なすことができます。 1 行だけが戻された場合、その結果は SELECT INTO ステートメントで指定したホスト変数に直接渡されます。

複数行が戻された場合は、*カーソル* を使ってそれらの行を一度に 1 行ずつ取り出さなければなりません。カーソルは、順序付きの行ブロック内の特定の行を指し示す、アプリケーション・プログラムで用いられる名前付き制御構造です。カーソルのコーディングおよび使用方法については、以下の節を参照してください。

- 87ページの『カーソルの宣言と使用』
- 87ページの『カーソルを用いた複数行の選択』
- 90ページの『例: カーソル・プログラム』

ホスト変数の使用

ホスト変数 とは、組み込み SQL ステートメントが参照する変数です。それによりデータベース・マネージャーとアプリケーション・プログラム間でデータをやり取りします。 *SQL ステートメント* でホスト変数を使用するときは、名前の接頭部にコロン (**:**) を付けてください。*ホスト言語ステートメント* でホスト変数を使用するときは、コロンは省略してください。

ホスト変数はコンパイル済みホスト言語で宣言され、 BEGIN DECLARE SECTION および END DECLARE SECTION ステートメントで区切られます。プリコンパイラーは、これらのステートメントによって宣言を検出することができます。

注: Java JDBC および SQLJ プログラムは、宣言セクションを使用しません。 Java のホスト変数は、通常の Java 変数宣言構文に従います。

ホスト変数は、ホスト言語のサブセットを用いて宣言されます。ホスト言語にサポートされている構文については、以下を参照してください。

- 609ページの『第20章 C および C++ でのプログラミング』
- 655ページの『第21章 Java でのプログラミング』
- 701ページの『第23章 COBOL でのプログラミング』
- 723ページの『第24章 FORTRAN でのプログラミング』
- 739ページの『第25章 REXX でのプログラミング』

以下の規則は、ホスト変数宣言セクションに当てはまります。

- ホスト変数はすべてソース・ファイルで宣言してから参照しなければならない。ただし、SQLDA 構造を参照するホスト変数は例外です。
- 複数の宣言セクションを 1 つのソース・ファイルで使用する場合もある。
- プリコンパイラーはホスト言語変数の効力範囲に従わない。

  SQL ステートメントに関しては、ホスト変数がどの単一ソース・ファイルで実際に宣言されているかにかかわらず、すべてのホスト変数の効力範囲はグローバルです。したがって、ホスト変数の名前はソース・ファイル内で固有でなければなりません。

  これは、DB2 プリコンパイラーがホスト変数の効力範囲をグローバルに変更するということではなく、定義されている効力範囲外にアクセスできるようにするということです。次の例を考えてください。

```
foo1(){
  .
  .
  .
  BEGIN SQL DECLARE SECTION;
  int x;
  END SQL DECLARE SECTION;
x=10;
  .
  .
  .
}


foo2(){
  .
  .
  .
  y=x;
```

```
  .
  .
  .
}
```

言語によっては、変数 x が foo2() 関数内で宣言されていなかったり、 x の値が foo2() 内で 10 に設定されていないため、どちらの場合も上記の例ではコンパイルが失敗します。こうした問題を避けるには、グローバル変数として x を宣言するか、 x をパラメーターとして foo2() 関数に渡すかのいずれかにする必要があります。以下のようにします。

```
foo1(){
.
.
.
  BEGIN SQL DECLARE SECTION;
  int x;
  END SQL DECLARE SECTION;
  x=10;
  foo2(x);
.
.
.
}


foo2(int x){
.
.
.
  y=x;
.
.
.
}
```

ホスト変数の宣言の詳細については、以下を参照してください。

- 80ページの『宣言生成プログラム - db2dclgn』では、db2dclgn ツールを使用して、ホスト変数宣言ソース・コードを自動的に生成する方法について説明しています。
- 80ページの表4 では、ホスト変数をソース・コードに表示する方法について例をあげて説明しています。
- 81ページの表5 では、サポートされるホスト言語でホスト変数を参照する方法について例をあげて説明しています。
- 743ページの『REXX でのホスト変数の命名』および 743ページの『REXX でのホスト変数の参照』では、 REXX によるホスト変数の命名および参照について説明しています。

宣言生成プログラム - db2dclgn

宣言生成プログラムは、データベース内の指定された表の宣言を生成することにより、アプリケーション開発の能率を高めることができます。これによって、アプリケーションに簡単に挿入できる組み込み SQL 宣言のソース・ファイルを作成します。 `db2dclgn` は、C/C++、Java、COBOL、FORTRAN の各言語をサポートします。

宣言ファイルを生成するには、 `db2dclgn` コマンドを次の形式で入力してください。

`db2dclgn -d` *database-name* `-t` *table-name* [*options*]

たとえば、 SAMPLE データベース内の STAFF 表の宣言を C 言語で出力ファイル `staff.h` に生成するには、次のコマンドを入力します。

```
db2dclgn -d sample -t staff -l C
```

生成される `staff.h` ファイルには以下が含まれます。

```
struct
{
  short id;
  struct
  {
    short length;
    char data[9];
  } name;
  short dept;
  char job[5];
  short years;
  double salary;
  double comm;
} staff;
```

`db2dclgn` の詳細については、*コマンド解説書* を参照してください。

表 4. ホスト変数の宣言

言語	ソース・コード例

C/C++	`EXEC SQL BEGIN DECLARE SECTION;`
	`  short     dept=38, age=26;`
	`  double    salary;`
	`  char      CH;`
	`  char      name1[9], NAME2[9];`
	`  /* C comment */`
	`  short     nul_ind;`
	`EXEC SQL END DECLARE SECTION;`

表 4. ホスト変数の宣言 (続き)

<table>
<tr><th>言語</th><th>ソース・コード例</th></tr>
<tr><td>Java</td><td><pre>// Note that Java host variable declarations follow
// normal Java variable declaration rules, and have
// no equivalent of a DECLARE SECTION
  short   dept=38, age=26;
  double   salary;
  char   CH;
  String   name1[9], NAME2[9];
  /* Java comment */
  short   nul_ind;</pre></td></tr>
<tr><td>COBOL</td><td><pre> EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   01 age   PIC S9(4) COMP-5 VALUE 26.
   01 DEPT   PIC S9(9) COMP-5 VALUE 38.
   01 salary   PIC S9(6)V9(3) COMP-3.
   01 CH   PIC X(1).
   01 name1   PIC X(8).
   01 NAME2   PIC X(8).
* COBOL comment
   01 nul-ind   PIC S9(4) COMP-5.
 EXEC SQL END DECLARE SECTION END-EXEC.</pre></td></tr>
<tr><td>FORTRAN</td><td><pre>   EXEC SQL BEGIN DECLARE SECTION
         integer*2   age   /26/
         integer*4   dept /38/
         real*8   salary
         character   ch
         character*8   name1,NAME2
C   FORTRAN comment
         integer*2   nul_ind
       EXEC SQL END DECLARE SECTION</pre></td></tr>
</table>

表 5. ホスト変数の参照

<table>
<tr><th>言語</th><th>ソース・コード例</th></tr>
<tr><td>C/C++</td><td><pre>EXEC SQL FETCH C1 INTO :cm;
printf("Commission = %f¥n", cm);</pre></td></tr>
<tr><td>JAVA (SQLJ)</td><td><pre>#SQL { FETCH :c1 INTO :cm };
System.out.println("Commission = " + cm);</pre></td></tr>
<tr><td>COBOL</td><td><pre>EXEC SQL FETCH C1 INTO :cm END-EXEC
DISPLAY 'Commission = ' cm</pre></td></tr>
<tr><td>FORTRAN</td><td><pre>EXEC SQL FETCH C1 INTO :cm
WRITE(*,*) 'Commission = ', cm</pre></td></tr>
</table>

標識変数の使用

Java 以外の言語で作成されたアプリケーションは、ヌル値を受け取ることができるホスト変数と*標識変数* とを関連付けることによって、ヌル値を受け取る準備をしておく必要があります。 Java アプリケーションは、ホスト変数の値を Java のヌル値と比較して、受け取った値がヌル値かどうかを判別します。標識変数は、データベース・マネージャーとホスト・アプリケーションにより共用されます。したがって、標識変数はアプリケーション内ではホスト変数として宣言しておくことが必要です。このホスト変数は、SQL データのタイプ SMALLINT に対応します。

標識変数は SQL ステートメントでホスト変数の直後に置かれ、接頭部としてコロンが付けられます。スペースを用いると標識変数とホスト変数を分けることができますが、このスペースは必須ではありません。ただし、ホスト変数と標識変数の間にコンマを使用しないでください。また、オプションの INDICATOR キーワードをホスト変数とその標識の間に置いて標識変数を指定することもできます。

標識変数は、サポートされているホスト言語で INDICATOR キーワードを用いて標識変数を使用する方法について示しています。

言語　　ソース・コード例

C/C++

```
EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind;
if ( cmind < 0 )
   printf( "Commission is NULL¥n" );
```

Java (SQLJ)

```
#SQL { FETCH :c1 INTO :cm };
if ( cm == null )
   System.out.println( "Commission is NULL¥n" );
```

COBOL

```
EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind END-EXEC
IF cmind LESS THAN 0
   DISPLAY 'Commission is NULL'
```

FORTRAN

```
EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind
IF ( cmind .LT. 0 ) THEN
   WRITE(*,*) 'Commission is NULL'
ENDIF
```

上の例で、*cmind* は負の値かどうかが検査されます。負の値ではない場合、アプリケーションは *cm* の戻り値を使用することができます。 負の値の場合、取り出される値は NULL で、*cm* は使用されません。この場合、データベース・マネージャーはホスト変数の値を変更しません。

注: データベースの構成パラメーター DFT_SQLMATHWARN を 'YES' に設定した場合、 *cmind* の値は -2 になることがあります。これは、算術計算エラーのある式を評価したため、または結果数値をホスト変数に変換しようとした時にオーバーフローが起きたために、 NULL になったことを示しています。

データ・タイプが NULL を処理できる場合、アプリケーションは NULL 標識を指定しなければなりません。そうでない場合は、エラーが発生することがあります。 NULL 標識を使用しない場合、SQLCODE -305 (SQLSTATE 22002) が戻されます。

SQLCA 構造が切り捨て警告を示す場合、標識変数の切り捨て検査を行うことができます。標識変数が正の値の場合は、切り捨てが行われます。

- TIME データ・タイプの秒の部分が切り捨てられると、標識値には切り捨てられたデータの秒の部分が含まれる。
- 他のすべてのストリングのデータ・タイプ (ラージ・オブジェクト (LOB) を除く) の場合、標識値は戻されたデータの実際の長さを表します。ユーザー定義特殊タイプ (UDT) は、それらの基本タイプと同じように扱われます。

INSERT または UPDATE ステートメントを処理中に、データベース・マネージャーは標識変数を検査します (標識変数がある場合)。標識変数が負の値の場合、データベース・マネージャーは目標の列値を NULL に設定します (NULL が使用できる場合)。 標識変数がゼロ以上の場合、データベース・マネージャーは関連付けられたホスト変数の値を使用します。

ホスト変数に割り当てられる際にストリング列の値が割り当てられた場合、 SQLCA 構造の SQLWARN1 フィールドに 'X' または 'W' が入れられる場合があります。ヌル終止符が切り捨てられた場合は、'N' が入ります。

下記の条件のすべてに適合した場合にのみ、データベース・マネージャーから 'X' の値が戻されます。
- 混合コード・ページ接続が、データベースのコード・ページからアプリケーションのコード・ページに文字ストリング・データを変換してデータの長さが変わる場合に行われる
- カーソルがブロック化されている
- 標識変数がアプリケーションにより指定されている

標識変数に戻される値は、アプリケーションのコード・ページでの文字ストリングの長さになります。

それ以外の場合でデータ切り捨てが行われると、 (NULL 終止符の切り捨てとは対照的に) データベース・マネージャーは必ず 'W' を戻します。この場合、データベース・マネージャーは、選択リスト項目のコード・ページ (アプリケーションのコード・ページであれ、データベースのコード・ページであれ、あるいは何もない場合であれ) にある結果文字ストリングの長さを指す標識変数の値をアプリケーションに戻します。詳細については、*SQL 解説書* を参照してください。

データ・タイプ

DB2 表の各列には、列の作成時に *SQL データ・タイプ* が付与されます。列にデータ・タイプを割り当てる方法については、 *SQL 解説書* の CREATE TABLE ステートメントを参照してください。データベース・マネージャーは、以下の列データ・タイプをサポートしています。

SMALLINT
16 ビットの符号付き整数。

INTEGER
32 ビットの符号付き整数。このタイプの同義語として **INT** を使用できます。

BIGINT 64 ビットの符号付き整数。

DOUBLE
倍精度浮動小数点。 **DOUBLE PRECISION** および **FLOAT(n)** (n は 24 より大きい) はこのタイプの同義語です。

REAL 単精度浮動小数点 **FLOAT(n)** (n は 24 より小さい) はこのタイプの同義語です。

DECIMAL
パック 10 進数。このタイプの同義語として **DEC**、**NUMERIC**、および **NUM** を使用できます。

CHAR 1〜254 バイトの長さの固定長文字ストリング。このタイプの同義語として **CHARACTER** を使用できます。

VARCHAR
1 バイトから 32672 バイトの長さの可変長文字ストリング。このタイプの同義語として **CHARACTER VARYING** および **CHAR VARYING** を使用できます。

LONG VARCHAR
1 バイトから 32 700 バイトの長さのロング可変長文字ストリング

CLOB 1 バイトから 2 ギガバイトの長さのラージ・オブジェクト可変長文字ストリング

BLOB 1 バイトから 2 ギガバイトの長さのラージ・オブジェクト可変長 2 進ストリング

DATE タイム・スタンプを表す長さ 10 バイトの文字ストリング

TIME タイムを表す長さ 8 バイトの文字ストリング

TIMESTAMP
タイム・スタンプを表す長さ 26 バイトの文字ストリング

以下のデータ・タイプは、 2 バイト文字セット (DBCS) および拡張 UNIX コード (EUC) 文字環境でしか使用できません。

GRAPHIC
: 長さ 1〜127 の 2 バイト文字のラージ・オブジェクト固定長グラフィック・ストリング

VARGRAPHIC
: 長さ 1〜16336 の 2 バイト文字の固定長グラフィック・ストリング

LONG VARGRAPHIC
: 長さ 1〜16 350 の 2 バイト文字のロング可変長グラフィック・ストリング

DBCLOB
: 長さ 1〜1 073 741 823 の 2 バイト文字のラージ・オブジェクト可変長グラフィック・ストリング

注:

1. サポートされているすべてのデータ・タイプに、NOT NULL 属性を指定できます。これは別のタイプとして扱われます。
2. 上記の一連のデータ・タイプは、ユーザー定義特殊タイプ (UDT) を定義することにより拡張することができます。 UDT は、組み込み SQL タイプの 1 つの表現を使用する、別個のデータ・タイプです。

サポートされるホスト言語のデータ・タイプは、データベース・マネージャーのデータ・タイプの大多数に対応しています。ホスト変数宣言には、これらのホスト言語のデータ・タイプだけが使用できます。プリコンパイラーは、ホスト変数宣言を検出すると、適切な SQL データ・タイプの値を判別します。データベース・マネージャーはこの値を使用して、アプリケーションとの間でやり取りするデータを変換します。

データベース・マネージャーが異なるデータ・タイプ間の比較と割り当てをどのように処理するかを理解することは、アプリケーション・プログラマーにとって重要です。つまり、データベース・マネージャーが 2 つの SQL 列データ・タイプと 2 つのホスト言語データ・タイプ (あるいはそのいずれか) で処理を行っているとしても、データ・タイプは割り当ておよび比較の操作中は互いに互換性を持たなければなりません。

データ・タイプの互換性に関する*一般的な* 規則とは、サポートされるホスト言語の数値データ・タイプはすべてデータベース・マネージャーの数値データ・タイプと比較して割り当てることができ、ホスト言語の文字タイプはすべてデータベース・マネージャーの文字タイプと互換性があるということです。数値タイプには文字タイプとの互換性がありません。ただし、この一般的な規則には、ラージ・オブジェクトでの処理時に課される特徴および制限に基づいた例外もあります。

SQL ステートメント内であれば、DB2 では互換性のあるデータ・タイプ同士での変換が可能です。たとえば、以下の SELECT ステートメントでは、SALARY と BONUS は DECIMAL 列ですが、各従業員の合計支給額は DOUBLE データとして戻されます。

```
SELECT EMPNO, DOUBLE(SALARY+BONUS) FROM EMPLOYEE
```

上記のステートメントの実行では、 DECIMAL と DOUBLE のデータ・タイプ間で変換が行われることに注目してください。 画面に表示される照会結果をもっと読みやすくするには、次の SELECT ステートメントを使用することができます。

```
SELECT EMPNO, DIGIT(SALARY+BONUS) FROM EMPLOYEE
```

アプリケーション内でデータを変換するには、この変換をサポートするその他のルーチン、クラス、組み込みタイプ、または API があるかコンパイラーのベンダーにお問い合わせください。

文字データ・タイプには文字変換が必要な場合もあります。アプリケーションのコード・ページがデータベースのコード・ページと異なる場合は、 529ページの『異なるコード・ページ間での変換』を参照してください。

サポートされる SQL データ・タイプのリストと、それに対応するホスト言語データ・タイプについては、以下を参照してください。
- C/C++ の場合、645ページの『C および C++ でのサポートされている SQL データ・タイプ』
- Java の場合、658ページの『Java でサポートされている SQL データ・タイプ』
- COBOL の場合、718ページの『COBOL でサポートされる SQL データ・タイプ』
- FORTRAN の場合、734ページの『FORTRAN でサポートされている SQL データ・タイプ』
- REXX の場合、749ページの『REXX でサポートされている SQL データ・タイプ』

SQL データ・タイプ、割り当てと比較の規則、およびデータ変換と変換エラーの詳細については、 *SQL 解説書* を参照してください。

STATIC プログラムでの標識変数の使用

次のコード・セグメントは、 72ページの『C の例: STATIC.SQC』で挙げた C バージョンの STATIC サンプル・プログラムに対応するセグメントへの修正を示しています。これらのセグメントは、ヌル可能なデータ列での標識変数のインプリメンテーションを示しています。この例では、STATIC プログラムは別の列 WORKDEPT を選択するために拡張されます。この列にはヌル値を使用することができます。標識変数は、使用前にホスト変数として宣言しておかなければなりません。

```
⋮

EXEC SQL BEGIN DECLARE SECTION;
  char  wd[3];
  short wd_ind;
  char firstname[13];

⋮

EXEC SQL END DECLARE SECTION;
```

```
⋮

/* CONNECT TO SAMPLE DATABASE */

⋮

EXEC SQL SELECT FIRSTNME, WORKDEPT INTO :firstname, :wd:wdind
  FROM EMPLOYEE
  WHERE LASTNAME = 'JOHNSON';

⋮
```

カーソルを用いた複数行の選択

SQL では、アプリケーションが行のセットを取り出すことができるようにするため、*カーソル* という手法を用います。

カーソルの概念を理解しやすくするために、データベース・マネージャーが*結果表* を作成し、そこに SELECT ステートメントを実行して検索されたすべての行を保持する場合を考えてみてください。カーソルを用いて結果表の*現在行* を識別して指示することにより、その表からの行をアプリケーションで使用できるようにします。カーソルを使用すると、アプリケーションは結果表から各行を順次取り出すことができ、最終的にはデータの終わり状態、すなわち NOT FOUND 状態、SQLCODE +100 (SQLSTATE 02000) になります。 SELECT ステートメントを実行した結果取り出された行のセットは、0、1、またはそれ以上の行で構成されます。これは探索条件を満たす行数によって決まります。

カーソル処理に必要な手順は以下のとおりです。

1. DECLARE CURSOR ステートメントを用いてカーソルを指定する。
2. OPEN ステートメントを用いて照会を実行し、結果表を作成する。
3. FETCH ステートメントを用いて行を一度に 1 行ずつ取り出す。
4. DELETE または UPDATE ステートメントを用いて行を処理する (必要な場合)。
5. CLOSE ステートメントを用いて行を終了する。

アプリケーションは同時に複数のカーソルを使用することができます。各カーソルには DECLARE CURSOR、OPEN、CLOSE、および FETCH ステートメントのセットが必要です。

アプリケーションが一連の行を選択し、カーソルを用いて一度に 1 行ずつ処理する方法の例については、 90ページの『例: カーソル・プログラム』を参照してください。

カーソルの宣言と使用

DECLARE CURSOR ステートメントはカーソルを定義、命名し、 SELECT ステートメントを用いて取り出した行のセットを識別します。

アプリケーションはカーソルに名前を割り当てます。この名前は、その後に続く OPEN、FETCH、および CLOSE ステートメントで参照されます。照会とは、任意の有効な選択ステートメントです。

カーソル・ステートメントの宣言は、静的 SELECT ステートメントに関連する DECLARE ステートメントを示しています。

言語　　ソース・コード例

C/C++

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT PNAME, DEPT FROM STAFF
  WHERE JOB=:host_var;
```

Java (SQLJ)

```
#sql iterator cursor1(host_var data type);
#sql cursor1 = { SELECT PNAME, DEPT FROM STAFF
                 WHERE JOB=:host_var };
```

COBOL

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT NAME, DEPT FROM STAFF
    WHERE JOB=:host-var END-EXEC.
```

FORTRAN

```
 EXEC SQL DECLARE C1 CURSOR FOR
+  SELECT NAME, DEPT FROM STAFF
+  WHERE JOB=:host_var
```

注: DECLARE ステートメントの位置は自由ですが、最初に使用するカーソルの位置よりも上に置く必要があります。

カーソルおよび作業単位に関する考慮事項

COMMIT または ROLLBACK 操作のカーソルのアクションは、カーソルがどのように定義されているかによって異なります。

読み取り専用カーソル

カーソルが読み取り専用であると判別され、かつ反復可能読み取り分離レベルを使用する場合、作業単位に必要なシステム表上に反復可能読み取りロックは依然として集められ保持されます。そのため、アプリケーションは読み取り専用カーソルの場合でも定期的に COMMIT ステートメントを発行することが重要です。

WITH HOLD オプション

アプリケーションが COMMIT ステートメントを発行してある作業単位を完了すると、すべてのオープン・カーソル (WITH HOLD オプションを使用して宣言されるものを除く) は、データベース・マネージャーにより自動的にクローズされます。

WITH HOLD で宣言されたカーソルは、複数の作業単位間でアクセスするリソースを保持します。カーソルを WITH HOLD で宣言した場合の影響は、作業単位がどのように終了するかによって決まります。

作業単位が COMMIT ステートメントで終了する場合、 WITH HOLD で定義されたオープン・カーソルは OPEN のままです。カーソルは結果表の次の論理行の前に置かれます。さらに、WITH HOLD で定義された OPEN カーソルを参照する準備済みのステートメントも保存されます。 COMMIT の直前にある、特定のカーソルに関連する FETCH および CLOSE 要求だけが有効です。 UPDATE WHERE CURRENT OF および DELETE WHERE CURRENT OF ステートメントは、同じ作業単位内で取り出された行の場合に限り有効です。 作業単位中にパッケージが再バインドされると、保留されたカーソルはすべてクローズします。

作業単位が ROLLBACK ステートメントで終了する場合、オープン・カーソルはすべてクローズされ、作業単位の間に獲得したロックはすべて解放され、その作業単位での処理に依存する準備済みステートメントはすべて消去されます。

たとえば、TEMPL 表に 1000 項目が入っており、すべての従業員の給与列を更新するとします。 100 行更新するたびに COMMIT ステートメントを発行するようにします。

1. WITH HOLD オプションを用いてカーソルを宣言する。

   ```
   EXEC SQL DECLARE EMPLUPDT CURSOR WITH HOLD FOR
     SELECT EMPNO, LASTNAME, PHONENO, JOBCODE, SALARY
     FROM TEMPL FOR UPDATE OF SALARY
   ```

2. カーソルをオープンし、一度に 1 行ずつ結果表からデータを取り出す。

   ```
   EXEC SQL OPEN EMPLUPDT
     .
     .
     .

   EXEC SQL FETCH EMPLUPDT
     INTO :upd_emp, :upd_lname, :upd_tele, :upd_jobcd, :upd_wage,
   ```

3. 行を更新または削除する場合は、 WHERE CURRENT OF オプションを用いて UPDATE または DELETE ステートメントを使用する。たとえば、現在行を更新するには、プログラムは以下のようにすることができます。

   ```
   EXEC SQL UPDATE TEMPL SET SALARY = :newsalary
     WHERE CURRENT OF EMPLUPDT
   ```

4. COMMIT を発行した後、別の行を更新する前に FETCH を発行しなければならない。

アプリケーションに SQLCODE -501 (SQLSTATE 24501) を検出し、処理するコードを組み込むようにしてください。この戻り値は、アプリケーションが次のいずれかである場合に、 FETCH または CLOSE ステートメントで戻されることがあります。

- WITH HOLD 宣言されたカーソルを使用している場合

- 2 つ以上の作業単位を実行し、かつ WITH HOLD カーソルが作業単位の境界を超えてオープンしている場合

アプリケーションがある表を消去したために、その表に依存しているパッケージが無効になると、そのパッケージは動的に再バインドされます。このような場合、データベース・マネージャーがカーソルをクローズするため、 FETCH または CLOSE ステートメントに SQLCODE -501 (SQLSTATE 24501) が戻されます。こうした状況で SQLCODE -501 (SQLSTATE 24501) を処理する方法は、カーソルから行を取り出したいかどうかによって決まります。

- カーソルから行を取り出したい場合は、カーソルをオープンしてから、 FETCH ステートメントを実行します。ただし、OPEN ステートメントによってカーソルの開始位置が移動するので注意してください。 COMMIT WORK ステートメントで保留されていた直前の位置は失われます。
- カーソルから行を取り出すことを望まない場合は、カーソルに対してもはやどんな SQL 要求も発行しません。

WITH RELEASE オプション: アプリケーションが WITH RELEASE オプションを使ってカーソルをクローズすると、 DB2 はカーソルが保持している読み取りロックをすべて解放しようとします。そうすると、カーソルは書き込みロックしか保持できなくなります。アプリケーションが RELEASE オプションを使わないでカーソルをクローズすると、作業単位の完了時に読み取りおよび書き込みロックは解放されます。

例: カーソル・プログラム

以下のサンプル・プログラムは、カーソルを定義して使用する SQL ステートメントを示しています。カーソルは静的 SQL を用いて処理されます。このサンプルは、以下のプログラム言語で入手可能です。

C cursor.sqc

Java Cursor.sqlj

COBOL cursor.sqb

REXX 言語は、静的 SQL をサポートしないため、サンプルはありません。動的にカーソルを処理する REXX の例については、 140ページの『例: 動的 SQL プログラム』を参照してください。

カーソル・プログラムの動作の仕組み

1. **カーソルを宣言する。** DECLARE CURSOR ステートメントはカーソル `c1` を照会に関連付けます。照会は、アプリケーションが FETCH ステートメントを用いて取り出す行を識別します。 `staff` の `job` フィールドは、結果表に指定されていなくても更新可能として定義されます。

2. **カーソルをオープンする。** カーソル c1 がオープンすると、データベース・マネージャーは照会を実行し、結果表を作成します。カーソルは第 1 行目より*前に* 置かれます。
3. **行を取り出す。** FETCH ステートメントはカーソルを次の行に置き、その行の内容をホスト変数に移動します。この行が*現在* 行になります。
4. **カーソルをクローズする。** CLOSE ステートメントを発行すると、カーソルに関連したリソースが解放されます。ただし、カーソルは再度オープンすることができます。

CHECKERR マクロ / 関数は、プログラム外部にあるエラー検査ユーティリティーです。エラー検査ユーティリティーの所在は、ご使用のプログラム言語により異なります。

C
DB2 API を呼び出す C プログラムの場合、 utilapi.c 内の sqlInfoPrint 関数は、 utilapi.h 内の API_SQL_CHECK として再定義されます。 C 組み込み SQL プログラムの場合、 utilemb.sqc 内の sqlInfoPrint 関数は、 utilemb.h 内の EMB_SQL_CHECK として再定義されます。

Java
SQL エラーは SQLException としてスローされ、アプリケーションの catch ブロックで処理されます。

COBOL
CHECKERR は checkerr.cbl という名前の外部プログラムです。

FORTRAN
CHECKERR は util.f ファイルにあるサブルーチンです。

このエラー検査ユーティリティーのソース・コードについては、 125ページの『プログラム例での GET ERROR MESSAGE の使用』を参照してください。

C の例: CURSOR.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[])
{

   EXEC SQL BEGIN DECLARE SECTION;
      char   pname[10];
      short  dept;
      char userid[9];
      char passwd[19];
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: CURSOR ¥n" );

   if (argc == 1)
   {
      EXEC SQL CONNECT TO sample;
  EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3)
   {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else
   {
      printf ("¥nUSAGE: cursor [userid passwd]¥n¥n");
      return 1;
   } /* endif */

   EXEC SQL DECLARE c1 CURSOR FOR   1
            SELECT name, dept FROM staff WHERE job='Mgr'
            FOR UPDATE OF job;

   EXEC SQL OPEN c1;   2
   EMB_SQL_CHECK("OPEN CURSOR");

   do
   {
      EXEC SQL FETCH c1 INTO :pname, :dept;   3
      if (SQLCODE != 0) break;

      printf( "%-10.10s in dept. %2d will be demoted to Clerk¥n",
         pname, dept );
   } while ( 1 );
```

```
   EXEC SQL CLOSE c1;   4
   EMB_SQL_CHECK("CLOSE CURSOR");

   EXEC SQL ROLLBACK;
   EMB_SQL_CHECK("ROLLBACK");
   printf( "¥nOn second thought -- changes rolled back.¥n" );

   EXEC SQL CONNECT RESET;
   EMB_SQL_CHECK("CONNECT RESET");
   return 0;
}
/* end of program : CURSOR.SQC */
```

Java の例: Cursor.sqlj

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator CursorByName(String name, short dept) ;
#sql iterator CursorByPos(String, short ) ;

class Cursor
{   static
  {   try
    {   Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
    }
    catch (Exception e)
    {   System.out.println ("¥n  Error loading DB2 Driver...¥n");
      System.out.println (e);
      System.exit(1);
    }
  }

  public static void main(String argv[])
  {   try
    {   System.out.println ("  Java Cursor Sample");

      String url = "jdbc:db2:sample";       // URL is jdbc:db2:dbname
      Connection con = null;

      // Set the connection
      if (argv.length == 0)
      {   // connect with default id/password
        con = DriverManager.getConnection(url);
      }
      else if (argv.length == 2)
      {   String userid = argv[0];
        String passwd = argv[1];

        // connect with user-provided username and password
        con = DriverManager.getConnection(url, userid, passwd);
      }
      else
      {   throw new Exception("¥nUsage: java Cursor [username password]¥n");
      }

      // Set the default context
      DefaultContext ctx = new DefaultContext(con);
      DefaultContext.setDefaultContext(ctx);

      // Enable transactions
      con.setAutoCommit(false);

      // Using cursors
      try
      {   CursorByName cursorByName;
        CursorByPos  cursorByPos;
```

```
            String name = null;
            short  dept=0;

            // Using the JDBC ResultSet cursor method
            System.out.println("¥nUsing the JDBC ResultSet cursor method");
            System.out.println(" with a 'bind by name' cursor ...¥n");

            #sql cursorByName = {
                  SELECT name, dept FROM staff WHERE job='Mgr' };  1
            while (cursorByName.next())  2
            {   name = cursorByName.name();  3
              dept = cursorByName.dept();

              System.out.print (" name= " + name);
              System.out.print (" dept= " + dept);
              System.out.print ("¥n");
            }
            cursorByName.close();  4


            // Using the SQLJ iterator cursor method
            System.out.println("¥nUsing the SQLJ iterator cursor method");
            System.out.println(" with a 'bind by position' cursor ...¥n");

            #sql cursorByPos = {
                   SELECT name, dept FROM staff WHERE job='Mgr' };  1  2
            while (true)
            {   #sql { FETCH :cursorByPos INTO :name, :dept };  3
              if (cursorByPos.endFetch()) break;

              System.out.print (" name= " + name);
              System.out.print (" dept= " + dept);
              System.out.print ("¥n");
            }
            cursorByPos.close();  4
          }
          catch( Exception e )
          {   throw e;
          }
          finally
          {   // Rollback the transaction
            System.out.println("¥nRollback the transaction...");
            #sql { ROLLBACK };
            System.out.println("Rollback done.");
          }
        }
        catch( Exception e )
        {   System.out.println (e);
        }
      }
    }
```

COBOL の例: CURSOR.SQB

```
       Identification Division.
       Program-ID. "cursor".

       Data Division.
       Working-Storage Section.

           copy "sqlenv.cbl".
           copy "sql.cbl".
           copy "sqlca.cbl".

           EXEC SQL BEGIN DECLARE SECTION END-EXEC.
       01 pname             pic x(10).
       77 dept              pic s9(4) comp-5.
       01 userid            pic x(8).
       01 passwd.
         49 passwd-length   pic s9(4) comp-5 value 0.
         49 passwd-name     pic x(18).
           EXEC SQL END DECLARE SECTION END-EXEC.

       77 errloc          pic x(80).

       Procedure Division.
       Main Section.
           display "Sample COBOL program: CURSOR".

           display "Enter your user id (default none): "
                with no advancing.
           accept userid.

           if userid = spaces
             EXEC SQL CONNECT TO sample END-EXEC
           else
             display "Enter your password : " with no advancing
             accept passwd-name.

      * Passwords in a CONNECT statement must be entered in a VARCHAR format
      * with the length of the input string.
           inspect passwd-name tallying passwd-length for characters
              before initial " ".

           EXEC SQL CONNECT TO sample USER :userid USING :passwd
              END-EXEC.
           move "CONNECT TO" to errloc.
           call "checkerr" using SQLCA errloc.

           EXEC SQL DECLARE c1 CURSOR FOR                              1
                    SELECT name, dept FROM staff
                    WHERE job='Mgr'
                    FOR UPDATE OF job END-EXEC.

           EXEC SQL OPEN c1 END-EXEC.                                  2
           move "OPEN CURSOR" to errloc.
           call "checkerr" using SQLCA errloc.
```

```
    perform Fetch-Loop thru End-Fetch-Loop
       until SQLCODE not equal 0.

    EXEC SQL CLOSE c1 END-EXEC.                                   4
    move "CLOSE CURSOR" to errloc.
    call "checkerr" using SQLCA errloc.

    EXEC SQL ROLLBACK END-EXEC.
    move "ROLLBACK" to errloc.
    call "checkerr" using SQLCA errloc.
    DISPLAY "On second thought -- changes rolled back.".

    EXEC SQL CONNECT RESET END-EXEC.
    move "CONNECT RESET" to errloc.
    call "checkerr" using SQLCA errloc.
End-Main.
    go to End-Prog.

Fetch-Loop Section.
    EXEC SQL FETCH c1 INTO :PNAME, :DEPT END-EXEC.                3
    if SQLCODE not equal 0
       go to End-Fetch-Loop.
    display pname, " in dept. ", dept,
       " will be demoted to Clerk".
End-Fetch-Loop. exit.

End-Prog.
    stop run.
```

検索済みデータの更新と削除

カーソルによって参照された行は、更新したり削除したりできます。更新可能な行の場合、カーソルに対応する照会は読み取り専用であってはなりません。照会が更新可能または削除可能になる条件については、 *SQL 解説書* の説明を参照してください。

検索されたデータの更新

カーソルを用いて更新を行うためには、 UPDATE ステートメントで WHERE CURRENT OF 文節を使用してください。結果表の列を更新したいシステムを示すには、FOR UPDATE 文節を使用します。 FOR UPDATE での列の指定は全部を選択しなくてもよいため、カーソルで明確に検索されない列でも更新することができます。 FOR UPDATE 文節を列名を使わずに指定すると、表の中のすべての列や、外部で全選択された最初の FROM 文節で識別された視点は更新可能であると見なされます。 FOR UPDATE 文節では、必要以上の列を指定しないでください。 FOR UPDATE 文節の余分な列に名前を付けると、 DB2 がデータにアクセスする能率を低下させる場合もあります。

検索されたデータの削除

カーソルを用いた削除は、DELETE ステートメントで WHERE CURRENT OF 文節を使用して行います。一般に、FOR UPDATE 文節はカーソルの現在行の削除には必要ありません。 SAA1 に設定された LANGLEVEL でプリコンパイルされ、 BLOCKING ALL でバインドされたアプリケーション内の SELECT ステートメントまたは DELETE ステートメントのいずれかに対して動的 SQL (動的 SQL については、133ページの『第5章 動的 SQL プログラムの作成』を参照) を使った場合だけは例外です。この場合、SELECT ステートメントで FOR UPDATE 文節を指定する必要があります。プリコンパイラー・オプションの詳細については、 *コマンド解説書* を参照してください。

DELETE ステートメントを使用すると、カーソルで参照される行を削除することができます。このとき、カーソルは次の 行の前に置かれたままになるため、カーソルに対して WHERE CURRENT OF 操作をさらに実行する前に、 FETCH ステートメントを発行する必要があります。

カーソルのタイプ

カーソルは以下の 3 種類に分類されます。

読み取り専用

このカーソルの行は読み取り専用で、更新することはできません。読み取り専用カーソルは、アプリケーションがデータを読み取る場合にだけ使用され、データの修正には使用されません。カーソルは、読み取り専用の選択ステートメントに基づいている場合に限り、読み取り専用と見なされます。更新可能でない結果表を定義する選択ステートメントについては、 『検索されたデータの更新』で説明した規則を参照してください。

読み取り専用カーソルにはパフォーマンス上の利点もあります。読み取り専用カーソルの詳細については、 *管理の手引き: インプリメンテーション* を参照してください。

更新可能
このカーソルの行は更新することができます。アプリケーションがカーソルの行の取り出しに伴ってデータを修正する場合に、更新可能カーソルを使用します。指定した照会は、表または視点を 1 つだけ参照することができます。また、照会には FOR UPDATE 文節を組み込んで、更新されるそれぞれの列に名前を付ける必要があります (LANGLEVEL MIA プリコンパイル・オプションを使用しない場合)。

未確定
定義またはコンテキストからはカーソルが更新可能か読み取り専用かを判別することができません。こうした状況は、本来なら読み取り専用のはずのカーソルが動的 SQL ステートメントによって変更される場合に生じます。

未確定カーソルは、プリコンパイル時またはバインド時に BLOCKING ALL オプションが指定されると読み取り専用と見なされます。そうでない場合は、更新可能と見なされます。

注: 動的に処理されるカーソルは常に未確定です。

カーソルが読み取り専用か更新可能か未確定かを判別するための基準を示した全リストについては、 *SQL 解説書* を参照してください。

例: OPENFTCH プログラム

この例では、カーソルを使用して表から選択し、カーソルをオープンし、その表から行を取り出します。そして、取り出したそれぞれの行に対して、削除すべきか更新すべきかを (単純な基準に基づいて) 判別します。このサンプルは、以下のプログラム言語で入手可能です。

C openftch.sqc

Java Openftch.sqlj および OpF_Curs.sqlj

COBOL openftch.sqb

REXX 言語は静的 SQL をサポートしないため、サンプルはありません。

OPENFTCH プログラムの動作の仕組み

1. **カーソルを宣言する。** DECLARE CURSOR ステートメントはカーソル `c1` を照会に関連付けます。照会は、アプリケーションが FETCH ステートメントを用いて取り出す行を識別します。 `staff` の `job` フィールドは、結果表に指定されていなくても更新可能として定義されます。

2. **カーソルをオープンする。**カーソル c1 がオープンすると、データベース・マネージャーは照会を実行し、結果表を作成します。カーソルは第 1 行目より*前に* 置かれます。
3. **行を取り出す。** FETCH ステートメントはカーソルを次の行に置き、その行の内容をホスト変数に移動します。この行が*現在* 行になります。
4. **現在行を更新または削除する。**現在行が更新されるか削除されます。更新か削除かは、 FETCH ステートメントで戻される dept の値によって決まります。

   UPDATE ステートメントは現在行の位置を変更しないため、カーソルの位置はこの行から変わりません。

   一方、DELETE ステートメントを用いると、*現在* 行は削除されるため、状況は変化します。これは、*次の* 行の前に置かれることと同じであり、追加の WHERE CURRENT OF 操作が実行される前に FETCH ステートメントを発行する必要があります。
5. **カーソルをクローズする。** CLOSE ステートメントを発行すると、カーソルに関連したリソースが解放されます。ただし、カーソルは再度オープンすることができます。

CHECKERR マクロ / 関数は、プログラム外部にあるエラー検査ユーティリティーです。エラー検査ユーティリティーの所在は、ご使用のプログラム言語により異なります。

C DB2 API を呼び出す C プログラムの場合、 utilapi.c 内の sqlInfoPrint 関数は、 utilapi.h 内の API_SQL_CHECK として再定義されます。 C 組み込み SQL プログラムの場合、 utilemb.sqc 内の sqlInfoPrint 関数は、 utilemb.h 内の EMB_SQL_CHECK として再定義されます。

Java SQL エラーは SQLException としてスローされ、アプリケーションの catch ブロックで処理されます。

COBOL CHECKERR は checkerr.cbl という名前の外部プログラムです。

このエラー検査ユーティリティーのソース・コードについては、 125ページの『プログラム例での GET ERROR MESSAGE の使用』を参照してください。

C の例: OPENFTCH.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[])
{

   EXEC SQL BEGIN DECLARE SECTION;
      char   pname[10];
      short  dept;
      char userid[9];
      char passwd[19];
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: OPENFTCH¥n" );

   if (argc == 1)
   {
      EXEC SQL CONNECT TO sample;
      EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3)
   {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else
   {
      printf ("¥nUSAGE: openftch [userid passwd]¥n¥n");
      return 1;
   } /* endif */

   EXEC SQL DECLARE c1 CURSOR FOR   1
            SELECT name, dept FROM staff WHERE job='Mgr'
            FOR UPDATE OF job;

   EXEC SQL OPEN c1;   2
   EMB_SQL_CHECK("OPEN CURSOR");

   do
   {
      EXEC SQL FETCH c1 INTO :pname, :dept;   3
      if (SQLCODE != 0) break;

      if (dept > 40)
      {
         printf( "%-10.10s in dept. %2d will be demoted to Clerk¥n",
            pname, dept );
```

```
            EXEC SQL UPDATE staff SET job = 'Clerk'    4
               WHERE CURRENT OF c1;
            EMB_SQL_CHECK("UPDATE STAFF");
         }
         else
         {
            printf ("%-10.10s in dept. %2d will be DELETED!¥n",
               pname, dept);
            EXEC SQL DELETE FROM staff WHERE CURRENT OF c1;
            EMB_SQL_CHECK("DELETE");
         } /* endif */
      } while ( 1 );

      EXEC SQL CLOSE c1;   5
      EMB_SQL_CHECK("CLOSE CURSOR");

      EXEC SQL ROLLBACK;
      EMB_SQL_CHECK("ROLLBACK");
      printf( "¥nOn second thought -- changes rolled back.¥n" );

      EXEC SQL CONNECT RESET;
      EMB_SQL_CHECK("CONNECT RESET");
      return 0;
   }
   /* end of program : OPENFTCH.SQC */
```

Java の例: Openftch.sqlj

OpF_Curs.sqlj

```
//  PURPOSE : This file, named OpF_Curs.sqlj, contains the definition
//            of the class OpF_Curs used in the sample program Openftch.

import sqlj.runtime.ForUpdate;
#sql public iterator OpF_Curs implements ForUpdate (String, short);
```

Openftch.sqlj

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

class Openftch
{   static
  {   try
    {   Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
    }
    catch (Exception e)
    {   System.out.println ("¥n  Error loading DB2 Driver...¥n");
      System.out.println (e);
      System.exit(1);
    }
  }

  public static void main(String argv[])
  {   try
    {   System.out.println ("  Java Openftch Sample");

      String url = "jdbc:db2:sample";       // URL is jdbc:db2:dbname
      Connection con = null;

      // Set the connection
      if (argv.length == 0)
      {   // connect with default id/password
        con = DriverManager.getConnection(url);
      }
      else if (argv.length == 2)
      {   String userid = argv[0];
        String passwd = argv[1];

        // connect with user-provided username and password
        con = DriverManager.getConnection(url, userid, passwd);
      }
      else
      {   throw new Exception(
                "¥nUsage: java Openftch [username password]¥n");
      } // if - else if - else

      // Set the default context
      DefaultContext ctx = new DefaultContext(con);
      DefaultContext.setDefaultContext(ctx);
```

```
// Enable transactions
con.setAutoCommit(false);

// Executing SQLJ positioned update/delete statements.
try
{   OpF_Curs forUpdateCursor;

  String name = null;
  short  dept=0;


  #sql forUpdateCursor =
  {   SELECT name, dept
          FROM staff
          WHERE job='Mgr'
  }; // #sql              1 2

  while (true)
  {   #sql
    {   FETCH :forUpdateCursor
            INTO :name, :dept
    }; // #sql                          3
    if (forUpdateCursor.endFetch()) break;

    if (dept > 40)
    {   System.out.println (
            name + " in dept. "
            + dept + " will be demoted to Clerk");
      #sql
      {   UPDATE staff SET job = 'Clerk'
          WHERE CURRENT OF :forUpdateCursor
      }; // #sql                        4
    }
    else
    {   System.out.println (
            name + " in dept. " + dept
            + " will be DELETED!");
      #sql
      {   DELETE FROM staff
          WHERE CURRENT OF :forUpdateCursor
      }; // #sql
    } // if - else
  }
  forUpdateCursor.close();   5
}
catch( Exception e )
{   throw e;
}
finally
{   // Rollback the transaction
  System.out.println("¥nRollback the transaction...");
  #sql { ROLLBACK };
  System.out.println("Rollback done.");
} // try - catch - finally
}
```

```
    catch( Exception e )
    {   System.out.println (e);
    } // try - catch
  } // main
} // class Openftch
```

COBOL の例: OPENFTCH.SQB

```
       Identification Division.
       Program-ID. "openftch".

       Data Division.
       Working-Storage Section.

           copy "sqlca.cbl".

           EXEC SQL BEGIN DECLARE SECTION END-EXEC.
       01 pname             pic x(10).
       01 dept              pic s9(4) comp-5.
       01 userid            pic x(8).
       01 passwd.
         49 passwd-length   pic s9(4) comp-5 value 0.
         49 passwd-name     pic x(18).
           EXEC SQL END DECLARE SECTION END-EXEC.

       77 errloc          pic x(80).

       Procedure Division.
       Main Section.
           display "Sample COBOL program: OPENFTCH".

      * Get database connection information.
           display "Enter your user id (default none): "
                with no advancing.
           accept userid.

           if userid = spaces
             EXEC SQL CONNECT TO sample END-EXEC
           else
             display "Enter your password : " with no advancing
             accept passwd-name.

      * Passwords in a CONNECT statement must be entered in a VARCHAR format
      * with the length of the input string.
           inspect passwd-name tallying passwd-length for characters
              before initial " ".

           EXEC SQL CONNECT TO sample USER :userid USING :passwd
               END-EXEC.
           move "CONNECT TO" to errloc.
           call "checkerr" using SQLCA errloc.

           EXEC SQL DECLARE c1 CURSOR FOR                              1
                    SELECT name, dept FROM staff
                    WHERE job='Mgr'
                    FOR UPDATE OF job END-EXEC.

           EXEC SQL OPEN c1 END-EXEC                                   2
           move "OPEN" to errloc.
           call "checkerr" using SQLCA errloc.
```

```
* call the FETCH and UPDATE/DELETE loop.
      perform Fetch-Loop thru End-Fetch-Loop
         until SQLCODE not equal 0.

      EXEC SQL CLOSE c1 END-EXEC.                                    5
      move "CLOSE" to errloc.
      call "checkerr" using SQLCA errloc.

      EXEC SQL ROLLBACK END-EXEC.
      move "ROLLBACK" to errloc.
      call "checkerr" using SQLCA errloc.
      display "On second thought -- changes rolled back.".

      EXEC SQL CONNECT RESET END-EXEC.
      move "CONNECT RESET" to errloc.
      call "checkerr" using SQLCA errloc.
  End-Main.
      go to End-Prog.

  Fetch-Loop Section.
      EXEC SQL FETCH c1 INTO :pname, :dept END-EXEC.                 3
      if SQLCODE not equal 0
         go to End-Fetch-Loop.

      if dept greater than 40
         go to Update-Staff.

  Delete-Staff.
      display pname, " in dept. ", dept,
         " will be DELETED!".

      EXEC SQL DELETE FROM staff WHERE CURRENT OF c1 END-EXEC.
      move "DELETE" to errloc.
      call "checkerr" using SQLCA errloc.

      go to End-Fetch-Loop.

  Update-Staff.
      display pname, " in dept. ", dept,
         " will be demoted to Clerk".

      EXEC SQL UPDATE staff SET job = 'Clerk'                        4
               WHERE CURRENT OF c1 END-EXEC.
      move "UPDATE" to errloc.
      call "checkerr" using SQLCA errloc.

  End-Fetch-Loop. exit.

  End-Prog.
      stop run.
```

高度なスクロール技法

この節では、高度なスクロール技法に関する以下のトピックについて説明します。
- すでに検索済みデータのスクロール
- データのコピーを保持する方法
- データを 2 度検索する方法
- 表の末尾の位置の確立
- 以前に検索されたデータの更新

すでに検索済みデータのスクロール

アプリケーションがデータベースからデータを検索するとき、 FETCH ステートメントを使うとデータを下方へスクロールすることができます。しかし、データベース・マネージャーの組み込み SQL ステートメントにはデータを上方へスクロールする機能 (上方 FETCH 機能に相当) がありません。一方、DB2 CLI と Java では読み取り専用のスクロール可能カーソルによる上方 FETCH 機能をサポートしています。スクロール可能カーソルに関する詳細については、 *コール・レベル・インターフェースの手引きおよび解説書* および 660ページの『Java アプリケーションおよびアプレットの作成』を参照してください。組み込み SQL アプリケーションの場合、すでに検索されたデータをスクロールするには以下の技法を使うことができます。

1. 取り出されたデータのコピーを保管しておき、何らかのプログラム技法を用いてそれをスクロールする方法。
2. SQL を用いて (一般的には 2 番目の SELECT ステートメントを使用して) データを再び検索する方法。

これらのオプションについて、以下の節で詳しく説明します。
- データのコピーを保持する方法
- データを 2 度検索する方法

データのコピーを保持する方法

アプリケーションは、取り出されたデータを仮想記憶域に保管することができます。そのデータが仮想記憶域に入りきらない場合、アプリケーションはそのデータを一時ファイルに書き込むことができます。この方法の利点は、データベース内のデータがトランザクションによる一時的な変更を受けた場合でさえも、ユーザーは、取り出されたデータとまったく同じものを、上方スクロールによって常に見ることができるという点です。

反復可能読み取りの分離レベルを使用すると、カーソルをクローズしたりオープンすることにより、トランザクションから検索したデータをもう一度検索することができます。検索結果のデータは、他のアプリケーションにより更新されることはありません。データの更新方法は、分離レベルおよびロックにより左右されます。

データを 2 度検索する方法

この技法は、データをもう一度見ようとする順序により異なります。

- 先頭からの検索
- 中間からの検索
- 2 番目の結果表内での行の順序
- 逆順での検索

先頭からの検索

データを先頭から再び検索するには、活動カーソルをクローズしてそれを再オープンするだけで済みます。このアクションによってカーソルは結果表の先頭に置かれます。ただし、アプリケーションがその表に対してロックを保持していない限り、他のユーザーがその表に変更を加える可能性があるので、以前に結果表の最初の行であったものが、最初の行でなくなるということもあり得ます。

中間からの検索

結果表の中ほどから 2 度目のデータ検索を行うには、 2 度目の SELECT ステートメントを実行し、そのステートメント上で 2 つ目のカーソルを宣言してください。たとえば、最初の SELECT ステートメントが次のものであるとします。

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  ORDER BY DEPTNO
```

今度は DEPTNO = 'M95' から始まる行に戻って、その場所から順番に行を取り出すとします。この場合は、次のようにコーディングしてください。

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  AND DEPTNO >= 'M95'
  ORDER BY DEPTNO
```

このステートメントによって、カーソルは希望する場所に置かれます。

2 番目の結果表内での行の順序

2 番目の結果表の行は、最初の結果表と同じ順序で表示されるとは限りません。データベース・マネージャーは、SELECT ステートメントが ORDER BY 機能を使用していない場合、行の順序を重要視しません。そのため、同じ DEPTNO 値を持つ行がいくつかある場合には、 2 番目の SELECT ステートメントが最初のものとは違う順序で行を検索する場合があります。保証されているのは、ORDER BY DEPTNO 文節での要求に従って、それらすべてが部門番号の順に並べられるということだけです。

同じ SQL ステートメントを同じホスト変数を指定して 2 度実行したとしても、順序付けが異なる場合があります。たとえば、2 度目の実行がなされるまでの間にカタログの統計が更新されたり、索引が作成されるか除去される場合もあります。その後で SELECT ステートメントをもう一度実行することも考えられます。

最初の SELECT が持っていなかった述部を 2 番目の SELECT が持っている場合、配列が変更することがあります。それはデータベース・マネージャーが新しい述部に対して索引を使用するということがあり得るからです。たとえば、この例で、データベース・マネージャーが最初のステートメントに対しては LOCATION 上の索引を選び、 2 番目のものに対しては DEPTNO 上の索引を選ぶ場合があります。行は索引キーの順に従って取り出されるため、 2 番目の順序は最初の順序と同じとは限りません。

この場合も、2 つの同様な SELECT ステートメントを実行したときに、統計が変更されず、索引の作成も除去も行われなかったとしても、行の順序が異なる場合があります。例では、LOCATION の異なる値が多数ある場合、データベース・マネージャーは両方のステートメント用に LOCATION 上で 1 つの索引を選択することができます。しかし、2 番目のステートメントの DEPTNO の値を次のように変えると、データベース・マネージャーは DEPTNO 上の索引を選ぶことがあります。

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  AND DEPTNO >= 'Z98'
  ORDER BY DEPTNO
```

SQL ステートメントの形式とこのステートメントの値との間にはわずかな関係しかないため、順序が ORDER BY 文節で固有のものとして定められているのでない限り、 2 つの異なった SQL ステートメントが同じ順序で行を戻してくるとは考えないでください。

逆順での検索

行の昇順がデフォルトの設定です。 DEPTNO のおのおのの値に対する行が 1 つしかない場合、次のステートメントは行を固有の昇順に指定します。

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  ORDER BY DEPTNO
```

同じ行を逆順に検索するには、次のステートメントのように順序を降順として指定してください。

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  ORDER BY DEPTNO DESC
```

2 番目のステートメント上のカーソルは、最初のステートメント上のカーソルからの順番とはまったく逆の順番に行を検索します。検索の順序は、最初のステートメントが固有の順序を指定している場合にのみ保証されます。

行を逆順で検索する場合、DEPTNO 列に、 1 つは昇順で、もう 1 つは降順の 2 つの索引を持つと便利です。

表の末尾の位置の確立

データベース・マネージャーは表内に保管されているデータを配列することは行いません。そのため、表の末尾は定義されていません。しかし、SQL ステートメントの結果としては順序が定義されます。

```
SELECT * FROM DEPARTMENT
  ORDER BY DEPTNO DESC
```

この例の場合、次のステートメントは DEPTNO の値が最も高い行にカーソルを位置付けします。

```
SELECT * FROM DEPARTMENT
  WHERE DEPTNO =
  (SELECT MAX(DEPTNO) FROM DEPARTMENT)
```

ただし、同じ値を持つ行がいくつかある場合には、カーソルはそれらのうちの最初の行に置かれることに注意してください。

以前に検索されたデータの更新

上方にスクロールして以前に検索されたデータを更新するには、 108ページの『すでに検索済みデータのスクロール』および 98ページの『検索されたデータの更新』で説明する技法を組み合わせて使用することができます。以下の 2 つの技法のいずれかを行うことができます。

1. 更新するデータ上に 2 番目のカーソルがあり、 SELECT ステートメントが制限されたエレメントをまったく使用していない場合には、カーソル制御 UPDATE ステートメントを使用できる。 2 番目のカーソルを、WHERE CURRENT OF 文節の中で指名してください。
2. それ以外の場合は、行の中のすべての値を指名するか、あるいは表の基本キーを指定する WHERE 文節を伴った UPDATE を使用する。 1 つのステートメントを、変数の異なった値について何度でも実行することができます。

例: UPDAT プログラム

UPDAT プログラムは動的 SQL を使用して、SAMPLE データベース内の STAFF 表にアクセスし、すべてのマネージャーを行員に変更します。それから、直前の作業単位をロールバックして、その変更を元に戻します。このサンプルは、以下のプログラム言語で入手可能です。

C	updat.sqc
Java	Updat.sqlj
COBOL	updat.sqb
REXX	updat.cmd

UPDAT プログラムの動作の仕組み

1. **SQLCA 構造を定義する。** INCLUDE SQLCA ステートメントは SQLCA 構造を定義し宣言します。また、その構造内のエレメントとして SQLCODE を定義します。SQLCA 構造の SQLCODE フィールドは、 SQL ステートメントおよびデータベース・マネージャー API 呼び出しの実行後に、データベース・マネージャーによりエラー情報を用いて更新されます。

   Java アプリケーションは、 SQLException オブジェクトに対して定義された方式によって SQLCODE および SQLSTATE にアクセスするので、同等の ″include SQLCA″ ステートメントは必要ありません。

   REXX アプリケーションには 1 つの SQLCA 構造のオカレンスがあります。これは SQLCA という名前で、アプリケーションが使用するために事前定義されているものです。アプリケーションの定義がなくてもこれを参照することができます。

2. **ホスト変数を宣言する。** BEGIN DECLARE SECTION および END DECLARE SECTION ステートメントは、ホスト変数宣言を区切ります。ホスト変数は、データベース・マネージャーとのデータの受け渡しのために使用されます。ホスト変数には、SQL ステートメントで参照される際に、接頭部としてコロン (:) が付けられます。

   Java および REXX アプリケーションは、ホスト変数を宣言する必要はありません。ただし (REXX の場合)、LOB ファイル参照変数およびロケーターは別です。ホスト変数のデータ・タイプおよびサイズは、変数の参照の実行時に決定されます。

3. **データベースに接続する。**プログラムは sample データベースに接続し、そのデータベースへの共用アクセスを要求します。 (START DATABASE MANAGER API 呼び出しまたは db2start コマンドが実行されていることを前提としています。) 共用アクセスを用いて同じデータベースに接続する他のプログラムにもアクセスが許可されます。

4. **UPDATE SQL ステートメントを実行する。** SQL ステートメントは、ホスト変数を使用して*静的に* 実行されます。 staff 表の job 列はホスト変数の値に設定され、job 列は Mgr の値を持ちます。

5. **DELETE SQL ステートメントを実行する。** SQL ステートメントは、ホスト変数を使用して*静的に* 実行されます。指定されたホスト変数 (jobUpdate/job-update/job_update) と同じ job 列の値を持つ行はすべて削除されます。

6. **INSERT SQL ステートメントを実行する。**列は STAFF 表に挿入されます。この挿入には、SQL ステートメントの実行より前に設定されたホスト変数を使用します。

7. **トランザクションを終了する。** ROLLBACK ステートメントを使って*作業単位* を終了します。以前に実行した SQL ステートメントは、COMMIT ステートメントを用いると永続的になり、 ROLLBACK ステートメントを用いると取り消されます。*作業単位* 内の SQL ステートメントはすべて影響を受けます。

CHECKERR マクロ / 関数は、プログラム外部にあるエラー検査ユーティリティーです。エラー検査ユーティリティーの所在は、ご使用のプログラム言語により異なります。

C
: DB2 API を呼び出す C プログラムの場合、 utilapi.c 内の sqlInfoPrint 関数は、 utilapi.h 内の API_SQL_CHECK として再定義されます。 C 組み込み SQL プログラムの場合、 utilemb.sqc 内の sqlInfoPrint 関数は、 utilemb.h 内の EMB_SQL_CHECK として再定義されます。

Java
: SQL エラーは SQLException としてスローされ、アプリケーションの catch ブロックで処理されます。

COBOL
: CHECKERR は checkerr.cbl という名前の外部プログラムです。

REXX
: CHECKERR は現行プログラムの終わりにあるプロシージャーです。

このエラー検査ユーティリティーのソース・コードについては、 125ページの『プログラム例での GET ERROR MESSAGE の使用』を参照してください。

C の例: UPDAT.SQC

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlenv.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;   1

int main(int argc, char *argv[])
{

   EXEC SQL BEGIN DECLARE SECTION;   2
      char statement[256];
      char userid[9];
      char passwd[19];
      char jobUpdate[6];
   EXEC SQL END DECLARE SECTION;

   printf( "¥nSample C program:  UPDAT ¥n");

   if (argc == 1)
   {
      EXEC SQL CONNECT TO sample;
  EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3)
   {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;  3
  EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else
   {
      printf ("¥nUSAGE: updat [userid passwd]¥n¥n");
      return 1;
   } /* endif */

   strcpy (jobUpdate, "Clerk");
   EXEC SQL UPDATE staff SET job = :jobUpdate WHERE job = 'Mgr';  4
   EMB_SQL_CHECK("UPDATE STAFF");
   printf ("All 'Mgr' have been demoted to 'Clerk'!¥n" );

   strcpy (jobUpdate, "Sales");
   EXEC SQL DELETE FROM staff WHERE job = :jobUpdate;  5
   EMB_SQL_CHECK("DELETE FROM STAFF");
   printf ("All 'Sales' people have been deleted!¥n");

   EXEC SQL INSERT INTO staff
      VALUES (999, 'Testing', 99, :jobUpdate, 0, 0, 0);   6
   EMB_SQL_CHECK("INSERT INTO STAFF");
   printf ("New data has been inserted¥n");
```

```
   EXEC SQL ROLLBACK; 7
   EMB_SQL_CHECK("ROLLBACK");
   printf( "On second thought -- changes rolled back.¥n" );

   EXEC SQL CONNECT RESET;
   EMB_SQL_CHECK("CONNECT RESET");
   return 0;
}
/* end of program : UPDAT.SQC */
```

Java の例: Updat.sqlj

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;


class Updat
{   static
  {   try
    {   Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
    }
    catch (Exception e)
    {   System.out.println ("¥n  Error loading DB2 Driver...¥n");
      System.out.println (e);
      System.exit(1);
    }
  }


  public static void main(String argv[])
  {   try
    {   System.out.println ("¥n  Java Updat Sample");

      String url = "jdbc:db2:sample";       // URL is jdbc:db2:dbname
      Connection con = null;

      // Set the connection                          3
      if (argv.length == 0)
      {   // connect with default id/password
        con = DriverManager.getConnection(url);
      }
      else if (argv.length == 2)
      {   String userid = argv[0];
        String passwd = argv[1];

        // connect with user-provided username and password
        con = DriverManager.getConnection(url, userid, passwd);
      }
      else
      {   throw new Exception("¥nUsage: java Updat [username password]¥n");
      }

      // Set the default context
      DefaultContext ctx = new DefaultContext(con);
      DefaultContext.setDefaultContext(ctx);

      // Enable transactions
      con.setAutoCommit(false);

      // UPDATE/DELETE/INSERT
      try
      {   String jobUpdate = null;

        jobUpdate="Clerk";
```

```
        #sql {UPDATE staff SET job = :jobUpdate WHERE job = 'Mgr'};   4
        System.out.println("¥nAll 'Mgr' have been demoted to 'Clerk'!");

        jobUpdate="Sales";
        #sql {DELETE FROM staff WHERE job = :jobUpdate};
        System.out.println("All 'Sales' people have been deleted!");  5

        #sql {INSERT INTO staff
          VALUES (999, 'Testing', 99, :jobUpdate, 0, 0, 0)};  6
        System.out.println("New data has been inserted");
      }
      catch( Exception e )
      {   throw e;
      }
      finally
      {   // Rollback the transaction
        System.out.println("¥nRollback the transaction...");
        #sql { ROLLBACK };                                           7
        System.out.println("Rollback done.");
      }
    }
    catch (Exception e)
    {   System.out.println (e);
    }
  }
}
```

COBOL の例: UPDAT.SQB

```
       Identification Division.
       Program-ID. "updat".

       Data Division.
       Working-Storage Section.

           copy "sql.cbl".
           copy "sqlenv.cbl".
           copy "sqlca.cbl".                                      1

           EXEC SQL BEGIN DECLARE SECTION END-EXEC.               2
       01 statement         pic x(80).
       01 userid            pic x(8).
       01 passwd.
         49 passwd-length   pic s9(4) comp-5 value 0.
         49 passwd-name     pic x(18).
       01 job-update        pic x(5).
           EXEC SQL END DECLARE SECTION END-EXEC.

      * Local variables
       77 errloc          pic x(80).
       77 error-rc        pic s9(9) comp-5.
       77 state-rc        pic s9(9) comp-5.

      * Variables for the GET ERROR MESSAGE API
      * Use application specific bound instead of BUFFER-SZ
       77 buffer-size     pic s9(4) comp-5 value 1024.
       77 line-width      pic s9(4) comp-5 value 80.
       77 error-buffer    pic x(1024).
       77 state-buffer    pic x(1024).

       Procedure Division.
       Main Section.
           display "Sample COBOL program:  UPDAT".

           display "Enter your user id (default none): "
                with no advancing.
           accept userid.

           if userid = spaces
             EXEC SQL CONNECT TO sample END-EXEC
           else
             display "Enter your password : " with no advancing
             accept passwd-name.

      * Passwords in a CONNECT statement must be entered in a VARCHAR format
      * with the length of the input string.
           inspect passwd-name tallying passwd-length for characters
              before initial " ".

           EXEC SQL CONNECT TO sample USER :userid USING :passwd       3
               END-EXEC.
           move "CONNECT TO" to errloc.
```

```
           call "checkerr" using SQLCA errloc.

           move "Clerk" to job-update.
           EXEC SQL UPDATE staff SET job=:job-update                     4
                    WHERE job='Mgr' END-EXEC.
           move "UPDATE STAFF" to errloc.
           call "checkerr" using SQLCA errloc.

           display "All 'Mgr' have been demoted to 'Clerk'!".

           move "Sales" to job-update.
           EXEC SQL DELETE FROM staff WHERE job=:job-update END-EXEC.    5
           move "DELETE FROM STAFF" to errloc.
           call "checkerr" using SQLCA errloc.

           display "All 'Sales' people have been deleted!".

           EXEC SQL INSERT INTO staff VALUES (999, 'Testing', 99,        6
                    :job-update, 0, 0, 0) END-EXEC.
           move "INSERT INTO STAFF" to errloc.
           call "checkerr" using SQLCA errloc.

           display "New data has been inserted".

           EXEC SQL ROLLBACK END-EXEC.                                   7
           move "ROLLBACK" to errloc.
           call "checkerr" using SQLCA errloc.

           DISPLAY "On second thought -- changes rolled back."

           EXEC SQL CONNECT RESET END-EXEC.
           move "CONNECT RESET" to errloc.
           call "checkerr" using SQLCA errloc.

       End-Prog.
           stop run.
```

REXX の例: UPDAT.CMD

注: REXX プログラムは静的 SQL を使用できません。このプログラムは動的 SQL で記述されています。

```
/* REXX program UPDAT.CMD */

parse version rexxType .
parse source platform .

if platform == 'AIX/6000' & rexxType == 'REXXSAA' then
do
  rcy = SysAddFuncPkg("db2rexx")
end
else
do
  if RxFuncQuery('SQLDBS') <> 0 then
    rcy = RxFuncAdd( 'SQLDBS',  'db2ar', 'SQLDBS'  )

  if RxFuncQuery('SQLEXEC') <> 0 then
    rcy = RxFuncAdd( 'SQLEXEC', 'db2ar', 'SQLEXEC' )
end

/* pull in command line arguments */
parse arg userid passwd .

/* check to see if the proper number of arguments have been passed in */
   PARSE ARG dbname userid password .
   if ((dbname = "" ) | ,
       (userid <> "" & password = "") ,
      ) then do
      SAY "USAGE: updat.cmd <dbname> [<userid> <password>]"

       exit -1
   end

   /* connect to database */
   SAY
   SAY 'Connect to' dbname
   IF password= "" THEN
      CALL SQLEXEC 'CONNECT TO' dbname
   ELSE
      CALL SQLEXEC 'CONNECT TO' dbname 'USER' userid 'USING' password

   CALL CHECKERR 'Connect to '
   SAY "Connected"

say 'Sample REXX program: UPDAT.CMD'

jobupdate = "'Clerk'"
st = "UPDATE staff SET job =" jobupdate "WHERE job = 'Mgr'"
call SQLEXEC 'EXECUTE IMMEDIATE :st'  4
call CHECKERR 'UPDATE'
say "All 'Mgr' have been demoted to 'Clerk'!"
```

```
jobupdate = "'Sales'"
st = "DELETE FROM staff WHERE job =" jobupdate
call SQLEXEC 'EXECUTE IMMEDIATE :st'  5
call CHECKERR 'DELETE'
say "All 'Sales' people have been deleted!"

st = "INSERT INTO staff VALUES (999, 'Testing', 99," jobupdate ", 0, 0, 0)"
call SQLEXEC 'EXECUTE IMMEDIATE :st'  6
call CHECKERR 'INSERT'
say 'New data has been inserted'

call SQLEXEC 'ROLLBACK'  7
call CHECKERR 'ROLLBACK'
say 'On second thought...changes rolled back.'

call SQLEXEC 'CONNECT RESET'
call CHECKERR 'CONNECT RESET'


CHECKERR:
  arg errloc

  if  ( SQLCA.SQLCODE = 0 ) then
    return 0
  else do
    say '--- error report ---'
    say 'ERROR occurred :' errloc
    say 'SQLCODE :' SQLCA.SQLCODE

    /******************************¥
    * GET ERROR MESSAGE API called *
    ¥******************************/
    call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
    say errmsg
    say '--- end error report ---'

    if (SQLCA.SQLCODE < 0 ) then
      exit
    else do
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
return 0
```

診断処理と SQLCA 構造

SQL ステートメントを発行し、データベース・マネージャー API を呼び出すアプリケーションには、戻りコードと SQLCA 構造を検査しエラー条件を正確に調べることが求められます。

戻りコード

ほとんどのデータベース・マネージャーは処理が正常であれば、ゼロの戻りコードを戻します。一般に、ゼロ以外の戻りコードは、 2 次エラー処理機構の SQLCA 構造が破壊された可能性があることを示します。この場合、呼び出された API は実行されません。 SQLCA 構造が破壊される原因として、構造に対して無効なアドレスを渡したことが考えられます。

SQLCODE および SQLSTATE

エラー情報は、SQLCA 構造の SQLCODE と SQLSTATE のフィールドに戻されます。SQLCA 構造は、すべての実行可能 SQL ステートメントとほとんどのデータベース・マネージャー API 呼び出しの実行後に更新されます。

実行可能 SQL ステートメントが入っているソース・ファイルには、 sqlca という名前を持つ SQLCA 構造が少なくとも 1 つあります。 SQLCA 構造は SQLCA インクルード・ファイルで定義されます。組み込み SQL ステートメントはないがデータベース・マネージャー API を呼び出すソース・ファイルには、 1 つまたは複数の SQLCA 構造を組み込むことができますが、各構造の名前は任意に付けられます。

ご使用のアプリケーションが FIPS 127-2 標準に準拠している場合、 SQLCA 構造の代わりに SQLSTATE および SQLCODE をホスト変数として宣言することができます。この方法に関しては、C または C++ アプリケーションの場合は 652ページの『C および C++ における SQLSTATE および SQLCODE 変数』、 COBOL アプリケーションの場合は 721ページの『COBOL での SQLSTATE および SQLCODE 変数』、または FORTRAN アプリケーションの場合は 736ページの『FORTRAN の SQLSTATE および SQLCODE 変数』を参照してください。

SQLCODE 値が 0 である場合は、正常に実行されたことを示します (SQLWARN 警告状態を伴うこともあります)。正の値は、ステートメントは正常に実行されたが、ホスト変数の切り捨てなどの警告を伴うことを意味します。負の値は、エラー状態が起こったことを意味します。

追加のフィールド SQLSTATE には、他の IBM データベース製品および SQL92 準拠のデータベース・マネージャーと一貫性がある標準化エラー・コードが含まれています。実際には、SQLSTATE は多くのデータベース・マネージャーと共通であるため、可搬性を考慮する場合は SQLSTATE を使用します。

SQLWARN フィールドには、SQLCODE がゼロの場合でも警告標識の配列が含まれます。 SQLWARN 配列の第 1 エレメントである SQLWARN0 には、その他のエレメントがすべてブランクである場合はブランクが入ります。その他のエレメントの少なくとも 1 つに警告文字が含まれている場合は、 SQLWARN0 には `W` が入ります。

SQLCA 構造の詳細については *管理 API 解説書*、 SQLCODE と SQLSTATE のエラー状態のリストについては、 *メッセージ解説書* を参照してください。

注: さまざまな IBM RDBMS サーバーにアクセスするアプリケーションを開発する場合は、以下のようにしてください。

- 可能な時点で、アプリケーションが SQLCODE ではなく SQLSTATE を検査するようにする。
- アプリケーションが DB2 コネクトを使用する場合は、 DB2 コネクトに付属しているマッピング機能を用いて、異なるデータベース間の SQLCODE 変換をマップする。

SQLCA 構造におけるトークンの切り捨て

SQLCA 構造ではトークンが切り捨てられることがあるため、診断目的ではトークン情報を使用しないでください。表および列の名前は最高 128 バイトの長さで定義できますが、 SQLCA トークンは、17 バイトと切り捨て終止符 (>) を加えた長さで切り捨てられます。アプリケーションの論理は、 `sqlerrmc` フィールドの実際の値によって決めるべきではありません。 SQLCA 構造およびトークンの切り捨ての説明については、*SQL 解説書* を参照してください。

WHENEVER ステートメントを用いたエラー処理

WHENEVER ステートメントがあると、プリコンパイラーは、実行中にエラー、警告、または行が見つからないなどの状態が起こった場合にアプリケーションに指定のラベルまで進むように指示するソース・コードを生成します。 WHENEVER ステートメントは、別の WHENEVER ステートメントが状況を変更するまで、後続の実行可能 SQL ステートメントに影響を与えます。

WHENEVER ステートメントには以下の 3 つの基本形式があります。

```
EXEC SQL WHENEVER SQLERROR   action
EXEC SQL WHENEVER SQLWARNING action
EXEC SQL WHENEVER NOT FOUND  action
```

上記のステートメントについて説明します。

SQLERROR

SQLCODE `< 0` である状態を示します。

SQLWARNING

SQLWARN(0) `= W` または SQLCODE `> 0` ですが、 `100` ではない状態を示します。

NOT FOUND
SQLCODE = 100 である状態を識別します。

いずれの場合も、*action* は以下のどちらかになります。

CONTINUE
アプリケーションの次の命令を続行するよう指示します。

GO TO *label*
GO TO の後に指定されたラベルの直後のステートメントに進むように指示します。 (GO TO は、2 つの語とすることも、GOTO として 1 つの語とすることもできます。)

WHENEVER ステートメントを使用しない場合、実行中にエラー、警告、または例外状態が起こると、デフォルトのアクションとして処理が継続されることになります。

WHENEVER ステートメントは、影響を及ぼす SQL ステートメントの前に指定しなければなりません。そうしないと、プリコンパイラーは実行可能 SQL ステートメント用に余分のエラー処理コードを生成しなければならないことを認識しません。 3 つの基本形式はいつでも任意に組み合わせて活動状態にすることができます。これら 3 つの形式の宣言順序は重要ではありません。無限ループ状態を避けるには、SQL ステートメントをハンドラー内部で実行する前に、 WHENEVER 処理が取り消されていることを確認してください。 WHENEVER SQLERROR CONTINUE ステートメントを使用すれば、SQL ステートメントをハンドラー内部で実行することができます。

WHENEVER ステートメントの詳細な説明については、 *SQL 解説書* を参照してください。

例外、シグナル、割り込みハンドラーについての考慮事項

例外、シグナル、または割り込みハンドラーは、例外が起きたときに制御を獲得するルーチンです。ここで適用されるハンドラーのタイプは、操作環境によって異なります。以下のとおりです。

Windows 32 ビット・オペレーティング・システム
`Ctrl-C` または `Ctrl-Break` を押すことにより、割り込みが生成されます。

OS/2
`Ctrl-C` または `Ctrl-Break` を押すことにより、オペレーティング・システム例外が生成されます。

UNIX
通常、`Ctrl-C` を押すと SIGINT 割り込みシグナルが生成されます。キーボードは簡単に再定義できるため、 SIGINT はマシン上のさまざまなキー・シーケンスで生成される場合があることに注意してください。

上記のリストにないオペレーティング・システムについては、 *アプリケーション構築の手引き* を参照してください。

例外、シグナル、および割り込みハンドラーの中には SQL ステートメントを置かないでください (COMMIT と ROLLBACK は例外)。これらのエラー状態が起きた場合には、データの矛盾を避けるために、 ROLLBACK するのが普通です。

例外 / シグナル / 割り込みハンドラーでの COMMIT および ROLLBACK のコーディングは、慎重に行ってください。これらのステートメントのいずれかをそれだけで呼び出す場合、 COMMIT または ROLLBACK は現行の SQL ステートメントが完了するまで実行されません (SQL ステートメントが実行中の場合)。これは、Ctrl-C ハンドラーから実行するには望ましい動作ではありません。

ROLLBACK を発行する前に、 INTERRUPT API (sqleintr/sqlgintr) を呼び出すことで解決できます。これは、現行の SQL 照会を中断させ (アプリケーションが SQL 照会を実行中の場合)、 ROLLBACK が即時に開始されるようにします。 ROLLBACK よりも COMMIT を実行するつもりならば、現行のコマンドを中断する必要はありません。

APPC を使用してリモート・データベース・サーバー (DB2 (AIX 版) または DB2 コネクトを使用したホスト・データベース・システム) にアクセスする場合に、アプリケーションは SIGUSR1 シグナルを受信することがあります。このシグナルは、リカバリー不可能エラーが発生したり SNA 接続が停止したときに、 SNA サービス/6000 により生成されます。 SIGUSR1 を処理するには、シグナル・ハンドラーをアプリケーションにインストールする必要があります。

さまざまなハンドラーの特定な詳細の考慮事項については、ご使用のプラットフォームの資料を参照してください。

出口リスト・ルーチンに関する考慮事項

出口リスト・ルーチンでは、SQL や DB2 API 呼び出しを使用しないでください。出口ルーチンの中ではデータベースから切断することはできないことに注意してください。

プログラム例での GET ERROR MESSAGE の使用

126ページの『C の例: UTILAPI.C』および 129ページの『COBOL の例: CHECKERR.CBL』で示しているコード・クリップは、渡される SQLCA に関する情報を GET ERROR MESSAGE API を使って得る方法について説明しています。

これらの例の作成に関する情報は、 README ファイルまたはこれらのサンプル・プログラムのヘッダー・セクションで見ることができます。

C の例: UTILAPI.C

```
#include <stdio.h>
#include <stdlib.h>
#include <sql.h>
#include <sqlenv.h>
#include <sqlda.h>
#include <sqlca.h>
#include <string.h>
#include <ctype.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;

/*#############################################################################
**    1. SQL_CHECK section
**
**        1.1 - SqlInfoPrint - prints on the screen everything that
**                             goes unexpected.
**        1.2 - TransRollback - rolls back the transaction
#############################################################################*/

/*****************************************************************************
**        1.1 - SqlInfoPrint - prints on the screen everything that
**                             goes unexpected.
*****************************************************************************/
int SqlInfoPrint( char *         appMsg,
                  struct sqlca * pSqlca,
                  int            line,
                  char *         file )
{   int   rc = 0;

    char  sqlInfo[1024];
    char  sqlInfoToken[1024];

    char  sqlstateMsg[1024];
    char  errorMsg[1024];


    if (pSqlca->sqlcode != 0 && pSqlca->sqlcode != 100)
    {   strcpy(sqlInfo, "");

        if( pSqlca->sqlcode < 0)
        {   sprintf( sqlInfoToken, "¥n---- error report ----¥n");
            strcat( sqlInfo, sqlInfoToken);
        }
        else
        {   sprintf( sqlInfoToken, "¥n---- warning report ----¥n");
            strcat( sqlInfo, sqlInfoToken);
        } /* endif */

        sprintf( sqlInfoToken, "  app. message      = %s¥n", appMsg);
        strcat( sqlInfo, sqlInfoToken);
        sprintf( sqlInfoToken, "  line              = %d¥n", line);
        strcat( sqlInfo, sqlInfoToken);
```

```
        sprintf( sqlInfoToken, "  file              = %s¥n", file);
        strcat( sqlInfo, sqlInfoToken);
        sprintf( sqlInfoToken, "  SQLCODE           = %ld¥n", pSqlca->sqlcode);
        strcat( sqlInfo, sqlInfoToken);

        /* get error message */
        rc = sqlaintp( errorMsg, 1024, 80, pSqlca);
        /* return code is the length of the errorMsg string */
        if( rc > 0)
        {   sprintf( sqlInfoToken, "%s¥n", errorMsg);
            strcat( sqlInfo, sqlInfoToken);
        }

        /* get SQLSTATE message */
        rc = sqlogstt( sqlstateMsg, 1024, 80, pSqlca->sqlstate);
        if (rc == 0)
        {   sprintf( sqlInfoToken, "%s¥n", sqlstateMsg);
            strcat( sqlInfo, sqlInfoToken);
        }


        if( pSqlca->sqlcode < 0)
        {   sprintf( sqlInfoToken, "--- end error report ---¥n");
            strcat( sqlInfo, sqlInfoToken);

            printf("%s", sqlInfo);
            return 1;
        }
        else
        {   sprintf( sqlInfoToken, "--- end warning report ---¥n");
            strcat( sqlInfo, sqlInfoToken);

            printf("%s", sqlInfo);
            return 0;
        } /* endif */
    } /* endif */

    return 0;
}

/******************************************************************************
**        1.2 - TransRollback - rolls back the transaction
******************************************************************************/
void TransRollback( )
{   int         rc = 0;

    /* rollback the transaction */
    printf( "¥nRolling back the transaction ...¥n") ;
    EXEC SQL ROLLBACK;
    rc = SqlInfoPrint( "ROLLBACK", &sqlca, __LINE__, __FILE__);
    if( rc == 0)
    {   printf( "The transaction was rolled back.¥n") ;
    }

}
```

Java の例: SQLException のキャッチ

JDBC および SQLJ アプリケーションは、 SQL 処理中にエラーが発生すると、 SQLException をスローします。使用中のアプリケーションは SQLException を受け取り、以下のコードでそれを表示します。

```
try {
   Statement stmt = connection.createStatement();
   int rowsDeleted = stmt.executeUpdate(
      "DELETE FROM employee WHERE empno = '000010'");
   System.out.println( rowsDeleted + " rows were deleted");
}

catch (SQLException sqle) {
   System.out.println(sqle);
}
```

SQLExceptions 処理の詳細については、 659ページの『Java の SQLSTATE および SQLCODE 値』を参照してください。

COBOL の例: CHECKERR.CBL

```
 Identification Division.
 Program-ID. "checkerr".

 Data Division.
 Working-Storage Section.

 copy "sql.cbl".

* Local variables
 77 error-rc        pic s9(9) comp-5.
 77 state-rc        pic s9(9) comp-5.

* Variables for the GET ERROR MESSAGE API
* Use application specific bound instead of BUFFER-SZ
* 77 buffer-size     pic s9(4) comp-5 value BUFFER-SZ.
* 77 error-buffer    pic x(BUFFER-SZ).
* 77 state-buffer    pic x(BUFFER-SZ).
 77 buffer-size     pic s9(4) comp-5 value 1024.
 77 line-width      pic s9(4) comp-5 value 80.
 77 error-buffer    pic x(1024).
 77 state-buffer    pic x(1024).

 Linkage Section.
 copy "sqlca.cbl" replacing ==VALUE "SQLCA   "== by == ==
                            ==VALUE 136==        by == ==.
 01 errloc          pic x(80).

 Procedure Division using sqlca errloc.
 Checkerr Section.
     if SQLCODE equal 0
        go to End-Checkerr.

     display "--- error report ---".
     display "ERROR occurred : ", errloc.
     display "SQLCODE : ", SQLCODE.

********************************
* GET ERROR MESSAGE API called *
********************************
     call "sqlgintp" using
                           by value     buffer-size
                           by value     line-width
                           by reference sqlca
                           by reference error-buffer
                     returning error-rc.

************************
* GET SQLSTATE MESSAGE *
************************
     call "sqlggstt" using
                           by value     buffer-size
                           by value     line-width
                           by reference sqlstate
```

```
                                by reference state-buffer
                          returning state-rc.

           if error-rc is greater than 0
              display error-buffer.

           if state-rc is greater than 0
              display state-buffer.

           if state-rc is less than 0
              display "return code from GET SQLSTATE =" state-rc.

           if SQLCODE is less than 0
              display "--- end error report ---"
              go to End-Prog.

           display "--- end error report ---"
           display "CONTINUING PROGRAM WITH WARNINGS!".
       End-Checkerr. exit program.
       End-Prog. stop run.
```

REXX の例: CHECKERR プロシージャー

```
parse version rexxType .
parse source platform .

if platform == 'AIX/6000' & rexxType == 'REXXSAA' then
do
  rcy = SysAddFuncPkg("db2rexx")
end
else
do
  if RxFuncQuery('SQLDBS') <> 0 then
    rcy = RxFuncAdd( 'SQLDBS',  'db2ar', 'SQLDBS'  )

  if RxFuncQuery('SQLEXEC') <> 0 then
    rcy = RxFuncAdd( 'SQLEXEC', 'db2ar', 'SQLEXEC' )
end

⋮

call CHECKERR 'INSERT'

⋮

CHECKERR:
  arg errloc

  if  ( SQLCA.SQLCODE = 0 ) then
    return 0
  else do
    say '--- error report ---'
    say 'ERROR occurred :' errloc
    say 'SQLCODE :' SQLCA.SQLCODE

    /******************************¥
    * GET ERROR MESSAGE API called *
    ¥******************************/
    call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
    say errmsg
    say '--- end error report ---'

    if (SQLCA.SQLCODE < 0 ) then
      exit
    else do
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
return 0

/* this variable (SYSTEM) must be user defined */
SYSTEM = AIX
if SYSTEM = OS2 then do
  if RxFuncQuery('SQLDBS') <> 0 then
    rcy = RxFuncAdd( 'SQLDBS',  'DB2AR', 'SQLDBS'  )
```

```
  if RxFuncQuery('SQLEXEC') <> 0 then
    rcy = RxFuncAdd( 'SQLEXEC', 'DB2AR', 'SQLEXEC' )
end

if SYSTEM = AIX then
  rcy = SysAddFuncPkg("db2rexx")

⋮

call CHECKERR 'INSERT'

⋮

CHECKERR:
  arg errloc

  if  ( SQLCA.SQLCODE = 0 ) then
    return 0
  else do
    say '--- error report ---'
    say 'ERROR occurred :' errloc
    say 'SQLCODE :' SQLCA.SQLCODE

    /*****************************¥
    * GET ERROR MESSAGE API called *
    ¥*****************************/
    call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
    say errmsg
    say '--- end error report ---'

    if (SQLCA.SQLCODE < 0 ) then
      exit
    else do
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
return 0
```

第5章 動的 SQL プログラムの作成

動的 SQL の使用目的 133
動的 SQL サポート・ステートメント . . 133
動的 SQL と静的 SQL との比較. . . . 134
PREPARE、DESCRIBE、FETCH、および SQLDA の使用. 137
カーソルの宣言および使用. 138
例: 動的 SQL プログラム 140
DYNAMIC プログラムの動作の仕組み 140
C の例: DYNAMIC.SQC 142
Java の例: Dynamic.java 144
COBOL の例: DYNAMIC.SQB 146
REXX の例: DYNAMIC.CMD 148
SQLDA の宣言. 150
最小の SQLDA 構造を用いたステートメントの準備 151
十分な数の SQLVAR 項目を指定した SQLDA の割り振り 152
SELECT ステートメントの記述 153
行を保持するためのストレージの獲得 . . 154
カーソルの処理 154
SQLDA 構造の割り振り. 155
SQLDA 構造を使用するデータの受け渡し 158
対話式 SQL ステートメントの処理 . . . 159
ステートメントのタイプの判別 . . . 159
可変リスト SELECT ステートメント 160
エンド・ユーザーからの SQL 要求の保管 160
例: ADHOC プログラム. 161
ADHOC プログラムの動作の仕組み 161
C の例: ADHOC.SQC 164
動的 SQL への変数入力 168
パラメーター・マーカーの使用 168
例: VARINP プログラム 169
VARINP プログラムの動作の仕組み 169
C の例: VARINP.SQC 171
Java の例: Varinp.java 173
COBOL の例: VARINP.SQB 175
DB2 コール・レベル・インターフェース (CLI) 177
組み込み SQL と DB2 CLI との比較 . . 177
DB2 CLI を使用する場合の利点 178
組み込み SQL か DB2 CLI かの決定 . . 180

動的 SQL の使用目的

動的 SQL は以下の場合に使用します。

- アプリケーションの実行時に SQL ステートメントの全体または一部を生成する必要がある。
- SQL ステートメントが参照するオブジェクトが、プリコンパイル時には存在しない。
- 現在のデータベース統計に基づき、常に最適なアクセス・パスをステートメントで使用したい。
- ステートメントのコンパイル環境を変更する (つまり、特殊レジスターを試してみる)。

動的 SQL サポート・ステートメント

動的 SQL サポート・ステートメントは、文字ストリング・ホスト変数とステートメント名を引き数として受け入れます。ホスト変数には、テキスト形式で動的に処理される SQL ステートメントが入っています。ステートメント・テキストは、アプリケーションのプリコンパイル時には処理されません。実際、ステートメント・テキストは、アプリケーションのプリコンパイル時には必要ありません。その代わりに、SQL ステートメン

© Copyright IBM Corp. 1993, 2001

トはプリコンパイルの段階でホスト変数と見なされ、その変数はアプリケーションを実行する間参照されます。このような SQL ステートメントを*動的* SQL と呼びます。

動的 SQL サポート・ステートメントでは SQL テキストが入っているホスト変数を実行可能形式に変換し、ステートメント名を参照して操作しなければなりません。それらのステートメントは以下のとおりです。

EXECUTE IMMEDIATE
ホスト変数を使用しないステートメントを準備し実行します。アプリケーション内のすべての EXECUTE IMMEDIATE ステートメントは、実行時には同じ場所にキャッシュされるため、最後のステートメントのみが認識されます。このステートメントは PREPARE および EXECUTE ステートメントの代用として使います。

PREPARE
SQL ステートメントの文字ストリング式をステートメントの実行可能書式に変換し、ステートメント名を割り当て、オプションでそのステートメントに関する情報を SQLDA 構造に入れます。

EXECUTE
前に準備した SQL ステートメントを実行します。このステートメントは、接続内で繰り返し実行することができます。

DESCRIBE
準備済みステートメントに関する情報を SQLDA に入れます。

アプリケーションは、ほとんどの SQL ステートメントを動的に実行することができます。サポートされている SQL ステートメントの全リストについては、 759ページの表38 を参照してください。

注: 動的 SQL ステートメントの内容は、静的 SQL ステートメントの場合と同じ構文に従っていますが、以下の点が異なります。
- 注釈は使用できない。
- ステートメントの先頭に EXEC SQL を使用してはならない。
- ステートメントをステートメント終了文字で終了してはならない。これに対する例外は CREATE TRIGGER ステートメントで、このステートメントの場合はセミコロン (;) を入れることができます。

動的 SQL と静的 SQL との比較

パフォーマンスのために静的 SQL または動的 SQL のどちらを使用するかは、プログラマーにとって常に関心の高い問題です。もちろんこれは、ユーザーの状態に基づいて決定されます。静的 SQL と動的 SQL のどちらを使用するかを決めるときには、 135ページの表6 を参考にしてください。 セキュリティーなどの問題は静的 SQL に影響を与えますし、 DB2 CLI を使用するか CLP を使用するかなどの環境の問題は、動的 SQL に影響を与えます。

決定する際には、ある特定の状況下で静的 SQL または動的 SQL のどちらを選んだらよいかに関して、以下の提案を考慮してください。 次の表に「両方」とある場合は、静的 SQL と動的 SQL のどちらでも変わりがないことを示します。 この情報はあくまでも一般的な提案に過ぎないことに注意してください。どちらを選ぶにしても、ご使用のアプリケーションの本来の用途や作業環境を考慮に入れる必要があります。はっきりしない場合は、まずステートメントを静的 SQL としてプロトタイプ化してから、次に動的 SQL としてプロトタイプ化し、その違いを比較することが最善の方法です。

表 6. 静的 SQL と動的 SQL の比較

考慮事項	推奨 SQL
SQL ステートメントを実行する時間	
• 2 秒未満	• 静的
• 2～10 秒	• 両方
• 11 秒以上	• 動的
データの均一性	
• データ配分が均一	• 静的
• やや不均一	• 両方
• 非常に不均一な配分	• 動的
範囲 (<、>、BETWEEN、LIKE) 述部	
• 非常に少ない	• 静的
• 随時	• 両方
• 頻繁	• 動的
繰り返し実行	
• 何回も実行 (10 回以上)	• 両方
• 少数回実行 (10 回未満)	• 両方
• 1 回だけ実行	• 静的
照会の種類	
• ランダム	• 動的
• 永久的	• 両方
ランタイム環境 (DML/DDL)	
• トランザクション処理 (DML のみ)	• 両方
• 混合 (DML および DDL - DDL はパッケージに影響を及ぼす)	• 動的
• 混合 (DML および DDL - DDL はパッケージに影響を及ぼさない)	• 両方
RUNSTATS の使用頻度	
• 非常にまれ	• 静的
• 普通	• 両方
• 頻繁	• 動的

一般に、動的 SQL を使用したアプリケーションは、使用前に SQL ステートメントをコンパイルする必要があるため、 SQL ステートメント当たりの始動 (初期) コストが大きくなります。コンパイルを行うと、動的 SQL の実行時間は静的 SQL の場合と同じ

ですが、最適化プログラムがより優れたアクセス・プランを選択することによりもっと速くなることがあります。初期のコンパイル・コストは、動的ステートメントを実行するたびに低減されます。複数のユーザーが同じステートメントを用いて同じアプリケーションを実行しているなら、ステートメントを発行した最初のアプリケーションだけにステートメントのコンパイル・コストが生じます。

DML と DDL が混在している環境では、アプリケーションの実行中にシステムがステートメントを暗黙のうちに再コンパイルすることがあるため、動的 SQL ステートメントのコンパイル・コストが変わる可能性があります。混合環境では、静的 SQL と動的 SQL との間の選択はパッケージが無効にされる頻度にも影響してきます。 DDL によってパッケージが無効にされる場合は、動的 SQL のほうが便利でしょう。実際に実行する照会だけが次回の使用時に再コンパイルされ、その他の照会は再コンパイルされないからです。静的 SQL の場合は、いったん無効にされるとパッケージ全体が再バインドされます。

特定のアプリケーションが上記の特性の混合したものであり、一方の特性には静的 SQL が適しており、他方の特性には動的 SQL が適していることがあります。この場合、明確な決定方法はありませんので、最も慣れていて使用しやすい手法を用いてください。前の表の考慮事項は、重要性の高い順におおまかに掲載してあることに注意してください。

注: 静的および動的 SQL はそれぞれ、DB2 最適化プログラムにとって重要な 2 種類に分けられます。それらは以下のとおりです。

1. ホスト変数を含まない静的 SQL

   これは、以下の場合にしか見られない、まれな状態です。
   - *初期化* コード
   - 初心者トレーニング用の例

   実際これは、実行時のパフォーマンスのオーバーヘッドがなく、 DB2 最適化プログラムの機能を十分に活用しているため、パフォーマンスの観点からは最善の組み合わせです。

2. ホスト変数を含む静的 SQL

   これは DB2 アプリケーションの従来の*継承* スタイルです。ステートメントのコンパイル中に獲得する PREPARE およびカタログ・ロックの実行時オーバーヘッドがなくなります。最適化プログラムは SQL ステートメント全体を認識しないため、残念ながら、最適化プログラムの全性能を活用することはできません。高度に均一化されていないデータ配分の場合には、特殊な問題があります。

3. パラメーター・マーカーを含まない動的 SQL

   これはランダム照会インターフェース (CLP など) 用の標準的なスタイルで、最適化プログラムに好まれるタイプの SQL です。複雑な照会の場合は、 PREPARE ステ

ートメントにオーバーヘッドがあると実行時間が短縮されるという利点があります。パラメーター・マーカーに関する詳細については、 168ページの『パラメーター・マーカーの使用』を参照してください。

4. パラメーター・マーカーを含む動的 SQL

   これは CLI アプリケーション用の最も一般的なタイプの SQL です。主な利点は、パラメーター・マーカーがあるために、 PREPARE ステートメント (主に選択または挿入) を繰り返し実行するうちに、 PREPARE ステートメントのコストを償却できるという点です。この償却は、すべての反復性のある動的 SQL アプリケーションに当てはまります。残念なことに、ホスト変数を含む静的 SQL と同様に、 DB2 最適化プログラムの一部は、情報のすべてを利用することができないために作動しません。最も効率的な方法としては、*ホスト変数を指定して静的 SQL を使用*することまたはパラメーター・マーカーなしで*動的 SQL を使用*すること をお勧めします。

PREPARE、DESCRIBE、FETCH、および SQLDA の使用

静的 SQL を使用すると、組み込み SQL ステートメントで使用されているホスト変数は、アプリケーションのコンパイル時に認識されます。動的 SQL を使用した場合には、組み込み SQL ステートメントとその結果としてのホスト変数は、アプリケーションを実行するまで認識されません。このように、動的 SQL アプリケーションの場合は、アプリケーションで使用するホスト変数のリストを扱う必要があります。(PREPARE を使用して) 準備された SELECT ステートメントのホスト変数情報を得るためには、 DESCRIBE ステートメントを使用し、その情報を SQL 記述子域 (SQLDA) に保管することができます。

注: Java アプリケーションは SQLDA 構造を使用しないので、 PREPARE または DESCRIBE ステートメントも使用しません。 JDBC アプリケーションでは、 `PreparedStatement` オブジェクトと `executeQuery()` 方式を使って `ResultSet` オブジェクトを生成することができますが、これはホスト言語カーソルと同等です。 SQLJ アプリケーションでは、 SQLJ イテレーター・オブジェクトを `CursorByPos` または `CursorByName` カーソルと共に宣言し、 FETCH ステートメントからデータを戻すこともできます。

アプリケーション内で DESCRIBE ステートメントを実行すると、データベース・マネージャーはホスト変数を SQLDA に定義します。ホスト変数を SQLDA に定義すると、 FETCH ステートメントにより、カーソルを使用してホスト変数に値を割り当てることができます。

PREPARE、DESCRIBE、および FETCH ステートメントに関する詳細な情報、そして SQLDA の説明については *SQL 解説書* を参照してください。

PREPARE、DESCRIBE、および FETCH ステートメントを使用し、 SQLDA を使用しない単純な動的 SQL プログラムの例については、 140ページの『例: 動的 SQL プログラム』を参照してください。 PREPARE、DESCRIBE、および FETCH ステートメント

を使用し、対話式 SQL ステートメントを処理する SQLDA を使用する動的 SQL プログラムの例については、 161ページの『例: ADHOC プログラム』を参照してください。

カーソルの宣言および使用

カーソルを動的に処理することは、静的 SQL を用いてカーソルを処理することとほとんど同じです。カーソルを宣言する際に、カーソルは照会と関連付けられます。

静的 SQL の場合、照会は 88ページの『カーソル・ステートメントの宣言』 にあるようなテキスト形式の SELECT ステートメントです。

動的 SQL の場合、照会は PREPARE ステートメントで割り当てられたステートメント名と関連付けられます。参照したホスト変数はパラメーター・マーカーで表されます。表7 は、動的 SELECT ステートメントに関連付けられた DECLARE ステートメントを示しています。

表 7. 動的 SELECT に関連付けられた DECLARE ステートメント

言語	ソース・コード例

C/C++	`strcpy( prep_string, "SELECT tabname FROM syscat.tables"`
	`"WHERE tabschema = ?" );`
	`EXEC SQL PREPARE s1 FROM :prep_string;`
	`EXEC SQL DECLARE c1 CURSOR FOR s1;`
	`EXEC SQL OPEN c1 USING :host_var;`

Java (JDBC)	`PreparedStatement prep_string = ("SELECT tabname FROM syscat.tables`
	`WHERE tabschema = ?" );`
	`prep_string.setCursor("c1");`
	`prep_string.setString(1, host_var);`
	`ResultSet rs = prep_string.executeQuery();`

COBOL	`MOVE "SELECT TABNAME FROM SYSCAT.TABLES WHERE TABSCHEMA = ?"`
	`TO PREP-STRING.`
	`EXEC SQL PREPARE S1 FROM :PREP-STRING END-EXEC.`
	`EXEC SQL DECLARE C1 CURSOR FOR S1 END-EXEC.`
	`EXEC SQL OPEN C1 USING :host-var END-EXEC.`

FORTRAN	`prep_string = 'SELECT tabname FROM syscat.tables WHERE tabschema = ?'`
	`EXEC SQL PREPARE s1 FROM :prep_string`
	`EXEC SQL DECLARE c1 CURSOR FOR s1`
	`EXEC SQL OPEN c1 USING :host_var`

静的カーソルと動的カーソルの主な相違点は、静的カーソルがプリコンパイル時に準備されるのに対して、動的カーソルは実行時に準備されることです。さらに、照会で参照されるホスト変数はパラメーター・マーカーで表され、カーソルをオープンする際に実行時ホスト変数で置き換えられます。

カーソルの使用方法の詳細については、以下の節を参照してください。
- 87ページの『カーソルを用いた複数行の選択』

- 90ページの『例: カーソル・プログラム』
- 750ページの『REXX でのカーソルの使用』

例: 動的 SQL プログラム

このサンプル・プログラムは、動的 SQL ステートメントに基づくカーソル処理を示しています。このプログラムは、名前列に STAFF という値を持つ表を除く SYSCAT.TABLES のすべての表をリストします。このサンプルは、以下のプログラム言語で入手可能です。

C dynamic.sqc

Java Dynamic.java

COBOL dynamic.sqb

REXX dynamic.cmd

DYNAMIC プログラムの動作の仕組み

1. **ホスト変数を宣言する。**このセクションには次の 3 つのホスト変数宣言が含まれています。

   table_name
   FETCH ステートメントの実行中に戻されたデータを保持するために使用する。

   st テキスト形式で動的 SQL ステートメントを保持するために使用する。

   parm_var
   st のパラメーター・マーカーを置き換えるためにデータ値を与える。

2. **ステートメントを準備する。**パラメーター・マーカー ('?') が 1 つ入っている SQL ステートメントは、ホスト変数にコピーされます。このホスト変数は、妥当性検査を行うために PREPARE ステートメントに渡されます。 PREPARE ステートメントは SQL テキストを分析し、プリコンパイラーまたはバインダーと同じ方法でパッケージのアクセス・セクションを準備します。これは、プリコンパイル中ではなく、実行時にだけ行われます。

3. **カーソルを宣言する。** DECLARE ステートメントはカーソルを、動的に準備した SQL ステートメントに関連付けます。準備済み SQL ステートメントが SELECT ステートメントである場合、結果表から行を取り出すにはカーソルが必要です。

4. **カーソルをオープンする。** OPEN ステートメントは、結果表の第 1 行目より前を指すように、先に宣言しておいたカーソルを初期化します。 USING 文節は、準備済み SQL ステートメントのパラメーター・マーカーを置き換えるためのホスト変数を指定します。ホスト変数のデータ・タイプおよび長さは、関連する列のタイプおよび長さと互換性がなければなりません。

5. **データを取り出す。** FETCH ステートメントを使用して、 NAME 列を結果表から table_name ホスト変数に移動します。ホスト変数は、プログラムが別の行を取り出すためにループバックする前に印刷されます。

6. **カーソルをクローズする。** CLOSE ステートメントはカーソルをクローズし、そのカーソルに関連するリソースを解放します。

CHECKERR マクロ / 関数は、プログラム外部にあるエラー検査ユーティリティーです。エラー検査ユーティリティーの所在は、ご使用のプログラム言語により異なります。

C
: DB2 API を呼び出す C プログラムの場合、 utilapi.c 内の sqlInfoPrint 関数は、 utilapi.h 内の API_SQL_CHECK として再定義されます。 C 組み込み SQL プログラムの場合、 utilemb.sqc 内の sqlInfoPrint 関数は、 utilemb.h 内の EMB_SQL_CHECK として再定義されます。

Java
: SQL エラーは SQLException としてスローされ、アプリケーションの catch ブロックで処理されます。

COBOL
: CHECKERR は checkerr.cbl という名前の外部プログラムです。

REXX
: CHECKERR は現行プログラムの終わりにあるプロシージャーです。

このエラー検査ユーティリティーのソース・コードについては、 125ページの『プログラム例での GET ERROR MESSAGE の使用』を参照してください。

C の例: DYNAMIC.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION;
      char  table_name[19];
      char  st[80];  1
      char  parm_var[19];
      char userid[9];
      char passwd[19];
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: DYNAMIC¥n" );

   if (argc == 1) {
      EXEC SQL CONNECT TO sample;
      EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else {
      printf ("¥nUSAGE: dynamic [userid passwd]¥n¥n");
      return 1;
   } /* endif */

   strcpy( st, "SELECT tabname FROM syscat.tables" );
   strcat( st, " WHERE tabname <> ?" );
   EXEC SQL PREPARE s1 FROM :st;  2
   EMB_SQL_CHECK("PREPARE");

   EXEC SQL DECLARE c1 CURSOR FOR s1;  3

   strcpy( parm_var, "STAFF" );
   EXEC SQL OPEN c1 USING :parm_var;  4
   EMB_SQL_CHECK("OPEN");
   do {
      EXEC SQL FETCH c1 INTO :table_name;  5
      if (SQLCODE != 0) break;

      printf( "Table = %s¥n", table_name );
   } while ( 1 );

   EXEC SQL CLOSE c1;  6
   EMB_SQL_CHECK("CLOSE");
```

```
   EXEC SQL COMMIT;
   EMB_SQL_CHECK("COMMIT");

   EXEC SQL CONNECT RESET;
   EMB_SQL_CHECK("CONNECT RESET");
   return 0;
}
/* end of program : DYNAMIC.SQC */
```

Java の例: Dynamic.java

```
import java.sql.*;

class Dynamic
{   static
  {   try
    {   Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
    }
    catch (Exception e)
    {   System.out.println ("¥n  Error loading DB2 Driver...¥n");
      System.out.println (e);
      System.exit(1);
    }
  }

  public static void main(String argv[])
  {   try
    {   System.out.println ("  Java Dynamic Sample");
      // Connect to Sample database

      Connection con = null;
      // URL is jdbc:db2:dbname
      String url = "jdbc:db2:sample";

      if (argv.length == 0)
      {   // connect with default id/password
        con = DriverManager.getConnection(url);
      }
      else if (argv.length == 2)
      {   String userid = argv[0];
        String passwd = argv[1];

        // connect with user-provided username and password
        con = DriverManager.getConnection(url, userid, passwd);
      }
      else
      {   throw new Exception("¥nUsage: java Dynamic [username password]¥n");
      }

      // Enable transactions
      con.setAutoCommit(false);

      // Perform dynamic SQL SELECT using JDBC
      try
      {   PreparedStatement pstmt1 = con.prepareStatement(
          "SELECT tabname FROM syscat.tables " +
          "WHERE tabname <> ? " +
          "ORDER BY 1");  2
      // set cursor name for the positioned update statement
      pstmt1.setCursorName("c1");                                 3
      pstmt1.setString(1, "STAFF");
      ResultSet rs = pstmt1.executeQuery();                       4

      System.out.print("¥n");
```

```
      while( rs.next() )                                              5
      {   String tableName = rs.getString("tabname");
        System.out.println("Table = " + tableName);
      };

      rs.close();
      pstmt1.close();                                                 7
      }
      catch( Exception e )
      {   throw e;
      }
      finally
      {   // Rollback the transaction
        System.out.println("¥nRollback the transaction...");
        con.rollback();
        System.out.println("Rollback done.");
      }
    }
    catch( Exception e )
    {   System.out.println(e);
    }
  }
}
```

COBOL の例: DYNAMIC.SQB

```
       Identification Division.
       Program-ID. "dynamic".

       Data Division.
       Working-Storage Section.

           copy "sqlenv.cbl".
           copy "sql.cbl".
           copy "sqlca.cbl".

           EXEC SQL BEGIN DECLARE SECTION END-EXEC.
       01 table-name      pic x(20).
       01 st              pic x(80).                                1
       01 parm-var        pic x(18).
       01 userid            pic x(8).
       01 passwd.
         49 passwd-length   pic s9(4) comp-5 value 0.
         49 passwd-name     pic x(18).
           EXEC SQL END DECLARE SECTION END-EXEC.

       77 errloc          pic x(80).

       Procedure Division.
       Main Section.
           display "Sample COBOL program: DYNAMIC".

           display "Enter your user id (default none): "
                with no advancing.
           accept userid.

           if userid = spaces
             EXEC SQL CONNECT TO sample END-EXEC
           else
             display "Enter your password : " with no advancing
             accept passwd-name.

      * Passwords in a CONNECT statement must be entered in a VARCHAR format
      * with the length of the input string.
           inspect passwd-name tallying passwd-length for characters
              before initial " ".

           EXEC SQL CONNECT TO sample USER :userid USING :passwd
               END-EXEC.
           move "CONNECT TO" to errloc.
           call "checkerr" using SQLCA errloc.

           move "SELECT TABNAME FROM SYSCAT.TABLES
      -       " ORDER BY 1
      -       " WHERE TABNAME <> ?" to st.
           EXEC SQL PREPARE s1 FROM :st END-EXEC.                        2
           move "PREPARE" to errloc.
           call "checkerr" using SQLCA errloc.
```

```
    EXEC SQL DECLARE c1 CURSOR FOR s1 END-EXEC.                        3

    move "STAFF" to parm-var.
    EXEC SQL OPEN c1 USING :parm-var END-EXEC.                         4
    move "OPEN" to errloc.
    call "checkerr" using SQLCA errloc.

    perform Fetch-Loop thru End-Fetch-Loop
       until SQLCODE not equal 0.

    EXEC SQL CLOSE c1 END-EXEC.                                        6
    move "CLOSE" to errloc.
    call "checkerr" using SQLCA errloc.

    EXEC SQL COMMIT END-EXEC.
    move "COMMIT" to errloc.
    call "checkerr" using SQLCA errloc.

    EXEC SQL CONNECT RESET END-EXEC.
    move "CONNECT RESET" to errloc.
    call "checkerr" using SQLCA errloc.

End-Main.
    go to End-Prog.

Fetch-Loop Section.
    EXEC SQL FETCH c1 INTO :table-name END-EXEC.                       5
    if SQLCODE not equal 0
       go to End-Fetch-Loop.
    display "TABLE = ", table-name.
End-Fetch-Loop. exit.

End-Prog.
    stop run.
```

REXX の例: DYNAMIC.CMD

```
/* REXX DYNAMIC.CMD */

parse version rexxType .
parse source platform .

if platform == 'AIX/6000' & rexxType == 'REXXSAA' then
do
  rcy = SysAddFuncPkg("db2rexx")
end
else
do
  if RxFuncQuery('SQLDBS') <> 0 then
    rcy = RxFuncAdd( 'SQLDBS',  'db2ar', 'SQLDBS'  )

  if RxFuncQuery('SQLEXEC') <> 0 then
    rcy = RxFuncAdd( 'SQLEXEC', 'db2ar', 'SQLEXEC' )
end

/* pull in command line arguments */
parse arg userid passwd .

/* check to see if the proper number of arguments have been passed in */
   PARSE ARG dbname userid password .
   if ((dbname = "" ) | ,
       (userid <> "" & password = "") ,
      ) then do
      SAY "USAGE: dynamic.cmd <dbname> [<userid> <password>]"

       exit -1
   end

   /* connect to database */
   SAY
   SAY 'Connect to' dbname
   IF password= "" THEN
      CALL SQLEXEC 'CONNECT TO' dbname
   ELSE
      CALL SQLEXEC 'CONNECT TO' dbname 'USER' userid 'USING' password

   CALL CHECKERR 'Connect to '
   SAY "Connected"

say 'Sample REXX program: DYNAMIC'

st = "SELECT tabname FROM syscat.tables WHERE tabname <> ? ORDER BY 1"
call SQLEXEC 'PREPARE s1 FROM :st'   2
call CHECKERR 'PREPARE'

call SQLEXEC 'DECLARE c1 CURSOR FOR s1'   3
call CHECKERR 'DECLARE'

parm_var = "STAFF"
call SQLEXEC 'OPEN c1 USING :parm_var'   4
```

```
do while ( SQLCA.SQLCODE = 0 )
  call SQLEXEC 'FETCH c1 INTO :table_name'   5
  if (SQLCA.SQLCODE = 0) then
    say 'Table = ' table_name
end

call SQLEXEC 'CLOSE c1'   6
call CHECKERR 'CLOSE'

call SQLEXEC 'CONNECT RESET'
call CHECKERR 'CONNECT RESET'


CHECKERR:
  arg errloc

  if  ( SQLCA.SQLCODE = 0 ) then
    return 0
    else do
    say '--- error report ---'
    say 'ERROR occurred :' errloc
    say 'SQLCODE :' SQLCA.SQLCODE

    /******************************¥
    * GET ERROR MESSAGE API called *
    ¥******************************/
    call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
    say errmsg
    say '--- end error report ---'

    if (SQLCA.SQLCODE < 0 ) then
      exit
    else do
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
return 0
```

SQLDA の宣言

SQLDA には不定数の SQLVAR 項目が入っており、 図2 に示されているように、各 SQLVAR 項目は 1 データ行に 1 つの列を記述するフィールドの集まりで構成されます。 SQLVAR 項目には、基本 SQLVAR と 2 次 SQLVAR という 2 つのタイプがあります。その 2 つのタイプについては、 *SQL 解説書* を参照してください。

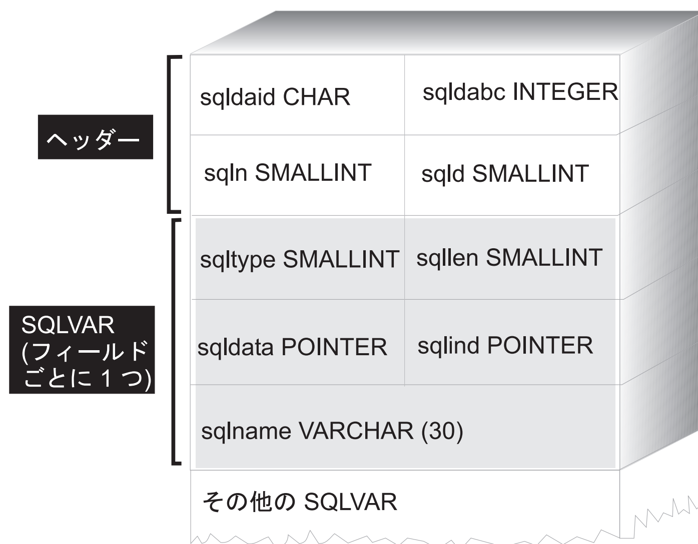


図 2. SQL 記述域 (SQLDA)

必要とされる SQLVAR 項目の数は結果表の列の数によって決まるため、アプリケーションは必要に応じて適切な数の SQLVAR エレメントを割り振ることができなければなりません。これは、以下の 2 とおりの方法で行うことができます。言及されている SQLDA のフィールドについては、 *SQL 解説書* を参照してください。

- 必要とされる最大の SQLDA (つまり、SQLVAR 項目の最大数を指定した SQLDA) を与える。結果表に戻すことができる列の最大数は 255 です。戻される列のいずれかが LOB タイプか独特なタイプの列である場合、SQLN の値は 2 倍になり、情報を入れるために必要な SQLVAR の数も 2 倍の 510 になります。しかし、SELECT ステートメントが 255 列を取り戻すことさえできないことが多いので、割り当てられたスペースの大部分が未使用となります。
- SQLVAR 項目を少なめに指定して、小さめの SQLDA を与える。この場合、結果表に SQLDA で使用できる SQLVAR 項目より多い列が入っていると、記述は戻されません。代わりに、データベース・マネージャーは SELECT ステートメントで検出される選択リスト項目の数を戻します。 アプリケーションは SQLDA に SQLVAR の

必要数を割り当ててから、 DESCRIBE ステートメントを使用して列記述を入手します。この方式の詳細については、『最小の SQLDA 構造を用いたステートメントの準備』を参照してください。

上記の方式の場合、最初にいくつの SQLVAR 項目を割り当てればよいかという疑問が生じます。各 SQLVAR エレメントは、最高 44 バイトのストレージを使用します (SQLDATA および SQLIND フィールドに割り振られるストレージはカウントしていません)。メモリーに十分余裕があれば、 SQLDA の最大サイズを割り振るという最初の方法を実行するのは簡単です。

より小さい SQLDA を割り当てるという 2 番目の方法は、メモリーの動的割り振りをサポートする C および C++ のようなプログラミング言語にしか適用できません。メモリーの動的割り振りをサポートしない COBOL や FORTRAN のような言語の場合、最初の方法を使用することが必要です。

最小の SQLDA 構造を用いたステートメントの準備

アプリケーションが、 SQLVAR 項目の入っていない `minsqlda` という名前の SQLDA 構造を宣言する場面を例にとって考えてみましょう。 SQLDA の SQLN フィールドは割り振られる SQLVAR 項目の数を記述します。この場合、SQLN は 0 に設定しなければなりません。次に、文字ストリング `dstring` から 1 つのステートメントを準備し、そしてその記述を `minsqlda` の中に入力するには、次の SQL ステートメントを発行します。 (C 構文を使用し、 `minsqlda` が SQLDA 構造へのポインターとして宣言されているものとします。)

```
EXEC SQL
  PREPARE STMT INTO :*minsqlda FROM :dstring;
```

`dstring` に含まれるステートメントが、各行に 20 列を戻す SELECT ステートメントであったとします。 PREPARE ステートメント (または DESCRIBE ステートメント) の後の SQLDA の SQLD フィールドには、準備済み SELECT ステートメントの結果行の列数が入っています。

SQLDA の SQLVAR は以下の場合に設定されます。

- SQLN >= SQLD であり、かつどの列も LOB または特殊タイプではない場合

  最初の SQLD SQLVAR 項目が設定され、SQLDOUBLED はブランクに設定されます。

- SQLN >= 2*SQLD であり、かつ少なくとも 1 つの列が LOB または特殊タイプである場合

  2* SQLD SQLVAR 項目が設定され、SQLDOUBLED は 2 に設定されます。

- SQLD <= SQLN < 2*SQLD であり、少なくとも 1 つの列が特殊タイプであり、かつ LOB の列はない場合

最初の SQLD SQLVAR 項目が設定され、SQLDOUBLED はブランクに設定されます。 SQLWARN バインド・オプションが YES の場合は、警告 SQLCODE +237 (SQLSTATE 01594) が発行されます。

以下の場合には、SQLDA 内の SQLVAR は設定されません (追加スペースを割り振り、DESCRIBE をもう一度指定するよう要求されます)。

- SQLN < SQLD であり、どの列も LOB または特殊タイプではない場合

  SQLVAR 項目は設定されず、SQLDOUBLED はブランクに設定されます。 SQLWARN バインド・オプションが YES の場合は、警告 SQLCODE +236 (SQLSTATE 01005) が発行されます。

  DESCRIBE を正常に実行するためには、SQLD SQLVAR を割り振ってください。
- SQLN < SQLD であり、少なくとも 1 つの列が特殊タイプであり、かつ LOB の列はない場合

  SQLVAR 項目は設定されず、SQLDOUBLED はブランクに設定されます。 SQLWARN バインド・オプションが YES の場合は、警告 SQLCODE +239 (SQLSTATE 01005) が発行されます。

  特殊タイプの名前を含む DESCRIBE を正常に実行するためには、 2*SQLD SQLVAR を割り振ってください。
- SQLN < 2*SQLD であり、かつ少なくとも 1 つの列が LOB である場合

  SQLVAR 項目は設定されず、SQLDOUBLED はブランクに設定されます。警告 SQLCODE +238 (SQLSTATE 01005) が発行されます (SQLWARN バインド・オプションの設定に関係なく)。

  DESCRIBE を正常に実行するためには、2*SQLD SQLVAR を割り振ってください。

BIND コマンドの SQLWARN オプションは、 DESCRIBE (または PREPARE...INTO) が以下の警告を戻すかどうかを制御します。
- SQLCODE +236 (SQLSTATE 01005)
- SQLCODE +237 (SQLSTATE 01594)
- SQLCODE +239 (SQLSTATE 01005)

アプリケーション・コードでは、これらの SQLCODE が戻される可能性のあることを常に考慮に入れておいてください。警告 SQLCODE +238 (SQLSTATE 01005) は、選択リストに LOB 列があり SQLDA に不適当な SQLVAR がある場合に、必ず戻されます。これは、結果セット内に LOB 列があるために SQLVAR の数が 2 倍になっていなければならないということをアプリケーションが認識できる唯一の方法です。

十分な数の SQLVAR 項目を指定した SQLDA の割り振り

結果表の列数が決まったら、ストレージを 2 番目の、フルサイズの SQLDA に割り振ることができます。たとえば、結果表に 20 列が含まれている (そのどれもが LOB 列でない) 場合、 2 番目の SQLDA 構造である `fulsqlda` は、少なくとも 20 の SQLVAR エレメント (または結果表に LOB または特殊タイプがある場合には 40 エレ

メント) を指定して割り振らなければなりません。この例のその他の部分では、LOB または特殊タイプは結果表に含まれないものとします。

SQLDA 構造のストレージ要件は以下のように構成されています。

- 長さ 16 バイトの固定長ヘッダー (SQLN および SQLD などのフィールドを含む)。
- SQLVAR 項目の可変長配列、それぞれのエレメントは長さが 44 バイト (32 ビット・プラットフォームの場合) または 56 バイト (64 ビット・プラットフォームの場合)。

fulsqlda に必要な SQLVAR 項目の数は、 minsqlda という SQLD フィールドに指定されました。その値は 20 です。そのため、この例で使用される fulsqlda に必要なストレージ割り振りは、以下のようになります。

```
16 + (20 * sizeof(struct sqlvar))
```

注: 64 ビット・プラットフォームの場合、 sizeof(struct sqlvar) および sizeof(struct sqlvar2) は 56 を戻します。 32 ビット・プラットフォームの場合、 sizeof(struct sqlvar) および sizeof(struct sqlvar2) は 44 を戻します。

この値は、ヘッダーのサイズに各 SQLVAR 項目のサイズの 20 倍を加えて、合計 896 バイトであることを表しています。

SQLDASIZE マクロを使用することにより、自分の計算をしないようにし、バージョン固有の従属関係をすべて回避することができます。

SELECT ステートメントの記述

fulsqlda に十分なスペースを割り振るためには、アプリケーションに以下のステップを含める必要があります。

1. fulsqlda の SQLN フィールドに値 20 を保管する。
2. 2 番目の SQLDA 構造 fulsqlda を用いて SELECT ステートメントに関する情報を入手する。これには、次の 2 とおりの方法があります。
   - minsqlda の代わりに fulsqlda を指定する別の PREPARE ステートメントを使用する。
   - fulsqlda を指定する DESCRIBE ステートメントを使用する。

DESCRIBE ステートメントを使用するとステートメントを 2 回準備するコストが省けるため、この方法のほうが好んで使用されます。 DESCRIBE ステートメントは、準備操作中に入手した前の情報を再度使用するだけで、その情報を新規の SQLDA 構造に入れます。次のステートメントを発行することができます。

```
EXEC SQL DESCRIBE STMT INTO :fulsqlda
```

このステートメントを実行すると、それぞれの SQLVAR エレメントには結果表の 1 つの列の記述が含まれます。

行を保持するためのストレージの獲得

SQLDA 構造を用いて結果表の任意の行を取り出すには、アプリケーションは事前に以下の処理を行っておく必要があります。

1. それぞれの SQLVAR 記述を分析して、その列の値に必要なスペースの量を判別する。

   SELECT が記述されている場合、ラージ・オブジェクト (LOB) の値について、SQLVAR に指定されるデータ・タイプは SQL_TYP_*x*LOB であることに注意してください。このデータ・タイプは一般的な LOB ホスト変数と同じであり、すべての LOB は 1 回でメモリーに保管されます。これは (数 MB までの) 小さい LOB の場合に作動しますが、このデータ・タイプを大きい LOB (1 GB) に使用することはできません。 SQLVAR 内のアプリケーションの列定義を変更して、SQL_TYP_*x*LOB_LOCATOR または SQL_TYPE_*x*LOB_FILE のいずれかにすることが必要になります。 (SQLVAR の SQLTYPE フィールドを変更する場合には、SQLLEN フィールドも変更する必要があるので注意してください。) SQLVAR 内の列定義を変更すると、アプリケーションではその新しいタイプに対して正しい容量のストレージを割り振ることができます。 LOB の詳細については、289ページの『第10章 オブジェクト関連機能の使用』を参照してください。
2. その列の値にストレージを割り振る。
3. 割り振ったストレージのアドレスを SQLDA 構造の SQLDATA フィールドに保管する。

これらのステップは、各列の記述を分析し、それぞれの SQLDATA フィールドの内容をその列の値を保持するだけの大きさをもつストレージと置き換えることによって行われます。長さ属性は、LOB タイプでないデータ項目に対する各 SQLVAR 項目の SQLLEN フィールドから判別されます。タイプが BLOB、CLOB、または DBCLOB の項目の場合、長さ属性は 2 番目の SQLVAR 項目の SQLLONGLEN フィールドから判別されます。

さらに、指定した列にヌルを使用できる場合、アプリケーションは SQLIND フィールドの内容を列の標識変数のアドレスと置き換えなければなりません。

カーソルの処理

SQLDA 構造の割り振りが適切に行われると、 SELECT ステートメントに関連するカーソルをオープンし、 FETCH ステートメントの USING DESCRIPTOR 文節を指定することによって行を取り出すことができます。

これが終了したら、カーソルをクローズし、動的に割り振ったメモリーを解放してください。

SQLDA 構造の割り振り

C 言語で SQLDA 構造を作成するには、ホスト言語で INCLUDE SQLDA ステートメントを組み込むか、または SQLDA インクルード・ファイルを組み込んで、構造定義を入手してください。次に、SQLDA のサイズは固定されていないため、アプリケーションは SQLDA へのポインターを宣言し、それにストレージを割り振らなければなりません。 SQLDA 構造の実際のサイズは、SQLDA を用いて渡される個別データ項目の数によって決まります。 (SQLDA を処理するアプリケーションのコーディング方法の例については、 161ページの『例: ADHOC プログラム』を参照してください。)

C/C++ プログラム言語では、SQLDA の割り振りを簡単に行うためにマクロが提供されています。このマクロの形式は以下のとおりです (例外として、HP-UX プラットフォームの場合は形式が異なります)。

```
#define SQLDASIZE(n) (offsetof(struct sqlda, sqlvar) + (n) × sizeof(struct sqlvar))
```

HP-UX プラットフォームの場合、このマクロの形式は以下のとおりです。

```
#define SQLDASIZE(n) (sizeof(struct sqlda) + (n-1) × sizeof(struct sqlvar))
```

このマクロを使用することによって、 n 個の SQLVAR エレメントに必要なストレージを計算することができます。

COBOL で SQLDA 構造を作成するには、 INCLUDE SQLDA ステートメントを組み込むか、または COPY ステートメントを使用します。最も多くの SQLVAR を制御し、その結果 SQLDA が使用するストレージの容量を制御したい場合は、 COPY ステートメントを使用してください。たとえば、SQLVAR のデフォルトの数を 1489 から 1 に変更するには、以下の COPY ステートメントを使用します。

```
COPY "sqlda.cbl"
  replacing --1489--
  by --1--.
```

FORTRAN 言語では、自己定義データ構造または動的割り振りは直接にはサポートされていません。 SQLDA インクルード・ファイルは FORTRAN では使用できません。これは、FORTRAN では SQLDA をデータ構造としてサポートできないためです。 FORTRAN プログラムでは、プリコンパイラーは INCLUDE SQLDA ステートメントを無視します。

ただし、FORTRAN プログラムで静的 SQLDA 構造に似た構造を作成し、これを SQLDA を使用できる任意の場所で使用することができます。 `sqldact.fsqldact.f` ファイルには、 FORTRAN で SQLDA 構造を宣言するのに役立つ定数が含まれています。

ポインター値を必要とする SQLDA エレメントに値を割り当てるには、SQLGADDR の呼び出しを実行してください。

次の表は、SQLVAR エレメントを 1 つ持つ SQLDA 構造の宣言および使用方法を示しています。

言語	ソース・コード例

C/C++	`#include <sqlda.h>`
	`struct sqlda *outda = (struct sqlda *)malloc(SQLDASIZE(1));`
	
	`/* DECLARE LOCAL VARIABLES FOR HOLDING ACTUAL DATA */`
	`double sal;`
	`short salind;`
	
	`/* INITIALIZE ONE ELEMENT OF SQLDA */`
	`memcpy( outda->sqldaid,"SQLDA   ",sizeof(outda->sqldaid));`
	`outda->sqln = outda->sqld = 1;`
	`outda->sqlvar[0].sqltype = SQL_TYP_NFLOAT;`
	`outda->sqlvar[0].sqllen  = sizeof( double );.`
	`outda->sqlvar[0].sqldata = (unsigned char *)&sal;`
	`outda->sqlvar[0].sqlind  = (short *)&salind;`

COBOL	`WORKING-STORAGE SECTION.`
	`77 SALARY          PIC S99999V99 COMP-3.`
	`77 SAL-IND         PIC S9(4)     COMP-5.`
	
	`EXEC SQL INCLUDE SQLDA END-EXEC`
	
	`* Or code a useful way to save unused SQLVAR entries.`
	`* COPY "sqlda.cbl" REPLACING --1489-- BY --1--.`
	
	`01 decimal-sqllen pic s9(4) comp-5.`
	`01 decimal-parts redefines decimal-sqllen.`
	`05 precision pic x.`
	`05 scale pic x.`
	
	`* Initialize one element of output SQLDA`
	`MOVE 1 TO SQLN`
	`MOVE 1 TO SQLD`
	`MOVE SQL-TYP-NDECIMAL TO SQLTYPE(1)`
	
	`* Length = 7 digits precision and 2 digits scale`
	
	`SET SQLDATA(1) TO ADDRESS OF SALARY`
	`SET SQLIND(1)  TO ADDRESS OF SAL-IND`

言語	ソース・コード例

FORTRAN

```
      include 'sqldact.f'

      integer*2  sqlvar1
      parameter ( sqlvar1 = sqlda_header_sz + 0*sqlvar_struct_sz )

C     Declare an Output SQLDA -- 1 Variable
      character    out_sqlda(sqlda_header_sz + 1*sqlvar_struct_sz)

      character*8  out_sqldaid     ! Header
      integer*4    out_sqldabc
      integer*2    out_sqln
      integer*2    out_sqld

      integer*2    out_sqltype1    ! First Variable
      integer*2    out_sqllen1
      integer*4    out_sqldata1
      integer*4    out_sqlind1
      integer*2    out_sqlnamel1
      character*30 out_sqlnamec1

      equivalence( out_sqlda(sqlda_sqldaid_ofs), out_sqldaid )
      equivalence( out_sqlda(sqlda_sqldabc_ofs), out_sqldabc )
      equivalence( out_sqlda(sqlda_sqln_ofs), out_sqln       )
      equivalence( out_sqlda(sqlda_sqld_ofs), out_sqld       )
      equivalence( out_sqlda(sqlvar1+sqlvar_type_ofs), out_sqltype1 )
      equivalence( out_sqlda(sqlvar1+sqlvar_len_ofs), out_sqllen1   )
      equivalence( out_sqlda(sqlvar1+sqlvar_data_ofs), out_sqldata1 )
      equivalence( out_sqlda(sqlvar1+sqlvar_ind_ofs), out_sqlind1   )
      equivalence( out_sqlda(sqlvar1+sqlvar_name_length_ofs),
     +             out_sqlnamel1                                   )
      equivalence( out_sqlda(sqlvar1+sqlvar_name_data_ofs),
     +             out_sqlnamec1                                   )

C     Declare Local Variables for Holding Returned Data.
      real*8       salary
      integer*2    sal_ind

C     Initialize the Output SQLDA (Header)
      out_sqldaid  = 'OUT_SQLDA'
      out_sqldabc  = sqlda_header_sz + 1*sqlvar_struct_sz
      out_sqln     = 1
      out_sqld     = 1
C     Initialize VAR1
      out_sqltype1 = SQL_TYP_NFLOAT
      out_sqllen1  = 8
      rc = sqlgaddr( %ref(salary), %ref(out_sqldata1) )
      rc = sqlgaddr( %ref(sal_ind), %ref(out_sqlind1) )
```

動的なメモリー割り振りをサポートしない言語では、 SQLVAR エレメントの希望数を指定した SQLDA をホスト言語で明示的に宣言しなければなりません。 SQLVAR のエレメントには、アプリケーションの必要に応じて決定されたとおりの十分な数を必ず宣言してください。

SQLDA 構造を使用するデータの受け渡し

ホスト変数のリストを使用してデータを受け渡すよりも、 SQLDA を使用してデータを受け渡すほうが、より高い柔軟性が得られます。たとえば、SQLDA を用いて、固有のホスト言語に対応するものをもたないデータ (C 言語の DECIMAL データなど) を転送することができます。 ADHOC と呼ばれるサンプル・プログラムは、この技法を用いた例です。 (161ページの『例: ADHOC プログラム』を参照。) 数値と記号名がどのように関連付けられているかを示す便利な相互参照リストについては、 表8 を参照してください。

表 8. DB2 V2 SQLDA SQL タイプ. 数値および対応する記号名

SQL 列名	SQLTYPE 数値	SQLTYPE 記号名[1]
DATE	384/385	SQL_TYP_DATE / SQL_TYP_NDATE
TIME	388/389	SQL_TYP_TIME / SQL_TYP_NTIME
TIMESTAMP	392/393	SQL_TYP_STAMP / SQL_TYP_NSTAMP
n/a[2]	400/401	SQL_TYP_CGSTR / SQL_TYP_NCGSTR
BLOB	404/405	SQL_TYP_BLOB / SQL_TYP_NBLOB
CLOB	408/409	SQL_TYP_CLOB / SQL_TYP_NCLOB
DBCLOB	412/413	SQL_TYP_DBCLOB / SQL_TYP_NDBCLOB
VARCHAR	448/449	SQL_TYP_VARCHAR / SQL_TYP_NVARCHAR
CHAR	452/453	SQL_TYP_CHAR / SQL_TYP_NCHAR
LONG VARCHAR	456/457	SQL_TYP_LONG / SQL_TYP_NLONG
n/a[3]	460/461	SQL_TYP_CSTR / SQL_TYP_NCSTR
VARGRAPHIC	464/465	SQL_TYP_VARGRAPH / SQL_TYP_NVARGRAPH
GRAPHIC	468/469	SQL_TYP_GRAPHIC / SQL_TYP_NGRAPHIC
LONG VARGRAPHIC	472/473	SQL_TYP_LONGRAPH / SQL_TYP_NLONGRAPH
FLOAT	480/481	SQL_TYP_FLOAT / SQL_TYP_NFLOAT
REAL[4]	480/481	SQL_TYP_FLOAT / SQL_TYP_NFLOAT
DECIMAL[5]	484/485	SQL_TYP_DECIMAL / SQL_TYP_DECIMAL
INTEGER	496/497	SQL_TYP_INTEGER / SQL_TYP_NINTEGER
SMALLINT	500/501	SQL_TYP_SMALL / SQL_TYP_NSMALL
n/a	804/805	SQL_TYP_BLOB_FILE / SQL_TYPE_NBLOB_FILE
n/a	808/809	SQL_TYP_CLOB_FILE / SQL_TYPE_NCLOB_FILE

表 8. DB2 V2 SQLDA SQL タイプ (続き). 数値および対応する記号名

SQL 列名	SQLTYPE 数値	SQLTYPE 記号名[1]
n/a	812/813	SQL_TYP_DBCLOB_FILE / SQL_TYPE_NDBCLOB_FILE
n/a	960/961	SQL_TYP_BLOB_LOCATOR / SQL_TYP_NBLOB_LOCATOR
n/a	964/965	SQL_TYP_CLOB_LOCATOR / SQL_TYP_NCLOB_LOCATOR
n/a	968/969	SQL_TYP_DBCLOB_LOCATOR / SQL_TYP_NDBCLOB_LOCATOR

注: これらの定義タイプは sql.h インクルード・ファイルにあり、インクルード・ファイル自体は、sqllib ディレクトリーの include サブディレクトリーにあります。 (たとえば、C プログラミング言語の場合は sqllib/include/sql.h となります。)

1. COBOL プログラミング言語の場合、SQLTYPE には下線 (_) を使用しませんが、その代わりにハイフン (-) を使用します。
2. これは NULL 終了グラフィック・ストリングです。
3. これは NULL 終了文字ストリングです。
4. SQLDA での REAL と DOUBLE の違いは長さの値です (4 または 8)。
5. 精度は最初のバイトにあります。位取りは 2 番目のバイトにあります。

対話式 SQL ステートメントの処理

動的 SQL を使用するアプリケーションを作成し、任意の SQL ステートメントを処理することができます。たとえば、アプリケーションがユーザーから SQL ステートメントを受け入れる場合、アプリケーションはステートメントについて事前にわかっていなくても、そのステートメントを実行できなければなりません。

PREPARE および DESCRIBE ステートメントを SQLDA 構造で使用することにより、アプリケーションは実行される SQL ステートメントのタイプを判別し、それに応じて処理することができます。

対話式 SQL ステートメントを処理するプログラムの例については、 161ページの『例: ADHOC プログラム』を参照してください。

ステートメントのタイプの判別

SQL ステートメントを準備する場合、ステートメントのタイプに関する情報は SQLDA 構造を調べて判別することができます。この情報はステートメントの準備時に INTO 文節を指定して SQLDA 構造に入れるか、または事前に準備されたステートメントに対して DESCRIBE ステートメントを発行することによって、 SQLDA 構造に入れることができます。

いずれの場合でも、データベース・マネージャーは SQLDA 構造の SQLD フィールドに 1 つの値を入れ、 SQL ステートメントにより生成された結果表に列の数を示します。 SQLD フィールドにゼロ (0) が入っている場合、このステートメントは SELECT ステートメントではありません。ステートメントはすでに準備されているため、EXECUTE ステートメントを使用してただちに実行することができます。

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、 USING 文節は *SQL 解説書* に記述されている方法で指定する必要があります。 USING 文節は、ホスト変数のリストか SQLDA 構造のどちらかを指定することができます。

SQLD フィールドが 0 より大きい場合、ステートメントは SELECT ステートメントであるため、次の節での説明に従って処理しなければなりません。

可変リスト SELECT ステートメント

可変リスト SELECT ステートメントとは、戻される列の数およびタイプがプリコンパイル時にはわからないステートメントのことです。この場合、アプリケーションには、結果表の行を保持するために宣言しなければならない正確なホスト変数がわかりません。

可変リスト SELECT ステートメントを処理するには、アプリケーションでは以下のようにすることができます。

1. **SQLDA を宣言する**。可変リスト SELECT ステートメントを処理するには、SQLDA 構造を必ず使用します。
2. **INTO 文節を使用してステートメントを PREPARE (準備) する**。アプリケーションは、宣言した SQLDA 構造に十分な SQLVAR エレメントがあるかどうかを判別します。十分なエレメントがない場合、アプリケーションは必要な数の SQLVAR エレメントを持つ別の SQLDA 構造を割り振り、新規の SQLDA を用いて追加の DESCRIBE ステートメントを発行します。
3. **SQLVAR エレメントを割り振る**。各 SQLVAR に必要なホスト変数および標識に、ストレージを割り振ります。このステップでは、それぞれの SQLVAR エレメントにデータの割り振りアドレスおよび標識変数を入れます。
4. **SELECT ステートメントを処理する**。カーソルは準備済みステートメントに関連付けられ、オープンされます。行は適切に割り振られた SQLDA 構造を用いて取り出されます。

これらのステップについては、以下の節で詳しく説明されています。

- 150ページの『SQLDA の宣言』
- 151ページの『最小の SQLDA 構造を用いたステートメントの準備』
- 152ページの『十分な数の SQLVAR 項目を指定した SQLDA の割り振り』
- 153ページの『SELECT ステートメントの記述』
- 154ページの『行を保持するためのストレージの獲得』
- 154ページの『カーソルの処理』

エンド・ユーザーからの SQL 要求の保管

アプリケーションで任意の SQL ステートメントを保管できる場合、これらをデータ・タイプが VARCHAR、LONG VARCHAR、CLOB、VARGRAPHIC、 LONG VARGRAPHIC、または DBCLOB の列を持つ表に保管することができます。その

VARGRAPHIC、LONG VARGRAPHIC、および DBCLOB データ・タイプは、 2 バイト文字サポート (DBCS) および拡張 UNIX コード (EUC) 環境でしか使用できないので注意してください。

ユーザーは、準備済みのバージョンの SQL ステートメントではなく、そのソースを保管しなければなりません。これは、表に保管されているバージョンを実行する前に、各ステートメントを検索して準備しなければならないことを意味します。つまり、アプリケーションは、文字ストリングから SQL ステートメントを準備し、このステートメントを動的に実行します。

例: ADHOC プログラム

この例では、対話式 SQL ステートメントを処理するための SQLDA の使用法を示します。

注: この例 adhoc.sqc は、C 言語のみを対象としています。

ADHOC プログラムの動作の仕組み

1. **SQLDA 構造を定義する。** INCLUDE SQLDA ステートメントは、データベース・マネージャーとプログラムとの間でデータを受け渡しする際に用いられる SQLDA 構造を定義および宣言します。
2. **SQLCA 構造を定義する。** INCLUDE SQLCA ステートメントは SQLCA 構造を定義し、その構造内のエレメントとして `SQLCODE` を定義します。 SQLCA 構造の SQLCODE フィールドは、SQL ステートメントの実行後に、データベース・マネージャーにより診断情報を用いて更新されます。
3. **ホスト変数を宣言する。** BEGIN DECLARE SECTION および END DECLARE SECTION ステートメントは、ホスト変数宣言を区切ります。ホスト変数には、SQL ステートメントで参照される際に、接頭部としてコロン (:) が付けられます。
4. **データベースに接続する。**プログラムはユーザー指定のデータベースに接続し、そのデータベースへの共用アクセスを要求します。 (START DATABASE MANAGER API 呼び出しまたは `db2start` コマンドが実行されていることを前提としています。) 共用アクセス・モードで同じデータベースに接続しようとする他のプログラムにもアクセスが許可されます。
5. **完了をチェックする。** SQLCA 構造は、CONNECT TO ステートメントが正常に終了したかどうかをチェックします。 `SQLCODE` 値が `0` のときは、接続が成功したことを示します。
6. **対話式プロンプト。** SQL ステートメントはプロンプトによって入力され、その後の処理のために `process_statement` 関数に送られます。
7. **トランザクションの終了 - COMMIT。**作業単位が終了したことをユーザーが確認したならば、COMMIT で終了します。最後の COMMIT 以後、入力された SQL ステートメントによって要求されたすべての変更がデータベースに保管されます。

8. **トランザクションの終了 - ROLLBACK**。作業単位が終了したことをユーザーが確認したならば、ROLLBACK で終了します。最後の COMMIT またはプログラムの開始以後、入力された SQL ステートメントによって要求されたすべての変更は取り消されます。
9. **データベースから切断する**。プログラムは CONNECT RESET ステートメントを実行して、データベースから切断します。実行時には、SQLCA が正常に終了したかどうかがチェックされます。
10. **SQL ステートメント・テキストをホスト変数にコピーする**。ステートメント・テキストは、ホスト変数 st で指定したデータ域にコピーされます。
11. **処理のために SQLDA を準備する**。初期 SQLDA 構造が宣言され、メモリーが init_da プロシージャーによって割り振られ、 SQL ステートメントによって生成される出力タイプが決められます。 PREPARE ステートメントから戻された SQLDA は、 SQL ステートメントから戻される列の数を報告します。
12. **SQLDA が存在している出力列を報告する**。 SQL ステートメントは SELECT ステートメントです。 SQLDA は init_da プロシージャーによって初期化され、準備済み SQL ステートメントが置かれるメモリー・スペースを割り振ります。
13. **SQLDA は出力列がないことを報告する**。戻される列がありません。 SQL ステートメントは EXECUTE ステートメントを使用して*動的に* 実行されます。
14. **SQLDA 用のメモリー・スペースを準備する**。メモリーは、SQLDA 内の列構造を反映して割り振られます。メモリーの所要量は、SQLDA 内の列構造の SQLTYPE と SQLLEN によって選択されます。
15. **カーソルを宣言およびオープンする**。 DECLARE ステートメントはカーソル pcurs を sqlStatement 内にあり動的に準備された SQL ステートメントに関連付け、その後カーソルはオープンされます。
16. **行を取り出す**。 FETCH ステートメントは、カーソルを次の行に位置付け、その行の内容を SQLDA に移動します。
17. **列タイトルを表示する**。取り出された最初の行が、列タイトル情報です。
18. **行情報を表示する**。連続して実行される FETCH によって収集された行情報が表示されます。
19. **カーソルをクローズする**。 CLOSE ステートメントはカーソルをクローズし、カーソルに関連付けられていたリソースを解放します。

EMB_SQL_CHECK マクロ / 関数は、プログラム外部にあるエラー検査ユーティリティーです。 DB2 API を呼び出す C プログラムの場合、 utilapi.c 内の sqlInfoPrint 関数は、 utilapi.h 内の API_SQL_CHECK として再定義されます。 C 組み込み SQL プログラムの場合、 utilemb.sqc 内の sqlInfoPrint 関数は、 utilemb.h 内の EMB_SQL_CHECK として再定義されます。このエラー検査ユーティリティーのソース・コードについては、125ページの『プログラム例での GET ERROR MESSAGE の使用』を参照してください。

この例では、 utilemb.sqc ファイルにユーティリティーとして提供されているたくさんの追加のプロシージャーを使用することに注意してください。そのプロシージャーには、以下のものがあります。

init_da 準備済み SQL ステートメントにメモリーを割り振ります。内部記述関数 SQLDASIZE を使用して、適正な記憶容量が計算されます。

alloc_host_vars
SQLDA ポインターからのデータ用にメモリーを割り振ります。

free_da
SQLDA データ構造を使用するために割り振られていたメモリーを解放します。

print_var
SQLDA SQLVAR 変数を印刷します。このプロシージャーは、まずデータ・タイプを判別してから、データの印刷に必要なサブルーチンを呼び出します。

display_da
渡されているポインターの出力を表示します。出力データの構造に関係のあるすべての情報は、プロシージャー print_var で検査されたものとして、このポインターから利用できます。

C の例: ADHOC.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlcodes.h>
#include <sqlda.h>  1
#include "utilemb.h"

#ifdef DB268K
  /* Need to include ASLM for 68K applications */
  #include <LibraryManager.h>
#endif

EXEC SQL INCLUDE SQLCA ;  2

#define SQLSTATE sqlca.sqlstate

int process_statement( char * ) ;

int main( int argc, char *argv[] ) {

   int rc ;
   char sqlInput[256] ;
   char st[1024] ;

   EXEC SQL BEGIN DECLARE SECTION ;  3
      char userid[9] ;
      char passwd[19] ;
   EXEC SQL END DECLARE SECTION ;

#ifdef DB268K
  /*
   Before making any API calls for 68K environment,
   need to initial the Library Manager
   */
   InitLibraryManager(0,kCurrentZone,kNormalMemory) ;
   atexit(CleanupLibraryManager) ;
#endif

   printf( "Sample C program : ADHOC interactive SQL¥n" ) ;

   /* Initialize the connection to a database. */
   if ( argc == 1 ) {
      EXEC SQL CONNECT TO sample ;
      EMB_SQL_CHECK( "CONNECT TO SAMPLE" ) ;
   }
   else if ( argc == 3 ) {
      strcpy( userid, argv[1] ) ;
      strcpy( passwd, argv[2] ) ;
      EXEC SQL CONNECT TO sample USER :userid USING :passwd ;  4
      EMB_SQL_CHECK( "CONNECT TO SAMPLE" ) ;  5
   }
   else {
```

```
      printf( "¥nUSAGE: adhoc [userid passwd]¥n¥n" ) ;
      return( 1 ) ;
   } /* endif */

   printf( "Connected to database SAMPLE¥n" ) ;

   /* Enter the continuous command line loop. */
   *sqlInput = '¥0' ;
   while ( ( *sqlInput != 'q' ) && ( *sqlInput != 'Q' ) ) {  6

      printf( "Enter an SQL statement or 'quit' to Quit :¥n" ) ;
      gets( sqlInput ) ;

      if ( ( *sqlInput == 'q' ) || ( *sqlInput == 'Q' ) ) break ;

      if ( *sqlInput == '¥0' ) { /* Don't process the statement */
         printf( "No characters entered.¥n" ) ;
         continue ;
      }

      strcpy( st, sqlInput ) ;
      while ( sqlInput[strlen( sqlInput ) - 1] == '¥¥' ) {
         st[strlen( st ) - 1] = '¥0' ;
         gets( sqlInput ) ;
         strcat( st, sqlInput ) ;
      }

      /* Process the statement. */
      rc = process_statement( st ) ;

   }

   printf( "Enter 'c' to COMMIT or Any Other key to ROLLBACK the transaction :¥n" ) ;
   gets( sqlInput ) ;
   if ( ( *sqlInput == 'c' ) || ( *sqlInput == 'C' ) ) {
      printf( "COMMITING the transactions.¥n" ) ;
      EXEC SQL COMMIT ;   7
      EMB_SQL_CHECK( "COMMIT" ) ;
   }
   else { /* assume that the transaction is to be rolled back */
      printf( "ROLLING BACK the transactions.¥n" ) ;
      EXEC SQL ROLLBACK ;  8
      EMB_SQL_CHECK( "ROLLBACK" ) ;
   }

   EXEC SQL CONNECT RESET ;  9
   EMB_SQL_CHECK( "CONNECT RESET" ) ;

   return( 0 ) ;

}

/******************************************************************************
 * FUNCTION : process_statement
 * This function processes the inputted statement and then prepares the
```

```
 *  procedural SQL implementation to take place.
 ******************************************************************************/
int process_statement ( char * sqlInput ) {

   int counter = 0 ;
   struct sqlda * sqldaPointer ;
   short sqlda_d ;

   EXEC SQL BEGIN DECLARE SECTION ; 3
      char st[1024] ;
   EXEC SQL END DECLARE SECTION ;

   strcpy( st, sqlInput ) ; 10
   /* allocate an initial SQLDA temp pointer to obtain information
      about the inputted "st" */

   init_da( &sqldaPointer, 1 ) ; 11

   EXEC SQL PREPARE statement1 from :st ;
   /* EMB_SQL_CHECK( "PREPARE" ) ; */

   EXEC SQL DESCRIBE statement1 INTO :*sqldaPointer ;

   /* Expecting a return code of 0 or SQL_RC_W236,
      SQL_RC_W237, SQL_RC_W238, SQL_RC_W239 for cases
      where this statement is a SELECT statment. */
   if ( SQLCODE != 0           &&
        SQLCODE != SQL_RC_W236 &&
        SQLCODE != SQL_RC_W237 &&
        SQLCODE != SQL_RC_W238 &&
        SQLCODE != SQL_RC_W239
      ) {
      /* An unexpected warning/error has occurred. Check the SQLCA. */
      EMB_SQL_CHECK( "DESCRIBE" ) ;
   } /* end if */

   sqlda_d = sqldaPointer->sqld ;
   free( sqldaPointer ) ;

   if ( sqlda_d > 0 ) { 12

      /* this is a SELECT statement, a number of columns
         are present in the SQLDA */

      if ( SQLCODE == SQL_RC_W236 || SQLCODE == 0)
         /* this out only needs a SINGLE SQLDA */
         init_da( &sqldaPointer, sqlda_d ) ;

      if ( SQLCODE == SQL_RC_W237 ||
           SQLCODE == SQL_RC_W238 ||
           SQLCODE == SQL_RC_W239 )
         /* this output contains columns that need a DOUBLED SQLDA */
         init_da( &sqldaPointer, sqlda_d * 2 ) ;

      /* need to reassign the SQLDA with the correct number
```

```
      of columns to the SQL statement */
   EXEC SQL DESCRIBE statement1 INTO :*sqldaPointer ;
   EMB_SQL_CHECK( "DESCRIBE" ) ;

   /* allocating the proper amount of memory
      space needed for the variables */
   alloc_host_vars( sqldaPointer ) ;  14

   /* Don't need to check the SQLCODE for declaration of cursors */
   EXEC SQL DECLARE pcurs CURSOR FOR statement1 ;  15

   EXEC SQL OPEN pcurs ;  15
   EMB_SQL_CHECK( "OPEN" ) ;

   EXEC SQL FETCH pcurs USING DESCRIPTOR :*sqldaPointer;  16
   EMB_SQL_CHECK( "FETCH" ) ;

   /* if the FETCH is successful, obtain data from SQLDA */
   /* display the column titles */
   display_col_titles( sqldaPointer ) ;  17

   /* display the rows that are fetched */
   while ( SQLCODE == 0 ) {
      counter++ ;
      display_da( sqldaPointer ) ;  18
      EXEC SQL FETCH pcurs USING DESCRIPTOR :*sqldaPointer ;
   }  /* endwhile */

   EXEC SQL CLOSE pcurs ;  19
   EMB_SQL_CHECK( "CLOSE CURSOR" ) ;
   printf( "¥n %d record(s) selected¥n¥n", counter ) ;

   /* Free the memory allocated to this SQLDA. */
   free_da( sqldaPointer ) ;

} else { /* this is not a SELECT statement, execute SQL statement */  13
   EXEC SQL EXECUTE statement1 ;
   EMB_SQL_CHECK( "Executing the SQL statement" ) ;
}  /* end if */

return( 0 ) ;

}     /* end of program : ADHOC.SQC */
```

動的 SQL への変数入力

この節では、動的 SQL アプリケーションでパラメーター・マーカーを使用してホスト変数情報を表現する方法について説明します。以下のトピックを扱います。
- パラメーター・マーカーの使用
- 例: VARINP プログラム

パラメーター・マーカーの使用

動的 SQL ステートメントにはホスト変数を入れることができません。それは、ホスト変数情報 (データ・タイプおよび長さ) がアプリケーションのプリコンパイルの間しか使用できないためです。実行時には、ホスト変数情報はありません。そのため、アプリケーション変数を表すには新しい方法が必要です。ホスト変数は疑問符 (?) で表されます。これは、*パラメーター・マーカー* と呼ばれます。パラメーター・マーカーは、ホスト変数が SQL ステートメントの内部で置換される位置を示します。パラメーター・マーカーは、 SQL ステートメント内部で使用するコンテキストによって想定されたデータ・タイプおよび長さを持っています。

パラメーター・マーカーのデータ・タイプが、これを使用しているステートメントの内容からはっきり判別できない場合は、 CAST を使用してタイプを指定することができます。このようなパラメーター・マーカーは、*タイプ付きパラメーター・マーカー* と見なされます。タイプ付きパラメーター・マーカーは、指定されたタイプのホスト変数と同様に扱われます。たとえば、ステートメント `SELECT ? FROM SYSCAT.TABLES` は、結果列のタイプが DB2 には認識されないため無効です。ただし、`SELECT CAST(? AS INTEGER) FROM SYSCAT.TABLES` は、パラメーター・マーカーが INTEGER を表すことがキャストによって*約束* されているため、 DB2 には結果列のタイプが認識されます。

パラメーター・マーカーが入っている文字ストリングは、次のような形となります。

```
DELETE FROM TEMPL WHERE EMPNO = ?
```

このステートメントが実行されると、 EXECUTE ステートメントの USING 文節によってホスト変数つまり SQLDA 構造が指定されます。ステートメントを実行する際に、ホスト変数の内容が使用されます。

SQL ステートメントにパラメーター・マーカーが 1 つ以上あると、 EXECUTE ステートメントの USING 文節はホスト変数 (各パラメーター・マーカーに 1 つずつ) のリストを指定するか、または各パラメーター・マーカーの SQLVAR 項目を持つ SQLDA を識別しなければなりません。 (LOB の場合は、各パラメーター・マーカーに SQLVAR が 2 つずつあることに注意してください。) ホスト変数リストまたは SQLVAR 項目は、ステートメント内のパラメーター・マーカーの順序に従って突き合わせが行われます。また、これらのデータ・タイプには互換性がなければなりません。

動的 SQL でのパラメーター・マーカーの使用は、静的 SQL でのホスト変数の使用に似ていることに注意してください。いずれの場合も、最適化プログラムは配布統計を使用せず、最適のアクセス・プランを選択することはありえません。

パラメーター・マーカーに適用される規則は、 *SQL 解説書* の PREPARE ステートメントの箇所にリストされています。

例: VARINP プログラム

これは、パラメーター・マーカーを探索および更新の条件内で使用している UPDATE の例です。このサンプルは、以下のプログラム言語で入手可能です。

C varinp.sqc

Java Varinp.java

COBOL varinp.sqb

VARINP プログラムの動作の仕組み

1. **SELECT SQL ステートメントを準備する。** PREPARE ステートメントを呼び出すと、SQL ステートメントが動的に準備されます。この SQL ステートメント内では、パラメーター・マーカーは `?` で明示されます。 `staff` の `job` フィールドは、結果表に指定されていなくても更新可能として定義されます。
2. **カーソルを宣言する。** DECLARE CURSOR ステートメントはカーソル `c1` を、 **1** で準備された照会に関連付けます。
3. **カーソルをオープンする。**カーソル `c1` がオープンすると、データベース・マネージャーは照会を実行し、結果表を作成します。カーソルは第 1 行目より*前に* 置かれます。
4. **UPDATE SQL ステートメントを準備する。** PREPARE ステートメントを呼び出すと、 SQL ステートメントが動的に準備されます。このステートメントのパラメーター・マーカーは `Clerk` に設定されますが、更新時に指定された列データ・タイプに適合している限り、設定を任意に動的に変更することも可能です。
5. **行を取り出す。** FETCH ステートメントはカーソルを次の行に置き、その行の内容をホスト変数に移動します。この行が*現在* 行になります。
6. **現在行を更新する。**現在行および指定された列 `job` は、渡されたパラメーター `parm_var` の内容を用いて更新されます。
7. **カーソルをクローズする。** CLOSE ステートメントを発行すると、カーソルに関連したリソースが解放されます。ただし、カーソルは再度オープンすることができます。

`CHECKERR` マクロ / 関数は、プログラム外部にあるエラー検査ユーティリティーです。エラー検査ユーティリティーの所在は、ご使用のプログラム言語により異なります。

C DB2 API を呼び出す C プログラムの場合、 utilapi.c 内の `sqlInfoPrint` 関数は、 utilapi.h 内の `API_SQL_CHECK` として再定義さ

れます。 C 組み込み SQL プログラムの場合、 utilemb.sqc 内の sqlInfoPrint 関数は、 utilemb.h 内の EMB_SQL_CHECK として再定義されます。

Java SQL エラーは SQLException としてスローされ、アプリケーションの catch ブロックで処理されます。

COBOL CHECKERR は checkerr.cbl という名前の外部プログラムです。

このエラー検査ユーティリティーのソース・コードについては、 125ページの『プログラム例での GET ERROR MESSAGE の使用』を参照してください。

C の例: VARINP.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[])
{

   EXEC SQL BEGIN DECLARE SECTION;
      char   pname[10];
      short  dept;
      char userid[9];
      char passwd[19];
      char st[255];
      char parm_var[6];
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: VARINP ¥n" );

   if (argc == 1)
   {
      EXEC SQL CONNECT TO sample;
  EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3)
   {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else
   {
      printf ("¥nUSAGE: varinp [userid passwd]¥n¥n");
      return 1;
   } /* endif */

   strcpy (st, "SELECT name, dept FROM staff ");
   strcat (st, "WHERE job = ? FOR UPDATE OF job");
   EXEC SQL PREPARE s1 FROM :st;  1
   EMB_SQL_CHECK("PREPARE");

   EXEC SQL DECLARE c1 CURSOR FOR s1;  2

   strcpy (parm_var, "Mgr");
   EXEC SQL OPEN c1 USING :parm_var;  3
   EMB_SQL_CHECK("OPEN");

   strcpy (parm_var, "Clerk");
   strcpy (st, "UPDATE staff SET job = ? WHERE CURRENT OF c1");
   EXEC SQL PREPARE s2 from :st;  4
```

```
   do
   {
      EXEC SQL FETCH c1 INTO :pname, :dept;  5
      if (SQLCODE != 0) break;

      printf( "%-10.10s in dept. %2d will be demoted to Clerk¥n",
              pname, dept );
      EXEC SQL EXECUTE s2 USING :parm_var;  6
      EMB_SQL_CHECK("EXECUTE");
   } while ( 1 );

   EXEC SQL CLOSE c1;   7
   EMB_SQL_CHECK("CLOSE CURSOR");

   EXEC SQL ROLLBACK;
   EMB_SQL_CHECK("ROLLBACK");
   printf( "¥nOn second thought -- changes rolled back.¥n" );

   EXEC SQL CONNECT RESET;
   EMB_SQL_CHECK("CONNECT RESET");
   return 0;
}
/* end of program : VARINP.SQC */
```

Java の例: Varinp.java

```
import java.sql.*;

class Varinp
{   static
  {   try
    {   Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
    }
    catch (Exception e)
    {   System.out.println ("¥n  Error loading DB2 Driver...¥n");
      System.out.println (e);
      System.exit(1);
    }
  }

  public static void main(String argv[])
  {   try
    {   System.out.println ("  Java Varinp Sample");
      // Connect to Sample database

      Connection con = null;
      // URL is jdbc:db2:dbname
     String url = "jdbc:db2:sample";

      if (argv.length == 0)
      {   // connect with default id/password
        con = DriverManager.getConnection(url);
      }
      else if (argv.length == 2)
      {   String userid = argv[0];
        String passwd = argv[1];

        // connect with user-provided username and password
        con = DriverManager.getConnection(url, userid, passwd);
      }
      else
      {   throw new Exception("¥nUsage: java Varinp [username password]¥n");
      }

      // Enable transactions
      con.setAutoCommit(false);

      // Perform dynamic SQL using JDBC
      try
      {   PreparedStatement pstmt1 = con.prepareStatement(
          "SELECT name, dept FROM staff WHERE job = ? FOR UPDATE OF job"); 1
      // set cursor name for the positioned update statement
      pstmt1.setCursorName("c1");                                          2
      pstmt1.setString(1, "Mgr");
      ResultSet rs = pstmt1.executeQuery();                                3

      PreparedStatement pstmt2 = con.prepareStatement(
          "UPDATE staff SET job = ? WHERE CURRENT OF c1");               4
      pstmt2.setString(1, "Clerk");
```

```
        System.out.print("¥n");
        while( rs.next() )                                                    5
        {   String name = rs.getString("name");
          short  dept = rs.getShort("dept");
          System.out.println(name + " in dept. " + dept
            + " will be demoted to Clerk");

          pstmt2.executeUpdate();                                             6
        };

        rs.close();
        pstmt1.close();                                                       7
        pstmt2.close();
        }
        catch( Exception e )
        {   throw e;
        }
        finally
        {   // Rollback the transaction
          System.out.println("¥nRollback the transaction...");
          con.rollback();
          System.out.println("Rollback done.");
        }
      }
      catch( Exception e )
      {   System.out.println(e);
      }
    }
  }
```

COBOL の例: VARINP.SQB

```
 Identification Division.
 Program-ID. "varinp".

 Data Division.
 Working-Storage Section.

     copy "sqlca.cbl".

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 pname               pic x(10).
 01 dept                pic s9(4) comp-5.
 01 st                  pic x(127).
 01 parm-var            pic x(5).
 01 userid              pic x(8).
 01 passwd.
   49 passwd-length     pic s9(4) comp-5 value 0.
   49 passwd-name       pic x(18).
     EXEC SQL END DECLARE SECTION END-EXEC.

 77 errloc            pic x(80).

 Procedure Division.
 Main Section.
     display "Sample COBOL program: VARINP".

* Get database connection information.
     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR format
* with the length of the input string.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".

     EXEC SQL CONNECT TO sample USER :userid USING :passwd
         END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.

     move "SELECT name, dept FROM staff
-         "   WHERE job = ? FOR UPDATE OF job" to st.
     EXEC SQL PREPARE s1 FROM :st END-EXEC.                              1
     move "PREPARE" to errloc.
     call "checkerr" using SQLCA errloc.

     EXEC SQL DECLARE c1 CURSOR FOR s1 END-EXEC.                         2
```

```
    move "Mgr" to parm-var.

    EXEC SQL OPEN c1 USING :parm-var END-EXEC                          3
    move "OPEN" to errloc.
    call "checkerr" using SQLCA errloc.

    move "Clerk" to parm-var.
    move "UPDATE staff SET job = ? WHERE CURRENT OF c1" to st.

    EXEC SQL PREPARE s2 from :st END-EXEC.                             4
    move "PREPARE S2" to errloc.
    call "checkerr" using SQLCA errloc.

* call the FETCH and UPDATE loop.
    perform Fetch-Loop thru End-Fetch-Loop
       until SQLCODE not equal 0.

    EXEC SQL CLOSE c1 END-EXEC.                                        7
    move "CLOSE" to errloc.
    call "checkerr" using SQLCA errloc.

    EXEC SQL ROLLBACK END-EXEC.
    move "ROLLBACK" to errloc.
    call "checkerr" using SQLCA errloc.
    DISPLAY "On second thought -- changes rolled back.".

    EXEC SQL CONNECT RESET END-EXEC.
    move "CONNECT RESET" to errloc.
    call "checkerr" using SQLCA errloc.
End-Main.
    go to End-Prog.

Fetch-Loop Section.
    EXEC SQL FETCH c1 INTO :pname, :dept END-EXEC.                     5
    if SQLCODE not equal 0
       go to End-Fetch-Loop.
    display pname, " in dept. ", dept,
       " will be demoted to Clerk".

    EXEC SQL EXECUTE s2 USING :parm-var END-EXEC.                      6
    move "EXECUTE" to errloc.
    call "checkerr" using SQLCA errloc.

End-Fetch-Loop. exit.

End-Prog.
    stop run.
```

DB2 コール・レベル・インターフェース (CLI)

組み込み SQL インターフェースを使用するアプリケーションには、 SQL ステートメントをコードに変換するプリコンパイラーが必要で、変換後そのコードはコンパイルされ、データベースにバインドされ、実行されます。対照的に、DB2 CLI アプリケーションは、プリコンパイルまたはバインドする必要がなく、代わりに関数の標準セットを使用して実行時に SQL ステートメントおよび関連サービスを実行します。

この違いは重要です。というのはこれまでプリコンパイラーは、個々のデータベース製品に特定のものであったからです。これは効果的にユーザーのアプリケーションをその製品に結び付けるものでした。 DB2 CLI を使用すると、どの特定のデータベース製品からも独立した可搬性のあるアプリケーションを作成することが可能になります。この独立性によって、 DB2 CLI アプリケーションはさまざまな DB2 データベース (DRDA データベースを含む) にアクセスするために再コンパイルまたは再バインドする必要がなく、ただ該当するデータベースに実行時に接続するだけで済むようになります。

組み込み SQL と DB2 CLI との比較

DB2 CLI と組み込み SQL は、次のような点でも異なります。

- DB2 CLI では、カーソルの明示宣言は必要ありません。必要に応じて DB2 CLI で生成されます。そして、アプリケーションはその生成されたカーソルを通常のカーソル取り出しモデルとして、複数行の SELECT ステートメント、および定位置 UPDATE と DELETE ステートメント用に使用します。
- OPEN ステートメントは DB2 CLI では使用しません。その代わりに、SELECT の実行によって自動的にカーソルがオープンされます。
- 組み込み SQL とは違って、DB2 CLI では、 EXECUTE IMMEDIATE ステートメントに相当するステートメント (SQLExecDirect() 関数) でパラメーター・マーカーの使用が可能です。
- DB2 CLI の COMMIT または ROLLBACK は、 SQL ステートメントとして渡されるのではなく、 SQLEndTran() 関数呼び出しによって発行されます。
- DB2 CLI はステートメント関連情報をアプリケーションのために管理し、*ステートメント・ハンドル* を提供してそれを要約オブジェクトとして参照できるようにします。このハンドルによって、アプリケーションが製品特有のデータ構造を使用する必要がなくなります。
- ステートメント・ハンドルと同様に、*環境ハンドル* および*接続ハンドル* は、すべてのグローバル変数および接続特有の情報を参照する方法を提供します。*記述子ハンドル* は、SQL ステートメントのパラメーターか、結果セットの列のどちらかの状況を記述します。
- DB2 CLI は、X/Open SQL CAE 仕様で定義された SQLSTATE 値を使用します。この形式および値のほとんどは、IBM リレーショナル・データベース製品で使用する値と一貫性がありますが、違いもあります。 (ODBC SQLSTATES と X/Open 定義の SQLSTATES の間にも違いがあります。)

- DB2 CLI はスクロール可能カーソルをサポートします。スクロール可能なカーソルを使用すると、静的カーソルによって次のようなスクロールが可能になります。
  - 1 行または複数行ごとに下方へ
  - 1 行または複数行ごとに上方へ
  - 最初の行から 1 行または複数行ごとに
  - 最後の行から 1 行または複数行ごとに

上記の違いは別にして、組み込み SQL と DB2 CLI には次の重要な共通の概念があります。 *DB2 CLI は組み込み SQL で動的に作成できる SQL ステートメントを実行することができます。*

注: さらに、DB2 CLI は複合 SQL ステートメントのような、動的に準備できない一部の SQL ステートメントも受け入れることができます。

759ページの表38 に各 SQL ステートメントをリストし、 DB2 CLI を使用して実行できるかどうかを示してあります。また、表には、コマンド行プロセッサーを使用してステートメントを対話式で実行できるかどうかも示してあります (これは SQL ステートメントをプロトタイピングするのに便利です)。

各 DBMS には動的に作成できるステートメントがさらにある場合もありますが、この場合には DB2 CLI がステートメントを DBMS に渡します。 1 つの例外があります。一部の DBMS では COMMIT および ROLLBACK ステートメントを動的に準備できますが渡されることはありません。その代わりに `SQLEndTran()` 関数を使用して、COMMIT または ROLLBACK ステートメントのいずれかを指定する必要があります。

DB2 CLI を使用する場合の利点

DB2 CLI インターフェースには、組み込み SQL よりも優れている点がいくつかあります。

- これはクライアント・サーバー環境に理想的です。この環境では、アプリケーションの作成時には宛先データベースは不明です。アプリケーションにどのデータベース・サーバーが接続するかに関係なく、 SQL ステートメント実行用の一貫性のあるインターフェースが得られます。
- プリコンパイラーへの従属性が排除されているので、アプリケーションの移植性が高まります。
- 個々の DB2 CLI アプリケーションを各データベースにバインドする必要はなく、すべての DB2 CLI アプリケーションについて、 DB2 CLI に付いているバインド・ファイルを一度バインドする必要があるだけです。いったんこれを汎用にすると、アプリケーションに必要な管理の量を著しく減らすことができます。
- DB2 CLI アプリケーションを複数のデータベースに接続することができます。同一アプリケーションから同一データベースに複数接続することもできます。各接続に

は、各自のコミット範囲があります。アプリケーションでマルチスレッド化の使用により同一結果になる組み込み SQL を使用するよりは CLI を使用する方がずっと簡単です。

- DB2 CLI では、アプリケーション制御の、複雑になることの多いデータ域の必要がなくなります (たとえば SQLDA や SQLCA など、一般的に組み込み SQL アプリケーションに関連しているものです)。その代わりに、DB2 CLI が必要なデータ構造を割り振って制御し、アプリケーションが参照できるよう*ハンドル* を与えます。
- DB2 CLI では、マルチスレッドのスレッド保護アプリケーションの開発が可能になります。この場合、各スレッドは、独自の接続および他のスレッドとは別のコミット効力範囲を持つことができます。 DB2 CLI は、上記のデータ域を除去し、特定のハンドルでアプリケーションにアクセス可能なデータ構造のすべてを関連付けることにより、このことを成し遂げます。組み込み SQL とは違って、マルチスレッドの CLI アプリケーションではコンテキスト管理の DB2 API のいずれかを呼び出す必要はありません。これは、DB2 CLI ドライバーで自動的に操作されます。
- DB2 CLI では、拡張パラメーターの入力と取り出しの機能が備えられており、データの配列が入力時に指定され、結果セットの複数行を直接配列に取り出し、複数の結果セットを生成するステートメントを実行します。
- DB2 CLI では、スキーマ (表、列、外部キー、 1 次キーなど) 情報を照会するための一貫性のあるインターフェースが与えられます。この情報はさまざまな DBMS カタログ表に入っています。返される結果セットは DBMS 間で一貫性があります。したがって、アプリケーションは、データベース・サーバーのリリース間のカタログ変更や、さまざまなデータベース・サーバー間のカタログの違いの影響を受けません。その結果、アプリケーションでは、バージョン固有およびサーバー固有のカタログ照会は書き込まれません。
- 拡張データ変換も DB2 CLI に備えられており、さまざまな SQL と C データ・タイプ間での情報の変換時にアプリケーション・コードが少なくて済みます。
- DB2 CLI には、ODBC と X/Open CLI の両方の関数が組み込まれており、両方とも業界仕様として受け入れられています。 DB2 CLI は、最新の ISO CLI 標準にも合わせています。アプリケーション開発者がこれらの仕様で得た知識は、 DB2 CLI の開発に直接応用することができ、その逆も可能です。このインターフェースは、関数ライブラリーについての知識はあるが、ホスト言語に SQL ステートメントを組み込む製品特定の方法についてはあまり知らないプログラマーにとって、直観的に理解できるものです。
- DB2 CLI には、 DB2 ユニバーサル・データベース (つまり DB2 (MVS/ESA 版) バージョン 5 またはそれ以降) のサーバーにあるストアード・プロシージャーから生成される複数行と結果セットを取り出す機能が備えられています。しかし、この機能は DataJoiner のバージョン 2 サーバーによりアクセス可能なサーバーにストアード・プロシージャーがある場合に、組み込み SQL を使用しているバージョン 5 の DB2 ユニバーサル・データベースのクライアントのために用意されているものです。
- DB2 CLI は、サーバー側でスクロール可能なカーソルをサポートし、配列出力と共に使用することが可能です。これは、「Page Up」、「Page Down」、「Home」、お

よび「End」キーを使用するスクロール・ボックスで、データベース情報を表示する GUI アプリケーションに役立ちます。読み取り専用カーソルをスクロール可能と宣言した後、結果セットを通して 1 行または複数行単位で下方または上方に移動することができます。下記において、オフセットを指定することにより、複数の行を取り出すことも可能です。
  - 現在行
  - 結果セットの開始または終了位置
  - 以前にブックマークで設定した特定の行
- DB2 CLI アプリケーションは、 CLI および組み込み SQL アプリケーションが結果セットを記述するのと同じように、 SQL ステートメントにパラメーターを動的に記述できます。これによって、あらかじめパラメーター・マーカーのデータ・タイプを知らなくても、パラメーター・マーカーを含む SQL ステートメントを動的に処理することが可能になります。 SQL ステートメントが準備されると、記述子情報がパラメーターのデータ・タイプの詳細と共に返されます。

組み込み SQL か DB2 CLI かの決定

どのインターフェースを選択するかは、使用するアプリケーションによって決まります。

DB2 CLI は、移植性が要求される、照会ベースのグラフィカル・ユーザー・インターフェース (GUI) アプリケーションに理想的です。前述の利点のため、 DB2 CLI の方が明らかにどんなアプリケーションにもふさわしいように見えるかもしれません。しかし、考慮しなければならない要素が 1 つあります。静的 SQL と動的 SQL の比較です。組み込みアプリケーションでは静的 SQL を使用する方がはるかに簡単です。

CLI アプリケーションでの静的 SQL の使用法については、次のサイトの Web ページをご覧ください。

```
http://www.ibm.com/software/data/db2/udb/staticcli
```

静的 SQL には、次のようないくつかの利点があります。

- パフォーマンス

  動的 SQL は実行時に準備され、静的 SQL はプリコンパイル時に準備されます。より多くの処理が必要になると同時に、準備ステップのために実行時に追加のネットワーク通信量が生じることがあります。しかし、この準備ステップ (およびネットワーク通信量) は、 DB2 CLI のアプリケーションが据え置き準備の場合には、必須ではありません。

  静的 SQL が動的 SQL より常に良いパフォーマンスが得られるわけではない、というのも重要なことです。動的 SQL では、新規の索引などのデータベースに対する変更事項を利用でき、現行のデータベース統計を使って最適のアクセス・プランを選択することができます。さらに、ステートメントのプリコンパイルは、キャッシュされる場合に避けることができます。
- カプセル化およびセキュリティー

静的 SQL では、オブジェクト (表や視点など) に対する権限はパッケージに関連付けられ、パッケージのバインド時に妥当性検査されます。このため、データベース管理者は、特定のパッケージに関する実行権を 1 つのユーザーの集まりに付与する (つまり、特権をパッケージにカプセル化する) だけで済み、各データベース・オブジェクトへの明示アクセス権を付与する必要はありません。動的 SQL では、権限は実行時にステートメント単位で妥当性検査されます。したがって、ユーザーは各データベース・オブジェクトへの明示アクセス権を付与してもらわなければなりません。これによってこれらのユーザーは、アクセスする必要のないオブジェクトの部分にアクセスすることが許可されます。

- 組み込み SQL は、C または C++ 以外の言語でサポートされます。
- 固定照会の選択の場合、組み込み SQL はより簡単です。

アプリケーションで両方のインターフェースの利点が必要な場合は、静的 SQL を含むストアード・プロシージャーを作成して、 DB2 CLI アプリケーションで静的 SQL を利用できます。ストアード・プロシージャーは、DB2 CLI アプリケーション内から呼び出され、サーバーで実行されます。ストアード・プロシージャーを作成すると、どの DB2 CLI または ODBC アプリケーションでもこれを呼び出すことができます。 詳細については、*コール・レベル・インターフェースの手引きおよび解説書* を参照してください。

CLI アプリケーションでの静的 SQL の使用法については、次のサイトの Web ページをご覧ください。

```
http://www.ibm.com/software/data/db2/udb/staticcli
```

DB2 CLI と組み込み SQL の両方を使う混合アプリケーションを作成して、それぞれの利点を活用することもできます。この場合、DB2 CLI を使用して基本のアプリケーションを作成し、パフォーマンスまたはセキュリティー上の理由のために静的 SQL を使用してキー・モジュールを作成します。このためアプリケーション設計が複雑になるので、ストアード・プロシージャーがアプリケーション要件に合わない場合に限りこの方法を使用してください。 詳細については、 *コール・レベル・インターフェースの手引きおよび解説書* の『*組み込み SQL と DB2 CLI の混合*』を参照してください。

結局、それぞれのインターフェースをいつ使用するかの判断は、 1 つの要因によるというのではなく、個々の必要性と以前の経験によって決まります。

第6章 一般的な DB2 アプリケーションの技法

生成列 184
識別列 184
| 順次値の生成 185
| シーケンスの振る舞いの制御 187
| シーケンス・オブジェクトによるパフォーマンスの向上 188
| シーケンス・オブジェクトと識別列の比較 189
宣言済み一時表 189
保管点によるトランザクションの制御 . . . 191
| アプリケーションの保管点と複合 SQL ブロックの比較 193
保管点の SQL ステートメントのリスト 194
保管点の制約事項. 195
保管点およびデータ定義言語 (DDL) . . 195
保管点およびバッファー化挿入 197
カーソル・ブロック化付きの保管点の使用 197
保管点および XA に準拠したトランザクション・マネージャー 198

DB2 では、データベース・アプリケーション開発での一般的な問題を、組み込み SQL を使用して処理できるようにします。

生成列 DB2 では、GENERATED ALWAYS AS 文節を使用して、生成列を表に組み込むことができます。これにより、厄介な挿入トリガーおよび更新トリガーを使用する必要はありません。生成列は、SQL 式から得られた値を自動的に更新します。

識別列 DB2 アプリケーション開発者は、多くの場合、表の行ごとに基本キーを作成する必要があります。識別列を基本キーとして使用する表を作成すると、 DB2 は自動的に固有の値を挿入します。識別列を使用するとロック競合の数が減るので、アプリケーションのパフォーマンスが上がるという益があります。

シーケンス・オブジェクト
シーケンス・オブジェクトは、SQL ステートメントで使用するための順次値を生成するデータベース・オブジェクトです。

宣言済み一時表
宣言済み一時表は通常の表に類似したものですが、データベースに接続している間のみ存在するものです。この表に対してロッキングをすることや、ログ記録を取ることはできません。アプリケーションで多量のデータを処理するために表を作成して、アプリケーションによるデータの操作が終了する際にそれらの表を除去する場合には、宣言済み一時表を使用することを検討してください。宣言済み一時表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、並行ユーザー用のアプリケーションでは、より簡単にアプリケーション開発を行うことができます。

外部保管点
COMMIT および ROLLBACK ステートメントによりトランザクション全体の振る舞いを制御することができますが、保管点を使うとトランザクション内でより細分化された制御を行うことが可能です。保管点ブロックは、いくつかの SQL ステートメントをグループ化します。保管点ブロックにあるサブステート

© Copyright IBM Corp. 1993, 2001

メントの 1 つでエラーが起きる場合には、障害が起きているサブステートメントだけをロールバックしてから、他のサブステートメントの作業を完了できます。

生成列

生成列とは、各行の値を挿入操作または更新操作からではなく、式から派生する列です。更新トリガーおよび挿入トリガーを組み合わせて使用すると同様のことが行えますが、生成列を使用すると、派生した値が式と一貫したものであることを保証できます。

表で生成列を作成するには、列で GENERATED ALWAYS AS 文節を使用して、列の値が派生する式を含めてください。 GENERATED ALWAYS AS 文節は、 ALTER TABLE または CREATE TABLE ステートメントに含められます。次の例では、 “c1” および “c2” という通常の 2 つの列と、表の通常の列から派生した “c3” および “c4” という 2 つの生成された列の入った表を作成します。

```
CREATE TABLE T1(c1 INT, c2 DOUBLE,
                c3 DOUBLE GENERATED ALWAYS AS (c1 + c2),
                c4 GENERATED ALWAYS AS
                  (CASE
                     WHEN c1 > c2 THEN 1
                     ELSE NULL
                   END)
               );
```

アプリケーションのパフォーマンスを向上させるための生成列の使用法については、 *管理の手引き* を参照してください。生成列を作成する方法については、 *SQL 解説書* の CREATE TABLE ステートメントを参照してください。

識別列

DB2 アプリケーション開発者は、識別列を使用して表の各行に対して数値列の値を自動的に生成することができます。この値を固有値として生成し、識別列を基本キーとして定義できます。識別列を作成するには、 CREATE TABLE または ALTER TABLE ステートメントに IDENTITY 文節を含めてください。

アプリケーションがデータベースの外に独自のカウンターを生成する際に生じる、並行性およびパフォーマンス上の問題を回避するため、アプリケーション内で識別列を使用してください。固有な基本キーを自動生成するのに識別列を使用しない場合には、単一行の表にカウンターを保管するのが一般的な設計方法です。各トランザクションはこの表をロックして、数を増分してからトランザクションをコミットして、カウンターのロックを解除します。しかし、残念ながら、この設計では、カウンターを増分できるのは一度に 1 つのトランザクションのみです。

それとは対照的に、識別列を使用して基本キーを自動的に生成すると、アプリケーションでより高度なレベルの並行性を実現できます。識別列では、トランザクションがカウ

ンターをロックしなくてもいいように DB2 はカウンターを保持します。カウンターを増分したコミットされていないトランザクションが、他の後続するトランザクションによるカウンターの増分を阻止しないので、識別列を使用するアプリケーションはパフォーマンスが向上します。

識別列のカウンターは、トランザクションに関係なく増分されたり減分されたりします。あるトランザクションが識別カウンターを 2 回増分する場合、他のトランザクションが同一の識別カウンターを並行して増分することがあるので、生成される 2 つの数の値には差があるかもしれません。

トランザクションがロールバックされたため、またはキャッシュに入れられた値すべてが割り当てられる前に、非活動化された (正常にまたは異常に) 値がデータベースによってキャッシュに入れられたため、識別列のカウンターの数が離れたものになる場合があります。

識別列を持つ表に新規行を挿入した後で、生成された値を検索するには、`identity_val_local()` 関数を使用します。

識別列の詳細については、 *管理の手引き* を参照してください。 CREATE TABLE および ALTER TABLE ステートメントの IDENTITY 文節についての詳細は、 *SQL 解説書* を参照してください。

順次値の生成

順次値を生成することは、一般的なデータベース・アプリケーション開発の問題です。この問題を解決する最善の方法は、SQL でシーケンス・オブジェクトとシーケンス式を使用することです。各シーケンス・オブジェクト は、固有の名前が付けられたデータベース・オブジェクトであり、シーケンス式によってのみアクセスできます。シーケンス式 には、 PREVVAL 式と NEXTVAL 式の 2 つがあります。 PREVVAL 式は、アプリケーション・プロセスで、指定されたシーケンス・オブジェクトについて生成された最新の値を返します。 PREVAL 式と同じステートメントで発生する NEXTVAL 式は、そのステートメントの PREVAL 式で生成された値に対して影響を与えません。 NEXTVAL シーケンス式は、シーケンス・オブジェクトの値を増やして、そのシーケンス・オブジェクトの新しい値を返します。

シーケンス・オブジェクトを作成するには、CREATE SEQUENCE ステートメントを発行します。たとえば、デフォルトの属性を使用して id_values というシーケンス・オブジェクトを作成するには、次のステートメントを発行します。

```
CREATE SEQUENCE id_values
```

アプリケーション・セッションで、シーケンス・オブジェクトの最初の値を生成するには、次のように NEXTVAL 式を使用して VALUES ステートメントを発行します。

```
VALUES NEXTVAL FOR id_values

1
-----------
          1

            1 record(s) selected.
```

シーケンス・オブジェクトの現行値を表示するには、 PREVVAL 式を使用して VALUES ステートメントを発行します。

```
VALUES PREVVAL FOR id_values

1
-----------
          1

            1 record(s) selected.
```

シーケンス・オブジェクトの現行値は繰り返し検索することができます。シーケンス・オブジェクトが返す値は、NEXTVAL 式を発行するまで変わりません。以下の例では、アプリケーション・プロセスの NEXTVAL 式がシーケンス・オブジェクトの値を増やすまで、 PREVVAL は値 1 を返します。

```
VALUES PREVVAL FOR id_values

1
-----------
          1

            1 record(s) selected.

VALUES PREVVAL FOR id_values

1
-----------
          1

            1 record(s) selected.

VALUES NEXTVAL FOR id_values

1
-----------
          2

            1 record(s) selected.

VALUES PREVVAL FOR id_values

1
```

```
-----------
          2

            1 record(s) selected.
```

シーケンス・オブジェクトの次の値で列の値を更新するには、次のように UPDATE ステートメントに NEXTVAL 式を組み込みます。

```
UPDATE staff
  SET id = NEXTVAL FOR id_values
  WHERE id = 350
```

シーケンス・オブジェクトの次の値を使用して新しい行を表に挿入するには、次のように INSERT ステートメントに NEXTVAL 式を組み込みます。

```
INSERT INTO staff (id, name, dept, job)
  VALUES (NEXTVAL FOR id_values, 'Kandil', 51, 'Mgr')
```

PREVVAL 式と NEXTVAL 式について詳しくは、*SQL 解説書* を参照してください。

シーケンスの振る舞いの制御

アプリケーションの要求を満たすようにシーケンス・オブジェクトの振る舞いを調整することができます。 CREATE SEQUENCE ステートメントを発行して新しいシーケンス・オブジェクトを作成する場合、および既存のシーケンス・オブジェクトに対して ALTER SEQUENCE を発行する場合は、シーケンス・オブジェクトの属性を変更します。指定可能なシーケンス・オブジェクトの属性のいくつかを以下に示します。

データ・タイプ
CREATE SEQUENCE ステートメントの AS 文節は、シーケンス・オブジェクトの数値データ・タイプを指定します。 *SQL 解説書* の付録『SQL 制限』に示されているように、このデータ・タイプはシーケンス・オブジェクトの使用可能な最小値と最大値を決定します。シーケンス・オブジェクトのデータ・タイプを変更することはできません。代わりに、DROP SEQUENCE ステートメントを発行してから新しいデータ・タイプで CREATE SEQUENCE ステートメントを発行することによりシーケンス・オブジェクトをドロップする必要があります。

開始値
CREATE SEQUENCE ステートメントの START WITH 文節は、シーケンス・オブジェクトの初期値を設定します。 ALTER SEQUENCE ステートメントの RESTART WITH 文節は、シーケンス・オブジェクトの値を指定値にリセットします。

最小値
MINVALUE 文節は、シーケンス・オブジェクトの最小値を設定します。

最大値
MAXVALUE 文節は、シーケンス・オブジェクトの最大値を設定します。

増分値
INCREMENT BY 文節は、各 NEXTVAL 式がシーケンス・オブジェクトの現行値に追加する値を設定します。シーケンス・オブジェクトの値を減らすには、負の値を指定します。

シーケンス循環

CYCLE 文節は、シーケンス・オブジェクトの値が最小値または最大値に達したとき、次の NEXTVAL 式でそれぞれ最小値または最大値を初期値に戻します。

たとえば、各 NEXTVAL 式で開始時の最小値が 0、最大値が 1000、増分値が 2 で、最大値に達したときに最小値に戻る id_values というシーケンス・オブジェクトを作成するには、次のステートメントを発行します。

```
CREATE SEQUENCE id_values
  START WITH 0
  INCREMENT BY 2
  MAXVALUE 1000
  CYCLE
```

CREATE SEQUENCE ステートメントと ALTER SEQUENCE ステートメントについて詳しくは、 *SQL 解説書* を参照してください。

シーケンス・オブジェクトによるパフォーマンスの向上

識別列のように、シーケンス・オブジェクトを使用して値を生成する場合、一般に、他の方法と比べてアプリケーションのパフォーマンスが向上します。シーケンス・オブジェクトを制御する別の方法として、現行値を保管する単一列表を作成し、トリガーを使用して、またはアプリケーションの制御下でその値を増やす方法があります。単一列表にアプリケーションが並行してアクセスする分散環境では、順番に表にアクセスすることを強制するために必要になるロックが、パフォーマンスに大きく影響します。

シーケンス・オブジェクトは、単一列表を使用する方法に関連するロック発行を行わずに、シーケンス値をメモリーにキャッシュして DB2 の応答時間を改善することができます。シーケンス・オブジェクトを使用するアプリケーションのパフォーマンスを最大にするには、シーケンス・オブジェクトが適切な量のシーケンス値を確実にキャッシュするようにします。 CREATE SEQUENCE ステートメントおよび ALTER SEQUENCE ステートメントの CACHE 文節は、 DB2 が生成してメモリーに保管するシーケンス値の最大数を指定します。

シーケンス・オブジェクトが順序正しく値を生成する必要があり、システム障害またはデータベース非活動化でその順序が途切れないようにする場合、 ORDER および NO CACHE 文節を CREATE SEQUENCE ステートメントで使用します。 NO CACHE 文節は、生成された値が途切れないことを保証します。この場合、シーケンス・オブジェクトが新しい値を生成するたびにデータベース・ログに書き込むため、アプリケーションのパフォーマンスが低下します。トランザクションがロールバックし、要求したシーケンス値を実際には使用しないために、依然としてギャップが存在する場合があることに注意してください。

シーケンス・オブジェクトと識別列の比較

シーケンス・オブジェクトと識別列は DB2 アプリケーションに対して同じような目的を果たすために使用されているように見えますが、重要な違いがあります。識別列は、単一表の列の値を自動的に生成します。シーケンス・オブジェクトは、SQL ステートメントで使用可能な順次値を要求時に生成します。

宣言済み一時表

宣言済み一時表 とは、一時表を作成したアプリケーションによって発行された SQL ステートメントにのみアクセス可能な一時表を表します。宣言済み一時表は、アプリケーションがデータベースに接続している間しか有効ではありません。

宣言済み一時表を使用して、アプリケーションの潜在的パフォーマンスの向上を図ってください。宣言済み一時表を作成する場合には、 DB2 はシステム・カタログ表に項目を挿入しないため、サーバーでカタログの競合による問題が起きることはありません。通常の表の場合とは異なり、 DB2 は宣言済み一時表またはその行をロックせず、宣言済み一時表またはその表の内容をログに記録しません。現行のアプリケーションで多量のデータを処理するために表を作成し、アプリケーションによるデータの操作が終了する際にそれらの表を除去する場合には、通常の表のかわりに宣言済み一時表を使用することを検討してください。

並行ユーザー用にアプリケーションを開発する場合には、宣言済み一時表が役立ちます。通常の表とは異なり、宣言済み一時表では名前が重複しても問題は起こりません。アプリケーションの各インスタンスでは、 DB2 は同じ名前の宣言済み一時表を作成することができます。たとえば、多量の一時データを処理するのに通常の表を使用する並行ユーザーのためにアプリケーションを作成する場合には、アプリケーションの各インスタンスが一時データを保持する通常の表に対して固有な名前を使用する必要があります。一般的には、ある時点で使用されている表の名前をたどる別の表を作成します。しかし、宣言済み一時表を使用すると、一時データに対して 1 つの宣言済み一時表を指定するだけで済みます。 DB2 は、アプリケーションの各インスタンスで固有な表が使用されていることを保証します。

宣言済み一時表を使用するには、次のようなステップを実行します。

ステップ 1. USER TEMPORARY TABLESPACE が存在することを確認する。 USER TEMPORARY TABLESPACE が存在しない場合には、 CREATE USER TEMPORARY TABLESPACE ステートメントを発行します。

ステップ 2. アプリケーションで DECLARE GLOBAL TEMPORARY TABLE ステートメントを発行する。

宣言済み一時表のスキーマは必ず SESSION になります。 SQL ステートメントで宣言済み一時表を使用するには、 SESSION スキーマ修飾子を使用して表を明示的に参照するか、 SESSION の DEFAULT スキーマを使用して修飾されていない参照を修飾して

ください。次の例では、以下のステートメントで TT1 という宣言済み一時表を作成すると、表の名前は必ず SESSION というスキーマ名で修飾されます。

```
DECLARE GLOBAL TEMPORARY TABLE TT1
```

前述の例で作成された宣言済み一時表から *column1* 列の内容を選択するには、次のようなステートメントを使用します。

```
SELECT column1 FROM SESSION.TT1;
```

DB2 では、SESSION スキーマを使用して持続する表も作成できることに注目してください。 SESSION.TT3 という修飾名で持続する表を作成すると、 SESSION.TT3 という修飾名の宣言済み一時表を作成できます。この場合、同一の修飾名を持つ持続表および宣言済み一時表への参照は、 DB2 によって宣言済み一時表に解決されます。持続表と宣言済み一時表の混同を避けるには、 SESSION スキーマを使用して持続表を作成しないでください。

SESSION スキーマで修飾された表、ビュー、または別名への静的 SQL 参照を含むアプリケーションを作成する場合、 DB2 プリコンパイラーはバインド実行時にはそのステートメントをコンパイルせずに、そのステートメントに対して「コンパイルが必要」なことを記します。そして、実行時に DB2 はそのステートメントをコンパイルします。この動作は、*増分バインド* として知られています。 DB2 は、SESSION スキーマによって修飾されている表、ビュー、および別名への静的 SQL 参照の増分バインドを自動的に実行します。これらのステートメントで増分バインドが使用可能になるように、BIND または PRECOMPILE コマンドで VALIDATE RUN オプションを指定する必要はありません。

トランザクションに対して DECLARE GLOBAL TEMPORARY TABLE ステートメントを含む ROLLBACK ステートメントを出すと、 DB2 は宣言済み一時表を除去します。宣言済み一時表に DROP TABLE ステートメントを出すと、そのトランザクションに ROLLBACK ステートメントを出しても空の宣言済み一時表が復元されるだけです。DROP TABLE ステートメントの ROLLBACK は、宣言済み一時表の行を復元しません。

宣言済み一時表のデフォルトの振る舞いでは、トランザクションがコミットされる際に表からすべての行が削除されます。しかし、1 つまたは複数の WITH HOLD カーソルが宣言済み一時表でまだオープンされている場合には、トランザクションをコミットしても DB2 は表から行を削除しません。トランザクションをコミットする際にすべての行が削除されないようにするには、 DECLARE GLOBAL TEMPORARY TABLE で ON COMMIT PRESERVE ROWS 文節を使用して一時表を作成してください。

トランザクション内で INSERT、UPDATE、または DELETE ステートメントを使用して宣言済み一時表の内容を変更した後に、そのトランザクションをロールバックした場合、 DB2 は宣言済み一時表のすべての行を削除します。 INSERT、UPDATE、または

DELETE ステートメントを使用して宣言済み一時表の内容を変更しようして、そのステートメントが失敗すると、 DB2 は宣言済み一時表のすべての行を削除します。

区分データベース環境では、ノードで障害が発生すると、その障害が発生したノードに区分が存在する宣言済み一時表は使用できなくなります。その後、これらの使用不能な宣言済み一時表にアクセスしようとすると、エラー (SQL1477N) が起きます。アプリケーションによって使用不能な宣言済み一時表が検出されると、アプリケーションはその表を除去するか、 DECLARE GLOBAL TEMPORARY TABLE ステートメントで WITH REPLACE 文節を指定して、その表を再作成することができます。

宣言済み一時表には、いくつかの制限があります。たとえば、宣言済み一時表では索引、別名、またはビューを定義することができません。 IMPORT および LOAD を使用して宣言済み一時表を移植することもできません。 DECLARE GLOBAL TEMPORARY TABLE ステートメントの完全な構文、および宣言済み一時表の制限についての完全なリストについては、 *SQL 解説書* を参照してください。

保管点によるトランザクションの制御

アプリケーションの保管点は、トランザクションまたは作業単位において、 SQL ステートメントのサブセットによって実行される作業に対して制御を実施します。アプリケーション内で保管点を設定し、後でその保管点を解放するか、または保管点を設定した後で実行された作業をロールバックすることができます。 1 つのトランザクション内で複数の保管点を使用できますが、それらの保管点をネストすることはできません。次の例では、1 つのトランザクション内で 2 つの保管点を使用して、アプリケーションの振る舞いを制御しています。

アプリケーションの保管点を使用したオーダーの例:

```
INSERT INTO order ...
INSERT INTO order_item ... lamp

-- set the first savepoint in the transaction
SAVEPOINT before_radio ON ROLLBACK RETAIN CURSORS
  INSERT INTO order_item ... Radio
  INSERT INTO order_item ... Power Cord
  -- Pseudo-SQL:
  IF SQLSTATE = "No Power Cord"
    ROLLBACK TO SAVEPOINT before_radio
RELEASE SAVEPOINT before_radio

-- set the second savepoint in the transaction
SAVEPOINT before_checkout ON ROLLBACK RETAIN CURSORS
  INSERT INTO order ... Approval
  -- Pseudo-SQL:
  IF SQLSTATE = "No approval"
    ROLLBACK TO SAVEPOINT before_checkout

-- commit the transaction, which releases the savepoint
COMMIT
```

前述の例では、最初の保管点は 2 つのデータ・オブジェクトの間の従属関係を強制します。その従属関係はオブジェクト自体に組み込まれているわけではありません。ラジオと電源コードの間の関係を説明するために参照保全は使用しません。一方が存在して他方が存在しないことがあるからです。しかし、電源コードを付けないでラジオを顧客に出荷することはありませんし、ラジオの電源コードがないからといってトランザクション全体をロールバックし、電気スタンドのオーダーまでキャンセルしようとも思わないでしょう。アプリケーションの保管点は、このオーダーを完了するために必要な事細かな制御を提供します。

ROLLBACK TO SAVEPOINT ステートメントを発行するときに、対応する保管点は自動的に解放されるわけではありません。 RELEASE SAVEPOINT ステートメントを使用して保管点が明示的に解放されるか、トランザクションまたは作業単位を終了することによって暗黙的に解放されるまで、後続の SQL ステートメントがその保管点に関連付けられます。これは、1 つの保管点に対して複数の ROLLBACK TO SAVEPOINT ステートメントを発行できることを意味します。

保管点によって、複数の COMMIT および ROLLBACK ステートメントを使用するよりもパフォーマンスは向上し、アプリケーション設計の見栄えが良くなります。
COMMIT ステートメントを発行するときに、DB2 は付加的な作業を行って、現行トランザクションをコミットし、新規のトランザクションを開始する必要があります。保管点を使用すると、複数の COMMIT ステートメントの発行という追加の作業を必要とせずに、トランザクションを小さい単位またはステップに分割することができます。次の例は、保管点の代わりに複数のトランザクションを使用することによって発生する、パフォーマンス上の問題点を示します。

複数のトランザクションを使用したオーダーの例:

```
INSERT INTO order ...
INSERT INTO order_item ... lamp
-- commit current transaction, start new transaction
COMMIT

INSERT INTO order_item ... Radio
INSERT INTO order_item ... Power Cord
-- Pseudo-SQL:
IF SQLSTATE = "No Power Cord"
  -- roll back current transaction, start new transaction
  ROLLBACK
ELSE
  -- commit current transaction, start new transaction
  COMMIT

INSERT INTO order ... Approval
-- Pseudo-SQL:
IF SQLSTATE = "No approval"
  -- roll back current transaction, start new transaction
```

```
  ROLLBACK
ELSE
  -- commit current transaction, start new transaction
  COMMIT
```

複数のコミット・ポイントを使用する別の問題は、オブジェクトがコミットされている可能性があるため、それが完全に完了する前に他のアプリケーションに見えてしまうことです。 192ページでは、すべてのアイテムが追加される前に、またさらに悪いことにオーダーが承認される前に、その注文を別のユーザーが利用できるようになっています。アプリケーションの保管点を使用すると、「不正データ」に対するこのような公開を避けながら、操作に対して事細かな制御を行うことができます。

アプリケーションの保管点と複合 SQL ブロックの比較

保管点は、複合 SQL ブロックと比較して、次の利点を提供しています。

- トランザクションの制御の拡張
- ロック競合の減少化
- アプリケーション・ロジックとの統合の向上

複合 SQL ブロックは ATOMIC の場合と NOT ATOMIC の場合があります。ATOMIC 複合 SQL ブロック内のステートメントが失敗すると、複合 SQL ブロック全体がロールバックされます。 NOT ATOMIC 複合 SQL ブロック内のステートメントが失敗すると、トランザクションのコミットまたはロールバック (複合 SQL ブロック全体を含む) がアプリケーションによって制御されます。それと比較して、保管点の効力範囲内の 1 つのステートメントが失敗すると、アプリケーションは保管点の効力範囲内のすべてのステートメントをロールバックできます。しかし、保管点の効力範囲外にあるステートメントによって実行された作業はコミットできません。このオプションは、191ページで説明されています。保管点の作業がロールバックされると、保管点の前に 2 つの INSERT ステートメントの作業がコミットされます。あるいは、アプリケーションは、保管点の効力範囲内にあるステートメントを含む、トランザクション内のすべてのステートメントによって実行された作業をコミットできます。

複合 SQL ブロックを発行すると同時に、DB2 はステートメントの複合 SQL ブロック全体に必要なロックを獲得します。アプリケーションの保管点を設定すると、保管点の効力範囲内のステートメントが発行されるたびに、 DB2 はロックを獲得します。保管点のロック動作は、複合 SQL ブロックの場合よりもロック競合の大幅な減少につながります。そのため、アプリケーションが複合 SQL ステートメントによって実行されたロックを必要としない場合は、保管点を使用するのが最善です。

複合 SQL ブロックはステートメントの完全セットを単一のステートメントとして実行します。アプリケーションは、ステートメントを複合 SQL ブロックに追加するために、制御構造または関数を使用することはできません。それと比較して、アプリケーションの保管点を設定すると、アプリケーションは、他のアプリケーション関数またはメソッドを呼び出すことにより、 while ループなどの制御構造を介して、または動的

SQL ステートメントを使用して、保管点の効力範囲内の SQL ステートメントを発行することができます。アプリケーションの保管点は、 SQL ステートメントをアプリケーション・ロジックと直感的な方法で自由に統合できるようにします。

たとえば、194 ページでは、アプリケーションは保管点を設定し、保管点の効力範囲内で 2 つの INSERT ステートメントを発行します。アプリケーションは IF ステートメントを使用し、それが真であれば関数 add_batteries() を呼び出します。 add_batteries() 関数は、このコンテキストで保管点の効力範囲に含まれている SQL ステートメントを発行します。最後に、アプリケーションは、保管点内で実行された作業 (add_batteries() 関数によって発行された SQL ステートメントを含む) をロールバックするか、またはトランザクション全体で実行された作業をコミットします。

保管点と SQL ステートメントをアプリケーション・ロジック内で統合する例:

```
void add_batteries()
{
  -- the work performed by the following statement
  -- is controlled by the savepoint set in main()
  INSERT INTO order_item ... Batteries
}

void main(int argc, char[] *argv)
{
  INSERT INTO order ...
  INSERT INTO order_item ... lamp

  -- set the first savepoint in the transaction
  SAVEPOINT before_radio ON ROLLBACK RETAIN CURSORS
    INSERT INTO order_item ... Radio
    INSERT INTO order_item ... Power Cord

    if (strcmp(Radio..power_source(), "AC/DC"))
    {
      add_batteries();
    }

    -- Pseudo-SQL:
    IF SQLSTATE = "No Power Cord"
      ROLLBACK TO SAVEPOINT before_radio
  COMMIT
}
```

保管点の SQL ステートメントのリスト

次の SQL ステートメントで保管点を作成および制御することができます。

SAVEPOINT

保管点を設定するには、 SAVEPOINT SQL ステートメントを発行します。コードをより鮮明なものにするには、保管点のために分かりやすい名前を選ぶことができます。以下に例を示します。

```
SAVEPOINT savepoint1 ON ROLLBACK RETAIN CURSORS
```

RELEASE SAVEPOINT

保管点を解放するには、 RELEASE SAVEPOINT SQL ステートメントを発行します。 RELEASE SAVEPOINT SQL ステートメントを使用して保管点を明示的に解放しない場合には、トランザクションの終わりで解放されます。以下に例を示します。

```
RELEASE SAVEPOINT savepoint1
```

ROLLBACK TO SAVEPOINT

保管点へロールバックするには、 ROLLBACK TO SAVEPOINT という SQL ステートメントを発行します。以下に例を示します。

```
ROLLBACK TO SAVEPOINT
```

SAVEPOINT、RELEASE SAVEPOINT、および ROLLBACK TO SAVEPOINT ステートメントの完全な構文については、 *SQL 解説書* を参照してください。

保管点の制約事項

DB2 ユニバーサル・データベースでは、アプリケーションでの保管点の使用に関して次のような制約があります。

アトミック複合 SQL

DB2 では、アトミック複合 SQL での保管点は使用できません。保管点内ではアトミック複合 SQL を使用することはできません。

ネストされた保管点

DB2 では、別の保管点での保管点の使用をサポートしておりません。

トリガー

DB2 では、トリガーでの保管点の使用をサポートしておりません。

SET INTEGRITY ステートメント

保管点内では、 DB2 は SET INTEGRITY ステートメントを DDL ステートメントとして扱います。保管点での DDL の使用の詳細については、『保管点およびデータ定義言語 (DDL)』 を参照してください。

保管点およびデータ定義言語 (DDL)

DB2 は保管点内に DDL ステートメントを組み込むことを可能にします。 DDL ステートメントを実行する保管点をアプリケーションが正常に解放すると、そのアプリケーションは DDL によって作成された SQL オブジェクトを継続的に使用することができます。ただし、DDL ステートメントを実行する保管点に対してアプリケーションが ROLLBACK TO SAVEPOINT ステートメントを発行すると、 DB2 はこれらの DDL ステートメントの効力に依存するカーソルをすべて無効とマーク付けします。

次の例では、アプリケーションは ROLLBACK TO SAVEPOINT ステートメントを発行した後に、以前に開かれた 3 つのカーソルからフェッチすることを試みます。

```
SAVEPOINT savepoint_name;
   PREPARE s1 FROM 'SELECT FROM t1';
   --issue DDL statement for t1
      ALTER TABLE t1 ADD COLUMN...
   PREPARE s2 FROM 'SELECT FROM t2';
   --issue DDL statement for t3
      ALTER TABLE t3 ADD COLUMN...
   PREPARE s3 FROM 'SELECT FROM t3';
   OPEN c1 USING s1;
   OPEN c2 USING s2;
   OPEN c3 USING s3;
ROLLBACK TO SAVEPOINT
FETCH c1; --invalid (SQLCODE -910)
FETCH c2; --successful
FETCH c3; --invalid (SQLCODE -910)
```

ROLLBACK TO SAVEPOINT ステートメントでは、 DB2 は "c1" および "c3" カーソルを無効としてマーク付けします。なぜなら、それらのカーソルが依存していた SQL オブジェクトが保管点内の DDL ステートメントによって操作されたからです。しかし、この例の "c2" カーソルを使用した FETCH は、ROLLBACK TO SAVEPOINT ステートメントの後でも正常に実行されます。

無効なカーソルをクローズするには、 CLOSE ステートメントを発行します。無効なカーソルに対して FETCH を発行すると、 DB2 によって SQLCODE -910 が戻されます。無効なカーソルに対して OPEN ステートメントを発行すると、 DB2 によって SQLCODE -502 が戻されます。無効なカーソルに対して UPDATE または DELETE WHERE CURRENT OF ステートメントを発行すると、 DB2 によって SQLCODE -910 が戻されます。

保管点内では、 DB2 は NOT LOGGED INITIALLY 特性を持つ表と一時表を次のように扱います。

NOT LOGGED INITIALLY 表

保管点内では、 NOT LOGGED INITIALLY 特性付きの表を作成したり、 NOT LOGGED INITIALLY を持つように表を変更することができます。しかし、これらの保管点では、 DB2 は ROLLBACK TO SAVEPOINT ステートメントを ROLLBACK WORK ステートメントとして扱い、トランザクション全体をロールバックします。

保管点内での DECLARE TEMPORARY TABLE

保管点内で一時表が宣言されると、 ROLLBACK TO SAVEPOINT ステートメントによって一時表が除去されます。

保管点外での DECLARE TEMPORARY TABLE

保管点外で一時表が宣言されると、 ROLLBACK TO SAVEPOINT ステートメントは一時表を除去しません。

保管点およびバッファー化挿入

DB2 アプリケーションのパフォーマンスを向上させるには、 INSERT BUF オプションを使用してバインドを行うことによって、アプリケーションでバッファー化挿入を使用できます。アプリケーションがバッファー化挿入および保管点の両方を利用する場合、DB2 は SAVEPOINT、RELEASE SAVEPOINT、OR ROLLBACK TO SAVEPOINT ステートメントを実行する前にバッファーをフラッシュします。

アプリケーション内でバッファー化挿入を使用する方法についての詳細は、571ページの『バッファー化挿入の使用』を参照してください。アプリケーションのプリコンパイルおよびバインドについての詳細は、 *コマンド解説書* を参照してください。

カーソル・ブロック化付きの保管点の使用

アプリケーションが保管点を使用する場合、 BLOCKING NO というプリコンパイル・オプションを指定することにより、プリコンパイルまたはバインドを行うときにカーソル・ブロックを回避することを考慮してください。カーソルをブロック化すると、複数の行が事前に取り出されるのでアプリケーションのパフォーマンスは向上しますが、保管点およびブロック化カーソルを使用するアプリケーションによって戻されたデータは、データベースにコミットされたデータを反映しない場合があります。

BLOCKING NO を使用してアプリケーションをプリコンパイルしないで、 ROLLBACK TO SAVEPOINT が発生した後に FETCH ステートメントを発行する場合には、FETCH ステートメントは削除されたデータを取り出すかもしれません。たとえば、次のような SQL を含んだアプリケーションが BLOCKING NO オプションなしでプリコンパイルされたとします。

```
CREATE TABLE t1(c1 INTEGER);
DECLARE CURSOR c1 AS 'SELECT c1 FROM t1 ORDER BY c1';
INSERT INTO t1 VALUES (1);
SAVEPOINT showFetchDelete;
   INSERT INTO t1 VALUES (2);
   INSERT INTO t1 VALUES (3);
   OPEN CURSOR c1;
   FETCH c1; --get first value and cursor block
   ALTER TABLE t1... --add constraint
ROLLBACK TO SAVEPOINT;
FETCH c1; --retrieves second value from cursor block
```

アプリケーションが “t1” 表に対して最初の FETCH を発行すると、 DB2 サーバーは列値 (1、2、および 3) のブロックをクライアント・アプリケーションに送ります。これらの列値はクライアントによってローカルで保管されます。アプリケーションが ROLLBACK TO SAVEPOINT SQL ステートメントを発行すると、列値 '2' および '3' が表から削除されます。 ROLLBACK TO SAVEPOINT ステートメント後は、表で次の FETCH が行われると、表にその値がないとしても、列値 '2' が戻されます。アプリケーションはカーソル・ブロック化オプションを利用してパフォーマンスを向上させ、ローカルに保管したデータにアクセスするために、この値を受け取ります。

アプリケーションのプリコンパイルおよびバインドについての詳細は、 *コマンド解説書* を参照してください。

保管点および XA に準拠したトランザクション・マネージャー

XA に準拠するトランザクション・マネージャーが XA_END 要求を発行する際にアプリケーションでアクティブな保管点が存在する場合には、 DB2 は RELEASE SAVEPOINT ステートメントを発行します。

第3部 ストアード・プロシージャー

© Copyright IBM Corp. 1993, 2001

第7章 ストアード・プロシージャー

ストアード・プロシージャーの概説 201
ストアード・プロシージャーの利点 202
ストアード・プロシージャーの作成 205
クライアント・アプリケーション . . . 207
ホスト変数の割り振り 207
ストアード・プロシージャーの呼び出し 207
クライアント・アプリケーションの実行 208
サーバー上でのストアード・プロシージャー 208
ストアード・プロシージャーの登録 208
変数宣言と CREATE PROCEDURE の例 222
ストアード・プロシージャーの SQL ステートメント 223
ネストされたストアード・プロシージャー 224
再帰的ストアード・プロシージャーでのカーソルの使用 225
制約事項 226
OLE 自動化ストアード・プロシージャーの作成 227
OUT パラメーターのストアード・プロシージャーの例 228
OUT クライアントの説明 230
OUT クライアント・アプリケーションの例: Java 233
OUT クライアント・アプリケーションの例: C 235
OUT ストアード・プロシージャーの説明 236
OUT パラメーターのストアード・プロシージャーの例: Java 237
OUT パラメーターのストアード・プロシージャーの例: C 239
コード・ページに関する考慮事項 . . . 241
C++ に関する考慮事項 241
グラフィック・ホスト変数に関する考慮事項 242
マルチサイト更新に関する考慮事項 . . . 242
ストアード・プロシージャーのパフォーマンスの向上 242
CHAR パラメーターではなく VARCHAR パラメーターを使用する 243
DB2 がシステム・カタログ内のストアード・プロシージャーを検索するように強制する 243
NOT FENCED ストアード・プロシージャー 244
ストアード・プロシージャーからの結果セットの戻り 246
例: ストアード・プロシージャーからの結果セットを戻す 247
C の例: SPSERVER.SQC (one_result_set_to_client) 249
Java の例: Spserver.java (resultSetToClient) 250
問題の解決 257

ストアード・プロシージャーの概説

ストアード・プロシージャーを使用すると、クライアント / サーバー・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。*ストアード・プロシージャー* とは、データベース・サーバーでアクセス可能な共用ライブラリーの関数のことです。ストアード・プロシージャーはデータベースをローカルにアクセスして、クライアント・アプリケーション に情報を戻します。ストアード・プロシージャーによって、リモート・アプリケーションがサーバーに複数の SQL ステートメントを渡すときに生じるオーバーヘッドを減らすことができます。クライアント・アプリケーションは、単一の CALL

© Copyright IBM Corp. 1993, 2001

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し、データベースへのアクセス作業を実行します。そして、その結果をクライアント・アプリケーションに戻します。

SQL プロシージャー という SQL を使用して、ストアード・プロシージャーを作成することができます。 SQL プロシージャーの作成の詳細については、 259ページの『第8章 SQL プロシージャーの作成』を参照してください。また、C または Java などの言語を使用してストアード・プロシージャーを作成することができます。ストアード・プロシージャーと同じ言語でクライアント・アプリケーションを作成する必要はありません。クライアント・アプリケーションとストアード・プロシージャーの言語が異なる場合には、クライアントとストアード・プロシージャー間の値の受け渡しは DB2 によって透過的に行われます。

DB2 ストアード・プロシージャー・ビルダー (SPB) を使用すると、 Java または SQL ストアード・プロシージャーを開発するのに役立ちます。 SPB を一般的なアプリケーション開発ツール (Microsoft Visual Studio および IBM Visual Age for Java など) を使って組み込むことができます。あるいは、独立したユーティリティーとして使用することもできます。ストアード・プロシージャーを作成する助けとして、 SPB では基本設計パターンの説明、SQL 照会の作成、ストアード・プロシージャーを呼び出す際にかかるパフォーマンス上のコストの推定などを行う設計援助機能を備えています。

DB2 ストアード・プロシージャーの詳細については、 283ページの『第9章 IBM DB2 ストアード・プロシージャー・ビルダー』を参照してください。

ストアード・プロシージャーの利点

図3 には、一般的なデータベース・マネージャー・アプリケーションがデータベース・サーバー上にあるデータベースにアクセスする方法が示されています。

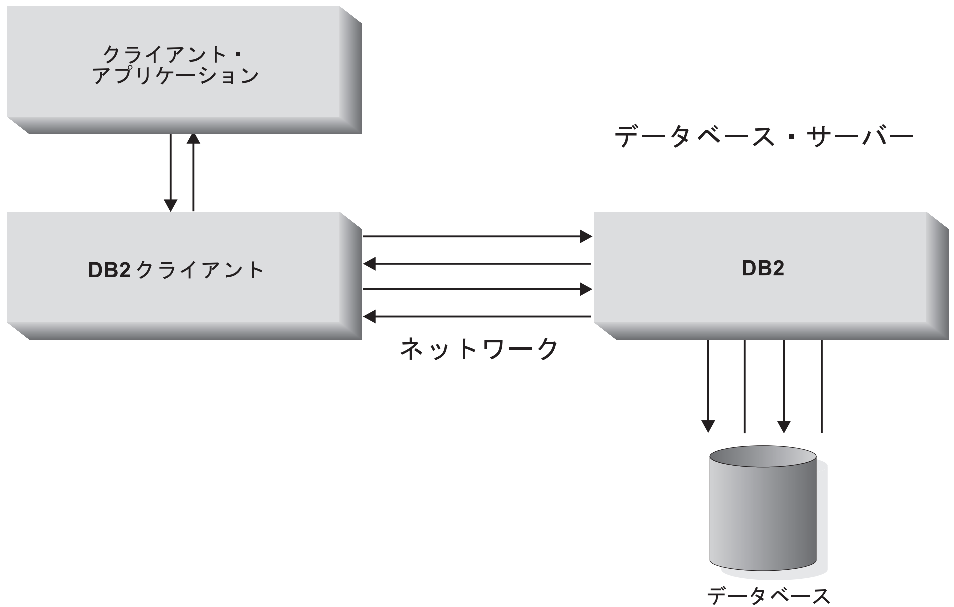


図 3. サーバー上のデータベースにアクセスするアプリケーション

データベースへのアクセスは、すべてネットワークを介して行われるので、ある場合にはこれがパフォーマンスの低下を招く結果になります。

ストアード・プロシージャーを使用すると、クライアント・アプリケーションは、データベース・サーバー上のストアード・プロシージャーに制御を渡すことができます。このようにすると、ストアード・プロシージャーはネットワークを介して不必要なデータ伝送を行わずに データベース・サーバー上で中間処理を実行することが可能になります。クライアントで実際に必要とされるレコードだけが伝送されることになります。この結果、ネットワーク通信量は低減し、全体のパフォーマンスは向上します。 図4 ではこの機能が示されています。

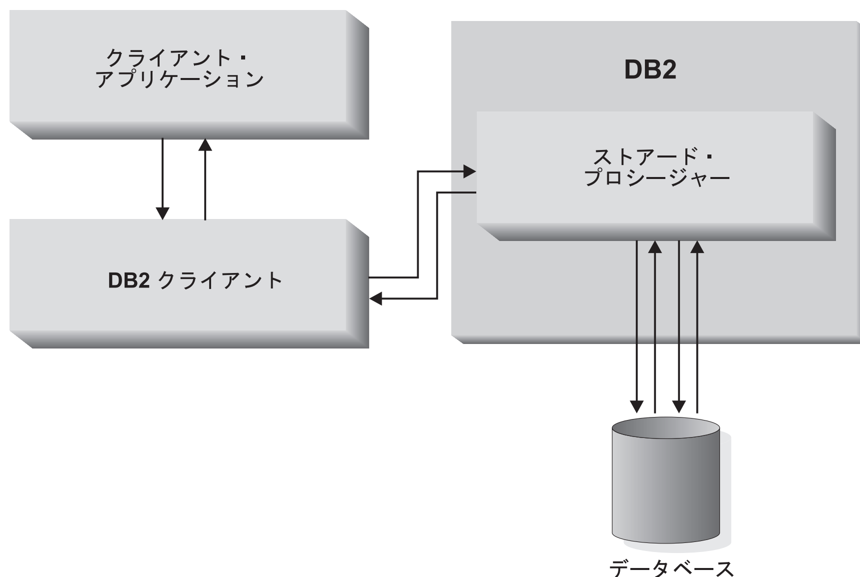


図 4. ストアード・プロシージャーを使用するアプリケーション

アプリケーションでストアード・プロシージャーを使用すると、以下のような利点があります。

- ネットワーク通信量の低減。

  ストアード・プロシージャーを使用して大量のデータを処理するために適切に設計されたアプリケーションは、クライアントが必要とするデータだけを戻します。このため、ネットワークを介して伝送されるデータの量は低減されます。

- サーバー集中作業のパフォーマンスの向上。

  グループ化される SQL ステートメントの数が多いほど、ネットワーク通信量を減らすことができます。典型的なアプリケーションでは、各 SQL ステートメントごとに、ネットワークを通る 2 回の通信が必要ですが、ストアード・プロシージャーの技法を使用するアプリケーションでは、 SQL ステートメントのグループ ごとにこれが必要となります。これによって通信の回数が減り、各通信に関連したオーバーヘッドが削減されます。

- データベース・サーバー上にのみ存在する機能へのアクセス。以下のことが含まれます。
  - サーバー上にディレクトリーをリストするコマンド (LIST DATABASE DIRECTORY および LIST NODE DIRECTORY など) を、サーバー上でのみ実行することができる。
  - サーバー・コンピューターのメモリーやディスク・スペースが大きければ、ストアード・プロシージャーはそれらの利点も活用することができる。

– データベース・サーバーにのみインストールされている付加的なソフトウェアがあれば、ストアード・プロシージャーはそれにもアクセスできる。

ストアード・プロシージャーの作成

ストアード・プロシージャーを含めたアプリケーション設計は、別個のクライアントおよびサーバー・アプリケーションから構成されます。ストアード・プロシージャー と呼ばれるサーバー・アプリケーションは、サーバー上の共用ライブラリーまたはクラス・ライブラリーに含まれています。ストアード・プロシージャーは、データベースが存在するサーバー・インスタンスでコンパイルおよびアクセスする必要があります。クライアント・アプリケーション には、ストアード・プロシージャーへの CALL ステートメントが含まれています。 CALL ステートメントにより、ストアード・プロシージャーにパラメーターを渡したり、そこからパラメーターを戻したりできます。ストアード・プロシージャーおよびクライアント・アプリケーションは、別の言語で作成することができます。クライアント・アプリケーションは、ストアード・プロシージャーとは別のプラットフォームで実行できます。

クライアント・アプリケーションは、以下の作業を実行します。

1. 任意指定のデータ構造およびホスト変数のストレージを宣言し、割り振り、そして初期化する。
2. CONNECT TO ステートメントを実行するか、または暗黙の接続を行うことによって、データベースに接続する。詳細については、*SQL 解説書* を参照してください。
3. SQL CALL ステートメントによりストアード・プロシージャーを呼び出す。
4. データベースに対し、COMMIT または ROLLBACK を発行する。

   注: ストアード・プロシージャーは COMMIT または ROLLBACK ステートメントを出すことができますが、クライアント・アプリケーションが COMMIT または ROLLBACK を出すようにすることをお勧めします。これにより、クライアント・アプリケーションがストアード・プロシージャーによって戻されるデータを評価できるようにし、トランザクションをコミットするかロールバックするかを決定できます。
5. データベースから切断する。

なお、上記のいずれのステップにおいても SQL ステートメントをコード化できます。

ストアード・プロシージャーが呼び出されると、以下の作業が実行されます。

1. クライアント・アプリケーションからパラメーターを受け取る。
2. クライアント・アプリケーションと同じトランザクションの下で、データベース・サーバー上で実行する。
3. 任意で 1 つまたは複数の COMMIT または ROLLBACK ステートメントを出す。

   注: ストアード・プロシージャーは COMMIT または ROLLBACK ステートメントを出すことができますが、クライアント・アプリケーションが COMMIT または ROLLBACK ステートメントを出すようにすることをお勧めします。これによ

り、クライアント・アプリケーションがストアード・プロシージャーによって戻されるデータを評価できるようにし、トランザクションをコミットするかロールバックするかを決定できます。

4. SQLCA 情報と任意指定の出力データを、クライアント・アプリケーションに戻す。

ストアード・プロシージャーは、クライアント・アプリケーションに呼び出されると実行されます。サーバー・プロシージャーが処理を終了すると、制御はクライアントに戻されます。複数のストアード・プロシージャーを 1 つのライブラリーに入れておくことができます。

この章では、以下のパラメーター・スタイルを使って、ストアード・プロシージャーを作成する方法を説明します。

DB2SQL
: ストアード・プロシージャーは、 CREATE PROCEDURE ステートメントで宣言したパラメーターを、クライアント・アプリケーションの CALL ステートメントからホスト変数として受け取ります。 DB2 は、 DB2SQL ストアード・プロシージャーの追加パラメーターを割り当てます。

GENERAL
: ストアード・プロシージャーは、クライアント・アプリケーションで CALL ステートメントからホスト変数としてパラメーターを受け取ります。 クライアント・アプリケーションにヌル標識を直接渡すことはしません。 GENERAL は、 DB2 ユニバーサル・データベース (OS/390 版) の SIMPLE ストアード・プロシージャーと同等です。

GENERAL WITH NULLS
: ユーザーによって宣言される各パラメーターでは、 DB2 は対応する INOUT パラメーターのヌル標識を割り当てます。 GENERAL と同様に、パラメーターはホスト変数として渡されます。 GENERAL WITH NULLS は、 DB2 ユニバーサル・データベース (OS/390 版) の SIMPLE WITH NULLS ストアード・プロシージャーと同等です。

JAVA
: ストアード・プロシージャーは、 SQLJ ルーチン仕様に準拠したパラメーター受け渡し規則を使用します。ホスト変数として IN パラメーターを、単一記入項目配列として OUT および INOUT パラメーターを受け取ります。

前述のパラメーター・スタイルについて、 CREATE PROCEDURE ステートメントを使って各ストアード・プロシージャーを登録する必要があります。 CREATE PROCEDURE ステートメントは、ストアード・プロシージャーごとに、プロシージャー名、引き数、位置、およびパラメーター・スタイルを指定します。パラメーター・スタイルを指定することにより、 DB2 ファミリー間でのストアード・プロシージャー・コードの移植性とスケーラビリティーが向上します。

DB2 ユニバーサル・データベースのバージョン 6 以前の DB2 バージョンのみでサポートされているストアード・プロシージャーのパラメーター・スタイル (DB2DARI お

よび DB2GENERAL) を使用する場合、 789ページの『付録C. DB2DARI および DB2GENERAL ストアード・プロシージャーと UDF』を参照してください。

クライアント・アプリケーション

クライアント・アプリケーションは、ストアード・プロシージャーを呼び出す前にいくつかのステップを実行します。まずデータベースに接続し、 SQLDA 構造またはホスト変数の宣言、割り振り、初期化を行う必要があります。 SQL CALL ステートメントは、一連のホスト変数または SQLDA 構造を受け入れることができます。 SQL CALL ステートメントおよび SQLDA 構造の詳細については、 *SQL 解説書* を参照してください。クライアント・アプリケーションでの SQLDA 構造の使用については、 789ページの『付録C. DB2DARI および DB2GENERAL ストアード・プロシージャーと UDF』を参照してください。

ホスト変数の割り振り

必要な入力ホスト変数をストアード・プロシージャーのクライアント側で割り振るための手順を以下に示します。

1. ストアード・プロシージャーに渡されるすべての入力変数に見合った十分な数のホスト変数を宣言する。
2. ストアード・プロシージャーからクライアントに値を戻すためにも使用できる入力ホスト変数を決める。
3. ストアード・プロシージャーからクライアントに戻される追加の値のためのホスト変数を宣言する。

ストアード・プロシージャーのクライアント部分を作成する際には、ホスト変数を入出力の両方に使用することによって、可能なかぎり多くのホスト変数を多重定義するようにします。これにより、複数のホスト変数を処理するための効率が高くなります。たとえば、SQLCODE をクライアントからストアード・プロシージャーに戻す場合には、 INTEGER として宣言された入力ホスト変数を使用して SQLCODE を戻すようにしてください。

注: これらの構造のストレージは、データベース・サーバー上に割り振らないでください。データベース・マネージャーは、クライアント・アプリケーションが割り振ったストレージに基づいて自動的に重複ストレージを割り振ります。ストアード・プロシージャー側の入出力パラメーターに対するストレージ・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に作成されたストレージ・ポインターと置き換えようとすると、 SQLCODE -1133 (SQLSTATE 39502) のエラーが発生します。

ストアード・プロシージャーの呼び出し

データベースのロケーションに保管されているストアード・プロシージャーは、 SQL の CALL ステートメントを使用して呼び出すことができます。 CALL ステートメントの詳細な説明については、*SQL 解説書* を参照してください。ストアード・プロシージャーを呼び出す場合、CALL ステートメントを使用することをお勧めします。

クライアント・アプリケーションの実行

クライアント・アプリケーションは、ストアード・プロシージャーを呼び出す前に、データベース接続が行われたかどうかを確認しなければなりません。そうしないと、エラーが戻されます。データベース接続およびデータ構造の初期化の後、クライアント・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し、要求されるデータをすべて渡します。アプリケーションはデータベースから切り離されます。なお、上記のいずれのステップにおいても SQL ステートメントをコード化できます。

サーバー上でのストアード・プロシージャー

ストアード・プロシージャーは、SQL CALL ステートメントにより呼び出され、クライアント・アプリケーションによって渡されたデータを使用して実行されます。データベース・マネージャーのストアード・プロシージャーを CREATE PROCEDURE ステートメントに登録する際のパラメーター・スタイルによって、ストアード・プロシージャーがクライアント・アプリケーションからデータを受け取る方法が決まります。

ストアード・プロシージャーの登録

CREATE PROCEDURE ステートメントを使用するには、以下を宣言しなければなりません。

- プロシージャー名
- 各パラメーターのモード、名前、および SQL データ・タイプ
- 外部名と位置
- パラメーター・スタイル

CREATE PROCEDURE では以下も宣言しなければなりません。

- ストアード・プロシージャーを FENCED と NOT FENCED のどちらで実行するか
- プロシージャー本体に含まれる SQL ステートメントのタイプ (もしあれば)

CREATE PROCEDURE ステートメントについては、 *SQL 解説書* の中からさらに多くの情報を得ることができます。そこでは、DB2 ファミリーとの互換性を考慮した完全な構文やオプションを記載しています。 CREATE PROCEDURE ステートメントの一般的な使用法は以下のとおりです。

プロシージャー名: 固有の数のパラメーターを受け入れるプロシージャーに同じ名前を使用することによってのみ、ストアード・プロシージャーを多重定義することができます。 DB2 はデータ・タイプを区別しないため、パラメーターのデータ・タイプに基づいてストアード・プロシージャーを多重定義することはできません。

たとえば、次の 2 つの CREATE PROCEDURE ステートメントを発行すると、それぞれ 1 つのパラメーターと 2 つのパラメーターを受け入れるため、有効です。

```
CREATE PROCEDURE OVERLOAD (IN VAR1 INTEGER) ...
CREATE PROCEDURE OVERLOAD (IN VAR1 INTEGER, IN VAR2 INTEGER) ...
```

一方、次の例では 2 番目のストアード・プロシージャーは登録されません。なぜなら、同じ名前の最初のストアード・プロシージャーとパラメーターの数が同じだからです。

```
CREATE PROCEDURE OVERLOADFAIL (IN VAR1 INTEGER) ...
CREATE PROCEDURE OVERLOADFAIL (IN VAR2 VARCHAR(15)) ...
```

パラメーター・モード: *明示パラメーター* は、 CREATE PROCEDURE ステートメントのパラメーター・リストで明示的に宣言するパラメーターです。*暗黙パラメーター* は、 DB2 によって自動的に提供されるパラメーターです。たとえば、 PARAMETER STYLE GENERAL WITH NULLS ストアード・プロシージャーは、明示パラメーターのヌル標識の配列を自動的に提供します。ストアード・プロシージャーを作成する際には、ストアード・プロシージャーの明示パラメーターおよび暗黙パラメーターの両方を考慮する必要があります。クライアント・アプリケーションを作成する際には、ストアード・プロシージャーの明示パラメーターを処理するだけで済みます。明示パラメーターはすべて IN、OUT、または INOUT パラメーターのいずれかとして宣言する必要があり、名前と SQL データ・タイプを一緒に宣言する必要があります。 CREATE PROCEDURE ステートメントの例については、 222ページの『変数宣言と CREATE PROCEDURE の例』を参照してください。

IN
: クライアント・アプリケーションからストアード・プロシージャーに値を渡しますが、制御がクライアント・アプリケーションに戻る際にはクライアント・アプリケーションに値を戻しません。

OUT
: ストアード・プロシージャーの終了時に、クライアント・アプリケーションに渡された値を保管します。

INOUT
: クライアント・アプリケーションからストアード・プロシージャーに値を渡し、ストアード・プロシージャーの終了時にクライアント・アプリケーションに値を戻します。

位置: CREATE PROCEDURE ステートメントの EXTERNAL 文節により、データベース・マネージャーはストアード・プロシージャーを含んでいるライブラリーの位置を認識します。ライブラリーの絶対パス (Java ストアード・プロシージャーの場合は jar 名) を指定しない場合、データベース・マネージャーは*関数ディレクトリー* を検索します。*関数ディレクトリー* とは、オペレーティング・システムに定義されたディレクトリーで、以下のようなものです。

UNIX オペレーティング・システム
: `sqllib/function`

OS/2 または Windows 32 ビット・オペレーティング・システム
: *instance_name*¥function。ここで、*instance_name* は *DB2INSTPROF* インスタンス固有のレジストリー設定の値を表します。 *DB2INSTPROF* が設定されていない場合、 *instance_name* は *%DB2PATH%* 環境変数の値を表します。 *%DB2PATH%* 環境変数のデフォルト値は、 DB2 のインストール先のパスです。

instance_name¥function にストアード・プロシージャーが見つからない場合、DB2 は *PATH* および *LIBPATH* 環境変数で定義されたディレクトリーを検索します。

たとえば、 DB2 が C:¥sqllib ディレクトリーにインストールされており、*DB2INSTPROF* レジストリーが設定されていない Windows 32 ビットの関数ディレクトリーは、次のようになります。

```
C:¥sqllib¥function
```

注: ライブラリー名にはストアード・プロシージャー名とは異なる名前を指定してください。 DB2 は、検索パスで同じ名前のライブラリーを見つけると、そのライブラリーと同じ名前のストアード・プロシージャーを FENCED DB2DARI プロシージャーとして実行します。

C 言語のストアード・プロシージャーの場合は、以下のように指定してください。

- ライブラリー名。形式は以下のいずれか。
  - 関数ディレクトリーにあるライブラリー
  - ライブラリー名を含む絶対パス
- ライブラリー内のストアード・プロシージャー用のエントリー・ポイント。エントリー・ポイントを指定しないと、データベース・マネージャーはデフォルトのエントリー・ポイントを使用します。 AIX 上の IBM XL C コンパイラーを使用すると、ライブラリー内にあるエクスポート済みの任意の関数名をデフォルトのエントリー・ポイントとして指定できます。 これは、ストアード・プロシージャー呼び出しまたは CREATE FUNCTION ステートメントにライブラリー名しか指定されていない場合に呼び出される関数です。デフォルトのエントリー・ポイントを指定するには、リンクの段階で `-e` オプションを使用します。たとえば、`-e` *funcname* とすると、 *funcname* がデフォルトのエントリー・ポイントになります。その他の UNIX プラットフォームでは、このようなメカニズムはありません。そのため、デフォルトのエントリー・ポイントは DB2 によってライブラリー自体と同じ名前になるように想定されます。

UNIX ベースのシステムでは、たとえば `mymod!proc8` と指定すると、データベース・マネージャーは `sqllib/function/mymod` ライブラリーを指し、そのライブラリー内のエントリー・ポイント `proc8` を使用します。 Windows 32 ビットおよび OS/2 オペレーティング・システムでは、 `mymod!proc8` と指定すると、データベース・マネージャーは関数ディレクトリーから `mymod.dll` ファイルをロードし、ダイナミック・リンク・ライブラリー (DLL) 内の `proc8()` プロシージャーを呼び出します。

LANGUAGE JAVA ストアード・プロシージャーの場合は、以下の構文を使用してください。

```
[<jar-file-name>:]<class-name>.<method-name> (java-method-signature)
```

次のリストは Java ストアード・プロシージャーの EXTERNAL キーワードを定義します。

jar-file-name

データベースにインストールされた jar ファイルにストアード・プロシージャー・メソッドが含まれている場合、この値を含める必要があります。このキーワードは jar ファイルの名前を表しており、コロン (:) によって区切られています。 jar ファイル名を指定しないと、データベース・マネージャーは関数ディレクトリーでクラスを探します。 jar ファイルのインストールの詳細については、 689ページの『Java ストアード・プロシージャーおよび UDF』を参照してください。

class-name

ストアード・プロシージャー・メソッドを含んだクラスの名前。クラスがパッケージの一部になっている場合は、完全なパッケージ名を接頭部として含める必要があります。

method-name

ストアード・プロシージャー・メソッドの名前

java-method-signature

メソッド用の Java パラメーター・データ・タイプのリスト。これらのデータ・タイプは、プロシージャーまたは関数名の後に指定されるシグニチャーのデフォルト Java タイプ・マッピングに対応している必要があります。たとえば、SQL タイプが INTEGER のデフォルトの Java マッピングは `int` で、`java.lang.Integer` ではありません。デフォルトの Java タイプのマッピングのリストについては、658ページの表32 を参照してください。

たとえば、`MyPackage.MyClass.myMethod` を指定する場合、データベース・マネージャーは、 `MyPackage` パッケージ内の `MyClass` クラスにある `myMethod` メソッドを使用します。ここで、コロン (:) 区切り文字の代わりに、ピリオド (.) 区切り文字が使用されているため、 DB2 は `MyPackage` が jar ファイルではなくてパッケージであることを認識します。 DB2 は、関数ディレクトリーで `MyPackage` パッケージを検索します。

関数ディレクトリーの詳細については、 209ページの『位置』を参照してください。

言語: C/C++ の場合は、CREATE PROCEDURE ステートメントで LANGUAGE C を宣言します。 Java ストアード・プロシージャーの場合は、LANGUAGE JAVA を宣言します。 Windows 32 ビット・オペレーティング・システム上で OLE ストアード・プロシージャーを使用する場合には、 LANGUAGE OLE を宣言します。 COBOL ストアード・プロシージャーの場合は、LANGUAGE COBOL を宣言します。 Fortran または REXX ストアード・プロシージャーの場合は、 DB2DARI ストアード・プロシージャーとして作成する必要があります。 DB2DARI ストアード・プロシージャーの作成の詳細については、789ページの『付録C. DB2DARI および DB2GENERAL ストアード・プロシージャーと UDF』を参照してください。

LANGUAGE C

データベース・マネージャーは ANSI C の呼び出しおよび連係規則を使って、

ストアード・プロシージャーを呼び出します。このオプションは大部分の C/C++ ストアード・プロシージャーで使用してください。

LANGUAGE JAVA
データベース・マネージャーは Java クラス内のメソッドとしてストアード・プロシージャーを呼び出します。このオプションはすべての Java ストアード・プロシージャーで使用してください。

LANGUAGE OLE
データベース・マネージャーは OLE 関数としてストアード・プロシージャーを呼び出します。 Windows の 32 ビット・オペレーティング・システムの OLE ストアード・プロシージャーでは、このオプションを使用してください。 CREATE PROCEDURE ステートメントを出す前に、 REGSVR32 コマンドを使用して OLE ストアード・プロシージャーを含んだ DLL を登録する必要があります。 OLE ストアード・プロシージャーは、FENCED モードで実行する必要があります。 OLE ストアード・プロシージャーの使用についての詳細は、 *アプリケーション構築の手引き* を参照してください。

LANGUAGE COBOL
データベース・マネージャーは COBOL の呼び出しおよび連係規則を使って、ストアード・プロシージャーを呼び出します。このオプションは COBOL ストアード・プロシージャーで使用してください。

パラメーターをサブルーチンとして渡す: PROGRAM TYPE SUB の C ストアード・プロシージャーは、引き数をサブルーチンとして受け入れます。数値データ・タイプ・パラメーターをポインターとして受け渡します。文字データ・タイプを適切な長さの配列として渡します。たとえば、次のような C ストアード・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, short *arg2, char arg[4])
```

Java のストアード・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。たとえば、次のような Java のストアード・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[])
```

パラメーターを main 関数として渡す: 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ストアード・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 のストアード・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。

- DB2 は、パラメーター配列の最初のエレメントの値をストアード・プロシージャーの名前に設定する。
- ストアード・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*。
  - パラメーターを含む配列。たとえば、*argv[]*
- ストアード・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 ストアード・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]*) をストアード・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、ストアード・プロシージャーの CREATE PROCEDURE ステートメントで宣言されるパラメーターと対応します。たとえば、次のような組み込み型の C ストアード・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'
```

次に示すストアード・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。

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

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

  /* argv[0] contains the procedure name, so 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 */
```

PARAMETER STYLE: 表9 では、DB2 バージョン 7 の CREATE PROCEDURE ステートメントで使用できる、パラメーター・スタイル (横軸) と言語 (縦軸) の組み合わせを要約しています。

表 9. CREATE PROCEDURE: パラメーター・スタイルと言語の有効な組み合わせ

	GENERAL, GENERAL WITH NULLS	JAVA	DB2SQL	DB2DARI	DB2GENERAL
LANGUAGE C	Y	N	Y	Y	N
LANGUAGE JAVA	N	Y	N	N	Y
LANGUAGE OLE	N	N	Y	N	N
LANGUAGE COBOL	Y	N	Y	N	N

GENERAL

ストアード・プロシージャーは、クライアント・アプリケーションで CALL ステートメントからホスト変数としてパラメーターを受け取ります。 クライアント・アプリケーションにヌル標識を直接渡すことはしません。 GENERAL は、LANGUAGE C または LANGUAGE COBOL オプションも指定している場合に限り、使用できます。

DB2 ユニバーサル・データベース (OS/390 版) の互換性の注意: GENERAL は SIMPLE と同等です。

PARAMETER STYLE GENERAL ストアード・プロシージャーは、PROGRAM TYPE 文節の値で指示されている方法でパラメーターを受け入れます。次の例では、 PROGRAM TYPE SUBROUTINE を使用して 2 つのパラメーターを受け入れる PARAMETER STYLE GENERAL ストアード・プロシージャーを示します。

```
SQL_API_RC SQL_API_FN one_result_set_to_client
   (double *insalary, sqlint32 *out_sqlerror)
{
  EXEC SQL INCLUDE SQLCA;

  EXEC SQL WHENEVER SQLERROR GOTO return_error;

  EXEC SQL BEGIN DECLARE SECTION;
```

```
    double l_insalary;
  EXEC SQL END DECLARE SECTION;

  l_insalary = *insalary;
  *out_sqlerror = 0;

  EXEC SQL DECLARE c3 CURSOR FOR
    SELECT name, job, CAST(salary AS INTEGER)
    FROM staff
    WHERE salary > :l_insalary
    ORDER BY salary;

  EXEC SQL OPEN c3;
  /* Leave cursor open to return result set */

  return (0);

  /* Copy SQLCODE to OUT parameter if SQL error occurs */
  return_error:
  {
    *out_sqlerror = SQLCODE;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    return (0);
  }

} /* end one_result_set_to_client function */
```

GENERAL WITH NULLS

ユーザーによって宣言される各パラメーターでは、 DB2 は対応する INOUT パラメーターのヌル標識を割り当てます。 GENERAL と同様に、パラメーターはホスト変数として渡されます。 GENERAL WITH NULLS は、LANGUAGE C または LANGUAGE COBOL オプションも指定している場合に限り、使用できます。

DB2 ユニバーサル・データベース (OS/390 版) の互換性の注意: GENERAL WITH NULLS は、SIMPLE WITH NULLS と同様のものです。

PARAMETER STYLE GENERAL WITH NULLS というストアード・プロシージャーは、 PROGRAM TYPE 文節の値で指示されている方法でパラメーターを受け入れ、ヌル標識の配列で宣言されている各パラメーターに対して 1 つのエレメントを割り当てます。次の SQL は、 PROGRAM TYPE SUB を使用して、 INOUT パラメーター 1 つと OUT パラメーター 2 つを渡す PARAMETER STYLE GENERAL WITH NULLS ストアード・プロシージャーを登録します。

```
CREATE PROCEDURE INOUT_PARAM (INOUT medianSalary DOUBLE,
   OUT errorCode INTEGER, OUT errorLabel CHAR(32))
   DYNAMIC RESULT SETS 0
   LANGUAGE C
   PARAMETER STYLE GENERAL WITH NULLS
   NO DBINFO
   FENCED
```

```
    MODIFIES SQL DATA
    PROGRAM TYPE SUB
    EXTERNAL NAME 'spserver!inout_param'
```

次の C コードは、 GENERAL WITH NULLS ストアード・プロシージャーによって必要とされるヌル標識を宣言および使用する方法を示します。

```
SQL_API_RC SQL_API_FN inout_param (double *inoutMedian,
  sqlint32 *out_sqlerror, char buffer[33], sqlint16 nullinds[3])
{
  EXEC SQL INCLUDE SQLCA;

  EXEC SQL WHENEVER SQLERROR GOTO return_error;

  if (nullinds[0] < 0)
  {
    /* NULL value was received as input, so return NULL output */
    nullinds[0] = -1;
    nullinds[1] = -1;
    nullinds[2] = -1;
  }
  else
  {
    int counter = 0;
    *out_sqlerror = 0;
    medianSalary = *inoutMedian;

    strcpy(buffer, "DECLARE inout CURSOR");
    EXEC SQL DECLARE inout CURSOR FOR
      SELECT CAST(salary AS DOUBLE) FROM staff
      WHERE salary > :medianSalary
      ORDER BY salary;

    nullinds[1] = 0;
    nullinds[2] = 0;

    strcpy(buffer, "SELECT COUNT INTO numRecords");
    EXEC SQL SELECT COUNT(*) INTO :numRecords
      FROM staff
      WHERE salary > :medianSalary;

    if (numRecords != 0)
    /* At least one record was found */
    {
      strcpy(buffer, "OPEN inout");
      EXEC SQL OPEN inout USING :medianSalary;

      strcpy(buffer, "FETCH inout");
      while (counter < (numRecords / 2 + 1)) {
        EXEC SQL FETCH inout INTO :medianSalary;

        *inoutMedian = medianSalary;
        counter = counter + 1;
      }
```

```
        strcpy(buffer, "CLOSE inout");
        EXEC SQL CLOSE inout;
      }
      else /* No records were found */
      {
        /* Return 100 to indicate NOT FOUND error */
        *out_sqlerror = 100;
      }
    }

    return (0);

    /* Copy SQLCODE to OUT parameter if SQL error occurs */
    return_error:
    {
      *out_sqlerror = SQLCODE;
      EXEC SQL WHENEVER SQLERROR CONTINUE;
      return (0);
    }

  } /* end inout_param function */
```

JAVA ストアード・プロシージャーは、 *SQLJ* ルーチン 仕様に準拠したパラメーターを渡す場合の規則を使用します。ホスト変数として IN パラメーターを、単一記入項目配列として OUT および INOUT パラメーターを受け取ります。JAVA は、LANGUAGE JAVA オプションも指定している場合に限り、使用できます。

DB2SQL

DB2SQL ストアード・プロシージャーの C 関数定義は、次のような暗黙パラメーターを、 CREATE PROCEDURE ステートメントで宣言されるパラメーターの定義に追加する必要があります。

```
sqlint16 nullinds[n],   1
char sqlst[6],          2
char qualname[28],      3
char specname[19],      4
char diagmsg[71],       5
```

DB2 はストアード・プロシージャーに以下の引き数を渡します。

1. DB2 は、暗黙 SMALLINT INOUT パラメーターの配列を、明示パラメーターのヌル標識として割り当てます。この配列は、*n* サイズです。ここで、*n* は明示パラメーターの数を表します。
2. SQLSTATE 値の暗黙 CHAR(5) OUT パラメーター。
3. 修飾されたストアード・プロシージャー名の暗黙 CHAR(27) IN パラメーター。
4. ストアード・プロシージャーの固有名の暗黙 CHAR(18) IN パラメーター。
5. SQL 診断ストリングの暗黙 CHAR(70) OUT パラメーター。

DB2SQL は、LANGUAGE C または LANGUAGE COBOL オプションも指定している場合に限り、指定できます。たとえば、次の CREATE PROCEDURE ステートメントは、 PARAMETER STYLE DB2SQL ストアード・プロシージャーを登録します。

```
CREATE PROCEDURE DB2SQL_EXAMPLE (IN job CHAR(8), OUT salary DOUBLE)
   DYNAMIC RESULT SETS 0
   LANGUAGE C
   PARAMETER STYLE DB2SQL
   NO DBINFO
   FENCED
   READS SQL DATA
   PROGRAM TYPE SUB
   EXTERNAL NAME 'spserver!db2sqlexample'
```

次の規則を使用してストアード・プロシージャーを作成します。

- PARAMETER STYLE DB2SQL ストアード・プロシージャーは、ヌル標識の配列を渡します (明示パラメーターごとに 1 つのエレメントを渡す)。 IN または INOUT パラメーターのヌル標識エレメントが負の値である場合は、クライアント・アプリケーションがそのパラメーターでヌル値を渡したことを示します。出力パラメーターが NULL でないことを示すには、 OUT または INOUT パラメーターのヌル標識エレメントの値を 0 に設定します。出力パラメーターが NULL であることを示すには、 OUT または INOUT パラメーターのヌル標識エレメントの値を -1 に設定します。
- 前述のとおり、 DB2SQL パラメーターのストアード・プロシージャー・シグニチャーに引き数を追加します。
- DB2SQL SQLSTATE (CHAR(5)) および診断メッセージ (ヌル終了 CHAR(70)) パラメーターの値を設定して、 SQLCA のカスタマイズされた値をクライアントに戻すことができます。

たとえば、次の組み込み型の C ストアード・プロシージャーは、 PARAMETER STYLE DB2SQL ストアード・プロシージャーのコーディングを示します。

```
SQL_API_RC SQL_API_FN db2sql_example (
  char injob[9],         /* Input - CHAR(8)   */
  double *salary,        /* Output - DOUBLE   */
  sqlint16 nullinds[2],
  char sqlst[6],
  char qualname[28],
  char specname[19],
  char diagmsg[71]
)
{
  EXEC SQL INCLUDE SQLCA;

  if (nullinds[0] < 0)
  {
    /* NULL value was received as input, so return NULL output */
```

```
        nullinds[1] = -1;
        /* Set custom SQLSTATE to return to client. */
        strcpy(sqlst, "38100");
        /* Set custom message to return to client. */
        strcpy(diagmsg, "Received null input on call to DB2SQL_EXAMPLE.");

      }
      else
      {
        EXEC SQL SELECT (CAST(AVG(salary) AS DOUBLE))
          INTO :outsalary INDICATOR :outsalaryind
          FROM employee
          WHERE job = :injob;

        *salary = outsalary;
        nullinds[1] = outsalaryind;
      }
      return (0);
    } /* end db2sql_example function */
```

次の組み込み型 C クライアント・アプリケーションは、 DB2SQL_EXAMPLE ストアード・プロシージャーを呼び出す CALL ステートメントを出します。この例では、CALL ステートメントの各パラメーターに対してヌル標識が含まれていることに注目してください。この例では *in_jobind* ヌル標識が 0 に設定されて、非 NULL 値が IN パラメーター (*in_job* というホスト変数で表される) のストアード・プロシージャーに渡されていることが示されています。OUT パラメーターのヌル標識が -1 に設定されて、これらのパラメーターのストアード・プロシージャーには入力が渡されていないことが示されます。

```
int db2sqlparm(char out_lang[9], char job_name[9])
{
   int testlang;

   EXEC SQL BEGIN DECLARE SECTION;
     /* Declare host variables for passing data to DB2SQL_EXAMPLE */
     char in_job[9];
     sqlint16 in_jobind;
     double out_salary = 0;
     sqlint16 out_salaryind;
   EXEC SQL END DECLARE SECTION;

   /*****************************************************¥
   * Call DB2SQL_EXAMPLE stored procedure                 *
   ¥*****************************************************/

   testlang = strncmp(out_lang, "C", 1);
   if (testlang != 0) {
      /* Only LANGUAGE C procedures can be PARAMETER STYLE DB2SQL,
         so do not call the DB2SQL_EXAMPLE stored procedure */
      printf("¥nStored procedures are not implemented in C.¥n"
             "Skipping the call to DB2SQL_EXAMPLE.¥n");
   }
   else {
```

```
    strcpy(procname, "DB2SQL_EXAMPLE");
    printf("¥nCALL stored procedure named %s¥n", procname);

    /* out_salary is an OUT parameter, so set the
       null indicator to -1 to indicate no input value */
    out_salaryind = -1;

    strcpy(in_job, job_name);

    /* in_job is an IN parameter, so check to
        see if there is any input value */
    if (strlen(in_job) == 0)
    {
       /* in_job is null, so set the null indicator
          to -1 to indicate there is no input value */
       in_jobind = -1;
       printf("with NULL input, to return a custom
               SQLSTATE and diagnostic message¥n");
    }
    else
    {
       /* in_job is not null, so set the null indicator
          to 0 to indicate there is an input value */
       in_jobind = 0;
    }

    /* DB2SQL_EXAMPLE is PS DB2SQL, so pass
       a null indicator for each parameter */
    EXEC SQL CALL :procname (:in_job:in_jobind,
        :out_salary:out_salaryind);

    /* DB2SQL stored procedures can return a custom
       SQLSTATE and diagnostic message, so instead of
       using the EMB_SQL_CHECK macro to check the value
       of the returned SQLCODE, check the SQLCA structure for
       the value of the SQLSTATE and the diagnostic message */

    /* Check value of returned SQLSTATE */
    if (strncmp(sqlca.sqlstate, "00000", 5) == 0) {
      printf("Stored procedure returned successfully.¥n");
      printf("Average salary for job %s = %9.2f¥n",
        in_job, out_salary);
    }
    else {
       printf("Stored procedure failed with SQLSTATE %s.¥n",
         sqlca.sqlstate);
       printf("Stored procedure returned the following
         diagnostic message:¥n");
       printf("  ¥"%s¥"¥n", sqlca.sqlerrmc);
    }
 }

return 0;
}
```

DB2GENERAL

ストアード・プロシージャーは、 DB2 Java ストアード・プロシージャーだけにサポートされているパラメーターを渡す場合の規則を使用します。 DB2GENERAL は、LANGUAGE JAVA オプションも指定している場合に限り、使用できます。

移植性を向上させるために、Java ストアード・プロシージャーは、PARAMETER STYLE JAVA 規則を使って作成してください。 DB2GENERAL パラメーター・スタイルのストアード・プロシージャーの作成については、789ページの『付録C. DB2DARI および DB2GENERAL ストアード・プロシージャーと UDF』を参照してください。

DB2DARI

ストアード・プロシージャーは、C 言語の呼び出しおよび連係規則に準拠したパラメーター受け渡し規則を使用します。このオプションは DB2 ユニバーサル・データベースにのみサポートされており、 LANGUAGE C オプションも指定している場合に限り、使用できます。

DB2 ファミリー間での移植性を向上させるために、 LANGUAGE C ストアード・プロシージャーは GENERAL または GENERAL WITH NULLS パラメーター・スタイルを使って作成してください。 DB2DARI パラメーター・スタイルのストアード・プロシージャーを作成する場合は、 789ページの『付録C. DB2DARI および DB2GENERAL ストアード・プロシージャーと UDF』を参照してください。

DBINFO 構造の受け渡し: LANGUAGE C ストアード・プロシージャーで、パラメーター・タイプに GENERAL、 GENERAL WITH NULLS、または DB2SQL を指定している場合、追加のパラメーターを受け入れるようにストアード・プロシージャーを作成することもできます。 CREATE PROCEDURE ステートメントで DBINFO を指定すると、呼び出しパラメーターとともに、 DB2 クライアントに関する情報を含む DBINFO 構造をストアード・プロシージャーに渡すように、クライアント・アプリケーションに指示することができます。 DBINFO 構造には以下の値が含まれます。

データベース名

クライアントが接続されているデータベースの名前。

アプリケーション許可 ID

アプリケーションの実行時許可 ID。

コード・ページ

データベースのコード・ページ。

スキーマ名

ストアード・プロシージャーには該当しません。

表名　ストアード・プロシージャーには該当しません。

列名　ストアード・プロシージャーには該当しません。

データベースのバージョンとリリース
: ストアード・プロシージャーを呼び出すデータベース・サーバーのバージョン、リリース、および修正レベル。

プラットフォーム
: データベース・サーバーのプラットフォーム。

表関数の結果列の数
: ストアード・プロシージャーには該当しません。

DBINFO 構造の詳細については、420ページの『DBINFO 構造』を参照してください。

変数宣言と CREATE PROCEDURE の例

以下の例では、SAMPLE データベースを用いた仮定的なシナリオで使用する、ストアード・プロシージャーのソース・コードと CREATE PROCEDURE ステートメントを具体的に示します。

IN および OUT パラメーターの使用: Java ストアード・プロシージャー `GET_LASTNAME` を作成するとします。これには *empno* (SQL タイプ VARCHAR) が与えられており、 SAMPLE データベースの EMPLOYEE 表から *lastname* (SQL タイプ CHAR) を戻します。 Java クラス `StoredProcedure` の `getname` メソッドとして、プロシージャーを作成します。このクラスは `myJar` という名前でインストールされた JAR ファイルに含まれています。最後に、C でコード化されたクライアント・アプリケーションで、ストアード・プロシージャーを呼び出します。

1. ストアード・プロシージャーのソース・コードに、以下の 2 つのホスト変数を宣言します。

   ```
   String empid;
   String name;
   ...
   #sql { SELECT lastname INTO :empid FROM employee WHERE empno=:empid }
   ```

2. ストアード・プロシージャーを次の CREATE PROCEDURE ステートメントで登録します。

   ```
   CREATE PROCEDURE GET_LASTNAME (IN EMPID CHAR(6), OUT NAME VARCHAR(15))
       EXTERNAL NAME 'myJar:StoredProcedure.getname'
       LANGUAGE JAVA PARAMETER STYLE JAVA FENCED
       READS SQL DATA
   ```

3. C で作成されたクライアント・アプリケーションからストアード・プロシージャーを呼び出します。

   ```
   EXEC SQL BEGIN DECLARE SECTION;
      struct name { short int; char[15] }
      char[7] empid;
   EXEC SQL END DECLARE SECTION;
      ...
   EXEC SQL CALL GET_LASTNAME (:empid, :name);
   ```

INOUT パラメーターの使用: 次の例では、C ストアード・プロシージャー *GET_MANAGER* を作成するとします。これには *deptnumb* (SQL タイプ SMALLINT) が与えられており、 SAMPLE データベースの ORG 表から *manager* (SQL タイプ SMALLINT) を戻します。

1. *deptnumb* および *manager* の SQL データ・タイプはどちらも SMALLINT であるため、ストアード・プロシージャーでは 1 つの変数 *onevar* で定義することができます。そして、クライアント・アプリケーションとの間で値をやり取りします。

```
EXEC SQL BEGIN DECLARE SECTION;
  short onevar = 0;
EXEC SQL END DECLARE SECTION;
```

2. ストアード・プロシージャーを次の CREATE PROCEDURE ステートメントで登録します。

```
CREATE PROCEDURE GET_MANAGER (INOUT onevar SMALLINT)
   EXTERNAL NAME 'stplib!getman'
   LANGUAGE C PARAMETER STYLE GENERAL FENCED
   READS SQL DATA
```

3. Java で作成されたクライアント・アプリケーションからストアード・プロシージャーを呼び出します。

```
short onevar = 0;
  ...
#SQL { CALL GET_MANAGER (:INOUT onevar) };
```

ストアード・プロシージャーの SQL ステートメント

ストアード・プロシージャーには SQL ステートメントを含めることができます。 CREATE PROCEDURE ステートメントを発行する際に、ストアード・プロシージャーに含まれている SQL ステートメントがあれば、そのタイプを指定する必要があります。ストアード・プロシージャーを登録する際に値を指定しない場合、データベース・マネージャーは MODIFIES SQL DATA を使用します。ストアード・プロシージャーで使用されている SQL のタイプを制限するには、以下の 4 つのオプションのいずれかを使用できます。

NO SQL

ストアード・プロシージャーが SQL ステートメントを実行できないことを示す。ストアード・プロシージャーで SQL ステートメントを実行しようとすると、ステートメントは SQLSTATE 38001 を戻します。

CONTAINS SQL

SQL データの読み取りおよび変更ができない SQL ステートメントがストアード・プロシージャーでは実行できないことを示します。 SQL データを読み取ったり、変更したりする SQL ステートメントをストアード・プロシージャーで実行しようとすると、ステートメントは SQLSTATE 38004 を戻します。ストアード・プロシージャーでサポートされていないステートメントは SQLSTATE 38003 を戻します。

READS SQL DATA
SQL データを変更しない SQL ステートメントのいくつかは、ストアード・プロシージャーで実行できることを示します。データを変更する SQL ステートメントをストアード・プロシージャーで実行しようとすると、ステートメントは SQLSTATE 38002 を戻します。ストアード・プロシージャーでサポートされていないステートメントは SQLSTATE 38003 を戻します。

MODIFIES SQL DATA
ストアード・プロシージャーによってサポートされていないステートメントを除けば、どんな SQL ステートメントでもストアード・プロシージャーで実行できることを示します。ストアード・プロシージャーでサポートされていない SQL ステートメントをストアード・プロシージャーが実行しようとすると、ステートメントは SQLSTATE 38003 を戻します。

CREATE PROCEDURE ステートメントの詳細については、 *SQL 解説書* を参照してください。

ネストされたストアード・プロシージャー

ネストされた ストアード・プロシージャーとは、別のストアード・プロシージャーを呼び出すものを指します。 DB2 アプリケーションでは、この技法の使用に関して次のような制約事項があります。

- ストアード・プロシージャーは LANGUAGE C または LANGUAGE SQL としてカタログ化されなければなりません。
- 呼び出し元のストアード・プロシージャーは、同じ LANGUAGE 文節を使用してカタログ化されたストアード・プロシージャーしか呼び出せません。ネストされた呼び出しに関してだけ、 LANGUAGE C と LANGUAGE SQL が同一言語と見なされます。たとえば、 LANGUAGE C ストアード・プロシージャーは、 SQL プロシージャーを呼び出すことができます。
- 呼び出し元のストアード・プロシージャーは、より高い SQL データ・アクセス・レベルでカタログ化されたストアード・プロシージャーを呼び出すことができません。たとえば、 CONTAINS SQL データ・アクセスでカタログ化されたストアード・プロシージャーは、 NO SQL または CONTAINS SQL データ・アクセスでカタログ化されたストアード・プロシージャーを呼び出すことができますが、 READS SQL DATA または MODIFIES SQL DATA でカタログ化されたストアード・プロシージャーを呼び出すことはできません。
- 最高で 16 レベルまでのネストされたストアード・プロシージャー呼び出しがサポートされています。たとえば、 PROC1 というストアード・プロシージャーが PROC2 を呼び出し、 PROC2 が PROC3 を呼び出す場合、これは 3 レベルのネストされたストアード・プロシージャーを表します。
- 呼び出し元および呼び出し先のストアード・プロシージャーは、ネストのどのレベルであっても NOT FENCED としてカタログ化することはできません。

ネストされた SQL プロシージャーは、 1 つまたは複数の結果セットをクライアント・アプリケーションまたは呼び出し元プロシージャーに戻すことができます。 SQL プロシージャーからの結果セットをクライアント・アプリケーションに戻すには、 WITH RETURN TO CLIENT 文節を使用して DECLARE CURSOR ステートメントを出します。 SQL プロシージャーからの結果セットを呼び出し元に戻すには (ここで、呼び出し元はクライアント・アプリケーションまたは呼び出し元のストアード・プロシージャーを表す)、 WITH RETURN TO CALLER 文節を使用して DECLARE CURSOR ステートメントを出します。

ネストされた組み込み SQL ストアード・プロシージャー (C で作成されたもの) およびネストされた CLI ストアード・プロシージャーは、結果セットをクライアント・アプリケーションまたは呼び出し元のストアード・プロシージャーに戻すことはできません。ストアード・プロシージャーが終了するときに、ネストされた組み込み SQL ストアード・プロシージャーまたはネストされた CLI ストアード・プロシージャーが、ストアード・プロシージャーの終了時にカーソルを開いたままにする場合、 DB2 がカーソルをクローズします。ストアード・プロシージャーから結果セットを戻す方法についての詳細は、 246ページの『ストアード・プロシージャーからの結果セットの戻り』を参照してください。

再帰的ストアード・プロシージャーでのカーソルの使用

SQL プロシージャーまたは組み込み SQL で書かれたストアード・プロシージャーの使用時にエラーを回避するには、再帰的 CALL ステートメントを発行する前に、すべてのオープン・カーソルをクローズしてください。

たとえば、次のようなコードのフラグメントを含むストアード・プロシージャー MYPROC があるとします。

```
OPEN c1;
CALL MYPROC();
CLOSE c1;
```

MYPROC が再帰的 CALL ステートメントを発行するときに、カーソル c1 がまだオープンしているため、MYPROC が呼び出されたときに DB2 はエラーを返します。 DB2 から返される特定のエラーは、MYPROC がカーソルで実行するアクションによって異なります。

MYPROC を正常に呼び出すためには、次の例のように、ネストされた CALL ステートメントの*前に* オープン・カーソルをクローズするように、MYPROC を書き換えてください。

```
OPEN c1;
CLOSE c1;
CALL MYPROC();
```

ネストされた CALL ステートメントを発行する前にオープン・カーソルをすべてクローズして、エラーを回避してください。

制約事項

ストアード・プロシージャーを作成する際には、以下の制約事項に従う必要があります。

- 標準入出力ストリームは使用できません。たとえば、 Java の System.out.println()、 C/C++ の printf()、COBOL での display の呼び出しです。ストアード・プロシージャー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、ファイルに書き出すことはできます。
- ストアード・プロシージャーを登録する CREATE PROCEDURE ステートメントで許可されている SQL ステートメントだけを含めます。 NO SQL、 READS SQL DATA、 CONTAINS SQL、または MODIFIES SQL DATA 文節を使用してストアード・プロシージャーをカタログ化する方法についての詳細は、 223ページの『ストアード・プロシージャーの SQL ステートメント』を参照してください。
- 次のいずれかの条件 (あるいは両方) が当てはまる場合には、 COMMIT ステートメントをストアード・プロシージャーで使用できません。
  - NO SQL 文節を使用してストアード・プロシージャーをカタログ化した。
  - マルチサイト更新を行っ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。
- ストアード・プロシージャーでは、次のものを含む、接続に関連するステートメントまたはコマンドは実行できません。
  - BACKUP
  - CONNECT
  - CONNECT TO
  - CONNECT RESET
  - CREATE DATABASE
  - DROP DATABASE
  - FORWARD RECOVERY
  - RESTORE
- UNIX ベースのシステムでは、NOT FENCED ストアード・プロシージャーは、 DB2 エージェント・プロセスのユーザー ID で実行します。 FENCED ストアード・プロシージャーは、 db2dari 実行可能ファイルのユーザー ID で実行されます。このユーザー ID は、 sqllib¥adm の .fenced ファイルの所有者に設定されます。このユーザー ID はストアード・プロシージャーに利用できるシステム・リソースを制御します。 db2dari 実行可能プログラムの詳細については、ご使用のプラットフォームに適した版の*概説およびインストール* を参照してください。
- 同じ数のパラメーターを受け入れるストアード・プロシージャーは、そのパラメーターの SQL データ・タイプが異なっているとしても、多重定義することはできません。

- ストアード・プロシージャーには、現行処理を終了させるコマンドを含めることはできません。ストアード・プロシージャーは、現行プロセスを終了させることなく常にクライアントに制御を戻さなければなりません。

OLE 自動化ストアード・プロシージャーの作成

OLE (オブジェクトのリンクと埋め込み) オートメーションは、 Microsoft Corporation の OLE 2.0 アーキテクチャーの一部です。 DB2 は、OLE オートメーション・オブジェクトの方式を外部ストアード・プロシージャーとして呼び出すことができます。 OLE オートメーションについては、 441ページの『OLE オートメーション UDF の作成』を参照してください。

OLE オートメーション・オブジェクトのコード化が終わったら、 CREATE PROCEDURE 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ストアード・プロシージャーとして登録する必要があります。 OLE 自動化ストアード・プロシージャーを登録するには、 LANGUAGE OLE 文節付きの CREATE PROCEDURE ステートメントを発行します。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。 OLE 自動化オブジェクトはインプロセス・サーバー (.DLL) としてインプリメントされる必要があります。

次の CREATE PROCEDURE ステートメントは、 OLE オートメーション・オブジェクト “db2smpl.salary” にある “median” 方式の “median” という自動化ストアード・プロシージャーを登録します。

```
CREATE PROCEDURE median (INOUT sal DOUBLE)
   EXTERNAL NAME 'db2smpl.salary!median'
   LANGUAGE OLE
   FENCED
   PARAMETER STYLE DB2SQL
```

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++ で作成された関数の呼び出し規則と同一です。

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自動的に処理します。サポートされている OLE オートメーション・タイプおよび SQL タイプの間の DB2 マッピングのリストについては、 444ページの表16 を参照してください。 SQL タイプから BASIC や C/C++ などの OLE プログラム言語への DB2 マッピングについては、 445ページの表17 を参照してください。

DB2 と OLE オートメーション・ストアード・プロシージャーの間で受け渡しされるデータは、参照呼び出しとして受け渡しされます。 DB2 は、以前に参照された表に載っていない、 DECIMAL または LOCATORS などの SQL タイプ、ブールや CURRENCY などの OLE オートメーション・タイプを、サポートしません。 BSTR にマップされる文字とグラフィック・データは、データベース・コード・ページから UCS-2 (Unicode としても知られている、 IBM コード・ページ 13488) スキーマに変換されます。戻される際に、データはデータベース・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。データベース・コード・ページから UCS-2 に、および UCS-2 からデータベース・コード・ページに変換するコード・ページ変換テーブルがインストールされていない場合、SQLCODE -332 (SQLSTATE 57017) を受け取ります。

OLE オートメーション・オブジェクトのコード化が終わったら、 CREATE PROCEDURE 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ストアード・プロシージャーとして登録する必要があります。 OLE 自動化ストアード・プロシージャーを登録するには、 LANGUAGE OLE 文節付きの CREATE PROCEDURE ステートメントを発行します。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。 OLE 自動化オブジェクトはインプロセス・サーバー (.DLL) としてインプリメントされる必要があります。

OUT パラメーターのストアード・プロシージャーの例

以下に、OUT ホスト変数の使用方法を示すサンプル・プログラムを挙げます。クライアント・アプリケーションは、 SAMPLE データベース中の従業員の給与の中央値を調べるストアード・プロシージャーを呼び出します。 (中央値の定義は、値の半分がその上下にあるというものです。) 次にその給与の中央値は、OUT ホスト変数を使用してクライアント・アプリケーションに戻されます。

このサンプル・プログラムは、SAMPLE データベース中のすべての従業員の給与の中央値を計算します。中央値を計算する既存の SQL 列関数がないため、給与の中央値は以下のようなアルゴリズムによって繰り返し計算することができます。
1. 表中のレコード数 *n* を調べる。
2. レコードを給与に基づいて並べる。
3. 行位置が `n / 2 + 1` であるレコードが検出されるまでレコードを取り出し続ける。
4. このレコードから給与の中央値を読み取る。

ストアード・プロシージャーの技法およびブロック・カーソルのいずれも使用しないアプリケーションは、 図5 で示されているように、ネットワークを経由して各給与を取り出さなければなりません。

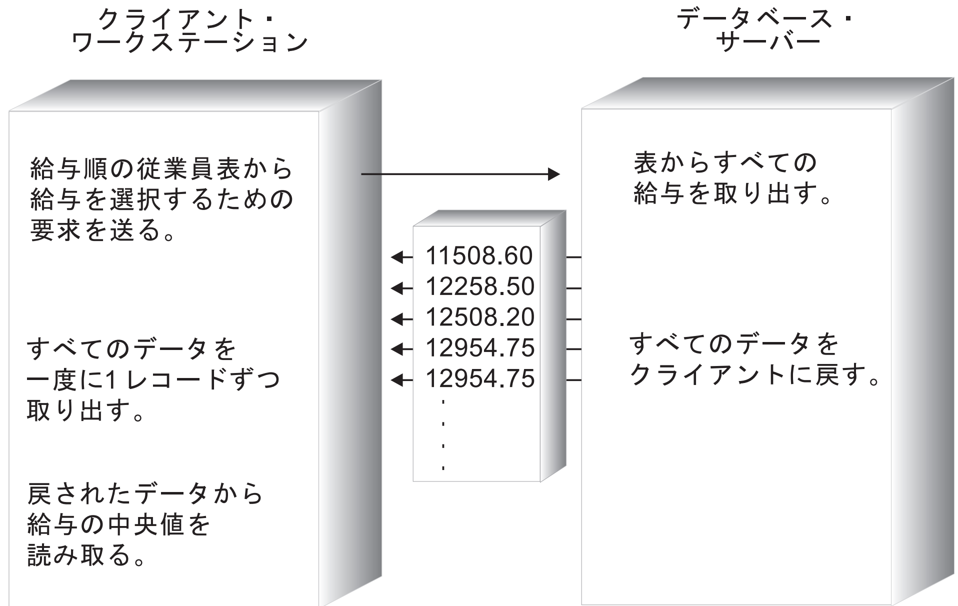


図 5. ストアード・プロシージャーを使用しない場合の中央値の例

第 n / 2 + 1 行の給与のみが必要であるため、アプリケーションはそれ以外のデータをすべて廃棄しますが、それはネットワークを経由して伝送された後のことです。

ストアード・プロシージャー技法を使用するアプリケーションは、ストアード・プロシージャーが不要なデータを処理したり破棄し、給与の中央値だけをクライアント・アプリケーションに戻すように設計できます。 図6 ではこの機能が示されています。

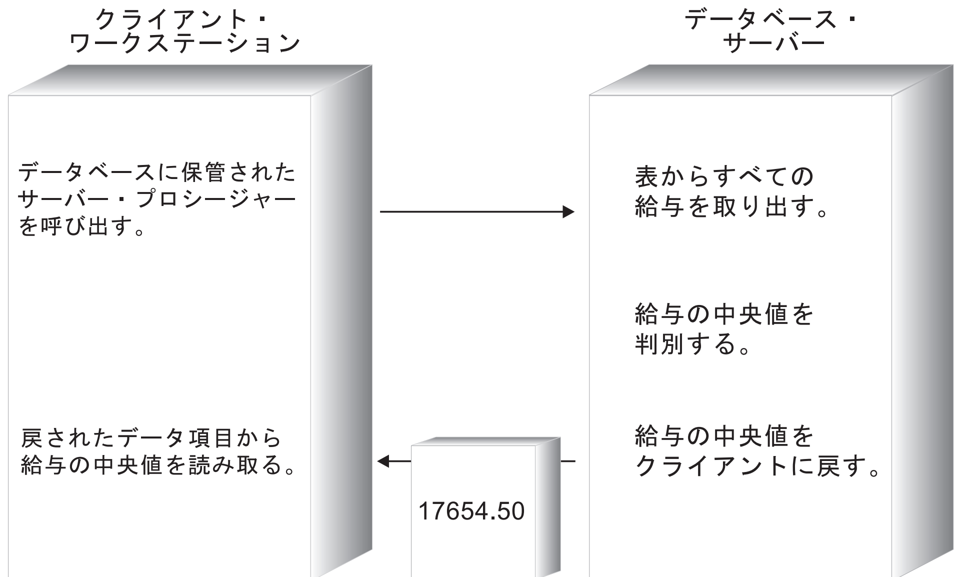


図 6. ストアード・プロシージャーを使用する場合の OUT パラメーターの例

『OUT クライアントの説明』は、 OUT ホスト変数クライアント・アプリケーションとストアード・プロシージャーの例を示します。 Java では以下のようなサンプル・プログラムを使用できます。

クライアント・アプリケーション Outcli.java

ストアード・プロシージャー Outsrv.sqlj

C では以下のようなサンプル・プログラムを使用できます。

クライアント・アプリケーション spclient.sqc

ストアード・プロシージャー spserver.sqc

OUT クライアントの説明

1. **インクルード・ファイル。** C クライアント・アプリケーションには、以下のファイルが含まれます。

   SQL 記号 SQL_TYP_FLOAT を定義する

   SQLDA 記述子域を定義する

   SQLCA エラー処理用の連絡域を定義する

   JDBC クライアント・アプリケーションは、以下のパッケージをインポートします。

   java.sql.* ユーザーのクライアントにインプリメントした Java の JDBC クラス。

java.math.BigDecimal
DB2 DECIMAL データ・タイプの Java サポートを提供します。

2. **データベースに接続する。**アプリケーションは、ストアード・プロシージャーを呼び出す前に、データベースに接続する必要があります。
3. **自動確定をオフにする。**クライアント・アプリケーションは、ストアード・プロシージャーを呼び出す前に自動確定を明示的に使用不能にします。自動確定を使用不能にすることにより、クライアント・アプリケーションは、ストアード・プロシージャーの制御が実行する作業がロールバックかコミット済みかを制御することができます。この例のストアード・プロシージャーは、クライアント・アプリケーションが条件ステートメントを使用して、ストアード・プロシージャーが実行する作業を簡単に確定またはロールバックできるようにするため、 SQLCODE 値を含む OUT パラメーターを戻します。
4. **ホスト変数を宣言および初期化する。**このステップでは、ホスト変数を宣言し、初期化します。 Java プログラムでは、ストアード・プロシージャーを呼び出す前に、各 INOUT または OUT パラメーターのデータ・タイプを登録する必要があります。
5. **ストアード・プロシージャーを呼び出す。**クライアント・アプリケーションは、 3 つのパラメーターを付けた CALL ステートメントを使用して SAMPLE データベースのストアード・プロシージャー `OUTPARAM` を呼び出します。
6. **出力パラメーターを検索する。** JDBC クライアント・アプリケーションは、ストアード・プロシージャーが戻す出力パラメーターの値を明示的に検索する必要があります。 C/C++ クライアント・アプリケーションの場合、 DB2 は、クライアント・アプリケーションが CALL ステートメントを実行するとき、 CALL ステートメントで使われているホスト変数の値を更新します。
7. **戻される SQLCODE の値を検査する。**クライアント・アプリケーションは、 SQLCODE を含む OUT パラメーターの値を検査して、トランザクションをロールバックするか確定するかを判別します。
8. **データベースから切断する。** DB2 の空きシステム・リソースが各接続で保持されるのを避けるには、クライアント・アプリケーションを終了する前に、データベースへの接続を明示的にクローズする必要があります。

`CHECKERR` マクロ / 関数は、プログラム外部にあるエラー検査ユーティリティーです。エラー検査ユーティリティーの所在は、ご使用のプログラム言語により異なります。

C
DB2 API を呼び出す C プログラムの場合、 utilapi.c 内の `sqlInfoPrint` 関数は、 utilapi.h 内の `API_SQL_CHECK` として再定義されます。 C 組み込み SQL プログラムの場合、 utilemb.sqc 内の `sqlInfoPrint` 関数は、 utilemb.h 内の `EMB_SQL_CHECK` として再定義されます。

Java
SQL エラーは `SQLException` としてスローされ、アプリケーションの `catch` ブロックで処理されます。

COBOL
`CHECKERR` は `checkerr.cbl` という名前の外部プログラムです。

このエラー検査ユーティリティーのソース・コードについては、 125ページの『プログラム例での GET ERROR MESSAGE の使用』を参照してください。

OUT クライアント・アプリケーションの例: Java

```
import java.sql.*;           // JDBC classes                                    1
import java.math.BigDecimal;  // BigDecimal support for packed decimal type

class Spclient
{
  static String sql = "";
  static String procName = "";
  static String inLanguage = "";
  static CallableStatement callStmt;
  static int outErrorCode = 0;
  static String outErrorLabel = "";
  static double outMedian = 0;

  static
  {
    try
    {
      System.out.println();
      System.out.println("Java Stored Procedure Sample");
      Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").newInstance();
    }
   catch (Exception e)
    {
      System.out.println("¥nError loading DB2 Driver...¥n");
      e.printStackTrace();
    }
  }

  public static void main(String argv[])
  {
    Connection con = null;
    // URL is jdbc:db2:dbname
    String url = "jdbc:db2:sample";

     try
    {
      // connect to sample database
      // connect with default id/password
      con = DriverManager.getConnection(url);                                   2

      // turn off autocommit
      con.setAutoCommit(false);                                                   3

      outLanguage(con);
      outParameter(con);
      inParameters(con);
      inoutParam(con, outMedian);
      resultSet(con);
      twoResultSets(con);
      allDataTypes(con);

      // rollback any changes to the database
      con.rollback();                                                             8
```

```
      con.close();
    }
    catch (Exception e)
    {
      try { con.close(); } catch (Exception x) { }
      e.printStackTrace ();
    }
  } // end main

  public static void outParameter(Connection con)
    throws SQLException
  {
    // prepare the CALL statement for OUT_PARAM
    procName = "OUT_PARAM";
    sql = "CALL " + procName + "(?, ?, ?)";
    callStmt = con.prepareCall(sql);

    // register the output parameter                                        4
    callStmt.registerOutParameter (1, Types.DOUBLE);
    callStmt.registerOutParameter (2, Types.INTEGER);
    callStmt.registerOutParameter (3, Types.CHAR);

    // call the stored procedure                                            5
    System.out.println ("¥nCall stored procedure named " + procName);
    callStmt.execute();

    // retrieve output parameters                                           6
    outMedian = callStmt.getDouble(1);
    outErrorCode = callStmt.getInt(2);
    outErrorLabel = callStmt.getString(3);

    if (outErrorCode == 0) {                                                7
      System.out.println(procName + " completed successfully");
      System.out.println ("Median salary returned from OUT_PARAM = "
        + outMedian);
    }
    else  { // stored procedure failed
      System.out.println(procName + " failed with SQLCODE "
        + outErrorCode);
      System.out.println(procName + " failed at " + outErrorLabel);
    }
  }
}
```

OUT クライアント・アプリケーションの例: C

```
#include <stdio.h>                                                        1
#include <stdlib.h>
#include <sql.h>
#include <sqlda.h>
#include <sqlca.h>
#include <string.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;
   /* Declare host variable for stored procedure name */
   char procname[254];

   /* Declare host variables for stored procedure error handling */
   sqlint32 out_sqlcode;                                                  4
   char out_buffer[33];
EXEC SQL END DECLARE SECTION;

int main(int argc, char *argv[]) {

   EXEC SQL CONNECT TO sample;                                            2
   EMB_SQL_CHECK("CONNECT TO SAMPLE");

   outparameter();

   EXEC SQL ROLLBACK;
   EMB_SQL_CHECK("ROLLBACK");
   printf("¥nStored procedure rolled back.¥n¥n");

   /* Disconnect from Remote Database */
   EXEC SQL CONNECT RESET;                                                8
   EMB_SQL_CHECK("CONNECT RESET");
   return 0;
}

int outparameter() {
   /********************************************************¥
   * Call OUT_PARAM stored procedure                         *
   ¥********************************************************/
   EXEC SQL BEGIN DECLARE SECTION;
     /* Declare host variables for passing data to OUT_PARAM */
     double out_median;
   EXEC SQL END DECLARE SECTION;

   strcpy(procname, "OUT_PARAM");
   printf("¥nCALL stored procedure named %s¥n", procname);

   /* OUT_PARAM is PS GENERAL, so do not pass a null indicator */
   EXEC SQL CALL :procname (:out_median, :out_sqlcode, :out_buffer);   5 6
   EMB_SQL_CHECK("CALL OUT_PARAM");
   /* Check that the stored procedure executed successfully */
   if (out_sqlcode == 0)                                                  7
```

```
    {
      printf("Stored procedure returned successfully.¥n");

      /***********************************************************¥
       * Display the median salary returned as an output parameter *
       ¥***********************************************************/

      printf("Median salary returned from OUT_PARAM = %8.2f¥n", out_median);

    }
    else
    { /* print the error message, roll back the transaction */
      printf("Stored procedure returned SQLCODE %d¥n", out_sqlcode);
      printf("from procedure section labelled ¥"%s¥".¥n", out_buffer);
    }

    return 0;
 }
```

OUT ストアード・プロシージャーの説明

1. **シグニチャーを宣言する。**プロシージャーは 3 つのパラメーターを戻します (中央値の場合は DOUBLE、SQLCODE の場合は INTEGER、およびエラー・メッセージの場合は CHAR)。各言語のプログラミングに関する章で指定した DB2 タイプ・マッピングを使用して、ストアード・プロシージャー関数定義の引き数と同じデータ・タイプを指定する必要があります。
2. **給料によって CURSOR 順序を宣言する。**複数のデータ行で作業するために、 C ストアード・プロシージャーは DECLARE CURSOR ステートメントを発行し、 JDBC ストアード・プロシージャーは ResultSet オブジェクトを作成します。 ORDER BY SALARY 節を使用すると、ストアード・プロシージャーは昇順で給料を検索することができます。
3. **全従業員の数を判別する。**単純な SELECT ステートメントと COUNT 関数を使用して、 EMPLOYEE 表の従業員の数を検索します。
4. **給与の中央値を取り出す。**給与の中央値を変数に割り当てるまで、FETCH ステートメントを続けて発行します。
5. **出力変数に給料の中央値を割り当てる。**クライアント・アプリケーションに給料の中央値を戻すには、ストアード・プロシージャー関数か、 OUT パラメーターに対応するメソッド宣言の引き数に値を割り当てます。
6. **クライアント・アプリケーションに戻す。**クライアントに値を戻すのは、PARAMETER STYLE DB2DARI ストアード・プロシージャーだけです。 DB2DARI ストアード・プロシージャーについて詳しくは、 789ページの『付録C. DB2DARI および DB2GENERAL ストアード・プロシージャーと UDF』を参照してください

OUT パラメーターのストアード・プロシージャーの例: Java

```
import java.sql.*;              // JDBC classes
import COM.ibm.db2.jdbc.app.*;  // DB2 JDBC classes
import java.math.BigDecimal;  // Packed Decimal class

public class Spserver
{
   public static void outParameter (double[] medianSalary,
         int[] errorCode, String[] errorLabel) throws SQLException        1
   {   try
      {
         int numRecords;
         int counter = 0;
         errorCode[0] = 0; // SQLCODE = 0 unless SQLException occurs

         // Get caller's connection to the database
         Connection con = DriverManager.getConnection("jdbc:default:connection");
         errorLabel[0] = "GET CONNECTION";

         String query = "SELECT COUNT(*) FROM staff";
         errorLabel[0] = "PREPARE COUNT STATEMENT";
         PreparedStatement stmt = con.prepareStatement(query);
         errorLabel[0] = "GET COUNT RESULT SET";
         ResultSet rs = stmt.executeQuery();

         // move to first row of result set
         rs.next();

         // set value for the output parameter
         errorLabel[0] = "GET NUMBER OF RECORDS";
         numRecords = rs.getInt(1);                                        3

         // clean up first result set
         rs.close();
         stmt.close();

         // get salary result set
         query = "SELECT CAST(salary AS DOUBLE) FROM staff "
            + "ORDER BY salary";
         errorLabel[0] = "PREPARE SALARY STATEMENT";
         PreparedStatement stmt2 = con.prepareStatement(query);
         errorLabel[0] = "GET SALARY RESULT SET";
         ResultSet rs2 = stmt2.executeQuery();                             2

         while (counter < (numRecords / 2 + 1))
         {
            errorLabel[0] = "MOVE TO NEXT ROW";
            rs2.next();                                                    4
            counter++;
         }
         errorLabel[0] = "GET MEDIAN SALARY";
         medianSalary[0] = rs2.getDouble(1);                               5

         // clean up resources
         rs2.close();
         stmt2.close();
         con.close();                                                      6

      }
```

```
      catch (SQLException sqle)
      {
         errorCode[0] = sqle.getErrorCode();
      }
   }
}
```

OUT パラメーターのストアード・プロシージャーの例: C

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlca.h>
#include <sqludf.h>
#include <sql.h>
#include <memory.h>

/* Declare function prototypes for this stored procedure library */

SQL_API_RC SQL_API_FN out_param (double *, sqlint32 *, char *);              1

EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;
/* Declare host variables for basic error-handling */
  sqlint32 out_sqlcode;
  char buffer[33];

/* Declare host variables used by multiple stored procedures */
  sqlint16 numRecords;
  double medianSalary;
EXEC SQL END DECLARE SECTION;

SQL_API_RC SQL_API_FN out_param (double *outMedianSalary,
  sqlint32 *out_sqlerror, char buffer[33])
{
  EXEC SQL INCLUDE SQLCA;

  EXEC SQL WHENEVER SQLERROR GOTO return_error;

  int counter = 0;
  *out_sqlerror = 0;

  strcpy(buffer, "DECLARE c1");
  EXEC SQL DECLARE c1 CURSOR FOR                                            2
    SELECT CAST(salary AS DOUBLE) FROM staff
    ORDER BY salary;

  strcpy(buffer, "SELECT");
  EXEC SQL SELECT COUNT(*) INTO :numRecords FROM staff;                     3

  strcpy(buffer, "OPEN");
  EXEC SQL OPEN c1;

  strcpy(buffer, "FETCH");
  while (counter < (numRecords / 2 + 1)) {
    EXEC SQL FETCH c1 INTO :medianSalary;                                   4

    /* Set value of OUT parameter to host variable */
    *outMedianSalary = medianSalary;                                        5
    counter = counter + 1;
```

```
  }

  strcpy(buffer, "CLOSE c1");
  EXEC SQL CLOSE c1;

  return (0);                                                    6

  /* Copy SQLCODE to OUT parameter if SQL error occurs */
return_error:
  {
    *out_sqlerror = SQLCODE;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    return (0);
  }

} /* end out_param function */
```

コード・ページに関する考慮事項

コード・ページに関する考慮事項は、サーバーにより異なります。

クライアント・プログラム (たとえばコード・ページ A を使用している) が、別のコード・ページ (たとえば、コード・ページ Z) を使用しているデータベースにアクセスするリモート・ストアード・プロシージャーを呼び出すと、以下のことが起きます。

1. 入力文字ストリング・パラメーター (ホスト変数で定義される、またはクライアント・アプリケーションにおいて SQLDA で定義されるもののいずれも) が、アプリケーションのコード・ページ (A) からデータベースのコード・ページ (Z) に変換される。 SQLDA で FOR BIT DATA として定義されるデータについては、変換は行われません。
2. 入力パラメーターが変換されると、データベース・マネージャーはそれ以上のコード・ページ変換を行わない。

   そのため、**データベースと同じコード・ページを使ってストアード・プロシージャーを実行しなければなりません。** すなわち、この例ではコード・ページ Z を使用しなければなりません。データベースと同じコード・ページを使ってサーバー・プロシージャーをプリコンパイル、コンパイル、そしてバインドしなければなりません。
3. ストアード・プロシージャーが完了すると、データベース・マネージャーは出力文字ストリング・パラメーター (ホスト変数として、またはクライアント・アプリケーションの SQLDA で定義) と SQLCA 文字フィールドを、データベースのコード・ページ (Z) からアプリケーションのコード・ページ (A) に変換する。 SQLDA で FOR BIT DATA として定義されるデータについては、変換は行われません。

注: ストアード・プロシージャーのパラメーターがサーバーで FOR BIT DATA として定義されている場合、 SQLDA に明示的に指定されるかどうかにかかわりなく、DB2 ユニバーサル・データベース (OS/390 版) または DB2 ユニバーサル・データベース (AS/400 版) への CALL ステートメントについては、変換は行われません。 (詳細については、*SQL 解説書* にある SQLDA の項を参照してください。)

このトピックの詳細については、529ページの『異なるコード・ページ間での変換』を参照してください。

C++ に関する考慮事項

ストアード・プロシージャーを C++ 言語で作成する場合には、以下の例のように `extern "C"` を使用してプロシージャー名を宣言できます。

```
extern "C" SQL_API_RC SQL_API_FN proc_name( short *parm1, char *parm2)
```

`extern "C"` は、C++ コンパイラーによる関数名のタイプ修飾 (すなわちマングル) を防止します。 この宣言の場合を除き、ストアード・プロシージャーを呼び出す際には、関数名のタイプ修飾も組み込む必要があります。

グラフィック・ホスト変数に関する考慮事項

パラメーター入力や出力によってグラフィック・データを送受信する、 C または C++ で書かれたストアード・プロシージャーはすべて、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グラフィック・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、ストアード・プロシージャー内の SQL ステートメントで操作されるグラフィック・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、グラフィック・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。グラフィック・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、ストアード・プロシージャー内の wchar_t 形式のグラフィック・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 ストアード・プロシージャー内で使用することができ、ストアード・プロシージャーのインターフェースを介してではなく、ストアード・プロシージャー内で、SQL ステートメント内のグラフィック・データに影響を及ぼします。 NOT FENCED ストアード・プロシージャーは、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ストアード・プロシージャーに渡したり、ストアード・プロシージャーから送られてくるグラフィック・データは、 WCHARTYPE オプションによってどのようにプリコンパイルされたかに関係なく、 DBCS 形式になります。

C アプリケーションにおけるグラフィック・データの処理については、 638ページの『C および C++ でのグラフィック・ホスト変数の処理』を参照してください。 EUC コード・セットおよびアプリケーションのガイドラインについては、 535ページの『日本語および中国語 (繁体字) EUC および UCS-2 コード・セットに関する考慮事項』、および 539ページの『ストアード・プロシージャーに関する考慮事項』を参照してください。

マルチサイト更新に関する考慮事項

アプリケーションが CONNECT TYPE 2 を使用して呼び出すストアード・プロシージャーは、動的にも静的にも COMMIT または ROLLBACK を発行できません。

ストアード・プロシージャーのパフォーマンスの向上

ストアード・プロシージャーのパフォーマンスを向上させるには、以下の技法の 1 つ以上をインプリメントすることを考慮してください。

- CHAR パラメーターではなく VARCHAR パラメーターを使用する (『CHAR パラメーターではなく VARCHAR パラメーターを使用する』で説明)
- DB2_STPROC_LOOKUP_FIRST レジストリー変数を ON に設定する (『DB2 がシステム・カタログ内のストアード・プロシージャーを検索するように強制する』で説明)
- ストアード・プロシージャーを NOT FENCED ストアード・プロシージャーとしてカタログ化する (244ページの『NOT FENCED ストアード・プロシージャー』で説明)

CHAR パラメーターではなく VARCHAR パラメーターを使用する

CHAR パラメーターではなく VARCHAR パラメーターを使用することにより、ストアード・プロシージャーのパフォーマンスを向上させることができます。 CHAR データ・タイプではなく VARCHAR データ・タイプを使用すると、 DB2 はパラメーターを渡す前にパラメーターにスペースを埋め込まなくても済みます。これにより、ネットワーク間でのパラメーターの伝送に必要な時間が減ることになります。

たとえば、クライアント・アプリケーションが CHAR(200) パラメーターとしてストリング `"A SHORT STRING"` をストアード・プロシージャーに渡す場合、 DB2 はパラメーターに 186 個のスペースを埋め込み、ストリングをヌルで終了させてから、ネットワークを介して 200 文字ストリングとヌル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。

それと比較して、VARCHAR(200) パラメーターとしてストリング ″A SHORT STRING″ をストアード・プロシージャーに渡すと、 DB2 はネットワークを介して 14 文字ストリングを渡すだけで済みます。

DB2 がシステム・カタログ内のストアード・プロシージャーを検索するように強制する

ストアード・プロシージャーを呼び出す場合、DB2 のデフォルトの振る舞いでは、 sqllib/function および sqllib/function/unfenced ディレクトリーを検索して、ストアード・プロシージャーと同じ名前を持つ共用ライブラリーを検索します。その後、共用ライブラリーの名前を検索して、システム・カタログ内のストアード・プロシージャーを探します。 PARAMETER TYPE DB2DARI のストアード・プロシージャーだけが共用ライブラリーと同じ名前を持つことができるため、 DB2 のデフォルトの振る舞いが有効なのは DB2DARI ストアード・プロシージャーだけです。別の PARAMETER TYPE でカタログ化されているストアード・プロシージャーを使用する場合、 DB2 が前述のディレクトリーの検索に費やす時間が、ストアード・プロシージャーのパフォーマンスの低下を招きます。

PARAMETER TYPE DB2DARI としてカタログ化されていないストアード・プロシージャーのパフォーマンスを拡張するには、 DB2_STPROC_LOOKUP_FIRST レジストリー変数の値を ON に設定します。このレジストリー変数を設定すると、前述のディレクト

リーを検索する前に、共用ライブラリーの名前を検索してシステム・カタログ内のストアード・プロシージャーを探すよう DB2 に強制します。

DB2_STPROC_LOOKUP_FIRST レジストリー変数の値を ON に設定するには、 CLP から次のコマンドを発行します。

```
db2set DB2_STPROC_LOOKUP_FIRST=ON
```

NOT FENCED ストアード・プロシージャー

ストアード・プロシージャーは、*FENCED* または *NOT FENCED* ストアード・プロシージャーとして実行できます。どちらとして実行するかは、CREATE PROCEDURE ステートメントに FENCED として登録したか、それとも NOT FENCED として登録したかによって決まります。

NOT FENCED ストアード・プロシージャーは、データベース・マネージャーと同じアドレス・スペース (DB2 エージェントのアドレス・スペース) で実行されます。ストアード・プロシージャーを NOT FENCED として実行すると、 FENCED ストアード・プロシージャーとして実行した場合に比べてパフォーマンスが向上します。なぜなら、FENCED ストアード・プロシージャーはデフォルトでは特殊な DB2 プロセスで実行されるからです。このプロセスのアドレス・スペースは DB2 システム・コントローラーとは異なります。

注:

1. NOT FENCED ストアード・プロシージャーを実行することによりパフォーマンスの向上を期待できる一方で、ユーザー・コードによりデータベース制御構造が不慮にまたは意図的に破損される危険性もあります。パフォーマンスによる効用を最大にする必要がある場合は、 NOT FENCED ストアード・プロシージャーだけを使用しなければなりません。ストアード・プロシージャーを NOT FENCED として実行する前に、すべてのストアード・プロシージャーを完全にテストしてください。
2. NOT FENCED ストアード・プロシージャーの実行中に重大エラーが起こった場合、データベース・マネージャーは、エラーがストアード・プロシージャー・コードとデータベース・コードのどちらで起こったのかを判別し、適切なリカバリーを試みます。

デバッグを目的とする場合は、*ローカルな FENCED ストアード・プロシージャー* を使用することを検討してください。ローカルの FENCED プロシージャーは、PARAMETER STYLE DB2DARI プロシージャーです。ローカルの FENCED プロシージャーを呼び出すには、 CALL <*library-name*>!<*entry-point*> を出します。ここで、*library-name* は共用ライブラリーの名前を表し、 *entry-point* はストアード・プロシージャーの共用ライブラリーの入り口点を表します。共用ライブラリーと入り口点の名前が同じ場合には、 CALL <*entry-point*> を出すことができます。

NOT FENCED および通常の FENCED ストアード・プロシージャーでは、デバッガーが余分なアドレス・スペースにもアクセスするので、デバッグ活動は複雑になります。

ローカルな FENCED ストアード・プロシージャーはアプリケーションのアドレス・スペースで実行されるので、デバッガーはアプリケーション・コードとストアード・プロシージャー・コードの両方にアクセスできます。ローカルな FENCED ストアード・プロシージャーをデバッグに使用できるようにするには、以下のステップを実行してください。

1. ストアード・プロシージャーを FENCED ストアード・プロシージャーとして登録する。
2. DB2_STPROC_ALLOW_LOCAL_FENCED レジストリー変数を true に設定する。レジストリー変数の詳細については、*管理の手引き: インプリメンテーション* を参照してください。
3. DB2 サーバーと同じマシン上でクライアント・アプリケーションを実行する。

注: ローカルな FENCED ストアード・プロシージャーをデバッグする際には、 226ページの『制約事項』にリストされた制約事項に違反したステートメントを使用しないように、注意する必要があります。 DB2 は、ローカルな FENCED ストアード・プロシージャーに対する呼び出しを、クライアント・アプリケーションのサブルーチンへの呼び出しとみなします。したがって、ローカルな FENCED ストアード・プロシージャーには、通常のストアード・プロシージャーに対する制約事項に違反するステートメントを含めることもできます。たとえば、プロシージャー本体に CONNECT ステートメントを含めることができます。

NOT FENCED ストアード・プロシージャーを作成する場合は、使用するオペレーティング・システムによっては、それがスレッド環境で実行されることもあるので注意が必要です。したがって、ストアード・プロシージャーは、これらの変数へのアクセスが直列化するように、常に再入可能であるかまたはその静的変数を管理するものでなければなりません。

注: ストアード・プロシージャーでは静的データを使用することはできません。なぜなら、DB2 はストアード・プロシージャー内の静的データがそれ以降の呼び出しで再度初期化されたかどうかを保証できないからです。

NOT FENCED ストアード・プロシージャーは、必ず WCHARTYPE NOCONVERT オプションを使用して作成してください。詳細については、640ページの『C および C++ での WCHARTYPE プリコンパイラー・オプション』を参照してください。

DB2 は、NOT FENCED ストアード・プロシージャーでは以下に示す機能をサポートしていません。

- 16 ビット
- マルチスレッド化
- ネストされた呼び出し: 呼び出しまたは別のストアード・プロシージャーからの呼び出し
- 結果セット: クライアント・アプリケーションまたは呼び出し元へ結果セットを戻す
- REXX

以下の DB2 API およびすべての DB2 CLI API は、 NOT FENCED ストアード・プロシージャーではサポートされません。
- BIND
- EXPORT
- IMPORT
- PRECOMPILE PROGRAM
- ROLLFORWARD DATABASE

ストアード・プロシージャーからの結果セットの戻り

DB2 CLI、ODBC、JDBC、または SQLJ クライアント・アプリケーションに 1 つまたは複数の結果セットを戻すように、ストアード・プロシージャーをコーディングすることができます。このサポートには以下のものが含まれます。

- DB2 CLI、ODBC、JDBC、および SQLJ クライアントだけが結果セットを受け入れることができる。
- DB2 ストアード・プロシージャーが DataJoiner バージョン 2 サーバーによってアクセス可能なサーバー上にある場合、組み込み SQL を使用する DB2 クライアントは複数の結果セットにアクセスできる。ホストおよび AS/400 プラットフォーム上にあるストアード・プロシージャーは、 DB2 コネクト・クライアントに複数の結果セットを戻すことができます。 DB2 ユニバーサル・データベース・サーバー上にあるストアード・プロシージャーは、ホストおよび AS/400 クライアントに複数の結果セットを戻すことができます。詳細については、DataJoiner、あるいはホストまたは AS/400 プラットフォームに関する製品資料を参照してください。
- クライアント・アプリケーション・プログラムが、戻される結果セットを記述できる。
- 結果セットは、アプリケーションによって逐次形式で処理されなければならない。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、ある結果セット上のカーソルをクローズし、次の結果セットの上でそれをオープンするために、特別な呼び出しが行われます (DB2 CLI の場合は `SQLMoreResults`、 JDBC の場合は `getMoreResults`、 SQLJ の場合は `getNextResultSet`)。
- ストアード・プロシージャーは、結果セット上でカーソルの宣言やオープンを行ったり、プロシージャーを終了する際にカーソルをオープンしたままにしておくことにより、その結果セットが戻されるように指示する。オープンしたままのカーソルが複数ある場合は、結果セットはそのカーソルがオープンされていたときの順序で戻されます。
- 読み取られていない行または取り出されていない行は結果セットに戻される。
- 結果セットを戻すストアード・プロシージャーは FENCED モードで実行する必要がある。
- COMMIT または ROLLBACK は WITH HOLD カーソルを除くすべてのカーソルをクローズする。

- DB2CLI.PROCEDURES 表中の RESULT_SETS 列は、ストアード・プロシージャーが結果セットを戻すかどうかを示す。 CREATE PROCEDURE ステートメントでストアード・プロシージャーを宣言すると、 DYNAMIC RESULT SETS 文節はストアード・プロシージャーによって戻された結果セットの数を示すようにこの値を設定します。

結果セットの処理についての追加情報は、以下を参照してください。

- DB2 CLI の場合、*コール・レベル・インターフェースの手引きおよび解説書* を参照してください。
- Java の場合、JDBC および SQLJ 仕様へのリンクについては、 `http://www.ibm.com/software/data/db2/java/` の DB2 Java Enablement Web ページを参照してください。

例: ストアード・プロシージャーからの結果セットを戻す

このストアード・プロシージャーの例では、以下のサポートされている言語で結果セットをクライアント・アプリケーションに戻す方法を示しています。

C spserver.sqc

Java Spserver.java

このストアード・プロシージャーの例は、 IN パラメーターを 1 つ受け入れ、 OUT パラメーター 1 つと結果セット 1 つを戻します。ストアード・プロシージャーでは IN パラメーターが使用されて、 STAFF 表で SALARY が IN パラメーターよりも大きい行の NAME、 JOB、SALARY 列の値を含む結果セットが作成されます。

1 CREATE PROCEDURE ステートメントの DYNAMIC RESULT SETS 文節を使用して、ストアード・プロシージャーを登録します。たとえば、C の組み込み SQL で作成されたストアード・プロシージャーを登録するには、以下のようなステートメントを発行します。

```
CREATE PROCEDURE RESULT_SET_CLIENT
   (IN salValue DOUBLE, OUT sqlCode INTEGER)
   DYNAMIC RESULT SETS 1
   LANGUAGE C
   PARAMETER STYLE GENERAL
   NO DBINFO
   FENCED
   READS SQL DATA
   PROGRAM TYPE SUB
   EXTERNAL NAME 'spserver!one_result_set_to_client'
```

2 C ストアード・プロシージャーの組み込み SQL では、 DECLARE CURSOR および OPEN CURSOR ステートメントを使用してオープン・カーソルを作成します。 CLI ストアード・プロシージャーでは、 `SQLPrepare` および `SQLBindParameter` API を使用して結果セットを作成します。 JDBC で作成された Java のストアード・プロシージャーでは、 `prepareStatement` および `executeQuery` メソッドを使用して結果セットを作成します。

3 カーソルまたは結果セットをクローズせずにデータベースへの接続をクローズします。このステップは、 C のストアード・プロシージャーの組み込み SQL には、適用されません。

4 Java ストアード・プロシージャー: PARAMETER STYLE JAVA ストアード・プロシージャーが戻す結果セットごとに、それに対応する *ResultSet[]* 引き数をストアード・プロシージャー方式のシグニチャーに含める必要があります。

C の例: SPSERVER.SQC (one_result_set_to_client)

```
SQL_API_RC SQL_API_FN one_result_set_to_client
  (double *insalary, sqlint32 *out_sqlerror)
{
  EXEC SQL INCLUDE SQLCA;

  EXEC SQL WHENEVER SQLERROR GOTO return_error;

  l_insalary = *insalary;
  *out_sqlerror = 0;

  EXEC SQL DECLARE c3 CURSOR FOR      2
    SELECT name, job, CAST(salary AS INTEGER)
    FROM staff
    WHERE salary > :l_insalary
    ORDER BY salary;

  EXEC SQL OPEN c3;                   2
  /* Leave cursor open to return result set */

  return (0);   3

  /* Copy SQLCODE to OUT parameter if SQL error occurs */
return_error:
  {
    *out_sqlerror = SQLCODE;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    return (0);
  }

} /* end one_result_set_to_client function */
```

Java の例: Spserver.java (resultSetToClient)

```
public static void resultSetToClient
     (double inSalaryThreshold, // double input
      int[] errorCode,            // SQLCODE output
      ResultSet[] rs)             // ResultSet output 4
      throws SQLException
{
   errorCode[0] = 0; // SQLCODE = 0 unless SQLException occurs

   try {
      // Get caller's connection to the database
      Connection con =
        DriverManager.getConnection("jdbc:default:connection");

      // get salary result set using a parameter marker
      String query = "SELECT name, job, CAST(salary AS DOUBLE) " +
              "FROM staff " +
              "WHERE salary > ? " +
              "ORDER BY salary";

      // prepare the SQL statement
      PreparedStatement stmt = con.prepareStatement(query);

      // set the value of the parameter marker (?)
      stmt.setDouble(1, inSalaryThreshold);

      // get the result set that will be returned to the client
      rs[0] = stmt.executeQuery(); 2

      // to return a result set to the client, do not close ResultSet
      con.close(); 3
   }

     catch (SQLException sqle)
   {
        errorCode[0] = sqle.getErrorCode();
   }
}
```

例: 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: このクライアント・アプリケーションの例では、以下のサポートされている言語でストアード・プロシージャーからの結果セットを受け入れる方法が示されています。

C (CLI を使用) spclient.c

Java Spclient.java

このクライアント・アプリケーションの例は、 `RESULT_SET_CLIENT` ストアード・プロシージャーを呼び出し、 1 つの結果セットを受け入れます。それから、クライアント・アプリケーションは結果セットの内容を表示します。

1 CREATE PROCEDURE ステートメントで宣言したパラメーターに対応する引き数を指定したストアード・プロシージャーを呼び出します。

2 JDBC アプリケーションは、`getNextResultSet` メソッドを使用して、ストアード・プロシージャーの最初の結果セットを受け入れます。

3 結果セットから行を取り出します。 CLI クライアントの例では、 `while` ループを使用して結果セットのすべての行を取り出して表示しています。 JDBC クライアントの例は、結果セットのすべての行を取り出して表示する `fetchAll` というクラス方式を呼び出します。

CLI の例: SPCLIENT.C (one_result_set_to_client):

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlcli1.h>
#include <sqlca.h>
#include "utilcli.h"          /* Header file for CLI sample code */

     SQLCHAR      stmt[50];
     SQLINTEGER   out_sqlcode;
     char         out_buffer[33];
     SQLINTEGER   indicator;
     struct sqlca sqlca;
     SQLRETURN    rc,rc1  ;
     char         procname[254];
     SQLHANDLE    henv;  /* environment handle */
     SQLHANDLE    hdbc;  /* connection handle */
     SQLHANDLE    hstmt1; /* statement handle */
     SQLHANDLE    hstmt2; /* statement handle */
     SQLRETURN    sqlrc = SQL_SUCCESS;
     double       out_median;

     int oneresultset1(SQLHANDLE);

int main(int argc, char *argv[])
 {
     SQLHANDLE    hstmt; /* statement handle */
     SQLHANDLE    hstmt_oneresult; /* statement handle */

     char         dbAlias[SQL_MAX_DSN_LENGTH + 1] ;
     char         user[MAX_UID_LENGTH + 1] ;
     char         pswd[MAX_PWD_LENGTH + 1] ;

     /* Declare  variables for passing data to INOUT_PARAM */
     double inout_median;

    /* checks the command line arguments */
     rc = CmdLineArgsCheck1( argc, argv, dbAlias, user, pswd );
     if ( rc != 0 )  return( 1 ) ;

    /* allocate an environment handle */
    printf("¥n    Allocate an environment handle.¥n");
    sqlrc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv ) ;
    if ( sqlrc != SQL_SUCCESS )
    {   printf( "¥n--ERROR while allocating the environment handle.¥n" ) ;
        printf( "  sqlrc             = %d¥n", sqlrc);
        printf( "  line              = %d¥n", __LINE__);
        printf( "  file              = %s¥n", __FILE__);
        return( 1 ) ;
    }

    /* allocate a database connection handle */
    printf("    Allocate a database connection handle.¥n");
    sqlrc = SQLAllocHandle( SQL_HANDLE_DBC, henv, &hdbc ) ;
```

```
 HANDLE_CHECK( SQL_HANDLE_ENV, henv, sqlrc, &henv, &hdbc ) ;

 /* connect to the database */
 printf( "    Connecting to the database %s ...¥n", dbAlias ) ;
 sqlrc = SQLConnect( hdbc,
                  (SQLCHAR *)dbAlias, SQL_NTS,
                  (SQLCHAR *)user, SQL_NTS,
                  (SQLCHAR *)pswd, SQL_NTS
                ) ;
 HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, sqlrc, &henv, &hdbc ) ;
 printf( "    Connected to the database %s.¥n", dbAlias ) ;


 /* set AUTOCOMMIT off */
 sqlrc = SQLSetConnectAttr( hdbc,
                            SQL_ATTR_AUTOCOMMIT,
                            SQL_AUTOCOMMIT_OFF, SQL_NTS) ;
 HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, sqlrc, &henv, &hdbc ) ;

 /* allocate one or more statement handles */
 printf("    Allocate a statement handle.¥n");
 sqlrc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt ) ;
 HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, sqlrc, &henv, &hdbc ) ;
 sqlrc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt_oneresult ) ;
 HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, sqlrc, &henv, &hdbc ) ;

/*********************************************************¥
* Call oneresultsettocaller stored procedure             *
¥*********************************************************/
 rc = oneresultset1(hstmt_oneresult);
 rc = SQLFreeHandle( SQL_HANDLE_STMT, hstmt_oneresult ) ;
 HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, rc, &henv, &hdbc ) ;

 /* ROLLBACK, free resources, and exit */

  rc = SQLEndTran( SQL_HANDLE_DBC, hdbc, SQL_COMMIT ) ;
  HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, rc, &henv, &hdbc ) ;

  printf("¥nStored procedure rolled back.¥n¥n");

/* Disconnect from Remote Database */

  rc = SQLFreeHandle( SQL_HANDLE_STMT, hstmt ) ;
  HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, rc, &henv, &hdbc ) ;

  printf( "¥n>Disconnecting .....¥n" ) ;
  rc = SQLDisconnect( hdbc ) ;
  HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, rc, &henv, &hdbc ) ;

  rc = SQLFreeHandle( SQL_HANDLE_DBC, hdbc ) ;
  HANDLE_CHECK( SQL_HANDLE_DBC, hdbc, rc, &henv, &hdbc ) ;

  rc = SQLFreeHandle( SQL_HANDLE_ENV,  henv ) ;
  if ( rc != SQL_SUCCESS ) return( SQL_ERROR )  ;
```

```
    return( SQL_SUCCESS ) ;
}

    int oneresultset1(hstmt)
    SQLHANDLE    hstmt; /* statement handle */
  {
  /*********************************************************¥
  * Call one_result_set_to_client stored procedure          *
  ¥*********************************************************/

   double          insalary = 20000;
   SQLINTEGER      salary_int;
   SQLSMALLINT     num_cols;
   char            name[40];
   char            job[10];

   strcpy(procname, "RESULT_SET_CALLER");

   printf("¥nCALL stored procedure:  %s¥n", procname);

   strcpy((char*)stmt,"CALL RESULT_SET_CALLER ( ?,? )");
   rc = SQLPrepare(hstmt, stmt, SQL_NTS);
   STMT_HANDLE_CHECK( hstmt, rc);

   /* Bind the parameter to application variables () */
   rc = SQLBindParameter(hstmt, 1,
                         SQL_PARAM_INPUT, SQL_C_DOUBLE,
                         SQL_DOUBLE,0,
                         0, &insalary,
                         0, NULL);
   rc = SQLBindParameter(hstmt, 2,
                         SQL_PARAM_OUTPUT, SQL_C_LONG,
                         SQL_INTEGER,0,
                         0, &out_sqlcode,
                         0, NULL);

   STMT_HANDLE_CHECK( hstmt, rc);
   rc = SQLExecute(hstmt);
   rc1 = SQLGetSQLCA(henv, hdbc, hstmt, &sqlca);
   STMT_HANDLE_CHECK( hstmt, rc);

   rc = SQLNumResultCols( hstmt, &num_cols ) ;
   STMT_HANDLE_CHECK( hstmt, rc);
   printf("Result set returned %d columns¥n", num_cols);

   /* bind columns to variables */
   rc = SQLBindCol( hstmt, 1, SQL_C_CHAR, name, 40, &indicator);
   STMT_HANDLE_CHECK( hstmt, rc);
   rc = SQLBindCol( hstmt, 2, SQL_C_CHAR, job, 10, &indicator);
   STMT_HANDLE_CHECK( hstmt, rc);
   rc = SQLBindCol( hstmt, 3, SQL_C_LONG, &salary_int, 0, &indicator);
   STMT_HANDLE_CHECK( hstmt, rc);

   /* fetch result set returned from stored procedure */
```

1

```
    rc = SQLFetch( hstmt );                                                    2
    rc1 = SQLGetSQLCA(henv, hdbc, hstmt, &sqlca);

    STMT_HANDLE_CHECK( hstmt, rc);

    printf("¥n--------Name---------,  --JOB--, ---Salary--  ¥n");
    while (rc  == SQL_SUCCESS && rc != SQL_NO_DATA_FOUND )                      3
    {
    printf("%20s,%10s,    %d¥n",name,job,salary_int);

    rc = SQLFetch( hstmt );
    }

    STMT_HANDLE_CHECK( hstmt, rc);

   /* Check that the stored procedure executed successfully */
    if (rc == SQL_SUCCESS) {
     printf("Stored procedure returned successfully.¥n");
     }
     else {
     printf("Stored procedure returned SQLCODE %d¥n", out_sqlcode);
     }
    rc = SQLCloseCursor(hstmt);

    return(rc);
}
```

Java の例: Spclient.java (resultSetToClient):

```
// prepare the CALL statement for RESULT_SET_CLIENT
procName = "RESULT_SET_CLIENT";
sql = "CALL " + procName + "(?, ?)"; 1
callStmt = con.prepareCall(sql);

// set input parameter to median value passed back by OUT_PARAM
callStmt.setDouble (1, outMedian);

// register the output parameter
callStmt.registerOutParameter (2, Types.INTEGER);

// call the stored procedure
System.out.println ("¥nCall stored procedure named " + procName);
callStmt.execute();

// retrieve output parameter
outErrorCode = callStmt.getInt(2);

if (outErrorCode == 0) {
   System.out.println(procName + " completed successfully");
   ResultSet rs = callStmt.getResultSet(); 2
   while (rs.next()) {
      fetchAll(rs); 3
   }

   // close ResultSet
   rs.close();
}
else  { // stored procedure failed
   System.out.println(procName + " failed with SQLCODE "
     + outErrorCode);
}
```

問題の解決

ストアード・プロシージャー・アプリケーションが適正に実行されない場合には、以下のことを確認してください。

- そのストアード・プロシージャーが、正しい呼び出し手順、コンパイル・オプションなどを用いて作成されていること。
- そのアプリケーションが、クライアント・アプリケーションとストアード・プロシージャーの両方が同じワークステーション上にある状態で、ローカルに実行されていること。
- ストアード・プロシージャーが、*アプリケーション構築の手引き* にある指示に従って、適切なロケーションに保管されていること。

  たとえば、OS/2 環境では、 FENCED ストアード・プロシージャーのダイナミック・リンク・ライブラリーは、データベース・サーバー上の `instance_name¥function` ディレクトリーにあります。
- アプリケーション (DB2 CLI および JDBC で作成されているものを除く) が、データベースにバインドされていること。
- そのストアード・プロシージャーが、クライアント・アプリケーションに SQLCA エラー情報を実際に戻すこと。
- ストアード・プロシージャーの関数名が、*大文字小文字を区別する* ため、クライアントとサーバーで厳密に一致していること。
- ストアード・プロシージャーを CREATE PROCEDURE ステートメントに登録する場合は、ストアード・プロシージャーの関数名がそのライブラリー名と一致していないこと。

  たとえば、データベース・マネージャーは、 Windows 32 ビット・オペレーティング・システムのライブラリー `myfunc.dll` に含まれているストアード・プロシージャー `myfunc` を、DB2DARI 関数として実行します。この場合、そのストアード・プロシージャーに関連付けられた CREATE PROCEDURE ステートメントにどんな値が指定されていてもかまいません。

注: Java ストアード・プロシージャーのデバッグの詳細については、 685ページの『Java でのストアード・プロシージャーのデバッグ』を参照してください。

コンパイラーに付属するデバッガーを使用して、他のアプリケーションと同じようにローカルな FENCED ストアード・プロシージャーをデバッグできます。提供されるデバッガーの使用法については、コンパイラーの資料をご覧ください。

たとえば、Windows NT 上の Visual Studio™ に付属するデバッガーを使用するには、以下のステップを実行してください。

ステップ 1. `DB2_STPROC_ALLOW_LOCAL_FENCED` レジストリー変数を `true` に設定する。

ステップ 2. `-Zi` および `-Od` フラグを指定して、ストアード・プロシージャー DLL のソース・ファイルをコンパイルし、 `-DEBUG` オプションを使ってその DLL をリンクする。

ステップ 3. 作成された DLL をサーバーの *instance_name* `¥function` ディレクトリーにコピーする。

ステップ 4. Visual Studio デバッガーを使って、サーバー上のクライアント・アプリケーションを呼び出す。クライアント・アプリケーション `outcli.exe` の場合は、次のコマンドを入力します。

```
msdev spclient.exe
```

ステップ 5. 「Visual Studio デバッガー (Visual Studio debugger)」ウィンドウがオープンしたら、**「プロジェクト (Project)」—>「設定 (Settings)」**を選択する。

ステップ 6. **「デバッグ (Debug)」**タブをクリックする。

ステップ 7. **「カテゴリ (Category)」**の矢印をクリックして、**「追加の DLL (Additional DLLs)」**を選択する。

ステップ 8. **「新規 (New)」**ボタンをクリックして、新規モジュールを作成する。

ステップ 9. **「ブラウズ (Browse)」**ボタンをクリックして、**「ブラウズ (Browse)」**ウィンドウをオープンする。

ステップ 10. モジュール `spserver.dll` を選択して、**「OK」**をクリックし、**「設定 (Settings)」**ウィンドウをクローズする。

ステップ 11. ストアード・プロシージャーのソース・ファイルをオープンして、ブレークポイントを設定する。

ステップ 12. 「実行 (Go)」ボタンをクリックする。ストアード・プロシージャーが呼び出されると、Visual Studio デバッガーは停止します。

ステップ 13. この時点で、Visual Studio デバッガーを使って、ストアード・プロシージャーをデバッグすることができます。

Visual Studio デバッガーの使用に関する詳細については、 Visual Studio の製品資料を参照してください。

第8章 SQL プロシージャーの作成

SQL プロシージャーおよび外部プロシージャーの比較 260
有効な SQL プロシージャー本体ステートメント 261
CREATE PROCEDURE ステートメントの発行 262
SQL プロシージャーでの処理条件 263
  条件ハンドラーの宣言 264
  SIGNAL および RESIGNAL ステートメント 266
  SQL プロシージャーの SQLCODE および SQLSTATE 変数 266
SQL プロシージャーでの動的 SQL の使用 267
ネストされた SQL プロシージャー 269
  ネストされた SQL プロシージャー間でのパラメーターの受け渡し 269
  ネストされた SQL プロシージャーからの結果セットの戻り 269
  ネストされた SQL プロシージャーでの制約事項 269
SQL プロシージャーからの結果セットの戻り 270
  呼び出し元またはクライアントへ結果セットを戻す 271
    クライアントへ結果セットを戻す . . 271
    呼び出し元へ結果セットを戻す . . . 271
  呼び出し元として結果セットを受け取る 272
SQL プロシージャーのデバッグ 273
  SQL プロシージャーのエラー・メッセージを表示する 273
  中間ファイルを使用した SQL プロシージャーのデバッグ 275
SQL プロシージャーの例 276

SQL プロシージャー とは、 CREATE PROCEDURE ステートメントにプロシージャー論理が含まれているストアード・プロシージャーを表します。 CREATE PROCEDURE ステートメントの中でコードが含まれている部分のことを、*プロシージャー本体* と呼びます。

SQL プロシージャーを作成するには、他の DDL ステートメントと同じように CREATE PROCEDURE ステートメントを発行するだけです。さらに、IBM DB2 ストアード・プロシージャー・ビルダーを使用すると、 DB2 にストアード・プロシージャーを定義したり、 SQL プロシージャーにソース・ステートメントを指定したり、プロシージャーを実行する用意をしたりすることができます。 IBM DB2 ストアード・プロシージャー・ビルダーの詳細については、 283ページの『第9章 IBM DB2 ストアード・プロシージャー・ビルダー』を参照してください。

この章では、プロシージャー本体を含む CREATE PROCEDURE ステートメントの作成方法が扱われています。 CREATE PROCEDURE ステートメントおよびプロシージャー本体の構文の詳細については、 *SQL 解説書* を参照してください。 IBM DB2 ストアード・プロシージャー・ビルダーを使用して SQL プロシージャーを作成する方法については、 283ページの『第9章 IBM DB2 ストアード・プロシージャー・ビルダー』を参照してください。

SQL プロシージャーおよび外部プロシージャーの比較

外部ストアード・プロシージャーの定義と同じように、 SQL プロシージャー定義には、以下のような情報が記載されています。

- プロシージャーの名前。
- パラメーター属性。
- プロシージャーが作成された言語。 SQL プロシージャーの場合、この言語は SQL になります。
- プロシージャーについてのその他の情報 (プロシージャーの固有名およびプロシージャーによって戻された結果セットの数など)。

外部ストアード・プロシージャーの CREATE PROCEDURE ステートメントとは異なり、 SQL プロシージャーの CREATE PROCEDURE ステートメントでは EXTERNAL 文節が指定されません。その代わり、 SQL プロシージャーにはそのストアード・プロシージャーのソース・ステートメントを含むプロシージャー本体があります。

次の例では、簡単なストアード・プロシージャーの CREATE PROCEDURE ステートメントが示されています。プロシージャー名、プロシージャーから (またはプロシージャーへ) 受け渡されるパラメーターのリスト、および LANGUAGE パラメーターはストアード・プロシージャーすべてに共通するものです。ただし、SQL の LANGUAGE 値およびプロシージャー本体を形成する BEGIN...END ブロックは、 SQL プロシージャー特有のものです。

```
CREATE PROCEDURE UPDATE_SALARY_1          1
(IN EMPLOYEE_NUMBER CHAR(6),              2
IN RATE INTEGER)                          2
LANGUAGE SQL                              3
BEGIN
   UPDATE EMPLOYEE                        4
   SET SALARY = SALARY * (1.0 * RATE / 100.0 )
   WHERE EMPNO = EMPLOYEE_NUMBER;
END
```

前述の例の注記:

1 ストアード・プロシージャーの名前は UPDATE_SALARY_1 です。

2 2 つのパラメーターのデータ・タイプは CHAR(6) および INTEGER です。どちらも入力パラメーターです。

3 LANGUAGE SQL は、これが SQL プロシージャーであることを示しています。プロシージャー本体は他のパラメーターに後続します。

4 プロシージャー本体は単一の SQL UPDATE ステートメントで構成され、従業員表の行を更新します。

SQL プロシージャー本体では、 OUT パラメーターを式の値として使用することはできません。 OUT パラメーターへ値を割り当てるには、割り当てステートメントを使用す

るか、 SELECT、VALUES および FETCH ステートメントの INTO 文節のターゲット変数とするしか方法はありません。 IN パラメーターは、割り当てまたは INTO 文節のターゲットとして使用することはできません。

有効な SQL プロシージャー本体ステートメント

プロシージャー本体は、単一の SQL プロシージャー・ステートメントで構成されます。プロシージャー本体では、以下のようなステートメントを使用できます。

割り当てステートメント
: 出力パラメーターまたは SQL 変数に値を割り当てます。 SQL 変数とは、プロシージャー本体内のみで定義されて使用される変数です。 IN パラメーターに値を割り当てることはできません。

CASE ステートメント
: 1 つまたは複数の条件の評価に基づいて、実行パスを選択します。このステートメントは、 *SQL 解説書* に記述されている CASE 式と同様なものです。

FOR ステートメント
: 表の各行でステートメントまたはステートメント・グループを実行します。

GET DIAGNOSTICS ステートメント
: GET DIAGNOSTICS ステートメントは、以前の SQL ステートメントに関する情報を戻します。

GOTO ステートメント
: プログラム制御を SQL ルーチン内のユーザー定義のラベルに移します。

IF ステートメント
: 条件の評価に基づいて実行パスを選択します。

ITERATE ステートメント
: 制御の流れをラベル付けされたブロックまたはループに渡します。

LEAVE ステートメント
: プログラム制御をループまたはコード・ブロックの外へ移します。

LOOP ステートメント
: ステートメントまたはステートメント・グループを複数回実行します。

REPEAT ステートメント
: ステートメントまたはステートメント・グループを検索条件が真になるまで実行します。

RESIGNAL ステートメント
: RESIGNAL ステートメントは、エラーまたは警告条件を再度シグナルするために条件ハンドラー内で使用されます。このステートメントは、エラーまたは警告を指定された SQLSTATE とともに戻します。また、メッセージ・テキストも一緒に戻されるように任意指定することができます。

RETURN ステートメント
制御を SQL プロシージャーから呼び出し元に戻します。さらに、整数値を呼び出し元に戻すこともできます。

SIGNAL ステートメント
SIGNAL ステートメントは、エラーまたは警告条件を通知するために使用されます。このステートメントは、エラーまたは警告を指定された SQLSTATE とともに戻します。また、メッセージ・テキストも一緒に戻されるように任意指定することができます。

SQL ステートメント
SQL プロシージャー本体には、 759ページの『付録A. サポートされる SQL ステートメント』 に記載されている SQL ステートメントをすべて含めることができます。

WHILE ステートメント
指定された条件が真である間、ステートメントまたはステートメント・グループを繰り返し実行します。

複合ステートメント
このリストにある他の種類のステートメントを 1 つまたは複数個含めたり、 SQL 変数宣言、条件ハンドラー、またはカーソル宣言を含めることができます。

SQL プロシージャー本体で使用可能な SQL ステートメントの完全なリストについては、 759ページの『付録A. サポートされる SQL ステートメント』を参照してください。これらのステートメントの詳細記述および構文については、 *SQL 解説書* を参照してください。

CREATE PROCEDURE ステートメントの発行

CREATE PROCEDURE ステートメントを DB2 コマンド行プロセッサー (DB2 CLP) スクリプトとして発行するには、スクリプト内の SQL ステートメントで代替終了文字を使用する必要があります。デフォルトの DB2 CLP スクリプトでは、セミコロン (';') 文字が SQL プロシージャー本体内の SQL ステートメントの終了文字として使用されています。

DB2 CLP スクリプトで別の終了文字を使用する場合には、標準 SQL ステートメントでは使用されていない文字を選択してください。次の例では、 script.db2 という名前の DB2 CLP スクリプトで、アットマーク ('@') が終了文字として使用されています。

```
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
@
```

コマンド行からの DB2 CLP スクリプトを処理するには、以下のような構文を使用します。

```
db2 -tdterm-char -vf script-name
```

ここで、 *term-char* は終了文字を表し、 *script-name* は処理する DB2 CLP スクリプト名を表します。たとえば、前述のスクリプトを処理するには、 CLP から以下のコマンドを実行します。

```
db2 -td@ -vf script.db2
```

SQL プロシージャーでの処理条件

条件ハンドラー は、ある条件が発生する際の SQL プロシージャーの振る舞いを決定します。一般的な DB2 条件、特定の SQLSTATE 値の定義された条件、または特定の SQLSTATE 値の定義された条件に関して、 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。一般的な条件および独自の条件を定義するための方法についての詳細は、264ページの『条件ハンドラーの宣言』を参照してください。

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 条件が発行され、それぞれの条件に対してハンドラーを宣言した場合には、 DB2 によって制御が対応するハンドラーに渡されます。その特定の条件に対してハンドラーを宣言しなかった場合、 DB2 は SQLSTATE および SQLCODE 変数にその条件に対応する値を設定して、制御をプロシージャー本体の次のステートメントに渡します。

SQL プロシージャーのステートメントが SQLEXCEPTION 条件を起こしており、その特定の SQLSTATE または SQLEXCEPTION 条件に対してハンドラーを宣言してある場合には、 DB2 によって制御がそのハンドラーに渡されます。 DB2 が正常にハンドラーを実行する場合には、 SQLSTATE および SQLCODE 値はそれぞれ ‘00000’ と 0 を戻します。

SQL プロシージャーのステートメントが SQLEXCEPTION 条件を起こしており、その特定の SQLSTATE または SQLEXCEPTION 条件に対してハンドラーを宣言していない場合、 DB2 は SQL プロシージャーを終了してからクライアントに戻ります。

条件ハンドラーの宣言

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 によって *condition* に合致する条件が起こされる場合には、 DB2 制御を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-type

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複合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複合ステートメントの SQL 操作がロールバックされることを指定します。 *SQL-procedure-statement* が完了した後に、ハンドラーが含まれる複合ステートメントの後から実行が継続されます。

注: UNDO ハンドラーは ATOMIC 複合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 または SQLSTATE が '02000' になる条件を識別します。

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。

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。独自の条件を定義する方法の詳細については、 *SQL 解説書* を参照してください。

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複合ステートメントを、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。複合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、複合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。複合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。

注: 条件ハンドラー内に別の条件ハンドラーを定義することはできません。

次の例は、単純な条件ハンドラーを表したものです。

例: CONTINUE ハンドラー: このハンドラーは、DB2 によって NOT FOUND 条件が起こされる際に、 *at_end* というローカル変数に 1 という値を割り当てます。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 not_found CONDITION FOR SQLSTATE '02000';
DECLARE CONTINUE HANDLER FOR not_found SET at_end=1;
```

例: EXIT ハンドラー: プロシージャーは、 NO_TABLE を SQLSTATE 42704 (*name* は未定義名) の条件名として宣言します。 NO_TABLE の条件ハンドラーは、 `Table does not exist` というストリングを OUT_BUFFER という出力パラメーターに置きます。それから、そのハンドラーが宣言された複合ステートメントから SQL プロシージャーを終了させます。

```
DECLARE NO_TABLE CONDITION FOR SQLSTATE '42704';
DECLARE EXIT HANDLER FOR NO_TABLE
   BEGIN
       SET OUT_BUFFER='Table does not exist';
   END
```

例: UNDO ハンドラー: SQLSTATE 42704 では、プロシージャーは SQLSTATE の名前を定義せずに UNDO 条件ハンドラーを宣言します。ハンドラーは SQL プロシージャーが現行の作業単位をロールバックするようにして、 `Table does not exist` ストリングを OUT_BUFFER 出力パラメーターに置いてから、ハンドラーが宣言された複合ステートメントを終了します。

```
DECLARE UNDO HANDLER FOR SQLSTATE '42704'
   BEGIN
       SET OUT_BUFFER='Table does not exist';
   END;
```

注: UNDO ハンドラーは ATOMIC 複合ステートメントのみで宣言できます。

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、カスタム定義された SQLSTATE で 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。プロシージャーによって SQLSTATE 72822 を起こす SIGNAL ステートメントが実行される場合、 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 に再定義して 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 に再定義されて、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 CONDITION HANDLER FOR SQLSTATE '72822'
   BEGIN
       IF ( var = 'OK' )
          RESIGNAL '72623' SET MESSAGE_TEXT = 'Got SQLSTATE 72822';
       ELSE
          RESIGNAL '72319' SET MESSAGE_TEXT = var;
   END;

SIGNAL SQLSTATE '72822';
```

SIGNAL および RESIGNAL ステートメントの詳細については、 *SQL 解説書* を参照してください。

SQL プロシージャーの SQLCODE および SQLSTATE 変数

SQL プロシージャーをデバッグする助けとして、 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';
```

さらに、CONTINUE 条件ハンドラーを使用して、 SQLSTATE および SQLCODE の値を SQL プロシージャー本体のローカル変数に割り当てられます。それから、これらのローカル変数を使用してプロシージャー論理を制御したり、値を出力パラメーターとして戻すことができます。以下の例では、SQL プロシージャーは制御を各 SQL ステートメントに続くステートメントに戻します。 SQLCODE は、RETCODE というローカル変数に設定します。

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

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

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、または SQLSTATE 値は ‘00000’ に設定されます。

SQL プロシージャーでの動的 SQL の使用

外部ストアード・プロシージャーと同様、 SQL プロシージャーは動的 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しょう。 SQL プロシージャーで動的 SQL を発行するのに PREPARE および EXECUTE ステートメントを使用するには、SQL プロシージャー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。

ステップ 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のに十分な大きさの VARCHAR タイプの変数を宣言します。

ステップ 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 ('?') 記号を、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。

ステップ 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。

ステップ 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・マーカーが含まれている場合、 USING 文節を使用して変数の値と置換します。

注: SQL プロシージャーの 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 プロシージャーがそのステートメント名を定義した効力範囲を出ると、 DB2 はステートメント名をアクセスできなくなります。複合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。

例: 動的 SQL ステートメント: 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。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、部門 *deptno* 内の従業員の行を DEPT_*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・マーカーが含まれています。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
```

ネストされた SQL プロシージャー

SQL プロシージャーに CALL ステートメントを組み込んで他の SQL プロシージャーを呼び出すことができます。この機能はネストされた ストアード・プロシージャーと呼ばれています。この機能によって、既存の SQL プロシージャーを再使用してもっと複雑なアプリケーションを設計することが可能になります。

ネストされた SQL プロシージャー間でのパラメーターの受け渡し

呼び出し元 SQL プロシージャー内からターゲット SQL プロシージャーを呼び出すには、呼び出し元の CALL ステートメントに適切な数およびタイプのパラメーターを含めるだけです。ターゲットが OUT パラメーターを戻す場合には、呼び出し元は戻された値を独自のステートメントで使用することができます。

たとえば、 “SALES_TARGET” という名前のターゲット SQL プロシージャーを呼び出し、 INTEGER タイプの単一の OUT パラメーターを受け入れる SQL プロシージャーを、以下のような SQL で作成できます。

```
CREATE PROCEDURE NEST_SALES(OUT budget DECIMAL(11,2))
LANGUAGE SQL
BEGIN
   DECLARE total INTEGER DEFAULT 0;
   SET total = 6;
   CALL SALES_TARGET(total);
   SET budget = total * 10000;
END
```

ネストされた SQL プロシージャーからの結果セットの戻り

ターゲット SQL プロシージャーによって結果セットが戻される場合、呼び出し元またはクライアント・アプリケーションのいずれかが結果セットを受け取ります。どちらが受け取るかは、ターゲット SQL プロシージャーが発行する DECLARE CURSOR ステートメントによって決まります。ターゲット内の WITH RETURN TO CLIENT 文節が含まれている DECLARE CURSOR ステートメントでは、呼び出し元は結果セットを受け取りません。 WITH RETURN TO CLIENT カーソルでは、結果セットはクライアント・アプリケーションに直接戻されます。

ネストされた SQL プロシージャーから結果セットを戻す方法についての詳細は、 271 ページの『呼び出し元またはクライアントへ結果セットを戻す』を参照してください。

ネストされた SQL プロシージャーでの制約事項

アプリケーション・アーキテクチャーを設計する際には、以下のような制約事項を念頭に置いてください。

LANGUAGE

SQL プロシージャーは、 SQL または C で作成されたストアード・プロシー

ジャーしか呼び出せません。 SQL プロシージャー内からは他のホスト言語のストアード・プロシージャーを呼び出すことはできません。

16 レベルのネスト

SQL プロシージャーでは、最大 16 レベルのネストされた呼び出ししか含められません。 SQL プロシージャー A が SQL プロシージャー B を呼び出し、SQL プロシージャー B が SQL プロシージャー C を呼び出すシナリオは、3 レベルのネストされた呼び出しの例です。

再帰

再帰的に呼び出す SQL プロシージャーを作成することができます。再帰的 SQL プロシージャーは、前述の最大レベルのネストについての制約事項に準拠する必要があります。

セキュリティー

SQL プロシージャーは、より高い SQL データ・アクセス・レベルでカタログ化されたターゲット SQL プロシージャーを呼び出すことができません。たとえば、CONTAINS SQL 文節で作成された SQL プロシージャーは CONTAINS SQL 文節または NO SQL 文節のいずれかで作成された SQL プロシージャーを呼び出せますが、 READS SQL DATA 文節または MODIFIES SQL DATA 文節のいずれかで作成された SQL プロシージャーは呼び出せません。

NO SQL 文節で作成された SQL プロシージャーは CALL ステートメントを発行できません。

SQL プロシージャーからの結果セットの戻り

SQL プロシージャーから結果セットを戻す方法は、外部ストアード・プロシージャーから結果セットを戻す方法に似ています。 SQL プロシージャーからの結果セットを受け入れるには、クライアント・アプリケーションは CLI、 JDBC、または SQLJ アプリケーション・プログラミング・インターフェースを使用する必要があります。他の SQL プロシージャーを呼び出す SQL プロシージャーは、それらのプロシージャーからの結果セットを受け入れることができます。 SQL プロシージャーからの結果セットを戻すには、以下のような SQL プロシージャーを作成します。

1. CREATE PROCEDURE ステートメントの DYNAMIC RESULT SETS 文節を使用して、SQL プロシージャーによって戻される結果セットの数を宣言します。
2. DECLARE CURSOR ステートメントを使用してカーソルを宣言します。
3. OPEN CURSOR ステートメントを使用してカーソルをオープンします。
4. カーソルをクローズせずに SQL プロシージャーを終了します。

たとえば、INOUT パラメーター *threshold* の値に応じて、単一の結果セットを戻す SQL プロシージャーを以下のように作成できます。

```
CREATE PROCEDURE RESULT_SET (INOUT threshold SMALLINT)
LANGUAGE SQL
DYNAMIC RESULT SETS 1
BEGIN
```

```
   DECLARE cur1 CURSOR WITH RETURN TO CALLER FOR
       SELECT name, job, years
       FROM staff
       WHERE years < threshold;
   OPEN cur1;
END
```

呼び出し元またはクライアントへ結果セットを戻す

ご使用のアプリケーションがネストされた SQL プロシージャーから結果セットを戻す場合には、 DECLARE CURSOR ステートメントの WITH RETURN 文節を使用して DB2 が適切な位置に結果セットを戻すようにする必要があります。ターゲット SQL プロシージャーが結果セットを呼び出し元 SQL プロシージャーに戻す場合には、呼び出し元は ALLOCATE CURSOR および ASSOCIATE RESULT SET LOCATOR ステートメントを使用して結果セットにアクセスし、使用する必要があります。

クライアントへ結果セットを戻す

SQL プロシージャーからの結果セットをクライアント・アプリケーションに必ず戻すようにするには、結果セットに関連した DECLARE CURSOR ステートメントの WITH RETURN TO CLIENT 文節を使用します。次の例では、 SQL プロシージャー “CLIENT_SET” は DECLARE CURSOR ステートメントの 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。これは、“CLIENT_SET” がネストされた SQL プロシージャーの CALL ステートメントであるときも同様です。

```
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
```

呼び出し元へ結果セットを戻す

呼び出し元がクライアント・アプリケーションであるかまたは別の SQL プロシージャーであるかにかかわりなく SQL プロシージャーの直接の呼び出し元へ結果セットを戻すには、結果セットに関連した DECLARE CURSOR ステートメントの 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 の呼び出し元に戻します。

```
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
```

呼び出し元として結果セットを受け取る

呼び出し元の SQL プロシージャーがターゲット SQL プロシージャーから結果セットを受け取ることを期待している場合には、 ALLOCATE CURSOR および ASSOCIATE RESULT SET LOCATOR ステートメントを使用して結果セットにアクセスし、使用する必要があります。

ASSOCIATE RESULT SET LOCATOR

呼び出し元に 1 つまたは複数の結果セットを戻すターゲット SQL ステートメントへの CALL ステートメントの後は、呼び出し元の SQL プロシージャーはこのステートメントを発行して、戻される結果セットにそれぞれ結果セット・ロケーター変数を割り当てる必要があります。たとえば、ターゲット SQL プロシージャーから 3 つの結果セットを受け取るはずの呼び出し元 SQL プロシージャーは、以下のような SQL で構成されるかもしれません。

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

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

ALLOCATE CURSOR

呼び出し元 SQL プロシージャーで ALLOCATE CURSOR ステートメントを使用して、ターゲット SQL プロシージャーから戻された結果セットをオープンします。 ALLOCATE CURSOR ステートメントを使用するには、 ASSOCIATE RESULT SET LOCATORS ステートメントを通して結果セットは既に結果セット・ロケーターに関連していなければなりません。 SQL プロシージャーが ALLOCATE CURSOR ステートメントを発行すると、ALLOCATE CURSOR で宣言されているカーソル名を使用して結果セットから行を取り出すことができます。前述の ASSOCIATE LOCATORS の例を拡張するには、以下のような SQL を使用して SQL プロシージャーは最初に戻された結果セットから行を取り出すことができます。

```
DECLARE result1 RESULT_SET_LOCATOR VARYING;
DECLARE result2 RESULT_SET_LOCATOR VARYING;
DECLARE result3 RESULT_SET_LOCATOR VARYING;
CALL targetProcedure();
ASSOCIATE RESULT SET LOCATORS(result1, result2, result3)
   WITH PROCEDURE targetProcedure;
ALLOCATE rsCur CURSOR FOR result1;
WHILE (at_end = 0) DO
```

```
      SET total1 = total1 + var1;
      SET total2 = total2 + var2;
      FETCH FROM rsCur INTO var1, var2;
   END WHILE;
```

SQL プロシージャーのデバッグ

SQL プロシージャーを作成した後には、 262ページの『CREATE PROCEDURE ステートメントの発行』で記述されているように CREATE PROCEDURE ステートメントを発行する必要があります。ある場合では、 CREATE PROCEDURE ステートメントへの応答として DB2 によってエラーが戻されるかもしれません。 DB2 によって戻されたエラーについて、さらに情報を (エラーを訂正するための説明および提案) 検索するには、 CLP で以下のようなコマンドを実行します。

```
db2 "? error-code"
```

ここで、 *error-code* は、エラーによって戻された SQLCODE または SQLSTATE を表します。たとえば、 CREATE PROCEDURE ステートメントが SQLCODE “SQL0469N” (“パラメーター・モードは無効です”) 付きのエラーを戻す場合には、以下のようなコマンドを実行します。

```
db2 "? SQL0469"
```

DB2 は以下のようなメッセージを戻します。

```
Explanation:  One of the following errors occurred:

o   a parameter in an SQL procedure is declared as OUT and is
    used as input in the procedure body

o   a parameter in an SQL procedure is declared as IN and is
    modified in the procedure body.

User Response:  Change the attribute of the parameter to INOUT,
or change the use of the parameter within the procedure.
```

メッセージが表示された後は、 “User Response” セクションの提案に沿って SQL プロシージャーを変更してください。

SQL プロシージャーのエラー・メッセージを表示する

SQL プロシージャーの CREATE PROCEDURE ステートメントを発行すると DB2 は SQL プロシージャー本体の構文を受け入れるかもしれませんが、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するのに失敗するかもしれません。このような状況下では、 DB2 は通常、エラー・メッセージを含むログ・ファイルを作成します。このログ・ファイルおよび他の中間ファイルは、 275ページの『中間ファイルを使用した SQL プロシージャーのデバッグ』に記載されています。

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 *$DB2PATH*/function/routine/sqlproc/*$DATABASE*/*$SCHEMA*/tmp

ここで、*$DB2PATH* はインスタンス・ディレクトリーの位置を表し、*$DATABASE* はデータベース名を表し、 *$SCHEMA* は SQL プロシージャーを作成するのに使用されるスキーマ名を表します。

Windows NT

%DB2PATH%¥function¥routine¥sqlproc¥*%DB%*¥*%SCHEMA%*¥tmp

ここで、*%DB2PATH%* はインスタンス・ディレクトリーの位置を表し、*%DB%* はデータベース名を表し、 *%SCHEMA%* は SQL プロシージャーを作成するのに使用されるスキーマ名を表します。

さらに、アプリケーションの CALL ステートメントを発行して、サンプルのストアード・プロシージャーである db2udp!get_error_messages を呼び出せます。それには、以下のような構文を使用します。

```
CALL db2udp!get_error_messages(schema-name, file-name, message-text)
```

ここで *schema-name* は、 SQL プロシージャーのスキーマを表す入力パラメーターです。 *file-name* は SQL プロシージャーの生成されたファイル名を表す入力パラメーターであり、 *message-text* はメッセージ・ログ・ファイルでメッセージ・テキストを戻す出力パラメーターです。

たとえば、以下のような Java アプリケーションを使用して SQL プロシージャーのエラー・メッセージを表示できます。

```
public static String getErrorMessages(Connection  con,
  String procschema, String filename) throws Exception
{
    String filecontents = null;
    // prepare the CALL statement
    CallableStatement stmt = null;
    try
    {
        String sql = "Call db2udp!get_error_messages(?, ?, ?) ";
        stmt = con.prepareCall (sql);

        // set all parameters (input and output)
        stmt.registerOutParameter( 3, java.sql.Types.LONGVARCHAR  );
        stmt.setString( 1, procschema );
        stmt.setString( 2, filename );

        // call the stored procedure
        boolean isrs = stmt.execute();
        filecontents = stmt.getString(3);

        System.out.println("SQL Procedure - getErrorMessages "
```

```
                + filecontents);
            return filecontents;
        }
        catch (Exception e) { throw e; }
        finally
        {
            if (stmt != null) stmt.close();
        }
    }
```

SQL プロシージャーのエラー・メッセージを表示するには、以下のような C アプリケーションを使用できます。

```
int getErrors(char inputSchema[9], char inputFilename[9],
                         char outputFilecontents[32000])
{
  EXEC SQL BEGIN DECLARE SECTION;
    char    procschema[100] = "";
    char    filename[100] = "";
    char    filecontents[32000] = "";
  EXEC SQL END DECLARE SECTION;

  strcpy (procschema, inputSchema);
  strcpy (filename, inputFilename);

  EXEC SQL CALL "db2udp!get_error_messages"
    (:procschema, :filename, :filecontents);
  if ( sqlca.sqlcode != 0 )
  {
    printf("Call failed. Code: %d¥n", sqlca.sqlcode);
    return 1;
  }
  else
  {
    printf("¥nSQL Procedure - getErrors:¥n%s¥n", filecontents);
  }
  strcpy (outputFilecontents, filecontents);
  return 0;
}
```

注: DB2 が作成するのに失敗した SQL プロシージャーのエラー・メッセージを表示する前に、SQL プロシージャーのプロシージャー名と生成されたファイル名の両方を知る必要があります。 CREATE PROCEDURE ステートメントの一部としてプロシージャー・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
```

中間ファイルを使用した SQL プロシージャーのデバッグ

SQL プロシージャーで CREATE PROCEDURE ステートメントを発行して DB2 が SQL プロシージャー本体の構文を受け入れる場合には、 DB2 はいくつかの中間ファイ

ルを使用して SQL プロシージャーを作成します。 SQL プロシージャーが正常に作成されると、 DB2 は中間ファイルを除去してシステム・リソースを保護します。 DB2 によって CREATE PROCEDURE 構文が認められますが SQL プロシージャーの作成が失敗すると、プリコンパイル・バインド、および CREATE PROCEDURE プロセスのコンパイル段階を追跡するログ・ファイルが保持されます。

UNIX システムでは、 DB2 は中間ファイルを保持するため、 *instance*/function/routine/sqlproc/*dbAlias*/*schema* という基本ディレクトリーを使用します。ここで、*instance* は DB2 インスタンスのパスを表し、 *dbAlias* はデータベース別名を表します。また、*schema* は CREATE PROCEDURE ステートメントが発行されたスキーマを表します。

OS/2 および Windows 32 ビット・オペレーティング・システムでは、 DB2 は中間ファイルを保管するため、 *instance*¥function¥routine¥sqlproc¥*dbAlias*¥ *schema* という基本ディレクトリーを使用します。ここで、*instance* は DB2 インスタンスのパスを表し、 *dbAlias* はデータベース別名を表します。また、*schema* は CREATE PROCEDURE ステートメントが発行されたスキーマを表します。

SQL プロシージャーが正常に作成されましたが、 CALL ステートメントから予期された結果が戻されない場合には、中間ファイルを調べることができるでしょう。 DB2 が中間ファイルを除去しないようにするには、以下のコマンドを使用して DB2_SQLROUTINE_KEEP_FILES DB2 レジストリー変数を “yes” に設定します。

```
db2set DB2_SQLROUTINE_KEEP_FILES="yes"
```

DB2 でレジストリー変数の新しい値を使用するには、データベースを再始動する必要があります。

SQL プロシージャーの例

このセクションでは、 SQL プロシージャー本体に表示される各ステートメントの使用方法を示した例が記載されています。これらおよび他の SQL プロシージャーの例については (SQL 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含む)、以下のようなディレクトリーを参照してください。

UNIX オペレーティング・システム
: *$HOME*/sqllib/samples/sqlproc。ここで *$HOME* は、 DB2 インスタンス・ディレクトリーの位置を表します。

Windows 32 ビット・オペレーティング・システム
: *%DRIVE%*¥sqllib¥samples¥sqlproc。ここで *%DRIVE%* は、 DB2 のインストール先のドライブを表します。

例 1: CASE ステートメント: 以下の SQL プロシージャーは、CASE ステートメントの使用方法を示したものです。このプロシージャーでは、入力パラメーターとして ID 番

号および従業員のランク付けを受け取ります。 CASE ステートメントは、ランク付けごとに異なった UPDATE ステートメントを使用して従業員の給与およびボーナスを変更します。

```
CREATE PROCEDURE UPDATE_SALARY
(IN employee_number CHAR(6), IN rating INT)
LANGUAGE SQL
BEGIN
  DECLARE not_found CONDITION FOR SQLSTATE '02000';
  DECLARE EXIT HANDLER FOR not_found
    SIGNAL SQLSTATE '02444';

  CASE rating
    WHEN 1 THEN
      UPDATE employee
      SET salary = salary * 1.10, bonus = 1000
      WHERE empno = employee_number;
    WHEN 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 CASE;
END
```

例 2: ネストされた IF および WHILE ステートメントを使った複合ステートメント: 以下の例では、ネストされた IF ステートメント、 WHILE ステートメント、および割り当てステートメントを含んだ複合ステートメントを示します。また、この例では、エラー・コードのクラスに対して SQL 変数、カーソル、およびハンドラーを宣言する方法も示します。

このプロシージャーは、部門番号を入力パラメーターとして受け取ります。プロシージャー本体の WHILE ステートメントは、部門の各従業員の給与およびボーナスを取り出します。 WHILE ステートメント内の IF ステートメントは、各従業員の労働年数および現在の給与に応じて給与を更新します。この部門に存在するすべての従業員のレコードが処理された場合、従業員レコードを検索する FETCH ステートメントは SQLSTATE 20000 を受け取ります。 *not_found* 条件ハンドラーによって WHILE ステートメントの検索条件が偽になり、WHILE ステートメントの実行は終了します。

```
CREATE PROCEDURE BUMP_SALARY_IF (IN deptnumber SMALLINT)
LANGUAGE SQL
BEGIN
   DECLARE v_salary DOUBLE;
   DECLARE v_years SMALLINT;
   DECLARE v_id SMALLINT;
   DECLARE at_end INT DEFAULT 0;
   DECLARE not_found CONDITION FOR SQLSTATE '02000';

   -- CAST salary as DOUBLE because SQL procedures do not support DECIMAL
```

```
    DECLARE C1 CURSOR FOR
      SELECT id, CAST(salary AS DOUBLE), years
      FROM staff;
    DECLARE CONTINUE HANDLER FOR not_found
      SET at_end = 1;

    OPEN C1;
    FETCH C1 INTO v_id, v_salary, v_years;
    WHILE at_end = 0 DO
      IF (v_salary < 2000 * v_years)
        THEN UPDATE staff
          SET salary = 2150 * v_years
          WHERE id = v_id;
        ELSEIF (v_salary < 5000 * v_years)
          THEN IF (v_salary < 3000 * v_years)
            THEN UPDATE staff
              SET salary = 3000 * v_years
              WHERE id = v_id;
          ELSE UPDATE staff
            SET salary = v_salary * 1.10
            WHERE id = v_id;
          END IF;
        ELSE UPDATE staff
          SET job = 'PREZ'
          WHERE id = v_id;
      END IF;
      FETCH C1 INTO v_id, v_salary, v_years;
    END WHILE;
    CLOSE C1;
 END
```

例 3: グローバル一時表および結果セット付きのネストされた SQL プロシージャーの使用:

ASSOCIATE RESULT SET LOCATOR および ALLOCATE CURSOR ステートメントを使用して、呼び出し先 SQL プロシージャー `temp_table_insert` から呼び出し元 SQL プロシージャー `temp_table_create` に結果セットを戻す方法を以下の例に示します。この例では、呼び出し先 SQL プロシージャーが呼び出し元 SQL プロシージャーによって作成されたグローバル一時表を使用できることも示されています。

この例では、クライアント・アプリケーションまたは別の SQL プロシージャーが `temp_table_create` を呼び出します。そして、`temp_table_create` はグローバル一時表 SESSION.TTT を作成してから `temp_table_insert` を呼び出します。

SESSION.TTT グローバル一時表を使用するため、 `temp_table_insert` には DECLARE GLOBAL TEMPORARY TABLE ステートメントが含まれています。これは、`temp_table_create` が SESSION.TTT を作成するために発行するステートメントと同様のものです。 `temp_table_create` が発行するステートメントとの違いは、`temp_table_insert` の DECLARE GLOBAL TEMPORARY TABLE ステートメントがいつも偽である IF に含まれていることにあります。 IF ステートメントは、 DB2 がグ

ローバル一時表を 2 度目に作成しないようにしますが、 SQL プロシージャーが後続するステートメントでグローバル一時表を使用するのを可能にします。

別の SQL プロシージャーで作成されたグローバル一時表から結果セットを戻すには、temp_table_insert は DECLARE CURSOR ステートメントを新しい範囲内で発行する必要があります。 temp_table_insert は、新しい効力範囲の要求を満たす DECLARE CURSOR および OPEN CURSOR ステートメントを複合 SQL ブロックで発行します。SQL プロシージャーが終了する前にカーソルはクローズされないので、 DB2 は結果セットを呼び出し元 temp_table_create に戻します。

呼び出し先 SQL プロシージャーから結果セットを受け入れるため、temp_table_create は、 temp_table_insert を結果セットの発信元であることを識別する ASSOCIATE RESULT SET LOCATOR ステートメントを発行します。それから、temp_table_create は、結果セット・ロケーターの ALLOCATE CURSOR ステートメントを発行して結果セットをオープンします。 ALLOCATE CURSOR ステートメントが正常に実行されると、 SQL プロシージャーは通常どおり結果セットで作業を行えます。この例では、temp_table_create は結果セットからすべての行を取り出し、出力パラメーターに列の値を追加します。

注: グローバル一時表を使用する SQL プロシージャーで CREATE PROCEDURE ステートメントを発行する前に、ユーザー一時表スペースを作成する必要があります。ユーザー一時表スペースを作成するには、以下の SQL ステートメントを発行します。

```
CREATE USER TEMPORARY TABLESPACE ts1
   MANAGED BY SYSTEM USING ('ts1file');
```

ここで、*ts1* はユーザー一時表スペースの名前を表し、 *ts1file* は表スペースによって使用されるコンテナー名を表します。

```
CREATE PROCEDURE temp_table_create(IN parm1 INTEGER, IN parm2 INTEGER,
   OUT parm3 INTEGER, OUT parm4 INTEGER)
   LANGUAGE SQL
   BEGIN
      DECLARE loc1 RESULT_SET_LOCATOR VARYING;
      DECLARE total3,total4 INTEGER DEFAULT 0;
      DECLARE rcolumn1, rcolumn2 INTEGER DEFAULT 0;
      DECLARE result_set_end INTEGER DEFAULT 0;
      DECLARE CONTINUE HANDLER FOR NOT FOUND, SQLEXCEPTION, SQLWARNING
      BEGIN
        SET result_set_end = 1;
      END;
      --Create the temporary table that is used in both this SQL procedure
      --and in the SQL procedure called by this SQL procedure.
      DECLARE GLOBAL TEMPORARY TABLE ttt(column1 INT, column2 INT)
         NOT LOGGED;
      --Insert rows into the temporary table.
      --The result set includes these rows.
      INSERT INTO session.ttt(column1, column2) VALUES ( parm1+1, parm2+1);
      INSERT INTO session.ttt(column1, column2) VALUES ( parm1+2, parm2+2);
```

```
      --Make a nested call to the 'temp_table_insert' SQL procedure.
      CALL temp_table_insert(parm1, parm2);
      --Issue the ASSOCIATE RESULT SET LOCATOR statement to
      --accept a single result set from 'temp_table_insert'.
      --If 'temp_table_insert' returns multiple result sets,
      --you must declare one locator variable (for example,
      --ASSOCIATE RESULT SET LOCATOR(loc1, loc2, loc3) for each result set.
      ASSOCIATE RESULT SET LOCATOR(loc1) WITH PROCEDURE temp_table_insert;
      --The ALLOCATE statement is similar to the OPEN statement.
      --It makes the result set available in this SQL procedure.
      ALLOCATE cursor1 CURSOR FOR RESULT SET loc1;
      --Insert rows into the temporary table.
      --The result set does not include these rows.
      INSERT INTO session.ttt(column1, column2) VALUES ( parm1+5, parm2+5);
      INSERT INTO session.ttt(column1, column2) VALUES ( parm1+6, parm2+6);
      SET result_set_end = 0;
      --Fetch the columns from the first row of the result set.
      FETCH FROM cursor1 INTO rcolumn1, rcolumn2;
      WHILE (result_set_end = 0) DO
        SET total3 = total3 + rcolumn1;
        SET total4 = total4 + rcolumn2;
        --Fetch columns from the result set for the
        --next iteration of the WHILE loop.
        FETCH FROM cursor1 INTO rcolumn1, rcolumn2;
      END WHILE;
      CLOSE cursor1;
      SET parm3 = total3;
      SET parm4 = total4;
   END @

CREATE PROCEDURE temp_table_insert (IN parm1 INTEGER, IN parm2 INTEGER )
   LANGUAGE SQL
   BEGIN
      DECLARE result_set_end INTEGER DEFAULT 0;
      DECLARE CONTINUE HANDLER FOR NOT FOUND BEGIN
        SET result_set_end = 1;
      END;
      --To use a temporary table that is created by a different stored
      --procedure, include a DECLARE GLOBAL TEMPORARY TABLE statement
      --inside a condition statement that always evaluates to false.
      IF (1 = 0) THEN
        DECLARE GLOBAL TEMPORARY TABLE ttt(column1 INT, column2 INT)
           NOT LOGGED;
      END IF;
      --Insert rows into the temporary table.
      --The result set includes these rows.
      INSERT INTO session.ttt(column1, column2) VALUES ( parm1+3, parm2+3);
      INSERT INTO session.ttt(column1, column2) VALUES ( parm1+4, parm2+4);
      --To return a result set from the temporary table, issue
      --the DECLARE CURSOR statement inside a new scope, such as
      --a compound SQL statement (BEGIN...END block).
      --Issue the DECLARE CURSOR statement after the DECLARE
      --GLOBAL TEMPORARY TABLE statement.
      BEGIN
         --The WITH RETURN TO CALLER clause causes the SQL procedure
```

```
      --to return its result set to the calling procedure.
      DECLARE cur1 CURSOR WITH RETURN TO CALLER
         FOR SELECT * FROM session.ttt;
      --To return a result set, open a cursor without closing the cursor.
     OPEN cur1 ;
   END;
END
```

第9章 IBM DB2 ストアード・プロシージャー・ビルダー

ストアード・プロシージャー・ビルダーの説明 283
ストアード・プロシージャー・ビルダーを使用する利点 284
ストアード・プロシージャーの新規作成 284
既存のストアード・プロシージャーでの作業 285
ストアード・プロシージャー・ビルダー・プロジェクトの作成 286
ストアード・プロシージャーのデバッグ 286

ストアード・プロシージャー・ビルダーの説明

ストアード・プロシージャー・ビルダーは、 DB2 ストアード・プロシージャーの迅速な開発をサポートするグラフィカル・アプリケーションです。ストアード・プロシージャー・ビルダーを使用すると以下のような作業を行えます。

- 新規のストアード・プロシージャーを作成する
- ローカルおよびリモート DB2 サーバーでストアード・プロシージャーを作成する
- 既存のストアード・プロシージャーを変更して再作成する
- インストールされたストアード・プロシージャーをテストしてからデバッグする

ストアード・プロシージャー付きのアプリケーションを作成するために、ストアード・プロシージャー・ビルダーは DB2 ユニバーサル・データベース・ファミリー全体 (OS/2、 OS/390、AS/400、AIX、HP-UX、Linux、 Solaris 実行環境および Windows 32 ビット・オペレーティング・システムを含む) をサポートする単一の開発環境を提供します。

ストアード・プロシージャー・ビルダーでサポートされているプラットフォーム:

ストアード・プロシージャー・ビルダーは、アプリケーション開発クライアント (AIX 版、Solaris (Solaris** Operating Environment**) 版、 Windows 32 ビット・オペレーティング・システム版) のオプション・コンポーネントです。

クライアント上のストアード・プロシージャー・ビルダーを使用すると、以下のプラットフォームで DB2 ユニバーサル・データベース・サーバーの Java ストアード・プロシージャーおよび SQL プロシージャーを構築および活用できます。

ストアード・プロシージャー言語	サポートされている DB2 UDB プラットフォーム
Java	OS/2、OS/390、AIX、HP-UX、Linux、Solaris、および Windows 32 ビット・オペレーティング・システム
SQL	OS/2、OS/390、AS/400、AIX、HP-UX、Linux、Solaris、および Windows 32 ビット・オペレーティング・システム

© Copyright IBM Corp. 1993, 2001

SQL ストアード・プロシージャーをエクスポートして、既存の Java クラス・ファイルから Java ストアード・プロシージャーを作成できます。快適な開発環境を提供するために、ストアード・プロシージャー・ビルダーのコード・エディターでは、デフォルトのキーの結び付け以外に vi または emacs を使用することができます。

ストアード・プロシージャー・ビルダーの立ち上げ:

Windows 32 ビット・オペレーティング・システムでは、コマンド行で db2spb コマンドを実行することにより、 DB2 ユニバーサル・データベース・プログラム・グループから、または以下のいずれかの開発アプリケーションからストアード・プロシージャー・ビルダーを立ち上げることができます。

- Microsoft Visual C++ 5.0 および 6.0
- Microsoft Visual Basic 5.0 および 6.0
- IBM VisualAge for Java

AIX および Solaris のクライアントでは、コマンド行で db2spb コマンドを実行してストアード・プロシージャー・ビルダーを立ち上げることができます。

ストアード・プロシージャー・ビルダーを Java とともにインプリメントすると、データベースへの接続はすべて JDBC (Java Database Connectivity) で管理されます。 JDBC ドライバーを使用すると、ローカル DB2 の別名すべてに接続できます。あるいは、ホスト、ポート、およびデータベース名を指定できるその他のデータベースにも接続できます。

注: ストアード・プロシージャー・ビルダーを使用するには、開発用の DB2 データベースに接続する必要があります。ストアード・プロシージャー・ビルダーの使用方法の詳細については、 IBM DB2 ストアード・プロシージャー・ビルダーのオンライン・ヘルプを参照してください。

ストアード・プロシージャー・ビルダーを使用する利点

ストアード・プロシージャー・ビルダーはストアード・プロシージャーを作成、インストール、およびテストするための使いやすい開発環境を提供します。これにより、DB2 サーバーでのストアード・プロシージャーの登録、構築、およびインストールに関する詳細な点よりも、ストアード・プロシージャーのロジックに注意を向けることができます。ストアード・プロシージャー・ビルダーは、ストアード・プロシージャーを開発したプラットフォームとは異なるプラットフォームで構築することを可能にすることにより、プラットフォーム間アプリケーションの開発をサポートします。

ストアード・プロシージャーの新規作成

ストアード・プロシージャー・ビルダーを使用すると、 DB2 データベース・サーバーでのストアード・プロシージャーの作成およびインストールのプロセスを、大幅に単純

化することができます。ストアード・プロシージャー・ウィザードおよび SQL Assistant を使用すると、ストアード・プロシージャーの開発が簡単になります。

ストアード・プロシージャー・ビルダーでは、 Java または SQL で作成された可搬性の高いストアード・プロシージャーを作成できます。ストアード・プロシージャー・ウィザードを使用して基本的な SQL 構造を作成します。次に、高度で複雑なストアード・プロシージャー論理が含まれるように、ソース・コード・エディターを使用してストアード・プロシージャーを修正します。

ストアード・プロシージャーを作成する際には、 1 つの結果セット、複数の結果セット、または出力パラメーターだけのいずれかを戻すように選択できます。ストアード・プロシージャーがデータベース表を作成または更新する際には、結果セットを戻されないように選択できます。ストアード・プロシージャー・ウィザードを使用して、ストアード・プロシージャーがクライアント・アプリケーションからホスト変数の値を受け取るように、ストアード・プロシージャーの入出力パラメーターを定義することができます。さらに、ストアード・プロシージャーでは複数の SQL ステートメントを作成できます。ストアード・プロシージャーは case 値を受け取ってから、複数の照会の 1 つを選択します。

ターゲット・データベースでストアード・プロシージャーを構築するには、ストアード・プロシージャー・ウィザードで**「終了 (Finish)」**をクリックするだけです。 CREATE PROCEDURE ステートメントを使用して、 DB2 にストアード・プロシージャーを手動で登録する必要はありません。

既存のストアード・プロシージャーでの作業

データベース・サーバーにストアード・プロシージャーを正常に構築した後は、プロシージャーを変更、再構築、実行、およびテストできます。ストアード・プロシージャーを変更することによって、複雑なストアード・プロシージャー論理が含まれるように、いくつかのメソッドをコードに追加できます。ストアード・プロシージャー・ビルダーでストアード・プロシージャーをオープンすると、エディターにソース・コードが表示されます。エディターは、 Java または SQL で作成されたストアード・プロシージャーに依存する言語です。

ストアード・プロシージャー・ビルダー内からストアード・プロシージャーを実行すると、プロシージャーが正常にインストールされているかを確認するためのテストを行うことができます。ストアード・プロシージャーを実行すると、ストアード・プロシージャーの設定方法に合わせて、入力したテスト入力パラメーターの値に基づいた結果セットを戻すことができます。ストアード・プロシージャーをテストすると、ストアード・プロシージャーが DB2 データベース・サーバーに正常にインストールされていることがわかるので、クライアント・アプリケーションのプログラミングが容易になります。それにより、クライアント・アプリケーションの作成およびデバッグだけに注意を向けることができます。

ストアード・プロシージャー・ビルダーの「プロジェクト (Project)」ウィンドウを使用すると、ストアード・プロシージャーを除去したり、別のデータベース接続にコピーしたりできます。

ストアード・プロシージャー・ビルダー・プロジェクトの作成

新規および既存のストアード・プロシージャー・ビルダー・プロジェクトをオープンすると、「プロジェクト (Project)」ウィンドウは DB2 データベースに常駐しているすべてのストアード・プロシージャーの中で、現在接続されているものを表示します。ストアード・プロシージャーをフィルターして、名前またはスキーマに基づいてプロシージャーを表示することができます。ストアード・プロシージャー・ビルダー・プロジェクトは、データベースに正常に構築されなかった接続情報およびストアード・プロシージャー・オブジェクトだけを保管します。

ストアード・プロシージャーのデバッグ

ストアード・プロシージャー・ビルダーおよび IBM 分散デバッガー (別売品) を使用すると、 DB2 サーバーにインストールされたストアード・プロシージャーをリモート操作でデバッグすることができます。ストアード・プロシージャーをデバッグするには、ストアード・プロシージャーをデバッグ・モードで構築してからクライアント IP アドレスのデバッグ項目を追加します。それから、ストアード・プロシージャーを実行する必要があります。アプリケーション・プログラム内からストアード・プロシージャーをデバッグする必要はありません。ストアード・プロシージャーのテストは、呼び出しアプリケーション・プログラムのテストとは別個に行えます。

ストアード・プロシージャー・ビルダーを使用すると、ストアード・プロシージャーのデバッグ表のデバッグ項目を変更、追加、および除去する権限が与えられているすべてのストアード・プロシージャーを表示できます。データベース管理者または選択したストアード・プロシージャーを作成したユーザーであるならば、ストアード・プロシージャーをデバッグする権限を他のユーザーに授与できます。

第4部 オブジェクト関連プログラミング

© Copyright IBM Corp. 1993, 2001

第10章 オブジェクト関連機能の使用

DB2 オブジェクト拡張を使用する理由 . . . 289
DB2 のオブジェクト関連機能 289
ユーザー定義の特殊タイプ 292
オブジェクトの振る舞いの定義: ユーザー定義のルーチン 292

DB2 オブジェクト拡張を使用する理由

現代プログラミング言語技術における重要な最新開発の 1 つに、*オブジェクト指向* があります。オブジェクト指向は、アプリケーション・ドメイン内のエンティティーを分類し、互いに関連する独立したオブジェクトとしてモデル化する概念です。オブジェクトの外部的な振る舞いおよび特性は外部に出されますが、オブジェクトの内部インプリメンテーションの詳細は隠れたままになります。オブジェクト指向により、アプリケーション・ドメインのオブジェクトの類似点と相違点を明らかにし、それらのオブジェクトを関連したタイプごとにグループ化することができます。同一タイプのオブジェクトは、そのタイプ固有の振る舞いの集合を共用するため、アプリケーション・ドメインのオブジェクトの振る舞いを反映して同じように振る舞います。

DB2 のオブジェクト拡張により、関連技術の利点の上に成り立つオブジェクト技術の多くの利点を実現化できます。 関係システムでは、データ・タイプを使用して、そのインスタンス (またはオブジェクト) が保管される表の列中のデータを記述します。このようなインスタンスでの操作は、そのような表現が許可されているすべての場所で呼び出し可能な演算子または関数によってサポートされます。

DB2 のオブジェクト拡張では、オブジェクト指向 (OO) の概念と方法論をそこに組み込むことができます。

DB2 のオブジェクト関連機能

データをオブジェクト指向でモデル化するのに役立つ以下のようなオブジェクト指向機能があります。

とても大きなオブジェクト用のデータ・タイプ

システムでモデル化する必要のあるデータが大きくて複雑な場合があります (たとえば、テキスト、音声、技術データ、またはビデオなど)。このサイズのオブジェクトでは、 VARCHAR または VARGRAPHIC データ・タイプは十分な大きさではないかもしれません。 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ために、3 つのデータ・タイプを備えています。その 3 つのデータ・タイプとは、2 進ラージ・オブジェクト (BLOB)、 1 バイト文字ラージ・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。

© Copyright IBM Corp. 1993, 2001

ユーザー定義のデータ・タイプ

ユーザー定義のタイプでオブジェクトのセマンティクスを制御できます。たとえば、ご使用のアプリケーションで “text” というタイプまたは “address” というタイプが必要になるかもしれません。これらのタイプは組み込みタイプとしては存在しません。しかし、 DB2 のオブジェクト関連機能を使用すると、これらのタイプを定義してデータベースで使用できます。

さらに、ユーザー定義のタイプは以下のように分類できます。

特殊タイプ

特殊タイプは既存の DB2 組み込みデータ・タイプに基づいたものです。つまり、特殊タイプは内部的には組み込みタイプと同様です。しかし、これらのタイプのセマンティクスを定義することができます。さらに、DB2 にはとても大きな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。特殊タイプは以下の中の 1 つのラージ・オブジェクト (LOB) データ・タイプに基づいています。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できるかもしれません。

構造タイプ

構造タイプは、単一タイプのもとにオブジェクト属性の集合を集めるためのものです。

ユーザー定義の振る舞い

DB2 がオブジェクトに操作を行うことを可能にするため、 SQL または外部言語で独自のルーチンを作成することができます。ユーザー定義のルーチンには、以下の 2 つのタイプがあります。

ユーザー定義関数 (UDF)

UDF とはユーザーが定義できる関数で、組み込み関数や演算子と同様に SQL 照会でのオブジェクトの操作をサポートします。 UDF を使用して、ユーザー定義タイプだけではなくいかなるタイプの列値を操作できます。

ユーザー定義メソッド

UDF と同様、オブジェクトの振る舞いを定義しますが、特定のユーザー定義構造タイプに固くカプセル化されています。

索引拡張

索引拡張は、DB2 による構造タイプおよび特殊タイプの索引方法を指定します。索引拡張を作成するには、 CREATE INDEX EXTENSION ステートメントを発行する必要があります。 CREATE INDEX EXTENSION ステートメントは、構造タイプまたは特殊タイプの値を索引キーに変換するための外部表関数を指定して、DB2 がこれらの索引キーを検索してパフォーマンスを最適化する方法を定義します。

これらの表関数の作成方法については、 409ページの『第15章 ユーザー定義関数 (UDF) とメソッドの作成』 を参照してください。索引拡張を使用して、構造タイプおよび特殊タイプを使用するアプリケーションのパフォーマンスを向上させるための方法についての詳細は、 *管理の手引き* を参照してください。 CREATE INDEX EXTENSION ステートメントの詳細については、 *SQL 解説書* を参照してください

制約　制約とは、ユーザーが定義してデータベースが実施する規則のことです。以下のような 4 つの制約があります。

固有　表のキーが固有なものであるようにします。固有キーを構成する列に対して行われるどんな変更に対しても、それが固有であるかどうかの検査が行われます。

参照保全
挿入、更新、および削除操作に対して参照制約を強要します。これは、すべての外部キーの値が有効であることを示すデータベースの状態です。

表検査　表が作成または更新された際に、変更されたデータが指定された条件に違反していないかどうかを検査します。

トリガー
トリガーはある表に関連した SQL ステートメントから成り、その表に対してデータ変更操作が行われる際に自動的に活動化されます。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。

固有制約、参照保全、および表検査制約についての詳細は、 *管理の手引き* を参照してください。トリガーについての詳細は、 499ページの『第16章 活動状態の DBMS でのトリガーの使用』を参照してください。

一般のアプリケーションでのオブジェクト指向機能の使用

DB2 のオブジェクト指向機能間には、重要な協同性があります。 DB2 オブジェクト指向のメカニズムは、オブジェクト指向のアプリケーションのサポートに限らず、他の目的にも使用することができます。 C++ などの一般的なオブジェクト指向プログラミング言語があらゆる種類の非オブジェクト指向アプリケーションを実行するために使用されるように、 DB2 に備えられているオブジェクト指向のメカニズムも、あらゆる種類の非オブジェクト指向アプリケーションをサポートするのに非常に役立ちます。 DB2 のオブジェクト関連機能は、あらゆるデータベース・アプリケーションをモデル化するために使用できる汎用メカニズムです。したがって、こうした DB2 オブジェクト拡張は、一般的なアプリケーションのサポートを向上する他に、非一般の、すなわちオブジェクト指向のアプリケーションにも拡張的なサポートを行います。

ユーザー定義の特殊タイプ

特殊タイプは既存の組み込みタイプに基づいています。たとえば、 USDollar および Canadian_Dollar などの様々な通貨を表すための特殊タイプがあるとしましょう。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・タイプとして表示されます。

強力タイピング

同一の組み込みタイプに基づいて異なった特殊タイプを持てますが、特殊タイプは強力タイピング の特性を持ちます。強力タイピング特性では、そのようなタイプのインスタンスを、そのタイプの別のインスタンス以外とは直接比較できないようにします。これにより、変換処理を行わずに USDollar と Canadian_Dollar を直接加算するなどの、セマンティクス的にはありえないような操作を阻止します。特殊タイプのインスタンスでどんな操作が行えるかは、ユーザーが定義します。

タイプの振る舞い

USDollar または Canadian_Dollar のインスタンスでどんな操作が許可されるかをどのように定義できるでしょうか。ユーザー定義の関数を使用して、特殊タイプで可能な振る舞いを定義します。関数を登録することによって USDollar のインスタンスを加算するような簡単な操作を行えます。この関数は、USDollar を入力として受け取る簡単な組み込み加算操作です。このような関数を定義するのにアプリケーションを作成する必要はありません。

しかし、USDollar タイプを入力として受け取り、それを Canadian_Dollar タイプに変換するもっと複雑な関数を作成したいと思われるかもしれません。ユーザー定義の関数についての詳細は、 389ページの『第14章 ユーザー定義関数 (UDF) およびメソッド』を参照してください。

制約を使用すると、保全性の規則を実装できます。

ラージ・オブジェクト

特殊タイプでモデル化するオブジェクトは非常に大きいものかもしれません。しかし、DB2 には非常に大きなオブジェクトを保管および操作するための新しい組み込みタイプがあります。特殊タイプは以下の中の 1 つのラージ・オブジェクト (LOB) データ・タイプに基づいています。これらを音声またはビデオなどに使用できるかもしれません。

オブジェクトの振る舞いの定義: ユーザー定義のルーチン

オブジェクトの振る舞いを定義するには、以下のようなユーザー定義関数およびメソッドを使用できます。

ユーザー定義関数

UDF とはユーザーが定義できる関数で、組み込み関数や演算子と同様に SQL 照会でのオブジェクトの操作をサポートします。 (UDF を使用して、ユーザー定義タイプだけではなくすべてのタイプの列値を操作できます。) ですから、

ユーザー定義タイプのインスタンス (特殊または構造化されたもの) は、表の列または行の中に保管され、SQL 照会によって UDF で操作されます。たとえば、LENGTH という特殊タイプのインスタンスと WIDTH という特殊タイプのインスタンスを受け取って、面積を計算してから照会に戻す、以下のような AREA という関数を定義できます。

```
SELECT ID, area(length, width) AS area
FROM Property
WHERE area > 10000;
```

メソッド

UDF と同様、メソッドはオブジェクトの振る舞いを定義しますが、以下のような点で関数とは異なります。

- メソッドは特定のユーザー定義構造タイプに密接に関連しており、ユーザー定義タイプと同じスキーマに保管されています。
- メソッドは列内の値として保管されているユーザー定義タイプで呼び出すことができますし、参照解除演算子 (->) を使用して構造タイプへの範囲参照を行うことができます。
- メソッドは、関数を呼び出すのとは別の SQL 構文を使用して呼び出されます。
- DB2 は、メソッドへの修飾されていない参照を、そのメソッドが呼び出されたタイプで解決します。メソッドを呼び出したタイプがそのメソッドを定義していないなら、 DB2 は、そのメソッドを呼び出したタイプのスーパータイプのメソッドを呼び出すことによってメソッドの参照を解決しようとします。

列内に保管されている構造タイプのメソッドを呼び出すには、構造タイプの名前 (または、構造タイプに分解できる式) を呼び出しに含め、メソッド呼び出し演算子 (..) を続け、さらにその後ろにメソッドの名前を続けます。構造タイプの範囲参照のメソッドを呼び出すには、参照解除演算子 (->) を使用して構造タイプへの参照を含め、メソッド呼び出し演算子を続けます。それから、メソッドの名前を指定します。

DB2 のオブジェクト・リレーション機能についての詳細は、以下を参照してください。

- 305ページの『第12章 複合オブジェクトの処理: ユーザー定義の構造型』
- 295ページの『第11章 ユーザー定義特殊タイプ』
- 365ページの『第13章 ラージ・オブジェクト (LOB) の使用』
- 389ページの『第14章 ユーザー定義関数 (UDF) およびメソッド』
- 409ページの『第15章 ユーザー定義関数 (UDF) とメソッドの作成』
- 499ページの『第16章 活動状態の DBMS でのトリガーの使用』

第11章 ユーザー定義特殊タイプ

特殊タイプを使用する理由 295
特殊タイプの定義 296
修飾されない特殊タイプの分析 296
CREATE DISTINCT TYPE の使用例 . . . 297
例: 通貨 297
例: ジョブ・アプリケーション 297
特殊タイプを使用した、表の定義 297
例: 売上 297
例: 応募の書式 298
特殊タイプの操作 298
特殊タイプの操作例 299
例: 特殊タイプと定数の比較 299
例: 特殊タイプ間のキャスト 300
例: 特殊タイプを使用した比較 301
例: 特殊タイプを使用したソース派生 UDF 302
例: 特殊タイプを使用した割り当て . . . 302
例: 動的 SQL での割り当て 302
例: 異なる特殊タイプを使用した割り当て 303
例: UNION 形式での特殊タイプの使用 304

特殊タイプを使用する理由

ユーザー定義特殊タイプ と呼ばれる、作成済みのデータ・タイプは、ご使用の DB2 アプリケーションで使用することができます。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。 UDF の詳細については、389ページの『第14章 ユーザー定義関数 (UDF) およびメソッド』を参照してください。

3. **一貫した振る舞い。**

   強力タイピングにより、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現に依存しないので、さまざまなことを自在に実行できます。

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、組み込みデータ・タイプと同じ方法で内部に表されるので、組み込みデータ・タイプが組み込み関数、比較演算子、索引などをインプリメントするのに使用するのと同じ効果的なコードを共用します。

© Copyright IBM Corp. 1993, 2001

特殊タイプの定義

特殊タイプは、表、索引、UDF などの他のオブジェクトと同様に、 CREATE ステートメントで定義しなければなりません。

CREATE DISTINCT TYPE ステートメントを使用して新規の特殊タイプを定義します。ステートメント構文およびそのオプションすべての詳細については、 *SQL 解説書* で説明されています。

CREATE DISTINCT TYPE ステートメントに関して、以下のことに注意してください。

1. 新規の特殊タイプの名前は、修飾された、または修飾されない名前のいずれでもかまいません。それがステートメントの許可 ID と異なるスキーマで修飾される場合は、データベース上で DBADM 権限を持たなければなりません。
2. 特殊タイプのソース・タイプは、DB2 が特殊タイプの内部表現に使用するタイプです。したがって、組み込みデータ・タイプ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。
3. WITH COMPARISONS 文節は、特殊タイプのインスタンスの比較演算をサポートする関数を DB2 で生成するよう DB2 に指示するために使用します。この文節は、ソース・タイプで比較演算がサポートされる場合 (たとえば INTEGER および DATE) は必要で、サポートされない場合 (たとえば LONG VARCHAR および BLOB) は禁止されています。

注: DB2 は特殊タイプ定義の一部として、以下のものに対し常にキャスト機能を生成します。

- ソース・タイプの標準名を使用した、特殊タイプからソース・タイプへのキャスト。たとえば、FLOAT に基づいて特殊タイプを作成した場合、DOUBLE と呼ばれるキャスト関数が作成されます。
- ソース・タイプから特殊タイプへのキャスト。特殊タイプへの追加のキャストが生成される条件の説明については、 *SQL 解説書* を参照してください。

これらの機能は、照会中の特殊タイプを操作するために重要です。

修飾されない特殊タイプの分析

関数パスは、修飾されないタイプ名や関数の参照の分析に使用されます。ただし、タイプ名または関数を以下のように処理した場合を除きます。

- 作成
- 除去
- 注釈

修飾されない関数参照の分析方法の詳細については、 403ページの『修飾された関数参照の使用法』を参照してください。

CREATE DISTINCT TYPE の使用例

以下に CREATE DISTINCT TYPE の使用例を示します。

- 例: 通貨
- 例: ジョブ・アプリケーション

例: 通貨

たとえば、異なる通貨を扱う必要のあるアプリケーションを作成中で、照会の際に、これらの通貨を直接比較または操作するのを DB2 が認めないようにしたいとします。なお、異なる通貨の値を比較する場合には、必ず変換が必要です。それで、必要な数の特殊タイプを定義します。特殊タイプは、表現したい通貨ごとに 1 つずつ必要です。

```
CREATE DISTINCT TYPE US_DOLLAR AS DECIMAL (9,2) WITH COMPARISONS
CREATE DISTINCT TYPE CANADIAN_DOLLAR AS DECIMAL (9,2) WITH COMPARISONS
CREATE DISTINCT TYPE EURO AS DECIMAL (9,2) WITH COMPARISONS
```

なお、比較演算子は、DECIMAL (9,2) 上でサポートされるので、 WITH COMPARISONS を指定しなければなりません。

例: ジョブ・アプリケーション

たとえば、会社への応募者が記入した書式を DB2 の表中に保持したいので、そのような書式から情報を抽出する関数を使用するとします。これらの関数は標準文字ストリングに適用できないため (戻すことになっている情報を確実に検出できるわけでないため)、記入された書式を表示する特殊タイプを定義します。

```
CREATE DISTINCT TYPE PERSONAL.APPLICATION_FORM AS CLOB(32K)
```

DB2 は CLOB の比較をサポートしないので、 WITH COMPARISONS という文節は指定しません。 DBADM 権限を持っていたために自分の許可 ID と異なるスキーマ名を指定したので、応募者の書式を扱うすべての特殊タイプおよび UDF を同じスキーマに保持したいとします。

特殊タイプを使用した、表の定義

特殊タイプの定義が済むと、タイプが特殊タイプである列を持つ表を定義できます。以下に CREATE TABLE の使用例を示します。

- 例: 売上
- 例: 応募の書式

例: 売上

たとえば、以下のようにさまざまな国における会社の売上を保持する表を定義したいとします。

```
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)
```

上記の例の特殊タイプは、 297ページの『例: 通貨』中の CREATE DISTINCT TYPE ステートメントと同じものを使用して作成しています。なお、ここでは検査の制約を使用しています。検査制約の詳細については、 *SQL 解説書* を参照してください。

例: 応募の書式

たとえば、以下のように応募者が記入した書式を保持する表を定義する必要があるとします。

```
CREATE TABLE APPLICATIONS
  (ID               SYSIBM.INTEGER,
   NAME             VARCHAR (30),
   APPLICATION_DATE SYSIBM.DATE,
   FORM             PERSONAL.APPLICATION_FORM)
```

特殊タイプ名の修飾子が自分の許可 ID と異なるので、それは完全に修飾されたことになり、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・タイプの修飾を省略できます。たとえば、`SET CURRENT FUNCTION PATH = cheryl` を実行でき、現行関数パスの特殊レジスターの値は `"CHERYL"` で `"SYSIBM"` は含まれません。ここで、CHERYL.INTEGER タイプが定義されていない場合でも、 COL1 のタイプが SYSIBM.INTEGER であると常に SYSIBM が考慮されるので、 `CREATE TABLE FOO(COL1 INTEGER)` というステートメントは正常に実行されます。

ただし、ご希望であれば、組み込みデータ・タイプを完全に修飾することもできます。現行関数パスの使用に関する詳細については、 *SQL 解説書* を参照してください。

特殊タイプの操作

特殊タイプに関連する最も重要な概念の 1 つに*強力タイピング* があります。強力タイピングは、特殊タイプで定義された関数および演算子のみをそのインスタンスに適用できることを保証します。

強力タイピングは、特殊タイプのインスタンスが正しいことを保証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。

強力タイピングの結果、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。

特殊タイプの操作例

特殊タイプの操作例として以下のものを挙げます。

- 例: 特殊タイプと定数の比較
- 例: 特殊タイプ間のキャスト
- 例: 特殊タイプを使用した比較
- 例: 特殊タイプを使用したソース派生 UDF
- 例: 特殊タイプを使用した割り当て
- 例: 動的 SQL での割り当て
- 例: 異なる特殊タイプを使用した割り当て
- 例: UNION 形式での特殊タイプの使用

例: 特殊タイプと定数の比較

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
```

例: 特殊タイプ間のキャスト

たとえば、カナダ・ドルを米ドルに変換する UDF を定義したいとします。このとき、DB2 の外部で管理されているファイルから現行の為替相場を求めることができるとします。そこで、カナダ・ドルの値を取得し、為替相場のファイルにアクセスし、米ドルの額に対応した値を戻すような UDF を定義します。

一見、このような UDF を定義するのは容易に見えるかもしれません。しかし、C は DECIMAL の値をサポートしません。異なる通貨を表す特殊タイプは、DECIMAL と定義しました。 10 進精度を失わずに DECIMAL の値を表すことができる、 C によって提供される唯一のデータ・タイプは DOUBLE なので、 UDF は、DOUBLE の値を受け取り、戻す必要があります。このため、UDF は以下のように定義しなければなりません。

```
CREATE FUNCTION CDN_TO_US_DOUBLE(DOUBLE) RETURNS DOUBLE
  EXTERNAL NAME '/u/finance/funcdir/currencies!cdn2us'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  NOT DETERMINISTIC
  NO EXTERNAL ACTION
  FENCED
```

UDF が 2 回呼び出される間にカナダ・ドルおよび米ドルの為替相場が変化する場合があるので、 UDF は NOT DETERMINISTIC として宣言します。

ここで問題なのは、どのようにしてカナダ・ドルを UDF に渡して、それから米ドルを受け取るかということです。カナダ・ドルは、DECIMAL の値にキャストしなければなりません。 DECIMAL の値は DOUBLE にキャストしなければなりません。また、戻された DOUBLE の値を DECIMAL に、DECIMAL の値を米ドルにキャストしなければなりません。

このようなキャストは、ソース派生 UDF を定義する際に DB2 により自動的に実行され、その UDF のパラメーターおよび戻りタイプはソース関数のものと正確には一致しません。そのため、ソース派生 UDF を 2 つ定義する必要があります。最初のものは、DECIMAL 表記に DOUBLE の値を渡します。 2 番目のものは、特殊タイプに DECIMAL の値を渡します。すなわち、以下のものを定義します。

```
CREATE FUNCTION CDN_TO_US_DEC (DECIMAL(9,2)) RETURNS DECIMAL(9,2)
  SOURCE CDN_TO_US_DOUBLE (DOUBLE)

CREATE FUNCTION US_DOLLAR (CANADIAN_DOLLAR) RETURNS US_DOLLAR
  SOURCE CDN_TO_US_DEC (DECIMAL())
```

US_DOLLAR(C1) (C1 はタイプがカナダ・ドルの列) のような US_DOLLAR 関数の呼び出しは、この以下の呼び出しと同じように作用します。

```
US_DOLLAR (DECIMAL(CDN_TO_US_DOUBLE (DOUBLE (DECIMAL (C1)))))
```

つまりこれは (カナダ・ドルの) C1 が 10 進数にキャストされ、次に倍精度値にキャストされてから CDN_TO_US_DOUBLE 関数に渡されるということです。この関数は、為替相場のファイルにアクセスし、倍精度値 (米ドルで額を表す) を戻します。この値は、まず 10 進数に、その後米ドルにキャストされます。

ユーロを米ドルに変換する関数は、上記の例と同様になります。

```
CREATE FUNCTION EURO_TO_US_DOUBL(DOUBLE)
  RETURNS DOUBLE
  EXTERNAL NAME '/u/finance/funcdir/currencies!euro2us'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  NOT DETERMINISTIC
  NO EXTERNAL ACTION
  FENCED

CREATE FUNCTION EURO_TO_US_DEC (DECIMAL(9,2))
  RETURNS DECIMAL(9,2)
  SOURCE EURO_TO_US_DOUBL (DOUBLE)

CREATE FUNCTION US_DOLLAR(EURO) RETURNS US_DOLLAR
  SOURCE EURO_TO_US_DEC (DECIMAL())
```

例: 特殊タイプを使用した比較

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。 各総額を貨幣上比較できないので、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ということです。

例: 特殊タイプを使用したソース派生 UDF

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。

例: 特殊タイプを使用した割り当て

新規の応募者が記入した書式をデータベースに保管する場合を考えてみましょう。ここで、記入された書式を表すの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るものとします。

```
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ので、キャスト機能を明示的に呼び出してその文字ストリングを特殊タイプの `personal.application_form` に変換することはしません。

例: 動的 SQL での割り当て

『例: 特殊タイプを使用した割り当て』で示されたのと同じステートメントを動的 SQL で使用したい場合は、次のようにパラメーター・マーカーを使用します。

```
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。

例: 異なる特殊タイプを使用した割り当て

たとえば、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、302ページの『例: 特殊タイプを使用したソース派生 UDF』中のユーロのソース派生 UDF と同じ 2 つの 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、各国における各製品についての 1 年間の総売上米ドルで、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
```

異なる特殊タイプ同士を直接互いに割り当てることはできないので、カナダ・ドルおよびユーロの金額を米ドルに明示的にキャストします。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。

例: UNION 形式での特殊タイプの使用

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視点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、同じ特殊タイプ同士でないと一緒に比較できないので、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。なお、キャスト指定では、特殊タイプおよびそのソース・タイプ間のキャストしかできないので、機能表記法を使用してキャストしなければなりません。

第12章 複合オブジェクトの処理: ユーザー定義の構造型

構造型の概説 306
構造型階層の作成. 307
参照タイプとその表示タイプ 309
参照タイプのキャストと比較 310
他のシステムによって生成されたルーチン 310
タイプの振る舞いの定義 312
オブジェクトのタイプ付き表への保管 . . 313
タイプ付き表の中のオブジェクト間の関係の定義 314
オブジェクトの列への保管. 315
構造型の付加的な特性 316
タイプ付き表での構造型の使用 318
タイプ付き表の作成 318
表のタイプの定義. 318
オブジェクト ID の命名 318
表階層での位置の指定 319
SELECT 特権は継承されたものであることの表示 319
列オプションの定義. 320
参照列の効力範囲の定義 320
タイプ付き表への挿入 320
参照タイプの使用. 322
参照タイプの比較. 323
セマンティクス関係を定義するための参照の使用 323
参照保全と効力範囲が指定された参照との相違. 325
タイプ付き視点の作成 325
ユーザー定義タイプ (UDT) またはタイプ・マッピングの除去 327
視点の更新または除去 328
タイプ付き表の照会 328
参照を逆参照する照会 329
DEREF 組み込み関数 330
タイプに関連したその他の組み込み関数 330
付加的な照会仕様技法 331
ONLY を使用して特定のタイプのオブジェクトを戻す 331
タイプ述部を使用して戻されるタイプを制限する 331
OUTER を使用してすべての可能性のある属性を戻す 332
その他のヒント 333
システム生成オブジェクト ID の定義 333
オブジェクト ID 列に対する制約の作成 334
列タイプとしての構造型の作成と使用 . . . 335
構造型インスタンスの列への挿入 . . . 335
| 構造化タイプ属性を列に挿入する . . . 336
構造型列を持つ表の定義 336
構造型属性を持つタイプの定義 336
構造型値が入っている行の挿入 337
構造型値の検索と変更 338
属性の検索 338
サブタイプの属性へのアクセス . . . 339
属性の変更 339
タイプについての情報を戻す 340
transform のタイプとの関連付け 340
transform グループの命名についての推奨事項 341
transform グループを指定する必要のある場合 342
外部ルーチン用の transform グループの指定 343
動的 SQL 用の transform グループの設定 343
静的 SQL 用の transform グループの設定 343
ホスト言語プログラムへのマッピングの作成: transform 関数. 344
オブジェクトの外部ルーチンとの交換: function transform 345
transform 関数の要約. 354
サブタイプ・データの DB2 からの検索 (バインドアウト) 355
サブタイプ・データを DB2 に戻す (バインドイン). 359
構造型ホスト変数の処理 363
構造型ホスト変数の宣言 363

© Copyright IBM Corp. 1993, 2001

構造型の記述 363

構造型の概説

構造型は、*属性* で構成される明確な構造を持つオブジェクトをモデル化するのに役立ちます。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、幾何学形状は、デカルト座標のリストを属性として持っています。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 があります。

タイプを作成するには、タイプの名前、属性名、およびデータ・タイプを指定する必要があります。また、このタイプの参照タイプをシステム内で表現する方法を指定することもできます。 `BusinessUnit_t` タイプを作成する SQL は、次のとおりです。

```
CREATE TYPE BusinessUnit_t AS
  (Name VARCHAR(20),
  Headcount INT)
  REF USING INT
  MODE DB2SQL;
```

AS 文節では、タイプに関連した属性定義を指定します。 `BusinessUnit_t` は、 `Name` と `Headcount` という 2 つの属性を持つタイプです。構造型を作成するには、 CREATE TYPE ステートメントに MODE DB2SQL 文節を入れます。 REF USING 文節の詳細については、 309ページの『参照タイプとその表示タイプ』を参照してください。

構造型は、従来のリレーショナル・データ・タイプの機能を拡張しています。拡張された主な 2 つの機能は、継承というプロパティーと、構造型のインスタンスを表の中に行としてまたは列の中に値として保管する機能です。次のセクションで、これらの機能を概説します。

継承 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、構造型は、構造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。たとえば、構造型 `Person_t` に、 `Name`、`Age`、および `Address` の属性があるとします。 `Person_t` のサブタイプは、 `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` とすることができます。

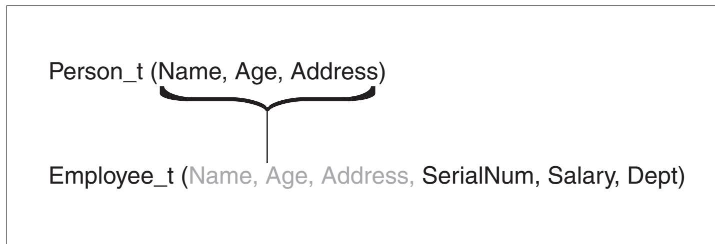


図 7. 構造型 Employee_t は Person_t を継承している

構造型のインスタンスの保管

構造型インスタンスは、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。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・タイプは、指示されている構造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、タイプ付き表といいます。

- 列の中に値として。オブジェクトを表の列に保管するには、列を構造型として定義します。次のステートメントは、 Address_t 構造型に属する構造型 Address を持つ Properties 表を作成します。

```
CREATE TABLE Properties
   (ParcelNum INT,
   Photo BLOB(2K),
   Address Address_t)
   ...
```

構造型階層の作成

別の構造型の下に 構造型を作成することができます。この場合、新しく作成された構造型は、元の構造型のサブタイプ になります。元の構造型は、スーパータイプ です。サブタイプは、スーパータイプのすべての属性を継承し、さらに独自の追加の属性を持つこともできます。

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。本書の

例に使用されているタイプ階層を図8 に示します。 Address_t のタイプ階層は、335ページの『構造型インスタンスの列への挿入』で定義されています。

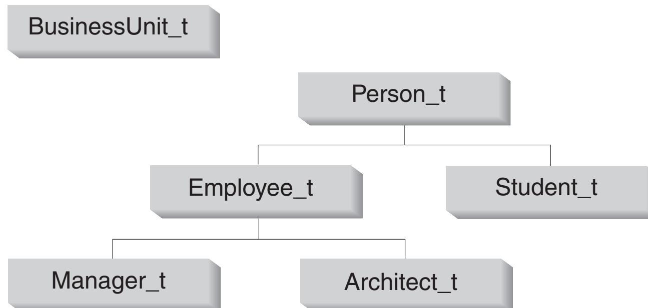


図 8. タイプ階層 (BusinessUnit_t と Person_t)

図8 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、タイプ階層全体で保持されるということです。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。

306ページの『構造型の概説』で定義されているタイプ BusinessUnit_t は、サブタイプを持っていません。 335ページの『構造型インスタンスの列への挿入』で定義されている Address_t は、 Germany_addr_t、Brazil_addr_t、および US_addr_t というサブタイプを持っています。

Person_t の CREATE TYPE ステートメントは、 Person_t が INSTANTIABLE であると宣言しています。 INSTANTIABLE または NOT INSTANTIABLE 文節を使用した構造型の宣言については、 316ページの『構造型の付加的な特性』を参照してください。

次の SQL ステートメントは、Person_t タイプ階層を作成します。

```
CREATE TYPE Person_t AS
   (Name VARCHAR(20),
   Age INT,
   Address Address_t)
   INSTANTIABLE
   REF USING VARCHAR(13) FOR BIT DATA
   MODE DB2SQL;

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。

注: Person_t タイプから作成されるタイプ付き表には、構造型 Address_t の列 Address が入っています。すべての構造型の列の場合と同様に、この列の構造型のための transform 関数を定義する必要があります。 transform 関数の定義方法の詳細については、 344ページの『ホスト言語プログラムへのマッピングの作成: transform 関数』を参照してください。

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。

参照タイプとその表示タイプ

DB2 は、開発者が作成するすべての構造型について、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、参照先の構造型のことを*参照先タイプ* といいます。タイプ付き表では、318ページの『タイプ付き表での構造型の使用』で説明されているように、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、参照タイプをタイプ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、タイプ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、行の参照をタイプ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。参照の使用法の詳細については、322ページの『参照タイプの使用』を参照してください。タイプ付き表の詳細については、313ページの『オブジェクトのタイプ付き表への保管』を参照してください。

参照は大文字で入力します。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・タイプを使用します。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。

参照タイプのキャストと比較

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、 *SQL 解説書* で説明されている CAST WITH 文節をオプションで指定することができます。この文節を指定すると、これら 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、307ページの『構造型階層の作成』の CREATE TYPE Person_t ステートメントは、次の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)
```

タイプ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。参照タイプ用の比較演算子の詳細については、 *SQL 解説書* を参照してください。

他のシステムによって生成されたルーチン

開発者が作成するすべての構造型について、 DB2 は、構造型の値を構成、監視、変更するのに使用できる一連の関数を暗黙的に作成します。つまり、たとえば、タイプ `Person_t` のために、このタイプの作成時に DB2 は次の関数とメソッドを作成するということです。

constructor 関数

タイプと同じ名前の関数が作成されます。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ヌル設定になったタイプのインスタンスを戻します。 Person_t のために作成される関数は、たとえば、次のステートメントを実行した場合と同じようになります。

```
CREATE FUNCTION Person_t ( ) RETURNS Person_t
```

サブタイプ Manager_t については、次のステートメントが実行された場合と同じように constructor 関数が作成されます。

```
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。データのタイプ付き表への挿入方法の詳細については、 337ページの『構造型値が入っている行の挿入』を参照してください。

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タイプのインスタンスのことを、そのメソッドの対象 インスタンスといいます。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。

age のために DB2 が作成する mutator メソッドは、たとえば、次のステートメントが実行された場合と同様になります。

```
ALTER TYPE Person_t
   ADD METHOD AGE(int)
   RETURNS Person_t;
```

オブジェクトの変更の詳細については、 338ページの『構造型値の検索と変更』を参照してください。

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、予期タイプまたは予期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 メソッドは、たとえば、DB2 が次のステートメントを発行する場合と同様になります。

```
ALTER TYPE Person_t
   ADD METHOD AGE()
   RETURNS INTEGER;
```

observer メソッドの使用法の詳細については、 338ページの『構造型値の検索と変更』を参照してください。

構造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。メソッドの呼び出しの詳細については、 *SQL 解説書* を参照してください。

タイプの振る舞いの定義

構造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)
   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。

注: DB2 は現在、動的ディスパッチングをサポートしていません。これは、あるタイプのためにあるメソッドを宣言したなら、そのメソッドを同じ数のパラメーターを使用して、サブタイプのために再定義することはできないということです。この回避方法としては、TYPE 述部を使用して動的タイプを判別してから、 TREAT AS 文節を使用してそれぞれの動的タイプのために異なるメソッドを呼び出すという方法があります。サブタイプを処理する transform 関数の例については、 355ページの『サブタイプ・データの DB2 からの検索 (バインドアウト)』を参照してください。

メソッドの登録、書き込み、および呼び出しの詳細については、 389ページの『第14章 ユーザー定義関数 (UDF) およびメソッド』および 409ページの『第15章 ユーザー定義関数 (UDF) とメソッドの作成』を参照してください。

オブジェクトのタイプ付き表への保管

構造型のインスタンスは、タイプ付き表の中に行として保管する (タイプのそれぞれの属性は別々の列に保管される) か、列の中にオブジェクトとして保管する (タイプの属性はすべて 1 つの列に保管される) ことができます。タイプ付き表は識別のための属性を持っています。つまり、別の表は参照を使用して、インスタンスの属性にアクセスできるということです。他の表からインスタンスを参照する必要がある場合は、タイプ付き表を使用しなければなりません。他の表がオブジェクトを識別する必要がない場合は、オブジェクトを列に保管することについて考慮してください。

オブジェクトが表の中に行として保管されると、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。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。次の例は、Person というインスタンスを保管するための CREATE TABLE ステートメントです。

```
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 10. Person タイプ付き表

Oid	Name	Age	Address
a	Andrew	29	

プログラムは、タイプ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 11. 更新後の Person タイプ付き表

Oid	Name	Age	Address
a	Andrew	30	

Employee_t という Person_t のサブタイプがあるので、 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、別の表に保管する必要がありま

す。この表のことを*副表* といいます。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 12. Employee タイプ副表

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';
```

これら 2 つの表に関する興味深い点は、 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できるということです。この機能のことを*代用性* といいます。これについては、316ページの『構造型の付加的な特性』で説明されています。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 13. 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	(ヌル)
s	Susan	39	(ヌル)

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;
```

代用性の詳細については、316ページの『構造型の付加的な特性』を参照してください。

タイプ付き表の中のオブジェクト間の関係の定義

あるタイプ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関係を定義することができます。同じタイプ付き表の中にあるオブジェクト間の関係を定

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タイプ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9 で図解されています。

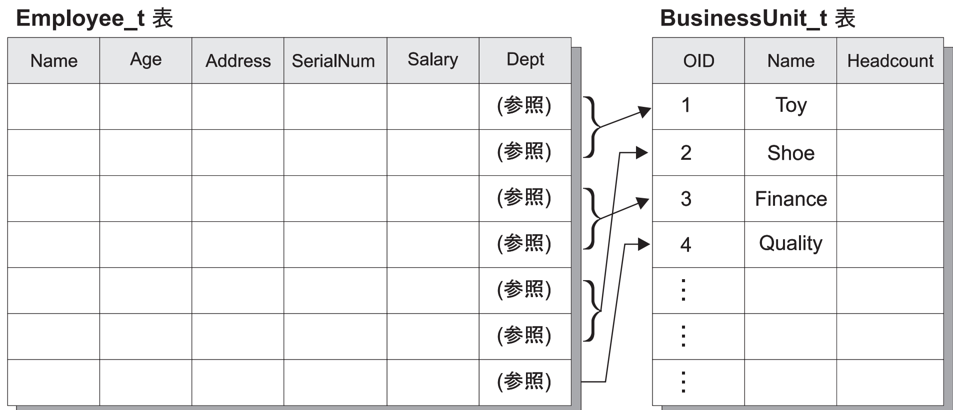


図 9. Employee_t から BusinessUnit_t に対する構造型参照

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、複数の表の間の関係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、関係を逆参照して、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* 演算子といい、 -> 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;
```

タイプ付き表に対する照会の作成方法の詳細については、 328ページの『タイプ付き表の照会』を参照してください。

オブジェクトの列への保管

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・タイプ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)

をタイプ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10 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。

Person

Name (VARCHAR)	Age (INT)	Address (Address_t)			
		Street	Number	City	State

図 10. 構造型としての 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。 Address_t タイプ階層は、335ページの『構造型インスタンスの列への挿入』で定義されています。

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';
```

上記の例では、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ています。 WHERE 文節は、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という 2 つの述部によって、 UPDATE ステートメントの操作を制約しています。

構造型の付加的な特性

代用性 タイプ付き表に 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、Person_t からタイプ付き表を作成してその表のすべての行を選択すると、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 情報も受け取ります。代用性の特性は、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。

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。

構造型をパラメーターとして関数に渡す場合、または構造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。スカラー関数が 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、サブタイプの内の 1 つ (US_addr_t など) のインスタンスを渡すことができます。表関数は、構造型の列を戻すことはできません。

列または表が、あるタイプで定義されていても、そこに他の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のが重要な場合があります。列、行、または関数パラメーターの中の構造型の定義のことを*静的タイプ* といいます。構造型インスタンスの実際のタイプのことを*動的タイプ* といいます。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 330ページの『タイプに関連したその他の組み込み関数』で説明されている TYPE_NAME、TYPE_SCHEMA、および TYPE_ID 組み込み関数を使用します。

インスタンス生成の可否

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、タイプ階層においてさらに正確にする余地のあるモデルとしての役割を果たします。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視点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。

タイプ付き表での構造型の使用

タイプ付き表の作成

タイプ付き表は、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、タイプ付き表を作成することができます。構造型の階層を基にしたタイプ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データベース表に保管するには、対応する表階層を作成する必要があります。

次の例は、315ページの図9 に示されているタイプ階層に基づく表階層の作成方法を示しています。

BusinessUnit タイプ付き表を作成する 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タイプ付き表は、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持することになります。この表は、構造型 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

タイプ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、すべてのタイプ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 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。オブジェクト ID を自動的に生成するよう DB2 を構成する方法の詳細については、 333ページの『システム生成オブジェクト ID の定義』を参照してください。

表階層での位置の指定

Person タイプ付き表のタイプは 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、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、 331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』で説明されている 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 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 特権を保持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 特権が授与されます。副表の作成者が、SELECT 特権の授与者になります。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。 INHERIT SELECT PRIVILEGES の詳細については、 *SQL 解説書* を参照してください。

特権は、表階層のすべてのレベルで別々に授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持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 特権を授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、タイプ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 列にヌル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)
```

参照列の効力範囲の定義

WITH OPTIONS の別の使用法は、列の効力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*効力範囲* 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;
```

参照および参照の効力範囲の指定の詳細については、322ページの『参照タイプの使用』を参照してください。

タイプ付き表への挿入

前述の例で構造型を作成した後、また対応する表と副表を作成した後のデータベースの構造は、 321ページの図11 のようになります。

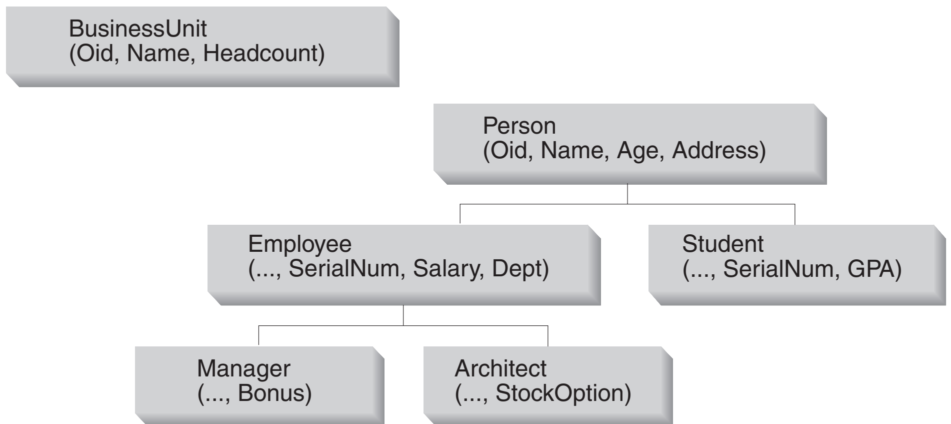


図 11. タイプ付き表の階層

階層を設定すれば、INSERT ステートメントを通常どおりに使用して表にデータを挿入することができます。唯一の相違点は、オブジェクト ID 列にデータを必ず挿入しなければならないこと、またそれぞれの表または副表の中のオブジェクトの追加の属性に任意でデータを挿入することです。オブジェクト ID 列のタイプは REF (大文字で入力される) であるので、構造型を作成した時にシステムによって生成された cast 関数を使用して、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。

```
INSERT INTO BusinessUnit (Oid, Name, Headcount)
   VALUES(BusinessUnit_t(1), 'Toy', 15);

INSERT INTO BusinessUnit (Oid, Name, Headcount)
   VALUES(BusinessUnit_t(2), 'Shoe', 10);

INSERT INTO Person (Oid, Name, Age)
   VALUES(Person_t('a'), 'Andrew', 20);

INSERT INTO Person (Oid, Name, Age)
   VALUES(Person_t('b'), 'Bob', 30);

INSERT INTO Person (Oid, Name, Age)
   VALUES(Person_t('c'), 'Cathy', 25);

INSERT INTO Employee (Oid, Name, Age, SerialNum, Salary, Dept)
   VALUES(Employee_t('d'), 'Dennis', 26, 105, 30000, BusinessUnit_t(1));

INSERT INTO Employee (Oid, Name, Age, SerialNum, Salary, Dept)
   VALUES(Employee_t('e'), 'Eva', 31, 83, 45000, BusinessUnit_t(2));

INSERT INTO Employee (Oid, Name, Age, SerialNum, Salary, Dept)
   VALUES(Employee_t('f'), 'Franky', 28, 214, 39000, BusinessUnit_t(2));

INSERT INTO Student (Oid, Name, Age, SerialNum, GPA)
   VALUES(Student_t('g'), 'Gordon', 19, '10245', 4.7);
```

```
INSERT INTO Student (Oid, Name, Age, SerialNum, GPA)
   VALUES(Student_t('h'), 'Helen', 20, '10357', 3.5);

INSERT INTO Manager (Oid, Name, Age, SerialNum, Salary, Dept, Bonus)
   VALUES(Manager_t('i'), 'Iris', 35, 251, 55000, BusinessUnit_t(1), 12000);

INSERT INTO Manager (Oid, Name, Age, SerialNum, Salary, Dept,
Bonus)
   VALUES(Manager_t('j'), 'Christina', 10, 317, 85000, BusinessUnit_t(1),
25000);

INSERT INTO Manager (Oid, Name, Age, SerialNum, Salary, Dept, Bonus)
   VALUES(Manager_t('k'), 'Ken', 55, 482, 105000, BusinessUnit_t(2), 48000);

INSERT INTO Architect (Oid, Name, Age, SerialNum, Salary, Dept, StockOption)
   VALUES(Architect_t('l'), 'Leo', 35, 661, 92000, BusinessUnit_t(2), 20000);
```

前述の例では、住所は挿入されていません。列に構造型を挿入する方法の詳細については、 337ページの『構造型値が入っている行の挿入』を参照してください。

タイプ付き表に行を挿入する時は、挿入されたそれぞれの行の最初の値は、表に挿入されるデータのオブジェクト ID になっていなければなりません。また、非タイプ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にデータを入れなければなりません。最後に、参照値が指定されているふさわしいタイプの式であればどれでも、参照属性を初期化するのに使用することができる点に注意してください。前述の例では、従業員の Dept 参照は、ふさわしくタイプ・キャストされた定数としての入力です。しかし、次の例で示されているように、副照会を使用して参照を入手することもできます。

```
INSERT INTO Architect (Oid, Name, Age, SerialNum, Salary, Dept, StockOption)
   VALUES(Architect_t('m'), 'Brian', 7, 882, 112000,
   (SELECT Oid FROM BusinessUnit WHERE name = 'Toy'), 30000);
```

参照タイプの使用

それぞれの構造型について、DB2 は対応する参照タイプをサポートしています。たとえば、Person_t タイプを作成すると、 DB2 は REF(Person_t) というタイプを自動的に作成します。 REF(Person_t) タイプの表示タイプ (および Person_t のすべてのサブタイプの REF タイプ)は、デフォルトで VARCHAR (16) FOR BIT DATA になっていますが、 CREATE TYPE ステートメントに REF USING 文節を使用して、別の表示タイプを選択することができます。この参照タイプは、構造型のインスタンスを保管するために作成するタイプ付き表のオブジェクト ID 列の基盤となります。たとえば、参照タイプのデフォルトの表示タイプを使用して、ルート・タイプ People_t を作成する場合は、関連した People 表のオブジェクト ID 列は、 VARCHAR(16) FOR BIT DATA に基づいたものになります。

参照タイプの比較

参照タイプは大文字で入力します。参照を定数と比較するには、定数をふさわしい参照タイプにキャストしてから、または参照タイプを基本タイプにキャストしてから比較を行います。特定のタイプ階層内の参照は、すべて同じ参照表示タイプになります。このようになっているので、S と T が共通のスーパータイプを持っていれば、 REF(S) と REF(T) を比較することができます。オブジェクト ID 列が固有でなければならないのは 1 つの表階層内だけなので、それぞれ異なる行を参照しているとしても、ある表階層内の REF(T) の値が、別の表階層内の REF(T) の値と等しいこともあり得ます。

セマンティクス関係を定義するための参照の使用

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関係を定義することができます。たとえば、BusinessUnit と Person 表階層では、それぞれの従業員の部門は、図12 に示されているように、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。

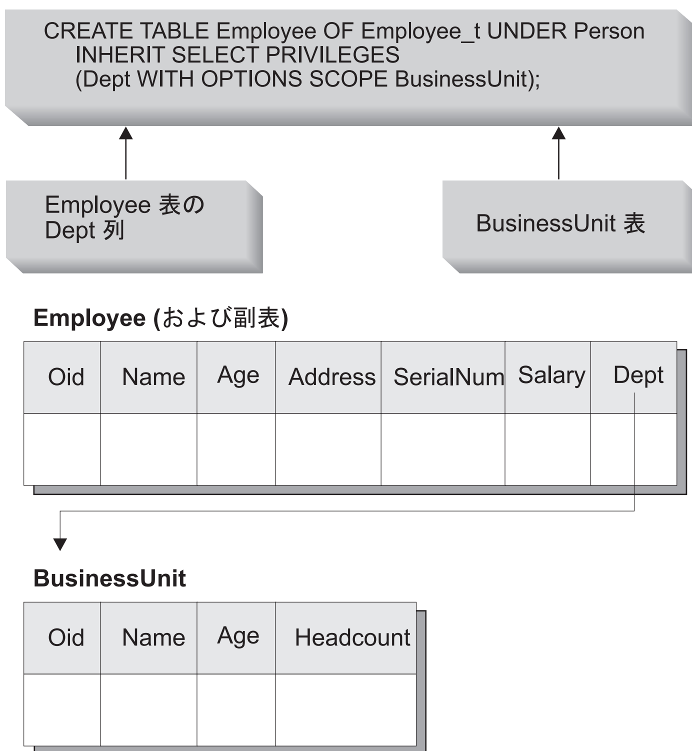


図 12. Dept 属性は BusinessUnit オブジェクトを参照している

自己参照関係: 同一のタイプ付き表の中のオブジェクトへの効力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 つのタイプ付き表と製造業社用に 1 つのタイプ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);
```

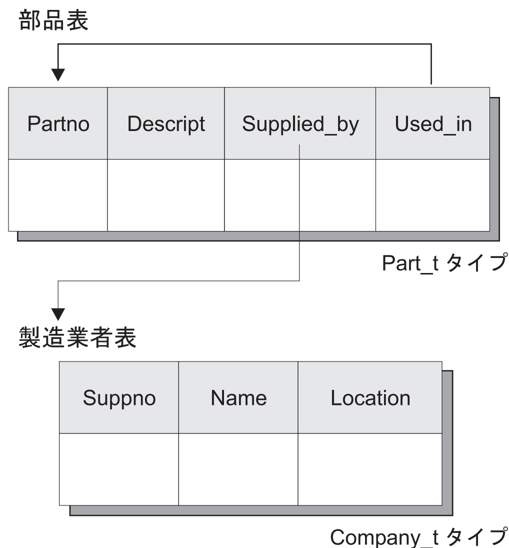


図 13. 自己参照の効力範囲の例

効力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、効力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。詳細については、329ページの『参照を逆参照する照会』を参照してください。

参照保全と効力範囲が指定された参照との相違

効力範囲が指定された参照は表の中のオブジェクト間の関係を定義しますが、これは参照保全関係とは異なります。効力範囲は、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。効力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。たとえば、Employee 表の中の Dept 列は、 BusinessUnit 表の中に存在していない BusinessUnit オブジェクト ID 列を参照することができます。これらの関係においてオブジェクトを必ず存在させるようにするには、表の間に参照制約を追加する必要があります。詳細については、334ページの『オブジェクト ID 列に対する制約の作成』を参照してください。

タイプ付き視点の作成

CREATE VIEW ステートメントを使用して、タイプ付き視点を作成することができます。たとえば、タイプ付き BusinessUnit 表の視点を作成するには、適切な属性を持つ構造型を定義してから、その構造型を使用してタイプ付き視点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、指示された構造型の属性を視点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 構造型を視点の列の基礎とします。

視点の 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・タイプ VARCHAR にキャストしてから、データ・タイプ VARCHAR からデータ・タイプ REF(VBusinessUnit_t) にキャストする必要があります。

MODE DB2SQL 文節は、タイプ付き視点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、タイプ付き CREATE TABLE ステートメントの REF IS...文節と同じです。これは、視点の最初の列である視点のオブジェクト ID 列の名前 (この例では VObjectID) を指定します。ルート・タイプ上でタイプ付き視点を作成する場合は、視点のオブジェクト ID 列を指定する必要があります。サブタイプ上でタイプ付き視点を作成する場合は、視点はオブジェクト 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 列の初期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 の後ろにある視点の本体 は、視点の内容を決定する 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、初期オブジェクト ID 列を含むタイプ付き視点の列タイプと互換性を保っている必要があります。

タイプ付き視点階層の作成を示すために、次の例は、いくつかの機密データが省略された、また事前に 318ページの『タイプ付き表の作成』で作成した Person 表階層のタイプ区別が除去された視点階層を定義します。

```
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視点階層を作成するのに必要な構造型を作成します。

上記の最初のタイプ付き CREATE VIEW ステートメントは、階層のルート視点 VPerson を作成し、 VBusinessUnit 視点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 視点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、視点 VPerson の下に副視点 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 文節は視点階層を設定します。スーパー視点として、同じスキーマの中に副視点を作成する必要があります。タイプ付き表と同様に、副視点はスーパー視点から列を継承します。 VEmployee 視点の中の行は、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、タイプ付き CREATE TABLE ステートメントを発行する時と同じです。

INHERIT SELECT PRIVILEGES 文節の詳細については、 319ページの『SELECT 特権は継承されたものであることの表示』を参照してください。タイプ付き視点定義の中の WITH OPTIONS 文節の効果も、タイプ付き表定義の中の 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、スーパー視点列を強制的に読み取り専用としてマークするので、これ以降の副視点定義は、読み取り専用となっている同じ列の式を指定できます。

視点に VEmployee 視点の 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、参照列に効力範囲を関連付ける必要があります。視点の参照列に効力範囲を指定しないで、基礎表または視点列に効力範囲を指定してある場合は、基礎列の効力範囲が視点の参照列に渡されます。 WITH OPTIONS 文節を使用して、視点の参照列に明示的に効力範囲を割り当てることができます。前述の例では、VEmployee 視点の Dept 列は、 VBusinessUnit 視点を効力範囲として受け取ります。基礎表または視点列に効力範囲が指定されていない場合で、視点定義で明示的に効力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効力範囲が割り当てられている場合は、参照列には効力範囲が指定されません。

SQL 解説書 には、タイプ付き視点の照会に対する制限に関連したいくつかの重要な規則が説明されているので、タイプ付き視点を作成して使用する前に注意深く読んでください。

ユーザー定義タイプ (UDT) またはタイプ・マッピングの除去

DROP ステートメントを使用して、ユーザー定義タイプまたはタイプ・マッピングを除去することができます。タイプ・マッピングの詳細については、593ページの『データ・タイプ・マッピングの処理』を参照してください。次の場合には UDT を除去することはできません。

- 既存の表または視点の列定義で使用されている場合。
- 既存のタイプ付き表またはタイプ付き視点 (構造型) のタイプとして使用されている場合。
- 別の構造型のスーパータイプとして使用されている場合。

デフォルトのタイプ・マッピングを除去することはできません。別のタイプ・マッピングを作成して上書きすることしかできないのです。

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ユーザー定義関数 (UDF) を除去しようとします。視点、トリガー、表検査制約、または別の UDF が従属している UDF を除去することはできません。 DB2 が従属 UDF を除去できない場合は、DB2 は UDT を除去しません。 UDT を除去すると、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 を除去しようとする場合は、関連した transform を除去することを考慮してください。 transform を除去す

るには、DROP TRANSFORM ステートメントを発行します。 DROP TRANSFORM ステートメントの完全な構文については、 *SQL 解説書* を参照してください。除去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除去することはできません。

視点の更新または除去

ALTER VIEW ステートメントは、効力範囲を追加するよう参照タイプ列を更新することによって、既存の視点を変更します。視点に対してこれ以外の変更を行うには、視点を除去してから再作成する必要があります。

視点を更新する際には、効力範囲がまだ定義されていない既存の参照タイプ列に効力範囲を追加する必要があります。さらに、その参照タイプはスーパー視点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・タイプは、 REF (タイプ付き表名またはタイプ付き視点名のタイプ) でなければなりません。

ALTER VIEW ステートメントの詳細については、 *SQL 解説書* を参照してください。

EMP_VIEW の除去方法の例は次のとおりです。

```
DROP VIEW EMP_VIEW;
```

除去された視点に従属するすべての視点は、作動不能になります。作動不能視点の詳細については、 *管理の手引き*『作動不能視点の回復』の節を参照してください。

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。詳細については、 *管理の手引き*『ステートメントの従属関係』の節を参照してください。

表階層の場合と同様に、次の例のように階層のルート視点を指定して、 1 つのステートメントの中の視点階層全体を除去することができます。

```
DROP VIEW HIERARCHY VPerson;
```

視点の除去と作成の詳細については、 *SQL 解説書* を参照してください。

タイプ付き表の照会

必要な SELECT 権限を持っている場合は、非タイプ付き表を照会するのと同じ方法で、タイプ付き表を照会することができます。照会は、SELECT およびそのすべての副表のターゲットの修飾行から、要求した列を戻します。たとえば、Person 表階層の中のデータに対する次の照会は、すべての人物の名前と年齢 (つまり、Person 表とその副表のすべての行) を戻します。列の 1 つが構造型列である場合の、同様の照会の作成方法の詳細については、 338ページの『構造型値の検索と変更』を参照してください。

```
SELECT Name, Age
FROM Person;
```

この照会の結果は、次のようになります。

```
NAME                 AGE
-------------------- -----------
Andrew               29
Bob                  30
Cathy                25
Dennis               26
Eva                  31
Franky               28
Gordon               19
Helen                20
Iris                 35
Christina            10
Ken                  55
Leo                  35
Brian                 7
Susan                39
```

参照を逆参照する照会

効力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 表に効力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、またはヌル値を使用できるところではヌル値を戻します。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、*逆参照演算子* (->) 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。

効力範囲が指定された参照式 -> *ターゲット・タイプ付き表の中の列*

次の例は、BusinessUnit 表から Name 列を獲得するために、逆参照演算子を使用しています。

```
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。 324ページの図13 の部品表について考慮してみましょう。次の照会は、部品の製造業社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 (*効力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 タイプ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;
```

前述の例は、Employee 表の Name の値を戻すために逆参照演算子を使用し、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。

このセクションで説明した組み込み関数の詳細については、 *SQL 解説書* を参照してください。

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 タイプ付き表に対する SELECT 特権が必要です。

付加的な照会仕様技法

ONLY を使用して特定のタイプのオブジェクトを戻す

サブタイプのオブジェクトではなく特定のタイプのオブジェクトだけを戻す照会を行うには、 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。

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バージョンは、次のとおりです。

```
<expression> IS OF (<type_name>[, ...])
```

ここで、*expression* は構造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
```

OUTER を使用してすべての可能性のある属性を戻す

DB2 が構造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、表または視点の列と、もしあればそれに続く表または視点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。ヌル値は、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視点に対する 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視点の情報はすべて、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 タイプ付き表の Address 列には構造型が入っているので、追加の関数を定義し、その列からデータを戻すために追加の SQL を発行する必要があります。構造型の列からデータを戻す方法の詳細については、 338ページの『構造型値の検索と変更』を参照してください。これらの追加のステップを実行したとして、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                 ...
```

その他のヒント

システム生成オブジェクト ID の定義

DB2 が固有のオブジェクト ID を自動的に生成するようにするには、GENERATE_UNIQUE 関数を使用します。 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ておきます。 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;
```

タイプ付き表定義は、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。

タイプ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'));
```

オブジェクト ID 列に対する制約の作成

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固有な制約または基本キー制約を追加するよう、タイプ付き表を更新する必要があります。たとえば、図14 に示されているような、従業員に対する自己参照関係を作成するとします。この自己参照関係では、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。

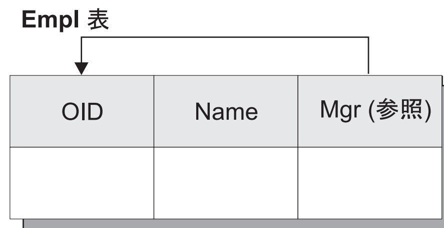


図 14. 自己参照タイプの例

自己参照関係を作成するには、次のステップを実行します。

ステップ 1. タイプを作成する。

```
CREATE TYPE Empl_t AS
   (Name VARCHAR(10), Mgr REF(Empl_t))
   MODE DB2SQL;
```

ステップ 2. タイプ付き表を作成する。

```
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固有制約を追加する。

```
ALTER TABLE Empl ADD CONSTRAINT pk1 UNIQUE(Oid);
```

ステップ 4. 外部キー制約を追加する。

```
ALTER TABLE Empl ADD CONSTRAINT fk1 FOREIGN KEY(Mgr)
   REFERENCES Empl (Oid);
```

列タイプとしての構造型の作成と使用

このセクションでは、ユーザー定義の構造型を列というタイプとして使用することに関係する主なタスクを説明します。このセクションをお読みになる前に、 306ページの『構造型の概説』 の資料に精通しておいてください。

構造型インスタンスの列への挿入

構造型は、表、視点、または列のコンテキストで使用することができます。構造型を作成する時に、ユーザー定義タイプの振る舞いとタイプ属性の両方をカプセル化することができます。タイプの振る舞いを組み込むには、 CREATE TYPE または ALTER TYPE ステートメントを使用して、メソッド・シグニチャーを指定します。メソッドの作成方法の詳細については、389ページの『第14章 ユーザー定義関数 (UDF) およびメソッド』を参照してください。

図15 は、このセクションで例として使用されているタイプ階層を示しています。ルート・タイプは Address_t です。これには、それぞれの国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。

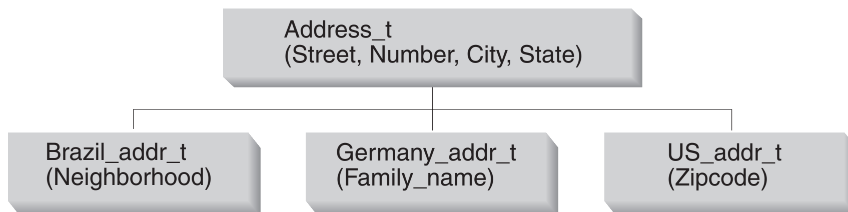


図 15. Address_t タイプの構造型階層

```
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;
```

構造化タイプ属性を列に挿入する

ユーザー定義構造化タイプの属性を、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、以下のような状態になります。

```
- PERSON_T is a structured type that includes the attribute NAME
  of type VARCHAR(30).
- T1 is a table that includes a column C1 of type VARCHAR(30).
- personhv is the host variable declared for type PERSON_T in the
  programming language.
```

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)
```

構造型列を持つ表の定義

データ・レコードの中での構造型のレイアウトに関心がなければ、構造型の列を持つ表の作成については、追加の構文はありません。たとえば、次のステートメントは、Address_t タイプの列を Customer_List 非タイプ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。構造型の挿入方法の詳細については、 337ページの『構造型値が入っている行の挿入』を参照してください。

データ・レコード内での構造型のレイアウトに関心がある場合は、 CREATE TYPE ステートメントの中で INLINE LENGTH 文節を使用して、行の中のその他の値と一緒にインラインで保管する、構造型列のインスタンスの最大サイズを指示することができます。 INLINE LENGTH 文節の詳細については、 *SQL 解説書* の中の CREATE TYPE (構造化) ステートメントを参照してください。

構造型属性を持つタイプの定義

タイプは、構造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除去するようタイプを (使用される前に) 更新できます。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、または別の構造型の属性として使用できます。

構造型値が入っている行の挿入

構造型を作成すると、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 タイプ付き表に追加して、その行に住所を入れるとします。組み込みデータ・タイプの場合と同様に、 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、すべての属性がヌル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 のタイプが Address_t として定義されていますが、代用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。

構造型のインスタンスを作成する度に、構造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 Address_t()..street(street)..number(number)
       ..city(city)..state(state)..zip(zip);
```

次の例は、前述の例の 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'));
```

構造型値の検索と変更

アプリケーションおよびユーザー定義の関数が、構造型列の中のデータにアクセスする方法はいくつかあります。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。これは、344ページの『ホスト言語プログラムへのマッピングの作成: transform 関数』で説明されています。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;
```

しかし、このセクションでは、 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。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。

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';
```

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。前述の例は、静的タイプ Address_t の属性だけにアクセスします。 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;
```

注: サブタイプの属性がすべて同じタイプである場合か、サブタイプの属性を同じタイプにキャストできる場合に、構造型のサブタイプを判別するのに使用できるのは前述の方法だけです。前述の例では、zip、family_name、および neighborhood は、すべて VARCHAR タイプまたは CHAR タイプであるので、同じタイプにキャストすることができます。

TREAT 式または TYPE 述部の構文の詳細については、 *SQL 解説書* を参照してください。

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';
```

タイプについての情報を戻す

330ページの『タイプに関連したその他の組み込み関数』で説明されているように、組み込み関数を使用して、特定のタイプの名前、スキーマ、または初期タイプ 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';
```

transform のタイプとの関連付け

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 定義を更新するには、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。

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 関数が削除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 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。 355ページの『サブタイプ・データの DB2 からの検索 (バインドアウト)』で説明されているように、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);
```

『transform グループを指定する必要のある場合』で説明されているように、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_t 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ないでください。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 という名前を使おうとし、このグループ名は指定の構造型のために指定されたものと想定します。指定の構造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おうとし、このグループ名はこの構造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。

transform グループを指定する必要のある場合

指定の構造型のために定義されている transform グループが多数ある場合は、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 つの状況があります。

- 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。詳細については、 343ページの『外部ルーチン用の transform グループの指定』を参照してください。
- 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。詳細については、343ページの『静的 SQL 用の transform グループの設定』を参照してください。
- 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。詳細については、343ページの『動的 SQL 用の transform グループの設定』を参照してください。

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 ;
```

動的 SQL 用の transform グループの設定

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;
```

静的 SQL 用の transform グループの設定

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
... ;
```

PRECOMPILE および BIND コマンドの詳細については、 *コマンド解説書* を参照してください。

ホスト言語プログラムへのマッピングの作成: transform 関数

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、338ページの『属性の検索』で説明されているように、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、DB2 がオブジェクトを*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。

注: 次のトピックでは、アプリケーションが、常に `Address_t` などの既知の正確なタイプを受け取る単純な事例を取り上げます。外部ルーチンまたはクライアント・プログラムが、 `Address_t`、`Brazil_addr_t`、`Germany_addr_t`、または `US_addr_t` を受け取るような実際にありそうなシナリオについては説明されていません。しかし、外部ルーチンまたはクライアント・プログラムが、すべてのタイプまたはサブタイプを動的に処理する必要のある複雑な事例に基本プロセスを適用しようとする場合は、事前に基本プロセスを理解しておく必要があります。サブタイプ・インスタンスを動的に処理する方法の詳細については、 355ページの『サブタイプ・データの DB2 からの検索 (バインドアウト)』を参照してください。

オブジェクトの外部ルーチンとの交換: function transform

このセクションでは、 *function transform* という transform の特定のタイプを説明します。 DB2 は、これらの 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をやり取りします。 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、350ページの『オブジェクトのクライアント・プログラムとの交換: client transform』で説明されているように、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;
```

346ページの図16 は、DB2 が住所を処理する方法を示しています。

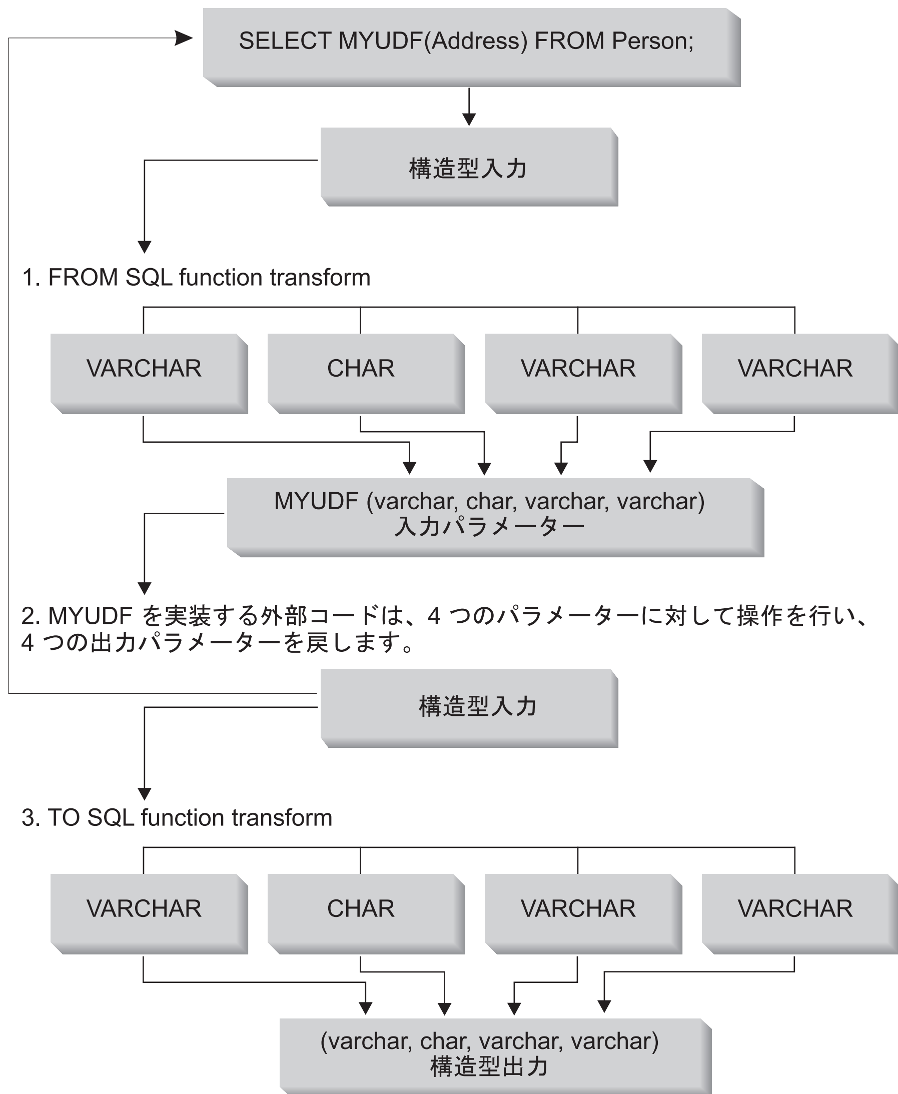


図 16. 構造型パラメーターの外部ルーチンとの交換

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・タイプ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 つの対応するヌル標識パラメーターと構造型の

ための 1 つのヌル標識と一緒に、 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。詳細については、348ページの『構造型パラメーターを外部ルーチンに渡す』を参照してください。

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ヌル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%';
```

SQL を本体として持つルーチンを使用した function transform の実装: 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します。この関数のことを *SQL を本体として持つルーチン* といいます。 SQL を本体として持つルーチンを作成するには、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。 346ページの図16 で示されているように、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。詳細については、340ページの『transform のタイプとの関連付け』を参照してください。

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。

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。

構造型パラメーターを外部ルーチンに渡す: 構造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ヌル標識と、構造型のためのヌル標識を渡す必要があります。次の例は、構造型 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ヌル標識 (address_ind) と、Address_t タイプのそれぞれの属性につき 1 つのヌル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ヌル標識もあります。次のコードは、UDF を実装する関数のための 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,
   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)
```

構造型パラメーターを外部ルーチンに渡す: complex: ルーチンが *st1* と *st2* という 2 つの異なる構造型パラメーターを受け入れ、 *st3* という別の構造型を戻すとします。

```
CREATE FUNCTION myudf (int, st1, st2)
   RETURNS st3
```

表 14. 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 を実装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、分解された構造型の属性のヌル標識および構造型のそれぞれのインスタンスのためのヌル標識が入っています。

```
void SQL_API_FN myudf(
    SQLUDF_INTEGER *INT,
 /* Decompose st1 input */
    SQLUDF_VARCHAR *st1_att1,
    SQLUDF_INTEGER  *st1_att2,
 /* Decompose st2 input */
    SQLUDF_VARCHAR *st2_att1,
```

```
      SQLUDF_CHAR   *st2_att2,
      SQLUDF_INTEGER *st2_att3,
   /* Decompose 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
   )
```

オブジェクトのクライアント・プログラムとの交換: client transform: このセクションでは、*client* transform について説明します。 client transform は、構造型をクライアント・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;
```

351ページの図17 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。

SELECT **Address** FROM Person INTO: **addhv** WHERE...;

1. FROM SQL **function** transform

フラットな住所属性

2. FROM SQL **client** transform

VARCHAR

サーバー

クライアント

3. 住所を VARCHAR として検索した後は、
クライアントはその属性を分解して、
どのようにでもその属性にアクセスすることができます。

図 17. 構造型のクライアント・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 function transform に渡す必要があります。
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ります。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ほど単純に行えます。通常、属性のエンコードと分解は 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。

   プラットフォーム間でデータを処理する方法の詳細については、 353ページの『データ変換についての考慮事項』を参照してください。
3. クライアント・プログラムが値を処理します。

352ページの図18 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。

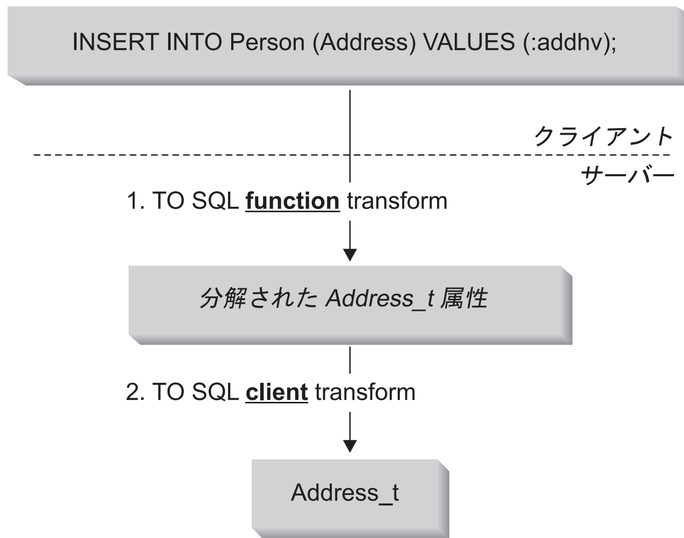


図 18. クライアントからの構造型のバインドイン

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します。
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 function transform への入力として使用されます。
3. TO SQL function transform は、住所を構成してそれをデータベースに戻します。

外部 UDF を使用した client transform の実装: 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複号を行う外部 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 function transform 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 DB2SQL;
```

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;
```

クライアントからのバインドインのための client transform: 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 function transform 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 DB2SQL;
```

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。

呼び出す function transform を DB2 に知らせる方法 to_sql_from_client と from_sql_to_client の両方の 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。詳細については、340ページの『transform のタイプとの関連付け』を参照してください。

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ることを覚えておいてください。たとえば、バイト順序づけ体系が異なるプラットフォーム間でデータが転送される時は、正しい数値を復元するために、数値データに対してバイト逆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・タイプは、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 実装で取られる方法に似た方法で、バイナリー・データ・タイプに変換されたオブジェクトのためのプラットフォー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。
  - 個々のデータ・タイプを正しくエンコードまたは復号し、データ汚損やプログラム障害を避けるために、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・プラットフォー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、プラットフォー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。 DBINFO の使用法の詳細については、411ページの『DB2 から UDF に渡される引き数』を参照してください。

  データ変換の詳細については、517ページの『各国語サポートに関する考慮』を参照してください。

注: 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すべきです。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。

transform 関数の要約

355ページの表15 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 15. *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	復号
transform 関数のパラメーター	構造型	組み込みタイプの行	構造型	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)	構造型	1 つの組み込みタイプ	構造型
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複号が、 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、 function transform に依存する必要はありません。

サブタイプ・データの DB2 からの検索 (バインドアウト)

前述のセクションのほとんどの情報は、アプリケーションが既知の正確なタイプを渡していることを前提としています。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。このセクションでは、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方法を説明します。

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、次のステップを実行します。

ステップ 1. 住所のすべてのバリエーションのための FROM SQL function transform 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 6 でバインド実行時に指定された TRANSFORM GROUP であるからです。

transform グループには、 358ページの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、 358ページの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・タイプ 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 を発行します。これら 2 つの UDF は、356ページの3 で定義されています。これらの UDF は両方とも、それぞれの構造型を、 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。 356ページの3 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、 356ページの1 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、 356ページの2 で発行されています。

サブタイプ・データを DB2 に戻す (バインドイン)

355ページの『サブタイプ・データの DB2 からの検索 (バインドアウト)』で説明されているアプリケーションが住所の値を操作すると、そのアプリケーションは変更された値をデータベースに挿入する必要があります。次の構文を使用して、構造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);
```

構造型のための INSERT ステートメントを実行するには、アプリケーションは次のステップを実行する必要があります。

ステップ 1. 住所のすべてのバリエーションのための TO SQL function transform 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、UNION ALL 文節で結合された SELECT ステートメントに TREAT を指定して、 正しい FROM SQL 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` の中でこの transform を探します。なぜなら、これは 6 でバインド実行時に指定された TRANSFORM GROUP であるからです。 DB2 は、必要とする transform 関数 `stream_address` を検出します。これは 5 で、ルート・タイプ `Address_t` と関連付けられています。

`stream_address` は、 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 を発行して、 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、360ページの3 で定義されています。それぞれの UDF は入力 VARCHAR(150) を取り、ふさわしい構造型の属性を内部的に構成し、このようにして構造型を戻します。

構造型を戻すために、それぞれの UDF には、出力属性値を構造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。 360ページの3 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。

359ページの1 の SQL を本体として持つ transform 関数と、 360ページの2 の transform グループとの関連は、 360ページの3 の CREATE FUNCTION ステートメントの中で指定されています。

構造型ホスト変数の処理

構造型ホスト変数の宣言

静的 SQL の中の構造型ホスト変数を検索または送信するには、その構造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;
```

構造型の記述

構造型変数を指定した 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。

構造型の実際の名前は、SQLVAR2 の中に戻されます。 SQLDA の構造の詳細については、*SQL 解説書* を参照してください。

第13章 ラージ・オブジェクト (LOB) の使用

LOB について 365
ラージ・オブジェクト・データ・タイプ (BLOB、CLOB、DBCLOB) について . . . 366
ラージ・オブジェクト・ロケーターについて 367
例: CLOB 値を使う作業のためのロケーターの使用 369
　LOBLOC プログラム例の作動方法 . . . 369
　C の例: LOBLOC.SQC 370
　COBOL の例: LOBLOC.SQB 372
例: LOB 式の評価の据え置き 375
　LOBEVAL プログラム例の作動方法. . . 376
　C の例: LOBEVAL.SQC. 377
　COBOL の例: LOBEVAL.SQB. 379
　標識変数および LOB ロケーター . . . 382
LOB ファイル参照変数 382
例: ファイルへのドキュメントの抽出 . . . 384
　LOBFILE プログラム例の作動方法 . . . 384
　C の例: LOBFILE.SQC 385
　COBOL の例: LOBFILE.SQB 386
例: CLOB 列へのデータの挿入 388

LOB について

LONG VARCHAR および LONG VARGRAPHIC データ・タイプ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処理の混合したもの、イメージなど、さまざまなデータ・タイプ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・タイプを備えています。その 3 つのデータ・タイプとは、2 進ラージ・オブジェクト (BLOB)、単一バイト文字ラージ・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。

ラージ・オブジェクト (LOB) の保管だけでなく、データベース中の各 LOB を参照したり、それを使用したり修正したりする方法も必要です。 DB2 の各表には、関連する LOB データが大量に含まれている場合があります。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 4 テラバイトまでとなります。特別な行の LOB 列中には、どの時点でも *ラージ・オブジェクト値* が含まれます。

LOB は、他のデータ・タイプの場合と同様にホスト変数を使用して参照および操作することができます。ただし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が、 LOB 値を保留するのに不十分な場合もあります。このような大きな値の操作は、他の方法で行わなければなりません。*ロケーター* は、データベース・サーバーでラージ・オブジェクト値を識別および操作したり、特定の LOB 値を抽出する場合に使用すると便利です。*ファイル参照変数* は、ラージ・オブジェクト (またはその大部分) をクライアント間で物理的に移動する場合に使用すると便利です。

© Copyright IBM Corp. 1993, 2001

注: DB2 は、JDBC および SQLJ アプリケーションに対する LOB サポートを提供しています。 Java アプリケーションでの LOB の使用の詳細については、 668ページの『JDBC 2.0』を参照してください。

次の項では、上記のトピックについて詳しく説明します。

ラージ・オブジェクト・データ・タイプ (BLOB、CLOB、DBCLOB) について

ラージ・オブジェクトのデータ・タイプは、サイズがゼロバイトから 2 ギガバイト -1 の範囲内のデータを保管します。

ラージ・オブジェクトの 3 つのデータ・タイプは、それぞれ次のように定義されます。

- 文字ラージ・オブジェクト (CLOB) －関連付けられたコード・ページを持つ複数の単一バイト文字から成る文字ストリング。このデータ・タイプは、情報量が標準 VARCHAR データ・タイプの限界 (上限 4K バイト) を超えるテキスト指向情報を保留しておくのに最適です。他の文字タイプとの互換性だけでなく、情報のコード・ページ変換もサポートされます。
- 2 バイト文字ラージ・オブジェクト (DBCLOB) －関連付けられたコード・ページを持つ複数の 2 バイト文字から成る文字ストリング。このデータ・タイプは、 2 バイト文字セットが使用されるテキスト指向情報を保留しておくのに最適です。これもまた、他の文字タイプとの互換性があるだけでなく、情報のコード・ページ変換もサポートされます。
- 2 進ラージ・オブジェクト (BLOB) －関連付けられたコード・ページを持たない複数のバイトから成る 2 進ストリング。このデータ・タイプは、 2 進データを保管してそれをユーザー定義特殊タイプ (UDT) で使用できる完全なソース・タイプにすることができるのでこの中で最も役に立ちます。ソース・タイプとして BLOB を使用する UDT は、イメージ、音声、グラフィック、および業務やアプリケーションに特有のその他のデータ・タイプを保管するために作成されます。 UDT の詳細については、295ページの『第11章 ユーザー定義特殊タイプ』を参照してください。

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。それは、2 GB までの LOB を保管できるレコードの外側にラージ・オブジェクト・データを保管することです。ラージ・オブジェクト記述子にアクセスすると、 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・タイプのうち、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。特定サイズの LOB 記述子については、 *SQL 解説書* で CREATE TABLE ステートメントを参照してください。

CREATE TABLE 上の 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。
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・タイプを保管するための LONG 表スペースを識別できます。 CREATE TABLE および ALTER TABLE ステートメントの詳細については、 *SQL 解説書* を参照してください。

LOB が非常に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。 DB2 が 1 GB より大きい LOB 値のロギングを実行しなくても、データベースのログは数百メガバイトに近い LOB 値ですぐに限界に達するかもしれません。 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。このトピックの詳細については、 *管理の手引き* を参照してください。

ラージ・オブジェクト・ロケーターについて

概念的には、LOB ロケーターはこれまでにあった単純な考えを表すものです。すなわち、はるかに大きな値を参照する小さくて管理しやすい値を使用します。具体的には、 LOB ロケーターはホスト変数に保管されている 4 バイトの値で、プログラムはこのホスト変数を使用してデータベース・システム中に保留されている LOB 値 (または LOB 式) を参照することができます。プログラムは、 LOB ロケーターを使用して LOB 値が標準ホスト変数に保管されているかのように操作できます。 LOB ロケーターを使用する場合の相違点は、 LOB 値をサーバーからアプリケーションへ移送する (そして再度戻す) 必要がないということです。

LOB ロケーターは、データベース中の列または物理ストレージの場所ではなく、 LOB 値または LOB 式に関連しています。そのため、ロケーターに LOB 値を設定した後は、元の列または表上で、そのロケーターによって参照される値に影響を及ぼすような操作を行うことはできません。ロケーターに関連する値は、作業単位が終了するまで、または最初にくるロケーターが明示的に解放されるまで有効です。 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。

LOB ロケーターは、DB2 と UDF の間で渡すこともできます。 UDF が LOB ロケーターを使用して LOB 値を操作する際に用いられる特殊な API があります。これらの API の詳細については、459ページの『UDF のパラメーターや結果としての LOB ロケーターの使用』を参照してください。

LOB 値を選択する際には、次の 3 つ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。
- 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。
- すべての LOB 値をファイル参照変数に指定する。 LOB 値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。

プログラム内で LOB 値を使用すると、プログラマーが最適な手段を決めるのに役立ちます。 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除去します。

すべての LOB 値が、そのサイズにかかわらずプログラムに必要な場合は、必ず LOB を転送しなければなりません。この場合でも、3 つのオプションが使用可能です。標準またはファイル・ホスト変数にすべての値を指定できますが、次の例に示すようにロケーターに LOB 値を指定し、それをロケーターから標準ホスト変数へ少しずつ読み込む方が良いでしょう。

例: CLOB 値を使う作業のためのロケーターの使用

この例では、アプリケーション・プログラムが LOB 値のロケーターを検索し、そのロケーターを使用して LOB 値からデータを抽出します。プログラムは、この方法で LOB データの一部分に必要なだけのストレージ (サイズはプログラムにより決まる)を割り振るので、カーソルを使用して一度呼び出しを行うだけでかまいません。

LOBLOC プログラム例の作動方法

1. **ホスト変数を宣言する。** BEGIN DECLARE SECTION および END DECLARE SECTION ステートメントは、ホスト変数宣言を区切ります。ホスト変数には、SQL ステートメントで参照される際に、接頭部としてコロン (:) が付けられます。
   CLOB LOCATOR ホスト変数が宣言されます。
2. **ホスト変数 LOCATOR に LOB 値を取り出す。** CURSOR および FETCH ルーチンを使用して、データベース中の LOB フィールドの場所をホスト変数のロケーターに与えます。
3. **LOB LOCATORS を解放する。**この例で使用されている LOB LOCATORS が解放され、ロケーターをその前の値から解放します。

CHECKERR マクロ / 関数は、プログラム外部にあるエラー検査ユーティリティーです。エラー検査ユーティリティーの所在は、ご使用のプログラム言語により異なります。

C DB2 API を呼び出す C プログラムの場合、 utilapi.c 内の sqlInfoPrint 関数は、 utilapi.h 内の API_SQL_CHECK として再定義されます。 C 組み込み SQL プログラムの場合、 utilemb.sqc 内の sqlInfoPrint 関数は、 utilemb.h 内の EMB_SQL_CHECK として再定義されます。

COBOL CHECKERR は checkerr.cbl. という名前の外部プログラムです。

FORTRAN CHECKERR は util.f ファイルにあるサブルーチンです。

このエラー検査ユーティリティーのソース・コードについては、 125ページの『プログラム例での GET ERROR MESSAGE の使用』を参照してください。

C の例: LOBLOC.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION;  1
      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;

   printf( "Sample C program: LOBLOC¥n" );

   if (argc == 1) {
      EXEC SQL CONNECT TO sample;
  EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else {
      printf ("¥nUSAGE: lobloc [userid passwd]¥n¥n");
      return 1;
   } /* endif */

   /* Employee A10030 is not included in the following select, because
      the lobeval program manipulates the record for A10030 so that it is
      not compatible with lobloc */

   EXEC SQL DECLARE c1 CURSOR FOR
            SELECT empno, resume FROM emp_resume WHERE resume_format='ascii'
            AND empno <> 'A00130';

   EXEC SQL OPEN c1;
   EMB_SQL_CHECK("OPEN CURSOR");

   do {
      EXEC SQL FETCH c1 INTO :number, :resume :lobind;   2
      if (SQLCODE != 0) break;
      if (lobind < 0) {
```

```
         printf ("NULL LOB indicated¥n");
      } else {
         /* EVALUATE the LOB LOCATOR */
         /* Locate the beginning of "Department Information" section */
         EXEC SQL VALUES (POSSTR(:resume, 'Department Information'))
            INTO :deptInfoBeginLoc;
         EMB_SQL_CHECK("VALUES1");

         /* Locate the beginning of "Education" section (end of "Dept.Info" */
         EXEC SQL VALUES (POSSTR(:resume, 'Education'))
            INTO :deptInfoEndLoc;
         EMB_SQL_CHECK("VALUES2");

         /* Obtain ONLY the "Department Information" section by using SUBSTR */
         EXEC SQL VALUES(SUBSTR(:resume, :deptInfoBeginLoc,
            :deptInfoEndLoc - :deptInfoBeginLoc)) INTO :deptBuffer;
         EMB_SQL_CHECK("VALUES3");

         /* Append the "Department Information" section to the :buffer var. */
         EXEC SQL VALUES(:buffer || :deptBuffer) INTO :buffer;
         EMB_SQL_CHECK("VALUES4");
      } /* endif */
   } while ( 1 );

   printf ("%s¥n",buffer);

   EXEC SQL FREE LOCATOR :resume, :deptBuffer;  3
   EMB_SQL_CHECK("FREE LOCATOR");

   EXEC SQL CLOSE c1;
   EMB_SQL_CHECK("CLOSE CURSOR");

   EXEC SQL CONNECT RESET;
   EMB_SQL_CHECK("CONNECT RESET");
   return 0;
}
/* end of program : LOBLOC.SQC */
```

COBOL の例: LOBLOC.SQB

```
Identification Division.
Program-ID. "lobloc".

Data Division.
Working-Storage Section.

    copy "sqlenv.cbl".
    copy "sql.cbl".
    copy "sqlca.cbl".

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.                          1
01 userid            pic x(8).
01 passwd.
  49 passwd-length   pic s9(4) comp-5 value 0.
  49 passwd-name     pic x(18).
01 empnum            pic x(6).
01 di-begin-loc      pic s9(9) comp-5.
01 di-end-loc        pic s9(9) comp-5.
01 resume            USAGE IS SQL TYPE IS CLOB-LOCATOR.
01 di-buffer         USAGE IS SQL TYPE IS CLOB-LOCATOR.
01 lobind            pic s9(4) comp-5.
01 buffer            USAGE IS SQL TYPE IS CLOB(1K).
    EXEC SQL END DECLARE SECTION END-EXEC.

77 errloc          pic x(80).

Procedure Division.
Main Section.
    display "Sample COBOL program: LOBLOC".

* Get database connection information.
    display "Enter your user id (default none): "
         with no advancing.
    accept userid.

    if userid = spaces
      EXEC SQL CONNECT TO sample END-EXEC
    else
      display "Enter your password : " with no advancing
      accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR
* format with the length of the input string.
    inspect passwd-name tallying passwd-length for characters
       before initial " ".

    EXEC SQL CONNECT TO sample USER :userid USING :passwd
        END-EXEC.
    move "CONNECT TO" to errloc.
    call "checkerr" using SQLCA errloc.

* Employee A10030 is not included in the following select, because
* the lobeval program manipulates the record for A10030 so that it is
```

```
* not compatible with lobloc

      EXEC SQL DECLARE c1 CURSOR FOR
               SELECT empno, resume FROM emp_resume
               WHERE resume_format = 'ascii'
               AND empno <> 'A00130' END-EXEC.

      EXEC SQL OPEN c1 END-EXEC.
      move "OPEN CURSOR" to errloc.
      call "checkerr" using SQLCA errloc.

      Move 0 to buffer-length.

      perform Fetch-Loop thru End-Fetch-Loop
         until SQLCODE not equal 0.

* display contents of the buffer.
      display buffer-data(1:buffer-length).

      EXEC SQL FREE LOCATOR :resume, :di-buffer END-EXEC.          3
      move "FREE LOCATOR" to errloc.
      call "checkerr" using SQLCA errloc.

      EXEC SQL CLOSE c1 END-EXEC.
      move "CLOSE CURSOR" to errloc.
      call "checkerr" using SQLCA errloc.

      EXEC SQL CONNECT RESET END-EXEC.
      move "CONNECT RESET" to errloc.
      call "checkerr" using SQLCA errloc.
 End-Main.
         go to End-Prog.

 Fetch-Loop Section.
      EXEC SQL FETCH c1 INTO :empnum, :resume :lobind              2
         END-EXEC.

      if SQLCODE not equal 0
         go to End-Fetch-Loop.

* check to see if the host variable indicator returns NULL.
      if lobind less than 0 go to NULL-lob-indicated.

* Value exists.  Evaluate the LOB locator.
* Locate the beginning of "Department Information" section.
      EXEC SQL VALUES (POSSTR(:resume, 'Department Information'))
               INTO :di-begin-loc END-EXEC.
      move "VALUES1" to errloc.
      call "checkerr" using SQLCA errloc.

* Locate the beginning of "Education" section (end of Dept.Info)
      EXEC SQL VALUES (POSSTR(:resume, 'Education'))
                INTO :di-end-loc END-EXEC.
      move "VALUES2" to errloc.
      call "checkerr" using SQLCA errloc.
```

```
     subtract di-begin-loc from di-end-loc.

* Obtain ONLY the "Department Information" section by using SUBSTR
     EXEC SQL VALUES (SUBSTR(:resume, :di-begin-loc,
              :di-end-loc))
              INTO :di-buffer END-EXEC.
     move "VALUES3" to errloc.
     call "checkerr" using SQLCA errloc.

* Append the "Department Information" section to the :buffer var
     EXEC SQL VALUES (:buffer || :di-buffer) INTO :buffer
              END-EXEC.
     move "VALUES4" to errloc.
     call "checkerr" using SQLCA errloc.

     go to End-Fetch-Loop.

 NULL-lob-indicated.
     display "NULL LOB indicated".

 End-Fetch-Loop. exit.

 End-Prog.
            stop run.
```

例: 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、指定される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。これは、LOB 式の*評価の据え置き* と呼ばれます。

この例では、特別な再開 (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複写を行わずに、新規の再開をアセンブル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。

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 関数最適化プログラムが 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、ディスク I/O が少なくて済みます。

要約すると、以下の場合に LOB ロケーターはプログラミング・シナリオの数に観念的に適合します。
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。
4. (LOB 式の評価を据え置くことにより) パフォーマンスを重視する場合。

LOBEVAL プログラム例の作動方法

1. **ホスト変数を宣言する。** BEGIN DECLARE SECTION および END DECLARE SECTION ステートメントは、ホスト変数宣言を区切ります。ホスト変数には、SQL ステートメントで参照される際に、接頭部としてコロン (:) が付けられます。
CLOB LOCATOR ホスト変数が宣言されます。
2. **ホスト変数 LOCATOR に LOB 値を取り出す。** CURSOR および FETCH ルーチンを使用して、データベース中の LOB フィールドの場所をホスト変数のロケーターに与えます。
3. **LOCATORS を使用して LOB データを操作する。**次に続く 5 つの SQL 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。これは、LOB LOCATORS を使用して行います。
4. **LOB データがターゲット宛先に移動する。**ターゲット宛先に割り当てられた LOB の評価は、この SQL ステートメントまで延期されます。この LOB ステートメントの評価は据え置かれます。
5. **LOB LOCATORS を解放する。**この例で使用されている LOB LOCATORS が解放され、ロケーターをその前の値から解放します。

CHECKERR マクロ / 関数は、プログラム外部にあるエラー検査ユーティリティーです。エラー検査ユーティリティーの所在は、ご使用のプログラム言語により異なります。

C
DB2 API を呼び出す C プログラムの場合、 utilapi.c 内の sqlInfoPrint 関数は、 utilapi.h 内の API_SQL_CHECK として再定義されます。 C 組み込み SQL プログラムの場合、 utilemb.sqc 内の sqlInfoPrint 関数は、 utilemb.h 内の EMB_SQL_CHECK として再定義されます。

COBOL
CHECKERR は checkerr.cbl. という名前の外部プログラムです。

このエラー検査ユーティリティーのソース・コードについては、 125ページの『プログラム例での GET ERROR MESSAGE の使用』を参照してください。

C の例: LOBEVAL.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION; 1
      char userid[9];
      char passwd[19];
      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;
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: LOBEVAL¥n" );

   if (argc == 1) {
      EXEC SQL CONNECT TO sample;
  EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else {
      printf ("¥nUSAGE: lobeval [userid passwd]¥n¥n");
      return 1;
   } /* endif */

   /* delete any instance of "A00130" from
     previous executions of this sample */
   EXEC SQL DELETE FROM emp_resume WHERE empno = 'A00130';

   /* Use a single row select to get the document */
   EXEC SQL SELECT resume INTO :hv_doc_locator1 FROM emp_resume
      WHERE empno = '000130' AND resume_format = 'ascii';  2
   EMB_SQL_CHECK("SELECT");

   /* Use the POSSTR function to locate the start of
      sections "Department Information" & "Education" */
   EXEC SQL VALUES (POSSTR(:hv_doc_locator1, 'Department Information'))
      INTO :hv_start_deptinfo; 3
   EMB_SQL_CHECK("VALUES1");
```

```
   EXEC SQL VALUES (POSSTR(:hv_doc_locator1, 'Education'))
      INTO :hv_start_educ;
   EMB_SQL_CHECK("VALUES2");

   /* Replace Department Information Section with nothing */
   EXEC SQL VALUES (SUBSTR(:hv_doc_locator1, 1, :hv_start_deptinfo -1)
      || SUBSTR (:hv_doc_locator1, :hv_start_educ))
      INTO :hv_doc_locator2;
   EMB_SQL_CHECK("VALUES3");

   /* Move Department Information Section into the hv_new_section_buffer */
   EXEC SQL VALUES (SUBSTR(:hv_doc_locator1, :hv_start_deptinfo,
      :hv_start_educ -:hv_start_deptinfo)) INTO :hv_new_section_buffer;
   EMB_SQL_CHECK("VALUES4");

   /* Append our new section to the end (assume it has been filled in)
      Effectively, this just moves the Department Information to the bottom
      of the resume. */
   EXEC SQL VALUES (:hv_doc_locator2 || :hv_new_section_buffer) INTO
      :hv_doc_locator3;
   EMB_SQL_CHECK("VALUES5");

   /* Store this resume section in the table. This is where the LOB value
      bytes really move */
   EXEC SQL INSERT INTO emp_resume VALUES ('A00130', 'ascii',
      :hv_doc_locator3);   4
   EMB_SQL_CHECK("INSERT");

   printf ("LOBEVAL completed¥n");

   /* free the locators */   5
   EXEC SQL FREE LOCATOR :hv_doc_locator1, :hv_doc_locator2, : hv_doc_locator3;
   EMB_SQL_CHECK("FREE LOCATOR");

   EXEC SQL CONNECT RESET;
   EMB_SQL_CHECK("CONNECT RESET");
   return 0;
}
/* end of program : LOBEVAL.SQC */
```

COBOL の例: LOBEVAL.SQB

```
 Identification Division.
 Program-ID. "lobeval".

 Data Division.
 Working-Storage Section.

     copy "sqlenv.cbl".
     copy "sql.cbl".
     copy "sqlca.cbl".

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.                     1
 01 userid            pic x(8).
 01 passwd.
   49 passwd-length   pic s9(4) comp-5 value 0.
   49 passwd-name     pic x(18).
 01 hv-start-deptinfo pic s9(9) comp-5.
 01 hv-start-educ     pic s9(9) comp-5.
 01 hv-return-code    pic s9(9) comp-5.
 01 hv-new-section-buffer USAGE IS SQL TYPE IS CLOB(5K).
 01 hv-doc-locator1   USAGE IS SQL TYPE IS CLOB-LOCATOR.
 01 hv-doc-locator2   USAGE IS SQL TYPE IS CLOB-LOCATOR.
 01 hv-doc-locator3   USAGE IS SQL TYPE IS CLOB-LOCATOR.
     EXEC SQL END DECLARE SECTION END-EXEC.

 77 errloc          pic x(80).

 Procedure Division.
 Main Section.
     display "Sample COBOL program: LOBEVAL".

* Get database connection information.
     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR
* format with the length of the input string.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".

     EXEC SQL CONNECT TO sample USER :userid USING :passwd
         END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.

* Delete any instance of "A00130" from previous executions
     EXEC SQL DELETE FROM emp_resume
```

```
                 WHERE empno = 'A00130' END-EXEC.

      * use a single row select to get the document
           EXEC SQL SELECT resume INTO :hv-doc-locator1                2
                    FROM emp_resume
                    WHERE empno = '000130'
                    AND resume_format = 'ascii' END-EXEC.
           move "SELECT" to errloc.
           call "checkerr" using SQLCA errloc.

      * use the POSSTR function to locate the start of sections
      * "Department Information" & "Education"
           EXEC SQL VALUES (POSSTR(:hv-doc-locator1,
                    'Department Information'))
                    INTO :hv-start-deptinfo END-EXEC.                  3
           move "VALUES1" to errloc.
           call "checkerr" using SQLCA errloc.

           EXEC SQL VALUES (POSSTR(:hv-doc-locator1,
                    'Education')) INTO :hv-start-educ END-EXEC.
           move "VALUES2" to errloc.
           call "checkerr" using SQLCA errloc.

      * replace Department Information section with nothing
           EXEC SQL VALUES (SUBSTR(:hv-doc-locator1, 1,
                    :hv-start-deptinfo - 1) ||
                    SUBSTR(:hv-doc-locator1, :hv-start-educ))
                    INTO :hv-doc-locator2 END-EXEC.
           move "VALUES3" to errloc.
           call "checkerr" using SQLCA errloc.

      * move Department Information section into hv-new-section-buffer
           EXEC SQL VALUES (SUBSTR(:hv-doc-locator1,
                    :hv-start-deptinfo,
                    :hv-start-educ - :hv-start-deptinfo))
                    INTO :hv-new-section-buffer END-EXEC.
           move "VALUES4" to errloc.
           call "checkerr" using SQLCA errloc.

      * Append the new section to the end (assume it has been filled)
      * Effectively, this just moves the Dept Info to the bottom of
      * the resume.
           EXEC SQL VALUES (:hv-doc-locator2 ||
                    :hv-new-section-buffer)
                    INTO :hv-doc-locator3 END-EXEC.
           move "VALUES5" to errloc.
           call "checkerr" using SQLCA errloc.

      * Store this resume in the table.
      * This is where the LOB value bytes really move.
           EXEC SQL INSERT INTO emp_resume                             4
                    VALUES ('A00130', 'ascii', :hv-doc-locator3)
                    END-EXEC.
           move "INSERT" to errloc.
           call "checkerr" using SQLCA errloc.
```

```
    display "LOBEVAL completed".

    EXEC SQL FREE LOCATOR :hv-doc-locator1, :hv-doc-locator2,          5
             :hv-doc-locator3 END-EXEC.
    move "FREE LOCATOR" to errloc.
    call "checkerr" using SQLCA errloc.

    EXEC SQL CONNECT RESET END-EXEC.
    move "CONNECT RESET" to errloc.
    call "checkerr" using SQLCA errloc.

End-Prog.
    stop run.
```

標識変数および LOB ロケーター

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、その値が NULL であると示す標識変数に負の値が指定されます。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。 NULL 情報は、標識変数値を使用するクライアントに対してローカルに保たれます。サーバーは、有効なロケーターを持つ NULL 値をトラックしません。

LOB ファイル参照変数

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。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 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。

オブジェクトが非常に大きい場合は、ファイルは本来のコンテナーとなります。実際、ほとんどの LOB は、サーバー上でデータベースへ移動される前に、クライアント上でファイルに保管されたデータとして開始するでしょう。ファイル参照変数を使用して、LOB データの移動を補助することができます。プログラムは、クライアント・ファイルからデータベース・エンジンへ直接 LOB データを転送するためにファイル参照変数を使用します。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・ルーチンを作成する必要はありません。

注: 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。

ファイル参照変数には BLOB、CLOB、DBCLOB というデータ・タイプがあります。これはデータ (入力) のソースまたは、データ (出力) のターゲットのどちらかとして使用されます。ファイル参照変数には、ファイルの相対ファイル名か絶対パス名を指定できます。 (後者をお勧めします。) ファイル名の長さは、アプリケーション・プログラム内で指定されます。ファイル参照変数のデータ長の一部は、入力中は使用されません。出力中はデータ長は、アプリケーション・リクエスター・コードによって、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。

ファイル参照変数を使用する場合、入出力の両方に様々なオプションがあります。ファイル参照変数構造に `file_option` フィールドを設定することにより、ファイルの処置を選択することが必要です。次に、入出力両方の値の場合に当てはまるフィールドの指定のための選択肢を示します。

入力ファイル参照変数を使用する場合の値 (示されているのは C 用) とオプションは次のとおりです。

- **SQL_FILE_READ** (標準ファイル)－オープン、読み込み、クローズを行うことができるファイル。 DB2 は、ファイルをオープンするときにファイル内のデータの長さ (バイト数) を判別します。そして、ファイル参照変数構造の `data_length` フィールドを介してその長さを戻します。 (COBOL の場合の値は SQL-FILE-READ で、FORTRAN の場合は sql_file_read です。)

出力ファイル参照変数を使用する場合の値とオプションは次のとおりです。

- **SQL_FILE_CREATE** (新規ファイル)－ 新規のファイルを作成するオプション。ファイルが既に存在している場合はエラー・メッセージを戻します。 (COBOL の場合の値は SQL-FILE-CREATE で、FORTRAN の場合は sql_file_create です。)
- **SQL_FILE_OVERWRITE** (上書きファイル)－ 既存のファイルがない場合に新規のファイルを作成するオプション。ファイルが既に存在している場合は、そのファイルのデータに新規のデータが上書きされます。 (COBOL の場合の値は SQL-FILE-OVERWRITE で、FORTRAN の場合は sql_file_overwrite です。)
- **SQL_FILE_APPEND** (追加ファイル)－ ファイルが存在する場合、そのファイルに出力が追加されるオプション。ファイルが存在しない場合は、新規のファイルが作成される。 (COBOL の場合の値は SQL-FILE-APPEND で、 FORTRAN の場合は sql_file_append です。)

注:

1. 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、グラフィック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。 EUC 環境における DBCLOB ファイルの詳細については、 539ページの『DBCLOB ファイルに関する考慮事項』を参照してください。
2. 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。

ファイル参照変数の詳細については、 *SQL 解説書* を参照してください。

例: ファイルへのドキュメントの抽出

このプログラムの例は、 CLOB エレメントがどのようにして表から外部ファイルへ取り出されるかを示しています。

LOBFILE プログラム例の作動方法

1. **ホスト変数を宣言する。** BEGIN DECLARE SECTION および END DECLARE SECTION ステートメントは、ホスト変数宣言を区切ります。ホスト変数には、SQL ステートメントで参照される際に、接頭部としてコロン (:) が付けられます。
   CLOB FILE REFERENCE ホスト変数が宣言されます。
2. **CLOB FILE REFERENCE ホスト変数を設定する。** FILE REFERENCE の属性が設定されます。特に指定しない限り、完全に宣言されたパスを持たないファイル名が現行の作業ディレクトリーに配置されます。
3. **CLOB FILE REFERENCE ホスト変数への選択。** `resume` というフィールドからデータがホスト変数によって参照されるファイル名に対して選択されます。

`CHECKERR` マクロ / 関数は、プログラム外部にあるエラー検査ユーティリティーです。エラー検査ユーティリティーの所在は、ご使用のプログラム言語により異なります。

C
: DB2 API を呼び出す C プログラムの場合、 utilapi.c 内の `sqlInfoPrint` 関数は、 utilapi.h 内の `API_SQL_CHECK` として再定義されます。 C 組み込み SQL プログラムの場合、 utilemb.sqc 内の `sqlInfoPrint` 関数は、 utilemb.h 内の `EMB_SQL_CHECK` として再定義されます。

COBOL
: `CHECKERR` は `checkerr.cbl` という名前の外部プログラムです。

このエラー検査ユーティリティーのソース・コードについては、 125ページの『プログラム例での GET ERROR MESSAGE の使用』を参照してください。

C の例: LOBFILE.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sql.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION; 1
      SQL TYPE IS CLOB_FILE resume;
      short lobind;
      char userid[9];
      char passwd[19];
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: LOBFILE¥n" );

   if (argc == 1) {
      EXEC SQL CONNECT TO sample;
  EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      EMB_SQL_CHECK("CONNECT TO SAMPLE");
   }
   else {
      printf ("¥nUSAGE: lobfile [userid passwd]¥n¥n");
      return 1;
   } /* endif */

   strcpy (resume.name, "RESUME.TXT");  2
   resume.name_length = strlen("RESUME.TXT");
   resume.file_options = SQL_FILE_OVERWRITE;

   EXEC SQL SELECT resume INTO :resume :lobind FROM emp_resume  3
      WHERE resume_format='ascii' AND empno='000130';

   if (lobind < 0) {
      printf ("NULL LOB indicated ¥n");
   } else {
      printf ("Resume for EMPNO 000130 is in file : RESUME.TXT¥n");
   } /* endif */

   EXEC SQL CONNECT RESET;
   EMB_SQL_CHECK("CONNECT RESET");
   return 0;
}
/* end of program : LOBFILE.SQC */
```

COBOL の例: LOBFILE.SQB

```
 Identification Division.
 Program-ID. "lobfile".

 Data Division.
 Working-Storage Section.

     copy "sqlenv.cbl".
     copy "sql.cbl".
     copy "sqlca.cbl".

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.                    1
 01 userid           pic x(8).
 01 passwd.
   49 passwd-length   pic s9(4) comp-5 value 0.
   49 passwd-name     pic x(18).
 01 resume           USAGE IS SQL TYPE IS CLOB-FILE.
 01 lobind           pic s9(4) comp-5.
     EXEC SQL END DECLARE SECTION END-EXEC.

 77 errloc          pic x(80).

 Procedure Division.
 Main Section.
     display "Sample COBOL program: LOBFILE".

* Get database connection information.
     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR
* format with the length of the input string.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".

     EXEC SQL CONNECT TO sample USER :userid USING :passwd
         END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.

     move "RESUME.TXT" to resume-NAME.                           2
     move 10 to resume-NAME-LENGTH.
     move SQL-FILE-OVERWRITE to resume-FILE-OPTIONS.

     EXEC SQL SELECT resume INTO :resume :lobind                 3
              FROM emp_resume
              WHERE resume_format = 'ascii'
```

```
             AND empno = '000130' END-EXEC.
    if lobind less than 0 go to NULL-LOB-indicated.

    display "Resume for EMPNO 000130 is in file : RESUME.TXT".
    go to End-Main.

NULL-LOB-indicated.
    display "NULL LOB indicated".

End-Main.
    EXEC SQL CONNECT RESET END-EXEC.
    move "CONNECT RESET" to errloc.
    call "checkerr" using SQLCA errloc.
End-Prog.
           stop run.
```

例: CLOB 列へのデータの挿入

次の C プログラム・セグメントのパス記述には、以下の変数があります。

- userid は、1 ユーザー用のディレクトリーを表す。
- dirname は、『userid』 のサブディレクトリー名を表す。
- filnam.1 は、表に挿入したいドキュメントのうちのいずれか 1 つの名前にする。
- clobtab は、CLOB データ・タイプを使用した表の名前にする。

以下の例は、 :hv_text_file によって参照される標準ファイルからデータを CLOB 列へ挿入する方法を示しています (ただしこの例で使用されているパス名は、UNIX ベースのシステムの場合のものです)。

```
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; /* this is a 'regular' file */

EXEC SQL INSERT INTO CLOBTAB
  VALUES(:hv_text_file);
```

第14章 ユーザー定義関数 (UDF) およびメソッド

関数とメソッドについて 389
関数とメソッドを使用する理由 390
UDF とメソッドの概念 393
関数とメソッドのインプリメント 394
関数とメソッドの作成 395
関数とメソッドの登録 395
UDF とメソッドの登録例 395
例: べき乗 396
例: ストリングの検索 396
例: BLOB ストリング検索 397
例: UDT のストリング検索 398
例: UDT パラメーターを指定した外部関数 398
例: UDT での AVG 399
例: 計算 399
例: OLE オートメーション・オブジェクトを使った計算 400
例: 文書 ID を戻す表関数 400
関数とメソッドの使用法 401
関数の参照 401
関数呼び出しの例 402
関数でのパラメーター・マーカーの使用法 403
修飾された関数参照の使用法 403
修飾されない関数参照の使用法 404
関数参照の要約 405

関数とメソッドについて

ユーザー定義関数 は、 SQL にユーザー独自の拡張を定義できる機構です。 DB2 が提供する組み込み関数は便利な関数の集合ですが、これらはユーザーの要求のすべてを満たすわけではありません。 DB2 が提供する全関数のリストについては、 *SQL 解説書* の『サポートされている関数』を参照してください。

メソッド を使用すれば SQL オブジェクトの振る舞いを定義できるため、 UDF と同じように SQL に独自の拡張を書き込むことができます。 UDF と異なる点は、メソッドは表内で列として保管されている構造型にしか関連付けることができないという点です。

以下のことが必要な場合に SQL を拡張しなければなりません。

- **カスタマイズ**

  DB2 にはユーザーのアプリケーションに固有の関数はありません。関数が、単純な変換、通常の計算または複雑な多変量分析のいずれであっても、おそらく UDF を使用してジョブを実行できます。

- **柔軟性**

  DB2 の組み込み関数は、ユーザーのアプリケーションに含めたい変化をすべて許可するわけではありません。

- **標準化**

  ユーザー・サイトのプログラムの多くは、同じ関数の基本集合をインプリメントしますが、そのすべてのインプリメンテーションにはわずかな違いがあります。このため、確実に一貫した結果を受け取るとは限りません。このような関数を UDF の形式

© Copyright IBM Corp. 1993, 2001

で一度正しく使用すると、これらのプログラムはすべて、直接 SQL で同じインプリメンテーションを使用して一貫した結果を提供できます。

- **オブジェクト関連サポート**

  295ページの『第11章 ユーザー定義特殊タイプ』と 305ページの『第12章 複合オブジェクトの処理: ユーザー定義の構造型』で説明するように、特殊タイプと構造型は機能を拡張し、DB2 の安全性を高めるのに非常に役立ちます。列内に保管されている構造型の振る舞いは、メソッドを作成することによって定義できます。同様に、特殊タイプに対する処理は、関数を作成することによって行えます。

関数とメソッドを使用する理由

DB2 のアプリケーションを作成する場合、要求されたアクションまたは演算を次のどちらかとして実行することができます。

- UDF として
- メソッドとして
- アプリケーションのサブルーチンとして

新しい演算は、ご使用のアプリケーションのサブルーチンとして実行する方が簡単のように見えますが、 UDF の使用をお勧めするのにはもっともな理由があります。

- **再使用**

  新しい操作を、ユーザー・サイトに存在する他のユーザーやプログラムが活用できる場合、 UDF やメソッドはその操作を再利用するのに役立ちます。さらに、データベースの使用者が式を利用できる場合はいつでも SQL で操作を直接呼び出すことができます。 UDF の場合、データベースは、関数引き数のデータ・タイプのプロモーション (たとえば DECIMAL から DOUBLE など) を自動的に多数処理し、ユーザーの操作が互換性のある別のデータ・タイプに適用できるようにします。

  自分の新規の操作をサブルーチンとして使用し、他の人たちが各自のプログラムでそれを使用できるようにすると便利で、これによって、DB2 に関数を定義する必要をなくすことができます。この場合、関係する他のすべてのアプリケーション開発者に通知し、利用するサブルーチンを効率的にパッケージする必要があります。ただし、通常はコマンド行プロセッサー (CLP) を使用してデータベースにアクセスするような、対話式ユーザーは無視されます。 CLP ユーザーは、データベース中の UDF やメソッド以外の関数を使用できません。このことは SQL (Lotus Approach など) を使用しかつ再コンパイルできない他のすべてのツールにも当てはまります。

- **パフォーマンス**

  UDF やメソッドをご使用のアプリケーションから呼び出す代わりに、データベース・エンジンから直接呼び出すと、パフォーマンスが飛躍的に向上しますが、処理を続けるために操作によってデータを修飾すると特に向上します。あるデータの処理を行う単純なシナリオを例に考えてみます。ただし、関数 `SELECTION_CRITERIA()` として表されるいくつかの選択基準が満たされているものとします。この場合、ご使用のアプリケーションは次の選択ステートメントを発行することができます。

```
SELECT A,B,C FROM T
```

アプリケーションは、各行を受け取るたびにデータに対して SELECTION_CRITERIA を実行し、データの処理を続けるかどうかを判別します。この例では、表 T のすべての行がアプリケーションに戻されなければなりません。しかし、SELECTION_CRITERIA() が UDF として使用されている場合は、アプリケーションは次のステートメントを発行することができます。

```
SELECT A,B,C FROM T WHERE SELECTION_CRITERIA(A,B) = 1
```

この場合、関係のある行のみがアプリケーションとデータベース間のインターフェースを介して渡されます。表が大きい場合、または SELECTION_CRITERIA に有効なフィルター機能が提供されている場合、パフォーマンスは格段に向上します。

また、ラージ・オブジェクト (LOB) を扱う場合にも UDF を使用してパフォーマンスを向上させることができます。 LOB タイプ中のある値から情報を抽出する関数があると、データベース・サーバー上ですぐに抽出を実行し、抽出した値のみをアプリケーションに戻すことができます。この方が、すべての LOB 値をアプリケーションに戻してから抽出を行うよりも効率的です。この関数を UDF としてパッケージするパフォーマンス値は、場合によってはかなり大きくなることもあります。 (LOB ロケーターを使用することによって、LOB の一部を抽出することもできます。同様のシナリオの例については、375ページの『例: LOB 式の評価の据え置き』を参照してください。)

さらに、CREATE FUNCTION ステートメントの RETURNS TABLE 文節を使用すれば、*表関数* と呼ばれる UDF を定義することができます。表関数を使用すると、DB2 データベースの外にあるデータ (非リレーショナル・データ・ストアを含む) に対して関係操作および SQL の能力を非常に効率的に使用できます。表関数は、様々なタイプおよび意味を持つ個々のスカラー値を引き数とし、呼び出し元の SQL ステートメントに表を戻します。関係のあるデータのみを生成する表関数を作成して、不必要な行や列をなくすことができます。表関数の詳細については (使用できる場所についての規則を含む)、 *SQL 解説書* を参照してください。

表を戻すメソッドは作成できません。

- **特殊タイプの振る舞い**

  ユーザー定義特殊タイプ (UDT) (*特殊タイプ* とも言う) は、 UDF を使用してインプリメントすることができます。 UDT の詳細については、295ページの『第11章 ユーザー定義特殊タイプ』を参照してください。 UDT に関する追加情報と、ここで説明したキャスト*機能* の重要な概念については、 *SQL 解説書* を参照してください。特殊タイプを作成する場合、特殊タイプとそのソース・タイプの間にキャスト機能が自動的に提供され、そのソース・タイプに基づいて =、>、 < などの比較演算子が与えられます。さらに追加する振る舞いがあれば、ユーザー自身が行わなければなりませ

ん。特殊タイプの振る舞いはデータベース中のすべての特殊タイプのユーザーがアクセスしやすい場所に明確に保持しておくべきであるため、そのインプリメンテーション機構として UDF を使用できます。

たとえば、1 メガバイトの BLOB で定義されている BOAT という特殊タイプがあるとします。 BLOB には、さまざまな未加工の仕様およびいくつかの図が含まれています。 BLOB ソース・タイプで定義されている特殊タイプを使用してボートのサイズを比較する場合、比較演算は自動的に生成されません。この場合、BOAT_COMPARE 関数を実行すると、選択した測度に基づいて一方のボートが他方より大きいかどうかを判別することができます。選択できるのは、変位、全長、メートル・トン、あるいは BOAT オブジェクトに基づいたその他の計算です。 BOAT_COMPARE 関数は、以下のように作成できます。

```
CREATE FUNCTION BOAT_COMPARE (BOAT, BOAT) RETURNS INTEGER ...
```

ユーザーの関数が、最初のボートの方が大きい場合は 1 を、 2 番目のボートの方が大きい場合に 2 を、両者が等しい場合は 0 を戻すならば、この関数を自分の SQL コードで使用してボートを比較することができます。たとえば、次の表を作成するとします。

```
CREATE TABLE BOATS_INVENTORY (
  BOAT_ID        CHAR(5),
  BOAT_TYPE      VARCHAR(25),
  DESIGNER       VARCHAR(40),
  OWNER          VARCHAR(40),
  DESIGN_DATE    DATE,
  SPEC           BOAT,
  ...                  )

CREATE TABLE MY_BOATS (
  BOAT_ID        CHAR(5),
  BOAT_TYPE      VARCHAR(25),
  DESIGNER       VARCHAR(40),
  DESIGN_DATE    DATE,
  ACQUIRE_DATE   DATE,
  ACQUIRE_PRICE  CANADIAN_DOLLAR,
  CURR_APPRAISL  CANADIAN_DOLLAR,
  SPEC           BOAT,
  ...                  )
```

すると次の SQL SELECT ステートメントを実行できます。

```
SELECT INV.BOAT_ID, INV.BOAT_TYPE, INV.DESIGNER,
  INV.OWNER, INV.DESIGN_DATE
  FROM BOATS_INVENTORY INV, MY_BOATS MY
  WHERE MY.BOAT_ID = '19GCC'
  AND BOAT_COMPARE(INV.SPEC, MY.SPEC) = 1
```

この単純な例を実行すると、 MY_BOATS 内の特定のボートよりも大きいすべてのボートを BOATS_INVENTORY から戻します。ただしこの例では、データベース・サーバー内で比較が行われているので、関係のある列のみがアプリケーションに戻され

ます。つまり、BOAT というデータ・タイプの値をまったく渡さないようにします。BOAT は 1 メガバイトの BLOB データ・タイプに基づいているので、これにより記憶とパフォーマンスが大幅に向上します。

UDF とメソッドの概念

次に、UDF およびメソッドのコーディングを行う前に知っておかなければならない重要な概念について説明します。

- 関数の完全名

  関数の完全名は、`<schema-name>.<function-name>` です。関数を参照するどの場所でもこの完全名を使用できます。以下に例を示します。

  ```
  SLICKO.BOAT_COMPARE    SMITH.FOO    SYSIBM.SUBSTR    SYSFUN.FLOOR
  ```

  `<schema-name>.` を省略することもできますが、その場合、DB2 は参照されている関数を識別しなければなりません。以下に例を示します。

  ```
  BOAT_COMPARE    FOO    SUBSTR    FLOOR
  ```

- 関数パス

  関数パス 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* 参照を分析する中心となるものです。関数を参照する DDL ステートメント内での関数パスの使用については、 *SQL 解説書* を参照してください。関数パスとは、スキーマ名の順序付けられたリストです。これにより、 UDT だけでなく UDF にも修飾されない関数参照を分析する 1 セットのスキーマが提供されます。関数参照が、パス内の複数のスキーマの関数に一致する場合は、パス内のスキーマの順序を使用してこの一致を分析します。関数パスは、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。一方動的 SQL の場合は、SET CURRENT FUNCTION PATH ステートメントにより設定されます。関数パスのデフォルトは、次のようになります。

  ```
  "SYSIBM","SYSFUN","<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。

- 多重定義関数名

  関数名は*多重定義* することができます。これはすなわち、複数の関数が同じスキーマ内でも同じ名前を持つことができるという意味です。ただし 2 つの関数が同じ*シグニチャー* を持つことはできません。シグニチャーは、定義順に並んだ関数パラメーターすべての定義済みデータ・タイプに連結した、修飾された関数名として定義することができます。多重定義関数の例については、 397ページの『例: BLOB ストリング検索』を参照してください。

- 関数選択アルゴリズム

  修飾された参照または修飾されない参照のいずれの場合も多重定義の事実および、すべての関数参照に*最適なもの* を選択する関数パスを考慮に入れるのは、*関数選択ア*

ルゴリズム です。組み込み関数および SYSFUN スキーマ中の関数 (IBM 提供のものも含む) の参照も、関数選択アルゴリズムを使用して処理されます。

- 関数のタイプ

  ユーザー定義関数はそれぞれ、*スカラー関数*、*列関数*、または*表関数* として分類されます。*スカラー関数* は、呼び出されるたびに単一の値応答を戻します。たとえば、組み込み関数 `SUBSTR()` はスカラー関数です。スカラー UDF およびメソッドは、外部 (C などのプログラミング言語でコード化されている) とソース (既存関数のインプリメンテーションを使用している) のいずれかです。

  列関数 は、類似した値の集合 (データの列) を受け取り、この値の集合から単一の値応答を戻します。これらの関数は、DB2 では*総計関数* とも呼ばれます。列関数の例としては、組み込み関数 AVG() があります。 DB2 に対して、外部列 UDF は定義できませんが、組み込み列関数をソースとする列 UDF は定義できます。これは特殊タイプの場合に便利です。たとえば、基本タイプ INTEGER によって定義された特殊タイプ `SHOESIZE` が存在する場合、 `AVG(SHOESIZE)` という UDF を、既存の組み込み列関数 `AVG(INTEGER)` をソースとする列関数として定義できます。

  表関数 は、参照元の SQL ステートメントに表を戻します。表関数を参照できるのは、 SELECT ステートメントの FROM 文節内だけです。 このような関数を使用すると、SQL 言語を DB2 以外のデータに適用したり、 DB2 以外のデータを取得してそれを DB2 の表に入れることができます。たとえば、DB2 以外のデータで構成されるファイルを表に変換したり、 WWW やオペレーティング・システムからデータを入手してそれを表として戻すことができます。表関数は、外部関数にすることしかできません。

関数パスの概念、SET CURRENT FUNCTION PATH ステートメント、および関数選択アルゴリズムについては、 *SQL 解説書* で詳しく説明しています。 FUNCPATH プリコンパイルおよびバインド・オプションについては、 *コマンド解説書* で詳しく説明しています。

UDF やメソッドおよび組み込み関数を連合システムのデータ・ソース関数にマッピングする概念については、 *SQL 解説書* を参照してください。そのようなマップの作成の指針については、600ページの『データ・ソース関数の呼び出し』を参照してください。

関数とメソッドのインプリメント

外部 UDF またはメソッドのインプリメントのプロセスには、以下のステップが必要です。

1. UDF またはメソッドを作成する
2. UDF またはメソッドをコンパイルする
3. UDF またはメソッドをリンクする
4. UDF またはメソッドをデバッグする
5. UDF またはメソッドを DB2 に登録する

これらのステップが正常に完了すると、UDF またはメソッドを、 CREATE VIEW などの DML または DDL ステートメントで使用することができます。 UDF とメソッドの作成および定義について以下の各項で説明し、その後に UDF とメソッドの使用法を説明します。 UDF とメソッドのコンパイルとリンクに関する説明は、 *アプリケーション構築の手引き* を参照してください。 UDF とメソッドのデバッグについては、 497ページの『UDF のデバッグ』を参照してください。

関数とメソッドの作成

UDF とメソッドの作成方法については、 409ページの『第15章 ユーザー定義関数 (UDF) とメソッドの作成』で詳しく説明しています。ここでは、DB2 と UDF またはメソッド間のインターフェース、コーディングについての考慮事項、コーディングの例、およびデバッグ情報についての詳細が述べられています。 UDF とメソッドのコンパイルとリンクに関連するタスクについては、 *アプリケーション構築の手引き* を参照してください。

関数とメソッドの登録

DB2 への UDF やメソッドの登録は、実コードを作成し、それを完全にテストしてから行わなければなりません。なお、UDF やメソッドは実際に作成する前に定義することもできます。しかし、実行の際に問題が発生しないように、登録前に UDF やメソッドを作成し、広範囲にわたってテストを行うようにしてください。 UDF とメソッドのテストの詳細については、 497ページの『UDF のデバッグ』を参照してください。

CREATE FUNCTION ステートメントを使用して、UDF を DB2 に定義 (または登録) します。メソッドを DB2 に登録するには、CREATE TYPE ステートメントまたは ALTER TYPE ステートメントを使用して特定の構造型のためのメソッドを定義してから、 CREATE METHOD ステートメントを使用してそのメソッド本体をメソッド仕様に関連付けます。これらのステートメントとそのオプションの詳細については、 *SQL 解説書* を参照してください。

UDF とメソッドの登録例

次に、UDF やメソッドを登録できるさまざまな代表的な場合を具体例を挙げて説明します。例には次のものが含まれます。
- 例: べき乗
- 例: ストリングの検索
- 例: UDT のストリング検索
- 例: UDT パラメーターを指定した外部関数
- 例: UDT での AVG
- 例: 計算

これらの例では、以下のことに注意してください。

- キーワードまたはキーワード / 値の指定は、表示を一貫して理解しやすくするために常に同じ順で表されます。これらの CREATE FUNCTION ステートメントまたは CREATE METHOD ステートメントの 1 つを実際に作成する場合、関数名およびパラメーターのデータ・タイプのリストの後に任意の順序で指定できます。
- EXTERNAL NAME 文節の指定は、 DB2 (UNIX 版) プラットフォーム用の指定が示されます。 UNIX 以外のプラットフォームでこれらの例を実行する場合は、変更を加える必要があります。たとえば、スラッシュ (/) すべてを円記号 (¥) に変換し、 `C:` などのドライブ文字を追加すると、 OS/2 や Windows 環境で例を実行できます。 EXTERNAL NAME 文節の詳細な説明については、 *SQL 解説書* を参照してください。

例: べき乗

たとえば、浮動小数点値をべき乗するために外部 UDF を作成し、それを MATH スキーマに登録したい場合を考えます。このとき DBADM 権限を持っているものとします。関数を広範囲にわたってテストした結果、保全性が損なわれることがないと分かっているので、この関数を NOT FENCED として定義します。 DBADM 権限を持っているので、 CREATE_NOT_FENCED というデータベース権限を持つことになります。これは、関数を NOT FENCED として定義するために必要になります。

```
CREATE FUNCTION MATH.EXPON (DOUBLE, DOUBLE)
  RETURNS DOUBLE
  EXTERNAL NAME '/common/math/exponent'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  NOT FENCED
```

この例では、システムは NOT NULL CALL というデフォルトを使用します。いずれか一方の引き数が NULL である場合、結果は NULL でなければならないため、これは要求を満たします。また、スクラッチパッドが必要なく、最終呼び出しを行う必要がないので、 NO SCRATCHPAD および NO FINAL CALL というデフォルトが使用されます。 EXPON が並列にしてはいけない理由がないので、ALLOW PARALLELISM デフォルトが使用されます。

例: ストリングの検索

Willie という人が UDF を作成して、引き数として渡される所定の短いストリングを同じく引き数として渡される所定の CLOB 値内で検索すると仮定します。 UDF は、そのストリングを検出すると CLOB 内のそのストリングの位置を戻し、検出しないとゼロを戻します。 UDF が完全にはテストされていない恐れがあると、この関数に対するデータベースの保全性にかかわるため、この関数を FENCED として定義します。

さらに、Willie は FLOAT の結果を戻す関数を作成しています。このとき、SQL で使用されると必ず INTEGER を戻すことが分かっているものとします。すると、次の関数を作成することができます。

```
CREATE FUNCTION FINDSTRING (CLOB(500K), VARCHAR(200))
  RETURNS INTEGER
  CAST FROM FLOAT
  SPECIFIC "willie_find_feb95"
  EXTERNAL NAME '/u/willie/testfunc/testmod!findstr'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  FENCED
```

CAST FROM 文節を使用しているのは、 UDF 本体から実際に戻される FLOAT 値を INTEGER にキャストして、それからその値を、UDF を使用するステートメントに戻すよう指定するためです。 *SQL 解説書* に説明されているように、 INTEGER 組み込み関数はこのキャストを実行できます。また、関数に対してユーザー指定の名前を付け、後に DDL (398ページの『例: UDT のストリング検索』を参照) で参照することもできます。 NULL 値を処理する UDF は作成されなかったので、 NOT NULL CALL というデフォルトを使用します。また、スクラッチパッドがないので、 NO SCRATCHPAD および NO FINAL CALL というデフォルトを使用します。 FINDSTRING を並列にしてはいけない理由がないので、 ALLOW PARALLELISM デフォルトが使用されます。

例: BLOB ストリング検索

この関数は、CLOB だけでなく BLOB 上でも実行できるようにしたいので、最初のパラメーターとして BLOB を指定する FINDSTRING をもう 1 つ定義します。

```
CREATE FUNCTION FINDSTRING (BLOB(500K), VARCHAR(200))
  RETURNS INTEGER
  CAST FROM FLOAT
  SPECIFIC "willie_fblob_feb95"
  EXTERNAL NAME '/u/willie/testfunc/testmod!findstr'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  FENCED
```

この例は、UDF 名の多重定義を具体的に説明し、複数の UDF やメソッドが同じ本体を共用できることを示しています。なお、BLOB は CLOB に割り当てることはできませんが、同じソース・コードを使用できます。 DB2 と UDF の間で BLOB と CLOB のプログラミング・インターフェースが同じ (長さの次にデータがある) なので、上の例ではプログラミング上の問題はありません。 DB2 は、特別な関数本体を使用する UDF が同じ本体を使用する別の UDF と一貫しているかどうか検査しません。

例: UDT のストリング検索

この例は、前述の例に続くものです。 397ページの『例: BLOB ストリング検索』の FINDSTRING 関数には満足していますが、ここでソース・タイプを BLOB とする特殊タイプ BOAT を定義したとしましょう。すると、BOAT というデータ・タイプを持つ値の演算を行う FINDSTRING も必要になるので、 FINDSTRING をもう 1 つ作成します。この関数は、397ページの『例: BLOB ストリング検索』で BLOB 値に対して演算を行った FINDSTRING をソースとします。この例では、FINDSTRING をさらに多重定義する点に注目してください。

```
CREATE FUNCTION FINDSTRING (BOAT, VARCHAR(200))
  RETURNS INT
  SPECIFIC "slick_fboat_mar95"
  SOURCE SPECIFIC "willie_fblob_feb95"
```

この FINDSTRING 関数は 397ページの『例: BLOB ストリング検索』の FINDSTRING 関数とはシグニチャーが異なるので、名前を多重定義しても問題はありません。ユーザー独自の名前を付けて、後で DDL で参照できるようにしたいとします。 SOURCE 文節を使用しているので、EXTERNAL NAME 文節または、関数の属性を指定する関連したキーワードは使用できません。これらの属性は、ソース関数から取り出されます。最後に、397ページの『例: BLOB ストリング検索』では、ご使用のソース関数を識別する際に特定の関数名が明示的に示されていることに注意してください。これは修飾されない参照であるので、このソース関数が常駐するスキーマは関数パスにあるはずで、そうでなければ参照は分析されません。

例: UDT パラメーターを指定した外部関数

BOAT を入手し、その設計属性を調べ、そのボートのコストをカナダ・ドルで生成する、別の UDF を作成しました。労働コストはユーロ、円、または US ドルで内部的に計算されるとしても、この関数は、ボートを作成するのに必要なコストを、要求された通貨であるカナダ・ドルで出す必要があります。これは、関数が為替相場の Web ページから現在の為替相場の情報を取り入れなければならないということを意味し、その答えは、Web ページ内で検出するものによって決まります。このため、関数は NOT DETERMINISTIC (または VARIANT) になります。

```
CREATE FUNCTION BOAT_COST (BOAT)
  RETURNS INTEGER
  EXTERNAL NAME '/u/marine/funcdir/costs!boatcost'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  NOT DETERMINISTIC
  NO EXTERNAL ACTION
  FENCED
```

CAST FROM および SPECIFIC は指定されておらず、 NOT DETERMINISTIC が指定されていることに注意してください。ここでも、安全のために FENCED が選択されます。

例: UDT での AVG

この例は、CANADIAN_DOLLAR 特殊タイプで AVG 列関数をインプリメントします。CANADIAN_DOLLAR の定義については、 297ページの『例: 通貨』を参照してください。強力タイピングは、特殊タイプ上で AVG 組み込み関数を使用しないようにします。 CANADIAN_DOLLAR のソース・タイプが DECIMAL であったと分かるので、AVG を、AVG (DECIMAL) 組み込み関数をソースとしてインプリメントします。これができるかどうかは、DECIMAL から CANADIAN_DOLLAR へ、またその逆方向にキャストできるかどうかによりますが、 DECIMAL は CANADIAN_DOLLAR のソース・タイプであるので、これらのキャストは実行できるということが分かります。

```
CREATE FUNCTION AVG (CANADIAN_DOLLAR)
  RETURNS CANADIAN_DOLLAR
  SOURCE "SYSIBM".AVG(DECIMAL(9,2))
```

なお、別の AVG 関数がご使用の関数パスに潜在する場合に備えて SOURCE 文節中で関数名を修飾しています。

例: 計算

単純な計算関数は、最初に 1 を戻し、呼び出されるたびに結果を 1 ずつ増分します。この関数は SQL 引き数を取らず、応答が呼び出しごとに変化するので、 NOT DETERMINISTIC 関数として定義されています。 これは、戻される最後の値を保管するためにスクラッチパッドを使用し、呼び出されるたびにこの値を増分して戻します。このとき、この関数を厳密にテストした結果、データベース上で DBADM 権限を持っているので、これを NOT FENCED と定義します。 (DBADM は、CREATE_NOT_FENCED を暗黙指定します。)

```
CREATE FUNCTION COUNTER ()
  RETURNS INT
  EXTERNAL NAME '/u/roberto/myfuncs/util!ctr'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  NOT DETERMINISTIC
  NOT FENCED
  SCRATCHPAD
  DISALLOW PARALLEL
```

パラメーターは定義されず、空括弧のみとなります。上記の関数は SCRATCHPAD を指定し、デフォルト指定の NO FINAL CALL を使用します。この場合、スクラッチパッドのデフォルト・サイズ (100 バイト) は十分大きく、最後の呼び出しによってストレージを解放する必要はないので、 NO FINAL CALL が指定されています。 COUNTER 関数が正しく動作するためには、単一のスクラッチパッドを使う必要があるので、 DISALLOW PARALLEL を追加して、DB2 がこの関数を並列に実行しないようにします。この COUNTER 関数のインプリメンテーションについては、478ページの『例: カウンター』を参照してください。

例: OLE オートメーション・オブジェクトを使った計算

この例では、前に挙げた計算例を OLE (オブジェクトのリンクと埋め込み) オートメーション・オブジェクト counter としてインプリメントします。これには、呼び出しの数を記録するためのインスタンス変数 nbrOfInvoke があります。 UDF が呼び出されるたびに、オブジェクトの increment メソッドは、 nbrOfInvoke インスタンスを増分し、その時点での状態を戻します。 オートメーション・オブジェクトは、 OLE プログラム ID (progID) bert.bcounter を指定して Windows レジストリーに登録されます。

```
CREATE FUNCTION bcounter ()
  RETURNS integer
  EXTERNAL NAME 'bert.bcounter!increment'
  LANGUAGE OLE
  PARAMETER STYLE DB2SQL
  SCRATCHPAD
  NOT DETERMINISTIC
  FENCED
  NULL CALL
  NO SQL
  NO EXTERNAL ACTION
  DISALLOW PARALLEL;
```

クラス counter のインプリメンテーションは、 491ページの『例: BASIC でのカウンター OLE オートメーション UDF』と 492ページの『例: C++ でのカウンター OLE オートメーション UDF』に示されています。 DB2 による OLE サポートの詳細については、441ページの『OLE オートメーション UDF の作成』を参照してください。

例: 文書 ID を戻す表関数

特定の対象域 (最初のパラメーター) と一致し、特定のストリング (2 番目のパラメーター) を含む、テキスト管理システム内の各文書について単一の文書 ID 列を戻す、表関数を作成しました。 この UDF は、テキスト管理システムの関数を使用して、即座に文書を識別します。

```
CREATE FUNCTION DOCMATCH (VARCHAR(30), VARCHAR(255))
  RETURNS TABLE (DOC_ID CHAR(16))
  EXTERNAL NAME '/common/docfuncs/rajiv/udfmatch'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  NOT FENCED
  SCRATCHPAD
  NO FINAL CALL
  DISALLOW PARALLEL
  CARDINALITY 20
```

この UDF は、単一セッションのコンテキスト内では常に同じ表を戻すので、 DETERMINISTIC として定義されています。 RETURNS 文節で、列名 DOC_ID を含む、DOCMATCH の出力が定義されることに注意してください。 FINAL CALL は、各

表関数ごとに指定する必要はありません。さらに、表関数は並列に実行できないので、DISALLOW PARALLEL キーワードが追加されています。 DOCMATCH の出力のサイズはよく変化しますが、 CARDINALITY 20 を代表値として指定し、DB2 最適化プログラムが優れた決定を行えるようにしています。

通常、この表関数は、次のように文書テキストを含む表と組み合わせて使用されます。

```
SELECT T.AUTHOR, T.DOCTEXT
FROM DOCS as T, TABLE(DOCMATCH('MATHEMATICS', 'ZORN''S LEMMA')) as F
WHERE T.DOCID = F.DOC_ID
```

FROM 文節で表関数を指定するための特殊な構文 (TABLE キーワード) に注意してください。この呼び出しでは、`docmatch()` 表関数は、 `Zorn's Lemma` を参照する各 `mathematics` 文書について、単一行 DOC_ID を含む行を戻します。これらの DOC_ID 値が結合し、作成者の名前と文書テキストを取得して、マスター文書表になります。

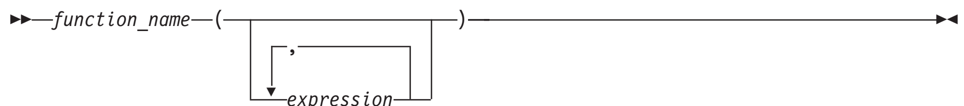
関数とメソッドの使用法

スカラー UDF / メソッドと列 UDF / メソッドは、式が有効な場所 (すべての列関数について、有効性を制限する追加の規則があります) であれば、 SQL ステートメント内のどこでも呼び出せます。表 UDF を参照できるのは、SELECT の FROM 文節内だけです。 *SQL 解説書* では、これらのすべてのコンテキストについて詳しく説明しています。なお、この節では比較的単純な SELECT ステートメントのコンテキストに重点を置いて例を挙げて説明していますが、これらはこのコンテキスト以外でも使用できます。

使用法の概要、*関数パス* の重要性、および*関数選択* のアルゴリズムについては、 393 ページの『UDF とメソッドの概念』を参照してください。これらの概念は両方とも、*SQL 解説書* で詳しく説明されています。関数に対するデータ操作言語 (DML) 参照の分析には、関数選択のアルゴリズムを使用するので、これがどのように作用するのかを理解しておくことが重要です。

関数の参照

各関数の参照には、UDF または組み込み関数のいずれの場合も次の構文が含まれています。



上記の例で、 `function_name` は、修飾された関数名かまたは修飾されない関数名のいずれかで、引き数は 0 ～ 90 までの値となります。また、式に以下のものが含まれます。

- 修飾された、または修飾されない列名
- 定数

- ホスト変数
- 特殊レジスター
- パラメーター・マーカー。 (パラメーター・マーカーを使用する際の制限については、 *SQL 解説書* の中のパラメーター・マーカーの規則を説明している項を参照してください。)

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、その列を含む列参照が適切な効力範囲を持つことが必要です。結合で参照される表関数では、引き数が他の表や表関数にある列と関係している場合、その表や表関数は、参照を含む表関数の前に、FROM 文節で示されていなければならない、ということを意味します。表関数の引き数で列を使用する際の規則の詳細については、 *SQL 解説書* を参照してください。

関数呼び出しの例

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、上記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
```

関数でのパラメーター・マーカーの使用法

重要な制約事項には、パラメーター・マーカーが関係しています。単に次のようにコード化するだけでは不十分です。

```
BLOOP(?)
```

関数選択のロジックでは引き数がどのデータ・タイプになるのか分からないので、これにより参照を分析することはできません。 CAST という指定を使用して、パラメーター・マーカーにタイプを指定できます。たとえば INTEGER とすると、関数選択のロジックは次の処理を進めることができます。

```
BLOOP(CAST(? AS INTEGER))
```

修飾された関数参照の使用法

修飾された関数参照を使用する場合、 DB2 がそのスキーマに一致する関数のみを検索するようにします。たとえば、次のようなステートメントになります。

```
SELECT PABLO.BLOOP(COLUMN1) FROM T
```

これで PABLO というスキーマの BLOOP 関数のみが考慮されます。このとき、 SERGE というユーザーが BLOOP 関数を定義したことや、 BLOOP 組み込み関数の有無は問題になりません。たとえば、PABLO というユーザーがご使用のスキーマで次の 2 つの BLOOP 関数を定義したとします。

```
CREATE FUNCTION BLOOP (INTEGER) RETURNS ...
CREATE FUNCTION BLOOP (DOUBLE) RETURNS ...
```

このようにして BLOOP は PABLO スキーマ内で多重定義され、最適な BLOOP が、関数選択アルゴリズムにより引き数 COLUMN1 のデータ・タイプに従って選択されます。この場合、いずれの PABLO.BLOOP も数値引き数を取り、 COLUMN1 がいずれかの数値タイプでないならばそのステートメントは実行できません。一方 COLUMN1 が SMALLINT または INTEGER のいずれかである場合は、関数選択は最初の BLOOP に変換されますが、 COLUMN1 が DECIMAL、DOUBLE、REAL、または BIGINT ならば 2 番目の BLOOP が選択されます。

以下に、この例のポイントを挙げます。

1. この例は、引き数のプロモーションを示しています。最初の BLOOP は、INTEGER パラメーターによって定義されますが、 SMALLINT 引き数に渡すことができます。関数選択アルゴリズムは、組み込みデータ・タイプ (詳細については、 *SQL 解説書* を参照) 間のプロモーションをサポートし、 DB2 は、データ値の変換を適切に行います。
2. 何らかの理由のために SMALLINT または INTEGER 引き数を指定した 2 番目の BLOOP を呼び出したい場合は、ご使用のステートメント内で以下のようなアクションを明示的に実行しなければなりません。

   ```
   SELECT PABLO.BLOOP(DOUBLE(COLUMN1)) FROM T
   ```

3. 代わりに DECIMAL または DOUBLE 引き数を指定した最初の BLOOP を呼び出したい場合は、厳密な目的の下で以下のアクションのうちのどちらかを選んで実行します。

```
SELECT PABLO.BLOOP(INTEGER(COLUMN1)) FROM T
SELECT PABLO.BLOOP(FLOOR(COLUMN1)) FROM T
SELECT PABLO.BLOOP(CEILING(COLUMN1)) FROM T
SELECT PABLO.BLOOP(INTEGER(ROUND(COLUMN1,0))) FROM T
```

その他の関数については、 *SQL 解説書* を参照してください。 INTEGER 関数は、SYSIBM というスキーマの組み込み関数です。 FLOOR、CEILING、および ROUND 関数は、DB2 と共に出荷される UDF で、これらは他の多くの便利な関数と共に SYSFUN スキーマ中にあります。

修飾されない関数参照の使用法

一致する関数を検索する場合に、修飾された関数参照の代わりに修飾されない関数参照を使用すると、 DB2 は関数パスを使用して参照を修飾します。 DROP FUNCTION または COMMENT ON FUNCTION 関数の場合、それらが修飾されないものであれば、その参照は現行の許可 ID を使用して修飾されます。このため、*使用する関数パスが何であるか、またご使用の現行関数パスのスキーマに矛盾したパスがある場合はそれがどのようなものであるかを知っておくことが重要です。* たとえば、ユーザーが PABLO で、ユーザーの静的 SQL ステートメントが次のようになっているとします。ここで、COLUMN1 のデータ・タイプは INTEGER です。

```
SELECT BLOOP(COLUMN1) FROM T
```

そして、403ページの『修飾された関数参照の使用法』で例証されている 2 つの BLOOP 関数を作成しており、それらのうちいずれか 1 つを選択したいとします。次のデフォルトの関数パスが使用された場合、 SYSIBM または SYSFUN に矛盾する BLOOP がないと、 (COLUMN1 が INTEGER であるので) 最初の BLOOP 関数が選択されます。

```
"SYSIBM","SYSFUN","PABLO"
```

しかし、前に別の目的で作成したスクリプトを、コンパイルおよびバインドを行うために使用しているということを忘れていたとします。そのスクリプトでは、FUNCPATH パラメーターを明示的にコーディングして、現行の作業に適用されない以下の関数パスを別の理由で指定しました。

```
"KATHY","SYSIBM","SYSFUN","PABLO"
```

Kathy が自分専用に BLOOP 関数を定義していた場合は、関数選択は Kathy の関数を首尾よく解決でき、ユーザーのステートメントはエラーを起こさずに実行される。 DB2 は、ユーザーが自分自身の行っていることを理解していると見なすので、ユーザーには通知しません。ユーザーは、ステートメントからの誤った出力を識別し、要求される修正を行うことについて責任を負うことになります。

関数参照の要約

修飾された、および修飾されない関数参照のいずれの場合も、関数選択のアルゴリズムは適当な関数、つまり組み込みおよびユーザー定義関数の両方を調べます。これらの関数は以下のものを伴います。

- 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

(上記の説明中の*適当な関数* とは、修飾された参照の場合は*名前の付いたスキーマの関数*、修飾されない参照の場合は*関数パスのスキーマの関数* を意味します。) アルゴリズムにより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、判別要素として現行の関数パスが使用されます。このアルゴリズムについては、*SQL 解説書* で詳しく説明されています。

403ページの『修飾された関数参照の使用法』の最後に引用されている機能は、同じ関数に対する参照の場合でも*関数参照をネストできる* という注目すべきものです。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、列関数が含まれる場合はいくつかの制限があります。

非常に簡単な例を示します。

```
CREATE FUNCTION BLOOP (INTEGER) RETURNS INTEGER ...
CREATE FUNCTION BLOOP (DOUBLE) RETURNS INTEGER ...
```

ここで次の DML ステートメントについて考えます。

```
SELECT BLOOP( BLOOP(COLUMN1)) FROM T
```

COLUMN1 が DECIMAL または DOUBLE 列である場合は、内部の BLOOP 参照は、上で定義されている 2 番目の BLOOP に変換されます。この BLOOP は INTEGER を戻すので、外部の BLOOP は最初の BLOOP に変換されます。

また、COLUMN1 が SMALLINT または INTEGER 列である場合は、内部の BLOOP 参照は、上で定義されている最初の BLOOP に変換されます。この BLOOP は INTEGER を戻すので、外部の BLOOP も最初の BLOOP に変換されます。この場合、同じ関数に対してネストされた参照を見ていることになります。

関数参照についてさらに重要なポイントを以下に示します。

- 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ができます。除算 (/) 演算子を多重定義する UDF の例については、470ページの『例: 整数除算演算子』を参照してください。

- 関数選択アルゴリズムは、特別な関数に対して分析を行う際、参照のコンテキストを考慮しない。以下は、前の BLOOP 関数を少し修正したものです。

  ```
  CREATE FUNCTION BLOOP (INTEGER) RETURNS INTEGER ...
  CREATE FUNCTION BLOOP (DOUBLE) RETURNS CHAR(10)...
  ```

  ここで、次の SELECT ステートメントを作成したとします。

  ```
  SELECT 'ABCDEFG' CONCAT BLOOP(SMALLINT_COL) FROM T
  ```

  SMALLINT 引き数を使用して分析を行った結果、最適な関数は上で定義されている最初の BLOOP であるので、 CONCAT の 2 番目のオペランドはデータ・タイプ INTEGER に変換されます。 CONCAT にはストリング引き数が必要なので、ストリングは実行できません。最初の BLOOP がなければ、もう一方の BLOOP が選択され、ステートメントは正常に実行されます。

  ステートメントを実行できなくする他のタイプのコンテキスト上の矛盾は、指定された関数参照が、スカラー関数か列関数を必要とするコンテキストにおいて表関数に変換される場合です。また、逆の場合が生じることもあります。参照が、表関数が必要なときにスカラー関数や列関数に変換される場合です。

- UDF とメソッドは、 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る。 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 値を具体化します。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。したがっ

て、396ページの『例: ストリングの検索』で定義されている FINDSTRING を参照すると、以下の両方がプログラムで有効になります。

```
... 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 値全体が具体化されることはありません。代わりに、LOB LOCATOR が UDF に渡され、UDF は特殊な UDF API を使用して、 LOB 値の実際のバイトを操作します (詳細については、459ページの『UDF のパラメーターや結果としての LOB ロケーターの使用』を参照してください)。

  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。この機能は、非分離として定義されている UDF に限定されます。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。

- UDF も特殊タイプを使用してパラメーターまたは結果として定義できる。 (前の例で説明済み。) DB2 は、特殊タイプのソース・データ・タイプの書式で値を UDF に渡します。

  ホスト変数から発生し、UDF の引き数として使用され、特殊タイプとして定義される対応したパラメーターを持つ特殊タイプの値は、**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**。特殊タイプのホスト言語タイプはありません。一方、DB2 の強力タイピングにはホスト言語が必要です。そうしないと、結果があいまいになります。 BLOB で定義される BOAT 特殊タイプおよび 398ページの『例: UDT パラメーターを指定した外部関数』中にある引き数として BOAT タイプのオブジェクトを取る BOAT_COST UDF について考えてみます。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、ご使用の関数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ります。

第15章 ユーザー定義関数 (UDF) とメソッドの作成

説明 409
DB2 と UDF 間のインターフェース . . . 411
DB2 から UDF に渡される引き数 . . . 411
UDF 引き数の使用の要約 424
SQL データ・タイプの UDF への受け渡し方法 426
32 ビット・プラットフォームおよび 64 ビット・プラットフォームでのスクラッチパッドの作成 434
UDF インクルード・ファイル: sqludf.h 435
Java ユーザー定義の関数の作成および使用 436
Java UDF のコーディング 437
Java UDF の実行の仕方の変更 438
Java の表関数実行モデル 439
OLE オートメーション UDF の作成 . . . 441
OLE オートメーション UDF の作成と登録 442
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 443
SQL データ・タイプの OLE オートメーション UDF への受け渡し方法 443
BASIC と C++ での OLE オートメーション UDF のインプリメンテーション . . 445
BASIC での OLE オートメーション UDF 445
C++ での OLE オートメーション UDF 446
OLE DB 表関数 448
OLE DB 表関数の作成 449
完全修飾行セット名 451
OLE DB Provider のサーバー名の定義 452
ユーザー・マッピングの定義 453
サポートされる OLE DB データ・タイプ 453
スクラッチパッドに関する考慮事項. . . . 456
表関数に関する考慮事項 458
表関数のエラー処理 458
スカラー関数のエラー処理. 459
UDF のパラメーターや結果としての LOB ロケーターの使用. 459
LOB ロケーター使用のシナリオ 464
コーディングに関するその他の考慮事項 . . 464
ヒントとアドバイス 465
UDF に関する制限と警告 467
UDF コードの例 470
例: 整数除算演算子 470
例: CLOB の折り返し、母音の検出 . . . 474
例: カウンター. 478
例: 天気表関数. 480
例: LOB ロケーターを使用する関数 . . 488
例: BASIC でのカウンター OLE オートメーション UDF 491
例: C++ でのカウンター OLE オートメーション UDF. 492
例: BASIC でのメール OLE オートメーション表関数 495
UDF のデバッグ 497

説明

この節では、UDF とメソッドの作成方法について説明します。 UDF のコーディング規則とメソッドのコーディング規則はほとんど同じですが、次の点が異なります。

- DB2 は各メソッドをそれぞれ別々の構造型に関連付けるため、 DB2 からメソッドへと渡される最初の引き数は常に、そのメソッドが呼び出される構造型のインスタンスです。
- UDF とは異なり、メソッドは表を戻すことができません。 FROM 文節の引き数としてメソッドを呼び出すことはできません。

© Copyright IBM Corp. 1993, 2001

UDF を作成する際の指針とメソッドを作成する際の指針は、前述の点を除いては同じであるため、 UDF とメソッドの作成に関するこの章のその他の説明は、 UDF についてのみ述べられている場合でも、 UDF とメソッドの両方にあてはまります。

1 つの単純ロジックだけを含んでいるというような小さな UDF の場合は、 *SQL 形式の UDF* の使用を考慮してください。 SQL 形式の UDF を作成するには、 SQL を使用して作成されたメソッド本体を組み込む CREATE FUNCTION ステートメントまたは CREATE METHOD ステートメントを発行してください。外部 UDF を指すことはしないでください。 SQL 形式の UDF を使用すれば、外部言語や外部コンパイラーを使用しなくても、単一のステップだけで UDF を宣言および定義できます。 SQL 形式の UDF を使用すれば、パフォーマンスを向上できる可能性もあります。 DB2 最適化プログラムにアクセス可能な SQL を使用して、メソッド本体が作成されているからです。

次の例は、 SQL 形式の UDF を作成するシンプルな CREATE FUNCTION ステートメントを示したものです。

```
CREATE FUNCTION tan(double x)
RETURNS double
NO EXTERNAL ACTION
DETERMINISTIC
LANGUAGE SQL
CONTAINS SQL
RETURN sin(x) / cos(x);
```

SQL 形式の関数に関する詳細については、 *SQL 解説書* を参照してください。

まず、DB2 と UDF 間のインターフェースに関する予備知識を説明し、続いて UDF をインプリメントする方法について説明します。 UDF の定義に関する主な考慮事項の 1 つとしてスクラッチパッドの有無が重要視されます。

以下に、この項で使用する一般的な考慮事項を示します。

- UDF の定義と使用に関する重要事項については 389ページの『第14章 ユーザー定義関数 (UDF) およびメソッド』で説明し、ここでは説明しません。この項では UDF をインプリメントする方法を中心に説明します。
- C、C++、または Java で作成された*外部 UDF* をインプリメントするには、以下の手順を実行しなければなりません。
  - UDF を作成する
  - UDF をコンパイルする
  - UDF をリンクする
  - CREATE FUNCTION ステートメントを使って UDF を登録する
  - UDF をテストおよびデバッグする

  UDF のコンパイルとリンクに関する情報は、*アプリケーション構築の手引き* にあります。

- 441ページの『OLE オートメーション UDF の作成』で説明されているように、 OLE (オブジェクトのリンクと埋め込み) を使用することにより、UDF を呼び出すことができます。
- CREATE FUNCTION ステートメントだけを使って、 *OLE DB 表関数* (OLE DB データ・ソースからの表を戻す関数) を定義することができます。 OLE DB 表関数に関する詳細については、448ページの『OLE DB 表関数』を参照してください。

ソース派生 UDF (外部 UDF とは異なる) は、別々のコード形式でインプリメントする必要はありません。このような UDF は、他の属性と一緒に、そのソース関数と同じようにインプリメントされます。

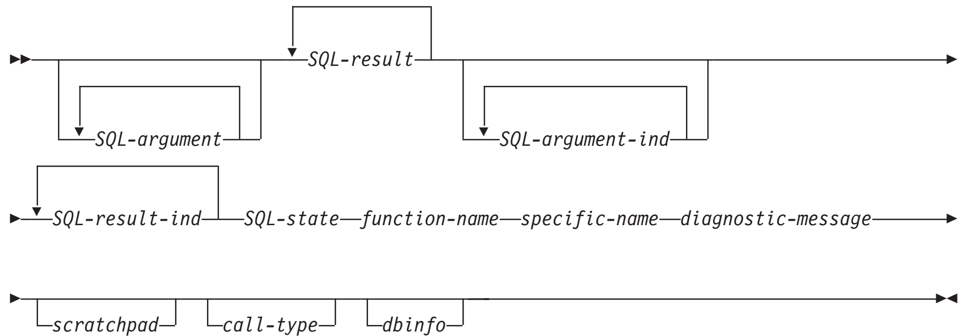
DB2 と UDF 間のインターフェース

この項では、DB2 と UDF の間のインターフェースについて特定の詳細情報と、インターフェースを管理できるようにする `sqludf.h` インクルード・ファイルについて説明します。このインクルード・ファイルは C および C++ で作成された UDF にのみ適用されます。 Java による UDF のコーディングについては、437ページの『Java UDF のコーディング』を参照してください。

DB2 から UDF に渡される引き数

関数への DML 参照で指定される SQL 引き数に加えて、 DB2 は追加の引き数を外部 UDF に渡します。 C と C++ の場合、これらの引き数はすべて 412ページの『UDF への引き数の受け渡し』に示されている順序で渡されます。 Java UDF は、 *SQL 引き数* と *SQL 結果* 引き数の 2 つの引き数のみを取りますが、他の方式を呼び出してそれ以外の情報にアクセスすることもできます。 Java UDF では、以降に説明する結果の *SQL 状態* および*診断メッセージ* の引き数についても、同じ制限が課されます。 Java による UDF のコーディングについては、437ページの『Java UDF のコーディング』を参照してください。

引き数を UDF へ渡す構文



注: 外部関数に渡される上記の各引き数は値へのポインターであり、実際の値ではありません。

以下に各引き数について説明します。

SQL 引き数 (SQL-argument)

この引き数は、UDF を呼び出す前に DB2 により設定されます。この値は *n* 回繰り返されます。 *n* は関数参照で指定された引き数の数です。 これらの各引き数の値は、関数呼び出しで指定された式から取得されます。その値は、CREATE FUNCTION ステートメント中で該当するパラメーター定義のデータ・タイプで表されます。これらのデータ・タイプが C 言語構成にマップする方法については、 426ページの『SQL データ・タイプの UDF への受け渡し方法』で説明します。

DB2 は、データ・タイプとサーバーのプラットフォームに応じて、*SQL-argument* が表すデータの位置を調整します。

SQL 結果 (SQL-result)

この引き数は、DB2 に戻る前に UDF により設定されます。スカラー関数の場合、SQL 結果は 1 つだけあります。表関数の場合、CREATE FUNCTION ステートメントの RETURNS TABLE 文節で定義される関数の各列ごとに、 1 つの SQL 結果があります。それぞれの SQL 結果は、RETURNS TABLE 文節で定義される列の位置に対応します。つまり、最初の SQL 結果引き数は、RETURNS TABLE 文節で定義される最初の列に対応します。 2 番目以降の SQL 結果についても同様です。

スカラー関数と表関数のどちらの場合でも、 DB2 はバッファーを割り振り、そのアドレスを UDF に渡します。 UDF はそれぞれの結果の値をバッファーに入れます。 DB2 は、データ・タイプで示される値を含むのに十分なバッファーを割り振ります。スカラー関数の場合、このデータ・タイプは、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、データ・タイプは、

RETURNS TABLE(...) 文節で定義されます。これらのデータ・タイプが C 言語構成にどのようにマップされるかについては、 426ページの『SQL データ・タイプの UDF への受け渡し方法』を参照してください。

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 *DB 情報* 引き数を参照してください。

戻される各値については (つまり、スカラー関数の場合は単一の値、また一般に、表関数の場合は複数の値について)、UDF コードが、結果のデータ・タイプと長さに必要なバイト数よりも多くのバイトを戻さないようにしてください。 DB2 は、UDF 本体が結果バッファーの上限より数バイト多く書き込んだかどうかを判別しようとし、 SQLCODE -450 (SQLSTATE 39501) を戻します。ただし、DB2 により検出されない、UDF による大幅な上書きは、予期しない結果や異常終了になる可能性があります。

DB2 は、データ・タイプとサーバーのプラットフォームに応じて、 *SQL-result* が表すデータの位置を調整します。

SQL 引き数標識 (SQL-argument-ind)

この引き数は、UDF を呼び出す前に DB2 により設定されます。 UDF は、この引き数を使用して *SQL 引き数* がヌルかどうかを判別します。 *n* 番目の *SQL 引き数標識* は、 *n* 番目の *SQL 引き数標識* (上述) に対応します。 この引き数には以下の値のうちの 1 つが入ります。

0 ヌル以外の引き数があります。

-1 ヌルの引き数があります。

関数を NOT NULL CALL で定義すると、UDF 本体はヌル値に対する検査を行う必要がありません。ただし、NULL CALL で定義されると引き数はヌルとなる可能性があるため、 UDF はその検査を行う必要があります。

標識は SMALLINT 値の形式をとります。これは、426ページの『SQL データ・タイプの UDF への受け渡し方法』で説明するように UDF で定義することができます。 DB2 は、データ・タイプとサーバーのプラットフォームに応じて、 *SQL-argument-ind* が表すデータの位置を調整します。

SQL 結果標識 (SQL-result-ind)

この引き数は、DB2 に戻る前に UDF により設定されます。また、この引き数は各 *SQL 結果* 引き数について 1 つ存在しています。

UDF は、この引き数を使用して特定の結果値がヌルかどうかを示します。

0 または正の整数

結果はヌル値ではありません。

負の整数

結果はヌル値です。詳細については、負の SQL 結果標識値の解釈を参照してください。

負の SQL 結果標識値の解釈

以下の場合に、DB2 は関数結果をヌル (-2) 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ヌルになっている場合
- SQL 結果標識が負の値である場合

これは、関数を NOT NULL CALL オプションで定義した場合にも当てはまります。

関数を NOT NULL CALL で定義した場合でも、UDF 本体は結果の標識を設定しなければなりません。たとえば、分母がゼロである場合、除算関数は結果をヌルに設定することができます。

標識は SMALLINT 値の形式をとります。これは、426ページの『SQL データ・タイプの UDF への受け渡し方法』で説明するように UDF で定義することができます。

RESULT 列リストを用いた表関数最適化が UDF により使用されている場合、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。

DB2 は、データ・タイプとサーバーのプラットフォームに応じて、*SQL-result-ind* が表すデータの位置を調整します。

SQL 状態 (SQL-state)

この引き数は、DB2 に戻る前に UDF により設定されます。これは CHAR(5) 値の形式をとります。 UDF では、426ページの『SQL データ・タイプの UDF への受け渡し方法』で説明するように CHAR(5) で引き数を定義し、 UDF がその引き数を使って警告またはエラー条件を知らせることができるようにしてください。この引き数には、関数の呼び出し時に '00000' の値が入ります。UDF には以下の値を設定できます。

00000 この関数コードは警告状態やエラー状態を検出しません。

01H*xx* この関数コードは警告状態を検出しました。その結果、SQL 警告の SQLCODE +462 (SQLSTATE 01H*xx*) が戻されます。ここで、'*xx*' は任意のストリングです。

02000 表関数への FETCH 呼び出しの場合にのみ有効です。表にこれ以上行がないことを意味します。

38502 UDF 本体が SQL 呼び出しを行おうとして、エラーの SQLCODE -487 (SQLSTATE 38502) を受け取った場合の特殊な値。 UDF では SQL を使用できないため、これと同じエラーを DB2 を介して戻すようにします。

その他の 38*xxx*
この関数コードはエラー状態を検出しました。その結果、SQL エラーの SQLCODE -443 (SQLSTATE 38*xxx*) が戻されます。ここで、'*xxx*' は任意のストリングです。 380*xx* ～ 384*xx* は、 SQL92 国際標準のドラフト拡張用に予約されているため、使用しないでください。また、385*xx* は、 IBM によって予約されているため、使用しないでください。

その他の値はエラー状態として扱われ、その結果 SQLCODE -463 (SQLSTATE 39001) が発生します。

関数名 (function-name)
この引き数は、UDF を呼び出す前に DB2 により設定されます。これは、DB2 から UDF コードに渡される、修飾された関数名です。この変数は VARCHAR(27) 値の形式をとります。 UDF では、必ず VARCHAR(27) で引き数を定義してください。詳細については、426ページの『SQL データ・タイプの UDF への受け渡し方法』を参照してください。

渡される関数名の形式は以下のとおりです。

```
<schema-name>.<function-name>
```

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、複数の外部関数に同じ UDF 本体を使用してもその呼び出し時にはそれらの関数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` の中にあり、戻される関数名が `obj.op.rotate` の場合、スキーマ名が `obj` なのか `obj.op` なのかがはっきりしません。

特定名 (specific-name)
この引き数は、UDF を呼び出す前に DB2 により設定されます。これは、DB2 から UDF コードに渡される関数の特定の名前です。この変数は VARCHAR(18) 値の形式をとります。 UDF では、必ず VARCHAR(18) で引

き数を定義してください。詳細については、426ページの『SQL データ・タイプの UDF への受け渡し方法』を参照してください。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 FUNCTION 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。

関数名 引き数の場合と同じように、この値を渡すのは、UDF を呼び出している特定の関数を明確に区別するためです。

診断メッセージ (diagnostic-message)

この引き数は、DB2 に戻る前に UDF により設定されます。 UDF 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。これは VARCHAR(70) 値の形式をとります。 UDF では、必ず VARCHAR(70) で引き数を定義してください。詳細については、426ページの『SQL データ・タイプの UDF への受け渡し方法』を参照してください。

上述した *SQL-state* 引き数を用いて UDF がエラーまたは警告のいずれかを戻す場合、ここには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、UDF を呼び出す前に最初の文字をヌルに設定します。 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により決まります。これは、DB2 がトークンを切り捨てて、 SQLCA によって指定された合計トークン長の制限に合わせることがあるからです。 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、テキスト内には使用しないでください。

UDF コードは、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。 DB2 は、UDF 本体がこのバッファーの上限を超えて定義されたかどうかを、文字 SQLCODE -450 (SQLSTATE 39501) により判別しようとします。ただし、UDF が上書きされると、予測不可能な結果や異常終了を引き起こして DB2 により検出されない場合があります。

DB2 では、UDF から DB2 に戻されるメッセージ・トークンがデータベースと同じコード・ページにあることを前提とします。使用している UDF がこの場合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、 UDF はメッセージ・トークンを任意のコード・ページに戻します。

スクラッチパッド (scratchpad)

この引き数は、UDF を呼び出す前に DB2 により設定されます。これは、UDF に対する CREATE FUNCTION ステートメントに SCRATCHPAD キーワードを指定した場合にのみ存在します。この引き数は、以下のエレメントを持ち、任意の LOB データ・タイプの値を渡すために使わ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 2 進数の 0 に初期化されます。

  スカラー関数の場合、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 DB2 による参照や修正を行うことはできません。

  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 呼び出しより前に上記のように初期化されます。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 呼び出しごとに上記のように初期化され、スクラッチパッドの内容は完全に OPEN 呼び出しの合間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。 OPEN 呼び出しの合間に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 と同じタイプを使って UDF 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。詳細については、426ページの『SQL データ・タイプの UDF への受け渡し方法』を参照してください。

UDF コードがスクラッチパッド・バッファーの外側で変更を行わないことを確認してください。 DB2 は、UDF 本体がこのバッファーの上限を超えて定義されたかどうかを、文字 SQLCODE -450 (SQLSTATE 39501) により判別しようとしますが、 UDF が大幅に上書きされると予測不可能な結果や異常終了を引き起こし、その結果 DB2 がエラーを出さないこともあります。

スクラッチパッドを使用するスカラー UDF が副照会で参照される場合、 DB2 は副照会の呼び出しの合間にスクラッチパッドをリフレッシュすることに決めることがあります。 UDF で FINAL CALL が指定されている場合、このリフレッシュは、*最終呼び出し* が行われた後に起こります。

DB2 は、データ・フィールドの位置がどのデータ・タイプのストレージでも合うよう、スクラッチパッドを初期化します。その結果、スクラッチパッド構造の一部または全体 (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。スクラッチパッドの宣言およびそれへのアクセスに関する詳細については、 434ページの『32 ビット・プラットフォームおよび 64 ビット・プラットフォームでのスクラッチパッドの作成』を参照してください。

呼び出しタイプ (call-type)

この引き数は、もし存在するなら、UDF を呼び出す前に DB2 により設定されます。スカラー関数の場合、この引き数は CREATE FUNCTION ステートメントに FINAL CALL を指定する場合にのみ存在しますが、表関数の場合には常に存在します。これは、*スクラッチパッド* 引き数の後に指定します。*スクラッチパッド* 引き数がないときは *診断メッセージ* 引き数の後に指定します。この引き数は INTEGER 値の形式をとります。 UDF の引き数定義が、INTEGER に適していることを確認してください。詳細については、426ページの『SQL データ・タイプの UDF への受け渡し方法』を参照してください。

現在使用可能なすべての値が以下にリストされていますが、「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ではなく、予期されるすべての値を明示的にテストする切り替えまたは CASE ステートメントを UDF に含める必要があります。これは、将来付加的な呼び出しタイプを追加できるようにするため、および明示的に条件 C をテストしない場合には、新しい条件が追加されると問題が発生するからです。

注:

1. すべての呼び出しタイプで、 UDF が *SQL 状態* と*診断メッセージ* の戻り値を設定するのが適切な場合もあります。 この情報は、それぞれの呼び出しタイプの後に続く記述の中では繰り返されません。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. インクルード・ファイル `sqludf.h` は、UDF で使用するためのものであり、 435ページの『UDF インクルード・ファイル: sqludf.h』で説明されています。記号を含むこのファイルは以下の呼び出しタイプの値を定義し、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、*呼び出しタイプ (call-type)* には次のものが含まれます。

-1 これは、このステートメントに対する UDF への *FIRST 呼び出し*です。スクラッチパッド (存在する場合) は、 UDF が呼び出されるときに 2 進数のゼロに設定されます。すべての引き数値が渡され、UDF 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 に対する FIRST 呼び出しは、応答を作成して戻すことが期待されているため、NORMAL 呼び出しに似ています。

SCRATCHPAD が指定されていても FINAL CALL が指定されていない場合、 UDF は最初の呼び出しを識別するためにこの呼び出しタイプ引き数を使用しないことに注意してください。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。

0 これは *NORMAL 呼び出し* です。すべての SQL 入力値が渡され、UDF が結果を作成して戻すことが期待されています。 UDF が *SQL 状態* および*診断メッセージ* 情報を戻す可能性もあります。

1 これは *FINAL 呼び出し* です。すなわち、 *SQL 引き数* の値も *SQL 引き数標識* の値も渡されず、これらの値が予測不可能な結果となるかどうかを調べます。 スクラッチパッドも渡される場合は、この値は前の呼び出し時のままです。 UDF はこの時点でリソースを解放することになります。

リソースの解放

スカラー UDF 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ことになります。 SCRATCHPAD も指定され、リソースを追跡するために使用される場合は、 FINAL CALL が UDF に指定されている場合には、FINAL 呼び出しがリソースを解放するのが自然です。 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、*呼び出しタイプ (call-type)* には次のものがあります。

-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、スクラッチパッドは 2 進ゼロに設定されます。引き数値が表関数に渡され、メモリーの獲得または他の 1 回限りのリソース初期化の実行が選択されます。これは OPEN 呼び出しではなく、この呼び出しの後に OPEN 呼び出しが続くことに注意してください。 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。

-1 これは、*OPEN 呼び出し* です。 NO FINAL CALL が指定される場合には、スクラッチパッド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。

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。スクラッチパッドが 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。

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。

結合や副照会が含まれている場合、 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。

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。

リソースの解放

獲得したリソースを解放する UDF 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 の組に 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 の組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。 (表関数が副照会または結合関数中にある場合、 DB2 最適化プログラム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 の組が複数回指定される可能性が高いことに注意してください。)

dbinfo (DB 情報)

この引き数は、UDF を呼び出す前に DB2 により設定されます。これは、UDF に対する CREATE FUNCTION 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。この引き数は、ヘッダー・ファイル `sqludf.h` で定義された `sqludf_dbinfo` 構造で、このヘッダー・ファイルについては、435 ページの『UDF インクルード・ファイル: sqludf.h』で説明しています。この構造内で名前と ID を含む変数は、 DB2 のこのリリースで指定可能な最長の値より長いことがありますが、将来のリリースと互換性を持つよう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。上述の*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。 ヌル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)

   アプリケーション実行時許可 ID。このフィールドは、128 文字の長 ID です。ヌル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. データベース・コード・ページ (codepg)

   2 つの 48 バイト長の構造が合併したもので、1 つは DB2 ユニバーサル・データベースが使用し、もう 1 つは将来の使用のために予約されています。 DB2 ユニバーサル・データベースが使用する構造には、以下のフィールドがあります。

   a. SBCS。 1 バイトのコード・ページで、無符号長整数です。
   b. DBCS。 2 バイトのコード・ページで、無符号長整数です。
   c. COMP。複合コード・ページで、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。ヌル終了符や埋め込みは含まれません。上述の*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、UDF 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。ヌル終了符や埋め込みは含まれません。上述の*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。 *スキーマ名* とこのフィールドが合わさって、完全修飾表名になります。

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。ヌル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 (MVS/ESA 版) または (OS/390 版)

      ARI SQL/DS

      QSQ DB2 ユニバーサル・データベース (AS/400 版)

      SQL DB2 ユニバーサル・データベース
    - *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. 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・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_MVS OS/390

    SQLUDF_PLATFORM_NT Windows NT

    SQLUDF_PLATFORM_OS2 OS/2

    SQLUDF_PLATFORM_SUN Solaris 実行環境版

    SQLUDF_PLATFORM_WINDOWS
    Windows 95 および Windows 98

    SQLUDF_PLATFORM_UNKNOWN
    不明なプラットフォーム

    上記のリストに含まれていないその他のプラットフォームについては、`sqludf.h` ファイルの内容を参照してください。

14. 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。

15. 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。 2 文字の長さに定義されています。

16. プロシージャー ID (procid)

    ルーチンの呼び出し側がカタログ化されたストアード・プロシージャーの場合、 DBINFO 構造における、プロシージャーまたは関数に渡される procid フィールドの値はゼロです。この場合、procid の値は、SYSCAT.PROCEDURES 表の PROCEDURE_ID 列に記録されている呼び出し側プロシージャーの ID になります。その他の場合は、procid フィールドについて返される値は 0 です。

17. 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。 32 文字の長さに定義されています。

18. 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。これがスカラー関数である場合、このポインターはヌルです。

    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。

19. 固有のアプリケーション ID (appl_id)

このフィールドは、ヌル文字で終了する 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 への接続を一意的に識別します。このフィールドは、データベースに接続するたびに再生成されます。

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
```

20. 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。 20 文字の長さに定義されています。

UDF 引き数の使用の要約

次に、上記の引き数について要約し、それらを DB2 と外部 UDF 間のインターフェースでどのように使用するのかを説明します。

スカラー関数の場合、引き数は次のとおりです。

- *SQL 引き数 (SQL-argument)*

  関数参照で識別された値を DB2 から UDF に渡します。各 SQL 引き数ごとに 1 つあります。
- *SQL 結果 (SQL-result)*

  UDF により生成された結果値を、DB2 および関数参照が行われた SQL ステートメントに渡します。
- *SQL 引き数標識 (SQL-argument-ind)*

  この変数の位置は *SQL 引き数* に対応し、特定の引き数がヌルであるかどうかを UDF に知らせます。各 *SQL 引き数* ごとに 1 つあります。
- *SQL 結果標識 (SQL-result-ind)*

  UDF は、この引き数を使用して、 *SQL 結果* の関数結果にヌルが入っているかどうかを DB2 に報告します。
- *SQL 状態 (SQL-state) と診断メッセージ (diagnostic-message)*

  UDF は、この引き数を使用して、例外情報を DB2 に送ります。
- *関数名 (function-name) と特定名 (specific-name)*

  DB2 は、この引き数を使用して、参照されている関数が何であるかを UDF に伝えます。
- *スクラッチパッド (scratchpad) と呼び出しタイプ (call-type)*

DB2 は、この引き数を使用して、呼び出しの合間の UDF 状態の保存を管理します。*スクラッチパッド (scratchpad)* は、 DB2 により作成および初期化された後 UDF により管理されます。 DB2 は、*呼び出しタイプ (call-type)* 引き数を使って呼び出しのタイプを UDF に知らせます。

- *DB 情報 (dbinfo)*

  DB2 によって UDF に渡される構造で、追加情報を含みます。

表関数は論理的には参照元の SQL ステートメントに表を戻しますが、 DB2 と表関数の間の物理インターフェースは行単位です。表関数の場合、引き数は次のとおりです。

- *SQL 引き数 (SQL-argument)*

  関数参照で識別された値を DB2 から UDF に渡します。この引き数は、FETCH 呼び出しに対して、 OPEN 呼び出しと FIRST 呼び出しの場合と同じ値を持ちます。各 SQL 引き数に 1 つあります。

- *SQL 結果 (SQL-result)*

  UDF によって戻されている行の個々の列値を戻すのに使用されます。 CREATE FUNCTION ステートメントの RETURNS TABLE (...) 文節で定義される結果列の値ごとに、いずれかの引き数があります。

- *SQL 引き数標識 (SQL-argument-ind)*

  この引き数の位置付けは *SQL 引き数 (SQL-argument)* 値に対応し、特定の引き数がヌルであるかどうかを UDF に知らせます。各 SQL 引き数に 1 つあります。

- *SQL 結果標識 (SQL-result-ind)*

  UDF は、この引き数を使用して、表関数の出力行で戻された個々の列値がヌルであるかどうかを DB2 に報告します。この引き数の位置は、*SQL 結果 (SQL-result)* 引き数に対応します。

- *SQL 状態 (SQL-state) と診断メッセージ (diagnostic-message)*

  UDF は、この引き数を使用して、例外情報と end-of-table (表の終わり) 条件を DB2 に送ります。

- *関数名 (function-name) と特定名 (specific-name)*

  DB2 は、この引き数を使用して、参照されている関数が何であるかを UDF に伝えます。

- *スクラッチパッド (scratchpad) と呼び出しタイプ (call-type)*

  DB2 は、この引き数を使用して、呼び出しの合間の UDF 状態の保存を管理します。*スクラッチパッド (scratchpad)* は、 DB2 により作成および初期化された後 UDF により管理されます。 DB2 は、*呼び出しタイプ (call-type)* 引き数を使って呼び出しのタイプを UDF に知らせます。表関数の場合、これらの呼び出しタイプは OPEN、FETCH、CLOSE で、オプションで FIRST と FINAL にすることもできます。

- *DB 情報 (dbinfo)*

  DB2 によって UDF に渡される構造で、追加情報を含みます。

UDF、*SQL 結果*、*SQL 結果標識*、および *SQL 状態* の通常の値出力は、 DB2 から UDF に渡される引き数を使って DB2 に戻されることに注意してください。 UDF は、関数の意味では何も戻さないように定義されています (つまり、関数の戻りタイプは void です)。次の例の `void` 定義と `return` ステートメントを参照してください。

```
#include ...
 void SQL_API_FN divid(
      ... arguments ... )
{
      ... UDF body ...
      return;
}
```

上記の例で、`SQL_API_FN` は、サポートされるオペレーティング・システムごとに異なる関数の呼び出し規則を指定するマクロです。 このマクロは、ストアード・プロシージャーや UDF を作成する場合に必要です。

UDF のプログラミング例については、470ページの『UDF コードの例』を参照してください。

SQL データ・タイプの UDF への受け渡し方法

この項では、UDF パラメーターと結果の両方に有効なタイプを識別し、対応する引き数を C や C++ 言語の UDF で定義する方法をそれぞれ指定します。 Java UDF の型定義については、658ページの『Java でサポートされている SQL データ・タイプ』を参照してください。 `sqludf.h` インクルード・ファイルとそこで定義されるタイプを使用すると、さまざまなデータ・タイプおよびコンパイラーに当てはまる言語変数および構造を自動的に生成できます。 たとえば、BIGINT では、SQLUDF_BIGINT データ・タイプを使用して、異なるコンパイラー間での 64 ビットの整数型の名前の違いを隠すことができます。このインクルード・ファイルについては、435ページの『UDF インクルード・ファイル: sqludf.h』で説明しています。

これは、引き数値の書式を管理する CREATE FUNCTION ステートメントで定義される各関数パラメーターのデータ・タイプです。 引き数のデータ・タイプからのプロモーションは、この書式でデータを受け取る必要はありません。 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。引き数のプロモーションについては、*SQL 解説書* で説明されています。

関数結果の場合、書式を定義する CREATE FUNCTION ステートメントの CAST FROM 文節で指定されるデータ・タイプとなります。 CAST FROM 文節がない場合は、RETURNS 文節で指定されるデータ・タイプが書式を定義します。

以下の例での CAST FROM 文節は、UDF 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、UDF は以下に示すように SMALLINT を生成するように定義しなければなりません。 CAST FROM データ・タイプは RETURNS データ・タイプに対してキャスト*可能* でなければならないため、任意に他のデータ・タイプを選ぶことはできません。データ・タイプ間のキャストについては、 *SQL 解説書* で説明されています。

以下に、SQL タイプとその C 言語での表示を示します。 Java の SQL タイプ表記のリストについては、 658ページの『Java でサポートされている SQL データ・タイプ』を参照してください。 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 UDF 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 `short` として表します。

  整数の UDF パラメーターを定義する際は、SMALLINT ではなく INTEGER を使用するようにしてください。これは、DB2 が SMALLINT 引き数を INTEGER にプロモートしないためです。たとえば、UDF を次のように定義するとします。

  ```
  CREATE FUNCTION SIMPLE(SMALLINT)...
  ```

  以下に例を示します。

  ```
  short    *arg1;          /* example for SMALLINT */
  short    *arg1_null_ind; /* example for any null indicator */
  ```

  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。 上の例で 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 `sqlint32` として表します。 DB2 インクルード・ファイル `sqlsystm.h` は、このタイプを各プラットフォームに合った 32 ビットの整数として定義します。

  以下に例を示します。

  ```
  sqlint32 *arg2;          /* example for INTEGER */
  ```

- BIGINT

  正しい例。 C で `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 を指定して、定義を選出する必要があります。

- 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・タイプ (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。

  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、 UDF 内の値に算術計算を実行しやすくなるということです。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。

- REAL

  正しい例。 C で float として表します。

  以下に例を示します。

```
float *result;          /* example for REAL */
```

- DOUBLE、DOUBLE PRECISION、または FLOAT

  正しい例。 C で double として表します。

  以下に例を示します。

  ```
  double  *result;        /* example for DOUBLE   */
  ```

- 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 char...[n+1] として表します (これは、C の NULL 終了ストリングなので、最後の文字はヌルつまり X'00' です)。

  以下に例を示します。

  ```
  char     arg1[14];      /* example for CHAR(13)   */
  char    *arg1;          /* also perfectly acceptable */
  ```

  CHAR(n) パラメーターの場合、DB2 は常にデータの n バイトをバッファーに移動し、 *n+1* バイトをヌルに設定します。 RETURNS CHAR(n) 値の場合、DB2 は常に n バイトを受け取り、 *n+1* バイトを無視します。 この場合は、最初の n 文字にヌル文字を誤って含めないように注意してください。そうしないと、DB2 はデータの通常部分としてのみこの文字を認識し、後になって外見的に異例の結果を生じることがあります。

  FOR BIT DATA が指定されている場合、 UDF 内で関数を処理する通常の C ストリングを使用する際は注意してください。これらの関数の多くはストリングの区切り文字としてヌルを探すので、ヌル文字 (X'00') がデータ値の中にあっても正しいことになります。

  文字の UDF パラメーターを定義する際は、CHAR よりも VARCHAR を使用するようにしてください。これは、DB2 が VARCHAR 引き数を CHAR にプロモートしないためです。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。 上の例で、'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。 C で次のような構造として表します。

```
struct sqludf_vc_fbd
{
   unsigned short length;         /* length of data */
   char           data[1];        /* first char of data */
};
```

[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。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、ヌル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 UDF にパラメーターとして正しく渡されます。 RETURNS 文節の場合、UDF に渡される長さはバッファーの長さです。 UDF 本体は、構造変数 length を使っ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 char...[n+1] として表します。 (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 はヌルを (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。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) 値の場合、UDF 本体は実際の値をヌルを使って区切る必要があります。これは、DB2 がこのヌル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 perfectly acceptable */
  ```

- GRAPHIC(n)

  正しい例。 C で sqldbchar[n+1] として表します。(これは NULL 終了グラフィック・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プラットフォーム上では、 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。 これら 2 つのデータ・タイプの詳細については、639ページの『C および C++ での wchar_t または sqldbchar データ・タイプの選択』を参照してください。

  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 バイトをヌルに設定します。 DB2 から UDF 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、640ページの『C および C++ での WCHARTYPE プリコンパイラー・オプショ

ン』で説明する 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) 値の場合、DB2 は常に n 個の 2 バイト文字を受け取り、それ以降のバイトを無視します。

グラフィック UDF 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、UDF 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。上の例で、*graphic_literal* は VARGRAPHIC データとして解釈されるリテラル DBCS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 perfectly acceptable */
```

- VARGRAPHIC(n)

  正しい例。 C で `sqldbchar[n+1]` として表します。 (これは NULL 終了グラフィック・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プラットフォーム上では、 `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。 これら 2 つのデータ・タイプの詳細については、639ページの『C および C++ での wchar_t または sqldbchar データ・タイプの選択』を参照してください。

  VARGRAPHIC(n) パラメーターの場合、 DB2 はグラフィック・ヌルを (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。グラフィック・ヌルは、グラフィック・ストリングの最後の文字の全バイトに 2 進ゼロ ('¥0's) が含まれていることを示します。 DB2 から UDF 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、640ページの『C および C++ での WCHARTYPE プリコンパイラー・オプション』で説明する 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) 値の場合、 UDF 本体は実際の値をグラフィック・ヌルを使って区切る必要があります。これは、DB2 がこのグラフィック・ヌル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 perfectly acceptable  */
```

- LONG VARGRAPHIC

  正しい例。 C で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 が長さ 2 バイトとして定義されているプラットフォームでは、上記の例の sqldbchar の箇所に 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。 これら 2 つのデータ・タイプの詳細については、639ページの『C および C++ での wchar_t または sqldbchar データ・タイプの選択』を参照してください。

[1] は、単にコンパイラーに対する配列を示しています。グラフィック文字を 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ることになります。

これらは NULL 終了グラフィック・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 UDF にパラメーターとして明示的に渡されます。 DB2 から UDF 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、640ページの『C および C++ での WCHARTYPE プリコンパイラー・オプション』で説明する 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 文節の場合、UDF に渡される長さはバッファーの長さです。 UDF 本体は、構造変数 length を使っ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   */
```

- DATE

  正しい例。 C で CHAR(10) として、つまり char...[11] として表します。日付の値は、常に ISO 書式 yyyy-mm-dd で UDF に渡されます。

  以下に例を示します。

  ```
  char     arg1[11];      /* example for DATE       */
  char    *result;        /* also perfectly acceptable */
  ```

- TIME

  正しい例。 C で CHAR(8) として、つまり char...[9] として表します。時間の値は、常に ISO 書式 hh.mm.ss で UDF に渡されます。

  以下に例を示します。

  ```
  char    *arg;           /* example for DATE          */
  char     result[9];     /* also perfectly acceptable */
  ```

- TIMESTAMP

正しい例。 C で CHAR(8) として、つまり char...[27] として表します。タイム・スタンプの値は、常に yyyy-mm-dd-hh.mm.ss.nnnnnn 書式で渡されます。

以下に例を示します。

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

- BLOB(n) と CLOB(n)

  正しい例。 C で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 UDF にパラメーターとして正しく渡されます。 RETURNS 文節の場合、UDF に戻される長さはバッファーの長さです。 UDF 本体は、構造変数 length を使っ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 が長さ 2 バイトとして定義されているプラットフォームでは、上記の例の sqldbchar の箇所に 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。 これら 2 つのデータ・タイプの詳細については、639ページの『C および C++ での wchar_t または sqldbchar データ・タイプの選択』を参照してください。

  [1] は、単にコンパイラーに対する配列を示しています。グラフィック文字を 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 終了グラフィック・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 UDF にパラメーターとして正しく渡されます。DB2 から UDF 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、640ページの『C および C++ での WCHARTYPE プリコンパイラー・オプション』で説明する 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 文節の場合、UDF に渡される長さはバッファーの長さです。 UDF 本体は、構造変数 length を使っ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、UDF 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

  AS LOCATOR タイプ修飾子は、UDF のパラメーター定義および結果定義においてのみ有効です。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。 このタイプ修飾子を指定すると、UDF には LOB 値全体ではなく 4 バイトのロケーターが渡されます。

  以下に例を示します。

```
sqludf_locator      *arg1;  /* locator argument */
sqludf_locator      *result; /* locator result */
```

タイプ udf_locator はヘッダー・ファイル sqludf.h で定義されています。このヘッダー・ファイルについては、435ページの『UDF インクルード・ファイル: sqludf.h』で説明されています。これらのロケーターの使用については、459ページの『UDF のパラメーターや結果としての LOB ロケーターの使用』で説明されています。

32 ビット・プラットフォームおよび 64 ビット・プラットフォームでのスクラッチパッドの作成

UDF を 32 ビット・プラットフォームと 64 ビット・プラットフォームの間で移送できるようにするには、 64 ビット値を含むスクラッチパッドを作成および使用する仕方を

変えなければなりません。 1 つまたは複数の 64 ビット値 (64 ビット・ポインターや sqlint64 BIGINT 変数など) を含むスクラッチパッド構造では、明示的な長さ変数を宣言してはなりません。たとえば、以下の例を実行すると、構造宣言の中に明示的な長さ変数が含まれているため、 64 ビット・プラットフォームではデータ位置合わせ例外になる可能性があります。

```
struct scratch1
{
  sqlint32 length;
  char chars[4];
  sqlint64 bigint_var;
};
```

上記の例において、32 ビット・プラットフォームと 64 ビット・プラットフォームの間での移送が可能になるようにスクラッチパッド構造を宣言するには、その構造に関連した明示的な長さ変数の宣言を取り除く必要があります。以下の例では、明示的な長さ変数を宣言せずにスクラッチパッド構造を宣言しています。

```
struct scratch1
{
  sqlint64 bigint_var;
  char chars[4];
};
```

UDF 内で明示的な長さ変数を宣言していないスクラッチパッド構造にアクセスするには、以下のような形式でスクラッチパッドを参照します。

```
struct scratchpad_data * data =
  (struct scratchpad_data*)scratch_pointer->data;
```

ここで、 *scratch_pointer* は UDF の *sqludf_scratchpad* ポインターを、 *data* はスクラッチパッドの内容を表しています。

UDF インクルード・ファイル: sqludf.h

このインクルード・ファイルには、 UDF を定義する際に役立つ構造、定義、および値が含まれます。このインクルード・ファイルは任意に使用できますが、470ページの『UDF コードの例』で示すサンプル UDF の例ではこのファイルを使用します。 UDF 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 インクルード・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

- DB 情報の構造

2. すべての SQL データ・タイプに対する C 言語型定義。 SQL 引き数に対応する UDF 引き数と、SQL データ・タイプを持つ結果を定義するために使用されます。データ・タイプは、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・タイプ名であり、FBD は For Bit Data を表します。

   また、AS LOCATOR 付加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。

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、411ページの『DB2 から UDF に渡される引き数』で定義した *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 引き数がヌルであるかどうかをテストするマクロ。

6. UDF に渡された LOB ロケーターによって LOB 値を操作するのに用いる、API の関数プロトタイプ。

次の項では、UDF の例を示して `sqludf.h` の包含と使用について説明します。

Java ユーザー定義の関数の作成および使用

Java の UDF は、他の言語の場合と同じように作成して使用できますが、ほんのわずかな違いもあります。 UDF のコーディングが終了した後、CREATE FUNCTION ステートメントを使用して、その UDF をデータベースに登録します。このステートメントを使用して Java UDF を登録することについては、 *SQL 解説書* を参照してください。その後ご使用のアプリケーションの SQL でそれを参照することができます。 UDF は FENCED または NOT FENCED を使用することができます。オプションを使用して、UDF の実行の仕方を変えることも可能です。 438ページの『Java UDF の実行の仕方の変更』を参照してください。

UDFsrv.java のサンプルでは、いくつかの Java UDF 方式の本体のサンプルが示されています。UDFcli.java と UDFclie.sqlj には、関連した CREATE FUNCTION ステートメントとそれらの UDF を呼び出すサンプルがあります。 そのサンプルと、それをコンパイルして実行するための README の指示については、 sqllib/samples/java ディレクトリーをご覧ください。

Java UDF のコーディング

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 を宣言する場合、予期される Java シグニチャーを指定して、Java メソッドとして呼び出されます。

```
public void name ( T1 a, T2 b, T3 c, T4 d)  { .....}
```

ここで、各パラメーターは以下のとおりです。

- *name* は、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任意の変数名
- *d* は、UDF の計算結果を示す任意の変数名

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 を UDF で呼び出す場合、DB2 は、次のシグニチャーを持つ UDF の Java インプリメンテーションを予期します。

```
import COM.ibm.db2.app.*;
public class sample extends UDF {
   public void test3(String arg1, Blob arg2, String arg3,
                     int result) { ... }
}
```

表関数を実行する Java UDF は、さらに多くの引き数を必要とします。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。この項の最初の例で、int result 変数は 4 の索引を持っています。 2 番目の例では、result1 〜 result3 は、2 〜 4 を指しています。 UDF が戻す前には設定されていない出力引き数の値は、NULL になります。

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 UDF では Java 標準 I/O ストリーム (System.in、System.out、および System.err) を使用できません。 Java UDF の例は、 sqllib/samples/java ディレクトリーのファイル DB2Udf.java を参照してください。

UDF を実行するのに使用するすべての Java クラス・ファイルは、 sqllib/function ディレクトリーか、対応するサブディレクトリーに常駐する必要があることに注意してください。 684ページの『Java クラスを置く場所』を参照してください。

Java UDF の実行の仕方の変更

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。詳細については、この次のサブセクションにある NO FINAL CALL 実行モデルを参照してください。

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 への呼び出しの合間に情報を保管するため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、 439ページの『Java の表関数実行モデル』の実行モデルによって示されているように、 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。

`COM.ibm.db2.app.UDF` インターフェースの詳細は、797ページの『COM.ibm.db2.app.UDF』を参照してください。このインターフェースには、UDF 内部に組み込める便利な呼び出し (`setSQLstate` や `getDBinfo` など) が記述されています。

Java の表関数実行モデル

Java で作成された表関数の場合、表関数において重要な特定のステートメントを DB2 が処理する各時点で生じることを理解しておくことは大切です。この情報の詳細は、以下の表で説明されています。それぞれの囲みの下部では、Web から情報を引き出す一般的な表関数に対して、コードを作成して実行できることを示しています。 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。
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、次の行のデータまたは 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。

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 をインプリメントする 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。
3. これらのモデルは、C/C++ および OLE の他の UDF 言語と完全に互換性があります。

OLE オートメーション UDF の作成

OLE (オブジェクトのリンクと埋め込み) オートメーションは、 Microsoft Corporation の 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呈示できます。 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呈示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 (Component Object Model; コンポーネント・オブジェクト・モデル) は、OLE の土台をなすものです。 OLE オートメーション UDF 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 UDF として呼び出すことができます。

この項では、読者が OLE オートメーションに関する用語や概念に精通していることを前提としています。本書では、OLE の初歩的な説明は行いません。 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。

OLE オートメーション UDF を例示する、 DB2 アプリケーション開発クライアントに組み込まれたサンプル・アプリケーションのリストについては、 785ページの表50 を参照してください。

OLE オートメーション UDF の作成と登録

OLE オートメーション UDF 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 登録データベース (レジストリー) に登録し、オートメーション・オブジェクトを識別するようにします。 progID は、プロセス内 (.DLL) またはローカル (.EXE) OLE オートメーション・サーバー、または DCOM (分散 COM) を介してリモート・サーバーを識別できます。 OLE オートメーション UDF は、スカラー関数か表関数のいずれかです。

OLE オートメーション・オブジェクトのコード化が終わったら、 SQL CREATE FUNCTION 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 UDF として登録する必要があります。 OLE オートメーション UDF の登録は、 C または C++ の外部 UDF の登録と非常に類似していますが、以下のオプションを使用する必要があります。

- LANGUAGE OLE
- FENCED (OLE オートメーション UDF は FENCED モードで実行する必要があるため)

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
  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++ で作成された関数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)
```

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

OLE オートメーション UDF (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 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 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。 SQL CREATE FUNCTION 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。 SQL ステートメント内で UDF が参照されるたびに、個々の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。

SQL データ・タイプの OLE オートメーション UDF への受け渡し方法

DB2 は、SQL タイプと OLE オートメーション・タイプの間でタイプ変換を処理します。次の表では、サポートされるデータ・タイプと、それらがどのようにマップされるかを要約しています。 OLE オートメーション・タイプから BASIC や C/C++ などのインプリメント用プログラミング言語へのマッピングについては、 445ページの表17 で説明しています。

表 16. SQL と OLE オートメーション・データ・タイプのマッピング

SQL タイプ	OLE オートメーション・タイプ	OLE オートメーション・タイプの説明
SMALLINT	short	16 ビットの符号付き整数
INTEGER	long	32 ビットの符号付き整数
REAL	float	32 ビットの IEEE 浮動小数点数
FLOAT または DOUBLE	double	64 ビットの IEEE 浮動小数点数
DATE	DATE	1899 年 12 月 30 日からの日数を表す、64 ビットの浮動小数点分数
TIME	DATE	
TIMESTAMP	DATE	
CHAR(*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*)[1]	SAFEARRAY[unsigned char]	8 バイト無符号データ項目の 1 次元 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*)[1]	SAFEARRAY[unsigned char]	
LONG VARCHAR[1]	SAFEARRAY[unsigned char]	
BLOB(*n*)	SAFEARRAY[unsigned char]	

注:

1. FOR BIT DATA を指定

DB2 と OLE オートメーション UDF の間で受け渡しされるデータは、参照呼び出しとして受け渡しされます。 表に載っていない、BIGINT、DECIMAL、 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グラフィック・データは、データベース・コード・ページから UCS-2 (Unicode としても知られている、 IBM コード・ページ 13488) スキーマに変換されます。戻される際に、データはデータベース・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。データベース・コード・ページから UCS-2 に、および

UCS-2 からデータベース・コード・ページに変換するコード・ページ変換テーブルがインストールされていない場合、 SQLCODE -332 (SQLSTATE 57017) を受け取ります。

BASIC と C++ での OLE オートメーション UDF のインプリメンテーション

OLE オートメーション UDF 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 や C++ を例として取り上げ、 OLE オートメーション UDF をインプリメントする方法を示します。

表17 では、様々な SQL データ・タイプから中間の OLE オートメーション・データ・タイプへのマッピング、および目的の言語 (BASIC または C++) のデータ・タイプへのマッピングを示しています。OLE データ・タイプは言語に依存しません (つまり、 444 ページの表16 の内容は、すべての言語にあてはまります)。

表 17. SQL および OLE データ・タイプから BASIC および C++ データ・タイプへのマッピング

SQL タイプ	OLE オートメーション・タイプ	UDF 言語	
		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*)、VARCHAR(*n*)、 LONG VARCHAR、CLOB(*n*)	BSTR	String	BSTR
GRAPHIC(*n*)、VARGRAPHIC(*n*)、 LONG GRAPHIC、DBCLOB(*n*)	BSTR	String	BSTR
CHAR(*n*)[1]、 VARCHAR(*n*)[1]、 LONG VARCHAR[1]、 BLOB(*n*)	SAFEARRAY[unsigned char]	Byte()	SAFEARRAY

注:

1. FOR BIT DATA を指定

BASIC での OLE オートメーション UDF

BASIC で OLE オートメーション UDF 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る SQL データ・タイプに対応している BASIC データ・タイプを使用する必要があります。

442ページの『OLE オートメーション UDF の作成と登録』の `bcounter` OLE オートメーション UDF を BASIC で宣言すると、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)
```

OLE 表オートメーションの例は、495ページの『例: BASIC でのメール OLE オートメーション表関数』にあります。

C++ での OLE オートメーション UDF

445ページの表17 では、SQL データ・タイプに対応する C++ データ・タイプと、それらがどのように OLE オートメーション・タイプにマップされるかを示しています。

increment OLE オートメーション UDF を C++ で宣言すると、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。呈示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 UDF の場合、 UDF 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、 UDF 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。 UDF 後書き引き数の場合、 sqlstate は [out] パラメーター、関数と関数の特定の名前は [in] パラメーター、スクラッチパッドは [in,out] パラメーター、呼び出しタイプは [in] パラメーターです。

スカラー関数には 1 つの出力パラメーターと出力標識が含まれますが、表関数には、CREATE FUNCTION ステートメントの RETURN 列の数に対応する複数の出力パラメーターと出力標識が含まれます。

OLE オートメーションは、ストリングを処理する BSTR データ・タイプを定義します。 BSTR は、OLECHAR: typedef OLECHAR *BSTR へのポインターとして定義されます。 BSTR の割り振りおよび解放の際には、 OLE では呼び出される側が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、という規則が適用されます。この規則は、DB2 と OLE オートメーション UDF に対しては、次に示す意味があります。呼び出される側が SAFEARRAY** として受け取る 1 次元のバイト配列にも、同じ規則が適用されます。

- [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、呼び出される側での解放および再割り振りが可能です。 [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++ UDF 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 分離プロセス) 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。

OLE DB 表関数

Microsoft OLE DB は、OLE/COM インターフェースのセットで、さまざまな情報源に保管されたデータに対する統一されたアクセスを、アプリケーションに提供します。 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 ユニバーサル・データベースは、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、アプリケーション開発の労力を削減することができます。 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。そのため必要とされるのは、言語タイプ OLEDB の

表関数を登録し、 OLE DB Provider と、データ・ソースと関係のある行セットを参照することだけです。 OLE DB 表関数を利用するために、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 ユニバーサル・データベースで使用するには、 OLE DB 2.0 またはそれ以降をインストールする必要があります (これは、 Microsoft 社の http://www.microsoft.com から入手できます)。 OLE DB をインストールしていないまま OLE DB 表関数を呼び出そうとすると、 DB2 は SQLCODE 465、SQLSTATE 58032、理由コード 35 を発行します。システム要件と、ご使用のデータ・ソースで使用可能な OLE DB Providers については、データ・ソース資料を参照してください。
OLE DB 表関数を定義および使用するサンプルのリストについては、 765ページの『付録B. サンプル・プログラム』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。完全修飾名にすることもできる行セット名の詳細については、451ページの『完全修飾行セット名』を参照してください。 OLE DB データ・タイプの DB2 データ・タイプへのマッピングについては、 453ページの『サポートされる OLE DB データ・タイプ』を参照してください。 CREATE FUNCTION ステートメントの完全な構文および EXTERNAL NAME 文節の規則については、 *SQL 解説書* を参照してください。

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'
```

ここで、各パラメーターは以下のとおりです。

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 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 452ページの『OLE DB Provider のサーバー名の定義』での説明のように定義してあるとすると、次の CREATE FUNCTION ステートメントを使用することができます。

```
CREATE FUNCTION orders ()
  RETURNS TABLE (orderid INTEGER, ...)
  LANGUAGE OLEDB
  EXTERNAL NAME 'Nwind!orders';
```

OLE DB 表関数により、任意の文字ストリング・データ・タイプの入力パラメーターを 1 つ指定することもできます。入力パラメーターを使用して、 OLE DB Provider にコマンド・テキストを直接渡します。入力パラメーターを定義する場合、EXTERNAL NAME 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・タイプは、 CREATE FUNCTION ステートメントの RETURNS TABLE 定義と互換性がある必要があります。行セットの列名とのバインドは、一致する列名に基づいているため、列を正しく命名していることを確かめる必要があります。

以下の例では、 Microsoft SQL Server 7.0™ データベースから保管情報を検索する OLE DB 表関数を登録します。 EXTERNAL NAME 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;
```

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。

OLE DB Provider のサーバー名の定義

OLE DB Provider のサーバー名を定義するためには、 CREATE WRAPPER OLEDB ステートメントを使用して、データベースごとに OLE DB ラッパーを 1 つずつ登録しておく必要があります。 CREATE WRAPPER OLEDB の詳細については、 *インストールおよび構成 補足* を参照してください。

多くの CREATE FUNCTION ステートメントに使用できる OLE DB データ・ソースのサーバー名を指定するには、 CREATE SERVER ステートメントを以下のように使用します。

- DB2 内で OLE DB Provider を識別する名前を指定する。
- WRAPPER OLEDB を指定する。
- CONNECTSTRING オプションに接続情報を指定する。

たとえば、以下の CREATE SERVER ステートメントを使用して、 Microsoft Access OLE DB Provider のサーバー名 Nwind を定義することができます。

```
CREATE SERVER Nwind
  WRAPPER OLEDB
  OPTIONS (CONNECTSTRING 'Provider=Microsoft.Jet.OLEDB.3.51;
           Data Source=c:¥msdasdk¥bin¥oledb¥nwind.mdb');
```

サーバー名 Nwind を使用して、 CREATE FUNCTION ステートメントの OLE DB Provider を識別することができます。たとえば以下のとおりです。

```
CREATE FUNCTION orders ()
  RETURNS TABLE (orderid INTEGER, ...)
  LANGUAGE OLEDB
  EXTERNAL NAME 'Nwind!orders';
```

CREATE SERVER ステートメントの完全な構文については、*SQL 解説書* を参照してください。 OLE DB Provider のユーザー・マッピングの情報については、 『ユーザー・マッピングの定義』を参照してください。

ユーザー・マッピングの定義

DB2 ユーザーのユーザー・マッピング を指定して、代替ユーザー名およびパスワードを使用して OLE DB データ・ソースにアクセスできるようにします。特定のユーザーにユーザー名をマップするには、 CREATE USER MAPPING ステートメントを使用してユーザー・マッピングを定義することができます。すべてのユーザーによって共用されるユーザー・マッピングを提供するには、ユーザー名とパスワードを CREATE FUNCTION ステートメントまたは CREATE SERVER ステートメントの接続ストリングに追加します。たとえば、OLE DB サーバー Nwind 上で DB2 ユーザー JOHN に固有のユーザー・マッピングを作成するには、以下の CREATE USER MAPPING ステートメントを使用します。

```
CREATE USER MAPPING FOR john
  SERVER Nwind
  OPTIONS (REMOTE_AUTHID 'dave', REMOTE_PASSWORD 'mypwd');
```

OLE DB 表関数 orders を呼び出すすべての DB2 ユーザーに対して同等のアクセスを提供するには、 CREATE FUNCTION または CREATE SERVER ステートメントのどちらかで以下の CONNECTSTRING を使用します。

```
CREATE FUNCTION orders ()
  RETURNS TABLE (orderid INTEGER, ...)
  LANGUAGE OLEDB
  EXTERNAL NAME '!orders!Provider=Microsoft.Jet.OLEDB.3.51;User ID=dave;
                 Password=mypwd;Data Source=c:¥msdasdk¥bin¥oledb¥nwind.mdb';
```

DB2 は、以下のユーザー・マッピング規則を適用します。

- ユーザー・マッピングが定義されている場合、 DB2 はマップされた許可情報を使用して OLE DB Provider に接続しますが、この場合 CONNECTSTRING の既存のユーザー ID とパスワードを上書きする可能性があります。
- ユーザー・マッピングが定義されていない場合、情報が与えられていなければ、 DB2 は CONNECTSTRING からの許可情報を使用します。
- ユーザー・マッピングが定義されておらず、CONNECTSTRING に許可情報が与えられていない場合、 DB2 は (Provider がサポートしているなら) 現行の DB2 許可情報を使用します。

CREATE USER MAPPING ステートメントの完全な構文については、 *SQL 解説書* を参照してください。

サポートされる OLE DB データ・タイプ

以下の表では、DB2 データ・タイプを OLE DB データ・タイプにマップする方法が示されています。これについては *Microsoft OLE DB 2.0 Programmer's Reference and*

Data Access SDK, Microsoft Press, 1998 に記載されています。マッピング表を使用して、OLE DB 表関数に適切な RETURNS TABLE 列を定義します。たとえば、データ・タイプ INTEGER の列を使用して OLE DB 表関数を定義する場合、 DB2 は OLE DB 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・タイプから OLE DB データ・タイプへのマッピングについては、 OLE DB Provider の資料を参照してください。 ANSI SQL、Microsoft Access、および Microsoft SQL Server プロバイダーが、それぞれのデータ・タイプを OLE DB データ・タイプにマップする方法の例については、 *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998 を参照してください。

表 18. *DB2 データ・タイプの OLE DB へのマッピング*

DB2 データ・タイプ	OLE DB データ・タイプ
SMALLINT	DBTYPE_I2
INTEGER	DBTYPE_I4
BIGINT	DBTYPE_I8
REAL	DBTYPE_R4
FLOAT/DOUBLE	DBTYPE_R8
DEC (p, s)	DBTYPE_NUMERIC (p, s)
DATE	DBTYPE_DBDATE
TIME	DBTYPE_DBTIME
TIMESTAMP	DBTYPE_DBTIMESTAMP
CHAR(N)	DBTYPE_STR
VARCHAR(N)	DBTYPE_STR
LONG VARCHAR	DBTYPE_STR
CLOB(N)	DBTYPE_STR
CHAR(N) FOR BIT DATA	DBTYPE_BYTES
VARCHAR(N) FOR BIT DATA	DBTYPE_BYTES
LONG VARCHAR FOR BIT DATA	DBTYPE_BYTES
BLOB(N)	DBTYPE_BYTES
GRAPHIC(N)	DBTYPE_WSTR
VARGRAPHIC(N)	DBTYPE_WSTR
LONG GRAPHIC	DBTYPE_WSTR
DBCLOB(N)	DBTYPE_WSTR

注: OLE DB データ・タイプ変換規則は、 *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、以下のとおりです。

- OLE DB データ・タイプ DBTYPE_CY を検索するには、データを OLE DB データ・タイプ DBTYPE_NUMERIC(19,4) に変換することができ、これは DB2 データ・タイプ DEC(19,4) にマップされます。
- OLE DB データ・タイプ DBTYPE_I1 を検索するには、データを OLE DB データ・タイプ DBTYPE_I2 に変換することができ、これは DB2 データ・タイプ SMALLINT にマップされます。
- OLE DB データ・タイプ DBTYPE_GUID を検索するには、データを OLE DB データ・タイプ DBTYPE_BYTES に変換することができ、これは DB2 データ・タイプ CHAR(12) FOR BIT DATA にマップされます。

スクラッチパッドに関する考慮事項

UDF がスクラッチパッドを使用するかどうかに影響を及ぼす要素は重要であるため、この項を特別に設けて説明します。コーディングに関するその他の考慮事項については、464ページの『コーディングに関するその他の考慮事項』で説明します。

UDF を再入可能にコーディングすることは重要です。これは主に、UDF に対する参照の多くが関数本体の同一コピーを使用するためです。実際は、このような多くの*参照* は異なるステートメントやアプリケーションで行われることさえあります。ただし、関数はある呼び出しから次の呼び出しへの状態を保管する必要があることに注意してください。この関数は次の 2 種類に分類されます。

1. 正しいかどうかが保管状態に依存する関数。

   このような関数の例として、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 ずつ増分するという、単純な *counter* 関数を示します。この関数を使用すると、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 ...
   ```

   このタイプの関数は NOT DETERMINISTIC (または VARIANT) です。その出力は、単にその SQL 引き数の値に左右されません。この *counter* 関数については、478ページの『例: カウンター』で説明します。

2. いくつかの初期化処理を 1 回だけで実行できることによりパフォーマンスを改善できる関数。

   このような関数の例として、文書アプリケーションの一部の *match* 関数があり、この関数は指定した文書が与えられたストリングを含む場合には 'Y' を、それ以外の場合は '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* が行おうとする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` を含むすべての文書 ID のリストを、 DB2 の外側で保持される文書アプリケーションから検索します。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 がこのリストに含まれているかどうかを確認します。

特にこの *match* 関数は DETERMINISTIC (または NOT VARIANT) です。その結果は、入力される引き数値のみに左右されます。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。

上述の 2 つの要求を両方とも満たす方法は、 CREATE FUNCTION ステートメントに SCRATCHPAD を指定することです。

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

この SCRATCHPAD キーワードは、スクラッチパッドを関数に割り振って保持するよう DB2 に命じます。 DB2 は、スクラッチパッドを 2 進数のゼロに初期化します。表関数に NO FINAL CALL (デフォルト) が指定されている場合、 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 を指定する場合、 DB2 はその後、スクラッチパッドの内容を検査したり変更したりしません。 スクラッチパッドは各呼び出し時に関数に渡されます。関数は再入可能の場合もありますが、 DB2 は関数の状態に関する情報をスクラッチパッドに保存します。

したがって *counter* の場合、戻される最後の値はスクラッチパッドに保持されます。また *match* の例は、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持し、それ以外の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持します。

UDF は、システム・リソースを獲得する必要があると認識されるため、 FINAL CALL キーワードを使って定義できます。このキーワードは、ステートメントの終わりの処理で UDF を呼び出すよう DB2 に命じるため、 UDF はそのシステム・リソースを解放することができます。特にスクラッチパッドのサイズは固定であるため、UDF はそれ自体にメモリーを割り振り、最終呼び出しを使ってメモリーを解放する必要があります。たとえば上記の *match* 関数は、与えられたテキスト・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  FINAL CALL;
```

スクラッチパッドを使用し、副照会で参照される UDF の場合、副照会の合間に、DB2 は最終呼び出しをして (UDF がそのように指定されている場合)、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 を常に副照会で使用している場合、 FINAL CALL や呼び出しタイプ引き数を使って UDF を定義したり、 *2 進数のゼロ* 条件を必ず検査することによって、この可能性を回避することができます。

FINAL CALL を指定する場合は、UDF 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。

表関数に関する考慮事項

外部表関数は、参照元の SQL に表を送達する UDF です。表関数参照は、SELECT 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。
- DB2 とユーザー定義のスカラー関数の間で使用される標準インターフェースは、表関数を扱えるように拡張されています。 *SQL 結果* 引き数は表関数に対して繰り返され、各インスタンスは、CREATE FUNCTION ステートメントの RETURNS TABLE 文節の定義で戻される列に対応します。 *SQL 結果標識* も同様に繰り返されます。各インスタンスは、対応する *SQL 結果* インスタンスと関係しています。
- 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 *DB 情報* 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 n* 指定があります。 これを指定することにより、DB2 最適化プログラム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。

表関数のエラー処理

表関数呼び出しのエラー処理モデルは、以下のとおりです。

1. FIRST 呼び出しが失敗すると、それ以降の呼び出しは行われません。
2. FIRST 呼び出しが成功すると、ネストした OPEN、FETCH、および CLOSE 呼び出しが行われ、必ず FINAL 呼び出しが行われます。

3. OPEN 呼び出しが失敗すると、FETCH 呼び出しも CLOSE 呼び出しも行われません。
4. OPEN 呼び出しが成功すると、FETCH 呼び出しと CLOSE 呼び出しが行われます。
5. FETCH 呼び出しが失敗すると、それ以降 FETCH 呼び出しが行われることはありませんが、CLOSE 呼び出しが行われます。

注: このモデルは、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` 分離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。

スカラー関数のエラー処理

FINAL CALL 指定を使って定義されたスカラー UDF のエラー処理モデルは以下のとおりです。

- FIRST 呼び出しが失敗すると、それ以降の呼び出しは行われません。
- FIRST 呼び出しが成功すると、ステートメントの処理によって保証されているように、さらに NORMAL 呼び出しが行われ、FINAL 呼び出しは必ず行われます。
- NORMAL 呼び出しが失敗すると、それ以降 NORMAL 呼び出しが行われることはありませんが、 (FINAL CALL が指定されている場合は) FINAL 呼び出しが行われます。

すなわち、FIRST 呼び出し時にエラーが戻される場合、FINAL 呼び出しは行われないため、エラーを戻す前に UDF を終結処理する必要があります。

表関数のエラー処理モデルは、この章の 458ページの『表関数に関する考慮事項』の項で定義されています。

注: このモデルは、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` 分離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。

UDF のパラメーターや結果としての LOB ロケーターの使用

`AS LOCATOR` は、任意の LOB データ・タイプや、 CREATE FUNCTION ステートメントの LOB タイプに基づいた任意の特殊タイプに追加できます。これは、渡されるパラメーターと戻される結果の両方に適用されます。この場合、DB2 は次のことを行います。

- パラメーターの場合、DB2 は、LOB 値全体の代わりに 4 バイトのロケーターを渡します。このロケーターには、実バイトを取得および操作する際に、特殊な API (後述) に関連して、様々な方法で使用されます。 UDF が数バイトの値しか必要としない場合、保存値は消去されます。

  ストレージ LOB 全体のメモリーを割り振る必要はありません。

  パフォーマンス 値全体を具体化する際に、大量の入出力時間やバイト移動指示が必要になることがあります。

- 結果の場合、LOB 値全体ではなく 4 バイトのロケーターを戻します。この場合もまた、ストレージおよびパフォーマンスの利点があります。

ロケーター値は修正しないでください。修正すると値は使用できなくなり、API がエラーを戻します。

これらの特殊 API は、NOT FENCED として定義された UDF でのみ使用できます。これは、バグのある UDF によってシステムが損害を受ける可能性があるので、テスト段階にある UDF を実動データベース上で使用してはならない、ということを暗に示しています。テスト・データベースでの作業であれば、 UDF にバグがあったとしても、永続する損害は生じません。 UDF にエラーがないことが分かったら、実動データベースに適用できます。

次の API は、sqludf.h UDF インクルード・ファイルに含まれている関数プロトタイプを使って定義されています。

```
extern int sqludf_length(
    sqludf_locator*     udfloc_p,       /* in:  User-supplied LOB locator value */
    sqlint32*           Return_len_p    /* out: Return the length of the LOB value */
);
extern int sqludf_substr(
    sqludf_locator*     udfloc_p,       /* in:  User-supplied LOB locator value */
    sqlint32            start,          /* in:  Substring start value (starts at 1) */
    sqlint32            length,         /* in:  Get this many bytes */
    unsigned char*      buffer_p,       /* in:  Read into this buffer */
    sqlint32*           Return_len_p    /* out: Return the length of the LOB value */
);
extern int sqludf_append(
    sqludf_locator*     udfloc_p,       /* in:  User-supplied LOB locator value */
    unsigned char*      buffer_p,       /* in:  User's data buffer */
    sqlint32            length,         /* in:  Length of data to be appended */
    sqlint32*           Return_len_p    /* out: Return the length of the LOB value */
);
extern int sqludf_create_locator(
    int                 loc_type,       /* in:  BLOB, CLOB or DBCLOB? */
    sqludf_locator**    Loc_p           /* out: Return a ptr to a new locator */
);
extern int sqludf_free_locator(
    sqludf_locator*     loc_p           /* in: User-supplied LOB locator value */
);
```

次に、これらの API がどのように動作するかを説明します。データ・タイプに関係なく、すべての長さはバイト数で、 1 バイト文字や 2 バイト文字単位ではないことに注意してください。

戻りコード。各 API ごとに DB2 によって UDF に戻される戻りコードは以下のとおりです。

0 成功しました。

-1 API に渡されたロケーターは、呼び出しを行う前に `sqludf_free_locator()` によって解放されました。

-2 FENCED モードの UDF で呼び出しが試行されました。

-3 API に無効な入力値が指定されました。各 API に固有の、無効な入力値の例については、以下のその説明を参照してください。

その他 無効なロケーターか、他のエラー (たとえばメモリー・エラー) です。これらの場合に戻される値は、エラー条件に対応する SQLCODE です。たとえば、-423 は無効なロケーターを意味します。これら「その他」のコードとともに UDF に戻る前に、 DB2 がエラーの重大度について判断することに注意してください。重大エラーの場合、DB2 はエラーが発生したことを記録し、UDF が DB2 に戻された場合には、UDF がエラー SQLSTATE を DB2 に戻すかどうかにかかわらず、 DB2 はそのエラー条件に対して適切な処置を取ります。重大エラーではない場合、DB2 はエラーが発生したことを記録せず、正しい処置を取れるかどうかの判断を UDF に任せるか、またはエラー SQLSTATE を DB2 に戻します。

- `sqludf_length()`

  LOB ロケーターが与えられると、そのロケーターが表す LOB 値の長さを戻します。該当するロケーターは通常、DB2 によって UDF に渡されるロケーターですが、UDF が (`sqludf_append()` を使って) 作成中の結果値を表すロケーターである場合もあります。

  通常、UDF はロケーターを受け取ったときに LOB 値の長さを調べる場合に、この API を使用します。

  戻りコード **3** は以下のことを示します。

  – udfloc_p (ロケーターのアドレス) がゼロである

  – return_len_p (長さを書き込む場所のアドレス) がゼロである

- `sqludf_substr()`

  LOB ロケーター、LOB 内の開始位置、要求する長さ、およびバッファーへのポインターが与えられると、この API は、バイトをバッファーに入れ、移動できたバイトの数を戻します。 (当然ながら、UDF は、要求する長さに十分足りるバッファーを提供しなければなりません。) 移動されるバイトの数は、要求した長さよりも短いことがあります。たとえば、位置 101 から始まる 50 バイトを要求したときに LOB 値の長さが 120 だけである場合、この API は 20 バイトのみを移動します。

  通常、UDF は、ロケーターを受け取ったときに LOB 値のバイトを調べる場合、この API を使用します。

  戻りコード **3** は以下のことを示します。

  – udfloc_p (ロケーターのアドレス) がゼロである

  – 開始部分が 1 より小さい

– 長さは負の数である

– buffer_p (バッファー・アドレス) がゼロである

– return_len_p (長さを書き込む場所のアドレス) がゼロである

• `sqludf_append()`

LOB ロケーター、データが入っているデータ・バッファーへのポインター、および追加されるデータの長さが与えられると、この API は LOB 値の終わりにデータを追加して、追加されたバイトの長さを戻します。 (追加される長さが、追加するために与えられた長さと常に等しいことに注意してください。全体の長さを追加できない場合、 `sqludf_append()` の呼び出しは、戻りコード `other` を出して失敗します。)

通常、UDF は、結果が AS LOCATOR で定義され、 `sqludf_create_locator()` を使ってロケーターを作成した後で結果値を 1 つずつ追加して作成するときに、この API を使用します。この場合、作成処理を完了した後、UDF は結果引き数が指す場所にロケーターを移動します。

この API を使って入力ロケーターに追加することもできます。これは、UDF 内で値を操作する際に最大の柔軟性が得られるという点では便利ですが、 SQL ステートメントの LOB 値やデータベースに保管されている LOB 値には影響を及ぼしません。

この API を使用して、かなり大きい LOB 値であっても少しずつ作成することができます。結果が作成されるまでにかなりの追加をする場合、このタスクのパフォーマンスは以下を行うことによって改善することができます。

– 大きなアプリケーション制御ヒープを割り当てる (APP_CTL_HEAP_SZ は、データベース・マネージャー構成パラメーター)。

– さらに大きなバッファーを指定して、追加の回数を減らす。たとえば、50 バイトのバッファーを 20 回追加する代わりに、 1000 バイトの追加を 1 回だけ行う。

sqludf_append() API によって大きな LOB 値を大量に作成する SQL アプリケーションは、使用可能なディスク・スペース容量の制限によってエラーを検出することがあります。これらのエラーが発生する可能性は、以下を行うことによって削減することができます。

– それぞれの追加にさらに大きいバッファーを使用する。

– ステートメント間での COMMIT を頻繁に行う。

– SELECT ステートメントの各行でこの API による LOB 値が作成される場合、CURSOR WITH HOLD を使用して、行間で COMMIT を行う。

戻りコード **3** は以下のことを示します。

– udfloc_p (ロケーターのアドレス) がゼロである

– 長さは負の数である

– buffer_p (バッファー・アドレス) がゼロである

• `sqludf_create_locator()`

データ・タイプ (たとえば SQL_TYP_CLOB) が与えられると、ロケーターを作成します。 (データ・タイプ値は、外部アプリケーション・ヘッダー・ファイル sql.h で定義されています。)

通常、UDF は、UDF 結果が AS LOCATOR を指定して定義されており、sqludf_append() を使って結果値を作成する場合に、この API を使用します。他には、LOB 値を内部的に操作する場合にこの API を使用します。

戻りコード **3** は以下のことを示します。

– udfloc_p (ロケーターのアドレス) がゼロである

– loc_type が、有効な 3 つの値の 1 つではない

– loc_p (ロケーターを書き込む場所のアドレス) がゼロである

• sqludf_free_locator()

渡されたロケーターを解放します。

この API を使用して、 sqludf_create_locator() API を使用して作成されて内部操作でしか使用されなかったロケーターを解放します。 UDF に渡されたロケーターは必ずしも解放する必要はありません。 sqludf_create_locator() を介して UDF によって作成されたロケーターが、出力として UDF の外に渡される場合、それは必ずしも解放する必要はありません。

戻りコード **3** は以下のことを示します。

– udfloc_p (ロケーターのアドレス) がゼロである

これらの API の使用には、以下の注意事項が適用されます。

注:

1. LOB ロケーターを戻すよう定義されている UDF には、使用可能ないくつかの値があります。以下のものが戻されます。
   - 渡された入力ロケーター
   - sqludf_append() によって追加され、それに渡された入力ロケーター
   - sqludf_create_locator() によって作成され、 sqludf_append() によって追加されたロケーター
2. 表関数は、1 つ以上の LOB ロケーターを戻すものとして定義することができます。そのおのおのは、前の項目で説明された値のいずれかです。そのような表関数では、いくつかの表関数列の出力と同じロケーターを戻すことも有効です。
3. 入力引き数として表関数に渡された LOB ロケーターは、行生成プロセスの期間全体で有効です。事実、1 行を生成している間に、そのような LOB ロケーターを使用して LOB に表関数を追加することができます。次の行上に追加されたバイトが現れます。
4. UDF からの LOB ロケーター出力として DB2 で生成された LOB を表示するために内部制御機構を使用すると、1950 バイト使用します。この理由により、またソートへの入力である行サイズに制限があるため、 UDF の LOB ロケーターとして生成

された複数の LOB をソートしようとする照会は、関係する他の列のサイズに応じて、1 行につき (最大で) 2 つの値に制限されます。これと同じ制限は、表に挿入される行にも適用されます。

LOB ロケーター使用のシナリオ

ここでは、実際に起こりうる、LOB ロケーターの有用性を示すシナリオを簡単に要約します。次の 4 つのシナリオは、ロケーターの使用のあらましで、必要な空間を減らして効率を上げる方法を示します。

- 入力 LOB の各部分への可変アクセス。

  UDF は、sqludf_substr() を使って LOB 値の最初の部分を調べ、そこで見つかったサイズ変数に基づき、再び sqludf_substr() を使って 1 億バイトの LOB 値の中から自由に数バイトを読み取ることができます。

- 入力 LOB のほとんど各部分を 1 つずつ処理する。

  この UDF は、LOB 値内で何かを検索しています。大抵の場合は初めの方で見つかりますが、ときには 1 億バイトの値全体を走査しなければならないこともあります。 UDF は、sqludf_length() を使ってこの特定の値のサイズを検出し、sqludf_substr() への呼び出しをループに入れることによって、この値を 1 000 バイトずつ調べます。このとき、開始位置として変数を使用し、ループをまわるたびに変数を 1 000 ずつ増やします。検索しているものが見つかるまで、この方法を続行します。

- 2 つの入力 LOB のうちの 1 つを戻す

  この UDF は入力として 2 つの LOB ロケーターを取り、出力として 1 つの LOB ロケーターを戻します。まず、2 つの入力を検査して比較し、受け取ったバイトを sqludf_substr() を使って読み取り、何らかのアルゴリズムに基づいて 2 つのうちのどちらを選択するかを決定します。決定したら、選択した方の入力のロケーターを UDF 結果引き数で示されたバッファーにコピーして、終了します。

- 入力 LOB をカット・アンド・ペーストし、結果を戻す。

  UDF に、LOB 値と、たいていは前もって想定された処理方法を示す他の引き数が渡されます。 UDF は、出力としてロケーターを作成し、出力値を順番に作成します。このとき、結果値のほとんどは、他の入力引き数に含まれる指示に基づき、sqludf_substr() を使って読み取る入力 LOB の異なる部分から取ります。この処理が終わると、最後に結果ロケーターを UDF 結果引き数が指すバッファーにコピーし、終了します。

コーディングに関するその他の考慮事項

この項では、UDF のインプリメント、留意すべきこと、禁止事項などに関する考慮事項を追加して説明します。

ヒントとアドバイス

以下に、UDF を正常にインプリメントするために考慮すべき点を示します。

- **UDF 本体は保護する必要がある。**実行可能な関数本体が DB2 により何らかの方法で収集または保護されることはありません。 CREATE FUNCTION ステートメントは、本体のみを指します。関数および関数に依存するデータベース・アプリケーションの保全性を保持するには、関数を含むディレクトリーへのアクセスを管理したり、関数本体そのものを保護することにより、関数本体が誤ってまたは故意に削除されたり置換されることがないようにしてください。
- DB2 は、DB2 と SQL 間のインターフェースにおけるすべてのバッファー (すなわち、すべての SQL 引き数と関数の戻り値) にポインターを渡す。 UDF 引き数をポインターとして必ず定義してください。
- 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 UDF に提供されることを意味します。 UDF がその入力パラメーターを変更する場合、その変更は SQL 値または処理に影響を及ぼしませんが、 DB2 が誤動作を起こすことがあります。
- OLE オートメーションでは、入力パラメーターを変更しないでください。そうでないと、メモリー・リソースが解放されず、メモリー・リークが発生することがあります。

  OLE ライブラリー・バージョンの大きなミスマッチがある場合、または OLE ライブラリーの初期化の時に障害が発生した場合、データベース・マネージャーは、SQLCODE -465 (SQLSTATE 58032) と理由コード 34 を戻します (`Failure to initialize OLE library`)。
- UDF は、操作中のすべてのプラットフォーム上で**必ず**再入可能にする。そうすることで、UDF の 1 つのコピーを複数のステートメントおよびアプリケーションで並行して使用できます。

  SCRATCHPAD 機能を使用すると、再入可能に課せられた制限の多くを受けずに済みます。
- 現在使用中の関数の本体を修正すると (再コンパイルおよび再リンクなど)、 DB2 はトランザクション内で*関数の変更* を行わない。 ただし、こうした修正を動的に行うと、その後のトランザクションで使用されるコピーは変更されることがあります。ご使用のオペレーティング・システムによっては、使用中の UDF 本体を変更できないこともあります。 このような修正は行わない方がよいでしょう。
- 動的メモリーを UDF に割り振る際は、DB2 に戻る前にそのメモリーを解放する。これは特に NOT FENCED の場合に重要です。ただし、SCRATCHPAD 機能を使用すると、呼び出し間で UDF が必要とする動的メモリーを備えることができます。スクラッチパッドをこの方法で使用する場合は、ステートメントの終わりの処理で割り振られたメモリーを解放できるように、 UDF に対する CREATE FUNCTION で FINAL CALL 属性を指定してください。このようにするのは、UDF を繰り返し使用することにより、システムが時間とともにメモリーを使い果たしてしまうことがあるためです。

このことは、UDF が使用する他のシステム・リソースにも当てはまります。

- NOT NULL CALL オプションを使用することに意味がある場合は、これを使用する。この CREATE FUNCTION オプションの場合、それぞれの SQL 引き数がヌルであるかどうか、ヌル値を持つ場合に正しく実行されるかどうかを検査する必要がありません。
- UDF からの結果が入力 SQL 引き数以外の要素に左右される場合は、 NOT DETERMINISTIC オプションを使用してください。このオプションを指定すると、SQL コンパイラーが最適化を実行する際に矛盾した結果を生じないようにすることができます。
- UDF が内部でも外部でも確実に実行されなければならない結果を持つ場合は、EXTERNAL ACTION オプションを使用する。 EXTERNAL ACTION を使用すると、 SQL コンパイラーが最適化を実行する際に一定の状況での UDF の呼び出しを妨げないようにすることができます。
- FENCED と UNFENCED のどちらを選択すべきかについての考慮事項:

FENCED UDF

FENCED UDF は自己プロセスの中で実行されるため、意識的にであれ無意識にであれ、ほとんどの DB2 内部制御およびデータ域にアクセスできません。したがって、データベースには FENCED UDF を選択するのが無難と言えます。ただし、NOT FENCED UDF を選択した場合ほどではないとはいえ、プログラミング・エラーを含む FENCED UDF が DB2 をダウンさせる可能性は依然としてあります。たとえば、戻り変数を何回も上書きするような UDF は、 DB2 を異常終了させる可能性があります。

UNFENCED UDF

NOT FENCED UDF は DB2 のエンジン・プロセスに直接ロードされて実行されるため、 NOT FENCED UDF は FENCED UDF よりもパフォーマンスが優れています。 NOT FENCED UDF は、プロセス通信のオーバーヘッドによるパフォーマンスの低下を防ぎます。ただし、NOT FENCED UDF は DB2 内部制御やデータ域にアクセスしたり更新する場合があります。 NOT FENCED UDF を正しく作成しないと、 DB2 をダウンさせる可能性は FENCED UDF の場合よりも高くなります。

これらのことから明らかなとおり、 FENCED と NOT FENCED UDF を両方とも使用する場合は、以下の点に注意する必要があります。

– UDF を確実に正しく作成する必要があります。

– 十分な考慮の下に UDF を設計し、コードを注意深く検討する必要があります。

– 万が一 UDF が正しく作成されていなくても問題が起きない環境 (たとえば、テスト・データベース) で UDF をテストする必要があります。

UDF が原因で起きる異常終了のほとんどは DB2 によって取り込まれ、 -430 SQLCODE が戻されます。これにより、データベースの破壊が防止されます。とはいえ、特定のタイプの UDF の誤動作 (戻り値バッファーを何回も上書きするなど)

は、 UDF だけでなく DB2 をもダウンさせる場合があります。可変長データを戻す UDF や、戻り値バッファーに移動させる必要のあるバイト数を計算する UDF を作成する場合には、特に十分な注意を払ってください。

- UDF と EUC コード・セットの使用に関する考慮事項については、 539ページの『UDF に関する考慮事項』を参照してください。
- NOT FENCED を使用した UDF を実行するアプリケーションでは、そのような UDF を最初に呼び出すときに、 `UDF_MEM_SZ` 構成パラメーターに指示されたメモリー・サイズのブロックが作成されます。 その後は、ステートメント単位で、このメモリーのブロックから必要に応じて、 DB2 と NOT FENCED を使用した UDF との間でのやりとりで使用されるメモリーを割り振ったり割り振り解除したりします。

  FENCED UDF の場合、異なるメモリーのブロックが同じ仕方で使用されます。プロセス間でそのメモリーが共用されるという点が異なります。 NOT FENCED を使用した UDF と FENCED を使用した UDF の両方を 1 つのアプリケーションが使用する場合は、 `UDF_MEM_SZ` パラメーターにそれぞれのサイズが指定された、 2 つの別個のメモリー・ブロックが使用されます。この構成パラメーターの詳細については、*管理の手引き* を参照してください。
- 以下の状況では、DISALLOW PARALLELISM オプションを使用する。
  - スカラー UDF で、UDF が同一コピーの実行に完全に依存する場合。一般に、NOT DETERMINISTIC SCRATCHPAD UDF の場合にこのようになります。 (例については、456ページの『スクラッチパッドに関する考慮事項』で指定された `counter` UDF を参照してください。)
  - 単一の参照のために同時に複数の区画上で UDF を実行したくない場合。
  - 表関数を指定している場合。

  そうでない場合、ALLOW PARALLELISM (デフォルト) を指定する必要があります。

UDF に関する制限と警告

この項では、UDF で回避すべき項目について説明します。

1. 一般に 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。次の 2 点は例外です。
   a. シグナル・ハンドラーや例外ハンドラーを登録すると、 DB2 がそれと同じハンドラーを使用できなくなり、予期せぬエラーを引き起こす。
   b. 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 UDF のコーディングとテストには注意してください。

2. 'DB2' で始まるすべての環境変数の値は、データベース・マネージャーが `db2start` で開始される時点で収集され、 UDF (分離であるかどうかは関係ない) で使用できる。 ただし `DB2CKPTR` 環境変数だけは例外です。環境変数は*取り込まれる* 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、 UDF では使用できません。
3. 外部 UDF に渡される LOB については、 *UDF 共用メモリー・サイズ* の DB2 システム構成パラメーターにより指定される最大サイズに制限される。このパラメーターに指定できる最大数は 256M です。 DB2 におけるデフォルト設定は 1M です。このパラメーターの詳細については、 *管理の手引き* を参照してください。
4. 画面およびキーボードに対する入出力は行わない方がよい。 DB2 の処理モデルでは、UDF 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、ファイルに書き出すことはできます。

   注: DB2 は、UDF で行われる外部の入出力を DB2 独自のトランザクションと同期しようとはしません。したがって、たとえば UDF がトランザクション処理中にファイルに書き出され、そのトランザクションが後に何らかの理由で戻されると、 DB2 はそのファイルへの書き出しを発見したりやり直したりしようとはしません。
5. UNIX ベースのシステムでは、 UDF は DB2 エージェント・プロセスのユーザー ID (NOT FENCED) や、実行可能な `db2udf` を持つユーザー ID (FENCED) 下で実行される。 このユーザー ID は、UDF に利用できるシステム・リソースを制御します。ご使用のプラットフォーム用の `db2udf` 実行可能ファイルについては、 *概説およびインストール* を参照してください。
6. 保護されたリソース (1 回で 1 つの処理にだけアクセスできるリソース) を UDF の内部で使用する際は、UDF 間のデッドロックを避けなければならない。 2 つ以上の UDF のデッドロックが発生すると、DB2 はその条件を検出できません。
7. 文字データは、データベースのコード・ページ内の外部関数に渡される。同様に、関数から出力される文字ストリングは、データベースによりデータベースのコード・ページを使用すると想定されます。アプリケーションのコード・ページがデータベースのコード・ページとは異なる場合は、 SQL ステートメント内の別の値に対してと同様の方法でコード・ページの変換が行われます。このような変換は、FOR BIT DATA を文字パラメーターの属性として、または CREATE FUNCTION ステートメントの結果としてコーディングすることにより回避できます。文字パラメーターが FOR BIT DATA 属性を用いて定義されていない場合、 UDF コードはデータベースのコード・ページ内の引き数を受け取ります。

   CREATE FUNCTION で DBINFO オプションを使用すると、データベース・コード・ページが UDF に渡されることに注意してください。この情報を使用して、コード・ページを選ぶ UDF を多数の異なるコード・ページで機能するように定義することができます。
8. C++ を使用して UDF を作成する場合は、次のようにして関数名を宣言できます。

   ```
   extern "C" void SQL_API_FN udf( ...arguments... )
   ```

`extern "C"` は、 C++ コンパイラーによる関数名のタイプ修飾 (すなわち「マングル」) を防止します。この宣言の場合を除き、CREATE FUNCTION ステートメントを発行する場合には、関数名のタイプ修飾も組み込む必要があります。

UDF コードの例

次の UDF コード例が DB2 で提供されます。

例: 整数除算演算子

例: CLOB の折り返し、母音の検出

例: カウンター

提供されるすべての例の記載ページと、それらの呼び出し方法については、765ページの『付録B. サンプル・プログラム』を参照してください。

UDF のコンパイルとリンクに関する説明は、 *アプリケーション構築の手引き* を参照してください。

それぞれの UDF 例には、対応する CREATE FUNCTION ステートメントと、使用法を示す短いシナリオが付いています。このシナリオはすべて、次の TEST という表を使います。 TEST 表は、このシナリオで行われる一定の項目を説明するために慎重に加工されています。以下にこの表の定義を示します。

```
CREATE TABLE TEST (INT1 INTEGER,
                   INT2 INTEGER,
                   PART CHAR(5),
                   DESCR CLOB(33K))
```

この表の作成後、CLP を使って次のステートメントを出してその内容を表示します。

```
SELECT INT1, INT2, PART, SUBSTR(DESCR,1,50) FROM TEST
```

CLOB 列で SUBSTR 関数を使用すると、出力が読みやすくなることに注目してください。次のような CLP 出力が表示されます。

```
INT1        INT2        PART  4
----------- ----------- ----- --------------------------------------------------
         16           1 brain The only part of the body capable of forgetting.
          8           2 heart The seat of the emotions?
          4           4 elbow That bendy place in mid-arm.
          2           0 -     -
         97          16 xxxxx Unknown.
  5 record(s) selected.
```

例およびそれに続くシナリオを読む際に、表 TEST での上記のような情報を参照してください。

例: 整数除算演算子

DB2 で整数除算処理をすると、除数がゼロのときにエラー SQLCODE -802 (SQLSTATE 22003) が戻され、ステートメントが終了してしまうため都合が悪いとしましょう。 (DFT_SQLMATHWARN 構成パラメーターで *friendly 算術計算* を使用できるようにする場合、 DB2 はこの状況下でエラーを戻すのではなく NULL を戻します。) 整数除算が NULL を戻すようにするには、次の UDF をコーディングします。

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sqludf.h>
#include <sqlca.h>
#include <sqlda.h>

 /*************************************************************************
*  function divid: performs integer divid, but unlike the / operator
*                  shipped with the product, gives NULL when the
*                  denominator is zero.
*
*                This function does not use the constructs defined in the
*                "sqludf.h" header file.
*
*     inputs:  INTEGER num     numerator
*              INTEGER denom   denominator
*     output:  INTEGER out     answer
**************************************************************************/
#ifdef __cplusplus
extern "C"
#endif
void SQL_API_FN divid (
   sqlint32 *num,                        /* numerator */
   sqlint32 *denom,                      /* denominator */
   sqlint32 *out,                        /* output result */
   short *in1null,                       /* input 1 NULL indicator */
   short *in2null,                       /* input 2 NULL indicator */
   short *outnull,                       /* output NULL indicator */
   char *sqlstate,                       /* SQL STATE */
   char *funcname,                       /* function name */
   char *specname,                       /* specific function name */
   char *mesgtext) {                     /* message text insert */

   if (*denom == 0) {      /* if denominator is zero, return null result */
      *outnull = -1;
   } else {                /* else, compute the answer */
      *out = *num / *denom;
      *outnull = 0;
   } /* endif */
}
/* end of UDF : divid */
```

この UDF の場合、次の点に注意してください。

- 定義済み入力引き数が 2 つ、出力引き数が 1 つある。
- void を戻すように定義する。通常の UDF 出力は入力引き数を使って戻されることを覚えておいてください。
- SQL_API_FN を関数定義に含める際に、関数ソースがプラットフォームを超えて確実に移送されるように設計する。この場合、次のステートメントを UDF ソース・ファイルに入れる必要があります。

```
#include <sqlsystm.h>
```

- 入力引き数がヌルかどうかを検査しない。これは、NOT NULL CALL パラメーターが、以下に示す CREATE FUNCTION ステートメントでデフォルトとして指定されているためです。

以下に、この UDF に対する CREATE FUNCTION ステートメントを示します。

```
CREATE FUNCTION MATH."/"(INT,INT)
  RETURNS INT
  NOT FENCED
  DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  EXTERNAL NAME '/u/slick/udfx/div' ;
```

(上のステートメントは、この UDF の AIX バージョン向けです。その他のプラットフォームの場合は、EXTERNAL NAME 文節で指定された値を修正する必要があります。)

このステートメントでは次の点に注意してください。

- MATH スキーマに入れるように定義する。ユーザー ID が異なるスキーマで UDF を定義するためには、データベースでの DBADM 権限が必要です。
- 関数名は、SQL 除算演算子と同様に "/" と定義する。つまりこの UDF は、*インフィックス表記* (たとえば A / B) または*関数表記* (たとえば "/"(A,B)) のいずれかを使って、組み込み / 演算子と同様に呼び出されます。 以下を参照してください。
- プログラムにエラーがないことを確信しているので、 UDF を NOT FENCED として定義する方を選択した。
- デフォルトの NOT NULL CALL を使用した。これにより DB2 はどちらかの引き数がヌルの場合に、関数の本体を呼び出さずにヌルを結果として戻します。

ここで、次の 1 組のステートメント (CLP 入力を示す) を実行するとします。

```
SET CURRENT FUNCTION PATH = SYSIBM, SYSFUN, SLICK
SELECT INT1, INT2, INT1/INT2, "/"(INT1,INT2) FROM TEST
```

CLP から以下の出力が表示されます(データベース構成パラメーター DFT_SQLMATHWARN で *friendly 算術計算* を使用可能にしない場合)。

```
INT1        INT2        3           4
----------- ----------- ----------- -----------
         16           1          16          16
          8           2           4           4
          4           4           1           1
SQL0802N  Arithmetic overflow or other arithmetic exception occurred.
SQLSTATE=22003
```

CURRENT FUNCTION PATH の特殊レジスターを、 "/" UDF が定義されている MATH スキーマ以外のスキーマの連結に設定したので、 SQL0802N エラー・メッセージが出されます。したがって、DB2 の組み込み除算演算子を実行しており、その演算子は、「ゼロによる除算」条件が発生した際にエラーとなるよう定義されています。 TEST 表の中の 4 番目の行はこの条件を提供します。

ただし、関数のパスを変更する場合は、そのパスの SYSIBM の前に MATH を入れて SELECT ステートメントを再実行してください。

```
SET CURRENT FUNCTION PATH = MATH, SYSIBM, SYSFUN, SLICK
SELECT INT1, INT2, INT1/INT2, "/"(INT1,INT2) FROM TEST
```

次に、以下の CLP 出力が示すような要求どおりの動作を取得します。

```
INT1        INT2        3           4
----------- ----------- ----------- -----------
         16           1          16          16
          8           2           4           4
          4           4           1           1
          2           0           -           -
         97          16           6           6
  5 record(s) selected.
```

上記の例では、次の点に注意してください。

- SET CURRENT FUNCTION PATH ステートメントは、次のステートメントで使用される現行関数のパスを動的 SQL のように、 MATH スキーマを SYSIBM の前に置くよう変更する。
- 4 番目の行は除算の結果としてヌルを示し、ステートメントは継続される。
- 両方の構文 (*インフィックス表記* および *プレフィックス表記*) は、この特殊 UDF を呼び出すために使用できる。これは、特殊 UDF の名前が組み込み演算子と同じであり、その両方とも上記の例で*使用されて* 同じ結果となるためです。
- 実施上の注意として、組み込み関数および組み込み演算子が DB2 に定義されている仕方のため、 SMALLINT での演算子として "/" は使えない。 DB2 関数選択のアルゴリズムは、正確には一致しないユーザー定義の "/" よりも、正確に一致する組み込み "/" 演算子を選択します。 この表面上の矛盾を解決する方法はいろいろあります。 SMALLINT 引き数は、"/" を呼び出す前に INTEGER に明確にキャストすることができます。たとえば、`INT1 / INTEGER(SMINT1)` のようにキャストすることができます (この例で列 SMINT1 は SMALLINT と想定されます)。 また、さらによい方法は、追加の UDF を登録して "/" 演算子を多重定義し、その最初および 2 番目のパラメーターを SMALLINT として定義することです。 このような追加の UDF は `MATH."/"` に基づくことができます。

  この場合、非常に一般的な関数セットに対して次の 3 つの追加関数を CREATE して、整数除算を完全に処理しなければなりません。

```
CREATE FUNCTION MATH."/"(SMALLINT,SMALLINT)
  RETURNS INT
  SOURCE MATH."/"(INT,INT)

CREATE FUNCTION MATH."/"(SMALLINT,INT)
  RETURNS INT
  SOURCE MATH."/"(INT,INT)

CREATE FUNCTION MATH."/"(INT,SMALLINT)
  RETURNS INT
  SOURCE MATH."/"(INT,INT)
```

上の 3 つの UDF が追加されたとしても、追加のコードを MATH."/" に基づいて記述する必要はありません。

このような 4 つの "/" 関数を定義する場合に、整数除算での新しい動作を利用したい方は、関数のパスの SYSIBM の前に MATH を置くだけで通常通りに SQL を定義できます。

前の例では BIGINT データ・タイプを考慮に入れていませんが、拡張して簡単に BIGINT を組み込むことができます。

例: CLOB の折り返し、母音の検出

テキストを処理するアプリケーションに役立つ 2 つの UDF をコーディングしたと仮定します。最初の UDF は、テキスト・ストリングを *n* 番目のバイト以降で*折り返し* ます。この例でいう折り返しとは、元来は *n* バイトの後ろにあった部分を、元来は *n+1* バイトの前にあった部分に置くことを意味します。 言い換えれば、UDF は最初の n バイトをストリングの先頭から終わりに移動します。 2 番目の関数は、テキスト・ストリングの中の最初の母音の位置を戻します。これらの関数は両方とも udf.c サンプル・ファイルにコード化されています。

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sqludf.h>
#include <sqlca.h>
#include <sqlda.h>
#include "util.h"

 /*************************************************************************
*  function fold: input string is folded at the point indicated by the
*                 second argument.
*
*     input: CLOB    in1         input string
*            INTEGER in2         position to fold on
*            CLOB    out         folded string
*************************************************************************/
#ifdef __cplusplus
extern "C"
```

```
#endif
void SQL_API_FN fold (
   SQLUDF_CLOB     *in1,                 /* input CLOB to fold */
   SQLUDF_INTEGER  *in2,                 /* position to fold on */
   SQLUDF_CLOB     *out,                 /* output CLOB, folded */
   SQLUDF_NULLIND  *in1null,             /* input 1 NULL indicator */
   SQLUDF_NULLIND  *in2null,             /* input 2 NULL indicator */
   SQLUDF_NULLIND  *outnull,             /* output NULL indicator */
   SQLUDF_TRAIL_ARGS) {                  /* trailing arguments */

   SQLUDF_INTEGER len1;

   if (SQLUDF_NULL(in1null) || SQLUDF_NULL(in2null)) {
      /* one of the arguments is NULL.  The result is then "INVALID INPUT" */
      strcpy( ( char * ) out->data, "INVALID INPUT" ) ;
      out->length = strlen("INVALID INPUT");
   } else {
      len1 = in1->length;               /* length of the CLOB */

      /* build the output by folding at position "in2" */
      strncpy( ( char * ) out->data, &in1->data[*in2], len1 - *in2 ) ;
      strncpy( ( char * ) &out->data[len1 - *in2], in1->data, *in2 ) ;
      out->length = in1->length;
   } /* endif */
   *outnull = 0;                        /* result is always non-NULL */
}
/* end of UDF : fold */

 /*************************************************************************
*  function findvwl: returns the position of the first vowel.
*                 returns an error if no vowel is found
*                 when the function is created, must be defined as
*                 NOT NULL CALL.
*     inputs: VARCHAR(500) in
*     output: INTEGER      out
*************************************************************************/
#ifdef __cplusplus
extern "C"
#endif
void SQL_API_FN findvwl (
   SQLUDF_VARCHAR   *in,                 /* input character string */
   SQLUDF_SMALLINT  *out,                /* output location of vowel */
   SQLUDF_NULLIND   *innull,             /* input NULL indicator */
   SQLUDF_NULLIND   *outnull,            /* output NULL indicator */
   SQLUDF_TRAIL_ARGS) {                  /* trailing arguments */

   short i;                              /* local indexing variable */

   for (i=0; (i < (short)strlen(in) &&         /* find the first vowel */
      in[i] != 'a' && in[i] != 'e' && in[i] != 'i' &&
      in[i] != 'o' && in[i] != 'u' && in[i] != 'y' &&
      in[i] != 'A' && in[i] != 'E' && in[i] != 'I' &&
      in[i] != 'O' && in[i] != 'U' && in[i] != 'Y'); i++);
   if (i == strlen( ( char * ) in )) {               /* no vowels found */
         /* error state */
```

```
      strcpy( ( char * ) sqludf_sqlstate, "38999" ) ;
            /* message insert */
      strcpy( ( char * ) sqludf_msgtext, "findvwl: No Vowel" ) ;
   } else {                             /* a vowel was found at "i" */
      *out = i + 1;
      *outnull = 0;
   } /* endif */
}
/* end of UDF : findvwl */
```

上記の UDF では次のことに注意してください。

- sqludf.h を組み込み、そのファイルに含まれている引き数定義およびマクロを使用する。
- fold() 関数は、引き数がヌルであっても呼び出され、この場合はストリング INVALID INPUT を戻す。他方、findvwl() 関数は、引き数がヌルの場合は呼び出されません。 sqludf.h で定義される SQLUDF_NULL() マクロを使用して、 fold() におけるヌルの引き数を検査します。
- findvwl() 関数は、エラー SQLSTATE およびメッセージ・トークンを設定する。
- fold() 関数は、 CLOB データ・タイプをそのテキスト入力引き数として持つ他に、 CLOB 値を戻す。 findvwl() の入力引き数は、VARCHAR です。

以下に、これらの UDF に対する CREATE FUNCTION ステートメントを示します。

```
CREATE FUNCTION FOLD(CLOB(100K),INT)
  RETURNS CLOB(100K)
  FENCED
  DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  LANGUAGE C
  NULL CALL
  PARAMETER STYLE DB2SQL
  EXTERNAL NAME 'udf!fold' ;

CREATE FUNCTION FINDV(VARCHAR(500))
  RETURNS INTEGER
  NOT FENCED
  DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  LANGUAGE C
  NOT NULL CALL
  PARAMETER STYLE DB2SQL
  EXTERNAL NAME 'udf!findvwl' ;
```

上記の CREATE FUNCTION ステートメントは UNIX ベースのプラットフォーム用です。その他のプラットフォームの場合、上記のステートメントの EXTERNAL NAME 文節内に指定された値を修正する必要があります。 CREATE FUNCTION ステートメントは、 DB2 に添付されている calludf.sqc サンプル・プログラムの中にあります。

これらの CREATE ステートメントに関しては、次の点に注意してください。

- 関数のスキーマ名は、デフォルトにステートメントの許可 ID になる。
- エラーが起こらないという保証がないため、 FOLD を FENCED として定義する方を選択した。ただし、FINDV は NOT FENCED。
- FOLD に対して NULL CALL をコーディングした。これは、どちらかの入力引き数がヌルでも fold() が呼び出されることを意味します。つまり、関数をコーディングする方法と同じです。 FINDV は NOT NULL CALL にコーディングされますが、これもコードと一致します。
- どちらもデフォルトには ALLOW PARALLELISM になります。

以上の結果、次のステートメントを正常に実行することができます。

```
SELECT SUBSTR(DESCR,1,30), SUBSTR(FOLD(DESCR,6),1,30) FROM TEST
```

このステートメントに対する CLP からの出力結果は次のとおりです。

```
1                              2
------------------------------ ------------------------------
The only part of the body capa ly part of the body capable of
The seat of the emotions?      at of the emotions?The se
That bendy place in mid-arm.   endy place in mid-arm.That b
-                              INVALID INPUT
Unknown.                       n.Unknow
  5 record(s) selected.
```

SUBSTR 組み込み関数を使用すると、選択された CLOB 値がより見やすく表示されます。ここでは、出力が折り返される様子が示されます (2、3、および 5 行目によく示されています。これは最初の行より短い CLOB 値を持つためです。折り返しは SUBSTR を使用するとさらに明らかです)。また、入力テキスト・ストリング (列 DESCR) がヌルの場合に INVALID INPUT ストリングが FOLD UDF により戻される様子が (4 行目で) 示されます。この SELECT では、関数参照の単純なネストも示されます。 FOLD への参照は SUBSTR 関数参照の引き数の範囲内で行われます。

次に、以下のステートメントを実行するとします。

```
SELECT PART, FINDV(PART) FROM TEST
```

CLP 出力は次のようになります。

```
PART  2
----- -----------
brain           3
heart           2
elbow           1
-               -
SQL0443N  User defined function "SLICK.FINDV" (specific name
"SQL950424135144750") has returned an error SQLSTATE with diagnostic
text "findvwl: No Vowel".  SQLSTATE=38999
```

この例では、findvwl() により戻される 38999 SQLSTATE 値およびエラー・メッセージ・トークンが処理される様子を示します。メッセージ SQL0443N は、この情報をユーザーに戻します。 5 行目の PART 列には母音は含まれておらず、これは UDF 内のエラーを引き起こす条件です。

この例では、引き数のプロモーションが行われます。 PART 列は CHAR(5) であり、VARCHAR にプロモートされて FINDV に渡されます。

最後に、FINDV についての CREATE ステートメント内で NOT NULL CALL が指定された結果として、 4 行目で DB2 により FINDV からヌルの出力が生成された様子に注意してください。

次のステートメントをご覧ください。

```
SELECT SUBSTR(DESCR,1,25), FINDV(CAST (DESCR AS VARCHAR(60) ) )
FROM TEST
```

このステートメントが CLP で実行されると、以下のような出力を生成します。

```
1                          2
------------------------- -----------
The only part of the body           3
The seat of the emotions?           3
That bendy place in mid-a           3
-                                   -
Unknown.                            1
  5 record(s) selected.
```

この SELECT ステートメントでは、FINDV が VARCHAR 入力引き数で作動する様子が示されます。これを行うために列 DESCR を VARCHAR にキャストする方法に注意してください。キャストを行わないと CLOB が VARCHAR にプロモートされないため、 CLOB 引き数で FINDV を使用することはできません。この場合も、組み込み SUBSTR 関数は DESCR 列の値を見やすく表示させるために使用されます。

ここでもまた、 NOT NULL CALL のために FINDV からヌルの結果が 4 行目で生成されることに注意してください。

例: カウンター

SELECT ステートメント内の行数だけを数えたいと仮定します。このため、カウンターを増分して戻す UDF を定義します。この UDF はスクラッチパッドを使用します。

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sqludf.h>
#include <sqlca.h>
#include <sqlda.h>

/* structure scr defines the passed scratchpad for the function "ctr" */
struct scr {
```

```
   sqlint32 len;
   sqlint32 countr;
   char not_used[96];
} ;

 /*************************************************************************
*  function ctr: increments and reports the value from the scratchpad.
*
*                This function does not use the constructs defined in the
*                "sqludf.h" header file.
*
*     input:  NONE
*     output: INTEGER out      the value from the scratchpad
**************************************************************************/
#ifdef __cplusplus
extern "C"
#endif
void SQL_API_FN ctr (
   sqlint32 *out,                          /* output answer (counter) */
   short *outnull,                     /* output NULL indicator */
   char *sqlstate,                     /* SQL STATE */
   char *funcname,                     /* function name */
   char *specname,                     /* specific function name */
   char *mesgtext,                     /* message text insert */
   struct scr *scratchptr) {           /* scratch pad */

   *out = ++scratchptr->countr;      /* increment counter & copy out */
   *outnull = 0;
}
/* end of UDF : ctr */
```

この UDF の場合、次のことに注意してください。

- SQL_API_FN を定義するための sqlsystm.h はインクルードします。
- 定義される入力 SQL 引き数はないが、値は戻す。
- スクラッチパッドの入力引き数を 4 つの標準後書き引き数、すなわち *SQL 状態 (SQL-state)*、*関数名 (function-name)*、*特定名 (specific-name)*、およびメッセージ・テキスト *(message-text)* の後に追加する。
- 渡されるスクラッチパッドをマップする構造定義を含む。

以下に、この UDF に対する CREATE FUNCTION ステートメントを示します。

```
CREATE FUNCTION COUNTER()
  RETURNS INT
  SCRATCHPAD
  NOT FENCED
  NOT DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  EXTERNAL NAME 'udf!ctr'
  DISALLOW PARALLELISM;
```

(上のステートメントは、この UDF の AIX バージョン向けです。その他のプラットフォームの場合は、EXTERNAL NAME 文節で指定された値を修正する必要があります。)

このステートメントに関して、次の点に注意してください。

- 入力パラメーターを定義しない。これはコードと同じです。
- SCRATCHPAD がコーディングされ、 DB2 がそれを割り振ってスクラッチパッドの引き数を正しく初期化して受け渡す。
- エラーがないことがはっきりしているため、UDF を NOT FENCED として定義することを選択した。
- UDF は SQL の入力引き数 (この場合はない) のみに依存しないため、 UDF を NOT DETERMINISTIC に指定した。
- UDF の正常な動作は単一のスクラッチパッドに依存しているので、正しく DISALLOW PARALLELISM を指定した。

以上の結果、次のステートメントを正常に実行することができます。

```
SELECT INT1, COUNTER(), INT1/COUNTER() FROM TEST
```

このステートメントを CLP を介して実行すると、以下のように出力されます。

```
INT1        2           3
----------- ----------- -----------
         16           1          16
          8           2           4
          4           3           1
          2           4           0
         97           5          19
  5 record(s) selected.
```

2 番目の列には、COUNTER() 出力がそのまま示されていることに注目してください。 3 番目の列には、SELECT ステートメント内の COUNTER() に対する 2 つの別個の参照が、それぞれ独自のスクラッチパッドを獲得することが示されます。それぞれが独自のスクラッチパッドを獲得しなかった場合、 2 番目の列の出力結果は 1 2 3 4 5 という正しい順序とはならず、 1 3 5 7 9 となります。

例: 天気表関数

次に示す例は、表関数 tfweather_u (DB2 のサンプル・プログラム tblsrv.c に収録) で、この関数は米国の各都市の気象情報を戻します。各都市の天気データがサンプル・プログラムに組み込まれていますが、プログラム中の注釈が示すように、天気データを外部ファイルから読み込むことも可能です。このデータには、都市の名前とそれに続いてその都市の気象情報が入っています。この順番で、その他の都市についても繰り返されます。 DB2 には、この表関数を呼び出し、取り出された天気データを tfweather_u 表関数を使用して印刷するクライアント・アプリケーション (tblcli.sqc) が付属しています。

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sql.h>
#include <sqludf.h> /* for use in compiling User Defined Function */

#define   SQL_NOTNULL   0   /* Nulls Allowed - Value is not Null */
#define   SQL_ISNULL   -1   /* Nulls Allowed - Value is Null */

/* Short and long city name structure */
typedef struct {
  char * city_short ;
  char * city_long ;
} city_area ;

/* Scratchpad data */
/* Preserve information from one function call to the next call */
typedef struct {
  /* FILE * file_ptr; if you use weather data text file */
  int file_pos ;  /* if you use a weather data buffer */
} scratch_area ;

/* Field descriptor structure */
typedef struct {
  char fld_field[31] ;                     /* Field data */
  int  fld_ind ;            /* Field null indicator data */
  int  fld_type ;                          /* Field type */
  int  fld_length ;  /* Field length in the weather data */
  int  fld_offset ;  /* Field offset in the weather data */
} fld_desc ;

/* Short and long city name data */
city_area cities[] = {
  { "alb", "Albany, NY"                  },
  { "atl", "Atlanta, GA"                 },
  .
  .
  .
  { "wbc", "Washington DC, DC"           },
  /* You may want to add more cities here */

  /* Do not forget a null termination */
  { ( char * ) 0, ( char * ) 0           }
} ;

/* Field descriptor data */
fld_desc fields[] = {
  { "", SQL_ISNULL, SQL_TYP_VARCHAR, 30,  0 }, /* city          */
  { "", SQL_ISNULL, SQL_TYP_INTEGER,  3,  2 }, /* temp_in_f     */
  { "", SQL_ISNULL, SQL_TYP_INTEGER,  3,  7 }, /* humidity      */
  { "", SQL_ISNULL, SQL_TYP_VARCHAR,  5, 13 }, /* wind          */
  { "", SQL_ISNULL, SQL_TYP_INTEGER,  3, 19 }, /* wind_velocity */
  { "", SQL_ISNULL, SQL_TYP_FLOAT,    5, 24 }, /* barometer     */
  { "", SQL_ISNULL, SQL_TYP_VARCHAR, 25, 30 }, /* forecast      */
  /* You may want to add more fields here */
```

```
  /* Do not forget a null termination */
  { ( char ) 0, 0, 0, 0, 0 }
} ;

/* Following is the weather data buffer for this example. You */
/* may want to keep the weather data in a separate text file. */
/* Uncomment the following fopen() statement.  Note that you  */
/* have to specify the full path name for this file.          */
char * weather_data[] = {
   "alb.forecast",
   "   34   28%    wnw   3  30.53 clear",
   "atl.forecast",
   "   46   89%   east  11  30.03 fog",
   .
   .
   .
   "wbc.forecast",
   "   38   96%    ene  16  30.31 light rain",
   /* You may want to add more weather data here */

   /* Do not forget a null termination */
   ( char * ) 0
} ;

#ifdef __cplusplus
extern "C"
#endif
/* This is a subroutine. */
/* Find a full city name using a short name */
int get_name( char * short_name, char * long_name ) {

    int name_pos = 0 ;

    while ( cities[name_pos].city_short != ( char * ) 0 ) {
       if (strcmp(short_name, cities[name_pos].city_short) == 0) {
          strcpy( long_name, cities[name_pos].city_long ) ;
          /* A full city name found */
          return( 0 ) ;
       }
       name_pos++ ;
    }
    /* Could not find such city in the city data */
    strcpy( long_name, "Unknown City" ) ;
    return( -1 ) ;

}

#ifdef __cplusplus
extern "C"
#endif
/* This is a subroutine. */
/* Clean all field data and field null indicator data */
int clean_fields( int field_pos ) {
```

```
    while ( fields[field_pos].fld_length != 0 ) {
       memset( fields[field_pos].fld_field, '¥0', 31 ) ;
       fields[field_pos].fld_ind = SQL_ISNULL ;
       field_pos++ ;
    }
    return( 0 ) ;

}

#ifdef __cplusplus
extern "C"
#endif
/* This is a subroutine. */
/* Fills all field data and field null indicator data ... */
/* ... from text weather data */
int get_value( char * value, int field_pos ) {

    fld_desc * field ;
    char field_buf[31] ;
    double * double_ptr ;
    int * int_ptr, buf_pos ;

    while ( fields[field_pos].fld_length != 0 ) {
       field = &fields[field_pos] ;
       memset( field_buf, '¥0', 31 ) ;
       memcpy( field_buf,
               ( value + field->fld_offset ),
               field->fld_length ) ;
       buf_pos = field->fld_length ;
       while ( ( buf_pos > 0 ) &&
               ( field_buf[buf_pos] == ' ' ) )
          field_buf[buf_pos--] = '¥0' ;
       buf_pos = 0 ;
       while ( ( buf_pos < field->fld_length ) &&
               ( field_buf[buf_pos] == ' ' ) )
          buf_pos++ ;
       if ( strlen( ( char * ) ( field_buf + buf_pos ) ) > 0 ||
            strcmp( ( char * ) ( field_buf + buf_pos ), "n/a") != 0 ) {
          field->fld_ind = SQL_NOTNULL ;

          /* Text to SQL type conversion */
          switch( field->fld_type ) {
            case SQL_TYP_VARCHAR:
                 strcpy( field->fld_field,
                         ( char * ) ( field_buf + buf_pos ) ) ;
                 break ;
            case SQL_TYP_INTEGER:
                 int_ptr = ( int * ) field->fld_field ;
                 *int_ptr = atoi( ( char * ) ( field_buf + buf_pos ) ) ;
                 break ;
            case SQL_TYP_FLOAT:
                 double_ptr = ( double * ) field->fld_field ;
                 *double_ptr = atof( ( char * ) ( field_buf + buf_pos ) ) ;
                 break ;
            /* You may want to add more text to SQL type conversion here */
```

```
          }

        }
        field_pos++ ;
    }
    return( 0 ) ;

}

#ifdef __cplusplus
extern "C"
#endif
void SQL_API_FN weather( /* Return row fields */
              SQLUDF_VARCHAR * city,
              SQLUDF_INTEGER * temp_in_f,
              SQLUDF_INTEGER * humidity,
              SQLUDF_VARCHAR * wind,
              SQLUDF_INTEGER * wind_velocity,
              SQLUDF_DOUBLE  * barometer,
              SQLUDF_VARCHAR * forecast,
              /* You may want to add more fields here */

              /* Return row field null indicators */
              SQLUDF_NULLIND * city_ind,
              SQLUDF_NULLIND * temp_in_f_ind,
              SQLUDF_NULLIND * humidity_ind,
              SQLUDF_NULLIND * wind_ind,
              SQLUDF_NULLIND * wind_velocity_ind,
              SQLUDF_NULLIND * barometer_ind,
              SQLUDF_NULLIND * forecast_ind,
              /* You may want to add more field indicators here */

              /* UDF always-present (trailing) input arguments */
              SQLUDF_TRAIL_ARGS_ALL
            ) {

  scratch_area * save_area ;
  char line_buf[81] ;
  int line_buf_pos ;

  /* SQLUDF_SCRAT is part of SQLUDF_TRAIL_ARGS_ALL */
  /* Preserve information from one function call to the next call */
  save_area = ( scratch_area * ) ( SQLUDF_SCRAT->data ) ;

  /* SQLUDF_CALLT is part of SQLUDF_TRAIL_ARGS_ALL */
  switch( SQLUDF_CALLT ) {

    /* First call UDF: Open table and fetch first row */
    case SQL_TF_OPEN:
         /* If you use a weather data text file specify full path */
         /* save_area->file_ptr = fopen("/sqllib/samples/c/tblsrv.dat",
                                        "r"); */
         save_area->file_pos = 0 ;
         break ;
```

```
/* Normal call UDF: Fetch next row */
case SQL_TF_FETCH:
     /* If you use a weather data text file */
     /* memset(line_buf, '¥0', 81); */
     /* if (fgets(line_buf, 80, save_area->file_ptr) == NULL) { */
     if ( weather_data[save_area->file_pos] == ( char * ) 0 ) {

        /* SQLUDF_STATE is part of SQLUDF_TRAIL_ARGS_ALL */
        strcpy( SQLUDF_STATE, "02000" ) ;

        break ;
     }
     memset( line_buf, '¥0', 81 ) ;
     strcpy( line_buf, weather_data[save_area->file_pos] ) ;
     line_buf[3] = '¥0' ;

     /* Clean all field data and field null indicator data */
     clean_fields( 0 ) ;

     /* Fills city field null indicator data */
     fields[0].fld_ind = SQL_NOTNULL ;

     /* Find a full city name using a short name */
     /* Fills city field data */
     if ( get_name( line_buf, fields[0].fld_field ) == 0 ) {
        save_area->file_pos++ ;
        /* If you use a weather data text file */
        /* memset(line_buf, '¥0', 81); */
        /* if (fgets(line_buf, 80, save_area->file_ptr) == NULL) { */
        if ( weather_data[save_area->file_pos] == ( char * ) 0 ) {
           /* SQLUDF_STATE is part of SQLUDF_TRAIL_ARGS_ALL */
           strcpy( SQLUDF_STATE, "02000" ) ;
           break ;
        }
        memset( line_buf, '¥0', 81 ) ;
        strcpy( line_buf, weather_data[save_area->file_pos] ) ;
        line_buf_pos = strlen( line_buf ) ;
        while ( line_buf_pos > 0 ) {
           if ( line_buf[line_buf_pos] >= ' ' )
              line_buf_pos = 0 ;
           else {
              line_buf[line_buf_pos] = '¥0' ;
              line_buf_pos-- ;
           }
        }
     }

     /* Fills field data and field null indicator data ... */
     /* ... for selected city from text weather data */
     get_value( line_buf, 1 ) ;  /* Skips city field */

     /* Builds return row fields */
     strcpy( city, fields[0].fld_field ) ;
     memcpy( (void *) temp_in_f,
             fields[1].fld_field,
```

```
                sizeof( SQLUDF_INTEGER ) ) ;
        memcpy( (void *) humidity,
                fields[2].fld_field,
                sizeof( SQLUDF_INTEGER ) ) ;
        strcpy( wind, fields[3].fld_field ) ;
        memcpy( (void *) wind_velocity,
                fields[4].fld_field,
                sizeof( SQLUDF_INTEGER ) ) ;
        memcpy( (void *) barometer,
                fields[5].fld_field,
                sizeof( SQLUDF_DOUBLE ) ) ;
        strcpy( forecast, fields[6].fld_field ) ;

        /* Builds return row field null indicators */
        memcpy( (void *) city_ind,
                &(fields[0].fld_ind),
                sizeof( SQLUDF_NULLIND ) ) ;
        memcpy( (void *) temp_in_f_ind,
                &(fields[1].fld_ind),
                sizeof( SQLUDF_NULLIND ) ) ;
        memcpy( (void *) humidity_ind,
                &(fields[2].fld_ind),
                sizeof( SQLUDF_NULLIND ) ) ;
        memcpy( (void *) wind_ind,
                &(fields[3].fld_ind),
                sizeof( SQLUDF_NULLIND ) ) ;
        memcpy( (void *) wind_velocity_ind,
                &(fields[4].fld_ind),
                sizeof( SQLUDF_NULLIND ) ) ;
        memcpy( (void *) barometer_ind,
                &(fields[5].fld_ind),
                sizeof( SQLUDF_NULLIND ) ) ;
        memcpy( (void *) forecast_ind,
                &(fields[6].fld_ind),
                sizeof( SQLUDF_NULLIND ) ) ;

        /* Next city weather data */
        save_area->file_pos++ ;

        break ;

    /* Special last call UDF for cleanup (no real args!): Close table */
    case SQL_TF_CLOSE:
        /* If you use a weather data text file */
        /* fclose(save_area->file_ptr); */
        /* save_area->file_ptr = NULL; */
        save_area->file_pos = 0 ;
        break ;

  }

}
```

この UDF コードでは、次のことに注意してください。

- スクラッチパッドが定義される。 row 変数は OPEN 呼び出しで初期化され、 iptr 配列と nbr_rows 変数は OPEN 時に *mystery* 関数によって埋められます。
- FETCH は、row を索引として使用しながら iptr 配列を横断し、関係のある値を iptr の現行エレメントから out_c1、 out_c2、out_c3 結果値ポインターが指すロケーションに移動する。
- 最後に、OPEN によって獲得されてスクラッチパッドに固定されたストレージを CLOSE が解放する。

以下に、この UDF に対する CREATE FUNCTION ステートメントを示します。

```
CREATE FUNCTION tfweather_u()
  RETURNS TABLE (CITY VARCHAR(25),
                 TEMP_IN_F INTEGER,
                 HUMIDITY INTEGER,
                 WIND VARCHAR(5),
                 WIND_VELOCITY INTEGER,
                 BAROMETER FLOAT,
                 FORECAST VARCHAR(25))
  SPECIFIC tfweather_u
  DISALLOW PARALLELISM
  NOT FENCED
  DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  SCRATCHPAD
  NO FINAL CALL
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  EXTERNAL NAME 'tf_dml!weather';
```

上の CREATE FUNCTION ステートメントは、この UDF の UNIX バージョン用です。その他のプラットフォームの場合は、EXTERNAL NAME 文節で指定された値を修正する必要があります。

このステートメントに関して、次の点に注意してください。

- 入力は取らず、7 つの出力列を戻す。
- SCRATCHPAD が指定されるので、DB2 はこれを割り振り、スクラッチパッド引き数を適切に初期化して渡す。
- NO FINAL CALL が指定される。
- 関数は、SQL の入力引き数のみに依存しないので、NOT DETERMINISTIC として指定される。つまり、mystery 関数に依存し、実行ごとに内容が異なると想定しています。
- DISALLOW PARALLELISM は表関数には必須である。
- CARDINALITY 100 は、戻される行数の予想値で、DB2 最適化プログラムに渡される。

- DBINFO は使用されず、関数を参照する特定のステートメントに必要な列のみを戻す最適化はインプリメントされません。
- NOT NULL CALL が指定されるので、入力 SQL 引き数のどれかがヌルの場合、この UDF は呼び出されない。また、この条件を調べる必要はありません。

例: LOB ロケーターを使用する関数

この UDF は、入力 LOB にロケーターを取り、この入力 LOB のサブセットである別の LOB にロケーターを戻します。 2 番目の入力値として一定の基準が渡され、それにより正確に入力 LOB を分ける方法を UDF に伝えます。

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sql.h>
#include <sqlca.h>
#include <sqlda.h>
#include <sqludf.h>
#include "util.h"

void SQL_API_FN lob_subsetter(
          udf_locator * lob_input,   /* locator of LOB value to carve up */
          char     * criteria,         /* criteria for carving */
          udf_locator    * lob_output,  /* locator of result LOB value */
          sqlint16 * inp_nul,
          sqlint16 * cri_nul,
          sqlint16 * out_nul,
          char     * sqlstate,
          char     * funcname,
          char     * specname,
          char     * msgtext ) {

       /* local vars */
       short j;              /* local indexing var */
       int   rc;             /* return code variable for API calls */
       sqlint32  input_len;  /* receiver for input LOB length */
       sqlint32  input_pos;  /* current position for scanning input LOB */
       char lob_buf[100];    /* data buffer */
       sqlint32  input_rec;  /* number of bytes read by sqludf_substr */
       sqlint32  output_rec; /* number of bytes written by sqludf_append */

       /*---------------------------------------------
        * UDF Program Logic Starts Here
        *---------------------------------------------
        * What we do is create an output handle, and then
        * loop over the input, 100 bytes at a time.
        * Depending on the "criteria" passed in, we may decide
        * to append the 100 byte input lob segment to the output, or not.
        *---------------------------------------------
        * Create the output locator, right in the return buffer.
        */

       rc = sqludf_create_locator(SQL_TYP_CLOB, &lob_output);
```

```
       /* Error and exit if unable to create locator */
       if (rc) {
          memcpy (sqlstate, "38901", 5);
          /* special sqlstate for this condition */
          goto exit;
       }
       /* Find out the size of the input LOB value */
       rc = sqludf_length(lob_input, &input_len) ;
       /* Error and exit if unable to find out length */
       if (rc) {
          memcpy (sqlstate, "38902", 5);
          /* special sqlstate for this condition */
          goto exit;
       }
       /* Loop to read next 100 bytes, and append to result if it meets
        * the criteria.
        */
       for (input_pos = 0; (input_pos < input_len); input_pos += 100) {
         /* Read the next 100 (or less) bytes of the input LOB value */
         rc = sqludf_substr(lob_input, input_pos, 100,
                            (unsigned char *) lob_buf, &input_rec) ;
         /* Error and exit if unable to read the segment */
         if (rc) {
            memcpy (sqlstate, "38903", 5);
            /* special sqlstate for this condition */
            goto exit;
         }
         /* apply the criteria for appending this segment to result
          * if (...predicate involving buffer and criteria...) {
          * The condition for retaining the segment is TRUE...
          * Write that buffer segment which was last read in
          */
         rc = sqludf_append(lob_output,
                    (unsigned char *) lob_buf, input_rec, &output_rec) ;
         /* Error and exit if unable to read the 100 byte segment */
         if (rc) {
            memcpy (sqlstate, "38904", 5);
            /* special sqlstate for this condition */
            goto exit;
         }
         /* } end if criteria for inclusion met */
       } /* end of for loop, processing 100-byte chunks of input LOB
         * if we fall out of for loop, we are successful, and done.
 */
   *out_nul = 0;
exit: /* used for errors, which will override null-ness of output. */
   return;
}
```

この UDF コードでは、次のことに注意してください。

- sqludf_create_locator() 呼び出しで使用されるタイプ SQL_TYP_CLOB を定義する sql.h と、タイプ udf_locator を定義する sqludf.h がインクルードされる。

- 最初の入力引き数と 3 番目の入力引き数 (関数の出力を表す) は、 sqludf_locator へのポインターとして定義される。つまり、これらの入力引き数は CREATE FUNCTION における AS LOCATOR の指定を表します。
- CREATE FUNCTION ステートメントで NOT NULL CALL が指定されているので、UDF はいずれの入力引き数についてもヌルであるかどうかのテストは行わない。
- エラーが発生した場合、 UDF は sqlstate を 38*xxx* に設定して終了する。これは、UDF を参照しているステートメントの実行を停止するのに十分です。選択する実際の 38*xxx* SQLSTATE 値は DB2 にとって重要ではなく、使用する UDF で例外条件が生じた場合にそれを区別する役目を果たします。
- 包含基準には何も指定されていない。この場合には、何らかの仕方でこの特定のバッファー内容がテストを通るかどうかを判別し、最後のバッファーが部分バッファーであるかどうかを示すよう想定されています。
- 追加されるデータの長さとして input_rec 変数を使用することにより、 UDF は部分バッファー条件に対応します。

以下に、この UDF に対する CREATE FUNCTION ステートメントを示します。

```
CREATE FUNCTION carve(CLOB(50M), VARCHAR(255) )
  RETURNS CLOB(50M)
  NOT NULL CALL
  NOT FENCED
  DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  EXTERNAL NAME '/u/wilfred/udfs/lobudfs!lob_subsetter' ;
```

(上のステートメントは、この UDF の AIX バージョン向けです。その他のプラットフォームの場合は、EXTERNAL NAME 文節で指定された値を修正する必要があります。)

このステートメントに関して、次の点に注意してください。

- NOT NULL CALL が指定されるので、入力 SQL 引き数のどれかがヌルの場合、この UDF は呼び出されない。また、この条件を調べる必要はありません。
- 関数は NOT FENCED に定義される。 API は NOT FENCED でのみ動作することを思い出してください。 NOT FENCED は、定義者がデータベース上で CREATE_NOT_FENCED 権限を持っている必要があることを意味します (これは、DBADM 権限によって暗黙的に指定されます)。
- この関数は DETERMINISTIC として指定される。これは、特定の入力 CLOB と特定の一連の基準について、結果は常に同じであることを意味します。

以上の結果、次のステートメントを正常に実行することができます。

```
UPDATE tablex
  SET col_a = 99,
      col_b = carve (:hv_clob, '...criteria...')
  WHERE tablex_key = :hv_key;
```

この UDF を使用すると、ホスト変数 :hv_clob によって表される CLOB 値をサブセット化し、ホスト変数 :hv_key によって表される行を更新できます。

この更新例では、:hv_clob をアプリケーション内で CLOB_LOCATOR として定義することができます。″carve″ UDF に渡されるのは、このロケーターではありません。:hv_clob は、ステートメントを実行している DB2 中心エージェントに「バインド」されている場合、 CLOB としてのみ認識されます。これが UDF に渡されると、DB2 はその値の新しいロケーターを生成します。 CLOB とロケーターの間でのこの変換の繰り返しは、資源を消費するものではありません。これには、余分のメモリー・コピーや入出力は関係しません。

例: BASIC でのカウンター OLE オートメーション UDF

次の例では、Microsoft Visual BASIC を使用してカウンター・クラスをインプリメントします。 カウンター・クラスにはインスタンス変数 nbrOfInvoke があり、呼び出しの数を記録します。 このクラスのコンストラクターはその数を 0 に初期化します。増分メソッドは nbrOfInvoke を 1 ずつ増分し、現在の状態を戻します。

```
Description="Example in SQL Reference"
Name="bert"
Class=bcounter; bcounter.cls
ExeName32="bert_app.exe"


VERSION 1.0 CLASS
BEGIN
  SingleUse = -1  'True
END
Attribute VB_Name = "bcounter"
Attribute VB_Creatable = True
Attribute VB_Exposed = True
Option Explicit
Dim nbrOfInvoke As Long

Public Sub increment(output As Long, _
                     output_ind As Integer, _
                     sqlstate As String, _
                     fname As String, _
                     fspecname As String, _
                     msg As String, _
                     scratchpad() As Byte, _
                     calltype As Long)

    nbrOfInvoke = nbrOfInvoke + 1
```

```
End Sub

Private Sub Class_Initialize()
  nbrOfInvoke = 0
End Sub

Private Sub Class_Terminate()

End Sub
```

bcounter クラスは OLE オートメーション・オブジェクトとしてインプリメントされ、progId bert.bcounter の下で登録されています。オートメーション・サーバーは、プロセス内サーバーまたはローカル・サーバーとしてコンパイルできます。これは DB2 に対して透過です。次の CREATE FUNCTION ステートメントは、 UDF bcounter を increment メソッドに外部インプリメンテーションとして登録します。

```
CREATE FUNCTION bcounter () RETURNS integer
   EXTERNAL NAME 'bert.bcounter!increment'
   LANGUAGE OLE
   FENCED
   SCRATCHPAD
   FINAL CALL
   NOT DETERMINISTIC
   NULL CALL
   PARAMETER STYLE DB2SQL
   NO SQL
   NO EXTERNAL ACTION
   DISALLOW PARALLEL;
```

以下の照会を実行するとします。

```
SELECT INT1, BCOUNTER() AS COUNT, INT1/BCOUNTER() AS DIV FROM TEST
```

結果は、前の例とまったく同じです。

```
INT1        COUNT       DIV
----------- ----------- -----------
         16           1          16
          8           2           4
          4           3           1
          2           4           0
         97           5          19

  5 record(s) selected.
```

例: C++ でのカウンター OLE オートメーション UDF

次の例は、前に挙げた BASIC のカウンター・クラスを C++ でインプリメントします。ここでは、コードの一部分のみを示します。サンプル全体のリストは、/sqllib/samples/ole ディレクトリーにあります。

増分メソッドは、カウンター・インターフェースの記述の一部として、オブジェクト記述言語で記述されます。

```
interface ICounter : IDispatch
{
    ...
    HRESULT increment([out] long    *out,
                      [out] short   *outnull,
                      [out] BSTR    *sqlstate,
                      [in]  BSTR    *fname,
                      [in]  BSTR    *fspecname,
                      [out] BSTR    *msgtext,
                      [in,out] SAFEARRAY (unsigned char) *spad,
                      [in]  long    *calltype);
    ...
}
```

C++ での COM CCounter クラス定義には、 nbrOfInvoke と同様に increment メソッドの宣言が含まれます。

```
class FAR CCounter : public ICounter
{
        ...
        STDMETHODIMP CCounter::increment(long    *out,
                                         short   *outnull,
                                         BSTR    *sqlstate,
                                         BSTR    *fname,
                                         BSTR    *fspecname,
                                         BSTR    *msgtext,
                                         SAFEARRAY **spad,
                                         long *calltype );
        long nbrOfInvoke;
        ...
};
```

このメソッドの C++ インプリメンテーションは、BASIC コードと似ています。

```
STDMETHODIMP CCounter::increment(long    *out,
                                 short   *outnull,
                                 BSTR    *sqlstate,
                                 BSTR    *fname,
                                 BSTR    *fspecname,
                                 BSTR    *msgtext,
                                 SAFEARRAY **spad,
                                 long *calltype)
{

  nbrOfInvoke = nbrOfInvoke + 1;
  *out = nbrOfInvoke;

  return NOERROR;
};
```

上の例では、sqlstate と msgtext は、タイプ BSTR* の [out] パラメーターです。つまり、DB2 はヌルへのポインターをこの UDF に渡します。これらのパラメーターの値を戻すのに、 UDF はストリングを割り振ってそれを DB2 に戻し (たとえば、 *sqlstate = SysAllocString (L"01H00") とする)、 DB2 はメモリーを解放します。パ

ラメーター fname および fspecname は、[in] パラメーターです。 DB2 は、メモリーを割り振り、その中に UDF によって読み取られる値を渡し、それからそのメモリーを解放します。

クラス CCounter のクラス factory はカウンター・オブジェクトを作成します。クラス factory は、単独使用または複数使用のオブジェクトとして登録できます (この例では示されません)。

```
STDMETHODIMP CCounterCF::CreateInstance(IUnknown FAR* punkOuter,
                                        REFIID riid,
                                        void FAR* FAR* ppv)
{

  CCounter *pObj;
        ...
  // create a new counter object
  pObj = new CCounter;
        ...
};
```

CCounter クラスは、ローカル・サーバーとしてインプリメントされ、 progId bert.ccounter の下に登録されます。次の CREATE FUNCTION ステートメントは、UDF ccounter を外部インプリメンテーションとして increment メソッドに登録します。

```
CREATE FUNCTION ccounter () RETURNS integer
  EXTERNAL NAME 'bert.ccounter!increment'
  LANGUAGE OLE
  FENCED
  SCRATCHPAD
  FINAL CALL
  NOT DETERMINISTIC
  NULL CALL
  PARAMETER STYLE DB2SQL
  NO SQL
  NO EXTERNAL ACTION
  DISALLOW PARALLEL;
```

次の照会を処理している間、 DB2 はクラス CCounter の 2 つの異なるインスタンスを作成します。 照会で UDF を参照するごとに 1 つのインスタンスが作成されます。 ccounter UDF 登録で scratchpad オプションが指定されているので、この 2 つのインスタンスは照会全体で再利用できます。

```
SELECT INT1, CCOUNTER() AS COUNT, INT1/CCOUNTER() AS DIV FROM TEST
```

結果は、前の例とまったく同じです。

```
INT1        COUNT       DIV
----------- ----------- -----------
         16           1          16
          8           2           4
          4           3           1
```

```
          2            4            0
         97            5           19

  5 record(s) selected.
```

例: BASIC でのメール OLE オートメーション表関数

次の例は、Microsoft Visual BASIC を使用してクラスをインプリメントします。このクラスには、メッセージ・ヘッダー情報とメッセージの部分メッセージ・テキストを Microsoft Exchange で取得するパブリック・メソッド・リストがあります。 このメソッドのインプリメンテーションは、 MAPI (メッセージング API) に対する OLE オートメーション・インターフェースを提供する OLE メッセージ交換を利用します。

```
Description="Mail OLE Automation Table Function"
Module=MainModule; MainModule.bas
Class=Header; Header.cls
ExeName32="tfmapi.dll"
Name="TFMAIL"

VERSION 1.0 CLASS
BEGIN
  MultiUse = -1  'True
END
Attribute VB_Name = "Header"
Attribute VB_Creatable = True
Attribute VB_Exposed = True
Option Explicit

Dim MySession As Object
Dim MyMsgColl As Object
Dim MyMsg As Object
Dim CurrentSender As Object
Dim name As Variant
Const SQL_TF_OPEN = -1
Const SQL_TF_CLOSE = 1
Const SQL_TF_FETCH = 0


Public Sub List(timereceived As Date, subject As String, size As Long, _
                text As String, ind1 As Integer, ind2 As Integer, _
                ind3 As Integer, ind4 As Integer, sqlstate As String, _
                fname As String, fspecname As String, msg As String, _
                scratchpad() As Byte, calltype As Long)


    If (calltype = SQL_TF_OPEN) Then

       Set MySession = CreateObject("MAPI.Session")

       MySession.Logon ProfileName:="Profile1"
       Set MyMsgColl = MySession.Inbox.Messages

       Set MyMsg = MyMsgColl.GetFirst
```

```
    ElseIf (calltype = SQL_TF_CLOSE) Then

       MySession.Logoff
       Set MySession = Nothing

    Else

      If (MyMsg Is Nothing) Then

         sqlstate = "02000"

      Else

         timereceived = MyMsg.timereceived
         subject = Left(MyMsg.subject, 15)
         size = MyMsg.size
         text = Left(MyMsg.text, 30)

         Set MyMsg = MyMsgColl.GetNext

      End If
    End If
End Sub
```

表関数 OPEN を呼び出すと、CreateObject ステートメントがメール・セッションを作成し、 logon メソッドがメール・システムにログオンします (ユーザー名とパスワードの発行は省略されます)。メール・インボックスのメッセージ収集により、最初のメッセージが取得されます。 FETCH 呼び出しがあると、メッセージ・ヘッダー情報と、現在のメッセージの最初の 30 文字が、表関数出力パラメーターに割り当てられます。メッセージが残っていない場合は、SQLSTATE 02000 が戻されます。 CLOSE 呼び出しがあると、この例はログオフしてセッション・オブジェクトを何もない状態に設定します。これにより、前の参照オブジェクトに関連したシステムおよびメモリー・リソースは、他の変数からオブジェクトが参照されない場合に解放されます。

以下に、この UDF に対する CREATE FUNCTION ステートメントを示します。

```
CREATE FUNCTION MAIL()
    RETURNS TABLE (TIMERECIEVED DATE,
                   SUBJECT VARCHAR(15),
                   SIZE INTEGER,
                   TEXT VARCHAR(30))
    EXTERNAL NAME 'tfmail.header!list'
    LANGUAGE OLE
    PARAMETER STYLE DB2SQL
    NOT DETERMINISTIC
    FENCED
    NULL CALL
    SCRATCHPAD
    FINAL CALL
    NO SQL
    EXTERNAL ACTION
    DISALLOW PARALLEL;
```

照会の例を次に示します。

```
SELECT * FROM TABLE (MAIL()) AS M

TIMERECEIVED SUBJECT         SIZE        TEXT
------------ --------------- ----------- ------------------------------
01/18/1997   Welcome!               3277 Welcome to Windows Messaging!
01/18/1997   Invoice                1382 Please process this invoice. T
01/19/1997   Congratulations        1394 Congratulations to the purchas

  3 record(s) selected.
```

UDF のデバッグ

UDF は、データベースに損傷を与える恐れのない環境でデバッグすることが大切です。UDF が正しいという確信が得られるまで、テスト・データベース・インスタンスでテストする必要があります。このことは、FENCED や NOT FENCED UDF にもあてはまります。これらのタイプはいずれも、正しく作成されていないと DB2 を誤動作させる可能性があるからです。 UDF を FENCED として定義したほうが、 NOT FENCED として定義するよりも保全性の維持や機密漏れからの保護が得られますが、確実な保証はありません。いずれにせよ、十分な検討やテストを重ねることを含め、ふさわしい仕方でコーディングすることが必要になります。

DB2 は、ストレージを誤って修正する特定の限られたタイプの処置があるかを検査します (たとえば、 UDF が、多すぎる文字をスクラッチパッドや結果のバッファーに移動する場合)。この場合、そうした誤動作が検出されると DB2 はエラー SQLCODE -450 (SQLSTATE 39501) を戻します。また DB2 は、UDF が異常終了して SQLCODE -430 (SQLSTATE 38503) を戻したり、ユーザーが UDF を中断して SQLCODE -431 (SQLSTATE 38504) を戻したりする場合に、秩序正しく終了するように設計されています。

FENCED UDF を選択している場合であっても、戻り値バッファーを何回も上書きすることは、 UDF と DB2 を異常終了させる原因になり得ます。バイトを戻り値バッファーに移動させる UDF を設計、コーディング、検討する際には十分な注意を払ってください。たとえば、移動前に移動の必要があるバイト数を計算する UDF を作成する場合には、特に注意してください。 C では、この役割を果たす関数として memcpy が使用されるのが一般的です。バイトを戻り値バッファーに移動させる UDF の場合には、境界ケース (余分な short 値や long 値) を綿密に調べてください。

セキュリティーとデータベースの保全性にとって、 UDF を DB2 に対していったんデバッグして定義してしまえば、 UDF の本体を保護することが大切です。これは特に、UDF が NOT FENCED として定義されている場合に重要です。誰かが (自分自身を含む) 不慮または故意に、デバッグされていないコードを使って使用可能な UDF を上書きすると、 UDF はデータベースが非分離の場合に、そのデータベースを破棄するか、あるいはデータベースを安全に保護できないと考えられます。

残念ながら、ソース・レベルのデバッガーを UDF で簡単に実行する方法はありません。それにはいくつかの理由があります。

- UDF がストレージにあって利用可能な場合に、デバッガーを一度に開始するタイミングが難しい。
- UDF は特殊ユーザー ID を使ったデータベース処理で実行されるため、ユーザーがこの処理に接続できない。

UDF は、通常は `stdout` が意味を持たないバックグラウンド処理で実行されるため、`printf()` などの有効なデバッグ・ツールは UDF のデバッグには役立たないことに注意してください。 `printf()` を使用する代わりに、UDF にファイル出力論理を備えたり、デバッグする目的で表示データおよび制御情報をファイルに書き込むことができます。

UDF をデバッグするその他の技法は、ドライバーのプログラムを記述して UDF をデータベース環境の外側に呼び出す方法です。この技法を使うと、欄外に書いたか誤った全種類の入力引き数を使って UDF を呼び出し、それを不正に実行させることができます。この環境では、 `printf()` またはソース・レベルのデバッガーを使用することは問題ではありません。

第16章 活動状態の DBMS でのトリガーの使用

トリガーを使う理由 499
  トリガーの利点 500
トリガーの概説 501
トリガー・イベント 502
影響される行のセット 503
トリガーの細分性 503
トリガー活動化時間 503
変位変数 505
変換表 506
トリガー・アクション 507
  トリガー・アクション条件 508
  トリガー SQL ステートメント 508
  SQL トリガー・ステートメント内の関数 509
トリガーのカスケード 510
参照制約との対話 510
複数トリガーの順序付け 510
トリガー、制約、UDT、UDF、および LOB 間の協調 511
  情報の抽出 512
  表における操作の妨害 512
  業務規則の定義 513
  アクションの定義 513

トリガーを使う理由

データベース・マネージャーを非活動状態のシステムから活動状態のシステムに変えるには、トリガー関数で表される機能を使用してください。*トリガー* は、特定の基本表に対する更新操作によって活動化される、または*起動される* 一連のアクションを定義します。これらのアクションは、データベースに他の変更を行ったり、 DB2 の外で操作を実行したり (たとえば、電子メールを送ったり、ファイルにレコードを書き込む)、更新操作を妨害する例外を起こしたりすることがあります。

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、トリガーを使っ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。この規則を検査するには、増える前と後の給料の額を比較しなければなりません。データの複数の状態が関係していない規則では、検査と参照保全に関する制約がより適していることがあります (詳細については、*SQL 解説書* を参照してください)。検査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、要約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 (as in Sample Tables)
  COMPANY_STATS (NBEMP, NBPRODUCT, REVENUE)
```

© Copyright IBM Corp. 1993, 2001

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 関数を使っ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 UDF を呼び出す。過渡的でないデータの妥当性検査は、通常は検査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をインプリメントすることです。
- 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)。

トリガーの利点

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。

トリガーの概説

トリガーを作成したら、それを表と関連付けます。この表は、トリガーの*対象表* と呼ばれます。*更新操作* 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。更新操作は、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 つのタイプの更新操作の 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更新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します。

- トリガーの名前
- 対象表の名前
- トリガー活動化時間 (更新操作実行の BEFORE または AFTER)
- トリガー・イベント (INSERT、DELETE、または UPDATE)
- 以前の値の変位変数 (存在する場合)
- 新しい値の変位変数 (存在する場合)
- 以前の値の変換表 (存在する場合)
- 新しい値の変換表 (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。
- トリガー作成タイム・スタンプ
- 現行関数パス

CREATE TRIGGER ステートメントの詳細については、 *SQL 解説書* を参照してください。

トリガー・イベント

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 MODE DB2SQL
  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 つだけの更新操作に関連付けることができます。以下のような更新操作があります。

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 を使わ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 MODE DB2SQL
  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
```

影響される行のセット

トリガー・イベントは、その SQL 操作により影響される対象表内の 1 セットの行を定義します。たとえば、次の UPDATE ステートメントを parts 表で実行するとします。

```
UPDATE PARTS
  SET ON_HAND = ON_HAND + 100
  WHERE  PART_NO > 15000
```

関連したトリガーに影響された 1 セットの行には、 *part_no* が 15 000 を超える parts 表の行がすべて含まれます。

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。

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 を使っ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 MODE DB2SQL
  UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

FOR EACH STATEMENT の細分性を使っ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 MODE DB2SQL
  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 (『トリガー活動化時間』で説明)。

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 (前) か AFTER (後) 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 MODE DB2SQL
  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。 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・マネージャーが検査した後で起きます。 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。

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、これを使っ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 トリガーを活動化した後で保全性に関する制約を検査すると、トリガー・イベントがその制約に違反する可能性があります。

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論理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。 AFTER トリガーは、トリガー SQL 操作により違反されることのある保全性に関する制約が検査された後で実行されることに注意してください。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更新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ると、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。

BEFORE および AFTER 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にはさまざまな SQL 操作が使用されます。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。 BEFORE および AFTER トリガーで指定できる一連の SQL 操作については、 507ページの『トリガー・アクション』で説明します。同様に、BEFORE および AFTER トリガーには、さまざま

な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と保証できないためです。

変位変数

FOR EACH ROW トリガーを実行する際に、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 つの変位変数を使って参照します。変位変数には、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 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 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 MODE DB2SQL
  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
```

上で説明した変位変数の OLD および NEW の定義により、すべての変位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。変位変数は、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 の両方の変位変数を参照できます。

INSERT

INSERT トリガーは、NEW 変位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 変位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: 変位変数は FOR EACH ROW トリガーに対してのみ指定できます。 FOR EACH STATEMENT トリガーでは変位変数を参照しても、影響される行のうち変位変数が参照中の行を指定することはできません。

変換表

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 つの変換表を使って参照します。変換表には、変位変数のように 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 種類の変換表のセマンティクスは次のとおりです。

OLD_TABLE 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 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 MODE DB2SQL
  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。

変換表は読み取り専用です。変換表は、トリガー・イベントに対して定義できる変位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 の両方の変換表を参照できます。

INSERT

INSERT トリガーは、NEW_TABLE 変換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 変換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: 変換表が 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* の効力範囲はトリガー本体です。 この効力範囲の*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に優先します。 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変換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。

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 MODE DB2SQL
 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
```

トリガー・アクション条件

507ページの『トリガー・アクション』で説明したように、*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。

トリガー SQL ステートメント

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。 前に説明したように、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。

トリガー SQL ステートメントのリストや、 BEFORE および AFTER トリガーの詳細については、 *SQL 解説書* を参照してください。

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。

SQL トリガー・ステートメント内の関数

ユーザー定義関数 (UDF) を含む関数は、トリガー SQL ステートメント内で呼び出されます。次の例を考えます。

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW AS N_ROW
  FOR EACH ROW MODE DB2SQL
  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
```

トリガー SQL ステートメントが非修飾の関数名を持つ関数呼び出しを含む場合、その関数呼び出しはトリガー作成時の関数パスに基づいて変換されます。関数の変換に関する詳細については、 *SQL 解説書* を参照してください。

| UDF は、SQL、Java、C または C++ プログラミング言語で定義されます。 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。 (UDF については、409ページの『第15章 ユーザー定義関数 (UDF) とメソッドの作成』で説明しています。) 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 SQL 以外のタイプの操作を実行することができます。たとえばこのような 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。

また関数は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検査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。この関数の呼び出しと使用に関する詳細は、 *SQL 解説書* を参照してください。

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 を使っ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 MODE DB2SQL
  BEGIN ATOMIC
  VALUES CASE
    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')
    WHEN NEW_EMP.HIREDATE - CURRENT DATE > 600.
      THEN SEND_MOTE('persmgr',NEW_EMP.EMPNO,'late.txt')
    END;
  END
```

トリガーのカスケード

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。

参照制約との対話

上で説明したように、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。

複数トリガーの順序付け

CREATE TRIGGER ステートメントを使っ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・イベントおよびその操作により直接的または間接的に (他の参照制約により再帰的

に) 生じた参照を制約する処置により活動化された、 1 つまたは複数の AFTER 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 MODE DB2SQL
  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 MODE DB2SQL
     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って、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性化の結果を反映するデータベースで実行されると保証できます。

トリガー、制約、UDT、UDF、および LOB 間の協調

以下の項では、トリガーおよび制約を利用して UDT、UDF、および LOB を使用するアプリケーション構造をモデル化する方法について説明します。トリガーを使うと、次のことが行えます。

- そのような構造から情報を抽出し、その情報を (構造内に隠す代わりに) 表の列に明確に保持する
- そのような構造をアプリケーション領域で管理する保全の規則を定義する
- 構造内の特定の値を受けて行う必要のある重要なアクションを示す

情報の抽出

ELECTRONIC_MAIL 表の列 MESSAGE 内に、完全な電子メール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 UDF を使っ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。これが行われると、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、ELECTRONIC_MAIL 表に新しい列を追加して 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。

```
ALTER TABLE ELECTRONIC_MAIL
  ADD COLUMN SENDER    VARCHAR (200)
  ADD COLUMN RECEIVER  VARCHAR (200)
  ADD COLUMN SENT_ON   DATE
  ADD COLUMN SUBJECT   VARCHAR (200)

CREATE TRIGGER EXTRACT_INFO
  NO CASCADE BEFORE INSERT ON ELECTRONIC_MAIL
  REFERENCING NEW AS N
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    SET N.SENDER = SENDER(N.MESSAGE);
    SET N.RECEIVER = RECEIVER(N.MESSAGE);
    SET N.SENT_ON = SENDING_DATE(N.MESSAGE);
    SET N.SUBJECT = SUBJECT(N.MESSAGE);
  END
```

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。

表における操作の妨害

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 MODE DB2SQL
```

```
WHEN (SUBJECT(N.MESSAGE) = 'undelivered mail')
BEGIN ATOMIC
  SIGNAL SQLSTATE '85101' ('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検査の制約を定義する。

```
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。

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、トリガーを使っ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 MODE DB2SQL
  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
```

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 表:

```
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 MODE DB2SQL
  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
```

第5部 DB2 プログラミングに関する考慮事項

© Copyright IBM Corp. 1993, 2001

第17章 複合環境におけるプログラミング

各国語サポートに関する考慮 517
照合順序の概説 518
照合順序. 518
照合順序のソート順序: EBCDIC および ASCII の例. 521
照合順序の指定 522
コード・ページ値の導出 523
アプリケーション・プログラム中のロケールの導出. 524
DB2 によるロケールの導出方法 . . . 524
各国語サポート・アプリケーションの開発 525
SQL ステートメントのコーディング 525
リモート・ストアード・プロシージャーと UDF のコーディング. 527
混合コード・ページ環境でのパッケージ名の考慮事項 527
プリコンパイルおよびバインド . . . 528
アプリケーションの実行 528
最後の注意点 528
異なるコード・ページ間での変換 . . 529
DBCS 文字セット. 532
拡張 UNIX コード (EUC) 文字セット 533
DBCS 環境での CLI/ODBC/JDBC/SQLJ プログラムの実行 534
日本語および中国語 (繁体字) EUC および UCS-2 コード・セットに関する考慮事項 535
EUC および 2 バイトが混在するクライアントおよびデータベースに関する考慮事項. 537
中国語 (繁体字) のユーザーに関する考慮事項 538
日本語または中国語 (繁体字) EUC アプリケーションの開発 538
混合コード・セット環境における開発 540
Unicode データベースに接続されるアプリケーション 548
マルチサイト更新に関する考慮事項. . . . 549
リモート作業単位. 549
マルチサイト更新. 549
マルチサイト更新をいつ使用するか 550
マルチサイト更新アプリケーションの SQL のコーディング. 551
マルチサイト更新アプリケーションのプリコンパイル 553
マルチサイト更新アプリケーションの構成パラメーターの指定 554
マルチサイト更新に関する制限 . . . 555
ホストまたは AS/400 サーバーへのアクセス 556
マルチスレッドのデータベースのアクセス 557
マルチスレッドの使用に際しての推奨事項 558
コード・ページおよび国別/地域別コードを処理するマルチスレッド UNIX アプリケーション 559
複数のスレッドの使用に際して潜んでいる落とし穴. 559
複数のコンテキストがデッドロックするのを避ける方法. 560
並行トランザクション 561
並行トランザクションを使用する際に気を付けるべき落とし穴 562
並行トランザクションのデッドロックの回避 563
X/Open XA インターフェース・プログラミングに関する考慮事項 563
アプリケーション連係 567
ネットワークを通じた大量データの処理 . . 567

各国語サポートに関する考慮

この節では、アプリケーションで考慮すべき各国語サポート (NLS) による問題を説明します。主なトピックを以下に示します。

- 照合順序
- 異なるコード・ページ間での変換

© Copyright IBM Corp. 1993, 2001

- コード・ページ値の導出
- アプリケーション・プログラム中のロケールの導出
- 各国語サポート・アプリケーションの開発

照合順序の概説

照合順序

データベース・マネージャーは、*照合順序* によって文字データを比較します。照合順序とは、一組の文字のうちどちらを大きくまたは小さく、あるいは等しく判断するかの順序付けです。

注: FOR BIT DATA 属性および BLOB データで定義した文字ストリング・データは、2 進ソート順序に従ってソートされます。

たとえば、照合順序は特定の文字の小文字と大文字を同等に判断するために使用されます。

データベース・マネージャーにより、固有の照合順序を持つデータベースの作成が可能となります。以降の節では、データベースで使用する特定の照合順序の決定と実際の使用に役立つ情報を示します。

データベースの 1 バイト文字はそれぞれ、内部では 0～255 (16 進数表記で X'00'～X'FF') の固有の番号で表されます。 この番号を、文字の*コード・ポイント* といいます。ひとかたまりの文字に割り当てられた数字を*コード・ページ* と呼びます。 照合順序は、ソート後の順序列の中で各文字を配置したい位置とコード・ポイントとの間のマッピングです。位置の数値は、照合順序における文字の*重み* と呼ばれます。最も単純な照合順序では、コード・ポイントと重みとが同じです。これは*基本順序* と呼ばれます。

たとえば、文字 B と b のコード・ポイントがそれぞれ X'42' と X'62' であるとします。照合順序表に従い、いずれもソートの重みが X'42' (B) である場合、これらは同様に照合されます。 B のソートの重みが X'9E' で、 b のソートの重みが X'9D' の場合は、 b が B の前にソートされます。照合順序表は各文字の重みを指定します。表はコード・ページとは異なります。コード・ページは各文字のコード・ポイントを指定します。

次の例を考えてみてください。 ASCII 文字 A～Z は X'41'～X'5A' で表されます。照合順序を記述する場合、これらが (無関係な文字が間に入らずに) ソートされて連続する順序になっていれば、 X'41'、X'42'、... X'59'、X'5A' と記述できます。

マルチバイト文字の 16 進数値もまた重みとして使用されます。たとえば、2 バイト文字 A と B のコード・ポイントがそれぞれ X'8260' と X'8261' であるとします。そして、X'82'、X'60'、および X'61' の照合の重みを使用して、これらの 2 つの文字をコード・ポイントに従ってソートします。

照合順序における重みは、固有である必要はありません。たとえば、ある文字の大文字の文字と小文字に同じ重みを与えることができます。

照合順序によりすべての 256 コード・ポイントの重みを決めると、照合順序の指定が容易になります。各文字の重みは、その文字のコード・ポイントを使用して判別することができます。

すべての場合に、 DB2 はデータベース作成時に指定された照合表を使用します。コード・ポイント表に現れる順序でマルチバイト文字をソートする場合は、データベースの作成時に照合順序として IDENTITY を指定しなければなりません。

注: GRAPHIC フィールドの 2 バイト文字および Unicode 文字の場合、ソート順序は常に IDENTITY です。

文字比較: 照合順序が確定されると、2 つの文字の比較はコード・ポイント値を直接に比較する代わりに、その重みを比較することによって実行されます。

固有でない重みが使用されると、異なる文字が同じ文字として比較される場合があります。このため、ストリング比較は次の 2 段階のプロセスをとることがあります。

1. 各ストリングの文字を重みにより比較する。
2. ステップ 1 での比較の結果、同等である場合は、コード・ポイント値に基づき各ストリングの文字を比較する。

照合順序に 256 の固有の重みが含まれる場合には、最初のステップのみが実行されます。照合順序が基本順序の場合は、2 番目のステップのみが実行されます。いずれの場合もパフォーマンスが向上します。

文字比較の詳細については、 *SQL 解説書* を参照してください

大文字 / 小文字の混在した比較: 大文字小文字に無関係な文字比較を実行するために、 TRANSLATE 関数を使用して、大文字小文字の混在した列データを大文字に変換した上で、データを選択および比較することができます。ただし、この変換は比較の目的でのみ行われます。次のデータを例にとって考えてみましょう。

```
Abel
abels
ABEL
abel
ab
Ab
```

次の SELECT ステートメントを作成したとします。

```
SELECT c1 FROM T1 WHERE TRANSLATE(c1) LIKE 'AB%'
```

以下を戻します。

```
ab
Ab
abel
Abel
ABEL
abels
```

″v1″ ビューの作成時に、以下の SELECT ステートメントも指定し、大文字でビューを比較して、大文字小文字が混在した形で INSERT 表を要求します。

```
CREATE VIEW v1 AS SELECT TRANSLATE(c1) FROM T1
```

データベース・レベルで、 sqlecrea (データベースの作成 API) の一部として照合順序を設定することができます。これにより、″a″ を ″A″ より前に処理するか、″A″ を ″a″ の後に処理するか、またはどちらも同じ重みで処理するかを決定することができます。同じ重みを持たせるならば、ORDER BY 文節を使用して照合またはソートするときに、等しく処理されます。 ″A″ と ″a″ はすべてのセンスで等しいため、 ″A″ は常に ″a″ の前に処理されます。ソートするときの唯一の基準は 16 進値です。

そのため、次のように入力すると

```
SELECT c1 FROM T1 WHERE c1 LIKE 'ab%'
```

以下を戻します。

```
ab
abel
abels
```

および

```
SELECT c1 FROM T1 WHERE c1 LIKE 'A%'
```

以下を戻します。

```
Abel
Ab
ABEL
```

次のステートメントは、

```
SELECT c1 FROM T1 ORDER BY c1
```

以下を戻します。

```
ab
Ab
abel
Abel
ABEL
abels
```

したがって、sqlecrea だけでなく、スカラー関数 TRANSLATE() を使用することもできます。ただし、照合順序を指定できるのは、sqlecrea だけであることに注意してください。コマンド行プロセッサー (CLP) から照合順序を指定することはできません。 TRANSLATE() 関数については、 *SQL 解説書* を参照してください。 sqlecrea の詳細については、 *管理 API 解説書* を参照してください。

次のように UCASE 関数を使用することもできます。ただし、この場合 DB2 は選択に索引を使用するのではなく、表走査を実行することに注意してください。

```
SELECT * FROM EMP WHERE UCASE(JOB) = 'NURSE'
```

照合順序のソート順序: EBCDIC および ASCII の例

データベース内のデータをソートする順序は、そのデータベースに定義した照合順序によって決まります。たとえば、データベース A は EBCDIC コード・ページのデフォルトの照合順序を使用し、データベース B は ASCII コード・ページのデフォルトの照合順序を使用すると仮定します。これら 2 つのデータベースのソート順序には、以下の図19 で示すような違いがあります。

```
SELECT.....
  ORDER BY COL2

EBCDIC-Based Sort         ASCII-Based Sort

COL2                      COL2
----                      ----
V1G                       7AB
Y2W                       V1G
7AB                       Y2W
```

図 19. EBCDIC ベースの順序におけるソート順序と ASCII ベースの順序におけるソート順序の相違例

同様に、データベースでの文字比較は、そのデータベースに定義した照合順序によって決まります。それで、データベース A が EBCDIC コード・ページのデフォルトの照合順序を使用し、データベース B は ASCII コード・ページのデフォルトの照合順序を使用する場合、その 2 つのデータベースにおける文字比較の結果は異なります。 522ページの図20 で、その違いについて示します。

```
SELECT.....
  WHERE COL2 > 'TT3'

EBCDIC-Based Results     ASCII-Based Results

COL2                     COL2
----                     ----
TW4                      TW4
X72                      X72
39G
```

図 20. EBCDIC ベースの順序における文字比較と ASCII ベースの順序における文字比較の相違例

連合データベースを作成する場合には、ユーザーの照合順序がデータ・ソースでの照合順序と一致するよう指定してください。こうすると、『後入れ先出し』 の機会が最大になるため、照会のパフォーマンスは可能な限り増大します。後入れ先出しの分析と照会パフォーマンスの関係について詳しくは、 *管理の手引き: インプリメンテーション* を参照してください。

照合順序の指定

データベースの照合順序は、データベースの作成時に指定されます。 いったんデータベースを作成してしまうと、照合順序は変更できません。

CREATE DATABASE API は、データベース記述子ブロック (SQLEDBDESC) と呼ばれるデータ構造を受け入れます。この構造内では、ユーザー独自の照合順序を定義できます。

データベースの照合順序の指定は次のように行ってください。
- 使用したい SQLEDBDESC 構造を渡すか、または
- NULL ポインターを渡す。オペレーティング・システム (現在の国別/地域別コードとコード・ページに基づく) の照合順序が使用されます。これは、SQL_CS_SYSTEM (0) と等しい SQLDBCSS を指定することと同じです。

SQLEDBDESC 構造には次のものが含まれています。

SQLDBCSS
データベース照合順序のソースを示す 4 バイトの整数です。有効な値は以下のとおりです。

SQL_CS_SYSTEM
オペレーティング・システム (現在の国別/地域別コードとコード・ページに基づく) の照合順序が使用されます。

SQL_CS_USER
照合順序は、SQLDBUDC フィールドの値によって指定されます。

SQL_CS_NONE
: 照合順序は基本順序です。ストリングは最初のバイトから、単純なコード・ポイント比較により 1 バイトごとに比較されます。

注: これらの定数は、SQLENV インクルード・ファイルに定義されています。

SQLDBUDC
: 256 バイトのフィールドです。 n 番目のバイトには、データベースのコード・ページの n 番目の文字のソートの重みが含まれています。 SQLDBCSS が SQL_CS_USER と異なる場合、このフィールドは無視されます。

照合順序のサンプル: デフォルトに使用されるワークステーションの照合順序ではなく、 EBCDIC 照合順序を使用したデータベースを作成しやすくするためのいくつかのサンプルが (インクルード・ファイルとして) 提供されています。

これらのインクルード・ファイルでの照合順序は、 SQLEDBDESC 構造の SQLDBUDC フィールドに指定することができます。これらの照合順序は、別の照合順序の構造のモデルとしても使うことができます。

照合順序を含むインクルード・ファイルについての詳しい説明は、次の節を参照してください。

- C/C++ の場合、611ページの『C および C++ のインクルード・ファイル』
- COBOL の場合、702ページの『COBOL のインクルード・ファイル』
- FORTRAN の場合、724ページの『FORTRAN のインクルード・ファイル』

コード・ページ値の導出

アプリケーションのコード・ページ は、データベース接続時の活動環境から導出されます。 DB2CODEPAGE レジストリー変数が設定されていれば、その値がアプリケーションのコード・ページとなります。ただし、DB2 はオペレーティング・システムから適当なコード・ページ値を決定するため、必ずしもレジストリー変数 DB2CODEPAGE を設定する必要はありません。 DB2CODEPAGE レジストリー変数を誤った値に設定すると、予測できない結果が生じる場合があります。

データベースのコード・ページ は、データベースの作成時に指定される (明示的にまたはデフォルトにより) 値から取得されます。以下は、さまざまな操作環境において*活動環境* が定められる方法を定義したものです。

UNIX
: UNIX ベースの環境の場合、活動環境は言語、区域およびコード・セットに関する情報を含むロケール設定値により決定されます。

OS/2
: OS/2 の場合、1 次および 2 次のコード・ページは `CONFIG.SYS` ファイルで指定されます。 あるセッションの中でコード・ページを表示したり動的に変更するには、 `chcp` コマンドを使用できます。

Windows 32 ビット・オペレーティング・システム

Windows 32 ビット・オペレーティング・システムの場合、DB2CODEPAGE 環境変数が設定されていない場合、そのコード・ページは、レジストリー内にある ANSI コード・ページの設定値から取得されます。

コード・ページ値の環境マッピングの完全なリストは、 *管理の手引き* を参照してください。

アプリケーション・プログラム中のロケールの導出

ロケールの設定は、Windows と UNIX ベースでそれぞれ異なります。 UNIX ベースのシステムでは 2 つのロケールがあります。

- 環境ロケールは、使用したい言語、通貨記号などの指定を可能にする。
- プログラム・ロケールには、実行中のプログラムで使用されている言語、通貨記号などが含まれている。

Windows では、「コントロール パネル」の「地域の設定」で国別設定ができます。ただし、UNIX ベースのシステムのような環境ロケールはありません。

プログラムは、開始されるとデフォルトの C ロケールを取得します。 環境ロケールのコピーは取得しません。 プログラム・ロケールを ″C″ 以外のいずれかに設定すると、DB2 ユニバーサル・データベースは現行のプログラム・ロケールを使用して、アプリケーション環境のコード・ページおよびテリトリー設定を決定します。そうでない場合は、これらの値はオペレーティング・システム環境から得られます。 `setlocale()` はスレッド・セーフではないため、 `setlocale()` をアプリケーション内から出すと、プロセス全体に新しいロケールが設定されることに注意してください。

DB2 によるロケールの導出方法

UNIX ベースのシステムでは、DB2 で使用される活動ロケールはロケールの LC_CTYPE 部分によって決定されます。詳細については、ご使用のオペレーティング・システムに対応する NLS の資料を参照してください。

- プログラム・ロケールの LC_CTYPE の値が ″C″ と異なる場合、 DB2 はその値を対応するコード・ページにマッピングすることによって、アプリケーションのコード・ページを決定します。
- LC_CTYPE が ″C″ と同じ値の場合 (″C″ ロケール)、DB2 は、環境ロケールに従い、 `setlocale()` 関数を使用してプログラム・ロケールを設定します。
- LC_CTYPE が ″C″ の値のままである場合、 DB2 はデフォルトの設定として米国英語環境とコード・ページ 819 (ISO 8859-1) を想定します。
- LC_CTYPE がすでに ″C″ ではない場合、その新しい値が対応するコード・ページのマッピングに使用されます。特定のプラットフォームのデフォルト・ロケールについい

ては、 *管理の手引き* を参照してください。特定のプラットフォームにおけるアプリケーションの作成についての追加情報は、 *アプリケーション構築の手引き* を参照してください。

各国語サポート・アプリケーションの開発

静的 SQL ステートメント中の定数文字ストリングは、バインド時にアプリケーションのコード・ページからデータベースのコード・ページに変換され、このデータベースのコード・ページの表現形式で実行時に使用されます。 このような変換が適切でない場合にそれを避けるには、ストリング定数の代わりにホスト変数を使用できます。

プログラムに固定文字ストリングが含まれる場合、同じコード・ページを使用して、アプリケーションをプリコンパイル、バインド、コンパイル、および実行することを強くお勧めします。 Unicode データベースの場合には、ストリング定数の代わりにホスト変数を使用する必要があります。 これは、バインド段階と実行段階のいずれにおいても、サーバーによるデータ変換が行われる可能性があるためです。プログラムの中で固定文字ストリングが使われている場合には、この点に注意する必要があります。これらの組み込みストリングは、バインド実行時に、そのバインド・フェーズで有効であるコード・ページに基づいて変換されます。 7 ビット ASCII 文字は DB2 ユニバーサル・データベースのサポートするすべてのコード・ページで共通なので、問題は発生しません。非 ASCII 文字については、バインドと実行において、同一の変換表が同一のコード・ページを用いて使用されているかどうか確認してください。アプリケーションが活動コード・ページを決定する方法の説明については、 523ページの『コード・ページ値の導出』を参照してください。

アプリケーションが取得する外部データは、アプリケーションのコード・ページに存在すると想定されます。これには、ファイルやユーザー入力から得られるデータも含まれます。アプリケーション以外のソースからのデータが、アプリケーションと同じコード・ページを使用していることを確認してください。

C または C++ のアプリケーションでグラフィック・データを使用するホスト変数を使用する場合は、特殊なプリコンパイラー、アプリケーション・パフォーマンス、アプリケーション設計を考慮する必要があります。これらの考慮事項の詳細については、 638ページの『C および C++ でのグラフィック・ホスト変数の処理』を参照してください。アプリケーションで EUC コード・セットを処理する場合は、 535ページの『日本語および中国語 (繁体字) EUC および UCS-2 コード・セットに関する考慮事項』を参照し、そこに示されている指針を考慮してください。

SQL ステートメントのコーディング

SQL ステートメントのコーディングは言語に依存していません。 SQL キーワードは大文字、小文字、またはそれらが混在した形で入力できますが、いずれも本書に示されているとおりに入力しなければなりません。 SQL ステートメントにおけるデータベース・オブジェクト名、ホスト変数、およびプログラム・ラベルは、ご使用のアプリケー

ション・コード・ページでサポートされる文字でなければなりません。外字セットの詳細については、 *SQL 解説書* を参照してください。

サーバーはファイル名を変換しません。 ファイル名をコーディングするには、ASCII 不変セットを使用するか、またはファイル・システムに物理的に保管される 16 進数値でパスを指定します。

マルチバイト環境では、不変文字セットに属さない特殊な文字が 4 つあります。それらは次の 4 つです。

- 2 バイトのパーセント文字と 2 バイトの下線記号文字。この 2 つは LIKE 処理で使用されます。LIKE の詳細については、 *SQL 解説書* を参照してください。
- 2 バイトの空白文字。グラフィック・ストリングに空白を埋め込むために使用します。
- 2 バイトの置換文字。ソース・コード・ページとターゲット・コード・ページとの間に対応するマッピングがない場合のコード・ページ変換時に代替の置換文字として使用されます。

以下の表は、上記 4 つの文字の各コード・ポイントをコード・ページごとに示したものです。

表 19. 特殊 2 バイト文字のコード・ポイント

コード・ページ	2 バイト パーセント記号	2 バイト 下線記号	2 バイト 空白文字	2 バイト 置換文字
932	X'8193'	X'8151'	X'8140'	X'FCFC'
938	X'8193'	X'8151'	X'8140'	X'FCFC'
942	X'8193'	X'8151'	X'8140'	X'FCFC'
943	X'8193'	X'8151'	X'8140'	X'FCFC'
948	X'8193'	X'8151'	X'8140'	X'FCFC'
949	X'A3A5'	X'A3DF'	X'A1A1'	X'AFFE'
950	X'A248'	X'A1C4'	X'A140'	X'C8FE'
954	X'A1F3'	X'A1B2'	X'A1A1'	X'F4FE'
964	X'A2E8'	X'A2A5'	X'A1A1'	X'FDFE'
970	X'A3A5'	X'A3DF'	X'A1A1'	X'AFFE'
1381	X'A3A5'	X'A3DF'	X'A1A1'	X'FEFE'
1383	X'A3A5'	X'A3DF'	X'A1A1'	X'A1A1'
13488	X'FF05'	X'FF3F'	X'3000'	X'FFFD'
1363	X'A3A5'	X'A3DF'	X'A1A1'	X'A1E0'
1386	X'A3A5'	X'A3DF'	X'A1A1'	X'FEFE'
5039	X'8193'	X'8151'	X'8140'	X'FCFC'

Unicode に関する考慮事項: Unicode データベースでは、GRAPHIC スペースが X'0020' であり、 euc-Japan および euc-Taiwan データベースに使用される GRAPHIC スペースの X'3000' とは異なっています。 X'0020' と X'3000' のどちらも Unicode 規格のスペース文字です。 GRAPHIC スペースのコード・ポイントの違いは、 EUC データベースのデータと Unicode データベースのデータを比較する際に考慮する必要があります。

Unicode データベースの詳細については、*管理の手引き: 計画* を参照してください。

リモート・ストアード・プロシージャーと UDF のコーディング

リモートに実行されるストアード・プロシージャーをコーディングする場合は、次の考慮が必要です。

- ストアード・プロシージャーのデータが、データベースのコード・ページになくてはならない。
- SQLDA を使用して、文字データ・タイプでストアード・プロシージャーとやり取りされるデータは、実際に文字データを含んでいなければならない。クライアント・アプリケーションのコード・ページがデータベースのコード・ページと異なる場合は、数値データとデータ構造を渡してはなりません。これは、サーバーが SQLDA 内のすべての文字データを変換するためです。このような変換を避けるには、データを BLOB のデータ・タイプを使用して 2 進ストリング形式で定義するか、または文字データを FOR BIT DATA として定義してデータを渡します。

デフォルト設定では、DB2 DARI ストアード・プロシージャーと UDF を呼び出すと、それらはデフォルトの各国語環境で実行します。これは、データベースの各国語環境と一致しない場合があります。その結果、C `wchar_t` グラフィック・ホスト変数や関数など、国 / 地域またはコード・ページに特有の操作を使用すると、期待どおりに作動しないことがあります。ストアード・プロシージャーまたは UDF の呼び出し時には、必ず適切な環境 (適用される場合) が初期化されるようにする必要があります。

混合コード・ページ環境でのパッケージ名の考慮事項

パッケージ名は、PRECOMPILE PROGRAM コマンドまたは API の呼び出し時に判別されます。デフォルト設定では、パッケージ名はアプリケーション・プログラムのソース・ファイルの最初の 8 バイト (ファイル拡張子は除く) に基づいて生成され、大文字で表示されます。任意に、名前を明示的に定義することができます。パッケージ名の由来とは無関係に、異なるコード・ページ環境で実行している場合、パッケージ名の文字が不変の文字セットを使用するようにしなければなりません。そうでないと、パッケージ名の修正に伴って、問題が発生することがあります。データベース・マネージャーがアプリケーションのパッケージを検出することができないか、または、クライアント側のツールがパッケージの正確な名前を表示しないということが生じます。

パッケージ名の文字のいずれかが、データベースのコード・ページの有効な文字に直接マッピングしない場合に、文字変換が理由となって、パッケージ名が修正されます。そのような場合、置換文字は変換されない文字を置き換えます。この場合の修正後には、パッケージ名がアプリケーションのコード・ページに戻されても、元のパッケージ名と一致しなくなることがあります。この振る舞いが望ましくない例としては、 DB2 データベース・ディレクターを使用してパッケージをリストしたり、その処理を行ったりする場合です。表示されるパッケージ名が、予期している名前と一致しないことがあります。

パッケージ名の変換問題を回避するには、アプリケーションとデータベースの双方のコード・ページで有効な文字だけを使用することです。

プリコンパイルおよびバインド

プリコンパイル / バインド時には、プリコンパイラーが実行中のアプリケーションとなります。プリコンパイルの要求に先立つデータベース接続の際の活動コード・ページは、プリコンパイル済みステートメント、および SQLCA 中に戻される文字データに適用されます。

アプリケーションの実行

実行時は、データベース接続時のユーザー・アプリケーションの活動コード・ページが、その接続の続いている間有効となります。すべてのデータはこのコード・ページに基づいて解釈されます。これには、動的 SQL ステートメント、ユーザー入力データ、ユーザー出力データ、および SQLCA 中の文字フィールドが含まれます。

最後の注意点

これらの指針に従わないと、予測不能な結果になる場合があります。これらの条件はデータベース・マネージャーによっては検出できないため、エラー・メッセージや警告メッセージは出されません。たとえば、C アプリケーションに、 1 つの列が C1 CHAR(20) と定義されている表 T1 に対する処理を行う次の SQL ステートメントが含まれるとします。

```
(0)  EXEC SQL CONNECT TO GLOBALDB;
(1)  EXEC SQL INSERT INTO T1 VALUES ('a-constant');
     strcpy(sqlstmt, "SELECT C1 FROM T1 WHERE C1='a-constant');
(2)  EXEC SQL PREPARE S1 FROM :sqlstmt;
```

ここで:

```
application code page at bind time = x
application code page at execution time = y
database code page = z
```

バインド実行時、ステートメント (1) の '*a-constant*' は、コード・ページ **x** からコード・ページ **z** に変換されます。この変換は、(x→z) と示すことができます。

実行時には、ステートメント (1) が実行されると、 '*a-constant*' (x→z) が表に挿入されます。 しかし、ステートメント (2) の WHERE 文節は '*a-constant*' (y→z) で実行されま

す。定数のコード・ポイントが 2 つの変換 (x→z と y→z) で値が異なるものだった場合、ステートメント (2) の SELECT は、ステートメント (1) によって挿入されたデータの取り出しに失敗することがあります。

異なるコード・ページ間での変換

最適なパフォーマンスを得るためには、アプリケーションが常にデータベースと同じコード・ページを使用するのが理想的です。ただし、これが常に実用的または可能であるとは限りません。 DB2 製品は、アプリケーションとデータベースが異なったコード・ページを使用できるようにするコード・ページ変換をサポートします。 1 つのコード・ページの文字は、データ保全性を保持するために別のコード・ページにマップされなければなりません。

文字変換はいつ行われるか: コード・ページ変換は、以下のような状況において行われます。

- データベースにアクセスしているクライアントまたはアプリケーションが、データベースのコード・ページとは異なったコード・ページで作動している場合。

  このようなデータベース変換は、アプリケーションのコード・ページからデータベースのコード・ページに変換する場合にも、データベースのコード・ページからアプリケーションのコード・ページに変換する場合にも、データベース・サーバー・マシンにて行われます。

  状況によっては、クライアント / サーバー文字変換を最小化したり除去することさえできる場合があります。

  – たとえば、コード・ページ 850 を通常使用する OS/2 および Windows クライアント・アプリケーションの環境に一致させるために、コード・ページ 850 を使用した Windows NT データベースを作成することができます。

    Windows ODBC アプリケーションを Windows データベース・クライアント内の IBM DB2 ODBC ドライバーとともに使用する場合は、 `odbc.ini` または `db2cli.ini` ファイル内で TRANSLATEDLL および TRANSLATEOPTION キーワードを使用することによってこの問題が解決される場合があります。

  – たとえば、コード・ページ 850 を通常使用する OS/2 および DOS クライアント・アプリケーションの環境に一致させるために、コード・ページ 850 を使用した DB2 (AIX 版) データベースを作成することができます。

  注: DB2 (OS/2 版) バージョン 1.0 およびバージョン 1.2 データベース・サーバーは、異なるコード・ページ間での文字変換をサポートしません。サーバーとクライアントでのコード・ページが矛盾していないかを確認してください。サポートされているコード・ページの変換については、 *管理の手引き* を参照してください。

- PC/IXF ファイルをインポートするクライアントまたはアプリケーションが、インポートされているファイルと異なるコード・ページで作動している場合。

このデータ変換は、クライアントがデータベース・サーバーにアクセスする前に、データベース・クライアント・マシンで行われます。そのアプリケーションがデータベースのコード・ページとは異なったコード・ページで作動している場合には、さらに別のデータ変換が行われます (前述のもの)。

データ変換が行われる場合は、これはインポート・ユーティリティーの呼び出し方によっても異なります。詳細については、 *管理の手引き* を参照してください。

- AS/400 サーバー上のデータへのアクセスに DB2 コネクトが用いられる場合。この場合、データの受け取り側が文字データを変換します。たとえば、DB2 (MVS/ESA 版) に送信されるデータは、DB2 (MVS/ESA 版) によって、該当する MVS コード化文字セット ID (CCSID) に変換されます。 DB2 (MVS/ESA 版) から DB2 コネクト・マシンに送り返されるデータは、 DB2 コネクトによって変換されます。詳しくは、 *DB2 コネクト 使用者の手引き* を参照してください。

以下の状況では、文字変換は行われ**ません** 。

- ファイル名。ファイル名には ASCII 不変セットを使うか、またはファイル・システム中に物理的に保管される 16 進値でファイル名を指定してください。ファイル名を SQL ステートメントの一部として組み込むと、ステートメント変換の一部としてファイル名が変換されることに注意してください。
- FOR BIT DATA 属性が割り当てられている列のデータ、またはそれをターゲットとするデータ、処理結果が FOR BIT または BLOB となる SQL 操作で使用されるデータのうちいずれか。そのいずれの場合も、データはバイト・ストリームとして扱われ、変換は生じません。[1] 異なるコード・ページ間でのストリングの割り当て、比較、および結合に関連した規則については、 *SQL 解説書* を参照してください。
- 必要なコード・ページの組み合わせに対するサポートがない、あるいはインストールされていない DB2 製品またはプラットフォーム。このような場合には、アプリケーションを実行しようとしても SQLCODE -332 (SQLSTATE 57017) が戻されます。

コード・ページ変換時の文字置換: アプリケーションがあるコードから別のコードに変換されるとき、ターゲットのコード・ページで 1 つまたは複数の文字が表示されなくなる場合もあります。このことが生じた場合、DB2 はターゲット・ストリング内の表示されない文字の位置に、*置換* 文字を挿入します。この置換文字は、その後、ストリングの有効な部分と見なされます。置換が行われた場合には、SQLCA の SQLWARN10 標識が ‘W’ に設定されます。

注: WCHARTYPE CONVERT プリコンパイル・オプションを使用した場合、置換が行われても、その文字に警告のフラグが付けられることはありません。

サポートされるコード・ページ変換: データ変換が発生する場合、**ソース・コード・ページ**から**ターゲット・コード・ページ**への変換が行われます。

1. ただし、FOR BIT DATA と定義された列に挿入されるリテラルは、変換される SQL ステートメントの一部であれば、変換されます。

ソース・コード・ページは、データのソースにより決定されます。アプリケーションのデータは、アプリケーションのコード・ページと同じソース・コード・ページを持ち、データベースからのデータはデータベースのコード・ページと同じソース・コード・ページを持ちます。

ターゲット・コード・ページの決定はより複雑で、最終的にデータが配置される場所だけでなく、中間操作の規則も考慮されます。

- データが中間の操作なしで直接アプリケーションからデータベースに移動された場合、ターゲット・コード・ページはデータベースのコード・ページとなる。
- PC/IXF ファイルからデータベースにデータがインポートされる場合、文字変換には次の 2 つの手順があります。
  1. PC/IXF ファイルのコード・ページ (ソース・コード・ページ) からアプリケーションのコード・ページ (ターゲット・コード・ページ) への変換
  2. アプリケーションのコード・ページ (ソース・コード・ページ) からデータベースのコード・ページ (ターゲット・コード・ページ) への変換

  変換のステップが 2 つ発生する可能性がある場合は注意が必要です。文字データが失われる可能性をなくすためには、必ず*管理の手引き* にリストされているサポートされる文字変換に従ってください。さらに、各グループ内で、ソースとターゲットの両方のコード・ページに存在する文字だけに、有効な変換が行われます。 他の文字は、『代入』 として使用され、ターゲット・コード・ページからソース・コード・ページに戻されるときにのみ意味を持ちます (必ずしも上記の 2 つのステップの変換プロセスによって無意味な変換が行われるとは限りません)。 アプリケーションのコード・ページがデータベースのコード・ページと同じであれば、そのような問題は防げます。
- ソースがアプリケーションのコード・ページ、データベースのコード・ページ、 FOR BIT DATA または BLOB データのいずれかである文字データに対する処理によりデータが導出される場合、データ変換は一連の規則に基づいたものとなる。最終的なターゲット・コード・ページが定められる前に、データ項目の一部またはすべてが、中間結果に変換されなければならない場合があります。これらの規則の要約、および個々の演算子と述部を持つ特定のアプリケーションについては、 *SQL 解説書* を参照してください。

DB2 ユニバーサル・データベースでサポートされているコード・ページのリストについては、 *管理の手引き* を参照してください。『グループ』 というヘッダーの下に示されている値によって、どこで変換がサポートされるかを判別できます。コード・ページは、同じ IBM 定義言語グループにリストされている任意のコード・ページにも変換できます。たとえば、コード・ページ 437 は、37、819、850、1051、1252、または 1275 に変換することができます。

注: マルチバイト・コード・ページ同士でコード・ページ変換 (たとえば、DBCS と EUC) が行われる場合、ストリングの長さが変化する可能性があります。

コード・ページ変換の拡張係数: アプリケーションが DB2 データベース・サーバーへの接続を正常に完了したならば、戻された SQLCA の次のフィールドを調べる必要があります。

- SQLERRMC フィールドの 2 番目のトークン (それぞれのトークンは X'FF' で区切られています) は、データベースのコード・ページを示します。 SQLERRMC フィールドの 9 番目のトークンは、アプリケーションのコード・ページを示します。アプリケーションのコード・ページを照会し、それをデータベースのコード・ページと比較すると、確立されている接続が文字変換の行われるタイプのものかどうかがアプリケーションに知らされます。
- SQLERRD 配列の 1 番目および 2 番目の項目。 SQLERRD(1) には、アプリケーション・コード・ページからデータベース・コード・ページに変換された場合に見込まれる、混合文字データ (CHAR データ・タイプ) の長さについての最大拡張係数または縮小係数に等しい整数値が入っています。 SQLERRD(2) には、データベース・コード・ページからアプリケーション・コード・ページへの変換が行われた場合に見込まれる、混合文字データ (CHAR データ・タイプ) の長さについての最大拡張係数または縮小係数に等しい整数値が入っています。 0 または 1 の値は、長さが変わらないことを示し、1 より大きい値は長さが長くなることを、負の値は切り捨てが生じることを示します。 CONNECT ステートメントの使用については、 *SQL 解説書* を参照してください。

グラフィック・ストリング・データについての考慮事項は、コード・ページが異なる状況には当てはまりません。 このようなストリングのおのおのの長さは、データがアプリケーションのコード・ページであるかデータベースのコード・ページであるかに関係なく、常に同じ文字数になります。

コード・ページが異なる場合の処理については、 540ページの『コード・ページが異なる場合の状況』を参照してください。

DBCS 文字セット

結合した 1 バイト文字セット (SBCS) または 2 バイト文字セット (DBCS) の各コード・ページでは、 1 バイト文字と 2 バイト文字の両方のコード・ポイントを使用できます。このことは、通常は 2 バイトのコード・ポイントの最初のバイトに対して未定義のコード・ポイントまたは割り当て済みのコード・ポイントのいずれかの剰余分と一緒に、単一バイト文字が混合しているコード表の 256 使用可能コード・ポイントのサブセットを予約することにより行います。以下の表で、これらのコード・ポイントについて示します。

表 20. 混合している文字セットのコード・ポイント

国 / 地域	サポートされている混合コード・ページ	単一バイト文字のコード・ポイント	2 バイト文字の最初のバイトのコード・ポイント
日本	932、943	X'00'-X'7F', X'A1'-X'DF'	X'81'-X'9F', X'E0'-X'FC'
日本	942	X'00'-X'80', X'A0'-X'DF', X'FD'-X'FF'	X'81'-X'9F', X'E0'-X'FC'
台湾	938 (*)	X'00'-X'7E'	X'81'-X'FC'
台湾	948 (*)	X'00'-X'80', X'FD', X'FE'	X'81'-X'FC'
韓国	949	X'00'-X'7F'	X'8F'-X'FE'
台湾	950	X'00'-X'7E'	X'81'-X'FE'
中国	1381	X'00'-X'7F'	X'8C'-X'FE'
韓国	1363	X'00'-X'7F'	X'81'-X'FE'
中国	1386	X'00'	X'81'-X'FE'
注: (*) これは古いコード・ページなので、お勧めしません。			

こうした区分のどこにも割り当てられていないコード・ポイントは定義されていません。それは、単一バイトの未定義のコード・ポイントとして処理されます。

暗黙の DBCS コード表ごとに、有効な最初のバイトの 2 番目のバイトとしてそれぞれ利用できる 256 のコード・ポイントがあります。 2 番目のバイト値は、X'40' から X'7E'、および X'80' から X'FE' までの任意の値にすることができます。 DBCS 環境では、DB2 が個々の 2 バイト文字に対して妥当性検査を行うことはありませんのでご注意ください。

拡張 UNIX コード (EUC) 文字セット

EUC コード・ページはそれぞれ、1 バイト文字のコード・ポイントと、最大 3 つの異なるマルチバイト文字のコード・ポイント・セットを両方とも使用することができます。このことは、それぞれの暗黙の 1 バイト文字 SBCS コード・ページ ID で利用可能な 256 のコード・ポイントのサブセットを予約することにより行います。コード・ポイントの剰余分は未定義であったり、マルチバイト文字のエレメントとして割り当てられたり、マルチバイト文字の単一シフト接頭部として割り当てられたりします。以下の表で、これらのコード・ポイントについて示します。

表 21. 日本語 EUC コード・ポイント

グループ	1 番目のバイト	2 番目のバイト	3 番目のバイト	4 番目のバイト
G0	X'20'-X'7E'	n/a	n/a	n/a
G1	X'A1'-X'FE'	X'A1'-X'FE'	n/a	n/a
G2	X'8E'	X'A1'-X'FE'	n/a	n/a
G3	X'8E'	X'A1'-X'FE'	X'A1'-X'FE'	n/a

表 22. 韓国語 EUC コード・ポイント

グループ	1 番目のバイト	2 番目のバイト	3 番目のバイト	4 番目のバイト
G0	X'20'-X'7E'	n/a	n/a	n/a
G1	X'A1'-X'FE'	X'A1'-X'FE'	n/a	n/a
G2	n/a	n/a	n/a	n/a
G3	n/a	n/a	n/a	n/a

表 23. 中国語 (繁体字) EUC コード・ポイント

グループ	1 番目のバイト	2 番目のバイト	3 番目のバイト	4 番目のバイト
G0	X'20'-X'7E'	n/a	n/a	n/a
G1	X'A1'-X'FE'	X'A1'-X'FE'	n/a	n/a
G2	X'8E'	X'A1'-X'FE'	X'A1'-X'FE'	X'A1'-X'FE'
G3	n/a	n/a	n/a	n/a

表 24. 中国語 (簡体字) EUC コード・ポイント

グループ	1 番目のバイト	2 番目のバイト	3 番目のバイト	4 番目のバイト
G0	X'20'-X'7E'	n/a	n/a	n/a
G1	X'A1'-X'FE'	X'A1'-X'FE'	n/a	n/a
G2	n/a	n/a	n/a	n/a
G3	n/a	n/a	n/a	n/a

こうした区分のどこにも割り当てられていないコード・ポイントは定義されていません。それは、単一バイトの未定義のコード・ポイントとして処理されます。

DBCS 環境での CLI/ODBC/JDBC/SQLJ プログラムの実行

JDBC および SQLJ プログラムは、DB2 CLI/ODBC ドライバーを使用して DB2 にアクセスするので、これらのプログラムは同一の構成ファイル (db2cli.ini) を使用します。DBCS 環境で DB2 ユニバーサル・データベースにアクセスする Java プログラムを実行する場合には、この構成ファイルに以下のような項目を追加する必要があります。

PATCH1 = 65536
: これにより、ドライバーは実際にはグラフィック・リテラルである文字リテラルの前に「G」を手動で挿入することを強制されます。 2 バイト環境で作業している際には、この PATCH1 値を必ず設定する必要があります。

PATCH1 = 64
: ドライバーは、グラフィック出力ストリングをヌルで終了するように強制されます。これは、2 バイト環境で Microsoft Access を使用する際に必要です。PATCH1 値も使用する必要があるならば、これら 2 つの値を加算して (64+65536)、PATCH1=65600 を設定します。複数の PATCH1 値を指定する方法については、下記の注 2 を参照してください。

PATCH2 = 7
: ドライバーは、すべてのグラフィック列のデータ・タイプを文字列のデータ・タイプにマップするように強制されます。これは 2 バイト環境では必須です。

PATCH2 = 10
: この設定は、EUC (拡張 UNIX コード) 環境だけで使用されます。これにより、CLI ドライバーによって、文字変数 (CHAR、VARCHAR など) に、JDBC ドライバーに適した形式でデータが提供されます。この設定をしないと、これらの文字タイプのデータは JDBC では使用できません。

注:

1. これらのキーワードは、 db2cli.ini ファイルのデータベース固有のスタンザに設定されます。複数のデータベースに対してキーワードを設定したい場合には、db2cli.ini の各データベースのスタンザに対してそれを繰り返す必要があります。
2. 複数の PATCH1 値を設定するには、個々の値を加算してから合計値を使用します。 PATCH1 を 64 と 65536 の両方に設定するには、PATCH1=65600 (64+65536) と設定します。すでに他の PATCH1 値を設定している場合には、既存の数値を、その既存の数値に新しい PATCH1 値を加えた合計によって置換してください。
3. 複数の PATCH2 値を設定するには、それらをコンマで区切ったストリング (PATCH1 オプションとは異なる) で指定します。 PATCH2 値を 1 および 7 に設定するには、PATCH2="1,7" とします。

これらのキーワードの設定方法については、*インストールおよび構成 補足* を参照してください。

日本語および中国語 (繁体字) EUC および UCS-2 コード・セットに関する考慮事項

拡張 UNIX コード (EUC) とは、UNIX 系の操作環境において 1 つないし 4 つまでの文字セットをサポートできる一連のエンコード規則のことを言います。このエンコード規則は、制御文字が文字セットの一部を分離させるために使用される、 7 ビットおよび

8 ビット・データをエンコードするための ISO 2022 定義に基づいています。しかし EUC は、コード・セットのエンコード・スキーマではなく、コード・ページの集合を指定する手段です。 EUC に基づくコード・セットは、基本的には EUC エンコード規則に従いますが、固有のインスタンスに関連付けられた固有の文字セットをも識別します。たとえば、日本語用である IBM-eucJP コード・セットは、 EUC エンコード規則に従って日本工業規格文字のエンコードも参照します。サポートされるコード・ページの一覧は、ご使用のプラットフォームの*概説およびインストール* を参照してください。

グラフィック (つまり 2 バイト文字) データに対するデータベースおよびクライアント・アプリケーション・サポートは、長さが 2 バイト以上の文字エンコードを使用する EUC コード・ページで実行される場合、制限されてしまいます。 DB2 ユニバーサル・データベース製品は、すべての文字が 2 バイト幅でなければならないという厳密な規則をグラフィック・データに適用します。このような規則のために、日本語および中国語 (繁体字) EUC コード・ページからはほとんどの文字が使用できません。このような状況の解決を目的として、日本語および中国語 (繁体字) EUC グラフィック・データを表示するためのサポートが、アプリケーション・レベルとデータベース・レベルの双方で備えられました。このサポートでは、まったく別個のエンコード・スキーマを使用します。

日本語もしくは中国語 (繁体字) EUC コード・ページで作成されたデータベースでは、グラフィック・データの保管および操作が Unicode UCS-2 コード・セットによって行われます。このコード・セットは、完全な Unicode 文字レパートリーの適格なサブセットの 1 つといえる 2 バイトのエンコード・スキーマです。これと同様に、このコード・ページで実行されるアプリケーションは、グラフィック・データを UCS-2 エンコード・データとしてデータベース・サーバーに送信します。このサポートにより、EUC コード・ページで実行されるアプリケーションでも、 DBCS コード・ページで実行されるアプリケーションであるかのように同じタイプのデータにアクセスすることが可能になります。 EUC 環境に関する追加情報は、*SQL 解説書* を参照してください。 UCS-2 に関連付けられた IBM 定義のコード・ページ ID は 1200 で、同じコード・ページの CCSID 番号は 13488 です。 eucJP または eucTW データベースのグラフィック・データは、CCSID 番号 13488 を使用します。 Unicode データベースでは、 GRAPHIC データに CCSID 1200 を使用します。

DB2 ユニバーサル・データベースは、UCS-2 を使用してエンコードできるすべての Unicode 文字をサポートします。ただし、文字の構成、分解、正規化は実行しません。 Unicode 規格の詳細は、Unicode Consortium の Web サイトである `www.unicode.org`、および Addison Wesley Longman, Inc. 発行の Unicode Standard ブックの最新版から入手できます。

これらの文字セットを使用するアプリケーションまたはデータベースを処理する場合には、 UCS-2 エンコード・データの処理を考慮する必要があります。 UCS-2 グラフィック・データをアプリケーションの EUC コード・ページに変換すると、データ長が大きくなる可能性があります。データ拡張の詳細については、 532ページの『コード・ペー

ジ変換の拡張係数』を参照してください。また、大量のデータを表示しようとする場合、バッファーを割り振ったり、 1 連のフラグメント化にあるデータを変換および表示したりする必要の生じる場合があります。

以下の節では、そのような環境内にあるデータを処理する方法について説明します。以下の節の中では、EUC という用語が、日本語および中国語 (繁体字) EUC 文字セットだけを指して用いられています。以下の節の説明は、DB2 韓国語または中国語 (簡体字) EUC サポートにはあてはまりません。これら 2 つの文字セットのグラフィック・データは EUC エンコードによって表示されるからです。

EUC および 2 バイトが混在するクライアントおよびデータベースに関する考慮事項

EUC および 2 バイトが混在しているコード・ページ環境にあるデータベース・ オブジェクトは、クライアントおよびデータベースのコード・ページ間で変換するために、オブジェクト名の長さが拡張または縮小することがあり、管理が複雑になります。特に、管理コマンドおよびユーティリティーの多くは、入力または出力パラメーターとして使用する文字ストリングの長さに限界を設けて明示しています。これらの限界は、特に明示されていなければ、一般にクライアント側にも適用されます。たとえば、表名の限界は 128 バイトです。 2 バイト・コード・ページでは 128 バイトの文字ストリングが、EUC コード・ページではもっと長く、たとえば 135 バイトとなることがあります。この 135 バイトの表名は、宛先の 2 バイト・データベースでは有効であっても、REORGANIZE TABLE などのコマンドで入力パラメーターとして使用すると、無効と見なされます。同様に、データベースのコード・ページからアプリケーションのコード・ページへ変換した後、出力パラメーターに許可される最大長を超えてしまうことがあります。これは、変換エラーまたは出力データの切り捨てのいずれかの原因となります。

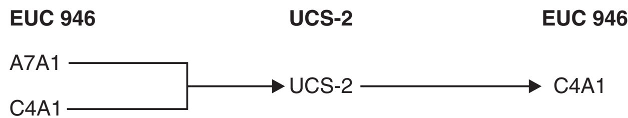
EUC と 2 バイト文字が混在している環境で管理コマンドおよびユーティリティーをよく使用する場合は、データベース・オブジェクトおよびその関連データを、サポートされる限界より長く定義する必要があります。 2 バイト・クライアントから EUC データベースを管理する場合は、 EUC クライアントから 2 バイト・データベースを管理するよりも、制限が少なくなります。通常、2 バイト文字ストリングの長さは、対応する EUC 文字ストリングの長さと同じかそれ以下です。これは一般に、文字ストリング長の限界を適用するよりも、問題が少なくてすみます。

注: SQL ステートメントの場合、入力パラメーターの妥当性検査は、ステートメント全体がデータベースのコード・ページに変換されるまで、実施されません。したがって、クライアントのコード・ページで表すときには、技術的に許可されるよりも長い文字ストリングを使用することができます。ただし、それはデータベースのコード・ページで表すときには、長さの要件に適合する必要があります。

中国語 (繁体字) のユーザーに関する考慮事項

中国語 (繁体字) の標準定義のため、2 バイトまたは EUC コード・ページと UCS-2 との間で、ある種の文字を変換する際に、副次作用が発生することがあります。変換した場合に、コード・セットの別の文字と同じ UCS-2 コード・ポイントを共用する文字が 189 文字 (187 の部首および 2 つの数字) あります。これらの文字を 2 バイトまたは EUC に戻すと、元のコード・ポイントではなく、同じ UCS-2 コード・ポイントを共用する同じ漢字のコード・ポイントに変換されます。表示された文字は同じように見えますが、実はコード・ポイントが異なっています。アプリケーションの設計によっては、この振る舞いを考慮に入れなければならないことがあります。

例として、EUC コード・ページ 964 のコード・ポイント `A7A1` を UCS-2 に変換した後、元のコード・ページ EUC 946 に戻した場合にどうなるかを考えてみましょう。



上図のように、コード・ポイント `A7A1` および `C4A1` は、変換後、コード・ポイント `C4A1` になります。

EUC コード・ページ 946 (中国語 (繁体字) EUC) または 950 (中国語 (繁体字) Big-5) と UCS-2 とのコード・ページ変換表が必要であれば、 製品およびサービス技術ライブラリーのホーム・ページを参照してください。 URL は次のとおりです。 (`http://www.ibm.com/software/data/db2/library/`)

日本語または中国語 (繁体字) EUC アプリケーションの開発

EUC アプリケーションを開発するにあたっては、次の事柄を考慮する必要があります。

- グラフィック・データの処理
- 混合コード・セット環境における開発

ストアード・プロシージャーに関する追加の考慮事項については、 539ページの『ストアード・プロシージャーに関する考慮事項』を参照してください。言語固有のアプリケーション開発に関する考慮事項については、次の箇所で説明しています。

- 643ページの『C および C++ での日本語または中国語 (繁体字) EUC、および UCS-2 に関する考慮事項』 (C および C++ の場合)
- 721ページの『COBOL での日本語または中国語 (繁体字) EUC、および UCS-2 に関する考慮事項』 (COBOL 言語の場合)
- 737ページの『FORTRAN での日本語または中国語 (繁体字) EUC、および UCS-2 に関する考慮事項』 (FORTRAN の場合)
- 756ページの『REXX の日本語または中国語 (繁体字) EUC に関する考慮事項』 (REXX の場合)

グラフィック・データの処理: この節では、グラフィック・データの処理を目的とした、EUC アプリケーション開発の考慮事項について説明します。この節では、グラフィック定数の処理について、また UDF 内のグラフィック・データ、ストアード・プロシージャー、 DBCLOB ファイルの処理について、さらには照合についても説明します。

グラフィック定数: グラフィック定数 (すなわちリテラル) は、混合文字データとして分類され、SQL ステートメントの一部ともなります。日本語または中国語 (繁体字) EUC クライアントからの SQL ステートメント内のグラフィック定数は、データベース・サーバーによってエンコードされたグラフィックに、暗黙に変換されます。 EUC エンコード文字を構成するグラフィック・リテラルであれば、 SQL アプリケーション内で使用することができます。そのようなリテラルは、EUC データベース・サーバーによってグラフィック・データベース・コード・セットに変換され、UCS-2 になります。EUC クライアントからのグラフィック定数には、CS0 7 ビット ASCII 文字や日本語 EUC CS2 (カタカナ) 文字などの、単一幅の文字を絶対に含めないでください。

グラフィック定数の追加情報については、 *SQL 解説書* を参照してください。

UDF に関する考慮事項: UDF は、データベース・サーバーで呼び出される、データベースと同じコード・セットでエンコードされたデータを処理する手段です。日本語または中国語 (繁体字) コード・セットで実行されるデータベースの場合は、そのデータベースを作成する EUC コード・セットによって、混合文字データがエンコードされます。グラフィック・データは UCS-2 によってエンコードされます。このことは、UCS-2 によってエンコードされるグラフィック・データを UDF は認識し処理する必要があることを意味します。

たとえば、グラフィック・ストリングを混合文字ストリングに変換する VARCHAR という UDF を作成したとします。この場合 VARCHAR 関数は、データベースが EUC コード・セットで作成されているのであれば、 UCS-2 としてエンコードされたグラフィック・ストリングを EUC 表記に変換しなければなりません。

ストアード・プロシージャーに関する考慮事項: 日本語または中国語 (繁体字) EUC コード・セットで実行されるストアード・プロシージャーは、 UCS-2 によってエンコードされたグラフィック・データを認識および処理できるよう準備しなければなりません。そのようなコード・セットを実行する場合、ストアード・プロシージャーの入力 / 出力 SQLDA によって送受信したグラフィック・データは、UCS-2 を使用してエンコードします。

DBCLOB ファイルに関する考慮事項: DBCLOB ファイルに関して 2 つの重要な考慮事項があります。

- DBCLOB ファイル・データは、アプリケーションの EUC コード・ページにあるものと見なされます。 EUC DBCLOB ファイルの場合、読み取り時にデータはクライアントで UCS-2 に変換され、書き込み時にはクライアントで UCS-2 から他形式に変換されます。

- サーバー上で読み取られたり書き込まれるバイト数は、ファイルから読み取ったりファイルに書き込んだりした UCS-2 エンコード文字の数に基づいて、ファイル参照変数のデータ長フィールドに戻されます。ファイルから読み取られたりファイルに書き込んだりする実際のバイト数は、その値よりも大きくなる場合があります。

照合: グラフィック・データは 2 進順序でソートされます。混合データは各バイトに適用されるデータベースの照合シーケンスでソートされます。ソート順序については、*SQL 解説書* を参照してください。同一の国 / 地域用であっても EUC コード・セットの場合と DBCS コード・セットの場合では文字を順序付ける仕方に違いがあるため、同じデータであっても EUC データベース内のデータをソートする場合と DBCS データベース内のデータをソートする場合とでは異なる結果が生じる可能性があります。

混合コード・セット環境における開発

EUC と DBCS が混在している環境でアプリケーションを開発する場合、状況によってはデータ長の増減が生じます。この節ではそのような場合の考慮事項について、次に示すトピックに分けて取り扱っています。
- コード・ページが異なる場合の状況
- クライアント・ベースのパラメーターの検証
- DESCRIBE ステートメントの使用
- 固定長または可変長データ・タイプの使用
- コード・ページ変換によるストリング長のオーバーフロー
- Unicode データベースに接続されるアプリケーション
- ストリング変換に関する規則
- 文字変換によりデータ・タイプの限界を超える場合
- ストアード・プロシージャーにおけるコード・ページ変換

コード・ページが異なる場合の状況: アプリケーションのコード・ページやデータベースのコード・ページが使用する文字エンコード・スキーマによっては、ソース・コード・ページからターゲット・コード・ページに変換される際に、ストリングのデータ長が変わる可能性があります。このようなデータ長の変化は、多くの場合、異なるエンコード・スキーマ (たとえば DBCS と EUC) を使用するマルチバイト・コード・ページ間での変換に起因します。

ほとんどの場合、データ長が短くなるよりもデータ長が長くなる方が多くのあるいは深刻な問題が生じます。それは、メモリーの割り振りに余裕がある方が不足するよりも当然勝っているからです。データ長拡張の可能性があるかどうかによっては、データの送信や取り出しに関するアプリケーションの考慮事項を別個に扱う必要が生じます。 さらに、データ長の拡張または縮小が見られそうな場合の*最善* 状況と*最悪* 状況の違いに注意することも重要です。正の値 (拡張の可能性を示す) は、*最悪* 状況を乗算係数で示します。 たとえば、SQLERRD(1) または SQLERRD(2) フィールドに 2 の値が示された場合、それは変換後のデータの処理にストレージのストリング長が変換前の最大 2 倍必要になることを意味します。これが*最悪* 標識です。この例で*最善* の状況とは、変換後もデータ長が変わらないことです。

SQLERRD(1) または SQLERRD(2) に負の値 (縮小の可能性を示す) が示される場合も、同様の*最悪* 拡張係数を表します。たとえば、値 -1 は必要となるストレージの最大値が変換前のストリング長と同じであることを示します。負の値によって、ストレージのサイズが変換前よりも小さくてよいことが示される場合もありますが、受信側のアプリケーションがソース・データの構造をはじめから識別しているのでないかぎり、負の値がそのような状況を示すために使われることはほとんどありません。

変換後に拡張が行われても大丈夫なよう十分な量のストレージを割り振るためには、値 max_target_length と同量のストレージを割り振るようにしてください。この値は次のような計算から得られます。

1. データの拡張係数を判別する。

   アプリケーションからデータベースにデータを転送する場合

   ```
   expansion_factor = ABS[SQLERRD(1)]
   if expansion_factor = 0
      expansion_factor = 1
   ```

   データベースからアプリケーションにデータを転送する場合

   ```
   expansion_factor = ABS[SQLERRD(2)]
   if expansion_factor = 0
      expansion_factor = 1
   ```

   上記の計算において、ABS は絶対値を参照しています。

   expansion_factor = 0 の検査は不可欠です。というのは、DB2 ユニバーサル・データベース製品の中には SQLERRD(1) および SQLERRD(2) に 0 を戻すものがあるからです。そのようなサーバーでは、データの拡張または縮小が生じるようなコード・ページの変換をサポートしていません。このことは拡張係数 1 によって示されます。

2. 中間長を計算する。

   ```
   temp_target_length = actual_source_length * expansion_factor
   ```

3. ターゲット・データ・タイプの最大長を判別する。

ターゲット・データ・タイプ	タイプの最大長 (type_maximum_length)
CHAR	254
VARCHAR	32 672
LONG VARCHAR	32 700
CLOB	2 147 483 647

4. ターゲットの最大長を判別する。

   ```
   1  if temp_target_length < actual_source_length
         max_target_length = type_maximum_length
      else
   ```

```
2  if temp_target_length > type_maximum_length
      max_target_length = type_maximum_length
   else
3  max_target_length = temp_target_length
```

長さの計算中に生じるかもしれないオーバーフローを許容するには、上記のチェックのすべてが必要になります。 それぞれのチェックを以下に詳しく説明します。

1 ステップ 2 の temp_target_length の計算中に数値のオーバーフローが生じました。

2 つの正の値を乗算した結果がデータ・タイプの最大値を超えてしまう場合、*行が折り返され*、2 つの値のうちの大きい方に満たない分の値が返されます。

たとえば、2 バイト符号付き整数 (CLOB 以外のデータ・タイプの長さに使用される) の最大値は 32 767 です。仮に actual_source_length が 25 000 で拡張係数が 2 であるとすると、 temp_target_length は当然のことながら 50 000 になります。 2 バイト符号付き整数にこの値では大きすぎるので、行は折り返され、値 -15 536 が返されます。

CLOB データ・タイプの場合には、4 バイト符号付き整数が長さに使用されます。 4 バイト符号付き整数の最大値は 2 147 483 647 です。

2 temp_target_length がデータ・タイプの最大値を超過しています。

データ・タイプをステップ 3 にリストされた値よりも長くすることはできません。

変換によって、そのデータ・タイプで認められている以上のスペースが必要とされる場合には、より大きなデータ・タイプを使用すれば結果を保持できるようになるかもしれません。たとえば、CHAR(250) 値が変換後のストリングの保持に 500 バイトを必要とする場合、 CHAR 値の最大長は 254 バイトなのでそのストリングを保持することはできません。しかし、VARCHAR(500) を使用すれば、そのストリングを変換後も保持できます。詳細については、547ページの『文字変換によりデータ・タイプの限界を超える場合』を参照してください。

3 temp_target_length は適切な長さになっています。

データベースへの接続時に返された SQLERRD(1) および SQLERRD(2) 値と上記の計算を基にして、文字変換後にはストリングの長さがどうなるかを判別できます。一般に、0 または 1 の値は、長さが変わらないことを示し、1 より大きい値は長さが長くなることを、負の値は切り捨てが生じることを示します。 (‘0’ の値が示されるのは下位の DB2 ユニバーサル・データベース製品からだけであることに注意してください。) さらに、これらの値は他のデータベース・サーバー製品では定義されていないことにも注意してください。 543ページの表25 には、DB2 ユニバーサル・データベースを使用した場合のさまざまなアプリケーション・コード・ページとデータベース・コード・ページの組み合わせにおける理想的な値が示されています。

表 25. CONNECT 時の SQLCA.SQLERRD 設定値

アプリケーション・コード・ページ	データベース・コード・ページ	SQLERRD(1)	SQLERRD(2)
SBCS	SBCS	+1	+1
DBCS	DBCS	+1	+1
eucJP	eucJP	+1	+1
eucJP	DBCS	-1	+2
DBCS	eucJP	+2	-1
eucTW	eucTW	+1	+1
eucTW	DBCS	-1	+2
DBCS	eucTW	+2	-1
eucKR	eucKR	+1	+1
eucKR	DBCS	+1	+1
DBCS	eucKR	+1	+1
eucCN	eucCN	+1	+1
eucCN	DBCS	+1	+1
DBCS	eucCN	+1	+1

データベース・サーバーでの拡張: SQLERRD(1) 項目にデータベース・サーバーでの拡張が示された場合、アプリケーションの側では、クライアントでは有効だった長さ依存の文字データがデータベース・サーバーにおいては変換の影響で無効になる可能性はないかについて考慮する必要が生じます。たとえば、DB2 製品では、列名の長さが 128 バイト以下でなければなりません。しかし、文字ストリングが DBCS コード・ページによって 128 バイトの長さでエンコードされていても、EUC コード・ページに変換すると、この 128 バイトの制限を超えてしまう場合があります。つまり、アプリケーション・コード・ページとデータベース・コード・ページとが同じだったときには有効だった動作でも、これらのコード・ページが異なる場合には無効になり得る、ということです。コード・ページが異なる状況下で EUC および DBCS データベースを設計する際には、これらの点に注意してください。

アプリケーションでの拡張: SQLERRD(2) 項目にクライアント・アプリケーションでの拡張が示された場合、アプリケーションの側では、長さ依存の文字データが変換後にはどの程度拡張されるかを把握する必要が生じます。たとえば、CHAR(128) 桁の行を取り出すとします。この場合、データベース・コード・ページとアプリケーション・コード・ページが同じであれば、返されるデータの長さは当然 128 バイトで問題はありません。しかし、この両者のコード・ページが異なる場合には、 DBCS コード・ページでは 128 バイトでエンコードされたデータでも EUC コード・ページに変換すると 128 バイトを超えてしまう可能性があります。したがって、このような場合には、ストリング全体を取り出せるようにするため、追加のストレージを割り振る必要がでてきます。

クライアント・ベースのパラメーターの検証: クライアントとサーバーとの間で文字データの拡張や縮小が行われることの重大な影響の 1 つとして、クライアント・アプリケーションとデータベース・サーバーとの間で受け渡しされるデータを検証する必要が生じることが挙げられます。クライアントとサーバー間でコード・ページが異なる場合、クライアントでは有効と判断されたデータが、コード・ページ変換の結果、データベース・サーバーでは無効になるという状況も十分に生じ得ます。これとは反対に、クライアントでは無効だったデータが、変換の結果、データベース・サーバーで有効になるという場合もあります。

クライアントとサーバー間でコード・ページが異なる場合には、特定のエンド・ユーザー・アプリケーションや API ライブラリーであらゆる処理が行えなくなる可能性があります。さらに、一部のパラメーター (ストリング長など) はクライアントでコマンドや API によって検証されるのに対し、 SQL ステートメント内のトークンはデータベースのコード・ページに変換されるまで検証されません。その結果、コード・ページが異なる環境でも SQL ステートメントを使用してはデータベース・オブジェクト (表など) にアクセスできるのに、特定のコマンドもしくは API ではその同じオブジェクトにアクセスできないという状況が生じ得ます。

ここであるアプリケーションの例を考えてみましょう。そのアプリケーションはエンド・ユーザーが作成した表に含まれているデータを返そうとしています。その表の名前が 128 バイトよりも長くならないことに注目してください。ではさっそく、以下に示されているこのサンプル・アプリケーションのシナリオを考慮してみましょう。

1. DBCS データベースを作成します。 DBCS クライアントからは、表 (t1) が作成されました。この表の名前の長さはちょうど 128 バイトです。この表名にはストリングが EUC に変換されると 2 バイトよりも長くなる文字がいくつか含まれています。結果として、EUC 表記の場合には、この表名の長さは 131 バイトになります。 DBCS から DBCS への間の接続では拡張がないので、データベース環境では表名が 128 バイトになり、 CREATE TABLE は正常に実行されます。
2. EUC クライアントが DBCS データベースに接続します。このクライアントからは、表 (t2) が作成されました。この表の名前の長さは、EUC としてエンコードされた場合は 120 バイトで、 DBCS に変換されると 100 バイトになります。 DBCS データベースの表名は 100 バイトです。 CREATE TABLE は正常に実行されました。
3. EUC クライアントから、表 (t3) が作成されました。この表の名前は 64 の EUC 文字 (131 バイト) で構成されています。この名前を DBCS に変換すると、この名前の長さは 128 バイトの制限に合わせて切り捨てられることになります。 CREATE TABLE は正常に実行されました。
4. EUC クライアントが DBCS データベース内の表のおのおの (t1、t2、および t3) に対してアプリケーションを呼び出したところ、結果は次のようになりました。

表	結果
t1	アプリケーションはこの表名を無効と判断します。長さが 131 バイトだからです。

t2 正しい結果が表示されます。
t3 アプリケーションはこの表名を無効と判断します。長さが 131 バイトだからです。

5. EUC クライアントによって CLP から DBCS データベースを照会します。この表名はクライアントでは 131 バイトですが、照会は成功します。この表名がサーバーでは 128 バイトになるからです。

DESCRIBE ステートメントの使用: EUC データベースに対して DESCRIBE を実行すると、データベース内の GRAPHIC 列の定義に基づいた、混合文字と GRAPHIC 列についての情報が返されてきます。この情報は、クライアントのコード・ページに変換される前のサーバーのコード・ページに基づいています。

アプリケーション・コンテキスト (例: `VALUES SUBSTR(?,1,2)`) で解決された選択リスト項目に対して DESCRIBE を実行し、その結果文字データまたはグラフィック・データが関係していた場合、返されてくる SQLLEN 値に加えて、返されてくるコード・ページも評価してください。返されてきたコード・ページがアプリケーション・コード・ページと同じ場合、拡張は行われません。返されてきたコード・ページがデータベース・コード・ページと同じ場合は、拡張の可能性があります。 FOR BIT DATA (コード・ページ 0) である選択リスト項目やアプリケーション・コード・ページ内の選択リスト項目は、アプリケーションに返されても変換されません。したがって、報告される長さに拡張や縮小はありません。

EUC アプリケーションと DBCS データベース: アプリケーションのコード・ページが EUC コード・ページの場合、 DBCS コード・ページのデータベースに対して DESCRIBE を発行すると、 CHAR および GRAPHIC 列について返された情報がデータベース・コンテキストに返されます。たとえば、 DESCRIBE の一環として返された CHAR(5) 列には、SQLLEN フィールドに 5 の値があります。 EUC 以外のデータの場合は、この列からデータを取り出すと、ストレージを 5 バイト割り振ることになります。 EUC データの場合には、このことはあてはまりません。 DBCS から EUC へのコード・ページ変換が行われると、CHAR 列の文字に使用されるエンコードに違いがあるため、データ長が大きくなる可能性があります。たとえば、中国語 (繁体字) 文字セットの場合、このようなデータ拡張が最大で 2 倍になります。つまり、DBCS エンコードでの最大文字長が 2 バイトのとき、EUC では 4 バイトになる可能性があります。日本語コード・セットの場合も、2 倍の拡張が生じ得ます。ただ中国語の場合とは異なり、日本語 DBCS で最大文字長が 2 バイトだったものが、日本語 EUC では 3 バイトになることもあることに注意してください。このような拡張は係数 1.5 によってのみ知ることができますが、日本語 DBCS では単一バイト・カタカナ文字がたったの 1 バイトなのに対し、日本語 EUC では 2 バイトと違いがあります。最大サイズの判別に関する詳細については、 532ページの『コード・ページ変換の拡張係数』を参照してください。

文字変換の結果としてのデータ長の変化は、混合文字データの場合にのみ起こるものです。グラフィック文字データのエンコードは、エンコード・スキーマがどのようなもの

であろうと、常に同じ長さ、つまり 2 バイトです。データを誤って失わないようにするため、コード・ページが異なるなどの状況が存在しているかどうか、およびそれが EUC アプリケーションと DBCS データベースとの間のものであるかどうかなどについて評価する必要があります。データベース・コード・ページとアプリケーション・コード・ページがどのようなものかは、CONNECT ステートメントから返された SQLCA 内にあるトークンから判別できます。詳細については、523ページの『コード・ページ値の導出』または *SQL 解説書* を参照してください。そのような状況が存在する場合、アプリケーションの側では、そのエンコード・スキーマの最大拡張係数に基づいて混合文字データ用の追加のストレージを割り振る必要が生じます。

DBCS アプリケーションと EUC データベース: アプリケーション・コード・ページが DBCS コード・ページで、 DESCRIBE を EUC データベースに対して発行する場合にも、545ページの『EUC アプリケーションと DBCS データベース』で説明したのと同様の事柄が生じます。ただしこちらの場合には、SQLLEN フィールドの値で示されるほどのストレージは必要とされません。この状況で最悪なケースとなるのは、すべてのデータが EUC で単一バイトまたは 2 バイトのどちらかであることです。この場合、SQLLEN に示されたのとまったく同じバイト数が DBCS エンコード・スキーマで必要になります。このような状況を除いては、SQLLEN で示されるよりも少ないバイト数で済みます。どの EUC 文字を保管するにも最大で 2 バイトあれば十分だからです。

固定長または可変長データ・タイプの使用: DBCS コード・ページと EUC コード・ページ間での変換時に生じ得るストリング長の変化を考慮に入れると、固定長データ・タイプを使わない方が良いかもしれません。ブランク埋め込みが必要になるかどうかによっては、DESCRIBE の実行後、SQLTYPEを固定長文字ストリングから可変長文字ストリングに変更する方が良いかもしれません。たとえば、EUC と DBCS 間の接続で最大拡張係数 2 が示されている場合、 (545ページの『EUC アプリケーションと DBCS データベース』の例 CHAR(5) を再び使用すると) アプリケーションは 10 バイト割り振る必要があります。

SQLTYPE が固定長の場合、EUC アプリケーションは列を DBCS データ (それ自体は末尾部分の空白として 5 バイトを含めることができる) から変換された EUC データ・ストリームとして受け取ることになります。コード・ページ変換によってデータ・エレメントがその最大サイズまで至らない場合には、これにさらにブランクが埋め込まれます。 SQLTYPE が可変長であれば、CHAR(5) 列の中身の元の意味は保ったまま、ソースの 5 バイトに 5 〜 10 バイトのターゲットを含めることができます。これと同様に、データが縮小する場合にも (DBCS アプリケーションと EUC データベース間で)、可変長データ・タイプの処理を考慮した方が良いかもしれません。

余分のスペースを割り振ったりデータ・タイプをプロモートする代わりに、データをフラグメント化させるという方法もあります。たとえば、同じ VARCHAR(3000) を選択して変換後も 6000 バイト含められるようにするため、2 つの選択手段、すなわち `SUBSTR(VC3000, 1, LENGTH(VC3000)/2)` と `SUBSTR(VC3000, (LENGTH(VC3000)/2)+1)` を別々に実行して 2 つの VARCHAR(3000) アプリケーション域に分けることができま

す。 このメソッドを使用できるのは、データ・タイプがもうこれ以上プロモートできないという場合です。たとえば、日本語 DBCS コード・ページでエンコードされ、最大長が 2 ギガバイトの CLOB では、日本語 EUC コード・ページでエンコードする際、そのサイズをおそらく最大 2 回プロモートできます。言い換えると、データをフラグメント化しなければならなくなるのは、 2 ギガバイトを超えデータ・タイプのサポートがなくなってからです。

コード・ページ変換によるストリング長のオーバーフロー: EUC と DBCS が混在するコード・ページ環境では、ストリング全体を 1 つの列に収めきれるだけのスペースを割り振っていないと、変換後に問題が生じる場合があります。この場合、ストリング長の最大拡張は 2 倍になります。拡張が列の容量を超える場合には、SQLCODE -334 (SQLSTATE 22524) が返されます。

このような結果、次のような、すぐには明らかにならなかったり前もっては考慮できない状況に陥ります。
- SQL ステートメントは 32 765 バイトよりも長くすることはできない。ステートメントがかなり複雑だったり、変換時に拡張しそうな定数やデータベース・オブジェクトを多数使用している場合には、意外に早くこの制限に到達してしまいます。
- 変換時拡張の結果として許可されている SQL ID の最大長は、短 ID で 8 バイト、長 ID で 128 バイト。
- 変換時に、ホスト言語 ID は最大長 255 バイトまで拡張できる。
- SQLCA 構造内の文字フィールドが変換される場合も、最大長として定義されている以上の拡張は許可されていない。

ストリング変換に関する規則: 混合したコード・ページ環境用のアプリケーションを設計する場合、以下の状況については、 *SQL 解説書* を参照してください。
- 集合演算 (UNION、INTERSECT、および EXCEPT) の全選択における対応するストリング列
- 連結のオペランド
- 述部のオペランド (LIKE を除く)
- CASE ステートメントの結果式
- スカラー関数 COALESCE (および VALUE) の引き数
- IN 述部の IN リストの式値
- 複数行の VALUES 文節の対応する式

上記の状況では、データベース・コード・ページではなく、アプリケーション・コード・ページに対して起こる変換を考慮しています。

文字変換によりデータ・タイプの限界を超える場合: EUC と DBCS が混在するコード・ページ環境では、混合文字やグラフィック・ストリングの長さがそのデータ・タイプで許可されている最大長を超えていると、変換後に問題が生じる場合があります。拡張後のストリング長がデータ・タイプの制限を超える場合、データ・タイプのプロモートは行われません。代わりに、許可されている拡張の最大値を超えたことを示すエラー・メッセージが返されます。この状況は、挿入時よりも、述部の評価中によく起

こります。挿入時には、アプリケーションが容易に列幅を識別でき、最大拡張係数も容易に把握できるからです。大半の場合、文字変換のこの副次作用は、最大長がもっと長い関連データ・タイプに値をキャストすることによって、回避できます。たとえば、CHAR 値の最大長は 254 バイトなのに対し、 VARCHAR の最大値は 32672 バイトです。 拡張によってデータ・タイプの最大長を超えてしまう場合には、SQLCODE -334 (SQLSTATE 22524) が返されます。

ストアード・プロシージャーにおけるコード・ページ変換: ホスト変数に指定された混合文字データやグラフィック・データ、および `sqleproc()` または SQL CALL 呼び出しでの SQLDA は、アプリケーション・コード・ページとデータベース・コード・ページが異なる場合に変換が起きます。 変換の結果としてストリング長の拡張が行われる場合、その拡張に対応できるだけのスペースが割り振られていないと、SQLCODE -334 (SQLSTATE 22524) が返されてしまいます。そのためストアード・プロシージャーを開発する際には、生じ得る拡張に備えて十分のスペースを割り振っておく必要があります。拡張に対応できるだけのスペースを割り振ることができるよう、可変長データ・タイプを使用するようにしてください。

Unicode データベースに接続されるアプリケーション

前述の節、540ページの『混合コード・セット環境における開発』に記載されている情報は、 Unicode データベースにも適用できることに注目してください。

どんなコード・ページ環境のアプリケーションでも、 Unicode データベースに接続できます。 Unicode データベースに接続するアプリケーションに対して、データベース・マネージャーは、文字ストリング・データを、アプリケーション・コード・ページとデータベース・コード・ページ (UTF-8) との間で変換します。 Unicode データベースでは、GRAPHIC データは UCS-2 ビッグ・エンディアン配列になります。しかし、コマンド行プロセッサーを使用してグラフィック・データを検索する場合には、グラフィック文字もクライアント・コード・ページに変換されます。この変換によって、コマンド行プロセッサーは、グラフィック文字を現行のフォントで表示できます。データベース・マネージャーによって UCS-2 文字がクライアント・コード・ページに変換される場合には、一部のデータが失われる可能性があります。データベース・マネージャーがクライアント・コード・ページ内の有効な文字に変換できない文字は、そのコード・ページのデフォルトの置換文字に変換されます。

DB2 がコード・ページからの文字を UTF-8 に変換すると、文字のコード・ページおよびコード・ポイントに応じて、その文字を表す合計のバイトの数は増減します。 UTF-8 では 7 ビット ASCII は不変です。各 ASCII 文字は 1 バイトを要します。非 ASCII 文字は、それぞれ 1 バイトになります。 UTF-8 変換の詳細については、*管理の手引き*、または Unicode 標準についての文書を参照してください。

Unicode データベースに接続するアプリケーションでは、グラフィック・データはすでに Unicode モードです。 DBCS データベースに接続するアプリケーションでは、グラフィック・データは、アプリケーションの DBCS コード・ページとデータベースの DBCS コード・ページとの間で変換されます。 Unicode アプリケーション自体も、

Unicode へのまたは Unicode からの必要な変換を実行する必要があります。あるいは、グラフィック・データに対しては、 WCHARTYPE CONVERT オプションを設定して `wchar_t` を使用する必要があります。 このオプションの詳細については、638ページの『C および C++ でのグラフィック・ホスト変数の処理』を参照してください。

マルチサイト更新に関する考慮事項

この節では、アプリケーションがどのようにしてリモート・データベースを作動させるか、またそれがどのようにして一度に複数のデータベースを作動させるかを説明します。主な点を次に示します。
- リモート作業単位
- マルチサイト更新

DB2 では、BACKUP、RESTORE、DROP DATABASE、CREATE DATABASE などのリモート・サーバー関数を、ローカル・アプリケーションであるかのように実行することができます。これらの関数をリモートに使用することの詳細については、 *管理の手引き* を参照してください。

リモート作業単位

作業単位とは、単一の論理トランザクションのことです。これは、すべての処理の実行が成功するか、または全体としては不成功と考えられる一連の SQL ステートメントから成り立っています。

リモート作業単位により、ユーザーまたはアプリケーション・プログラムは、作業単位あたり 1 つのロケーションでデータの読み取りまたは更新が行えます。これは、作業単位内での 1 つのデータベースへのアクセスをサポートします。アプリケーション・プログラムは複数のリモート・データベースにアクセスできますが、作業単位内では 1 つのデータベースにしかアクセスできません。

リモート作業単位には次の特徴があります。
- 作業単位あたり複数の要求がサポートされる。
- 作業単位あたり複数のカーソルがサポートされる。
- 各作業単位で 1 つのデータベースしかアクセスできない。
- アプリケーション・プログラムは、作業単位をコミットまたはロールバックします。特定のエラーの状況では、サーバーは作業単位をロールバックします。

マルチサイト更新

マルチサイト更新は、分散作業単位 (DUOW) および 2 フェーズ・コミットとしても知られています。これは、アプリケーションが、複数のリモート・データベース・サーバーにあるデータを、保全性を確実にしながら更新できるようにする関数です。マルチサイト更新のよい例として、銀行でのトランザクションを挙げることができます。このトランザクションでは、ある口座の預金が別のデータベース・サーバーの口座に移されます。このようなトランザクションでは、1 つの口座での借り方の更新のコミットが、別

の口座への貸し方の更新の処理がコミットされる時点で実行されることが重要です。マルチサイト更新の考慮事項は、これらの口座によって表されるデータが 2 つの異なるデータベース・サーバーによって管理される状況に適用されます。

マルチサイト更新を使用すれば、 1 つの作業単位内で複数の DB2 ユニバーサル・データベースのデータベースの読み取りおよび更新を行えます。 DB2 コネクトをインストールしているか、 DB2 ユニバーサル・データベース エンタープライズ・エディションで提供されている DB2 コネクト機能を使用している場合、ホストまたは AS/400 データベース・サーバー (DB2 ユニバーサル・データベース (OS/390 版) および DB2 ユニバーサル・データベース (AS/400 版) など) とともに、マルチサイト更新を使用することもできます。マルチサイト更新を他のデータベース・サーバーとともに使用する場合は、 824ページの『DB2 コネクトでのマルチサイト更新』で説明しているように特定の制限があります。

トランザクション・マネージャーは、複数のデータベース間でのコミットを調整します。 TxSeries CICS のようなトランザクション処理 (TP) モニター環境では、 TP モニターは独自のトランザクション・マネージャーを使用します。そうでない場合は、DB2 とともに供給されるトランザクション・マネージャーが使用されます。 DB2 ユニバーサル・データベース (OS/2 版)、UNIX、および Windows 32 ビット・オペレーティング・システムは、 XA (拡張アーキテクチャー) 準拠のリソース管理プログラムです。DB2 コネクトを使ってアクセスするホストおよび AS/400 データベース・サーバーは、XA 準拠のリソース管理プログラムです。また、DB2 ユニバーサル・データベース・トランザクション・マネージャーは、 XA 準拠トランザクション・マネージャーではないこと、すなわちトランザクション・マネージャーが調整できるのは DB2 データベースだけであることに注意してください。

マルチサイト更新の詳細については、*管理の手引き* を参照してください。

マルチサイト更新をいつ使用するか

マルチサイト更新は、複数のデータベースで処理を行い、データ保全を保持したい場合に最も有効です。たとえば、銀行の各支店がそれぞれのデータベースを持っていた場合、現金転送アプリケーションは、次のことを行います。

- 送金側のデータベースに接続する。
- 送金側の口座残高を読み取り、金額が十分あることを確認する。
- 送金側の口座残高から転送金額を差し引く。
- 受取側のデータベースに接続する。
- 受取側の口座残高に転送金額を加算する。
- データベースをコミットする。

1 つの作業単位でこれを行うことにより、両方のデータベースが更新されているか、またはいずれも更新されないようにできます。

マルチサイト更新アプリケーションの SQL のコーディング

表26 は、マルチサイト更新の SQL ステートメントのコーディング方法を説明しています。左の列は、マルチサイト更新を使用しない SQL ステートメントを示し、右の列は、マルチサイト更新と類似したステートメントを示しています。

表 26. RUOW およびマルチサイト更新 SQL ステートメント

RUOW ステートメント	マルチサイト更新ステートメント

CONNECT TO D1
  SELECT
  UPDATE
  COMMIT
 	
CONNECT TO D2
  INSERT
  COMMIT
 	
CONNECT TO D1
  SELECT
  COMMIT
CONNECT RESET
	CONNECT TO D1
	  SELECT
	  UPDATE
	
	CONNECT TO D2
	  INSERT
	  RELEASE CURRENT
	
	SET CONNECTION D1
	  SELECT
	  RELEASE D1
	  COMMIT

左列の SQL ステートメントは、各作業単位あたり 1 つのデータベースのみにアクセスします。これがリモート作業単位 (RUOW) アプリケーションです。

右列の SQL ステートメントは、1 つの作業単位内で複数のデータベースにアクセスします。これはマルチサイト更新アプリケーションです。

SQL ステートメントの中には、マルチサイト更新アプリケーションにおいて異なったコーディングおよび解釈をされるものもあります。

- 他のデータベースに接続する前に現行の作業単位をコミットまたはロールバックする必要はない。
- 他のデータベースへの接続時には、現行接続は切断されない。その代わりに、*休止*状態となります。 CONNECT ステートメントが失敗しても、現行接続には影響しません。
- データベースへの現行または休止接続がすでに存在している場合、USER/USING 文節では接続することはできない。
- SET CONNECTION ステートメントを使用することにより、休止接続を現行接続に変更することができる。

  休止データベースに CONNECT ステートメントを出すことによっても、同じことが行える。これは、SQLRULES を STD に設定した場合は行えません。プリコンパイラー・オプション、SET CLIENT コマンド、API のいずれかを使用することにより、SQLRULES の値を設定できます。 SQLRULES (DB2) のデフォルトにより、CONNECT ステートメントを使用する接続の切り替えが可能になります。

- 選択時に、別のデータベースに切り換えてから元のデータベースに戻しても、カーソル位置は変わらない。
- CONNECT RESET は、現行接続の切断および現行作業単位の暗黙でのコミットを行わない。その代わり、デフォルトのデータベース (すでに定義されている場合) に明示的に接続するのと同じことになります。暗黙の接続が定義されていない場合、SQLCODE -1024 (SQLSTATE 08003) が戻されます。
- RELEASE ステートメントを使用することにより、次の COMMIT で切断される接続をマークすることができる。 RELEASE CURRENT ステートメントは現行接続に適用され、 RELEASE *connection (接続名)* は指定した接続に適用され、 RELEASE ALL ステートメントはすべての接続に適用されます。

  解放とマークされた接続は、次の COMMIT で除去されるまで使用できる。ロールバックは接続の除去を行わないため、すでに存在している接続への再試行が可能です。DISCONNECT ステートメント (またはプリコンパイラー・オプション) を使用して、コミットまたはロールバック後に接続を除去します。
- COMMIT ステートメントは、作業単位 (現行または休止) におけるすべてのデータベースをコミットする。
- ROLLBACK ステートメントは、作業単位におけるすべてのデータベースをロールバックし、すべてのデータベースで保持されるカーソルを、アクセスされたかどうかにかかわらずクローズする。
- すべての接続 (休止接続および解放とマークされた接続など) は、アプリケーションのプロセスが終了すると切断する。
- 接続が成功すると (オプション指定のない CONNECT ステートメントも含む。これは、現行接続の照会のみを実行する)、数値が SQLCA の SQLERRD(3) および SQLERRD(4) フィールドに戻される。

  SQLERRD(3) フィールドは、接続されたデータベースが作業単位において現在更新可能かどうかについての情報を戻します。次に、戻される可能性のある値を示します。

  1 更新可能

  2 読み取り専用

  SQLERRD(4) フィールドにより戻される、接続の現在の特性についての情報を次に示します。

  0 適用不可能。 この状態は、更新および 1 フェーズ・コミットの使用を行う下位レベルのクライアントから実行された場合にのみ発生する。

  1 1 フェーズ・コミット。

  2 1 フェーズ・コミット (読み取り専用)。この状態は、DB2 コネクトの同期点管理プログラムを開始しないで、 DB2 コネクトを使ってアクセスするホストまたは AS/400 データベース・マネージャーにのみ適用できます。

  3 2 フェーズ・コミット

  ツールまたはユーティリティーを作成中、接続が読み取り専用の場合にユーザーにメッセージを出したい場合があるかもしれません。

マルチサイト更新アプリケーションのプリコンパイル

マルチサイト更新アプリケーションをプリコンパイルするときには、 CLP 接続をタイプ 1 接続に設定する必要があります。そのように設定しないと、アプリケーションのプリコンパイルを試行したときに、 SQLCODE 30090 (SQLSTATE 25000) を受け取ります。接続タイプの設定について詳しくは、 *コマンド解説書* を参照してください。 以下のプリコンパイラー・オプションは、マルチサイト更新を使用するアプリケーションをプリコンパイルするときに使用します。

CONNECT (1 | 2)
: このアプリケーションが 551ページの『マルチサイト更新アプリケーションの SQL のコーディング』で説明されるように、マルチサイト更新アプリケーションで SQL 構文を使用することを示すには、 CONNECT 2 を指定してください。デフォルトの設定の CONNECT 1 は、SQL 構文の通常の (RUOW) 規則がアプリケーションに適用されることを意味します。

SYNCPOINT (ONEPHASE | TWOPHASE | NONE)
: SYNCPOINT TWOPHASE および DB2 がトランザクションを調整するように指定する場合、 DB2 にはトランザクションの状態情報を管理するデータベースが必要です。アプリケーションを配置するときには、データベース・マネージャー構成パラメーター TM_DATABASE を構成して、このデータベースを定義する必要があります。 TM_DATABASE データベース・マネージャー構成パラメーターについて詳しくは、 *管理の手引き* を参照してください。これらの SYNCPOINT オプションがプログラムの処理にどのような影響を与えるかについては、 *SQL 解説書* の概念についての節を参照してください。

SQLRULES (DB2 | STD)
: ISO/ANSI SQL92 を基にした DB2 規則または標準 (STD) 規則がマルチサイト更新アプリケーションで使用されるべきかどうかを指定します。 DB2 規則では、休止データベースに CONNECT ステートメントを出すことができます。 STD 規則ではできません。

DISCONNECT (EXPLICIT | CONDITIONAL | AUTOMATIC)
: RELEASE ステートメントで解放とマークされたデータベースのみ (EXPLICIT)、オープンされている WITH HOLD カーソルがないすべてのデータベース (CONDITIONAL)、またはすべての接続 (AUTOMATIC) のいずれのデータベース接続が COMMIT 時に切断されるかを指定します。

上記のプリコンパイラー・オプションの詳細については、 *コマンド解説書* を参照してください。

マルチサイト更新プリコンパイラー・オプションは、最初のデータベース接続時に有効になります。 SET CLIENT API を使用することにより、接続がない場合に (接続が確立される前、またはすべての接続切断後)、接続の設定を置き換えることができます。 QUERY CLIENT API を使用することにより、アプリケーション・プロセスの現行接続の設定を照会することができます。

バインド・プログラムは、アプリケーション・プログラムで参照されるオブジェクトが存在しないと失敗します。マルチサイト更新アプリケーションに対処する方法は 3 つあります。

- アプリケーションを複数のファイルに分割し、それぞれが 1 つのデータベースにのみアクセスする。それぞれのファイルがアクセスする 1 つのデータベースに対して、各ファイルを用意し、バインドします。
- 各表がそれぞれのデータベースに存在することを確保できる。たとえば、銀行のそれぞれの支店のデータベースは、同じ表を持っています (データを除き)。
- 動的 SQL のみを使用できる。

マルチサイト更新アプリケーションの構成パラメーターの指定

ホストまたは AS/400 データベースに接続した XA トランザクション・マネージャーが調整するマルチサイト更新の実行について詳しくは、 *DB2 コネクト 使用者の手引き* を参照してください。

以下の構成パラメーターは、マルチサイト更新を実行するアプリケーションに影響を与えます。構成パラメーターは、LOCKTIMEOUT 以外は、データベース・マネージャー構成パラメーターです。 LOCKTIMEOUT は、データベース構成パラメーターです。

TM_DATABASE

2 フェーズ・コミット・トランザクションで、どのデータベースがトランザクション・マネージャーとして動作するか指定します。

RESYNC_INTERVAL

未確定トランザクションを再び同期させるまでのシステムの待ち時間 (秒) を指定します。 (未確定トランザクションとは、2 フェーズ・コミットの第 1 段階まで成功し、第 2 段階で失敗するようなトランザクションのことです。)

LOCKTIMEOUT

ロックの待ち時間がタイムアウトになり、指定したデータベースの現行のトランザクションをロールバックするまでの秒数を指定します。アプリケーションは、明示的に ROLLBACK を発行して、マルチサイト更新に関与しているすべてのデータベースをロールバックする必要があります。 LOCKTIMEOUT は、データベース構成パラメーターです。

TP_MON_NAME

TP モニターがある場合その名前を指定します。

SPM_RESYNC_AGENT_LIMIT

SNA を使用して AS/400 サーバーと再同期操作を実行できる、同時エージェントの数を指定します。

SPM_NAME

- SPM が TCP/IP 2PC 接続で使用される場合、 SPM_NAME はネットワーク内で一意的な ID でなければなりません。 DB2 インスタンスを作成する場合、 DB2 は SPM_NAME のデフォルトとして TCP/IP ホスト名から派生し

たものを使用します。ユーザーの環境でこの値を受け入れることができない場合は、変更することができます。ホスト・データベース・サーバーと TCP/IP 接続できるようにするため、デフォルトを受け入れ可能にする必要があります。ホストまたは AS/400 データベース・サーバーとの SNA 接続の場合、この値はご使用の SNA 製品で定義した SNA LU プロファイルと一致していなければなりません。

- SPM が SNA 2PC 接続で使用される場合、 SPM 名は 2PC に使用されている LU_NAME に設定されなければなりません。
- SPM が TCP/IP および SNA の両方に対して使用される場合、 2PC に使用されている LU_NAME を使用しなければなりません。

注: ホストまたは AS/400 データベース・サーバーを使った環境でのマルチサイト更新には、SPM が必要です。詳細については、*DB2 コネクト 使用者の手引き* を参照してください。

SPM_LOG_SIZE

現在の接続状態など接続に関する情報を記録するため、 SPM が使用する 1 次および 2 次ログ・ファイルの各ページ数 (4K バイト単位)。

上記の構成パラメーターの詳細については、 *管理の手引き* を参照してください。

マルチサイト更新に関する制限

次の制限が DB2 のマルチサイト更新に適用されます。

- TxSeries CICS のようなトランザクション・プロセス (TP) モニター環境では、DISCONNECT ステートメントはサポートされない。 DISCONNECT を TP モニターとともに使用する場合、SQLCODE -30090 (SQLSTATE 25000) を受け取ります。DISCONNECT ではなく、COMMIT が後に続く RELEASE を使用してください。
- 接続タイプ 2 環境では、動的 COMMIT と ROLLBACK はサポートされない。 この環境で COMMIT を使用すると、SQLCODE -925 (SQLSTATE 2D521) が戻され、拒否されます。この環境で ROLLBACK を使用すると SQLCODE -926 (SQLSTATE 2D521) が戻され、拒否されます。
- プリコンパイラー・オプション DISCONNECT CONDITIONAL は、バージョン 1 データベースへの接続には使用できない。バージョン 1 データベースへの接続は、保持カーソルがオープンの場合でも COMMIT で切断されます。
- WITH HOLD と宣言されたカーソルがマルチサイト更新でサポートされている場合でも、 DISCONNECT を成功させるためには、WITH HOLD と宣言されたすべてのカーソルをクローズし、 DISCONNECT 要求の前に COMMIT を出さなければならない。
- TP モニター環境のサービスがトランザクション・マネージャーで使用される際、マルチサイト更新オプションは暗黙的に CONNECT Type 2、 SYNCPOINT

TWOPHASE、SQLRULES DB2、DISCONNECT EXPLICIT となる。これらのオプションをプリコンパイルまたは SET CLIENT API と変更する必要はありませんので、これは無視されます。

- 以下の API をマルチサイト更新 (CONNECT Type 2) で使用すると、それらの API はマルチサイト更新ではサポートされないため、アプリケーションは SQLCODE -30090 (SQLSTATE 25000) を受け取ります。

```
BACKUP DATABASE
BIND
EXPORT
IMPORT
LOAD
MIGRATE DATABASE
PRECOMPILE PROGRAM
RESTART DATABASE
RESTORE DATABASE
REORGANIZE TABLE
ROLLFORWARD DATABASE
```

- ストアード・プロシージャーは、マルチサイト更新内ではサポートされる。ただし、マルチサイト更新 (CONNECT Type 2) で COMMIT および ROLLBACK ステートメントを発行するストアード・プロシージャーは、それらのステートメントがマルチサイト更新ではサポートされないため、 SQLCODE -30090 (SQLSTATE 25000) を受け取ります。

ホストまたは AS/400 サーバーへのアクセス

さまざまなデータベース・システムにアクセス (または更新) できるアプリケーションを開発する場合は、次の手順に従ってください。

1. ユーザーのアプリケーションがアクセスするすべてのデータベース・システムでサポートされているプリコンパイル / バインド・オプションおよび SQL ステートメントを使用する。たとえば、ストアード・プロシージャーは必ずしもすべてのプラットフォームでサポートされるわけではありません。

   IBM 製品の場合は、コーディングを始める**前**に、 *SQL 解説書* を参照してください。

2. 可能な時点で、アプリケーションが SQLCODE ではなく SQLSTATE を検査するようにする。

   アプリケーションが DB2 コネクトを使用する場合に SQLCODE を使用する際は、DB2 コネクトによって提供されるマッピング機能を用いて、異なるデータベース間の SQLCODE 変換をマップするようにする。

3. サポートする計画のホストまたは AS/400 データベース (DB2 ユニバーサル・データベース (OS/390 版)、OS/400、または DB2 (VSE および VM 版)) でアプリケーションをテストする。詳細については、*DB2 コネクト 使用者の手引き* を参照してください。

ホストまたは AS/400 データベース・システムにアクセスする方法の詳細については、811ページの『付録D. ホストまたは AS/400 環境でのプログラミング』を参照してください。

マルチスレッドのデータベースのアクセス

いくつかのオペレーティング・システムに共通する特徴は、1 つのプロセスで実行プログラムの複数のスレッドを実行できることです。これにより、アプリケーションが非同期のイベントを処理することができ、ポーリング機能がなくても容易にイベント駆動アプリケーションを作成できます。この節では、データベース・マネージャーが複数のスレッドを処理する方法を解説し、留意すべき設計の指針を示します。ご使用のプラットフォームがマルチスレッド化機能をサポートしているかどうかを判別するには、 *アプリケーション構築の手引き* を参照してください。

この節は、マルチスレッドのアプリケーション開発に関する用語 (クリティカル・セクションおよびセマフォーなど) に精通されている方を対象としています。これらの用語について詳しくない方は、ご使用のオペレーティング・システムのプログラミングに関する資料を調べてください。

DB2 アプリケーションは、コンテキスト を使用して複数のスレッドから SQL ステートメントを実行することができます。コンテキストとは、アプリケーションがすべての SQL ステートメントおよび API 呼び出しを実行する環境のことです。すべての接続、作業単位、および他のデータベース・リソースは、特定のコンテキストに関連付けられています。各コンテキストは、アプリケーション内の 1 つまたは複数のスレッドに関連付けられています。

各実行可能 SQL ステートメントでは、最初のランタイム・サービス呼び出しは常にラッチを取得しようとします。 成功すると処理を続行しますが、 (他のスレッドの SQL ステートメントがすでにラッチを取得しているために) 失敗すると、呼び出しは信号セマフォーでこれがポストされるまでブロックされ、それからラッチを取得し処理を続行します。ラッチは SQL ステートメントが処理を終了するまで保持され、その SQL ステートメントに対して生成された最後のランタイム・サービス呼び出しにより解放されます。

最終的な結果として、他のスレッドが SQL ステートメントを同時に実行しようとしても各 SQL ステートメントはアトミック単位で実行されます。これにより内部データ構造は、異なるスレッドによって同時に変更されることがなくなります。 API もランタイム・サービスを使用したラッチを使用します。したがって、API には、各コンテキスト内のランタイム・サービス・ルーチンと同じ制限が課されます。

デフォルト設定では、すべてのアプリケーションに、すべてのデータベース・アクセスで使用する単一のコンテキストがあります。単一スレッドのアプリケーションではこれで十分ですが、 SQL ステートメントを逐次化すると、単一コンテキストはマルチスレ

ッド・アプリケーションには不適当になります。次の DB2 API を使用すれば、アプリケーションは各スレッドに別個のコンテキストを接続して、スレッド間でコンテキストを渡すことができるようになります。

- sqleSetTypeCtx()
- sqleBeginCtx()
- sqleEndCtx()
- sqleAttachToCtx()
- sqleDetachFromCtx()
- sqleGetCurrentCtx()
- sqleInterruptCtx()

コンテキストはプロセス内のスレッド間で交換できますが、プロセス間では交換できません。複数のコンテキストの使用方法の 1 つは、並行トランザクションのサポートです。上記のコンテキストの API の使用方法の詳細は、 *管理 API 解説書* および 561ページの『並行トランザクション』を参照してください。

マルチスレッドの使用に際しての推奨事項

マルチスレッドのアプリケーションからデータベースをアクセスする際には、これらの指針に従ってください。

- **連続したデータ構造の変更。**

  アプリケーションは、SQL ステートメントまたはデータベース・マネージャー・ルーチンが、あるスレッドで処理されている間に、 SQL ステートメントおよびデータベース・マネージャー・ルーチンが使用するユーザー定義のデータ構造が、別のスレッドによって変更されていないことを確認する必要があります。 たとえば、他のスレッドの SQL ステートメントが SQLDA を使用している場合は、スレッドが SQLDA を再び割り振ることができないようにしてください。

- **個々のデータ構造の使用を考える。**

  繰り返しを避けるため、各スレッドにそれぞれのユーザー定義のデータを渡す方が容易であるといえます。これは特に、SQLCA の場合にそういえます。 SQLCA は個々の実行可能 SQL ステートメントばかりでなく、すべてのデータベース・マネージャー・ルーチンによって使用されるからです。 SQLCA に関連したこの問題を避けるための代替手段が 3 つあります。

  1. 最初のスレッド以外のスレッドが使用するルーチンにはすべて、その先頭に `struct sqlca sqlca` を追加して、EXEC SQL INCLUDE SQLCA を使用する。
  2. EXEC SQL INCLUDE SQLCA を、グローバル効力範囲内に置くのではなく、SQLを含む各ルーチンの内部に置く。
  3. EXEC SQL INCLUDE SQLCA を `#include "sqlca.h"` に置き換え、 SQL を使用するルーチンの先頭に `"struct sqlca sqlca"` を追加する。

コード・ページおよび国別/地域別コードを処理するマルチスレッド UNIX アプリケーション

| AIX、Solaris 実行環境、HP-UX、および Silicon Graphics IRIX では、データベース接続で使用されるコード・ページおよび国別 / 地域別コードを実行時に照会するために使用される関数に対して変更が加えられました。これらは現在ではスレッド・セーフですが、多数の並行データベース接続を使用するマルチスレッド・アプリケーションでは、ロック競合 (およびその結果、パフォーマンスの低下) が起きる可能性があります。

マルチスレッド・アプリケーションでのロック競合を減らすために、新しい環境変数 (DB2_FORCE_NLS_CACHE) が作成されました。 DB2_FORCE_NLS_CACHE が TRUE に設定されると、コード・ページおよび国別 / 地域別コード情報は、スレッドが最初にアクセスする際に保管されます。その時点から、キャッシュされた情報は、この情報を要求する他のすべてのスレッドで使用されます。この情報を保管するとロック競合は削減され、状況によってはパフォーマンスが向上します。

接続間のロケール設定をアプリケーションが変更する場合には、DB2_FORCE_NLS_CACHE を TRUE に設定するべきではありません。そのように設定すると、ロケール設定が変更されても、元のロケール情報が戻されます。一般的には、マルチスレッド・アプリケーションはロケール設定を変更しません。これにより、アプリケーションはスレッド・セーフのままであることができます。

複数のスレッドの使用に際して潜んでいる落とし穴

複数のスレッドを使用するアプリケーションは、当然のことながら、単一スレッドを使用するアプリケーションよりも複雑です。この余分の複雑さにより、予期しない問題がいくつか生じる可能性が潜んでいます。マルチスレッドのアプリケーションを作成するときには、次の事柄に注意を払ってください。

- **2 つ以上のコンテキスト間におけるデータベースの従属関係**

  アプリケーション内の各コンテキストには、データベース・オブジェクトに対するロックなど、それぞれ固有のデータベース・リソースがあります。このため、2 つのコンテキストが同じデータベース・オブジェクトにアクセスしている場合、デッドロックを引き起こす可能性があります。データベース・マネージャーはデッドロックを検出し、一方のコンテキストが SQLCODE -911 を受け取ると、その作業単位がロールバックされます。

- **2 つ以上のコンテキスト間でのアプリケーションの従属関係**

  コンテキスト間の従属関係を確立するプログラミング技法に注意してください。ラッチ、セマフォー、およびクリティカル・セクションは、そのような従属関係を確立するプログラミング技法の例です。アプリケーションに 2 つのコンテキストがあり、そのコンテキスト間にはアプリケーションの従属関係とデータベースの従属関係のどちらもが存在する場合、アプリケーションがデッドロックする可能性があります。従属関係のあるものがデータベース・マネージャーの管理範囲外にある場合、デッドロックが検出されないので、アプリケーションは中断またはハングします。

この種の問題の例として、2 つのコンテキストがあり、そのどちらも共通のデータ構造にアクセスするアプリケーションを考えてみましょう。両方のコンテキストがそのデータ構造を同時に変更することを避けるため、データ構造はセマフォーによって保護されます。コンテキストは次のようになります。

```
context 1
SELECT * FROM TAB1 FOR UPDATE....
UPDATE TAB1 SET....
get semaphore
access data structure
release semaphore
COMMIT

context 2
get semaphore
access data structure
SELECT * FROM TAB1...
release semaphore
COMMIT
```

最初のコンテキストが SELECT および UPDATE ステートメントを正常に実行しているときに、 2 番目のコンテキストがセマフォーを獲得してデータ構造にアクセスするとします。最初のコンテキストがセマフォーを獲得しようとしますが、2 番目のコンテキストがセマフォーを保持しているため、獲得できません。ここで 2 番目のコンテキストは表 TAB1 から行を読み取ろうとしますが、最初のコンテキストが保持するデータベース・ロックによってその操作が停止してしまいます。アプリケーションは、コンテキスト 1 がコンテキスト 2 の前に完了できず、コンテキスト 2 がコンテキスト 1 の完了を待っている状態になります。アプリケーションはデッドロックしますが、データベース・マネージャーはセマフォーの従属関係を知らないため、コンテキストはロールバックされません。そのため、アプリケーションは延期状態になってしまいます。

複数のコンテキストがデッドロックするのを避ける方法

データベース・マネージャーはスレッド間のデッドロックを検出できないため、デッドロックしないように (または少なくともそれを回避できるように) アプリケーションを設計してコーディングしなければなりません。上の例では、いくつかの方法でデッドロックを回避することができます。

- セマフォーを獲得する前に保持していたロックをすべて解除する。

  コンテキスト 1 のコードを変更して、セマフォーを獲得する前にコミットを実行するようにします。

- セマフォーによって保護されたセクションの内部に、SQL ステートメントをコーディングしない。

  コンテキスト 2 のコードを変更して、SELECT を実行する前にセマフォーを解除するようにします。

- すべての SQL ステートメントをセマフォー内部にコーディングする。

コンテキスト 1 のコードを変更して、SELECT ステートメントを実行する前にセマフォーを獲得するようにします。この技法は、動作はしますが、あまりお勧めできません。というのは、セマフォーはデータベース・マネージャーへのアクセスを逐次化するため、複数のスレッドを使用する効果が発揮されないからです。

- LOCKTIMEOUT データベース構成パラメーターを -1 以外の値に設定する。

  これではデッドロックを防ぐことはできませんが、実行を再開することはできます。コンテキスト 2 は、要求されたロックを獲得できないため、結局はロールバックされます。ロールバックのエラーを処理するときには、コンテキスト 2 はセマフォーを解除しなければなりません。セマフォーを解除すると、コンテキスト 1 が継続でき、コンテキスト 2 が解放されて作動を再試行します。

デッドロックを回避する技法を、前述の例に当てはめて示しましたが、この方法はすべてのマルチスレッド・アプリケーションに適用することができます。一般に、保護リソースを扱うようにデータベース・マネージャーを扱うならば、マルチスレッド・アプリケーションで問題が生じることはありません。

並行トランザクション

アプリケーションが、*並行トランザクション* と呼ぶ複数の独立した接続を持っていると便利である場合がしばしばあります。トランザクションを使用すると、アプリケーションは一度に複数のデータベースに接続でき、また同じデータベースへの複数の独立した接続を確立することもできます。

557ページの『マルチスレッドのデータベースのアクセス』に説明されているコンテキスト API を組み込むと、アプリケーションで並行トランザクションを使用することができます。アプリケーション内で作成されるコンテキストは、それぞれ他のコンテキストとは独立しています。ということは、他のコンテキストによって実行される COMMIT または ROLLBACK ステートメントなどの活動に影響されずに、新たにコンテキストを作成し、そのコンテキストを使用してデータベースに接続し、データベースに SQL ステートメントを実行できるということです。

たとえば、ユーザーがあるデータベースに対して SQL ステートメントを実行し、同時に別のデータベースで実行している活動のログを保存するというアプリケーションを作成するとしましょう。ログは最新のものでなければなりませんから、ログを更新するたびに COMMIT ステートメントを発行することが必要ですが、ログに対して発行したコミットがユーザーの SQL ステートメントに影響することのないようにしたいと思います。このようなときにこそ、並行トランザクションを使います。アプリケーション内で、次の 2 つのコンテキストを作成します。 1 つのコンテキストはユーザーのデータベースに接続し、ユーザーの SQL すべてに対して使用するもので、別のコンテキストはログ・データベースに接続し、ログの更新に使用します。このように設計することにより、ログ・データベースへの変更をコミットしても、ユーザーの現在の作業単位には影響は及びません。

並行トランザクションのもう 1 つの利点は、ある接続でカーソルに対する処理がロールバックされても、他の接続のカーソルには何の影響もないということです。 1 つの接続でロールバックが行われた後も、他の接続では終了した処理とカーソル位置はそのまま保持されます。

並行トランザクションを使用する際に気を付けるべき落とし穴

並行トランザクションを使用するアプリケーションでは、単一接続を使用するアプリケーションを作成する場合には起こりえない問題にいくつか直面することがあります。並行トランザクションを用いたアプリケーションを作成する場合には、以下の注意が必要です。

- 2 つ以上のコンテキスト間でのデータベースの従属関係

  アプリケーション内の各コンテキストには、データベース・オブジェクトに対するロックなど、それぞれ固有のデータベース・リソースがあります。そのため、2 つのコンテキストが同じデータベース・オブジェクトにアクセスしようとすると、デッドロック状態になってしまう可能性があります。データベース・マネージャーはデッドロックを検出し、一方のコンテキストが SQLCODE -911 を受け取り、そのコンテキストの作業単位がロールバックされます。

- 2 つ以上のコンテキスト間におけるアプリケーションの従属関係

  単一のスレッド内で複数のコンテキストを切り換えると、そのコンテキスト間に従属関係が作成されます。コンテキストにデータベースの従属関係もある場合は、デッドロックが起こる可能性があります。一部の従属関係はデータベース・マネージャーの管理範囲外にあるため、デッドロックが検出されず、アプリケーションが中断してしまうこともあります。

  この種の問題の例として、次のアプリケーションを検討してみましょう。

  ```
  context 1
  UPDATE TAB1 SET COL = :new_val

  context 2
  SELECT * FROM TAB1
  COMMIT

  context 1
  COMMIT
  ```

  最初のコンテキスト (context 1) が UPDATE ステートメントを正常に実行したとします。 UPDATE は、TAB1 のすべての行をロックします。次に、コンテキスト 2 (context 2) は TAB1 のすべての行を選択しようとします。 2 つのコンテキストは独立しているので、コンテキスト 2 はコンテキスト 1 がロックを保持する間待機します。しかし、コンテキスト 1 はコンテキスト 2 が実行を終了するまでロックを解放できません。こうして、アプリケーションはデッドロック状態になりますが、データベース・マネージャーはコンテキスト 1 がコンテキスト 2 を待機していることを知らないため、どちらか一方のコンテキストを強制的にロールバックすることはしません。そのため、アプリケーションは延期状態になってしまいます。

並行トランザクションのデッドロックの回避

データベース・マネージャーはコンテキスト間のデッドロックを検出できないため、デッドロックしないように (または少なくともデッドロックを回避できるように) アプリケーションを設計してコーディングしなければなりません。上の例では、いくつかの方法でデッドロックを回避することができます。

- コンテキストを切り換える前に保持しているロックをすべて解除する。

  コンテキスト 2 に切り換える前にコンテキスト 1 がコミットを実行するように、コードを変更します。

- 一度に 2 つ以上のコンテキストから同じオブジェクトにアクセスしない。

  同じコンテキストから更新と選択の両方が実行されるように、コードを変更します。

- LOCKTIMEOUT データベース構成パラメーターを -1 以外の値に設定する。

  これではデッドロックを防ぐことはできませんが、実行を再開することはできます。コンテキスト 2 は、要求されたロックを獲得できないため、結局はロールバックされます。コンテキスト 2 がロールバックされれば、コンテキスト 1 は実行を継続でき (これによってロックは解除され)、コンテキスト 2 は処理を再試行します。

デッドロックを回避する技法を上記の例を参考にして示しましたが、この方法は並行トランザクションを使用するすべてのアプリケーションに適用できます。

X/Open XA インターフェース・プログラミングに関する考慮事項

X/Open® XA インターフェースは、複数のリソースの変更を調整すると同時にこれらの変更の保全性を確保するさいの標準とされています。*トランザクション・プロセス・モニター* として知られるソフトウェア製品は一般に XA インターフェースを使用し、DB2 はこのインターフェースをサポートするので、そのような環境ではリソースとして 1 つ以上の DB2 データベースに同時にアクセスできます。データベース・マネージャーで提供されている XA インターフェース・サポートの概念およびそのインプリメントについては、 *管理の手引き: 計画* を参照してください。 ご使用のプラットフォームが X/Open XA インターフェースをサポートしているかどうか判別するには、 *アプリケーション構築の手引き* を参照してください。

TP モニターから独立して実行されているアプリケーションと異なるモデルがトランザクション・プロセスに使用されるため、分散トランザクション処理 (DTP) 環境で操作を行う場合、 DB2 には特別な考慮が必要となります。このトランザクション・プロセスのモデルの特性を次に示します。

1. 複数の種類のリカバリー可能なリソース (DB2 データベースなど) をトランザクション内で変更できます。
2. 実行されているトランザクションの保全性を確保するため、2 フェーズ・コミットを使用してリソースが更新されます。
3. アプリケーション・プログラムは、トランザクションのコミットまたはロールバックの要求を、リソースの管理プログラムではなく TP モニター製品に送ります。たと

えば、CICS 環境では、アプリケーションは EXEC CICS SYNCPOINT を発行してトランザクションをコミットするので、EXEC SQL COMMIT を DB2 に発行することは無効かつ不要です。

4. トランザクションの実行許可は、TP モニターと関連するソフトウェアにより事前に選別されるため、 DB2 などのリソース管理プログラムは TP モニターを 1 つの許可ユーザーとみなします。たとえば、CICS トランザクションの使用には必ず CICS による確認が必要であり、データベースへアクセスする特権は、 CICS アプリケーションを呼び出すエンド・ユーザーではなく CICS に付与されなければなりません。
5. 複数のプログラム (トランザクション) は通常待機させられ、データベース・サーバーで実行されます (DB2 では、単一で、長時間にわたり実行されるアプリケーション・プログラムとなります)。

この環境の固有の性質により、ここで実行するようにコーディングされたアプリケーションに対する DB2 の動作および要件は、特殊なものとなります。

- 分散作業単位のプリコンパイラー・オプションまたはクライアントの設定をしなくても、複数のデータベースを作業単位内で更新および接続することができる。
- DISCONNECT ステートメントは許可されないため、使用した場合は SQLCODE -30090 (SQLSTATE 25000) が戻されて拒否される。
- RELEASE ステートメントを使用することにより、トランザクションがコミットされた際にデータベース接続を解放するように指定することができる。ただし、この方法はあまりお勧めできません。 接続が解放されていた場合、その後のトランザクションにおいて許可を要求されることなくデータベースに接続するには、 SET CONNECTION ステートメントを使用してください。
- COMMIT および ROLLBACK ステートメントは、TP モニター・トランザクションがアクセスするストアード・プロシージャー内では使用できない。
- 2 フェーズ・コミットの流れがトランザクションでは明示的に無効の場合 (これらは、 XA インターフェースの専門用語で *LOCAL* と呼ばれます)、このトランザクション内でアクセスできるデータベースは 1 つだけである。このデータベースは、SNA 接続を使用してアクセスするホストまたは AS/400 データベースにすることはできません。 TCP/IP 接続を使用する DB2 (OS/390 版) バージョン 5 へのローカル・トランザクションがサポートされています。
- LOCAL トランザクションは、各トランザクションの最後に SQL COMMIT または SQL ROLLBACK を出す。出さない場合は、そのトランザクションは次に処理されるトランザクションの一部と判断されてしまいます。
- 現行データベース接続間の交換は、SQL CONNECT または SQL SET CONNECTION を使用することにより行われる。接続に使用される許可は、CONNECT ステートメントのパスワードまたはユーザー ID を指定しても変更できません。

- 表、視点、または索引などのデータベース・オブジェクトが動的 SQL ステートメントにおいて完全には修飾されない場合、ユーザー ID ではなく TP モニターが動作している場合の確認 ID で暗黙的に修飾される。
- LOCAL ではないトランザクションに対する DB2 COMMIT または ROLLBACK ステートメントは拒否される。次のコードが戻されます。
  - 静的 COMMIT では SQLCODE -925 (SQLSTATE 2D521)
  - 静的 ROLLBACK では SQLCODE -926 (SQLSTATE 2D521)
  - 動的 COMMIT では SQLCODE -426 (SQLSTATE 2D528)
  - 動的 ROLLBACK では SQLCODE -427 (SQLSTATE 2D529)
- COMMIT または ROLLBACK への CLI 要求もまた拒否される。
- データベースにより開始されたロールバックの処理

  DTP 環境において、RM がグローバル・トランザクションの自らのブランチを終了させるためにロールバック (システム・エラーまたはデッドロックなどのため) を開始した場合、トランザクション・マネージャーが同期点要求を開始するまで、同アプリケーションのプロセスからの要求をそれ以上処理させないでください。 これには、ストアード・プロシージャー中で発生したデッドロックも含まれます。データベース・マネージャーでは、CICS 環境で CICS SYNCPOINT ROLLBACK コマンドを使用するようなトランザクション・マネージャーの同期点サービスを利用して、グローバル・トランザクションをロールバックしなければならないことを通知する SQLCODE -918 (SQLSTATE 51021) を戻し、後に続く SQL 要求をすべて拒否することを意味します。何らかの理由で TM がトランザクションをコミットするように要求すると、RM は TM にそのロールバックについて通知し、TM が他の RM をロールバックするようにします。
- WITH HOLD と宣言されたカーソル

  WITH HOLD と宣言されたカーソルは、CICS トランザクション・プロセス・モニターの XA/DTP 環境でサポートされます。

  WITH HOLD と宣言されたカーソルがサポートされない場合は OPEN ステートメントが拒否され、SQLCODE -30090 (SQLSTATE 25000)、理由コード 03 が戻されます。

  トランザクションは、WITH HOLD と指定されるカーソルがもう必要なくなった場合に、明示的にクローズされるようにしなければなりません。クローズされない場合は他のトランザクションにより継承され、リソースの不必要な使用または衝突を引き起こす結果となります。
- データベースを更新または変更するステートメントは、 2 フェーズ・コミット要求の流れをサポートしないデータベースに対しては実行できない。たとえば、DRDA プロトコル (DRDA2) のレベル 2 の環境での、ホストまたは AS/400 データベース・サーバーのアクセスはサポートできない (824ページの『DB2 コネクトでのマルチサイト更新』を参照)。

- XA 環境でデータベースが更新をサポートするかどうかは、CONNECT ステートメントを出すことにより実行時に決定される。データベースが更新可能の場合、3 番目の SQLERRD トークンの値は 1 です。その他の場合の値は 2 となります。
- 更新が制限される場合に使用が認められる SQL ステートメントは、以下のものだけである。

```
CONNECT
DECLARE
DESCRIBE
EXECUTE IMMEDIATE (where the first token or keyword is SET but
                   not SET CONSTRAINTS)
OPEN CURSOR
FETCH CURSOR
CLOSE CURSOR
PREPARE (where the first token or keyword that is not blank or
         left parenthesis is SET (other than SET CONSTRAINTS),
         SELECT, WITH, or VALUES)
SELECT...INTO
VALUES...INTO
```

  他の処理はすべて SQLCODE -30090 (SQLSTATE 25000) が戻されて拒否されます。

  PREPARE ステートメントは、SELECT ステートメントの作成時にのみ使用可能です。 EXECUTE IMMEDIATE ステートメントは、 DB2 ユニバーサル・データベース (OS/390 版) からの SET SQLID ステートメントのように、出力値を戻さない SQL SET ステートメントの実行が可能です。
- API 制限:

  データベースにおいて内部的にコミットを出し、2 フェーズ・コミット・プロセスを回避する API は、SQLCODE -30090 (SQLSTATE 25000) が戻されて拒否されます。これらの API のリストについては 555ページの『マルチサイト更新に関する制限』を参照してください。これらの API はマルチサイト更新ではサポートされません (Connect Type 2)。
- アプリケーションは単一スレッドでなければならない。

  マルチスレッド・アプリケーションを開発する場合、1 つのスレッドのみが SQL を使用するか、または代わりにマルチプロセス設計を使用して、同じ作業単位内の異なるスレッドからの SQL ステートメントのインターリーブを避けるようにしてください。トランザクション・マネージャーがマルチプロセスまたはマルチスレッドをサポートする場合、 1 つのスレッドが別のスレッドの前に同期点に対して実行されるように、スレッドが連続する構成にしてください。 例としては、AIX/CICS での ***all_operation*** の **XASerialize** オプションがあります。 この情報を含む AIX/CICS XAD ファイルに関する詳細については、 *管理の手引き: 計画* を参照してください。

上記の制限は、XA インターフェースを使用した TP モニター環境で実行されるアプリケーションに適用されることに注意してください。 DB2 データベースが XA インターフェースによる使用を定義されていない場合、これらの制限は適用されませんが、 DB2

が次のトランザクションの実行に悪影響を与えることのないようにトランザクションをコーディングする必要があることはいうまでもありません。

アプリケーション連係

実行可能なアプリケーションを作成するには、アプリケーション・オブジェクトを、言語ライブラリー、オペレーティング・システム・ライブラリー、通常のデータベース・マネージャー・ライブラリー、および TP モニターとトランザクション・マネージャー製品のライブラリーとリンクする必要があります。

ネットワークを通じた大量データの処理

201ページの『第7章 ストアード・プロシージャー』で説明しているストアード・プロシージャーの技法と、 *管理の手引き: インプリメンテーション* で説明している行のブロック化を組み合わせることにより、ネットワークを通じて大量のデータを受け渡す必要のあるアプリケーションのパフォーマンスを大幅に向上させることができます。

ネットワークを通じて、配列、大量のデータ、データのパッケージを受け渡すアプリケーションは、 SQLDA データ構造または転送メカニズムとしてのホスト変数を使用して、データをブロック化して受け渡すことができます。この技法は、構造をサポートするホスト言語ではかなりの効果があります。

クライアントのアプリケーションまたはサーバー・プロシージャーのいずれもネットワークを通じてデータを受け渡すことができます。次のデータ・タイプのいずれかを使用してデータを受け渡すことができます。
- VARCHAR
- LONG VARCHAR
- CLOB
- BLOB

次のグラフィック・タイプを使用してもまた受け渡すことができます。
- VARGRAPHIC
- LONG VARGRAPHIC
- DBCLOB

このトピックの詳細については、84ページの『データ・タイプ』を参照してください。

注: この技法を使用するさいは、文字変換の機能を考慮してください。
VARCHAR、LONG VARCHAR、または CLOB などの文字ストリング・データ、または VARGRAPHIC、LONG VARGRAPHIC、または DBCLOB などのグラフィック・データ・タイプのいずれかのデータを受け渡し、アプリケーションのコード・ページがデータベースのコード・ページと異なる場合、文字データ以外のデータも文字データであるかのように変換されます。文字変換を避けるには、BLOB のデータ・タイプの変数でデータを受け渡してください。

データ変換がいつどのように発生するかの詳細については、 529ページの『異なるコード・ページ間での変換』を参照してください。

第18章 区分データベース環境におけるプログラミング上の考慮事項

パフォーマンスの向上 569
FOR READ ONLY カーソルの使用 . . . 569
指示 DSS およびローカル・バイパスの使用 569
指示 DSS 570
ローカル・バイパスの使用 570
バッファー化挿入の使用 571
バッファー化挿入の使用に関する考慮事項 574
バッファー化挿入の使用に関する制限 576
例: 大量データの抽出 (largevol.c) . . . 577
テスト環境の作成 583
エラー処理考慮事項 583
重大エラー 583
マージされた複数の SQLCA 構造 . . . 584
エラーを戻した区画を識別する 585
デバッグ 585
ループ状態または延期状態のアプリケーションの診断 585

パフォーマンスの向上

区分データベース環境におけるパフォーマンス向上の利点を利用するために、特別なプログラミング技法の使用を考慮してください。たとえば、アプリケーションが複数のデータベース・マネージャー区画から DB2 データにアクセスする場合、ここに説明する情報を考慮する必要があります。区分データベース環境についての概説は、 *管理の手引き* および *SQL 解説書* を参照してください。

FOR READ ONLY カーソルの使用

読み取り専用のカーソルを宣言する場合は、 FOR READ ONLY または FOR FETCH ONLY を、OPEN CURSOR 宣言の中に含めてください。 (FOR READ ONLY と FOR FETCH ONLY は同等のステートメントです。) FOR READ ONLY カーソルによって、コーディネーター区画が一度に複数の行を取り出すことができるようになり、その後の FETCH ステートメントのパフォーマンスが大幅に向上します。 FOR READ ONLY カーソルを明示的に宣言しない場合は、コーディネーター区画がそれらを更新可能カーソルとして扱います。更新可能カーソルは、コーディネーター区画に 1 回の FETCH につき 1 行だけの取り出しを要求するので、パフォーマンスはかなり低下します。

指示 DSS およびローカル・バイパスの使用

オンライン・トランザクション処理 (OLTP) アプリケーションを最適化するために、すべてのデータ区画の処理を要求する単純 SQL ステートメントの使用を避けたい場合があるかもしれません。その場合、アプリケーションを設計して、 SQL ステートメントがただ 1 つの区画からデータを取り出せるようにします。この技法は、コーディネーター区画が、関連する 1 つまたはすべての区画と通信する際のパフォーマンスの低下を防ぎます。

© Copyright IBM Corp. 1993, 2001

指示 DSS

分散サブセクション (DSS) は、並列照会に応じた処理を一部必要とするデータベース区画にサブセクションを送信するアクションです。また、DSS は呼び出し固有の値 (OLTP 環境にある変数の値など) を使って、サブセクションの始まりも記述します。*指示 DSS* は、表区分化キーを使用して、単一の区画を照会するように指示します。このタイプの照会をアプリケーションで使用して、すべてのノードに照会をブロードキャストした場合に発生するコーディネーター区画のオーバーヘッドを防ぎます。

指示 DSS の利点を生かした SELECT ステートメントの部分例は、以下のようになります。

```
SELECT ... FROM t1
  WHERE PARTKEY=:hostvar
```

コーディネーター区画は照会を受け取ると、 *:hostvar* に対するデータのサブセットを保持している区画を判別して、その特定の区画に照会するように指示します。

指示 DSS を使ってアプリケーションを最適化するには、複合照会を複数の単純照会に分けてください。たとえば以下の照会では、コーディネーター区画が、区分化キーを複数の値と突き合わせます。この照会を満たすデータは複数の区画にあるため、コーディネーター区画は、照会をすべての区画にブロードキャストします。

```
SELECT ... FROM t1
  WHERE PARTKEY IN (:hostvar1, :hostvar2)
```

その代わりに、照会を複数の SELECT ステートメント (各 SELECT ステートメントには 1 つのホスト変数がある) に分けます。または、 UNION を指定した 1 つの SELECT ステートメントでこれと同じ結果を得ることもできます。コーディネーター区画はさらに単純な SELECT ステートメントを利用して、指示 DSS を使って必要な区画とだけ通信します。最適化された照会は次のようになります。

```
SELECT ... AS res1 FROM t1
  WHERE PARTKEY=:hostvar1
  UNION
SELECT ... AS res2 FROM t1
  WHERE PARTKEY=:hostvar2
```

上の手法では、UNION 内 SELECT の数が区画の数よりも大幅に少ない場合に限って、パフォーマンスが向上することに注意してください。

ローカル・バイパスの使用

特別な形式の指示 DSS 照会は、コーディネーター区画にあるデータにのみアクセスします。コーディネーター区画は他の区画と通信することなく照会を完了するため、この形式は*ローカル・バイパス* と呼ばれます。

ローカル・バイパスは可能な場合には自動的に使用可能になります。しかし、トランザクションに用いるデータがある区画にトランザクションをルーティングすれば、ローカル・バイパスをより一層活用できます。これを行うための 1 つの手法として、 1 つの

リモート・クライアントに各区画との接続を維持させる方法があります。そうすると、トランザクションは、入力区分化キーに基づいた適切な接続を使えるようになります。別の手法として、トランザクションを区画ごとにグループ化して、各区画に対して別個のアプリケーション・サーバーを割り当てる方法もあります。

トランザクション・データのある区画の数を判別するには、 `sqlugrpn` API (行区画数の入手) を使用します。この API を使うと、アプリケーションは、特定の区分化キーの行の区画数を効果的に計算できます。 `sqlugrpn` API の詳細については、 *管理 API 解説書* を参照してください。

もう 1 つの手法として、 `db2atld` ユーティリティーを使って入力データを区画数に分け、各区画に対してアプリケーションのコピーを実行する方法があります。 `db2atld` ユーティリティーの詳細については、 *コマンド解説書* を参照してください。

バッファー化挿入の使用

バッファー化挿入は、表キューを利用して、挿入する行をバッファーに蓄積することによって、パフォーマンスを大幅に向上させる挿入ステートメントです。バッファー化挿入を使用するには、アプリケーションは INSERT BUF オプションを使用して準備またはバインドされている必要があります。

バッファー化挿入を行うと、挿入を実行するアプリケーションのパフォーマンスを大幅に向上させることができます。一般に、バッファー化挿入を使用できるのは、単一挿入ステートメント (他のデータベース修正ステートメントはない) をループ内で使用して多数の行を挿入し、データのソースが INSERT ステートメントの VALUES 文節にあるアプリケーションにおいてです。通常、INSERT ステートメントは 1 つ以上のホスト変数を参照し、ループを連続して実行するうちにその値を変更します。 VALUES 文節は、単一行または複数行を指定することができます。

一般的な意思決定支援アプリケーションでは、新規データのロードと定期的な挿入が必要となります。このデータは、膨大な数の行になることがあります。表をロードするときに、バッファー化挿入を使用するアプリケーションを準備しバインドすることができます。

アプリケーションがバッファー化挿入を使用するには、PREP コマンドを使用して、アプリケーション・プログラムのソース・ファイルを処理するか、または生成されたバインド・ファイルに対して BIND コマンドを使用します。いずれの場合も、INSERT BUF オプションを指定する必要があります。アプリケーションのバインドに関する詳細は、58ページの『バインド』を参照してください。アプリケーションの準備に関する詳細は、51ページの『ソース・ファイルの作成と準備』を参照してください。

注: バッファー化挿入により、以下のステップが生じます。

1. データベース・マネージャーが、表の常駐する各ノードにつき、4 KB のバッファーを 1 つオープンします。

2. アプリケーションが INSERT ステートメントを VALUES 文節を指定して発行すると、行 (1 つまたは複数) が該当するバッファー (1 つまたは複数) に置かれます。
3. データベース・マネージャーが制御をアプリケーションに戻します。
4. バッファーがいっぱいになるときにバッファー内の行が区画に送信されるか、または部分的に満たされたバッファー内に行が送信されます。部分的に満たされたバッファーは、次の事柄が発生すると、フラッシュされます。
   - アプリケーションが COMMIT (アプリケーションの終了により暗黙または明示的に) または ROLLBACK を発行したとき。
   - アプリケーションが、保管点を取る別のステートメントを発行するとき。OPEN、FETCH、および CLOSE カーソル・ステートメントは、保管点を取らせず、オープンしているバッファー化挿入のクローズもしません。

     以下の SQL ステートメントは、オープンしているバッファー化挿入をクローズします。
     - BEGIN COMPOUND SQL
     - COMMIT
     - DDL
     - DELETE
     - END COMPOUND SQL
     - EXECUTE IMMEDIATE
     - GRANT
     - 別の表への INSERT
     - バッファー化挿入を実行するのと名前が同じ動的ステートメントの PREPARE
     - REDISTRIBUTE NODEGROUP
     - RELEASE SAVEPOINT
     - REORG
     - REVOKE
     - ROLLBACK
     - ROLLBACK TO SAVEPOINT
     - RUNSTATS
     - SAVEPOINT
     - SELECT INTO
     - UPDATE
     - 他の任意のステートメントの実行、ただし、バッファー化 INSERT の繰り返しの実行 (ループ) ではない
     - アプリケーションの終了

     次の API は、オープンしているバッファー化挿入をクローズします。
     - BIND (API)
     - REBIND (API)
     - RUNSTATS (API)

- REORG (API)
- REDISTRIBUTE (API)

これらの状況のいずれにおいても、別のステートメントがバッファー化挿入をクローズすると、すべてのノードがバッファーを受け取って行が挿入されるのを、コーディネーター・ノードが待機します。すべての行が正常に挿入されたならば、バッファー化挿入をクローズするステートメントを実行します。詳細については、574ページの『バッファー化挿入の使用に関する考慮事項』を参照してください。

区分データベース環境での標準インターフェース (バッファー化挿入を使用しない) は、次のステップを行って一度に 1 行をロードします (アプリケーションが区画の 1 つでローカルに実行していることを前提とします)。

1. コーディネーター・ノードが、同じノードにあるデータベース・マネージャーに行を渡します。
2. データベース・マネージャーは、行を挿入する区画を判別するために間接的なハッシュを使用します。
   - ターゲット区画が行を受け取ります。
   - ターゲット区画が行をローカルに挿入します。
   - ターゲット区画が応答をコーディネーター・ノードに送信します。
3. コーディネーター・ノードがターゲット区画から応答を受け取ります。
4. コーディネーター・ノードがアプリケーションに応答を与えます。

   挿入は、アプリケーションが COMMIT を発行するまでコミットされません。
5. VALUES 文節を含む INSERT ステートメントは、行数や行のエレメントのタイプに関係なく、バッファー化挿入の候補となります。つまり、エレメントとして、定数、特殊レジスター、ホスト変数、式、関数などを使用できます。

指定された VALUES 文節のある INSERT ステートメントに対して、 DB2 SQL コンパイラーは、セマンティクス、パフォーマンス、または実現の考慮事項に基づいて、挿入をバッファー化することはできません。 INSERT BUF オプションを使用してアプリケーションを準備またはバインドする場合は、それがバッファー化挿入に従属していないことを確認してください。つまり、

- エラーは、バッファー化挿入に関しては非同期に、通常の挿入に関しては同期して報告することができます。非同期に報告される場合、挿入エラーはバッファー内の次の挿入に関してか、またはバッファーをクローズする*他の* ステートメントに関して報告することができます。エラーを報告するステートメントは実行されません。 たとえば、COMMIT ステートメントを使用してバッファー化挿入ループをクローズする場合を考えてみましょう。コミットは、以前の挿入との重複キーに関する SQLCODE -803 (SQLSTATE 23505) を報告します。このシナリオでは、コミットは実行されま

せん。アプリケーションに、たとえば、バッファー化挿入ループに入る前に実行される何らかの更新を実際にコミットさせたい場合、 COMMIT ステートメントを再発行する必要があります。

- 挿入される行は、バッファー化挿入を使用せずにカーソルを使用する SELECT ステートメントでは、すぐに表示できます。しかしバッファー化挿入では、行はすぐには表示されません。 INSERT BUF オプションを指定してアプリケーションをプリコンパイルまたはバインドする場合は、これらのカーソル選択行に従属するようにアプリケーションを作成しないでください。

バッファー化挿入によって、次のパフォーマンス上の利点が生じます。

- ターゲット区画が受け取るバッファーごとに 1 つのメッセージのみがターゲット区画からコーディネーター・ノードに送信されます。
- バッファーには、多数の行が含まれることがあります (特に、行が短い場合)。
- コーディネーター・ノードが新しい行を受信している間に区画間で挿入が行われている場合、並列処理が発生します。

INSERT BUF でバインドするアプリケーションは、バッファー化挿入をクローズするステートメントまたは API が発行される前に、 VALUES 文節を持つ同じ INSERT ステートメントが繰り返されるように作成すべきです。

注: バッファー化挿入がトランザクション・ログを一杯にするのを防ぐためには、定期的にコミットを実行する必要があります。

バッファー化挿入の使用に関する考慮事項

バッファー化挿入は、アプリケーション・プログラムに影響する可能性のある振る舞いを示します。この振る舞いは、バッファー化挿入の非同期特性により生じます。行の区分化キーの値に基づいて、挿入された各行は、正しい区画を指すバッファーに入れられます。これらのバッファーは、いっぱいになるか、またはフラッシュを引き起こすイベントが発生すると、宛先区画に送信されます。次の事柄に注意して、アプリケーションの設計およびコーディングの際にそれらを考慮に入れる必要があります。

- 行の挿入に関する特定のエラー条件は、INSERT ステートメントの実行時には報告されません。それらは後ほど、INSERT (または別の表への INSERT) 以外のステートメント、たとえば DELETE、UPDATE、COMMIT、または ROLLBACK などのうちの最初のものが実行されるときに報告されます。バッファー化挿入ステートメントをクローズするステートメントまたは API は、エラー報告書を参照できます。また、挿入自体の呼び出しは、それ以前の行の挿入のエラーを参照できます。さらに、バッファー化挿入エラーが別のステートメント、たとえば UPDATE や COMMIT などによって報告されると、 DB2 はそのステートメントの実行をしません。
- *行のグループ* の挿入中に検出されたエラーは、そのグループのすべての行をバックアウトさせます。行のグループとは、バッファー化挿入ステートメントの実行によって挿入されるすべての行のことです。
  - 作業単位の開始から

– ステートメントが準備されて以降 (それが動的の場合)
– 別の更新ステートメントの直前の実行以降。バッファー化挿入をクローズ (または フラッシュ) するステートメントのリストは、 571ページの『バッファー化挿入の使用』を参照してください。

- 挿入される行は、SELECT をカーソルを使用して実行する場合、同じアプリケーション・プログラムによる INSERT の後で発行される SELECT ステートメントにすぐに表示することはできません。

バッファー化 INSERT ステートメントは、オープンまたはクローズのいずれかの状態になっています。ステートメントの最初の呼び出しで、バッファー化 INSERT がオープンし、行が該当するバッファーに追加され、制御がアプリケーションに戻されます。その後の呼び出しでは、行がバッファーに追加され、ステートメントはオープンしたままにされます。ステートメントがオープンしている間、バッファーがその宛先区画に送信されることがあります。その場合、行がターゲット表の区画に挿入されます。バッファー化挿入をクローズするステートメントまたは API が、バッファー化 INSERT ステートメントのオープン中に呼び出された場合 (*別の* バッファー化 INSERT ステートメントの呼び出しも含む)、またはオープンしているバッファー化 INSERT ステートメントに対して PREPARE ステートメントが発行された場合、オープン・ステートメントは、新規の要求を処理する前に、クローズします。バッファー化 INSERT ステートメントがクローズすると、残ったバッファーはフラッシュされます。次に、行がターゲット区画に送信され、挿入されます。すべてのバッファーが送信されて、すべての行が挿入された後でのみ、新規の要求が処理を開始します。

INSERT ステートメントのクローズ中にエラーが検出されると、新規要求の SQLCA が、エラーの記述でいっぱいになり、新規要求は実行されません。また、そのオープン*以来*、バッファー化 INSERT ステートメントによって挿入された行のグループ全体が、データベースから除去されます。アプリケーションの状態は、検出された特定のエラーに定義された状態になります。たとえば、以下のとおりです。

- エラーがデッドロックの場合、トランザクションはロールバックされます (バッファー化挿入セクションのオープン前に行った変更も含む)。
- エラーが固有キー違反の場合、データベースの状態は、ステートメントのオープン前と同じです。トランザクションは活動状態のままで、ステートメントのオープン前に行った変更は影響されません。

たとえば、バッファー化挿入オプションを指定してバインドする、次のアプリケーションを例にして考えてみましょう。

```
EXEC SQL UPDATE t1 SET COMMENT='about to start inserts';
DO UNTIL EOF OR SQLCODE < 0;
  READ VALUE OF hv1 FROM A FILE;
  EXEC SQL INSERT INTO t2 VALUES (:hv1);
  IF 1000 INSERTS DONE, THEN DO
     EXEC SQL INSERT INTO t3 VALUES ('another 1000 done');
```

```
        RESET COUNTER;
      END;
    END;
    EXEC SQL COMMIT;
```

ファイルに 8 000 個の値が含まれているが、値 3 258 は正しくない (たとえば、固有キー違反) とします。 1 000 個の行を挿入すると、次の SQL ステートメントの実行を引き起こし、 INSERT INTO t2 ステートメントをクローズします。 4 回目に 1 000 個の行の挿入を実行しているときに、値 3 258 のエラーが検出されます。さらに値を挿入した後で (必ずしも次の値とは限らない)、検出されることもあります。この状況では、エラー・コードは INSERT INTO t2 ステートメントに対して戻されます。

また、表 t3 に対して挿入しようとすると、エラーが検出されることもあります。この動作は、INSERT INTO t2 ステートメントをクローズします。この状況では、エラーは表 t2 に適用されるとしても、エラー・コードは INSERT INTO t3 ステートメントに対して戻されます。

代わりに、3 900 行を挿入するとします。行番号 3 258 に関するエラーを通知される前に、アプリケーションはループを終了して、COMMIT を発行しようとします。固有キー違反戻りコードが、COMMIT ステートメントに対して発行され、 COMMIT は実行されません。アプリケーションがデータベース (すなわち遠端) にある 3000 行をコミットするには (EXEC SQL INSERT INTO t3 ... の最後の実行は、それら 3000 行の保管点を終了する)、COMMIT を再発行する必要があります。同様の考慮事項が、ROLLBACK にもあてはまります。

注: バッファー化挿入を使用する際には、戻される SQLCODES を注意深くモニターして、表が未決状態にならないようにすべきです。たとえば、上の例で THEN DO ステートメントから SQLCODE < 0 文節を除去すると、表は行数が定まらないまま終了してしまいます。

バッファー化挿入の使用に関する制限

以下の制約事項が適用されます。

- バッファー化挿入を利用するアプリケーションは、次のいずれか 1 つが真でなければなりません。
  - アプリケーションは PREP によって準備されているか、または BIND コマンドによってバインドされている必要があり、 INSERT BUF オプションを指定する。
  - アプリケーションは BIND または SQL_INSERT_BUF オプションを指定した PREP API を使用してバインドする必要がある。
- VALUES 文節のある INSERT ステートメントで、長いフィールドまたは LOBS が明示または暗黙の列リストに含まれる場合、そのステートメントでは INSERT BUF オプションが無視されて、バッファー化挿入ではなく通常の挿入操作セクションが実行されます。これは、エラー条件ではないので、エラーまたは警告メッセージは発行されません。

- 全選択を行う INSERT は INSERT BUF によって影響されません。バッファー化 INSERT は、このタイプの INSERT のパフォーマンスを改善しません。
- バッファー化挿入は、アプリケーションでのみ使用でき、CLP の発行する挿入では使用できません。後者の挿入は、EXECUTE IMMEDIATE ステートメントによって実行されるからです。

アプリケーションは、サポートされる任意のクライアント・プラットフォームから実行することができます。

例: 大量データの抽出 (largevol.c)

DB2 ユニバーサル・データベースは、並列照会処理のための優れた機能を提供しますが、アプリケーションまたは EXPORT コマンドの接続の単一点は、大量のデータを抽出する際に、障害になる可能性があります。これが生じるのは、データベース・マネージャーからアプリケーションへのデータの受け渡しが、単一ノード (通常はシングル・プロセッサー) 上で実行される CPU 集中処理であるからです。

DB2 ユニバーサル・データベースは、障害を解決するために、プロセッサーの数を増やして、抽出したデータのボリュームが時間単位に正比例するような、いくつかの手段を提供しています。次の例は、これらの手法の背後にある基本的な考えを説明しています。

EMPLOYEE という表があるとします。この表は、20 個のノードに保管されています。正規の部門に属する (すなわち、WORKDEPT はヌルではない) すべての従業員の郵送リスト (FIRSTNME (名)、 LASTNAME (姓)、JOB (担当作業)) を生成するとします。

次の照会は、各ノードで並列に実行し、その後、単一ノード (コーディネーター・ノード) で全体の応答セットを生成します。

```
SELECT FIRSTNME, LASTNAME, JOB FROM EMPLOYEE WHERE WORKDEPT IS NOT NULL
```

しかし、次の照会をデータベース内の各区画で実行できます (つまり、区画が 5 つあれば、各区画に 1 つずつ、合計 5 つの別個の照会が必要となる)。それぞれの照会は、照会を実行した特定の区画にレコードがあるすべての従業員名のセットを生成します。それぞれのローカル結果セットは、ファイルにリダイレクトすることができます。その後、結果セットは、単一の結果セットにマージする必要があります。

AIX では、ネットワーク・ファイル・システム (NFS) ファイルの特性を使用して、そのマージを自動化することができます。すべての区画が、応答セットを NFS マウント上の同じファイルに送信する場合、結果はマージされます。応答を大きなバッファーにブロック化せずに NFS を使用すると、パフォーマンスがかなり低下する原因になることに注意してください。

```
SELECT FIRSTNME, LASTNAME, JOB FROM EMPLOYEE WHERE WORKDEPT IS NOT NULL
                 AND NODENUMBER(NAME) = CURRENT NODE
```

結果は、ローカル・ファイル (つまり、最終的な結果は 20 個のファイルとなり、それぞれに応答セット全体の一部が含まれる) か、単一の NFS マウント・ファイルのどちらかに保管することができます。

次の例では、2 番目の手法を使用するので、結果は、20 個のノードにわたってマウントされた NFS である単一ファイルに保管されます。 NFS ロック・メカニズムによって、異なる区画から結果ファイルへの書き込みが確実にシリアル化されます。この例は、明示されているように、 NFS ファイル・システムをインストールした AIX プラットフォームでのみ実行されることに注意してください。

```
#define _POSIX_SOURCE
#define INCL_32

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sqlenv.h>
#include <errno.h>
#include <sys/access.h>
#include <sys/flock.h>
#include <unistd.h>

#define BUF_SIZE 1500000  /* Local buffer to store the fetched records */
#define MAX_RECORD_SIZE 80 /* >= size of one written record */

int main(int argc, char *argv[]) {

  EXEC SQL INCLUDE SQLCA;
    EXEC SQL BEGIN DECLARE SECTION;
       char dbname[10];  /* Database name (argument of the program) */
       char userid[9];
       char passwd[19];
       char first_name[21];
       char last_name[21];
       char job_code[11];
  EXEC SQL END DECLARE SECTION;

       struct flock unlock ;  /* structures and variables for handling */
       struct flock lock ;  /* the NFS locking mechanism */
       int lock_command ;
       int lock_rc ;
       int iFileHandle ;  /* output file */
       int iOpenOptions = 0 ;
       int iPermissions ;
       char * file_buf ;  /* pointer to the buffer where the fetched
                             records are accumulated */
       char * write_ptr ;  /* position where the next record is written */
       int buffer_len = 0 ;  /* length of used portion of the buffer */

    /* Initialization */

       lock.l_type = F_WRLCK;  /* An exclusive write lock request */
```

```
      lock.l_start = 0;  /* To lock the entire file */
      lock.l_whence = SEEK_SET;
      lock.l_len = 0;
      unlock.l_type = F_UNLCK;  /* An release lock request */
      unlock.l_start = 0;  /* To unlock the entire file */
      unlock.l_whence = SEEK_SET;
      unlock.l_len = 0;
      lock_command = F_SETLKW;  /* Set the lock */
      iOpenOptions = O_CREAT;  /* Create the file if not exist */
      iOpenOptions |= O_WRONLY;  /* Open for writing only */

   /* Connect to the database */

      if (argc == 3) {
        strcpy( dbname, argv[2] ); /* get  database name from the argument */
         EXEC SQL CONNECT TO :dbname IN SHARE MODE ;
         if ( SQLCODE != 0 ) {
            printf( "Error: CONNECT TO the database failed. SQLCODE = %ld¥n",
             SQLCODE );
     exit(1);
 }
      }
      else if ( argc == 5 ) {
        strcpy( dbname, argv[2] ); /* get  database name from the argument */
strcpy (userid, argv[3]);
strcpy (passwd, argv[4]);
EXEC SQL CONNECT TO :dbname IN SHARE MODE USER :userid USING :passwd;
if ( SQLCODE != 0 ) {
   printf( "Error: CONNECT TO the database failed. SQLCODE = %ld¥n",
           SQLCODE );
     exit( 1 );
        }
      }
      else {
  printf ("¥nUSAGE: largevol txt_file database [userid passwd]¥n¥n");
  exit( 1 ) ;
      } /* endif */

     /* Open the input file with the specified access permissions */

    if ( ( iFileHandle = open(argv[1], iOpenOptions, 0666 ) ) == -1 ) {
      printf( "Error: Could not open %s.¥n", argv[2] ) ;
      exit( 2 ) ;
   }

   /* Set up error and end of table escapes */

   EXEC SQL WHENEVER SQLERROR GO TO ext ;
   EXEC SQL WHENEVER NOT FOUND GO TO cls ;

   /* Declare and open the cursor */

   EXEC SQL DECLARE c1 CURSOR FOR
            SELECT firstnme, lastname, job FROM employee
            WHERE workdept IS NOT NULL
```

```
         AND NODENUMBER(lastname) = CURRENT NODE;
    EXEC SQL OPEN c1 ;

    /* Set up the temporary buffer for storing the fetched result */

    if ( ( file_buf = ( char * ) malloc( BUF_SIZE ) ) == NULL ) {
       printf( "Error: Allocation of buffer failed.¥n" ) ;
       exit( 3 ) ;
    }
    memset( file_buf, 0, BUF_SIZE ) ; /* reset the buffer */
    buffer_len = 0 ;  /* reset the buffer length */
    write_ptr = file_buf ;  /* reset the write pointer */
    /* For each fetched record perform the following    */
    /*  - insert it into the buffer following the       */
    /*    previously stored record                      */
    /*  - check if there is still enough space in  the  */
    /*    buffer for the next record and lock/write/    */
    /*    unlock the file and initialize the buffer     */
    /*    if not                                        */

    do {
       EXEC SQL FETCH c1 INTO :first_name, :last_name, :job_code;
        buffer_len += sprintf( write_ptr, "%s %s %s¥n",
                               first_name, last_name, job_code );
        buffer_len = strlen( file_buf ) ;
       /* Write the content of the buffer to the file if */
       /* the buffer reaches the limit                   */
       if ( buffer_len >= ( BUF_SIZE - MAX_RECORD_SIZE ) ) {
        /*  get excl. write lock */
        lock_rc = fcntl( iFileHandle, lock_command, &lock );
 if ( lock_rc != 0 ) goto file_lock_err;
 /*  position at the end of file */
 lock_rc = lseek( iFileHandle, 0, SEEK_END );
if ( lock_rc < 0 ) goto file_seek_err;
/* write the buffer */
lock_rc = write( iFileHandle,
                               ( void * ) file_buf, buffer_len );
if ( lock_rc < 0 ) goto file_write_err;
  /* release the lock */
 lock_rc = fcntl( iFileHandle, lock_command, &unlock );
 if ( lock_rc != 0 ) goto file_unlock_err;
 file_buf[0] = '¥0' ;  /* reset the buffer */
  buffer_len = 0 ;  /* reset the buffer length */
  write_ptr = file_buf ;  /* reset the write pointer */
       }
       else {
          write_ptr = file_buf + buffer_len ;  /* next write position */
       }
    } while (1) ;

cls:
    /* Write the last piece of data out to the file */
    if (buffer_len > 0) {
       lock_rc = fcntl(iFileHandle, lock_command, &lock);
       if (lock_rc != 0) goto file_lock_err;
```

```
      lock_rc = lseek(iFileHandle, 0, SEEK_END);
       if (lock_rc < 0) goto file_seek_err;
       lock_rc = write(iFileHandle, (void *)file_buf, buffer_len);
       if (lock_rc < 0) goto file_write_err;
       lock_rc = fcntl(iFileHandle, lock_command, &unlock);
      if (lock_rc != 0) goto file_unlock_err;
    }
    free(file_buf);
close(iFileHandle);
    EXEC SQL CLOSE c1;
    exit (0);
 ext:
    if ( SQLCODE != 0 )
       printf( "Error:  SQLCODE = %ld.¥n", SQLCODE );
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL CONNECT RESET;
    if ( SQLCODE != 0 ) {
       printf( "CONNECT RESET Error:  SQLCODE = %ld¥n", SQLCODE );
       exit(4);
    }
    exit (5);
 file_lock_err:
    printf("Error: file lock error = %ld.¥n",lock_rc);
      /* unconditional unlock of the file */
    fcntl(iFileHandle, lock_command, &unlock);
    exit(6);
file_seek_err:
    printf("Error: file seek error = %ld.¥n",lock_rc);
      /* unconditional unlock of the file */
    fcntl(iFileHandle, lock_command, &unlock);
   exit(7);
file_write_err:
    printf("Error: file write error = %ld.¥n",lock_rc);
      /* unconditional unlock of the file */
    fcntl(iFileHandle, lock_command, &unlock);
    exit(8);
file_unlock_err:
    printf("Error: file unlock error = %ld.¥n",lock_rc);
      /* unconditional unlock of the file */
   fcntl(iFileHandle, lock_command, &unlock);
    exit(9);
}
```

この手法は、単一表からの選択だけでなく、さらに複雑な照会にも適用されます。ただし、照会が配列されていない操作を必要とする (つまり、 Explain がコーディネーター・サブセクションの他に複数のサブセクションを表示する) 場合に、照会をすべての区画で並行して実行すると、いくつかの区画であまりにも多くのプロセスが発生することになります。この状況では、必要なだけの区画で照会の結果を一時表 TEMP に保管した後、最終的な抽出を TEMP から並行して実行できます。

選択した担当作業種別のみにしたがって、すべての従業員を抽出する場合は、次のようにして、FIRSTNME、LASTNAME、JOB という名前の列のある TEMP 表を定義することができます。

```
INSERT INTO TEMP
SELECT FIRSTNME, LASTNAME, JOB FROM EMPLOYEE WHERE WORKDEPT IS NOT NULL
       AND EMPNO NOT IN (SELECT EMPNO FROM EMP_ACT WHERE
                         EMPNO<200)
```

次いで、TEMP に対して並列抽出を実行します。

TEMP 表を定義するときには、次の事柄を考慮します。

- 照会が集約 GROUP BY を指定する場合は、 TEMP の区分化キーを GROUP BY 列のサブセットとして定義する必要があります。
- TEMP 表の区分化キーは、表が定義される区画間で均等に分散されるように、十分なカーディナリティー (つまり、応答セット内の別個の値の数) を持っている必要があります。
- TEMP 表を NOT LOGGED INITIALLY 属性を指定して作成し、その表を作成した作業単位を COMMIT して、獲得されたカタログ・ロックをすべて解放してください。
- TEMP 表を使用する時は、 1 つの作業単位内で次のステートメントを出してください。
  1. ALTER TABLE TEMP ACTIVATE NOT LOGGED INITIALLY WITH EMPTY TABLE (TEMP 表を空にしてログ記録をオフにする)
  2. INSERT INTO TEMP SELECT FIRSTNAME...
  3. COMMIT

  このような手法によって、ログ記録を取らずに、またカタログ競合なしで、大きな応答セットを表に挿入できます。 NOT LOGGED 状態を活動化した作業単位でいずれかのエラーが発生すると、 TEMP 表は使えなくなることに注意してください。このことが起きた場合は、TEMP 表を除去して再作成しなければなりません。この理由から、再作成が困難と思われる表にデータを追加する際は、この手法を使わないで ください。

最終的な応答セット (すべてのノードからマージされる部分的な応答セット) をソートする必要がある場合は、次のようにすることができます。

- 最終的な SELECT に SORT BY 文節を指定します。
- それぞれの区画上の別のファイルに抽出を行います。
- それぞれのファイルを 1 つの出力セットにマージします。そのためには、たとえば、`sort -m` AIX コマンドを使用します。

テスト環境の作成

DB2 エンタープライズ・エディションは、 DB2 エンタープライズ拡張エディションのように区分化キー制限を強制することができるので、 DB2 エンタープライズ・エディションによって、区分データベース環境のアプリケーションのためのテスト環境を作成することができます。

1. DB2 エンタープライズ・エディションを使ってデータベース設計のモデルを作成します。
2. 実稼働環境で区画を超えてデータを分配するのに使用するサンプル表を、 PARTITIONING KEY 文節を使って作成します。
3. テスト・データベースに対してアプリケーションを開発して実行します。

DB2 エンタープライズ・エディションは、 DB2 エンタープライズ拡張エディションと整合性がある区分化キー制限を強制し、ご使用のアプリケーションに役立つテスト環境を提供します。

エラー処理考慮事項

区画環境では、DB2 は SQL ステートメントをサブセクションに細分化し、その各部分が関連データを持つ区画で処理されます。結果として、アプリケーションにアクセスできない区画で、エラーが発生することがあります。これは、単一区画環境では生じません。

次の事柄を考慮する必要があります。
- 非 CURSOR (EXECUTE) 非重大エラー
- CURSOR 非重大エラー
- 重大エラー
- マージされた複数の SQLCA 構造
- エラーを戻した区画を識別する方法

重大エラーが原因でアプリケーションが異常終了した場合、未確定トランザクションがデータベースに残っている可能性があります。 (1 つのフェーズが正常に完了すると、未確定トランザクションはグローバル・トランザクションに関係付けられますが、後続のフェーズが完了する前にシステムに障害が起き、データベースは不整合状態のままになります。) それらの処理に関する情報は、 *管理の手引き* を参照してください。

重大エラー

重大エラーが DB2 ユニバーサル・データベースで発生すると、以下のいずれかが発生します。

- エラーが発生したノード上のデータベース・マネージャーが遮断します。

  活動中の作業単位がロールバックされません。

この状況では、ノード、および遮断が生じたときにノードで活動中だったデータベースを回復する必要があります。

- すべてのエージェントが、エラーの発生したノードでデータベースを強制的に切断します。

  そのデータベースのすべての作業単位がロールバックされます。

  この状況では、エラーの発生したノードにあるデータベースが、不整合とマークされます。それにアクセスしようとすると、SQLCODE -1034 (SQLSTATE 58031) または SQLCODE -1015 (SQLSTATE 55025) のいずれかが戻されます。ユーザーまたは別のノードの他のアプリケーションが、このノードのデータベースにアクセスできるようにするには、そのデータベースに対して RESTART DATABASE コマンドを実行する必要があります。このコマンドについては、*コマンド解説書* を参照してください。

重大エラー SQLCODE -1224 (SQLSTATE 55032) は、さまざまな理由で発生します。このメッセージを受け取ったならば、SQLCA を調べてみます。そこに、障害が起きたノードが示されています。その後、ノード間で共用している `db2diag.log` ファイルで、詳細を調べます。追加情報については、585ページの『エラーを戻した区画を識別する』を参照してください。

マージされた複数の SQLCA 構造

1 つの SQL ステートメントは、異なるノード上の多数のエージェントが実行することができ、各エージェントが異なるエラーまたは警告のために異なる SQLCA を戻すことがあります。コーディネーター・エージェントも独自の SQLCA を持っています。さらに、SQLCA には、グローバル数を示すフィールドもあります (行カウントを示す *sqlerrd* フィールドなど)。アプリケーションに整合性のある視点を提供するため、すべての SQLCA 値を 1 つの構造にマージします。この構造は、*SQL 解説書* に解説されています。

エラー報告は次のようになされます。

- 重大エラー条件は必ず報告されます。重大エラーを報告するとすぐに、重大エラーが SQLCA に追加されます。
- 重大エラーが発生していなければ、デッドロック・エラーが他のエラーに優先します。
- 他のすべてのエラーについては、最初の負の SQLCODE の SQLCA がアプリケーションに戻されます。
- 負の SQLCODE が検出されなければ、最初の警告 (つまり正の SQLCODE) の SQLCA がアプリケーションに戻されます。ただし、表の 1 つの区画は空でも別の区画にデータが入っている場合、その表でデータ処理操作を行っても、上記のようにはなりません。SQLCODE +100 がアプリケーションに戻されるのは、表のすべての区画で空であるか、または UPDATE ステートメントの WHERE 文節を満たす行が表にないために、すべての区画のエージェントが SQL0100W を戻した場合だけです。

- すべてのエラーおよび警告において、*sqlwarn* フィールドに、すべてのエージェントから受け取った警告標識が入っています。
- 行カウントを示す *sqlerrd* フィールドの値は、すべてのエージェントから累計されます。

アプリケーションは、最初のエラーまたは警告の原因となった問題を修正した後、続いて次のエラーまたは警告を受け取ることがあります。エラーは SQLCA に報告されて、検出した最初のエラーが他のエラーに優先するかを確認します。これは、以前のエラーが引き起こしたエラーが、修正したばかりのエラーを重ね書きしていないか確認するためです。重大エラーおよびデッドロック・エラーは、最高の優先順位を与えられています。なぜなら、それらについてはコーディネーター・エージェントがただちに処置を行う必要があるからです。

エラーを戻した区画を識別する

区画がエラーまたは警告を戻す場合、その番号が SQLCA の SQLERRD(6) フィールドに示されます。このフィールド内の番号は、`db2nodes.cfg` ファイルでその区画に指定されたものと同じです。

SQL ステートメントまたは API 呼び出しが成功した場合は、このフィールドの区画番号は無効です。

SQLCA については、*SQL 解説書* を参照してください。

デバッグ

次の節で解説されているツールは、アプリケーションのデバッグに使用することができます。詳細については、*問題判別の手引き* を参照してください。

ループ状態または延期状態のアプリケーションの診断

照会またはアプリケーションを開始した後、それが延期状態 (活動を表示しない) またはループ状態 (活動は表示するが、結果をアプリケーションに戻していない) であることに気付く場合があります。ロック・タイムアウトをオンにしているかを確認してください。しかし、ある状況では、エラーが戻されません。これらの状況では、データベース・システム・モニター・スナップショットや、 *問題判別の手引き* で解説されているツールが役に立ちます。

アプリケーションのデバッグに役立つデータベース・システム・モニターの機能の 1 つは、すべての活動中エージェントの状況の表示です。スナップショットを最大限に活用するには、アプリケーションの実行前に (できれば DB2START の実行直後に)、ステートメントのコレクションが実行されているかを、次のように確認してください。

```
db2_all "db2 UPDATE MONITOR SWITCHES USING STATEMENT ON"
```

アプリケーションまたは照会が、停止またはループしているように思える場合は、次のコマンドを実行します。

```
db2_all "db2 GET SNAPSHOT FOR AGENTS ON データベース
```

スナップショットから収集した情報の読み取り方法、およびデータベース・システム・モニターの使用についての詳細は、 *システム・モニター 手引きおよび解説書* を参照してください。

第19章 DB2 連合システム用のプログラムの作成

DB2 連合システムの紹介 587
データ・ソース表および視点へのアクセス 588
ニックネームによる作業 588
データ・ソース表および視点についての情報のカタログ. 589
考慮事項および制約事項 589
列オプションの定義 590
視点でのニックネームの使用 592
データ保全性を保守するための分離レベルの使用 592
データ・タイプ・マッピングの処理. . . . 593
ローカルに定義するデータ・タイプをDB2 が判別する方法. 593
デフォルトのデータ・タイプ・マッピング 593
デフォルト・タイプ・マッピングを上書きして新しいタイプ・マッピングを作成する方法 594
1 つまたは複数のデータ・ソースに適用されるタイプ・マッピングの定義. . 594
特定の表のタイプ・マッピングの変更 594
ラージ・オブジェクト (LOB) サポート . . 595
DB2 による LOB の検索方法. 595
LOB ストリーミング. 596
LOB 具体化. 596
アプリケーションが LOB ロケーターを使用する方法 596
LOB での制約事項 597
LOB と非 LOB データ・タイプの間のマッピング. 597
分散要求を使用したデータ・ソースの照会 597
分散要求のコーディング 597
副照会による要求. 598
セット演算子による要求 598
結合の要求 599
サーバー・オプションを使用して最適化を容易にする 599
データ・ソース関数の呼び出し 600
DB2 がデータ・ソース関数を呼び出せるようにする 600
関数を呼び出す際のオーバーヘッドの削減 601
CREATE FUNCTION MAPPING ステートメントでの関数名の指定 602
関数マッピングの中断 602
パススルーを使用してデータ・ソースを直接照会する. 603
パススルー・セッションでの SQL 処理 603
考慮事項および制約事項 603
すべてのデータ・ソースにパススルーを使用する 604
Oracle データ・ソースでパススルーを使用する. 604

DB2 連合システムの紹介

DB2 連合システム とは、以下のもので構成されている分散コンピューター・システムです。

- DB2 サーバー。*連合サーバー* と呼ばれます。
- 連合サーバーが照会を送信する先の複数の半自律データ・ソース。それぞれのデータ・ソースは、リレーショナル・データベース管理システムのインスタンスと、そのインスタンスがサポートするデータベースで構成されます。 DB2 連合システムのデータ・ソースには、 Oracle インスタンスおよび DB2 ファミリーのメンバーのインスタンスを含めることができます。

クライアント・アプリケーションには、データ・ソースは 1 つの収集データベースに見えます。しかし、アプリケーションは実際にはデータベースとのインターフェースを持

© Copyright IBM Corp. 1993, 2001

っています。これが*連合データベース* と呼ばれるもので、連合サーバーの内部にあります。データ・ソースからデータを取得するには、DB2 SQL での照会を連合データベースへ実行依頼します。次に、DB2 は照会を適切なデータ・ソースに配布し、要求されたデータを収集して、このデータをアプリケーションに戻します。

アプリケーションは、DB2 SQL を使用して、 DB2 が認識できるすべてのデータ・タイプ (ただし、LOB データ・タイプは除く) の値を要求できます。データ・ソースに書き込む (たとえば、データ・ソース表を更新する) には、アプリケーションは、パススルーと呼ばれる特殊モードでデータ・ソース自体の SQL を使用しなければなりません。

連合データベースのシステム・カタログには、データベース内のオブジェクトについての情報に限らず、データ・ソースおよびそれらの中にある特定の表、視点、関数についての情報も含まれます。カタログには、連合システム全体についての情報が含まれます。したがって、これは*グローバル・カタログ* と呼ばれます。

DB2 連合システムのさらに詳しい概要については、 *管理の手引き: 計画* を参照してください。拡張された概要については、 *SQL 解説書* を参照してください。アプリケーションが実行依頼できる DB2 SQL 照会の例については、 597ページの『分散要求を使用したデータ・ソースの照会』を参照してください。パススルーの詳細については、 603 ページの『パススルーを使用してデータ・ソースを直接照会する』を参照してください。

データ・ソース表および視点へのアクセス

この節では、データ・ソース表および視点にアクセスし、使用するのに役立つ情報を提供します。ここでは、以下のトピックについて説明します。

- 表および視点に割り当てるニックネーム。連合サーバーはそれらのニックネームを参照できます。
- 表および視点にアクセスする際に、データ・ソースでのデータ保全性を保守する助けとなる分離レベル。

ニックネームによる作業

ニックネーム とは、アプリケーションがデータ・ソース表または視点を参照するための ID です。この節では、以下の事柄を扱います。

- ニックネームを付けられた表についての情報が、グローバル・カタログにどのように提供されるかを説明します。
- ニックネームによる作業の際に注意しておくべき考慮事項および制約事項をリストします。
- 照会を最適化するために設定できるパラメーターを説明します。
- ニックネームによって参照される視点を使用する方法を説明します。

データ・ソース表および視点についての情報のカタログ

データ・ソース表または視点にニックネームが付けられると、 DB2 は、この表または視点からデータを検索する方法を計画する際に最適化プログラムが使用できる情報でグローバル・カタログを更新します。この情報には、一例として、表または視点の名前、および表の列または視点の列の名前と属性が含まれます。

表の場合には、以下の情報も含まれます。

- 統計 (たとえば、行数および行が存在するページの数)。 DB2 が必ず最新の統計を取得するためには、ニックネームを付ける前に、表に対してデータ・ソースの RUNSTATS コマンドと同等の作業を実行することをお勧めします。
- 表に索引がある場合は、その索引の説明。表に索引がない場合には、索引定義に一般的に含まれるメタデータ (たとえば、表の中のどの列に固有値があるか、および固有の行があるかどうかなど) をカタログに提供できます。表のニックネームに対して CREATE INDEX ステートメントを実行すると、メタデータを生成できます。このようなメタデータを集合的に*索引仕様* と言います。この場合、ステートメントは索引仕様だけを生成し、実際の索引は作成しないことに注意してください。このステートメントの説明については、 *SQL 解説書* を参照してください。

グローバル・カタログにデータ・ソース表についてのどのような情報が保管されているかを調べるには、 SYSCAT.TABLES および SYSCAT.COLUMNS カタログ視点を照会します。カタログに表の索引についてのどのような情報が保管されているか、または特定の索引視点に何が含まれているかを調べるには、 SYSCAT.INDEXES カタログ視点を照会します。これらの視点の詳細については、 *SQL 解説書* を参照してください。表および索引の情報でグローバル・カタログを更新することの詳細については、 *管理の手引き: インプリメンテーション* を参照してください。

考慮事項および制約事項

以下の作業を実行する際には、いくつかの考慮事項および制約事項に注意する必要があります。

- ニックネームを定義、変更、および除去する
- 表および視点をニックネームによって参照する
- ニックネームによって参照された表および視点で操作を実行する

ニックネームの定義、変更、および除去:

- 表または視点にニックネームを定義するには、 CREATE NICKNAME ステートメントを使用します。このステートメントは、以下の事柄を行います。
  - Oracle 表または視点を名前で参照する。
  - DB2 ファミリーの表または視点を名前で参照する。別名がある場合には、その別名で参照する。
- 同じ表または視点に複数のニックネームを定義できます。また、CREATE ALIAS ステートメントを使用すると、ニックネームの別名も定義できます。

- ニックネームを変更するには、いったん除去してから置き換える必要があります。除去するには、DROP NICKNAME ステートメントを使用します。置き換えるには、CREATE NICKNAME ステートメントを使用します。
- ニックネームを除去すると、このニックネームを使用して定義された視点が作動不能になり、これに依存した計画が無効になります。

CREATE NICKNAME、CREATE ALIAS、および DROP NICKNAME ステートメントについての資料は、 *SQL 解説書* を参照してください。

ニックネームによる表および視点の参照:

- データ・ソース表または視点にニックネームを付けると、この表または視点はそのニックネームを使用しなければ参照できなくなります (パススルー・セッションを除く)。たとえば、DB2MVS1.PERSON.DEPT という表を表すのにニックネーム DEPT を定義した場合、ステートメント SELECT * FROM DEPT は使用できますが、SELECT * FROM DB2MVS1.PERSON.DEPT は使用できません。しかし、パススルー・セッションでは、データ・ソース名で表または視点にアクセスする必要があります。
- CREATE TRIGGER ステートメントでは、ニックネームを参照できません。
- CREATE TABLE ステートメントの summary-table-definition 文節でニックネームを参照する場合、この文節には DEFINITION ONLY キーワードも指定する必要があります。

ニックネームの付いた表または視点での操作の実行:

- COMMENT ON ステートメントは、ニックネームおよびニックネームで定義されている列には有効です。このステートメントはグローバル・カタログを更新しますが、データ・ソース・カタログは更新しません。
- GRANT および REVOKE ステートメントは、特定の特権およびすべてのユーザーとグループのニックネームに有効です。しかし、DB2 はニックネームが参照する表または視点には、対応する GRANT または REVOKE を発行しません。ニックネーム特権の詳細については、 *管理の手引き: 計画* を参照してください。
- データ・ソースは読み取り専用です。そのため、以下のようになります。
  - INSERT、UPDATE、および DELETE ステートメントはニックネームには無効です。
  - ニックネームに UNION ALL 節を含む視点は更新できません。
- ニックネームに対しては DB2 ユーティリティー (RUNSTATS、IMPORT、EXPORT など) を実行できません。

列オプションの定義

表または視点にニックネームを定義する際には、グローバル・カタログに、表または視点の特定の列についての情報を提供することができます。この情報は、*列オプション* と

呼ばれるパラメーターに割り当てた値の形式で指定します。値は、大文字でも小文字でも指定できます。 表27 では、列オプションとそれらの値を説明します。

表 27. 列オプションとその設定

オプション	有効な設定	デフォルト設定
numeric_string	‘y’ はい。この列には数字データのストリングだけが含まれています。重要: この列に、数値データに後書きブランクが続いたストリングしか含まれていない場合には、 ‘y’ を指定しないようにお勧めします。 ‘n’ いいえ。この列は数字データのストリングに限定されません。 列の numeric_string を ‘y’ と設定することにより、この列にはブランクが含まれていないことを最適化プログラムに通知することになります (ブランクがあると、列のデータのソートができなくなる可能性があります)。	‘n’
varchar_no_trailing_blanks	特定の VARCHAR 列に後書きブランクがないかどうかを指示します。 ‘y’ はい。後書きブランクは、この VARCHAR 列にはありません。 ‘n’ いいえ。後書きブランクは、この VARCHAR 列にあります。 データ・ソース VARCHAR 列に埋め込みブランクが含まれていない場合、最適化プログラムがこの列にアクセスする戦略は、後書きブランクが含まれているかどうかということに依存してきます。デフォルトでは、最適化プログラムは、後書きブランクが実際に入っているものと『みなします』。この前提のもとで、最適化プログラムは照会の修正に関係するアクセス戦略を開発するので、結果として、これらの列から戻される値はユーザーが予期したものとなります。しかし、VARCHAR 列に後書きブランクがなく、そのことを最適化プログラムに認識させると、アクセス戦略の開発はさらに効果的になります。特定の列に後書きブランクがないことを最適化プログラムに通知するには、この列を ALTER NICKNAME ステートメントに指定します (指針については、 *SQL 解説書* を参照してください)。	‘n‘

ALTER NICKNAME ステートメントで列オプションを設定します。このステートメントの詳細については、 *SQL 解説書* を参照してください。

視点でのニックネームの使用

視点でニックネームを使用するには、主に次の 2 つの方法があります。

- データ・ソース視点にニックネームを作成する。連合サーバーは、データ・ソース視点のニックネームを、データ・ソース表のニックネームと同じように扱います。
- ニックネームの付いたデータ・ソース表および視点の連合データベース視点を作成する。たとえば、連合サーバーは異なる位置にある基礎表の結合に適応できるため、異なるデータ・ソースにある基礎表の連合データベース視点を簡単に定義できます。このような複数位置対応の視点は、グローバルな統合データベースに対して高度のデータ独立性を提供します。これは、複数のローカル表で定義される視点が、集中化されたリレーショナル・データベース・マネージャーに対して高度のデータ独立性を提供するのと同じです。このグローバル視点のメカニズムは、連合サーバーが高度のデータ独立性を提供する方法の 1 つです。

  データ・ソース・データが連合データベース視点を作成する処置は、『ニックネームでの視点の作成』と呼ばれることがあります。この句は、作成される視点について、CREATE VIEW ステートメントの全選択が、その視点に含まれることになるそれぞれの表および視点のニックネームを参照しなければならないことを示唆しています。

視点には、それ自体の統計または索引はありません。視点はデータベース上にある実際の表ではないためです。このことは、視点の構造および内容が単一基礎表と全く同じである場合であっても、当てはまります。統計および索引の詳細については、 *管理の手引き: インプリメンテーション* を参照してください。

データ保全性を保守するための分離レベルの使用

データ・ソース表のデータ保全性を保守するには、その表の行が特定の分離レベルでロックされるように要求します。たとえば、ある行へのアクセスを 1 つに限るには、この行に反復可能読み取り (RR) 分離レベルを指定します。

連合サーバーは、要求する分離レベルを、データ・ソースの対応するレベルにマップします。これを説明するために、 593ページの表28 では以下のものをリストします。

- 要求できる分離レベル。以下のとおりです。

  CS　カーソル固定

  RR　反復可能読み取り

  RS　読み取り固定

  UR　非コミット読み取り
- 要求されたレベルがマップする先の Oracle 分離レベル。

表 28. 連合サーバーと Oracle データ・ソースの分離レベルの比較

連合サーバー (DB2)	CS	RR	RS	UR
Oracle	デフォルト	トランザクション読み取り専用	トランザクション読み取り専用	カーソル固定と同じ

データ・タイプ・マッピングの処理

データ・ソース表にニックネームを作成すると、 DB2 はその表についての情報をグローバル・カタログに含めます。この情報には、ニックネーム、表の名前、すべての列の名前が含まれますが、これに限られるわけではありません。列ごとに以下の情報が含まれます。

- データ・ソースで列に定義されたデータ・タイプ。 (この節では、このタイプをリモート・タイプとします。)
- DB2 にサポートされ、連合データベースに登録される、対応するデータ・タイプ。 (この節は、このタイプをローカル・タイプとします。)

この節では、DB2 がデータ・タイプ・マッピングを使用して、データ・ソース表の列に、DB2 がサポートするどのデータ・タイプを定義すべきかを判別する方法を説明します。次に、データ・タイプ・マッピング (略して『タイプ・マッピング』と呼ばれることもある) について、 2 つの節に分けて説明します。まず、デフォルト・マッピングについて説明し、次にデフォルト・マッピングを上書きして新しいマッピングを作成する方法を説明します。

ローカルに定義するデータ・タイプを DB2 が判別する方法

DB2 はリモート列に使用するローカル・タイプをどのように判別するのでしょうか。 DB2 は、データ・ソースの列のタイプと、対応するローカル・タイプとの間のマッピングを調べ、後者を選択します。たとえば、DB2 が提供するデフォルト・マッピングでは、 DB2 (VSE および VM 版) のデータ・タイプ CHAR は 254 バイトまでサポートし、 DB2 データ・タイプ CHAR を示しています。したがって、DB2 (VSE および VM 版) 表にニックネームを作成する際に、表の列 C1 に最大長 200 バイトのデータ・タイプ CHAR がある場合、このデフォルトを上書きしないかぎり、 C1 には DB2 タイプ CHAR がローカルに定義されます。

デフォルトのデータ・タイプ・マッピング

RDBMS 間の相違により、データ・ソースのデータ・タイプと連合サーバーのデータ・タイプの間のデフォルト・マッピングは、 1 対 1 とは限りません。しかし、要求されたすべての値が戻されたことを確認するには、このマッピングで十分です。

たとえば、次の 2 つの間にデフォルトのタイプ・マッピングがあるとします。

- Oracle タイプが NUMBER(9,0) (ここで、9 は最大精度、0 は最大位取り)
- DB2 タイプが INTEGER。最大長 4 バイト。

タイプ NUMBER(9,0) の列 C2 がある Oracle 表にニックネームを付けたとします。デフォルト・マッピングを変更しない場合は、 C2 のタイプはローカルに INTEGER と定義されます。さらに、4 バイトの INTEGER は最大精度 10 をサポートするため、連合サーバーから C2 が照会される際には、必ず C2 のすべての値が戻されます。

デフォルトのデータ・タイプ・マッピングのリストについては、 *SQL 解説書* を参照してください。

デフォルト・タイプ・マッピングを上書きして新しいタイプ・マッピングを作成する方法

前述の例が示すように、デフォルト・マッピングではローカル・タイプとリモート・タイプが類似しているために、リモート・タイプが定義されているリモート列を参照する際に、ローカルとリモートの両方のタイプに適合するすべての値が戻されます。しかし、代替マッピングが必要な場合もあります。その場合のシナリオを考慮します。

1 つまたは複数のデータ・ソースに適用されるタイプ・マッピングの定義

Oracle データ・タイプの 3 つの表の特定の列には、タイム・スタンプを表すデータ・タイプ DATE があります。デフォルト・マッピングでは、このタイプはローカル DB2 タイプ TIMESTAMP になります。デフォルトを変更しないで 3 つのテーブルのニックネームを作成する場合、 TIMESTAMP はこれらの列にローカルに定義され、列の DB2 照会でタイム・スタンプが生成されます。しかし、このような照会で時刻だけを生成したいと思っているとします。この場合、Oracle DATE を DB2 タイプ TIME にマップしてから、デフォルトを上書きします。こうすると、ニックネームを作成するときに、TIMESTAMP ではなく TIME が列にローカルに定義されます。結果として、これらを照会する際、タイム・スタンプの時刻の部分だけが戻されることになります。デフォルト・タイプ・マッピングを上書きするには、 CREATE TYPE MAPPING ステートメントを使用できます。

CREATE TYPE MAPPING ステートメントでは、新しいマッピングを特定のデータ・ソースに適用するか (たとえば、組織の 1 部門が使用するデータ・ソース)、特定のタイプのすべてのデータ・ソースに適用するか (たとえば、すべての Oracle データ・ソース)、あるいはあるタイプの特定のバージョンのすべてのデータ・ソースに適用するか (たとえば、すべての Oracle 8.0.3 のデータ・ソース) のいずれかを指定できます。

特定の表のタイプ・マッピングの変更

特定のテーブルのタイプ・マッピングのローカル・タイプを変更できます。たとえば、Oracle データ・タイプ NUMBER(32,3) は、デフォルトでは DB2 データ・タイプ DOUBLE という浮動小数点データ・タイプにマップします。従業員情報についての Oracle 表で、列 BONUS がデータ・タイプ NUMBER(32,3) で定義されているとします。マッピングにより、BONUS の照会では以下のような値を戻します。

```
5.0000000000000E+002
1.0000000000000E+003
```

ここで、+002 は、小数点を右に 2 桁分移動させなければならないことを意味し、+003 は、小数点を右に 3 桁分移動させなければならないことを意味します。

BONUS の照会でドル単位のような値を戻すようにするには、この表について、実際のボーナスの形式を反映するような精度と位取りで NUMBER(32,3) を DB2 DECIMAL タイプに再マップします。たとえば、ボーナスのドル部分が 6 桁を超えることがないとわかっている場合には、 NUMBER(32,3) を DECIMAL(8,2) に再マップできます。この新しいマッピングの制約の下では、 BONUS の照会で戻される値は以下のようになります。

```
 500.00
1000.00
```

特定の表の列のタイプ・マッピングを変更するには、 ALTER NICKNAME ステートメントを使用します。このステートメントを使用すると、ニックネームが定義されている表の列にローカルに定義されているタイプを変更できます。

ラージ・オブジェクト (LOB) サポート

連合データベース・システムでは、リモート・データ・ソースにある LOB にアクセスし、それらを操作することができます。 LOB は非常に大きい場合があるため、リモート・データ・ソースから LOB を転送するときに時間がかかることがあります。 DB2 連合データベースはデータ・ソースからの LOB データの転送を最小限に抑えようとします。また、DB2 で LOB を具体化せずに、要求された LOB データをデータ・ソースから要求側アプリケーションに直接送達しようとします。

この節では、以下の事柄を扱います。

- DB2 による LOB の検索方法
- アプリケーションが LOB ロケーターを使用する方法
- LOB での制約事項
- LOB と非 LOB データ・タイプの間のマッピング
- システムのチューニング

DB2 による LOB の検索方法

DB2 連合システムは、次の 2 つのメカニズムを使用して LOB を検索します。それは、LOB ストリーミングと LOB 具体化です。

LOB ストリーミング

LOB ストリーミングでは、LOB データは段階的に検索されます。 DB2 は、完全にプッシュダウンされた照会の結果セット内にあるデータに対して LOB ストリーミングを使用します。たとえば、以下の照会を考慮します。

```
SELECT empname, picture FROM orc_emp_table WHERE empno = '01192345'
```

ここで、*picture* は LOB 列を表し、 *orc_emp_table* は、従業員データを含む Oracle 表を参照するニックネームを表します。 DB2 照会プロセッサーは、Oracle データ・ソースで照会全体を実行することにした場合、 *picture* 列にストリーミングのマークを付けます。実行時に、DB2 は LOB にストリーミングのマークが付けられていることに気づくと、データ・ソースから段階的に LOB を検索します。そして、DB2 はデータをアプリケーションのメモリー・スペースに転送します。

LOB 具体化

LOB 具体化では、リモート LOB データは DB2 によって検索され、連合サーバーにローカルに保管されます。 DB2 は、以下の場合に LOB 具体化を使用します。

- LOB 列の据え置きまたはストリーミングができない。
- データが転送される前に、関数を LOB 列にローカルに適用する必要がある。これは、DB2 がリモート・データ・ソースで使用できない関数を補正する際に行われます。たとえば、Microsoft SQL Server は SUBSTR 関数を LOB 列に提供しません。補正するには、DB2 は LOB 列をローカルに具体化し、 DB2 SUBSTR 関数を検査済みの LOB に適用します。

アプリケーションが LOB ロケーターを使用する方法

アプリケーションは、LOB ロケーターにリモート・データ・ソースに保管されている LOB を要求できます。 LOB ロケーターはホスト変数に保管されている 4 バイトの値で、プログラムはこのホスト変数を使用してデータベース・システム中に保留されている LOB 値 (または LOB 式) を参照することができます。プログラムは、 LOB ロケーターを使用して LOB 値が標準ホスト変数に保管されているかのように操作できます。 LOB ロケーターを使用する場合の相違点は、 LOB 値をサーバーからアプリケーションへ移送する (そして再度戻す) 必要がないということです。 LOB ロケーターの詳細については、367ページの『ラージ・オブジェクト・ロケーターについて』を参照してください。

DB2 はリモート・データ・ソースから LOB を検索し、それらを DB2 に保管してから、保管された LOB に対して LOB ロケーターを発行します。 LOB ロケーターは、以下の場合に解放されます。

- アプリケーションが ″FREE LOCATOR″ SQL ステートメントを出すとき
- アプリケーションが COMMIT ステートメントを出すとき
- DB2 が再始動されるとき

LOB での制約事項

LOB を使用および検索する際には、次のことに注意してください。

- DB2 はリモート LOB をファイル参照変数にバインドできません。
- LOB はパススルー・モードではサポートされません。

LOB と非 LOB データ・タイプの間のマッピング

データ・ソースで DB2 LOB データ・タイプを非 LOB データ・タイプにマップできる場合があります。データ・ソースにおいて DB2 LOB タイプの列とその相手側の列の間でマッピングを作成する必要がある場合、可能であれば相手側に LOB データ・タイプを使用してください。

マッピングを作成するには、type mapping DDL ステートメントを使用します。以下に例を示します。

```
CREATE TYPE MAPPING my_oracle_lob FROM sysibm.clob TO SERVER TYPE oracle TYPElong
```

ここで、各パラメーターは以下のとおりです。

my_oracle_lob
: タイプ・マッピングの名前です。

sysibm.clob
: DB2 CLOB データ・タイプです。

oracle
: 接続先のサーバーのタイプです。

long
: 相手側の Oracle データ・タイプです。

分散要求を使用したデータ・ソースの照会

連合データベースに実行依頼される照会では、単一データ・ソースによって生成される結果を要求できます。しかし、通常は、複数のデータ・ソースによって生成される結果を要求します。通常の照会は複数のデータ・ソースに分散されるため、これは*分散要求*と呼ばれます。

この節では、以下の事柄を扱います。

- 分散要求をコード化する方法を示します。
- 特定の分散要求の最適化を行わせる方法を紹介します。

分散要求のコーディング

一般に、分散要求では 3 つの SQL 規則のうちの 1 つまたは複数を使用して、データの検索元、つまり、副照会、セット演算子、および結合副選択を指定します。この節では、次のようなシナリオでのコンテキストでの例を示します。ある連合サーバーが DB2 ユニバーサル・データベース (OS/390 版) データ・ソース、 DB2 ユニバーサル・データベース (AS/400 版) データ・ソース、および Oracle データ・ソースにアクセスでき

るように構成されています。それぞれのデータ・ソースには、従業員の情報を含む表が保管されています。この連合サーバーは、それらの表がある位置を示すニックネームによって表を参照します。表にはそれぞれ、UDB390_EMPLOYEES、AS400_EMPLOYEES、および ORA_EMPLOYEES というニックネームが付けられています。 (ニックネームがデータ・ソースを参照する必要はありません。このシナリオのニックネームがデータ・ソースを参照するのは、表が異なる RDBMS にあることを強調しているにすぎません。) Oracle データ・ソースには、ORA_EMPLOYEES に加えて、ORA_COUNTRIES というニックネームの表があります。ここには、従業員が住んでいる国についての情報が含まれています。

副照会による要求

表 AS400_EMPLOYEES には、アジア在住の従業員の電話番号があります。また、電話番号に関連する国番号も含まれていますが、そのコードがどの国を表すかはリストされていません。しかし、表 ORA_COUNTRIES にはコードと国の両方がリストされています。次の照会では、副照会を使用して、中国の国別コードを検出します。さらに、SELECT および WHERE 文節を使用して、この特定のコードを必要とする電話番号を持つ従業員を AS400_EMPLOYEES にリストします。

```
SELECT name, telephone
   FROM djadmin.as400_employees
   WHERE country_code IN
      (SELECT country_code
         FROM djadmin.ora_countries
         WHERE country_name = 'CHINA')
```

セット演算子による要求

連合サーバーは、以下の 3 つのセット演算子をサポートします。

- UNION

  このセット演算子を使用すると、 2 つ以上の任意の SELECT ステートメントに適合する行を結合できます。

- EXCEPT

  このセット演算子を使用すると、最初の SELECT ステートメントに適合し、 2 番目のステートメントには適合しない行を検索できます。

- INTERSECT

  このセット演算子を使用すると、両方の SELECT ステートメントに適合する行を検索できます。

3 つのすべてのセット演算子は、ALL オペランドを指定して、重複行が結果から削除されないように示すことができます。こうすると、余分なソート作業をする必要がなくなります。

以下の照会では、 AS400_EMPLOYEES および UDB390_EMPLOYEES 表の両方に存在するすべての従業員名および国別コードを検索します。これは、それぞれの表が異なるデータ・ソースにある場合でも実行されます。

```
SELECT name, country_code
   FROM as400_employees
INTERSECT
SELECT name, country_code
   FROM udb390_employees
```

結合の要求

関係結合では、複数の表から検索された列の組み合わせを含む結果セットを生成します。結果セットの行のサイズを制限する条件を指定しなければならないことに注意してください。

以下の照会では、2 つの表にリストされている国別コードを比較して、従業員の名前とそれに対応する国名を結合します。それぞれの表は異なるデータ・ソースにあります。

```
SELECT t1.name, t2.country_name
   FROM djadmin.as400_employees t1, djadmin.ora_countries t2
   WHERE t1.country_code = t2.country_code
```

サーバー・オプションを使用して最適化を容易にする

連合システムのユーザーは、サーバー・オプション と呼ばれるパラメーターを使用して、データ・ソースに全体として適用される情報をグローバル・カタログに提供したり、 DB2 とデータ・ソースとの対話を制御したりできます。たとえば、データ・ソースの基本として使用できるインスタンスの ID をカタログするには、データベース管理者はその ID を値としてサーバー・オプション 『node』 に割り当てます。

サーバー・オプションの中には、 DB2 とデータ・ソースとの対話の主なエリアをアドレッシングするものがあります。これが照会の最適化です。たとえば、列オプション 『varchar_no_trailing_blanks』 を使用して、後書きブランクのない特定のデータ・ソース VARCHAR 列を DB2 最適化プログラムに通知できるのと同じように、サーバー・オプション (これも 『varchar_no_trailing_blanks』と呼ばれる) を使用して、 VARCHAR 列に後書きブランクのないデータ・ソースを最適化プログラムに通知することができます。最適化プログラムがアクセス戦略を作成するのにこのような情報がどのように役立つかについての要約は、 591ページの表27 を参照してください。

さらに、サーバー・オプション 『plan_hints』 を、 DB2 が Oracle データ・ソースにステートメント・フラグメントを提供することを可能にする値に設定できます。このステートメント・フラグメントは*計画のヒント* と呼ばれ、 Oracle 最適化プログラムがジョブを実行するための助けになります。特に、計画のヒントを使用すると、表にアクセスする際にどの索引を使用するか、およびデータから結果セットを検索する際にどの表結合順序列を使用するか、などの事柄を、最適化プログラムが決定するのに役立ちます。

通常は、データベース管理者が連合システムにサーバー・オプションを設定します。しかし、プログラマーは照会を最適化する助けとなるこれらのオプションを十分に活用できます。たとえば、データ・ソース ORACLE1 および ORACLE2 について、

plan_hints サーバー・オプションがデフォルトの ‘n’ (いいえ、このデータ・ソースに計画のヒントを提供しません) に設定されているとします。また、ORACLE1 および ORACLE2 からのデータの分散要求を作成し、計画のヒントを使用して、それらのデータ・ソースでの最適化プログラムがこのデータにアクセスするための戦略を改善するとします。アプリケーションが連合データベースに接続されている間は、デフォルトを ‘y’ (はい、計画のヒントを提供します) を設定して上書きできます。接続が完了すると、設定は自動的に ‘n’ に戻ります。

連合データベースへの接続の期間についてサーバー・オプションを設定するには、 SET SERVER OPTION ステートメントを使用します。設定を有効にするために、必ず CONNECT ステートメントの直後にこのステートメントを指定してください。さらに、ステートメントをすぐ指定できるように準備しておくことをお勧めします。

SET SERVER OPTION ステートメントについての資料は、 *SQL 解説書* を参照してください。すべてのサーバー・オプションとそれらの設定の詳細については、 *管理の手引き: インプリメンテーション* を参照してください。

データ・ソース関数の呼び出し

この節では、以下の事柄を実行する方法を説明します。

- DB2 が認識しないデータ・ソース関数を呼び出せるようにする
- DB2 が関数を呼び出す際に消費されるオーバーヘッドを削減する
- 互いにマップする関数の名前を指定する
- 関数間のマッピングの使用を中断する

DB2 がデータ・ソース関数を呼び出せるようにする

DB2 が認識しないデータ・ソース関数を呼び出すようにしたいと思うかもしれません。そのような関数は、ユーザー定義関数か、または DB2 が認識しない新規の組み込み関数である可能性があります。

DB2 が認識しないデータ・ソース関数にアクセスできるようにするには、その前に、この関数と連合データベースに保管されている関数の間にマッピングを作成しておく必要があります。マッピングを作成するには、連合データベース関数を選択し、マッピングを作成するための DDL ステートメントへ実行依頼します。このステートメントは CREATE FUNCTION MAPPING と呼ばれます。

連合データベース関数は、既存の関数または関数テンプレート、あるいはユーザーが作成する関数または関数テンプレートでも構いません。 (*関数テンプレート* とは、実行可能コードがない部分関数です。) CREATE FUNCTION ステートメントを使用すると、関数または関数テンプレートを作成できます。

データ・ソース関数とその連合データベース関数は、以下の点で対応していなければなりません。

- 両方に同じ数の入力パラメーターがなければなりません。
- データ・ソース関数の入力パラメーターのデータ・タイプは、連合データベースに保管されている相手の関数の入力パラメーターのデータ・タイプと互換性がなければなりません。

CREATE FUNCTION MAPPING および CREATE FUNCTION ステートメントについての資料は、 *SQL 解説書* を参照してください。

関数を呼び出す際のオーバーヘッドの削減

連合サーバー関数からデータ・ソース関数へのマッピングのための DDL (CREATE FUNCTION MAPPING ステートメント) には、データ・ソース関数が呼び出される際に消費されると思われるオーバーヘッドの見積統計が含まれます。たとえば、データ・ソース関数を呼び出すのに必要な命令の数の見積もり、およびこの関数に渡される引き数のバイトごとに拡張される入出力の数の見積もりを指定できます。これらの見積もりはグローバル・カタログに保管されており、 SYSCAT.FUNCMAPOPTIONS 視点で参照できます。さらに、 DB2 関数 (関数テンプレートではない) がマッピングに参加する場合、この関数が呼び出される際に消費されるオーバーヘッドの見積もりがカタログに含まれます。この見積もりは、SYSCAT.FUNCTIONS 視点で参照できます。

マッピングが作成された後は、 DB2 関数を参照する分散要求を実行依頼できます。たとえば、DOLLAR という DB2 ユーザー定義関数を US_DOLLAR という Oracle ユーザー定義関数にマップした場合、要求は US_DOLLAR ではなく DOLLAR を指定することになります。要求を処理する際に、最適化プログラムは複数のアクセス戦略を評価します。この中には、DB2 関数の呼び出しのオーバーヘッドの見積もりを反映するものもあれば、データ・ソース関数の呼び出しのオーバーヘッドの見積もりを反映するものもあります。予想されるオーバーヘッドの量が最も少ない戦略が、使用される戦略となります。

消費されるオーバーヘッドの見積もりが変更される場合、グローバル・カタログにこの変更を記録できます。データ・ソース関数の新しい見積もりを記録するには、まず、関数マッピングを除去または使用不可にします (この方法については、 602ページの『関数マッピングの中断』を参照してください)。次に、 CREATE FUNCTION MAPPING ステートメントで新しい見積もりを指定し、そのステートメントでマッピングを再作成します。ステートメントを実行する際に、新しい見積もりが SYSCAT.FUNCTIONS カタログ視点に追加されます。 DB2 関数の見積もりの変更を記録するには、 SYSSTAT.FUNCTIONS カタログ視点を直接更新してください。

統計の見積もりは CREATE FUNCTION MAPPING ステートメントに指定します。それには、*関数マッピング・オプション* と呼ばれるパラメーターに、これらの見積もりを値として割り当てます。 602ページの表29 では、これらのオプションと値について説明します。

表 29. 関数マッピング・オプションとそれらの設定

オプション	有効な設定	デフォルト設定
ios_per_invoc	データ・ソース関数の呼び出しごとの入出力数の見積もり	‘0’
insts_per_invoc	データ・ソース関数の呼び出しごとに処理される命令の数の見積もり	‘450’
ios_per_argbyte	データ・ソース関数に渡される引き数セットのバイトごとに拡張される入出力の数の見積もり	‘0’
insts_per_argbyte	データ・ソース関数に渡される引き数セットのバイトごとに処理される命令の数の見積もり	‘0’
percent_argbytes	データ・ソース関数が実際に読み取る入力引き数バイトの平均パーセントの見積もり	‘100’
initial_ios	データ・ソース関数が呼び出される最初と最後に実行される入出力の数の見積もり	‘0’
initial_insts	データ・ソース関数が呼び出される最初と最後に処理される命令の数の見積もり	‘0’

DROP FUNCTION MAPPING ステートメント、 SYSCAT.FUNCTIONS および SYSSTAT.FUNCTIONS 視点、さらに SYSCAT.FUNCMAPOPTIONS 視点の詳細については、 *SQL 解説書* を参照してください。

CREATE FUNCTION MAPPING ステートメントでの関数名の指定

CREATE FUNCTION MAPPING ステートメントのコード化の方法は、マップするオブジェクトの名前が同じか違うかによって異なります。同じ名前の 2 つの関数 (または関数テンプレートと関数) の間でマッピングを作成する場合、この名前を *function-name* パラメーターに割り当てる必要があります。

一方、名前が異なる場合は次のようにします。

- 連合データベース関数または関数テンプレートの名前を、 *function-name* パラメーターに割り当てます。
- 『remote_name』 という関数マッピング・オプションを指定し、データ・ソース関数の名前をこのオプションに割り当てます。名前は、255 文字より少ない数でなければなりません。

関数マッピングの中断

関数マッピングの使用を中断したい場合には、以下のガイドラインに従ってください。

- このマッピングが SYSCAT.FUNCMAPPINGS カタログ視点にリストされている場合には、マッピングを削除します。これは、DROP FUNCTION MAPPING ステートメントを使用して実行します。

- SYSCAT.FUNCMAPPINGS 視点にリストされていないデフォルト・マッピングを中断するには、マッピングを使用不能にします。これは、『disable』という関数マッピング・オプションを ‘y’ (はい、この関数マッピングを使用不能にします) に設定すると、CREATE FUNCTION MAPPING ステートメントで実行できます。デフォルトは、‘n’ です。

パススルーを使用してデータ・ソースを直接照会する

パススルー という機能を使用して、各データ・ソースに固有のデータ・ソースを SQL で照会できます。この節では、以下の事柄を扱います。

- 連合サーバーおよびそれに関連するデータ・ソースが、パススルー・セッションでどのような SQL ステートメントを処理するかを説明します。
- パススルーを使用する際に注意すべき考慮事項および制約事項をリストします。

パススルー・セッションでの SQL 処理

SQL ステートメントが DB2 とデータ・ソースのどちらによって処理されるかは、以下の規則で指定されます。

- パススルー・セッションで静的ステートメントの処理依頼が行われると、そのステートメントは処理するため連合サーバーに送信されます。
- データ・ソースに対する SQL ステートメントの処理依頼をパススルー・セッションで行う場合には、そのステートメントをセッションで動的に準備して、セッションがまだ開いている間にステートメントを実行する必要があります。
  - SELECT ステートメントを実行依頼する場合には、 PREPARE ステートメントと一緒に準備し、それから OPEN、FETCH、および CLOSE ステートメントを使用して、照会結果にアクセスします。
  - SELECT 以外でサポートされているステートメントの場合、 2 つのオプションがあります。
    - PREPARE ステートメントを使ってサポートされているステートメントを準備し、 EXECUTE ステートメントを使ってそのステートメントを実行する。
    - EXECUTE IMMEDIATE ステートメントを使って、そのステートメントを準備して実行する。
- パススルー・セッション中に COMMIT または ROLLBACK コマンドを実行した場合、このコマンドは現行の作業単位 (UOW) を完了します。

考慮事項および制約事項

パススルーを使用する際には、いくつかの考慮事項および制約事項に注意する必要があります。一般的な性質のものもあれば、 Oracle データ・ソースだけに適用される事項もあります。

すべてのデータ・ソースにパススルーを使用する

以下の情報はすべてのデータ・ソースに適用されます。

- パススルー・セッションで準備したステートメントは、同一のパススルー・セッションで実行する必要があります。パススルー・セッションで準備したステートメントを、同一のパススルー・セッション以外で実行すると、失敗します (SQLSTATE 56098)。
- パススルーはデータ・ソースへの書き込みに使用できます。たとえば、表の行の挿入、更新、および削除に使用できます。しかし、パススルー・セッションでは、UPDATE および DELETE ステートメントで WHERE CURRENT OF 条件を使用できないことに注意してください。
- 1 つのアプリケーションで、異なるデータ・ソースに有効な SET PASSTHRU ステートメントを同時に複数持つことができます。アプリケーションが複数の SET PASSTHRU ステートメントを発行したとしても、パススルー・セッションは実際にはネストされません。連合サーバーは、あるデータ・ソースから別のデータ・ソースにアクセスするためにパススルーすることはありません。むしろ、サーバーはそれぞれのデータ・ソースに直接アクセスします。
- 複数のパススルー・セッションが同時にオープンしている場合には、それぞれのセッションで作業単位を終了するたびに必ず COMMIT を発行してください。すると、セッションを終了する必要があるときには、 SET PASSTHRU RESET ステートメントを 1 回実行するだけで、これを行えます。
- パススルー・セッションで SQL ステートメントに定義されるホスト変数は、 H*n* の形式を取ります。ここで、 H は大文字であり、*n* は固有の整数です。 *n* の値は、ゼロで始まる連続番号でなければなりません。
- 一度に複数のデータ・ソースはパススルーできません。
- パススルーはストアード・プロシージャー呼び出しをサポートしません。
- パススルーでは、SELECT INTO ステートメントはサポートされていません。

Oracle データ・ソースでパススルーを使用する

以下の情報は Oracle データ・ソースに適用されます。

- 以下の制約事項は、リモート・クライアントがコマンド行プロセッサー (CLP) からパススルー・モードで SELECT ステートメントを発行する際に適用されます。クライアント・コードが DB2 ユニバーサル・データベース バージョン 5 以前の DB2 アプリケーション開発クライアントである場合には、 SELECT は SQLCODE -30090 と理由コード 11 を出します。このエラーを避けるためには、リモート・クライアントがバージョン 5 またはそれ以上の DB2 アプリケーション開発クライアントを使用する必要があります。
- Oracle サーバーに対して発行される DDL ステートメントはすべて解析時に実行されるので、トランザクションのセマンティクスには従いません。操作は、終了時に Oracle によって自動的にコミットされます。ロールバックが発生しても、DDL はロールバックされません。

- 生データ・タイプから SELECT ステートメントを発行する際には、 RAWTOHEX 関数を使用して 16 進値を受け取ります。生データ・タイプへの INSERT を実行する場合には、16 進数表示を使用します。

第6部 言語に関する考慮事項

© Copyright IBM Corp. 1993, 2001

第20章 C および C++ でのプログラミング

C および C++ でのプログラミングに関する考慮事項. 609
C および C++ での言語制約事項 609
  C および C++ での 3 文字表記 609
  C++ のタイプ修飾に関する考慮事項 . . 610
C および C++ の入出力ファイル 610
C および C++ のインクルード・ファイル 611
  C および C++ のインクルード・ファイル 614
C および C++ での組み込み SQL ステートメント 615
C および C++ でのホスト変数 616
  C および C++ でのホスト変数の命名 . . 616
  C および C++ でのホスト変数の宣言 . . 618
  C および C++ での標識変数 622
  C または C++ でのグラフィック・ホスト変数宣言. 622
  C または C++ での LOB データ宣言 . . 625
  C または C++ における LOB ロケーター宣言 627
  C または C++ におけるファイル参照宣言 628
  C および C++ でのホスト変数の初期化 629
  C マクロ展開 629
  C および C++ でのホスト構造サポート 631
  C および C++ での標識表. 632
  C および C++ での Null 終了ストリング 633
  C および C++ でのポインター・データ・タイプ 635
  C および C++ でのクラス・データ・メンバーのホスト変数としての使用 636
  C および C++ での修飾およびメンバー演算子の使用 638
  C および C++ でのグラフィック・ホスト変数の処理 638
    C および C++ でのマルチバイト文字のエンコード 639
    C および C++ での wchar_t または sqldbchar データ・タイプの選択 . . . 639
    C および C++ での WCHARTYPE プリコンパイラー・オプション 640
  C および C++ での日本語または中国語 (繁体字) EUC、および UCS-2 に関する考慮事項 643
C および C++ でのサポートされている SQL データ・タイプ 645
  C および C++ における FOR BIT DATA 650
| C/C++ ストアード・プロシージャー、関数、
| およびメソッドのタイプ 650
C および C++ における SQLSTATE および SQLCODE 変数 652

C および C++ でのプログラミングに関する考慮事項

特定のホスト言語によるプログラミングの考慮事項について、以降の節で説明します。言語制限、ホスト言語別のインクルード・ファイル、組み込み SQL ステートメント、ホスト変数、およびサポートされるホスト変数のデータ・タイプについての情報が含まれています。

C および C++ での言語制約事項

以下の節では、C/C++ 言語の制限について説明します。

C および C++ での 3 文字表記

C または C++ 文字セットの文字の中には、すべてのキーボードで使用できないものもあります。これらの文字は、*3 文字表記* と呼ばれる一続きの 3 つの文字を使用して C

© Copyright IBM Corp. 1993, 2001

または C++ のソース・プログラムに入力することができます。 3 文字表記は SQL ステートメントでは認識されません。プリコンパイラーは、ホスト変数宣言内で以下の 3 文字表記を認識します。

3 文字表記	定義
??(	左大括弧 '['
??)	右大括弧 ']'
??<	左中括弧 '{'
??>	右中括弧 '}'

以下に示すその他の 3 文字表記は、C または C++ ソース・プログラムの別の場所で使用されることがあります。

3 文字表記	定義
??=	ハッシュ・マーク '#'
??/	円記号 '¥'
??'	脱字記号 '^'
??!	縦棒 '|'
??-	波形記号 '~'

C++ のタイプ修飾に関する考慮事項

C++ を使用してストアード・プロシージャーや UDF を作成する場合、次のようにしてそのプロシージャーや UDF を宣言することができます。

```
extern "C" ...procedure or function declaration...
```

`extern "C"` は、C++ コンパイラーが関数名をタイプ修飾するのを防ぎます。 この宣言の場合を除き、ストアード・プロシージャーを呼び出したり、CREATE FUNCTION ステートメントを発行する場合には、関数名のタイプ修飾も組み込む必要があります。

C および C++ の入出力ファイル

デフォルトには、入力ファイルに次のような拡張子を付けることができます。

.sqc サポートされているすべてのプラットフォーム上の C ファイル

.sqC UNIX プラットフォーム上の C++ ファイル

.sqx OS/2 および Windows 32 ビット・オペレーティング・システム上の C++ ファイル

デフォルトでは、対応するプリコンパイラーの出力ファイルは、以下の拡張子が付けられます。

.c サポートされているすべてのプラットフォーム上の C ファイル

.C UNIX プラットフォーム上の C++ ファイル

.cxx OS/2 および Windows 32 ビット・オペレーティング・システム上の C++ ファイル

OUTPUT プリコンパイル・オプションを使用することにより、出力修正後のソース・ファイルの名前とパスを無効にできます。 TARGET C または TARGET CPLUSPLUS プリコンパイル・オプションを使用する場合、入力ファイルに拡張子は必要ありません。

C および C++ のインクルード・ファイル

C および C++ でのホスト言語固有のインクルード・ファイル (ヘッダー・ファイル) には、ファイル拡張子の `.h` が付いています。アプリケーションで使用するためのインクルード・ファイルについて、以下で説明します。

SQL (sql.h)
このファイルには、バインダー、プリコンパイラー、およびエラー・メッセージ検索 API 用の言語固有プロトタイプが含まれています。また、システム定数も定義されています。

SQLADEF (sqladef.h)
プリコンパイル済みの C および C++ アプリケーションが使用する関数プロトタイプが含まれています。

SQLAPREP (sqlaprep.h)
このファイルには、独自のプリコンパイラーの作成に必要な定義が入っています。

SQLCA (sqlca.h)
このファイルは SQL 連絡域 (SQLCA) 構造を定義します。 SQLCA には、データベース・マネージャーが SQL ステートメントおよび API 呼び出しの実行に関するエラー情報をアプリケーションに提供するために使用する変数が含まれています。

SQLCLI (sqlcli.h)
コール・レベル・インターフェース (DB2 CLI) アプリケーションを作成するために必要な関数プロトタイプと定数が含まれています。このファイル内の関数は、X/Open コール・レベル・インターフェースおよび ODBC コア・レベルの両方に共通です。

SQLCLI1 (sqlcli1.h)
DB2 CLI でより拡張された機能を使用するコール・レベル・インターフェース (DB2 CLI) を作成するために必要な関数プロトタイプおよび定数が含まれています。このファイル内の関数の大部分は、 X/Open コール・レベル・インターフェースおよび ODBC レベル 1 に共通です。さらに、このファイルには X/Open 専用の関数および DB2 固有の関数も含まれています。

このファイルには、sqlcli.h と sqlext.h (ODBC レベル 2 API 定義) の両方が含まれます。

SQLCODES (sqlcodes.h)
このファイルは、SQLCA 構造の SQLCODE フィールドで使用する定数を定義します。

SQLDA (sqlda.h)
このファイルは SQL 記述子域 (SQLDA) 構造を定義します。 SQLDA はアプリケーションとデータベース・マネージャーとの間でデータをやりとりするために使用されます。

SQLEAU (sqleau.h)
このファイルには、DB2 セキュリティー監査 API に必要な定数および構造の定義が含まれています。これらの API を使用する場合は、プログラムにこのファイルを組み込む必要があります。このファイルにはまた、監査証跡レコード内のフィールドの定数およびキーワード値の定義も含まれています。これらの定義は、外部または取引先の監査証跡抽出プログラムで使用できます。

SQLENV (sqlenv.h)
このファイルは、データベース環境 API に対する言語固有の呼び出し、およびそれらのインターフェースの構造、定数、戻りコードを定義します。

SQLEXT (sqlext.h)
X/Open コール・レベル・インターフェースの仕様の一部ではないこれらの ODBC レベル 1 およびレベル 2 API の関数プロトタイプと定数が含まれており、 Microsoft 社の許可を得て使用します。

SQLE819A (sqle819a.h)
データベースのコード・ページが 819 (ISO ラテン 1) の場合、このシーケンスは、ホスト CCSID 500 (EBCDIC 各国対応) バイナリー照合に従って、FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQLE819B (sqle819b.h)
データベースのコード・ページが 819 (ISO ラテン 1) の場合、このシーケンスは、ホスト CCSID 037 (EBCDIC 米国英語) バイナリー照合に従って、FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQLE850A (sqle850a.h)
データベースのコード・ページが 850 (ASCII ラテン 1) の場合、このシーケンスは、ホスト CCSID 500 (EBCDIC 各国対応) バイナリー照合に従って、FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQLE850B (sqle850b.h)
データベースのコード・ページが 850 (ASCII ラテン 1) の場合、このシーケ

ンスは、ホスト CCSID 037 (EBCDIC 米国英語) バイナリー照合に従って、 FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQLE932A (sqle932a.h)
データベースのコード・ページが 932 (ASCII 日本語) の場合、このシーケンスは、ホスト CCSID 5035 (EBCDIC 日本語) バイナリー照合に従って、 FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQLE932B (sqle932b.h)
データベースのコード・ページが 932 (ASCII 日本語) の場合、このシーケンスは、ホスト CCSID 5026 (EBCDIC 日本語) バイナリー照合に従って、 FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQLJACB (sqljacb.h)
DB2 コネクト・インターフェースの定数、構造、および制御ブロックを定義します。

SQLMON (sqlmon.h)
このファイルは、データベース・システム・モニター API に対する言語固有の呼び出し、およびそれらのインターフェースの構造、定数、戻りコードを定義します。

SQLSTATE (sqlstate.h)
このファイルは、SQLCA 構造の SQLSTATE フィールドで使用する定数を定義します。

SQLSYSTM (sqlsystm.h)
このファイルには、データベース・マネージャー API およびデータ構造が用いるプラットフォーム固有の定義が含まれています。

SQLUDF (sqludf.h)
ユーザー定義関数 (UDF) を作成するための定数およびインターフェース構造を定義します。このファイルの詳細については、435ページの『UDF インクルード・ファイル: sqludf.h』を参照してください。

SQLUTIL (sqlutil.h)
このファイルは、ユーティリティー API に対する言語固有の呼び出し、およびそれらのインターフェースに必要な構造、定数、コードを定義します。

SQLUV (sqluv.h)
非同期ログ読み取り API および表のロード / アンロードを行うベンダーが使用する API の構造、定数、プロトタイプなどを定義します。

SQLUVEND (sqluvend.h)
記憶管理ベンダーが使用する API の構造、定数およびプロトタイプを定義します。

SQLXA (sqlxa.h)
X/Open XA インターフェースを使用するアプリケーションが使用する関数プロトタイプと定数が含まれます。

C および C++ のインクルード・ファイル

ファイルの組み込みには、EXEC SQL INCLUDE ステートメントを使用する方法と、#include マクロを使用する方法の 2 つがあります。プリコンパイラーは #include を無視し、 EXEC SQL INCLUDE ステートメントを使用して組み込まれたファイルのみを処理します。

`EXEC SQL INCLUDE` を使用して組み込んだファイルを検索する際に、 DB2 C プリコンパイラーはまず最初に現行ディレクトリーを検索し、次いで DB2INCLUDE 環境変数に指定されたディレクトリーを検索します。次の例を考えてみてください。

- `EXEC SQL INCLUDE payroll;`

  INCLUDE ステートメントに指定されたファイルが、上のように、単引用符 (') で囲まれていない場合、 C プリコンパイラーは各ディレクトリーで最初に `payroll.sqc` を検索し、次に `payroll.h` を検索します。 UNIX オペレーティング・システムの場合、C プリコンパイラーは、 `payroll.sqC`、`payroll.sqx`、 `payroll.hpp`、`payroll.h` の順で各ディレクトリーを検索します。 OS/2 または Windows 32 ビット・オペレーティング・システムの場合、 C++ プリコンパイラーは、 `payroll.sqx`、`payroll.hpp`、`payroll.h` の順で検索し、最後にそれがロックする各ディレクトリーを検索します。

- `EXEC SQL INCLUDE 'pay/payroll.h';`

  ファイル名が単引用符 (') で囲まれている場合、すでに説明したとおり、名前に拡張子は追加されません。

  単引用符 (') 内のファイル名に完全パスが含まれていない場合、 DB2INCLUDE の中身によってそのファイルの検索が行われ、何らかのパスが INCLUDE ファイル名に指定されます。たとえば、UNIX ベースのシステムでは、 DB2INCLUDE は '`/disk2:myfiles/c`' に設定され、 C/C++ プリコンパイラーは '`./pay/payroll.h`'、'`/disk2/pay/payroll.h`'、 '`./myfiles/c/pay/payroll.h`' の順に探索します。プリコンパイラー・メッセージには、実際にファイルが見つかったパスが表示されます。OS/2 および Windows ベースのプラットフォームでは、上の例のスラッシュを円記号 (¥) に置き換えます。

注: DB2INCLUDE の設定は、DB2 コマンド行プロセッサーによってキャッシュされます。何らかの CLP コマンドを実行した後に DB2INCLUDE の設定を変更する場合は、 TERMINATE コマンドを入力してから、データベースに接続し直し、あとは通常どおりプリコンパイルしてください。

プリコンパイラーは、コンパイラー・エラーを元のソースに戻して関連付けるために、出力ファイルに ANSI #line マクロを生成します。これにより、コンパイラーはプリコンパイラー出力ではなく、ソースまたは組み込まれたソース・ファイルのファイル名と行番号を使用してエラーを報告することができます。

PREPROCESSOR オプションを指定する場合は、プリコンパイラーにより生成されるすべての #line マクロは、外部 C プリプロセッサーからプリプロセスされたファイルを参照します。 PREPROCESSOR オプションの詳細については、 629ページの『C マクロ展開』を参照してください。

ソース・コードをオブジェクト・コードに関連付けるデバッガーやその他のツールの中には、#line マクロを使用して常に正常に作動するわけではないものもあります。使用したいツールが期待どおりに動作しない場合は、プリコンパイル時に (DB2 PREP と共に使用される) NOLINEMACRO オプションを使用してください。これにより、#line マクロは生成されなくなります。

C および C++ での組み込み SQL ステートメント

組み込み SQL ステートメントは次の 3 つのエレメントからなります。

エレメント	正しい構文
ステートメント初期化指定子	EXEC SQL
ステートメント・ストリング	任意の有効な SQL ステートメント
ステートメント終了文字	セミコロン (;)

以下に例を示します。

```
EXEC SQL SELECT col INTO :hostvar FROM table;
```

組み込み SQL ステートメントには、以下の規則が適用されます。

- SQL ステートメント・ストリングは、キーワード対または別の行として同じ行で開始することができる。ステートメント・ストリングは、複数行にわたる長さであってもかまいません。ただし、対になった EXEC SQL キーワードを複数行に分割してはなりません。
- SQL ステートメント終了文字を使用しなければならない。使用しない場合、プリコンパイラーはアプリケーション内の次の終了文字まで処理を継続します。これにより、不確定のエラーが起こる恐れがあります。

  C/C++ 注釈は、ステートメント開始記号の前、またはステートメント終了文字の後に入れることができます。
- 複数の SQL ステートメントと C/C++ ステートメントは同じ行に置くことができる。以下に例を示します。

  ```
  EXEC SQL OPEN c1; if (SQLCODE >= 0) EXEC SQL FETCH c1 INTO :hv;
  ```
- SQL プリコンパイラーは、CR/LF および TAB を引用符付きストリングにそのまま残す。
- SQL 注釈は、組み込み SQL ステートメントの一部となっている行であればどこでも使用することができる。この注釈は、動的に実行するステートメントでは使用できません。 SQL 注釈の形式は、ダブル・ダッシュ (--) の後に 0 個以上の文字ストリン

グが続き、行末で終了します。 SQL 注釈は SQL ステートメント終了文字の後に置かないでください。これを行うと、注釈が C/C++ 言語の一部のように見えるため、コンパイル・エラーの原因となるからです。

注釈は、ブランクを使用できる静的ステートメント・ストリング内であればどこでも使用することができます。 C/C++ 注釈区切り文字の /* */、または SQL 注釈記号の (--) を使用してください。 //-style C++ 注釈は静的 SQL ステートメントでは使用できませんが、プログラム内の他の場所では使用できます。プリコンパイラーは、SQL ステートメントを処理する前に注釈を削除します。動的 SQL ステートメントでは、C および C++ の注釈区切り文字である /* */ または // を使用することは**できません**。ただし、プログラム内の他の場所では使用できます。

- C および C++ アプリケーションでは、 SQL ストリング・リテラルや区切り ID は次の行にわたって続けることができる。このことを行うには、中断させたい行の末尾に円記号 (¥) を使用します。以下に例を示します。

```
EXEC SQL SELECT "NA¥
ME" INTO :n FROM staff WHERE name='Sa¥
nders';
```

  改行文字 (復帰や改行など) はいずれも、ストリングまたは区切り ID には含まれません。

- 行末文字およびタブ文字などの空白文字の置換は、次のように行われる。
  - 引用符の外 (ただし、SQL ステートメント内) では、行末文字または TAB 文字は単一スペースと置き換えられる。
  - 引用符内では、ストリングが C プログラムに適合した形で継続されていれば、行末文字は消去される。 TAB は修正されません。

  行末および TAB に使用される実際の文字は、プラットフォームごとに異なります。たとえば、OS/2 は復帰 / 改行を行末に使用するのに対し、UNIX ベースのシステムは改行のみを使用します。

C および C++ でのホスト変数

ホスト変数は、SQL ステートメント内で参照される C または C++ の言語変数です。これにより、アプリケーションは入力データをデータベース・マネージャーに渡し、またデータベース・マネージャーから出力データを受け取ることができます。アプリケーションのプリコンパイルが行われると、コンパイラーはホスト変数をその他の C/C++ 変数と同様に使用します。ホスト変数の命名、宣言、および使用は、以下の節で述べる規則に従って行ってください。

C および C++ でのホスト変数の命名

SQL プリコンパイラーは、宣言された名前によってホスト変数を識別します。以下の規則が適用されます。

- 変数名の長さは最大 255 文字。

- ホスト変数名は、システムの予約語である SQL、sql、DB2、および db2 以外の接頭部で開始する。以下に例を示します。

```
EXEC SQL BEGIN DECLARE SECTION;
  char  varsql;     /* allowed */
  char  sqlvar;     /* not allowed */
  char  SQL_VAR;    /* not allowed */
EXEC SQL END DECLARE SECTION;
```

- プリコンパイラーは、ホスト変数名をモジュールに対してグローバルであると見なす。ただし、これは、ホスト変数をグローバル変数として宣言しなければならないという意味ではありません。ホスト変数を関数内でローカル変数として宣言しても全く問題ありません。たとえば、次ページのコーディングは正しいものとして処理されます。

```
void f1(int i)
{
EXEC SQL BEGIN DECLARE SECTION;
  short host_var_1;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT COL1 INTO :host_var_1 from TBL1;
}
void f2(int i)
{
EXEC SQL BEGIN DECLARE SECTION;
  short host_var_2;
EXEC SQL END DECLARE SECTION;
EXEC SQL INSERT INTO TBL1 VALUES (:host_var_2);
}
```

複数のローカル・ホスト変数に同じ名前を付けることも可能です。ただしこの場合、それらの変数がすべて同じタイプかつ同じサイズであることが条件です。このことを行うには、そのようなホスト変数の最初の出現箇所を BEGIN DECLARE SECTION ステートメントと END DECLARE SECTION ステートメントの間でプリコンパイラーに宣言し、残りの変数の宣言については宣言セクション外でそのままにしておきます。以下のコーディング例は、このことを示したものです。

```
void f3(int i)
{
EXEC SQL BEGIN DECLARE SECTION;
  char host_var_3[25];
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT COL2 INTO :host_var_3 FROM TBL2;
}
void f4(int i)
{
char host_var_3[25];
EXEC SQL INSERT INTO TBL2 VALUES (:host_var_3);
}
```

f3 と f4 は同じモジュールにあり、 host_var_3 はどちらの関数でも同じタイプかつ長さなので、プリコンパイラーへの宣言は 1 回で済みます。

C および C++ でのホスト変数の宣言

ホスト変数宣言の識別には、SQL の宣言セクションを使用しなければなりません。これは、それ以降の SQL ステートメントで参照が可能なホスト変数をプリコンパイラーに知らせます。

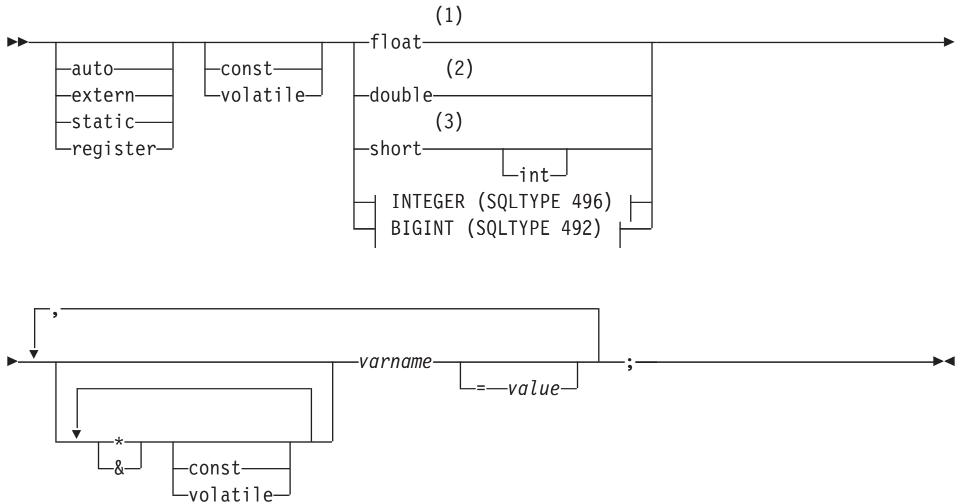
C/C++ プリコンパイラーは、有効な C または C++ 宣言のサブセットのみを有効なホスト変数宣言として認識します。これらの宣言は、数値変数または文字変数のいずれかを宣言します。ホスト変数のタイプとして typedef は使用できません。ホスト変数は、グループ化して単一のホスト構造にすることができます。ホスト構造についての詳細は、 631ページの『C および C++ でのホスト構造サポート』を参照してください。C++ クラスのデータ・メンバーは、ホスト変数として宣言できます。クラスの詳細については、636ページの『C および C++ でのクラス・データ・メンバーのホスト変数としての使用』を参照してください。

数値ホスト変数は、数値の SQL 入出力値に対する入出力値として使用することができます。文字ホスト変数は任意の文字、日付、時間またはタイム・スタンプの SQL 入出力値に対する入出力値として使用できます。アプリケーションでは、出力変数が受け取る値を含めることのできる長さを持つようにしなければなりません。

構造型のホスト変数の宣言方法についての詳細は、 363ページの『構造型ホスト変数の宣言』を参照してください。

619ページのC または C++ における数値ホスト変数の構文は、C または C++ における数値ホスト変数の宣言構文を示します。

C または C++ における数値ホスト変数の構文



INTEGER (SQLTYPE 496)

BIGINT (SQLTYPE 492)

注:

1 REAL (SQLTYPE 480)、長さ 4

2 DOUBLE (SQLTYPE 480)、長さ 8

3 SMALLINT (SQLTYPE 500)

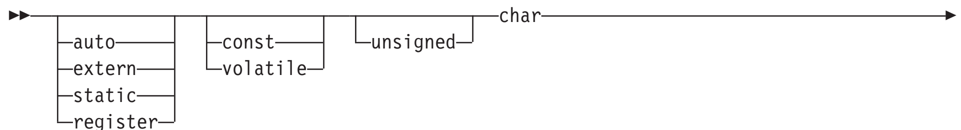
4 アプリケーションの移行性を最大限にするには、 INTEGER および BIGINT ホスト変数でそれぞれ sqlint32 および sqlint64 を使用してください。 デフォルトでは、 long のホスト変数を使用すると、 long が 64 ビットである 64 BIT UNIX

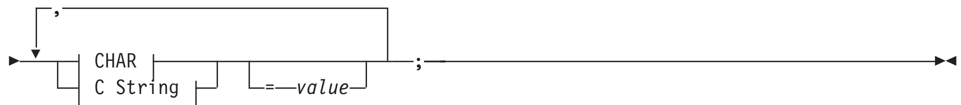
などでプリコンパイル・エラー SQL0402 が起きます。 PREP オプション LONGERROR NO を使用して、 DB2 が long 変数を受け入れ可能なホスト変数型として認めるようにしてください。 そして、それらを BIGINT 変数として扱ってください。

5 アプリケーションの移行性を最大限にするには、 INTEGER および BIGINT ホスト変数でそれぞれ sqlint32 および sqlint64 を使用してください。 BIGINT データ・タイプを使用するには、 プラットフォームで 64 ビットの整数値がサポートされていなければなりません。 デフォルトでは、 long のホスト変数を使用すると、 long が 64 ビットである 64 BIT UNIX などでプリコンパイル・エラー SQL0402 が起きます。 PREP オプション LONGERROR NO を使用して、 DB2 が long 変数を受け入れ可能なホスト変数型として認めるようにしてください。 そして、それらを BIGINT 変数として扱ってください。

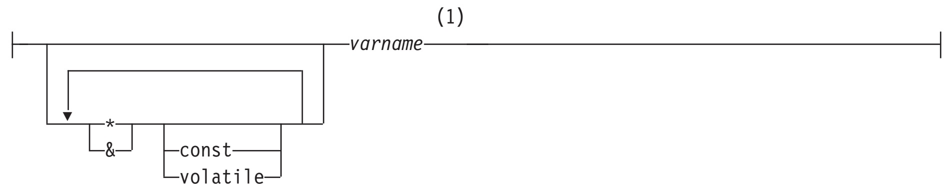
書式 1: C/C++ における固定および NULL 終了文字ホスト変数の構文は、C または C++ における固定および NULL 終了文字ホスト変数の宣言構文を示します。

書式 1: C/C++ における固定および NULL 終了文字ホスト変数の構文

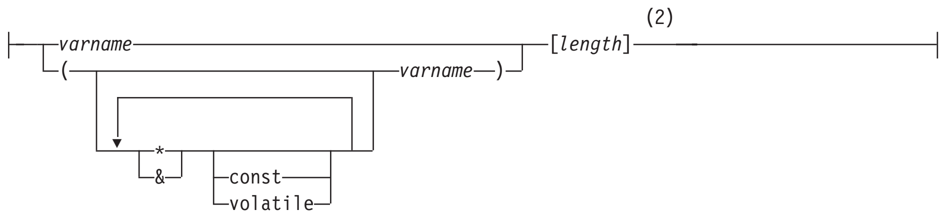




CHAR



C String



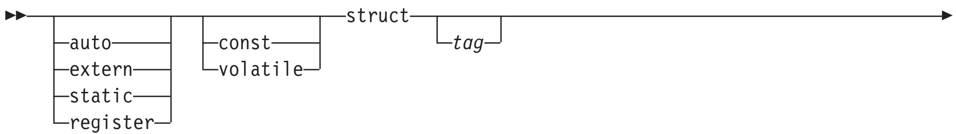
注:

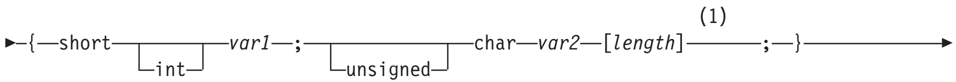
1 CHAR (SQLTYPE 452)、長さ 1

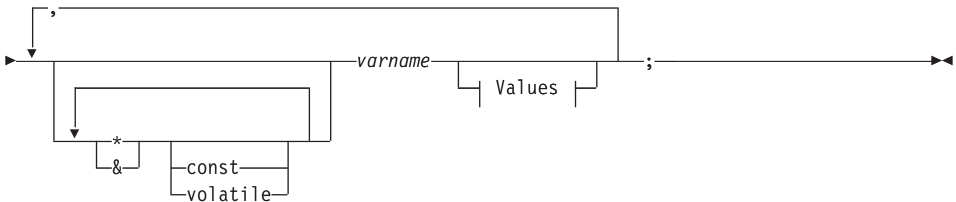
2 NULL 終了 C ストリング (SQLTYPE 460); 長さは任意の有効な定数式

書式 2: C/C++ における可変長文字ホスト変数の構文は、C または C++ における可変長文字ホスト変数の宣言構文を示します。

書式 2: C/C++ における可変長文字ホスト変数の構文







Values

注:

1 書式 2 では、length は任意の有効な定数式。評価後の値で、ホスト変数が VARCHAR (SQLTYPE 448) または LONG VARCHAR (SQLTYPE 456) のどちらであるかが判別されます。

可変長文字ホスト変数に関する考慮事項:

1. データベース・マネージャーは、可能な場合には必ず文字データを**書式 1** または**書式 2** に変換するが、**書式 1** が列タイプ CHAR または VARCHAR に対応するのに対し、**書式 2** は列タイプ VARCHAR および LONG VARCHAR に対応する。
2. **書式 1** を長さ指定子の *[n]* と共に使用した場合、評価後の長さ指定子の値は 32672 より大きくてはならず、変数に含めるストリングは NULL で終わらなければならない。
3. **書式 2** を使用する場合は、評価後の長さ指定子の値は 32 700 より小さくなくてはならない。
4. **書式 2** では、 *var1* および *var2* は単純変数参照 (演算子ではない) でなければならず、ホスト変数として使用することはできない (*varname* がホスト変数)。
5. *varname* を単純変数とすることができるか、または **varname* などの演算子を含むことができる。詳細については、635ページの『C および C++ でのポインター・データ・タイプ』を参照してください。
6. プリコンパイラーはすべてのホスト変数の SQLTYPE および SQLLEN を判別する。ホスト変数が SQL ステートメント内に標識変数とともにある場合、そのステートメントの持続期間中は、 SQLTYPE は基本の SQLTYPE プラス 1 となるように割り当てられます。
7. プリコンパイラーは、C または C++ において構文的に無効であるものも宣言できる場合がある。特定の宣言構文に疑問がある場合には、ご使用のコンパイラーに関する資料をご覧ください。

C および C++ での標識変数

標識変数のデータ・タイプは `short` と宣言してください。

C または C++ でのグラフィック・ホスト変数宣言

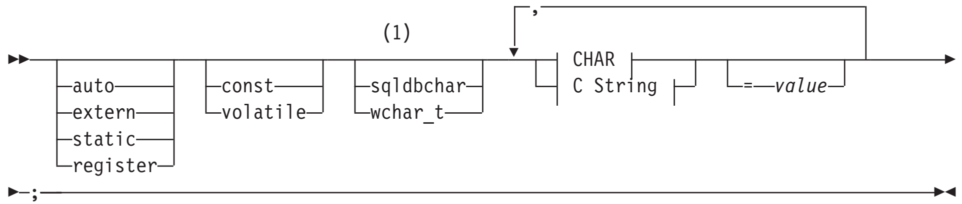
グラフィック・ホスト変数の宣言書式は、以下の 3 つのうちのいずれかです。

- 単純グラフィック書式
- NULL 終了グラフィック書式
- VARGRAPHIC 構造書式

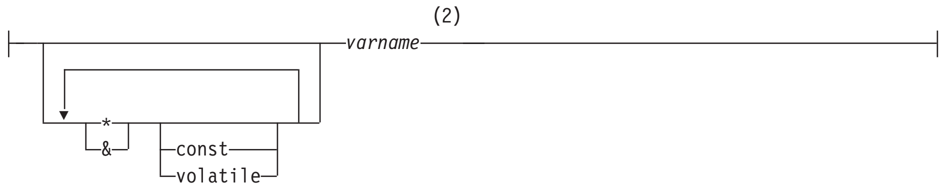
グラフィック・ホスト変数の詳細については、638ページの『C および C++ でのグラフィック・ホスト変数の処理』を参照してください。

623ページのグラフィック宣言の構文 (単純グラフィック書式および NULL 終了グラフィック書式) は、単純グラフィック書式および NULL 終了グラフィック書式を用いるグラフィック・ホスト変数の宣言構文を示します。

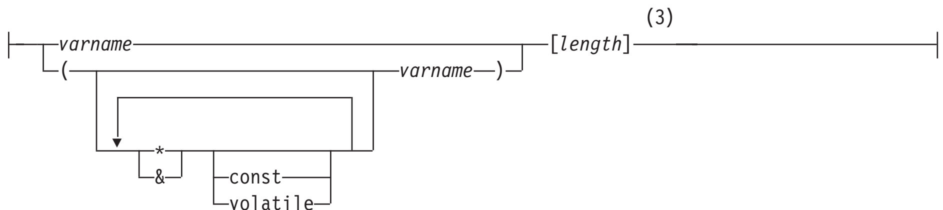
グラフィック宣言の構文 (単純グラフィック書式および NULL 終了グラフィック書式)



CHAR



C String



注:

1 2 つのグラフィック・タイプのどちらを使用するかを判別するための基準については、 639ページの『C および C++ での wchar_t または sqldbchar データ・タイプの選択』を参照してください。

2 GRAPHIC (SQLTYPE 468)、長さ 1

3 NULL 終了グラフィック・ストリング (SQLTYPE 400)

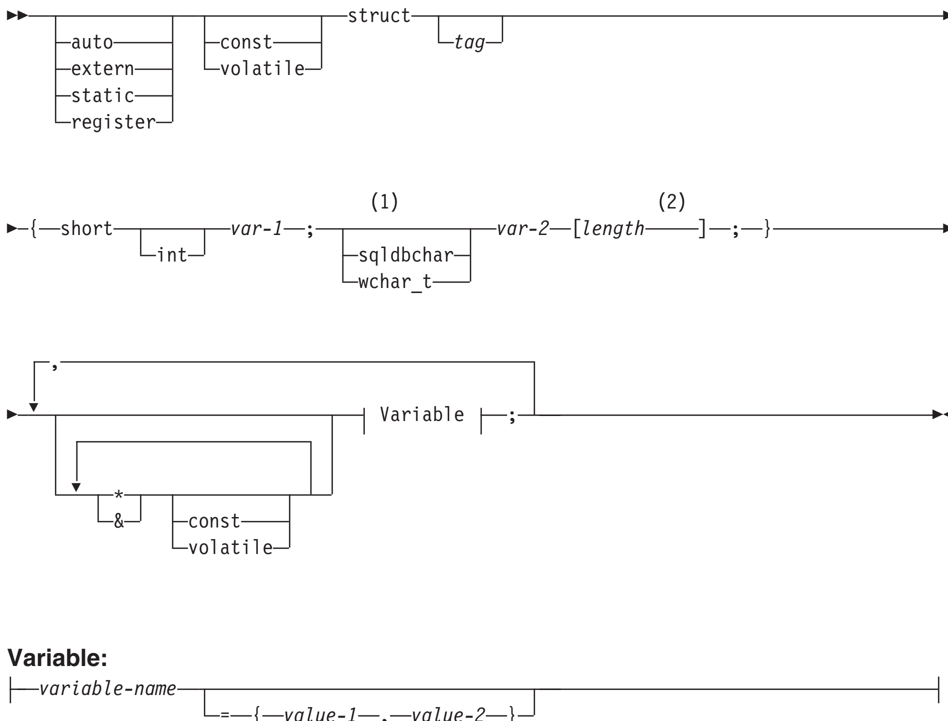
グラフィック・ホスト変数に関する考慮事項:

1. 単純グラフィック書式では、SQLTYPE が 468 または 469 で長さ 1 の固定長のグラフィック・ストリング・ホスト変数が宣言される。
2. *value* は初期化指定子である。 WCHARTYPE CONVERT プリコンパイラー・オプションを使用している場合は、ワイド・キャラクターのストリング・リテラル (L-リテラル) を使用してください。

3. *length* は任意の有効な定数式にすることができる。評価後の値は、1 ～ 16 336 (VARGRAPHIC の最大長) の範囲でなければなりません。
4. NULL 終了グラフィック・ストリングは、標準レベルのプリコンパイラー・オプションの値に基づいてさまざまに処理されます。詳細については、633ページの『C および C++ での Null 終了ストリング』を参照してください。

グラフィック宣言の構文 (VARGRAPHIC 構造書式) は、VARGRAPHIC 構造書式を用いるグラフィック・ホスト変数の宣言構文を示します。

グラフィック宣言の構文 (VARGRAPHIC 構造書式)



Variable:

```
├──variable-name──┬──────────────────────────────────┬──────────────┤
                  └─=──{──value-1──,──value-2──}──┘
```

注:

1 2 つのグラフィック・タイプのどちらを使用するかを判別するための基準については、639ページの『C および C++ での wchar_t または sqldbchar データ・タイプの選択』を参照してください。

2 length は、任意の有効な定数式。 評価後の値で、 ホスト変数が VARGRAPHIC (SQLTYPE 464) または LONG VARGRAPHIC (SQLTYPE 472) のどちらであるかが判別されます。 length の値は、1 ～ 16350 (LONG VARGRAPHIC の最大長) の範囲でなければなりません。

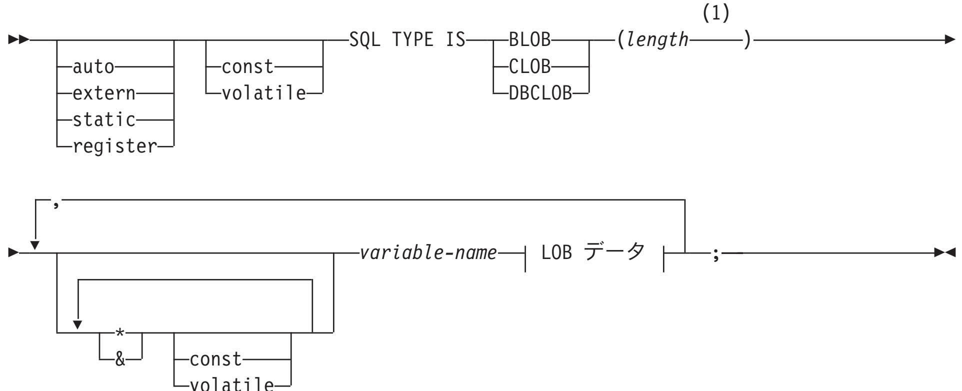
グラフィック宣言 (VARGRAPHIC 構造書式) に関する考慮事項:

1. *var-1* および *var-2* は単純変数参照 (演算子ではない) でなければならず、ホスト変数として使用することはできない。
2. *value-1* および *value-2* は、 *var-1* と *var-2* に対する初期化指定子である。 WCHARTYPE CONVERT プリコンパイラー・オプションを使用している場合、 *value-1* は整数でなければならず、 *value-2* はワイド・キャラクター・ストリング・リテラル (L-リテラル) を使用してください。
3. struct *tag* は他のデータ域を定義するために使用できるが、それ自体はホスト変数としては使用できない。

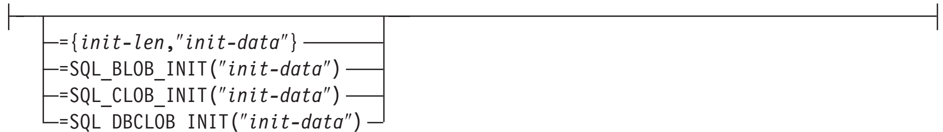
C または C++ での LOB データ宣言

C/C++ におけるラージ・オブジェクト (LOB) ホスト変数の構文は、C または C++ におけるラージ・オブジェクト (LOB) ホスト変数の宣言構文を示します。

C/C++ におけるラージ・オブジェクト (LOB) ホスト変数の構文



LOB データ



注:

1 length は、任意の有効な定数式。これには定数 K、M、または G を使用できる。 BLOB および CLOB の評価後の length 値は、 1 <= length <= 2 147 483 647 でなければなりません。 DBCLOB の評価後の length 値は 1 <= length <= 1 073 741 823 でなければなりません。

LOB ホスト変数に関する考慮事項:

1. 関数に渡される LOB タイプのホスト変数に対してタイプ検査と関数分解を実行できるように、 3 タイプの LOB を区別するための SQL TYPE IS 文節が必要である。
2. SQL TYPE IS、BLOB、CLOB、DBCLOB、K、M、G は、大文字と小文字が混在してもかまわない。
3. 初期化ストリング *"init-data"* に許可される最大長は、ストリング区切り文字を含めて 32702 バイトである (プリコンパイラー内の C/C++ ストリングの限界と同じ)。
4. 初期化長である *init-len* は、数値の定数でなければならない (すなわち、K、M、G は使用できない)。
5. LOB の長さを指定しなければならない。すなわち、次の宣言は無効です。

   ```
   SQL TYPE IS BLOB my_blob;
   ```
6. LOB を宣言内で初期化しないと、プリコンパイラーで生成されたコード内での初期化は行われない。
7. DBCLOB を初期化する場合、ユーザーは、ストリングに 'L' (ワイド・キャラクター・ストリングを表す) という接頭部を付けること。

   注: ワイド・キャラクター・リテラル、たとえば `L"Hello"` は、 WCHARTYPE CONVERT プリコンパイル・オプションを選択した場合に、プリコンパイル済みプログラムでのみ使用すべきである。
8. プリコンパイラーは、ホスト変数のタイプをキャストするために使用できる構造タグを生成する。

BLOB の例:

宣言:

```
static Sql Type is Blob(2M) my_blob=SQL_BLOB_INIT("mydata");
```

この結果、以下の構造が生成されます。

```
static struct my_blob_t {
      sqluint32        length;
      char             data[2097152];
 } my_blob=SQL_BLOB_INIT("mydata");
```

CLOB の例:

宣言:

```
volatile sql type is clob(125m) *var1, var2 = {10, "data5data5"};
```

この結果、以下の構造が生成されます。

```
volatile struct var1_t {
     sqluint32        length;
     char             data[131072000];
} * var1, var2 = {10, "data5data5"};
```

DBCLOB の例:

宣言:

```
SQL TYPE IS DBCLOB(30000) my_dbclob1;
```

WCHARTYPE NOCONVERT オプション指定でプリコンパイルされ、その結果、以下の構造が生成されます。

```
struct my_dbclob1_t {
     sqluint32        length;
     sqldbchar        data[30000];
} my_dbclob1;
```

宣言:

```
SQL TYPE IS DBCLOB(30000) my_dbclob2 = SQL_DBCLOB_INIT(L"mydbdata");
```

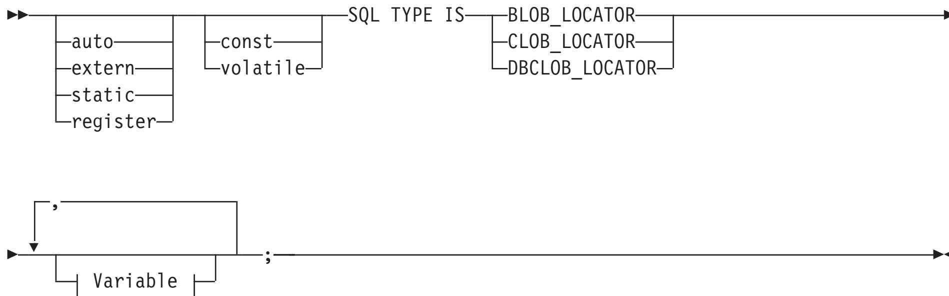
WCHARTYPE CONVERT オプション指定でプリコンパイルされ、その結果、以下の構造が生成されます。

```
struct my_dbclob2_t {
     sqluint32        length;
     wchar_t          data[30000];
} my_dbclob2 = SQL_DBCLOB_INIT(L"mydbdata");
```

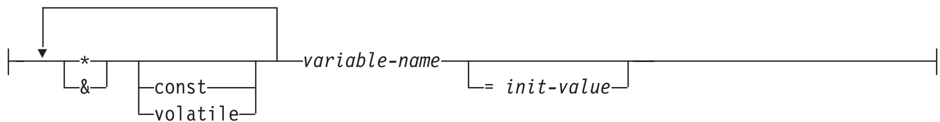
C または C++ における LOB ロケーター宣言

C/C++ におけるラージ・オブジェクト (LOB) ロケーター・ホスト変数の構文 は、C または C++ におけるラージ・オブジェクト (LOB) ロケーター・ホスト変数の宣言構文を示します。

C/C++ におけるラージ・オブジェクト (LOB) ロケーター・ホスト変数の構文



Variable



LOB ロケーター・ホスト変数に関する考慮事項:

1. SQL TYPE IS、BLOB-LOCATOR、CLOB-LOCATOR、DBCLOB-LOCATOR は、大文字、小文字、またはその混合のいずれでもかまわない。
2. *init-value* により、ポインターの初期化およびロケーター変数の参照ができる。他のタイプの初期化は、無意味となる。

CLOB ロケーターの例 (他のタイプの LOB ロケーターの場合も同様):

宣言:

```
SQL TYPE IS CLOB_LOCATOR my_locator;
```

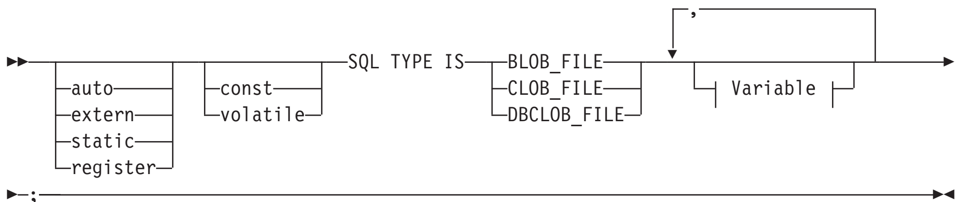
この結果、以下の宣言が生成されます。

```
sqlint32 my_locator;
```

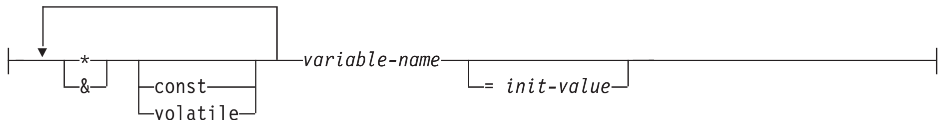
C または C++ におけるファイル参照宣言

C/C++ におけるファイル参照ホスト変数の構文は、C または C++ におけるファイル参照ホスト変数の宣言構文を示します。

C/C++ におけるファイル参照ホスト変数の構文



Variable



注:

- SQL TYPE IS、BLOB-FILE、CLOB-FILE、DBCLOB-FILE は、大文字、小文字、またはその混合のいずれでもかまわない。

CLOB ファイル参照の例 (その他の LOB ファイル参照の型宣言も同様):

宣言:

```
static volatile SQL TYPE IS BLOB_FILE my_file;
```

この結果、以下の構造が生成されます。

```
static volatile struct {
     sqluint32        name_length;
     sqluint32        data_length;
     sqluint32        file_options;
               char    name[255];
} my_file;
```

C および C++ でのホスト変数の初期化

C++ の宣言セクションでは、括弧を使用してホスト変数を初期化することはできません。次に、宣言セクション内での初期化の正しい方法と誤った方法の例を示します。

```
EXEC SQL BEGIN DECLARE SECTION;
  short my_short_2 = 5;       /* correct   */
  short my_short_1(5);        /* incorrect */
EXEC SQL END DECLARE SECTION;
```

C マクロ展開

C/C++ プリコンパイラーは、宣言セクション内の宣言で使用された C マクロを直接処理できません。代わりに、まず、外部 C プリプロセッサーでソース・ファイルをプリプロセスしなければなりません。これを実行するには、PREPROCESSOR オプションを使って、C プリプロセッサーを起動するためのコマンドをプリコンパイラーに指定します。

PREPROCESSOR オプションを指定すると、プリコンパイラーはまず、 SQL INCLUDE ステートメントで参照されているすべてのファイルの内容をソース・ファイルに結合させることによって、すべての SQL INCLUDE ステートメントを処理します。次にプリコンパイラーは、修正したソース・ファイルを入力として指定するコマンドを使用して、外部 C プリプロセッサーを起動します。プリプロセス済みのファイル (拡張子 ″.i″ によってプリコンパイラーは識別) は、プリコンパイルの残りのプロセスでの新しいソース・ファイルとして使用されます。

プリコンパイラーにより生成された任意の #line マクロは、元のソース・ファイルを参照することはありません。代わりに、プリプロセスされたファイルを参照します。コンパイラー・エラーを元のソース・ファイルに関連付けるには、プリプロセスされたファイルに注釈を保存します。これにより、ヘッダー・ファイルを含むオリジナル・ソー

ス・ファイルのあらゆるセクションを位置指定できます。注釈を保存するオプションは、通常、C プリプロセッサーで使用でき、 PREPROCESSOR オプションを使用して、指定するコマンドにオプションを含められます。 C プリプロセッサーには、任意の #line マクロ自体を出力させてはなりません。これには、プリコンパイラーにより生成されたものが誤って混在している可能性があるためです。

マクロ展開の使用上の注意:

1. PREPROCESSOR オプションを使用して指定するコマンドには、すべての望むオプションを含めることができますが、入力ファイルの名前を含めることはできません。たとえば、AIX 上の IBM C には、次のオプションを使用できます。

   ```
   xlC -P -DMYMACRO=1
   ```

2. プリコンパイラーは、このコマンドによって、拡張子 .i の付いたプリプロセス済みのファイルの生成を予期します。ただし、プリプロセス済みのファイルを生成するためにリダイレクトを使用することはできません。たとえば、以下のオプションを使用してプリプロセス済みファイルを生成することは**できません**。

   ```
   xlC -E > x.i
   ```

3. 外部 C プリプロセッサーが検出したエラーは、元のソース・ファイルに対応する名前に拡張子 .err を付けたファイルにレポートされます。

たとえば、以下のようにソース・コード内でマクロ展開を使用することができます。

```
#define SIZE 3

EXEC SQL BEGIN DECLARE SECTION;
  char a[SIZE+1];
  char b[(SIZE+1)*3];
  struct
  {
    short length;
    char  data[SIZE*6];
  } m;
  SQL TYPE IS BLOB(SIZE+1) x;
  SQL TYPE IS CLOB((SIZE+2)*3) y;
  SQL TYPE IS DBCLOB(SIZE*2K) z;
EXEC SQL END DECLARE SECTION;
```

PREPROCESSOR オプションを使用した後は、上記の宣言は以下のように解決します。

```
EXEC SQL BEGIN DECLARE SECTION;
  char a[4];
  char b[12];
  struct
  {
    short length;
    char  data[18];
  } m;
  SQL TYPE IS BLOB(4) x;
  SQL TYPE IS CLOB(15) y;
  SQL TYPE IS DBCLOB(6144) z;
EXEC SQL END DECLARE SECTION;
```

C および C++ でのホスト構造サポート

ホスト構造サポートを使用すると、C/C++ プリコンパイラーは、ホスト変数を単一のホスト構造にグループ化することができます。これにより、SQL ステートメントで同じセットのホスト変数を参照するのが簡単になります。たとえば、以下のホスト構造は、SAMPLE データベース内の STAFF 表のいくつかの列へのアクセスに使用できます。

```
struct tag
   {
     short id;
     struct
     {
       short length;
       char  data[10];
     } name;
     struct
     {
       short   years;
       double salary;
     } info;
   } staff_record;
```

ホスト構造のフィールドは、有効な任意のホスト変数タイプにすることができます。これらには、すべての数字、文字、およびラージ・オブジェクト・タイプが含まれます。ネストされるホスト構造は、25 レベルまでサポートされます。上の例では、フィールド info は副構造であるのに対し、フィールド name は副構造ではなく、VARCHAR フィールドを示しています。同じ原則は、LONG VARCHAR、VARGRAPHIC および LONG VARGRAPHIC にも当てはまります。ホスト構造へのポインターもサポートされます。

SQL ステートメントでホスト構造にグループ化されるホスト変数を参照するには、以下の 2 つの方法があります。

1. SQL ステートメントでホスト構造名を参照する。

   ```
   EXEC SQL SELECT id, name, years, salary
        INTO :staff_record
        FROM staff
        WHERE id = 10;
   ```

   プリコンパイラーは staff_record の参照を、ホスト構造内で宣言されたすべてのフィールドをコンマで区切ったリストに変換します。他のホスト変数またはフィールドとの重複を避けるために、それぞれのフィールドは、すべてのレベルのホスト構造名で修飾されます。これは以下の使用法と同じです。

2. SQL ステートメントで完全修飾ホスト変数名を参照する。

   ```
   EXEC SQL SELECT id, name, years, salary
        INTO :staff_record.id, :staff_record.name,
             :staff_record.info.years, :staff_record.info.salary
        FROM staff
        WHERE id = 10;
   ```

同じ名前のホスト変数がない場合でも、フィールド名を参照する際には完全修飾しなければなりません。修飾された副構造も参照できます。上の例では、:staff_record.info.years, :staff_record.info.salary を、:staff_record.info に置換することができます。

ホスト構造への参照 (1 番目の例) は、コンマで区切ったフィールドのリストと等しいため、このタイプの参照はエラーとなる場合があります。以下に例を示します。

```
EXEC SQL DELETE FROM :staff_record;
```

ここでの DELETE ステートメントは、1 バイト文字ベースの変数を予期しています。代わりにホスト構造を指定すると、ステートメントはプリコンパイル時エラーの結果になる可能性があります。

```
SQL0087N  Host variable "staff_record" is a structure used where structure
references are not permitted.
```

SQL0087N エラーの原因となる可能性があるホスト構造のこの他の使用には、PREPARE、EXECUTE IMMEDIATE、CALL、標識変数、および SQLDA 参照などがあります。このような状況では、個々のフィールドへの参照と同じように (2 番目の例)、フィールドを 1 つしか持たないホスト構造が許可されます。

C および C++ での標識表

インディケーター表は、ホスト構造で使用される標識変数の集合です。これは、短整数の配列として宣言しなければなりません。以下に例を示します。

```
short ind_tab[10];
```

上の例は、エレメントが 10 個のインディケーター表を宣言します。以下に、これを SQL ステートメントで使用する方法を示します。

```
EXEC SQL SELECT id, name, years, salary
     INTO :staff_record INDICATOR :ind_tab
     FROM staff
     WHERE id = 10;
```

以下の表では、それぞれのホスト構造フィールドとそれに対応する標識変数をリストしています。

staff_record.id
: ind_tab[0]

staff_record.name
: ind_tab[1]

staff_record.info.years
: ind_tab[2]

staff_record.info.salary
: ind_tab[3]

注: インディケーター表エレメント、たとえば ind_tab[1] は、 SQL ステートメントで個々に参照することはできません。キーワード INDICATOR はオプションです。構造化フィールドとインディケーターの数が一致している必要はありません。余分

のインディケーターが未使用だったり、インディケーターが割り当てられていない余分のフィールドがあってもかまいません。

インディケーター表の代わりにスカラー標識変数を使用して、ホスト構造の最初のフィールドにインディケーターを提供することもできます。これは、1 つのエレメントだけのインディケーター表を持つことと同じです。以下に例を示します。

```
short scalar_ind;

EXEC SQL SELECT id, name, years, salary
          INTO :staff_record INDICATOR :scalar_ind
          FROM staff
          WHERE id = 10;
```

ホスト構造の代わりにホスト変数を指定してインディケーター表を指定すると、インディケーター表の最初のエレメント、たとえば ind_tab[0] しか使用されません。

```
EXEC SQL SELECT id
          INTO :staff_record.id INDICATOR :ind_tab
          FROM staff
          WHERE id = 10;
```

短整数の配列がホスト構造内で宣言される場合、以下のようになります。

```
struct tag
{
  short i[2];
} test_record;
```

SQL ステートメントで test_record が参照されるときに配列がエレメントに展開されると、 :test_record は、:test_record.i[0], :test_record.i[1] と同等になります。

C および C++ での Null 終了ストリング

C/C++ の NULL 終了ストリングは、独自の SQLTYPE (文字の場合は 460/461 で、グラフィックの場合は 468/469) を持ちます。

C/C++ の NULL 終了ストリングは、 LANGLEVEL プリコンパイラー・オプションの値に基づいてさまざまに処理されます。これらの SQLTYPE のうちの 1 つのホスト変数と宣言された長さ *n* を SQL ステートメント内に指定し、さらにデータのバイト数 (文字タイプの場合) または 2 バイト文字 (グラフィック・タイプの場合) が *k* である場合を以下に説明します。

- PREP コマンドの LANGLEVEL オプションが SAA1 (デフォルト) の場合:

出力の場合:

k と n の関係 ...	**結果 ...**
k **>** *n*	*n* 文字はターゲットのホスト変数に移動され、 SQLWARN1 は 'W'、

	SQLCODE 0 (SQLSTATE 01004) に設定される。ヌル終止符はストリング内では使用されません。標識変数をホスト変数で指定した場合には、標識変数の値は *k* に設定されます。
k **=** *n*	*k* 文字はターゲットのホスト変数に移動され、 SQLWARN1 は 'N'、また SQLCODE 0 (SQLSTATE 01004) に設定される。ヌル終止符はストリング内では使用されません。標識変数をホスト変数で指定した場合には、標識変数の値は 0 に設定されます。
k **<** *n*	*k* 文字はターゲットのホスト変数に移動され、ヌル文字は文字 *k* + *1* に置かれる。標識変数をホスト変数で指定した場合には、標識変数の値は 0 に設定されます。

入力の場合: データベース・マネージャーは、最後がヌル終止符ではないこれらの SQLTYPE のうちの 1 つの入力ホスト変数を発見すると、文字 *n+1* にヌル終止符文字が含まれると想定します。

- PREP コマンドの LANGLEVEL オプションが MIA である場合:

出力の場合:

k と n の関係 ...	**結果 ...**
k **>=** *n*	*n* - *1* 文字はターゲットのホスト変数に移動され、 SQLWARN1 は 'W'、また SQLCODE 0 (SQLSTATE 01501) に設定される。 *n* 番目の文字はヌル終止符に設定されます。標識変数をホスト変数で指定した場合には、標識変数の値は *k* に設定されます。
k + *1* **=** *n*	*k* 文字はターゲットのホスト変数に移動され、ヌル終止符は文字 *n* に置かれる。標識変数を

	ホスト変数で指定した場合には、標識変数の値は 0 に設定されます。
k + 1 **<** *n*	*k* 文字はターゲットのホスト変数に移動され、文字 *k + 1* で開始している右側に *n - k -1* 個のブランクが追加されます。その後、ヌル終止符が文字 *n* に置かれます。標識変数をホスト変数で指定した場合には、標識変数の値は 0 に設定されます。

入力の場合: データベース・マネージャーが、最後がヌル終止符ではないこれらの SQLTYPE のうちの 1 つの入力ホスト変数を発見すると、SQLCODE -302 (SQLSTATE 22501) が戻されます。

長さが *n* の SQLTYPE 460 のホスト変数を他の SQL コンテキスト内に指定すると、上で定義したように、長さ *n* の VARCHAR データ・タイプとして処理されます。長さが *n* の SQLTYPE 468 のホスト変数をその他の SQL コンテキスト内に指定した場合には、上での定義のように、長さ *n* の VARGRAPHIC データ・タイプとして処理されます。

C および C++ でのポインター・データ・タイプ

ホスト変数は、特定のデータ・タイプへのポインターとして、以下のような制限付きで宣言されることがあります。

- ホスト変数をポインターとして宣言する場合、その他のホスト変数を、同じソース・ファイル内で同じ名前で宣言することはできない。以下の例は無効です。

```
char mystring[20];
char (*mystring)[20];
```

- NULL 終了文字配列へのポインターを宣言する場合は、括弧を使用する。その他のすべての場合は、括弧を使用することはできません。以下に例を示します。

```
EXEC SQL BEGIN DECLARE SECTION;
  char  (*arr)[10];  /* correct   */
  char  *(arr);      /* incorrect */
  char  *arr[10];    /* incorrect */
EXEC SQL END DECLARE SECTION;
```

  この例で、最初の宣言は 10 バイトの文字配列へのポインターです。これは有効なホスト変数となっています。 2 番目は無効な宣言です。文字へのポインターでは括弧は使用できません。 3 番目の宣言はポインターの配列です。このデータ・タイプはサポートされていません。

以下のようなホスト変数の宣言があるとします。

```
char *ptr
```

この宣言は受け入れられますが、*長さが未指定の NULL 終了文字ストリング* を意味するわけではありません。その代わり、これは*固定長の単一文字のホスト変数へのポインター* であることを表します。これは意図された宣言ではないかもしれません。別の文字ストリングを指示することができるポインター・ホスト変数を定義するには、上記の最初の宣言書式を用いてください。

- SQL ステートメントでポインター・ホスト変数を使用する場合は、以下の例のように、宣言されているのと同じ数のアスタリスクを前に付ける。

```
EXEC SQL BEGIN DECLARE SECTION;
  char (*mychar)[20];    /* Pointer to character array of 20 bytes */
EXEC SQL END DECLARE SECTION;

EXEC SQL SELECT column INTO :*mychar FROM table;   /* Correct */
```

- ホスト変数名には、アスタリスクだけが演算子として使用できる。
- アスタリスクは名前の一部と見なされないため、ホスト変数名の最大長は指定されたアスタリスクの数の影響を受けない。
- SQL ステートメント内でポインター変数を使用する場合は必ず、最適化レベルのプリコンパイル・オプション (OPTLEVEL) をデフォルト設定の 0 (最適化を行わない) のままにしておく。これは、データベース・マネージャーが SQLDA の最適化を行わないということを意味します。

C および C++ でのクラス・データ・メンバーのホスト変数としての使用

クラス・データ・メンバーは、ホスト変数として宣言できます (クラスまたはオブジェクト自身ではなく)。以下は、使用方法を説明する例です。

```
class STAFF
{
    private:
         EXEC SQL BEGIN DECLARE SECTION;
           char        staff_name[20];
           short int   staff_id;
           double      staff_salary;
         EXEC SQL END DECLARE SECTION;
         short       staff_in_db;
    .
    .
};
```

データ・メンバーへは、クラス・メンバー関数内の C++ コンパイラーにより提供される暗黙の *this* ポインターを介して、 SQL ステートメント内で直接アクセスできるだけです。 SQL ステートメント内でオブジェクト・インスタンス (`SELECT name INTO :my_obj.staff_name ...` など) を明示的に修飾することは**できません**。

SQL ステートメント内でクラス・データ・メンバーを直接参照する場合は、データベース・マネージャーが *this* ポインターを使用して参照を解決します。こうした理由から、最適化レベルのプリコンパイル・オプション (OPTLEVEL) は、デフォルト設定の 0 (最適化を行わない) のままにしておいてください。これは、データベース・マネージャーが SQLDA の最適化を行わないということを意味します。 (これは、ポインター・ホスト変数が SQL ステートメントに含まれていれば常に当てはまります。)

次の例は、SQL ステートメント内でホスト変数として宣言したクラス・データ・メンバーを、直接使用する方法を示しています。

```
    class STAFF
    {
⋮
        public:
⋮
           short int hire( void )
           {
             EXEC SQL INSERT INTO staff ( name,id,salary )
               VALUES ( :staff_name, :staff_id, :staff_salary );
             staff_in_db = (sqlca.sqlcode == 0);
             return sqlca.sqlcode;
           }
    };
```

この例では、クラス・データ・メンバーである `staff_name`、`staff_id`、および `staff_salary` が INSERT ステートメント内で直接使用されています。これらはホスト変数として宣言されているため (636ページの『クラス・データ・メンバーをホスト変数として宣言する例』の例を参照)、 *this* ポインターを用いて、現行対象に対して暗黙のうちに修飾されています。 SQL ステートメントでは、 *this* ポインターを介してアクセスすることができないデータ・メンバーも参照することができます。 これは、ポインターまたは参照ホスト変数を使用してこれらを間接的に参照することにより行うことができます。

次の例は、2 番目のオブジェクトである *otherGuy* を獲得する、 *asWellPaidAs* という新しい方法を示しています。この方法では、SQL ステートメント内でメンバーを直接参照できないため、ローカル・ポインターまたは参照ホスト変数を介して間接的にメンバーを参照します。

```
     short int STAFF::asWellPaidAs( STAFF otherGuy )
     {
         EXEC SQL BEGIN DECLARE SECTION;
           short &otherID = otherGuy.staff_id
           double otherSalary;
         EXEC SQL END DECLARE SECTION;
         EXEC SQL SELECT SALARY INTO :otherSalary
```

```
            FROM STAFF WHERE id = :otherID;
         if( sqlca.sqlcode == 0 )
            return staff_salary >= otherSalary;
         else
            return 0;
   }
```

C および C++ での修飾およびメンバー演算子の使用

組み込み SQL ステートメント内で、C++ 効力範囲解決演算子 '::' を使用したり、C/C++ メンバー演算子 '.' または '->' を使用したりすることは**できません**。これと同じことは、ローカル・ポインターまたは参照変数を使用することにより、簡単に行うことができます。ローカル・ポインターや参照変数は SQL ステートメントの外部に設定して使用したい効力範囲内の変数を指定し、その後は SQL ステートメント内でこれを参照するために使用されます。以下に、正しい使用方法の例を示します。

```
EXEC SQL BEGIN DECLARE SECTION;
  char (& localName)[20] = ::name;
EXEC SQL END DECLARE SECTION;
EXEC SQL
  SELECT name INTO :localName FROM STAFF
  WHERE name = 'Sanders';
```

C および C++ でのグラフィック・ホスト変数の処理

C または C++ で作成されたアプリケーションでグラフィック・データを処理するには、 wchar_t C/C++ データ・タイプまたは DB2 提供の sqldbchar データ・タイプに基づくホスト変数を使用してください。これら 2 つのホスト変数は、GRAPHIC、VARGRAPHIC、または DBCLOB などの表の列に割り当てることができます。たとえば、表の GRAPHIC または VARGRAPHIC 列から、DBCS データを更新したり選択したりすることができます。

グラフィック・ホスト変数には、以下のような 3 つの有効な書式があります。

- 単純グラフィック書式

  単純グラフィック・ホスト変数は、GRAPHIC(1) SQL データ・タイプに相当する 468/469 の SQLTYPE を持っています。 (623ページのグラフィック宣言の構文 (単純グラフィック書式および NULL 終了グラフィック書式) を参照。)

- NULL 終了グラフィック書式

  NULL 終了とは、グラフィック・ストリングの最後の文字のバイトがすべて 2 進ゼロ ('¥0') である状態を言います。これらは SQLTYPE が 400/401 となります。 (623ページのグラフィック宣言の構文 (単純グラフィック書式および NULL 終了グラフィック書式) を参照。)

- VARGRAPHIC 構造書式

VARGRAPHIC 構造ホスト変数は、長さが 1 から 16 336 バイトの間の場合は SQLTYPE が 464/465 となります。この変数の長さが 2000 から 16 350 バイトの間の場合は、SQLTYPE が 472/473 となります。 (624ページのグラフィック宣言の構文 (VARGRAPHIC 構造書式) を参照。)

C および C++ でのマルチバイト文字のエンコード

文字のエンコード・スキーマの中には、特に東アジアの国々の文字には 1 つの文字を表すのにマルチバイトを必要とするものがあります。このデータの外部表現は文字の*マルチバイト文字コード* 表現と呼ばれ、 2 バイト文字 (2 バイトで表される文字) を含みます。 DB2 のグラフィック・データは、2 バイト文字からなります。

2 バイト文字で文字ストリングを扱うためには、アプリケーションでデータの内部表現を使用するのが便利です。この内部表現は、2 バイト文字の*ワイド・キャラクター・コード* 表現と呼ばれており、通常 `wchar_t` C/C++ データ・タイプで使用される形式です。ワイド・キャラクター・データの処理やワイド・キャラクター形式データのマルチバイト形式との変換を行うためには、 ANSI C および X/OPEN Portability Guide 4 (XPG4) に準拠するサブルーチンを使用することができます。

アプリケーションでは、文字データをマルチバイト形式またはワイド・キャラクター形式のどちらかで処理できますが、データベース・マネージャーとの対話は、DBCS (マルチバイト) 文字コードでしか行うことができないことに注意してください。つまり、データの GRAPHIC 列への保管や GRAPHIC 列からの検索は、DBCS 形式で行われます。 WCHARTYPE プリコンパイラー・オプションは、ワイド・キャラクター形式のアプリケーション・データがデータベース・エンジンで交換される際に、これをマルチバイト形式に変換したり元に戻したりするために使用されます。

C および C++ での wchar_t または sqldbchar データ・タイプの選択

DB2 グラフィック・データのサイズおよびエンコードは、特定のコード・ページではどのプラットフォームでも同じですが、 ANSI C または C++ `wchar_t` データ・タイプのサイズおよび内部書式は、使用するコンパイラーとプラットフォームによって異なります。しかしながら、`sqldbchar` データ・タイプは、DB2 によってサイズが 2 バイトと定義されており、データベース内で保管されるのと同じ形式で DBCS および UCS-2 データを操作する、可搬性のある方法が使用されています。 UCS-2 データの詳細については、 535ページの『日本語および中国語 (繁体字) EUC および UCS-2 コード・セットに関する考慮事項』および*管理の手引き* を参照してください。

DB2 C グラフィック・ホスト変数タイプはすべて、`wchar_t` か `sqldbchar` によって定義できます。 WCHARTYPE CONVERT プリコンパイル・オプション (640ページの『C および C++ での WCHARTYPE プリコンパイラー・オプション』で説明) を使用してアプリケーションを構築する場合には、必ず `wchar_t` の方を使用してください。

注: Windows プラットフォーム上で WCHARTYPE CONVERT オプションを指定する際には、 Windows プラットフォーム上の wchar_t は Unicode であることに注意してください。したがって、ご使用の C/C++ コンパイラーの wchar_t が Unicode で

ない場合には、 wcstombs() 関数呼び出しは SQLCODE -1421 (SQLSTATE=22504) を出して失敗します。この場合、WCHARTYPE NOCONVERT オプションを指定したり、ご使用のプログラムから wcstombs() および mbstowcs() 関数を明示的に呼び出したりすることができます。

WCHARTYPE NOCONVERT プリコンパイル・オプションを使用してアプリケーションを構築する場合には、異なる DB2 クライアントとサーバー・プラットフォーム間でも最大限の可搬性を得られるよう、 `sqldbchar` の方を使用してください。 WCHARTYPE NOCONVERT を使用する場合でも `wchar_t` は使えますが、 `wchar_t` が 2 バイトで定義されているプラットフォームだけに限ります。

ホスト変数宣言で `wchar_t` か `sqldbchar` を誤って使用すると、プリコンパイル時に SQLCODE 15 (SQLSTATE ではない) が返されます。

C および C++ での WCHARTYPE プリコンパイラー・オプション

WCHARTYPE プリコンパイラー・オプションを使用すると、 C/C++ アプリケーションでどのグラフィック文字形式を使用するかを指定できます。このオプションにより、グラフィック・データをマルチバイト形式またはワイド・キャラクター形式のどちらにするかを柔軟に選択することができます。 WCHARTYPE オプションには、次の 2 つの値があります。

CONVERT

WCHARTYPE CONVERT オプションを選択した場合、文字コードはグラフィック・ホスト変数とデータベース・マネージャーとの間で変換されます。グラフィック入力ホスト変数の場合、ワイド・キャラクター形式からマルチバイト DBCS 文字形式への文字コード変換は、データがデータベース・マネージャーに送信される前に ANSI C 関数の `wcstombs()` を使用して行われます。グラフィック出力ホスト変数の場合には、マルチバイト DBCS 文字形式からワイド・キャラクター形式への文字コード変換は、データベース・マネージャーから受け取られたデータがホスト変数に保管される前に、 ANSI C 関数の `mbstowcs()` を使用して実行されます。

WCHARTYPE CONVERT を使用する利点は、それによってアプリケーションが、データベース・マネージャーと通信する前に、データをマルチバイト形式に明示的に変換しなくても、ワイド・キャラクター・ストリング (L-リテラル、'wc' ストリング関数など) を処理するための ANSI C 機構を十分に利用できることです。この短所としては、暗黙のうちに変換を行うことによってアプリケーション実行時のパフォーマンスに影響を及ぼすことがあり、さらにメモリー要件が大きくなる恐れがあることが挙げられます。

WCHARTYPE CONVERT を選択した場合は、すべてのグラフィック・ホスト変数を、`sqldbchar` ではなく、`wchar_t` を使用して宣言してください。

WCHARTYPE CONVERT 振る舞いは希望するが、アプリケーションはプリコンパイルする必要がない場合 (たとえば、CLI アプリケーション)、コンパイル時に C プリプロセッサー・マクロ `SQL_WCHART_CONVERT` を定義してくださ

い。これによって、DB2 ヘッダー・ファイルの特定の定義でデータ・タイプ sqldbchar ではなく、 wchar_t が使用されます。

注: WCHARTYPE CONVERT プリコンパイル・オプションは、現在、 DB2 Windows 3.1 クライアントで実行するプログラムではサポートされていません。それらのプログラムには、デフォルト (WCHARTYPE NOCONVERT) を使用してください。

NOCONVERT (デフォルト)

WCHARTYPE NOCONVERT オプションを選択した場合、あるいは WCHARTYPE オプションをまったく指定しない場合は、アプリケーションとデータベース・マネージャーの間で暗黙の文字コード変換は行われません。グラフィック・ホスト変数内のデータは、未変更の DBCS 文字としてデータベース・マネージャーとの間で送受信されます。これにはパフォーマンスを向上させるという利点がありますが、短所としてアプリケーションが wchar_t ホスト変数内のワイド・キャラクター・データの使用をやめるか、またはデータベース・マネージャーとのインターフェースをとる際にデータのマルチバイト形式への変換のために wcstombs() および mbstowcs() 関数を明示的に呼び出さなければならないということがあります。

WCHARTYPE NOCONVERT を選択した場合は、他の DB2 クライアント / サーバー・プラットフォームへの可搬性を最大限に得られるようにするため、すべてのグラフィック・ホスト変数を sqldbchar タイプを使用して宣言してください。

詳細については、*コマンド解説書* を参照してください。

注意すべきその他の指針としては以下のものがあります。

- wchar_t または sqldbchar サポートは DBCS データの処理のために使用されるため、これを使用する場合は DBCS または EUC で使用可能なハードウェアとソフトウェアが必要になる。このサポートが使用可能であるのは、DBCS 環境の DB2 ユニバーサル・データベースか、または UCS-2 データベースに接続されている任意のアプリケーション (1 バイト・アプリケーションを含む) で GRAPHIC データを処理している場合のみです。
- DBCS 以外の文字と、DBCS 以外の文字に変換できるワイド・キャラクターは、グラフィック・ストリング内では使用してはならない。 *DBCS 以外の文字* とは、 1 バイト文字と、2 バイト文字以外の文字のことを指します。グラフィック・ストリングには、その値に 2 バイト文字のコード・ポイントのみが含まれているかを確認するための妥当性検査は行われません。グラフィック・ホスト変数には、DBCS データか、または WCHARTYPE CONVERT が有効な場合は、 DBCS データに変換されるワイド・キャラクター・データしか含めることができません。 2 バイト文字と 1 バイト文字が混在しているデータは、文字ホスト変数に保管してください。混合データのホスト変数は WCHARTYPE オプションの設定の影響を受けないことに注意してください。

- WCHARTYPE NOCONVERT プリコンパイル・オプションを使用しているアプリケーションでは、 L リテラルをグラフィック・ホスト変数とともに使用しない。これは、L リテラルがワイド・キャラクター形式であるためです。 L リテラルは、L という接頭部を付けた C 言語のワイド・キャラクター・ストリング・リテラルであり、データ・タイプは "array of wchar_t" です。たとえば、L"dbcs-string" は L リテラルです。
- WCHARTYPE CONVERT プリコンパイル・オプションを使用しているアプリケーションでは、 L リテラルを使用して wchar_t ホスト変数を初期化するが、 SQL ステートメントでは使用できない。 SQL ステートメントでは、 L リテラルを使用する代わりに WCHARTYPE の設定から独立しているグラフィック・ストリング定数を使用してください。
- WCHARTYPE オプションの設定は、ホスト変数だけでなく SQLDA 構造を使用してデータベース・マネージャーとの間で受け渡しするグラフィック・データに影響を与える。 WCHARTYPE CONVERT が有効な場合、 SQLDA を介してアプリケーションから受け取られるグラフィック・データはワイド・キャラクター形式と見なされ、wcstombs() を暗黙のうちに呼び出して DBCS 形式に変換されます。同様に、アプリケーションが受け取るグラフィック出力データは、アプリケーション・ストレージに保管される前にワイド・キャラクター形式に変換されています。
- 境界域が設定されていないストアード・プロシージャーは、 WCHARTYPE NOCONVERT オプションを用いてプリコンパイルしなければならない。通常、境界域の設定されていないストアード・プロシージャーは CONVERT または NOCONVERT のいずれかのオプションを用いてプリコンパイルすることができますが、これにより、ストアード・プロシージャーに含まれる SQL ステートメントに操作されるグラフィック・データの形式は影響を受けます。ただしどちらの場合も、SQLDA を介してストアード・プロシージャーに渡されるグラフィック・データはすべて DBCS 形式となります。その上、 SQLDA を介してストアード・プロシージャーから渡されるデータは、 DBCS 形式でなければなりません。
- アプリケーションがデータベース・アプリケーション・リモート・インターフェース (DARI) のインターフェース (sqleproc() API) を介してストアード・プロシージャーを呼び出す場合、入力 SQLDA 内のグラフィック・データはすべて、呼び出しているアプリケーションの WCHARTYPE 設定に関係なく、DBCS 形式でなければならない。または UCS-2 データベースに接続されている場合は、UCS-2 でなければならない。同じく、出力 SQLDA 内のグラフィック・データはすべて、 WCHARTYPE 設定に関係なく、 DBCS 形式、または UCS-2 データベースに接続されている場合は UCS-2 形式で戻されます。
- アプリケーションが SQL CALL ステートメントを介してストアード・プロシージャーを呼び出す場合は、呼び出しているアプリケーションの WCHARTYPE 設定に従って、 SQLDA でグラフィック・データが変換される。

- ユーザー定義関数 (UDF) に渡されるグラフィック・データは、常に DBCS 形式である。その上、UDF から戻されるグラフィック・データはすべて、 DBCS データベースでは DBCS 形式、 EUC および UCS-2 データベースでは UCS-2 形式と見なされます。
- DBCLOB ファイル参照変数の使用により DBCLOB ファイルに保管されるデータは、 DBCS 形式か、または UCS-2 データベースの場合には、UCS-2 形式で保管されます。同様に、DBCLOB ファイルからの入力データは、 DBCS 形式か、または UCS-2 データベースの場合には UCS-2 形式のいずれかで検索されます。

注:

1. C 言語アプリケーションを WCHARTYPE CONVERT オプションを使用してプリコンパイルする場合、 DB2 は変換関数がデータを渡すときに、入力出力両方のアプリケーションのグラフィック・データを妥当性検査します。 CONVERT オプションを使用**しない**場合は、グラフィック・データの変換は行われず、したがって検証も行われません。このことが CONVERT/NOCONVERT 混合環境では、無効なデータが NOCONVERT アプリケーションによって挿入され、それを CONVERT アプリケーションが取り出したりする場合に、問題の原因になります。このようなデータの変換は失敗し、CONVERT アプリケーションでの FETCH 時に、SQLCODE -1421 (SQLSTATE 22504) が返されます。
2. WCHARTYPE CONVERT プリコンパイル・オプションは、現在、DB2 Windows 3.1 クライアントで実行するプログラムではサポートされていません。この場合には、デフォルトの WCHARTYPE NOCONVERT オプションを使用してください。

C および C++ での日本語または中国語 (繁体字) EUC、および UCS-2 に関する考慮事項

アプリケーション・コード・ページが日本語または中国語 (繁体字) EUC の場合、またはアプリケーションが UCS-2 データベースと接続されている場合、 CONVERT オプションか NOCONVERT オプションのどちらか、および `wchar_t` または `sqldbchar` グラフィック・ホスト変数、または入力 / 出力 SQLDA を使用することにより、データベース・サーバーで GRAPHIC 列にアクセスできます。この節で *DBCS 形式* に言及する場合、それは EUC データ用の UCS-2 エンコード・スキーマを指します。次の 2 つのケースを考えてみてください。

- CONVERT オプションを使用する場合

  DB2 クライアントによって、グラフィック・データがワイド・キャラクター形式からご使用のアプリケーション・コード・ページに変換され、その後、入力 SQLDA をデータベース・サーバーに送信する前に UCS-2 に変換します。グラフィック・データはすべて、UCS-2 コード・ページ ID によってタグ付けされたデータベース・サーバーに送られます。混合文字データは、アプリケーション・コード・ページ ID によってタグ付けされます。クライアントによってデータベースからグラフィック・データが取り出されると、そのグラフィック・データは UCS-2 コード・ページ ID によってタグ付けされます。 DB2 クライアントが、データを UCS-2 からクライアント・アプリケーション・コード・ページへ変換し、さらにそれをワイド・キャラクター形

式に変換します。ホスト変数の代わりに入力 SQLDA を使用した場合は、グラフィック・データを必ずワイド・キャラクター形式でエンコードする必要があります。このデータは UCS-2 に変換され、その後データベース・サーバーに送られます。上記の変換はパフォーマンスに影響を及ぼします。

- NOCONVERT オプションを使用する場合

  グラフィック・データは UCS-2 によってエンコードされ、 UCS-2 コード・ページでタグ付けされたものと DB2 からは見なされます。変換は行われません。 DB2 は、グラフィック・ホスト変数を単にバケットとして使用されるものとみなします。NOCONVERT オプションを選択した場合、データベース・サーバーから取り出されるグラフィック・データは、 UCS-2 によってエンコードされたアプリケーションに渡されます。アプリケーション・コード・ページから UCS-2、および UCS-2 からアプリケーション・コード・ページへの変換は、すべてユーザーの責任で行うことになります。 UCS-2 としてタグ付けされたデータは、変換や置換なしでデータベース・サーバーに送られます。

変換を最小限に抑えるには、 NOCONVERT オプションを使用してアプリケーション内で変換を処理するか、または GRAPHIC 列を使用しないかのいずれかです。 `wchar_t` エンコードが 2 バイト Unicode のクライアント環境 (たとえば Windows NT または AIX バージョン 4.3 およびそれ以降など) の場合には、 NOCONVERT オプションを使用して直接 UCS-2 で作業できます。この場合、ご使用のアプリケーションはビッグ・エンディアンとリトル・エンディアン・アーキテクチャーとの違いを扱わなければなりません。 NOCONVERT オプションを使用する場合、 DB2 ユニバーサル・データベースは、常に 2 バイト・ビッグ・エンディアンである sqldbchar を使用します。

UCS-2 への変換後 (NOCONVERT 指定の場合) や、ワイド・キャラクター形式への変換 (CONVERT 指定の場合) によって、 IBM-eucJP/IBM-eucTW CS0 (7 ビット ASCII) データおよび IBM-eucJP CS2 (カタカナ) データをグラフィック・ホスト変数に割り当てることはしないでください。これは、どちらの EUC コード・セットも UCS-2 から PC DBCS へと変換すると単一バイトになってしまうためです。

通常、eucJP および eucTW は GRAPHIC データを UCS-2 として保管しますが、これらのデータベースにある GRAPHIC データは非 ASCII eucJP または eucTW データのままです。特に、そのような GRAPHIC データに埋め込まれるスペースは、DBCS スペースです (UCS-2、U+3000 では表意文字スペースとも呼ばれます)。しかし、UCS-2 データベースの場合には、 GRAPHIC データに UCS-2 文字を含めることができ、スペースの埋め込みは UCS-2 スペース、U+0020 を使用して実行されます。 UCS-2 データベースから UCS-2 を検索する場合と、 eucJP および eucTW データベースから UCS-2 データを検索する場合には、この違いに注意してください。

一般的な EUC アプリケーション開発の指針については、 535ページの『日本語および中国語 (繁体字) EUC および UCS-2 コード・セットに関する考慮事項』を参照してください。

C および C++ でのサポートされている SQL データ・タイプ

特定の事前定義済み C および C++ データ・タイプは、データベース・マネージャーの列タイプに対応しています。これらの C/C++ データ・タイプのみが、ホスト変数として宣言できます。

表30 は、それぞれの列タイプに対応する C/C++ の列タイプを示しています。プリコンパイラーはホスト変数宣言を検出すると、該当する SQL データ・タイプの値を判別します。データベース・マネージャーはこの値を使用して、アプリケーションとの間でやりとりするデータを変換します。

注: DB2 ホスト言語で、DATALINK データ・タイプをサポートするホスト変数はありません。

表 30. C/C++ 宣言にマップされた SQL データ・タイプ

SQL 列タイプ[1]	C/C++ データ・タイプ	SQL 列タイプ記述
SMALLINT (500 または 501)	short short int sqlint16	16 ビットの符号付き整数
INTEGER (496 または 497)	long long int sqlint32[2]	32 ビットの符号付き整数
BIGINT (492 または 493)	long long long __int64 sqlint64[3]	64 ビットの符号付き整数
REAL[4] (480 または 481)	float	単精度浮動小数点
DOUBLE[5] (480 または 481)	double	倍精度浮動小数点
DECIMAL(*p*,*s*) (484 または 485)	厳密な対応なし; double を使用	パック 10 進数 (パック 10 進フィールドを文字データとして操作するために CHAR および DECIMAL 関数を使用することを推奨。)
CHAR(1) (452 または 453)	char	単一文字
CHAR(*n*) (452 または 453)	厳密な対応なし; char[*n+1*] を使用 (n はデータを保持するだけの十分な大きさ) 1<=*n*<=254	固定長文字ストリング

表 30. C/C++ 宣言にマップされた SQL データ・タイプ (続き)

SQL 列タイプ[1]	C/C++ データ・タイプ	SQL 列タイプ記述

VARCHAR(*n*)
(448 または 449)
	struct tag {
	short int;
	char[*n*]
	}
	
	1<=*n*<=32 672
		2 バイトのストリング長指定子を持つ、NULL 終了可変文字以外のストリング

	代替使用; char[*n+1*] を使用 (n はデータを保持するだけの十分な大きさ) 1<=*n*<=32 672	NULL 終了可変長文字ストリング **注:** 460/461 の SQL タイプを割り当てられる。

LONG VARCHAR
(456 または 457)
	struct tag {
	short int;
	char[*n*]
	}
	
	32 673<=*n*<=32 700
		2 バイトのストリング長指定子を持つ、NULL 終了可変文字以外のストリング

CLOB(*n*) (408 または 409)	sql type is clob(*n*) 1<=*n*<=2 147 483 647	4 バイトのストリング長指定子を持つ、NULL 終了可変文字以外のストリング
CLOB ロケーター変数[6] (964 または 965)	sql type is clob_locator	サーバー上の CLOB エンティティーを識別する
CLOB ファイル参照変数[6] (920 または 921)	sql type is clob_file	CLOB データを含むファイルの記述子
BLOB(*n*) (404 または 405)	sql type is blob(*n*) 1<=*n*<=2 147 483 647	4 バイト・ストリング長標識のヌル終了可変長バイナリー・ストリングでない
BLOB ロケーター変数[6] (960 または 961)	sql type is blob_locator	サーバー上の BLOB エンティティーを識別する
BLOB ファイル参照変数[6] (916 または 917)	sql type is blob_file	BLOB データを含むファイルの記述子
DATE (384 または 385)	NULL 終了文字書式	NULL 終止符を収容するために最低 11 文字を使用できる。
	VARCHAR 構造書式	最低 10 文字を使用できる。
TIME (388 または 389)	NULL 終了文字書式	NULL 終止符を収容するために最低 9 文字を使用できる。
	VARCHAR 構造書式	最低 8 文字を使用できる。
TIMESTAMP (392 または 393)	NULL 終了文字書式	NULL 終止符を収容するために最低 27 文字を使用できる。
	VARCHAR 構造書式	最低 26 文字が使用できる。
注: 以下のデータ・タイプは、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(1) (468 または 469)	sqldbchar	単一の 2 バイト文字

表 30. C/C++ 宣言にマップされた SQL データ・タイプ (続き)

SQL 列タイプ[1]	C/C++ データ・タイプ	SQL 列タイプ記述
GRAPHIC(*n*) (468 または 469)	厳密な対応なし; sqldbchar[*n+1*] を使用 (n はデータを保持するだけの十分な大きさ) 1<=*n*<=127	固定長 2 バイト文字ストリング

VARGRAPHIC(*n*)
(464 または 465)
	struct tag {
	short int;
	sqldbchar[*n*]
	}
	
	1<=*n*<=16 336
		2 バイトのストリング長指定子を持つ、NULL 終了可変 2 バイト文字以外のストリング

	代替使用; sqldbchar[*n+1*] を使用 (n はデータを保持するだけの十分な大きさ) 1<=*n*<=16 336	NULL 終了可変長 2 バイト文字ストリング **注:** 400/401 の SQL タイプを割り当てられる。

LONG VARGRAPHIC
(472 または 473)
	struct tag {
	short int;
	sqldbchar[*n*]
	}
	
	16 337<=*n*<=16 350
		2 バイトのストリング長指定子を持つ、NULL 終了可変 2 バイト文字以外のストリング

注: 以下のデータ・タイプは、WCHARTYPE CONVERT 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(1) (468 または 469)	wchar_t	• 単一のワイド・キャラクター (C タイプの場合) • 単一の 2 バイト文字 (列タイプの場合)
GRAPHIC(*n*) (468 または 469)	厳密な対応なし; wchar[*n+1*] を使用 (n はデータを保持するだけの十分な大きさ) 1<=*n*<=127	固定長 2 バイト文字ストリング

VARGRAPHIC(*n*)
(464 または 465)
	struct tag {
	short int;
	wchar_t [*n*]
	}
	
	1<=*n*<=16 336
		2 バイトのストリング長指定子を持つ、NULL 終了可変 2 バイト文字以外のストリング

	代替使用; char[*n+1*] を使用 (n はデータを保持するだけの十分な大きさ) 1<=*n*<=16 336	NULL 終了可変長 2 バイト文字ストリング **注:** 400/401 の SQL タイプを割り当てられる。

LONG VARGRAPHIC
(472 または 473)
	struct tag {
	short int;
	wchar_t [*n*]
	}
	
	16 337<=*n*<=16 350
		2 バイトのストリング長指定子を持つ、NULL 終了可変 2 バイト文字以外のストリング

注: 以下のデータ・タイプは DBCS または EUC 環境でのみ使用できる。		
DBCLOB(*n*) (412 または 413)	sql type is dbclob(*n*) 1<=*n*<=1 073 741 823	4 バイトのストリング長指定子を持つ、NULL 終了可変 2 バイト文字以外のストリング

表 30. C/C++ 宣言にマップされた SQL データ・タイプ (続き)

SQL 列タイプ[1]	C/C++ データ・タイプ	SQL 列タイプ記述
DBCLOB ロケーター変数[6] (968 または 969)	sql type is dbclob_locator	サーバー上の DBCLOB エンティティーを識別する
DBCLOB ファイル参照変数[6] (924 または 925)	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**の下の最初の数字は標識変数が提供されないことを示し、 2 番目の数字は標識変数が提供されることを示します。標識変数は、ヌル値を示したり、切り捨てられたストリングの長さを保留するのに必要です。これらは、これらのデータ・タイプの SQLDA の SQLTYPE フィールドに示される値です。
2. プラットフォームと互換性を持たせるには、sqlint32 を使用してください。 64 ビットの UNIX プラットフォームでは、″long″ は 64 ビット整数です。 64 ビットの Windows オペレーティング・システムおよび 32 ビットの UNIX プラットフォームでは、″long″ は 32 ビットの整数です。
3. プラットフォームで互換性を持たせるには、sqlint64 を使用してください。 DB2 ユニバーサル・データベースの sqlsystm.h ヘッダー・ファイルは、 Microsoft コンパイラーを使用する場合には、 Windows NT プラットフォームで sqlint64 を ″__int64″ とタイプ定義します。また、32 ビットの UNIX プラットフォームでは ″long long″ と 62 ビットの UNIX プラットフォームでは ″long″ とタイプ定義します。
4. FLOAT(*n*) ここで 0 < *n* < 25 の場合、REAL と同義。 SQLDA での REAL と DOUBLE の違いは長さの値です (4 または 8)。
5. 次の SQL タイプは、DOUBLE と同義語。
   - FLOAT
   - FLOAT(*n*)。ここで、*n* の取る範囲は 24 < *n* < 54。
   - DOUBLE PRECISION
6. これは列タイプではなく、ホスト変数タイプである。

以下に、サポートされている SQL データ・タイプのために宣言されたホスト変数を使用したサンプルの SQL 宣言セクションを示します。

```
EXEC SQL BEGIN DECLARE SECTION;

  .
  .
  .
      short     age = 26;              /* SQL type  500 */
      short     year;                  /* SQL type  500 */
      sqlint32  salary;                /* SQL type  496 */
      sqlint32  deptno;                /* SQL type  496 */
      float     bonus;                 /* SQL type  480 */
      double    wage;                  /* SQL type  480 */
      char      mi;                    /* SQL type  452 */
      char      name[6];               /* SQL type  460 */
      struct   {
                short len;
                char data[24];
               } address;              /* SQL type  448 */
      struct   {
                short len;
                char data[32695];
               } voice;                /* SQL type  456 */
      sql type is clob(1m)
```

```
         chapter;                /* SQL type  408 */
sql type is clob_locator
         chapter_locator;        /* SQL type  964 */
sql type is clob_file
         chapter_file_ref;       /* SQL type  920 */
sql type is blob(1m)
         video;                  /* SQL type  404 */
sql type is blob_locator
         video_locator;          /* SQL type  960 */
sql type is blob_file
         video_file_ref;         /* SQL type  916 */
sql type is dbclob(1m)
         tokyo_phone_dir;        /* SQL type  412 */
sql type is dbclob_locator
         tokyo_phone_dir_lctr;   /* SQL type  968 */
sql type is dbclob_file
         tokyo_phone_dir_flref;  /* SQL type  924 */
struct   {
         short len;
         sqldbchar data[100];
        } vargraphic1;           /* SQL type  464 */
                                 /* Precompiled with
                                 WCHARTYPE NOCONVERT option */
struct   {
         short len;
         wchar_t data[100];
        } vargraphic2;           /* SQL type  464 */
                                 /* Precompiled with
                                 WCHARTYPE CONVERT option */
struct   {
         short len;
         sqldbchar data[10000];
        } long_vargraphic1;      /* SQL type  472 */
                                 /* Precompiled with
                                 WCHARTYPE NOCONVERT option */
struct   {
         short len;
         wchar_t data[10000];
        } long_vargraphic2;      /* SQL type  472 */
                                 /* Precompiled with
                                 WCHARTYPE CONVERT option */
sqldbchar graphic1[100];         /* SQL type  468 */
                                 /* Precompiled with
                                 WCHARTYPE NOCONVERT option */
wchar_t   graphic2[100];         /* SQL type  468 */
                                 /* Precompiled with
                                 WCHARTYPE CONVERT option */
char      date[11];              /* SQL type  384 */
char      time[9];               /* SQL type  388 */
char      timestamp[27];         /* SQL type  392 */
short     wage_ind;              /* Null indicator */
```

⋮

```
EXEC SQL END DECLARE SECTION;
```

以下に、サポートされている C/C++ データ・タイプに関するその他の規則を示します。

- データ・タイプ `char` は `char` または `unsigned char` と宣言することができる。
- データベース・マネージャーは、 NULL で終了する可変長文字ストリング・データ・タイプ char[*n*] (データ・タイプ 460) を、 VARCHAR(*m*) として処理する。
  - LANGLEVEL が SAA1 の場合、ホスト変数の長さ *m* は、 char[*n*] 内の文字ストリングの長さ *n* または最初のヌル終止符 (¥0) の前のバイト数のいずれか小さい方と等しくなる。
  - LANGLEVEL が MIA の場合には、ホスト変数の長さ *m* は最初のヌル終止符 (¥0) の前のバイト数と等しくなる。
- データベース・マネージャーは、 null で終了する可変長グラフィック・ストリング・データ・タイプ `wchar_t[n]` または `sqldbchar[n]` (データ・タイプ 400) を、 VARGRAPHIC(*m*) として処理する。
  - LANGLEVEL が SAA1 の場合、ホスト変数の長さ *m* は、`wchar_t[n]` または `sqldbchar[n]` 内の文字ストリングの長さ *n*、または最初のグラフィック・ヌル終止符の前の文字数のいずれか小さい方と等しくなる。
  - LANGLEVEL が MIA の場合には、ホスト変数の長さ *m* は最初のグラフィック・ヌル終止符の前の文字数と等しくなる。
- 無符号数値データ・タイプはサポートされていない。
- C/C++ データ・タイプの `int` は、その内部表現がマシン依存型であるため使用できない。

C および C++ における FOR BIT DATA

標準的な C または C++ のストリング・タイプである 460 は、FOR BIT DATA に指定された列に使用しないでください。データベース・マネージャーは、NULL 文字が検出されると、このデータ・タイプを切り捨てます。 VARCHAR (SQL タイプ 448) または CLOB (SQL タイプ 408) のどちらかを使用してください。

C/C++ ストアード・プロシージャー、関数、およびメソッドのタイプ

次の表は、ストアード・プロシージャー、UDF、およびメソッドの SQL データ・タイプおよび C データ・タイプ間のサポートされるマッピングをリストしています。

表 31. C/C++ 宣言にマップされた SQL データ・タイプ

SQL 列名	C/C++ データ・タイプ	SQL 列タイプ記述
SMALLINT (500 または 501)	sqlint16	16 ビットの符号付き整数
INTEGER (496 または 497)	sqlint32	32 ビットの符号付き整数
BIGINT (492 または 493)	sqlint64	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)	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・タイプ (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*) (452 または 453)	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	固定長、ヌル終了文字ストリング
CHAR(*n*) FOR BIT DATA (452 または 453)	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	固定長文字ストリング
VARCHAR(*n*) (448 または 449) (460 または 461)	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	ヌル終了可変長ストリング

VARCHAR(*n*) FOR BIT DATA
(448 または 449)
	struct {
	sqluint16 長;
	char[*n*]
	}
	
	1<=*n*<=32 672
		ヌル終了可変長文字ストリングでない

LONG VARCHAR
(456 または 457)
	struct {
	sqluint16 長;
	char[*n*]
	}
	
	32 673<=*n*<=32 700
		ヌル終了可変長文字ストリングでない

CLOB(*n*)
(408 または 409)
	struct {
	sqluint32 長;
	char data[n];
	}
	
	1<=*n*<=2 147 483 647
		4 バイトストリング長標識のヌル終了可変長文字ストリングでない

BLOB(*n*)
(404 または 405)
	struct {
	sqluint32 長;
	char data[n];
	}
	
	1<=*n*<=2 147 483 647
		4 バイト・ストリング長標識のヌル終了可変長バイナリー・ストリングでない

表 31. C/C++ 宣言にマップされた SQL データ・タイプ (続き)

SQL 列名	C/C++ データ・タイプ	SQL 列タイプ記述
DATE (384 または 385)	char[11]	ヌル終了文字形式
TIME (388 または 389)	char[9]	ヌル終了文字形式
TIMESTAMP (392 または 393)	char[27]	ヌル終了文字形式
注: 以下のデータ・タイプは WCHARTYPE NOCONVERT オプションで再コンパイル時に DBCS または EUC 環境でのみ使用可能です。		
GRAPHIC(*n*) (468 または 469)	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	固定長、ヌル終了 2 バイト文字ストリング
VARGRAPHIC(*n*) (400 または 401)	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	ヌル終了、可変長 2 バイト文字ストリングでない

LONG VARGRAPHIC
(472 または 473)
	struct {
	sqluint16 長;
	sqldbchar[*n*]
	}
	
	16 337<=*n*<=16 350
		ヌル終了、可変長 2 バイト文字ストリングでない

DBCLOB(*n*)
(412 または 413)
	struct {
	sqluint32 長;
	sqldbchar data[n];
	}
	
	1<=*n*<=1 073 741 823
		4 バイトストリング長標識のヌル終了可変長文字ストリングでない

C および C++ における SQLSTATE および SQLCODE 変数

LANGLEVEL プリコンパイル・オプションを SQL92E の値とともに使用すると、次の 2 つの宣言をホスト変数として組み込めます。

```
EXEC SQL BEGIN DECLARE SECTION;
  char      SQLSTATE[6]
  sqlint32  SQLCODE;

.
.
.

EXEC SQL END DECLARE SECTION;
```

これらのいずれも指定しない場合は、SQLCODE 宣言はプリコンパイル中であると見なされます。このオプションを使用するときには、INCLUDE SQLCA ステートメントを指定してはならないことに注意してください。

複数のソース・ファイルから成るアプリケーションでは、上の例のように、最初のソース・ファイルで SQLCODE および SQLSTATE 変数を定義することができます。その後のソース・ファイルは、次のようにその定義を修正する必要があります。

```
extern sqlint32 SQLCODE;
extern char     SQLSTATE[6];
```

第21章 Java でのプログラミング

Java のプログラミングに関する考慮事項 . . 655
SQLJ と JDBC の比較 656
他の言語と比較した Java の利点. . . . 656
Java における SQL セキュリティー. . . 656
Java のソースおよび出力ファイル . . . 656
Java クラス・ライブラリー 657
Java パッケージ 657
Java でサポートされている SQL データ・タイプ 658
Java の SQLSTATE および SQLCODE 値 659
Java のトレース機能. 660
Java アプリケーションおよびアプレットの作成 660
動作の仕組み 661
JDBC プログラミング 663
DB2Appl プログラムの実行方法. . . . 664
JDBC の例: DB2Appl.java 665
JDBC アプリケーションの配布 666
JDBC アプレットの配布および実行. . . 666
JDBC アプレット・サーバーへの接続 . . 667
JDBC 2.0 668
JDBC 2.0 コア API サポート. . . . 668
JDBC 2.0 オプション・パッケージ API サポート 669
JDBC 2.0 互換性 670
SQLJ プログラミング 671
DB2 SQLJ サポート. 671
DB2 SQLJ の制限事項 672
Java での SQL ステートメントの組み込み 674
SQLJ におけるイテレーターの動作の宣言 674
SQLJ の例: App.sqlj 676
Java のホスト変数 680
SQLJ におけるストアード・プロシージャーおよび関数への呼び出し. 680
SQLJ プログラムのコンパイルと実行 . . 681
SQLJ 変換プログラム・オプション . . . 682
Java のストアード・プロシージャーおよび UDF 683
Java クラスを置く場所 684
Java ルーチン・クラスの更新. 685
Java でのストアード・プロシージャーのデバッグ. 685
デバッグの準備 686
デバッグ表への書き込み 687
デバッガーの呼び出し 688
Java ストアード・プロシージャーおよび UDF 689
JAR ファイルのインストール、置換、および除去 690
Java での関数定義 691
JDBC 1.2 で LOB およびグラフィカル・オブジェクトを使用する 693
JDBC および SQLJ の相互運用性 694
セッション共用 694
Java での接続リソース管理 694

Java のプログラミングに関する考慮事項

DB2 ユニバーサル・データベースは、2 つの標準ベースの Java プログラミング API をインプリメントします。 Java Database Connectivity (JDBC) および Java Embedded SQL (SQLJ) です。この章では、JDBC および SQLJ プログラミングを概説しますが、その中でも特に、DB2 に固有な面に焦点を当てています。 JDBC および SQLJ の仕様のリンクについては、 `http://www.ibm.com/software/data/db2/java/` の DB2 ユニバーサル・データベース Java Web サイトを参照してください。

© Copyright IBM Corp. 1993, 2001

SQLJ と JDBC の比較

JDBC API を使用すると、データベースに対する動的 SQL 呼び出しを行う Java プログラムを作成することができます。 SQLJ アプリケーションは、 JDBC を基礎として使用してデータベースへの接続や SQL エラー処理などのタスクを実行しますが、SQLJ ソース・ファイルに組み込み静的 SQL ステートメントを含めることもできます。 Java ソース・コードをコンパイルできるようにするには、 SQLJ 変換プログラムを使用して SQLJ ソース・ファイルを変換しなければなりません。

JDBC および SQLJ アプリケーションを作成することの詳細については、 *アプリケーション構築の手引き* を参照してください。

他の言語と比較した Java の利点

組み込み SQL を含むプログラム言語は、ホスト言語と呼ばれます。 Java は、SQL の組み込み方の点で、従来のホスト言語 C、COBOL、および FORTRAN とはかなり異なります。

- SQLJ および JDBC はオープン標準であり、他の標準互換データベース・システムから DB2 ユニバーサル・データベースに SQLJ または JDBC アプリケーションを簡単に移植することができます。
- 複合データを表すすべての Java タイプ、および可変サイズのデータには、識別値 `null` があります。これは、SQL NULL 状態を示すために使用でき、 Java プログラムでは、他のホスト言語に備えられている NULL 標識の代替になります。
- Java は、異なるプラットフォーム間でも修正なしで移植可能な (″スーパー・ポータブル″ とも ″ダウンロード可能″ とも呼ばれる) プログラムをサポートするように設計されています。 Java のクラスのタイプ・システムとインターフェースを使用して、この機能はコンポーネント・ソフトウェアを実行可能にします。特に、Java で作成された SQLJ 変換プログラムは、許可、スキーマ検査、タイプ検査、トランザクション、およびリカバリー機能などの既存のデータベース機能を利用したり、特定のデータベース用に最適化されたコードを生成したりするために、データベース・ベンダーにより専用に作成されたコンポーネントを呼び出すことができます。
- Java は、異機種ネットワークでの 2 進移行性を持つように設計されています。これは、静的 SQL を使用するデータベース・アプリケーションの 2 進移行性を保証するものです。

Java における SQL セキュリティー

デフォルトでは、JDBC プログラムは、そのプログラムを実行している人に割り当てられている特権を使用して SQL ステートメントを実行します。一方、SQLJ プログラムは、そのプログラムを作成した人に割り当てられている特権を使用して SQL ステートメントを実行します。

Java のソースおよび出力ファイル

ソース・ファイルは、次のような拡張子を持ちます。

.java Java ソース・ファイル。プリコンパイルの必要なし。 これらのファイルは、Java 開発環境に組み込まれている javac Java コンパイラーを使用してコンパイルできます。

.sqlj SQLJ ソース・ファイル。 sqlj 変換プログラムでの変換が必要です。 変換プログラムは、以下のものを作成します。

- 1 つかそれ以上の .class バイトコード・ファイル
- 接続コンテキストごとに 1 つの .ser プロファイル

対応する出力ファイルは、次のような拡張子を持ちます。

.class JDBC および SQLJ バイトコード・コンパイル済みファイル。

.ser SQLJ 順次化プロファイル・ファイル。それぞれのプロファイル・ファイルについて、 db2profc ユーティリティーを使用してデータベースにパッケージを作成します。

SQLJ プログラムのコンパイルおよび実行方法の例は、 681ページの『SQLJ プログラムのコンパイルと実行』を参照してください。

Java クラス・ライブラリー

DB2 ユニバーサル・データベースは、 JDBC および SQLJ サポートにクラス・ライブラリーを提供します。これは、CLASSPATH に指定するか、アプレットに組み込まなければなりません。以下のものがあります。

db2java.zip
JDBC ドライバーおよび JDBC および SQLJ サポート・クラス。ストアード・プロシージャーおよび UDF サポートを含む。

sqlj.zip SQLJ 変換プログラム・クラス・ファイル。

runtime.zip
SQLJ アプリケーションおよびアプレットへの Java ランタイム・サポート。

Java パッケージ

ご使用のアプリケーションの DB2 に組み込まれているクラス・ライブラリーを使用するには、該当するインポート・パッケージ・ステートメントを、ソース・ファイルの先頭に含めなければなりません。ご使用の Java アプリケーションに以下のパッケージを使用できます。

java.sql.*
JDK に組み込む JDBC API。このパッケージは、すべての JDBC および SQLJ プログラムにインポートしなければなりません。

sqlj.runtime.*
それぞれの DB2 クライアントに組み込まれている SQLJ サポート。このパッケージは、すべての SQLJ プログラムにインポートしなければなりません。

sqlj.runtime.ref.*
: それぞれの DB2 クライアントに組み込まれている SQLJ サポート。このパッケージは、すべての SQLJ プログラムにインポートしなければなりません。

Java でサポートされている SQL データ・タイプ

表32 では、 JDBC 仕様のデータ・タイプ・マッピングに基づいた、各 SQL データ・タイプに等しい Java の値を示しています。 JDBC バージョン 1.2 とバージョン 2.0 ドライバーのどちらを使用するかによって、いくつかのマッピングは異なることにも注意してください。 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・スキーマを使って変換します。これらのマッピングは、 Java アプリケーションと PARAMETER STYLE JAVA ストアード・プロシージャーと UDF とで使用します。 PARAMETER STYLE DB2GENERAL ストアード・プロシージャーおよび UDF のデータ・タイプ・マッピングの詳細については、 794ページの『サポートされる SQL データ・タイプ』を参照してください。

注: DB2 によりサポートされるどのプログラム言語にも、 DATALINK データ・タイプのホスト言語サポートはありません。

表 32. Java 宣言にマップされる SQL データ・タイプ

SQL 列名	Java データ・タイプ	SQL 列タイプ記述
SMALLINT (500 または 501)	short	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 (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)	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)
VARCHAR(*n*) (448 または 449)	String	可変長文字ストリング
LONG VARCHAR (456 または 457)	String	long 可変長文字ストリング
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)

表 32. Java 宣言にマップされる SQL データ・タイプ (続き)

SQL 列名	Java データ・タイプ	SQL 列タイプ記述
VARCHAR(*n*) FOR BIT DATA	バイト[]	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)	JDBC 1.2: バイト[] JDBC 2.0: java.sql.Blob	ラージ・オブジェクト可変長 2 進ストリング
CLOB(*n*) (408 または 409)	JDBC 1.2: String JDBC 2.0: 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)	JDBC 1.2: String JDBC 2.0: 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
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

Java の SQLSTATE および SQLCODE 値

SQL エラーが生じると、 JDBC および SQLJ プログラムは `SQLException` をスローします。 `SQLException` のインスタンスの SQLSTATE、SQLCODE、または SQLMSG 値を検索するには、対応するインスタンス・メソッドを呼び出します。以下のようにします。

SQL 戻りコード	SQLException メソッド
SQLCODE	`SQLException.getErrorCode()`
SQLMSG	`SQLException.getMessage()`
SQLSTATE	`SQLException.getSQLState()`

以下に例を示します。

```
int sqlCode=0;      // Variable to hold SQLCODE
String sqlState="00000";  // Variable to hold SQLSTATE

try
{
  // JDBC statements may throw SQLExceptions
  stmt.executeQuery("Your JDBC statement here");

  // SQLJ statements may also throw SQLExeptions
  #sql {..... your SQLJ statement here ......};
```

```
}

/* Here's how you can check for SQLCODEs and SQLSTATE */

catch (SQLException e)
{
  sqlCode = e.getErrorCode() // Get SQLCODE
  sqlState = e.getSQLState() // Get SQLSTATE

  if (sqlCode == -190 || sqlState.equals("42837"))
  {
    // Your code here to handle SQLCODE -190 or SQLSTATE 42837
  }
  else
  {
    // Your code here to handle other errors
  }
  System.err.println(e.getMessage()) ; // Print the exception
  System.exit(1);         // Exit
}
```

Java のトレース機能

CLI/ODBC/JDBC トレース機能および DB2 トレース機能の両方の db2trc を使用して、 JDBC または SQLJ プログラムに関係ある問題を診断することができます。上記のトレースを実行する方法の詳細については、 *問題判別の手引き* に説明されています。

実行時呼び出しトレース機能は、SQLJ プログラムにインストールすることもできます。このユーティリティーは、プログラムに関連するプロファイルを処理します。プログラムが App_SJProfile0 というプロファイルを使用するとします。プログラムに呼び出しトレースをインストールするには、次のコマンドを使用します。

```
profdb App_SJProfile0.ser
```

profdb ユーティリティーは、 Java 仮想マシンを使用して、クラス sqlj.runtime.profile.util.AuditorInstaller の main() 方式を実行します。 AuditorInstaller クラスの使用法およびオプションの詳細については、 http://www.ibm.com/software/data/db2/java にあるDB2 Java Web サイトにアクセスしてください。

Java アプリケーションおよびアプレットの作成

ご使用のアプリケーションやアプレットが JDBC または SQLJ のいずれを使用するかにかかわらず、 Sun Microsystems の提供する JDBC 仕様に精通している必要があります。 JDBC および SQLJ リソースのリンクについては、 http://www.ibm.com/software/data/db2/java/ の DB2 Java Web サイトを参照してください。 この仕様書は、JDBC API を呼び出してデータベースにアクセスする方法と、データベース中のデータを操作する方法を説明しています。

加えて、JDBC への DB の拡張機能とそのいくつかの制限について学ぶために、この節を読み通す必要があります (668ページの『JDBC 2.0』を参照)。 Java によって UDF または ストアード・プロシージャーの作成を計画している場合、他の言語とは異なる Java 言語特有の考慮事項が記されている 436ページの『Java ユーザー定義の関数の作成および使用』および 689ページの『Java ストアード・プロシージャーおよび UDF』 を参照してください。

JDBC および SQLJ アプリケーションおよびアプレットを作成および実行するには、*アプリケーション構築の手引き* の指示に従ってオペレーティング・システム環境を設定しなければなりません。

動作の仕組み

DB2 の Java 使用可能性には、以下の 3 つの独立コンポーネントがあります。

- JDBC を使用して DB2 にアクセスする、 Java で作成されたクライアント・アプリケーションおよびアプレットのサポート (663ページの『JDBC プログラミング』を参照)。
- SQLJ を使用して DB2 にアクセスする、 Java で作成されたクライアント・アプリケーションおよびアプレットのプリコンパイルおよびバインド・サポート (671ページの『SQLJ プログラミング』を参照)
- サーバー上の Java UDF およびストアード・プロシージャーのサポート (683ページの『Java のストアード・プロシージャーおよび UDF』を参照)。

Java のアプリケーション・サポート: 662ページの図21 は、DB2 JDBC アプリケーションが動作する方法を示しています。 DB2 JDBC アプリケーションは、DB2 CLI アプリケーションと考えることができます。そのアプリケーションを Java 言語を使って作成するだけです。 Java のネイティブな方式で、JDBC の呼び出しは DB2 CLI の呼び出しに変換されます。 JDBC は、 DB2 クライアントから DB2 CLI を介して DB2 サーバーに至る流れを要求します。

SQLJ アプリケーションはこの JDBC サポートを使用します。それに加えて、SQLJ 実行時クラスが、プリコンパイルおよびバインド段階でデータベースにバインドされる任意の SQL パッケージを認証および実行する必要があります。

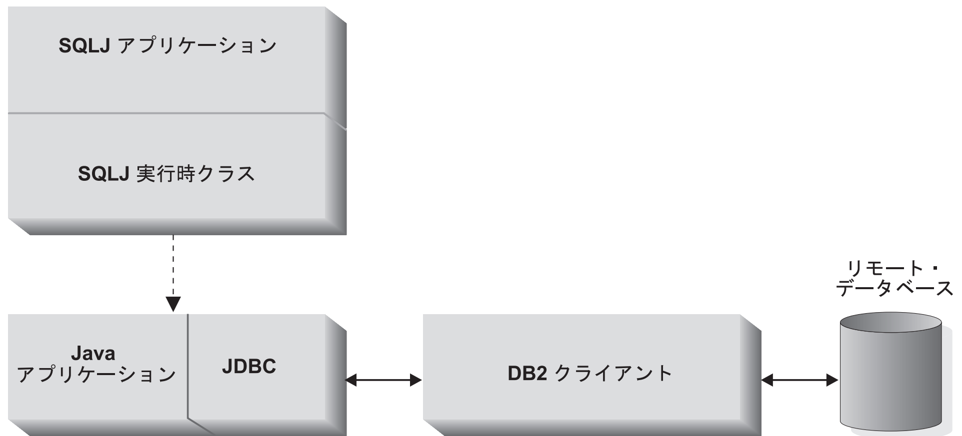


図 21. DB2 Java アプリケーションのインプリメンテーション

Java のアプレット・サポート: 663ページの図22 は、 JDBC アプレット・ドライバー (ネット・ドライバー としても知られている) が動作する方法を示しています。ドライバーは JDBC クライアントおよび JDBC サーバー (`db2jd`) から成っています。JDBC クライアント・ドライバーは、アプレットとともに Web ブラウザーにロードされます。アプレットによって DB2 データベースへの接続が要求された場合、クライアントは、Web サーバーが実行されているマシンの JDBC サーバーに対して TCP/IP ソケットをオープンします。接続が確立された後、クライアントは、TCP/IP 接続を介してアプレットから JDBC サーバーへ、後続のデータベースごとにアクセス要求を送ります。それから JDBC サーバーは、タスクを実行するために対応する CLI (ODBC) 呼び出しを行います。完了すると、 JDBC サーバーはその接続を介してクライアントに結果を送り返します。

SQLJ アプレットは、 JDBC クライアント・ドライバーの上に SQLJ クライアント・ドライバーを追加します。そうでない場合には、JDBC アプレットと同じように実行します。

DB2 JDBC サーバーの開始についての情報は、 コマンド*解説書* にある `db2jstrt` コマンドを参照してください。

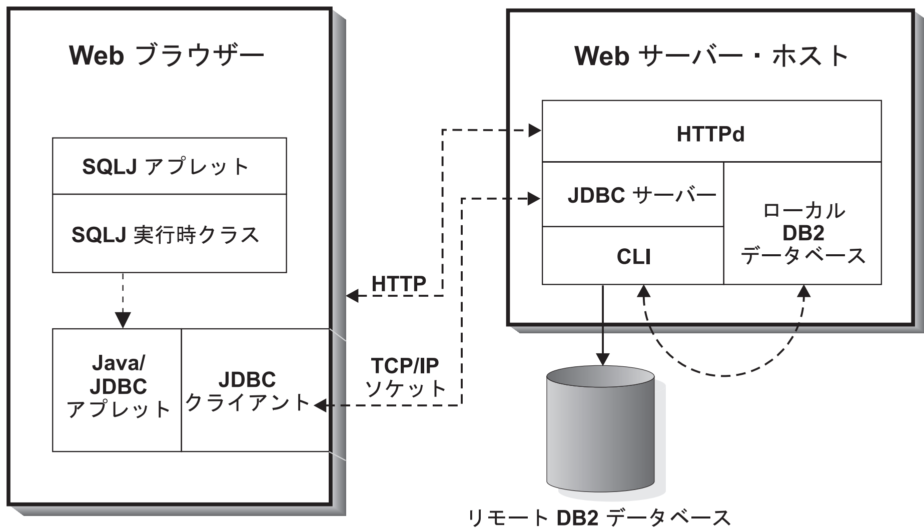


図 22. DB2 Java アプレットのインプリメンテーション

JDBC プログラミング

アプリケーションとタスクは両方とも、通常は以下のタスクを実行します。

1. 適切な Java パッケージとクラスをインポートする (`java.sql.*`)。
2. 適切な JDBC ドライバーをロードする (アプリケーションの場合、`COM.ibm.db2.jdbc.app.DB2Driver`。アプレットの場合、`COM.ibm.db2.jdbc.net.DB2Driver`)。
3. データベースへの接続。 Sun 社の JDBC 仕様書で定義されている URL でロケーションを指定し、 `db2` サブプロトコルを使用します。アプレットの場合は、アプレット・サーバー用のユーザー ID、パスワード、ホスト名、およびポート番号を指定する必要があります。明示的に代替値を指定しない場合、アプリケーションは、 DB2 クライアント・カタログからユーザー ID とパスワードのデフォルトを暗黙的に使用します。
4. SQL ステートメントをデータベースに渡す。
5. 結果を受信する。
6. 接続をクローズする。

プログラムをコーディングした後、他のすべての Java プログラムのようにコンパイルします。何らかの特別なプリコンパイルまたはバインド・ステップを実行する必要はありません。

DB2Appl プログラムの実行方法

以下のサンプル・プログラム DB2Appl.java は、 DB2 に JDBC プログラムをコーディングする方法を示しています。

1. **JDBC パッケージをインポートする。**すべての JDBC および SQLJ プログラムは、JDBC をインポートしなければなりません。
2. **接続オブジェクトを宣言する。** 接続オブジェクトは、データベース接続を確立して管理します。
3. **データベース URL 変数を設定する。** DB2 アプリケーション・ドライバーは、jdbc:db2:>*database-name*< の形式の URL を受け入れます。
4. **データベースに接続する。** DriverManager.getConnection() 方式は、以下のパラメーターを指定して使用されるのが一般的です。

   getConnection(String *url*)

   デフォルト・ユーザー ID およびパスワードを指定して、 *url* で指定されるデータベースへの接続を確立します。

   getConnection(String *url*, String *userid*, String *password*)

   userid および *passwd* でそれぞれ指定されるユーザー ID およびパスワードの値を使用して、 *url* で指定されるデータベースへの接続を確立します。
5. **ステートメント・オブジェクトを作成する。**ステートメント・オブジェクトは、SQL ステートメントをデータベースに送信します。
6. **SQL SELECT ステートメントを実行する。** SQL ステートメントに対して executeQuery() 方式を使用します。 SELECT ステートメントと同様に、これは単一の結果セットを戻します。 この結果を ResultSet オブジェクトに割り当てます。
7. **ResultSet から行を検索する。** ResultSet オブジェクトを使用すると、 SQL に組み込まれるホスト言語で結果セットをカーソルのように扱うことができます。 ResultSet.next() 方式では、次の行にカーソルを移動し、結果セットで最終行になると、boolean false を戻します。結果セット処理の制限事項は、データベース・マネージャー構成パラメーターにより使用可能にされた JDBC API のレベルによって異なります。
   - JDBC 2.0 API では、結果セットを前後にスクロールすることができます。
   - JDBC 1.2 API では、ResultSet.next() 方式で結果セットを前方にスクロールすることしか許可されていません。
8. **列の値を戻す。** ResultSet.getString(*n*) は、 n^{th} 列の値を String オブジェクトとして戻します。
9. **SQL UPDATE ステートメントを実行する。** SQL UPDATE ステートメントに対して executeUpdate() 方式を使用します。 この方式では、更新された行数を int 値として戻します。

JDBC の例: DB2Appl.java

```
import java.sql.*; 1

class DB2Appl {

   static {
      try {
         Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").newInstance();
      } catch (Exception e) {
         System.out.println(e);
      }
   }

   public static void main(String argv[]) {
      Connection con = null; 2

      // URL is jdbc:db2:dbname
      String url = "jdbc:db2:sample"; 3

      try {
         if (argv.length == 0) {
            // connect with default id/password
            con = DriverManager.getConnection(url);
            }
         else if (argv.length == 2) {
            String userid = argv[0];
            String passwd = argv[1];

            // connect with user-provided username and password
            con = DriverManager.getConnection(url, userid, passwd); 4
            }
         else {
            System.out.println("Usage: java DB2Appl [username password]");
            System.exit(0);
         }

         // retrieve data from the database
         System.out.println("Retrieve some data from the database...");
         Statement stmt = con.createStatement(); 5
         ResultSet rs = stmt.executeQuery("SELECT * from employee"); 6

         System.out.println("Received results:");

         // display the result set
         // rs.next() returns false when there are no more rows
         while (rs.next()) { 7
            String a = rs.getString(1); 8
            String str = rs.getString(2);

            System.out.print(" empno= " + a);
            System.out.print(" firstname= " + str);
            System.out.print("");
         }
```

```
          rs.close();
          stmt.close();

          // update the database
          System.out.println("Update the database... ");
          stmt = con.createStatement();
          int rowsUpdated = stmt.executeUpdate("UPDATE employee
         SET firstnme = 'SHILI' where empno = '000010'"); 9

          System.out.print("Changed "+rowsUpdated);

          if (1 == rowsUpdated)
             System.out.println(" row.");
          else
             System.out.println(" rows.");

          stmt.close();
          con.close();
       } catch( Exception e ) {
          System.out.println(e);
       }
    }
}
```

JDBC アプリケーションの配布

JDBC アプリケーションの配布は、他のすべての Java アプリケーションの場合と同様に行います。アプリケーションは DB2 サーバーとの通信に DB2 クライアントを使用するため、セキュリティーについての考慮事項は特にありません。権限に関する検査は DB2 クライアントにより実行されます。

クライアント・マシン上でアプリケーションを実行するには、そのマシンに以下のものをインストールしなければなりません。
- Java 仮想マシン (JVM)。任意の Java コードを実行するのに必要です。
- DB2 クライアント。DB2 JDBC ドライバーも含みます。

アプリケーションを作成するには、ご使用のオペレーティング・システムに JDK もインストールしなければなりません。 Java 環境の設定、DB2 Java アプリケーションの作成、および DB2 Java アプリケーションの実行の詳細については、 *アプリケーション構築の手引き* を参照してください。

JDBC アプレットの配布および実行

他の Java アプレットと同様に、 JDBC アプレットは、ネットワーク (イントラネットまたはインターネット) を介して配布します。一般的にアプレットは、ハイパーテキスト・マークアップ言語 (HTML) のページに組み込まれます。たとえば、サンプル・アプレット DB2Applt.java (sqllib/samples/java にある) を呼び出すためには、次の <APPLET> タグを使用します。

```
<applet code="DB2Applt.class" width=325 height=275 archive="db2java.zip">
   <param name="server" value="webhost">
   <param name="port" value="6789">
</applet>
```

アプレットを実行するには、クライアント・マシンに Java を使用できる Web ブラウザーだけが必要です。 HTML ページをロードするときには、アプレット・タグは、ご使用のブラウザーが Java アプレットおよび db2java.zip クラス・ライブラリーをダウンロードするように指示します。これには、COM.ibm.db2.jdbc.net クラスによりインプリメントされた DB2 JDBC ドライバーも含まれます。 アプレットが DB2 に接続するために JDBC API を呼び出す場合、 JDBC ドライバーは Web サーバーで稼働している JDBC アプレット・サーバーを介して、 DB2 データベースと分割接続します。

注: Web ブラウザーがサーバーから db2java.zip を確実にダウンロードできるようにするには、クライアント上の CLASSPATH 環境変数に db2java.zip が含まれない ようにします。クライアントが db2java.zip のローカル・バージョンを使用している場合、ご使用のアプレットは正常に機能しない場合があります。

Java アプレットの作成および配布の詳細については、 *アプリケーション構築の手引き* を参照してください。

JDBC アプレット・サーバーへの接続

Java アプレットが使用する db2java.zip ファイルが JDBC アプレット・サーバーと同じフィックスパック・レベルにあることが重要です。通常の状況では、db2java.zip は、663ページの図22 に示されているように、JDBC アプレット・サーバーが実行されている Web サーバーからロードされます。これにより一致が保証されます。ただし、構成が別の場所から db2java.zip をロードする Java アプレットを持っている場合は、不一致が起きることがあります。 DB2 バージョン 7.1 FixPak 2 以前では、これにより予期しない障害が発生する可能性がありました。 DB2 バージョン 7.1 FixPak 2 では、2 つのファイル間でのフィックスパック・レベルの一致は、接続時に厳しく強制されます。不一致が見つかると、接続は拒否され、クライアントは次の例外の 1 つを受け取ります。

- db2java.zip が DB2 バージョン 7.1 FixPak 2 以降の場合:

```
COM.ibm.db2.jdbc.DB2Exception: [IBM][JDBC Driver]
CLI0621E  Unsupported JDBC server configuration.
```

- db2java.zip が DB2 バージョン 7.1 FixPak 2 以前の場合:

```
COM.ibm.db2.jdbc.DB2Exception: [IBM][JDBC Driver]
CLI0601E  ステートメント・ハンドルが無効か、またはステートメントがクローズされました。
SQLSTATE=S1000
```

不一致が起きた場合、JDBC アプレット・サーバーは次のメッセージの 1 つを jdbcerr.log ファイルに記録します。

- JDBC アプレット・サーバーが DB2 バージョン 7.1 FixPak 2 以降の場合:

```
jdbcFSQLConnect: JDBC Applet Server and client (db2java.zip)
versions do not match. Unable to proceed with connection., einfo= -111
```

- JDBC アプレット・サーバーが DB2 バージョン 7.1 FixPak 2 以前の場合:

```
jdbcServiceConnection(): Invalid Request Received., einfo= 0
```

JDBC 2.0

JDBC 2.0 は、Sun 社の JDBC の最新のバージョンです。このバージョンの JDBC には、**コア API** と**オプション・パッケージ API** の、2 つのパーツが定義されています。JDBC 仕様の詳細については、 `http://www.software.ibm.com/data/db2/java/` にある DB2 ユニバーサル・データベース Java Web サイトを参照してください。

ご使用のオペレーティング・システムに応じて JDBC 2.0 ドライバーをインストールする方法について詳しくは、 *アプリケーション構築の手引き* を参照してください。

JDBC 2.0 コア API サポート

DB2 JDBC 2.0 ドライバーは JDBC 2.0 コア API をサポートしますが、仕様で定義されているすべての機能をサポートするわけではありません。 DB2 JDBC 2.0 ドライバーは、以下の JDBC 2.0 コア API 機能をサポートします。

- スクロール可能選択不可 ResultSet
- `java.sql.Statement`、 `java.sql.PreparedStatement`、および `java.sql.CallableStatement` のバッチ更新
- `java.sql.Blob` サポート
- `java.sql.Clob` サポート

DB2 JDBC 2.0 ドライバーは、以下の機能はサポートしていません。

- 更新可能スクロール可能結果セット
- 新規の SQL タイプ (配列、参照、特殊、Java オブジェクト、構造)
- カスタマイズ済み SQL タイプ・マッピング
- Java ストアード・プロシージャーの java.sql.Blob または java.sql.Clob、 UDF、あるいはメソッド
- スクロール可能選択可能 ResultSets (スクロール・タイプ ResultSet.TYPE_SCROLL_SENSITIVE)
- ResultSet.setFetchDirection(int) (無視、例外をスローしない)
- ResultSet.setFetchSize(int) (無視、例外をスローしない)
- Statement.setFetchSize(int) (無視、例外をスローしない)
- ResultSet.getTime(int, Calendar)
- ResultSet.getTimestamp(int, Calendar)
- CallableStatement.getClob()
- CallableStatement.getBlob()

JDBC 2.0 オプション・パッケージ API サポート

DB2 JDBC 2.0 ドライバーは、以下の JDBC 2.0 オプション・パッケージ API 機能をサポートします。

命名データベース用 Java Naming and Directory Interface (JNDI): DB2 では、命名データベース用 Java Naming and Directory Interface (JNDI) の以下のサポートが提供されています。

javax.naming.Context

このインターフェースは、 DataSource オブジェクトのストレージと検索を処理する COM.ibm.db2.jndi.DB2Context によってインプリメントされます。論理データ・ソース名と物理データベース情報 (データベース名など) の永続的な関連をサポートするため、これらの関連は .db2.jndi という名前のファイルに保管されます。アプリケーションの場合、このファイルは USER.HOME 環境変数で指定したディレクトリーに常駐します (または、存在していなければ作成されます)。アプレットの場合、lookup() 操作を円滑に行うためには、このファイルを Web サーバーのルート・ディレクトリーに作成する必要があります。アプレットは、このクラスの bind()、 rebind()、unbind()、および rename() メソッドはサポートしません。アプリケーションが行うことができるのは、 DataSource オブジェクトの JNDI へのバインドだけです。

javax.sql.Datasource

このインターフェースは、 COM.ibm.db2.jdbc.DB2DataSource によってインプリメントされます。このクラスのオブジェクトは、 javax.naming.Context をインプリメントする際に保管することができます。このクラスは、接続プールのサポートにも利用できます。

| DB2DataSource は、以下のメソッドをサポートします。

| • public void setDatabaseName(String databaseName)
| • public void setServerName(String serverName)
| • public void setPortNumber(int portNumber)

javax.naming.InitialContextFactory

このインターフェースは、 DB2Context のインスタンスを作成する COM.ibm.db2.jndi.DB2InitialContextFactory によってインプリメントされます。 アプリケーションは、JAVA.NAMING.FACTORY.INITIAL 環境変数の値を自動的に COM.ibm.db2.jndi.DB2InitialContextFactory に設定します。このクラスをアプレットで使用するには、以下の構文を使用して InitialContext() を呼び出します。

```
Hashtable env = new Hashtable( 5 );
env.put( "java.naming.factory.initial",
         "COM.ibm.db2.jndi.DB2InitialContextFactory" );
Context ctx = new InitialContext( env );
```

接続プール: DB2ConnectionPoolDataSource および DB2PooledConnection は、ユーザー独自の接続プールをインプリメントするのに必要なフックを提供します。以下のとおりです。

javax.sql.ConnectionPoolDataSource
このインターフェースは、 COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource によってインプリメントされ、 COM.ibm.db2.jdbc.DB2PooledConnection オブジェクトを制作する場所になります。

javax.sql.PooledConnection
このインターフェースは、 COM.ibm.db2.jdbc.DB2PooledConnection によってインプリメントされます。

Java トランザクション API (JTA): DB2 は、DB2 JDBC アプリケーション・ドライバーを使って Java トランザクション API (JTA) をサポートします。 DB2 は、DB2 JDBC ネット・ドライバーを使った JTA はサポートしません。

javax.sql.XAConnection
このインターフェースは、 COM.ibm.db2.jdbc.DB2XAConnection によってインプリメントされます。

javax.sql.XADataSource
このインターフェースは、 COM.ibm.db2.jdbc.DB2XADataSource によってインプリメントされ、 COM.ibm.db2.jdbc.DB2PooledConnection オブジェクトを制作する場所になります。

javax.transactions.xa.XAResource
このインターフェースは、 COM.ibm.db2.jdbc.app.DBXAResource によってインプリメントされます。

javax.transactions.xa.Xid
このインターフェースは、 COM.ibm.db2.jdbc.DB2Xid によってインプリメントされます。

JDBC 2.0 互換性

この仕様のバージョンは、以前のバージョン (1.2) と下位互換性があります。ただし、DB2 JDBC 1.2 ドライバーは、 LOB タイプを JDBC 1.2 仕様の拡張としてサポートします。この拡張は、新しい仕様の下位互換性には含まれません。つまり、JDBC 1.2 ドライバーの LOB サポートに依存している既存の JDBC アプリケーションは、新しいドライバーでは正常に実行しない可能性があるということです。 LOB およびグラフィック・タイプの DB2 JDBC 1.2 ドライバー・サポートの詳細については、 693ページの『JDBC 1.2 で LOB およびグラフィカル・オブジェクトを使用する』を参照してください。問題を訂正するため、 JDBC 2.0 ドライバーが提供する LOB サポートを利用するようアプリケーションを変更することを検討してください。

注: ストアード・プロシージャーや UDF の LOB および漢字タイプ用の DB2 JDBC 2.0 ドライバー・サポートは使用できません。ストアード・プロシージャーや UDF の LOB または漢字タイプを使用するには、 JDBC 1.2 ドライバー・サポートを使用する必要があります。

SQLJ プログラミング

DB2 SQLJ サポートは、SQLJ ANSI 標準をベースにしています。 ANSI Web サイトおよびその他の SQLJ リソースへのポインターについての詳細は、`http://www.ibm.com/software/data/db2/java` にある DB2 Java Web サイトにアクセスしてください。 この章には、 SQLJ プログラミングの概説および DB2 SQLJ サポートに固有の情報が含まれます。

以下の SQL 構成が SQLJ プログラムに含まれています。

- 照会; たとえば、SELECT ステートメントおよび式。
- SQL データ変更ステートメント (DML); たとえば、 INSERT、 UPDATE、 DELETE。
- データ・ステートメント; たとえば、FETCH、SELECT..INTO。
- トランザクション制御; たとえば、COMMIT、ROLLBACK など。
- データ定義言語 (DDL、スキーマ操作言語とも呼ばれる); たとえば、CREATE、DROP、ALTER。
- ストアード・プロシージャーへの呼び出し; たとえば、CALL MYPROC(:x, :y, :z)
- 関数の呼び出し; たとえば、VALUES(MYFUN(:x))

DB2 SQLJ サポート

DB2 SQLJ サポートは、DB2 アプリケーション開発クライアントで提供されます。DB2 クライアントが提供する JDBC サポートとともに DB2 SQLJ サポートを使用すると、 Java Embedded SQL アプリケーション、アプレット、ストアード・プロシージャー、およびユーザー定義関数 (UDF) を作成、構築、および実行できるようにします。これらには静的 SQL が含まれ、 DB2 データベースにバインドされている組み込み SQL を使用します。

DB2 アプリケーション開発クライアントが提供する SQLJ サポートには、以下のものが含まれます。

- SQLJ 変換プログラム、`sqlj`。 SQLJ プログラムにある組み込み SQL を Java ソース・ステートメントで置換し、 SQLJ プログラムで見つかった SQL 操作についての情報を含む直列化プロファイルを生成します。 SQLJ 変換プログラムは、`sqllib/java/sqlj.zip` ファイルを使用します。
- SQLJ 実行時クラス。 `sqllib/java/runtime.zip` で使用可能です。

- DB2 SQLJ プロファイル・カスタマイザー、db2profc。生成されたプロファイルに保管される SQL ステートメントをプリコンパイルし、 DB2 データベースにパッケージを生成します。
- DB2 SQLJ プロファイル・プリンター、db2profp。カスタマイズ済みの DB2 プロファイルの目次を平文で印刷します。
- SQLJ プロファイル監査プログラム・インストーラー profdb。デバッグするクラス監査プログラムを 2 進プロファイルの既存のセットにインストール (またはアンインストール) します。いったんインストールされると、アプリケーションの実行時に行われる RTStatement と RTResultSet の呼び出しはすべて、ファイル (または標準出力) のログに記録されます。次いで、予期される振る舞いやトレース・エラーが検査されます。実行時に基本 RTStatement および RTResultSet 呼び出しインターフェースに対して行われる呼び出しだけが監査されることに注意してください。
- SQLJ プロファイル変換ツール、profconv。直列化プロファイルのインスタンスをクラス・バイトコード書式に変換する。ブラウザーの中には、アプレットに関連するリソース・ファイルから直列化オブジェクトをロードするためのサポートがないものもあります。対処方法として、このユーティリティーを実行して変換を実行する必要があります。

db2profc および db2profp コマンドの詳細については、 *コマンド解説書* を参照してください。 SQLJ 実行時クラスについての詳細は、 http://www.ibm.com/software/data/db2/java にある DB2 Java Web サイトにアクセスしてください。

DB2 SQLJ の制限事項

SQLJ を使用して DB2 アプリケーションを作成する際には、以下の制限事項に注意してください。

- DB2 SQLJ サポートは、 SQL ステートメントの発行に関する標準 DB2 ユニバーサル・データベースの制限事項に準拠します。
- 位置指定された UPDATE および DELETE ステートメントは、複合 SQL ステートメントでは有効なサブステートメントではありません。
- プリコンパイル・オプション ″DATETIME″ はサポートされていません。国際標準化機構規格の日付および時刻形式しかサポートされていません。
- プリコンパイル・オプション ″PACKAGE USING package-name″ は、変換プログラムによって生成されるパッケージの名前を指定します。名前が入力されない場合には、プロファイルの名前 (拡張子は除く。すべて大文字に変換して表記) が使用されます。最大長は 8 文字です。 SQLJ プロファイル名には接尾部 _SJProfile*N* があるため (*N* はプロファイル・キー番号)、プロファイル名は常に 8 文字より長くなります。デフォルト・パッケージ名は、プロファイル番号の最初の (8 - *pfKeyNumLen*) 文字にプロファイル・キー番号をつなげて作成されます。ここで、*pfKeyNumLen* はプロファイル名のプロファイル・キー番号の長さです。プロファイル・キー番号の長さが 7 文字より長い場合には、最後の 7 桁が警告なしで使用されます。以下に例を示します。

```
profile name                        default package name
---------------------               --------------------
App_SJProfile1                      App_SJP1
App_SJProfile123                    App_S123
App_SJProfile1234567                A1234567
App_SJProfile12345678               A2345678
```

- java.math.BigDecimal ホスト変数が使用される場合、そのホスト変数の精度および位取りはアプリケーションの変換中は使用できません。 10 進数ホスト変数の精度および位取りが、これを使用しているステートメントの内容からはっきり判別できない場合には、 CAST を使用して精度および位取りを指定することができます。
- タイプ java.math.BigInteger の Java 変数は、 SQL ステートメントではホスト変数として使用できません。

ブラウザーの中には、アプレットに関連するリソース・ファイルから直列化オブジェクトをロードするためのサポートがないものもあります。これらのブラウザーでアプレット Applt をロードしようとすると、次のエラー・メッセージが出されます。

```
java.lang.ClassNotFoundException: Applt_SJProfile0
```

対処方法として、直列化プロファイルを Java クラス形式で保管されたプロファイルに変換するユーティリティーがあります。このユーティリティーは、sqlj.runtime.profile.util.SerProfileToClass と呼ばれる Java クラスです。これは、直列化プロファイルのリソース・ファイルを入力として使用し、プロファイルを含む Java クラスを出力として生成します。プロファイルを変換するには、以下のコマンドを使用します。

```
profconv Applt_SJProfile0.ser
```

または

```
java sqlj.runtime.profile.util.SerProfileToClass Applt_SJProfile0.ser
```

結果として、クラス Applt_SJProfile0.class が作成されます。アプレットによって使用される .ser 形式のすべてのプロファイルを、.class 形式のプロファイルで置換します。

SQLJ アプレットには、 db2java.zip および runtime.zip ファイルの両方が必要です。すべてのアプレット・クラス (db2java.zip および runtime.zip の中のクラス) を、単一の Jar ファイルにパッケージしないように選択した場合には、 db2java.zip と runtime.zip (コンマで区切る) の両方を "applet" タグのアーカイブ・パラメーターに入れます。アーカイブ・タグで複数の zip ファイルをサポートしないブラウザーの場合は、そのタグで db2java.zip を指定し、ご使用の Web ブラウザーにアクセスできる作業ディレクトリーで、現在のアプレット・クラスを指定して runtime.zip を unzip します。

Java での SQL ステートメントの組み込み

SQLJ での静的 SQL ステートメントは、 *SQLJ 文節* に表示されます。 SQLJ 文節は、Java プログラム内の SQL ステートメントがデータベースと通信するための機構です。

SQLJ 変換プログラムは以下のような構造になっているため、 SQLJ 文節および SQL ステートメントを認識します。

- SQLJ 文節は、トークン #sql で始まる
- SQLJ 文節は、セミコロンで終了する

最も単純な SQLJ 文節は、*実行可能文節* であり、トークン #sql の後に、括弧で囲まれた SQL ステートメントが続いた形で構成されます。たとえば、Java ステートメントが正常に表示されるところでは、必ず以下の SQLJ 文節が表示されます。これは、TAB という名前の表のすべての行を削除することを目的としています。

```
#sql { DELETE FROM TAB };
```

SQLJ 実行可能文節では、括弧の中に表示されるトークンは、ホスト変数を除けば SQL トークンです。ホスト変数はすべてコロン文字で区別されているため、変換プログラムはこれを識別できます。 SQL トークンは、SQLJ 実行可能文節の括弧の外側には表示されません。たとえば、以下の Java メソッドでは SQL 表に引き数を挿入します。メソッド本体は、ホスト変数 *x*、*y*、および *z* を含む SQLJ 実行可能文節で構成されています。

```
void m (int x, String y, float z) throws SQLException
{
    #sql { INSERT INTO TAB1 VALUES (:x, :y, :z) };
}
```

一般に、SQL トークンは大文字小文字を区別しないため (ただし、二重引用符で区切られた ID を除く)、大文字だけ、小文字だけ、または大文字と小文字を混ぜて使用することができます。一方、Java トークンは大文字小文字を区別します。違いをわかりやすくするために、例では、大文字小文字を区別しない SQL トークンは大文字で、 Java トークンは小文字または大文字と小文字の混合で表示します。この章全体では、小文字の null は Java の "null" 値を指し、大文字の NULL は SQL の null 値を指します。

SQLJ におけるイテレーターの動作の宣言

データを表から検索する SQL ステートメントとは異なり位置指定された UPDATE および DELETE 操作を実行するアプリケーションや、 holdability または returnability 属性を指定したイテレーターを使用するアプリケーションには、 2 つの Java ソース・ファイルが必要です。 1 つのソース・ファイルでイテレーターを public と宣言します。適当であれば、with および implements 節を付加します。

holdability または returnability 属性の値を設定するには、対応する属性用の with 節を使用してイテレーターを宣言する必要があります。以下の例では、holdability 属性を、イテレーター WithHoldCurs に応じて true に設定します。

```
#sql public iterator WithHoldCurs with (holdability=true) (String EmpNo);
```

位置指定した更新を実行するイテレーターには、 sqlj.runtime.ForUpdate インターフェースをインプリメントする implements 節が必要です。たとえば、次のようなイテレーター DelByName を file1.sqlj で宣言するとします。

```
#sql public iterator DelByName implements sqlj.runtime.ForUpdate(String EmpNo);
```

すると、変換しコンパイルされたイテレーターを別のソース・ファイルで使用することができます。イテレーターを使用するには、次のようにします。

1. 生成されたイテレーター・クラスのインスタンスを宣言する
2. 位置指定された UPDATE または DELETE の SELECT ステートメントをイテレーター・インスタンスに割り当てる
3. イテレーターを使用して、位置指定された UPDATE または DELETE ステートメントを実行する

file2.sqlj の位置指定された DELETE に DelByName を使用するには、 『位置指定されたイテレーターを使用して行を削除する』にあるようなステートメントを実行してください。

```
  {
    DelByName deliter; // Declare object of DelByName class
    String enum;
1   #sql deliter = { SELECT EMPNO FROM EMP WHERE WORKDEPT='D11'};
    while (deliter.next())
    {
2     enum = deliter.EmpNo(); // Get value from result table
3     #sql { DELETE WHERE CURRENT OF :deliter };
      // Delete row where cursor is positioned
    }
  }
```

注:

1. **1** この SQLJ 文節は SELECT ステートメントを実行し、 SELECT ステートメントの結果テーブルを含むイテレーター・オブジェクトを構成してから、変数 *deliter* にイテレーター・オブジェクトを割り当てます。
2. **2** このステートメントは、次に削除される行にイテレーターを位置指定します。
3. **3** この SQLJ 文節は、位置指定された DELETE を実行します。

SQLJ の例: App.sqlj

次の SQLJ アプリケーション、App.sqlj では、静的 SQL を使用して DB2 サンプル・データベースの EMPLOYEE 表からデータを検索および更新します。

1. **イテレーターを宣言する。** この節では、次の 2 つのタイプのイテレーターを宣言します。

   App_Cursor1
   : 列データ・タイプおよび名前を宣言し、列名に従って列の値を戻します (列名指定バインド)。

   App_Cursor2
   : 列データ・タイプおよび名前を宣言し、列位置ごとに列の値を戻します (列位置指定バインド)。

2. **イテレーターを初期化する。** イテレーター・オブジェクト `cursor1` は、照会の結果を使用して初期化されます。 照会の結果は、`cursor1` に保管されます。
3. **イテレーターを次の行に拡張する。** `cursor1.next()` メソッドは、検索する行がなくなると、ブール `false` を戻します。
4. **データを移動する。** 指定されたアクセス機構メソッド `empno()` は、現在の行に `empno` という列の値を戻します。指定されたアクセス機構メソッド `firstnme()` は、現在の行に `firstnme` という列の値を戻します。
5. **ホスト変数に渡すデータを SELECT する。** SELECT ステートメントは、表の行数をホスト変数 *count1* に渡します。
6. **イテレーターを初期化する。** イテレーター・オブジェクト *cursor2* は、照会の結果を使用して初期化されます。 照会の結果は、*cursor2* に保管されます。
7. **データを取り出す。** FETCH ステートメントは、 ByPos カーソルで宣言された最初の列の現行の値を結果テーブルからホスト変数 *str2* に戻します。
8. **FETCH..INTO ステートメントが正常に実行されたことを検査する。** イテレーターが行に位置指定されない場合、つまり行の取り出しの最後の試みが失敗した場合、`endFetch()` メソッドはブール `true` を戻します。 行の取り出しの最後の試みが成功した場合には、 `endFetch()` メソッドは `false` を戻します。 `next()` メソッドが呼び出されると、 DB2 は行を取り出そうとします。 FETCH...INTO ステートメントは、暗黙的に `next()` メソッドを呼び出します。
9. **イテレーターをクローズする。** `close()` メソッドは、イテレーターによって保留にされているリソースを解放します。システム・リソースが適宜解放されるようにするためには、イテレーターを明示的にクローズする必要があります。

JDBC の例: App.sqlj:

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator App_Cursor1 (String empno, String firstnme) ; 1
#sql iterator App_Cursor2 (String) ;
```

```
class App
{

  /**********************
   **  Register Driver **
   **********************/

 static
 {
   try
   {
     Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").newInstance();
   }
   catch (Exception e)
   {
     e.printStackTrace();
   }
 }

  /********************
   **      Main      **
   ********************/

 public static void main(String argv[])
 {
   try
   {
     App_Cursor1 cursor1;
     App_Cursor2 cursor2;

     String str1 = null;
     String str2 = null;
     long   count1;

     // URL is jdbc:db2:dbname
     String url = "jdbc:db2:sample";

     DefaultContext ctx = DefaultContext.getDefaultContext();
     if (ctx == null)
     {
       try
       {
         // connect with default id/password
         Connection con = DriverManager.getConnection(url);
         con.setAutoCommit(false);
         ctx = new DefaultContext(con);
       }
       catch (SQLException e)
       {
         System.out.println("Error: could not get a default context");
         System.err.println(e) ;
         System.exit(1);
       }
       DefaultContext.setDefaultContext(ctx);
```

```
}

// retrieve data from the database
System.out.println("Retrieve some data from the database.");
#sql cursor1 = {SELECT empno, firstnme FROM employee}; 2

// display the result set
// cursor1.next() returns false when there are no more rows
System.out.println("Received results:");
while (cursor1.next()) 3
{
  str1 = cursor1.empno(); 4
  str2 = cursor1.firstnme();

  System.out.print (" empno= " + str1);
  System.out.print (" firstname= " + str2);
  System.out.print ("");
}
cursor1.close(); 9

// retrieve number of employee from the database
#sql { SELECT count(*) into :count1 FROM employee }; 5
if (1 == count1)
  System.out.println ("There is 1 row in employee table");
else
  System.out.println ("There are " + count1
                      + " rows in employee table");

// update the database
System.out.println("Update the database. ");
#sql { UPDATE employee SET firstnme = 'SHILI' WHERE empno = '000010' };

// retrieve the updated data from the database
System.out.println("Retrieve the updated data from the database.");
str1 = "000010";
#sql cursor2 = {SELECT firstnme FROM employee WHERE empno = :str1}; 6

// display the result set
// cursor2.next() returns false when there are no more rows
System.out.println("Received results:");
while (true)
{
  #sql { FETCH :cursor2 INTO :str2 }; 7
  if (cursor2.endFetch()) break; 8

  System.out.print (" empno= " + str1);
  System.out.print (" firstname= " + str2);
  System.out.print ("");
}
cursor2.close(); 9

// rollback the update
System.out.println("Rollback the update.");
#sql { ROLLBACK work };
System.out.println("Rollback done.");
```

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

Java のホスト変数

組み込み SQL ステートメントへの引き数は、ホスト変数 を介して渡されます。ホスト変数とは、SQL ステートメントに表示されるホスト言語の変数です。ホスト変数は、次の 3 つの部分に分けられます。

- コロン接頭部。:。
- オプションのパラメーター・モード ID。IN、OUT、または INOUT。
- Java ホスト変数。これは、パラメーター、変数、またはフィールドの Java ID です。

Java プログラムでは Java ID を評価しても副次作用がないため、 SQLJ 文節を置換するために生成される Java コードに Java ID が何度も表示されることがあります。

次の照会には、ホスト変数 *:x* が含まれています。これは、照会を含む効力範囲に表示される Java 変数、フィールド、またはパラメーター *x* です。

```
SELECT  COL1,  COL2  FROM  TABLE1  WHERE  :x  > COL3
```

複合 SQL で指定されるホスト変数はすべて、デフォルトの入力ホスト変数です。これを出力ホスト変数としてマークするには、ホスト変数の前にパラメーター・モード ID OUT または INOUT を指定する必要があります。以下に例を示します。

```
#sql {begin compound atomic static
     select count(*) into :OUT count1 from employee;
     end compound}
```

SQLJ におけるストアード・プロシージャーおよび関数への呼び出し

データベースには、*ストアード・プロシージャー* 、*ユーザー定義関数*、および*ユーザー定義メソッド* が入っています。ストアード・プロシージャー、ユーザー定義関数、およびユーザー定義メソッドは、データベースで実行する名前付きスキーマです。 Java ステートメントとして表示される SQLJ 実行可能文節は、以下のようにして 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します。

```
#sql { CALL SOME_PROC(:INOUT myarg) };
```

ストアード・プロシージャーには、IN、OUT、または INOUT パラメーターがあります。上記の場合、ホスト変数 *myarg* の値は、その文節を実行すると変更されます。
SQLJ 実行可能節は、SQL VALUES 構成を用いて関数を呼び出します。たとえば、整数を戻す関数 F があるとします。以下の例では、関数を呼び出して、その結果を Java ローカル変数 *x* に割り当てます。

```
{
    int x;
    #sql x = { VALUES( F(34) ) };
}
```

SQLJ プログラムのコンパイルと実行

プログラム名 MyClass の SQLJ プログラムを実行するには、以下のようにします。

1. Java ソース・コードを組み込み SQL で変換して、 Java ソース・コード MyClass.java およびプロファイル MyClass_SJProfile0.ser、 MyClass_SJProfile1.ser、... (それぞれの接続コンテキストごとに 1 つのプロファイル) を生成します。

   ```
   sqlj MyClass.sqlj
   ```

   sqlj.properties ファイルを指定せずに SQLJ 変換プログラムを使用すると、変換プログラムは以下の値を使用します。

   ```
   sqlj.url=jdbc:db2:sample
   sqlj.driver=COM.ibm.db2.jdbc.app.DB2Driver
   sqlj.online=sqlj.semantics.JdbcChecker
   sqlj.offline=sqlj.semantics.OfflineChecker
   ```

   sqlj.properties ファイルを指定する場合は、以下のオプションが設定されていることを確認してください。

   ```
   sqlj.url=jdbc:db2:dbname
   sqlj.driver=COM.ibm.db2.jdbc.app.DB2Driver
   sqlj.online=sqlj.semantics.JdbcChecker
   sqlj.offline=sqlj.semantics.OfflineChecker
   ```

   ここで、*dbname* はデータベースの名前です。また、コマンド行でこれらのオプションを指定することもできます。たとえば、MyClass の変換時にデータベース mydata を指定するには、次のコマンドを実行できます。

   ```
   sqlj -url=jdbc:db2:mydata MyClass.sqlj
   ```

   -compile=false 節でコンパイル・オプションの設定を明示的にオフにしないかぎり、 SQLJ 変換プログラムは変換されたソース・コードをクラス・ファイルに自動的にコンパイルします。

2. 生成されたプロファイルに DB2 SQLJ カスタマイザーをインストールし、 DB2 データベース *dbname* に DB2 パッケージを作成します。

   ```
   db2profc -user=user-name -password=user-password -url=jdbc:db2:dbname
     -prepoptions="bindfile using MyClass0.bnd package using MyClass0"
     MyClass_SJProfile0.ser
   db2profc -user=user-name -password=user-password -url=jdbc:db2:dbname
     -prepoptions="bindfile using MyClass1.bnd package using MyClass1"
     MyClass_SJProfile1.ser
   ...
   ```

3. SQLJ プログラムを実行します。

   ```
   java MyClass
   ```

変換プログラムは、 SQLJ プロファイルがカスタマイズされるデータベースの SQL 構文を生成します。たとえば、以下のようにします。

```
i = { VALUES (  F(:x) ) };
```

が SQLJ 変換プログラムによって変換され、

```
? = VALUES (F (?))
```

として、生成されるプロファイルに保管されます。 DB2 ユニバーサル・データベースに接続すると、 DB2 は、VALUE ステートメントを次のようにカスタマイズします。

```
VALUES(F(?)) INTO ?
```

一方、DB2 ユニバーサル・データベース (OS/390 版) データベースに接続すると、DB2 は VALUE ステートメントを次のようにカスタマイズします。

```
SELECT F(?) INTO ? FROM SYSIBM.SYSDUMMY1
```

DB2 ユニバーサル・データベースに対して DB2 SQLJ プロファイル・カスタマイザー `db2profc` を実行して、バインド・ファイルを生成する場合、バインド・ファイルに `VALUES` 文節があるときには、バインド・ファイルを使用して DB2 (OS/390 版) データベースにバインドすることはできません。これは、DB2 (OS/390 版) データベースに対してバインド・ファイルを生成し、これを DB2 ユニバーサル・データベースにバインドしようとする場合にも適用されます。

DB2 SQLJ プログラムの作成および実行の詳細については、 *アプリケーション構築の手引き* を参照してください。

SQLJ 変換プログラム・オプション

SQLJ 変換プログラムは、 DB2 PRECOMPILE コマンドと同じプリコンパイル・オプションをサポートします。ただし、以下のものは例外です。

```
CONNECT
DISCONNECT
DYNAMICRULES
NOLINEMACRO
OPTLEVEL
OUTPUT
SQLCA
SQLFLAG
SQLRULES
SYNCPOINT
TARGET
WCHARTYPE
```

SQLJ 変換プログラムによって生成されたプロファイルの目次を平文で印刷するには、次のようにして `profp` ユーティリティーを使用します。

```
profp MyClass_SJProfile0.ser
profp MyClass_SJProfile1.ser
  ...
```

DB2 でカスタマイズされたプロファイルのバージョンの目次を平文で印刷するには、次のようにして db2profp ユーティリティーを使用します。ここで、*dbname* はデータベースの名前です。

```
db2profp -user=user-name -password=user-password -url=jdbc:db2:dbname
         MyClass_SJProfile0.ser
db2profp -user=user-name -password=user-password -url=jdbc:db2:dbname
         MyClass_SJProfile1.ser
    ...
```

Java のストアード・プロシージャーおよび UDF

Java のストアード・プロシージャーおよび UDF は、他のプログラミング言語と同じように作成し、使用できます。 Java コードを作成する際に知っている必要のある、プログラミングに関する考慮事項があります (691ページの『Java での関数定義』を参照)。 Java ストアード・プロシージャーおよび UDF を登録する 必要もあります。ストアード・プロシージャーを登録する方法の詳細については、 201ページの『第7章 ストアード・プロシージャー』を参照してください。 UDF を登録する方法の詳細については、*SQL 解説書* の CREATE FUNCTION ステートメントを参照してください。

サーバー上で UDF およびストアード・プロシージャーを実行するため、 DB2 は JVM を呼び出します。データベースを始動する前に、 DB2 サーバーに適切な Java Development Kit (JDK) または Java Runtime Environment をインストールし、構成するようにしてください。

JVM の実行時ライブラリーは、システム探索パスで使用可能になっていなければなりません (PATH または LIBPATH または LD_LIBRARY_PATH、および CLASSPATH)。 Java 環境設定の詳細については、 *アプリケーション構築の手引き* を参照してください。

DB2 は、Java UDF またはストアード・プロシージャーが最初に呼び出された時に、JVM をロードまたは開始します。 NOT FENCED を使用した UDF およびストアード・プロシージャーの場合、 DB2 はデータベース・インスタンスにつき 1 回 JVM をロードし、データベース・エンジンのアドレス・スペース内部で最善のパフォーマンスになるように実行します。 FENCED を使用した UDF の場合、 DB2 は db2udf プロセス内でまったく別の JVM を使用します。同様に、FENCED を使用したストアード・プロシージャーは、 db2dari プロセス内でまったく別の JVM を使用します。どの場合にも、JVM は組み込み処理が終了するまでロードされたままです。

注: ローカル・クライアントのあるデータベース・サーバーのノード・タイプを実行している場合、 MAXDARI `maxdari` データベース・マネージャー 構成パラメーターをゼロ以外の値に設定してから、 Java ストアード・プロシージャーを呼び出す必要があります。

`sqllib/samples/java` ディレクトリーで提供されている Java ストアード・プロシージャーの例を調べることができます。 DB2 に含まれているサンプル・プログラムのリストについては、 765ページの『付録B. サンプル・プログラム』を参照してください。

ストアード・プロシージャーまたは UDF をインプリメントするのに使用するすべての 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 ファイルか、または オペレーティング・システムの正しいストアード・プロシージャーまたは UDF パスに常駐する必要があることに注意してください (『Java クラスを置く場所』を参照)。

注: 混合コード・ページのデータベース・サーバーでは、 Java ユーザー定義関数およびストアード・プロシージャーは、 CLOB タイプ引き数を使用できません。なぜなら、ラージ混合コード・ページ・ストリングの文字境界へのランダム・アクセスがインプリメントされていないためです。 SBCS データベースの場合は、すべての LOB タイプが完全にサポートされます。一方、混合データベースの場合は、 BLOB および DBCLOB タイプがサポートされます。対処方法として、混合データベース・システムで実行するアプリケーションは、 CLOB 引き数を DBCLOB、LONG VARGRAPHIC、または LONG VARCHAR タイプに変換しなければなりません。 UDF の場合は、CAST 演算子を使用してこれを実行できます。

Java クラスを置く場所

個々の Java クラス・ファイルをストアード・プロシージャーおよび UDF に使用するか、またはクラス・ファイルを JAR ファイルに収集してからデータベースにその JAR ファイルをインストールするかを選択できます。 JAR ファイルを使用することにした場合には、 689ページの『Java ストアード・プロシージャーおよび UDF』で詳しい指示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する場合には、 CALL SQLJ.REFRESH_CLASSES() ステートメントを発行して、 DB2 が更新された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() について詳しくは、 685ページの『Java ルーチン・クラスの更新』を参照してください。

DB2 がストアード・プロシージャーおよび UDF を検出し、使用できるようにするには、該当するクラス・ファイルを*関数ディレクトリー* に保管する必要があります。これは、ご使用のオペレーティング・システムに以下のように定義されたディレクトリーです。

UNIX オペレーティング・システム

`sqllib/function`

OS/2 または Windows 32 ビット・オペレーティング・システム

instance_name¥function。ここで、*instance_name* は *DB2INSTPROF* インスタンス固有のレジストリー設定の値を表します。

たとえば、DB2 が `C:¥sqllib` ディレクトリーにインストールされており、*DB2INSTPROF* レジストリー設定が指定されていない Windows NT Server の関数ディレクトリーは、次のようになります。

```
C:¥sqllib¥function
```

個々のクラス・ファイルを使用することにした場合には、クラス・ファイルをご使用のオペレーティング・システムに該当するディレクトリーに保管する必要があります。Java パッケージの一部分としてクラスを宣言した場合、関数ディレクトリーの下に対応するサブディレクトリーを作成し、そこにファイルを置いてください。たとえば、Linux システム用のクラス `ibm.tests.test1` を作成する場合、対応する Java バイトコード・ファイル (test1.class) は sqllib/function/ibm/tests に保管してください。

DB2 を起動する JVM は、Java ファイルを見つけるのに、 CLASSPATH 環境変数を使用します。 DB2 は、関数ディレクトリーおよび sqllib/java/db2java.zip を CLASSPATH 設定の前に追加します。

JVM が Java クラス・ファイルを検索できるように環境を設定するには、 `jdk11_path` 構成パラメーターを設定してください。そうしない場合、デフォルトが使用されます。また、`java_heap_sz` 構成パラメーターを設定して、アプリケーションのためにヒープ・サイズを増やす必要があります。構成パラメーターの詳細については、*管理の手引き* を参照してください。

Java ルーチン・クラスの更新

Java ルーチン・クラスを更新する場合には、 DB2 に新しいクラスをロードさせるため、 SQLJ.REFRESH_CLASSES() ステートメントも発行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場合、 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、クラスを最新表示します。

注: データベース・マネージャーを停止および再始動しないで、 NOT FENCED ルーチンを更新することができます。

Java でのストアード・プロシージャーのデバッグ

DB2 は、 AIX または Windows NT Server 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を提供します。デバッグを呼び出す最も簡単な方法は、 DB2 ストアード・プロシージ

ャー・ビルダーを使用することです。実行方法の詳細については、ストアード・プロシージャー・ビルダーのオンライン・ヘルプを参照してください。

この節には、以下の項目が含まれています。
- デバッグの準備
- デバッグ表への書き込み
- デバッガーの呼び出し

デバッグの準備

1. JDK 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・コードがサーバーに保管されている場合には、 CLASSPATH 環境変数を Java ソース・コード・ディレクトリーに組み込むように設定するか、または 684 ページの『Java クラスを置く場所』に定義されているように、関数ディレクトリーにソース・コードを保管します。
   - db2set コマンドを使用して、インスタンスのデバッグを使用可能にします。

     ```
     db2set DB2ROUTINE_DEBUG=ON
     ```
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・コードを含むディレクトリーに DB2_DBG_PATH 環境変数を設定する。
4. デバッグ・テーブルを作成する。

   デバッガーを呼び出すのにストアード・プロシージャー・ビルダ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: DB2 エンタープライズ拡張エディション ・システムでは、デフォルト・ノードグループは USERSPACE1 表スペースでは IBMDEFAULTGROUP であり、システムに定義されたすべてのノードで構成されています。 DB2 エンタープライズ拡張エディション構成でストアード・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 つの調整プログラム・ノードがなければならず、そのノードだけを含むノードグループを定義しなければなりません。
5. クライアント上でデバッガー・デーモンを開始する。

   ストアード・プロシージャー・クライアントから、次のコマンドでデバッガー・デーモンを開始します。

   ```
   db2dbugd -qport=portno
   ```

ここで、*portno* は未使用の TCP/IP ポート番号です。値を指定しない場合には、デバッガーはデフォルト・ポート番号として 8000 を使用します。 Windows 32 ビット・オペレーティング・システムでは、 DB2 フォルダーにあるデバッガー・デーモンのショートカットをクリックして、デフォルト・ポート番号でデバッガーを開始することもできます。

デバッグ表への書き込み

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、INSERT、UPDATE、DELETE 特権のあるユーザーだけが、基礎表 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、ユーザー視点 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。したがって、この節の続きでは、あるユーザーがユーザー視点を介して表に書き込むことを前提とします。

ストアード・プロシージャー・ビルダーを使用してデバッグを呼び出す場合には、デバッガー・ユーティリティーを使用してデバッグ表に書き込み、それを管理できます。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実行します。

```
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')
```

ここで、各パラメーターは以下のとおりです。

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
ストアード・プロシージャーの固有名。これは、CREATE PROCEDURE コマンドで提供された固有名か、固有名が提供されていない場合には、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、ユーザーが *USER1* で、デバッグするクライアントの IP アドレスが 123.234.111.222 のストアード・プロシージャー *MySchema.myProc* のデバッグを使用可能にするには、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', '123.234.111.222')
```

ストアード・プロシージャーを除去する場合、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。デバッグ表と DB2 カタログを同期化する場合は、デバッグ情報を手動で削除する必要があります。

デバッグ表を手動で作成した場合であっても、ストアード・プロシージャー・ビルダーで作成した場合であっても、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 33. DB2DBG.ROUTINE_DEBUG 表定義

列名	データ・タイプ	属性	記述
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' (Stored 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** - ROUTINE_SCHEMA.SPECIFICNAME で指定されたストアード・プロシージャーのデバッグを使用可能にする。 • **N** - ROUTINE_SCHEMA.SPECIFICNAME で指定されたストアード・プロシージャーのデバッグを使用不可にする。これがデフォルトです。
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。
この表の基本キーは、AUTHID、TYPE、ROUTINE_SCHEMA、SPECIFICNAME です。			

デバッガーの呼び出し

上記の手順を正常に実行できた場合、ストアード・プロシージャーを呼び出すと、デバッグ表に指定した IP アドレスのクライアントでデバッガーを起動します。

デバッガーでは、ソース・コードのステップスルー、変数の表示、およびソース・コードへのブレークポイントの設定を行えます。デバッガーの使用の詳細については、オンライン・ヘルプにあるデバッガーの資料を参照してください。

Java ストアード・プロシージャーおよび UDF

Java ストアード・プロシージャーおよび UDF (これらを *Java ルーチン* という) は、DB2 カタログに登録する必要があります。 DB2 ユニバーサル・データベース バージョン 7 では、 Java ルーチンを登録および展開するために *SQLJ ルーチン* のコア仕様をサポートします。 CREATE PROCEDURE および CREATE FUNCTION ステートメントにある PARAMETER STYLE JAVA を使用して、 SQLJ ルーチンに準拠することを指定できます。

あるいは、DB2 では DB2 V5 および V5.2 PARAMETER STYLE DB2GENERAL ストアード・プロシージャーおよび UDF をサポートします。詳細については、789ページの『付録C. DB2DARI および DB2GENERAL ストアード・プロシージャーと UDF』を参照してください。

Java 関数またはストアード・プロシージャーを登録するには、以下の手順に従ってください。

1. Java ルーチンを Java メソッドで作成する。 Java ソース・コードを Java クラス・ファイルにコンパイルします。 Java ストアード・プロシージャーの作成の詳細については、 201ページの『第7章 ストアード・プロシージャー』を参照してください。 Java UDF の作成の詳細については、 436ページの『Java ユーザー定義の関数の作成および使用』を参照してください。
2. Java ルーチンを含むクラス・ファイルを *jar* ファイルに収集する。 1 つまたは複数のクラス・ファイルを 1 つの JAR ファイルに収集できます。 JAR ファイルを作成する手順については、 *アプリケーション構築の手引き* を参照してください。
3. DB2 インスタンスに JAR ファイルをインストールする。コマンド行から CALL SQLJ.INSTALL_JAR ステートメントを使用する方法については、690ページの『JAR ファイルのインストール、置換、および除去』を参照してください。また、アプリケーションで、あるいは CLP から `sqlj.install_jar` プロシージャーを呼び出すこともできます。
4. Java ルーチンに該当する CREATE PROCEDURE または CREATE FUNCTION SQL ステートメントを発行する。
   - CREATE PROCEDURE ステートメントの使用についての説明および例は、 208ページの『ストアード・プロシージャーの登録』を参照してください。
   - CREATE FUNCTION ステートメントの使用についての説明および例は、 *SQL 解説書* を参照してください。

JAR ファイルをインストールする際には、 DB2 は JAR ファイルから Java クラス・ファイルを抽出し、システム・カタログにそれぞれのクラスを登録します。 DB2 は、JAR ファイルを関数ディレクトリーの `jar/`*`schema`* サブディレクトリーにコピーしま

す。 DB2 は、JAR ファイルの新しいコピーに *jar-id* 文節で指定された名前を付けます。 DB2 インスタンスにすでにインストールされている JAR ファイルは直接変更しないでください。代わりに、 CALL SQLJ.REMOVE_JAR および CALL SQLJ.REPLACE_JAR コマンドを使用すると、インストールされた JAR ファイルを除去または置換できます。

JAR ファイルのインストール、置換、および除去

DB2 インスタンスに JAR ファイルをインストールしたり、置換したりするには、コマンド行プロセッサーで以下のコマンド構文を使用できます。

```
►►─┤ CALL ├──────────────────────────────────────────────────────────►◄
```

CALL

```
                                    (1)                   (2)
►►──┬─SQLJ.INSTALL_JAR─┬──(─'─jar-url──────'─,─'─jar-id──────'─)──────────►◄
    └─SQLJ.REPLACE_JAR─┘
```

注:

1 インストールまたは置換される JAR ファイルを含む URL を指定します。 サポートされる URL 体系は、'file:' だけです。

2 データベースの JAR ID を jar-url で指定したファイルに関連付けるように指定します。

注: OS/2 および Windows 32 ビット・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登録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に固有なものにするには、そのインスタンスの *DB2INSTPROF* に固有な値を指定する必要があります。

たとえば、 `file:/home/db2inst/classes/` ディレクトリーにある `Proc.jar` ファイルを DB2 インスタンスにインストールするには、コマンド行プロセッサーから次のコマンドを出します。

```
CALL SQLJ.INSTALL_JAR('file:/home/db2inst/classes/Proc.jar' , 'myproc_jar')
```

SQL コマンドが今後 `Procedure.jar` ファイルを使用する際には、 `myproc_jar` という名前で参照します。 JAR ファイルをデータベースから削除するには、以下の構文の CALL REMOVE_JAR コマンドを使用します。

```
►►─┤ CALL ├──────────────────────────────────────────────────────────►◄
```

CALL

```
                                    (1)
►►─SQLJ.REMOVE_JAR─(─'─jar-id─────────'─)─────────────────────────────►◄
```

注:

1 データベースから削除される JAR ファイルの JAR ID を指定します。

データベースから JAR ファイル myProc_jar を除去するには、コマンド行プロセッサーに次のコマンドを入力します。

```
CALL SQLJ.REMOVE_JAR('myProc_jar')
```

Java での関数定義

Java ルーチンを作成するには、public クラスにある、対応する public static メソッドをコーディングする必要があります。また、throws SQLException 文節を使用して Java ルーチンを宣言しなければなりません。メソッドのシグニチャーおよび残りのメソッド宣言を、メソッド本体からの出力に対応するようにコーディングしてください。

Java の値を戻さない関数: 呼び出し側のプログラムに値を戻さないメソッドを作成するには、メソッドが void を戻すように宣言し、メソッド本体に渡す必要のあるパラメーターをシグニチャーに含めます。単純な UDPATE を実行し、クライアント・アプリケーションに値を戻さないストアード・プロシージャーを、以下のように作成できます。

```
public class JavaExamples {
   public static void updateJob(String oldJob, String newJob)
         throws SQLException {
      Connection conn=DriverManager.getConnection("jdbc:ibm.db2.sample");
      PreparedStatement stmt = conn.prepareStatement("UPDATE employee
         SET job = ? WHERE job = ?");
      stmt.setString(1, newJob);
      stmt.setString(2, oldJob);
      stmt.executeUpdate();
      conn.close();
      return;
      }
   }
```

Java の単一値を戻す関数: それぞれの SQL データ・タイプに対応する Java 戻りタイプで、単一値を戻す Java メソッドを宣言します (658ページの『Java でサポートされている SQL データ・タイプ』を参照)。 SQL INTEGER 値を戻すスカラー UDF を以下のように作成します。

```
public class JavaExamples {
   public static int getDivision(String division) throws SQLException {
      if (division.equals("Corporate")) return 1;
      else if (division.equals("Eastern")) return 2;
      else if (division.equals("Midwest")) return 3;
      else if (division.equals("Western")) return 4;
      else return 5;
   }
}
```

Java の複数の値を戻す関数: ストアード・プロシージャーとしてカタログされた Java メソッドは、 1 つまたは複数の値を戻します。また、複数の結果セットを戻す Java ストアード・プロシージャーを作成することもできます。
246ページの『ストアード・プロシージャーからの結果セットの戻り』を参照してください。あらかじめ決められた数の値を戻すメソッドをコーディングするには、戻りタイプ void を宣言し、メソッドのシグニチャーに出力のタイプを配列として含めます。指定されたしきい値より低い収入を得ている従業員のうち、年齢の高い順から 2 人の名前、勤続年数、および収入を戻すストアード・プロシージャーを作成するには、以下のようにします。

```
public Class JavaExamples {
   public static void lowSenioritySalary
      (String[] name1, int[] years1, BigDecimal[] salary1,
       String[] name2, int[] years2, BigDecimal[] salary2,
       Integer threshhold) throws SQLException {
      #sql iterator ByNames (String name, int years, BigDecimal salary);
      ByNames result;
      #sql result = {"SELECT name, years, salary
                      FROM staff
                      WHERE salary < :threshhold
                      ORDER BY years DESC"};
      if (result.next()) {
         name1[0] = result.name();
         years1[0] = result.years();
         salary1[0] = result.salary();
      }
      else {
         name1[0] = "****";
         return;
      }
      if (result.next()) {
         name2[0] = result.name();
         years2[0] = result.years();
         salary2[0] = result.salary();
      }
      else {
         name2[0] = "****";
         return;
      }
   }
}
```

JDBC 1.2 で LOB およびグラフィカル・オブジェクトを使用する

JDK 1.2 の JDBC 2.0 仕様は、 LOB およびグラフィック・タイプのサポートを定義します。 DB2 JDBC 2.0 ドライバー・サポートの詳細については、 668ページの『JDBC 2.0』を参照してください

| **注:** ストアード・プロシージャーや UDF の LOB および漢字タイプ用の DB2 JDBC 2.0 ドライバー・サポートは使用できません。ストアード・プロシージャーや UDF の LOB または漢字タイプを使用するには、 JDBC 1.2 LOB サポートを使用する必要があります。 DB2 JDBC 2.0 ドライバーで DB2 JDBC 1.2 LOB サポートを使用する方法について詳しくは、 670ページの『JDBC 2.0 互換性』を参照してください。

| しかし、JDBC 1.2 仕様では、ラージ・オブジェクト (LOB) またはグラフィック・タイプについて明示的に説明していません。 JDBC 1.2 ドライバーを使用する場合、 DB2 は以下のサポートを LOB およびグラフィック・タイプに提供します。

ご使用のアプリケーションで LOB またはグラフィック・タイプを使用する場合には、LOB を対応する LONGVAR タイプとみなします。 LOB タイプは SQL 中に最大長が宣言されているので、宣言された限界より長い配列またはストリングを戻さないことを確認してください。この考慮事項は、SQL ストリング・タイプにも同様に当てはまります。

GRAPHIC および DBCLOB データ・タイプは、対応する CHAR タイプとして扱います。

| DB2 クライアントは、サーバー・コード・ページから Unicode に直接データを変換します。以下の JDBC API は、Unicode からまたは Unicode へとデータを変換します。

getString
: サーバー・コード・ページから Unicode に変換します。

setString
: Unicode からサーバー・コード・ページに変換します。

getUnicodeStream
: サーバー・コード・ページから Unicode に変換します。

setUnicodeStream
: Unicode からサーバー・コード・ページに変換します。

以下の JDBC API には、クライアント・コード・ページとサーバー・コード・ページ間の変換が関係します。

setAsciiStream
: クライアント・コード・ページからサーバー・コード・ページに変換します。

getAsciiStream
サーバー・コード・ページからクライアント・コード・ページに変換します。

JDBC および SQLJ の相互運用性

SQLJ 言語は、プログラムが作成される時点で認識される静的 SQL 操作を直接サポートします。実行時までに特定の SQL ステートメントのすべてまたは一部を判別できない場合には、これは動的操作です。 SQLJ から動的 SQL 操作を実行するには、JDBC を使用します。 ConnectionContext オブジェクトには JDBC Connection オブジェクトが含まれます。これを使用して、動的 SQL ステートメントに必要な JDBC Statement オブジェクトを作成できます。

すべての SQLJ ConnectionContext クラスには、引き数として JDBC Connection を取るコンストラクターが含まれます。このコンストラクターを使用して SQLJ ConnectionContext インスタンスを作成し、その基本データベース接続を JDBC Connection のインスタンスと共用します。

SQLJ ConnectionContext インスタンスにはすべて、 JDBC Connection インスタンスを戻す `getConnection()` メソッドがあります。戻される JDBC Connection は、基本データベース接続を SQLJ ConnectionContext と共用します。これを使用すると、 JDBC API で説明されているような動的 SQL 操作を実行できます。

セッション共用

前述の相互運用性メソッドでは、 SQLJ で使用される接続の抽象化と JDBC で使用される接続の抽象化との間の変換を提供します。どちらの抽象化も同じデータベース・セッション、つまり、基本データベース接続を使用します。したがって、メソッドの呼び出しが一方のオブジェクトのセッション状態に影響を及ぼす場合、他方のオブジェクトにも影響を与えることになります。なぜなら、実際に影響を受けているのは共用されている基本セッションだからです。

JDBC は、新しく作成された接続のセッション状態のデフォルトを定義します。たいていの場合、SQLJ はこのデフォルトを使用します。しかし、新しく作成された JDBC Connection にデフォルトで自動コミット・モードがある場合、 SQLJ ConnectionContext の作成時に自動コミット・モードを明示的に指定する必要があります。

Java での接続リソース管理

ConnectionContext インスタンスの `close()` メソッドを呼び出すと、関連する JDBC Connection インスタンスと基本データベース接続がクローズされることになります。 ConnectionContext は、他の ConnectionContext または JDBC Connection (あるいはその両方) と基本データベース接続を共用していることがあるため、 ConnectionContext をクローズするときに基本データベース接続もクローズされてしまうのは望ましくありません。プログラマーは、基本データベース接続を実際にクローズしなくても、 ConnectionContext (たとえば、ステートメント・ハンドル) によって保守されるリソース

を解放したいと思うかもしれません。このために、ConnectionContext クラスは、基本データベース接続をクローズするかどうかを示すブール引き数を取る close() メソッドもサポートします。データベース接続をクローズする場合には定数 CLOSE_CONNECTION を、接続を保持するには KEEP_CONNECTION を使用します。 close() には引き数を取らない変形したメソッドもあり、これは close(CLOSE_CONNECTION) をす早く呼び出します。

ConnectionContext インスタンスが不要情報として収集される前に明示的にクローズしていない場合には、 ConnectionContext の終了処理メソッドによって close(KEEP_CONNECTION) が呼び出されます。これにより、接続を使用しているかもしれない他の JDBC および SQLJ オブジェクトの基本データベース接続を保守しつつ、通常の不要情報収集プロセスでは接続関連リソースを再利用できるようになります。他の JDBC や SQLJ オブジェクトが接続を使用していない場合、不要情報収集プロセスによってこのデータベース接続はクローズされ、再利用されることに注意してください。

SQLJ ConnectionContext オブジェクトと JDBC Connection オブジェクトはどちらも、 close() メソッドに応答します。 SQLJ プログラムを作成する際には、 ConnectionContext オブジェクトだけで close() メソッドを呼び出せば十分です。これは、ConnectionContext をクローズすることで、それに関連する JDBC Connection もクローズするためです。しかし、 ConnectionContext の getConnection() メソッドによって戻される JDBC Connection をクローズするだけでは不十分です。これは、JDBC Connection の close() メソッドが、そこに含まれている ConnectionContext をクローズしないためです。したがって、ConnectionContext によって保守されるリソースは、不要情報の収集が実行されるまで解放されません。

close() の変形したメソッドが ConnectionContext インスタンスで呼び出されている場合、 ConnectionContext の isClosed() メソッドは true を戻します。 isClosed() が true を戻す場合、 close() を呼び出してもクローズされません。また、その他のメソッドの呼び出しは未定義です。

第22章 Perl でのプログラミング

Perl でのプログラミングに関する考慮事項 697
Perl の制約事項 697
Perl を使ったデータベースへの接続. . . . 698
Perl での取り出し結果 698
Perl のパラメーター・マーカー 699
Perl の SQLSTATE および SQLCODE 変数 699
Perl DB2 アプリケーションの例 700

Perl でのプログラミングに関する考慮事項

Perl は、数多くのオペレーティング・システムで自由に使用できるポピュラーなプログラム言語です。 Perl データベース・インターフェース (DBI) モジュール (http://www.perl.com から利用可能) で DBD::DB2 ドライバー (http://www.ibm.com/software/data/db2/perl から利用可能) を使用すると、 Perl を使った DB2 を作成することができます。

Perl はインタープリター言語であり、 Perl DBI モジュールは動的 SQL を使用するため、 DB2 アプリケーションのプロトタイプをす早く作成および修正する上で、 Perl は理想的な言語です。 Perl DBI モジュールは、 CLI および JDBC と大変よく似たインターフェースを使用するため、 Perl プロトタイプを簡単に CLI および JDBC に移植することができます。

大部分のデータベース・ベンダーは、 Perl DBI モジュールのデータベース・ドライバーを提供しています。これは、さまざまなデータベース・サーバーからデータにアクセスするアプリケーションを作成するために、 Perl を使用できることも意味します。たとえば、DBD::Oracle データベース・ドライバーを使用して Oracle データベースに接続する Perl DB2 アプリケーションを作成し、 Oracle データベースからデータを取り出し、 DBD::DB2 データベース・ドライバーを使用して DB2 データベースにデータを挿入することができます。

Perl の制約事項

Perl DBI モジュールがサポートするのは、動的 SQL だけです。複数回ステートメントを実行する必要がある場合には、ステートメントを準備する `prepare` 呼び出しを発行して、 Perl DB2 アプリケーションのパフォーマンスを改善することができます。

ワークステーションにインストールする DBD::DB2 ドライバー・バージョンの制限に関する最新情報については、 DBD::DB2 パッケージにある CAVEATS ファイルを参照してください。

© Copyright IBM Corp. 1993, 2001

Perl を使ったデータベースへの接続

Perl が DBI モジュールをロードできるようにするには、 DB2 アプリケーションに次の行を含める必要があります。

```
use DBI;
```

DBI モジュールは、`DBI->connect` ステートメントを使用してデータベース・ハンドルを作成すると、 DBD::DB2 ドライバーを自動的にロードします。以下の構文を使用します。

```
my $dbhandle = DBI->connect('dbi:DB2:dbalias', $userID, $password);
```

ここで、各パラメーターは以下のとおりです。

$dbhandle
: connect ステートメントが戻すデータベース・ハンドル。

dbalias
: DB2 データベース・ディレクトリーにカタログされている DB2 別名

$userID
: データベースへの接続で使用するユーザー ID

$password
: データベースへの接続で使用するユーザー ID のパスワード

Perl での取り出し結果

Perl DBI モジュールは動的 SQL しかサポートしていないため、 Perl DB2 アプリケーションではホスト変数は使用しないでください。 SQL 照会から結果を戻すには、以下のステップを実行します。

ステップ 1. 『Perl を使ったデータベースへの接続』で説明されているようにして、データベース・ハンドルを作成します。

ステップ 2. 作成したデータベース・ハンドルからステートメント・ハンドルを作成します。たとえば、ストリング引き数として SQL ステートメントと一緒に `prepare` を呼び出し、データベース・ハンドルからステートメント・ハンドル *$sth* を戻すことができます。以下に示す Perl ステートメントは、その例です。

```
my $sth = $dbhandle->prepare(
   'SELECT firstnme, lastname
      FROM employee '
   );
```

ステップ 3. ステートメント・ハンドルで `execute` を呼び出して、 SQL ステートメントを実行します。 `execute` が正常に呼び出されると、 結果セットがステ

ートメント・ハンドルに関連付けられます。 たとえば、以下の Perl ステートメントを使用すると、前の例で準備したステートメントを実行できます。

```
#Note: $rc represents the return code for the execute call
my $rc = $sth->execute();
```

ステップ 4. `fetchrow()` への呼び出しを使用して、ステートメント・ハンドルに関連付けられた結果セットから行を取り出します。 Perl DBI は、列ごとに値を 1 つ指定した配列として行を戻します。たとえば、以下の Perl ステートメントを使用すると、前の例にあるステートメント・ハンドルからすべての行を戻すことができます。

```
while (($firstnme, $lastname) = $sth->fetchrow()) {
   print "$firstnme $lastname¥n";
}
```

Perl のパラメーター・マーカー

指定したフィールドごとに別々の入力値を使用して、準備したステートメントを実行できるようにするため、 Perl DBI モジュールはパラメーター・マーカーを使ってステートメントを準備し、実行します。 SQL ステートメントにパラメーター・マーカーを入れるには、疑問符 (?) 文字を使用します。

以下の Perl コードは、 SELECT ステートメントの WHERE 節のパラメーター・マーカーを受け入れるステートメント・ハンドルを作成します。その後、このコードは、入力値 25000 および 35000 を使用して 2 度ステートメントを実行し、パラメーター・マーカーを置換します。

```
my $sth = $dbhandle->prepare(
   'SELECT firstnme, lastname
      FROM employee
      WHERE salary > ?'
   );

my $rc = $sth->execute(25000);

⋮

my $rc = $sth->execute(35000);
```

Perl の SQLSTATE および SQLCODE 変数

Perl DBI のデータベース・ハンドルまたはステートメント・ハンドルに関連付けられた SQLSTATE を戻すには、 `state` メソッドを呼び出します。たとえば、データベース・ハンドル $dbhandle に関連付けられた SQLSTATE を戻すには、次の Perl ステートメントをアプリケーションに組み込みます。

```
my $sqlstate = $dbhandle->state;
```

Perl DBI のデータベース・ハンドルまたはステートメント・ハンドルに関連付けられた SQLCODE を戻すには、 err メソッドを呼び出します。 Perl DBI のデータベース・ハンドルまたはステートメント・ハンドルに関連付けられた SQLCODE のメッセージを戻すには、 errstr メソッドを呼び出します。たとえば、データベース・ハンドル $dbhandle に関連付けられた SQLCODE を戻すには、次の Perl ステートメントをアプリケーションに組み込みます。

```
my $sqlcode = $dbhandle->err;
```

Perl DB2 アプリケーションの例

```
#!/usr/bin/perl
use DBI;

my $database='dbi:DB2:sample';
my $user='';
my $password='';

my $dbh = DBI->connect($database, $user, $password)
   or die "Can't connect to $database: $DBI::errstr";

my $sth = $dbh->prepare(
   q{ SELECT firstnme, lastname
      FROM employee }
   )
   or die "Can't prepare statement: $DBI::errstr";

my $rc = $sth->execute
   or die "Can't execute statement: $DBI::errstr";

print "Query will return $sth->{NUM_OF_FIELDS} fields.¥n¥n";
print "$sth->{NAME}->[0]: $sth->{NAME}->[1]¥n";

while (($firstnme, $lastname) = $sth->fetchrow()) {
   print "$firstnme: $lastname¥n";
}

# check for problems which may have terminated the fetch early
warn $DBI::errstr if $DBI::err;

$sth->finish;
$dbh->disconnect;
```

第23章 COBOL でのプログラミング

COBOL でのプログラミングに関する考慮事項 701
COBOL での言語制限 701
COBOL の入力および出力ファイル 701
COBOL のインクルード・ファイル 702
COBOL での SQL ステートメントの組み込み 705
COBOL のホスト変数 707
  COBOL でのホスト変数の命名 707
  ホスト変数の宣言 708
  COBOL の標識変数 711
  COBOL における LOB 宣言 711
  COBOL における LOB ロケーター宣言 713
  COBOL におけるファイル参照宣言 . . . 713
  COBOL でのホスト構造サポート . . . 714
  COBOL のインディケーター表 716
  COBOL グループ・データ項目での REDEFINES の使用 717
  BINARY/COMP-4 COBOL データ・タイプの使用 717
COBOL でサポートされる SQL データ・タイプ 718
  COBOL での FOR BIT DATA 721
COBOL での SQLSTATE および SQLCODE 変数 721
COBOL での日本語または中国語 (繁体字) EUC、および UCS-2 に関する考慮事項 . . 721
オブジェクト指向 COBOL 722

COBOL でのプログラミングに関する考慮事項

特定のホスト言語によるプログラミングの考慮事項について、以降のページで説明します。言語制限、ホスト言語別のインクルード・ファイル、組み込み SQL ステートメント、ホスト変数、およびサポートされるホスト変数のデータ・タイプについての情報が含まれています。組み込み SQL ステートメント、言語制限、およびホスト変数にサポートされるデータ・タイプについては、 Micro Focus COBOL の資料を参照してください。

COBOL での言語制限

API ポインターの長さは、すべて 4 バイトです。 API 呼び出しで値パラメーターとして使用する整数の変数はすべて、 USAGE COMP-5 文節で宣言しなければなりません。

OS/2 で Micro Focus 16 ビット COBOL を使用するときは、それぞれのデータベース・マネージャー API 呼び出しの前に下線記号を 2 つ (__) 入れてください。そうすれば、コンパイラーは十分なセグメント・アドレッシングを使用して、 *by reference* パラメーターを渡します。

COBOL の入力および出力ファイル

デフォルトには、入力ファイルの拡張子は `.sqb` ですが、 TARGET プリコンパイル・オプション (TARGET ANSI_COBOL、TARGET IBMCOB、TARGET MFCOB、または TARGET MFCOB16) を使用した場合は、入力ファイルに任意の拡張子を付けることができます。

© Copyright IBM Corp. 1993, 2001

デフォルトには、出力ファイルの拡張子は .cbl ですが、 OUTPUT プリコンパイル・オプションを使用すれば、出力修正後のソース・ファイルに新しい名前やパスを指定できます。

COBOL のインクルード・ファイル

COBOL のホスト言語固有のインクルード・ファイルには、ファイル拡張子 .cbl が付いています。 IBM COBOL コンパイラーの「システム/390 ホスト・データ・タイプ・サポート」機能を使用する場合、ご使用のアプリケーションの DB2 インクルード・ファイルは以下のディレクトリーにあります。

```
$HOME/sqllib/include/cobol_i
```

DB2 サンプル・プログラムを、提供されたスクリプト・ファイルを使用して構築する場合、スクリプト・ファイルで指定されているインクルード・ファイルのパスを、cobol_i ディレクトリー (cobol_a ディレクトリーではない) に変更する必要があります。

IBM COBOL コンパイラーの ″システム/390 ホスト・データ・タイプ・サポート″ 機能を使用**しない**場合、またはこのコンパイラーの以前のバージョンを使用する場合、ご使用のアプリケーションの DB2 インクルード・ファイルは以下のディレクトリーにあります。

```
$HOME/sqllib/include/cobol_a
```

アプリケーションで使用するためのこれらのインクルード・ファイルについて以下で説明します。

SQL (sql.cbl)
このファイルには、バインダー、プリコンパイラー、およびエラー・メッセージ検索 API 用の言語固有プロトタイプが含まれています。また、システム定数も定義されています。

SQLAPREP (sqlaprep.cbl)
このファイルには、独自のプリコンパイラーの作成に必要な定義が入っています。

SQLCA (sqlca.cbl)
このファイルは SQL 連絡域 (SQLCA) 構造を定義します。 SQLCA には、データベース・マネージャーが SQL ステートメントおよび API 呼び出しの実行に関するエラー情報をアプリケーションに提供するために使用する変数が含まれています。

SQLCA_92 (sqlca_92.cbl)
このファイルには、SQL 連絡域 (SQLCA) 構造の FIPS SQL92 Entry Level 準拠版が入っています。このファイルは、FIPS SQL92 Entry Level standard に準拠する DB2 アプリケーションを作成するときには、 sqlca.cbl の代わりに組み込む必要があります。 sqlca_92.cbl

ファイルは、 LANGLEVEL プリコンパイラー・オプションを SQL92E に設定すると、 DB2 プリコンパイラーが自動的に組み込みます。

SQLCODES (sqlcodes.cbl)
このファイルは、SQLCA 構造の SQLCODE フィールドで使用する定数を定義します。

SQLDA (sqlda.cbl)
このファイルは SQL 記述子域 (SQLDA) 構造を定義します。 SQLDA はアプリケーションとデータベース・マネージャーとの間でデータをやりとりするために使用されます。

SQLEAU (sqleau.cbl)
このファイルには、DB2 セキュリティー監査 API に必要な定数および構造の定義が含まれています。これらの API を使用する場合は、プログラムにこのファイルを組み込む必要があります。このファイルにはまた、監査証跡レコード内のフィールドの定数およびキーワード値の定義も含まれています。これらの定義は、外部または取引先の監査証跡抽出プログラムで使用できます。

SQLENV (sqlenv.cbl)
このファイルは、データベース環境 API に対する言語固有の呼び出し、およびそれらのインターフェースの構造、定数、戻りコードを定義します。

SQLETSD (sqletsd.cbl)
このファイルは、表スペース記述子構造 SQLETSDESC を定義します。これはデータベース作成 API である sqlgcrea に渡されます。

SQLE819A (sqle819a.cbl)
データベースのコード・ページが 819 (ISO ラテン 1) の場合、このシーケンスは、ホスト CCSID 500 (EBCDIC 各国対応) バイナリー照合に従って、 FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQLE819B (sqle819b.cbl)
データベースのコード・ページが 819 (ISO ラテン 1) の場合、このシーケンスは、ホスト CCSID 037 (EBCDIC 米国英語) バイナリー照合に従って、 FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQLE850A (sqle850a.cbl)
データベースのコード・ページが 850 (ASCII ラテン 1) の場合、このシーケンスは、ホスト CCSID 500 (EBCDIC 各国対応) バイナリー照合に従って、 FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQLE850B (sqle850b.cbl)
データベースのコード・ページが 850 (ASCII ラテン 1) の場合、このシーケンスは、ホスト CCSID 037 (EBCDIC 米国英語) バイナリー照合に従って、 FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQLE932A (sqle932a.cbl)
データベースのコード・ページが 932 (ASCII 日本語) の場合、このシーケンスは、ホスト CCSID 5035 (EBCDIC 日本語) バイナリー照合に従って、 FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQLE932B (sqle932b.cbl)
データベースのコード・ページが 932 (ASCII 日本語) の場合、このシーケンスは、ホスト CCSID 5026 (EBCDIC 日本語) バイナリー照合に従って、 FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQL1252A (sql1252a.cbl)
データベースのコード・ページが 1252 (Windows ラテン 1) の場合、このシーケンスは、ホスト CCSID 500 (EBCDIC 各国対応) バイナリー照合に従って、FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQL1252B (sql1252b.cbl)
データベースのコード・ページが 1252 (Windows ラテン 1) の場合、このシーケンスは、ホスト CCSID 037 (EBCDIC 米国英語) バイナリー照合に従って、FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQLMON (sqlmon.cbl)
このファイルは、データベース・システム・モニター API に対する言語固有の呼び出し、およびそれらのインターフェースの構造、定数、戻りコードを定義します。

SQLMONCT (sqlmonct.cbl)
このファイルには、データベース・システム・モニター API を呼び出すのに必要な定数定義とローカル・データ構造定義が含まれています。

SQLSTATE (sqlstate.cbl)
このファイルは、SQLCA 構造の SQLSTATE フィールドで使用する定数を定義します。

SQLUTBCQ (sqlutbcq.cbl)
このファイルは、表スペース・コンテナー照会データ構造

SQLB-TBSCONTQRY-DATA を定義します。これは、表スペース・コンテナー照会 API である sqlgstsc、sqlgftcq、および sqlgtcq で使用されます。

SQLUTBSQ (sqlutbsq.cbl)
このファイルは、表スペース照会データ構造 SQLB-TBSQRY-DATA を定義します。これは、表スペース照会 API である sqlgstsq、sqlgftsq、および sqlgtsq で使用されます。

SQLUTIL (sqlutil.cbl)
このファイルは、ユーティリティー API に対する言語固有の呼び出し、およびそれらのインターフェースに必要な構造、定数、コードを定義します。

COBOL での SQL ステートメントの組み込み

組み込み SQL ステートメントは次の 3 つのエレメントからなります。

エレメント	正しい COBOL 構文
一組のキーワード	EXEC SQL
ステートメント・ストリング	任意の有効な SQL ステートメント
ステートメント終了文字	END-EXEC.

以下に例を示します。

```
EXEC SQL SELECT col INTO :hostvar FROM table END-EXEC.
```

組み込み SQL ステートメントには、以下の規則が適用されます。

- 実行可能な SQL ステートメントは、PROCEDURE DIVISION になければなりません。 SQL ステートメントの前には、COBOL ステートメントとして段落名を付けることができます。
- SQL ステートメントは、領域 A (列 8 〜 11) または領域 B (列 12 〜 72) のどちらからも開始することができます。
- 各 SQL ステートメントは、EXEC SQL で開始して END-EXEC で終了します。各 SQL ステートメントは、修正済みソース・ファイル内の注釈として、 SQL プリコンパイラーに含まれます。
- SQL ステートメント終了文字を使用しなければならない。使用しない場合、プリコンパイラーはアプリケーション内の次の終了文字まで処理を継続します。これにより、不確定のエラーが起こる恐れがあります。
- SQL 注釈は、組み込み SQL ステートメントの一部となっている行であればどこでも使用することができる。この注釈は、動的に実行するステートメントでは使用できません。 SQL 注釈の形式は、ダブル・ダッシュ (--) の後に 0 個以上の文字ストリングが続き、行末で終了します。 SQL 注釈を SQL ステートメントの後に置かないよ

うにしてください。これを行うと、注釈が COBOL 言語の一部のように見えるため、コンパイル・エラーの原因となるからです。

- COBOL の注釈は、組み込み SQL ステートメント内の*ほとんど* の場所で使用できます。例外は以下のとおりです。
  - 注釈は EXEC と SQL との間では使用できない。
  - 注釈は動的に実行されるステートメントでは使用できない。
- SQL ステートメントは、COBOL 言語と同じ行継続規則に従います。ただし、EXEC SQL キーワードを 2 行に分割しないでください。
- SQL ステートメントを含んだファイルの組み込みに COBOL COPY ステートメントは使用できません。 SQL ステートメントはモジュールがコンパイルされる前にプリコンパイルされます。プリコンパイラーは、COBOL COPY ステートメントを無視します。代わりに SQL INCLUDE ステートメントを使用して、これらのファイルを組み込んでください。

  INCLUDE ファイルを検索する際、DB2 COBOL プリコンパイラーはまず最初に現行ディレクトリーを検索し、次いで DB2INCLUDE 環境変数に指定されたディレクトリーを検索します。次の例を考えてみてください。

  - `EXEC SQL INCLUDE payroll END-EXEC.`

    INCLUDE ステートメントに指定されたファイルが単引用符 (') で囲まれていない場合、上記のとおり、プリコンパイラーは最初 `payroll.sqb` を検索し、次に `payroll.cpy`、次に `payroll.cbl`、最後にそれがロックする各ディレクトリーを検索します。
  - `EXEC SQL INCLUDE 'pay/payroll.cbl' END-EXEC.`

    ファイル名が単引用符 (') で囲まれている場合、すでに説明したとおり、名前に拡張子は追加されません。

    単引用符 (') 内のファイル名に完全パスが含まれていない場合、 DB2INCLUDE の中身によってそのファイルの検索が行われ、何らかのパスが INCLUDE ファイル名に指定されます。たとえば、DB2 (AIX 版) では、 DB2INCLUDE は '`/disk2:myfiles/cobol`' に設定され、プリコンパイラーは '`./pay/payroll.cbl`'、'`/disk2/pay/payroll.cbl`'、 '`./myfiles/cobol/pay/payroll.cbl`' の順に探索します。プリコンパイラー・メッセージには、実際にファイルが見つかったパスが表示されます。 OS/2 および Windows プラットフォームでは、上の例のスラッシュを円記号 (¥) に置き換えます。

  注: DB2INCLUDE の設定は、DB2 コマンド行プロセッサーによってキャッシュされます。何らかの CLP コマンドを実行した後に DB2INCLUDE の設定を変更する場合は、 TERMINATE コマンドを入力してから、データベースに接続し直し、あとは通常どおりプリコンパイルしてください。
- ストリング定数を次の行に継続するには、継続行の列 7 に '-' を、また列 12 またはそれ以降にストリング区切り文字を入れなければなりません。
- SQL 算術演算子はブランクで区切らなければなりません。

- 全ラインの COBOL 注釈は、SQL ステートメント内を含め、プログラムのどこで使用してもかまいません。
- SQL ステートメント内のホスト変数の参照時に宣言されたとおりのホスト変数を使用する。
- 行末文字およびタブ文字などの空白文字の置換は、次のように行われる。
  - 引用符の外 (ただし、SQL ステートメント内) では、行末文字または TAB 文字は単一スペースと置き換えられる。
  - 引用符内では、COBOL プログラムに適合した形でストリングが継続されていれば、行末文字は消去されます。 TAB は修正されません。

  行末および TAB に使用される実際の文字は、プラットフォームごとに異なります。たとえば、OS/2 は復帰 / 改行を行末に使用するのに対し、UNIX ベースのシステムは改行のみを使用します。

COBOL のホスト変数

ホスト変数は、SQL ステートメント内で参照される COBOL の言語変数です。これにより、アプリケーションは入力データをデータベース・マネージャーに渡し、またデータベース・マネージャーから出力データを受け取ることができます。アプリケーションがプリコンパイルされると、コンパイラーはホスト変数を他の COBOL 変数と同様に使用します。 ホスト変数を命名、宣言、および使用する際には、以下に示す規則に従ってください。

COBOL でのホスト変数の命名

SQL プリコンパイラーは、宣言された名前によってホスト変数を識別します。以下の規則が適用されます。

- 変数名は 255 文字以内の長さで指定する。
- ホスト変数は、SQL、sql、DB2、または db2 以外の接頭部で開始する。これらはシステムが使用する予約語です。
- これから説明する宣言構文を使用する FILLER 項目はグループ・ホスト変数宣言では許可されており、プリコンパイラーはその項目を無視します。ただし、SQL DECLARE セクション内で FILLER を複数回使用した場合、プリコンパイラーは失敗します。 VARCHAR、LONG VARCHAR、 VARGRAPHIC、または LONG VARGRAPHIC 宣言には、FILLER 項目を組み込むことができません。
- ハイフンは、ホスト変数名として使用できます。

  SQL は、スペースで囲まれたハイフンを減算演算子として解釈します。ハイフンをホスト変数名として使用する場合は、スペースを入れないでください。
- REDEFINES 文節は、ホスト変数宣言では許可されています。
- レベル-88 宣言は、ホスト変数宣言セクションでは許可されていますが、無視されます。

ホスト変数の宣言

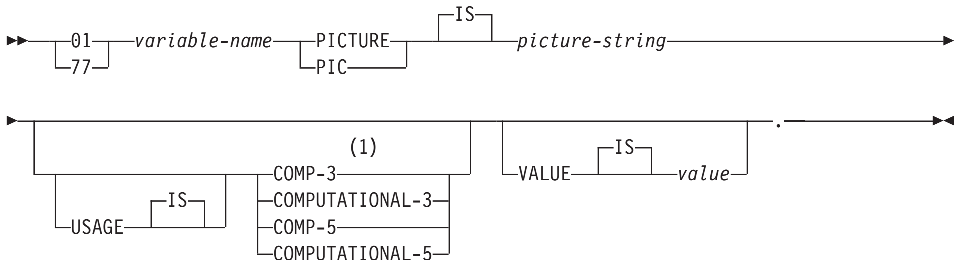
ホスト変数宣言の識別には、SQL の宣言セクションを使用しなければなりません。これは、それ以降の SQL ステートメントで参照が可能なホスト変数をプリコンパイラーに知らせます。

構造型のホスト変数の宣言方法についての詳細は、 363ページの『構造型ホスト変数の宣言』を参照してください。

COBOL プリコンパイラーは、有効な COBOL 宣言のサブセットのみを認識します。

COBOL における数値ホスト変数の構文は、数値ホスト変数の構文を示します。

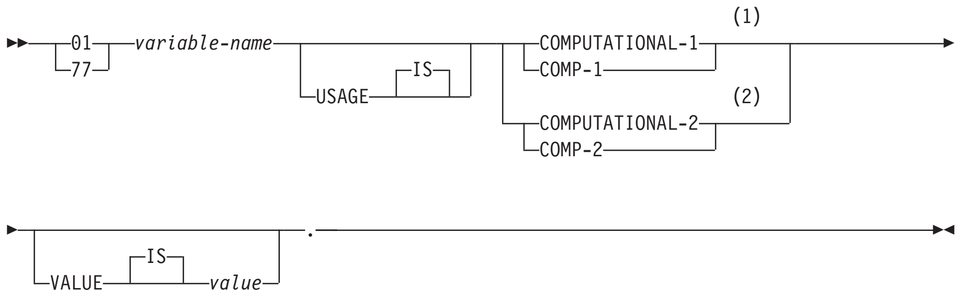
COBOL における数値ホスト変数の構文



注:

1 COMP-3 の代わりに PACKED-DECIMAL を使用できます。

浮動小数点



注:

1 REAL (SQLTYPE 480)、長さ 4

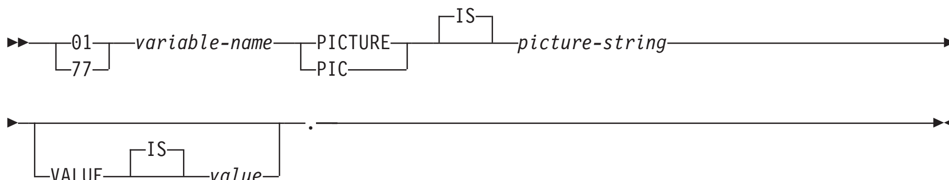
2 DOUBLE (SQLTYPE 480)、長さ 8

数値ホスト変数に関する考慮事項:

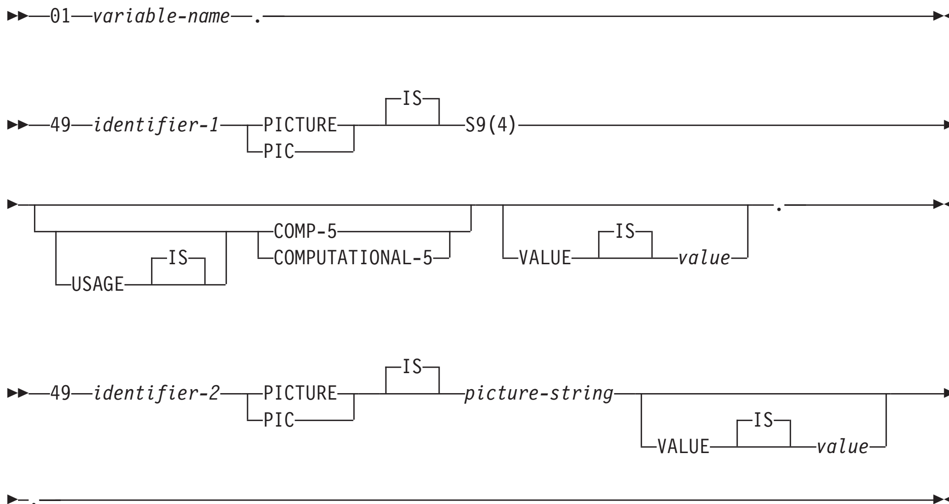
1. *Picture-string* は、以下のいずれかの形式でなければならない。
   - S9(m)V9(n)
   - S9(m)V
   - S9(m)
2. 数字の 9 は拡張することができる (たとえば、″S9(3)″ の代わりに ″S999″ とすることができます)。
3. *m* および *n* は、正の整数でなければならない。

COBOL における文字ホスト変数の構文: 固定長は、文字ホスト変数の構文を示します。

COBOL における文字ホスト変数の構文: 固定長



可変長



文字ホスト変数に関する考慮事項:

1. *Picture-string* の形式は、*X(m)* でなければならない。 X は拡張できます (たとえば、″X(3)″ の代わりに ″XXX″ にできます)。
2. *m* は、1 〜 254 までの長さの固定長ストリングである。

3. *m* は、1 ～ 32 700 までの長さの可変長ストリングである。
4. *m* が 32 672 よりも大きい場合、そのホスト変数は LONG VARCHAR ストリングと見なされ、使用が制限される。
5. X および 9 を、PICTURE 文節内のピクチャー文字として使用する。他の文字は使用できません。
6. 可変長ストリングは、長さ項目と値項目とから構成される。適切な COBOL 名を、長さ項目およびストリング項目として使用できます。ただし、SQL ステートメント内の集合名を使用して可変長ストリングを参照してください。
7. 以下の例に示すような CONNECT ステートメントでは、 COBOL 文字ストリング・ホスト変数 dbname および userid の後続ブランクは、処理前に削除されます。

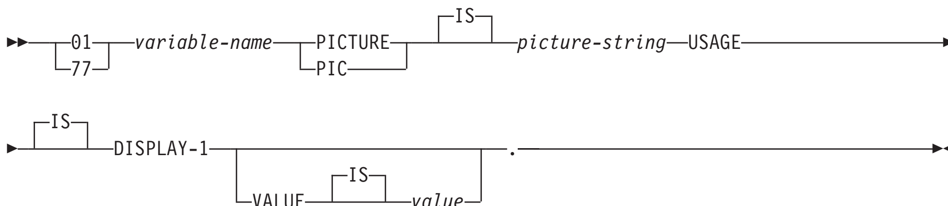
```
EXEC SQL CONNECT TO :dbname USER :userid USING :p-word
END-EXEC.
```

ただし、パスワードではブランクが有効なため、 p-word ホスト変数を VARCHAR データ項目として宣言する必要があります。こうすることにより、アプリケーションは CONNECT ステートメントのパスワードの有効な長さを明示的に指示することができます。以下はその例です。

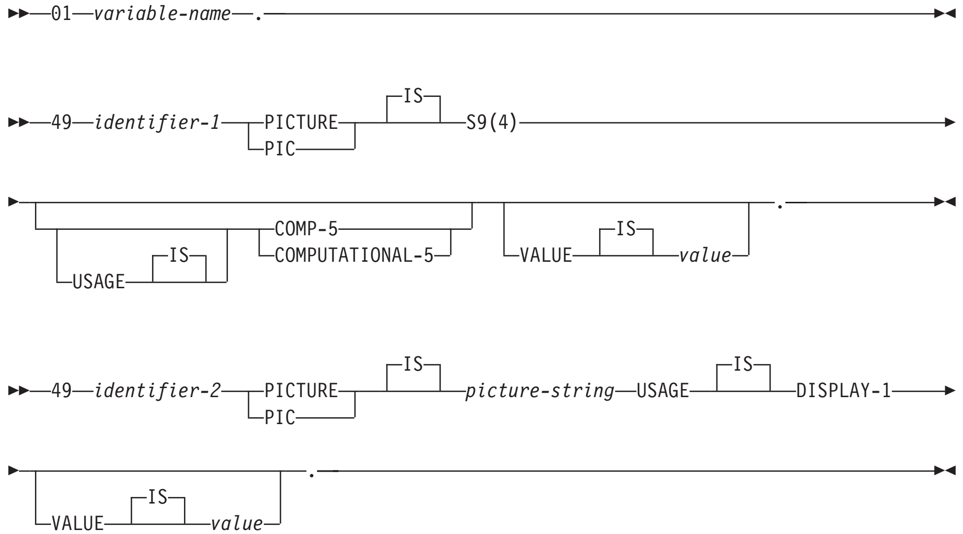
```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 dbname PIC X(8).
01 userid PIC X(8).
01 p-word.
    49 L PIC S9(4) COMP-5.
    49 D PIC X(18).
EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
    MOVE "sample" TO dbname.
    MOVE "userid" TO userid.
    MOVE "password" TO D OF p-word.
    MOVE 8          TO L of p-word.
EXEC SQL CONNECT TO :dbname USER :userid USING :p-word
END-EXEC.
```

COBOL におけるグラフィック・ホスト変数の構文: 固定長は、グラフィック・ホスト変数の構文を示します。

COBOL におけるグラフィック・ホスト変数の構文: 固定長



可変長



グラフィック・ホスト変数に関する考慮事項:

1. *Picture-string* の形式は、 *G(m)* でなければならない。 G は拡張できます (たとえば、″G(3)″ の代わりに ″GGG″ とできます)。
2. *m* は、1 ～ 127 までの長さの固定長ストリングである。
3. *m* は、1 ～ 16 350 までの長さの可変長ストリングである。
4. *m* が 16 336 よりも大きい場合、そのホスト変数は LONG VARGRAPHIC ストリングと見なされ、使用が制限される。

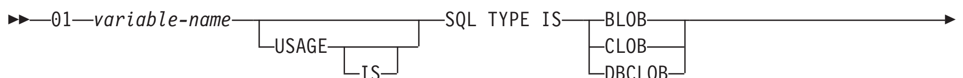
COBOL の標識変数

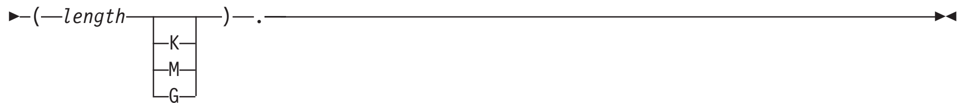
標識変数のデータ・タイプは、PIC S9(4) COMP-5 と宣言してください。

COBOL における LOB 宣言

COBOL における LOB ホスト変数の構文は、COBOL におけるラージ・オブジェクト (LOB) ホスト変数の宣言構文を示します。

COBOL における LOB ホスト変数の構文





LOB ホスト変数に関する考慮事項:

1. BLOB および CLOB の場合は、 1 <= lob-length <= 2 147 483 647 である。
2. DBCLOB の場合は、 1 <= lob-length <= 1 073 741 823 である。
3. SQL TYPE IS、BLOB、CLOB、DBCLOB、K、M、G は、大文字、小文字、またはその混合のいずれでもかまわない。
4. LOB 宣言内での初期化はできない。
5. ホスト変数名により、プリコンパイラー生成コード内の LENGTH および DATA に接頭語が付けられる。

BLOB の例:

宣言:

```
01 MY-BLOB USAGE IS SQL TYPE IS BLOB(2M).
```

この結果、以下の構造が生成されます。

```
01 MY-BLOB.
   49 MY-BLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-BLOB-DATA PIC X(2097152).
```

CLOB の例:

宣言:

```
01 MY-CLOB USAGE IS SQL TYPE IS CLOB(125M).
```

この結果、以下の構造が生成されます。

```
01 MY-CLOB.
   49 MY-CLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-CLOB-DATA PIC X(131072000).
```

DBCLOB の例:

宣言:

```
     01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(30000).
```

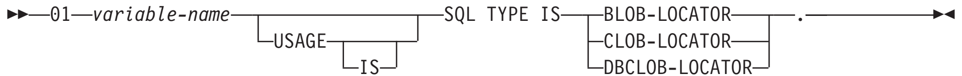
この結果、以下の構造が生成されます。

```
01 MY-DBCLOB.
   49 MY-DBCLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-DBCLOB-DATA PIC G(30000) DISPLAY-1.
```

COBOL における LOB ロケーター宣言

COBOL における LOB ロケーター・ホスト変数の構文は、COBOL におけるラージ・オブジェクト (LOB) ロケーター・ホスト変数の宣言の構文を示します。

COBOL における LOB ロケーター・ホスト変数の構文



LOB ロケーター・ホスト変数に関する考慮事項:

1. SQL TYPE IS、BLOB-LOCATOR、CLOB-LOCATOR、DBCLOB-LOCATOR は、大文字、小文字、またはその混合のいずれでもかまわない。
2. ロケーターの初期化はできない。

BLOB ロケーターの例 (他のタイプの LOB ロケーターの場合も同様):

宣言:

```
01 MY-LOCATOR USAGE SQL TYPE IS BLOB-LOCATOR.
```

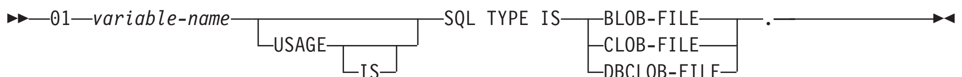
この結果、以下の宣言が生成されます。

```
01 MY-LOCATOR PIC S9(9) COMP-5.
```

COBOL におけるファイル参照宣言

COBOL におけるファイル参照ホスト変数の構文は、COBOL におけるファイル参照ホスト変数の宣言構文を示します。

COBOL におけるファイル参照ホスト変数の構文



- SQL TYPE IS、BLOB-FILE、CLOB-FILE、DBCLOB-FILE は、大文字、小文字、またはその混合のいずれでもかまわない。

BLOB ファイル参照の例 (他のタイプの LOB の場合も同様):

宣言:

```
01 MY-FILE USAGE IS SQL TYPE IS BLOB-FILE.
```

この結果、以下の宣言が生成されます。

```
01 MY-FILE.
   49 MY-FILE-NAME-LENGTH PIC S9(9) COMP-5.
   49 MY-FILE-DATA-LENGTH PIC S9(9) COMP-5.
   49 MY-FILE-FILE-OPTIONS PIC S9(9) COMP-5.
   49 MY-FILE-NAME PIC X(255).
```

COBOL でのホスト構造サポート

COBOL プリコンパイラーは、ホスト変数宣言セクション内のグループ・データ項目をサポートします。グループ・データ項目は特に、 SQL ステートメント内の基本データ項目の集合を手早く参照する方法を提供します。たとえば、以下のグループ・データ項目は、 SAMPLE データベースの STAFF 表にある列にアクセスするために使用することができます。

```
01 staff-record.
    05 staff-id       pic s9(4) comp-5.
    05 staff-name.
        49 l          pic s9(4) comp-5.
        49 d          pic x(9).
    05 staff-info.
        10 staff-dept pic s9(4) comp-5.
        10 staff-job  pic x(5).
```

宣言セクション内のグループ・データ項目は、上記のホスト変数のタイプを従属データ項目として含むことができます。これには、すべてのタイプのラージ・オブジェクトの他に、数値および文字のすべてのタイプが含まれます。グループ・データ項目は、最高 10 レベルまでネストさせることができます。 上の例のように、レベル 49 の従属項目には VARCHAR 文字タイプを宣言しなければならないことに注意してください。レベルが 49 でない場合は、VARCHAR は 2 つの従属項目を持つグループ・データ項目と見なされ、グループ・データ項目の宣言および使用の規則に従います。上記の例では、 `staff-info` はグループ・データ項目であり、 `staff-name` は VARCHAR です。同じ原則は、LONG VARCHAR、VARGRAPHIC および LONG VARGRAPHIC にも当てはまります。グループ・データ項目は、02 〜 49 の範囲のどのレベルにおいても宣言することができます。

グループ・データ項目とその従属項目には、以下の 4 とおりの使用法があります。

使用法 1

グループ全体を SQL ステートメント内の単一のホスト変数として参照します。

```
EXEC SQL SELECT id, name, dept, job
  INTO :staff-record
  FROM staff WHERE id = 10 END-EXEC.
```

プリコンパイラーは staff-record の参照を、 `staff-record` 内で宣言されたすべての従属項目をコンマで区切ったリストへと変換します。名前の重複を避けるために、各基本項目はグループ名により修飾されます。これは以下の使用法と同じです。

使用法 2

グループ・データ項目の 2 番目の使用法です。

```
EXEC SQL SELECT id, name, dept, job
  INTO
  :staff-record.staff-id,
  :staff-record.staff-name,
  :staff-record.staff-info.staff-dept,
  :staff-record.staff-info.staff-job
  FROM staff WHERE id = 10 END-EXEC.
```

注: staff-id への参照は、接頭部 staff-record. を使ったグループ名で修飾されており、ピュアな COBOL の場合のように staff-record の staff-id によってではありません。

staff-record の従属項目と同じ名前を持つホスト変数がその他にない場合は、上記のステートメントは使用法 3 と同様に、明示的なグループ修飾を取り除いてコーディングできます。

使用法 3

ここでは、特定のグループ項目を修飾しない、通常の COBOL の方式で従属項目が参照されています。

```
EXEC SQL SELECT id, name, dept, job
  INTO
  :staff-id,
  :staff-name,
  :staff-dept,
  :staff-job
  FROM staff WHERE id = 10 END-EXEC.
```

純粋な COBOL と同様に、特定の従属項目が固有に識別できれば、プリコンパイラーに受け入れられます。たとえば、staff-job が複数のグループに現れるとすると、プリコンパイラーはあいまいな参照であることを示すエラーを出します。

```
SQL0088N Host variable "staff-job" is ambiguous.
```

使用法 4

あいまい参照を解決するために、従属項目の部分修飾を使用することができます。たとえば、以下のようにします。

```
EXEC SQL SELECT id, name, dept, job
  INTO
  :staff-id,
  :staff-name,
  :staff-info.staff-dept,
  :staff-info.staff-job
  FROM staff WHERE id = 10 END-EXEC.
```

使用法 1 のような単一のグループ項目のみの参照は、コンマで区切った従属項目のリストに対応するため、このタイプの参照はエラーとなる場合があります。以下に例を示します。

```
EXEC SQL CONNECT TO :staff-record END-EXEC.
```

ここでの CONNECT ステートメントは、1 バイト文字ベースの変数を予期しています。 staff-record グループ・データ項目を与えると、このホスト変数は以下のようなプリコンパイル・エラーとなります。

```
SQL0087N Host variable "staff-record" is a structure used where
         structure references are not permitted.
```

この他に SQL0087N を引き起こすグループ項目の使用法には、 PREPARE、 EXECUTE IMMEDIATE、 CALL、標識変数、および SQLDA 参照を含むものがあります。このような状態では、前述の使用法 2、3 および 4 での個々の従属項目の参照がそうであるように、従属項目を 1 つしか持たないグループを使用することができます。

COBOL のインディケーター表

COBOL プリコンパイラーは、グループ・データ項目での使用に便利な標識変数の表の定義をサポートします。以下のように宣言します。

```
01 <indicator-table-name>.
    05 <indicator-name> pic s9(4) comp-5
                        occurs <table-size> times.
```

以下に例を示します。

```
01 staff-indicator-table.
    05 staff-indicator pic s9(4) comp-5
                       occurs 7 times.
```

このインディケーター表は、上記のグループ項目の最初の形式で効率的に使用できます。

```
EXEC SQL SELECT id, name, dept, job
  INTO :staff-record :staff-indicator
  FROM staff WHERE id = 10 END-EXEC.
```

ここでは、プリコンパイラーが staff-indicator はインディケーター表として宣言されていることを検出し、 SQL ステートメントの処理の際に、これを個々のインディケーターの参照に拡張します。 staff-indicator(1) は staff-record の staff-id、staff-indicator(2) は staff-record の staff-name、というように関連付けられます。

注: データ項目内の従属項目よりも k 個多いインディケーター項目がインディケーター表に存在する場合 (たとえば、staff-indicator に項目が 10 個ある場合は、 k=6 となります)、インディケーター表の終端の k 個の余分な項目は無視されます。同様に、インディケーター項目が従属項目よりも k 個少ない場合は、グループ項目内

の最後の k 個の項目は、対応するインディケーターを持ちません。 **SQL ステートメント内のインディケーター表のエレメントは、個別に参照できることに注意してください。**

COBOL グループ・データ項目での REDEFINES の使用

ホスト変数の宣言に、REDEFINES 文節を使用することができます。 REDEFINES 文節を使っていくつかのグループ・データ項目を宣言し、そのグループ・データ項目が SQL ステートメント内で全体として参照される場合、 REDEFINES 文節を含む従属項目は展開されません。以下に例を示します。

```
01 foo.
  10 a pic s9(4) comp-5.
  10 a1 redefines a pic x(2).
  10 b pic x(10).
```

SQL ステートメント内での foo の参照は、次のようになります。

```
... INTO :foo ...
```

上のステートメントは、次のステートメントと等価です。

```
... INTO :foo.a, :foo.b ...
```

つまり、従属項目 a1 は、REDEFINES 文節で宣言されていますが、そのような状況では自動的に展開されません。 a1 があいまいでない場合は、次のように、SQL ステートメント内の REDEFINES 文節で明示的に従属項目を参照することができます。

```
... INTO :foo.a1 ...
```

または

```
... INTO :a1 ...
```

BINARY/COMP-4 COBOL データ・タイプの使用

DB2 COBOL プリコンパイラーは、整数ホスト変数および標識が許可されているときには常に、BINARY、COMP、および COMP-4 データ・タイプの使用をサポートします。ただしそれは、ターゲット COBOL コンパイラーが BINARY、 COMP、または COMP-4 データ・タイプを、COMP-5 データ・タイプと等しく見なす (または等しく見なすようにできる) 場合に限ります。本書では、そのようなホスト変数および標識を、タイプ COMP-5 と示します。 COMP、COMP-4、BINARY COMP および COMP-5 を等価として扱う、 DB2 のサポートするターゲット・コンパイラーは、次のとおりです。

- IBM COBOL Set for AIX
- Micro Focus COBOL for AIX
- IBM COBOL Visual Set for OS/2 (-qbinary(native) オプション・セット付き)
- IBM VisualAge for COBOL for OS/2、 Windows NT および Windows 95、 (-qbinary(native) オプション・セット付き)

COBOL でサポートされる SQL データ・タイプ

特定の定義済み COBOL のデータ・タイプは、列のタイプに対応しています。ホスト変数として宣言できるのは、これらの COBOL データ・タイプのみです。

表34 は、それぞれの列タイプに対応する COBOL データ・タイプを示します。プリコンパイラーはホスト変数宣言を検出すると、該当する SQL データ・タイプの値を判別します。データベース・マネージャーはこの値を使用して、アプリケーションとの間でやりとりするデータを変換します。

ホスト変数で使用できるデータ記述すべてが認識されるわけではありません。 COBOL データ項目は、次の表に説明されているデータ項目と一致しなければなりません。他のデータ項目を使用すると、エラーが発生します。

注: DB2 ホスト言語で、DATALINK データ・タイプをサポートするホスト変数はありません。

表 34. COBOL 宣言にマップされた SQL データ・タイプ

SQL 列タイプ[1]	COBOL データ・タイプ	SQL 列タイプ記述
SMALLINT (500 または 501)	01 name PIC S9(4) COMP-5	16 ビットの符号付き整数
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.	サーバー上の CLOB エンティティーを識別する

表 34. COBOL 宣言にマップされた SQL データ・タイプ (続き)

SQL 列タイプ[1]	COBOL データ・タイプ	SQL 列タイプ記述
CLOB ファイル参照変数[4] (920 または 921)	01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE.	CLOB データを含むファイルの記述子
BLOB(*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 2 進ストリング
BLOB ロケーター変数[4] (960 または 961)	01 MY-BLOB-LOCATOR USAGE IS SQL TYPE IS BLOB-LOCATOR.	サーバー上の BLOB エンティティーを識別する。
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・タイプは、DBCS 環境でのみ使用可能です。		
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の 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.	サーバー上の 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**の下の最初の数字は標識変数が提供されないことを示し、 2 番目の数字は標識変数が提供されることを示します。標識変数は、ヌル値を示したり、切り捨てられたストリングの長さを保留するのに必要です。これらは、これらのデータ・タイプの SQLDA の SQLTYPE フィールドに示される値です。
2. FLOAT(*n*) ここで $0 < n < 25$ の場合、REAL と同義。 SQLDA での REAL と DOUBLE の違いは長さの値です (4 または 8)。
3. 次の SQL タイプは、DOUBLE と同義語。
   - FLOAT
   - FLOAT(*n*)。ここで、*n* の取る範囲は $24 < n < 54$。
   - DOUBLE PRECISION
4. これは列タイプではなく、ホスト変数タイプである。

サポートされている SQL データ・タイプそれぞれについて宣言されたホスト変数を含んだ、 SQL 宣言のサンプルを次に示します。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
*
  01 age        PIC S9(4) COMP-5.
  01 divis      PIC S9(9) COMP-5.
  01 salary     PIC S9(6)V9(3) COMP-3.
  01 bonus      USAGE IS COMP-1.
  01 wage       USAGE IS COMP-2.
  01 nm         PIC X(5).
  01 varchar.
     49 leng    PIC S9(4) COMP-5.
     49 strg    PIC X(14).
  01 longvchar.
     49 len     PIC S9(4) COMP-5.
     49 str     PIC X(6027).
  01 MY-CLOB USAGE IS SQL TYPE IS CLOB(1M).
  01 MY-CLOB-LOCATOR USAGE IS SQL TYPE IS CLOB-LOCATOR.
  01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE.
  01 MY-BLOB USAGE IS SQL TYPE IS BLOB(1M).
  01 MY-BLOB-LOCATOR USAGE IS SQL TYPE IS BLOB-LOCATOR.
  01 MY-BLOB-FILE USAGE IS SQL TYPE IS BLOB-FILE.
  01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(1M).
  01 MY-DBCLOB-LOCATOR USAGE IS SQL TYPE IS DBCLOB-LOCATOR.
  01 MY-DBCLOB-FILE USAGE IS SQL TYPE IS DBCLOB-FILE.
  01 MY-PICTURE PIC G(16000) USAGE IS DISPLAY-1.
  01 dt         PIC X(10).
  01 tm         PIC X(8).
  01 tmstmp     PIC X(26).
  01 wage-ind   PIC S9(4) COMP-5.
*
EXEC SQL END DECLARE SECTION END-EXEC.
```

以下に、サポートされる COBOL データ・タイプのその他の規則を示します。

- PIC S9 および COMP-3/COMP-5 がある場所では、これらが必須である。
- VARCHAR、LONG VARCHAR、VARGRAPHIC、LONG VARGRAPHIC およびすべての LOB 変数タイプを除くすべての列タイプについて、レベル番号 01 の代わりに 77 を使用できる。
- DECIMAL (p、s) 列タイプのホスト変数を宣言する際には、以下の規則が適用される。次のサンプルを参照してください。

```
01 identifier PIC S9(m)V9(n) COMP-3
```

  - 10 進小数点を表すには、V を使用する。
  - *n* および *m* の値は、1 またはそれ以上でなければならない。
  - $n + m$ の値が 31 を超えてはならない。
  - *s* の値は *n* の値と等しい。
  - *p* の値は $n + m$ の値と等しい。
  - 反復因数 *(n)* および *(m)* は任意で使用できる。次の例はすべて有効となります。

```
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 を使用できる。

- 配列は、COBOL プリコンパイラーではサポート**されません**。

COBOL での FOR BIT DATA

一定のデータベース列には FOR BIT DATA を宣言できます。通常は文字を含むこれらの列は、2 進情報を保持するために使用されます。 2 進データを含めることのできる COBOL ホスト変数のタイプは、 CHAR(*n*)、VARCHAR、 LONG VARCHAR、および BLOB データ・タイプです。これらのデータ・タイプは、FOR BIT DATA 属性を用いて処理を行う場合に使用してください。

COBOL での SQLSTATE および SQLCODE 変数

LANGLEVEL プリコンパイル・オプションを SQL92E の値とともに使用すると、次の 2 つの宣言をホスト変数として組み込めます。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 SQLSTATE PICTURE X(5).
01 SQLCODE  PICTURE S9(9) USAGE COMP.
.
.
.
EXEC SQL END DECLARE SECTION END-EXEC.
```

これらのいずれも指定しない場合は、SQLCODE 宣言はプリコンパイル中であると見なされます。また、’01’ は ’77’、’PICTURE’ は ’PIC’ とすることもできます。このオプションを使用するときには、INCLUDE SQLCA ステートメントを指定してはならないことに注意してください。

複数のソース・ファイルから成るアプリケーションでは、上の例のように、最初のソース・ファイルで SQLCODE および SQLSTATE 変数を定義することができます。

COBOL での日本語または中国語 (繁体字) EUC、および UCS-2 に関する考慮事項

eucJp または eucTW コード・セットで実行されている、または UCS-2 データベースに接続されているアプリケーションから送られてくるグラフィック・データはすべて、UCS-2 コード・ページ ID でタグ付けされます。アプリケーションの側では、グラフィック文字ストリングをデータベース・サーバーに送る前に UCS-2 に変換しておく必要があります。同様に、UCS-2 データベースからアプリケーションが取り出すグラフィック・データ、または EUC eucJP または eucTW コードで実行されているアプリケーションがデータベースから取り出すグラフィック・データも、 UCS-2 によってエンコー

ドされます。このため、アプリケーションの側では、UCS-2 データで表示しようとする場合を除き、内部的に UCS-2 からご使用のアプリケーションのコード・ページに変換する必要があります。

このような変換は、SQLDA へのデータのコピー前、また SQLDA からのデータのコピー後に実行する必要があるため、 UCS-2 への変換および UCS-2 からの変換は、ご使用のアプリケーションが担当することになります。 DB2 ユニバーサル・データベースでは、アプリケーションからアクセス可能な変換ルーチンは提供していません。その代わりに、ご使用のオペレーティング・システムから呼び出し可能なシステム・コールを使用してください。 UCS-2 データベースの場合は、VARCHAR および VARGRAPHIC スカラー関数の使用を考慮することができます。

これらの関数の詳細については、*SQL 解説書* を参照してください。一般的な EUC アプリケーション開発の指針については、 535ページの『日本語および中国語 (繁体字) EUC および UCS-2 コード・セットに関する考慮事項』を参照してください。

オブジェクト指向 COBOL

オブジェクト指向 COBOL を使用する場合、以下の規則に従う必要があります。

- SQL ステートメントは、1 コンパイル単位内の最初のプログラムまたはクラスだけに使用できます。これは、プリコンパイラーは、参照する最初の作業用ストレージ・セクションに一時作業データを挿入するためです。
- オブジェクト指向 COBOL プログラムでは、SQL ステートメントが含まれるそれぞれのクラスには、クラス・レベルの作業用ストレージ・セクションがなければなりません。これは、このセクションが空である場合にも当てはまります。このセクションを使用して、プリコンパイラーが生成したデータ定義を保管します。

第24章 FORTRAN でのプログラミング

FORTRAN でのプログラミングに関する考慮事項 723
FORTRAN の言語制限 723
FORTRAN での参照による呼び出し. . . 723
FORTRAN でのデバッグと注釈行 . . . 724
FORTRAN でのプリコンパイルに関する考慮事項. 724
FORTRAN の入力および出力ファイル . . . 724
FORTRAN のインクルード・ファイル . . . 724
FORTRAN でのファイルの組み込み. . . 727
FORTRAN での SQL ステートメントの組み込み 728
FORTRAN のホスト変数 729
FORTRAN でのホスト変数の命名 . . . 729
ホスト変数の宣言 729
FORTRAN の標識変数 732
FORTRAN における LOB 宣言 732
FORTRAN における LOB ロケーター宣言 733
FORTRAN におけるファイル参照宣言 733
FORTRAN でサポートされている SQL データ・タイプ 734
FORTRAN の SQLSTATE および SQLCODE 変数 736
FORTRAN でのマルチバイト文字セットに関する考慮事項 737
FORTRAN での日本語または中国語 (繁体字) EUC、および UCS-2 に関する考慮事項 . . 737

FORTRAN でのプログラミングに関する考慮事項

特定のホスト言語によるプログラミングの考慮事項について、以降のページで説明します。言語制限、ホスト言語別のインクルード・ファイル、組み込み SQL ステートメント、ホスト変数、およびサポートされるホスト変数のデータ・タイプについての情報が含まれています。

注: FORTRAN のサポートは DB2 バージョン 5 において確立され、今後 FORTRAN のサポートを拡張する予定はありません。たとえば、FORTRAN プリコンパイラーは、表名などの SQL オブジェクト ID を処理できません。これは 18 バイトより長いからです。 19 ～ 128 バイト長の表名などの、バージョン 5 より後の DB2 に追加された機能を使用したい場合には、 FORTRAN 以外の言語でアプリケーションを作成する必要があります。

FORTRAN の言語制限

以下の節では、FORTRAN 言語の制限について説明します。

FORTRAN での参照による呼び出し

API パラメーターの中には、呼び出し変数に値ではなくアドレスを必要とするものもあります。データベース・マネージャーは、それらのパラメーターの指定を単純化する GET ADDRESS、DEREFERENCE ADDRESS、および COPY MEMORY API を提供します。これらの API に関する詳しい説明は、 *管理 API 解説書* を参照してください。

© Copyright IBM Corp. 1993, 2001

FORTRAN でのデバッグと注釈行

一部の FORTRAN コンパイラーは、 1 列目が 'D' または 'd' である行を条件行とみなします。これらの行は、デバッグのためにコンパイルすることも、注釈として取り扱うことも可能です。プリコンパイラーは、1 列目が 'D' または 'd' である行を常に注釈とみなします。

FORTRAN でのプリコンパイルに関する考慮事項

以下の項目は、プリコンパイル処理に影響を及ぼします。

- プリコンパイラーは、連続記入行の 1 ～ 5 列目には数字、ブランク、およびタブ文字しか認めない。
- `.sqf` ソース・ファイルでは、ホレリス定数はサポートされない。

影響を与える可能性があるその他のプリコンパイルに関する考慮事項については、 *アプリケーション構築の手引き* を参照してください。

FORTRAN の入力および出力ファイル

デフォルトには、入力ファイルの拡張子は `.sqf` ですが、 TARGET プリコンパイル・オプションを使用した場合は、入力ファイルに任意の拡張子を付けることができます。

デフォルトには、出力ファイルの拡張子は `.f` (UNIX プラットフォーム) または `.for` (OS/2 および Windows ベースのプラットフォーム) ですが、 OUTPUT プリコンパイル・オプションを使用すれば、出力修正後のソース・ファイルに新しい名前やパスを指定できます。

FORTRAN のインクルード・ファイル

FORTRAN ホスト言語に特定のインクルード・ファイルのファイル拡張子は、 UNIX プラットフォームの場合は `.f` です。 OS/2 の場合は `.for` です。以下の FORTRAN インクルード・ファイルをアプリケーションで使用することができます。

SQL (sql.f)
このファイルには、バインダー、プリコンパイラー、およびエラー・メッセージ検索 API 用の言語固有プロトタイプが含まれています。また、システム定数も定義されています。

SQLAPREP (sqlaprep.f)
このファイルには、独自のプリコンパイラーの作成に必要な定義が入っています。

SQLCA (sqlca_cn.f, sqlca_cs.f)
このファイルは SQL 連絡域 (SQLCA) 構造を定義します。 SQLCA には、データベース・マネージャーが SQL ステートメントおよび API 呼び出しの実行に関するエラー情報をアプリケーションに提供するために使用する変数が含まれています。

FORTRAN アプリケーションには 2 つの SQLCA ファイルが提供されます。 `sqlca_cs.f` はデフォルトで、IBM SQL 互換フォーマットで SQLCA 構造を定義します。 `SQLCA NONE` オプションを指定して `sqlca_cn.f` ファイルをプリコンパイルすると、定義される SQLCA 構造のパフォーマンスが向上します。

SQLCA_92 (sqlca_92.f)

このファイルには、SQL 連絡域 (SQLCA) 構造の FIPS SQL92 Entry Level 準拠版が入っています。このファイルは、FIPS SQL92 Entry Level standard に適合する DB2 アプリケーションを作成する際に、`sqlca_cn.f` または `sqlca_cs.f` のどちらかのファイルの代りに組み込まれる必要があります。 LANGLEVEL プリコンパイラー・オプションに SQL92E が設定されていれば、 `sqlca_92.f` ファイルは DB2 プリコンパイラーによって自動的に組み込まれます。

SQLCODES (sqlcodes.f)

このファイルは、SQLCA 構造の SQLCODE フィールドで使用する定数を定義します。

SQLDA (sqldact.f)

このファイルは SQL 記述子域 (SQLDA) 構造を定義します。 SQLDA はアプリケーションとデータベース・マネージャーとの間でデータをやりとりするために使用されます。 FORTRAN プログラムで SQLDA をコーディングする方法の詳細は、 155ページの『SQLDA 構造の割り振り』を参照してください。

SQLEAU (sqleau.f)

このファイルには、DB2 セキュリティー監査 API に必要な定数および構造の定義が含まれています。これらの API を使用する場合は、プログラムにこのファイルを組み込む必要があります。このファイルにはまた、監査証跡レコード内のフィールドの定数およびキーワード値の定義も含まれています。これらの定義は、外部または取引先の監査証跡抽出プログラムで使用できます。

SQLENV (sqlenv.f)

このファイルは、データベース環境 API に対する言語固有の呼び出し、およびそれらのインターフェースの構造、定数、戻りコードを定義します。

SQLE819A (sqle819a.f)

データベースのコード・ページが 819 (ISO ラテン 1) の場合、このシーケンスは、ホスト CCSID 500 (EBCDIC 各国対応) バイナリー照合に従って、 FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQLE819B (sqle819b.f)

データベースのコード・ページが 819 (ISO ラテン 1) の場合、この

シーケンスは、ホスト CCSID 037 (EBCDIC 米国英語) バイナリー照合に従って、 FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQLE850A (sqle850a.f)
データベースのコード・ページが 850 (ASCII ラテン 1) の場合、このシーケンスは、ホスト CCSID 500 (EBCDIC 各国対応) バイナリー照合に従って、 FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQLE850B (sqle850b.f)
データベースのコード・ページが 850 (ASCII ラテン 1) の場合、このシーケンスは、ホスト CCSID 037 (EBCDIC 米国英語) バイナリー照合に従って、 FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQLE932A (sqle932a.f)
データベースのコード・ページが 932 (ASCII 日本語) の場合、このシーケンスは、ホスト CCSID 5035 (EBCDIC 日本語) バイナリー照合に従って、 FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQLE932B (sqle932b.f)
データベースのコード・ページが 932 (ASCII 日本語) の場合、このシーケンスは、ホスト CCSID 5026 (EBCDIC 日本語) バイナリー照合に従って、 FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQL1252A (sql1252a.f)
データベースのコード・ページが 1252 (Windows ラテン 1) の場合、このシーケンスは、ホスト CCSID 500 (EBCDIC 各国対応) バイナリー照合に従って、FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQL1252B (sql1252b.f)
データベースのコード・ページが 1252 (Windows ラテン 1) の場合、このシーケンスは、ホスト CCSID 037 (EBCDIC 米国英語) バイナリー照合に従って、FOR BIT DATA ではない文字ストリングをソートします。このファイルは CREATE DATABASE API が使用します。

SQLMON (sqlmon.f)
このファイルは、データベース・システム・モニター API に対する言語固有の呼び出し、およびそれらのインターフェースの構造、定数、戻りコードを定義します。

SQLSTATE (sqlstate.f)
このファイルは、SQLCA 構造の SQLSTATE フィールドで使用する定数を定義します。

SQLUTIL (sqlutil.f)
このファイルは、ユーティリティー API に対する言語固有の呼び出し、およびそれらのインターフェースに必要な構造、定数、コードを定義します。

FORTRAN でのファイルの組み込み

ファイルの組み込みには、EXEC SQL INCLUDE ステートメントを使用する方法と、FORTRAN INCLUDE ステートメントを使用する方法の 2 つがあります。プリコンパイラーは FORTRAN INCLUDE ステートメントを無視し、 EXEC SQL ステートメントを使用して組み込まれたファイルのみを処理します。

INCLUDE ファイルを検索する際、 DB2 FORTRAN プリコンパイラーはまず最初に現行ディレクトリーを検索し、次いで DB2INCLUDE 環境変数に指定されたディレクトリーを検索します。次の例を考えてみてください。

- `EXEC SQL INCLUDE payroll`

  INCLUDE ステートメントに指定されたファイルが単引用符 (') で囲まれていない場合、上記のとおり、プリコンパイラーは最初 `payroll.sqf` を検索し、次に `payroll.f` (OS/2 では `payroll.for`)、最後にそれがロックする各ディレクトリーを検索します。

- `EXEC SQL INCLUDE 'pay/payroll.f'`

  ファイル名が単引用符 (') で囲まれている場合、すでに説明したとおり、名前に拡張子は追加されません。 (OS/2 の場合、ファイルは `'pay¥payroll.for'` と指定されます。)

  単引用符 (') 内のファイル名に完全パスが含まれていない場合、 DB2INCLUDE の中身によってそのファイルの検索が行われ、何らかのパスが INCLUDE ファイル名に指定されます。たとえば、DB2 (AIX 版) では、 DB2INCLUDE は ‘`/disk2:myfiles/fortran`’ に設定され、プリコンパイラーは ‘`./pay/payroll.f`’、‘`/disk2/pay/payroll.f`’、 ‘`./myfiles/cobol/pay/payroll.f`’ の順に探索します。プリコンパイラー・メッセージには、実際にファイルが見つかったパスが表示されます。 OS/2 では、上記の例でスラッシュ (/) の代りに円記号 (¥) を使用し、拡張子 `'f'` の代りに `'for'` を使用します。

注: DB2INCLUDE の設定は、DB2 コマンド行プロセッサーによってキャッシュされます。何らかの CLP コマンドを実行した後に DB2INCLUDE の設定を変更する場合は、 TERMINATE コマンドを入力してから、データベースに接続し直し、あとは通常どおりプリコンパイルしてください。

FORTRAN での SQL ステートメントの組み込み

組み込み SQL ステートメントは次の 3 つのエレメントからなります。

エレメント	正しい FORTRAN 構文
キーワード	EXEC SQL
ステートメント・ストリング	ブランクを区切り文字として使用した有効な SQL ステートメント
ステートメント終了文字	ソース行の終わり

ソース行の終端は、ステートメント終了文字として機能します。行が継続する場合、ステートメント終了文字は連続記入行の最終行の終端に位置します。

以下に例を示します。

```
EXEC SQL SELECT COL INTO :hostvar FROM TABLE
```

組み込み SQL ステートメントには、以下の規則が適用されます。

- SQL ステートメントは 7 〜 72 列までの間でコーディングする。
- 全行の FORTRAN 注釈、または SQL 注釈を使用する。ただし、SQL ステートメント内で FORTRAN 行末注釈 '!' 文字は使用できません。それ以外の場所であれば、この注釈文字はホスト変数宣言を含めてどこでも使用できます。
- 組み込み SQL ステートメントをコーディングするさいは、 FORTRAN ステートメントにその必要がない場合でも、ブランクを区切り文字として使用する。
- 各 FORTRAN ソース行に対しては、SQL ステートメントのみを使用する。複数の行が必要なステートメントには、通常の FORTRAN の連結規則が適用されます。ただし、対になった EXEC SQL キーワードを複数行に分割してはなりません。
- SQL 注釈は、組み込み SQL ステートメントの一部となっている行であればどこでも使用することができる。この注釈は、動的に実行するステートメントでは使用できません。 SQL 注釈の形式は、ダブル・ダッシュ (--) の後に 0 個以上の文字ストリングが続き、行末で終了します。
- FORTRAN の注釈は、組み込み SQL ステートメント内の*ほとんど* の場所で使用することができる。例外は以下のとおりです。
  - 注釈は EXEC と SQL との間では使用できない。
  - 注釈は動的に実行されるステートメントでは使用できない。
  - 行末で ! を使用して FORTRAN の注釈をコーディングすることは、組み込み SQL ではサポートされていない。
- SQL ステートメント内で実定数を指定する際には、指数表記法を使用する。データベース・マネージャーは、SQL ステートメント内にある 10 進小数点を持った数字ストリングを、実定数ではなく 10 進定数と解釈します。
- 最初の実行可能 FORTRAN ステートメントよりも前にある SQL ステートメントでは、ステートメント番号は無効になる。 SQL ステートメントにこれと対応するステ

ートメント番号が付けられている場合、プリコンパイラーは、SQL ステートメントの直前に置かれるラベル付き CONTINUE ステートメントを生成します。

- SQL ステートメント内のホスト変数の参照時に宣言されたとおりのホスト変数を使用する。
- 行末文字およびタブ文字などの空白文字の置換は、次のように行われる。
  - 引用符の外 (ただし、SQL ステートメント内) では、行末文字または TAB 文字は単一スペースと置き換えられる。
  - 引用符内では、FORTRAN プログラムに適合した形でストリングが継続されていれば、行末文字は消去される。 TAB は修正されません。

行末および TAB に使用される実際の文字は、プラットフォームごとに異なります。たとえば、OS/2 は復帰 / 改行を行末に使用するのに対し、UNIX ベースのシステムは改行のみを使用します。

FORTRAN のホスト変数

ホスト変数は、SQL ステートメント内で参照される FORTRAN 言語変数となります。これにより、アプリケーションは入力データをデータベース・マネージャーに渡し、またデータベース・マネージャーから出力データを受け取ることができます。アプリケーションがコンパイルされると、コンパイラーはホスト変数を他の FORTRAN 変数として使用します。ホスト変数は以下の指示に従って命名、宣言、および使用してください。

FORTRAN でのホスト変数の命名

SQL プリコンパイラーは、宣言された名前によってホスト変数を識別します。この場合、以下の規則が適用されます。

- 変数名は 255 文字以内の長さで指定する。
- ホスト変数は、`SQL`、`sql`、`DB2`、または `db2` 以外の接頭部で開始する。これらはシステムが使用する予約語です。

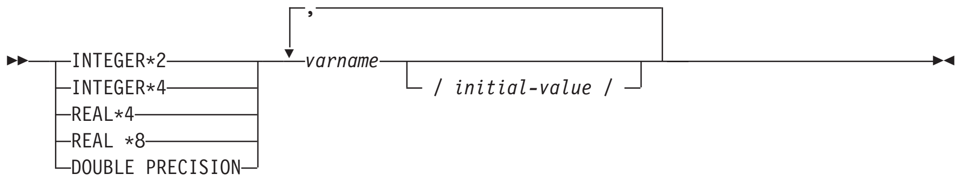
ホスト変数の宣言

ホスト変数宣言の識別には、SQL の宣言セクションを使用しなければなりません。これは、それ以降の SQL ステートメントで参照が可能なホスト変数をプリコンパイラーに知らせます。

FORTRAN プリコンパイラーは、有効な FORTRAN 宣言のサブセットのみを有効な変数宣言として認識します。これらの宣言は、数値変数または文字変数のいずれかを宣言します。数値ホスト変数は、数値の SQL 入出力値に対する入出力値として使用することができます。文字ホスト変数は任意の文字、日付、時間またはタイム・スタンプの SQL 入出力値に対する入出力値として使用できます。プログラマーは、出力変数が受け取る値を含めることのできる長さを持つようにコーディングしなければなりません。FORTRAN における数値ホスト変数の構文は、数値ホスト変数の構文を示します。

構造型のホスト変数の宣言方法についての詳細は、 363ページの『構造型ホスト変数の宣言』を参照してください。

FORTRAN における数値ホスト変数の構文

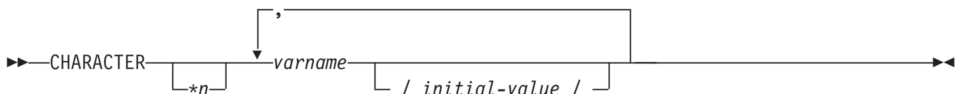


数値ホスト変数に関する考慮事項:

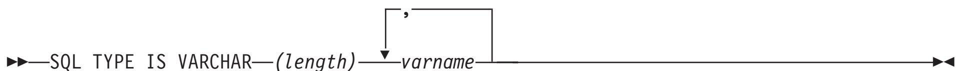
1. REAL*8 と DOUBLE PRECISION は同値である。
2. REAL*8 定数のインディケーターには、D ではなく E を使用する。

FORTRAN における文字変数の構文: 固定長は、文字ホスト変数の構文を示します。

FORTRAN における文字変数の構文: 固定長



可変長



文字ホスト変数に関する考慮事項:

1. *n の最大値は 254。
2. 長さ (length) が 1 と 32 672 の間である場合、ホスト変数のタイプは VARCHAR(448)。
3. 長さ (length) が 32 673 と 32 700 の間である場合、ホスト変数のタイプは LONG VARCHAR(SQLTYPE 456)。
4. 宣言内で VARCHAR ホスト変数および LONG VARCHAR ホスト変数を初期設定することは許可されていない。

VARCHAR の例:

宣言:

```
sql type is varchar(1000) my_varchar
```

この結果、以下の構造が生成されます。

```
 character    my_varchar(1000+2)
 integer*2    my_varchar_length
 character    my_varchar_data(1000)
 equivalence( my_varchar(1),
+             my_varchar_length )
 equivalence( my_varchar(3),
+             my_varchar_data )
```

アプリケーションは、たとえば、ホスト変数の内容の設定や検査のために、my_varchar_length および my_varchar_data の両方を処理できます。 SQL ステートメントでベース名 (ここでは、my_varchar) を使用して、 VARCHAR 全体を参照します。

LONG VARCHAR の例:

宣言:

```
sql type is varchar(10000) my_lvarchar
```

この結果、以下の構造が生成されます。

```
 character    my_lvarchar(10000+2)
 integer*2    my_lvarchar_length
 character    my_lvarchar_data(10000)
 equivalence( my_lvarchar(1),
+             my_lvarchar_length )
 equivalence( my_lvarchar(3),
+             my_lvarchar_data )
```

アプリケーションは、たとえば、ホスト変数の内容の設定や検査のために、my_lvarchar_length および my_lvarchar_data の両方を処理できます。 SQL ステートメントでベース名 (ここでは、my_lvarchar) を使用して、 LONG VARCHAR 全体を参照します。

注: 以下の例に示すような CONNECT ステートメントでは、 FORTRAN 文字ストリング・ホスト変数 dbname および userid の後続ブランクは、処理前に削除されます。

```
EXEC SQL CONNECT TO :dbname USER :userid USING :passwd
```

しかし、パスワードにはブランクも有効であるため、パスワードのホスト変数を VARCHAR として宣言し、実際のパスワードの長さを反映するように長さ (length) フィールドを設定しなければなりません。

```
EXEC SQL BEGIN DECLARE SECTION
  character*8 dbname, userid
  sql type is varchar(18) passwd
EXEC SQL END DECLARE SECTION
character*18 passwd_string
equivalence(passwd_data,passwd_string)
dbname = 'sample'
userid = 'userid'
```

```
passwd_length= 8
passwd_string = 'password'
EXEC SQL CONNECT TO :dbname USER :userid USING :passwd
```

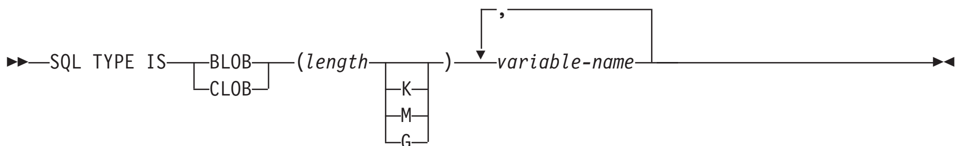
FORTRAN の標識変数

標識変数は、INTEGER*2 データ・タイプとして宣言します。

FORTRAN における LOB 宣言

FORTRAN におけるラージ・オブジェクト (LOB) ホスト変数の構文は、FORTRAN におけるラージ・オブジェクト (LOB) ホスト変数宣言の構文を示します。

FORTRAN におけるラージ・オブジェクト (LOB) ホスト変数の構文



LOB ホスト変数に関する考慮事項:

1. GRAPHIC タイプは、FORTRAN ではサポートされていない。
2. SQL TYPE IS、BLOB、CLOB、K、M、G は、大文字、小文字、またはその混合のいずれでもかまわない。
3. BLOB および CLOB の場合は、 1 <= lob-length <= 2 147 483 647 である。
4. LOB 宣言内での LOB の初期化はできない。
5. ホスト変数名には、プリコンパイラー生成コードにおいて 'length' と 'data' という接頭部がある。

BLOB の例:

宣言:

```
sql type is blob(2m) my_blob
```

この結果、以下の構造が生成されます。

```
 character    my_blob(2097152+4)
 integer*4    my_blob_length
 character    my_blob_data(2097152)
 equivalence( my_blob(1),
+             my_blob_length )
 equivalence( my_blob(5),
+             my_blob_data )
```

CLOB の例:

宣言:

```
sql type is clob(125m) my_clob
```

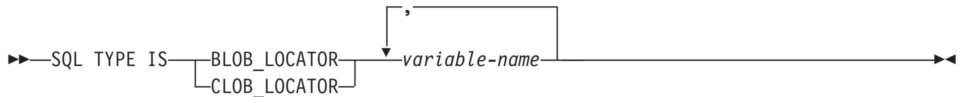
この結果、以下の構造が生成されます。

```
 character    my_clob(131072000+4)
 integer*4    my_clob_length
 character    my_clob_data(131072000)
 equivalence( my_clob(1),
+             my_clob_length )
 equivalence( my_clob(5),
+             my_clob_data )
```

FORTRAN における LOB ロケーター宣言

FORTRAN におけるラージ・オブジェクト (LOB) ロケーター・ホスト変数の構文は、FORTRAN におけるラージ・オブジェクト (LOB) ロケーター・ホスト変数の宣言の構文を示します。

FORTRAN におけるラージ・オブジェクト (LOB) ロケーター・ホスト変数の構文



LOB ロケーター・ホスト変数に関する考慮事項:

1. GRAPHIC タイプは、FORTRAN ではサポートされていない。
2. SQL TYPE IS、BLOB_LOCATOR、CLOB_LOCATOR は、大文字、小文字、またはその混合のいずれでもかまわない。
3. ロケーターの初期化はできない。

CLOB ロケーターの例 (BLOB ロケーターと同様):

宣言:

```
SQL TYPE IS CLOB_LOCATOR my_locator
```

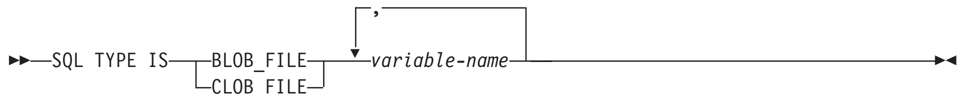
この結果、以下の宣言が生成されます。

```
 integer*4 my_locator
```

FORTRAN におけるファイル参照宣言

FORTRAN におけるファイル参照ホスト変数の構文は、FORTRAN におけるファイル参照ホスト変数の宣言の構文を示します。

FORTRAN におけるファイル参照ホスト変数の構文



ファイル参照ホスト変数に関する考慮事項:

1. GRAPHIC タイプは、FORTRAN ではサポートされていない。
2. SQL TYPE IS、BLOB_FILE、CLOB_FILE は、大文字、小文字、またはその混合のいずれでもかまわない。

BLOB ファイル参照変数の例 (CLOB ファイル参照変数と同様):

```
SQL TYPE IS BLOB_FILE my_file
```

この結果、以下の宣言が生成されます。

```
 character     my_file(267)
 integer*4     my_file_name_length
 integer*4     my_file_data_length
 integer*4     my_file_file_options
 character*255 my_file_name
 equivalence(  my_file(1),
+              my_file_name_length )
 equivalence(  my_file(5),
+              my_file_data_length )
 equivalence(  my_file(9),
+              my_file_file_options )
 equivalence(  my_file(13),
+              my_file_name )
```

FORTRAN でサポートされている SQL データ・タイプ

特定の定義済み FORTRAN のデータ・タイプは、データベース・マネージャーの列のタイプに対応しています。ホスト変数として宣言できるのは、これらの FORTRAN データ・タイプのみです。

表35 は、それぞれの列タイプに対応する FORTRAN データ・タイプを示します。プリコンパイラーはホスト変数宣言を検出すると、該当する SQL データ・タイプの値を判別します。データベース・マネージャーはこの値を使用して、アプリケーションとの間でやりとりするデータを変換します。

注: DB2 ホスト言語で、DATALINK データ・タイプをサポートするホスト変数はありません。

表 35. FORTRAN 宣言にマップされる SQL データ・タイプ

SQL 列タイプ[1]	FORTRAN データ・タイプ	SQL 列タイプ記述
SMALLINT (500 または 501)	INTEGER*2	16 ビットの符号付き整数

表 35. *FORTRAN 宣言にマップされる SQL データ・タイプ (続き)*

SQL 列タイプ[1]	FORTRAN データ・タイプ	SQL 列タイプ記述
INTEGER (496 または 497)	INTEGER*4	32 ビットの符号付き整数
REAL[2] (480 または 481)	REAL*4	単精度浮動小数点
DOUBLE[3] (480 または 481)	REAL*8	倍精度浮動小数点
DECIMAL(*p*,*s*) (484 または 485)	厳密に対応するものがない ; REAL*8 を使用	パック 10 進数
CHAR(*n*) (452 または 453)	CHARACTER**n*	長さが *n* の固定長文字ストリング。 *n* の範囲は 1 ～ 254
VARCHAR(*n*) (448 または 449)	SQL TYPE IS VARCHAR(*n*) *n* の範囲は 1 ～ 32 672	可変長文字ストリング
LONG VARCHAR (456 または 457)	SQL TYPE IS VARCHAR(*n*) *n* の範囲は 32 673 ～ 32 700	long 可変長文字ストリング
CLOB(*n*) (408 または 409)	SQL TYPE IS CLOB (n) *n* の範囲は 1 ～ 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)	SQL TYPE IS CLOB_LOCATOR	サーバー上の 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)	SQL TYPE IS CLOB_FILE	CLOB データを含むファイルの記述子
BLOB(*n*) (404 または 405)	SQL TYPE IS BLOB(n) *n* の範囲は 1 ～ 2 147 483 647	ラージ・オブジェクト可変長 2 進ストリング
BLOB ロケーター変数[4] (960 または 961)	SQL TYPE IS BLOB_LOCATOR	サーバー上の 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)	SQL TYPE IS BLOB_FILE	BLOB データを含むファイルの記述子
DATE (384 または 385)	CHARACTER*10	10 バイトの文字ストリング
TIME (388 または 389)	CHARACTER*8	8 バイトの文字ストリング
TIMESTAMP (392 または 393)	CHARACTER*26	26 バイトの文字ストリング

注:

1. **SQL 列タイプ**の下の最初の数字は標識変数が提供されないことを示し、 2 番目の数字は標識変数が提供されることを示します。標識変数は、ヌル値を示したり、切り捨てられたストリングの長さを保留するのに必要です。これらは、これらのデータ・タイプの SQLDA の SQLTYPE フィールドに示される値です。
2. FLOAT(*n*) ここで $0 < n < 25$ の場合、REAL と同義。 SQLDA での REAL と DOUBLE の違いは長さの値です (4 または 8)。
3. 次の SQL タイプは、DOUBLE と同義語。
   - FLOAT
   - FLOAT(*n*)。ここで、*n* の取る範囲は $24 < n < 54$。
   - DOUBLE PRECISION
4. これは列タイプではなく、ホスト変数タイプである。

サポートされているデータ・タイプそれぞれについて宣言されたホスト変数を含む、SQL 宣言のサンプルを以下に示します。

```
EXEC SQL BEGIN DECLARE SECTION
  INTEGER*2     AGE  /26/
  INTEGER*4     DEPT
  REAL*4        BONUS
  REAL*8        SALARY
  CHARACTER     MI
  CHARACTER*112 ADDRESS
  SQL TYPE IS VARCHAR (512) DESCRIPTION
  SQL TYPE IS VARCHAR (32000) COMMENTS
  SQL TYPE IS CLOB (1M) CHAPTER
  SQL TYPE IS CLOB_LOCATOR CHAPLOC
  SQL TYPE IS CLOB_FILE  CHAPFL
  SQL TYPE IS BLOB (1M) VIDEO
  SQL TYPE IS BLOB_LOCATOR VIDLOC
  SQL TYPE IS BLOB_FILE VIDFL
  CHARACTER*10  DATE
  CHARACTER*8   TIME
  CHARACTER*26  TIMESTAMP
  INTEGER*2     WAGE_IND
EXEC SQL END DECLARE SECTION
```

以下に、サポートされる FORTRAN データ・タイプのその他の規則を示します。

- VARCHAR、LONG VARCHAR、または CLOB ホスト変数を使用して、 254 文字よりも長い動的 SQL ステートメントを定義できる。

FORTRAN の SQLSTATE および SQLCODE 変数

LANGLEVEL プリコンパイル・オプションを SQL92E の値とともに使用すると、次の 2 つの宣言をホスト変数として組み込めます。

```
EXEC SQL BEGIN DECLARE SECTION;
  CHARACTER*5 SQLSTATE
  INTEGER     SQLCOD
  .
  .
  .
EXEC SQL END DECLARE SECTION
```

これらのどちらも指定されない場合、プリコンパイル・ステップの間、SQLCOD 宣言が仮定されます。変数には 'SQLSTATE' または 'SQLSTA' と名前が付けられます。このオプションを使用するときには、INCLUDE SQLCA ステートメントを指定してはならないことに注意してください。

複数のソース・ファイルがあるアプリケーションの場合、各ソース・ファイルに SQLCOD と SQLSTATE の宣言が上記のように組み込まれることがあります。

FORTRAN でのマルチバイト文字セットに関する考慮事項

FORTRAN では、グラフィック (マルチバイト) ホスト変数データ・タイプはサポートされていません。 `character` データ・タイプによって、混合文字ホスト変数だけがサポートされています。グラフィック・データを含むユーザー SQLDA を作成することは可能です。

FORTRAN での日本語または中国語 (繁体字) EUC、および UCS-2 に関する考慮事項

eucJp または eucTW コード・セットで実行されている、または UCS-2 データベースに接続されているアプリケーションから送られてくるグラフィック・データはすべて、UCS-2 コード・ページ ID でタグ付けされます。アプリケーションの側では、グラフィック文字ストリングをデータベース・サーバーに送る前に UCS-2 に変換しておく必要があります。同様に、UCS-2 データベースからアプリケーションが取り出すグラフィック・データ、または EUC eucJP または eucTW コードで実行されているアプリケーションがデータベースから取り出すグラフィック・データも、 UCS-2 によってエンコードされます。このため、アプリケーションの側では、UCS-2 データで表示しようとする場合を除き、内部的に UCS-2 からご使用のアプリケーションのコード・ページに変換する必要があります。

このような変換は、SQLDA へのデータのコピー前、また SQLDA からのデータのコピー後に実行する必要があるため、 UCS-2 への変換および UCS-2 からの変換は、ご使用のアプリケーションが担当することになります。 DB2 ユニバーサル・データベースでは、アプリケーションからアクセス可能な変換ルーチンは提供していません。その代わりに、ご使用のオペレーティング・システムから呼び出し可能なシステム・コールを使用してください。 UCS-2 データベースの場合は、VARCHAR および VARGRAPHIC スカラー関数の使用を考慮することができます。

これらの関数の詳細については、*SQL 解説書* を参照してください。

一般的な EUC アプリケーション開発の指針については、 535ページの『日本語および中国語 (繁体字) EUC および UCS-2 コード・セットに関する考慮事項』を参照してください。

第25章 REXX でのプログラミング

REXX でのプログラミングに関する考慮事項 739
REXX の言語制限 740
REXX における SQLEXEC、SQLDBS、および SQLDB2 の登録 740
REXX での SQL ステートメントの組み込み 741
REXX のホスト変数 743
REXX でのホスト変数の命名 743
REXX でのホスト変数の参照 743
REXX の標識変数 744
事前定義 REXX 変数 744
REXX の LOB ホスト変数 746
REXX における LOB ロケーター宣言 746
REXX における LOB ファイル参照宣言 747
REXX での LOB ホスト変数のクリア 748
REXX でサポートされている SQL データ・タイプ 749
REXX でのカーソルの使用 750
REXX の実行要件 751
REXX のバインド・ファイル 752
REXX の API 構文 752
REXX のストアード・プロシージャー . . . 754
REXX におけるストアード・プロシージャーの呼び出し 754
REXX のクライアントに関する考慮事項 755
REXX のサーバーに関する考慮事項 756
SQLDA 10 進フィールドの精度と SCALE 値の検索 756
REXX の日本語または中国語 (繁体字) EUC に関する考慮事項 756

REXX でのプログラミングに関する考慮事項

特定のホスト言語によるプログラミングの考慮事項について、以降のページで説明します。組み込み SQL ステートメント、言語制限、およびサポートされるホスト変数のデータ・タイプについての情報が含まれます。

注: REXX のサポートは DB2 バージョン 5 において確立され、今後 REXX のサポートを拡張する予定はありません。たとえば、REXX は、表名などの SQL オブジェクト ID を処理できません。これは 18 バイトより長いからです。 19 〜 128 バイト長の表名などの、バージョン 5 より後の DB2 に追加された機能を使用したい場合には、 REXX 以外の言語でアプリケーションを作成する必要があります。

REXX はインタープリター言語なので、プリコンパイラー、コンパイラー、またはリンカーを使用しません。その代り、3 つの DB2 API を使用して、REXX の DB2 アプリケーションを作成します。 DB2 のさまざまなエレメントをアクセスするには、以下の API を使用してください。

SQLEXEC
SQL 言語をサポートします。

SQLDBS
DB2 API のコマンド形式のバージョンをサポートします。

SQLDB2
コマンド行プロセッサーへの REXX 特定のインターフェースをサポートしま

す。このインターフェースを使用する際の制限および詳細については、 752ページの『REXX の API 構文』を参照してください。

REXX の言語制限

ステートメントまたはコマンドにトークンを含めることができます。それは、REXX 変数に対応できる SQLEXEC、SQLDBS、および SQLDB2 ルーチンに渡されます。 この場合、REXX インタープリターは SQLEXEC または SQLDB2 を呼び出す前に、変数の値を置換します。

この状態を避けるには、ステートメントのストリングを、引用符 (' ' または
" ") で囲んでください。引用符で囲まない場合、変数名と同じトークンがあると、そのトークンは REXX インタープリターによって解決され、SQLEXEC、SQLDBS、または SQLDB2 ルーチンには渡されません。

REXX/SQL では、複合 SQL はサポートされません。

REXX/SQL ストアード・プロシージャーは、 OS/2 および Windows 32 ビット・オペレーティング・システムでサポートされていますが、 AIX ではサポートされていません。

REXX における SQLEXEC、SQLDBS、および SQLDB2 の登録

アプリケーションで、DB2 API のいずれかを使用する前、または SQL ステートメントを発行する前に、SQLDBS、SQLDB2、および SQLEXEC ルーチンを登録しなければなりません。この登録は、REXX インタープリターに REXX/SQL の入り口点を知らせます。 OS/2 プラットフォームと AIX プラットフォームでは、登録のための方法が少し異なります。次の例は、それぞれのルーチンの登録について正しい構文を示します。

OS/2 または Windows での登録のサンプル

```
/* ------------ Register SQLDBS with REXX  -------------------------*/
If Rxfuncquery('SQLDBS')  <> 0 then
    rcy = Rxfuncadd('SQLDBS','DB2AR','SQLDBS')
If rcy ¥= 0 then
    do
      say 'SQLDBS was not successfully added to the REXX environment'
      signal rxx_exit
    end

/* ------------ Register SQLDB2 with REXX  -------------------------*/
If Rxfuncquery('SQLDB2')  <> 0 then
    rcy = Rxfuncadd('SQLDB2','DB2AR','SQLDB2')
If rcy ¥= 0 then
    do
      say 'SQLDB2 was not successfully added to the REXX environment'
      signal rxx_exit
    end
```

```
/* ----------------- Register SQLEXEC with REXX  --------------------*/
If Rxfuncquery('SQLEXEC') <> 0 then
    rcy = Rxfuncadd('SQLEXEC','DB2AR','SQLEXEC')
If rcy ¥= 0 then
    do
      say 'SQLEXEC was not successfully added to the REXX environment'
      signal rxx_exit
    end
```

AIX での登録のサンプル

```
/* ------------ Register SQLDBS, SQLDB2 and SQLEXEC with REXX --------*/
rcy = SysAddFuncPkg("db2rexx")
If rcy ¥= 0 then
  do
    say 'db2rexx was not successfully added to the REXX environment'
    signal rxx_exit
  end
```

OS/2 では、RxFuncAdd コマンドはすべてのセッションにつき 1 回だけ実行する必要があります。

AIX では、SysAddFuncPkg をすべての REXX/SQL アプリケーションで実行しなければなりません。

RXfuncadd API および SysAddFuncPkg API の詳細については、それぞれ OS/2 および AIX 版の REXX の資料に記載されています。

REXX での SQL ステートメントの組み込み

SQL ステートメントの処理には、SQLEXEC ルーチンを使用してください。 SQLEXEC ルーチンの文字ストリング引き数は以下のエレメントから成ります。

- SQL キーワード
- 事前定義 ID
- ステートメント・ホスト変数

有効な SQL ステートメントを SQLEXEC ルーチンに渡すことにより、それぞれの要素を要求します。以下の構文を使用します。

```
CALL SQLEXEC 'statement'
```

SQL ステートメントは複数行に渡って継続が可能です。ステートメントのそれぞれの部分は単一引用符で囲み、以下に示すように、追加ステートメントのテキストとの区切りとしてコンマを使用してください。

```
CALL SQLEXEC 'SQL text',
             'additional text',
                  .
                  .
                  .
             'final text'
```

以下に、REXX での組み込み SQL の例を示します。

```
statement = "UPDATE STAFF SET JOB = 'Clerk' WHERE JOB = 'Mgr'"
CALL SQLEXEC 'EXECUTE IMMEDIATE :statement'
IF ( SQLCA.SQLCODE < 0) THEN
   SAY 'Update Error:  SQLCODE = ' SQLCA.SQLCODE
```

この例では、更新が正常に行われたかどうかを判断するために、 SQLCA 構造の SQLCODE フィールドが検査されています。

組み込み SQL ステートメントには、以下の規則が適用されます。

- 次の組み込み SQL ステートメントは、SQLEXEC ルーチンに直接渡すことができる。

  CALL
  CLOSE
  COMMIT
  CONNECT
  CONNECT TO
  CONNECT RESET
  DECLARE
  DESCRIBE
  DISCONNECT
  EXECUTE
  EXECUTE IMMEDIATE
  FETCH
  FREE LOCATOR
  OPEN
  PREPARE
  RELEASE
  ROLLBACK
  SET CONNECTION

  他の SQL ステートメントは、SQLEXEC ルーチンとともに EXECUTE IMMEDIATE ステートメント、または PREPARE と EXECUTE ステートメントを使用して動的に処理される。
- REXX では、CONNECT および SET CONNECTION ステートメントでホスト変数を使用することはできない。
- カーソル名およびステートメント名は、次のように事前定義される。

c1 〜 c100

WITH HOLD オプションを使用せずに宣言した、範囲が *c1* 〜 *c50* のカーソルのカーソル名、および WITH HOLD オプションを使用して宣言した、範囲が *c51* 〜 *c100* のカーソルのカーソル名です。

カーソル名の ID は、DECLARE、OPEN、FETCH、および CLOSE ステートメントに使用されます。これは、SQL 要求で使用されるカーソルを識別します。

s1 〜 s100

範囲が *s1* 〜 *s100* のステートメント名です。

ステートメント名の ID は、DECLARE、DESCRIBE、PREPARE、および EXECUTE ステートメントに使用されます。

カーソル名およびステートメント名には、事前定義 ID を使用しなければならない。その他の名前は認められません。

- カーソルを宣言する際には、DECLARE ステートメント内のカーソル名とステートメント名が対応していなければならない。たとえば、*c1* をカーソル名として使用する場合、ステートメント名には *s1* を使用しなければなりません。
- SQL ステートメント内では注釈を使用しない。

REXX のホスト変数

ホスト変数は、SQL ステートメント内で参照される REXX の言語変数です。これにより、アプリケーションは入力データを DB2 に渡し、また DB2 から出力データを受け取ることができます。 REXX アプリケーションは LOB ロケーターおよび LOB ファイル参照変数を除き、ホスト変数を宣言する必要はありません。ホスト変数のデータ・タイプおよびサイズは、変数の参照の実行時に決定されます。ホスト変数の命名および使用の際には、以下の規則を適用してください。

REXX でのホスト変数の命名

正しく命名された REXX 変数は、すべてホスト変数として使用できます。変数名の長さは 64 文字までです。ピリオドを変数名の最後の文字として使用しないでください。ホスト変数名には、英字、数字、および @、_、!、.、?、$ といった文字を使用できます。

REXX でのホスト変数の参照

REXX インタープリターは、プロシージャー内のストリングで引用符で囲まれていないものをすべて検査します。ストリングが現行の REXX 変数プール内の変数を表している場合には、 REXX がそのストリングを現行値と置き換えます。以下に、REXX におけるホスト変数の参照方法を示します。

```
CALL SQLEXEC 'FETCH C1 INTO :cm'
SAY 'Commission = ' cm
```

文字ストリングが数値データ・タイプに変換されないようにするため、ストリングを以下の例のように単一引用符で囲んでください。

```
VAR = '100'
```

REXX は、3 バイトの文字ストリング 100 に変数 *VAR* をセットします。単一引用符がストリングの一部になっている場合は、次の例に従ってください。

```
VAR = "'100'"
```

CHARACTER フィールドに数値データを挿入する場合、REXX インタープリターは、数値データを整数データとみなします。したがって、数値ストリングを明示的に連結して、単一引用符で囲む必要があります。

REXX の標識変数

REXX における標識変数のデータ・タイプは、10 進小数点を持たない数です。以下に、INDICATOR キーワードを使用した REXX における標識変数の例を示します。

```
CALL SQLEXEC 'FETCH C1 INTO :cm INDICATOR :cmind'
IF ( cmind < 0 )
   SAY 'Commission is NULL'
```

上記の例では、cmind が負の値かどうか検査されます。負の値ではない場合、アプリケーションは cm の戻り値を使用することができます。負の値の場合、取り出される値は NULL で、cm は使用されません。この場合、データベース・マネージャーはホスト変数の値を変更しません。

事前定義 REXX 変数

SQLEXEC、SQLDBS および SQLDB2 は一定の操作の結果として、事前定義 REXX 変数をセットします。それらの変数は以下のとおりです。

RESULT

各操作により、戻りコードがセットされます。使用される値は以下のとおりです。

n — *n* は、フォーマットされたメッセージのバイト数を示す正の値です。この値を戻すのは GET ERROR MESSAGE API のみです。

0 — API が実行されています。REXX 変数 SQLCA には、API の完了状況が含まれます。 SQLCA.SQLCODE がゼロでない場合は、その値に関連したテキスト・メッセージが SQLMSG に含まれます。

-1 — API を完了するために十分なメモリーがありません。要求されたメッセージは戻されません。

-2 — SQLCA.SQLCODE が 0 にセットされます。メッセージは戻されません。

-3 — SQLCA.SQLCODE に無効な SQLCODE が含まれています。メッセージは戻されません。

-6 SQLCA REXX 変数が作成できません。これは、十分なメモリーがないか、または何らかの理由で REXX 変数プールが使用できないということを示します。

-7 SQLMSG REXX 変数が作成できません。これは、十分なメモリーがないか、または何らかの理由で REXX 変数プールが使用できないということを示します。

-8 REXX 変数プールから SQLCA.SQLCODE REXX 変数を取り出すことができません。

-9 取り出しの際に、SQLCA.SQLCODE REXX 変数の切り捨てが行われました。この変数の長さは最大 5 バイトまでです。

-10 SQLCA.SQLCODE REXX 変数を、ASCII から有効な長整数に変換できません。

-11 REXX 変数プールから SQLCA.SQLERRML REXX 変数を取り出すことができます。

-12 取り出しの際に、SQLCA.SQLERRML REXX 変数の切り捨てが行われました。この変数の長さは最大 2 バイトまでです。

-13 SQLCA.SQLERRML REXX 変数を、ASCII から有効な短整数に変換できません。

-14 REXX 変数プールから SQLCA.SQLERRMC REXX 変数を取り出すことができません。

-15 取り出しの際に、SQLCA.SQLERRMC REXX 変数の切り捨てが行われました。この変数の長さは最大 70 バイトまでです。

-16 エラー・テキストに指定された REXX 変数をセットできません。

-17 REXX 変数プールから SQLCA.SQLSTATE REXX 変数を取り出すことができません。

-18 取り出しの際に、SQLCA.SQLSTATE REXX 変数の切り捨てが行われました。この変数の長さは最大 2 バイトまでです。

注: -8 ～ -18 の値を戻すのは、GET ERROR MESSAGE API のみです。

SQLMSG

SQLCA.SQLCODE が 0 でない場合、この値にはエラー・コードに関連したテキスト・メッセージが含まれます。

SQLISL

分離レベルです。使用される値は以下のとおりです。

RR 反復可能読み取り。

RS 読み取り固定。

CS カーソル固定。これがデフォルトです。

UR 非コミット読み取り。

NC コミットなし (NC は、一部の AS/400 サーバーでしかサポートされていません)。

SQLCA

SQL ステートメントの処理の後に更新された SQLCA 構造と、DB2 API が呼び出されます。この構造の項目については、*管理 API 解説書* で説明されています。

SQLRODA

CALL ステートメントを使用して呼び出される、ストアード・プロシージャーの入出力 SQLDA 構造です。データベース・アプリケーション・リモート・インターフェース (DARI) API を使用して呼び出される、出力 SQLDA ストアード・プロシージャーでもあります。 この構造の項目については、*管理 API 解説書* で説明されています。

SQLRIDA

データベース・アプリケーション・リモート・インターフェース (DARI) API を使用して呼び出される、ストアード・プロシージャーの入力 SQLDA 構造です。この構造の項目については、*管理 API 解説書* で説明されています。

SQLRDAT

データベース・アプリケーション・リモート・インターフェース (DARI) API を使用して呼び出される、サーバー・プロシージャーの SQLCHAR 構造です。この構造の項目については、*管理 API 解説書* で説明されています。

REXX の LOB ホスト変数

REXX ホスト変数に LOB 列を取り出してくる場合、この列は単純 (つまり、カウントはされない) ストリングとして保管されます。これは、文字ベースの SQL タイプ (たとえば、CHAR、VARCHAR、GRAPHIC、LONG など) すべてと同じ方法で処理されます。入力では、ホスト変数の内容のサイズが 32K を超える場合、または以下に説明する基準を満たしている場合には、適切な LOB タイプが割り当てられます。

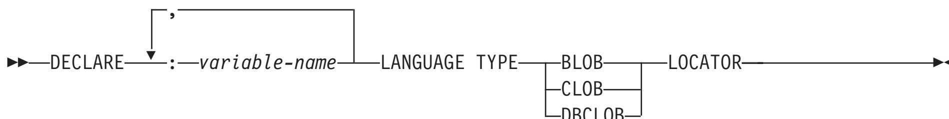
REXX SQL では、以下に示すホスト変数のストリングの内容により、LOB タイプが決定されます。

ホスト変数のストリングの内容	使用される LOB タイプ
:hv1=’通常の引用符付きストリングが 32K を超えている’	CLOB
:hv2=″’組み込み区切り引用符 ″,″ 付きのストリングが 32K を超えている’″	CLOB
:hv3=″G’G で開始する組み込み区切り単一引用符 ″,″ 付きの DBCS が 32K を超えている’″	DBCLOB
:hv4=″BIN’BIN で開始する組み込み区切り単一引用符 ″,″ 付きのストリングが任意の長さである’″	BLOB

REXX における LOB ロケーター宣言

747ページのREXX における LOB ロケーター・ホスト変数の構文は、REXX における LOB ロケーター・ホスト変数の宣言の構文を示します。

REXX における LOB ロケーター・ホスト変数の構文



アプリケーション内で LOB ロケーター・ホスト変数を宣言しなければなりません。これらの宣言が出てくると、REXX/SQL はプログラムのそれ以降の部分で、宣言されたホスト変数をロケーターとして取り扱います。ロケーターの値は、内部形式で REXX 変数に保管されます。

以下に例を示します。

```
CALL SQLEXEC 'DECLARE :hv1, :hv2 LANGUAGE TYPE CLOB LOCATOR'
```

エンジンから戻された LOB により表現されるデータは、以下に示す形式の FREE LOCATOR ステートメントを使用して、 REXX/SQL 内で解放することができます。

FREE LOCATOR ステートメントの構文

以下に例を示します。

```
CALL SQLEXEC 'FREE LOCATOR :hv1, :hv2'
```

REXX における LOB ファイル参照宣言

アプリケーション内で LOB ファイル参照ホスト変数を宣言しなければなりません。これらの宣言が出てくると、REXX/SQL はプログラムのそれ以降の部分で、宣言されたホスト変数を LOB ファイル参照として取り扱います。

REXX における LOB ファイル参照変数の構文は、REXX における LOB ファイル参照ホスト変数の宣言の構文を示します。

REXX ファイル参照宣言

以下に例を示します。

```
CALL SQLEXEC 'DECLARE :hv3, :hv4 LANGUAGE TYPE CLOB FILE'
```

REXX におけるファイル参照変数には、3 つのフィールドが含まれます。上記の例では、以下のものがその 3 つのフィールドに相当します。

hv3.FILE_OPTIONS.
: アプリケーションによりセットされ、ファイルの使用法を指示します。

hv3.DATA_LENGTH.
: DB2 によりセットされ、ファイルのサイズを指示します。

hv3.NAME.
: アプリケーションにより、LOB ファイルの名前に設定されます。

FILE_OPTIONS の場合は、アプリケーションが以下のキーワードを設定します。

キーワード (整数値)
: **意味**

READ (2)
: ファイルが入力に使用されます。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。ファイル内のデータの長さは、 (アプリケーションの要求側のコードにより) ファイルのオープン時に計算されます。

CREATE (8)
: 出力において、新しい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となります。ファイルの長さ (バイト単位) は、ファイル参照変数構造の DATA_LENGTH フィールドに戻されます。

OVERWRITE (16)
: 出力において、ファイルがすでに存在する場合はそれを上書きし、そうでない場合は新しいファイルを作成します。ファイルの長さ (バイト単位) は、ファイル参照変数構造の DATA_LENGTH フィールドに戻されます。

APPEND (32)
: ファイルがすでに存在する場合はそこに出力が追加され、そうでない場合は新しいファイルが作成されます。ファイルに追加されるデータ (ファイル全体の長さではない) の長さ (バイト単位) は、ファイル参照変数構造の DATA_LENGTH フィールドに戻されます。

注: REXX では、ファイル参照ホスト変数は複合変数です。したがって、NAME、NAME_LENGTH、および FILE_OPTIONS フィールドは、宣言するだけでなく、値も設定しなければなりません。

REXX での LOB ホスト変数のクリア

OS/2 では、プログラムの終了後も効力を持つ REXX SQL LOB ロケーターおよびファイル参照ホスト変数宣言を、明示的にクリアしなければならない場合があります。これは、実行中のセッションがクローズされるまでアプリケーション・プロセスが終了しないためです。 REXX SQL LOB 宣言がクリアされないと、 LOB アプリケーションの実行後に同一セッション内で実行されている他のアプリケーションの妨げになります。

宣言をクリアする構文を示します。

```
CALL SQLEXEC "CLEAR SQL VARIABLE DECLARATIONS"
```

このステートメントは、LOB アプリケーションの終端にコーディングしなければなりません。直前のアプリケーションで宣言がクリアされていない場合があるので、宣言をクリアするための回避的な手段として、このステートメントを任意の場所にコーディングできます (たとえば、REXX SQL アプリケーションの最初)。

REXX でサポートされている SQL データ・タイプ

特定の定義済み REXX のデータ・タイプは、 DB2 の列のタイプに対応しています。表36 は、 SQLEXEC および SQLDBS が REXX 変数をどのように解釈して、その内容を DB2 のデータ・タイプに変換するかを示しています。

注: DB2 ホスト言語で、DATALINK データ・タイプをサポートするホスト変数はありません。

表 36. REXX 宣言にマップされる SQL 列タイプ

SQL 列タイプ[1]	REXX データ・タイプ	SQL 列タイプ記述
SMALLINT (500 または 501)	10 進小数点を持たない -32 768 〜 32 767 の数	16 ビットの符号付き整数
INTEGER (496 または 497)	10 進小数点を持たない -2 147 483 648 〜 2 147 483 647 の数	32 ビットの符号付き整数
REAL[2] (480 または 481)	-3.40282346 x 10^{38} 〜 3.40282346 x 10^{38} の浮動小数の数	単精度浮動小数点
DOUBLE[3] (480 または 481)	-1.79769313 x 10^{308} 〜 1.79769313 x 10^{308} の浮動小数の数	倍精度浮動小数点
DECIMAL(*p,s*) (484 または 485)	10 進小数点を持つ数	パック 10 進数
CHAR(*n*) (452 または 453)	前後に引用符 (') を持つストリング。2 つの引用符を除くと、長さが *n* になる。 先行および後続ブランク、または浮動小数の E 以外の非数値文字を持つ、長さ *n* のストリング	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)	CHAR(*n*) と等しい	長さが *n* の可変長文字ストリング。 *n* の範囲は 1 〜 4000 まで
LONG VARCHAR (456 または 457)	CHAR(*n*) と等しい	長さが *n* の可変長文字ストリング。 *n* の範囲は 1 〜 32 700 まで
CLOB(*n*) (408 または 409)	CHAR(*n*) と等しい	長さが *n* のラージ・オブジェクト可変長文字ストリング。 *n* の範囲は 1 〜 2 147 483 647 まで
CLOB ロケーター変数[4] (964 または 965)	DECLARE *:var_name* LANGUAGE TYPE CLOB LOCATOR	サーバー上の 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)	DECLARE *:var_name* LANGUAGE TYPE CLOB FILE	CLOB データを含むファイルの記述子

表 36. REXX 宣言にマップされる SQL 列タイプ (続き)

SQL 列タイプ[1]	REXX データ・タイプ	SQL 列タイプ記述
BLOB(*n*) (404 または 405)	前後にアポストロフィーを持つストリングで、BIN が先行する。先行の BIN と 2 つのアポストロフィーを除くと、*n* 文字となる	長さが *n* のラージ・オブジェクト可変長 2 進ストリング。 *n* の範囲は 1 〜 2 147 483 647 まで
BLOB ロケーター変数[4] (960 または 961)	DECLARE *:var_name* LANGUAGE TYPE BLOB LOCATOR	サーバー上の 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)	DECLARE *:var_name* LANGUAGE TYPE BLOB FILE	BLOB データを含むファイルの記述子
DATE (384 または 385)	CHAR(*10*) と等しい	10 バイトの文字ストリング
TIME (388 または 389)	CHAR(*8*) と等しい	8 バイトの文字ストリング
TIMESTAMP (392 または 393)	CHAR(*26*) と等しい	26 バイトの文字ストリング
注: 以下のデータ・タイプは、DBCS 環境でのみ使用可能です。		
GRAPHIC(*n*) (468 または 469)	前後にアポストロフィーを持つストリングで、G または N が先行する。先行の文字と 2 つのアポストロフィーを除くと、 *n* 個の DBCS 文字となる	長さが *n* の固定長グラフィック・ストリング。 *n* の範囲は 1 〜 127 まで
VARGRAPHIC(*n*) (464 または 465)	GRAPHIC(*n*) と等しい	長さが *n* の可変長グラフィック・ストリング。 *n* の範囲は 1 〜 2000 まで
LONG VARGRAPHIC (472 または 473)	GRAPHIC(*n*) と等しい	長さが *n* の長可変長グラフィック・ストリング。 *n* の範囲は、1 〜 16 350 まで
DBCLOB(*n*) (412 または 413)	GRAPHIC(*n*) と等しい	長さが *n* のラージ・オブジェクト可変長グラフィック・ストリング。 *n* の範囲は 1 〜 1 073 741 823 まで
DBCLOB ロケーター変数[4] (968 または 969)	DECLARE *:var_name* LANGUAGE TYPE DBCLOB LOCATOR	サーバー上の 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)	DECLARE *:var_name* LANGUAGE TYPE DBCLOB FILE	DBCLOB データを含むファイルの記述子

注:

1. **列タイプ**の最初の番号は、標識変数が提供されないことを示し、2 番目の番号は標識変数が提供されることを示す。標識変数は、ヌル値を示したり、切り捨てられたストリングの長さを保留するのに必要です。
2. FLOAT(*n*) ここで $0 < n < 25$ の場合、REAL と同義。 SQLDA での REAL と DOUBLE の違いは長さの値です (4 または 8)。
3. 次の SQL タイプは、DOUBLE と同義語。
   - FLOAT
   - FLOAT(*n*)。ここで、*n* の取る範囲は $24 < n < 54$。
   - DOUBLE PRECISION
4. これは列タイプではなく、ホスト変数タイプである。

REXX でのカーソルの使用

REXX においてカーソルを宣言する際、カーソルは照会と関連付けられます。照会は、PREPARE ステートメントで割り当てられたステートメント名と関連付けられます。参

照したホスト変数はパラメーター・マーカーで表されます。以下の例は、動的 SELECT ステートメントに関連する DECLARE ステートメントを示しています。

```
prep_string = "SELECT TABNAME FROM SYSCAT.TABLES WHERE TABSCHEMA = ?"
CALL SQLEXEC 'PREPARE S1 FROM :prep_string';
CALL SQLEXEC 'DECLARE C1 CURSOR FOR S1';
CALL SQLEXEC 'OPEN C1 USING :schema_name';
```

REXX の実行要件

REXX アプリケーションではプリコンパイル、およびリンクは行われません。

OS/2 の場合、アプリケーション・ファイルの拡張子は .CMD になっている必要があります。この拡張子を付けると、アプリケーションをオペレーティング・システムのコマンド・プロンプトから直接実行できます。

Windows 32 ビット・オペレーティング・システムの場合、アプリケーション・ファイルには、任意の名前を付けることができます。ファイルを作成後、次のように REXX インタープリターを起動して、アプリケーションをオペレーティング・システムのコマンド・プロンプトから実行できます。

```
REXX file_name
```

AIX では、アプリケーション・ファイルに任意の拡張子を付けることができます。次の 2 つの方法のいずれかを使用して、アプリケーションを実行できます。

1. シェル・コマンド・プロンプトで、`rexx name` と入力する。 `name` は REXX プログラムの名前です。
2. REXX プログラムの先頭行に ″magic number″ (#!) が含まれ、 REXX/6000 インタープリターのあるディレクトリーが識別されている場合は、 REXX プログラムの名前をシェル・コマンド・プロンプトに入力することにより、これを実行できます。たとえば、REXX/6000 インタープリター・ファイルが /usr/bin ディレクトリーにある場合は、 REXX プログラムの先頭行を以下のようにします。

   ```
   #! /usr/bin/rexx
   ```

   シェル・コマンド・プロンプトに次のコマンドを入力すると、プログラムが実行可能になります。

   ```
   chmod +x name
   ```

   シェル・コマンド・プロンプトにファイル名を入力して、 REXX プログラムを実行してください。

注: AIX では、LIBPATH 環境変数をセットして、 REXX SQL ライブラリー `db2rexx` があるディレクトリーを含めます。以下に例を示します。

```
export LIBPATH=/lib:/usr/lib:/usr/lpp/db2_07_01/lib
```

REXX のバインド・ファイル

REXX アプリケーションをサポートするために、5 つのバインド・ファイルが提供されています。それらのファイルの名前は、DB2UBIND.LST ファイルの中に組み込まれています。各バインド・ファイルは、それぞれ別々の分離レベルを使用してプリコンパイルされます。したがって、5 つの異なるパッケージがデータベース内に保管されます。

以下に、5 つのバインド・ファイルを示します。

DB2ARXCS.BND
カーソル固定の分離レベルをサポートします。

DB2ARXRR.BND
反復可能読み取りの分離レベルをサポートします。

DB2ARXUR.BND
非コミット読み取りの分離レベルをサポートします。

DB2ARXRS.BND
読み取り固定の分離レベルをサポートします。

DB2ARXNC.BND
コミットなしの分離レベルをサポートします。この分離レベルは、AS/400 データベース・サーバーを作動させる際に使用されます。他のデータベースでは、非コミット読み取りの分離レベルと同様に動作します。

注: これらのファイルを、データベースに明示的にバインドしなければならない場合もあります。

SQLEXEC ルーチンを使用する場合は、カーソル固定により作成されたパッケージがデフォルトのパッケージとして使用されます。他の分離レベルを使用する必要がある場合は、データベースに接続する前に SQLDBS CHANGE SQL ISOLATION LEVEL API を使用して分離レベルを変更できます。分離レベルを変更すると、その後に続く SQLEXEC ルーチンに対する呼び出しが、新たに指定した分離レベルと関連付けられます。

OS/2 REXX アプリケーションは、セッション内の他の REXX プログラムの設定が変更されていないことを確認しない限り、デフォルトの分離レベルが有効であると見なしません。 REXX アプリケーションは、データベースに接続する前に分離レベルを明示的に設定しなければなりません。

REXX の API 構文

DB2 API を呼び出すには、次の構文により SQLDBS ルーチンを使用します。

```
CALL SQLDBS 'command string'
```

DB2 API の処理方法については、 *管理 API 解説書* の DB2 API の章に詳しい説明があります。

SQLDBS ルーチンを使用しても、使用する DB2 API を呼び出せない (したがって、*管理 API 解説書* にはリストされていない) 場合、 REXX アプリケーション内から DB2 コマンド行プロセッサー (CLP) を呼び出して、その API を呼び出してください。 DB2 CLP は、標準出力装置または特定のファイルのどちらかに出力するように命令します。その理由で、REXX アプリケーションは、呼び出された DB2 API からその出力に直接アクセスできません。また、呼び出された API が正常に処理されたかどうかを容易には判断することもできません。しかし、SQLDB2 API は DB2 CLP へのインターフェースを提供しています。そのインターフェースは、各呼び出し後に複合の REXX 変数である SQLCA を設定して、呼び出された API ごとに処理が正常だったか失敗だったかを、直接 REXX アプリケーションにフィードバックできます。

SQLDB2 ルーチンを使用して、次の構文により DB2 API を呼び出すことができます。

```
CALL SQLDB2 'command string'
```

`'command string'` は、コマンド行プロセッサー (CLP) で処理可能なストリングです。 CLP で処理可能なストリングの構文は、 *コマンド解説書* を参照してください。

SQLDB2 を使用して DB2 API を呼び出すことは、次の場合の他は CLP を直接呼び出すのと同じです。

- 実行可能 CLP の呼び出しが SQLDB2 の呼び出しで置換された場合 (他のすべての CLP オプションとパラメーターは、同じように指定されている)。
- SQLDB2 の呼び出し後に REXX の複合変数 SQLCA が設定されたが、実行可能 CLP の呼び出し後には設定されなかった場合。
- SQLDB2 を呼び出した時、CLP のディスプレイ出力のデフォルトはオフに設定されていますが、実行可能 CLP を呼び出す場合、ディスプレイ出力はオンに設定されています。 CLP のディスプレイ出力をオンに切り替えるには、 SQLDB2 に `+o` または `-o-` オプションを渡します。

SQLDB2 の呼び出し後、REXX 変数だけが SQLCA に設定されるので、 DB2 API を呼び出すには、このルーチンだけを使用してください。 DB2 API は SQLDBS インターフェースですが、SQLCA 以外のデータを戻しませんし、現在のところ組み込まれてはいません。したがって SQLDB2 では、以下の DB2 API だけがサポートされています。

データベースの活動化 (Activate Database)
ノードの追加 (Add Node)
DB2 バージョン 1 用バインド (Bind for DB2 Version 1)(1) (2)
DB2 バージョン 2 または 5 用バインド (Bind for DB2 Version 2 or 5)(1)
ノードでのデータベース作成 (Create Database at Node)
ノードでのデータベースのドロップ (Drop Database at Node)
ノードのドロップの検査 (Drop Node Verify)

データベースの非活動化 (Deactivate Database)
登録抹消 (Deregister)
ロード (Load)[3]
照会のロード (Load Query)
プログラムのプリコンパイル (Precompile Program)[1]
パッケージの再バインド (Rebind Package)[1]
ノード・グループの再分散 (Redistribute Nodegroup)
登録 (Register)
データベース・マネージャーの開始 (Start Database Manager)
データベース・マネージャーの停止 (Stop Database Manager)

SQLDB2 がサポートする DB2 API に関する注:

1. これらのコマンドは、SQLDB2 インターフェースを介して CONNECT ステートメントを必要とします。 SQLDB2 インターフェースを使用した接続では、SQLEXEC インターフェースに接続できません。また、SQLEXEC インターフェースを使用した接続では、SQLDB2 インターフェースに接続できません。
2. SQLDB2 インターフェースを介して OS/2 でサポートされます。
3. Load API 用の任意指定の出力パラメーター `pLoadInfoOut` は、 REXX のアプリケーションに戻りません。 Load API とそのパラメーターに関する詳細は、 *管理 API 解説書* を参照してください。

注: SQLDB2 ルーチンは、上にリストされた DB2 API のためだけに使用されるように意図されていますが、 SQLDBS ルーチンを通じてサポートされていない、他の DB2 API のためにも使用することができます。あるいは、REXX アプリケーション内から CLP を介してアクセスすることが可能です。

REXX のストアード・プロシージャー

REXX SQL アプリケーションは、SQL CALL ステートメントを使用して、データベース・サーバーでストアード・プロシージャーを呼び出せます。ストアード・プロシージャーは、AIX システム上の REXX を除き、そのサーバー上でサポートされる任意の言語で作成することができます。 (クライアント・アプリケーションは AIX 上の REXX で作成できますが、他の言語と同様、AIX 上の REXX で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。)

REXX におけるストアード・プロシージャーの呼び出し

CALL ステートメントを使用すると、クライアント・アプリケーションがサーバー・ストアード・プロシージャーとの間でデータをやりとりできるようになります。入出力データ用のインターフェースは、ホスト変数のリストになっています (詳しくは、*SQL 解説書* を参照)。 REXX は一般に、ホスト変数のタイプとサイズをその内容に基づいて判別するため、 CALL に渡される出力専用変数は予期出力と同じタイプとサイズを持つダミーの データを用いて初期化されます。

データは、CALL ステートメントの USING DESCRIPTOR 構文を使用し、 SQLDA REXX 変数を介して、ストアード・プロシージャーに渡されます。 表37 に SQLDA が設定される方法を示します。表の中の ':value' は、アプリケーションに必要な値を含む REXX ホスト変数のステムです。 DESCRIPTOR の場合、'n' は SQLDA の特定の *sqlvar* エレメントを示す数値です。右側の数値は、表37 の下の注意事項に対応しています。

表 37. CALL ステートメントを使用したストアード・プロシージャーの、クライアント側の REXX SQLDA

USING DESCRIPTOR	:value.SQLD	1	
	:value.n.SQLTYPE	1	
	:value.n.SQLLEN	1	
	:value.n.SQLDATA	1	2
	:value.n.SQLDIND	1	2

注:

1. ストアード・プロシージャーを呼び出す前に、クライアント・アプリケーションは適切なデータを使用して REXX 変数を初期化しなければならない。

   SQL CALL ステートメントが実行されると、データベース・マネージャーはストレージを割り振り、 REXX 変数プールから REXX 変数の値を取り出します。 CALL ステートメントで使用される SQLDA では、データベース・マネージャーは、SQLTYPE および SQLLEN 値に基づいて、ストレージを SQLDATA および SQLIND フィールドに割り振ります。

   REXX ストアード・プロシージャーの場合 (つまり、呼び出されるプロシージャーが OS/2 REXX で作成されている)、クライアントにより CALL ステートメントまたは DARI API のいずれかのタイプから渡されるデータは、以下の事前定義名を使用してデータベース・サーバーの REXX 変数プールに入れられます。

   SQLRIDA
   REXX 入力 SQLDA 変数の事前定義名

   SQLRODA
   REXX 出力 SQLDA 変数の事前定義名

2. ストアード・プロシージャーが終了すると、データベース・マネージャーはストアード・プロシージャーからも変数の値を取り出す。それらの値はクライアント・アプリケーションに戻され、クライアントの REXX 変数プールの中に置かれます。

REXX のクライアントに関する考慮事項

CALL ステートメントでホスト変数を使用するときは、サーバー・プロシージャーからホスト変数に戻されるデータすべてと互換性のあるタイプの値に、各ホスト変数を初期化してください。この初期化は、対応する標識が負であっても実行しなければなりません。

記述子を使用するときは、SQLDATA を初期化して、サーバー・プロシージャーから戻されるデータすべてと互換性のあるタイプのデータを含める必要があります。この初期化は、SQLIND フィールドに負の値が入っていても実行しなければなりません。

REXX のサーバーに関する考慮事項

事前定義された出力 sqlda SQLRODA の SQLDATA フィールドおよび SQLIND (ヌル値可タイプの場合) のすべてが初期化されていることを確認してください。たとえば、SQLRODA.SQLD が 2 の場合、(たとえ、対応する標識が負で、データがクライアントに戻されなくても) 次のフィールドには同じデータが入っているはずです。

- SQLRODA.1.SQLDATA
- SQLRODA.2.SQLDATA

SQLDA 10 進フィールドの精度と SCALE 値の検索

REXX プログラムで SQLDA 出力を初期化した場合、データベース・マネージャーから戻された SQLDA 構造から 10 進フィールドの精度と位取り値を取り出すには、`sqllen.scale` 値と `sqllen.precision` 値を使用します。以下に例を示します。

```
.
.
.
/* INITIALIZE ONE ELEMENT OF OUTPUT SQLDA */
io_sqlda.sqld = 1
io_sqlda.1.sqltype = 485           /* DECIMAL DATA TYPE */
io_sqlda.1.sqllen.scale  = 2       /* DIGITS RIGHT OF DECIMAL POINT */
io_sqlda.1.sqllen.precision  = 7   /* WIDTH OF DECIMAL  */
io_sqlda.1.sqldata = 00000.00      /* HELPS DEFINE DATA FORMAT */
io_sqlda.1.sqlind = -1             /* NO INPUT DATA */
.
.
.
```

REXX の日本語または中国語 (繁体字) EUC に関する考慮事項

REXX アプリケーションは、日本語や中国語 (繁体字) の EUC 環境ではサポートされません。

第7部 付録

付録A. サポートされる SQL ステートメント

表38:

- Linux、OS/2、UNIX、および Windows 32 ビット・オペレーティング・システムの DB2 ユニバーサル・データベースでサポートされる SQL ステートメントをすべてリストします。
- それらを動的に実行できるかどうかを示す ('X' で表す)
- それらがコマンド行プロセッサー (CLP) でサポートされるかどうかを示す ('X' で表す)
- そのステートメントが DB2 コール・レベル・インターフェース (DB2 CLI) を使って実行できるかどうかを示す ('X' または DB2 CLI 関数名で表す)
- そのステートメントを SQL プロシージャーで実行できるかどうかを示す ('X' で表す)

表38 を早見表として使用することもできます。構文など、すべてのステートメントの完全な説明については、 *SQL 解説書* を参照してください。

表 38. SQL ステートメント (DB2 ユニバーサル・データベース)

SQL ステートメント	**動的[1]**	**コマンド行プロセッサー (CLP)**	**コール・レベル・インターフェース[3] (CLI)**	**SQL プロシージャー**
ALLOCATE CURSOR				X
割り当てステートメント				X
ASSOCIATE LOCATORS				X
ALTER { BUFFERPOOL, NICKNAME,[10] NODEGROUP, SERVER,[10] TABLE, TABLESPACE, USER MAPPING,[10] TYPE, VIEW }	X	X	X	
BEGIN DECLARE SECTION[2]				
CALL		X[9]	X[4]	X
CASE ステートメント				X
CLOSE		X	`SQLCloseCursor()`, `SQLFreeStmt()`	X
COMMENT ON	X	X	X	X
COMMIT	X	X	`SQLEndTran`, `SQLTransact()`	X

表 38. SQL ステートメント (DB2 ユニバーサル・データベース) (続き)

SQL ステートメント	動的[1]	コマンド行プロセッサー (CLP)	コール・レベル・インターフェース[3] (CLI)	SQL プロシージャー
複合 SQL (組み込み)			X[4]	
複合ステートメント				X
CONNECT (タイプ 1)		X	SQLBrowseConnect(), SQLConnect(), SQLDriverConnect()	
CONNECT (タイプ 2)		X	SQLBrowseConnect(), SQLConnect(), SQLDriverConnect()	
CREATE { ALIAS, BUFFERPOOL, DISTINCT TYPE, EVENT MONITOR, FUNCTION, FUNCTION MAPPING,[10] INDEX, INDEX EXTENSION, METHOD, NICKNAME,[10] NODEGROUP, PROCEDURE, SCHEMA, TABLE, TABLESPACE, TRANSFORM TYPE MAPPING,[1] TRIGGER, USER MAPPING,[10] TYPE, VIEW, WRAPPER[10] }	X	X	X	X[11]
DECLARE CURSOR[2]		X	SQLAllocStmt()	X
DECLARE GLOBAL TEMPORARY TABLE	X	X	X	X
DELETE	X	X	X	X
DESCRIBE[8]		X	SQLColAttributes(), SQLDescribeCol(), SQLDescribParam()[6]	
DISCONNECT		X	SQLDisconnect()	
DROP	X	X	X	X[11]
END DECLARE SECTION[2]				
EXECUTE			SQLExecute()	X
EXECUTE IMMEDIATE			SQLExecDirect()	X
EXPLAIN	X	X	X	X

表 38. SQL ステートメント (DB2 ユニバーサル・データベース) (続き)

SQL ステートメント	動的[1]	コマンド行プロセッサー (CLP)	コール・レベル・インターフェース[3] (CLI)	SQL プロシージャー
FETCH		X	SQLExtendedFetch()[7], SQLFetch(), SQLFetchScroll()[7]	X
FLUSH EVENT MONITOR	X	X	X	
FOR ステートメント				X
FREE LOCATOR			X[4]	X
GET DIAGNOSTICS				X
GOTO ステートメント				X
GRANT	X	X	X	X
IF ステートメント				X
INCLUDE[2]				
INSERT	X	X	X	X
ITERATE				X
LEAVE ステートメント				X
LOCK TABLE	X	X	X	X
LOOP ステートメント				X
OPEN		X	SQLExecute(), SQLExecDirect()	X
PREPARE			SQLPrepare()	X
REFRESH TABLE	X	X	X	
RELEASE		X		X
RELEASE SAVEPOINT	X	X	X	X
RENAME TABLE	X	X	X	
RENAME TABLESPACE	X	X	X	
REPEAT ステートメント				X
RESIGNAL ステートメント				X
RETURN ステートメント				X
REVOKE	X	X	X	
ROLLBACK	X	X	SQLEndTran(), SQLTransact()	X
SAVEPOINT	X	X	X	X
SELECT ステートメント	X	X	X	X
SELECT INTO				X

表 38. SQL ステートメント (DB2 ユニバーサル・データベース) (続き)

SQL ステートメント	動的[1]	コマンド行プロセッサー (CLP)	コール・レベル・インターフェース[3] (CLI)	SQL プロシージャー
SET CONNECTION		X	SQLSetConnection()	
SET CURRENT DEFAULT TRANSFORM GROUP	X	X	X	X
SET CURRENT DEGREE	X	X	X	X
SET CURRENT EXPLAIN MODE	X	X	X, SQLSetConnectAttr()	X
SET CURRENT EXPLAIN SNAPSHOT	X	X	X, SQLSetConnectAttr()	X
SET CURRENT PACKAGESET				
SET CURRENT QUERY OPTIMIZATION	X	X	X	X
SET CURRENT REFRESH AGE	X	X	X	X
SET EVENT MONITOR STATE	X	X	X	X
SET INTEGRITY	X	X	X	
SET PASSTHRU[10]	X	X	X	X
SET PATH	X	X	X	X
SET SCHEMA	X	X	X	X
SET SERVER OPTION[10]	X	X	X	X
SET 変位-変数[5]	X	X	X	X
SIGNAL ステートメント				X
SIGNAL SQLSTATE[5]	X	X	X	
UPDATE	X	X	X	X
VALUES INTO				X
WHENEVER[2]				
WHILE ステートメント				X

表 38. SQL ステートメント (DB2 ユニバーサル・データベース) (続き)

SQL ステートメント	動的[1]	コマンド行プロセッサー (CLP)	コール・レベル・インターフェース[3] (CLI)	SQL プロシージャー

注:

1. このリストのすべてのステートメントは静的 SQL としてコーディングできますが、動的 SQL としてコーディングできるのは X マークの付いたステートメントだけです。
2. このステートメントは実行できません。
3. X は、該当するステートメントが `SQLExecDirect()` または `SQLPrepare()` と `SQLExecute()` のどちらによっても実行できるという意味です。等価の DB2 CLI 関数があると、その関数名がリストされます。
4. このステートメントは動的ではないものの、DB2 CLI によって `SQLExecDirect()` または `SQLPrepare()` と `SQLExecute()` のどちらかを呼び出すときに、このステートメントが指定されます。
5. これは CREATE TRIGGER ステートメントでのみ使用することができます。
6. SQL DESCRIBE ステートメントは、出力の記述にのみ使用できます。 DB2 CLI では、入力も記述できます (`SQLDescribeParam()` 関数を使用)。
7. SQL FETCH ステートメントは、一度に 1 列を 1 方向に取り出す場合にのみ使用できます。 DB2 CLI の `SQLExtendedFetch()` および `SQLFetchScroll()` 関数では、配列から取り出すことができます。さらに、どちらの方向にも、また結果セットのどの位置でも取り出し可能です。
8. DESCRIBE SQL ステートメントの構文は、CLP DESCRIBE コマンドとは違っています。 DESCRIBE SQL ステートメントについては、 *SQL 解説書* を参照してください。 DESCRIBE CLP コマンドについては、 *コマンド解説書* を参照してください。
9. CALL がコマンド行プロセッサーから発行された場合、以下のプロシージャーとその個別パラメーターだけがサポートされます。 (690ページの『JAR ファイルのインストール、置換、および除去』を参照)。
10. ステートメントがサポートされるのは、連合データベース・サーバーの場合だけです。
11. SQL プロシージャーが発行できるのは、索引、表、およびビュー用の CREATE および DROP ステートメントだけです。

付録B. サンプル・プログラム

この節では、DB2 に用意されているサンプル・プログラムについて説明します。すべてのサンプル・プログラムは、 `sqllib` ディレクトリーの `samples` サブディレクトリーの中に入っています。サポートされている言語ごとにサブディレクトリーが 1 つずつあります。

本書で使用されるサンプル・プログラムは、サポートされるホスト言語による組み込み SQL ステートメントおよび API 呼び出しの例を示しています。サンプル・プログラムは、短くかつ単純に作られています。プロダクション・アプリケーションは、API 呼び出しおよび SQL ステートメントからの戻りコード、特に SQLCODE または SQLSTATE を調べなければなりません。エラー条件、SQLCODE、および SQLSTATE の処理については、 122ページの『診断処理と SQLCA 構造』を参照してください。これらのプログラムを現在使用している環境でインストール、作成、および実行する方法については、 *アプリケーション構築の手引き* を参照してください。

注:

1. この節では、DB2 がサポートするすべてのプラットフォーム用のプログラム言語のサンプル・プログラムを記載しています。以下に示す例のすべてがサポートされるプログラミング言語に移植されているわけではありません。
2. DB2 サンプル・プログラムは、保証はまったくありませんが、「そのまま」使用できます。ユーザーおよび IBM 以外の方は、品質、パフォーマンス、何らかの欠陥の訂正のすべてのリスクをご承知いただきます。

サンプル・プログラムは、DB2 アプリケーション開発 (DB2 AD) クライアントに付属しています。サンプル・プログラムをテンプレートとして使用して、独自のアプリケーションを作成することができます。

サンプル・プログラムのファイル拡張子は、サポートされる各言語ごとに異なり、各言語内でも、組み込み SQL プログラムと非組み込み SQL プログラムとでは異なります。さらには、各言語内のプログラム・グループごとにも異なっています。これらのサンプル・ファイル拡張子を分類したのが、次の表です。

言語別のサンプル・ファイル拡張子
767ページの表39

プログラム・グループ別のサンプル・ファイル拡張子
767ページの表40

次の表は、サンプル・プログラムをタイプで分類しています。

© Copyright IBM Corp. 1993, 2001

組み込み SQL なしの DB2 API サンプル・プログラム
769ページの表41

DB2 API 組み込み SQL サンプル・プログラム
773ページの表42

DB2 API なしの組み込み SQL サンプル・プログラム
775ページの表43

ユーザー定義関数サンプル・プログラム
777ページの表44

DB2 CLI サンプル・プログラム
777ページの表45

Java JDBC サンプル・プログラム
779ページの表46

Java SQLJ サンプル・プログラム
780ページの表47

SQL プロシージャー・サンプル・プログラム
782ページの表48

ActiveX Data Object、Remote Data Objects、および Microsoft Transaction Server サンプル・プログラム
784ページの表49

オブジェクトのリンクと埋め込み (OLE) オートメーションのサンプル・プログラム
785ページの表50

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
786ページの表51

コマンド行プロセッサー・プロセッサー (CLP) サンプル・プログラム
787ページの表52

ログ管理ユーザー出口プログラム
787ページの表53

注:

1. 773ページの表42 には、 DB2 API と組み込み SQL ステートメントの両方を持つプログラムが入っています。すべての DB2 API サンプル・プログラムについては、769ページの表41 および 773ページの表42 の両方を参照してください。 すべての組み込み SQL サンプル・プログラム (Java SQLJ を除く) については、 773ページの表42 および 775ページの表43 の両方を参照してください。
2. UDF サンプル・プログラムの 777ページの表44 には、 DB2 CLI UDF プログラムが含まれていません。それらについては、777ページの表45 を参照してください。

表 39. 言語別のサンプル・ファイル拡張子

言語	ディレクトリー	組み込み SQL を含むプログラム	組み込み SQL を含まないプログラム
C	samples/c samples/cli (CLI プログラム)	.sqc	.c
C++	samples/cpp	.sqC (UNIX) .sqx (Windows および OS/2)	.C (UNIX) .cxx (Windows および OS/2)
COBOL	samples/cobol samples/cobol_mf	.sqb	.cbl
JAVA	samples/java	.sqlj	.java
REXX	samples/rexx	.cmd	.cmd

表 40. プログラム・グループ別のサンプル・ファイル拡張子

サンプル・グループ	ディレクトリー	ファイル拡張子
ADO, RDO, MTS	samples¥ADO¥VB (Visual Basic) samples¥ADO¥VC (Visual C++) samples¥RDO samples¥MTS	.bas .frm .vbp (Visual Basic) .cpp .dsp .dsw (Visual C++)
CLP	samples/clp	.db2
OLE	samples¥ole¥msvb (Visual Basic) samples¥ole¥msvc (Visual C++)	.bas .vbp (Visual Basic) .cpp (Visual C++)
OLE DB	samples¥oledb	.db2
SQL プロシージャー	samples/sqlproc	.db2 .c .sqc (クライアント・アプリケーション)
ユーザー出口	samples/c	.cad (OS/2) .cadsm (UNIX および Windows) .cdisk (UNIX および Windows) .ctape (UNIX)

注:

ディレクトリー区切り文字

UNIX では /。 OS/2 および Windows プラットフォームでは ¥。表の中では、Windows または OS/2 あるいはその両方でのみ使用可能なディレクトリー以外は、 UNIX の区切り文字が使用されます。

ファイル拡張子

拡張子が 1 つしか存在しない表にあるサンプルに提供されます。

組み込み SQL を含むプログラム
このプログラムは、プリコンパイルが必要です。 REXX 組み込み SQL プログラムは、プログラムの実行時に組み込み SQL ステートメントが解釈されるので例外になります。

IBM COBOL サンプル
AIX、OS/2、および Windows 32 ビット・オペレーティング・システムだけで、 `cobol` サブディレクトリーに提供されます。

Micro Focus Cobol サンプル
AIX、HP-UX、OS/2、Solaris 実行環境、および Windows 32 ビット・オペレーティング・システムだけで、 `cobol_mf` サブディレクトリーに提供されます。

Java サンプル
Java UDF に加えて、 JDBC (Java Database Connectivity) アプレット、アプリケーション、およびストアード・プロシージャー、Java Embedded SQL (SQLJ) アプレット、アプリケーション、およびストアード・プロシージャー。 Java サンプルは、サポートされるすべての DB2 プラットフォーム上で使用可能です。

REXX サンプル
AIX、OS/2、および Windows NT オペレーティング・システムだけで提供されます。

CLP サンプル
SQL ステートメントを実行するコマンド行プロセッサーのスクリプトです。

OLE サンプル
Microsoft Visual Basic および Microsoft Visual C++ のオブジェクトのリンクと埋め込み (OLE) のためのサンプルで、 Windows 32 ビット・オペレーティング・システム上でのみ提供されます。

ADO、RDO、および MTS サンプル
Microsoft Visual Basic および Microsoft Visual C++ の ActiveX Data Object サンプル、および Microsoft Visual Basic の Remote Data Objects と Microsoft Transaction Server サンプルで、 Windows 32 ビット・オペレーティング・システム上でのみ提供されます。

ユーザー出口サンプル
データベース・ログ・ファイルを保存し検索するのに使用する、ログ管理ユーザー出口プログラムです。ファイルは、`.c` 拡張子を付けて名前変更し、 C 言語プログラムとしてコンパイルしなければなりません。

サンプル・プログラムは、DB2 がインストールされているディレクトリーの `samples` サブディレクトリーに入っています。サポートされている言語ごとにサブディレクトリ

ーが 1 つずつあります。以下の例では、サポートされている各プラットフォーム用に C または C++ で作成されたサンプルがある場所を探す方法を示しています。

- **UNIX プラットフォームの場合**

  組み込み SQL および DB2 API プログラムの C ソース・コードは、データベース・インスタンス・ディレクトリーの下にある sqllib/samples/c にあります。 DB2 CLI プログラムの C ソース・コードは、 sqllib/samples/cli にあります。サンプル表にあるプログラムの追加情報については、 DB2 インスタンスの下の該当する samples サブディレクトリーの README ファイルを参照してください。 README ファイルには、本書でリストされていない追加のサンプルが含まれることがあります。

- **OS/2 および Windows 32 ビット・オペレーティング・システムの場合**

  組み込み SQL と DB2 API プログラムの C ソース・コードは、 DB2 インストール・ディレクトリーの下にある %DB2PATH%¥samples¥c にあります。 DB2 CLI プログラムの C ソース・コードは、 %DB2PATH%¥samples¥cli にあります。 変数 %DB2PATH% で、 DB2 のインストール先を判別できます。 DB2 がインストールされているドライブによっては、 %DB2PATH% は *drive*:¥sqllib を指します。サンプル表にあるプログラムの追加情報については、該当する %DB2PATH%¥samples サブディレクトリーの README ファイルを参照してください。 README ファイルには、本書でリストされていない追加のサンプルが含まれることがあります。

ご使用のプラットフォームについて 767ページの表39 で言及されていなかった場合は、ご使用の環境に固有の情報を、*アプリケーション構築の手引き* で参照してください。

サンプル・プログラム・ディレクトリーは、たいていのプラットフォームでは一般に読み取り専用です。サンプル・プログラムは、変更または作成する前に、ユーザーの作業ディレクトリーにコピーしてください。

組み込み SQL なしの DB2 API サンプル

表 41. 組み込み SQL なしの DB2 API サンプル・プログラム

サンプル・プログラム	組み込み API
backrest	• sqlbftcq - 表スペース・コンテナー照会の取り出し • sqlbstsc - 表スペース・コンテナーの設定 • sqlfudb - データベース構成の更新 • sqlubkp - データベースのバックアップ • sqluroll - データベースのロールフォワード • sqlurst - データベースの復元
checkerr	• sqlaintp - エラー・メッセージの入手 • sqlogstt - SQLSTATE メッセージの入手

表 41. 組み込み SQL なしの DB2 API サンプル・プログラム (続き)

サンプル・プログラム	組み込み API
cli_info	• sqleqryi - クライアント情報の照会 • sqleseti - クライアント情報の設定
client	• sqleqryc - クライアントの照会 • sqlesetc - クライアントの設定
d_dbconf	• sqleatin - 接続 • sqledtin - 切り離し • sqlfddb - データベース構成デフォルト値の入手
d_dbmcon	• sqleatin - 接続 • sqledtin - 切り離し • sqlfdsys - データベース・マネージャー構成デフォルト値の入手
db_udcs	• sqleatin - 接続 • sqlecrea - データベースの作成 • sqledrpd - データベースの除去
db2mon	• sqleatin - 接続 • sqlmon - モニター・スイッチの入手 / 更新 • sqlmonss - スナップショットの入手 • sqlmonsz - sqlmonss() 出力バッファーに必要なサイズの見積もり • sqlmrset - モニターのリセット
dbcat	• sqlecadb - データベースのカタログ化 • sqledcls - データベース・ディレクトリー・スキャンのクローズ • sqledgne - 次のデータベース・ディレクトリー・エントリーの入手 • sqledosd - データベース・ディレクトリー・スキャンのオープン • sqleuncd - データベースのアンカタログ

表 41. 組み込み SQL なしの DB2 API サンプル・プログラム (続き)

サンプル・プログラム	組み込み API
dbcmt	• sqledcgd - データベースのコメントの変更 • sqledcls - データベース・ディレクトリー・スキャンのクローズ • sqledgne - 次のデータベース・ディレクトリー・エントリーの入手 • sqledosd - データベース・ディレクトリー・スキャンのオープン • sqleisig - シグナル・ハンドラーのインストール
dbconf	• sqleatin - 接続 • sqlecrea - データベースの作成 • sqledrpd - データベースの除去 • sqlfrdb - データベース構成のリセット • sqlfudb - データベース構成の更新 • sqlfxdb - データベース構成の入手
dbinst	• sqleatcp - 接続およびパスワードの変更 • sqleatin - 接続 • sqledtin - 切り離し • sqlegins - インスタンス
dbmconf	• sqleatin - 接続 • sqledtin - 切り離し • sqlfrsys - データベース・マネージャー構成のリセット • sqlfusys - データベース・マネージャー構成の更新 • sqlfxsys - データベース・マネージャー構成の入手
dbsnap	• sqleatin - 接続 • sqlmonss - スナップショットの入手
dbstart	• sqlepstart - データベース・マネージャーの開始
dbstop	• sqlefrce - アプリケーションの強制 • sqlepstp - データベース・マネージャーの停止

表 41. 組み込み SQL なしの DB2 API サンプル・プログラム (続き)

サンプル・プログラム	組み込み API
dcscat	• sqlegdad - DCS データベースのカタログ化 • sqlegdcl - DCS ディレクトリー・スキャンのクローズ • sqlegdel - DCS データベースのアンカタログ • sqlegdge - データベースの DCS ディレクトリー・エントリーの入手 • sqlegdgt - DCS ディレクトリー・エントリーの入手 • sqlegdsc - DCS ディレクトリー・スキャンのオープン
dmscont	• sqleatin - 接続 • sqlecrea - データベースの作成 • sqledrpd - データベースの除去
ebcdicdb	• sqleatin - 接続 • sqlecrea - データベースの作成 • sqledrpd - データベースの除去
migrate	• sqlemgdb - データベースの移行
monreset	• sqleatin - 接続 • sqlmrset - モニターのリセット
monsz	• sqleatin - 接続 • sqlmonss - スナップショットの入手 • sqlmonsz - sqlmonss() 出力バッファーに必要なサイズの見積もり
nodecat	• sqlectnd - ノードのカタログ化 • sqlencls - ノード・ディレクトリー・スキャンのクローズ • sqlengne - 次のノード・ディレクトリー・エントリーの入手 • sqlenops - ノード・ディレクトリー・スキャンのオープン • sqleuncn - ノードのアンカタログ
restart	• sqlerstd - データベースの再始動
setact	• sqlesact - アカウンティング・ストリングの設定
setrundg	• sqlesdeg - 実行時間の程度の設定
sws	• sqleatin - 接続 • sqlmon - モニター・スイッチの入手 / 更新

表 41. 組み込み SQL なしの DB2 API サンプル・プログラム (続き)

サンプル・プログラム	組み込み API
utilapi	• sqlaintp - エラー・メッセージの入手 • sqlogstt - SQLSTATE メッセージの入手

DB2 API 組み込み SQL サンプル

表 42. DB2 API 組み込み SQL サンプル・プログラム

サンプル・プログラム	組み込み API
asynrlog	• sqlurlog - 非同期読み取りログ
autocfg	• db2AutoConfig -- 自動構成 • db2AutoConfigMemory -- 自動構成の空きメモリー • sqlfudb -- データベース構成の更新 • sqlfusys -- データベース・マネージャー構成の更新 • sqlesetc -- クライアントの設定 • sqlaintp -- SQLCA メッセージ
dbauth	• sqluadau - 許可の入手
dbstat	• sqlureot - 表の再編成 • sqlustat - Runstats
expsamp	• sqluexpr - エクスポート • sqluimpr - インポート
impexp	• sqluexpr - エクスポート • sqluimpr - インポート
loadqry	• db2LoadQuery - 照会のロード
makeapi	• sqlabndx - バインド • sqlaprep - プログラムのプリコンパイル • sqlepstp - データベース・マネージャーの停止 • sqlepstr - データベース・マネージャーの開始
rebind	• sqlarbnd - 再バインド

表 42. DB2 API 組み込み SQL サンプル・プログラム (続き)

サンプル・プログラム	組み込み API
rechist	• sqlubkp - データベースのバックアップ • sqluhcls - リカバリー・ヒストリー・ファイル・スキャンのクローズ • sqluhgne - 次のリカバリー・ヒストリー・ファイル・エントリーの入手 • sqluhops - リカバリー・ヒストリー・ファイル・スキャンのオープン • sqluhprn - リカバリー・ヒストリー・ファイルの枝取り • sqluhupd - リカバリー・ヒストリー・ファイルの更新
tabscont	• sqlbctcq - 表スペース・コンテナー照会のクローズ • sqlbftcq - 表スペース・コンテナー照会の取り出し • sqlbotcq - 表スペース・コンテナー照会のオープン • sqlbtcq - 表スペース・コンテナー照会 • sqlefmem - 空きメモリー
tabspace	• sqlbctsq - 表スペース照会のクローズ • sqlbftpq - 表スペース照会の取り出し • sqlbgtss - 表スペース統計の入手 • sqlbmtsq - 表スペース照会 • sqlbotsq - 表スペース照会のオープン • sqlbstpq - 単一表スペース照会 • sqlefmem - 空きメモリー
tload	• sqluexpr - エクスポート • sqluload - ロード • sqluvqdp - 表の表スペースの静止

表 42. DB2 API 組み込み SQL サンプル・プログラム (続き)

サンプル・プログラム	組み込み API
tspace	• sqlbctcq - 表スペース・コンテナー照会のクローズ • sqlbctsq - 表スペース照会のクローズ • sqlbftcq - 表スペース・コンテナー照会の取り出し • sqlbftpq - 表スペース照会の取り出し • sqlbgtss - 表スペース統計の入手 • sqlbmtsq - 表スペース照会 • sqlbotcq - 表スペース・コンテナー照会のオープン • sqlbotsq - 表スペース照会のオープン • sqlbstpq - 単一表スペース照会 • sqlbstsc - 表スペース・コンテナーの設定 • sqlbtcq - 表スペース・コンテナー照会 • sqlefmem - 空きメモリー
utilemb	• sqlaintp - エラー・メッセージの入手 • sqlogstt - SQLSTATE メッセージの入手

DB2 API なしの組み込み SQL サンプル

表 43. DB2 API なしの組み込み SQL サンプル・プログラム

サンプル・プログラム名	プログラムの説明
adhoc	動的 SQL および SQLDA 構造を使用して SQL コマンドを対話式に処理する方法を示します。 SQL コマンドはユーザーによって入力され、SQL コマンドに応じた出力が返されます。詳細については、161ページの『例: ADHOC プログラム』を参照してください。
advsql	CASE、CAST、およびスカラー全選択などの拡張 SQL 式の使用例を示します。
blobfile	2 進ラージ・オブジェクト (BLOB) の操作例を、 BLOB 値をサンプル・データベースから読み取ってそれをファイル内に置くことにより示します。この内容は、外部ビューアーを使用して表示できます。
columns	動的 SQL を使用して処理されるカーソルの使用例を示します。このプログラムは、指定のスキーマ名の下にある SYSCAT.COLUMNS から結果セットをリストします。
cursor	静的 SQL を使用するカーソルの使用例を示します。 詳細については、90ページの『例: カーソル・プログラム』を参照してください。

表 43. DB2 API なしの組み込み SQL サンプル・プログラム (続き)

サンプル・プログラム名	プログラムの説明
delet	データベースから項目を削除する静的 SQL の使用例を示します。
dynamic	動的 SQL を使用するカーソルの使用例を示します。
joinsql	拡張 SQL 結合式の使用例を示します。
largevol	区分データベース環境で並列照会処理を行う例、および結果セットの組み合わせを自動化するための NFS ファイル・システムの使用例を示します。 AIX でのみ使用可能です。詳細については、577ページの『例: 大量データの抽出 (largevol.c)』を参照してください。
lobeval	LOB ロケーターの使用例を示し、実際の LOB データの評価を遅らせます。詳細については、376ページの『LOBEVAL プログラム例の作動方法』を参照してください。
lobfile	LOB ファイル・ハンドルの使用例を示します。詳細については、384ページの『LOBFILE プログラム例の作動方法』を参照してください。
lobloc	LOB ロケーターの使用例を示します。詳細については、369ページの『LOBLOC プログラム例の作動方法』を参照してください。
lobval	LOB の使用例を示します。
openftch	静的 SQL を使用した行の取り出し、更新、および削除について示します。詳細については、99ページの『OPENFTCH プログラムの動作の仕組み』を参照してください。
recursql	拡張 SQL 再帰的照会の使用例を示します。
sampudf	表エントリーを修正するための、ユーザー定義タイプ (UDT) およびユーザー定義関数 (UDF) の使用例を示します。このプログラムで宣言される UDF は、ソース UDF です。
spclient	spserver 共用ライブラリー内のストアード・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。
spcreate.db2	spserver プログラムによって作成されたストアード・プロシージャーを登録するための CREATE PROCEDURE ステートメントを含む CLP スクリプト。
spdrop.db2	spserver プログラムによって作成されたストアード・プロシージャーの登録を解除するために必要な DROP PROCEDURE ステートメントを含む CLP スクリプト。
spserver	ストアード・プロシージャーのデモを示すサーバー・プログラム。クライアント・プログラムは spclient です。
static	情報を検索する静的 SQL を示します。詳細については、69ページの『例: 静的 SQL プログラム』を参照してください。
tabsql	拡張 SQL 表式の使用例を示します。
tbdefine	表の作成方法および除去方法を示します。

表 43. DB2 API なしの組み込み SQL サンプル・プログラム (続き)

サンプル・プログラム名	プログラムの説明
thdsrver	スレッドの作成と管理用の POSIX スレッド API の使用例を示します。プログラムは、コンテキストのプールを保守します。 generate_work 関数がメインから実行され、作業スレッドが実行する動的 SQL ステートメントを作成します。コンテキストが使用可能になったとき、スレッドが作成され、指定された作業を行うためにディスパッチされます。生成された作業は、 sample データベースの STAFF 表または EMPLOYEE 表のどちらかからの項目を削除するステートメントで成っています。このプログラムは、UNIX プラットフォームでのみ使用可能です。
trigsql	拡張 SQL トリガーおよび制約の使用例を示します。
udfcli	udfsrv プログラムによって作成されるユーザー定義関数 (UDF) の呼び出しを例示し、 sample データベースの表にアクセスするためにサーバー上に保管されます。
updat	データベースを更新する静的 SQL の使用例を示します。詳細については、111ページの『例: UPDAT プログラム』を参照してください。
varinp	パラメーター・マーカーを使用した組み込み動的 SQL ステートメント呼び出しへの変数入力を示します。詳細については、169ページの『VARINP プログラムの動作の仕組み』を参照してください。

ユーザー定義関数のサンプル

表 44. ユーザー定義関数のサンプル・プログラム

サンプル・プログラム名	プログラムの説明
DB2Udf.java	整数除算、文字ラージ・オブジェクト (CLOB) の操作、および Java インスタンス変数の使用を含む複数のタスクの実例を示す Java UDF。
udfsrv.c	ユーザー定義関数 ScalarUDF でライブラリーを作成し、サンプル・データベース表にアクセスします。
UDFsrv.java	Java ユーザー定義関数 (UDF) の使用例を示します。

DB2 コール・レベル・インターフェースのサンプル

表 45. DB2 ユニバーサル・データベースのサンプル CLI プログラム

サンプル・プログラム名	プログラムの説明
共通ユーティリティー・ファイル	
utilcli.c	CLI サンプルで使用されるユーティリティー関数。

表 45. DB2 ユニバーサル・データベースのサンプル CLI プログラム (続き)

サンプル・プログラム名	プログラムの説明
utilapi.c	DB2 API を呼び出すユーティリティー関数。
アプリケーション・レベル - DB2 と CLI のアプリケーション・レベルを扱うサンプル。	
apinfo.c	アプリケーション・レベル情報の入手および設定方法。
aphndls.c	ハンドルの割り当ておよび解放方法。
apsqlca.c	SQLCA データの処理方法。
インストール・イメージ・レベル - DB2 と CLI のインストール・イメージ・レベルを扱うサンプル。	
ilinfo.c	インストール・レベル情報 (CLI ドライバーのバージョンなど) の入手および設定方法。
インスタンス・レベル - DB2 と CLI のインスタンス・レベルを扱うサンプル。	
ininfo.c	インスタンス・レベル情報の入手および設定方法。
データベース・レベル - DB2 内のデータベース・オブジェクトを扱うサンプル。	
dbconn.c	データベースからの接続および切断方法。
dbinfo.c	データベース・レベルの情報の入手および設定方法。
dbmconn.c	複数のデータベースからの接続および切断方法 (DB2 API を使用して 2 番目のデータベースを作成および除去する)。
dbmuse.c	複数のデータベースとのトランザクションの実行方法 (DB2 API を使用して 2 番目のデータベースを作成および除去する)。
dbnative.c	ODBC エスケープ文節を含むステートメントを、データ・ソース特有の形式に変換する方法。
dbuse.c	データベース・オブジェクトの処理方法。
dbusemx.sqc	組み込み SQL とともに単一データベースを使用する方法。
表レベル - DB2 内の表オブジェクトを扱うサンプル。	
tbconstr.c	表の制約の処理方法。
tbconstr.c	表の作成、更新、および除去方法。
tbinfo.c	表レベルの情報の入手および設定方法。
tbmod.c	表内の情報の修正方法。
tbread.c	表内の情報の読み取り方法。
データ・タイプ・レベル - データ・タイプを扱うサンプル。	
dtinfo.c	データ・タイプに関する情報の入手方法。
dtlob.c	LOB データの読み取りおよび書き込み方法。
dtudt.c	ユーザー定義特殊タイプの作成、使用、および除去方法。
UDF レベル - ユーザー定義関数を示すサンプル。	

表 45. DB2 ユニバーサル・データベースのサンプル CLI プログラム (続き)

サンプル・プログラム名	プログラムの説明
udfcli.c	udfsrv.c 内のユーザー定義関数を呼び出すクライアント・アプリケーション。
udfsrv.c	udfcli.c サンプルによって呼び出されるユーザー定義関数 ScalarUDF。
ストアード・プロシージャー・レベル - CLI 内のストアード・プロシージャーを示すサンプル。	
spcreate.db2	CREATE PROCEDURE ステートメントを発行するための CLP スクリプト。
spdrop.db2	カタログからストアード・プロシージャーを除去するための CLP スクリプト。
spclient.c	spserver.c 内で宣言されるサーバー関数を呼び出すために使用されるクライアント・プログラム。
spserver.c	サーバー上で構築および実行されるストアード・プロシージャー関数。
spcall.c	ストアード・プロシージャーを呼び出すためのプログラム。

注: samples/cli ディレクトリー内には、次のファイルもあります。
- README - すべてのサンプル・ファイルのリスト
- makefile - すべてのファイルの makefile
- アプリケーションおよびストアード・プロシージャー用のビルド・ファイル

Java サンプル

表 46. JDBC (Java Database Connectivity) サンプル・プログラム

サンプル・プログラム名	プログラムの説明
DB2Appl.java	呼び出しユーザー特権を使用してサンプル・データベースを照会する、JDBC アプリケーション。
DB2Applt.java	JDBC アプレット・ドライバーを使用してデータベースを照会する JDBC アプレット。 DB2Applt.html で指定されるユーザー名、パスワード、サーバー、およびポート番号パラメーターを使用します。
DB2Applt.html	DB2Applt アプレット・サンプル・プログラムを組み込む HTML ファイル。このファイルは、サーバーおよびユーザー情報でカスタマイズする必要があります。
DB2UdCli.java	Java ユーザー定義関数、 DB2Udf を呼び出す Java クライアント・アプリケーション。
Dynamic.java	動的 SQL を使うカーソルの使用例を示します。
MRSPcli.java	これは、サーバー・プログラム MRSPsrv を呼び出すクライアント・プログラムです。このプログラムは、Java ストアード・プロシージャーから戻される複数の結果セットを示します。

表 46. JDBC (Java Database Connectivity) サンプル・プログラム (続き)

サンプル・プログラム名	プログラムの説明
MRSPsrv.java	これは、クライアント・プログラム MRSPcli が呼び出すサーバー・プログラムです。このプログラムは、Java ストアード・プロシージャーから戻される複数の結果セットを示します。
Outcli.java	SQLJ ストアード・プロシージャー、 Outsrv を呼び出す Java クライアント・アプリケーション。
PluginEx.java	新しいメニュー項目およびツールバー・ボタンを、 DB2 Web コントロール・センターに追加する Java プログラム。
Spclient.java	Spserver ストアード・プロシージャー・クラス内の PARAMETER STYLE JAVA ストアード・プロシージャーを呼び出す JDBC クライアント・アプリケーション。
Spcreate.db2	ストアード・プロシージャーとして Spserver クラス内に含まれているメソッドを登録するための CREATE PROCEDURE ステートメントを含む CLP スクリプト。
Spdrop.db2	Spserver クラス内に含まれているストアード・プロシージャーの登録を解除するために必要な DROP PROCEDURE ステートメントを含む CLP スクリプト。
Spserver.java	PARAMETER STYLE JAVA ストアード・プロシージャーのデモを示す JDBC プログラム。クライアント・プログラムは Spclient.java です。
UDFcli.java	Java ユーザー定義関数ライブラリー UDFsrv にある関数を呼び出す、 JDBC クライアント・アプリケーション。
UseThrds.java	SQL ステートメントを非同期で実行するためにスレッドを使用する方法を示します (CLI サンプル async.c の JDBC バージョン)。
V5SpCli.java	DB2GENERAL ストアード・プロシージャー、V5Stp.java を呼び出す Java クライアント・アプリケーション。
V5Stp.java	サーバー上の EMPLOYEE 表を更新し、クライアントに新しい給与および給与計算情報を戻す DB2GENERAL ストアード・プロシージャーのデモを示します。クライアント・プログラムは V5SpCli.java です。
Varinp.java	パラメーター・マーカーを使用した組み込み動的 SQL ステートメント呼び出しへの変数入力を示します。

表 47. Java Embedded SQL (SQLJ) サンプル・プログラム

サンプル・プログラム名	プログラムの説明
App.sqlj	サンプル・データベースの EMPLOYEE 表からデータを検索し更新するために、静的 SQL を使用します。
Applt.sqlj	JDBC アプレット・ドライバーを使用してデータベースを照会するアプレット。 Applt.html で指定されるユーザー名、パスワード、サーバー、およびポート番号パラメーターを使用します。

表 47. *Java Embedded SQL (SQLJ)* サンプル・プログラム *(*続き*)*

サンプル・プログラム名	プログラムの説明
Applt.html	Applt アプレット・サンプル・プログラムを組み込む HTMLファイル。このファイルは、サーバーおよびユーザー情報でカスタマイズする必要があります。
Cursor.sqlj	静的 SQL を使用するイテレーターを示します。
OpF_Curs.sqlj	Openftch プログラム用のクラス・ファイル。
Openftch.sqlj	静的 SQL を使用した行の取り出し、更新、および削除について示します。
Outsrv.sqlj	SQLDA 構造を使用するストアード・プロシージャーを示します。このプログラムは、sample データベースの STAFF 表にある従業員の給与の中央値を SQLDA に記入します。データベースの処理 (中央値を求める) 後、ストアード・プロシージャーは、値を記入した SQLDA と SQLCA 状況とを JDBC クライアント・アプリケーション、 Outcli に戻します。
Stclient.sqlj	SQLJ ストアード・プロシージャー・プログラム Stserver によって作成された PARAMETER STYLE JAVA ストアード・プロシージャーを呼び出す SQLJ クライアント・アプリケーション。
Stcreate.db2	ストアード・プロシージャーとして Stserver クラス内に含まれているメソッドを登録するための CREATE PROCEDURE ステートメントを含む CLP スクリプト。
Stdrop.db2	Stserver クラス内に含まれているストアード・プロシージャーの登録を解除するために必要な DROP PROCEDURE ステートメントを含む CLP スクリプト。
Stserver.sqlj	PARAMETER STYLE JAVA ストアード・プロシージャーのデモを示す SQLJ プログラム。クライアント・プログラムは Stclient.sqlj です。
Static.sqlj	静的 SQL を使用して情報を検索します。
Stp.sqlj	サーバー上の EMPLOYEE 表を更新し、 JDBC クライアント・プログラム StpCli に新しい給料および給与計算情報を返すストアード・プロシージャー。
UDFclie.sqlj	Java ユーザー定義関数ライブラリー UDFsrv から関数を呼び出す、クライアント・アプリケーション。
Updat.sqlj	静的 SQL を使用してデータベースを更新します。

SQL プロシージャーのサンプル

表 48. SQL プロシージャーのサンプル・プログラム

サンプル・プログラム名	プログラムの説明
basecase.db2	UPDATE_SALARY プロシージャーは、″sample″ データベースの ″staff″ 表内の ″empno″ の IN パラメーターによって識別される従業員の給料を上昇させます。このプロシージャーは、″rating″ の IN パラメーターを使用する CASE ステートメントによって上昇率を判別します。
basecase.sqc	UPDATE_SALARY プロシージャーを呼び出します。
baseif.db2	UPDATE_SALARY_IF プロシージャーは、″sample″ データベースの ″staff″ 表内の ″empno″ の IN パラメーターによって識別される従業員の給料を上昇させます。 このプロシージャーは、″rating″ の IN パラメーターを使用する IF ステートメントによって上昇率を判別します。
baseif.sqc	UPDATE_SALARY_IF プロシージャーを呼び出します。
dynamic.db2	CREATE_DEPT_TABLE プロシージャーは、動的 DDL を使用して新しい表を作成します。表の名前は、このプロシージャーの IN パラメーターの値に基づいています。
dynamic.sqc	CREATE_DEPT_TABLE プロシージャーを呼び出します。
iterate.db2	ITERATOR プロシージャーは、FETCH ループを使用して、 ″department″ 表からデータを検索します。 ″deptno″ 列の値が ’D11’ でない場合、修正されたデータが ″department″ 表の中に挿入されます。 ″deptno″ 列が ’D11’ である場合、 ITERATE ステートメントは、LOOP ステートメントの冒頭に制御の流れを渡します。
iterate.sqc	ITERATOR プロシージャーを呼び出します。
leave.db2	LEAVE_LOOP プロシージャーは、 ″not_found″ 条件ハンドラーが LEAVE ステートメントを呼び出す前に LOOP ステートメント内で実行される FETCH 操作の数をカウントします。 LEAVE ステートメントは、制御の流れがループから出て、ストアード・プロシージャーを完了するようにします。
leave.sqc	LEAVE_LOOP プロシージャーを呼び出します。
loop.db2	LOOP_UNTIL_SPACE プロシージャーは、カーソルが ″midinit″ 列のスペース (’ ’) 値を持つ行を検索するまで、 LOOP ステートメント内で実行される FETCH 操作の数をカウントします。 LOOP ステートメントは、制御の流れがループから出て、ストアード・プロシージャーを完了するようにします。
loop.sqc	LOOP_UNTIL_SPACE プロシージャーを呼び出します。
nestcase.db2	BUMP_SALARY プロシージャーは、ネストされた CASE ステートメントを使用して、 ″sample″ データベースの ″staff″ 表から部署の IN パラメーターによって識別される部署内の従業員の給料を上昇させます。
nestcase.sqc	BUMP_SALARY プロシージャーを呼び出します。

表 48. SQL プロシージャーのサンプル・プログラム (続き)

サンプル・プログラム名	プログラムの説明
nestif.db2	BUMP_SALARY_IF プロシージャーは、ネストされた IF ステートメントを使用して、 ″sample″ データベースの ″staff″ 表から部署の IN パラメーターによって識別される部署内の従業員の給料を上昇させます。
nestif.sqc	BUMP_SALARY_IF プロシージャーを呼び出します。
repeat.db2	REPEAT_STMT プロシージャーは、カーソルが行を検索できなくなるまで、繰り返しステートメント内で実行される FETCH 操作の数をカウントします。条件ハンドラーは、制御の流れが繰り返しループから出て、ストアード・プロシージャーを完了するようにします。
repeat.sqc	REPEAT_STMT プロシージャーを呼び出します。
rsultset.c	MEDIAN_RESULT_SET プロシージャーを呼び出し、給与の中央値を表示し、SQL プロシージャーによって生成された結果セットを表示します。このクライアントは、結果セットを扱える CLI API で書き出されます。
rsultset.db2	MEDIAN_RESULT_SET プロシージャーは、 ″sample″ データベースの ″staff″ 表から ″dept″ の IN パラメーターによって識別される部署内の従業員の給与の中央値を入手します。給与の中央値は、給料の OUT パラメーターに割り当てられ、 ″rsultset″ クライアントに戻されます。次に、このプロシージャーは、 WITH RETURN カーソルをオープンし、中央値よりも高い給料の従業員の結果セットを戻します。このプロシージャーは、クライアントに結果セットを戻します。
spserver.db2	この CLP スクリプト内の SQL プロシージャーは、基本的なエラー処理、ネストされたストアード・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呼び出し側アプリケーションに対して結果セットを戻すことについてのデモを示します。 CLI サンプル・ディレクトリーで、″spcall″ アプリケーションを使用してプロシージャーを呼び出すことができます。 C および CPP サンプル・ディレクトリーで、 ″spclient″ アプリケーションを使用して、結果セットを戻さないプロシージャーを呼び出すこともできます。
whiles.db2	DEPT_MEDIAN プロシージャーは、 ″sample″ データベースの ″staff″ 表から ″dept″ の IN パラメーターによって識別される部署内の従業員の給与の中央値を入手します。給与の中央値は、給料の OUT パラメーターに割り当てられ、 ″whiles″ クライアントに戻されます。次に、whiles クライアントは、給与の中央値を印刷します。
whiles.sqc	DEPT_MEDIAN プロシージャーを呼び出します。

ADO、RDO、および MTS サンプル

表 49. ADO、RDO、および MTS サンプル

サンプル・プログラム名	プログラムの説明
Bank.vbp	カスタマーの口座でトランザクションを実行する機能を使用して、銀行の支店に関するデータを作成し、保守する RDO プログラム。このプログラムには、アプリケーションがデータを格納するために必要な表を作成する DDLが入っているので、ユーザーが指定する任意のデータベースに使用できます。
Blob.vbp	この ADO プログラムは、BLOB データの検索について示します。 sample データベースの emp_photo 表から、ピクチャーを検索し、表示します。また、このプログラムは emp_photo 表のイメージを、ローカル・ファイルからのイメージに置き換えることができます。
BLOBAccess.dsw	このサンプルは、Microsoft Visual C++ を使用した ADO/Blob 強調表示アクセスを示します。これは Visual Basic サンプル、Blob.vbp と類似しています。 BLOB サンプルには次の 2 つの main 関数があります。 1. サンプル・データベースから BLOB を読み取り、画面に表示する 2. ファイルから BLOB を読み取り、データベースに挿入する (インポート)
Connect.vbp	この ADO プログラムは、sample データベースに対して接続オブジェクトを作成し、接続を確立します。いったん完了すると、プログラムは切断され、終了します。
Commit.vbp	このアプリケーションは、 ADO の自動確定 / 手動確定機能の使用例を示します。プログラムは、従業員番号および名前を、 sample データベースの EMPLOYEE 表で照会します。ユーザーは、自動確定または手動確定モードのどちらでデータベースに接続するか、オプションで選択できます。自動確定モードでは、ユーザーがレコード上で行うすべての変更は、データベースで自動的に更新されます。手動確定モードでは、ユーザーはトランザクションを開始してから、変更を行う必要があります。トランザクションが開始してから行われた変更は、ロールバックを実行すれば取り消すことができます。トランザクションをコミットすることによって、変更を永続的に保管できます。プログラムを自動的に終了すると、変更がロールバックされます。

表 49. ADO、RDO、および MTS サンプル (続き)

サンプル・プログラム名	プログラムの説明
db2com.vbp	この Visual Basic プロジェクトは、Microsoft Transaction Server を使用した、データベースの更新例を示します。また、クライアント・プログラム db2mts.vbp が使用するサーバー DLL を作成します。 Visual Basic プロジェクトには、以下の 4 つのクラス・モジュールがあります。 • UpdateNumberColumn.cls • UpdateRow.cls • UpdateStringColumn.cls • VerifyUpdate.cls このプログラムの場合、一時表 DB2MTS が sample データベースに作成されます。
db2mts.vbp	これは、Microsoft Transaction Server を使用して、 db2com.vbp から作成されたサーバー DLL を呼び出す、クライアント・プログラム用の Visual Basic プロジェクトです。
Select-Update.vbp	この ADO プログラムは、Connect.vbp と同じ機能を実行しますが、 GUI インターフェースも提供しています。このインターフェースを使用して、ユーザーは sample データベースの ORG 表に保管されたデータを表示、更新、および削除することができます。
Sample.vbp	この Visual Basic プロジェクトは Keyset カーソルを使用し、ADO を経由して、sample データベース中のすべてのデータにグラフィカル・ユーザー・インターフェースを提供します。
VarCHAR.dsp	ADO を使用してテキスト・フィールドとして VarChar にアクセスする、 Visual C++ プログラム。グラフィカル・ユーザー・インターフェースを提供するので、ユーザーは sample データベースの ORG 表のデータを表示および更新できます。

オブジェクトのリンクと埋め込みのサンプル

表 50. オブジェクトのリンクと埋め込み (OLE) サンプル・プログラム

サンプル・プログラム名	プログラムの説明
sales	Microsoft Excel スプレッドシート上でのロールアップ照会を示します (Visual Basic で実現)。
names	Lotus Notes アドレス・ブックを照会します (Visual Basic で実現)。
inbox	OLE/ メッセージ機能を使用する Microsoft Exchange インボックス E メール・メッセージを照会します (Visual Basic で実現)。

表 50. オブジェクトのリンクと埋め込み (OLE) サンプル・プログラム (続き)

サンプル・プログラム名	プログラムの説明
invoice	Microsoft Word インボイス文書を E メール接続として送信する OLE オートメーション・ユーザー定義関数 (Visual Basic で実現)。
bcounter	インスタンス変数を使用したスクラッチパッドのデモを示す OLE オートメーション・ユーザー定義関数 (Visual Basic で実現)。
ccounter	カウンター OLE オートメーション・ユーザー定義関数 (Visual C++ で実現)。
salarysrv	sample データベースの STAFF 表の給与の中央値を計算する OLE オートメーション・ストアード・プロシージャー (Visual Basic で実現)。
salarycltvc	Visual Basic ストアード・プロシージャー salarysrv を呼び出す Visual C++ 組み込み SQL サンプル。
salarycltvb	Visual Basic ストアード・プロシージャー salarysrv を呼び出す Visual Basic DB2 CLI サンプル。
testcli	ストアード・プロシージャー tstsrv を呼び出す OLE オートメーション組み込み SQL クライアント・アプリケーション (Visual C++ で実現)。
tstsrv	クライアントとストアード・プロシージャーの間での各種の受け渡しのデモを示す OLE オートメーション・ストアード・プロシージャー (Visual C++ で実現)。

表 51. 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

サンプル・プログラム名	プログラムの説明
jet.db2	Microsoft.Jet.OLEDB.3.51 Provider
mapi.db2	INTERSOLV Connect OLE DB for MAPI
msdaora.db2	Microsoft OLE DB Provider (Oracle 用)
msdasql.db2	Microsoft OLE DB Provider (ODBC ドライバー用)
msidxs.db2	Microsoft OLE DB Index Server Provider
notes.db2	INTERSOLV Connect OLE DB (ノーツ用)
sampprov.db2	Microsoft OLE DB Sample Provider
sqloledb.db2	Microsoft OLE DB Provider (SQL サーバー用)

コマンド行プロセッサーのサンプル

表 52. コマンド行プロセッサー (CLP) サンプル・プログラム

サンプル・ファイル名	ファイルの説明
const.db2	CHECK CONSTRAINT 節がある表を作成します。
cte.db2	共通表式を示します。この拡張 SQL ステートメントの例を示す同等のサンプル・プログラムは、 tabsql です。
flt.db2	再帰的照会を示します。この拡張 SQL ステートメントの例を示す同等のサンプル・プログラムは、 recursql です。
join.db2	表の外部結合を示します。この拡張 SQL ステートメントの例を示す同等のサンプル・プログラムは、 joinsql です。
stock.db2	トリガーの使用例を示します。この拡張 SQL ステートメントの例を示す同等のサンプル・プログラムは、 trigsql です。
testdata.db2	ランダムに生成されるテスト・データを表に入れるための RAND() および TRANSLATE() などの DB2 組み込み関数を使用します。
thaisort.db2	このスクリプトは、タイ語を使用するユーザー専用です。タイ語の発音順のソートでは、主な母音と子音の事前ソートとスワップが必要になります。同じように、正確なソート順序でデータを表示するために事後ソートも必要になります。ファイルでタイ語のソートを実現するには、事前ソートと事後ソートを行う UDF 関数を作成し、表を作成します。次いで、表データをソートするためにその表に対して関数を呼び出します。このプログラムを実行するには、最初に C ソース・ファイル udf.c からユーザー定義関数プログラム udf を作成しなければなりません。

ログ管理ユーザー出口サンプル

表 53. ログ管理ユーザー出口サンプル・プログラム

サンプル・ファイル名	ファイルの説明
db2uext2.cadsm	これは、データベース・ログ・ファイルを保存し検索するために、 ADSTAR DSM (ADSM) API を使用するサンプル・ユーザー出口です。このサンプルは、タイム・スタンプと受け取ったパラメーターを含む呼び出しの監査証跡 (オプションごとに別々のファイルに保存) を提供します。また、問題判別のために、エラーのタイム・スタンプとエラー分離ストリングを含む呼び出しのエラー証跡も提供します。これらのオプションは使用不能であることがあります。このファイルは、db2uext2.c に名前変更し、C プログラムとしてコンパイルしなければなりません。 UNIX および Windows 32 ビット・オペレーティング・システムで使用可能です。 OS/2 版は、db2uexit.cad です。

表 53. ログ管理ユーザー出口サンプル・プログラム (続き)

サンプル・ファイル名	ファイルの説明
`db2uexit.cad`	これは、db2uext2.cadsm の OS/2 版です。このファイルは、`db2uexit.c` に名前変更し、 C プログラムとしてコンパイルしなければなりません。
`db2uext2.cdisk`	これは、出荷時の特定のプラットフォームに合わせた、システム・コピー・コマンドを使用するサンプル・ユーザー出口です。このプログラムは、タイム・スタンプと受け取ったパラメーターを含む呼び出しの監査証跡 (オプションごとに別々のファイルに保存) を提供します。また、問題判別のために、エラーのタイム・スタンプとエラー分離ストリングを含む呼び出しのエラー証跡も提供します。これらのオプションは使用不能であることがあります。このファイルは、`db2uext2.c` に名前変更し、C プログラムとしてコンパイルしなければなりません。 UNIX および Windows 32 ビット・オペレーティング・システムで使用可能です。
`db2uext2.ctape`	これは、出荷時の特定の UNIX プラットフォームに合わせた、システム・テープ・コマンドを使用するサンプル・ユーザー出口です。プログラムは、データベース・ログ・ファイルをアーカイブおよび検索します。システム・テープ・コマンドの制限すべてが、このユーザー出口の制限になります。このサンプルは、タイム・スタンプと受け取ったパラメーターを含む呼び出しの監査証跡 (オプションごとに別々のファイルに保存) を提供します。また、問題判別のために、エラーのタイム・スタンプとエラー分離ストリングを含む呼び出しのエラー証跡も提供します。これらのオプションは使用不能であることがあります。このファイルは、`db2uext2.c` に名前変更し、C プログラムとしてコンパイルしなければなりません。 UNIX プラットフォームでのみ使用可能です。

付録C. DB2DARI および DB2GENERAL ストアード・プロシージャーと UDF

DB2DARI ストアード・プロシージャー . . 789
クライアント・アプリケーションでの SQLDA の使用. 789
DB2DARI クライアントにおけるホスト変数の使用 790
ストアード・プロシージャーにおける SQLDA の使用. 790
データ構造の操作. 791
データ構造使用法の要約 791
入出力 SQLDA および SQLCA 構造 . . 792
DB2DARI ストアード・プロシージャーの戻り値 793
DB2GENERAL UDF およびストアード・プロシージャー 794
サポートされる SQL データ・タイプ . . 794
Java ストアード・プロシージャーと UDF のクラス. 796
COM.ibm.db2.app.StoredProc 796
COM.ibm.db2.app.UDF 797
COM.ibm.db2.app.Lob. 800
COM.ibm.db2.app.Blob 800
COM.ibm.db2.app.Clob 801
NOT FENCED ストアード・プロシージャー 801
入力 SQLDA プログラムの例. 802
入力 SQLDA クライアント・アプリケーション例の動作の仕組み 803
C の例: V5SPCLI.SQC 805
入力 SQLDA ストアード・プロシージャー例の動作の仕組み. 808
C の例: V5SPSRV.SQC 809

この章では、DB2DARI および DB2GENERAL パラメーター様式のストアード・プロシージャーおよび DB2GENERAL UDF の作成方法を説明します。

DB2DARI ストアード・プロシージャー

DB2DARI ストアード・プロシージャーは、呼び出されると、以下の機能を実行します。

1. クライアント・アプリケーションから SQLDA データ構造を受け取る。 (ホスト変数は、SQL CALL ステートメントが実行されたときにデータベース・マネージャーが生成した SQLDA データ構造を介して渡されます。)
2. クライアント・アプリケーションと同じトランザクションの下で、データベース・サーバー上で実行する。
3. SQLCA 情報と任意指定の出力データを、クライアント・アプリケーションに戻す。

クライアント・アプリケーションでの SQLDA の使用

ストアード・プロシージャーを呼び出す前に、使用される SQLDA 構造ごとに以下のステップを実行してください。

1. 要求される基本 SQLVAR エレメント数を指定して、構造にストレージを割り振る。
2. SQLN フィールドを、割り振られる SQLVAR エレメントの数に設定する。

© Copyright IBM Corp. 1993, 2001

3. SQLD フィールドを、実際に使用される SQLVAR エレメントの数に設定する。
4. 以下のようにして、使用される各 SQLVAR エレメントを初期化する。
   - SQLTYPE フィールドを適当なデータ・タイプに設定する。
   - SQLLEN フィールドをそのデータ・タイプのサイズに設定する。
   - SQLTYPE および SQLLEN の値に基づいて、SQLDATA および SQLIND フィールドにストレージを割り振る。

アプリケーションが FOR BIT DATA として定義されている文字ストリングを使用して作動する場合は、 SQLDAID フィールドを初期化して、 FOR BIT DATA 定義および FOR BIT DATA エレメントを定義する各 SQLVAR の SQLNAME フィールドを SQLDA に含めるように指示する必要があります。

アプリケーションが大きなオブジェクトで作動する場合、すなわち CLOB、BLOB、DBCLOB などのタイプのデータで作動する場合、 SQLVAR の第 2 エレメントも初期化する必要があります。 SQLDA 構造の詳細については、 *SQL 解説書* を参照してください。

DB2DARI クライアントにおけるホスト変数の使用

SQLVAR を宣言するには、207ページの『ホスト変数の割り振り』で説明されているのと同じ方法を使います。さらに、クライアント・アプリケーションは、出力専用 SQLVAR の標識を 791ページの『データ構造の操作』で説明されているように -1 に設定しなければなりません。これにより、SQLDATA ポインターの内容ではなく標識のみが送られるので、パラメーターを渡す機構のパフォーマンスが向上します。 SQLTYPE フィールドは、これらのパラメーター用のヌル値可データ・タイプに設定しなければなりません。 SQLTYPE がヌル値不可のデータ・タイプである場合、データベース・マネージャーは標識変数を検査しません。

ストアード・プロシージャーにおける SQLDA の使用

ストアード・プロシージャーは、SQL CALL ステートメントにより呼び出され、クライアント・アプリケーションによって渡されたデータを使用して実行されます。情報は、ストアード・プロシージャーの SQLDA 構造を使用して、クライアント・アプリケーションに戻されます。

SQL CALL ステートメントのパラメーターは、入出力両方のパラメーターとして扱われ、ストアード・プロシージャーのために以下の形式に変換されます。

```
SQL_API_RC SQL_API_FN proc_name( void *reserved1,
                                 void *reserved2,
                                 struct sqlda *inoutsqlda,
                                 struct sqlca *sqlca )
```

SQL_API_FN は、サポートされるオペレーティング・システムごとに異なる関数の呼び出し規則を指定するマクロです。このマクロは、ストアード・プロシージャーや UDF を作成する場合に必要です。

以下に、CALL ステートメントによるサーバーのパラメーター・リストへのマップ方法の例を示します。

```
CALL OUTSRV (:empno:empind,:salary:salind)
```

この呼び出しに対するパラメーターは、SQLVAR を 2 つ持つ SQLDA 構造に変換されます。最初の SQLVAR は、empno ホスト変数および empind 標識変数を指します。 2 番目の SQLVAR は、salary ホスト変数および salind 標識変数を指します。

注: SQLDA 構造は、そのエレメントの数である SQLD が 0 に設定されている場合は、ストアード・プロシージャーに渡されない。この場合、SQLDA が渡されないと、ストアード・プロシージャーは NULL ポインターを受け取ります。

データ構造の操作

データベース・マネージャーは、重複した SQLDA 構造を自動的にデータベース・サーバーに割り振ります。ネットワーク通信量を低減するために重要なのは、入力専用および出力専用のホスト変数をそれぞれ指示することです。クライアントのプロシージャーは、出力専用 SQLVAR の標識を -1 に設定しなければなりません。サーバーのプロシージャーは、入力専用 SQLVAR の標識を -128 に設定しなければなりません。これにより、データベース・マネージャーは、渡される SQLVAR を選択できるようになります。

なお、クライアントまたはサーバーが (SQLVAR を渡さないと指示して) 標識変数を負の値に設定した場合、それはリセットされません。ストアード・プロシージャーまたはクライアント・コードで SQLVAR が参照するホスト変数に値を与える場合、その標識変数は、値が渡されるようにゼロまたは正の値に設定しなければなりません。たとえば、1 つの出力専用パラメーターを取り、以下のように呼び出されるストアード・プロシージャーを例に考えてみます。

```
empind = -1;
EXEC SQL CALL storproc(:empno:empind);
```

ストアード・プロシージャーは、最初の SQLVAR に値を設定する際に標識の値を負ではない値に設定するため、結果は empno に戻されます。

データ構造使用法の要約

表54 には、ストアード・プロシージャーのアプリケーションによるさまざまな構造フィールドの使用法が要約されています。その表の中で、sqlda とはストアード・プロシージャーに渡される SQLDA 構造のことで、 n とは SQLDA の特定の SQLVAR エレメントを示す数値です。右側の数字は、表の後の注番号を示します。

表 54. ストアード・プロシージャーのパラメーター変数

入出力 SQLDA	sqlda.SQLDAID				4					
	sqlda.SQLDABC				4					
	sqlda.SQLN		2		4					
	sqlda.SQLD		2	3		5				
入出力 SQLVAR	sqlda.n.SQLTYPE		2	3		5				
	sqlda.n.SQLLEN		2	3		5				
	sqlda.n.SQLDATA	1	2	3			6		8	
	sqlda.n.SQLIND	1	2	3			6		8	9
	sqlda.n.SQLNAME.length						6	7		
	sqlda.n.SQLNAME.data						6	7		
	sqlda.n.SQLDATATYPE_NAME		2	3		5				
	sqlda.n.SQLLONGLEN		2	3		5				
	sqlda.n.SQLDATALEN	1	2	3			6	7		
SQLCA (全エレメント)							6	7		

注:

ストアード・プロシージャーを呼び出す前に、クライアント・アプリケーションは次のことを行う必要があります。

1. SQLTYPE および SQLLEN に基づいてポインター・エレメントのためのストレージを割り振る。
2. 適切なデータを使ってエレメントを初期化する。

アプリケーションによって呼び出されると、データベース・マネージャーは次のことを行います。

3. 元のエレメント中のデータを、ストアード・プロシージャーで割り振られている重複エレメントに送る。 SQLN エレメントは、SQLD エレメント中のデータで初期化されます。

呼び出しの際、ストアード・プロシージャーは次のことを行えます。

4. 重複エレメント中のデータを変える。データは、妥当性検査をされたりクライアント・アプリケーションに戻されたりしないので、必要に応じて変更することができる。

ストアード・プロシージャーが終了すると、データベース・マネージャーは次のことを行います。

5. 重複エレメント中のデータを検査する。そのフィールド内の値が元のエレメント中のデータと一致していないと、エラーが戻されます。
6. 重複エレメント中のデータを元のエレメントに戻す。
7. データは、妥当性検査をされないので、必要に応じて変更することができる。
8. エレメントにより指示されたデータは、妥当性検査をされたりクライアント・アプリケーションに戻されたりしないので、必要に応じて変更することができる。
9. SQLIND フィールドは、列のタイプがヌル値可でないことを SQLTYPE が示す場合は受け渡しされない。

入出力 SQLDA および SQLCA 構造

ストアード・プロシージャーは、 SQLDA 構造の入力変数に渡されたすべての情報を使用して実行されます。情報は、SQLDA の出力変数でクライアントに戻されます。SQLDA の SQLD、SQLTYPE、および SQLLEN フィールドは、データが戻される前にクライアント・アプリケーションにより設定されている元の値と比較されるため、これらのフィールドの値は変更しないでください。これらが異なる場合は、以下の SQLCODE のうちいずれか 1 つが戻されます。

SQLCODE -1113 (SQLSTATE 39502)
変数のデータ・タイプ (SQLTYPE の値) が変更されました。

SQLCODE -1114 (SQLSTATE 39502)
変数の長さ (SQLLEN の値) が変更されました。

SQLCODE -1115 (SQLSTATE 39502)
SQLD フィールドが変更されました。

さらに、SQLDATA および SQLIND フィールドにより指定される値は変更可能ですが、これらのフィールドのポインターは変更しないでください。

注: 同じ変数を入出力の両方に使用することができます。

SQLCA 情報は、ストアード・プロシージャーが戻る前にストアード・プロシージャーの SQLCA パラメーターに明示的にコピーしなければなりません。

DB2DARI ストアード・プロシージャーの戻り値

ストアード・プロシージャーの戻り値は、クライアント・アプリケーションに戻されることはありません。データベース・マネージャーは、終了時にメモリーから解放するかどうかを判別するのに戻り値を使用します。

ストアード・プロシージャーは、以下のいずれかの値を戻します。

SQLZ_DISCONNECT_PROC
ライブラリーを解放 (アンロード) するようにデータベース・マネージャーに指示します。

SQLZ_HOLD_PROC
サーバー・ライブラリーをメイン・メモリーに保持するようデータベース・マネージャーに指示し、ライブラリーがストアード・プロシージャーの次の呼び出し時に使用できるようにします。これによりパフォーマンスが向上します。

ストアード・プロシージャーが一度だけ呼び出される場合は、`SQLZ_DISCONNECT_PROC` が戻されます。

クライアント・アプリケーションが、同じストアード・プロシージャーを呼び出すために複数の呼び出しを行った場合には、 `SQLZ_HOLD_PROC` がストアード・プロシージャーの戻り値にならなければなりません。ストアード・プロシージャーは、アンロードされません。

`SQLZ_HOLD_PROC` が使用される場合、最後のストアード・プロシージャー呼び出し要求は、メイン・メモリーからストアード・プロシージャー・ライブラリーを解放するために、 `SQLZ_DISCONNECT_PROC` という値を戻さなければなりません。戻されないと、ライブラリーはデータベース・マネージャーが停止するまでメイン・メモリーに残ります。ストアード・プロシージャーに対する警告として、クライアント・アプリケーションは、パラメーターの 1 つに最後の呼び出しを示すフラグを入れて渡すことがあります。

DB2GENERAL UDF およびストアード・プロシージャー

PARAMETER STYLE DB2GENERAL UDF およびストアード・プロシージャーは Java で作成されるので、これ以降は単に Java UDF およびストアード・プロシージャーと呼びます。 DB2GENERAL UDF およびストアード・プロシージャーを作成することは、サポートされている他のプログラム言語で UDF およびストアード・プロシージャーを作成することと非常によく似ています。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。

サポートされる SQL データ・タイプ

PARAMETER STYLE DB2GENERAL UDF およびストアード・プロシージャーを呼び出す場合、 表55 に記述されているように、 DB2 は SQL タイプと Java タイプとを変換します。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 55. DB2 SQL タイプおよび Java オブジェクト

SQL 列名	Java タイプ (UDF)	Java タイプ (ストアード・プロシージャー)
SMALLINT (500/501)	short	short
INTEGER (496/497)	int	int
BIGINT (492/493)	long	long
FLOAT (480/481)	double	double
REAL (480/481)[1]	float	float
DECIMAL(p,s) (484/485)	java.math.BigDecimal	java.math.BigDecimal
NUMERIC(p,s) (504/505)	java.math.BigDecimal	java.math.BigDecimal
CHAR(*n*) (452/453)	String	String
CHAR(*n*) FOR BIT DATA (452/453)	Blob	Blob
C NULL 終了ストリング (400/401)[2]	n/a	String
VARCHAR(*n*)(448/449)	String	String
VARCHAR(*n*) FOR BIT DATA (448/449)	Blob	Blob
LONG VARCHAR (456/457)	String	String
LONG VARCHAR FOR BIT DATA (456/457)	Blob	Blob
GRAPHIC(*n*) (468/469)	String	String
C NULL 終了グラフィック・ストリング (460/461)[2]	n/a	String

表 55. DB2 SQL タイプおよび Java オブジェクト (続き)

SQL 列名	**Java タイプ (UDF)**	**Java タイプ (ストアード・プロシージャー)**
VARGRAPHIC(*n*) (464/465)	String	String
LONG VARGRAPHIC (472/473)[3]	String	String
BLOB(*n*)(404/405)[3]	Blob	Blob
CLOB(*n*) (408/409)[3]	Clob	Clob
DBCLOB(*n*) (412/413)[3]	Clob	Clob
DATE (384/385)[4]	String	String
TIME (388/389)[4]	String	String
TIMESTAMP (392/393)[4]	String	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. C NULL 終了グラフィック・ストリングなどのような、括弧で囲まれたタイプは、呼び出しているアプリケーションがいくつかのタイプのホスト変数の付いた組み込み SQL を使用する場合に、ストアード・プロシージャーで発生します。
3. 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。クラスの詳細は、796ページの『Java ストアード・プロシージャーと UDF のクラス』を参照してください。
4. C でコード化される UDF と同様に、SQL DATE、TIME、および TIMESTAMP 値は、Java でエンコードされる ISO ストリングを使用します。

`COM.ibm.db2.app.Blob` および `COM.ibm.db2.app.Clob` クラスの例は、 LOB データ・タイプ (BLOB、 CLOB、および DBCLOB) を示します。これらのクラスは、入力として渡される LOB を読み込み、出力として戻される LOB を書き込む限定インターフェースを提供します。 LOB の読み込みおよび書き込みは、標準 Java I/O ストリーム・オブジェクトを通して起こります。 Blob クラスの場合、`getInputStream()` および `getOutputStream()` ルーチンは、 BLOB の内容を一度にバイト単位で処理する、 InputStream または OutputStream オブジェクトを戻します。 Clob の場合、`getReader()` および getWriter() 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。

Java ストアード・プロシージャーと UDF のクラス

Java ストアード・プロシージャーは、Java UDF と非常によく似ています。表関数のように、複数の出力がある場合があります。ヌル値の規則も同じですし、出力にも同じ set ルーチンを使用します。主な違いは、ストアード・プロシージャーを含む Java クラスは、 COM.ibm.db2.app.UDF クラスの代わりに、 COM.ibm.db2.app.StoredProc クラスから継承することです。
COM.ibm.db2.app.StoredProc クラスの詳細については、
『COM.ibm.db2.app.StoredProc』を参照してください。

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

以下の節では、これらのクラスの動作のパブリックな性質を説明します。

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、結果設定呼び出しあるいは状況設定呼び出しなど、LOB 引き数での操作を行えません。この規則に違反すると、Java 例外が出されます。

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・タイプは一致し、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。

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 例外が出されます。

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 と共に発呼者に戻されます。

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() を使っ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記号定義は、 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 発呼者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・タイプは一致し、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、例外が出されます。

```
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、例外が出されます。

```
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[] 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。

COM.ibm.db2.app.Lob

このクラスは、ユーザー定義関数やストアード・プロシージャーの内部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。

COM.ibm.db2.app.Blob

このクラスのインスタンスは、BLOB を示すために UDF またはストアード・プロシージャー入力としてデータベースによって渡されます。そのインスタンス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がありますが、実行中の UDF またはストアード・プロシージャーの中だけで作成します。その外にあるオブジェクトを使用すると、例外が出されます。

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。

COM.ibm.db2.app.Clob

このクラスのインスタンスは、CLOB または DBCLOB を示すために UDF またはストアード・プロシージャー入力としてデータベースによって渡されます。このインスタンスは、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がありますが、実行中の UDF またはストアード・プロシージャーの中だけで作成します。その外にあるオブジェクトを使用すると、例外が出されます。

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。

NOT FENCED ストアード・プロシージャー

DB2DARI ストアード・プロシージャーが NOT FENCED ストアード・プロシージャーとして実行されるよう指示するには、 *アプリケーション構築の手引き* に示されているディレクトリーの中にこれを指定してください。 NOT FENCED ストアード・プロシージャーの詳細については、 244ページの『NOT FENCED ストアード・プロシージャー』を参照してください。

入力 SQLDA プログラムの例

以下に、入力 SQLDA 構造の使用方法を示すサンプル・プログラムを挙げます。クライアント・アプリケーションは、Presidents という名前の表を作成し、その表にデータをロードするストアード・プロシージャーを呼び出します。

このプログラムにより、Presidents という名前の表が SAMPLE データベース内に作成されます。次いで、表中に Washington、Jefferson、および Lincoln という値を挿入します。

ストアード・プロシージャーを使用しないと、このサンプル・プログラムは 図23 に示すように、各 SQL ステートメントを処理するために 4 つの別個の要求でネットワークを経由してデータを伝送するように設計されます。

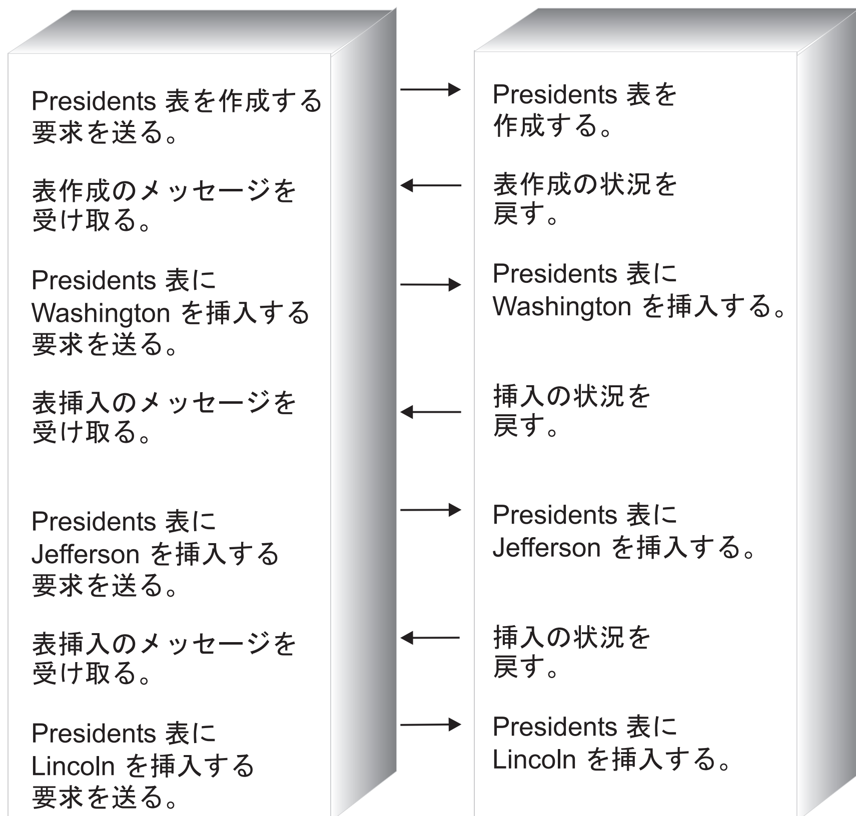


図 23. ストアード・プロシージャーを使用しない場合の入力 SQLDA の例

これとは異なり、サンプル・プログラムがストアード・プロシージャーの技法を活用して 1 つの要求ですべてのデータをネットワークを経由して伝送する場合には、サーバ

ー・プロシージャーが SQL ステートメントをグループとして実行することが可能になります。この技法は図24 に示されています。

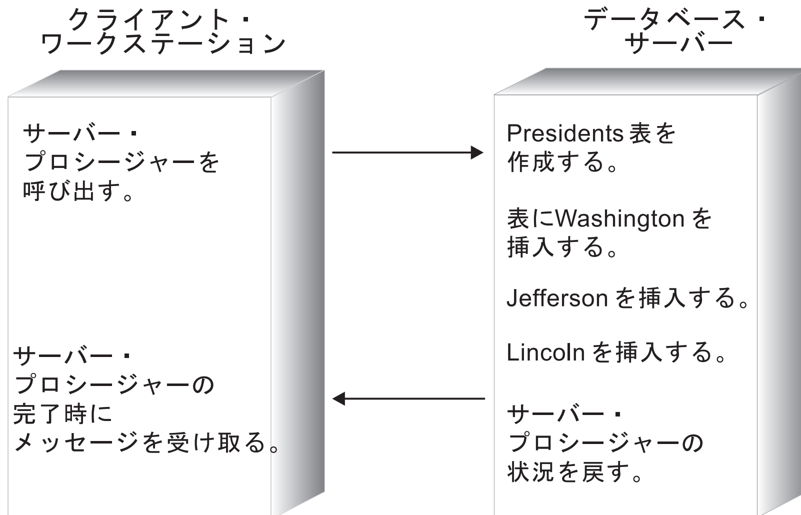


図 24. ストアード・プロシージャーを使用する場合の入力 SQLDA の例

入力 SQLDA クライアント・アプリケーション、および入力 SQLDA ストアード・プロシージャーの例を 803ページに示します。

入力 SQLDA クライアント・アプリケーション例の動作の仕組み

1. **入力 SQLDA 構造を初期化する**。入力 SQLDA の以下のフィールドが初期化されます。
   - SQLN および SQLD エレメントは、割り振られ使用されている SQLVAR エレメントの合計数に設定されます。
   - SQLTYPE エレメントは、文字データ・タイプを示すように設定されます。
   - 最初の SQLDATA エレメントは表の名前に設定されます。 2 番目から 4 番目の SQLDATA エレメントは `Washington`、`Jefferson`、 `Lincoln` に設定されます。
   - SQLLEN エレメントは、各 SQLDATA エレメントの長さ (さらに C 言語ヌル終止符用に 1 バイト加える) に設定されます。
   - SQLIND エレメントは `NULL` に設定されます。
2. **サーバー・プロシージャーを呼び出す**。アプリケーションはデータベース `sample` の位置から次のステートメントを使用してプロシージャー `inpsrv` を呼び出します。
   a. CALL ステートメント (ホスト変数指定)
   b. CALL ステートメント (SQLDA 指定)

`CHECKERR` マクロ / 関数は、プログラム外部にあるエラー検査ユーティリティーです。エラー検査ユーティリティーの所在は、ご使用のプログラム言語により異なります。

C
DB2 API を呼び出す C プログラムの場合、 utilapi.c 内の sqlInfoPrint 関数は、 utilapi.h 内の API_SQL_CHECK として再定義されます。 C 組み込み SQL プログラムの場合、 utilemb.sqc 内の sqlInfoPrint 関数は、 utilemb.h 内の EMB_SQL_CHECK として再定義されます。

COBOL
CHECKERR は checkerr.cbl という名前の外部プログラムです。

FORTRAN
CHECKERR は util.f ファイルにあるサブルーチンです。

REXX
CHECKERR は現行プログラムの終わりにあるプロシージャーです。

このエラー検査ユーティリティーのソース・コードについては、 125ページの『プログラム例での GET ERROR MESSAGE の使用』を参照してください。

C の例: V5SPCLI.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlca.h>
#include <sqlda.h>
#include <sqlutil.h>
#include "util.h"

#define  CHECKERR(CE_STR)   if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION;
     char database[9];
     char userid[9];
     char passwd[19];
     char procname[255]  = "inpsrv";
     char table_name[11]    = "PRESIDENTS";
     char data_item0[21]    = "Washington";
     char data_item1[21]    = "Jefferson";
     char data_item2[21]    = "Lincoln";
     short tableind, dataind0, dataind1, dataind2;
   EXEC SQL END DECLARE SECTION;

   /* Declare Variables for CALL USING */
   struct sqlca    sqlca;
   struct sqlda   *inout_sqlda = NULL;
   char eBuffer[1024]; /* error message buffer */


   if (argc != 4) {
      printf ("¥nUSAGE: inpcli remote_database userid passwd¥n¥n");
      return 1;
   }

   strcpy (database, argv[1]);
   strcpy (userid, argv[2]);
   strcpy (passwd, argv[3]);
   /* Connect to Remote Database */
   printf("CONNECT TO Remote Database.¥n");
   EXEC SQL CONNECT TO :database USER :userid USING :passwd;
   CHECKERR ("CONNECT TO SAMPLE");

   /*********************************************************¥
   * Call the Remote Procedure via CALL with Host Variables *
   ¥*********************************************************/
   printf("Use CALL with Host Variable to invoke the Server Procedure"
      " named inpsrv.¥n");
   tableind = dataind0 = dataind1 = dataind2 = 0;

   EXEC SQL CALL :procname (:table_name:tableind, :data_item0:dataind0,
      :data_item1:dataind1, :data_item2:dataind2);  2a
```

```
/* COMMIT or ROLLBACK the transaction */
if (SQLCODE == 0)
{ EXEC SQL COMMIT;
  printf("Server Procedure Complete.¥n¥n");
}
else
{ /* print the error message, roll back the transaction and return */
  sqlaintp (eBuffer, 1024, 80, &sqlca);
  printf("¥n%s¥n", eBuffer);

  EXEC SQL ROLLBACK;
  printf("Server Procedure Transaction Rolled Back.¥n¥n");
  return 1;
}


/* Allocate and Initialize Input SQLDA */    1
inout_sqlda = (struct sqlda *)malloc( SQLDASIZE(4) );
inout_sqlda->sqln = 4;
inout_sqlda->sqld = 4;

inout_sqlda->sqlvar[0].sqltype = SQL_TYP_NCSTR;
inout_sqlda->sqlvar[0].sqldata = table_name;
inout_sqlda->sqlvar[0].sqllen  = strlen( table_name ) + 1;
inout_sqlda->sqlvar[0].sqlind  = &tableind;

inout_sqlda->sqlvar[1].sqltype = SQL_TYP_NCSTR;
inout_sqlda->sqlvar[1].sqldata = data_item0;
inout_sqlda->sqlvar[1].sqllen  = strlen( data_item0 ) + 1;
inout_sqlda->sqlvar[1].sqlind  = &dataind0;

inout_sqlda->sqlvar[2].sqltype = SQL_TYP_NCSTR;
inout_sqlda->sqlvar[2].sqldata = data_item1;
inout_sqlda->sqlvar[2].sqllen  = strlen( data_item1 ) + 1;
inout_sqlda->sqlvar[2].sqlind  = &dataind1;

inout_sqlda->sqlvar[3].sqltype = SQL_TYP_NCSTR;
inout_sqlda->sqlvar[3].sqldata = data_item2;
inout_sqlda->sqlvar[3].sqllen  = strlen( data_item2 ) + 1;
inout_sqlda->sqlvar[3].sqlind  = &dataind2;

/***********************************************¥
* Call the Remote Procedure via CALL with SQLDA *
¥***********************************************/
printf("Use CALL with SQLDA to invoke the Server Procedure named "
   "inpsrv.¥n");

tableind = dataind0 = dataind1 = dataind2 = 0;
inout_sqlda->sqlvar[0].sqlind  = &tableind;
inout_sqlda->sqlvar[1].sqlind  = &dataind0;
inout_sqlda->sqlvar[2].sqlind  = &dataind1;
inout_sqlda->sqlvar[3].sqlind  = &dataind2;

EXEC SQL CALL :procname USING DESCRIPTOR :*inout_sqlda;   2b
/* COMMIT or ROLLBACK the transaction */
```

```
   if (SQLCODE == 0)
   { EXEC SQL COMMIT;
     printf("Server Procedure Complete.¥n¥n");
   }
   else
   { /* print the error message, roll back the transaction and return */
     sqlaintp (eBuffer, 1024, 80, &sqlca);
     printf("¥n%s¥n", eBuffer);

     EXEC SQL ROLLBACK;
     printf("Server Procedure Transaction Rolled Back.¥n¥n");
     return 1;
   }

   /* Free allocated memory */
   free( inout_sqlda );

   /* Drop the PRESIDENTS table created by the stored procedure */
   EXEC SQL DROP TABLE PRESIDENTS;
   CHECKERR("DROP TABLE");

   /* Disconnect from Remote Database */
   EXEC SQL CONNECT RESET;
   CHECKERR ("CONNECT RESET");
   return 0;
}
/* end of program : inpcli.sqc */
```

入力 SQLDA ストアード・プロシージャー例の動作の仕組み

1. **サーバー・プロシージャーを宣言する。**サーバー・プロシージャーは、SQLDA および SQLCA 構造へのポインターを受け取ります。
2. **表を作成する。** SQLDA 構造の最初の SQLVAR に渡されたデータを使用して、CREATE TABLE ステートメントが構成され、Presidents という名前の表の作成を実行します。
3. **挿入ステートメントを準備する。**パラメーター・マーカー ? を持つ INSERT ステートメントが準備されます。
4. **データを挿入する。**直前に準備された INSERT ステートメントは、SQLDA 構造の 2〜4 番目の SQLVAR に渡されたデータを使用して実行します。パラメーター・マーカーは Washington、Jefferson、Lincoln の値に置き換えられます。これらの値が Presidents 表に挿入されます。
5. **クライアント・アプリケーションに戻す。**サーバー・プロシージャーはこの SQLCA をクライアント・アプリケーションの SQLCA にコピーし、トランザクションが正常に終了した場合に COMMIT ステートメントを発行して、値 SQLZ_DISCONNECT_PROC を戻し、サーバー・プロシージャーからの呼び出しがこれ以上はないことを知らせます。

注: サーバー・プロシージャーは、AIX システム上では REXX で作成することはできません。

C の例: V5SPSRV.SQC

```
#include <memory.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlutil.h>

#ifdef __cplusplus
extern "C"
#endif
SQL_API_RC SQL_API_FN inpsrv(void *reserved1,        1
           void *reserved2,
           struct sqlda   *inout_sqlda,
           struct sqlca   *ca)
{
   /* Declare a local SQLCA */
   EXEC SQL INCLUDE SQLCA;
   /* Declare Host Variables */
   EXEC SQL BEGIN DECLARE SECTION;
     char table_stmt[80]  = "CREATE TABLE ";
     char insert_stmt[80] = "INSERT INTO ";
     char insert_data[21];
   EXEC SQL END DECLARE SECTION;

   /* Declare Miscellanous Variables */
   int  cntr = 0;
   char *table_name;
   char *data_items[3];
   short  data_items_length[3];
   int  num_of_data = 0;

   /*-----------------------------------------------------------------*/
   /* Assign the data from the SQLDA to local variables so that we    */
   /* don't have to refer to the SQLDA structure further.  This will  */
   /* provide better portability to other platforms such as DB2 MVS   */
   /* where they receive the parameter list differently.              */
   /*-----------------------------------------------------------------*/


   table_name  = inout_sqlda->sqlvar[0].sqldata;
   num_of_data = inout_sqlda->sqld - 1;

   for (cntr = 0; cntr < num_of_data; cntr++)
   {
      data_items[cntr] = inout_sqlda->sqlvar[cntr+1].sqldata;
      data_items_length[cntr] = inout_sqlda->sqlvar[cntr+1].sqllen;
   }

   /*-----------------------------------------------------------------*/
   /* Create President Table                                          */
   /* - For simplicity, we'll ignore any errors from the              */
   /*   CREATE TABLE so that you can run this program even when the   */
   /*   table already exists due to a previous run.                   */
   /*-----------------------------------------------------------------*/
```

```
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   strcat(table_stmt, table_name);
   strcat(table_stmt, " (name CHAR(20))");   2

   EXEC SQL EXECUTE IMMEDIATE :table_stmt;

   EXEC SQL WHENEVER SQLERROR GOTO ext;

   /*-------------------------------------------------------------------*/
   /* Generate and execute a PREPARE for an INSERT statement, and     */
   /* then insert the three presidents.                               */
   /*-------------------------------------------------------------------*/

   strcat(insert_stmt, table_name );
   strcat(insert_stmt, "  VALUES (?)");   3

   EXEC SQL PREPARE S1 FROM :insert_stmt;

   for (cntr = 0; cntr < num_of_data; cntr++)
   {
      strncpy(insert_data, data_items[cntr], data_items_length[cntr]);
      insert_data[data_items_length[cntr]] = '¥0';
      EXEC SQL EXECUTE S1 USING :insert_data;   4
   }

   /*-------------------------------------------------------------------*/
   /* Return to caller                                                */
   /*   -  Copy the SQLCA                                             */
   /*   -  Update the output SQLDA.  Since there's no output to       */
   /*      return, we are setting the indicator values to -128 to     */
   /*      return only a null value.                                  */
   /*-------------------------------------------------------------------*/

ext:   5
   memcpy(ca, &sqlca, sizeof(struct sqlca));
   if (inout_sqlda != NULL)
   {
     for (cntr = 0; cntr < inout_sqlda->sqld; cntr++)
     {
       *(inout_sqlda->sqlvar[cntr].sqlind) = -128;
     }
   }

   return(SQLZ_DISCONNECT_PROC);
}
```

付録D. ホストまたは AS/400 環境でのプログラミング

この節には、*DB2 コネクト 使用者の手引き* と共通の情報が含まれています。この節で不明な用語や概念があったときは、 *DB2 コネクト 使用者の手引き* を参照してください。

DB2 コネクトにより、アプリケーション・プログラムは System/390 および AS/400 サーバー上の DB2 データベース内のデータにアクセスできるようになります。たとえば、Windows 上で稼働しているアプリケーションは、 DB2 ユニバーサル・データベース (OS/390 版) データベース内のデータにアクセスできます。 AS/400 環境で稼働する新しいアプリケーションを作成するか、または既存のアプリケーションをこの環境で稼働するように修正できます。ある環境でアプリケーションを開発し、それを別の環境に移植することもできます。

DB2 コネクトを使用すると、 DB2 ユニバーサル・データベース (OS/390 版) などのホスト・データベース製品でサポートされている場合、以下の API をホストとともに使用することができます。

- 静的および動的 SQL
- DB2 コール・レベル・インターフェース
- Microsoft ODBC API
- JDBC

SQL ステートメントの中には、リレーショナル・データベース製品により異なるものもあります。以下のような SQL ステートメントに遭遇する場合があります。

- どの標準に準拠しているかにかかわらず、使用するデータベース製品すべてで同一の SQL ステートメント
- *SQL 解説書* で説明されており、すべての IBM リレーショナル・データベース製品で使用可能な SQL ステートメント
- アクセスするデータベース・システムに固有の SQL ステートメント

初めの 2 つのカテゴリーの SQL ステートメントには高い移植性がありますが、 3 番目のカテゴリーのものは変更を加える必要があります。一般に、データ定義言語 (DDL) 内の SQL ステートメントは、データ操作言語 (DML) 内の SQL ステートメントよりも移植性が低くなります。

DB2 コネクトは、DB2 ユニバーサル・データベースではサポートされていない一部の SQL ステートメントを受け入れます。 DB2 コネクトは、これらのステートメントを AS/400 サーバーに渡します。各種プラットフォームでの制限 (列の長さの最大値など) の詳細については、 *SQL 解説書* を参照してください。

© Copyright IBM Corp. 1993, 2001

別の CICS 製品 (たとえば、CICS for AIX) の下で実行するために CICS アプリケーションを OS/390 または VSE から移動する場合、 DB2 コネクトを使用して OS/390 または VSE データベースにアクセスすることもできます。詳細については、*CICS/6000 アプリケーション・プログラミング・ガイド* および *CICS カスタマイズおよび操作* を参照してください。

注: DB2 コネクトなしの DB2 専用プロトコルを使用するほうが効果的ですが、 DB2 コネクトと DB2 ユニバーサル・データベース・バージョン 7 データベースを使用することもできます。 DB2 ユニバーサル・データベース・バージョン 7 データベースに DB2 コネクトを使用している場合、抽象データ・タイプがサポートされていないなど、 DB2 コネクト自体の制限がある場合を除いて、以下の節で示される非互換性による問題の大部分は当てはまりません。

AS/400 環境でプログラミングする場合は、以下の各要素を考慮してください。

- データ定義言語 (DDL) の使用
- データ操作言語 (DML) の使用
- データ制御言語 (DCL) の使用
- 接続と切断
- プリコンパイル
- ソート順序の定義
- 参照保全の管理
- ロック
- SQLCODE と SQLSTATE の相違点
- システム・カタログの使用
- 分離レベル
- ストアード・プロシージャー
- NOT ATOMIC 複合 SQL
- 分散作業単位
- DB2 コネクトがサポートまたは拒否する SQL ステートメント

データ定義言語 (DDL) の使用

それぞれのシステムでストレージの処理方法が違っているため、 DDL ステートメントは IBM データベース製品により異なります。 AS/400 サーバー・システム上では、データベースを設計してから CREATE TABLE ステートメントを発行するまでにいくつかの手順を踏むことが必要な場合もあります。たとえば論理オブジェクトの設計が、一連のステートメントによりそれらのオブジェクトのストレージでの物理表現に変換されます。

AS/400 サーバー・データベースにプリコンパイルを行う場合、プリコンパイラーは AS/400 サーバーに多数の DDL ステートメントを渡します。アプリケーションを実行中のシステムのデータベースには、同一ステートメントはプリコンパイルされません。たとえば、OS/2 アプリケーションでは、 CREATE STORGROUP ステートメントは DB2 ユニバーサル・データベース (OS/390 版) データベースを正常にプリコンパイルしますが、 DB2 (OS/2 版) データベースはプリコンパイルしません。

データ操作言語 (DML) の使用

一般に、DML ステートメントは高い移植性を持ちます。 SELECT、 INSERT、 UPDATE、および DELETE ステートメントは、 IBM リレーショナル・データベース製品すべてで同様に使用されます。ほとんどのアプリケーションは主として、 DB2 コネクト・プログラムがサポートしている DML SQL ステートメントを使用します。

数値データ・タイプ

数値データが DB2 ユニバーサル・データベースに転送されると、データ・タイプが変更されることがあります。数値およびゾーン 10 進数 SQLTYPE (DB2 ユニバーサル・データベース (AS/400 版) でサポート) は、固定 (パック) 10 進数 SQLTYPE に変換されます。

混合バイト・データ

混合バイト・データは、同一の列にある拡張 UNIX コード (EUC) 文字セット、 2 バイト文字セット (DBCS)、および 1 バイト文字セット (SBCS) の文字からなります。 EBCDIC でデータを保管するシステム (OS/390、OS/400、 VSE、および VM) では、シフトアウト文字とシフトイン文字により 2 バイト・データの始点と終点がマークされます。データを ASCII 形式で保管するシステム (OS/2 および UNIX など) では、シフトイン文字およびシフトアウト文字は必要ありません。

アプリケーションが混合バイト・データを ASCII システムから EBCDIC システムに転送する場合、シフト文字を入れる余地を確保してください。それぞれのデータを SBCS データから DBCS データへ切り替える際に、データ長に 2 バイトが追加されます。移植性を高めるためには、混合バイト・データを使用するアプリケーションにおいて可変長文字ストリングを使用してください。

長フィールド

長フィールド (254 文字より長いストリング) は、別のシステムでは扱いが異なります。 AS/400 サーバーでは長形式フィールドにはスカラー関数のサブセットしかサポートしていません。たとえば、DB2 ユニバーサル・データベース (OS/390 版) では長形式フィールドには **LENGTH** および **SUBSTR** 関数しか使用できません。また、AS/400 サーバーでは特定の SQL ステートメントに対して異なる処理が必要になります。たとえば、DB2 (VSE および VM 版) では INSERT ステートメントにはホスト変数、 SQLDA、またはヌル値しか使用できません。

ラージ・オブジェクト (LOB) データ・タイプ

LOB データ・タイプは DB2 コネクトによってサポートされます。

ユーザー定義タイプ (UDT)

DB2 コネクトは、ユーザー定義特殊タイプをサポートしています。抽象データ・タイプはサポートしていません。

ROWID データ・タイプ

ROWID データ・タイプは、 DB2 コネクトによってビット・データ用の VARCHAR として扱われます。

64 ビット整数 (BIGINT) データ・タイプ

8 バイト (64 ビット) 整数は、DB2 コネクトによってサポートされます。 BIGINT 内部データ・タイプは、データの精度を保ちながら大規模データベースのカーディナリティーをサポートするために使われます。

データ制御言語 (DCL) の使用

それぞれの IBM リレーショナル・データベース管理システムは、 GRANT および REVOKE SQL ステートメントにさまざまなレベルの細分性を提供しています。各データベース管理システムに合った SQL ステートメントを確かめるには、それぞれの製品の資料を調べてみてください。

接続と切断

DB2 コネクトは、パラメーターなしの CONNECT だけでなく、 CONNECT TO および CONNECT RESET もサポートしています。アプリケーションが明示的な CONNECT TO ステートメントを実行せずに SQL ステートメントを呼び出すと、デフォルトのアプリケーション・ サーバー (定義されている場合) への*暗黙の* 接続が実行されます。

データベースに接続すると、 SQLCA の SQLERRP フィールドにリレーショナル・データベース管理システムを識別する情報が戻されます。アプリケーション・サーバーが IBM のリレーショナル・データベースである場合、 SQLERRP の先頭の 3 文字には次のいずれかが含まれます。

DSN DB2 ユニバーサル・データベース (OS/390 版)

ARI DB2 (VSE および VM 版)

QSQ DB2 ユニバーサル・データベース (AS/400 版)

SQL DB2 ユニバーサル・データベース

DB2 コネクトの使用中に CONNECT TO ステートメントまたは NULL の CONNECT ステートメントを発行する場合、 SQLCA の SQLERRMC フィールドに戻る国別コードまたは領域トークンはブランクになります。アプリケーション・サーバーの CCSID は、コード・ページまたはコード・セット・トークンとして戻されます。

CONNECT RESET ステートメント (タイプ 1 の接続の場合)、 RELEASE および COMMIT ステートメント (タイプ 2 の接続の場合)、または DISCONNECT ステートメント (いずれのタイプの接続にも使用できるが、 TP モニター環境では使用できない) を使用して、明示的に切断を行えます。

接続が明示的に切断されていないのに、アプリケーションを通常のように終了すると、DB2 コネクトは結果データを暗黙にコミットします。

注: アプリケーションがエラーを示す SQLCODE を受け取っても、正常に終了することができます。この場合、DB2 コネクトはデータをコミットします。データをコミットしたくない場合は、ROLLBACK コマンドを使用してください。

FORCE コマンドを使用すると、選択したユーザーまたはすべてのユーザーをデータベースから切断することができます。これは、AS/400 サーバー・データベースでサポートされています。ユーザーは、DB2 コネクト・ワークステーションから強制的に切断されます。

プリコンパイル

IBM のリレーショナル・データベース・システムのプリコンパイラーは、システムによりいくらか差があります。 DB2 ユニバーサル・データベース用のプリコンパイラーは、以下の点で AS/400 サーバー・プリコンパイラーとは異なっています。

- アプリケーション全体で、1 つのパスしか作成しない。
- DB2 ユニバーサル・データベースのデータベースに対してバインドする際、正常なバインドのためにはオブジェクトが存在しなければならない。 VALIDATE RUN はサポートされていません。

ブロック化

DB2 コネクトは DB2 データベース・マネージャーのブロック化バインド・オプションをサポートしています。

UNAMBIG
確定カーソルのみをブロック化する (デフォルトの設定)。

ALL 未確定カーソルをブロック化する。

NO カーソルをブロック化しない。

DB2 コネクト・プログラムは、 DB2 データベース・マネージャーの構成ファイルに定義されているブロック・サイズを RQRIOBLK フィールドに使用します。現行バージョンの DB2 コネクトでは、 32 767 までのブロック・サイズをサポートしています。

DB2 データベース・マネージャー構成ファイルにそれより大きい値が指定されても、DB2 コネクトは 32 767 を値として使用しますが、 DB2 データベース・マネージャー構成ファイルをリセットすることはしません。ブロック化は、動的 SQL の場合も静的 SQL の場合も同じブロック・サイズを使用して、同じ方法で処理されます。

注: ほとんどの AS/400 サーバー・システムは動的カーソルを未確定カーソルと見なしていますが、 DB2 ユニバーサル・データベース・システムは一部の動的カーソルを確定カーソルとみなします。混乱を避けるため、DB2 コネクトには BLOCKING ALL を指定できます。

管理 API 解説書 および*コマンド解説書* でリストされている CLP、コントロール・センター、または API を使用し、 DB2 データベース・マネージャー構成ファイルでブロック・サイズを指定します。

パッケージ属性

パッケージには、以下の属性があります。

集合 ID
パッケージの ID です。 PREP コマンドで指定できます。

所有者
パッケージの所有者の許可 ID です。 PREP または BIND コマンドで指定できます。

作成者
パッケージをバインドするユーザー名です。

修飾子
パッケージ内のオブジェクトの暗黙修飾子です。 PREP または BIND コマンドで指定できます。

それぞれの AS/400 サーバー・システムには、これらの属性の使用に制限があります。

DB2 ユニバーサル・データベース (OS/390 版)
4 つの属性すべてが異なってもかまいません。異なる修飾子を使用するには、特別の管理特権が必要です。これらの属性の使用条件の詳細については、DB2 ユニバーサル・データベース (OS/390 版) 用の *コマンド解説書* を参照してください。

DB2 (VSE および VM 版)
すべての属性が一致しなければなりません。 USER1 がバインド・ファイルを作成し (PREP を使用して)、 USER2 が実際にバインドを実行する場合、 USER2 は USER1 のバインドに対する DBA 権限が必要です。属性に使用できるのは USER1 のユーザー名だけです。

DB2 ユニバーサル・データベース (AS/400 版)
修飾子が集合名と一致しなければなりません。修飾子と所有権の関係は、オブジェクトに対する特権の付与および取り消しに影響します。ログオンされたユーザー名は集合 ID で修飾されていない限り、作成者または所有者になりま

す。集合 ID で修飾されている場合は、集合 ID が所有者です。集合 ID は、修飾子として使用される以前にすでに存在していなければなりません。

DB2 ユニバーサル・データベース
4 つの属性すべてが異なってもかまいません。異なる所有者の使用には管理権限が必要であり、バインダーは (スキーマがすでに存在すれば) スキーマに対する CREATEIN 特権を持っていなければなりません。

注: DB2 コネクトには、DB2 ユニバーサル・データベース (OS/390 版) および DB2 ユニバーサル・データベースのために、 *SET CURRENT PACKAGESET* コマンドが備えられています。

C ヌル終了ストリング

CNULREQD バインド・オプションは、LANGLEVEL オプションを使用して指定されたヌル終了ストリングの処理を一時変更します。

LANGLEVEL オプションが MIA または SAA1 の設定値で準備された場合に、ヌル終了ストリングがどのように処理されるかについては、 633ページの『C および C++ での Null 終了ストリング』を参照してください。

CNULREQD デフォルト設定は YES です。このオプションが YES に設定されている場合、ヌル終了ストリングは MIA 規格に準拠して解釈されます。 DB2 ユニバーサル・データベース (OS/390 版) に接続する場合、 CNULREQD を YES に設定するよう強くお勧めします。 (ヌル終了ストリングに関して) SAA1 規格でコーディングされたアプリケーションをバインドする場合は、 CNULREQD オプションを NO に設定する必要があります。 NO に設定しなかった場合、LANGLEVEL を SAA1 に設定して入力したとしても、ヌル終了ストリングは MIA 規格に準拠して解釈されてしまいます。

スタンドアロンの SQLCODE および SQLSTATE

ISO/ANS SQL92 で定義されているスタンドアロンの SQLCODE および SQLSTATE 変数は、 LANGLEVEL SQL92E プリコンパイル・オプションによってサポートされています。 LANGLEVEL がサポートされていなければ、プリコンパイル時にそのことを示す SQL0020W 警告が出されます。この警告は、*コマンド解説書* の LANGLEVEL MIA の下にリストされている、 LANGLEVEL SQL92E のサブセットである機能だけに適用されます。

ソート順序の定義

EBCDIC と ASCII の違いは、さまざまなデータベース製品においてソート順序の違いの原因となり、また ORDER BY および GROUP BY 文節に影響を与えます。この差を最小化するための 1 つの方法は、 EBCDIC のソート順序を模倣したユーザー定義照合順序を作成することです。照合順序を指定できるのは新しいデータベースの作成時のみです。詳細については、 *管理 API 解説書*、および *コマンド解説書* を参照してください。

注: データベース表は、DB2 ユニバーサル・データベース (OS/390 版) に ASCII 形式で保管されています。このため、DB2 コネクトと DB2 ユニバーサル・データベース (OS/390 版) との間の変換が高速になり、データを変換して並べ直すときに使用しなければならないフィールド手順を実行する必要はなくなります。

参照保全の管理

システムにより、参照保全の処理方法が異なります。

DB2 ユニバーサル・データベース (OS/390 版)
基本キーを使用して外部キーを作成できるように、基本キーに索引を付けなければなりません。表は自己参照が可能です。

DB2 (VSE および VM 版)
外部キーには自動的に索引が付けられます。表は自己参照を行えません。

DB2 ユニバーサル・データベース (AS/400 版)
外部キーには自動的に索引が付けられます。表は自己参照が可能です。

DB2 ユニバーサル・データベース
DB2 ユニバーサル・データベースのデータベースの場合、索引は、固有制約や基本キーに対して自動的に作成されます。表は自己参照が可能です。

その他の規則は、カスケードのレベルによって異なります。

ロック

データベース・サーバーがロックを実行する方法は、一部のアプリケーションに影響を与えることがあります。たとえば、行レベルのロックおよびカーソル固定の分離レベルで設計されているアプリケーションは、ページ・レベルのロックを実行しているシステムに直接移植することはできません。このような基礎的な差があるため、アプリケーションを調整する必要があります。

DB2 ユニバーサル・データベース (OS/390 版) および DB2 ユニバーサル・データベース製品には、ロックをタイムアウトにし、待機中のアプリケーションにエラー戻りコードを送信する機能があります。

SQLCODE と SQLSTATE の相違点

それぞれの IBM リレーショナル・データベース製品は、類似したエラーに対して必ずしも同一の SQLCODE を出すとは限りません。この問題は次の 2 とおりの方法で解決できます。

- アプリケーション・エラーに対し、SQLCODE ではなく SQLSTATE を使用する。

  SQLSTATE は、さまざまなデータベース製品間でほぼ同じ意味を持ち、これらの製品は SQLCODE に対応した SQLSTATE を出します。

- SQLCODE をあるシステムから別のシステムへマップする。

  デフォルト設定では、DB2 コネクトは SQLCODE とトークンを、それぞれの IBM ホストまたは AS/400 サーバー・システムからユーザーの DB2 コネクト・システムにマップします。デフォルト・マッピングを一時変更したい場合、または SQLCODE マッピングを持たないデータベース・サーバー (IBM 以外のデータベース・サーバー) を使用している場合に、独自の SQLCODE マッピング・ファイルを指定することができます。 SQLCODE マッピングを作動させないでおくこともできます。

  詳細については、*DB2 コネクト 使用者の手引き* を参照してください。

システム・カタログの使用

システム・カタログは IBM のデータベース製品により異なります。たいていの差異は、視点を使用することによりマスクされます。詳しくは、使用しているデータベース・サーバーの資料を参照してください。

CLI 内のカタログ関数は、DB2 ファミリー間で同一の API、およびカタログ照会の結果のセットをサポートすることにより、この問題を解決します。

検索割り当て時の数値変換のオーバーフロー

検索割り当て時に数値変換のオーバーフローがある場合、 IBM リレーショナル・データベース製品によってその処理は異なります。たとえば、DB2 ユニバーサル・データベース (OS/390 版) および DB2 ユニバーサル・データベースから、浮動列を整数のホスト変数として取り出すとします。浮動値を整数値に変換するときに、変換のオーバーフローが生じることがあります。デフォルトには、DB2 ユニバーサル・データベース (OS/390 版) は警告の SQLCODE とヌル値をアプリケーションに戻します。対照的に、DB2 ユニバーサル・データベースは変換オーバーフロー・エラーを戻します。適切なサイズのホスト変数にして取り出すことにより、アプリケーション側で検索割り当て時の数値変換のオーバーフローを避けるようにお勧めします。

分離レベル

DB2 コネクト、アプリケーションを準備 (prep) またはバインド (bind) するときに、以下の分離レベルを受け入れます。

RR 反復可能読み取り

RS 読み取り固定

CS カーソル固定

UR 非コミット読み取り

NC コミットなし

分離レベルは、最も厳重な保護から最も緩い保護へという順番でリストされています。ユーザーが指定した分離レベルを AS/400 サーバーがサポートしていない場合、サポートされているレベルの中で、指定の分離レベルの次に保護が厳重なものが使用されます。

表56 は、それぞれの AS/400 アプリケーション・サーバーで各分離レベルを指定した場合の結果を示します。

表 56. 分離レベル

DB2 コネクト	DB2 ユニバーサル・データベース (OS/390 版)	DB2 (VSE および VM 版)	DB2 ユニバーサル・データベース (AS/400 版)	DB2 ユニバーサル・データベース
RR	RR	RR	注 1	RR
RS	注 2	RR	COMMIT(*ALL)	RS
CS	CS	CS	COMMIT(*CS)	CS
UR	注 3	CS	COMMIT(*CHG)	UR
NC	注 4	注 5	COMMIT(*NONE)	UR

注:

1. DB2 ユニバーサル・データベース (AS/400 版) には、RR と一致する COMMIT オプションがありません。DB2 ユニバーサル・データベース (AS/400 版) は、すべての表をロックすることにより RR をサポートしています。
2. バージョン 3.1 では RR であったものは、バージョン 4.1 (APAR PN75407) またはバージョン 5.1 では RS になります。
3. バージョン 3.1 では CS であったものは、バージョン 4.1 またはバージョン 5.1 では UR になります。
4. バージョン 3.1 では CS であったものは、バージョン 4.1 (APAR PN60988) またはバージョン 5.1 では UR になります。
5. DB2 (VSE および VM 版) では、分離レベル NC はサポートされません。

DB2 ユニバーサル・データベース (AS/400 版) を使用すると、アプリケーションが UR の分離レベルにバインドされ、かつ ALL に設定されたブロッキングにバインドされる場合、あるいは分離レベルが NC に設定される場合に、ジャーナルされていない表にアクセスできます。

ストアード・プロシージャー

- 呼び出し

クライアント側のプログラムは、SQL CALL ステートメントを出して、サーバー側のプログラムを呼び出すことができます。この場合、各サーバーの作業はその他のサーバーの作業と若干異なります。

OS/390

スキーマ名は 8 バイト以下の長さでなければならず、プロシージャー名は 18 バイト以下の長さでなければならず、ストアード・プロシージャーはサーバー上の SYSIBM.SYSPROCEDURES カタログに定義しなければなりません。

VSE または VM

プロシージャー名は 18 バイト以上の長さでなければならず、サーバー上の SYSTEM.SYSROUTINES カタログに定義しなければなりません。

OS/400

プロシージャー名は SQL の ID でなければなりません。 DECLARE PROCEDURE または CREATE PROCEDURE ステートメントを使用し、ストアード・プロシージャーを突き止めるために実際のパス名 (スキーマ名または集合名) を指定することもできます。

REXX/SQL に組み込まれた CALL ステートメントは CALL USING DESCRIPTOR にマッピングを行うので、 DB2 AS/400 用から REXX/SQL への CALL ステートメントはすべて、アプリケーションによって動的に作成および実行されなければなりません。

SQL CALL ステートメントの構文については、 *SQL 解説書* を参照してください。

DB2 ユニバーサル・データベース上でサーバー・プログラムを起動する場合、そのサーバー・プログラムが DB2 ユニバーサル・データベース (OS/390 版)、DB2 ユニバーサル・データベース (AS/400 版)、または DB2 (VSE および VM 版) で使うものと同じパラメーター規則を使用することができます。 DB2 ユニバーサル・データベースのストアード・プロシージャーの起動の詳細については、 201ページの『第7章 ストアード・プロシージャー』を参照してください。他のプラットフォームのパラメーターの規則の詳細については、そのプラットフォームの DB2 製品の資料を参照してください。

ストアード・プロシージャー内の全 SQL ステートメントは、クライアント側の SQL プログラムにより開始される SQL 作業単位の一部として実行されます。

- ストアード・プロシージャーとの間で、特別の意味を持つインディケーター値をやりとりしない。

  DB2 ユニバーサル・データベースとの間で、システムはなんであれ標識変数に指定されたものを渡します。ただし、DB2 コネクトを使用している場合は、0、-1、および -128 以外の値を標識変数に渡すことはできません。

- サーバー側のアプリケーションで発生したエラーまたは警告を戻すパラメーターを定義する。

DB2 ユニバーサル・データベース上のサーバー・プログラムは、エラーまたは警告を戻すように SQLCA を更新できますが、 DB2 ユニバーサル・データベース (OS/390 版) または DB2 ユニバーサル・データベース (AS/400 版) でのストアード・プロシージャーには、そのようなサポートはありません。ストアード・プロシージャーからエラー・コードを戻したい場合は、これをパラメーターとして渡してください。サーバーが SQLCODE および SQLCA にセットできるエラーは、システムが検出したエラーだけです。

- 現在のところ、ストアード・プロシージャーの結果セットを戻すことができるホストまたは AS/400 アプリケーション・サーバーは、 DB2 (VSE および VM 版) バージョン 7 およびそれ以降、および DB2 ユニバーサル・データベース (OS/390 版) バージョン 5.1 およびそれ以降だけです。

ストアード・プロシージャー・ビルダー

DB2 ストアード・プロシージャー・ビルダーは、ストアード・プロシージャーを作成、インストール、およびテストするための使いやすい開発環境を提供します。これにより、DB2 サーバーでのストアード・プロシージャーの登録、構築、およびインストールに関する詳細というよりも、ストアード・プロシージャーのロジックに注意を向けることができます。さらに、ストアード・プロシージャー・ビルダーを使用すると、あるオペレーティング・システムで開発したストアード・プロシージャーを、他のサーバー・オペレーティング・システムで構築できます。

ストアード・プロシージャー・ビルダーは、迅速な開発をサポートするグラフィカル・アプリケーションです。ストアード・プロシージャー・ビルダーを使用すると以下のような作業を行えます。

- 新規のストアード・プロシージャーを作成する。
- ローカルおよびリモート DB2 サーバーでストアード・プロシージャーを作成する。
- 既存のストアード・プロシージャーを変更して再作成する。
- インストールされたストアード・プロシージャーをテストしてからデバッグする。

ストアード・プロシージャー・ビルダーは DB2 ユニバーサル・データベースのプログラム・グループとは別個のアプリケーションとして立ち上げることもできますし、以下の開発アプリケーションのいずれかから立ち上げることもできます。

- Microsoft Visual Studio
- Microsoft Visual Basic
- IBM VisualAge for Java

DB2 (OS/390 版) のコントロール・センターからストアード・プロシージャー・ビルダーを立ち上げることもできます。ストアード・プロシージャー・ビルダーは、コントロール・センターの「ツール」メニュー、ツールバー、または「ストアード・プロシージャー」フォルダーから別個のメニューとして始動することができます。さらに、「ストアード・プロシージャー・ビルダー・プロジェクト」ウィンドウから、 DB2 (OS/390

版) サーバーに作成した SQL ストアード・プロシージャーを 1 つかそれ以上選択して、コマンド行プロセッサー (CLP) で実行可能な指定したファイルにエクスポートすることができます。

ストアード・プロシージャー・ビルダーは、プロジェクトを使用して作業を管理します。それぞれのストアード・プロシージャー・ビルダー・プロジェクトは、 DB2 (OS/390 版) サーバーなどの特定のデータベースへの接続を保管します。さらに、各データベース上にストアード・プロシージャーのサブセットを表示するためのフィルターを作成することができます。新規または既存のストアード・プロシージャー・ビルダー・プロジェクトを開くとき、その名前、スキーマ、言語、または集合 ID (OS/390 版のみ) に基づくストアード・プロシージャーを表示するため、ストアード・プロシージャーをフィルターすることができます。

接続情報は、ストアード・プロシージャー・ビルダー・プロジェクトに保管されます。そのため、既存のプロジェクトを開くと、そのデータベースのユーザー ID とパスワードを入力するプロンプトが自動的に表示されます。「SQL ストアード・プロシージャーの挿入」ウィザードを使用して、 DB2 (OS/390 版) サーバーで SQL ストアード・プロシージャーを作成することができます。 DB2 (OS/390 版) サーバーに作成した SQL ストアード・プロシージャーの場合、特定のコンパイル、プリリンク、リンク、バインド、実行時、WLM 環境、および外部セキュリティー・オプションを設定することができます。

さらに、SQL ストアード・プロシージャーに関する SQL 見積情報 (SQL ストアード・プロシージャーが実行中のスレッドに応じた CPU 時間や他の DB2 見積情報についての情報を含む) を取得することができます。特に、ラッチ / ロックの競合待ち時間、取得したページ数、読み取り I/O の数、および書き込み I/O の数に関する見積情報を取得することができます。

見積情報を取得するため、ストアード・プロシージャー・ビルダーは DB2 (OS/390 版) に接続し、 SQL ステートメントを実行して、どの程度の CPU 時間と SQL ストアード・プロシージャーが使われるかを知るため、ストアード・プロシージャー (DSNWSPM) を呼び出します。

NOT ATOMIC 複合 SQL

複合 SQL を使用すると、複数の SQL ステートメントをグループ化して単一の実行可能ブロックにすることができます。これによりネットワークのオーバーヘッドが減少し、応答時間が短縮されます。

DB2 コネクトは NOT ATOMIC 複合 SQL をサポートしています。つまり、エラーの発生した後も、複合 SQL の処理が継続されます。 (ATOMIC 複合 SQL を使用した場合、これは DB2 コネクトではサポートされていないため、エラーになったときに複合 SQL のグループ全体がロールバックされてしまいます)。

ステートメントは、アプリケーション・サーバーによって終了されるまで継続して実行されます。一般に、複合 SQL ステートメントの実行は重大エラーの場合にのみサポートされます。

NOT ATOMIC 複合 SQL は、サポートされる AS/400 アプリケーション・サーバーすべてで使用できます。

複数の SQL エラーが発生した場合、先頭から順に 7 つまでの失敗したステートメントの SQLSTATE が、複数のエラーが発生したことを示すメッセージとともに SQLCA の SQLERRMC フィールドに戻されます。詳細については、 *SQL 解説書* を参照してください。

DB2 コネクトでのマルチサイト更新

DB2 コネクトを使用すると、マルチサイト更新 (2 フェーズ・コミットとしても知られる) を実行できます。マルチサイト更新とは、 1 つの分散作業単位 (DUOW) 内で複数のデータベースを更新することです。この機能を使用できるかどうかには、以下のいくつかの要素が関係しています。

- ご使用のアプリケーション・プログラムは、 CONNECT 2 および SYNCPOINT TWOPHASE オプションを指定してプリコンパイルされていなければなりません。
- SNA ネットワークに接続している場合は、 DB2 コネクト エンタープライズ・エディション (AIX 版、OS/2 版、および Windows NT 版) バージョン 7 の同期点管理機能が提供する 2 フェーズ・コミット・サポートを使用することができます。この機能を使用すると、次のホスト・データベース・サーバーが分散作業単位に参加できるようになります。
  - DB2 (AS/400 版) バージョン 3.1 またはそれ以降
  - DB2 (MVS/ESA 版) バージョン 3.1 またはそれ以降
  - DB2 (OS/390 版) バージョン 5.1 またはそれ以降
  - DB2 (VSE および VM 版) バージョン 5.1 またはそれ以降

  以上はネイティブの DB2 UDB アプリケーション、および外部のトランザクション処理 (TP) モニター・プログラム (IBM TXSeries、 CICS for Open Systems、BEA Tuxedo、Encina Monitor、および Microsoft Transaction Server など) によって調整されているアプリケーションに当てはまります。

  注: BEA Tuxedo について詳しくは、 *DB2 コネクト 使用者の手引き* を参照してください。
- TCP/IP ネットワークに接続している場合は、 DB2 (OS/390 版) バージョン 5.1 またはそれ以降が分散作業単位に参加できます。アプリケーションがトランザクション処理モニター・プログラム (IBM TXSeries、 CICS for Open Systems、Encina Monitor、および Microsoft Transaction Server など) で制御されている場合は、同期点管理機能を使わなければなりません。

DB2 アプリケーションと TP モニター・アプリケーションの両方が、共通の DB2 コネクト エンタープライズ・エディション サーバーを使って TCP/IP 接続を介してホスト・データにアクセスする場合は、同期点管理機能を使用しなければなりません。

単一の DB2 コネクト エンタープライズ・エディション サーバーを使って、 SNA および TCP/IP の両方のネットワーク・プロトコルを使用してホスト・データにアクセスする場合は、同期点管理機能を使用しなければなりません。これは、DB2 アプリケーションと TP モニター・アプリケーションの両方に当てはまります。

DB2 コネクトでサポートされている AS/400 サーバー SQL ステートメント

以下のステートメントは AS/400 サーバー処理では正常にコンパイルされますが、 DB2 ユニバーサル・データベース・システムでは処理されません。

- ACQUIRE
- DECLARE (modifier.(qualifier.)table_name TABLE ...
- LABEL ON

これらのステートメントはコマンド行プロセッサーでもサポートされています。

次のステートメントは、AS/400 サーバー処理についてはサポートされていますが、バインド・ファイルまたはパッケージには追加されません。また、コマンド行プロセッサーはこれらのステートメントをサポートしません。

- DESCRIBE statement_name INTO descriptor_name USING NAMES
- PREPARE statement_name INTO descriptor_name USING NAMES FROM ...

プリコンパイラーは、以下の条件を前提事項としています。

- ホスト変数に変数が入力されている。
- ステートメントに固有のセクション番号が割り当てられている。

DB2 コネクトで拒否される AS/400 サーバー SQL ステートメント

以下の SQL ステートメントは、DB2 コネクトでもコマンド行プロセッサーでもサポートされていません。

- COMMIT WORK RELEASE
- DECLARE state_name, statement_name STATEMENT
- DESCRIBE statement_name INTO descriptor_name USING xxxx (xxxx は ANY、BOTH、または LABELS です)
- PREPARE statement_name INTO descriptor_name USING xxxx FROM :host_variable (xxxx は ANY、BOTH、または LABELS です)
- PUT ...
- ROLLBACK WORK RELEASE

- SET :host_variable = CURRENT ...

DB2 (VSE および VM 版) の拡張動的 SQL ステートメントは拒否され、 -104 および構文エラーの SQLCODE が出されます。

付録E. EBCDIC バイナリー照合のシミュレート

DB2 では、ユーザー定義の照合順序にしたがって文字ストリングを照合できます。この機能は、EBCDIC バイナリー照合のシミュレートに使えます。

EBCDIC 照合をシミュレートする方法の一例として、コード・ページ 850 の ASCII データベースを作成し、コード・ページ 500 の EBCDIC データベースに実際にデータが存在することを前提として文字ストリングを照合するとしましょう。コード・ページ 500 の定義については 830ページの図26、コード・ページ 850 の定義については 831ページの図27 を参照してください。

EBCDIC コード・ページ 500 のデータベースにある 4 つの文字を相対的に照合することについて考えます。これらの文字は次のように 2 進で照合されています。

文字	コード・ページ 500 コード・ポイント
'a'	X'81'
'b'	X'82'
'A'	X'C1'
'B'	X'C2'

コード・ページ 500 の バイナリー照合順序 (希望する順序) は、次のとおりです。

```
'a' < 'b' < 'A' < 'B'
```

データベースを ASCII コード・ページ 850 で作成する場合、バイナリー照合の結果は以下のようになります。

文字	コード・ページ 850 コード・ポイント
'a'	X'61'
'b'	X'62'
'A'	X'41'
'B'	X'42'

コード・ページ 850 の バイナリー照合 (希望する順序ではない) は、次のとおりです。

```
'A' < 'B' < 'a' < 'b'
```

希望する順序を実現するには、ユーザー定義照合順序でデータベースを作成する必要があります。 `sqle850a.h` インクルード・ファイルの DB2 には、まさにこの目的のために照合順序のサンプルが付属しています。 `sqle850a.h` の内容は、828ページの図25 に示されています。

```
#ifndef SQL_H_SQLE850A
#define SQL_H_SQLE850A

#ifdef __cplusplus
extern "C" {
#endif

unsigned char sqle_850_500[256] = {
0x00,0x01,0x02,0x03,0x37,0x2d,0x2e,0x2f,0x16,0x05,0x25,0x0b,0x0c,0x0d,0x0e,0x0f,
0x10,0x11,0x12,0x13,0x3c,0x3d,0x32,0x26,0x18,0x19,0x3f,0x27,0x1c,0x1d,0x1e,0x1f,
0x40,0x4f,0x7f,0x7b,0x5b,0x6c,0x50,0x7d,0x4d,0x5d,0x5c,0x4e,0x6b,0x60,0x4b,0x61,
0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0x7a,0x5e,0x4c,0x7e,0x6e,0x6f,
0x7c, 0xc1 , 0xc2 ,0xc3,0xc4,0xc5,0xc6,0xc7,0xc8,0xc9,0xd1,0xd2,0xd3,0xd4,0xd5,
0xd6,0xd7,0xd8,0xd9,0xe2,0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0x4a,0xe0,0x5a,0x5f,
0x6d,0x79, 0x81 , 0x82 ,0x83,0x84,0x85,0x86,0x87,0x88,0x89,0x91,0x92,0x93,0x94,
0x95,0x96,0x97,0x98,0x99,0xa2,0xa3,0xa4,0xa5,0xa6,0xa7,0xa8,0xa9,0xc0,0xbb,0xd0,
0xa1,0x07,0x68,0xdc,0x51,0x42,0x43,0x44,0x47,0x48,0x52,0x53,0x54,0x57,0x56,0x58,
0x63,0x67,0x71,0x9c,0x9e,0xcb,0xcc,0xcd,0xdb,0xdd,0xdf,0xec,0xfc,0x70,0xb1,0x80,
0xbf,0xff,0x45,0x55,0xce,0xde,0x49,0x69,0x9a,0x9b,0xab,0xaf,0xba,0xb8,0xb7,0xaa,
0x8a,0x8b,0x2b,0x2c,0x09,0x21,0x28,0x65,0x62,0x64,0xb4,0x38,0x31,0x34,0x33,0xb0,
0xb2,0x24,0x22,0x17,0x29,0x06,0x20,0x2a,0x46,0x66,0x1a,0x35,0x08,0x39,0x36,0x30,
0x3a,0x9f,0x8c,0xac,0x72,0x73,0x74,0x0a,0x75,0x76,0x77,0x23,0x15,0x14,0x04,0x6a,
0x78,0x3b,0xee,0x59,0xeb,0xed,0xcf,0xef,0xa0,0x8e,0xae,0xfe,0xfb,0xfd,0x8d,0xad,
0xbc,0xbe,0xca,0x8f,0x1b,0xb9,0xb6,0xb5,0xe1,0x9d,0x90,0xbd,0xb3,0xda,0xfa,0xea,
d0x3e,0x41
};
#ifdef __cplusplus
}
#endif

#endif /* SQL_H_SQLE850A */
```

図 25. ユーザー定義の照合順序 - sqle_850_500

コード・ページ 500 の バイナリー照合をコード・ページ 850 の文字で実現する方法を確かめるには、`sqle_850_500` にある照合順序のサンプルを調べてください。コード・ページ 850 のそれぞれの文字の照合順序における重要度は、コード・ページ 500 の対応するコード・ポイントに相当する以上のものではありません。

たとえば、文字 ‘a’ を考えてみましょう。この文字は、831ページの図27 に示されているように、コード・ページ 850 ではコード・ポイント X'61' になります。配列 `sqle_850_500` で、文字 ‘a’ には X'81' の重要度 (つまり、配列 `sqle_850_500` の 98 番目のエレメント) が割り当てられています。

データベースが上記のサンプルに示したユーザー定義の照合順序で作成されている場合、これら 4 つの文字の照合方法はどうなるか考慮してみます。

文字	コード・ページ 850 コード・ポイント / 重要度 (sqle_850_500 からの)
’a’	X'61' / X'81'
’b’	X'62' / X'82'
’A’	X'41' / X'C1'

'B' X'42' / X'C2'

重要度別のコード・ページ 850 のユーザー定義照合 (希望する照合) は、次のとおりです。

```
'a' < 'b' < 'A' < 'B'
```

この例では、正確な重要度を指定して希望する照合を実現し、希望する動作をシミュレートしています。

実際の照合順序を注意深く観察すると、順序そのものは単に変換表になっているだけで、ソース・コード・ページがデータベースのコード・ページ (850)、ターゲット・コード・ページが希望する バイナリー照合コード・ページ (500) であることが分かります。 DB2 提供の他の照合順序のサンプルにより、さまざまな変換が可能になります。必要な変換テーブルが DB2 で提供されていない場合は、 IBM 資料 *Character Data Representation Architecture, Reference and Registry* (SC09-2190) から追加の変換テーブルを入手できます。追加の変換テーブルは、この資料に同梱されている CD-ROM に収録されています。

照合順序の詳細については、518ページの『照合順序の概説』を参照してください。 DB2 で提供されている照合順序、およびユーザー定義の照合順序でデータベースを作成する方法を示したサンプル・プログラム (db_udcs.c) については、 *管理 API 解説書* に記載されている CREATE DATABASE API を参照してください。

HEX DIGITS 1ST → 2ND ↓	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0	(SP) SP010000	& SM030000	- SP100000	ø LO610000	Ø LO620000	° SM190000	µ SM170000	¢ SC040000	{ SM110000	} SM140000	\ SM070000	0 ND100000
-1	(RSP) SP300000	é LE110000	/ SP120000	É LE120000	a LA010000	j LJ010000	~ SD190000	£ SC020000	A LA020000	J LJ020000	÷ SA060000	1 ND010000
-2	â LA150000	ê LE150000	Â LA160000	Ê LE160000	b LB010000	k LK010000	s LS010000	¥ SC050000	B LB020000	K LK020000	S LS020000	2 ND020000
-3	ä LA170000	ë LE170000	Ä LA180000	Ë LE180000	c LC010000	l LL010000	t LT010000	· SD630000	C LC020000	L LL020000	T LT020000	3 ND030000
-4	à LA130000	è LE130000	À LA140000	È LE140000	d LD010000	m LM010000	u LU010000	© SM520000	D LD020000	M LM020000	U LU020000	4 ND040000
-5	á LA110000	í LI110000	Á LA120000	Í LI120000	e LE010000	n LN010000	v LV010000	§ SM240000	E LE020000	N LN020000	V LV020000	5 ND050000
-6	ã LA190000	î LI150000	Ã LA200000	Î LI160000	f LF010000	o LO010000	w LW010000	¶ SM250000	F LF020000	O LO020000	W LW020000	6 ND060000
-7	å LA270000	ï LI170000	Å LA280000	Ï LI180000	g LG010000	p LP010000	x LX010000	¼ NF040000	G LG020000	P LP020000	X LX020000	7 ND070000
-8	ç LC410000	ì LI130000	Ç LC420000	Ì LI140000	h LH010000	q LQ010000	y LY010000	½ NF010000	H LH020000	Q LQ020000	Y LY020000	8 ND080000
-9	ñ LN190000	ß LS610000	Ñ LN200000	` SD130000	i LI010000	r LR010000	z LZ010000	¾ NF050000	I LI020000	R LR020000	Z LZ020000	9 ND090000
-A	[SM060000	] SM080000	¦ SM650000	: SP130000	« SP170000	ª SM210000	¡ SP030000	¬ SM660000	(SHY) SP320000	¹ ND011000	² ND021000	³ ND031000
-B	. SP110000	$ SC030000	, SP080000	# SM010000	» SP180000	º SM200000	¿ SP160000	| SM130000	ô LO150000	û LU150000	Ô LO160000	Û LU160000
-C	< SA030000	* SM040000	% SM020000	@ SM050000	ð LD630000	æ LA510000	Ð LD620000	¯ SM150000	ö LO170000	ü LU170000	Ö LO180000	Ü LU180000
-D	(SP060000	) SP070000	_ SP090000	' SP050000	ý LY110000	¸ SD410000	Ý LY120000	¨ SD170000	ò LO130000	ù LU130000	Ò LO140000	Ù LU140000
-E	+ SA010000	; SP140000	> SA050000	= SA040000	þ LT630000	Æ LA520000	Þ LT640000	´ SD110000	ó LO110000	ú LU110000	Ó LO120000	Ú LU120000
-F	! SP020000	^ SD150000	? SP150000	" SP040000	± SA020000	¤ SC010000	® SM530000	× SA070000	õ LO190000	ÿ LY170000	Õ LO200000	(EO)

Code Page 00500

図 26. コード・ページ 500

HEX DIGITS 1ST → 2ND ↓	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0		► SM590000	(SP) SP010000	0 ND100000	@ SM050000	P LP020000	` SD130000	p LP010000	Ç LC420000	É LE120000	á LA110000	░ SF140000	└ SF020000	ð LD630000	Ó LO120000	(SHY) SP320000
-1	☺ SS000000	◄ SM630000	! SP020000	1 ND010000	A LA020000	Q LQ020000	a LA010000	q LQ010000	ü LU170000	æ LA510000	í LI110000	▒ SF150000	┴ SF070000	Đ LD620000	ß LS610000	± SA020000
-2	☻ SS010000	↕ SM760000	" SP040000	2 ND020000	B LB020000	R LR020000	b LB010000	r LR010000	é LE110000	Æ LA520000	ó LO110000	▓ SF160000	┬ SF060000	Ê LE160000	Ô LO160000	‗ SM100000
-3	♥ SS020000	‼ SP330000	# SM010000	3 ND030000	C LC020000	S LS020000	c LC010000	s LS010000	â LA150000	ô LO150000	ú LU110000	│ SF110000	├ SF080000	Ë LE180000	Ò LO140000	¾ NF050000
-4	♦ SS030000	¶ SM250000	$ SC030000	4 ND040000	D LD020000	T LT020000	d LD010000	t LT010000	ä LA170000	ö LO170000	ñ LN190000	┤ SF090000	─ SF100000	È LE140000	õ LO190000	¶ SM250000
-5	♣ SS040000	§ SM240000	% SM020000	5 ND050000	E LE020000	U LU020000	e LE010000	u LU010000	à LA130000	ò LO130000	Ñ LN200000	Á LA120000	┼ SF050000	ı LI610000	Õ LO200000	§ SM240000
-6	♠ SS050000	▬ SM700000	& SM030000	6 ND060000	F LF020000	V LV020000	f LF010000	v LV010000	å LA270000	û LU150000	ª SM210000	Â LA160000	ã LA190000	Í LI120000	µ SM170000	÷ SA060000
-7	• SM570000	↨ SM770000	' SP050000	7 ND070000	G LG020000	W LW020000	g LG010000	w LW010000	ç LC410000	ù LU130000	º SM200000	À LA140000	Ã LA200000	Î LI160000	þ LT630000	¸ SD410000
-8	◘ SM570001	↑ SM320000	(SP060000	8 ND080000	H LH020000	X LX020000	h LH010000	x LX010000	ê LE150000	ÿ LY170000	¿ SP160000	© SM520000	╚ SF380000	Ï LI180000	Þ LT640000	° SM190000
-9	○ SM750000	↓ SM330000	) SP070000	9 ND090000	I LI020000	Y LY020000	i LI010000	y LY010000	ë LE170000	Ö LO180000	® SM530000	╣ SF230000	╔ SF390000	┘ SF040000	Ú LU120000	¨ SD170000
-A	◙ SM750002	→ SM310000	* SM040000	: SP130000	J LJ020000	Z LZ020000	j LJ010000	z LZ010000	è LE130000	Ü LU180000	¬ SM660000	║ SF240000	╩ SF400000	┌ SF010000	Û LU160000	˙ SD630000
-B	♂ SM280000	← SM300000	+ SA010000	; SP140000	K LK020000	[SM060000	k LK010000	{ SM110000	ï LI170000	ø LO610000	½ NF010000	╗ SF250000	╦ SF410000	█ SF610000	Ù LU140000	¹ ND011000
-C	♀ SM290000	∟ SA420000	, SP080000	< SA030000	L LL020000	\ SM070000	l LL010000	| SM130000	î LI150000	£ SC020000	¼ NF040000	╝ SF260000	╠ SF420000	▄ SF570000	ý LY110000	³ ND031000
-D	♪ SM930000	↔ SM780000	- SP100000	= SA040000	M LM020000	] SM080000	m LM010000	} SM140000	ì LI130000	Ø LO620000	¡ SP030000	¢ SC040000	═ SF430000	¦ SM650000	Ý LY120000	² ND021000
-E	♫ SM910000	▲ SM600000	. SP110000	> SA050000	N LN020000	^ SD150000	n LN010000	~ SD190000	Ä LA180000	× SA070000	« SP170000	¥ SC050000	╬ SF440000	Ì LI140000	¯ SM150000	■ SM470000
-F	☼ SM690000	▼ SV040000	/ SP120000	? SP150000	O LO020000	_ SP090000	o LO010000	⌂ SM790000	Å LA280000	ƒ SC070000	» SP180000	┐ SF030000	¤ SC010000	▀ SF600000	´ SD110000	(RSP) SP300000

Code Page 00850

図 27. コード・ページ 850

付録F. DB2 ライブラリーの使用法

DB2 ユニバーサル・データベース ライブラリーは、オンライン・ヘルプ、ブック (PDF および HTML)、および HTML 形式のサンプル・プログラムから成っています。このセクションでは、ユーザーに提供される情報について紹介し、その入手方法を示します。

オンライン製品情報をご利用になるには、インフォメーション・センターを使用することができます。詳細については、849ページの『インフォメーション・センターを使用した情報へのアクセス』を参照してください。ここではタスク情報、DB2 ブック、トラブルシューティング情報、サンプル・プログラム、および Web の DB2 情報を見ることができます。

DB2 PDF ファイルおよびハードコピー版資料

DB2 情報

以下に示す表では、DB2 ブックを 4 つのカテゴリーに分類しています。

DB2 の手引きおよび解説書
これらの資料は、すべてのプラットフォームに共通の DB2 情報を含んでいます。

DB2 のインストールおよび構成の情報
これらの資料は、特定のプラットフォーム上の DB2 ごとに用意されています。たとえば、 OS/2、Windows、および UNIX ベースのプラットフォームで稼働するそれぞれの DB2 用に、別個の*概説およびインストール* 資料が用意されています。

プラットフォーム共通のサンプル・プログラム (HTML 形式)
これらのサンプルは、アプリケーション開発クライアントとともにインストールされるサンプル・プログラムの HTML 版です。これらのサンプルは参考用であり、実際のプログラムに代わるものではありません。

リリース情報
これらのファイルには、DB2 ブックには含められなかった最新の情報が記載されています。

インストール情報、リリース情報、およびチュートリアルは、製品 CD-ROM から HTML 形式で参照することができます。ほとんどの資料は、製品 CD-ROM から HTML 形式で表示できますし、 DB2 の資料 CD-ROM から Adobe Acrobat (PDF) 形

© Copyright IBM Corp. 1993, 2001

式で表示し印刷することができます。 IBM にハードコピー版の資料を注文したい場合は、 845ページの『印刷資料の注文方法』を参照してください。注文可能な資料については、以下の表をご覧ください。

OS/2 および Windows プラットフォームの場合、 HTML ファイルは sqllib¥doc¥html ディレクトリーにインストールできます。 DB2 情報はいくつかの言語で提供されています。しかし、すべての言語に翻訳されているわけではありません。ある言語で情報が提供されていない場合は、英語版の情報が提供されます。

UNIX プラットフォームの場合、言語ごとに異なる複数の HTML ファイルを doc/*%L*/html ディレクトリーにインストールできます。ここで、*%L* は地域を表しています。詳細については、適切な*概説およびインストールの手引き* を参照してください。

DB2 ブックを入手して情報を利用するには、次のようなさまざまな方法があります。

- 848ページの『オンライン情報の表示』
- 853ページの『オンライン情報の検索』
- 845ページの『印刷資料の注文方法』
- 845ページの『PDF 資料の印刷』

表 57. DB2 情報

資料名	説明	資料番号 PDF ファイル名	HTML ディレクトリー
	DB2 の手引きおよび解説書情報		
管理の手引き	*管理の手引き: 計画* は、データベース概念について概説し、設計 (たとえば、論理および物理データベース設計) に関する情報を提供し、高い可用性について解説しています。	SC88-8513 db2d1x70	db2d0
	管理の手引き: インプリメンテーション は、設計、データベースへのアクセス、監査、バックアップ、およびリカバリーなどのインプリメンテーションについて説明しています。	SC88-8511 db2d2x70	
	管理の手引き: パフォーマンス は、データベース環境について解説し、さらにアプリケーションのパフォーマンスの評価と調整の方法について説明しています。	SC88-8512 db2d3x70	
管理 API 解説書	データベースの管理に使用できる DB2 アプリケーション・プログラミング・インターフェース (API) およびデータ構造について説明します。また、この資料は、アプリケーションから API を呼び出す方法も示します。	SC88-8514 db2b0x70	db2b0
アプリケーション構築の手引き	環境設定に関する情報を提供し、Windows、OS/2、および UNIX ベースのプラットフォームでの DB2 アプリケーションのコンパイル、リンク、実行の各ステップについて説明します。	SC88-8515 db2axx70	db2ax
APPC, CPI-C, and SNA Sense Codes	DB2 ユニバーサル・データベース製品をご使用中に発生する可能性のあるセンス・コード APPC、CPI-C、および SNA についての一般情報を提供します。 HTML 形式でのみご利用いただけます。	資料番号なし db2apx70	db2ap

表 57. DB2 情報 (続き)

資料名	説明	資料番号 PDF ファイル名	HTML ディレクトリー
アプリケーション開発の手引き	DB2 データベースにアクセスするアプリケーションを、組み込み SQL または Java (JDBC および SQLJ) を使用して開発する方法について説明します。さらに、ストアード・プロシージャーの作成方法、ユーザー定義関数の作成方法、ユーザー定義タイプの作成方法、トリガーの使用方法、区画化されている環境または統合されているシステムでのアプリケーションの開発方法などについて解説されています。	SC88-8516 db2a0x70	db2a0
コール・レベル・インターフェースの手引きおよび解説書	DB2 データベースにアクセスするアプリケーションを、 DB2 コール・レベル・インターフェース (Microsoft ODBC 仕様互換の呼び出し可能 SQL) を使用して開発する方法について説明します。	SC88-8517 db2l0x70	db2l0
コマンド解説書	コマンド行プロセッサーの使用法について説明し、データベースの管理に使用できる DB2 コマンドについて解説しています。	SC88-8518 db2n0x70	db2n0
コネクティビティー 補足	DB2 (AS/400 版)、DB2 (OS/390 版)、DB2 (MVS 版)、または DB2 (VM 版) を DRDA アプリケーション・リクエスターとして DB2 ユニバーサル・データベースとともに使用するためのセットアップ情報および参照情報を提供します。また、この資料は DRDA アプリケーション・サーバーを DB2 コネクト アプリケーション・リクエスターとともに使用する方法の詳細を示します。 HTML と PDF でのみ利用可能	資料番号なし db2h1x70	db2h1
データ移動ユーティリティー手引きおよび解説書	データの移動を行う DB2 ユーティリティー (インポート、エクスポート、ロード、AutoLoader、および DPROP など) の使用法について説明しています。	SC88-8522 db2dmx70	db2dm

表 57. DB2 情報 (続き)

資料名	説明	資料番号 PDF ファイル名	HTML ディレクトリー
データウェアハウスセンター 管理の手引き	データウェアハウスセンターを使用してデータウェアハウスを構築および保守する方法を説明します。	SC88-8545 db2ddx70	db2dd
データウェアハウスセンター アプリケーション統合の手引き	プログラマーがアプリケーションをデータウェアハウスセンターおよび情報カタログ・マネージャーと統合するのに役立つ情報を提供します。	SC88-8546 db2adx70	db2ad
DB2 コネクト 使用者の手引き	DB2 コネクト製品の概念、プログラミング、および一般的な使用方法に関する情報を提供します。	SC88-8521 db2c0x70	db2c0
DB2 クエリー・パトローラー 管理の手引き	DB2 クエリー・パトローラー・システムの運用の概説を行い、運用および管理に関する詳細情報、および管理用グラフィカル・ユーザー・インターフェース・ユーティリティーについてのタスク情報を提供します。	SC88-8525 db2dwx70	db2dw
DB2 クエリー・パトローラー 使用者の手引き	DB2 クエリー・パトローラーのツールや関数の使用方法を説明します。	SC88-8527 db2wwx70	db2ww
用語集	DB2 およびそのコンポーネントで使用される用語の定義を示します。 HTML 形式と *SQL 解説書* で利用可能	資料番号なし db2t0x70	db2t0
イメージ、オーディオ、およびビデオ・エクステンダー 管理およびプログラミングの手引き	DB2 エクステンダーの一般情報について提供し、画像、音声、およびビデオ (IAV) エクステンダーの管理と構成について、および IAV エクステンダーを使用したプログラミングについて説明しています。さらに、参照情報、診断情報 (メッセージ解説)、およびサンプルも収録されています。	SC88-8609 dmbu7x70	dmbu7
情報カタログ・マネージャー 管理の手引き	情報カタログを管理するためのガイドです。	SC88-8547 db2dix70	db2di
情報カタログ・マネージャー プログラミングの手引きおよび解説書	情報カタログ・マネージャー用の体系化されたインターフェースの定義を示します。	SC88-8549 db2bix70	db2bi

表 57. DB2 情報 (続き)

資料名	説明	資料番号 PDF ファイル名	HTML ディレクトリー
情報カタログ・マネージャー 使用者の手引き	情報カタログ・マネージャー・ユーザー・インターフェースの使用に関する情報を提供します。	SC88-8548 db2aix70	db2ai
インストールおよび構成補足	プラットフォーム固有の DB2 クライアントの計画、インストール、およびセットアップのガイドです。この補足資料には、バインド、クライアント / サーバー通信の設定、 DB2 GUI ツール、DRDA AS、分散インストール、分散要求の構成、および異機種データ・ソースへのアクセスについても説明されています。	GC88-8524 db2iyx70	db2iy
メッセージ解説書	DB2、情報カタログ・マネージャー、およびデータウェアハウスセンターから出されるメッセージとコードをリストし、取るべき処置を解説しています。	第 1 巻 GC88-8543 db2m1x70 第 2 巻 GC88-8544 db2m2x70	db2m0
OLAP Integration Server Administration Guide	OLAP Integration Server の Administration Manager コンポーネントの使用方法を説明します。	SC27-0787 db2dpx70	n/a
OLAP Integration Server Metaoutline User's Guide	標準の OLAP Metaoutline インターフェースを使用して (Metaoutline Assistant を使用するのではなく) OLAP metaoutline を作成しデータを取り込む方法を説明しています。	SC27-0784 db2upx70	n/a
OLAP Integration Server Model User's Guide	(Model Assistant ではなく) 標準的な OLAP Model Interface を使用して OLAP モデルを作成する方法を説明します。	SC27-0783 db2lpx70	n/a
OLAP のセットアップおよびユーザーズ・ガイド	OLAP スターター・キットの構成およびセットアップに関する情報を提供します。	SC88-8652 db2ipx70	db2ip
Hyperion Essbase スプレッドシート アドイン ユーザーズ ガイド for Excel	Excel 作表計算プログラムを使用して OLAP データを分析する方法を説明します。	SC88-8724 db2epx70	db2ep

表 57. DB2 情報 (続き)

資料名	説明	資料番号 PDF ファイル名	HTML ディレクトリー
Hyperion Essbase スプレッドシート アドイン ユーザーズ ガイド for Lotus 1-2-3	ロータス 1-2-3 作表計算プログラムを使用して OLAP データを分析する方法を説明します。	SC88-8723 db2tpx70	db2tp
レプリケーションの手引きおよび解説書	DB2 に付属の IBM レプリケーション・ツールの計画、構成、管理、および使用方法に関する情報を提供します。	SC88-8550 db2e0x70	db2e0
地理情報エクステンダー使用者の手引きおよび解説書	地理情報エクステンダーのインストール、構成、管理、プログラミング、およびトラブルシューティングに関する情報を提供します。また、地理情報データの概念についての重要事項を示し、地理情報エクステンダー固有の参照情報 (メッセージおよび SQL) を提供します。	SC88-8624 db2sbx70	db2sb
SQL 概説	SQL の概念を紹介し、構造体とタスクの例を多数提供しています。	SC88-8539 db2y0x70	db2y0
SQL 解説書	SQL の構文、セマンティクス、および言語規則について説明します。また、この資料には、各リリース間の互換性、製品の制限事項、およびカタログ・ビューも含まれます。	第 1 巻 SC88-8540 db2s1x70 第 2 巻 SC88-8657 db2s2x70	db2s0
システム・モニター 手引きおよび解説書	データベースおよびデータベース・マネージャーに関連したさまざまな情報を収集する方法を示します。この資料は、この情報を利用して、データベース活動の把握、パフォーマンス向上、および問題原因の判別を行う方法を説明しています。	SC88-8523 db2f0x70	db2f0

表 57. DB2 情報 (続き)

資料名	説明	資料番号 PDF ファイル名	HTML ディレクトリー
テキスト・エクステンダー 管理およびプログラミング	DB2 エクステンダーの一般情報、テキスト・エクステンダーの管理および構成情報、およびテキスト・エクステンダーを使用したプログラミングの方法について解説します。この資料には、参照情報、診断情報 (メッセージ解説)、およびサンプルが含まれています。	SC88-8610 desu9x70	desu9
問題判別の手引き	エラーの原因の判別、問題からの回復、および DB2 カスタマー・サービスの支援の下での診断ツールの使用法を記載しています。	GD88-7271 db2p0x70	db2p0
新機能	DB2 ユニバーサル・データベース バージョン 7 の新しい機能および拡張機能について説明します。	SC88-8541 db2q0x70	db2q0
	DB2 のインストールおよび構成の情報		
DB2 コネクト エンタープライズ・エディション (OS/2 および Windows 版) 概説およびインストール	OS/2 および Windows 32 ビット・オペレーティング・システム版の DB2 コネクト エンタープライズ・エディションで、計画、移行、インストール、および構成を行う場合の情報を提供します。また、この資料はサポートされている多数のクライアントのインストールおよびセットアップについても説明します。	GC88-8520 db2c6x70	db2c6
DB2 コネクト エンタープライズ・エディション (UNIX 版) 概説およびインストール	UNIX ベースのプラットフォームでの DB2 コネクト エンタープライズ・エディションの計画、移行、インストール、構成、およびタスクに関する情報を提供します。また、この資料はサポートされている多数のクライアントのインストールおよびセットアップについても説明します。	GC88-8519 db2cyx70	db2cy

表 57. DB2 情報 (続き)

資料名	説明	資料番号 PDF ファイル名	HTML ディレクトリー
DB2 コネクト パーソナル・エディション 概説およびインストール	OS/2 および Windows 32 ビット オペレーティング・システムの DB2 コネクト パーソナル・エディションで、計画、移行、インストール、および構成を行う場合のタスク情報を提供します。また、この資料はサポートされているすべてのクライアントのインストールおよびセットアップについても説明します。	GC88-8533 db2c1x70	db2c1
DB2 コネクト パーソナル・エディション (Linux 版) 概説およびインストール	サポートされる Linux 配布プログラムの DB2 コネクト パーソナル・エディションで、計画、インストール、移行、および構成を行う場合の情報を提供します。	GC88-8528 db2c4x70	db2c4
DB2 データ・リンク・マネージャー 概説およびインストール	AIX および Windows 32 ビット オペレーティング・システムの DB2 データ・リンク・マネージャーで、計画、インストール、構成を行う場合の情報を提供します。	GC88-8532 db2z6x70	db2z6
DB2 エンタープライズ拡張エディション (UNIX 版) 概説およびインストール	UNIX ベースのプラットフォームでの DB2 エンタープライズ拡張エディションの計画、インストール、および構成に関する情報を提供します。また、この資料はサポートされている多数のクライアントのインストールおよびセットアップについても説明します。	GC88-8530 db2v3x70	db2v3
DB2 エンタープライズ拡張エディション (Windows 版) 概説およびインストール	Windows 32 ビット オペレーティング・システムの DB2 エンタープライズ拡張エディションで、計画、インストール、および構成を行う場合の情報を提供します。また、この資料はサポートされている多数のクライアントのインストールおよびセットアップについても説明します。	GC88-8529 db2v6x70	db2v6

表 57. DB2 情報 (続き)

資料名	説明	資料番号 PDF ファイル名	HTML ディレクトリー
DB2 ユニバーサル・データベース (OS/2 版) 概説およびインストール	OS/2 オペレーティング・システムでの DB2 ユニバーサル・データベースの計画、インストール、移行、および構成に関する情報を提供します。また、この資料はサポートされている多数のクライアントのインストールおよびセットアップについても説明します。	GC88-8534 db2i2x70	db2i2
DB2 ユニバーサル・データベース (UNIX 版) 概説およびインストール	UNIX ベースのプラットフォームでの DB2 ユニバーサル・データベースの計画、インストール、移行、および構成に関する情報を提供します。 また、この資料はサポートされている多数のクライアントのインストールおよびセットアップについても説明します。	GC88-8536 db2ixx70	db2ix
DB2 ユニバーサル・データベース (Windows 版) 概説およびインストール	Windows 32 ビット・オペレーティング・システムの DB2 ユニバーサル・データベースで、計画、インストール、移行、および構成を行う場合の情報を提供します。また、この資料はサポートされている多数のクライアントのインストールおよびセットアップについても説明します。	GC88-8537 db2i6x70	db2i6
DB2 パーソナル・エディション 概説およびインストール	OS/2 および Windows 32 ビット・オペレーティング・システム版の DB2 ユニバーサル・データベース パーソナル・エディションで、計画、インストール、移行、および構成を行う場合の情報を提供します。	GC88-8535 db2i1x70	db2i1
DB2 パーソナル・エディション (Linux 版) 概説およびインストール	サポートされる Linux 配布プログラムの DB2 ユニバーサル・データベース パーソナル・エディションで、計画、インストール、移行、および構成を行う場合の情報を提供します。	GC88-8538 db2i4x70	db2i4
DB2 クエリー・パトローラー インストールの手引き	DB2 クエリー・パトローラーのインストール情報を提供します。	GC88-8526 db2iwx70	db2iw

表 57. DB2 情報 (続き)

資料名	説明	資料番号 PDF ファイル名	HTML ディレクトリー
ウェアハウス・マネージャー インストールの手引き	ウェアハウス・エージェント、ウェアハウス・トランスフォーマー、および情報カタログ・マネージャーのインストール情報を提供します。	GC88-8572 db2idx70	db2id
プラットフォーム共通のサンプル・プログラム (HTML 形式)			
サンプル・プログラム (HTML)	DB2 のサポートするすべてのプラットフォームでのプログラム言語用に、サンプル・プログラム (HTML 形式) を提供します。これらのサンプル・プログラムは、参照用としてのみ提供されています。サンプルは、すべてのプログラミング言語で利用できるわけではありません。 HTML サンプルが利用できるのは、 DB2 アプリケーション開発クライアントがインストールされている場合だけです。 プログラムの詳細については、*アプリケーション構築の手引き* を参照してください。	資料番号なし	db2hs
リリース情報			
DB2 コネクト リリース情報	DB2 コネクトの資料には含められなかった最新の情報が収録されています。	注 #2 を参照してください。	db2cr
DB2 インストール情報	DB2 ブックには含められなかったインストールに関する最新の情報が収録されています。	製品 CD-ROM からのみ利用できます。	
DB2 リリース情報	DB2 ブックには含められなかった DB2 製品とその機能に関する最新の情報が収録されています。	注 #2 を参照してください。	db2ir

注:

1. ファイル名の 6 桁目の文字 *x* は、その資料の言語を表します。たとえば、ファイル名 `db2d0e70` は、*管理の手引き* の英語版であることを示し、ファイル名 `db2d0f70` は同じ資料のフランス語版を示します。資料の言語を表すためにファイル名の 6 桁目で使用されている文字は以下のとおりです。

言語	ID
ブラジル・ポルトガル語	b
ブルガリア語	u
チェコ語	x
デンマーク語	d
オランダ語	q
英語	e
フィンランド語	y
フランス語	f
ドイツ語	g
ギリシャ語	a
ハンガリー語	h
イタリア語	i
日本語	j
韓国語	k
ノルウェー語	n
ポーランド語	p
ポルトガル語	v
ロシア語	r
簡体字中国語	c
スロベニア語	l
スペイン語	z
スウェーデン語	s
繁体字中国語	t
トルコ語	m

2. DB2 ブックには含められなかった最新の情報が、『リリース情報』で HTML 形式および ASCII ファイルとして利用できます。 HTML 版は、インフォメーション・センターおよび製品 CD-ROM からご利用になれます。 ASCII ファイルの参照方法:
   - UNIX ベースのプラットフォームでは、ファイル Release.Notes を参照してください。このファイルは DB2DIR/Readme/*%L* ディレクトリーにあります。ここで *%L* は地域名を、 `DB2DIR` は以下のものを表します。
     – /usr/lpp/db2_07_01 (AIX の場合)
     – /opt/IBMdb2/V7.1 (HP-UX、DYNIX/ptx、Solaris、および Silicon Graphics IRIX の場合)
     – /usr/IBMdb2/V7.1 (Linux の場合)
   - これ以外のプラットフォームでは、ファイル RELEASE.TXT を参照してください。このファイルは、製品がインストールされているディレクトリーにあります。 OS/2 プラットフォームでは、 **IBM DB2** フォルダーをダブルクリックし、 **Release Notes** アイコンをダブルクリックすることもできます。

PDF 資料の印刷

資料のハードコピー版が必要な場合、 DB2 の資料 CD-ROM にある PDF ファイル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することができます。ライブラリー内の各資料のファイルについては、835ページの表57 を参照してください。

Adobe Acrobat Reader の最新版は、Adobe の Web サイト http://www.adobe.co.jp/ から入手できます。

PDF ファイルは、DB2 の資料 CD-ROM に収録されており、ファイル拡張子 PDF が付いています。 PDF ファイルにアクセスするには以下のようにします。

1. DB2 の資料 CD-ROM を挿入します。 UNIX ベースのプラットフォームの場合は、DB2 資料 CD-ROM をマウントします。マウントの手順については、*概説およびインストール* を参照してください。
2. Acrobat Reader を起動します。
3. 以下に示すいずれかの位置から必要な PDF ファイルを開きます。
   - OS/2 および Windows プラットフォームでは:

     x:¥doc¥*language* ディレクトリー。ここで、*x* は CD-ROM ドライブを、 *language* は 2 桁の言語を表す国コード (たとえば、EN は英語) を示します。
   - UNIX ベースのプラットフォームでは:

     CD-ROM の */cdrom*/doc/*%L* ディレクトリー。ここで、*/cdrom* は CD-ROM のマウント・ポイントを、 *%L* は地域名を表します。

さらに、PDF ファイルを CD-ROM からローカル・ドライブまたはネットワーク・ドライブにコピーし、そこから参照することもできます。

印刷資料の注文方法

ハードコピー版の DB2 ブックは、個別に注文することができます。資料を注文するには、IBM 承認の販売業者または営業担当員に連絡してください。

DB2 オンライン文書

オンライン・ヘルプへのアクセス

すべての DB2 コンポーネントで、オンライン・ヘルプを利用できます。以下の表に、さまざまな種類のヘルプを示します。

ヘルプの種類	内容	利用方法
コマンド・ヘルプ	コマンド行プロセッサーのコマンド構文について説明します。	コマンド行プロセッサーの対話モードから、次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 コマンドのヘルプが表示されます。
クライアント構成アシスタントのヘルプ コマンド・センターのヘルプ コントロール・センターのヘルプ データウェアハウスセンターのヘルプ イベント・アナライザーのヘルプ 情報カタログ・マネージャーのヘルプ サテライト管理センターのヘルプ スクリプト・センターのヘルプ	そのウィンドウまたはノートブックで実行できるタスクについて説明します。このヘルプは、知っておく必要のある概説および前提条件に関する情報を含みます。また、ウィンドウやノートブックの制御の使用方法を示します。	ウィンドウまたはノートブックから、**「ヘルプ (Help)」**プッシュボタンをクリックするか、または **F1** キーを押します。

ヘルプの種類	内容	利用方法

メッセージ・ヘルプ	メッセージの原因、および取るべき処置を説明します。	コマンド行プロセッサーの対話モードから、次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。
		一度に 1 画面分のメッセージ・ヘルプを表示させるには、次のように入力します。
		`? XXXnnnnn | more`
		メッセージ・ヘルプをファイルに保管するには、次のように入力します。
		`? XXXnnnnn > filename.ext`
		ここで、*filename.ext* はメッセージ・ヘルプを保管するファイルを表します。

SQL ヘルプ	SQL ステートメントの構文について説明します。	コマンド行プロセッサーの対話モードから、次のように入力します。 `help statement` ここで、*statement* は SQL ステートメントを表します。 たとえば、help SELECT と入力すると、 SELECT ステートメントのヘルプが表示されます。 **注:** UNIX ベースのプラットフォームでは、SQL ヘルプを利用できません。
SQLSTATE ヘルプ	SQL 状態およびクラス・コードについて説明します。	コマンド行プロセッサーの対話モードから、次のように入力します。 `? sqlstate or ? class code` ここで、*sqlstate* は有効な 5 桁の SQL 状態を、*class code* は SQL 状態の最初の 2 桁を表します。 たとえば、`? 08003` によって SQL 状態 08003 のヘルプが表示され、 `? 08` によってクラス・コード 08 のヘルプが表示されます。

オンライン情報の表示

この製品に付属のブックは、ハイパーテキスト・マークアップ言語 (HTML) ソフトコピー形式です。ソフトコピー形式では情報を検索または表示したり、ハイパーテキスト・リンクを利用して関連情報に移動したりすることができます。また、1 つの端末を超えてライブラリーを容易に共用することができます。

オンライン・ブックやサンプル・プログラムは、 HTML バージョン 3.2 仕様に準拠するすべてのブラウザーを使って表示できます。

オンライン・ブックまたはサンプル・プログラムは、次のようにして表示します。

- DB2 管理ツールを実行している場合、インフォメーション・センターを使用します。
- ブラウザーで、**「ファイル (File)」 —> 「ページを開く (Open Page)」** をクリックします。次のようなページを開いて、DB2 情報に関する説明とリンクを表示してください。
  - UNIX ベースのプラットフォームでは、以下のページを開きます。

    ```
    INSTHOME/sqllib/doc/%L/html/index.htm
    ```

    ここで *%L* はロケール名です。
  - その他のプラットフォームでは、以下のページを開きます。

    ```
    sqllib¥doc¥html¥index.htm
    ```

    パスは DB2 がインストールされているドライブです。

    インフォメーション・センターをインストールしていない場合、 **DB2 Information** アイコンをダブルクリックしてページを開くことができます。このアイコンは、ご使用のシステムに応じて、製品のメイン・フォルダー内または Windows 「スタート」メニューにあります。

Netscape ブラウザーのインストール

システムに Web ブラウザーがインストールされていない場合、製品の箱の中にある Netscape CD-ROM から Netscape をインストールすることができます。インストールに関する詳細な説明については、以下を参照してください。

1. Netscape CD-ROM を挿入します。
2. UNIX ベースのプラットフォームでは、CD-ROM をマウントします。マウントの手順については、*概説およびインストール* を参照してください。
3. インストールの手順については、 CDNAV*nn*.txt ファイルを参照します。ここで、*nn* は 2 桁の言語 ID を表します。ファイルは CD-ROM のルート・ディレクトリーにあります。

インフォメーション・センターを使用した情報へのアクセス

インフォメーション・センターを使用すると、DB2 製品情報にす早くアクセスすることができます。インフォメーション・センターは、DB2 管理ツールを使用できるすべてのプラットフォームで利用できます。

インフォメーション・センターは「インフォメーション・センター (Information Center)」アイコンをダブルクリックすることによってオープンできます。このアイコンのある場所はシステムによって異なります。メイン・プロダクト・フォルダーか Windows の**「スタート」**メニューのどちらかです。

Windows プラットフォームの DB2 では、ツールバーおよび**ヘルプ**・メニューを使用して、インフォメーション・センターにアクセスすることもできます。

インフォメーション・センターは 6 種類の情報を提供します。適切なタブをクリックすると、種類ごとに提供されているトピックが表示されます。

タスク (Tasks) DB2 を使用して実行できる主要なタスク。

参照 (Reference)
DB2 参照情報 (キーワード、コマンド、API など)。

ブック (Books) DB2 ブック。

トラブルシューティング (Troubleshooting)
エラー・メッセージのカテゴリーと、メッセージに対するリカバリー処置。

サンプル・プログラム (Sample Programs)
DB2 アプリケーション開発クライアントに付属のサンプル・プログラム。 DB2 アプリケーション開発クライアントをインストールしていない場合、このタブは表示されません。

Web WWW 上にある DB2 情報。この情報にアクセスするには、ご使用のシステムから Web への接続が必要です。

リストから項目を 1 つ選択すると、インフォメーション・センターはビューアーを立ち上げて情報を表示します。選択した情報の種類に応じて、ビューアーはシステム・ヘルプ・ビューアー、エディター、または Web ブラウザーです。

インフォメーション・センターには検索機能が備わっており、リストを参照せずに特定のトピックを探すことができます。

テキストの全検索を行うには、インフォメーション・センター内のハイパーテキスト・リンク**「DB2 オンライン情報の検索 (Search DB2 Online Information)」**検索フォームに従います。

通常、HTML 検索サーバーは自動的に始動します。 HTML 情報の検索がうまくいかない場合は、以下の方法の 1 つを使用して、検索サーバーを始動しなければならない場合もあります。

Windows では
: **「スタート」**をクリックし、**「プログラム」—>「IBM DB2」—>「Information」—>「Start HTML Search Server」**を選択します。

OS/2 では
: **「DB2 (OS/2 版)」**フォルダーをダブルクリックして、**「Start HTML Search Server」**アイコンをダブルクリックします。

HTML 情報の検索でこの他の問題が発生した場合は、リリース情報を参照してください。

注: 検索機能は、Linux、DYNIX/ptx、および Silicon Graphics IRIX 環境では利用できません。

DB2 ウィザードの使用

ウィザードを使用すると、各タスクをステップごとに進めることによって、さまざまな管理タスクを遂行することができます。ウィザードは、コントロール・センターおよびクライアント構成アシスタントを通して使用できます。以下の表では、ウィザードとその目的をリストしています。

注: データベース作成、索引作成、マルチサイト更新の構成、およびパフォーマンス構成ウィザードは、区分データベース環境で使用できます。

ウィザード	内容	利用方法
データベース追加 *(Add Database)*	クライアント・ワークステーション上にデータベースのカタログを作成します。	クライアント構成アシスタントから、**「追加 (Add)」**をクリックします。
データベース・バックアップ *(Backup Database)*	バックアップ計画を決定、作成、およびスケジュールします。	「コントロール・センター (Control Center)」からバックアップするデータベースを右クリックし、**「バックアップ (Backup)」—>「ウィザードを使用するデータベース (Database Using Wizard)」**を選択します。

ウィザード	内容	利用方法
マルチサイト更新の構成 (Configure Multisite Update)	マルチサイト更新、分散トランザクション、または 2 フェーズ・コミットを構成します。	「コントロール・センター (Control Center)」から、**「データベース (Databases)」**フォルダーを右クリックして、**「マルチサイト更新 (Multisite Update)」**を選択します。
データベース作成 (Create Database)	データベースを作成し、いくつかの基本的な構成タスクを実行します。	「コントロール・センター (Control Center)」から、**「データベース (Databases)」**フォルダーを右クリックして、**「作成 (Create)」—>「ウィザードを使用するデータベース (Database Using Wizard)」**を選択します。
表作成 (Create Table)	基本的なデータ・タイプを選択して、表の基本キーを作成します。	「コントロール・センター (Control Center)」から、**「表 (Tables)」**アイコンを右クリックして、**「作成 (Create)」—>「ウィザードを使用する表 (Table Using Wizard)」**を選択します。
表スペース作成 (Create Table Space)	新しい表スペースを作成します。	「コントロール・センター (Control Center)」から、**「表スペース (Table Spaces)」**アイコンを右クリックして、**「作成 (Create)」—>「ウィザードを使用する表スペース (Table Space Using Wizard)」**を選択します。
索引作成 (Create Index)	すべての照会について、作成すべき索引および除去すべき索引を提案します。	「コントロール・センター (Control Center)」から、**「索引 (Index)」**アイコンを右クリックして、**「作成 (Create)」—>「ウィザードを使用する索引 (Index Using Wizard)」**を選択します。

ウィザード	内容	利用方法
パフォーマンス構成 (Performance Configuration)	ビジネス要件に適合するように構成パラメーターを更新して、データベースのパフォーマンスを調整します。	「コントロール・センター (Control Center)」から、調整したいデータベースを右クリックして、**「ウィザードを使用するパフォーマンスの構成 (Configure Performance Using Wizard)」**を選択します。 区分データベース環境では、「Database Partitions」視点から、調整したい最初のデータベース区画を右クリックして、**「ウィザードを使用するパフォーマンスの構成 (Configure Performance Using Wizard)」**を選択します。
データベース復元 (Restore Database)	障害の後、データベースを回復します。どのバックアップを使用し、どのログを再生するかの判別を支援します。	「コントロール・センター (Control Center)」から復元するデータベースを右クリックし、**「復元 (Restore)」—>「ウィザードを使用するデータベース (Database Using Wizard)」**を選択します。

文書サーバーのセットアップ

デフォルトでは、DB2 情報はローカル・システムにインストールされます。つまり、DB2 情報にアクセスする必要のある各担当者が同じファイルをインストールする必要があります。 DB2 情報を 1 か所に格納するには、次のようにします。

1. ¥sqllib¥doc¥html のすべてのファイルとサブディレクトリーを、ローカル・システムから Web サーバーにコピーします。各ブックには独自のサブディレクトリーがあり、そのブックを構成する必要な HTML および GIF ファイルが入っています。ディレクトリー構造は常に同じ状態に保つ必要があります。
2. Web サーバーを構成して、ファイルを新しい場所で検索するようにします。さらに詳しい情報については、 *インストールおよび構成 補足* の NetQuestion 付録を参照してください。
3. インフォメーション・センターの Java バージョンをご使用の場合は、すべての HTML ファイルのベース URL を指定できます。この URL はブックのリストに使用してください。
4. 資料ファイルが表示されるようになったなら、よく使うトピックにはブックマークを付けておいてください。ブックマークを付けるページは、たとえば以下のものがあります。
   - ブックのリスト

- 頻繁に使用されるブックの目次
- 頻繁に参照する情報 (たとえば、ALTER TABLE トピックなど)
- 検索フォーム

中央のマシンから DB2 ユニバーサル・データベース オンライン文書ファイルを提供する方法については、 *インストールおよび構成 補足* の NetQuestion 付録を参照してください。

オンライン情報の検索

HTML ファイルの情報を検索するには、以下の方法のどれか 1 つを使用してください。

- 最上部にある**「検索 (Search)」**をクリックします。検索フォームを使用して特定のトピックを見つけます。この機能は、 Linux、DYNIX/ptx、または Silicon Graphics IRIX 環境ではご利用になれません。
- 最上部にある**「索引 (Index)」**をクリックします。索引を使用して、ブック内の特定のトピックを見つけます。
- HTML 資料またはヘルプの目次あるいは索引を表示してから、 Web ブラウザーの検索機能を利用してブック内の特定のトピックを見つけます。
- Web ブラウザーのブックマーク機能を使用して、特定のトピックにす早く戻ります。
- インフォメーション・センターの検索機能を使用して、特定のトピックを検索します。詳しくは、849ページの『インフォメーション・センターを使用した情報へのアクセス』を参照してください。

付録G. 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、プログラムまたはサービスの操作性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。実施権の許諾については、下記の宛先に書面にてご照会ください。

〒 106-0032 東京都港区六本木 3丁目 2-31
AP 事業所
IBM World Trade Asia Corporation
Intellectual Property Law & 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。

本書は定期的に見直され、必要な変更 (たとえば、技術的に不適切な表現や誤植など) 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 Ltd.
Office of the Lab Director
1150 Eglinton Avenue East
Toronto, Ontario
M3C 1H7
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・プログラムのすべての部分、またはすべての派生した創作物には、次のように、著作権表示を入れていただく必要があります。

© (お客様の会社名) (西暦年). このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。 © Copyright IBM Corp. _年を入れる_. All Rights Reserved.

商標

アスタリスク (*) 付きの以下の用語は、IBM Corporation の商標です。

ACF/VTAM	IBM
AISPO	IMS
AIX	IMS/ESA
AIX/6000	LAN DistanceMVS
AIXwindows	MVS/ESA
AnyNet	MVS/XA
APPN	Net.Data
AS/400	OS/2
BookManager	OS/390
CICS	OS/400
C Set++	PowerPC
C/370	QBIC
DATABASE 2	QMF
DataHub	RACF
DataJoiner	RISC System/6000
DataPropagator	RS/6000
DataRefresher	S/370
DB2	SP
DB2 Connect	SQL/DS
DB2 Extenders	SQL/400
DB2 OLAP Server	System/370
DB2 Universal Database	System/390
Distributed Relational	SystemView
Database Architecture	VisualAge
DRDA	VM/ESA
eNetwork	VSE/ESA
Extended Services	VTAM
FFST	WebExplorer
First Failure Support Technology	WIN-OS/2

以下は、それぞれ各社の商標または登録商標です。

Tivoli および NetView は Tivoli Systems Inc. の商標です。

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。

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。

UNIX は、The Open Group がライセンスしている米国およびその他の国における登録商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標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］

アプリケーション環境、プログラミングの 12
アプリケーション設計
  エラー処理の指針 123
  エンド・ユーザー要求の保管 160
  カーソル処理の考慮事項 88
  可変リスト・ステートメントの処理 160
  コード・ポイント、特殊文字の 526
  サンプル・プログラム 111
  指針 24
  十分な SQLVAR エンティティーの宣言 150
  静的 SQL の使用の利点 68
  データ値の制御についての考慮事項 28
  データの受け渡しの指針 158
  データの関連 30
  データベース値の受け取り 82
  データへのアクセス 25
  動的 SQL キャッシュ 68
  動的 SQL の使用の概説 133
  入力 SQLDA ストアード・プロシージャーの例 808
  入力 SQLDA プロシージャーの例 802
  バインド 51
  パッケージの名前変更 59
  パラメーター・マーカーの使用 168
  表関数に関する考慮事項 458
  プリコンパイルとバインド 51
アプリケーション設計 *(続き)*
  変数のないステートメントの実行 134
  文字変換、ストアード・プロシージャーにおける 527
  文字変換、SQL ステートメントの 525
  文字変換に関する考慮事項 525
  ロジック、アプリケーションにおける 32
  2 度目のデータ検索 109
  2 バイト文字のサポート (DBCS) 526
  COBOL の日本語および中国語 (繁体字) EUC に関する考慮事項 721
  COBOL の要件、インクルード・ファイル 702
  DB2 アプリケーションのコーディングの概説 12
  OLE オートメーション UDF 441
  REXX の要件、ルーチンの登録における 740
  SELECT ステートメントの記述 153
  SQLDA 構造の作成の指針 155
  UDF 内での LOB ロケーターの使用 459
アプリケーションのロジック
  データ値の制御についての考慮事項 30
  データの関連についての考慮事項 31
アプリケーションのロジックについての考慮事項
  ストアード・プロシージャー 32
  トリガー 32
  ユーザー定義関数 32
アプリケーション・ドメインおよびオブジェクト指向 289
アプリケーション・プログラミング・インターフェース (API)
  概説 40
  制約事項、XA 環境における 566
  設定、スレッド間のコンテキスト
    sqleAttachToCtx() 558
    sqleBeginCtx() 558
    sqleDetachFromCtx() 558
    sqleEndCtx() 558
    sqleGetCurrentCtx() 558
    sqleInterruptCtx() 558
    sqleSetTypeCtx() 558
  タイプ 40
  の使用 40
  JDBC アプリケーションの 693
  REXX の構文 752
暗黙の接続 814
移植、アプリケーションの 811
移植性 178
一時表 189
一貫した振る舞いおよび特殊タイプ 295
一貫性
  データの 20
一貫性、データの 20
移動、ファイル参照変数を使用してのラージ・オブジェクトの 365
インクルード・ファイル
  検索、COBOL での 706
  検索、FORTRAN での 727
  COBOL 要件 702
  C/C++ での検索 614
  C/C++ の SQLADEF 611
  C/C++ の SQLAPREP 611
  C/C++ の SQLCA 611
  C/C++ の SQLCLI 611
  C/C++ の SQLCLI1 611
  C/C++ の SQLCODES 612
  C/C++ の SQLDA 612
  C/C++ の SQLE819A 612

© Copyright IBM Corp. 1993, 2001

インクルード・ファイル *(続き)*
C/C++ の SQLE819B 612
C/C++ の SQLE850A 612
C/C++ の SQLE850B 612
C/C++ の SQLE932A 613
C/C++ の SQLE932B 613
C/C++ の SQLEAU 612
C/C++ の SQLENV 612
C/C++ の SQLEXT 612
C/C++ の SQLJACB 613
C/C++ の SQLMON 613
C/C++ の SQLSTATE 613
C/C++ の SQLSYSTM 613
C/C++ の SQLUDF 613
C/C++ の SQLUTIL 613
C/C++ の SQLUV 613
C/C++ の SQLUVEND 613
C/C++ の SQLXA 614
C/C++ のインクルード・ファイル 611
C/C++ 要件 611
FORTRAN 要件 724
SQL
COBOL 702
FORTRAN 724
SQL1252A
COBOL 704
FORTRAN 726
SQL1252B
COBOL 704
FORTRAN 726
SQLAPREP
COBOL 702
FORTRAN 724
SQLCA
COBOL 702
FORTRAN 724
SQLCA_92
COBOL 702
FORTRAN 725
SQLCA_CN
FORTRAN 724
SQLCA_CS
FORTRAN 724
SQLCODES
COBOL 703

インクルード・ファイル *(続き)*
SQLCODES *(続き)*
FORTRAN 725
SQLDA
COBOL 703
FORTRAN 725
SQLDACT
FORTRAN 725
SQLE819A
COBOL 703
FORTRAN 725
SQLE819B
COBOL 703
FORTRAN 725
SQLE850A
COBOL 703
FORTRAN 726
SQLE850B
COBOL 704
FORTRAN 726
SQLE932A
COBOL 704
FORTRAN 726
SQLE932B
COBOL 704
FORTRAN 726
SQLEAU
COBOL 703
FORTRAN 725
SQLENV
COBOL 703
FORTRAN 725
SQLETSD
COBOL 703
SQLMON
COBOL 704
FORTRAN 726
SQLMONCT
COBOL 704
SQLSTATE
COBOL 704
FORTRAN 727
SQLUTBCQ
COBOL 704
SQLUTBSQ
COBOL 705

インクルード・ファイル *(続き)*
SQLUTIL
COBOL 705
FORTRAN 727
インスタンス生成の可否 317
インスタンスの保管、オブジェクト指向データ・タイプの、 289
インストール
Netscape ブラウザー 848
インディケーター表の COBOL サポート 716
インフィックス表記および UDF 405
インフォメーション・センター 849
インプリメント、UDF の 394
ウィザード
索引 851
タスクを遂行する 850
データベース作成 851
データベース追加 850, 851, 852
データベース復元 852
バックアップ・データベース 850
パフォーマンス構成 851
表作成 851
表スペース作成 851
マルチサイト更新の構成 850
売上の例、CREATE TABLE を使用した 297
エラー検出、バッファー化挿入における 574
エラー処理
インクルード・ファイル
COBOL での 702, 703, 704
FORTRAN の 725, 727
インクルード・ファイル、C/C++ の 613
延期状態のアプリケーションの 585
概説 122
考慮事項、区分データベース環境での 583
識別、エラーを戻した区画の 585
プリコンパイル時 56
報告 584

エラー処理 *(続き)*
マージされた複数の SQLCA 構造 584
リセット 18
ループ状態のアプリケーションの 585
C/C++ 言語プリコンパイラー 614
SQLCA 構造 584
SQLCA による 17
SQLCODE 584
WHENEVER ステートメント 17
WHENEVER ステートメントの使用 123
エラー・コード 17
エラー・メッセージ
エラー状態フラグ 122
警告状態フラグ 122
タイム・スタンプ、プリコンパイル時 63
例外状態フラグ 122
SQLCA 構造 122
SQLSTATE 122
SQLWARN 構造 122
延期状態のアプリケーション
診断 585
オートメーション・サーバー、OLE の 442
オープン状態、バッファー化挿入の 575
応募の書式の例、CREATE TABLE を使用した 297
オブジェクト ID
自動生成 333
に対する制約の作成 334
の表示タイプの選択 322
オブジェクト ID 列 309, 310
命名 318
オブジェクト関連
アプリケーション・ドメインおよびオブジェクト指向 289
制約機構 289
データ・タイプ 289
定義 289
トリガー 289
オブジェクト関連 *(続き)*
DB2 オブジェクト拡張を使用する理由 289
LOB 289
UDT および UDF 289
オブジェクト指向 COBOL の制約事項 722
オブジェクト指向および UDF 390
オブジェクト指向の拡張および特殊タイプ 295
オブジェクトのリンクと埋め込み (OLE) 441
オブジェクト・インスタンス
OLE オートメーション UDF の 443
重みの定義 518
オンライン情報
検索 853
表示 848
オンライン・ヘルプ 845

[カ行]

カーソル
更新可能 99
作業単位の完了 88
サンプル・プログラム 99
処理、サンプル・プログラム 90, 140
処理の要約 87
宣言 88
動的 816
動的 SQL での処理 138
表の末尾に置く 111
複数行の取り出し 87
未確定 99, 815
未確定カーソル 815
命名、REXX での 742
命名および定義 87
読み取り専用 87, 98
読み取り専用のアプリケーション要件 88
CLI での使用 177
COMMIT ステートメントの発行 88
FOR FETCH ONLY 98
カーソル *(続き)*
SQLDA 構造での処理 154
WITH HOLD による宣言 88
カーソル、WITH HOLD と宣言された
X/Open XA インターフェース 565
カーソル固定 818
カーソルの使用法、REXX における 750
解決、問題の
数値変換のオーバーフロー 819
開始、トランザクションの 20
解除、接続の
CMS アプリケーション 21
DB2 への 21
階層
構造型 307
外部キー 818
カウンター OLE オートメーション UDF オブジェクトの例、BASIC での 491
カウンター OLE オートメーション UDF オブジェクトの例、C++ による 492
カウンターの例、UDF の 478
拡張 UNIX コード (EUC)
アプリケーションでの拡張 543
拡張のサンプル 544
規則、ストリング変換に関する 547
クライアント・ベースのパラメーターの検証 544
グラフィック定数 539
グラフィック・データの処理 539
固定長または可変長データ・タイプ 546
異なるコード・ページ 540
混合コード・セット環境 540
混合しているコード・ページ 537
サーバーでの拡張 543
使用、DESCRIBE ステートメントの 545
照合に関する考慮事項 540

拡張 UNIX コード (EUC) *(続き)*
ストアード・プロシージャーに関する考慮事項 539
中国語 (繁体字) コード・セット 535, 538
中国語 (繁体字) に関する考慮事項 538
中国語 (繁体字)、C/C++ での 643
日本語、C/C++ での 643
日本語および中国語 (繁体字)
COBOL に関する考慮事項 721
FORTRAN に関する考慮事項 737
日本語コード・セット 535, 538
文字ストリング長のオーバーフロー 547
文字セット 533
文字変換、ストアード・プロシージャーにおける 548
文字変換によるオーバーフロー 547
2 バイト・コード・ページ 537
DBCLOB ファイルに関する考慮事項 539
REXX における中国語 (繁体字) 756
REXX における日本語 756
UDF に関する考慮事項 539
拡張、AS/400 サーバー上におけるデータの 813
拡張性および特殊タイプ 295
拡張動的 SQL ステートメント
DB2 コネクトでサポートされていない 826
確定カーソル 815
カスケード 818
カスケード、トリガーの 510
各国語サポート (NLS)
コード・ページ 529
考慮事項 517
混合バイト・データ 813
文字変換 529
活動化時間とトリガー 503
カプセル化および特殊タイプ 295
可変長ストリング 813
環境 API
インクルード・ファイル、C/C++ の 612
インクルード・ファイル、FORTRAN の 725
COBOL のインクルード・ファイル 703
環境、プログラミングの 12
環境ハンドル 177
関数
SQL トリガー・ステートメントに対する 509
関数参照
UDF の要約 405
関数選択アルゴリズムおよび UDF 393
関数ディレクトリー 209
関数テンプレート 600
関数パスおよび UDF 393
関数マッピング
オプション 601
CREATE FUNCTION MAPPING ステートメント 600, 601
関数名、UDF に渡される 415
関数呼び出しの例 402
完全修飾名 451
簡単な保守、トリガーを使った 500
完了コード 17
キー
外部 818
基本 818
記号置換、C/C++ 言語制限 629
記述子ハンドル 177
規則、本書で使用されている 8
規則、ラージ・オブジェクトの操作を制御する 289
機能
参照するための構文 401
スカラー関数 394
総計関数 394
表関数 394
列関数 394
基本キー 818
逆参照演算子 315
を使用した照会 329
キャスト
UDF 407
キャスト機能 391
行
カーソルによる複数行の取り出し 98
順序の制御 109
トリガーにより影響される行のセット 503
取り出し、1 行 69
表中での位置付け順序 111
1 行選択、SELECT INTO ステートメントを使用しての 70
SQLDA を用いた検索 154
行のブロック化
パフォーマンス向上のためのカスタマイズ 567
共用メモリー・サイズ、UDF のための 467
強力タイピングおよび特殊タイプ 298
行レベルのロック 818
切り捨て
標識変数 83
ホスト変数 83
国別コード
SQLCA の SQLERRMC フィールドの 815
区分データベース環境
エラー処理考慮事項 583
エラー発生時の識別 585
重大エラー考慮事項 583
パフォーマンスの向上 569
組み込み SQL
データへのアクセスについての考慮事項 26
組み込み SQL ステートメント
概説 49
規則、COBOL の場合の 705
規則、C/C++ における 615
規則、FORTRAN における 728
構文規則 50
注釈の規則 615, 705, 728
ホスト変数の参照 81
例 50

クライアント / サーバー
コード・ページ変換 529
クライアント・アプリケーション
ストアード・プロシージャーの実行 208
クライアント・ベースのパラメーターの検証
拡張 UNIX コードに関する考慮事項 544
クラス
データ・メンバー、C/C++ でのホスト変数としての 636
グラフィカル・オブジェクト
考慮事項、Java に関する 693
グラフィック定数
中国語 (繁体字) コード・セット 539
日本語コード・セット 539
グラフィック・ストリング
文字変換 532
グラフィック・データ
中国語 (繁体字) コード・セット 535, 539
日本語コード・セット 535, 539
グラフィック・データ・タイプ
選択 639
グラフィック・ホスト変数
COBOL 710
C/C++ 622
クリティカル・セクション 559
クリティカル・セクションのルーチン、複数のスレッドにおける指針 559
グループ、行の
バッファー化挿入における 574
クローズ、バッファー化挿入 572
クローズ状態、バッファー化挿入の 575
グローバルな実施、トリガーを使った業務規則の 500
警告、トリガーによりサポートされる 500
警告メッセージ、切り捨て 83
計算および定義の例、UDF の 399
継承
ONLY 文節を使用した制御 319
結果コード 17
結果セット
ストアード・プロシージャーからの 246
SQL プロシージャーからの戻り 270
言語 ID
ブック 843
検索
オンライン情報 849, 853
検索、インクルード・ファイルの
COBOL 706
C/C++ 614
FORTRAN 727
検索、データの
更新 111
上方スクロールの技法 108
2 度 109
コーディネーター・ノード
バッファー化挿入を使用しない振る舞い 573
コーディング、Java UDF の 437
コード・セット
SQLCA の SQLERRMC フィールドの 815
コード・ページ
各国語サポート (NLS) 529
サポートされる Windows コード・ページ 523
処理、アプリケーションでの拡張の 543
処理、異なる場合の 532, 540
処理、サーバーでの拡張の 543
ストアード・プロシージャーに関する考慮事項 241
ストレージの割り振り、異なる場合の 541
バインドの考慮事項 60
文字変換 529
ロケール
アプリケーション中の導出 524
DB2 のロケールの導出法 524
DB2CODEPAGE レジストリー変数 523
コード・ページ *(続き)*
SQLCA の SQLERRMC フィールドの 815
コード・ポイント 518
コード・ポイントの定義 518
コール・レベル・インターフェース (CLI)
概説 177
組み込み SQL と DB2 CLI の比較 177
使用する場合の利点 178, 180
更新、視点の 328
更新操作 501
構成パラメーター
LOCKTIMEOUT 561
構造化照会言語 (SQL)
ステートメント、要約 37
表、サポートされるステートメント 759
構造型
インスタンス生成可能タイプ 317
インスタンスの作成 311
インスタンスの単一の値としての検索 338
インスタンスの列への挿入 337
インスタンスをクライアント・アプリケーションに渡す 350
オブジェクトの列への保管 315
概説 306
階層 307
行オブジェクトの参照 309
行としての保管 313
継承 306
構造型属性の定義 336
サブタイプ属性の検索 339
スキーマ名の検索 330
静的タイプ 317
属性 306
属性値の検索 311
属性の更新 311, 338, 339
属性の列への挿入 336
タイプ階層内でのサブタイプへのアクセス 316
タイプ付き表の作成 313
タイプ名の検索 330

構造型 *(続き)*
ためのホスト変数の宣言 363
動的タイプ 317
とのインスタンスの比較 331
内部 ID の検索 330
に対するメソッドの呼び出し 312
についての情報を戻す 340
非インスタンス生成可能タイプ 317
表示タイプ 310
振る舞いの定義 312
保管 307
メソッドの呼び出し 337
利点 306
列への保管 335
constructor 関数 311
DESCRIBE ステートメント 363
MODE DB2SQL 文節 306
mutator メソッド 311
observer メソッド 311
tranform 関数を使用したサブタイプのバインドイン 359
構造定義、sqludf.h での 435
構文
行の切れを避けた組み込み SQL ステートメント 616
空白文字での組み込み SQL ステートメントの置換 616
組み込み SQL ステートメント
COBOL での 705
COBOL 内の注釈 705
FORTRAN での 728
FORTRAN における注釈 728
宣言セクション
COBOL での 708
FORTRAN での 729
文字ホスト変数 621
C/C++ セクションでの宣言 618
C/C++ での組み込み SQL ステートメント 615
C/C++ での組み込み SQL ステートメントのコメント 616
C/C++ のグラフィック・ホスト変数 622
構文 *(続き)*
REXX での SQL ステートメントの処理 741
REXX における組み込み SQL ステートメントの注釈 743
構文、関数を参照するための 401
効力範囲が指定された参照
参照保全との比較 325
考慮事項
アプリケーションのロジック、サーバーにおける 32
データ値の制御 28
データの関連の制御 30
データへのアクセス 25
DB2 アプリケーション設計 24
コストの例、UDT の 398
固定長または可変長データ・タイプ
拡張 UNIX コードに関する考慮事項 546
異なるコード・ページ 540
ストレージの割り振り 541
コマンド
EXCSQLSTT 825
FORCE 815
コマンド行プロセッサー 752
ユーティリティーのプロトタイプ化 44
固有キー違反
バッファー化挿入 575
混合コード・セット環境
アプリケーション設計 540
混合している拡張 UNIX コードの考慮事項 537
混合バイト・データ 813
コンテキスト
その間でのアプリケーションの従属関係 559
その間でのデータベースの従属関係 559
その間でのデッドロックを避ける 560
マルチスレッド DB2 アプリケーションでの設定 557
コンパイル 57
コンパイル、UDF の 394
コンパイル済みアプリケーション、パッケージの作成 54

[サ行]

サーバー・オプション 599
再開の例、CREATE DISTINCT TYPE を使用した 297
最終呼び出し、UDF に対する 418
再使用および UDF 390
最新情報 844
最大サイズの定義、ラージ・オブジェクト列の 366
再入可能な
ストアード・プロシージャー 245
UDF 456
再バインド
説明 64
REBIND PACKAGE コマンド 64
作業環境
セットアップ 40
テスト・データベース、作成のガイドライン 40
作業単位
カーソルに関する考慮事項 88
完了 88
分散 549
リモート 549
索引ウィザード 851
索引拡張 290
作成
Java UDF 436
Java ストアード・プロシージャー 683
OLE オートメーション UDF 442
作成可能な単独使用 OLE オートメーション・サーバー 447
作成可能な複数使用 OLE オートメーション・サーバー 447
作成者属性
パッケージ 816
サブタイプ 307

サブタイプ *(続き)*
のための transform 関数の作成 355
OUTER を使用して属性を戻す 332
tranform 関数を使用したバインドイン 359
算術計算エラー
UDF における 413
参照
逆参照演算子 315
参照制約との比較 315
を使用した関係の定義 314
参照タイプ
キャスト 310
定義 309
の表示タイプの選択 322
比較 310, 323
参照の効力範囲の指定 320
参照保全 818
効力範囲が指定された参照との比較 325
データの関連についての考慮事項 30
参照保全制約
データ値の制御についての考慮事項 29
参照列
効力範囲の定義 320
タイプ付き視点での効力範囲の割り当て 327
サンプル・プログラム
アプリケーション・プログラミング・インターフェース (API) 765
組み込み SQL ステートメント 765
場所 765
プラットフォーム共通の 843
HTML 843
Java UDF 436
Java ストアード・プロシージャー 683
シーケンス、記述 185
識別順序 518
識別列 184
シグナル・ハンドラー
インストール、サンプル・プログラム 111
SQL ステートメントでの 124
シグニチャー、2 つの関数および同様の 393
自己参照タイプ付き表 324
自己参照表 818
指針
アプリケーションのロジック、サーバーにおける 32
データ値の制御 28
データの関連の制御 30
データへのアクセス 25
DB2 アプリケーション設計 24
システム構成パラメーター、共用メモリー・サイズのための 467
システム・カタログ
視点含意の除去 328
使用 819
システム・カタログ視点
ユーティリティーのプロトタイプ化 45
実行要件、REXX の 751
視点
システム・カタログ 819
システム・カタログの含意の除去 328
除去 328
制約事項 328
データ値の制御についての考慮事項 30
データ・ソース視点 588
変更 328
シフトアウト文字とシフトイン文字 813
集合 ID 属性
パッケージ 816
DB2 ユニバーサル・データベース (AS/400 版) 816
収集 816
修飾およびメンバー演算子、C/C++ における 638
修飾子属性
さまざまなプラットフォーム 816
修飾子属性 *(続き)*
パッケージ 816
修飾されない関数参照の例 404
修飾されない参照 393
重大エラー
考慮事項、区分データベース環境での 583
柔軟性および特殊タイプ 295
終了、暗黙にトランザクションを 22
終了、トランザクションの 21
終了、Java メソッドの 438
出力ファイルの拡張子、C/C++ 言語 610
使用
Java UDF 436
Java ストアード・プロシージャー 683
照会製品、データのアクセスについての考慮事項 28
障害追及
ストアード・プロシージャー 257
Visual Studio の使用 257
条件ハンドラー
概説 263
例 265
SQL プロシージャー 264
照合
中国語 (繁体字) コード・セット 540
日本語コード・セット 540
照合順序
インクルード・ファイル
COBOL での 703
FORTRAN での 725
大文字小文字に無関係な比較 519
概説 518
コード・ポイント 518
識別順序 518
指定 522
ソート順序の例 521
そのサンプル 523
マルチバイト文字 518
文字比較での使用 519

照合順序 *(続き)*
C/C++ のインクルード・ファイル 612
EBCDIC と ASCII 818
EBCDIC と ASCII のソート順序の例 521
EBCDIC バイナリー照合のシミュレート 827
使用の例、修飾された関数参照の 403
使用の例、UNION 形式での特殊タイプの 304
使用法、本書の 4
使用例、CLOB 値を使う作業のためのロケーターの 369
除去、視点の 328
所有者属性
パッケージ 816
処理、REXX での SQL ステートメントの 741
迅速なアプリケーション開発、トリガーを使った 500
診断メッセージ、UDF に渡される 416
スーパータイプ 307
数値データ・タイプ 813
数値変換のオーバーフロー 819
数値ホスト変数
COBOL 708
C/C++ 618
FORTRAN 730
据え置きの例、LOB 式の評価の 375
スカラー関数 394
呼び出しタイプ引き数の内容 418
スキーマ名および UDF 393
スクラッチパッド、UDF に渡される 411, 416
スクラッチパッドおよび UDF 438, 456
スクラッチパッドに関する考慮事項
OLE オートメーション UDF の 443
ステートメント
接続 814
ステートメント *(続き)*
呼び出し 820
ACQUIRE 825
BEGIN DECLARE SECTION 14
COMMIT 21
COMMIT WORK RELEASE 825
CONNECT 19
CREATE STORGROUP 812
CREATE TABLESPACE 812
DECLARE 825
DELETE 813
DESCRIBE 825
END DECLARE SECTION 14
GRANT 814
INCLUDE SQLCA 17
INSERT 813
LABEL ON 825
PREPARE 825
ROLLBACK 21, 815
SELECT 813
SET CURRENT 826
UPDATE 813
ステートメント・ハンドル 177
ストアード・プロシージャー
アプリケーションの障害追及 257
アプリケーションのロジックについての考慮事項 32
位置 209
一般的な 820
置く場所 209
概説 201, 208
許可されている SQL ステートメント 223
クライアント・アプリケーション 208
グラフィック・ホスト変数に関する考慮事項 242
結果セットの戻り 246
コード・ページに関する考慮事項 241
作成および使用、Java での 683
サポートされている言語 211
初期化、REXX 変数の 754
ストレージの割り振り 207
制約事項 226
ストアード・プロシージャー *(続き)*
体系 205
中国語 (繁体字) コード・セット 539
デバッグ 244
ストアード・プロシージャー・ビルダーの使用 685
登録、Java での 683
名前の多重定義 208
日本語コード・セット 539
入力 SQLDA ストアード・プロシージャーの例 808
入力 SQLDA プロシージャーの例 802
ネストされた 224
の CALL ステートメント 269
パス 209
パラメーター 209, 222, 223
パラメーター・モードの宣言 209
ホスト変数 207
マルチサイト更新に関する考慮事項 242
文字変換 527
文字変換、EUC 548
要件 205
呼び出し 207
利点 202
例 222
CALL ステートメント 207, 208
CONTAINS SQL 文節 223
CREATE PROCEDURE ステートメント 208
CREATE PROCEDURE による登録 208
C++ に関する考慮事項 241
db2dari 実行可能ファイル 226
DBINFO 構造の受け渡し 221
EXTERNAL 文節 209
FOR BIT DATA に関する考慮事項 241
LANGUAGE 文節 211
main 関数として作成される 212
NOT FENCED 244
OUT パラメーターのクライアント・プログラム 230

ストアード・プロシージャー *(続き)*
PARAMETER STYLE 文節 214
PROGRAM TYPE 文節 212
REXX アプリケーション 754
SQLDA および SQLCA 構造の使用 792
ストアード・プロシージャー (DB2DARI)
制約事項 791
データ構造の使用法 791
パラメーター変数
sqlda.n.SQLDAT 791
sqlda.n.SQLDATALEN 791
sqlda.n.SQLDATATYPE_NAME 791
sqlda.n.SQLIND 791
sqlda.n.SQLLEN 791
sqlda.n.SQLLONGLEN 791
sqlda.n.SQLNAME.data 791
sqlda.n.SQLNAME.length 791
sqlda.n.SQLTYPE 791
sqlda.SQLDABC 791
sqlda.SQLDAID 791
sqlda.SQLN 791
標識変数の使用 791
戻り値 793
呼び出し規則
パラメーター変換 791
SQL_API_FN 791
NOT FENCED 801
ストアード・プロシージャー・ビルダー
概説 822
環境設定 686
機能 822
デバッグ表 687
ストリング検索の例、UDT での 398
ストリング探索、BLOB での 397
ストリングの検索および UDF の定義の例 396
ストレージ
行を保持するための割り振り 154
十分な SQLVAR エンティティーの宣言 150
ストレージの割り振り、コード・ページが異なる場合 541
スナップショット・モニター
延期状態またはループ状態のアプリケーションの診断 585
制御情報、ラージ・オブジェクト・データにアクセスするための 366
制限
ストアード・プロシージャー (DB2DARI) 791
成功コード 17
整数除算演算子の例、UDF の 470
生成列 184
静的 SQL
概説 68
コーディング、データを検索および操作するステートメントの 77
構造型用の transform グループ 343
考慮事項 134
サンプル・プログラム 69
静的更新プログラムの例 111
動的 SQL との比較 134
プリコンパイル、利点 63
ホスト変数の使用 77
DB2 コネクト・サポート 811
静的タイプ 317
制約機構、ラージ・オブジェクトの 289
制約事項
バッファー化挿入の使用 576
COBOL での 701
C/C++ での 629
FORTRAN での 723
REXX での 740
UDF の 467
制約事項、DB2DARI ストアード・プロシージャーの 791
セクション番号 825
設計、DB2 アプリケーションの指針 24
接続
暗黙の接続 814
ヌル CONNECT 814
接続 *(続き)*
CONNECT RESET ステートメント 814
CONNECT TO ステートメント 814
接続、DB2 アプリケーション・プログラムへの 19
接続ハンドル 177
セットアップ、文書サーバーの 852
セットアップ、DB2 プログラムの 14
セマフォー 559
セマンティック上の振る舞い、ストアード・オブジェクトの 289
宣言
標識変数 82
ホスト変数の規則 77
宣言済み一時表 189
宣言セクション
ステートメントの規則 77
COBOL での 708, 720
C/C++ での 618, 648
db2dclgn による作成 80
FORTRAN での 729, 735
全選択
バッファー化挿入考慮事項 576
全選択考慮事項 576
前提条件、プログラミングの 12
ソース派生 UDF 300
ソース・ファイル
作成、概説 51
修正済みソース・ファイル、定義 55
ファイル名拡張子 55
要件 56
SQL ファイル拡張子 51
ソース・レベルのデバッガーおよび UDF 497
ソート、照合順序の指定 522
ソート順序
照合順序 818
定義 818
相違点、異なる DB2 製品間の 812
相違点、SQLCODE と SQLSTATE の 819
総計関数 394

操作、ラージ・オブジェクトの　289
挿入
　バッファー化挿入を使用しない　573
挿入の例、CLOB 列へのデータの　388
属性　306

[タ行]

ターゲット区画
　バッファー化挿入を使用しない振る舞い　573
第 1 呼び出し、UDF に対する　418
大幅な移動の DB2 検査　497
タイプ
　ROWID　814
タイプ修飾
　ストアード・プロシージャーにおける　241
　UDF における　468
タイプ修飾に関する考慮事項
　C++　610
タイプ述部
　を使用した戻されるタイプの制限　331
タイプ付き視点
　作成　325
　サブタイプ上での作成　325
　の中の参照列への効力範囲の割り当て　327
　の本体　326
　ルート・タイプ上での作成　325
タイプ付き視点の作成　325
タイプ付き表
　オブジェクト ID の挿入　322
　オブジェクト ID 列　318
　オブジェクトの挿入　320
　階層の位置の決定　319
　効力範囲の定義　320
　作成　318
　サブタイプ属性を戻す　332
　自己参照　324
　タイプ階層内でのサブタイプへのアクセス　314
　定義　313
タイプ付き表 *(続き)*
　での特権の制御　319
　の間の関係の定義　314, 323
　のデータの選択　328
　副表の作成　313
タイプ変換
　SQL タイプと OLE オートメーション・タイプ間の　443
タイプまたは引き数、UDF 内でのプロモーション　426
タイプ・マッピング　593
　除去の制限　327
　OLE オートメーション・タイプと BASIC タイプ　445
　OLE オートメーション・タイプと C++ タイプ　445
タイムアウト、ロックの　818
代用性　314, 316
代理関数　500
対話式 SQL
　処理、サンプル・プログラム　161
多重定義
　関数名　393
　ストアード・プロシージャー名　208
多重定義、SQLCA においての回避　18
逐次化、データ構造の　558
逐次化、SQL ステートメント実行の　557
中国語 (繁体字)
　拡張 UNIX コードに関する考慮事項　538
　2 バイトに関する考慮事項　538
中国語 (繁体字) EUC コード・セット
　REXX に関する考慮事項　756
中国語 (繁体字) コード・セット　535
　を使用したアプリケーション開発　538
　C/C++ に関する考慮事項　643
注釈
　SQL の規則　705, 728
注釈、SQL の規則　615
抽出
　大量データ　577
抽出例、ドキュメントからファイル (表中の CLOB エレメント) へ　384
長フィールド　813
通貨の例、CREATE DISTINCT TYPE を使用した　297
通常呼び出し、UDF に対する　418
データ
　大量の抽出　577
　抽出時の障害の回避　577
データ値の制御についての考慮事項
　アプリケーションのロジックと変数のタイプ　30
　参照保全制約　29
　データ・タイプ　29
　表検査の制約　29
　CHECK OPTION を使用した視点　30
データ構造
　ストアード・プロシージャーの割り振り　207
　操作、DB2DARI ストアード・プロシージャーの　791
　ユーザー定義の、複数のスレッドにおける　558
　SQLEDBDESC　522
データ構造を宣言する　13
データ制御言語 (DCL)　814
データ操作言語 (DML)　813
データ定義言語 (DDL)　812
　保管点での発行　195
データ転送
　更新　111
データの関連についての考慮事項
　アプリケーションのロジック　31
　参照保全　30
　トリガー　31
データベース記述子ブロック (SQLEDBDESC)、照合順序の指定　522
データベース作成ウィザード　851
データベース追加ウィザード　850, 851, 852

データベースの作成、照合順序の指定 522
データベース・アクセス
異なるコンテキストの使用 557
マルチスレッドの使用 557
データベース・バックアップ・ウィザード 850
データベース・マネージャー API
ストアード・プロシージャーを使用しての呼び出し 204
定義、サンプル・プログラム 111
データへのアクセスについての考慮事項
組み込み SQL 26
使用、照会製品の 28
DB2 コール・レベル・インターフェース (DB2 CLI) 27
JDBC 27
Microsoft 仕様 28
ODBC 27
Perl の使用 28
REXX 26
データ・ソース、連合システムでの
関数の呼び出し 600
照会に分散要求を使用 597
データ・タイプのマッピング 593
テーブルへのアクセス、視点 588
パススルーを使用した照会 603
分離レベルのマッピング 592
マッピング、DB2 関数の 600
データ・タイプ
オブジェクト関連 289
拡張 UNIX コードに関する考慮事項 546
クラス・データ・メンバー、C/C++ での宣言 636
サポートされる
COBOL での 718
COBOL におけるその規則 720
FORTRAN での 734
FORTRAN におけるその規則 736
データ・タイプ *(続き)*
数値 813
説明 14
選択、グラフィック・タイプの 639
タイプのリストおよび UDF 内でのタイプの表示 426
データ値の制御についての考慮事項 29
変換
DB2 と COBOL 間の 718
DB2 と FORTRAN 間の 734
DB2 と OLE DB 表関数の間の 453
変換についての考慮事項 353
ポインター、C/C++ での宣言 635
文字変換によるオーバーフロー 547
10 進数
FORTRAN での 735
BLOB 365
CLOB 365
CLOB、C/C++ における 650
COBOL での 718
C/C++ 645, 650
C/C++ での 645
DB2 と C/C++ との間の変換 645
DB2 と OLE オートメーションの間の変換 445
DB2 と REXX 間の変換 749
DBCLOB 365
FOR BIT DATA、COBOL における 721
FOR BIT DATA、C/C++ における 650
FORTRAN での 734
Java 658
Java ストアード・プロシージャー (DB2GENERAL) 794
OLE オートメーション 445
SQL 列タイプのリスト 84
UDF への受け渡し方法 426
VARCHAR、C/C++ における 650
データ・タイプ・マッピング 593
データ・ソースの作成 594
デフォルト 593
特定の列について作成 594
ALTER NICKNAME ステートメント 595
CREATE TYPE MAPPING ステートメント 594
出口ルーチンの使用制限 125
テスト、UDF の 497
テストおよびデバッグのユーティリティー
更新、システム・カタログ統計 44
システム・カタログ視点 44
データベース・システム・モニター 44
標識機能 44
Explain 機能 44
テスト環境
区分データベース環境の 583
セットアップ 40
テスト・データベース、作成のガイドライン 40
テスト・データ
生成 42
テスト・データベース
勧告 41
作成 40
CREATE DATABASE API 41
デッドロック
バッファー挿入におけるエラー報告 575
複数のコンテキストで避ける 560
マルチスレッドのアプリケーションでの 559
出入力、画面とキーボードと UDF への 467
デバッグ
ストアード・プロシージャー 244, 257
ストアード・プロシージャー・ビルダーの使用 685
Java プログラム 660
SQL プロシージャー 273, 275

デバッグ *(続き)*
SQLJ プログラム 660
Visual Studio の使用 257
デバッグ、UDF の 497
テリトリー
SQLCA の SQLERRMC フィールドの 815
同期点マネージャー 555
動的 SQL
カーソル処理 138
カーソル処理、サンプル・プログラム 140
概説 133
許可についての考慮事項 37
構造型用の transform グループ 343
考慮事項 134
サポート・ステートメントのリスト 133
制限 134
静的 SQL との比較 134
静的との構文上の相違 134
動的 SQL との対比 68
任意ステートメントの処理 159
パラメーター・マーカー 168
DB2 コネクト・サポート 811
EXECUTE IMMEDIATE ステートメントの要約 134
EXECUTE ステートメントの要約 134
PREPARE ステートメントの要約 134
PREPARE、DESCRIBE、および FETCH の使用 137
SQLDA の宣言 150
動的カーソル 816
動的ステートメント
バインド 59
動的タイプ 317
動的メモリーの割り振り、UDF 内での 465
登録
OLE オートメーション UDF 442
UDF 394
登録、Java ストアード・プロシージャーの 683
特殊タイプ 391
解決、非修飾の特殊タイプ 296
強力タイピング 298
操作
例 298
特殊タイプの定義 296
表の定義 297
特殊レジスター
CURRENT EXPLAIN MODE 59
CURRENT FUNCTION PATH 59
CURRENT QUERY OPTIMIZATION 59
特定名、UDF に渡される 415
トランザクション
暗黙に終了 22
コミット、作業の 21
作業のロールバック 21
終了、トランザクションの 21
説明 20
トランザクションの開始 20
プログラムの終了
COMMIT および ROLLBACK ステートメント 22
保管点 191
トランザクション・プロセス・モニター
X/Open XA インターフェース 563
トランザクション・ログ、バッファー化挿入考慮事項 574
トランザクション・ログ考慮事項
バッファー化挿入に関する 574
トリガー
アプリケーションのロジックについての考慮事項 32
影響される行のセット 503
および DB2 オブジェクト拡張 289
概説 501
カスケード 510
活動化時間 503
参照制約、トリガーを使用しての対話 510
参照制約との対話 510
トリガー *(続き)*
データの関連についての考慮事項 31
定義 499
トリガー SQL ステートメント 507
トリガーの細分性 503
トリガーを使う理由 499
トリガー・アクション条件 507
トリガー・イベント 502
複数トリガーの順序付け 510
変位変数 505
変換表 506
利点 500
AFTER トリガー 503, 508
BEFORE トリガー 503, 508
DELETE 操作 502
INSERT 操作 502
SQL トリガー・ステートメントを含む関数
RAISE_ERROR 組み込み関数 509
UDT、UDF、および LOB との協調 511
UPDATE 操作 502
WHEN 文節 507
トリガー・イベント、UPDATE、INSERT または DELETE などの 502
取り消し、変更の 21
取り出し
行 98
複数行 87
1 行 70

[ナ行]

長いフィールドの制限
バッファー化挿入の使用 576
名前変更、パッケージの 59
ニックネーム
関連情報のカタログ 589
考慮事項、制約事項 589
視点での使用 592
CREATE NICKNAME ステートメント 589

日本語 EUC コード・セット
REXX に関する考慮事項 756
日本語および中国語 (繁体字) EUC コード・セット
COBOL に関する考慮事項 721
FORTRAN に関する考慮事項 737
日本語コード・セット 535
を使用したアプリケーション開発 538
C/C++ に関する考慮事項 643
入出力、画面とキーボードと UDF への 467
入力および出力ファイル
COBOL 701
C/C++ 610
FORTRAN 724
入力ファイルの拡張子、C/C++ 言語 610
ネストされたストアード・プロシージャー 224
結果セットの戻り 269
再帰 269
制約事項 269
パラメーターの受け渡し 269
SQL プロシージャー 269

[ハ行]

バインド
オプション 58
概説 58
考慮事項 60
実行据え置き 61
動的ステートメント 59
バインド・ファイル記述ユーティリティー、db2bfd 62
バインド・オプション
EXPLSNAP 60
FUNCPATH 60
INSERT BUF 574
QUERYOPT 60
バインド・ファイル
下位互換性 60
サポート、REXX アプリケーションに対する 752
バインド・ファイル *(続き)*
プリコンパイル・オプション 55
パススルー
考慮事項、制約事項 603
COMMIT ステートメント 603, 604
SET PASSTHRU RESET ステートメント 604
SET PASSTHRU ステートメント 604
SQL 処理 603
派生した列 184
パッケージ
作成 58
作成、コンパイル済みアプリケーションの 54
サポート、REXX アプリケーションに対する 752
属性 816
タイム・スタンプ・エラー 63
名前変更 59
パッケージ属性
作成者 816
修飾子 816
所有者 816
パッケージの作成、コンパイル済みアプリケーションの 54
バッファー化挿入
エラー検出、その最中の 574
オープン状態 575
概説 571
グループ、行の 574
クローズ状態 575
クローズするステートメント 572
固有キー違反 575
使用に関する考慮事項 574
使用の制限 576
その非同期特性 574
デッドロック・エラー 575
トランザクション・ログ考慮事項 574
長いフィールドの制限 576
バッファー化挿入におけるエラー報告 575
バッファー・サイズ 571
バッファー化挿入 *(続き)*
部分的に満たされた 572
保管点に関する考慮事項 197, 572
利点 574
CLP ではサポートされていない 577
INSERT BUF バインド・オプション 574
SELECT バッファー化挿入 575
バッファー・サイズ、バッファー化挿入の 571
母音の検出、UDF 例における CLOB の折り返し 474
パフォーマンス
影響する要因、静的 SQL 68
区分データベース環境における改善 569
向上
使用、ストアード・プロシージャーの 202
指示 DSS を用いた改善 570
静的 SQL ステートメントのプリコンパイル 63
データのブロックの受け渡し 567
動的 SQL キャッシュ 68
パッケージによる最適化 63
バッファー化挿入を使った改善 571
読み取り専用カーソルを用いた改善 98
ラージ・オブジェクト 367
ローカル・バイパスを用いた改善 570
LOB ロケーターの使用による向上 459
NOT FENCED ストアード・プロシージャー 244
READ ONLY カーソルを用いた改善 569
UDF 390
パフォーマンスおよび特殊タイプ 295
パフォーマンス構成ウィザード 851

パフォーマンスの利点
　バッファー化挿入での　574
パラメーター・マーカー　177
　関数内での例　403
　動的 SQL での使用　168
　任意ステートメントの処理　159
　プログラミング例　169
　SQLVAR 項目　168
ハンドラー
　概説　263
　例　265
ハンドル
　環境ハンドル　177
　記述子ハンドル　177
　ステートメント・ハンドル　177
　接続ハンドル　177
反復可能読み取りの技法　108
比較の例、特殊タイプが関係する 299, 301
引き数、DB2 から UDF に渡される 411
引き数タイプ、UDF 内でのプロモーション　426
非実行 SQL ステートメント
　DECLARE CURSOR　19
　INCLUDE　19
　INCLUDE SQLDA　19
非修飾表名
　解決　60
非同期イベント　557
非同期特性、バッファー化挿入の 574
非分離ストアード・プロシージャー 244
表
　一時　189
　カーソルを末尾に置く　111
　コミット、変更の　21
　データ・ソース表　588
　lob-options-clause、CREATE TABLE ステートメントの　367
　tablespace-options-clause、CREATE TABLE ステートメントの　367
表関数　394, 412
　アプリケーション設計に関する考慮事項　458
表関数 *(続き)*
　呼び出しタイプ引き数の内容 419
　Java での　437
　OLE DB　448
　SQL 結果引き数　412
表関数の例　400
表検査の制約、データ値の制御についての考慮事項　29
表作成ウィザード　851
表示
　オンライン情報　848
標識機能ユーティリティー、プリコンパイルに使用する　57
標識変数
　使用、DB2DARI ストアード・プロシージャーにおける　791
　宣言　82
　ヌル可能列での使用　86
　例　86
　COBOL におけるその規則　711
　C/C++ におけるその規則　622
　FORTRAN におけるその規則 732
　REXX におけるその規則　744
標識変数および LOB ロケーター 382
表示タイプ　310
表スペース作成ウィザード　851
表名
　解決、非修飾の　60
ファイル拡張子
　サンプル・プログラム　765
ファイル参照宣言、REXX における 747
ファイル参照変数
　出力値　383
　使用例　384
　入力値　382
　LOB 操作のための　365
復元、データの　21
復元ウィザード　852
複合 SQL
　NOT ATOMIC　823
複数トリガーの順序付け　510
副表
　作成　313
　副表の属性の挿入　319
ブック　833, 845
浮動小数点パラメーター、UDF への 428
フラッシュされたバッファー化挿入 572
プリコンパイラー
　オプション　55
　概説　50
　サポートされている言語　12
　出力のタイプ　55
　COBOL　701
　C/C++ 3 文字表記　609
　C/C++ 記号置換　609
　C/C++ 言語　638
　C/C++ 言語のデバッグ　614
　C/C++ マクロ処理　609
　C/C++ 文字セット　609
　C/C++ #include マクロ　609
　DB2 コネクト・サポート　812, 815
　FORTRAN　723
プリコンパイル　56
　アクセス、 DB2 コネクトを介してホストまたは AS/400 アプリケーション・サーバーへの　56
　オプション、更新可能カーソル 99
　概説　54
　動的 SQL ステートメントのサポート　133
　標識機能ユーティリティー　57
　複数のサーバーへのアクセス　56
　例　54
プリコンパイル・オプション
　WCHARTYPE
　　NOCONVERT　245
プリプロセッサー機能と SQL プリコンパイラー　629
プログラミング、枠組みの　23
プログラミングに関する考慮事項
　各国語サポート　517
　異なるコード・ページ間の変換 517

プログラミングに関する考慮事項 *(続き)*
照合順序 517
複合環境におけるプログラミング 517
ホストまたは AS/400 環境での 811
ホストまたは AS/400 サーバーにアクセスする 556
COBOL での 701
C/C++ での 609
FORTRAN での 723
REXX での 739
X/Open XA インターフェース 563
プログラム ID (progID)、OLE オートメーション UDF の 442
プログラム変数のタイプ、データ値の制御についての考慮事項 30
ブロック化 815
分散環境 811
分散サブセクション (DSS) 570
分散要求
コーディング 597
最適化 599
分離レベル 592, 819, 820
ページ・レベルのロック 818
べき乗および UDF の定義例 396
変位変数、OLD および NEW
トリガー・イベントのタイプに基づく変位変数 505
変換
文字 813
変換表
トリガー・イベントのタイプに基づく 506
OLD_TABLE と NEW_TABLE 506
変更のコミット、表 21
変数
SQLCODE 652, 721, 736
SQLSTATE 652, 721, 736
変数、REXX に事前定義された 744
変数を宣言する 13
報告、エラーの 584
保管、ラージ・オブジェクトの 289
保管点 191
アトミック複合 SQL 195
カーソルのブロック化 197
制約事項 195
データ定義言語 195
トリガー 195
ネストされた 195
バッファー化挿入 197
SET INTEGRITY ステートメント 195
XA トランザクション・マネージャー 198
保管点、バッファー化挿入考慮事項 572
保護、UDF の 465
ホスト変数
関連付け、SQL ステートメントとの 15
クラス・データ・メンバー、C/C++ での処理 636
グラフィック・データ 638
クリア、REXX における LOB ホスト変数の 748
決定、列に使用する定義方法の 16
構造型の宣言 363
参照
COBOL での 707
FORTRAN での 729
ストアード・プロシージャー内での割り振り 207
ストアード・プロシージャーに関する考慮事項 242
ストアード・プロシージャーのための初期化 207
静的 SQL 77
宣言 77
COBOL での 708
FORTRAN での 729
宣言、グラフィックの
COBOL での 710
宣言、サンプル・プログラム 111
宣言、データ・タイプへのポインターとしての 635
宣言、例 80
ホスト変数 *(続き)*
宣言の規則 77
選択、グラフィック・データ・タイプの 639
データのブロックの受け渡しに使用 567
定義 77
動的 SQL での使用 133
ファイル参照宣言、COBOL における 713
ファイル参照宣言、FORTRAN における 733
ファイル参照宣言、REXX における 747
変数リスト・ステートメントを用いた宣言 160
マルチバイト文字のエンコード 639
命名
COBOL での 707
FORTRAN での 729
C または C++ における LOB データ宣言 625
C/C++ でのグラフィック宣言 622
C/C++ での参照 616
C/C++ での初期化 629
C/C++ での宣言 618
C/C++ での命名 616
C/C++ における LOB ロケーター宣言 627
C/C++ におけるファイル参照宣言 628
db2dclgn による宣言 80
FORTRAN、概説 729
LOB データ、REXX の 746
LOB データ宣言、COBOL における 711
LOB データ宣言、FORTRAN における 732
LOB ロケーター宣言、COBOL における 713
LOB ロケーター宣言、FORTRAN における 733
LOB ロケーター宣言、REXX における 746

ホスト変数 *(続き)*
NULL 終了ストリング、C/C++ での処理 633
REXX での 743
REXX での参照 743
REXX での命名 743
SQL からの参照、例 81
WCHARTYPE プリコンパイラー・オプション 640
ホストまたは AS/400
ホスト・サーバーにアクセスする 556
ホストまたは AS/400 環境
プログラミング 811
ホストまたは AS/400 サーバーとワークステーションとの違い 825

［マ行］

マクロ、sqludf.h での 435
マクロ処理、C/C++ 言語の 609
マルチサイト更新
いつ使用するか 550
一般的な考慮事項 549
概説 549
構成パラメーター 554
サポート 824
ストアード・プロシージャーに関する考慮事項 242
制約事項 554
マルチサイト更新アプリケーション用の SQL コーディング 551
DB2 コネクト・サポート 824
マルチサイト更新構成パラメーター
LOCKTIMEOUT 554
RESYNC_INTERVAL 554
SPM_LOG_NAME 555
SPM_NAME 554
SPM_RESYNC_AGENT_LIMIT 554
TM_DATABASE 554
TP_MON_NAME 554
マルチサイト更新の構成ウィザード 850
マルチスレッド
コンテキスト間でのアプリケーションの従属関係 559
マルチスレッド *(続き)*
コンテキスト間でのデッドロックを避ける 560
コンテキスト間におけるデータベースの従属関係 559
指針 558
DB2 アプリケーションでの使用 557
マルチバイトに関する考慮事項
中国語 (繁体字) コード・セット、C/C++ での 643
日本語および中国語 (繁体字) EUC コード・セット
COBOL での 721
FORTRAN での 737
日本語コード・セット、C/C++ での 643
REXX における中国語 (繁体字) EUC コード・セット 756
REXX における日本語 EUC コード・セット 756
マルチバイト文字のサポート
コード・ポイント、特殊文字の 526
マルチバイト・コード・ページ
中国語 (繁体字) コード・セット 535, 538
日本語コード・セット 535, 538
未確定カーソル 815
矛盾状態 20
矛盾データ 20
メール OLE オートメーション UDF オブジェクトの例、BASIC による 495
メソッド
インプリメント 390
作成 395, 409
定義 389
登録 395
呼び出し 312
呼び出し演算子 312
理論的説明 390
メソッド呼び出し
OLE オートメーション UDF の 443
メッセージ・ファイルの定義 56
メモリー
LOB ロケーターの使用による所要量の減少 459
メモリーの割り振り、コード・ページが異なる場合 541
メモリーの割り振り、UDF 内での動的 465
メモリー・サイズ、UDF のために共用される 467
メンバー演算子、C/C++ での制限 638
文字セット
拡張 UNIX コード (EUC) 533
文字比較、概要 519
文字変換 813
アプリケーションの実行時 528
各国語サポート (NLS) 529
規則、ストリング変換に関する 547
行われる時 529
コーディング、ストアード・プロシージャーの 527, 548
サポートされるコード・ページ 530
ストリング長のオーバーフロー 547
ストリング長のオーバーフロー、データ・タイプを超えた 547
展開 532
プリコンパイルおよびバインド時 528
プログラミングに関する考慮事項 525
SQL ステートメントのコーディング 525
Unicode (UCS-2) 548
文字ホスト変数
固定および NULL 終了、C/C++ での 620
COBOL 709
C/C++ 可変長 621
C/C++ での可変長 621
C/C++ の固定および NULL 終了 620
FORTRAN 730
文字ラージ・オブジェクト 365

モデル、DB2 プログラミングの 23
モデル化、独立オブジェクトとしてのエンティティーの 289
戻りコード 17
SQLCA 構造 122
問題解決
ストアード・プロシージャー 257

[ヤ行]

ユーザーが更新できるカタログ統計
ユーティリティーのプロトタイプ化 45
ユーザー定義関数、アプリケーションのロジックについての考慮事項 32
ユーザー定義タイプ
DB2 コネクトがサポートする 814
ユーザー定義タイプ (UDT)
除去の制限 327
ユーザー定義の照合順序 818, 827
ユーザー定義のソース派生関数の例、特殊タイプでの 302
ユーザー定義のタイプの除去 327
ユーティリティー API
インクルード・ファイル、COBOL アプリケーションの 703, 704, 705
インクルード・ファイル、C/C++ アプリケーションの 613
インクルード・ファイル、FORTRAN アプリケーションの 727
呼び出し、REXX アプリケーションから DB2 CLP を 752
呼び出し、REXX アプリケーションからの 752
呼び出し、UDF の 401
呼び出し規則
UDF の 426
呼び出しタイプ 458
スカラー関数の場合の内容 418
表関数の場合の内容 419
呼び出しタイプ、UDF に渡される 418

[ラ行]

ラージ・オブジェクト値 365
ラージ・オブジェクト記述子 365
ラッチ
複数のスレッドにおける状態 557
ランタイム・サービス
複数のスレッド、ラッチへの影響 557
リモート作業単位 549
リリース情報 844
リンク
概説 57
リンク、UDF の 394
ルート・タイプ 308
ループ状態のアプリケーション
診断 585
例
異なる特殊タイプが関係する割り当て 303
サンプル SQL 宣言セクション、サポートされる SQL データ・タイプの 648
据え置き、LOB 式の評価の 375
宣言する、COBOL を使用して BLOB ファイル参照を 713
宣言する、COBOL を使用して BLOB ロケーターを 713
宣言する、COBOL を使用して BLOB を 712
宣言する、COBOL を使用して CLOB を 712
宣言する、COBOL を使用して DBCLOB を 712
宣言する、FORTRAN を使用して BLOB ファイル参照を 734
宣言する、FORTRAN を使用して BLOB を 732
宣言する、FORTRAN を使用して CLOB ファイル・ロケーターを 733
例 *(続き)*
宣言する、FORTRAN を使用して CLOB を 732
探索および更新でのパラメーター・マーカーの使用 169
抽出、ドキュメントからファイル (表中の CLOB エレメント) への 384
動的 SQL 内での割り当て 302
特殊タイプが関係する比較 299, 301
特殊タイプが関係する割り当て 302
特殊タイプでのユーザー定義のソース派生関数 302
ADHOC.SQC C プログラム・リスト 164
BLOB データ宣言 626
CLOB 値を使う作業のためのロケーターの使用 369
CLOB データ宣言 626
CLOB ファイル参照 629
CLOB 列へのデータの挿入 388
CLOB ロケーター 628
CREATE DISTINCT TYPE を使用した通貨 297
CREATE DISTINCT TYPE を使用しての再開 297
CREATE TABLE を使用した売上 297
CREATE TABLE を使用した応募の書式 297
DB2Appl.java 666
DBCLOB データ宣言 627
DYNAMIC.CMD REXX プログラム・リスト 148
Dynamic.java Java プログラム・リスト 144
DYNAMIC.SQB COBOL プログラム・リスト 146
DYNAMIC.SQC C プログラム・リスト 142
FORTRAN における文字ホスト変数の構文 730, 731
Java アプレット 666

例 *(続き)*
LOBEVAL.SQB COBOL プログラム・リスト 379
LOBEVAL.SQC C プログラム・リスト 377
LOBFILE.SQB COBOL プログラム・リスト 386
LOBFILE.SQC C プログラム・リスト 385
LOBLOC.SQB COBOL プログラム・リスト 372
LOBLOC.SQC C プログラム・リスト 370
SQL ステートメント内でのクラス・データ・メンバーの使用 637
SQLEXEC、SQLDBS、および SQLDB2 の登録 741
SQLEXEC、SQLDBS、および SQLDB2 の登録、REXX の場合 740
UNION 形式での特殊タイプの使用 304
V5SPCLI.SQC C プログラム・リスト 805
V5SPSRV.SQC C プログラム・リスト 809
Varinp.java Java プログラム・リスト 173
VARINP.SQB COBOL プログラム・リスト 175
VARINP.SQC C プログラム・リスト 171
列
サポートされる SQL データ・タイプ 84
識別 184
生成された 184
ヌル値の設定 83
ヌル可能データ列での標識変数の使用 86
派生した 184
列オプション 320
説明 590
ALTER NICKNAME ステートメント 591
列関数 394
列タイプ
概説 335
作成 335
列タイプ、COBOL での作成 718
列タイプ、C/C++ での作成 645
列タイプ、FORTRAN での作成 734
連合システム
関数マッピング 600
関数マッピング・オプション 601
サーバー・オプション 599
紹介 587
データ保全性 592
データ・ソース関数 600
データ・ソース表、視点
考慮事項、制約事項 589
情報のカタログ 589
ニックネーム 588
データ・タイプ・マッピング 593
ニックネーム 588
パススルー 603
分散要求 597
分離レベル 592
列オプション 590
ローカル・バイパス 570
ロールバック、変更の 21
ロケーター、LOB 操作のための 365
ロケール
アプリケーション中の導出 524
DB2 の導出法 524
ロック
行レベル 818
タイムアウト 818
バッファー化挿入エラー 575
ページ・レベル 818

［ワ行］

渡す、スレッド間でコンテキストを 558
割り当ての例、異なる特殊タイプが関係する 303
割り当ての例、動的 SQL 内での 302
割り当ての例、特殊タイプが関係する 302
割り込み、SIGUSR1 125
割り込みハンドル、SQL ステートメントでの 124
割り振り、UDF 内での動的メモリーの 465

［数字］

2 進ラージ・オブジェクト 365
2 バイト文字セット
構成パラメーター 534
異なるコード・ページ 540
混合コード・セット環境 540
照合に関する考慮事項 540
中国語 (繁体字) コード・セット 538
中国語 (繁体字) に関する考慮事項 538
日本語コード・セット 538
2 バイト文字セット (DBCS)
中国語 (繁体字) コード・セット 535
日本語コード・セット 535
2 バイト文字ラージ・オブジェクト 365
2 バイト・コード・ページ 537
3 文字表記 609
64 ビット整数 (BIGINT) データ・タイプ
DB2 コネクト・バージョン 7 がサポートする 814

A

ACQUIRE ステートメント 825
ActiveX Data Object 仕様
DB2 でサポートされる 28
ADD METHOD 312
ADHOC.SQC C プログラム・リスト 164
ADO 仕様
DB2 でサポートされる 28

AFTER トリガー 503, 508
ALLOW PARALLEL 文節 438
ALTER NICKNAME ステートメント
データ・タイプ・マッピング 595
列オプション 591
APPC、割り込みの処理 125
ARI (DB2 (VSE および VM 版)) 814
AS LOCATOR 文節 459
ASCII
混合バイト・データ 813
ソート順序 818
ATOMIC 複合 SQL
DB2 コネクトではサポートされない 823
AVG の例、UDT での 399

B

BASIC 言語
インプリメンテーション、OLE オートメーション UDF の 442
BASIC タイプと OLE オートメーション・タイプ 445
BEFORE トリガー 503, 508
BEGIN DECLARE SECTION 14
BigDecimal Java タイプ 658
BIGINT SQL データ・タイプ 84
COBOL 718
C/C++ 645
FORTRAN 734
Java 658
Java ストアード・プロシージャー (DB2GENERAL) 794
OLE DB 表関数 453
BIGINT パラメーター、UDF への 427
BINARY データ・タイプ、COBOL における 717
BIND API 61
BIND API、パッケージの作成 58
bind files for REXX 752
BIND PACKAGE コマンド
再バインド 64
BLOB (2 進ラージ・オブジェクト)
使用および定義 365
blob C/C++ タイプ 645
BLOB FORTRAN タイプ 734
BLOB SQL データ・タイプ 84, 445
COBOL 718
C/C++ 645
FORTRAN 734
Java 658
Java ストアード・プロシージャー (DB2GENERAL) 794
OLE DB 表関数 453
REXX 749
BLOB パラメーター、UDF への 433
BLOB-FILE COBOL タイプ 718
BLOB-LOCATOR COBOL タイプ 718
blob_file C/C++ タイプ 645
BLOB_FILE FORTRAN タイプ 734
blob_locator C/C++ タイプ 645
BLOB_LOCATOR FORTRAN タイプ 734
BSTR OLE オートメーション・タイプ 445

C

C NULL 終了グラフィック・ストリング SQL データ・タイプ 445
C NULL 終了ストリング SQL データ・タイプ 445
C 言語型定義、sqludf.h での 435
CALL USING DESCRIPTOR ステートメント (OS/400) 821
CALL ステートメント
さまざまなプラットフォーム 820
初期化する、クライアントをストアード・プロシージャー (DB2DARI) 用に
SQLDA 構造 789
ストアード・プロシージャーの呼び出し 207
Java での 680
CARDINALITY 指定、表関数での 458
CAST FROM 文節 412
CAST FROM 文節、CREATE FUNCTION ステートメント内の 426
CHAR 414
char C/C++ タイプ 645
CHAR SQL データ・タイプ 84, 445
COBOL 718
C/C++ 645
FORTRAN 734
Java 658
Java ストアード・プロシージャー (DB2GENERAL) 794
OLE DB 表関数 453
REXX 749
CHAR タイプ 693
CHAR パラメーター、UDF への 429
CHARACTER パラメーター、UDF への 429
CHARACTER*n FORTRAN タイプ 734
CHECKERR REXX プログラム・リスト 131
CHECKERR.CBL プログラム・リスト 129
CICS 812
CICS SYNCPOINT ROLLBACK コマンド 565
CLASSPATH 環境変数 684
CLI 177
client transform
概説 350
外部 UDF を使用して実装された 352
クライアント・アプリケーションからのインスタンスのバインドイン 353
データ・タイプ変換についての考慮事項 353
clob C/C++ タイプ 645
CLOB FORTRAN タイプ 734
CLOB SQL データ・タイプ 84, 445

CLOB SQL データ・タイプ *(続き)*
COBOL 718
C/C++ 645
FORTRAN 734
Java 658
Java ストアード・プロシージャー (DB2GENERAL) 794
OLE DB 表関数 453
REXX 749
CLOB パラメーター、UDF への 433
CLOB (文字ラージ・オブジェクト)
使用および定義 365
CLOB-FILE COBOL タイプ 718
CLOB-LOCATOR COBOL タイプ 718
clob_file C/C++ タイプ 645
CLOB_FILE FORTRAN タイプ 734
clob_locator C/C++ タイプ 645
CLOB_LOCATOR FORTRAN タイプ 734
CLOSE 呼び出し 419
COBOL
インクルード・ファイルのリスト 702
インディケーター表 716
オブジェクト指向の制約事項 722
規則、標識変数の 711
サポートされるデータ・タイプ 718
宣言、ホスト変数の 708
日本語および中国語 (繁体字) EUC に関する考慮事項 721
入力および出力ファイル 701
の制限 701
ファイル参照宣言 713
LOB データ宣言 711
LOB ロケーター宣言 713
SQL ステートメントの組み込み 49
COBOL 言語
データ・タイプ、サポートされる 718
COBOL データ・タイプ
BINARY 717
COBOL データ・タイプ *(続き)*
BLOB 718
BLOB-FILE 718
BLOB-LOCATOR 718
CLOB 718
CLOB-FILE 718
CLOB-LOCATOR 718
COMP 717
COMP-1 718
COMP-3 718
COMP-4 717
COMP-5 718
DBCLOB 718
DBCLOB-FILE 718
DBCLOB-LOCATOR 718
PICTURE (PIC) 文節 718
USAGE 文節 718
COLLECTION パラメーター 60
COMMIT WORK RELEASE ステートメント
サポートされない 825
DB2 コネクトでサポートされていない 825
COMMIT ステートメント 13
カーソルとの関連 88
終了、トランザクションの 21, 22
パススルー 603, 604
COMP および COMP-4 データ・タイプ、COBOL における 717
COMP-1、COBOL タイプにおける 718
COMP-3、COBOL タイプにおける 718
COMP-5、COBOL タイプにおける 718
COM.ibm.db2.app.Blob 800
COM.ibm.db2.app.Clob 801
COM.ibm.db2.app.Lob 800
COM.ibm.db2.app.StoredProc 796
COM.ibm.db2.app.UDF 437, 797
COM.ibm.db2.jdbc.app.DB2Driver 663
COM.ibm.db2.jdbc.net.DB2Driver 663
CONNECT
アプリケーション・プログラム 19
CONNECT *(続き)*
サンプル・プログラム 111
SQLCA.SQLERRD 設定値 543
CONNECT RESET ステートメント
終了、トランザクションの 22
CONNECT TYPE 2
ストアード・プロシージャーに関する考慮事項 242
CONNECT 時の SQLCA.SQLERRD 設定値 543
CONNECT ステートメント 13
constructor 関数 311
CONVERT
WCHARTYPE
ストアード・プロシージャーにおける 242
CREATE DATABASE API
SQLEDBDESC 構造 522
CREATE DISTINCT TYPE ステートメント
およびキャスト機能 391
使用例 297
特殊タイプを定義する 296
CREATE FUNCTION MAPPING ステートメント
関数名の指定 602
関数を呼び出す際のオーバーヘッドの削減 601
関数を呼び出す際のオーバーヘッドの見積もり 601
データ・ソース関数を連合サーバーに認識させる 600
CREATE FUNCTION ステートメント 416, 418, 420, 458, 459
連合システムでの 600
CAST FROM 文節 426
Java UDF 438
OLE オートメーション UDF の 442
RETURNS 文節 426
UDF を登録する 395
CREATE METHOD 312
CREATE METHOD ステートメント
メソッドを登録する 395
CREATE PROCEDURE ステートメント 208, 683

CREATE SERVER ステートメント 452
CREATE STORGROUP ステートメント
  DB2 コネクト・サポート 812
CREATE TABLE ステートメント
  使用例 297
  の中での列オプションの定義 320
  LOB 列の定義 367
  lob-options-clause 367
  tablespace-options-clause 367
CREATE TABLESPACE ステートメント
  DB2 コネクト・サポート 812
CREATE TRIGGER ステートメント
  概説 501
  順序、トリガー活動化の 503
  複数トリガー 510
  REFERENCING 文節 506
CREATE TYPE
  構造型 308
CREATE TYPE MAPPING ステートメント 594
CREATE TYPE ステートメント
  MODE DB2SQL 文節 306
  REF USING 文節 309
CREATE USER MAPPING ステートメント 453
CREATE VIEW ステートメント
  タイプ付き視点の作成 325
ctr() UDF C プログラム・リスト 478
CURRENT EXPLAIN MODE レジスター 59
CURRENT FUNCTION PATH レジスター 59
CURRENT QUERY OPTIMIZATION レジスター 59
CURSOR.SQB COBOL プログラム・リスト 96
CURSOR.SQC C プログラム・リスト 92
Cursor.sqlj Java プログラム・リスト 94
C++
  考慮事項、UDF に関する 468
  ストアード・プロシージャーに関する考慮事項 241
  タイプ修飾に関する考慮事項 610
C++ タイプと OLE オートメーション・タイプ 445
C/C++ 言語
  インクルード・ファイル、必要とされる 611
  サポートされるデータ・タイプ 645
  修飾演算子、制限 638
  初期化、ホスト変数 629
  処理、クラス・データ・メンバーの 636
  処理、NULL 終了ストリング 633
  宣言、グラフィック・ホスト変数 622
  宣言、ホスト変数の 618
  中国語 (繁体字) EUC に関する考慮事項 643
  データ・タイプ、サポートされる 645
  データ・タイプへのポインター、C/C++ での宣言 635
  日本語 EUC に関する考慮事項 643
  入力および出力ファイル 610
  ファイル参照宣言 628
  プログラミングにおける制限 609
  ホスト変数、命名 616
  メンバー演算子、制限 638
  文字セット 609
  3 文字表記 609
  LOB データ宣言 625
  LOB ロケーター宣言 627
  SQL ステートメントの組み込み 49
C/C++ データ・タイプ
  ヌル終了文字形式 645
  blob 645
  blob_file 645
C/C++ データ・タイプ *(続き)*
  blob_locator 645
  char 645
  clob 645
  clob_file 645
  clob_locator 645
  dbclob 645
  dbclob_file 645
  dbclob_locator 645
  double 645
  float 645
  long 645
  long int 645
  long long 645
  long long int 645
  short 645
  short int 645
  sqldbchar 645
  sqlint64 645
  VARCHAR 構造書式 645
  wchar_t 645

D

DATE OLE オートメーション・タイプ 445
DATE SQL データ・タイプ 84, 445
  COBOL 718
  C/C++ 645
  FORTRAN 734
  Java 658
  Java ストアード・プロシージャー (DB2GENERAL) 794
  OLE DB 表関数 453
  REXX 749
DATE パラメーター、UDF への 432
DB 情報 458
DB 情報、UDF に渡される 420
DB 情報引き数のエレメント 420
  アプリケーション許可 ID (authid) 421
  アプリケーション許可 ID の長さ (authidlen) 421
  固有のアプリケーション ID (appl_id) 423

DB 情報引き数のエレメント *(続き)*
スキーマ名 (tbschema) 421
スキーマ名の長さ (tbschemalen) 421
データベース名 (dbname) 421
データベース名の長さ (dbnamelen) 421
データベース・コード・ページ (codepg) 421
バージョン / リリース番号 (ver_rel) 422
表関数列の項目 (numtfcol) 422
表関数列リスト (tfcolumn) 423
表名 (tbname) 421
表名の長さ (tbnamelen) 421
プラットフォーム 422
プロシージャー ID (procid) 423
列名 (colname) 422
列名の長さ (colnamelen) 422
appl_id (固有のアプリケーション ID) 423
authid (アプリケーション許可 ID) 421
authidlen (アプリケーション許可 ID の長さ) 421
codepg (データベース・コード・ページ) 421
colname (列名) 422
colnamelen (列名の長さ) 422
dbname (データベース名) 421
dbnamelen (データベース名の長さ) 421
numtfcol (表関数列の項目) 422
procid (プロシージャー ID) 423
tbname (表名) 421
tbnamelen (表名の長さ) 421
tbschema (スキーマ名) 421
tbschemalen (スキーマ名の長さ) 421
tfcolumn (表関数列リスト) 423
(ver_rel) バージョン / リリース番号 422
DB2 BIND コマンド
作成、パッケージの 58
DB2 PREP コマンド
概説 54
DB2 PREP コマンド *(続き)*
例 54
DB2 コール・レベル・インターフェース (DB2 CLI)
データへのアクセスについての考慮事項 27
DB2 コネクト 811
分離レベル 819
DB2 コネクト・プログラミング上の考慮事項 811
DB2 システム・コントローラー 244
DB2 プログラム
セットアップ 14
DB2 ユニバーサル・データベース (AS/400 版)
FOR BIT DATA ストアード・プロシージャーに関する考慮事項 241
DB2 ユニバーサル・データベース (OS/390 版)
FOR BIT DATA ストアード・プロシージャーに関する考慮事項 241
DB2 ライブラリー
印刷版のブックの注文 845
インフォメーション・センター 849
ウィザード 850
オンライン情報の検索 853
オンライン情報の表示 848
オンライン・ヘルプ 845
構成内容 833
最新情報 844
セットアップ、文書サーバーの 852
ブック 833
ブックの言語 ID 843
PDF 資料の印刷 845
DB2Appl.java
アプリケーションの例 666
DB2ARXCS.BND REXX バインド・ファイル 752
DB2ARXNC.BND REXX バインド・ファイル 752
DB2ARXRR.BND REXX バインド・ファイル 752
DB2ARXRS.BND REXX バインド・ファイル 752
DB2ARXUR.BND REXX バインド・ファイル 752
db2bfd、バインド・ファイル記述ユーティリティー 62
DB2CODEPAGE レジストリー変数 523
db2dari 実行可能ファイル 226
DB2DARI ストアード・プロシージャー 214
db2dclgn コマンド 80
db2diag.log ファイル 584
DB2GENERAL ストアード・プロシージャー 214
DB2INCLUDE 環境変数 614, 706, 727
db2nodes.cfg ファイル 585
db2udf 実行可能ファイル 467
DB2Udf.java 436
DB2、DB2 コネクトを使用した 811
DB2_SQLROUTINE_KEEP_FILES 276
DBCLOB
中国語 (繁体字) コード・セット 539
日本語コード・セット 539
DBCLOB (2 バイト文字ラージ・オブジェクト)
使用および定義 365
dbclob C/C++ タイプ 645
DBCLOB SQL データ・タイプ 84, 445
COBOL 718
C/C++ 645
Java 658
Java ストアード・プロシージャー (DB2GENERAL) 794
OLE DB 表関数 453
REXX 749
DBCLOB タイプ 693
DBCLOB パラメーター、UDF への 433
DBCLOB-FILE COBOL タイプ 718

DBCLOB-LOCATOR COBOL タイプ 718
dbclob_file C/C++ タイプ 645
dbclob_locator C/C++ タイプ 645
DBCS 535
DBINFO キーワード 420
DCL (データ制御言語) 814
DDL (データ定義言語) 812
DECIMAL SQL データ・タイプ 84, 445
  COBOL 718
  C/C++ 645
  FORTRAN 734
  Java 658
  Java ストアード・プロシージャー (DB2GENERAL) 794
  OLE DB 表関数 453
  REXX 749
DECIMAL パラメーター、UDF への 428
DECLARE CURSOR ステートメント 19
DECLARE CURSOR ステートメント、概説 87
DECLARE PROCEDURE ステートメント (OS/400) 821
DECLARE STATEMENT
  DB2 コネクト・サポート 825
DECLARE ステートメント 825
DELETE
  トリガー 506
  DB2 コネクト・サポート 813
DELETE 操作とトリガー 502
DEREF 関数
  定義 330
  必要な特権 331
DESCRIBE ステートメント 825
  拡張 UNIX コードに関する考慮事項 545
  拡張 UNIX コードの考慮事項、EUC データベース 546
  構造型 363
  任意ステートメントの処理 159
  2 バイト文字セットに関する考慮事項 545
  DB2 コネクト・サポート 825
DFT_SQLMATHWARN 構成パラメーター 413
divid() UDF C プログラム・リスト 470
DML (データ操作言語) 813
double C/C++ タイプ 645
double Java タイプ 658
double OLE オートメーション・タイプ 445
DOUBLE PRECISION パラメーター、UDF への 429
DOUBLE SQL データ・タイプ 84, 445
DOUBLE パラメーター、UDF への 428, 429
DROP ステートメント
  タイプ・マッピング 327
  ユーザー定義タイプ 327
DSN (DB2 ユニバーサル・データベース (OS/390 版)) 814
DSS (分散サブセクション) 570
DUOW 549
DYNAMIC.CMD REXX プログラム・リスト 148
Dynamic.java Java プログラム・リスト 144
DYNAMIC.SQB COBOL プログラム・リスト 146
DYNAMIC.SQC C プログラム・リスト 142

E

EBCDIC
  混合バイト・データ 813
  ソート順序 818
END DECLARE SECTION 14
EUC 535
EUC (拡張 UNIX コード)
  文字セット 533
EXCSQLSTT コマンド 825
EXEC SQL INCLUDE SQLCA
  マルチスレッドに関する考慮事項 558
EXEC SQL INCLUDE ステートメント、C/C++ 言語制限 614
EXECUTE IMMEDIATE ステートメントの要約 134
EXECUTE ステートメントの要約 134
Explain スナップショット 60
EXPLAIN、ユーティリティーのプロトタイプ化 45
EXPLSNAP バインド・オプション 60
extern 宣言
  C++ 610
EXTERNAL ACTION オプションおよび UDF 465
EXTERNAL NAME 文節 451, 452
EXTERNAL 文節 209

F

FENCED オプションおよび UDF 465
FETCH ステートメント
  繰り返しアクセスの技法 108
  上方スクロールの技法 108
  ホスト変数 137
  SQLDA 構造の使用 154
FETCH 呼び出し 419
FINAL CALL キーワード 418
FINAL CALL 文節 419
findvwl() UDF C プログラム・リスト 474
FIPS 127-2 標準 17
FIRST 呼び出し 419
float C/C++ タイプ 645
float OLE オートメーション・タイプ 445
FLOAT SQL データ・タイプ 84, 445
  COBOL 718
  C/C++ 645
  FORTRAN 734
  Java 658
  Java ストアード・プロシージャー (DB2GENERAL) 794
  OLE DB 表関数 453
  REXX 749

FLOAT パラメーター、UDF への 429
fold() UDF C プログラム・リスト 474
FOR BIT DATA
  ストアード・プロシージャーに関する考慮事項 241
  データ・タイプ、C/C++ における 650
FOR BIT DATA SQL データ・タイプ 445
FOR BIT DATA 修飾子、UDF の 429
FOR EACH ROW トリガー 503
FOR EACH STATEMENT トリガー 503
FOR UPDATE 文節 98
FORCE コマンド 815
FORTRAN
  参照、ホスト変数の 729
  データ・タイプ、サポートされる 734, 735
  日本語および中国語 (繁体字) EUC に関する考慮事項 737
  入力および出力ファイル 724
  標識変数の規則 732
  ファイル参照宣言 733
  プログラミングにおける制限 723
  プログラミングに関する考慮事項 723
  ホスト変数、概説 729
  LOB データ宣言 732
  LOB ロケーター宣言 733
  SQL ステートメントの組み込み 49
FORTRAN 言語
  データ・タイプ、サポートされる 734
FORTRAN データ・タイプ
  BLOB 734
  BLOB_FILE 734
  BLOB_LOCATOR 734
  CHARACTER*n 734
  CLOB 734
  CLOB_FILE 734
FORTRAN データ・タイプ *(続き)*
  CLOB_LOCATOR 734
  INTEGER*2 734
  INTEGER*4 734
  REAL*2 734
  REAL*4 734
  REAL*8 734
FROM SQL transform 344
FUNCPATH バインド・オプション 60
function transform
  概説 345
  パラメーターを外部ルーチンに渡す 348
  SQL を本体として持つルーチンとして実装された 347

G

GENERAL WITH NULLS ストアード・プロシージャー 214
GENERAL ストアード・プロシージャー 214
GET ERROR MESSAGE API 125, 744
getAsciiStream JDBC メソッド 693
getString JDBC メソッド 693
getUnicodeStream JDBC メソッド 693
GRANT ステートメント
  表階層に対する発行 319
  DB2 コネクト・サポート 814
GRAPHIC SQL データ・タイプ
  COBOL 718
  C/C++ 645
  FORTRAN でサポートされていない 734
  Java 658
  Java ストアード・プロシージャー (DB2GENERAL) 794
  OLE DB 表関数 453
  REXX 749
GRAPHIC タイプ 693
GRAPHIC パラメーター、UDF への 430
GROUP BY 文節
  ソート順序 818

H

HTML
  サンプル・プログラム 843
HTML ページ
  Java アプレットへのタグ付け 666

I

IBM DB2 Universal Database Project Add-In for Microsoft Visual C++ 33, 35
IBM DB2 Universal Database Tools Add-In for Microsoft Visual C++ の活動化 36
IN ストアード・プロシージャー・パラメーター 209, 222
INCLUDE SQLCA
  疑似コード 17
INCLUDE SQLDA ステートメント 19
INCLUDE SQLDA ステートメント、SQLDA 構造の作成 155
INCLUDE ステートメント 19
INHERIT SELECT PRIVILEGES 文節 319
INOUT ストアード・プロシージャー・パラメーター 209, 223
INSERT BUF バインド・オプション
  バッファー化挿入 574
INSERT ステートメント
  を使用したタイプ付き表への挿入 320
  CLP ではサポートされていない 577
  DB2 コネクト・サポート 813
  VALUES 文節を指定 573
INSERT 操作とトリガー 502
Int Java タイプ 658
INTEGER 418
INTEGER SQL データ・タイプ 84, 445

INTEGER SQL データ・タイプ (続き)
COBOL 718
C/C++ 645
FORTRAN 734
Java 658
Java ストアード・プロシージャー (DB2GENERAL) 794
OLE DB 表関数 453
REXX 749
INTEGER または INT パラメーター、UDF への 427
INTEGER*2 FORTRAN タイプ 734
INTEGER*4 FORTRAN タイプ 734
IS OF 述部
を使用した戻されるタイプの制限 331
ISO 10646 標準規格 535
ISO 2022 標準規格 535
ISO/ANS SQL92 817
ISO/ANS SQL92 標準 17

J

Java
アプリケーションの配布および実行 666
アプリケーション・サポート 661
アプレットの配布および実行 666
アプレット・サポート 662
概説 655
ストアード・プロシージャー 689, 690
例 691
接続プール 670
デバッグ 660
トランザクション API (JTA) 670
比較、他の言語との 656
比較、SQLJ と JDBC の 656
DB2 の概説、サポート 661
JAR ファイルのインストール 689, 690
Java (続き)
JDBC 2.0 オプション・パッケージ API サポート 669
JDBC 指定 660
JDBC のサンプル・プログラム 664
JNDI サポート 669
SQL ステートメントの組み込み 49
SQLCODE 659
SQLJ (Embedded SQLJ for Java) 671
アプレット 672
位置指定された DELETE ステートメント 674
位置指定された UPDATE ステートメント 674
イテレーターの宣言 674
カーソルの宣言 674
制約事項 672
文節の例 674
変換プログラム 671
db2profc 671
db2profp 671
holdability 674
profconv 671
returnability 674
SQL ステートメントの組み込み 674
SQLJ (Java Embedded SQL)
ストアード・プロシージャーの呼び出し 680
プログラムの例 676
ホスト変数 680
SQLJ 指定 660
SQLMSG 659
SQLSTATE 659
UDF (ユーザー定義関数) 689, 690
例 691
Java Database Connectivity 663
Java I/O ストリーム
System.err 437
System.in 437
System.out 437
Java Naming and Directory Interface (JNDI) 669
Java UDF に関する考慮事項 411
Java アプリケーション
グラフィカル・オブジェクトおよびラージ・オブジェクト 693
SCRATCHPAD に関する考慮事項 438
UDF のシグニチャー 437
Java クラス・ファイル
置く場所 684
CLASSPATH 環境変数 684
java_heap_sz 構成パラメーター 684
jdk11_path 構成パラメーター 684
JAVA ストアード・プロシージャー 214
Java データ・タイプ
BigDecimal 658
Blob 658
double 658
Int 658
java.math.BigDecimal 658
short 658
String 658
Java における接続プール 670
Java パッケージおよびクラス 663
COM.ibm.db2.app 658
java.math.BigDecimal Java タイプ 658
java_heap_sz 構成パラメーター 684
JDBC
データへのアクセスについての考慮事項 27
比較、SQLJ との 656
プログラムの例 664
1.22 ドライバー 668
2.0 オプション・パッケージ API 669
2.0 コア API 668
2.0 ドライバー 668
COM.ibm.db2.jdbc.app.DB2Driver 663
COM.ibm.db2.jdbc.net.DB2Driver 663

JDBC *(続き)*
getAsciiStream メソッド 693
getString メソッド 693
getUnicodeStream メソッド 693
setAsciiStream メソッド 693
setString メソッド 693
setUnicodeStream メソッド 693
SQLJ 相互運用性 694
jdk11_path 構成パラメーター 684
JNDI (Java Naming and Directory Interface) 669
JTA (Java トランザクション API) 670

L

LABEL ON ステートメント 825
LANGLEVEL SQL92E プリコンパイル・オプション 817
LANGLEVEL プリコンパイル・オプション
MIA 650
SAA1 650
SQL92E および SQLSTATE または SQLCODE 変数の使用 652, 721, 736
LANGUAGE OLE 文節 442
LOB データ・タイプ
DB2 コネクト・バージョン 7 がサポートする 814
LOB (ラージ・オブジェクト)
値のプログラミング・オプション 368
および DB2 オブジェクト拡張 289
考慮事項、Java に関する 693
操作 289
トリガー、UDT、および UDF との協調 511
ファイル参照変数 365
出力値 383
使用例 384
入力値 382
SQL_FILE _APPEND、出力値オプション 383
LOB (ラージ・オブジェクト) *(続き)*
ファイル参照変数 *(続き)*
SQL_FILE_CREATE、出力値オプション 383
SQL_FILE_OVERWRITE、出力値オプション 383
SQL_FILE_READ、入力値オプション 382
保管 289
ラージ・オブジェクト値 365
ラージ・オブジェクト記述子 365
ロケーター 365, 367
使用例 369, 375
標識変数 382
プログラミング・シナリオ 375
LOB ロケーター
使用のシナリオ 464
UDF で使用される 459
LOB ロケーター API、UDF 内で使用される
sqludf_append API 460
sqludf_create_locator API 460
sqludf_free_locator API 460
sqludf_length API 460
sqludf_substr API 460
LOB ロケーターのサンプル・プログラム・リスト 488
LOBEVAL.SQB COBOL プログラム・リスト 379, 386
LOBEVAL.SQC C プログラム・リスト 377, 385
LOBLOC.SQB COBOL プログラム・リスト 372
LOBLOC.SQC C プログラム・リスト 370
lob-options-clause、CREATE TABLE ステートメントの 367
LOCKTIMEOUT 構成パラメーター 561
LOCKTIMEOUT マルチサイト更新構成パラメーター 554
long C/C++ タイプ 645
long int C/C++ タイプ 645
long long C/C++ タイプ 645
long long int C/C++ タイプ 645
long OLE オートメーション・タイプ 445
LONG VARCHAR
ストレージ限界 365
パラメーター、UDF への 429
LONG VARCHAR SQL データ・タイプ 84, 445
COBOL 718
C/C++ 645
FORTRAN 734
Java 658
Java ストアード・プロシージャー (DB2GENERAL) 794
OLE DB 表関数 453
REXX 749
LONG VARGRAPHIC
ストレージ限界 365
パラメーター、UDF への 431
LONG VARGRAPHIC SQL データ・タイプ 84, 445
COBOL 718
C/C++ 645
FORTRAN 734
Java 658
Java ストアード・プロシージャー (DB2GENERAL) 794
OLE DB 表関数 453
REXX 749
LONGVAR タイプ 693

M

maxdari 構成パラメーター 683
MIA 650
Microsoft Exchange、メールの使用例 495
Microsoft Transaction Server 仕様
データへのアクセスについての考慮事項 28
Microsoft Visual C++
IBM DB2 Universal Database Project Add-In 33
Microsoft 仕様
データへのアクセスについての考慮事項 28

Microsoft 仕様 *(続き)*
ADO (ActiveX Data Object) 28
MTS (Microsoft Transaction Server) 28
RDO (Remote Data Object) 28
Visual Basic 28
Visual C++ 28
MODE DB2SQL 文節 306
mutator メソッド 311

N

Netscape ブラウザー
インストール 848
NOCONVERT
WCHARTYPE
ストアード・プロシージャーにおける 242
NOLINEMACRO
PREP オプション 614
NOT ATOMIC 複合 SQL 823
NOT DETERMINISTIC オプションおよび UDF 465
NOT FENCED LOB ロケーター
UDF 460
NOT FENCED ストアード・プロシージャー
考慮事項 245
使用しての作業 244
プリコンパイル 245
NOT NULL CALL オプションおよび UDF 465
NOT NULL CALL 文節 413
NULL 値
受け取りの準備 82
NULL 終了文字書式 C/C++ タイプ 645
NULL で終了する 650
NUMERIC SQL データ・タイプ 445
COBOL 718
C/C++ 645
FORTRAN 734
Java 658
Java ストアード・プロシージャー (DB2GENERAL) 794
NUMERIC SQL データ・タイプ *(続き)*
OLE DB 表関数 453
REXX 749
NUMERIC パラメーター、UDF への 428

O

observer メソッド 311
ODBC
データへのアクセスについての考慮事項 27
OLE DB
サーバー名の使用 450
接続ストリングの使用 450
表関数 448
完全修飾名 451
サーバーの識別 452
作成 449
ユーザー・マッピングの定義 453
CREATE SERVER ステートメント 452
CREATE USER MAPPING ステートメント 453
EXTERNAL NAME 文節 451
CONNECTSTRING オプションの使用 450
DB2 でサポートされる 28
OLE オートメーション UDF
インプリメンテーション 442
オブジェクト・インスタンス 443
作成可能な単独使用 OLE オートメーション・サーバー 447
作成可能な複数使用 OLE オートメーション・サーバー 447
スクラッチパッドに関する考慮事項 443
BASIC でのインプリメンテーション 445
C++ でのインプリメンテーション 446
UDF 441
OLE オートメーション・オブジェクトの計算例 400
OLE オートメーション・オブジェクトを使った計算 400
OLE オートメーション・サーバー 442
OLE オートメーション・タイプ 443
OLE オートメーション・タイプと BASIC タイプ 445
OLE オートメーション・タイプと C++ タイプ 445
OLE オートメーション・データ・タイプ 445
BSTR 445
DATE 445
double 445
float 445
long 445
SAFEARRAY 445
short 445
OLE キーワード 441
OLE プログラム ID (progID) 442
OLE メッセージ交換の例 495
ONLY 文節
を使用した戻されるタイプの制限 331
OPENFTCH.SQB COBOL プログラム・リスト 106
OPENFTCH.SQC C プログラム・リスト 101
Openftch.sqlj Java プログラム・リスト 103
ORDER BY 文節
ソート順序 818
OS/400、DB2 コネクトを使用した 811
OUT ストアード・プロシージャー・パラメーター 209, 222
OUTER キーワード
を指定してサブタイプ属性を戻す 332

P

PDF 845
PDF 資料の印刷 845
Perl
  データへのアクセスについての考慮事項 28
PICTURE (PIC) 文節、COBOL タイプにおける 718
PREP オプション
  NOLINEMACRO 614
PREPARE ステートメント
  任意ステートメントの処理 159
  要約 134
  DB2 コネクト・サポート 825
printf()、UDF をデバッグするための 497
PUT ステートメント
  DB2 コネクトでサポートされていない 825

Q

QSQ (DB2 ユニバーサル・データベース (AS/400 版)) 814
QUERYOPT バインド・オプション 60

R

RAISE_ERROR 組み込み関数 509
RDO 仕様
  DB2 でサポートされる 28
REAL SQL データ・タイプ 84, 445
  COBOL 718
  C/C++ 645
  FORTRAN 734
  Java 658
  Java ストアード・プロシージャー (DB2GENERAL) 794
  OLE DB 表関数 453
  REXX 749
REAL パラメーター、UDF への 428
REAL*2 FORTRAN タイプ 734
REAL*4 FORTRAN タイプ 734
REAL*8 FORTRAN タイプ 734
REDEFINES、COBOL での 717
REF USING 文節 309
REFERENCING 文節、CREATE TRIGGER ステートメント内の 506
RELEASE SAVEPOINT ステートメント 195
Remote Data Object 仕様
  DB2 でサポートされる 28
REORGANIZE TABLE コマンド 537
RESULT REXX 事前定義変数 744
RESYNC_INTERVAL マルチサイト更新構成パラメーター 554
RETURNS TABLE 文節 412
RETURNS 文節、CREATE FUNCTION ステートメント内の 426
REVOKE ステートメント
  ステートメント 814
  表階層に対する発行 319
  DB2 コネクト・サポート 814
REXX
  カーソル ID 742
  クリア、LOB ホスト変数の 748
  サポートされた SQL ステートメント 742
  事前定義変数 744
  実行要件 751
  中国語 (繁体字) に関する考慮事項 756
  データへのアクセスについての考慮事項 26
  データ・タイプ、サポートされる 749
  日本語に関する考慮事項 756
  のストアード・プロシージャー 754
  の制限 740
  バインド・ファイル 752
  標識変数 744, 750
  プログラミングに関する考慮事項 740
  変数の初期化 754
REXX *(続き)*
  呼び出し、アプリケーションから DB2 CLP を 752
  ルーチンの登録 740
  API 構文 752
  LOB データ 746
  LOB ファイル参照宣言 747
  LOB ロケーター宣言 746
  SQLEXEC、SQLDBS、および SQLDB2 の登録 740
REXX API
  SQLDB2 739, 752
  SQLDBS 739
  SQLEXEC 739
REXX および C++ データ・タイプ 749
ROLLBACK TO SAVEPOINT ステートメント 195
ROLLBACK WORK RELEASE
  DB2 コネクトでサポートされていない 825
ROLLBACK ステートメント 13, 815
  カーソルとの関連 88
  終了、トランザクションの 22
  取り消し、変更の 21
  復元、データの 21
  ロールバック、変更の 21
ROWID データ・タイプ
  DB2 コネクト・バージョン 7 がサポートする 814
RQRIOBLK フィールド 815
RUOW 549

S

SAA1 650
SAFEARRAY OLE オートメーション・タイプ 445
SAVEPOINT ステートメント 194
SCRATCHPAD オプション
  OLE オートメーション UDF の 443
SCRATCHPAD キーワード 416, 418, 438, 456
SCRATCHPAD 文節 419

SELECT INTO ステートメント
　概説　70
SELECT ステートメント
　可変リストの概説　160
　検索されたデータの更新　111
　スーパー表からの特権の継承　319
　タイプ付き表　328
　の中の逆参照演算子　329
　の中の効力範囲が指定された参照　329
　バッファー化挿入考慮事項　575
　複数行の取り出し　87
　2 度目のデータ検索　109
　DB2 コネクト・サポート　813
　DECLARE CURSOR ステートメント　87
　EXECUTE ステートメントとの関連　134
　SQLDA の宣言　150
　SQLDA の割り振り後の記述　153
SET CURRENT FUNCTION PATH ステートメント　394
SET CURRENT PACKAGESET ステートメント　60
SET CURRENT ステートメント
　DB2 コネクト・サポート　826
SET PASSTHRU RESET ステートメント　604
SET PASSTHRU ステートメント　604
SET SERVER OPTION ステートメント　600
setAsciiStream JDBC メソッド　693
setString JDBC メソッド　693
setUnicodeStream JDBC メソッド　693
short C/C++ タイプ　645
short int C/C++ タイプ　645
short Java タイプ　658
short OLE オートメーション・タイプ　445
SIGNAL SQLSTATE SQL ステートメント、入力データの妥当性検査を行う　500
SIGUSR1 割り込み　125
SIMPLE WITH NULLSストアード・プロシージャー　214
SIMPLE ストアード・プロシージャー　214
SMALLINT　413
SMALLINT SQL データ・タイプ　84, 445
　COBOL　718
　C/C++　645
　FORTRAN　734
　Java　658
　Java ストアード・プロシージャー (DB2GENERAL)　794
　OLE DB 表関数　453
　REXX　749
SMALLINT パラメーター、UDF に対する　427
SmartGuides
　ウィザード　850
SPM_LOG_SIZE マルチサイト更新構成パラメーター　555
SPM_NAME マルチサイト更新構成パラメーター　554
SPM_RESYNC_AGENT_LIMIT マルチサイト更新構成パラメーター　554
SQL
　許可についての考慮事項　37
　許可についての考慮事項、静的 SQL の　38
　許可についての考慮事項、動的 SQL の　37
　許可についての考慮事項、API を使用するさいの　39
　動的に作成された　178
SQL インクルード・ファイル
　COBOL アプリケーションの　702
　C/C++ アプリケーションの　611
　FORTRAN アプリケーションの　724
SQL 結果　418, 458
SQL 結果、UDF に渡される　412
SQL 結果標識　418, 458
SQL 結果標識、UDF に渡される　413
SQL コードのプロトタイプ化　44
SQL 状態、UDF に渡される　414
SQL ステートメント
　エンド・ユーザー要求の保管　160
　区分　811
　サポート、REXX での　742
　シグナル・ハンドラー　125
　ストアード・プロシージャーの使用によるグループ化　204
　例外ハンドラー　125
　割り込みハンドラー　125
　COBOL 構文　705
　C/C++ 構文　615
　DB2 コネクト・サポート　825
　FORTRAN 構文　728
　REXX 構文　741
SQL ステートメント実行
　逐次化　557
SQL 宣言セクション　14
SQL 通信域 (SQLCA)　17
SQL データ・タイプ　443, 445
　BIGINT　84
　BLOB　84, 445
　CHAR　84, 445
　CLOB　84, 445
　COBOL　718
　C/C++ への変換　645
　DATE　84, 445
　DBCLOB　84, 445
　DECIMAL　84
　DOUBLE　445
　FLOAT　84, 445
　FOR BIT DATA　445
　FORTRAN　734
　GRAPHIC　445
　INTEGER　84, 445
　Java　658
　LONG GRAPHIC　445
　LONG VARCHAR　84, 445
　LONG VARGRAPHIC　84, 445
　NUMERIC　445
　OLE DB 表関数　453
　REAL　84, 445
　REXX　749
　SMALLINT　84, 445

SQL データ・タイプ *(続き)*
TIME 84, 445
TIMESTAMP 84, 445
VARCHAR 84, 445
VARGRAPHIC 84, 445
SQL データ・タイプ、UDF に渡される 426
SQL 引き数 418
SQL 引き数、DB2 から UDF に渡される 411
SQL 引き数、UDF に渡される 412
SQL 引き数標識 418
SQL 引き数標識、UDF に渡される 413
SQL プロシージャー
結果セットの受け取り 272
結果セットの戻り 269, 270
再帰 269
条件ハンドラー 264
制約事項 269
デバッグ 273, 275
動的 SQL 267
の CALL ステートメント 269
ログ・ファイル 275
RESIGNAL 266
SIGNAL 266
SQL1252A インクルード・ファイル
COBOL アプリケーションの 704
FORTRAN アプリケーションの 726
SQL1252B インクルード・ファイル
COBOL アプリケーションの 704
FORTRAN アプリケーションの 726
SQL92 817
SQLADEF インクルード・ファイル
C/C++ アプリケーションの 611
SQLAPREP インクルード・ファイル
COBOL アプリケーションの 702
C/C++ アプリケーションの 611
FORTRAN アプリケーションの 724
SQLCA
エラー発生時の不完全な挿入 575
多重定義の回避 18
バッファー化挿入におけるエラー報告 575
マルチスレッドに関する考慮事項 558
SQLERRMC フィールド 815, 824
SQLERRP フィールド 814
SQLCA structure
ストアード・プロシージャー内での使用 792
SQLCA インクルード・ファイル
COBOL アプリケーションの 702
C/C++ アプリケーションの 611
FORTRAN アプリケーションの 724
SQLCA 構造
インクルード・ファイル
COBOL アプリケーションの 702
FORTRAN アプリケーションの 724
インクルード・ファイル、C/C++ の 611
概説 122
警告 83
多重定義の回避 123
定義、サンプル・プログラム 111
報告、エラーの 584
マージされた複数の構造 584
要件 122
sqlerrd 584
SQLERRD(6) フィールド 585
SQLWARN1 フィールド 83
SQLCA 事前定義変数 744
SQLCA_92 インクルード・ファイル
COBOL アプリケーションの 702
FORTRAN アプリケーションの 725
SQLCA_92 構造
インクルード・ファイル
FORTRAN アプリケーションの 725
SQLCA_CN インクルード・ファイル 724
SQLCA_CS インクルード・ファイル 724
SQLCHAR 構造
データの受け渡し 158
SQLCLI インクルード・ファイル
C/C++ アプリケーションの 611
SQLCLI1 インクルード・ファイル
C/C++ アプリケーションの 611
SQLCODE
組み込み SQLCA 17
構造 122
スタンドアロン 817
プラットフォームの相違点 819
報告、エラーの 584
Java プログラムの 659
SQLCODE -1015 583
SQLCODE -1034 583
SQLCODE -1224 584
SQLCODES インクルード・ファイル
COBOL アプリケーションの 703
C/C++ アプリケーションの 612
FORTRAN アプリケーションの 725
SQLDA
マルチスレッドに関する考慮事項 558
SQLDA インクルード・ファイル
COBOL アプリケーションの 703
C/C++ アプリケーションの 612
FORTRAN アプリケーションの 725
SQLDA 構造
サーバー入力プロシージャーの例 808
最小構造を用いたステートメントの準備 151
作成 (割り振り) 155
作成、ホスト言語の例 156

SQLDA 構造 *(続き)*
十分な SQLVAR エンティティーの宣言 152
使用、サンプル・プログラム 161
初期化する、ストアード・プロシージャー (DB2DARI) 用に 789
ストアード・プロシージャーで使用されるフィールド
SQLDATA 792
SQLIND 792
SQLLEN 792
SQLTYPE 792
ストアード・プロシージャー内での使用 792
宣言 150
操作、DB2DARI ストアード・プロシージャーを使った 791
データの受け渡し 158
データのブロックの受け渡しに使用 567
入力 SQLDA プロシージャーの例 802
PREPARE ステートメントとの関連 134
SQLDACT インクルード・ファイル 725
SQLDATA フィールド 792
sqlda.n.SQLDAT 791
sqlda.n.SQLDATALEN 791
sqlda.n.SQLDATATYPE_NAME 791
sqlda.n.SQLIND 791
sqlda.n.SQLLEN 791
sqlda.n.SQLLONGLEN 791
sqlda.n.SQLNAME.data 791
sqlda.n.SQLNAME.length 791
sqlda.n.SQLTYPE 791
sqlda.SQLDABC 791
sqlda.SQLDAID 791
sqlda.SQLN 791
SQLDB2 REXX API 739, 752
SQLDB2 の登録、REXX の場合の 740
sqldbchar C/C++ タイプ 645
sqldbchar データ・タイプ 430, 431, 433, 639
sqldbchar と wchar_t、データ・タイプの選択 639
SQLDBS REXX API 739
SQLDBS の登録、REXX の場合の 740
SQLE819A インクルード・ファイル
COBOL アプリケーションの 703
C/C++ アプリケーションの 612
FORTRAN アプリケーションの 725
SQLE819B インクルード・ファイル
COBOL アプリケーションの 703
C/C++ アプリケーションの 612
FORTRAN アプリケーションの 725
SQLE850A インクルード・ファイル
COBOL アプリケーションの 703
FORTRAN アプリケーションの 726
SQLE850B インクルード・ファイル
COBOL アプリケーションの 704
FORTRAN アプリケーションの 726
SQLE859A インクルード・ファイル
C/C++ アプリケーションの 612
SQLE859B インクルード・ファイル
C/C++ アプリケーションの 612
SQLE932A インクルード・ファイル
COBOL アプリケーションの 704
C/C++ アプリケーションの 613
FORTRAN アプリケーションの 726
SQLE932B インクルード・ファイル
COBOL アプリケーションの 704
C/C++ アプリケーションの 613
FORTRAN アプリケーションの 726
sqleAttachToCtx() API 558
SQLEAU インクルード・ファイル
COBOL アプリケーションの 703
C/C++ アプリケーションの 612
FORTRAN アプリケーションの 725
sqleBeginCtx() API 558
sqleDetachFromCtx() API 558
sqleEndCtx() API 558
sqleGetCurrentCtx() API 558
sqleInterruptCtx() API 558
SQLENV インクルード・ファイル
COBOL アプリケーションの 703
C/C++ アプリケーションの 612
FORTRAN アプリケーションの 725
SQLERRD(1) 532, 540, 542
SQLERRD(2) 532, 540, 542
SQLERRD(3)
XA 環境における 565
SQLERRMC フィールド、SQLCA の 532, 815, 824
SQLERRP フィールド、SQLCA の 814
sqleSetTypeCtx() API 558
SQLETSD インクルード・ファイル
COBOL アプリケーションの 703
SQLException
検索、SQLCODE の 659
検索、SQLMSG の 659
検索、SQLSTATE の 659
処理 128
SQLEXEC
処理、REXX での SQL ステートメントの 741
SQLEXEC REXX API 739
SQLEXEC の登録、REXX の場合の 740
SQLEXT インクルード・ファイル
CLI アプリケーションの 612
SQLIND フィールド 792
sqlint64 C/C++ タイプ 645
SQLISL 事前定義変数 744

SQLJ (Embedded SQLJ for Java)
JDBC (Java Database Connectivity) との比較 656
SQLJ (Java Embedded SQL)
アプレット 672
位置指定された DELETE ステートメント 674
位置指定された UPDATE ステートメント 674
イテレーターの宣言 674
カーソルの宣言 674
概説 671
ストアード・プロシージャーの呼び出し 680
制約事項 672
プログラムの例 676
文節の例 674
ホスト変数 680
db2profc コマンド 671
db2profp コマンド 671
holdability 674
Java データベース接続機能 (JDBC) 相互運用性 694
profconv コマンド 671
returnability 674
SQL ステートメントの組み込み 674
translator コマンド 671
SQLJ イテレーターの holdability 674
SQLJ イテレーターの returnability 674
SQLJACB インクルード・ファイル
C/C++ アプリケーションの 613
SQLLEN フィールド 792
SQLMON インクルード・ファイル
COBOL アプリケーションの 704
C/C++ アプリケーションの 613
FORTRAN アプリケーションの 726
SQLMONCT インクルード・ファイル
COBOL アプリケーションの 704
SQLMSG
Java プログラムの 659
SQLMSG 事前定義変数 744
SQLRDAT 事前定義変数 744
SQLRIDA 事前定義変数 744
SQLRODA 事前定義変数 744
SQLSTATE
スタンドアロン 817
相違点 819
CLI での 177
Java プログラムの 659
SQLCA の SQLERRMC フィールドの 824
SQLSTATE インクルード・ファイル
COBOL アプリケーションの 704
C/C++ アプリケーションの 613
FORTRAN アプリケーションの 727
SQLSTATE フィールド、エラー・メッセージの 122
SQLSYSTM インクルード・ファイル
C/C++ アプリケーションの 613
SQLTYPE フィールド 792
SQLUDF インクルード・ファイル
説明 435
C/C++ アプリケーションの 613
UDF インターフェース 411
sqludf.h インクルード・ファイル 426
sqludf.h インクルード・ファイル、UDF の 435
sqludf_append API 460
sqludf_create_locator API 460
sqludf_free_locator API 460
sqludf_length API 460
sqludf_substr API 460
SQLUTBCQ インクルード・ファイル
COBOL アプリケーションの 704
SQLUTBSQ インクルード・ファイル
COBOL アプリケーションの 705
SQLUTIL インクルード・ファイル
COBOL アプリケーションの 705
SQLUTIL インクルード・ファイル *(続き)*
C/C++ アプリケーションの 613
FORTRAN アプリケーションの 727
SQLUV インクルード・ファイル
C/C++ アプリケーションの 613
SQLUVEND インクルード・ファイル
C/C++ アプリケーションの 613
SQLVAR エンティティー
十分な数の宣言 152
不定数の宣言 150
SQLWARN 構造の概説 122
SQLXA インクルード・ファイル
C/C++ アプリケーションの 614
SQLZ_DISCONNECT_PROC 戻り値 793
SQLZ_HOLD_PROC 戻り値 793
SQL/DS、DB2 コネクトを使用した 811
SQL_API_FN マクロ 426, 791
SQL_FILE_READ、入力値オプション 382
SQL_WCHART_CONVERT プリプロセッサー・マクロ 640
STATIC.SQB COBOL プログラム・リスト 75
STATIC.SQC C プログラム・リスト 72
Static.sqlj Java プログラム・リスト 73
String 658
SYSCAT.FUNCMAPOPTIONS カタログ視点 601
SYSCAT.FUNCTIONS カタログ視点 601
SYSIBM.SYSPROCEDURES カタログ (OS/390) 821
SYSSTAT.FUNCTIONS カタログ視点 601
System.err Java I/O ストリーム 437
System.in Java I/O ストリーム 437
System.out Java I/O ストリーム 437

T

tablespace-options-clause、CREATE TABLE ステートメントの 367
tfweather_u table 関数の C プログラム・リスト 480
TIME SQL データ・タイプ 84, 445
  COBOL 718
  C/C++ 645
  FORTRAN 734
  Java 658
  Java ストアード・プロシージャー (DB2GENERAL) 794
  OLE DB 表関数 453
  REXX 749
TIME パラメーター、UDF への 432
TIMESTAMP SQL データ・タイプ 84, 445
  COBOL 718
  C/C++ 645
  FORTRAN 734
  Java 658
  Java ストアード・プロシージャー (DB2GENERAL) 794
  OLE DB 表関数 453
  REXX 749
TIMESTAMP パラメーター、UDF への 432
TM_DATABASE マルチサイト更新構成パラメーター 554
TO SQL transform 344
TP_MON_NAME マルチサイト更新構成パラメーター 554
transform 関数
  オブジェクトを外部ルーチンへ渡す 344, 345
  構造型との関連付け 340
  構造型をクライアント・アプリケーションに渡す 350
  サブタイプのバインドイン 359
  サブタイプ・パラメーターの処理 355
  マッピング構造型属性 344
  要約表 354
transform グループ
  外部ルーチン用の 343
  静的 SQL用の 343
  動的 SQL 用の 343
  命名の推奨事項 341
TREAT 式 339
TYPE_ID 関数 330
TYPE_NAME 関数 330
TYPE_SCHEMA 関数 330

U

UCS-2 535
UDF および LOB タイプ 406
UDF と DB2 の間の引き数 412
  関数名 415
  診断メッセージ 416
  スクラッチパッド 416
  特定名 415
  呼び出しタイプ 418
  DB 情報 420
  SQL 結果 412
  SQL 結果標識 413
  SQL 引き数 412
  SQL 引き数標識 413
  SQL-state 414
UDF (ユーザー定義関数)
  一般的な考慮事項 405
  インフィックス表記 405
  インプリメント 390
  インプリメントの手順 394
  および DB2 オブジェクト拡張 289
  概念 393
  画面とキーボードへの入出力 467
  関数参照の要約 405
  関数選択アルゴリズム 393
  関数の参照 401
  関数のタイプ 394
  関数パス 393
  関数名の多重定義 393
  キャスト 407
  共用メモリー・サイズ 467
  警告 467
  コーディング、Java での 437
UDF (ユーザー定義関数) *(続き)*
  コーディングに関するヒントとアドバイス 465
  コード・ページの相違点 467
  再入可能 UDF 456
  作成 395, 409
  作成および使用、Java での 436
  システム構成パラメーター、共用メモリー・サイズのための 467
  修飾されない参照 393
  スキーマ名および UDF 393
  スクラッチパッドに関する考慮事項 456
  制限と警告 467
  ソース派生 300
  タイプのリストおよび UDF 内でのタイプの表示 426
  中国語 (繁体字) コード・セット 539
  定義 389
  デバッグ、UDF の 497
  登録 395
  トリガー、UDT、および LOB との協調 511
  日本語コード・セット 539
  表関数 458
  保管、関数の状態での 456
  保護されたリソースを使用する際の考慮事項 467
  呼び出し 401
    関数内でのパラメーター・マーカー 403
    修飾された関数参照 403
    修飾されない関数参照 404
  呼び出し規則 426
  理論的説明 390
  例 474
  例、UDF コードの 470
  割り振り、UDF 内での動的メモリーの 465
  C++ に関する考慮事項 468
  DB2 から UDF への引き数の受け渡し 411
  DB2 と UDF 間のインターフェース 411
  db2udf 実行可能ファイル 467

UDF (ユーザー定義関数) *(続き)*
DETERMINISTIC 457
EXTERNAL ACTION オプション 465
FENCED オプション 465
Java に関する考慮事項 411
LOB タイプ 406
LOB ロケーター使用のシナリオ 464
LOB ロケーターの使用 459
NOT DETERMINISTIC 456
NOT DETERMINISTIC オプション 465
NOT FENCED 472
NOT NULL CALL 472
NOT NULL CALL オプション 465
OLE オートメーション
UDF 441
SCRATCHPAD 457
SQL データ・タイプの受け渡し方法 426
SQLUDF インクルード・ファイル 411, 435
SQL_API_FN 471
SUBSTR 組み込み関数 477
UDT (ユーザー定義タイプ)
および DB2 オブジェクト拡張 289
トリガー、UDF、および LOB との協調 511
Unicode
Java 693
Unicode (UCS-2)
中国語 (繁体字) コード・セット 535
日本語コード・セット 535
文字変換 548
文字変換によるオーバーフロー 547
UDF に関する考慮事項 539
UPDATE ステートメント
DB2 コネクト・サポート 813
UPDATE 操作とトリガー 502
UPDAT.CMD REXX プログラム・リスト 120
UPDAT.SQB COBOL プログラム・リスト 118
UPDAT.SQC C プログラム・リスト 114
Updat.sqlj Java プログラム・リスト 116
USAGE 文節、COBOL タイプにおける 718
USER MAPPING、OLE DB 表関数の 453
UTILAPI.C プログラム・リスト 126

V

V5SPCLI.SQC C プログラム・リスト 805
V5SPSRV.SQC C プログラム・リスト 809
VALIDATE RUN
DB2 コネクト・サポート 815
VALUES 文節
INSERT ステートメントで 573
VARCHAR 415, 416
VARCHAR FOR BIT DATA パラメーター、UDF への 429
VARCHAR SQL データ・タイプ 84, 445
C または C++ 650
COBOL 718
C/C++ 645
FORTRAN 734
Java 658
Java ストアード・プロシージャー (DB2GENERAL) 794
OLE DB 表関数 453
REXX 749
VARCHAR 構造書式 C/C++ タイプ 645
VARGRAPHIC SQL データ・タイプ 84, 445
COBOL 718
C/C++ 645
FORTRAN 734
Java 658
Java ストアード・プロシージャー (DB2GENERAL) 794
VARGRAPHIC SQL データ・タイプ *(続き)*
OLE DB 表関数 453
REXX 749
VARGRAPHIC データ 650
VARGRAPHIC パラメーター、UDF への 431
Varinp.java Java プログラム・リスト 173
VARINP.SQB COBOL プログラム・リスト 175
VARINP.SQC C プログラム・リスト 171
Visual Basic
DB2 でサポートされる 28
Visual C++
DB2 でサポートされる 28
IBM DB2 Universal Database Project Add-In 33

W

WCHARTYPE
指針 641
ストアード・プロシージャーにおける 242
WCHARTYPE プリコンパイラー・オプション 245, 640
wchar_t C/C++ タイプ 645
wchar_t データ・タイプ 430, 431, 433, 639
wchar_t と sqldbchar、データ・タイプの選択 639
WHENEVER SQLERROR CONTINUE ステートメント 18
WHENEVER ステートメント
エラー処理 123
SQL ステートメントを使用する上での注意 18
SQLCA を使用したエラー標識 17
Windows コード・ページ
サポートされるコード・ページ 523
DB2CODEPAGE レジストリー変数 523

Windows 登録データベース
OLE オートメーション UDF の 442
WITH OPTIONS 文節
参照列の効力範囲の定義　320
を使用した列オプションの定義 320

X

X/Open XA インターフェース　563
カーソル、WITH HOLD と宣言された　565
単一スレッドのアプリケーション 566
特性、トランザクション処理 563
トランザクション　564
保管点　198
マルチスレッド・アプリケーション　566
API 制限　566
CICS 環境　563
COMMIT と ROLLBACK　564
DISCONNECT　564
SET CONNECTION　564
SQL CONNECT　564
XA 環境　565
XASerialize　566

［特殊文字］

#ifdef、C/C++ 言語制限　629
#include マクロ、C/C++ 言語制限 614
#line マクロ、C/C++ 言語制限　614

IBM と連絡をとる

技術上の問題がある場合は、時間をとって*問題判別の手引き* に定義されている処置を検討し、それらの提案を実行した後で、お客様サポートに連絡をとってください。この資料には、お客様サポートがお客様を支援するために必要とする情報が説明されています。

製品情報

以下の情報は英語で提供されます。内容は英語版製品に関する情報です。

http://www.ibm.com/software/data/
: DB2 World Wide Web ページには、ニュース、製品説明、研修スケジュールなどの DB2 に関する最新情報が提供されています。ただし、提供されている情報は英語です。

http://www.ibm.com/software/data/db2/library/
: 「DB2 Product and Service Technical Library」では、よくされる質問 (FAQ)、修正内容、資料、および最新の DB2 技術情報などの情報へのアクセスが提供されています。

注: この情報のご提供は英語のみとなりますのでご注意ください。

http://www.elink.ibmlink.ibm.com/pbl/pbl/
: 「International Publications」注文用 Web サイトでは、マニュアルの注文方法についての情報を提供しています。ただし、提供されている情報は英語です。

http://www.ibm.com/education/certify/
: IBM の「Professional Certification Program」Web サイトでは、 DB2 を含むさまざまな IBM 製品の認証テストの情報を提供しています。ただし、提供されている情報は英語です。

ftp.software.ibm.com
: 匿名でログオンしてください。ディレクトリー `/ps/products/db2` には、 DB2 および多数の他製品に関連したデモ、修正プログラム、情報、およびツールがあります。ただし、提供されている情報は英語です。

comp.databases.ibm-db2, bit.listserv.db2-l
: これらのインターネット・ニュースグループは、ユーザーが DB2 製品に関する自分の経験について話し合うために利用できます。ただし、提供されている情報は英語です。

© Copyright IBM Corp. 1993, 2001

Compuserve: GO IBMDB2

このコマンドを入力すると、IBM DB2 Family forum にアクセスできます。すべての DB2 製品が、このフォーラムでサポートされています。ただし、提供されている情報は英語です。

米国以外の国で IBM に連絡する方法については、 *IBM Software Support Handbook* の Appendix A を参照してください。この資料にアクセスするには、 Web ページ: http://www.ibm.com/support/ にアクセスし、ページの最下部にある「IBM Software Support Handbook」リンク・ボタンを選択します。

注: 国によっては、 IBM が承認している販売業者が、 IBM サポート・センターの代わりにそれら販売業者のサポート・センターに連絡する場合があります。

Printed in Japan

SC88-8516-01

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