SQL Anywhere サーバ
文書バージョン: 17 – 2016-05-11

SQL Anywhere - プログラミング

目次

1 SQL Anywhere サーバ - プログラミング 7

1.1 プログラミングインタフェース 9

インタフェースライブラリ通信プロトコル 10

1.2 SQL を使用したアプリケーション開発 11

アプリケーションでの SQL 文の実行 12

準備文 13

カーソルの使用法 16

カーソルの原則 19

カーソルタイプ 25

カーソルの属性 27

結果セット記述子 45

アプリケーション内のトランザクション 46

1.3 .NET アプリケーションプログラミング 51

SQL Anywhere .NET データプロバイダ 52

.NET データプロバイダチュートリアル 100

SQL Anywhere ASP.NET プロバイダ 118

SQL Anywhere .NET API リファレンス 126

1.4 OLE DB と ADO の開発 127

OLE DB 128

OLE DB プロバイダによる ADO プログラミング 129

OLE DB 接続パラメータ 137

OLE DB 接続プーリング 139

OLE DB 独立性レベル 139

Microsoft リンクサーバ 140

サポートされる OLE DB インタフェース 144

OLE DB プロバイダの登録 148

1.5 ODBC サポート 148

ODBC アプリケーションの開発要件 150

ODBC アプリケーションの開発 152

サンプル ODBC プログラム 157

ODBC ハンドル 159

ODBC 接続関数 161

ODBC によって変更されるサーバオプション 165

SQLSetConnectAttr 拡張接続属性 166

Windows DllMain 関数の考慮事項 . . . 168
SQL 文の実行方法 . . . 169
64 ビット ODBC での考慮事項 . . . 174
データアラインメントの要件 . . . 177
ODBC アプリケーションの結果セット . . . 178
ストアドプロシージャの考慮事項 . . . 184
ODBC エスケープ構文 . . . 185
ODBC のエラー処理 . . . 188
1.6 データベースにおける Java . . . 191
データベースにおける Java の概要 . . . 192
Java のエラー処理 . . . 193
チュートリアル: データベース内の Java の使用 . . . 193
Java クラスをデータベースにインストールする方法 . . . 201
データベース内の Java クラスの特殊な機能 . . . 204
Java VM を起動し、停止する方法 . . . 208
Java VM でのシャットダウンフック . . . 208
1.7 XS JavaScript アプリケーションプログラミング . . . 209
1.8 JDBC サポート . . . 212
JDBC アプリケーション . . . 213
JDBC ドライバ . . . 215
SQL Anywhere JDBC ドライバ . . . 216
jConnect JDBC ドライバ . . . 218
JDBC プログラムの構造 . . . 223
クライアント側 JDBC アプリケーションとサーバ側 JDBC アプリケーションの違い . . . 224
クライアント側サンプル JDBS アプリケーション . . . 225
サーバ側サンプル JDBS アプリケーション . . . 229
JDBC 接続についての注意 . . . 232
JDBC を使用したサーバ側データアクセス . . . 234
JDBC コールバック . . . 244
JDBC エスケープ構文 . . . 247
SQL Anywhere JDBC API のサポート . . . 251
1.9 Node.js アプリケーションプログラミング . . . 251
1.10 Embedded SQL . . . 251
開発プロセスの概要 . . . 254
Embedded SQL プリプロセッサ . . . 255
対応コンパイラ . . . 259
Embedded SQL ヘッダファイル . . . 259
インポートライブラリ . . . 260
Embedded SQL サンプルプログラム . . . 261

Embedded SQL プログラムの構造. 261
Windows での DBLIB の動的ロード. 262
サンプル Embedded SQL プログラム. 263
Embedded SQL のデータ型. 268
Embedded SQL のホスト変数. 272
SQLCA (SQL Communication Area). 281
静的 SQL と動的 SQL. 286
SQLDA (SQL descriptor area). 289
Embedded SQL を使用してデータをフェッチする方法. 299
Embedded SQL を使用したワイド挿入. 305
Embedded SQL を使用したワイド削除. 309
Embedded SQL を使用したワイドマージ. 312
Embedded SQL を使用して long 値を送信し、取得する方法. 316
Embedded SQL での簡単なストアドプロシージャ. 323
Embedded SQL を使用した要求管理. 325
Embedded SQL を使用したデータベースのバックアップ. 326
ライブラリ関数のリファレンス. 326
Embedded SQL 文のまとめ. 372
1.11 SQL Anywhere データベース API for C/C++. 373
SQL Anywhere C API のサポート. 374
SQL Anywhere C API リファレンス. 376
1.12 外部呼び出しインタフェース. 376
外部呼び出しを使ったプロシージャと関数. 377
外部関数のプロトタイプ. 378
外部関数呼び出しインタフェースのメソッド. 392
データ型の処理. 396
外部ライブラリをアンロードする方法. 398
1.13 外部環境のサポート. 398
CLR 外部環境. 399
ESQL 外部環境と ODBC 外部環境. 403
Java 外部環境. 411
JavaScript 外部環境. 416
Perl 外部環境. 421
PHP 外部環境. 425
1.14 Perl DBI サポート. 431
DBD::SQLAnywhere. 431
Windows での DBD::SQLAnywhere のインストール. 432
UNIX と Mac OS X での DBD::SQLAnywhere のインストール. 434
DBD::SQLAnywhere を使用する Perl スクリプト. 437

1.15 Python およびデータベースアクセス . . . 441
Windows での sqlanydb のインストール . . . 442
UNIX と Mac OS X での sqlanydb のインストール . . . 443
Django ドライバ (sqlany-django) のインストール . . . 445
SQLAlchemy ダイアレクト (sqlalchemy-sqlany) のインストール . . . 445
sqlanydb を使用する Python スクリプト . . . 446
1.16 PHP サポート . . . 451
SQL Anywhere PHP 拡張 . . . 451
SQL Anywhere PHP API リファレンス . . . 463
1.17 Ruby サポート . . . 514
Ruby プログラミング . . . 514
SQL Anywhere Ruby API リファレンス . . . 524
1.18 SAP Open Client のサポート . . . 554
Open Client アーキテクチャ . . . 556
Open Client アプリケーション作成に必要なもの . . . 557
Open Client データ型マッピング . . . 557
Open Client アプリケーションでの SQL . . . 559
SQL Anywhere における Open Client の既知の制限 . . . 561
SQL Anywhere を Open Server として使用するためのシステム稼働条件 . . . 562
Open Server としてのデータベースサーバの設定 . . . 563
1.19 OData のサポート . . . 564
OData サーバのアーキテクチャ . . . 565
OData プロトコルの制限事項 . . . 567
OData サーバのセキュリティの考慮事項 . . . 568
OData サーバの設定方法 . . . 570
OData サーバの例 . . . 572
繰返可能要求の設定方法 . . . 572
クロスサイトリクエストフォージェリ攻撃からの保護方法 . . . 574
OData プロデューサのサービスモデルを作成する方法 . . . 575
サードパーティ HTTP サーバ用に OData プロデューサを設定する方法 . . . 576
OSDL 文のリファレンス . . . 579
1.20 HTTP Web サービス . . . 586
HTTP Web サーバとしてのデータベースサーバ . . . 587
Web クライアントを使用した Web サービスへのアクセス . . . 627
Web サービスエラーコードリファレンス . . . 661
HTTP Web サービスの例 . . . 663
1.21 3 層コンピューティングと分散トランザクション . . . 694
3 層コンピューティングのアーキテクチャ . . . 695
分散トランザクション . . . 697

1.22 データベースツールのプログラミング . . . 699
データベースツールインタフェース (DBTools) . . . 699
SQL Anywhere データベースツール C API リファレンス . . . 706
ソフトウェアコンポーネントの終了コード . . . 786
1.23 データベースおよびアプリケーションの配備 . . . 787
配備の種類 . . . 788
ファイルの配布方法 . . . 789
インストールディレクトリとファイル名 . . . 790
Windows 用 *Deployment* ウィザード . . . 793
Windows へのサイレントインストール . . . 797
Linux/UNIX および MAC OS X の Deployment ウィザード . . . 801
Linux/UNIX と Mac OS X でのサイレントインストール . . . 803
クライアントアプリケーションを配備するための要件 . . . 805
管理ツールの配備 . . . 837
マニュアルの配備 . . . 860
データベースサーバの配備 . . . 860
Windows での DLL の登録 . . . 867
外部環境のサポートの配備 . . . 867
暗号化の配備 . . . 870
LDAP の配備 . . . 871
組み込みデータベースアプリケーションの配備 . . . 871
1.24 このマニュアルの印刷、再生、および再配布 . . . 876

1 SQL Anywhere サーバ - プログラミング

このマニュアルでは、C、C++、Java、Perl、PHP、Python、Ruby、および Microsoft .NET 用のプログラミング言語 (Microsoft Visual Basic や Microsoft Visual C# など) を使用してデータベースアプリケーションを構築、配備する方法について説明します。Microsoft ADO.NET、OLE DB、ODBC などのさまざまなプログラミングインタフェースについても説明します。

このセクションの内容:

プログラミングインタフェース [9 ページ]
複数のデータアクセスプログラミングインタフェースが用意されており、使用するアプリケーションとアプリケーション開発環境を自由に選択できます。

SQL を使用したアプリケーション開発 [11 ページ]
ADO.NET、JDBC、ODBC、Embedded SQL、OLE DB、Open Client などの各種アプリケーションプログラミングインタフェースを使用して、アプリケーションから SQL を実行できます。

.NET アプリケーションプログラミング [51 ページ]
SQL Anywhere .NET Data Provider の API を含め、SQL Anywhere を .NET で使用します。

OLE DB と ADO の開発 [127 ページ]
OLE DB および ADO アプリケーションの OLE DB プロバイダがソフトウェアに含まれています。

ODBC サポート [148 ページ]
ODBC (Open Database Connectivity) は、Microsoft が開発した標準 CLI (コールレベルインタフェース) です。SQL Access Group CLI 仕様に基づいています。

データベースにおける Java [191 ページ]
データベースサーバでは、データベース環境内から Java クラスを実行するメカニズムがサポートされています。データベースサーバから Java メソッドを使用すると、強力な方法でプログラミング論理をデータベースに追加できます。

XS JavaScript アプリケーションプログラミング [209 ページ]
SQL Anywhere XS JavaScript ドライバは、SQL Anywhere データベースへの接続、SQL クエリの発行、結果セットの取得に使用できます。

JDBC サポート [212 ページ]
JDBC は、Java アプリケーション用のコールレベルインタフェースです。JDBC を使用すると、さまざまなリレーショナルデータベースに同一のインタフェースでアクセスできます。さらに、高いレベルのツールとインタフェースを構築するため基盤にもなります。

Node.js アプリケーションプログラミング [251 ページ]
Node.js API は、SQL Anywhere データベースへの接続、SQL クエリの発行、結果セットの取得に使用できます。

Embedded SQL [251 ページ]
C または C++ のソースファイルに組み込まれた SQL 文を、Embedded SQL と呼びます。プリプロセッサがそれらの SQL 文をランタイムライブラリの呼び出しに変換します。Embedded SQL は ISO/ANSI および IBM 規格です。

SQL Anywhere データベース API for C/C++ [373 ページ]
SQL Anywhere C アプリケーションプログラミングインタフェース (API) は、C/C++ 言語用のデータアクセス API です。

外部呼び出しインタフェース [376 ページ]
ストアドプロシージャまたは関数から外部ライブラリの関数を呼び出すことができます。

外部環境のサポート [398 ページ]
7 つの外部ランタイム環境がサポートされています。これには、C/C++ で記述された Embedded SQL と ODBC アプリケーション、Java、JavaScript、Perl、PHP、または Microsoft .NET Framework Common Language Runtime (CLR) に基づく Microsoft C# や Microsoft Visual Basic などの言語で記述されたアプリケーションが含まれます。

Perl DBI サポート [431 ページ]
DBD::SQLAnywhere は Perl Database Independent Interface (DBI) 用のデータベースドライバで、Perl 言語用のデータアクセス API です。

Python およびデータベースアクセス [441 ページ]
Python 拡張モジュール sqlanydb、sqlany-django、sqlalchemy-sqlany は、データベースに接続し、SQL クエリを発行し、結果セットを取得するときに Python の使用をサポートします。

PHP サポート [451 ページ]
PHP を使用してデータベースアプリケーションを開発できます。

Ruby サポート [514 ページ]
Ruby を使用してデータベースアプリケーションを開発できます。

SAP Open Client のサポート [554 ページ]
SAP Open Client は、カスタマアプリケーション、サードパーティ製品、その他の SAP 製品に対して、Open Server との通信に必要なインタフェースを提供します。

OData のサポート [564 ページ]
OData (Open Data Protocol) は、RESTful HTTP を介したデータサービスを有効にします。URI (Uniform Resource Identifiers) を通して操作を行い、情報にアクセスして情報を変更できるようになります。

HTTP Web サービス [586 ページ]
SQL Anywhere を使用して、HTTP Web サービスを作成できます。また、他の Web サーバとの間で、要求の送信と応答の受信を行うことができます。

3 層コンピューティングと分散トランザクション [694 ページ]
データベースサーバは、トランザクションサーバによって調整された分散トランザクションに関わるリソースマネージャとして使用できます。

データベースツールのプログラミング [699 ページ]
バックアップやアンロードなどの一般的なデータベースタスクを実装するアプリケーションを C または C++ で記述することができます。

データベースおよびアプリケーションの配備 [787 ページ]
データベースアプリケーションを完了したら、エンドユーザにアプリケーションを配備します。

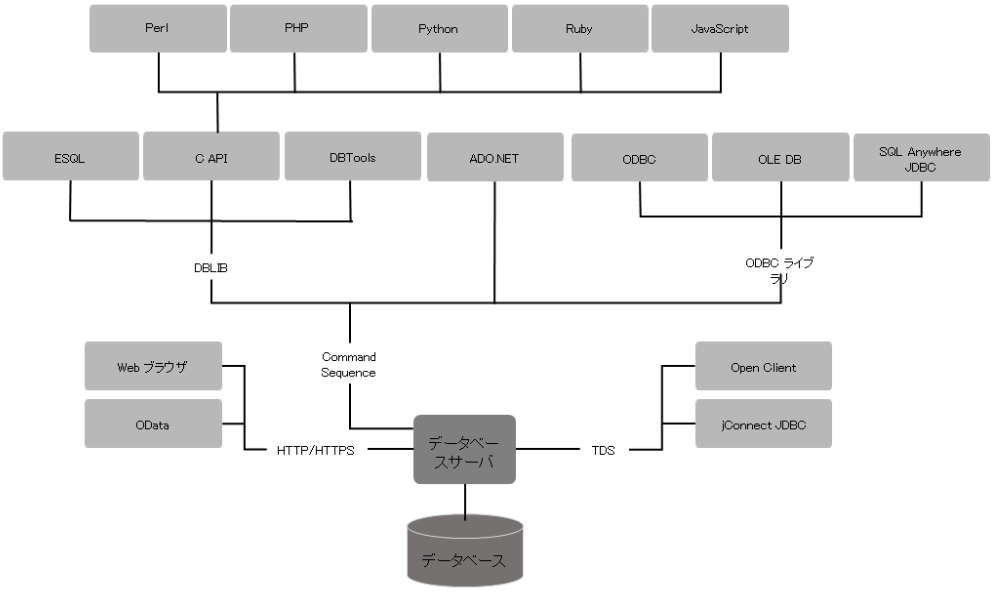
このマニュアルの印刷、再生、および再配布 [876 ページ]
次の条件に従うかぎり、このマニュアルの全部または一部を使用、印刷、再生、配布することができます。

1.1 プログラミングインタフェース

複数のデータアクセスプログラミングインタフェースが用意されており、使用するアプリケーションとアプリケーション開発環境を自由に選択できます。

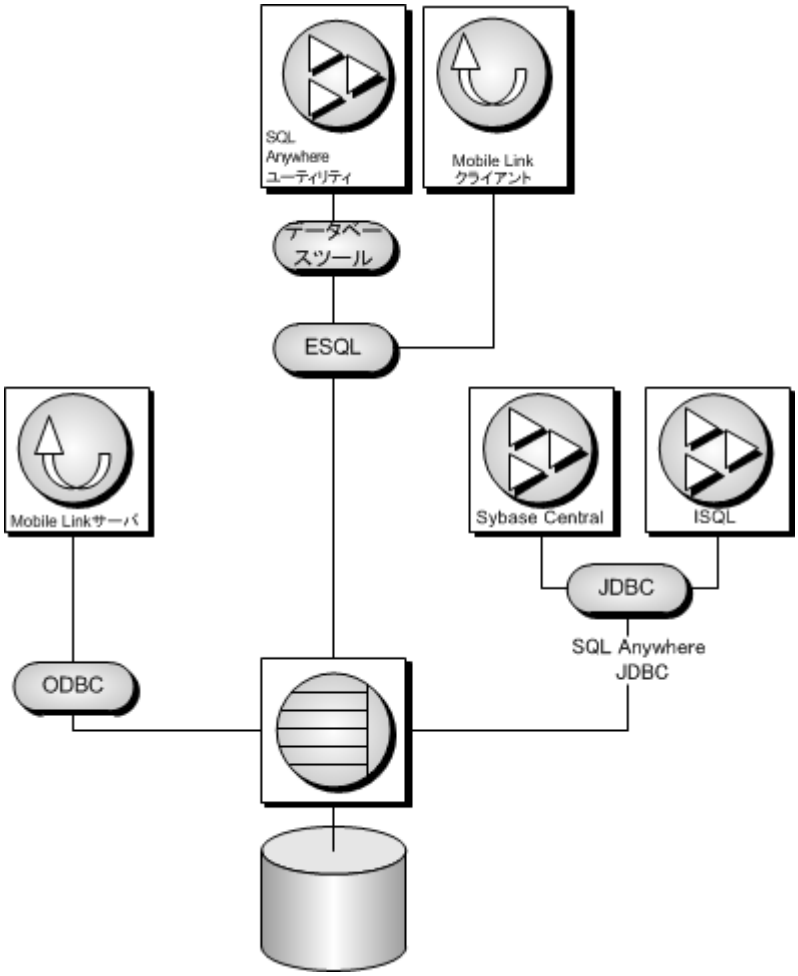
サポートされる SQL Anywhere プログラミングインタフェースとプロトコル

次の図はサポートされるインタフェースと使用されるインタフェースライブラリを表示しています。通常、インタフェースライブラリとインタフェースの名前は同じです。



SQL Anywhere アプリケーション

SQL Anywhere に付属のアプリケーションは、これらのインタフェースのいくつかを使用します。



このセクションの内容:

インタフェースライブラリ通信プロトコル [10 ページ]
各インタフェースライブラリは、1 つの通信プロトコルを使用してデータベースサーバと通信します。

1.1.1 インタフェースライブラリ通信プロトコル

各インタフェースライブラリは、1 つの通信プロトコルを使用してデータベースサーバと通信します。

SQL Anywhere は、**Command Sequence** (**CmdSeq**) と **Tabular Data Stream** (**TDS**) という 2 つの通信プロトコルをサポートしています。これらのプロトコルは内部に組み込まれているので、ほとんどの場合どちらを使用しても変わりがありません。開発環境は、プロトコルではなく利用できるツールによって異なります。

2 つのプロトコルの主な相違は、データベースと接続してみるとわかります。Command Sequence を使用するアプリケーションと TDS を使用するアプリケーションは、データベースとデータベースサーバを識別する方法が異なるので、接続パラメータが異なります。

Command Sequence

このプロトコルは、SQL Anywhere、SQL Anywhere JDBC ドライバ、および Embedded SQL、ODBC、OLE DB、ADO.NET の各 API で使用されます。クライアント側データの転送は、**CmdSeq** プロトコルでサポートされています。

TDS

このプロトコルは、SAP Adaptive Server Enterprise、jConnect JDBC ドライバ、および SAP Open Client の各アプリケーションで使用されます。クライアント側データの転送は、**TDS** プロトコルではサポートされていません。

1.2 SQL を使用したアプリケーション開発

ADO.NET、JDBC、ODBC、Embedded SQL、OLE DB、Open Client などの各種アプリケーションプログラミングインタフェースを使用して、アプリケーションから SQL を実行できます。

このセクションの内容:

アプリケーションでの SQL 文の実行 [12 ページ]
アプリケーションに SQL 文をインクルードする方法は、使用するアプリケーション開発ツールとプログラミングインタフェースによって異なります。

準備文 [13 ページ]
文がデータベースへ送信されるたびに、データベースサーバは一連の手順を実行する必要があります。

カーソルの使用法 [16 ページ]
アプリケーションでクエリを実行すると、結果セットが複数のローで構成されます。通常は、アプリケーションが受け取るローの数は、クエリを実行するまでわかりません。

カーソルの原則 [19 ページ]
カーソルを使用するには、いくつかの基本的な手順に従います。

カーソルタイプ [25 ページ]
各種プログラミングインタフェースが、データベースカーソルの全側面をサポートしているわけではありません。用語が異なっている可能性もあります。プログラミングインタフェースごとに利用可能なカーソルとオプションの間のマッピングについては、次の資料で説明しています。

カーソルの属性 [27 ページ]
カーソルが開くと結果セットに関連付けられます。一度開いたカーソルは一定時間開いたままになります。カーソルが開いている間、カーソルに関連付けられた結果セットは変更される可能性があります。変更は、カーソル自体を使用して行われるか、独立性レベルの稼働条件に基づいて他のトランザクションで行われます。

結果セット記述子 [45 ページ]
アプリケーションによっては、アプリケーション内で完全に指定できない SQL 文を構築するものがあります。表示するカラムをユーザが選択できるレポートアプリケーションなど、文がユーザからの応答に依存していて、ユーザの応答がないと、検索する情報をアプリケーションが正確に把握できない場合があります。

アプリケーション内のトランザクション [46 ページ]
トランザクションはアトミックな SQL 文をまとめたものです。トランザクション内の文はすべて実行されるか、どれも実行されないかのどちらかです。

1.2.1 アプリケーションでの SQL 文の実行

アプリケーションに SQL 文をインクルードする方法は、使用するアプリケーション開発ツールとプログラミングインタフェースによって異なります。

ADO.NET

さまざまな ADO.NET オブジェクトを使用して、SQL 文を実行できます。SACommand オブジェクトはその 1 つの例です。

```
SACommand cmd = new SACommand(
     "DELETE FROM Employees WHERE EmployeeID = 105", conn );
cmd.ExecuteNonQuery();
```

ODBC

ODBC プログラミングインタフェースに直接書き込む場合、関数呼び出し部分に SQL 文を記述します。たとえば、次の C 言語の関数呼び出しは DELETE 文を実行します。

```
SQLExecDirect( stmt,
    "DELETE FROM Employees
     WHERE EmployeeID = 105",
    SQL_NTS );
```

JDBC

JDBC プログラミングインタフェースを使っている場合、statement オブジェクトのメソッドを呼び出して SQL 文を実行できます。次に例を示します。

```
stmt.executeUpdate(
    "DELETE FROM Employees
     WHERE EmployeeID = 105" );
```

Embedded SQL

Embedded SQL を使っている場合、キーワード EXEC SQL を C 言語の SQL 文の前に置きます。次にコードをプリプロセッサに通してから、コンパイルします。次に例を示します。

```
EXEC SQL EXECUTE IMMEDIATE
 'DELETE FROM Employees
  WHERE EmployeeID = 105';
```

SAP Open Client

SAP Open Client インタフェースを使っている場合、関数呼び出し部分に SQL 文を記述します。たとえば、次の一組の呼び出しは DELETE 文を実行します。

```
ret = ct_command( cmd, CS_LANG_CMD,
                  "DELETE FROM Employees
                   WHERE EmployeeID=105"
                 CS_NULLTERM,
                 CS_UNUSED);
ret = ct_send(cmd);
```

アプリケーションに SQL をインクルードする方法の詳細については、使用している開発ツールのマニュアルを参照してください。ODBC または JDBC を使っている場合、そのインタフェース用ソフトウェア開発キットを調べてください。

データベースサーバ内のアプリケーション

ストアドプロシージャとトリガは、データベースサーバ内で実行するアプリケーションまたはその一部として、さまざまな方法で動作します。この場合、ストアドプロシージャの多くのテクニックも使用できます。

データベース内の Java クラスはサーバ外部の Java アプリケーションとまったく同じ方法で JDBC インタフェースを使用できます。

関連情報

SQL Anywhere .NET データプロバイダ [52 ページ]
ODBC サポート [148 ページ]
JDBC サポート [212 ページ]
Embedded SQL [251 ページ]
SAP Open Client のサポート [554 ページ]
JDBC サポート [212 ページ]

1.2.2 準備文

文がデータベースへ送信されるたびに、データベースサーバは一連の手順を実行する必要があります。

- 文を解析し、内部フォームに変換します。このプロセスは文の準備とも呼ばれます。
- データベースオブジェクトへの参照がすべて正確であるかどうかを確認します。たとえば、クエリで指定されたカラムが実際に存在するかどうかをチェックします。
- 文にジョインまたはサブクエリが含まれている場合、クエリオプティマイザがアクセスプランを生成します。
- これらすべての手順を実行してから文を実行します。

準備文の再使用によるパフォーマンスの改善

同じ文を繰り返し使用します (たとえば、1 つのテーブルに多くのローを挿入します) 場合、文の準備を繰り返し行うことにより著しいオーバヘッドが生じます。このオーバヘッドを解消するため、データベースプログラミングインタフェースによっては、準備文の使用方法を提示するものもあります。準備文とは、一連のプレースホルダを含む文です。文を実行するときには、文全体を何度も準備せずに、プレースホルダに値を割り当てます。

たくさんのローを挿入するときなど、同じ動作を何度も繰り返す場合は、準備文を使用すると便利です。

通常、準備文を使用するには次の手順が必要です。

文を準備する

ここでは通常、値の代りにプレースホルダを文に入力します。

準備文を繰り返し実行する

ここでは、文を実行するたびに、使用する値を入力します。実行するたびに文を準備する必要ありません。

文を削除する

ここでは、準備文に関連付けられたリソースを解放します。この手順を自動的に処理するプログラミングインタフェースもあります。

一度だけ使用する文は準備しない

通常、一度だけの実行には文を準備しません。準備と実行を別々に行うと、わずかではあってもパフォーマンスペナルティが生じ、アプリケーションに不要な煩雑さを招きます。

ただし、インタフェースによっては、カーソルに関連付けするために文を準備する必要があります。

文の準備と実行命令の呼び出しは SQL の一部ではなく、インタフェースによって異なります。各プログラミングインタフェースによって、準備文を使用する方法が示されます。

このセクションの内容:

準備文の概要 [14 ページ]
準備文を使用する一般的な手順は同じですが、詳細はインタフェースごとに異なります。

関連情報

カーソルの使用法 [16 ページ]

1.2.2.1 準備文の概要

準備文を使用する一般的な手順は同じですが、詳細はインタフェースごとに異なります。

異なるインタフェースでの準備文の使用方法を比較すると、この違いがはっきりわかります。

準備文を使用するには、一般的に次のタスクを実行します。

1. 文を準備します。
2. 文中の値を保持するパラメータをバインドします。
3. 文中のバウンドパラメータに値を割り当てます。
4. 文を実行します。
5. 必要に応じて手順 3 と 4 を繰り返します。
6. 終了したら、文を削除します。JDBC では、Java ガーベジコレクションメカニズムにより文が削除されます。

ADO.NET での準備文の使用

ADO.NET で準備文を使用するには、一般的に次のタスクを実行します。

1. 文を保持する SACommand オブジェクトを作成します。

```
SACommand cmd = new SACommand(
     "SELECT * FROM Employees WHERE Surname = ?", conn );
```

2. 文中のパラメータのデータ型を宣言します。
   SACommand.CreateParameter メソッドを使用します。

```
SAParameter param = cmd.CreateParameter();
param.SADbType = SADbType.Char;
param.Direction = ParameterDirection.Input;
param.Value = "Smith";
cmd.Parameters.Add(param);
```

3. Prepare メソッドを使って文を準備します。
4. 次の文を実行します。

```
SADataReader reader = cmd.ExecuteReader();
```

ADO.NET を使用して文を準備する例については、*%SQLANYSAMP17%*¥SQLAnywhere¥ADO.NET¥SimpleWin32 にあるソースコードを参照してください。

ODBC での準備文の使用

ODBC で準備文を使用するには、一般的に次のタスクを実行します。

1. SQLPrepare を使って文を準備します。
2. SQLBindParameter を使って文のパラメータをバインドします。
3. SQLExecute を使って文を実行します。
4. SQLFreeStmt を使って文を削除します。

ODBC を使用して文を準備する例については、*%SQLANYSAMP17%*¥SQLAnywhere¥ODBCPrepare にあるソースコードを参照してください。

ODBC 準備文の詳細については、ODBC SDK のマニュアルを参照してください。

JDBC での準備文の使用

JDBC で準備文を使用するには、一般的に次のタスクを実行します。

1. 接続オブジェクトの prepareStatement メソッドを使って文を準備します。これによって準備文オブジェクトが返されます。
2. 準備文オブジェクトの適切な *set*Type メソッドを使って文パラメータを設定します。Type は割り当てられるデータ型です。
3. 準備文オブジェクトの適切なメソッドを使って文を実行します。挿入、更新、削除には、executeUpdate メソッドを使います。

JDBC を使用して文を準備する例については、ソースコードファイル *%SQLANYSAMP17%*¥SQLAnywhere¥JDBC¥JDBCExample.java を参照してください。

Embedded SQL での準備文の使用

Embedded SQL で準備文を使用するには、一般的に次のタスクを実行します。

1. EXEC SQL PREPARE 文を使用して文を準備します。
2. 文中のパラメータに値を割り当てます。
3. EXEC SQL EXECUTE 文を使用して文を実行します。
4. EXEC SQL DROP 文を使用して、その文に関連するリソースを解放します。

Open Client での準備文の使用

Open Client で準備文を使用するには、一般的に次のタスクを実行します。

1. CS_PREPARE 型パラメータで ct_dynamic 関数を使用して文を準備します。
2. ct_param を使用して文のパラメータを設定します。
3. CS_EXECUTE を type パラメータに指定した ct_dynamic を使用して文を実行します。
4. CS_DEALLOC を type パラメータに指定した ct_dynamic を使用して文に関連付けられたリソースを解放します。

関連情報

Open Client アプリケーションでの SQL [559 ページ]
より効率的なアクセスのために準備文を使用する方法 [238 ページ]
準備文の実行 [172 ページ]

1.2.3 カーソルの使用法

アプリケーションでクエリを実行すると、結果セットが複数のローで構成されます。通常は、アプリケーションが受け取るローの数は、クエリを実行するまでわかりません。

カーソルを使うと、アプリケーションでクエリの結果セットを処理する方法が提供されます。カーソルを使用する方法と使用可能なカーソルの種類は、使用するプログラミングインタフェースによって異なります。

接続で使用されているカーソルとその内容を判断するのに役立つシステムプロシージャがいくつかあります。

- sa_list_cursors システムプロシージャ
- sa_describe_cursor システムプロシージャ
- sa_copy_cursor_to_temp_table システムプロシージャ

カーソルを使うと、プログラミングインタフェースで次のようなタスクを実行できます。

- クエリの結果をループします。
- 結果セット内の任意の場所で基本となるデータの挿入、更新、削除を実行します。

プログラミングインタフェースによっては、特別な機能を使用して、結果セットを自分のアプリケーションに返す方法をチューニングできるものもあります。この結果、アプリケーションのパフォーマンスは大きく向上します。

このセクションの内容:

カーソル [17 ページ]
カーソルとは、結果セットに関連付けられた名前です。結果セットは、SELECT 文かストアドプロシージャ呼び出しによって取得されます。

カーソルを使用する利点 [18 ページ]
サーバ側のカーソルはデータベースアプリケーションでは必要ありませんが、いくつかの利点があります。

関連情報

カーソルの可用性 [25 ページ]

1.2.3.1 カーソル

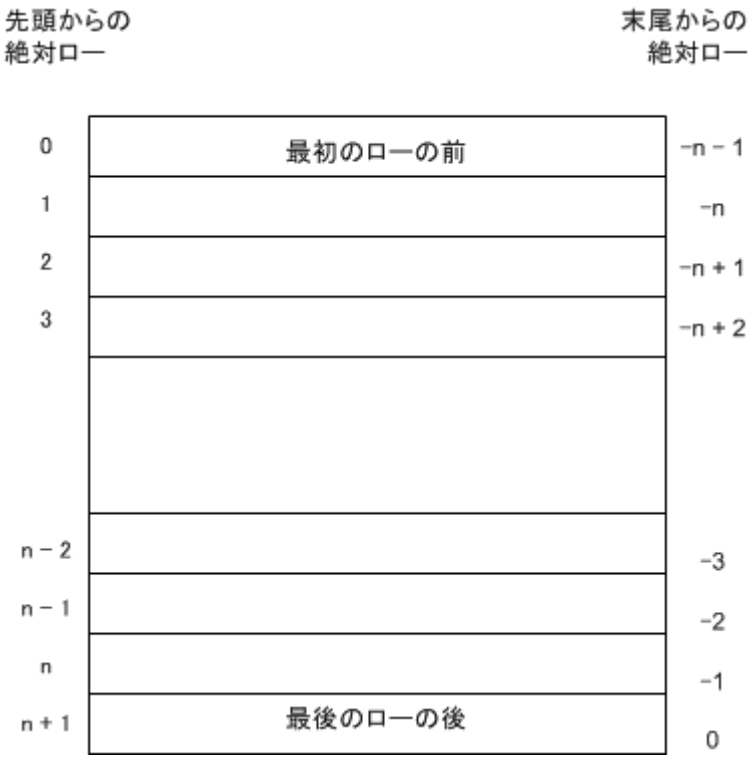
カーソルとは、結果セットに関連付けられた名前です。結果セットは、SELECT 文かストアドプロシージャ呼び出しによって取得されます。

カーソルは、結果セットのハンドルです。カーソルには、結果セット内の適切に定義された位置が必ず指定されています。カーソルを使用すると、1 回につき 1 つのローのデータを調べて操作できます。カーソルでは、クエリ結果内で前方または後方に移動できます。

カーソル位置

カーソルは、次の場所に置くことができます。

- 結果セットの最初のローの前
- 結果セット内の 1 つのロー上
- 結果セットの最後のローの後



カーソル位置と結果セットは、データベースサーバで管理されます。ローは、クライアントによってフェッチされて、1 回に 1 つまたは複数のローを表示して処理できます。結果セット全体がクライアントに配信される必要はありません。

1.2.3.2 カーソルを使用する利点

サーバ側のカーソルはデータベースアプリケーションでは必要ありませんが、いくつかの利点があります。

次の理由により、サーバ側のカーソルはクライアント側のカーソルより推奨されます。

応答時間

サーバ側のカーソルでは、最初のローがクライアントによってフェッチされる前に、結果セット全体をアセンブルする必要がありません。クライアント側のカーソルでは、最初のローがクライアントによってフェッチされる前に、結果セット全体を取得して転送する必要があります。

クライアント側メモリ

結果セットのサイズが大きい場合、結果セット全体をクライアント側で取得すると、クライアントに必要なメモリ容量が増えることがあります。

同時実行性の制御

アプリケーションでデータを更新する場合にサーバ側のカーソルを使用しない場合、変更を適用するために UPDATE、INSERT、DELETE などの別の SQL 文をデータベースサーバに送信します。この方法では、結果セットがクライアントに送信されたためにデータベースの対応するローが変わった場合には、同時実行性の問題が生じる可能性があります。結果として、他のクライアントによる更新が失われる可能性があります。

サーバ側のカーソルは基本となるデータへのポインタとして機能します。したがって、適切な独立性レベルを設定することによって、クライアントが加えた変更に適切な同時性制約を課すことができます。

1.2.4 カーソルの原則

カーソルを使用するには、いくつかの基本的な手順に従います。

1. 文を準備して実行します。
   インタフェースの通常の方法を使用して文を実行します。文を準備して実行するか、文を直接実行します。
   ADO.NET の場合、SACommand.ExecuteReader メソッドのみがカーソルを返します。このコマンドは、読み込み専用、前方専用のカーソルを提供します。
2. 文が結果セットを返すかどうかを確認するためにテストします。
   結果セットを作成する文を実行する場合、カーソルは暗黙的に開きます。カーソルが開かれると、結果セットの第 1 ローの前に配置されます。
3. 結果をフェッチします。
   簡単なフェッチ操作を行うと、結果セット内の次のローにカーソルが移動しますが、結果セット内でより複雑な移動も可能です。
4. カーソルを閉じます。
   カーソルでの作業が終了したら、閉じて関連するリソースを解放します。
5. 文を解放します。
   準備した文を使った場合は、それを解放してメモリを再利用します。

Embedded SQL でカーソルを使用するためのアプローチは、他のインタフェースで使用されるアプローチと異なります。
Embedded SQL でカーソルを使用するには、これらの一般的な手順に従います。

1. 文を準備します。
   通常、カーソルでは文字列ではなくステートメントハンドルが使用されます。ハンドルを使用可能にするために、文を準備します。
2. カーソルを宣言します。
   各カーソルは、単一の SELECT 文か CALL 文を参照します。カーソルを宣言するとき、カーソル名と参照した文を入力します。
3. カーソルを開きます。
   CALL 文の場合、カーソルを開くと、1 番目のローが取得されるポイントまでプロシージャが実行されます。
4. 結果をフェッチします。
   簡単なフェッチ操作を行うと、結果セット内の次のローにカーソルが移動しますが、結果セット内でより複雑な移動も可能です。どのフェッチ操作が実行可能であるかは、カーソルの宣言方法によって決定されます。
5. カーソルを閉じます。
   カーソルでの作業が終わったら、カーソルを閉じます。これにより、カーソルに関連付けられているリソースが解放されます。
6. 文を削除します。
   文に関連付けられているメモリを解放するには、文を削除する必要があります。

このセクションの内容:

カーソル位置 [20 ページ]
カーソルを開くと最初のローの前に置かれます。カーソル位置は、クエリ結果の最初か最後を基準とした絶対位置、または現在のカーソル位置を基準とした相対位置に移動できます。

カーソルを開くときのカーソルの動作 [21 ページ]
カーソルを開くときに、カーソルの独立性レベルと期間を設定できます。

カーソルによるローのフェッチ [21 ページ]
カーソルを使用してクエリの結果セットをもっとも簡単に処理するには、ローがなくなるまで結果セットのすべてのローをループします。

複数ローのフェッチ [22 ページ]
複数ローのフェッチとローのプリフェッチを混同しないでください。複数のローのフェッチはアプリケーションによって実行されます。一方、プリフェッチはアプリケーションに対して透過的で、同様にパフォーマンスが向上します。

スクロール可能なカーソル [22 ページ]
ODBC と Embedded SQL では、スクロール可能なカーソルと、スクロール可能で動的なカーソルを使う方法があります。これらの方法では、結果セット内で一度にローをいくつか前方または後方へ移動できます。

ローの修正に使用するカーソル [22 ページ]
カーソルには、クエリから結果セットを読み込む以外にも可能なことがあります。カーソルの処理中に、データベース内のデータ修正もできます。

更新可能な文 [23 ページ]
SELECT 文の句は、更新可能な文とカーソルにさまざまな影響を与える可能性があります。

キャンセルされるカーソル操作 [25 ページ]
インタフェース機能で要求をキャンセルできます。

関連情報

準備文 [13 ページ]
Embedded SQL を使用してデータをフェッチする方法 [299 ページ]

1.2.4.1 カーソル位置

カーソルを開くと最初のローの前に置かれます。カーソル位置は、クエリ結果の最初か最後を基準とした絶対位置、または現在のカーソル位置を基準とした相対位置に移動できます。

カーソル位置の変更方法とカーソルで可能な操作は、プログラミングインタフェースが制御します。

カーソルでフェッチできるローの位置番号は、integer 型のサイズによって管理されます。integer に格納できる値より 1 小さい 2147483646 までの番号が付けられたローをフェッチできます。ローの位置番号に、クエリ結果の最後を基準として負の数を使用している場合、integer に格納できる負の最大値より 1 大きい数までの番号のローをフェッチできます。

現在のカーソル位置でローを更新または削除するには、位置付け更新と位置付け削除という特別な操作を使用できます。先頭のローの前か、末尾のローの後にカーソルがある場合、対応するカーソルローがないことを示すエラーが返されます。

注記

asensitive カーソルに挿入や更新をいくつか行うと、カーソル位置に問題が生じます。SELECT 文に ORDER BY 句を指定しないかぎり、挿入された行はカーソル内の予測可能な位置に配置されません。場合によっては、カーソルを閉じてもう一度開かないと、挿入したローがまったく表示されないことがあります。これは、カーソルを開くためにワークテーブルを作成する必要がある場合に起こります。

UPDATE 文によって、カーソル内のローが移動することがあります。これは、既存のインデックスを使用する ORDER BY 句がカーソルに指定されている場合に発生します (ワークテーブルは作成されません)。STATIC SCROLL カーソルを使うとこの問題はなくなりますが、より資源を消費します。

1.2.4.2 カーソルを開くときのカーソルの動作

カーソルを開くときに、カーソルの独立性レベルと期間を設定できます。

独立性レベル

カーソルに操作の独立性レベルを明示的に設定して、トランザクションの現在の独立性レベルと区別できます。これを行うには、isolation_level オプションを設定します。

期間

デフォルトでは、Embedded SQL のカーソルはトランザクションの終了時に閉じます。WITH HOLD でカーソルを開くと、接続終了まで、または明示的に閉じるまでカーソルを開いたままにできます。ADO.NET、ODBC、JDBC、Open Client では、デフォルトでトランザクションの終了時までカーソルを開いたままにします。

1.2.4.3 カーソルによるローのフェッチ

カーソルを使用してクエリの結果セットをもっとも簡単に処理するには、ローがなくなるまで結果セットのすべてのローをループします。

これらの手順を実行すると、このタスクを実行できます。

1. カーソル (Embedded SQL) を宣言して開くか、結果セット (ODBC、JDBC、Open Client) または SADataReader オブジェクト (ADO.NET) を返す文を実行します。
2. 「`Row Not Found`」(ローが見つかりません) というエラーが表示されるまで、次のローをフェッチし続けます。
3. カーソルを閉じます。

次のローをフェッチするために使用する方法は、使用するインタフェースによって異なります。次に例を示します。

ADO.NET

SADataReader.Read メソッドを使用します。

ODBC

SQLFetch、SQLExtendedFetch、または SQLFetchScroll が次のローにカーソルを進め、データを返します。

JDBC

ResultSet オブジェクトの next メソッドがカーソルを進め、データを返します。

Embedded SQL

FETCH 文が同じ操作を実行します。

Open Client

ct_fetch 関数が次のローにカーソルを進め、データを返します。

関連情報

ODBC アプリケーションの結果セット [178 ページ]
Java から結果セットを返す方法 [242 ページ]

Embedded SQL でのカーソル [300 ページ]
Open Client カーソルの管理 [560 ページ]

1.2.4.4 複数ローのフェッチ

複数ローのフェッチとローのプリフェッチを混同しないでください。複数のローのフェッチはアプリケーションによって実行されます。一方、プリフェッチはアプリケーションに対して透過的で、同様にパフォーマンスが向上します。

一度に複数のローをフェッチすると、パフォーマンスを向上させることができます。

インタフェースによっては、配列内の次のいくつかのフィールドへ複数のローを一度にフェッチすることができます。一般的に、実行する個々のフェッチ操作が少なければ少ないほど、サーバが応答する個々の要求が少なくなり、パフォーマンスが向上します。複数のローを取り出すように修正された FETCH 文をワイドフェッチと呼ぶこともあります。複数のローのフェッチを使うカーソルは、ブロックカーソルまたはファットカーソルとも呼ばれます。

複数ローフェッチの使用

- ODBC では、SQL_ATTR_ROW_ARRAY_SIZE 属性または SQL_ROWSET_SIZE 属性を設定して、SQLFetchScroll または SQLExtendedFetch をそれぞれ呼び出したときに返されるローの数を設定できます。
- Embedded SQL では、FETCH 文で ARRAY 句を使用して、一度にフェッチされるローの数を制御します。
- Open Client と JDBC は複数のローのフェッチをサポートしません。これらのインタフェースではプリフェッチを使用します。

1.2.4.5 スクロール可能なカーソル

ODBC と Embedded SQL では、スクロール可能なカーソルと、スクロール可能で動的なカーソルを使う方法があります。これらの方法では、結果セット内で一度にローをいくつか前方または後方へ移動できます。

JDBC または Open Client インタフェースではスクロール可能なカーソルはサポートされていません。

プリフェッチはスクロール可能な操作には適用されません。たとえば、逆方向へのローのフェッチにより、前のローがいくつかプリフェッチされることはありません。

1.2.4.6 ローの修正に使用するカーソル

カーソルには、クエリから結果セットを読み込む以外にも可能なことがあります。カーソルの処理中に、データベース内のデータ修正もできます。

この操作は一般に位置付け挿入、更新、削除の操作と呼ばれます。また、挿入操作の場合は、これを PUT 操作ともいいます。

すべてのクエリの結果セットで、位置付け更新と削除ができるわけではありません。更新不可のビューにクエリを実行すると、基本となるテーブルへの変更は行われません。また、クエリがジョインを含む場合、ローの削除を行うテーブルまたは更新するカラムを、操作の実行時に指定してください。

テーブル内の任意の挿入されていないカラムに NULL を入力できるかデフォルト値が指定されている場合だけ、カーソルを使った挿入を実行できます。

複数のローが value-sensitive (キーセット駆動型) カーソルに挿入される場合、これらのローはカーソル結果セットの最後に表示されます。これらのローは、クエリの WHERE 句と一致しない場合や、ORDER BY 句が通常、これらを結果セットの別の場所に配置した場合でも、カーソル結果セットの最後に表示されます。この動作はプログラミングインタフェースとは関係ありません。たとえば、この動作は、Embedded SQL の PUT 文または ODBC SQLBulkOperations 関数を使用するときに適用されます。挿入された最新のローの AUTOINCREMENT カラムの値は、カーソルの最後のローを選択することによって確認できます。たとえば、Embedded SQL の場合、`FETCH ABSOLUTE -1 cursor-name` を使用することで値を取得できます。この動作のため、value-sensitive カーソルに対する最初の複数のローの挿入は負荷が大きくなる可能性があります。

ODBC、JDBC、Embedded SQL、Open Client では、カーソルを使ったデータ操作が許可されますが、ADO.NET では許可されません。Open Client の場合、ローの削除と更新はできますが、ローの挿入は単一テーブルのクエリだけです。

どのテーブルからローを削除するか

カーソルを使って位置付け削除を試行する場合、ローを削除するテーブルは次のように決定されます。

1. DELETE 文に FROM 句が含まれない場合、カーソルは単一テーブルだけにあります。
2. カーソルがジョインされたクエリ用の場合 (ジョインがあるビューの使用を含めて)、FROM 句が使われます。指定したテーブルの現在のローだけが削除されます。ジョインに含まれた他のテーブルは影響を受けません。
3. FROM 句が含まれ、テーブル所有者が指定されない場合、テーブル仕様値がどの相関名に対しても最初に一致します。
4. 相関名がある場合、テーブル仕様値は相関名で識別されます。
5. 相関名がない場合、テーブル仕様値はカーソルのテーブル名として明確に識別可能にします。
6. FROM 句が含まれ、テーブル所有者が指定されている場合、テーブル仕様値はカーソルのテーブル名として明確に指定可能にします。
7. 位置付け DELETE 文はビューでカーソルを開くときに使用できます。ただし、ビューが更新可能である場合にかぎられます。

1.2.4.7 更新可能な文

SELECT 文の句は、更新可能な文とカーソルにさまざまな影響を与える可能性があります。

読み込み専用文の更新可能性

カーソル宣言で FOR READ ONLY を指定するか、FOR READ ONLY 句を文に含めると、文は読み込み専用になります。FOR READ ONLY 句を指定するか、クライアント API を使用している場合に読み込み専用カーソルを宣言すると、その他の更新可能性の指定は上書きされます。

SELECT 文の最も外側のブロックに ORDER BY 句が含まれている場合、文で FOR UPDATE を指定しないと、カーソルは読み込み専用になります。SQL の SELECT 文で FOR XML を指定すると、カーソルは読み込み専用になります。それ以外の場合、カーソルは更新可能です。

更新可能な文と同時制御

更新可能な文の場合、カーソルに対してオプティミスティックとペシミスティックの両方の同時実行性制御メカニズムがあり、スクロール操作中の結果セットの一貫性が保たれます。これらのメカニズムは、セマンティックとトレードオフは異なりますが、INSENSITIVE カーソルやスナップショットアイソレーションに代わる方法です。

FOR UPDATE の指定は、カーソルの更新可能性に影響する場合があります。ただし、FOR UPDATE 構文には、同時実行性制御に対するその他の影響はありません。FOR UPDATE で追加のパラメータを指定すると、次の 2 つの同時制御オプションのいずれかを組み込むように文の処理が変更されます。

ペシミスティック

カーソルの結果セットにフェッチされたすべてのローに対して、データベースサーバは意図的ローロックを取得して、別のトランザクションによってローが更新されないようにします。

オプティミスティック

データベースサーバで使用されるカーソルのタイプがキーセット駆動型カーソル (insensitive ローメンバーシップ、value-sensitive) に変えられ、結果内のローが任意のトランザクションによって変更または削除されると、アプリケーションに通知されるようになります。

ペシミスティックまたはオプティミスティック同時実行性は、DECLARE CURSOR 文または FOR 文のオプション、または特定のプログラミングインタフェースの同時実行性設定 API を使用して、カーソルレベルで指定します。文が更新可能でカーソルに同時制御メカニズムが指定されていない場合は、文の仕様が使用されます。構文は次のとおりです。

FOR UPDATE BY LOCK

データベースサーバは、結果セットのフェッチされたローに対する意図的ローロックを取得します。これは、トランザクションが COMMIT または ROLLBACK されるまで保持される長時間のロックです。

FOR UPDATE BY { VALUES | TIMESTAMP }

データベースサーバは、キーセット駆動型カーソルを使用して、結果セットをスクロールしているときにローが変更または削除された場合にアプリケーションが通知されるようにします。

更新可能な文の制限

FOR UPDATE (`column-list`) を指定すると、後続の UPDATE WHERE CURRENT OF 文では指定された結果セットの属性のみ変更できるよう制限されます。

1.2.4.8 キャンセルされるカーソル操作

インタフェース機能で要求をキャンセルできます。

カーソル操作実行要求をキャンセルした場合は、カーソルの位置は確定されません。要求をキャンセルしたら、カーソルを絶対位置によって見つけるか、カーソルを閉じます。

1.2.5 カーソルタイプ

各種プログラミングインタフェースが、データベースカーソルの全側面をサポートしているわけではありません。用語が異なっている可能性もあります。プログラミングインタフェースごとに利用可能なカーソルとオプションの間のマッピングについては、次の資料で説明しています。

このセクションの内容:

カーソルの可用性 [25 ページ]
すべてのインタフェースがすべてカーソルタイプをサポートするわけではありません。

カーソルのプロパティ [26 ページ]
プログラミングインタフェースから明示的または暗黙的にカーソルタイプを要求します。インタフェースライブラリが異なれば、使用できるカーソルタイプは異なります。

ブックマークとカーソル [27 ページ]
ODBC にはブックマークがあります。これはカーソル内のローの識別に使う値です。

ブロックカーソル [27 ページ]
ODBC にはブロックカーソルと呼ばれるカーソルタイプがあります。ブロックカーソルを使うと、SQLFetchScroll または SQLExtendedFetch を使って単一のローではなく、ローのブロックをフェッチできます。

関連情報

カーソルの属性 [27 ページ]

1.2.5.1 カーソルの可用性

すべてのインタフェースがすべてカーソルタイプをサポートするわけではありません。

- ADO.NET は、forward-only、read-only のカーソルのみをサポートします。
- ADO/OLE DB と ODBC では、すべてのカーソルタイプがサポートされています。
- Embedded SQL ではすべてのカーソルタイプがサポートされています。
- JDBC の場合:
  - SQL Anywhere JDBC ドライバでは、insensitive、sensitive、forward-only asensitive カーソルの宣言が許可されています。

- jConnect では、SQL Anywhere JDBC ドライバと同じ方法で、insensitive、sensitive、forward-only asensitive カーソルの宣言がサポートされています。ただし、jConnect の基本的な実装では、asensitive カーソルのセマンティックのみがサポートされています。
- SAP Open Client でサポートされているのは asensitive カーソルだけです。また、ユニークではない更新可能なカーソルを使用すると、パフォーマンスが著しく低下します。

関連情報

ODBC アプリケーションの結果セット [178 ページ]
カーソルの要求 [42 ページ]

1.2.5.2 カーソルのプロパティ

プログラミングインタフェースから明示的または暗黙的にカーソルタイプを要求します。インタフェースライブラリが異なれば、使用できるカーソルタイプは異なります。

たとえば、JDBC と ODBC では使用できるカーソルタイプは異なります。

各カーソルタイプは、複数の特性によって定義されます。

ユニーク性

カーソルがユニークであることを宣言すると、クエリは、各ローをユニークに識別するために必要なすべてのカラムを返すように設定されます。これは、プライマリキー内にあるすべてのカラムを返すということをしばしば意味します。必要だが指定されないすべてのカラムは結果セットに追加されます。デフォルトでは、カーソルタイプは非ユニークです。

更新可能性

読み込み専用として宣言されたカーソルは、位置付け更新と位置付け削除のどちらの操作でも使用されません。デフォルトでは、更新可能のカーソルタイプに設定されています。

スクロール動作

結果セットを移動するときにカーソルが異なる動作をするように宣言できます。カーソルによっては、現在のローまたはその次のローしかフェッチできません。結果セットを後方に移動したり、前方に移動したりできるカーソルもあります。

感知性

データベースに加えた変更を、カーソルを使用して表示/非表示にすることができます。

これらの特性に応じて、パフォーマンスやデータベースサーバでのメモリ使用量にかなりの影響をもたらすことがあります。

さまざまな特性を持つカーソルを組み合わせて使用できます。特定のタイプのカーソルを要求すると、できるだけ適切な特性が使用されます。

一部の特性を使用できない場合もあります。たとえば、insensitive カーソルは読み込み専用です。更新可能な insensitive カーソルをアプリケーションが要求すると、代わりに、別のカーソルタイプ (value-sensitive カーソル) が使用されます。

1.2.5.3 ブックマークとカーソル

ODBC にはブックマークがあります。これはカーソル内のローの識別に使う値です。

value-sensitive カーソルと insensitive カーソルのブックマークがサポートされます。たとえば、ODBC カーソルタイプの SQL_CURSOR_STATIC と SQL_CURSOR_KEYSET_DRIVEN ではブックマークがサポートされますが、カーソルタイプ SQL_CURSOR_DYNAMIC と SQL_CURSOR_FORWARD_ONLY ではブックマークがサポートされません。

1.2.5.4 ブロックカーソル

ODBC にはブロックカーソルと呼ばれるカーソルタイプがあります。ブロックカーソルを使うと、SQLFetchScroll または SQLExtendedFetch を使って単一のローではなく、ローのブロックをフェッチできます。

ブロックカーソルは Embedded SQL ARRAY フェッチと同じ動作をします。

1.2.6 カーソルの属性

カーソルが開くと結果セットに関連付けられます。一度開いたカーソルは一定時間開いたままになります。カーソルが開いている間、カーソルに関連付けられた結果セットは変更される可能性があります。変更は、カーソル自体を使用して行われるか、独立性レベルの稼働条件に基づいて他のトランザクションで行われます。

カーソルには、基本となるデータを表示できるように変更できるものと、変更を反映しないものがあります。基本となるデータの変更に対する感知性によって、カーソルの動作 (カーソルの感知性) は変わります。

さまざまな感知性特性を備えたカーソルが提供されています。

メンバーシップ、順序、値の変更

基本となるデータに加えた変更は、カーソルの結果セットの次の部分に影響を及ぼします。

メンバーシップ

結果セットのローのセットです。プライマリキー値で指定されています。

順序

結果セットにあるローの順序です。

値

結果セットにあるローの値です。

たとえば、次のような従業員情報を記載した簡単なテーブルで考えてみます (EmployeeID はプライマリキーカラムです)。

EmployeeID	Surname
1	Whitney

EmployeeID	Surname
2	Cobb
3	Chin

以下のクエリのカーソルは、プライマリキーの順序でテーブルからすべての結果を返します。

```
SELECT EmployeeID, Surname
FROM Employees
ORDER BY EmployeeID;
```

結果セットのメンバーシップは、ローを追加するか削除すると変更されます。値を変更するには、テーブル内の名前をどれか変更します。ある従業員のプライマリキー値を変更すると順序が変更される場合があります。

表示できる変更、表示できない変更

カーソルを開いた後、独立性レベルの稼働条件に基づいて、カーソルの結果セットのメンバーシップ、順序、値を変更できます。使用するカーソルタイプに応じて、これらの変更を反映するために、アプリケーションが表示する結果セットが変更されることも変更されないこともあります。

基本となるデータに加えた変更は、カーソルを使って表示または非表示にできます。表示できる変更とは、カーソルの結果セットに反映されている変更のことです。基本となるデータに加えた変更が、カーソルが表示する結果セットに反映されない場合は、非表示です。

このセクションの内容:

カーソルの感知性 [29 ページ]
カーソルは、基本となるデータの変更に対する感知性に基づいて分類されます。いいかえると、カーソル感知性は、表示される変更によって定義されます。

カーソルの感知性の例: 削除されたロー [30 ページ]
この例では、簡単なクエリを使って、異なるカーソルが、削除中の結果セットのローに対してどのように応答するかを見ていきます。

カーソルの感知性の例: 更新されたロー [31 ページ]
この例では、簡単なクエリを使って、順序が変更されるように現在更新されている結果セット内のローに対して、カーソルがどのように応答するかを見ていきます。

Insensitive カーソル [33 ページ]
insensitive カーソルには、insensitive メンバーシップ、順序、値が指定されています。カーソルが開かれた後の変更は表示されません。

sensitive カーソル [34 ページ]
sensitive カーソルは、読み取り専用か更新可能なカーソルタイプで使用されます。

asensitive カーソル [35 ページ]
asensitive カーソルには、メンバーシップ、順序、値に対する明確に定義された感知性はありません。感知性の持つ柔軟性によって、asensitive カーソルのパフォーマンスは最適化されます。

value-sensitive カーソル [36 ページ]
value-sensitive カーソルは、メンバーシップに対しては感知せず、結果セットの順序と値に対しては感知します。

カーソルの感知性とパフォーマンス [38 ページ]

カーソルのパフォーマンスとその他のプロパティの間には、トレードオフ関係があります。特に、カーソルを更新できるようにした場合は、カーソルによるクエリの処理と配信で、パフォーマンスを制約する制限事項が課されます。また、カーソル感知性に稼働条件を設けると、カーソルのパフォーマンスが制約されることがあります。

カーソルの感知性と独立性レベル [41 ページ]
カーソルの感知性と独立性レベルはどちらも同時制御の問題を処理しますが、それぞれ方法や使用するトレードオフのセットが異なります。

カーソルの要求 [42 ページ]
クライアントアプリケーションからカーソルタイプを要求すると、1 つのカーソルが返されます。カーソルは、プログラミングインタフェースで指定したカーソルタイプではなく、基本となるデータでの変更に対する結果セットの感知性によって定義されます。

関連情報

カーソル [17 ページ]

1.2.6.1 カーソルの感知性

カーソルは、基本となるデータの変更に対する感知性に基づいて分類されます。いいかえると、カーソル感知性は、表示される変更によって定義されます。

insensitive カーソル

カーソルが開いているとき、結果セットは固定です。基本となるデータに加えられた変更はすべて非表示です。

sensitive カーソル

カーソルが開いた後に結果セットを変更できます。基本となるデータに加えられた変更内容はすべて表示されます。

asensitive カーソル

変更は、カーソルを使用して表示される結果セットのメンバーシップ、順序、または値に反映されます。

value-sensitive カーソル

基本となるデータの順序または値の変更は参照可能です。カーソルが開いているとき、結果セットのメンバーシップは固定です。

カーソルの稼働条件は異なるため、実行にさまざまな制約があり、そのため、パフォーマンスに影響します。

関連情報

Insensitive カーソル [33 ページ]
sensitive カーソル [34 ページ]
asensitive カーソル [35 ページ]
value-sensitive カーソル [36 ページ]
カーソルの感知性とパフォーマンス [38 ページ]

1.2.6.2 カーソルの感知性の例: 削除されたロー

この例では、簡単なクエリを使って、異なるカーソルが、削除中の結果セットのローに対してどのように応答するかを見ていきます。

次の一連のイベントを考えてみます。

1. アプリケーションが、次のようなサンプルデータベースに対するクエリについてカーソルを開きます。

```
SELECT EmployeeID, Surname
FROM Employees
ORDER BY EmployeeID;
```

EmployeeID	Surname
102	Whitney
105	Cobb
129	Chin
...	...

2. アプリケーションがカーソルを使って最初のローをフェッチします (102)。
3. アプリケーションがカーソルを使ってその次のローをフェッチします (105)。
4. 別のトランザクションが、employee 102 (Whitney) を削除して変更をコミットします。

この場合、カーソルアクションの結果は、カーソルの感知性によって異なります。

insensitive カーソル

DELETE は、カーソルを使用して表示される結果セットのメンバーシップにも値にも反映されません。

アクション	結果
前のローをフェッチします	ローのオリジナルコピーを返します (102)。
最初のローをフェッチします (絶対フェッチ)	ローのオリジナルコピーを返します (102)。
2 番目のローをフェッチします (絶対フェッチ)	未変更のローを返します (105)。

sensitive カーソル

結果セットのメンバーシップが変更されたため、ロー (105) は結果セットの最初のローになります。

アクション	結果
前のローをフェッチします	`Row Not Found` を返します。前のローが存在しません。
最初のローをフェッチします (絶対フェッチ)	ロー 105 を返します。
2 番目のローをフェッチします (絶対フェッチ)	ロー 129 を返します。

value-sensitive カーソル

結果セットのメンバーシップは固定であり、ロー 105 は、結果セットの 2 番目のローのままです。DELETE はカーソルの値に反映され、結果セットに有効なホールを作成します。

アクション	結果
前のローをフェッチします	`No current row of cursor` を返します。最初のローが以前存在したカーソルにホールがあります。
最初のローをフェッチします (絶対フェッチ)	`No current row of cursor` を返します。最初のローが以前存在したカーソルにホールがあります。
2 番目のローをフェッチします (絶対フェッチ)	ロー 105 を返します。

asensitive カーソル

変更に対して、結果セットのメンバーシップおよび値は確定されません。前のロー、最初のロー、または 2 番目のローのフェッチに対する応答は、特定のクエリ最適化方法によって異なります。また、その方法にワークテーブル構成が含まれているかどうか、フェッチ中のローがクライアントからプリフェッチされたものかどうかによっても異なります。

多くのアプリケーションで感知性の重要度は高くはなく、その場合、asensitive カーソルは利点をもたらします。特に、前方専用や読み取り専用のカーソルを使用している場合は、基本となる変更は表示されません。また、高い独立性レベルで実行している場合は、基本となる変更は禁止されます。

1.2.6.3 カーソルの感知性の例: 更新されたロー

この例では、簡単なクエリを使って、順序が変更されるように現在更新されている結果セット内のローに対して、カーソルがどのように応答するかを見ていきます。

次の一連のイベントを考えてみます。

1. アプリケーションが、次のようなサンプルデータベースに対するクエリについてカーソルを開きます。

```
SELECT EmployeeID, Surname
FROM Employees
ORDER BY EmployeeID;
```

EmployeeID	Surname
102	Whitney
105	Cobb
129	Chin
...	...

2. アプリケーションがカーソルを使って最初のローをフェッチします (102)。
3. アプリケーションがカーソルを使ってその次のローをフェッチします (105)。
4. 別のトランザクションが employee 102 (Whitney) の従業員 ID を 165 に更新して変更をコミットします。

この場合、カーソルアクションの結果は、カーソルの感知性によって異なります。

insensitive カーソル

UPDATE は、カーソルを使用して表示される結果セットのメンバーシップと値のどちらにも反映されません。

アクション	結果
前のローをフェッチします	ローのオリジナルコピーを返します (102)。
最初のローをフェッチします (絶対フェッチ)	ローのオリジナルコピーを返します (102)。
2 番目のローをフェッチします (絶対フェッチ)	未変更のローを返します (105)。

sensitive カーソル

結果セットのメンバーシップが変更されたため、ロー (105) は結果セットの最初のローになります。

アクション	結果
前のローをフェッチします	SQLCODE 100 を返します。結果セットのメンバーシップは変更されたため、105 が最初のローになります。カーソルが最初のローの前の位置に移動します。
最初のローをフェッチします (絶対フェッチ)	ロー 105 を返します。
2 番目のローをフェッチします (絶対フェッチ)	ロー 129 を返します。

また、sensitive カーソルでフェッチすると、ローが前回読み取られてから変更されている場合、SQLE_ROW_UPDATED_WARNING 警告が返されます。警告が出されるのは 1 回だけです。同じローを再びフェッチしても警告は発生しません。

同様に、前回フェッチした後で、カーソルを使ってローを更新したり削除した場合には、SQLE_ROW_UPDATED_SINCE_READ エラーが返されます。sensitive カーソルで更新や削除を行うには、修正されたローをアプリケーションでもう一度フェッチします。

カーソルによってカラムが参照されなくても、任意のカラムを更新すると警告やエラーの原因となります。たとえば、Surname を返すクエリにあるカーソルは、Salary カラムだけが修正されていても、更新をレポートします。

value-sensitive カーソル

結果セットのメンバーシップは固定であり、ロー 105 は、結果セットの 2 番目のローのままです。UPDATE はカーソルの値に反映され、結果セットに有効な "ホール" を作成します。

アクション	結果
前のローをフェッチします	SQLCODE 100 を返します。結果セットのメンバーシップは変更されたため、105 が最初のローになります。カーソルはホール上にあります。つまり、ロー 105 の前にあります。
最初のローをフェッチします (絶対フェッチ)	SQLCODE -197 を返します。結果セットのメンバーシップは変更されたため、105 が最初のローになります。カーソルはホール上にあります。つまり、ロー 105 の前にあります。
2 番目のローをフェッチします (絶対フェッチ)	ロー 105 を返します。

asensitive カーソル

変更に対して、結果セットのメンバーシップおよび値は確定されません。前のロー、最初のロー、または 2 番目のローのフェッチに対する応答は、特定のクエリ最適化方法によって異なります。また、その方法にワークテーブル構成が含まれているかどうか、フェッチ中のローがクライアントからプリフェッチされたものかどうかによっても異なります。

注記

更新警告とエラーの状態はバルクオペレーションモード (-b データベースサーバオプション) では発生しません。

1.2.6.4 Insensitive カーソル

insensitive カーソルには、insensitive メンバーシップ、順序、値が指定されています。カーソルが開かれた後の変更は表示されません。

insensitive カーソルは、読み込み専用のカーソルタイプだけで使用されます。

標準

insensitive カーソルは、ISO/ANSI 規格の insensitive カーソル定義と ODBC の静的カーソルに対応しています。

プログラミングインタフェース

インタフェース	カーソルタイプ	コメント
ODBC、ADO/OLE DB	静的	更新可能な静的カーソルが要求された場合は、代わりに value-sensitive カーソルが使用される
Embedded SQL	INSENSITIVE	
JDBC	INSENSITIVE	
Open Client	サポート対象外	

説明

insensitive カーソルは常に、クエリの選択基準に合ったローを、ORDER BY 句が指定した順序で返します。

カーソルが開かれている場合は、insensitive カーソルの結果セットがワークテーブルとして完全に実体化されます。その結果は次のようになります。

- 結果セットのサイズが大きい場合は、それを管理するためディスクスペースとメモリの要件が重要になる場合があります。
- 結果セットがワークテーブルとしてアセンブルされるより前にアプリケーションに返されるローはありません。このため、複雑なクエリでは、最初のローがアプリケーションに返される前に遅れが生じる場合があります。
- 後続のローはワークテーブルから直接フェッチできるため、処理が早くなります。クライアントライブラリは 1 回に複数のローをプリフェッチできるため、パフォーマンスはさらに向上します。

- insensitive カーソルは、ROLLBACK または ROLLBACK TO SAVEPOINT には影響を受けません。

1.2.6.5 sensitive カーソル

sensitive カーソルは、読み取り専用か更新可能なカーソルタイプで使用されます。

このカーソルには、sensitive なメンバーシップ、順序、値が指定されています。

標準

sensitive カーソルは、ISO/ANSI 規格の sensitive カーソル定義と ODBC の動的カーソルに対応しています。

プログラミングインタフェース

インタフェース	カーソルタイプ	コメント
ODBC、ADO/OLE DB	動的	
Embedded SQL	SENSITIVE	要求されているワークテーブルがなく、prefetch オプションが Off に設定されている場合は、DYNAMIC SCROLL カーソルの要求に応じて提供されることもあります。
JDBC	SENSITIVE	sensitive セマンティックは、SQL Anywhere JDBC ドライバで完全にサポートされる

説明

sensitive カーソルでのプリフェッチは無効です。カーソルを使用した変更や他のトランザクションからの変更など、変更はどれもカーソルを使用して表示できます。上位の独立性レベルでは、ロックを実行しなければならないという理由から、他のトランザクションで行われた変更のうち、一部が非表示になっている場合もあります。

カーソルのメンバーシップ、順序、すべてのカラム値に対して加えられた変更は、すべて表示されます。たとえば、sensitive カーソルにジョインが含まれており、基本となるテーブルの 1 つにある値がどれか 1 つでも修正されると、その基本のローで構成されたすべての結果ローには新しい値が表示されます。結果セットのメンバーシップと順序はフェッチのたびに変更できます。

sensitive カーソルは常に、クエリの選択基準に合ったローを、ORDER BY 句が指定した順序で返します。更新は、結果セットのメンバーシップ、順序、値に影響する場合があります。

sensitive カーソルを実装するときには、sensitive カーソルの稼働条件によって、次のような制限が加えられます。

- ローのプリフェッチはできません。プリフェッチされたローに加えた変更は、カーソルを介して表示されないからです。これは、パフォーマンスに影響を与えます。

- sensitive カーソルを実装する場合は、作成中のワークテーブルを使用しないでください。ワークテーブルとして保管されたローに加えた変更はカーソルを介して表示されないからです。
- ワークテーブルの制限事項では、オプティマイザによるジョインメソッドの選択を制限しません。これは、パフォーマンスに影響を及ぼす可能性があります。
- クエリによっては、カーソルを sensitive にするワークテーブルを含まないプランをオプティマイザが構成できません。通常、ワークテーブルは、中間結果をソートしたりグループ分けしたりするときに使用されます。インデックスからローにアクセスできる場合、ソートにワークテーブルは不要です。どのクエリがワークテーブルを使用するかを正確に述べることはできませんが、次のようなクエリでは必ずワークテーブルを使用します。
  - UNION クエリ。ただし、UNION ALL クエリでは必ずしもワークテーブルは使用されません。
  - ORDER BY 句を持つ文 (ORDER BY カラムにインデックスが存在しない場合)。
  - ハッシュジョインを使用して最適化されたクエリ全般。
  - DISTINCT 句または GROUP BY 句を必要とする多くのクエリ。

  この場合、アプリケーションにエラーが返されるか、カーソルタイプが sensitive に変更されて警告が返されます。

1.2.6.6 asensitive カーソル

asensitive カーソルには、メンバーシップ、順序、値に対する明確に定義された感知性はありません。感知性の持つ柔軟性によって、asensitive カーソルのパフォーマンスは最適化されます。

asensitive カーソルは、読み取り専用のカーソルタイプだけに使用されます。

標準

asensitive カーソルは、ISO/ANSI 規格で定めた asensitive カーソルの定義と、感知性について特別な指定のない ODBC カーソルに対応しています。

プログラミングインタフェース

インタフェース	カーソルタイプ
ODBC、ADO/OLE DB	感知性は未指定
Embedded SQL	DYNAMIC SCROLL

説明

クエリを最適化してアプリケーションにローを返すときに使用する方法に対して、asensitive カーソルの要求では制約がほとんどありません。このため、asensitive カーソルを使うと最適なパフォーマンスを得られます。特に、オプティマイザは中間結果をワークテーブルとして実体化するというような措置をとる必要はありません。また、クライアントはローをプリフェッチできます。

基本のローに加えた変更の表示については保証されません。表示される変更と、表示されない変更があります。メンバーシップと順序はフェッチのたびに変わります。特に、基本のローを更新しても、カーソルの結果には、更新されたカラムの一部しか反映されないことがあります。

asensitive カーソルでは、クエリの選択内容と順序に一致するローを返すことは保証されません。ローのメンバーシップはカーソルを開いたときには固定されていますが、その後加えられる基本の値への変更は結果に反映されます。

asensitive カーソルは常に、カーソルのメンバーシップが確立された時点で顧客の WHERE 句と ORDER BY 句に一致したローを返します。カーソルが開かれた後でカラム値が変わると、WHERE 句と ORDER BY 句に一致しなくなったローは返される場合があります。

1.2.6.7 value-sensitive カーソル

value-sensitive カーソルは、メンバーシップに対しては感知せず、結果セットの順序と値に対しては感知します。

value-sensitive カーソルは、読み取り専用か更新可能なカーソルタイプで使用されます。

標準

value-sensitive カーソルは、ISO/ANSI 規格の定義に対応していません。このカーソルは、ODBC キーセット駆動型カーソルに対応します。

プログラミングインタフェース

インタフェース	カーソルタイプ	コメント
ODBC、ADO/OLE DB	キーセット駆動型	
Embedded SQL	SCROLL	
JDBC	INSENSITIVE と CONCUR_UPDATABLE	SQL Anywhere JDBC ドライバでは、更新可能な INSENSITIVE カーソルの要求は value-sensitive カーソルで応答されます。
Open Client と jConnect	サポートされていません。	

説明

変更した基本のローで構成されているローをアプリケーションがフェッチすると、そのアプリケーションは更新された値を表示します。また、SQL_ROW_UPDATED ステータスがアプリケーションに発行されます。削除された基本のローで構成されているローをアプリケーションがフェッチした場合は、SQL_ROW_DELETED ステータスがアプリケーションに発行されます。

プライマリキー値に加えられた変更によって、結果セットからローが削除されます (削除として処理され、その後、挿入が続きます)。カーソルまたは外部から結果セットのローが削除されると、特別のケースが発生し、同じキー値を持つ新しいキーが挿入されます。この結果、新しいローと、それが表示されていた古いローが置き換えられます。

結果セットのローが、クエリの選択内容や順序指定に一致するという保証はありません。ローのメンバーシップは開かれた時に固定であるため、ローが後で変更されて WHERE 句または ORDER BY 句と一致しなくなっても、ローのメンバーシップと位置はいずれも変更されません。

どの値にも、カーソルを使用して行われた変更に対する感知性があります。カーソルを使用して行われた変更に対するメンバーシップの感知性は、ODBC オプションの SQL_STATIC_SENSITIVITY によって制御されます。このオプションが ON になっている場合は、カーソルを使った挿入によってそのカーソルにローが追加されます。それ以外の場合は、結果セットに挿入は含まれません。カーソルを使って削除すると、結果セットからローが削除され、SQL_ROW_DELETED ステータスを返すホールは回避されます。

value-sensitive カーソルはキーセットテーブルを使用します。カーソルが開かれている場合は、結果セットを構成する各ローの識別情報がワークテーブルに入力されます。結果セットをスクロールする場合、結果セットのメンバーシップを識別するためにキーセットテーブルが使用されますが、値は必要に応じて基本のテーブルから取得されます。

value-sensitive カーソルのメンバーシッププロパティは固定であるため、アプリケーションはカーソル内のローの位置を記憶でき、これらの位置が変更されないことが保証されます。

- ローが更新されたか、カーソルが開かれた後に更新された可能性がある場合、ローがフェッチされた時点で SQLE_ROW_UPDATED_WARNING が返されます。警告が生成されるのは 1 回だけです。同じローをもう一度フェッチしても、警告は生成されません。
  更新されたカラムがカーソルによって参照されていなくても、任意のカラムを更新すると警告の原因となります。たとえば、Surname と GivenName に対するカーソルは、Birthdate カラムだけが修正された場合でも更新の内容をレポートします。これらの更新警告とエラー条件は、バルクオペレーションモード (-b データベースサーバオプション) でローのロックが解除されている場合は発生しません。
- 前回フェッチした後に修正されたローで位置付け UPDATE 文または DELETE 文の実行を試みると、SQLE_ROW_UPDATED_SINCE_READ エラーが返されて、その文はキャンセルされます。アプリケーションでもう一度ローをフェッチすると UPDATE または DELETE が許可されます。
  更新されたカラムがカーソルによって参照されていなくても、ローの任意のカラムを更新するとエラーの原因となります。バルクオペレーションモードでは、エラーは発生しません。
- カーソルが開かれた後にカーソルまたは別のトランザクションからローを削除した場合は、カーソルにホールが作成されます。カーソルのメンバーシップは固定なので、ローの位置は予約されています。ただし、DELETE オペレーションは、変更されたローの値に反映されます。このホールでローをフェッチすると、現在のローがないことを示す -197 SQLCODE エラーが返され、カーソルはホールの上に配置されたままになります。sensitive カーソルを使用するとホールを回避できます。sensitive カーソルのメンバーシップは値とともに変化するからです。

value-sensitive カーソル用にローをプリフェッチすることはできません。この要件は、パフォーマンスに影響を及ぼすことがあります。

複数ローの挿入

複数のローを value-sensitive カーソルを介して挿入する場合、新しいローは結果セットの最後に表示されます。

関連情報

ローの修正に使用するカーソル [22 ページ]
カーソルの感知性の例: 削除されたロー [30 ページ]

1.2.6.8 カーソルの感知性とパフォーマンス

カーソルのパフォーマンスとその他のプロパティの間には、トレードオフ関係があります。特に、カーソルを更新できるようにした場合は、カーソルによるクエリの処理と配信で、パフォーマンスを制約する制限事項が課されます。また、カーソル感知性に稼働条件を設けると、カーソルのパフォーマンスが制約されることがあります。

カーソルの更新可能性と感知性がパフォーマンスに影響を与える仕組みを理解するには、カーソルによって表示される結果がどのようにしてデータベースからクライアントアプリケーションまで送信されるかを理解する必要があります。

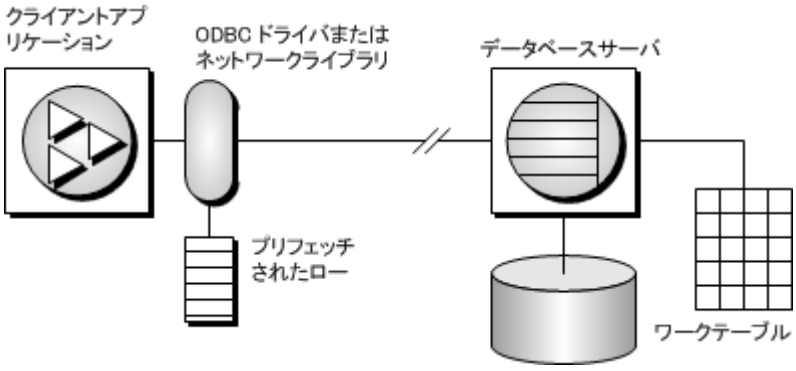
特に、パフォーマンス上の理由から、結果が中間の 2 つのロケーションに格納されることを理解する必要があります。

ワークテーブル

中間結果または最終結果はワークテーブルとして保管されます。value-sensitive カーソルは、プライマリキー値のワークテーブルを使用します。また、クエリの特性によって、オプティマイザが選択した実行プランでワークテーブルを使用するようになります。

プリフェッチ

クライアント側の通信はローを取り出してクライアント側のバッファに格納することで、データベースサーバに対するローごとの個別の要求を回避します。



感知性と更新可能性は中間のロケーションの使用を制限します。

このセクションの内容:

プリフェッチ [39 ページ]
プリフェッチはクライアント/サーバのラウンドトリップを削減してパフォーマンスを高め、1 つのローやローのブロックごとにサーバへ個別に要求しないで多数のローを使用可能にすることによってスループットを高めます。

更新内容の消失 [40 ページ]
更新可能なカーソルを使用する場合は、更新内容の消失から保護する必要があります。

1.2.6.8.1 プリフェッチ

プリフェッチはクライアント/サーバのラウンドトリップを削減してパフォーマンスを高め、1 つのローやローのブロックごとにサーバへ個別に要求しないで多数のローを使用可能にすることによってスループットを高めます。

プリフェッチは複数ローのフェッチとは異なります。プリフェッチはクライアントアプリケーションから明確な命令がなくても実行できます。プリフェッチはサーバからローを取り出し、クライアント側のバッファに格納しますが、クライアントアプリケーションがそれらのローを使用できるのは、アプリケーションが適切なローをフェッチしてからになります。

デフォルトでは、単一ローがアプリケーションによってフェッチされるたびに、クライアントライブラリによって複数のローがプリフェッチされます。クライアントライブラリは余分なローをバッファに格納します。

アプリケーションからのプリフェッチの制御

- prefetch オプションを使って、プリフェッチするかどうか制御できます。単一の接続では prefetch オプションを [常に]、[条件付き]、または [オフ] に設定できます。デフォルトでは [条件付き] に設定されています。
- Embedded SQL では、BLOCK 句を使用して、カーソルを開くときにカーソルベースで、または個別の FETCH オペレーションで、プリフェッチを制御できます。
  アプリケーションでは、サーバから 1 つのフェッチに含められるローの最大数を、BLOCK 句で指定できます。たとえば、一度に 5 つのローをフェッチして表示する場合は、BLOCK 5 を使用します。BLOCK 0 を指定すると、一度に 1 つのレコードがフェッチされ、FETCH RELATIVE 0 がサーバから同じローを常に再度フェッチするようになります。
  アプリケーションの接続パラメータを設定してプリフェッチをオフにすることもできますが、prefetch オプションを Off に設定するよりは、BLOCK 0 と指定する方が効果的です。
- value-sensitive カーソルタイプでは、プリフェッチはデフォルトで無効です。
- Open Client では、カーソルが宣言されてから開かれるまでの間に CS_CURSOR_ROWS で ct_cursor を使ってプリフェッチの動作を制御できます。

パフォーマンスが向上する可能性が高い場合に、プリフェッチするロー数が動的に増えます。これには、次の条件を満たすカーソルが含まれます。

- カーソルがサポートされるカーソルタイプのいずれかを使用しています。

  ODBC と OLE DB

  前方専用および読み込み専用 (デフォルト) のカーソル
  Embedded SQL

  DYNAMIC SCROLL (デフォルト)、NO SCROLL、および INSENSITIVE のカーソル
  ADO.NET

  すべてのカーソル
- カーソルが FETCH NEXT 操作のみ実行します (絶対フェッチ、相対フェッチ、後方フェッチは実行しません)。
- アプリケーションが、フェッチの間にホスト変数の型を変更したり、GET DATA 文を使用してチャンク単位でカラムのデータ取得を行ったりしません (GET DATA 文を *1* つ使用して、値を取得することはできます)。

1.2.6.8.2 更新内容の消失

更新可能なカーソルを使用する場合は、更新内容の消失から保護する必要があります。

更新内容の消失は、2 つ以上のトランザクションが同じローを更新して、どのトランザクションも別のトランザクションによって変更されたことに気付かず、2 番目の変更が最初の変更内容を上書きしてしまう場合に生じます。

このような問題について、次の例で説明します。

1. アプリケーションが、次のようなサンプルデータベースに対するクエリについてカーソルを開きます。

```
SELECT ID, Quantity
FROM Products;
```

ID	Quantity
300	28
301	54
302	75
…	…

2. アプリケーションが、カーソルを介して ID = 300 のローをフェッチします。
3. 次の文を使用して別のトランザクションがローを更新します。

```
UPDATE Products
SET Quantity = Quantity - 10
WHERE ID = 300;
```

4. アプリケーションが、カーソルを使用してローを `(Quantity - 5)` の値に更新します。
5. 最終的な正しいロー値は 13 になります。カーソルによってローがプリフェッチされていた場合は、そのローの新しい値は 23 になります。別のトランザクションが更新した内容は失われます。

データベースアプリケーションでは、前もって値の検証を行わずにローの内容を変更すると、どの独立性レベルにおいても更新内容が消失する可能性があります。より高い独立性レベル (2 と 3) では、ロック (読み込み、意図的、書き込みロック) を使用して、アプリケーションでいったん読み込まれたローの内容を別のトランザクションが変更できないように設定できます。一方、独立性レベル 0 と 1 では、更新内容が消失する可能性が高くなります。独立性レベルが 0 の場合、データがその後変更されることを防ぐための読み込みロックは取得されません。独立性レベルが 1 の場合は、現在のローだけがロックされます。スナップショットアイソレーションを使用している場合、更新内容の消失は起こりません。これは、古い値を変更しようとすると必ず更新の競合が発生するからです。さらに、独立性レベル 1 でプリフェッチを使用した場合も更新内容が消失する可能性があります。これは、アプリケーションが位置設定されている結果セットロー (クライアントのプリフェッチバッファ内) は、サーバがカーソル内で位置設定されている現在のローとは異なる場合があるためです。

独立性レベルが 1 の場合にカーソルで更新内容が消失されるのを防ぐため、データベースサーバは、アプリケーションで指定可能な 3 種類の同時制御メカニズムをサポートしています。

1. ローをフェッチするときに、カーソルの各ローに対する意図的ローロックの取得。意図的ロックを取得することで、他のトランザクションが同じローに対して意図的ロックや書き込みロックを取得できないようにし、同時更新の発生を防ぎます。ただし、意図的ロックでは読み込みローロックをブロックしないため、読み込み専用文の同時実行性には影響しません。
2. value-sensitive カーソルの使用。value-sensitive カーソルを使用して、基本となるローに対する変更や削除を追跡できるため、アプリケーションはそれに応じて応答できます。
3. FETCH FOR UPDATE の使用。特定のローに対する意図的ローロックを取得します。

これらのメカニズムの指定方法は、アプリケーションで使用されるインタフェースによって異なります。SELECT 文に関する最初の 2 つのメカニズムについては、次のようになります。

- ODBC では、アプリケーションで更新可能なカーソルを宣言するときに SQLSetStmtAttr 関数でカーソル同時実行性パラメータを指定する必要があるため、更新内容の消失は発生しません。このパラメータは、SQL_CONCUR_LOCK、SQL_CONCUR_VALUES、SQL_CONCUR_READ_ONLY、SQL_CONCUR_TIMESTAMP のいずれかです。SQL_CONCUR_LOCK を指定すると、データベースサーバはローに対する意図的ロックを取得します。SQL_CONCUR_VALUES と SQL_CONCUR_TIMESTAMP の場合は、value-sensitive カーソルが使用されます。SQL_CONCUR_READ_ONLY はデフォルトのパラメータで、読み込み専用カーソルに使用されます。
- JDBC では、文の同時実行性設定は ODBC の場合と似ています。JDBC ドライバでは、JDBC 同時実行性の値として RESULTSET_CONCUR_READ_ONLY と RESULTSET_CONCUR_UPDATABLE がサポートされています。最初の値は ODBC の同時実行性設定 SQL_CONCUR_READ_ONLY に対応し、読み込み専用文を指定します。2 番目の値は、ODBC の SQL_CONCUR_LOCK 設定に対応し、更新内容の消失を防ぐためにローの意図的ロックが使用されます。value-sensitive カーソルは、JDBC ドライバでは直接指定できません。
- jConnect では、更新可能なカーソルは API レベルではサポートされますが、(TDS を使用する) 基本の実装ではカーソルを使用した更新はサポートされていません。その代わりに、jConnect では個別の UPDATE 文をデータベースサーバに送信して、特定のローを更新します。更新内容が失われないようにするには、アプリケーションを独立性レベル 2 以上で実行してください。アプリケーションはカーソルから個別の UPDATE 文を発行できますが、UPDATE 文の WHERE 句で条件を指定してローを読み込んだ後でロー値が変更されていないことを UPDATE 文で必ず確認するようにしてください。
- Embedded SQL では、同時実行性の指定は SELECT 文自体またはカーソル宣言に構文を含めることで設定できます。SELECT 文で構文 SELECT...FOR UPDATE BY LOCK を使用すると、データベースサーバは結果セットに対する意図的ローロックを取得します。
  または、SELECT...FOR UPDATE BY [VALUES | TIMESTAMP] を使用すると、データベースサーバはカーソルタイプを value-sensitive に変更するため、そのカーソルを使用して特定のローを最後に読み込んだ後でローが変更された場合、アプリケーションには FETCH 文に対する警告 (SQLE_ROW_UPDATED_WARNING)、または UPDATE WHERE CURRENT OF 文に対するエラー (SQLE_ROW_UPDATED_SINCE_READ) のいずれかが返されます。ローが削除されている場合も、アプリケーションにはエラー (SQLE_NO_CURRENT_ROW) が返されます。

FETCH FOR UPDATE 機能は Embedded SQL と ODBC インタフェースでもサポートされていますが、詳細は使用している API によって異なります。

Embedded SQL の場合、アプリケーションは FETCH の代わりに FETCH FOR UPDATE を使用してローに対する意図的ロックを取得します。ODBC の場合、アプリケーションは API 呼び出しの SQLSetPos を使用し、オペレーション引数 SQL_POSITION または SQL_REFRESH とロックタイプ引数 SQL_LOCK_EXCLUSIVE を指定して、ローに対する意図的ロックを取得します。このロックは長期間のロックであり、トランザクションがコミットまたはロールバックされるまで保持されます。

1.2.6.9 カーソルの感知性と独立性レベル

カーソルの感知性と独立性レベルはどちらも同時制御の問題を処理しますが、それぞれ方法や使用するトレードオフのセットが異なります。

トランザクションの独立性レベルを選択することで (通常は接続レベルを選択)、データベースのローに対するロックの種類とタイミングを設定します。ロックすると、他のトランザクションはデータベースのローにアクセスしたり修正したりできなくなります。通常、保持するロックの数が多くなるほど、同時に実行されているトランザクションにおける同時実行レベルは低くなると予期されます。

ただし、ローをロックしても、同じトランザクションの別の部分では更新が行われます。したがって、更新可能な複数のカーソルを保持する 1 つのトランザクションでは、ローをロックしたとしても、更新内容の消失などの現象が起こらないとは保証されません。

スナップショットアイソレーションは、各トランザクションでデータベースの一貫したビューを表示することで、読み込みロックの必要性を排除します。完全に直列可能なトランザクション (独立性レベル 3) に依存せず、独立性レベル 3 を使用することで同時実行性を失うことなく、データベースの一貫したビューを問い合わせできるというのは大きな利点です。ただし、スナップショットアイソレーションの場合は、すでに実行中の同時実行のスナップショットトランザクションとまだ開始していないスナップショットトランザクションの両方の要件を満たすために修正されたローのコピーを保持する必要があるため、多大なコストがかかります。このようにコピーを保持する必要があるため、スナップショットアイソレーションの使用は更新を頻繁に行う負荷の高いトランザクションには適していない場合があります。

これに対して、カーソルの感知性は、カーソルの結果に対してどの変更を表示するか (または表示しないか) を決定します。カーソルの感知性はカーソルベースで指定するため、他のトランザクションと同じトランザクションの更新アクティビティの両方に影響しますが、影響度は指定されたカーソルタイプによって異なります。カーソルの感知性を設定しても、データベースのローをロックするタイミングを直接指定することにはなりません。ただし、カーソルの感知性と独立性レベルを組み合わせて使用することで、特定のアプリケーションで発生する可能性のある各種の同時実行シナリオを制御できます。

1.2.6.10 カーソルの要求

クライアントアプリケーションからカーソルタイプを要求すると、1 つのカーソルが返されます。カーソルは、プログラミングインタフェースで指定したカーソルタイプではなく、基本となるデータでの変更に対する結果セットの感知性によって定義されます。

要求されたカーソルタイプに基づいて、そのカーソルタイプに合う動作がカーソルに指定されます。

クライアントがカーソルタイプを要求すると、それに応じてカーソル感知性が設定されます。

このセクションの内容:

ADO.NET のカーソル要求 [43 ページ]
前方専用、読み取り専用のカーソルは、SACommand.ExecuteReader で使用することができます。SADataAdapter オブジェクトは、カーソルではなくクライアント側結果セットを使用します。

ADO/OLE DB と ODBC でのカーソル要求 [43 ページ]
次の表に、スクロール可能な各種の ODBC カーソルタイプに応じて設定されるカーソル感知性を示します。

JDBC でのカーソル要求 [44 ページ]
SQL Anywhere JDBC ドライバでは、insensitive、sensitive、forward-only asensitive の 3 つのカーソルタイプがサポートされています。

Embedded SQL でのカーソル要求 [44 ページ]
Embedded SQL アプリケーションからカーソルを要求するには、DECLARE 文にカーソルタイプを指定します。次の表に、各要求に応じて設定されるカーソル感知性を示します。

Open Client でのカーソル要求 [45 ページ]
Open Client の基本のプロトコル (TDS) でサポートされるカーソルは、forward-only、read-only、asensitive のみです。

1.2.6.10.1 ADO.NET のカーソル要求

前方専用、読み取り専用のカーソルは、SACommand.ExecuteReader で使用することができます。SADataAdapter オブジェクトは、カーソルではなくクライアント側結果セットを使用します。

1.2.6.10.2 ADO/OLE DB と ODBC でのカーソル要求

次の表に、スクロール可能な各種の ODBC カーソルタイプに応じて設定されるカーソル感知性を示します。

ODBC のスクロール可能なカーソルタイプ	カーソル感知性
STATIC	Insensitive
KEYSET-DRIVEN	Value-sensitive
DYNAMIC	Sensitive
MIXED	Value-sensitive

MIXED カーソルを取得するには、カーソルタイプを SQL_CURSOR_KEYSET_DRIVEN に指定し、SQL_ATTR_KEYSET_SIZE でキーセット駆動型カーソルのキーセット内のロー数を指定します。キーセットサイズが 0 (デフォルト) の場合、カーソルは完全にキーセット駆動型になります。キーセットサイズが 0 より大きい場合、カーソルは mixed (キーセット内はキーセット駆動型で、キーセット以外では動的) になります。デフォルトのキーセットサイズは 0 です。キーセットサイズが 0 より大きく、ローセットサイズ (SQL_ATTR_ROW_ARRAY_SIZE) より小さいと、エラーになります。

ODBC カーソル特性は、要求するカーソルタイプを判断するのに役立ちます。

例外

STATIC カーソルが更新可能なカーソルとして要求された場合は、代わりに value-sensitive カーソルが提供され、警告メッセージが発行されます。

DYNAMIC カーソルまたは MIXED カーソルが要求され、ワークテーブルを使用しなければクエリを実行できない場合、警告メッセージが発行され、代わりに asensitive カーソルが提供されます。

関連情報

カーソルの属性 [27 ページ]
ODBC カーソル特性 [180 ページ]

1.2.6.10.3 JDBC でのカーソル要求

SQL Anywhere JDBC ドライバでは、insensitive、sensitive、forward-only asensitive の 3 つのカーソルタイプがサポートされています。

SQL Anywhere JDBC ドライバでは、JDBC ResultSet オブジェクトに対してこの 3 種類のカーソルタイプがサポートされています。ただし、指定されたカーソルタイプに必要なセマンティックに基づいてデータベースサーバがアクセスプランを構築できない場合もあります。このような場合、データベースサーバはエラーを返すか、別のカーソルタイプに置き換えます。

関連情報

sensitive カーソル [34 ページ]

1.2.6.10.4 Embedded SQL でのカーソル要求

Embedded SQL アプリケーションからカーソルを要求するには、DECLARE 文にカーソルタイプを指定します。次の表に、各要求に応じて設定されるカーソル感知性を示します。

カーソルタイプ	カーソル感知性
NO SCROLL	Asensitive
DYNAMIC SCROLL	Asensitive
SCROLL	Value-sensitive
INSENSITIVE	Insensitive
SENSITIVE	Sensitive

例外

DYNAMIC SCROLL カーソルまたは NO SCROLL カーソルを UPDATABLE カーソルとして要求すると、sensitive または value-sensitive カーソルが返されます。どちらのカーソルが返されるかは保証されません。こうした不確定さは、asensitive の動作定義と矛盾しません。

INSENSITIVE カーソルが UPDATABLE (更新可能) として要求された場合は、value-sensitive カーソルが返されます。

DYNAMIC SCROLL カーソルが要求された場合、prefetch データベースオプションが Off に設定されている場合、クエリの実行プランにワークテーブルが使われない場合には、sensitive カーソルが返されます。ここでも、こうした不確定性は、asensitive の動作定義と矛盾しません。

1.2.6.10.5 Open Client でのカーソル要求

Open Client の基本のプロトコル (TDS) でサポートされるカーソルは、forward-only、read-only、asensitive のみです。

1.2.7 結果セット記述子

アプリケーションによっては、アプリケーション内で完全に指定できない SQL 文を構築するものがあります。表示するカラムをユーザが選択できるレポートアプリケーションなど、文がユーザからの応答に依存していて、ユーザの応答がないと、検索する情報をアプリケーションが正確に把握できない場合があります。

そのような場合、アプリケーションは、結果セットの性質と結果セットの内容の両方についての情報を検索する方法を必要とします。結果セットの性質についての情報を記述子と呼びます。記述子を用いて、返されるカラムの数や型を含むデータ構造体を識別します。アプリケーションが結果セットの性質を認識していると、内容の検索が簡単に行えます。

この結果セットメタデータ (データの性質と内容に関する情報) は記述子を使用して操作します。結果セットのメタデータを取得し、管理することを記述と呼びます。

通常はカーソルが結果セットを生成するので、記述子とカーソルは密接にリンクしています。ただし、記述子の使用をユーザに見えないように隠しているインタフェースもあります。通常、記述子を必要とする文は SELECT 文か、結果セットを返すストアドプロシージャのどちらかです。

カーソルベースの操作で記述子を使う手順は次のとおりです。

1. 記述子を割り付けます。インタフェースによっては明示的割り付けが認められているものもありますが、ここでは暗黙的に行います。
2. 文を準備します。
3. 文を記述します。文がストアドプロシージャの呼び出しかバッチであり、結果セットがプロシージャ定義において RESULT 句によって定義されていない場合、カーソルを開いてから記述を行います。
4. 文 (Embedded SQL) に対してカーソルを宣言して開くか、文を実行します。
5. 必要に応じて記述子を取得し、割り付けられた領域を修正します。多くの場合これは暗黙的に実行されます。
6. 文の結果をフェッチし、処理します。
7. 記述子の割り付けを解除します。
8. カーソルを閉じます。
9. 文を削除します。これはインタフェースによっては自動的に行われます。

実装の注意

- Embedded SQL では、SQLDA (SQL Descriptor Area) 構造体に記述子の情報があります。
- ODBC では、SQLAllocHandle を使って割り付けられた記述子ハンドルで記述子のフィールドへアクセスできます。SQLSetDescRec、SQLSetDescField、SQLGetDescRec、SQLGetDescField を使用して、これらのフィールドを操作できます。
  または、SQLDescribeCol と SQLColAttributes を使ってカラムの情報を取得することもできます。
- Open Client では、ct_dynamic を使って文を準備し、ct_describe を使って文の結果セットを記述します。ただし、ct_command を使って、SQL 文を最初に準備しないで送信し、ct_results を使って返されたローを 1 つずつ処理することもできます。これは Open Client アプリケーション開発を操作する場合に一般的な方法です。

- JDBC では、java.sql.ResultSetMetaData クラスが結果セットについての情報を提供します。
- たとえば、INSERT 文などでは、記述子を使用してデータベースサーバにデータを送信することもできます。ただし、これは結果セットの記述子とは種類が異なります。

関連情報

SQLDA (SQL descriptor area) [289 ページ]

1.2.8 アプリケーション内のトランザクション

トランザクションはアトミックな SQL 文をまとめたものです。トランザクション内の文はすべて実行されるか、どれも実行されないかのどちらかです。

このセクションの内容:

オートコミットまたは手動コミットモード [46 ページ]
データベースプログラミングインタフェースは、手動コミットモードまたはオートコミットモードで操作できます。

独立性レベルの設定 [50 ページ]
isolation_level データベースオプションを使って、現在の接続の独立性レベルを設定できます。

カーソルとトランザクション [51 ページ]
一般的に、COMMIT が実行されると、カーソルは閉じられます。

1.2.8.1 オートコミットまたは手動コミットモード

データベースプログラミングインタフェースは、手動コミットモードまたはオートコミットモードで操作できます。

手動コミットモード

オペレーションがコミットされるのは、アプリケーションが明示的なコミットオペレーションを実行した場合、または ALTER TABLE 文やその他のデータ定義文を実行する場合などのように、データベースサーバが自動コミットを実行した場合だけです。手動コミットモードを連鎖モードとも呼びます。

ネストされたトランザクションやセーブポイントなどのトランザクションをアプリケーションで使用するには、手動コミットモードで操作します。

オートコミットモード

文はそれぞれ、個別のトランザクションとして処理されます。これは、各 SQL 文の最後に COMMIT 文を付加して実行するのと同じ効果があります。オートコミットモードを非連鎖モードとも呼びます。

オートコミットモードは、使用中のアプリケーションのパフォーマンスや動作に影響することがあります。使用するアプリケーションでトランザクションの整合性が必要な場合は、オートコミットを使用しないでください。

このセクションの内容:

オートコミットの動作を制御する方法 [47 ページ]
アプリケーションのコミット動作を制御する方法は、使用しているプログラミングインタフェースによって異なります。オートコミットの実装は、インタフェースに応じて、クライアント側またはサーバ側で行うことができます。

オートコミット実装の詳細 [50 ページ]
オートコミットモードでは、使用するインタフェースやプロバイダ、またはオートコミット動作の制御方法に応じて、やや動作が異なります。

1.2.8.1.1 オートコミットの動作を制御する方法

アプリケーションのコミット動作を制御する方法は、使用しているプログラミングインタフェースによって異なります。オートコミットの実装は、インタフェースに応じて、クライアント側またはサーバ側で行うことができます。

オートコミットモードの制御 (ADO.NET)

デフォルトでは、ADO.NET プロバイダはオートコミットモードで動作します。明示的トランザクションを使用するには、SAConnection.BeginTransaction メソッドを使用します。

オートコミットモードの制御 (Embedded SQL)

デフォルトでは、Embedded SQL アプリケーションは手動コミットモードで動作します。オートコミットを一時的に有効化するには、次の文を実行して auto_commit データベースオプション (サーバ側オプション) を On に設定します。

```
SET TEMPORARY OPTION auto_commit='On'
```

オートコミットモードの制御 (JDBC)

デフォルトでは、JDBC はオートコミットモードで動作します。オートコミットモードを OFF にするには、次に示すように、接続オブジェクトの setAutoCommit メソッドを使用します。

```
conn.setAutoCommit( false );
```

オートコミットモードの制御 (ODBC)

デフォルトでは、ODBC はオートコミットモードで動作します。オートコミットを OFF にする方法は、ODBC を直接使用しているか、アプリケーション開発ツールを使用しているかによって異なります。ODBC インタフェースに直接プログラミングしている場合には、SQL_ATTR_AUTOCOMMIT 接続属性を設定してください。次の例では、オートコミットを無効化します。

```
SQLSetConnectAttr( hdbc, SQL_ATTR_AUTOCOMMIT, (SQLPOINTER)SQL_AUTOCOMMIT_OFF, 0 );
```

オートコミットモードの制御 (OLE DB)

デフォルトでは、OLE DB プロバイダはオートコミットモードで動作します。明示的トランザクションを使用するには、ITransactionLocal::StartTransaction、ITransaction::Commit、ITransaction::Abort メソッドを使用します。

オートコミットモードの制御 (Open Client)

デフォルトでは、Open Client 経由で行われた接続はオートコミットモードで動作します。この動作を変更するには、次の文を使用して、作業中のアプリケーションで chained データベースオプション (サーバ側オプション) を On に設定します。

```
SET OPTION chained='On'
```

オートコミットモードの制御 (Perl)

デフォルトでは、Perl はオートコミットモードで動作します。オートコミットを無効化するには、AutoCommit オプションを設定します。

```
my $dbh = DBI->connect( "DBI:SQLAnywhere:$connstr", '', '', {AutoCommit => 0} );
```

オートコミットモードの制御 (PHP)

デフォルトでは、PHP はオートコミットモードで動作します。オートコミットを無効化するには、sasql_set_option 関数を使用します。

```
$result = sasql_set_option( $conn, "auto_commit", "Off" );
```

オートコミットモードの制御 (Python)

デフォルトでは、Python は手動コミットモードで動作します。オートコミットを有効化するには、次の文を実行して auto_commit データベースオプション (サーバ側オプション) を On に設定します。

```
cursor.execute("SET TEMPORARY OPTION auto_commit='On'")
```

オートコミットモードの制御 (Ruby)

デフォルトでは、Ruby は手動コミットモードで動作します。オートコミットを有効化するには、次の文を実行して auto_commit データベースオプション (サーバ側オプション) を On に設定します。

```
rc = api.sqlany_execute_immediate( conn, "SET TEMPORARY OPTION auto_commit='On'" )
```

オートコミットモードの制御 (サーバの場合)

デフォルトでは、データベースサーバは手動コミットモードで動作します。オートコミットを一時的に有効化するには、次の文を実行して auto_commit データベースオプション (サーバ側オプション) を On に設定します。

```
SET TEMPORARY OPTION auto_commit='On'
```

注記

ADO.NET、JDBC、ODBC、OLE DB などの API を使用している場合、auto_commit サーバオプションを直接設定しないでください。オートコミットを有効化または無効化するには、API 固有のメカニズムを使用してください。たとえば、ODBC では SQLSetConnectAttr を使用して SQL_ATTR_AUTOCOMMIT 接続属性を設定します。API を使用する場合、ドライバでオートコミットの現在の設定を追跡できます。

注記

Interactive SQL で SET TEMPORARY OPTION 文を使用して auto_commit オプションを設定すると、同じ名前の Interactive SQL オプションを設定するため、auto_commit オプションを設定できません。EXECUTE IMMEDIATE 文にこの文を埋め込むことはできますが、予期しない動作を招く可能性があるため、おすすめできません。デフォルトでは、Interactive SQL は手動コミットメントモードで動作します (auto_commit = 'Off')。

関連情報

オートコミット実装の詳細 [50 ページ]
トランザクション処理 [79 ページ]

1.2.8.1.2 オートコミット実装の詳細

オートコミットモードでは、使用するインタフェースやプロバイダ、またはオートコミット動作の制御方法に応じて、やや動作が異なります。

一部のアプリケーションプログラミングインタフェースは、デフォルトで手動コミットモードで動作します。その他のアプリケーションプログラミングインタフェースは、デフォルトで自動コミット (オートコミット) モードで動作します。デフォルトのモードを決定するための各 API については、マニュアルを参照してください。

オートコミットの動作は、アプリケーションプログラミングインタフェースに応じて制御されます。

API メソッドの実行

オートコミットを有効化または無効化するため、一部のアプリケーションプログラミングインタフェースではネイティブの呼び出し可能なメソッドを提供します。例として、ADO.NET、ADO/OLE DB、JDBC、ODBC、Perl、PHP などです。

たとえば、ODBC アプリケーションで、次のように API 呼び出しを使用してオートコミットを有効化します。

```
SQLSetConnectAttr( hdbc, SQL_ATTR_AUTOCOMMIT, (SQLPOINTER)SQL_AUTOCOMMIT_ON, 0 );
```

文のオプションの実行

オートコミットを有効化または無効化するには、一部のアプリケーションプログラミングインタフェースでは SQL SET OPTION 文の実行が必要です。例として、ESQL、Python、Ruby などです。次の文では、この接続に対してのみ、データベースサーバでオートコミットを一時的に有効化します。

```
SET TEMPORARY OPTION auto_commit='On'
```

SAP Open Client インタフェースを使用している場合、CHAINED オプションを設定してオートコミットの動作を操作します。CHAINED オプションは、TDS アプリケーションの互換性のために提供されています。jConnect JDBC ドライバを使用している場合、CHAINED オプションを設定するのではなく、JDBC setAutoCommit メソッドを呼び出す必要があります。

注記

アプリケーションプログラミングインタフェースが、オートコミットの有効化または無効化専用の呼び出し可能なネイティブメソッドを提供している場合、そのインタフェースを使用する必要があります。

1.2.8.2 独立性レベルの設定

isolation_level データベースオプションを使って、現在の接続の独立性レベルを設定できます。

ODBC など、インタフェースによっては、接続時に接続の独立性レベルを設定できます。このレベルは isolation_level データベースオプションを使って、後でリセットできます。

INSERT、UPDATE、DELETE、SELECT、UNION、EXCEPT、INTERSECT の各文に OPTION 句を含めることによって、各文で指定したオプション設定を isolation_level database データベースオプションに対するテンポラリ設定やパブリック設定よりも優先させることができます。

1.2.8.3 カーソルとトランザクション

一般的に、COMMIT が実行されると、カーソルは閉じられます。

この動作には、2 つの例外があります。

- close_on_endtrans データベースオプションが Off に設定されています。
- カーソルが WITH HOLD で開かれています。Open Client と JDBC ではデフォルトです。

この 2 つのどちらかに当てはまる場合、カーソルは COMMIT 時に開いたままになります。

ROLLBACK とカーソル

トランザクションがロールバックされた場合、WITH HOLD でオープンされたカーソルを除いて、カーソルは閉じられます。ただし、ロールバック後のカーソルの内容は、信頼性が高くありません。

ISO SQL3 標準の草案には、ロールバックについて、すべてのカーソルは (WITH HOLD でオープンされたカーソルも) 閉じられるべきだと述べられています。この動作は ansi_close_cursors_on_rollback オプションを On に設定して得られます。

セーブポイント

トランザクションがセーブポイントへロールバックされ、ansi_close_cursors_on_rollback オプションが On に設定されていると、SAVEPOINT 後に開かれたすべてのカーソルは (WITH HOLD でオープンされたカーソルも) 閉じられます。

カーソルと独立性レベル

トランザクションが SET OPTION 文を使って isolation_level オプションを変更する間、接続の独立性レベルを変更できます。ただし、この変更は開いているカーソルには反映されません。

WITH HOLD 句が snapshot、statement-snapshot、および readonly-statement-snapshot の各独立性レベルで使用されている場合、スナップショットの開始時にコミットされたすべてのローのスナップショットが表示されます。カーソルが開かれたトランザクションの開始以降の、現在の接続で完了された変更もすべて表示されます。

1.3 .NET アプリケーションプログラミング

SQL Anywhere .NET Data Provider の API を含め、SQL Anywhere を .NET で使用します。

このセクションの内容:

SQL Anywhere .NET データプロバイダ [52 ページ]

Microsoft ADO.NET は、ODBC、OLE DB、ADO について最新のデータアクセス API です。ADO.NET は、Microsoft .NET Framework に適したデータアクセスコンポーネントであり、リレーショナルデータベースシステムにアクセスできます。

.NET データプロバイダチュートリアル [100 ページ]
Simple サンプルプロジェクトと Table Viewer サンプルプロジェクトを使用して、.NET プロバイダを使用した .NET アプリケーションプログラミングについて説明します。付属のチュートリアルでは、Microsoft Visual Studio を使用して Simple Viewer .NET データベースアプリケーションを構築する手順をひととおり説明します。

SQL Anywhere ASP.NET プロバイダ [118 ページ]
SQL Anywhere ASP.NET プロバイダは、Microsoft SQL Server の標準の ASP.NET プロバイダに代わり、SQL Anywhere を使用して Web サイトを運用できるようにします。

SQL Anywhere .NET API リファレンス [126 ページ]
SQL Anywhere .NET Data Provider の API を含め、SQL Anywhere を .NET で使用します。

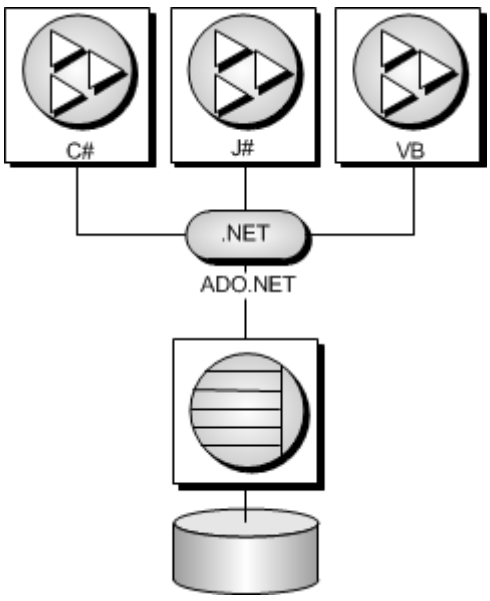
1.3.1 SQL Anywhere .NET データプロバイダ

Microsoft ADO.NET は、ODBC、OLE DB、ADO について最新のデータアクセス API です。ADO.NET は、Microsoft .NET Framework に適したデータアクセスコンポーネントであり、リレーショナルデータベースシステムにアクセスできます。

SQL Anywhere NET データプロバイダは、Sap.Data.SQLAnywhere ネームスペースを実装しており、.NET でサポートされている任意の言語 (Microsoft C# や Microsoft Visual Basic .NET など) でプログラムを作成したり、SQL Anywhere データベースからデータにアクセスしたりできます。

Microsoft ADO.NET アプリケーション

オブジェクト指向型言語を使用してインターネットおよびイントラネットのアプリケーションを開発し、SQL Anywhere .NET データプロバイダを使用してアプリケーションをデータベースサーバに接続できます。



このセクションの内容:

SQL Anywhere.NET データプロバイダ機能 [53 ページ]
Microsoft .NET Framework は、3 つの異なるネームスペースでサポートされています。

.NET サンプルプロジェクト [54 ページ]
SQL Anywhere .NET データプロバイダには、複数のサンプルプロジェクトが用意されています。

Microsoft Visual Studio プロジェクトでの SQL Anywhere .NET データプロバイダの使用 [55 ページ]
Microsoft Visual Studio で SQL Anywhere .NET データプロバイダを使用して .NET アプリケーションを開発するには、SQL Anywhere .NET データプロバイダへの参照と、ソースコードで SQL Anywhere .NET データプロバイダクラスを参照する行の両方を追加します。

Microsoft .NET データベースの接続 [56 ページ]
データベースに接続するには、SAConnection オブジェクトを作成する必要があります。接続文字列はオブジェクトを作成する場合に指定するか、ConnectionString プロパティを設定して後で確立できます。

データへのアクセスとデータの操作 [62 ページ]
SQL Anywhere .NET データプロバイダでは、SACommand クラスまたは SADataAdapter クラスを使用してデータにアクセスできます。

ストアドプロシージャ [78 ページ]
SQL Anywhere .NET データプロバイダでは SQL ストアドプロシージャを使用できます。

トランザクション処理 [79 ページ]
SATransaction オブジェクトを使用して複数の文をグループ化できます。各トランザクションは、データベースへの変更内容を確定する COMMIT メソッドの呼び出し、またはトランザクションのすべてのオペレーションをキャンセルする ROLLBACK メソッドの呼び出しで終了します。

Microsoft .NET エラー処理 [80 ページ]
アプリケーションは発生するあらゆるエラーを処理するように設計する必要があります。例外がスローされるときに SAException オブジェクトが作成されます。例外に関する情報は、SAException オブジェクトに格納されます。

Entity Framework のサポート [81 ページ]
SQL Anywhere .NET データプロバイダでは Entity Framework 5.0 および 6.0 がサポートされています。Entity Framework は Microsoft から別個のパッケージとして入手できます。

.NET でのトレースのサポート [95 ページ]
.NET データプロバイダでは、.NET のトレーシング機能を使用したトレースをサポートしています。

SQL Anywhere .NET データプロバイダのアンマネージコード [99 ページ]
.NET データプロバイダが .NET アプリケーションによって最初にロードされると (通常、SAConnection を使用したデータベース接続の作成時)、プロバイダのアンマネージコードを含む DLL がアンパックされます。

1.3.1.1 SQL Anywhere.NET データプロバイダ機能

Microsoft .NET Framework は、3 つの異なるネームスペースでサポートされています。

Sap.Data.SQLAnywhere

ADO.NET オブジェクトモデルは、万能型のデータアクセスオブジェクトモデルです。ADO.NET コンポーネントは、データ操作によるデータアクセスを要素として組み込むよう設計されました。そのため、ADO.NET には DataSet と .NET

Framework データプロバイダという 2 つの中心的なコンポーネントがあります。.NET Framework データプロバイダは、Connection、Command、DataReader、DataAdapter オブジェクトからなるコンポーネントのセットです。OLE DB または ODBC のオーバヘッドを加えずにデータベースサーバと直接通信する .NET Entity Framework データプロバイダが含まれています。.NET データプロバイダは、.NET ネームスペースでは Sap.Data.SQLAnywhere として表現されます。

SQL Anywhere .NET データプロバイダのネームスペースについては、このマニュアルで説明します。

System.Data.Oledb

このネームスペースは、OLE DB データソースをサポートしています。これは、Microsoft .NET Framework 固有の部分です。System.Data.Oledb を OLE DB プロバイダの SAOLEDB とともに使用して、データベースにアクセスできます。

System.Data.Odbc

このネームスペースは、ODBC データソースをサポートしています。これは、Microsoft .NET Framework 固有の部分です。System.Data.Odbc を ODBC ドライバとともに使用して、データベースにアクセスできます。

.NET データプロバイダを使用する場合、次のような主な利点がいくつかあります。

- .NET 環境では、.NET データプロバイダは、データベースに対するネイティブアクセスを提供します。サポートされている他のプロバイダとは異なり、このデータプロバイダはデータベースサーバと直接通信を行うため、ブリッジテクノロジを必要としません。
- そのため、.NET データプロバイダは、OLE DB や ODBC のデータプロバイダより処理速度が高速です。これは、SQL Anywhere データベースへのアクセスするためのデータプロバイダです。

1.3.1.2 .NET サンプルプロジェクト

SQL Anywhere .NET データプロバイダには、複数のサンプルプロジェクトが用意されています。

DeployUtility

これは ClickOnce の配備における SQL Anywhere .NET データプロバイダのアンマネージコード部分の配備を支援するためのコード例です。

LinqSample

SQL Anywhere .NET データプロバイダと C# を使用して、統合言語クエリ、セット、変換操作を示す、Windows 用の .NET Framework サンプルプロジェクト。

SimpleWin32

接続をクリックしたときに Employees テーブルの名前が設定された簡単なリストボックスを示す、Microsoft Windows 用の Microsoft .NET Framework サンプルプロジェクト。

SimpleXML

Microsoft ADO.NET を使用してデータベースから XML データを取得する方法を示す、Microsoft Windows 用の Microsoft .NET Framework サンプルプロジェクト。Microsoft C#、Visual Basic、Microsoft Visual C++ のサンプルが用意されています。

SimpleViewer

Microsoft Windows 用の Microsoft .NET Framework サンプルプロジェクト。

TableViewer

SQL 文を入力および実行可能な、Microsoft Windows 用の Microsoft .NET Framework サンプルプロジェクト。

関連情報

ClickOnce と .NET データプロバイダのアンマネージコード DLL [809 ページ]
チュートリアル:SimpleWin32 の Simple コードサンプルの使用 [101 ページ]
チュートリアル:Microsoft Visual Studio を使用したシンプルな .NET データベースアプリケーションの開発 [109 ページ]
チュートリアル:Table Viewer コードサンプルの使用 [105 ページ]

1.3.1.3 Microsoft Visual Studio プロジェクトでの SQL Anywhere .NET データプロバイダの使用

Microsoft Visual Studio で SQL Anywhere .NET データプロバイダを使用して .NET アプリケーションを開発するには、SQL Anywhere .NET データプロバイダへの参照と、ソースコードで SQL Anywhere .NET データプロバイダクラスを参照する行の両方を追加します。

手順

1. Microsoft Visual Studio を起動し、プロジェクトを作成するか、開きます。
2. ソリューションエクスプローラウィンドウで、参照設定を右クリックし、参照の追加をクリックします。

   参照によって、インクルードする必要があるプロバイダが示され、SQL Anywhere .NET データプロバイダのコードが検索されます。
3. *.NET* タブをクリックし (またはアセンブリ/拡張機能を開き)、リストをスクロールして、次のいずれかを見つけます。
   - Sap.Data.SQLAnywhere for .NET 3.5
   - Sap.Data.SQLAnywhere for .NET 4
   - Sap.Data.SQLAnywhere for .NET 4.5
   - Sap.Data.SQLAnywhere for Entity Framework 6

   プロバイダの選択は、プロジェクトの Target フレームワークで制御できます。
4. 目的のプロバイダをクリックし、選択したプロバイダのチェックボックスが存在する場合はオンにされていることを確認し、*OK* をクリックします。

   プロジェクトのソリューションエクスプローラウィンドウの参照設定フォルダにプロバイダが追加されます。
5. ソースコードにディレクティブを追加し、SQL Anywhere .NET データプロバイダのネームスペースと定義済みの型を簡単に使用できるようにします。

   次の行をプロジェクトに追加します。
   - C# を使用している場合、ソースコードの先頭にある `using` ディレクティブのリストに次の行を追加します。

     ```
     using Sap.Data.SQLAnywhere;
     ```
   - Visual Basic を使用している場合、ソースコードの先頭に次の行を追加します。

     ```
     Imports Sap.Data.SQLAnywhere
     ```

結果

SQL Anywhere .NET データプロバイダが、SQL Anywhere .NET アプリケーションで使用できるように設定されます。

例

次の C# の例は、*using* ディレクティブが指定された場合に接続オブジェクトを作成する方法を示します。

```
SAConnection conn = new SAConnection();
```

次の C# の例は、*using* ディレクティブが指定されていない場合に接続オブジェクトを作成する方法を示します。

```
Sap.Data.SQLAnywhere.SAConnection conn =
    new Sap.Data.SQLAnywhere.SAConnection();
```

次の Microsoft Visual Basic の例は、*Imports* ディレクティブが指定された場合に接続オブジェクトを作成する方法を示します。

```
Dim conn As New SAConnection()
```

次の Microsoft Visual Basic の例は、*Imports* ディレクティブが指定されていない場合に接続オブジェクトを作成する方法を示します。

```
Dim conn As New Sap.Data.SQLAnywhere.SAConnection()
```

1.3.1.4 Microsoft .NET データベースの接続

データベースに接続するには、SAConnection オブジェクトを作成する必要があります。接続文字列はオブジェクトを作成する場合に指定するか、ConnectionString プロパティを設定して後で確立できます。

設計が良くできているアプリケーションは、データベースに接続しようとするときに発生するエラーを処理します。

データベースとの接続は、接続が開かれると作成され、接続が閉じられると解放されます。

Microsoft C# の SAConnection の例

次の Microsoft C# コードは、サンプルデータベースとの接続を開いて閉じるボタンクリックハンドラを作成します。例外ハンドラが含まれています。

```
private void button1_Click(object sender, EventArgs e)
{
    SAConnection conn = new SAConnection("Data Source=SQL Anywhere 17
Demo;Password="+pwd);
    try
    {
        conn.Open();
        conn.Close();
    }
    catch (SAException ex)
```

```
    {
        MessageBox.Show(ex.Errors[0].Source + " : " +
             ex.Errors[0].Message + " (" +
             ex.Errors[0].NativeError.ToString() + ")",
             "Failed to connect");
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message, "Failed to connect");
    }
}
```

Microsoft Visual Basic の SAConnection の例

次の Microsoft Visual Basic コードは、サンプルデータベースとの接続を開いて閉じるボタンクリックハンドラを作成します。例外ハンドラが含まれています。

```
Private Sub Button1_Click(ByVal sender As System.Object, _
     ByVal e As System.EventArgs) Handles Button1.Click
    Dim conn As New SAConnection("Data Source=SQL Anywhere 17 Demo;Password="+pwd)
    Try
        conn.Open()
        conn.Close()
    Catch ex As SAException
        MessageBox.Show(ex.Errors(0).Source & " : " & _
                 ex.Errors(0).Message & " (" & _
                 ex.Errors(0).NativeError.ToString() & ")", _
                 "Failed to connect")
    Catch ex as Exception
        MessageBox.Show(ex.Message, "Failed to connect")
    End Try
End Sub
```

このセクションの内容:

.NET 接続パラメータ [57 ページ]
接続パラメータを接続文字列で使用し、データベースサーバに接続し、認証を取得します。

.NET 接続プーリング [61 ページ]
SQL Anywhere .NET データプロバイダは、ネイティブ .NET 接続プーリングをサポートしています。接続プーリングを使用すると、アプリケーションは、データベースへの新しい接続を繰り返し作成しなくても、接続ハンドルをプールに保存して再使用できるようにして、既存の接続を再使用できます。

接続状態 [62 ページ]
アプリケーションからデータベースへの接続が確立したら、接続がまだ開かれているかについて接続状態を確認してから、要求をデータベースサーバに送信できます。

1.3.1.4.1 .NET 接続パラメータ

接続パラメータを接続文字列で使用し、データベースサーバに接続し、認証を取得します。

接続文字列を .NET アプリケーションで指定するのは、接続オブジェクトが作成された場合です。または、接続オブジェクトの ConnectionString プロパティを設定して接続文字列を指定することもできます。

- ```
  SAConnection conn = new SAConnection("connection-string");
  ```
- ```
  SAConnection conn = new SAConnection();
  conn.ConnectionString = "connection-string";
  ```

キーワード=値のペアをセミコロンで区切り、接続パラメータを指定します。次に例を示します。

```
SAConnection conn = new SAConnection(
        "Host=sqla-host:2638;Server=sqla-server;UserID=JSmith;Password=secret");
```

接続パラメータ名では大文字と小文字が区別されません。たとえば、UserID と userid は同じです。接続パラメータにスペースが含まれている場合は、それも保持する必要があります。

接続パラメータ値では、大文字と小文字を区別するものもあります。たとえば、パスワードでは通常、大文字と小文字が区別されます。

接続パラメータ

接続パラメータ	説明
Connection lifetime	プールされる接続の最大寿命 (秒) を指定します。接続が最大寿命よりも長く開かれている場合、閉じられると、接続はプールされません。デフォルトは 0 で、上限はありません。 `Connection lifetime=seconds`
Connection timeout	データベースサーバへの接続試行を終了し、エラーを生成するまで、接続を待機する時間 (秒) を指定します。デフォルトは 15 です。代わりに、Connect Timeout を使用することもできます。 `Connection timeout=seconds`

DatabaseName	データベースの名前を識別します。代わりに、DBN を使用することもできます。
	`DatabaseName=db-name`
	このパラメータは、接続オブジェクトの読み取り専用プロパティ Database を設定します。プロパティは、次の方法でクエリできます。
	`String database = conn.Database;`

Data Source	データソース名を識別します。代わりに、DataSourceName と DSN を使用することもできます。 `Data Source=datasource-name`

接続パラメータ	説明
Enlist	Microsoft 分散トランザクションコーディネーターでトランザクションをエンリストするかどうかを指定します。デフォルトは **true** です。 `Enlist=boolean-value` **false** に設定した場合、Microsoft 分散トランザクションコーディネーターでトランザクションをエンリストされません。 分散トランザクションを操作するには、各コンピュータ上で分散トランザクションコーディネーター (DTC) サービスを実行する必要があります。Microsoft Windows の *[サービス]* ウィンドウから DTC を開始または停止できます。DTC サービスは、MSDTC という名前で表示されます。
FileDataSourceName	ファイルデータソース名を識別します。代わりに、FileDSN を使用することもできます。 `FileDataSourceName=file-name`
Host	ホストコンピュータ名または IP アドレスとポート番号を識別します。 `Host=host-spec[:host-port]`
InitString	データベースサーバへの接続が確立された直後に実行される SQL 文を指定します。 `InitString=sql-statement`
Max pool size	接続プールの最大サイズを指定します。デフォルトは 100 です。 `Max pool size=number`
Min pool size	接続プールの最小サイズを指定します。デフォルトは 0 です。 `Min pool size=number`
Password	データベースユーザパスワードを指定します。代わりに、PWD を使用することもできます。 `PWD=passcode`

接続パラメータ	説明
Persist security info	Password (PWD) 接続パラメータを接続オブジェクトの ConnectionString プロパティに保持するかどうかを示します。デフォルトは **false** です。 `Persist security info=boolean-value` **true** に設定した場合、Password (PWD) 接続パラメータが元の接続文字列に指定されている場合、アプリケーションは ConnectionString プロパティからユーザのパスワードを取得できます。
Pooling	接続プーリングを有効または無効にします。デフォルトは **true** です。 `Pooling=boolean-value`

Server	データベースサーバのホスト名とポートを識別します。代わりに、ServerName を使用することもできます。
	`Server=sqla-server:port`
	このパラメータは、接続オブジェクトの読み取り専用プロパティ DataSource を設定します。プロパティは、次の方法でクエリできます。
	`String datasource = conn.DataSource;`

User ID	データベースユーザ名を指定します。代わりに、Username、UserID、UID を使用することもできます。 `UID=username` **注記** アプリケーション設定ファイルで接続文字列に User ID を使用できないようにします。UserID と UID を代わりに使用できます。

ここでは、すべての接続パラメータを説明していません。他の接続パラメータについては、データベース接続パラメータに関するトピックを参照してください。

接続文字列の例

- ConnectionString を設定する接続オブジェクトを作成し、データベースサーバに接続します。

```
SAConnection conn = new SAConnection(
```

```
    "Host=sqla-host:2638;Server=sqla-server;UserID=JSmith;Password=secret" );
conn.Open();
```

- 接続オブジェクトを作成し、その接続オブジェクトの ConnectionString を設定し、データベースサーバに接続します。データベースサーバに接続すると、SET TEMPORARY OPTION 文が実行され、認証されたアプリケーションのデータベーススシグネチャに対してアプリケーションシグネチャを検証します。

```
SAConnection conn = new SAConnection();
conn.ConnectionString =
    "Host=sqla-host:2638;Server=sqla-server;Database=sqla-db;" +
    "UID=JSmith;PWD=secret;" +
    "InitString=SET TEMPORARY OPTION connection_authentication=" +
    "'Company=MyCo;" +
    "Application=MyApp;" +
    "Signature=0fa55157edb8e14d818e...'"
conn.Open();
```

- ユーザから取得したユーザ ID とパスワードを使用して IP アドレスで識別されたコンピュータで実行されているデータベースサーバに接続します。

```
SAConnection conn = new SAConnection();
conn.ConnectionString =
    "Host=10.7.185.43:2638;Server=sqla-server;Database=sqla-db;" +
    "UID=" + user_name + ";PWD=" + pass_code;
conn.Open();
```

1.3.1.4.2 .NET 接続プーリング

SQL Anywhere .NET データプロバイダは、ネイティブ .NET 接続プーリングをサポートしています。接続プーリングを使用すると、アプリケーションは、データベースへの新しい接続を繰り返し作成しなくても、接続ハンドルをプールに保存して再使用できるようにして、既存の接続を再使用できます。

接続プーリングは、*Pooling* オプションを使用して有効または無効にします。デフォルトでは、接続プーリングは有効になっています。

最大プールサイズは、*Max Pool Size* オプションを使用して接続文字列に設定します。最小または初期プールサイズは、*Min Pool Size* オプションを使用して接続文字列に設定します。デフォルトの最大プールサイズは 100 で、デフォルトの最小プールサイズは 0 です。

```
"Data Source=SQL Anywhere 17 Demo;Pooling=true;Max Pool Size=50;Min Pool
Size=5;Password="+pwd
```

アプリケーションは、最初にデータベースに接続しようとするときに、指定したものと同じ接続パラメータを使用する既存の接続があるかどうかプールを調べます。一致する接続がある場合は、その接続が使用されます。ない場合は、新しい接続が使用されます。接続を切断すると、接続がプールに戻されて再使用できるようになります。

接続プーリングは、統合ログインや Kerberos ログインなどの非標準のデータベース認証ではサポートされていません。ユーザ ID とパスワードの認証のみがサポートされます。

データベースサーバも接続プーリングをサポートしています。この機能は、ConnectionPool (CPOOL) 接続パラメータを使用して制御します。ただし、SQL Anywhere .NET データプロバイダでは、このサーバ機能は使用されず、無効になります (CPOOL=NO)。代わりに、接続プーリングはすべて .NET クライアントアプリケーションで実行されます (クライアント側接続プーリング)。

1.3.1.4.3 接続状態

アプリケーションからデータベースへの接続が確立したら、接続がまだ開かれているかについて接続状態を確認してから、要求をデータベースサーバに送信できます。

接続が閉じている場合、ユーザに適切なメッセージを返信するか、接続を開き直すように試みることができます。

SAConnection クラスには、接続状態の確認に使用できる State プロパティがあります。取り得る状態値は ConnectionState.Open と ConnectionState.Closed です。

次のコードは、SAConnection オブジェクトが初期化されているかどうかを確認し、初期化されている場合は、接続が開かれていることを確認します。接続が開かれていない場合は、ユーザにメッセージが返されます。

```
if ( conn == null || conn.State != ConnectionState.Open )
{
    MessageBox.Show( "Connect to a database first", "Not connected" );
    return;
}
```

1.3.1.5 データへのアクセスとデータの操作

SQL Anywhere .NET データプロバイダでは、SACommand クラスまたは SADataAdapter クラスを使用してデータにアクセスできます。

SACommand オブジェクト

.NET のデータにアクセスして操作する場合、SACommand オブジェクトを使用する方法をお奨めします。

SACommand オブジェクトを使用して、データベースからデータを直接取得または修正する SQL 文を実行できます。SACommand オブジェクトを使用すると、データベースに対して直接 SQL 文を発行し、ストアドプロシージャを呼び出すことができます。

SACommand オブジェクトでは、SADataReader を使用してクエリまたはストアドプロシージャから読み込み専用結果セットが返されます。SADataReader は 1 回に 1 つのローのみを返しますが、クライアント側のライブラリはプリフェッチバッファリングを使用して 1 回に複数のローをプリフェッチするため、これによってパフォーマンスが低下することはありません。

SACommand オブジェクトを使用すると、オートコミットモードで操作しなくても、変更をトランザクションにグループ化できます。SATransaction オブジェクトを使用する場合、ローがロックされるため、他のユーザがこれらのローを修正できなくなります。

SADataAdapter オブジェクト

SADataAdapter オブジェクトは、結果セット全体を DataSet に取り出します。DataSet は、データベースから取り出されたデータの、切断されたストアです。DataSet のデータは編集できます。編集が終了すると、SADataAdapter オブジェクトは、DataSet の変更内容に応じてデータベースを更新します。SADataAdapter を使用する場合、他のユーザによる DataSet 内のローの修正を禁止する方法はありません。このため、発生する可能性がある競合を解消するためのロジックをアプリケーションに構築する必要があります。

SADataAdapter オブジェクトとは異なり、SACommand オブジェクト内で SADataReader を使用してデータベースからローをフェッチする方法の場合、パフォーマンス上の影響はありません。

このセクションの内容:

SACommand:ExecuteReader と ExecuteScalar を使用したデータのフェッチ [64 ページ]
SACommand オブジェクトを使用して、データベースに対して SQL 文を実行したりストアドプロシージャを呼び出したりできます。ExecuteReader または ExecuteScalar メソッドを使用して、データベースからデータを取り出すことができます。

SACommand: GetSchemaTable を使用した結果セットのスキーマのフェッチ [65 ページ]
GetSchemaTable メソッドを使用して結果セット内のカラムに関するスキーマ情報を取得できます。

SACommand: ExecuteNonQuery を使用したローの挿入、削除、更新 [66 ページ]
SACommand オブジェクトを使用してローを挿入、更新、削除するには、ExecuteNonQuery メソッドを使用します。ExecuteNonQuery メソッドは、結果セットを返さないクエリ (SQL 文またはストアドプロシージャ) を発行します。

SACommand: 新しく挿入したローのプライマリキー値の取得 [67 ページ]
更新するテーブルにオートインクリメントプライマリキーがある場合は、UUID を使用します。また、プライマリキーがプライマリキープールのものである場合は、ストアドプロシージャを使用して、データソースによって生成されたプライマリキー値を取得できます。

SADataAdapter: 概要 [68 ページ]
SADataAdapter オブジェクトは、結果セットを DataTable に取り出します。DataSet は、テーブルのコレクション (DataTables) と、これらのテーブル間の関係と制約です。DataSet は、.NET Framework に組み込まれており、データベースへの接続に使用されるデータプロバイダとは関係ありません。

SADataAdapter: Fill を使用したデータの DataTable へのフェッチ [70 ページ]
SADataAdapter を使用すると、Fill メソッドを使用して DataTable を表示グリッドにバインドすることによってクエリの結果を DataTable に設定し、結果セットを表示できます。

SADataAdapter: FillSchema を使用した DataTable のフォーマット [71 ページ]
SADataAdapter を使用すると、FillSchema メソッドを使用して、DataTable のスキーマが特定のクエリのスキーマと一致するように設定できます。DataTable のカラムの属性は、SADataAdapter オブジェクトの SelectCommand と一致します。

SADataAdapter:Add および Update を使用したローの挿入 [72 ページ]
SADataAdapter を使用すると、Add および Update メソッドを使用してローを挿入できます。

SADataAdapter:Delete および Update を使用したローの削除 [73 ページ]
SADataAdapter を使用すると、Delete および Update メソッドを使用してローを削除できます。

SADataAdapter:Update を使用したローの更新 [74 ページ]
SADataAdapter を使用すると、Update メソッドを使用してローを更新できます。

SADataAdapter: 新しく挿入したローのプライマリキー値の取得 [75 ページ]
更新するテーブルにオートインクリメントプライマリキーがある場合は、UUID を使用します。また、プライマリキーがプライマリキープールのものである場合は、ストアドプロシージャを使用して、データソースによって生成されたプライマリキー値を取得できます。

.NET アプリケーションでの BLOB 処理 [76 ページ]
長い文字列値またはバイナリデータをフェッチする場合、データを分割してフェッチするメソッドがいくつかあります。バイナリデータの場合は GetBytes メソッド、文字列データの場合は GetChars メソッドを使用します。

時間値 [77 ページ]
.NET Framework には Time 構造体はありません。データベースから時間値をフェッチするには、GetTimeSpan メソッドを使用します。

1.3.1.5.1 SACommand:ExecuteReader と ExecuteScalar を使用したデータのフェッチ

SACommand オブジェクトを使用して、データベースに対して SQL 文を実行したりストアドプロシージャを呼び出したりできます。ExecuteReader または ExecuteScalar メソッドを使用して、データベースからデータを取り出すことができます。

ExecuteReader

結果セットを返す SQL クエリを発行します。このメソッドは、前方専用、読み込み専用のカーソルを使用します。結果セット内のローを 1 方向で簡単にループできます。

ExecuteScalar

単一の値を返す SQL クエリを発行します。これは、結果セットの最初のローの最初のカラムの場合や、COUNT または AVG などの集約値を返す SQL 文の場合があります。このメソッドは、前方専用、読み込み専用のカーソルを使用します。

SACommand オブジェクトを使用する場合、SADataReader を使用して、ジョインに基づく結果セットを取り出すことができます。ただし、変更 (挿入、更新、または削除) を行うことができるのは、単一テーブルのデータのみです。ジョインに基づく結果セットは更新できません。

SADataReader を使用する場合、指定したデータ型で結果を返すための Get メソッドが複数あります。

Microsoft C# の ExecuteReader の例

次の Microsoft C# コードは、サンプルデータベースへの接続を開き、ExecuteReader メソッドを使用して Employees テーブルの従業員の姓を含む結果セットを作成します。

```
SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 17 Demo;Password="+pwd);
conn.Open();
SACommand cmd = new SACommand("SELECT Surname FROM Employees", conn);
SADataReader reader = cmd.ExecuteReader();
listEmployees.BeginUpdate();
while (reader.Read())
{
    listEmployees.Items.Add(reader.GetString(0));
}
listEmployees.EndUpdate();
reader.Close();
conn.Close();
```

Microsoft Visual Basic の ExecuteReader の例

次の Microsoft Visual Basic コードは、サンプルデータベースへの接続を開き、ExecuteReader メソッドを使用して Employees テーブルの従業員の姓を含む結果セットを作成します。

```
Dim conn As New SAConnection("Data Source=SQL Anywhere 17 Demo;Password="+pwd)
conn.Open()
Dim cmd As New SACommand("SELECT Surname FROM Employees", conn)
Dim reader As SADataReader = cmd.ExecuteReader()
```

```
ListEmployees.BeginUpdate()
Do While (reader.Read())
    ListEmployees.Items.Add(reader.GetString(0))
Loop
ListEmployees.EndUpdate()
reader.Close()
conn.Close()
```

Microsoft C# の ExecuteScalar の例

次の Microsoft C# コードは、サンプルデータベースへの接続を開き、ExecuteScalar メソッドを使用して Employees テーブルの男性従業員の人数を取得します。

```
SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 17 Demo;Password="+pwd);
conn.Open();
SACommand cmd = new SACommand(
    "SELECT COUNT(*) FROM Employees WHERE Sex = 'M'", conn );
int count = (int) cmd.ExecuteScalar();
textBox1.Text = count.ToString();
conn.Close();
```

1.3.1.5.2 SACommand: GetSchemaTable を使用した結果セットのスキーマのフェッチ

GetSchemaTable メソッドを使用して結果セット内のカラムに関するスキーマ情報を取得できます。

SADataReader クラスの GetSchemaTable メソッドは、現在の結果セットに関する情報を取得します。GetSchemaTable メソッドは、標準 .NET DataTable オブジェクトを返します。このオブジェクトは、結果セット内のすべてのカラムに関する情報 (カラムプロパティを含む) を提供します。

C# のスキーマ情報の例

次の例は、GetSchemaTable メソッドを使用して結果セットに関する情報を取得し、DataTable オブジェクトを画面上のデータグリッドにバインドします。

```
SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 17 Demo;Password="+pwd );
conn.Open();
SACommand cmd = new SACommand("SELECT * FROM Employees", conn);
SADataReader reader = cmd.ExecuteReader();
DataTable schema = reader.GetSchemaTable();
reader.Close();
conn.Close();
dataGridView1.DataSource = schema;
```

1.3.1.5.3 SACommand: ExecuteNonQuery を使用したローの挿入、削除、更新

SACommand オブジェクトを使用してローを挿入、更新、削除するには、ExecuteNonQuery メソッドを使用します。ExecuteNonQuery メソッドは、結果セットを返さないクエリ (SQL 文またはストアドプロシージャ) を発行します。

変更 (挿入、更新、または削除) を行うことができるのは、単一テーブルのデータのみです。ジョインに基づく結果セットは更新できません。SACommand オブジェクトを使用するには、データベースに接続してください。

AUTOINCREMENT プライマリキーのプライマリキー値の取得に関する詳細については、新しく挿入したローのプライマリキー値の取得に関するマニュアルを参照してください。

SQL 文の独立性レベルを設定するには、SACommand オブジェクトを SATransaction オブジェクトの一部として使用します。SATransaction オブジェクトを使用しないでデータを修正すると、プロバイダはオートコミットモードで動作し、実行した変更内容は即座に適用されます。

C# の ExecuteNonQuery DELETE および INSERT の例

次の例は、サンプルデータベースへの接続を開き、ExecuteNonQuery メソッドを使用して、ID が 600 以上の部署をすべて削除し、2 つの新しいローを Departments テーブルに追加します。そして、更新されたテーブルをデータグリッドに表示します。

```
SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 17 Demo;Password="+pwd);
conn.Open();
SACommand deleteCmd = new SACommand(
    "DELETE FROM Departments WHERE DepartmentID >= 600",
    conn);
deleteCmd.ExecuteNonQuery();
SACommand insertCmd = new SACommand(
    "INSERT INTO Departments(DepartmentID, DepartmentName) VALUES( ?, ? )",
    conn );
SAParameter parm = new SAParameter();
parm.SADbType = SADbType.Integer;
insertCmd.Parameters.Add( parm );
parm = new SAParameter();
parm.SADbType = SADbType.Char;
insertCmd.Parameters.Add( parm );
insertCmd.Parameters[0].Value = 600;
insertCmd.Parameters[1].Value = "Eastern Sales";
int recordsAffected = insertCmd.ExecuteNonQuery();
insertCmd.Parameters[0].Value = 700;
insertCmd.Parameters[1].Value = "Western Sales";
recordsAffected = insertCmd.ExecuteNonQuery();
SACommand selectCmd = new SACommand(
    "SELECT * FROM Departments", conn );
SADataReader dr = selectCmd.ExecuteReader();
System.Windows.Forms.DataGrid dataGrid;
dataGrid = new System.Windows.Forms.DataGrid();
dataGrid.Location = new Point(15, 50);
dataGrid.Size = new Size(275, 200);
dataGrid.CaptionText = "SACommand Example";
this.Controls.Add(dataGrid);
dataGrid.DataSource = dr;
dr.Close();
conn.Close();
```

C# の ExecuteNonQuery UPDATE の例

次の例は、サンプルデータベースへの接続を開き、ExecuteNonQuery メソッドを使用して、DepartmentID が 100 である Departments テーブルのすべてのローで DepartmentName カラムを "Engineering" に更新します。そして、更新されたテーブルをデータグリッドに表示します。

```
SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 17 Demo;Password="+pwd);
conn.Open();
SACommand updateCmd = new SACommand(
    "UPDATE Departments SET DepartmentName = 'Engineering' " +
    "WHERE DepartmentID = 100", conn );
int recordsAffected = updateCmd.ExecuteNonQuery();
SACommand selectCmd = new SACommand(
    "SELECT * FROM Departments", conn );
SADataReader dr = selectCmd.ExecuteReader();
System.Windows.Forms.DataGrid dataGrid;
dataGrid = new System.Windows.Forms.DataGrid();
dataGrid.Location = new Point(15, 50);
dataGrid.Size = new Size(275, 200);
dataGrid.CaptionText = "SACommand Example";
this.Controls.Add(dataGrid);
dataGrid.DataSource = dr;
dr.Close();
conn.Close();
```

関連情報

トランザクション処理 [79 ページ]
SACommand: 新しく挿入したローのプライマリキー値の取得 [67 ページ]

1.3.1.5.4 SACommand: 新しく挿入したローのプライマリキー値の取得

更新するテーブルにオートインクリメントプライマリキーがある場合は、UUID を使用します。また、プライマリキーがプライマリキープールのものである場合は、ストアドプロシージャを使用して、データソースによって生成されたプライマリキー値を取得できます。

C# の SACommand プライマリキーの例

次の例は、新しく挿入されたローに対して生成されるプライマリーキーの取得方法を示します。この例では、SACommand オブジェクトを使用して SQL ストアドプロシージャを呼び出し、返されるプライマリーキーを SAParameter オブジェクトを使用して取得します。デモンストレーションのため、この例ではサンプルテーブル (adodotnet_primarykey) とストアドプロシージャ (sp_adodotnet_primarykey) を作成して、ローの挿入とプライマリー値の取得に使用します。

```
SAConnection conn = new SAConnection(
```

```
    "Data Source=SQL Anywhere 17 Demo;Password="+pwd );
conn.Open();
SACommand cmd = conn.CreateCommand();
cmd.CommandText = "DROP TABLE adodotnet_primarykey";
cmd.ExecuteNonQuery();
cmd.CommandText = "CREATE TABLE IF NOT EXISTS adodotnet_primarykey (" +
    "ID INTEGER DEFAULT AUTOINCREMENT, " +
    "Name CHAR(40) )";
cmd.ExecuteNonQuery();
cmd.CommandText = "CREATE or REPLACE PROCEDURE sp_adodotnet_primarykey(" +
    "out p_id int, in p_name char(40) )" +
    "BEGIN " +
    "INSERT INTO adodotnet_primarykey( name ) VALUES( p_name );" +
    "SELECT @@IDENTITY INTO p_id;" +
    "END";
cmd.ExecuteNonQuery();
cmd.CommandText = "sp_adodotnet_primarykey";
cmd.CommandType = CommandType.StoredProcedure;
SAParameter parmId = new SAParameter();
parmId.SADbType = SADbType.Integer;
parmId.Direction = ParameterDirection.Output;
cmd.Parameters.Add(parmId);
SAParameter parmName = new SAParameter();
parmName.SADbType = SADbType.Char;
parmName.Direction = ParameterDirection.Input;
cmd.Parameters.Add(parmName);
parmName.Value = "R & D --- Command";
cmd.ExecuteNonQuery();
int id1 = (int)parmId.Value;
System.Console.WriteLine("Primary key=" + id1);
parmName.Value = "Marketing --- Command";
cmd.ExecuteNonQuery();
int id2 = (int)parmId.Value;
System.Console.WriteLine("Primary key=" + id2);
parmName.Value = "Sales --- Command";
cmd.ExecuteNonQuery();
int id3 = (int)parmId.Value;
System.Console.WriteLine("Primary key=" + id3);
parmName.Value = "Shipping --- Command";
cmd.ExecuteNonQuery();
int id4 = (int)parmId.Value;
System.Console.WriteLine("Primary key=" + id4);
cmd.CommandText = "SELECT * FROM adodotnet_primarykey";
cmd.CommandType = CommandType.Text;
SADataReader dr = cmd.ExecuteReader();
conn.Close();
dataGridView1.DataSource = dr;
```

1.3.1.5.5 SADataAdapter: 概要

SADataAdapter オブジェクトは、結果セットを DataTable に取り出します。DataSet は、テーブルのコレクション (DataTables) と、これらのテーブル間の関係と制約です。DataSet は、.NET Framework に組み込まれており、データベースへの接続に使用されるデータプロバイダとは関係ありません。

SADataAdapter を使用する場合、DataTable を設定し、DataTable への変更内容に基づいてデータベースを更新するために、データベースに接続しておく必要があります。ただし、DataTable を一度設定すれば、データベースと切断されていても DataTable を修正できます。

変更内容をデータベースに即座に適用したくない場合、WriteXml メソッドを使用して DataSet (データかスキーマまたはその両方を含む) を XML ファイルに書き込むことができます。これにより、後で ReadXml メソッドを使用して DataSet をロードすることで、変更を適用できるようになります。以下では 2 つの例を示します。

```
ds.WriteXml("Employees.xml");
ds.WriteXml("EmployeesWithSchema.xml", XmlWriteMode.WriteSchema);
```

詳細については、.NET Framework のマニュアルの WriteXml と ReadXml を参照してください。

Update メソッドを呼び出して DataSet からデータベースに変更を適用すると、SADataAdapter は、実行された変更を分析してから、必要に応じて適切な文 (INSERT、UPDATE、または DELETE) を呼び出します。DataSet を使用する場合、変更 (挿入、更新、または削除) を適用できるのは、単一テーブル内のデータに対してのみです。ジョインに基づく結果セットは更新できません。更新しようとしているローを別のユーザがロックしている場合、例外がスローされます。

⚠ 警告

DataSet に対する変更はすべて、接続が切断している間に行われます。データベース内のこれらのローはアプリケーションによってロックされません。DataSet の変更がデータベースに適用されるときに発生する可能性がある競合を解消できるようアプリケーションを設計してください。これは、自分の変更がデータベースに適用される前に自分が修正しているデータを別のユーザが変更しようとするような場合です。

SADataAdapter を使用するときの競合の解消

SADataAdapter オブジェクトを使用する場合、データベース内のローはロックされません。つまり、DataSet からデータベースに変更を適用するときに競合が発生する可能性があります。このため、アプリケーションには、発生する競合を解消または記録するロジックを採用する必要があります。

アプリケーションのロジックが対応すべき競合には、次のようなものがあります。

ユニークなプライマリキー

2 人のユーザが新しいローをテーブルに挿入する場合、ローごとにユニークなプライマリキーが必要です。AUTOINCREMENT プライマリキーがあるテーブルの場合、DataSet の値とデータソースの値の同期がとれなくなる可能性があります。

新たに挿入されたローに関して AUTOINCREMENT プライマリキーの値を取得できます。

同じ値に対して行われた更新

2 人のユーザが同じ値を修正する場合、どちらの値が正しいかを確認するロジックをアプリケーションに採用する必要があります。

スキーマの変更

DataSet で更新したテーブルのスキーマを別のユーザが修正する場合、データベースに変更を適用するとこの更新が失敗します。

データの同時実行性

同時実行アプリケーションは、一連の一貫性のあるデータを参照する必要があります。SADataAdapter はフェッチするローをロックしないため、いったん DataSet を取り出してからオフラインで処理する場合、別のユーザがデータベース内の値を更新できます。

これらの潜在的な問題の多くは、SACommand、SADataReader、SATransaction オブジェクトを使用して変更をデータベースに適用することによって回避できます。このうち、SATransaction オブジェクトを使用することをお奨めします。これは、

SATransaction オブジェクトを使用すると、トランザクションに独立性レベルを設定できるほか、他のユーザが修正できないようにローをロックできるためです。

競合の解消プロセスを簡素化するために、INSERT、UPDATE、DELETE 文をストアドプロシージャ呼び出しとして設定できます。INSERT、UPDATE、DELETE 文をストアドプロシージャに入れることによって、オペレーションが失敗したときのエラーを取得できます。文のほかにも、エラー処理のロジックをストアドプロシージャに追加することによって、オペレーションが失敗したときに、エラーをログファイルに記録したりオペレーションを再試行したりするなど、適切なアクションが行われるようにすることができます。

関連情報

SADataAdapter: 新しく挿入したローのプライマリキー値の取得 [75 ページ]
SACommand: ExecuteNonQuery を使用したローの挿入、削除、更新 [66 ページ]

1.3.1.5.6 SADataAdapter: Fill を使用したデータの DataTable へのフェッチ

SADataAdapter を使用すると、Fill メソッドを使用して DataTable を表示グリッドにバインドすることによってクエリの結果を DataTable に設定し、結果セットを表示できます。

SADataAdapter を設定する場合、結果セットを返す SQL 文を指定できます。Fill を呼び出して DataTable を設定する場合、前方専用、読み込み専用のカーソルを使用してすべてのローが 1 回のオペレーションでフェッチされます。結果セット内のすべてのローが読み込まれると、カーソルは閉じます。DataTable の行に行われた変更は、Update メソッドを使用してデータベースに反映できます。

SADataAdapter オブジェクトを使用して、ジョインに基づく結果セットを取り出すことができます。ただし、変更 (挿入、更新、または削除) を行うことができるのは、単一テーブルのデータのみです。ジョインに基づく結果セットは更新できません。

> ⚠ 警告
>
> DataTable に対して行う変更は、元のデータベーステーブルとは別に行われます。データベース内のこれらのローはアプリケーションによってロックされません。DataTable の変更がデータベースに適用されるときに発生する可能性がある競合を解消できるようアプリケーションを設計してください。これは、自分の変更がデータベースに適用される前に自分が修正しているデータを別のユーザが変更しようとするような場合です。

DataTable を使用した C# の SADataAdapter Fill の例

次の例は、SADataAdapter を使用して DataTable を設定する方法を示します。Results という新しい DataTable オブジェクトと、新しい SADataAdapter オブジェクトを作成します。SADataAdapter の Fill メソッドを使用して、クエリの結果を DataTable に設定します。そして、DataTable を画面上のグリッドにバインドします。

```
SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 17 Demo;Password="+pwd );
```

```
conn.Open();
DataTable dt = new DataTable("Results");
SADataAdapter da = new SADataAdapter("SELECT * FROM Employees", conn);
da.Fill(dt);
conn.Close();
dataGridView1.DataSource = dt;
```

DataSet を使用した C# の SADataAdapter Fill の例

次の例は、SADataAdapter を使用して DataTable を設定する方法を示します。新しい DataSet オブジェクトと、新しい SADataAdapter オブジェクトを作成します。SADataAdapter の Fill メソッドを使用して、Results という DataTable テーブルを DataSet 内に作成した後、クエリの結果を設定します。そして、Results DataTable を画面上のグリッドにバインドします。

```
SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 17 Demo;Password="+pwd );
conn.Open();
DataSet ds = new DataSet();
SADataAdapter da = new SADataAdapter("SELECT * FROM Employees", conn);
da.Fill(ds, "Results");
conn.Close();
dataGridView1.DataSource = ds.Tables["Results"];
```

1.3.1.5.7 SADataAdapter: FillSchema を使用した DataTable のフォーマット

SADataAdapter を使用すると、FillSchema メソッドを使用して、DataTable のスキーマが特定のクエリのスキーマと一致するように設定できます。DataTable のカラムの属性は、SADataAdapter オブジェクトの SelectCommand と一致します。

Fill メソッドとは異なり、DataTable に行は保存されません。

DataTable を使用した C# の SADataAdapter FillSchema の例

次の例は、FillSchema メソッドを使用して結果セットと同じスキーマを持つ新しい DataTable オブジェクトを設定する方法を示します。そして、Additions DataTable を画面上のグリッドにバインドします。

```
SAConnection conn = new SAConnection( "Data Source=SQL Anywhere 17
Demo;Password="+pwd );
conn.Open();
SADataAdapter da = new SADataAdapter("SELECT * FROM Employees", conn);
DataTable dt = new DataTable("Additions");
da.FillSchema(dt, SchemaType.Source);
conn.Close();
dataGridView1.DataSource = dt;
```

DataSet を使用した C# の SADataAdapter FillSchema の例

次の例は、FillSchema メソッドを使用して結果セットと同じスキーマを持つ新しい DataTable オブジェクトを設定する方法を示します。Merge メソッドを使用して DataTable を DataSet に追加します。そして、Additions DataTable を画面上のグリッドにバインドします。

```
SAConnection conn = new SAConnection( "Data Source=SQL Anywhere 17
Demo;Password="+pwd );
conn.Open();
SADataAdapter da = new SADataAdapter("SELECT * FROM Employees", conn);
DataTable dt = new DataTable("Additions");
da.FillSchema(dt, SchemaType.Source);
DataSet ds = new DataSet();
ds.Merge(dt);
conn.Close();
dataGridView1.DataSource = ds.Tables["Additions"];
```

1.3.1.5.8 SADataAdapter:Add および Update を使用したローの挿入

SADataAdapter を使用すると、Add および Update メソッドを使用してローを挿入できます。

C# の SADataAdapter Insert の例

この例は、SADataAdapter の Update メソッドを使用してテーブルにローを追加する方法を示します。この例では、SADataAdapter の SelectCommand プロパティと Fill メソッドを使用して、Departments テーブルを DataTable にフェッチします。次に、2 つの新しいローを DataTable に追加し、SADataAdapter の InsertCommand プロパティと Update メソッドを使用して、DataTable からの Departments テーブルを更新します。

```
SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 17 Demo;Password="+pwd );
conn.Open();
SACommand deleteCmd = new SACommand(
    "DELETE FROM Departments WHERE DepartmentID >= 600", conn);
deleteCmd.ExecuteNonQuery();
SADataAdapter da = new SADataAdapter();
da.MissingMappingAction = MissingMappingAction.Passthrough;
da.MissingSchemaAction = MissingSchemaAction.Add;
da.SelectCommand = new SACommand(
    "SELECT * FROM Departments", conn );
da.InsertCommand = new SACommand(
    "INSERT INTO Departments( DepartmentID, DepartmentName ) " +
    "VALUES( ?, ? )", conn );
da.InsertCommand.UpdatedRowSource =
    UpdateRowSource.None;
SAParameter parm = new SAParameter();
parm.SADbType = SADbType.Integer;
parm.SourceColumn = "DepartmentID";
parm.SourceVersion = DataRowVersion.Current;
da.InsertCommand.Parameters.Add( parm );
parm = new SAParameter();
parm.SADbType = SADbType.Char;
```

```
parm.SourceColumn = "DepartmentName";
parm.SourceVersion = DataRowVersion.Current;
da.InsertCommand.Parameters.Add( parm );
DataTable dataTable = new DataTable( "Departments" );
int rowCount = da.Fill( dataTable );
DataRow row1 = dataTable.NewRow();
row1[0] = 600;
row1[1] = "Eastern Sales";
dataTable.Rows.Add( row1 );
DataRow row2 = dataTable.NewRow();
row2[0] = 700;
row2[1] = "Western Sales";
dataTable.Rows.Add( row2 );
rowCount = da.Update( dataTable );
dataTable.Clear();
rowCount = da.Fill( dataTable );
conn.Close();
dataGridView1.DataSource = dataTable;
```

1.3.1.5.9 SADataAdapter:Delete および Update を使用したローの削除

SADataAdapter を使用すると、Delete および Update メソッドを使用してローを削除できます。

C# の SADataAdapter Delete の例

次の例は、SADataAdapter の Update メソッドを使用してテーブルからローを削除する方法を示します。この例では、2 つの新しいローを Departments テーブルに追加した後、SADataAdapter の SelectCommand プロパティと Fill メソッドを使用して、Departments テーブルを DataTable にフェッチします。次に、DataTable からいくつかのローを削除し、SADataAdapter の DeleteCommand プロパティと Update メソッドを使用して、DataTable からの Departments テーブルを更新します。

```
SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 17 Demo;Password="+pwd );
conn.Open();
SACommand prepCmd = new SACommand("", conn);
prepCmd.CommandText =
    "DELETE FROM Departments WHERE DepartmentID >= 600";
prepCmd.ExecuteNonQuery();
prepCmd.CommandText =
    "INSERT INTO Departments VALUES (600, 'Eastern Sales', 902)";
prepCmd.ExecuteNonQuery();
prepCmd.CommandText =
    "INSERT INTO Departments VALUES (700, 'Western Sales', 902)";
prepCmd.ExecuteNonQuery();
SADataAdapter da = new SADataAdapter();
da.MissingMappingAction = MissingMappingAction.Passthrough;
da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
da.SelectCommand = new SACommand(
    "SELECT * FROM Departments", conn);
da.DeleteCommand = new SACommand(
    "DELETE FROM Departments WHERE DepartmentID = ?",
    conn);
da.DeleteCommand.UpdatedRowSource = UpdateRowSource.None;
SAParameter parm = new SAParameter();
```

```
parm.SADbType = SADbType.Integer;
parm.SourceColumn = "DepartmentID";
parm.SourceVersion = DataRowVersion.Original;
da.DeleteCommand.Parameters.Add(parm);
DataTable dataTable = new DataTable("Departments");
int rowCount = da.Fill(dataTable);
foreach (DataRow row in dataTable.Rows)
{
    if (Int32.Parse(row[0].ToString()) > 500)
    {
        row.Delete();
    }
}
rowCount = da.Update(dataTable);
dataTable.Clear();
rowCount = da.Fill(dataTable);
conn.Close();
dataGridView1.DataSource = dataTable;
```

1.3.1.5.10 SADataAdapter:Update を使用したローの更新

SADataAdapter を使用すると、Update メソッドを使用してローを更新できます。

C# の SADataAdapter Update の例

次の例は、SADataAdapter の Update メソッドを使用してテーブル内のローを更新する方法を示します。この例では、2 つの新しいローを Departments テーブルに追加した後、SADataAdapter の SelectCommand プロパティと Fill メソッドを使用して、Departments テーブルを DataTable にフェッチします。次に、DataTable の値の一部を変更し、SADataAdapter の UpdateCommand プロパティと Update メソッドを使用して、DataTable からの Departments テーブルを更新します。

```
SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 17 Demo;Password="+pwd );
conn.Open();
SACommand prepCmd = new SACommand("", conn);
prepCmd.CommandText =
    "DELETE FROM Departments WHERE DepartmentID >= 600";
prepCmd.ExecuteNonQuery();
prepCmd.CommandText =
    "INSERT INTO Departments VALUES (600, 'Eastern Sales', 902)";
prepCmd.ExecuteNonQuery();
prepCmd.CommandText =
    "INSERT INTO Departments VALUES (700, 'Western Sales', 902)";
prepCmd.ExecuteNonQuery();
SADataAdapter da = new SADataAdapter();
da.MissingMappingAction = MissingMappingAction.Passthrough;
da.MissingSchemaAction = MissingSchemaAction.Add;
da.SelectCommand = new SACommand(
    "SELECT * FROM Departments", conn );
da.UpdateCommand = new SACommand(
    "UPDATE Departments SET DepartmentName = ? " +
    "WHERE DepartmentID = ?",
    conn );
da.UpdateCommand.UpdatedRowSource = UpdateRowSource.None;
SAParameter parm = new SAParameter();
parm.SADbType = SADbType.Char;
parm.SourceColumn = "DepartmentName";
```

```
parm.SourceVersion = DataRowVersion.Current;
da.UpdateCommand.Parameters.Add( parm );
parm = new SAParameter();
parm.SADbType = SADbType.Integer;
parm.SourceColumn = "DepartmentID";
parm.SourceVersion = DataRowVersion.Original;
da.UpdateCommand.Parameters.Add( parm );
DataTable dataTable = new DataTable( "Departments" );
int rowCount = da.Fill( dataTable );
foreach ( DataRow row in dataTable.Rows )
{
    if (Int32.Parse(row[0].ToString()) > 500)
    {
        row[1] = (string)row[1] + "_Updated";
    }
}
rowCount = da.Update( dataTable );
dataTable.Clear();
rowCount = da.Fill( dataTable );
conn.Close();
dataGridView1.DataSource = dataTable;
```

1.3.1.5.11 SADataAdapter: 新しく挿入したローのプライマリキー値の取得

更新するテーブルにオートインクリメントプライマリキーがある場合は、UUID を使用します。また、プライマリキーがプライマリキープールのものである場合は、ストアドプロシージャを使用して、データソースによって生成されたプライマリキー値を取得できます。

C# の SADataAdapter プライマリキーの例

次の例は、新しく挿入されたローに対して生成されるプライマリーキーの取得方法を示します。この例では、SADataAdapter オブジェクトを使用して SQL ストアドプロシージャを呼び出し、返されるプライマリーキーを SAParameter オブジェクトを使用して取得します。デモンストレーションのため、この例ではサンプルテーブル (adodotnet_primarykey) とストアドプロシージャ (sp_adodotnet_primarykey) を作成して、ローの挿入とプライマリー値の取得に使用します。

```
SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 17 Demo;Password="+pwd );
conn.Open();
SACommand cmd = conn.CreateCommand();
cmd.CommandText = "DROP TABLE adodotnet_primarykey";
cmd.ExecuteNonQuery();
cmd.CommandText = "CREATE TABLE IF NOT EXISTS adodotnet_primarykey (" +
    "ID INTEGER DEFAULT AUTOINCREMENT, " +
    "Name CHAR(40) )";
cmd.ExecuteNonQuery();
cmd.CommandText = "CREATE or REPLACE PROCEDURE sp_adodotnet_primarykey(" +
    "out p_id int, in p_name char(40) )" +
    "BEGIN " +
    "INSERT INTO adodotnet_primarykey( name ) VALUES( p_name );" +
    "SELECT @@IDENTITY INTO p_id;" +
    "END";
cmd.ExecuteNonQuery();
SADataAdapter da = new SADataAdapter();
da.MissingMappingAction = MissingMappingAction.Passthrough;
```

```
da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
da.SelectCommand = new SACommand(
    "SELECT * FROM adodotnet_primarykey", conn);
da.InsertCommand = new SACommand(
    "sp_adodotnet_primarykey", conn);
da.InsertCommand.CommandType = CommandType.StoredProcedure;
da.InsertCommand.UpdatedRowSource = UpdateRowSource.OutputParameters;
SAParameter parmId = new SAParameter();
parmId.SADbType = SADbType.Integer;
parmId.Direction = ParameterDirection.Output;
parmId.SourceColumn = "ID";
parmId.SourceVersion = DataRowVersion.Current;
da.InsertCommand.Parameters.Add(parmId);
SAParameter parmName = new SAParameter();
parmName.SADbType = SADbType.Char;
parmName.Direction = ParameterDirection.Input;
parmName.SourceColumn = "Name";
parmName.SourceVersion = DataRowVersion.Current;
da.InsertCommand.Parameters.Add(parmName);
DataTable dataTable = new DataTable("Departments");
da.FillSchema(dataTable, SchemaType.Source);
DataRow row = dataTable.NewRow();
row[0] = -1;
row[1] = "R & D --- Adapter";
dataTable.Rows.Add(row);
row = dataTable.NewRow();
row[0] = -2;
row[1] = "Marketing --- Adapter";
dataTable.Rows.Add(row);
row = dataTable.NewRow();
row[0] = -3;
row[1] = "Sales --- Adapter";
dataTable.Rows.Add(row);
row = dataTable.NewRow();
row[0] = -4;
row[1] = "Shipping --- Adapter";
dataTable.Rows.Add(row);
DataSet ds = new DataSet();
ds.Merge(dataTable);
da.Update(ds, "Departments");
conn.Close();
dataGridView1.DataSource = ds.Tables["Departments"];
```

1.3.1.5.12 .NET アプリケーションでの BLOB 処理

長い文字列値またはバイナリデータをフェッチする場合、データを分割してフェッチするメソッドがいくつかあります。バイナリデータの場合は GetBytes メソッド、文字列データの場合は GetChars メソッドを使用します。

それ以外の場合、データベースからフェッチする他のデータと同じ方法で BLOB データが処理されます。

C# の GetChars BLOB の例

次の例は、結果セットから 3 つのカラムを読み込みます。最初の 2 つのカラムは整数で、3 番目のカラムは LONG VARCHAR です。GetChars メソッドを使用して 3 番目のカラムを 100 文字のチャンクに読み込み、このカラムの長さを計算します。

```
SAConnection conn = new SAConnection(
```

```
    "Data Source=SQL Anywhere 17 Demo;Password="+pwd );
conn.Open();
SACommand cmd = new SACommand("SELECT * FROM MarketingInformation", conn);
SADataReader reader = cmd.ExecuteReader();
int idValue;
int productIdValue;
int length = 100;
char[] buf = new char[length];
while (reader.Read())
{
    idValue = reader.GetInt32(0);
    productIdValue = reader.GetInt32(1);
    long blobLength = 0;
    long charsRead;
    while ((charsRead = reader.GetChars(2, blobLength, buf, 0, length))
            == (long)length)
    {
        blobLength += charsRead;
    }
    blobLength += charsRead;
}
reader.Close();
conn.Close();
```

1.3.1.5.13 時間値

.NET Framework には Time 構造体はありません。データベースから時間値をフェッチするには、GetTimeSpan メソッドを使用します。

このメソッドは、データを .NET Framework TimeSpan オブジェクトとして返します。

C# の TimeSpan の例

次の例は、時間を TimeSpan として返す GetTimeSpan メソッドを使用します。

```
SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 17 Demo;Password="+pwd );
conn.Open();
SACommand cmd = new SACommand("SELECT 123, CURRENT TIME", conn);
SADataReader reader = cmd.ExecuteReader();
while (reader.Read())
{
    int ID = reader.GetInt32(0);
    TimeSpan time = reader.GetTimeSpan(1);
}
reader.Close();
conn.Close();
```

1.3.1.6 ストアドプロシージャ

SQL Anywhere .NET データプロバイダでは SQL ストアドプロシージャを使用できます。

ExecuteReader メソッドを使用して、結果セットを返すストアドプロシージャを呼び出します。また、ExecuteNonQuery メソッドを使用して、結果セットを返さないストアドプロシージャを呼び出します。ExecuteScalar メソッドを使用して、単一値のみを返すストアドプロシージャを呼び出します。

SAParameter オブジェクトを追加してパラメータをストアドプロシージャに渡すことができます。

C# のパラメータ付きストアドプロシージャ呼び出しの例

次の例は、ストアドプロシージャを呼び出してパラメータを渡す 2 つの方法を示します。この例では、ストアドプロシージャから返される結果セットを SADataReader を使用してフェッチします。

```
SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 17 Demo;Password="+pwd );
conn.Open();
bool method1 = true;
SACommand cmd = new SACommand("", conn);
if (method1)
{
    cmd.CommandText = "ShowProductInfo";
    cmd.CommandType = CommandType.StoredProcedure;
}
else
{
    cmd.CommandText = "call ShowProductInfo(?)";
    cmd.CommandType = CommandType.Text;
}
SAParameter param = cmd.CreateParameter();
param.SADbType = SADbType.Integer;
param.Direction = ParameterDirection.Input;
param.Value = 301;
cmd.Parameters.Add(param);
SADataReader reader = cmd.ExecuteReader();
reader.Read();
int ID = reader.GetInt32(0);
string name = reader.GetString(1);
string description = reader.GetString(2);
decimal price = reader.GetDecimal(6);
reader.Close();
listBox1.BeginUpdate();
listBox1.Items.Add("Name=" + name +
    " Description=" + description + " Price=" + price);
listBox1.EndUpdate();
conn.Close();
```

関連情報

SACommand: ExecuteNonQuery を使用したローの挿入、削除、更新 [66 ページ]
SACommand:ExecuteReader と ExecuteScalar を使用したデータのフェッチ [64 ページ]

1.3.1.7 トランザクション処理

SATransaction オブジェクトを使用して複数の文をグループ化できます。各トランザクションは、データベースへの変更内容を確定する COMMIT メソッドの呼び出し、またはトランザクションのすべてのオペレーションをキャンセルする ROLLBACK メソッドの呼び出しで終了します。

トランザクションが完了したら、さらに変更を行うための SATransaction オブジェクトを新しく作成する必要があります。この動作は、COMMIT または ROLLBACK を実行した後もトランザクションが閉じられるまで持続する ODBC および Embedded SQL とは異なります。

トランザクションを作成しない場合、デフォルトでは、.NET データプロバイダはオートコミットモードで動作します。挿入、更新、または削除の各処理後には COMMIT が暗黙的に実行され、オペレーションが完了すると、データベースが変更されます。この場合、変更はロールバックできません。

トランザクションの独立性レベルの設定

デフォルトでは、トランザクションに対してデータベースの独立性レベルが使用されます。トランザクションを開始するときに IsolationLevel プロパティを使用してトランザクションに対して独立性レベルを指定できます。独立性レベルは、トランザクション内で実行されるすべての文に対して適用されます。.NET データプロバイダは、スナップショットアイソレーションをサポートしています。

SQL 文を実行するときにデータベースサーバで使用されるロックは、トランザクションの独立性レベルによって異なります。

分散トランザクション処理

.NET 2.0 フレームワークで、トランザクションアプリケーションを記述するためのクラスが含まれる新しいネームスペース System.Transactions が導入されました。クライアントアプリケーションで 1 つまたは複数の参加者が存在する分散トランザクションを作成し、そのトランザクションに参加できます。クライアントアプリケーションでは、TransactionScope クラスを使用して、暗黙的にトランザクションを作成できます。接続オブジェクトでは、TransactionScope によって作成されたアンビエントトランザクションの存在を検出し、自動的にエンリストできます。クライアントアプリケーションでは、CommittableTransaction を作成し、EnlistTransaction メソッドを呼び出してエンリストすることもできます。この機能は .NET データプロバイダでサポートされます。分散トランザクションは、パフォーマンスに大きい影響を与えます。データベーストランザクションは非分散トランザクションのみで使用してください。

C# の SATransaction の例

次の例は、コミットやロールバックができるように INSERT をトランザクションにラップする方法を示します。SATransaction オブジェクトを使用してトランザクションを作成し、SACommand オブジェクトを使用して SQL 文の実行にリンクします。独立性レベル 2 (RepeatableRead) を指定して、他のデータベースユーザがローを更新できないようにします。トランザクションがコ

ミットまたはロールバックされると、ローのロックは解放されます。トランザクションを使用しない場合、.NET データプロバイダはオートコミットモードで動作し、データベースの変更内容をロールバックできません。

```
SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 17 Demo;Password="+pwd );
conn.Open();
string stmt = "UPDATE Products SET UnitPrice = 2000.00 " +
    "WHERE Name = 'Tee shirt'";
bool goAhead = false;
SATransaction trans = conn.BeginTransaction(SAIsolationLevel.RepeatableRead);
SACommand cmd = new SACommand(stmt, conn, trans);
int rowsAffected = cmd.ExecuteNonQuery();
if (goAhead)
    trans.Commit();
else
    trans.Rollback();
conn.Close();
```

1.3.1.8 Microsoft .NET エラー処理

アプリケーションは発生するあらゆるエラーを処理するように設計する必要があります。例外がスローされるときに SAException オブジェクトが作成されます。例外に関する情報は、SAException オブジェクトに格納されます。

SQL Anywhere .NET データプロバイダは、実行時にエラーが発生した場合はいつでも SAException オブジェクトを作成して例外をスローします。各 SAException オブジェクトは SAError オブジェクトのリストから成り、これらのエラーオブジェクトにはエラーメッセージとコードが含まれます。

エラーは競合とは異なります。競合は、データベースに変更が適用されたときに発生します。このため、アプリケーションには、競合が発生したときに正しい値を計算したり競合のログを取ったりするプロセスを採用する必要があります。

Microsoft C# のエラー処理の例

次の Microsoft C# コードは、サンプルデータベースとの接続を開くボタンクリックハンドラを作成します。接続を確立できない場合、例外ハンドラにより 1 つ以上のメッセージが表示されます。

```
private void button1_Click(object sender, EventArgs e)
{
    SAConnection conn = new SAConnection(
        "Data Source=SQL Anywhere 17 Demo;Password="+pwd);
    try
    {
        conn.Open();
    }
    catch (SAException ex)
    {
        for (int i = 0; i < ex.Errors.Count; i++)
        {
            MessageBox.Show(ex.Errors[i].Source + " : " +
             ex.Errors[i].Message + " (" +
             ex.Errors[i].NativeError.ToString() + ")",
             "Failed to connect");
        }
    }
    catch (Exception ex)
```

```
    {
        MessageBox.Show(ex.Message, "Failed to connect");
    }
}
```

Microsoft Visual Basic のエラー処理の例

次の Microsoft Visual Basic コードは、サンプルデータベースとの接続を開くボタンクリックハンドラを作成します。接続を確立できない場合、例外ハンドラにより 1 つ以上のメッセージが表示されます。

```
Private Sub Button1_Click(ByVal sender As System.Object, _
     ByVal e As System.EventArgs) Handles Button1.Click
    Dim conn As New SAConnection("Data Source=SQL Anywhere 17 Demo;Password="+pwd)
    Try
        conn.Open()
    Catch ex As SAException
        For i = 0 To ex.Errors.Count - 1
            MessageBox.Show(ex.Errors(i).Source & " : " & _
              ex.Errors(i).Message & " (" & _
              ex.Errors(i).NativeError.ToString() & ")", _
              "Failed to connect")
        Next i
    Catch ex as Exception
        MessageBox.Show(ex.Message, "Failed to connect")
    End Try
End Sub
```

関連情報

Simple サンプルプロジェクトの説明 [103 ページ]
Table Viewer サンプルプロジェクトの説明 [107 ページ]

1.3.1.9 Entity Framework のサポート

SQL Anywhere .NET データプロバイダでは Entity Framework 5.0 および 6.0 がサポートされています。Entity Framework は Microsoft から別個のパッケージとして入手できます。

Entity Framework 5.0 または 6.0 を使用するには、Microsoft の NuGet Package Manager を使用して Microsoft Visual Studio に追加する必要があります。

Entity Framework の新機能の 1 つに Code First があります。この機能では異なる開発ワークフローが可能です。Microsoft Visual C# .NET または Microsoft Visual Basic .NET のクラスを記述するだけでデータモデルが定義され、データベースオブジェクトにマッピングされます。デザイナを開く必要も XML マッピングファイルを定義する必要もありません。または、データの注釈または Fluent API を使用して追加の設定を実行することもできます。モデルを使用して、データベーススキーマを生成するか、既存のデータベースにマッピングすることができます。

次の例は、このモデルを使用して新しいデータベースオブジェクトを作成します。

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Linq;
using Sap.Data.SQLAnywhere;
namespace CodeFirstExample
{
    [Table( "EdmCategories", Schema = "DBA" )]
    public class Category
    {
        public string CategoryID { get; set; }
        [MaxLength( 64 )]
        public string Name { get; set; }
        public virtual ICollection<Product> Products { get; set; }
    }
    [Table( "EdmProducts", Schema = "DBA" )]
    public class Product
    {
        public int ProductId { get; set; }
        [MaxLength( 64 )]
        public string Name { get; set; }
        public string CategoryID { get; set; }
        public virtual Category Category { get; set; }
    }
    [Table( "EdmSuppliers", Schema = "DBA" )]
    public class Supplier
    {
        [Key]
        public string SupplierCode { get; set; }
        [MaxLength( 64 )]
        public string Name { get; set; }
    }
    public class Context : DbContext
    {
        public Context() : base() { }
        public Context( string connStr ) : base( connStr ) { }
        public DbSet<Category> Categories { get; set; }
        public DbSet<Product> Products { get; set; }
        public DbSet<Supplier> Suppliers { get; set; }
        protected override void OnModelCreating( DbModelBuilder modelBuilder )
        {
            modelBuilder.Entity<Supplier>().Property( s => s.Name ).IsRequired();
        }
    }

    class Program
    {
        static void Main( string[] args )
        {
            Database.DefaultConnectionFactory = new SAConnectionFactory();
            Database.SetInitializer<Context>(
                new DropCreateDatabaseAlways<Context>() );
            using ( var db = new Context(
                "DSN=SQL Anywhere 17 Demo;Password="+pwd ) )
            {
                var query = db.Products.ToList();
            }
        }
    }
}
```

この例を構築して実行するには、次のアセンブリ参照を追加する必要があります。

```
EntityFramework
Sap.Data.SQLAnywhere.v4.5
System.ComponentModel.DataAnnotations
System.Data.Entity
```

次の例は、既存のデータベースにマッピングします。

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Linq;
using Sap.Data.SQLAnywhere;
namespace CodeFirstExample
{
    [Table( "Customers", Schema = "GROUPO" )]
    public class Customer
    {
        [Key()]
        public int ID { get; set; }
        public string SurName { get; set; }
        public string GivenName { get; set; }
        public string Street { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public string Country { get; set; }
        public string PostalCode { get; set; }
        public string Phone { get; set; }
        public string CompanyName { get; set; }
        public virtual ICollection<Contact> Contacts { get; set; }
    }
    [Table( "Contacts", Schema = "GROUPO" )]
    public class Contact
    {
        [Key()]
        public int ID { get; set; }
        public string SurName { get; set; }
        public string GivenName { get; set; }
        public string Title { get; set; }
        public string Street { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public string Country { get; set; }
        public string PostalCode { get; set; }
        public string Phone { get; set; }
        public string Fax { get; set; }
        [ForeignKey( "Customer" )]
        public int CustomerID { get; set; }
        public virtual Customer Customer { get; set; }
    }
    public class Context : DbContext
    {
        public Context() : base() { }
        public Context( string connStr ) : base( connStr ) { }
        public DbSet<Contact> Contacts { get; set; }
        public DbSet<Customer> Customers { get; set; }
    }
    class Program
    {
        static void Main( string[] args )
        {
            Database.DefaultConnectionFactory = new SAConnectionFactory();
            Database.SetInitializer<Context>( null );
            using ( var db = new Context(
```

```
                "DSN=SQL Anywhere 17 Demo;Password="+pwd ) )
            {
                foreach ( var customer in db.Customers.ToList() )
                {
                    Console.WriteLine( "Customer - " + string.Format(
                      "{0}, {1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}, {9}",
                        customer.ID, customer.SurName, customer.GivenName,
                        customer.Street, customer.City, customer.State,
                        customer.Country, customer.PostalCode,
                        customer.Phone, customer.CompanyName ) );
                    foreach ( var contact in customer.Contacts )
                    {
                        Console.WriteLine( "    Contact - " + string.Format(
                          "{0}, {1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}, {9}, {10}",
                            contact.ID, contact.SurName, contact.GivenName,
                            contact.Title,
                            contact.Street, contact.City, contact.State,
                            contact.Country, contact.PostalCode,
                            contact.Phone, contact.Fax ) );
                    }
                }
            }
        }
    }
```

Microsoft .NET Framework Data Provider for Microsoft SQL Server (SqlClient) と .NET データプロバイダとの間には、実装の詳細部分でいくつかの違いがあり、それらを理解しておく必要があります。

1. 新しいクラス SAConnectionFactory (IDbConnectionFactory を実装) が追加されています。次のように、Database.DefaultConnectionFactory に SAConnectionFactory のインスタンスを設定してから、データモデルを作成します。

   ```
   Database.DefaultConnectionFactory = new SAConnectionFactory();
   ```

2. Entity Framework の Code First では、原則として規則によるコーディングを行います。Entity Framework では、規則をコーディングすることによってデータモデルが推測されます。また、自動的な処理が多数行われます。場合によっては、開発者が Entity Framework のすべての規則を把握できないこともあります。しかし、SQL Anywhere のようなデータベース管理システムで意味を持たないコード規則もあります。Microsoft SQL Server とこれらのデータベースサーバ間には、いくつかの違いがあります。
   - Microsoft SQL Server では、単一のサインオンで複数のデータベースにアクセスできます。SQL Anywhere では、同時に 1 つのデータベースにのみ接続できます。
   - ユーザがパラメータのないコンストラクタを使用してユーザ定義の DbContext を作成した場合、SqlClient では統合セキュリティを使用してローカルコンピュータの Microsoft SQL Server Express に接続します。ユーザがすでにログインマッピングを作成している場合、.NET プロバイダは統合ログインを使用してデフォルトのサーバに接続します。
   - Entity Framework から DbDeleteDatabase または DbCreateDatabase を呼び出したときに、SqlClient は既存のデータベースを削除して新しいデータベースを作成します (Microsoft SQL Server Express Edition のみ)。.NET プロバイダはデータベースの削除または作成を決して行わず、データベースオブジェクト (テーブル、関係、制約など) を作成または削除します。ユーザは最初にデータベースを作成する必要があります。
   - IDbConnectionFactory.CreateConnection メソッドでは、文字列パラメータ "nameOrConnectionString" がデータベース名 (Microsoft SQL Server では初期カタログ) または接続文字列として扱われます。ユーザが DbContext の接続文字列を指定しない場合、SqlClient は、ユーザ定義の DbContext クラスのネームスペースを初期カタログとして使用し、ローカルコンピュータの SQL Express サーバに自動的に接続します。*SQL Anywhere* では、このパラメータに接続文字列のみ含めることができます。データベース名は無視され、代わりに統合ログインが使用されます。

3. Microsoft SQL Server SqlClient API では、データ注釈属性 TimeStamp を持つカラムが、Microsoft SQL Server のデータ型 timestamp/rowversion にマッピングされます。Microsoft SQL Server の timestamp/rowversion については開発者の間で誤解されている面があります。Microsoft SQL Server の timestamp/rowversion データ型は、*SQL Anywhere* や他のほとんどの RDBMS とは異なります。
   - Microsoft SQL Server の timestamp/rowversion は binary(8) です。日付と時刻の組み合わせ値はサポートされていません。SQL Anywhere でサポートされている timestamp というデータ型は、Microsoft SQL Server の datetime データ型と同等です。
   - Microsoft SQL Server の timestamp/rowversion 値はユニークであることが保証されています。SQL Anywhere の timestamp 値はユニークではありません。
   - Microsoft SQL Server の timestamp/rowversion 値は、ローが更新されるたびに変更されます。

   TimeStamp データ注釈属性は .NET データプロバイダではサポートされていません。
4. Entity Framework 4.1 のデフォルトでは、スキーマ名または所有者名が常に dbo に設定されます。これは Microsoft SQL Server のデフォルトのスキーマです。ただし、dbo は SQL Anywhere データベースには適していません。SQL Anywhere では、データ注釈または Fluent API のどちらかを使用して、テーブル名を持つスキーマ名または所有者名 (GROUPO など) を指定する必要があります。次に例を示します。

```
namespace CodeFirstTest
{
    public class Customer
    {
        [Key()]
        public int ID { get; set; }
        public string SurName { get; set; }
        public string GivenName { get; set; }
        public string Street { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public string Country { get; set; }
        public string PostalCode { get; set; }
        public string Phone { get; set; }
        public string CompanyName { get; set; }
        public virtual ICollection<Contact> Contacts { get; set; }
    }
    public class Contact
    {
        [Key()]
        public int ID { get; set; }
        public string SurName { get; set; }
        public string GivenName { get; set; }
        public string Title { get; set; }
        public string Street { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public string Country { get; set; }
        public string PostalCode { get; set; }
        public string Phone { get; set; }
        public string Fax { get; set; }
        [ForeignKey( "Customer" )]
        public int CustomerID { get; set; }
        public virtual Customer Customer { get; set; }
    }
    [Table( "Departments", Schema = "GROUPO" )]
    public class Department
    {
        [Key()]
        public int DepartmentID { get; set; }
        public string DepartmentName { get; set; }
        public int DepartmentHeadID { get; set; }
    }
    public class Context : DbContext
```

```
    {
        public Context() : base() { }
        public Context( string connStr ) : base( connStr ) { }
        public DbSet<Contact> Contacts { get; set; }
        public DbSet<Customer> Customers { get; set; }
        public DbSet<Department> Departments { get; set; }
        protected override void OnModelCreating( DbModelBuilder modelBuilder )
        {
            modelBuilder.Entity<Contact>().ToTable( "Contacts", "GROUPO" );
            modelBuilder.Entity<Customer>().ToTable( "Customers", "GROUPO" );
        }
    }
}
```

このセクションの内容:

新しいデータベースに対する Code First [86 ページ]
Microsoft の Code First to a New Database チュートリアルを試す場合、いくつかの手順を異なる方法で実行する必要があります。

既存のデータベースに対する Code First [88 ページ]
Microsoft の Code First to an Existing Database チュートリアルを試す場合、いくつかの手順を異なる方法で実行する必要があります。

新しいデータベースに対する Model First [90 ページ]
Microsoft の Model First チュートリアルを試す場合、いくつかの手順を異なる方法で実行する必要があります。

既存のデータベースに対する Database First [92 ページ]
Microsoft の Database First to an Existing Database チュートリアルを試す場合、いくつかの手順を異なる方法で実行する必要があります。

1.3.1.9.1 新しいデータベースに対する Code First

Microsoft の Code First to a New Database チュートリアルを試す場合、いくつかの手順を異なる方法で実行する必要があります。

前提条件

コンピュータに Microsoft Visual Studio および .NET Framework がインストールされている必要があります。

CREATE ANY OBJECT システム権限が必要です。

コンテキスト

Microsoft の Code First to a New Database チュートリアルは、Microsoft の .NET データプロバイダ用に設計されています。次の手順は、SQL Anywhere .NET Data Provider を代わりに使用した場合のガイダンスです。チュートリアルを試す前に、この手順を見直してください。

手順

1. Entity Framework 6 の場合、このバージョンに対応した SQL Anywhere .NET Data Provider がインストールされていることを確認します。Microsoft Visual Studio のインスタンスが実行されていないことを確認します。管理者権限でコマンドプロンプトを開き、次の手順を実行します。

```
cd %SQLANY17%¥Assembly¥v4.5
SetupVSPackage.exe /i /v EF6
```

2. Microsoft Visual Studio プロジェクトを作成する前に、Interactive SQL でサンプルデータベースに接続し、以下の SQL スクリプトを実行して、別のチュートリアルで作成している場合は Blogs テーブル、Posts テーブル、Users テーブルを削除します。

```
DROP TABLE IF EXISTS [GROUPO].[Blogs];
DROP TABLE IF EXISTS [GROUPO].[Posts];
DROP TABLE IF EXISTS [GROUPO].[Users];
```

3. Microsoft Visual Studio を起動します。Microsoft Visual Studio 2013 および Entity Framework 6.1.3 を使用して正常に実行された手順を次に示します。
4. 手順のいずれかで、NuGet Package Manager を使用し、Entity Framework 6 の最新バージョンをプロジェクトにインストールする必要があります。
   この手順で、`App.config` ファイルが作成されます。ただし、このままでは SQL Anywhere での使用に適していません。
5. `App.config` の内容を次のように置き換えます。

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://
go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework"
type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
EntityFramework, Version=6.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" requirePermission="false" />
  </configSections>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>
  <system.data>
      <DbProviderFactories>
          <clear />
          <add name="SQL Anywhere 17 Data Provider"
invariant="Sap.Data.SQLAnywhere" description=".Net Framework Data Provider for
SQL Anywhere 17" type="Sap.Data.SQLAnywhere.SAFactory, Sap.Data.SQLAnywhere.EF6,
Version=17.0.0.10094, Culture=neutral, PublicKeyToken=f222fc4333e0d400" />
      </DbProviderFactories>
  </system.data>
  <entityFramework>
    <defaultConnectionFactory type="Sap.Data.SQLAnywhere.SAConnectionFactory,
Sap.Data.SQLAnywhere.EF6, Version=17.0.0.10094, Culture=neutral,
PublicKeyToken=f222fc4333e0d400">
        <parameters>
            <parameter value="DSN=SQL Anywhere 17 Demo;PWD=sql" />
        </parameters>
    </defaultConnectionFactory>
    <providers>
        <provider invariantName="Sap.Data.SQLAnywhere"
type="Sap.Data.SQLAnywhere.SAProviderServices, Sap.Data.SQLAnywhere.EF6,
Version=17.0.0.10094, Culture=neutral, PublicKeyToken=f222fc4333e0d400" />
    </providers>
```

```
  </entityFramework>
</configuration>
```

6. App.config に記載されているデータプロバイダバージョン番号をすべて更新します。バージョン番号は、現在インストールしているデータプロバイダのバージョンに一致する必要があります。
7. App.config を更新したら、プロジェクトを作成する必要があります。
8. チュートリアルの残りの手順を実行します。

結果

これで、テーブルがデータベースに存在していない場合に Code First 方式を使用する Entity Framework アプリケーションが完成しました。

1.3.1.9.2 既存のデータベースに対する Code First

Microsoft の Code First to an Existing Database チュートリアルを試す場合、いくつかの手順を異なる方法で実行する必要があります。

前提条件

コンピュータに Microsoft Visual Studio および .NET Framework がインストールされている必要があります。

CREATE ANY OBJECT システム権限が必要です。

コンテキスト

Microsoft の Code First to an Existing Database チュートリアルは、Microsoft の .NET データプロバイダ用に設計されています。次の手順は、SQL Anywhere .NET Data Provider を代わりに使用した場合のガイダンスです。チュートリアルを試す前に、この手順を見直してください。

手順

1. Entity Framework 6 の場合、このバージョンに対応した SQL Anywhere .NET Data Provider がインストールされていることを確認します。Microsoft Visual Studio のインスタンスが実行されていないことを確認します。管理者権限でコマンドプロンプトを開き、次の手順を実行します。

```
cd %SQLANY17%¥Assembly¥v4.5
SetupVSPackage.exe /i /v EF6
```

2. Microsoft Visual Studio プロジェクトを作成する前に、Interactive SQL でサンプルデータベースに接続し、以下の SQL スクリプトを実行して Blogs テーブル、Posts テーブル、Users テーブルを設定します。これは Microsoft のチュートリアルに記載されている内容と似ていますが、ここでは *dbo* ではなく、GROUPO スキーマ所有者を使用します。

```
CREATE TABLE [GROUPO].[Blogs] (
    [BlogId] INT DEFAULT AUTOINCREMENT NOT NULL,
    [Name] NVARCHAR (200) NULL,
    [Url]  NVARCHAR (200) NULL,
    CONSTRAINT [PK_GROUPO.Blogs] PRIMARY KEY CLUSTERED ([BlogId] ASC)
);

CREATE TABLE [GROUPO].[Posts] (
    [PostId] INT DEFAULT AUTOINCREMENT NOT NULL,
    [Title] NVARCHAR (200) NULL,
    [Content] NTEXT NULL,
    [BlogId] INT NOT NULL,
    CONSTRAINT [PK_GROUPO.Posts] PRIMARY KEY CLUSTERED ([PostId] ASC),
    CONSTRAINT [FK_GROUPO.Posts_GROUPO.Blogs_BlogId] FOREIGN KEY ([BlogId])
        REFERENCES [GROUPO].[Blogs] ([BlogId]) ON DELETE CASCADE
);
INSERT INTO [GROUPO].[Blogs] ([Name],[Url])
VALUES ('SAP SQL Anywhere', 'http://scn.sap.com/community/sql-anywhere')

INSERT INTO [GROUPO].[Blogs] ([Name],[Url])
VALUES ('SAP Business Trends', 'http://scn.sap.com/community/business-trends')
CREATE TABLE [GROUPO].[Users]
(
    [Username] NVARCHAR(50) NOT NULL PRIMARY KEY,
    [DisplayName] LONG NVARCHAR NULL
)
```

3. Microsoft Visual Studio を起動します。Microsoft Visual Studio 2013 および Entity Framework 6.1.3 を使用して正常に実行された手順を次に示します。
4. Microsoft Visual Studio *Server Explorer* で、*.NET Framework Data Provider for SQL Anywhere17* を使用してデータ接続を作成します。*ODBC* データソースに SQL Anywhere 17 Demo を使用します。ユーザ *ID* フィールドとパスワードフィールドに値を入力します。テスト接続ボタンを使用し、データベースに接続できることを確認します。
5. 新しいプロジェクトを作成したら、NuGet Package Manager を使用し、Entity Framework 6 の最新バージョンをプロジェクトにすぐにインストールします。
   この手順で、`App.config` ファイルが作成されます。ただし、このままでは SQL Anywhere での使用に適していません。
6. `App.config` の内容を次のように置き換えます。

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://
go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework"
type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
EntityFramework, Version=6.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" requirePermission="false" />
  </configSections>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>
  <system.data>
      <DbProviderFactories>
          <clear />
          <add name="SQL Anywhere 17 Data Provider"
invariant="Sap.Data.SQLAnywhere" description=".Net Framework Data Provider for
```

```
SQL Anywhere 17" type="Sap.Data.SQLAnywhere.SAFactory, Sap.Data.SQLAnywhere.EF6,
Version=17.0.0.10094, Culture=neutral, PublicKeyToken=f222fc4333e0d400" />
      </DbProviderFactories>
  </system.data>
  <entityFramework>
    <defaultConnectionFactory type="Sap.Data.SQLAnywhere.SAConnectionFactory,
Sap.Data.SQLAnywhere.EF6, Version=17.0.0.10094, Culture=neutral,
PublicKeyToken=f222fc4333e0d400">
    </defaultConnectionFactory>
    <providers>
        <provider invariantName="Sap.Data.SQLAnywhere"
type="Sap.Data.SQLAnywhere.SAProviderServices, Sap.Data.SQLAnywhere.EF6,
Version=17.0.0.10094, Culture=neutral, PublicKeyToken=f222fc4333e0d400" />
    </providers>
  </entityFramework>
</configuration>
```

7. `App.config` に記載されているデータプロバイダバージョン番号をすべて更新します。バージョン番号は、現在インストールしているデータプロバイダのバージョンに一致する必要があります。
8. `App.config` を更新したら、プロジェクトを作成する必要があります。これは、Reverse Engineer Model の手順の前に実行する必要があります。
9. *Entity Data Model Wizard* で、接続文字列に機密データを含めるオプションを選択します。
10. すべてのテーブルを選択するのではなく、テーブルで GROUPO を展開し、Blogs テーブルと Posts テーブルを選択します。
11. チュートリアルの残りの手順を実行します。

結果

これで、データベースにすでに存在しているテーブルに対して Code First 方式を使用する Entity Framework アプリケーションが完成しました。

1.3.1.9.3 新しいデータベースに対する Model First

Microsoft の Model First チュートリアルを試す場合、いくつかの手順を異なる方法で実行する必要があります。

前提条件

コンピュータに Microsoft Visual Studio および .NET Framework がインストールされている必要があります。

CREATE ANY OBJECT システム権限が必要です。

コンテキスト

Microsoft の Model First チュートリアルは、Microsoft の .NET データプロバイダ用に設計されています。次の手順は、SQL Anywhere .NET Data Provider を代わりに使用した場合のガイダンスです。チュートリアルを試す前に、この手順を見直してください。

手順

1. Entity Framework 6 の場合、このバージョンに対応した SQL Anywhere .NET Data Provider がインストールされていることを確認します。Microsoft Visual Studio のインスタンスが実行されていないことを確認します。管理者権限でコマンドプロンプトを開き、次の手順を実行します。

   ```
   cd %SQLANY17%¥Assembly¥v4.5
   SetupVSPackage.exe /i /v EF6
   ```

2. Microsoft Visual Studio プロジェクトを作成する前に、Interactive SQL でサンプルデータベースに接続し、以下の SQL スクリプトを実行して、別のチュートリアルで作成している場合は Blogs テーブル、Posts テーブル、Users テーブルを削除します。

   ```
   DROP TABLE IF EXISTS [GROUPO].[Blogs];
   DROP TABLE IF EXISTS [GROUPO].[Posts];
   DROP TABLE IF EXISTS [GROUPO].[Users];
   ```

3. Microsoft Visual Studio を起動します。
4. Microsoft Visual Studio *Server Explorer* で、*.NET Framework Data Provider for SQL Anywhere17* を使用してデータ接続を作成します。*ODBC* データソースに SQL Anywhere 17 Demo を使用します。ユーザ *ID* フィールドとパスワードフィールドに値を入力します。テスト接続ボタンを使用し、データベースに接続できることを確認します。
5. 新しいプロジェクトを作成したら、NuGet Package Manager を使用し、Entity Framework 6 の最新バージョンをプロジェクトにすぐにインストールします。
   この手順で、`App.config` ファイルが作成されます。ただし、このままでは SQL Anywhere での使用に適していません。
6. `App.config` の内容を次のように置き換えます。

   ```
   <?xml version="1.0" encoding="utf-8"?>
   <configuration>
     <configSections>
       <!-- For more information on Entity Framework configuration, visit http://
   go.microsoft.com/fwlink/?LinkID=237468 -->
       <section name="entityFramework"
   type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
   EntityFramework, Version=6.0.0.0, Culture=neutral,
   PublicKeyToken=b77a5c561934e089" requirePermission="false" />
     </configSections>
     <startup>
       <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
     </startup>
     <system.data>
         <DbProviderFactories>
             <clear />
             <add name="SQL Anywhere 17 Data Provider"
   invariant="Sap.Data.SQLAnywhere" description=".Net Framework Data Provider for
   SQL Anywhere 17" type="Sap.Data.SQLAnywhere.SAFactory, Sap.Data.SQLAnywhere.EF6,
   Version=17.0.0.10094, Culture=neutral, PublicKeyToken=f222fc4333e0d400" />
   ```

```
      </DbProviderFactories>
  </system.data>
  <entityFramework>
    <defaultConnectionFactory type="Sap.Data.SQLAnywhere.SAConnectionFactory,
Sap.Data.SQLAnywhere.EF6, Version=17.0.0.10094, Culture=neutral,
PublicKeyToken=f222fc4333e0d400">
        <parameters>
            <parameter value="DSN=SQL Anywhere 17 Demo;PWD=sql" />
        </parameters>
    </defaultConnectionFactory>
    <providers>
        <provider invariantName="Sap.Data.SQLAnywhere"
type="Sap.Data.SQLAnywhere.SAProviderServices, Sap.Data.SQLAnywhere.EF6,
Version=17.0.0.10094, Culture=neutral, PublicKeyToken=f222fc4333e0d400" />
    </providers>
  </entityFramework>
</configuration>
```

7. `App.config` に記載されているデータプロバイダバージョン番号をすべて更新します。バージョン番号は、現在インストールしているデータプロバイダのバージョンに一致する必要があります。
8. `App.config` を更新したら、プロジェクトを作成する必要があります。これは、Entity Framework Designer を使用する前に実行する必要があります。
9. データベースを生成する手順の前に、`BloggingModel.edmx [Diagram1]` (設計フォーム) のプロパティを開き、データベーススキーマ名を GROUPO に設定し、*DDL 生成テンプレート*を `SSDLToSA17.tt (VS)` に設定します。
10. *Generate Database Wizard* で、接続文字列に機密データを含めるオプションを選択します。
11. チュートリアルの残りの手順を実行します。

結果

これで、テーブルがデータベースに存在していない場合に Model First 方式を使用する Entity Framework アプリケーションが完成しました。

1.3.1.9.4 既存のデータベースに対する Database First

Microsoft の Database First to an Existing Database チュートリアルを試す場合、いくつかの手順を異なる方法で実行する必要があります。

前提条件

コンピュータに Microsoft Visual Studio および .NET Framework がインストールされている必要があります。

CREATE ANY OBJECT システム権限が必要です。

コンテキスト

Microsoft の Database First to an Existing Database チュートリアルは、Microsoft の .NET データプロバイダ用に設計されています。次の手順は、SQL Anywhere .NET Data Provider を代わりに使用した場合のガイダンスです。チュートリアルを試す前に、この手順を見直してください。

手順

1. Entity Framework 6 の場合、このバージョンに対応した SQL Anywhere .NET Data Provider がインストールされていることを確認します。Microsoft Visual Studio のインスタンスが実行されていないことを確認します。管理者権限でコマンドプロンプトを開き、次の手順を実行します。

   ```
   cd %SQLANY17%¥Assembly¥v4.5
   SetupVSPackage.exe /i /v EF6
   ```

2. Microsoft Visual Studio プロジェクトを作成する前に、Interactive SQL でサンプルデータベースに接続し、以下の SQL スクリプトを実行して Blogs テーブル、Posts テーブル、Users テーブルを設定します。これは Microsoft のチュートリアルに記載されている内容と似ていますが、ここでは *dbo* ではなく、GROUPO スキーマ所有者を使用します。

   ```
   CREATE TABLE [GROUPO].[Blogs] (
       [BlogId] INT DEFAULT AUTOINCREMENT NOT NULL,
       [Name] NVARCHAR (200) NULL,
       [Url]  NVARCHAR (200) NULL,
       CONSTRAINT [PK_GROUPO.Blogs] PRIMARY KEY CLUSTERED ([BlogId] ASC)
   );

   CREATE TABLE [GROUPO].[Posts] (
       [PostId] INT DEFAULT AUTOINCREMENT NOT NULL,
       [Title] NVARCHAR (200) NULL,
       [Content] NTEXT NULL,
       [BlogId] INT NOT NULL,
       CONSTRAINT [PK_GROUPO.Posts] PRIMARY KEY CLUSTERED ([PostId] ASC),
       CONSTRAINT [FK_GROUPO.Posts_GROUPO.Blogs_BlogId] FOREIGN KEY ([BlogId])
           REFERENCES [GROUPO].[Blogs] ([BlogId]) ON DELETE CASCADE
   );
   INSERT INTO [GROUPO].[Blogs] ([Name],[Url])
   VALUES ('SAP SQL Anywhere', 'http://scn.sap.com/community/sql-anywhere')

   INSERT INTO [GROUPO].[Blogs] ([Name],[Url])
   VALUES ('SAP Business Trends', 'http://scn.sap.com/community/business-trends')
   CREATE TABLE [GROUPO].[Users]
   (
       [Username] NVARCHAR(50) NOT NULL PRIMARY KEY,
       [DisplayName] LONG NVARCHAR NULL
   )
   ```

3. Microsoft Visual Studio を起動します。Microsoft Visual Studio 2013 および Entity Framework 6.1.3 を使用して正常に実行された手順を次に示します。
4. Microsoft Visual Studio *Server Explorer* で、*.NET Framework Data Provider for SQL Anywhere17* を使用してデータ接続を作成します。*ODBC データソース*に SQL Anywhere 17 Demo を使用します。**ユーザ** *ID* フィールドと**パスワード**フィールドに値を入力します。**テスト接続**ボタンを使用し、データベースに接続できることを確認します。
5. 新しいプロジェクトを作成したら、NuGet Package Manager を使用し、Entity Framework 6 の最新バージョンをプロジェクトにすぐにインストールします。
   この手順で、`App.config` ファイルが作成されます。ただし、このままでは SQL Anywhere での使用に適していません。

6. App.config の内容を次のように置き換えます。

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://
go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework"
type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
EntityFramework, Version=6.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" requirePermission="false" />
  </configSections>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>
  <system.data>
      <DbProviderFactories>
          <clear />
          <add name="SQL Anywhere 17 Data Provider"
invariant="Sap.Data.SQLAnywhere" description=".Net Framework Data Provider for
SQL Anywhere 17" type="Sap.Data.SQLAnywhere.SAFactory, Sap.Data.SQLAnywhere.EF6,
Version=17.0.0.10094, Culture=neutral, PublicKeyToken=f222fc4333e0d400" />
      </DbProviderFactories>
  </system.data>
  <entityFramework>
    <defaultConnectionFactory type="Sap.Data.SQLAnywhere.SAConnectionFactory,
Sap.Data.SQLAnywhere.EF6, Version=17.0.0.10094, Culture=neutral,
PublicKeyToken=f222fc4333e0d400">
    </defaultConnectionFactory>
    <providers>
        <provider invariantName="Sap.Data.SQLAnywhere"
type="Sap.Data.SQLAnywhere.SAProviderServices, Sap.Data.SQLAnywhere.EF6,
Version=17.0.0.10094, Culture=neutral, PublicKeyToken=f222fc4333e0d400" />
    </providers>
  </entityFramework>
</configuration>
```

7. App.config に記載されているデータプロバイダバージョン番号をすべて更新します。バージョン番号は、現在インストールしているデータプロバイダのバージョンに一致する必要があります。
8. App.config を更新したら、プロジェクトを作成する必要があります。これは、Reverse Engineer Model の手順の前に実行する必要があります。
9. *Entity Data Model Wizard* で、接続文字列に機密データを含めるオプションを選択します。
10. すべてのテーブルを選択するのではなく、テーブルで GROUPO を展開し、Blogs テーブルと Posts テーブルを選択します。
11. チュートリアルの残りの手順を実行します。

結果

これで、データベースにすでに存在しているテーブルに対して Database First 方式を使用する Entity Framework アプリケーションが完成しました。

1.3.1.10 .NET でのトレースのサポート

.NET データプロバイダでは、.NET のトレーシング機能を使用したトレースをサポートしています。

デフォルトでは、トレースは無効です。トレースを有効にするには、アプリケーションの設定ファイルでトレースソースを指定します。次は、設定ファイルの例です。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.diagnostics>
<sources>
 <source name="Sap.Data.SQLAnywhere"
         switchName="SASourceSwitch"
         switchType="System.Diagnostics.SourceSwitch">
  <listeners>
   <add name="ConsoleListener"
        type="System.Diagnostics.ConsoleTraceListener"/>
   <add name="EventListener"
        type="System.Diagnostics.EventLogTraceListener"
        initializeData="MyEventLog"/>
   <add name="TraceLogListener"
        type="System.Diagnostics.TextWriterTraceListener"
        initializeData="myTrace.log"
        traceOutputOptions="ProcessId, ThreadId, Timestamp"/>
   <remove name="Default"/>
  </listeners>
 </source>
</sources>
<switches>
 <add name="SASourceSwitch" value="All"/>
 <add name="SATraceAllSwitch" value="1" />
 <add name="SATraceExceptionSwitch" value="1" />
 <add name="SATraceFunctionSwitch" value="1" />
 <add name="SATracePoolingSwitch" value="1" />
 <add name="SATracePropertySwitch" value="1" />
</switches>
</system.diagnostics>
</configuration>
```

上記の設定ファイルでは、4 種類のトレースリスナが参照されています。

ConsoleTraceListener

トレース出力またはデバッグ出力は、標準出力か標準エラーストリームのどちらかに送られます。Microsoft Visual Studio を使用する場合、出力は出力ウィンドウに表示されます。

DefaultTraceListener

このリスナは、名前 "Default" を使用して、自動的に Debug.Listeners および Trace.Listeners コレクションに追加されます。トレース出力またはデバッグ出力は、標準出力か標準エラーストリームのどちらかに送られます。Microsoft Visual Studio を使用する場合、出力は出力ウィンドウに表示されます。ConsoleTraceListener により生成される出力の重複を避けるため、このリスナは削除されます。

EventLogTraceListener

トレース出力またはデバッグ出力は、*initializeData* オプションで識別される EventLog に送られます。この例では、イベントログの名前は MyEventLog です。システムイベントログに書き込むには管理者権限が必要であり、この方法でアプリケーションをデバッグすることはおすすめしません。

TextWriterTraceListener

トレース出力またはデバッグ出力は TextWriter に送られ、*initializeData* オプションで識別されるファイルにストリームが書き込まれます。

上記のトレースリスナへのトレース出力を無効にするには、*<listeners>* にある対応する *add* エントリを削除します。

トレースの設定情報は、`App.config` というファイル名で、アプリケーションのプロジェクトフォルダに置かれます。このファイルが存在しない場合は、Microsoft Visual Studio を使用して作成し、プロジェクトに追加できます。それには、▶ 追加 ▶ 新しい項目 ▶を選択し、アプリケーション構成ファイルを選択します。

どのリスナでも traceOutputOptions を指定できます。次のオプションがあります。

Callstack

コールスタックを書き込みます。コールスタックは、Environment.StackTrace プロパティの戻り値で表されます。

DateTime

日付と時刻を書き込みます。

LogicalOperationStack

論理演算スタックを書き込みます。論理演算スタックは、CorrelationManager.LogicalOperationStack プロパティの戻り値で表されます。

None

要素を書き込みません。

ProcessId

プロセス ID を書き込みます。プロセス ID は、Process.Id プロパティの戻り値で表されます。

ThreadId

スレッド ID を書き込みます。スレッド ID は、現在のスレッドの Thread.ManagedThreadId プロパティの戻り値で表されます。

Timestamp

タイムスタンプを書き込みます。タイムスタンプは、System.Diagnostics.Stopwatch.GetTimeStamp メソッドの戻り値で表されます。

前述のサンプルの設定ファイルでは、TextWriterTraceListener に対してのみトレース出力オプションを指定しています。

特定のトレースオプションを設定することで、トレース対象を限定できます。デフォルトでは、数値のトレースオプション設定はすべて 0 です。設定できるトレースオプションには次の項目などがあります。

SASourceSwitch

SASourceSwitch は、次の値のいずれかを取ることができます。Off の場合、トレーシングはありません。

Off

イベントを許可しません。

Critical

重大なイベントのみを許可します。

Error

重大なイベントとエラーイベントを許可します。

Warning

重大なイベント、エラーイベント、警告イベントを許可します。

Information

重大なイベント、エラーイベント、警告イベント、情報イベントを許可します。

Verbose

重大なイベント、エラーイベント、警告イベント、情報イベント、冗長イベントを許可します。

ActivityTracing

停止、開始、中断、転送、再開イベントを許可します。

All

すべてのイベントを許可します。

次に設定例を示します。

```
<add name="SASourceSwitch" value="Error"/>
```

SATraceAllSwitch

すべてのトレースオプションが有効になります。すべてのオプションが選択されるため、その他のオプションを設定する必要はありません。このオプションを選択した場合は、個々のオプションを無効にできません。たとえば、次のようにしても、例外トレースは無効になりません。

```
<add name="SATraceAllSwitch" value="1" />
<add name="SATraceExceptionSwitch" value="0" />
```

SATraceExceptionSwitch

すべての例外が記録されます。トレースメッセージの形式は次のとおりです。

```
<Type|ERR> message='message_text'[ nativeError=error_number]
```

nativeError=error_number は、SAException オブジェクトが存在する場合に限り表示されます。

SATraceFunctionSwitch

すべてのメソッドスコープの開始と終了が記録されます。トレースメッセージの形式は次のいずれかです。

```
enter_nnn <sa.class_name.method_name|API> [object_id#][parameter_names]
leave_nnn
```

nnn は、スコープのネストレベル 1、2、3 などを表す整数です。省略可能な parameter_names は、スペースで区切られたパラメータ名のリストです。

SATracePoolingSwitch

すべての接続プーリングが記録されます。トレースメッセージの形式は次のいずれかです。

```
<sa.ConnectionPool.AllocateConnection|CPOOL> connectionString='connection_text'
<sa.ConnectionPool.RemoveConnection|CPOOL> connectionString='connection_text'
<sa.ConnectionPool.ReturnConnection|CPOOL> connectionString='connection_text'
<sa.ConnectionPool.ReuseConnection|CPOOL> connectionString='connection_text'
```

SATracePropertySwitch

すべてのプロパティの設定と取得が記録されます。トレースメッセージの形式は次のいずれかです。

```
<sa.class_name.get_property_name|API> object_id#
<sa.class_name.set_property_name|API> object_id#
```

このセクションの内容:

.NET アプリケーションのトレース設定 [98 ページ]

TableViewer サンプルアプリケーションでトレースを有効にするには、作成する設定ファイルの中で、ConsoleTraceListener および TextWriterTraceListener リスナを参照し、デフォルトのリスナを削除し、本来は 0 に設定されるスイッチをすべて有効にします。

関連情報

.NET データアクセスアーキテクチャガイド
データアクセスのトレース

1.3.1.10.1 .NET アプリケーションのトレース設定

TableViewer サンプルアプリケーションでトレースを有効にするには、作成する設定ファイルの中で、ConsoleTraceListener および TextWriterTraceListener リスナを参照し、デフォルトのリスナを削除し、本来は 0 に設定されるスイッチをすべて有効にします。

前提条件

Microsoft Visual Studio がインストールされている必要があります。

手順

1. Microsoft Visual Studio で TableViewer サンプルを開きます。

   Microsoft Visual Studio を起動し、*%SQLANYSAMP17%*¥SQLAnywhere¥ADO.NET¥TableViewer¥TableViewer.sln を開きます。
2. App.config というアプリケーションファイルを作成します。

   追加 > 新しい項目 を選択し、アプリケーション構成ファイルを選択します。このファイルに次の構成情報を入力します。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.diagnostics>
<sources>
 <source name="Sap.Data.SQLAnywhere"
         switchName="SASourceSwitch"
         switchType="System.Diagnostics.SourceSwitch">
  <listeners>
   <add name="ConsoleListener"
        type="System.Diagnostics.ConsoleTraceListener"/>
   <add name="TraceLogListener"
        type="System.Diagnostics.TextWriterTraceListener"
        initializeData="myTrace.log"
```

```
        traceOutputOptions="ProcessId, ThreadId, Timestamp"/>
    <remove name="Default"/>
   </listeners>
  </source>
</sources>
<switches>
 <add name="SASourceSwitch" value="All"/>
 <add name="SATraceAllSwitch" value="1" />
 <add name="SATraceExceptionSwitch" value="1" />
 <add name="SATraceFunctionSwitch" value="1" />
 <add name="SATracePoolingSwitch" value="1" />
 <add name="SATracePropertySwitch" value="1" />
</switches>
</system.diagnostics>
</configuration>
```

3. アプリケーションを再構築します。
4. デバッグ > デバッグの開始 をクリックします。

結果

アプリケーションの実行が完了すると、トレース出力が `bin¥Debug¥myTrace.log` ファイルに記録されます。

次のステップ

Microsoft Visual Studio の出力ウィンドウでトレースログを表示します。

関連情報

http://msdn.microsoft.com/en-us/library/ms971550.aspx

1.3.1.11 SQL Anywhere .NET データプロバイダのアンマネージコード

.NET データプロバイダが .NET アプリケーションによって最初にロードされると (通常、SAConnection を使用したデータベース接続の作成時)、プロバイダのアンマネージコードを含む DLL がアンパックされます。

プロバイダによって、次の方式を使用して識別されたディレクトリのサブディレクトリに `dbdata17.dll` ファイルが格納されます。

1. プロバイダがアンロードで最初に使用しようとするディレクトリは、次のものによって返されたうちの最初のディレクトリです。
   - *TMP* 環境変数によって識別されたパス。
   - *TEMP* 環境変数によって識別されたパス。

- *USERPROFILE* 環境変数によって識別されたパス。
- Windows ディレクトリ。

2. 識別されたディレクトリにアクセスできない場合、プロバイダは現在の作業ディレクトリの使用を試みます。
3. 現在の作業ディレクトリにアクセスできない場合、プロバイダはアプリケーション自体のロード元ディレクトリの使用を試みます。

サブディレクトリ名は GUID の形式をとり、サフィックスが付きます。このサフィックスには、バージョン番号、32 ビットではない (x64 などの) 場合のマシンアーキテクチャタグ、一意性の保証に使用されるインデックス番号が含まれます。次は、32 ビットアーキテクチャのサブディレクトリ名の例です。

```
{16AA8FB8-4A98-4757-B7A5-0FF22C0A6E33}_1700_1
```

次は、64 ビットアーキテクチャのサブディレクトリ名の例です。

```
{16AA8FB8-4A98-4757-B7A5-0FF22C0A6E33}_1700.x64_1
```

1.3.2 .NET データプロバイダチュートリアル

Simple サンプルプロジェクトと Table Viewer サンプルプロジェクトを使用して、.NET プロバイダを使用した .NET アプリケーションプログラミングについて説明します。付属のチュートリアルでは、Microsoft Visual Studio を使用して Simple Viewer .NET データベースアプリケーションを構築する手順をひととおり説明します。

これらのサンプルプロジェクトは、Microsoft Visual Studio 2005 以降のバージョンで使用できます。Microsoft Visual Studio 2008 以降のバージョンを使用している場合、Microsoft Visual Studio アップグレードウィザードを実行しなければならない場合があります。

このセクションの内容:

チュートリアル:SimpleWin32 の Simple コードサンプルの使用 [101 ページ]
Simple プロジェクトを例に、.NET データプロバイダを使用してデータベースから結果セットを取得する方法を示します。

チュートリアル:Table Viewer コードサンプルの使用 [105 ページ]
TableViewer プロジェクトを例に、.NET データプロバイダを使用してデータベースに接続し、SQL 文を実行し、DataGrid オブジェクトを使用して結果を表示します。

チュートリアル:Microsoft Visual Studio を使用したシンプルな .NET データベースアプリケーションの開発 [109 ページ]
このチュートリアルでは、Visual Studio を使用して Simple Viewer .NET データベースアプリケーションを構築する手順をひととおり説明します。

1.3.2.1 チュートリアル:SimpleWin32 の Simple コードサンプルの使用

Simple プロジェクトを例に、.NET データプロバイダを使用してデータベースから結果セットを取得する方法を示します。

前提条件

SELECT ANY TABLE システム権限が必要です。

コンテキスト

Simple プロジェクトはサンプルに付属しています。このプロジェクトでは、Employees テーブルからの名前を設定する簡単なリストボックスの例を示します。コンピュータに Microsoft Visual Studio および Microsoft .NET Framework がインストールされている必要があります。

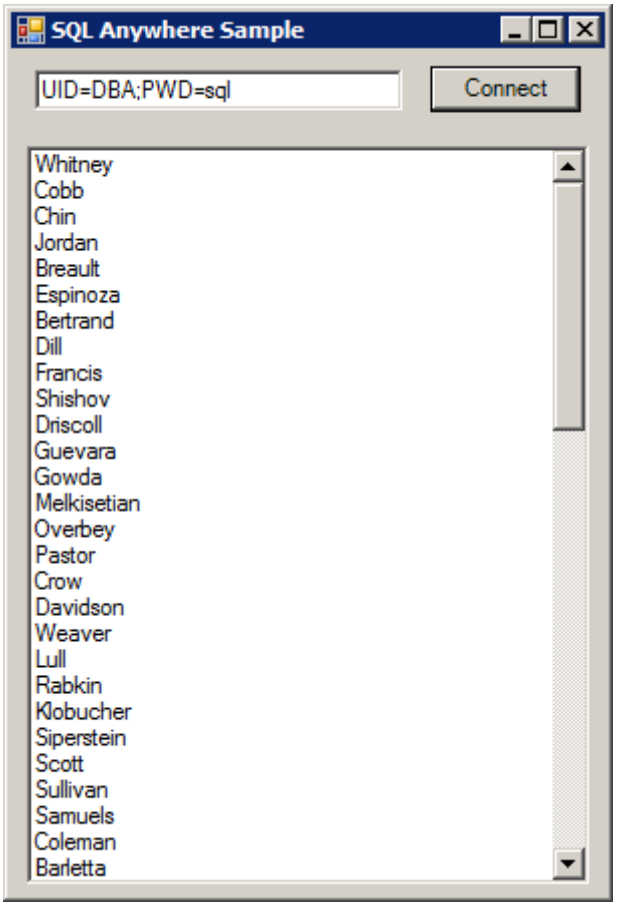
手順

1. Microsoft Visual Studio を起動します。
2. ファイル > 開く > プロジェクト をクリックします。
3. *%SQLANYSAMP17%*¥SQLAnywhere¥ADO.NET¥SimpleWin32 を参照し、Simple.sln プロジェクトを開きます。
4. コンピュータに Microsoft Visual Studio および Microsoft .NET Framework がインストールされている必要があります。プロジェクトで .NET データプロバイダを使用するには、データプロバイダへの参照を追加する必要があります。これはすでに Simple コードサンプルで行われています。データプロバイダ (Sap.Data.SQLAnywhere) への参照を表示するには、ソリューションエクスプローラウィンドウで参照設定フォルダを開きます。
5. また、データプロバイダクラスを参照する using ディレクティブもソースコードに追加してください。これはすでに Simple コードサンプルで行われています。using ディレクティブを表示するには、次の手順に従います。
   - プロジェクトのソースコードを開きます。ソリューションエクスプローラウィンドウで Form1.cs を右クリックし、コードの表示をクリックします。
     上部セクションの using ディレクティブに次の行が表示されます。

     ```
     using Sap.Data.SQLAnywhere;
     ```

     この行は Microsoft C# プロジェクトに必要です。Microsoft Visual Basic .NET を使用している場合、ソースコードに同等の Imports 行を追加する必要があります。
6. デバッグ > デバッグなしで開始 をクリックするか、Ctrl + F5 を押して、Simple サンプルを実行します。
7. *SQL Anywhere Sample* ウィンドウで、サンプルデータベースのユーザ ID とパスワード認証情報を変更し、接続をクリックします。

   アプリケーションがサンプルデータベースに接続され、次のように各従業員の姓がウィンドウに表示されます。



8. *SQL Anywhere Sample* ウィンドウを閉じ、アプリケーションを終了してサンプルデータベースとの接続を切断します。これによって、データベースサーバも停止します。

結果

これで、Microsoft .NET データプロバイダを使用してデータベースから結果セットを取得するシンプルな .NET アプリケーションを構築して実行しました。

例

完全なアプリケーションは、*%SQLANYSAMP17%*`¥SQLAnywhere¥ADO.NET¥SimpleWin32` のサンプルディレクトリにあります。

このセクションの内容:

Simple サンプルプロジェクトの説明 [103 ページ]
Simple プロジェクトのコードを検証することで、SQL Anywhere .NET データプロバイダのいくつかの主要機能について説明します。

関連情報

Microsoft Visual Studio プロジェクトでの SQL Anywhere .NET データプロバイダの使用 [55 ページ]

1.3.2.1.1 Simple サンプルプロジェクトの説明

Simple プロジェクトのコードを検証することで、SQL Anywhere .NET データプロバイダのいくつかの主要機能について説明します。

Simple プロジェクトでは、サンプルディレクトリにあるサンプルデータベース *demo.db* を使用します。

Simple プロジェクトについては、一度に数行を説明します。サンプルのすべてのコードが含まれているわけではありません。コード全体を確認するには、プロジェクトファイル *%SQLANYSAMP17%*`¥SQLAnywhere¥ADO.NET¥SimpleWin32¥Simple.sln` を開きます。

制御の宣言

次のコードは、txtConnectString という名前のテキストボックス、btnConnect という名前のボタン、listEmployees という名前のリストボックスを宣言します。

```
private System.Windows.Forms.TextBox txtConnectString;
private System.Windows.Forms.Button btnConnect;
private System.Windows.Forms.ListBox listEmployees;
```

データベースへの接続

btnConnect_Click メソッドは、SAConnection 接続オブジェクトを宣言して初期化します。

```
SAConnection conn = new SAConnection("Data Source=SQL Anywhere 17 Demo;"
    + txtConnectString.Text);
```

SAConnection オブジェクトは、Open メソッドが呼び出されると、接続文字列を使用してサンプルデータベースに接続します。

```
conn.Open();
```

クエリの定義

SQL 文は SACommand オブジェクトを使用して実行されます。次のコードは、SACommand コンストラクタを使用し、コマンドオブジェクトを宣言して作成します。このコンストラクタは、実行されるクエリを表す文字列と、クエリが実行される接続を表す SAConnection オブジェクトを受け入れます。

```
SACommand cmd = new SACommand("SELECT Surname FROM Employees", conn);
```

結果の表示

クエリの結果は、SADataReader オブジェクトを使用して取得されます。次のコードは、ExecuteReader コンストラクタを使用して SADataReader オブジェクトを宣言し、作成します。このコンストラクタは、SACommand オブジェクトである cmd のメンバーであり、事前に宣言されています。ExecuteReader はコマンドテキストを実行するために接続に送信し、SADataReader を構築します。

```
SADataReader reader = cmd.ExecuteReader();
```

次のコードは、SADataReader オブジェクトに格納されているローをループし、これらをリストボックスコントロールに追加します。Read メソッドが呼び出されるたびに、データリーダーは結果セットから別のローを取り返します。読み込まれるローごとに新しい項目がリストボックスに追加されます。データリーダーは、引数 0 で GetString メソッドを使用して、結果セットローの最初のカラムを取得します。

```
listEmployees.BeginUpdate();
while( reader.Read() )
{
    listEmployees.Items.Add(reader.GetString(0));
}
listEmployees.EndUpdate();
```

終了

メソッドの終わりにある次のコードは、データリーダーオブジェクトと接続オブジェクトを閉じます。

```
reader.Close();
conn.Close();
```

エラー処理

実行時に .NET データプロバイダオブジェクトから発生したエラーは、ウィンドウに表示されて処理されます。次のコードは、エラーを取得してそのメッセージを表示します。

```
catch (SAException ex)
{
    MessageBox.Show(ex.Errors[0].Message);
```

```
}
```

1.3.2.2 チュートリアル:Table Viewer コードサンプルの使用

TableViewer プロジェクトを例に、.NET データプロバイダを使用してデータベースに接続し、SQL 文を実行し、DataGrid オブジェクトを使用して結果を表示します。

前提条件

コンピュータに Microsoft Visual Studio および Microsoft .NET Framework がインストールされている必要があります。

SELECT ANY TABLE システム権限が必要です。

コンテキスト

TableViewer プロジェクトはサンプルに付属しています。Table Viewer プロジェクトは Simple プロジェクトよりも複雑です。これを使用して、データベースへの接続、テーブルの選択、データベースでの SQL 文の実行ができます。

手順

1. Microsoft Visual Studio を起動します。
2. ファイル > 開く > プロジェクト をクリックします。
3. *%SQLANYSAMP17%*¥SQLAnywhere¥ADO.NET¥TableViewer を参照し、TableViewer.sln プロジェクトを開きます。
4. プロジェクトで .NET データプロバイダを使用するには、データプロバイダ DLL への参照を追加する必要があります。これはすでに Table Viewer コードサンプルで行われています。データプロバイダ (Sap.Data.SQLAnywhere) への参照を表示するには、ソリューションエクスプローラウィンドウで参照設定フォルダを開きます。
5. また、データプロバイダクラスを参照する using ディレクティブもソースコードに追加してください。これはすでに Table Viewer コードサンプルで行われています。using ディレクティブを表示するには、次の手順に従います。
   - プロジェクトのソースコードを開きます。ソリューションエクスプローラウィンドウで TableViewer.cs を右クリックし、コードの表示をクリックします。
   - 上部セクションの using ディレクティブに次の行が表示されます。

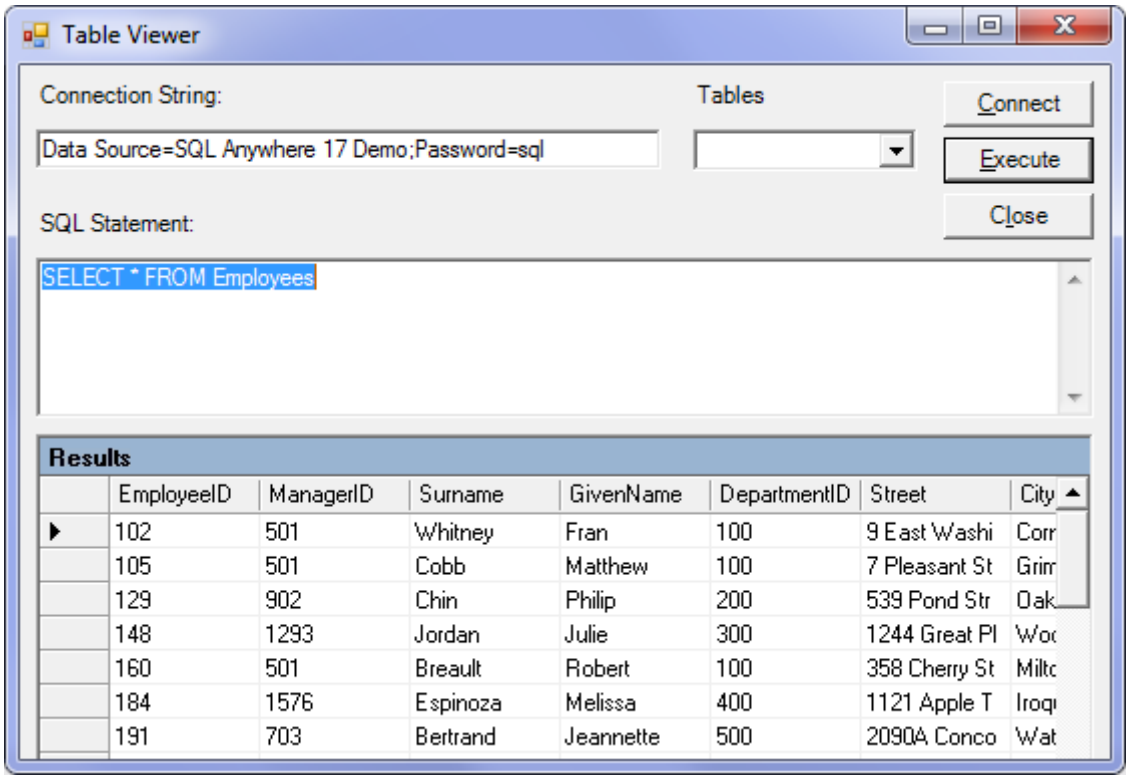
     ```
     using Sap.Data.SQLAnywhere;
     ```

     この行は Microsoft C# プロジェクトに必要です。Microsoft Visual Basic を使用している場合、ソースコードに同等の Imports 行を追加する必要があります。

6. デバッグ ▶ デバッグなしで開始 をクリックするか、Ctrl + F5 を押して、Table Viewer サンプルを実行します。

   アプリケーションがサンプルデータベースに接続します。
7. *Table Viewer* ウィンドウで、*Connection String* テキストボックスを変更し、DBA ユーザの有効なデータベースパスワードを入力します。接続をクリックします。
8. *Table Viewer* ウィンドウで *Execute* をクリックします。

   アプリケーションは、サンプルデータベースの Employees テーブルからデータを取り出し、次のようにクエリ結果を *Results* データグリッドに表示します。



   このアプリケーションから別の SQL 文を実行することもできます。これを行うには、*SQL 文*ウィンドウ枠に SQL 文を入力し、実行をクリックします。
9. *Table Viewer* ウィンドウを閉じ、アプリケーションを終了してサンプルデータベースとの接続を切断します。これによって、データベースサーバも停止します。

結果

これで、Microsoft .NET データプロバイダを使用してデータベースに接続し、SQL 文を実行し、DataGrid オブジェクトを使用して結果を表示することによって、.NET アプリケーションを構築して実行しました。

例

完全なアプリケーションは、*%SQLANYSAMP17%*`¥SQLAnywhere¥ADO.NET¥TableViewer` のサンプルディレクトリにあります。

このセクションの内容:

Table Viewer サンプルプロジェクトの説明 [107 ページ]
Table Viewer プロジェクトのコードを検証することで、SQL Anywhere .NET データプロバイダのいくつかの主要機能について説明します。

関連情報

Microsoft Visual Studio プロジェクトでの SQL Anywhere .NET データプロバイダの使用 [55 ページ]

1.3.2.2.1 Table Viewer サンプルプロジェクトの説明

Table Viewer プロジェクトのコードを検証することで、SQL Anywhere .NET データプロバイダのいくつかの主要機能について説明します。

Table Viewer プロジェクトでは、サンプルディレクトリにあるサンプルデータベース *demo.db* を使用します。

Table Viewer プロジェクトについては、一度に数行を説明します。サンプルのすべてのコードが含まれているわけではありません。コード全体を確認するには、プロジェクトファイル *%SQLANYSAMP17%*`¥SQLAnywhere¥ADO.NET¥TableViewer¥TableViewer.sln` を開きます。

制御の宣言

次のコードは、label1 および label2 という名前の Label、txtConnectString という名前の TextBox、btnConnect という名前のボタン、txtSQLStatement という名前の TextBox、btnExecute という名前のボタン、dgResults という名前の DataGrid を宣言します。

```
private System.Windows.Forms.Label label1;
private System.Windows.Forms.TextBox txtConnectString;
private System.Windows.Forms.Label label2;
private System.Windows.Forms.Button btnConnect;
private System.Windows.Forms.TextBox txtSQLStatement;
private System.Windows.Forms.Button btnExecute;
private System.Windows.Forms.DataGrid dgResults;
```

接続オブジェクトの宣言

SAConnection 型は、初期化されていない接続オブジェクトを宣言するために使用されます。SAConnection オブジェクトは、データソースへの固有接続を表すために使用されます。

```
private SAConnection _conn;
```

データベースへの接続

txtConnectString オブジェクトの Text プロパティのデフォルト値は "Data Source=SQL Anywhere 17 Demo;Password=sql" です。アプリケーションユーザは txtConnectString テキストボックスに新しい値を入力することで、この値を上書きできます。このデフォルト値がどのように設定されるかは、`TableViewer.cs` で Windows Form Designer Generated Code という記述のあるリージョンを開くことで確認できます。このリージョンで、次のコード行を探します。

```
this.txtConnectString.Text = "Data Source=SQL Anywhere 17 Demo;Password=passwd";
```

SAConnection オブジェクトは、この接続文字列を使用してデータベースに接続します。次のコードは、SAConnection コンストラクタを使用して接続文字列の設定された新しい接続オブジェクトを作成します。その後、Open メソッドを使用して接続を確立します。

```
_conn = new SAConnection( txtConnectString.Text );
_conn.Open();
```

クエリの定義

txtSQLStatement オブジェクトの Text プロパティのデフォルト値は "SELECT * FROM Employees" です。アプリケーションユーザは txtSQLStatement テキストボックスに新しい値を入力することで、この値を上書きできます。

SQL 文は SACommand オブジェクトを使用して実行されます。次のコードは、SACommand コンストラクタを使用し、コマンドオブジェクトを宣言して作成します。このコンストラクタは、実行されるクエリを表す文字列と、クエリが実行される接続を表す SAConnection オブジェクトを受け入れます。

```
SACommand cmd = new SACommand( txtSQLStatement.Text.Trim(), _conn );
```

結果の表示

クエリの結果は、SADataReader オブジェクトを使用して取得されます。次のコードは、ExecuteReader コンストラクタを使用して SADataReader オブジェクトを宣言し、作成します。このコンストラクタは、SACommand オブジェクトである cmd のメンバーであり、事前に宣言されています。ExecuteReader はコマンドテキストを実行するために接続に送信し、SADataReader を構築します。

```
SADataReader dr = cmd.ExecuteReader();
```

次のコードは、SADataReader オブジェクトを DataGrid オブジェクトに接続します。これにより結果カラムが画面に表示されるようになります。次に、SADataReader オブジェクトが閉じます。

```
dgResults.DataSource = dr;
dr.Close();
```

エラー処理

アプリケーションがデータベースに接続しようとしたときやテーブルコンボボックスにデータを移植するときにエラーが発生した場合、次のコードは、エラーを取得し、そのメッセージを表示します。

```
try
{
  _conn = new SAConnection( txtConnectString.Text );
  _conn.Open();
  SACommand cmd = new SACommand( "SELECT table_name FROM sys.systable " +
      "WHERE creator = 101 AND table_type != 'TEXT'", _conn );
  SADataReader   dr = cmd.ExecuteReader();
  comboBoxTables.Items.Clear();
  while ( dr.Read() )
  {
    comboBoxTables.Items.Add( dr.GetString( 0 ) );
  }
  dr.Close();
}
catch( SAException ex )
{
  MessageBox.Show( ex.Errors[0].Source + " : " + ex.Errors[0].Message + " (" +
                   ex.Errors[0].NativeError.ToString() + ")",
                   "Failed to connect" );
}
```

1.3.2.3 チュートリアル:Microsoft Visual Studio を使用したシンプルな .NET データベースアプリケーションの開発

このチュートリアルでは、Visual Studio を使用して Simple Viewer .NET データベースアプリケーションを構築する手順をひととおり説明します。

前提条件

SELECT ANY TABLE システム権限が必要です。新しい行を挿入するには、テーブルの所有者であるか、INSERT ANY TABLE 権限を持っているか、またはテーブルに対する INSERT 権限を持っていることが必要です。既存の行の内容を変更するには、更新されるテーブルの所有者であるか、変更されるカラムに対する UPDATE 権限を持っているか、または UPDATE ANY TABLE システム権限を持っていることが必要です。行を削除するには、テーブルの所有者であるか、テーブルに対する SELECT 権限および DELETE 権限を持っている必要があります。

1. レッスン 1:Table Viewer の作成 [110 ページ]
   Microsoft Visual Studio、サーバエクスプローラ、および SQL Anywhere .NET データプロバイダを使用して、サンプルデータベース内の 1 つのテーブルにアクセスするアプリケーションを作成します。このアプリケーションを使用して、テーブルのローを調べたり更新を実行することができます。
2. レッスン 2:同期データコントロールの追加 [114 ページ]
   前のレッスンで作成したフォームにデータグリッドコントロールを追加します。

1.3.2.3.1 レッスン 1:Table Viewer の作成

Microsoft Visual Studio、サーバエクスプローラ、および SQL Anywhere .NET データプロバイダを使用して、サンプルデータベース内の 1 つのテーブルにアクセスするアプリケーションを作成します。このアプリケーションを使用して、テーブルのローを調べたり更新を実行することができます。

前提条件

コンピュータに Microsoft Visual Studio および .NET Framework がインストールされている必要があります。

このチュートリアルの冒頭に記載されているロールと権限を持っている必要があります。

コンテキスト

このチュートリアルは、Microsoft Visual Studio と .NET Framework に基づきます。プロジェクト *%SQLANYSAMP17%* を開くと、完全なアプリケーションを確認できます。

手順

1. Microsoft Visual Studio を起動します。
2. ファイル > 新規 > プロジェクト をクリックします。

   新しいプロジェクトウィンドウが表示されます。
   a. 新しいプロジェクトウィンドウの左ウィンドウ枠で、プログラミング言語として *Visual Basic* または *Visual C#* をクリックします。
   b. *Windows* サブカテゴリで、*Windows* アプリケーション (VS 2005 の場合) または *Windows* フォームアプリケーション (VS 2008 以降の場合) をクリックします。
   c. インストールされている SQL Anywhere .NET データプロバイダに対応する .NET Framework バージョンを選択します。たとえば、インストールしたバージョンが 4.5 の場合、ドロップダウンリストから *.NET Framework 4.5* を選択します。
   d. プロジェクトの名前フィールドに、`MySimpleViewer` と入力します。
   e. *OK* をクリックし、新規プロジェクトを作成します。
3. 表示 > サーバエクスプローラ をクリックします。
4. サーバエクスプローラウィンドウで、データ接続、接続の追加をクリックします。
5. 接続の追加ウィンドウで次の作業を実行します。
   a. 接続の追加を以前に使用したことがある場合は、変更をクリックしてデータソースを変更します。
   b. データソースのリストが表示されます。表示されたデータソースのリストから *.NET Framework Data Provider for SQL Anywhere17* を選択し、*OK* をクリックします。
   c. データソースで、*ODBC* データソースをクリックし、***SQL Anywhere 17 Demo*** を選択するか、入力します。

> **注記**
>
> Microsoft Visual Studio の接続の追加ウィザードを 64 ビット Windows で使用する場合は、64 ビットのシステムデータソース名 (DSN) のみがユーザデータソース名と一緒に含まれます。32 ビットのシステムデータソース名は表示されません。Microsoft Visual Studio の 32 ビット設計環境では、[テスト接続] ボタンを使用すると、64 ビットのシステム DSN と等価な 32 ビットのシステム DSN を使用して接続の確立が試みられます。32 ビットのシステム DSN が存在しない場合、テストは失敗します。

d. ログイン情報のパスワードフィールドに、サンプルデータベースパスワード (*sql*) を入力します。
e. テスト接続をクリックして、サンプルデータベースに接続できることを確認します。
f. *OK* をクリックします。

SQL Anywhere.demo17 という新しい接続がサーバエクスプローラウィンドウに表示されます。

6. サーバエクスプローラウィンドウで、テーブル名が表示されるまで *SQL Anywhere.demo17* 接続を展開します。

   (Microsoft Visual Studio 2005 のみ) 次の手順を試してみます。

   a. Products テーブルを右クリックし、テーブルデータの表示をクリックします。

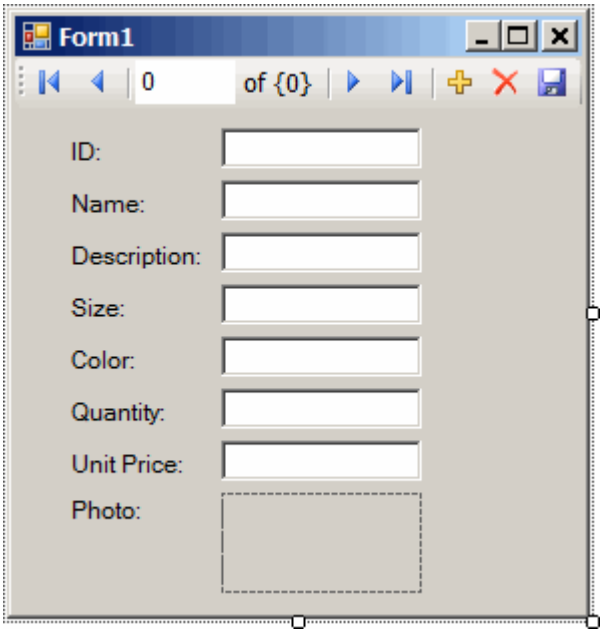
      Products テーブルのローとカラムがウィンドウに表示されます。
   b. テーブルデータのウィンドウを閉じます。

7. データ > 新しいデータソースの追加 をクリックします。Microsoft Visual Studio の新しいバージョンでは、これは プロジェクト > 新しいデータソースの追加 です。
8. データソース構成ウィザードで、次の作業を実行します。
   a. データソースの種類を選択ページでデータベースをクリックし、次へをクリックします。
   b. (Microsoft Visual Studio 2010 以降) データベースモデルページでデータセットをクリックし、次へをクリックします。
   c. データ接続ページで、*SQL Anywhere.demo17* をクリックし、はい、重要情報を接続文字列に含めますをクリックして、次へをクリックします。
   d. 接続文字列をアプリケーション構成ファイルに保存するページで、次の名前で接続を保存するが選択されていることを確認して次へをクリックします。
   e. データベースオブジェクトの選択ページでテーブルをクリックし、完了をクリックします。
9. データ > データソースの表示 をクリックします。Microsoft Visual Studio の新しいバージョンでは、これは 表示 > その他のウィンドウ > データソース です。

   データソースウィンドウが表示されます。

   データソースウィンドウで Products テーブルを展開します。

   a. [Products] をクリックし、ドロップダウンリストから詳細をクリックします。
   b. [Photo] をクリックし、ドロップダウンリストから *Picture Box* をクリックします。
   c. [Products] をクリックして、フォーム (Form1) にドラッグします。



データセットコントロールと複数のラベル付きテキストフィールドがフォームに表示されます。

10. フォームで、[Photo] の横にあるピクチャボックスをクリックします。
    a. ボックスの形を四角形に変更します。
    b. ピクチャボックスの右上の右矢印をクリックします。

       Picture Box タスクウィンドウが開きます。
    c. サイズモードドロップダウンリストから、ズームをクリックします。
    d. *Picture Box* タスクウィンドウを閉じるには、ウィンドウの外側で任意の場所をクリックします。
11. プロジェクトをビルドし、実行します。
    a. ビルド > ソリューションのビルド をクリックします。
    b. デバッグ > デバッグの開始 をクリックします。

       アプリケーションがサンプルデータベースに接続されて、Products テーブルの最初のローがテキストボックスとピクチャボックスに表示されます。

c. コントロールのボタンを使用して、結果セットのローをスクロールできます。
d. ロー番号をスクロールコントロールに入力すると、結果セットのローに直接アクセスできます。
e. テキストボックスを使用して結果セットの値を更新し、*Save Data* ボタンをクリックして値を保存できます。

12. アプリケーションを終了してプロジェクトを保存します。

結果

これで、Microsoft Visual Studio、サーバエクスプローラ、SQL Anywhere .NET データプロバイダを使用して、シンプルでありながら強力な Microsoft .NET アプリケーションが作成されました。

次のステップ

次のレッスンに進みます。次のレッスンでは、このレッスンで作成したフォームにデータグリッドコントロールを追加します。

タスクの概要: チュートリアル:Microsoft Visual Studio を使用したシンプルな .NET データベースアプリケーションの開発 [109 ページ]

次のタスク: レッスン 2:同期データコントロールの追加 [114 ページ]

1.3.2.3.2 レッスン 2:同期データコントロールの追加

前のレッスンで作成したフォームにデータグリッドコントロールを追加します。

前提条件

このチュートリアルのこれまでのレッスンを完了している必要があります。

このチュートリアルの冒頭に一覧されているロールと権限を持っている必要があります。

コンテキスト

このコントロールは、結果セット内のナビゲーションに合わせて内容を自動的に更新します。

プロジェクト *%SQLANYSAMP17%*`¥SQLAnywhere¥ADO.NET¥SimpleViewer¥SimpleViewer.sln` を開くと、完全なアプリケーションを確認できます。

手順

1. Microsoft Visual Studio を起動し、MySimpleViewer プロジェクトをロードします。
2. **データソース**ウィンドウで DataSet1 を右クリックし、**デザイナで *DataSet* を編集**をクリックします。
3. **データセットデザイナ**ウィンドウの空白領域を右クリックし、**追加** > *TableAdapter* をクリックします。
4. *TableAdapter* **構成ウィザード**で次の作業を実行します。
   a. **データ接続の選択**ページで、**次へ**をクリックします。
   b. **コマンドの種類を選択します**ページで、*SQL* **ステートメントを使用する**をクリックし、**次へ**をクリックします。
   c. *SQL* **ステートメントの入力**ページで、**クエリビルダ**をクリックします。
   d. **テーブルの追加**ウィンドウの**ビュー**タブをクリックし、*ViewSalesOrders* をクリックして**追加**をクリックします。
   e. **閉じる**をクリックして**テーブルの追加**ウィンドウを閉じます。
5. ウィンドウのすべてのセクションが表示されるように、**クエリビルダ**ウィンドウを拡大します。
   a. すべてのチェックボックスが表示されるように、*ViewSalesOrders* ウィンドウを拡大します。
   b. *Region* をクリックします。
   c. *Quantity* をクリックします。
   d. *ProductID* をクリックします。
   e. *ViewSalesOrders* ウィンドウの下のグリッドで、[ProductID] カラムの**出力**の下にあるチェックボックスをオフにします。
   f. [ProductID] カラムで、**フィルタ**セルに疑問符 (?) を入力し、フォームの他の場所をクリックします。これで ProductID の WHERE 句が生成されます。

次のような SQL クエリが作成されました。

```
SELECT   Region, Quantity
FROM     GROUPO.ViewSalesOrders
WHERE    (ProductID = :Param1)
```

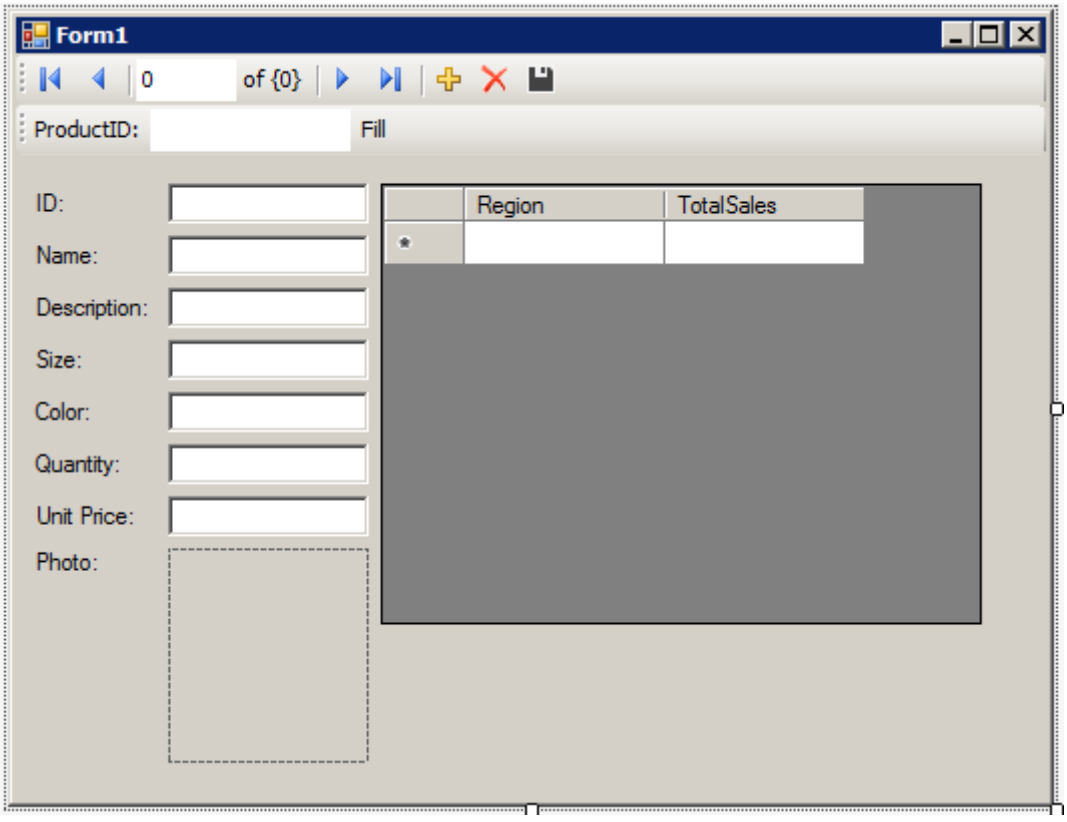
6. この SQL クエリを次のように変更します。
   a. `Quantity` を `SUM(Quantity) AS TotalSales` に変更します。
   b. `GROUP BY Region` を WHERE 句の後に続くクエリの末尾に追加します。

   変更した SQL クエリは次のようになります。

```
SELECT   Region, SUM(Quantity) as TotalSales
FROM     GROUPO.ViewSalesOrders
WHERE    (ProductID = :Param1)
GROUP BY Region
```

7. *OK* をクリックします。
8. 終了をクリックします。

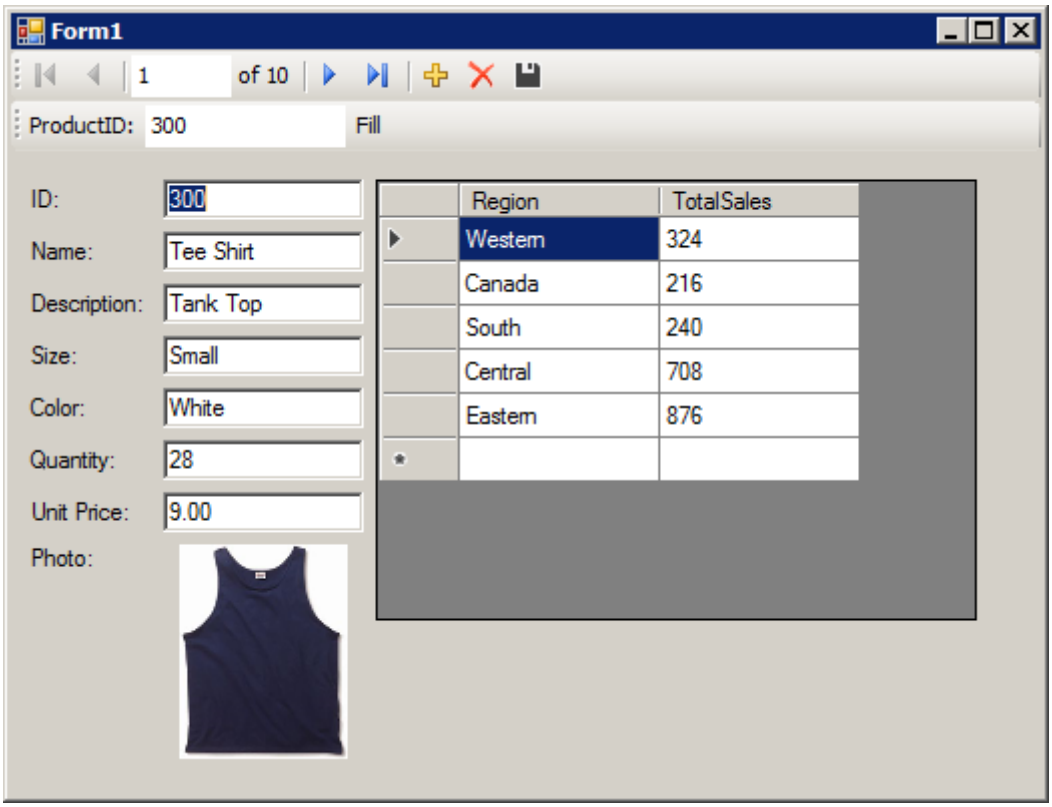
   ViewSalesOrders という新しい *TableAdapter* がデータセットデザイナウィンドウに追加されました。
9. フォームデザインタブ (Form1.cs [Design]) をクリックします。
   - フォームを右方向にドラッグして拡大し、新しいコントロールの領域を確保します。
10. データソースウィンドウで [ViewSalesOrders] を展開します。
    a. [ViewSalesOrders] をクリックし、ドロップダウンリストから *DataGridView* をクリックします。
    b. [ViewSalesOrders] をクリックして、フォーム (Form1) にドラッグします。



データグリッドビューのコントロールがフォームに表示されます。

11. プロジェクトをビルドし、実行します。
    - ビルド ▸ ソリューションのビルド をクリックします。
    - デバッグ ▸ デバッグの開始 をクリックします。
    - *Param1* または *ProductID* (Microsoft Visual Studio 2010 以降) テキストボックスに、300 などの製品 ID を入力して *Fill* をクリックします。

    入力した製品 ID の販売概要が、データグリッドビューに地域別に表示されます。



フォームのもう一方のコントロールを使用して、結果セットのローを移動することもできます。

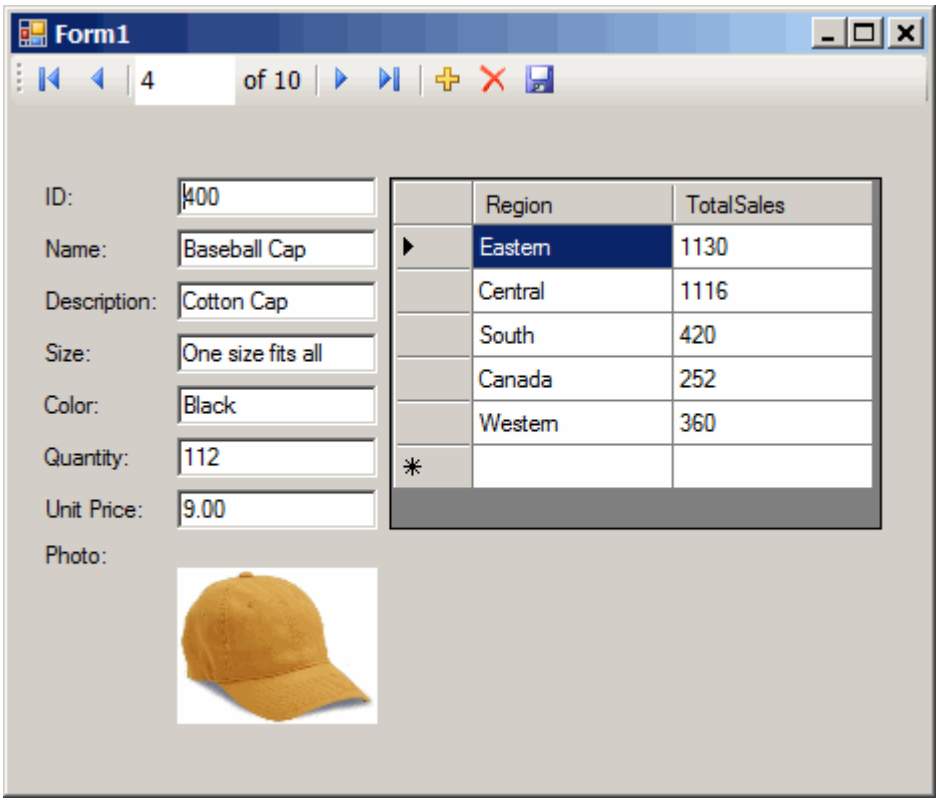
ただし、2 つのコントロールが互いに同期された状態になっていることが理想的です。これを実現するための手順を次に示します。

12. アプリケーションを終了してプロジェクトを保存します。
13. Fill ストリップは不要なのでフォームから削除します。
    - デザインフォーム (Form1) で、*Fill* の右側にある Fill ストリップを右クリックして、削除をクリックします。

    Fill ストリップがフォームから削除されます。
14. 2 つのコントロールを次のようにして同期します。
    a. デザインフォーム (Form1) で [ID] テキストボックスを右クリックし、プロパティをクリックします。
    b. イベントボタン (電球で表示される) をクリックします。
    c. *TextChanged* イベントが見つかるまでスクロールダウンします。
    d. *TextChanged* をクリックし、ドロップダウンリストから *fillToolStripButton_Click* をクリックします。Microsoft Visual Basic を使用している場合は、*FillToolStripButton_Click* というイベントです。

e. *fillToolStripButton_Click* をダブルクリックして、`fillToolStripButton_Click` イベントハンドラに関するフォームのコードウィンドウを開きます。
f. `param1ToolStripTextBox` または `productIDToolStripTextBox` (Microsoft Visual Studio 2010) への参照を検索し、これを `iDTextBox` に変更します。Microsoft Visual Basic を使用している場合は、`IDTextBox` というテキストボックスです。
g. プロジェクトを再度ビルドし、実行します。

15. アプリケーションフォームに表示されるナビゲーションコントロールが 1 つだけになりました。
    - 結果セットを移動すると、それに合わせて現在の製品の販売概要が更新され、データグリッドビューに地域ごとに表示されます。



16. アプリケーションを終了してプロジェクトを保存します。

結果

これで、結果セットの移動に合わせて内容を自動的に更新するコントロールが追加されました。

このチュートリアルでは、強力なツールである Microsoft Visual Studio、サーバエクスプローラ、および SQL Anywhere .NET データプロバイダを組み合わせて使用して、データベースアプリケーションを作成する方法を学習しました。

タスクの概要: チュートリアル:Microsoft Visual Studio を使用したシンプルな .NET データベースアプリケーションの開発 [109 ページ]

前のタスク: レッスン 1:Table Viewer の作成 [110 ページ]

1.3.3 SQL Anywhere ASP.NET プロバイダ

SQL Anywhere ASP.NET プロバイダは、Microsoft SQL Server の標準の ASP.NET プロバイダに代わり、SQL Anywhere を使用して Web サイトを運用できるようにします。

プロバイダは 5 つあります。

メンバーシッププロバイダ

メンバーシッププロバイダは、認証と承認のサービスを提供します。メンバーシッププロバイダは、新しいユーザやパスワードの作成、ユーザの ID の検証に使用します。

ロールプロバイダ

ロールプロバイダには、役割の作成、役割へのユーザの追加、役割の削除のためのメソッドがあります。ロールプロバイダは、グループへのユーザの割り当てや、パーミッションの管理に使用します。

プロファイルプロバイダ

プロファイルプロバイダには、ユーザ情報の読み込み、保存、取得のためのメソッドがあります。プロファイルプロバイダは、ユーザ設定の保存に使用します。

Web パーツパーソナル化プロバイダ

Web パーツパーソナル化プロバイダには、パーソナル化した Web ページのコンテンツやレイアウトをロードしたり保存したりするためのメソッドがあります。Web パーツパーソナル化プロバイダは、ユーザによる Web サイトのパーソナル化したビューの作成を可能にするために使用します。

ヘルスモニタリングプロバイダ

ヘルスモニタリングプロバイダには、配備された Web アプリケーションのステータスをモニタリングするためのメソッドがあります。ヘルスモニタリングプロバイダは、アプリケーションのパフォーマンスのモニタリング、問題のあるアプリケーションやシステムの特定、重要なイベントのログと確認に使用します。

SQL Anywhere ASP.NET プロバイダで使用されるデータベースサーバのスキーマは、標準の ASP.NET プロバイダで使用されるスキーマと同じです。データを操作し、保存する方法は同じです。

SQL Anywhere ASP.NET プロバイダの設定を終了したら、Microsoft Visual Studio ASP.NET の Web サイト管理ツールを使用して、ユーザと役割の作成および管理を実行できます。また、Microsoft Visual Studio の Login、LoginView、PasswordRecovery の各ツールを使用して Web サイトにセキュリティを追加できます。プロバイダのより高度な機能を使用したり、独自のログイン制御を作成したりするには、静的ラッパークラスを使用します。

このセクションの内容:

ASP.NET プロバイダのデータベース設定 [119 ページ]
SQL Anywhere ASP.NET プロバイダを実装するために、必要なスキーマを新しいデータベースまたは既存のデータベースに追加できます。

接続文字列の登録 [121 ページ]
接続文字列を登録するには、ODBC データソースの使用または完全な接続文字列の使用の 2 通りの方法があります。

SQL Anywhere ASP.NET プロバイダの登録 [121 ページ]
デフォルトのプロバイダではなく SQL Anywhere ASP.NET プロバイダを使用するように Web アプリケーションを構成する必要があります。

メンバーシッププロバイダの XML 属性 [123 ページ]
メンバーシッププロバイダの記述に使用される複数の XML 属性があります。

ロールプロバイダのテーブルスキーマ [124 ページ]
SARoleProvider では、プロバイダデータベースの aspnet_Roles テーブルに役割情報が格納されます。SARoleProvider に関連付けられているネームスペースは iAnywhere.Web.Security です。Roles テーブル内の各レコードは 1 つの役割に対応します。

プロファイルプロバイダのテーブルスキーマ [125 ページ]
SAProfileProvider では、プロバイダデータベースの aspnet_Profile テーブルにプロファイルデータが格納されます。SAProfileProvider に関連付けられているネームスペースは iAnywhere.Web.Security です。Profile テーブル内の各レコードは 1 人のユーザの持続されるプロファイルプロパティに対応します。

Web パーツパーソナル化プロバイダのテーブルスキーマ [125 ページ]
SAPersonalizationProvider では、プロバイダデータベースの aspnet_Paths テーブルにパーソナル化されたユーザコンテンツが保存されます。SAPersonalizationProvider に関連付けられているネームスペースは iAnywhere.Web.Security です。

ヘルスモニタリングプロバイダのテーブルスキーマ [126 ページ]
SAWebEventProvider では、プロバイダデータベースの aspnet_WebEvent_Events テーブルに Web イベントのログが取られます。SAWebEventProvider に関連付けられているネームスペースは iAnywhere.Web.Security です。WebEvents_Events テーブル内の各レコードは 1 つの Web イベントに対応します。

1.3.3.1 ASP.NET プロバイダのデータベース設定

SQL Anywhere ASP.NET プロバイダを実装するために、必要なスキーマを新しいデータベースまたは既存のデータベースに追加できます。

データベースにスキーマを追加するには、`SASetupAspNet.exe` を実行します。実行された `SASetupAspNet.exe` はデータベースに接続し、SQL Anywhere ASP.NET プロバイダに必要なテーブルとストアドプロシージャを作成します。SQL Anywhere ASP.NET プロバイダのすべてのリソースには、`aspnet_` というプレフィックスが付きます。既存のデータベースリソースとの名前の競合を最小限に抑えるためには、プロバイダのデータベースリソースを任意のデータベースユーザでインストールします。

`SASetupAspNet.exe` を実行するには、ウィザードまたはコマンドラインを使用できます。ウィザードを使用するには、アプリケーションを実行するか、コマンドライン文を引数なしで実行します。コマンドラインを使用して `SASetupAspNet.exe` にアクセスする場合は、引数に疑問符 (-?) を指定すると、データベース構成に関する詳細なヘルプを表示できます。

データベース接続の設定

作成するデータベースオブジェクトに適した権限を持つユーザで、データベースに接続します。次の最小の権限セットが必要です。

- CREATE ANY TABLE
- SELECT ANY TABLE
- DELETE ANY TABLE
- INSERT ANY TABLE
- ALTER ANY TABLE
- DROP ANY TABLE

- CREATE ANY PROCEDURE
- ALTER ANY PROCEDURE
- DROP ANY PROCEDURE
- CREATE ANY INDEX
- ALTER ANY INDEX
- DROP ANY INDEX

リソース所有者の指定

ウィザードとコマンドライン (*-U <ユーザ>*) で新しいリソース (テーブルとプロシージャ) の所有者を指定できます。デフォルトの所有者は DBA です。ウィザードを使用して、異なる所有者を指定できます。SQL Anywhere ASP.NET プロバイダの接続文字列を指定する場合、UserID (UID) はここで指定するユーザと一致している必要があります。指定されたユーザに追加の特権を付与する必要はありません。ユーザはリソースを所有し、ウィザードによってこのユーザに作成されるテーブルとストアドプロシージャに対する完全な特権を持っています。

機能の選択とデータの保持

特定の機能を選択して追加や削除することができます。共通のコンポーネントは自動的にインストールされます。アンインストールされている機能の削除を選択しても効果はありません。すでにインストールされている機能の追加を選択すると、その機能が再インストールされます。デフォルトでは、選択した機能に関連付けられているテーブル内のデータは保持されます。ユーザがテーブルのスキーマを大幅に変更した場合は、テーブル内に保存されたデータを自動的に保持できない場合があります。新規に再インストールする必要がある場合は、データの保持をオフにできます。

メンバーシッププロバイダとロールプロバイダは同時にインストールする必要があります。メンバーシッププロバイダがロールプロバイダと一緒にインストールされていない場合、Microsoft Visual Studio ASP.NET の Web サイト管理ツールの効率性が減少します。

詳細情報

Tutorial: Creating an ASP.NET Web Page Using SQL Anywhere というホワイトペーパーでは、SQL Anywhere と Microsoft Visual Studio 2010 を使用してデータベース駆動型 ASP.NET Web サイトを作成する方法を紹介しています。

関連情報

SQL Anywhere と Microsoft .NET

1.3.3.2 接続文字列の登録

接続文字列を登録するには、ODBC データソースの使用または完全な接続文字列の使用の 2 通りの方法があります。

- ODBC データソースを、ODBC データソースアドミニストレータを使用して登録し、名前で参照できます。
- または、完全な接続文字列を指定できます。次に例を示します。

```
connectionString="SERVER=MyServer;DBN=MyDatabase;UID=DBA;PWD=passwd"
```

<connectionStrings> 要素を web.config ファイルに追加すると、接続文字列とそのプロバイダをアプリケーションで参照できるようになります。更新は 1 箇所で実装できます。

接続文字列を登録する XML コードサンプル

```
<connectionStrings>
   <add name="MyConnectionString"
      connectionString="DSN=MyDataSource"
      providerName="Sap.Data.SQLAnywhere"/>
</connectionStrings>
```

1.3.3.3 SQL Anywhere ASP.NET プロバイダの登録

デフォルトのプロバイダではなく SQL Anywhere ASP.NET プロバイダを使用するように Web アプリケーションを構成する必要があります。

SQL Anywhere ASP.NET プロバイダを登録するには、次の操作を行います。

- iAnywhere.Web.Security アセンブリへの参照を Web サイトに追加します。
- web.config ファイルの <system.web> 要素に、各プロバイダのエントリを追加します。
- SQL Anywhere ASP.NET プロバイダの名前をアプリケーションの defaultProvider 属性に追加します。

プロバイダのデータベースには、複数のアプリケーションのデータを格納できます。この場合、個々の SQL Anywhere ASP.NET プロバイダについて、各アプリケーションの applicationName 属性が同じである必要があります。applicationName の値を指定しなかった場合、プロバイダデータベースで、各プロバイダに同じ名前が割り当てられます。

以前に登録した接続文字列を参照するには、connectionString 属性を connectionStringName 属性に置き換えます。

メンバーシッププロバイダを登録する XML コードサンプル

```
<membership defaultProvider="SAMembershipProvider">
   <providers>
      <add name="SAMembershipProvider"
         type="iAnywhere.Web.Security.SAMembershipProvider"
```

```
         connectionStringName="MyConnectionString"
         applicationName="MyApplication"
         commandTimeout="30"
                  enablePasswordReset="true"
         enablePasswordRetrieval="false"
         maxInvalidPasswordAttempts="5"
         minRequiredNonalphanumericCharacters="1"
         minRequiredPasswordLength="7"
         passwordAttemptWindow="10"
         passwordFormat="Hashed"
         requiresQuestionAndAnswer="true"
         requiresUniqueEmail="true"
         passwordStrengthRegularExpression="" />
   </providers>
</membership>
```

ロールプロバイダを登録する XML コードサンプル

```
<roleManager enabled="true" defaultProvider="SARoleProvider">
   <providers>
      <add name="SARoleProvider"
         type="iAnywhere.Web.Security.SARoleProvider"
         connectionStringName="MyConnectionString"
         applicationName="MyApplication"
         commandTimeout="30" />
   </providers>
</roleManager>
```

プロファイルプロバイダを登録する XML コードサンプル

```
<profile defaultProvider="SAProfileProvider">
   <providers>
      <add name="SAProfileProvider"
         type="iAnywhere.Web.Security.SAProfileProvider"
         connectionStringName="MyConnectionString"
         applicationName="MyApplication"
         commandTimeout="30" />
   </providers>
      <properties>
         <add name="UserString" type="string"
            serializeAs="Xml" />
         <add name="UserObject" type="object"
            serializeAs="Binary" />
   </properties>
</profile>
```

パーソナル化プロバイダを登録する XML コードサンプル

```
<webParts>
   <personalization defaultProvider="SAPersonalizationProvider">
      <providers>
```

```
         <add name="SAPersonalizationProvider"
            type="iAnywhere.Web.Security.SAPersonalizationProvider"
            connectionStringName="MyConnectionString"
            applicationName="MyApplication"
            commandTimeout="30" />
      </providers>
   </personalization>
</webParts>
```

ヘルスモニタリングプロバイダを登録する XML コードサンプル

```
<healthMonitoring enabled="true">
                  ...
   <providers>
      <add name="SAWebEventProvider"
         type="iAnywhere.Web.Security.SAWebEventProvider"
         connectionStringName="MyConnectionString"
         commandTimeout="30"
         bufferMode="Notification"
         maxEventDetailsLength="Infinite" /
   </providers>
                ...
</healthMonitoring>
```

関連情報

メンバーシッププロバイダの XML 属性 [123 ページ]
ロールプロバイダのテーブルスキーマ [124 ページ]
プロファイルプロバイダのテーブルスキーマ [125 ページ]
Web パーツパーソナル化プロバイダのテーブルスキーマ [125 ページ]
ヘルスモニタリングプロバイダのテーブルスキーマ [126 ページ]
How To:ASP.NET 2.0 でヘルスモニタリング機能を使用する方法

1.3.3.4 メンバーシッププロバイダの XML 属性

メンバーシッププロバイダの記述に使用される複数の XML 属性があります。

カラム名	説明
name	プロバイダの名前。
type	`iAnywhere.Web.Security.SAMembershipProvider`
connectionStringName	`<connectionStrings>` 要素で指定された接続文字列の名前。

カラム名	説明
connectionString	接続文字列。省略可能です。connectionStringName が指定されていない場合に必要。
applicationName	プロバイダデータを関連付けるアプリケーション名。
commandTimeout	サーバ呼び出しのタイムアウト値 (秒単位)。
enablePasswordReset	有効なエントリは true または false です。
enablePasswordRetrieval	有効なエントリは true または false です。
maxInvalidPasswordAttempts	有効なエントリは true または false です。
minRequiredNonalphanumericCharacters	有効なパスワードに最低限含める必要がある特殊文字の数。
minRequiredPasswordLength	パスワードに最低限必要な長さ。
passwordAttemptWindow	有効なパスワードまたはパスワードの解答の指定までに、連続して失敗した試行を追跡する時間。
passwordFormat	有効なエントリは Clear、Hashed、または Encrypted です。
requiresQuestionAndAnswer	有効なエントリは true または false です。
requiresUniqueEmail	有効なエントリは true または false です。
passwordStrengthRegularExpression	パスワードの評価に使用される正規表現。

1.3.3.5 ロールプロバイダのテーブルスキーマ

SARoleProvider では、プロバイダデータベースの aspnet_Roles テーブルに役割情報が格納されます。SARoleProvider に関連付けられているネームスペースは iAnywhere.Web.Security です。Roles テーブル内の各レコードは 1 つの役割に対応します。

SARoleProvider では aspnet_UsersInRoles テーブルを使用して、役割がユーザに割り当てられます。

カラム名	説明
name	プロバイダの名前。
type	iAnywhere.Web.Security.SARoleProvider
connectionStringName	<connectionStrings> 要素で指定された接続文字列の名前。
connectionString	接続文字列。省略可能です。connectionStringName が指定されていない場合に必要。
applicationName	プロバイダデータを関連付けるアプリケーション名。
commandTimeout	サーバ呼び出しのタイムアウト値 (秒単位)。

1.3.3.6 プロファイルプロバイダのテーブルスキーマ

SAProfileProvider では、プロバイダデータベースの aspnet_Profile テーブルにプロファイルデータが格納されます。SAProfileProvider に関連付けられているネームスペースは iAnywhere.Web.Security です。Profile テーブル内の各レコードは 1 人のユーザの持続されるプロファイルプロパティに対応します。

カラム名	説明
name	プロバイダの名前。
type	`iAnywhere.Web.Security.SAProfileProvider`
connectionStringName	`<connectionStrings>` 要素で指定された接続文字列の名前。
connectionString	接続文字列。省略可能です。`connectionStringName` が指定されていない場合に必要。
applicationName	プロバイダデータを関連付けるアプリケーション名。
commandTimeout	サーバ呼び出しのタイムアウト値 (秒単位)。

1.3.3.7 Web パーツパーソナル化プロバイダのテーブルスキーマ

SAPersonalizationProvider では、プロバイダデータベースの aspnet_Paths テーブルにパーソナル化されたユーザコンテンツが保存されます。SAPersonalizationProvider に関連付けられているネームスペースは iAnywhere.Web.Security です。

SAPersonalizationProvider では、aspnet_PersonalizationPerUser テーブルと aspnet_PersonalizationAllUsers テーブルを使用して、Web パーツパーソナル化のステータスが保存されているパスが定義されます。PathID カラムは、aspnet_Paths テーブル内の同じ名前のカラムを参照します。

カラム名	説明
name	プロバイダの名前。
type	`iAnywhere.Web.Security.SAPersonalizationProvider`
connectionStringName	`<connectionStrings>` 要素で指定された接続文字列の名前。
connectionString	接続文字列。省略可能です。`connectionStringName` が指定されていない場合に必要。
applicationName	プロバイダデータを関連付けるアプリケーション名。
commandTimeout	サーバ呼び出しのタイムアウト値 (秒単位)。

1.3.3.8 ヘルスモニタリングプロバイダのテーブルスキーマ

SAWebEventProvider では、プロバイダデータベースの aspnet_WebEvent_Events テーブルに Web イベントのログが取られます。SAWebEventProvider に関連付けられているネームスペースは iAnywhere.Web.Security です。WebEvents_Events テーブル内の各レコードは 1 つの Web イベントに対応します。

カラム名	説明
name	プロバイダの名前。
type	`iAnywhere.Web.Security.SAWebEventProvider`
connectionStringName	`<connectionStrings>` 要素で指定された接続文字列の名前。
connectionString	接続文字列。省略可能です。`connectionStringName` が指定されていない場合に必要。
commandTimeout	サーバ呼び出しのタイムアウト値 (秒単位)。
maxEventDetailsLength	各イベントの詳細文字列の最大長または Infinite。

関連情報

How To:ASP.NET 2.0 でヘルスモニタリング機能を使用する方法

1.3.4 SQL Anywhere .NET API リファレンス

SQL Anywhere .NET Data Provider の API を含め、SQL Anywhere を .NET で使用します。

ネームスペース

- `Sap.Data.SQLAnywhere`
- `Sap.SQLAnywhere.Server`

> **注記**
>
> API リファレンスマニュアルをお探しですか。マニュアルをローカルにインストールした場合は、Windows のスタートメニューを使用してアクセスするか (Microsoft Windows)、`C:¥Program Files¥SQL Anywhere 17¥Documentation` にナビゲートします。
>
> また、DocCommentXchange の Web で、SAP SQL Anywhere API リファレンスマニュアルにアクセスすることもできます。http://dcx.sap.com

1.4 OLE DB と ADO の開発

OLE DB および ADO アプリケーションの OLE DB プロバイダがソフトウェアに含まれています。

OLE DB は、Microsoft が開発した一連のコンポーネントオブジェクトモデル (COM: Component Object Model) インタフェースです。さまざまな情報ソースに格納されているデータに対して複数のアプリケーションから同じ方法でアクセスしたり、追加のデータベースサービスを実装したりできるようにします。これらのインタフェースは、データストアに適した多数の DBMS 機能をサポートし、データを共有できるようにします。

ADO は、OLE DB システムインタフェースを通じてさまざまなデータソースに対してプログラムからアクセス、編集、更新するためのオブジェクトモデルです。ADO も Microsoft が開発したものです。OLE DB プログラミングインタフェースを使用しているほとんどの開発者は、OLE DB API に直接記述するのではなく、ADO API に記述しています。

ADO インタフェースと ADO.NET を混同しないでください。ADO.NET は別のインタフェースです。

OLE DB と ADO のプログラミングに関するマニュアルについては、Microsoft Developer Network を参照してください。OLE DB と ADO の開発に関する製品固有の情報については、このマニュアルを使用してください。

このセクションの内容:

OLE DB [128 ページ]
OLE DB は Microsoft が提供するデータアクセスモデルです。OLE DB は、Component Object Model (COM) インタフェースを使用します。ODBC と違って、OLE DB は、データソースが SQL クエリプロセッサを使用することを仮定していません。

OLE DB プロバイダによる ADO プログラミング [129 ページ]
ADO (Microsoft ActiveX Data Objects) は Automation インタフェースを通じて公開されるデータアクセスオブジェクトモデルで、オブジェクトに関する予備知識がなくても、クライアントアプリケーションが実行時にオブジェクトのメソッドとプロパティを発見できるようにします。

OLE DB 接続パラメータ [137 ページ]
OLE DB 接続パラメータは、Microsoft が定義しています。OLE DB プロバイダは、これらの接続パラメータのサブセットをサポートしています。

OLE DB 接続プーリング [139 ページ]
OLE DB の .NET Framework データプロバイダは、OLE DB セッションプーリングを使用して接続を自動的にプールします。

OLE DB 独立性レベル [139 ページ]
OLE DB プロバイダでは、追加の独立性レベルがサポートされます。下記の一覧では、最初の 4 つが標準の OLE DB 独立性レベルです。最後の 3 つはデータベースサーバによってサポートされます。

Microsoft リンクサーバ [140 ページ]
OLE DB プロバイダを使用してデータベースへのアクセスを取得する Microsoft リンクサーバを作成することができます。SQL クエリの発行には、Microsoft の 4 部構成のテーブル参照構文または Microsoft の OPENQUERY SQL 関数を使用できます。

サポートされる OLE DB インタフェース [144 ページ]
OLE DB API はインタフェースのセットで構成されています。

OLE DB プロバイダの登録 [148 ページ]
ソフトウェアに付属のインストーラを使用して SAOLEDB プロバイダがインストールされるときに、SAOLEDB プロバイダはそれ自身を登録します。

関連情報

SQL Anywhere .NET データプロバイダ [52 ページ]

1.4.1 OLE DB

OLE DB は Microsoft が提供するデータアクセスモデルです。OLE DB は、Component Object Model (COM) インタフェースを使用します。ODBC と違って、OLE DB は、データソースが SQL クエリプロセッサを使用することを仮定していません。

ソフトウェアには SAOLEDB という **OLE DB** プロバイダが含まれています。このプロバイダは、サポートされている Windows プラットフォームで使用できます。

また、SQL Anywhere に内蔵の ODBC ドライバとともに、Microsoft OLE DB Provider for ODBC (MSDASQL) を使用して、データベースにアクセスすることもできます。

MSDASQL の代わりに SAOLEDB を使用することには、いくつかの利点があります。

- カーソルによる更新など、OLE DB/ODBC ブリッジを使用している場合には利用できない機能がいくつかあります。
- OLE DB プロバイダを使用する場合は、配備時に ODBC は必要ありません。
- MSDASQL によって、OLE DB クライアントはどの ODBC ドライバでも動作しますが、各 ODBC ドライバが備えている機能のすべてを利用できるかどうかは、保証されていません。SAOLEDB を使用すると、OLE DB プログラミング環境からデータベースソフトウェアのすべての機能を利用できます。

このセクションの内容:

OLE DB プラットフォームのサポート [129 ページ]
OLE DB プロバイダは、Microsoft Data Access Components (MDAC) 2.8 以降のバージョンで動作するように設計されています。

OLE DB での分散トランザクション [129 ページ]
OLE DB ドライバを、分散トランザクション環境のリソースマネージャとして使用できます。

1.4.1.1 OLE DB プラットフォームのサポート

OLE DB プロバイダは、Microsoft Data Access Components (MDAC) 2.8 以降のバージョンで動作するように設計されています。

1.4.1.2 OLE DB での分散トランザクション

OLE DB ドライバを、分散トランザクション環境のリソースマネージャとして使用できます。

関連情報

3 層コンピューティングと分散トランザクション [694 ページ]

1.4.2 OLE DB プロバイダによる ADO プログラミング

ADO (Microsoft ActiveX Data Objects) は Automation インタフェースを通じて公開されるデータアクセスオブジェクトモデルで、オブジェクトに関する予備知識がなくても、クライアントアプリケーションが実行時にオブジェクトのメソッドとプロパティを発見できるようにします。

Automation によって、Microsoft Visual Basic のようなスクリプト記述言語は標準のデータアクセスオブジェクトモデルを使用できるようになります。ADO は OLE DB を使用してデータアクセスを提供します。

OLE DB プロバイダを使用して、ADO プログラミング環境からデータベースソフトウェアのすべての機能を利用できます。

Microsoft Visual Basic および ADO を使用し、データベースへの接続、SQL クエリの実行、結果セットの取得などの基本的なタスクを示しますこれは、ADO を使用したプログラミングに関する完全なガイドではありません。

コードサンプルは、*%SQLANYSAMP17%*`¥SQLAnywhere¥VBSampler¥vbsampler.sln` プロジェクトファイルにあります。

ADO によるプログラミングについては、開発ツールのマニュアルを参照してください。

このセクションの内容:

Connection オブジェクトを使用してデータベースに接続する方法 [130 ページ]
次の Microsoft Visual Basic ルーチンにより、Connection オブジェクトを使用してデータベースに接続します。

Command オブジェクトを使用して文を実行する方法 [131 ページ]
次のルーチンは Command オブジェクトを使用して、単純な SQL 文をデータベースに送信します。

Recordset オブジェクトを使用して結果セットを取得する方法 [132 ページ]
ADO の Recordset オブジェクトは、クエリの結果セットを表します。これを使用して、データベースのデータを参照できます。

Recordset オブジェクト [134 ページ]

ADO の Recordset はカーソルを表します。Recordset オブジェクトの CursorType プロパティを宣言することでカーソルのタイプを選択してから、Recordset を開きます。カーソルタイプの選択は、Recordset で行える操作を制御し、パフォーマンスを左右します。

Recordset オブジェクトを使用したカーソルによるローの更新 [135 ページ]

OLE DB プロバイダでは、カーソルを使用して結果セットを更新できます。この機能は、MSDASQL プロバイダでは使用できません。

ADO のトランザクション [136 ページ]

デフォルトでは、ADO を使用したデータベースの変更は実行と同時にコミットされます。これには、明示的な更新、および Recordset の UpdateBatch メソッドも含まれます。

1.4.2.1 Connection オブジェクトを使用してデータベースに接続する方法

次の Microsoft Visual Basic ルーチンにより、Connection オブジェクトを使用してデータベースに接続します。

サンプルコード

このルーチンは、フォームに cmdTestConnection というコマンドボタンを配置し、その Click イベントに次のルーチンをペーストすることで試用できます。プログラムを実行し、ボタンをクリックして接続と切断を行います。

```
Private Sub cmdTestConnection_Click( _
        ByVal eventSender As System.Object, _
        ByVal eventArgs As System.EventArgs) _
        Handles cmdTestConnection.Click

    ' Declare variables
    Dim myPwd as String
    Dim myConn As New ADODB.Connection
    Dim myCommand As New ADODB.Command
    Dim cAffected As Integer
    On Error GoTo HandleError
    ' Establish the connection
    myConn.Provider = "SAOLEDB"
    myConn.ConnectionString = _
        "Data Source=SQL Anywhere 17 Demo;" + _
    "Password=" + myPwd
    myConn.Open()
    MsgBox("Connection succeeded")
    myConn.Close()
    Exit Sub
HandleError:
    MsgBox(ErrorToString(Err.Number))
    Exit Sub
End Sub
```

注記

この例では、次の作業を行います。

- ルーチンで使われる変数を宣言します。
- OLE DB プロバイダを使用して、サンプルデータベースへの接続を確立します。
- Command オブジェクトを使用して簡単な文を実行し、データベースサーバメッセージウィンドウにメッセージを表示します。
- 接続を閉じます。

関連情報

OLE DB 接続パラメータ [137 ページ]

1.4.2.2 Command オブジェクトを使用して文を実行する方法

次のルーチンは Command オブジェクトを使用して、単純な SQL 文をデータベースに送信します。

サンプルコード

このルーチンは、フォームに cmdUpdate というコマンドボタンを配置し、その Click イベントに次のルーチンをペーストすることで試用できます。プログラムを実行し、ボタンをクリックして接続、データベースサーバメッセージウィンドウへのメッセージの表示、切断を行います。

```
Private Sub cmdUpdate_Click( _
        ByVal eventSender As System.Object, _
        ByVal eventArgs As System.EventArgs) _
        Handles cmdUpdate.Click

    ' Declare variables
    Dim myPwd as String
    Dim myConn As New ADODB.Connection
    Dim myCommand As New ADODB.Command
    Dim cAffected As Integer
    On Error GoTo HandleError
    ' Establish the connection
    myConn.Provider = "SAOLEDB"
    myConn.ConnectionString = _
        "Data Source=SQL Anywhere 17 Demo;" + _
    "Password=" + myPwd
    myConn.Open()
    'Execute a command
    myCommand.CommandText = _
        "UPDATE Customers SET GivenName='Liz' WHERE ID=102"
    myCommand.ActiveConnection = myConn
    myCommand.Execute(cAffected)
    MsgBox(CStr(cAffected) & " rows affected.", _
        MsgBoxStyle.Information)
```

```
    myConn.Close()
    Exit Sub
HandleError:
    MsgBox(ErrorToString(Err.Number))
    Exit Sub
End Sub
```

注意

サンプルコードは、接続を確立した後、Command オブジェクトを作成し、CommandText プロパティを update 文に、ActiveConnection プロパティを現在の接続に設定します。次に update 文を実行し、この更新で影響を受けるローの数をウィンドウに表示します。

この例では、更新はデータベースに送られ、実行と同時にコミットされます。

カーソルを使用して更新を実行することもできます。

関連情報

ADO のトランザクション [136 ページ]
Recordset オブジェクトを使用したカーソルによるローの更新 [135 ページ]

1.4.2.3 Recordset オブジェクトを使用して結果セットを取得する方法

ADO の Recordset オブジェクトは、クエリの結果セットを表します。これを使用して、データベースのデータを参照できます。

サンプルコード

このルーチンは、フォームに cmdQuery というコマンドボタンを配置し、その Click イベントに次のルーチンをペーストすることで試用できます。プログラムを実行し、ボタンをクリックして接続、データベースサーバメッセージウィンドウへのメッセージの表示を行います。次にクエリの実行、最初の 2、3 のローのウィンドウへの表示、切断を行います。

```
Private Sub cmdQuery_Click( _
        ByVal eventSender As System.Object, _
        ByVal eventArgs As System.EventArgs) _
        Handles cmdQuery.Click
    ' Declare variables
    Dim myPwd as String
    Dim i As Integer
    Dim myConn As New ADODB.Connection
    Dim myCommand As New ADODB.Command
    Dim myRS As New ADODB.Recordset

    On Error GoTo ErrorHandler
```

```
    ' Establish the connection
    myConn.Provider = "SAOLEDB"
    myConn.ConnectionString = _
        "Data Source=SQL Anywhere 17 Demo;" + _
    "Password=" + myPwd
    myConn.CursorLocation = _
        ADODB.CursorLocationEnum.adUseServer
    myConn.Mode = _
        ADODB.ConnectModeEnum.adModeReadWrite
    myConn.IsolationLevel = _
        ADODB.IsolationLevelEnum.adXactCursorStability
    myConn.Open()

    'Execute a query
    myRS = New ADODB.Recordset
    myRS.CacheSize = 50
    myRS.let_Source("SELECT * FROM Customers")
    myRS.let_ActiveConnection(myConn)
    myRS.CursorType = ADODB.CursorTypeEnum.adOpenKeyset
    myRS.LockType = ADODB.LockTypeEnum.adLockOptimistic
    myRS.Open()

    'Scroll through the first few results
    myRS.MoveFirst()
    For i = 1 To 5
        MsgBox(myRS.Fields("CompanyName").Value, _
            MsgBoxStyle.Information)
        myRS.MoveNext()
    Next

    myRS.Close()
    myConn.Close()
    Exit Sub

ErrorHandler:
    MsgBox(ErrorToString(Err.Number))
    Exit Sub
End Sub
```

注意

この例の Recordset オブジェクトは、Customers テーブルに対するクエリの結果を保持します。For ループは最初にあるいくつかのローをスクロールして、各ローに対する CompanyName の値を表示します。

次に、ADO からカーソルを使用する簡単な例を示します。

関連情報

Recordset オブジェクト [134 ページ]

1.4.2.4 Recordset オブジェクト

ADO の Recordset はカーソルを表します。Recordset オブジェクトの CursorType プロパティを宣言することでカーソルのタイプを選択してから、Recordset を開きます。カーソルタイプの選択は、Recordset で行える操作を制御し、パフォーマンスを左右します。

カーソルタイプ

ADO には、カーソルタイプに対する固有の命名規則があります。

使用できるカーソルタイプ、対応するカーソルタイプの定数、それらと同等の SQL Anywhere のタイプは、次のとおりです。

ADO カーソルタイプ	ADO 定数	データベースソフトウェアのタイプ
動的カーソル	adOpenDynamic	動的スクロールカーソル
キーセットカーソル	adOpenKeyset	スクロールカーソル
静的カーソル	adOpenStatic	非反映型カーソル
前方向カーソル	adOpenForwardOnly	非スクロールカーソル

サンプルコード

次のコードは、ADO の Recordset オブジェクトに対してカーソルタイプを設定します。

```
Dim myRS As New ADODB.Recordset
myRS.CursorType = ADODB.CursorTypeEnum.adOpenDynamic
```

関連情報

カーソルタイプ [25 ページ]
カーソルのプロパティ [26 ページ]

1.4.2.5 Recordset オブジェクトを使用したカーソルによるローの更新

OLE DB プロバイダでは、カーソルを使用して結果セットを更新できます。この機能は、MSDASQL プロバイダでは使用できません。

レコードセットの更新

データベースは Recordset を通じて更新できます。

```
Private Sub cmdUpdateThroughCursor_Click( _
        ByVal eventSender As System.Object, _
        ByVal eventArgs As System.EventArgs) _
        Handles cmdUpdateThroughCursor.Click
    ' Declare variables
    Dim myPwd As String
    Dim i As Integer
    Dim myConn As New ADODB.Connection
    Dim myRS As New ADODB.Recordset
    Dim SQLString As String
    On Error GoTo HandleError
    ' Connect
    myConn.Provider = "SAOLEDB"
    myConn.ConnectionString = _
        "Data Source=SQL Anywhere 17 Demo;" + _
        "Password=" + myPwd
    myConn.Open()
    myConn.BeginTrans()
    SQLString = "SELECT * FROM Customers"
    myRS.Open(SQLString, myConn, _
        ADODB.CursorTypeEnum.adOpenDynamic, _
        ADODB.LockTypeEnum.adLockBatchOptimistic)
    If myRS.BOF And myRS.EOF Then
        MsgBox("Recordset is empty!", 16, "Empty Recordset")
    Else
        MsgBox("Cursor type: " & CStr(myRS.CursorType), _
            MsgBoxStyle.Information)
        myRS.MoveFirst()
        For i = 1 To 3
            MsgBox("Row: " & CStr(myRS.Fields("ID").Value), _
                MsgBoxStyle.Information)
            If i = 2 Then
                myRS.Update("City", "Toronto")
                myRS.UpdateBatch()
            End If
            myRS.MoveNext()
        Next i
        myRS.Close()
    End If
    myConn.CommitTrans()
    myConn.Close()
    Exit Sub
HandleError:
    MsgBox(ErrorToString(Err.Number))
    Exit Sub
End Sub
```

注意

Recordset で adLockBatchOptimistic 設定を使用すると、myRS.Update メソッドはデータベース自体には何も変更を加えません。代わりに、Recordset のローカルコピーを更新します。

myRS.UpdateBatch メソッドはデータベースサーバに対して更新を実行しますが、コミットはしません。このメソッドは、トランザクションの内部で実行されるためです。トランザクションの外部で UpdateBatch メソッドを呼び出した場合、変更はコミットされます。

myConn.CommitTrans メソッドは、変更をコミットします。Recordset オブジェクトはこのときまでに閉じられているため、データのローカルコピーが変更されたかどうかが問題になることはありません。

1.4.2.6 ADO のトランザクション

デフォルトでは、ADO を使用したデータベースの変更は実行と同時にコミットされます。これには、明示的な更新、および Recordset の UpdateBatch メソッドも含まれます。

しかし、前の項では、トランザクションを使用するために、Connection オブジェクトで BeginTrans メソッドと RollbackTrans メソッドまたは CommitTrans メソッドを使用できると説明しました。

トランザクションの独立性レベルは、Connection オブジェクトのプロパティとして設定されます。IsolationLevel プロパティは、次の値のいずれかを取ることができます。

ADO 独立性レベル	定数	データベースサーバの独立性レベル
未指定	adXactUnspecified	不適用。0 に設定します。
混沌	adXactChaos	サポートされていません。0 に設定します。
参照	adXactBrowse	0
コミットされない読み出し	adXactReadUncommitted	0
カーソル安定性	adXactCursorStability	1
コミットされた読み出し	adXactReadCommitted	1
繰り返し可能読み出し	adXactRepeatableRead	2
独立	adXactIsolated	3
直列化可能	adXactSerializable	3
Snapshot	2097152	SNAPSHOT
文のスナップショット	4194304	STATEMENT SNAPSHOT
読み込み専用文のスナップショット	8388608	READONLY STATEMENT SNAPSHOT

1.4.3 OLE DB 接続パラメータ

OLE DB 接続パラメータは、Microsoft が定義しています。OLE DB プロバイダは、これらの接続パラメータのサブセットをサポートしています。

一般的な接続文字列は、次のようなものです。

```
"Provider=SAOLEDB;Data Source=myDsn;Initial Catalog=myDbn;
    User ID=myUid;Password=myPwd"
```

次に示すのは、プロバイダがサポートする OLE DB 接続パラメータです。場合によっては、OLE DB 接続パラメータがデータベースサーバ接続パラメータと同一 (Password など) または類似 (User ID など) します。これらの接続パラメータの多くでスペースの使用に注意してください。

Provider

このパラメータは、OLE DB プロバイダ (SAOLEDB) を特定するために使用されます。

User ID

この接続パラメータは、UserID (UID) 接続パラメータに直接マップされます。次に例を示します。`User ID=DBA`

Password

この接続パラメータは、Password (PWD) 接続パラメータに直接マップされます。次に例を示します。`Password=sql`

Data Source

この接続パラメータは、DataSourceName (DSN) 接続パラメータに直接マップされます。次に例を示します。`Data Source=SQL Anywhere 17 Demo`

Initial Catalog

この接続パラメータは、DatabaseName (DBN) 接続パラメータに直接マップされます。次に例を示します。`Initial Catalog=demo`

Location

この接続パラメータは、Host 接続パラメータに直接マップされます。パラメータ値は Host パラメータ値と同じ形式になります。次に例を示します。`Location=localhost:4444`

Extended Properties

この接続パラメータは、データベースサーバに固有のすべての接続パラメータに引き渡すために、OLE DB によって使用されます次に例を示します。`Extended Properties="UserID=DBA;DBKEY=V3moj3952B;DBF=demo.db"`

ADO は、この接続パラメータを使用して、認識しないすべての接続パラメータを収集して渡します。

Microsoft の接続ウィンドウの中には、*[Prov String]* または *[Provider String]* というフィールドがあるものがあります。このフィールドの内容が、Extended Properties への値として渡されます。

OLE DB Services

この接続パラメータは、OLE DB プロバイダからは直接処理されません。これは、ADO で接続プーリングを制御します。

Prompt

この接続パラメータは、接続がエラーを処理する方法を管理します。可能なプロンプト値は、1、2、3、4 のいずれかです。その意味は、DBPROMPT_PROMPT (1)、DBPROMPT_COMPLETE (2)、DBPROMPT_COMPLETEREQUIRED (3)、DBPROMPT_NOPROMPT (4) です。

デフォルトのプロンプト値は 4 であり、これは、プロバイダが接続ウィンドウを表示しないことを意味します。プロンプト値を 1 に設定すると、接続ウィンドウが常に表示されるようになります。プロンプト値を 2 に設定すると、初期接続試行が失敗

した場合に接続ウィンドウが表示されるようになります。プロンプト値を 3 に設定すると、初期接続試行が失敗した場合に接続ウィンドウが表示されるようになりますが、プロバイダはデータソースへの接続には必要ない情報に対する制御を無効にします。

Window Handle

アプリケーションは、適用できる場合、親ウィンドウのハンドルを渡すことができ、また、ウィンドウハンドルが適用できないか、プロバイダがどのウィンドウも表示しない場合、NULL ポインタを渡すことができます。ウィンドウハンドル値は通常は 0 (NULL) です。

その他の OLE DB 接続パラメータも指定できますが、OLE DB プロバイダからは無視されます。

OLE DB プロバイダは、起動されるときに、OLE DB 接続パラメータ用のプロパティ値を取得します。次に示すのは、Microsoft の RowsetViewer アプリケーションから取得した典型的なプロパティ値のセットです。

```
User ID 'DBA'
Password 'sql'
Location 'localhost:4444'
Initial Catalog 'demo'
Data Source 'testds'
Extended Properties 'appinfo=api=oledb'
Prompt 2
Window Handle 0
```

このパラメータ値のセットからプロバイダが構築する接続文字列は、次のとおりです。

```
'DSN=testds;HOST=localhost:4444;DBN=demo;UID=DBA;PWD=sql;appinfo=api=oledb'
```

OLE DB プロバイダは、接続文字列、Window Handle、Prompt 値を、作成するデータベースサーバ接続呼び出しへのパラメータとして使用します。

これは、簡単な ADO 接続文字列の例です。

```
connection.Open "Provider=SAOLEDB;Location=localhost:4444;UserID=DBA;Pwd=sql"
```

ADO は、接続文字列を解析して、認識できない接続パラメータすべてを Extended Properties で渡します。OLE DB プロバイダは、起動されるときに、OLE DB 接続パラメータ用のプロパティ値を取得します。次に示すのは、前述の接続文字列で使用された ADO アプリケーションから取得されたプロパティ値のセットです。

```
User ID ''
Password ''
Location 'localhost:4444'
Initial Catalog ''
Data Source ''
Extended Properties 'UserID=DBA;Pwd=sql'
Prompt 4
Window Handle 0
```

このパラメータ値のセットからプロバイダが構築する接続文字列は、次のとおりです。

```
'HOST=localhost:4444;UserID=DBA;Pwd=sql'
```

プロバイダは、接続文字列、Window Handle、Prompt 値を、作成するデータベースサーバ接続呼び出しへのパラメータとして使用します。

関連情報

OLE DB 接続プーリング [139 ページ]

1.4.4 OLE DB 接続プーリング

OLE DB の .NET Framework データプロバイダは、OLE DB セッションプーリングを使用して接続を自動的にプールします。

アプリケーションが接続を閉じても、実際には接続は閉じません。代わりに、接続は一定時間保持されます。アプリケーションが接続を再度開くと、ADO/OLE DB では、アプリケーションが同じ接続文字列を使用することを認識し、開いている接続を再使用します。たとえば、アプリケーションで Open/Execute/Close を 100 回実行しても、実際には 1 回開いて、1 回閉じるのみです。最後の閉じる操作は、約 1 分のアイドル時間が経過した後で行われます。

接続が外的手段 (*SQL Central* などの管理ツールを使用した強制切断など) で終了した場合は、サーバとの次の対話まで、この状態が発生したことが ADO/OLE DB によって把握されません。強制的に切断する場合は、十分に注意してください。

接続プーリングを制御するフラグは DBPROPVAL_OS_RESOURCEPOOLING (1) です。このフラグは、接続文字列の接続パラメータを使用してオフにできます。

接続文字列で **OLE DB Services=-2** と指定すると、接続プーリングが無効になります。サンプル接続文字列を以下に示します。

```
Provider=SAOLEDB;OLE DB Services=-2;...
```

接続文字列で **OLE DB Services=-4** と指定すると、接続プーリングとトランザクションのエンリストが無効になります。サンプル接続文字列を以下に示します。

```
Provider=SAOLEDB;OLE DB Services=-4;...
```

接続プーリングを無効にして、アプリケーションで同じ接続文字列を使用して接続を頻繁に開いたり閉じたりすると、パフォーマンスが低下します。

関連情報

SQL サーバの接続プーリング (ADO.NET)
プロバイダサービスのデフォルトの無効化
OLE DB、ODBC、Oracle の接続プーリング (ADO.NET)

1.4.5 OLE DB 独立性レベル

OLE DB プロバイダでは、追加の独立性レベルがサポートされます。下記の一覧では、最初の 4 つが標準の OLE DB 独立性レベルです。最後の 3 つはデータベースサーバによってサポートされます。

```
#define ISOLATIONLEVEL_READUNCOMMITTED              0x000100
```

```
#define ISOLATIONLEVEL_READCOMMITTED                0x001000
#define ISOLATIONLEVEL_REPEATABLEREAD               0x010000
#define ISOLATIONLEVEL_SERIALIZABLE                 0x100000
#define ISOLATIONLEVEL_SNAPSHOT                     0x200000
#define ISOLATIONLEVEL_STATEMENT_SNAPSHOT           0x400000
#define ISOLATIONLEVEL_READONLY_STATEMENT_SNAPSHOT  0x800000
```

SNAPSHOT 独立性を指定する例を次に示します。

```
hr = pITransactionLocal->StartTransaction( ISOLATIONLEVEL_SNAPSHOT, 0, NULL,
&ulTransactionLevel );
```

関連情報

ADO のトランザクション [136 ページ]

1.4.6 Microsoft リンクサーバ

OLE DB プロバイダを使用してデータベースへのアクセスを取得する Microsoft リンクサーバを作成することができます。SQL クエリの発行には、Microsoft の 4 部構成のテーブル参照構文または Microsoft の OPENQUERY SQL 関数を使用できます。

4 部分構成構文の例を次に示します。

```
SELECT * FROM SADATABASE.demo.GROUPO.Customers
```

この例では、SADATABASE はリンクサーバの名前であり、demo はカタログ名またはデータベース名であり、GROUPO はデータベースのテーブル所有者であり、また、Customers はデータベースのテーブル名です。

もう 1 つの例では、Microsoft の OPENQUERY 関数を使用しています。

```
SELECT * FROM OPENQUERY( SADATABASE, 'SELECT * FROM Customers' )
```

OPENQUERY 構文では、2 番目の SELECT 文 ('SELECT * FROM Customers') がデータベースサーバに渡され、実行されます。

複雑なクエリの場合、クエリ全体がデータベースサーバで評価されるため、OPENQUERY を使用する方が適しています。4 部構成の構文では、SQL Server は、クエリによって参照されるすべてのテーブルの内容を取得してから、クエリを評価できます (たとえば、WHERE、JOIN、ネストされたクエリを使用するクエリなど)。非常に大きいテーブルを必要とするクエリの場合、4 部分構成の構文を使用すると処理時間が非常に長くなる可能性があります。次の 4 部構成のクエリの例では、SQL Server は OLE DB プロバイダ経由で、テーブル全体に対する単純な SELECT (WHERE 句なし) をデータベースサーバに渡してから、WHERE 条件自体を評価します。

```
SELECT ID, Surname, GivenName FROM [SADATABASE].[demo].[GROUPO].[Customers]
WHERE Surname = 'Elkins'
```

結果セットの 1 つのローを SQL Server に返す代わりに、すべてのローが返され、この結果セットは SQL Server によって 1 つのローに減らされます。次の例では同じ結果が生成されますが、1 つのローのみが SQL Server に返されます。

```
SELECT * FROM OPENQUERY( SADATABASE,
    'SELECT ID, Surname, GivenName FROM [GROUPO].[Customers]
    WHERE Surname = ''Elkins''' )
```

Microsoft SQL Server 対話型アプリケーションまたは SQL Server スクリプトを使用して、OLE DB プロバイダを使用するリンクサーバを設定できます。

このセクションの内容:

インタラクティブなアプリケーションを使用したリンクサーバの設定 [141 ページ]
Microsoft SQL Server のインタラクティブなアプリケーションを使用して、OLE DB プロバイダを使用してデータベースへのアクセスを取得する Microsoft リンクサーバを作成します。

スクリプトを使用したリンクサーバの設定 [143 ページ]
Microsoft SQL Server スクリプトを使用して Microsoft リンクサーバ定義を設定します。

1.4.6.1 インタラクティブなアプリケーションを使用したリンクサーバの設定

Microsoft SQL Server のインタラクティブなアプリケーションを使用して、OLE DB プロバイダを使用してデータベースへのアクセスを取得する Microsoft リンクサーバを作成します。

前提条件

Microsoft SQL Server 2000 以降

リンクサーバの設定の前に、Windows 7 以降の Windows を使用するときに考慮すべきことがいくつかあります。Microsoft SQL Server は、システムのサービスとして実行します。Windows 7 以降のバージョンでのサービスの設定方法によっては、サービスが共有メモリ接続を使用できない可能性、サーバを起動できない可能性、およびユーザデータソース定義にアクセスできない可能性があります。たとえば、ネットワークサービスとしてログインするサービスがサーバを起動できない、共有メモリ経由で接続できない、ユーザデータソースをアクセスできないなどがあります。これらの状況において、データベースサーバは、事前に起動させる必要があり、TCPIP 接続プロトコルを使用する必要があります。また、データソースが使用される予定の場合、システムデータソースにする必要があります。

手順

1. Microsoft SQL Server 2005/2008 の場合は、Microsoft SQL Server Management Studio を起動します。他のバージョンの Microsoft SQL Server の場合は、このアプリケーション名とリンクサーバの設定手順が異なることがあります。

オブジェクトエクスプローラウィンドウ枠で、サーバオブジェクト > リンクサーバ を展開します。リンクサーバを右クリックし、新しいリンクサーバをクリックします。

2. 全般ページに必要な情報を入力します。

   全般ページのリンクサーバフィールドにリンクサーバ名を指定します (上記の例では SADATABASE)。

   その他のデータソースオプションを選択して、プロバイダリストから *SQL Anywhere OLE DB Provider 17* を選択します。

   プロダクト名フィールドには、任意の内容を入力できます (アプリケーション名など)。

   データソースフィールドには、ODBC データソース名を指定できます (DSN)。これは便利なオプションであり、また、データソース名は必須ではありません。システム DSN を使用する場合、Microsoft SQL Server の 32 ビットバージョン用の 32 ビット DSN であるか、Microsoft SQL Server の 64 ビットバージョンの 64 ビット DSN である必要があります。

   ```
   Data Source: SQL Anywhere 17 Demo
   ```

   プロバイダ文字列フィールドには、UserID (UID)、ServerName (Server)、DatabaseFile (DBF) などの追加の接続パラメータを含めることができます。

   ```
   Provider string: Server=myserver;DBF=sample.db
   ```

   場所フィールドは、Host 接続パラメータと同等のものを指定できます (localhost:4444 や 10.25.99.253:2638 など)。

   ```
   Location: AppServer-pc:2639
   ```

   初期カタログフィールドには、データベースの名前や接続先 (demo など) を含むことができます。データベースは、事前に開始しておく必要があります。

   ```
   Initial Catalog: demo
   ```

   これら最後の 4 つのフィールドとセキュリティページからのユーザ ID とパスワードの組み合わせは、データベースサーバへの接続に成功するための十分な情報を含む必要があります。

3. データベースのユーザ ID とパスワードは、プレーンテキストとして公開されてしまうプロバイダ文字列に接続文字列として指定する代わりに、セキュリティページで入力できます。

   Microsoft SQL Server 2005/2008 では、このセキュリティコンテキストを使用するオプションをクリックし、リモートログインとリモートパスワードフィールドに値を入力します (パスワードはアスタリスクで表示されます)。

4. サーバオプションページに移動します。

   RPC および *RPC* 出力オプションを有効にします。

   選択方法は、Microsoft SQL Server のバージョンによって異なります。Microsoft SQL Server 2000 の場合、この 2 つのオプションを指定するために 2 つのチェックボックスをオンにする必要があります。Microsoft SQL Server 2005/2008 の場合、このオプションは True/False で設定します。すべて True に設定されていることを確認してください。ストアドプロシージャまたはファンクションの呼び出しをデータベースで実行し、出入力パラメータの受け渡しを正常に行うには、リモートプロシージャコール (*RPC*) のオプションを選択する必要があります。

5. *InProcess* 許可プロバイダオプションを選択します。

   選択方法は、Microsoft SQL Server のバージョンによって異なります。Microsoft SQL Server 2000 の場合、プロバイダオプションボタンをクリックすると、このオプションを選択できるページに移動します。Microsoft SQL Server 2005/2008 の場合、リンクサーバ > プロバイダ の下で SAOLEDB.17 プロバイダを右クリックし、プロパティをクリ

ックします。*InProcess* 許可チェックボックスがオンになっていることを確認します。この *Inprocess* オプションがオンになっていないと、クエリが失敗します。

6. 他のプロバイダオプションは無視できます。これらのオプションのいくつかは Microsoft SQL Server の下位互換性に関連しており、Microsoft SQL Server が OLE DB プロバイダと対話する方法に影響を与えません。例はネストされたクエリと *LIKE* 演算子のサポートです。他のオプションを選択した場合、構文エラーまたはパフォーマンスの低下が発生することがあります。

結果

Microsoft リンクサーバが設定されます。

1.4.6.2 スクリプトを使用したリンクサーバの設定

Microsoft SQL Server スクリプトを使用して Microsoft リンクサーバ定義を設定します。

前提条件

Microsoft SQL Server 2005 以降

リンクサーバの設定の前に、Windows 7 以降の Windows を使用するときに考慮すべきことがいくつかあります。Microsoft SQL Server は、システムのサービスとして実行します。Windows 7 以降のバージョンでのサービスの設定方法によっては、サービスが共有メモリ接続を使用できない可能性、サーバを起動できない可能性、およびユーザデータソース定義にアクセスできない可能性があります。たとえば、ネットワークサービスとしてログインするサービスがサーバを起動できない、共有メモリ経由で接続できない、ユーザデータソースをアクセスできないなどがあります。これらの状況において、データベースサーバは、事前に起動させる必要があり、TCPIP 接続プロトコルを使用する必要があります。また、データソースが使用される予定の場合、システムデータソースにする必要があります。

コンテキスト

次の手順を使用して次のスクリプトに必要な変更を加えた後、Microsoft SQL Server で実行してください。

```
USE [master]
GO
EXEC master.dbo.sp_addlinkedserver @server=N'SADATABASE',
     @srvproduct=N'SAP DBMS', @provider=N'SAOLEDB.17',
     @datasrc=N'SQL Anywhere 17 Demo',
     @provstr=N'host=localhost:4444;server=myserver;dbn=demo'
GO
EXEC master.dbo.sp_serveroption @server=N'SADATABASE',
     @optname=N'rpc', @optvalue=N'true'
GO
EXEC master.dbo.sp_serveroption @server=N'SADATABASE',
     @optname=N'rpc out', @optvalue=N'true'
```

```
GO
-- Set remote login
EXEC master.dbo.sp_addlinkedsrvlogin @rmtsrvname = N'SADATABASE',
     @locallogin = NULL , @useself = N'False',
     @rmtuser = N'DBA', @rmtpassword = N'sql'
GO
-- Set global provider "allow in process" flag
EXEC master.dbo.sp_MSset_oledb_prop N'SAOLEDB.17', N'AllowInProcess', 1
```

手順

1. 新しいリンクサーバ名を選択します (例では SADATABASE が使用されています)。
2. オプションのデータソース名を選択します (例では SQL Anywhere 17 Demo が使用されています)。
3. オプションのプロバイダ文字列を選択します (例では `N'host=localhost:4444;server=myserver;dbn=demo'` が使用されています)。
4. リモートユーザ ID とパスワードを選択します (例では N'DBA' と N'sql' が使用されています)。

結果

変更されたスクリプトを Microsoft SQL Server で実行し、新しいリンクサーバを作成できます。

1.4.7 サポートされる OLE DB インタフェース

OLE DB API はインタフェースのセットで構成されています。

次の表は OLE DB ドライバによる各インタフェースのサポートを示します。

インタフェース	内容	制限事項
IAccessor	クライアントのメモリとデータストアの値のバインドを定義します。	DBACCESSOR_PASSBYREF はサポートされていません。 DBACCESSOR_OPTIMIZED はサポートされていません。
IAlterIndex IAlterTable	テーブル、インデックス、カラムを変更します。	サポートされていません。
IChapteredRowset	区分化されたローセットで、ローセットのローを別々の区分でアクセスできます。	サポートされていません。データベースサーバは、区分化されたローセットはサポートしません。
IColumnsInfo	ローセットのカラムについての簡単な情報を得ます。	サポートされています。

インタフェース	内容	制限事項
IColumnsRowset	ローセットにあるオプションのメタデータカラムについての情報を得て、カラムメタデータのローセットを取得します。	サポートされています。
ICommand	SQL 文を実行します。	設定できなかったプロパティを見つけるための、DBPROPSET_PROPERTIESINERROR による ICommandProperties::GetProperties の呼び出しは、サポートされていません。
ICommandPersist	command オブジェクトの状態を保持します (アクティブなローセットは保持しません)。保持されているこれらの command オブジェクトは、PROCEDURES か VIEWS ローセットを使用すると、続けて列挙できます。	サポートされています。
ICommandPrepare	コマンドを準備します。	サポートされています。
ICommandProperties	コマンドが作成したローセットに、Rowset プロパティを設定します。ローセットがサポートするインタフェースを指定するのに、最も一般的に使用されます。	サポートされています。
ICommandText	ICommand に SQL 文を設定します。	DBGUID_DEFAULT SQL ダイアレクトのみサポートされています。
ICommandWithParameters	コマンドに関するパラメータ情報を、設定または取得します。	スカラー値のベクトルとして格納されているパラメータは、サポートされていません。 BLOB パラメータのサポートはありません。
IConvertType		サポートされています。
IDBAsynchNotify IDBAsynchStatus	非同期処理。 データソース初期化の非同期処理、ローセットの移植などにおいて、クライアントにイベントを通知します。	サポートされていません。
IDBCreateCommand	セッションからコマンドを作成します。	サポートされています。
IDBCreateSession	データソースオブジェクトからセッションを作成します。	サポートされています。
IDBDataSourceAdmin	データソースオブジェクトを作成/破壊/修正します。このオブジェクトはクライアントによって使用される COM オブジェクトです。このインタフェースは、データストア (データベース) の管理には使用されません。	サポートされていません。

インタフェース	内容	制限事項
IDBInfo	このプロバイダにとってユニークなキーワードについての情報を検索します (非標準の SQL キーワードを検索します)。 また、テキスト一致クエリで使用されるリテラルや特定の文字、その他のリテラル情報についての情報を検索します。	サポートされています。
IDBInitialize	データソースオブジェクトと列挙子を初期化します。	サポートされています。
IDBProperties	データソースオブジェクトまたは列挙子のプロパティを管理します。	サポートされています。
IDBSchemaRowset	標準フォーム (ローセット) にあるシステムテーブルの情報を取得します。	サポートされています。
IErrorInfo IErrorLookup IErrorRecords	Microsoft ActiveX エラーオブジェクトサポート。	サポートされています。
IGetDataSource	インタフェースポインタを、セッションのデータソースオブジェクトに返します。	サポートされています。
IIndexDefinition	データストアにインデックスを作成または削除します。	サポートされていません。
IMultipleResults	コマンドから複数の結果 (ローセットやローカウント) を取り出します。	サポートされています。
IOpenRowset	名前でデータベーステーブルにアクセスする非 SQL 的な方法。	サポートされています。 名前でテーブルを開くのはサポートされていますが、GUID で開くのはサポートされていません。
IParentRowset	区分化/階層ローセットにアクセスします。	サポートされていません。
IRowset	ローセットにアクセスします。	サポートされています。
IRowsetChange	ローセットデータへの変更を許し、変更をデータストアに反映させます。 BLOB に対する InsertRow/SetData は実装されていません。	サポートされています。
IRowsetChapterMember	区分化/階層ローセットにアクセスします。	サポートされていません。
IRowsetCurrentIndex	ローセットのインデックスを動的に変更します。	サポートされていません。
IRowsetFind	指定された値と一致するローを、ローセットの中から検索します。	サポートされていません。
IRowsetIdentity	ローのハンドルを比較します。	サポートされていません。
IRowsetIndex	データベースインデックスにアクセスします。	サポートされていません。

インタフェース	内容	制限事項
IRowsetInfo	ローセットプロパティについての情報を検索する、または、ローセットを作成したオブジェクトを検索します。	サポートされています。
IRowsetLocate	ブックマークを使用して、ローセットのローを検索します。	サポートされています。
IRowsetNotify	ローセットのイベントに COM コールバックインタフェースを提供します。	サポートされています。
IRowsetRefresh	トランザクションで参照可能な最後のデータの値を取得します。	サポートされていません。
IRowsetResynch	以前の OLE DB 1.x のインタフェースで、IRowsetRefresh に変わりました。	サポートされていません。
IRowsetScroll	ローセットをスクロールして、ローデータをフェッチします。	サポートされていません。
IRowsetUpdate	Update が呼ばれるまで、ローセットデータの変更を遅らせます。	サポートされています。
IRowsetView	既存のローセットにビューを使用します。	サポートされていません。
ISequentialStream	BLOB カラムを取り出します。	読み出しのみのサポートです。 このインタフェースを使用した SetData はサポートされていません。
ISessionProperties	セッションプロパティ情報を取得します。	サポートされています。
ISourcesRowset	データソースオブジェクトと列挙子のローセットを取得します。	サポートされています。
ISQLErrorInfo ISupportErrorInfo	Microsoft ActiveX エラーオブジェクトサポート。	サポートされています。
ITableDefinition ITableDefinitionWithConstraints	制約を使用して、テーブルを作成、削除、変更します。	サポートされています。
ITransaction	トランザクションをコミットまたはアボートします。	すべてのフラグがサポートされているわけではありません。
ITransactionJoin	分散トランザクションをサポートします。	すべてのフラグがサポートされているわけではありません。
ITransactionLocal	セッションでトランザクションを処理します。 すべてのフラグがサポートされているわけではありません。	サポートされています。
ITransactionOptions	トランザクションでオプションを取得または設定します。	サポートされていません。
IViewChapter	既存のローセットでビューを使用します。特に、後処理フィルタやローのソートを適用するために利用されます。	サポートされていません。

インタフェース	内容	制限事項
IViewFilter	ローセットの内容を、一連の条件と一致するローに制限します。	サポートされていません。
IViewRowset	ローセットを開くときに、ローセットの内容を、一連の条件と一致するローに制限します。	サポートされていません。
IViewSort	ソート順をビューに適用します。	サポートされていません。

1.4.8 OLE DB プロバイダの登録

ソフトウェアに付属のインストーラを使用して SAOLEDB プロバイダがインストールされるときに、SAOLEDB プロバイダはそれ自身を登録します。

この登録プロセスには、レジストリの COM セクションにレジストリエントリを作成することも含まれます。このエントリによって、ADO は SAOLEDB プロバイダが呼び出されたときに DLL を見つけることができます。DLL のロケーションを変更した場合は、その情報を再度登録する必要があります。

例

プロバイダがインストールされているディレクトリから次のコマンドを実行すると、OLE DB プロバイダが登録されます。

```
regsvr32 dboledb17.dll
regsvr32 dboledba17.dll
```

64 ビット Windows を使用している場合、上記のコマンドで 64 ビットプロバイダが登録されます。32 ビット OLE DB プロバイダを登録する場合は、次のコマンドを使用できます。

```
c:¥Windows¥SysWOW64¥regsvr32 dboledb17.dll
c:¥Windows¥SysWOW64¥regsvr32 dboledba17.dll
```

1.5 ODBC サポート

ODBC (Open Database Connectivity) は、Microsoft が開発した標準 CLI (コールレベルインタフェース) です。SQL Access Group CLI 仕様に基づいています。

ODBC アプリケーションは、ODBC ドライバを提供するあらゆるデータソースに使用できます。ODBC ドライバを持つ他のデータソースにアプリケーションを移植できるようにしたい場合は、プログラミングインタフェースとして ODBC を使用することをお奨めします。

ODBC は低レベルインタフェースです。データベースサーバのほとんどすべての機能をこのインタフェースで使用できます。Windows オペレーティングシステムでは、ODBC を DLL として使用できます。UNIX 用には共有オブジェクトライブラリとして提供されます。

このセクションの内容:

ODBC アプリケーションの開発要件 [150 ページ]
次の図に示すように、各種の開発ツールとプログラミング言語を使用してアプリケーションを開発でき、ODBC API を使用してデータベースサーバにアクセスできます。

ODBC アプリケーションの開発 [152 ページ]
ODBC 関数を呼び出す C/C++ ソースファイルには、プラットフォーム固有の ODBC ヘッダファイルが必要です。各プラットフォーム固有のヘッダファイルは、ODBC のメインヘッダファイル `odbc.h` を含みます。このヘッダファイルには、ODBC プログラムの作成に必要なすべての関数、データ型、定数の定義が含まれています。

サンプル ODBC プログラム [157 ページ]
ソフトウェアには、サンプル ODBC プログラムが含まれています。

ODBC ハンドル [159 ページ]
ODBC アプリケーションは、小さいハンドルセットを使用して、ODBC コンテキスト、データベース接続、SQL 文を追跡します。ハンドルはポインタ型で、64 ビットのアプリケーションで 64 ビット値、32 ビットのアプリケーションで 32 ビット値です。

ODBC 接続関数 [161 ページ]
ODBC には 3 種類の接続関数があります。

ODBC によって変更されるサーバオプション [165 ページ]
ODBC ドライバは、データベースサーバへの接続時に一部のテンポラリサーバオプションを設定します。

SQLSetConnectAttr 拡張接続属性 [166 ページ]
ODBC ドライバは、拡張された一部の接続属性をサポートしています。

Windows DllMain 関数の考慮事項 [168 ページ]
Windows ダイナミックリンクライブラリ内の DllMain 関数から直接的または間接的に ODBC 関数を呼び出さないでください。DllMain のエントリポイント関数は、簡単な初期化および終了処理のみを実行するためのものです。SQLFreeHandle、SQLFreeConnect、SQLFreeEnv などの ODBC 関数を呼び出した場合、デッドロックや循環依存が発生する可能性があります。

SQL 文の実行方法 [169 ページ]
ODBC には、SQL 文を実行するための関数がいくつか含まれます。

64 ビット ODBC での考慮事項 [174 ページ]
SQLBindCol、SQLBindParameter、SQLGetData などの ODBC 関数を使用する場合、一部のパラメータは SQLLEN や SQLULEN として関数プロトタイプに型指定されます。

データアラインメントの要件 [177 ページ]
SQLBindCol、SQLBindParameter、または SQLGetData を使用する場合、カラムまたはパラメータには C データ型が指定されます。

ODBC アプリケーションの結果セット [178 ページ]
ODBC アプリケーションはカーソルを使用して、結果セットを操作および更新します。ソフトウェアは、多種多様なカーソルとカーソル処理をサポートしています。

ストアドプロシージャの考慮事項 [184 ページ]
ODBC アプリケーションからストアドプロシージャを作成して呼び出し、その結果を処理することができます。

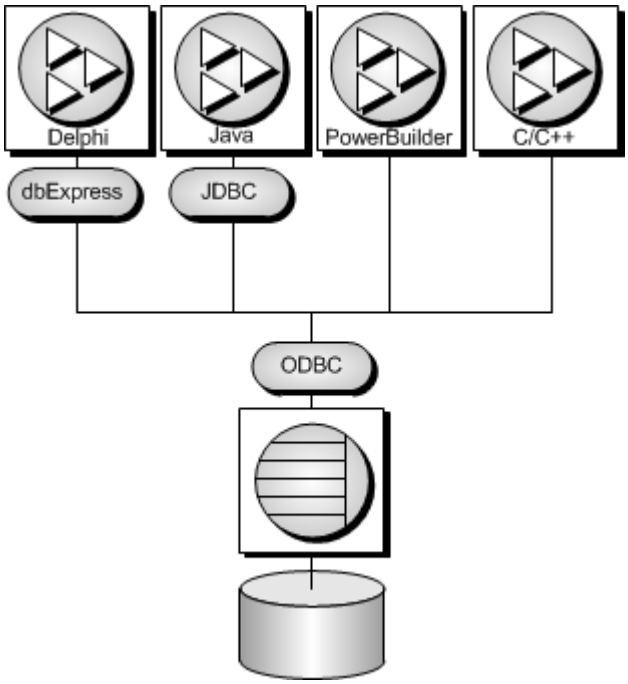
ODBC エスケープ構文 [185 ページ]
ODBC エスケープ構文は、任意の ODBC アプリケーションで使用できます。エスケープ構文を使用して、使用しているデータベース管理システムとは関係なく、共通の関数セットを呼び出すことができます。

ODBC のエラー処理 [188 ページ]

ODBC のエラーは、各 ODBC 関数呼び出しからの戻り値と、SQLError 関数または SQLGetDiagRec 関数を使用してレポートされます。

1.5.1 ODBC アプリケーションの開発要件

次の図に示すように、各種の開発ツールとプログラミング言語を使用してアプリケーションを開発でき、ODBC API を使用してデータベースサーバにアクセスできます。



データベースサーバ用の ODBC アプリケーションを作成するには、次の環境が必要です。

- 使用している環境に合ったプログラムを作成できる C/C++ コンパイラ。
- Microsoft ODBC Software Development Kit。このキットは、Microsoft Developer Network から入手でき、マニュアルと ODBC アプリケーションをテストする補足ツールが入っています。

> **注記**
>
> すでに ODBC サポート機能がある一部のアプリケーション開発ツールには、ODBC インタフェースを意識する必要のない、独自のプログラミングインタフェースが用意されています。このマニュアルには、これらのツールの使用方法についての説明はありません。

ODBC ドライバマネージャ

Microsoft Windows には ODBC ドライバマネージャが同梱されています。Linux、UNIX、Mac OS X では、データベースサーバソフトウェアにドライバマネージャが含まれています。

このセクションの内容:

ODBC 準拠 [151 ページ]
ODBC ドライバは、Microsoft Data Access Kit 2.7 の一部として提供されている ODBC 3.5 をサポートしています。

1.5.1.1 ODBC 準拠

ODBC ドライバは、Microsoft Data Access Kit 2.7 の一部として提供されている ODBC 3.5 をサポートしています。

ODBC のサポートレベル

ODBC の機能は、準拠のレベルによって異なります。機能はコア、レベル 1、またはレベル 2 のいずれかです。レベル 2 は ODBC を完全にサポートします。

ODBC ドライバでサポートされている機能

ODBC 3.5 仕様は次のようにサポートされています。

コア準拠

すべてのコアレベル機能がサポートされます。

レベル 1 準拠

ODBC 関数の非同期実行を除くすべてのレベル 1 機能がサポートされます。

複数のスレッドによる単一接続の共有がサポートされます。各スレッドからの要求は、データベースサーバによって直列化されます。

レベル 2 準拠

下記を除くすべてのレベル 2 機能がサポートされます。

- 3 語で構成されるビュー名とテーブル名。これはデータベースサーバには当てはまりません。
- 特定の独立した文についての ODBC 関数の非同期実行。
- ログイン要求と SQL クエリをタイムアウトする機能。

関連情報

ODBC Programmer's Reference

1.5.2 ODBC アプリケーションの開発

ODBC 関数を呼び出す C/C++ ソースファイルには、プラットフォーム固有の ODBC ヘッダファイルが必要です。各プラットフォーム固有のヘッダファイルは、ODBC のメインヘッダファイル `odbc.h` を含みます。このヘッダファイルには、ODBC プログラムの作成に必要なすべての関数、データ型、定数の定義が含まれています。

C/C++ ソースファイルに ODBC ヘッダファイルをインクルードするには、次のタスクを実行します。

1. ソースファイルに、該当するプラットフォーム固有のヘッダファイルを参照するインクルード行を追加します。使用する行は次のとおりです。

オペレーティングシステム	インクルード行
Windows	#include "ntodbc.h"
Unix	#include "unixodbc.h"

2. ヘッダファイルがあるディレクトリを、コンパイラのインクルードパスに追加します。
   プラットフォーム固有のヘッダファイルと `odbc.h` は、どちらもデータベースサーバソフトウェアのディレクトリの `SDK ¥Include` サブディレクトリにインストールされます。
3. UNIX 用の ODBC アプリケーションを構築するときは、正しいデータアラインメントとサイズを取得するために、32 ビットのアプリケーションの場合はマクロ "UNIX"、64 ビットのアプリケーションの場合は "UNIX64" を定義する必要があることがあります。ただし、次に示すサポートされるコンパイラのいずれかを使用している場合は、マクロの定義は不要です。
   - サポートされるプラットフォームにインストールされている GNU C/C++ コンパイラ
   - Linux 用の Intel C/C++ コンパイラ (icc)
   - Linux または Solaris 用の SunPro C/C++ コンパイラ
   - AIX 用の VisualAge C/C++ コンパイラ
   - HP-UX 用の C/C++ コンパイラ (cc/aCC)

ソースコードが書き込まれると、アプリケーションをコンパイルしてリンクできます。

このセクションの内容:

Windows での ODBC アプリケーション [153 ページ]
アプリケーションをリンクする場合は、ODBC の関数にアクセスできるように、適切なインポートライブラリファイルにリンクします。

UNIX での ODBC アプリケーション [153 ページ]
UNIX 用の ODBC ドライバマネージャはデータベースサーバソフトウェアに同梱されており、サードパーティ製のドライバマネージャも利用できます。下記の情報では、ODBC ドライバマネージャを使用しない ODBC アプリケーションの構築方法について説明します。

UNIX 用 SQL Anywhere ODBC ドライバマネージャ [155 ページ]
UNIX 用 ODBC ドライバマネージャは、データベースサーバソフトウェアに付属しています。

unixODBC ドライバマネージャ [155 ページ]
バージョン 2.2.14 より前の unixODBC では、64 ビット ODBC 仕様の一部が Microsoft の規定とは異なって実装されています。この違いにより、unixODBC ドライバマネージャを 64 ビット ODBC ドライバで使用すると問題が生じます。

UNIX 用 UTF-32ODBC ドライバマネージャ [156 ページ]

ワイド呼び出しをサポートする ODBC ドライバは 16 ビットの SQLWCHAR 用に構築されているため、SQLWCHAR を 32 ビット (UTF-32) の数量として定義する ODBC ドライバマネージャのバージョンは、ODBC ドライバで使用できません。

1.5.2.1 Windows での ODBC アプリケーション

アプリケーションをリンクする場合は、ODBC の関数にアクセスできるように、適切なインポートライブラリファイルにリンクします。

インポートライブラリでは、ODBC ドライバマネージャ `odbc32.dll` のエントリポイントが定義されます。このドライバマネージャは、ODBC ドライバ `dbodbc17.dll` をロードします。

通常、インポートライブラリは Microsoft プラットフォーム SDK の `Lib` ディレクトリ構造下に格納されています。

オペレーティングシステム	インポートライブラリ
Windows (32 ビット)	`Lib¥odbc32.lib`
Windows (64 ビット)	`Lib¥x64¥odbc32.lib`

例

次のコマンドは、プラットフォーム固有のインポートライブラリがあるディレクトリを、LIB 環境変数内のライブラリディレクトリのリストに追加する方法を示します。

```
set LIB=%LIB%;c:¥mssdk¥v7.0¥lib
```

次のコマンドは、Microsoft のコンパイルおよびリンクツールを使用して、`odbc.c` に保存されたアプリケーションをコンパイルしてリンクする方法を示します。

```
cl odbc.c /Ic:¥sa17¥SDK¥Include odbc32.lib
```

1.5.2.2 UNIX での ODBC アプリケーション

UNIX 用の ODBC ドライバマネージャはデータベースサーバソフトウェアに同梱されており、サードパーティ製のドライバマネージャも利用できます。下記の情報では、ODBC ドライバマネージャを使用しない ODBC アプリケーションの構築方法について説明します。

ODBC ドライバ

ODBC ドライバは、共有オブジェクトまたは共有ライブラリです。シングルスレッドアプリケーションとマルチスレッドアプリケーションには、ODBC ドライバの別々のバージョンが用意されています。汎用の ODBC ドライバは、使用中のスレッドモデルを検出し、シングルスレッドまたはマルチスレッドのライブラリを直接呼び出します。

ODBC ドライバのファイルは、次のとおりです。

オペレーティングシステム	スレッドモデル	ODBC ドライバ
(Mac OS X と HP-UX 以外のすべての UNIX)	汎用	libdbodbc*17*.so (libdbodbc*17*.so.1)
(Mac OS X と HP-UX 以外のすべての UNIX)	シングルスレッド	libdbodbc*17*_n.so (libdbodbc*17*_n.so.1)
(Mac OS X と HP-UX 以外のすべての UNIX)	マルチスレッド	libdbodbc*17*_r.so (libdbodbc*17*_r.so.1)
HP-UX	汎用	libdbodbc*17*.sl (libdbodbc*17*.sl.1)
HP-UX	シングルスレッド	libdbodbc*17*_n.sl (libdbodbc*17*_n.sl.1)
HP-UX	マルチスレッド	libdbodbc*17*_r.sl (libdbodbc*17*_r.sl.1)
Mac OS X	汎用	libdbodbc*17*.dylib
Mac OS X	シングルスレッド	libdbodbc*17*_n.dylib
Mac OS X	マルチスレッド	libdbodbc*17*_r.dylib

ライブラリは、バージョン番号 (カッコ内に表示) を使用して共有ライブラリへのシンボリックリンクとしてインストールされます。

さらに、Mac OS X では次のバンドルも使用できます。

オペレーティングシステム	スレッドモデル	ODBC ドライバ
Mac OS X	シングルスレッド	dbodbc*17*.bundle
Mac OS X	マルチスレッド	dbodbc*17*_r.bundle

UNIX で ODBC アプリケーションをリンクする場合、アプリケーションを汎用 ODBC ドライバ libdbodbc*17* に対してリンクします。アプリケーションを配備するときに、適切な (またはすべての) ODBC ドライババージョン (非スレッドまたはスレッド) がユーザのライブラリパスに含まれていることを確認します。

データソース情報

ODBC ドライバマネージャの存在が検出されなかった場合、ODBC ドライバはデータソース情報にシステム情報ファイルを使用します。

1.5.2.3 UNIX 用 SQL Anywhere ODBC ドライバマネージャ

UNIX 用 ODBC ドライバマネージャは、データベースサーバソフトウェアに付属しています。

`libdbodm17` 共有オブジェクトは、サポートされているすべての UNIX プラットフォームで ODBC ドライバマネージャとして使用できます。ドライバマネージャは、バージョン 3.0 以降の ODBC ドライバのロードに使用できます。ドライバマネージャは ODBC 1.0/2.0 呼び出しと ODBC 3.x 呼び出し間のマッピングを実行しません。したがって、ドライバマネージャを使用しているアプリケーションでは、バージョン 3.0 以降の ODBC 機能セットを使用するように制限してください。ドライバマネージャは、スレッドアプリケーションと非スレッドアプリケーションのどちらでも使用できます。

ドライバマネージャでは、指定された接続に対する ODBC 呼び出しのトレーシングを実行できます。トレーシング機能を有効にするには、TraceLevel と TraceLog ディレクティブを使用します。この 2 つのディレクティブは、接続文字列 (SQLDriverConnect を使用している場合) の一部として、または DSN エントリ内に指定できます。TraceLog ディレクティブは、接続のためのトレース出力を含むトレースログファイルを識別します。TraceLevel ディレクティブは、目的のトレーシング情報の量を制御します。トレースのレベルは次のとおりです。

NONE

トレーシング情報を表示しません。

MINIMAL

ルーチン名とパラメータを出力に含めます。

LOW

ルーチン名とパラメータのほかに戻り値を出力に含めます。

MEDIUM

ルーチン名、パラメータ、戻り値のほかに実行日時を出力に含めます。

HIGH

ルーチン名、パラメータ、戻り値、実行日時のほかにパラメータタイプを出力に含めます。

サードパーティの UNIX 用 ODBC ドライバマネージャも使用できます。詳細については、ドライバマネージャに付属のマニュアルを参照してください。

関連情報

unixODBC ドライバマネージャ [155 ページ]
UNIX 用 UTF-32ODBC ドライバマネージャ [156 ページ]

1.5.2.4 unixODBC ドライバマネージャ

バージョン 2.2.14 より前の unixODBC では、64 ビット ODBC 仕様の一部が Microsoft の規定とは異なって実装されています。この違いにより、unixODBC ドライバマネージャを 64 ビット ODBC ドライバで使用すると問題が生じます。

このような問題を回避するためには、両者の違いについて認識します。相違点の 1 つとして挙げられるのが、SQLLEN と SQLULEN の定義です。SQLLEN と SQLULEN は、Microsoft 64 ビット ODBC 仕様では 64 ビット型であり、64 ビット

ODBC ドライバでも 64 ビット数として処理されることを想定しています。unixODBC の一部の実装では、これらの 2 つを 32 ビット数として定義しているため、64 ビット ODBC ドライバとインタフェースする際に問題が生じます。

64 ビットのプラットフォームで問題を回避するには、次の 3 つの処理が必要になります。

1. `sql.h` や `sqlext.h` などの unixODBC ヘッダをインクルードするのではなく、`unixodbc.h` ヘッダファイルをインクルードします。これにより、SQLLEN および SQLULEN は正しく定義されます。unixODBC 2.2.14 以降のバージョンのヘッダファイルでは、この問題が修正されています。
2. すべてのパラメータで正しい型を使用していることを確認する必要があります。正しいヘッダファイルや C/C++ コンパイラの強力な型チェックを使用すると便利です。また、ODBC ドライバがポインタを介して間接的に設定したすべての変数についても、正しい型を使用していることを確認する必要があります。
3. リリース 2.2.14 より前の unixODBC ドライバマネージャは使用せずに、ODBC ドライバに直接リンクしてください。たとえば、libodbc 共有オブジェクトを ODBC ドライバ共有オブジェクトにリンクします。

   ```
   libodbc.so.1 -> libdbodbc17_r.so.1
   ```

   または、利用可能なプラットフォームのデータベースサーバソフトウェアに含まれているドライバマネージャを使用することもできます。

関連情報

UNIX 用 SQL Anywhere ODBC ドライバマネージャ [155 ページ]
UNIX での ODBC アプリケーション [153 ページ]
64 ビット ODBC での考慮事項 [174 ページ]

1.5.2.5 UNIX 用 UTF-32ODBC ドライバマネージャ

ワイド呼び出しをサポートする ODBC ドライバは 16 ビットの SQLWCHAR 用に構築されているため、SQLWCHAR を 32 ビット (UTF-32) の数量として定義する ODBC ドライバマネージャのバージョンは、ODBC ドライバで使用できません。

このような場合に備えて ANSI 専用バージョンの ODBC ドライバが用意されています。このバージョンの ODBC ドライバでは、ワイド呼び出しインタフェース (SQLConnectW など) はサポートされていません。

ドライバの共有オブジェクト名は `libdbodbcansi17_r` です。ドライバのスレッド変形のみが提供されています。Mac OS X では、dylib に加え、バンドル形式 (`dbodbcansi17_r.bundle`) のドライバも使用できます。Real Basic などの特定のフレームワークは dylib で動作しないため、バンドルが必要になります。

通常の ODBC ドライバでは、SQLWCHAR 文字列が UTF-16 文字列として処理されます。このドライバは、iODBC などの、SQLWCHAR 文字列を UTF-32 文字列として処理する一部の ODBC ドライバマネージャでは使用できません。Unicode 対応のドライバを扱う場合、これらのドライバマネージャでは、アプリケーションからのナロー呼び出しがドライバへのワイド呼び出しに変換されます。ANSI 専用ドライバではこの動作が回避されます。このため、アプリケーションがワイド呼び出しを行わないかぎり、ANSI 専用ドライバをこのようなドライバマネージャで使用することができます。iODBC を介したワイド呼び出しや、同様のセマンティックを持つ他のドライバマネージャは、引き続きサポートされません。

1.5.3 サンプル ODBC プログラム

ソフトウェアには、サンプル ODBC プログラムが含まれています。

サンプルは *%SQLANYSAMP17%*`¥SQLAnywhere` サブディレクトリ (Windows) または *$SQLANYSAMP17*`/sqlanywhere` サブディレクトリ (UNIX) にあります。

ディレクトリ内の ODBC の 4 文字で始まるサンプルプログラムは、データベースへの接続や文の実行など、簡単な ODBC 作業をそれぞれ示します。完全なサンプル ODBC プログラムは、ファイル *%SQLANYSAMP17%*`¥SQLAnywhere¥C¥odbc.c` (Windows) または *$SQLANYSAMP17*`/sqlanywhere/c/odbc.c` (UNIX) にあります。このプログラムの動作は、同じディレクトリにある Embedded SQL 動的 CURSOR のサンプルプログラムと同じです。

このセクションの内容:

Windows 用の ODBC サンプルプログラムの構築 [157 ページ]
ODBC サンプルプログラムを構築して実行し、データベースへの接続や文の実行などの ODBC タスクを実行する仕組みを確認します。

UNIX 用の ODBC サンプルプログラムの構築 [158 ページ]
ODBC サンプルプログラムを構築して実行し、データベースへの接続や文の実行などの ODBC タスクを実行する仕組みを確認します。

ODBC サンプルプログラムの実行 [159 ページ]
適切なプラットフォームでファイルを実行し、サンプル ODBC プログラムをロードできます。

関連情報

サンプル Embedded SQL プログラム [263 ページ]

1.5.3.1 Windows 用の ODBC サンプルプログラムの構築

ODBC サンプルプログラムを構築して実行し、データベースへの接続や文の実行などの ODBC タスクを実行する仕組みを確認します。

前提条件

最新バージョンの Microsoft Visual Studio が必要です。

Microsoft Visual Studio による x86/x64 プラットフォームのビルドでは、コンパイルとリンクに適した環境を設定する必要があります。これは通常、Microsoft Visual Studio の `vcvars32.bat` または `vcvars64.bat` (Microsoft Visual Studio の旧バージョンでは `vcvarsamd64.bat`) を使用して行います。

コンテキスト

%SQLANYSAMP17%¥SQLAnywhere¥C ディレクトリにあるバッチファイルを使用して、すべてのサンプルアプリケーションをコンパイルしてリンクできます。

手順

1. コマンドプロンプトを開き、*%SQLANYSAMP17%*¥SQLAnywhere¥C ディレクトリに移動します。
2. build.bat バッチファイルを実行します。

   ビルドのエラーが発生した場合は、ターゲットプラットフォーム (x86 または x64) を build.bat の引数として指定します。次に例を示します。

   ```
   build x64
   ```

結果

サンプル ODBC プログラムが構築されます。

1.5.3.2 UNIX 用の ODBC サンプルプログラムの構築

ODBC サンプルプログラムを構築して実行し、データベースへの接続や文の実行などの ODBC タスクを実行する仕組みを確認します。

コンテキスト

$SQLANYSAMP17/sqlanywhere/c ディレクトリにあるシェルスクリプトを使用して、すべてのサンプルアプリケーションをコンパイルしてリンクできます。

手順

1. コマンドシェルを開き、*$SQLANYSAMP17*/sqlanywhere/c ディレクトリに移動します。

   コマンドシェルを開き、*$SQLANYSAMP17*/samples/sqlanywhere/c ディレクトリに移動します。
2. build.sh シェルスクリプトを実行します。

結果

サンプル ODBC プログラムが構築されます。

1.5.3.3 ODBC サンプルプログラムの実行

適切なプラットフォームでファイルを実行し、サンプル ODBC プログラムをロードできます。

- 32 ビットの Windows では、*%SQLANYSAMP17%*¥SQLAnywhere¥C¥odbcwin.exe を実行します。
- 64 ビットの Windows では、*%SQLANYSAMP17%*¥SQLAnywhere¥C¥odbcx64.exe を実行します。
- UNIX では、*$SQLANYSAMP17*/sqlanywhere/c/odbc を実行します。

ファイルを実行後、サンプルデータベースのいずれかのテーブルを選択します。たとえば、**Customers** または **Employees** を入力します。

1.5.4 ODBC ハンドル

ODBC アプリケーションは、小さいハンドルセットを使用して、ODBC コンテキスト、データベース接続、SQL 文を追跡します。ハンドルはポインタ型で、64 ビットのアプリケーションで 64 ビット値、32 ビットのアプリケーションで 32 ビット値です。

次のハンドルは、ODBC アプリケーションで使用されます。

環境

環境ハンドルは、データにアクセスするグローバルコンテキストを提供します。すべての ODBC アプリケーションは、起動時に環境ハンドルを 1 つだけ割り付け、アプリケーションの終了時にそれを解放します。

次のコードは、環境ハンドルを割り付ける方法を示します。

```
SQLRETURN rc;
SQLHENV env;
rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env );
```

接続

接続は、アプリケーションとデータソースの間のリンクです。アプリケーションは、その環境に対応する接続を複数確立できます。接続ハンドルを割り付けても、接続は確立されません。最初に接続ハンドルを割り付けてから、接続の確立に使用します。

次のコードは、接続ハンドルを割り付ける方法を示します。

```
SQLRETURN rc;
SQLHDBC  dbc;
rc = SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
```

文

ステートメントハンドルを使用すると、SQL 文を実行し、パラメータや結果セットなど、ステートメントハンドルに関連する情報を設定、処理できます。各接続には、複数のステートメントハンドルを関連付けることができます。ステートメントハンドルは、クエリ実行 (フェッチなどのカーソル操作) でも、非クエリ実行 (INSERT、UPDATE、DELETE 文など) でも使用されます。

次のコードは、ステートメントハンドルを割り付ける方法を示します。

```
SQLRETURN rc;
SQLHSTMT stmt;
rc = SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
```

このセクションの内容:

ODBC ハンドルを割り付ける方法 [160 ページ]
ODBC は、ハンドルを使用することでアプリケーションで参照される 4 種類のオブジェクト (環境、接続、文、記述子) を定義します。

ODBC のサンプル [161 ページ]
%SQLANYSAMP17%¥SQLAnywhere¥ODBCConnect¥odbcconnect.cpp にある次の簡単な ODBC プログラムは、サンプルデータベースに接続し、直後に切断します。

1.5.4.1 ODBC ハンドルを割り付ける方法

ODBC は、ハンドルを使用することでアプリケーションで参照される 4 種類のオブジェクト (環境、接続、文、記述子) を定義します。

ODBC プログラムで利用可能なハンドルの型は、次のとおりです。

項目	ハンドルの型
環境	SQLHENV
接続	SQLHDBC
文	SQLHSTMT
記述子	SQLHDESC

ODBC ハンドルを使用するには、次のタスクを実行します。

1. SQLAllocHandle 関数を呼び出します。
2. 後続の関数呼び出しでハンドルを使用します。
3. SQLFreeHandle を使用してオブジェクトを解放します。

SQLAllocHandle は、次のパラメータを取ります。

- 割り付ける項目の型を示す識別子
- 親項目のハンドル
- 割り付けるハンドルのロケーションへのポインタ

SQLFreeHandle は、次のパラメータを取ります。

- 解放する項目の型を示す識別子
- 解放する項目のハンドル

例

次のコードフラグメントは、環境ハンドルを割り付け、解放します。

```
SQLRETURN rc;
SQLHENV env;
rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env );
if( rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO )
{
    .
    .
    .
}
SQLFreeHandle( SQL_HANDLE_ENV, env );
```

関連情報

ODBC のエラー処理 [188 ページ]
SQLAllocHandle 関数と SQLFreeHandle 関数

1.5.4.2 ODBC のサンプル

%SQLANYSAMP17%¥SQLAnywhere¥ODBCConnect¥odbcconnect.cpp にある次の簡単な ODBC プログラムは、サンプルデータベースに接続し、直後に切断します。

この例では、データベースサーバに接続するための環境の設定に必要な手順のほか、サーバから切断してリソースを解放するのに必要な手順も示しています。

1.5.5 ODBC 接続関数

ODBC には 3 種類の接続関数があります。

どの接続関数を使用するかは、アプリケーションの配備方法と使用方法によって決まります。

SQLConnect

最も簡単な接続関数です。

SQLConnect は、データソース名と、オプションでユーザ ID とパスワードをパラメータに取ります。データソース名をアプリケーションにハードコードする場合は、SQLConnect を使用します。

SQLDriverConnect

接続文字列を使用してデータソースに接続します。

SQLDriverConnect を使用すると、アプリケーションはデータソース定義の外部にある接続情報を使用できます。また、ODBC ドライバに対して接続情報を確認するように要求できます。

データソースを指定しないで接続することもできます。代わりに、ODBC ドライバ名が指定されます。次の例では、すでに実行されているサーバとデータベースに接続します。

```
SQLSMALLINT cso;
SQLCHAR     scso[2048];
SQLDriverConnect( hdbc, NULL,
   "Driver=SQL Anywhere 17;UID=DBA;PWD=passwd", SQL_NTS,
   scso, sizeof(scso)-1,
   &cso, SQL_DRIVER_NOPROMPT );
```

SQLBrowseConnect

SQLDriverConnect と同様に、接続文字列を使用してデータソースに接続します。

SQLBrowseConnect を使用すると、アプリケーションは独自のウィンドウを構築して接続情報を要求するプロンプトを表示したり、特定のドライバで使用されるデータソースを参照したりできます。

このセクションの内容:

ODBC 接続の確立 [162 ページ]
アプリケーションに ODBC 接続を確立し、データベース操作を行います。

関連情報

SQLConnect 関数、SQLDriverConnect 関数、SQLBrowseConnect 関数

1.5.5.1 ODBC 接続の確立

アプリケーションに ODBC 接続を確立し、データベース操作を行います。

コンテキスト

完全なサンプルは、*%SQLANYSAMP17%*¥SQLAnywhere¥ODBCConnect¥odbcconnect.cpp にあります。

手順

1. ODBC 環境を割り付けます。次に例を示します。

```
SQLRETURN rc;
SQLHENV   env;
rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env );
```

2. ODBC のバージョンを宣言します。次に例を示します。

```
rc = SQLSetEnvAttr( env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0 );
```

アプリケーションが ODBC バージョン 3 に従うことを宣言すると、SQLSTATE 値およびその他のバージョン依存の機能が適切な動作に設定されます。

3. ODBC 接続ハンドルを割り付けます。次に例を示します。

```
rc = SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
```

4. 接続前に必要な接続属性を設定します。

接続属性には、接続を確立する前または後に必ず設定するものと、確立前に設定しても後に設定してもかまわないものがあります。SQL_AUTOCOMMIT 属性は、接続の確立前にでも後にでも設定できる属性です。

```
rc = SQLSetConnectAttr( dbc, SQL_AUTOCOMMIT, (SQLPOINTER)SQL_AUTOCOMMIT_OFF, 0 );
```

デフォルトでは、ODBC はオートコミットモードで動作します。このモードは、SQL_AUTOCOMMIT を false に設定してオフにすることができます。

5. 必要な場合は、データソースまたは接続文字列をアセンブルします。

アプリケーションによっては、データソースや接続文字列をハードコードしたり、柔軟性を高めるために外部に格納したりできます。

6. ODBC 接続を確立します。次に例を示します。

```
if (rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO)
{
   printf( "dbc allocated¥n" );
   rc = SQLConnect( dbc,
      (SQLCHAR *) "SQL Anywhere 17 Demo", SQL_NTS,
      (SQLCHAR *) my_userid, SQL_NTS,
      (SQLCHAR *) my_password, SQL_NTS );
   if (rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO)
   {
       // Successfully connected.
```

ODBC に渡される各文字列には、固有の長さがあります。長さがわからない場合は、ASCII 文字列の場合は NULL 文字 (¥0)、ワイド文字列の場合は NULL ワイド文字列 (¥0¥0) で終端をマークした **NULL** で終了された文字列であることを示す SQL_NTS を渡すことができます。

結果

アプリケーションをビルドして実行すると、ODBC 接続が確立されます。

このセクションの内容:

接続属性を設定する方法 [164 ページ]
SQLSetConnectAttr 関数を使用して、接続の詳細を制御します。たとえば、次の文は ODBC のオートコミットを無効にします。

接続属性を取得する方法 [164 ページ]
SQLGetConnectAttr 関数を使用して、接続の詳細を取得します。たとえば、次の文は接続の状態を返します。

ODBC アプリケーションでのスレッドと接続 [165 ページ]
マルチスレッド ODBC アプリケーションを開発できます。各スレッドに対して別個の接続を使用します。

ODBC 接続障害 [165 ページ]
ODBC ドライバで必要なすべてのファイルをデプロイしなかった場合、データベースへの接続時に次のエラーが発生する可能性があります。

1.5.5.1.1 接続属性を設定する方法

SQLSetConnectAttr 関数を使用して、接続の詳細を制御します。たとえば、次の文は ODBC のオートコミットを無効にします。

```
rc = SQLSetConnectAttr( dbc, SQL_ATTR_AUTOCOMMIT, (SQLPOINTER)SQL_AUTOCOMMIT_OFF,
0 );
```

接続のさまざまな側面を、接続パラメータを介して制御できます。

関連情報

SQLSetConnectAttr 関数

1.5.5.1.2 接続属性を取得する方法

SQLGetConnectAttr 関数を使用して、接続の詳細を取得します。たとえば、次の文は接続の状態を返します。

```
rc = SQLGetConnectAttr( dbc, SQL_ATTR_CONNECTION_DEAD,
   (SQLPOINTER)&closed, SQL_IS_INTEGER, 0 );
```

SQLGetConnectAttr 関数を使用して SQL_ATTR_CONNECTION_DEAD 属性を取得すると、接続が切断されていた場合、切断後にサーバに要求が送信されていなくても、値 SQL_CD_TRUE が返されます。接続が切断したかどうかの確認は、サーバに要求を送信しないで行われ、切断された接続は数秒以内に検出されます。接続が切断されるのには、アイドルタイムアウトなどの複数の理由があります。

関連情報

SQLGetConnectAttr 関数

1.5.5.1.3 ODBC アプリケーションでのスレッドと接続

マルチスレッド ODBC アプリケーションを開発できます。各スレッドに対して別個の接続を使用します。

複数のスレッドに対して単一の接続を使用できます。ただし、データベースサーバは、1 つの接続を使って同時に複数の要求を出すことを許可しません。あるスレッドが長時間かかる文を実行すると、他のすべてのスレッドはその要求が終わるまで待たされます。

1.5.5.1.4 ODBC 接続障害

ODBC ドライバで必要なすべてのファイルをデプロイしなかった場合、データベースへの接続時に次のエラーが発生する可能性があります。

```
[IM004][Microsoft][ODBC Driver Manager] Driver's SQLAllocHandle on SQL_HANDLE_ENV
failed
```

ODBC ドライバの適切な動作に必要なすべてのファイルをインストールしていることを確認してください。

関連情報

ODBC クライアントの配備 [816 ページ]

1.5.6 ODBC によって変更されるサーバオプション

ODBC ドライバは、データベースサーバへの接続時に一部のテンポラリサーバオプションを設定します。

次のオプションが、以下で示すように設定されます。

date_format

yyyy-mm-dd

date_order

ymd

isolation_level

SQLSetConnectAttr の SQL_ATTR_TXN_ISOLATION/SA_SQL_ATTR_TXN_ISOLATION 属性の設定に基づく。次のオプションを使用できます。

```
SQL_TXN_READ_UNCOMMITTED
SQL_TXN_READ_COMMITTED
SQL_TXN_REPEATABLE_READ
SQL_TXN_SERIALIZABLE
SA_SQL_TXN_SNAPSHOT
SA_SQL_TXN_STATEMENT_SNAPSHOT
SA_SQL_TXN_READONLY_STATEMENT_SNAPSHOT
```

time_format

hh:nn:ss

timestamp_format

yyyy-mm-dd hh:nn:ss.ssssss

timestamp_with_time_zone_format

yyyy-mm-dd hh:nn:ss.ssssss +hh:nn

ODBC ドライバの一貫した動作を保証するには、これらのオプションの設定を変更しないでください。

関連情報

ODBC トランザクションの独立性レベル [179 ページ]

1.5.7 SQLSetConnectAttr 拡張接続属性

ODBC ドライバは、拡張された一部の接続属性をサポートしています。

SA_REGISTER_MESSAGE_CALLBACK

メッセージは、SQL MESSAGE 文を使用してクライアントアプリケーションからデータベースサーバに送信できます。実行時間が長いデータベースサーバ文によってメッセージも生成できます。

メッセージハンドラルーチンを作成して、これらのメッセージを捕捉できます。メッセージハンドラのコールバックプロトタイプを次に示します。

```
void SQL_CALLBACK message_handler(
SQLHDBC sqlany_dbc,
unsigned char  msg_type,
long code,
unsigned short  length,
char * message
);
```

`msg_type` に指定できる次の値は、`sqldef.h` で定義されています。

MESSAGE_TYPE_INFO

メッセージタイプは INFO でした。

MESSAGE_TYPE_WARNING

メッセージタイプは WARNING でした。

MESSAGE_TYPE_ACTION

メッセージタイプは ACTION でした。

MESSAGE_TYPE_STATUS

メッセージタイプは STATUS でした。

MESSAGE_TYPE_PROGRESS

メッセージタイプは PROGRESS でした。このタイプのメッセージは、BACKUP DATABASE や LOAD TABLE などの実行時間が長いデータベースサーバ文によって生成されます。

メッセージに関連付けられている SQLCODE を `code` に指定することができます。指定がない場合、`code` パラメータの値は 0 です。

メッセージの長さは `length` に記述されています。

メッセージへのポインタは `message` に記述されています。メッセージ文字列は、NULL で終了しません。この問題を処理するようにアプリケーションを設計する必要があります。次はその例です。

```
memcpy( mybuff, msg, len );
mybuff[ len ] = '¥0';
```

メッセージハンドラを ODBC に登録するには、次のようにして SQLSetConnectAttr 関数を呼び出します。

```
rc = SQLSetConnectAttr(
   hdbc,
   SA_REGISTER_MESSAGE_CALLBACK,
   (SQLPOINTER) &message_handler, SQL_IS_POINTER );
```

メッセージハンドラの登録を ODBC から解除するには、次のようにして SQLSetConnectAttr 関数を呼び出します。

```
rc = SQLSetConnectAttr(
   hdbc,
   SA_REGISTER_MESSAGE_CALLBACK,
   NULL, SQL_IS_POINTER );
```

SA_GET_MESSAGE_CALLBACK_PARM

メッセージハンドラのコールバックルーチンに渡される SQLHDBC 接続ハンドルの値を取得するには、SA_GET_MESSAGE_CALLBACK_PARM パラメータを指定して SQLGetConnectAttr 関数を呼び出します。

```
SQLHDBC callback_hdbc = NULL;
rc = SQLGetConnectAttr(
   hdbc,
   SA_GET_MESSAGE_CALLBACK_PARM,
   (SQLPOINTER) &callback_hdbc, 0, 0 );
```

戻り値は、メッセージハンドラのコールバックルーチンに渡されるパラメータ値と同じです。

SA_REGISTER_VALIDATE_FILE_TRANSFER_CALLBACK

これは、ファイル転送の検証コールバック関数を登録するために使用します。転送を許可する前に、ODBC ドライバは検証コールバックが存在する場合は、それを呼び出します。ストアドプロシージャからなどの間接文の実行中にクライアントのデータ転送が要求された場合、ODBC ドライバはクライアントアプリケーションで検証コールバックが登録されていないかぎり転送を許可しません。どのような状況で検証の呼び出しが行われるかについては、以下でより詳しく説明します。

コールバックプロトタイプを次に示します。

```
int SQL_CALLBACK file_transfer_callback(
void * sqlca,
char * file_name,
int is_write
);
```

`file_name` パラメータは、読み込みまたは書き込み対象のファイルの名前です。`is_write` パラメータは、読み込み (クライアントからサーバへの転送) が要求された場合は 0、書き込みが要求された場合は 0 以外の値になります。ファイル転送が許可されない場合、コールバック関数は 0 を返します。それ以外の場合は 0 以外の値を返します。

データのセキュリティ上、サーバはファイル転送を要求している文の実行元を追跡します。サーバは、文がクライアントアプリケーションから直接受信されたものかどうかを判断します。クライアントからデータ転送を開始する際に、サーバは文の実行元に関する情報をクライアントソフトウェアに送信します。クライアント側では、クライアントアプリケーションから直接送信された文を実行するためにデータ転送が要求されている場合にかぎり、ODBC ドライバはデータの転送を無条件で許可します。それ以外の場合は、上述の検証コールバックがアプリケーションで登録されていることが必要です。登録されていない場合、転送は拒否されて文が失敗し、エラーが発生します。データベース内に既に存在しているストアドプロシージャがクライアントの文で呼び出された場合、ストアドプロシージャそのものの実行はクライアントの文で開始されたものと見なされません。ただし、クライアントアプリケーションでテンポラリストアドプロシージャを明示的に作成してストアドプロシージャを実行した場合、そのプロシージャはクライアントによって開始されたものとしてサーバは処理します。同様に、クライアントアプリケーションでバッチ文を実行する場合も、バッチ文はクライアントアプリケーションによって直接実行されるものと見なされます。

SA_SQL_ATTR_TXN_ISOLATION

これは、拡張されたトランザクションの独立性レベルを設定するために使用します。次の例は、Snapshot 独立性レベルを設定します。

```
SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
SQLSetConnectAttr( dbc, SA_SQL_ATTR_TXN_ISOLATION,
      SA_SQL_TXN_SNAPSHOT, SQL_IS_UINTEGER );
```

関連情報

ODBC トランザクションの独立性レベル [179 ページ]

1.5.8 Windows DllMain 関数の考慮事項

Windows ダイナミックリンクライブラリ内の DllMain 関数から直接的または間接的に ODBC 関数を呼び出さないでください。DllMain のエントリポイント関数は、簡単な初期化および終了処理のみを実行するためのものです。SQLFreeHandle、SQLFreeConnect、SQLFreeEnv などの ODBC 関数を呼び出した場合、デッドロックや循環依存が発生する可能性があります。

適切でないプログラミングコードの例を次に示します。Microsoft ODBC ドライバマネージャは、ODBC ドライバへの最後のアクセスが完了したことを検出すると、ドライバのアンロードを実行します。ODBC ドライバが停止すると、アクティブなスレッドがすべて停止します。スレッドの終了の結果、DllMain への再帰的なスレッド detach 呼び出しが発生します。DllMain への呼び出しは直列化されていて、呼び出しが進行中であるため、新しいスレッド detach 呼び出しが開始されることはありません。ODBC ドライバはスレッドが終了するまで永久に待機することになり、アプリケーションがハングします。

```
BOOL APIENTRY DllMain( HMODULE hinstDLL,
    DWORD  fdwReason,
    LPVOID lpvReserved
    )
{
    HANDLE      *handles;
    switch( fdwReason ) {
.
.
.
    case DLL_THREAD_DETACH:
        /* do thread cleanup */
```

```
        handles = (HANDLE *) TlsGetValue( TlsIndex );
        if( handles != NULL )
        {
            SQLHENV     tls_henv;
            SQLHDBC     tls_hdbc;

            tls_henv = (SQLHENV) handles[0];
            tls_hdbc = (SQLHDBC) handles[1];
            if( tls_hdbc != NULL )
                SQLFreeHandle( SQL_HANDLE_DBC, tls_hdbc );
            if( tls_henv != NULL )
                SQLFreeHandle( SQL_HANDLE_ENV, tls_henv );
            handles[0] = NULL;
            handles[1] = NULL;
        }
        break;
.
.
.
    }
    return TRUE;        /* indicate success */
}
```

関連情報

Dynamic-Link Library Best Practices

1.5.9 SQL 文の実行方法

ODBC には、SQL 文を実行するための関数がいくつか含まれます。

直接実行

データベースサーバは SQL 文を解析したうえで、実行プランを準備し、SQL 文を実行します。解析とアクセスプランの準備を、文の準備と呼びます。

準備後の実行

文の準備が、実行とは別々に行われます。繰り返し実行される文の場合は、準備後に実行することでそのたびに準備する必要がなくなり、パフォーマンスが向上します。

このセクションの内容:

文の直接実行 [170 ページ]
ODBC アプリケーションで SQL 文を実行するには、SQLAllocHandle を使用して文のハンドルを割り付け、SQLExecDirect 関数を呼び出して文を実行します。

バウンドパラメータを使用した文の実行 [170 ページ]
バインドされたパラメータを使用して SQL 文を構成して実行し、実行時に文のパラメータ値を設定します。

準備文の実行 [172 ページ]
準備文を実行すると、繰り返し使用する文のパフォーマンスが向上します。

1.5.9.1 文の直接実行

ODBC アプリケーションで SQL 文を実行するには、SQLAllocHandle を使用して文のハンドルを割り付け、SQLExecDirect 関数を呼び出して文を実行します。

いずれかのパラメータを文の一部として含めます (たとえば、WHERE 句はその引数を指定します)。または、バインドされたパラメータを使用して文を作成することもできます。

SQLExecDirect 関数は、ステートメントハンドル、SQL 文字列、長さまたは終了インジケータをパラメータに取ります。この場合、終了インジケータは NULL で終了された文字列インジケータです。文には、パラメータを指定することもできます。

エラーチェックを含む完全なサンプルについては、*%SQLANYSAMP17%*`¥SQLAnywhere¥ODBCExecute¥odbcexecute.cpp` を参照してください。

例

次の例は、ハンドル dbc を使用した接続時に、名前が stmt の SQL_HANDLE_STMT 型のハンドルを割り付ける方法を示します。

```
SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
```

次の例は、文を宣言して実行する方法を示します。

```
SQLCHAR deletestmt[ STMT_LEN ] =
  "DELETE FROM Departments WHERE DepartmentID = 201";
SQLExecDirect( stmt, deletestmt, SQL_NTS) ;
```

通常、deletestmt の宣言は関数の先頭で行います。

関連情報

バウンドパラメータを使用した文の実行 [170 ページ]
SQLExecDirect 関数

1.5.9.2 バウンドパラメータを使用した文の実行

バインドされたパラメータを使用して SQL 文を構成して実行し、実行時に文のパラメータ値を設定します。

前提条件

サンプルを正常に実行するために、次のシステム権限が必要になります。

- Departments テーブルでの INSERT

コンテキスト

バインドされたパラメータを準備文で使用すると、複数回実行される文のパフォーマンスが向上します。

手順

1. SQLAllocHandle を使用して文にハンドルを割り付けます。

   たとえば、次の文はハンドル `dbc` を使用した接続時に、`stmt` という名前の `SQL_HANDLE_STMT` 型のハンドルを割り付けます。

   ```
   SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
   ```

2. SQLBindParameter を使用して文のパラメータをバインドします。

   たとえば、次の行は、department ID、department name、manager ID、および文の文字列の値を保持する変数を宣言します。次に、stmt ステートメントハンドルを使用して実行される文の 1 番目、2 番目、3 番目のパラメータにパラメータをバインドします。

   ```
   #defined DEPT_NAME_LEN 40
   SQLLEN cbDeptID = 0, cbDeptName = SQL_NTS, cbManagerID = 0;
   SQLSMALLINT deptID, managerID;
   SQLCHAR deptName[ DEPT_NAME_LEN + 1 ];
   SQLCHAR insertstmt[ STMT_LEN ] =
     "INSERT INTO Departments "
     "( DepartmentID, DepartmentName, DepartmentHeadID ) "
     "VALUES (?, ?, ?)";
   SQLBindParameter( stmt, 1, SQL_PARAM_INPUT,
       SQL_C_SSHORT, SQL_INTEGER, 0, 0,
       &deptID, 0, &cbDeptID );
   SQLBindParameter( stmt, 2, SQL_PARAM_INPUT,
       SQL_C_CHAR, SQL_CHAR, DEPT_NAME_LEN, 0,
       deptName, 0,&cbDeptName );
   SQLBindParameter( stmt, 3, SQL_PARAM_INPUT,
       SQL_C_SSHORT, SQL_INTEGER, 0, 0,
       &managerID, 0, &cbManagerID );
   ```

3. パラメータに値を割り当てます。

   たとえば、次の行は、手順 2 のフラグメントのパラメータに値を割り当てます。

   ```
   deptID = 201;
   strcpy( (char * ) deptName, "Sales East" );
   managerID = 902;
   ```

   通常、これらの変数はユーザのアクションに応じて設定されます。

4. SQLExecDirect を使って文を実行します。

   たとえば、次の行は、ステートメントハンドル `insertstmt` の `stmt` に保持されている文の文字列を実行します。

   ```
   SQLExecDirect( stmt, insertstmt, SQL_NTS );
   ```

結果

ビルドして実行されると、アプリケーションは SQL 文を実行します。

次のステップ

前述のコードフラグメントには、エラーチェックは含まれていません。エラーチェックを含む完全なサンプルについては、*%SQLANYSAMP17%*¥SQLAnywhere¥ODBCExecute¥odbcexecute.cpp を参照してください。

関連情報

準備文の実行 [172 ページ]

1.5.9.3 準備文の実行

準備文を実行すると、繰り返し使用する文のパフォーマンスが向上します。

前提条件

サンプルを正常に実行するために、次のシステム権限が必要になります。

- Departments テーブルでの INSERT

手順

1. SQLPrepare を使って文を準備します。

   たとえば、次のコードフラグメントは、INSERT 文の準備方法を示します。

   ```
   rc = SQLPrepare( stmt,
       "INSERT INTO Departments( DepartmentID, DepartmentName, DepartmentHeadID ) "
       "VALUES (?, ?, ?)",
       SQL_NTS );
   ```

   この例では次のようになっています。

   rc

   操作の成功または失敗をテストするリターンコードを受け取ります。

   stmt

文にハンドルを割り付け、後で参照できるようにします。
?

疑問符は文のパラメータのためのプレースホルダです。プレースホルダは文の中に置いて、どこでホスト変数にアクセスするかを指定します。プレースホルダは、疑問符 (?) かホスト変数参照です (ホスト変数名の前にはコロンを付けます)。ホスト変数参照の場合も、実際の文テキスト内のホスト変数名は、対応するパラメータがそれにバインドされることを示すプレースホルダの役割しかありません。実際のパラメータ名に一致する必要はありません。

2. SQLBindParameter を使用して文のパラメータ値をバインドします。

   たとえば、次の関数呼び出しは DepartmentID 変数の値をバインドします。

```
rc = SQLBindParameter( stmt,
                  1,
                  SQL_PARAM_INPUT,
                  SQL_C_SSHORT,
                  SQL_INTEGER,
                  0,
                  0,
                  &sDeptID,
                  0,
                  &cbDeptID );
```

   この例では次のようになっています。

   rc

   操作の成功または失敗をテストするリターンコードが設定されます。
   stmt

   はステートメントハンドルです。
   1

   は、この呼び出しで最初のプレースホルダの値が設定されることを示します。
   SQL_PARAM_INPUT

   は、パラメータが入力文であることを示します。
   SQL_C_SHORT

   は、アプリケーションで C データ型が使用されていることを示します。
   SQL_INTEGER

   は、データベース内で SQL データ型が使用されていることを示します。

   次の 2 つのパラメータは、カラム精度と 10 進数の小数点以下の桁数を示します。ともに、0 は整数を表します。
   &sDeptID

   は、パラメータ値のバッファへのポインタです。
   0

   はバッファの長さを示すバイト数です。
   &cbDeptID

   はパラメータ値の長さが設定されるバッファへのポインタです。

3. 他の 2 つのパラメータをバインドし、sDeptId に値を割り当てます。
4. 次の文を実行します。

```
rc = SQLExecute( stmt );
```

手順 2 ～ 4 は、複数回実行できます。

5. 文を削除します。

   文を削除すると、文自体に関連付けられているリソースが解放されます。文の削除には、SQLFreeHandle を使用します。

結果

ビルドして実行されると、アプリケーションは準備文を実行します。

次のステップ

前述のコードフラグメントには、エラーチェックは含まれていません。エラーチェックを含む完全なサンプルについては、*%SQLANYSAMP17%*¥SQLAnywhere¥ODBCPrepare¥odbcprepare.cpp を参照してください。

関連情報

準備文 [13 ページ]

1.5.10 64 ビット ODBC での考慮事項

SQLBindCol、SQLBindParameter、SQLGetData などの ODBC 関数を使用する場合、一部のパラメータは SQLLEN や SQLULEN として関数プロトタイプに型指定されます。

参照している Microsoft の『ODBC API Reference』マニュアルによっては、同じパラメータが SQLINTEGER や SQLUINTEGER として記述されている場合があります。

SQLLEN および SQLULEN のデータ項目は、64 ビットの ODBC アプリケーションでは 64 ビット、32 ビットの ODBC アプリケーションでは 32 ビットになります。SQLINTEGER および SQLUINTEGER のデータ項目は、すべてのプラットフォームで 32 ビットです。

この問題を説明するために、次の ODBC 関数プロトタイプを Microsoft の旧版の『ODBC API Reference』から抜粋しました。

```
SQLRETURN SQLGetData(
     SQLHSTMT     StatementHandle,
     SQLUSMALLINT ColumnNumber,
     SQLSMALLINT  TargetType,
     SQLPOINTER   TargetValuePtr,
     SQLINTEGER   BufferLength,
     SQLINTEGER  *StrLen_or_IndPtr);
```

Microsoft Visual Studio バージョン 8 の sql.h にある実際の関数プロトタイプと比較してください。

```
SQLRETURN  SQL_API SQLGetData(
    SQLHSTMT      StatementHandle,
    SQLUSMALLINT  ColumnNumber,
    SQLSMALLINT   TargetType,
    SQLPOINTER    TargetValue,
    SQLLEN        BufferLength,
    SQLLEN       *StrLen_or_Ind);
```

BufferLength パラメータと StrLen_or_Ind パラメータが、SQLINTEGER 型ではなく SQLLEN 型と指定されるようになったことがわかります。64 ビットのプラットフォームでは、32 ビット数ではなく 64 ビット数であることが Microsoft のマニュアルからわかります。

異種プラットフォーム間でのコンパイルの問題を回避するために、データベースサーバソフトウェアには独自の ODBC ヘッダファイルがあります。Windows プラットフォームの場合、ntodbc.h ヘッダファイルをインクルードします。Linux などの UNIX プラットフォームの場合は、unixodbc.h ヘッダファイルをインクルードします。これらのヘッダファイルを使用することで、対象プラットフォーム用の ODBC ドライバとの互換性が確保されます。

次の表に示すのは、一般的な ODBC のタイプの一部です。64 ビットのプラットフォームと 32 ビットのプラットフォームでストレージサイズが同じものもあれば、異なるものもあります。

ODBC API	64 ビットのプラットフォーム	32 ビットのプラットフォーム
SQLINTEGER	32 ビット	32 ビット
SQLUINTEGER	32 ビット	32 ビット
SQLLEN	64 ビット	32 ビット
SQLULEN	64 ビット	32 ビット
SQLSETPOSIROW	64 ビット	16 ビット
SQL_C_BOOKMARK	64 ビット	32 ビット
BOOKMARK	64 ビット	32 ビット

データ変数とパラメータを間違って宣言すると、ソフトウェアが正しく動作しない可能性があります。

次の表は、64 ビットのサポートの導入とともに変更された、ODBC API の関数プロトタイプを示します。影響を受けるパラメータが記載されています。関数プロトタイプで使用される実際のパラメータ名と Microsoft のマニュアルに記載されているパラメータ名が異なる場合は、Microsoft の記載名がカッコ内に示されています。パラメータ名は、Microsoft Visual Studio バージョン 8 のヘッダファイルで使用されるものです。

ODBC API	パラメータ (マニュアル記載のパラメータ名)
SQLBindCol	SQLLEN BufferLength SQLLEN *Strlen_or_Ind
SQLBindParam	SQLULEN LengthPrecision SQLLEN *Strlen_or_Ind

ODBC API	パラメータ (マニュアル記載のパラメータ名)
SQLBindParameter	SQLULEN cbColDef (ColumnSize) SQLLEN cbValueMax (BufferLength) SQLLEN *pcbValue (Strlen_or_IndPtr)
SQLColAttribute	SQLLEN *NumericAttribute
SQLColAttributes	SQLLEN *pfDesc
SQLDescribeCol	SQLULEN *ColumnSize (ColumnSizePtr)
SQLDescribeParam	SQLULEN *pcbParamDef (ParameterSizePtr)
SQLExtendedFetch	SQLLEN irow (FetchOffset) SQLULEN *pcrow (RowCountPtr)
SQLFetchScroll	SQLLEN FetchOffset
SQLGetData	SQLLEN BufferLength SQLLEN *Strlen_or_Ind (Strlen_or_IndPtr)
SQLGetDescRec	SQLLEN *Length (LengthPtr)
SQLParamOptions	SQLULEN crow, SQLULEN *pirow
SQLPutData	SQLLEN Strlen_or_Ind
SQLRowCount	SQLLEN *RowCount (RowCountPtr)
SQLSetConnectOption	SQLULEN Value
SQLSetDescRec	SQLLEN Length SQLLEN *StringLength (StringLengthPtr) SQLLEN *Indicator (IndicatorPtr)
SQLSetParam	SQLULEN LengthPrecision SQLLEN *Strlen_or_Ind (Strlen_or_IndPtr)
SQLSetPos	SQLSETPOSIROW irow (RowNumber)
SQLSetScrollOptions	SQLLEN crowKeyset
SQLSetStmtOption	SQLULEN Value

ポインタを介して ODBC API 呼び出しに渡され、ODBC API 呼び出しから返される値の一部は、64 ビットのアプリケーションに対応するために変更されました。たとえば、次の SQLSetStmtAttr および SQLSetDescField 関数の値は、SQLINTEGER/SQLUINTEGER ではなくなりました。SQLGetStmtAttr および SQLGetDescField 関数の該当するパラメータに関しても同様です。

ODBC API	Value/ValuePtr 変数の型
SQLSetStmtAttr(SQL_ATTR_FETCH_BOOKMARK_PTR)	SQLLEN * value

ODBC API	Value/ValuePtr 変数の型
SQLSetStmtAttr(SQL_ATTR_KEYSET_SIZE)	SQLULEN value
SQLSetStmtAttr(SQL_ATTR_MAX_LENGTH)	SQLULEN value
SQLSetStmtAttr(SQL_ATTR_MAX_ROWS)	SQLULEN value
SQLSetStmtAttr(SQL_ATTR_PARAM_BIND_OFFSET_PTR)	SQLULEN * value
SQLSetStmtAttr(SQL_ATTR_PARAMS_PROCESSED_PTR)	SQLULEN * value
SQLSetStmtAttr(SQL_ATTR_PARAMSET_SIZE)	SQLULEN value
SQLSetStmtAttr(SQL_ATTR_ROW_ARRAY_SIZE)	SQLULEN value
SQLSetStmtAttr(SQL_ATTR_ROW_BIND_OFFSET_PTR)	SQLULEN * value
SQLSetStmtAttr(SQL_ATTR_ROW_NUMBER)	SQLULEN value
SQLSetStmtAttr(SQL_ATTR_ROWS_FETCHED_PTR)	SQLULEN * value
SQLSetDescField(SQL_DESC_ARRAY_SIZE)	SQLULEN value
SQLSetDescField(SQL_DESC_BIND_OFFSET_PTR)	SQLLEN * value
SQLSetDescField(SQL_DESC_ROWS_PROCESSED_PTR)	SQLULEN * value
SQLSetDescField(SQL_DESC_DISPLAY_SIZE)	SQLLEN value
SQLSetDescField(SQL_DESC_INDICATOR_PTR)	SQLLEN * value
SQLSetDescField(SQL_DESC_LENGTH)	SQLLEN value
SQLSetDescField(SQL_DESC_OCTET_LENGTH)	SQLLEN value
SQLSetDescField(SQL_DESC_OCTET_LENGTH_PTR)	SQLLEN * value

関連情報

MDAC 2.7 での ODBC 64 ビット API の変更点

1.5.11 データアラインメントの要件

SQLBindCol、SQLBindParameter、または SQLGetData を使用する場合、カラムまたはパラメータには C データ型が指定されます。

プラットフォームによっては、指定された型の値をフェッチまたは格納するために、各カラム用のストレージ (メモリ) を適切にアラインする必要があります。ODBC ドライバは、データアラインメントが適切かどうかを確認します。オブジェクトが適切に揃っていない場合、ODBC ドライバは *"Invalid string or buffer length"* (無効な文字列またはバッファ長) というメッセージ (*SQLSTATE* HY090 または S1090) を出力します。

次の表は、Sun Sparc、Itanium-IA64、ARM ベースのデバイスなどのプロセッサに対するメモリアラインメント要件を示したものです。データ値のメモリアドレスは、示された値の倍数である必要があります。

C データ型	必要なアラインメント
SQL_C_CHAR	なし
SQL_C_BINARY	なし
SQL_C_GUID	なし
SQL_C_BIT	なし
SQL_C_STINYINT	なし
SQL_C_UTINYINT	なし
SQL_C_TINYINT	なし
SQL_C_NUMERIC	なし
SQL_C_DEFAULT	なし
SQL_C_SSHORT	2
SQL_C_USHORT	2
SQL_C_SHORT	2
SQL_C_DATE	2
SQL_C_TIME	2
SQL_C_TIMESTAMP	2
SQL_C_TYPE_DATE	2
SQL_C_TYPE_TIME	2
SQL_C_TYPE_TIMESTAMP	2
SQL_C_WCHAR	2 (すべてのプラットフォームでバッファサイズは 2 の倍数であることが必要)
SQL_C_SLONG	4
SQL_C_ULONG	4
SQL_C_LONG	4
SQL_C_FLOAT	4
SQL_C_DOUBLE	8 (ARM の場合は 4)
SQL_C_SBIGINT	8
SQL_C_UBIGINT	8

x86、x64、および PowerPC プラットフォームではメモリアラインメントは必要ありません。x64 プラットフォームには、Advanced Micro Devices (AMD) AMD64 プロセッサや Intel Extended Memory 64 Technology (EM64T) プロセッサなどがあります。

1.5.12 ODBC アプリケーションの結果セット

ODBC アプリケーションはカーソルを使用して、結果セットを操作および更新します。ソフトウェアは、多種多様なカーソルとカーソル処理をサポートしています。

このセクションの内容:

ODBC トランザクションの独立性レベル [179 ページ]
SQLSetConnectAttr を使用して、接続に関するトランザクションの独立性レベルを設定できます。

ODBC カーソル特性 [180 ページ]
文を実行して結果セットを操作する ODBC 関数は、カーソルを使用してタスクを実行します。アプリケーションは SQLExecute 関数または SQLExecDirect 関数を実行するたびに、暗黙的にカーソルを開きます。

データの取得 [181 ページ]
データベースからローを取り出すには、SQLExecute または SQLExecDirect を使用して SELECT 文を実行します。これにより文に対するカーソルが開きます。

カーソルを使用したローの更新と削除 [183 ページ]
カーソルを使用してローを更新および削除することができます。

ブックマーク [183 ページ]
ODBC にはブックマークがあります。これはカーソル内のローの識別に使用する値です。ODBC ドライバは、value-sensitive と insensitive カーソルにブックマークをサポートします。たとえば、ODBC カーソルタイプの SQL_CURSOR_STATIC と SQL_CURSOR_KEYSET_DRIVEN ではブックマークがサポートされますが、カーソルタイプ SQL_CURSOR_DYNAMIC と SQL_CURSOR_FORWARD_ONLY ではブックマークがサポートされません。

関連情報

カーソルの原則 [19 ページ]

1.5.12.1 ODBC トランザクションの独立性レベル

SQLSetConnectAttr を使用して、接続に関するトランザクションの独立性レベルを設定できます。

ソフトウェアに用意されているトランザクションの独立性レベルを決定する特性は、次のとおりです。

SQL_TXN_READ_UNCOMMITTED

独立性レベルを 0 に設定します。この属性値を設定すると、別のユーザによる変更から読み込まれたデータは分離され、その変更内容は表示されません。READ 文の再実行は別のユーザによって影響されます。繰り返し可能読み出しはサポートされていません。これは独立性レベルのデフォルト値です。

SQL_TXN_READ_COMMITTED

独立性レベルを 1 に設定します。この属性値を設定すると、別のユーザによる変更から読み込まれたデータは分離されず、その変更内容は表示されます。READ 文の再実行は別のユーザによって影響されます。繰り返し可能読み出しはサポートされていません。

SQL_TXN_REPEATABLE_READ

独立性レベルを 2 に設定します。この属性値を設定すると、別のユーザによる変更から読み込まれたデータは分離され、その変更内容は表示されません。READ 文の再実行は別のユーザによって影響されます。繰り返し可能読み出しはサポートされています。

SQL_TXN_SERIALIZABLE

独立性レベルを 3 に設定します。この属性値を設定すると、別のユーザによる変更から読み込まれたデータは分離され、その変更内容は表示されません。READ 文の再実行は別のユーザによって影響されません。繰り返し可能読み出しはサポートされています。
SA_SQL_TXN_SNAPSHOT

独立性レベルを Snapshot に設定します。この属性値を設定すると、トランザクション全体のデータベースに関する単一ビューが表示されます。
SA_SQL_TXN_STATEMENT_SNAPSHOT

独立性レベルを Statement-snapshot に設定します。この属性値を設定すると、Snapshot 独立性よりデータの整合性は低くなりますが、トランザクションを長時間実行したためにバージョン情報を格納するテンポラリファイルのサイズが大きくなりすぎる場合には有益です。
SA_SQL_TXN_READONLY_STATEMENT_SNAPSHOT

独立性レベルを Readonly-statement-snapshot に設定します。この属性値を設定すると、Statement-snapshot 独立性よりデータの整合性は低くなりますが、更新の競合は回避されます。このため、この属性は元々異なる独立性レベルで実行することを想定していたアプリケーションを移植するのに最も適しています。

このオプションを Snapshot、Statement-snapshot、または Readonly-statement-snapshot に設定する場合は、allow_snapshot_isolation データベースオプションを On に設定する必要があります。

例

次のフラグメントは、独立性レベルを Snapshot に設定します。

```
SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
SQLSetConnectAttr( dbc, SQL_ATTR_TXN_ISOLATION,
      SA_SQL_TXN_SNAPSHOT, SQL_IS_UINTEGER );
```

関連情報

SQLSetConnectAttr 関数
Microsoft オープンデータベースコネクティビティ (ODBC)

1.5.12.2 ODBC カーソル特性

文を実行して結果セットを操作する ODBC 関数は、カーソルを使用してタスクを実行します。アプリケーションは SQLExecute 関数または SQLExecDirect 関数を実行するたびに、暗黙的にカーソルを開きます。

結果セットを前方にのみ移動し、更新はしないアプリケーションの場合、カーソルの動作は比較的単純です。ODBC アプリケーションは、デフォルトではこの動作を要求します。ODBC は読み込み専用で前方専用のカーソルを定義します。この場合、データベースサーバではパフォーマンスが向上するように最適化されたカーソルが提供されます。

数多くのグラフィカルユーザインタフェースアプリケーションのように、結果セット内で前後にスクロールするアプリケーションの場合、カーソルの動作はかなり複雑です。ODBC では、アプリケーションに適した動作を組み込めるように、さまざまなスクロール可能カーソルを定義しています。データベースサーバには、ODBC のスクロール可能カーソルタイプに適合するカーソルのフルセットが用意されています。

必要な ODBC カーソル特性を設定するには、文の属性を定義する SQLSetStmtAttr 関数を呼び出します。SQLSetStmtAttr 関数は、結果セットを作成する文の実行前に呼び出してください。

SQLSetStmtAttr を使用すると、多数のカーソル特性を設定できます。データベースサーバに用意されているカーソルタイプを決定する特性は、次のとおりです。

SQL_ATTR_CURSOR_SCROLLABLE

スクロール可能カーソルの場合は SQL_SCROLLABLE、前方専用カーソルの場合は SQL_NONSCROLLABLE に設定します。SQL_NONSCROLLABLE がデフォルトです。

SQL_ATTR_CONCURRENCY

次のいずれかの値に設定します。

SQL_CONCUR_READ_ONLY

更新禁止になります。SQL_CONCUR_READ_ONLY がデフォルトです。

SQL_CONCUR_LOCK

ローを確実に更新できるロックの最下位レベルを使用します。

SQL_CONCUR_ROWVER

オプティミスティック同時実行制御を使用すると、keyset-driven カーソルを利用して、結果セットをスクロールするときにローが変更または削除されるとアプリケーションへ通知できるようになります。

SQL_CONCUR_VALUES

オプティミスティック同時実行制御を使用すると、keyset-driven カーソルを利用して、結果セットをスクロールするときにローが変更または削除されるとアプリケーションへ通知できるようになります。

例

次のフラグメントは、読み込み専用のスクロール可能カーソルを要求します。

```
SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
SQLSetStmtAttr( stmt, SQL_ATTR_CURSOR_SCROLLABLE,
      SQL_SCROLLABLE, SQL_IS_UINTEGER );
```

関連情報

データの取得 [181 ページ]
value-sensitive カーソル [36 ページ]
SQLSetStmtAttr 関数

1.5.12.3 データの取得

データベースからローを取り出すには、SQLExecute または SQLExecDirect を使用して SELECT 文を実行します。これにより文に対するカーソルが開きます。

次に、SQLFetch または SQLFetchScroll を使用し、カーソルを介してローをフェッチします。これらの関数では、結果セットから次のローセットのデータをフェッチし、バインドされているすべてのカラムのデータを返します。SQLFetchScroll を使用する

と、ローセットを絶対位置や相対位置で指定したり、ブックマークによって指定できます。ODBC 2.0 仕様の古い SQLExtendedFetch は、SQLFetchScroll に置き換えられました。

アプリケーションは、SQLFreeHandle を使用して文を解放するときにカーソルを閉じます。

カーソルから値をフェッチするため、アプリケーションは SQLBindCol か SQLGetData のいずれかを使用します。SQLBindCol を使用すると、フェッチのたびに値が自動的に取り出されます。SQLGetData を使用する場合は、フェッチ後にカラムごとに呼び出してください。

LONG VARCHAR または LONG BINARY などのカラムの値を分割してフェッチするには、SQLGetData を使用します。または、SQL_ATTR_MAX_LENGTH 文の属性を、カラムの値全体を十分に保持できる大きさの値に設定する方法もあります。SQL_ATTR_MAX_LENGTH のデフォルト値は 256 KB です。

ODBC ドライバは、ODBC 仕様で意図されたものとは異なる方法で SQL_ATTR_MAX_LENGTH を実装しています。本来 SQL_ATTR_MAX_LENGTH は、大きなフェッチをトランケートするメカニズムとして使用されることを意図しています。この処理は、データの最初の部分だけを表示するプレビューモードで行われる可能性があります。たとえば、4 MB の blob をサーバからクライアントアプリケーションに転送するのではなく、その先頭 500 バイトだけが転送される可能性があります (SQL_ATTR_MAX_LENGTH が 500 に設定された場合)。ODBC ドライバでは、この実装をサポートしていません。

SQLBindCol を使用して NUMERIC または DECIMAL のカラムを SQL_C_NUMERIC ターゲットタイプにバインドすると、データ値は SQL_NUMERIC_STRUCT の 128 ビット項目 (*val*) に保存されます。この項目に保存できる最大精度は 38 です。一方、データベースサーバでサポートされる NUMERIC 最大精度は 127 です。NUMERIC または DECIMAL のカラムの精度が 38 より大きい場合、精度が失われないように、そのカラムを SQL_C_CHAR としてバインドする必要があります。

次のコードフラグメントは、クエリに対してカーソルを開き、そのカーソルを介してデータを取り出します。わかりやすくするためにエラーチェックは省いています。このフラグメントは、完全なサンプルから抜粋したものです。これは、*%SQLANYSAMP17%*`¥SQLAnywhere¥ODBCSelect¥odbcselect.cpp` にあります。

```
int main( int argc, char* argv[] )
{
    #define DEPT_NAME_LEN 40
    SQLLEN          cbDeptID = 0, cbDeptName = SQL_NTS, cbManagerID = 0;
    SQLCHAR         deptName[ DEPT_NAME_LEN + 1];
    SQLSMALLINT     deptID, managerID;
    SQLHENV         env;
    SQLHDBC         dbc;
    SQLHSTMT        stmt;
    SQLRETURN       retcode;
    argc = ProcessOptions( argv );
    if( argc < 0 ) return( 1 );
    retcode = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env );
    if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
        /* Set the ODBC version environment attribute */
        retcode = SQLSetEnvAttr( env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3,
0);
        retcode = SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
        if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
            retcode = SQLConnect( dbc,
                    (SQLCHAR*) DataSourceName, SQL_NTS,
                    (SQLCHAR*) UserName, SQL_NTS,
                    (SQLCHAR*) Password, SQL_NTS );
            if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
                retcode = SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
                if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
                    SQLBindCol( stmt, 1, SQL_C_SSHORT, &deptID, 0, &cbDeptID);
                    SQLBindCol( stmt, 2, SQL_C_CHAR, deptName, sizeof(deptName),
&cbDeptName);
                    SQLBindCol( stmt, 3, SQL_C_SSHORT, &managerID, 0, &cbManagerID);
                    retcode = SQLExecDirect( stmt, (SQLCHAR * )
                            "SELECT DepartmentID, DepartmentName, DepartmentHeadID "
```

```
                                "FROM Departments "
                                "ORDER BY DepartmentID", SQL_NTS );
                        if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
{
                            while( ( SQLFetch( stmt ) ) != SQL_NO_DATA ){
                                printf( "%d %20s %d¥n", deptID, deptName, managerID );
                            }
                        }
                    }
                    SQLFreeHandle( SQL_HANDLE_STMT, stmt );
                }
                SQLDisconnect( dbc );
            }
            SQLFreeHandle( SQL_HANDLE_DBC, dbc );
        }
        SQLFreeHandle( SQL_HANDLE_ENV, env );
        return( (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) ? 0 : 1 );
}
```

カーソルでフェッチできるローの位置番号は、integer 型のサイズによって管理されます。32 ビット integer に格納できる値より 1 小さい 2147483646 までの番号が付けられたローをフェッチできます。ローの位置番号に、クエリ結果の最後を基準として負の数を使用している場合、integer に格納できる負の最大値より 1 大きい数までの番号のローをフェッチできます。

1.5.12.4 カーソルを使用したローの更新と削除

カーソルを使用してローを更新および削除することができます。

位置付け UPDATE 文を使用する場合、SELECT ...FOR UPDATE 文を実行する必要はありません。

次の条件を満たしている限り、カーソルは自動的に更新可能です。

- 基本となるクエリが更新をサポートしている。
  つまり、結果のカラムに対するデータ操作文が有効であるかぎり、位置付けデータ操作文をカーソルに対して実行できます。
  ansi_update_constraints データベースオプションは更新可能なクエリの種類を制限します。
- カーソルタイプが更新をサポートしている。
  読み込み専用カーソルを使用している場合、結果セットを更新できません。

ODBC で位置付け更新と位置付け削除を実行するには、2 つの手段があります。

- SQLSetPos 関数を使用する。
  指定されたパラメータ (SQL_POSITION、SQL_REFRESH、SQL_UPDATE、SQL_DELETE) に応じて、SQLSetPos はカーソル位置を設定し、アプリケーションがデータをリフレッシュしたり、結果セットのデータを更新または削除できるようにします。
  これは、データベースサーバで使用する方法です。
- SQLExecute を使用して、位置付け UPDATE 文と位置付け DELETE 文を送信する。この方法は、データベースサーバでは使用しないでください。

1.5.12.5 ブックマーク

ODBC にはブックマークがあります。これはカーソル内のローの識別に使用する値です。ODBC ドライバは、value-sensitive と insensitive カーソルにブックマークをサポートします。たとえば、ODBC カーソルタイプの SQL_CURSOR_STATIC と

SQL_CURSOR_KEYSET_DRIVEN ではブックマークがサポートされますが、カーソルタイプ SQL_CURSOR_DYNAMIC と SQL_CURSOR_FORWARD_ONLY ではブックマークがサポートされません。

ODBC 3.0 より前のバージョンでは、データベースはブックマークをサポートするかどうかを指定するだけであり、カーソルタイプごとにブックマークの情報を提供するインタフェースはありませんでした。このため、サポートされているカーソルブックマークの種類を示す手段が、データベースサーバにはありませんでした。ODBC 2 アプリケーションでは、ODBC ドライバはブックマークをサポートしています。したがって、動的カーソルにブックマークを使用することもできますが、この組み合わせは使用しないでください。

1.5.13 ストアドプロシージャの考慮事項

ODBC アプリケーションからストアドプロシージャを作成して呼び出し、その結果を処理することができます。

プロシージャと結果セット

プロシージャには、結果セットを返すものと返さないものの 2 種類があります。SQLNumResultCols を使用すると、そのどちらであるかを確認できます。プロシージャが結果セットを返さない場合は、結果カラムの数が 0 になります。結果セットがある場合は、他のカーソルの場合と同様に、SQLFetch または SQLExtendedFetch を使用して値をフェッチできます。

プロシージャへのパラメータは、パラメータマーカ (疑問符) を使用して渡してください。INPUT、OUTPUT、または INOUT パラメータのいずれについても、SQLBindParameter を使用して各パラメータマーカ用の記憶領域を割り当てます。

複数の結果セットを処理するために ODBC は、プロシージャが定義した結果セットではなく、現在実行中のカーソルを記述します。したがって、ODBC はストアドプロシージャ定義の RESULT 句で定義されているカラム名を常に記述するわけではありません。この問題を回避するため、プロシージャ結果セットのカーソルでカラムのエイリアスを使用できます。

例

例 1

この例では、結果セットを返さないプロシージャを作成して呼び出します。このプロシージャは、INOUT パラメータを 1 つ受け取り、その値を増分します。この例では、プロシージャが結果セットを返さないため、変数 `num_columns` の値は 0 になります。わかりやすくするためにエラーチェックは省いています。

```
HDBC dbc;
SQLHSTMT stmt;
SQLINTEGER I;
SQLSMALLINT num_columns;
SQLAllocStmt( dbc, &stmt );
SQLExecDirect( stmt,
    "CREATE PROCEDURE Increment( INOUT a INT )"
    "BEGIN "
    "    SET a = a + 1 "
    "END", SQL_NTS );
/* Call the procedure to increment 'I' */
I = 1;
SQLBindParameter( stmt, 1, SQL_C_LONG, SQL_INTEGER, 0, 0, &I, NULL );
SQLExecDirect( stmt, "CALL Increment( ? )", SQL_NTS );
SQLNumResultCols( stmt, &num_columns );
```

例 2

この例では、結果セットを返すプロシージャを呼び出しています。ここでは、プロシージャが 2 つのカラムの結果セットを返すため、変数 num_columns の値は 2 になります。わかりやすくするため、エラーチェックは省略しています。

```
SQLRETURN rc;
SQLHDBC dbc;
SQLHSTMT stmt;
SQLSMALLINT num_columns;
SQLCHAR ID[ 10 ];
SQLCHAR Surname[ 20 ];
SQLExecDirect( stmt,
      "CREATE PROCEDURE EmployeeList() "
      "RESULT( ID CHAR(10), Surname CHAR(20) ) "
      "BEGIN "
      "    SELECT EmployeeID, Surname FROM Employees "
      "END", SQL_NTS );
/* Call the procedure - print the results */
SQLExecDirect( stmt, "CALL EmployeeList()", SQL_NTS );
SQLNumResultCols( stmt, &num_columns );
SQLBindCol( stmt, 1, SQL_C_CHAR, &ID, sizeof(ID), NULL );
SQLBindCol( stmt, 2, SQL_C_CHAR, &Surname, sizeof(Surname), NULL );
for( ;; )
{
   rc = SQLFetch( stmt );
   if( rc == SQL_NO_DATA )
   {
      rc = SQLMoreResults( stmt );
      if( rc == SQL_NO_DATA ) break;
   }
   else
   {
      do_something( ID, Surname );
   }
}
SQLCloseCursor( stmt );
SQLFreeHandle( SQL_HANDLE_STMT, stmt );
```

1.5.14 ODBC エスケープ構文

ODBC エスケープ構文は、任意の ODBC アプリケーションで使用できます。エスケープ構文を使用して、使用しているデータベース管理システムとは関係なく、共通の関数セットを呼び出すことができます。

注記

エスケープ構文を使用しない場合は、文の属性 NOSCAN を設定することによって、ODBC アプリケーションのエスケープ構文解析をオフにします。すると、ODBC ドライバが SQL 文を実行のためにデータベースサーバに送信する前に行う全 SQL 文のスキャンが行われなくなるため、パフォーマンスが向上します。次の文によって、文の属性 NOSCAN を設定します。

```
SQLSetStmtAttr ( hstmt, SQL_ATTR_NOSCAN, (SQLPOINTER) SQL_NOSCAN_ON,
SQL_IS_UINTEGER );
```

エスケープ構文の一般的な形式は次のようになります。

```
{ keyword parameters }
```

次のキーワードセットがあります。

{d date-string}

date-string は、データベースサーバが受け取ることのできる任意の日付値です。
{t time-string}

time-string は、データベースサーバが受け取ることのできる任意の時刻値です。
{ts date-string time-string}

date/time-string は、データベースサーバが受け取ることのできる任意のタイムスタンプ値です。
{guid uuid-string}

uuid-string は、任意の有効な GUID 文字列です (例: 41dfe9ef-db91-11d2-8c43-006008d26a6f)。
{oj outer-join-expr}

outer-join-expr は、データベースサーバが受け取ることのできる有効な OUTER JOIN 式です。
{?= call func(p1, ...)}

func は、データベースサーバが受け取ることのできる任意の有効な関数呼び出しです。
{call proc(p1, ...)}

proc は、データベースサーバが受け取ることのできる任意の有効なストアドプロシージャ呼び出しです。
{fn func(p1, ...)}

func は、以下に示すいずれかの関数ライブラリです。

エスケープ構文を使用して、ODBC ドライバによって実装される関数ライブラリにアクセスできます。このライブラリには、数値、文字列、時刻、日付、システム関数が含まれています。

たとえば、次のコマンドを実行すると、データベース管理システムの種類にかかわらず現在の日付を取得できます。

```
SELECT { FN CURDATE() }
```

次の表は、ODBC ドライバによってサポートされている関数を示します。

ODBC ドライバがサポートする関数

数値関数	文字列関数	システム関数	日付/時刻関数
ABS	ASCII	DATABASE	CURDATE
ACOS	BIT_LENGTH	IFNULL	CURRENT_DATE
ASIN	CHAR	USER	CURRENT_TIME
ATAN	CHAR_LENGTH	CONVERT	CURRENT_TIMESTAMP
ATAN2	CHARACTER_LENGTH		CURTIME
CEILING	CONCAT		DAYNAME
COS	DIFFERENCE		DAYOFMONTH
COT	INSERT		DAYOFWEEK
DEGREES	LCASE		DAYOFYEAR

数値関数	文字列関数	システム関数	日付/時刻関数
EXP	LEFT		EXTRACT
FLOOR	LENGTH		HOUR
LOG	LOCATE		MINUTE
LOG10	LTRIM		MONTH
MOD	OCTET_LENGTH		MONTHNAME
PI	POSITION		NOW
POWER	REPEAT		QUARTER
RADIANS	REPLACE		SECOND
RAND	RIGHT		TIMESTAMPADD
ROUND	RTRIM		TIMESTAMPDIFF
SIGN	SOUNDEX		WEEK
SIN	SPACE		YEAR
SQRT	SUBSTRING		
TAN	UCASE		
TRUNCATE			

ODBC TIMESTAMPADD、TIMESTAMPDIFF

ODBC ドライバは TIMESTAMPADD 関数および TIMESTAMPDIFF 関数を対応するデータサーバの DATEADD 関数および DATEDIFF 関数にマップします。TIMESTAMPADD 関数と TIMESTAMPDIFF 関数の構文は次のとおりです。

```
{ fn TIMESTAMPADD( interval, integer-expr, timestamp-expr ) }
```

`integer-expr` の間隔 (`interval` 型) を `timestamp-expr` に追加して計算されたタイムスタンプを返します。`interval` の有効な値を次に示します。

```
{ fn TIMESTAMPDIFF( interval, timestamp-expr1, timestamp-expr2 ) }
```

`interval` 型の間隔を `timestamp-expr2` と `timestamp-expr1` の差を表す整数値で返します。`interval` の有効な値を次に示します。

interval	DATEADD/DATEDIFF date-part のマッピング
SQL_TSI_YEAR	YEAR
SQL_TSI_QUARTER	QUARTER
SQL_TSI_MONTH	MONTH
SQL_TSI_WEEK	WEEK
SQL_TSI_DAY	DAY

interval	DATEADD/DATEDIFF date-part のマッピング
SQL_TSI_HOUR	HOUR
SQL_TSI_MINUTE	MINUTE
SQL_TSI_SECOND	SECOND
SQL_TSI_FRAC_SECOND	MICROSECOND - DATEADD および DATEDIFF 関数はナノ秒の精度をサポートしません。

Interactive SQL

ODBC エスケープ構文は JDBC エスケープ構文と同じです。JDBC を使用する Interactive SQL では、波括弧 ({) は必ず二重にしてください。括弧の間にスペースを入れないでください。"{{" は使用できますが、"{ {" は使用できません。また、文中に改行文字を使用できません。ストアドプロシージャは Interactive SQL で解析されないため、ストアドプロシージャではエスケープ構文を使用できません。

たとえば、2013 年 2 月の週数を取得するには、Interactive SQL で次の文を実行します。

```
SELECT {{ fn TIMESTAMPDIFF(SQL_TSI_WEEK, '2013-02-01T00:00:00',
'2013-03-01T00:00:00'  ) }}
```

1.5.15 ODBC のエラー処理

ODBC のエラーは、各 ODBC 関数呼び出しからの戻り値と、SQLError 関数または SQLGetDiagRec 関数を使用してレポートされます。

SQLError 関数は、バージョン 3 よりも前の ODBC で使用されていました。バージョン 3 では、SQLError 関数は使用されなくなり、SQLGetDiagRec 関数が代わりに使用されるようになりました。

すべての ODBC 関数は、次のいずれかのステータスコードの SQLRETURN を返します。

ステータスコード	説明
SQL_SUCCESS	エラーはありません。
SQL_SUCCESS_WITH_INFO	関数は完了しましたが、SQLError を呼び出すと警告が示されます。 このステータスは、返される値が長すぎてアプリケーションが用意したバッファに入りきらない場合によく使用されます。
SQL_ERROR	関数はエラーのため完了しませんでした。SQLError を呼び出すと、エラーに関する詳細な情報を取得できます。

ステータスコード	説明
SQL_INVALID_HANDLE	パラメータとして渡された環境、接続、またはステートメントハンドルが不正です。 このステータスは、すでに解放済みのハンドルを使用した場合、あるいはハンドルが NULL ポインタである場合によく使用されます。
SQL_NO_DATA_FOUND	情報がありません。 このステータスは、カーソルからフェッチするときに、カーソルにそれ以上ローがないことを示す場合によく使用されます。
SQL_NEED_DATA	パラメータにデータが必要です。 これは、SQLParamData と SQLPutData の ODBC SDK マニュアルで説明されている高度な機能です。

あらゆる環境、接続、文のハンドルに対して、エラーまたは警告が 1 つ以上発生する可能性があります。SQLError または SQLGetDiagRec を呼び出すたびに、1 つのエラーに関する情報が返され、その情報が削除されます。SQLError または SQLGetDiagRec を呼び出してすべてのエラーを削除しなかった場合は、同じハンドルをパラメータに取る関数が次に呼び出された時点で、残ったエラーが削除されます。

SQLError の各呼び出しで、環境、接続、文に対応する 3 つのハンドルを渡します。最初の呼び出しは、SQL_NULL_HSTMT を使用して接続に関するエラーを取得しています。同様に、SQL_NULL_DBC と SQL_NULL_HSTMT を同時に使用して呼び出すと、環境ハンドルに関するエラーが取得されます。

SQLGetDiagRec を呼び出すたびに、環境、接続、または文のハンドルを渡すことができます。最初の呼び出しでは、型 SQL_HANDLE_DBC のハンドルを渡して、接続に関連するエラーを取得します。2 つ目の呼び出しでは、型 SQL_HANDLE_STMT のハンドルを渡して、直前に実行した文に関連するエラーを取得します。

エラー (SQL_ERROR 以外) があるうちは SQL_SUCCESS が返され、エラーがなくなると SQL_NO_DATA_FOUND が返されます。

例

例 1

次のコードフラグメントは SQLError とリターンコードを使用しています。

```
// ODBC 2.0
RETCODE rc;
HENV env;
HDBC dbc;
HSTMT stmt;
SDWORD err_native;
UCHAR err_state[6];
UCHAR err_msg[ 512 ];
SWORD err_ind;
rc = SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
if( rc == SQL_ERROR )
{
   printf( "Allocation failed¥n" );
   for(;;)
   {
      rc = SQLError( env, dbc, SQL_NULL_HSTMT, err_state, &err_native,
                err_msg, sizeof(err_msg), &err_ind );
      if( rc  < SQL_SUCCESS ) break;
```

```
      if( rc == SQL_NO_DATA_FOUND ) break;
      printf( "[%s:%d] %s¥n", err_state, err_native, err_msg );
   }
   return;
}
rc = SQLExecDirect( stmt,
       "DELETE FROM SalesOrderItems WHERE ID=2015",
       SQL_NTS );
if( rc == SQL_ERROR )
{
   printf( "Failed to delete items¥n" );
   for(;;)
   {
      rc = SQLError( env, dbc, stmt, err_state, &err_native,
               err_msg, sizeof(err_msg), &err_ind );
      if( rc  < SQL_SUCCESS ) break;
      if( rc == SQL_NO_DATA_FOUND ) break;
      printf( "[%s:%d] %s¥n", err_state, err_native, err_msg );
   }
   return;
}
```

例 2

次のコードフラグメントは SQLGetDiagRec とリターンコードを使用しています。

```
// ODBC 3.0
SQLRETURN rc;
SQLHDBC dbc;
SQLHSTMT stmt;
SQLSMALLINT rec;
SQLINTEGER err_native;
SQLCHAR err_state[6];
SQLCHAR err_msg[ 512 ];
SQLSMALLINT err_ind;
rc = SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
if( rc == SQL_ERROR )
{
   printf( "Failed to allocate handle¥n" );
   for( rec = 1; ; rec++ )
   {
      rc = SQLGetDiagRec( SQL_HANDLE_DBC, dbc, rec, err_state, &err_native,
               err_msg, sizeof(err_msg), &err_ind );
      if( rc  < SQL_SUCCESS ) break;
      if( rc == SQL_NO_DATA_FOUND ) break;
      printf( "[%s:%d] %s¥n", err_state, err_native, err_msg );
   }
   return;
}
rc = SQLExecDirect( stmt,
       "DELETE FROM SalesOrderItems WHERE ID=2015",
       SQL_NTS );
if( rc == SQL_ERROR )
{
   printf( "Failed to delete items¥n" );
   for( rec = 1; ; rec++ )
   {
      rc = SQLGetDiagRec( SQL_HANDLE_STMT, stmt, rec, err_state, &err_native,
               err_msg, sizeof(err_msg), &err_ind );
      if( rc  < SQL_SUCCESS ) break;
      if( rc == SQL_NO_DATA_FOUND ) break;
      printf( "[%s:%d] %s¥n", err_state, err_native, err_msg );
   }
   return;
}
```

1.6 データベースにおける Java

データベースサーバでは、データベース環境内から Java クラスを実行するメカニズムがサポートされています。データベースサーバから Java メソッドを使用すると、強力な方法でプログラミング論理をデータベースに追加できます。

データベースでの Java サポートの特長を次に示します。

- アプリケーションの異なるレイヤ (クライアント、中間層、またはサーバなど) で Java コンポーネントを再使用したり、最も意味がある場所で使用したりします。
- データベースに論理を構築する場合、Java は SQL ストアドプロシージャ言語よりも高機能な言語です。
- データベースおよびサーバの整合性、セキュリティ、堅牢性を保ちながら、データベースサーバで Java を使用することができます。

SQLJ 標準

データベース内の Java は、SQLJ Part 1 で提唱されている標準 (ANSI/INCITS 331.1-1999) に準拠しています。SQLJ Part 1 は、Java の静的メソッドを SQL ストアドプロシージャおよび関数として呼び出すための仕様です。

このセクションの内容:

データベースにおける Java の概要 [192 ページ]
SQL ストアドプロシージャ構文は、SQL から Java メソッドを呼び出せるように拡張されています。

Java のエラー処理 [193 ページ]
Java アプリケーションでのエラーによって、そのエラーを表す例外オブジェクトが生成されます (「例外のスロー」と呼ばれます)。スローされた例外がキャッチされ、アプリケーションの特定のレベルで正しく処理されない限り、その例外は Java プログラムを終了させます。

チュートリアル: データベース内の Java の使用 [193 ページ]
このチュートリアルでは、Java メソッドの作成および SQL からの呼び出しに関する手順を示します。

Java クラスをデータベースにインストールする方法 [201 ページ]
Java クラスは、単一のクラスまたは JAR としてデータベースにインストールできます。

データベース内の Java クラスの特殊な機能 [204 ページ]
次の資料では、データベース内で Java クラスを使用したときの機能について説明します。

Java VM を起動し、停止する方法 [208 ページ]
Java VM は、最初の Java オペレーションが実行されると自動的にロードされます。Java VM をマニュアルで起動または停止するには、START JAVA 文または STOP JAVA 文を使用できます。

Java VM でのシャットダウンフック [208 ページ]
データベース内で Java サポートを提供するときに使用する組み込みの Java VM クラスローダでは、アプリケーションを通じてシャットダウンフックをインストールできます。

1.6.1 データベースにおける Java の概要

SQL ストアドプロシージャ構文は、SQL から Java メソッドを呼び出せるように拡張されています。

サポート内容は次のとおりです。

データベースから Java を実行できる

外部 Java VM は、データベースサーバに代わって Java コードを実行します。

Java からデータにアクセスできる

Java コードは、データベースからデータにアクセスできます。

SQL が保持される

Java を使用しても、既存の SQL 文の動作や他の Java 以外のリレーショナルデータベースの動作は変更されません。

Java はデータベースで効果的に使用できるようないくつかの機能を提供します。

- コンパイル時のエラーチェック
- 十分に定義されたエラー処理方法論による組み込みエラー処理
- 組み込みガーベジコレクション (メモリリカバリ)
- バグを起こしやすいプログラミング技術の排除
- 高度なセキュリティ機能
- プラットフォームに依存しない Java コードの実行

Java 言語は SQL より強力です。Java はオブジェクト指向型言語であるため、その命令 (ソースコード) はクラスの形式を取ります。データベースで Java を実行するには、データベースの外部で Java 命令を作成し、それらをデータベースの外部でコンパイルし、Java 命令を保持するバイナリファイルであるコンパイル済みクラス (バイトコード) にします。

コンパイル済みクラスは、ストアドプロシージャと同じくらい簡単に同じ方法で、クライアントアプリケーションから呼び出すことができます。Java クラスには、サブジェクトに関する情報と計算論理の両方を含めることができます。たとえば、Employees クラスを作成し、Employees テーブルの操作を実行する各種のメソッドを使用して作業を行う Java コードを設計、記述し、コンパイルすることができます。Java のクラスはオブジェクトとしてデータベースにインストールし、SQL のカバー関数やプロシージャを記述して Java クラスのメソッドを呼び出します。

これらのクラスは、インストール後、ストアドプロシージャを使用してデータベースサーバで実行できます。たとえば、次の文は Java プロシージャへのインタフェースを作成するものです。

```
CREATE PROCEDURE MyMethod( )
EXTERNAL NAME 'JDBCExample.MyMethod()V'
LANGUAGE JAVA;
```

データベースサーバは、Java 開発環境ではなく、Java クラスのランタイム環境を支援するものです。Java の記述やコンパイルには、Java Development Kit (JDK) などの Java 開発環境が必要です。また、Java クラスを実行するためには Java Runtime Environment も必要です。

Java Development Kit に含まれる、Java API の一部であるクラスの多くを使用できます。また、Java 開発者が作成し、コンパイルしたクラスも使用できます。

データベースサーバは Java VM を起動します。Java VM はコンパイル済みの Java 命令を解釈し、データベースサーバに代わってそれらを実行します。データベースサーバは必要に応じて Java VM を自動的に起動するので、ユーザがわざわざ Java VM を起動したり、停止したりする必要はありません。

データベースサーバの SQL 要求プロセッサは、Java VM に呼び出されて、Java 命令を実行できるように拡張されました。このプロセッサは Java VM からの要求も処理でき、Java からのデータアクセスを可能にしました。

関連情報

Java クラスをデータベースにインストールする方法 [201 ページ]

1.6.2 Java のエラー処理

Java アプリケーションでのエラーによって、そのエラーを表す例外オブジェクトが生成されます (「例外のスロー」と呼ばれます)。スローされた例外がキャッチされ、アプリケーションの特定のレベルで正しく処理されない限り、その例外は Java プログラムを終了させます。

Java API クラスとカスタム作成のクラスは両方とも、例外をスローできます。実際、ユーザは独自の例外クラスを作成でき、それらの例外クラスはカスタム作成されたエラーのクラスをスローします。

例外が発生したメソッド本体に例外ハンドラがない場合は、例外ハンドラの検索が呼び出しスタックを継続します。呼び出しスタックの一番上に達し、例外ハンドラが見つからなかった場合は、アプリケーションを実行する Java インタプリターのデフォルトの例外ハンドラが呼び出され、プログラムが終了します。

SQL 文が Java メソッドを呼び出し、未処理の例外がスローされると、SQL エラーが生成されます。サーバメッセージウィンドウに、Java 例外の完全なテキストと Java スタックトレースが表示されます。

1.6.3 チュートリアル: データベース内の Java の使用

このチュートリアルでは、Java メソッドの作成および SQL からの呼び出しに関する手順を示します。

前提条件

次のシステム権限が必要です。

- MANAGE ANY EXTERNAL ENVIRONMENT
- SET ANY SYSTEM OPTION
- SERVER OPERATOR
- MANAGE ANY EXTERNAL OBJECT
- CREATE PROCEDURE

コンテキスト

Java クラスをコンパイルし、データベースにインストールすることで、使用できるようにする方法を説明します。また、SQL 文から、クラスとそのメンバーとメソッドにアクセスする方法についても説明します。

ここでは、Java コンパイラ (javac) や Java VM などの Java Development Kit (JDK) のインストールを完了していることを前提とします。

サンプルで使用するソースコードとバッチファイルは、*%SQLANYSAMP17%*`¥SQLAnywhere¥JavaInvoice` にあります。

1. レッスン 1: Java プログラムのコンパイル [194 ページ]
   データベースで Java を使用する場合は、まず Java コードを記述してコンパイルします。
2. レッスン 2: Java VM の選択 [195 ページ]
   Java 仮想マシン (VM) を特定するようにデータベースサーバを設定します。データベースごとに異なる Java VM を指定できるため、ALTER EXTERNAL ENVIRONMENT 文を使用して Java VM のロケーション (パス) を指定できます。
3. レッスン 3: Java クラスのインストール [197 ページ]
   Java クラスをデータベースにインストールすると、SQL で使用できるようになります。
4. レッスン 4: Java クラスのメソッドの呼び出し [199 ページ]
   クラスで Java メソッドを呼び出すラッパーとして動作するストアドプロシージャまたは関数を作成します。

1.6.3.1 レッスン 1: Java プログラムのコンパイル

データベースで Java を使用する場合は、まず Java コードを記述してコンパイルします。

前提条件

Java コンパイラ (javac) や Java Runtime Environment (JRE) などの Java Development Kit (JDK) をインストールします。

このチュートリアルの冒頭に一覧されているロールと権限を持っている必要があります。

コンテキスト

データベースサーバは、CLASSPATH 環境変数を使用して、クラスのインストールのときにファイルを検出します。

手順

1. コマンドプロンプトを開き、*%SQLANYSAMP17%*¥SQLAnywhere¥JavaInvoice フォルダに移動します。

   ```
   cd %SQLANYSAMP17%¥SQLAnywhere¥JavaInvoice
   ```

2. 次のコマンドを使用して、Java ソースコードの例をコンパイルします。

   ```
   javac Invoice.java
   ```

3. この手順は省略可能です。データベースサーバを起動する前に、コンパイル済みクラスファイルの場所が CLASSPATH 環境変数に含まれていることを確認します。使用されるのはデータベースサーバの CLASSPATH であり、Interactive SQL を実行するクライアントの CLASSPATH ではありません。次に例を示します。

   ```
   SET CLASSPATH=%SQLANYSAMP17%¥SQLAnywhere¥JavaInvoice
   ```

結果

javac コマンドは、データベースにインストールできるクラスファイルを作成します。

次のステップ

次のレッスンに進みます。

タスクの概要: チュートリアル: データベース内の Java の使用 [193 ページ]

次のタスク: レッスン 2: Java VM の選択 [195 ページ]

1.6.3.2 レッスン 2: Java VM の選択

Java 仮想マシン (VM) を特定するようにデータベースサーバを設定します。データベースごとに異なる Java VM を指定できるため、ALTER EXTERNAL ENVIRONMENT 文を使用して Java VM のロケーション (パス) を指定できます。

前提条件

このチュートリアルのこれまでのレッスンを完了している必要があります。

このチュートリアルの冒頭に一覧されているロールと権限を持っている必要があります。

コンテキスト

Java Runtime Environment (JRE) がインストールされていない場合、バージョン 1.6 以降 (JRE 6 以降) であればどの Java JRE でもインストールして使用できます。ほとんどの Java インストーラでは、JAVA_HOME または JAVAHOME のいずれかの環境変数を設定しますどちらの環境変数もない場合は、いずれかを手動で作成し、Java VM のルートディレクトリを示すように設定できます。ただし、ALTER EXTERNAL ENVIRONMENT 文を使用している場合は、この設定は必要ありません。

手順

1. Interactive SQL を使用してパーソナルデータベースサーバを起動し、サンプルデータベースに接続します。

   ```
   dbisql -c "DSN=SQL Anywhere 17 Demo;PWD=sql"
   ```

2. この手順は省略可能です。次のような文を実行します。

   ```
   ALTER EXTERNAL ENVIRONMENT JAVA
   LOCATION 'c:¥¥jdk1.8.0¥¥jre¥¥bin¥¥java.exe';
   ```

   Java VM の場所が ALTER EXTERNAL ENVIRONMENT JAVA 文の LOCATION 句を使用して指定されており、その場所が正しくない場合、データベースサーバは Java VM をロードしません。

   Java VM の場所が LOCATION 句を使用して指定されない場合、データベースサーバは次のように Java VM の場所を検索します。

   - JAVA_HOME 環境変数を確認します。
   - JAVAHOME 環境変数を確認します。
   - システム PATH を確認します。
   - VM が見つからなかった場合は、エラーを返します。

3. この手順は省略可能です。Java VM の起動に必要な追加コマンドラインオプションを指定するには、java_vm_options データベースオプションを使用します。`java-options` を有効な Java VM オプションの文字列に置き換えます。

   ```
   SET OPTION PUBLIC.java_vm_options=java-options;
   ```

4. START JAVA 文を使用して、Java VM を起動します。

   ```
   START JAVA;
   ```

   この文は Java VM のプリロードを試みます。データベースサーバが Java VM を検索して起動できない場合、エラーメッセージが発行されます。必要な場合、データベースサーバが自動的に Java VM をロードするので、この文はオプションです。

結果

ALTER EXTERNAL ENVIRONMENT JAVA 文の LOCATION 句は、Java VM の場所を示します。START JAVA 文は、Java VM をロードします。

次のステップ

次のレッスンに進みます。

タスクの概要: チュートリアル: データベース内の Java の使用 [193 ページ]

前のタスク: レッスン 1: Java プログラムのコンパイル [194 ページ]

次のタスク: レッスン 3: Java クラスのインストール [197 ページ]

1.6.3.3 レッスン 3: Java クラスのインストール

Java クラスをデータベースにインストールすると、SQL で使用できるようになります。

前提条件

このチュートリアルのこれまでのレッスンを完了している必要があります。

このチュートリアルの冒頭に一覧されているロールと権限を持っている必要があります。

コンテキスト

データベースサーバは、CLASSPATH 環境変数を使用して、クラスのインストールのときにファイルを検出します。INSTALL JAVA 文にリスト表示されているファイルがデータベースサーバの CLASSPATH 環境変数によって指定されたディレクトリまたは ZIP ファイルにある場合、サーバは正常にファイルを検索し、クラスをインストールします。

手順

1. CLASSPATH 環境変数が最初のレッスンから正しく設定されているかどうかを確認します。`Invoice.class` ファイルを含むディレクトリが含まれている必要があります。

   ```
   CALL sa_split_list( xp_getenv( 'CLASSPATH' ), ';' );
   ```

2. Interactive SQL を使用して、次のような文を実行します。コンパイル済みクラスファイルの場所への `path` は、データベースサーバの CLASSPATH を使用して検索できる場合には必要ありません。`path` は、指定された場合、データベースサーバにアクセスできる必要があります。

   ```
   INSTALL JAVA NEW
   FROM FILE 'path¥¥Invoice.class';
   ```

この手順でエラーが発生した場合、JAVA VM パスが正しく設定されていることを確認してください。次の文は、データベースサーバが使用する JAVA VM 実行ファイルへのパスを返します。

```
SELECT db_property('JavaVM');
```

この結果が NULL の場合、パスが正しく設定されていないことになります。ALTER EXTERNAL ENVIRONMENT JAVA LOCATION 文でパスを設定した場合、現在の設定を次のように判別できます。

```
SELECT location FROM SYS.SYSEXTERNENV WHERE name = 'java';
```

パスが設定されていない場合、またはパスの設定が間違っている場合は、前のレッスンの手順 2 に戻ってください。

結果

クラスがサンプルデータベースにインストールされます。

この後クラスファイルに行った変更は、データベースでのクラスファイルのコピーに自動的に反映されるわけではありません。クラスファイルが再コンパイルされるたびに、INSTALL JAVA UPDATE 文を使用して、クラスファイルをデータベースに再ロードします。

次のステップ

次のレッスンに進みます。

タスクの概要: チュートリアル: データベース内の Java の使用 [193 ページ]

前のタスク: レッスン 2: Java VM の選択 [195 ページ]

次のタスク: レッスン 4: Java クラスのメソッドの呼び出し [199 ページ]

関連情報

Java クラスをデータベースにインストールする方法 [201 ページ]

1.6.3.4 レッスン 4: Java クラスのメソッドの呼び出し

クラスで Java メソッドを呼び出すラッパーとして動作するストアドプロシージャまたは関数を作成します。

前提条件

このチュートリアルのこれまでのレッスンを完了している必要があります。

このチュートリアルの冒頭に一覧されているロールと権限を持っている必要があります。

コンテキスト

Invoice 例からのコンパイル済みメソッドを含む Java クラスファイルがデータベースにロードされました。

手順

1. サンプルクラスの Invoice.main メソッドを呼び出す次の SQL ストアドプロシージャを作成します。

   ```
   CREATE PROCEDURE InvoiceMain( IN arg1 CHAR(50) )
   EXTERNAL NAME 'Invoice.main([Ljava/lang/String;)V'
   LANGUAGE JAVA;
   ```

   このストアドプロシージャは、Java メソッドのラッパーとして動作します。
2. ストアドプロシージャを呼び出して、Java メソッドを呼び出します。

   ```
   CALL InvoiceMain( 'to you' );
   ```

   データベースサーバメッセージログに "Hello to you" というメッセージが表示されるのを確認できます。データベースサーバによって、System.out から出力データがリダイレクトされています。
3. 次のストアドプロシージャは、Invoice クラスの Java メソッドに引数を渡して、戻り値を取得する方法について説明します。Java ソースコードを確認すると、Invoice クラスの `init` メソッドは文字列引数と double 引数の両方を取ることがわかります。文字列引数は `Ljava/lang/String;` を使用して指定されます。double 引数は `D` を使用して指定されます。メソッドは void を返し、これは `V` を使用して右側のカッコの後ろに指定されます。

   ```
   CREATE PROCEDURE init( IN arg1 CHAR(50),
                         IN arg2 DOUBLE,
                         IN arg3 CHAR(50),
                         IN arg4 DOUBLE)
   EXTERNAL NAME 'Invoice.init(Ljava/lang/String;DLjava/lang/String;D)V'
   LANGUAGE JAVA;
   ```
4. 次の関数は引数をとらない Java メソッドを呼び出し、double (`D`) または文字列 (`Ljava/lang/String;`) を返します。

   ```
   CREATE FUNCTION rateOfTaxation()
   ```

```
RETURNS DOUBLE
EXTERNAL NAME 'Invoice.rateOfTaxation()D'
LANGUAGE JAVA;
CREATE FUNCTION totalSum()
RETURNS DOUBLE
EXTERNAL NAME 'Invoice.totalSum()D'
LANGUAGE JAVA;
CREATE FUNCTION getLineItem1Description()
RETURNS CHAR(50)
EXTERNAL NAME 'Invoice.getLineItem1Description()Ljava/lang/String;'
LANGUAGE JAVA;
CREATE FUNCTION getLineItem1Cost()
RETURNS DOUBLE
EXTERNAL NAME 'Invoice.getLineItem1Cost()D'
LANGUAGE JAVA;
CREATE FUNCTION getLineItem2Description()
RETURNS CHAR(50)
EXTERNAL NAME 'Invoice.getLineItem2Description()Ljava/lang/String;'
LANGUAGE JAVA;
CREATE FUNCTION getLineItem2Cost()
RETURNS DOUBLE
EXTERNAL NAME 'Invoice.getLineItem2Cost()D'
LANGUAGE JAVA;
```

5. 次は、Invoice クラスの `init` メソッドのラッパーとして動作するストアドプロシージャに対するサンプルコールについて説明します。

```
CALL init( 'Shirt', 10.00, 'Jacket', 25.00 );
```

6. 次の SELECT 文は、Invoice クラスの他のメソッドをいくつか呼び出します。

```
SELECT getLineItem1Description() as Item1,
       getLineItem1Cost() as Item1Cost,
       getLineItem2Description() as Item2,
       getLineItem2Cost() as Item2Cost,
       rateOfTaxation() as TaxRate,
       totalSum() as Cost;
```

SELECT 文が 6 つのカラムを返す場合。

Item1	Item1Cost	Item2	Item2Cost	TaxRate	Cost
Shirt	10	Jacket	25	0.15	40.25

結果

Java クラスのメソッドのラッパーとして動作するストアドプロシージャまたは関数が作成されました。これらのレッスンでは、Java メソッドの作成および SQL からの呼び出しに関する手順を示しました。

タスクの概要: チュートリアル: データベース内の Java の使用 [193 ページ]

前のタスク: レッスン 3: Java クラスのインストール [197 ページ]

1.6.4 Java クラスをデータベースにインストールする方法

Java クラスは、単一のクラスまたは JAR としてデータベースにインストールできます。

単一のクラス

単一のクラスを、コンパイル済みクラスファイルからデータベースにインストールできます。通常、クラスファイルには拡張子 `.class` が付いています。

JAR ファイル

一連のクラスが、圧縮された JAR ファイルと圧縮されていない JAR ファイルのいずれかに保持されている場合は、一度に全部をインストールできます。通常、JAR ファイルには拡張子 `.jar` または `.zip` が付いています。データベースサーバは、JAR ユーティリティで作成されたすべての圧縮 JAR ファイルと、その他の JAR 圧縮スキームをサポートしています。

このセクションの内容:

クラスファイルの作成 [201 ページ]
データベースで Java を使用する最初のステップは、Java アプリケーションまたはクラスファイルを作成することです。

クラスファイルのインストール [202 ページ]
Java クラスをデータベースにインストールし、データベースで使用できるようにします。

JAR ファイルのインストール [202 ページ]
JAR ファイルをデータベースにインストールし、データベースで使用できるようにします。

クラスと JAR ファイルの更新 [203 ページ]
SQL Central を使用して、クラスと JAR ファイルを更新されたコピーに置換します。

1.6.4.1 クラスファイルの作成

データベースで Java を使用する最初のステップは、Java アプリケーションまたはクラスファイルを作成することです。

それぞれの手順の詳細は、Java 開発ツールを使用しているかどうかによって異なりますが、独自のクラスを作成する手順は一般的に次のようになっています。

1. クラスを定義します。
   クラスを定義する Java コードを記述します。
2. クラスに名前を付けて保存します。
   クラス宣言 (Java コード) を拡張子 `.java` が付いたファイルに保存します。ファイル名とクラス名が同じで、大文字と小文字の使い分けが一致していることを確認します。
   たとえば、クラス Utility は、ファイル `Utility.java` に保存されます。
3. クラスをコンパイルします。
   この手順では、Java コードを含むクラス宣言を、バイトコードを含む新しい個別のファイルにします。新しいファイルの名前は、Java コードファイル名と同じですが拡張子 `.class` が付きます。コンパイルされた Java クラスは、コンパイルを行ったプラットフォームやランタイム環境のオペレーティングシステムに関係なく、Java Runtime Environment で実行することができます。

1.6.4.2 クラスファイルのインストール

Java クラスをデータベースにインストールし、データベースで使用できるようにします。

前提条件

クラスをインストールするには、MANAGE ANY EXTERNAL OBJECT システム権限が必要です。

インストールするクラスファイルのパスとファイル名を確認します。

手順

1. *SQL CentralSQL Anywhere17* プラグインを使用してデータベースに接続します。
2. **外部環境**フォルダを開きます。
3. このフォルダの中にある *Java* フォルダを開きます。
4. 右ウィンドウ枠を右クリックし、**新規** > *Java* **クラス** をクリックします。
5. ウィザードの指示に従います。

結果

クラスはデータベースにインストールされており、使用可能です。

1.6.4.3 JAR ファイルのインストール

JAR ファイルをデータベースにインストールし、データベースで使用できるようにします。

前提条件

JAR をインストールするには、MANAGE ANY EXTERNAL OBJECT システム権限が必要です。

インストールする JAR ファイルのパスとファイル名を確認します。JAR ファイルには、JAR または ZIP という拡張子を付けることができます。各 JAR ファイルは、データベースに名前を持っています。通常は、JAR ファイルと同じ名前で拡張子のないものを使用します。たとえば、JAR ファイル `myjar.zip` をインストールした場合は、JAR 名を `myjar` にします。

コンテキスト

関連するクラスセットをまとめて "パッケージ" に集め、1 つまたは複数のパッケージを JAR ファイルに保管することは、便利で一般的に行われている方法です。

手順

1. *SQL Central* でデータベースに接続します。
2. **外部環境**フォルダを開きます。
3. このフォルダの中にある *Java* フォルダを開きます。
4. 右ウィンドウ枠を右クリックし、**新規** ▶ *JAR ファイル* をクリックします。
5. ウィザードの指示に従います。

結果

JAR ファイルはデータベースにインストールされており、使用可能です。

1.6.4.4 クラスと JAR ファイルの更新

SQL Central を使用して、クラスと JAR ファイルを更新されたコピーに置換します。

前提条件

クラスまたは JAR を更新するには、MANAGE ANY EXTERNAL OBJECT システム権限が必要です。

より新しいバージョンのコンパイル済みクラスファイルまたは JAR ファイルを使用できるようにします。

コンテキスト

新しい定義を使用するのは、クラスのインストール後に設定された新しい接続か、クラスのインストール後最初にそのクラスを使用する接続だけです。Java VM がクラス定義をロードすると、クラス定義は接続が閉じるまでメモリに保存されます。現在の接続で Java クラスまたはクラスを基にしたオブジェクトを使用している場合、新しいクラス定義を使用するには接続をいったん切断し、その後再接続します。

手順

1. *SQL CentralSQL Anywhere17* プラグインを使用してデータベースに接続します。
2. **外部環境**フォルダを開きます。
3. このフォルダの中にある *Java* フォルダを開きます。
4. 更新するクラスまたは JAR ファイルが含まれたサブフォルダを探します。
5. そのクラスまたは JAR ファイルをクリックし、**ファイル** > **更新** をクリックします。
6. **更新**ウィンドウで、更新するクラスまたは JAR ファイルのロケーションと名前を指定します。**参照**をクリックして検索することもできます。

結果

新しい定義を使用するのは、クラスのインストール後に設定された新しい接続か、クラスのインストール後最初にそのクラスを使用する接続だけです。Java VM がクラス定義をロードすると、クラス定義は接続が閉じるまでメモリに保存されます。現在の接続で Java クラスまたはクラスを基にしたオブジェクトを使用している場合、新しいクラス定義を使用するには接続をいったん切断し、その後再接続します。

次のステップ

Java クラスまたは JAR ファイルは、それぞれクラス名または JAR ファイル名を右クリックし、**更新**を選択して更新することもできます。

また、**プロパティ**ウィンドウの**一般**タブで**すぐに更新**をクリックして更新することもできます。

1.6.5 データベース内の Java クラスの特殊な機能

次の資料では、データベース内で Java クラスを使用したときの機能について説明します。

このセクションの内容:

main メソッドを呼び出す方法 [205 ページ]
通常 Java アプリケーションを (データベース外で) 起動するには、main メソッドを持つクラス上で Java VM を起動します。

Java アプリケーションでのスレッド [206 ページ]
java.lang.Thread パッケージの機能を使用すると、Java アプリケーションでマルチスレッドを使用できます。

No Such Method Exception [206 ページ]
Java メソッドを呼び出す際に不正な数の引数を指定したり、不正なデータ型を使用した場合、Java VM から `java.lang.NoSuchMethodException` エラーが返されます。引数の数と型を確認してください。

Java メソッドから結果セットを返す方法 [206 ページ]

呼び出しを行う環境に結果セットを返す Java メソッドを書き、LANGUAGE JAVA 句と EXTERNAL NAME 句で宣言された SQL ストアドプロシージャ内にそのメソッドをラップします。

Java からストアドプロシージャを経由して返される値 [207 ページ]
EXTERNAL NAME 句と LANGUAGE JAVA 句を使用して作成したストアドプロシージャは、Java メソッドのラッパーとして使用できます。Java メソッドからストアドプロシージャの SQL OUT パラメータまたは INOUT パラメータにパラメータデータを返すための特殊な技法があります。

Java のセキュリティ管理 [208 ページ]
Java には、セキュリティマネージャが用意されています。これを使用すると、ファイルアクセスやネットワークアクセスなど、セキュリティが問題となるアプリケーションの機能に対するユーザのアクセスを制御できます。

1.6.5.1 main メソッドを呼び出す方法

通常 Java アプリケーションを (データベース外で) 起動するには、main メソッドを持つクラス上で Java VM を起動します。

たとえば *%SQLANYSAMP17%*`¥SQLAnywhere¥JavaInvoice¥Invoice.java` ファイルの Invoice クラスには main メソッドがあります。次のようなコマンドを使用して、このクラスをコマンドラインから実行すると、main メソッドが実行されます。

```
java Invoice
```

SQL からのクラスの main メソッドの呼び出し

Java で作成されたクラスの main メソッドを呼び出すには、次の手順を実行します。

1. 引数として文字列配列を指定し、Java main メソッドを宣言します。

   ```
   public static void main( java.lang.String args[] )
   {
   ...
   }
   ```

2. このメソッドをラップするストアドプロシージャを作成します。

   ```
   CREATE PROCEDURE JavaMain( IN arg CHAR(50) )
   EXTERNAL NAME 'JavaClass.main([Ljava/lang/String;)V'
   LANGUAGE JAVA;
   ```

3. SQL CALL 文を使用して main メソッドを呼び出します。

   ```
   CALL JavaMain( 'Hello world' );
   ```

   SQL 言語の制限により、渡せるのは 1 つの文字列のみです。

1.6.5.2 Java アプリケーションでのスレッド

java.lang.Thread パッケージの機能を使用すると、Java アプリケーションでマルチスレッドを使用できます。

Java アプリケーション内のスレッドは、同期、中断、再開、一時停止、または停止することができます。

1.6.5.3 No Such Method Exception

Java メソッドを呼び出す際に不正な数の引数を指定したり、不正なデータ型を使用した場合、Java VM から `java.lang.NoSuchMethodException` エラーが返されます。引数の数と型を確認してください。

関連情報

レッスン 4: Java クラスのメソッドの呼び出し [199 ページ]

1.6.5.4 Java メソッドから結果セットを返す方法

呼び出しを行う環境に結果セットを返す Java メソッドを書き、LANGUAGE JAVA 句と EXTERNAL NAME 句で宣言された SQL ストアドプロシージャ内にそのメソッドをラップします。

Java メソッドから結果セットを返すには、次のタスクを実行します。

1. パブリッククラスで、Java メソッドが public と static として宣言されていることを確認します。
2. メソッドが返すと思われる各結果セットについて、そのメソッドが java.sql.ResultSet[] 型のパラメータを持っていることを確認します。これらの結果セットパラメータは、必ずパラメータリストの最後になります。
3. このメソッドでは、まず java.sql.ResultSet のインスタンスを作成して、それを ResultSet[] パラメータの 1 つに割り当てます。
4. EXTERNAL NAME 句と LANGUAGE JAVA 句を使用して、SQL ストアドプロシージャを作成します。この型のプロシージャは、Java メソッドのラッパーです。結果セットを返す他のプロシージャと同じ方法で、SQL プロシージャの結果セット上でカーソルを使用することができます。

例

次に示す簡単なクラスには 1 つのメソッドがあり、そのメソッドはクエリを実行して、呼び出しを行った環境に結果セットを返します。

```
import java.sql.*;
public class MyResultSet
{
  public static void return_rset( ResultSet[] rset1 )
       throws SQLException
  {
    Connection conn = DriverManager.getConnection( "jdbc:default:connection" );
    Statement stmt = conn.createStatement( );
    ResultSet rset = stmt.executeQuery(
```

```
                        "SELECT Surname " +
                        "FROM Customers" );
    rset1[0] = rset;
  }
}
```

結果セットを公開するには、そのプロシージャから返された結果セットの数と Java メソッドのシグネチャを指定する CREATE PROCEDURE 文を使用します。

結果セットを指定する CREATE PROCEDURE 文は、次のように定義します。

```
CREATE PROCEDURE result_set( )
  RESULT ( SurName person_name_t )
  DYNAMIC RESULT SETS 1
  EXTERNAL NAME 'MyResultSet.return_rset([Ljava/sql/ResultSet;)V'
  LANGUAGE JAVA;
```

結果セットを返す SQL プロシージャでカーソルを開くのと同じように、このプロシージャ上でカーソルを開くことができます。

文字列 `([Ljava/sql/ResultSet;)V` は Java メソッドのシグネチャで、パラメータおよび戻り値の個数と型を簡潔に文字で表現したものです。

関連情報

Java から結果セットを返す方法 [242 ページ]

1.6.5.5 Java からストアドプロシージャを経由して返される値

EXTERNAL NAME 句と LANGUAGE JAVA 句を使用して作成したストアドプロシージャは、Java メソッドのラッパーとして使用できます。Java メソッドからストアドプロシージャの SQL OUT パラメータまたは INOUT パラメータにパラメータデータを返すための特殊な技法があります。

Java は、INOUT または OUT パラメータの明示的なサポートはしていません。ただし、パラメータの配列は使用できます。たとえば、整数の OUT パラメータを使用するには、1 つの整数だけの配列を作成します。

```
public class Invoice
{
  public static boolean testOut( int[] param )
  {
    param[0] = 123;
    return true;
  }
}
```

次のプロシージャでは、testOut メソッドを使用します。

```
CREATE PROCEDURE testOut( OUT p INTEGER )
EXTERNAL NAME 'Invoice.testOut([I)Z'
LANGUAGE JAVA;
```

文字列 `([I)Z` は Java メソッドのシグネチャで、メソッドが単一のパラメータを持ち、このパラメータが整数の配列であり、ブール値を返すことを示しています。OUT または INOUT パラメータとして使用するメソッドパラメータが、OUT または INOUT パラメータの SQL データ型に対応する Java データ型の配列になるように、メソッドを定義します。

これをテストするには、初期化されていない変数を使用してストアドプロシージャを呼び出します。

```
CREATE VARIABLE zap INTEGER;
CALL testOut( zap );
SELECT zap;
```

結果セットは `123` です。

1.6.5.6 Java のセキュリティ管理

Java には、セキュリティマネージャが用意されています。これを使用すると、ファイルアクセスやネットワークアクセスなど、セキュリティが問題となるアプリケーションの機能に対するユーザのアクセスを制御できます。

1.6.6 Java VM を起動し、停止する方法

Java VM は、最初の Java オペレーションが実行されると自動的にロードされます。Java VM をマニュアルで起動または停止するには、START JAVA 文または STOP JAVA 文を使用できます。

Java オペレーションを実行する準備として、明示的に Java VM をロードする場合は、次の文を実行します。

```
START JAVA;
```

Java を使用していないときに STOP JAVA 文を実行すると、Java VM をアンロードできます。構文は次のとおりです。

```
STOP JAVA;
```

1.6.7 Java VM でのシャットダウンフック

データベース内で Java サポートを提供するときに使用する組み込みの Java VM クラスローダでは、アプリケーションを通じてシャットダウンフックをインストールできます。

これらのシャットダウンフックは、JVM ランタイムでアプリケーションによってインストールされるシャットダウンフックによく似ています。データベース内の Java サポートを使用している接続が STOP JAVA 文を実行するか、または切断する場合、その接続用のクラスローダは、アンロードの前に、その特定の接続用にインストールされているすべてのシャットダウンフックを実行します。データベース内にすべての Java クラスをインストールするデータベースアプリケーション内の通常の Java については、シャットダウンフックのインストールは必須でありません。クラスローダシャットダウンフックは、十分な注意を払って使用する必要があり、Java を停止する特定の接続用に割り当てられたシステム全体のリソースをクリーンアップするためのみに使用するべきです。また、jdbc:default 接続はクラスローダシャットダウンフックが呼び出される前にすでに閉じられているので、シャットダウンフック内では jdbc:default JDBC 要求は許可されません。

Java VM クラスローダのあるシャットダウンフックをインストールするには、アプリケーションで Java コンパイラクラスパス内に `sajvm.jar` を含む必要があり、また、次のようなコードを実行する必要があります。

```
SDHookThread hook = new SDHookThread( ... );
ClassLoader classLoader = Thread.currentThread( ).getContextClassLoader( );
((ianywhere.sa.jvm.SAClassLoader)classLoader).addShutdownHook( hook );
```

SDHookThread クラスは標準 Thread クラスを拡張したものであり、前述のコードは現在の接続用のクラスローダによってロードされたクラスから実行される必要があります。データベース内にインストールされたクラスと、その後に外部環境呼び出し経由で呼び出されたクラスは、正しい Java VM クラスローダによって自動的に実行されます。

Java VM クラスローダリストからシャットダウンフックを削除するには、アプリケーションで次のようなコードを実行する必要があります。

```
ClassLoader classLoader = Thread.currentThread( ).getContextClassLoader( );
((ianywhere.sa.jvm.SAClassLoader)classLoader).removeShutdownHook( hook );
```

前述のコードは、現在の接続用のクラスローダによってロードされたクラスから実行される必要があります。

1.7 XS JavaScript アプリケーションプログラミング

SQL Anywhere XS JavaScript ドライバは、SQL Anywhere データベースへの接続、SQL クエリの発行、結果セットの取得に使用できます。

SQL Anywhere XS JavaScript ドライバを使用すると、JavaScript 環境からデータベースと対話できます。SQL Anywhere XS API は、SAPHANA XS エンジンで提供されている SAP HANA XS JavaScript Database API に基づいています。様々なバージョンの Node.js. のドライバが使用可能です。

XS JavaScript ドライバは、SQL Anywhere 外部環境サポートを使用して JavaScript で記述したサーバ側アプリケーションもサポートしています。

> **注記**
>
> XS JavaScript ドライバに加え、最低限の機能を備えた軽量の Node.js ドライバを使用すると、小さな結果セットを処理できます。このドライバを使用する Node.js アプリケーションプログラミングについては、マニュアルの別の場所に記載されています。 lightweight ドライバは、小さな結果セットを取得するためにデータベースサーバに簡単にすばやく接続する必要がある単純な Web アプリケーションに最適です。ただし、JavaScript アプリケーションが大きな結果を処理する必要がある場合、制御権限を拡大する必要がある場合、またはより完全なアプリケーションプログラミングインタフェースにアクセスする必要がある場合、XS JavaScript ドライバを使用する必要があります。

Node.js をコンピュータにインストールする必要があります。Node.js 実行ファイルを格納しているフォルダは、PATH に含まれている必要があります。Node.js ソフトウェアは nodejs.org で入手できます。

Node.js がドライバを探す場合、NODE_PATH 環境変数が XS JavaScript ドライバの場所を含んでいることを確認します。次に、Microsoft Windows の例を示します。

```
SET NODE_PATH=%SQLANY17%¥Node
```

> **注記**
>
> Mac OS X 10.11 上で、SQLANY_API_DLL 環境変数を `libdbcapi_r.dylib` へのフルパスに設定します。

次に、XS JavaScript ドライバを使用する単純な Node.js アプリケーションを示します。

```
var sqla = require( 'sqlanywhere-xs' );
var cstr = { Server     : 'demo',
             UserID     : 'DBA',
             Password   : 'sql'
           };
var conn = sqla.createConnection( cstr );
conn.connect();
console.log( 'Connected' );
stmt = conn.prepareStatement( "SELECT * FROM Customers" );
stmt.execute();
var result = stmt.getResultSet();
while ( result.next() )
{
    console.log( result.getString(1) +
            " " + result.getString(3) +
            " " + result.getString(2) );
}
conn.disconnect();
console.log( 'Disconnected' );
```

このプログラムはサンプルデータベースに接続し、SQL SELECT 文を実行し、結果セットの最初の 3 つのカラム (各顧客の ID、GivenName、Surname) のみを表示します。その後、データベースからの接続を切断します。

この JavaScript コードがファイル `xs-sample.js` に格納されていたとします。このプログラムを実行するには、コマンドプロンプトを開き、次の文を実行します。NODE_PATH 環境変数が適切に設定されていることを確認します。

```
node xs-sample.js
```

次の JavaScript では、準備文とバッチの使用例を示します。この例では、データベーステーブルを作成し、そこに 100 個の正の整数を格納します。

```
var sqla = require( 'sqlanywhere-xs' );
var cstr = { Server     : 'demo',
             UserID     : 'DBA',
             Password   : 'sql'
           };
var conn = sqla.createConnection( cstr );
conn.connect();
conn.prepareStatement( "CREATE OR REPLACE TABLE mytable( col1 INT)" ).execute();
var stmt = conn.prepareStatement( "INSERT INTO mytable VALUES(?)" );
var BATCHSIZE = 100;
stmt.setBatchSize(BATCHSIZE);
for ( var i = 1; i <= BATCHSIZE; i++ )
{
    stmt.setInteger( 1, i );
    stmt.addBatch();
}
stmt.executeBatch();
stmt.close();
conn.commit();
conn.disconnect();
```

次の JavaScript では、準備文と例外処理の使用例を示します。`getcustomer` 関数は、指定した顧客のテーブルローに対応するハッシュまたはエラーメッセージを返します。

```
function getcustomer( conn, customer_id )
{
    try
    {
        var query = 'SELECT * FROM Customers WHERE ID=?';

        var pstmt = conn.prepareStatement(query);
        pstmt.setInteger( 1, customer_id );
        var rs = pstmt.executeQuery();
        if ( rs.next() )
        {
            return(
            {
                id          : rs.getInteger(1),
                surname     : rs.getString(2),
                givenname   : rs.getString(3),
                street      : rs.getString(4),
                city        : rs.getString(5),
                state       : rs.getString(6),
                country     : rs.getString(7),
                postalcode  : rs.getString(8),
                phone       : rs.getString(9),
                companyname : rs.getString(10)
            } );
        }
        else
        {
            return 'No customer with ID=' + customer_id;
        }
    }
    catch (ex)
    {
        return ex.message;
    }
    finally
    {
        if( pstmt )
        {
            pstmt.close();
        }
    }
}
var sqla = require( 'sqlanywhere-xs' );
var cstr = { Server     : 'demo',
             UserID     : 'DBA',
             Password   : 'sql'
            };
var conn = sqla.createConnection( cstr );
conn.connect();
for ( var i = 200; i < 225; i++ )
{
    console.log( getcustomer( conn, i ) );
}
conn.close();
conn.disconnect();
```

注記

API リファレンスマニュアルをお探しですか。マニュアルをローカルにインストールした場合は、Windows のスタートメニューを使用してアクセスするか (Microsoft Windows)、`C:¥Program Files¥SQL Anywhere 17¥Documentation` にナビゲートします。

また、DocCommentXchange の Web で、SAP SQL Anywhere API リファレンスマニュアルにアクセスすることもできます。http://dcx.sap.com

関連情報

nodejs.org

1.8 JDBC サポート

JDBC は、Java アプリケーション用のコールレベルインタフェースです。JDBC を使用すると、さまざまなリレーショナルデータベースに同一のインタフェースでアクセスできます。さらに、高いレベルのツールとインタフェースを構築するため基盤にもなります。

JDBC は Java の標準部分になっており、JDK に含まれています。

ソフトウェアには、Type 2 ドライバである SQL Anywhere JDBC のドライバが含まれています。

また、ソフトウェアでは、SAP から利用できる jConnect という pure Java の JDBC ドライバもサポートされています。

JDBC は、クライアント側のアプリケーションプログラミングインタフェースとして使用することもできますし、データベース機能で Java を使用することで、データベースサーバ内で JDBC を使用してデータにアクセスすることもできます。

このセクションの内容:

JDBC アプリケーション [213 ページ]
JDBC API を使用してデータベースに接続する Java アプリケーションを開発できます。データベースソフトウェアに含まれる複数のアプリケーションでは JDBC が使用されます。

JDBC ドライバ [215 ページ]
SQL Anywhere JDBC ドライバと jConnect ドライバがサポートされています。

SQL Anywhere JDBC ドライバ [216 ページ]
SQL Anywhere JDBC ドライバには、pure Java である jConnect JDBC ドライバに比べてパフォーマンスや機能の面で利点があるため、その使用が推奨されます。ただし、SQL Anywhere JDBC ドライバでは pure Java ソリューションは提供されません。

jConnect JDBC ドライバ [218 ページ]
アプレットから JDBC を使用する場合は、jConnect JDBC ドライバを介してデータベースに接続してください。

JDBC プログラムの構造 [223 ページ]
通常、JDBC アプリケーションはデータベースに接続し、1 つ以上の SQL 文を実行し、結果セットを処理した後、データベースから切断します。

クライアント側 JDBC アプリケーションとサーバ側 JDBC アプリケーションの違い [224 ページ]
クライアント側 JDBC アプリケーションとサーバ側 JDBC アプリケーションには、わずかな違いがあります。

クライアント側サンプル JDBS アプリケーション [225 ページ]

通常の JDBC アプリケーションは、データベースサーバに接続し、SQL クエリを発行し、複数の結果セットを処理してから終了します。

サーバ側サンプル JDBS アプリケーション [229 ページ]
通常の JDBC アプリケーションは、データベースサーバに接続し、SQL クエリを発行し、複数の結果セットを処理してから終了します。

JDBC 接続についての注意 [232 ページ]
クライアント側の JDBC とサーバ側の JDBC に違いがあることを認識してください。ここでは、オートコミットの動作や独立性レベルなどの各側面を説明します。

JDBC を使用したサーバ側データアクセス [234 ページ]
SQL から呼び出し可能な Java メソッドとして実装されたデータベーストランザクションロジックには、従来の SQL ストアドプロシージャに比べて大きな利点があります。

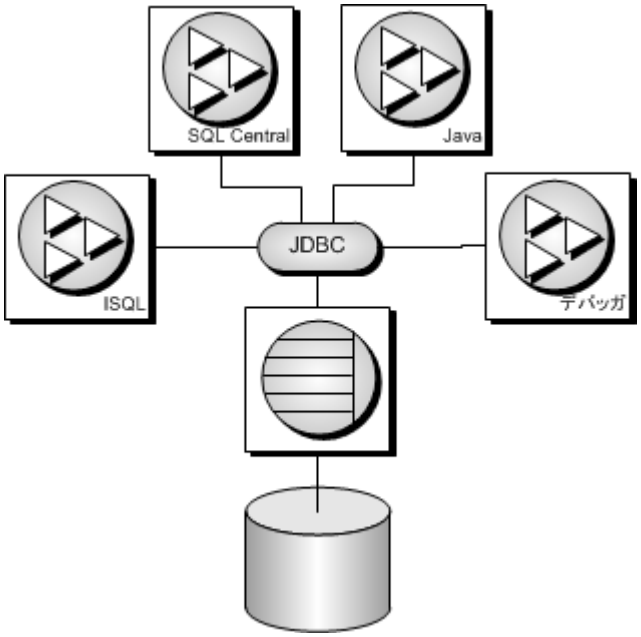
JDBC コールバック [244 ページ]
SQL Anywhere JDBC ドライバでは、2 つの非同期コールバックをサポートしています。1 つは SQL MESSAGE 文を処理し、もう 1 つはファイル転送要求を検証します。これらは、ODBC ドライバでサポートされているコールバックに似ています。

JDBC エスケープ構文 [247 ページ]
JDBC エスケープ構文は、InteractiveSQL を含む JDBC アプリケーションで使用できます。エスケープ構文を使用して、使用しているデータベース管理システムとは関係なくストアドプロシージャを呼び出すことができます。

SQL Anywhere JDBC API のサポート [251 ページ]
java.sql.Blob インタフェースの一部のオプションメソッドは、SQL Anywhere JDBC ドライバではサポートされていません。

1.8.1 JDBC アプリケーション

JDBC API を使用してデータベースに接続する Java アプリケーションを開発できます。データベースソフトウェアに含まれる複数のアプリケーションでは JDBC が使用されます。



JDBC はクライアントアプリケーションからとデータベース内からの両方で使用できます。JDBC を使用する Java クラスは、データベースにプログラミング論理を組み込むための、SQL ストアドプロシージャに代わるさらに強力な方法です。

JDBC は Java アプリケーションを操作するための SQL インタフェースです。Java からリレーショナルデータにアクセスするには、JDBC 呼び出しを使用します。

クライアントアプリケーションという用語は、ユーザのコンピュータで動作するアプリケーションを指す場合と、中間層アプリケーションサーバで動作する論理を指す場合があります。

具体例では、JDBC アプリケーションに固有の機能を示します。JDBC プログラミングの詳細については、JDBC プログラミングの参考書を参照してください。

JDBC は次の方法で使用できます。

クライアント側で **JDBC** を使用する

Java クライアントアプリケーションはデータベースサーバに対して JDBC 呼び出しができます。接続は JDBC ドライバを介して行われます。

SQL Anywhere JDBC ドライバ (Type 2 JDBC ドライバ) はソフトウェアに含まれています。pure Java アプリケーション用の jConnect ドライバ (Type 4 JDBC ドライバ) もサポートされます。

データベース側で **JDBC** を使用する

データベースにインストールされている Java クラスは JDBC 呼び出しを行って、データベース内のデータにアクセスしたり、修正したりできます。これには内部 JDBC ドライバを使用します。

JDBC リソース

サンプルのソースコード

この例で示したソースコードは、*%SQLANYSAMP17%*¥SQLAnywhere¥JDBC ディレクトリにあります。

必要なソフトウェア

jConnect ドライバを使用するには、TCP/IP が必要です。

関連情報

jConnect JDBC ドライバ [218 ページ]
jConnect for JDBC

1.8.2 JDBC ドライバ

SQL Anywhere JDBC ドライバと jConnect ドライバがサポートされています。

これらの JDBC ドライバには次の特性があります。

SQL Anywhere JDBC ドライバ

このドライバは、Command Sequence クライアント/サーバプロトコルを使用してデータベースサーバと通信します。このドライバの動作は、ODBC、Embedded SQL、OLE DB アプリケーションと一貫性があります。SQL Anywhere JDBC ドライバは、データベースに接続する際に推奨される JDBC ドライバです。このドライバは、JRE 1.6 以降でのみ使用できます。

このドライバは、JDBC ドライバの JAVA VM への登録を自動的に実行します。したがって、sajdbc4.jar ファイルをクラスファイルパスに配置し、jdbc:sqlanywhere で始まる URL を指定して、DriverManager.getConnection() を単に呼び出すだけで十分です。

この JDBC ドライバには、OSGi (Open Services Gateway initiative) バンドルとしてのロードを可能にするマニフェスト情報が含まれています。

SQL Anywhere JDBC ドライバでは、NCHAR データのメタデータから java.sql.Types.NCHAR、NVARCHAR、または LONGNVARCHAR のカラム型が返されるようになりました。また、アプリケーションで、Get/SetString と Get/SetClob メソッドの代わりに Get/SetNString または Get/SetNClob メソッドを使用して NCHAR データをフェッチできるようになりました。

jConnect

このドライバは、100% pure Java ドライバです。TDS クライアント/サーバプロトコルを使用してデータベースサーバと通信します。

使用するドライバを選択するときには、次の要因を考慮してください。

機能

SQL Anywhere JDBC ドライバでは、データベースに接続したときに完全にスクロール可能なカーソルを使用できます。jConnect JDBC ドライバでは、データベースに接続したときに、スクロール可能なカーソルを使用できますが、結果セットはクライアント側でキャッシュされます。jConnect JDBC ドライバでは、Adaptive Server Enterprise データベースに接続したときに、完全にスクロール可能なカーソルを使用できます。

Pure Java

jConnect ドライバは Type 4 のドライバであり、Pure Java ソリューションです。SQL Anywhere JDBC ドライバは Type 2 のドライバであり、Pure Java ソリューションではありません。

> **注記**
>
> SQL Anywhere JDBC ドライバは SQL Anywhere ODBC ドライバをロードしないので、Type 1 ドライバではありません。

パフォーマンス

ほとんどの用途において、パフォーマンスは SQL Anywhere JDBC ドライバの方が jConnect ドライバより優れています。

互換性

jConnect ドライバで使用される TDS プロトコルは、Adaptive Server Enterprise と共有されます。ドライバの動作の一部は、このプロトコルで制御されており、Adaptive Server Enterprise との互換性を持つように設定されています。

関連情報

プラットフォーム別 SQL Anywhere コンポーネント
jConnect for JDBC

1.8.3 SQL Anywhere JDBC ドライバ

SQL Anywhere JDBC ドライバには、pure Java である jConnect JDBC ドライバに比べてパフォーマンスや機能の面で利点があるため、その使用が推奨されます。ただし、SQL Anywhere JDBC ドライバでは pure Java ソリューションは提供されません。

このセクションの内容:

SQL Anywhere JDBC ドライバをロードする方法 [217 ページ]
JDBC ドライバがクラスファイルパスにあることを確認します。

SQL Anywhere JDBC ドライバ接続文字列 [217 ページ]
SQL Anywhere JDBC ドライバを介してデータベースに接続するには、データベースの URL を指定します。

関連情報

JDBC ドライバ [215 ページ]

1.8.3.1 SQL Anywhere JDBC ドライバをロードする方法

JDBC ドライバがクラスファイルパスにあることを確認します。

```
set classpath=%SQLANY17%¥java¥sajdbc4.jar;%classpath%
```

JDBC ドライバは新しい自動ドライバ登録を利用します。ドライバがクラスファイルパスにあると、実行が起動された時点でドライバが自動的にロードされます。

必要なファイル

JDBC ドライバの Java コンポーネントは、ソフトウェアインストール環境の `Java` サブディレクトリにインストールされている `sajdbc4.jar` ファイルに含まれています。Windows の場合、ネイティブコンポーネントはソフトウェアインストール環境の `bin32` または `bin64` サブディレクトリの `dbjdbc17.dll` です。UNIX の場合、ネイティブコンポーネントは `libdbjdbc17.so` です。このコンポーネントは、システムパスにあることが必要です。Mac OS X 10.11 では、`java.library.path` に `libdbjdbc.dylib` へのフルパスを含める必要があります。例: java -Djava.library.path=$SQLANY17/lib64

関連情報

JDBC クライアントの配備 [827 ページ]

1.8.3.2 SQL Anywhere JDBC ドライバ接続文字列

SQL Anywhere JDBC ドライバを介してデータベースに接続するには、データベースの URL を指定します。

データベースへの接続方法の例を下記に示します。

```
Connection con = DriverManager.getConnection( "jdbc:sqlanywhere:DSN=SQL Anywhere 17
Demo;Password="+pwd );
```

URL には、jdbc:sqlanywhere: の後に接続文字列が含まれています。`sajdbc4.jar` ファイルがクラスファイルパスにあると、SQL Anywhere JDBC ドライバは自動的にロードされて URL を処理します。便宜上、例では ODBC データソース (DSN) を指定していますが、データソース接続パラメータとともに、またはデータソース接続パラメータの代わりに、明示的な接続パラメータをセミコロンで区切って使用することもできます。

データソースを使用しない場合は、必要な接続パラメータをすべて接続文字列で指定する必要があります。

```
Connection con =
DriverManager.getConnection( "jdbc:sqlanywhere:UserID=DBA;Password=passwd;Start=..."
 );
```

ODBC ドライバと ODBC ドライバマネージャがどちらも使用されないため、Driver 接続パラメータは必要ありません。指定しても、無視されます。

次の別の例では、TCP/IP ポート 2638 を使用してホストコンピュータ Elora で動作するサーバ Acme 上のデータベース SalesDB に接続します。

```
Connection con = DriverManager.getConnection(
        "jdbc:sqlanywhere:UserID=DBA;Password=passwd;Host=Elora:
2638;ServerName=Acme;DatabaseName=SalesDB" );
```

1.8.4 jConnect JDBC ドライバ

アプレットから JDBC を使用する場合は、jConnect JDBC ドライバを介してデータベースに接続してください。

jConnect ドライバは、SAP Service Marketplace の SAP ソフトウェアダウンロードセンタから個別にダウンロードできます。SDK FOR SAP ASE を検索してください。jConnect のマニュアルはインストールに付属しています。https://service.sap.com/sap/support/notes/2093510 でドライバを検索できます。

jConnect は `jconn4.jar` という名前の JAR ファイルとして提供されています。このファイルは、jConnect をインストールした場所にあります。

jConnect ドライバのファイル

jConnect は `jconn4.jar` という名前の JAR ファイルとして提供されています。このファイルは、jConnect をインストールした場所にあります。

jConnect 用クラスファイルパスの設定

アプリケーションで jConnect を使用するには、コンパイル時と実行時に、jConnect クラスをクラスファイルパスに指定します。これにより、Java コンパイラと Java ランタイムが必要なファイルを見つけられるようになります。

次のコマンドは、既存の CLASSPATH 環境変数に jConnect ドライバを追加します。ここで、`jconnect-path` は jConnect のインストールディレクトリです。

```
set classpath=jconnect-path¥classes¥jconn4.jar;%classpath%
```

jConnect クラスのインポート

jConnect のクラスは、すべて com.sap.jdbc4.jdbc にあります。これらのクラスを各ソースファイルの先頭でインポートしてください。

```
import com.sap.jdbc4.jdbc.*
```

パスワードの暗号化

パスワードは jConnect 接続用に暗号化できます。次の例で説明します。

```
Connection  connection = null;
Properties props = new Properties();
props.put( "ENCRYPT_PASSWORD", "true" );
props.put( "User", "DBA" );
props.put( "Password", "passwd" );
try {
    Driver drvr = (Driver)
(Class.forName( "com.sap.jdbc4.jdbc.SybDriver" ).newInstance());
    connection = drvr.connect( "jdbc:sap:Tds:localhost:2638", props );
}
catch( Exception e ) {
    System.err.println( "Error! Could not connect" );
    System.err.println( e.getMessage() );
    printExceptions( (SQLException)e );
    connection = null;
}
```

このセクションの内容:

jConnect システムオブジェクトのデータベースへのインストール [220 ページ]
jConnect システムオブジェクトをデータベースに追加し、jConnect を使用してシステムテーブル情報 (データベースメタデータ) にアクセスできるようにします。

jConnect ドライバをロードする方法 [221 ページ]
jConnect ドライバがクラスファイルパスにあることを確認します。ドライバファイル `jconn4.jar` は、jConnect のインストールディレクトリの `classes` サブディレクトリにあります。

jConnect ドライバ接続文字列 [221 ページ]
jConnect を介してデータベースに接続するには、データベースの URL を指定します。

関連情報

jConnect for JDBC

1.8.4.1 jConnect システムオブジェクトのデータベースへのインストール

jConnect システムオブジェクトをデータベースに追加し、jConnect を使用してシステムテーブル情報 (データベースメタデータ) にアクセスできるようにします。

前提条件

ALTER DATABASE、BACKUP DATABASE、SERVER OPERATOR システム権限が必要です。また、このデータベースに他の接続がないことが必要です。

データベースファイルをバックアップしてからアップグレードしてください。データベースをアップグレードしようとして失敗した場合、データベースは使用できなくなります。

コンテキスト

dbinit ユーティリティを使用すると、jConnect システムオブジェクトはデフォルトでデータベースにインストールされます。jConnect システムオブジェクトは、データベースの作成時、またはその後でも、データベースのアップグレードを実行してデータベースに追加できます。次のように、*SQL Central* からデータベースをアップグレードできます。

手順

1. *SQL Central* を使用してデータベースに接続します。
2. *[ツール]* > *[SQL Anywhere17]* をクリックし、*[データベースのアップグレード]* をクリックします。
3. データベースアップグレードウィザードの指示に従います。

結果

jConnect システムオブジェクトがデータベースに追加されます。

または、Upgrade ユーティリティ (dbupgrad) を使用してデータベースに接続し、jConnect システムオブジェクトをデータベースにインストールします。次はその例です。

```
dbupgrad -c "UID=DBA;PWD=passwd;SERVER=myServer;DBN=myDatabase
```

1.8.4.2 jConnect ドライバをロードする方法

jConnect ドライバがクラスファイルパスにあることを確認します。ドライバファイル `jconn4.jar` は、jConnect のインストールディレクトリの `classes` サブディレクトリにあります。

```
set classpath=.;c:¥jConnect-7_0¥classes¥jconn4.jar;%classpath%
```

ドライバがクラスファイルパスにあると、実行が起動された時点でドライバが自動的にロードされます。

1.8.4.3 jConnect ドライバ接続文字列

jConnect を介してデータベースに接続するには、データベースの URL を指定します。

データベースへの接続方法の例を下記に示します。

```
Connection con = DriverManager.getConnection( "jdbc:sybase:Tds:localhost:2638",
"DBA", "sql" );
```

URL は次のように構成されます。

```
jdbc:sybase:Tds:host:port
```

個々のコンポーネントの説明は次のとおりです。

jdbc:sybase:Tds

TDS アプリケーションプロトコルを使用する、jConnect JDBC ドライバ。

host

サーバが動作しているコンピュータの IP アドレスまたは名前。同一ホスト接続を確立している場合は、ログインしているコンピュータシステムを意味する localhost を使用できます。

port

データベースサーバが受信しているポート番号。データベースサーバで使用されるデフォルトポート番号は 2638 です。

接続文字列の長さは、253 文字未満にしてください。

パーソナルサーバを使用している場合は、サーバの起動時に TCP/IP サポートオプションを必ず含めてください。

このセクションの内容:

jConnect 接続文字列でデータベースを指定する方法 [222 ページ]
各データベースサーバには、1 つまたは複数のデータベースを一度にロードできます。jConnect 経由の接続時に設定する URL でデータベースではなくサーバを指定する場合、そのサーバのデフォルトのデータベースに対して接続が試行されます。

jConnect 接続でのデータベースオプションの設定 [223 ページ]
アプリケーションが jConnect ドライバを使用してデータベースに接続するとき、sp_tsql_environment ストアドプロシージャが呼び出されます。sp_tsql_environment プロシージャでは、Adaptive Server Enterprise の動作と互換性を保つためのデータベースオプションを設定します。

1.8.4.3.1 jConnect 接続文字列でデータベースを指定する方法

各データベースサーバには、1 つまたは複数のデータベースを一度にロードできます。jConnect 経由の接続時に設定する URL でデータベースではなくサーバを指定する場合、そのサーバのデフォルトのデータベースに対して接続が試行されます。

次のいずれかの方法で拡張形式の URL を提示することによって、特定のデータベースを指定できます。

ServiceName パラメータの使用

```
jdbc:sybase:Tds:host:port?ServiceName=database
```

疑問符に続けて一連の割り当てを入力するのは、URL に引数を指定する標準的な方法です。ServiceName の大文字と小文字は区別されません。等号 (=) の前後にはスペースを入れないでください。`database` パラメータはデータベース名で、サーバ名ではありません。データベース名にはパスやファイルサフィックスを含めることはできません。例:

```
Connection con = DriverManager.getConnection(
      "jdbc:sybase:Tds:localhost:2638?ServiceName=demo", "DBA", "sql");
```

RemotePWD パラメータの使用

追加の接続パラメータをサーバに渡すための対処方法があります。

この手法により、RemotePWD フィールドを使用して、データベース名やデータベースファイルなどの追加の接続パラメータを指定できます。put メソッドを使用して、Properties フィールドに RemotePWD を設定します。

次のコードは、このフィールドの使い方を示します。

```
import java.util.Properties;
.
.
.
Properties props = new Properties();
props.put( "User", "DBA" );
props.put( "Password", "passwd" );
props.put( "RemotePWD", ",DatabaseFile=mydb.db" );

Connection con = DriverManager.getConnection(
    "jdbc:sybase:Tds:localhost:2638", props );
```

例で示しているように、DatabaseFile 接続パラメータの前にはカンマを入力してください。DatabaseFile パラメータを使用すると、jConnect を使用してサーバ上でデータベースを起動できます。デフォルトでは、データベースは AutoStop=YES で起動されます。utility_db を DatabaseFile (DBF) または DatabaseName (DBN) 接続パラメータで指定すると (例: DBN=utility_db)、ユーティリティデータベースが自動的に起動します。

1.8.4.3.2 jConnect 接続でのデータベースオプションの設定

アプリケーションが jConnect ドライバを使用してデータベースに接続するとき、sp_tsql_environment ストアドプロシージャが呼び出されます。sp_tsql_environment プロシージャでは、Adaptive Server Enterprise の動作と互換性を保つためのデータベースオプションを設定します。

1.8.5 JDBC プログラムの構造

通常、JDBC アプリケーションはデータベースに接続し、1 つ以上の SQL 文を実行し、結果セットを処理した後、データベースから切断します。

通常の JDBC アプリケーションの要素を次に示します。

Connection オブジェクトを作成する

DriverManager クラスの getConnection メソッドで、Connection オブジェクトが作成され、データベースとの接続が確立します。

Statement オブジェクトを作成する

Connection オブジェクトの createStatement メソッドで、Statement オブジェクトが作成されます。

SQL 文を実行する

Statement オブジェクトの実行メソッドで、データベース環境内で SQL 文が実行されます。1 つまたは複数の結果セットが使用できる場合、ブール型の結果 `true` が返されます。

1 つまたは複数の結果セットを処理する

Statement オブジェクトの getResultSet メソッドと getMoreResults メソッドを使用し、結果セットを取得します。

ResultSet オブジェクトを使用し、SQL 文から返されたデータを一度に 1 つのローずつ取得します。

各結果セットのローをループする

ResultSet オブジェクトの次のメソッドを使用し、結果セットの次のローを指定します。次のローが存在しない場合、ブール型の結果 `false` が返されます。

それぞれのローに入る値の検索

カラムの名前か位置のどちらかを指定すると、ResultSet オブジェクトの各カラムに入る値が検索されます。getString メソッドを使用すると、現在のローにあるカラムから値を取得することができます。

JDBC オブジェクトを適切なタイミングでリリースする

各 JDBC オブジェクトのクローズメソッドで、リソースをシステムにリリースします。

次は前述した原則の例です。

```
import java.io.*;
import java.sql.*;
public class results
{
    public static void main(String[] args) throws Exception
    {
        Connection conn = null;
        try
        {
            conn = DriverManager.getConnection(
```

```
                    "jdbc:sqlanywhere:uid=DBA;pwd=sql" );
            String SQL = "BEGIN¥n"
                + " SELECT * FROM Departments "
                + "     ORDER BY DepartmentID;¥n"
                + " SELECT EmployeeID, Surname, GivenName "
                + "     FROM Employees e "
                + "     JOIN Departments d "
                + "     ON DepartmentHeadID = EmployeeID "
                + "     ORDER BY d.DepartmentID¥n"
                + "END";
            Statement stmt = conn.createStatement();
            if( stmt.execute(SQL) )
            {
                ResultSet rs = stmt.getResultSet();
                while( rs != null )
                {
                    System.out.println( "¥n---Result set---¥n" );
                    while (rs.next())
                    {
                        System.out.println(rs.getString(1)
                                + ", " + rs.getString(2)
                                + ", " + rs.getString(3) );
                    }
                    if( stmt.getMoreResults() )
                    {
                        rs = stmt.getResultSet();
                    }
                    else
                    {
                        rs.close();
                        rs = null;
                    }
                }
            }
            stmt.close();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        if( conn != null) conn.close();
    }
}
```

Java オブジェクトは、JDBC オブジェクトを使用してデータベースと対話し、データを取得できます。

1.8.6 クライアント側 JDBC アプリケーションとサーバ側 JDBC アプリケーションの違い

クライアント側 JDBC アプリケーションとサーバ側 JDBC アプリケーションには、わずかな違いがあります。

クライアント側

クライアントコンピュータから JDBC を使用している場合、SQL Anywhere JDBC ドライバまたは jConnect JDBC ドライバのどちらかを使用して接続が確立されます。ユーザ ID やパスワードなどの接続の詳細は、接続を確立する DriverManager.getConnection に引数として渡されます。データベースサーバは、同じコンピュータシステムまたは別のコンピュータシステムで実行されます。JDBC アプリケーションは、クライアントコンピュータからアクセス可能な 1 つまたは複数のクラスファイルに含まれています。

サーバ側

データベースサーバから JDBC を使用している場合、接続はすでに確立されています。"jdbc:default:connection" という文字列が DriverManager.getConnection に渡され、JDBC アプリケーションは現在のユーザ接続の文脈内で動作できるようになります。これは簡単で効率が良く、安全な操作です。その理由は、接続を確立するためにクライアントユーザがデータベースセキュリティをすでに満たしているためです。ユーザ ID とパスワードなどの認証情報がすでに提供されているので、もう一度提供する必要はありません。JDBC アプリケーションからの接続は、アプリケーションを起動したデータベースでのみ確立できます。JDBC アプリケーションは、データベースに格納されている 1 つまたは複数のクラスファイルに含まれています。

URL の構成に 1 つの条件付きの文を使用することによってクライアントとサーバの両方で実行できるように、JDBC クラスを作成できます。内部接続には "jdbc:default:connection" のみが必要ですが、外部接続にはユーザ ID、パスワード、ホスト名、ポート番号などの接続情報が必要です。

1.8.7 クライアント側サンプル JDBS アプリケーション

通常の JDBC アプリケーションは、データベースサーバに接続し、SQL クエリを発行し、複数の結果セットを処理してから終了します。

下記の完全な Java アプリケーションは、実行中のデータベースに接続し、SQL クエリを発行し、複数の結果セットを処理して表示してから終了します。デフォルトでは、SQL Anywhere JDBC ドライバを使用してデータベースに接続します。jConnect ドライバを使用するには、データベースユーザ ID とパスワード、ドライバ名 (jConnect)、および必要に応じて SQL クエリ (引用符で囲みます) をコマンドラインで指定します。

SQL Anywhere JDBC ドライバを使用すると、データベースメタデータをいつでも使用できます。

jConnect を使用する JDBC アプリケーションからデータベースシステムテーブル (データベースメタデータ) にアクセスする場合は、jConnect システムオブジェクトのセットをデータベースに追加してください。これらのプロシージャは、すべてのデータベースにデフォルトでインストールされています。dbinit -i オプションを指定すると、このインストールは行われません。

この例では、サンプルデータベースを使用して、データベースサーバがすでに起動されていることを前提としています。このサンプルのソースコードは、*%SQLANYSAMP17%*¥SQLAnywhere¥JDBC¥JDBCConnect.java にあります。

```
import java.io.*;
import java.sql.*;
public class JDBCConnect
{
    public static void main( String args[] )
    {
        try
        {
            String userID = "";
            String password = "";
            String driver = "jdbc4";
            String SQL =
                  "BEGIN"
                + " SELECT * FROM Departments"
                + "     ORDER BY DepartmentID;"
                + " SELECT d.DepartmentID, GivenName, Surname, EmployeeID"
                + "     FROM Employees e"
                + "     JOIN Departments d"
                + "     ON DepartmentHeadID = EmployeeID"
                + "     ORDER BY d.DepartmentID;"
                + "END";
            if( args.length > 0 ) userID = args[0];
            if( args.length > 1 ) password = args[1];
            if( args.length > 2 ) driver = args[2];
            if( args.length > 3 ) SQL = args[3];
```

```
            Connection con;
            if( driver.compareToIgnoreCase( "jconnect" ) == 0 )
            {
                con = DriverManager.getConnection(
                    "jdbc:sybase:Tds:localhost:2638", userID, password);
            }
            else
            {
                con = DriverManager.getConnection(
                    "jdbc:sqlanywhere:uid=" + userID + ";pwd=" + password);
            }
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery(SQL);
            while( rs != null )
            {
                while (rs.next())
                {
                    for( int i = 1;
                         i <= rs.getMetaData().getColumnCount();
                         i++ )
                    {
                        if( i > 1 ) System.out.print(", ");
                        System.out.print(rs.getString(i));
                    }
                    System.out.println();
                }
                if( stmt.getMoreResults() )
                {
                    System.out.println();
                    rs = stmt.getResultSet();
                }
                else
                {
                    rs.close();
                    rs = null;
                }
            }
            stmt.close();
            con.close();
        }
        catch (SQLException sqe)
        {
            System.out.println("Unexpected exception : " +
                               sqe.toString() + ", sqlstate = " +
                               sqe.getSQLState());
            System.exit(1);
        }
        catch (Exception e)
        {
            e.printStackTrace();
            System.exit(1);
        }
        System.exit(0);
    }
}
```

このセクションの内容:

クライアント側 JDBC アプリケーションの動作方法 [227 ページ]

この JDBC アプリケーションは、データベースサーバに接続し、SQL クエリを発行し、複数の結果セットを処理してから終了します。

クライアント側サンプル JDBS アプリケーションの実行 [228 ページ]

動作している JDBC アプリケーションの作成に必要な手順を理解するために、サンプル JDBC アプリケーションをコンパイルして実行します。

関連情報

jConnect JDBC ドライバ [218 ページ]
サーバ側サンプル JDBS アプリケーション [229 ページ]

1.8.7.1 クライアント側 JDBC アプリケーションの動作方法

この JDBC アプリケーションは、データベースサーバに接続し、SQL クエリを発行し、複数の結果セットを処理してから終了します。

パッケージのインポート

このアプリケーションにはいくつかのパッケージが必要で、そのパッケージは `JDBCConnect.java` の 1 行目でインポートされます。

- java.io パッケージには Java 入出力クラスが含まれています。このクラスはコマンドプロンプトウィンドウに出力するために必要です。
- java.sql パッケージには JDBC クラスが含まれていて、このクラスはすべての JDBC アプリケーションで必要です。

アプリケーションの構造

Java アプリケーションには `main` という名前のメソッドを持つクラスが必要です。これはプログラムの起動時に呼び出されるメソッドです。この簡単な例では、JDBCConnect.main がアプリケーションで唯一のパブリックメソッドです。

この JDBCConnect.main メソッドでは、次のタスクを実行します。

1. データベースユーザ ID、パスワード、JDBC ドライバ名、SQL クエリをオプションのコマンドライン引数から取得します。選択したドライバによって、SQL Anywhere JDBC ドライバまたは jConnect 7.0 ドライバがロードされます (クラスファイルにある場合)。
2. 選択された JDBC ドライバ URL を使用して、実行中のデータベースに接続します。getConnection メソッドで、指定された URL を使用して接続が確立されます。
3. 文オブジェクトを作成します。このオブジェクトは SQL 文のコンテナーです。
4. SQL クエリを実行して結果セットオブジェクトを作成します。
5. 結果セットを反復処理して、カラム情報を表示します。
6. 追加の結果セットの有無を確認し、別の結果セットがある場合は、前の手順を繰り返します。
7. 結果セット、文、接続の各オブジェクトを閉じます。

1.8.7.2 クライアント側サンプル JDBS アプリケーションの実行

動作している JDBC アプリケーションの作成に必要な手順を理解するために、サンプル JDBC アプリケーションをコンパイルして実行します。

前提条件

Java Development Kit (JDK) をインストールしている必要があります。

コンテキスト

JDBC を使用した 2 つの異なるタイプの接続を確立できます。1 つはクライアント側接続で、他はサーバ側接続です。次の例ではクライアント側の接続を使用します。

手順

1. コマンドプロンプトで、*%SQLANYSAMP17%*¥SQLAnywhere¥JDBC ディレクトリに移動します。
2. 次のコマンドを使用して、サンプルデータベースを含むローカルコンピュータ上のデータベースサーバを起動します。

   ```
   dbsrv17 "%SQLANYSAMP17%¥demo.db"
   ```
3. CLASSPATH 環境変数を設定します。SQL Anywhere JDBC ドライバは sajdbc4.jar に含まれています。

   ```
   set classpath=.;%SQLANY17%¥java¥sajdbc4.jar
   ```

   jConnect ドライバを使用している場合は、CLASSPATH を次のように設定します (jconnect-path は jConnect インストールディレクトリです)。

   ```
   set classpath=.;jconnect-path¥classes¥jconn4.jar
   ```
4. 次のコマンドを実行してサンプルをコンパイルします。

   ```
   javac JDBCConnect.java
   ```
5. 次のコマンドを実行してサンプルを実行します。

   ```
   java JDBCConnect Browser browse
   ```

   デフォルト SQL クエリが実行されます。

   接続が失敗すると、エラーメッセージが表示されます。必要な手順をすべて実行したかどうかを確認します。CLASSPATH が正しく設定されていることもチェックしてください。設定が間違っていると、クラスを検索できません。
6. 省略可能です。別のユーザ ID、パスワード、JDBC ドライバを選択する場合は、コマンドライン引数を指定します。

   ```
   java JDBCConnect Updater update jconnect
   ```

7. 省略可能です。コマンドラインに SQL クエリを含めます。

```
java JDBCConnect Browser browse jdbc4 "SELECT * FROM Customers"
```

結果

結果セットが表示されます。

1.8.8 サーバ側サンプル JDBS アプリケーション

通常の JDBC アプリケーションは、データベースサーバに接続し、SQL クエリを発行し、複数の結果セットを処理してから終了します。

下記の完全な Java アプリケーションは、サーバ側接続を使用して実行中のデータベースに接続し、SQL クエリを発行し、複数の結果セットを処理して表示してから終了します。

サーバ側 JDBC アプリケーションから接続を確立する方が、外部接続を確立するよりも簡単です。ユーザはすでにデータベースに接続されているので、アプリケーションでは現在の接続を使用できるからです。

この例のソースコードは、`JDBCConnect.java` 例の改変バージョンで、*%SQLANYSAMP17%*`¥SQLAnywhere¥JDBC¥JDBCConnect2.java` にあります。

```
import java.io.*;
import java.sql.*;
public class JDBCConnect2
{
    public static void main( String args[] )
    {
        try
        {
            String SQL =
                  "BEGIN"
                + " SELECT * FROM Departments"
                + "     ORDER BY DepartmentID;"
                + " SELECT d.DepartmentID, GivenName, Surname, EmployeeID"
                + "     FROM Employees e"
                + "     JOIN Departments d"
                + "     ON DepartmentHeadID = EmployeeID"
                + "     ORDER BY d.DepartmentID;"
                + "END";
            if( args.length > 0 && args[0] != null && args[0].length() > 0 )
                SQL = args[0];
            Connection con = DriverManager.getConnection(
                "jdbc:default:connection" );
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery(SQL);
            while( rs != null )
            {
                while (rs.next())
                {
                    for( int i = 1;
                         i <= rs.getMetaData().getColumnCount();
                         i++ )
                    {
                        if( i > 1 ) System.out.print(", ");
```

```
                        System.out.print(rs.getString(i));
                    }
                    System.out.println();
                }
                if( stmt.getMoreResults() )
                {
                    System.out.println();
                    rs = stmt.getResultSet();
                }
                else
                {
                    rs.close();
                    rs = null;
                }
            }
            stmt.close();
            con.close();
        }
        catch (SQLException sqe)
        {
            System.out.println("Unexpected exception : " +
                               sqe.toString() + ", sqlstate = " +
                               sqe.getSQLState());
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

このセクションの内容:

サーバ側 JDBC アプリケーションの動作方法 [230 ページ]

サーバ側 JDBC アプリケーションは、次のことを除いてサンプルクライアント側 JDBC アプリケーションとほぼ同じです。

サーバ側サンプル JDBS アプリケーションの実行 [231 ページ]

動作しているサーバ側 JDBC アプリケーションの作成に必要な手順を理解するために、サンプル JDBC アプリケーションをコンパイルして実行します。

1.8.8.1 サーバ側 JDBC アプリケーションの動作方法

サーバ側 JDBC アプリケーションは、次のことを除いてサンプルクライアント側 JDBC アプリケーションとほぼ同じです。

1. ユーザ ID、パスワード、JDBC ドライバ引数は不要です。
2. 現在の接続を使用して、稼働中のデフォルトデータベースに接続します。getConnection 呼び出しで指定した URL は次のように変更されています。

   ```
   Connection con = DriverManager.getConnection(
       "jdbc:default:connection" );
   ```

3. 出力はすべてサーバメッセージウィンドウにリダイレクトされます。
4. `System.exit()` 文は削除されています。

1.8.8.2 サーバ側サンプル JDBS アプリケーションの実行

動作しているサーバ側 JDBC アプリケーションの作成に必要な手順を理解するために、サンプル JDBC アプリケーションをコンパイルして実行します。

前提条件

Java Development Kit (JDK) をインストールしている必要があります。

次のシステム権限が必要です。

- MANAGE ANY EXTERNAL OBJECT による Java クラスのインストール
- CREATE PROCEDURE と CREATE EXTERNAL REFERENCE による外部プロシージャの作成

コンテキスト

JDBC を使用した 2 つの異なるタイプの接続を確立できます。1 つはクライアント側接続で、他はサーバ側接続です。次の例ではサーバ側の接続を使用します。

手順

1. コマンドプロンプトで、*%SQLANYSAMP17%*¥SQLAnywhere¥JDBC ディレクトリに移動します。

   ```
   cd %SQLANYSAMP17%¥SQLAnywhere¥JDBC
   ```

2. サーバ側 JDBC の場合、サーバが異なる現在の作業ディレクトリから起動されないかぎり、CLASSPATH 環境変数を設定する必要はありません。

   ```
   set classpath=.;%SQLANYSAMP17%¥SQLAnywhere¥JDBC
   ```

3. 次のコマンドを使用して、サンプルデータベースを含むローカルコンピュータ上のデータベースサーバを起動します。

   ```
   dbsrv17 "%SQLANYSAMP17%¥demo.db"
   ```

4. 次のコマンドを入力してサンプルをコンパイルします。

   ```
   javac JDBCConnect2.java
   ```

5. Interactive SQL を使用して、クラスをサンプルデータベースにインストールします。次の文を実行します (クラスファイルへのパスが必要な場合があります)。

   ```
   INSTALL JAVA NEW
   FROM FILE 'JDBCConnect2.class';
   ```

SQL Central を使用してクラスをインストールすることもできます。サンプルデータベースに接続している間に、*[外部環境]* フォルダの *[Java]* サブフォルダを開き、*[ファイル]* > *[新規]* > *[Java クラス]* をクリックします。ウィザードの指示に従います。

6. クラスの JDBCConnect2.main メソッドのラッパーとして動作する JDBCConnect という名前のストアドプロシージャを定義します。

```
CREATE OR REPLACE PROCEDURE JDBCConnect(IN arg LONG VARCHAR)
  EXTERNAL NAME 'JDBCConnect2.main([Ljava/lang/String;)V'
  LANGUAGE JAVA;
```

7. 次のように JDBCConnect2.main メソッドを呼び出します。

```
CALL JDBCConnect('');
```

   セッション内で初めて Java クラスが呼び出されると、Java VM がロードされます。これには数秒間かかる場合があります。

8. データベースサーバメッセージウィンドウに結果セットが表示されたことを確認します。

   接続が失敗すると、エラーメッセージが表示されます。必要な手順をすべて実行したかどうかを確認します。

9. 別の SQL クエリを次のように実行してください。

```
CALL JDBCConnect('SELECT * FROM Products;SELECT * FROM Departments');
```

   データベースサーバメッセージウィンドウに別の結果セットが表示されます。

結果

データベースサーバメッセージウィンドウに結果セットが表示されます。

1.8.9 JDBC 接続についての注意

クライアント側の JDBC とサーバ側の JDBC に違いがあることを認識してください。ここでは、オートコミットの動作や独立性レベルなどの各側面を説明します。

オートコミットの動作

JDBC の仕様により、デフォルトでは各データ操作文が実行されると、COMMIT が実行されます。現在、クライアント側の JDBC はコミットを実行し (オートコミットは true)、サーバ側の JDBC はコミットを実行しない (オートコミットは false) ように動作します。クライアント側とサーバ側のアプリケーションの両方で同じ動作を実行するには、次のような文を使用します。

```
con.setAutoCommit( false );
```

この文で、con は現在の接続オブジェクトです。オートコミットを true に設定することもできます。

トランザクションの独立性レベルの設定

トランザクションの独立性レベルを設定するには、次のいずれかの値を使用して、アプリケーションで Connection.setTransactionIsolation メソッドを呼び出す必要があります。

SQL Anywhere JDBC ドライバ用:

- TRANSACTION_NONE
- TRANSACTION_READ_COMMITTED
- TRANSACTION_READ_UNCOMMITTED
- TRANSACTION_REPEATABLE_READ
- TRANSACTION_SERIALIZABLE
- sap.jdbc4.sqlanywhere.IConnection.SA_TRANSACTION_SNAPSHOT
- sap.jdbc4.sqlanywhere.IConnection.SA_TRANSACTION_STATEMENT_SNAPSHOT
- sap.jdbc4.sqlanywhere.IConnection.SA_TRANSACTION_STATEMENT_READONLY_SNAPSHOT

次の例は、SQL Anywhere JDBC ドライバを使用して、トランザクションの独立性レベルを SNAPSHOT に設定します。

```
try
{
    con.setTransactionIsolation(
        sap.jdbc4.sqlanywhere.IConnection.SA_TRANSACTION_SNAPSHOT
    );
}
catch( Exception e )
{
    System.err.println( "Error! Could not set isolation level" );
    System.err.println( e.getMessage() );
    printExceptions( (SQLException)e );
}
```

接続デフォルト

サーバ側の JDBC からデフォルト値で新しい接続を作成するのは、`getConnection( "jdbc:default:connection" )` の最初の呼び出しだけです。後続の呼び出しは、接続プロパティを変更せずに、現在の接続のラッパーを返します。最初の接続でオートコミットを false に設定すると、同じ Java コード内の後続の getConnection 呼び出しでは、オートコミットが false に設定された接続を返します。

接続を閉じるときに接続プロパティをデフォルト値に復元し、後続の接続を標準の JDBC 値で取得できるようにします。これを行うには、次のコードを実行します。

```
Connection con =
    DriverManager.getConnection("jdbc:default:connection");
boolean oldAutoCommit = con.getAutoCommit();
try
{
     // main body of code here
}
finally
{
    con.setAutoCommit( oldAutoCommit );
}
```

ここに記載された説明は、オートコミットだけでなく、トランザクションの独立性レベルや読み込み専用モードなどのその他の接続プロパティにも適用されます。

SQL Anywhere JDBC ドライバを使用した接続の失敗

JDBC ドライバで必要なすべてのファイルをデプロイしなかった場合、データベースへの接続時に `Invalid ODBC handle` エラーが発生する可能性があります。JDBC ドライバの適切な動作に必要なすべてのファイルをインストールしていることを確認してください。

関連情報

JDBC クライアントの配備 [827 ページ]

1.8.10 JDBC を使用したサーバ側データアクセス

SQL から呼び出し可能な Java メソッドとして実装されたデータベーストランザクションロジックには、従来の SQL ストアドプロシージャに比べて大きな利点があります。

専用の SQL ストアドプロシージャ定義構文を使用して、Java メソッドへのインタフェースが実装されます。JDBC を使用するものを含めた Java メソッドの呼び出しは、完全に SQL 文で構成された SQL ストアドプロシージャの呼び出しとよく似ています。以降のトピックでは、データベース (サーバ側 JDBC) から JDBC を使用する方法について説明しますが、例ではクライアント側アプリケーション用に JDBC を記述する方法も示します。

その他のプログラミングインタフェースと同様に、JDBC の SQL 文は静的または動的のどちらでもかまいません。静的 SQL 文は Java アプリケーション内で構成され、データベースに送信されます。データベースサーバは文を解析し、実行プランを選択して SQL 文を実行します。

同じまたはよく似た SQL 文を何度も実行する (たとえば 1 つのテーブルに何度も挿入する) 場合、静的 SQL では著しいオーバヘッドが生じる可能性があります。これは、SQL 文が準備の手順を毎回実行する必要があるためです。

反対に、動的 SQL 文にはプレースホルダがあります。これらのプレースホルダを使用して文を一度準備すれば、それ以降は準備をしなくても何度も文を実行できます。

以降のトピックでは、静的 SQL 文と動的 SQL 文の実行例と、バッチ (ワイド挿入、ワイド削除、ワイド更新、ワイドマージ) の実行例を示します。

このセクションの内容:

サーバ側 JDBC の例の設定 [235 ページ]
サンプル Java アプリケーションをコンパイルし、データベースにインストールします。

JDBC を使用した挿入、更新、削除 [236 ページ]
INSERT、UPDATE、DELETE など結果セットを返さない静的 SQL 文の実行には、Statement クラスの executeUpdate メソッドを使用します。CREATE TABLE などの文やその他のデータ定義文も、executeUpdate を使用して実行できます。

JDBC からの静的 INSERT 文と DELETE 文の使用 [237 ページ]
サンプル JDBC アプリケーションがデータベースサーバから呼び出され、静的 SQL 文を使用して Departments テーブルでローを挿入、削除します。

より効率的なアクセスのために準備文を使用する方法 [238 ページ]
Statement インタフェースを使用する場合は、データベースに送信するそれぞれの文を解析してアクセスプランを生成し、文を実行します。実行する前の手順を、文の準備と呼びます。

JDBC からの準備 INSERT 文と DELETE 文の使用 [240 ページ]
サンプル JDBC アプリケーションをデータベースサーバから呼び出し、準備文を使用して Departments テーブルでローを挿入、削除します。

JDBC バッチメソッド [241 ページ]

PreparedStatement クラスの addBatch メソッドは、バッチ (またはワイド) 挿入、削除、更新、マージを実行するために使用します。次に、このメソッドの使用に関するガイドラインの一部を示します。

Java から結果セットを返す方法 [242 ページ]
呼び出し元の環境に 1 つ以上の結果セットを返す Java メソッドを記述し、SQL ストアドプロシージャにこのメソッドをラップします。

JDBC から結果セットを返す [243 ページ]
サンプル JDBC アプリケーションをデータベースサーバから呼び出し、複数の結果セットを返します。

データベースで JDBC を使用する場合の権限の要件 [244 ページ]
適切な権限のセットを使用することで、Java クラスでメソッドを実行できます。

1.8.10.1 サーバ側 JDBC の例の設定

サンプル Java アプリケーションをコンパイルし、データベースにインストールします。

前提条件

Java Development Kit (JDK) をインストールしている必要があります。

クラスをインストールするには、MANAGE ANY EXTERNAL OBJECT システム権限が必要です。

コンテキスト

JDBC サンプルのソースコードは *%SQLANYSAMP17%*`¥SQLAnywhere¥JDBC¥JDBCExample.java` にあります。

手順

1. コマンドプロンプトで、`JDBCExample.java` ソースコードをコンパイルします。

   ```
   javac JDBCExample.java
   ```

2. Interactive SQL からデータベースに接続します。
3. Interactive SQL で次の文を実行して、`JDBCExample.class` ファイルをサンプルデータベースにインストールします。

   ```
   INSTALL JAVA NEW
   FROM FILE 'JDBCExample.class';
   ```

   データベースサーバがクラスファイルと同じディレクトリから起動されず、クラスファイルへのパスがデータベースサーバの CLASSPATH にリスト表示されていない場合、クラスファイルへのパスを INSTALL 文に含める必要があります。

SQL Central を使用してクラスをインストールすることもできます。サンプルデータベースに接続している間に、*[外部環境]* フォルダの *[Java]* サブフォルダを開き、*[ファイル]* > *[新規]* > *[Java クラス]* をクリックします。ウィザードの指示に従います。

結果

JDBCExample クラスファイルがデータベースにインストールされ、デモの準備が完了しました。このクラスファイルは、以降のトピックで使用します。

1.8.10.2 JDBC を使用した挿入、更新、削除

INSERT、UPDATE、DELETE など結果セットを返さない静的 SQL 文の実行には、Statement クラスの executeUpdate メソッドを使用します。CREATE TABLE などの文やその他のデータ定義文も、executeUpdate を使用して実行できます。

Statement クラスの addBatch、clearBatch、executeBatch メソッドも使用できます。Statement クラスの executeBatch メソッドの動作に関して JDBC 仕様は不明確であるため、SQL Anywhere JDBC ドライバでこのメソッドを使用する場合には、次の点に注意してください。

- SQL 例外または結果セットが発生すると、バッチ処理がすぐに停止します。バッチ処理が停止すると、executeBatch メソッドによって BatchUpdateException がスローされます。BatchUpdateException で getUpdateCounts メソッドを呼び出すと、ローカウントの整数配列が返されます。バッチが失敗する前のカウントのセットには、負でない有効な更新カウントが含まれ、バッチが失敗したとき以降のすべてのカウントには、-1 の値が含まれます。BatchUpdateException を SQLException にキャストすると、バッチ処理の停止理由に関する詳細が示されます。
- バッチは、clearBatch メソッドが明示的に呼び出された場合にのみ、クリアされます。したがって、executeBatch メソッドを繰り返し呼び出すと、バッチが何度も再実行されます。また、execute(sql_query) または executeQuery(sql_query) を呼び出すと、指定された SQL クエリは正しく実行されますが、基盤となるバッチはクリアされません。このため、executeBatch メソッドの後で、execute(sql_query) を呼び出してから、executeBatch メソッドを再度呼び出すと、一連のバッチ文が実行されてから指定の SQL クエリが実行され、その後で一連のバッチ文が再実行されます。

次のコードフラグメントは、INSERT 文の実行方法を示しています。ここでは、引数として InsertStatic に渡された Statement オブジェクトを使用しています。

```
public static void InsertStatic( Statement stmt )
{
  try
  {
    int iRows = stmt.executeUpdate(
      "INSERT INTO Departments (DepartmentID, DepartmentName)"
      + " VALUES (201, 'Eastern Sales')" );
    // Print the number of rows inserted
    System.out.println(iRows + " rows inserted");
  }
  catch (SQLException sqe)
  {
    System.out.println("Unexpected exception : " +
              sqe.toString() + ", sqlstate = " +
              sqe.getSQLState());
  }
  catch (Exception e)
  {
```

```
      e.printStackTrace();
    }
  }
```

注記

- このコードフラグメントは、*%SQLANYSAMP17%*¥SQLAnywhere¥JDBC ディレクトリに含まれている JDBCExample.java ファイルの一部です。
- executeUpdate メソッドは整数を返します。この整数は、操作の影響を受けるローの番号を表しています。この場合、INSERT が成功すると、値 1 が返されます。
- サーバ側のクラスとして実行すると、System.out.println の出力結果はデータベースサーバのメッセージウィンドウに表示されます。

1.8.10.3 JDBC からの静的 INSERT 文と DELETE 文の使用

サンプル JDBC アプリケーションがデータベースサーバから呼び出され、静的 SQL 文を使用して Departments テーブルでローを挿入、削除します。

前提条件

外部プロシージャを作成するには、CREATE PROCEDURE および CREATE EXTERNAL REFERENCE システム権限が必要です。また、修正しているデータベースオブジェクトでの SELECT、DELETE、INSERT 権限も必要です。

手順

1. Interactive SQL からサンプルデータベースに接続します。
2. JDBCExample クラスがインストールされていることを確認します。
3. クラスの JDBCExample.main メソッドのラッパーとして動作する JDBCExample という名前のストアドプロシージャを定義します。

   ```
   CREATE PROCEDURE JDBCExample( IN arg LONG VARCHAR )
     EXTERNAL NAME 'JDBCExample.main([Ljava/lang/String;)V'
     LANGUAGE JAVA;
   ```

4. 次のように JDBCExample.main メソッドを呼び出します。

   ```
   CALL JDBCExample( 'insert' );
   ```

   引数の文字列に 'insert' を指定すると、InsertStatic メソッドが呼び出されます。

5. ローが Departments テーブルに追加されたことを確認します。

   ```
   SELECT * FROM Departments;
   ```

   サンプルプログラムでは、Departments テーブルの更新された内容をデータベースサーバメッセージウィンドウに表示します。
6. DeleteStatic という名前のサンプルクラスには、追加されたばかりのローを削除する同じようなメソッドがあります。次のように JDBCExample.main メソッドを呼び出します。

   ```
   CALL JDBCExample( 'delete' );
   ```

   引数の文字列に `'delete'` を指定すると、DeleteStatic メソッドが呼び出されます。
7. ローが Departments テーブルから削除されたことを確認します。

   ```
   SELECT * FROM Departments;
   ```

   Departments テーブルの更新された内容が表示されます。

結果

サーバ側 JDBC アプリケーションで静的 SQL 文を使用して、ローがテーブルで挿入、削除されます。Departments テーブルの更新された内容がデータベースサーバメッセージウィンドウに表示されます。

関連情報

サーバ側 JDBC の例の設定 [235 ページ]

1.8.10.4 より効率的なアクセスのために準備文を使用する方法

Statement インタフェースを使用する場合は、データベースに送信するそれぞれの文を解析してアクセスプランを生成し、文を実行します。実行する前の手順を、文の準備と呼びます。

PreparedStatement インタフェースを使用すると、パフォーマンス上有利になります。これによりプレースホルダを使用して文を準備し、文の実行時にプレースホルダへ値を割り当てることができます。

たくさんのローを挿入するときなど、同じ動作を何度も繰り返す場合は、準備文を使用すると特に便利です。

例

次の例では、PreparedStatement インタフェースの使い方を解説しますが、単一のローを挿入するのは、準備文の正しい使い方ではありません。

JDBCExamples クラスの次の InsertDynamic メソッドによって、準備文を実行します。

```
public static void InsertDynamic( Connection con,
```

```
                String ID, String name )
{
  try
  {
    // Build the INSERT statement
    // ? is a placeholder character
    String sqlStr = "INSERT INTO Departments " +
            "( DepartmentID, DepartmentName ) " +
            "VALUES ( ? , ? )";
    // Prepare the statement
    PreparedStatement stmt =
      con.prepareStatement( sqlStr );
    // Set some values
    int idValue = Integer.valueOf( ID );
    stmt.setInt( 1, idValue );
    stmt.setString( 2, name );
    // Execute the statement
    int iRows = stmt.executeUpdate();
    // Print the number of rows inserted
    System.out.println(iRows + " rows inserted");
  }
  catch (SQLException sqe)
  {
    System.out.println("Unexpected exception : " +
                sqe.toString() + ", sqlstate = " +
                sqe.getSQLState());
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}
```

注記

- このコードフラグメントは、*%SQLANYSAMP17%*¥SQLAnywhere¥JDBC ディレクトリに含まれている JDBCExample.java ファイルの一部です。
- executeUpdate メソッドは整数を返します。この整数は、操作の影響を受けるローの番号を表しています。この場合、INSERT が成功すると、値 1 が返されます。
- サーバ側のクラスとして実行すると、System.out.println の出力結果はデータベースサーバのメッセージウィンドウに表示されます。

1.8.10.5 JDBC からの準備 INSERT 文と DELETE 文の使用

サンプル JDBC アプリケーションをデータベースサーバから呼び出し、準備文を使用して Departments テーブルでローを挿入、削除します。

前提条件

外部プロシージャを作成するには、CREATE PROCEDURE および CREATE EXTERNAL REFERENCE システム権限が必要です。また、修正しているデータベースオブジェクトでの SELECT、DELETE、INSERT 権限も必要です。

手順

1. Interactive SQL からサンプルデータベースに接続します。
2. JDBCExample クラスがインストールされていることを確認します。
3. クラスの JDBCExample.Insert メソッドのラッパーとして動作する JDBCInsert という名前のストアドプロシージャを定義します。

   ```
   CREATE PROCEDURE JDBCInsert( IN arg1 INTEGER, IN arg2 LONG VARCHAR )
     EXTERNAL NAME 'JDBCExample.Insert(ILjava/lang/String;)V'
     LANGUAGE JAVA;
   ```

4. 次のように JDBCExample.Insert メソッドを呼び出します。

   ```
   CALL JDBCInsert( 202, 'Southeastern Sales' );
   ```

   Insert メソッドにより InsertDynamic メソッドが呼び出されます。
5. ローが Departments テーブルに追加されたことを確認します。

   ```
   SELECT * FROM Departments;
   ```

   サンプルプログラムでは、Departments テーブルの更新された内容をデータベースサーバメッセージウィンドウに表示します。
6. DeleteDynamic という名前のサンプルクラスには、追加されたばかりのローを削除する同じようなメソッドがあります。

   クラスの JDBCExample.Delete メソッドのラッパーとして動作する JDBCDelete という名前のストアドプロシージャを定義します。

   ```
   CREATE PROCEDURE JDBCDelete( IN arg1 INTEGER )
     EXTERNAL NAME 'JDBCExample.Delete(I)V'
     LANGUAGE JAVA;
   ```

7. 次のように JDBCExample.Delete メソッドを呼び出します。

   ```
   CALL JDBCDelete( 202 );
   ```

   Delete メソッドにより DeleteDynamic メソッドが呼び出されます。

8. ローが Departments テーブルから削除されたことを確認します。

```
SELECT * FROM Departments;
```

Departments テーブルの更新された内容が表示されます。

結果

サーバ側 JDBC アプリケーションで準備 SQL 文を使用して、ローがテーブルで挿入、削除されます。Departments テーブルの更新された内容がデータベースサーバメッセージウィンドウに表示されます。

関連情報

準備文 [13 ページ]
サーバ側 JDBC の例の設定 [235 ページ]

1.8.10.6 JDBC バッチメソッド

PreparedStatement クラスの addBatch メソッドは、バッチ (またはワイド) 挿入、削除、更新、マージを実行するために使用します。次に、このメソッドの使用に関するガイドラインの一部を示します。

1. Connection クラスのいずれかの prepareStatement メソッドを使用して、SQL 文を準備します。

```
String sqlStr = "INSERT INTO Departments( DepartmentID, DepartmentName ) VALUES
( ? , ? )";
PreparedStatement pstmt = con.prepareStatement( sqlStr );
```

2. 準備された文のパラメータを設定し、バッチとして追加します。次に示す大まかな例は、それぞれ m 個のパラメータを持つ n 個のバッチを作成するものです。

```
for( i=0; i < n; i++ )
{
    pstmt.set...( 1, param_1 );
    pstmt.set...( 2, param_2 );
    .
    .
    .
    pstmt.set...( m , param_m );
    pstmt.addBatch();
}
```

次の例は、それぞれ 2 個のパラメータを持つ 5 個のバッチを作成します。最初のパラメータは整数、2 番目のパラメータは文字列です。

```
for( i=0; i < 5; i++ )
{
    pstmt.setInt( 1, idValue[i] );
    pstmt.setString( 2, name[i] );
```

```
    pstmt.addBatch();
}
```

3. PreparedStatement クラスの executeBatch メソッドを使用して、バッチを実行する必要があります。

```
int[] affected = pstmt.executeBatch();
```

BLOB パラメータはバッチでサポートされていません。

SQL Anywhere JDBC ドライバを使用してバッチ挿入を実行する場合は、小さいカラムサイズを使用してください。バッチ挿入を使用して、大きなバイナリデータまたは文字データを long binary カラムまたは long varchar カラムに挿入すると、パフォーマンスが低下する可能性があるため、推奨しません。パフォーマンスが低下するのは、バッチ挿入されるローをすべて保持するために JDBC ドライバが大容量のメモリを割り当てる必要があるためです。

このようなケースを除けば、バッチによる挿入、削除、更新、マージは、単独で実行するよりも高いパフォーマンスが得られるはずです。

1.8.10.7 Java から結果セットを返す方法

呼び出し元の環境に 1 つ以上の結果セットを返す Java メソッドを記述し、SQL ストアドプロシージャにこのメソッドをラップします。

次のコードフラグメントは、複数の結果セットをこの Java プロシージャの呼び出し元に返す方法を示しています。ここでは、3 つの executeQuery 文を使用して 3 つの異なる結果セットを取得します。

```
public static void Results( ResultSet[] rset )
       throws SQLException
{
    // Demonstrate returning multiple result sets
    Connection con = DriverManager.getConnection(
                    "jdbc:default:connection" );
    rset[0] = con.createStatement().executeQuery(
        "SELECT * FROM Employees" +
        "   ORDER BY EmployeeID" );
    rset[1] = con.createStatement().executeQuery(
        "SELECT * FROM Departments" +
        "   ORDER BY DepartmentID" );
    rset[2] = con.createStatement().executeQuery(
        "SELECT i.ID,i.LineID,i.ProductID,i.Quantity," +
        "       s.OrderDate,i.ShipDate," +
        "       s.Region,e.GivenName||' '||e.Surname" +
        "   FROM SalesOrderItems AS i" +
        "   JOIN SalesOrders AS s" +
        "   JOIN Employees AS e" +
        "   WHERE s.ID=i.ID" +
        "         AND s.SalesRepresentative=e.EmployeeID" );
    con.close();
}
```

注記

- このサーバ側 JDBC の例は、*%SQLANYSAMP17%*¥SQLAnywhere¥JDBC ディレクトリに含まれている JDBCExample.java ファイルの一部です。

- この例では、getConnection を使用して、稼働中のデフォルトデータベースへの接続を取得します。
- executeQuery メソッドによって結果セットが返されます。

1.8.10.8 JDBC から結果セットを返す

サンプル JDBC アプリケーションをデータベースサーバから呼び出し、複数の結果セットを返します。

前提条件

外部プロシージャを作成するには、CREATE PROCEDURE および CREATE EXTERNAL REFERENCE システム権限が必要です。また、修正しているデータベースオブジェクトでの SELECT、DELETE、INSERT 権限も必要です。

手順

1. Interactive SQL からサンプルデータベースに接続します。
2. JDBCExample クラスがインストールされていることを確認します。
3. クラスの JDBCExample.Results メソッドのラッパとして動作する JDBCResults という名前のストアドプロシージャを定義します。

   次に例を示します。

   ```
   CREATE PROCEDURE JDBCResults(OUT args LONG VARCHAR)
     DYNAMIC RESULT SETS 3
     EXTERNAL NAME 'JDBCExample.Results([Ljava/sql/ResultSet;)V'
     LANGUAGE JAVA;
   ```

   例では 3 つの結果セットを返します。
4. 次の Interactive SQL オプションを設定すると、クエリのすべての結果が表示されます。
   a. *[ツール]* > *[オプション]* をクリックします。
   b. *SQL Anywhere* をクリックします。
   c. *[結果]* タブをクリックします。
   d. *[表示できるローの最大数]* の値を **5000** に設定します。
   e. *[すべての結果セットを表示]* をクリックします。
   f. *[OK]* をクリックします。
5. JDBCExample.Results メソッドを呼び出します。

   ```
   CALL JDBCResults();
   ```
6. 3 つの結果タブ *[結果セット 1]*、*[結果セット 2]*、*[結果セット 3]* のそれぞれを確認します。

結果

3 つの異なる結果セットは、サーバ側 JDBC アプリケーションから返されます。

関連情報

サーバ側 JDBC の例の設定 [235 ページ]

1.8.10.9 データベースで JDBC を使用する場合の権限の要件

適切な権限のセットを使用することで、Java クラスでメソッドを実行できます。

アクセス権限

データベースのすべての Java クラスと同様、JDBC 文が含まれているクラスには、Java メソッドのラッパーとして動作するストアドプロシージャを実行する権限が GRANT EXECUTE 文で付与されているどのユーザでもアクセスできます。

実行権限

Java クラスは Java メソッドのラッパーとして動作しているストアドプロシージャの権限で実行されます (デフォルトでは、これが SQL SECURITY DEFINER です)。

1.8.11 JDBC コールバック

SQL Anywhere JDBC ドライバでは、2 つの非同期コールバックをサポートしています。1 つは SQL MESSAGE 文を処理し、もう 1 つはファイル転送要求を検証します。これらは、ODBC ドライバでサポートされているコールバックに似ています。

メッセージは、SQL MESSAGE 文を使用してクライアントアプリケーションからデータベースサーバに送信できます。実行時間が長いデータベースサーバ文によってメッセージも生成できます。

メッセージハンドラルーチンを作成して、これらのメッセージを捕捉できます。次に、メッセージハンドラのコールバックルーチンの例を示します。

```
class T_message_handler implements sap.jdbc4.sqlanywhere.ASAMessageHandler
{
    private final int MSG_INFO      = 0x80 | 0;
    private final int MSG_WARNING   = 0x80 | 1;
    private final int MSG_ACTION    = 0x80 | 2;
    private final int MSG_STATUS    = 0x80 | 3;
    T_message_handler()
    {
    }
    public SQLException messageHandler(SQLException sqe)
    {
        String msg_type = "unknown";

        switch( sqe.getErrorCode() ) {
            case MSG_INFO:      msg_type = "INFO    ";   break;
            case MSG_WARNING:   msg_type = "WARNING";   break;
```

```
            case MSG_ACTION:    msg_type = "ACTION ";   break;
            case MSG_STATUS:    msg_type = "STATUS ";   break;
        }

        System.out.println( msg_type + ": " + sqe.getMessage() );
        return sqe;
    }
}
```

クライアントファイル転送要求を検証できます。転送を許可する前に、JDBC ドライバは検証コールバックが存在する場合は、それを呼び出します。ストアドプロシージャからなどの間接文の実行中にクライアントのデータ転送が要求された場合、JDBC ドライバはクライアントアプリケーションで検証コールバックが登録されていないかぎり転送を許可しません。どのような状況で検証の呼び出しが行われるかについては、以下でより詳しく説明します。次に、ファイル転送検証のコールバックルーチンの例を示します。

```
class T_filetrans_callback implements
sap.jdbc4.sqlanywhere.SAValidateFileTransferCallback
{
    T_filetrans_callback()
    {
    }

    public int callback(String filename, int is_write)
    {
        System.out.println( "File transfer granted for file " + filename
+
                            " with an is_write value of " +
is_write );
        return( 1 ); // 0 to disallow, non-zero to allow
    }
}
```

filename 引数は、読み込みまたは書き込み対象のファイルの名前です。is_write パラメータは、読み込み (クライアントからサーバへの転送) が要求された場合は 0、書き込みが要求された場合は 0 以外の値になります。ファイル転送が許可されない場合、コールバック関数は 0 を返します。それ以外の場合は 0 以外の値を返します。

データのセキュリティ上、サーバはファイル転送を要求している文の実行元を追跡します。サーバは、文がクライアントアプリケーションから直接受信されたものかどうかを判断します。クライアントからデータ転送を開始する際に、サーバは文の実行元に関する情報をクライアントソフトウェアに送信します。クライアント側では、クライアントアプリケーションから直接送信された文を実行するためにデータ転送が要求されている場合にかぎり、JDBC ドライバはデータの転送を無条件で許可します。それ以外の場合は、上述の検証コールバックがアプリケーションで登録されていることが必要です。登録されていない場合、転送は拒否されて文が失敗し、エラーが発生します。データベース内に既存しているストアドプロシージャがクライアントの文で呼び出された場合、ストアドプロシージャそのものの実行はクライアントの文で開始されたものと見なされません。ただし、クライアントアプリケーションでテンポラリストアドプロシージャを明示的に作成してストアドプロシージャを実行した場合、そのプロシージャはクライアントによって開始されたものとしてサーバは処理します。同様に、クライアントアプリケーションでバッチ文を実行する場合も、バッチ文はクライアントアプリケーションによって直接実行されるものと見なされます。

次のサンプル Java アプリケーションは、SQL Anywhere JDBC ドライバでサポートされているコールバックの使用を示しています。クラスパスにファイル *%SQLANY17%*¥java¥sajdbc4.jar を置きます。

```
import java.io.*;
import java.sql.*;
import java.util.*;
public class callback
{
    public static void main (String args[]) throws IOException
    {
        Connection          con = null;
        Statement           stmt;
        System.out.println ( "Starting... " );
```

```
        con = connect();
        if( con == null )
        {
            return; // exception should already have been reported
        }
        System.out.println ( "Connected... " );
        try
        {
            // create and register message handler callback
            T_message_handler message_worker = new T_message_handler();

((sap.jdbc4.sqlanywhere.IConnection)con).setASAMessageHandler( message_worker );

            // create and register validate file transfer callback
            T_filetrans_callback filetran_worker = new T_filetrans_callback();

((sap.jdbc4.sqlanywhere.IConnection)con).setSAValidateFileTransferCallback( filetran
_worker );

            stmt = con.createStatement();

            // execute message statements to force message handler to be called
            stmt.execute( "MESSAGE 'this is an info   message' TYPE INFO TO
CLIENT" );
            stmt.execute( "MESSAGE 'this is an action message' TYPE ACTION TO
CLIENT" );
            stmt.execute( "MESSAGE 'this is a warning message' TYPE WARNING TO
CLIENT" );
            stmt.execute( "MESSAGE 'this is a status  message' TYPE STATUS TO
CLIENT" );

            System.out.println( "¥n==================¥n" );

            stmt.execute( "set temporary option allow_read_client_file='on'" );
            try
            {
                stmt.execute( "drop procedure read_client_file_test" );
            }
            catch( SQLException dummy )
            {
                // ignore exception if procedure does not exist
            }
            // create procedure that will force file transfer callback to be called
            stmt.execute( "create procedure read_client_file_test()" +
                          "begin" +
                          "    declare v long binary;" +
                          "    set v = read_client_file('sample.txt');" +
                          "end" );

            // call procedure to force validate file transfer callback to be called
            try
            {
                stmt.execute( "call read_client_file_test()" );
            }
            catch( SQLException filetrans_exception )
            {
                // Note: Since the file transfer callback returns 1,
                // do not expect a SQL exception to be thrown
                System.out.println( "SQLException: " +
                                    filetrans_exception.getMessage() );
            }
            stmt.close();
            con.close();
            System.out.println( "Disconnected" );
        }
        catch( SQLException sqe )
        {
            printExceptions(sqe);
```

```
        }
    }

    private static Connection connect()
    {
        Connection  connection;

        System.out.println( "Using jdbc4 driver"  );
        try
        {
            connection = DriverManager.getConnection(
                    "jdbc:sqlanywhere:uid=DBA;pwd=sql" );
        }
        catch( Exception e )
        {
            System.err.println( "Error! Could not connect" );
            System.err.println( e.getMessage() );
            printExceptions( (SQLException)e );
            connection = null;
        }
        return connection;
    }
    static private void printExceptions(SQLException sqe)
    {
        while (sqe != null)
        {
            System.out.println("Unexpected exception : " +
                "SqlState: " + sqe.getSQLState()  +
                " " + sqe.toString() +
                ", ErrorCode: " + sqe.getErrorCode());
            System.out.println( "==================¥n" );
            sqe = sqe.getNextException();
        }
    }
}
```

1.8.12 JDBC エスケープ構文

JDBC エスケープ構文は、InteractiveSQL を含む JDBC アプリケーションで使用できます。エスケープ構文を使用して、使用しているデータベース管理システムとは関係なくストアドプロシージャを呼び出すことができます。

エスケープ構文の一般的な形式は次のようになります。

```
{ keyword parameters }
```

次のキーワードセットがあります。

{d date-string}

date-string は、データベースサーバが受け取ることのできる任意の日付値です。

{t time-string}

time-string は、データベースサーバが受け取ることのできる任意の時刻値です。

{ts date-string time-string}

date/time-string は、データベースサーバが受け取ることのできる任意のタイムスタンプ値です。

{guid uuid-string}

uuid-string は、任意の有効な GUID 文字列です (例: 41dfe9ef-db91-11d2-8c43-006008d26a6f)。

{oj outer-join-expr}

outer-join-expr は、データベースサーバが受け取ることのできる有効な OUTER JOIN 式です。

{?= call func(p1, ...)}

func は、データベースサーバが受け取ることのできる任意の有効な関数呼び出しです。

{call proc(p1, ...)}

proc は、データベースサーバが受け取ることのできる任意の有効なストアドプロシージャ呼び出しです。

{fn func(p1, ...)}

func は、以下に示すいずれかの関数ライブラリです。

エスケープ構文を使用して、JDBC ドライバによって実装される関数ライブラリにアクセスできます。このライブラリには、数値、文字列、時刻、日付、システム関数が含まれています。

たとえば、次のコマンドを実行すると、データベース管理システムの種類にかかわらず現在の日付を取得できます。

```
SELECT { FN CURDATE() }
```

使用できる関数は、使っている JDBC ドライバによって異なります。次の表は、SQL Anywhere JDBC ドライバと jConnect ドライバによってサポートされている関数を示します。

SQL Anywhere JDBC ドライバがサポートする関数

数値関数	文字列関数	システム関数	日付/時刻関数
ABS	ASCII	DATABASE	CURDATE
ACOS	BIT_LENGTH	IFNULL	CURRENT_DATE
ASIN	CHAR	USER	CURRENT_TIME
ATAN	CHAR_LENGTH		CURRENT_TIMESTAMP
ATAN2	CHARACTER_LENGTH		CURTIME
CEILING	CONCAT		DAYNAME
COS	DIFFERENCE		DAYOFMONTH
COT	INSERT		DAYOFWEEK
DEGREES	LCASE		DAYOFYEAR
EXP	LEFT		EXTRACT
FLOOR	LENGTH		HOUR
LOG	LOCATE		MINUTE
LOG10	LTRIM		MONTH
MOD	OCTET_LENGTH		MONTHNAME
PI	POSITION		NOW
POWER	REPEAT		QUARTER
RADIANS	REPLACE		SECOND

数値関数	文字列関数	システム関数	日付/時刻関数
RAND	RIGHT		WEEK
ROUND	RTRIM		YEAR
SIGN	SOUNDEX		
SIN	SPACE		
SQRT	SUBSTRING		
TAN	UCASE		
TRUNCATE			

jConnect がサポートする関数

数値関数	文字列関数	システム関数	日付/時刻関数
ABS	ASCII	DATABASE	CURDATE
ACOS	CHAR	IFNULL	CURTIME
ASIN	CONCAT	USER	DAYNAME
ATAN	DIFFERENCE	CONVERT	DAYOFMONTH
ATAN2	LCASE		DAYOFWEEK
CEILING	LENGTH		HOUR
COS	REPEAT		MINUTE
COT	RIGHT		MONTH
DEGREES	SOUNDEX		MONTHNAME
EXP	SPACE		NOW
FLOOR	SUBSTRING		QUARTER
LOG	UCASE		SECOND
LOG10			TIMESTAMPADD
PI			TIMESTAMPDIFF
POWER			YEAR
RADIANS			
RAND			
ROUND			
SIGN			
SIN			
SQRT			
TAN			

JDBC TIMESTAMPADD、TIMESTAMPDIFF

JDBC ドライバは TIMESTAMPADD 関数および TIMESTAMPDIFF 関数を対応する DATEADD 関数および DATEDIFF 関数にマップします。TIMESTAMPADD 関数と TIMESTAMPDIFF 関数の構文は次のとおりです。

```
{ fn TIMESTAMPADD( interval, integer-expr, timestamp-expr ) }
```

integer-expr の間隔 (interval 型) を timestamp-expr に追加して計算されたタイムスタンプを返します。interval の有効な値を次に示します。

```
{ fn TIMESTAMPDIFF( interval, timestamp-expr1, timestamp-expr2 ) }
```

interval 型の間隔を timestamp-expr2 と timestamp-expr1 の差を表す整数値で返します。interval の有効な値を次に示します。

interval	DATEADD/DATEDIFF date-part のマッピング
SQL_TSI_YEAR	YEAR
SQL_TSI_QUARTER	QUARTER
SQL_TSI_MONTH	MONTH
SQL_TSI_WEEK	WEEK
SQL_TSI_DAY	DAY
SQL_TSI_HOUR	HOUR
SQL_TSI_MINUTE	MINUTE
SQL_TSI_SECOND	SECOND
SQL_TSI_FRAC_SECOND	MICROSECOND - DATEADD および DATEDIFF 関数はナノ秒の精度をサポートしません。

Interactive SQL

Interactive SQL では、波括弧 ({) は必ず二重にしてください。括弧の間にスペースを入れないでください。"{{" は使用できますが、"{ {" は使用できません。また、文中に改行文字を使用できません。ストアドプロシージャは Interactive SQL で解析されないため、ストアドプロシージャではエスケープ構文を使用できません。

たとえば、2013 年 2 月の週数を取得するには、Interactive SQL で次の文を実行します。

```
SELECT {{ fn TIMESTAMPDIFF(SQL_TSI_WEEK, '2013-02-01T00:00:00',
'2013-03-01T00:00:00'  ) }}
```

1.8.13 SQL Anywhere JDBC API のサポート

java.sql.Blob インタフェースの一部のオプションメソッドは、SQL Anywhere JDBC ドライバではサポートされていません。

サポートされていないメソッドは、次のとおりです。

- long position(Blob pattern, long start);
- long position(byte[] pattern, long start);
- OutputStream setBinaryStream(long pos)
- int setBytes(long pos, byte[] bytes)
- int setBytes(long pos, byte[] bytes, int offset, int len);
- void truncate(long len);

1.9 Node.js アプリケーションプログラミング

Node.js API は、SQL Anywhere データベースへの接続、SQL クエリの発行、結果セットの取得に使用できます。

Node.js ドライバを使用すると、Joyent の Node.js ソフトウェアプラットフォームで JavaScript を使用してデータベースに接続し、クエリを実行できます。様々なバージョンの Node.js. のドライバが使用可能です。

API インタフェースは SAP HANA Node.js Client にきわめて似ており、文の接続、接続解除、実行、準備が可能です。

ドライバは、NPM (Node Packaged Modules) Web サイト (https://npmjs.org/) で入手できます。

また、https://github.com/sqlanywhere/node-sqlanywhere からダウンロードすることもできます。

> **注記**
>
> API リファレンスマニュアルをお探しですか。マニュアルをローカルにインストールした場合は、Windows のスタートメニューを使用してアクセスするか (Microsoft Windows)、`C:¥Program Files¥SQL Anywhere 17¥Documentation` にナビゲートします。
>
> また、DocCommentXchange の Web で、SAP SQL Anywhere API リファレンスマニュアルにアクセスすることもできます。http://dcx.sap.com

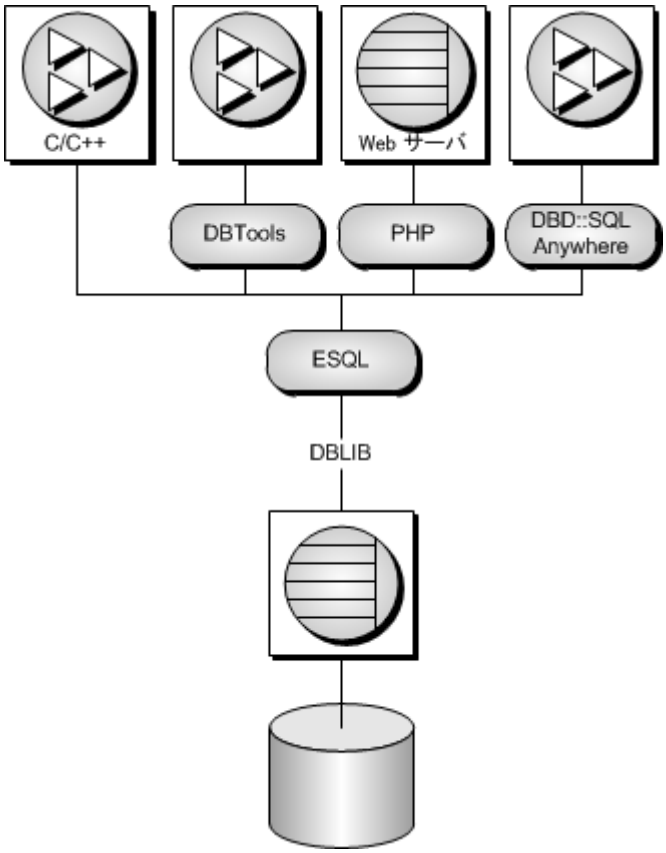
1.10 Embedded SQL

C または C++ のソースファイルに組み込まれた SQL 文を、Embedded SQL と呼びます。プリプロセッサがそれらの SQL 文をランタイムライブラリの呼び出しに変換します。Embedded SQL は ISO/ANSI および IBM 規格です。

Embedded SQL は他のデータベースや他の環境に移植可能であり、あらゆる動作環境で同等の機能を実現します。
Embedded SQL は、それぞれの製品で使用可能なすべての機能を提供する包括的な低レベルインタフェースです。
Embedded SQL を使用するには、C または C++ プログラミング言語に関する知識が必要です。

Embedded SQL アプリケーション

Embedded SQL インタフェースを使用してデータベースサーバにアクセスする C または C++ アプリケーションを開発できます。たとえば、コマンドラインデータベースツールは、このような方法で開発されたアプリケーションです。



Embedded SQL は、C と C++ プログラミング言語用のデータベースプログラミングインタフェースです。Embedded SQL は、C と C++ のソースコードが混合された (埋め込まれた) SQL 文で構成されます。この SQL 文は **Embedded SQL** プリプロセッサによって C または C++ のソースコードに翻訳され、その後ユーザによってコンパイルされます。

実行時に、Embedded SQL アプリケーションは DBLIB と呼ばれるインタフェースライブラリのインタフェースライブラリを使用してデータベースサーバと通信します。DBLIB は、ほとんどのプラットフォームでの、ダイナミックリンクライブラリ (**DLL**) または共有オブジェクトです。

- Windows オペレーティングシステムでは、インタフェースライブラリは `dblib17.dll` です。
- UNIX オペレーティングシステムでは、インタフェースライブラリはオペレーティングシステムによって異なり、`libdblib17.so`、`libdblib17.sl`、または `libdblib17.a` です。
- Mac OS X では、インタフェースライブラリは `libdblib17.dylib.1` です。

2 種類の Embedded SQL が用意されています。静的な Embedded SQL は、動的な Embedded SQL に比べて使用方法は単純ですが、柔軟性は乏しくなります。

このセクションの内容:

開発プロセスの概要 [254 ページ]

プログラムのプリプロセッサ処理とコンパイルが成功すると、プログラムを DBLIB 用のインポートライブラリとリンクして、実行ファイルを作成できます。データベースサーバが実行中のとき、この実行ファイルは DBLIB を使用してデータベースサーバと交信します。

Embedded SQL プリプロセッサ [255 ページ]
Embedded SQL プリプロセッサの実行プログラム名は `sqlpp` です。

対応コンパイラ [259 ページ]
Windows コンパイラと UNIX コンパイラの両方が Embedded SQL プリプロセッサとともに使用されています。

Embedded SQL ヘッダファイル [259 ページ]
ヘッダファイルはすべて、ソフトウェアインストールディレクトリの `SDK¥Include` サブディレクトリにインストールされています。

インポートライブラリ [260 ページ]
インポートライブラリは、ソフトウェアインストールディレクトリのサブディレクトリにインストールされています。

Embedded SQL サンプルプログラム [261 ページ]
ここで、Embedded SQL プログラムの非常に簡単な例を示します。

Embedded SQL プログラムの構造 [261 ページ]
SQL 文は通常の C または C++ コードの内部に置かれ (埋め込まれ) ます。Embedded SQL 文は、必ず、EXEC SQL で始まり、セミコロン (;) で終わります。

Windows での DBLIB の動的ロード [262 ページ]
インポートライブラリとリンクしなくてもよいように、ソフトウェアインストールディレクトリの `SDK¥C` サブディレクトリにある `esqldll.c` モジュールを使用して、Embedded SQL アプリケーションから DBLIB を動的にロードします。

サンプル Embedded SQL プログラム [263 ページ]
ソフトウェアには、サンプル Embedded SQL プログラムが含まれています。

Embedded SQL のデータ型 [268 ページ]
プログラムとデータベースサーバ間で情報を転送するには、それぞれのデータについてデータ型を設定します。

Embedded SQL のホスト変数 [272 ページ]
ホスト変数とは、Embedded SQL プリプロセッサが認識する C 変数です。ホスト変数はデータベースサーバに値を送ったり、データベースサーバから値を受け取ったりするのに使用できます。

SQLCA (SQL Communication Area) [281 ページ]
SQLCA (**SQL Communication Area**) とは、データベースへの要求のたびに、アプリケーションとデータベースサーバの間で、統計情報とエラーをやりとりするのに使用されるメモリ領域です。

静的 SQL と動的 SQL [286 ページ]
SQL 文を C プログラムに埋め込むには、静的 SQL と動的 SQL の 2 つの方法があります。

SQLDA (SQL descriptor area) [289 ページ]
SQLDA (SQL Descriptor Area) は動的 SQL 文で使用されるインタフェース構造体です。この構造体を使用して、ホスト変数と SELECT 文の結果に関する情報を、データベースとの間で受け渡しします。SQLDA はヘッダファイル `sqlda.h` に定義されています。

Embedded SQL を使用してデータをフェッチする方法 [299 ページ]
Embedded SQL でデータをフェッチするには SELECT 文を使用します。

Embedded SQL を使用したワイド挿入 [305 ページ]
INSERT 文は一度に複数のローを挿入するように使用できます。こうするとパフォーマンスが向上することがあります。これをワイド挿入または配列挿入といいます。

Embedded SQL を使用したワイド削除 [309 ページ]
DELETE 文を使用すると、任意の行のセットを削除できます。これにより、パフォーマンスが向上する可能性もあります。これをワイド削除または配列削除といいます。

Embedded SQL を使用したワイドマージ [312 ページ]
MERGE 文は複数のローセットをテーブルにマージするように使用できます。こうするとパフォーマンスが向上することがあります。これをワイドマージまたは配列マージといいます。

Embedded SQL を使用して long 値を送信し、取得する方法 [316 ページ]
Embedded SQL アプリケーションで LONG VARCHAR、LONG NVARCHAR、LONG BINARY の値を送信し、取り出す方法は、他のデータ型とは異なります。

Embedded SQL での簡単なストアドプロシージャ [323 ページ]
Embedded SQL を使用して、ストアドプロシージャの作成と呼び出しを行うことができます。

Embedded SQL を使用した要求管理 [325 ページ]
通常の Embedded SQL アプリケーションは、各データベース要求が完了するまで待ってから次のステップを実行する必要があります。そのため、アプリケーションで複数の実行スレッドを使用することで、他のタスクと同時に実行できます。

Embedded SQL を使用したデータベースのバックアップ [326 ページ]
データベースのバックアップには BACKUP 文の使用をお奨めします。

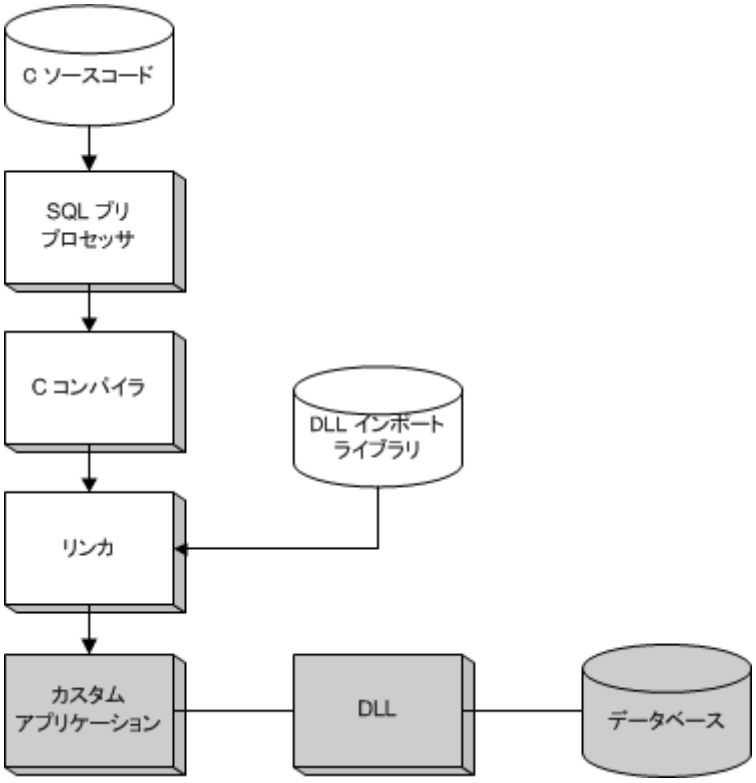
ライブラリ関数のリファレンス [326 ページ]
Embedded SQL プリプロセッサはインタフェースライブラリまたは DLL 内の関数呼び出しを生成します。Embedded SQL プリプロセッサが生成する呼び出しの他に、データベース操作を容易にする一連のライブラリ関数も用意されています。

Embedded SQL 文のまとめ [372 ページ]
必ず、Embedded SQL 文の前には EXEC SQL、後ろにはセミコロン (;) を付けてください。

1.10.1 開発プロセスの概要

プログラムのプリプロセッサ処理とコンパイルが成功すると、プログラムを DBLIB 用のインポートライブラリとリンクして、実行ファイルを作成できます。データベースサーバが実行中のとき、この実行ファイルは DBLIB を使用してデータベースサーバと交信します。

プログラムのプリプロセッサ処理はデータベースサーバが動作していなくても実行できます。



Windows では、Microsoft Visual C++ 用に 32 ビットと 64 ビットのインポートライブラリが用意されています。DLL で関数を呼び出すアプリケーションを開発する方法の 1 つとしてインポートライブラリを使用できます。ただし、ライブラリを動的にロードすることにより、インポートライブラリの使用を避けるほうをお奨めします。

関連情報

Windows での DBLIB の動的ロード [262 ページ]

1.10.2 Embedded SQL プリプロセッサ

Embedded SQL プリプロセッサの実行プログラム名は `sqlpp` です。

プリプロセッサのコマンドラインを次に示します。

```
sqlpp [ options ] sql-filename [ output-filename ]
```

プリプロセッサは、C または C++ のソースファイル内の Embedded SQL 文を C コードに変換し、その結果を出力ファイルに出力します。次に、C または C++ コンパイラを使用して出力ファイルを処理します。Embedded SQL を含んだソースプログラムの拡張子は通常 `.sqc` です。デフォルトの出力ファイル名は、`sql-filename` に拡張子 `.c` を付けたものになります。`sql-filename` に拡張子 `.c` が付いている場合、デフォルトの出力ファイル名拡張子は `.cc` に変更されます。

> **注記**
>
> 新しいメジャーバージョンのデータベースインタフェースライブラリを使用するようにアプリケーションを再構築するときは、同じバージョンの SQL プリプロセッサで Embedded SQL ファイルを前処理する必要があります。

プリプロセッサのオプションを次の表に示します。

オプション	説明
-d	データ領域サイズを減らすコードを生成します。データ構造体を再利用し、実行時に初期化してから使用します。これはコードサイズを増加させます。
-e`level`	指定した標準に含まれない静的 Embedded SQL をエラーとして通知します。`level` 値は、使用する標準を表す。たとえば、`sqlpp -e c03 ...` は、コア ANSI/ISO SQL 規格に含まれない構文を通知します。サポートされる `level` 値は、次のとおりです。 **c08** コア SQL/2008 構文でない構文を通知します。 **p08** 上級レベル SQL/2008 構文でない構文を通知します。 **c03** コア SQL/2003 構文でない構文を通知します。 **p03** 上級レベル SQL/2003 構文でない構文を通知します。 **c99** コア SQL/1999 構文でない構文を通知します。 **p99** 上級レベル SQL/1999 構文でない構文を通知します。 **e92** 初級レベル SQL/1992 構文でない構文を通知します。 **i92** 中級レベル SQL/1992 構文でない構文を通知します。 **f92** 上級レベル SQL/1992 構文でない構文を通知します。 **t** 標準ではないホスト変数型を通知します。 **u** Ultra Light がサポートしていない構文を通知します。 以前のバージョンのソフトウェアと互換性を保つために、e、i、f を指定することもできます。これらはそれぞれ e92、i92、f92 に対応します。

オプション	説明
*-h*width	sqlpp によって出力される行の最大長を width に制限します。行の内容が次の行に続くことを表す文字は円記号 (¥) です。また、width に指定できる最小値は 10 です。
-k	コンパイルされるプログラムが SQLCODE のユーザ宣言をインクルードすることをプリプロセッサに通知します。定義は long 型である必要がありますが、宣言セクション内で指定する必要はありません。
*-m*mode	Embedded SQL アプリケーションで明示的に指定されていない場合、カーソルの更新可能性モードを指定します。mode は次のいずれかを指定します。 **HISTORICAL** 以前のバージョンでは、Embedded SQL のカーソルは (クエリや ansi_update_constraints オプション値に応じて) デフォルトで FOR UPDATE または READ ONLY になっていました。明示的なカーソル更新可能性は DECLARE CURSOR で指定されました。この動作を維持するためには、このオプションを使用してください。 **READONLY** Embedded SQL のカーソルはデフォルトで READ ONLY になります。明示的なカーソル更新可能性は PREPARE で指定されます。これはデフォルトの動作です。READ ONLY カーソルによってパフォーマンスが向上する場合があります。
-n	C ファイルに行番号情報を生成します。これは、生成された C コード内の適切な場所にある #line ディレクティブで構成されます。使用しているコンパイラが #line ディレクティブをサポートしている場合、このオプションを使うと、コンパイラは SQC ファイル (Embedded SQL が含まれるファイル) の中の行番号を使ってその場所のエラーをレポートします。これは、SQL プリプロセッサによって生成された C ファイルの中の行番号を使って、その場所のエラーをレポートすることとは対照的です。また、ソースレベルデバッガも、#line ディレクティブを間接的に使用します。このため、SQC ソースファイルを表示しながらデバッグできます。
*-o*operating-system	ターゲットオペレーティングシステムを指定します。サポートされているオペレーティングシステムは次のとおりです。 **WINDOWS** Microsoft Windows **UNIX** 32 ビットの UNIX アプリケーションを作成している場合はこのオプションを指定します。 **UNIX64** 64 ビットの UNIX アプリケーションを作成している場合はこのオプションを指定します。
-q	クワイエットモード。メッセージを表示しません。

オプション	説明
-r-	再入力不可コードを生成します。
`-Slen`	プリプロセッサが C ファイルに出力する文字列の最大サイズを設定します。この値より長い文字列は、文字のリスト (`'a'`、`'b'`、`'c'` など) を使用して初期化されます。ほとんどの C コンパイラには、処理できる文字列リテラルのサイズに制限があります。このオプションを使用して上限を設定します。デフォルト値は 500 です。
-u	Ultra Light 用コードを生成します。
`-Wlevel`	指定した標準に含まれない静的 Embedded SQL を警告として通知します。`level` 値は、使用する標準を表す。たとえば `sqlpp -w c08 ...` は、コア SQL/2008 構文に含まれない SQL 構文を通知します。サポートされる `level` 値は、次のとおりです。 **c08** コア SQL/2008 構文でない構文を通知します。 **p08** 上級レベル SQL/2008 構文でない構文を通知します。 **c03** コア SQL/2003 構文でない構文を通知します。 **p03** 上級レベル SQL/2003 構文でない構文を通知します。 **c99** コア SQL/1999 構文でない構文を通知します。 **p99** 上級レベル SQL/1999 構文でない構文を通知します。 **e92** 初級レベル SQL/1992 構文でない構文を通知します。 **i92** 中級レベル SQL/1992 構文でない構文を通知します。 **f92** 上級レベル SQL/1992 構文でない構文を通知します。 **t** 標準ではないホスト変数型を通知します。 **u** Ultra Light がサポートしていない構文を通知します。 以前のバージョンのソフトウェアと互換性を保つために、e、i、f を指定することもできます。これらはそれぞれ e92、i92、f92 に対応します。
-x	マルチバイト文字列をエスケープシーケンスに変更して、コンパイラをパススルーできるようにします。

オプション	説明
-Zcs	照合順を指定します。推奨する照合順のリストを表示するには、コマンドプロンプトで dbinit -l を実行してください。 照合順は、プリプロセッサにプログラムのソースコードで使用されている文字を理解させるために使用します。たとえば、識別子に使用できるアルファベット文字の識別などに使用されます。-z が指定されていない場合、プリプロセッサは、オペレーティングシステムと SALANG および SACHARSET 環境変数に基づいて、使用する合理的な照合順を決定しようとします。
sql-filename	処理される Embedded SQL が含まれる C プログラムまたは C++ プログラム。
output-filename	Embedded SQL プリプロセッサが作成した C 言語のソースファイル。

関連情報

マルチスレッドまたは再入可能コードでの SQLCA 管理 [284 ページ]

1.10.3 対応コンパイラ

Windows コンパイラと UNIX コンパイラの両方が Embedded SQL プリプロセッサとともに使用されています。

次のコンパイラが使用されています。

オペレーティングシステム	コンパイラ	バージョン
Windows	Microsoft Visual C++	6.0 以降
UNIX	GNU またはネイティブコンパイラ	

1.10.4 Embedded SQL ヘッダファイル

ヘッダファイルはすべて、ソフトウェアインストールディレクトリの SDK¥Include サブディレクトリにインストールされています。

ファイル名	説明
sqlca.h	メインヘッダファイル。すべての Embedded SQL プログラムにインクルードされます。このファイルは SQLCA (SQL Communication Area) の構造体定義と、すべての Embedded SQL データベースインタフェース関数のプロトタイプを含みます。

ファイル名	説明
sqlda.h	SQLDA (SQL Descriptor Area) の構造体定義。動的 SQL を使用する Embedded SQL プログラムにインクルードされます。
sqldef.h	Embedded SQL インタフェースのデータ型定義。このファイルはデータベースサーバを C プログラムから起動するのに必要な構造体定義とリターンコードも含みます。
sqlerr.h	SQLCA の sqlcode フィールドに返されるエラーコードの定義。
sqlstate.h	SQLCA の sqlstate フィールドに返される ANSI/ISO SQL 標準エラーステータスの定義。
pshpk1.h, pshpk4.h, poppk.h	構造体のパックを正しく処理するためのヘッダ。

1.10.5 インポートライブラリ

インポートライブラリは、ソフトウェアインストールディレクトリのサブディレクトリにインストールされています。

Windows プラットフォーム上では、インポートライブラリはすべて、ソフトウェアインストールディレクトリの SDK¥Lib サブディレクトリにインストールされています。Windows 用のインポートライブラリは、SDK¥Lib¥x86 および SDK¥Lib¥x64 サブディレクトリに格納されています。エクスポート定義のリストは、SDK¥Lib¥Def¥dblib.def に格納されています。

UNIX プラットフォーム上では、インポートライブラリはすべて、ソフトウェアインストールディレクトリの lib32 および lib64 サブディレクトリにインストールされています。

Mac OS X プラットフォーム上では、インポートライブラリはすべて、ソフトウェアインストールディレクトリの System/lib32 および System/lib64 サブディレクトリにインストールされています。

オペレーティングシステム	コンパイラ	インポートライブラリ
Windows	Microsoft Visual C++	dblibtm.lib
UNIX (非スレッドアプリケーション)	全コンパイラ	libdblib*17*.so、 libdbtasks*17*.so、 libdblib*17*.sl、 libdbtasks*17*.sl
UNIX (スレッドアプリケーション)	全コンパイラ	libdblib*17*_r.so、 libdbtasks*17*_r.so、 libdblib*17*_r.sl、 libdbtasks*17*_r.sl
Mac OS X (スレッドアプリケーション)	全コンパイラ	libdblib*17*.dylib、 libdbtasks*17*.dylib
Mac OS X (スレッドアプリケーション)	全コンパイラ	libdblib*17*_r.dylib、 libdbtasks*17*_r.dylib

libdbtasks*17* ライブラリは、libdblib*17* ライブラリに呼び出されます。コンパイラの中には、libdbtasks*17* を自動的に検出するものもあります。そうでない場合は、ユーザが明示的に指定する必要があります。

1.10.6 Embedded SQL サンプルプログラム

ここで、Embedded SQL プログラムの非常に簡単な例を示します。

```
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;
main()
{
  db_init( &sqlca );
  EXEC SQL WHENEVER SQLERROR GOTO error;
  EXEC SQL CONNECT "DBA" IDENTIFIED BY "sql";
  EXEC SQL UPDATE Employees
    SET Surname =  'Plankton'
    WHERE EmployeeID = 195;
  EXEC SQL COMMIT WORK;
  EXEC SQL DISCONNECT;
  db_fini( &sqlca );
  return( 0 );
error:
  printf( "update unsuccessful -- sqlcode = %ld¥n",
    sqlca.sqlcode );
  db_fini( &sqlca );
  return( -1 );
}
```

この例では、データベースに接続して、従業員番号 195 の姓を更新し、変更内容をコミットして終了します。Embedded SQL コードと C コード間のやり取りはまったくありません。この例では、C コードはフロー制御だけに使用されています。WHENEVER 文はエラーチェックに使用されています。エラーアクション (この例では GOTO) はエラーを起こした SQL 文の後で実行されます。

関連情報

Embedded SQL を使用してデータをフェッチする方法 [299 ページ]

1.10.7 Embedded SQL プログラムの構造

SQL 文は通常の C または C++ コードの内部に置かれ (埋め込まれ) ます。Embedded SQL 文は、必ず、EXEC SQL で始まり、セミコロン (;) で終わります。

Embedded SQL 文の途中に、C 言語の通常のコメントを記述できます。

Embedded SQL を使用する C プログラムでは、ソースファイル内のどの Embedded SQL 文よりも前に、必ず次の文を置きます。

```
EXEC SQL INCLUDE SQLCA;
```

Embedded SQL を使用するすべての C プログラムは、初めに SQLCA を初期化する必要があります。

```
db_init( &sqlca );
```

C プログラムが最初に実行する Embedded SQL 文の 1 つは、CONNECT 文である必要があります。CONNECT 文はデータベースサーバに接続し、ユーザ ID を指定します。このユーザ ID は接続中に実行されるすべての SQL 文の認証に使用されます。

Embedded SQL 文には C コードを生成しないものや、データベースとのやりとりをしないものもあります。このような文は CONNECT 文の前に記述できます。よく使われるのは、INCLUDE 文と、エラー処理を指定する WHENEVER 文です。

Embedded SQL を使用したすべての C プログラムは、初期化された SQLCA をすべてファイナライズする必要があります。

```
db_fini( &sqlca );
```

1.10.8 Windows での DBLIB の動的ロード

インポートライブラリとリンクしなくてもよいように、ソフトウェアインストールディレクトリの `SDK¥C` サブディレクトリにある `esqldll.c` モジュールを使用して、Embedded SQL アプリケーションから DBLIB を動的にロードします。

コンテキスト

このタスクは、アプリケーションを必要な関数定義が含まれるダイナミックリンクライブラリ (DLL) の静的インポートライブラリにリンクする通常の方法の代替となる方法です。

UNIX プラットフォームで DBLIB を動的にロードする場合にも同様のタスクを使用できます。

手順

1. アプリケーションでは、db_init_dll を呼び出して DBLIB DLL をロードし、db_fini_dll を呼び出して DBLIB DLL を解放します。db_init_dll はすべてのデータベースインタフェース関数の前に呼び出してください。db_fini_dll の後には、インタフェース関数の呼び出しはできません。

   db_init と db_fini ライブラリ関数も呼び出してください。
2. Embedded SQL プログラムでは、EXEC SQL INCLUDE SQLCA 文の前に `esqldll.h` ヘッダファイルをインクルードするか、`sqlca.h` をインクルードしてください。`esqldll.h` ヘッダファイルは `sqlca.h` をインクルードします。
3. SQL の OS マクロを定義します。`sqlca.h` にインクルードされるヘッダファイル `sqlos.h` は、適切なマクロを判断して、定義しようとします。しかし、プラットフォームとコンパイラの組み合わせによっては、定義に失敗することがあります。その場合は、このヘッダファイルの先頭に #define を追加するか、コンパイラオプションを使用してマクロを定義してください。どの Windows オペレーティングシステムでも _SQL_OS_WINDOWS を定義します。
4. `esqldll.c` をコンパイルします。
5. インポートライブラリにリンクする代わりに、オブジェクトモジュール `esqldll.obj` を Embedded SQL アプリケーションオブジェクトにリンクします。

結果

Embedded SQL アプリケーションを実行すると、DBLIB インタフェース DLL が動的にロードされます。

例

インタフェースライブラリを動的にロードする方法を示すサンプルプログラムは、*%SQLANYSAMP17%*¥SQLAnywhere¥ESQLDynamicLoad ディレクトリにあります。ソースコードは sample.sqc にあります。

次の例では、Windows で esqldll.c からのコードを使用して、sample.sqc をコンパイルしてリンクします。

```
sqlpp sample.sqc
cl sample.c %SQLANY17%¥sdk¥c¥esqldll.c /I%SQLANY17%¥sdk¥include Advapi32.lib
```

1.10.9 サンプル Embedded SQL プログラム

ソフトウェアには、サンプル Embedded SQL プログラムが含まれています。

サンプルプログラムは *%SQLANYSAMP17%*¥SQLAnywhere¥C ディレクトリにあります。

- 静的カーソルを使用した Embedded SQL サンプルである cur.sqc は静的 SQL 文の使い方を示します。
- 動的カーソルを使用した Embedded SQL サンプルである dcur.sqc は、動的 SQL 文の使い方を示します。

サンプルプログラムで重複するコードの量を減らすために、メインライン部分とデータ出力関数は別ファイルになっています。これは、文字モードシステムでは mainch.c、ウィンドウ環境では mainwin.c です。

サンプルプログラムには、それぞれ次の 3 つのルーチンがあり、メインライン部分から呼び出されます。

WSQLEX_Init

データベースに接続し、カーソルを開きます。

WSQLEX_Process_Command

ユーザのコマンドを処理し、必要に応じてカーソルを操作します。

WSQLEX_Finish

カーソルを閉じ、データベースとの接続を切断します。

メインライン部分の機能を次に示します。

1. WSQLEX_Init ルーチンを呼び出します。
2. ユーザからコマンドを受け取り、ユーザが終了するまで WSQL_Process_Command を呼び出して、ループします。
3. WSQLEX_Finish ルーチンを呼び出します。

データベースへの接続は Embedded SQL の CONNECT 文に適切なユーザ ID とパスワードを指定して実行します。

これらのサンプルに加えて、ソフトウェアには、特定のプラットフォームで使用できる機能を例示するプログラムとソースファイルも用意されています。

このセクションの内容:

静的カーソルのサンプル [264 ページ]

これはカーソル使用法の例です。

静的カーソルのサンプルプログラムの実行 [265 ページ]
静的カーソルのサンプルプログラムを実行します。

動的カーソルのサンプル [266 ページ]
このサンプルは、動的 SQL SELECT 文でのカーソルの使用方法を示しています。

動的カーソルのサンプルプログラムの実行 [267 ページ]
動的カーソルのサンプルプログラムを実行します。

1.10.9.1 静的カーソルのサンプル

これはカーソル使用法の例です。

ここで使用されているカーソルはサンプルデータベースの Employees テーブルから情報を取り出します。カーソルは静的に宣言されています。つまり、情報を取り出す実際の SQL 文はソースプログラムにハードコードされています。この例はカーソルの機能を理解するには格好の出発点です。動的カーソルのサンプルでは、この最初のサンプルを使って、これを動的 SQL 文を使用するものに書き換えます。

open_cursor ルーチンは、特定の SQL クエリ用のカーソルを宣言し、同時にカーソルを開きます。

1 ページ分の情報の表示は print ルーチンが行います。このルーチンは、カーソルから 1 つのローをフェッチして表示する動作を `pagesize` 回繰り返します。フェッチルーチンは警告条件 (「ローが見つかりません (SQLCODE 100)」など) を検査し、そうした条件が見つかった場合に適切なメッセージを表示します。また、このプログラムは、カーソルの位置を現在のデータページの先頭に表示されているローの前に変更します。

move、top、bottom ルーチンは適切な形式の FETCH 文を使用して、カーソルを位置付けます。この形式の FETCH 文は実際のデータの取得はしません。単にカーソルを位置付けるだけです。また、汎用の相対位置付けルーチン move はパラメータの符号に応じて移動方向を変えるように実装されています。

ユーザがプログラムを終了すると、カーソルは閉じられ、データベース接続も解放されます。カーソルは ROLLBACK WORK 文によって閉じられ、接続は DISCONNECT によって解放されます。

関連情報

静的カーソルのサンプルプログラムの実行 [265 ページ]
動的カーソルのサンプル [266 ページ]

1.10.9.2 静的カーソルのサンプルプログラムの実行

静的カーソルのサンプルプログラムを実行します。

前提条件

Windows の場合、最新バージョンの Microsoft Visual Studio が必要です。

Microsoft Visual Studio による x86/x64 プラットフォームのビルドでは、コンパイルとリンクに適した環境を設定する必要があります。これは通常、Microsoft Visual Studio の `vcvars32.bat` または `vcvars64.bat` (Microsoft Visual Studio の旧バージョンでは `vcvarsamd64.bat`) を使用して行います。

コンテキスト

実行ファイルと対応するソースコードは *%SQLANYSAMP17%*`¥SQLAnywhere¥C` ディレクトリにあります。

手順

1. サンプルデータベース *demo.db* を起動します。
2. サンプルプログラムの構築用ファイルには、サンプルコードが用意されています。

   Windows では、`build.bat` バッチファイルを実行します。

   ビルドのエラーが発生した場合は、ターゲットプラットフォーム (`x86` または `x64`) を `build.bat` の引数として指定します。次に例を示します。

   ```
   build x64
   ```

   UNIX では、サンプルの構築にシェルスクリプトの `build.sh` を使用してください。
3. 32 ビットの Windows の例では、ファイル `curwin.exe` を実行します。

   64 ビットの Windows の例では、ファイル `curx64.exe` を実行します。

   UNIX の例では、ファイル `cur` を実行します。
4. 画面に表示される指示に従います。

結果

さまざまなコマンドでデータベースカーソルを操作し、クエリ結果を画面に出力できます。実行するコマンド文字を入力してください。システムによっては、文字入力の後、[Enter] キーを押す必要があります。

1.10.9.3 動的カーソルのサンプル

このサンプルは、動的 SQL SELECT 文でのカーソルの使用方法を示しています。

動的カーソルのサンプルプログラム (dcur) では、ユーザは N コマンドによって参照したいテーブルを選択できます。プログラムは、そのテーブルの情報を画面に入るかぎり表示します。

起動したら、プロンプトに対して次の接続文字列を入力してください。次はその例です。

```
UID=DBA;PWD=sql;DBF=demo.db
```

Embedded SQL を使用する C プログラムは、*%SQLANYSAMP17%*¥SQLAnywhere¥C ディレクトリにあります。

dcur プログラムは Embedded SQL インタフェース関数の db_string_connect を使用してデータベースに接続します。この関数はデータベース接続に使用する接続文字列をサポートします。

open_cursor ルーチンは、まず SELECT 文を作成します。

```
SELECT * FROM table-name
```

table-name はルーチンに渡されたパラメータです。この文字列を使用して動的 SQL 文を準備します。

Embedded SQL の DESCRIBE 文は、SELECT 文の結果を SQLDA 構造体に設定するために使用されます。

> **注記**
>
> SQLDA のサイズの初期値は 3 になっています。この値が小さすぎる場合、データベースサーバの返した SELECT リストの実際のサイズを使用して、正しいサイズの SQLDA を割り付けます。
>
> その後、SQLDA 構造体にはクエリの結果を示す文字列を保持するバッファが設定されます。fill_s_sqlda ルーチンは SQLDA のすべてのデータ型を DT_STRING 型に変換し、適切なサイズのバッファを割り付けます。

その後、この文のためのカーソルを宣言して開きます。カーソルを移動して閉じるその他のルーチンは同じです。

fetch ルーチンの場合は多少異なり、ホスト変数のリストの代わりに、SQLDA 構造体に結果を入れます。print ルーチンは大幅に変更され、SQLDA 構造体から結果を取り出して画面の幅一杯まで表示します。print ルーチンは各カラムの見出しを表示するために SQLDA の名前フィールドも使用します。

関連情報

動的カーソルのサンプルプログラムの実行 [267 ページ]
静的カーソルのサンプル [264 ページ]

1.10.9.4 動的カーソルのサンプルプログラムの実行

動的カーソルのサンプルプログラムを実行します。

前提条件

Windows の場合、最新バージョンの Microsoft Visual Studio が必要です。

Microsoft Visual Studio による x86/x64 プラットフォームのビルドでは、コンパイルとリンクに適した環境を設定する必要があります。これは通常、Microsoft Visual Studio の `vcvars32.bat` または `vcvars64.bat` (Microsoft Visual Studio の旧バージョンでは `vcvarsamd64.bat`) を使用して行います。

コンテキスト

実行ファイルと対応するソースコードは *%SQLANYSAMP17%*`¥SQLAnywhere¥C` ディレクトリにあります。

手順

1. サンプルデータベース *demo.db* を起動します。
2. サンプルプログラムの構築用ファイルには、サンプルコードが用意されています。

   Windows では、`build.bat` バッチファイルを実行します。

   ビルドのエラーが発生した場合は、ターゲットプラットフォーム (`x86` または `x64`) を `build.bat` の引数として指定します。次に例を示します。

   ```
   build x64
   ```

   UNIX では、サンプルの構築にシェルスクリプトの `build.sh` を使用してください。
3. 32 ビットの Windows の例では、ファイル `dcurwin.exe` を実行します。

   64 ビットの Windows の例では、ファイル `dcurx64.exe` を実行します。

   ストアドプロシージャまたは関数から外部ライブラリの関数を呼び出すことができます。Windows オペレーティングシステムでは DLL、UNIX では共有オブジェクトの関数を呼び出すことができます。

   UNIX の例では、ファイル `dcur` を実行します。
4. 各サンプルプログラムのユーザインタフェースはコンソールタイプであり、プロンプトでコマンドを入力して操作します。次の接続文字列を入力してサンプルデータベースに接続します。

   ```
   DSN=SQL Anywhere 17 Demo;PWD=sql
   ```
5. 各サンプルプログラムでテーブルを選択するように要求されます。サンプルデータベース内のテーブルを 1 つ選択します。たとえば、**`Customers`** または **`Employees`** を入力します。

6. 画面に表示される指示に従います。

結果

さまざまなコマンドでデータベースカーソルを操作し、クエリ結果を画面に出力できます。実行するコマンド文字を入力してください。システムによっては、文字入力の後、[Enter] キーを押す必要があります。

1.10.10 Embedded SQL のデータ型

プログラムとデータベースサーバ間で情報を転送するには、それぞれのデータについてデータ型を設定します。

Embedded SQL データ型定数の前には DT_ が付けられ、`sqldef.h` ヘッダファイル内にあります。ホスト変数はサポートされるどのデータ型についても作成できます。これらのデータ型は、データをデータベースと受け渡しするために SQLDA 構造体で使用することもできます。

これらのデータ型の変数を定義するには、`sqlca.h` にリストされている DECL_ マクロを使用します。たとえば、変数が BIGINT 値を保持する場合は DECL_BIGINT と宣言できます。

次のデータ型が、Embedded SQL プログラミングインタフェースでサポートされます。

DT_BIT

8 ビット符号付き整数。
DT_SMALLINT

16 ビット符号付き整数。
DT_UNSSMALLINT

16 ビット符号なし整数です。
DT_TINYINT

8 ビット符号付き整数。
DT_BIGINT

64 ビット符号付き整数。
DT_UNSBIGINT

64 ビット符号なし整数です。
DT_INT

32 ビット符号付き整数。
DT_UNSINT

32 ビット符号なし整数です。
DT_FLOAT

4 バイト浮動小数点数。
DT_DOUBLE

8 バイト浮動小数点数。

DT_DECIMAL

パック 10 進数 (独自フォーマット)。

```
typedef struct TYPE_DECIMAL {
  char array[1];
} TYPE_DECIMAL;
```

DT_STRING

CHAR 文字セット内の NULL で終了する文字列。データベースがブランクを埋め込まれた文字列で初期化されると、文字列にブランクが埋め込まれます。

DT_NSTRING

NCHAR 文字セット内の NULL で終了する文字列。データベースがブランクを埋め込まれた文字列で初期化されると、文字列にブランクが埋め込まれます。

DT_DATE

有効な日付データを含み、NULL で終了する文字列。

DT_TIME

有効な時間データを含み、NULL で終了する文字列。

DT_TIMESTAMP

有効なタイムスタンプを含み、NULL で終了する文字列。

DT_FIXCHAR

CHAR 文字セット内のブランクが埋め込まれた固定長文字列。最大長は 32767 で、バイト単位で指定します。データは NULL で終了しません。

DT_NFIXCHAR

NCHAR 文字セット内のブランクが埋め込まれた固定長文字列。最大長は 32767 で、バイト単位で指定します。データは NULL で終了しません。

DT_VARCHAR

CHAR 文字セット内の 2 バイトの長さフィールドを持つ可変長文字列。データを送信する場合は、長さフィールドに値を設定してください。送信できる最大長は 32767 バイトです。データをフェッチする場合は、データベースサーバが長さフィールドに値を設定します。フェッチできる最大長は 32767 バイトです。データは NULL で終了せず、ブランクも埋め込まれません。sqldata フィールドは、ちょうど sqllen + 2 バイトの長さのこのデータ領域を指します。

```
typedef struct VARCHAR {
 a_sql_ulen len;
 char       array[1];
} VARCHAR;
```

DT_NVARCHAR

NCHAR 文字セット内の 2 バイトの長さフィールドを持つ可変長文字列。データを送信する場合は、長さフィールドに値を設定してください。送信できる最大長は 32767 バイトです。データをフェッチする場合は、データベースサーバが長さフィールドに値を設定します。フェッチできる最大長は 32767 バイトです。データは NULL で終了せず、ブランクも埋め込まれません。sqldata フィールドは、ちょうど sqllen + 2 バイトの長さのこのデータ領域を指します。

```
typedef struct NVARCHAR {
 a_sql_ulen len;
 char       array[1];
} NVARCHAR;
```

DT_LONGVARCHAR

CHAR 文字セット内の長い可変長文字列。

```
typedef struct LONGVARCHAR {
 a_sql_uint32 array_len;  /* number of allocated bytes in array */
 a_sql_uint32 stored_len; /* number of bytes stored in array
                           * (never larger than array_len) */
 a_sql_uint32 untrunc_len;/* number of bytes in untruncated expression
                           * (may be larger than array_len) */
 char         array[1];   /* the data */
} LONGVARCHAR, LONGNVARCHAR, LONGBINARY;
```

32767 バイトを超えるデータには、LONGVARCHAR 構造体を使用できます。このように大きいデータの場合は、全体を一度にフェッチする方法と、GET DATA 文を使用して分割してフェッチする方法があります。また、サーバに対しても、全体を一度に送信する方法と、SET 文を使用してデータベース変数に追加することで分割して送信する方法があります。データは NULL で終了せず、ブランクも埋め込まれません。

DT_LONGNVARCHAR

NCHAR 文字セット内の長い可変長文字列。マクロによって、構造体が次のように定義されます。

```
typedef struct LONGVARCHAR {
 a_sql_uint32 array_len;  /* number of allocated bytes in array */
 a_sql_uint32 stored_len; /* number of bytes stored in array
                           * (never larger than array_len) */
 a_sql_uint32 untrunc_len;/* number of bytes in untruncated expression
                           * (may be larger than array_len) */
 char         array[1];   /* the data */
} LONGVARCHAR, LONGNVARCHAR, LONGBINARY;
```

32767 バイトを超えるデータには、LONGNVARCHAR 構造体を使用できます。このように大きいデータの場合は、全体を一度にフェッチする方法と、GET DATA 文を使用して分割してフェッチする方法があります。また、サーバに対しても、全体を一度に送信する方法と、SET 文を使用してデータベース変数に追加することで分割して送信する方法があります。データは NULL で終了せず、ブランクも埋め込まれません。

DT_BINARY

2 バイトの長さフィールドを持つ可変長バイナリデータ。データを送信する場合は、長さフィールドに値を設定してください。送信できる最大長は 32767 バイトです。データをフェッチする場合は、データベースサーバが長さフィールドに値を設定します。フェッチできる最大長は 32767 バイトです。データは NULL で終了せず、ブランクも埋め込まれません。sqldata フィールドは、ちょうど sqllen + 2 バイトの長さのこのデータ領域を指します。

```
typedef struct BINARY {
 a_sql_ulen len;
 char       array[1];
} BINARY;
```

DT_LONGBINARY

長いバイナリデータ。マクロによって、構造体が次のように定義されます。

```
typedef struct LONGVARCHAR {
 a_sql_uint32 array_len;  /* number of allocated bytes in array */
 a_sql_uint32 stored_len; /* number of bytes stored in array
                           * (never larger than array_len) */
 a_sql_uint32 untrunc_len;/* number of bytes in untruncated expression
                           * (may be larger than array_len) */
 char         array[1];   /* the data */
} LONGVARCHAR, LONGNVARCHAR, LONGBINARY;
```

32767 バイトを超えるデータには、LONGBINARY 構造体を使用できます。このように大きいデータの場合は、全体を一度にフェッチする方法と、GET DATA 文を使用して分割してフェッチする方法があります。また、サーバに対しても、全体を一度に送信する方法と、SET 文を使用してデータベース変数に追加することで分割して送信する方法があります。

DT_TIMESTAMP_STRUCT

タイムスタンプの各部分に対応するフィールドを持つ SQLDATETIME 構造体。

```
typedef struct sqldatetime {
 unsigned short year; /* for example 1999 */
 unsigned char  month; /* 0-11 */
 unsigned char  day_of_week; /* 0-6 0=Sunday */
 unsigned short day_of_year; /* 0-365 */
 unsigned char  day; /* 1-31 */
 unsigned char  hour; /* 0-23 */
 unsigned char  minute; /* 0-59 */
 unsigned char  second; /* 0-59 */
 unsigned long  microsecond; /* 0-999999 */
} SQLDATETIME;
```

SQLDATETIME 構造体は、型が DATE、TIME、TIMESTAMP (または、いずれかの型に変換できるもの) のフィールドを取り出すのに使用できます。アプリケーションは、日付に関して独自のフォーマットで処理をすることがあります。この構造体の中のデータをフェッチすると、このデータを簡単に操作できます。また、型が DATE、TIME、TIMESTAMP のフィールドは、文字型であれば、どの型でもフェッチと更新が可能です。

SQLDATETIME 構造体を介してデータベースに日付、時刻、またはタイムスタンプを入力しようとすると、day_of_year と day_of_week メンバーは無視されます。

DT_VARIABLE

NULL で終了する文字列。文字列は SQL 変数名です。その変数の値をデータベースサーバが使用します。このデータ型はデータベースサーバにデータを与えるときにだけ使用されます。データベースサーバからデータをフェッチするときには使用できません。

これらの構造体は `sqlca.h` ファイルに定義されています。VARCHAR、NVARCHAR、BINARY、DECIMAL、LONG の各データ型は、データ格納領域が長さ 1 の文字配列のため、ホスト変数の宣言には向いていません。しかし、動的な変数の割り付けや他の変数の型変換を行うのには有効です。

データベースの DATE 型と TIME 型

データベースのさまざまな DATE 型と TIME 型に対応する、Embedded SQL インタフェースのデータ型はありません。これらの型はすべて SQLDATETIME 構造体または文字列を使用してフェッチと更新を行います。

関連情報

Embedded SQL を使用して long 値を送信し、取得する方法 [316 ページ]

1.10.11 Embedded SQL のホスト変数

ホスト変数とは、Embedded SQL プリプロセッサが認識する C 変数です。ホスト変数はデータベースサーバに値を送ったり、データベースサーバから値を受け取ったりするのに使用できます。

ホスト変数はとても使いやすいものですが、制限もあります。SQLDA (SQL Descriptor Area) という構造体を使用するデータベースサーバと情報をやりとりするには、動的 SQL の方が一般的です。Embedded SQL プリプロセッサは、ホスト変数が使用されている文ごとに SQLDA を自動的に生成します。

バッチではホスト変数を使用できません。SET 文のサブクエリ内ではホスト変数を使用できません。

このセクションの内容:

Embedded SQL ホスト変数の宣言 [272 ページ]
ホスト変数は、宣言セクションに配置して定義します。ANSI Embedded SQL 標準によると、ホスト変数は通常の C の変数宣言を次のように囲んで定義します。

Embedded SQL ホスト変数データ型 [273 ページ]
ホスト変数として使用できる C のデータ型は非常に限られています。また、ホスト変数の型には、対応する C の型がないものもあります。

Embedded SQL ホスト変数の使用方法 [276 ページ]
ホスト変数はさまざまな状況で使用できます。

Embedded SQL インジケータ変数 [278 ページ]
インジケータ変数とは、データのやりとりをするときに補足的な情報を保持する C 変数のことです。

関連情報

静的 SQL と動的 SQL [286 ページ]

1.10.11.1 Embedded SQL ホスト変数の宣言

ホスト変数は、宣言セクションに配置して定義します。ANSI Embedded SQL 標準によると、ホスト変数は通常の C の変数宣言を次のように囲んで定義します。

```
EXEC SQL BEGIN DECLARE SECTION;
/* C variable declarations */
EXEC SQL END DECLARE SECTION;
```

こうして定義されたホスト変数は、任意の SQL 文で値定数の代わりに使用できます。データベースサーバが文を実行するときは、ホスト変数の値が使用されます。ホスト変数をテーブル名やカラム名の代わりに使用することはできません。その場合は動的 SQL が必要です。ホスト変数については、SQL 文の中で許容される他の識別子と区別するために、変数名の前にコロン (:) を付けます。

Embedded SQL プリプロセッサを使用すると、DECLARE SECTION 内でのみ C 言語コードがスキャンされます。したがって、DECLARE SECTION 内では TYPEDEF 型および構造体は使用できませんが、変数の初期化は行えます。

例

INSERT 文でホスト変数を使用するコード例です。プログラム側で変数に値を設定してから、データベースに挿入しています。

```
EXEC SQL BEGIN DECLARE SECTION;
long employee_number;
char employee_name[50];
char employee_initials[8];
char employee_phone[15];
EXEC SQL END DECLARE SECTION;
/* program fills in variables with appropriate values
*/
EXEC SQL INSERT INTO Employees
VALUES (:employee_number, :employee_name,
:employee_initials, :employee_phone );
```

関連情報

静的カーソルのサンプル [264 ページ]

1.10.11.2 Embedded SQL ホスト変数データ型

ホスト変数として使用できる C のデータ型は非常に限られています。また、ホスト変数の型には、対応する C の型がないものもあります。

`sqlca.h` ヘッダファイルに定義されているマクロを使用すると、NCHAR、VARCHAR、NVARCHAR、LONGVARCHAR、LONGNVARCHAR、BINARY、LONGBINARY、DECIMAL、DT_FIXCHAR、DT_NFIXCHAR、DATETIME (SQLDATETIME)、BIT、BIGINT、または UNSIGNED BIGINT 型のホスト変数を宣言できます。マクロは次のように使用します。

```
EXEC SQL BEGIN DECLARE SECTION;
DECL_NCHAR                  v_nchar[10];
DECL_VARCHAR( 10 )          v_varchar;
DECL_NVARCHAR( 10 )         v_nvarchar;
DECL_LONGVARCHAR( 32768 )   v_longvarchar;
DECL_LONGNVARCHAR( 32768 )  v_longnvarchar;
DECL_BINARY( 4000 )         v_binary;
DECL_LONGBINARY( 128000 )   v_longbinary;
DECL_DECIMAL( 30, 6 )       v_decimal;
DECL_FIXCHAR( 10 )          v_fixchar;
DECL_NFIXCHAR( 10 )         v_nfixchar;
DECL_DATETIME               v_datetime;
DECL_BIT                    v_bit;
DECL_BIGINT                 v_bigint;
DECL_UNSIGNED_BIGINT        v_ubigint;
EXEC SQL END DECLARE SECTION;
```

プリプロセッサは、Embedded SQL 宣言セクション内のこれらのマクロを認識し、変数を適切な型として処理します。10 進数のフォーマットは独自フォーマットであるため、DECIMAL (DT_DECIMAL, DECL_DECIMAL) 型を使用しないでください。

次の表は、ホスト変数で使用できる C 変数の型と、対応する Embedded SQL インタフェースのデータ型を示します。

C データ型	Embedded SQL のインタフェースのデータ型	説明
`short si;` `short int si;`	DT_SMALLINT	16 ビット符号付き整数。
`unsigned short int usi;`	DT_UNSSMALLINT	16 ビット符号なし整数です。
`long l;` `long int l;`	DT_INT	32 ビット符号付き整数。
`unsigned long int ul;`	DT_UNSINT	32 ビット符号なし整数です。
`DECL_BIGINT ll;`	DT_BIGINT	64 ビット符号付き整数。
`DECL_UNSIGNED_BIGINT ull;`	DT_UNSBIGINT	64 ビット符号なし整数です。
`float f;`	DT_FLOAT	4 バイトの単精度浮動小数点値です。
`double d;`	DT_DOUBLE	8 バイトの倍精度浮動小数点値です。
`char a[n]; /*n>=1*/`	DT_STRING	CHAR 文字セット内の NULL で終了する文字列ですです。データベースがブランクを埋め込まれた文字列で初期化されると、文字列にブランクが埋め込まれます。この変数には、n-1 バイトと NULL ターミネータが保持されます。
`char *a;`	DT_STRING	CHAR 文字セット内の NULL で終了する文字列ですです。この変数は、最大 32766 バイトと NULL ターミネータを保持できる領域を指します。
`DECL_NCHAR a[n]; /*n>=1*/`	DT_NSTRING	NCHAR 文字セット内の NULL で終了する文字列です。データベースがブランクを埋め込まれた文字列で初期化されると、文字列にブランクが埋め込まれます。この変数には、n-1 バイトと NULL ターミネータが保持されます。
`DECL_NCHAR *a;`	DT_NSTRING	NCHAR 文字セット内の NULL で終了する文字列です。この変数は、最大 32766 バイトと NULL ターミネータを保持できる領域を指します。
`DECL_VARCHAR(n) a;`	DT_VARCHAR	CHAR 文字セット内の 2 バイトの長さフィールドを持つ可変長文字列です。文字列は NULL で終了せず、ブランクも埋め込まれません。n の最大値は 32767 (バイト単位) です。

C データ型	Embedded SQL のインタフェースのデータ型	説明
`DECL_NVARCHAR(n) a;`	DT_NVARCHAR	NCHAR 文字セット内の 2 バイトの長さフィールドを持つ可変長文字列です。文字列は NULL で終了せず、ブランクも埋め込まれません。n の最大値は 32767 (バイト単位) です。
`DECL_LONGVARCHAR(n) a;`	DT_LONGVARCHAR	CHAR 文字セット内の 4 バイトの長さフィールドを 3 つ持つ長い可変長文字列です。文字列は NULL で終了せず、ブランクも埋め込まれません。
`DECL_LONGNVARCHAR(n) a;`	DT_LONGNVARCHAR	NCHAR 文字セット内の 4 バイトの長さフィールドを 3 つ持つ長い可変長文字列です。文字列は NULL で終了せず、ブランクも埋め込まれません。
`DECL_BINARY(n) a;`	DT_BINARY	2 バイトの長さフィールドを持つ可変長バイナリデータです。n の最大値は 32767 (バイト単位) です。
`DECL_LONGBINARY(n) a;`	DT_LONGBINARY	4 バイトの長さフィールドを 3 つ持つ長い可変長バイナリデータです。
`char a; /*n=1*/ DECL_FIXCHAR(n) a;`	DT_FIXCHAR	CHAR 文字セット内の固定長文字列です。ブランクが埋め込まれますが、NULL で終了しません。n の最大値は 32767 (バイト単位) です。
`DECL_NCHAR a; /*n=1*/ DECL_NFIXCHAR(n) a;`	DT_NFIXCHAR	NCHAR 文字セット内の固定長文字列です。ブランクが埋め込まれますが、NULL で終了しません。n の最大値は 32767 (バイト単位) です。
`DECL_DATETIME a;`	DT_TIMESTAMP_STRUCT	SQLDATETIME 構造体です。

文字セット

DT_FIXCHAR、DT_STRING、DT_VARCHAR、DT_LONGVARCHAR の場合、文字データはアプリケーションの CHAR 文字セット内にあります。この文字セットは、通常、アプリケーションのロケールの文字セットです。アプリケーションでは、CHARSET 接続パラメータを使用するか、db_change_char_charset 関数を呼び出すことで CHAR 文字セットを変更できます。

DT_NFIXCHAR、DT_NSTRING、DT_NVARCHAR、DT_LONGNVARCHAR の場合、文字データはアプリケーションの NCHAR 文字セット内にあります。デフォルトでは、アプリケーションの NCHAR 文字セットは CHAR 文字セットと同じです。アプリケーションでは、db_change_nchar_charset 関数を呼び出すことで NCHAR 文字セットを変更できます。

データの長さ

使用している CHAR や NCHAR 文字セットに関係なく、すべてのデータ長はバイトで指定します。

サーバとアプリケーションの間で文字セットを変換する場合は、変換されたデータを処理するためのバッファが十分に確保されていることを確認し、データがトランケートされた場合に追加の GET DATA 文を発行するのはアプリケーション側の責任です。

文字ポインタ

pointer to char (`char * a`) として宣言されたホスト変数は、データベースインタフェースでは 32767 バイトの長さであると見なされます。pointer to char 型のホスト変数を使用してデータベースから情報を取り出す場合は、ポインタの指すバッファを、データベースから返ってくる可能性のある値を格納するのに十分な大きさにしてください。

これはかなりの危険性があります。プログラムが作成された後でデータベースのカラムの定義が変更され、カラムのサイズが大きくなっている可能性があるからです。そうなると、ランダムメモリが破壊される可能性があります。関数のパラメータに pointer to char を渡す場合でも、配列を宣言して使用する方が安全です。この方法により、Embedded SQL 文で配列のサイズを知ることができます。

ホスト変数のスコープ

標準のホスト変数の宣言セクションは、C 変数を宣言できる通常の場所であれば、どこにでも記述できます。C の関数のパラメータの宣言セクションにも記述できます。C 変数は通常のスコープを持っています (定義されたブロック内で使用可能)。ただし、Embedded SQL プリプロセッサは C コードをスキャンしないため、C ブロックを重視しません。

Embedded SQL プリプロセッサに関しては、ホスト変数はソースファイルにおいてグローバルです。同じ名前のホスト変数は使用できません。

関連情報

db_change_char_charset 関数 [336 ページ]
db_change_nchar_charset 関数 [337 ページ]

1.10.11.3 Embedded SQL ホスト変数の使用方法

ホスト変数はさまざまな状況で使用できます。

ホスト変数は次の箇所で使用できます。

- SELECT、INSERT、UPDATE、DELETE 文で数値定数または文字列定数を使用できる場所。

- SELECT 文と FETCH 文の INTO 句。
- また、ホスト変数は、文の名前、カーソル名、Embedded SQL 固有の文のオプション名の代わりに使用できます。
- CONNECT、DISCONNECT、SET CONNECT 文では、ホスト変数はサーバ名、データベース名、接続名、ユーザ ID、パスワード、接続文字列の代わりに使用できます。
- SET OPTION と GET OPTION では、オプション値の代わりにホスト変数を使用できます。

ホスト変数は次の場合には使用できません。

- ホスト変数は、どの文でもテーブル名、カラム名としては使用できません。
- バッチではホスト変数を使用できません。
- SET 文のサブクエリ内ではホスト変数を使用できません。

SQLCODE および SQLSTATE ホスト変数

ISO/ANSI 標準によれば、Embedded SQL ソースファイルの Embedded SQL 宣言セクション内で次の特別なホスト変数を宣言できます。

```
long SQLCODE;
char SQLSTATE[6];
```

これらの変数を使用する場合、データベース要求を生成する任意の Embedded SQL 文 (DECLARE SECTION、INCLUDE、WHENEVER SQLCODE などを除く EXEC SQL 文) の後で、変数が設定されます。したがって、SQLCODE および SQLSTATE ホスト変数は、データベース要求を生成するすべての Embedded SQL 文のスコープで参照可能である必要があります。

次に示すのは、有効な Embedded SQL です。

```
EXEC SQL INCLUDE SQLCA;
// declare SQLCODE with global scope
EXEC SQL BEGIN DECLARE SECTION;
long SQLCODE;
EXEC SQL END DECLARE SECTION;
sub1() {
 EXEC SQL BEGIN DECLARE SECTION;
 char SQLSTATE[6];
 EXEC SQL END DECLARE SECTION;
 exec SQL CREATE TABLE ...
}
sub2() {
 EXEC SQL BEGIN DECLARE SECTION;
 char SQLSTATE[6];
 EXEC SQL END DECLARE SECTION;
 exec SQL DROP TABLE ...
}
```

次の例は、SQLSTATE が関数 sub2 のスコープ内で定義されていないため、有効な Embedded SQL ではありません。

```
EXEC SQL INCLUDE SQLCA;
sub1() {
 EXEC SQL BEGIN DECLARE SECTION;
 char SQLSTATE[6];
 EXEC SQL END DECLARE SECTION;
 exec SQL CREATE TABLE...
}
sub2() {
 exec SQL DROP TABLE...
```

```
}
```

Embedded SQL プリプロセッサの -k オプションを使用すると、Embedded SQL 宣言セクションのスコープの範囲外で SQLCODE 変数を宣言できます。

関連情報

Embedded SQL プリプロセッサ [255 ページ]

1.10.11.4 Embedded SQL インジケータ変数

インジケータ変数とは、データのやりとりをするときに補足的な情報を保持する C 変数のことです。

インジケータ変数の役割は、場合によってまったく異なります。

NULL 値

アプリケーションが NULL 値を扱えるようにします。

文字列のトランケーション

フェッチした値がホスト変数に収まるようにトランケートされた場合に、アプリケーションが対応できるようにします。

変換エラー

エラー情報を保持します。

インジケータ変数は a_sql_len 型のホスト変数で、SQL 文では通常のホスト変数の直後に書きます。たとえば、次の INSERT 文では、:ind_phone がインジケータ変数です。

```
EXEC SQL INSERT INTO Employees
 VALUES (:employee_number, :employee_name,
 :employee_initials, :employee_phone:ind_phone );
```

フェッチ時または実行時にデータベースサーバからローを受信しなかった場合 (エラーが発生したか、結果セットの末尾に到達した場合)、インジケータの値は変更されません。

> **注記**
>
> 32 ビット長および 64 ビット長とインジケータを今後使用できるように、Embedded SQL インジケータ変数での short int の使用は推奨されません。代わりに a_sql_len を使用してください。

このセクションの内容:

インジケータ変数: SQL NULL 値 [279 ページ]
SQL での NULL を同じ名前の C 言語の定数と混同しないでください。

インジケータ変数: トランケートされた値 [280 ページ]
インジケータ変数は、ホスト変数に収まるようにトランケートされたフェッチされた変数があるかどうかを示します。これによって、アプリケーションがトランケーションに適切に対応できるようになります。

インジケータ変数: 変換エラー [280 ページ]
デフォルトでは、conversion_error データベースオプションは On に設定され、データ型変換が失敗するとエラーになってローは返されません。

インジケータ変数値のまとめ [280 ページ]
インジケータ変数値は、取り出されたカラム値に関する情報を伝達するために使用されます。

1.10.11.4.1 インジケータ変数: SQL NULL 値

SQL での NULL を同じ名前の C 言語の定数と混同しないでください。

SQL 言語では、NULL は属性が不明であるか情報が適切でないかのいずれかを表します。C 言語の NULL は空のポインタ定数として参照され、メモリのロケーションを指さないポインタ値を表します。

このマニュアルで使用されている NULL は、上記の SQL データベースでの意味を指します。

NULL は、カラムに定義されるどのデータ型の値とも同じではありません。したがって、NULL 値をデータベースに渡したり、結果に NULL を受取ったりするためには、通常のホスト変数の他に何か特別なものが必要です。このために使用されるのが、インジケータ変数です。

NULL を挿入する場合のインジケータ変数

INSERT 文は、次のようにインジケータ変数を含むことができます。

```
EXEC SQL BEGIN DECLARE SECTION;
short int employee_number;
char employee_name[50];
char employee_initials[6];
char employee_phone[15];
a_sql_len ind_phone;
EXEC SQL END DECLARE SECTION;
/*
This program fills in the employee number,
name, initials, and phone number.
*/
if( /* Phone number is unknown */ ) {
 ind_phone = -1;
} else {
 ind_phone = 0;
}
EXEC SQL INSERT INTO Employees
 VALUES (:employee_number, :employee_name,
 :employee_initials, :employee_phone:ind_phone );
```

インジケータ変数の値が –1 の場合は、NULL が書き込まれます。値が 0 の場合は、employee_phone の実際の値が書き込まれます。

NULL をフェッチする場合のインジケータ変数

インジケータ変数は、データをデータベースから受け取るときにも使用されます。この場合は、NULL 値がフェッチされた (インジケータが負) ことを示すために使用されます。NULL 値がデータベースからフェッチされたときにインジケータ変数が渡されない場合は、エラーが発生します (SQLE_NO_INDICATOR)。

1.10.11.4.2 インジケータ変数: トランケートされた値

インジケータ変数は、ホスト変数に収まるようにトランケートされたフェッチされた変数があるかどうかを示します。これによって、アプリケーションがトランケーションに適切に対応できるようになります。

フェッチの際に値がトランケートされると、インジケータ変数は正の値になり、トランケーション前のデータベース値の実際の長さを示します。データベース値の実際の長さが 32767 バイトを超える場合は、インジケータ変数は 32767 になります。

1.10.11.4.3 インジケータ変数: 変換エラー

デフォルトでは、conversion_error データベースオプションは On に設定され、データ型変換が失敗するとエラーになってローは返されません。

この場合、インジケータ変数を使用して、どのカラムでデータ型変換が失敗したかを示すことができます。データベースオプション conversion_error を Off にすると、データ型変換が失敗した場合はエラーではなく CANNOT_CONVERT 警告を発します。変換エラーが発生したカラムにインジケータ変数がある場合、その変数の値は-2 になります。

conversion_error オプションを Off にすると、データをデータベースに挿入するときに変換が失敗した場合は NULL 値が挿入されます。

1.10.11.4.4 インジケータ変数値のまとめ

インジケータ変数値は、取り出されたカラム値に関する情報を伝達するために使用されます。

次の表は、インジケータ変数の使用法をまとめたものです。

インジケータの値	データベースに渡す値	データベースから受け取る値
> 0	ホスト変数値	取り出された値はトランケートされています。インジケータ変数は実際の長さを示します。
0	ホスト変数値	フェッチが成功、または conversion_error が On に設定されています。
-1	NULL 値	NULL 結果。
-2	NULL 値	変換エラー (conversion_error が Off に設定されている場合のみ)。SQLCODE は CANNOT_CONVERT 警告を示します。

インジケータの値	データベースに渡す値	データベースから受け取る値
< -2	NULL 値	NULL 結果。

1.10.12 SQLCA (SQL Communication Area)

SQLCA (**SQL Communication Area**) とは、データベースへの要求のたびに、アプリケーションとデータベースサーバの間で、統計情報とエラーをやりとりするのに使用されるメモリ領域です。

SQLCA は、アプリケーションとデータベース間の通信リンクのハンドルとして使用されます。データベースサーバとやりとりする必要のあるデータベースライブラリ関数には SQLCA が必ず渡されます。また、Embedded SQL 文でも必ず暗黙的に渡されます。

インタフェースライブラリ内には、グローバル SQLCA 変数が 1 つ定義されています。Embedded SQL プリプロセッサはこのグローバル SQLCA 変数の外部参照と、そのポインタの外部参照を生成します。外部参照の名前は sqlca、型は SQLCA です。ポインタの名前は sqlcaptr です。実際のグローバル変数は、インポートライブラリ内で宣言されています。

SQLCA は、`sqlca.h` ヘッダファイルで定義されています。このファイルは、ソフトウェアのインストールディレクトリの `SDK¥Include` サブディレクトリにあります。

SQLCA にはエラーコードが入る

SQLCA を参照すると、特定のエラーコードの検査ができます。データベースへの要求でエラーがあると、sqlcode フィールドと sqlstate フィールドにエラーコードが入ります。sqlcode や sqlstate などの SQLCA のフィールドを参照するために、C マクロが定義されています。

このセクションの内容:

SQLCA のフィールド [281 ページ]
SQLCA は、複数のフィールドを持つ構造体です。

マルチスレッドまたは再入可能コードでの SQLCA 管理 [284 ページ]
Embedded SQL 文はマルチスレッドまたは再入可能コードでも使用できます。

複数の SQLCA [285 ページ]
再入力不可コードを生成する Embedded SQL プリプロセッサオプション (-r-) を使用しないでください。再入可能コードは、静的に初期化されたグローバル変数を使用できないため、若干サイズが大きく、遅いコードになります。ただし、その影響は最小限です。

1.10.12.1 SQLCA のフィールド

SQLCA は、複数のフィールドを持つ構造体です。

これらのフィールドには、次のような意味があります。

sqlcaid

SQLCA 構造体の ID として文字列 SQLCA が格納される 8 バイトの文字フィールド。このフィールドはデバッグ時にメモリの中身を見るときに役立ちます。

sqlcabc

SQLCA 構造体の長さ (136 バイト) が入る 32 ビットの整数。

sqlcode

データベースが検出した要求エラーのエラーコードが入る 32 ビットの整数。エラーコードの定義はヘッダファイル `sqlerr.h` にあります。エラーコードは、0 (ゼロ) は成功、正は警告、負はエラーを示します。

sqlerrml

sqlerrmc フィールドの情報の長さ。

sqlerrmc

エラーメッセージに挿入される文字列。挿入されない場合もあります。エラーメッセージに 1 つまたは複数のプレースホルダ文字列 (`%1`、`%2`、...) があると、このフィールドの文字列と置換されます。

たとえば、「`Table Not Found`」(テーブルが見つかりません) というエラーが発生した場合、sqlerrmc にはテーブル名が入り、これがエラーメッセージの適切な場所に挿入されます。

sqlerrp

予約済み。

sqlerrd

32 ビット整数の汎用配列。

sqlwarn

予約済み。

sqlstate

SQLSTATE ステータス値。ANSI SQL 標準では、SQLCODE 値のほかに、SQL 文からのこの型の戻り値が定義されます。SQLSTATE 値は NULL で終了する長さ 5 の文字列で、前半 2 バイトがクラス、後半 3 バイトがサブクラスを表します。各バイトは 0 ～ 9 の数字、または、A ～ Z の英大文字です。

0 ～ 4 または A ～ H の文字で始まるクラス、サブクラスはすべて SQL 標準で定義されています。それ以外のクラスとサブクラスは実装依存です。SQLSTATE 値 '00000' はエラーや警告がなかったことを意味します。

sqlerror 配列

sqlerror フィールドの配列要素を次に示します。

sqlerrd[1] (SQLIOCOUNT)

文を完了するために必要とされた入出力操作の実際の回数。

データベースサーバによって、文の実行ごとにこの値が 0 にリセットされることはありません。一連の文を実行する前にこの変数が 0 にリセットされるようにプログラムすることもできます。最後の文が実行された後、この値は一連の文の入出力操作の合計回数になります。

sqlerrd[2] (SQLCOUNT)

このフィールドの値の意味は実行中の文によって変わります。

INSERT、UPDATE、PUT、DELETE 文

文によって影響を受けたローの数。

OPEN 文と RESUME 文

カーソルを開いたとき、または再開したとき、このフィールドには、カーソル内の実際のロー数 (0 以上の値)、または、その推定値 (負の数で、その絶対値が推定値) が入ります。データベースサーバが、ローをカウントしなくてもこの値を計算できる場合、これは実際のローの数になります。row_counts オプションを使って、常にローの実際の数を返すようにデータベースを設定することもできます。

FETCH カーソル文

SQLCOUNT フィールドは、警告 SQLE_NOTFOUND が返った場合に設定されます。このフィールドには、FETCH RELATIVE または FETCH ABSOLUTE 文によってカーソル位置の可能な範囲を超えたローの数が入ります (カーソルは、ローの上にも、最初のローより前または最後のローより後にも置くことができます)。ワイドフェッチの場合、SQLCOUNT は実際にフェッチされたローの数であり、要求されたローの数と同じかそれより少なくなります。ワイドフェッチ中に SQLE_NOTFOUND が設定されるのは、ローがまったく返されなかった場合のみです。

ローが見つからなくても位置が有効な場合は、値は 0 です。たとえば、カーソル位置が最後のローのときに FETCH RELATIVE 1 を実行した場合です。カーソルの最後を超えてフェッチしようとした場合、値は正の数です。カーソルの先頭を超えてフェッチしようとした場合、値は負の数です。

GET DATA 文

SQLCOUNT フィールドには値の実際の長さが入っています。

DESCRIBE 文

WITH VARIABLE RESULT 句を使用して複数の結果セットを返す可能性のあるプロシージャを記述した場合、SQLCOUNT は次のいずれかの値に設定されます。

0

結果セットは変更される場合があります。各 OPEN 文の後でプロシージャコールを記述し直してください。

1

結果セットは固定です。再度記述する必要はありません。

SQLE_SYNTAX_ERROR 構文エラーの場合、このフィールドには文内のおおよそのエラー検出位置が入ります。

sqlerrd[3] (SQLIOESTIMATE)

文の完了に必要な入出力操作の推定回数。このフィールドは OPEN 文または EXPLAIN 文によって値が設定されます。

関連情報

Embedded SQL を使用したワイドフェッチ [302 ページ]

1.10.12.2 マルチスレッドまたは再入可能コードでの SQLCA 管理

Embedded SQL 文はマルチスレッドまたは再入可能コードでも使用できます。

ただし、単一接続の場合は、アクティブな要求は 1 接続あたり 1 つに制限されます。マルチスレッドアプリケーションにおいて、セマフォを使ったアクセス制御をしない場合は、1 つのデータベース接続を各スレッドで共用しないでください。

データベースを使用する各スレッドが別々の接続を使用する場合は制限がまったくありません。ランタイムライブラリは SQLCA を使用してスレッドのコンテキストを区別します。したがって、データベースを同時に使用するスレッドには、それぞれ専用の SQLCA が必要です。ただし、例外として、スレッドで db_cancel_request 関数を使用すると、そのスレッドの SQLCA を使用している別のスレッドで実行されている文をキャンセルできます。

次は再入可能マルチスレッド Embedded SQL コードの例です。

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <ctype.h>
#include <stdlib.h>
#include <process.h>
#include <windows.h>
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;
#define TRUE 1
#define FALSE 0
// multithreading support
typedef struct a_thread_data {
    SQLCA sqlca;
    int num_iters;
    int thread;
    int done;
} a_thread_data;
// each thread's ESQL test
EXEC SQL SET SQLCA "&thread_data->sqlca";
static void PrintSQLError( a_thread_data * thread_data )
/*****************************************************/
{
  char                buffer[200];
  printf( "%d: SQL error %d -- %s ... aborting¥n",
      thread_data->thread,
      SQLCODE,
      sqlerror_message( &thread_data->sqlca,
              buffer, sizeof( buffer ) ) );
    exit( 1 );
}
EXEC SQL WHENEVER SQLERROR { PrintSQLError( thread_data ); };
static void do_one_iter( void * data )
{
  a_thread_data *  thread_data = (a_thread_data *)data;
  int    i;
  EXEC SQL BEGIN DECLARE SECTION;
  char    user[ 20 ];
  EXEC SQL END DECLARE SECTION;
  if( db_init( &thread_data->sqlca ) != 0 ) {
    for( i = 0; i < thread_data->num_iters; i++ ) {
     EXEC SQL CONNECT "dba" IDENTIFIED BY "sql";
     EXEC SQL SELECT USER INTO :user;
     EXEC SQL DISCONNECT;
    }
    printf( "Thread %d did %d iters successfully¥n",
        thread_data->thread, thread_data->num_iters );
    db_fini( &thread_data->sqlca );
  }
  thread_data->done = TRUE;
```

```
}
int main()
{
  int num_threads = 4;
  int thread;
  int num_iters = 300;
  int num_done = 0;
  a_thread_data *thread_data;
  thread_data = (a_thread_data *)malloc( sizeof(a_thread_data) * num_threads );
  for( thread = 0; thread < num_threads; thread++ ) {
    thread_data[ thread ].num_iters = num_iters;
    thread_data[ thread ].thread = thread;
    thread_data[ thread ].done = FALSE;
    if( _beginthread( do_one_iter,
      8096,
      (void *)&thread_data[thread] ) <= 0 ) {
      printf( "FAILED creating thread.¥n" );
      return( 1 );
    }
  }
    while( num_done != num_threads ) {
    Sleep( 1000 );
    num_done = 0;
    for( thread = 0; thread < num_threads; thread++ ) {
      if( thread_data[ thread ].done == TRUE ) {
        num_done++;
      }
    }
  }
  return( 0 );
}
```

関連情報

Embedded SQL を使用した要求管理 [325 ページ]

1.10.12.3 複数の SQLCA

再入力不可コードを生成する Embedded SQL プリプロセッサオプション (-r-) を使用しないでください。再入可能コードは、静的に初期化されたグローバル変数を使用できないため、若干サイズが大きく、遅いコードになります。ただし、その影響は最小限です。

プログラムで使用する各 SQLCA は db_init を呼び出して初期化し、最後に db_fini を呼び出してクリーンアップします。

Embedded SQL 文の SET SQLCA を使用して、データベース要求で別の SQLCA を使用することを Embedded SQL プリプロセッサに伝えます。通常は、`EXEC SQL SET SQLCA 'task_data->sqlca';` のような文をプログラムの先頭またはヘッダファイルに置いて、SQLCA 参照がタスク固有のデータを指すようにします。この文はコードを生成しないため、パフォーマンスに影響はありません。この文はプリプロセッサ内部の状態を変更して、指定の文字列で SQLCA を参照するようにします。

各スレッドには専用の SQLCA が必要です。この要件は、Embedded SQL を使用していて、アプリケーションの複数のスレッドから呼び出される、共有ライブラリ (DLL など) 内のコードにも適用されます。

複数の SQLCA のサポートは、サポートされる任意の Embedded SQL 環境で使用できますが、再入可能コードでは必須です。

複数のデータベースに接続するために複数の SQLCA を使用したり、単一のデータベースに対して複数の接続を持つ必要はありません。

各 SQLCA は、無名の接続を 1 つ持つことができます。各 SQLCA はアクティブな接続、つまり現在の接続を持ちます。

特定のデータベース接続に対するすべての操作では、その接続が確立されたときに使用されたのと同じ SQLCA を使用します。

> **注記**
>
> 異なる接続に対する操作には通常のレコードロックメカニズムが適用されるため、各操作が互いにブロックしてデッドロックを発生させる可能性があります。

1.10.13 静的 SQL と動的 SQL

SQL 文を C プログラムに埋め込むには、静的 SQL と動的 SQL の 2 つの方法があります。

このセクションの内容:

静的 SQL 文 [286 ページ]
: 標準的 SQL のデータ操作文とデータ定義文はすべて、文の前に EXEC SQL を付け、後ろにセミコロン (;) を付けることで、C プログラムに埋め込むことができます。このような文を静的文と呼びます。

動的 SQL 文 [287 ページ]
: C 言語では、文字列は文字の配列に格納されます。動的 SQL 文は C 言語の文字列で構築されます。これらの文は PREPARE 文と EXECUTE 文を使用して実行できます。

動的 SELECT 文 [288 ページ]
: 単一のローだけを返す SELECT 文は、動的に準備し、その後に EXECUTE 文に INTO 句を指定してローを 1 つだけ取り出すようにできます。ただし、複数ローを返す SELECT 文では動的カーソルを使用します。

1.10.13.1 静的 SQL 文

標準的 SQL のデータ操作文とデータ定義文はすべて、文の前に EXEC SQL を付け、後ろにセミコロン (;) を付けることで、C プログラムに埋め込むことができます。このような文を静的文と呼びます。

静的文にはホスト変数への参照を含めることができます。ホスト変数は文字列定数または数値定数の代わりにしか使えません。カラム名やテーブル名としては使用できません。このような操作には動的文が必要です。

関連情報

Embedded SQL のホスト変数 [272 ページ]

1.10.13.2 動的 SQL 文

C 言語では、文字列は文字の配列に格納されます。動的 SQL 文は C 言語の文字列で構築されます。これらの文は PREPARE 文と EXECUTE 文を使用して実行できます。

これらの SQL 文は静的文と同じ方法でホスト変数を参照することはできません。C 言語の変数は、C プログラムの実行中に変数名でアクセスできないためです。

SQL 文と C 言語の変数との間で情報をやりとりするために、**SQLDA** (**SQL Descriptor Area**) という構造体が使用されます。EXECUTE 文で USING 句を使ってホスト変数のリストを指定すると、この構造体が Embedded SQL プリプロセッサによって自動的に用意されます。ホスト変数のリストは、準備文の適切な位置にあるプレースホルダに順番に対応しています。

プレースホルダは文の中に置いて、どこでホスト変数にアクセスするかを指定します。プレースホルダは、疑問符 (?) か静的文と同じホスト変数参照です (ホスト変数名の前にはコロンを付けます)。ホスト変数参照の場合も、実際の文テキスト内のホスト変数名は SQLDA を参照することを示すプレースホルダの役割しかありません。

データベースに情報を渡すのに使用するホスト変数をバインド変数と呼びます。

例

```
EXEC SQL BEGIN DECLARE SECTION;
char      comm[200];
char      street[30];
char      city[20];
a_sql_len cityind;
long      empnum;
EXEC SQL END DECLARE SECTION;
...
sprintf( comm,
    "UPDATE %s SET Street = :?, City = :?"
    "WHERE EmployeeID = :?",
    tablename );
EXEC SQL PREPARE S1 FROM :comm FOR UPDATE;
EXEC SQL EXECUTE S1 USING :street, :city:cityind, :empnum;
```

この方法では、文中にいくつのホスト変数があるかを知っている必要があります。通常はそのようなことはありません。そこで、自分で SQLDA 構造体を設定し、この SQLDA を EXECUTE 文の USING 句で指定します。

DESCRIBE BIND VARIABLES 文は、準備文内にあるバインド変数のホスト変数名を返します。これにより、C プログラムでホスト変数を管理するのが容易になります。一般的な方法を次に示します。

```
EXEC SQL BEGIN DECLARE SECTION;
char comm[200];
EXEC SQL END DECLARE SECTION;
...
sprintf( comm,
    "UPDATE %s SET Street = :street, City = :city"
   " WHERE EmployeeID = :empnum",
   tablename );
EXEC SQL PREPARE S1 FROM :comm FOR UPDATE;
/* Assume that there are no more than 10 host variables.
 * See next example if you cannot put a limit on it. */
sqlda = alloc_sqlda( 10 );
EXEC SQL DESCRIBE BIND VARIABLES FOR S1 INTO sqlda;
/* sqlda->sqld will tell you how many
   host variables there were. */
/* Fill in SQLDA_VARIABLE fields with
   values based on name fields in sqlda. */
...
EXEC SQL EXECUTE S1 USING DESCRIPTOR sqlda;
```

```
free_sqlda( sqlda );
```

SQLDA の内容

SQLDA は変数記述子の配列です。各記述子は、対応する C プログラム変数の属性、または、データベースがデータを入出力するロケーションを記述します。

- データ型
- `type` が文字列型の場合は長さ
- メモリアドレス
- インジケータ変数

インジケータ変数と NULL

インジケータ変数はデータベースに NULL 値を渡したり、データベースから NULL 値を取り出すのに使用されます。インジケータ変数は、データベース操作中にトランケーション条件が発生したことをデータベースサーバが示すのにも使用されます。インジケータ変数はデータベースの値を受け取るのに十分な領域がない場合、正の値に設定されます。

関連情報

SQLDA (SQL descriptor area) [289 ページ]
Embedded SQL インジケータ変数 [278 ページ]

1.10.13.3 動的 SELECT 文

単一のローだけを返す SELECT 文は、動的に準備し、その後に EXECUTE 文に INTO 句を指定してローを 1 つだけ取り出すようにできます。ただし、複数ローを返す SELECT 文では動的カーソルを使用します。

動的カーソルでは、結果はホスト変数のリスト、または FETCH 文 (FETCH INTO と FETCH USING DESCRIPTOR) で指定する SQLDA に入ります。通常、SELECT リスト項目の数は不明であるため、多くの場合、SQLDA を使用します。DESCRIBE SELECT LIST 文で SQLDA に SELECT リスト項目の型を設定します。その後、fill_sqlda 関数または fill_s_sqlda 関数を使用して、値のための領域を割り付けます。情報は FETCH USING DESCRIPTOR 文で取り出します。

次は典型的な例です。

```
EXEC SQL BEGIN DECLARE SECTION;
char comm[200];
EXEC SQL END DECLARE SECTION;
int actual_size;
SQLDA * sqlda;
...
```

```
sprintf( comm, "SELECT * FROM %s", table_name );
EXEC SQL PREPARE S1 FROM :comm;
/* Initial guess of 10 columns in result.
   If it is wrong, it is corrected right
   after the first DESCRIBE by reallocating
   sqlda and doing DESCRIBE again. */
sqlda = alloc_sqlda( 10 );
EXEC SQL DESCRIBE SELECT LIST FOR S1
  INTO sqlda;
if( sqlda->sqld > sqlda->sqln )
{
  actual_size = sqlda->sqld;
  free_sqlda( sqlda );
  sqlda = alloc_sqlda( actual_size );
  EXEC SQL DESCRIBE SELECT LIST FOR S1
    INTO sqlda;
}
fill_sqlda( sqlda );
EXEC SQL DECLARE C1 CURSOR FOR S1;
EXEC SQL OPEN C1;
EXEC SQL WHENEVER NOTFOUND {break};
for( ;; )
{
  EXEC SQL FETCH C1 USING DESCRIPTOR sqlda;
  /* do something with data */
}
EXEC SQL CLOSE C1;
EXEC SQL DROP STATEMENT S1;
```

注記

リソースを無駄に消費しないように、文は使用後に削除してください。

動的カーソルのサンプルは、動的 SELECT 文でのカーソルの使用方法を示しています。

関連情報

動的カーソルのサンプル [266 ページ]
ライブラリ関数のリファレンス [326 ページ]

1.10.14 SQLDA (SQL descriptor area)

SQLDA (SQL Descriptor Area) は動的 SQL 文で使用されるインタフェース構造体です。この構造体を使用して、ホスト変数と SELECT 文の結果に関する情報を、データベースとの間で受け渡しします。SQLDA はヘッダファイル `sqlda.h` に定義されています。

データベースのインタフェース共有ライブラリまたは DLL には、SQLDA の管理に使用できる関数が用意されています。

ホスト変数を静的 SQL 文で使用するときは、プリプロセッサがホスト変数用の SQLDA を構成します。実際にデータベースサーバとの間でやりとりされるのは、この SQLDA です。

このセクションの内容:

SQLDA ヘッダファイル [290 ページ]
SQLDA (SQL Descriptor Area) データ構造は、sqlda.h ヘッダファイルによって記述されます。

SQLDA のフィールド [291 ページ]
SQLDA (SQL Descriptor Area) は、多数のフィールドで構成されたデータ構造です。

SQLDA のホスト変数の記述 [292 ページ]
SQLDA の sqlvar 構造体がそれぞれ 1 つのホスト変数を記述しています。

DESCRIBE 実行後の SQLDA の sqllen フィールドの値 [293 ページ]
DESCRIBE 文は、データベースから取り出したデータを格納するために必要なホスト変数、またはデータベースにデータを渡すために必要なホスト変数に関する情報を取得します。

データを送信するときの SQLDA の sqllen フィールドの値 [295 ページ]
データベースに送信されるデータの長さの決定方法は、データ型によって異なります。

データを取得するときの SQLDA の sqllen フィールドの値 [297 ページ]
データベースから取得されるデータの長さの決定方法は、データ型によって異なります。

関連情報

ライブラリ関数のリファレンス [326 ページ]

1.10.14.1 SQLDA ヘッダファイル

SQLDA (SQL Descriptor Area) データ構造は、sqlda.h ヘッダファイルによって記述されます。

sqlda.h の内容を、次に示します。

```
#ifndef _SQLDA_H_INCLUDED
#define _SQLDA_H_INCLUDED
#define II_SQLDA
#include "sqlca.h"
#if defined( _SQL_PACK_STRUCTURES )
  #if defined( _MSC_VER ) && _MSC_VER > 800
    #pragma warning(push)
    #pragma warning(disable:4103)
  #endif
  #include "pshpk1.h"
#endif
#define SQL_MAX_NAME_LEN    30
#define _sqldafar
typedef short int a_sql_type;
struct sqlname {
  short int   length; /* length of char data */
  char        data[ SQL_MAX_NAME_LEN ]; /* data */
};
struct sqlvar {          /* array of variable descriptors */
  short int   sqltype;   /* type of host variable */
  a_sql_len   sqllen;    /* length of host variable */
  void        *sqldata;  /* address of variable */
  a_sql_len   *sqlind;   /* indicator variable pointer */
  struct sqlname sqlname;
};
```

```
#if defined( _SQL_PACK_STRUCTURES )
  #include "poppk.h"
  /* The SQLDA should be 4-byte aligned */
  #include "pshpk4.h"
#endif
struct sqlda {
  unsigned char   sqldaid[8]; /* eye catcher "SQLDA" */
  a_sql_int32     sqldabc; /* length of sqlda structure */
  short int       sqln;    /* descriptor size in number of entries */
  short int       sqld;    /* number of variables found by DESCRIBE */
  struct sqlvar   sqlvar[1]; /* array of variable descriptors */
};
#define SCALE(sqllen)       ((sqllen)/256)
#define PRECISION(sqllen)   ((sqllen)&0xff)
#define SET_PRECISION_SCALE(sqllen,precision,scale)   ¥
               sqllen = (scale)*256 + (precision)
#define DECIMALSTORAGE(sqllen)  (PRECISION(sqllen)/2 + 1)
typedef struct sqlda    SQLDA;
typedef struct sqlvar   SQLVAR, SQLDA_VARIABLE;
typedef struct sqlname  SQLNAME, SQLDA_NAME;
#ifndef SQLDASIZE
#define SQLDASIZE(n)  ( sizeof( struct sqlda ) +  ¥
             (n-1) * sizeof( struct sqlvar) )
#endif
#if defined( _SQL_PACK_STRUCTURES )
  #include "poppk.h"
  #if defined( _MSC_VER ) && _MSC_VER > 800
    #pragma warning(pop)
  #endif
#endif
#endif
```

1.10.14.2 SQLDA のフィールド

SQLDA (SQL Descriptor Area) は、多数のフィールドで構成されたデータ構造です。

SQLDA のフィールドの意味を次に示します。

フィールド	説明
sqldaid	SQLDA 構造体の ID として文字列 SQLDA が格納される 8 バイトの文字フィールド。このフィールドはデバッグ時にメモリの中身を見るときに役立ちます。
sqldabc	SQLDA 構造体の長さが入る 32 ビットの整数。
sqln	sqlvar 配列に割り付けられた変数記述子の数。
sqld	有効な変数記述子の数 (ホスト変数の記述情報を含む)。このフィールドは DESCRIBE 文で設定されます。また、このフィールドは、データベースサーバにデータを渡すときに設定できます。
sqlvar	struct sqlvar 型の記述子の配列。各要素がホスト変数を記述します。

1.10.14.3 SQLDA のホスト変数の記述

SQLDA の sqlvar 構造体がそれぞれ 1 つのホスト変数を記述しています。

sqlvar 構造体のフィールドの意味を次に示します。

sqltype

この記述子で記述している変数の型。

低位ビットは NULL 値が可能かどうかを示します。有効な型と定数の定義は `sqldef.h` ヘッダファイルにあります。

このフィールドは DESCRIBE 文で設定されます。データベースサーバにデータを渡したり、データベースサーバからデータを取り出したりするときに、このフィールドはどの型にでも設定できます。必要な型変換は自動的に行われます。

sqllen

変数の長さ。sqllen 値は a_sql_len 型です。長さが実際に何を意味するかは、型情報と SQLDA の使用法によって決まります。

データ型 LONG VARCHAR、LONG NVARCHAR、LONG BINARY の場合は、sqllen フィールドの代わりに、データ型の構造体 DT_LONGVARCHAR、DT_LONGNVARCHAR、または DT_LONGBINARY の array_len フィールドが使用されます。

sqldata

この変数が占有するメモリへのポインタ。このメモリ領域は sqltype と sqllen フィールドに合致させてください。

UPDATE 文、INSERT 文では、sqldata ポインタが NULL ポインタの場合、この変数は操作に関係しません。FETCH では、sqldata ポインタが NULL ポインタの場合、データは返されません。つまり、sqldata ポインタが返すカラムは、バインドされていないカラムです。

DESCRIBE 文が LONG NAMES を使用している場合、このフィールドは結果セットカラムのロングネームを保持します。さらに、DESCRIBE 文が DESCRIBE USER TYPES 文の場合は、このフィールドはカラムではなくユーザ定義のデータ型のロングネームを保持します。型がベースタイプの場合、フィールドは空です。

sqlind

インジケータ値へのポインタ。インジケータ値は a_sql_len 型です。負のインジケータ値は NULL 値を意味します。正のインジケータ値は、この変数が FETCH 文でトランケートされたことを示し、インジケータ値にはトランケートされる前のデータの長さが入ります。conversion_error データベースオプションを Off に設定した場合、–2 の値は変換エラーを示します。

sqlind ポインタが NULL ポインタの場合、このホスト変数に対応するインジケータ変数はありません。

sqlind フィールドは、DESCRIBE 文でパラメータタイプを示すのにも使用されます。ユーザ定義のデータ型の場合、このフィールドは DT_HAS_USERTYPE_INFO に設定されます。この場合、DESCRIBE USER TYPES を実行すると、ユーザ定義のデータ型についての情報を取得できます。

sqlname

次のような VARCHAR に似た構造体。

```
struct sqlname {
   short int  length;
   char  data[ SQL_MAX_NAME_LEN ];
};
```

DESCRIBE 文によって設定され、それ以外では使用されません。このフィールドは DESCRIBE 文の 2 つのフォーマットによって意味が異なります。

SELECT LIST

名前データバッファには SELECT リストの対応する項目のカラム見出しが入ります。

BIND VARIABLES

名前データバッファにはバインド変数として使用されたホスト変数名が入ります。無名のパラメータマーカが使用されている場合は、"?" が入ります。

DESCRIBE SELECT LIST 文では、指定のインジケータ変数にはすべて、SELECT リスト項目が更新可能かどうかを示すフラグが設定されます。このフラグの詳細は、`sqldef.h` ヘッダファイルにあります。

DESCRIBE 文が DESCRIBE USER TYPES 文の場合、このフィールドはカラムではなくユーザ定義のデータ型のロングネームを保持します。型がベースタイプの場合、フィールドは空です。

関連情報

Embedded SQL インジケータ変数 [278 ページ]
Embedded SQL のデータ型 [268 ページ]

1.10.14.4 DESCRIBE 実行後の SQLDA の sqllen フィールドの値

DESCRIBE 文は、データベースから取り出したデータを格納するために必要なホスト変数、またはデータベースにデータを渡すために必要なホスト変数に関する情報を取得します。

次の表は、データベースのさまざまな型に対して DESCRIBE 文 (SELECT LIST 文と BIND VARIABLE 文の両方) が返す sqllen と sqltype 構造体のメンバーの値を示します。ユーザ定義のデータベースデータ型の場合、ベースタイプが記述されます。

プログラムでは DESCRIBE の返す型と長さを使用できます。別の型も使用できます。データベースサーバはどの型でも型変換を行います。sqldata フィールドの指すメモリは sqltype と sqllen フィールドに合致させてください。Embedded SQL の型は、sqltype でビット処理 AND と DT_TYPES を指定して (sqltype & DT_TYPES) 取得します。

データベースのフィールドの型	返される Embedded SQL の型	describe で返される長さ (バイト単位)
BIGINT	DT_BIGINT	8
BINARY(n)	DT_BINARY	n
BIT	DT_BIT	1
CHAR(n)	DT_FIXCHAR[1]	データベースの文字セットからクライアントの CHAR 文字セットへの変換時の最大データ拡張に n を掛けた値。この長さが 32767 バイトを超える場合、返される Embedded SQL の型は、32767 バイト長の DT_LONGVARCHAR になります。

データベースのフィールドの型	返される Embedded SQL の型	describe で返される長さ (バイト単位)
CHAR(n CHAR)	DT_FIXCHAR[1]	クライアントの CHAR 文字セット内の文字の最大長に n を掛けた値。この長さが 32767 バイトを超える場合、返される Embedded SQL の型は、32767 バイト長の DT_LONGVARCHAR になります。
DATE	DT_DATE	フォーマットされた文字列の最大長
DECIMAL(p,s)	DT_DECIMAL	SQLDA の長さフィールドの低位バイトが p に、高位バイトが s に設定されます。sqlda.h の PRECISION および SCALE マクロを参照してください。
DOUBLE	DT_DOUBLE	8
FLOAT	DT_FLOAT	4
INT	DT_INT	4
LONG BINARY	DT_LONGBINARY	32767
LONG NVARCHAR	DT_LONGVARCHAR / DT_LONGNVARCHAR[2]	32767
LONG VARCHAR	DT_LONGVARCHAR	32767
NCHAR(n)	DT_FIXCHAR / DT_NFIXCHAR[2]	クライアントの NCHAR 文字セット内の文字の最大長に n を掛けた値。この長さが 32767 バイトを超える場合、返される Embedded SQL の型は、32767 バイト長の DT_LONGNVARCHAR になります。
NVARCHAR(n)	DT_VARCHAR / DT_NVARCHAR[2]	クライアントの NCHAR 文字セット内の文字の最大長に n を掛けた値。この長さが 32767 バイトを超える場合、返される Embedded SQL の型は、32767 バイト長の DT_LONGNVARCHAR になります。
REAL	DT_FLOAT	4
SMALLINT	DT_SMALLINT	2
TIME	DT_TIME	フォーマットされた文字列の最大長
TIMESTAMP	DT_TIMESTAMP	フォーマットされた文字列の最大長
TINYINT	DT_TINYINT	1
UNSIGNED BIGINT	DT_UNSBIGINT	8
UNSIGNED INT	DT_UNSINT	4
UNSIGNED SMALLINT	DT_UNSSMALLINT	2
VARCHAR(n)	DT_VARCHAR[1]	データベースの文字セットからクライアントの CHAR 文字セットへの変換時の最大データ拡張に n を掛けた値。この長さが 32767 バイトを超える場合、返される Embedded SQL の型は、32767 バイト長の DT_LONGVARCHAR になります。

データベースのフィールドの型	返される Embedded SQL の型	describe で返される長さ (バイト単位)
VARCHAR(n CHAR)	DT_VARCHAR[1]	クライアントの CHAR 文字セット内の文字の最大長に n を掛けた値。この長さが 32767 バイトを超える場合、返される Embedded SQL の型は、32767 バイト長の DT_LONGVARCHAR になります。

[1] クライアントの CHAR 文字セットの最大バイト長が 32767 バイトを超える場合、CHAR および VARCHAR について返される型は DT_LONGVARCHAR になります。

[2] クライアントの NCHAR 文字セットの最大バイト長が 32767 バイトを超える場合、NCHAR および NVARCHAR について返される型は DT_LONGNVARCHAR になります。NCHAR、NVARCHAR、LONG NVARCHAR はそれぞれデフォルトで DT_FIXCHAR、DT_VARCHAR、DT_LONGVARCHAR と記述されます。db_change_nchar_charset 関数が呼び出された場合、これらの型はそれぞれ DT_NFIXCHAR、DT_NVARCHAR、DT_LONGNVARCHAR と記述されます。

関連情報

Embedded SQL のデータ型 [268 ページ]
db_change_nchar_charset 関数 [337 ページ]

1.10.14.5 データを送信するときの SQLDA の sqllen フィールドの値

データベースに送信されるデータの長さの決定方法は、データ型によって異なります。

LONG 以外のデータ型では、SQLDA の `sqlvar` 構造体の `sqllen` フィールドは、データバッファの最大サイズを表します。たとえば、VARCHAR(300) として記述されるカラムには、そのカラムの最大長を表す 300 の `sqllen` 値が含まれます。DT_FIXCHAR などのブランクを埋め込まれたデータ型では、`sqllen` フィールドはデータバッファの最大サイズを表します。整数や DT_TIMESTAMP_ STRUCT などの固定サイズのデータ型では、`sqllen` フィールドは無視され、長さを指定する必要はありません。LONG のデータ型では、`array_len` フィールドによってデータバッファの最大長が指定されます。データを送信または取得するときには、`sqllen` フィールドは変更されません。

以下のテーブルで示されているデータ型のみが許可されます。DT_DATE、DT_TIME、DT_TIMESTAMP 型は、情報を送信または取得するときは DT_STRING と同じものとして扱われます。値は現在の日付形式に従って文字列としてフォーマットされます。

Embedded SQL のデータ型	長さを設定するプログラム動作
DT_BIGINT	固定サイズ。アクションは不要です。
DT_BINARY(n)	BINARY 構造体の `array` フィールドのデータのバイト単位の長さに `len` を設定します。最大長は 32767 です。
DT_BIT	固定サイズ。アクションは不要です。
DT_DATE	末尾の NULL 文字によって長さが決まります。

Embedded SQL のデータ型	長さを設定するプログラム動作
DT_DOUBLE	固定サイズ。アクションは不要です。
DT_FIXCHAR(n)	sqllen フィールドは、sqldata 領域のバイト単位の長さを決定します。データはブランクが埋め込まれる必要があります。最大長は 32767 です。
DT_FLOAT	固定サイズ。アクションは不要です。
DT_INT	固定サイズ。アクションは不要です。
DT_LONGBINARY	sqllen フィールドは無視されます。array_len フィールドによって array フィールドの最大長が指定されます。array フィールドのデータのバイト単位の長さに stored_len を設定します。32767 を超えるバイトを送信できます。
DT_LONGNVARCHAR	sqllen フィールドは無視されます。array_len フィールドによって array フィールドの最大長が指定されます。array フィールドのデータのバイト単位の長さに stored_len を設定します。32767 を超えるバイトを送信できます。
DT_LONGVARCHAR	sqllen フィールドは無視されます。array_len フィールドによって array フィールドの最大長が指定されます。array フィールドのデータのバイト単位の長さに stored_len を設定します。32767 を超えるバイトを送信できます。
DT_NFIXCHAR(n)	sqllen フィールドは、sqldata 領域のバイト単位の長さを決定します。データはブランクが埋め込まれる必要があります。最大長は 32767 です。
DT_NSTRING	末尾の NULL 文字によって長さが決まります。
DT_NVARCHAR(n)	NVARCHAR 構造体の array フィールドのデータのバイト単位の長さに len を設定します。
DT_SMALLINT	固定サイズ。アクションは不要です。
DT_STRING	末尾の NULL 文字によって長さが決まります。
DT_TIME	末尾の NULL 文字によって長さが決まります。
DT_TIMESTAMP	末尾の NULL 文字によって長さが決まります。
DT_TIMESTAMP_STRUCT	固定サイズ。アクションは不要です。
DT_UNSBIGINT	固定サイズ。アクションは不要です。
DT_UNSINT	固定サイズ。アクションは不要です。
DT_UNSSMALLINT	固定サイズ。アクションは不要です。
DT_VARCHAR(n)	VARCHAR 構造体の array フィールドのデータのバイト単位の長さに len を設定します。最大長は 32767 です。
DT_VARIABLE	末尾の NULL 文字によって長さが決まります。

関連情報

動的 SQL を使用した LONG データの送信 [322 ページ]

1.10.14.6 データを取得するときの SQLDA の sqllen フィールドの値

データベースから取得されるデータの長さの決定方法は、データ型によって異なります。

LONG 以外のデータ型では、SQLDA の `sqlvar` 構造体の `sqllen` フィールドは、データバッファの最大サイズを表します。たとえば、VARCHAR(300) として記述されるカラムには、そのカラムの最大長を表す 300 の `sqllen` 値が含まれます。DT_FIXCHAR などのブランクを埋め込まれたデータ型では、`sqllen` フィールドはデータバッファの最大サイズを表します。整数や DT_TIMESTAMP_ STRUCT などの固定サイズのデータ型では、`sqllen` フィールドは無視され、長さを指定する必要はありません。LONG のデータ型では、`array_len` フィールドによってデータバッファの最大長が指定されます。データを送信または取得するときには、`sqllen` フィールドは変更されません。

以下のテーブルで示されているデータ型のみが許可されます。DT_DATE、DT_TIME、DT_TIMESTAMP 型は、情報を送信または取得するときは DT_STRING と同じものとして扱われます。値は現在の日付形式に従って文字列としてフォーマットされます。

Embedded SQL のデータ型	値のフェッチ前にデータ領域の長さを指定する方法	値のフェッチ前に値の長さを決定する方法
DT_BIGINT	固定サイズ。アクションは不要です。	固定サイズ。アクションは不要です。
DT_BINARY(n)	`sqllen` フィールドは、BINARY 構造体の `array` フィールドのバイト単位の最大長に、`len` フィールドのサイズ (n+2) を加えた長さに設定されます。最大長は 32767 です。	BINARY 構造体の `len` フィールドは、BINARY 構造体の `array` フィールドのデータの実際のバイト単位の長さに更新されます。長さが 32765 を超えることはありません。
DT_BIT	固定サイズ。アクションは不要です。	固定サイズ。アクションは不要です。
DT_DATE	`sqllen` フィールドは、`sqldata` 領域のバイト単位の長さに設定されます。最大長は 32767 です。	NULL 文字は文字列の最後に配置されます。
DT_DOUBLE	動作不要です。	動作不要です。
DT_FIXCHAR(n)	`sqllen` フィールドは、`sqldata` 領域のバイト単位の長さに設定されます。最大長は 32767 です。	値は、`sqllen` 値に対してブランクが埋め込まれます。
DT_FLOAT	固定サイズ。アクションは不要です。	固定サイズ。アクションは不要です。
DT_INT	固定サイズ。アクションは不要です。	固定サイズ。アクションは不要です。
DT_LONGBINARY	`sqllen` フィールドは無視されます。`array_len` フィールドによって `array` フィールドの最大長が指定されます。32767 を超えるバイトをフェッチできます。	`stored_len` および `untrunc_len` フィールドが更新されます。
DT_LONGNVARCHAR	`sqllen` フィールドは無視されます。`array_len` フィールドによって `array` フィールドの最大長が指定されます。32767 を超えるバイトをフェッチできます。	`stored_len` および `untrunc_len` フィールドが更新されます。
DT_LONGVARCHAR	`sqllen` フィールドは無視されます。`array_len` フィールドによって `array` フィールドの最大長が指定されます。32767 を超えるバイトをフェッチできます。	`stored_len` および `untrunc_len` フィールドが更新されます。

Embedded SQL のデータ型	値のフェッチ前にデータ領域の長さを指定する方法	値のフェッチ前に値の長さを決定する方法
DT_NFIXCHAR	sqllen フィールドは、sqldata 領域のバイト単位の長さに設定されます。最大長は 32767 です。	値は、sqllen 値に対してブランクが埋め込まれます。
DT_NSTRING	sqllen フィールドは、sqldata 領域のバイト単位の長さに設定されます。最大長は 32767 です。	文字列の最大長は 32766 バイトです。NULL 文字は文字列の最後に配置されます。
DT_NVARCHAR(n)	sqllen フィールドは、NVARCHAR 構造体の array フィールドのバイト単位の最大長に、len フィールドのサイズ (n+2) を加えた長さに設定されます。最大長は 32767 です。	NVARCHAR 構造体の len フィールドは、NVARCHAR 構造体の array フィールドのデータの実際のバイト単位の長さに更新されます。長さが 32765 を超えることはありません。
DT_SMALLINT	固定サイズ。アクションは不要です。	固定サイズ。アクションは不要です。
DT_STRING	sqllen フィールドは、sqldata 領域のバイト単位の長さに設定されます。最大長は 32767 です。	文字列の最大長は 32766 バイトです。NULL 文字は文字列の最後に配置されます。
DT_TIME	sqllen フィールドは、sqldata 領域のバイト単位の長さに設定されます。最大長は 32767 です。	NULL 文字は文字列の最後に配置されます。
DT_TIMESTAMP	sqllen フィールドは、sqldata 領域のバイト単位の長さに設定されます。最大長は 32767 です。	NULL 文字は文字列の最後に配置されます。
DT_TIMESTAMP_ STRUCT	固定サイズ。アクションは不要です。	固定サイズ。アクションは不要です。
DT_UNSBIGINT	固定サイズ。アクションは不要です。	固定サイズ。アクションは不要です。
DT_UNSINT	固定サイズ。アクションは不要です。	固定サイズ。アクションは不要です。
DT_UNSSMALLINT	固定サイズ。アクションは不要です。	固定サイズ。アクションは不要です。
DT_VARCHAR(n)	sqllen フィールドは、VARCHAR 構造体の array フィールドのバイト単位の最大長に、len フィールドのサイズ (n+2) を加えた長さに設定されます。最大長は 32767 です。	VARCHAR 構造体の len フィールドは、array フィールドのデータの実際のバイト単位の長さに更新されます。長さが 32765 を超えることはありません。

関連情報

動的 SQL を使用した LONG データの取得 [319 ページ]

1.10.15 Embedded SQL を使用してデータをフェッチする方法

Embedded SQL でデータをフェッチするには SELECT 文を使用します。

これには 2 つの場合があります。

SELECT 文がローを返さないか、1 つだけ返す場合

INTO 句を使用して、戻り値をホスト変数に直接割り当てます。

SELECT 文が複数のローを返す可能性がある場合

カーソルを使用して結果セットのローを管理します。

このセクションの内容:

ローを返さないか、1 つだけ返す SELECT 文 [299 ページ]
「シングルロークエリ」がデータベースから取り出すローの数は多くても 1 つだけです。

Embedded SQL でのカーソル [300 ページ]
カーソルは、結果セットに複数のローがあるクエリからローを取り出すために使用されます。

Embedded SQL を使用したワイドフェッチ [302 ページ]
FETCH 文は一度に複数のローをフェッチするように変更できます。こうするとパフォーマンスが向上することがあります。これをワイドフェッチまたは配列フェッチといいます。

1.10.15.1 ローを返さないか、1 つだけ返す SELECT 文

「シングルロークエリ」がデータベースから取り出すローの数は多くても 1 つだけです。

シングルロークエリの SELECT 文では、INTO 句が SELECT リストの後、FROM 句の前にきます。INTO 句には、SELECT リストの各項目の値を受け取るホスト変数のリストを指定します。SELECT リスト項目と同数のホスト変数を指定してください。ホスト変数と一緒に、結果が NULL であることを示すインジケータ変数も指定できます。

SELECT 文が実行されると、データベースサーバは結果を取り出して、ホスト変数に格納します。クエリの結果、複数のローが取り出されると、データベースサーバはエラーを返します。

クエリの結果、選択されたローが存在しない場合、ローが見つからないことを示すエラー (SQLCODE 100) が返されます。エラーと警告は、SQLCA 構造体で返されます。

例

次のコードは Employees テーブルから正しくローをフェッチできた場合は 1 を、ローが存在しない場合は 0 を、エラーが発生した場合は –1 を返します。

```
EXEC SQL BEGIN DECLARE SECTION;
long      id;
char      name[41];
char      sex;
char      birthdate[15];
a_sql_len ind_birthdate;
EXEC SQL END DECLARE SECTION;
...
int find_employee( long employee_id )
```

```
{
  id = employee_id;
  EXEC SQL SELECT GivenName ||
    ' ' || Surname, Sex, BirthDate
    INTO :name, :sex,
      :birthdate:ind_birthdate
    FROM Employees
    WHERE EmployeeID = :id;
  if( SQLCODE == SQLE_NOTFOUND )
  {
    return( 0 ); /* employee not found */
  }
  else if( SQLCODE < 0 )
  {
    return( -1 ); /* error */
  }
  else
  {
    return( 1 ); /* found */
  }
}
```

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.15.2 Embedded SQL でのカーソル

カーソルは、結果セットに複数のローがあるクエリからローを取り出すために使用されます。

カーソルは、SQL クエリのためのハンドルつまり識別子であり、結果セット内の位置を示します。Embedded SQL でのカーソル管理では、次の手順を実行します。

1. DECLARE CURSOR 文を使って、特定の SELECT 文のためのカーソルを宣言します。
2. OPEN 文を使って、カーソルを開きます。
3. FETCH 文を使って、一度に 1 つのローをカーソルから取り出します。
4. 警告「Row Not Found」(ローが見つかりません) が返されるまで、ローをフェッチします。
   エラーと警告は、SQLCA 構造体で返されます。
5. CLOSE 文を使ってカーソルを閉じます。

デフォルトによって、カーソルはトランザクション終了時 (COMMIT または ROLLBACK 時) に自動的に閉じられます。WITH HOLD 句を指定して開いたカーソルは、明示的に閉じるまで以降のトランザクション中も開いたままになります。

次は、簡単なカーソル使用の例です。

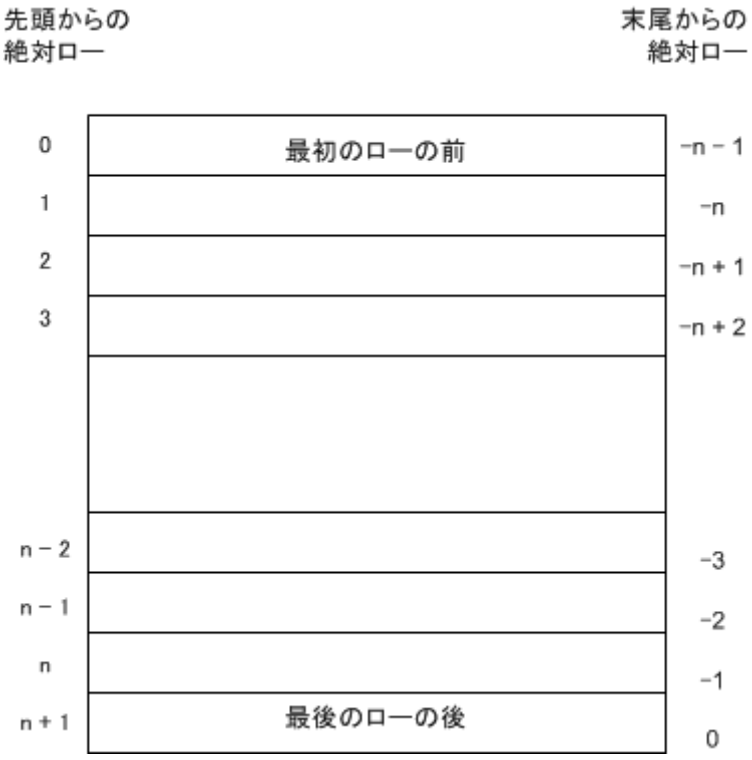
```
void print_employees( void )
{
  EXEC SQL BEGIN DECLARE SECTION;
  char       name[50];
  char       sex;
  char       birthdate[15];
  a_sql_len ind_birthdate;
  EXEC SQL END DECLARE SECTION;
```

```
  EXEC SQL DECLARE C1 CURSOR FOR
    SELECT GivenName || ' ' || Surname, Sex, BirthDate FROM Employees;
  EXEC SQL OPEN C1;
  for( ;; )
  {
    EXEC SQL FETCH C1 INTO :name, :sex, :birthdate:ind_birthdate;
    if( SQLCODE == SQLE_NOTFOUND )
    {
      break;
    }
    else if( SQLCODE < 0 )
    {
      break;
    }
    if( ind_birthdate < 0 )
    {
      strcpy( birthdate, "UNKNOWN" );
    }
    printf( "Name: %s Sex: %c Birthdate: %s¥n", name, sex, birthdate );
  }
  EXEC SQL CLOSE C1;
}
```

カーソル位置

カーソルは、次のいずれかの位置にあります。

- ローの上
- 最初のローの前
- 最後のローの後



カーソルを開くと最初のローの前に置かれます。カーソル位置は FETCH 文を使用して移動できます。カーソルはクエリ結果の先頭または末尾を基点にした絶対位置に位置付けできます。カーソルの現在位置を基準にした相対位置にも移動できます。

カーソルの現在位置のローを更新または削除するために、特別な位置付け型の UPDATE 文と DELETE 文があります。先頭のローの前か、末尾のローの後にカーソルがある場合、カーソルに対応するローがないことを示すエラーが返されます。

PUT 文で、カーソルにローを挿入できます。

カーソル位置に関する問題

DYNAMIC SCROLL カーソルに挿入や更新をいくつか行うと、カーソルの位置の問題が生じます。SELECT 文に ORDER BY 句を指定しないかぎり、データベースサーバはカーソル内の予測可能な位置にはローを挿入しません。場合によっては、カーソルを閉じてもう一度開かないと、挿入したローが表示されないことがあります。

これは、カーソルを開くためにテンポラリテーブルを作成する必要がある場合に起こります。

UPDATE 文は、カーソル内でローを移動させることがあります。これは、既存のインデックスを使用する ORDER BY 句がカーソルに指定されている場合に発生します (テンポラリテーブルは作成されません)。

関連情報

カーソルの原則 [19 ページ]
SQLCA (SQL Communication Area) [281 ページ]
静的カーソルのサンプル [264 ページ]
動的カーソルのサンプル [266 ページ]

1.10.15.3 Embedded SQL を使用したワイドフェッチ

FETCH 文は一度に複数のローをフェッチするように変更できます。こうするとパフォーマンスが向上することがあります。これをワイドフェッチまたは配列フェッチといいます。

ワイドプットとワイド挿入もサポートされます。

Embedded SQL でワイドフェッチを使用するには、次のような FETCH 文をコードに含めます。

```
EXEC SQL FETCH ... ARRAY nnn
```

ARRAY nnn は FETCH 文の最後の項目です。フェッチ回数を示す nnn にはホスト変数も使用できます。SQLDA 内の変数の数はローあたりのカラム数と nnn との積にしてください。最初のローは SQLDA の変数 0 から (ロー当たりのカラム数) - 1 に入り、以後のローも同様です。

各カラムは、SQLDA の各ローと同じ型にしてください。型が同じでない場合、SQLDA_INCONSISTENT エラーが返されます。

サーバはフェッチしたレコード数を SQLCOUNT に返します。この値は、エラーまたは警告がないかぎり、常に正の数です。ワイドフェッチでは、エラーがない状態で SQLCOUNT が 1 の場合、有効なローが 1 つフェッチされたことを示します。

例

次は、ワイドフェッチの使用例です。詳しい例は *%SQLANYSAMP17%*`¥SQLAnywhere¥esqlwidefetch ¥widefetch.sqc` にあります。

```
static SQLDA * PrepareSQLDA(
        a_sql_statement_number  stat0,
        unsigned                width,
        unsigned                *cols_per_row )
{
    int                     num_cols;
    unsigned                row, col, offset;
    SQLDA *                 sqlda;
    EXEC SQL BEGIN DECLARE SECTION;
    a_sql_statement_number  stat;
    EXEC SQL END DECLARE SECTION;
    stat = stat0;
    sqlda = alloc_sqlda( 100 );
    if( sqlda == NULL ) return( NULL );
    EXEC SQL DESCRIBE :stat INTO sqlda;
    *cols_per_row = num_cols = sqlda->sqld;
    if( num_cols * width > sqlda->sqln ) {
        free_sqlda( sqlda );
        sqlda = alloc_sqlda( num_cols * width );
        if( sqlda == NULL ) return( NULL );
        EXEC SQL DESCRIBE :stat INTO sqlda;
    }
    // copy first row in SQLDA setup by describe
    // to following (wide) rows
    sqlda->sqld = num_cols * width;
    offset = num_cols;
    for( row = 1; row < width; row++ ) {
        for( col = 0; col < num_cols; col++, offset++ ) {
            sqlda->sqlvar[offset].sqltype = sqlda->sqlvar[col].sqltype;
            sqlda->sqlvar[offset].sqllen = sqlda->sqlvar[col].sqllen;
            // optional: copy described column name
            memcpy( &sqlda->sqlvar[offset].sqlname,
                    &sqlda->sqlvar[col].sqlname,
                    sizeof( sqlda->sqlvar[0].sqlname ) );
        }
    }
    fill_s_sqlda( sqlda, 40 );
    return( sqlda );
err:
    return( NULL );
}
static void PrintFetchedRows( SQLDA * sqlda,
                              unsigned cols_per_row )
{
    long                    rows_fetched;
    int                     row, col, offset;
    if( SQLCOUNT == 0 ) {
        rows_fetched = 1;
    } else {
        rows_fetched = SQLCOUNT;
    }
    printf( "Fetched %d Rows:¥n", rows_fetched );
    for( row = 0; row < rows_fetched; row++ ) {
        for( col = 0; col < cols_per_row; col++ ) {
            offset = row * cols_per_row + col;
            printf( " ¥"%s¥"", (char *)sqlda->sqlvar[offset].sqldata );
        }
        printf( "¥n" );
```

```
    }
}
static int DoQuery( char * query_str0,
                    unsigned fetch_width0 )
{
    SQLDA *                 sqlda;
    unsigned                cols_per_row;

    EXEC SQL BEGIN DECLARE SECTION;
    a_sql_statement_number  stat;
    char *                  query_str;
    unsigned                fetch_width;
    EXEC SQL END DECLARE SECTION;
    query_str = query_str0;
    fetch_width = fetch_width0;
    EXEC SQL PREPARE :stat FROM :query_str;
    EXEC SQL DECLARE QCURSOR CURSOR FOR :stat FOR READ ONLY;
    EXEC SQL OPEN QCURSOR;
    sqlda = PrepareSQLDA( stat, fetch_width, &cols_per_row );
    if( sqlda == NULL ) {
        printf( "Error allocating SQLDA¥n" );
        return( SQLE_NO_MEMORY );
    }
    for( ;; ) {
        EXEC SQL FETCH QCURSOR INTO DESCRIPTOR sqlda ARRAY :fetch_width;
        if( SQLCODE != SQLE_NOERROR ) break;
        PrintFetchedRows( sqlda, cols_per_row );
    }
    EXEC SQL CLOSE QCURSOR;
    EXEC SQL DROP STATEMENT :stat;
    free_filled_sqlda( sqlda );
err:
    return( SQLCODE );
}
int main( int argc, char *argv[] )
{
    int     result_code;

    argc = ProcessOptions( argv );
    if( argc < 0 ) {
        return( 1 );
    }
    db_init( &sqlca );
    db_string_connect( &sqlca, ConnectStr );
    if( SQLCODE != SQLE_NOERROR ) {
        PrintSQLError();
        goto err;
    }
    result_code = DoQuery( QueryStr, FetchWidth );
    EXEC SQL DISCONNECT;
    db_fini( &sqlca );
    return( (result_code == 0) ? EXIT_OKAY : EXIT_FAIL );
err:
    db_fini( &sqlca );
    return( EXIT_FAIL );
}
```

ワイドフェッチの使用上の注意

- PrepareSQLDA 関数では、alloc_sqlda 関数を使用して SQLDA 用のメモリを割り付けています。この関数では、alloc_sqlda_noind 関数を使用する場合とは違って、インジケータ変数用の領域を確保できます。

- フェッチされたローの数が要求より少ないが 0 ではない場合 (たとえばカーソルの終端に達したとき)、SQLDA のフェッチされなかったローに対応する項目は、インジケータ変数に値を設定して、NULL として返されます。インジケータ変数が存在しない場合は、エラーが発生します (SQLE_NO_INDICATOR: NULL の結果に対してインジケータ変数がありません)。
- フェッチしようとしたローが更新され、警告 (SQLE_ROW_UPDATED_WARNING) が出された場合、フェッチは警告を引き起こしたロー上で停止します。そのときまでに処理されたすべてのロー (警告を起こしたローも含む) の値が返されます。SQLCOUNT には、フェッチしたローの数 (警告を引き起こしたローも含む) が入ります。SQLDA の残りの項目はすべて NULL になります。
- フェッチしようとしたローが削除またはロックされ、エラー (SQLE_NO_CURRENT_ROW または SQLE_LOCKED) が発生した場合、SQLCOUNT にはエラー発生までに読み込まれたローの数が入ります。この値にはエラーを起こしたローは含まれません。SQLDA にはローの値は入りません。エラーの時は、SQLDA に値が返らないためです。SQLCOUNT の値は、ローを読み込む必要がある場合、カーソルの再位置付けに使用できます。

関連情報

alloc_sqlda 関数 [329 ページ]
alloc_sqlda_noind 関数 [330 ページ]

1.10.16 Embedded SQL を使用したワイド挿入

INSERT 文は一度に複数のローを挿入するように使用できます。こうするとパフォーマンスが向上することがあります。これをワイド挿入または配列挿入といいます。

Embedded SQL でワイド挿入を使用するには、INSERT 文を準備し、次のようにコードで実行します。

```
EXEC SQL EXECUTE ... ARRAY nnn
```

ARRAY nnn は EXECUTE 文の最後の部分です。バッチサイズを示す nnn にはホスト変数も使用できます。SQLDA 内の変数の数は、実行される文のプレースホルダの数とバッチサイズとの積にしてください。

各変数は、SQLDA の各ローと同じ型にしてください。型が同じでない場合、SQLDA_INCONSISTENT エラーが返されます。

例

次のコードは、ワイド挿入の使用例です。

```
// [wideinsert.sqc]
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sqldef.h"
EXEC SQL INCLUDE SQLCA;
EXEC SQL WHENEVER SQLERROR { PrintSQLError();
                             goto err; };
static void PrintSQLError()
{
    char buffer[200];
    printf( "SQL error %d -- %s¥n",
            SQLCODE,
```

```
            sqlerror_message( &sqlca,
                              buffer,
                              sizeof( buffer ) ) );
}
unsigned        RowsToInsert = 20;
unsigned short  BatchSize    = 5;
char *          ConnectStr   = "";
static void Usage()
{
    fprintf( stderr, "Usage: wideinsert [options] ¥n" );
    fprintf( stderr, "Options:¥n" );
    fprintf( stderr, "   -n nnn          : number of rows to insert (default:
20)¥n" );
    fprintf( stderr, "   -b nnn          : insert nnn rows at a time (default:
5)¥n" );
    fprintf( stderr, "   -c conn_str     : database connection string (required)
¥n" );
}
static int ArgumentIsASwitch( char * arg )
{
#if defined( UNIX )
    return ( arg[0] == '-' );
#else
    return ( arg[0] == '-' ) || ( arg[0] == '/' );
#endif
}
static int ProcessOptions( char * argv[] )
{
    int             argc;
    char *          arg;
    char            opt;
#define _get_arg_param()                                                    ¥
            arg += 2;                                                       ¥
            if( !arg[0] ) arg = argv[++argc];                               ¥
            if( arg == NULL )                                               ¥
            {                                                               ¥
                fprintf( stderr, "Missing argument parameter¥n" );          ¥
                return( -1 );                                               ¥
            }
    for( argc = 1; (arg = argv[argc]) != NULL; ++ argc )
    {
        if( !ArgumentIsASwitch( arg ) ) break;
        opt = arg[1];
        switch( opt )
        {
        case 'n':
            _get_arg_param();
            RowsToInsert = atol( arg );
            break;
        case 'b':
            _get_arg_param();
            BatchSize = (unsigned short) atol( arg );
            break;
        case 'c':
            _get_arg_param();
            ConnectStr = arg;
            break;
        default:
            fprintf( stderr, "**** Unknown option: -%c¥n", opt );
            Usage();
            return( -1 );
        }
    }
    if( ConnectStr[0] == '¥0' )
    {
        fprintf( stderr, "A database connection string is required.¥n" );
        Usage();
        return( -1 );
```

```
    }
    return( argc );
}
static int DoInsert( unsigned rows_to_insert, unsigned short batch_size )
{
    SQLDA                   *sqlda;
    SQLVAR                  *var;
    int                     row_number = 1;

    EXEC SQL BEGIN DECLARE SECTION;
    char                    insert_stmt[ 100 ];
    unsigned short          array_size;
    unsigned                row_count;
    EXEC SQL END DECLARE SECTION;
    // Create the table for inserting the rows
    EXEC SQL DROP TABLE IF EXISTS WideInsertSample;
    EXEC SQL CREATE TABLE WideInsertSample( Col1 int, Col2 char(20), Col3 int );
    #define NUM_PARAMS 3
    sprintf( insert_stmt, "INSERT INTO WideInsertSample( Col1, Col2, Col3 )
VALUES( ?, ?, ? )" );

    sqlda = alloc_sqlda_noind( batch_size * NUM_PARAMS );
    if( sqlda == NULL )
    {
        printf( "Error allocating SQLDA¥n" );
        return( SQLE_NO_MEMORY );
    }

    // Prepare the static parts of the SQLDA object for the insert
    sqlda->sqld = batch_size * NUM_PARAMS;
    for( unsigned short current_row = 0; current_row < batch_size; current_row++ )
    {
        var = &sqlda->sqlvar[ current_row * NUM_PARAMS + 0 ];
        var->sqltype = DT_INT;
        var->sqllen = sizeof( int );
        var = &sqlda->sqlvar[ current_row * NUM_PARAMS + 1 ];
        var->sqltype = DT_STRING;
        var->sqllen = 30;
        var = &sqlda->sqlvar[ current_row * NUM_PARAMS + 2 ];
        var->sqltype = DT_INT;
        var->sqllen = sizeof( int );
   }

    fill_sqlda( sqlda );

    printf( "Insert %u rows into table ¥"WideInsertSample¥" with batch size %u.
¥n",
        rows_to_insert,
        batch_size );

    EXEC SQL PREPARE wide_insert_stmt FROM :insert_stmt;
    while( rows_to_insert > 0 )
    {
        if( rows_to_insert > batch_size )
        {
            array_size = batch_size;
        }
        else
        {
            array_size = (unsigned short) rows_to_insert;
        }
        // Fill in data values
        for( unsigned short current_row = 0; current_row < array_size; current_row
++ )
        {
            *(int *)sqlda->sqlvar[ current_row * NUM_PARAMS + 0 ].sqldata =
row_number;
```

```
            sprintf( (char *)sqlda->sqlvar[ current_row * NUM_PARAMS +
1 ].sqldata,
                     "This is row #%u", row_number );
            *(int *)sqlda->sqlvar[ current_row * NUM_PARAMS + 2 ].sqldata =
row_number * 2;
            row_number++;
        }
        // Insert a batch of rows
        EXEC SQL EXECUTE wide_insert_stmt USING DESCRIPTOR sqlda
ARRAY :array_size;
        printf( "Inserted %u rows; ", SQLCOUNT );

        EXEC SQL SELECT COUNT(*) INTO :row_count FROM WideInsertSample;
        printf( "total %u rows in table.¥n", row_count );

        rows_to_insert -= array_size;
    }
    printf( "Done.¥n" );

    EXEC SQL COMMIT;
    EXEC SQL DROP STATEMENT wide_insert_stmt;
    free_filled_sqlda( sqlda );
err:
    return( SQLCODE );
}
int main( int argc, char *argv[] )
{
    int     result_code;

    argc = ProcessOptions( argv );
    if( argc < 0 )
    {
        return( 1 );
    }
    db_init( &sqlca );
    db_string_connect( &sqlca, ConnectStr );
    if( SQLCODE != SQLE_NOERROR )
    {
        PrintSQLError();
        goto err;
    }
    result_code = DoInsert( RowsToInsert, BatchSize );
    EXEC SQL DISCONNECT;
    db_fini( &sqlca );
    return( (result_code == 0) ? EXIT_OKAY : EXIT_FAIL );
err:
    db_fini( &sqlca );
    return( EXIT_FAIL );
}
```

ワイド挿入の使用上の注意

- SQLDA のサイズは、挿入するパラメータまたは列の数 (次の例では NUM_PARAMS) をかけたバッチのサイズ (一度に挿入する最大行数) に基づいています。SQLDA のメモリは、alloc_sqlda_noind 関数で割り当てられています。この関数は、インジケータ変数の余地を割り当てません。

  ```
  sqlda = alloc_sqlda_noind( batch_size * NUM_PARAMS );
  ```

- 一度に 1 行および 1 列ずつ SQLDA 全体が初期化されます。通常、次の例のように、ゼロベースのオフセットを使用して行または列ごとの SQLDA の位置が計算されます。

```
var = &sqlda->sqlvar[ row_index * num_params + parameter_index ]
var->sqltype = column_type;
var->sqllen = column_size;
```

- これを実行したら、fill_sqlda ルーチンを呼び出し、バッチのすべての行で列値のバッファを割り当てることができます。準備している INSERT 文を実行する前に、ゼロベースオフセットを使用してバッファに値が入ります。整数値を格納する例を次に示します。

```
*(int *)sqlda->sqlvar[ current_row * NUM_PARAMS + current_column ].sqldata =
row_number * 2;
```

- 準備している INSERT 文を EXEC SQL EXECUTE 文を使用して実行し、挿入する行数を ARRAY 句で指定します。次はその例です。

```
EXEC SQL EXECUTE wide_insert_stmt USING DESCRIPTOR sqlda ARRAY :array_size;
```

- 挿入された行の数は SQLCOUNT で返されます。この値は、エラーや警告がない限り、常にゼロを上回っています。

```
printf( "Inserted %u rows; ", SQLCOUNT );
```

1.10.17 Embedded SQL を使用したワイド削除

DELETE 文を使用すると、任意の行のセットを削除できます。これにより、パフォーマンスが向上する可能性もあります。これをワイド削除または配列削除といいます。

Embedded SQL でワイド削除を使用するには、DELETE 文を準備し、次のようにコードで実行します。

```
EXEC SQL EXECUTE ... ARRAY nnn
```

ARRAY nnn は EXECUTE 文の最後の部分です。バッチサイズを示す nnn にはホスト変数も使用できます。SQLDA 内の変数の数は、実行される文のプレースホルダの数とバッチサイズとの積にしてください。

各変数は、SQLDA の各ローと同じ型にしてください。型が同じでない場合、SQLDA_INCONSISTENT エラーが返されます。

例

次のコードは、ワイド削除の使用例です。

```
// [widedelete.sqc]
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sqldef.h"
EXEC SQL INCLUDE SQLCA;
EXEC SQL WHENEVER SQLERROR { PrintSQLError();
                             goto err; };
static void PrintSQLError()
{
    char buffer[200];
    printf( "SQL error %d -- %s¥n",
            SQLCODE,
            sqlerror_message( &sqlca,
                              buffer,
```

```
                                sizeof( buffer ) ) );
}
unsigned        RowsToDelete[100];
unsigned short  BatchSize    = 0;
char *          ConnectStr   = "";
static void Usage()
{
    fprintf( stderr, "Usage: widedelete [options] ¥n" );
    fprintf( stderr, "Options:¥n" );
    fprintf( stderr, "   -n nnn          : row number to delete (specify up to
100 -n arguments)¥n" );
    fprintf( stderr, "   -c conn_str     : database connection string (required)
¥n" );
}
static int ArgumentIsASwitch( char * arg )
{
#if defined( UNIX )
    return ( arg[0] == '-' );
#else
    return ( arg[0] == '-' ) || ( arg[0] == '/' );
#endif
}
static int ProcessOptions( char * argv[] )
{
    int             argc;
    char *          arg;
    char            opt;
#define _get_arg_param()                                                    ¥
            arg += 2;                                                       ¥
            if( !arg[0] ) arg = argv[++argc];                               ¥
            if( arg == NULL )                                               ¥
            {                                                               ¥
                fprintf( stderr, "Missing argument parameter¥n" );          ¥
                return( -1 );                                               ¥
            }
    for( argc = 1; (arg = argv[argc]) != NULL; ++ argc )
    {
        if( !ArgumentIsASwitch( arg ) ) break;
        opt = arg[1];
        switch( opt )
        {
        case 'n':
            _get_arg_param();
            RowsToDelete[BatchSize++] = atol( arg );
            break;
        case 'c':
            _get_arg_param();
            ConnectStr = arg;
            break;
        default:
            fprintf( stderr, "**** Unknown option: -%c¥n", opt );
            Usage();
            return( -1 );
        }
    }
    if( ConnectStr[0] == '¥0' )
    {
        fprintf( stderr, "A database connection string is required.¥n" );
        Usage();
        return( -1 );
    }
    return( argc );
}
static int DoDelete( unsigned *rows_to_delete, unsigned short batch_size )
{
    SQLDA                   *sqlda;
    SQLVAR                  *var;
```

```
    EXEC SQL BEGIN DECLARE SECTION;
    char                    delete_stmt[ 100 ];
    unsigned short          array_size;
    unsigned                row_count;
    EXEC SQL END DECLARE SECTION;
    sprintf( delete_stmt, "DELETE FROM WideInsertSample WHERE Col1 = ?" );

    sqlda = alloc_sqlda_noind( batch_size );
    if( sqlda == NULL )
    {
        printf( "Error allocating SQLDA¥n" );
        return( SQLE_NO_MEMORY );
    }

    // Prepare the static parts of the SQLDA object for the delete
    sqlda->sqld = batch_size;
    for( unsigned short current_row = 0; current_row < batch_size; current_row++ )
    {
        var = &sqlda->sqlvar[ current_row ];
        var->sqltype = DT_INT;
        var->sqllen = sizeof( int );
    }

    fill_sqlda( sqlda );

    printf( "Delete %u rows from table ¥"WideInsertSample¥".¥n", batch_size );

    EXEC SQL PREPARE wide_delete_stmt FROM :delete_stmt;
    array_size = batch_size;
    // Fill in data values
    for( unsigned short current_row = 0; current_row < array_size; current_row++ )
    {
        *(int *)sqlda->sqlvar[ current_row ].sqldata =
rows_to_delete[ current_row ];
    }
    // Delete a batch of rows
    EXEC SQL EXECUTE wide_delete_stmt USING DESCRIPTOR sqlda ARRAY :array_size;
    printf( "Deleted %u rows; ", SQLCOUNT );

    EXEC SQL SELECT COUNT(*) INTO :row_count FROM WideInsertSample;
    printf( "total %u rows in table.¥n", row_count );

    EXEC SQL COMMIT;
    EXEC SQL DROP STATEMENT wide_delete_stmt;
    free_filled_sqlda( sqlda );
err:
    return( SQLCODE );
}
int main( int argc, char *argv[] )
{
    int     result_code;

    argc = ProcessOptions( argv );
    if( argc < 0 )
    {
        return( 1 );
    }
    db_init( &sqlca );
    db_string_connect( &sqlca, ConnectStr );
    if( SQLCODE != SQLE_NOERROR )
    {
        PrintSQLError();
        goto err;
    }
    result_code = DoDelete( RowsToDelete, BatchSize );
    EXEC SQL DISCONNECT;
    db_fini( &sqlca );
    return( (result_code == 0) ? EXIT_OKAY : EXIT_FAIL );
```

```
err:
    db_fini( &sqlca );
    return( EXIT_FAIL );
}
```

ワイド削除の使用上の注意

- この例を試すには、ワイド挿入のトピックの例を基に、WideInsertSample テーブルに値を入れてください。
- SQLDA のサイズは、DELETE 文のパラメータの数をかけたバッチのサイズ (一度に削除する最大行数) に基づいています。この例では、1 パラメータのみが存在しますが、もっと多くのパラメータでも構いません。SQLDA のメモリは、alloc_sqlda_noind 関数で割り当てられています。この関数は、インジケータ変数の余地を割り当てません。

  ```
  sqlda = alloc_sqlda_noind( batch_size );
  ```

- 一度に 1 行および 1 パラメータずつ SQLDA 全体が初期化されます。通常、次の例 (この DELETE の例では、num_params は 1 です) のように、ゼロベースのオフセットを使用して行またはパラメータごとの SQLDA の位置が計算されます。

  ```
  var = &sqlda->sqlvar[ row_index * num_params + parameter_index ]
  var->sqltype = column_type;
  var->sqllen = column_size;
  ```

- これを実行したら、fill_sqlda ルーチンを呼び出し、バッチのすべての行でパラメータ値のバッファを割り当てることができます。準備している DELETE 文を実行する前に、ゼロベースオフセットを使用してバッファに値が入ります。整数の行番号を格納する例を次に示します。

  ```
  *(int *)sqlda->sqlvar[ current_row ].sqldata = rows_to_delete[ current_row ];
  ```

- 準備している DELETE 文を EXEC SQL EXECUTE 文を使用して実行し、削除する行数を ARRAY 句で指定します。次はその例です。

  ```
  EXEC SQL EXECUTE wide_delete_stmt USING DESCRIPTOR sqlda ARRAY :array_size;
  ```

- 削除された行の数は SQLCOUNT で返されます。この値は、指定した行に一致する行がない場合や、エラーや警告があった場合を除き、常にゼロを上回っています。

  ```
  printf( "Deleted %u rows; ", SQLCOUNT );
  ```

1.10.18 Embedded SQL を使用したワイドマージ

MERGE 文は複数のローセットをテーブルにマージするように使用できます。こうするとパフォーマンスが向上することがあります。これをワイドマージまたは配列マージといいます。

Embedded SQL でワイドマージを使用するには、MERGE 文を準備し、次のようにコードで実行します。

```
EXEC SQL EXECUTE ... ARRAY nnn
```

ARRAY nnn は EXECUTE 文の最後の部分です。バッチサイズを示す nnn にはホスト変数も使用できます。SQLDA 内の変数の数は、実行される文のプレースホルダの数とバッチサイズとの積にしてください。

各変数は、SQLDA の各ローと同じ型にしてください。型が同じでない場合、SQLDA_INCONSISTENT エラーが返されます。

例

次のコードは、ワイドマージの例です。

```
// [widemerge.sqc]
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sqldef.h"
EXEC SQL INCLUDE SQLCA;
EXEC SQL WHENEVER SQLERROR { PrintSQLError();
                             goto err; };
static void PrintSQLError()
{
    char buffer[200];
    printf( "SQL error %d -- %s¥n",
            SQLCODE,
            sqlerror_message( &sqlca,
                              buffer,
                              sizeof( buffer ) ) );
}
char *          ConnectStr   = "";
static void Usage()
{
    fprintf( stderr, "Usage: widemerge [options] ¥n" );
    fprintf( stderr, "Options:¥n" );
    fprintf( stderr, "   -c conn_str     : database connection string (required)
¥n" );
}
static int ArgumentIsASwitch( char * arg )
{
#if defined( UNIX )
    return ( arg[0] == '-' );
#else
    return ( arg[0] == '-' ) || ( arg[0] == '/' );
#endif
}
static int ProcessOptions( char * argv[] )
{
    int             argc;
    char *          arg;
    char            opt;
#define _get_arg_param()                                                ¥
            arg += 2;                                                   ¥
            if( !arg[0] ) arg = argv[++argc];                           ¥
            if( arg == NULL )                                           ¥
            {                                                           ¥
                fprintf( stderr, "Missing argument parameter¥n" );      ¥
                return( -1 );                                           ¥
            }
    for( argc = 1; (arg = argv[argc]) != NULL; ++ argc )
    {
        if( !ArgumentIsASwitch( arg ) ) break;
        opt = arg[1];
        switch( opt )
        {
        case 'c':
            _get_arg_param();
            ConnectStr = arg;
            break;
        default:
            fprintf( stderr, "**** Unknown option: -%c¥n", opt );
            Usage();
            return( -1 );
        }
```

```
    }
    if( ConnectStr[0] == '¥0' )
    {
        fprintf( stderr, "A database connection string is required.¥n" );
        Usage();
        return( -1 );
    }
    return( argc );
}
static int DoMerge()
{
    SQLDA                   *sqlda;
    SQLVAR                  *var;
    char                    *region_id;
    unsigned                rep_id;

    EXEC SQL BEGIN DECLARE SECTION;
    char                    merge_stmt[ 200 ];
    unsigned short          batch_size;
    unsigned                row_count;
    EXEC SQL END DECLARE SECTION;
    // Create the table for inserting/merging the rows
    EXEC SQL CREATE OR REPLACE TABLE LocalSalesOrders
            (
                ID                  integer NOT NULL,
                CustomerID          integer NOT NULL,
                OrderDate           date NOT NULL,
                FinancialCode       char(2) NULL,
                Region              char(7) NULL,
                SalesRepresentative integer NOT NULL,
                CONSTRAINT SalesOrdersKey PRIMARY KEY (ID)
            );
    strcpy( merge_stmt,
            "MERGE INTO LocalSalesOrders "
            "USING WITH AUTO NAME"
            "("
            "    SELECT * FROM SalesOrders"
            "    WHERE Region = ? AND SalesRepresentative = ?"
            ") AS line_items "
            "ON PRIMARY KEY "
            "WHEN NOT MATCHED THEN INSERT" );
    #define NUM_PARAMS 2

    batch_size = 4;

    sqlda = alloc_sqlda_noind( batch_size * NUM_PARAMS );
    if( sqlda == NULL )
    {
        printf( "Error allocating SQLDA¥n" );
        return( SQLE_NO_MEMORY );
    }
    // Prepare the static parts of the SQLDA object for the merge
    sqlda->sqld = batch_size * NUM_PARAMS;
    for( unsigned short current_row = 0; current_row < batch_size; current_row++ )
    {
        var = &sqlda->sqlvar[ current_row * NUM_PARAMS + 0 ];
        var->sqltype = DT_STRING;
        var->sqllen = 10;
        var = &sqlda->sqlvar[ current_row * NUM_PARAMS + 1 ];
        var->sqltype = DT_INT;
        var->sqllen = sizeof( int );
    }

    fill_sqlda( sqlda );

    printf( "Merge %u rowsets into table.¥n", batch_size );

    EXEC SQL PREPARE wide_merge_stmt FROM :merge_stmt;
```

```
    // Fill in data values
    for( unsigned short current_row = 0; current_row < batch_size; current_row++ )
    {
        switch( current_row % 4 ) {
        case 0:
            region_id = "Eastern";
            rep_id = 299;
            break;
        case 1:
            region_id = "Western";
            rep_id = 299;
            break;
        case 2:
            region_id = "Eastern";
            rep_id = 856;
            break;
        case 3:
            region_id = "Western";
            rep_id = 856;
            break;
        }
        sprintf( (char *)sqlda->sqlvar[ current_row * NUM_PARAMS + 0 ].sqldata,
region_id );
                *(int *)sqlda->sqlvar[ current_row * NUM_PARAMS + 1 ].sqldata =
rep_id;
    }
    // Merge batches of rowsets
    EXEC SQL EXECUTE wide_merge_stmt USING DESCRIPTOR sqlda ARRAY :batch_size;
    printf( "Merged %u rows; ", SQLCOUNT );
    // Verify count
    EXEC SQL SELECT COUNT(*) INTO :row_count FROM LocalSalesOrders;
    printf( "total %u rows in table.¥n", row_count );

    EXEC SQL COMMIT;
    EXEC SQL DROP STATEMENT wide_merge_stmt;
    free_filled_sqlda( sqlda );
err:
    return( SQLCODE );
}
int main( int argc, char *argv[] )
{
    int     result_code;

    argc = ProcessOptions( argv );
    if( argc < 0 )
    {
        return( 1 );
    }
    db_init( &sqlca );
    db_string_connect( &sqlca, ConnectStr );
    if( SQLCODE != SQLE_NOERROR )
    {
        PrintSQLError();
        goto err;
    }
    result_code = DoMerge();
    EXEC SQL DISCONNECT;
    db_fini( &sqlca );
    return( (result_code == 0) ? EXIT_OKAY : EXIT_FAIL );
err:
    db_fini( &sqlca );
    return( EXIT_FAIL );
}
```

ワイドマージの使用上の注意

- この例では、サンプルデータベースを使用します。
- SQLDA のサイズは、MERGE 文のプレースホルダパラメータの数をかけたバッチのサイズ (マージの最大数) に基づいています。この例では、2 つのパラメータがあります。SQLDA のメモリは、alloc_sqlda_noind 関数で割り当てられています。この関数は、インジケータ変数の余地を割り当てません。

```
sqlda = alloc_sqlda_noind( batch_size * NUM_PARAMS );
```

- 一度に 1 行および 1 パラメータずつ SQLDA 全体が初期化されます。通常、次の例のように、ゼロベースのオフセットを使用して行またはパラメータごとの SQLDA の位置が計算されます。

```
var = &sqlda->sqlvar[ row_index * num_params + parameter_index ];
var->sqltype = column_type;
var->sqllen = column_size;
```

- これを実行したら、fill_sqlda ルーチンを呼び出し、バッチのすべての行でパラメータ値のバッファを割り当てることができます。準備している MERGE 文を実行する前に、ゼロベースオフセットを使用してバッファに値が入ります。リージョン文字列と特定の行の表示された値を格納する例を次に示します。

```
sprintf( (char *)sqlda->sqlvar[ current_row * NUM_PARAMS + 0 ].sqldata,
region_id );
         *(int *)sqlda->sqlvar[ current_row * NUM_PARAMS + 1 ].sqldata = rep_id;
```

- 準備している MERGE 文を EXEC SQL EXECUTE 文を使用して実行し、バッチのサイズを ARRAY 句で指定します。次はその例です。

```
EXEC SQL EXECUTE wide_merge_stmt USING DESCRIPTOR sqlda ARRAY :batch_size;
```

- 影響を受けた行の数は SQLCOUNT で返されます。この値は、指定した行に一致する行がない (行がすでに存在していたなど) 場合や、エラーや警告があった場合を除き、常にゼロを上回っています。

```
printf( "Merged %u rows; ", SQLCOUNT );
```

1.10.19 Embedded SQL を使用して long 値を送信し、取得する方法

Embedded SQL アプリケーションで LONG VARCHAR、LONG NVARCHAR、LONG BINARY の値を送信し、取り出す方法は、他のデータ型とは異なります。

標準的な SQLDA フィールドのデータ長は 32767 バイトに制限されています。これは、長さの情報を保持するフィールド (sqllen、*sqlind) が 16 ビット値であるためです。これらの値を 32 ビット値に変更すると、既存のアプリケーションが中断します。

LONG VARCHAR、LONG NVARCHAR、LONG BINARY の値の記述方法は、他のデータ型の場合と同じです。

静的 SQL 構造体

データ型 LONG BINARY、LONG VARCHAR、LONG NVARCHAR の割り付けられた長さ、格納された長さ、トランケートされていない長さを保持するために、別々のフィールドが使用されます。静的 SQL データ型は、`sqlca.h` に次のように定義されています。

```
#define DECL_LONGVARCHAR( size )         ¥
  struct { a_sql_uint32    array_len;    ¥
           a_sql_uint32    stored_len;   ¥
           a_sql_uint32    untrunc_len;  ¥
           char            array[size+1];¥
         }
#define DECL_LONGNVARCHAR( size )        ¥
  struct { a_sql_uint32    array_len;    ¥
           a_sql_uint32    stored_len;   ¥
           a_sql_uint32    untrunc_len;  ¥
           char            array[size+1];¥
         }
#define DECL_LONGBINARY( size )          ¥
  struct { a_sql_uint32    array_len;    ¥
           a_sql_uint32    stored_len;   ¥
           a_sql_uint32    untrunc_len;  ¥
           char            array[size];  ¥
         }
```

`size+1` の割り当ては、LONGVARCHAR/LONGNVARCHAR `array` がクライアントライブラリによって NULL で終了することを示しません。余分なバイトは、データベースからフェッチされたチャンクを NULL で終了するアプリケーション用に含まれます。`stored_len` フィールドを使用して、フェッチされるデータの量を決定します。

これらのマクロのいずれかを Embedded SQL の DECLARE SECTION で使用する場合、`array_len` フィールドは `size` に初期化されます。使用しない場合、`array_len` フィールドは初期化されません。

動的 SQL 構造体

動的 SQL の場合は、必要に応じて sqltype フィールドを DT_LONGVARCHAR、DT_LONGNVARCHAR、または DT_LONGBINARY に設定します。対応する LONGVARCHAR、LONGNVARCHAR、LONGBINARY の構造体は、次のとおりです。

```
typedef struct LONGVARCHAR {
    a_sql_uint32    array_len;
    a_sql_uint32    stored_len;
    a_sql_uint32    untrunc_len;
    char            array[1];
} LONGVARCHAR, LONGNVARCHAR, LONGBINARY;
```

構造体メンバーの定義

静的 SQL 構造体と動的 SQL 構造体のいずれの場合も、構造体メンバーは次のように定義します。

array_len

(送信と取得)構造体の配列部分に割り付けられたバイト数

stored_len

(送信と取得)配列に格納されるバイト数。常に array_len および untrunc_len 以下になります。

untrunc_len

(取得のみ)値がトランケートされなかった場合に配列に格納されるバイト数。常に stored_len 以上になります。トランケートが発生すると、値は array_len より大きくなります。

このセクションの内容:

静的 SQL を使用した LONG データの取得 [318 ページ]
静的 SQL を使用して LONG VARCHAR、LONG NVARCHAR、LONG BINARY の値を受信します。

動的 SQL を使用した LONG データの取得 [319 ページ]
動的 SQL を使用して LONG VARCHAR、LONG NVARCHAR、LONG BINARY の値を受信します。

静的 SQL を使用した LONG データの送信 [321 ページ]
Embedded SQL アプリケーションから静的 SQL を使用して、LONG 値をデータベースに送信します。

動的 SQL を使用した LONG データの送信 [322 ページ]
Embedded SQL アプリケーションから動的 SQL を使用して、LONG 値をデータベースに送信します。

1.10.19.1 静的 SQL を使用した LONG データの取得

静的 SQL を使用して LONG VARCHAR、LONG NVARCHAR、LONG BINARY の値を受信します。

手順

1. 必要に応じて、DECL_LONGVARCHAR、DECL_LONGNVARCHAR、または DECL_LONGBINARY 型のホスト変数を宣言します。array_len フィールドの値は自動的に設定されます。
2. FETCH、GET DATA、または EXECUTE INTO を使用してデータを取り出します。次の情報が設定されます。

   sqlind

   sqlind フィールドはインジケータを指します。このインジケータの値は、値が NULL の場合は負、トランケーションなしの場合は 0 で、トランケートされていない最大 32767 バイトの正の長さです。インジケータの値が正の場合、代わりに untrunc_len フィールドを使用してください。

   stored_len

   配列に格納されるバイト数。常に array_len および untrunc_len 以下になります。

   untrunc_len

   値がトランケートされなかった場合に配列に格納されるバイト数。常に stored_len 以上になります。トランケートが発生すると、値は array_len より大きくなります。

   array

   この領域にはフェッチされるデータが含まれます。データは NULL で終了しません。

結果

LONG データは静的 SQL を使用して取得されます。

例

次のコードフラグメントは、静的 Embedded SQL を使用して LONG VARCHAR データを取得するメカニズムを示しています。実際のアプリケーションではありません。

```
EXEC SQL BEGIN DECLARE SECTION;
// SQLPP initializes longdata.array_len to 128K
DECL_LONGVARCHAR(131072) longdata;
EXEC SQL END DECLARE SECTION;
// Init longdata for fetching data, not required
// since these fields are updated by FETCH
longdata.stored_len = 0;
longdata.untrunc_len = 0;
memset( longdata.array, 0, 131072 );
EXEC SQL FETCH ABSOLUTE 1 c1 INTO :longdata;
printf( "Length fetched %d, Actual length %d, Data[0..19] ¥"%20.20s¥"¥n",
    longdata.stored_len, longdata.untrunc_len, longdata.array );
```

関連情報

Embedded SQL インジケータ変数 [278 ページ]

1.10.19.2 動的 SQL を使用した LONG データの取得

動的 SQL を使用して LONG VARCHAR、LONG NVARCHAR、LONG BINARY の値を受信します。

手順

1. sqltype フィールドを必要に応じて DT_LONGVARCHAR、DT_LONGNVARCHAR、または DT_LONGBINARY に設定します。
2. sqldata フィールドを、LONGVARCHAR、LONGNVARCHAR、または LONGBINARY 構造体を指すように設定します。

   LONGVARCHARSIZE(n)、LONGNVARCHARSIZE(n)、または LONGBINARYSIZE(n) マクロを使用して、array フィールドに n バイトのデータを保持するために割り付ける合計バイト数を決定できます。
3. ホスト変数構造体の array_len フィールドを、array フィールドに割り付けるバイト数に設定します。
4. FETCH、GET DATA、または EXECUTE INTO を使用してデータを取り出します。次の情報が設定されます。

   sqlind

sqlind フィールドはインジケータを指します。このインジケータの値は、値が NULL の場合は負、トランケーションなしの場合は 0 で、トランケートされていない最大 32767 バイトの正の長さです。インジケータの値が正の場合、代わりに untrunc_len フィールドを使用してください。

stored_len

配列に格納されるバイト数。常に array_len および untrunc_len 以下になります。

untrunc_len

値がトランケートされなかった場合に配列に格納されるバイト数。常に stored_len 以上になります。トランケートが発生すると、値は array_len より大きくなります。

array

この領域にはフェッチされるデータが含まれます。データは NULL で終了しません。

結果

LONG データは動的 SQL を使用して取得されます。

例

次のコードフラグメントは、動的 Embedded SQL を使用して LONG VARCHAR データを取り出すメカニズムを示しています。実際のアプリケーションではありません。

```
#define DATA_LEN 128000
void get_test_var()
{
  LONGVARCHAR *longptr;
  SQLDA       *sqlda;
  SQLVAR      *sqlvar;
  sqlda = alloc_sqlda( 1 );
  longptr = (LONGVARCHAR *)malloc(
              LONGVARCHARSIZE( DATA_LEN ) );
  if( sqlda == NULL || longptr == NULL )
  {
    fatal_error( "Allocation failed" );
  }
  // init longptr for receiving data
  longptr->array_len = DATA_LEN;
  // init sqlda for receiving data
  // (sqllen is unused with DT_LONG types)
  sqlda->sqld = 1; // using 1 sqlvar
  sqlvar = &sqlda->sqlvar[0];
  sqlvar->sqltype = DT_LONGVARCHAR;
  sqlvar->sqldata = longptr;
  printf( "fetching test_var¥n" );
  EXEC SQL PREPARE select_stmt FROM 'SELECT test_var';
  EXEC SQL EXECUTE select_stmt INTO DESCRIPTOR sqlda;
  EXEC SQL DROP STATEMENT select_stmt;
  printf( "stored_len: %d, untrunc_len: %d, "
          "1st char: %c, last char: %c¥n",
        longptr->stored_len,
        longptr->untrunc_len,
        longptr->array[0],
        longptr->array[DATA_LEN - 1] );
  free_sqlda( sqlda );
  free( longptr );
}
```

関連情報

Embedded SQL インジケータ変数 [278 ページ]

1.10.19.3 静的 SQL を使用した LONG データの送信

Embedded SQL アプリケーションから静的 SQL を使用して、LONG 値をデータベースに送信します。

手順

1. 必要に応じて、DECL_LONGVARCHAR、DECL_LONGNVARCHAR、または DECL_LONGBINARY 型のホスト変数を宣言します。
2. NULL を送信する場合は、インジケータ変数を負の値に設定します。
3. ホスト変数構造体の stored_len フィールドを、array フィールド内のデータのバイト数に設定します。
4. カーソルを開くか、文を実行して、データを送信します。

結果

Embedded SQL アプリケーションでの、データベースへの LONG 値の送信準備ができています。

例

次のコードフラグメントは、静的 Embedded SQL を使用して LONG VARCHAR データを送信するメカニズムを示しています。実際のアプリケーションではありません。

```
#define CURRENT_LEN (64*1024)
EXEC SQL BEGIN DECLARE SECTION;
// SQLPP initializes longdata.array_len to 128K
DECL_LONGVARCHAR(131072) longdata;
EXEC SQL END DECLARE SECTION;
void set_test_var()
{
  // init longdata for sending data
  longdata.stored_len = CURRENT_LEN;
  memset( longdata.array, 'a', CURRENT_LEN );
  printf( "Setting test_var to %d a's¥n", CURRENT_LEN );
  EXEC SQL SET test_var = :longdata;
}
```

関連情報

Embedded SQL インジケータ変数 [278 ページ]

1.10.19.4 動的 SQL を使用した LONG データの送信

Embedded SQL アプリケーションから動的 SQL を使用して、LONG 値をデータベースに送信します。

手順

1. sqltype フィールドを必要に応じて DT_LONGVARCHAR、DT_LONGNVARCHAR、または DT_LONGBINARY に設定します。
2. NULL を送信する場合は、`* sqlind` を負の値に設定します。
3. NULL 値を送信しない場合は、sqldata フィールドを LONGVARCHAR、LONGNVARCHAR、LONGBINARY ホスト変数構造体を指すように設定します。

   LONGVARCHARSIZE(n)、LONGNVARCHARSIZE(n)、または LONGBINARYSIZE(n) マクロを使用して、array フィールドに n バイトのデータを保持するために割り付ける合計バイト数を決定できます。
4. ホスト変数構造体の array_len フィールドを、array フィールドに割り付けるバイト数に設定します。
5. ホスト変数構造体の stored_len フィールドを、array フィールド内のデータのバイト数に設定します。このバイト数は array_len 以下にしてください。
6. カーソルを開くか、文を実行して、データを送信します。

結果

Embedded SQL アプリケーションでの、データベースへの LONG 値の送信準備ができています。

関連情報

Embedded SQL インジケータ変数 [278 ページ]

1.10.20 Embedded SQL での簡単なストアドプロシージャ

Embedded SQL を使用して、ストアドプロシージャの作成と呼び出しを行うことができます。

CREATE PROCEDURE は、CREATE TABLE など、他のデータ定義文と同じように埋め込むことができます。また、ストアドプロシージャを実行する CALL 文を埋め込むこともできます。次のコードフラグメントは、Embedded SQL でストアドプロシージャを作成して実行する方法を示しています。

```
EXEC SQL CREATE PROCEDURE pettycash(
 IN Amount DECIMAL(10,2) )
BEGIN
 UPDATE account
 SET balance = balance - Amount
 WHERE name = 'bank';
 UPDATE account
 SET balance = balance + Amount
 WHERE name = 'pettycash expense';
END;
EXEC SQL CALL pettycash( 10.72 );
```

ホスト変数の値をストアドプロシージャに渡す場合、または出力変数を取り出す場合は、CALL 文を準備して実行します。次のコードフラグメントは、ホスト変数の使用方法を示しています。EXECUTE 文では、USING 句と INTO 句の両方を使用しています。

```
EXEC SQL BEGIN DECLARE SECTION;
double  hv_expense;
double  hv_balance;
EXEC SQL END DECLARE SECTION;
hv_expense = 20.00;
db_init( &sqlca );
EXEC SQL CONNECT USING 'UID=DBA;PWD=passwd';
EXEC SQL CREATE OR REPLACE PROCEDURE pettycash(
   IN expense  DECIMAL(10,2),
   OUT endbalance DECIMAL(10,2) )
BEGIN
  UPDATE account
  SET balance = balance - expense
  WHERE name = 'bank';
  UPDATE account
  SET balance = balance + expense
  WHERE name = 'pettycash expense';
  SET endbalance = ( SELECT balance FROM account
         WHERE name = 'bank' );
END;
EXEC SQL PREPARE S1 FROM 'CALL pettycash( ?, ? )';
EXEC SQL EXECUTE S1 USING :hv_expense INTO :hv_balance;
```

このセクションの内容:

結果セットを持つストアドプロシージャ [324 ページ]
データベースプロシージャでは SELECT 文も使用できます。プロシージャの宣言に RESULT 句を使用して、結果セットのカラムの数、名前、型を指定します。

1.10.20.1 結果セットを持つストアドプロシージャ

データベースプロシージャでは SELECT 文も使用できます。プロシージャの宣言に RESULT 句を使用して、結果セットのカラムの数、名前、型を指定します。

結果セットのカラムは出力パラメータとは異なります。結果セットを持つプロシージャでは、SELECT 文の代わりに CALL 文を使用してカーソル宣言を行うことができます。

```
EXEC SQL BEGIN DECLARE SECTION;
 char hv_name[100];
EXEC SQL END DECLARE SECTION;
EXEC SQL CREATE PROCEDURE female_employees()
  RESULT( name char(50) )
BEGIN
  SELECT GivenName || Surname FROM Employees
  WHERE Sex = 'f';
END;
EXEC SQL PREPARE S1 FROM 'CALL female_employees()';
EXEC SQL DECLARE C1 CURSOR FOR S1;
EXEC SQL OPEN C1;
for(;;)
{
  EXEC SQL FETCH C1 INTO :hv_name;
  if( SQLCODE != SQLE_NOERROR ) break;
  printf( "%s¥n", hv_name );
}
EXEC SQL CLOSE C1;
```

この例では、プロシージャは EXECUTE 文ではなく OPEN 文を使用して呼び出されています。OPEN 文の場合は、SELECT 文が見つかるまでプロシージャが実行されます。このとき、C1 はデータベースプロシージャ内の SELECT 文のためのカーソルです。操作を終了するまで FETCH 文のすべての形式 (後方スクロールと前方スクロール) を使用できます。CLOSE 文によってプロシージャの実行が終了します。

この例では、たとえプロシージャ内の SELECT 文の後に他の文があっても、その文は実行されません。SELECT の後の文を実行するには、RESUME cursor-name 文を使用してください。RESUME 文は警告 (SQLE_PROCEDURE_COMPLETE)、または別のカーソルが残っていることを意味する SQLE_NOERROR を返します。次は select が 2 つあるプロシージャの例です。

```
EXEC SQL CREATE PROCEDURE people()
  RESULT( name char(50) )
BEGIN
 SELECT GivenName || Surname
 FROM Employees;
 SELECT GivenName || Surname
 FROM Customers;
END;
EXEC SQL PREPARE S1 FROM 'CALL people()';
EXEC SQL DECLARE C1 CURSOR FOR S1;
EXEC SQL OPEN C1;
while( SQLCODE == SQLE_NOERROR )
{
  for(;;)
  {
    EXEC SQL FETCH C1 INTO :hv_name;
    if( SQLCODE != SQLE_NOERROR ) break;
    printf( "%s¥n", hv_name );
  }
  EXEC SQL RESUME C1;
}
EXEC SQL CLOSE C1;
```

CALL 文の動的カーソル

ここまでの例は静的カーソルを使用していました。CALL 文では完全に動的なカーソルも使用できます。

DESCRIBE 文はプロシージャコールでも完全に機能します。DESCRIBE OUTPUT で、結果セットの各カラムを記述した SQLDA を生成します。

プロシージャに結果セットがない場合、SQLDA にはプロシージャの INOUT パラメータまたは OUT パラメータの記述が入ります。DESCRIBE INPUT 文はプロシージャの IN または INOUT の各パラメータを記述した SQLDA を生成します。

DESCRIBE ALL

DESCRIBE ALL は IN、INOUT、OUT、RESULT セットの全パラメータを記述します。DESCRIBE ALL は SQLDA のインジケータ変数に追加情報を設定します。

CALL 文を記述すると、インジケータ変数の DT_PROCEDURE_IN ビットと DT_PROCEDURE_OUT ビットが設定されます。DT_PROCEDURE_IN は IN または INOUT パラメータを示し、DT_PROCEDURE_OUT は INOUT または OUT パラメータを示します。プロシージャの RESULT カラムでは、両方のビットがクリアされています。

DESCRIBE OUTPUT の後、これらのビットは結果セットを持っている文 (OPEN、FETCH、RESUME、CLOSE を使用する必要がある) と、持っていない文 (EXECUTE を使用する必要がある) を区別するのに使用できます。

複数の結果セット

複数の結果セットを返すプロシージャにおいて、結果セットの形が変わる場合は、各 RESUME 文の後で再記述してください。

カーソルの現在位置を再記述するには、文ではなくカーソルを記述します。

関連情報

動的 SELECT 文 [288 ページ]

1.10.21 Embedded SQL を使用した要求管理

通常の Embedded SQL アプリケーションは、各データベース要求が完了するまで待ってから次のステップを実行する必要があります。そのため、アプリケーションで複数の実行スレッドを使用することで、他のタスクと同時に実行できます。

1 つの実行スレッドを使用する必要がある場合は、db_register_a_callback function 関数を DB_CALLBACK_WAIT オプションとともに使用してコールバック関数を登録することにより、ある程度のマルチタスクを実現できます。コールバック関数は、データベースサーバまたはクライアントライブラリがデータベース要求を処理しているあいだ、インタフェースライブラリによって繰り返し呼び出されます。

コールバック関数では、別のデータベース要求を開始することはできませんが、db_cancel_request 関数を使用して現在の要求をキャンセルすることはできます。メッセージハンドラ内で db_is_working 関数を使用して、処理中のデータベース要求があるかどうか判断できます。

関連情報

db_register_a_callback 関数 [347 ページ]
db_cancel_request 関数 [335 ページ]
db_is_working 関数 [343 ページ]

1.10.22 Embedded SQL を使用したデータベースのバックアップ

データベースのバックアップには BACKUP 文の使用をお奨めします。

別の方法として、db_backup 関数を使用して Embedded SQL アプリケーションでオンラインバックアップを実行できます。Backup ユーティリティも、この関数を使用しています。

データベースツールの DBBackup 関数を使用して、Backup ユーティリティと直接やりとりすることもできます。

他のどのバックアップ方法でも希望どおりのバックアップができない場合にのみ、db_backup 関数を使用するプログラムを記述してください。

関連情報

db_backup 関数 [331 ページ]

1.10.23 ライブラリ関数のリファレンス

Embedded SQL プリプロセッサはインタフェースライブラリまたは DLL 内の関数呼び出しを生成します。Embedded SQL プリプロセッサが生成する呼び出しの他に、データベース操作を容易にする一連のライブラリ関数も用意されています。

このような関数のプロトタイプは EXEC SQL INCLUDE SQLCA 文で含めます。

DLL のエントリポイント

DLL のエントリポイントはすべて同じです。ただし、プロトタイプには、次のように各 DLL に適した変更子が付きます。

エントリポイントを移植可能な方法で宣言するには、`sqlca.h` で定義されている _esqlentry_ を使用します。Windows プラットフォームでは、これは、__stdcall の値に解析されます。

このセクションの内容:

alloc_sqlda 関数 [329 ページ]
SQLDA を割り付けます。

alloc_sqlda_noind 関数 [330 ページ]
SQLDA にインジケータ変数用の領域を割り付けません。

db_backup 関数 [331 ページ]
データベースをバックアップします。

db_cancel_request 関数 [335 ページ]
アクティブな要求をキャンセルします。

db_change_char_charset 関数 [336 ページ]
この接続用にアプリケーションの CHAR 文字セットを変更します。

db_change_nchar_charset 関数 [337 ページ]
この接続用にアプリケーションの NCHAR 文字セットを変更します。

db_find_engine 関数 [338 ページ]
ローカルデータベースサーバのステータス情報を返します。

db_fini 関数 [339 ページ]
データベースインタフェースまたは DLL で使用されたリソースを解放します。

db_get_property 関数 [340 ページ]
接続するデータベースインタフェースまたはサーバに関する情報を取得します。

db_init 関数 [342 ページ]
データベースインタフェースライブラリを初期化します。

db_is_working 関数 [343 ページ]
データベース要求が処理中であるかどうかを示します。

db_locate_servers 関数 [343 ページ]
TCP/IP で受信しているローカルネットワーク上のすべてのデータベースサーバに関する情報を取得します。

db_locate_servers_ex 関数 [345 ページ]
TCP/IP で受信しているローカルネットワーク上のすべてのデータベースサーバに関する情報を取得します。

db_register_a_callback 関数 [347 ページ]
コールバック関数を登録します。

db_start_database 関数 [351 ページ]
サーバでデータベースを開始します。

db_start_engine 関数 [352 ページ]
データベースサーバを起動します。

db_stop_database 関数 [353 ページ]
サーバでデータベースを停止します。

db_stop_engine 関数 [354 ページ]
データベースサーバを停止します。

db_string_connect 関数 [356 ページ]
データベースサーバ上のデータベースに接続します。

db_string_disconnect 関数 [357 ページ]
データベースサーバ上のデータベースから現在または他の接続を切断します。

db_string_ping_server 関数 [358 ページ]
サーバの検索が可能かどうかを判断し、オプションで、データベースへの正常な接続が実行できるかどうかを判断します。

db_time_change 関数 [359 ページ]
クライアント側での時刻の変更をサーバに通知します。

DBAlloc 関数 [359 ページ]
SQLDA 変数のメモリを割り当てます。

DBFree 関数 [360 ページ]
DBAlloc または DBRealloc を使用して割り当てられたメモリを解放します。

DBRealloc 関数 [361 ページ]
SQLDA 変数のメモリを再割り当てします。

fill_s_sqlda 関数 [362 ページ]
すべてのデータ型を DT_STRING に変更して、SQLDA の各記述子に記述されている各変数に領域を割り付けます。

fill_sqlda 関数 [363 ページ]
SQLDA の各記述子に記述されている各変数に領域を割り付けます。

fill_sqlda_ex 関数 [364 ページ]
LONG データ型用の特別な処理を使用して、SQLDA の各記述子に記述されている各変数に領域を割り付けます。

free_filled_sqlda 関数 [365 ページ]
各 sqldata ポインタに割り付けられていたメモリと、SQLDA 自体に割り付けられていた領域を解放します。

free_sqlda 関数 [366 ページ]
SQLDA に割り付けられているメモリを解放します。

free_sqlda_noind 関数 [367 ページ]
インジケータ変数ポインタは無視して、SQLDA に割り付けられているメモリを解放します。

sql_needs_quotes 関数 [368 ページ]
SQL 識別子に引用符が必要かどうかを調べます。

sqlda_storage 関数 [369 ページ]
値を格納するために必要な記憶領域の量を調べます。

sqlda_string_length 関数 [370 ページ]
値を C の文字列として格納するために必要な記憶領域の量を調べます。

sqlerror_message 関数 [371 ページ]
エラーメッセージテキストを取得します。

1.10.23.1 alloc_sqlda 関数

SQLDA を割り付けます。

構文

```
struct sqlda * alloc_sqlda( unsigned numvar );
```

パラメータ

numvar

割り付ける変数記述子の数。

戻り値

成功した場合は SQLDA へのポインタを返し、十分なメモリがない場合は null ポインタを返します。

備考

SQLDA に `numvar` 変数の記述子を割り付けます。SQLDA の sqln フィールドを `numvar` に初期化します。インジケータ変数用の領域が割り付けられ、この領域を指すようにインジケータポインタが設定されて、インジケータ値が 0 に初期化されます。メモリを割り付けできない場合は、NULL ポインタが返されます。この関数は alloc_sqlda_noind 関数の代わりに使用できます。

関連情報

SQLDA (SQL descriptor area) [289 ページ]
fill_s_sqlda 関数 [362 ページ]
fill_sqlda 関数 [363 ページ]
fill_sqlda_ex 関数 [364 ページ]
free_sqlda 関数 [366 ページ]

1.10.23.2 alloc_sqlda_noind 関数

SQLDA にインジケータ変数用の領域を割り付けません。

構文

```
struct sqlda * alloc_sqlda_noind( unsigned numvar );
```

パラメータ

numvar

割り付ける変数記述子の数。

戻り値

成功した場合は SQLDA へのポインタを返し、十分なメモリがない場合は null ポインタを返します。

備考

SQLDA に `numvar` 変数の記述子を割り付けます。SQLDA の sqln フィールドを `numvar` に初期化します。インジケータ変数用の領域は割り付けられず、インジケータポインタは NULL ポインタとして設定されます。メモリを割り付けできない場合は、NULL ポインタが返されます。

関連情報

SQLDA (SQL descriptor area) [289 ページ]
fill_s_sqlda 関数 [362 ページ]
fill_sqlda 関数 [363 ページ]
fill_sqlda_ex 関数 [364 ページ]
free_sqlda_noind 関数 [367 ページ]

1.10.23.3 db_backup 関数

データベースをバックアップします。

構文

```
void db_backup(
SQLCA * sqlca,
int op,
int file_num,
unsigned long page_num,
struct sqlda * sqlda);
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

op

実行する動作または操作。

file_num

データベースのファイル番号。

page_num

データベースのページ番号。範囲内の値は、0 からページの最大数から 1 を引いた値までになります。

sqlda

SQLDA 構造体へのポインタ。

権限

ユーザとして BACKUP DATABASE システム権限に接続しているか、SYS_RUN_REPLICATION_ROLE システムロールを持っている必要があります。

備考

この関数を使用してアプリケーションにバックアップ機能を追加することもできますが、このタスクには BACKUP 文を使用することをおすすめします。

実行されるアクションは、op パラメータの値によって決まります。

DB_BACKUP_START

これを呼び出してからバックアップを開始します。1 つのデータベースサーバに対して同時に実行できるバックアップはデータベースごとに 1 つだけです。バックアップが完了するまで (op 値に DB_BACKUP_END を指定して db_backup が呼び出されるまで)、データベースチェックポイントは無効にされます。バックアップが開始できない場合は、SQLCODE が SQLE_BACKUP_NOT_STARTED になります。それ以外の場合は、sqlca の SQLCOUNT フィールドには各データベースページのサイズが設定されます。バックアップは一度に 1 ページずつ処理されます。

file_num、page_num、および sqlda パラメータは無視されます。

DB_BACKUP_OPEN_FILE

file_num で指定されたデータベースファイルを開きます。これによって、指定されたファイルの各ページを DB_BACKUP_READ_PAGE を使用してバックアップできます。有効なファイル番号は、ルートデータベースファイルの場合は 0 から DB_BACKUP_MAX_FILE の値までで、トランザクションログファイルの場合は 0 から DB_BACKUP_TRANS_LOG_FILE の値までです。指定されたファイルが存在しない場合は、SQLCODE は SQLE_NOTFOUND になります。その他の場合は、SQLCOUNT はファイルのページ数を含み、SQLIOESTIMATE にはデータベースファイルが作成された時間を示す 32 ビットの値 (POSIX time_t) が含まれます。オペレーティングシステムファイル名は SQLCA の sqlerrmc フィールドにあります。

page_num および sqlda パラメータは無視されます。

DB_BACKUP_READ_PAGE

file_num で指定されたデータベースファイルから 1 ページを読み込みます。page_num の値は、0 から、DB_BACKUP_OPEN_FILE オペレーションを使用した db_backup に対する呼び出しの成功によって SQLCOUNT に返されるページ数未満の値までです。その他の場合は、SQLCODE は SQLE_NOTFOUND になります。sqlda 記述子は、バッファを指す DT_BINARY または DT_LONG_BINARY 型の変数で設定してください。このバッファは、DB_BACKUP_START オペレーションを使用した db_backup の呼び出しで SQLCOUNT フィールドに返されるサイズのバイナリデータを保持するのに十分な大きさにしてください。

DT_BINARY データは、2 バイトの長さフィールドの後に実際のバイナリデータを含んでいるので、バッファはページサイズより 2 バイトだけ大きくなければなりません。

> **注記**
>
> この呼び出しによって、指定されたデータベースのページがバッファにコピーされます。ただし、バックアップメディアにバッファを保存するのはアプリケーションの役割です。

DB_BACKUP_READ_RENAME_LOG

このアクションは、トランザクションログの最後のページが返された後にデータベースサーバがトランザクションログの名前を変更して新しいログを開始する点を除けば、DB_BACKUP_READ_PAGE と同じです。

データベースサーバが現時点でログの名前を変更できない場合 (バージョン 7.0.x 以前のデータベースで、完了していないトランザクションがある場合など) は、SQLE_BACKUP_CANNOT_RENAME_LOG_YET エラーが設定されます。この場合は、返されたページを使用しないで、要求を再発行して SQLE_NOERROR を受け取ってからページを書き込んでください。SQLE_NOTFOUND 条件を受け取るまでページを読むことを続けてください。

SQLE_BACKUP_CANNOT_RENAME_LOG_YET エラーは、何回も、複数のページについて返されることがあります。リトライループでは、要求が多すぎてサーバが遅くなることがないように遅延を入れてください。

SQLE_NOTFOUND 条件を受け取った場合は、トランザクションログはバックアップに成功してファイルの名前は変更されています。古い方のトランザクションログファイルの名前は、SQLCA の sqlerrmc フィールドに返されます。

db_backup を呼び出した後に、sqlda->sqlvar[0].sqlind の値を調べてください。この値が 0 より大きい場合は、最後のログページは書き込まれていて、トランザクションログファイルの名前は変更されています。新しい名前はまだ sqlca.sqlerrmc にありますが、SQLCODE 値は SQLE_NOERROR になります。

この後、ファイルを閉じてバックアップを終了するとき以外は、db_backup を再度呼び出さないでください。再度呼び出すと、バックアップされているログファイルの 2 番めのコピーが得られ、SQLE_NOTFOUND を受け取ります。

DB_BACKUP_CLOSE_FILE

1 つのファイルの処理が完了したときに呼び出して、`file_num` で指定されたデータベースファイルを閉じます。

`page_num` および `sqlda` パラメータは無視されます。

DB_BACKUP_END

バックアップの最後に呼び出します。このバックアップが終了するまで、他のバックアップは開始できません。チェックポイントが再度有効にされます。

`file_num`、`page_num`、および `sqlda` パラメータは無視されます。

DB_BACKUP_PARALLEL_START

並列バックアップを開始します。DB_BACKUP_START と同様、1 つのデータベースサーバに対して同時に実行できるバックアップはデータベースごとに 1 つだけです。バックアップが完了するまで (`op` 値に DB_BACKUP_END を指定して db_backup が呼び出されるまで)、データベースチェックポイントは無効にされます。バックアップが開始できない場合は、SQLE_BACKUP_NOT_STARTED を受け取ります。それ以外の場合は、sqlca の SQLCOUNT フィールドには各データベースページのサイズが設定されます。

`file_num` パラメータは、トランザクションログの名前を変更し、トランザクションログの最後のページが返された後で新しいログを開始するようデータベースに指示します。値が 0 以外の場合、トランザクションログの名前が変更されるか、トランザクションログが再起動されます。それ以外の場合は、名前の変更も再起動も行われません。このパラメータにより、並列バックアップオペレーションの間は実行できない DB_BACKUP_READ_RENAME_LOG オペレーションが必要なくなります。

`page_num` パラメータは、データベースのページ数で表したクライアントバッファの最大サイズをデータベースサーバに通知します。サーバ側では、並列バックアップの読み込みで連続したページブロックを読み込もうとします。この値によって、サーバは割り付けるブロックのサイズを知ることができます。`nnn` の値を渡すと、サーバはクライアントが最大 `nnnn` ページのデータベースページをサーバから一度に受け入れる準備があることを認識します。サーバは、`nnn` ページのブロックに十分なメモリを割り付けられない場合、`nnn` より小さいサイズのページブロックを返す可能性があります。クライアント側で DB_BACKUP_PARALLEL_START を呼び出すまでデータベースページのサイズがわからない場合は、DB_BACKUP_INFO オペレーションでこの値をサーバに渡すことができます。この値は、バックアップページを取得する初回の呼び出し (DB_BACKUP_PARALLEL_READ) を実行する前に指定する必要があります。

> **注記**
>
> db_backup を使用して並列バックアップを開始すると、ライタースレッドは作成されません。db_backup の呼び出し元でデータを受け取り、ライターとして動作するようにしてください。

DB_BACKUP_INFO

このパラメータは、並列バックアップに関する追加情報をデータベースに提供します。`file_num` パラメータは提供される情報の種類を示し、`page_num` パラメータには値が指定されます。DB_BACKUP_INFO で次の追加情報を指定できます。

DB_BACKUP_INFO_CHKPT_LOG

これは、クライアント側では BACKUP 文の WITH CHECKPOINT LOG オプションと同等です。DB_BACKUP_CHKPT_COPY の page_num 値は COPY を示しますが、DB_BACKUP_CHKPT_NOCOPY の値は NO COPY を示します。値が指定されないと、デフォルトで COPY に設定されます。

DB_BACKUP_INFO_PAGES_IN_BLOCK

page_num 引数には、1 つのブロックで送信される最大ページ数が含まれます。

DB_BACKUP_INFO_WAIT_AFTER_END トランザクションが完了してトランザクションログの名前変更またはトランケートが行われるまで待機します。これは、クライアント側では BACKUP 文の WAIT AFTER END オプションと同等です。page_num 値は無視されます。DB_BACKUP_INFO_WAIT_AFTER_END は、バージョン 16 以前の SQL Anywhere データベースサーバでは無視されます。

DB_BACKUP_PARALLEL_READ

このオペレーションでは、データベースサーバから 1 ブロック分のページを読み込みます。DB_BACKUP_OPEN_FILE オペレーションを使用してバックアップするファイルをすべて開いてから DB_BACKUP_PARALLEL_READ オペレーションを呼び出します。DB_BACKUP_PARALLEL_READ では、file_num および page_num 引数は無視されます。

sqlda 記述子は、バッファを指す DT_LONGBINARY 型の変数で設定してください。DB_BACKUP_START_PARALLEL または DB_BACKUP_INFO オペレーションで指定した、nnn ページのサイズのバイナリデータを格納するのに十分なバッファを確保してください。

サーバは特定のデータベースファイルについてデータベースページの連続したブロックを返します。ブロックの最初のページのページ番号は、SQLCOUNT フィールドに返されます。ページが含まれているファイルのファイル番号は SQLIOESTIMATE フィールドに返され、この値は DB_BACKUP_OPEN_FILE 呼び出しで使用されるファイル番号の 1 つに一致します。返されるデータのサイズは DT_LONGBINARY 変数の stored_len フィールドから取得でき、常にデータベースページのサイズの倍数になります。この呼び出しによって返されるデータには指定されたファイルの連続したページのブロックが含まれていますが、別のデータブロックが順番に返されることを想定したり、データベースファイルのすべてのページが別のデータベースファイルのページの前に返されると想定することは危険です。呼び出し元では、他の別個のファイルの一部分や、別の呼び出しによって開かれたデータベースファイルの一部分を順番に関係なく受信できるよう準備しておく必要があります。

アプリケーションでは、読み込むデータのサイズが 0 になるか、sqlda->sqlvar[0].sqlind の値が 0 より大きくなるまで、このオペレーションを繰り返し呼び出してください。トランザクションログの名前を変更するか再起動してバックアップを開始すると、SQLERROR は SQLE_BACKUP_CANNOT_RENAME_LOG_YET に設定される場合があります。この場合は、返されたページを使用しないで、要求を再発行して SQLE_NOERROR を受け取ってからデータを書き込んでください。SQLE_BACKUP_CANNOT_RENAME_LOG_YET エラーは、何回も、複数のページについて返されることがあります。リトライループでは、要求が多すぎてデータベースサーバが遅くなることがないように遅延を入れてください。最初の 2 つの条件のいずれかを満たすまで、引き続きページを読み込みます。

dbbackup ユーティリティは、次のようなアルゴリズムを使用します。これは、C のコードではなく、エラーチェックは含んでいません。

```
sqlda->sqld = 1;
sqlda->sqlvar[0].sqltype = DT_LONGBINARY
/* Allocate LONGBINARY value for page buffer. It MUST have */
/* enough room to hold the requested number (128) of database pages */
sqlda->sqlvar[0].sqldata = allocated buffer
/* Open the server files needing backup */
for file_num = 0 to DB_BACKUP_MAX_FILE
  db_backup( ... DB_BACKUP_OPEN_FILE, file_num ... )
  if SQLCODE == SQLE_NOERROR
    /* The file exists */
    num_pages = SQLCOUNT
    file_time = SQLIOESTIMATE
    open backup file with name from sqlca.sqlerrmc
```

```
end for
/* read pages from the server, write them locally */
while TRUE
  /* file_no and page_no are ignored */
  db_backup( &sqlca, DB_BACKUP_PARALLEL_READ, 0, 0, &sqlda );
  if SQLCODE != SQLE_NOERROR
     break;
  if buffer->stored_len == 0 || sqlda->sqlvar[0].sqlind > 0
     break;
  /* SQLCOUNT contains the starting page number of the block    */
  /* SQLIOESTIMATE contains the file number the pages belong to */
  write block of pages to appropriate backup file
end while
/* close the server backup files */
for file_num = 0 to DB_BACKUP_MAX_FILE
    /* close backup file */
    db_backup( ... DB_BACKUP_CLOSE_FILE, file_num ... )
end for
/* shut down the backup */
db_backup( ... DB_BACKUP_END ... )
/* cleanup */
free page buffer
```

関連情報

SQLCA (SQL Communication Area) [281 ページ]
SQLDA (SQL descriptor area) [289 ページ]
Embedded SQL のデータ型 [268 ページ]

1.10.23.4 db_cancel_request 関数

アクティブな要求をキャンセルします。

構文

```
int db_cancel_request( SQLCA * sqlca );
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

戻り値

キャンセル要求が送信された場合は 1、要求が送信されなかった場合は 0 を返します。

備考

現在アクティブなデータベースサーバ要求をキャンセルします。この関数は、キャンセル要求を送信する前に、データベースサーバ要求がアクティブかどうかを調べます。

戻り値が 0 でないことが、要求がキャンセルされたことを意味するわけではありません。キャンセル要求とデータベースまたはサーバからの応答が行き違いになるようなタイミング上の危険性はほとんどありません。このような場合は、関数が TRUE を返しても、キャンセルは効力を持ちません。

db_cancel_request 関数は非同期で呼び出すことができます。別の要求が使用している可能性のある SQLCA を使用して非同期で呼び出すことができるのは、データベースインタフェースライブラリではこの関数と db_is_working だけです。

カーソル操作実行要求をキャンセルした場合は、カーソルの位置は確定されません。キャンセルしたあとは、カーソルを絶対位置に位置付けるか、閉じます。

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.23.5 db_change_char_charset 関数

この接続用にアプリケーションの CHAR 文字セットを変更します。

構文

```
unsigned int db_change_char_charset(
SQLCA * sqlca,
char * charset );
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

charset

文字セットを表す文字列。

戻り値

変更が成功した場合は 1 を返し、それ以外の場合は 0 を返します。

備考

DT_FIXCHAR、DT_VARCHAR、DT_LONGVARCHAR、および DT_STRING 型を使用して送信およびフェッチされたデータは CHAR 文字セットにあります。

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.23.6 db_change_nchar_charset 関数

この接続用にアプリケーションの NCHAR 文字セットを変更します。

構文

```
unsigned int db_change_nchar_charset(
SQLCA * sqlca,
char * charset );
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

charset

文字セットを表す文字列。

戻り値

変更が成功した場合は 1 を返し、それ以外の場合は 0 を返します。

備考

DT_NFIXCHAR、DT_NVARCHAR、DT_LONGNVARCHAR、DT_NSTRING ホスト変数型を使用して送信およびフェッチされたデータの文字セットは NCHAR です。

db_change_nchar_charset 関数が呼び出されないと、すべてのデータは CHAR 文字セットを使用して送信およびフェッチされます。通常、Unicode データを送信およびフェッチするアプリケーションでは、NCHAR 文字セットを UTF-8 に設定します。

この関数が呼び出される場合、文字セットのパラメータは一般に "UTF-8" です。NCHAR 文字セットは UTF-16 に設定できません。

Embedded SQL の場合、NCHAR、NVARCHAR、LONG NVARCHAR はそれぞれデフォルトで DT_FIXCHAR、DT_VARCHAR、DT_LONGVARCHAR と記述されます。db_change_nchar_charset 関数が呼び出された場合、これらの型はそれぞれ DT_NFIXCHAR、DT_NVARCHAR、DT_LONGNVARCHAR と記述されます。

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.23.7 db_find_engine 関数

ローカルデータベースサーバのステータス情報を返します。

構文

```
unsigned short db_find_engine(
SQLCA * sqlca,
char * name );
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

name

NULL またはサーバの名前を含む文字列。

戻り値

サーバのステータスを unsigned short 値として返します。共有メモリにサーバが見つからない場合は 0 を返します。

備考

name という名前のローカルデータベースサーバのステータス情報を示す unsigned short 値を返します。指定された名前のサーバが共有メモリを介して見つからない場合、戻り値は 0 です。0 以外の値は、ローカルサーバが現在稼働中であることを示します。

name に NULL ポインタが指定されている場合は、デフォルトデータベースサーバについて情報が返されます。

戻り値の各ビットには特定の情報が保持されています。さまざまな情報に対するビット表現の定数は、sqldef.h ヘッダファイルに定義されています。次にその意味を説明します。

DB_ENGINE

このフラグは常に設定されています。

DB_CLIENT

このフラグは常に設定されています。

DB_CAN_MULTI_DB_NAME

このフラグは使用されなくなりました。

DB_DATABASE_SPECIFIED

このフラグは常に設定されています。

DB_ACTIVE_CONNECTION

このフラグは常に設定されています。

DB_CONNECTION_DIRTY

このフラグは使用されなくなりました。

DB_CAN_MULTI_CONNECT

このフラグは使用されなくなりました。

DB_NO_DATABASES

このフラグは、サーバがデータベースを起動していない場合に設定されます。

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.23.8 db_fini 関数

データベースインタフェースまたは DLL で使用されたリソースを解放します。

構文

```
int db_fini( SQLCA * sqlca );
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

戻り値

成功した場合は 0 以外の値を返します。それ以外の場合は 0 を返します。

備考

この関数は、データベースインタフェースまたは DLL で使用されたリソースを解放します。db_fini が呼び出された後に、他のライブラリ呼び出しをしたり、Embedded SQL 文を実行したりしないでください。処理中にエラーが発生すると、SQLCA 内でエラーコードが設定され、関数は 0 を返します。エラーがなければ、0 以外の値が返されます。

使用する SQLCA ごとに 1 回ずつ db_fini を呼び出します。

Windows ダイナミックリンクライブラリ内の DllMain 関数から直接的または間接的に db_fini 関数を呼び出さないでください。DllMain のエントリポイント関数は、簡単な初期化および終了処理のみを実行するためのものです。db_fini を呼び出した場合、デッドロックや循環依存が発生する可能性があります。

Ultra Light アプリケーションでは、同等の db_fini メソッドがあります。

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.23.9 db_get_property 関数

接続するデータベースインタフェースまたはサーバに関する情報を取得します。

構文

```
unsigned int db_get_property(
SQLCA * sqlca,
a_db_property property,
char * value_buffer,
int value_buffer_size );
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

a_db_property

要求されるプロパティ。DB_PROP_CLIENT_CHARSET、DB_PROP_SERVER_ADDRESS、または DB_PROP_DBLIB_VERSION のいずれかです。

value_buffer

NULL で終了する文字列としてプロパティ値が入ります。

value_buffer_size

末尾の NULL 文字用の領域を含む、文字列 value_buffer の最大長。

戻り値

正常終了すると 1 を返し、それ以外は 0 を返します。

備考

この関数は、接続するデータベースインタフェースまたはサーバに関する情報を取得するために使用します。

次のプロパティがサポートされます。

DB_PROP_CLIENT_CHARSET

このプロパティ値はクライアントの文字セットを取得します ("windows-1252" など)。

DB_PROP_SERVER_ADDRESS

このプロパティ値は、現在の接続のサーバネットワークアドレスを印刷可能な文字列として取得します。共有メモリプロトコルは、アドレスに対して必ず空の文字列を返します。TCP/IP プロトコルは、空でない文字列アドレスを返します。

DB_PROP_DBLIB_VERSION

このプロパティ値は、データベースインタフェースライブラリのバージョンを取得します ("17.0.4.1297" など)。

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.23.10 db_init 関数

データベースインタフェースライブラリを初期化します。

構文

```
int db_init( SQLCA * sqlca );
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

戻り値

成功した場合は 0 以外の値を返します。それ以外の場合は 0 を返します。

備考

この関数は、データベースインタフェースライブラリを初期化します。この関数は、他のライブラリが呼び出される前、および Embedded SQL 文が実行される前に呼び出します。インタフェースライブラリがプログラムのために必要とするリソースは、この呼び出しで割り付けられて初期化されます。

プログラムの最後でリソースを解放するには db_fini を使用します。処理中にエラーが発生した場合は、SQLCA に渡されて 0 が返されます。エラーがなかった場合は、0 以外の値が返され、Embedded SQL 文と関数の使用を開始できます。

通常は、この関数を一度だけ呼び出して、ヘッダファイル `sqlca.h` に定義されているグローバル変数 sqlca のアドレスを渡してください。DLL または Embedded SQL を使用する複数のスレッドがあるアプリケーションを作成する場合は、使用する SQLCA ごとに 1 回ずつ db_init を呼び出します。

Ultra Light アプリケーションでは、同等の db_init メソッドがあります。

関連情報

SQLCA (SQL Communication Area) [281 ページ]
マルチスレッドまたは再入可能コードでの SQLCA 管理 [284 ページ]

1.10.23.11 db_is_working 関数

データベース要求が処理中であるかどうかを示します。

構文

```
unsigned short db_is_working( SQLCA * sqlca );
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

戻り値

アプリケーションで `sqlca` を使用するデータベース要求が処理中である場合は 1、`sqlca` を使用する要求が処理中でない場合は 0。

備考

この関数は、非同期で呼び出すことができます。別の要求が使用している可能性のある SQLCA を使用して非同期で呼び出すことができるのは、データベースインタフェースライブラリではこの関数と db_cancel_request だけです。

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.23.12 db_locate_servers 関数

TCP/IP で受信しているローカルネットワーク上のすべてのデータベースサーバに関する情報を取得します。

構文

```
unsigned int db_locate_servers(
SQLCA * sqlca,
```

```
SQL_CALLBACK_PARM callback_address,
void * callback_user_data );
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

callback_address

コールバック関数のアドレス。

callback_user_data

データを格納するユーザ定義領域のアドレス。

戻り値

正常終了すると 1 を返し、それ以外は 0 を返します。

備考

TCP/IP で受信しているローカルネットワーク上のすべてのデータベースサーバをリストして、dblocate ユーティリティによって表示される情報にプログラムからアクセスできるようにします。

コールバック関数には、次のプロトタイプが必要です。

```
int (*)( SQLCA * sqlca,
a_server_address * server_addr,
void * callback_user_data );
```

コールバック関数は、検出されたサーバごとに呼び出されます。コールバック関数が 0 を返すと、db_locate_servers はサーバ間の反復を中止します。

コールバック関数に渡される sqlca と callback_user_data は、db_locate_servers に渡されるものと同じです。2 番目のパラメータは a_server_address 構造体へのポインタです。a_server_address は次の内容を `sqlca.h` で定義します。

```
typedef struct a_server_address {
    a_sql_uint32 port_type;
    a_sql_uint32 port_num;
    char         *name;
    char         *address;
} a_server_address;
```

port_type

この時点では常に PORT_TYPE_TCP です (`sqlca.h` 内では 6 に定義されています)。

port_num

このサーバが受信している TCP ポート番号です。

name

サーバ名があるバッファを指します。

address

サーバの IP アドレスがあるバッファを指します。

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.23.13 db_locate_servers_ex 関数

TCP/IP で受信しているローカルネットワーク上のすべてのデータベースサーバに関する情報を取得します。

構文

```
unsigned int db_locate_servers_ex(
SQLCA * sqlca,
SQL_CALLBACK_PARM callback_address,
void * callback_user_data,
unsigned int bitmask);
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

callback_address

コールバック関数のアドレス。

callback_user_data

データを格納するユーザ定義領域のアドレス。

bitmask

DB_LOOKUP_FLAG_NUMERIC、DB_LOOKUP_FLAG_ADDRESS_INCLUDES_PORT、DB_LOOKUP_FLAG_DATABASES のいずれかで構成されるマスク。

戻り値

正常終了すると 1 を返し、それ以外は 0 を返します。

備考

TCP/IP で受信しているローカルネットワーク上のすべてのデータベースサーバをリストして、dblocate ユーティリティによって表示される情報にプログラムからアクセスできるようにします。また、コールバック関数に渡されるアドレスの選択に使用するマスクパラメータを提供します。

コールバック関数には、次のプロトタイプが必要です。

```
int (*)( SQLCA * sqlca,
a_server_address * server_addr,
void * callback_user_data );
```

コールバック関数は、検出されたサーバごとに呼び出されます。コールバック関数が 0 を返すと、db_locate_servers_ex はサーバ間の反復を中止します。

コールバック関数に渡される sqlca と callback_user_data は、db_locate_servers に渡されるものと同じです。2 番目のパラメータは a_server_address 構造体へのポインタです。a_server_address は次の内容を `sqlca.h` で定義します。

```
typedef struct a_server_address {
    a_sql_uint32   port_type;
    a_sql_uint32   port_num;
    char           *name;
    char           *address;
    char           *dbname;
} a_server_address;
```

port_type

この時点では常に PORT_TYPE_TCP です (`sqlca.h` 内では 6 に定義されています)。

port_num

このサーバが受信している TCP ポート番号です。

name

サーバ名があるバッファを指します。

address

サーバの IP アドレスがあるバッファを指します。

dbname

データベース名があるバッファを指します。

3 つのビットマスクフラグがサポートされています。

- DB_LOOKUP_FLAG_NUMERIC
- DB_LOOKUP_FLAG_ADDRESS_INCLUDES_PORT
- DB_LOOKUP_FLAG_DATABASES

これらのフラグは `sqlca.h` で定義されており、OR を使用して併用できます。

DB_LOOKUP_FLAG_NUMERIC は、コールバック関数に渡されたアドレスがホスト名ではなく IP アドレスであることを確認します。

DB_LOOKUP_FLAG_ADDRESS_INCLUDES_PORT では、コールバック関数に渡された a_server_address 構造体内の TCP/IP ポート番号がアドレスに含まれていることを示します。

DB_LOOKUP_FLAG_DATABASES は、検出されたデータベースごと、または検出されたデータベースサーバごと (データベース情報の送信をサポートしていないバージョン 9.0.2 以前のデータベースサーバの場合) にコールバック関数が 1 回呼び出されることを示します。

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.23.14 db_register_a_callback 関数

コールバック関数を登録します。

構文

```
void db_register_a_callback(
SQLCA * sqlca,
a_db_callback_index index,
( SQL_CALLBACK_PARM ) callback );
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

index

後述するコールバックのタイプを特定するインデックス値。

callback

ユーザ定義コールバック関数のアドレス。

備考

この関数は、コールバック関数を登録します。

DB_CALLBACK_WAIT コールバックを登録しない場合は、デフォルトでは何もアクションを実行しません。アプリケーションはブロックして、データベースの応答を待ちます。MESSAGE TO CLIENT 文のコールバックを登録してください。

コールバックを削除するには、callback 関数として NULL ポインタを渡します。

index パラメータに指定できる値を次に示します。

DB_CALLBACK_DEBUG_MESSAGE

指定の関数がデバッグメッセージごとに 1 回呼び出され、デバッグメッセージのテキストを含む NULL で終了する文字列が渡されます。デバッグメッセージは、LogFile のファイルに記録されるメッセージです。デバッグメッセージをこのコールバックに渡すには、LogFile 接続パラメータを使用する必要があります。通常、この文字列の末尾の NULL 文字の直前に改行文字 (¥n) が付いています。コールバック関数のプロトタイプを次に示します。

```
void SQL_CALLBACK debug_message_callback(
SQLCA * sqlca,
char * message_string );
```

DB_CALLBACK_START

プロトタイプを次に示します。

```
void SQL_CALLBACK start_callback( SQLCA * sqlca );
```

この関数は、データベース要求がサーバに送信される直前に呼び出されます。DB_CALLBACK_START は、Windows でのみ使用されます。

DB_CALLBACK_FINISH

プロトタイプを次に示します。

```
void SQL_CALLBACK finish_callback( SQLCA * sqlca );
```

この関数は、データベース要求に対する応答を DBLIB インタフェース DLL が受け取ったあとに呼び出されます。DB_CALLBACK_FINISH は、Windows オペレーティングシステムでのみ使用されます。

DB_CALLBACK_CONN_DROPPED

プロトタイプを次に示します。

```
void SQL_CALLBACK conn_dropped_callback (
SQLCA * sqlca,
char * conn_name );
```

この関数は、DROP CONNECTION 文を通じた活性タイムアウトのため、またはデータベースサーバがシャットダウンされているために、データベースサーバが接続を切断しようとするときに呼び出されます。複数の接続を区別できるように、接続名 conn_name が渡されます。接続が無名の場合は、値が NULL になります。

DB_CALLBACK_WAIT

プロトタイプを次に示します。

```
void SQL_CALLBACK wait_callback( SQLCA * sqlca );
```

この関数は、データベースサーバまたはクライアントライブラリがデータベース要求を処理しているあいだ、インタフェースライブラリによって繰り返し呼び出されます。

DB_CALLBACK_MESSAGE

この関数は、要求の処理中にサーバから受け取ったメッセージをアプリケーションが処理できるようにするために使用します。メッセージは、SQL MESSAGE 文を使用してクライアントアプリケーションからデータベースサーバに送信できます。実行時間が長いデータベースサーバ文によってメッセージも生成できます。

コールバックプロトタイプを次に示します。

```
void SQL_CALLBACK message_callback(
SQLCA * sqlca,
unsigned char msg_type,
an_sql_code code,
unsigned short length,
char *  msg
);
```

msg_type パラメータは、メッセージの重大度を示します。異なるメッセージタイプを異なる方法で処理できます。msg_type に指定できる次の値は、sqldef.h で定義されています。

MESSAGE_TYPE_INFO

メッセージタイプは INFO でした。

MESSAGE_TYPE_WARNING

メッセージタイプは WARNING でした。

MESSAGE_TYPE_ACTION

メッセージタイプは ACTION でした。

MESSAGE_TYPE_STATUS

メッセージタイプは STATUS でした。

MESSAGE_TYPE_PROGRESS

メッセージタイプは PROGRESS でした。このタイプのメッセージは、BACKUP DATABASE や LOAD TABLE などの実行時間が長いデータベースサーバ文によって生成されます。

code パラメータにはメッセージに関連付けられた SQLCODE を指定できます。それ以外の場合、値は 0 です。

length パラメータは、メッセージの長さを示します。

msg パラメータは、メッセージテキストを指し示します。メッセージは、NULL で終了しません。

DBLIB、ODBC および C API クライアントは、DB_CALLBACK_MESSAGE コールバックを使用して進行メッセージを取得します。たとえば、Interactive SQL のコールバックは STATUS と INFO メッセージを履歴タブに表示しますが、ACTION と WARNING メッセージはウィンドウに表示されます。アプリケーションがこのコールバックを登録しない場合は、デフォルトのコールバックが使用されます。これは、すべてのメッセージをサーバログファイルに書き込みます (デバッグがオンでログファイルが指定されている場合)。さらに、メッセージタイプ MESSAGE_TYPE_WARNING と MESSAGE_TYPE_ACTION は、オペレーティングシステムに依存した方法で表示されます。

アプリケーションによってメッセージコールバックが登録されていない場合、クライアントに送信されたメッセージは LogFile 接続パラメータが指定された際にメッセージログファイルに保存されます。また、クライアントに送信された ACTION または STATUS メッセージは、Windows オペレーティングシステムではウィンドウに表示され、UNIX オペレーティングシステムでは stderr に記録されます。

DB_CALLBACK_VALIDATE_FILE_TRANSFER

これは、ファイル転送の検証コールバック関数を登録するために使用します。転送を許可する前に、クライアントライブラリは検証コールバックが存在している場合は、それを呼び出します。ストアドプロシージャからなどの間接文の実行中にクライアントのデータ転送が要求された場合、クライアントライブラリはクライアントアプリケーションで検証コールバックが登録されていないかぎり転送を許可しません。どのような状況で検証の呼び出しが行われるかについては、以下でより詳しく説明します。

コールバックプロトタイプを次に示します。

```
int SQL_CALLBACK file_transfer_callback(
SQLCA * sqlca,
char * file_name,
int is_write
);
```

file_name パラメータは、読み込みまたは書き込み対象のファイルの名前です。is_write パラメータは、読み込み(クライアントからサーバへの転送) が要求された場合は 0、書き込みが要求された場合は 0 以外の値になります。ファイル転送が許可されない場合、コールバック関数は 0 を返します。それ以外の場合は 0 以外の値を返します。

データのセキュリティ上、サーバはファイル転送を要求している文の実行元を追跡します。サーバは、文がクライアントアプリケーションから直接受信されたものかどうかを判断します。クライアントからデータ転送を開始する際に、サーバは文の実行元に関する情報をクライアントソフトウェアに送信します。クライアント側では、クライアントアプリケーションから直接送信された文を実行するためにデータ転送が要求されている場合にかぎり、Embedded SQL クライアントライブラリはデータの転送を無条件で許可します。それ以外の場合は、上述の検証コールバックがアプリケーションで登録されていることが必要です。登録されていない場合、転送は拒否されて文が失敗し、エラーが発生します。データベース内に既に存在しているストアドプロシージャがクライアントの文で呼び出された場合、ストアドプロシージャそのものの実行はクライアントの文で開始されたものと見なされません。ただし、クライアントアプリケーションでテンポラリストアドプロシージャを明示的に作成してストアドプロシージャを実行した場合、そのプロシージャはクライアントによって開始されたものとしてサーバは処理します。同様に、クライアントアプリケーションでバッチ文を実行する場合も、バッチ文はクライアントアプリケーションによって直接実行されるものと見なされます。

例

以下の例は、MESSAGE コールバックとその登録方法を示します。

```
#include <stdio.h>
#include <stdlib.h>
#include "sqldef.h"
EXEC SQL INCLUDE SQLCA;
EXEC SQL SET SQLCA "&sqlca";
void SQL_CALLBACK messages( SQLCA *sqlca,
    unsigned char   msg_type,
    an_sql_code     sqlcode,
    unsigned short  length,
    char *          msg )
{
    size_t  mlen;
    char    *mtype;
    char    mbuffer[80];
    switch( msg_type )
    {
        case MESSAGE_TYPE_INFO:
            mtype = "INFO";
            break;
        case MESSAGE_TYPE_WARNING:
            mtype = "WARNING";
            break;
        case MESSAGE_TYPE_ACTION:
            mtype = "ACTION";
            break;
        case MESSAGE_TYPE_STATUS:
            mtype = "STATUS";
            break;
        case MESSAGE_TYPE_PROGRESS:
            mtype = "PROGRESS";
            break;
    }
    mlen = __min( length, sizeof(mbuffer) );
```

```
    strncpy( mbuffer, msg, mlen );
    mbuffer[mlen] = '¥0';
    printf( "Message was ¥"%s¥", type %s, SQLCODE(%d)¥n", mbuffer, mtype,
sqlcode );
}
int main( int argc, char *argv[] )
{
    EXEC SQL BEGIN DECLARE SECTION;
    char *sqlstr = "MESSAGE 'All is well' TYPE INFO TO CLIENT";
    char  connectstr[80];
    EXEC SQL END DECLARE SECTION;

    if( argc <= 1 ) return 1; // no password
    db_init( &sqlca );
    strcpy( connectstr, "DSN=SQL Anywhere 17 Demo;PWD=" );
    strcat( connectstr, argv[1] );
    db_string_connect( &sqlca, connectstr );
    db_register_a_callback( &sqlca, DB_CALLBACK_MESSAGE,
(SQL_CALLBACK_PARM)messages );
    EXEC SQL EXECUTE IMMEDIATE :sqlstr;
    db_string_disconnect( &sqlca, "" );
    db_fini( &sqlca );
    return 0;
}
```

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.23.15 db_start_database 関数

サーバでデータベースを開始します。

構文

```
unsigned int db_start_database( SQLCA * sqlca, char * parms );
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

parms

NULL で終了する文字列で、KEYWORD=value 形式のパラメータ設定をセミコロンで区切ったリストが含まれています。次に例を示します。

```
"UID=DBA;PWD=passwd;DBF=c:¥¥db¥¥mydatabase.db"
```

戻り値

成功した場合は 0 以外を返します。それ以外の場合は 0 を返します。

備考

可能であれば、データベースは既存のサーバで起動します。そうでない場合は、新しいサーバを起動します。

データベースがすでに実行している場合、または正しく起動した場合、戻り値は true (0 以外) であり、SQLCODE には 0 が設定されます。エラー情報は SQLCA に返されます。

ユーザ ID とパスワードがパラメータに指定されても、それらは無視されます。

データベースの開始と停止に必要な権限は、サーバコマンドラインで -gd オプションを使用して設定します。

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.23.16 db_start_engine 関数

データベースサーバを起動します。

構文

```
unsigned int db_start_engine( SQLCA * sqlca, char * parms );
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

parms

NULL で終了する文字列で、KEYWORD=value 形式のパラメータ設定をセミコロンで区切ったリストが含まれています。次に例を示します。

```
"UID=DBA;PWD=passwd;DBF=c:¥¥db¥¥mydatabase.db"
```

戻り値

成功した場合は 0 以外を返します。それ以外の場合は 0 を返します。

備考

データベースサーバが実行されていない場合、データベースサーバを起動します。

データベースサーバがすでに実行している場合、または正しく起動した場合、戻り値は TRUE (0 以外) であり、SQLCODE には 0 が設定されます。エラー情報は SQLCA に返されます。

次の db_start_engine 呼び出しは、データベースサーバを起動し、指定されたデータベースをロードして、サーバに demo という名前を付けます。

```
db_start_engine( &sqlca, "DBF=demo.db;START=dbsrv17" );
```

ForceStart (FORCE) 接続パラメータを使用して YES に設定しない限り、db_start_engine 関数は、サーバを起動する前にサーバに接続を試みます。これは、すでに稼働しているサーバに起動を試みないようにするためです。

ForceStart 接続を YES に設定した場合は、サーバを起動する前にサーバに接続を試みることはありません。これにより、次の 1 組のコマンドが期待どおりに動作します。

1. server_1 と名付けたデータベースサーバを起動します。

   ```
   dbsrv17 -n server_1 demo.db
   ```

2. 新しいサーバを強制的に起動しそれに接続します。

   ```
   db_start_engine( &sqlda,
     "START=dbsrv17 -n server_2 mydb.db;ForceStart=YES" )
   ```

ForceStart (FORCE) を使用せず、ServerName (Server) パラメータも使用しない場合、2 番目のコマンドでは server_1 に接続しようとします。db_start_engine 関数を実行しても、StartLine (START) パラメータの -n オプションでサーバ名を取得することはできません。

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.23.17 db_stop_database 関数

サーバでデータベースを停止します。

構文

```
unsigned int db_stop_database( SQLCA * sqlca, char * parms );
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

parms

NULL で終了する文字列で、KEYWORD=`value` 形式のパラメータ設定をセミコロンで区切ったリストが含まれています。次に例を示します。

```
"UID=DBA;PWD=passwd;DBF=c:¥¥db¥¥mydatabase.db"
```

戻り値

成功した場合は 0 以外を返します。それ以外の場合は 0 を返します。

備考

ServerName (Server) で識別されるサーバ上の DatabaseName (DBN) で識別されるデータベースを停止します。ServerName を指定しない場合は、デフォルトサーバが使用されます。

デフォルトでは、この関数は既存の接続があるデータベースは停止させません。Unconditional (UNC) を **`yes`** に設定した場合は、既存の接続に関係なくデータベースは停止します。

戻り値 TRUE は、エラーがなかったことを示します。

データベースの開始と停止に必要な権限は、サーバコマンドラインで -gd オプションを使用して設定します。

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.23.18 db_stop_engine 関数

データベースサーバを停止します。

構文

```
unsigned int db_stop_engine( SQLCA * sqlca, char * parms );
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

parms

NULL で終了する文字列で、KEYWORD=value 形式のパラメータ設定をセミコロンで区切ったリストが含まれています。次に例を示します。

```
"UID=DBA;PWD=passwd;DBF=c:¥¥db¥¥mydatabase.db"
```

戻り値

成功した場合は 0 以外を返します。それ以外の場合は 0 を返します。

備考

データベースサーバの実行を終了します。この関数によって実行されるステップは次のとおりです。

- ServerName (Server) パラメータと一致する名前のローカルデータベースサーバを探します。ServerName の指定がない場合は、デフォルトのローカルデータベースサーバを探します。
- 一致するサーバが見つからない場合は、この関数は正常に値を返します。
- チェックポイントをとってすべてのデータベースを停止するように指示する要求をサーバに送信します。
- データベースサーバをアンロードします。

デフォルトでは、この関数は既存の接続があるデータベースサーバは停止させません。Unconditional=yes パラメータを指定した場合は、既存の接続に関係なくデータベースサーバは停止します。

C のプログラムでは、dbstop を生成する代わりにこの関数を使用できます。戻り値 TRUE は、エラーがなかったことを示します。

db_stop_engine の使用には、-gk サーバオプションで設定される権限が適用されます。

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.23.19 db_string_connect 関数

データベースサーバ上のデータベースに接続します。

構文

```
unsigned int db_string_connect( SQLCA * sqlca, char * parms );
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

parms

NULL で終了する文字列で、KEYWORD=value 形式のパラメータ設定をセミコロンで区切ったリストが含まれています。次に例を示します。

```
"UID=DBA;PWD=passwd;DBF=c:¥¥db¥¥mydatabase.db"
```

戻り値

成功した場合は 0 以外を返します。それ以外の場合は 0 を返します。

備考

Embedded SQL CONNECT 文に対する拡張機能を提供します。

この関数で使用されるアルゴリズムについては、接続のトラブルシューティングのトピックを参照してください。

戻り値は、接続の確立に成功した場合は TRUE (0 以外)、失敗した場合は FALSE (0) です。サーバの起動、データベースの開始、または接続に対するエラー情報は SQLCA に返されます。

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.23.20 db_string_disconnect 関数

データベースサーバ上のデータベースから現在または他の接続を切断します。

構文

```
unsigned int db_string_disconnect(
    SQLCA * sqlca,
    char * parms );
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

parms

NULL で終了する文字列で、keyword=value 形式の接続パラメータをセミコロンで区切ったリストが含まれています。次に例を示します。

```
"DSN=SQL Anywhere 17 Demo;ConnectionName=esql-con-20130228;PWD=sql"
```

戻り値

成功した場合は 0 以外を返します。それ以外の場合は 0 を返します。

備考

この関数は、ConnectionName (CON) 接続パラメータ (指定されている場合) で識別される接続を切断します。他のパラメータはすべて無視されます。

文字列に ConnectionName パラメータを指定しない場合は、無名の接続が解除されます。これは、Embedded SQL DISCONNECT 文と同等の機能です。接続に成功した場合、戻り値は TRUE です。エラー情報は SQLCA に返されます。

この関数は、AutoStop=yes 接続パラメータを使用して起動されたデータベースへの接続が他にない場合は、そのデータベースを停止します。また、サーバが AutoStop=yes パラメータを使用してすでに起動されており、実行中のデータベースが他にない場合も、サーバは停止します。

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.23.21 db_string_ping_server 関数

サーバの検索が可能かどうかを判断し、オプションで、データベースへの正常な接続が実行できるかどうかを判断します。

構文

```
unsigned int db_string_ping_server(
SQLCA * sqlca,
char * connect_string,
unsigned int connect_to_db );
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

connect_string

connect_string は、通常の接続文字列です。サーバとデータベースの情報を含んでいる場合もあれば、含んでいない場合もあります。

connect_to_db

connect_to_db が 0 以外 (TRUE) なら、この関数はサーバ上のデータベースに接続を試みます。TRUE を返すのは、接続文字列で指定したサーバ上の指定したデータベースに接続できた場合のみです。

connect_to_db が 0 なら、この関数はサーバの検索を試みるだけです。TRUE を返すのは、接続文字列でサーバを検索できた場合のみです。データベースには接続を試みません。

戻り値

サーバまたはデータベースを正常に検索できた場合は TRUE (0 以外)、検索できなかった場合は FALSE (0)。サーバまたはデータベースの検索に対するエラー情報は SQLCA に返されます。

備考

この関数は、サーバの検索が可能かどうかを判断するために使用され、オプションで、データベースへの正常な接続が実行できるかどうかを判断するために使用されます。

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.23.22 db_time_change 関数

クライアント側での時刻の変更をサーバに通知します。

構文

```
unsigned int db_time_change(
SQLCA * sqlca);
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

備考

この関数を使用すると、クライアントは、クライアント側での時刻の変更をサーバに通知できます。この関数は、タイムゾーン調整を再計算して、その値をサーバに送信します。Windows プラットフォームでは、アプリケーションが WM_TIMECHANGE メッセージを受信したときにこの関数を呼び出す必要があります。これにより、時刻の変更、タイムゾーンの変更、または夏時間に伴う変更があっても、UTC タイムスタンプの整合性が保たれます。

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.23.23 DBAlloc 関数

SQLDA 変数のメモリを割り当てます。

構文

```
void * DBAlloc( size_t size );
```

パラメータ

size

割り当てるバイト数。

戻り値

DBAlloc は、割り当てられた領域への void ポインタ、または使用可能なメモリが不足している場合は NULL を返します。

備考

この関数を使用して、Embedded SQL データ領域のメモリを割り当てます。

例

この例では、3 番目の列の sqltype が DT_LONGVARCHAR であることを仮定しています。fill_sqlda はデータ領域として最大で 32767 バイトのメモリを割り当てるため、データ領域が解放されて、新しい領域が最大サイズで 100K バイトの文字列として割り当てられます。LONGVARCHAR データ構造は初期化されます。free_filled_sqlda 関数への後続の呼び出しで、この新しいデータ領域が解放されます。

```
#define MAXLEN (100*1024)
LONGVARCHAR *lvcdata;
fill_sqlda( sqlda );
DBFree( sqlda->sqlvar[ 2 ].sqldata );
sqlda->sqlvar[ 2 ].sqldata = DBAlloc( LONGVARCHARSIZE(MAXLEN) );
lvcdata = (LONGVARCHAR *)sqlda->sqlvar[ 2 ].sqldata;
lvcdata->array_len = MAXLEN;
lvcdata->stored_len = 0;
lvcdata->untrunc_len = 0;
```

1.10.23.24 DBFree 関数

DBAlloc または DBRealloc を使用して割り当てられたメモリを解放します。

構文

```
void DBFree( void * ptr );
```

パラメータ

ptr

DBAlloc または DBRealloc 関数によって割り当てられたメモリへのポインタ。

備考

この関数を使用して、Embedded SQL データ領域のメモリを解放します。

例

この例では、3 番目の列の sqltype が DT_LONGVARCHAR であることを仮定しています。fill_sqlda はデータ領域として最大で 32767 バイトのメモリを割り当てるため、データ領域は、最大サイズで 100K バイトの文字列として解放および再割り当てが行われます。LONGVARCHAR データ構造は初期化されます。free_filled_sqlda 関数への後続の呼び出しで、この新しいデータ領域が解放されます。

```
#define MAXLEN (100*1024)
LONGVARCHAR *lvcdata;
fill_sqlda( sqlda );
DBFree( sqlda->sqlvar[ 2 ].sqldata );
sqlda->sqlvar[ 2 ].sqldata = DBAlloc( LONGVARCHARSIZE(MAXLEN) );
lvcdata = (LONGVARCHAR *)sqlda->sqlvar[ 2 ].sqldata;
lvcdata->array_len = MAXLEN;
lvcdata->stored_len = 0;
lvcdata->untrunc_len = 0;
```

1.10.23.25 DBRealloc 関数

SQLDA 変数のメモリを再割り当てします。

構文

```
void * DBRealloc( void * ptr, size_t size );
```

パラメータ

ptr

前回の DBAlloc または DBRealloc の呼び出しによって割り当てられたメモリへのポインタ。

size

割り当てるバイト数。

戻り値

DBRealloc は、割り当てられた領域への void ポインタを返すか、使用可能なメモリが不足している場合は NULL を返します。

備考

この関数を使用して、Embedded SQL データ領域のメモリを再割り当てします。

例

この例では、3 番目の列の sqltype が DT_LONGVARCHAR であることを仮定しています。fill_sqlda はデータ領域として最大で 32767 バイトのメモリを割り当てるため、データ領域は、最大サイズで 100K バイトの文字列として再割り当てが行われます。LONGVARCHAR データ構造は初期化されます。free_filled_sqlda 関数への後続の呼び出しで、この新しいデータ領域が解放されます。

```
#define MAXLEN (100*1024)
LONGVARCHAR *lvcdata;
fill_sqlda( sqlda );
sqlda->sqlvar[ 2 ].sqldata = DBRealloc( sqlda->sqlvar[ 2 ].sqldata,
LONGVARCHARSIZE(MAXLEN) );
lvcdata = (LONGVARCHAR *)sqlda->sqlvar[ 2 ].sqldata;
lvcdata->array_len = MAXLEN;
lvcdata->stored_len = 0;
lvcdata->untrunc_len = 0;
```

1.10.23.26 fill_s_sqlda 関数

すべてのデータ型を DT_STRING に変更して、SQLDA の各記述子に記述されている各変数に領域を割り付けます。

構文

```
struct sqlda * fill_s_sqlda(
struct sqlda * sqlda,
unsigned int maxlen );
```

パラメータ

sqlda

SQLDA 構造体へのポインタ。

maxlen

文字列を割り付ける最大バイト数。

戻り値

成功した場合は sqlda を返します。十分なメモリがない場合は NULL を返します。

備考

この関数は、sqlda 内のすべてのデータ型を DT_STRING 型に変更することを除いて、fill_sqlda と同じです。SQLDA によって最初に指定されたデータ型の文字列表現を保持するために十分な領域が割り付けられます。最大 maxlen - 1 バイトまでです。このメモリのアドレスは、対応する記述子の sqldata フィールドに割り当てられます。maxlen の最大値は 32767 です。

DT_STRING 変数タイプでは、sqllen が更新されて NULL ターミネータが挿入されます。

SQLDA は、free_filled_sqlda 関数を使用して解放する必要があります。

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.23.27 fill_sqlda 関数

SQLDA の各記述子に記述されている各変数に領域を割り付けます。

構文

```
struct sqlda * fill_sqlda( struct sqlda * sqlda );
```

パラメータ

sqlda

SQLDA 構造体へのポインタ。

戻り値

成功した場合は sqlda を返します。十分なメモリがない場合は NULL を返します。

備考

sqlda の各記述子に記述されている各変数に領域を割り付け、このメモリのアドレスを対応する記述子の sqldata フィールドに割り当てます。記述子に示されるデータベースのタイプと長さに対して十分な領域が割り付けられます。

fill_sqlda は、DT_LONGVARCHAR、DT_LONGNVARCHAR、DT_LONGBINARY 型をそれぞれ DT_VARCHAR、DT_NVARCHAR、DT_BINARY に変換します。

特定の変数タイプでは、sqllen が更新されて、NULL ターミネータまたは 2 バイトサイズの len フィールドが挿入されます。

SQLDA は、free_filled_sqlda 関数を使用して解放する必要があります。

fill_sqlda(sqlda) は、fill_sqlda_ex(sqlda, 0) と同じです。

関連情報

SQLCA (SQL Communication Area) [281 ページ]
fill_sqlda_ex 関数 [364 ページ]
free_filled_sqlda 関数 [365 ページ]

1.10.23.28 fill_sqlda_ex 関数

LONG データ型用の特別な処理を使用して、SQLDA の各記述子に記述されている各変数に領域を割り付けます。

構文

```
struct sqlda * fill_sqlda_ex( struct sqlda * sqlda , unsigned int flags );
```

パラメータ

sqlda

SQLDA 構造体へのポインタ。

flags

0 または FILL_SQLDA_FLAG_RETURN_DT_LONG

戻り値

成功した場合は sqlda を返します。十分なメモリがない場合は NULL を返します。

備考

sqlda の各記述子に記述されている各変数に領域を割り付け、このメモリのアドレスを対応する記述子の sqldata フィールドに割り当てます。記述子に示されるデータベースのタイプと長さに対して十分な領域が割り付けられます。

特定の変数タイプでは、sqllen が更新されて、NULL ターミネータまたは 2 バイトサイズの len フィールドが挿入されます。

SQLDA は、free_filled_sqlda 関数を使用して解放する必要があります。

1 つのフラグビット FILL_SQLDA_FLAG_RETURN_DT_LONG がサポートされています。このフラグは sqlca.h で定義されています。

FILL_SQLDA_FLAG_RETURN_DT_LONG は、DT_LONGVARCHAR、DT_LONGNVARCHAR、DT_LONGBINARY 型を入力された記述子に保持します。このフラグビットを指定しない場合、fill_sqlda_ex は DT_LONGVARCHAR、DT_LONGNVARCHAR、DT_LONGBINARY 型をそれぞれ DT_VARCHAR、DT_NVARCHAR、DT_BINARY に変換します。DT_LONG 型を使用すると、DT_VARCHAR、DT_NVARCHAR、DT_BINARY における制限である 32765 バイトではなく、32767 バイトをフェッチできます。

fill_sqlda(sqlda) は、fill_sqlda_ex(sqlda, 0) と同じです。

関連情報

SQLCA (SQL Communication Area) [281 ページ]
fill_sqlda 関数 [363 ページ]
free_filled_sqlda 関数 [365 ページ]

1.10.23.29 free_filled_sqlda 関数

各 sqldata ポインタに割り付けられていたメモリと、SQLDA 自体に割り付けられていた領域を解放します。

構文

```
void free_filled_sqlda( struct sqlda * sqlda );
```

パラメータ

sqlda

SQLDA 構造体へのポインタ。

備考

各 sqldata ポインタに割り付けられていたメモリと、SQLDA 自体に割り付けられていた領域を解放します。NULL ポインタであるものは解放されません。

これが呼び出されるのは、SQLDA の sqldata フィールドの割り付けに fill_sqlda、fill_sqlda_ex、または fill_s_sqlda が使用された場合のみです。

この関数を呼び出すと、free_sqlda が自動的に呼び出されて、alloc_sqlda が割り付けたすべての記述子が解放されます。

関連情報

SQLCA (SQL Communication Area) [281 ページ]
alloc_sqlda 関数 [329 ページ]
fill_s_sqlda 関数 [362 ページ]
fill_sqlda 関数 [363 ページ]
fill_sqlda_ex 関数 [364 ページ]
free_sqlda 関数 [366 ページ]

1.10.23.30 free_sqlda 関数

SQLDA に割り付けられているメモリを解放します。

構文

```
void free_sqlda( struct sqlda * sqlda );
```

パラメータ

sqlda

SQLDA 構造体へのポインタ。

備考

この `sqlda` に割り付けられている領域を解放し、fill_sqlda などで割り付けられたインジケータ変数領域を解放します。各 sqldata ポインタによって参照されているメモリは解放しません。

関連情報

SQLDA (SQL descriptor area) [289 ページ]
alloc_sqlda 関数 [329 ページ]
fill_s_sqlda 関数 [362 ページ]
fill_sqlda 関数 [363 ページ]
fill_sqlda_ex 関数 [364 ページ]

1.10.23.31 free_sqlda_noind 関数

インジケータ変数ポインタは無視して、SQLDA に割り付けられているメモリを解放します。

構文

```
void free_sqlda_noind( struct sqlda * sqlda );
```

パラメータ

sqlda

SQLDA 構造体へのポインタ。

備考

この `sqlda` に割り付けられている領域を解放します。各 sqldata ポインタによって参照されているメモリは解放しません。インジケータ変数ポインタは無視されます。

関連情報

SQLDA (SQL descriptor area) [289 ページ]
alloc_sqlda 関数 [329 ページ]
fill_s_sqlda 関数 [362 ページ]
fill_sqlda 関数 [363 ページ]
fill_sqlda_ex 関数 [364 ページ]

1.10.23.32 sql_needs_quotes 関数

SQL 識別子に引用符が必要かどうかを調べます。

構文

```
unsigned int sql_needs_quotes( SQLCA *sqlca, char * str );
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

str

SQL 識別子の候補である文字列。

戻り値

文字列を SQL 識別子として使用するときに二重引用符で囲む必要があるかどうかを示す TRUE または FALSE を返します。

備考

この関数は、引用符が必要かどうか調べるための要求を生成してデータベースサーバに送信します。関連する情報は、sqlcode フィールドに格納されます。

戻り値とコードの組み合わせには、次の 3 つの場合があります。

return = FALSE、sqlcode = 0

この文字列に引用符は必要ありません。

return = TRUE

sqlcode は常に SQLE_WARNING となり、文字列には引用符が必要です。

return = FALSE

sqlcode が 0 でも SQLE_WARNING でもない場合は、このテストでは確定できません。

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.23.33 sqlda_storage 関数

値を格納するために必要な記憶領域の量を調べます。

構文

```
a_sql_uint32 sqlda_storage( struct sqlda * sqlda, int varno );
```

パラメータ

sqlda

SQLDA 構造体へのポインタ。

varno

0 から始まる sqlvar ホスト変数のインデックス。

戻り値

sqlda->sqlvar[varno] に記述された変数の値を格納するために必要な記憶領域の量を表す符号なし 32 ビット整数値を返します。例として、これには DT_STRING と DT_NSTRINGNULL の NULL で終了する文字と、DT_VARCHAR、DT_BINARY、DT_LONGVARCHAR、DT_LONGNVARCHAR などのタイプのオーバヘッドが含まれます。

備考

一部のタイプには、指定したサイズの変数を格納するために必要な記憶領域の量を返すヘッダファイル `sqlca.h` および `sqlda.h` で定義されるマクロも含まれます。

_BINARYSIZE(n)

指定した長さの BINARY を格納するために必要な記憶領域の量を返します。

_VARCHARSIZE(n)

指定した長さの VARCHAR を格納するために必要な記憶領域の量を返します。

DECIMALSTORAGE(n)

指定した長さの DECIMAL を格納するために必要な記憶領域の量を返します。

LONGBINARYSIZE(n)

指定した長さの LONGBINARY を格納するために必要な記憶領域の量を返します。

LONGNVARCHARSIZE(n)

指定した長さの LONGNVARCHAR を格納するために必要な記憶領域の量を返します。

LONGVARCHARSIZE(n)

指定した長さの LONGVARCHAR を格納するために必要な記憶領域の量を返します。

これらのサイズを使用して、変数に割り当てる記録領域の量を決定できます。

関連情報

SQLDA (SQL descriptor area) [289 ページ]

1.10.23.34 sqlda_string_length 関数

値を C の文字列として格納するために必要な記憶領域の量を調べます。

構文

```
a_sql_uint32 sqlda_string_length( struct sqlda * sqlda, int varno );
```

パラメータ

sqlda

SQLDA 構造体へのポインタ。

varno

sqlvar ホスト変数のインデックス。

戻り値

C の文字列 (DT_STRING データ型) の長さを表す符号なし 32 ビット整数値を返します。これは、変数 sqlda->sqlvar[varno] を保持するためにどのデータ型の場合にも必要です。NULL 終了文字も含まれます。

例

この例では、すべての列が C 文字列 (DT_STRING) としてフェッチされます。sqlda_string_length には NULL 終了文字が含まれるため、sqllen フィールドはこの文字を除外するように設定されます。sqlda_string_length の呼び出しの後に新しい sqltype を設定する必要があります。

```
for( col = 0; col < sqlda->sqld; col++ ) {
    sqlda->sqlvar[col].sqllen = sqlda_string_length( sqlda, col ) - 1;
    sqlda->sqlvar[col].sqltype = DT_STRING;
}
```

```
fill_sqlda( sqlda );
```

関連情報

SQLDA (SQL descriptor area) [289 ページ]

1.10.23.35 sqlerror_message 関数

エラーメッセージテキストを取得します。

構文

```
char * sqlerror_message( SQLCA * sqlca, char * buffer, int max );
```

パラメータ

sqlca

SQLCA 構造体へのポインタ。

buffer

メッセージを入れるバッファ (最大 `max` 文字)。

max

バッファの最大長。

戻り値

エラーメッセージを含んでいる文字列へのポインタを返します。エラーが示されなかった場合は NULL を返します。

備考

エラーメッセージを含んでいる文字列へのポインタを返します。エラーメッセージには、SQLCA 内のエラーコードに対するテキストが含まれます。エラーがなかった場合は、NULL ポインタが返されます。エラーメッセージは、指定されたバッファに入れられ、必要に応じて長さ `max` にトランケートされます。

関連情報

SQLCA (SQL Communication Area) [281 ページ]

1.10.24 Embedded SQL 文のまとめ

必ず、Embedded SQL 文の前には EXEC SQL、後ろにはセミコロン (;) を付けてください。

Embedded SQL 文は大きく 2 つに分類できます。標準の SQL 文は、単純に EXEC SQL とセミコロン (;) で囲んで、C プログラム内に置いて使用します。CONNECT、DELETE、SELECT、SET、UPDATE には、Embedded SQL でのみ使用できる追加形式があります。この追加の形式は、Embedded SQL 固有の文になります。

いくつかの SQL 文は Embedded SQL 固有であり、C プログラム内でのみ使用できます。

標準的なデータ操作文とデータ定義文は、Embedded SQL アプリケーションから使用できます。また、次の文は Embedded SQL プログラミング専用です。

ALLOCATE DESCRIPTOR 文 [ESQL]

記述子にメモリを割り付けます。

CLOSE 文 [ESQL] [SP]

カーソルを閉じます。

CONNECT 文 [ESQL] [Interactive SQL]

データベースに接続します。

DEALLOCATE DESCRIPTOR 文 [ESQL]

記述子のメモリを再使用します。

宣言セクション [ESQL]

データベースとのやりとりに使用するホスト変数を宣言します。

DECLARE CURSOR 文 [ESQL] [SP]

カーソルを宣言します。

DELETE 文 (位置付け) [ESQL] [SP]

カーソルの現在位置のローを削除します。

DESCRIBE 文 [ESQL]

特定の SQL 文用のホスト変数を記述します。

DISCONNECT 文 [ESQL] [Interactive SQL]

データベースサーバとの接続を切断します。

DROP STATEMENT 文 [ESQL]

準備文が使用したリソースを解放します。

EXECUTE 文 [ESQL]

特定の SQL 文を実行します。

EXPLAIN 文 [ESQL]

特定のカーソルの最適化方式を説明します。
FETCH 文 [ESQL] [SP]

カーソルからローをフェッチします。
GET DATA 文 [ESQL]

カーソルから長い値をフェッチします。
GET DESCRIPTOR 文 [ESQL]

SQLDA 内の変数に関する情報を取り出します。
GET OPTION 文 [ESQL]

特定のデータベースオプションの設定を取得します。
INCLUDE 文 [ESQL]

SQL 前処理用のファイルをインクルードします。
OPEN 文 [ESQL] [SP]

カーソルを開きます。
PREPARE 文 [ESQL]

特定の SQL 文を準備します。
PUT 文 [ESQL]

カーソルにローを挿入します。
SET CONNECTION 文 [Interactive SQL] [ESQL]

アクティブな接続を変更します。
SET DESCRIPTOR 文 [ESQL]

SQLDA 内で変数を記述し、SQLDA にデータを置きます。
SET SQLCA 文 [ESQL]

SQLCA をデフォルトのグローバル SQLCA 以外のものに設定します。
UPDATE (位置付け) 文 [ESQL] [SP]

カーソルの現在位置のローを更新します。
WHENEVER 文 [ESQL]

SQL 文でエラーが発生した場合の動作を指定します。

1.11 SQL Anywhere データベース API for C/C++

SQL Anywhere C アプリケーションプログラミングインタフェース (API) は、C/C++ 言語用のデータアクセス API です。

C API 仕様は、実際に使用されているデータベースとは関係なく一貫したデータベースインタフェースを提供する一連の関数、変数、規則を定義します。C API を使用すると、SQL Anywhere データベースサーバに C/C++ アプリケーションから直接アクセスできるようになります。

このセクションの内容:

SQL Anywhere C API のサポート [374 ページ]
SQL Anywhere C アプリケーションプログラミングインタフェース (API) は、C/C++ 言語用のデータアクセス API です。C API 仕様は、実際に使用されているデータベースとは関係なく一貫したデータベースインタフェースを提供する一連の関数、変数、規則を定義します。
SQL Anywhere C API リファレンス [376 ページ]
特定の C API 要素がヘッダファイルに定義されています。

1.11.1 SQL Anywhere C API のサポート

SQL Anywhere C アプリケーションプログラミングインタフェース (API) は、C/C++ 言語用のデータアクセス API です。C API 仕様は、実際に使用されているデータベースとは関係なく一貫したデータベースインタフェースを提供する一連の関数、変数、規則を定義します。

C API を使用すると、SQL Anywhere データベースサーバで実行されているデータベースに C/C++ アプリケーションから直接アクセスできるようになります。

C API は DBLIB パッケージの上層に位置し、Embedded SQL で実装されています。DBLIB に代わるものではありませんが、この API は、C や C++ によるアプリケーションの作成を簡単にします。C API を使用するのに、Embedded SQL に関する高度な知識は必要ありません。

また、C API により、PHP、Perl、Python、Ruby など、複数のインタプリター型プログラミング言語での C や C++ ラッパードライバの作成が簡単になります。

API 配布ファイル

この API は、Microsoft Windows システムではダイナミックリンクライブラリ (DLL) (`dbcapi.dll`) として、UNIX システムでは共有オブジェクト (`libdbcapi.so`) として作成されています。DLL は、DLL が構築されるソフトウェアバージョンの DBLIB パッケージに静的にリンクされます。`dbcapi.dll` ファイルがロードされると、対応する `dblibX.dll` ファイルがオペレーティングシステムによってロードされます。`dbcapi.dll` を使用するアプリケーションは、このファイルに直接リンクするか、または動的にロードできます。

C API のデータ型やエントリポイントに関する説明は `sacapi.h` ヘッダファイルを参照してください。このファイルは、インストールされているソフトウェアの `sdk¥dbcapi` ディレクトリにあります。

スレッドサポート

C API ライブラリはスレッド非対応であるため、相互排除が必要なタスクは実行しません。ライブラリをスレッドアプリケーションで使用する際には、1 つの接続につき 1 つの要求しか受けることができません。したがって、接続固有のリソースにアクセスする場合はアプリケーションが相互排除を実行します。接続固有のリソースには、接続ハンドル、準備文、結果セットオブジェクトが含まれます。

C API の例

C API の使用例については、インストールされているソフトウェアの `sdk¥dbcapi¥examples` サブディレクトリにある例を参照してください。

callback.cpp

これは、コールバックを作成および使用する方法を示した例です。

connecting.cpp

これは、接続オブジェクトを作成し、データベースに接続する方法を示した例です。

dbcapi_isql.cpp

これは、ISQL 風のアプリケーションを記述する方法を示した例です。

fetching_a_result_set.cpp

これは、結果セットからデータをフェッチする方法を示した例です。

fetching_multiple_from_sp.cpp

これは、ストアドプロシージャから複数の結果セットをフェッチする方法を示した例です。

preparing_statements.cpp

これは、文を準備および実行する方法を示した例です。

send_retrieve_full_blob.cpp

これは、blob を 1 つのチャンクで挿入および取得する方法を示した例です。

send_retrieve_part_blob.cpp

これは、blob を複数のチャンクで挿入し、複数のチャンクで取得する方法を示した例です。

このセクションの内容:

インタフェースライブラリの動的ロード [375 ページ]
C API ライブラリのメソッドにアクセスするには、動的にリンクを使用します。

1.11.1.1 インタフェースライブラリの動的ロード

C API ライブラリのメソッドにアクセスするには、動的にリンクを使用します。

DLL の動的にロードするためのコードについては `sacapidll.c` ソースファイルを参照してください。このファイルは、インストールされているソフトウェアの `sdk¥dbcapi` サブディレクトリにあります。アプリケーションでは、必ず `sacapidll.h` ヘッダファイルを使用し、ソースコードを `sacapidll.c` にインクルードする必要があります。sqlany_initialize_interface メソッドを使用して DLL を動的にロードし、エントリポイントを検索できます。インストールされているソフトウェアに付属の例を参照してください。

1.11.2 SQL Anywhere C API リファレンス

特定の C API 要素がヘッダファイルに定義されています。

ヘッダファイル

- `sacapi.h`
- `sacapidll.h`

備考

`sacapi.h` ヘッダファイルでは、SQL Anywhere C API エントリポイントを定義します。

`sacapidll.h` ヘッダファイルでは、C API ライブラリの初期化関数とファイナライズ関数を定義します。ソースファイルに `sacapidll.h` をインクルードし、`sacapidll.c` からソースコードをインクルードする必要があります。

> **注記**
>
> API リファレンスマニュアルをお探しですか。マニュアルをローカルにインストールした場合は、Windows のスタートメニューを使用してアクセスするか (Microsoft Windows)、`C:¥Program Files¥SQL Anywhere 17¥Documentation` にナビゲートします。
>
> また、DocCommentXchange の Web で、SAP SQL Anywhere API リファレンスマニュアルにアクセスすることもできます。http://dcx.sap.com

1.12 外部呼び出しインタフェース

ストアドプロシージャまたは関数から外部ライブラリの関数を呼び出すことができます。

Windows オペレーティングシステムでは DLL、UNIX では共有オブジェクトの関数を呼び出すことができます。

以降では、外部関数呼び出しインタフェースを使用する方法について説明します。サンプルの外部ストアドプロシージャと、プロシージャに含まれる DLL を構築するために必要なファイルは、次のフォルダに格納されています。*%SQLANYSAMP17%*`¥SQLAnywhere¥ExternalProcedures`

> **警告**
>
> プロシージャから呼び出された外部ライブラリは、サーバのメモリを共有します。プロシージャから呼び出した外部ライブラリがメモリ処理のエラーを含んでいると、サーバそのものがクラッシュしたり、データベースが損傷したりする可能性があります。運用データベースに配備する前にライブラリをテストする必要があります。

説明するインタフェースは、廃止された以前のインタフェースの代わりに使用します。バージョン 7.0.x 以前の古いインタフェースを使用して作成されたライブラリもサポートされますが、新しく開発する場合は、最新のインタフェースを使用してください。UNIX プラットフォームと 64 ビット Windows を含むすべての 64 ビットプラットフォームでは、新しいインタフェースを使用してください。

データベースサーバは、MAPI 電子メールの送信などでこの機能を使用するシステムプロシージャのセットを含みます。

このセクションの内容:

外部呼び出しを使ったプロシージャと関数 [377 ページ]
次のように、ライブラリ (ダイナミックリンクライブラリ (DLL) または共有オブジェクト) 内の C/C++ 関数を呼び出す SQL ストアドプロシージャを作成できます。

外部関数のプロトタイプ [378 ページ]
C または C++ で記述した関数で使用するインタフェースは、ソフトウェアインストールディレクトリの SDK¥Include サブディレクトリの dllapi.h および extfnapi.h という 2 つのヘッダファイルで定義されます。これらのヘッダファイルは、外部関数のプロトタイプのプラットフォームに依存する機能を処理します。

外部関数呼び出しインタフェースのメソッド [392 ページ]
外部関数呼び出しインタフェースのメソッドは、データベースサーバにデータを送信、またはデータベースサーバからデータを取得する場合に使用します。

データ型の処理 [396 ページ]
ほとんどの SQL データ型を外部ライブラリに渡すことができます。

外部ライブラリをアンロードする方法 [398 ページ]
システムプロシージャ sa_external_library_unload を使用して、ライブラリが使用されていないときに外部ライブラリをアンロードすることができます。

1.12.1 外部呼び出しを使ったプロシージャと関数

次のように、ライブラリ (ダイナミックリンクライブラリ (DLL) または共有オブジェクト) 内の C/C++ 関数を呼び出す SQL ストアドプロシージャを作成できます。

```
CREATE PROCEDURE coverProc( parameter-list )
  EXTERNAL NAME 'myFunction@myLibrary'
  LANGUAGE C_ESQL32;
```

外部ライブラリを参照するプロシージャまたは関数を作成するには、CREATE EXTERNAL REFERENCE システム権限が必要です。

この方法でストアドプロシージャまたは関数を定義するときは、外部 DLL の関数へのブリッジを作成することになります。ストアドプロシージャまたはファンクションは、それ以外のタスクを実行できません。

同様に、ライブラリ内の C/C++ 関数を呼び出す SQL ストアド関数を次のように作成できます。

```
CREATE FUNCTION coverFunc( parameter-list )
  RETURNS data-type
  EXTERNAL NAME 'myFunction@myLibrary'
  LANGUAGE C_ESQL32;
```

これらの文では、EXTERNAL NAME 句は関数の名前とその関数があるライブラリを示します。この例では、myFunction がライブラリのエクスポートされる関数名で、myLibrary がライブラリの名前 (たとえば、myLibrary.dll または myLibrary.so) です。

LANGUAGE 句は、関数が外部環境で呼び出されることを示しています。LANGUAGE 句では、C_ESQL32、C_ESQL64、C_ODBC32、C_ODBC64 のいずれか 1 つを指定できます。32 または 64 のサフィックスは、関数が 32 ビットまたは 64 ビットのアプリケーションとしてコンパイルされていることを示しています。ODBC 指定は、アプリケーションが ODBC API を使用することを示しています。ESQL 指定は、アプリケーションが Embedded SQL API、C API、その他の非 ODBC API を使用するか、または API を一切使用しない場合があることを示しています。

LANGUAGE 句を省略すると、関数を含むライブラリはデータベースサーバのアドレス領域にロードされます。外部関数は呼び出されるとサーバの一部として実行されます。この場合、関数によって障害が引き起こされると、データベースサーバは終了します。したがって、関数のロードおよび実行は外部環境で行うことを推奨します。関数が原因で外部環境に障害が発生した場合、データベースサーバは実行し続けます。

parameter-list の引数の型と順序は、ライブラリ関数によって定義されている引数と対応しなければなりません。ライブラリ関数は、特別なインタフェースを使ってプロシージャ引数にアクセスします。

外部関数から返される値や結果セットは、ストアドプロシージャまたは関数によって、呼び出し元の環境に返すことができます。

外部関数を参照するストアドプロシージャまたは関数に、その他の文を含めることはできません。このストアドプロシージャまたは関数の目的は、関数の引数を取ること、関数を呼び出すこと、関数から返ってきた値と引数を呼び出し元の環境に返すことです。このプロシージャ呼び出しの IN、INOUT、OUT パラメータは、通常のプロシージャの場合と同じように使用できます。入力された値は外部関数に渡され、関数によって変更されたパラメータは OUT または INOUT パラメータを通して、またはストアド関数の RETURNS 結果として、呼び出し元の環境に返されます。

あるオペレーティングシステムではある関数を呼び出し、もう 1 つのオペレーティングシステムでは (おそらく同じ機能の) 別の関数を呼び出せます。この場合の構文は、関数名にオペレーティングシステム名をプレフィクスとして付けます。オペレーティングシステムの識別子は **Unix** にしてください。次に例を示します。

```
CREATE FUNCTION func ( parameter-list )
  RETURNS data-type
  EXTERNAL NAME 'Unix:function-name@library.so;function-name@library.dll';
```

関数のリストにサーバのオペレーティングシステムのエントリがなく、オペレーティングシステムが指定されていないエントリを含む場合、データベースサーバはそのエントリのこの関数を呼び出します。

関連情報

外部環境のサポート [398 ページ]
外部関数のプロトタイプ [378 ページ]

1.12.2 外部関数のプロトタイプ

C または C++ で記述した関数で使用するインタフェースは、ソフトウェアインストールディレクトリの SDK¥Include サブディレクトリの dllapi.h および extfnapi.h という 2 つのヘッダファイルで定義されます。これらのヘッダファイルは、外部関数のプロトタイプのプラットフォームに依存する機能を処理します。

関数のプロトタイプ

関数名は、CREATE PROCEDURE または CREATE FUNCTION 文に参照された関数名と一致しなければなりません。次の CREATE FUNCTION 文が実行されたと仮定します。

```
CREATE FUNCTION cover-name ( parameter-list )
  RETURNS data-type
  EXTERNAL NAME 'function-name@library.dll';
```

C/C++ 関数の定義は、次の形式で指定します。

```
#include "dllapi.h"
#include "extfnapi.h"
extern "C"
{
    void _entry function-name( an_extfn_api *api, void *argument_handle )
    {
        // ...
    }
}
```

この関数は、戻り値を返さず、一連のコールバック関数を呼び出すために使用した構造体へのポインタと、SQL プロシージャによって指定された引数へのハンドルを引数として使用します。

例

次の例では、文字列を大文字に変換し、その結果を返す `upstring` という関数を実装します。

```
#include "dllapi.h"
#include "extfnapi.h"
extern "C"
{
    a_sql_uint32 _entry extfn_use_new_api(void)
    {
        return(EXTFN_API_VERSION);
    }
    void _callback extfn_cancel(void *cancel_handle)
    {
        *(short *)cancel_handle = 1;
    }
    void _entry upstring(an_extfn_api *api, void *arg_handle)
    {
        short               result;
        short               canceled;
        an_extfn_value      arg;
        an_extfn_value      retval;
        a_sql_data_type     data_type;
        unsigned            offset;
        char                *string;
        canceled = 0;
        api->set_cancel(arg_handle, &canceled);
        result = api->get_value(arg_handle, 1, &arg);
        if (canceled || result == 0 || arg.data == NULL)
        {
            return; // no parameter or parameter is NULL
        }
        data_type = arg.type & DT_TYPES;
        string = (char *)malloc(arg.len.total_len + 1);
        offset = 0;
        for (; result != 0; )
        {
            if (arg.data == NULL) break;
```

```
            memcpy(&string[offset], arg.data, arg.piece_len);
            offset += arg.piece_len;
            string[offset] = '¥0';
            if (arg.piece_len == 0) break;
            if (canceled) break;
            result = api->get_piece(arg_handle, 1, &arg, offset);
        }
        if (!canceled)
        {
            switch (data_type)
            {
            case DT_NSTRING:
            case DT_NFIXCHAR:
            case DT_NVARCHAR:
            case DT_LONGNVARCHAR:
            case DT_STRING:
            case DT_FIXCHAR:
            case DT_VARCHAR:
            case DT_LONGVARCHAR:
                _strupr_s(string, (size_t)offset + 1);
                retval.type = DT_LONGVARCHAR;
                retval.data = string;
                retval.piece_len = retval.len.total_len =
(a_sql_uint32)strlen(string);
                api->set_value(arg_handle, 0, &retval, 0);
            default:
                break;
            }
        }
        free(string);
        return;
    }
}
```

Windows では、装飾されていない名前を使用してダイナミックリンクライブラリ (DLL) からエントリポイントがエクスポートされるように、この C コードをコンパイルしてリンクする必要があります。通常、これは次の例のようにリンカー EXPORTS ファイルを使用して行います。

```
EXPORTS
extfn_use_new_api
extfn_cancel
upstring
```

`upstring` 関数を呼び出す SQL 文の例を次に示します。

```
CREATE OR REPLACE FUNCTION upstring( IN instr LONG VARCHAR )
RETURNS LONG VARCHAR
EXTERNAL NAME 'upstring@mystring.dll';
SELECT upstring('Hello world!');
```

このセクションの内容:

extfn_use_new_api メソッド [381 ページ]
外部ライブラリが外部関数呼び出しインタフェースを使って記述されていることをデータベースサーバに通知するには、外部ライブラリで extfn_use_new_api 関数をエクスポートします。

extfn_cancel メソッド [382 ページ]
外部ライブラリで取り消し処理がサポートされていることをデータベースサーバに通知するには、外部ライブラリで extfn_cancel 関数をエクスポートします。

extfn_post_load_library メソッド [383 ページ]

extfn_post_load_library 関数が実装され外部ライブラリで公開された場合、外部ライブラリがロードされバージョンチェックが実行された後、この関数はデータベースサーバによって実行されます。その後、外部ライブラリ内の定義されたその他の関数が呼び出されます。

extfn_pre_unload_library メソッド [384 ページ]
extfn_pre_unload_library 関数が実装され外部ライブラリで公開された場合、外部ライブラリをアンロードする前にデータベースサーバによってすぐに実行されます。

an_extfn_api 構造体 [384 ページ]
an_extfn_api 構造体は、呼び出し元の SQL 環境と通信するために使用します。この構造体は、`extfnapi.h` というヘッダファイルで定義されています。このファイルは、ソフトウェアのインストールディレクトリの `SDK¥Include` サブディレクトリにあります。

an_extfn_value 構造体 [387 ページ]
an_extfn_value 構造体は、呼び出し元の SQL 環境からパラメータデータにアクセスするために使用します。

an_extfn_result_set_info 構造体 [388 ページ]
an_extfn_result_set_info 構造体は、結果セットを呼び出し元の SQL 環境に返す際に使用されます。

an_extfn_result_set_column_info 構造体 [389 ページ]
an_extfn_result_set_column_info 構造体は、結果セットを説明するために使用します。

an_extfn_result_set_column_data 構造体 [391 ページ]
an_extfn_result_set_column_data 構造体は、カラムのデータ値を返すために使用します。

1.12.2.1 extfn_use_new_api メソッド

外部ライブラリが外部関数呼び出しインタフェースを使って記述されていることをデータベースサーバに通知するには、外部ライブラリで extfn_use_new_api 関数をエクスポートします。

構文

```
extern "C" a_sql_uint32 SQL_CALLBACK extfn_use_new_api( void );
```

戻り値

関数は符号なし 32 ビット整数値を返します。戻り値は、`extfnapi.h` で定義した EXTFN_API_VERSION のインタフェースバージョン番号です。戻り値が 0 の場合は、古い廃止されたインタフェースが使用されていることを示します。

備考

関数がライブラリによってエクスポートされない場合、データベースサーバは古いインタフェースが使用されていると見なします。UNIX プラットフォームと 64 ビット Windows を含むすべての 64 ビットプラットフォームでは、新しいインタフェースを使用してください。

この関数の一般的な実装を次に示します。

```
extern "C" a_sql_uint32 SQL_CALLBACK extfn_use_new_api( void )
{
    return( EXTFN_API_VERSION );
}
```

関連情報

an_extfn_api 構造体 [384 ページ]

1.12.2.2 extfn_cancel メソッド

外部ライブラリで取り消し処理がサポートされていることをデータベースサーバに通知するには、外部ライブラリで extfn_cancel 関数をエクスポートします。

構文

```
extern "C" void _callback extfn_cancel( void *cancel_handle );
```

パラメータ

cancel_handle

操作する変数へのポインタ。

備考

この関数は、実行中の SQL 文が取り消されるとデータベースサーバによって非同期に呼び出されます。

この関数は cancel_handle を使用して、SQL 文が取り消されたことを外部ライブラリ関数に示すフラグを設定します。

関数がライブラリによってエクスポートされない場合、データベースサーバは取り消し処理がサポートされていないものと見なします。

この関数の一般的な実装を次に示します。

```
#include "dllapi.h"
extern "C"
{
    void _callback extfn_cancel(void *cancel_handle)
    {
        *(short *)cancel_handle = 1;
    }
```

```
}
```

関連情報

an_extfn_api 構造体 [384 ページ]

1.12.2.3 extfn_post_load_library メソッド

extfn_post_load_library 関数が実装され外部ライブラリで公開された場合、外部ライブラリがロードされバージョンチェックが実行された後、この関数はデータベースサーバによって実行されます。その後、外部ライブラリ内の定義されたその他の関数が呼び出されます。

構文

```
extern "C" void _entry extfn_post_load_library( void );
```

備考

この関数は、ライブラリ内のいずれかの関数が呼び出される前にライブラリ全体にわたる設定を行うライブラリ固有の要件がある場合にのみ必要です。

外部ライブラリがロードされ、バージョンチェックが実行された後、この関数はデータベースサーバによって非同期で呼び出されます。その後、外部ライブラリ内の定義されたその他の関数が呼び出されます。

例

```
#include "dllapi.h"
extern "C"
{
    void _entry extfn_post_load_library( void )
    {
        MessageBox(NULL, L"Library loaded", L"Application Notification", MB_OK |
MB_TASKMODAL);
    }
}
```

関連情報

an_extfn_api 構造体 [384 ページ]

1.12.2.4 extfn_pre_unload_library メソッド

extfn_pre_unload_library 関数が実装され外部ライブラリで公開された場合、外部ライブラリをアンロードする前にデータベースサーバによってすぐに実行されます。

構文

```
extern "C" void _entry extfn_pre_unload_library( void );
```

備考

この関数は、ライブラリがアンロードされる前にライブラリ全体にわたるクリーンアップを行うライブラリ固有の要件がある場合にのみ必要です。

この関数は、外部ライブラリをアンロードする直前に、データベースサーバによって非同期で呼び出されます。

例

```
#include "dllapi.h"
extern "C"
{
    void _entry extfn_post_load_library( void )
    {
        MessageBox(NULL, L"Library unloading", L"Application Notification", MB_OK
| MB_TASKMODAL);
    }
}
```

関連情報

an_extfn_api 構造体 [384 ページ]

1.12.2.5 an_extfn_api 構造体

an_extfn_api 構造体は、呼び出し元の SQL 環境と通信するために使用します。この構造体は、`extfnapi.h` というヘッダファイルで定義されています。このファイルは、ソフトウェアのインストールディレクトリの `SDK¥Include` サブディレクトリにあります。

構文

```
typedef struct an_extfn_api {
    short (SQL_CALLBACK *get_value)(
                void *          arg_handle,
```

```
                a_sql_uint32    arg_num,
                an_extfn_value *value
                );
    short (SQL_CALLBACK *get_piece)(
                void *          arg_handle,
                a_sql_uint32    arg_num,
                an_extfn_value *value,
                a_sql_uint32    offset
                );
    short (SQL_CALLBACK *set_value)(
                void *          arg_handle,
                a_sql_uint32    arg_num,
                an_extfn_value *value
                short           append
                );
    void (SQL_CALLBACK *set_cancel)(
                void *          arg_handle,
                void *          cancel_handle
                );
} an_extfn_api;
```

プロパティ

get_value

このコールバック関数を使用して、指定されたパラメータの値を取得します。次の例は、パラメータ 1 の値を取得します。

```
result = api->get_value( arg_handle, 1, &arg )
if( result == 0 || arg.data == NULL )
{
    return; // no parameter or parameter is NULL
}
```

get_piece

このコールバック関数を使用して、指定されたパラメータの値の次のチャンクを取得します (存在する場合)。次の例は、パラメータ 1 の残りの部分を取得します。

```
cmd = (char *)malloc( arg.len.total_len + 1 );
offset = 0;
for( ; result != 0; )
{
     if( arg.data == NULL ) break;
     memcpy( &cmd[offset], arg.data, arg.piece_len );
     offset += arg.piece_len;
     cmd[offset] = '¥0';
     if( arg.piece_len == 0 ) break;
     result = api->get_piece( arg_handle, 1, &arg, offset );
}
```

set_value

このコールバック関数を使用して、指定されたパラメータの値を設定します。次の例は、関数の RETURNS 句の戻り値 (パラメータ 0) を設定します。

```
an_extfn_value      retval;
int ret = -1;
// set up the return value struct
retval.type = DT_INT;
retval.data = (void*) &ret;
```

```
retval.piece_len = retval.len.total_len =
    (a_sql_uint32) sizeof( int );
api->set_value( arg_handle, 0, &retval, 0 );
```

set_cancel

このコールバック関数を使用して、extfn_cancel メソッドによって設定できる変数へのポインタを確立します。次はその例です。

```
short               canceled = 0;
api->set_cancel( arg_handle, &canceled );
```

備考

an_extfn_api 構造体へのポインタは、呼び出し元から外部関数へ渡されます。次に例を示します。

```
#include "dllapi.h"
#include "extfnapi.h"
extern "C"
{
    void _entry upstring(an_extfn_api *api, void *arg_handle)
    {
        short               result;
        short               canceled;
        an_extfn_value      arg;

        canceled = 0;
        api->set_cancel( arg_handle, &canceled );

        result = api->get_value( arg_handle, 1, &arg );
        if( canceled || result == 0 || arg.data == NULL )
        {
            return; // no parameter or parameter is NULL
        }
        .
        .
        .
    }
}
```

任意のコールバック関数を使用する場合、2 番目のパラメータとして外部関数に渡された引数ハンドルを戻す必要があります。

関連情報

an_extfn_value 構造体 [387 ページ]
extfn_cancel メソッド [382 ページ]

1.12.2.6 an_extfn_value 構造体

an_extfn_value 構造体は、呼び出し元の SQL 環境からパラメータデータにアクセスするために使用します。

構文

```
typedef struct an_extfn_value {
    void *           data;
    a_sql_uint32     piece_len;
    union {
        a_sql_uint32 total_len;
        a_sql_uint32 remain_len;
    } len;
    a_sql_data_type  type;
} an_extfn_value;
```

プロパティ

data

このパラメータのデータへのポインタ。

piece_len

パラメータのセグメントの長さ。これは、total_len 以下となります。

total_len

パラメータの合計長。文字列の場合は文字列の長さを表します (NULL ターミネータは含まれません)。このプロパティは、get_value コールバック関数を呼び出した後に設定されます。get_piece コールバック関数を呼び出した後は無効になります。

remain_len

パラメータがセグメント単位で取得された場合の、取得されていない残りの部分の長さとなります。このプロパティは、get_piece コールバック関数の各呼び出しの後に設定されます。

type

パラメータのタイプを示します。これは DT_INT、DT_FIXCHAR、DT_BINARY などの Embedded SQL データ型の 1 つです。

備考

外部関数インタフェースが次の SQL 文を使って記述されていると仮定します。

```
CREATE FUNCTION mystring( IN instr LONG VARCHAR )
RETURNS LONG VARCHAR
EXTERNAL NAME 'mystring@mystring.dll';
```

次のコードフラグメントは、an_extfn_value タイプのオブジェクトのプロパティにアクセスする方法を示したものです。この例では、この関数 (mystring) の入力パラメータ 1 (instr) は SQL LONGVARCHAR 文字列であると予期されています。

```
an_extfn_value      arg;
result = api->get_value( arg_handle, 1, &arg );
if( result == 0 || arg.data == NULL )
{
    return; // no parameter or parameter is NULL
}
if( arg.type != DT_LONGVARCHAR )
{
    return; // unexpected type of parameter
}
cmd = (char *)malloc( arg.len.total_len + 1 );
offset = 0;
for( ; result != 0; )
{
     if( arg.data == NULL ) break;
     memcpy( &cmd[offset], arg.data, arg.piece_len );
     offset += arg.piece_len;
     cmd[offset] = '¥0';
     if( arg.piece_len == 0 ) break;
     result = api->get_piece( arg_handle, 1, &arg, offset );
}
```

関連情報

Embedded SQL のデータ型 [268 ページ]
an_extfn_api 構造体 [384 ページ]

1.12.2.7 an_extfn_result_set_info 構造体

an_extfn_result_set_info 構造体は、結果セットを呼び出し元の SQL 環境に返す際に使用されます。

構文

```
typedef struct an_extfn_result_set_info {
    a_sql_uint32                        number_of_columns;
    an_extfn_result_set_column_info     *column_infos;
    an_extfn_result_set_column_data     *column_data_values;
} an_extfn_result_set_info;
```

プロパティ

number_of_columns

結果セット内のカラム数。

column_infos

結果セットカラムの記述へのリンク。

column_data_values

結果セットカラムデータの記述へのリンク。

備考

次のコードフラグメントは、このタイプのオブジェクトのプロパティを設定する方法を示したものです。

```
int columns = 2;
an_extfn_result_set_info    rs_info;
rs_info.number_of_columns   = columns;
rs_info.column_infos        = col_info;
rs_info.column_data_values  = col_data;
```

関連情報

an_extfn_result_set_column_info 構造体 [389 ページ]
an_extfn_result_set_column_data 構造体 [391 ページ]

1.12.2.8 an_extfn_result_set_column_info 構造体

an_extfn_result_set_column_info 構造体は、結果セットを説明するために使用します。

構文

```
typedef struct an_extfn_result_set_column_info {
    char *                          column_name;
    a_sql_data_type                 column_type;
    a_sql_uint32                    column_width;
    a_sql_uint32                    column_index;
    short int                       column_can_be_null;
} an_extfn_result_set_column_info;
```

プロパティ

column_name

NULL で終了する文字列であるカラム名をポイントします。

column_type

カラムのタイプを示します。これは DT_INT、DT_FIXCHAR、DT_BINARY などの Embedded SQL データ型の 1 つです。

column_width

char(n)、varchar(n) および binary(n) 宣言の最大幅を定義します。その他のデータ型については 0 に設定されています。

column_index

カラムの序数位置 (開始値は 1)。

column_can_be_null

カラムが NULL 入力可の場合は 1、NULL 入力不可の場合は 0 に設定されます。

備考

次のコードフラグメントは、このタイプのオブジェクトのプロパティを設定する方法、および結果セットを呼び出し元の SQL 環境に記述する方法を示すものです。

```
// set up column descriptions
an_extfn_result_set_column_info *col_info =
    (an_extfn_result_set_column_info *)
    malloc( columns * sizeof(an_extfn_result_set_column_info) );
// DepartmentID          INTEGER NOT NULL
col_info[0].column_name  = "DepartmentID";
col_info[0].column_type  = DT_INT;
col_info[0].column_width = 0;
col_info[0].column_index = 1;
col_info[0].column_can_be_null = 0;
// DepartmentName        CHAR(40) NOT NULL
col_info[1].column_name  = "DepartmentName";
col_info[1].column_type  = DT_FIXCHAR;
col_info[1].column_width = 40;
col_info[1].column_index = 2;
col_info[1].column_can_be_null = 0;
api->set_value( arg_handle,
                EXTFN_RESULT_SET_ARG_NUM,
                (an_extfn_value *)&rs_info,
                EXTFN_RESULT_SET_DESCRIBE );
```

関連情報

Embedded SQL のデータ型 [268 ページ]
an_extfn_result_set_info 構造体 [388 ページ]
an_extfn_result_set_column_data 構造体 [391 ページ]

1.12.2.9 an_extfn_result_set_column_data 構造体

an_extfn_result_set_column_data 構造体は、カラムのデータ値を返すために使用します。

構文

```
typedef struct an_extfn_result_set_column_data {
    a_sql_uint32                        column_index;
    void *                              column_data;
    a_sql_uint32                        data_length;
    short                               append;
} an_extfn_result_set_column_data;
```

プロパティ

column_index

カラムの序数位置 (開始値は 1)。

column_data

カラムデータを格納するバッファへのポインタ。

data_length

データの実際の長さ。

append

カラム値をチャンク単位で返すために使用します。カラム値の一部を返す場合は 1、それ以外の場合は 0 に設定します。

備考

次のコードフラグメントは、このタイプのオブジェクトのプロパティを設定する方法、および結果セットのローを呼び出し元の SQL 環境に返す方法を示すものです。

```
int DeptNumber = 400;
char * DeptName = "Marketing";
an_extfn_result_set_column_data *col_data =
    (an_extfn_result_set_column_data *)
    malloc( columns * sizeof(an_extfn_result_set_column_data) );
col_data[0].column_index = 1;
col_data[0].column_data  = &DeptNumber;
col_data[0].data_length  = sizeof( DeptNumber );
col_data[0].append  = 0;
col_data[1].column_index = 2;
col_data[1].column_data  = DeptName;
col_data[1].data_length  = strlen(DeptName);
col_data[1].append  = 0;
api->set_value(  arg_handle,
                 EXTFN_RESULT_SET_ARG_NUM,
                 (an_extfn_value *)&rs_info,
                 EXTFN_RESULT_SET_NEW_ROW_FLUSH );
```

関連情報

an_extfn_result_set_info 構造体 [388 ページ]
an_extfn_result_set_column_info 構造体 [389 ページ]

1.12.3 外部関数呼び出しインタフェースのメソッド

外部関数呼び出しインタフェースのメソッドは、データベースサーバにデータを送信、またはデータベースサーバからデータを取得する場合に使用します。

get_value コールバック

```
short (SQL_CALLBACK *get_value)
(
    void *          arg_handle,
    a_sql_uint32    arg_num,
    an_extfn_value *value
);
```

get_value コールバック関数は、外部関数とのインタフェースとして動作するストアドプロシージャまたは関数に渡されたパラメータの値を取得するために使用します。失敗した場合は 0 を返し、それ以外の場合は 0 以外の結果を返します。get_value を呼び出した後、an_extfn_value 構造体の total_len フィールドには値全体の長さが入ります。piece_len フィールドには、get_value を呼び出して取得された部分の長さが入ります。piece_len フィールドは必ず total_len 以下の値になります。piece_len が total_len よりも小さい場合は、2 番目の関数 get_piece を呼び出して残りの部分を取得できます。total_len フィールドが有効になるのは、最初の get_value が呼び出された後です。このフィールドは、get_piece の呼び出しによって変更される remain_len フィールドでオーバーレイされます。したがって、total_len フィールドの値を後で使用する場合は、get_value を呼び出した直後にこのフィールドの値を保存しておく必要があります。

get_piece コールバック

```
short (SQL_CALLBACK *get_piece)
(
    void *          arg_handle,
    a_sql_uint32    arg_num,
    an_extfn_value *value,
    a_sql_uint32    offset
);
```

パラメータ全体の値を一度に返せない場合は、get_piece 関数を繰り返し呼び出して、パラメータ値の残りの部分を取得できます。

get_value と get_piece の両方を呼び出して返されたすべての piece_len 値の合計が、get_value を呼び出した後で total_len フィールドに返された初期値に加算されます。get_piece の呼び出し後、total_len をオーバーレイする remain_len フィールドには未取得分の長さ (値) が示されます。

get_value コールバックおよび get_piece コールバックの使用

次に示すのは、get_value と get_piece を使用して、LONG VARCHAR パラメータなどの文字列パラメータの値を取得する例です。

外部関数のラッパーが次のように宣言されているとします。

```
CREATE PROCEDURE mystring( IN instr LONG VARCHAR )
  EXTERNAL NAME 'mystring@mystring.dll';
```

外部関数を SQL から呼び出すには、次のような文を使用します。

```
CALL mystring('Hello world!');
```

C で記述された mystring 関数を Windows オペレーティングシステムに実装する例を次に示します。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>
#include "dllapi.h"
#include "extfnapi.h"
BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved )
{
    return TRUE;
}
extern "C"
{
    a_sql_uint32 _entry extfn_use_new_api( void )
    {
        return( EXTFN_API_VERSION );
    }
    void _entry mystring( an_extfn_api *api, void *arg_handle )
    {
        short               result;
        an_extfn_value      arg;
        unsigned            offset;
        char                *string;
        result = api->get_value( arg_handle, 1, &amp;arg );
        if( result == 0 || arg.data == NULL )
        {
            return; // no parameter or parameter is NULL
        }
        string = (char *)malloc( arg.len.total_len + 1 );
        offset = 0;
        for( ; result != 0; ) {
             if( arg.data == NULL ) break;
             memcpy( &amp;string[offset], arg.data, arg.piece_len );
             offset += arg.piece_len;
             string[offset] = '¥0';
             if( arg.piece_len == 0 ) break;
             result = api->get_piece( arg_handle, 1, &amp;arg, offset );
        }
        MessageBoxA( NULL, string,
              "Application Notification",
              MB_OK | MB_TASKMODAL );
        free( string );
        return;
    }
}
```

set_value コールバック

```
short (SQL_CALLBACK *set_value)
(
    void *          arg_handle,
    a_sql_uint32    arg_num,
    an_extfn_value *value
    short           append
);
```

set_value コールバック関数は、OUT パラメータおよびストアド関数の RETURNS 結果の値を設定するために使用します。RETURNS 値を設定するには、arg_num 値を 0 にします。次はその例です。

```
an_extfn_value      retval;
retval.type = DT_LONGVARCHAR;
retval.data = result;
retval.piece_len = retval.len.total_len = (a_sql_uint32) strlen( result );
api->set_value( arg_handle, 0, &retval, 0 );
```

set_value の append 引数は、指定されたデータを既存のデータと置き換えるか (false)、または既存のデータに追加するか (true) を指定します。`append=FALSE` を指定して set_value を呼び出してから、同じ引数に対して `append=TRUE` を指定して呼び出してください。固定長データ型の場合、append 引数は無視されます。

NULL を返すには、an_extfn_value 構造体の data フィールドを NULL に設定します。

set_cancel コールバック

```
void (SQL_CALLBACK *set_cancel)
(
    void *arg_handle,
    void *cancel_handle
);
```

外部関数では、IN または INOUT パラメータの値を取得し、OUT パラメータおよびストアド関数の RETURNS 結果の値を設定することができます。ただし、取得したパラメータ値が無効になっていたり、値を設定する必要がなくなっている場合があります。これは、SQL 文の実行が取り消された場合に発生します。また、アプリケーションがデータベースサーバから不意に切断された場合に発生することもあります。この問題を処理するには、extfn_cancel という特別なエントリポイントをライブラリに定義します。この関数が定義されていると、サーバは実行中の SQL 文が取り消される場合に必ずこの関数を呼び出します。

extfn_cancel 関数は、適切と思われるさまざまな方法で使用できるハンドルを指定して呼び出されます。一般的にこのハンドルは、SQL 文の呼び出しが取り消されたことを示すフラグを間接的に設定する場合に使用します。

渡されるハンドルの値は、set_cancel コールバック関数を使用して、外部ライブラリの関数で設定することができます。例:

```
void _callback extfn_cancel( void *cancel_handle )
{
    *(short *)cancel_handle = 1;
}

void _entry mystring( an_extfn_api *api, void *arg_handle )
{
.
.
.
```

```
    short  canceled = 0;
    api->set_cancel( arg_handle, &canceled );
.
.
.
    if( canceled )
```

静的なグローバル "canceled" 変数を設定することは不適切です。これは、通常はありえない、すべての接続のすべての SQL 文が取り消されると誤って解釈されることがあるためです。そのために、set_cancel コールバック関数が提供されています。set_cancel を呼び出す前に、"canceled" 変数を必ず初期化してください。

外部関数の重要ポイントにある "canceled" 変数の設定を確認しておく必要があります。たとえば、get_value や set_value などの外部ライブラリ呼び出しインタフェース関数の呼び出し前や後などが重要ポイントです。extfn_cancel が呼び出された結果、変数が設定されると、外部関数は適切な終了アクションを実行することができます。前述の例に基づいたコードフラグメントを次に示します。

```
if( canceled )
{
    free( string );
    return;
}
```

注記

任意の引数の get_piece 関数は、同じ引数の get_value 関数の直後にのみ呼び出すことができます。

OUT パラメータで get_value を呼び出すと、an_extfn_value 構造体の type フィールドに引数のデータ型が設定されて返され、an_extfn_value 構造体の data フィールドに NULL が設定されて返されます。

データベースサーバソフトウェアの `SDK¥Include` フォルダのヘッダファイル `extfnapi.h` には、追加の注記がいくつか含まれます。

次の表に、an_extfn_api に定義されている関数が false を返す条件を示します。

関数	true の場合は 0 を返し、それ以外の場合は 1 を返す条件
get_value()	• arg_num が無効な場合。たとえば、arg_num が外部関数の引数の数より大きい場合。
get_piece()	• arg_num が無効な場合。たとえば、arg_num が前に呼び出された get_value で使用した引数番号に対応していない場合。 • オフセットが arg_num 引数の値の合計長より大きい場合。 • get_value が呼び出される前に呼び出された場合。
set_value()	• arg_num が無効な場合。たとえば、arg_num が外部関数の引数の数より大きい場合。 • 引数 arg_num が入力専用の場合。 • 指定された値の型が引数 arg_num の型と一致しない場合。

1.12.4 データ型の処理

ほとんどの SQL データ型を外部ライブラリに渡すことができます。

データ型

外部ライブラリに渡されるのは、次に示す SQL データ型です。

SQL データ型	sqldef.h	C データ型
CHAR	DT_FIXCHAR	指定された長さの文字データ
VARCHAR	DT_VARCHAR	指定された長さの文字データ
LONG VARCHAR、TEXT	DT_LONGVARCHAR	指定された長さの文字データ
UNIQUEIDENTIFIERSTR	DT_FIXCHAR	指定された長さの文字データ
XML	DT_LONGVARCHAR	指定された長さの文字データ
NCHAR	DT_NFIXCHAR	指定された長さの UTF-8 文字データ
NVARCHAR	DT_NVARCHAR	指定された長さの UTF-8 文字データ
LONG NVARCHAR、NTEXT	DT_LONGNVARCHAR	指定された長さの UTF-8 文字データ
UNIQUEIDENTIFIER	DT_BINARY	16 バイト長のバイナリデータ
BINARY	DT_BINARY	指定された長さのバイナリデータ
VARBINARY	DT_BINARY	指定された長さのバイナリデータ
LONG BINARY	DT_LONGBINARY	指定された長さのバイナリデータ
TINYINT	DT_TINYINT	1 バイト整数
[UNSIGNED] SMALLINT	DT_SMALLINT、DT_UNSSMALLINT	[符号なし] 2 バイト整数
[UNSIGNED] INT	DT_INT、DT_UNSINT	[符号なし] 4 バイト整数
[UNSIGNED] BIGINT	DT_BIGINT、DT_UNSBIGINT	[符号なし] 8 バイト整数
REAL、FLOAT(1-24)	DT_FLOAT	単精度浮動小数点数
DOUBLE、FLOAT(25-53)	DT_DOUBLE	倍精度浮動小数点数

日付データ型または時刻データ型は使用できません。また、DECIMAL または NUMERIC データ型 (通貨データ型を含む) も使用できません。

INOUT または OUT パラメータに値を指定するには、set_value 関数を使用します。IN と INOUT パラメータを読み取るには、get_value 関数を使用します。

パラメータのデータ型の判別

get_value を呼び出した後で、an_extfn_value 構造体の type フィールドを使用して、パラメータのデータ型情報を取得できます。次のサンプルコードは、パラメータのデータ型を識別する方法を示します。

```
an_extfn_value      arg;
a_sql_data_type     data_type;
api->get_value( arg_handle, 1, &arg );
data_type = arg.type & DT_TYPES;
switch( data_type )
{
case DT_FIXCHAR:
case DT_VARCHAR:
case DT_LONGVARCHAR:
    break;
default:
    return;
}
```

UTF-8 データ型

NCHAR、NVARCHAR、LONG NVARCHAR、NTEXT などの UTF-8 データ型は、UTF-8 でエンコードされた文字列として渡されます。Windows MultiByteToWideChar 関数などの関数を使用して、UTF-8 文字列をワイド文字列 (Unicode) に変換できます。

NULL を渡す

すべての引数に有効な値として NULL を渡すことができます。外部ライブラリの関数は、すべてのデータ型の戻り値として NULL を渡すことができます。

戻り値

外部関数に戻り値を設定するには、arg_num パラメータ値に 0 を指定して set_value 関数を呼び出します。arg_num を 0 に設定せずに set_value を呼び出すと、関数の結果は NULL になります。

ストアド関数呼び出しの戻り値のデータ型を設定する必要もあります。次のコードフラグメントは、戻り値のデータ型を設定する方法を示します。

```
an_extfn_value      retval;
retval.type = DT_LONGVARCHAR;
retval.data = result;
retval.piece_len = retval.len.total_len = (a_sql_uint32) strlen( result );
api->set_value( arg_handle, 0, &retval, 0 );
```

関連情報

Embedded SQL のホスト変数 [272 ページ]

1.12.5 外部ライブラリをアンロードする方法

システムプロシージャ sa_external_library_unload を使用して、ライブラリが使用されていないときに外部ライブラリをアンロードすることができます。

このプロシージャには、オプションのパラメータ LONG VARCHAR を 1 つ指定します。このパラメータは、ライブラリをアンロードするように指定し、ライブラリのロードで使用されるファイルパス文字列と一致する必要があります。パラメータが指定されていない場合、使用されていないすべての外部ライブラリがアンロードされます。

次に示すのは、外部関数ライブラリをアンロードする例です。

```
CALL sa_external_library_unload( 'library.dll' );
```

一連の外部関数を開発するときにこの関数を使用すると、新しいバージョンのライブラリをインストールするためにデータベースサーバを停止する必要がないので便利です。

次の例は、このシステムプロシージャを使用する方法を示します。

```
CREATE OR REPLACE PROCEDURE mystring( IN instr LONG VARCHAR )
EXTERNAL NAME 'mystring@c:¥¥projects¥¥mystring¥¥debug¥¥mystring.dll';
CALL mystring('Hello world!');
CALL sa_external_library_unload( 'c:¥¥projects¥¥mystring¥¥debug¥¥mystring.dll' );
```

1.13 外部環境のサポート

7 つの外部ランタイム環境がサポートされています。これには、C/C++ で記述された Embedded SQL と ODBC アプリケーション、Java、JavaScript、Perl、PHP、または Microsoft .NET Framework Common Language Runtime (CLR) に基づく Microsoft C# や Microsoft Visual Basic などの言語で記述されたアプリケーションが含まれます。

データベースサーバによってロードされたダイナミックリンクライブラリまたは共有オブジェクトを独自のアドレス空間にロードできますが、この場合のリスクは、ネイティブ関数で障害が発生した場合、データベースサーバがクラッシュすることです。データベースサーバの外部環境でコンパイル済みネイティブ関数を実行できると、サーバへのこれらのリスクをなくすことができます。

次に示すのは、外部環境サポートの概要です。

- START EXTERNAL ENVIRONMENT 文と STOP EXTERNAL ENVIRONMENT 文は、外部環境を要求に応じて起動または停止するために使用されます。外部環境は必要に応じて自動的に起動または停止するので、これらの文はオプションです。
- ALTER EXTERNAL ENVIRONMENT 文は、外部環境の場所を設定または変更するために使用されます。
- COMMENT ON EXTERNAL ENVIRONMENT 文は、外部環境のコメントを追加するために使用されます。
- データベースサーバで使用するように外部環境が設定されたら、オブジェクトをデータベースにインストールし、それらのオブジェクトを外部環境内で使用するストアドプロシージャおよび関数を作成することができます。

- INSTALL EXTERNAL OBJECT 文は、JavaScript、Perl、または PHP 外部オブジェクト (Perl スクリプトなど) をファイルまたは式からデータベースにインストールするために使用されます。データベースにインストールされた外部オブジェクトは、外部ストアドプロシージャおよび関数の定義で使用できます。
- COMMENT ON EXTERNAL ENVIRONMENT OBJECT 文は、外部環境オブジェクトのコメントを追加するために使用されます。
- インストールされている JavaScript、Perl、PHP 外部オブジェクトをデータベースから削除するには、次に示す REMOVE EXTERNAL OBJECT 文を使用します。
- CREATE PROCEDURE 文と CREATE FUNCTION 文は、外部ストアドプロシージャおよび関数の定義を作成するために使用されます。これらはデータベース内の他のストアドプロシージャまたは関数と同様に使用できます。外部環境のストアドプロシージャまたはファンクションが呼び出されるると、データベースサーバは外部環境がまだ起動されていない場合は、これを自動的に起動し、外部環境がデータベースから外部オブジェクトをフェッチして実行するために必要なあらゆる情報を送信します。実行結果の結果セットや戻り値は、必要に応じて返されます。

このセクションの内容:

CLR 外部環境 [399 ページ]
CLR のストアドプロシージャおよび関数を SQL ストアドプロシージャと同様にデータベースから呼び出すことができます。

ESQL 外部環境と ODBC 外部環境 [403 ページ]
Embedded SQL または ODBC を使用する外部コンパイル済みネイティブ関数は、SQL ストアドプロシージャと同じ方法でデータベースから呼び出すことができます。

Java 外部環境 [411 ページ]
Java メソッドを SQL ストアドプロシージャと同様にデータベースから呼び出すことができます。

JavaScript 外部環境 [416 ページ]
JavaScript のストアドプロシージャおよび関数を SQL ストアドプロシージャと同様にデータベースから呼び出すことができます。

Perl 外部環境 [421 ページ]
Perl のストアドプロシージャおよび関数を SQL ストアドプロシージャと同様にデータベースから呼び出すことができます。

PHP 外部環境 [425 ページ]
PHP のストアドプロシージャおよび関数を SQL ストアドプロシージャと同様にデータベースから呼び出すことができます。

1.13.1 CLR 外部環境

CLR のストアドプロシージャおよび関数を SQL ストアドプロシージャと同様にデータベースから呼び出すことができます。

CLR ストアドプロシージャまたはファンクションの動作は、SQL ストアドプロシージャまたはファンクションと同じです。ただし、プロシージャまたはファンクションのコードは Microsoft C# または Visual Basic などの Microsoft .NET 言語で記述され、その実行はデータベースサーバの外側 (つまり別の Microsoft .NET 実行ファイル内) で行われます。

この Microsoft .NET 実行ファイルのインスタンスは、データベースごとに 1 つだけです。CLR 関数およびストアドプロシージャを実行するすべての接続が、同じ Microsoft .NET 実行インスタンスを使用します。ただし、ネームスペースは接続ごとに異なります。静的変数は接続の間持続しますが、接続間で共有することはできません。

デフォルトでは、データベースサーバで、データベースサーバ上で動作しているデータベースごとに 1 つの CLR 外部環境が使用されていました。-sclr コマンドラインオプションまたは SingleCLRInstanceVersion データベースサーバオプションを使用すると、データベースサーバで実行されているすべてのデータベースで 1 つの CLR 外部環境を使用するように要求できます。CLR 外部環境をどのデータベースで開始する前に、このオプションを指定する必要があります。

外部 CLR 関数またはプロシージャを呼び出すには、ロードする DLL およびアセンブリ内で呼び出す関数を定義する EXTERNAL NAME 文字列を指定して、対応するストアドプロシージャまたはファンクションを定義します。ストアドプロシージャまたはファンクションを定義する際は、LANGUAGE CLR も指定する必要があります。次に、宣言の例を示します。

```
CREATE PROCEDURE clr_stored_proc(
    IN p1 INT,
    IN p2 UNSIGNED SMALLINT,
    OUT p3 LONG VARCHAR)
EXTERNAL NAME 'MyCLRTest.dll::MyCLRTest.Run( int, ushort, out string )'
LANGUAGE CLR;
```

この例では、`clr_stored_proc` というストアドプロシージャを実行すると、DLL `MyCLRTest.dll` をロードし、関数 MyCLRTest.Run を呼び出します。clr_stored_proc プロシージャは 3 つの SQL パラメータを受け取ります。そのうち 2 つはそれぞれ INT 型と UNSIGNED SMALLINT 型の IN パラメータで、もう 1 つは LONG VARCHAR 型の OUT パラメータです。Microsoft .NET 側で、この 3 つのパラメータは int 型と ushort 型の入力引数、および string 型の出力引数に変換されます。out 引数のほかに、CLR 関数では ref 引数も使用できます。対応するストアドプロシージャに INOUT パラメータがある場合、ユーザは ref CLR 引数を宣言する必要があります。

次の表は、CLR 引数の各種データ型と、それに対応する推奨される SQL データ型を示します。

CLR のデータ型	推奨される SQL データ型
bool	bit
byte	tinyint
short	smallint
ushort	unsigned smallint
int	int
uint	unsigned int
long	bigint
ulong	unsigned bigint
decimal	numeric
float	real
double	double
DateTime	timestamp
string	long varchar
byte[]	long binary

DLL の宣言では、相対パスまたは絶対パスのどちらかを指定できます。相対パスが指定された場合、外部 Microsoft .NET 実行ファイルはそのパスだけでなく、それ以外の場所についても DLL を検索します。ただし、グローバルアセンブリキャッシュ (GAC) では DLL を検索しません。

既存の Java ストアドプロシージャおよび関数と同様に、CLR ストアドプロシージャおよび関数もサーバ側の要求をデータベースに戻して、結果セットを返すことができます。また、Java と同じように、Console.Out および Console.Error に出力される情報は、すべてデータベースサーバメッセージウィンドウに自動的にリダイレクトされます。

サーバ側の要求の作成方法、および CLR 関数またはストアドプロシージャから結果セットを返す方法の詳細については、*%SQLANYSAMP17%*¥SQLAnywhere¥ExternalEnvironments¥CLR ディレクトリのサンプルを参照してください。

CLR をデータベースで使用するには、データベースサーバが CLR 外部環境実行ファイルを検出して開始できることを確認してください。この実行ファイルには複数のバージョンがあります。

表 1: サポートされている .NET CLR バージョン

.NET バージョン	ファイル名
2.0/3.5	dbextclr[VER_MAJOR]
4.x	dbextclr[VER_MAJOR]_v4.5

選択した CLR 外部環境モジュールが ALTER EXTERNAL ENVIRONMENT 文で識別されていることを確認します。次はその例です。

```
ALTER EXTERNAL ENVIRONMENT CLR LOCATION 'dbextclr[VER_MAJOR]_v4.5'
```

ソフトウェアの新しいバージョンに移植できるように、現在のソフトウェアリリースバージョン番号ではなく [VER_MAJOR] を LOCATION 文字列に使用することをおすすめします。データベースサーバによって [VER_MAJOR] が適切なバージョン番号に置き換えられます。

使用している .NET バージョンに対応するファイルが *%SQLANY17%*¥Bin32 または *%SQLANY17%*¥Bin64 フォルダにあることを確認してください。デフォルトでは、64 ビットデータベースサーバは 64 ビットバージョンのモジュールを使用し、32 ビットデータベースサーバは 32 ビットバージョンのモジュールを使用します。理想としては、.NET アプリケーションはすべての CPU プラットフォームを対象にする必要があります。

データベースサーバが、選択した CLR 外部環境実行ファイルを検出して開始できるかどうかを確認するには、次の文を実行します。

```
START EXTERNAL ENVIRONMENT CLR;
```

データベースサーバが CLR を開始できない場合は、データベースサーバが CLR 外部環境実行ファイルを検出できない可能性があります。SACIrClassLoader の動作が停止したというメッセージが表示された場合は、SetupVSPackage を実行して現在のバージョンの .NET データプロバイダをインストールします。

START EXTERNAL ENVIRONMENT CLR 文は、データベースサーバが CLR 実行ファイルを開始できるかどうかを確認する以外で使用することはありません。通常、CLR ストアドプロシージャまたはファンクションを呼び出すと、CLR は自動的に開始されます。

これと同様に、CLR のインスタンスを停止するために STOP EXTERNAL ENVIRONMENT CLR 文を使用する必要もありません。インスタンスは、接続が切断されると自動的に停止します。STOP EXTERNAL ENVIRONMENT CLR 文は、接続用に CLR インスタンスを解放します。これを実行する理由はいくつかあります。

- CLR を完了しており、一部のリソースを解放したい場合。
- ALTER EXTERNAL ENVIRONMENT 文を使用して CLR バージョンを変更し、CLR インスタンスが現在実行中の場合。
- CLR によってロードされたアセンブリを再ロード/再起動したい場合。

Perl、PHP、Java 外部環境とは異なり、CLR 環境ではデータベースに何もインストールする必要がありません。したがって、CLR 外部環境を使用する前に INSTALL 文を実行する必要がありません。

次の例に示す C# で記述された関数は、外部環境で実行できます。

```
public class StaticTest
{
    private static int val = 0;
    public static int GetValue() {
        val += 1;
        return val;
    }
}
```

この関数をダイナミックリンクライブラリにコンパイルすると、外部環境から呼び出すことができます。`dbextclr17.exe` という実行イメージファイルがデータベースサーバによって開始され、この実行イメージファイルがダイナミックリンクライブラリをロードします。この実行ファイルについては、複数の異なるバージョンが提供されています。たとえば Windows では、32 ビットと 64 ビットの実行ファイルがあります。1 つは 32 ビットバージョンのデータベースサーバ用、もう 1 つは 64 ビットバージョンのデータベースサーバ用です。

Microsoft C# コンパイラを使用して、このアプリケーションをダイナミックリンクライブラリに構築するには、次のようなコマンドを使用します。上の例のソースコードは、`StaticTest.cs` というファイルにあるものと仮定しています。

```
csc /target:library /out:clrtest.dll StaticTest.cs
```

このコマンドは、コンパイル済みのコードを `clrtest.dll` という DLL に置きます。コンパイル済みの Microsoft C# 関数 GetValue を呼び出すには、Interactive SQL を使用して、ラッパーを次のように定義します。

```
CREATE FUNCTION stc_get_value()
RETURNS INT
EXTERNAL NAME 'clrtest.dll::StaticTest.GetValue() int'
LANGUAGE CLR;
```

CLR では、EXTERNAL NAME 文字列は 1 行の SQL 文で指定されます。場合によっては、EXTERNAL NAME 文字列に DLL のパスを含めて、DLL を検出できるようにする必要があります。依存アセンブリの場合 (たとえば、`myLib.dll` に `myOtherLib.dll` 内の関数を呼び出すコードがある、または何らかの形で前者が後者に依存する場合) は、.NET Framework によって依存性がロードされます。指定されたアセンブリは CLR 外部環境によってロードされますが、依存アセンブリが確実にロードされるようにするためには追加の手順が必要になる場合があります。解決策としては、Microsoft .NET Framework にインストールされている Microsoft gacutil ユーティリティを使用してすべての依存性をグローバルアセンブリキャッシュ (GAC) に登録するという方法があります。カスタム開発のライブラリを使用する場合、gacutil を使用して GAC に登録する前に、厳密な名前キーでライブラリが署名してある必要があります。

サンプルのコンパイル済み C# 関数を実行するには、次の文を実行します。

```
SELECT stc_get_value();
```

Microsoft C# 関数が呼び出されるたびに、整数値の結果が新しく生成されます。返される値は、1、2、3、の順に続きます。

データベースでの CLR サポートの使用に関する詳細および例については、*%SQLANYSAMP17%*`¥SQLAnywhere¥ExternalEnvironments¥CLR` ディレクトリのサンプルを参照してください。

1.13.2 ESQL 外部環境と ODBC 外部環境

Embedded SQL または ODBC を使用する外部コンパイル済みネイティブ関数は、SQL ストアドプロシージャと同じ方法でデータベースから呼び出すことができます。

データベースサーバソフトウェアでは、コンパイル済み C または C++ コードを含むダイナミックリンクライブラリまたは共有オブジェクトをアドレス空間にロードし、このライブラリで以降の関数を呼び出すことができます。ネイティブ関数を呼び出すことができれば効率は良くなりますが、ネイティブ関数が誤動作した場合は、重大な結果を招きかねません。特に、ネイティブ関数が無限ループに入った場合は、データベースサーバがハングする可能性があります。また、ネイティブ関数が原因で障害が発生した場合は、データベースサーバがクラッシュする可能性があります。

このため、データベースサーバのアドレス空間の外 (外部環境) でコンパイル済みネイティブ関数を呼び出したほうが良いでしょう。コンパイル済みネイティブ関数を外部環境で実行することには、次のような大きなメリットがあります。

- コンパイル済みネイティブ関数が誤作動した場合でも、データベースサーバはハングまたはクラッシュしません。
- ODBC、Embedded SQL (ESQL)、または C API を使用するようネイティブ関数を作成可能で、データベースサーバに接続せずにサーバ側の呼び出しをデータベースサーバに戻すことができます。
- ネイティブ関数は結果セットをデータベースサーバに返すことができます。
- 外部環境では、32 ビットのデータベースサーバが 64 ビットのコンパイル済みネイティブ関数と通信できます。また、その逆も可能です。コンパイル済みネイティブ関数がデータベースサーバのアドレス空間に直接ロードされた場合、これは不可能です。32 ビットのライブラリは 32 ビットのサーバ、64 ビットのライブラリは 64 ビットのサーバでしかロードできません。

コンパイル済みネイティブ関数を外部環境で実行した場合、データベースサーバ内で実行した場合よりも多少パフォーマンスが低下します。

また、ネイティブ関数と情報の受け渡しをするためには、コンパイル済みネイティブ関数は外部呼び出しインタフェースを使用する必要があります。

コンパイル済みネイティブ C 関数をデータベースサーバ内でなく外部環境で実行するには、ストアドプロシージャまたは関数を EXTERNAL NAME 句で定義し、後続の LANGUAGE 属性で C_ESQL32、C_ESQL64、C_ODBC32、C_ODBC64 のいずれか 1 つを指定します。

Perl、PHP、JavaScript、および Java の外部環境とは異なり、データベースにソースコードやコンパイル済みオブジェクトはインストールしません。したがって、ESQL および ODBC の外部環境を使用する前に INSTALL 文を実行する必要がありません。

次の例に示す C++ で記述された関数は、データベースサーバ内でも外部環境内でも実行できます。

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}
// Note: extfn_use_new_api used only for
// execution in the database server
extern "C" __declspec( dllexport )
a_sql_uint32 extfn_use_new_api( void )
{
```

```
    return( EXTFN_API_VERSION );
}

extern "C" __declspec( dllexport )
void SimpleCFunction(
    an_extfn_api *api,
    void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;
    int *               intptr;
    int                 i, j, k;

    j = 1000;
    k = 0;
    for( i = 1; i <= 4; i++ )
    {
        result = api->get_value( arg_handle, i, &arg );
        if( result == 0 || arg.data == NULL ) break;
        if( arg.type & DT_TYPES != DT_INT ) break;
        intptr = (int *) arg.data;
        k += *intptr * j;
        j = j / 10;
    }
    retval.type = DT_INT;
    retval.data = (void*)&k;
    retval.piece_len = retval.len.total_len =
       (a_sql_uint32) sizeof( int );
    api->set_value( arg_handle, 0, &retval, 0 );
    return;
}
```

この関数をダイナミックリンクライブラリまたは共有オブジェクトにコンパイルすると、外部環境から呼び出すことができます。dbexternc*17* という実行イメージファイルがデータベースサーバによって開始され、この実行イメージファイルがダイナミックリンクライブラリまたは共有オブジェクトをロードします。この実行ファイルについては、複数の異なるバージョンが提供されています。たとえば Windows では、32 ビットと 64 ビットの実行ファイルがあります。

32 ビットまたは 64 ビットバージョンのデータベースサーバを使用でき、どちらのバージョンのデータベースサーバでも 32 ビットまたは 64 ビットバージョンの dbexternc*17* を開始できます。これは、外部環境を使用するメリットの 1 つです。データベースサーバによって開始された dbexternc*17* は、接続が切断されるか STOP EXTERNAL ENVIRONMENT 文が (正しい環境名で) 実行されるまで終了しません。外部環境呼び出しを実行する接続には、dbexternc*17* のコピーがそれぞれ与えられます。

コンパイル済みネイティブ関数 SimpleCFunction を呼び出すには、次のようにラッパーを定義します。

```
CREATE FUNCTION SimpleCDemo(
  IN arg1 INT,
  IN arg2 INT,
  IN arg3 INT,
  IN arg4 INT )
RETURNS INT
EXTERNAL NAME 'SimpleCFunction@c:¥¥c¥¥extdemo.dll'
LANGUAGE C_ODBC32;
```

これは、コンパイル済みネイティブ関数をデータベースサーバのアドレス空間にロードする場合の記述方法と、ほとんど同じです。ただ 1 つ異なるのは、LANGUAGE C_ODBC32 句を使用することです。この句は、SimpleCDemo が外部環境で実行される関数であり、32 ビットの ODBC 呼び出しを使用することを指定しています。C_ESQL32、C_ESQL64、C_ODBC32、C_ODBC64 の言語の指定は、サーバ側の要求を作成するときに、外部 C 関数で 32 ビットまたは 64 ビットの ODBC 呼び出し、ESQL 呼び出し、または C API 呼び出しのどれを行うのかをデータベースサーバに知らせます。

サーバ側の要求を作成する際にネイティブ関数が ODBC 呼び出し、ESQL 呼び出し、C API 呼び出しのいずれも使用しない場合は、32 ビットのアプリケーションには C_ODBC32 または C_ESQL32 を、64 ビットのアプリケーションには C_ODBC64 または C_ESQL64 を使用できます。上記の外部 C 関数はこれに該当します。この関数はこれらの API を一切使用しません。

サンプルのコンパイル済みネイティブ関数を実行するには、次の文を実行します。

```
SELECT SimpleCDemo(1,2,3,4);
```

サーバ側の ODBC を使用するには、C/C++ コードでデフォルトのデータベース接続を使用する必要があります。データベース接続のハンドルを取得するには、EXTFN_CONNECTION_HANDLE_ARG_NUM 引数を指定して get_value を呼び出します。この引数は、新しい外部環境接続を開くのではなく、現在の外部環境接続を返すようにデータベースサーバに伝えます。

```
#include <windows.h>
#include <stdio.h>
#include "odbc.h"
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}

extern "C" __declspec( dllexport )
void ServerSideFunction( an_extfn_api *api, void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;
    SQLRETURN           ret;
    ret = -1;
    // set up the return value struct
    retval.type = DT_INT;
    retval.data = (void*) &ret;
    retval.piece_len = retval.len.total_len =
        (a_sql_uint32) sizeof( int );
    result = api->get_value( arg_handle,
                    EXTFN_CONNECTION_HANDLE_ARG_NUM,
                    &arg );
    if( result == 0 || arg.data == NULL )
    {
        api->set_value( arg_handle, 0, &retval, 0 );
        return;
    }
    HDBC dbc = (HDBC)arg.data;
    HSTMT stmt = SQL_NULL_HSTMT;
    ret = SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
    if( ret != SQL_SUCCESS ) return;
    ret = SQLExecDirect( stmt,
            (SQLCHAR *) "INSERT INTO odbcTab "
                "SELECT table_id, table_name "
                "FROM SYS.SYSTAB", SQL_NTS );
    if( ret == SQL_SUCCESS )
    {
        SQLExecDirect( stmt,
            (SQLCHAR *) "COMMIT", SQL_NTS );
    }
    SQLFreeHandle( SQL_HANDLE_STMT, stmt );

    api->set_value( arg_handle, 0, &retval, 0 );
```

```
    return;
}
```

上記の ODBC コードが `extodbc.cpp` ファイルに保存されている場合、次のコマンドを使用して Windows 用に構築できます (データベースサーバソフトウェアがフォルダ `c:¥sa17` にインストールされており、Microsoft Visual C++ もインストールされていることが前提です)。

```
cl extodbc.cpp /LD /Ic:¥sa17¥sdk¥include odbc32.lib
```

次の例では、テーブルを作成し、コンパイル済みネイティブ関数を呼び出すストアドプロシージャのラッパーを定義してから、ネイティブ関数を呼び出してテーブルにデータを移植します。

```
CREATE TABLE odbcTab(c1 int, c2 char(128));
CREATE FUNCTION ServerSideODBC( )
RETURNS INT
EXTERNAL NAME 'ServerSideFunction@extodbc.dll'
LANGUAGE C_ODBC32;
SELECT ServerSideODBC();
// The following statement should return two identical rows
SELECT COUNT(*) FROM odbcTab
UNION ALL
SELECT COUNT(*) FROM SYS.SYSTAB;
```

同様に、サーバ側の ESQL を使用するには、C/C++ コードでデフォルトのデータベース接続を使用する必要があります。データベース接続のハンドルを取得するには、EXTFN_CONNECTION_HANDLE_ARG_NUM 引数を指定して get_value を呼び出します。この引数は、新しい外部環境接続を開くのではなく、現在の外部環境接続を返すようにデータベースサーバに伝えます。

```
#include <windows.h>
#include <stdio.h>
#include "sqlca.h"
#include "sqlda.h"
#include "extfnapi.h"
BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}
EXEC SQL INCLUDE SQLCA;
static SQLCA *_sqlc;
EXEC SQL SET SQLCA "_sqlc";
EXEC SQL WHENEVER SQLERROR { ret = _sqlc->sqlcode; };
extern "C" __declspec( dllexport )
void ServerSideFunction( an_extfn_api *api, void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;
    EXEC SQL BEGIN DECLARE SECTION;
    char *stmt_text =
        "INSERT INTO esqlTab "
            "SELECT table_id, table_name "
            "FROM SYS.SYSTAB";
    char *stmt_commit =
        "COMMIT";
    EXEC SQL END DECLARE SECTION;
    int ret = -1;
    // set up the return value struct
    retval.type = DT_INT;
    retval.data = (void*) &ret;
    retval.piece_len = retval.len.total_len =
```

```
        (a_sql_uint32) sizeof( int );
    result = api->get_value( arg_handle,
                    EXTFN_CONNECTION_HANDLE_ARG_NUM,
                    &arg );
    if( result == 0 || arg.data == NULL )
    {
        api->set_value( arg_handle, 0, &retval, 0 );
        return;
    }
    ret = 0;
    _sqlc = (SQLCA *)arg.data;
    EXEC SQL EXECUTE IMMEDIATE :stmt_text;
    EXEC SQL EXECUTE IMMEDIATE :stmt_commit;
    api->set_value( arg_handle, 0, &retval, 0 );
}
```

上記の Embedded SQL 文が `extesql.sqc` ファイルに保存されている場合、次のコマンドを使用して Windows 用に構築できます (データベースサーバソフトウェアがフォルダ `c:¥sa17` にインストールされており、Microsoft Visual C++ もインストールされていることが前提です)。

```
sqlpp extesql.sqc extesql.cpp
cl extesql.cpp /LD /Ic:¥sa17¥sdk¥include c:¥sa17¥sdk¥lib¥x86¥dblibtm.lib
```

次の例では、テーブルを作成し、コンパイル済みネイティブ関数を呼び出すストアドプロシージャのラッパーを定義してから、ネイティブ関数を呼び出してテーブルにデータを移植します。

```
CREATE TABLE esqlTab(c1 int, c2 char(128));
CREATE FUNCTION ServerSideESQL( )
RETURNS INT
EXTERNAL NAME 'ServerSideFunction@extesql.dll'
LANGUAGE C_ESQL32;
SELECT ServerSideESQL();
// The following statement should return two identical rows
SELECT COUNT(*) FROM esqlTab
UNION ALL
SELECT COUNT(*) FROM SYS.SYSTAB;
```

前述の例のように、サーバ側の C API 呼び出しを使用するには、C/C++ コードでデフォルトのデータベース接続を使用する必要があります。データベース接続のハンドルを取得するには、EXTFN_CONNECTION_HANDLE_ARG_NUM 引数を指定して get_value を呼び出します。この引数は、新しい外部環境接続を開くのではなく、現在の外部環境接続を返すようにデータベースサーバに伝えます。次の例は、接続ハンドルを取得し、C API 環境を初期化し、接続ハンドルを C API で使用できる接続オブジェクト (a_sqlany_connection) に変換するためのフレームワークを示したものです。

```
#include <windows.h>
#include "sacapidll.h"
#include "extfnapi.h"
BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}
extern "C" __declspec( dllexport )
void ServerSideFunction( an_extfn_api *extapi, void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;
    unsigned            offset;
    char                *cmd;
```

```
    SQLAnywhereInterface  capi;
    a_sqlany_connection * sqlany_conn;
    unsigned int          max_api_ver;
    result = extapi->get_value( arg_handle,
                    EXTFN_CONNECTION_HANDLE_ARG_NUM,
                    &arg );
    if( result == 0 || arg.data == NULL )
    {
        return;
    }
    if( !sqlany_initialize_interface( &capi, NULL ) )
    {
        return;
    }
    if( !capi.sqlany_init( "MyApp",
            SQLANY_CURRENT_API_VERSION,
            &max_api_ver ) )
    {
        sqlany_finalize_interface( &capi );
        return;
    }
    sqlany_conn = sqlany_make_connection( arg.data );
    // processing code goes here
    capi.sqlany_fini();
    sqlany_finalize_interface( &capi );
    return;
}
```

上記の C コードが `extcapi.c` ファイルに保存されている場合、次のコマンドを使用して Windows 用に構築できます (データベースサーバソフトウェアがフォルダ `c:¥sa17` にインストールされており、Microsoft Visual C++ もインストールされていることが前提です)。

```
cl /LD /Tp extcapi.c /Tp c:¥sa17¥SDK¥C¥sacapidll.c
      /Ic:¥sa17¥SDK¥Include c:¥sa17¥SDK¥Lib¥X86¥dbcapi.lib
```

次の例では、コンパイル済みネイティブ関数を呼び出すストアドプロシージャのラッパーを定義してから、ネイティブ関数を呼び出します。

```
CREATE FUNCTION ServerSideC()
RETURNS INT
EXTERNAL NAME 'ServerSideFunction@extcapi.dll'
LANGUAGE C_ESQL32;
SELECT ServerSideC();
```

上記の例の LANGUAGE 属性では C_ESQL32 を指定しています。64 ビットのアプリケーションの場合は C_ESQL64 を使用します。C API は ESQL と同じレイヤ (ライブラリ) に構築されているため、Embedded SQL の LANGUAGE 属性を使用する必要があります。

前述のとおり、外部環境呼び出しを実行する接続は、`dbexternc17` のコピーをそれぞれ開始します。この実行可能アプリケーションは、最初の外部環境呼び出しが実行される際にサーバによって自動的にロードされます。ただし、START EXTERNAL ENVIRONMENT 文を使用して `dbexternc17` をプリロードすることもできます。外部環境呼び出しを初めて実行する際のわずかな遅延を回避したい場合には便利です。次に、この文の例を示します。

```
START EXTERNAL ENVIRONMENT C_ESQL32
```

`dbexternc17` のプリロードは、外部関数をデバッグする場合も便利です。デバッガを使用して実行中の `dbexternc17` プロセスにアタッチし、外部関数にブレークポイントを設定できます。

STOP EXTERNAL ENVIRONMENT 文は、ダイナミックリンクライブラリや共有オブジェクトを更新する場合に便利です。現在の接続でネイティブライブラリローダの `dbexternc17` を終了し、ダイナミックリンクライブラリや共有オブジェクトへのアクセス

を解放します。複数の接続が同じダイナミックリンクライブラリまたは共有オブジェクトを使用している場合は、dbexternc17 の各コピーを終了する必要があります。STOP EXTERNAL ENVIRONMENT 文には、適切な外部環境名を指定する必要があります。次に、この文の例を示します。

```
STOP EXTERNAL ENVIRONMENT C_ESQL32
```

外部関数から結果セットを返すためには、コンパイル済みネイティブ関数は外部呼び出しインタフェースを使用する必要があります。

次のコードフラグメントは、結果セット情報の構造体を設定する方法を示したものです。カラムカウント、カラム情報の構造体の配列へのポインタ、カラムデータ値の構造体の配列へのポインタを含んでいます。この例は、C API も使用しています。

```
an_extfn_result_set_info    rs_info;
int columns = capi.sqlany_num_cols( sqlany_stmt );
an_extfn_result_set_column_info *col_info =
    (an_extfn_result_set_column_info *)
    malloc( columns * sizeof(an_extfn_result_set_column_info) );
an_extfn_result_set_column_data *col_data =
    (an_extfn_result_set_column_data *)
    malloc( columns * sizeof(an_extfn_result_set_column_data) );
rs_info.number_of_columns   = columns;
rs_info.column_infos        = col_info;
rs_info.column_data_values  = col_data;
```

次のコードフラグメントは、結果セットを記述する方法を示したものです。C API を使用して、C API によって実行された SQL クエリのカラム情報を取得します。C API から取得した各カラムの情報は、カラムの名前、型、幅、インデックス、および NULL 値インジケータに変換され、結果セットの記述に使用されます。

```
a_sqlany_column_info        info;
for( int i = 0; i < columns; i++ )
{
    if( sqlany_get_column_info( sqlany_stmt, i, &info ) )
    {
        // set up a column  description
        col_info[i].column_name  = info.name;
        col_info[i].column_type  = info.native_type;
        switch( info.native_type )
        {
            case DT_DATE:       // DATE is converted to string by C API
            case DT_TIME:       // TIME is converted to string by C API
            case DT_TIMESTAMP:  // TIMESTAMP is converted to string by C API
            case DT_DECIMAL:    // DECIMAL is converted to string by C API
                col_info[i].column_type  = DT_FIXCHAR;
                break;
            case DT_FLOAT:      // FLOAT is converted to double by C API
                col_info[i].column_type  = DT_DOUBLE;
                break;
            case DT_BIT:        // BIT is converted to tinyint by C API
                col_info[i].column_type  = DT_TINYINT;
                break;
        }
        col_info[i].column_width = info.max_size;
        col_info[i].column_index = i + 1; // column indices are origin 1
        col_info[i].column_can_be_null = info.nullable;
    }
}
// send the result set description
if( extapi->set_value( arg_handle,
                    EXTFN_RESULT_SET_ARG_NUM,
                    (an_extfn_value *)&rs_info,
                    EXTFN_RESULT_SET_DESCRIBE ) == 0 )
{
    // failed
```

```
    free( col_info );
    free( col_data );
    return;
}
```

結果セットが記述されると、結果セットのローを返すことができます。次のコードフラグメントは、結果セットのローを返す方法を示したものです。C API を使用して、C API によって実行された SQL クエリのローをフェッチします。C API によって返されたローは、呼び出しを行った環境に 1 つずつ送り返されます。カラムデータ値の構造体の配列に格納してから、各ローを返す必要があります。カラムデータ値の構造体はカラムインデックス、データ値へのポインタ、データ長、追加フラグから構成されます。

```
a_sqlany_data_value *value = (a_sqlany_data_value *)
    malloc( columns * sizeof(a_sqlany_data_value) );
while( capi.sqlany_fetch_next( sqlany_stmt ) )
{
    for( int i = 0; i < columns; i++ )
    {
        if( capi.sqlany_get_column( sqlany_stmt, i, &value[i] ) )
        {
            col_data[i].column_index = i + 1;
            col_data[i].column_data  = value[i].buffer;
            col_data[i].data_length  = (a_sql_uint32)*(value[i].length);
            col_data[i].append  = 0;
            if( *(value[i].is_null) )
            {
                // Received a NULL value
                col_data[i].column_data = NULL;
            }
        }
    }
    if( extapi->set_value(  arg_handle,
                        EXTFN_RESULT_SET_ARG_NUM,
                        (an_extfn_value *)&rs_info,
                        EXTFN_RESULT_SET_NEW_ROW_FLUSH ) == 0 )
    {
        // failed
        free( value );
        free( col_data );
        free( col_data );
        extapi->set_value( arg_handle, 0, &retval, 0 );
        return;
    }
}
```

サーバ側の要求の作成方法、および外部関数から結果セットを返す方法の詳細については、*%SQLANYSAMP17%*`¥SQLAnywhere¥ExternalEnvironments¥ExternC` のサンプルを参照してください。

関連情報

外部呼び出しインタフェース [376 ページ]

1.13.3 Java 外部環境

Java メソッドを SQL ストアドプロシージャと同様にデータベースから呼び出すことができます。

Java メソッドの動作は、SQL ストアドプロシージャまたは関数と同じです。ただし、Java メソッドのコードは Java で記述され、その実行はデータベースサーバの外側 (つまり Java VM 環境内) で行われます。

データベースごとに Java VM のインスタンスが 1 つずつ存在できます。または、データベースサーバごとに Java VM のインスタンスが 1 つずつ存在できます (つまり、すべてのデータベースが同じインスタンスを使用します)。

Java ストアドプロシージャは結果セットを返すことができます。

データベースでの Java のサポートを使用するためには、いくつかの前提条件があります。

- Java Runtime Environment のコピーをデータベースサーバコンピュータにインストールする必要があります。
- データベースサーバが Java 実行ファイル (Java VM) を検出できる必要があります。

Java をデータベースで使用するには、データベースサーバが Java 実行ファイルを検出して開始できることを確認してください。これは、次の文を実行して確認できます。

```
START EXTERNAL ENVIRONMENT JAVA;
```

データベースサーバが Java を開始できない場合は、データベースサーバが Java 実行ファイルを検出できないことが問題の原因であると考えられます。この場合は、ALTER EXTERNAL ENVIRONMENT 文を実行して、Java 実行ファイルのロケーションを明示的に設定してください。実行ファイル名を必ず含めてください。

```
ALTER EXTERNAL ENVIRONMENT JAVA
  LOCATION 'java-path';
```

例:

```
ALTER EXTERNAL ENVIRONMENT JAVA
  LOCATION 'c:¥¥jdk1.8.0¥¥jre¥¥bin¥¥java.exe';
```

次の SQL クエリを実行して、データベースサーバが使用する Java VM のロケーションを問い合わせることができます。

```
SELECT db_property('JAVAVM');
```

START EXTERNAL ENVIRONMENT JAVA 文は、データベースサーバが Java VM を開始できるかどうかを確認する以外で使用することはありません。通常、Java ストアドプロシージャまたはファンクションを呼び出すと、Java VM は自動的に開始されます。

これと同様に、Java のインスタンスを停止するために STOP EXTERNAL ENVIRONMENT JAVA 文を使用する必要もありません。インスタンスは、データベースへの接続がすべて切断されると自動的に停止します。ただし、Java をこれ以上使用することがなく、一部のリソースを解放する必要がある場合は、STOP EXTERNAL ENVIRONMENT JAVA 文によって Java VM の使用カウントを減分できます。

データベースサーバが Java VM 実行ファイルを開始できることを確認したら、次に必要な Java クラスコードをデータベースにインストールします。これは、INSTALL JAVA 文を使用して行います。たとえば、次の文を実行して Java クラスをファイルからデータベースにインストールできます。

```
INSTALL JAVA
  NEW
  FROM FILE 'java-class-file';
```

データベースには、Java JAR ファイルもインストールできます。

```
INSTALL JAVA
  NEW
  JAR 'jar-name'
  FROM FILE 'jar-file';
```

Java クラスは、次のようにして変数からインストールできます。

```
CREATE VARIABLE JavaClass LONG VARCHAR;
SET JavaClass = xp_read_file('java-class-file');
INSTALL JAVA
  NEW
  FROM JavaClass;
```

Java JAR ファイルは、次のようにして変数からインストールできます。

```
CREATE VARIABLE JavaJar LONG VARCHAR;
SET JavaJar = xp_read_file('jar-file');
INSTALL JAVA
  NEW
  JAR 'jar-name'
  FROM JavaJar;
```

Java クラスをデータベースから削除するには、次のように REMOVE JAVA 文を使用します。

```
REMOVE JAVA CLASS java-class
```

Java JAR をデータベースから削除するには、次のように REMOVE JAVA 文を使用します。

```
REMOVE JAVA JAR 'jar-name'
```

既存の Java クラスを変更するには、次のように INSTALL JAVA 文の UPDATE 句を使用します。

```
INSTALL JAVA
  UPDATE
  FROM FILE 'java-class-file'
```

データベース内の既存の Java JAR ファイルを更新することもできます。

```
INSTALL JAVA
  UPDATE
  JAR 'jar-name'
  FROM FILE 'jar-file';
```

Java クラスは、次のようにして変数から更新できます。

```
CREATE VARIABLE JavaClass LONG VARCHAR;
SET JavaClass = xp_read_file('java-class-file');
INSTALL JAVA
  UPDATE
  FROM JavaClass;
```

Java JAR ファイルは、次のようにして変数から更新できます。

```
CREATE VARIABLE JavaJar LONG VARCHAR;
SET JavaJar = xp_read_file('jar-file');
INSTALL JAVA
  UPDATE
  FROM JavaJar;
```

Java クラスをデータベースにインストールしたら、次に Java メソッドへのインタフェースとなるストアドプロシージャおよび関数を作成できます。EXTERNAL NAME 文字列には、Java メソッドを呼び出し、OUT パラメータおよび戻り値を返すために必要な情報が含まれています。EXTERNAL NAME 句の LANGUAGE 属性には JAVA を指定する必要があります。EXTERNAL NAME 句のフォーマットは次のとおりです。

```
EXTERNAL NAME 'java-call' LANGUAGE JAVA
```

```
java-call :
[package-name.]class-name.method-name method-signature
```

```
method-signature :
( [ field-descriptor, ... ] ) return-descriptor
```

```
field-descriptor および return-descriptor :
Z
| B
| S
| I
| J
| F
| D
| C
| V
| [descriptor
| Lclass-name;
```

Java メソッドシグネチャは、パラメータの型と戻り値の型を簡潔に文字で表現したものです。パラメータの数がメソッドシグネチャに指定された数字よりも小さい場合は、この差が DYNAMIC RESULT SETS に指定された数と等しくなるようにします。また、プロシージャパラメータリストよりも多いメソッドシグネチャ内の各パラメータには、メソッドシグネチャ `[Ljava/sql/ResultSet;` が必要です。

`field-descriptor` および `return-descriptor` には次の意味があります。

フィールドタイプ	Java データ型
B	byte
C	char
D	double
F	float
I	int
J	long
L `class-name`;	クラス `class-name` のインスタンス。クラス名は、完全に修飾された名前です。また、名前内のドットは / に置き換える必要があります。例: java/lang/String
S	short
V	void
Z	Boolean
[	配列の各次元ごとに 1 つ使用します

例:

```
double some_method(
  boolean a,
  int b,
  java.math.BigDecimal c,
  byte [][] d,
  java.sql.ResultSet[] rs )
{
}
```

この例では、次のシグネチャを得られます。

```
'(ZILjava/math/BigDecimal;[[B[Ljava/sql/ResultSet;)D'
```

次のプロシージャは、Java メソッドへのインタフェースを作成するものです。この Java メソッドはいかなる値も返しません (V)。

```
CREATE PROCEDURE insertfix()
  EXTERNAL NAME 'JDBCExample.InsertFixed()V'
  LANGUAGE JAVA;
```

次のプロシージャは、String ([Ljava/lang/String;) 入力引数を持つ Java メソッドへのインタフェースを作成するものです。この Java メソッドはいかなる値も返しません (V)。

```
CREATE PROCEDURE InvoiceMain( IN arg1 CHAR(50) )
  EXTERNAL NAME 'Invoice.main([Ljava/lang/String;)V'
  LANGUAGE JAVA;
```

次のプロシージャは、Java メソッド Invoice.init へのインタフェースを作成するものです。この Java メソッドは、文字列引数 (Ljava/lang/String;)、double (D)、別の文字列引数 (Ljava/lang/String;)、別の double (D) を受け取り、値を返しません (V)。

```
CREATE PROCEDURE init( IN arg1 CHAR(50),
                       IN arg2 DOUBLE,
                       IN arg3 CHAR(50),
                       IN arg4 DOUBLE)
  EXTERNAL NAME 'Invoice.init(Ljava/lang/String;DLjava/lang/String;D)V'
  LANGUAGE JAVA;
```

次に示す Java の例は、文字列を受け取り、それをデータベースサーバメッセージウィンドウに書き込む関数 `main` を含んでいます。また、Java 文字列を返す関数 `whoAreYou` も含んでいます。

```
import java.io.*;
public class Hello
{
    public static void main( String[] args )
    {
        System.out.print( "Hello" );
               for ( int i = 0; i  < args.length; i++ )
                   System.out.print( " " + args[i] );
        System.out.println();
    }
    public static String whoAreYou()
    {
        return( "I am an external Java method." );
    }
}
```

上記の Java コードは、Hello.java ファイルにあり、Java コンパイラを使用してコンパイルします。生成されるクラスファイルは次のようにデータベースにロードされます。

```
INSTALL JAVA
  NEW
  FROM FILE 'Hello.class';
```

Hello クラスの main メソッドへのインタフェースとなるストアドプロシージャは、Interactive SQL を使用して次のように作成されます。

```
CREATE PROCEDURE HelloDemo( IN name LONG VARCHAR )
  EXTERNAL NAME 'Hello.main([Ljava/lang/String;)V'
  LANGUAGE JAVA;
```

main の引数は、java.lang.String の配列として記述されます。Interactive SQL を使用し、次の SQL 文を実行してインタフェースをテストしてください。

```
CALL HelloDemo('This is a test');
```

メッセージは、データベースサーバメッセージウィンドウに表示されます。System.out への出力はすべてサーバメッセージウィンドウにリダイレクトされます。

Hello クラスの whoAreYou メソッドへのインタフェースとなる関数は、Interactive SQL を使用して次のように作成されます。

```
CREATE FUNCTION WhoAreYou()
  RETURNS LONG VARCHAR
  EXTERNAL NAME 'Hello.whoAreYou(V)Ljava/lang/String;'
  LANGUAGE JAVA;
```

whoAreYou 関数は、java.lang.String を返すと記述されます。Interactive SQL を使用し、次の SQL 文を実行してインタフェースをテストしてください。

```
SELECT WhoAreYou();
```

Java 外部環境が開始されなかった理由をトラブルシューティングする場合、つまり、Java 呼び出しが行われたときにアプリケーションが "メインスレッドが見つかりません" というエラーを受け取った場合、DBA は次のことを確認する必要があります。

- Java VM のビット数がデータベースサーバと異なる場合、VM と同じビット数のクライアントライブラリがデータベースサーバコンピュータにインストールされていることを確認します。
- sajdbc4.jar 共有オブジェクトと dbjdbc*17*/libdbjdbc*17* 共有オブジェクトのソフトウェアビルドが同じであることを確認します。
- データベースサーバコンピュータに複数の sajdbc4.jar がある場合は、すべて同じソフトウェアバージョンに同期されていることを確認します。
- データベースサーバコンピュータが非常にビジーである場合、タイムアウトが原因でエラーがレポートされる可能性があります。

関連情報

Java メソッドから結果セットを返す方法 [206 ページ]
データベースにおける Java [191 ページ]

レッスン 4: Java クラスのメソッドの呼び出し [199 ページ]

1.13.4 JavaScript 外部環境

JavaScript のストアドプロシージャおよび関数を SQL ストアドプロシージャと同様にデータベースから呼び出すことができます。

JavaScript 関数の動作は、SQL プロシージャまたは関数と同じです。ただし、プロシージャまたは関数のコードは JavaScript で記述され、その実行はデータベースサーバの外側 (つまり JavaScript 実行インスタンス内) で行われます。JavaScript 関数を使用する接続ごとに JavaScript 実行環境の独立したインスタンスがあります。この動作は、Java ストアドプロシージャおよび関数と異なります。Java の場合、接続ごとに 1 つのインスタンスではなく、各データベースの Java VM に 1 つのインスタンスがあります。JavaScript と Java のもう 1 つの大きな相違点は、JavaScript 関数は結果セットを返さないのに対し、Java ストアドプロシージャは結果セットを返すことができる点です。

JavaScript をデータベースで使用するには、データベースサーバソフトウェアに付属の Node.js、XS JavaScript ドライバ、JavaScript 外部環境モジュールをインストールする必要があります。

このセクションの内容:

JavaScript 外部環境変数のインストール [416 ページ]
外部環境モジュールをインストールし、データベースの JavaScript をサポートします。

JavaScript をデータベースから使用する方法 [418 ページ]
SQL 文を使用してデータベースから JavaScript コードを実行します。

関連情報

XS JavaScript アプリケーションプログラミング [209 ページ]

1.13.4.1 JavaScript 外部環境変数のインストール

外部環境モジュールをインストールし、データベースの JavaScript をサポートします。

前提条件

- Node.js をデータベースサーバコンピュータにインストールする必要があります。
- データベースサーバは、Node.js バイナリフォルダを PATH に含めるなどの方法で Node.js 実行ファイル (Microsoft Windows では `C:¥Program Files¥nodejs¥node.exe`) を検出できる必要があります。

- データベースサーバソフトウェアに付属の SQL Anywhere XS JavaScript ドライバおよび JavaScript 外部環境サポートモジュールをデータベースサーバコンピュータにインストールする必要があります。これらのモジュールは、*%SQLANY17%*¥Node にあります。

手順

1. XS JavaScript ドライバおよび JavaScript 外部環境サポートモジュールへのパスを追加します。次に、Microsoft Windows の例を示します。

   ```
   SET NODE_PATH=%SQLANY17%¥Node
   ```

2. データベースサーバが Node.js 実行ファイルを検出して開始できることを確認します。データベースサーバを起動して接続します。次の SQL 文を実行します。

   ```
   START EXTERNAL ENVIRONMENT JS;
   ```

3. Node.js を開始できない場合、データベースサーバは Node.js 実行ファイルをおそらく検出できていません。この場合は、ALTER EXTERNAL ENVIRONMENT 文を実行して、Node.js 実行ファイルのロケーションを明示的に設定してください。実行ファイル名を必ず含めてください。

   例:

   ```
   ALTER EXTERNAL ENVIRONMENT JS
     LOCATION 'c:¥¥Program Files¥¥nodejs¥¥node.exe';
   ```

   START EXTERNAL ENVIRONMENT JS 文は、データベースサーバが Node.js を開始できることを確認します。通常、JavaScript 関数を呼び出すと、Node.js が自動的に開始されます。

   これと同様に、Node.js のインスタンスを停止するために STOP EXTERNAL ENVIRONMENT JS 文を使用する必要もありません。インスタンスは、接続が切断されると自動的に停止します。ただし、JavaScript をこれ以上使用することがなく、一部のリソースを解放する必要がある場合は、STOP EXTERNAL ENVIRONMENT JS 文を使用して接続のための Node.js インスタンスを解放します。

結果

JavaScript 外部環境モジュールがインストールされます。

1.13.4.2 JavaScript をデータベースから使用する方法

SQL 文を使用してデータベースから JavaScript コードを実行します。

データベースサーバが Node.js 実行ファイルを開始できることを確認したら、INSTALL 文を使用し、必要な JavaScript コードをデータベースにインストールできます。たとえば、次の文を実行して JavaScript スクリプトをファイルからデータベースにインストールできます。

```
INSTALL EXTERNAL OBJECT 'javascript-object'
  NEW
  FROM FILE 'javascript-file'
  ENVIRONMENT JS;
```

また、次のように、JavaScript コードを式から構築してインストールできます。

```
INSTALL EXTERNAL OBJECT 'javascript-object'
  NEW
  FROM VALUE 'javascript-statements'
  ENVIRONMENT JS;
```

また、次のように、JavaScript コードを変数から構築してインストールできます。

```
CREATE VARIABLE javascript-variable LONG VARCHAR;
SET javascript-variable = 'javascript-statements';
INSTALL EXTERNAL OBJECT 'javascript-object'
  NEW
  FROM VALUE javascript-variable
  ENVIRONMENT JS;
```

JavaScript コードをデータベースから削除するには、次のように REMOVE 文を使用します。

```
REMOVE EXTERNAL OBJECT 'javascript-object';
```

既存の JavaScript コードを変更するには、次のように INSTALL EXTERNAL OBJECT 文の UPDATE 句を使用します。

```
INSTALL EXTERNAL OBJECT 'javascript-object'
  UPDATE
  FROM FILE 'javascript-file'
  ENVIRONMENT JS;
```

```
INSTALL EXTERNAL OBJECT 'javascript-object'
  UPDATE
  FROM VALUE 'javascript-statements'
  ENVIRONMENT JS;
```

```
SET javascript-variable = 'javascript-statements';
INSTALL EXTERNAL OBJECT 'javascript-object'
  UPDATE
  FROM VALUE javascript-variable
  ENVIRONMENT JS;
```

JavaScript コードがデータベースにインストールされたら、それを呼び出す JavaScript ストアドプロシージャおよび関数を作成できます。JavaScript ストアドプロシージャおよび関数を作成するときは、LANGUAGE に必ず JS を指定します。また、EXTERNAL NAME 文字列には、JavaScript 関数を呼び出し、OUT パラメータを返して、値を返すために必要な情報が含まれています。

SQL ストアドプロシージャまたは関数は、あらゆるデータ型セットを引数に指定して作成できます。ただし、JavaScript 関数内で使用されるために、パラメータは number または string の間で変換されます。JavaScript 関数の戻り値は、ユーザ定義 SQL 関数のタイプに変換されます。

山括弧 (<>) 内の EXTERNAL NAME 文字列の先頭に、JavaScript 関数の戻り値のタイプを指定します。有効な文字は S (String)、B (Boolean)、D (Double)、I (Integer)、U (Unsigned Integer) です。JavaScript パラメータのタイプは、JavaScript 関数名の後の括弧内に文字 S、B、D、I、U のシーケンスとしてリストされます。

JavaScript では関数内の単純変数の参照による受け渡しが許可されていないため、左角括弧文字 ([) は、1 つの要素の配列が JavaScript ストアドプロシージャに渡されていることを示すために S、B、D、I、または U の文字を処理できます。これは、ストアドプロシージャ内の INOUT および OUT パラメータを許可するために実行されます。

次の例では、JavaScript 関数を呼び出すユーザ定義 SQL 関数を示します。

```
INSTALL EXTERNAL OBJECT 'SimpleJSExample'
  NEW
  FROM VALUE 'function SimpleJSFunction(
    thousand, hundred, ten, one )
    { return (thousand * 1000) +
      (hundred * 100) +
      (ten * 10) +
      one;
    }'
  ENVIRONMENT JS;
CREATE FUNCTION SimpleJSDemo(
    IN thousands INT,
    IN hundreds INT,
    IN tens INT,
    IN ones INT)
  RETURNS INT
  EXTERNAL NAME '<I><file=SimpleJSExample> SimpleJSFunction(IIII)'
  LANGUAGE JS;
// The number 1234 should appear
SELECT SimpleJSDemo(1,2,3,4);
```

file= の参照対象は、実際のファイルではなく JavaScript オブジェクトです。ユーザ定義 SQL 関数は、JavaScript 関数が 4 つの整数引数を取り、整数値を返すことを示します。

次の例では、JavaScript プロシージャを呼び出す INOUT および OUT パラメータを指定した SQL ストアドプロシージャを示します。

```
INSTALL EXTERNAL OBJECT 'JSInOutParam'
  NEW
  FROM VALUE 'function JSFunctionPlusOne( number1, number2 )
    {
         number1[0] = number1[0] + 1;
         number2[0] = number1[0] * 2;
    }'
  ENVIRONMENT JS;
CREATE PROCEDURE JSInOutDemo( INOUT num1 INT, OUT num2 INT )
  EXTERNAL NAME '<file=JSInOutParam> JSFunctionPlusOne([I[I)'
  LANGUAGE JS;
BEGIN
  DECLARE @x INT;
  DECLARE @y INT;
  SET @x = 5;
  CALL JSInOutDemo( @x, @y );
  SELECT @x, @y;
END
```

サーバ側 JavaScript を使用するには、JavaScript コードは特殊な `sa_dbcapi_handle` 変数を使用してデータベースサーバに接続します。次の例では、SQL を使用する空のテーブルを作成し、テーブルに値を入力する JavaScript 関数を呼び出します。

```
CREATE OR REPLACE TABLE JSTab(c1 INT, c2 CHAR(128));
INSTALL EXTERNAL OBJECT 'ServerSideJS'
  NEW
  FROM VALUE 'function sjs()
    {
      var sqla = require( ''sqlanywhere-xs'' );
      var conn = sqla.createConnection();
      conn.connect( sa_dbcapi_handle );
      conn.prepareStatement(
        "MESSAGE ''JavaScript says hello'' TO CONSOLE" )
        .execute();
      conn.prepareStatement( "DELETE FROM JSTab" )
        .execute();
      conn.prepareStatement(
        "INSERT INTO JSTab SELECT table_id, table_name FROM SYS.SYSTAB" )
        .execute();
      conn.disconnect();
  }'
  ENVIRONMENT JS;
CREATE PROCEDURE JavaScriptPopulateTable()
  EXTERNAL NAME '<file=ServerSideJS> sjs()'
  LANGUAGE JS;
CALL JavaScriptPopulateTable();
// The following should return 2 identical rows
SELECT count(*) FROM JSTab
UNION ALL
SELECT count(*) FROM SYS.SYSTAB;
```

JavaScript コードは、呼び出し側の接続ハンドルを使用してデータベースサーバに接続し、データベースサーバメッセージウィンドウにメッセージを記述するその接続の SQL MESSGAE 文を実行し、SQL DELETE 文と INSERT 文を実行し、データベースサーバとの接続を切断します。

次の JavaScript コードが `JSLogger.js` というファイルにあるとします。`temp` フォルダ内のファイルにメッセージを記録します。

```
function JSLogger( message )
{
    var fs = require('fs');
    fs.appendFileSync( '/temp/javascript.log', message + "¥r¥n" );
}
```

次の例では、INSTALL EXTERNAL OBJECT ... FROM FILE 句を使用してこのファイルの JavaScript コードをデータベースにロードします。JavaScript コードに対するインタフェースである SQL のカバープロシージャを定義します。次に、この SQL プロシージャを呼び出します。

```
INSTALL EXTERNAL OBJECT 'JSLoggerFile'
  NEW
  FROM FILE '¥¥temp¥¥JSLogger.js'
  ENVIRONMENT JS;
CREATE OR REPLACE PROCEDURE JSLogger( IN msg LONG VARCHAR )
  EXTERNAL NAME '<file=JSLoggerFile> JSLogger(S)'
  LANGUAGE JS;
CALL JSLogger( 'Hello world!' );
```

次の例も同じですが、JavaScript コードは SQL 文の文字列として埋め込まれます。埋め込まれた JavaScript コードでアポストロフィ文字とバックスラッシュ文字を適切に処理できるように注意する必要があります。

```
INSTALL EXTERNAL OBJECT 'JSLoggerObject'
  NEW
  FROM VALUE 'function JSLogger( message )
    {
        var fs = require(''fs'');
        fs.appendFileSync( ''/temp/javascript.log'', message + "¥¥r¥¥n" );
    }'
  ENVIRONMENT JS;
CREATE OR REPLACE PROCEDURE JSLogger( IN msg LONG VARCHAR )
  EXTERNAL NAME '<file=JSLoggerObject> JSLogger(S)'
  LANGUAGE JS;
CALL JSLogger( 'Hello world!' );
```

データベースでの JavaScript サポートの使用に関する詳細および例については、*%SQLANYSAMP17%*`¥SQLAnywhere¥ExternalEnvironments¥JavaScript` ディレクトリのサンプルを参照してください。

1.13.5 Perl 外部環境

Perl のストアドプロシージャおよび関数を SQL ストアドプロシージャと同様にデータベースから呼び出すことができます。

Perl ストアドプロシージャまたは関数の動作は、SQL ストアドプロシージャまたは関数と同じです。ただし、プロシージャまたは関数のコードは Perl で記述され、その実行はデータベースサーバの外側 (つまり Perl 実行インスタンス内) で行われます。Perl 実行ファイルのインスタンスは、Perl ストアドプロシージャおよび関数を使用する接続ごとに存在します。この動作は、Java ストアドプロシージャおよび関数と異なります。Java の場合、接続ごとに 1 つのインスタンスではなく、各データベースの Java VM に 1 つのインスタンスがあります。Perl と Java のもう 1 つの大きな相違点は、Perl ストアドプロシージャは結果セットを返さないのに対し、Java ストアドプロシージャは結果セットを返すことができる点です。

このセクションの内容:

Perl 外部環境変数のインストール [422 ページ]
外部環境モジュールをインストールし、データベースの Perl をサポートします。

Perl をデータベースから使用する方法 [423 ページ]
SQL 文を使用してデータベースから Perl コードを実行します。

関連情報

Perl DBI サポート [431 ページ]

1.13.5.1 Perl 外部環境変数のインストール

外部環境モジュールをインストールし、データベースの Perl をサポートします。

前提条件

1. Perl をデータベースサーバコンピュータにインストールする必要があります。また、データベースサーバで Perl 実行ファイルを検出できることが必要です。
2. DBD::SQLAnywhere ドライバをデータベースサーバコンピュータにインストールする必要があります。
3. Windows の場合、Microsoft Visual Studio がインストールされていることが必要です。これが前提条件であるのは、DBD::SQLAnywhere ドライバをインストールするために必要だからです。

上記の前提条件に加え、データベース管理者は、データベースサーバソフトウェアに付属の Perl 外部環境モジュールをインストールする必要もあります。

手順

Perl 外部環境モジュールをインストールするには、次のいずれかを選択します。

オプション	説明

Windows	データベースサーバソフトウェアの `SDK¥PerlEnv` サブディレクトリから次のコマンドを実行します。
	`perl Makefile.PL`
	`nmake`
	`nmake install`

UNIX	データベースサーバソフトウェアの `sdk/perlenv` サブディレクトリから次のコマンドを実行します。
	`perl Makefile.PL`
	`make`
	`make install`

結果

Perl 外部環境モジュールがインストールされます。

1.13.5.2 Perl をデータベースから使用する方法

SQL 文を使用してデータベースから Perl コードを実行します。

Perl をデータベースで使用するには、データベースサーバが Perl 実行ファイルを検出して開始できることを確認してください。次の操作でこれを確認します。

```
START EXTERNAL ENVIRONMENT PERL;
```

データベースサーバが Perl を開始できない場合は、データベースサーバが Perl 実行ファイルを検出できないことが問題の原因であると考えられます。この場合は、ALTER EXTERNAL ENVIRONMENT 文を実行して、Perl 実行ファイルのロケーションを明示的に設定してください。実行ファイル名を必ず含めてください。

```
ALTER EXTERNAL ENVIRONMENT PERL
  LOCATION 'perl-path';
```

例:

```
ALTER EXTERNAL ENVIRONMENT PERL
  LOCATION 'c:¥¥Perl¥¥bin¥¥perl.exe';
```

START EXTERNAL ENVIRONMENT PERL 文は、データベースサーバが Perl を開始できるかどうかを確認する以外で使用することはありません。通常、Perl ストアドプロシージャまたはファンクションを呼び出すと、Perl は自動的に開始されます。

これと同様に、Perl のインスタンスを停止するために STOP EXTERNAL ENVIRONMENT PERL 文を使用する必要もありません。インスタンスは、接続が切断されると自動的に停止します。ただし、Perl をこれ以上使用することがなく、一部のリソースを解放する必要がある場合は、STOP EXTERNAL ENVIRONMENT PERL 文を使用して接続のための Perl インスタンスを解放します。

データベースサーバが Perl 実行ファイルを開始できることを確認したら、次に必要な Perl コードをデータベースにインストールします。これは、INSTALL 文を使用して行います。たとえば、次の文を実行して Perl スクリプトをファイルからデータベースにインストールできます。

```
INSTALL EXTERNAL OBJECT 'perl-script'
  NEW
  FROM FILE 'perl-file'
  ENVIRONMENT PERL;
```

また、次のように、Perl コードを式から構築してインストールできます。

```
INSTALL EXTERNAL OBJECT 'perl-script'
  NEW
  FROM VALUE 'perl-statements'
  ENVIRONMENT PERL;
```

また、次のように、Perl コードを変数から構築してインストールできます。

```
CREATE VARIABLE PerlVariable LONG VARCHAR;
SET PerlVariable = 'perl-statements';
INSTALL EXTERNAL OBJECT 'perl-script'
  NEW
  FROM VALUE PerlVariable
  ENVIRONMENT PERL;
```

Perl コードをデータベースから削除するには、次のように REMOVE 文を使用します。

```
REMOVE EXTERNAL OBJECT 'perl-script';
```

既存の Perl コードを変更するには、次のように INSTALL EXTERNAL OBJECT 文の UPDATE 句を使用します。

```
INSTALL EXTERNAL OBJECT 'perl-script'
  UPDATE
  FROM FILE 'perl-file'
  ENVIRONMENT PERL;
```

```
INSTALL EXTERNAL OBJECT 'perl-script'
  UPDATE
  FROM VALUE 'perl-statements'
  ENVIRONMENT PERL;
```

```
SET PerlVariable = 'perl-statements';
INSTALL EXTERNAL OBJECT 'perl-script'
  UPDATE
  FROM VALUE PerlVariable
  ENVIRONMENT PERL;
```

Perl コードがデータベースにインストールされたら、必要な Perl ストアドプロシージャおよび関数を作成できます。Perl ストアドプロシージャおよび関数を作成するときは、LANGUAGE に必ず PERL を指定します。また、EXTERNAL NAME 文字列には、Perl サブルーチンを呼び出し、OUT パラメータを返して、値を返すために必要な情報が含まれています。次のグローバル変数は、各呼び出し時に Perl コードで使用できます。

グローバル変数	説明
$sa_perl_return	これは、関数呼び出しの戻り値を設定するために使用します。
$sa_perl_argN	N は正の整数 [0 .. n] です。これは、SQL 引数を Perl コードに渡すために使用します。たとえば、`$sa_perl_arg0` は引数 0、`$sa_perl_arg1` は引数 1 を示し、以降の引数も同様です。
$sa_perl_default_connection	これは、サーバ側の Perl 呼び出しを作成するために使用します。
$sa_output_handle	これは、Perl コードの出力をデータベースサーバメッセージウィンドウに送信するために使用します。

Perl ストアドプロシージャは、入出力の引数および戻り値にあらゆるデータ型セットを指定して作成できます。非バイナリのデータ型はすべて Perl 呼び出しの作成時に文字列にマッピングされますが、バイナリデータは数値の配列にマッピングされます。簡単な Perl の例を次に示します。

```
INSTALL EXTERNAL OBJECT 'SimplePerlExample'
  NEW
  FROM VALUE 'sub SimplePerlSub{
    return( ($_[0] * 1000) +
            ($_[1] * 100) +
            ($_[2] * 10) +
           $_[3] );
  }'
  ENVIRONMENT PERL;
CREATE FUNCTION SimplePerlDemo(
    IN thousands INT,
    IN hundreds INT,
    IN tens INT,
    IN ones INT)
  RETURNS INT
  EXTERNAL NAME '<file=SimplePerlExample>
```

```
    $sa_perl_return = SimplePerlSub(
      $sa_perl_arg0,
      $sa_perl_arg1,
      $sa_perl_arg2,
      $sa_perl_arg3)'
  LANGUAGE PERL;
// The number 1234 should appear
SELECT SimplePerlDemo(1,2,3,4);
```

次に示す Perl の例は、文字列を受け取り、それをデータベースサーバメッセージウィンドウに書き込みます。

```
INSTALL EXTERNAL OBJECT 'PerlConsoleExample'
  NEW
  FROM VALUE 'sub WriteToServerConsole { print $sa_output_handle $_[0]; }'
  ENVIRONMENT PERL;
CREATE PROCEDURE PerlWriteToConsole( IN str LONG VARCHAR)
  EXTERNAL NAME '<file=PerlConsoleExample>
    WriteToServerConsole( $sa_perl_arg0 )'
  LANGUAGE PERL;
// 'Hello world' should appear in the database server messages window
CALL PerlWriteToConsole( 'Hello world' );
```

サーバ側の Perl を使用するには、Perl コードに $sa_perl_default_connection 変数を使用する必要があります。次の例では、テーブルを作成してから Perl ストアドプロシージャを呼び出して、テーブルにデータを移植します。

```
CREATE TABLE perlTab(c1 int, c2 char(128));
INSTALL EXTERNAL OBJECT 'ServerSidePerlExample'
  NEW
  FROM VALUE 'sub ServerSidePerlSub
    { $sa_perl_default_connection->do(
        "INSERT INTO perlTab SELECT table_id, table_name FROM SYS.SYSTAB" );
      $sa_perl_default_connection->do(
        "COMMIT" );
    }'
  ENVIRONMENT PERL;
CREATE PROCEDURE PerlPopulateTable()
  EXTERNAL NAME '<file=ServerSidePerlExample> ServerSidePerlSub()'
  LANGUAGE PERL;
CALL PerlPopulateTable();
// The following should return 2 identical rows
SELECT count(*) FROM perlTab
UNION ALL
SELECT count(*) FROM SYS.SYSTAB;
```

データベースでの Perl サポートの使用に関する詳細および例については、*%SQLANYSAMP17%*`¥SQLAnywhere¥ExternalEnvironments¥Perl` ディレクトリのサンプルを参照してください。

1.13.6 PHP 外部環境

PHP のストアドプロシージャおよび関数を SQL ストアドプロシージャと同様にデータベースから呼び出すことができます。

PHP ストアドプロシージャまたは関数の動作は、SQL ストアドプロシージャまたは関数と同じです。ただし、プロシージャまたは関数のコードは PHP で記述され、その実行はデータベースサーバの外側 (つまり PHP 実行インスタンス内) で行われます。PHP 実行ファイルのインスタンスは、PHP ストアドプロシージャおよび関数を使用する接続ごとに存在します。この動作は、Java ストアドプロシージャおよび関数と異なります。Java の場合、接続ごとに 1 つのインスタンスではなく、各データベースの Java VM に 1 つのインスタンスがあります。PHP と Java のもう 1 つの大きな相違点は、PHP ストアドプロシージャは結果セットを返さないのに対し、Java ストアドプロシージャは結果セットを返すことができる点です。PHP で返すのは、PHP スクリプトの出力である LONG VARCHAR 型のオブジェクトだけです。

データベースで PHP を使用するには、最初に PHP をインストールする必要があります。詳細については、http://php.net を参照してください。

次に、http://scn.sap.com/docs/DOC-40537 からダウンロードできる PHP ドライバおよび PHP 外部環境モジュールをインストールし、これらのコンポーネントを配置してロードするように PHP を設定する必要があります。次の項では、これらの手順について説明します。

このセクションの内容:

PHP 外部環境モジュールの Windows または Linux へのインストール [426 ページ]
外部環境モジュールをインストールし、データベースの PHP をサポートします。

PHP をデータベースから使用する方法 [427 ページ]
SQL 文を使用してデータベースから PHP コードを実行します。

1.13.6.1 PHP 外部環境モジュールの Windows または Linux へのインストール

外部環境モジュールをインストールし、データベースの PHP をサポートします。

前提条件

1. PHP をデータベースサーバコンピュータにインストールする必要があります。また、データベースサーバで PHP 実行ファイルを検出できることが必要です。
2. SQL Anywhere PHP 拡張がデータベースサーバコンピュータにインストールされていて、PHP がこれを使用するように設定されていることが必要です。スレッド対応および非スレッド対応のいずれかの拡張を使用できます。

コンテキスト

上記 2 つの前提条件に加え、データベース管理者は PHP 外部環境モジュールをインストールする必要もあります。ビルド済みモジュールをインストールするには、適切なモジュールを php.ini で指定されている PHP 拡張ディレクトリにコピーします。UNIX では、シンボリックリンクを使用することもできます。次の手順は、これを行う方法を示します。

手順

1. ビルド済みバージョンの PHP 拡張および外部環境モジュールは、http://scn.sap.com/docs/DOC-40537 からダウンロードします。
2. PHP インストールディレクトリにある `php.ini` ファイルを探して、テキストエディタで開きます。extension_dir ディレクトリのロケーションを指定する行を探します。extension_dir に特定のディレクトリが設定されていない場合は、システムセキュリティの安全上、独立したディレクトリを指定します。

3. 目的の外部環境 PHP モジュールを PHP 拡張ディレクトリにコピーします。

   32 ビットおよび 64 ビットバージョンのモジュールは、さまざまな PHP バージョンで利用可能です。外部環境モジュールのファイル名の一部には次のパターンが含まれます (`5.x` は使用している PHP バージョンに対応します):

   ```
   php-5.x.0_sqlanywhere_extenv17
   ```

4. 外部環境 PHP モジュールを自動的にロードするために、行を `php.ini` ファイルの Dynamic Extensions セクションに追加します。選択したバージョンを反映するために `5.x` を変更します。

   Windows の場合、次の行を追加します。

   ```
   extension=php-5.x.0_sqlanywhere_extenv17.dll
   ```

   UNIX の場合、次の行を追加します。

   ```
   extension=php-5.x.0_sqlanywhere_extenv17.so
   ```

5. `php.ini` を保存して閉じます。
6. SQL Anywhere PHP 拡張も PHP の拡張ディレクトリにインストールされていることを確認します。

   PHP 拡張のインストール手順は別の場所に記載されています。

結果

外部環境モジュールがインストールされます。

関連情報

PHP クライアントの配備 [829 ページ]
PHP クライアントの配備 [829 ページ]

1.13.6.2 PHP をデータベースから使用する方法

SQL 文を使用してデータベースから PHP コードを実行します。

PHP をデータベースで使用するには、データベースサーバが PHP 実行ファイルを検出して開始できる必要があります。データベースサーバが PHP 実行ファイルを検出して開始できるかどうかを確認するには、次の文を実行します。

```
START EXTERNAL ENVIRONMENT PHP;
```

'外部実行ファイル' が見つからないというメッセージが表示された場合、問題の原因はデータベースサーバが PHP 実行ファイルを検出できていないことにあります。この場合は、ALTER EXTERNAL ENVIRONMENT 文を実行し、PHP 実行ファイル

のロケーション (実行ファイル名も含む) を明示的に設定するか、PHP 実行ファイルがあるディレクトリが PATH 環境変数に含まれていることを確認します。

```
ALTER EXTERNAL ENVIRONMENT PHP
  LOCATION 'php-path';
```

例:

```
ALTER EXTERNAL ENVIRONMENT PHP
  LOCATION 'c:¥¥php¥¥php.exe';
```

デフォルト設定に戻すには、次の文を実行します。

```
ALTER EXTERNAL ENVIRONMENT PHP
  LOCATION 'php';
```

'メインスレッド' が見つからないというメッセージが表示された場合は、次のことを確認します。

- `php-5.x.0_sqlanywhere` および `php-5.x.0_sqlanywhere_extenv17` の両モジュールが、`php.ini` ファイルの extension_dir が示すディレクトリに存在することを確認します。`5.x` は使用している PHP のバージョンを表します。前の項で説明したインストール手順を確認します。
- `php-5.x.0_sqlanywhere` および `php-5.x.0_sqlanywhere_extenv17` の両モジュールが、`php.ini` ファイルの Dynamic Extensions セクションにリストされていることを確認します。前の項で説明したインストール手順を確認します。
- `phpenv.php` が検出できることを確認します。データベースサーバの `bin32` フォルダが PATH に含まれていることを確認します。
- Windows の場合、64 ビットまたは 32 ビットの DLL (`dbcapi.dll`、`dblib17.dll`、`dbicu17.dll`、`dbicudt17.dll`、`dblgen17.dll`、`dbextenv17.dll`、`dbrsa17.dll`) を検出できることを確認します。データベースサーバの `bin32` または `bin64` フォルダが PATH に含まれていることを確認します。
- UNIX 系オペレーティングシステムの場合、32 ビットまたは 64 ビットの共有オブジェクト (`libdbcapi_r`、`libdblib17_r`、`libdbicu17_r`、`libdbicudt17`、`dblgen17.res`、`libdbextenv17_r`、`libdbrsa17_r`) を検出できることを確認します。データベースサーバの `bin32` または bin64 フォルダが PATH に含まれていることを確認します。
- 環境変数 PHPRC が設定されていないこと、または使用するバージョンの PHP を指していることを確認します。

START EXTERNAL ENVIRONMENT PHP 文は、データベースサーバが PHP を開始できるかどうかを確認する以外で使用することはありません。通常、PHP ストアドプロシージャまたはファンクションを呼び出すと、PHP は自動的に開始されます。

これと同様に、PHP のインスタンスを停止するために STOP EXTERNAL ENVIRONMENT PHP 文を使用する必要もありません。インスタンスは、接続が切断されると自動的に停止します。ただし、PHP をこれ以上使用することがなく、一部のリソースを解放する必要がある場合は、STOP EXTERNAL ENVIRONMENT PHP 文を使用して接続のための PHP インスタンスを解放します。

データベースサーバが PHP 実行ファイルを開始できることを確認したら、次に PHP コードをデータベースにインストールします。これは、INSTALL 文を使用して行います。たとえば、次の文を実行して、特定の PHP スクリプトをデータベースにインストールできます。

```
INSTALL EXTERNAL OBJECT 'php-script'
  NEW
  FROM FILE 'php-file'
  ENVIRONMENT PHP;
```

PHP コードは、次のようにして式から構築してインストールすることもできます。

```
INSTALL EXTERNAL OBJECT 'php-script'
  NEW
  FROM VALUE 'php-statements'
  ENVIRONMENT PHP;
```

PHP コードは、次のようにして変数から構築してインストールすることもできます。

```
CREATE VARIABLE PHPVariable LONG VARCHAR;
SET PHPVariable = 'php-statements';
INSTALL EXTERNAL OBJECT 'php-script'
  NEW
  FROM VALUE PHPVariable
  ENVIRONMENT PHP;
```

PHP コードをデータベースから削除するには、次のように REMOVE 文を使用します。

```
REMOVE EXTERNAL OBJECT 'php-script';
```

既存の PHP コードを変更するには、次のように INSTALL 文の UPDATE 句を使用します。

```
INSTALL EXTERNAL OBJECT 'php-script'
  UPDATE
  FROM FILE 'php-file'
  ENVIRONMENT PHP;
```

```
INSTALL EXTERNAL OBJECT 'php-script'
  UPDATE
  FROM VALUE 'php-statements'
  ENVIRONMENT PHP;
```

```
SET PHPVariable = 'php-statements';
INSTALL EXTERNAL OBJECT 'php-script'
  UPDATE
  FROM VALUE PHPVariable
  ENVIRONMENT PHP;
```

PHP コードがデータベースにインストールされたら、次に必要な PHP ストアドプロシージャおよび関数を作成できます。PHP ストアドプロシージャおよび関数を作成するときは、LANGUAGE に必ず PHP を指定します。また、EXTERNAL NAME 文字列には、PHP サブルーチンを呼び出し、OUT パラメータを返すために必要な情報が含まれています。

引数は $argv 配列で PHP スクリプトに渡されます。これは、PHP がコマンドラインから引数を受け取る方法 ($argv[1] が最初の引数) に似ています。出力パラメータを設定するには、出力パラメータを適切な $argv 要素に割り当てます。戻り値は、常にスクリプトの出力 (LONG VARCHAR) です。

PHP ストアドプロシージャは、入出力の引数にあらゆるデータ型セットを指定して作成できます。ただし、PHP スクリプト内で使用されるために、パラメータは boolean、integer、double、または string の間で変換されます。戻り値は、常に LONG VARCHAR 型のオブジェクトです。簡単な PHP の例を次に示します。

```
INSTALL EXTERNAL OBJECT 'SimplePHPExample'
  NEW
  FROM VALUE '<?php function SimplePHPFunction(
    $arg1, $arg2, $arg3, $arg4 )
    { return ($arg1 * 1000) +
      ($arg2 * 100) +
      ($arg3 * 10) +
      $arg4;
    } ?>'
```

```
  ENVIRONMENT PHP;
CREATE FUNCTION SimplePHPDemo(
  IN thousands INT,
  IN hundreds INT,
  IN tens INT,
  IN ones INT)
RETURNS LONG VARCHAR
EXTERNAL NAME '<file=SimplePHPExample> print SimplePHPFunction(
    $argv[1], $argv[2], $argv[3], $argv[4]);'
LANGUAGE PHP;
// The number 1234 should appear
SELECT SimplePHPDemo(1,2,3,4);
```

PHP では、EXTERNAL NAME 文字列は 1 行の SQL 文で指定されます。

サーバ側の PHP を使用するには、PHP コードでデフォルトのデータベース接続を使用します。データベース接続のハンドルを取得するには、空の文字列引数 ('' または "") を指定して sasql_pconnect を呼び出します。空の文字列引数を指定することで、新しい外部環境接続を開くのではなく、現在の外部環境接続を返すように PHP ドライバに伝えます。次の例では、テーブルを作成してから PHP ストアドプロシージャを呼び出して、テーブルにデータを移植します。

```
CREATE TABLE phpTab(c1 int, c2 char(128));
INSTALL EXTERNAL OBJECT 'ServerSidePHPExample'
  NEW
  FROM VALUE '<?php function ServerSidePHPSub() {
    $conn = sasql_pconnect( '''' );
    sasql_query( $conn,
    "INSERT INTO phpTab
       SELECT table_id, table_name FROM SYS.SYSTAB" );
    sasql_commit( $conn );
  } ?>'
  ENVIRONMENT PHP;
CREATE PROCEDURE PHPPopulateTable()
EXTERNAL NAME '<file=ServerSidePHPExample> ServerSidePHPSub()'
LANGUAGE PHP;
CALL PHPPopulateTable();
// The following should return 2 identical rows
SELECT count(*) FROM phpTab
UNION ALL
SELECT count(*) FROM SYS.SYSTAB;
```

PHP では、EXTERNAL NAME 文字列は 1 行の SQL 文で指定されます。上の例では、SQL での引用符の解析方法に従って、単一引用符が二重になっています。PHP ソースコードがファイル内にある場合は、単一引用符を二重にしません。

エラーをデータベースサーバに戻すには、PHP 例外をスローします。次の例は、これを行う方法を示します。

```
CREATE TABLE phpTab(c1 int, c2 char(128));
INSTALL EXTERNAL OBJECT 'ServerSidePHPExample'
  NEW
  FROM VALUE '<?php function ServerSidePHPSub() {
    $conn = sasql_pconnect( '''' );
    if( !sasql_query( $conn,
      "INSERT INTO phpTabNoExist
         SELECT table_id, table_name FROM SYS.SYSTAB" )
    ) throw new Exception(
      sasql_error( $conn ),
      sasql_errorcode( $conn )
    );
    sasql_commit( $conn );
  } ?>'
  ENVIRONMENT PHP;
CREATE PROCEDURE PHPPopulateTable()
  EXTERNAL NAME
    '<file=ServerSidePHPExample> ServerSidePHPSub()'
  LANGUAGE PHP;
```

```
CALL PHPPopulateTable();
```

上の例は、テーブル phpTabNoExist が見つからないことを示す SQLE_UNHANDLED_EXTENV_EXCEPTION エラーで終了します。

データベースでの PHP サポートの使用に関する詳細および例については、*%SQLANYSAMP17%*¥SQLAnywhere¥ExternalEnvironments¥PHP ディレクトリのサンプルを参照してください。

1.14 Perl DBI サポート

DBD::SQLAnywhere は Perl Database Independent Interface (DBI) 用のデータベースドライバで、Perl 言語用のデータアクセス API です。

Perl DBI API 仕様は、実際に使用されているデータベースとは関係なく一貫したデータベースインタフェースを提供する一連の関数、変数、規則を定義します。Perl DBI と DBD::SQLAnywhere を使用すると、Perl スクリプトから SQL Anywhere データベースサーバに直接アクセスできるようになります。

このセクションの内容:

DBD::SQLAnywhere [431 ページ]
DBD::SQLAnywhere は、Perl 用の Database Independent Interface (DBI) モジュールのドライバです。Perl DBI モジュールと DBD::SQLAnywhere をインストールすると、Perl から SQL Anywhere データベースの情報にアクセスして変更できるようになります。

Windows での DBD::SQLAnywhere のインストール [432 ページ]
サポートされている Windows プラットフォームに DBD::SQLAnywhere インタフェースをインストールし、Perl を使用してデータベースにアクセスします。

UNIX と Mac OS X での DBD::SQLAnywhere のインストール [434 ページ]
サポートされている UNIX および Mac OS X プラットフォームに DBD::SQLAnywhere インタフェースをインストールし、Perl を使用してデータベースにアクセスします。

DBD::SQLAnywhere を使用する Perl スクリプト [437 ページ]
DBD::SQLAnywhere インタフェースを使用する Perl スクリプトを記述できます。DBD::SQLAnywhere は、Perl DBI モジュールのドライバです。

1.14.1 DBD::SQLAnywhere

DBD::SQLAnywhere は、Perl 用の Database Independent Interface (DBI) モジュールのドライバです。Perl DBI モジュールと DBD::SQLAnywhere をインストールすると、Perl から SQL Anywhere データベースの情報にアクセスして変更できるようになります。

DBD::SQLAnywhere ドライバは、ithread が採用された Perl を使用するときにスレッドに対応します。

要件

DBD::SQLAnywhere インタフェースには、次のコンポーネントが必要です。

- Perl 5.6.0 以降。Windows では、ActivePerl 5.6.0 ビルド 616 以降が必要です。
- Perl DBI 1.34 以降。
- C コンパイラ。Windows では、Microsoft Visual C++ コンパイラのみがサポートされています。

1.14.2 Windows での DBD::SQLAnywhere のインストール

サポートされている Windows プラットフォームに DBD::SQLAnywhere インタフェースをインストールし、Perl を使用してデータベースにアクセスします。

前提条件

- ActivePerl 5.6.0 以降。
- (オプション) ActivePerl 5.16 以前に対応する DBD::SQLAnywhere ドライバをビルドする場合は、Microsoft Visual Studio が必要となります。ActivePerl 5.18 以降の場合は必要ありません。

手順

1. ActiveState のビルドが正しく動作している場合、そのリポジトリからドライバをインストールできます。コマンドプロンプトで、ActivePerl のインストールディレクトリの `bin` サブディレクトリに移動し、次のコマンドを実行します。

   ```
   ppm install DBD-SQLAnywhere
   ```

   ドライバが正しくインストールされれば終了です。
2. ActivePerl 5.18 以降を使用して自分でドライバをビルドする場合、次の手順を実行します (それ以前のバージョンの ActivePerl を使用する場合は後ろの説明を参照)。
   a. コマンドプロンプトで、書き込み可能な新しいディレクトリに、SQL Anywhere インストール環境の `SDK¥Perl` ディレクトリのコピーを作成し、このディレクトリに移動します。
   b. MinGW Perl パッケージをインストールします。

      ```
      ppm install MinGW
      ```

   c. Perl の `site¥bin` ディレクトリが PATH に追加されていない場合、追加します。
   d. コマンドプロンプトで次のコマンドを実行します。

      ```
      perl Makefile.PL
      dmake
      ```

e. DBD::SQLAnywhere をテストするには、サンプルデータベースファイルをカレントディレクトリにコピーして、テストを実行します。

```
demo.db
dmake test
```

テストが行われない場合は、ソフトウェアインストール環境の bin32 または bin64 サブディレクトリがパスに含まれていることを確認してください。

f. インストールを完了するには、同じプロンプトで次のコマンドを実行します。

```
dmake install
```

何らかの理由によって最初から作業をやり直す必要がある場合は、コマンド **dmake clean** を実行すると、部分的にビルドされたターゲットを削除できます。

3. ActivePerl 5.16 以前を使用して自分でドライバをビルドする場合、次の手順を実行します。

注意:Microsoft C が必要です。

a. Perl の site¥bin ディレクトリが PATH に追加されていない場合、追加します。

b. コマンドプロンプトで、書き込み可能な新しいディレクトリに、SQL Anywhere インストール環境の SDK¥Perl ディレクトリのコピーを作成し、このディレクトリに移動します。

c. Microsoft Visual Studio 用に、環境変数 PATH、LIB、および INCLUDE を正しく設定してください。Microsoft は、このためのバッチファイルを用意しています。32 ビットのビルドの場合、Microsoft Visual Studio のインストール環境の vc¥bin サブディレクトリに vcvars32.bat というバッチファイルが格納されています。64 ビットのビルドの場合は、このバッチファイルの 64 ビットバージョン (vcvars64.bat または vcvarsamd64.bat など) を探します。

d. Perl DBI がインストールされていない場合は、インストールしてください。そのためには、次のコマンドを実行します。

```
ppm install dbi
```

e. コマンドプロンプトで次のコマンドを実行します。

```
perl Makefile.PL
nmake
```

Microsoft Visual Studio や ActivePerl の古いバージョンを使用している場合で、マニフェストに関するエラーが出る場合、Makefile.PL 内の以下の行をコメント解除し、'perl Makefile.PL' から再度試してください。

```
#    $opts{LD} = "¥$(PERL) dolink.pl ¥$@";
```

生成された Makefile を手動で編集しないでください。それらの変更は、'perl Makefile.PL' を次に実行したときに失われてしまうからです。変更は必ず Makefile.PL のコマンドラインから、または Makefile.PL ファイルを編集することによって行ってください。

f. DBD::SQLAnywhere をテストするには、demo.db を SQL Anywhere のインストールディレクトリからカレントディレクトリにコピーします。

```
demo.db
nmake test
```

テストが行われない場合は、ソフトウェアインストール環境の bin32 または bin64 サブディレクトリがパスに含まれていることを確認してください。

g. インストールを完了するには、同じプロンプトで次のコマンドを実行します。

```
nmake install
```

何らかの理由によって最初から作業をやり直す必要がある場合は、コマンド **nmake clean** を実行すると、部分的にビルドされたターゲットを削除できます。

結果

Perl DBI モジュールと DBD::SQLAnywhere インタフェースが使用できるようになりました。

関連情報

方法:64 ビットの Visual C++ ツールセットをコマンドラインから有効にする
Perl DBI

1.14.3 UNIX と Mac OS X での DBD::SQLAnywhere のインストール

サポートされている UNIX および Mac OS X プラットフォームに DBD::SQLAnywhere インタフェースをインストールし、Perl を使用してデータベースにアクセスします。

前提条件

- Perl 5 (5.6.0 以上を推奨) のビルド、テスト、およびインストール。

  > **注記**
  >
  > Perl 5 のインストールとテストは非常に重要です。

- DBI モジュール (DBI 1.51 を推奨) のビルド、テスト、およびインストール (Perl のディストリビューションに含まれていない場合)。
- DBI モジュールのビルドを行うのに、SQL Anywhere ソフトウェアのインストールは必要ありません。ただし、モジュールを使用するには、SQL Anywhere クライアントをインストールしておく必要があります。
- DBI モジュールのテストを行うには、SQL Anywhere のクライアントおよびサーバをフルインストールしておくことを推奨します。インストールされていない場合、テストはスキップされます。

コンテキスト

これらの手順は、Linux、Solaris、AIX、HP-UX、および Mac OS X に適用されます。

手順

1. DBI モジュールが Perl のディストリビューションに含まれていない場合、CPAN (Comprehensive Perl Archive Network) から Perl DBI モジュールのソースをダウンロードします。
   a. このファイルの内容を新しいディレクトリに抽出します。
   b. コマンドプロンプトで、新しいディレクトリに変更し、次のコマンドを実行して Perl DBI モジュールをビルドします。

      ```
      perl Makefile.PL
      make
      ```

   c. 以下のファイルを SQL Anywhere のインストールディレクトリからカレントディレクトリにコピーします。

      ```
      demo.db
      ```

   d. 次のコマンドを使用して、Perl DBI モジュールをテストします。

      ```
      make test
      ```

   e. インストールを完了するには、同じプロンプトで次のコマンドを実行します。

      ```
      make install
      ```

      何らかの理由によって最初から作業をやり直す必要がある場合は、コマンド **`make clean`** を使用すれば、部分的にビルドされたターゲットを削除できます。
2. データベースサーバを実行するために環境が設定されていることを確認します。使用しているシェルに応じて適切なコマンドを入力して、ソフトウェアのインストールディレクトリから SQL Anywhere の設定スクリプトのコマンドを実行します。

   Bourne シェルとその派生シェル

   Bourne シェルとその派生シェルでは、このコマンド名は . (単一のピリオド) です。たとえば、SQL Anywhere が */opt/sqlanywhere17* にインストールされている場合、次の文を使用して `sa_config.sh` のソースを指定します (32 ビット環境では `bin32` を使用してください)。

   ```
   . /opt/sqlanywhere17/bin64/sa_config.sh
   ```

   Mac の場合は、次のコマンドを実行して `sa_config.sh` のソースを指定します。

   ```
   . /Applications/SQLAnywhere17/System/bin64/sa_config.sh
   ```

   C シェルとその派生シェル

   C シェルとその派生シェルの場合、コマンドは `source` です。たとえば、SQL Anywhere が */opt/sqlanywhere17* にインストールされている場合、次のコマンドを使用して `sa_config.csh` のソースを指定します (32 ビット環境では `bin32` を使用してください)。

   ```
   source  /opt/sqlanywhere17/bin64/sa_config.csh
   ```

> **注記**
>
> Mac OS X 10.11 上では、SQLANY_API_DLL 環境変数を `libdbcapi_r.dylib` へのフルパスに設定します。

3. シェルプロンプトで、書き込み可能な新しいディレクトリに、SQL Anywhere インストール環境の `sdk/perl` ディレクトリのコピーを作成し、このディレクトリに移動します。
4. 次のコマンドを実行して DBD::SQLAnywhere をビルドします。

   ```
   perl Makefile.PL
   make
   ```

5. DBD::SQLAnywhere をテストするには、サンプルデータベースファイルをカレントディレクトリにコピーして、テストを実行します。

   ```
   newdemo.sh
   make test
   ```

   テストが行われない場合は、ソフトウェアインストール環境の `bin32` または `bin64` サブディレクトリがパスに含まれていることを確認してください。
6. インストールを完了するには、同じプロンプトで次のコマンドを実行します。

   ```
   make install
   ```

   何らかの理由によって最初から作業をやり直す必要がある場合は、コマンド **`make clean`** を使用すれば、部分的にビルドされたターゲットを削除できます。

結果

Perl DBI モジュールと DBD::SQLAnywhere インタフェースが使用できるようになりました。

次のステップ

Perl DBI のソースツリーや `sdk/perl` ディレクトリのコピーは削除しても構いません。これらはもう必要ありません。

関連情報

Perl の包括的なアーカイブネットワーク

1.14.4 DBD::SQLAnywhere を使用する Perl スクリプト

DBD::SQLAnywhere インタフェースを使用する Perl スクリプトを記述できます。DBD::SQLAnywhere は、Perl DBI モジュールのドライバです。

このセクションの内容:

Perl DBI モジュール [437 ページ]
Perl スクリプトから DBD::SQLAnywhere インタフェースを使用するには、Perl DBI モジュールを使用することを最初に Perl に通知する必要があります。これを行うには、ファイルの先頭に次の行を挿入します。

Perl DBI を使用してデータベース接続を開いたり閉じたりする方法 [438 ページ]
通常、データベースに対して 1 つの接続を開いてから、一連の SQL 文を実行して必要なすべての操作を実行します。

Perl DBI を使用して結果セットを取得する方法 [438 ページ]
開かれた接続へのハンドルを取得したら、データベースに格納されているデータにアクセスして修正できます。最も単純な操作は、おそらくいくつかのローを取得して出力することです。

Perl DBI を使用して複数の結果セットを処理する方法 [439 ページ]
クエリからの複数の結果セットを処理するメソッドでは、結果セット間を移動する別のループの中でフェッチループを囲い込みます。

Perl DBI を使用してローを挿入する方法 [440 ページ]
ローを挿入するには、開かれた接続へのハンドルが必要です。最も簡単な方法は、パラメータ化された INSERT 文を使用する方法です。この場合、疑問符が値のプレースホルダとして使用されます。

1.14.4.1 Perl DBI モジュール

Perl スクリプトから DBD::SQLAnywhere インタフェースを使用するには、Perl DBI モジュールを使用することを最初に Perl に通知する必要があります。これを行うには、ファイルの先頭に次の行を挿入します。

```
use DBI;
```

また、Perl を厳密モードで実行することを強くおすすめします。明示的な変数定義を必須とするこの文によって、印刷上のエラーなどの一般的なエラーが原因の不可解なエラーが発生する可能性を大幅に減らせる見込みがあります。

```
#!/usr/local/bin/perl -w
#
use DBI;
use strict;
```

Perl DBI モジュールは、必要に応じて DBD ドライバ (DBD::SQLAnywhere など) を自動的にロードします。

1.14.4.2 Perl DBI を使用してデータベース接続を開いたり閉じたりする方法

通常、データベースに対して 1 つの接続を開いてから、一連の SQL 文を実行して必要なすべての操作を実行します。

接続を開くには、*connect* メソッドを使用します。戻り値は、接続時に後続の操作を行うために使用するデータベース接続のハンドルです。

connect メソッドのパラメータは、次のとおりです。

1. "DBI:SQLAnywhere:" とセミコロンで分けられた追加接続パラメータ。
2. ユーザ名。この文字列がブランクでないかぎり、接続文字列に ";UID=value" が追加されます。
3. パスワード値。この文字列がブランクでないかぎり、接続文字列に ";PWD=value" が追加されます。
4. デフォルト値のハッシュへのポインタ。AutoCommit、RaiseError、PrintError などの設定は、この方法で設定できます。

次のコードサンプルは、サンプルデータベースへの接続を開いて閉じます。このスクリプトを実行するには、データベースサーバとサンプルデータベースを事前に起動しておく必要があります。

```
#!/usr/local/bin/perl -w
#
use DBI;
use strict;
my $database = "demo";
my $data_src = "DBI:SQLAnywhere:SERVER=$database;DBN=$database";
my $uid      = "DBA";
my $pwd      = "sql";
my %defaults = (
     AutoCommit => 1, # Autocommit enabled.
     PrintError => 0  # Errors not automatically printed.
   );
my $dbh = DBI->connect($data_src, $uid, $pwd, ¥%defaults)
  or die "Cannot connect to $data_src: $DBI::errstr¥n";
$dbh->disconnect;
exit(0);
__END__
```

オプションとして、ユーザ名またはパスワード値を個々のパラメータとして指定する代わりに、これらをデータソース文字列に追加できます。このオプションを実施する場合は、該当する引数にブランク文字列を指定します。たとえば、上記の場合、接続を開く文を次の文に置き換えることにより、スクリプトを変更できます。

```
$data_src .= ";UID=$uid";
$data_src .= ";PWD=$pwd";
my $dbh = DBI->connect($data_src, '', '', ¥%defaults)
  or die "Cannot connect to $data_src: $DBI::errstr¥n";
```

1.14.4.3 Perl DBI を使用して結果セットを取得する方法

開かれた接続へのハンドルを取得したら、データベースに格納されているデータにアクセスして修正できます。最も単純な操作は、おそらくいくつかのローを取得して出力することです。

ローのセットを返す SQL 文は、先に準備してから実行する必要があります。*prepare* メソッドは、文のハンドルを返します。このハンドルを使用して文を実行し、結果セットに関するメタ情報と、結果セットのローを取得できます。

```
#!/usr/local/bin/perl -w
```

```
#
use DBI;
use strict;
my $database = "demo";
my $data_src = "DBI:SQLAnywhere:SERVER=$database;DBN=$database";
my $uid      = "DBA";
my $pwd      = "sql";
my $sel_stmt = "SELECT ID, GivenName, Surname
                FROM Customers
                ORDER BY GivenName, Surname";
my %defaults = (
     AutoCommit => 0, # Require explicit commit or rollback.
     PrintError => 0
   );
my $dbh = DBI->connect($data_src, $uid, $pwd, ¥%defaults)
  or die "Cannot connect to $data_src: $DBI::errstr¥n";
&db_query($sel_stmt, $dbh);
$dbh->rollback;
$dbh->disconnect;
exit(0);
sub db_query {
      my($sel, $dbh) = @_;
      my($row, $sth) = undef;
      $sth = $dbh->prepare($sel);
      $sth->execute;
      print "Fields:     $sth->{NUM_OF_FIELDS}¥n";
      print "Params:     $sth->{NUM_OF_PARAMS}¥n¥n";
      print join("¥t¥t", @{$sth->{NAME}}), "¥n¥n";
      while($row = $sth->fetchrow_arrayref) {
         print join("¥t¥t", @$row), "¥n";
      }
      $sth = undef;
}
__END__
```

準備文は、Perl 文のハンドルが破棄されないかぎりデータベースサーバから削除されません。文のハンドルを破棄するには、変数を再使用するか、変数を undef に設定します。finish メソッドを呼び出してもハンドルは削除されません。実際には、結果セットの読み込みを終了しないと決定した場合を除いて、finish メソッドは呼び出さないようにしてください。

ハンドルのリークを検出するために、データベースサーバでは、カーソルと準備文の数はデフォルトで接続ごとに最大 50 に制限されています。これらの制限を越えると、リソースガバナーによってエラーが自動的に生成されます。このエラーが発生したら、破棄されていない文のハンドルを確認してください。文のハンドルが破棄されていない場合は、prepare_cached を慎重に使用してください。

必要な場合、max_cursor_count と max_statement_count オプションを設定してこれらの制限を変更できます。

1.14.4.4 Perl DBI を使用して複数の結果セットを処理する方法

クエリからの複数の結果セットを処理するメソッドでは、結果セット間を移動する別のループの中でフェッチループを囲い込みます。

実行する前に、複数の結果セットを返す SQL 文を準備しておく必要があります。*prepare* メソッドは、文のハンドルを返します。このハンドルを使用して文を実行し、結果セットに関するメタ情報と、それぞれの結果セットのローを取得します。

```
#!/usr/local/bin/perl -w
#
use DBI;
use strict;
my $database = "demo";
```

```
my $data_src = "DBI:SQLAnywhere:SERVER=$database;DBN=$database";
my $uid      = "DBA";
my $pwd      = "sql";
my $sel_stmt = "SELECT ID, GivenName, Surname
                  FROM Customers
                  ORDER BY GivenName, Surname;
                SELECT *
                  FROM Departments
                  ORDER BY DepartmentID";
my %defaults = (
     AutoCommit => 0, # Require explicit commit or rollback.
     PrintError => 0
   );
my $dbh = DBI->connect($data_src, $uid, $pwd, ¥%defaults)
  or die "Cannot connect to $data_src: $DBI::errstr¥n";
&db_query($sel_stmt, $dbh);
$dbh->rollback;
$dbh->disconnect;
exit(0);
sub db_query {
      my($sel, $dbh) = @_;
      my($row, $sth) = undef;
      $sth = $dbh->prepare($sel);
      $sth->execute;
      do {
          print "Fields:     $sth->{NUM_OF_FIELDS}¥n";
          print "Params:     $sth->{NUM_OF_PARAMS}¥n¥n";
          print join("¥t¥t", @{$sth->{NAME}}), "¥n¥n";
          while($row = $sth->fetchrow_arrayref) {
             print join("¥t¥t", @$row), "¥n";
          }
          print "---end of results---¥n¥n";
      } while (defined $sth->more_results);
      $sth = undef;
}
__END__
```

1.14.4.5 Perl DBI を使用してローを挿入する方法

ローを挿入するには、開かれた接続へのハンドルが必要です。最も簡単な方法は、パラメータ化された INSERT 文を使用する方法です。この場合、疑問符が値のプレースホルダとして使用されます。

この文は最初に準備されてから、新しいローごとに 1 回実行されます。新しいローの値は、execute メソッドのパラメータとして指定されます。

次のサンプルプログラムは、2 人の新しい顧客を挿入します。ローの値はリテラル文字列として表示されますが、これらの値はファイルから読み込むことができます。

```
#!/usr/local/bin/perl -w
#
use DBI;
use strict;
my $database = "demo";
my $data_src = "DBI:SQLAnywhere:SERVER=$database;DBN=$database";
my $uid      = "DBA";
my $pwd      = "sql";
my $ins_stmt = "INSERT INTO Customers (ID, GivenName, Surname,
                        Street, City, State, Country, PostalCode,
                        Phone, CompanyName)
                  VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)";
my %defaults = (
```

```
      AutoCommit => 0, # Require explicit commit or rollback.
      PrintError => 0
    );
my $dbh = DBI->connect($data_src, $uid, $pwd, ¥%defaults)
      or die "Can't connect to $data_src: $DBI::errstr¥n";
&db_insert($ins_stmt, $dbh);
$dbh->commit;
$dbh->disconnect;
exit(0);
sub db_insert {
       my($ins, $dbh) = @_;
       my($sth) = undef;
       my @rows = (
         "801,Alex,Alt,5 Blue Ave,New York,NY,USA,10012,5185553434,BXM",
         "802,Zach,Zed,82 Fair St,New York,NY,USA,10033,5185552234,Zap"
        );
       $sth = $dbh->prepare($ins);
       my $row = undef;
       foreach $row ( @rows ) {
         my @values = split(/,/, $row);
         $sth->execute(@values);
       }
}
__END__
```

1.15 Python およびデータベースアクセス

Python 拡張モジュール sqlanydb、sqlany-django、sqlalchemy-sqlany は、データベースに接続し、SQL クエリを発行し、結果セットを取得するときに Python の使用をサポートします。

Python データベース API 仕様 v2.0 (PEP 249) は、実際に使用されているデータベースとは関係なく一貫したデータベースインタフェースを提供する一連のメソッドを定義します。Python 拡張モジュール sqlanydb は PEP 249 を実装します。sqlanydb モジュールをインストールすると、Python からデータベースの情報にアクセスして変更できるようになります。

詳細については、PEP 249 -- Python データベース API 仕様 v2.0 を参照してください。

Python 拡張モジュール sqlany-django を使用すると、データベースサーバを Django ベース Web サイトのバックエンドとして使用することができます。

Python 拡張モジュール sqlalchemy-sqlany を使用すると、データベースサーバとの通信が可能な SQLAlchemy アプリケーションを登録することができます。

このセクションの内容:

Windows での sqlanydb のインストール [442 ページ]
Windows で Python サポートを設定するには、ソフトウェアインストールの `SDK¥Python` サブディレクトリから適用可能な設定スクリプトを実行します。

UNIX と Mac OS X での sqlanydb のインストール [443 ページ]
UNIX および Mac OS X で Python サポートを設定するには、ソフトウェアインストールの `sdk/python` サブディレクトリから適用可能な設定スクリプトを実行します。

Django ドライバ (sqlany-django) のインストール [445 ページ]

Django は、Web サイトを作成するための Python ベースのフレームワークです。sqlany-django ドライバを使用すると、SQL Anywhere データベースサーバを Django ベース Web サイトのバックエンドとして使用することができます。

SQLAlchemy ダイアレクト (sqlalchemy-sqlany) のインストール [445 ページ]
SQLAlchemy は、Python ベースのツールキットで、オブジェクトリレーショナルマッパーです。sqlalchemy-sqlany ダイアレクトを使用すると、SQL Anywhere データベースサーバと通信する SQLAlchemy アプリケーションを登録することができます。

sqlanydb を使用する Python スクリプト [446 ページ]
次の資料では、sqlanydb インタフェースを使用する Python スクリプトを作成する方法の概要を説明します。

関連情報

SQL Anywhere クライアントインタフェース
PEP 249 - Python データベース API 仕様

1.15.1 Windows での sqlanydb のインストール

Windows で Python サポートを設定するには、ソフトウェアインストールの `SDK¥Python` サブディレクトリから適用可能な設定スクリプトを実行します。

前提条件

Python がインストールされていることを確認します。

ctypes モジュールは必須です。ctypes モジュールが存在しているかどうかをテストするには、Python を実行します。Python プロンプトで次の文を入力します。

```
import ctypes
```

エラーメッセージが表示された場合は、ctypes がインストールされていません。次はその例です。

```
>>> import ctypes
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ImportError: No module named ctypes
```

Python のインストール環境に ctypes が含まれていない場合は、インストールしてください。インストールは、SourceForge.net Web サイト () にあります。

ctypes は Peak EasyInstall でもインストールされます。

手順

1. 管理者権限で実行したコマンドプロンプトで、ソフトウェアインストール環境の `SDK¥Python` サブディレクトリに移動します。
2. 次のコマンドを実行して sqlanydb をインストールします。

   ```
   python setup.py install
   ```

3. 次のコマンドを実行し、サンプルデータベースファイルのコピーを現在のディレクトリに作成して、sqlanydb をテストします。

   ```
   newdemo
   dbsrv17 demo.db
   python Scripts¥test.py
   ```

   テストスクリプトによりデータベースサーバに接続され、SQL クエリが実行されます。テストが成功した場合、`sqlanydb successfully installed.` というメッセージが表示されます。

   テストが行われない場合は、ソフトウェアインストール環境の `bin32` または `bin64` サブディレクトリがパスに含まれていることを確認してください。

   サンプルデータベースとログファイルを現在のディレクトリから削除できます。

結果

これで、sqlanydb モジュールが使用可能になりました。

関連情報

SQL Anywhere クライアントインタフェース
Peak EasyInstall ダウンロード

1.15.2 UNIX と Mac OS X での sqlanydb のインストール

UNIX および Mac OS X で Python サポートを設定するには、ソフトウェアインストールの `sdk/python` サブディレクトリから適用可能な設定スクリプトを実行します。

前提条件

Python がインストールされていることを確認します。

ctypes モジュールは必須です。ctypes モジュールが存在しているかどうかをテストするには、Python を実行します。Python プロンプトで次の文を入力します。

```
import ctypes
```

エラーメッセージが表示された場合は、ctypes がインストールされていません。次はその例です。

```
>>> import ctypes
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ImportError: No module named ctypes
```

Python のインストール環境に ctypes が含まれていない場合は、インストールしてください。インストールは、SourceForge.net Web サイト () にあります。

ctypes は Peak EasyInstall でもインストールされます。

手順

1. データベースサーバを実行するために環境が設定されていることを確認します。使用しているシェルに応じて適切なコマンドを入力して、ソフトウェアのインストールディレクトリから SQL Anywhere の設定スクリプトのコマンドを実行します。

   Bourne シェルとその派生シェル

   Bourne シェルとその派生シェルでは、このコマンド名は . (単一のピリオド) です。たとえば、SQL Anywhere が */opt/sqlanywhere17* にインストールされている場合、次の文を使用して `sa_config.sh` のソースを指定します (32 ビット環境では `bin32` を使用してください)。

   ```
   .  /opt/sqlanywhere17/bin64/sa_config.sh
   ```

   Mac の場合は、次のコマンドを実行して `sa_config.sh` のソースを指定します。

   ```
   . /Applications/SQLAnywhere17/System/bin64/sa_config.sh
   ```

   C シェルとその派生シェル

   C シェルとその派生シェルの場合、コマンドは `source` です。たとえば、SQL Anywhere が */opt/sqlanywhere17* にインストールされている場合、次のコマンドを使用して `sa_config.csh` のソースを指定します (32 ビット環境では `bin32` を使用してください)。

   ```
   source  /opt/sqlanywhere17/bin64/sa_config.csh
   ```

   > **注記**
   >
   > Mac OS X 10.11 を使用している場合は、SQLANY_API_DLL 環境変数を `libdbcapi_r.dylib` へのフルパスに設定する必要があります。

2. シェルプロンプトで、ソフトウェアインストール環境の `sdk/python` サブディレクトリに移動します。
3. 次のコマンドを入力して sqlanydb をインストールします。

   ```
   python setup.py install
   ```

4. 次のコマンドを実行し、サンプルデータベースファイルのコピーを現在のディレクトリに作成して、sqlanydb をテストします。

```
newdemo
dbsrv17 demo.db
python scripts/test.py
```

テストスクリプトによりデータベースサーバに接続され、SQL クエリが実行されます。テストが成功した場合、`sqlanydb successfully installed.` というメッセージが表示されます。

テストが行われない場合は、ソフトウェアインストール環境の `bin32` または `bin64` サブディレクトリがパスに含まれていることを確認してください。

サンプルデータベースとログファイルを現在のディレクトリから削除できます。

結果

これで、sqlanydb モジュールが使用可能になりました。

関連情報

SQL Anywhere クライアントインタフェース
Peak EasyInstall ダウンロード

1.15.3 Django ドライバ (sqlany-django) のインストール

Django は、Web サイトを作成するための Python ベースのフレームワークです。sqlany-django ドライバを使用すると、SQL Anywhere データベースサーバを Django ベース Web サイトのバックエンドとして使用することができます。

SQL Anywhere Django ドライバの最新のソフトウェアおよびマニュアルは、PyPI (Python Package Index) Web サイト (https://pypi.python.org/pypi) で入手することができます。`sqlany-django` を検索します。

このドライバは、https://github.com/sqlanywhere/sqlany-django/ から入手することもできます。Django ドライバは、SQL Anywhere Python ドライバを必要とします。これは、SQL Anywhere インストールに含まれていますが、https://github.com/sqlanywhere/sqlanydb から入手することもできます。

1.15.4 SQLAlchemy ダイアレクト (sqlalchemy-sqlany) のインストール

SQLAlchemy は、Python ベースのツールキットで、オブジェクトリレーショナルマッパーです。sqlalchemy-sqlany ダイアレクトを使用すると、SQL Anywhere データベースサーバと通信する SQLAlchemy アプリケーションを登録することができます。

SQL Anywhere SQLAlchemy ダイアレクトの最新のソフトウェアおよびマニュアルは、PyPI (Python Package Index) Web サイト (https://pypi.python.org/pypi) で入手することができます。`sqlalchemy-sqlany` を検索します。

このダイアレクトは、https://github.com/sqlanywhere/sqlalchemy-sqlany から入手することもできます。SQLAlchemy ドライバは、SQL Anywhere Python ドライバを必要とします。これは、SQL Anywhere インストールに含まれていますが、https://github.com/sqlanywhere/sqlanydb から入手することもできます。

1.15.5 sqlanydb を使用する Python スクリプト

次の資料では、sqlanydb インタフェースを使用する Python スクリプトを作成する方法の概要を説明します。

このセクションの内容:

sqlanydb モジュール [446 ページ]
sqlanydb モジュールを Python スクリプトから使用するには、ファイルの先頭に *import* 文を配置して、最初に sqlanydb モジュールをロードする必要があります。

Python を使用してデータベース接続を開いたり閉じたりする方法 [447 ページ]
通常、データベースに対して 1 つの接続を開いてから、一連の SQL 文を実行して必要なすべての操作を実行します。

Python を使用して結果セットを取得する方法 [448 ページ]
開かれた接続へのハンドルを取得したら、データベースに格納されているデータにアクセスして修正できます。最も単純な操作は、おそらくいくつかのローを取得して出力することです。

Python を使用してローを挿入する方法 [448 ページ]
ローをテーブルに挿入する最も簡単な方法は、パラメータ化されていない INSERT 文を使用することです。この方法では、値は SQL 文の一部として指定されます。

Python でのデータベースの型変換 [449 ページ]
変換コールバックを登録すると、データベースサーバから結果がフェッチされるときに、Python オブジェクトへデータベースの型がマッピングされる方法を制御できます。

関連情報

Web サービスへの変数の指定 [645 ページ]

1.15.5.1 sqlanydb モジュール

sqlanydb モジュールを Python スクリプトから使用するには、ファイルの先頭に *import* 文を配置して、最初に sqlanydb モジュールをロードする必要があります。

```
import sqlanydb
```

Python でスレッドを使用している場合、sqlanydb モジュールはスレッド対応になります。

1.15.5.2 Python を使用してデータベース接続を開いたり閉じたりする方法

通常、データベースに対して 1 つの接続を開いてから、一連の SQL 文を実行して必要なすべての操作を実行します。

接続を開くには、*connect* メソッドを使用します。この戻り値は、接続時に後続の操作を行うために使用するデータベース接続のハンドルです。

connect メソッドのパラメータは、一連の "キーワード=値" ペアをカンマで区切って指定します。キーワードは、大文字と小文字を区別しません。

```
sqlanydb.connect( keyword=value, ... )
```

一般的な接続パラメータを次に示します。

DataSourceName="dsn"

この接続パラメータの省略形は DSN="dsn" です。たとえば、DataSourceName="SQL Anywhere 17 Demo" と指定します。

UserID="user-id"

この接続パラメータの省略形は UID="user-id" です。たとえば、UserID="DBA" と指定します。

Password="passwd"

この接続パラメータの省略形は PWD="passwd" です。たとえば、Password="sql" と指定します。

DatabaseFile="db-file"

この接続パラメータの省略形は DBF="db-file" です。たとえば、DatabaseFile="demo.db" と指定します。

次のコードサンプルは、サンプルデータベースへの接続を開いて閉じます。このスクリプトを実行するには、サンプルデータベースを備えたデータベースサーバを事前に起動しておく必要があります。

```
import sqlanydb
myuid = raw_input("Enter your user ID: ")
mypwd = raw_input("Enter your password: ")
# Create a connection object
con = sqlanydb.connect( userid=myuid, password=mypwd )
# Close the connection
con.close()
```

データベースサーバが手動で開始されないようにするには、データベースを備えたサーバを起動するように設定されたデータソースを使います。この例を次に示します。

```
import sqlanydb
mydsn = "SQL Anywhere 17 Demo"
myuid = raw_input("Enter your user ID: ")
mypwd = raw_input("Enter your password: ")
# Create a connection object
con = sqlanydb.connect( dsn=mydsn, userid=myuid, password=mypwd )
# Close the connection
con.close()
```

関連情報

Python データベース API 仕様

1.15.5.3 Python を使用して結果セットを取得する方法

開かれた接続へのハンドルを取得したら、データベースに格納されているデータにアクセスして修正できます。最も単純な操作は、おそらくいくつかのローを取得して出力することです。

cursor メソッドは、開いている接続に対してカーソルを作成するために使用します。*execute* メソッドは、結果セットを作成するために使用します。*fetchall* メソッドは、この結果セットからローを取得するために使用します。

```
import sqlanydb
myuid = raw_input("Enter your user ID: ")
mypwd = raw_input("Enter your password: ")
# Create a connection object, then use it to create a cursor
con = sqlanydb.connect( userid=myuid, password=mypwd )
cursor = con.cursor()
# Execute a SQL string
sql = "SELECT * FROM Employees"
cursor.execute(sql)
# Get a cursor description which contains column names
desc = cursor.description
print len(desc)
# Fetch all results from the cursor into a sequence,
# display the values as column name=value pairs,
# and then close the connection
rowset = cursor.fetchall()
for row in rowset:
    for col in range(len(desc)):
        print "%s=%s" % (desc[col][0], row[col] )
    print
cursor.close()
con.close()
```

1.15.5.4 Python を使用してローを挿入する方法

ローをテーブルに挿入する最も簡単な方法は、パラメータ化されていない INSERT 文を使用することです。この方法では、値は SQL 文の一部として指定されます。

新しい文が新しいローごとに構築されて実行されます。前の例でみたように、SQL 文を実行するにはカーソルが必要です。

次のサンプルプログラムは、2 人の新規顧客をサンプルデータベースに挿入します。切断される前に、データベースに対してトランザクションをコミットします。

```
import sqlanydb
myuid = raw_input("Enter your user ID: ")
mypwd = raw_input("Enter your password: ")
# Create a connection object, then use it to create a cursor
con = sqlanydb.connect( userid=myuid, pwd=mypwd )
cursor = con.cursor()
cursor.execute("DELETE FROM Customers WHERE ID > 800")
rows = ((801,'Alex','Alt','5 Blue Ave','New York','NY',
```

```
        'USA','10012','5185553434','BXM'),
        (802,'Zach','Zed','82 Fair St','New York','NY',
        'USA','10033','5185552234','Zap'))
# Set up a SQL INSERT
parms = ("'%s'," * len(rows[0]))[:-1]
sql = "INSERT INTO Customers VALUES (%s)" % (parms)
print sql % rows[0]
cursor.execute(sql % rows[0])
print sql % rows[1]
cursor.execute(sql % rows[1])
cursor.close()
con.commit()
con.close()
```

パラメータ化された INSERT 文を使用してローをテーブルに挿入する方法もあります。この場合、疑問符が値のプレースホルダとして使用されます。executemany メソッドは、ローのセットのメンバーごとに INSERT 文を実行するために使用します。新しいローの値は、1 つの引数として executemany メソッドに渡されます。

```
import sqlanydb
myuid = raw_input("Enter your user ID: ")
mypwd = raw_input("Enter your password: ")
# Create a connection object, then use it to create a cursor
con = sqlanydb.connect( userid=myuid, pwd=mypwd )
cursor = con.cursor()
cursor.execute("DELETE FROM Customers WHERE ID > 800")
rows = ((801,'Alex','Alt','5 Blue Ave','New York','NY',
        'USA','10012','5185553434','BXM'),
        (802,'Zach','Zed','82 Fair St','New York','NY',
        'USA','10033','5185552234','Zap'))
# Set up a parameterized SQL INSERT
parms = ("?," * len(rows[0]))[:-1]
sql = "INSERT INTO Customers VALUES (%s)" % (parms)
print sql
cursor.executemany(sql, rows)
cursor.close()
con.commit()
con.close()
```

どちらのサンプルプログラムも、ローのデータをテーブルに挿入する方法として適切であるようですが、いくつかの理由で後者の方が優れています。入力を要求されてデータ値が取得された場合、最初のサンプルプログラムでは SQL 文を含む不良データが挿入される可能性があります。最初のサンプルプログラムでは、execute メソッドはテーブルに挿入されるローごとに呼び出されます。2 番目のサンプルプログラムでは、すべてのローをテーブルに挿入するときに一度だけ executemany メソッドが呼び出されます。

1.15.5.5 Python でのデータベースの型変換

変換コールバックを登録すると、データベースサーバから結果がフェッチされるときに、Python オブジェクトへデータベースの型がマッピングされる方法を制御できます。

コールバックの登録には、モジュールレベルの register_converter メソッドを使用できます。このメソッドは、データベースタイプを最初のパラメータ、変換関数を 2 番目のパラメータとして呼び出されます。たとえば、データ型 DT_DECIMAL として記述されるカラムのデータに対して Decimal オブジェクトを作成するよう sqlanydb に要求する場合は、次の例を使用します。

```
import sqlanydb
import decimal
def convert_to_decimal(num):
    return decimal.Decimal(num)
```

```
sqlanydb.register_converter(sqlanydb.DT_DECIMAL, convert_to_decimal)
```

デフォルトでは、データ型 DT_BIT のカラムが整数型に変換されます。データ型 DT_BIT として記述されるカラムのデータに対して boolean オブジェクトを作成するよう sqlanydb に要求する場合は、次の例を使用します。

```
import sqlanydb
def convert_to_boolean(num):
    return num != 0
sqlanydb.register_converter(sqlanydb.DT_DECIMAL, convert_to_decimal)
```

コンバーターは、次のデータベースタイプに登録できます。

```
DT_DATE
DT_TIME
DT_TIMESTAMP
DT_VARCHAR
DT_FIXCHAR
DT_LONGVARCHAR
DT_STRING
DT_DOUBLE
DT_FLOAT
DT_DECIMAL
DT_INT
DT_SMALLINT
DT_BINARY
DT_LONGBINARY
DT_TINYINT
DT_BIGINT
DT_UNSINT
DT_UNSSMALLINT
DT_UNSBIGINT
DT_BIT
```

次の例は、小数の結果を、小数点以下の桁数をトランケートして整数に変換する方法を示します。アプリケーションの実行時には、給与額が整数値で表示されます。

```
import sqlanydb
def convert_to_int(num):
    return int(float(num))
sqlanydb.register_converter(sqlanydb.DT_DECIMAL, convert_to_int)
myuid = raw_input("Enter your user ID: ")
mypwd = raw_input("Enter your password: ")
# Create a connection object, then use it to create a cursor
con = sqlanydb.connect( userid=myuid, password=mypwd )
cursor = con.cursor()
# Execute a SQL string
sql = "SELECT * FROM Employees WHERE EmployeeID=105"
cursor.execute(sql)
# Get a cursor description which contains column metadata
desc = cursor.description
print ("Number of columns = %d¥n" % len(desc))
# Fetch all results from the cursor into a sequence and
# display the column metadata and values as name:value pairs
rowset = cursor.fetchall()
for row in rowset:
    col = 0
    for column in cursor.description:
        name, type_code, display_size, internal_size, precision, scale, null_ok =
column

        print ("Column name: %s" % name)
        print ("Type code: %s" % type_code)
        print ("Display size: %s" % display_size)
        print ("Internal size: %d" % internal_size)
        print ("Precision: %d" % precision)
```

```
        print ("Scale: %d" % scale)
        print ("Null OK: %d" % null_ok)
        print ("Value: %s" % row[col])
        if type_code == sqlanydb.BINARY:
            print ("Python type: BINARY")
        if type_code == sqlanydb.DATE:
            print ("Python type: DATE")
        if type_code == sqlanydb.DATETIME:
            print ("Python type: DATETIME")
        if type_code == sqlanydb.NUMBER:
            print ("Python type: NUMBER")
        if type_code == sqlanydb.STRING:
            print ("Python type: STRING")
        if type_code == sqlanydb.TIME:
            print ("Python type: TIME")
        if type_code == sqlanydb.TIMESTAMP:
            print ("Python type: TIMESTAMP")
        print ("")
        col = col + 1
cursor.close()
con.close()
```

1.16 PHP サポート

PHP を使用してデータベースアプリケーションを開発できます。

このセクションの内容:

SQL Anywhere PHP 拡張 [451 ページ]
PHP (Hypertext Preprocessor) は、オープンソーススクリプト言語です。PHP は汎用スクリプト言語として使用できますが、HTML 文書に組み込むことができるスクリプトを作成するときに役立つ言語として設計されました。

SQL Anywhere PHP API リファレンス [463 ページ]
PHP API のメソッドは、接続、クエリ、結果セット、トランザクション、文、およびその他などの複数のカテゴリにソートできます。

1.16.1 SQL Anywhere PHP 拡張

PHP (Hypertext Preprocessor) は、オープンソーススクリプト言語です。PHP は汎用スクリプト言語として使用できますが、HTML 文書に組み込むことができるスクリプトを作成するときに役立つ言語として設計されました。

クライアントによって頻繁に実行される JavaScript で作成されたスクリプトとは異なり、PHP スクリプトは Web サーバによって処理され、クライアントには処理結果の HTML 出力が送信されます。PHP の構文は、Java や Perl などのその他の一般的な言語の構文から派生されたものです。

動的な Web ページを開発するときに役立つ言語になるように、PHP にはデータベースから情報を取得する機能があります。データベースソフトウェアには、PHP から SQL Anywhere データベースにアクセスするための拡張が用意されています。PHP 拡張と PHP 言語を使用することによって、スタンドアロンのスクリプトを作成し、データベースの情報に依存する動的な Web ページを作成できます。

PHP 拡張を使用すると、PHP からデータベースにネイティブな方法でアクセスできます。この PHP 拡張は単純であり、他の PHP データアクセス方法では発生する可能性のあるシステムリソースのリークを防ぐことができるため、優先して使用するようにしてください。

Windows および Linux 用のビルド済みバージョンの PHP 拡張は、http://scn.sap.com/docs/DOC-40537 からダウンロードします。PHP 拡張のソースコードは、ソフトウェアインストール環境の `sdk¥php` サブディレクトリにインストールされています。

このセクションの内容:

PHP 拡張のテスト [452 ページ]
簡単なチェックを行って、PHP 拡張が正常に動作することを確認します。

PHP テストページの作成と実行 [453 ページ]
PHP が適切に設定されているかどうかをテストするいくつかの Web ページを作成し、実行します。

PHP スクリプトの開発 [455 ページ]
PHP 拡張を使用してデータベースにアクセスする PHP スクリプトを作成できます。

UNIX と Mac OS X での PHP 拡張のビルド [461 ページ]
PHP 拡張をビルドするには、ここで説明する手順に従います。

関連情報

PHP クライアントの配備 [829 ページ]
SQL Anywhere PHP モジュール

1.16.1.1 PHP 拡張のテスト

簡単なチェックを行って、PHP 拡張が正常に動作することを確認します。

前提条件

必要なすべての PHP コンポーネントがシステムにインストールされている必要があります。

手順

1. ソフトウェアインストール環境の `bin32` サブディレクトリがパスに含まれていることを確認してください。PHP 拡張では、`bin32` ディレクトリがパスに含まれている必要があります。

2. コマンドプロンプトで、次のコマンドを実行してサンプルデータベースを起動します。

```
dbsrv17 "%SQLANYSAMP17%¥demo.db"
```

このコマンドは、サンプルデータベースを使用してデータベースサーバを起動します。

3. コマンドプロンプトで、ソフトウェアインストール環境の SDK¥PHP¥Examples サブディレクトリに移動します。php 実行プログラムディレクトリがパスに含まれていることを確認します。次のコマンドを入力します。

```
php test.php
```

次のようなメッセージが表示されます。PHP コマンドが認識されない場合は、PHP がパスにあるかを確認します。

```
Installation successful
Using php-5.2.11_sqlanywhere.dll
Connected successfully
```

PHP 拡張がロードされない場合は、コマンド "php -i" を使用して PHP セットアップに関する情報を取得できます。このコマンドの出力で extension_dir と sqlanywhere を検索してください。

4. ここまで終了したら、データベースサーバメッセージウィンドウでシャットダウンをクリックして、データベースサーバを停止します。

結果

テストが成功し、PHP 拡張が正常に動作していることを示します。

関連情報

PHP テストページの作成と実行 [453 ページ]

1.16.1.2 PHP テストページの作成と実行

PHP が適切に設定されているかどうかをテストするいくつかの Web ページを作成し、実行します。

前提条件

PHP をインストールします。

コンテキスト

このプロシージャはすべての設定に適用されます。

手順

1. ルートの Web コンテンツディレクトリに `info.php` という名前のファイルを作成します。

   使用するディレクトリがわからない場合は、Web サーバの設定ファイルを確認してください。Apache のインストール環境では、多くの場合、そのコンテンツディレクトリの名前は `htdocs` です。

   Mac OS X では、Web コンテンツディレクトリの名前は使用しているアカウントによって異なります。

   - Mac OS X システムのシステム管理者である場合は、`/Library/WebServer/Documents` を使用します。
   - Mac OS X ユーザである場合は、`/Users/your-user-name/Sites/` にファイルを置きます。
2. ファイルに次のコードを挿入します。

   ```
   <?php phpinfo(); ?>
   ```

   phpinfo は、システム設定情報のページを生成する PHP 関数です。これによって、PHP と Web サーバのインストールがともに適切に機能していることが確認されます。
3. ファイル `connect.php` を `sdk¥php¥examples` ディレクトリからルートの Web コンテンツディレクトリにコピーします。これによって、PHP、PHP 拡張、データベースサーバのインストールがともに適切に機能していることが確認されます。
4. `sa_test.php` という名前のルートの Web コンテンツディレクトリにファイルを作成し、このファイルに次のコードを挿入します。

   ```
   <?php
     $conn = sasql_connect( "UID=DBA;PWD=sql" );
     $result = sasql_query( $conn, "SELECT * FROM Employees" );
     sasql_result_all( $result );
     sasql_free_result( $result );
     sasql_disconnect( $conn );
   ?>
   ```

   sa_test ページには、Employees テーブルの内容が表示されます。
5. 必要な場合は、Web サーバを起動します。

   たとえば、Apache Web サーバを起動するには、Apache のインストール環境の `bin` サブディレクトリから次のコマンドを実行します。

   ```
   apachectl start
   ```
6. データベースサーバを実行するために環境が設定されていることを確認します。使用しているシェルに応じて適切なコマンドを入力して、ソフトウェアのインストールディレクトリから SQL Anywhere の設定スクリプトのコマンドを実行します。

   Bourne シェルとその派生シェル

Bourne シェルとその派生シェルでは、このコマンド名は . (単一のピリオド) です。たとえば、SQL Anywhere が */opt/sqlanywhere17* にインストールされている場合、次の文を使用して `sa_config.sh` のソースを指定します (32 ビット環境では `bin32` を使用してください)。

```
.  /opt/sqlanywhere17/bin64/sa_config.sh
```

Mac の場合は、次のコマンドを実行して `sa_config.sh` のソースを指定します。

```
. /Applications/SQLAnywhere17/System/bin64/sa_config.sh
```

C シェルとその派生シェル

C シェルとその派生シェルの場合、コマンドは `source` です。たとえば、SQL Anywhere が */opt/sqlanywhere17* にインストールされている場合、次のコマンドを使用して `sa_config.csh` のソースを指定します (32 ビット環境では `bin32` を使用してください)。

```
source  /opt/sqlanywhere17/bin64/sa_config.csh
```

7. コマンドプロンプトで、次のコマンドを実行してサンプルデータベースを起動します (まだ行っていない場合)。

```
dbsrv17 "%SQLANYSAMP17%¥demo.db"
```

8. PHP と Web サーバがデータベースサーバで正常に機能していることをテストするには、サーバと同じコンピュータで実行されているブラウザからテストページにアクセスします。

テストページ	URL
`info.php`	`http://localhost/info.php`
`connect.php`	`http://localhost/connect.php`
`sa_test.php`	`http://localhost/sa_test.php`

結果

info ページには、`phpinfo()` 呼び出しからの出力が表示されます。

connect ページには、メッセージ `Connected successfully` が表示されます。

sa_test ページには、Employees テーブルの内容が表示されます。

1.16.1.3 PHP スクリプトの開発

PHP 拡張を使用してデータベースにアクセスする PHP スクリプトを作成できます。

次の例と他の例で使用されるソースコードは、ソフトウェアインストール環境の `SDK¥PHP¥Examples` サブディレクトリにあります。

このセクションの内容:

PHP を使用してデータベースに接続する方法 [456 ページ]
データベースに接続するには、標準のデータベース接続文字列を sasql_connect 関数のパラメータとしてデータベースサーバに渡します。

PHP を使用してデータベースからデータを取得する方法 [456 ページ]
Web ページでの PHP スクリプトの用途の 1 つとして、データベースに含まれる情報の取り出しと表示があります。

Web フォーム [460 ページ]
PHP では、Web フォームからユーザ入力を受け取って SQL クエリとしてデータベースサーバに渡し、返される結果を表示できます。

PHP アプリケーション内の BLOB [460 ページ]
バイナリラージオブジェクト (BLOB) には、あらゆる種類のデータを格納できます。Web ブラウザで読み取れるデータであれば、PHP スクリプトによって簡単にデータベースからそのデータを取り出して動的に生成したページに表示できます。

1.16.1.3.1 PHP を使用してデータベースに接続する方法

データベースに接続するには、標準のデータベース接続文字列を sasql_connect 関数のパラメータとしてデータベースサーバに渡します。

<?php と ?> の各タグは、このタグ間のコードを PHP が実行し、そのコードを PHP 出力に置き換えることを Web サーバに指示します。

この例のソースコードは、ソフトウェアインストール環境の `connect.php` という名前のファイルにあります。

```
<?php
  # Connect to the sample database
  $conn = sasql_connect( "UID=DBA;PWD=sql" );
  if( ! $conn ) {
      echo "Connection failed¥n";
  } else {
      echo "Connected successfully¥n";
      sasql_close( $conn );
  }?>
```

このスクリプトは、ローカルサーバのデータベースに接続しようとします。このコードを正常に実行するには、サンプルデータベース、または同じクレデンシャルを持つデータベースがローカルサーバで起動されている必要があります。

1.16.1.3.2 PHP を使用してデータベースからデータを取得する方法

Web ページでの PHP スクリプトの用途の 1 つとして、データベースに含まれる情報の取り出しと表示があります。

次に示す例で、役に立つテクニックをいくつか紹介します。

単純な **select** クエリ

次の PHP コードでは、Web ページに SELECT 文の結果セットを含める便利な方法を示します。この例は、サンプルデータベースに接続し、顧客のリストを返すように設計されています。

PHP スクリプトを実行するように Web サーバを設定している場合、このコードを Web ページに埋め込むことができます。

このサンプルのソースコードは、ソフトウェアインストール環境の `query.php` という名前のファイルにあります。

```
<?php
  # Connect to the sample database using the default user ID and password DBA/sql
  $conn = sasql_connect( "UID=DBA;PWD=sql" );
  if( ! $conn ) {
      echo "sasql_connect failed¥n";
  } else {
      echo "Connected successfully¥n";
      # Execute a SELECT statement
      $result = sasql_query( $conn, "SELECT * FROM Customers" );
      if( ! $result ) {
          echo "sasql_query failed!";
      } else {
          echo "query completed successfully¥n";
          # Generate HTML from the result set
          sasql_result_all( $result );
          sasql_free_result( $result );
      }
      sasql_close( $conn );
  }
?>
```

sasql_result_all 関数が、結果セットのすべてのローをフェッチし、それを表示するための HTML 出力テーブルを生成します。sasql_free_result 関数が、結果セットを格納するために使用されたリソースを解放します。

カラム名によるフェッチ

状況によって、結果セットからすべてのデータを表示する必要がない場合、または別の方法でデータを表示したい場合があります。次の例は、結果セットの出力フォーマットを自由に制御する方法を示します。PHP によって、必要とする情報を選択した希望の方法で表示できます。

このサンプルのソースコードは、ソフトウェアインストール環境の `fetch.php` という名前のファイルにあります。

```
<?php
  # Connect to the sample database using the user ID and password DBA/sql
  $conn = sasql_connect( "UID=DBA;PWD=sql" );
  if( ! $conn ) {
      die ("Connection failed");
  } else {
      # Connected successfully.
  }
  # Execute a SELECT statement
  $result = sasql_query( $conn, "SELECT * FROM Customers" );
  if( ! $result ) {
      echo "sasql_query failed!";
      return 0;
  } else {
      echo "query completed successfully¥n";
  }
  # Retrieve meta information about the results
```

```
  $num_cols = sasql_num_fields( $result );
  $num_rows = sasql_num_rows( $result );
  echo "Num of rows = $num_rows¥n";
  echo "Num of cols = $num_cols¥n";
  while( ($field = sasql_fetch_field( $result )) ) {
      echo "Field # : $field->id ¥n";
      echo "¥tname   : $field->name ¥n";
      echo "¥tlength : $field->length ¥n";
      echo "¥ttype   : $field->type ¥n";
  }
  # Fetch all the rows
  $curr_row = 0;
  while( ($row = sasql_fetch_row( $result )) ) {
      $curr_row++;
      $curr_col = 0;
      while( $curr_col < $num_cols ) {
          echo "$row[$curr_col]¥t|";
          $curr_col++;
      }
      echo "¥n";
  }
  # Clean up.
  sasql_free_result( $result );
  sasql_disconnect( $conn );
?>
```

sasql_fetch_array 関数が、テーブルの単一ローを返します。データは、カラム名とカラムインデックスを基準に取り出すことができます。

sasql_fetch_assoc が、テーブルの単一ローを連想配列として返します。カラム名をインデックスとして使用して、データを取得できます。次はその例です。

このサンプルのソースコードは、ソフトウェアインストール環境の `fetch_assoc.php` という名前のファイルにあります。

```
<?php
  # Connect to the sample database using the user ID and password DBA/sql
  $conn = sasql_connect("UID=DBA;PWD=sql");

  /* check connection */
  if( sasql_errorcode() ) {
      printf("Connect failed: %s¥n", sasql_error());
      exit();
  }

  $query = "SELECT Surname, Phone FROM Employees ORDER by EmployeeID";

  if( $result = sasql_query($conn, $query) ) {

      /* fetch associative array */
      while( $row = sasql_fetch_assoc($result) ) {
          printf ("%s (%s)¥n", $row["Surname"], $row["Phone"]);
      }

      /* free result set */
      sasql_free_result($result);
  }

  /* close connection */
  sasql_close($conn);
?>
```

この他に PHP インタフェースには 2 つの類似したメソッドがあります。sasql_fetch_row はカラムインデックスのみで検索し、sasql_fetch_object はカラム名のみで検索してローを返します。

sasql_fetch_object 関数の例については、`fetch_object.php` のサンプルスクリプトを参照してください。

ネストされた結果セット

SELECT 文がデータベースに送信されると、結果セットが返されます。sasql_fetch_row 関数と sasql_fetch_array 関数を使用すると、結果セットの個々のローからデータが取り出され、さらに問い合わせることができるカラムの配列としてそれぞれのローが返されます。

このサンプルのソースコードは、ソフトウェアインストール環境の `nested.php` という名前のファイルにあります。

```
<?php
  $conn = sasql_connect( "UID=DBA;PWD=sql" );
  if( $conn ) {
      // get the GROUPO user id
      $result = sasql_query( $conn,
            "SELECT user_id FROM SYS.SYSUSER " .
            "WHERE user_name='GROUPO'" );
      if( $result ) {
          $row = sasql_fetch_array( $result );
          $user = $row[0];
      } else {
          $user = 0;
      }
      // get the tables created by user GROUPO
      $result = sasql_query( $conn,
            "SELECT table_id, table_name FROM SYS.SYSTABLE " .
            "WHERE creator = $user" );
      if( $result ) {
          $num_rows = sasql_num_rows( $result );
          echo "Returned rows : $num_rows¥n";
          while( $row = sasql_fetch_array( $result ) ) {
              echo "Table: $row[1]¥n";
              $query = "SELECT table_id, column_name FROM SYS.SYSCOLUMN " .
                        "WHERE table_id = '$row[table_id]'";
              $result2 = sasql_query( $conn, $query );
              if( $result2 ) {
                  echo "Columns:";
                  while( $detailed = sasql_fetch_array( $result2 ) ) {
                      echo " $detailed[column_name]";
                  }
                  sasql_free_result( $result2 );
              }
              echo "¥n¥n";
          }
          sasql_free_result( $result );
      }
      sasql_disconnect( $conn );
  }
?>
```

上の例では、SQL 文で SYSTAB から各テーブルのテーブル ID とテーブル名を選択します。sasql_query 関数が、ローの配列を返します。スクリプトは、sasql_fetch_array 関数を使用してそれらのローを反復し、ローを配列から取り出します。内部反復がその各ローのカラムで行われ、それらの値が出力されます。

1.16.1.3.3 Web フォーム

PHP では、Web フォームからユーザ入力を受け取って SQL クエリとしてデータベースサーバに渡し、返される結果を表示できます。

次の例は、ユーザが SQL 文を使用してサンプルデータベースを問い合わせ、結果を HTML テーブルに表示する簡単な Web フォームを示しています。

このサンプルのソースコードは、ソフトウェアインストール環境の `webisql.php` という名前のファイルにあります。

```
<?php
    echo "<HTML>¥n";
    echo "<body onload=¥"document.getElementById('qbox').focus()¥">¥n";
    $qname = $_POST[qname];
    $qname = str_replace( "¥¥", "", $qname );
    echo "<form method=post action=webisql.php>¥n";
    echo "<br>Query : <input id=qbox type=text size=80 name=qname value=¥"$qname¥">
¥n";
    echo "<input type=submit>¥n";
    echo "</form>¥n";
    echo "<HR><br>¥n";
    if( ! $qname ) {
        echo "No Current Query¥n";
        return;
    }
    $conn = sasql_connect( "UID=DBA;PWD=sql" );
    if( ! $conn ) {
        echo "sasql_connect failed¥n";
    } else {
        $qname = str_replace( "¥¥", "", $qname );
        $result = sasql_query( $conn, $qname );
        if( ! $result ) {
            echo "sasql_query failed!";
        } else {
            if( sasql_field_count( $conn ) > 0 ) {
                sasql_result_all( $result, "border=1" );
                sasql_free_result( $result );
            } else {
                echo "The statement <h3>$qname></h3> executed successfully!";
            }
        }
        sasql_close( $conn );
    }
    echo "</body>¥n";
    echo "</html>¥n";
?>
```

この設計は、ユーザが入力する値に基づいてカスタマイズされる SQL クエリを作成することによって、複雑な Web フォームを処理できるように拡張できます。

1.16.1.3.4 PHP アプリケーション内の BLOB

バイナリラージオブジェクト (BLOB) には、あらゆる種類のデータを格納できます。Web ブラウザで読み取れるデータであれば、PHP スクリプトによって簡単にデータベースからそのデータを取り出して動的に生成したページに表示できます。

BLOB フィールドは、多くの場合、GIF や JPG 形式のイメージなどのテキスト以外のデータを格納するために使用します。サードパーティソフトウェアやデータ型変換を必要とせずに、さまざまな種類のデータを Web ブラウザに渡すことができます。次の例は、イメージをデータベースに追加し、それを再び取り出して Web ブラウザに表示する処理を示します。

このサンプルは、ソフトウェアインストール環境の image-insert.php ファイルと image-retrieve.php ファイルにあるサンプルコードに似ています。これらのサンプルでは、イメージを格納する BLOB カラムの使用についても示しています。

```
<?php
  $conn = sasql_connect( "UID=DBA;PWD=sql" )
        or die("Cannot connect to database");
  $create_table = "CREATE TABLE images (ID INTEGER PRIMARY KEY, img IMAGE)";
  sasql_query( $conn, $create_table);
  $insert = "INSERT INTO images VALUES (99, xp_read_file('SAP_logo.gif'))";
  sasql_query( $conn, $insert );
  $query = "SELECT img FROM images WHERE ID = 99";
  $result = sasql_query($conn, $query);
  $data = sasql_fetch_row($result);
  $img = $data[0];
  header("Content-type: image/gif");
  echo $img;
  sasql_disconnect($conn);
?>
```

バイナリデータをデータベースから直接 Web ブラウザに送信するには、スクリプトでヘッダ関数を使用してデータの MIME タイプを設定する必要があります。この例の場合、ブラウザは GIF イメージを受け取るように指定されているため、イメージが正しく表示されます。

1.16.1.4 UNIX と Mac OS X での PHP 拡張のビルド

PHP 拡張をビルドするには、ここで説明する手順に従います。

前提条件

UNIX と Mac OS X で PHP 拡張を構築する場合、次のソフトウェアをシステムにインストールしておく必要があります。

- SQL Anywhere PHP 拡張ソースコード。このコードは、sdk/php の下の SQL Anywhere ソフトウェアインストールに含まれています。
- PHP Web サイトからダウンロードおよびインストールできる PHP 開発パッケージ (php5-dev)。

コンテキスト

インストール作業の特定の手順を実行するためには、PHP をインストールした人と同じアクセス権が必要になります。UNIX ベースのシステムには通常 *sudo* コマンドがあり、十分なパーミッションを持たないユーザでも、権限を持ったユーザと同様に特定のコマンドを実行できます。

手順

1. 提供されているスクリプトのいずれかを使用して SQL Anywhere の環境変数のソースを指定します。

   Bourne シェルとその派生シェル

   Bourne シェルとその派生シェルでは、このコマンド名は . (単一のピリオド) です。たとえば、SQL Anywhere が */opt/sqlanywhere17* にインストールされている場合、次の文を使用して `sa_config.sh` のソースを指定します (32 ビット環境では `bin32` を使用してください)。

   ```
   .  /opt/sqlanywhere17/bin64/sa_config.sh
   ```

   Mac の場合は、次のコマンドを実行して `sa_config.sh` のソースを指定します。

   ```
   . /Applications/SQLAnywhere17/System/bin64/sa_config.sh
   ```

   C シェルとその派生シェル

   C シェルとその派生シェルの場合、コマンドは `source` です。たとえば、SQL Anywhere が */opt/sqlanywhere17* にインストールされている場合、次のコマンドを使用して `sa_config.csh` のソースを指定します (32 ビット環境では `bin32` を使用してください)。

   ```
   source  /opt/sqlanywhere17/bin64/sa_config.csh
   ```

2. SQL Anywhere PHP 拡張ソフトウェアを格納している SQL Anywhere`sdk/php` ディレクトリに変更します。
3. 次のコマンドを使用して拡張をビルドします。

   ```
   $ phpize
   $ ./configure --with-sqlanywhere
   $ make
   $ make test
   $ sudo make install
   ```

4. PHP ソフトウェアインストールディレクトリの `php.ini` ファイルを編集し、`enable_dl` オプションが次のように有効になっていることを確認します。

   ```
   enable_dl = On
   ```

5. PHP 拡張ディレクトリの内容を表示して、`sqlanywhere.so` を探し、PHP が拡張を認識していることを確認します。

   ```
   $ php -i | grep extension_dir
   $ ls <extension_dir>
   ```

結果

PHP 拡張のビルドに失敗した場合は、このコマンドの出力を記録し、SQL Anywhere フォーラムに投稿してサポートを受けてください。

関連情報

SAP SQL Anywhere PHP モジュール
SQL Anywhere クライアントインタフェース
SQL Anywhere PHP モジュール
Apache HTTP Server プロジェクト
SQL Anywhere フォーラム

1.16.2 SQL Anywhere PHP API リファレンス

PHP API のメソッドは、接続、クエリ、結果セット、トランザクション、文、およびその他などの複数のカテゴリにソートできます。

PHP API では、次の関数をサポートしています。

接続

- sasql_close
- sasql_connect
- sasql_disconnect
- sasql_error
- sasql_errorcode
- sasql_insert_id
- sasql_message
- sasql_pconnect
- sasql_set_option

クエリ

- sasql_affected_rows
- sasql_next_result
- sasql_query
- sasql_real_query
- sasql_store_result
- sasql_use_result

結果セット

- sasql_data_seek
- sasql_fetch_array
- sasql_fetch_assoc
- sasql_fetch_field
- sasql_fetch_object
- sasql_fetch_row
- sasql_field_count
- sasql_free_result
- sasql_num_rows
- sasql_result_all

トランザクション

- sasql_commit
- sasql_rollback

文

- sasql_prepare
- sasql_stmt_affected_rows
- sasql_stmt_bind_param
- sasql_stmt_bind_param_ex
- sasql_stmt_bind_result
- sasql_stmt_close
- sasql_stmt_data_seek
- sasql_stmt_errno
- sasql_stmt_error
- sasql_stmt_execute
- sasql_stmt_fetch
- sasql_stmt_field_count
- sasql_stmt_free_result
- sasql_stmt_insert_id
- sasql_stmt_next_result
- sasql_stmt_num_rows
- sasql_stmt_param_count
- sasql_stmt_reset
- sasql_stmt_result_metadata
- sasql_stmt_send_long_data
- sasql_stmt_store_result

その他

- sasql_escape_string
- sasql_get_client_info

このセクションの内容:

sasql_affected_rows [468 ページ]
最後の SQL 文の影響を受けるローの数を返します。

sasql_commit [469 ページ]
データベースのトランザクションを終了し、トランザクション中に加えられたすべての変更を永続的なものにします。

sasql_close [469 ページ]
開いていたデータベース接続を閉じます。

sasql_connect [470 ページ]
データベースへの接続を確立します。

sasql_data_seek [471 ページ]
ローにカーソルを配置します。

sasql_disconnect [472 ページ]
sasql_connect または sasql_pconnect によって開かれている接続を閉じます。

sasql_error [472 ページ]
最後に実行された関数のエラーテキストを返します。

sasql_errorcode [473 ページ]
最後に実行された関数のエラーコードを返します。

sasql_escape_string [474 ページ]
指定された文字列内の特殊文字をすべてエスケープします。

sasql_fetch_array [475 ページ]
結果セットから 1 つのローをフェッチします。

sasql_fetch_assoc [476 ページ]
連想配列として結果セットからローを 1 つフェッチします。

sasql_fetch_field [477 ページ]
特定のカラムに関する情報を含むオブジェクトを返します。

sasql_fetch_object [478 ページ]
オブジェクトとして結果セットからローを 1 つフェッチします。

sasql_fetch_row [479 ページ]
結果セットから 1 つのローをフェッチします。このローは、カラムインデックスのみによってインデックス付けが可能な配列として返されます。

sasql_field_count [480 ページ]
最後の結果に含まれているカラム (フィールド) の数を返します。

sasql_field_seek [480 ページ]
フィールドカーソルを特定のオフセットに設定します。

sasql_free_result [481 ページ]

sasql_query、sasql_store_result、または sasql_use_result から返される結果リソースに関連付けられているデータベースリソースを解放します。

sasql_get_client_info [482 ページ]
クライアントのバージョン情報を返します。

sasql_insert_id [482 ページ]
IDENTITY カラムまたは DEFAULT AUTOINCREMENT カラムに最後に挿入された値を返します。最後に挿入したテーブルに IDENTITY や DEFAULT AUTOINCREMENT カラムが含まれていないと、0 を返します。

sasql_message [483 ページ]
メッセージをサーバメッセージウィンドウに書き込みます。

sasql_multi_query [483 ページ]
1 つまたは複数の SQL クエリを準備して実行します。

sasql_next_result [484 ページ]
実行された最後のクエリの次の結果セットを準備します。

sasql_num_fields [485 ページ]
ロー内のフィールドの数を返します。

sasql_num_rows [486 ページ]
結果内のローの数を返します。

sasql_pconnect [487 ページ]
データベースへの永続的接続を確立します。

sasql_prepare [488 ページ]
指定された SQL 文字列を準備します。

sasql_query [489 ページ]
SQL クエリを準備して実行します。

sasql_real_escape_string [490 ページ]
指定された文字列内の特殊文字をすべてエスケープします。

sasql_real_query [491 ページ]
指定された接続リソースを使用して、データベースに対してクエリを実行します。

sasql_result_all [492 ページ]
すべての結果をフェッチし、オプションのフォーマット文字列に従って HTML 出力テーブルを生成します。

sasql_rollback [493 ページ]
データベースのトランザクションを終了し、トランザクション中に加えられたすべての変更を破棄します。

sasql_set_option [494 ページ]
指定した接続で、指定したオプションの値を設定します。

sasql_stmt_affected_rows [495 ページ]
文の実行の影響を受けるローの数を返します。

sasql_stmt_bind_param [496 ページ]
PHP 変数を文のパラメータにバインドします。

sasql_stmt_bind_param_ex [497 ページ]
PHP 変数を文のパラメータにバインドします。

sasql_stmt_bind_result [498 ページ]

実行されたステートメントの結果カラムに 1 つ以上の PHP 変数をバインドして、結果セットを返します。

sasql_stmt_close [498 ページ]
指定されたステートメントリソースを閉じて、関連付けられているリソースを解放します。

sasql_stmt_data_seek [499 ページ]
結果セット内で指定されたオフセットを検索します。

sasql_stmt_errno [500 ページ]
指定されたステートメントリソースを使用して最後に実行された文関数のエラーコードを返します。

sasql_stmt_error [501 ページ]
指定されたステートメントリソースを使用して最後に実行された文関数のエラーテキストを返します。

sasql_stmt_execute [501 ページ]
準備ステートメントを実行します。sasql_stmt_result_metadata を使用して、ステートメントが結果セットを返すかどうかを確認できます。

sasql_stmt_fetch [502 ページ]
文の結果からローを 1 つフェッチし、sasql_stmt_bind_result を使用してバインドされた変数にカラムを配置します。

sasql_stmt_field_count [503 ページ]
文の結果セット内のカラム数を返します。

sasql_stmt_free_result [503 ページ]
キャッシュされた文の結果セットを解放します。

sasql_stmt_insert_id [504 ページ]
IDENTITY カラムまたは DEFAULT AUTOINCREMENT カラムに最後に挿入された値を返します。最後に挿入したテーブルに IDENTITY や DEFAULT AUTOINCREMENT カラムが含まれていないと、0 を返します。

sasql_stmt_next_result [505 ページ]
文の次の結果に進みます。

sasql_stmt_num_rows [506 ページ]
結果セット内のロー数を返します。

sasql_stmt_param_count [506 ページ]
指定された準備文リソース内のパラメータ数を返します。

sasql_stmt_reset [507 ページ]
文オブジェクトを記述直後の状態にリセットします。

sasql_stmt_result_metadata [508 ページ]
指定された文の結果セットオブジェクトを返します。

sasql_stmt_send_long_data [508 ページ]
ユーザがパラメータデータをチャンク単位で送信できるようにします。

sasql_stmt_store_result [509 ページ]
クライアントで文の結果セット全体をキャッシュできるようになります。

sasql_store_result [510 ページ]
最後のクエリの結果セットを転送します。

sasql_sqlstate [512 ページ]
最新の SQLSTATE 文字列を返します。

sasql_use_result [512 ページ]

接続で最後に実行されたクエリの結果セットの取得を開始します。

1.16.2.1 sasql_affected_rows

最後の SQL 文の影響を受けるローの数を返します。

構文

```
int sasql_affected_rows( sasql_conn $conn )
```

パラメータ

$conn

接続関数から返される接続リソース。

戻り値

影響を受けたローの数。

備考

通常、この関数は INSERT 文、UPDATE 文、または DELETE 文に使用します。SELECT 文には sasql_num_rows 関数を使用します。

関連情報

sasql_num_rows [486 ページ]

1.16.2.2 sasql_commit

データベースのトランザクションを終了し、トランザクション中に加えられたすべての変更を永続的なものにします。

構文

```
bool sasql_commit( sasql_conn $conn )
```

パラメータ

$conn

接続関数から返される接続リソース。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

備考

auto_commit オプションが Off である場合にのみ有効です。

関連情報

sasql_rollback [493 ページ]
sasql_set_option [494 ページ]
sasql_pconnect [487 ページ]
sasql_disconnect [472 ページ]

1.16.2.3 sasql_close

開いていたデータベース接続を閉じます。

構文

```
bool sasql_close( sasql_conn $conn )
```

パラメータ

$conn

接続関数から返される接続リソース。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

1.16.2.4 sasql_connect

データベースへの接続を確立します。

構文

```
sasql_conn sasql_connect( string $con_str )
```

パラメータ

$con_str

有効な接続文字列。

戻り値

成功の場合は正の値の接続リソース、失敗の場合はエラーまたは 0。

関連情報

sasql_pconnect [487 ページ]
sasql_disconnect [472 ページ]

1.16.2.5 sasql_data_seek

ローにカーソルを配置します。

構文

```
bool sasql_data_seek( sasql_result $result, int row_num )
```

パラメータ

$result

sasql_query 関数によって返される結果リソース。

row_num

結果リソース内のカーソルの新しい位置を表す整数。たとえば、0 に指定するとカーソルは結果セットの最初のローに移動し、5 に指定すると 6 番目のローに移動します。負の数は結果セットの最後の位置に相対的なローを表します。たとえば、-1 に指定するとカーソルは結果セットの最後のローに移動し、-2 に指定すると最後から 2 番目のローに移動します。

戻り値

成功の場合は TRUE、エラーの場合は FALSE。

備考

sasql_query を使用して開かれた `$result` のロー `row_num` にカーソルを配置します。

関連情報

sasql_fetch_field [477 ページ]
sasql_fetch_array [475 ページ]
sasql_fetch_assoc [476 ページ]
sasql_fetch_row [479 ページ]
sasql_fetch_object [478 ページ]
sasql_query [489 ページ]

1.16.2.6 sasql_disconnect

sasql_connect または sasql_pconnect によって開かれている接続を閉じます。

構文

```
bool sasql_disconnect( sasql_conn $conn )
```

パラメータ

$conn

接続関数から返される接続リソース。

戻り値

成功の場合は TRUE、エラーの場合は FALSE。

関連情報

sasql_connect [470 ページ]
sasql_pconnect [487 ページ]

1.16.2.7 sasql_error

最後に実行された関数のエラーテキストを返します。

構文

```
string sasql_error( [ sasql_conn $conn ] )
```

パラメータ

$conn

接続関数から返される接続リソース。

戻り値

エラーが記述された文字列。

備考

エラーメッセージは接続ごとに格納されます。$conn を指定しないと、sasql_error は接続が使用できなかったときの最後のエラーメッセージを返します。たとえば、sasql_connect を呼び出して接続が失敗した場合、$conn のパラメータを設定せずに sasql_error を呼び出すと、エラーメッセージを取得します。対応するエラーコード値を取得する場合は、sasql_errorcode 関数を使用します。

関連情報

sasql_errorcode [473 ページ]
sasql_sqlstate [512 ページ]
sasql_set_option [494 ページ]
sasql_stmt_errno [500 ページ]
sasql_stmt_error [501 ページ]

1.16.2.8 sasql_errorcode

最後に実行された関数のエラーコードを返します。

構文

```
int sasql_errorcode( [ sasql_conn $conn ] )
```

パラメータ

$conn

接続関数から返される接続リソース。

戻り値

エラーコードを表す符号付き整数。エラーコードが 0 の場合は、処理が正常に終了したことを示します。エラーコードが正数の場合は、警告を伴う正常終了を示します。エラーコードが負数の場合は、処理が失敗したことを示します。

備考

エラーコードは接続ごとに格納されます。$conn を指定しないと、sasql_errorcode は接続が使用できなかったときの最後のエラーコードを返します。たとえば、sasql_connect を呼び出して接続が失敗した場合、$conn のパラメータを設定せずに sasql_errorcode を呼び出すと、エラーコードを取得します。対応するエラーメッセージを取得する場合は、the sasql_error 関数を使用します。

関連情報

sasql_connect [470 ページ]
sasql_pconnect [487 ページ]
sasql_error [472 ページ]
sasql_sqlstate [512 ページ]
sasql_set_option [494 ページ]
sasql_stmt_errno [500 ページ]
sasql_stmt_error [501 ページ]

1.16.2.9 sasql_escape_string

指定された文字列内の特殊文字をすべてエスケープします。

構文

```
string sasql_escape_string( sasql_conn $conn, string $str )
```

パラメータ

$conn

接続関数から返される接続リソース。

$string

エスケープする文字列。

戻り値

エスケープされた文字列。

備考

エスケープされる特殊文字は、¥r、¥n、'、"、;、¥、および NULL 文字です。この関数は、sasql_real_escape_string のエイリアスです。

関連情報

sasql_real_escape_string [490 ページ]
sasql_connect [470 ページ]

1.16.2.10 sasql_fetch_array

結果セットから 1 つのローをフェッチします。

構文

```
array sasql_fetch_array( sasql_result $result [, int $result_type ])
```

パラメータ

$result

sasql_query 関数によって返される結果リソース。

$result_type

このオプションのパラメータは、現在のローデータから生成される配列の種類を示す定数です。このパラメータに指定できる値は、定数 SASQL_ASSOC、SASQL_NUM、または SASQL_BOTH です。デフォルトで SASQL_BOTH になります。

SASQL_ASSOC 定数を使用すると、この関数は sasql_fetch_assoc 関数と同じ動作をし、SASQL_NUM 定数を使用すると sasql_fetch_row 関数と同じ動作をします。最後のオプション SASQL_BOTH を使用すると、両方の属性を持つ単一配列が作成されます。

戻り値

結果セットのローを表す配列。ローがない場合は FALSE。

備考

このローは、カラム名またはカラムインデックスによってインデックス付けが可能な配列として返されます。

関連情報

sasql_data_seek [471 ページ]
sasql_fetch_assoc [476 ページ]
sasql_fetch_field [477 ページ]
sasql_fetch_row [479 ページ]
sasql_fetch_object [478 ページ]

1.16.2.11 sasql_fetch_assoc

連想配列として結果セットからローを 1 つフェッチします。

構文

```
array sasql_fetch_assoc( sasql_result $result )
```

パラメータ

$result

sasql_query 関数によって返される結果リソース。

戻り値

結果セットからフェッチされたローを表す文字列の連想配列。配列内の各キーは結果セットの 1 つのカラムの名前を表します。結果セットにそれ以上ローがない場合は、FALSE を返します。

関連情報

sasql_data_seek [471 ページ]
sasql_fetch_field [477 ページ]
sasql_fetch_field [477 ページ]
sasql_fetch_row [479 ページ]
sasql_fetch_object [478 ページ]

1.16.2.12 sasql_fetch_field

特定のカラムに関する情報を含むオブジェクトを返します。

構文

```
object sasql_fetch_field( sasql_result $result [, int $field_offset ] )
```

パラメータ

$result

sasql_query 関数によって返される結果リソース。
$field_offset

取り出したい情報のカラム (フィールド) を表す整数。カラムは 0 から始まります。最初のカラムを取得するには、値 0 を指定します。このパラメータを省略すると、次のフィールドオブジェクトが返されます。

戻り値

次のプロパティを持つオブジェクトが返されます。

id

フィールド番号を示します。
name

フィールド名を示します。
numeric

フィールドが数値であるかどうかを示します。
length

フィールドのネイティブストレージサイズ、つまり native_type equal を DT_DECIMAL に設定したフィールドの表示サイズを返します。

type

フィールドのタイプを返します。

native_type

フィールドのネイティブタイプを返します。DT_FIXCHAR、DT_DECIMAL、DT_DATE などの値です。

precision

フィールドの数値精度を返します。このプロパティが設定されるのは、native_type が DT_DECIMAL のフィールドのみです。

scale

フィールドの数値の位取りを返します。このプロパティが設定されるのは、native_type が DT_DECIMAL のフィールドのみです。

関連情報

Embedded SQL のデータ型 [268 ページ]
sasql_data_seek [471 ページ]
sasql_fetch_array [475 ページ]
sasql_fetch_assoc [476 ページ]
sasql_fetch_row [479 ページ]
sasql_fetch_object [478 ページ]

1.16.2.13 sasql_fetch_object

オブジェクトとして結果セットからローを 1 つフェッチします。

構文

```
object sasql_fetch_object( sasql_result $result )
```

パラメータ

$result

sasql_query 関数によって返される結果リソース。

戻り値

結果セットからフェッチされたローを表すブジェクト。各プロパティ名は結果セットの 1 つのカラム名と一致します。結果セットにそれ以上ローがない場合は、FALSE を返します。

関連情報

sasql_data_seek [471 ページ]
sasql_fetch_field [477 ページ]
sasql_fetch_array [475 ページ]
sasql_fetch_assoc [476 ページ]
sasql_fetch_row [479 ページ]

1.16.2.14 sasql_fetch_row

結果セットから 1 つのローをフェッチします。このローは、カラムインデックスのみによってインデックス付けが可能な配列として返されます。

構文

```
array sasql_fetch_row( sasql_result $result )
```

パラメータ

$result

sasql_query 関数によって返される結果リソース。

戻り値

結果セットのローを表す配列。ローがない場合は FALSE。

関連情報

sasql_data_seek [471 ページ]

sasql_fetch_field [477 ページ]
sasql_fetch_array [475 ページ]
sasql_fetch_assoc [476 ページ]
sasql_fetch_object [478 ページ]

1.16.2.15 sasql_field_count

最後の結果に含まれているカラム (フィールド) の数を返します。

構文

```
int sasql_field_count( sasql_conn $conn )
```

パラメータ

$conn

接続関数から返される接続リソース。

戻り値

カラムの正数。`$conn` が有効でない場合は FALSE。

1.16.2.16 sasql_field_seek

フィールドカーソルを特定のオフセットに設定します。

構文

```
bool sasql_field_seek( sasql_result $result, int $field_offset )
```

パラメータ

$result

sasql_query 関数によって返される結果リソース。

$field_offset

取り出したい情報のカラム (フィールド) を表す整数。カラムは 0 から始まります。最初のカラムを取得するには、値 0 を指定します。このパラメータを省略すると、次のフィールドオブジェクトが返されます。

戻り値

成功の場合は TRUE、エラーの場合は FALSE。

備考

次の sasql_fetch_field 呼び出し時に、そのオフセットに関連付けられたカラムのフィールド定義を取得します。

1.16.2.17 sasql_free_result

sasql_query、sasql_store_result、または sasql_use_result から返される結果リソースに関連付けられているデータベースリソースを解放します。

構文

```
bool sasql_free_result( sasql_result $result )
```

パラメータ

$result

sasql_query 関数によって返される結果リソース。

戻り値

成功の場合は TRUE、エラーの場合は FALSE。

関連情報

sasql_query [489 ページ]

sasql_store_result [510 ページ]
sasql_use_result [512 ページ]

1.16.2.18 sasql_get_client_info

クライアントのバージョン情報を返します。

構文

```
string sasql_get_client_info( )
```

パラメータ

なし

戻り値

クライアントソフトウェアのバージョンを表す文字列。文字列は X.Y.Z.W の形式で返されます。X はメジャーバージョン番号、Y はマイナーバージョン番号、Z はパッチ番号、W はビルド番号を表します。たとえば、10.0.1.3616 のようになります。

1.16.2.19 sasql_insert_id

IDENTITY カラムまたは DEFAULT AUTOINCREMENT カラムに最後に挿入された値を返します。最後に挿入したテーブルに IDENTITY や DEFAULT AUTOINCREMENT カラムが含まれていないと、0 を返します。

構文

```
int sasql_insert_id( sasql_conn $conn )
```

パラメータ

$conn

接続関数から返される接続リソース。

戻り値

前回の INSERT 文で生成された AUTOINCREMENT カラムの ID。最後の挿入が AUTOINCREMENT カラムに影響しなかった場合は 0。`$conn` が有効でない場合は FALSE。

備考

sasql_insert_id 関数は、MySQL データベースとの互換性のために用意されています。

1.16.2.20 sasql_message

メッセージをサーバメッセージウィンドウに書き込みます。

構文

```
bool sasql_message( sasql_conn $conn, string $message )
```

パラメータ

$conn

接続関数から返される接続リソース。

$message

サーバメッセージウィンドウに書き込まれるメッセージ。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

1.16.2.21 sasql_multi_query

1 つまたは複数の SQL クエリを準備して実行します。

構文

```
bool sasql_multi_query( sasql_conn $conn, string $sql_str )
```

パラメータ

$conn

接続関数から返される接続リソース。

$sql_str

セミコロンで区切られた 1 つ以上の SQL 文。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

備考

指定された接続リソースを使用して、`$sql_str` によって指定された 1 つまたは複数の SQL クエリを準備して実行します。各クエリはセミコロンによって区切られます。

最初のクエリ結果は、sasql_use_result または sasql_store_result を使用して取得または格納できます。sasql_field_count を使用すると、クエリから結果セットが返されるかどうかを確認できます。

それ以降のクエリ結果は、sasql_next_result および sasql_use_result/sasql_store_result を使用して処理できます。

関連情報

sasql_store_result [510 ページ]
sasql_use_result [512 ページ]
sasql_field_count [480 ページ]

1.16.2.22 sasql_next_result

実行された最後のクエリの次の結果セットを準備します。

構文

```
bool sasql_next_result( sasql_conn $conn )
```

パラメータ

$conn

接続関数から返される接続リソース。

戻り値

取り出す結果セットが他にない場合は FALSE。取り出す結果セットが別にある場合は TRUE。sasql_use_result または sasql_store_result を呼び出して、次の結果セットを取り出します。

備考

`$conn` で実行された最後のクエリの次の結果セットを準備します。

関連情報

sasql_use_result [512 ページ]
sasql_store_result [510 ページ]

1.16.2.23 sasql_num_fields

ロー内のフィールドの数を返します。

構文

```
int sasql_num_fields( sasql_result $result )
```

パラメータ

$result

sasql_query 関数によって返される結果リソース。

戻り値

指定された結果セット内のフィールド数を返します。

備考

`$result` 内のローに含まれるフィールドの数を返します。

関連情報

sasql_num_rows [486 ページ]
sasql_query [489 ページ]

1.16.2.24 sasql_num_rows

結果内のローの数を返します。

構文

```
int sasql_num_rows( sasql_result $result )
```

パラメータ

$result

sasql_query 関数によって返される結果リソース。

戻り値

ローの数が厳密である場合は正の数、概数である場合は負の数。ローの厳密な数を取得するには、データベースオプション row_counts をデータベースで永続的に設定するか、接続で一時的に設定します。

備考

`$result` に含まれるローの数を返します。

関連情報

sasql_num_fields [485 ページ]
sasql_query [489 ページ]
sasql_set_option [494 ページ]

1.16.2.25 sasql_pconnect

データベースへの永続的接続を確立します。

構文

```
sasql_conn sasql_pconnect( string $con_str )
```

パラメータ

$con_str

有効な接続文字列。

戻り値

成功の場合は正の値の永続的接続リソース、失敗の場合はエラーまたは 0。

備考

sasql_connect の代わりに sasql_pconnect を使用すると、Apache が子プロセスを生成する方法に応じて、パフォーマンスが向上することがあります。場合によって、永続的な接続では、接続プーリングと同様にパフォーマンスが向上します。データベースサーバに接続数の制限がある場合 (たとえば、パーソナルデータベースサーバで同時接続の数を 10 に制限)、永続的な接続を使用するには注意が必要です。永続的な接続はそれぞれの子プロセスにアタッチされるので、使用可能な接続数を超えた子プロセスが Apache にあると、接続エラーが発生します。

関連情報

sasql_connect [470 ページ]
sasql_disconnect [472 ページ]

1.16.2.26 sasql_prepare

指定された SQL 文字列を準備します。

構文

```
sasql_stmt sasql_prepare( sasql_conn $conn, string $sql_str )
```

パラメータ

$conn

接続関数から返される接続リソース。

$sql_str

準備される SQL 文。適切な位置に疑問符を埋め込んで、文字列にパラメータマーカを含めることができます。

戻り値

文のオブジェクト。失敗した場合は FALSE。失敗の原因を判定するには、sasql_error 関数と sasql_errorcode 関数を使用します。

関連情報

sasql_stmt_param_count [506 ページ]
sasql_stmt_bind_param [496 ページ]
sasql_stmt_bind_param_ex [497 ページ]
sasql_error [472 ページ]
sasql_errorcode [473 ページ]
sasql_stmt_execute [501 ページ]
sasql_connect [470 ページ]
sasql_pconnect [487 ページ]

1.16.2.27 sasql_query

SQL クエリを準備して実行します。

構文

```
mixed sasql_query( sasql_conn $conn, string $sql_str [, int $result_mode ] )
```

パラメータ

$conn

接続関数から返される接続リソース。

$sql_str

データベースサーバによってサポートされている SQL 文。

$result_mode

SASQL_USE_RESULT または SASQL_STORE_RESULT (デフォルト) のどちらか。

戻り値

失敗した場合は FALSE。INSERT、UPDATE、DELETE、CREATE で成功した場合は TRUE。SELECT で成功した場合は sasql_result。

備考

sasql_connect または sasql_pconnect を使用してすでに開かれている、`$conn` によって識別される接続で、SQL クエリ `$sql_str` を準備、記述、実行します。

sasql_query 関数は、2 つの関数 (sasql_real_query と、sasql_store_result または sasql_use_result のいずれか 1 つ) を呼び出すことと同等です。

準備手順および記述手順で警告が生成された場合、実行に成功すると上書きされます。各手順から警告を取得する必要がある場合、sasql_prepare、sasql_error、sasql_errorcode 関数を使用します。

関連情報

sasql_real_query [491 ページ]
sasql_free_result [481 ページ]

sasql_fetch_array [475 ページ]
sasql_fetch_field [477 ページ]
sasql_fetch_object [478 ページ]
sasql_fetch_row [479 ページ]

1.16.2.28 sasql_real_escape_string

指定された文字列内の特殊文字をすべてエスケープします。

構文

```
string sasql_real_escape_string( sasql_conn $conn, string $str )
```

パラメータ

$conn

接続関数から返される接続リソース。

$string

エスケープする文字列。

戻り値

エスケープされた文字列。エラーの場合は FALSE。

備考

エスケープされる特殊文字は、¥r、¥n、'、"、;、¥、および NULL 文字です。

関連情報

sasql_escape_string [474 ページ]
sasql_connect [470 ページ]

1.16.2.29 sasql_real_query

指定された接続リソースを使用して、データベースに対してクエリを実行します。

構文

```
bool sasql_real_query( sasql_conn $conn, string $sql_str )
```

パラメータ

$conn

接続関数から返される接続リソース。

$sql_str

データベースサーバによってサポートされている SQL 文。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

備考

クエリ結果は、sasql_store_result または sasql_use_result を使用して取得または格納できます。sasql_field_count 関数を使用すると、クエリから結果セットが返されるかどうかを確認できます。

sasql_query 関数は、この関数と sasql_store_result または sasql_use_result のいずれか 1 つを呼び出すことと同等です。

関連情報

sasql_query [489 ページ]
sasql_store_result [510 ページ]
sasql_use_result [512 ページ]
sasql_field_count [480 ページ]

1.16.2.30 sasql_result_all

すべての結果をフェッチし、オプションのフォーマット文字列に従って HTML 出力テーブルを生成します。

構文

```
bool sasql_result_all( resource $result
[, $html_table_format_string
[, $html_table_header_format_string
[, $html_table_row_format_string
[, $html_table_cell_format_string
] ] ] ] )
```

パラメータ

$result

sasql_query 関数によって返される結果リソース。

$html_table_format_string

HTML テーブルに適用されるフォーマット文字列。たとえば、**"Border=1; Cellpadding=5"** のように記述します。特別な値 none を指定すると、HTML テーブルは作成されません。これは、カラム名やスクリプトをカスタマイズする場合に便利です。このパラメータの明示的値を指定しなくても済むように、パラメータ値には NULL を使用します。

$html_table_header_format_string

HTML テーブルのカラム見出しに適用されるフォーマット文字列。たとえば、**"bgcolor=#FF9533"** のように記述します。特別な値 none を指定すると、HTML テーブルは作成されません。これは、カラム名やスクリプトをカスタマイズする場合に便利です。このパラメータの明示的値を指定しなくても済むように、パラメータ値には NULL を使用します。

$html_table_row_format_string

HTML テーブルのローに適用されるフォーマット文字列。たとえば、**"onclick='alert('this')'"** のように記述します。交互に変わるフォーマットを使用する場合は、特別なトークン >< を使用します。トークンの左側は、奇数ローで使用するフォーマットを示し、トークンの右側は偶数ローで使用するフォーマットを示します。このトークンをフォーマット文字列に含めなかった場合は、すべてのローが同じフォーマットになります。このパラメータに明示的な値を指定したくない場合は、パラメータ値として NULL を使用します。

$html_table_cell_format_string

HTML テーブルローのセルに適用されるフォーマット文字列。たとえば、**"onclick='alert('this')'"** のように記述します。このパラメータに明示的な値を指定したくない場合は、パラメータ値として NULL を使用します。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

備考

`$result` のすべての結果をフェッチし、オプションのフォーマット文字列に従って HTML 出力テーブルを生成します。

関連情報

sasql_query [489 ページ]

1.16.2.31 sasql_rollback

データベースのトランザクションを終了し、トランザクション中に加えられたすべての変更を破棄します。

構文

```
bool sasql_rollback( sasql_conn $conn )
```

パラメータ

$conn

接続関数から返される接続リソース。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

備考

この関数は、auto_commit オプションが Off である場合にのみ有効です。

関連情報

sasql_commit [469 ページ]
sasql_set_option [494 ページ]

1.16.2.32 sasql_set_option

指定した接続で、指定したオプションの値を設定します。

構文

```
bool sasql_set_option( sasql_conn $conn, string $option, mixed $value )
```

パラメータ

$conn

接続関数から返される接続リソース。

$option

設定するオプションの名前。

$value

新しいオプションの値。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

備考

次のオプションの値を設定できます。

名前	説明	デフォルト
auto_commit	このオプションを on に設定すると、データベースサーバは各文の実行後にコミットします。	on
row_counts	このオプションを FALSE に設定すると、sasql_num_rows 関数は影響を受ける推定ロー数を返します。正確なロー数を取得するには、このオプションを TRUE に設定します。	FALSE

名前	説明	デフォルト
verbose_errors	このオプションを TRUE に設定すると、PHP ドライバは冗長エラーを返します。このオプションを FALSE に設定した場合、詳細なエラー情報を取得するには、sasql_error または sasql_errorcode 関数を呼び出してください。	TRUE

php.ini ファイルに次の行を追加することによって、オプションのデフォルト値を変更できます。次の例では、auto_commit オプションのデフォルト値が設定されます。

```
sqlanywhere.auto_commit=0
```

関連情報

sasql_commit [469 ページ]
sasql_error [472 ページ]
sasql_errorcode [473 ページ]
sasql_num_rows [486 ページ]
sasql_rollback [493 ページ]

1.16.2.33 sasql_stmt_affected_rows

文の実行の影響を受けるローの数を返します。

構文

```
int sasql_stmt_affected_rows( sasql_stmt $stmt )
```

パラメータ

$stmt

sasql_stmt_execute によって実行されたステートメントリソース。

戻り値

影響を受けたローの数。失敗した場合は FALSE。

関連情報

sasql_stmt_execute [501 ページ]

1.16.2.34 sasql_stmt_bind_param

PHP 変数を文のパラメータにバインドします。

構文

```
bool sasql_stmt_bind_param( sasql_stmt $stmt, string $types, mixed &$var_1 [, mixed &$var_2 .. ] )
```

パラメータ

$stmt

sasql_prepare 関数によって返された準備ステートメントリソース。

$types

対応するバインドの種類を指定する 1 つ以上の文字を含む文字列。次のいずれかを指定してください。string には *s*、integer には *i*、double には *d*、blob には *b* です。$types 文字列の長さは、$types パラメータに続くパラメータ ($var_1、$var_2、...) の数と一致する必要があります。文字数も、準備文内のパラメータマーカ (疑問符) の数と一致する必要があります。

$var_n

変数の参照。

戻り値

変数のバイドに成功した場合は TRUE、それ以外の場合は FALSE。

関連情報

sasql_prepare [488 ページ]
sasql_stmt_param_count [506 ページ]
sasql_stmt_bind_param_ex [497 ページ]
sasql_stmt_execute [501 ページ]
sasql_stmt_errno [500 ページ]

sasql_stmt_error [501 ページ]

1.16.2.35 sasql_stmt_bind_param_ex

PHP 変数を文のパラメータにバインドします。

構文

```
bool sasql_stmt_bind_param_ex( sasql_stmt $stmt, int $param_number, mixed &$var,
string $type [, bool $is_null [, int $direction ] ] )
```

パラメータ

$stmt

sasql_prepare 関数によって返された準備ステートメントリソース。

$param_number

パラメータ番号。これは 0 ～ (sasql_stmt_param_count($stmt) - 1) の間の数です。

$var

PHP 変数。PHP 変数への参照のみが許可されます。

$type

変数の種類。次のいずれかを指定してください。string には *s*、integer には *i*、double には *d*、blob には *b* です。

$is_null

変数値が NULL であるか否かを示します。

$direction

SASQL_D_INPUT、SASQL_D_OUTPUT、または SASQL_INPUT_OUTPUT を指定できます。

戻り値

変数のバイドに成功した場合は TRUE、それ以外の場合は FALSE。

関連情報

sasql_prepare [488 ページ]
sasql_stmt_param_count [506 ページ]
sasql_stmt_bind_param [496 ページ]

sasql_stmt_execute [501 ページ]

1.16.2.36 sasql_stmt_bind_result

実行されたステートメントの結果カラムに 1 つ以上の PHP 変数をバインドして、結果セットを返します。

構文

```
bool sasql_stmt_bind_result( sasql_stmt $stmt, mixed &$var1 [, mixed &$var2 .. ] )
```

パラメータ

$stmt

sasql_stmt_execute によって実行されたステートメントリソース。

$var1

sasql_stmt_fetch によって返される結果セットのカラムにバインドされる PHP 変数への参照。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

関連情報

sasql_stmt_execute [501 ページ]
sasql_stmt_fetch [502 ページ]

1.16.2.37 sasql_stmt_close

指定されたステートメントリソースを閉じて、関連付けられているリソースを解放します。

構文

```
bool sasql_stmt_close( sasql_stmt $stmt )
```

パラメータ

$stmt

sasql_prepare 関数によって返された準備ステートメントリソース。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

備考

この関数は、sasql_stmt_result_metadata によって返された結果オブジェクトも解放します。

関連情報

sasql_stmt_result_metadata [508 ページ]
sasql_prepare [488 ページ]

1.16.2.38 sasql_stmt_data_seek

結果セット内で指定されたオフセットを検索します。

構文

```
bool sasql_stmt_data_seek( sasql_stmt $stmt, int $offset )
```

パラメータ

$stmt

ステートメントリソース。

$offset

結果セット内のオフセット。これは、0 ～ (sasql_stmt_num_rows($stmt) - 1) の間の数です。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

関連情報

sasql_stmt_num_rows [506 ページ]

1.16.2.39 sasql_stmt_errno

指定されたステートメントリソースを使用して最後に実行された文関数のエラーコードを返します。

構文

```
int sasql_stmt_errno( sasql_stmt $stmt )
```

パラメータ

$stmt

sasql_prepare 関数によって返された準備ステートメントリソース。

戻り値

整数のエラーコード。

関連情報

sasql_stmt_error [501 ページ]
sasql_error [472 ページ]
sasql_errorcode [473 ページ]
sasql_prepare [488 ページ]
sasql_stmt_result_metadata [508 ページ]

1.16.2.40 sasql_stmt_error

指定されたステートメントリソースを使用して最後に実行された文関数のエラーテキストを返します。

構文

```
string sasql_stmt_error( sasql_stmt $stmt )
```

パラメータ

$stmt

sasql_prepare 関数によって返された準備ステートメントリソース。

戻り値

エラーが記述された文字列。

関連情報

sasql_stmt_errno [500 ページ]
sasql_error [472 ページ]
sasql_errorcode [473 ページ]
sasql_prepare [488 ページ]
sasql_stmt_result_metadata [508 ページ]

1.16.2.41 sasql_stmt_execute

準備ステートメントを実行します。sasql_stmt_result_metadata を使用して、ステートメントが結果セットを返すかどうかを確認できます。

構文

```
bool sasql_stmt_execute( sasql_stmt $stmt )
```

パラメータ

$stmt

sasql_prepare 関数によって返された準備ステートメントリソース。変数をバインドしてから実行してください。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

関連情報

sasql_prepare [488 ページ]
sasql_stmt_param_count [506 ページ]
sasql_stmt_bind_param [496 ページ]
sasql_stmt_bind_param_ex [497 ページ]
sasql_stmt_result_metadata [508 ページ]
sasql_stmt_bind_result [498 ページ]

1.16.2.42 sasql_stmt_fetch

文の結果からローを 1 つフェッチし、sasql_stmt_bind_result を使用してバインドされた変数にカラムを配置します。

構文

```
bool sasql_stmt_fetch( sasql_stmt $stmt )
```

パラメータ

$stmt

ステートメントリソース。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

関連情報

sasql_stmt_bind_result [498 ページ]

1.16.2.43 sasql_stmt_field_count

文の結果セット内のカラム数を返します。

構文

```
int sasql_stmt_field_count( sasql_stmt $stmt )
```

パラメータ

$stmt

ステートメントリソース。

戻り値

文の結果内のカラム数。文から結果が返されない場合は 0 を返します。

関連情報

sasql_stmt_result_metadata [508 ページ]

1.16.2.44 sasql_stmt_free_result

キャッシュされた文の結果セットを解放します。

構文

```
bool sasql_stmt_free_result( sasql_stmt $stmt )
```

パラメータ

$stmt

sasql_stmt_execute を使用して実行されたステートメントリソース。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

関連情報

sasql_stmt_execute [501 ページ]
sasql_stmt_store_result [509 ページ]

1.16.2.45 sasql_stmt_insert_id

IDENTITY カラムまたは DEFAULT AUTOINCREMENT カラムに最後に挿入された値を返します。最後に挿入したテーブルに IDENTITY や DEFAULT AUTOINCREMENT カラムが含まれていないと、0 を返します。

構文

```
int sasql_stmt_insert_id( sasql_stmt $stmt )
```

パラメータ

$stmt

sasql_stmt_execute によって実行されたステートメントリソース。

戻り値

前回の INSERT 文によって生成された、IDENTITY カラムまたは DEFAULT AUTOINCREMENT カラムの ID。最後の挿入が IDENTITY カラムまたは DEFAULT AUTOINCREMENT カラムに影響しなかった場合は 0。$stmt が有効でない場合は、FALSE (0) を返します。

関連情報

sasql_stmt_execute [501 ページ]

1.16.2.46 sasql_stmt_next_result

文の次の結果に進みます。

構文

```
bool sasql_stmt_next_result( sasql_stmt $stmt )
```

パラメータ

$stmt

ステートメントリソース。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

備考

別の結果セットがある場合は、現在キャッシュされている結果が破棄されて、それに関連付けられている結果セットオブジェクト (sasql_stmt_result_metadata によって返されたもの) が削除されます。

関連情報

sasql_stmt_result_metadata [508 ページ]

1.16.2.47 sasql_stmt_num_rows

結果セット内のロー数を返します。

構文

```
int sasql_stmt_num_rows( sasql_stmt $stmt )
```

パラメータ

$stmt

sasql_stmt_execute によって実行され、sasql_stmt_store_result が呼び出されたステートメントリソース。

戻り値

結果で使用できるロー数。失敗した場合は 0。

備考

結果セット内の実際のロー数は、sasql_stmt_store_result 関数が呼び出されて結果セット全体がバッファに格納された後でのみ特定できます。sasql_stmt_store_result 関数が呼び出されなかった場合は 0 が返されます。

関連情報

sasql_stmt_execute [501 ページ]
sasql_stmt_store_result [509 ページ]

1.16.2.48 sasql_stmt_param_count

指定された準備文リソース内のパラメータ数を返します。

構文

```
int sasql_stmt_param_count( sasql_stmt $stmt )
```

パラメータ

$stmt

sasql_prepare 関数によって返されるステートメントリソース。

戻り値

パラメータ数。エラーの場合は FALSE。

関連情報

sasql_prepare [488 ページ]
sasql_stmt_bind_param [496 ページ]
sasql_stmt_bind_param_ex [497 ページ]

1.16.2.49 sasql_stmt_reset

文オブジェクトを記述直後の状態にリセットします。

構文

```
bool sasql_stmt_reset( sasql_stmt $stmt )
```

パラメータ

$stmt

ステートメントリソース。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

備考

この関数は、`$stmt` オブジェクトを記述直後の状態にリセットします。バインドされた変数はすべてバインドを解除され、sasql_stmt_send_long_data を使用して送信されたデータはすべて削除されます。

関連情報

sasql_stmt_send_long_data [508 ページ]

1.16.2.50 sasql_stmt_result_metadata

指定された文の結果セットオブジェクトを返します。

構文

```
sasql_result sasql_stmt_result_metadata( sasql_stmt $stmt )
```

パラメータ

$stmt

準備され、実行されたステートメントリソース。

戻り値

sasql_result オブジェクト。文から結果が返されない場合は FALSE。

1.16.2.51 sasql_stmt_send_long_data

ユーザがパラメータデータをチャンク単位で送信できるようにします。

構文

```
bool sasql_stmt_send_long_data( sasql_stmt $stmt, int $param_number, string $data )
```

パラメータ

$stmt

sasql_prepare を使用して準備されたステートメントリソース。

$param_number

パラメータ番号。これは 0 ～ (sasql_stmt_param_count($stmt) - 1) の間の数です。

$data

送信されるデータ。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

備考

ユーザは、最初に sasql_stmt_bind_param または sasql_stmt_bind_param_ex を呼び出してから、データを送信します。バインドパラメータのデータ型は string または blob にする必要があります。この関数を繰り返して呼び出すと、結果は前に送信された内容に追加されます。

関連情報

sasql_stmt_bind_param [496 ページ]
sasql_stmt_bind_param_ex [497 ページ]
sasql_prepare [488 ページ]
sasql_stmt_param_count [506 ページ]

1.16.2.52 sasql_stmt_store_result

クライアントで文の結果セット全体をキャッシュできるようになります。

構文

```
bool sasql_stmt_store_result( sasql_stmt $stmt )
```

パラメータ

$stmt

sasql_stmt_execute を使用して実行されたステートメントリソース。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

備考

キャッシュされた結果は、関数 sasql_stmt_free_result を使用して解放できます。

関連情報

sasql_stmt_free_result [503 ページ]
sasql_stmt_execute [501 ページ]

1.16.2.53 sasql_store_result

最後のクエリの結果セットを転送します。

構文

```
sasql_result sasql_store_result( sasql_conn $conn )
```

パラメータ

$conn

接続関数から返される接続リソース。

戻り値

クエリが結果オブジェクトを返さない場合は FALSE。それ以外の場合は、結果のすべてのローを含む結果セットオブジェクト。結果はクライアントでキャッシュされます。

備考

この関数は、sasql_real_query の実行後に呼び出されます。sasql_real_query の実行に成功したら、sasql_store_result を使用し、クエリから返された結果セットに対するハンドルを取得します。結果セットハンドルは、サーバからフェッチされたデータを表します。

sasql_store_result 関数はクライアント側で結果セットをキャッシュします。クライアント側の結果セットのキャッシュを防ぐには、代わりに sasql_use_result を使用します。

例

```
<?php
    $conn = sasql_connect( "UID=DBA;PWD=sql" )
        or die( "Cannot connect to database" );
    if( sasql_real_query( $conn, "SELECT * FROM Customers" ) )
    {
        $num_cols = sasql_field_count( $conn );
        $result = sasql_store_result( $conn );
        while( $row = sasql_fetch_row( $result ) )
        {
            $curr_row++;
            $curr_col = 0;
            while( $curr_col < $num_cols ) {
                echo "$row[$curr_col]¥t|";
                $curr_col++;
            }
            echo "¥n";
        }
        sasql_free_result( $result );
        echo "$curr_row rows.¥n";
    }
    sasql_close( $conn );
?>
```

関連情報

sasql_data_seek [471 ページ]

sasql_real_query [491 ページ]

sasql_free_result [481 ページ]

sasql_use_result [512 ページ]

1.16.2.54 sasql_sqlstate

最新の SQLSTATE 文字列を返します。

構文

```
string sasql_sqlstate( sasql_conn $conn )
```

パラメータ

$conn

接続関数から返される接続リソース。

戻り値

現在の SQLSTATE コードを含む 5 文字の文字列を返します。「00000」という結果はエラーがないことを示します。

備考

SQLSTATE は、最後に実行された SQL 文が成功、エラー、または警告条件になったかどうかを示します。SQLSTATE コードは 5 文字で構成され、「00000」はエラーがないことを示します。この値は ISO/ANSI SQL 標準で定められています。

関連情報

sasql_error [472 ページ]
sasql_errorcode [473 ページ]

1.16.2.55 sasql_use_result

接続で最後に実行されたクエリの結果セットの取得を開始します。

構文

```
sasql_result sasql_use_result( sasql_conn $conn )
```

パラメータ

$conn

接続関数から返される接続リソース。

戻り値

クエリが結果オブジェクトを返さない場合は FALSE。それ以外の場合は、結果セットオブジェクト。結果はクライアントでキャッシュされません。

備考

この関数は、sasql_real_query の実行後に呼び出されます。sasql_real_query の実行に成功したら、sasql_use_result を使用し、クエリから返された結果セットに対するハンドルを取得します。結果セットハンドルは、サーバからまだフェッチされていないデータを表します。この結果セットハンドルを使用する各フェッチは、サーバに要求を送信し、結果セットのローを取得します。

sasql_use_result 関数は、クライアント側で結果セットをキャッシュできないようにします。クライアント側の結果セットをキャッシュするには、代わりに sasql_store_result を使用します。

例

```
<?php
    $conn = sasql_connect( "UID=DBA;PWD=sql" )
        or die( "Cannot connect to database" );
    if( sasql_real_query( $conn, "SELECT * FROM Customers" ) )
    {
        $num_cols = sasql_field_count( $conn );
        $result = sasql_use_result( $conn );
        while( $row = sasql_fetch_row( $result ) )
        {
            $curr_row++;
            $curr_col = 0;
            while( $curr_col < $num_cols ) {
                echo "$row[$curr_col]¥t|";
                $curr_col++;
            }
            echo "¥n";
        }
        sasql_free_result( $result );
        echo "$curr_row rows.¥n";
    }
    sasql_close( $conn );
?>
```

関連情報

sasql_data_seek [471 ページ]
sasql_real_query [491 ページ]
sasql_free_result [481 ページ]
sasql_store_result [510 ページ]

1.17 Ruby サポート

Ruby を使用してデータベースアプリケーションを開発できます。

このセクションの内容:

Ruby プログラミング [514 ページ]
SQL Anywhere では、3 種類の Ruby API (Application Programming Interface) がサポートされています。

SQL Anywhere Ruby API リファレンス [524 ページ]
Ruby 拡張 API によって、SQL アプリケーションの迅速な開発が可能になります。この拡張は、SQL Anywhere C API と呼ばれる低レベルのインタフェースを基礎として構築されます。

1.17.1 Ruby プログラミング

SQL Anywhere では、3 種類の Ruby API (Application Programming Interface) がサポートされています。

1 つ目はネイティブ Ruby API です。この API は、SQL Anywhere C API によって公開されているインタフェースを Ruby でラップします。

2 つ目は ActiveRecord のサポートです。ActiveRecord は、Web 開発フレームワーク Ruby on Rails の一部として普及しているオブジェクト関係マッピングです。

3 つ目は Ruby DBI のサポートです。DBI とともに使用できる Ruby データベースドライバ (DBD) が提供されています。

Ruby プロジェクトの SQL Anywhere では 3 つの別個のパッケージを利用できます。これらのパッケージをインストールする最も簡単な方法は、RubyGems を使用することです。

> **注記**
>
> Mac OS X 10.11 を使用している場合は、SQLANY_API_DLL 環境変数を `libdbcapi_r.dylib` へのフルパスに設定してください。

ネイティブ Ruby ドライバ

sqlanywhere

このパッケージは Ruby コードから SQL Anywhere データベースへのインタフェースを可能にする低レベルのドライバです。このパッケージは、SQL Anywhere C API によって公開されているインタフェースを Ruby でラップします。このパッケージは C 言語で記述され、Windows と Linux 用に、ソースまたは事前にコンパイルされた gem として提供されています。RubyGems がインストールされている場合は、次のコマンドを実行してこのパッケージを入手できます。

```
gem install sqlanywhere
```

他の Ruby インタフェースを使用するには、このパッケージが必要です。

ActiveRecord アダプタ

activerecord-sqlanywhere-adapter

このパッケージは、ActiveRecord とデータベースサーバの対話を可能にするアダプタです。ActiveRecord は、Web 開発フレームワーク Ruby on Rails の一部として普及しているオブジェクト関係マッピングです。このパッケージは Pure Ruby で記述され、ソースまたは gem フォーマットで提供されています。このアダプタでは sqlanywhere gem が使用され、この sqlanywhere gem に依存します。RubyGems がインストールされている場合は、次のコマンドを実行してこのパッケージとその依存ファイルをインストールできます。

```
gem install activerecord-sqlanywhere-adapter
```

Ruby/DBI ドライバ

dbi

このパッケージは Ruby 用の DBI ドライバです。RubyGems がインストールされている場合は、次のコマンドを実行してこのパッケージとその依存ファイルをインストールできます。

```
gem install dbi
```

dbd-sqlanywhere

このパッケージは、Ruby/DBI とデータベースサーバの対話を可能にするドライバです。Ruby/DBI は一般的な Perl DBI モジュールをモデルとする汎用のデータベースインタフェースです。このパッケージは Pure Ruby で記述され、ソースまたは gem フォーマットで提供されています。このドライバでは sqlanywhere gem が使用され、この sqlanywhere gem に依存します。RubyGems がインストールされている場合は、次のコマンドを実行してこのパッケージとその依存ファイルをインストールできます。

```
gem install dbd-sqlanywhere
```

このセクションの内容:

Ruby on Rails サポート [516 ページ]

Rails は、Ruby 言語で記述された Web 開発フレームワークです。その強みは、Web アプリケーションの開発にあります。Rails で開発を行う前に Ruby プログラミング言語に慣れておくことを強くおすすめします。

Ruby-DBI ドライバ [520 ページ]
SQL Anywhere DBI ドライバを使用する Ruby アプリケーションを作成できます。

関連情報

SQL Anywhere Ruby API リファレンス [524 ページ]
Ruby プログラミング
Ruby Gems
SQL Anywhere Ruby ドライバ
Ruby/DBI - Ruby のダイレクトデータベースアクセスレイヤ
Ruby on Rails
SQL Anywhere フォーラム

1.17.1.1 Ruby on Rails サポート

Rails は、Ruby 言語で記述された Web 開発フレームワークです。その強みは、Web アプリケーションの開発にあります。Rails で開発を行う前に Ruby プログラミング言語に慣れておくことを強くおすすめします。

Rails による開発を始める前に、Rails を設定する必要があります。

Rails の設定が完了したら、Ruby on Rails Web サイトのチュートリアルを試してください。チュートリアルを SQL Anywhere に適応させる手順は下記にあります。

このセクションの内容:

Rails サポートの設定 [517 ページ]
Rails によってサポートされたデータベース管理システムのセットに SQL Anywhere を追加します。

Rails の学習 [519 ページ]
Ruby on Rails Web サイトのチュートリアルで Rails 開発について学習します。

関連情報

SQL Anywhere Ruby API リファレンス [524 ページ]

1.17.1.1.1 Rails サポートの設定

Rails によってサポートされたデータベース管理システムのセットに SQL Anywhere を追加します。

手順

1. システムに Ruby のインタプリターをインストールします。
2. RubyGems をインストールします。これにより、Ruby パッケージのインストールが簡単になります。インストールする推奨バージョンについては、Ruby on Rails のダウンロードページを参照してください。
3. 次のコマンドを実行し、Rails とその依存性をインストールします。

   ```
   gem install rails
   ```

4. 適切な Ruby Development Kit (DevKit) をインストールします。
5. 次のコマンドを実行し、ActiveRecord サポート (activerecord-sqlanywhere-adapter) をインストールします。

   ```
   gem install activerecord-sqlanywhere-adapter
   ```

6. Rails によってサポートされたデータベース管理システムのセットに SQL Anywhere を追加します。本書作成時点での最新リリースバージョンは、Rails 3.1.13 です。
   a. Rails の `configs¥databases` ディレクトリに `sqlanywhere.yml` ファイルを作成し、データベースを設定します。Ruby を `¥Ruby` ディレクトリにインストールし、Rails のバージョン 3.1.13 をインストールした場合、このファイルへのパスは `¥Ruby¥lib¥ruby¥gems¥1.9.1¥gems¥railties-3.1.13¥lib¥rails¥generators¥rails¥app¥templates¥config¥databases` になります。このファイルの内容は次のようになります。

   ```
   #
   # SQL Anywhere database configuration
   #
   # This configuration file defines the patten used for
   # database filenames. If your application is called "blog",
   # then the database names will be blog_development,
   # blog_test, blog_production. The specified username and
   # password should permit DBA access to the database.
   #
   development:
     adapter: sqlanywhere
     server: <%= app_name %>
     database: <%= app_name %>_development
     username: DBA
     password: passwd
   # Warning: The database defined as "test" will be erased and
   # re-generated from your development database when you run "rake".
   # Do not set this db to the same as development or production.
   test:
     adapter: sqlanywhere
     server: <%= app_name %>
     database: <%= app_name %>_test
     username: DBA
     password: passwd
   production:
     adapter: sqlanywhere
     server: <%= app_name %>
     database: <%= app_name %>_production
     username: DBA
   ```

```
  password: passwd
```

sqlanywhere.yml ファイルは、Rails プロジェクトで database.yml ファイルを作成するためのテンプレートになります。次のデータベースオプションを指定できます。

adapter

(必須、デフォルトなし)。ActiveRecord アダプタを使用する場合、このオプションを sqlanywhere に設定する必要があります。

database

(必須、デフォルトなし)。このオプションは、接続文字列の "DatabaseName" に対応しています。

server

(オプション、デフォルトは *database* オプション)。このオプションは、接続文字列の "ServerName" に対応しています。

username

このオプションは、接続文字列の "UserID" に対応しています。

password

このオプションは、接続文字列の "Password" に対応しています。

encoding

(オプション、デフォルトは OS の文字セット)。このオプションは、接続文字列の "CharSet" に対応しています。

commlinks

(省略可)。このオプションは、接続文字列の "CommLinks" に対応しています。

connection_name

(省略可)。このオプションは、接続文字列の "ConnectionName" に対応しています。

b. Rails の app_base.rb ファイルを更新します。以前の手順と同じ条件の場合、このファイルはパス ¥Ruby¥lib¥ruby¥gems¥1.9.1¥gems¥railties-3.2.13¥lib¥rails¥generators¥app_base.rb にあります。app_base.rb ファイルを編集し、次の行を検索します。

```
DATABASES = %w( mysql oracle postgresql sqlite3 frontbase ibm_db sqlserver )
```

このリストに次のように sqlanywhere を追加します。

```
DATABASES = %w( sqlanywhere mysql oracle postgresql sqlite3 frontbase ibm_db
sqlserver )
```

結果

これで Rails サポートが正常に設定されました。

関連情報

Ruby on Rails

ダウンロード
Ruby Development Kit (DevKit)

1.17.1.1.2 Rails の学習

Ruby on Rails Web サイトのチュートリアルで Rails 開発について学習します。

前提条件

Rails は、SQL Anywhere 用に設定する必要があります。

Ruby on Rails Web サイト () のチュートリアルを下記のステップと組み合わせ、Rails 開発に習熟します。

手順

1. このチュートリアルには、**blog** プロジェクトを初期化するコマンドが記載されています。SQL Anywhere で使用するように **blog** プロジェクトを初期化するコマンドは次のようになります。

   ```
   rails new blog -d sqlanywhere
   ```

2. **blog** アプリケーションを作成したら、そのフォルダに移動して、そのアプリケーションで直接作業を続けます。

   ```
   cd blog
   ```

3. `Gemfile` ファイルを編集して、SQL Anywhere ActiveRecord アダプタ用の *gem* ディレクティブを含めます。下に示す行の後に新しいディレクティブを追加します。

   ```
   gem 'sqlanywhere'
   gem 'activerecord-sqlanywhere-adapter'
   ```

4. `config¥database.yml` ファイルは、開発、テスト、運用データベースを参照します。チュートリアルで示されているように *rake* コマンドでデータベースを作成する代わりに、プロジェクトの `db` ディレクトリに移動して、次のように 3 つのデータベースを作成します。

   ```
   cd db
   dbinit -dba DBA,passwd blog_development
   dbinit -dba DBA,passwd blog_test
   dbinit -dba DBA,passwd blog_production
   ```

5. データベースサーバと 3 つのデータベースを起動し、次のように `blog` ディレクトリに変更します。

   ```
   dbsrv17 -n blog blog_development.db blog_production.db blog_test.db
   cd ..
   ```

   コマンドラインのデータベースサーバ名 (`blog`) は、`config¥database.yml` ファイルの *server:* タグで指定した名前と同じにする必要があります。`sqlanywhere.yml` テンプレートファイルは、データベースサーバ名が、生成されるすべての `database.yml` ファイルのプロジェクト名と必ず同じになるように設定されています。

6. チュートリアルの残りの手順に従います。

結果

これで Ruby on Rails ソフトウェア開発の準備が整いました。

関連情報

Rails サポートの設定 [517 ページ]
Rails について

1.17.1.2 Ruby-DBI ドライバ

SQL Anywhere DBI ドライバを使用する Ruby アプリケーションを作成できます。

DBI モジュールのロード

Ruby アプリケーションから DBI:SQLAnywhere インタフェースを使用するには、Ruby DBI モジュールを使用する予定であることを最初に Ruby に通知する必要があります。これを行うには、Ruby のソースファイルの先頭近くに次の行を挿入します。

```
require 'dbi'
```

DBI モジュールは、必要に応じてデータベースドライバ (DBD) インタフェースを自動的にロードします。

接続を開いて閉じる

通常、データベースに対して 1 つの接続を開いてから、一連の SQL 文を実行して必要なすべての操作を実行します。接続を開くには、connect 関数を使用します。この戻り値は、接続時に後続の操作を行うために使用するデータベース接続のハンドルです。

connect 関数の呼び出しは、一般的に次の形式で行います。

```
dbh = DBI.connect('DBI:SQLAnywhere:server-name', user-id, password, options)
```

server-name

接続先のデータベースサーバ名です。"option1=value1;option2=value2;..." というフォーマットで接続文字列を指定することもできます。

user-id

有効なユーザ ID です。この文字列がブランクでないかぎり、接続文字列に ";UID=value" が追加されます。

password

ユーザ ID に対応するパスワードです。この文字列がブランクでないかぎり、接続文字列に ";PWD=value" が追加されます。

options

DatabaseName、DatabaseFile、ConnectionName などの追加接続パラメータのハッシュです。"option1=value1;option2=value2;..." というフォーマットで接続文字列に付加されます。

connect 関数を使用してみるには、データベースサーバとサンプルデータベースを起動してからサンプルの Ruby スクリプトを実行します。

```
dbsrv17 -n myserver "%SQLANYSAMP17%¥demo.db"
```

次のコードサンプルは、サンプルデータベースへの接続を開いて閉じます。次の例の文字列 "myserver" はサーバ名です。

```
require 'dbi'
DBI.connect('DBI:SQLAnywhere:myserver', 'DBA', 'sql') do |dbh|
    if dbh.ping
       print "Successfully Connected¥n"
       dbh.disconnect()
    end
end
```

サーバ名の代わりに接続文字列を指定することもできます。たとえば、上記の場合、connect 関数の最初のパラメータを次のように置き換えることにより、スクリプトを変更できます。

```
require 'dbi'
DBI.connect('DBI:SQLAnywhere:SERVER=myserver;DBN=demo', 'DBA', 'sql') do |dbh|
    if dbh.ping
       print "Successfully Connected¥n"
       dbh.disconnect()
    end
end
```

接続文字列にはユーザ ID とパスワードを指定できません。これらの引数を省略すると、Ruby DBI によってデフォルトのユーザ名とパスワードが自動的に入力されるため、UID または PWD 接続パラメータを接続文字列に含めないでください。含めた場合は、例外がスローされます。

次の例は、キーワードと値のペアのハッシュとして、追加接続パラメータを connect 関数に渡す方法を示しています。

```
require 'dbi'
DBI.connect('DBI:SQLAnywhere:myserver', 'DBA', 'sql',
       { :ConnectionName => "RubyDemo",
         :DatabaseFile => "demo.db",
         :DatabaseName => "demo" }
        ) do |dbh|
    if dbh.ping
       print "Successfully Connected¥n"
       dbh.disconnect()
    end
end
```

データの選択

開かれた接続へのハンドルを取得したら、データベースに格納されているデータにアクセスして修正できます。最も単純な操作は、おそらくいくつかのローを取得して出力することです。

SQL 文は最初に実行する必要があります。文から結果セットが返された場合、ステートメントハンドルを使用して、結果セットに関するメタ情報と、結果セットのローを取得できます。次の例では、メタデータからカラム名を取得し、フェッチされた各ローのカラム名と値を表示しています。

```
require 'dbi'
def db_query( dbh, sql )
    sth = dbh.execute(sql)
    print "# of Fields:  #{sth.column_names.size}¥n"
    sth.fetch do |row|
        print "¥n"
        sth.column_info.each_with_index do |info, i|
            unless info["type_name"] == "LONG VARBINARY"
                print "#{info["name"]}=#{row[i]}¥n"
            end
        end
    end
    sth.finish
end
begin
    dbh = DBI.connect('DBI:SQLAnywhere:demo', 'DBA', 'sql')
    db_query(dbh, "SELECT * FROM Products")
rescue DBI::DatabaseError => e
    puts "An error occurred"
    puts "Error code: #{e.err}"
    puts "Error message: #{e.errstr}"
    puts "Error SQLSTATE: #{e.state}"
ensure
    dbh.disconnect if dbh
end
```

表示される出力の最初の数行を次に示します。

```
# of Fields:  8
ID=300
Name=Tee Shirt
Description=Tank Top
Size=Small
Color=White
Quantity=28
UnitPrice=9.00
ID=301
Name=Tee Shirt
Description=V-neck
Size=Medium
Color=Orange
Quantity=54
UnitPrice=14.00
```

終了したら、finish を呼び出してステートメントハンドルを解放することが重要です。finish を呼び出さなかった場合、次のようなエラーが表示される場合があります。

```
Resource governor for 'prepared statements' exceeded
```

ハンドルのリークを検出するために、データベースサーバでは、カーソルと準備文の数はデフォルトで接続ごとに最大 50 に制限されています。これらの制限を越えると、リソースガバナーによってエラーが自動的に生成されます。このエラーが発生し

たら、破棄されていない文のハンドルを確認してください。文のハンドルが破棄されていない場合は、prepare_cached を慎重に使用してください。

必要な場合、max_cursor_count と max_statement_count オプションを設定してこれらの制限を変更できます。

ローの挿入

ローを挿入するには、開かれた接続へのハンドルが必要です。ローを挿入する最も簡単な方法は、パラメータ化された INSERT 文を使用する方法です。この場合、疑問符が値のプレースホルダとして使用されます。この文は最初に準備されてから、新しいローごとに 1 回実行されます。新しいローの値は、execute メソッドのパラメータとして指定されます。

```
require 'dbi'
def db_query( dbh, sql )
    sth = dbh.execute(sql)
    print "# of Fields:  #{sth.column_names.size}¥n"
    sth.fetch do |row|
        print "¥n"
        sth.column_info.each_with_index do |info, i|
            unless info["type_name"] == "LONG VARBINARY"
                print "#{info["name"]}=#{row[i]}¥n"
            end
        end
    end
    sth.finish
end
def db_insert( dbh, rows )
    sql = "INSERT INTO Customers (ID, GivenName, Surname,
                Street, City, State, Country, PostalCode,
                Phone, CompanyName)
           VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)"
    sth = dbh.prepare(sql);
    rows.each do |row|
        sth.execute(row[0],row[1],row[2],row[3],row[4],
                    row[5],row[6],row[7],row[8],row[9])
    end
end
begin
    dbh = DBI.connect('DBI:SQLAnywhere:demo', 'DBA', 'sql')
    rows = [
        [801,'Alex','Alt','5 Blue Ave','New York','NY','USA',
            '10012','5185553434','BXM'],
        [802,'Zach','Zed','82 Fair St','New York','NY','USA',
            '10033','5185552234','Zap']
       ]
    db_insert(dbh, rows)
    dbh.commit
    db_query(dbh, "SELECT * FROM Customers WHERE ID > 800")
rescue DBI::DatabaseError => e
    puts "An error occurred"
    puts "Error code: #{e.err}"
    puts "Error message: #{e.errstr}"
    puts "Error SQLSTATE: #{e.state}"
ensure
    dbh.disconnect if dbh
end
```

関連情報

Ruby/DBI - Ruby のダイレクトデータベースアクセスレイヤ

1.17.2 SQL Anywhere Ruby API リファレンス

Ruby 拡張 API によって、SQL アプリケーションの迅速な開発が可能になります。この拡張は、SQL Anywhere C API と呼ばれる低レベルのインタフェースを基礎として構築されます。

Ruby によるアプリケーション開発の長所を示すために、次の Ruby のサンプルプログラムについて考えてみます。このプログラムは、Ruby 拡張をロードし、サンプルデータベースに接続し、Products テーブルのカラム値を表示し、接続を切断し、終了します。

```
begin
    require 'rubygems'
    gem 'sqlanywhere'
    unless defined? SQLAnywhere
        require 'sqlanywhere'
    end
end
api = SQLAnywhere::SQLAnywhereInterface.new()
SQLAnywhere::API.sqlany_initialize_interface( api )
api.sqlany_init()
conn = api.sqlany_new_connection()
puts "Enter user ID"
myuid = STDIN.gets.chomp
puts "Enter password"
mypwd = STDIN.gets.chomp
api.sqlany_connect( conn, "DSN=SQL Anywhere 17 Demo;UserID="+myuid
+";Password="+mypwd )
stmt = api.sqlany_execute_direct( conn, "SELECT * FROM Products" )
num_rows = api.sqlany_num_rows( stmt )
num_rows.times {
    api.sqlany_fetch_next( stmt )
    num_cols = api.sqlany_num_cols( stmt )
    for col in 1..num_cols do
        info = api.sqlany_get_column_info( stmt, col - 1 )
        unless info[3]==1 # Don't do binary
            rc, value = api.sqlany_get_column( stmt, col - 1 )
            print "#{info[2]}=#{value}¥n"
        end
    end
    print "¥n"
}
api.sqlany_free_stmt( stmt )
api.sqlany_disconnect(conn)
api.sqlany_free_connection(conn)
api.sqlany_fini()
SQLAnywhere::API.sqlany_finalize_interface( api )
```

この Ruby プログラムの結果セットから出力される最初の 2 つのローは、次のとおりです。

```
ID=300
Name=Tee Shirt
Description=Tank Top
Size=Small
Color=White
Quantity=28
UnitPrice=9.00
```

```
ID=301
Name=Tee Shirt
Description=V-neck
Size=Medium
Color=Orange
Quantity=54
UnitPrice=14.00
```

このセクションの内容:

sqlany_affected_rows [526 ページ]
準備文の実行の影響を受けるローの数を返します。

sqlany_bind_param 関数 [527 ページ]
ユーザが指定するバッファを準備文のパラメータとしてバインドします。

sqlany_clear_error 関数 [528 ページ]
最後に格納されたエラーコードをクリアします。

sqlany_client_version 関数 [529 ページ]
現在のクライアントバージョンを返します。

sqlany_commit 関数 [529 ページ]
現在のトランザクションをコミットします。

sqlany_connect 関数 [530 ページ]
指定された接続オブジェクトと接続文字列を使用して、データベースサーバへの接続を作成します。

sqlany_describe_bind_param 関数 [531 ページ]
準備文のバインドパラメータを記述します。

sqlany_disconnect 関数 [532 ページ]
データベース接続を切断します。コミットされていないトランザクションはすべてロールバックされます。

sqlany_error 関数 [533 ページ]
接続オブジェクトに最後に格納されたエラーコードとエラーメッセージを返します。

sqlany_execute 関数 [534 ページ]
準備文を実行します。

sqlany_execute_direct 関数 [535 ページ]
文字列引数によって指定された SQL 文を実行します。

sqlany_execute_immediate 関数 [536 ページ]
指定された SQL 文を、結果セットを返さずにただちに実行します。結果セットを返さない文の場合に使用すると便利です。

sqlany_fetch_absolute 関数 [537 ページ]
結果セット内の現在のローを、指定されたロー番号に移し、そのローのデータをフェッチします。

sqlany_fetch_next 関数 [538 ページ]
結果セットから次のローを返します。この関数は、ローポインタを進めてから新しいローのデータをフェッチします。

sqlany_fini 関数 [539 ページ]
API によって割り付けられたリソースを解放します。

sqlany_free_connection 関数 [539 ページ]
接続オブジェクトに関連付けられているリソースを解放します。

sqlany_free_stmt 関数 [540 ページ]
文オブジェクトに関連付けられているリソースを解放します。

sqlany_get_bind_param_info 関数 [541 ページ]
sqlany_bind_param を使用してバインドされたパラメータに関する情報を取得します。

sqlany_get_column 関数 [542 ページ]
指定されたカラムについてフェッチされた値を返します。

sqlany_get_column_info 関数 [543 ページ]
指定された結果セットのカラムに対するカラム情報を取得します。

sqlany_get_next_result 関数 [544 ページ]
複数の結果セットクエリ内の次の結果セットに進みます。

sqlany_init 関数 [545 ページ]
インタフェースを初期化します。

sqlany_new_connection 関数 [546 ページ]
接続オブジェクトを作成します。

sqlany_num_cols 関数 [547 ページ]
結果セット内のカラム数を返します。

sqlany_num_params 関数 [548 ページ]
準備文で必要とされるパラメータ数を返します。

sqlany_num_rows 関数 [548 ページ]
結果セット内のロー数を返します。

sqlany_prepare 関数 [549 ページ]
指定された SQL 文字列を準備します。

sqlany_rollback 関数 [551 ページ]
現在のトランザクションをロールバックします。

sqlany_sqlstate 関数 [551 ページ]
現在の SQLSTATE を取得します。

カラムの型 [552 ページ]
次の Ruby クラスで、一部の Ruby 拡張関数によって返されるカラムの型が定義されています。

ネイティブのカラム型 [553 ページ]
次の表に、一部の Ruby 拡張関数によって返されるネイティブのカラム型を示します。

1.17.2.1 sqlany_affected_rows

準備文の実行の影響を受けるローの数を返します。

構文

```
sqlany_affected_rows ( $stmt )
```

パラメータ

$stmt

準備および実行が成功したものの、結果セットが返されなかった文。たとえば、INSERT 文、UPDATE 文、または DELETE 文が実行された場合です。

戻り値

影響を受けたローの数のスカラー値を返します。失敗した場合は -1 を返します。

例

```
affected = api.sqlany_affected( stmt )
```

関連情報

sqlany_execute 関数 [534 ページ]

1.17.2.2 sqlany_bind_param 関数

ユーザが指定するバッファを準備文のパラメータとしてバインドします。

構文

```
sqlany_bind_param ( $stmt, $index, $param )
```

パラメータ

$stmt

sqlany_prepare の実行によって返された文オブジェクト。

$index

パラメータのインデックス。数値は、0 ～ sqlany_num_params() - 1 の間である必要があります。

$param

sqlany_describe_bind_param から取得された、設定済みのバインドオブジェクト。

戻り値

成功した場合はスカラー値 1、失敗した場合は 0 を返します。

例

```
stmt = api.sqlany_prepare(conn, "UPDATE Contacts
     SET Contacts.ID = Contacts.ID + 1000
     WHERE Contacts.ID >= ?" )
rc, param = api.sqlany_describe_bind_param( stmt, 0 )
print "Param name = ", param.get_name(), "¥n"
print "Param dir = ", param.get_direction(), "¥n"
param.set_value(50)
rc = api.sqlany_bind_param( stmt, 0, param )
```

関連情報

sqlany_describe_bind_param 関数 [531 ページ]

1.17.2.3 sqlany_clear_error 関数

最後に格納されたエラーコードをクリアします。

構文

```
sqlany_clear_error ( $conn )
```

パラメータ

$conn

sqlany_new_connection から返された接続オブジェクト。

戻り値

NULL を返します。

例

```
api.sqlany_clear_error( conn )
```

関連情報

sqlany_new_connection 関数 [546 ページ]

1.17.2.4 sqlany_client_version 関数

現在のクライアントバージョンを返します。

構文

```
sqlany_client_version ( )
```

戻り値

クライアントのバージョン文字列のスカラー値を返します。

例

```
buffer = api.sqlany_client_version()
```

1.17.2.5 sqlany_commit 関数

現在のトランザクションをコミットします。

構文

```
sqlany_commit ( $conn )
```

パラメータ

$conn

コミット操作が実行される接続オブジェクト。

戻り値

成功した場合はスカラー値 1、失敗した場合は 0 を返します。

例

```
rc = api.sqlany_commit( conn )
```

関連情報

sqlany_rollback 関数 [551 ページ]

1.17.2.6 sqlany_connect 関数

指定された接続オブジェクトと接続文字列を使用して、データベースサーバへの接続を作成します。

構文

```
sqlany_connect ( $conn, $str )
```

パラメータ

$conn

sqlany_new_connection によって作成された接続オブジェクト。

$str

有効な接続文字列。

戻り値

接続が確立された場合はスカラー値 1、接続が失敗した場合は 0 を返します。sqlany_error を使用してエラーコードとエラーメッセージを取得します。

例

```
require 'sqlanywhere'
api = SQLAnywhere::SQLAnywhereInterface.new()
SQLAnywhere::API.sqlany_initialize_interface( api )
```

```
api.sqlany_init()
# Create a connection
conn = api.sqlany_new_connection()
puts "Enter user ID"
myuid = STDIN.gets.chomp
puts "Enter password"
mypwd = STDIN.gets.chomp
# Establish a connection
status = api.sqlany_connect( conn, "UID="+myuid+";PWD="+mypwd )
print "Connection status = #{status}¥n"
```

関連情報

sqlany_new_connection 関数 [546 ページ]
sqlany_error 関数 [533 ページ]

1.17.2.7 sqlany_describe_bind_param 関数

準備文のバインドパラメータを記述します。

構文

```
sqlany_describe_bind_param ( $stmt, $index )
```

パラメータ

$stmt

sqlany_prepare を使用して準備された文。

$index

パラメータのインデックス。数値は、0 ～ sqlany_num_params() - 1 の間である必要があります。

戻り値

2 要素の配列を返します。最初の要素は、成功した場合は 1、失敗した場合は 0 です。2 番目の要素は記述されたパラメータです。

備考

この関数により、呼び出し元は準備文のパラメータに関する情報を判断できます。提供される情報の量は、準備文のタイプ (ストアドプロシージャまたは DML) によって決まります。パラメータの方向 (入力、出力、または入出力) に関する情報は常に提供されます。

例

```
stmt = api.sqlany_prepare(conn, "UPDATE Contacts
     SET Contacts.ID = Contacts.ID + 1000
     WHERE Contacts.ID >= ?" )
rc, param = api.sqlany_describe_bind_param( stmt, 0 )
print "Param name = ", param.get_name(), "¥n"
print "Param dir = ", param.get_direction(), "¥n"
param.set_value(50)
rc = api.sqlany_bind_param( stmt, 0, param )
```

関連情報

sqlany_bind_param 関数 [527 ページ]
sqlany_prepare 関数 [549 ページ]

1.17.2.8 sqlany_disconnect 関数

データベース接続を切断します。コミットされていないトランザクションはすべてロールバックされます。

構文

```
sqlany_disconnect ( $conn )
```

パラメータ

$conn

sqlany_connect を使用して確立された接続の接続オブジェクト。

戻り値

成功した場合はスカラー値 1、失敗した場合は 0 を返します。

例

```
# Disconnect from the database
status = api.sqlany_disconnect( conn )
print "Disconnect status = #{status}¥n"
```

関連情報

sqlany_connect 関数 [530 ページ]
sqlany_new_connection 関数 [546 ページ]

1.17.2.9 sqlany_error 関数

接続オブジェクトに最後に格納されたエラーコードとエラーメッセージを返します。

構文

```
sqlany_error ( $conn )
```

パラメータ

$conn

sqlany_new_connection から返された接続オブジェクト。

戻り値

2 要素の配列を返します。最初の要素は SQL エラーコード、2 番目の要素はエラーメッセージ文字列です。

エラーコードでは、正の値が警告、負の値がエラー、0 が成功を示します。

例

```
code, msg = api.sqlany_error( conn )
print "Code=#{code} Message=#{msg}¥n"
```

関連情報

sqlany_connect 関数 [530 ページ]

1.17.2.10 sqlany_execute 関数

準備文を実行します。

構文

```
sqlany_execute ( $stmt )
```

パラメータ

$stmt

sqlany_prepare を使用して準備された文。

戻り値

成功した場合はスカラー値 1、失敗した場合は 0 を返します。

備考

sqlany_num_cols を使用して、文が結果セットを返したかどうか確認できます。

例

```
stmt = api.sqlany_prepare(conn, "UPDATE Contacts
     SET Contacts.ID = Contacts.ID + 1000
     WHERE Contacts.ID >= ?" )
rc, param = api.sqlany_describe_bind_param( stmt, 0 )
param.set_value(50)
rc = api.sqlany_bind_param( stmt, 0, param )
rc = api.sqlany_execute( stmt )
```

関連情報

sqlany_prepare 関数 [549 ページ]

1.17.2.11 sqlany_execute_direct 関数

文字列引数によって指定された SQL 文を実行します。

構文

```
sqlany_execute_direct ( $conn, $sql )
```

パラメータ

$conn

sqlany_connect を使用して確立された接続の接続オブジェクト。

$sql

SQL 文字列。SQL 文字列には、? のようなパラメータを含めることはできません。

戻り値

文オブジェクトを返します。失敗した場合は NULL を返します。

備考

文の準備と実行を 1 つの手順で実行する場合にこの関数を使用します。パラメータを持つ SQL 文を実行する際には使用しないでください。

例

```
stmt = api.sqlany_execute_direct( conn, "SELECT * FROM Employees" )
rc = api.sqlany_fetch_next( stmt )
rc, employeeID = api.sqlany_get_column( stmt, 0 )
rc, managerID  = api.sqlany_get_column( stmt, 1 )
rc, surname = api.sqlany_get_column( stmt, 2 )
rc, givenName = api.sqlany_get_column( stmt, 3 )
rc, departmentID = api.sqlany_get_column( stmt, 4 )
print employeeID, ",", managerID, ",",
     surname, ",", givenName, ",", departmentID, "¥n"
```

関連情報

sqlany_fetch_absolute 関数 [537 ページ]
sqlany_fetch_next 関数 [538 ページ]
sqlany_num_cols 関数 [547 ページ]
sqlany_get_column 関数 [542 ページ]

1.17.2.12 sqlany_execute_immediate 関数

指定された SQL 文を、結果セットを返さずにただちに実行します。結果セットを返さない文の場合に使用すると便利です。

構文

```
sqlany_execute_immediate ( $conn, $sql )
```

パラメータ

$conn

sqlany_connect を使用して確立された接続の接続オブジェクト。

$sql

SQL 文字列。SQL 文字列には、? のようなパラメータを含めることはできません。

戻り値

成功した場合はスカラー値 1、失敗した場合は 0 を返します。

例

```
rc = api.sqlany_execute_immediate(conn, "UPDATE Contacts
     SET Contacts.ID = Contacts.ID + 1000
     WHERE Contacts.ID >= 50" )
```

関連情報

sqlany_error 関数 [533 ページ]

1.17.2.13 sqlany_fetch_absolute 関数

結果セット内の現在のローを、指定されたロー番号に移し、そのローのデータをフェッチします。

構文

```
sqlany_fetch_absolute ( $stmt, $row_num )
```

パラメータ

$stmt

sqlany_execute または sqlany_execute_direct によって実行された文オブジェクト。

$row_num

フェッチされるローの番号。最初のローは 1、最後のローは -1。

戻り値

成功した場合はスカラー値 1、失敗した場合は 0 を返します。

例

```
stmt = api.sqlany_execute_direct( conn, "SELECT * FROM Employees" )
# Fetch the second row
rc = api.sqlany_fetch_absolute( stmt, 2 )
rc, employeeID = api.sqlany_get_column( stmt, 0 )
rc, managerID  = api.sqlany_get_column( stmt, 1 )
rc, surname = api.sqlany_get_column( stmt, 2 )
rc, givenName = api.sqlany_get_column( stmt, 3 )
rc, departmentID = api.sqlany_get_column( stmt, 4 )
print employeeID, ",", managerID, ",",
     surname, ",", givenName, ",", departmentID, "¥n"
```

関連情報

sqlany_error 関数 [533 ページ]
sqlany_execute 関数 [534 ページ]
sqlany_execute_direct 関数 [535 ページ]
sqlany_fetch_next 関数 [538 ページ]

1.17.2.14 sqlany_fetch_next 関数

結果セットから次のローを返します。この関数は、ローポインタを進めてから新しいローのデータをフェッチします。

構文

```
sqlany_fetch_next ( $stmt )
```

パラメータ

$stmt

sqlany_execute または sqlany_execute_direct によって実行された文オブジェクト。

戻り値

成功した場合はスカラー値 1、失敗した場合は 0 を返します。

例

```
stmt = api.sqlany_execute_direct( conn, "SELECT * FROM Employees" )
# Fetch the second row
rc = api.sqlany_fetch_next( stmt )
rc, employeeID = api.sqlany_get_column( stmt, 0 )
rc, managerID  = api.sqlany_get_column( stmt, 1 )
rc, surname = api.sqlany_get_column( stmt, 2 )
rc, givenName = api.sqlany_get_column( stmt, 3 )
rc, departmentID = api.sqlany_get_column( stmt, 4 )
print employeeID, ",", managerID, ",",
     surname, ",", givenName, ",", departmentID, "¥n"
```

関連情報

sqlany_error 関数 [533 ページ]
sqlany_execute 関数 [534 ページ]
sqlany_execute_direct 関数 [535 ページ]
sqlany_fetch_absolute 関数 [537 ページ]

1.17.2.15 sqlany_fini 関数

API によって割り付けられたリソースを解放します。

構文

```
sqlany_fini ( )
```

戻り値

NULL を返します。

例

```
# Disconnect from the database
api.sqlany_disconnect( conn )
# Free the connection resources
api.sqlany_free_connection( conn )
# Free resources the api object uses
api.sqlany_fini()
# Close the interface
SQLAnywhere::API.sqlany_finalize_interface( api )
```

関連情報

sqlany_init 関数 [545 ページ]

1.17.2.16 sqlany_free_connection 関数

接続オブジェクトに関連付けられているリソースを解放します。

構文

```
sqlany_free_connection ( $conn )
```

パラメータ

$conn

sqlany_new_connection によって作成された接続オブジェクト。

戻り値

NULL を返します。

例

```
# Disconnect from the database
api.sqlany_disconnect( conn )
# Free the connection resources
api.sqlany_free_connection( conn )
# Free resources the api object uses
api.sqlany_fini()
# Close the interface
SQLAnywhere::API.sqlany_finalize_interface( api )
```

関連情報

sqlany_new_connection 関数 [546 ページ]

1.17.2.17 sqlany_free_stmt 関数

文オブジェクトに関連付けられているリソースを解放します。

構文

```
sqlany_free_stmt ( $stmt )
```

パラメータ

$stmt

sqlany_prepare または sqlany_execute_direct の実行によって返された文オブジェクト。

戻り値

NULL を返します。

例

```
stmt = api.sqlany_prepare(conn, "UPDATE Contacts
     SET Contacts.ID = Contacts.ID + 1000
     WHERE Contacts.ID >= ?" )
rc, param = api.sqlany_describe_bind_param( stmt, 0 )
param.set_value(50)
rc = api.sqlany_bind_param( stmt, 0, param )
rc = api.sqlany_execute( stmt )
rc = api.sqlany_free_stmt( stmt )
```

関連情報

sqlany_prepare 関数 [549 ページ]
sqlany_execute_direct 関数 [535 ページ]

1.17.2.18 sqlany_get_bind_param_info 関数

sqlany_bind_param を使用してバインドされたパラメータに関する情報を取得します。

構文

```
sqlany_get_bind_param_info ( $stmt, $index )
```

パラメータ

$stmt

sqlany_prepare を使用して準備された文。

$index

パラメータのインデックス。数値は、0 ～ sqlany_num_params() - 1 の間である必要があります。

戻り値

2 要素の配列を返します。最初の要素は、成功した場合は 1、失敗した場合は 0 です。2 番目の要素は記述されたパラメータです。

例

```
# Get information on first parameter (0)
```

```
rc, param_info = api.sqlany_get_bind_param_info( stmt, 0 )
print "Param_info direction = ", param_info.get_direction(), "¥n"
print "Param_info output = ", param_info.get_output(), "¥n"
```

関連情報

sqlany_bind_param 関数 [527 ページ]
sqlany_describe_bind_param 関数 [531 ページ]
sqlany_prepare 関数 [549 ページ]

1.17.2.19 sqlany_get_column 関数

指定されたカラムについてフェッチされた値を返します。

構文

```
sqlany_get_column ( $stmt, $col_index )
```

パラメータ

$stmt

sqlany_execute または sqlany_execute_direct によって実行された文オブジェクト。

$col_index

取り出すカラムの数。カラムの数は、0 ～ sqlany_num_cols() - 1 の間です。

戻り値

2 要素の配列を返します。最初の要素は、成功した場合は 1、失敗した場合は 0 です。2 番目の要素はカラム値です。

例

```
stmt = api.sqlany_execute_direct( conn, "SELECT * FROM Employees" )
# Fetch the second row
rc = api.sqlany_fetch_next( stmt )
rc, employeeID = api.sqlany_get_column( stmt, 0 )
rc, managerID  = api.sqlany_get_column( stmt, 1 )
rc, surname = api.sqlany_get_column( stmt, 2 )
rc, givenName = api.sqlany_get_column( stmt, 3 )
rc, departmentID = api.sqlany_get_column( stmt, 4 )
```

```
print employeeID, ",", managerID, ",",
      surname, ",", givenName, ",", departmentID, "¥n"
```

関連情報

sqlany_execute 関数 [534 ページ]
sqlany_execute_direct 関数 [535 ページ]
sqlany_fetch_absolute 関数 [537 ページ]
sqlany_fetch_next 関数 [538 ページ]

1.17.2.20 sqlany_get_column_info 関数

指定された結果セットのカラムに対するカラム情報を取得します。

構文

```
sqlany_get_column_info ( $stmt, $col_index )
```

パラメータ

$stmt

sqlany_execute または sqlany_execute_direct によって実行された文オブジェクト。

$col_index

カラムの数は、0 ～ sqlany_num_cols() - 1 の間です。

戻り値

結果セット内のカラムに関する情報を格納した 9 要素の配列を返します。最初の要素は、成功した場合は 1、失敗した場合は 0 です。配列の各要素を次の表に示します。

要素番号	タイプ	説明
0	Integer	成功した場合は 1、失敗した場合は 0。
1	Integer	カラムのインデックス (0 ～ sqlany_num_cols() - 1)。
2	String	カラム名。

要素番号	タイプ	説明
3	Integer	カラム型。
4	Integer	カラムのネイティブ型。
5	Integer	カラム精度 (数値型の場合)。
6	Integer	カラムの位取り (数値型の場合)。
7	Integer	カラムサイズ。
8	Integer	カラムが NULL 入力可かどうか (1 = NULL 入力可、0 = NULL 入力不可)。

例

```
# Get column info for first column (0)
rc, col_num, col_name, col_type, col_native_type, col_precision, col_scale,
    col_size, col_nullable = api.sqlany_get_column_info( stmt, 0 )
```

関連情報

カラムの型 [552 ページ]
ネイティブのカラム型 [553 ページ]
sqlany_execute 関数 [534 ページ]
sqlany_execute_direct 関数 [535 ページ]
sqlany_prepare 関数 [549 ページ]

1.17.2.21 sqlany_get_next_result 関数

複数の結果セットクエリ内の次の結果セットに進みます。

構文

```
sqlany_get_next_result ( $stmt )
```

パラメータ

$stmt

sqlany_execute または sqlany_execute_direct によって実行された文オブジェクト。

戻り値

成功した場合はスカラー値 1、失敗した場合は 0 を返します。

例

```
stmt = api.sqlany_prepare(conn, "call two_results()" )
rc = api.sqlany_execute( stmt )
# Fetch from first result set
rc = api.sqlany_fetch_absolute( stmt, 3 )
# Go to next result set
rc = api.sqlany_get_next_result( stmt )
# Fetch from second result set
rc = api.sqlany_fetch_absolute( stmt, 2 )
```

関連情報

sqlany_execute 関数 [534 ページ]
sqlany_execute_direct 関数 [535 ページ]

1.17.2.22 sqlany_init 関数

インタフェースを初期化します。

構文

```
sqlany_init ( )
```

戻り値

2 要素の配列を返します。最初の要素は、成功した場合は 1、失敗した場合は 0 です。2 番目の要素は Ruby インタフェースバージョンです。

例

```
# Load the SQLAnywhere gem
begin
    require 'rubygems'
    gem 'sqlanywhere'
    unless defined? SQLAnywhere
        require 'sqlanywhere'
    end
end
# Create an interface
```

```
api = SQLAnywhere::SQLAnywhereInterface.new()
# Initialize the interface (loads the DLL/SO)
SQLAnywhere::API.sqlany_initialize_interface( api )
# Initialize our api object
api.sqlany_init()
```

関連情報

sqlany_fini 関数 [539 ページ]

1.17.2.23 sqlany_new_connection 関数

接続オブジェクトを作成します。

構文

```
sqlany_new_connection ()
```

戻り値

接続オブジェクトのスカラー値を返します。

備考

データベース接続が確立される前に接続オブジェクトが作成されている必要があります。接続オブジェクトからエラーが取得される場合があります。各接続で一度に処理できる要求は 1 つだけです。

例

```
require 'sqlanywhere'
api = SQLAnywhere::SQLAnywhereInterface.new()
SQLAnywhere::API.sqlany_initialize_interface( api )
api.sqlany_init()
# Create a connection
conn = api.sqlany_new_connection()
puts "Enter user ID"
myuid = STDIN.gets.chomp
puts "Enter password"
mypwd = STDIN.gets.chomp
# Establish a connection
status = api.sqlany_connect( conn, "UID="+myuid+";PWD="+mypwd )
print "Connection status = #{status}¥n"
```

関連情報

sqlany_connect 関数 [530 ページ]
sqlany_disconnect 関数 [532 ページ]

1.17.2.24 sqlany_num_cols 関数

結果セット内のカラム数を返します。

構文

```
sqlany_num_cols ( $stmt )
```

パラメータ

$stmt

sqlany_execute または sqlany_execute_direct によって実行された文オブジェクト。

戻り値

結果セット内のカラム数のスカラー値を返します。失敗した場合は -1 を返します。

例

```
stmt = api.sqlany_execute_direct( conn, "SELECT * FROM Employees" )
# Get number of result set columns
num_cols = api.sqlany_num_cols( stmt )
```

関連情報

sqlany_execute 関数 [534 ページ]
sqlany_execute_direct 関数 [535 ページ]
sqlany_prepare 関数 [549 ページ]

1.17.2.25 sqlany_num_params 関数

準備文で必要とされるパラメータ数を返します。

構文

```
sqlany_num_params ( $stmt )
```

パラメータ

$stmt

sqlany_prepare の実行によって返された文オブジェクト。

戻り値

準備文内のパラメータ数のスカラー値を返します。失敗した場合は -1 を返します。

例

```
stmt = api.sqlany_prepare(conn, "UPDATE Contacts
      SET Contacts.ID = Contacts.ID + 1000
      WHERE Contacts.ID >= ?" )
num_params = api.sqlany_num_params( stmt )
```

関連情報

sqlany_prepare 関数 [549 ページ]

1.17.2.26 sqlany_num_rows 関数

結果セット内のロー数を返します。

構文

```
sqlany_num_rows ( $stmt )
```

パラメータ

$stmt

sqlany_execute または sqlany_execute_direct によって実行された文オブジェクト。

戻り値

結果セット内のロー数のスカラー値を返します。ロー数が推定値の場合は負の値を返します。また、その推定値が、返された整数の絶対値となります。ロー数が正確な値の場合は正の値を返します。

備考

デフォルトでは、この関数は推定値のみを返します。正確なロー数が返されるようにするには、接続の ROW_COUNTS オプションを設定します。

複数の結果セットを返す文の場合、最初の結果セット内のロー数だけが返されます。sqlany_get_next_result を使用して次の結果セットに進んでも、sqlany_num_rows によって返されるのは最初の結果セット内のロー数のみです。

例

```
stmt = api.sqlany_execute_direct( conn, "SELECT * FROM Employees" )
# Get number of rows in result set
num_rows = api.sqlany_num_rows( stmt )
```

関連情報

sqlany_execute 関数 [534 ページ]
sqlany_execute_direct 関数 [535 ページ]

1.17.2.27 sqlany_prepare 関数

指定された SQL 文字列を準備します。

構文

```
sqlany_prepare ( $conn, $sql )
```

パラメータ

$conn

sqlany_connect を使用して確立された接続の接続オブジェクト。

$sql

準備される SQL 文。

戻り値

文オブジェクトのスカラー値を返します。失敗した場合は NULL を返します。

備考

文オブジェクトに関連付けられた文は sqlany_execute によって実行されます。sqlany_free_stmt を使用して、文オブジェクトに関連付けられたリソースを解放できます。

例

```
stmt = api.sqlany_prepare(conn, "UPDATE Contacts
     SET Contacts.ID = Contacts.ID + 1000
     WHERE Contacts.ID >= ?" )
rc, param = api.sqlany_describe_bind_param( stmt, 0 )
param.set_value(50)
rc = api.sqlany_bind_param( stmt, 0, param )
rc = api.sqlany_execute( stmt )
```

関連情報

sqlany_execute 関数 [534 ページ]
sqlany_free_stmt 関数 [540 ページ]
sqlany_num_params 関数 [548 ページ]
sqlany_describe_bind_param 関数 [531 ページ]
sqlany_bind_param 関数 [527 ページ]

1.17.2.28 sqlany_rollback 関数

現在のトランザクションをロールバックします。

構文

```
sqlany_rollback ( $conn )
```

パラメータ

$conn

ロールバック操作が実行される接続オブジェクト。

戻り値

成功した場合はスカラー値 1、失敗した場合は 0 を返します。

例

```
rc = api.sqlany_rollback( conn )
```

関連情報

sqlany_commit 関数 [529 ページ]

1.17.2.29 sqlany_sqlstate 関数

現在の SQLSTATE を取得します。

構文

```
sqlany_sqlstate ( $conn )
```

パラメータ

$conn

sqlany_new_connection から返された接続オブジェクト。

戻り値

現在の SQLSTATE を表す 5 文字のスカラー値を返します。

例

```
sql_state = api.sqlany_sqlstate( conn )
```

関連情報

sqlany_error 関数 [533 ページ]

1.17.2.30 カラムの型

次の Ruby クラスで、一部の Ruby 拡張関数によって返されるカラムの型が定義されています。

```
class Types
    A_INVALID_TYPE = 0
    A_BINARY       = 1
    A_STRING       = 2
    A_DOUBLE       = 3
    A_VAL64        = 4
    A_UVAL64       = 5
    A_VAL32        = 6
    A_UVAL32       = 7
    A_VAL16        = 8
    A_UVAL16       = 9
    A_VAL8         = 10
    A_UVAL8        = 11
end
```

1.17.2.31 ネイティブのカラム型

次の表に、一部の Ruby 拡張関数によって返されるネイティブのカラム型を示します。

ネイティブ型の値	ネイティブ型
384	DT_DATE
388	DT_TIME
390	DT_TIMESTAMP_STRUCT
392	DT_TIMESTAMP
448	DT_VARCHAR
452	DT_FIXCHAR
456	DT_LONGVARCHAR
460	DT_STRING
480	DT_DOUBLE
482	DT_FLOAT
484	DT_DECIMAL
496	DT_INT
500	DT_SMALLINT
524	DT_BINARY
528	DT_LONGBINARY
600	DT_VARIABLE
604	DT_TINYINT
608	DT_BIGINT
612	DT_UNSINT
616	DT_UNSSMALLINT
620	DT_UNSBIGINT
624	DT_BIT
628	DT_NSTRING
632	DT_NFIXCHAR
636	DT_NVARCHAR
640	DT_LONGNVARCHAR

1.18 SAP Open Client のサポート

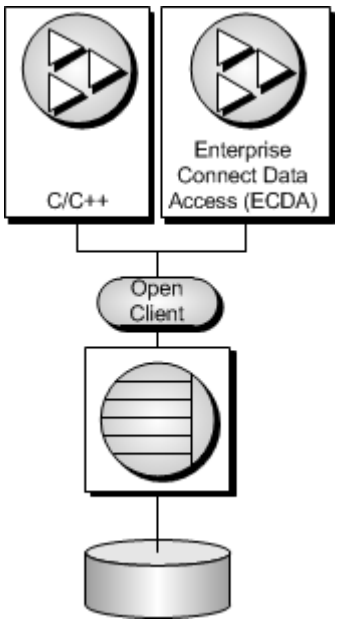
SAP Open Client は、カスタマアプリケーション、サードパーティ製品、その他の SAP 製品に対して、Open Server との通信に必要なインタフェースを提供します。

C または C++ で開発したアプリケーションを Open Client API を使用して Open Server に接続できます。

SQL Anywhere データベースサーバは、Open Client アプリケーションに対する Open Server として動作できます。この機能により、SAP Open Client アプリケーションは SQL Anywhere データベースにネイティブに接続できます。

どのようなときに **Open Client** を使用するか

SAP Adaptive Server Enterprise との互換性が必要なとき、または Open Client インタフェースをサポートする他の SAP 製品を使用しているときに、Open Client インタフェースの使用を検討します。



Enterprise Connect Data Access のサポート

データベースサーバは Open Server として動作できるため、SAP Enterprise Connect Data Access (ECDA) がサポートされます。

Enterprise Connect Data Access (ECDA) は、データの内容やデータの所在がわからなくても複数のデータソースにアクセスすることにより、企業内でデータを統合して表示することができます。また、ECDA は企業全体にわたってデータの異機種間ジョインを実行し、SAP Adaptive Server Enterprise、SAP IQ、SAP SQL Anywhere、IBM DB2、IBM VSAM、Oracle などのターゲットについて、プラットフォームを問わないテーブルジョインを可能にします。

SAP Open Client と Open Server

単に、既存の Open Client アプリケーションをデータベースサーバとともに使用する場合は、SAP Open Client と Open Server の詳細を知る必要はありません。しかし、これらのコンポーネントがどのように互いに適合するかを理解すれば、データベースの設定やアプリケーションの設定に役立ちます。

このデータベースサーバと、SAP Adaptive Server Enterprise などの SAP Database Management Server ファミリの他のメンバーは、**Open Server** として動作します。つまり、SAP から入手可能な **SAP Open Client** ライブラリを使用して、クライアントアプリケーションを開発できます。Open Client には、Client Library (CT-Library) インタフェース、旧式の DB-Library インタフェースの両方が含まれています。

Tabular Data Stream (TDS)

SAP Open Client と Open Server は、**Tabular Data Stream** (TDS) と呼ばれるアプリケーションプロトコルを使用して情報を交換します。SAP Open Client ライブラリを使用して構築されたすべてのアプリケーションは、TDS アプリケーションでもあります。これは、Open Client ライブラリが TDS インタフェースを使用するためです。ただし、(jConnect などの) 一部のアプリケーションは、Open Client ライブラリを使用しませんが、TDS アプリケーションです。これらのアプリケーションは、TDS プロトコルを使用して直接通信します。

数多くの Open Server が SAP Open Server ライブラリを使用して TDS へのインタフェースを処理していますが、独自に TDS への直接インタフェースを持つサーバもあります。SAP Adaptive Server Enterprise、SAP IQ、および SAP SQL Anywhere は、TDS をネイティブにサポートする Open Server の例です。

プログラミングインタフェースとアプリケーションプロトコル

データベースサーバによって、2 つのクライアントアプリケーションプロトコルがサポートされます。

SAP Open Client アプリケーション、および Enterprise Connect Data Access (ECDA) などの他の SAP アプリケーションは、TDS を使用します。

.NET、ODBC、JDBC、Embedded SQL、OLE DB、および他のアプリケーションインタフェースは、**Command Sequence** (CmdSeq) と呼ばれる別のアプリケーションプロトコルを使用します。

TDS による TCP/IP の使用

TDS や CmdSeq などのアプリケーションプロトコルは、ネットワークトラフィックを処理する下位レベルの通信プロトコルの一番上に位置します。TDS と CmdSeq は TCP/IP ネットワークプロトコルを使用して、コンピュータ間の通信を行います。また、SQL Anywhere は同一コンピュータでの通信用に設計された共有メモリプロトコルもサポートします。

このセクションの内容:

Open Client アーキテクチャ [556 ページ]

下記の情報は、SQL Anywhere に固有の SAP Open Client 機能の説明です。SAP Open Client アプリケーション開発の詳細なマニュアルは、SAP から入手できます。

Open Client アプリケーション作成に必要なもの [557 ページ]
Open Client アプリケーションを実行するためには、アプリケーションを実行するコンピュータに SAP Open Client コンポーネントをインストールして構成する必要があります。

Open Client データ型マッピング [557 ページ]
SAP Open Client は独自の内部データ型を持っており、そのデータ型は SQL Anywhere で使用されるものと細部が多少異なります。

Open Client アプリケーションでの SQL [559 ページ]
ここでは、Open Client アプリケーションでの SQL の使用について簡単に説明します。SQL Anywhere 固有の問題に焦点を当てます。

SQL Anywhere における Open Client の既知の制限 [561 ページ]
Open Client インタフェースを使用すると、SQL Anywhere データベースを、Adaptive Server Enterprise データベースとほとんど同じ方法で使用できます。

SQL Anywhere を Open Server として使用するためのシステム稼働条件 [562 ページ]
SQL Anywhere を Open Server として使用するためには、クライアント側とサーバ側でそれぞれ稼働条件があります。

Open Server としてのデータベースサーバの設定 [563 ページ]
データベースサーバが Open Server として使用されているときは、TCP/IP 通信プロトコルが必要です。このプロトコルは、ネットワークサーバに対してデフォルトで起動されます。

1.18.1 Open Client アーキテクチャ

下記の情報は、SQL Anywhere に固有の SAP Open Client 機能の説明です。SAP Open Client アプリケーション開発の詳細なマニュアルは、SAP から入手できます。

SAP Open Client は、プログラミングインタフェースとネットワークサービスの 2 つのコンポーネントから構成されています。

DB-Library と Client Library

SAP Open Client には、クライアントアプリケーションを記述するための主要なプログラミングインタフェースが 2 つ用意されています。DB-Library と Client-Library です。

Open Client DB-Library は、以前の Open Client アプリケーションをサポートする、Client-Library とはまったく別のプログラミングインタフェースです。DB-Library の詳細については、SAP Open Client 製品に付属する *DB-Library/C* リファレンスマニュアル (ドキュメント ID: DC32600-01-1570-01) を参照してください。

Client-Library プログラムも CS-Library に依存しています。CS-Library は、Client-Library アプリケーションと Server-Library アプリケーションの両方で使用されるルーチンを提供します。Client-Library アプリケーションは、Bulk-Library のルーチンを使用して高速データ転送を行うこともできます。Client-Library の詳細については、SAP Open Client 製品に付属する *Client-Library/C* リファレンスマニュアル (ドキュメント ID: DC32840-01-1570-02) を参照してください。

CS-Library と Bulk-Library はどちらも Open Client に含まれていますが、別々に使用できます。

ネットワークサービス

Open Client ネットワークサービスは、TCP/IP や DECnet などの特定のネットワークプロトコルをサポートする Net-Library を含みます。Net-Library インタフェースはアプリケーション開発者からは見えません。ただしプラットフォームによっては、アプリケーションがシステムネットワーク構成ごとに別の Net-Library ドライバを必要とする場合もあります。Net-Library ドライバの指定は、ホストプラットフォームにより、システムの Adaptive Server Enterprise 設定で行うか、またはプログラムをコンパイルしてリンクするときに行います。

ドライバ設定の詳細については、*Open Client and Open Server - Configuration Guide for Windows* (ドキュメント ID: DC35830-01-1570-02) または *Open Client and Open Server - Configuration Guide for UNIX* (ドキュメント ID: DC35831-01-1570-02) を参照してください。

Client-Library プログラムの作成方法については、*Client-Library/C* プログラマーズガイド (ドキュメント ID: DC35570-01-1570-01) を参照してください。

1.18.2 Open Client アプリケーション作成に必要なもの

Open Client アプリケーションを実行するためには、アプリケーションを実行するコンピュータに SAP Open Client コンポーネントをインストールして構成する必要があります。

これらのコンポーネントは、他の SAP 製品の一部としてインストールするか、ライセンス契約の条項に従って、SQL Anywhere とともに、これらのライブラリをオプションでインストールできます。

データベースサーバを実行しているコンピュータでは、Open Client アプリケーションは Open Client コンポーネントを一切必要としません。

Open Client アプリケーションを作成するには、SAP から入手可能な Open Client の開発バージョンが必要です。

デフォルトでは、SQL Anywhere データベースは大文字と小文字を区別しないように、Adaptive Server Enterprise データベースでは区別するように作成されます。

1.18.3 Open Client データ型マッピング

SAP Open Client は独自の内部データ型を持っており、そのデータ型は SQL Anywhere で使用されるものと細部が多少異なります。

このため、SQL Anywhere は Open Client アプリケーションで使用されるデータ型と SQL Anywhere で使用されるデータ型を内部的にマッピングします。

Open Client アプリケーションを作成するには、Open Client の開発バージョンが必要です。Open Client アプリケーションを使うには、そのアプリケーションが動作するコンピュータに、Open Client ランタイムをインストールし構成しておく必要があります。

SQL Anywhere サーバは Open Client アプリケーションをサポートするために、外部通信のランタイムを一切必要としません。

Open Client の各データ型は、同等の SQL Anywhere のデータ型にマッピングされます。Open Client のデータ型は、すべてサポートされます。

このセクションの内容:

Open Client データ型マッピングの範囲制限 [558 ページ]
データ型によっては、SQL Anywhere と Open Client で値範囲が異なります。このような場合には、データを検索または挿入するときにオーバーフローエラーが発生することがあります。

1.18.3.1 Open Client データ型マッピングの範囲制限

データ型によっては、SQL Anywhere と Open Client で値範囲が異なります。このような場合には、データを検索または挿入するときにオーバーフローエラーが発生することがあります。

次の表にまとめた Open Client アプリケーションのデータ型は、SQL Anywhere データ型にマッピングできますが、使用可能な値の範囲に制限があります。

通常、Open Client データ型からマッピングされる SQL Anywhere データ型のほうが使用可能な値の範囲が大きくなっています。その結果、Open Client アプリケーションでは大きすぎてフェッチできない値をデータベースに格納することができます。

データ型	Open Client の最小値	Open Client の最大値	SQL Anywhere の最小値	SQL Anywhere の最大値
MONEY	-922 377 203 685 477.5808	922 377 203 685 477.5807	-999 999 999 999 999.9999	999 999 999 999 999.9999
SMALLMONEY	-214 748.3648	214 748.3647	-999 999.9999	-999 999.9999
DATETIME [1]	January 1, 1753	December 31, 9999	January 1, 0001	December 31, 9999
SMALLDATETIME	January 1, 1900	June 6, 2079	January 1, 0001	December 31, 9999

[1] OpenClient 15.5 より前のバージョン用。これ以降のバージョンでは、0001-01-01 ～ 9999-12-31 の範囲内のすべての日付がサポートされています。

例

たとえば、Open Client の MONEY および SMALLMONEY データ型は、基本となる SQL Anywhere 実装の全数値範囲を超えることはありません。したがって、Open Client のデータ型 MONEY の境界を超える値をカラムに設定できます。クライアントがデータベースからそうした違反値をフェッチすると、エラーになります。

タイムスタンプ

クライアントが Open Client 15.1 以前を使用している場合、データベースに挿入した TIMESTAMP 値またはデータベースから取り出した TIMESTAMP 値は、日付部分が January 1, 1753 以降に制限され、時刻部分が秒の精度の 300 分の 1 に制限されています。ただし、クライアントが Open Client 15.5 以降を使用している場合は、TIMESTAMP 値に制限はありません。

1.18.4 Open Client アプリケーションでの SQL

ここでは、Open Client アプリケーションでの SQL の使用について簡単に説明します。SQL Anywhere 固有の問題に焦点を当てます。

このセクションの内容:

Open Client SQL 文の実行 [559 ページ]
SQL 文を Client Library 関数呼び出しに入れてデータベースサーバに送ります。

Open Client の準備文 [559 ページ]
ct_dynamic 関数を使用して準備文を管理します。

Open Client カーソルの管理 [560 ページ]
ct_cursor 関数を使用してカーソルを管理します。

Open Client の結果セット [561 ページ]
Open Client が結果セットを処理する方法は、他の SQL Anywhere インタフェースの方法とは異なります。

関連情報

SQL を使用したアプリケーション開発 [11 ページ]
Open Server

1.18.4.1 Open Client SQL 文の実行

SQL 文を Client Library 関数呼び出しに入れてデータベースサーバに送ります。

たとえば、次の一組の呼び出しは DELETE 文を実行します。

```
ret = ct_command(cmd, CS_LANG_CMD,
                  "DELETE FROM Employees
                   WHERE EmployeeID=105"
                 CS_NULLTERM,
                 CS_UNUSED);
ret = ct_send(cmd);
```

1.18.4.2 Open Client の準備文

ct_dynamic 関数を使用して準備文を管理します。

この関数には、実行するアクションを `type` パラメータで指定します。

Open Client で準備文を使用するには、次のタスクを実行します。

1. CS_PREPARE を type パラメータに指定した ct_dynamic 関数を使用して文を準備します。
2. ct_param を使用して文のパラメータを設定します。
3. CS_EXECUTE を type パラメータに指定した ct_dynamic を使用して文を実行します。
4. CS_DEALLOC を type パラメータに指定した ct_dynamic を使用して、文に関連付けられたリソースを解放します。

Open Client で準備文を使用する方法の詳細については、Open Client のマニュアルを参照してください。

1.18.4.3 Open Client カーソルの管理

ct_cursor 関数を使用してカーソルを管理します。

この関数には、実行するアクションを type パラメータで指定します。

サポートするカーソルタイプ

SQL Anywhere でサポートされるすべてのタイプのカーソルを、Open Client インタフェースを通じて使用できるわけではありません。スクロールカーソル、動的スクロールカーソル、または insensitive カーソルは、Open Client を通じて使用できません。

ユニークさと更新可能であることが、カーソルの 2 つの特性です。カーソルはユニーク (各ローが、アプリケーションに使用されるかどうかにかかわりなく、プライマリキーまたはユニーク情報を持つ) でもユニークでなくても構いません。カーソルは読み込み専用にも更新可能にもできます。カーソルが更新可能でユニークでない場合は、CS_CURSOR_ROWS の設定にかかわりなく、ローのプリフェッチが行われないので、パフォーマンスが低下する可能性があります。

カーソルを使用する手順

Embedded SQL などの他のインタフェースとは違って、Open Client はカーソルを、文字列として表現された SQL 文に対応させます。Embedded SQL の場合は、まず文を作成し、次にステートメントハンドルを使用してカーソルを宣言します。

Open Client でカーソルを使用するには、次のタスクを実行します。

1. Open Client のカーソルを宣言するには、CS_CURSOR_DECLARE を type パラメータに指定した ct_cursor を使用します。
2. カーソルを宣言したら、CS_CURSOR_ROWS を type パラメータに指定した ct_cursor を使用して、サーバからローをフェッチするたびにクライアント側にプリフェッチするローの数を制御できます。
   プリフェッチしたローをクライアント側に格納すると、サーバに対する呼び出し数を減らし、全体的なスループットとターンアラウンドタイムを改善できます。プリフェッチしたローは、すぐにアプリケーションに渡されるわけではなく、いつでも使用できるようにクライアント側のバッファに格納されます。
   prefetch データベースオプションの設定によって、他のインタフェースに対するローのプリフェッチを制御します。この設定は、Open Client 接続では無視されます。CS_CURSOR_ROWS 設定は、ユニークでない更新可能なカーソルについては無視されます。
3. Open Client のカーソルを開くには、CS_CURSOR_OPEN を type パラメータに指定した ct_cursor を使用します。

4. 各ローをアプリケーションにフェッチするには、ct_fetch を使用します。
5. カーソルを閉じるには、CS_CURSOR_CLOSE を指定した ct_cursor を使用します。
6. Open Client では、カーソルに対応するリソースの割り付けを解除する必要もあります。CS_CURSOR_DEALLOC を指定した ct_cursor を使用してください。CS_CURSOR_CLOSE とともに補足パラメータ CS_DEALLOC を指定して、これらの処理を 1 ステップで実行することもできます。

このセクションの内容:

カーソルによる Open Client のローの変更 [561 ページ]
Open Client では、カーソルが 1 つのテーブル用であればカーソル内でローを削除または更新できます。テーブルを更新するパーミッションを持っている必要があり、そのカーソルは更新のマークが付けられている必要があります。

1.18.4.3.1 カーソルによる Open Client のローの変更

Open Client では、カーソルが 1 つのテーブル用であればカーソル内でローを削除または更新できます。テーブルを更新するパーミッションを持っている必要があり、そのカーソルは更新のマークが付けられている必要があります。

フェッチを実行する代わりに、CS_CURSOR_DELETE または CS_CURSOR_UPDATE を指定した ct_cursor を使用してカーソルのローを削除または更新できます。

Open Client アプリケーションではカーソルからのローの挿入はできません。

1.18.4.4 Open Client の結果セット

Open Client が結果セットを処理する方法は、他の SQL Anywhere インタフェースの方法とは異なります。

Embedded SQL と ODBC では、結果を受け取る変数の適切な数と型を設定するために、クエリまたはストアドプロシージャを記述します。記述は文自体を対象に行います。

Open Client では、文を記述する必要はありません。代わりに、サーバから戻される各ローは内容に関する記述を持つことができます。ct_command と ct_send を使用して文を実行した場合、クエリに戻されたローのあらゆる処理に ct_results 関数を使用できます。

このようなロー単位の結果セット処理方式を使用したくない場合は、ct_dynamic を使用して SQL 文を作成し、ct_describe を使用してその結果セットを記述できます。この方式は、他のインタフェースにおける SQL 文の記述方式と密接に対応しています。

1.18.5 SQL Anywhere における Open Client の既知の制限

Open Client インタフェースを使用すると、SQL Anywhere データベースを、Adaptive Server Enterprise データベースとほとんど同じ方法で使用できます。

ただし、次に示すような制限があります。

- SQL Anywhere は Adaptive Server Enterprise のコミットサービスをサポートしません。
- クライアント/サーバ接続の機能によって、その接続で許可されているクライアント要求とサーバ応答のタイプが決まります。次の機能はサポートされていません。
  - CS_CSR_ABS
  - CS_CSR_FIRST
  - CS_CSR_LAST
  - CS_CSR_PREV
  - CS_CSR_REL
  - CS_DATA_BOUNDARY
  - CS_DATA_SENSITIVITY
  - CS_OPT_FORMATONLY
  - CS_PROTO_DYNPROC
  - CS_REG_NOTIF
  - CS_REQ_BCP
- SSL などのセキュリティオプションはサポートされていません。ただし、パスワードの暗号化はサポートされています。
- Open Client アプリケーションは TCP/IP を使用してデータベースサーバに接続できます。
  機能の詳細については、*Open Server Server-Library C* リファレンスマニュアルを参照してください。
- CS_DATAFMT を CS_DESCRIBE_INPUT とともに使用すると、パラメータが指定された変数を入力データとしてデータベースサーバに送信した場合、カラムのデータ型は返されません。

1.18.6 SQL Anywhere を Open Server として使用するためのシステム稼働条件

SQL Anywhere を Open Server として使用するためには、クライアント側とサーバ側でそれぞれ稼働条件があります。

サーバ側の稼働条件

SQL Anywhere を Open Server として使用するには、サーバ側に次の要素が必要です。

SQL Anywhere サーバコンポーネント

ネットワークを介して Open Server にアクセスする場合は、ネットワークサーバ (*dbsrv17*.exe) を使用します。パーソナルサーバ (*dbeng17*.exe) を Open Server として使用できるのは、同一コンピュータからの接続に限られます。

TCP/IP

ネットワーク接続をしていない場合でも、SQL Anywhere を Open Server として使用するには、TCP/IP プロトコルスタックが必要です。

クライアント側の稼働条件

クライアントアプリケーションを使用して SQL Anywhere を含む Open Server に接続するには、次の要素が必要です。

SAP Open Client コンポーネント

アプリケーションが SAP Open Client を使用する場合、TDS 経由でアプリケーションが通信するために必要なネットワークライブラリは、SAP Open Client ライブラリによって提供されます。

jConnect

アプリケーションが JDBC を使用する場合は、jConnect と Java ランタイム環境が必要です。

DSEdit

サーバ名を Open Client アプリケーションから使用できるようにするには、ディレクトリサービスエディタ DSEdit が必要です。UNIX プラットフォームでは、このユーティリティは sybinit と呼ばれます。

データベースサーバは、関連するデータベースオプションを、Open Client および jConnect アプリケーションと互換性のある値に自動的に設定します。これらのオプションの設定は、その接続中だけの一時的なものです。クライアントアプリケーションはこれらのオプションをいつでも独自に設定して変更できます。

関連情報

jConnect for JDBC

1.18.7 Open Server としてのデータベースサーバの設定

データベースサーバが Open Server として使用されているときは、TCP/IP 通信プロトコルが必要です。このプロトコルは、ネットワークサーバに対してデフォルトで起動されます。

パーソナルデータベースサーバは TCP/IP プロトコルをサポートするため、このサーバを、同一コンピュータ上の通信用 Open Server として使用できます。パーソナルサーバを起動するときは、プロトコルを要求する必要があります。たとえば、次のコマンドは TCP/IP 通信プロトコルを開始します。

```
dbeng17 -x tcpip -n myserver c:¥mydata.db
```

データベースサーバは、TDS 経由で Open Client アプリケーションにサービスを提供すると同時に、TCP/IP プロトコルを通じて他のアプリケーションにもサービスを提供できます。

SAP Open Client の設定

データベースサーバに接続するには、データベースサーバを実行しているコンピュータ名と、そのサーバが使用している TCP/IP ポートをクライアントコンピュータ側の interfaces ファイルに指定します。

1.19 OData のサポート

OData (Open Data Protocol) は、RESTful HTTP を介したデータサービスを有効にします。URI (Uniform Resource Identifiers) を通して操作を行い、情報にアクセスして情報を変更できるようになります。

SQL Anywhere を OData サーバとして設定する前に、OData プロトコルの概念について理解しておいてください。

このセクションの内容:

OData サーバのアーキテクチャ [565 ページ]
OData サーバは、OData プロデューサと HTTP サーバーで構成されます。

OData プロトコルの制限事項 [567 ページ]
OData プロデューサは、OData プロトコルバージョン 2 の仕様に準拠していますが、いくつかの制限と例外があります。

OData サーバのセキュリティの考慮事項 [568 ページ]
セキュリティ対策は、OData サーバを設定する前に考慮する必要があります。

OData サーバの設定方法 [570 ページ]
OData サーバの配備と設定を行います。

OData サーバの例 [572 ページ]
OData サーバのサンプルは、*%SQLANYSAMP17%*`¥SQLAnywhere¥` ディレクトリで使用できます。サンプルはデータベースサーバを起動し、OData クライアントを設定し、これらの間で要求と応答を送信する方法を説明します。

繰返可能要求の設定方法 [572 ページ]
繰返可能要求を使用すると、要求が OData プロデューサに送信されたかどうかがクライアントにはわからない場合でも、データベースの一貫性を損なうことや、意図しない副作用を発生させることなく、OData クライアントはデータ変更要求 (UPDATE や DELETE など) を再送信できます。

クロスサイトリクエストフォージェリ攻撃からの保護方法 [574 ページ]
CSRF トークンにより、OData プロデューサがクロスサイトリクエストフォージェリ攻撃から保護されます。

OData プロデューサのサービスモデルを作成する方法 [575 ページ]
オプションの OData Service Definition Language (OSDL) モデルファイルは、OData プロデューササービスモデルを指定して、特定のテーブル、ビュー、ストアドプロシージャ、巻子を公開できます。

サードパーティ HTTP サーバ用に OData プロデューサを設定する方法 [576 ページ]
サードパーティ HTTP サーバ用の OData プロデューサオプションは、設定ファイルで指定します。

OSDL 文のリファレンス [579 ページ]
OSDL 文は、OData プロデューササービスモデルを作成し、特定のテーブル、ビュー、ストアドプロシージャ、関数を公開します。

関連情報

OData プロトコル仕様

1.19.1 OData サーバのアーキテクチャ

OData サーバは、OData プロデューサと HTTP サーバーで構成されます。

OData プロデューサ

SQL Anywhere では、OData プロデューサ は、JDBC API を使用してデータベースサーバに接続する Java Servlet です。OData プロデューサは OData の要求を処理し、データベースと連携します。

次の表は、OData プロデューサが OData の概念をリレーショナルデータベースの概念にマッピングするしくみについて説明します。

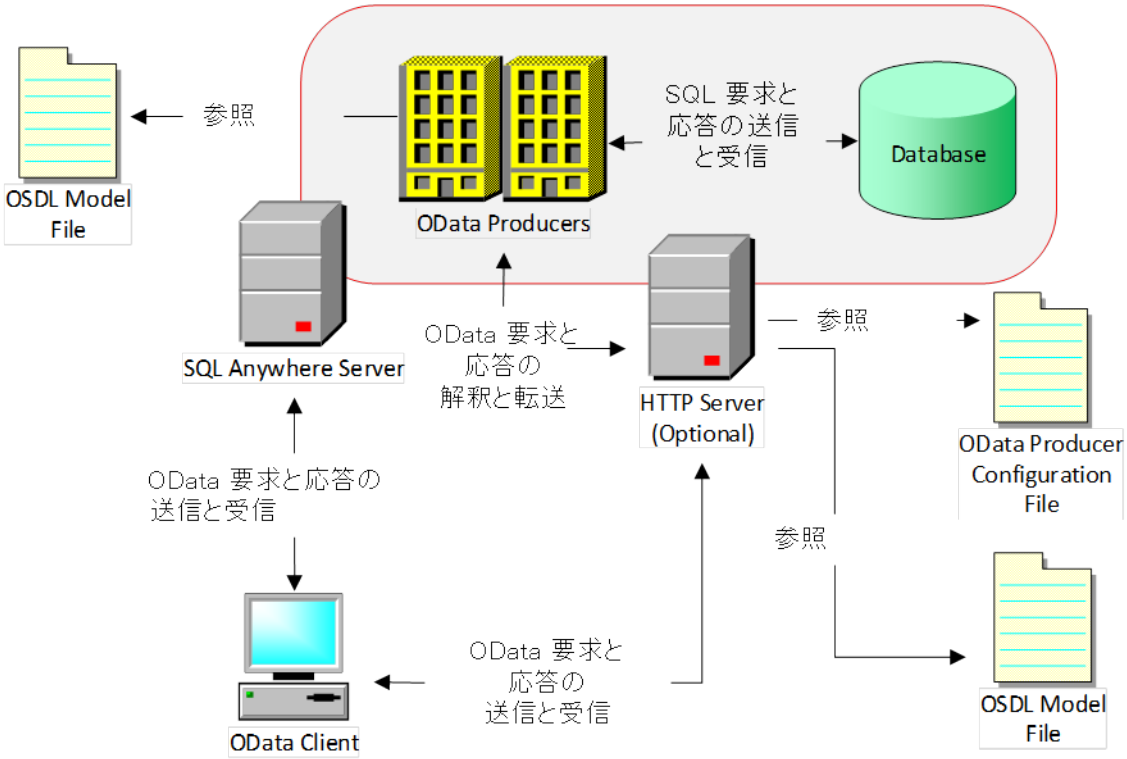
OData の概念	データベースの概念
エンティティ	ロー
エンティティタイプ	テーブル名またはビュー名
エンティティセット	各ローがエンティティタイプインスタンスであるテーブルロー
キー	プライマリキー
関連付けおよびナビゲーションプロパティ	外部キー
プロパティ	カラムまたはカラム値

HTTP サーバ

HTTP サーバは Web クライアントからの OData の要求を処理します。

データベースサーバは Jetty WebServer を OData の HTTP サーバとして使用します。この HTTP サーバは、OData プロデューサのホストとして機能するために必要な Java Servlet コンテナとして機能します。

または、Java Servlet をホストできるソリューションの場合、サードパーティ HTTP サーバを使用して OData 要求を処理することもできます。たとえば、IIS または Apache サーバが要求を Tomcat または Jetty サーバに転送するように設定できます。



OData クライアント要求は URI を通して HTTP サーバに送信され、OData プロデューサによって処理されます。その後、データベースサーバと連携してデータベース要求を発行し、OData 応答の内容を取得します。

各クライアントの OData スキーマは、クライアントのデータベース接続パーミッションに基づいています。クライアントは表示するパーミッションをもたないデータベースオブジェクトを表示または変更できません。

データベースへのクライアントアクセスは、事前に設定された接続文字列か Basic HTTP 認証を使用して許可できます。

OData プロトコルの詳細については、http://www.odata.org/documentation/odata-version-2-0 を参照してください。

関連情報

OData サーバのセキュリティの考慮事項 [568 ページ]
OData プロトコルの制限事項 [567 ページ]
サードパーティ HTTP サーバ用に OData プロデューサを設定する方法 [576 ページ]
OData サーバの例 [572 ページ]
http://www.odata.org/documentation/odata-version-2-0

1.19.2 OData プロトコルの制限事項

OData プロデューサは、OData プロトコルバージョン 2 の仕様に準拠していますが、いくつかの制限と例外があります。

これらの制限事項は、OData プロトコルの仕様により明示的に定義されていません。

OData プロトコルの詳細については、http://www.odata.org/documentation/odata-version-2-0 を参照してください。

スキーマの変更

変更を有効にして OData クライアントに変更を認識させるためにデータベーススキーマを変更する場合、OData プロデューサを再起動してモデルを変更します。

OData クライアントはメタデータをキャッシュするため、運用環境で別のサービスルートを使用するには、OData プロデューサを更新します。

検索文字列フィルタ

長い検索文字列に OData フィルタを使用すると (substringof、indexof など)、検索は最初の 254 バイトに対してのみ実行されます。

substringof の変更

substringof(s1, s2) フィルタは、s1 文字列が s2 のサブ文字列であるかどうかを返します。

$orderby クエリ

エンティティプロパティによってのみソートします。方向による順序付けはサポートされていますが、表現によるソートはサポートされていません。

プロキシテーブル

データベース内のプロキシテーブルであるエンティティをエンティティセットに作成するときは、すべてのキープロパティが明示的に指定されていることを確認します。

DefaultValue 属性

DefaultValue 属性は、データベース内の基本となるカラムに設定できるすべてのデフォルト値に適切とは限らないため、公開されません。

DefaultValue 属性がなくても、プロパティのデフォルト値が null とは限りません。

ストアドプロシージャ

複数の結果セットまたは変数結果 (呼び出し条件に基づくさまざまなデータ型) を返すストアドプロシージャは、サービス操作としてサポートされていません。

サービス操作は、OData プロデューササービスモデルファイルを使用して明示的に公開する必要があります。

INPUT パラメータはサポートされていません。

サポートされていない **OData** プロトコル機能

次の OData プロトコル機能は、OData プロデューサによってサポートされていません。

- 動的プロパティ
- 複合型 (サービス操作からの戻り値を除く)

- メディア型
- 取得エンティティセット要求のある If-Match (If-Match:* を除く) または If-None-Match HTTP ヘッダ

関連情報

OData サーバのセキュリティの考慮事項 [568 ページ]
サードパーティ HTTP サーバ用に OData プロデューサを設定する方法 [576 ページ]
OData サーバの例 [572 ページ]
http://www.odata.org/documentation/odata-version-2-0

1.19.3 OData サーバのセキュリティの考慮事項

セキュリティ対策は、OData サーバを設定する前に考慮する必要があります。

CSRF トークンとサービス操作

クロスサイトリクエストフォージェリから防御するために OData で CSRF トークンを使用する場合、OData プロデューサは、GET 要求で呼び出されたサービス操作がデータベースを変更しないことを想定しています。データベースを変更するすべてのサービス操作は、OData の慣例に従い、POST 要求のみに制限される必要があります。

繰返可能要求

odata_sys_repeatable_request システムテーブルは、繰返可能要求ごとに要求 URL および本文応答のコピーを格納します。このデータは、権限のないユーザには公開しないでください。

データベースで割り当てられた OData 管理者ユーザは、セキュアで、OData プロデューサ専用にする必要があります。ユーザは odata_sys_repeatable_request テーブルへの SELECT アクセス権を持ち、テーブル、インデックス、イベントを作成できる必要があります。

データベースの認証

運用システムにデータベース認証を使用します。テストの場合や、OData サービスモデルファイルを使用することで公開データベーステーブルを制限する読み取り専用プロデューサを使用している場合のみ、認証を使用しないことが推奨されます。

TLS プロトコルを使用しないトラフィック

運用環境への配備では、必ず TLS (SSL) を使用します。

OData 要求を処理するようにデータベースサーバを設定する場合、SSLKeyStore (KEYSTORE) プロトコルオプションを使用し、OData プロデューサを SecureOnly に設定します。OData プロデューサをサードパーティ HTTP サーバに配備する場合、OData サーブレットに安全なトラフィックのみを送信するようにサーバを設定します。

データベースサーバの起動時に SSLKeyStore オプションが指定されていない場合、OData プロデューサとクライアント間のトラフィック (ユーザ ID とパスワードを含む) はプレーンテキストで伝送されます。このオプションはデフォルト設定では指定されていません。

HTTPS 証明書

既知の信頼できる署名局と完全修飾ホスト名に対する強力な検証チェーンを備えたセキュアサーバ証明書を使用します。

次の Java keytool コマンドは、自己署名証明書を備えた Java キーストアを作成します (テストの場合のみ推奨)。

```
keytool -genkeypair -keyalg RSA -keysize 2048
    -keypass sample -validity 1825 -keystore mystore.jks
    -storepass sample -v -alias localhost -sigalg SHA256withRSA
```

OData 用のデータベースサーバを使用する場合、証明書情報を渡すには、`-xs odata` サーバオプションを使用します。サードパーティ HTTP サーバを OData 用に使用する場合、証明書情報を渡す方法についてはサーバのマニュアルを参照してください。

SQL Anywhere バージョン 12 以前のデータベース

HTTP サーバで OData プロデューサを使用して SQL Anywhere バージョン 12 以前のデータベースのクエリを行う場合、すべてのデータベースオブジェクトが公開されます。これらはユーザクレデンシャルに基づいてフィルタされます。

適切なデータベース権限なしでオブジェクトにアクセスしようとすると、データベースサーバはエラーを返します。

関連情報

サードパーティ HTTP サーバ用に OData プロデューサを設定する方法 [576 ページ]
OData プロトコルの制限事項 [567 ページ]
http://www.odata.org/documentation/odata-version-2-0
Jetty

1.19.4 OData サーバの設定方法

OData サーバの配備と設定を行います。

OData サーバの配備

OData サーバを配備するには、次の手順が必要です。

1. データベースサーバを配備します。データベースサーバの配備は、別のトピックで説明しています。
2. Java 外部環境呼び出しのサポートを配備します。外部環境呼び出しのサポートの配備は、別のトピックで説明しています。
3. SQL Anywhere JDBC ドライバを配備します。これには JDBC ドライバの JAR ファイルとサポートライブラリファイルが含まれます。JDBC ドライバの配備は、別のトピックで説明しています。
4. *%SQLANY17%*`¥Java` フォルダから次の JAR ファイルを配備します。
    1. `dbodata.jar`
    2. `jetty-all-server-9.3.7.jar`
    3. `servlet-api-3.1.0.jar`
    4. `sajdbc4.jar` (このファイルは、JDBC クライアントを配備する以前の手順の一部に含まれています)

データベースサーバの設定

データベースサーバは、**`-xs odata`** サーバオプションで起動した場合、定義済みの有効な OOData プロデューサを自動的にロードします。

HTML ファイルなど、他の非 OData コンテンツのサービスを OData サーバと同じポートで提供するために、データベースサーバを手動で設定したり使用したりすることはできません。ただし、**`-xs odata`** サーバオプションで指定できる HTTP サーバオプションもあります。

OData 操作用にデータベースサーバを設定して起動するには、次の手順が必要です。

1. 適切なサーバ設定で **`-xs odata`** サーバオプションを指定してデータベースサーバを起動します。たとえば、次の文では、ポート 8080 で OData 要求を受信するデータベースサーバを起動します。
   dbsrv17 **`-xs odata(ServerPort=8080) mydatabase.db`**
2. CREATE ODATA PRODUCER 文を使用して、新しい OData プロデューサを設定します。
3. (オプション) 繰返可能要求を設定します。
4. (オプション) クロスサイトリクエストフォージェリ (CSRF) 攻撃から保護します。

> **注記**
>
> -xs odata データベースサーバオプションは、Windows と Linux でサポートされます (ARM プラットフォームではサポートされません)。

サードパーティ HTTP サーバの設定

OData プロデューサを Apache や IIS などのサードパーティ HTTP サーバで使用するには、OData プロデューサをサーバに配備する必要があります。OData プロデューサは、バージョン 3.1 の Java Servlet API をサポートする Java Servlet (HTTP サーバにロード可能なサーブレット) として実行します。たとえば、Tomcat を Java Servlet コンテナとして使用し、Apache HTTP サーバとのペアを作成できます。

> **注記**
>
> IIS は Java Servlet を実行できませんが、Servlet 要求を IIS サーバからこれらを実行できる別のサーバにリダイレクトするコネクタを設定できます。

OData プロデューサを配備するプロセスは、HTTP サーバによって異なります。詳細については、使用している HTTP サーバのマニュアルを参照してください。

OData プロデューサの配備には以下の手順が必要です。

1. OData プロデューサ設定ファイルを作成します。
   OData プロデューサは、HTTP サーバのロギング設定を尊重します。HTTP サーバのロギングの詳細については、使用している HTTP サーバのマニュアルを参照してください。
2. (オプション) 繰返可能要求を設定します。
3. (オプション) クロスサイトリクエストフォージェリ (CSRF) 攻撃から保護します。
4. *%SQLANY17%*¥Java¥dbodata.jar とともに OData プロデューサ設定ファイルを配備します。
5. JDBC クライアントを配備します。これには JDBC ドライバとサポートファイルが含まれます。
6. dbodata.jar の com.sap.odata.producer.servlets.ODataServlet クラスを含むように HTTP サーバを設定します。これは以下の操作を実行します。
   1. 設定ファイルの ServicePath で指定したサブパスに対する要求を受信します。
   2. サーブレットのコンテキスト内で OData プロデューサ設定ファイルを検索します。
   3. OData プロデューサ設定ファイルの名前に設定した *odataConfigFile* サーブレットの init パラメータ値で初期化されます。
   4. (オプション) *odataProducerName* サーブレットの init パラメータ値を OData プロデューサの名前に設定します。複数の OData プロデューサが同じ設定ファイルを共有する場合は、この手順が必要です。
   5. (オプション) OData プロデューサは、HTTPS トラフィックからの要求、または指定されたインタフェース/ポート (可能な場合) からの要求のみを受け取ります。または、SecureOnly プロデューサオプションが必要になります。

> **注記**
>
> Java Servlet としての OData プロデューサは、Windows と Linux で公式にサポートされます (ARM プラットフォームではサポートされません)。

関連情報

JDBC クライアントの配備 [827 ページ]
サードパーティ HTTP サーバ用に OData プロデューサを設定する方法 [576 ページ]
OData サーバのセキュリティの考慮事項 [568 ページ]

繰返可能要求の設定方法 [572 ページ]
クロスサイトリクエストフォージェリ攻撃からの保護方法 [574 ページ]
OData サーバの例 [572 ページ]

1.19.5 OData サーバの例

OData サーバのサンプルは、*%SQLANYSAMP17%*¥SQLAnywhere¥ ディレクトリで使用できます。サンプルはデータベースサーバを起動し、OData クライアントを設定し、これらの間で要求と応答を送信する方法を説明します。

内容を変更する前に、元のサンプルのバックアップコピーを作成します。

すべてのサンプルには Java SE バージョン 7 以降が必要です。

.NET クライアントによる OData サーバとの接続

OData SalesOrder サンプルは、Microsoft .NET OData クライアントを使用して、データベースサーバに対する OData 要求を送信する方法を説明します。

詳細については、*%SQLANYSAMP17%*¥SQLAnywhere¥ODataSalesOrders¥readme.txt を参照してください。

Java クライアントによる OData サーバとの接続

OData セキュリティサンプルは、OData4J Java クライアントを使用して、サンプルデータベースに接続されるデータベースサーバに、HTTPS を介して OData 要求を送信する方法を説明します。

このサンプルは、OData プロデューサのサービスモデルとデータベース認証を使用する方法を説明します。

詳細については、*%SQLANYSAMP17%*¥SQLAnywhere¥ODataSecurity¥readme.txt を参照してください。

関連情報

サードパーティ HTTP サーバ用に OData プロデューサを設定する方法 [576 ページ]

1.19.6 繰返可能要求の設定方法

繰返可能要求を使用すると、要求が OData プロデューサに送信されたかどうかがクライアントにはわからない場合でも、データベースの一貫性を損なうことや、意図しない副作用を発生させることなく、OData クライアントはデータ変更要求 (UPDATE や DELETE など) を再送信できます。

要求を繰返可能にするには、値がグローバルに一意で大文字と小文字を区別しない有効な HTTP タイムスタンプと RequestID HTTP ヘッダを備えた有効な RepeatabilityCreation HTTP ヘッダをクライアントで使用する必要があります。RequestID の値を null にすることはできません。また、ASCII 非制御文字のみを使用し、最大長は 256 文字です。

OData プロデューサが要求を繰返可能として受け入れた場合、次の HTTP 応答ヘッダが応答に含まれます。

```
RepeatabilityResult: accepted
```

同じ作成時刻、要求 ID、URL、要求先ユーザを持つ要求を受信した場合、OData プロデューサは要求を実行しません。元の要求で生成されたものと同じ応答を (エラーも含めて) 返します。

繰返可能要求を有効にするには、次のタスクを実行します。

- OData 管理者ユーザをデータベースに作成します。
  OData プロデューサがデータベースサーバに埋め込まれている場合、このユーザは AUTHENTICATE ODATA システム権限を持っている必要があります。
- CREATE ODATA PRODUCER 文の ADMIN USER 句または OData プロデューサ設定ファイルの ODataAdmin オプションで新しいユーザを参照します。
  ユーザはセキュアで、OData プロデューサ専用にする必要があります。ユーザは odata_sys_repeatable_request テーブルへの SELECT アクセス権を持ち、テーブル、インデックス、イベントを作成できる必要があります。
- (オプション) CREATE ODATA PRODUCER 文の USING 句または OData プロデューサ設定ファイルで RepeatRequestForDays オプションを指定します。
  このオプションは、繰返可能要求の有効日数を示します。

> **注記**
>
> 繰返可能要求は GET 要求には対応していません。繰返可能サービス操作は POST 要求になっている必要があります。OData プロデューサは、GET 要求で呼び出されるサービス操作がデータベースを変更しないことを前提とします。データベースを変更するすべてのサービス操作は、OData の慣例に従い、POST 要求のみに制限される必要があります。

開発者は、プロデューサ設定ファイル内の RepeatRequestForDays データベースオプションで日数を指定することによって、繰返可能要求の有効期間を制御できます。デフォルト値は 2、最小値は 1、最大値は 31 です。値は整数で指定します。

繰返可能要求は、POST サービス操作とともに使用できるほか、バッチ要求の ChangeSet 内部で使用できます。バッチ要求の全体を繰返可能とマークすることはできません。

サポートテーブルとインフラストラクチャ

繰返可能要求が有効な状態で OData プロデューサが最初に起動されたとき、OData プロデューサは OData 管理者ユーザを使用して、繰返可能要求のサポートに必要なテーブル、インデックス、プロシージャ、イベントを作成します (これらのオブジェクトがまだ存在しない場合)。これらのアクションは最初の要求の処理時に実行されます。

次のデータベースオブジェクトが作成されます。

odata_sys_repeatable_request テーブル

このテーブルは OData 管理者ユーザによって所有されます。管理者ユーザはこのテーブルに対する挿入権限を PUBLIC に付与します。

odata_sys_repeatable_request インデックス

このインデックスは odata_sys_repeatable_request テーブルに対して作成されます。

odata_sys_option テーブル

このテーブルは OData 管理者ユーザによって所有されます。

odata_sys_repeatable_request_cleanup() プロシージャ

このプロシージャは OData 管理者ユーザによって所有され、RepeatRequestForDays オプションに基づいて odata_sys_repeatable_request テーブル内の古い繰返可能要求をクリーンアップします。

odata_sys_repeatable_request_cleanup イベント

この日次イベントは OData 管理者ユーザによって所有され、odata_sys_repeatable_request_cleanup プロシージャを呼び出します。

熟練した開発者は odata_sys_repeatable_request_cleanup イベントスケジュールと odata_sys_repeatable_request_cleanup プロシージャを変更して、日単位ではなく時間単位のクリーンアップを行ったり、odata_sys_repeatable_request テーブルにサイズ制限を追加したりすることができます。

開発者が OData 管理者ユーザを変更した場合は、(以前の管理者ユーザが所有する) 上記のオブジェクトをデータベースから削除する必要があります。

関連情報

サードパーティ HTTP サーバ用に OData プロデューサを設定する方法 [576 ページ]
OData サーバの例 [572 ページ]

1.19.7 クロスサイトリクエストフォージェリ攻撃からの保護方法

CSRF トークンにより、OData プロデューサがクロスサイトリクエストフォージェリ攻撃から保護されます。

CSRF トークンを有効にするには、CREATE ODATA PRODUCER 文の USING 句または OData プロデューサ設定ファイルで CSRFTokenTimeout オプションを指定します。

> **注記**
>
> CSRF トークンを使用できるのは、データベース認証を使用中で、サービスが読込専用の場合のみです。

CSRF トークンが有効になったら、クライアントは変更要求を実行する前に CSRF トークンを OData プロデューサから取得する必要があります。トークンを取得するために、クライアントは X-CSRF-Token HTTP 要求ヘッダを Fetch に設定した GET 要求を実行する必要があります (単語 Fetch では大文字と小文字が区別されるため、要求ヘッダは X-CSRF-Token:Fetch となります)。

OData プロデューサは、X-CSRF-Token ヘッダ応答、および sap-XSRF_`SID_client` という形式のクッキーで新しいトークンを送信することによって、トークン取得要求に応答します。このトークンは、有効期限が切れるまで、変更要求のたびに送信する必要があります。

> **注記**
>
> X-CSRF-Token 要求ヘッダは、有効なトークン 1 つまたはフェッチディレクティブを持つ必要があります。複数の X-CSRF-Token 要求ヘッダはサポートされていません。

変更要求を実行する場合、クライアントは cookie と X-CSRF-Token HTTP 要求ヘッダのトークンを送信する必要があります。BATCH 要求内の個々の変更要求では、cookie または X-CSRF-Token HTTP ヘッダを指定しないでください。この場合、BATCH 要求全体の cookie を指定する必要があります。バッチ操作で CSRF トークンを要求し、同じ BATCH 要求でそれを使用することはできません。

関連情報

サードパーティ HTTP サーバ用に OData プロデューサを設定する方法 [576 ページ]
OData サーバの例 [572 ページ]

1.19.8 OData プロデューサのサービスモデルを作成する方法

オプションの OData Service Definition Language (OSDL) モデルファイルは、OData プロデューササービスモデルを指定して、特定のテーブル、ビュー、ストアドプロシージャ、巻子を公開できます。

CREATE ODATA PRODUCER 文の MODEL 句または OData プロデューサ設定ファイルの *Model* オプションを使用すると、ODSL モデルファイルを参照するときに OData プロデューサにモデルが適用されます。

エンティティ、サービス操作、関連付けの公開はユーザ権限にのみ基づいています。次の表に、OData プロデューサがモデルの内容に基づいてエンティティおよび公開する関連付けを選択する仕組みを示します。

モデルの内容	エンティティ	関連付け
モデルなし	データベースから	データベースから
サービス操作	データベースから	データベースから
サービス操作あり、またはなしのエンティティ	モデルの制限を受ける	データベースから
関連付け、エンティティ、サービス操作	モデルの制限を受ける	モデルから

多対多の関係はモデルファイルでのみ定義可能です。

モデルで明示的に定義され、ユーザがアクセス可能なサービス操作のみが公開されます。モデルが提供されない場合、サービス操作は公開されません。

関連情報

サードパーティ HTTP サーバ用に OData プロデューサを設定する方法 [576 ページ]
OData サーバのアーキテクチャ [565 ページ]
OSDL 文のリファレンス [579 ページ]
service OSDL 文 [580 ページ]
OData サーバの例 [572 ページ]

1.19.9 サードパーティ HTTP サーバ用に OData プロデューサを設定する方法

サードパーティ HTTP サーバ用の OData プロデューサオプションは、設定ファイルで指定します。

サードパーティ HTTP サーバで OData プロデューサを使用する場合、設定ファイルが必要です。

> **注記**
>
> 設定ファイルでは、各バイトが 1 つの Latin1 文字である ISO 8859-1 文字エンコーディングを使用する必要があります。特殊文字および Latin1 にない文字は、Unicode エスケープ文字を使用してキーと要素で表します。エスケープシーケンスでは、単一の 'u' 文字のみが許可されています。native2ascii Java ツールを使用すると、他の文字エンコーディングとの間で設定ファイルを変換できます。

次の OData プロデューサオプションと接続パラメータ設定は、設定ファイルで指定できます。

1 つの設定ファイルを使用して、複数のプロデューサを設定できます。その場合、すべてのプロデューサに名前を付ける必要があります。プロデューサ名で使用できるのは、ASCII 文字、ASCII 数字、アンダースコア文字 ([a-zA-Z0-9_]) のみです。

オプションを特定の OData プロデューサに適用するには、[`producer-name.`] プレフィックス (下記の表を参照) を使用します。ここで、`producer-name` は、設定を適用する OData プロデューサの名前です。プレフィックスがないすべてのオプションはグローバルとみなされます。個々の OData プロデューサオプションは、指定されたグローバルオプションよりも優先されます。

オプション	説明
[`producer-name.`]*Authentication* = { *none* | *database* }	この OData プロデューサがデータベースへの接続に使用するクレデンシャルを指定します。 デフォルト設定の *database* は、各ユーザが DbConnectionString オプションに追加されるパーソナル化クレデンシャルで接続し、独自の完全なデータベース接続文字列を形成することを示します。これらのクレデンシャルは、HTTP 基本認証を使用して要求されます。 *none* 設定は、すべてのユーザが DbConnectionString オプションによって示される同じ接続文字列を使用して接続することを示します。この設定は、テストの場合や読み取り専用プロデューサに対して使用することを推奨します。詳細については、"OData サーバのセキュリティの考慮事項" を参照してください。
[`producer-name.`]*ConnectionPoolMaximum* =`num-max-connections`	この OData プロデューサが接続プールで使用するために開き続ける同時接続の最大数を示します。 少ない接続は、サーバの負荷に応じて、接続プールによって使用されます。 デフォルトでは、接続プールのサイズは、データベースサーバでサポートされる同時接続の最大数の半分に制限されています。

オプション	説明
[producer-name.]*CSRFTokenTimeout* =num-seconds-valid	CSRF トークンチェックを有効にし、トークンを有効にする秒数を指定します。 デフォルトでは、この値は 0 です。つまり、CSRF トークンチェックは無効です。それ以外の場合、秒数は 1 ～ 1800 の有効な整数である必要があります。
[producer-name.]*DbConnectionString* =connection-string	データベースに接続するための接続文字列を指定します。 認証オプションがデータベースに設定されている場合、接続文字列では *UID* および *PWD* パラメータを除外する必要があります。
[producer-name.]*DbProduct =sqlanywhere*	OData プロデューサが SQL Anywhere データベースの内容を処理することを指定します。
[producer-name.]*Model* =path-and-filename	OData サービス (ユーザ権限が適用されます) でどのテーブルとビューを公開するかを示す OData プロデューサのサービスモデルを含む OData Service Definition Language (OSDL) ファイルのパスとファイル名を指定します。 パスは、サーブレットのコンテキストおよび OData サーバの現在のディレクトリを基準にしたものです。パスおよびファイル名は、*${variable-name}* 形式の環境変数参照用に処理されます。 デフォルトの動作では、ユーザが権限を持つすべてのテーブルとビューが公開され、ストアドプロシージャや関数は公開されません。プライマリキーのないテーブルとビューは公開されません。
[producer-name.]*ModelConnectionString* =connection-string	この OData プロデューサが起動中に OSDL ファイルを検証するために使用する接続文字列を指定します。 OSDL の検証では、登録されたテーブルとカラムが存在すること、キーリストが適切に使用されていること、ファイルがセマンティック上正しいことを確認します。 接続文字列には、*UID* および *PWD* パラメータを含める必要があります (それらのパラメータを提供する *DSN* が含まれている場合を除きます)。 デフォルト動作では、OSDL ファイルが有効である (最初の要求の処理時に、モデルが検証される) ことが前提とされます。

オプション	説明
[producer-name.]*ODataAdmin* =admin-username:admin-password	OData 管理者として機能しているデータベースユーザをログイン情報で参照します。 OData プロデューサはこのユーザを使用して、繰返可能要求を管理するためのテーブル、プロシージャ、イベントを作成します。データベース管理者は、繰返可能要求をクリーンアップするためのイベントを変更できます。 OData サービスにアクセスするためにこのユーザを使用しないでください。
[producer-name.]*PageSize* =num-max-entities	次のリンクを発行する前に、取得エンティティセット応答に含めるエンティティの最大数を指定します。 デフォルト設定は 100 です。
[producer-name.]*ReadOnly* = { *true* | *false* }	変更要求を無視するかどうかを指定します。 デフォルト設定は false です。
[producer-name.]*SecureOnly* = { *true* | *false* }	プロデューサが HTTPS ポートのみで要求を受信するかどうかを指定します。 デフォルト設定は false です。
[producer-name.]*SecureOnly* = { days-number }	繰返可能要求を有効にする長さを指定します。 この値は、1 ～ 31 の範囲内の整数である必要があります。 デフォルト設定は 2 です。
[producer-name.]*ServiceOperationColumnNames* = { *generate* | *database* }	ReturnType で使用される ComplexType のプロパティ名を設定するときに、メタデータのカラム名を生成するのか、データベースの結果セットカラムから取得するのかを指定します。 デフォルト設定では生成します。
[producer-name.]*ServiceRoot* = /path-prefix	OData サーバで OData サービスのルートを指定します。 各 OData プロデューサには一意のサービスルートが必要で、各サービスルートを別のサービスルートのサブパスにすることはできません。各プロデューサのサービスルートは、そのプロデューサに対するサードパーティ HTTP サーバの設定と一致する必要があります。 デフォルト設定は */odata* です。 OData プロデューサリソースには、次の形式の URI でアクセスします。 scheme:host:port/path-prefix/resource-path[query-options]

例

次のサンプルでは、2 つの OData プロデューサ用に設定ファイルを構成する方法を示します。

```
# Shared OData Producer options
# -----------------------------
Authentication = none
ConnectionPoolMaximum = 10
CSRFTokenTimeout = F00
DbProduct = sqlanywhere
ODataAdmin = SuperODataAdmin:SecretPassword
PageSize = 100
ReadOnly = false
RepeatRequestForDays = 3
ServiceOperationColumnNames = generate
# OrderEntryProducer options
# --------------------------
OrderEntryProducer.Model = ../../ordermodel.osdl
OrderEntryProducer.ModelConnectionString =
uid=dba;pwd=passwd;server=orderentry;dbf=orderentry.db
OrderEntryProducer.ServiceRoot = /orders/
OrderEntryProducer.DbConnectionString =
uid=dba;pwd=passwd;server=orderentry;dbf=orderentry.db
# StaffingProducer options
# ------------------------
StaffingProducer.Model = ../../staffingmodel.osdl
StaffingProducer.ModelConnectionString =
uid=dba;pwd=passwd;server=staffing;dbf=staffing.db
StaffingProducer.ServiceRoot = /staffing/
StaffingProducer.DbConnectionString =
uid=dba;pwd=passwd;server=staffing;dbf=staffing.db
```

関連情報

OData サーバの設定方法 [570 ページ]
OData プロデューサのサービスモデルを作成する方法 [575 ページ]
OData サーバのセキュリティの考慮事項 [568 ページ]
OData サーバの例 [572 ページ]

1.19.10 OSDL 文のリファレンス

OSDL 文は、OData プロデューササービスモデルを作成し、特定のテーブル、ビュー、ストアドプロシージャ、関数を公開します。

行の先頭に # 文字を入力してコメントを作成します。

注記

すべてのエンティティ型、プロパティ、サービス操作、パラメータは有効な OData 識別子でなければなりません。OSDL ファイルは UTF8 でエンコーディングされている必要があります。

このセクションの内容:

service OSDL 文 [580 ページ]
サービスのネームスペースを作成します。

関連情報

OData プロデューサのサービスモデルを作成する方法 [575 ページ]
サードパーティ HTTP サーバ用に OData プロデューサを設定する方法 [576 ページ]
OData サーバの例 [572 ページ]

1.19.10.1 service OSDL 文

サービスのネームスペースを作成します。

構文

```
service [ namespace "namespace-name" ] {
    { entity-statement | association-statement | serviceop-statement }
    ...
}
```

パラメータ

namespace-name

パラメータが指定されている場合、*_Container* が `namespace-name` に追加されてコンテナ名が生成されます。
entity-statement

修飾されたエンティティ文。
association-statement

修飾された関係文。
serviceop-statement

修飾されたサービス操作文。

備考

サービスには、複数のエンティティ、関係、またはサービス操作への参照が含まれている場合もあります。

このセクションの内容:

association OSDL 文 [581 ページ]
基盤となる関係テーブルを使用する複雑な関係を含む、エンティティ間の関係を作成します。

entity OSDL 文 [583 ページ]
エンティティを作成します。

serviceop OSDL 文 [585 ページ]
サービスを HTTP GET または POST 操作として公開します。

1.19.10.1.1 association OSDL 文

基盤となる関係テーブルを使用する複雑な関係を含む、エンティティ間の関係を作成します。

構文

```
association "association-name"
    principal "principal-entityset-name" ("column-name"[ , ... ])
  multiplicity "[ 1 | 0...1 | 1..* | * ]"
    dependent "dependent-entityset-name" ("column-name"[ , ... ])
  multiplicity "[ 1 | 0...1 | 1..* | * ]"
    over "owner"."object-name" principal ("column-name"[ , ... ])
  dependent ("column-name"[ , ... ])
```

パラメータ

association-name

関係の名前。
principal-entityset-name

プリンシパルエンティティセットの名前。
dependent-entityset-name

従属エンティティセットの名前。
object-name

データベーステーブルまたはビューの名前。

備考

関係は、データベースで定義された外部キー (モデルで指定されていない場合) を使用して作成されます。

関係テーブルのない関係では、プリンシパル側の結合プロパティの数が、従属側の結合プロパティの数と一致している必要があります。

関係テーブルのある関係では、プリンシパル側の結合プロパティの数が関係テーブルのプリンシパル側の結合プロパティの数と一致している必要があり、従属側の結合プロパティの数が関係テーブルの従属側のプロパティの数と一致している必要があります。

2 つのテーブル間の関係を定義するとき、従属テーブルがプリンシパルテーブルを参照する必要があります。

プリンシパル側と従属側の結合プロパティの順序には関連性があります。プリンシパル側の最初のカラムは、従属側の最初のカラムと一致している、などです。一致するカラムは値も同じ型でなければなりません。

注記

OData プロデューサは、キーカラムまたはカーディナリティ制約の一意性を強制しません。プロデューサは、これらの制約を強制するため、基盤となるデータベースに依存しています。

例

ユーザ DBA が所有する以下のテーブルについて検討します。

```
CREATE OR REPLACE TABLE ThingHolder(
   ID INTEGER PRIMARY KEY DEFAULT AUTOINCREMENT,
   Name VARCHAR(50)
);
CREATE OR REPLACE TABLE Things(
    ID INTEGER PRIMARY KEY DEFAULT AUTOINCREMENT,
    Holder INTEGER,
    Description VARCHAR(50),
    FOREIGN KEY ThingHolderRef(Holder) REFERENCES ThingHolder
);
```

以下は、これらのテーブルの簡単な関係です。

```
service {
    entity "dba"."ThingHolder" as "ThingHolder"
        navigates( "ThingHolderRef" as "MyThings" );
    entity "dba"."Things" as "Things"
        without( "Holder" )
        navigates( "ThingHolderRef" as "MyThingHolder" );
    association "ThingHolderRef"
        principal "ThingHolder" ("ID") multiplicity "0..1"
        dependent "Things" ("holder") multiplicity "*";
}
```

ユーザ DBA が所有する以下のテーブルについて検討します。

```
CREATE OR REPLACE TABLE ThingHolder(
    ID INTEGER,
    Name VARCHAR(50) NULL,
    PRIMARY KEY (ID)
);
CREATE OR REPLACE TABLE Things(
    ID INTEGER,
    Description VARCHAR(50),
    PRIMARY KEY (ID)
);
CREATE OR REPLACE TABLE AssocTable (
    Holder      INTEGER,
    Thing       INTEGER,
    PRIMARY KEY (Holder, Thing),
    FOREIGN KEY ThingHolderRef( Holder ) REFERENCES ThingHolder,
    FOREIGN KEY ThingsRef( Thing ) REFERENCES Things
);
```

以下のサービスモデルは、関係テーブルを使用して関連付けられた最初の 2 つのテーブルを公開します。

```
service {
    entity "dba"."ThingHolder" as "ThingHolder"
        navigates( "ThingHolderRef" as "Things" );
    entity "dba"."Things" as "Things"
        navigates( "ThingHolderRef" as "ThingHolderRef" );
    association "ThingHolderRef"
        principal "ThingHolder" ("ID") multiplicity "0..1"
        dependent "Things" ("ID") multiplicity "*"
        over "dba"."AssocTable" principal ("holder") dependent ("thing");
}
```

1.19.10.1.2 entity OSDL 文

エンティティを作成します。

構文

```
entity "owner"."object-name" [ as "entityset-name" ] [ { with | without } ("included-or-
excluded-column-name"[ , ... ]) ]
    [ keys { ("key-column-name"[ , ... ]) | generate local "key-column-name" } ]
    [ concurrencytoken ("token-column-name" [ , ... ] ) ]
    [ navigates ("association-name" as "navprop-name" [ from { principal | dependent } ]
[, ...] ) ]
```

パラメータ

owner

OData メタデータ内で公開するデータベーステーブルまたはビューを所有するデータベースユーザの名前。

object-name

OData メタデータで公開するデータベーステーブルまたはビューの名前。

entityset-name

`object-name` により定義された公開テーブルまたはビューの名前。この名前は、ネームスペース内で一意である必要があり、他のサービス操作やエンティティセット名と競合してはなりません。デフォルトでは、OData プロデューサはエンティティセット名として `object-name`、次に `owner_object-name` を使用しようとします。

included-or-excluded-column-name

with 句と組み合わせて使用する場合、このパラメータは、`entityset-name` により定義されるエンティティセットに含めるカラム名を指定します。

without 句と組み合わせて使用する場合、このパラメータは、`entityset-name` により定義されるエンティティセットで除外するカラム名を指定します。

複数のカラム名は、参照できます。

key-column-name

keys 句と組み合わせて使用する場合、このパラメータは、`object-name` により定義されたテーブルまたはビューで指定されていない場合にプライマリキーとして使用するカラム名を指定します。

generate local 句と組み合わせて使用する場合、このパラメータは、Edm.Int64 データ型として定義された生成キーを指定します。結果の番号には、開始インデックス 1 が付けられます。

一意の ID を結果に提供するために、生成されたキーを使用します。エンティティの取得または逆参照するために使用することはできません。

生成キーは、現在のセッションの期間中のみ有効です。生成プロパティを $filter または $orderby で使用することはできません。生成プロパティを持つエンティティタイプを関係で使用することはできません。

キーは、複数のカラムを持つ可能性があります。
token-column-name

エンティティインスタンスが変更されると必ず変更されるカラム (プロパティ) の名前。更新トリガにより増分される DEFAULT CURRENT TIMESTAMP カラムや INTEGER カラムなどです。複数のカラム名を参照できます。
association-name

OSDL ファイルで定義された関係への参照。複数の関係を定義できます。
navprop-name

ナビゲーションプロパティ名の名前。

備考

次の制限事項が *keys* 句に適用されます。

- プライマリキーを含まないテーブルまたはビューを参照する場合、キーリストを指定します。
- プライマリキーを含むテーブルを参照する場合、キーリストを指定しないでください。

with キーワードや *without* キーワードに結び付けられたプロパティリストにプライマリキープロパティが含まれる場合、そのプライマリキープロパティは無視されます。プライマリキーカラムは必ず含まれます。

関係の両側が同じ型である場合は、*from* 句を指定します。

concurrencytoken 句は、インスタンスの要求と同時にエンティティインスタンスの状態を識別する ETags を生成します。ETag の生成に使用される SQL は、同時実行性トークンプロパティ (カラム) ごとに SHA256 ハッシュ関数を使用し、同時実行性トークンに含まれているプロパティ (カラム) の型とプロパティの数を考えると複雑になる場合があります。同時実行性トークンは、少数のプロパティで構成されている必要があります。どのプロパティも BLOB 型であってはなりません。

例

次は、テーブルとビューを公開するサービスモデルです。

```
service namespace "DBServer" {
    "dba"."TableWithPrimaryKey";
    "dba"."ViewWithoutPrimaryKey" keys("primary_key1", "primary_key2");
}
```

次は、ETag を生成するサービスモデルです。

```
service {
    entity "dba"."T1" as "T1" concurrencytoken("version") ;
```

```
    entity "dba"."T2" as "T2" concurrencytoken("ver_maj", "ver_min", "ver_bld");
    entity "dba"."T3" as "T3" concurrencytoken("ver_ts");
}
```

1.19.10.1.3 serviceop OSDL 文

サービスを HTTP GET または POST 操作として公開します。

構文

```
serviceop { get | post } "owner"."procedure-or-function-name
    [ as service-name ]
    [ returns multiplicity " { 0 | 1 | * } " ]
```

パラメータ

owner

OData メタデータ内で公開するデータベースストアドプロシージャまたは関数を所有するデータベースユーザの名前。

procedure-or-function-name

OData メタデータで公開するデータベースストアドプロシージャまたは関数の名前。

service-name

OData サービスを公開するための名前。この名前は、ネームスペース内で一意である必要があり、他のサービス操作やエンティティセット名と競合してはなりません。

デフォルトでは、OData プロデューサはサービス名として `procedure-or-function-name`、次に `owner_procedure-or-function-name` を使用しようとします。

returns multiplicity 句

多重度を示すため、次の値が使用されます。

"0"

この値は、出力のないサービス操作に使用します。

"1"

この値は、戻り値の型が単一の単純な型であるか、単一の複雑な型であるサービス操作に使用します。

"*"

この値は、戻り値の型が単純な型のコレクションであるか、複雑な型のコレクションであるサービス操作に使用します。

備考

一意の名前で毎回 2 回宣言することで、GET および POST 操作としてサービス操作を公開します。

OData プロデューサは、必要に応じて結果の複雑な型を作成します。サービス操作の複雑な型には、出力パラメータと、関数の戻り値またはストアドプロシージャの結果が含められます。

OData プロデューサは、GET 要求で呼び出されるサービス操作はデータベースを変更しないことを想定しています。データベースを変更するすべてのサービス操作は、OData の慣例に従い、POST 要求のみに制限される必要があります。

例

次の例は、複数のサービス操作を宣言します。

```
service {
    serviceop post "dba"."P0R0" as "POST0R0" returns multiplicity "0";
    serviceop get "dba"."P0R0" returns multiplicity "0";
    serviceop get "dba"."P1R0" returns multiplicity "0";
    serviceop get "dba"."P0R3" returns multiplicity "*";
    serviceop get "dba"."P1R1" returns multiplicity "1";
}
```

1.20 HTTP Web サービス

SQL Anywhere を使用して、HTTP Web サービスを作成できます。また、他の Web サーバとの間で、要求の送信と応答の受信を行うことができます。

このセクションの内容:

HTTP Web サーバとしてのデータベースサーバ [587 ページ]
データベースサーバには、データベースからオンライン Web サービスを提供できるようにする組み込み HTTP Web サーバが含まれています。

Web クライアントを使用した Web サービスへのアクセス [627 ページ]
データベースサーバを Web クライアントとして使用して、SQL Anywhere HTTP Web サーバによって実行される Web サービスまたは Apache や IIS などのサードパーティの Web サーバにアクセスできます。

Web サービスエラーコードリファレンス [661 ページ]
要求に失敗すると、HTTP サーバで標準の Web サービスエラーが生成されます。これらのエラーには、プロトコル標準と一貫性のある番号が割り当てられています。

HTTP Web サービスの例 [663 ページ]
Web サービスの実装サンプルのいくつかは、*%SQLANYSAMP17%*`¥SQLAnywhere¥HTTP` フォルダにあります。

1.20.1 HTTP Web サーバとしてのデータベースサーバ

データベースサーバには、データベースからオンライン Web サービスを提供できるようにする組み込み HTTP Web サーバが含まれています。

Web ブラウザとクライアントアプリケーションから送信される HTTP 要求を介した HTTP と SOAP を処理できます。Web サービスはデータベースに組み込まれているため、Web サーバのパフォーマンスが最適化されます。

Web サービスは、JDBC や ODBC などの従来のインタフェースの代わりになる方法をクライアントアプリケーションに提供します。追加のコンポーネントが必要なく、Perl や Python などのスクリプト言語を含む各種の言語で記述されたマルチプラットフォームクライアントアプリケーションからアクセスできるため、これらは簡単に配備されます。

データベースサーバは、HTTP Web サーバでの Web サービスの提供に加え、SOAP または HTTP クライアントアプリケーションとして機能できるため、インターネットで利用できる標準の Web サービス (SQL Anywhere HTTP Web サーバによって提供されるサービスなど) にアクセスできます。

このセクションの内容:

HTTP Web サーバとしてデータベースサーバを使用するためのクイックスタート [587 ページ]
SQL Anywhere HTTP Web サーバを起動し、Web サービスを作成して、Web ブラウザから Web サービスにアクセスできます。

HTTP Web サーバを起動する方法 [589 ページ]
SQL Anywhere データベースサーバに組み込まれている HTTP Web サーバを起動するには、-xs サーバオプションまたは sp_start_listener システムプロシージャを使用します。

Web サービスとは? [592 ページ]
Web サービスは、コンピュータ間のデータ転送と相互運用性を支援するソフトウェアを指します。

HTTP Web サーバで Web サービスアプリケーションを開発する方法 [603 ページ]
カスタマイズされた Web ページは、Web サービスから呼び出されるストアドプロシージャによって作成できます。非常に基本的な Web ページの例を書きに示します。

SQL AnywhereHTTP Web サーバを参照する方法 [623 ページ]
Web サービスの命名および設計方法によって、有効な URL の内容が決まります。

関連情報

Web クライアントを使用した Web サービスへのアクセス [627 ページ]

1.20.1.1 HTTP Web サーバとしてデータベースサーバを使用するためのクイックスタート

SQL Anywhere HTTP Web サーバを起動し、Web サービスを作成して、Web ブラウザから Web サービスにアクセスできます。

SQL Anywhere HTTP Web サーバおよび HTTP Web サービスを作成するには、次のタスクを実行します。

1. HTTP Web サーバとデータベースを起動します。たとえば、コマンドプロンプトで次のコマンドを実行します。

```
dbsrv17 -xs http(port=8082) "%SQLANYSAMP17%¥demo.db"
```

> **注記**
>
> ネットワーク上でアクセスできるデータベースサーバを起動するには、dbsrv17 コマンドを使用します。ローカルホストからのみアクセスできるパーソナルデータベースサーバを起動するには、dbeng17 コマンドを使用します。

   -xs http(port=8082) オプションは、ポート 8082 で HTTP 要求を待機するようにサーバに指示します。ポート 8082 で Web サーバがすでに動作している場合は、別のポート番号を使用します。
2. CREATE SERVICE 文を使用して、Web ブラウザからの要求に応答する Web サービスを作成します。
   1. 次のコマンドを実行することで、Interactive SQL を使用して *demo.db* データベースに接続します。

```
dbisql -c "dbf=%SQLANYSAMP17%¥demo.db;uid=DBA;pwd=sql"
```

   2. データベースに新しい Web サービスを作成します。
      Interactive SQL で次の SQL 文を実行します。

```
CREATE SERVICE SampleWebService
    TYPE 'web-service-type-clause'
    AUTHORIZATION OFF
    USER DBA
    AS SELECT 'Hello world!';
```

      `web-service-type-clause` を目的の Web サービスタイプに置き換えます。Web ブラウザの互換性を保つため、HTML type 句をおすすめします。汎用 HTTP Web サービスの type 句には、他に XML、RAW、JSON があります。
      CREATE SERVICE 文により、SELECT 文の結果セットを返す SampleWebService Web サービスが作成されます。この例では、この文から "Hello world!" が返されます。
      AUTHORIZATION OFF 句は、Web サービスのアクセスに認証が不要であることを示します。
      USER DBA 文は、サービス文を DBA ログイン名で実行する必要があることを示します。
      AS SELECT 句を指定すると、サービスからテーブルまたは関数を選択したり、データを直接表示したりできます。
      AS CALL は、ストアドプロシージャを呼び出す代替句として使用します。
3. Web ブラウザで Web サービスを表示します。
   SQL Anywhere HTTP Web サーバが実行されているコンピュータで、Microsoft Internet Explorer や Firefox などの Web ブラウザを開き、次の URL にアクセスします。

```
http://localhost:8082/demo/SampleWebService
```

   Web ブラウザは、この URL からポート 8082 上の HTTP Web サーバにリダイレクトされます。SampleWebService は "Hello world" と印字します。結果セットの出力は、手順 2 の `web-service-type-clause` で指定したフォーマットで表示されます。

他のサンプルリソース

サンプルプログラムは *%SQLANYSAMP17%*¥SQLAnywhere¥http ディレクトリにあります。

関連情報

HTTP Web サーバを起動する方法 [589 ページ]
Web サービスタイプ [592 ページ]

1.20.1.2 HTTP Web サーバを起動する方法

SQL Anywhere データベースサーバに組み込まれている HTTP Web サーバを起動するには、-xs サーバオプションまたは sp_start_listener システムプロシージャを使用します。

-xs サーバオプションを指定してデータベースサーバを起動すると、HTTP Web サーバが自動的に起動します。

HTTP Web サーバは、sp_start_listener システムプロシージャで起動することもできます。

どちらの方法でも、次のタスクを実行できます。

- Web サービス要求を受信する Web サービスプロトコルの有効化
- サーバポート、ロギング、タイムアウト条件、最大要求サイズなどのネットワークプロトコルオプションの設定

コマンドの一般的な形式を次に示します。

```
dbsrv17 -xs protocol-type(protocol-options) your-database-name.db
```

ストアドプロシージャ呼び出しの一般的な形式を次に示します。

```
CALL sp_start_listener(protocol-type,address-port,protocol-options);
```

`protocol-type` および `protocol-options` を、サポートされている次のいずれかのプロトコルとオプションに置き換えます。

HTTP

このプロトコルは、HTTP 接続を受信する場合に使用します。例を示します。

```
dbsrv17 -xs HTTP(PORT=8082) services.db
```

```
CALL sp_start_listener('HTTP','127.0.0.1:8082','LOG=c:¥¥temp¥¥http.log');
```

HTTPS

このプロトコルは、HTTPS 接続を受信する場合に使用します。TLS バージョン 1.0 以降がサポートされています。例を示します。

```
dbsrv17 -xs "HTTPS(FIPS=N;PORT=443;IDENTITY=C:¥Users¥Public¥Documents¥SQL
Anywhere 17¥Samples¥Certificates¥rsaserver.id;IDENTITY_PASSWORD=test)"
services.db
```

```
CALL sp_start_listener('HTTPS','127.0.0.1:443','LOG=c:¥¥temp¥
¥https.log;FIPS=N;IDENTITY=C:¥¥SQLA¥¥Samples¥¥Certificates¥
¥rsaserver.id;IDENTITY_PASSWORD=test');
```

サポートされている各プロトコルでは、ネットワークプロトコルオプションを使用できます。このオプションでは、プロトコルの挙動を制御できます。

このセクションの内容:

ネットワークプロトコルオプションの設定 [590 ページ]
ネットワークプロトコルオプションは、指定された Web サービスプロトコルを制御するための省略可能な設定です。

複数の HTTP Web サーバの起動方法 [591 ページ]
複数の HTTP Web サーバ構成では、データベース間で Web サービスを作成し、それを 1 つの Web サイトの一部として表示できます。

関連情報

Web サービスとは? [592 ページ]

1.20.1.2.1 ネットワークプロトコルオプションの設定

ネットワークプロトコルオプションは、指定された Web サービスプロトコルを制御するための省略可能な設定です。

-xs データベースサーバオプションを指定してデータベースサーバを起動すると、これらの設定をコマンドラインで実行できます。たとえば、次のコマンドラインは、PORT、FIPS、Identity、Identity_Password ネットワークプロトコルオプションを指定した HTTPS リスナを設定します。

```
dbsrv17 -xs https(PORT=544;FIPS=YES;
    IDENTITY=rsaserver.id;IDENTITY_PASSWORD=test) your-database-name.db
```

このコマンドは、`your-database-name.db` データベースに対して、HTTPS Web サービスプロトコルを有効にするデータベースサーバを起動します。

これらの設定は、sp_start_listener システムプロシージャを呼び出して設定することもできます。たとえば、次の SQL CALL は、FIPS、Identity、Identity_Password ネットワークプロトコルオプションを指定した HTTPS リスナを localhost (ポート 544 を使用) で設定します。

```
CALL
sp_start_listener('HTTPS','127.0.0.1:544','FIPS=YES;IDENTITY=rsaserver.id;IDENTITY_P
ASSWORD=test');
```

ネットワークプロトコルオプションは、Web サーバが次のタスクを実行することを示します。

- デフォルトの HTTPS ポート (443) ではなくポート 544 で受信します。
- FIPS 認定セキュリティアルゴリズムを有効にして、通信を暗号化します。
- パブリック証明書とプライベートキーが格納されている、指定された ID ファイル `rsaserver.id` を検索します。
- 指定した ID パスワード **`test`** を使用して、秘密鍵を解読します。

次の表に、Web サービスプロトコルで一般的に使用されるネットワークプロトコルオプションを示します。

ネットワークプロトコルオプション	使用可能な Web サービスプロトコル	説明
DatabaseName (DBN)	HTTP、HTTPS	Web 要求を処理するときに使用するデータベースの名前を指定します。また、REQUIRED や AUTO キーワードを使用して URL の一部としてデータベース名が必要かどうかを指定します。
FIPS	HTTPS	データベースファイルを暗号化したり、データベースクライアント/サーバ通信や Web サービスにおける通信を暗号化するために、FIPS 認定のセキュリティアルゴリズムを有効にします。
Identity	HTTPS	セキュア HTTPS 接続に使用する ID ファイルの名前を指定します。
Identity_Password	HTTPS	ID ファイルのパスワードを指定します。
LocalOnly (LO)	HTTP、HTTPS	データベースサーバが接続をローカルコンピュータのみに制限できるようにします。
LogFile (LOG)	HTTP、HTTPS	データベースサーバが Web 要求に関する情報を書き込むファイル名を指定します。
LogFormat (LF)	HTTP、HTTPS	データベースサーバがメッセージログファイルに書き込む Web 要求に関する情報メッセージのフォーマットを制御し、メッセージに表示されるフィールドを指定します。
LogOptions (LOPT)	HTTP、HTTPS	データベースサーバが Web 要求に関する情報を書き込むログに記録されるメッセージタイプを指定します。
ServerPort (PORT)	HTTP、HTTPS	データベースサーバが要求に対して受信しているポートを指定します。

ServerPort (PORT) プロトコルオプションは、sp_start_listener では使用できません。

1.20.1.2.2 複数の HTTP Web サーバの起動方法

複数の HTTP Web サーバ構成では、データベース間で Web サービスを作成し、それを 1 つの Web サイトの一部として表示できます。

-xs データベースサーバオプションの複数のインスタンスを使用して、複数の HTTP Web サーバを起動できます。このタスクは、HTTP Web サーバごとにユニークなポート番号を指定することによって実行されます。

例

この例のコマンドラインでは、2 つの HTTP Web サービスを起動します。1 つは `your-first-database.db` 用、もう 1 つは `your-second-database.db` 用です。

```
dbsrv17 -xs http(port=80;dbn=your-first-database),http(port=8800;dbn=your-second-database)
```

```
    your-first-database.db your-second-database.db
```

次の例は前の例と同等です。

```
dbsrv17 -xs http(port=80;dbn=your-first-database) -xs http(port=8800;dbn=your-
second-database)
    your-first-database.db your-second-database.db
```

1.20.1.3 Web サービスとは?

Web サービスは、コンピュータ間のデータ転送と相互運用性を支援するソフトウェアを指します。

Web サービスはビジネスロジックのセグメントをインターネットを介して使用できるようにします。HTTP Web サーバで Web サービスを管理すると、URL がクライアントで使用可能になります。URL の指定時に使用される表記規則によって、サーバと Web クライアント間の通信方法が決まります。

Web サービスの管理には、次のタスクが関係します。

- 管理する Web サービスのタイプの選択
- 対象となる Web サービスの作成と管理

Web サービスは、データベースで作成し、格納できます。

このセクションの内容:

Web サービスタイプ [592 ページ]
SQL Anywhere HTTP Web サーバは、HTML、XML、RAW、JSON、および SOAP/DISH Web サービスタイプをサポートしています。

Web サービスの管理 [594 ページ]
SQL 文は、Web サービスの作成、変更、削除、有効化または無効化に使用されます。

Web サービスの接続プーリング [602 ページ]
Web サービスを公開する各データベースでは、データベース接続のプールにアクセスできます。特定の USER 句で定義されるすべてのサービスで同じ接続プールグループを共有するように、プールはユーザ名ごとにグループ化されます。

1.20.1.3.1 Web サービスタイプ

SQL Anywhere HTTP Web サーバは、HTML、XML、RAW、JSON、および SOAP/DISH Web サービスタイプをサポートしています。

Web ブラウザまたはクライアントアプリケーションから SQL Anywhere HTTP Web サーバに対して Web サービス要求を行うと、要求が処理され、結果セットが応答で返されます。結果セットのフォーマットと結果セットを返す方法を制御するいくつかの Web サービスタイプが複数サポートされています。適切な Web サービスタイプを選択してから、CREATE SERVICE 文または ALTER SERVICE 文の TYPE 句を使用して Web サービスタイプを指定します。

サポートされる Web サービスタイプは次のとおりです。

HTML

文、関数、またはプロシージャの結果セットは、テーブルが含まれる HTML ドキュメントにフォーマットされます。Web ブラウザに、HTML ドキュメントの本文が表示されます。

XML

文、関数、またはプロシージャの結果セットは、XML ドキュメントとして返されます。XML でフォーマットされていない結果セットは、自動的に XML にフォーマットされます。Web ブラウザに、タグと属性を含む未加工の XML コードが表示されます。

XML フォーマットは、次の SQL 文の例に示すように、SELECT 文で FOR XML RAW 句を指定することと同じです。

```
SELECT * FROM table-name FOR XML RAW
```

RAW

文、関数、またはプロシージャの結果セットは、自動フォーマットなしで返されます。

このサービスタイプでは、結果セットを最大限に制御できます。ただし、ストアドプロシージャ内で必要なマークアップ (HTML、XML) を明示的に作成することによって応答を生成する必要があります。SA_SET_HTTP_HEADER システムプロシージャを使用して、MIME タイプを指定する HTTP Content-Type ヘッダを設定すると、Web ブラウザで結果セットを正しく表示できます。

Web ページは、RAW Web サービスタイプを使用してカスタマイズできます。

JSON

文、関数、またはプロシージャの結果セットは、JSON (JavaScript Object Notation) で返されます。JavaScript Object Notation (JSON) は、言語に依存しないテキストベースのデータ交換フォーマットで、JavaScript データの直列化のために開発されました。JSON には、次の 4 つの基本型があります。文字列、数値、ブール、および NULL です。また、JSON では、オブジェクトと配列という 2 つの構造型も表現されます。

このサービスは、Web アプリケーションに対して HTTP 呼び出しを行うために AJAX で使用されます。JSON タイプの例については、*%SQLANYSAMP17%*`¥SQLAnywhere¥HTTP¥json_sample.sql` を参照してください。

SOAP

文、関数、またはプロシージャの結果セットは、SOAP 応答として返されます。SOAP サービスには、SOAP をサポートする異なるクライアントアプリケーションに対してデータアクセスを提供する、共通データ交換標準が備えられています。SOAP 要求と応答のエンベロープは、HTTP (HTTP を介した SOAP) を使用して XML ペイロードとして転送されます。SOAP サービスへの要求は、汎用 HTTP 要求ではなく有効な SOAP 要求である必要があります。SOAP サービスの出力は、CREATE 文または ALTER SERVICE 文の FORMAT 属性と DATATYPE 属性を使用することで調節できます。

DISH

DISH サービス (SOAP ハンドラを決定) は、SOAP 終了ポイントです。DISH サービスからアクセスできるすべての SOAP 操作 (SOAP サービス) を記述する WSDL (Web Services Description Language) ドキュメントを公開します。SOAP クライアントツールキットは、WSDL に基づいたインタフェースを使用してクライアントアプリケーションを構築します。SOAP クライアントアプリケーションは、すべての SOAP 要求を SOAP 終了ポイント (DISH サービス) にリダイレクトします。

例

次の例は、RAW サービスタイプを使用した汎用 HTTP Web サービスの作成を示しています。

```
CREATE PROCEDURE sp_echotext(str LONG VARCHAR)
BEGIN
    CALL sa_set_http_header( 'Content-Type', 'text/plain' );
```

```
    SELECT str;
END;
CREATE SERVICE SampleWebService
    TYPE 'RAW'
    AUTHORIZATION OFF
    USER DBA
    AS CALL sp_echotext ( :str );
```

関連情報

Web ページをカスタマイズする方法 [604 ページ]
Web サービスを作成または変更する方法 [595 ページ]
HTTP 要求と SOAP 要求の構造 [660 ページ]
DISH サービスを作成する方法 [597 ページ]
SOAP over HTTP サービスを作成する方法 [596 ページ]
JSON の紹介

1.20.1.3.2 Web サービスの管理

SQL 文は、Web サービスの作成、変更、削除、有効化または無効化に使用されます。

Web サービスの管理には、次のタスクが関係します。

Web サービスの作成または変更

Web ブラウザインタフェースをサポートする Web アプリケーションを提供し、REST および SOAP 手法を使用して Web を介したデータ交換を提供するために、Web サービスを作成または変更します。

Web サービスの削除

Web サービスを削除すると、そのサービスに対して作成された後続の要求によって `404 Not Found` HTTP ステータスメッセージが返されます。`root` Web サービスが存在する場合、すべての未解決の要求 (意図的なまたは意図的でない要求) が処理されます。

Web サービスのコメント付け

コメント付けはオプションであり、Web サービスに関するドキュメントを提供できます。

root Web サービスの作成とカスタマイズ

`root` Web サービスを作成して、他の Web サービス要求と一致しない HTTP 要求を処理できます。

Web サービスの有効化と無効化

無効化された Web サービスからは、`404 Not Found` HTTP ステータスメッセージが返されます。CREATE SERVICE 文の METHOD 句は、特定の Web サービスに対して呼び出しできる HTTP メソッドを指定します。

このセクションの内容:

Web サービスを作成または変更する方法 [595 ページ]

Web サービスの作成または変更には、CREATE SERVICE 文または ALTER SERVICE 文をそれぞれ使用する必要があります。これらの文を Interactive SQL で実行すると、異なるタイプの Web サービスを作成できます。

Web サービスを削除する方法 [599 ページ]
Web サービスを削除すると、そのサービスに対して作成された後続の要求によって `404 Not Found` HTTP ステータスメッセージが返されます。**`root`** Web サービスが存在する場合、すべての未解決の要求 (意図的なまたは意図的でない要求) が処理されます。

Web サービスにコメントを付ける方法 [599 ページ]
Web サービスのドキュメントを提供するには、COMMENT ON SERVICE 文を使用する必要があります。

root Web サービスを作成し、カスタマイズする方法 [600 ページ]
どの Web サービス要求とも一致しない HTTP クライアント要求は、**`root`** Web サービスが定義されていると、**`root`** Web サービスによって処理されます。

Web サービスの SQL 文 [601 ページ]
SQL 文は、データベースに Web サービスを実装するために使用されます。

関連情報

HTTP Web サーバとしてデータベースサーバを使用するためのクイックスタート [587 ページ]

1.20.1.3.2.1 Web サービスを作成または変更する方法

Web サービスの作成または変更には、CREATE SERVICE 文または ALTER SERVICE 文をそれぞれ使用する必要があります。これらの文を Interactive SQL で実行すると、異なるタイプの Web サービスを作成できます。

このセクションの内容:

HTTP Web サービスを作成する方法 [596 ページ]
HTTP Web サービスは、HTML、XML、RAW、または JSON に分類されます。すべての HTTP Web サービスは、CREATE SERVICE 文および ALTER SERVICE 文の同じ構文を使用して作成または変更できます。

SOAP over HTTP サービスを作成する方法 [596 ページ]
SOAP は、多くの開発環境でサポートされているデータ交換標準です。

DISH サービスを作成する方法 [597 ページ]
DISH サービスは、SOAP サービスのグループの SOAP 終了ポイントとして機能します。また、DISH サービスでは、WSDL (Web Services Description Language) ドキュメントも自動的に構築され、WSDL で記述される SOAP サービスとのデータ交換に必要なインタフェースを SOAP クライアントツールキットで生成できるようになります。

1.20.1.3.2.1.1 HTTP Web サービスを作成する方法

HTTP Web サービスは、HTML、XML、RAW、または JSON に分類されます。すべての HTTP Web サービスは、CREATE SERVICE 文および ALTER SERVICE 文の同じ構文を使用して作成または変更できます。

例

Interactive SQL で次の文を実行して、HTTP Web サーバでサンプルの汎用 HTTP Web サービスを作成します。

```
CREATE SERVICE SampleWebService
    TYPE 'web-service-type-clause'
    URL OFF
    USER DBA
    AUTHORIZATION OFF
    AS sql-statement;
```

CREATE SERVICE 文は、SampleWebService という新しい Web サービスを作成し、`sql-statement` の結果セットを返します。`sql-statement` を SELECT 文に置き換えてテーブルまたはビューからデータを直接選択するか、CALL 文に置き換えてデータベース内のストアドプロシージャを呼び出すことができます。

`web-service-type-clause` を目的の Web サービスタイプに置き換えます。HTTP Web サービスに有効な句には、HTML、XML、RAW、JSON があります。

Web ブラウザで SampleWebService サービスにアクセスすることで、このサービスに対して生成された結果セットを表示できます。

関連情報

HTTP Web サーバで Web サービスアプリケーションを開発する方法 [603 ページ]
Web サービスタイプ [592 ページ]
SQL AnywhereHTTP Web サーバを参照する方法 [623 ページ]

1.20.1.3.2.1.2 SOAP over HTTP サービスを作成する方法

SOAP は、多くの開発環境でサポートされているデータ交換標準です。

SOAP ペイロードは、SOAP エンベロープと呼ばれる XML ドキュメントで構成されます。SOAP 要求エンベロープには、SOAP 操作 (SOAP サービス) が含まれ、適切なパラメータをすべて指定します。SOAP サービスは、要求を解析してパラメータを取得し、他のサービスと同様にストアドプロシージャまたはファンクションの呼び出しや選択を行います。SOAP サービスの表示レイヤは、DISH サービスの WSDL で指定された事前定義のフォーマットで、SOAP エンベロープ内のクライアントに結果セットのストリームを返します。

デフォルトでは、SOAP サービスのパラメータと結果セットは XmlSchema 文字列型のパラメータになります。DATATYPE ON は、入力パラメータと応答データで TRUE 型を使用することを指定します。DATATYPE を指定すると、WSDL 仕様が適宜変更され、クライアント SOAP ツールキットで適切な型のパラメータと応答オブジェクトを使用してインタフェースが生成されるようになります。

FORMAT 句は、異なる機能で特定の SOAP ツールキットを対象とするために使用します。DNET には、SOAP サービス応答を System.Data.DataSet オブジェクトとして処理する Microsoft .NET クライアントアプリケーションが備えられています。CONCRETE では、.NET や Java などのオブジェクト指向アプリケーションで、ローとカラムをパッケージ化する応答オブジェクトを生成できるようにする、より汎用的な構造が公開されます。XML からは、応答全体が XML ドキュメントとして返され、文字列として公開されます。クライアントは、XML パーサを使用して、データをさらに処理できます。CREATE SERVICE 文の FORMAT 句では、複数のクライアントアプリケーションタイプがサポートされています。

注記

DATATYPE 句は、SOAP サービスにのみ関係します (HTML でのデータ入力はありません)。FORMAT 句は、SOAP サービスまたは DISH サービスのいずれかで指定できます。SOAP サービスの FORMAT 指定は、DISH サービスの指定よりも優先されます。

例

Interactive SQL で次の文を実行して、HTTP を介した SOAP サービスを作成します。

```
CREATE SERVICE SampleSOAPService
    TYPE 'SOAP'
    DATATYPE ON
    FORMAT 'CONCRETE'
    USER DBA
    AUTHORIZATION OFF
    AS sql-statement;
```

関連情報

結果セットからの変数へのアクセス [648 ページ]
HTTP Web サービスの例 [663 ページ]
Web サービスタイプ [592 ページ]
DISH サービスを作成する方法 [597 ページ]
Simple Object Access Protocol (SOAP)

1.20.1.3.2.1.3 DISH サービスを作成する方法

DISH サービスは、SOAP サービスのグループの SOAP 終了ポイントとして機能します。また、DISH サービスでは、WSDL (Web Services Description Language) ドキュメントも自動的に構築され、WSDL で記述される SOAP サービスとのデータ交換に必要なインタフェースを SOAP クライアントツールキットで生成できるようになります。

HTTP を介した SOAP サービスの現在のワーキングセットは常に公開されるため、SOAP サービスの追加と削除には、DISH サービスの管理は必要ありません。

例

Interactive SQL で次の SQL 文を実行して、HTTP Web サーバでサンプルの SOAP サービスと DISH サービスを作成します。

```
CREATE SERVICE "Samples/TestSoapOp"
    TYPE 'SOAP'
    DATATYPE ON
    USER DBA
    AUTHORIZATION OFF
    AS CALL sp_echo(:i, :f, :s);
CREATE PROCEDURE sp_echo(i INTEGER, f REAL, s LONG VARCHAR)
RESULT( ret_i INTEGER, ret_f REAL, ret_s LONG VARCHAR )
BEGIN
    SELECT i, f, s;
END;
CREATE SERVICE "dnet_endpoint"
    TYPE 'DISH'
    GROUP "Samples"
    FORMAT 'DNET';
```

最初の CREATE SERVICE 文は、Samples/TestSoapOp という新しい SOAP サービスを作成します。

2 番目の CREATE SERVICE 文は、dnet_endpoint という新しい DISH サービスを作成します。GROUP 句の Samples 部分は、公開する SOAP サービスのグループを識別します。DISH サービスによって生成される WSDL ドキュメントを表示できます。SQL Anywhere HTTP Web サーバがコンピュータで実行中の場合、`http://localhost:port-number/dnet_endpoint` URL を使用してサービスにアクセスできます。`port-number` は、サーバが実行されているポート番号です。

この例では、SOAP 応答フォーマットを示す FORMAT 句は、SOAP サービスに含まれていません。したがって、SOAP 応答フォーマットは DISH サービスで指定され、SOAP サービスの FORMAT 句は上書きされません。この機能により、異なる機能のサービスを各 DISH 終了ポイントが SOAP クライアントに提供する、同種の DISH サービスを作成できます。

同種の DISH サービスの作成

SOAP サービス定義が DISH サービスに対する FORMAT 句の指定と異なる場合、フォーマットを定義する DISH サービス内で SOAP サービスのセットをグループ化することができます。このようにすると、異なる FORMAT 指定を使用して、複数の DISH サービスで同じグループの SOAP サービスを公開できます。TestSoapOp の例を展開すると、次の SQL 文を使用して、java_endpoint という別の DISH サービスを作成できます。

```
CREATE SERVICE "java_endpoint"
    TYPE 'DISH'
    GROUP "Samples"
    FORMAT 'CONCRETE';
```

この例では、SOAP クライアントが java_endpoint DISH サービスを介して TestSoapOp 操作用の Web サービス要求を行うと、SOAP クライアントは TestSoapOp_Dataset という応答オブジェクトを受信します。WSDL を調べて、dnet_endpoint と java_endpoint の違いを比較できます。この手法を使用して、特定の SOAP クライアントツールキットの条件を満たす SOAP 終了ポイントをすばやく構築できます。

関連情報

Web サービスタイプ [592 ページ]
HTTP Web サービスの例 [663 ページ]
SOAP over HTTP サービスを作成する方法 [596 ページ]
SQL AnywhereHTTP Web サーバを参照する方法 [623 ページ]

1.20.1.3.2.2 Web サービスを削除する方法

Web サービスを削除すると、そのサービスに対して作成された後続の要求によって `404 Not Found` HTTP ステータスメッセージが返されます。**`root`** Web サービスが存在する場合、すべての未解決の要求 (意図的なまたは意図的でない要求) が処理されます。

例

次の SQL 文を実行して、SampleWebService という名前の Web サービスを削除します。

```
DROP SERVICE SampleWebService;
```

関連情報

root Web サービスを作成し、カスタマイズする方法 [600 ページ]

1.20.1.3.2.3 Web サービスにコメントを付ける方法

Web サービスのドキュメントを提供するには、COMMENT ON SERVICE 文を使用する必要があります。

コメントは、`statement` 句を NULL に設定すると削除できます。

例

たとえば、次の SQL 文を実行して、SampleWebService という Web サービスに新しいコメントを作成します。

```
COMMENT ON SERVICE SampleWebService
    IS "This is a comment on my web service.";
```

1.20.1.3.2.4 root Web サービスを作成し、カスタマイズする方法

どの Web サービス要求とも一致しない HTTP クライアント要求は、**root** Web サービスが定義されていると、**root** Web サービスによって処理されます。

root Web サービスには、任意の HTTP 要求 (アプリケーションを構築する時点で URL が判明している必要はない) や、認識されない要求を処理する、簡単で柔軟な方法が備えられています。

例

この例では、データベース内のテーブルに格納されている **root** Web サービスを使用して、Web ブラウザや他の HTTP クライアントにコンテンツを提供する方法を示します。ポート 80 で受信するローカルの HTTP Web サーバを単一のデータベースで起動していることが前提となります。すべてのスクリプトは、Web サーバで実行されます。

Interactive SQL を介してデータベースサーバに接続し、次の SQL 文を実行して、クライアントで指定された url ホスト変数を PageContent というプロシージャに渡す **root** Web サービスを作成します。

```
CREATE SERVICE root
    TYPE 'RAW'
    AUTHORIZATION OFF
    SECURE OFF
    URL ON
    USER DBA
    AS CALL PageContent(:url);
```

URL ON 部分は、URL という名前の HTTP 変数からフルパスコンポーネントにアクセスできることを指定します。

次の SQL 文を実行して、ページコンテンツの格納に使用するテーブルを作成します。この例では、ページコンテンツは、その URL、MIME タイプ、コンテンツ自体によって定義されます。

```
CREATE TABLE Page_Content (
    url          VARCHAR(1024) NOT NULL PRIMARY KEY,
    content_type VARCHAR(128)  NOT NULL,
    image        LONG VARCHAR  NOT NULL
);
```

次の SQL 文を実行して、テーブルを移植します。この例では、index.html ページが要求されたときに HTTP クライアントに提供するコンテンツを定義します。

```
INSERT INTO Page_Content
VALUES(
    'index.html',
    'text/html',
    '<html><body><h1>Hello World</h1></body></html>'
);
COMMIT;
```

次の SQL 文を実行して、**root** Web サービスに渡される url ホスト変数を受け入れる PageContent プロシージャを実装します。

```
CREATE PROCEDURE PageContent(IN @url LONG VARCHAR)
RESULT ( html_doc LONG VARCHAR )
BEGIN
    DECLARE @status CHAR(3);
    DECLARE @type   VARCHAR(128);
    DECLARE @image  LONG VARCHAR;
    SELECT content_type, image INTO @type, @image
        FROM Page_Content
```

```
        WHERE url = @url;
    IF @image is NULL THEN
        SET @status = '404';
        SET @type = 'text/html';
        SET @image =  '<html><body><h1>404 - Page Not Found</h1>'
            || '<p>There is no content located at the URL "'
            || html_encode( @url ) || '" on this server.<p>'
            || '</body></html>';
    ELSE
        SET @status = '200';
    END IF;
    CALL sa_set_http_header( '@HttpStatus', @status );
    CALL sa_set_http_header( 'Content-Type', @type );
    SELECT @image;
END;
```

HTTP サーバへの要求が定義済みの他の Web サービスの URL と一致しない場合、**root** Web サービスは PageContent プロシージャを呼び出します。クライアントで指定された URL が Page_Content テーブルの url と一致するかどうかが、プロシージャによってチェックされます。SELECT 文は、応答をクライアントに送信します。クライアントで指定された URL がテーブルで検出されないと、汎用 `404 - Page Not Found` HTML ページが作成され、クライアントに送信されます。

ブラウザの中には、ブラウザ自体のページで 404 ステータスに対応するものもあるので、汎用ページが表示される保証はありません。

エラーメッセージでは、HTML_ENCODE 関数を使用して、クライアントで指定された URL 内の特殊文字がエンコードされます。

@HttpStatus ヘッダは、要求で返されるステータスコードを設定するために使用します。404 ステータスは Not Found エラーを示し、200 ステータスは OK を示します。'Content-Type' ヘッダは、要求で返されるコンテンツタイプを設定するために使用します。この例では、index.html ページのコンテンツ (MIME) タイプは text/html です。

関連情報

Web ページをカスタマイズする方法 [604 ページ]
SQL AnywhereHTTP Web サーバを参照する方法 [623 ページ]

1.20.1.3.2.5 Web サービスの SQL 文

SQL 文は、データベースに Web サービスを実装するために使用されます。

次の SQL 文を使用できます。

Web サービス関連の SQL 文	説明
CREATE SERVICE	HTTP または DISH サービス経由の新しい HTTP Web サービスまたは新しい SOAP を作成します。

Web サービス関連の SQL 文	説明
ALTER SERVICE	HTTP または DISH サービス経由の k 既存の HTTP Web サービスまたは SOAP を変更します。

COMMENT	データベースオブジェクトに対するコメントをシステムテーブルに格納します。
	Web サービスにコメントを付けるには、次の構文を使用します。
	`COMMENT ON SERVICE 'web-service-name'`
	`IS 'your comments'`

DROP SERVICE	Web サービスを削除します。

1.20.1.3.3 Web サービスの接続プーリング

Web サービスを公開する各データベースでは、データベース接続のプールにアクセスできます。特定の USER 句で定義されるすべてのサービスで同じ接続プールグループを共有するように、プールはユーザ名ごとにグループ化されます。

クエリを初めて実行するサービス要求は、最適化フェーズを経由して実行プランを確立する必要があります。プランのキャッシュと再使用が可能な場合、以降の実行では最適化フェーズを省略できます。HTTP 接続プーリングでは、プランを可能なかぎり再使用することによって、データベース接続におけるプランのキャッシュが活用されます。各サービスでは、再使用を最適化するために独自の接続リストが管理されます。ただし、負荷のピーク時には、サービスは同じユーザグループの接続のうち使用率の最も低い接続を横取りすることがあります。

いくつかのサービスの実行に対してパフォーマンスを最適化できるキャッシュされたプランが、一定期間に特定の接続で取得されることがあります。キャッシュされたプランの数は、max_plans_cached オプションで制御できます。

接続プールでは、特定のサービスに対する HTTP 要求が、データベース接続をプールから取得しようとします。HTTP セッションと異なり、プールされた接続は、接続スコープ変数やテンポラリテーブルなどの接続スコープ環境をリセットすることによって削除されます。

HTTP 接続プール内にプールされたデータベース接続は、ライセンス目的に使用される接続としてはカウントされません。プールから取得された接続は、ライセンスされた接続としてカウントされます。プールから接続を取得するときに HTTP 要求がライセンスの制限を越えると、`503 Service Temporarily Unavailable` ステータスが返されます。

接続プールは、Web サービスが AUTHORIZATION OFF を指定して定義されている場合にのみ利用できます。

プール内のデータベース接続は、データベースと接続のオプションが変更されても更新されません。

関連情報

Web サービス接続プロパティ [622 ページ]

1.20.1.4 HTTP Web サーバで Web サービスアプリケーションを開発する方法

カスタマイズされた Web ページは、Web サービスから呼び出されるストアドプロシージャによって作成できます。非常に基本的な Web ページの例を書きに示します。

```
CREATE SERVICE ROOT
    TYPE 'RAW'
    AUTHORIZATION OFF
    USER DBA
    AS CALL MyHomePage;
CREATE OR REPLACE PROCEDURE MyHomePage()
BEGIN
  CALL sa_set_http_header ( 'Content-Type', 'text/html' );
  SELECT '<html><body><p>Welcome to my web page</p></body></html>';
END
```

Web サーバのルート Web ページ (http://localhost:8082/ など) をブラウズすると、挨拶として「ようこそ」メッセージが表示されます。

動的な Web ページコンテンツを作成するストアドプロシージャを使用し、追加のサービスを作成することで、より洗練された Web ページを開発できます。

Web サービスアプリケーションの詳細な例については、*%SQLANYSAMP17%*¥SQLAnywhere¥HTTP ディレクトリを参照してください。

このセクションの内容:

Web ページをカスタマイズする方法 [604 ページ]
Web ページをカスタマイズするには、まず、HTTP Web サーバから呼び出される Web サービスのフォーマットを評価する必要があります。

クライアント提供の HTTP 変数とヘッダにアクセスする方法 [606 ページ]
HTTP クライアント要求の変数とヘッダには、Web サービスからアクセスできます。

クライアント提供の SOAP 要求ヘッダにアクセスする方法 [610 ページ]
SOAP 要求のヘッダは、NEXT_SOAP_HEADER 関数と SOAP_HEADER 関数を組み合わせて使用することによって取得できます。

HTTP サーバでの HTTP セッションの管理 [612 ページ]
Web アプリケーションでは、セッションをさまざまな方法でサポートできます。

文字セットの変換に関する考慮事項 [621 ページ]
デフォルトで、文字セット変換はテキストからなる出力結果セットで自動的に実行されます。バイナリオブジェクトなどの他のタイプの結果セットは、影響を受けません。

クロスサイトスクリプティングについての考慮事項 [621 ページ]
Web アプリケーションを開発する場合、そのアプリケーションがクロスサイトスクリプティング (XSS) に対して脆弱でないことを確認します。このタイプの脆弱性は、攻撃者が Web ページにスクリプトを注入しようとすると発生します。

Web サービスシステムプロシージャ [621 ページ]
次のシステムプロシージャは、Web サービスの操作時に使用します。

Web サービス関数 [622 ページ]
Web サービス関数は、Web サービス内の HTTP 要求と SOAP 要求の処理を支援します。

Web サービス接続プロパティ [622 ページ]
Web サービス接続プロパティには、CONNECTION_PROPERTY 関数を使用してアクセスします。

Web サービスオプション [623 ページ]
Web サービスオプションは、HTTP サーバ動作をさまざまな面から制御します。

関連情報

HTTP Web サーバを起動する方法 [589 ページ]
Web サービスとは? [592 ページ]

1.20.1.4.1 Web ページをカスタマイズする方法

Web ページをカスタマイズするには、まず、HTTP Web サーバから呼び出される Web サービスのフォーマットを評価する必要があります。

たとえば、Web サービスで HTML タイプを指定すると、Web ページは HTML でフォーマットされます。

RAW Web サービスタイプは、HTML や XML などの必要なマークアップを Web サービスのプロシージャと関数によって明示的に生成する必要があるため、最もカスタマイズしやすいタイプです。RAW タイプを使用するときに Web ページをカスタマイズするには、次のタスクを実行します。

- 呼び出されるストアドプロシージャで、HTTP コンテンツタイプヘッダフィールドを、text/html などの適切な MIME タイプに設定します。
- 呼び出されるストアドプロシージャから Web ページ出力を生成するときに、MIME タイプに適切なマークアップを適用します。

例

次の例は、RAW タイプを指定して、新しい Web サービスを作成する方法を示しています。

```
CREATE SERVICE WebServiceName
    TYPE 'RAW'
    AUTHORIZATION OFF
    URL ON
    USER DBA
    AS CALL HomePage( :url );
```

この例では、Web サービスが HomePage ストアドプロシージャを呼び出します。これには、URL の PATH コンポーネントを受信する 1 つの URL パラメータを定義する必要があります。

コンテンツタイプヘッダフィールドの設定

Web ブラウザでコンテンツを正しく表示するには、sa_set_http_header システムプロシージャを使用して HTTP コンテンツタイプヘッダを定義します。

次の例は、sa_set_http_header システムプロシージャで text/html MIME タイプを使用して、Web ページ出力を HTML でフォーマットする方法を示しています。

```
CREATE PROCEDURE HomePage (IN url LONG VARCHAR)
    RESULT (html_doc XML)
    BEGIN
        CALL sa_set_http_header ( 'Content-Type', 'text/html' );
        -- Your SQL code goes here.
        ...
    END
```

MIME タイプのタグ付け規則の適用

ストアドプロシージャのコンテンツタイプヘッダで指定された MIME タイプのタグ付け規則を適用する必要があります。タグを作成するための関数がいくつか提供されています。

次の例は、XMLCONCAT 関数と XMLELEMENT 関数を使用して HTML コンテンツを作成する方法を示しています。sa_set_http_header システムプロシージャを使用して、コンテンツタイプヘッダを text/html MIME タイプに設定していることが前提となります。

```
XMLCONCAT(
   CAST('<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">' AS XML),
   XMLELEMENT(
        'HTML',
        XMLELEMENT(
            'HEAD',
            XMLELEMENT('TITLE', 'My Home Page')
        ),
        XMLELEMENT(
            'BODY',
            XMLELEMENT('H1', 'My home on the web'),
            XMLELEMENT('P', 'Thank you for visiting my web site!')
        )
    )
)
```

データ型が XML ではない場合、要素内容は必ずエスケープされるので、前述の例は CAST 関数を使用します。それ以外の場合、特殊文字はエスケープされます (例: < には <)。

関連情報

Web サービスタイプ [592 ページ]
root Web サービスを作成し、カスタマイズする方法 [600 ページ]
Web サービス関数 [622 ページ]
SQL AnywhereHTTP Web サーバを参照する方法 [623 ページ]

1.20.1.4.2 クライアント提供の HTTP 変数とヘッダにアクセスする方法

HTTP クライアント要求の変数とヘッダには、Web サービスからアクセスできます。

- Web サービス文の宣言を使用して、ストアド関数またはストアドプロシージャに URL 変数をパラメータとして渡すことができます。
- HTTP クライアント要求の変数とヘッダにアクセスするには、ストアド関数またはストアドプロシージャで、HTTP_VARIABLE、NEXT_HTTP_VARIABLE、HTTP_HEADER、NEXT_HTTP_HEADER の各関数を呼び出すことができます。

このセクションの内容:

ホストパラメータを使用して HTTP 変数にアクセスする方法 [606 ページ]
クライアント指定の変数の名前をパラメータとして関数またはプロシージャ呼び出しに渡すときに、その変数の値を取得できます。

Web サービス関数を使用して HTTP 変数とヘッダにアクセスする方法 [607 ページ]
HTTP_VARIABLE、NEXT_HTTP_VARIABLE、HTTP_HEADER、NEXT_HTTP_HEADER の各関数は、Web サービスクライアントが指定した変数とヘッダに対して反復処理を実行するために使用できます。

1.20.1.4.2.1 ホストパラメータを使用して HTTP 変数にアクセスする方法

クライアント指定の変数の名前をパラメータとして関数またはプロシージャ呼び出しに渡すときに、その変数の値を取得できます。

例

次の例は、ShowTable という Web サービスで使用されるパラメータにアクセスする方法を示しています。

```
CREATE SERVICE ShowTable
    TYPE 'RAW'
    AUTHORIZATION ON
    AS CALL ShowTable( :user_name, :table_name );
CREATE PROCEDURE ShowTable(IN username VARCHAR(128), IN tablename VARCHAR(128))
BEGIN
    -- write SQL code utilizing the username and tablename variables here.
END;
```

サービスのホストパラメータは、プロシージャパラメータの宣言順にマッピングされます。上記の例では、user_name と table_name の各ホストパラメータが、username と tblname の各パラメータにそれぞれマッピングされます。

このサービスにアクセスする URL の例を次に示します。

```
http://localhost:8082/ShowTable?user_name=Groupo&table_name=Customers
```

関連情報

SQL AnywhereHTTP Web サーバを参照する方法 [623 ページ]

1.20.1.4.2.2 Web サービス関数を使用して HTTP 変数とヘッダにアクセスする方法

HTTP_VARIABLE、NEXT_HTTP_VARIABLE、HTTP_HEADER、NEXT_HTTP_HEADER の各関数は、Web サービスクライアントが指定した変数とヘッダに対して反復処理を実行するために使用できます。

HTTP_VARIABLE と HTTP_NEXT_VARIABLE を使用した変数のアクセス

ストアドプロシージャ内で NEXT_HTTP_VARIABLE 関数と HTTP_VARIABLE 関数を使用して、クライアントが指定したすべての変数に対して反復処理を実行できます。

HTTP_VARIABLE 関数では、変数名の値を取得できます。

NEXT_HTTP_VARIABLE 関数を使用すると、クライアントから送信されたすべての変数に対する反復処理を実行できます。最初の変数名を取得するために関数を初めて呼び出す場合は、NULL 値を渡します。返された変数名を HTTP_VARIABLE 関数呼び出しのパラメータとして使用して、変数の値を取得します。前の変数名を next_http_variable 呼び出しで渡すと、次の変数名が取得されます。最後の変数名が渡されると、NULL が返されます。

変数名に対する反復処理を行うことによって各変数名が確実に 1 回のみ返されますが、変数名の順序はクライアント要求での順序とは異なる場合があります。

次の例は、ShowDetail サービスにアクセスするクライアント要求で指定されたパラメータから値を検索する、HTTP_VARIABLE 関数の使用方法を示しています。

```
CREATE OR REPLACE SERVICE ShowDetail
    TYPE 'HTML'
    URL PATH OFF
    AUTHORIZATION OFF
    USER DBA
    AS CALL ShowDetail();
CREATE OR REPLACE PROCEDURE ShowDetail()
BEGIN
    DECLARE v_customer_id LONG VARCHAR;
    DECLARE v_product_id LONG VARCHAR;
    SET v_customer_id = HTTP_VARIABLE( 'customer_id' );
    SET v_product_id = HTTP_VARIABLE( 'product_id' );
    CALL ShowSalesOrderDetail( v_customer_id, v_product_id );
END;
```

このサービスにアクセスする URL の例を次に示します。

```
http://localhost:8082/ShowDetail?customer_id=103&product_id=300
```

次の例は、**image** 変数に関連するヘッダフィールド値から 3 つの属性を検索する方法を示しています。

```
SET v_name = HTTP_VARIABLE( 'image', NULL, 'Content-Disposition' );
SET v_type = HTTP_VARIABLE( 'image', NULL, 'Content-Type' );
SET v_image = HTTP_VARIABLE( 'image', NULL, '@BINARY' );
```

2 番目のパラメータに整数を指定すると、追加の値を検索できます。3 番目のパラメータでは、ヘッダフィールド値をマルチパート要求から検索できます。ヘッダフィールド名を指定してこの値を検索します。

詳細な例については、*%SQLANY17%*¥SQLAnywhere¥HTTP¥gallery.sql プロジェクトを参照してください。

HTTP_HEADER と NEXT_HTTP_HEADER を使用したヘッダへのアクセス

NEXT_HTTP_HEADER 関数と HTTP_HEADER 関数を使用して、HTTP 要求のヘッダを要求から取得できます。

HTTP_HEADER 関数は、指定された HTTP ヘッダフィールドの値を返します。

NEXT_HTTP_HEADER 関数は、HTTP ヘッダに対して反復され、次の HTTP ヘッダ名を返します。NULL を指定してこの関数を呼び出すと、最初のヘッダ名が返されます。後続のヘッダは、前のヘッダの名前を関数に渡すことによって取得されます。最後のヘッダ名が呼び出されると、NULL が返されます。

次の表に、いくつかの共通の HTTP 要求ヘッダとサンプル値を示します。

ヘッダ名	ヘッダ値
Accept	image/gif、image/x-xbitmap、image/jpeg、image/pjpeg、application/x-shockwave-flash、application/vnd.ms-excel、application/vnd.ms-powerpoint、application/msword、*/*
Accept-Language	en-us
Accept-Charset	utf-8、iso-8859-5;q=0.8
Accept-Encoding	gzip, deflate
User-Agent	Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.2; WOW64; SV1; .NET CLR 2.0.50727)
Host	localhost:8080
Connection	Keep-Alive

次の表は、特殊なヘッダとサンプル値を示します。

ヘッダ名	ヘッダ値
@HttpMethod	GET
@HttpURI	/demo/ShowHTTPHeaders
@HttpVersion	HTTP/1.1
@HttpQueryString	id=-123&version=109&lang=en

処理中の要求のステータスコードを設定するには、特別なヘッダ @HttpStatus を使用します。

次の例は、ヘッダの名前と値を HTML テーブルにフォーマットする方法を示しています。

ShowHTTPHeaders Web サービスを作成します。

```
CREATE OR REPLACE SERVICE ShowHTTPHeaders
    TYPE 'RAW'
    AUTHORIZATION OFF
    USER DBA
    AS CALL HTTPHeaderExample();
```

NEXT_HTTP_HEADER 関数を使用する HTTPHeaderExample プロシージャを作成してヘッダ名を取得してから、HTTP_HEADER 関数を使用してヘッダのすべてのインスタンスの値を取得します。

```
CREATE OR REPLACE PROCEDURE HTTPHeaderExample()
RESULT ( html_string LONG VARCHAR )
BEGIN
```

```
    DECLARE instance INTEGER;
    DECLARE header_name LONG VARCHAR;
    DECLARE header_value LONG VARCHAR;
    DECLARE header_query LONG VARCHAR;
    DECLARE table_rows XML;
    SET table_rows = NULL;
    SET header_name = NULL;
  header_loop:
    LOOP
        SET header_name = NEXT_HTTP_HEADER( header_name );
        IF header_name IS NULL THEN
            LEAVE header_loop
        END IF;
        SET instance = 0;
      value_loop:
        LOOP
            SET instance = instance + 1;
            SET header_value = HTTP_HEADER( header_name, instance );
            IF header_value IS NULL THEN
                LEAVE value_loop;
            END IF;
            -- Format header name, value, and instance number into an HTML table row
            SET table_rows = table_rows ||
                XMLELEMENT( name "tr",
                    XMLATTRIBUTES( 'left' AS "align",
                                   'top' AS "valign" ),
                    XMLELEMENT( name "td", header_name ),
                    XMLELEMENT( name "td", header_value ),
                    XMLELEMENT( name "td", instance ) );
        END LOOP;
    END LOOP;
    CALL sa_set_http_header( 'Content-Type', 'text/html' );
    SELECT XMLELEMENT( name "table",
                XMLATTRIBUTES( '' AS "BORDER",
                             '10' AS "CELLPADDING",
                              '0' AS "CELLSPACING" ),
                XMLELEMENT( name "th",
                    XMLATTRIBUTES( 'left' AS "align",
                                   'top'  AS "valign" ),
                                   'Header Name' ),
                XMLELEMENT( name "th",
                    XMLATTRIBUTES( 'left' AS "align",
                                   'top'  AS "valign" ),
                                   'Header Value' ),
                XMLELEMENT( name "th",
                    XMLATTRIBUTES( 'left' AS "align",
                                   'top'  AS "valign" ),
                                   'Header Instance' ),
                table_rows );
END;
```

Web ブラウザで ShowHTTPHeaders サービスにアクセスして、HTML テーブルに配置された要求ヘッダを確認します。このサービスにアクセスする URL の例を次に示します。

```
http://localhost:8082/ShowHTTPHeaders?customer_id=103&product_id=300
```

関連情報

Web サービスへの変数の指定 [645 ページ]

1.20.1.4.3 クライアント提供の SOAP 要求ヘッダにアクセスする方法

SOAP 要求のヘッダは、NEXT_SOAP_HEADER 関数と SOAP_HEADER 関数を組み合わせて使用することによって取得できます。

NEXT_SOAP_HEADER 関数は、SOAP 要求エンベロープに含まれる SOAP ヘッダに対して反復され、次の SOAP ヘッダ名を返します。NULL を指定して呼び出すと、最初のヘッダの名前が返されます。後続のヘッダは、NEXT_SOAP_HEADER 関数に前のヘッダの名前を渡すことによって取得されます。最後のヘッダの名前を指定して呼び出すと、NULL が返されます。

次の例は、SOAP ヘッダの取得を示しています。

```
SET hd_key = NEXT_SOAP_HEADER( hd_key );
    IF hd_key IS NULL THEN
      -- no more header entries
      LEAVE header_loop;
    END IF;
```

この関数を繰り返し呼び出すと、すべてのヘッダフィールドが一度だけ返されます。ただし、必ずしも SOAP 要求での表示順に表示されるとはかぎりません。

SOAP_HEADER 関数は、名前付きの SOAP ヘッダフィールドの値を返します。SOAP サービスから呼び出されていない場合は NULL を返します。Web サービスを介して SOAP 要求を処理する場合に使用します。指定したフィールド名のヘッダが存在しない場合、戻り値は NULL です。

この例は、Authentication という SOAP ヘッダを探します。このヘッダが見つかると、SOAP ヘッダ全体の値を抽出し、さらに @namespace 属性と mustUnderstand 属性の値を抽出します。SOAP ヘッダの値は、次の XML 文字列のようになります。

```
<Authentication xmlns="CustomerOrderURN" mustUnderstand="1">
  <userName pwd="none">
    <first>John</first>
    <last>Smith</last>
  </userName>
</Authentication>
```

このヘッダの場合、@namespace 属性値は CustomerOrderURN になります。

また、mustUnderstand 属性値は 1 になります。

この XML 文字列の内部構造を、XPath 文字列に **/*:Authentication/*:userName** を設定した OPENXML 演算子を使用して解析します。

```
SELECT * FROM OPENXML( hd_entry, xpath )
    WITH ( pwd LONG VARCHAR '@*:pwd',
        first_name LONG VARCHAR '*:first/text()',
        last_name LONG VARCHAR '*:last/text()' );
```

上記のサンプル SOAP ヘッダ値を使用した場合、SELECT 文は次のような結果セットを作成します。

pwd	first_name	last_name
none	John	Smith

この結果セットに対してカーソルが宣言され、3 つのカラム値が 3 つの変数にフェッチされます。この時点で、Web サービスに渡された関連性のある情報すべてが手中にあります。

例

次の例は、パラメータを含む SOAP 要求と SOAP ヘッダが Web サーバでどのように処理されるかを示しています。この例では、次の 2 つのパラメータを取る addItem SOAP 操作を実行します。int 型の *amount* および文字列型の *item*。sp_addItems プロシージャは、ユーザの姓と名を抽出する Authentication という SOAP ヘッダを処理します。値は sa_set_soap_header システムプロシージャを通じて SOAP 応答検証ヘッダに提供されます。応答は、次の 3 つのカラムの結果です。INT 型の *quantity*、LONG VARCHAR 型の *item*、LONG VARCHAR 型の *status*。

```
// create the SOAP service
CREATE SERVICE addItems
    TYPE 'SOAP'
    FORMAT 'CONCRETE'
    AUTHORIZATION OFF
    USER DBA
    AS CALL sp_addItems( :amount, :item );
// create SOAP endpoint for related services
CREATE SERVICE itemStore
    TYPE 'DISH'
    AUTHORIZATION OFF
    USER DBA;
// create the procedure that will process the SOAP requests for the addItems
service
CREATE PROCEDURE sp_addItems(count INT, item LONG VARCHAR)
RESULT(quantity INT, item LONG VARCHAR, status LONG VARCHAR)
BEGIN
    DECLARE hd_key LONG VARCHAR;
    DECLARE hd_entry LONG VARCHAR;
    DECLARE pwd LONG VARCHAR;
    DECLARE first_name LONG VARCHAR;
    DECLARE last_name LONG VARCHAR;
    DECLARE xpath LONG VARCHAR;
    DECLARE authinfo LONG VARCHAR;
    DECLARE namespace LONG VARCHAR;
    DECLARE mustUnderstand LONG VARCHAR;

  header_loop:
    LOOP
        SET hd_key = next_soap_header( hd_key );
        IF hd_key IS NULL THEN
            // no more header entries.
            leave header_loop;
        END IF;
        IF hd_key = 'Authentication' THEN
            SET hd_entry = soap_header( hd_key );
            SET xpath = '/*:' || hd_key || '/*:userName';
            SET namespace = soap_header( hd_key, 1, '@namespace' );
            SET mustUnderstand = soap_header( hd_key, 1, 'mustUnderstand' );
            BEGIN
                // parse for the pieces that you are interested in
                DECLARE crsr CURSOR FOR SELECT * FROM
                    OPENXML( hd_entry, xpath )
                        WITH ( pwd LONG VARCHAR '@*:pwd',
                               first_name LONG VARCHAR '*:first/text()',
                               last_name LONG VARCHAR '*:last/text()');
                OPEN crsr;
                FETCH crsr INTO pwd, first_name, last_name;
                CLOSE crsr;
            END;
            // build a response header, based on the pieces from the request
header
            SET authinfo = XMLELEMENT( 'Validation',
                XMLATTRIBUTES(
                    namespace as xmlns,
                    mustUnderstand as mustUnderstand ),
                    XMLELEMENT( 'first', first_name ),
```

```
                    XMLELEMENT( 'last', last_name ) );
            CALL sa_set_soap_header( 'authinfo', authinfo);
        END IF;
    END LOOP header_loop;
    //  code to validate user/session and check item goes here...
    SELECT count, item, 'available';
END;
```

関連情報

SOAP 要求ヘッダの管理 [640 ページ]

1.20.1.4.4 HTTP サーバでの HTTP セッションの管理

Web アプリケーションでは、セッションをさまざまな方法でサポートできます。

HTML フォーム内の非表示フィールドを使用して、複数の要求にまたがってクライアント/サーバデータを保持できます。別の方法として、AJAX 対応のクライアント側 JavaScript などの Web 2.0 手法では、クライアントステータスに基づいて非同期の HTTP 要求を行うことができます。セッション化された HTTP 要求を排他的に使用するために、データベース接続を保持することができます。

HTTP セッション内で作成または変更された接続スコープ変数とテンポラリテーブルには、特定の SessionID を指定する後続の HTTP 要求からアクセスできます。SessionID は、GET または POST HTTP 要求メソッドで指定するか、HTTP cookie ヘッダ内で指定できます。サーバは、SessionID 変数を指定した HTTP 要求を受信すると、一致するコンテキストをセッションレポジトリから検索します。セッションが検出されると、そのデータベース接続を使用して要求を処理します。セッションが使用中の場合は、HTTP 要求をキューに追加し、セッションが解放された時点で HTTP 要求をアクティブにします。

セッション ID の作成、削除、変更には、sa_set_http_option を使用できます。

HTTP セッションには、セッション条件を管理するために特殊な処理が必要です。1 つの SessionID で使用できるデータベース接続は 1 つしかないため、特定の SessionID に対する連続したクライアント要求は、サーバによってシリアル化されます。特定の SessionID に対して、16 個までの要求をキューに追加できます。セッションキューが満杯になると、その SessionID に対する後続の要求は `503 Service Unavailable` ステータスで拒否されます。

SessionID を初めて作成すると、SessionID がシステムによって即時に登録されます。SessionID を変更または削除する後続の要求は、HTTP 要求が終了した時点でのみ適用されます。この方法を使用すると、要求の処理がロールバックで終了する場合や、アプリケーションで SessionID を削除してリセットする場合に、動作の一貫性を維持できます。

HTTP 要求によって SessionID が変更されると、現在のセッションは削除され、保留中のセッションに置き換えられます。セッションによってキャッシュされたデータベース接続は、新しいセッションコンテキストに効率的に移動され、テンポラリテーブルや作成された変数などのステータスデータはすべて保持されます。

HTTP セッションの詳しい使用例については、*%SQLANYSAMP17%*`¥SQLAnywhere¥HTTP¥session.sql` を参照してください。

注記

各クライアントアプリケーション接続ではライセンスシートが保持されるため、未処理の接続数を最小限に抑えるために、古いセッションを削除し、適切なタイムアウトを設定する必要があります。HTTP セッションに関連する接続では、接続期間を通じてサーバデータベースでライセンスシートの保持が管理されます。

このセクションの内容:

HTTP セッションを作成する方法 [613 ページ]
セッションは、sa_set_http_option システムプロシージャで SessionID オプションを使用して作成できます。セッション ID の定義には、NULL 以外の任意の文字列を使用できます。

非アクティブな HTTP セッションを検出する方法 [617 ページ]
SessionCreateTime および SessionLastTime 接続プロパティを使用して、現在の接続がセッションコンテキスト内にあるかどうかを確認できます。いずれかの接続プロパティクエリから空の文字列が返された場合、HTTP 要求はセッションコンテキスト内で実行されていません。

HTTP セッションを削除するかセッション ID を変更する方法 [618 ページ]
HTTP セッションを削除するか、セッション ID を変更するには、いくつかの方法があります。

HTTP セッションの管理 [619 ページ]
HTTP 要求によって作成されたセッションはすぐにインスタンス化されるため、そのセッションコンテキストを必要とする後続の HTTP 要求はセッションによってキューに追加されます。

HTTP セッションのエラーコード [620 ページ]
HTTP セッションの管理中には、いくつかの共通のエラーが発生します。

1.20.1.4.4.1 HTTP セッションを作成する方法

セッションは、sa_set_http_option システムプロシージャで SessionID オプションを使用して作成できます。セッション ID の定義には、NULL 以外の任意の文字列を使用できます。

例

次の例では、HTTP Web サービスから呼び出したユーザ定義 SQL 関数内で一意のセッション ID を作成します。

```
CREATE OR REPLACE FUNCTION create_session()
RETURNS LONG VARCHAR
BEGIN
    DECLARE session_id LONG VARCHAR;
    DECLARE tm TIMESTAMP;
    SET tm = NOW(*);
    SET session_id = 'session_' || CONNECTION_PROPERTY('Number') || '_'
        CONVERT( VARCHAR, SECONDS(tm) * 1000 + DATEPART( MILLISECOND, tm ) );
    CALL sa_set_http_option( 'SessionID', session_id );
    SELECT CONNECTION_PROPERTY( 'SessionID' ) INTO session_id;
    RETURN( session_id );
END
```

別の HTTP 要求がセッション ID を所有していると、sa_set_http_option システムプロシージャからエラーが返されるため、一意性が保証されます。

CONNECTION_PROPERTY('SessionID') を使用すると、sa_set_http_option システムプロシージャでセッション ID 値セットを取得できます。セッション ID が接続用に定義されていない場合、結果は空の文字列です (これはセッションレス接続であることを示します)。

この関数で返される、空ではないセッション ID 文字列は、URL でも cookie でも使用できます。

このセクションの内容:

URL を使用してセッションを管理する方法 [614 ページ]
URL セッションステータス管理システムでは、クライアントアプリケーションまたは Web ブラウザによって URL 内に SessionID=`value` を含めることでセッション ID が指定されます。

cookie を使用してセッションを管理する方法 [616 ページ]
cookie セッションステータス管理システムでは、クライアントアプリケーションまたは Web ブラウザにより、URL ではなく HTTP cookie ヘッダ内でセッション ID が指定されます。

1.20.1.4.4.1.1 URL を使用してセッションを管理する方法

URL セッションステータス管理システムでは、クライアントアプリケーションまたは Web ブラウザによって URL 内に SessionID=`value` を含めることでセッション ID が指定されます。

セッションステート管理は、URL でも cookie でも実行できます。HTTP セッション情報は、GET 要求の URL または POST (x-www-form-urlencoded) 要求の本文に格納することも、HTTP cookie に格納することもできます。

次の URL は、URL を使用したセッション管理を示します。キーワード SessionID を含んでおり、セッション識別子 XXX を指定します。

```
http://localhost/mysession?SessionID=XXX
```

サーバは、SessionID 変数を指定した HTTP 要求を受信すると、一致するコンテキストをセッションレポジトリから検索します。セッションが検出されると、そのデータベース接続を使用して要求を処理します。

識別子 XXX のセッションが存在しない場合、要求は標準のセッションレス要求として (SessionID=XXX を指定しないかのように) 処理されます。

例

次の例は、セッションの作成と削除を行う RAW Web サービスを作成する方法を示しています。有効なセッション ID を指定して HTTP 要求を行うたびに、request_count という接続スコープ変数が増分されます。この例では、URL のみを使用した HTTP セッション管理を示します。

```
CREATE OR REPLACE FUNCTION create_session()
RETURNS LONG VARCHAR
BEGIN
    DECLARE session_id LONG VARCHAR;
    DECLARE tm TIMESTAMP;
    SET tm = NOW(*);
    SET session_id = 'session_' || CONNECTION_PROPERTY('Number') || '_'
        CONVERT( VARCHAR, SECONDS(tm) * 1000 + DATEPART( MILLISECOND, tm ) );
    CALL sa_set_http_option( 'SessionID', session_id );
    SELECT CONNECTION_PROPERTY( 'SessionID' ) INTO session_id;
    RETURN( session_id );
```

```
END
CREATE SERVICE mysession
    TYPE 'RAW'
    AUTHORIZATION OFF
    USER DBA
    AS CALL mysession_proc();
CREATE OR REPLACE PROCEDURE mysession_proc()
BEGIN
    DECLARE sesid LONG VARCHAR;
    DECLARE svcname LONG VARCHAR;
    DECLARE hostname LONG VARCHAR;
    DECLARE body LONG VARCHAR;

    SELECT CONNECTION_PROPERTY('SessionID') INTO sesid;
    SELECT CONNECTION_PROPERTY('HttpServiceName') INTO svcname;
    SELECT HTTP_HEADER( 'Host' ) INTO hostname;
    CALL sa_set_http_header ( 'Content-Type', 'text/html' );
    IF HTTP_VARIABLE('delete') IS NOT NULL THEN
        CALL sa_set_http_option( 'SessionID', NULL );
        SET body = '<html><body>Deleted ' || sesid
            || '</BR><a href="http://' || hostname || '/' || svcname || '">Start
Again</a>';
    ELSE IF sesid = '' THEN
        SET sesid = create_session();
        CREATE VARIABLE request_count INT;
        SET request_count = 0;
        SET body = '<html><body> Created session ID ' || sesid
            || '</br><a href="http://' || hostname || '/' || svcname
            || '?Sessionid=' || sesid || '"> Enter into Session</a>';
    ELSE
        SELECT CONNECTION_PROPERTY('SessionID') INTO sesid;
        SET request_count = request_count +1;
        SET body = '<html><body>Session ' || sesid || '</br>'
            || 'created ' || CONNECTION_PROPERTY('SessionCreateTime') || '</br>'
            || 'last access ' || CONNECTION_PROPERTY('SessionLastTime') || '</br>'
            || 'connection ID ' || CONNECTION_PROPERTY('Number') || '</br>'
            || '<h3>REQUEST COUNT is '|| request_count || '</h3><hr></br>'
            || '<a href="http://' || hostname || '/' || svcname
            || '?Sessionid=' || sesid || '">Enter into Session</a></br>'
            || '<a href="http://' || hostname || '/' || svcname
            || '?Sessionid=' || sesid || '&delete">Delete Session</a>';
        END IF;
    END IF;
    SELECT body;
END
```

セッション ID が接続用に定義されていない場合、CONNECTION_PROPERTY('SessionID') の結果は空の文字列です。この場合、mysession_proc プロシージャで新しいセッションが作成されます。

関連情報

非アクティブな HTTP セッションを検出する方法 [617 ページ]
HTTP セッションを削除するかセッション ID を変更する方法 [618 ページ]
Web サービスエラーコードリファレンス [661 ページ]

1.20.1.4.4.1.2 cookie を使用してセッションを管理する方法

cookie セッションステータス管理システムでは、クライアントアプリケーションまたは Web ブラウザにより、URL ではなく HTTP cookie ヘッダ内でセッション ID が指定されます。

cookie によるセッション管理は、sa_set_http_header システムプロシージャに 'Set-Cookie' HTTP 要求を指定することでサポートされます。

注記

クライアントアプリケーションまたは Web ブラウザで cookie を無効にできる場合、cookie ステータス管理に依存することはできません。URL と cookie の両方のステータス管理をサポートすることをおすすめします。セッション ID が URL と cookie の両方で提供されている場合、URL で提供したセッション ID が使用されます。

例

次の例は、セッションの作成と削除を行う RAW Web サービスを作成する方法を示しています。有効な SessionID を指定して HTTP 要求を行うたびに、request_count という接続スコープ変数が増分されます。この例では、cookie のみを使用した HTTP セッション管理を示します。これは、以前に紹介した mysession Web サービスのバリエーションです。

```
CREATE OR REPLACE FUNCTION set_session_cookie( session_id LONG VARCHAR, max_age
INTEGER )
RETURNS LONG VARCHAR
BEGIN
    SET cookie_str = 'sessionid=' || session_id || ';' ||
        ' expires=' || F_COOKIE_DATE( DATEADD( SECOND, max_age, CURRENT UTC
TIMESTAMP ) )|| ';' ||
        ' path=' || HTTP_ENCODE('/') || ';';
END
CREATE OR REPLACE FUNCTION f_cookie_date( in @expires timestamp )
RETURNS LONG VARCHAR
BEGIN
    DECLARE @dow ARRAY( 7 ) OF CHAR(3) = ARRAY( 'Sun', 'Mon', 'Tue', 'Wed',
'Thu', 'Fri', 'Sat' );
    DECLARE @moy ARRAY( 12 ) OF CHAR(3) = ARRAY( 'Jan', 'Feb', 'Mar', 'Apr',
'May', 'Jun',
    'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec' );
    RETURN @dow[[DATEPART( weekday, @expires ) ]] ||','
    || REPLACE(
    DATEFORMAT( @expires, 'DD-zzz-YY hh:nn:ss' ),
    'zzz',
    @moy[[ MONTH( @expires ) ]]
    )
END;
CREATE SERVICE mysession2
    TYPE 'RAW'
    AUTHORIZATION OFF
    USER DBA
    AS CALL mysession2_proc();
CREATE OR REPLACE PROCEDURE mysession2_proc()
BEGIN
    DECLARE sesid LONG VARCHAR;
    DECLARE svcname LONG VARCHAR;
    DECLARE hostname LONG VARCHAR;
    DECLARE body LONG VARCHAR;
    SELECT CONNECTION_PROPERTY('SessionID') INTO sesid;
    SELECT CONNECTION_PROPERTY('HttpServiceName') INTO svcname;
    SELECT HTTP_HEADER( 'Host' ) INTO hostname;
    CALL sa_set_http_header ( 'Content-Type', 'text/html' );
    IF HTTP_VARIABLE('delete') IS NOT NULL THEN
```

```
        CALL sa_set_http_option( 'SessionID', NULL );
        CALL set_session_cookie( sesid, 0 );
        SET body = '<html><body>Deleted ' || sesid
            || '</BR><a href="http://' || hostname || '/' || svcname || '">Start
Again</a>';
    ELSE IF sesid = '' THEN
        SET sesid = create_session();
        CALL set_session_cookie(sesid, 1*60*60);
        CREATE VARIABLE request_count INT;
        SET request_count = 0;
        SET body = '<html><body> Created session ID ' || sesid
            || '</br><a href="http://' || hostname || '/' || svcname
            || '"> Enter into Session</a>';
    ELSE
        SELECT CONNECTION_PROPERTY('SessionID') INTO sesid;
        SET request_count = request_count +1;
        SET body = '<html><body>Session ' || sesid || '</br>'
            || 'created ' || CONNECTION_PROPERTY('SessionCreateTime') || '</br>'
            || 'last access ' || CONNECTION_PROPERTY('SessionLastTime') || '</br>'
            || 'connection ID ' || CONNECTION_PROPERTY('Number') || '</br>'
            || '<h3>REQUEST COUNT is '|| request_count || '</h3><hr></br>'
            || '<a href="http://' || hostname || '/' || svcname
            || '">Enter into Session</a></br>'
            || '<a href="http://' || hostname || '/' || svcname
            || '?delete">Delete Session</a>';
        END IF;
    END IF;
    SELECT body;
END
```

関連情報

非アクティブな HTTP セッションを検出する方法 [617 ページ]
HTTP セッションを削除するかセッション ID を変更する方法 [618 ページ]

1.20.1.4.4.2 非アクティブな HTTP セッションを検出する方法

SessionCreateTime および SessionLastTime 接続プロパティを使用して、現在の接続がセッションコンテキスト内にあるかどうかを確認できます。いずれかの接続プロパティクエリから空の文字列が返された場合、HTTP 要求はセッションコンテキスト内で実行されていません。

SessionCreateTime 接続プロパティは、指定したセッションがいつ作成されたかを確認する基準になります。このプロパティは、SessionID を確立するために sa_set_http_option システムプロシージャを呼び出したときに、初めて定義されます。

SessionLastTime 接続プロパティは、最後に処理されたセッション要求が、その前の要求の終了時にデータベース接続を解放した時刻を示します。セッションが初めて作成されたときから、そのセッションを作成した要求が接続を解放するまでの間は、このプロパティに空の文字列が返されます。

注記

セッションのタイムアウト期間は、http_session_timeout オプションを使用して調整できます。

例

次の例は、SessionCreateTime と SessionLastTime 接続プロパティを使用したセッションの検出を示しています。

```
SELECT CONNECTION_PROPERTY( 'SessionCreateTime' ) INTO ses_create;
SELECT CONNECTION_PROPERTY( 'SessionLastTime' ) INTO ses_last;
```

関連情報

HTTP セッションを作成する方法 [613 ページ]
HTTP セッションを削除するかセッション ID を変更する方法 [618 ページ]

1.20.1.4.4.3 HTTP セッションを削除するかセッション ID を変更する方法

HTTP セッションを削除するか、セッション ID を変更するには、いくつかの方法があります。

セッションコンテキスト内でキャッシュされているデータベース接続を明示的に切断すると、セッションが削除されます。この方法によるセッションの削除はキャンセル操作と見なされ、そのセッションキューから解放された要求はすべてキャンセルステータスになります。この操作によって、セッションキューで待ち状態になっている未処理の要求は、すべて確実に終了します。同様に、サーバまたはデータベースが停止した場合も、データベース接続がすべてキャンセルされます。

セッションを削除するには、sa_set_http_option システムプロシージャで SessionID オプションを NULL または空の文字列に設定します。

次のコードは、セッションの削除に使用できます。

```
CALL sa_set_http_option( 'SessionID', null );
```

HTTP セッションを削除するか SessionID を変更すると、セッションキューで待ち状態になっている保留中の HTTP 要求が解放され、セッションコンテキスト外で実行できるようになります。保留中の要求では、同じデータベース接続は再使用されません。

セッション ID を既存のセッション ID に設定することはできません。SessionID を変更すると、古い SessionID を参照する保留中の要求は解放され、セッションレス要求として実行されます。新しい SessionID を参照する後続の要求では、古い SessionID によってインスタンス化された同じデータベース接続が再使用されます。

HTTP セッションを削除または変更すると、次の状態になります。

- セッションに属するデータベース接続が取得されたセッションが、現在の要求に継承されているかどうか、または、セッションレス要求によって新しいセッションがインスタンス化されたかどうかによって、動作が異なります。要求がセッションレスとして開始された場合は、セッションの作成または削除操作が即座に行われます。要求がセッションを継承している場合は、セッションの削除や SessionID の変更などのセッションステータスに関する変更は、要求が終了し変更内容がコミットされた後でのみ発生します。この動作の違いによって、1 つのクライアントで同じ SessionID を使用して同時要求を行ったとき、異常事態が発生する可能性があります。このような場合の対処を検討する必要があります。
- セッションを (保留中のセッションがない) 現在のセッションの SessionID に変更することは、エラーではなく、影響はありません。

- セッションを別の HTTP 要求によって使用されている SessionID に変更することは、エラーになります。
- 変更がすでに保留中のときにセッションを変更すると、保留中のセッションが削除され、新しい保留中のセッションが作成されます。保留中のセッションは、要求が正常に終了した場合にのみアクティブ化されます。
- 保留中のセッションがあるセッションを元の SessionID に戻すと、現在のセッションには変更を加えずに保留中のセッションが削除されます。

関連情報

非アクティブな HTTP セッションを検出する方法 [617 ページ]

1.20.1.4.4.4 HTTP セッションの管理

HTTP 要求によって作成されたセッションはすぐにインスタンス化されるため、そのセッションコンテキストを必要とする後続の HTTP 要求はセッションによってキューに追加されます。

この例では、sa_set_http_option プロシージャを使用してセッションがサーバで作成されると、ローカルホストクライアントは、データベース `dbname` で実行され、サービス session_service を実行する、指定されたセッション ID (session_63315422814117) のセッションに、次の URL を指定してアクセスできます。

```
http://localhost/dbname/session_service?sessionid=session_63315422814117
```

Web アプリケーションでは、HTTP Web サーバ内のアクティブなセッションの使用率を追跡する手段が必要になることがあります。セッションデータを取得するには、NEXT_CONNECTION 関数をアクティブなデータベース接続に対して繰り返し呼び出し、SessionID などのセッション関連のプロパティを確認します。

次の SQL 文は、アクティブなセッションの追跡方法を示します。

```
CREATE VARIABLE conn_id LONG VARCHAR;
CREATE VARIABLE the_sessionID LONG VARCHAR;
SELECT NEXT_CONNECTION( NULL, NULL ) INTO conn_id;
conn_loop:
    LOOP
        IF conn_id IS NULL THEN
            LEAVE conn_loop;
        END IF;
        SELECT CONNECTION_PROPERTY( 'SessionID', conn_id )
            INTO the_sessionID;
        IF the_sessionID != '' THEN
            PRINT 'conn_id = %1!, SessionID = %2!', conn_id, the_sessionID;
        ELSE
            PRINT 'conn_id = %1!', conn_id;
        END IF;
        SELECT NEXT_CONNECTION( conn_id, NULL ) INTO conn_id;
    END LOOP conn_loop;
    PRINT '¥n';
```

データベースサーバのメッセージウィンドウには、次の出力のようなデータが表示されます。

```
conn_id = 30
conn_id = 29, SessionID = session_63315442223323
conn_id = 28, SessionID = session_63315442220088
```

```
conn_id = 25, SessionID = session_63315441867629
```

セッションに属する接続を明示的に切断すると、接続が閉じられ、セッションが削除されます。切断される接続が HTTP 要求の処理において現在アクティブになっている場合、その要求は削除対象としてマーク付けされ、要求を終了するキャンセル通知が送信されます。要求が終了すると、セッションは削除され、接続は閉じられます。セッションを削除すると、そのセッションのキューで保留中だったすべての要求が、キューに再度追加されます。

接続が現在アクティブでない場合は、セッションは削除対象としてマーク付けされ、セッションタイムアウトキューの先頭に再度追加されます。セッションと接続は次回のタイムアウトサイクルで削除されます (通常は 5 秒以内)。削除対象としてマーク付けされたセッションはいずれも、新しい HTTP 要求では使用できません。

データベースが停止すると、すべてのセッションが失われます。

関連情報

HTTP セッションを作成する方法 [613 ページ]
HTTP セッションを削除するかセッション ID を変更する方法 [618 ページ]

1.20.1.4.4.5 HTTP セッションのエラーコード

HTTP セッションの管理中には、いくつかの共通のエラーが発生します。

新しい要求がアクセスしようとしたセッションで 16 を超える要求が保留になっていた場合、またはセッションをキューに追加しているときにエラーが発生した場合は、「`503 Service Unavailable`」のエラーが発生します。

クライアントの IP アドレスまたはホスト名がセッション作成者の IP アドレスまたはホスト名と一致しない場合は、`403 Forbidden` エラーが発生します。

存在しないセッションが指定された要求は、暗黙的にはエラーを生成しません。この状況を (SessionID、SessionCreateTime、または SessionLastTime 接続プロパティを確認することで) 検出して、適切なアクションを実行するのは、Web アプリケーションで行う必要があります。

関連情報

Web サービスエラーコードリファレンス [661 ページ]

1.20.1.4.5 文字セットの変換に関する考慮事項

デフォルトで、文字セット変換はテキストからなる出力結果セットで自動的に実行されます。バイナリオブジェクトなどの他のタイプの結果セットは、影響を受けません。

要求の文字セットは HTTP Web サーバの文字セットに変換され、結果セットはクライアントアプリケーションの文字セットに変換されます。複数の文字セットがリストされている場合、サーバでは要求リスト内の最初の適切な文字セットが使用されます。

文字セット変換は、sa_set_http_option システムプロシージャの HTTP オプション (CharsetConversion オプション) を設定することで有効または無効にできます。

次の例は、文字セットの自動変換をオフにする方法を示しています。

```
CALL sa_set_http_option('CharsetConversion', 'OFF');
```

sa_set_http_option システムプロシージャの 'AcceptCharset' オプションを使用して、文字セット変換が有効になっている場合の文字セットエンコードの設定を指定できます。

次の例は、Web サービスの文字セットエンコード優先度を ISO-8859-5 (サポートされていない場合は UTF-8) に指定する方法を示します。

```
CALL sa_set_http_option('AcceptCharset', 'iso-8859-5, utf-8');
```

文字セットはサーバ設定によって優先順位が付けられます。ただし、この選択には、クライアントの Accept-Charset 基準も考慮されます。クライアントに最も適し、かつ、このオプションでも指定されている文字セットが使用されます。

1.20.1.4.6 クロスサイトスクリプティングについての考慮事項

Web アプリケーションを開発する場合、そのアプリケーションがクロスサイトスクリプティング (XSS) に対して脆弱でないことを確認します。このタイプの脆弱性は、攻撃者が Web ページにスクリプトを注入しようとすると発生します。

Web アプリケーションコードが運用される前に、アプリケーション開発者やデータベース管理者がそのセキュリティの脆弱性の可能性について確認することを強くお奨めします。Open Web Application Security Project (https://www.owasp.org) には、Web アプリケーションのセキュリティを保護する方法に関する情報が含まれています。

ここにはクロスサイトスクリプティングの脆弱性の例があります。Web アプリケーションがページのリダイレクションをサポートしている場合、リダイレクトが行われる前に URL を検証する必要があります。たとえば、Web アプリケーションがユーザを元のページに戻すログイン機能をサポートしている場合、ユーザがリダイレクトされるページでユーザがオフサイトに移動しないことを確認する必要があります。次は、悪意のあるリダイレクトの使用例です。

```
https://www.mysite.com/login?referer=http://www.badsite.com/index.html
```

1.20.1.4.7 Web サービスシステムプロシージャ

次のシステムプロシージャは、Web サービスの操作時に使用します。

- sa_http_header_info システムプロシージャ

- sa_http_php_page システムプロシージャ
- sa_http_php_page_interpreted システムプロシージャ
- sa_http_variable_info システムプロシージャ
- sa_set_http_header システムプロシージャ
- sa_set_http_option システムプロシージャ
- sa_set_soap_header システムプロシージャ

関連情報

HTTP Web サーバとしてのデータベースサーバ [587 ページ]

1.20.1.4.8 Web サービス関数

Web サービス関数は、Web サービス内の HTTP 要求と SOAP 要求の処理を支援します。

利用可能な関数のリストについては、『SQL Anywhere サーバ - SQL リファレンス』マニュアルを参照してください。

また、Web サービスで使用できるシステムプロシージャも多数あります。

関連情報

HTTP Web サーバとしてのデータベースサーバ [587 ページ]

1.20.1.4.9 Web サービス接続プロパティ

Web サービス接続プロパティには、CONNECTION_PROPERTY 関数を使用してアクセスします。

次の構文を使用して、HTTP サーバの接続プロパティ値を SQL 関数またはプロシージャのローカル変数に格納します。

```
SELECT CONNECTION_PROPERTY('connection-property-name') INTO variable_name;
```

次に、Web サービスアプリケーションで一般的に使用される、ランタイム HTTP 要求の便利な接続プロパティを示します。

HttpServiceName

Web アプリケーションのサービス名オリジンを返します。

AuthType

接続時に使用される認証のタイプを返します。

ServerPort

データベースサーバの TCP/IP ポート番号または 0 を返します。

ClientNodeAddress

クライアント/サーバ接続のクライアント側に対応するノードを返します。
ServerNodeAddress

クライアント/サーバ接続のサーバ側に対応するノードを返します。
BytesReceived

クライアント/サーバ通信中に受信したバイト数を返します。
SessionID

接続のセッション ID がある場合は、そのセッション ID を返し、それ以外の場合は空の文字列を返します。
SessionCreateTime

HTTP セッションが作成された時刻を返します。
SessionLastTime

HTTP セッションにおける最後の要求の時刻を返します。

1.20.1.4.10 Web サービスオプション

Web サービスオプションは、HTTP サーバ動作をさまざまな面から制御します。

次の構文を使用して、HTTP サーバでパブリックオプションを設定します。

```
SET TEMPORARY OPTION PUBLIC.http_session_timeout=100;
```

次に、HTTP サーバでアプリケーション設定用に一般的に使用されるオプションを示します。

http_connection_pool_basesize

データベース接続の公称スレッショルドサイズを指定します。
http_connection_pool_timeout

未使用の接続を接続プールに保持できる時間の上限を指定します。
http_session_timeout

非アクティブ状態の HTTP セッションが維持されるデフォルトのタイムアウト時間 (分単位) を指定します。
request_timeout

1 つの要求を実行できる最大時間を設定します。
webservice_namespace_host

DISH サービスの仕様内で XML ネームスペースとして使用するホスト名を指定します。

1.20.1.5 SQL AnywhereHTTP Web サーバを参照する方法

Web サービスの命名および設計方法によって、有効な URL の内容が決まります。

URL によって、HTTP 要求または保護された HTTPS 要求を介して使用できる HTML コンテンツなどのリソースがユニークに指定されます。URL 構文は、HTTP Web サーバで定義された Web サービスへのアクセスが許可されます。

構文

```
{http|https}://host-name[:port-number][/dbn]/service-name[/path-name|?url-query]
```

パラメータ

host-name と port-number

Web サーバのロケーションを指定します。オプションとして、デフォルトの HTTP ポート番号または HTTPS ポート番号として定義されていない場合は、ポート番号を指定します。host-name には、Web サーバを実行中のコンピュータの IP アドレスを指定できます。port-number は、Web サーバを起動したときに使用されているポート番号に一致している必要があります。

dbn

データベース名を指定します。このデータベースは Web サーバで実行中であり、Web サービスが含まれている必要があります。

Web サーバで 1 つのデータベースしか実行されていない場合、またはプロトコルオプションの特定の HTTP/HTTPS リスナに対してデータベース名が指定されている場合は、dbn を指定する必要はありません。

service-name

アクセスする Web サービスの名前を指定します。この Web サービスは、dbn で指定されたデータベースに存在している必要があります。Web サービスを作成または変更する場合は、スラッシュ文字 (/) が有効であり、service-name の一部として使用できます。URL の残りの部分は、定義されたサービスと一致します。

service-name が未指定で、**root** Web サービスが定義されている場合は、クライアント要求が処理されます。要求を処理する適切なサービスをサーバで識別できない場合は、404 Not Found エラーが返されます。関連動作として、**root** Web サービスが存在せず、URL 基準に基づいて要求を処理できない場合は、404 Not Found エラーが生成されます。

path-name

サービス名の解決後、残りのスラッシュで区切られたパスは、Web サービスプロシージャからアクセスできます。URL ON を指定して作成されたサービスの場合は、指定された URL HTTP 変数を使用してパス全体にアクセスできます。URL ELEMENTS を指定して作成されたサービスの場合は、指定された HTTP 変数 URL1 ～ URL10 を使用して各パス要素にアクセスできます。

パス要素変数は、サービス文定義のパラメータ宣言でホスト変数として定義できます。別の方法として、またはこの方法と組み合わせて、HTTP_VARIABLE 関数呼び出しを使用してストアドプロシージャ内から HTTP 変数にアクセスすることもできます。

次の例は、Web サービスの作成に使用する、URL 句を ELEMENTS に設定した SQL 文を示しています。

```
CREATE SERVICE TestWebService
    TYPE 'HTML'
    URL ELEMENTS
    AUTHORIZATION OFF
    USER DBA
    AS CALL TestProcedure ( :url1, :url2 );
```

この TestWebService Web サービスは、url1 と url2 の各ホスト変数を明示的に参照するプロシージャを呼び出します。

この Web サービスには、次の URL を使用してアクセスできます。この場合、デフォルトのポートを介した localhost からデータベース demo で TestWebService が実行されていることが前提となります。

```
http://localhost/demo/TestWebService/Assignment1/Assignment2/Assignment3
```

この URL は、TestProcedure を実行し、Assignment1 値を url1 に割り当て、Assignment2 値を url2 に割り当てる TestWebService にアクセスします。オプションとして、HTTP_VARIABLE 関数を使用して、TestWebService から他のパス要素にアクセスできます。たとえば、HTTP_VARIABLE('url3') 関数呼び出しは Assignment3 を返します。

url-query

HTTP GET 要求は、HTTP 変数を指定するクエリコンポーネントがあるパスをたどることがあります。同様に、標準の application/x-www-form-urlencoded Content-Type を使用する POST 要求の本文は、要求本文内の HTTP 変数を渡すことができます。いずれの場合も、HTTP 変数は名前/値ペアとして渡されます。変数名は、等号で値から区切られます。変数は、アンパサンドで区切られます。

HTTP 変数は、サービス文のパラメータリスト内でホスト変数として明示的に宣言したり、サービス文のストアドプロシージャから HTTP_VARIABLE 関数を使用してアクセスしたりできます。

たとえば、次の SQL 文は、2 つのホスト変数を必要とする Web サービスを作成します。ホスト変数は、コロン (:) のプレフィクスで識別されます。

```
CREATE SERVICE ShowSalesOrderDetail
    TYPE 'HTML'
    URL OFF
    AUTHORIZATION OFF
    USER DBA
    AS CALL ShowSalesOrderDetail( :customer_id, :product_id );
```

デフォルトのポートを介した localhost 上の Web サーバで ShowSalesOrderDetail Web サービス が利用可能であると仮定した場合、次の URL を使用して Web サービスにアクセスできます。

```
http://localhost/demo/ShowSalesOrderDetail?customer_id=101&product_id=300
```

この URL は、ShowSalesOrderDetail にアクセスし、値 101 を customer_id に割り当て、値 300 を product_id に割り当てます。結果は、Web ブラウザに HTML フォーマットで表示されます。

備考

各 Web サービスには、独自の Web コンテンツがあります。通常、このコンテンツは、データベース内のカスタム関数とプロシージャによって生成されますが、SQL 文を指定する URL を使用してコンテンツを生成することもできます。別の方法として、またはこの方法と組み合わせて、専用サービスで処理されないすべての HTTP 要求を処理する **root** Web サービスを定義することもできます。通常、**root** Web サービスは、要求の URL とヘッダから、要求の処理方法を判断します。

サーバへの接続にユーザ名とパスワードが必要な場合は、Web ブラウザから入力が求められます。ブラウザの base64 によりユーザ入力が認証要求ヘッダ内でエンコードされ、要求が再送信されます。

Web サービスの URL 句が ON または ELEMENTS に設定されている場合、`path-name` と `url-query` の URL 構文プロパティを同時に使用すると、異なるフォーマットオプションのいずれかを使用して Web サービスにアクセスできるようになります。これらの構文プロパティを同時に使用する場合は、`path-name` フォーマットを最初に使用し、その後に `url-query` フォーマットを使用する必要があります。

次の例に示す SQL 文は、URL 句が ON に設定され、url 変数を定義する Web サービスを作成します。

```
CREATE SERVICE ShowSalesOrderDetail
    TYPE 'HTML'
    URL ON
    AUTHORIZATION OFF
    USER DBA
    AS CALL ShowSalesOrderDetail( :product_id, :url );
```

次に、url 値 101 と product_id 値 300 を割り当てる有効な URL のサンプルを示します。

- `http://localhost:80/demo/ShowSalesOrderDetail2/101?product_id=300`
- `http://localhost:80/demo/ShowSalesOrderDetail2?url=101&product_id=300`
- `http://localhost:80/demo/ShowSalesOrderDetail2?product_id=300&url=101`

`path-name` と `url-query` のコンテキストでホスト変数名が複数回割り当てられていると、最後の割り当てが常に優先されます。たとえば、次のサンプル URL は、url 値 101 と product_id 値 300 を割り当てます。

- `http://localhost:80/demo/ShowSalesOrderDetail2/302?url=101&product_id=300`
- `http://localhost:80/demo/ShowSalesOrderDetail2/String?product_id=300&url=101`

注記

ここに記載する情報は、RAW、XML、および HTML などの汎用 HTTP Web サービスタイプを使用する HTTP Web サービスに適用されます。ブラウザを使用して、SOAP 要求を発行することはできません。JSON サービスからは、AJAX を使用した Web サービスアプリケーションで使用される結果セットが返されます。

例

次の URL 構文は、デフォルトのポートを介してローカル HTTP サーバ上の demo というデータベースで実行されている gallery_image という Web サービスにアクセスするために使用されますが、この場合、gallery_image サービスは URL ON で定義されていると仮定します。

```
http://localhost/demo/gallery_image/sunset.jpg
```

この URL は、従来の Web サーバのディレクトリでグラフィックファイルを要求するように見えますが、HTTP Web サーバの入力パラメータとして `sunset.jpg` が指定された gallery_image サービスにアクセスします。

次の SQL 文は、この動作を実行するために、gallery サービスを HTTP サーバで定義する方法を示しています。

```
CREATE SERVICE gallery_image
    TYPE 'RAW'
    URL ON
    AUTHORIZATION OFF
    USER DBA
    AS CALL gallery_image ( :url );
```

gallery_image サービスは、同じ名前でプロシージャを呼び出して、クライアントで指定された URL を渡します。この Web サービス定義によってアクセスできる gallery_image プロシージャの実装例については、*%SQLANYSAMP17%*`¥SQLAnywhere¥HTTP¥gallery.sql` を参照してください。

関連情報

クライアント提供の HTTP 変数とヘッダにアクセスする方法 [606 ページ]
root Web サービスを作成し、カスタマイズする方法 [600 ページ]

1.20.2 Web クライアントを使用した Web サービスへのアクセス

データベースサーバを Web クライアントとして使用して、SQL Anywhere HTTP Web サーバによって実行される Web サービスまたは Apache や IIS などのサードパーティの Web サーバにアクセスできます。

データベースサーバを Web クライアントとして使用できるだけでなく、Web サービスは、クライアントアプリケーションに JDBC や ODBC などの従来のインタフェースの代替を提供します。追加のコンポーネントが必要なく、Perl や Python などのスクリプト言語を含む各種の言語で記述されたマルチプラットフォームクライアントアプリケーションからアクセスできるため、これらは簡単に配備されます。

このセクションの内容:

Web クライアントとしてのデータベースサーバの使用 [628 ページ]
データベースサーバを Web クライアントアプリケーションとして実行して HTTP サーバに接続し、一般的な HTTP Web サービスにアクセスします。

SQL Anywhere HTTP Web サーバへのアクセス [630 ページ]
Python および C# の 2 種類のクライアントアプリケーションを使用して、SQL Anywhere HTTP Web サーバにアクセスする方法を説明します。

Web クライアントアプリケーションの開発 [632 ページ]
SQL Anywhere データベースを、Web クライアントアプリケーションとして使用して、SQL Anywhere Web サービスまたはサードパーティの Web サーバ上で実行される Web サービスにアクセスできます。

HTTP 要求と SOAP 要求の構造 [660 ページ]
パラメータの代入中に使用する場合を除き、関数またはプロシージャのすべてのパラメータは、Web サービス要求の一部として渡されます。渡されるときのフォーマットは、Web サービス要求のタイプによって異なります。

Web クライアント要求のロギング方法 [661 ページ]
HTTP 要求やトランスポートデータを含む Web サービスクライアントの情報は、Web サービスクライアントログファイルに記録できます。

1.20.2.1 Web クライアントとしてのデータベースサーバの使用

データベースサーバを Web クライアントアプリケーションとして実行して HTTP サーバに接続し、一般的な HTTP Web サービスにアクセスします。

前提条件

あらゆるタイプのオンライン Web サーバに接続する Web クライアントアプリケーションを開発できますが、このトピックでは、ローカル SQL Anywhere HTTP サーバをポート 8082 で起動し、次の SQL 文を使用して作成された SampleXMLService という Web サービスに接続することを前提とします。

```
CREATE SERVICE SampleHTMLService
    TYPE 'HTML'
    USER DBA
    AUTHORIZATION OFF
    AS CALL sp_echo(:i, :f, :s);
CREATE PROCEDURE sp_echo(i INTEGER, f REAL, s LONG VARCHAR)
RESULT(ret_i INTEGER, ret_f REAL, ret_s LONG VARCHAR)
BEGIN
    SELECT i, f, s;
END;
```

コンテキスト

Web クライアントプロシージャは、それ自体へのアウトバウンド HTTP 要求を作成できず、同じデータベース上で実行されているローカルホスト SQL Anywhere Web サービスを呼び出すことができません。次の手順では、新規データベースを作成し、2 番目のデータベースサーバを起動して、作成した新規データベースに接続します。

Web クライアントアプリケーションを作成するには、次のタスクを実行します。

手順

1. クライアントデータベースが存在しない場合は、次のコマンドを実行して作成します。

   ```
   dbinit -dba DBA,passwd client-database-name
   ```

   client-database-name をクライアントデータベースの新しい名前に置き換えます。
2. 次のコマンドを実行して、クライアントデータベースを起動します。

   ```
   dbsrv17 client-database-name.db
   ```
3. 次のコマンドを実行して、Interactive SQL を通じてクライアントデータベースに接続します。

   ```
   dbisql -c "UID=DBA;PWD=passwd;SERVER=client-database-name"
   ```

4. 次の SQL 文を使用して、SampleHTMLService Web サービスに接続する新しいクライアントプロシージャを作成します。

```
CREATE PROCEDURE client_post(f REAL, i INTEGER, s VARCHAR(16), x VARCHAR(16))
    URL 'http://localhost:8082/SampleHTMLService'
    TYPE 'HTTP:POST'
    HEADER 'User-Agent:SATest';
```

5. 次の SQL 文を実行して、クライアントプロシージャを呼び出し、Web サーバに HTTP 要求を送信します。

```
CALL client_post(3.14, 9, 's varchar', 'x varchar');
```

client_post によって作成された HTTP POST 要求は、次のような出力になります。

```
POST /SampleHTMLService HTTP/1.0
ASA-Id: ea1746b01cd0472eb4f0729948db60a2
User-Agent: SATest
Accept-Charset: windows-1252, UTF-8, *
Date: Wed, 9 Jun 2015 21:55:01 GMT
Host: localhost:8082
Connection: close
Content-Type: application/x-www-form-urlencoded; charset=windows-1252
Content-Length: 58
&f=3.1400001049041748&i=9&s=s%20varchar&x=x%20varchar
```

結果

Web サーバ上で実行される Web サービス SampleHTMLService によって i、f、s の各パラメータ値が POST 要求から抽出され、パラメータとして sp_echo プロシージャに渡されます。パラメータ値 x は無視されます。sp_echo プロシージャによって結果セットが作成され、Web サービスに返されます。正しく対応付けられるように、クライアントと Web サービスの間でパラメータ名を決めておくことが重要です。

Web サービスで応答が作成されてクライアントに送り返されます。Interactive SQL に表示される出力は次のようになります。

属性	値	インスタンス
Status	HTTP /1.1 200 OK	1
Body	<html> <head> <title>/ SampleHTMLService</ title></head> <body> / SampleHTMLService <table border=1> <tr class="header"><th>ret _i</th> <th>ret_f</th> <th>ret_s</th> </tr> <tr><td>9</ td><td>3.1400001049041748 </td><td>s varchar</td>	1

属性	値	インスタンス
Date	Wed, 09 Jun 2015 21:55:01 GMT	1
Connection	close	1
Expires	Wed, 09 Jun 2015 21:55:01 GMT	1
Content-Type	text/html; charset=windows-1252	1
Server	SQLAnywhere/17.0.4	1

関連情報

HTTP Web サーバとしてデータベースサーバを使用するためのクイックスタート [587 ページ]
HTTP Web サーバで Web サービスアプリケーションを開発する方法 [603 ページ]

1.20.2.2 SQL Anywhere HTTP Web サーバへのアクセス

Python および C# の 2 種類のクライアントアプリケーションを使用して、SQL Anywhere HTTP Web サーバにアクセスする方法を説明します。

コンテキスト

あらゆるタイプのオンライン Web サーバに接続する Web クライアントアプリケーションを開発できますが、このトピックでは、ローカル SQL Anywhere HTTP サーバをポート 8082 で起動し、次の SQL 文を使用して作成された SampleXMLService という Web サービスに接続することを前提とします。

```
CREATE SERVICE SampleXMLService
    TYPE 'XML'
    USER DBA
    AUTHORIZATION OFF
    AS CALL sp_echo2(:i, :f, :s);
CREATE PROCEDURE sp_echo2(i INTEGER, f NUMERIC(6,2), s LONG VARCHAR )
RESULT( ret_i INTEGER, ret_f NUMERIC(6,2), ret_s LONG VARCHAR )
BEGIN
    SELECT i, f, s;
END;
```

C# または Python を使用して XML Web サービスにアクセスするには、次のタスクを実行します。

手順

1. HTTP サーバ上で Web サービスに接続するプロシージャを作成します。

SampleXMLService Web サービスにアクセスするコードを作成します。

オプション	説明

C# の場合は、次のコードを入力します。	`using System;`
	`using System.Xml;`
	`public class WebClient`
	`{`
	`  static void Main(string[] args)`
	`  {`
	`  XmlTextReader reader = new XmlTextReader(`
	`  "http://localhost:8082/SampleXMLService?i=5&f=3.14&s=hello");`
	`  while (reader.Read())`
	`  {`
	`  switch (reader.NodeType)`
	`  {`
	`  case XmlNodeType.Element:`
	`  if (reader.Name == "row")`
	`  {`
	`  Console.Write(reader.GetAttribute("ret_i") + " ");`
	`  Console.Write(reader.GetAttribute("ret_s") + " ");`
	`  Console.WriteLine(reader.GetAttribute("ret_f"));`
	`  }`
	`  break;`
	`  }`
	`  }`
	`  reader.Close();`
	`  }`
	`}`
	
	このコードを `DocHandler.cs` というファイルに保存します。
	
	プログラムをコンパイルするには、コマンドプロンプトで次のコマンドを実行します。
	
	`csc /out:DocHandler.exe DocHandler.cs`

Python の場合は、次のコードを入力します。	`import xml.sax`
	`class DocHandler( xml.sax.ContentHandler ):`
	`  def startElement( self, name, attrs ):`
	`  if name == 'row':`
	`  table_int = attrs.getValue( 'ret_i' )`
	`  table_string = attrs.getValue( 'ret_s' )`
	`  table_numeric = attrs.getValue( 'ret_f' )`
	`  print('%s %s %s' % ( table_int, table_string, table_numeric ))`
	`parser = xml.sax.make_parser()`
	`parser.setContentHandler( DocHandler() )`
	`parser.parse('http://localhost:8082/SampleXMLService?i=5&f=3.14&s=hello')`
	
	このコードを `DocHandler.py` というファイルに保存します。

2. HTTP サーバによって送信された結果セットに対する操作を実行します。

オプション	説明
C# の場合は、次のコマンドを実行します。	`DocHandler`
Python の場合は、次のコマンドを実行します。	`python DocHandler.py`

結果

アプリケーションによって次の出力が表示されます。

```
5 hello 3.14
```

関連情報

HTTP Web サーバで Web サービスアプリケーションを開発する方法 [603 ページ]
HTTP Web サーバとしてデータベースサーバを使用するためのクイックスタート [587 ページ]

1.20.2.3 Web クライアントアプリケーションの開発

SQL Anywhere データベースを、Web クライアントアプリケーションとして使用して、SQL Anywhere Web サービスまたはサードパーティの Web サーバ上で実行される Web サービスにアクセスできます。

SQL Anywhere Web クライアントアプリケーションを作成するには、Web サービスのターゲット終了ポイントを指定する URL 句などの設定句を使用して、ストアドプロシージャや関数を記述します。Web クライアントプロシージャは、本文がないことを除き、その他のストアドプロシージャと同様に使用します。Web クライアントプロシージャは、呼び出されると、アウトバウンド HTTP または SOAP 要求を作成します。Web クライアントプロシージャは、それ自体へのアウトバウンド HTTP 要求を作成できないようになっています。Web クライアントプロシージャでは、同じデータベース上で実行されている SQL Anywhere Web サービスを呼び出すことができません。

Web サービスアプリケーションの詳細な例については、*%SQLANYSAMP17%*`¥SQLAnywhere¥HTTP` ディレクトリを参照してください。

このセクションの内容:

Web クライアント関数とプロシージャの要件と推奨事項 [633 ページ]
Web サービスクライアントのプロシージャと関数には、Web サービスの終了ポイントを識別するために、URL 句を定義する必要があります。Web サービスクライアントのプロシージャまたはファンクションは、設定用の特殊な句があることを除き、その他のストアドプロシージャや関数と同様に使用します。

Web サービスへの変数の指定 [645 ページ]
Web サービスのタイプに応じて、さまざまな方法で Web サービスに変数を指定できます。

結果セットからの変数へのアクセス [648 ページ]
Web サービスクライアント呼び出しは、ストアド関数またはプロシージャで実行できます。

句の値に使用する代入パラメータ [658 ページ]
ストアドプロシージャまたは関数の宣言済みパラメータは、そのプロシージャまたは関数が実行されるたびに、句の定義内のプレースホルダを自動的に置き換えます。

関連情報

Web クライアント SQL 文 [645 ページ]

1.20.2.3.1 Web クライアント関数とプロシージャの要件と推奨事項

Web サービスクライアントのプロシージャと関数には、Web サービスの終了ポイントを識別するために、URL 句を定義する必要があります。Web サービスクライアントのプロシージャまたはファンクションは、設定用の特殊な句があることを除き、その他のストアドプロシージャや関数と同様に使用します。

CREATE PROCEDURE 文と CREATE FUNCTION 文を使用して、Web サーバに SOAP 要求または HTTP 要求を送信する Web クライアント関数とプロシージャを作成できます。

次の一覧に、Web クライアント関数とプロシージャの作成または変更に関する要件と推奨事項を示します。Web クライアント関数またはプロシージャを作成または変更する場合、次の情報を指定できます。

- Web サービスの終了ポイントを指定する絶対 URL を必要とする URL 句 (必須)
- 要求が HTTP であるか、HTTP を介した SOAP であるかを指定する TYPE 句 (推奨)
- クライアントアプリケーションにアクセスできるポート (省略可)
- HTTP 要求ヘッダを指定する HEADER 句 (省略可)
- SOAP 要求エンベロープ内の SOAP ヘッダ基準を指定する SOAPHEADER 句 (省略可。SOAP 要求の場合のみ)
- ネームスペース URI (SOAP 要求の場合のみ)

このセクションの内容:

Web クライアント URL 句 [634 ページ]
Web サービスの終了ポイントのロケーションを指定して、Web クライアント関数またはプロシージャがアクセスできるようにします。CREATE PROCEDURE 文と CREATE FUNCTION 文の URL 句により、アクセスする Web サービスの URL が提供されます。

Web サービスの要求タイプ [636 ページ]
Web クライアント関数またはプロシージャを作成するとき、Web サーバに送信するクライアント要求のタイプを指定できます。CREATE PROCEDURE 文と CREATE FUNCTION 文の TYPE 句は、要求をフォーマットしてから Web サーバに送信します。

Web クライアントポート [637 ページ]
ファイアウォールを介してサーバへの接続を確立するときに使用するポートを指定する必要がある場合があります。CREATE PROCEDURE 文と CREATE FUNCTION 文の CLIENTPORT 句を使用して、クライアントアプリケーションが TCP/IP を使用して通信するポート番号を指定できます。

HTTP 要求ヘッダの管理 [638 ページ]
CREATE PROCEDURE 文と CREATE FUNCTION 文の HEADER 句を使用して、HTTP 要求ヘッダを追加、変更、または削除できます。

SOAP 要求ヘッダの管理 [640 ページ]
SOAP 要求ヘッダは、SOAP エンベロープ内の XML フラグメントです。

SOAP ネームスペース URI の要件 [643 ページ]
ネームスペース URI は、特定の SOAP 操作の SOAP 要求エンベロープを作成するために使用される XML ネームスペースを指定します。ネームスペース URI が定義されていない場合は、URL 句のドメインコンポーネントが使用されます。

Web クライアント SQL 文 [645 ページ]
プロシージャと関数を作成するための SQL 文が、Web サービスアクセスメカニズムを定義するために使用されます。

1.20.2.3.1.1 Web クライアント URL 句

Web サービスの終了ポイントのロケーションを指定して、Web クライアント関数またはプロシージャがアクセスできるようにします。CREATE PROCEDURE 文と CREATE FUNCTION 文の URL 句により、アクセスする Web サービスの URL が提供されます。

HTTP サービスの URL の指定

URL 句内で HTTP スキームを指定すると、プロシージャまたは関数が HTTP プロトコルを使用したセキュリティ保護されていない通信に対応できるように設定されます。

次の文に、ポート 8082 の localhost にある HTTP Web サーバによって管理される dbname というデータベースにある SampleHTMLService という Web サービスに要求を送信するプロシージャを作成する方法を示します。

```
CREATE PROCEDURE client_sender(f REAL, i INTEGER, s VARCHAR(16))
    URL 'http://localhost:8082/dbname/SampleHTMLService'
    TYPE 'HTTP:POST'
    HEADER 'User-Agent:SATest';
```

データベース名は、HTTP サーバが複数のデータベースを管理する場合にのみ必要です。localhost を HTTP サーバのホスト名または IP アドレスに置き換えることができます。

HTTPS サービスの URL の指定

URL 句内で HTTPS スキームを指定すると、プロシージャまたは関数が TLS (トランスポートレイヤセキュリティ) 経由のセキュリティ保護された通信に対応できるように設定されます。

Web クライアントアプリケーションは、RSA サーバ証明書にアクセスするか、またはセキュア HTTPS 要求を発行するためにサーバ証明書に署名した証明書にアクセスする必要があります。サーバを認証し、中間者攻撃を防ぐために、クライアントプロシージャには証明書が必要です。

CREATE PROCEDURE 文と CREATE FUNCTION 文の CERTIFICATE 句を使用してサーバを認証し、セキュリティ保護されたデータチャネルを確立します。証明書をファイルに保存してファイル名を指定するか、証明書全体を文字列値として指定できます。両方を行うことはできません。

次の文に、ポート 8082 の localhost にある HTTPS サーバ内の dbname というデータベースにある SecureHTMLService という Web サービスに要求を送信するプロシージャを作成する方法を示します。

```
CREATE PROCEDURE client_sender(f REAL, i INTEGER, s VARCHAR(16))
    URL 'HTTPS://localhost:8082/dbname/SecureHTMLService'
    CERTIFICATE 'file=C:¥¥Users¥¥Public¥¥Documents¥¥SQL Anywhere 17¥¥Samples¥
¥Certificates¥¥rsaroot.crt'
    TYPE 'HTTP:POST'
    HEADER 'User-Agent:SATest';
```

この例の CERTIFICATE 句は、RSA サーバ証明書が *C:¥Users¥Public¥Documents¥SQL Anywhere 17¥Samples¥Certificates¥rsaroot.crt* ファイルにあることを示します。

> **注記**
>
> HTTPS_FIPS を指定すると、強制的に FIPS ライブラリが使用されます。HTTPS_FIPS を指定したときに、FIPS ライブラリがない場合は、代わりに FIPS 以外のライブラリが使用されます。

プロキシサーバの URL の指定

プロキシサーバを使用して送信する必要がある要求もあります。CREATE PROCEDURE 文と CREATE FUNCTION 文の PROXY 句を使用して、プロキシサーバの URL を指定し、その URL に要求をリダイレクトします。プロキシサーバは、最終送信先へ要求を転送し、応答を取得して、SQL Anywhere に応答を返します。

関連情報

HTTP Web サーバを起動する方法 [589 ページ]
Web クライアント SQL 文 [645 ページ]

1.20.2.3.1.2 Web サービスの要求タイプ

Web クライアント関数またはプロシージャを作成するとき、Web サーバに送信するクライアント要求のタイプを指定できます。CREATE PROCEDURE 文と CREATE FUNCTION 文の TYPE 句は、要求をフォーマットしてから Web サーバに送信します。

HTTP 要求フォーマットの指定

Web クライアント関数とプロシージャは、TYPE 句で指定されたフォーマットが HTTP プレフィックスで始まる場合、HTTP 要求を送信します。

たとえば、指定した URL に HTTP 要求を送信する PostOperation という HTTP プロシージャを作成するには、Web クライアントデータベースで次の SQL 文を実行します。

```
CREATE PROCEDURE PostOperation(a INTEGER, b CHAR(128))
    URL 'HTTP://localhost:8082/dbname/SampleWebService'
    TYPE 'HTTP:POST';
```

この例では、要求は HTTP:POST 要求としてフォーマットされます。これにより、次のような要求が生成されます。

```
POST /dbname/SampleWebService HTTP/1.0
ASA-Id: e88a416e24154682bf81694feaf03052
User-Agent: SQLAnywhere/17.0.4.1691
Accept-Charset: windows-1252, UTF-8, *
Date: Fri, 03 Feb 2012 15:02:49 GMT
Host: localhost:8082
Connection: close
Content-Type: application/x-www-form-urlencoded; charset=windows-1252
Content-Length: 12
a=123&b=data
```

SOAP 要求フォーマットの指定

Web クライアント関数とプロシージャは、TYPE 句で指定されたフォーマットが SOAP プレフィックスで始まる場合、HTTP 要求を送信します。

たとえば、指定した URL に SOAP 要求を送信する SoapOperation という SOAP プロシージャを作成するには、Web クライアントデータベースで次の文を実行します。

```
CREATE PROCEDURE SoapOperation(intVariable INTEGER, charVariable CHAR(128))
    URL 'HTTP://localhost:8082/dbname/SampleSoapService'
    TYPE 'SOAP:DOC';
```

この例では、このプロシージャを呼び出すと、URL に SOAP:DOC 要求が送信されます。これにより、次のような要求が生成されます。

```
POST /dbname/SampleSoapService HTTP/1.0
ASA-Id: e88a416e24154682bf81694feaf03052
User-Agent: SQLAnywhere/17.0.4.1691
Accept-Charset: windows-1252, UTF-8, *
```

```
Date: Fri, 03 Feb 2012 15:05:13 GMT
Host: localhost:8082
Connection: close
Content-Type: text/xml; charset=windows-1252
Content-Length: 428
SOAPAction: "HTTP://localhost:8082/SoapOperation"
<?xml version="1.0"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:m="HTTP://localhost:8082">
  <SOAP-ENV:Body>
    <m:SoapOperation>
      <m:intVariable>123</m:intVariable>
      <m:charVariable>data</m:charVariable>
    </m:SoapOperation>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

プロシージャ名は、本文内の <m:SoapOperation> タグに表示されます。プロシージャに対する 2 つのパラメータ intVariable と charVariable は、それぞれ <m:intVariable> と <m:charVariable> になります。

デフォルトでは、SOAP 要求が構築されるとき、SOAP 操作名としてストアドプロシージャ名が使用されます。パラメータ名は、SOAP エンベロープのタグ名内に示されます。サーバでは SOAP 要求内にこれらの名前があることを予期しているため、SOAP ストアドプロシージャを定義するときには、これらの名前を正確に参照してください。SET 句を使用すると、対象となるプロシージャの代替 SOAP 操作名を指定できます。最も簡単な場合 (1 つの文字列値を返す SOAP RPC など) を除き、プロシージャではなく関数の定義を使用してください。SOAP 関数は、OPENXML を使用して解析できる、完全な SOAP 応答エンベロープを返します。

関連情報

Web クライアント SQL 文 [645 ページ]
Web サービスへの変数の指定 [645 ページ]
HTTP 要求と SOAP 要求の構造 [660 ページ]

1.20.2.3.1.3 Web クライアントポート

ファイアウォールを介してサーバへの接続を確立するときに使用するポートを指定する必要がある場合があります。CREATE PROCEDURE 文と CREATE FUNCTION 文の CLIENTPORT 句を使用して、クライアントアプリケーションが TCP/IP を使用して通信するポート番号を指定できます。

ファイアウォールによって特定範囲のポートへのアクセスが制限されていないかぎり、この機能を使用しないでください。

たとえば、Web クライアントデータベースで次の SQL 文を実行して、5050 ～ 5060 の範囲のいずれかのポートまたはポート 5070 を使用して指定した URL に要求を送信する SomeOperation というプロシージャを作成します。

```
CREATE PROCEDURE SomeOperation()
    URL 'HTTP://localhost:8082/dbname/SampleWebService'
    CLIENTPORT '5050-5060,5070';
```

必要な場合は、ポート番号の範囲を指定します。ポート番号を 1 つ指定すると、一度に 1 つの接続のみが維持されます。クライアントアプリケーションは、いずれかのポート番号と接続が確立するまで、指定されたすべてのポート番号へのアクセスを試行します。接続を閉じた後、同じサーバとポートを使って新しい接続が作成されないように、数分のタイムアウト期間が開始されます。

この機能は、ClientPort ネットワークプロトコルオプションの設定と類似しています。

関連情報

Web クライアント SQL 文 [645 ページ]

1.20.2.3.1.4 HTTP 要求ヘッダの管理

CREATE PROCEDURE 文と CREATE FUNCTION 文の HEADER 句を使用して、HTTP 要求ヘッダを追加、変更、または削除できます。

名前を参照することによって、HTTP 要求ヘッダを抑制できます。ヘッダ名の後にコロン、値の順に指定して、HTTP 要求ヘッダ値を追加または変更します。ヘッダ値の指定は省略可能です。

たとえば、HTTP 要求ヘッダを制限する指定の URL に要求を送信する SomeOperation2 というプロシージャを作成するには、Web クライアントデータベースで次の SQL 文を実行します。

```
CREATE PROCEDURE SomeOperation2()
    URL 'HTTP://localhost:8082/dbname/SampleWebService'
    TYPE 'HTTP:GET'
    HEADER 'SOAPAction¥nDate¥nFrom:¥nCustomAlias:John Doe';
```

この例では、自動的に生成される Date ヘッダが抑制されます。From ヘッダは、含まれているものの、値が割り当てられていません。HTTP 要求には CustomAlias という新しいヘッダが含められ、このヘッダには値 John Doe が割り当てられます。GET 要求は次のようになります。

```
GET /dbname/SampleWebService HTTP/1.0
ASA-Id: e88a416e24154682bf81694feaf03052
User-Agent: SQLAnywhere/17.0.4.1691
Accept-Charset: windows-1252, UTF-8, *
From:
Host: localhost:8082
Connection: close
CustomAlias: John Doe
```

長いヘッダ値の折り返しがサポートされています。折り返すには、¥n の直後に 1 つ以上のスペースが必要です。

次の例は、長いヘッダ値のサポートを示します。

```
CREATE PROCEDURE SomeOperation3()
    URL 'HTTP://localhost:8082/dbname/SampleWebService'
    TYPE 'HTTP:POST'
    HEADER 'heading1: This long value¥n is really long for a header.¥n
    heading2:shortvalue';
```

POST 要求は次のようになります。

```
POST /dbname/SampleWebService HTTP/1.0
ASA-Id: e88a416e24154682bf81694feaf03052
User-Agent: SQLAnywhere/17.0.4.1691
Accept-Charset: windows-1252, UTF-8, *
Date: Fri, 03 Feb 2012 15:26:04 GMT
heading1: This long value is really long for a header.     heading2:shortvalue
Host: localhost:8082
Connection: close
Content-Type: application/x-www-form-urlencoded; charset=windows-1252
Content-Length: 0
```

注記

SOAP 関数またはプロシージャの作成時に、WSDL で指定された SOAP サービス URI に SOAPAction HTTP 要求ヘッダを設定してください。

自動生成される HTTP 要求ヘッダ

自動生成されるヘッダを修正すると、予期しない結果になる可能性があります。次の HTTP 要求ヘッダは、事前の対策を行わずに変更しないでください。

HTTP ヘッダ	説明
Accept-Charset	常に自動的に生成されます。このヘッダを変更または削除すると、予期しないデータ変換エラーが発生する可能性があります。
ASA-Id	常に自動的に生成されます。このヘッダは、デッドロックを防ぐため、クライアントアプリケーションがそれ自体に接続しないようにします。
権限	URL にクレデンシャルが含まれるときに自動生成されます。このヘッダを変更または削除すると、要求がエラーになる可能性があります。BASIC 認証だけがサポートされています。ユーザとパスワードの情報は、HTTPS を使用した接続時だけ含めるようにしてください。
接続	Connection: close は常に自動的に生成されます。クライアントアプリケーションは、永続的接続をサポートしません。変更した場合、接続がハングする可能性があります。
ホスト	常に自動的に生成されます。HTTP/1.1 クライアントが Host ヘッダを提供しない場合、400 Bad Request で応答するには HTTP/1.1 サーバが必要です。
Transfer-Encoding	チャンクモードで要求を通知するときに自動生成されます。このヘッダやチャンク値を削除すると、クライアントが CHUNK モードを使用しているときにエラーになります。
Content-Length	チャンクモード以外で要求を通知するときに自動生成されます。このヘッダは、本文のコンテンツ長をサーバに通知するために必要です。コンテンツ長が不正な場合、接続がハングするか、データが失われる可能性があります。

関連情報

Web クライアント SQL 文 [645 ページ]

1.20.2.3.1.5 SOAP 要求ヘッダの管理

SOAP 要求ヘッダは、SOAP エンベロープ内の XML フラグメントです。

SOAP 操作とそのパラメータはリモートプロシージャコール (RPC) であると考えることができますが、SOAP 要求ヘッダを使用して、特定の要求や応答内のメタ情報を転送できます。SOAP 要求ヘッダによって、認証情報やセッション基準などのアプリケーションのメタデータが転送されます。

SOAPHEADER 句は、SOAP 要求ヘッダエントリに準拠する有効な XML フラグメントである必要があります。複数の SOAP 要求ヘッダエントリを指定できます。ストアドプロシージャまたは関数によって、SOAP ヘッダ要素 (SOAP-ENV:Header) 内に SOAP 要求ヘッダエントリが自動的に注入されます。SOAPHEADER 値では、静的定数として宣言するか、代入パラメータメカニズムを使用して動的に設定できる SOAP ヘッダが指定されます。次に、サンプルの SOAP 要求の一部を示します。これにはそれぞれ Authentication および Session という 2 つの XML ヘッダが含まれています。

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:m="HTTP://localhost:8082">
  <SOAP-ENV:Header>
    <Authentication xmlns="CustomerOrderURN">
      <userName pwd="none" mustUnderstand="1">
        <first>John</first>
        <last>Smith</last>
      </userName>
    </Authentication>
    <Session xmlns="SomeSession">123456789</Session>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <m:SoapOperation>
      <m:intVariable>123</m:intVariable>
      <m:charVariable>data</m:charVariable>
    </m:SoapOperation>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP 呼び出しによって返される SOAP 応答ヘッダの処理は、関数とプロシージャによって異なります。最も柔軟で推奨される方法である関数を使用すると、SOAP 応答エンベロープ全体が受信されます。その後、OPENXML 演算子を使用して応答エンベロープを処理し、SOAP ヘッダと SOAP 本文のデータを抽出できます。プロシージャを使用すると、IN または INOUT 変数にマッピングする代入パラメータを使用する場合にのみ SOAP 応答ヘッダを抽出できます。SOAP プロシージャで許容される IN または INOUT パラメータは、1 つだけです。

Web サービス関数は、応答 SOAP エンベロープを解析して、ヘッダエントリを取得する必要があります。

例

次の例は、パラメータと SOAP ヘッダを送信する SOAP プロシージャと関数を作成する方法を示します。ラッパープロシージャは、Web サービスプロシージャコールにデータを取り込み、応答を処理するために使用されます。soapAddItemProc プロシージャは SOAP Web サービスプロシージャの使用方法を示し、soapAddItemFunc 関数は SOAP Web サービス

関数の使用方法を示し、httpAddItemFunc 関数は SOAP ペイロードが HTTP Web サービスプロシージャに渡される方法を示します。

次の例は、代入パラメータを使用して SOAP ヘッダを送信する SOAP クライアントプロシージャを示しています。1 つの INOUT パラメータを使用して SOAP ヘッダを受信しています。ラッパーストアドプロシージャ addItemProcWrapper から soapAddItemProc を呼び出して、パラメータを含む SOAP ヘッダを送受信する方法を示しています。

```
CREATE PROCEDURE soapAddItemProc(amount INT, item LONG VARCHAR,
        INOUT inoutheader LONG VARCHAR, IN inheader LONG VARCHAR)
    URL 'http://localhost:8082/itemStore'
    SET 'SOAP( OP=addItems )'
    TYPE 'SOAP:DOC'
    SOAPHEADER '!inoutheader!inheader';
CREATE PROCEDURE addItemProcWrapper(amount INT, item LONG VARCHAR,
        first_name LONG VARCHAR, last_name LONG VARCHAR)
BEGIN
    DECLARE io_header LONG VARCHAR;     // inout (write/read) soap header
    DECLARE resxml LONG VARCHAR;
    DECLARE soap_header_sent LONG VARCHAR;
    DECLARE i_header LONG VARCHAR;      // in (write) only soap header
    DECLARE err int;
    DECLARE crsr CURSOR FOR
        CALL soapAddItemProc( amount, item, io_header, i_header );
    SET io_header = XMLELEMENT( 'Authentication',
                        XMLATTRIBUTES('CustomerOrderURN' as xmlns),
                        XMLELEMENT('userName', XMLATTRIBUTES(
                                        'none' as pwd,
                                        '1' as mustUnderstand ),
                            XMLELEMENT( 'first', first_name ),
                            XMLELEMENT( 'last', last_name ) ) );
    SET i_header = '<Session xmlns="SomeSession">123456789</Session>';
    SET soap_header_sent = io_header || i_header;
    OPEN crsr;
    FETCH crsr INTO resxml, err;
    CLOSE crsr;
    SELECT resxml, err, soap_header_sent, io_header AS soap_header_received;
END;
/* example call to addItemProcWrapper */
CALL addItemProcWrapper( 5, 'shirt', 'John', 'Smith' );
```

次の例は、代入パラメータを使用して SOAP ヘッダを送信する SOAP クライアント関数を示しています。SOAP 応答エンベロープ全体が返されます。SOAP ヘッダは OPENXML 演算子を使用して解析できます。ラッパー関数 addItemFuncWrapper から soapAddItemFunc を呼び出して、パラメータを含む SOAP ヘッダを送受信する方法を示しています。また、OPENXML 演算子を使用して応答を処理する方法も示しています。

```
CREATE FUNCTION soapAddItemFunc(amount INT, item LONG VARCHAR,
        IN inheader1 LONG VARCHAR, IN inheader2 LONG VARCHAR )
    RETURNS XML
    URL 'http://localhost:8082/itemStore'
    SET 'SOAP(OP=addItems)'
    TYPE 'SOAP:DOC'
    SOAPHEADER '!inheader1!inheader2';
CREATE PROCEDURE addItemFuncWrapper(amount INT, item LONG VARCHAR,
        first_name LONG VARCHAR, last_name LONG VARCHAR )
BEGIN
    DECLARE i_header1 LONG VARCHAR;
    DECLARE i_header2 LONG VARCHAR;
    DECLARE res LONG VARCHAR;
    DECLARE ns LONG VARCHAR;
    DECLARE xpath LONG VARCHAR;
    DECLARE header_entry LONG VARCHAR;
    DECLARE localname LONG VARCHAR;
    DECLARE namespaceuri LONG VARCHAR;
    DECLARE r_quantity int;
```

```
    DECLARE r_item LONG VARCHAR;
    DECLARE r_status LONG VARCHAR;
    SET i_header1 = XMLELEMENT( 'Authentication',
                        XMLATTRIBUTES('CustomerOrderURN' as xmlns),
                        XMLELEMENT('userName', XMLATTRIBUTES(
                                        'none' as pwd,
                                        '1' as mustUnderstand ),
                            XMLELEMENT( 'first', first_name ),
                            XMLELEMENT( 'last',  last_name ) ) );
    SET i_header2 = '<Session xmlns="SessionURN">123456789</Session>';
    SET res = soapAddItemFunc( amount, item, i_header1, i_header2 );
    SET ns = '<ns xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"'
           || ' xmlns:mp="urn:sap-com:sa-xpath-metaprop"'
           || ' xmlns:customer="CustomerOrderURN"'
           || ' xmlns:session="SessionURN"'
           || ' xmlns:tns="http://localhost:8082"></ns>';
    // Process headers...
    SET xpath = '//SOAP-ENV:Header/*';
    BEGIN
         DECLARE crsr CURSOR FOR SELECT * FROM
                 OPENXML( res, xpath, 1, ns )
                      WITH (  "header_entry" LONG VARCHAR '@mp:xmltext',
                              "localname"    LONG VARCHAR '@mp:localname',
                              "namespaceuri" LONG VARCHAR '@mp:namespaceuri' );
        OPEN crsr;
        FETCH crsr INTO "header_entry", "localname", "namespaceuri";
        CLOSE crsr;
    END;
    // Process body...
    SET xpath = '//tns:row';
    BEGIN
         DECLARE crsr1 CURSOR FOR SELECT * FROM
                 OPENXML( res, xpath, 1, ns )
                      WITH (  "r_quantity" INT 'tns:quantity/text()',
                              "r_item"     LONG VARCHAR 'tns:item/text()',
                              "r_status"   LONG VARCHAR 'tns:status/text()' );
        OPEN crsr1;
        FETCH crsr1 INTO "r_quantity", "r_item", "r_status";
        CLOSE crsr1;
    END;
    SELECT r_item, r_quantity, r_status, header_entry, localname, namespaceuri;
END;
/* example call to addItemFuncWrapper */
CALL addItemFuncWrapper( 6, 'shorts', 'Jack', 'Smith' );
```

次の例は、HTTP:POST を SOAP ペイロード全体の転送手段として使用する方法を示しています。この方法では、Web サービス SOAP クライアントプロシージャを作成するのではなく、SOAP ペイロードを転送する Web サービス HTTP プロシージャを作成しています。ラッパープロシージャ addItemHttpWrapper から httpAddItemFunc を呼び出して、POST 関数の使用例を示しています。パラメータを含む SOAP ヘッダを送受信する方法と、応答を受け取る方法を示しています。

```
CREATE FUNCTION httpAddItemFunc(soapPayload XML)
    RETURNS XML
    URL 'http://localhost:8082/itemStore'
    TYPE 'HTTP:POST:text/xml'
    HEADER 'SOAPAction: "http://localhost:8082/addItems"';
CREATE PROCEDURE addItemHttpWrapper(amount INT, item LONG VARCHAR)
RESULT(response XML)
BEGIN
    DECLARE payload XML;
    DECLARE response XML;
    SET payload =
'<?xml version="1.0"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
  xmlns:m="http://localhost:8082">
  <SOAP-ENV:Body>
    <m:addItems>
      <m:amount>' || amount || '</m:amount>
      <m:item>' || item || '</m:item>
    </m:addItems>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>';
    SET response = httpAddItemFunc( payload );
    /* process response as demonstrated in addItemFuncWrapper */
    SELECT response;
END;
/* example call to addItemHttpWrapper */
CALL addItemHttpWrapper( 7, 'socks' );
```

制限事項

- サーバ側 SOAP サービスは、現在は input および output SOAP ヘッダ要件を定義できません。したがって、SOAP ヘッダメタデータを DISH サービスの WSDL 出力で使用することはできません。SOAP クライアントツールキットは、SOAP サービスの終了ポイント用に SOAP ヘッダインタフェースを自動生成することができません。
- SOAP ヘッダフォールトはサポートされていません。

関連情報

クライアント提供の SOAP 要求ヘッダにアクセスする方法 [610 ページ]

1.20.2.3.1.6 SOAP ネームスペース URI の要件

ネームスペース URI は、特定の SOAP 操作の SOAP 要求エンベロープを作成するために使用される XML ネームスペースを指定します。ネームスペース URI が定義されていない場合は、URL 句のドメインコンポーネントが使用されます。

サーバ側の SOAP プロセッサはこの URI を使用して、要求のメッセージ本文内にあるさまざまなエンティティの名前を解釈します。CREATE PROCEDURE 文と CREATE FUNCTION 文の NAMESPACE 句では、ネームスペース URI を指定します。

ネームスペース URI を指定しないと、プロシージャコールが成功しない可能性があります。通常、この情報は一般の Web サーバのマニュアルで説明されていますが、必要なネームスペース URI は、Web サーバから使用できる WSDL から取得できます。SQL Anywhere HTTP Web サーバと通信しようとしている場合は、DISH サービスにアクセスして、WSDL を生成できます。

通常、NAMESPACE は、wsdl:definition 要素内の WSDL ドキュメントの先頭で指定される targetNamespace 属性からコピーできます。後続の '/' は重要であるため、これを含める場合は注意してください。次に、指定された SOAP 操作の soapAction 属性を確認します。この属性は、後述するように、生成される SOAPAction HTTP ヘッダに対応している必要があります。

NAMESPACE 句は 2 つの役割を果たします。NAMESPACE 句は、SOAP エンベロープの本文のネームスペースを指定し、プロシージャに TYPE 'SOAP:DOC' が指定されている場合は、SOAPAction HTTP ヘッダのドメインコンポーネントとして使用されます。

次の例は、NAMESPACE 句の使用方法を示します。

```
CREATE FUNCTION an_operation(a_parameter LONG VARCHAR)
    RETURNS LONG VARCHAR
    URL 'http://wsdl.domain.com/fictitious.asmx'
    TYPE 'SOAP:DOC'
    NAMESPACE 'http://wsdl.domain.com/'
```

Interactive SQL で次の SQL 文を実行します。

```
SELECT an_operation('a_value');
```

この文では、次の出力のような SOAP 要求が生成されます。

```
POST /fictitious.asmx HTTP/1.0
SOAPAction: "http://wsdl.domain.com/an_operation"
Host: wsdl.domain.com
Content-Type: text/xml
Content-Length: 387
Connection: close
<?xml version="1.0"?>
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:m="http://wsdl.domain.com/">
  <SOAP-ENV:Body>
   <m:an_operation>
    <m:a_parameter>a_value</m:a_parameter>
   </m:an_operation>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

プレフィクス 'm' のネームスペースが http://wsdl.domain.com/ に設定され、SOAPAction HTTP ヘッダによって、SOAP 操作の完全に修飾された URL が指定されます。

後続のスラッシュは SQL Anywhere が正しく動作するための必要条件ではありませんが、診断が難しい応答障害の原因となる可能性があります。SOAPAction HTTP ヘッダは、後続のスラッシュに関係なく、正しく生成されます。

NAMESPACE が指定されていない場合、URL 句のドメインコンポーネントが SOAP 本文のネームスペースとして使用され、プロシージャが TYPE 'SOAP:DOC' の場合は、HTTP SOAPAction HTTP ヘッダの生成に使用されます。上記の例で NAMESPACE 句が省略された場合は、http://wsdl.domain.com がネームスペースとして使用されます。後続のスラッシュ '/' がないことが、わずかに異なります。SOAPAction HTTP ヘッダを含め、SOAP 要求のそれ以外のすべての点は、上記の例と同じです。

上記の SOAP:DOC で説明したように、NAMESPACE 句は SOAP 本文のネームスペースを指定するときに使用します。ただし、SOAPAction HTTP ヘッダは、空の値 SOAPAction: "" で生成されます。

SOAP:DOC 要求タイプを使用する場合は、SOAPAction HTTP ヘッダを作成するためにネームスペースも使用されます。

関連情報

Web サービスの要求タイプ [636 ページ]
Web クライアント SQL 文 [645 ページ]
DISH サービスを作成する方法 [597 ページ]

1.20.2.3.1.7 Web クライアント SQL 文

プロシージャと関数を作成するための SQL 文が、Web サービスアクセスメカニズムを定義するために使用されます。

次の SQL 文を使用して、Web クライアントの開発を支援できます。

Web クライアント関連の SQL 文	説明
CREATE FUNCTION	HTTP 要求または SOAP over HTTP 要求を行う Web クライアント関数を作成します。
ALTER FUNCTION	関数を変更します。
CREATE PROCEDURE	HTTP サーバへの HTTP 要求または SOAP 要求を作成する Web クライアントプロシージャを作成します。
ALTER PROCEDURE	プロシージャを変更します。

1.20.2.3.2 Web サービスへの変数の指定

Web サービスのタイプに応じて、さまざまな方法で Web サービスに変数を指定できます。

Web クライアントは、次のいずれかの方法を使用して、一般的な HTTP Web サービスに変数を指定できます。

- URL のサフィックス
- HTTP 要求の本文

標準 SOAP エンベロープの一部として変数を含めることによって、SOAP サービスタイプに変数を指定できます。

このセクションの内容:

URL による Web サービスへの変数の指定 [646 ページ]
HTTP Web サーバは、Web ブラウザによって URL で指定された変数を管理できます。

HTTP 要求の本文での変数の指定 [646 ページ]
Web クライアント関数またはプロシージャの TYPE 句で HTTP:POST を指定して、HTTP 要求の本文で変数を指定できます。

SOAP エンベロープでの変数の指定 [647 ページ]
Web クライアント関数またはプロシージャの SET SOAP オプションを使用して SOAP エンベロープで変数を指定し、SOAP 操作を設定できます。

関連情報

Web サービスタイプ [592 ページ]

1.20.2.3.2.1 URL による Web サービスへの変数の指定

HTTP Web サーバは、Web ブラウザによって URL で指定された変数を管理できます。

これらの変数は、次のいずれかの表記規則で表すことができます。

- 次の例のように、URL の末尾に変数を追加し、各パラメータ値をスラッシュ (/) で区切ります。

```
http://localhost/database-name/param1/param2/param3
```

- 次の例のように、変数を URL パラメータリストで明示的に定義します。

```
http://localhost/database-name/?arg1=param1&arg2=param2&arg3=param3
```

- 次の例のように、変数の URL への追加とパラメータリストでの定義を組み合わせます。

```
http://localhost/database-name/param4/param5?arg1=param1&arg2=param2&arg3=param3
```

Web サーバによる URL の解釈は、Web サービス URL 句の指定方法によって異なります。

関連情報

クライアント提供の HTTP 変数とヘッダにアクセスする方法 [606 ページ]
root Web サービスを作成し、カスタマイズする方法 [600 ページ]
SQL AnywhereHTTP Web サーバを参照する方法 [623 ページ]

1.20.2.3.2.2 HTTP 要求の本文での変数の指定

Web クライアント関数またはプロシージャの TYPE 句で HTTP:POST を指定して、HTTP 要求の本文で変数を指定できます。

デフォルトで、TYPE HTTP:POST では、application/x-www-form-urlencoded MIME タイプが使用されます。すべてのパラメータは urlencoded であり、要求の本文内で渡されます。オプションで、メディアタイプが指定されている場合、Content-Type 要求ヘッダは指定されたメディアタイプに自動的に調整され、要求の本文内で 1 つのパラメータ値がアップロードされます。

例

次の例では、XMLService という Web サービスが localhost Web サーバ上に存在すると想定しています。SQL Anywhere クライアントデータベースを設定して Interactive SQL を通じて接続し、次の SQL 文を実行します。

```
CREATE PROCEDURE SendXMLContent(xmlcode LONG VARCHAR)
    URL 'http://localhost/XMLService'
    TYPE 'HTTP:POST:text/xml';
```

この文で作成されるプロシージャを使用すると、HTTP 要求の本文で text/xml フォーマットで変数を送信できます。

Interactive SQL で次の SQL 文を実行して、HTTP 要求を XMLService Web サービスに送信します。

```
CALL SendXMLContent('<title>Hello World!</title>');
```

プロシージャコールによって xmlcode パラメータに値が割り当てられ、Web サービスに送信されます。

関連情報

クライアント提供の HTTP 変数とヘッダにアクセスする方法 [606 ページ]

1.20.2.3.2.3 SOAP エンベロープでの変数の指定

Web クライアント関数またはプロシージャの SET SOAP オプションを使用して SOAP エンベロープで変数を指定し、SOAP 操作を設定できます。

次のコードは、Web クライアント関数で SOAP 操作を設定する方法を示しています。

```
CREATE FUNCTION soapAddItemFunc(amount INT, item LONG VARCHAR)
    RETURNS XML
    URL 'http://localhost:8082/itemStore'
    SET 'SOAP(OP=addItems)'
    TYPE 'SOAP:DOC';
```

この例では、addItems は、soapAddItemFunc 関数にパラメータとして渡される **amount** 値と **item** 値を含む SOAP 操作です。

次のサンプルスクリプトを実行して要求を送信できます。

```
SELECT soapAddItemFunc(5, 'shirt');
```

soapAddItemFunc ファンクションコールの呼び出しによって、次のような SOAP エンベロープが生成されます。

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:m="http://localhost:8082">
  <SOAP-ENV:Body>
```

```
    <m:addItems>
      <m:amount>5</m:amount>
      <m:item>shirt</m:item>
    </m:addItems>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

上記の方法の代わりに、独自の SOAP ペイロードを作成し、HTTP ラッパーでサーバに送信できます。

SOAP サービスに対する変数は、標準 SOAP 要求の一部として含める必要があります。これ以外の方法で提供される値は無視されます。

次のコードは、カスタマイズされた SOAP エンベロープを構築する HTTP ラッパープロシージャを作成する方法を示しています。

```
CREATE PROCEDURE addItemHttpWrapper(amount INT, item LONG VARCHAR)
RESULT(response XML)
BEGIN
    DECLARE payload XML;
    DECLARE response XML;
    SET payload =
'<?xml version="1.0"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:m="http://localhost:8082">
  <SOAP-ENV:Body>
    <m:addItems>
      <m:amount>' || amount || '</m:amount>
      <m:item>' || item || '</m:item>
    </m:addItems>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>';
    SET response = httpAddItemFunc( payload );
    /* process response as demonstrated in addItemFuncWrapper */
    SELECT response;
END;
```

次のコードは、要求の送信に使用する Web クライアント関数を示しています。

```
CREATE FUNCTION httpAddItemFunc(soapPayload XML)
    RETURNS XML
    URL 'http://localhost:8082/itemStore'
    TYPE 'HTTP:POST:text/xml'
    HEADER 'SOAPAction: "http://localhost:8082/addItems"';
```

次のサンプルスクリプトを実行して要求を送信できます。

```
CALL addItemHttpWrapper( 7, 'socks' );
```

1.20.2.3.3 結果セットからの変数へのアクセス

Web サービスクライアント呼び出しは、ストアド関数またはプロシージャで実行できます。

関数を使用した場合、戻り値のタイプは CHAR、VARCHAR、LONG VARCHAR などの文字データ型です。返される値は、HTTP 応答の本文です。ヘッダ情報は含まれません。HTTP ステータス情報を含む要求に関する追加情報は、プロシージャによって返されます。したがって、追加情報にアクセスする場合は、プロシージャの使用をお奨めします。

SOAP プロシージャ

SOAP 関数からは SOAP 応答を含んだ XML ドキュメントが返されます。

SOAP 応答は構造化された XML ドキュメントであるため、デフォルトでは SQL Anywhere はこの情報を利用してさらに役立つ結果セットを作成しようとします。返された応答ドキュメント内の最上位レベルの各タグが抽出され、カラム名として使用されます。これらのタグのそれぞれの下にあるサブツリーの内容は、そのカラムのローの値として使用されます。

たとえば、次の SOAP 応答が返される場合、SQL Anywhere は次のデータセットを作成します。

```
<SOAP-ENV:Envelope
  xmlns:SOAPSDK1="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAPSDK2="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:SOAPSDK3="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
     <ElizaResponse xmlns:SOAPSDK4="SoapInterop">
        <Eliza>Hi, I'm Eliza. Nice to meet you.</Eliza>
     <ElizaResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Eliza
Hi, I'm Eliza. Nice to meet you.

この例では、応答ドキュメントは <SOAP-ENV:Body> タグ内にある <ElizaResponse> タグによって区切られています。

結果セットには、最上位レベルのタグの数だけのカラムが含まれます。SOAP 応答には最上位レベルのタグが 1 つしかないため、この結果セットのカラムは 1 つだけです。この最上位レベルタグである Eliza が、カラム名となります。

XML 処理機能

SOAP 応答を含む XML 結果セット内の情報には、OPENXML プロシージャを使用してアクセスできます。

次の例では、OPENXML プロシージャを使用して SOAP 応答の一部を抽出します。この例は、SYSWEBSERVICE テーブルの内容を公開するために SOAP サービスとして Web サービスを使用しています。

```
CREATE SERVICE get_webservices
    TYPE 'SOAP'
    AUTHORIZATION OFF
    USER DBA
    AS SELECT * FROM SYSWEBSERVICE;
```

2 番目の SQL Anywhere データベースで作成する必要のある次の Web クライアント関数は、この Web サービスへの呼び出しを発行します。この関数の戻り値は、SOAP 応答ドキュメント全体です。DNET がデフォルトの SOAP サービスフォーマットであるため、応答は .NET DataSet フォーマットになります。

```
CREATE FUNCTION get_webservices()
    RETURNS LONG VARCHAR
    URL 'HTTP://localhost/get_webservices'
    TYPE 'SOAP:DOC';
```

次の文は、OPENXML プロシージャを使用して結果セットの 2 つのカラムを抽出する方法を示しています。service_name カラムおよび secure_required カラムは、セキュアな SOAP サービスと、HTTPS が必要とされる場所をそれぞれ示します。

```
SELECT *
FROM OPENXML( get_webservices(), '//row' )
WITH ("Name"    CHAR(128) 'service_name',
      "Secure?" CHAR(1)   'secure_required' );
```

この文は、**row** ノードの子孫を選択することによって機能します。WITH 句は、目的の 2 つの要素に基づき、結果セットを作成します。get_webservices Web サービスのみが存在すると想定し、この関数は以下の結果セットを返します。

Name	Secure?
get_webservices	N

このセクションの内容:

Web サービスからの結果セットの取得 [650 ページ]
タイプ HTTP の Web サービスプロシージャは、応答に関する全情報を 3 つのカラムから成る結果セットで返します。この結果セットには、応答ステータス、ヘッダ情報、および本文が含まれます。

SOAP のデータ型 [652 ページ]
デフォルトでは、パラメータ入力の XML エンコードは String 型であり、SOAP サービスフォーマットの結果セット出力には、結果セット内のカラムのデータ型について具体的に記述する情報がまったく含まれていません。すべてのフォーマットで、パラメータのデータ型は String です。

SOAP の構造化されたデータ型 [657 ページ]
XML データ型は、Web サービス関数とプロシージャ内のパラメータおよび戻り値として使用できます。

1.20.2.3.3.1 Web サービスからの結果セットの取得

タイプ HTTP の Web サービスプロシージャは、応答に関する全情報を 3 つのカラムから成る結果セットで返します。この結果セットには、応答ステータス、ヘッダ情報、および本文が含まれます。

最初のカラムには Attribute という名前が付けられており、LONG VARCHAR データ型です。2 番目のカラムには Value という名前が付けられており、これも LONG VARCHAR データ型です。3 番目のカラムには instance という名前が付けられており、INTEGER データ型です。

結果セットには、応答ヘッダフィールドごとに 1 ロー、HTTP ステータス行 (Status 属性) に対して 1 ロー、応答本文 (Body 属性) に対して 1 ローが含まれます。

HEADERS 句で同じ名前を持つ複数のヘッダを指定する Web プロシージャは、すべてのヘッダを Web サーバに送信します。

次の例は、複数の Set-Cookie ヘッダフィールドを設定する Web サービスプロシージャの結果セットを表します。

属性	値	インスタンス
Status	HTTP /1.0 200 OK	1
Body	<!DOCTYPE HTML ... ><HTML> ... </HTML>	1

属性	値	インスタンス
Content-Type	text/html	1
Server	GWS/2.1	1
Content-Length	2234	1
Date	Mon, 18 Oct 2004, 16:00:00 GMT	1
Set-cookie	choice=chocolate; ...	1
Set-cookie	account=services...	2

例

例として使用する次の Web サービスストアドプロシージャを作成します。

```
CREATE OR REPLACE PROCEDURE SAPWebPage()
URL 'http://www.sap.com/index.html'
TYPE 'HTTP';
```

データベースサーバおよび接続しようとしている Web サーバとの間にプロキシサーバがある場合、PROXY 句をプロシージャ定義に追加しなければならないこともあります。次に、`proxy` というプロキシサーバの例を示します。

```
PROXY 'http://proxy:8080'
```

結果セットとして Web サービスから応答を取得するには、次の SELECT クエリを実行します。

```
SELECT * FROM SAPWebPage()
    WITH (Attribute LONG VARCHAR, Value LONG VARCHAR, Instance INT);
```

Web サービスプロシージャでは結果セットの形式を記述しないので、WITH 句でテンポラリビューを定義する必要があります。

クエリの結果をテーブルに格納できます。テーブルを作成して結果セットの値を格納するには、次の SQL 文を実行します。

```
CREATE TABLE StoredResults(
     Attribute LONG VARCHAR,
     Value     LONG VARCHAR,
     Instance  INT
);
```

StoredResults テーブルに結果セットを挿入します。

```
INSERT INTO StoredResults
    SELECT * FROM SAPWebPage()
    WITH (Attribute LONG VARCHAR, Value LONG VARCHAR, Instance INT);
```

SELECT 文を変更する句を追加します。たとえば、結果セットの特定のローのみが必要な場合は、WHERE 句を追加して SELECT の結果を 1 つのローに限定することができます。

```
SELECT * FROM SAPWebPage()
    WITH (Attribute LONG VARCHAR, Value LONG VARCHAR, Instance INT)
    WHERE Attribute = 'Status';
```

この SELECT 文は、結果セットからステータス情報のみを取得します。この文は呼び出しが成功したことを確認するために使用できます。

関連情報

Web クライアント SQL 文 [645 ページ]

1.20.2.3.3.2 SOAP のデータ型

デフォルトでは、パラメータ入力の XML エンコードは String 型であり、SOAP サービスフォーマットの結果セット出力には、結果セット内のカラムのデータ型について具体的に記述する情報がまったく含まれていません。すべてのフォーマットで、パラメータのデータ型は String です。

DNET フォーマットの場合、応答のスキーマセクション内ですべてのカラムは String として型指定されています。CONCRETE フォーマットと XML フォーマットの場合は、応答にデータ型情報が含まれません。このデフォルトの動作は、DATATYPE 句を使用して操作できます。

SQL Anywhere では、DATATYPE 句を使用してデータ型指定を有効にします。データ型情報は、すべての SOAP サービスフォーマットでパラメータ入力と結果セット出力 (応答) の XML エンコードに含めることができます。これにより、パラメータを String に明示的に変換するクライアントコードが不要になるため、SOAP ツールキットからのパラメータ受け渡しが簡単になります。たとえば整数は int として渡すことができます。XML コード化されたデータ型では SOAP ツールキットを使用してデータを解析し、適切な型にキャストします。

String データ型を排他的に使用する場合、アプリケーションでは結果セット内の各カラムのデータ型を暗黙的にわかっている必要があります。データ型指定が Web サーバで要求される場合は、必要ありません。データ型情報が含まれるかどうかを制御するために、Web サービスの定義時に DATATYPE 句を使用できます。

結果セット応答にデータ型指定を含めるようにする Web サービス定義の例を次に示します。

```
CREATE SERVICE "SASoapTest/EmployeeList"
    TYPE 'SOAP'
    AUTHORIZATION OFF
    SECURE OFF
    USER DBA
    DATATYPE OUT
    AS SELECT * FROM Employees;
```

この例では、サービスにはパラメータがないため、データ型情報は結果セット応答のみに対して要求されます。

型指定は、'SOAP' 型として定義されているすべての Web サービスに適用できます。

入力パラメータのデータ型指定

入力パラメータの型指定は、パラメータのデータ型を実際のデータ型として DISH サービスで生成される WSDL で公開するだけでサポートされます。

一般的な String パラメータ定義 (または型指定されていないパラメータ) は次のようになります。

```
<s:element minOccurs="0" maxOccurs="1" name="a_varchar" nillable="true"
type="s:string" />
```

String パラメータは nil 可能な場合があります。つまり、出現することもしないこともあります。

整数などの型指定されたパラメータの場合、そのパラメータは出現する必要があり、nil 可能ではありません。次はその例です。

```
<s:element minOccurs="1" maxOccurs="1" name="an_int" nillable="false"
type="s:int" />
```

出力パラメータのデータ型指定

'SOAP' 型であるすべての Web サービスでは、応答データ内のデータ型情報を公開できます。データ型は、ローセットカラム要素内の属性として公開されます。

SOAP FORMAT 'CONCRETE' Web サービスからの型指定された SimpleDataSet 応答の例を次に示します。

```
<SOAP-ENV:Body>
  <tns:test_types_concrete_onResponse>
   <tns:test_types_concrete_onResult xsi:type='tns:SimpleDataset'>
    <tns:rowset>
      <tns:row>
       <tns:lvc xsi:type="xsd:string">Hello World</tns:lvc>
       <tns:i xsi:type="xsd:int">99</tns:i>
       <tns:ii xsi:type="xsd:long">99999999</tns:ii>
       <tns:f xsi:type="xsd:float">3.25</tns:f>
       <tns:d xsi:type="xsd:double">.555555555555555582</tns:d>
       <tns:bin xsi:type="xsd:base64Binary">AAAAZg==</tns:bin>
       <tns:date xsi:type="xsd:date">2006-05-29-04:00</tns:date>
      </tns:row>
     </tns:rowset>
   </tns:test_types_concrete_onResult>
  <tns:sqlcode>0</tns:sqlcode>
  </tns:test_types_concrete_onResponse>
</SOAP-ENV:Body>
```

XML データを String として返す SOAP FORMAT 'XML' Web サービスからの応答の例を次に示します。内部ローセットは、コード化された XML で構成されます。ここでは、わかりやすいようにデコードされた形式で示されています。

```
<SOAP-ENV:Body>
 <tns:test_types_XML_onResponse>
  <tns:test_types_XML_onResult xsi:type='xsd:string'>
   <tns:rowset
     xmlns:tns="http://localhost/satest/dish"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <tns:row>
     <tns:lvc xsi:type="xsd:string">Hello World</tns:lvc>
     <tns:i xsi:type="xsd:int">99</tns:i>
     <tns:ii xsi:type="xsd:long">99999999</tns:ii>
     <tns:f xsi:type="xsd:float">3.25</tns:f>
     <tns:d xsi:type="xsd:double">.555555555555555582</tns:d>
     <tns:bin xsi:type="xsd:base64Binary">AAAAZg==</tns:bin>
     <tns:date xsi:type="xsd:date">2006-05-29-04:00</tns:date>
    </tns:row>
   </tns:rowset>
```

```
  </tns:test_types_XML_onResult>
  <tns:sqlcode>0</tns:sqlcode>
 </tns:test_types_XML_onResponse>
</SOAP-ENV:Body>
```

要素のネームスペースと XML スキーマでは、データ型情報だけでなく、XML パーサによる後処理に必要なすべての情報を提供します。データ型情報が結果セットに存在しない場合 (DATATYPE OFF または IN)、xsi:type と XML スキーマのネームスペース宣言は省略されます。

型指定された SimpleDataSet を返す SOAP FORMAT 'DNET' Web サービスの例を次に示します。

```
<SOAP-ENV:Body>
  <tns:test_types_dnet_outResponse>
   <tns:test_types_dnet_outResult xsi:type='sqlresultstream:SqlRowSet'>
    <xsd:schema id='Schema2'
       xmlns:xsd='http://www.w3.org/2001/XMLSchema'
       xmlns:msdata='urn:schemas-microsoft.com:xml-msdata'>
     <xsd:element name='rowset' msdata:IsDataSet='true'>
      <xsd:complexType>
       <xsd:sequence>
        <xsd:element name='row' minOccurs='0' maxOccurs='unbounded'>
         <xsd:complexType>
          <xsd:sequence>
           <xsd:element name='lvc' minOccurs='0' type='xsd:string' />
           <xsd:element name='ub' minOccurs='0' type='xsd:unsignedByte' />
           <xsd:element name='s' minOccurs='0' type='xsd:short' />
           <xsd:element name='us' minOccurs='0' type='xsd:unsignedShort' />
           <xsd:element name='i' minOccurs='0' type='xsd:int' />
           <xsd:element name='ui' minOccurs='0' type='xsd:unsignedInt' />
           <xsd:element name='l' minOccurs='0' type='xsd:long' />
           <xsd:element name='ul' minOccurs='0' type='xsd:unsignedLong' />
           <xsd:element name='f' minOccurs='0' type='xsd:float' />
           <xsd:element name='d' minOccurs='0' type='xsd:double' />
           <xsd:element name='bin' minOccurs='0' type='xsd:base64Binary' />
           <xsd:element name='bool' minOccurs='0' type='xsd:boolean' />
           <xsd:element name='num' minOccurs='0' type='xsd:decimal' />
           <xsd:element name='dc' minOccurs='0' type='xsd:decimal' />
           <xsd:element name='date' minOccurs='0' type='xsd:date' />
          </xsd:sequence>
         </xsd:complexType>
        </xsd:element>
       </xsd:sequence>
      </xsd:complexType>
     </xsd:element>
    </xsd:schema>
    <diffgr:diffgram xmlns:msdata='urn:schemas-microsoft-com:xml-msdata'
xmlns:diffgr='urn:schemas-microsoft-com:xml-diffgram-v1'>
     <rowset>
      <row>
       <lvc>Hello World</lvc>
       <ub>128</ub>
       <s>-99</s>
       <us>33000</us>
       <i>-2147483640</i>
       <ui>4294967295</ui>
       <l>-9223372036854775807</l>
       <ul>18446744073709551615</ul>
       <f>3.25</f>
       <d>.555555555555555582</d>
       <bin>QUJD</bin>
       <bool>1</bool>
       <num>123456.123457</num>
       <dc>-1.756000</dc>
       <date>2006-05-29-04:00</date>
      </row>
     </rowset>
```

```
    </diffgr:diffgram>
   </tns:test_types_dnet_outResult>
  <tns:sqlcode>0</tns:sqlcode>
  </tns:test_types_dnet_outResponse>
</SOAP-ENV:Body>
```

SQL Anywhere の型から XML スキーマの型へのマッピング

SQL Anywhere の型	XML スキーマの型	XML の例
CHAR	string	Hello World
VARCHAR	string	Hello World
LONG VARCHAR	string	Hello World
TEXT	string	Hello World
NCHAR	string	Hello World
NVARCHAR	string	Hello World
LONG NVARCHAR	string	Hello World
NTEXT	string	Hello World
UNIQUEIDENTIFIER	string	12345678-1234-5678-9012-123456789012
UNIQUEIDENTIFIERSTR	string	12345678-1234-5678-9012-123456789012
XML	これはユーザ定義の型です。パラメータは、複合型 (base64Binary、SOAP 配列、struct など) を表す有効な XML であることが想定されます。	<inputHexBinary xsi:type="xsd:hexBinary"> 414243 </inputHexBinary> (「ABC」と解釈されます)
BIGINT	long	-9223372036854775807
UNSIGNED BIGINT	unsignedLong	18446744073709551615
BIT	boolean	1
VARBIT	string	11111111
LONG VARBIT	string	00000000000000001000000000000000
DECIMAL	decimal	-1.756000
DOUBLE	double	.555555555555555582
FLOAT	float	12.3456792831420898
INTEGER	int	-2147483640
UNSIGNED INTEGER	unsignedInt	4294967295
NUMERIC	decimal	123456.123457
REAL	float	3.25

SQL Anywhere の型	XML スキーマの型	XML の例
SMALLINT	short	-99
UNSIGNED SMALLINT	unsignedShort	33000
TINYINT	unsignedByte	128
MONEY	decimal	12345678.9900
SMALLMONEY	decimal	12.3400
DATE	date	2006-11-21-05:00
DATETIME	dateTime	2006-05-21T09:00:00.000-08:00
SMALLDATETIME	dateTime	2007-01-15T09:00:00.000-08:00
TIME	time	14:14:48.980-05:00
TIMESTAMP	dateTime	2007-01-12T21:02:14.420-06:00
TIMESTAMP WITH TIME ZONE	dateTime	2007-01-12T21:02:14.420-06:00
BINARY	base64Binary	AAAAZg==
IMAGE	base64Binary	AAAAZg==
LONG BINARY	base64Binary	AAAAZg==
VARBINARY	base64Binary	AAAAZg==

1 つまたは複数のパラメータが NCHAR、NVARCHAR、LONG NVARCHAR、NTEXT のいずれかの型の場合、応答は UTF8 で出力されます。クライアントデータベースが UTF-8 文字コードを使用している場合は動作に変更はありません (NCHAR と CHAR のデータ型は同一であるため)。ただし、データベースが UTF-8 文字コードを使用していない場合は、NCHAR 以外のデータ型のパラメータはすべて UTF8 に変換されます。XML 宣言エンコードおよび HTTP ヘッダ Content-Type の値は、使用される文字コードに対応します。

XML スキーマの型から Java の型へのマッピング

XML スキーマの型	Java データ型
xsd:string	java.lang.String
xsd:integer	java.math.BigInteger
xsd:int	int
xsd:long	long
xsd:short	short
xsd:decimal	java.math.BigDecimal
xsd:float	float
xsd:double	double
xsd:boolean	boolean
xsd:byte	byte

XML スキーマの型	Java データ型
xsd:QName	javax.xml.namespace.QName
xsd:dateTime	javax.xml.datatype.XMLGregorianCalendar
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]
xsd:unsignedInt	long
xsd:unsignedShort	int
xsd:unsignedByte	short
xsd:time	javax.xml.datatype.XMLGregorianCalendar
xsd:date	javax.xml.datatype.XMLGregorianCalendar
xsd:g	javax.xml.datatype.XMLGregorianCalendar
xsd:anySimpleType	java.lang.Object
xsd:anySimpleType	java.lang.String
xsd:duration	javax.xml.datatype.Duration
xsd:NOTATION	javax.xml.namespace.QName

1.20.2.3.3.3 SOAP の構造化されたデータ型

XML データ型は、Web サービス関数とプロシージャ内のパラメータおよび戻り値として使用できます。

XML 戻り値

Web サービスクライアントとしてのデータベースは、関数やプロシージャを使用する Web サービスに対するインタフェースになることがあります。

戻り値の単純なデータ型には、結果セット内の文字列表現で十分な場合があります。この場合、ストアドプロシージャの使用が可能になります。

配列や構造体などの複雑なデータを返すときは、Web サービス関数を使用する方が適しています。関数の宣言では、RETURN 句で XML データ型を指定できます。目的の要素を抽出するために、返された XML は OPENXML を使用して解析することができます。

dateTime などの XML データの戻り値は、結果セット内にそのままで現れます。たとえば TIMESTAMP カラムが結果セットに含まれる場合は、文字列 (2006-12-25 12:00:00.000) ではなく、XML dateTime 文字列 (2006-12-25T12:00:00.000-05:00) のようにフォーマットされます。

XML パラメータ値

XML データ型は、Web サービス関数とプロシージャ内のパラメータとして使用できます。単純な型の場合、SOAP 要求の本文が生成されるときに、パラメータ要素が自動的に構成されます。ただし、XML パラメータタイプの場合、要素の XML 表現で追加のデータを提供する属性が必要になることがあるため、自動的に構成できません。そのため、データ型が XML のパラメータに対して XML を生成するときは、ルート要素の名前をパラメータ名と一致させる必要があります。

```
<inputHexBinary xsi:type="xsd:hexBinary">414243</inputHexBinary>
```

この XML 型は、パラメータを hexBinary XML 型として送信する方法の例を示しています。SOAP 終了ポイントは、パラメータ名 (XML 用語ではルート要素名) が inputHexBinary であることを想定しています。

Cookbook 定数

複雑な構造体や配列を構築するには、SQL Anywhere がネームスペースを参照する方法を知る必要があります。ここに示すプレフィクスは、SOAP 要求エンベロープ用に生成されるネームスペース宣言に対応しています。

XML プレフィクス	ネームスペース
xsd	http://www.w3.org/2001/XMLSchema
xsi	http://www.w3.org/2001/XMLSchema-instance
SOAP-ENC	http://schemas.xmlsoap.org/soap/encoding/
m	NAMESPACE 句で定義されたネームスペース

1.20.2.3.4 句の値に使用する代入パラメータ

ストアドプロシージャまたは関数の宣言済みパラメータは、そのプロシージャまたは関数が実行されるたびに、句の定義内のプレースホルダを自動的に置き換えます。

代入パラメータを使用すると、実行時に動的に句を設定する一般的な Web サービスプロシージャを作成できます。感嘆符 '!' の後に宣言されたパラメータの 1 つの名前が続いている部分文字列はすべて、パラメータの値で置換されます。こうして、実行時に 1 つ以上のパラメータ値が代入され、1 つ以上の句の値が抽出されます。

パラメータの代入を行うには、次の規則を順守する必要があります。

- 代入に使用するパラメータはすべて英数字にします。アンダースコアは使用できません。
- 代入パラメータの直後は、英数字以外の文字が続くか終了である必要があります。たとえば、!sizeXL は、X が英数字であるため、size というパラメータの値で置換されません。
- パラメータ名に一致しない代入パラメータは無視されます。
- 感嘆符 (!) は別の感嘆符でエスケープできます。

たとえば、次のプロシージャは、代入パラメータの使用方法を示します。URL と HTTP ヘッダの定義は、パラメータとして渡します。

```
CREATE PROCEDURE test(uid CHAR(128), pwd CHAR(128), headers LONG VARCHAR)
```

```
    URL 'http://!uid:!pwd@localhost/myservice'
    HEADER '!headers';
```

次の文を使用して **test** プロシージャを呼び出し、HTTP 要求を開始できます。

```
CALL test('dba', 'sql', 'NewHeader1:value1¥nNewHeader2:value2');
```

このプロシージャが呼び出されるたびに、異なる値を使用できます。

暗号化証明書の例

代入パラメータを使用してファイルからストアドプロシージャまたはストアド関数に暗号化証明書を渡すことができます。

次の例は、証明書を代入文字列として渡す方法を示します。

```
CREATE PROCEDURE secure(cert LONG VARCHAR)
URL 'https://localhost/secure'
TYPE 'HTTP:GET'
CERTIFICATE 'cert=!cert;company=test;unit=test;name=RSA Root';
```

証明書は次の呼び出しでファイルから読み込まれて **secure** に渡されます。

```
CALL secure( xp_read_file('C:¥¥Users¥¥Public¥¥Documents¥¥SQL Anywhere 17¥¥Samples¥
¥Certificates¥¥rsaroot.crt') );
```

この例は説明でのみ使用されます。CERTIFICATE 句の *file=* キーワードを使用して、証明書をファイルから直接読み込むことができます。

一致するパラメータ名がない場合の例

一致するパラメータ名のないプレースホルダは、自動的に削除されます。

たとえば、次のプロシージャでは、パラメータ size はプレースホルダを置換しません。

```
CREATE PROCEDURE orderitem (size CHAR(18))
    URL 'HTTP://localhost/salesserver/order?size=!sizeXL'
    TYPE 'SOAP:RPC';
```

この例では、!sizeXL は一致するパラメータがないプレースホルダであるため、常に削除されます。

パラメータはストアド関数またはストアドプロシージャの呼び出し時に、それらの本文内のプレースホルダを置換するためにも使用できます。特定の変数のプレースホルダが存在しない場合、パラメータとその値が要求の一部として渡されます。このようにして代入に使用されるパラメータは、要求の一部として渡されません。

1.20.2.4 HTTP 要求と SOAP 要求の構造

パラメータの代入中に使用する場合を除き、関数またはプロシージャのすべてのパラメータは、Web サービス要求の一部として渡されます。渡されるときのフォーマットは、Web サービス要求のタイプによって異なります。

文字またはバイナリデータ型でないパラメータ値は、要求に追加する前に文字列表現に変換されます。この処理は、値を文字型にキャストすることに相当します。変換は、関数またはプロシージャの呼び出し時に、データ型のフォーマットオプションの設定に従って行われます。具体的には、変換は precision、scale、timestamp_format などのオプションによって影響されます。

HTTP 要求の構造

HTTP:GET タイプのパラメータは URL コード化され、URL 内に配置されます。パラメータ名は、HTTP 変数の名前としてそのまま使用されます。たとえば、次のプロシージャは 2 つのパラメータを宣言します。

```
CREATE PROCEDURE test(a INTEGER, b CHAR(128))
    URL 'HTTP://localhost/myservice'
    TYPE 'HTTP:GET';
```

123 と 'xyz' という値を使ってこのプロシージャを呼び出す場合、要求に使用する URL は次に示したものと同等になります。

```
HTTP://localhost/myservice?a=123&b=xyz
```

タイプが HTTP:POST である場合、パラメータとその値は URL コード化され、要求の本文内に配置されます。2 つのパラメータと値の場合、ヘッダの後に、次のテキストが HTTP 要求の本文に表示されます。

```
a=123&b=xyz
```

SOAP 要求の構造

SOAP 要求に渡されたパラメータは、SOAP 仕様で指定されているように、要求本文の一部としてひとまとめにされます。

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:m="http://localhost:8082">
  <SOAP-ENV:Body>
    <m:test>
      <m:a>123</m:a>
      <m:b>abc</m:b>
    </m:test>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

関連情報

句の値に使用する代入パラメータ [658 ページ]

1.20.2.5 Web クライアント要求のロギング方法

HTTP 要求やトランスポートデータを含む Web サービスクライアントの情報は、Web サービスクライアントログファイルに記録できます。

Web サービスクライアントログファイルは、-zoc サーバオプションまたは sa_server_option システムプロシージャを使用して指定できます。

```
CALL sa_server_option( 'WebClientLogFile', 'clientinfo.txt' );
```

-zoc サーバオプションを指定すると、ロギングが自動的に有効になります。このファイルへのロギングの有効/無効を切り替えるには、sa_server_option システムプロシージャを使用します。

```
CALL sa_server_option( 'WebClientLogging', 'ON' );
```

1.20.3 Web サービスエラーコードリファレンス

要求に失敗すると、HTTP サーバで標準の Web サービスエラーが生成されます。これらのエラーには、プロトコル標準と一貫性のある番号が割り当てられています。

発生する可能性のある一般的なエラーは次のとおりです。

番号	名前	SOAP フォールト	説明
301	Moved permanently	Server	要求されたページは永続的に移動されました。サーバは自動的に、新しいロケーションに要求をリダイレクトします。
304	Not Modified	Server	サーバは、要求の情報に基づき、要求されたデータは前回の要求の後変更されていないため、再度送信する必要はないと判断しました。
307	Temporary Redirect	Server	要求されたページは移動されましたが、この変更は永続的なものではない可能性があります。サーバは自動的に、新しいロケーションに要求をリダイレクトします。
400	Bad Request	Client.BadRequest	HTTP 要求が正しくないか不正です。

番号	名前	SOAP フォールト	説明
401	Authorization Required	Client.Authorization	サービスを使用するのに認証が必要ですが、有効なユーザ名とパスワードが入力されていません。
403	Forbidden	Client.Forbidden	データベースにアクセスするパーミッションがありません。
404	Not Found	Client.NotFound	指定したデータベースがサーバで実行されていないか、指定した Web サービスが存在しません。
408	Request Timeout	Server.Timeout	要求の受信中に最大接続アイドル時間が超過しました。
410	Gone	Server	要求したリソースを使用できません。
411	HTTP Length Required	Client.LengthRequired	サーバは、クライアントが要求に Content-Length の指定を含めることを必要とします。通常、このエラーはデータをサーバにアップロードしているときに発生します。
413	Entity Too Large	Server	要求が最大許可サイズを超過しました。
414	URI Too Large	Server	URI の長さが最大長を超過しました。
500	Internal Server Error	Server	内部エラーが発生しました。要求が処理できませんでした。
501	Not Implemented	Server	HTTP 要求メソッドが GET、HEAD、または POST ではありません。
502	Bad Gateway	Server	要求されたドキュメントがサードパーティのサーバにあり、サーバがサードパーティのサーバからエラーを受け取りました。
503	Service Unavailable	Server	接続数が最大数を超過しました。

SOAP サービスが失敗すると、次の SOAP バージョン 1.1 標準で定義されているように、フォールトがクライアントに対して SOAP フォールトとして返されます。

- 要求を処理するアプリケーションのエラーによって SQLCODE が生成されると、クライアントの faultcode により SOAP フォールトが返されます。その場合、Procedure などのサブカテゴリが含まれることもあります。SOAP フォールト内の faultstring 要素には、エラーの詳しい説明が設定され、detail 要素には、数値の SQLCODE 値が指定されます。
- トランスポートプロトコルエラーが発生した場合、faultcode はエラーに応じて Client または Server に設定され、faultstring には `404 Not Found` などの HTTP トランスポートメッセージが設定され、detail 要素には数値の HTTP エラー値が設定されます。

- SQLCODE 値を返すアプリケーションエラーのために生成された SOAP フォールトメッセージは、`200 OK` という HTTP ステータスで返されます。

クライアントを SOAP クライアントとして識別できない場合は、生成された HTML ドキュメントで適切な HTTP エラーが返されます。

1.20.4 HTTP Web サービスの例

Web サービスの実装サンプルのいくつかは、*%SQLANYSAMP17%*`¥SQLAnywhere¥HTTP` フォルダにあります。

サンプルの詳細については、*%SQLANYSAMP17%*`¥SQLAnywhere¥HTTP¥readme.txt` を参照してください。

このセクションの内容:

チュートリアル:Web サーバを作成し、Web クライアントからアクセスします。[663 ページ]
このチュートリアルでは、Web サーバを作成し、Web クライアントデータベースサーバから要求を送信する方法について説明します。

チュートリアル: データベースサーバを使用した SOAP/DISH サービスへのアクセス [668 ページ]
このチュートリアルでは、Web クライアントが指定した華氏の値を摂氏に変換する SOAP サーバの作成方法について説明します。

チュートリアル:Microsoft Visual C# を使用した SOAP/DISH Web サービスへのアクセス [678 ページ]
このチュートリアルでは、Web サーバとして動作するデータベースサーバ上の SOAP/DISH サービスにアクセスする Microsoft Visual C# クライアントアプリケーションの作成方法について説明します。

チュートリアル: JAX-WS を使用した SOAP/DISH Web サービスへのアクセス [686 ページ]
このチュートリアルでは、Web サーバとして動作するデータベースサーバ上の SOAP/DISH サービスにアクセスする XML Web サービス (JAX-WS) クライアントアプリケーション用の Java API の作成方法について説明します。

1.20.4.1 チュートリアル:Web サーバを作成し、Web クライアントからアクセスします。

このチュートリアルでは、Web サーバを作成し、Web クライアントデータベースサーバから要求を送信する方法について説明します。

前提条件

次の項目が必要です。

- XML の知識
- MIME (Multipurpose Internet Mail Extensions) タイプの知識
- ソフトウェアの Web サービス機能の基本的な知識

このチュートリアルのレッスンを実行するには、次の権限が必要です。

- CREATE ANY OBJECT
- MANAGE ANY WEB SERVICE

コンテキスト

このチュートリアルでは、次の作業の方法について説明します。

- 新しい Web サービスデータベースを作成し、起動します。データベースサーバは Web サーバとして動作します。
- Web サービスを作成します。
- HTTP 要求に含まれている情報を返すプロシージャを設定します。
- 新しい Web クライアントデータベースを作成し、起動します。データベースサーバは Web クライアントとして動作します。
- HTTP:POST 要求を、Web クライアントから Web サーバに送信します。
- HTTP 応答を、Web サーバから Web クライアントに送信します。

1. レッスン 1: 要求を受信して応答を送信する Web サーバの設定 [664 ページ]
   Web サービスを実行する Web サーバを設定します。
2. レッスン 2:Web クライアントからの要求の送信と応答の受信 [666 ページ]
   POST メソッドを使用して要求を Web サーバに送信し、Web サーバの応答を受信するデータベースクライアントを設定します。

関連情報

HTTP Web サーバとしてのデータベースサーバ [587 ページ]

1.20.4.1.1 レッスン 1: 要求を受信して応答を送信する Web サーバの設定

Web サービスを実行する Web サーバを設定します。

前提条件

このチュートリアルの冒頭に一覧されているロールと権限を持っている必要があります。

手順

1. Web サービス定義を含めるために使用するデータベースを作成します。

   ```
   dbinit -dba DBA,passwd echo
   ```

2. このデータベースを使用して、ネットワークデータベースサーバを起動します。このサーバは Web サーバとして動作します。

   ```
   dbsrv17 -xs http(port=8082) -n echo echo.db
   ```

   HTTP Web サーバは、ポート 8082 で要求を受信するように設定されます。ネットワークで 8082 が許可されない場合は、異なるポート番号を使用します。

3. Interactive SQL を使用して、データベースサーバに接続します。

   ```
   dbisql -c "UID=DBA;PWD=passwd;SERVER=echo"
   ```

4. 着信要求を受け入れる新しい Web サービスを作成します。

   ```
   CREATE SERVICE EchoService
   TYPE 'RAW'
   AUTHORIZATION OFF
   SECURE OFF
   USER DBA
   AS CALL Echo();
   ```

   この文は、Web クライアントがサービスに要求を送ると、Echo という名前のストアドプロシージャを呼び出す EchoService という名前の新しいサービスを作成します。Web クライアントのフォーマットがない (RAW) HTTP 応答本文を生成します。別のユーザ ID でログインする場合、そのユーザ ID を反映するように USER DBA を変更する必要があります。

5. 着信要求を処理する Echo プロシージャを作成します。

   ```
   CREATE OR REPLACE PROCEDURE Echo()
   BEGIN
       DECLARE request_body LONG VARCHAR;
       DECLARE request_mimetype LONG VARCHAR;
       SET request_mimetype = http_header( 'Content-Type' );
       SET request_body = isnull( http_variable('text'), http_variable('body') );
       IF request_body IS NULL THEN
           CALL sa_set_http_header('Content-Type', 'text/plain' );
           SELECT 'failed'
       ELSE
           CALL sa_set_http_header('Content-Type', request_mimetype );
           SELECT request_body;
       END IF;
   END
   ```

   このプロシージャは、Content-Type ヘッダと Web クライアントに送信される応答の本文をフォーマットします。

結果

Web サーバは要求を受信して応答を送信するように設定されます。

次のステップ

次のレッスンに進みます。

タスクの概要: チュートリアル:Web サーバを作成し、Web クライアントからアクセスします。[663 ページ]

次のタスク: レッスン 2:Web クライアントからの要求の送信と応答の受信 [666 ページ]

関連情報

HTTP Web サーバとしてのデータベースサーバ [587 ページ]

1.20.4.1.2 レッスン 2:Web クライアントからの要求の送信と応答の受信

POST メソッドを使用して要求を Web サーバに送信し、Web サーバの応答を受信するデータベースクライアントを設定します。

前提条件

このチュートリアルのこれまでのレッスンを完了している必要があります。

このチュートリアルの冒頭に一覧されているロールと権限を持っている必要があります。

コンテキスト

このレッスンには、localhost への複数の参照が含まれています。Web クライアントを Web サーバと同じコンピュータで実行していない場合、localhost ではなく、前のレッスンの Web サーバのホスト名または IP アドレスを使用してください。

手順

1. Web クライアントプロシージャを含めるために使用することができるデータベースを作成します。

```
dbinit -dba DBA,passwd echo_client
```

2. ネットワークデータベースサーバでデータベースを開始します。このデータベースサーバは Web クライアントとして動作します。

```
dbsrv17 echo_client.db
```

3. Interactive SQL からデータベースサーバに接続します。

```
dbisql -c "UID=DBA;PWD=passwd;SERVER=echo_client"
```

4. 要求を Web サービスに送信する新しいストアドプロシージャを作成します。

```
CREATE OR REPLACE PROCEDURE SendWithMimeType(
    value LONG VARCHAR,
    mimeType LONG VARCHAR,
    urlSpec LONG VARCHAR
)
URL '!urlSpec'
TYPE 'HTTP:POST:!mimeType';
```

SendWithMimeType プロシージャには 3 つのパラメータがあります。*value* パラメータは、Web サービスに送信する要求の本文を表します。urlSpec パラメータは、Web サービスに接続するために使用する URL を示します。mimeType は、HTTP:POST に使用する MIME タイプを示します。

5. 要求を Web サーバに送信し、応答を取得します。

```
CALL SendWithMimeType('<hello>this is xml</hello>',
    'text/xml',
    'http://localhost:8082/EchoService'
);
```

http://localhost:8082/EchoService 文字列は、localhost で実行されポート 8082 で受信する Web サーバを示します。対象となる Web サービスの名前は EchoService です。

6. 異なる MIME タイプを試し、応答を確認します。

```
CALL SendWithMimeType('{"menu": {  "id": "file", "value": "File", "popup": {
    "menuitem": [{"value": "New", "onclick": "CreateNew()"},
                 {"value": "Open", "onclick": "Open()"},
                 {"value": "Close", "onclick": "Close()"} ] } } }',
    'application/json',
    'http://localhost:8082/EchoService'
);
```

結果

Web クライアントは、POST メソッドを使用して HTTP 要求を Web サーバに送信し、Web サーバの応答を受信するように設定されます。

例

次は、Interactive SQL によって表示される結果セットの例です。

属性	値	インスタンス
Status	HTTP/1.1 200 OK	1

属性	値	インスタンス
Body	<hello>this is xml</hello>	1
Server	SQLAnywhere/17.0.4.1234	1
Expires	Tue, 09 Oct 2012 21:06:01 GMT	1
Date	Tue, 09 Oct 2012 21:06:01 GMT	1
Content-Type	text/xml; charset=windows-1252	1
Connection	close	1

次のテキストは、Web サーバに送信される HTTP パケットを示します。

```
POST /EchoService HTTP/1.0
ASA-Id: 46758096650a44088c77237cc8719d5c
User-Agent: SQLAnywhere/17.0.4.1234
Accept-Charset: windows-1252, UTF-8, *
Date: Tue, 09 Oct 2012 21:06:01 GMT
Host: localhost:8082
Connection: close
Content-Type: text/xml; charset=windows-1252
Content-Length: 26
<hello>this is xml</hello>
```

次のテキストは、Web サーバからの応答を示します。

```
HTTP/1.1 200 OK
Date: Tue, 09 Oct 2012 21:06:01 GMT
Connection: close
Expires: Tue, 09 Oct 2012 21:06:01 GMT
Content-Type: text/xml; charset=windows-1252
Server: SQLAnywhere/17.0.4.1234
<hello>this is xml</hello>
```

タスクの概要: チュートリアル:Web サーバを作成し、Web クライアントからアクセスします。[663 ページ]

前のタスク: レッスン 1: 要求を受信して応答を送信する Web サーバの設定 [664 ページ]

1.20.4.2 チュートリアル: データベースサーバを使用した SOAP/DISH サービスへのアクセス

このチュートリアルでは、Web クライアントが指定した華氏の値を摂氏に変換する SOAP サーバの作成方法について説明します。

前提条件

次の項目が必要です。

- SOAP の知識
- ソフトウェアの Web サービス機能の基本的な知識

このチュートリアルのレッスンを実行するには、次の権限が必要です。

- CREATE ANY OBJECT
- MANAGE ANY WEB SERVICE
- SERVER OPERATOR

コンテキスト

このチュートリアルでは、次の作業の方法について説明します。

- 新しい Web サービスデータベースを作成し、起動します。データベースサーバは Web サーバとして動作します。
- SOAP Web サービスを作成します。
- クライアントが指定した華氏の値を摂氏の値に変換するプロシージャを設定します。
- 新しい Web クライアントデータベースを作成し、起動します。データベースサーバは Web クライアントとして動作します。
- SOAP 要求を、Web クライアントから Web サーバに送信します。
- SOAP 応答を、Web サーバから Web クライアントに送信します。

1. レッスン 1: SOAP 要求を受信し SOAP 応答を送信する Web サーバの設定 [670 ページ]
   新しいデータベースサーバを設定し、着信 SOAP 要求を処理する SOAP サービスを作成します。
2. レッスン 2: SOAP 要求を送信し SOAP 応答を受信する Web クライアントの設定 [673 ページ]
   SOAP 要求を送信し SOAP 応答を受信する Web クライアントを設定します。
3. レッスン 3: SOAP 要求の送信と SOAP 応答の受信 [676 ページ]
   前のレッスンで作成したラッパープロシージャを呼び出します。このプロシージャは、レッスン 1 で作成した Web サーバに SOAP 要求を送信します。

関連情報

HTTP Web サーバとしてのデータベースサーバ [587 ページ]

1.20.4.2.1 レッスン 1: SOAP 要求を受信し SOAP 応答を送信する Web サーバの設定

新しいデータベースサーバを設定し、着信 SOAP 要求を処理する SOAP サービスを作成します。

前提条件

このチュートリアルの冒頭に一覧されているロールと権限を持っている必要があります。

コンテキスト

Web サーバは対応する摂氏の値に変換される華氏の値を提供する SOAP 要求を予測します。

手順

1. Web サービス定義を含めるために使用するデータベースを作成します。

   ```
   dbinit -dba DBA,passwd ftc
   ```

2. このデータベースを使用して、データベースサーバを起動します。このサーバは Web サーバとして動作します。

   ```
   dbsrv17 -xs http(port=8082) -n ftc ftc.db
   ```

   HTTP Web サーバは、ポート 8082 で要求を受信するように設定されます。ネットワークで 8082 が許可されない場合は、異なるポート番号を使用します。

3. Interactive SQL を使用して、データベースサーバに接続します。

   ```
   dbisql -c "UID=DBA;PWD=passwd;SERVER=ftc"
   ```

4. 着信要求を受け入れる新しい DISH サービスを作成します。

   ```
   CREATE SERVICE soap_endpoint
       TYPE 'DISH'
       AUTHORIZATION OFF
       SECURE OFF
       USER DBA;
   ```

   この文は、着信 SOAP サービス要求を処理する soap_endpoint という新しい DISH サービスを作成します。別のユーザ ID でログインする場合、そのユーザ ID を反映するように USER DBA を変更する必要があります。

5. 華氏から摂氏への変換を処理する新しい SOAP サービスを作成します。

   ```
   CREATE SERVICE FtoCService
       TYPE 'SOAP'
       FORMAT 'XML'
       AUTHORIZATION OFF
   ```

```
    USER DBA
    AS CALL FToCConverter( :fahrenheit );
```

この文は、XML 形式の文字列を出力として生成する FtoCService という新しい SOAP サービスを作成します。Web クライアントがサービスに SOAP 要求を送信すると、FToCConverter というストアドプロシージャが呼び出されます。別のユーザ ID でログインする場合、そのユーザ ID を反映するように USER DBA を変更する必要があります。

6. 着信 SOAP 要求を処理する FToCConverter プロシージャを作成します。このプロシージャは、クライアントが指定した華氏の値を対応する摂氏の値に変換するための必要な計算を実行します。

```
CREATE OR REPLACE PROCEDURE FToCConverter( temperature FLOAT )
BEGIN
    DECLARE hd_key LONG VARCHAR;
    DECLARE hd_entry LONG VARCHAR;
    DECLARE alias LONG VARCHAR;
    DECLARE first_name LONG VARCHAR;
    DECLARE last_name LONG VARCHAR;
    DECLARE xpath LONG VARCHAR;
    DECLARE authinfo LONG VARCHAR;
    DECLARE namespace LONG VARCHAR;
    DECLARE mustUnderstand LONG VARCHAR;
header_loop:
    LOOP
        SET hd_key = NEXT_SOAP_HEADER( hd_key );
        IF hd_key IS NULL THEN
            -- no more header entries
            LEAVE header_loop;
        END IF;
        IF hd_key = 'Authentication' THEN
            SET hd_entry = SOAP_HEADER( hd_key );
            SET xpath = '/*:' || hd_key || '/*:userName';
            SET namespace = SOAP_HEADER( hd_key, 1, '@namespace' );
            SET mustUnderstand = SOAP_HEADER( hd_key, 1, 'mustUnderstand' );
            BEGIN
                -- parse the XML returned in the SOAP header
                DECLARE crsr CURSOR FOR
                    SELECT * FROM OPENXML( hd_entry, xpath )
                        WITH ( alias LONG VARCHAR '@*:alias',
                            first_name LONG VARCHAR '*:first/text()',
                            last_name LONG VARCHAR '*:last/text()' );
                OPEN crsr;
                FETCH crsr INTO alias, first_name, last_name;
                CLOSE crsr;
            END;
            -- build a response header
            -- based on the pieces from the request header
            SET authinfo =
                XMLELEMENT( 'Authentication',
                    XMLATTRIBUTES(
                        namespace as xmlns,
                        alias,
                        mustUnderstand ),
                        XMLELEMENT( 'first', first_name ),
                        XMLELEMENT( 'last', last_name ) );
            CALL SA_SET_SOAP_HEADER( 'authinfo', authinfo );
        END IF;
    END LOOP header_loop;
    SELECT ROUND((temperature - 32.0) * 5.0 / 9.0, 5) AS answer;
END;
```

NEXT_SOAP_HEADER 関数は、SOAP 要求に含まれるすべてのヘッダ名を繰り返すために LOOP 構造で使用され、NEXT_SOAP_HEADER 関数から NULL が返されるとループが終了します。

> **注記**
>
> この関数は、必ずしも SOAP 要求に表示される順序でヘッダを繰り返すとはかぎりません。

SOAP_HEADER 関数は、ヘッダ値またはヘッダ名が存在しない場合は NULL を返します。FToCConverter プロシージャは、Authentication というヘッダ名を検索し、@namespace 属性と mustUnderstand 属性を含めてヘッダ構造を抽出します。@namespace ヘッダ属性は、特定のヘッダエントリの名前空間 (xmlns) にアクセスするときに使用する特殊な属性です。

次に、可能な Authentication ヘッダ構造の XML 文字列表現を示します。@namespace 属性の値は "SecretAgent"、mustUnderstand の値は 1 です。

```
<Authentication xmlns="SecretAgent" mustUnderstand="1">
    <userName alias="99">
        <first>Susan</first>
        <last>Hilton</last>
    </userName>
</Authentication>
```

SELECT 文の OPENXML システムプロシージャでは、XPath 文字列 "/*:Authentication/*:userName" を使用して XML ヘッダを解析し、**alias** 属性値と、**first** および **last** タグの内容を抽出します。カーソルを使用して結果セットを処理し、3 つのカラム値をフェッチします。

この時点で、Web サービスに渡された関連性のある情報すべてが手中にあります。華氏表現された温度が取得され、Web サービスに渡されたいくつかの追加属性が SOAP ヘッダから取得されています。たとえば、取得した名前と別名 (alias) を照会して、該当人物が Web サービスの使用を許可されているかどうかを確認できます。ただし、この演習でその例は取り上げていません。

SET 文は、クライアントに送信する SOAP 応答を XML 形式で構築するために使用されます。次に、可能な SOAP 応答の XML 文字列表現を示します。これは、上記の Authentication ヘッダ構造の例に基づいています。

```
<Authentication xmlns="SecretAgent" alias="99" mustUnderstand="1">
  <first>Susan</first>
  <last>Hilton</last>
</Authentication>
```

SA_SET_SOAP_HEADER システムプロシージャは、クライアントに送信される SOAP 応答ヘッダを設定するために使用されます。

最後の SELECT 文は、指定された華氏の値を摂氏の値に変換するために使用されます。この情報は、クライアントに戻されます。

結果

華氏から摂氏への温度変換を行うサービスを提供する Web サーバが実行されます。このサービスは、クライアントからの SOAP ヘッダを処理して SOAP 応答をクライアントに送り返します。

次のステップ

次のレッスンに進みます。

タスクの概要: チュートリアル: データベースサーバを使用した SOAP/DISH サービスへのアクセス [668 ページ]

次のタスク: レッスン 2: SOAP 要求を送信し SOAP 応答を受信する Web クライアントの設定 [673 ページ]

関連情報

DISH サービスを作成する方法 [597 ページ]
SOAP ネームスペース URI の要件 [643 ページ]

1.20.4.2.2 レッスン 2: SOAP 要求を送信し SOAP 応答を受信する Web クライアントの設定

SOAP 要求を送信し SOAP 応答を受信する Web クライアントを設定します。

前提条件

このチュートリアルのこれまでのレッスンを完了している必要があります。

このチュートリアルの冒頭に一覧されているロールと権限を持っている必要があります。

コンテキスト

このレッスンには、localhost への複数の参照が含まれています。Web クライアントを Web サーバと同じコンピュータで実行していない場合は、localhost の代わりにレッスン 1 の Web サーバのホスト名または IP アドレスを使用します。

手順

1. 次のコマンドを実行して、Web クライアントデータベースを作成します。

```
dbinit -dba DBA,passwd ftc_client
```

2. 次のコマンドを使用してデータベースクライアントを起動します。

```
dbsrv17 ftc_client.db
```

3. 次のコマンドを使用して Interactive SQL でデータベースに接続します。

```
dbisql -c "UID=DBA;PWD=passwd;SERVER=ftc_client"
```

4. SOAP 要求を DISH サービスに送信する新しいストアドプロシージャを作成します。

   Interactive SQL で次の SQL 文を実行します。

```
CREATE OR REPLACE PROCEDURE FtoC( fahrenheit FLOAT,
    INOUT inoutheader LONG VARCHAR,
    IN inheader LONG VARCHAR )
  URL 'http://localhost:8082/soap_endpoint'
  SET 'SOAP(OP=FtoCService)'
  TYPE 'SOAP:DOC'
  SOAPHEADER '!inoutheader!inheader';
```

   URL 句の http://localhost:8082/soap_endpoint 文字列は、localhost で実行されポート 8082 で受信する Web サーバを示します。対象となる DISH Web サービスの名前は soap_endpoint であり、SOAP 終了ポイントとして動作します。

   SET 句は、呼び出す SOAP 処理の名前、またはサービス FtoCService を指定します。

   Web サービス要求作成時のデフォルトフォーマットは 'SOAP:RPC' です。この例で使用されているフォーマットは 'SOAP:DOC' です。これは 'SOAP:RPC' と似ていますが、より多くのデータ型を使用できます。SOAP 要求は必ず XML ドキュメントとして送信されます。SOAP 要求の送信メカニズムは 'HTTP:POST' です。

   FtoC のような Web サービスクライアントプロシージャの代入変数 (inoutheader、inheader) は英数字である必要があります。Web サービスクライアントが関数として宣言された場合、すべてのパラメータは IN モードのみになります (呼び出された側の関数では代入できません)。したがって、SOAP 応答ヘッダ情報を抽出するには、OPENXML またはその他の文字列関数を使用する必要があります。

5. 2 つの特殊な SOAP 要求ヘッダエントリを構築するラッパープロシージャを作成し、それらを FtoC プロシージャに渡して、サーバ応答を処理します。

   Interactive SQL で次の SQL 文を実行します。

```
CREATE OR REPLACE PROCEDURE FahrenheitToCelsius( Fahrenheit FLOAT )
BEGIN
  DECLARE io_header LONG VARCHAR;
  DECLARE in_header LONG VARCHAR;
  DECLARE result LONG VARCHAR;
  DECLARE err INTEGER;
  DECLARE crsr CURSOR FOR
    CALL FtoC( Fahrenheit, io_header, in_header );
  SET io_header =
    '<Authentication xmlns="SecretAgent" ' ||
    'mustUnderstand="1">' ||
    '<userName alias="99">' ||
    '<first>Susan</first><last>Hilton</last>' ||
    '</userName>' ||
    '</Authentication>';
  SET in_header =
    '<Session xmlns="SomeSession">' ||
    '123456789' ||
    '</Session>';
  MESSAGE 'send, soapheader=' || io_header || in_header;
  OPEN crsr;
```

```
  FETCH crsr INTO result, err;
  CLOSE crsr;
  MESSAGE 'receive, soapheader=' || io_header;
  SELECT Fahrenheit, Celsius
      FROM OPENXML(result, '//tns:answer', 1, result)
      WITH ("Celsius" FLOAT 'text()');
END;
```

最初の SET 文は、Web サーバにユーザクレデンシャルを通知する、SOAP ヘッダエントリの XML 表現を作成します。

```
<Authentication xmlns="SecretAgent" mustUnderstand="1">
  <userName alias="99">
    <first>Susan</first>
    <last>Hilton</last>
  </userName>
</Authentication>
```

2 番目の SET 文は、クライアントセッション ID を追跡する、SOAP ヘッダエントリの XML 表現を作成します。

```
<Session xmlns="SomeSession">123456789</Session>
```

6. OPEN 文によって FtoC プロシージャが呼び出されます。このプロシージャで、SOAP 要求が Web サーバに送信された後、Web サーバからの応答が処理されます。応答に含まれているヘッダは inoutheader に返されます。

結果

この時点で、SOAP 要求を Web サーバに送信し、SOAP 応答を Web サーバから受信できるクライアントが作成されました。

次のステップ

次のレッスンに進みます。

タスクの概要: チュートリアル: データベースサーバを使用した SOAP/DISH サービスへのアクセス [668 ページ]

前のタスク: レッスン 1: SOAP 要求を受信し SOAP 応答を送信する Web サーバの設定 [670 ページ]

次のタスク: レッスン 3: SOAP 要求の送信と SOAP 応答の受信 [676 ページ]

1.20.4.2.3 レッスン 3: SOAP 要求の送信と SOAP 応答の受信

前のレッスンで作成したラッパープロシージャを呼び出します。このプロシージャは、レッスン 1 で作成した Web サーバに SOAP 要求を送信します。

前提条件

このチュートリアルのこれまでのレッスンを完了している必要があります。

このチュートリアルの冒頭に一覧されているロールと権限を持っている必要があります。

手順

1. レッスン 2 の接続が開かれていない場合は、Interactive SQL でクライアントデータベースに接続します。

   ```
   dbisql -c "UID=DBA;PWD=passwd;SERVER=ftc_client"
   ```

2. SOAP 要求と SOAP 応答のロギングを有効にします。

   Interactive SQL で次の SQL 文を実行します。

   ```
   CALL sa_server_option('WebClientLogFile', 'soap.txt');
   CALL sa_server_option('WebClientLogging', 'ON');
   ```

   これらの呼び出しによって、SOAP 要求と SOAP 応答の内容を調べることができるようになります。要求と応答のログは `soap.txt` というファイルに記録されます。

3. ラッパープロシージャを呼び出して、SOAP 要求を送信し、SOAP 応答を受信します。

   Interactive SQL で次の SQL 文を実行します。

   ```
   CALL FahrenheitToCelsius(212);
   ```

   この呼び出しでは、華氏の値 212 が FahrenheitToCelsius プロシージャに渡されます。このプロシージャは、カスタマイズされた 2 つの SOAP ヘッダとともに値を FToC プロシージャに渡します。クライアント側のどちらのプロシージャも、前のレッスンで作成されています。

結果

FToC Web サービスプロシージャは、華氏の値と SOAP ヘッダを Web サーバに送信します。SOAP 要求には次の情報が含まれています。

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
  xmlns:m="http://localhost:8082">
  <SOAP-ENV:Header>
    <Authentication xmlns="SecretAgent" mustUnderstand="1">
      <userName alias="99">
        <first>Susan</first>
        <last>Hilton</last>
      </userName>
    </Authentication>
    <Session xmlns="SomeSession">123456789</Session>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <m:FtoCService>
      <m:fahrenheit>212</m:fahrenheit>
    </m:FtoCService>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

次に、FtoC プロシージャは Web サーバから応答を受信します。応答には、華氏の値に基づく結果セットが含まれています。SOAP 応答には次の情報が含まれています。

```
<SOAP-ENV:Envelope
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:tns='http://localhost:8082'>
  <SOAP-ENV:Header>
    <Authentication xmlns="SecretAgent" alias="99" mustUnderstand="1">
      <first>Susan</first>
      <last>Hilton</last>
    </Authentication>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <tns:FtoCServiceResponse>
      <tns:FtoCServiceResult xsi:type='xsd:string'>
        &lt;tns:rowset xmlns:tns=&quot;http://localhost:8082/ftc&quot;&gt;&#x0A;
        &lt;tns:row&gt;&#x0A;
        &lt;tns:answer&gt;100
        &lt;/tns:answer&gt;&#x0A;
        &lt;/tns:row&gt;&#x0A;
        &lt;/tns:rowset&gt;&#x0A;
      </tns:FtoCServiceResult>
      <tns:sqlcode>0</tns:sqlcode>
    </tns:FtoCServiceResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

<SOAP-ENV:Header> の内容は inoutheader に返されます。

SOAP 応答を調べてみると、結果セットが FToCService Web サービスによって応答の中でエンコードされていることがわかります。結果セットはデコードされて FahrenheitToCelsius プロシージャに返されます。華氏の値 212 が Web サーバに渡された場合の結果セットは次のようになります。

```
<tns:rowset xmlns:tns="http://localhost:8082/ftc">
  <tns:row>
    <tns:answer>100
    </tns:answer>
  </tns:row>
</tns:rowset>
```

FahrenheitToCelsius プロシージャの SELECT 文では、OPENXML 演算子を使用して SOAP 応答が解析され、tns:answer 構造体で定義された摂氏の値が抽出されます。

次の結果セットが Interactive SQL に生成されます。

```
Fahrenheit   Celsius
        212       100
```

例

次に、華氏の温度を摂氏の値に変換する SOAP Web サービスに対する別のサンプル呼び出しを示します。

```
CALL FahrenheitToCelsius(32);
```

次の結果セットが Interactive SQL に生成されます。

```
Fahrenheit   Celsius
         32         0
```

タスクの概要: チュートリアル: データベースサーバを使用した SOAP/DISH サービスへのアクセス [668 ページ]

前のタスク: レッスン 2: SOAP 要求を送信し SOAP 応答を受信する Web クライアントの設定 [673 ページ]

1.20.4.3 チュートリアル:Microsoft Visual C# を使用した SOAP/DISH Web サービスへのアクセス

このチュートリアルでは、Web サーバとして動作するデータベースサーバ上の SOAP/DISH サービスにアクセスする Microsoft Visual C# クライアントアプリケーションの作成方法について説明します。

前提条件

次の知識が必要です。

- SOAP の知識
- .NET フレームワークの知識
- ソフトウェアの Web サービス機能の基本的な知識

次のソフトウェアが必要です。

- Microsoft Visual Studio

このチュートリアルのレッスンを実行するには、次の権限が必要です。

- CREATE ANY OBJECT
- MANAGE ANY WEB SERVICE

コンテキスト

このチュートリアルでは、次の作業の方法について説明します。

- 新しい Web サービスデータベースを作成し、起動します。データベースサーバは Web サーバとして動作します。
- SOAP Web サービスを作成します。
- SOAP 要求に含まれている情報を返すプロシージャを設定します。
- WSDL ドキュメントを提供し、プロキシとして機能する DISH Web サービスを作成します。
- クライアントコンピュータで Microsoft Visual C# を設定し、Web サーバから WSDL ドキュメントをインポートします。
- WSDL ドキュメントの情報を使用して SOAP サービスから情報を取得する Java クライアントアプリケーションを作成します。

1. レッスン 1:SOAP 要求を受信し SOAP 応答を送信する Web サーバの設定 [679 ページ]
   Microsoft Visual C# クライアントアプリケーションの要求を処理する SOAP および DISH Web サービスが実行されている Web サーバが設定されました。
2. レッスン 2:Web サーバと通信するための Microsoft Visual C# アプリケーションの作成 [681 ページ]
   Web サーバと通信するための Microsoft Visual C# アプリケーションを作成します。

関連情報

HTTP Web サーバとしてのデータベースサーバ [587 ページ]

1.20.4.3.1 レッスン 1:SOAP 要求を受信し SOAP 応答を送信する Web サーバの設定

Microsoft Visual C# クライアントアプリケーションの要求を処理する SOAP および DISH Web サービスが実行されている Web サーバが設定されました。

前提条件

このレッスンを終了するには、最新バージョンの Microsoft Visual Studio が必要です。

このチュートリアルの冒頭に記載されているロールと権限を持っている必要があります。

手順

1. 次のコマンドを使用してサンプルデータベースを起動します。

   ```
   dbsrv17 -xs http(port=8082) "%SQLANYSAMP17%¥demo.db"
   ```

   このコマンドは、HTTP Web サーバがポート 8082 で要求を受信することを指定します。ネットワークで 8082 が許可されない場合は、異なるポート番号を使用します。
2. 次のコマンドを使用して Interactive SQL でデータベースサーバに接続します。

   ```
   dbisql -c "UID=DBA;PWD=sql;SERVER=demo"
   ```
3. 着信要求を受け入れる新しい SOAP サービスを作成します。

   Interactive SQL で次の SQL 文を実行します。

   ```
   CREATE SERVICE "SASoapTest/EmployeeList"
       TYPE 'SOAP'
       DATATYPE ON
       AUTHORIZATION OFF
       SECURE OFF
       USER DBA
       AS SELECT * FROM Employees;
   ```

   この文は、SOAP タイプを出力として生成する SASoapTest/EmployeeList という新しい SOAP Web サービスを作成します。Employees テーブルからすべてのカラムを選択し、結果セットをクライアントに返します。サービス名は、そのサービス名に出現するスラッシュ文字 (/) のため、引用符で囲まれています。別のユーザ ID でログインする場合、そのユーザ ID を反映するように USER DBA を変更する必要があります。

   DATATYPE ON は、明示的なデータ型情報が XML 結果セットの応答と入力パラメータで生成されることを示します。このオプションは、生成される WSDL ドキュメントに影響しません。

   FORMAT 句は指定されていないため、SOAP サービスフォーマットは、次のステップで宣言される関連 DISH サービスフォーマットによって指定されます。
4. SOAP サービスのプロキシとして機能し、WSDL ドキュメントを生成する新しい DISH サービスを作成します。

   Interactive SQL で次の SQL 文を実行します。

   ```
   CREATE SERVICE SASoapTest_DNET
       TYPE 'DISH'
       GROUP SASoapTest
       FORMAT 'DNET'
       AUTHORIZATION OFF
       SECURE OFF
       USER DBA;
   ```

   .NET からアクセスする DISH Web サービスは、FORMAT 'DNET' 句で宣言する必要があります。GROUP 句は、DISH サービスによって処理される必要がある SOAP サービスを識別します。前の手順で作成した EmployeeList サービスは、SASoapTest/EmployeeList として宣言されているため、GROUP SASoapTest の一部になります。別のユーザ ID でログインする場合、そのユーザ ID を反映するように USER DBA を変更する必要があります。
5. Web ブラウザで関連 WSDL ドキュメントにアクセスして、DISH Web サービスが機能していることを確認します。

   Web ブラウザを開き、`http://localhost:8082/demo/SASoapTest_DNET` にアクセスします。

   DISH は、ブラウザのウィンドウに表示される WSDL ドキュメントを自動生成します。

結果

Microsoft Visual C# クライアントアプリケーションの要求を処理できる SOAP/DISH Web サービスが実行されている Web サーバが設定されました。

次のステップ

次のレッスンに進みます。

タスクの概要: チュートリアル:Microsoft Visual C# を使用した SOAP/DISH Web サービスへのアクセス [678 ページ]

次のタスク: レッスン 2:Web サーバと通信するための Microsoft Visual C# アプリケーションの作成 [681 ページ]

関連情報

DISH サービスを作成する方法 [597 ページ]

1.20.4.3.2 レッスン 2:Web サーバと通信するための Microsoft Visual C# アプリケーションの作成

Web サーバと通信するための Microsoft Visual C# アプリケーションを作成します。

前提条件

このチュートリアルのこれまでのレッスンを完了している必要があります。

このチュートリアルの冒頭に一覧されているロールと権限を持っている必要があります。

このレッスンを終了するには、最新バージョンの Microsoft Visual Studio が必要です。

コンテキスト

このレッスンには、localhost への複数の参照が含まれています。Web クライアントを Web サーバと同じコンピュータで実行していない場合は、localhost の代わりに前のレッスンの Web サーバのホスト名または IP アドレスを使用します。

この例では、.NET Framework 2.0 の機能を使用しています。

手順

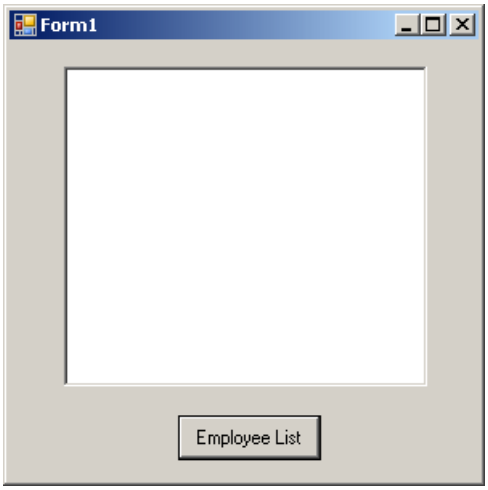
1. Microsoft Visual Studio を起動します。
2. 新しい Microsoft Visual C# *Windows* フォームのアプリケーションプロジェクトを作成します。

   空のフォームが表示されます。
3. オブジェクトに Web 参照を追加します。
   a. プロジェクト ▸ サービス参照の追加 をクリックします。
   b. [サービス参照の追加] ウィンドウで、詳細をクリックします。
   c. [サービス参照設定] ウィンドウで、*Web* 参照の追加をクリックします。
   d. [Web 参照の追加] ウィンドウで、*URL* フィールドに **http://localhost:8082/**demo**/SASoapTest_DNET** と入力します。
   e. 移動 (または緑の矢印) をクリックします。

      Microsoft Visual Studio に、SASoapTest_DNET サービスから使用できる EmployeeList メソッドが表示されます。
   f. 参照の追加をクリックします。

      Microsoft Visual Studio は、localhost をソリューションエクスプローラウィンドウ枠のプロジェクト *Web* リファレンスに追加します。
4. Web クライアントアプリケーションに適したオブジェクトを空のフォームに移植します。

   フォームが次の図のようになるように、ツールボックスウィンドウ枠から ListBox オブジェクトと Button オブジェクトをフォームにドラッグし、テキスト属性を更新します。



5. Web 参照にアクセスするプロシージャを作成し、使用可能なメソッドを使用します。

   [Employee List] ボタンをダブルクリックし、ボタンクリックイベントに次のコードを追加します。

```
int sqlCode;
listBox1.Items.Clear();
localhost.SASoapTest_DNET proxy = new localhost.SASoapTest_DNET();
DataSet results = proxy.EmployeeList(out sqlCode);
DataTableReader dr = results.CreateDataReader();
listBox1.BeginUpdate();
```

```
while (dr.Read())
{
    for (int i = 0; i < dr.FieldCount; i++)
    {
        string dataTypeName = dr.GetDataTypeName(i);
        string dataName = dr.GetName(i);
        System.Type ftype = dr.GetFieldType(i);
        System.TypeCode typeCode = System.Type.GetTypeCode(ftype);
        string columnName = "(" + dataTypeName + ")" + dataName + "=";
        if (dr.IsDBNull(i))
        {
            listBox1.Items.Add(columnName + "(null)");
        }
        else
        {
            switch (typeCode)
            {
                case System.TypeCode.Int32:
                    Int32 intValue = dr.GetInt32(i);
                    listBox1.Items.Add(columnName + intValue);
                    break;
                case System.TypeCode.Decimal:
                    Decimal decValue = dr.GetDecimal(i);
                    listBox1.Items.Add(columnName + decValue.ToString("c"));
                    break;
                case System.TypeCode.String:
                    string stringValue = dr.GetString(i);
                    listBox1.Items.Add(columnName + stringValue);
                    break;
                case System.TypeCode.DateTime:
                    DateTime dateValue = dr.GetDateTime(i);
                    listBox1.Items.Add(columnName + dateValue);
                    break;
                case System.TypeCode.Boolean:
                    Boolean boolValue = dr.GetBoolean(i);
                    listBox1.Items.Add(columnName + boolValue);
                    break;
                default:
                    listBox1.Items.Add(columnName + "(unsupported)");
                    break;
            }
        }
    }
    listBox1.Items.Add("");
}
listBox1.EndUpdate();
dr.Close();
```

6. アプリケーションを実行します。

   デバッグ > デバッグの開始 をクリックします。

7. Web データベースサーバと通信します。

   Employee List をクリックします。

   ListBox オブジェクトに、EmployeeList 結果セットが "(型)名=値" のペアで表示されます。次の出力は、ListBox オブジェクトでのエントリの表示方法を示します。

```
(Int32)EmployeeID=102
(Int32)ManagerID=501
(String)Surname=Whitney
(String)GivenName=Fran
(Int32)DepartmentID=100
(String)Street=9 East Washington Street
(String)City=Cornwall
(String)State=NY
```

```
(String)Country=USA
(String)PostalCode=02192
(String)Phone=6175553985
(String)Status=A
(String)SocialSecurityNumber=017349033
(Decimal)Salary=$45,700.00
(DateTime)StartDate=8/28/1984 12:00:00 AM
(DateTime)TerminationDate=(null)
(DateTime)BirthDate=6/5/1958 12:00:00 AM
(Boolean)BenefitHealthInsurance=True
(Boolean)BenefitLifeInsurance=True
(Boolean)BenefitDayCare=False
(String)Sex=F
```

Salary の値は、クライアントの通貨フォーマットに変換されます。

日付が格納され時刻がない値には、時刻に 00:00:00 つまり午前 0 時が割り当てられます (クライアントのロケール設定によって異なる形式で表示されます)。

NULL が格納された値は、DataTableReader.IsDBNull メソッドを使用してテストされます。

結果

Web サーバからの XML 応答には、フォーマットされた結果セットが含まれます。すべてのカラムデータは、データの文字列表現に変換されます。次の結果セットは、クライアントに送信されるときの結果セットのフォーマット方法を示します。

```
<row>
 <EmployeeID>102</EmployeeID>
 <ManagerID>501</ManagerID>
 <Surname>Whitney</Surname>
 <GivenName>Fran</GivenName>
 <DepartmentID>100</DepartmentID>
 <Street>9 East Washington Street</Street>
 <City>Cornwall</City>
 <State>NY</State>
 <Country>USA</Country>
 <PostalCode>02192</PostalCode>
 <Phone>6175553985</Phone>
 <Status>A</Status>
 <SocialSecurityNumber>017349033</SocialSecurityNumber>
 <Salary>45700.000</Salary>
 <StartDate>1984-08-28</StartDate>
 <BirthDate>1958-06-05</BirthDate>
 <BenefitHealthInsurance>1</BenefitHealthInsurance>
 <BenefitLifeInsurance>1</BenefitLifeInsurance>
 <BenefitDayCare>0</BenefitDayCare>
 <Sex>F</Sex>
 </row>
```

前のレッスンで DATATYPE ON 句が指定されたため、XML 結果セット応答にデータ型情報が生成されます。Web サーバからの応答の一部を次に示します。型情報はデータベースカラムのデータ型に一致します。

```
<xsd:element name='EmployeeID' minOccurs='0' type='xsd:int' />
<xsd:element name='ManagerID' minOccurs='0' type='xsd:int' />
<xsd:element name='Surname' minOccurs='0' type='xsd:string' />
<xsd:element name='GivenName' minOccurs='0' type='xsd:string' />
<xsd:element name='DepartmentID' minOccurs='0' type='xsd:int' />
<xsd:element name='Street' minOccurs='0' type='xsd:string' />
<xsd:element name='City' minOccurs='0' type='xsd:string' />
<xsd:element name='State' minOccurs='0' type='xsd:string' />
```

```
<xsd:element name='Country' minOccurs='0' type='xsd:string' />
<xsd:element name='PostalCode' minOccurs='0' type='xsd:string' />
<xsd:element name='Phone' minOccurs='0' type='xsd:string' />
<xsd:element name='Status' minOccurs='0' type='xsd:string' />
<xsd:element name='SocialSecurityNumber' minOccurs='0' type='xsd:string' />
<xsd:element name='Salary' minOccurs='0' type='xsd:decimal' />
<xsd:element name='StartDate' minOccurs='0' type='xsd:date' />
<xsd:element name='TerminationDate' minOccurs='0' type='xsd:date' />
<xsd:element name='BirthDate' minOccurs='0' type='xsd:date' />
<xsd:element name='BenefitHealthInsurance' minOccurs='0' type='xsd:boolean' />
<xsd:element name='BenefitLifeInsurance' minOccurs='0' type='xsd:boolean' />
<xsd:element name='BenefitDayCare' minOccurs='0' type='xsd:boolean' />
<xsd:element name='Sex' minOccurs='0' type='xsd:string' />
```

日付だけが格納されたカラムは、XML 応答で `yyyy-mm-dd` のようにフォーマットされます。

時刻だけが格納されたカラムは、XML 応答で `hh:mm:ss.nnn-HH:MM` または `hh:mm:ss.nnn+HH:MM` のようにフォーマットされます。ゾーンオフセット (-HH:MM または +HH:MM) は文字列のサフィックスとして付きます。

日付と時刻の両方が格納されたカラムは、XML 応答で `yyyy-mm-ddThh:mm:ss.nnn-HH:MM` または `yyyy-mm-ddThh:mm:ss.nnn+HH:MM` のようにフォーマットされます。日付と時刻は、文字 T で区切られます。ゾーンオフセット (-HH:MM または +HH:MM) は文字列のサフィックスとして付きます。

日付、時刻、タイムゾーンオフセットが格納されたカラムは、XML 応答で `yyyy-mm-ddThh:mm:ss.nnn-HH:MM` または `yyyy-mm-ddThh:mm:ss.nnn+HH:MM` のようにフォーマットされます。日付と時刻は、文字 T で区切られます。ゾーンオフセット (-HH:MM または +HH:MM) は文字列のサフィックスとして付きます。

太平洋タイムゾーンで実行されているサーバのいくつかの例について、以下に説明します。

```
<ADate>2013-01-27</ADate>
<ATime>12:34:56.000-08:00</ATime>
<ADateTime>2013-01-27T12:34:56.000-08:00</ADateTime>
<ADateTimeWithZone>2013-01-27T12:34:56.000+06:00</ADateTimeWithZone>
```

これらの日付と時刻が .NET アプリケーションで表示される形式は、クライアントのタイムゾーンとロケール設定によって異なります。次は、米国ロケールの東部タイムゾーンにあるクライアントの例です (今日の日付が 2013 年 1 月 28 日と仮定します)。

```
(DateTime)ADate=1/27/2013 12:00:00 AM
(DateTime)ATime=1/28/2013 3:34:56 PM
(DateTime)ADateTime=1/27/2013 3:34:56 PM
(DateTime)ADateTimeWithZone=1/27/2013 1:34:56 AM
```

TypeCode 列挙には別の日付と時刻はなく、TypeCode.DateTime のみがあります。このため、すべての結果に日付と時刻の両方が含まれます。

これらの日付と時刻の XML 応答でのデータタイプ情報については、以下に説明します。

```
<xsd:element name='ADate' minOccurs='0' type='xsd:date' />
<xsd:element name='ATime' minOccurs='0' type='xsd:time' />
<xsd:element name='ADateTime' minOccurs='0' type='xsd:dateTime' />
<xsd:element name='ADateTimeWithZone' minOccurs='0' type='xsd:dateTime' />
```

タスクの概要: チュートリアル:Microsoft Visual C# を使用した SOAP/DISH Web サービスへのアクセス [678 ページ]

前のタスク: レッスン 1:SOAP 要求を受信し SOAP 応答を送信する Web サーバの設定 [679 ページ]

1.20.4.4 チュートリアル: JAX-WS を使用した SOAP/DISH Web サービスへのアクセス

このチュートリアルでは、Web サーバとして動作するデータベースサーバ上の SOAP/DISH サービスにアクセスする XML Web サービス (JAX-WS) クライアントアプリケーション用の Java API の作成方法について説明します。

前提条件

次の項目が必要です。

- SOAP の知識
- Java および JAX-WS の知識
- ソフトウェアの Web サービス機能の基本的な知識

次のソフトウェアが必要です。

- JDK 1.7.0 以降のバージョン
- JAX-WS 2.2.7 以降のバージョン

このチュートリアルのレッスンを実行するには、次の権限が必要です。

- CREATE ANY OBJECT
- MANAGE ANY WEB SERVICE

コンテキスト

このチュートリアルでは、次の作業の方法について説明します。

- 新しい Web サービスデータベースを作成し、起動します。データベースサーバは Web サーバとして動作します。
- SOAP Web サービスを作成します。
- SOAP 要求に含まれている情報を返すプロシージャを設定します。
- WSDL ドキュメントを提供し、プロキシとして機能する DISH Web サービスを作成します。
- クライアントコンピュータで JAX-WS を使用し、Web サーバから WSDL ドキュメントを処理します。
- WSDL ドキュメントの情報を使用して SOAP サービスから情報を取得する Java クライアントアプリケーションを作成します。

1. レッスン 1: SOAP 要求を受信し SOAP 応答を送信する Web サーバの設定 [687 ページ]
   JAX-WS クライアントアプリケーションの要求を処理する SOAP および DISH Web サービスが実行されている Web サーバが設定されました。
2. レッスン 2: Web サーバと通信するための Java アプリケーションの作成 [690 ページ]
   DISH サービスで生成された WSDL ドキュメントを処理し、WSDL ドキュメントで定義されたスキーマに基づいてテーブルデータにアクセスする Java アプリケーションを作成します。

関連情報

HTTP Web サーバとしてのデータベースサーバ [587 ページ]

1.20.4.4.1 レッスン 1: SOAP 要求を受信し SOAP 応答を送信する Web サーバの設定

JAX-WS クライアントアプリケーションの要求を処理する SOAP および DISH Web サービスが実行されている Web サーバが設定されました。

前提条件

このチュートリアルの冒頭に一覧されているロールと権限を持っている必要があります。

コンテキスト

このレッスンでは、このチュートリアルの後で使用する Web サーバと単純な Web サービスを設定します。プロキシソフトウェアを使用して XML メッセージトラフィックを確認します。プロキシは、クライアントアプリケーションと Web サーバの間に挿入されます。

手順

1. 次のコマンドを使用してサンプルデータベースを起動します。

   ```
   dbsrv17 -xs http(port=8082) "%SQLANYSAMP17%¥demo.db"
   ```

   このコマンドは、HTTP Web サーバがポート 8082 で要求を受信することを指定します。ネットワークで 8082 が許可されない場合は、異なるポート番号を使用します。
2. 次のコマンドを使用して Interactive SQL でデータベースサーバに接続します。

   ```
   dbisql -c "UID=DBA;PWD=sql;SERVER=demo"
   ```
3. Employees テーブルカラムをリストするストアドプロシージャを作成します。

   Interactive SQL で次の SQL 文を実行します。

   ```
   CREATE OR REPLACE PROCEDURE ListEmployees()
   RESULT (
       EmployeeID            INTEGER,
       Surname               CHAR(20),
       GivenName             CHAR(20),
   ```

```
    StartDate               DATE,
    TerminationDate         DATE )
BEGIN
    SELECT EmployeeID, Surname, GivenName, StartDate, TerminationDate
    FROM Employees;
END;
```

これらの文は、結果セット出力の構造を定義する ListEmployees という新しいストアドプロシージャを作成し、Employees テーブルから特定のカラムを選択します。

4. 着信要求を受け入れる新しい SOAP サービスを作成します。

   Interactive SQL で次の SQL 文を実行します。

```
CREATE SERVICE "WS/EmployeeList"
    TYPE 'SOAP'
    FORMAT 'CONCRETE' EXPLICIT ON
    DATATYPE ON
    AUTHORIZATION OFF
    SECURE OFF
    USER DBA
    AS CALL ListEmployees();
```

   この文は、SOAP タイプを出力として生成する WS/EmployeeList という新しい SOAP Web サービスを作成します。Web クライアントがサービスに要求を送信すると、ListEmployees プロシージャが呼び出されます。サービス名は、そのサービス名に出現するスラッシュ文字 (/) のため、引用符で囲まれています。

   JAX-WS からアクセスする SOAP Web サービスは、FORMAT 'CONCRETE' 句で宣言する必要があります。EXPLICIT ON 句は、ListEmployees プロシージャの結果セットに基づいて、対応する DISH サービスで明示的なデータセットオブジェクトを記述する XML スキーマを生成することを示します。EXPLICIT 句の影響を受けるのは、生成される WSDL ドキュメントのみです。

   DATATYPE ON は、明示的なデータ型情報が XML 結果セットの応答と入力パラメータで生成されることを示します。このオプションは、生成される WSDL ドキュメントに影響しません。

   別のユーザ ID でログインする場合、そのユーザ ID を反映するように USER DBA を変更する必要があります。

5. SOAP サービスのプロキシとして機能し、WSDL ドキュメントを生成する新しい DISH サービスを作成します。

   Interactive SQL で次の SQL 文を実行します。

```
CREATE SERVICE WSDish
    TYPE 'DISH'
    FORMAT 'CONCRETE'
    GROUP WS
    AUTHORIZATION OFF
    SECURE OFF
    USER DBA;
```

   JAX-WS からアクセスする DISH Web サービスは、FORMAT 'CONCRETE' 句で宣言する必要があります。GROUP 句は、DISH サービスによって処理される必要がある SOAP サービスを識別します。前の手順で作成した EmployeeList サービスは、WS/EmployeeList として宣言されているため、GROUP WS の一部になります。別のユーザ ID でログインする場合、そのユーザ ID を反映するように USER DBA を変更する必要があります。

6. Web ブラウザで関連 WSDL ドキュメントにアクセスして、DISH Web サービスが機能していることを確認します。

   Web ブラウザを開き、`http://localhost:8082/demo/WSDish` に移動します。

DISH サービスは、ブラウザのウィンドウに表示される WSDL ドキュメントを自動生成します。EmployeeListDataset オブジェクトを確認します。次のような出力になります。

```
<s:complexType name="EmployeeListDataset">
<s:sequence>
<s:element name="rowset">
  <s:complexType>
  <s:sequence>
  <s:element name="row" minOccurs="0" maxOccurs="unbounded">
    <s:complexType>
    <s:sequence>
    <s:element minOccurs="0" maxOccurs="1" name="EmployeeID" nillable="true"
type="s:int" />
    <s:element minOccurs="0" maxOccurs="1" name="Surname" nillable="true"
type="s:string" />
    <s:element minOccurs="0" maxOccurs="1" name="GivenName" nillable="true"
type="s:string" />
    <s:element minOccurs="0" maxOccurs="1" name="StartDate" nillable="true"
type="s:date" />
    <s:element minOccurs="0" maxOccurs="1" name="TerminationDate"
nillable="true" type="s:date" />
    </s:sequence>
    </s:complexType>
  </s:element>
  </s:sequence>
  </s:complexType>
</s:element>
</s:sequence>
</s:complexType>
```

EmployeeListDataset は、EmployeeList SOAP サービスの FORMAT 'CONCRETE' 句と EXPLICIT ON 句で生成された明示的なオブジェクトです。この後のレッスンで、wsimport アプリケーションはこの情報を使用して、このサービス用の SOAP 1.1 クライアントインタフェースを生成します。

結果

JAX-WS クライアントアプリケーションの要求を処理できる SOAP/DISH Web サービスが実行されている Web サーバが設定されました。

次のステップ

次のレッスンに進みます。

タスクの概要: チュートリアル: JAX-WS を使用した SOAP/DISH Web サービスへのアクセス [686 ページ]

次のタスク: レッスン 2: Web サーバと通信するための Java アプリケーションの作成 [690 ページ]

関連情報

DISH サービスを作成する方法 [597 ページ]

1.20.4.4.2 レッスン 2: Web サーバと通信するための Java アプリケーションの作成

DISH サービスで生成された WSDL ドキュメントを処理し、WSDL ドキュメントで定義されたスキーマに基づいてテーブルデータにアクセスする Java アプリケーションを作成します。

前提条件

このチュートリアルのこれまでのレッスンを完了している必要があります。

前のレッスンで起動した Web サーバを実行している必要があります。

このチュートリアルの冒頭に一覧されているロールと権限を持っている必要があります。

コンテキスト

現時点では、このチュートリアルで使用する JAX-WS のバージョンは 2.2.9 で、JDK 1.8 に含まれています。次のステップはそのバージョンによって決まります。JDK に JAX-WS が存在するかどうかを確認するには、JDK `bin` ディレクトリで wsimport アプリケーションをチェックしてください。存在しない場合、http://jax-ws.java.net/ にアクセスし、最新バージョンの JAX-WS をダウンロードしてインストールします。

このレッスンには、localhost への複数の参照が含まれています。Web クライアントを Web サーバと同じコンピュータで実行していない場合は、localhost の代わりにレッスン 1 の Web サーバのホスト名または IP アドレスを使用します。

手順

1. コマンドプロンプトで、Java コードと生成ファイル用に新しい作業ディレクトリを作成します。この新しいディレクトリに移動します。
2. 次のコマンドを使用して、DISH Web サービスを呼び出し WSDL ドキュメントをインポートするインタフェースを生成します。

```
wsimport -keep -p employeelist "http://localhost:8082/demo/WSDish"
```

このコマンドが解析エラーで失敗した場合、*localhost* を 127.0.0.1 または現在使用している IP アドレスやホスト名に置き換えます。次に例を示します。

```
wsimport -keep -p employeelist "http://127.0.0.1:8082/demo/WSDish"
```

wsimport アプリケーションは、特定の URL から WSDL ドキュメントを取得します。WSDL ドキュメント用のインタフェースを作成するために .java ファイルを生成して、.class ファイルにコンパイルします。

keep オプションは、クラスファイルの生成後に Java ソースファイルを削除しないことを示します。生成された Java ソースコードを使用すると、生成されたクラスファイルについて理解できます。

p オプションは、Java ソースおよびクラスファイルを employeelist パッケージに生成する必要があることを示します。

wsimport アプリケーションは、現在の作業ディレクトリに employeelist という新しいサブフォルダを作成します。次に、そのディレクトリの内容を示します。

- EmployeeList.class
- EmployeeList.java
- EmployeeListDataset$Rowset$Row.class
- EmployeeListDataset$Rowset.class
- EmployeeListDataset.class
- EmployeeListDataset.java
- EmployeeListResponse.class
- EmployeeListResponse.java
- FaultMessage.class
- FaultMessage.java
- ObjectFactory.class
- ObjectFactory.java
- package-info.class
- package-info.java
- WSDish.class
- WSDish.java
- WSDishSoapPort.class
- WSDishSoapPort.java

3. 生成されたソースコードに定義されるデータセットオブジェクトに基づいて、データベースサーバからテーブルデータにアクセスする Java アプリケーションを作成します。

   次に、これを実行するサンプル Java アプリケーションを示します。ソースコードを、現在の作業ディレクトリに SoapDemo.java として保存します。現在の作業ディレクトリには、employeelist サブフォルダが含まれている必要があります。

```
// SoapDemo.java illustrates a web service client that
// calls the WSDish service and prints out the data.
import java.util.*;
import javax.xml.ws.*;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import javax.xml.datatype.*;
import employeelist.*;
public class SoapDemo
{
  public static void main( String[] args )
  {
```

```
    try {
      WSDish service = new WSDish();
      Holder<EmployeeListDataset> response =
          new Holder<EmployeeListDataset>();
      Holder<Integer> sqlcode = new Holder<Integer>();

      WSDishSoapPort port = service.getWSDishSoap();
      // This is the SOAP service call to EmployeeList
      port.employeeList( response, sqlcode );
      EmployeeListDataset result = response.value;
      EmployeeListDataset.Rowset rowset = result.getRowset();
      List<EmployeeListDataset.Rowset.Row> rows = rowset.getRow();
      String fieldType;
      String fieldName;
      String fieldValue;
      Integer fieldInt;
      XMLGregorianCalendar fieldDate;

      for ( int i = 0; i < rows.size(); i++ ) {
        EmployeeListDataset.Rowset.Row row = rows.get( i );
        fieldType = row.getEmployeeID().getDeclaredType().getSimpleName();
        fieldName = row.getEmployeeID().getName().getLocalPart();
        fieldInt = row.getEmployeeID().getValue();
        System.out.println( "(" + fieldType + ")" + fieldName +
                            "=" + fieldInt );

        fieldType = row.getSurname().getDeclaredType().getSimpleName();
        fieldName = row.getSurname().getName().getLocalPart();
        fieldValue = row.getSurname().getValue();
        System.out.println( "(" + fieldType + ")" + fieldName +
                            "=" + fieldValue );

        fieldType = row.getGivenName().getDeclaredType().getSimpleName();
        fieldName = row.getGivenName().getName().getLocalPart();
        fieldValue = row.getGivenName().getValue();
        System.out.println( "(" + fieldType + ")" + fieldName +
                             "=" + fieldValue );

        fieldType = row.getStartDate().getDeclaredType().getSimpleName();
        fieldName = row.getStartDate().getName().getLocalPart();
        fieldDate = row.getStartDate().getValue();
        System.out.println( "(" + fieldType + ")" + fieldName +
                            "=" + fieldDate );

        if ( row.getTerminationDate() == null ) {
          fieldType = "unknown";
          fieldName = "TerminationDate";
          fieldDate = null;
        } else {
          fieldType =
            row.getTerminationDate().getDeclaredType().getSimpleName();
          fieldName = row.getTerminationDate().getName().getLocalPart();
          fieldDate = row.getTerminationDate().getValue();
        }
        System.out.println( "(" + fieldType + ")" + fieldName +
                            "=" + fieldDate );
        System.out.println();
      }
    }
    catch (Exception x) {
      x.printStackTrace();
    }
  }
}
```

4. 次のコマンドを使用して、Java アプリケーションをコンパイルします。

```
set CLASSPATH=.;%CLASSPATH%
```

```
javac SoapDemo.java
```

5. 次のコマンドを使用して、アプリケーションを実行します。

```
java SoapDemo
```

6. アプリケーションは、Web サーバに要求を送信し、いくつかのローエントリが含まれたローセットを持つ EmployeeListResult で構成される XML 結果セット応答を受信します。

   このアプリケーションは、サーバが提供するすべてのカラムデータを標準のシステム出力に表示します。

   このアプリケーションで使用される Java メソッドの詳細については、javax.xml.bind.JAXBElement クラス API のマニュアルを参照してください。

結果

次に、実行中の SoapDemo からの出力例を示します。

```
(Integer)EmployeeID=102
(String)Surname=Whitney
(String)GivenName=Fran
(XMLGregorianCalendar)StartDate=1984-08-28
(unknown)TerminationDate=null
(Integer)EmployeeID=105
(String)Surname=Cobb
(String)GivenName=Matthew
(XMLGregorianCalendar)StartDate=1985-01-01
(unknown)TerminationDate=null
.
.
.
(Integer)EmployeeID=1740
(String)Surname=Nielsen
(String)GivenName=Robert
(XMLGregorianCalendar)StartDate=1994-06-24
(unknown)TerminationDate=null
(Integer)EmployeeID=1751
(String)Surname=Ahmed
(String)GivenName=Alex
(XMLGregorianCalendar)StartDate=1994-07-12
(XMLGregorianCalendar)TerminationDate=2008-04-18
```

TerminationDate カラムは、その値が NULL でない場合にのみ送信されます。この Java アプリケーションは、TerminationDate カラムが存在しない場合、それを検出するように設計されています。この例では、終了日に NULL 以外の値が設定され、Employees テーブルの最後のローが変更されました。

次の例は、Web サーバからの SOAP 応答を示します。応答には、クエリの実行結果の SQLCODE が含まれます。

```
<tns:EmployeeListResponse>
 <tns:EmployeeListResult xsi:type='tns:EmployeeListDataset'>
  <tns:rowset>
    <tns:row> ... </tns:row>
    .
    .
    .
    <tns:row>
      <tns:EmployeeID xsi:type="xsd:int">1751</tns:EmployeeID>
      <tns:Surname xsi:type="xsd:string">Ahmed</tns:Surname>
      <tns:GivenName xsi:type="xsd:string">Alex</tns:GivenName>
```

```
      <tns:StartDate xsi:type="xsd:dateTime">1994-07-12</tns:StartDate>
      <tns:TerminationDate xsi:type="xsd:dateTime">2010-03-22</tns:TerminationDate>
    </tns:row>
  </tns:rowset>
 </tns:EmployeeListResult>
 <tns:sqlcode>0</tns:sqlcode>
</tns:EmployeeListResponse>
```

各ローセットには、カラム名とデータ型が含まれます。

タスクの概要: チュートリアル: JAX-WS を使用した SOAP/DISH Web サービスへのアクセス [686 ページ]

前のタスク: レッスン 1: SOAP 要求を受信し SOAP 応答を送信する Web サーバの設定 [687 ページ]

1.21 3 層コンピューティングと分散トランザクション

データベースサーバは、トランザクションサーバによって調整された分散トランザクションに関わるリソースマネージャとして使用できます。

3 層環境では、クライアントアプリケーションと一連のリソースマネージャの間にアプリケーションサーバを置きますが、これが一般的な分散トランザクション環境です。アプリケーションサーバは、クライアントアプリケーションにトランザクション論理を提供し、一連の操作がアトミックに実行されることを保証します。SAP Enterprise Application Server (SAP EAServer) およびその他の一部のアプリケーションサーバはトランザクションサーバです。

SAP EAServer と Microsoft Transaction Server はともに、Microsoft DTC (分散トランザクションコーディネーター) を使用してトランザクションを調整します。データベースサーバは、DTC サービスによって制御された分散トランザクションをサポートします。そのため、前述したアプリケーションサーバのいずれかとともに、または DTC モデルに基づくその他のどのような製品とでも、データベースサーバを使用できます。DTC は **OLE** トランザクションを使用します。OLE トランザクションは **2** フェーズコミットのプロトコルを使用して、複数のリソースマネージャに関わるトランザクションを調整します。分散トランザクションを使用するには、DTC をインストールしておく必要があります。

データベースサーバを 3 層環境に統合する場合、作業のほとんどをアプリケーションサーバから行う必要があります。3 層コンピューティングの概念とアーキテクチャについて紹介し、データベースサーバの関連機能の概要について説明します。ここでは、アプリケーションサーバを設定してデータベースサーバとともに動作させる方法については説明しません。詳細については、使用しているアプリケーションサーバのマニュアルを参照してください。

また、アプリケーションの中で直接 DTC を使用して、複数のリソースマネージャにわたるトランザクションを調整することもできます。

このセクションの内容:

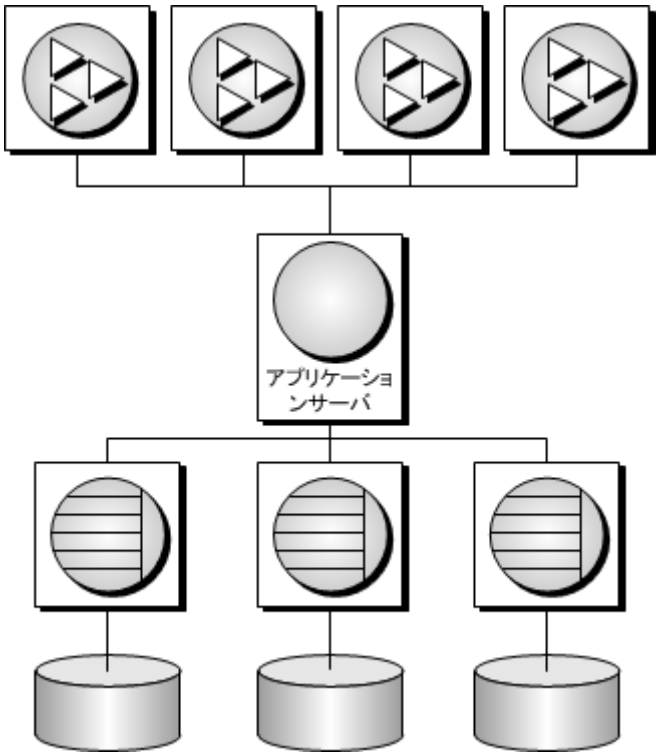
3 層コンピューティングのアーキテクチャ [695 ページ]
3 層コンピューティングの場合、アプリケーション論理は SAP EAServer (SAP Enterprise Application Server) などのアプリケーションサーバに格納されます。アプリケーションサーバは、リソースマネージャとクライアントアプリケーションの間に置かれます。

分散トランザクション [697 ページ]
データベースサーバは、分散トランザクションに関与している間、トランザクション制御をトランザクションサーバに渡します。また、データベースサーバは、トランザクション管理を暗黙的に実行しないようにします。データベースサーバが分散トランザクションを処理する場合、自動的に次の条件が適用されます。

1.21.1 3 層コンピューティングのアーキテクチャ

3 層コンピューティングの場合、アプリケーション論理は SAP EAServer (SAP Enterprise Application Server) などのアプリケーションサーバに格納されます。アプリケーションサーバは、リソースマネージャとクライアントアプリケーションの間に置かれます。

多くの場合、1 つのアプリケーションサーバから複数のリソースマネージャにアクセスできます。インターネットの場合、クライアントアプリケーションはブラウザベースであり、アプリケーションサーバは、通常、Web サーバの拡張機能です。



SAP EAServer は、アプリケーション論理をコンポーネントとして格納し、そのコンポーネントをクライアントアプリケーションから利用できるようにします。利用できるコンポーネントは、SAP PowerBuilder コンポーネント、JavaBeans、または COM コンポーネントです。

詳細については、SAP EAServer のマニュアルを参照してください。

このセクションの内容:

分散トランザクションに関する用語 [696 ページ]
分散トランザクションについて説明するときには、いくつかの共通の用語が使用されます。

アプリケーションサーバが DTC を使用する方法 [696 ページ]
SAP Enterprise Application Server (SAP EAServer) と Microsoft Transaction Server は、どちらもコンポーネントサーバです。アプリケーションロジックはコンポーネントとして格納され、クライアントアプリケーションから利用できます。

分散トランザクションのアーキテクチャ [697 ページ]

次の図は、分散トランザクションのアーキテクチャを示しています。この場合、リソースマネージャプロキシは ADO.NET、OLE DB、または ODBC です。

1.21.1.1 分散トランザクションに関する用語

分散トランザクションについて説明するときには、いくつかの共通の用語が使用されます。

- リソースマネージャは、トランザクションに関連するデータを管理するサービスです。ADO.NET、OLE DB、または ODBC を通してアクセスする場合、データベースサーバは分散トランザクション内でリソースマネージャとして動作できます。.NET データプロバイダ、OLE DB プロバイダ、ODBC ドライバは、クライアントコンピュータ上のリソースマネージャプロキシとして動作します。.NET データプロバイダは、DbProviderFactory と TransactionScope を使用して分散トランザクションをサポートします。
- アプリケーションコンポーネントは、リソースマネージャと直接通信しないでリソースディスペンサーと通信できます。リソースディスペンサーは、リソースマネージャへの接続または接続プールを管理します。データベースサーバがサポートする 2 つのリソースディスペンサは、ODBC ドライバマネージャと OLE DB です。
- トランザクションコンポーネントが (リソースマネージャを使用して) データベースとの接続を要求すると、アプリケーションサーバはトランザクションに関わるデータベース接続をエンリストします。DTC とリソースディスペンサーがエンリスト処理を実行します。

詳細については、使用しているトランザクションサーバのマニュアルを参照してください。

2 フェーズコミット

2 フェーズコミットを使用して、分散トランザクションを管理します。トランザクション処理が完了すると、トランザクションマネージャ (DTC) は、トランザクションにエンリストされたすべてのリソースマネージャにトランザクションをコミットする準備ができているかどうかを問い合わせます。このフェーズは、コミットの準備と呼ばれます。

すべてのリソースマネージャからコミット準備完了の応答があると、DTC は各リソースマネージャにコミット要求を送信し、トランザクションの完了をクライアントに通知します。1 つ以上のリソースマネージャが応答しない場合、またはトランザクションをコミットできないと応答した場合、トランザクションのすべての処理は、すべてのリソースマネージャにわたってロールバックされます。

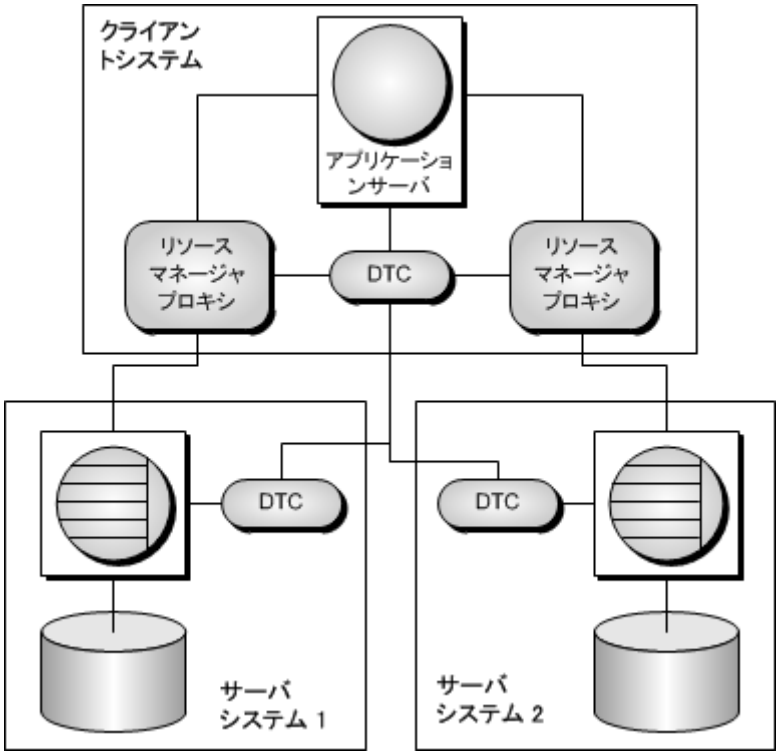
1.21.1.2 アプリケーションサーバが DTC を使用する方法

SAP Enterprise Application Server (SAP EAServer) と Microsoft Transaction Server は、どちらもコンポーネントサーバです。アプリケーションロジックはコンポーネントとして格納され、クライアントアプリケーションから利用できます。

各コンポーネントのトランザクション属性は、コンポーネントがどのようにトランザクションに関わるかを示します。コンポーネントを構築する場合、トランザクションの作業 (リソースマネージャとの接続、各リソースマネージャが管理するデータに対する操作など) をコンポーネントの中にプログラムする必要があります。ただし、トランザクション管理のロジックをコンポーネントに追加する必要はありません。トランザクション属性が設定され、コンポーネントにトランザクション管理が必要な場合、EAServer は、DTC を使用してトランザクションをエンリストし、2 フェーズコミット処理を管理します。

1.21.1.3 分散トランザクションのアーキテクチャ

次の図は、分散トランザクションのアーキテクチャを示しています。この場合、リソースマネージャプロキシは ADO.NET、OLE DB、または ODBC です。



この場合、単一のリソースディスペンサーが使用されています。アプリケーションサーバは、DTC にトランザクションの準備を要求します。DTC とリソースディスペンサーは、トランザクション内の各接続をエンリストします。作業を実行し、必要に応じてトランザクションステータスを DTC に通知するためには、各リソースマネージャが DTC とデータベースの両方にアクセスする必要があります。

分散トランザクションを操作するには、各コンピュータ上で分散トランザクションコーディネーター (DTC) サービスを実行する必要があります。Microsoft Windows の *[サービス]* ウィンドウから DTC を開始または停止できます。DTC サービスは、MSDTC という名前で表示されます。

詳細については、DTC または EAServer のマニュアルを参照してください。

1.21.2 分散トランザクション

データベースサーバは、分散トランザクションに関与している間、トランザクション制御をトランザクションサーバに渡します。また、データベースサーバは、トランザクション管理を暗黙的に実行しないようにします。データベースサーバが分散トランザクションを処理する場合、自動的に次の条件が適用されます。

- オートコミットが使用されている場合は、自動的にオートコミットがオフになります。
- 分散トランザクション中は、データ定義文 (副次的な効果としてコミットされます) を使用できません。

- アプリケーションが明示的な COMMIT または ROLLBACK を発行する場合に、トランザクションコーディネーターを介さないでデータベースサーバに対して直接発行すると、エラーが発生します。ただし、トランザクションは中止されません。
- 1 つの接続で処理できるのは、1 回に 1 つの分散トランザクションに限られます。
- 接続が分散トランザクションにエンリストされるときに、すべてのコミット操作が完了している必要があります。

このセクションの内容:

DTC の独立性レベル [698 ページ]
DTC には、独立性レベルのセットが定義されています。アプリケーションサーバは、この中からレベルを指定します。DTC の独立性レベルは、次のようにデータベースサーバの独立性レベルにマッピングされます。

分散トランザクションからのリカバリ [698 ページ]
コミットされていない操作の保留中にデータベースサーバにフォールトが発生した場合、トランザクションのアトミックな状態を保つために、起動時にこれらの操作をロールバックまたはコミットする必要があります。

1.21.2.1 DTC の独立性レベル

DTC には、独立性レベルのセットが定義されています。アプリケーションサーバは、この中からレベルを指定します。DTC の独立性レベルは、次のようにデータベースサーバの独立性レベルにマッピングされます。

DTC の独立性レベル	データベースサーバの独立性レベル
ISOLATIONLEVEL_UNSPECIFIED	0
ISOLATIONLEVEL_CHAOS	0
ISOLATIONLEVEL_READUNCOMMITTED	0
ISOLATIONLEVEL_BROWSE	0
ISOLATIONLEVEL_CURSORSTABILITY	1
ISOLATIONLEVEL_READCOMMITTED	1
ISOLATIONLEVEL_REPEATABLEREAD	2
ISOLATIONLEVEL_SERIALIZABLE	3
ISOLATIONLEVEL_ISOLATED	3

1.21.2.2 分散トランザクションからのリカバリ

コミットされていない操作の保留中にデータベースサーバにフォールトが発生した場合、トランザクションのアトミックな状態を保つために、起動時にこれらの操作をロールバックまたはコミットする必要があります。

分散トランザクションからコミットされていない操作がリカバリ中に検出されると、データベースサーバは DTC に接続を試み、検出された操作を保留または不明のトランザクションに再エンリストするように要求します。再エンリストが完了すると、DTC は未処理操作のロールバックまたはコミットをデータベースサーバに指示します。

再エンリスト処理が失敗すると、データベースサーバは不明の操作をコミットするかロールバックするかを判断できなくて、リカバリは失敗します。データの状態が保証されないことを前提にして、リカバリに失敗したデータベースをリカバリする場合は、次のデータベースサーバオプションを使って強制リカバリします。

-tmf

DTC が特定できないときは、未処理の操作をロールバックしてリカバリを続行します。

-tmt

-tmt オプションが指定されている場合に、指定した時間までに再エンリストが行われなかった場合、データベースリカバリは失敗します。-tmt と -tmf の両方が指定されている場合に、指定した時間までに再エンリストが行われなかった場合、操作をロールバックしてリカバリを続行します。

1.22 データベースツールのプログラミング

バックアップやアンロードなどの一般的なデータベースタスクを実装するアプリケーションを C または C++ で記述することができます。

このセクションの内容:

データベースツールインタフェース (DBTools) [699 ページ]
データベースサーバソフトウェアは、管理ツールおよび一連のデータベース管理ユーティリティを含んでいます。これらのデータベース管理ユーティリティを使用すると、データベースのバックアップ、データベースの作成、トランザクションログの SQL への変換などの作業を実行できます。

SQL Anywhere データベースツール C API リファレンス [706 ページ]
アプリケーションプログラムは、DBTools API ライブラリと情報を交換するために構造体を使用します。`dbrmt.h` に定義されている a_remote_sql 構造体を除くすべての構造体は、`dbtools.h` に定義されています。

ソフトウェアコンポーネントの終了コード [786 ページ]
すべてのデータベースツールライブラリのエントリポイントが終了コードを返します。データベースサーバおよびユーティリティ (dbsrv17、dbbackup、dbspawn など) でもこれらの終了コードが使用されます。

1.22.1 データベースツールインタフェース (DBTools)

データベースサーバソフトウェアは、管理ツールおよび一連のデータベース管理ユーティリティを含んでいます。これらのデータベース管理ユーティリティを使用すると、データベースのバックアップ、データベースの作成、トランザクションログの SQL への変換などの作業を実行できます。

サポートするプラットフォーム

すべてのデータベース管理ユーティリティはデータベースツールライブラリと呼ばれる共有ライブラリを使用します。Windows 向けのライブラリ名は `dbtool`*17*`.dll` です。Linux および UNIX 向けのライブラリ名は `libdbtool`*17*`_r.so` です。Mac OS X 向けのライブラリ名は libdbtool*17*_r.dylib です。

データベースツールライブラリを呼び出すことによって、独自のデータベース管理ユーティリティを開発したり、データベース管理機能をアプリケーションに組み込んだりできます。

データベースツールライブラリは、各データベース管理ユーティリティに対してそれぞれ関数、またはエントリポイントを持ちます。また、他のデータベースツール関数の使用前と使用後に、関数を呼び出す必要があります。

dbtools.h ヘッダファイル

dbtools ヘッダファイルは、DBTools ライブラリへのエントリポイントと、DBTools ライブラリとの間で情報をやりとりするために使用する構造体をリストします。dbtools.h ファイルはすべて、ソフトウェアインストールディレクトリの SDK¥Include サブディレクトリにインストールされています。エントリポイントと構造体メンバーの最新情報については、dbtools.h ファイルを参照してください。

dbtools.h ヘッダファイルには、他に次のようなファイルが含まれています。

sqlca.h

SQLCA 自身ではなく、さまざまなマクロの解析のために使用するものです。

dllapi.h

オペレーティングシステムと言語に依存するマクロのためのプリプロセッサマクロを定義します。

dbtlvers.h

DB_TOOLS_VERSION_NUMBER プリプロセッサマクロとその他のバージョン固有のマクロを定義します。

sqldef.h ヘッダファイル

sqldef.h ヘッダファイルはエラー戻り値も含みます。

dbrmt.h ヘッダファイル

dbrmt.h ヘッダファイルは、DBTools ライブラリへの DBRemoteSQL エントリポイントと、DBRemoteSQL エントリポイントとの間で情報をやりとりするために使用する構造体をリストします。dbrmt.h ファイルはすべて、ソフトウェアインストールディレクトリの SDK¥Include サブディレクトリにインストールされています。DBRemoteSQL エントリポイントと構造体メンバーの最新情報については、dbrmt.h ファイルを参照してください。

このセクションの内容:

DBTools インポートライブラリ [701 ページ]
DBTools 関数を使用するには、必要な関数定義を含む DBTools インポートライブラリにアプリケーションをリンクする必要があります。

DBTools ライブラリの初期化とファイナライズ [701 ページ]
他の DBTools 関数を使用する前に、DBToolsInit を呼び出す必要があります。DBTools ライブラリを使い終わったときは、DBToolsFini を呼び出してください。

DBTools 関数呼び出し [702 ページ]

すべてのツールは、まず構造体に値を設定し、次に DBTools ライブラリの関数 (またはエントリポイント) を呼び出すことによって実行します。各エントリポイントには、引数として単一構造体へのポインタを渡します。

コールバック関数 [703 ページ]
DBTools 構造体には MSG_CALLBACK 型または SHOULD_STOP_CALLBACK 型の要素がいくつかあります。それらはコールバック関数へのポインタです。

バージョン番号と互換性 [705 ページ]
各構造体にはバージョン番号を示すメンバーがあります。DBTools の関数を呼び出す前に、このバージョンフィールドに、アプリケーション開発に使用した DBTools ライブラリのバージョン番号を設定しておきます。

ビットフィールド [705 ページ]
DBTools 構造体の多くは、ビットフィールドを使用してブール情報を効率よく格納しています。

DBTools の例 [706 ページ]
このサンプルとコンパイル手順は、*%SQLANYSAMP17%*¥SQLAnywhere¥DBTools ディレクトリにあります。

1.22.1.1 DBTools インポートライブラリ

DBTools 関数を使用するには、必要な関数定義を含む DBTools インポートライブラリにアプリケーションをリンクする必要があります。

UNIX システムでは、インポートライブラリは不要です。libdbtool*17*.so (非スレッド) または libdbtool*17*_r.so (スレッド) に対して直接リンクします。

インポートライブラリ

DBTools インタフェースのインポートライブラリは、32 ビットおよび 64 ビット Windows プラットフォーム用に提供されています。dbtlstm.lib ライブラリは、ソフトウェアインストールディレクトリの SDK¥Lib¥x86 サブディレクトリと SDK¥Lib¥x64 サブディレクトリにあります。

1.22.1.2 DBTools ライブラリの初期化とファイナライズ

他の DBTools 関数を使用する前に、DBToolsInit を呼び出す必要があります。DBTools ライブラリを使い終わったときは、DBToolsFini を呼び出してください。

DBToolsInit と DBToolsFini 関数の主な目的は、DBTools ライブラリがメッセージ DLL または共有オブジェクトをロードおよびアンロードできるようにすることです。メッセージライブラリには、DBTools が内部的に使用する、ローカライズされたバージョンのすべてのエラーメッセージとプロンプトが含まれています。DBToolsFini を呼び出さないと、メッセージライブラリのリファレンスカウントが減分されず、アンロードされません。そのため、DBToolsInit と DBToolsFini の呼び出し回数が等しくなるよう注意してください。

次のコードは、DBTools を初期化してファイナライズする方法を示しています。

```
// Declarations
```

```
a_dbtools_info  info;
short           ret;
//Initialize the a_dbtools_info structure
memset( &info, 0, sizeof( a_dbtools_info) );
info.errorrtn = (MSG_CALLBACK)MyErrorRtn;
// initialize the DBTools library
ret = DBToolsInit( &info );
if( ret != EXIT_OKAY ) {
    // library initialization failed
    ...
}
// call some DBTools routines ...
...
// finalize the DBTools library
DBToolsFini( &info );
```

1.22.1.3 DBTools 関数呼び出し

すべてのツールは、まず構造体に値を設定し、次に DBTools ライブラリの関数 (またはエントリポイント) を呼び出すことによって実行します。各エントリポイントには、引数として単一構造体へのポインタを渡します。

次の例は、Windows オペレーティングシステムでの DBBackup 関数の使い方を示しています。

```
// Initialize the structure
a_backup_db backup_info;
memset( &backup_info, 0, sizeof( backup_info ) );
// Fill out the structure
backup_info.version = DB_TOOLS_VERSION_NUMBER;
backup_info.output_dir = "c:¥¥backup";
backup_info.connectparms ="UID=DBA;PWD=sql;DBF=demo.db";
backup_info.confirmrtn = (MSG_CALLBACK) ConfirmRtn ;
backup_info.errorrtn = (MSG_CALLBACK) ErrorRtn ;
backup_info.msgrtn = (MSG_CALLBACK) MessageRtn ;
backup_info.statusrtn = (MSG_CALLBACK) StatusRtn ;
backup_info.backup_database = TRUE;
// start the backup
DBBackup( &backup_info );
```

関連情報

SQL Anywhere データベースツール C API リファレンス [706 ページ]

1.22.1.4 コールバック関数

DBTools 構造体には MSG_CALLBACK 型または SHOULD_STOP_CALLBACK 型の要素がいくつかあります。それらはコールバック関数へのポインタです。

コールバック関数の使用

コールバック関数を使用すると、DBTools 関数はオペレーションの制御をユーザの呼び出し側アプリケーションに戻すことができます。DBTools ライブラリではコールバック関数を使用し、DBTools 関数からユーザに送られたメッセージを、次の 4 つの目的のために処理します。

確認

ユーザがアクションを確認する必要がある場合に呼び出されます。たとえば、バックアップディレクトリが存在しない場合、ツールライブラリはディレクトリを作成する必要があるか確認を求めます。

エラーメッセージ

オペレーション中にディスク容量が足りなくなった場合など、エラーが発生したときにメッセージを処理するために呼び出されます。

情報メッセージ

ツールがユーザにメッセージを表示するときに呼び出されます (アンロード中の現在のテーブル名など)。

ステータス情報

ツールがオペレーションのステータス (テーブルのアンロード処理の進捗率など) を表示するときに呼び出されます。

処理の停止 処理を停止する必要があるかどうかを決定する場合に呼び出されます。たとえば、`dbunload -aot` はこの関数を呼び出し、運用データベースのインクリメンタルバックアップとトランザクションログの再ビルドデータベースへの適用を続けるかどうかを決定します。

コールバック関数の構造体への割り当て

コールバックルーチンを構造体に直接割り当てることができます。次の文は、バックアップ構造体を使用した例です。

```
backup_info.errorrtn = (MSG_CALLBACK) MyFunction
```

次の文は、アンロード構造体を使用した例です。

```
unload_info.shouldstoprtn = (SHOULD_STOP_CALLBACK) MyStopFunction
```

MSG_CALLBACK と SHOULD_STOP_CALLBACK は `dllapi.h` ヘッダファイルで定義されています。ツールルーチンは、呼び出し側アプリケーションにメッセージ付きでコールバックできます。このメッセージは、ウィンドウ環境でも、文字ベースのシステムの標準出力でも、またはそれ以外のユーザインタフェースであっても、適切なユーザインタフェースに表示されます。

確認コールバック関数の例

次の確認ルーチンの例では、YES または NO をプロンプトに答えるようユーザに求め、ユーザの選択結果を戻します。

```
extern short _callback ConfirmRtn(
        char * question )
{
    int ret = IDNO;
    if( question != NULL ) {
        ret = MessageBox( HwndParent, question,
        "Confirm", MB_ICONEXCLAMATION|MB_YESNO );
    }
    return( ret == IDYES );
}
```

エラーコールバック関数の例

次はエラーメッセージ処理ルーチンの例です。エラーメッセージをウィンドウに表示します。

```
extern short _callback ErrorRtn(
        char * errorstr )
{
    if( errorstr != NULL ) {
        MessageBox( HwndParent, errorstr, "Backup Error", MB_ICONSTOP|MB_OK );
    }
    return( 0 );
}
```

メッセージコールバック関数の例

メッセージコールバック関数の一般的な実装では、メッセージを画面に表示します。

```
extern short _callback MessageRtn(
        char * messagestr )
{
    if( messagestr != NULL ) {
                    OutputMessageToWindow( messagestr );
    }
    return( 0 );
}
```

ステータスコールバック関数の例

ステータスコールバックルーチンは、ツールがオペレーションのステータス (テーブルのアンロード処理の進捗率など) を表示する必要がある場合に呼び出されます。一般的な実装では、メッセージを画面に表示するだけです。

```
extern short _callback StatusRtn(
        char * statusstr )
{
```

```
    if( statusstr != NULL ) {
        OutputMessageToWindow( statusstr );
    return( 0 );
}
```

処理停止コールバック関数の例

処理停止コールバックルーチンは、ツールが処理を続行すべきか停止すべきかを決定する必要がある場合に呼び出されます。実装は、処理停止ボタンを押すかどうかに基づいている場合があります。この例では、StopProcessing グローバル変数をゼロ以外の値に設定し、処理停止ボタンイベントを処理するコードがあることを想定しています。

```
extern short _callback StopProcessingRtn()
{
    return( StopProcessing );
}
```

1.22.1.5 バージョン番号と互換性

各構造体にはバージョン番号を示すメンバーがあります。DBTools の関数を呼び出す前に、このバージョンフィールドに、アプリケーション開発に使用した DBTools ライブラリのバージョン番号を設定しておきます。

DBTools ライブラリの現在のバージョンは、`dbtools.h` ヘッダファイルをインクルードするときに定義されます。

次の例では、現在のバージョンを a_backup_db 構造体のインスタンスに割り当てます。

```
backup_info.version = DB_TOOLS_VERSION_NUMBER;
```

バージョン番号を使用することによって、DBTools ライブラリのバージョンが新しくなってもアプリケーションを継続して使用できます。DBTools 関数は、DBTools 構造体に新しいメンバーが追加されても、アプリケーションが提示するバージョン番号を使用してアプリケーションが作動できるようにします。

DBTools 構造体が更新されたり、新しいバージョンのソフトウェアがリリースされると、バージョン番号が大きくなります。DB_TOOLS_VERSION_NUMBER を使用し、新しいバージョンの DBTools ヘッダファイルを使用してアプリケーションを再構築する場合は、新しいバージョンの DBTools ライブラリを配備してください。

1.22.1.6 ビットフィールド

DBTools 構造体の多くは、ビットフィールドを使用してブール情報を効率よく格納しています。

たとえば、バックアップ構造体には次のビットフィールドがあります。

```
a_bit_field     backup_database : 1;
a_bit_field     backup_logfile  : 1;
a_bit_field     no_confirm  : 1;
a_bit_field     quiet       : 1;
a_bit_field     rename_log  : 1;
a_bit_field     truncate_log : 1;
```

```
a_bit_field     rename_local_log: 1;
a_bit_field     server_backup   : 1;
```

各ビットフィールドは 1 ビット長です。これは、構造体宣言のコロンの右側の 1 によって示されています。a_bit_field に割り当てられている値に応じて、特定のデータ型が使用されます。a_bit_field は `dbtools.h` の先頭で設定され、設定値はオペレーティングシステムに依存します。

0 または 1 の値をビットフィールドに割り当てて、構造体のブール情報を渡します。

1.22.1.7 DBTools の例

このサンプルとコンパイル手順は、*%SQLANYSAMP17%*`¥SQLAnywhere¥DBTools` ディレクトリにあります。

サンプルプログラムコードは `main.cpp` にあります。このサンプルは、DBTools ライブラリを使用してデータベースのバックアップを作成する方法を示しています。

1.22.2 SQL Anywhere データベースツール C API リファレンス

アプリケーションプログラムは、DBTools API ライブラリと情報を交換するために構造体を使用します。`dbrmt.h` に定義されている a_remote_sql 構造体を除くすべての構造体は、`dbtools.h` に定義されています。

ヘッダファイル

- `dbtools.h`
- `dbrmt.h`

データベースツールの構造体

構造体の要素の多くは、対応するユーティリティのコマンドラインオプションに対応しています。たとえば、0 または 1 の値を取ることができるクワイエット (`quiet`) と呼ばれるメンバーを持つ構造体があります。このメンバーは、多くのユーティリティが使用するクワイエットオペレーション (-q) オプションに対応しています。

このファイルで定義されているデータ構造体は、ファイルの適用先となるメジャーバージョンの SQL Anywhere の DBTools API で使用できます。たとえば、バージョン 16.0.0 の `dbtools.h` ファイルを使用して構築されたアプリケーションからは、`dbtool12.dll` や `dbtool17.dll` にアクセスできません。

メジャーバージョンの中では、同じメジャーバージョンの以前または以後のバージョンの `dbtools.h` を使用して構築されたアプリケーションが動作し続けるように、構造体に変更が加えられます。以前のバージョンを使用して構築されたアプリケーションからは、新しいフィールドにアクセスできません。そのため、DBTools では、以前のバージョンと同じ動作になるデフォルト値が用意されています。通常はこの結果として、ポイントリリースで追加された新しいフィールドが構造体の最後に出現します。

同じメジャーリリースの中の以後のバージョンを使用して構築されたアプリケーションの動作は、構造体のバージョンフィールドで指定された値に依存します。指定したバージョン番号が以前のバージョンに対応している場合は、そのバージョンの `dbtools.h` を使用してアプリケーションを構築した場合とまったく同様に、アプリケーションから以前のバージョンの DBTools DLL を呼び出すことができます。指定したバージョン番号が現在のバージョンの場合は、以前のバージョンの DBTools DLL を使おうとするとエラーになります。

バージョン番号の定義の詳細については、`dbtlvers.h` を参照してください。

このセクションの内容:

DBBackup(const a_backup_db *) メソッド [710 ページ]
データベースをバックアップします。

DBChangeLogName(a_change_log *) メソッド [710 ページ]
トランザクションログファイルの名前を変更します。

DBCreate(a_create_db *) メソッド [711 ページ]
データベースを作成します。

DBCreatedVersion(a_db_version_info *) メソッド [712 ページ]
データベースファイルを作成した SQL Anywhere のバージョンをデータベースを起動せずに判別します。

DBErase(const an_erase_db *) メソッド [713 ページ]
データベースファイルかトランザクションファイルまたはその両方を消去します。

DBInfo(a_db_info *) メソッド [713 ページ]
データベースファイルに関する情報を戻します。

DBInfoDump(a_db_info *) メソッド [714 ページ]
データベースファイルに関する情報を戻します。

DBInfoFree(a_db_info *) メソッド [715 ページ]
DBInfoDump 関数の呼び出し後に、リソースを解放します。

DBLicense(const a_dblic_info *) メソッド [716 ページ]
データベースサーバのライセンス情報を修正またはレポートします。

DBLogFileInfo(const a_log_file_info *) メソッド [716 ページ]
実行中でないデータベースファイルのログファイルとミラーログファイルパスを戻します。

DBRemoteSQL(a_remote_sql *) メソッド [717 ページ]
SQL Remote Message Agent にアクセスします。

DBSynchronizeLog(const a_sync_db *) メソッド [718 ページ]
データベースを Mobile Link サーバと同期させます。

DBToolsFini(const a_dbtools_info *) メソッド [718 ページ]
アプリケーションが DBTools ライブラリを使い終わったときに、参照カウンタを減分して、リソースを解放します。

DBToolsInit(const a_dbtools_info *) メソッド [719 ページ]
DBTools ライブラリを使用できるよう準備します。

DBToolsVersion(void) メソッド [720 ページ]
DBTools ライブラリのバージョン番号を戻します。

DBTranslateLog(const a_translate_log *) メソッド [720 ページ]
トランザクションログファイルを SQL に変換します。

DBTruncateLog(const a_truncate_log *) メソッド [721 ページ]
トランザクションログファイルをトランケートします。

DBUnload(an_unload_db *) メソッド [722 ページ]
データベースをアンロードします。

DBUpgrade(const an_upgrade_db *) メソッド [723 ページ]
データベースファイルをアップグレードします。

DBValidate(const a_validate_db *) メソッド [723 ページ]
データベースの全部または一部を検証します。

オートチューン列挙 [724 ページ]
a_backup_db 構造体でライタの自動チューニングを制御するために使用します。

チェックポイント列挙 [725 ページ]
チェックポイントログのコピーを制御するために、a_backup_db 構造体で使用されます。

履歴列挙 [725 ページ]
a_backup_db 構造体でバックアップ履歴の有効化を制御するために使用します。

InMemory 列挙体 [726 ページ]
StartLine 接続パラメータに追加する必要があるインメモリモードスイッチ

LiveValidation 列挙体 [727 ページ]
a_validate_db 構造体で使用される、実行中のライブ検証のタイプ。

埋め込み列挙 [727 ページ]
ブランク埋め込み列挙は、a_create_db 内の blank_pad 設定を指定します。

単位列挙 [728 ページ]
db_size_unit の値を指定するために、a_create_db 構造体で使用されます。

アンロード列挙 [729 ページ]
an_unload_db 構造体で使用される、実行中のアンロードのタイプ。

ユーザリスト列挙 [729 ページ]
a_translate_log 構造体で使用される、ユーザリストのタイプ。

検証列挙 [730 ページ]
a_validate_db 構造体で使用される、実行中の検証のタイプ。

冗長列挙 [731 ページ]
冗長列挙は、出力のボリュームを指定します。

バージョン列挙 [731 ページ]
データベースを最初に作成した SQL Anywhere のバージョンを示すために、a_db_version_info 構造体で使用されます。

a_backup_db 構造体 [732 ページ]
DBTools ライブラリを使用してバックアップタスクを実行するために必要な情報を格納します。

a_change_log 構造体 [735 ページ]
DBTools ライブラリを使用して dblog タスクを実行するために必要な情報を格納します。

a_create_db 構造体 [738 ページ]
DBTools ライブラリを使用してデータベースを作成するために必要な情報を格納します。

a_db_info 構造体 [742 ページ]

DBTools ライブラリを使用して DBInfo 情報を戻すために必要な情報を格納します。

a_db_version_info 構造体 [745 ページ]
データベースの作成に使用された SQL Anywhere のバージョンに関する情報を保持します。

a_dblic_info 構造体 [746 ページ]
ライセンス情報などを格納します。

a_dbtools_info 構造体 [747 ページ]
DBTools ライブラリ呼び出しの初期化とファイナライズに使用する DBTools 情報コールバックです。

a_log_file_info 構造体 [748 ページ]
実行中でないデータベースのログファイルとミラーログファイル情報の取得に使用します。

a_name 構造体 [749 ページ]
名前の変数リストを指定します。

a_remote_sql 構造体 [749 ページ]
DBTools ライブラリを使用する dbremote ユーティリティが必要とする情報を格納します。

a_sync_db 構造体 [756 ページ]
DBTools ライブラリを使用する dbmlsync ユーティリティが必要とする情報を格納します。

a_syncpub 構造体 [764 ページ]
dbmlsync ユーティリティが必要とする情報を格納します。

a_sysinfo 構造体 [765 ページ]
DBTools ライブラリを使用する dbinfo と dbunload ユーティリティが必要とする情報を格納します。

a_table_info 構造体 [766 ページ]
a_db_info 構造体の一部として必要なテーブルに関する情報を格納します。

a_translate_log 構造体 [767 ページ]
DBTools ライブラリを使用してトランザクションログを変換するために必要な情報を格納します。

a_truncate_log 構造体 [771 ページ]
DBTools ライブラリを使用してトランザクションログをトランケートするために必要な情報を格納します。

a_validate_db 構造体 [772 ページ]
DBTools ライブラリを使用してデータベースを検証するために必要な情報を格納します。

an_erase_db 構造体 [774 ページ]
DBTools ライブラリを使用してデータベースを消去するために必要な情報を格納します。

an_unload_db 構造体 [775 ページ]
DBTools ライブラリを使用してデータベースをアンロードするため、または SQL Remote でリモートデータベースを抽出するために必要な情報を格納します。

an_upgrade_db 構造体 [783 ページ]
DBTools ライブラリを使用してデータベースをアップグレードするために必要な情報を格納します。

DBT_USE_DEFAULT_MIN_PWD_LEN 変数 [785 ページ]
DBT_USE_DEFAULT_MIN_PWD_LEN は、新しいデータベースでデフォルトの最小パスワード長を使用する (つまり、特定の長さを指定していない) ことを示します。

1.22.2.1 DBBackup(const a_backup_db *) メソッド

データベースをバックアップします。

構文

```
_crtn short _entry DBBackup (const a_backup_db * pdb)
```

パラメータ

pdb 正しく初期化された a_backup_db 構造体へのポインタ。

戻り値

リターンコード。ソフトウェアコンポーネント終了コードにリストされています。

備考

この関数は、dbbackup ユーティリティによって使用されます。

DBBackup 関数は、すべてのクライアント側のデータベースバックアップタスクを管理します。

サーバ側のバックアップを実行するには、BACKUP DATABASE 文を使用します。

関連情報

a_backup_db 構造体 [732 ページ]

1.22.2.2 DBChangeLogName(a_change_log *) メソッド

トランザクションログファイルの名前を変更します。

構文

```
_crtn short _entry DBChangeLogName (a_change_log * pcl)
```

パラメータ

pcl 正しく初期化された a_change_log 構造体へのポインタ。

戻り値

リターンコード。ソフトウェアコンポーネント終了コードにリストされています。

備考

この関数は、dblog ユーティリティによって使用されます。

トランザクションログユーティリティ (dblog) に -t オプションを指定すると、トランザクションログの名前が変更されます。DBChangeLogName は、この機能に対するプログラムインタフェースです。

関連情報

a_change_log 構造体 [735 ページ]

1.22.2.3 DBCreate(a_create_db *) メソッド

データベースを作成します。

構文

```
_crtn short _entry DBCreate (a_create_db * pcdb)
```

パラメータ

pcdb 正しく初期化された a_create_db 構造体へのポインタ。

戻り値

リターンコード。ソフトウェアコンポーネント終了コードにリストされています。

備考

この関数は、dbinit ユーティリティによって使用されます。

関連情報

a_create_db 構造体 [738 ページ]

1.22.2.4 DBCreatedVersion(a_db_version_info *) メソッド

データベースファイルを作成した SQL Anywhere のバージョンをデータベースを起動せずに判別します。

構文

```
_crtn short _entry DBCreatedVersion (a_db_version_info * pdvi)
```

パラメータ

pdvi 正しく初期化された a_db_version_info 構造体へのポインタ。

戻り値

リターンコード。ソフトウェアコンポーネント終了コードにリストされています。

備考

現在、この関数はバージョン 9 以前で作成されたデータベースとバージョン 10 以降で作成されたデータベースを区別するだけです。

失敗したことを示すコードが返された場合、バージョン情報は設定されません。

関連情報

a_db_version_info 構造体 [745 ページ]

バージョン列挙 [731 ページ]

1.22.2.5 DBErase(const an_erase_db *) メソッド

データベースファイルかトランザクションファイルまたはその両方を消去します。

構文

```
_crtn short _entry DBErase (const an_erase_db * pedb)
```

パラメータ

pedb 正しく初期化された an_erase_db 構造体へのポインタ。

戻り値

リターンコード。ソフトウェアコンポーネント終了コードにリストされています。

備考

この関数は、dberase ユーティリティによって使用されます。

関連情報

an_erase_db 構造体 [774 ページ]

1.22.2.6 DBInfo(a_db_info *) メソッド

データベースファイルに関する情報を戻します。

構文

```
_crtn short _entry DBInfo (a_db_info * pdbi)
```

パラメータ

pdbi 正しく初期化された a_db_info 構造体へのポインタ。

戻り値

リターンコード。ソフトウェアコンポーネント終了コードにリストされています。

備考

この関数は、dbinfo ユーティリティによって使用されます。

関連情報

a_db_info 構造体 [742 ページ]
DBInfoDump(a_db_info *) メソッド [714 ページ]
DBInfoFree(a_db_info *) メソッド [715 ページ]

1.22.2.7 DBInfoDump(a_db_info *) メソッド

データベースファイルに関する情報を戻します。

構文

```
_crtn short _entry DBInfoDump (a_db_info * pdbi)
```

パラメータ

pdbi 正しく初期化された a_db_info 構造体へのポインタ。

戻り値

リターンコード。ソフトウェアコンポーネント終了コードにリストされています。

備考

この関数を使用するのは、-u オプションが指定された dbinfo ユーティリティです。

関連情報

a_db_info 構造体 [742 ページ]
DBInfo(a_db_info *) メソッド [713 ページ]
DBInfoFree(a_db_info *) メソッド [715 ページ]

1.22.2.8 DBInfoFree(a_db_info *) メソッド

DBInfoDump 関数の呼び出し後に、リソースを解放します。

構文

```
_crtn short _entry DBInfoFree (a_db_info * pdbi)
```

パラメータ

pdbi 正しく初期化された a_db_info 構造体へのポインタ。

戻り値

リターンコード。ソフトウェアコンポーネント終了コードにリストされています。

関連情報

a_db_info 構造体 [742 ページ]
DBInfo(a_db_info *) メソッド [713 ページ]
DBInfoDump(a_db_info *) メソッド [714 ページ]

1.22.2.9 DBLicense(const a_dblic_info *) メソッド

データベースサーバのライセンス情報を修正またはレポートします。

構文

```
_crtn short _entry DBLicense (const a_dblic_info * pdi)
```

パラメータ

pdi 正しく初期化された a_dblic_info 構造体へのポインタ。

戻り値

リターンコード。ソフトウェアコンポーネント終了コードにリストされています。

関連情報

a_dblic_info 構造体 [746 ページ]

1.22.2.10 DBLogFileInfo(const a_log_file_info *) メソッド

実行中でないデータベースファイルのログファイルとミラーログファイルパスを戻します。

構文

```
_crtn short _entry DBLogFileInfo (const a_log_file_info * plfi)
```

パラメータ

plfi 正しく初期化された a_log_file_info 構造体へのポインタ。

戻り値

リターンコード。ソフトウェアコンポーネント終了コードにリストされています。

備考

この関数は、SQL Anywhere 10.0.0 以降を使用して作成されたデータベースにのみ機能します。

関連情報

a_log_file_info 構造体 [748 ページ]

1.22.2.11 DBRemoteSQL(a_remote_sql *) メソッド

SQL Remote Message Agent にアクセスします。

構文

```
_crtn short _entry DBRemoteSQL (a_remote_sql * prs)
```

パラメータ

prs 正しく初期化された a_remote_sql 構造体へのポインタ。

戻り値

リターンコード。ソフトウェアコンポーネント終了コードにリストされています。

関連情報

a_remote_sql 構造体 [749 ページ]

1.22.2.12 DBSynchronizeLog(const a_sync_db *) メソッド

データベースを Mobile Link サーバと同期させます。

構文

```
_crtn short _entry DBSynchronizeLog (const a_sync_db * psdb)
```

パラメータ

psdb 正しく初期化された a_sync_db 構造体へのポインタ。

戻り値

リターンコード。ソフトウェアコンポーネント終了コードにリストされています。

関連情報

a_sync_db 構造体 [756 ページ]

1.22.2.13 DBToolsFini(const a_dbtools_info *) メソッド

アプリケーションが DBTools ライブラリを使い終わったときに、参照カウンタを減分して、リソースを解放します。

構文

```
_crtn short _entry DBToolsFini (const a_dbtools_info * pdi)
```

パラメータ

pdi 正しく初期化された a_dbtools_info 構造体へのポインタ。

戻り値

リターンコード。ソフトウェアコンポーネント終了コードにリストされています。

備考

DBTools インタフェースを使用するアプリケーションは、終了時に DBToolsFini 関数を呼び出す必要があります。呼び出さない場合は、メモリリソースが失われる可能性があります。

関連情報

a_dbtools_info 構造体 [747 ページ]
DBToolsInit(const a_dbtools_info *) メソッド [719 ページ]

1.22.2.14 DBToolsInit(const a_dbtools_info *) メソッド

DBTools ライブラリを使用できるよう準備します。

構文

```
_crtn short _entry DBToolsInit (const a_dbtools_info * pdi)
```

パラメータ

pdi 正しく初期化された a_dbtools_info 構造体へのポインタ。

戻り値

リターンコード。ソフトウェアコンポーネント終了コードにリストされています。

備考

DBToolsInit 関数の主な目的は、SQL Anywhere メッセージライブラリをロードすることです。メッセージライブラリには、DBTools ライブラリ内の関数が使用する、ローカライズされたバージョンのエラーメッセージとプロンプトが含まれています。

DBTools インタフェースを使用するアプリケーションを開始する時は、他の DBTools 関数を呼び出す前に、DBToolsInit 関数を呼び出す必要があります。

関連情報

a_dbtools_info 構造体 [747 ページ]
DBToolsFini(const a_dbtools_info *) メソッド [718 ページ]

1.22.2.15 DBToolsVersion(void) メソッド

DBTools ライブラリのバージョン番号を戻します。

構文

```
_crtn short _entry DBToolsVersion (void )
```

備考

DBToolsVersion 関数を使用して、DBTools ライブラリのバージョンが、アプリケーションの開発に使用したバージョンより古くないことを確認します。DBTools のバージョンが開発時より新しい場合はアプリケーションを実行できますが、古い場合は実行できません。

1.22.2.16 DBTranslateLog(const a_translate_log *) メソッド

トランザクションログファイルを SQL に変換します。

構文

```
_crtn short _entry DBTranslateLog (const a_translate_log * ptl)
```

パラメータ

ptl 正しく初期化された a_translate_log 構造体へのポインタ。

戻り値

リターンコード。ソフトウェアコンポーネント終了コードにリストされています。

備考

この関数は、dbtran ユーティリティによって使用されます。

関連情報

a_translate_log 構造体 [767 ページ]

1.22.2.17 DBTruncateLog(const a_truncate_log *) メソッド

トランザクションログファイルをトランケートします。

構文

```
_crtn short _entry DBTruncateLog (const a_truncate_log * ptl)
```

パラメータ

ptl 正しく初期化された a_truncate_log 構造体へのポインタ。

戻り値

リターンコード。ソフトウェアコンポーネント終了コードにリストされています。

備考

この関数は、dbbackup ユーティリティによって使用されます。

関連情報

a_truncate_log 構造体 [771 ページ]

1.22.2.18 DBUnload(an_unload_db *) メソッド

データベースをアンロードします。

構文

```
_crtn short _entry DBUnload (an_unload_db * pudb)
```

パラメータ

pudb 正しく初期化された an_unload_db 構造体へのポインタ。

戻り値

リターンコード。ソフトウェアコンポーネント終了コードにリストされています。

備考

この関数は、dbunload ユーティリティと dbxtract ユーティリティによって使用されます。

関連情報

an_unload_db 構造体 [775 ページ]

1.22.2.19 DBUpgrade(const an_upgrade_db *) メソッド

データベースファイルをアップグレードします。

構文

```
_crtn short _entry DBUpgrade (const an_upgrade_db * pudb)
```

パラメータ

pudb 正しく初期化された an_upgrade_db 構造体へのポインタ。

戻り値

リターンコード。ソフトウェアコンポーネント終了コードにリストされています。

備考

この関数は、dbupgrad ユーティリティによって使用されます。

関連情報

an_upgrade_db 構造体 [783 ページ]

1.22.2.20 DBValidate(const a_validate_db *) メソッド

データベースの全部または一部を検証します。

構文

```
_crtn short _entry DBValidate (const a_validate_db * pvdb)
```

パラメータ

pvdb 正しく初期化された a_validate_db 構造体へのポインタ。

戻り値

リターンコード。ソフトウェアコンポーネント終了コードにリストされています。

備考

この関数は、dbvalid ユーティリティによって使用されます。

警告: テーブルまたはデータベース全体の検証は、データベースに変更を加えている接続がない場合に実行してください。そうしないと、実際に破損していなくても、何らかの形でデータベースが破損したことを示す重大なエラーがレポートされます。

関連情報

a_validate_db 構造体 [772 ページ]

1.22.2.21 オートチューン列挙

a_backup_db 構造体でライタの自動チューニングを制御するために使用します。

構文

```
enum Autotune
```

メンバー

メンバー名	説明
BACKUP_AUTO_TUNE_UNSPECIFIED	AUTO TUNE WRITERS 句を未指定にするために使用します。
BACKUP_AUTO_TUNE_ON	AUTO TUNE WRITERS ON 句の生成に使用します。
BACKUP_AUTO_TUNE_OFF	AUTO TUNE WRITERS OFF 句の生成に使用します。

関連情報

a_backup_db 構造体 [732 ページ]
DBBackup(const a_backup_db *) メソッド [710 ページ]

1.22.2.22 チェックポイント列挙

チェックポイントログのコピーを制御するために、a_backup_db 構造体で使用されます。

構文

```
enum Checkpoint
```

メンバー

メンバー名	説明
BACKUP_CHKPT_LOG_COPY	WITH CHECKPOINT LOG COPY 句の生成に使用します。
BACKUP_CHKPT_LOG_NOCOPY	WITH CHECKPOINT LOG NOCOPY 句の生成に使用します。
BACKUP_CHKPT_LOG_RECOVER	WITH CHECKPOINT LOG RECOVER 句の生成に使用します。
BACKUP_CHKPT_LOG_AUTO	WITH CHECKPOINT LOG AUTO 句の生成に使用します。
BACKUP_CHKPT_LOG_DEFAULT	WITH CHECKPOINT 句を省略するのに使用します。

関連情報

a_backup_db 構造体 [732 ページ]
DBBackup(const a_backup_db *) メソッド [710 ページ]

1.22.2.23 履歴列挙

a_backup_db 構造体でバックアップ履歴の有効化を制御するために使用します。

構文

```
enum History
```

メンバー

メンバー名	説明
BACKUP_HISTORY_UNSPECIFIED	HISTORY 句を未指定にするために使用します。
BACKUP_HISTORY_ON	HISTORY ON 句の生成に使用します。
BACKUP_HISTORY_OFF	HISTORY OFF 句の生成に使用します。

関連情報

a_backup_db 構造体 [732 ページ]
DBBackup(const a_backup_db *) メソッド [710 ページ]

1.22.2.24 InMemory 列挙体

StartLine 接続パラメータに追加する必要があるインメモリモードスイッチ

構文

```
enum InMemory
```

メンバー

メンバー名	説明
IM_NONE	StartLine 接続パラメータを変更しません。
IM_V	StartLine 接続パラメータに "-im v" (インメモリ検証モード) を追加します。 dbvalid コマンドラインツールはデフォルトで IM_V を使用します。
IM_C	StartLine 接続パラメータに "-im c" (インメモリチェックポイント専用モード) を追加します。
IM_NW	StartLine 接続パラメータに "-im nw" (インメモリ非書き込みモード) を追加します。

関連情報

a_validate_db 構造体 [772 ページ]

DBValidate(const a_validate_db *) メソッド [723 ページ]

1.22.2.25 LiveValidation 列挙体

a_validate_db 構造体で使用される、実行中のライブ検証のタイプ。

構文

```
enum LiveValidation
```

メンバー

メンバー名	説明
VALIDATE_NO_LIVE_OPTION	通常どおり検証します。
VALIDATE_WITH_DATA_LOCK	必要なテーブルでデータロックを取得します。
VALIDATE_WITH_SNAPSHOT	スナップショットを使用して検証します。

関連情報

a_validate_db 構造体 [772 ページ]
DBValidate(const a_validate_db *) メソッド [723 ページ]

1.22.2.26 埋め込み列挙

ブランク埋め込み列挙は、a_create_db 内の blank_pad 設定を指定します。

構文

```
enum Padding
```

メンバー

メンバー名	説明
NO_BLANK_PADDING	ブランク埋め込みを使用しません。
BLANK_PADDING	ブランク埋め込みを使用します。

関連情報

a_create_db 構造体 [738 ページ]
DBCreate(a_create_db *) メソッド [711 ページ]

1.22.2.27 単位列挙

db_size_unit の値を指定するために、a_create_db 構造体で使用されます。

構文

```
enum Unit
```

メンバー

メンバー名	説明
DBSP_UNIT_NONE	単位を指定しません。
DBSP_UNIT_PAGES	サイズをページ単位で指定します。
DBSP_UNIT_BYTES	サイズをバイト単位で指定します。
DBSP_UNIT_KILOBYTES	サイズをキロバイト単位で指定します。
DBSP_UNIT_MEGABYTES	サイズをメガバイト単位で指定します。
DBSP_UNIT_GIGABYTES	サイズをギガバイト単位で指定します。
DBSP_UNIT_TERABYTES	サイズをテラバイト単位で指定します。

関連情報

a_create_db 構造体 [738 ページ]
DBCreate(a_create_db *) メソッド [711 ページ]

1.22.2.28 アンロード列挙

an_unload_db 構造体で使用される、実行中のアンロードのタイプ。

構文

```
enum Unload
```

メンバー

メンバー名	説明
UNLOAD_ALL	データとスキーマの両方をアンロードします。
UNLOAD_DATA_ONLY	データをアンロードします。スキーマはアンロードしません。dbunload -d オプションと同等です。
UNLOAD_NO_DATA	データなし。スキーマのみをアンロードします。dbunload -n オプションと同等です。
UNLOAD_NO_DATA_FULL_SCRIPT	データなし。再ロードスクリプトに LOAD/INPUT 文を追加します。dbunload -nl オプションと同等です。
UNLOAD_NO_DATA_NAME_ORDER	データなし。オブジェクトが名前順に出力されます。

関連情報

an_unload_db 構造体 [775 ページ]
DBUnload(an_unload_db *) メソッド [722 ページ]

1.22.2.29 ユーザリスト列挙

a_translate_log 構造体で使用される、ユーザリストのタイプ。

構文

```
enum UserList
```

メンバー

メンバー名	説明
DBTRAN_INCLUDE_ALL	全ユーザの操作を含めます。
DBTRAN_INCLUDE_SOME	提供されるユーザリスト上のユーザの操作だけを含めます。
DBTRAN_EXCLUDE_SOME	提供されるユーザリスト上のユーザの操作を除外します。

関連情報

a_translate_log 構造体 [767 ページ]
DBTranslateLog(const a_translate_log *) メソッド [720 ページ]

1.22.2.30 検証列挙

a_validate_db 構造体で使用される、実行中の検証のタイプ。

構文

```
enum Validation
```

メンバー

メンバー名	説明
VALIDATE_NORMAL	デフォルトのチェックのみで検証します。
VALIDATE_DATA	(旧式)
VALIDATE_INDEX	(旧式)
VALIDATE_EXPRESS	エクスプレスチェックで検証します。dbvalid -fx オプションと同等です。
VALIDATE_FULL	(旧式)
VALIDATE_CHECKSUM	データベースチェックサムを検証します。dbvalid -s オプションと同等です。
VALIDATE_DATABASE	データベースを検証します。dbvalid -d オプションと同等です。
VALIDATE_COMPLETE	可能な検証をすべて実行します。

関連情報

a_validate_db 構造体 [772 ページ]
DBValidate(const a_validate_db *) メソッド [723 ページ]

1.22.2.31 冗長列挙

冗長列挙は、出力のボリュームを指定します。

構文

```
enum Verbosity
```

メンバー

メンバー名	説明
VB_QUIET	出力なし。
VB_NORMAL	通常の出力量。
VB_VERBOSE	冗長出力。デバッグ用。

関連情報

a_create_db 構造体 [738 ページ]
DBCreate(a_create_db *) メソッド [711 ページ]

1.22.2.32 バージョン列挙

データベースを最初に作成した SQL Anywhere のバージョンを示すために、a_db_version_info 構造体で使用されます。

構文

```
enum Version
```

メンバー

メンバー名	説明	値
VERSION_UNKNOWN	データベースを最初に作成した SQL Anywhere のバージョンを判別できません。	0
VERSION_PRE_10	データベースは、SQL Anywhere バージョン 9 以前を使用して作成されています。	9
VERSION_10	データベースは、SQL Anywhere バージョン 10 を使用して作成されています。	10
VERSION_11	データベースは、SQL Anywhere バージョン 11 を使用して作成されています。	11
VERSION_12	データベースは、SQL Anywhere バージョン 12 を使用して作成されています。	12
VERSION_16	データベースは、SQL Anywhere バージョン 16 を使用して作成されています。	16
VERSION_17	データベースは、SQL Anywhere バージョン 17 を使用して作成されています。	17

関連情報

a_db_version_info 構造体 [745 ページ]
DBCreatedVersion(a_db_version_info *) メソッド [712 ページ]

1.22.2.33 a_backup_db 構造体

DBTools ライブラリを使用してバックアップタスクを実行するために必要な情報を格納します。

構文

```
typedef struct a_backup_db
```

メンバー

a_backup_db のすべてのメンバー (継承されたメンバーも含みます) を以下に示します。

変数

変更子とタイプ	変数	説明
public unsigned short	version	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。
public MSG_CALLBACK	errorrtn	エラーメッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	msgrtn	情報メッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	confirmrtn	確認要求コールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	statusrtn	ステータスメッセージコールバックルーチンのアドレス、または NULL。
public const char *	output_dir	バックアップ用出力ディレクトリのパス。たとえば、"c:¥backup" などとなります。
public const char *	connectparms	データベース接続に必要なパラメータ。 次のような接続文字列のフォームを取ります。"UID=DBA;PWD=sql;DBF=demo.db". データベースサーバは、接続文字列の START パラメータによって起動されます。たとえば、"START=c:¥SQLAny17¥bin32¥dbsrv17.exe" などとなります。 次に START パラメータを含んだ完全な接続文字列の例を示します。"UID=DBA;PWD=sql;DBF=demo.db;START=c:¥SQLAny17¥bin32¥dbsrv17.exe"
public const char *	hotlog_filename	ライブバックアップファイルのファイル名。 dbbackup -l オプションによって設定されます。
public a_sql_uint32	page_blocksize	データブロック内のページ数。 0 に設定すると、デフォルトは 128 になります。dbbackup -b オプションによって設定されます。
public char	chkpt_log_type	チェックポイントログのコピーを制御します。 BACKUP_CHKPT_LOG_COPY、BACKUP_CHKPT_LOG_NOCOPY、BACKUP_CHKPT_LOG_RECOVER、BACKUP_CHKPT_LOG_AUTO、BACKUP_CHKPT_LOG_DEFAULT のうちの 1 つを指定します。dbbackup -k オプションによって設定されます。

変更子とタイプ	変数	説明
public char	backup_interrupted	0 以外の場合にオペレーションが中断されたことを示します。
public a_bit_field	backup_database	データベースをバックアップします。 dbbackup -d オプションによって TRUE に設定されます。
public a_bit_field	backup_logfile	トランザクションログファイルをバックアップします。 dbbackup -t オプションによって TRUE に設定されます。
public a_bit_field	no_confirm	確認せずに操作します。 dbbackup -y オプションによって TRUE に設定されます。
public a_bit_field	quiet	メッセージを出さずに操作します。 dbbackup -q オプションによって TRUE に設定されます。
public a_bit_field	rename_log	トランザクションログの名前を変更します。 dbbackup -r オプションによって TRUE に設定されます。
public a_bit_field	truncate_log	トランザクションログを削除します。 dbbackup -x オプションによって TRUE に設定されます。
public a_bit_field	rename_local_log	トランザクションログのローカルバックアップの名前を変更します。 dbbackup -n オプションによって TRUE に設定されます。
public a_bit_field	server_backup	BACKUP DATABASE を使用してサーバでバックアップを実行します。 dbbackup -s オプションによって TRUE に設定されます。
public a_bit_field	progress_messages	進行状況のメッセージを表示します。 dbbackup -p オプションによって TRUE に設定されます。
public a_bit_field	wait_before_start	バックアップ開始前にオープントランザクションが閉じられるまで待機します。 dbbackup -wb オプションによって TRUE に設定されます。

変更子とタイプ	変数	説明
public a_bit_field	wait_after_end	バックアップ終了前にオープントランザクションが閉じられるまで待機します。 dbbackup -wa オプションによって TRUE に設定されます。
public const char *	backup_comment	WITH COMMENT 句に使用するコメント
public char	auto_tune_writers	オートチューンライタの有効/無効を切り替えます。 BACKUP_AUTO_TUNE_UNSPECIFIED、BACKUP_AUTO_TUNE_ON、または BACKUP_AUTO_TUNE_OFF のいずれかに設定します。AUTO TUNE WRITERS OFF 句の生成に使用します。dbbackup -aw[-] オプションによって設定されます。
public char	backup_history	バックアップ履歴。 BACKUP_HISTORY_UNSPECIFIED、BACKUP_HISTORY_ON、または BACKUP_HISTORY_OFF のいずれかに設定します。dbbackup -h[-] オプションによって設定されます。

関連情報

チェックポイント列挙 [725 ページ]
DBBackup(const a_backup_db *) メソッド [710 ページ]

1.22.2.34 a_change_log 構造体

DBTools ライブラリを使用して dblog タスクを実行するために必要な情報を格納します。

構文

```
typedef struct a_change_log
```

メンバー

a_change_log のすべてのメンバー (継承されたメンバーも含みます) を以下に示します。

変数

変更子とタイプ	変数	説明
public unsigned short	version	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。
public MSG_CALLBACK	errorrtn	エラーメッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	msgrtn	情報メッセージコールバックルーチンのアドレス、または NULL。
public const char *	dbname	データベースファイル名。
public const char *	logname	トランザクションログファイル名。NULL の場合、ログは作成されません。
public const char *	mirrorname	トランザクションログミラーファイルの新しい名前。dblog -m オプションと同等です。
public const char *	zap_current_offset	現在のオフセットを指定の値に変更します。 このパラメータは、アンロードと再ロードの後で dbremote または dbmlsync の設定に合わせてトランザクションログをリセットする場合にだけ使用します。dblog -x オプションと同等です。
public const char *	zap_starting_offset	開始オフセットを指定の値に変更します。 このパラメータは、アンロードと再ロードの後で dbremote または dbmlsync の設定に合わせてトランザクションログをリセットする場合にだけ使用します。dblog -z オプションと同等です。
public const char *	zap_timeline	タイムラインを指定の値に変更します。 このパラメータは、アンロードと再ロードの後で dbremote または dbmlsync の設定に合わせてトランザクションログをリセットする場合に使用します。
public const char *	encryption_key	データベースファイルの暗号化キー。dblog -ek または -ep オプションと同等です。
public unsigned short	generation_number	新しい世代番号。予約済み。0 を使用します。
public a_bit_field	query_only	1 の場合、トランザクションログの名前を表示するだけです。0 の場合、ログ名の変更を許可します。
public a_bit_field	quiet	メッセージを出さずに操作します。 dblog -q オプションによって TRUE に設定されます。

変更子とタイプ	変数	説明
public a_bit_field	change_mirrorname	TRUE に設定した場合、ミラーログ名の変更を許可します。 dblog -m、-n、または -r オプションによって TRUE に設定されます。
public a_bit_field	change_logname	TRUE に設定した場合、トランザクションログ名の変更を許可します。 dblog -n または -t オプションによって TRUE に設定されます。
public a_bit_field	ignore_ltm_trunc	予約済み。FALSE を使用します。
public a_bit_field	ignore_remote_trunc	SQL Remote 用。 delete_old_logs オプションのために保存されているオフセットをリセットして、トランザクションログが不要になったときに削除できるようにします。dblog -ir オプションによって TRUE に設定されます。
public a_bit_field	set_generation_number	予約済み。FALSE を使用します。
public a_bit_field	ignore_dbsync_trunc	dbmlsync を使用している場合、delete_old_logs オプションのために保存されているオフセットをリセットして、トランザクションログが不要になったときに削除できるようにします。 dblog -is オプションによって TRUE に設定されます。
public a_sql_uint64	starting_offset	DBChangeLogName によって開始オフセットに設定されます。 この機能は、DB_TOOLS_VERSION_17_0_0 以降のバージョンに装備されてます。
public a_sql_uint64	current_relative_offset	dbname がデータベースファイルの場合、DBChangeLogName によって現在の相対オフセットに設定されます。 この機能は、DB_TOOLS_VERSION_17_0_0 以降のバージョンに装備されてます。
public a_sql_uint64	end_offset	dbname がトランザクションログファイルの場合、DBChangeLogName によって終了オフセットに設定されます。 この機能は、DB_TOOLS_VERSION_17_0_0 以降のバージョンに装備されてます。

関連情報

DBChangeLogName(a_change_log *) メソッド [710 ページ]

1.22.2.35 a_create_db 構造体

DBTools ライブラリを使用してデータベースを作成するために必要な情報を格納します。

構文

```
typedef struct a_create_db
```

メンバー

a_create_db のすべてのメンバー (継承されたメンバーも含みます) を以下に示します。

変数

変更子とタイプ	変数	説明
public unsigned short	version	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。
public MSG_CALLBACK	errorrtn	エラーメッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	msgrtn	情報メッセージコールバックルーチンのアドレス、または NULL。
public const char *	dbname	データベースファイル名。
public const char *	logname	新しいトランザクションログ名。 NULL は dbinit -n オプションと同じです。それ以外の場合は、設定する必要があります。
public const char *	startline	データベースサーバを開始するときに使用するコマンドライン。 たとえば、"c:¥SQLAny17¥bin32¥dbsrv17.exe" などとなります。NULL の場合、SQL Anywhere のデフォルトの START パラメータは "dbeng17 -gp <page_size> -c 10M" となります (page_size は以下で指定)。page_size >= 2048 の場合、"-c 10M" が追加されます。

変更子とタイプ	変数	説明
public const char *	default_collation	データベースの照合。dbinit -z オプションと同等です。
public const char *	nchar_collation	NULL でない場合、データベースの NCHAR 照合を行います。dbinit -zn オプションと同等です。
public const char *	encoding	文字セットエンコード。dbinit -ze オプションと同等です。
public const char *	mirrorname	トランザクションログミラー名。dbinit -m オプションと同等です。
public const char *	data_store_type	予約済み。NULL を使用します。
public const char *	encryption_key	データベースファイルの暗号化キー。 encrypt とともに使用すると、KEY 句が生成されます。dbinit -ek オプションと同等です。
public const char *	encryption_algorithm	暗号化アルゴリズム (AES、AES256、AES_FIPS、または AES256_FIPS)。 encrypt と encryption_key とともに使用すると、ALGORITHM 句が生成されます。dbinit -ea オプションと同等です。
public void *	iq_params	予約済み。NULL を使用します。
public int	db_size_unit	db_size とともに使用し、DBSP_UNIT_NONE、DBSP_UNIT_PAGES、DBSP_UNIT_BYTES、DBSP_UNIT_KILOBYTES、DBSP_UNIT_MEGABYTES、DBSP_UNIT_GIGABYTES、DBSP_UNIT_TERABYTES のうちいずれかを指定します。 DBSP_UNIT_NONE でない場合は、対応するキーワードを生成します (例: DATABASE SIZE 10 MB は db_size が 10 で db_size_unit が DBSP_UNIT_MEGABYTES の場合に生成されます)。データベースサイズ単位列挙を参照してください。
public unsigned int	db_size	0 でない場合、DATABASE SIZE 句を生成します。dbinit -dbs オプションと同等です。
public unsigned short	page_size	データベースのページサイズ。dbinit -p オプションと同等です。
public char	verbose	冗長列挙 (VB_QUIET、VB_NORMAL、VB_VERBOSE) を参照してください。

変更子とタイプ	変数	説明
public char	accent_sensitivity	'y' (はい)、'n' (いいえ)、または 'f' (フランス語) のいずれか。 ACCENT RESPECT、ACCENT IGNORE、ACCENT FRENCH 句のいずれかを生成します。
public char *	dba_uid	NULL でない場合、DBA USER xxx 句を生成します。dbinit -dba オプションと同等です。
public char *	dba_pwd	NULL でない場合、DBA PASSWORD xxx 句を生成します。dbinit -dba オプションと同等です。
public a_bit_field	blank_pad	NO_BLANK_PADDING または BLANK_PADDING のいずれかを設定します。 文字列の比較のときにブランクを有効とし、これを反映するインデックス情報を保持します。ブランク埋め込み列挙を参照してください。dbinit -b オプションと同等です。
public a_bit_field	respect_case	文字列の比較のときに大文字と小文字を区別するようにし、これを反映するインデックス情報を保持します。 dbinit -c オプションによって TRUE に設定されます。
public a_bit_field	encrypt	TRUE に設定した場合、ENCRYPTED ON 句が生成されます。encrypted_tables も設定されている場合は、ENCRYPTED TABLES ON 句が生成されます。 dbinit -e? オプションによって TRUE に設定されます。
public a_bit_field	avoid_view_collisions	TRUE に設定した場合、Watcom SQL 互換ビュー SYS.SYSCOLUMNS と SYS.SYSINDEXES の世代を除外します。 dbinit -k オプションによって TRUE に設定されます。
public a_bit_field	jconnect	TRUE に設定した場合、jConnect に必要なシステムプロシージャを含めます。 dbinit -i オプションによって FALSE に設定されます。

変更子とタイプ	変数	説明
public a_bit_field	checksum	ON の場合は TRUE に設定し、OFF の場合は FALSE に設定します。 CHECKSUM ON または CHECKSUM OFF 句のいずれかを生成します。dbinit -s オプションによって TRUE に設定されます。
public a_bit_field	encrypted_tables	TRUE に設定した場合、テーブルが暗号化されます。 encrypt とともに使用すると、ENCRYPTED ON 句の代わりに ENCRYPTED TABLE ON 句が生成されます。dbinit -et オプションによって TRUE に設定されます。
public a_bit_field	case_sensitivity_use_default	TRUE に設定した場合、ロケールの大文字と小文字の区別に関するデフォルトの設定を使用します。 この影響を受けるのは UCA だけです。TRUE に設定する場合は、CASE RESPECT 句を CREATE DATABASE 文に追加しないでください。
public a_bit_field	sys_proc_definer	バージョン 12.0.1 以前のシステムストアドプロシージャの SQL SECURITY モデルを保持する場合、TRUE に設定します。 dbinit -pd オプションによって TRUE に設定されます。
public unsigned short	min_pwd_len	データベースにおける新しいパスワードの最小長。 この機能は、DB_TOOLS_VERSION_17_0_0 以降のバージョンに装備されてます。

関連情報

埋め込み列挙 [727 ページ]
単位列挙 [728 ページ]
冗長列挙 [731 ページ]
DBCreate(a_create_db *) メソッド [711 ページ]

1.22.2.36 a_db_info 構造体

DBTools ライブラリを使用して DBInfo 情報を戻すために必要な情報を格納します。

構文

```
typedef struct a_db_info
```

メンバー

a_db_info のすべてのメンバー (継承されたメンバーも含みます) を以下に示します。

変数

変更子とタイプ	変数	説明
public unsigned short	version	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。
public MSG_CALLBACK	errorrtn	エラーメッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	msgrtn	情報メッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	statusrtn	ステータスメッセージコールバックルーチンのアドレス、または NULL。
public a_table_info *	totals	a_table_info 構造体へのポインタ。
public const char *	connectparms	データベース接続に必要なパラメータ。 次のような接続文字列のフォームを取ります。"UID=DBA;PWD=sql;DBF=demo.db". データベースサーバは、接続文字列の START パラメータによって起動されます。たとえば、"START=c:¥SQLAny17¥bin32¥dbsrv17.exe" などとなります。 次に START パラメータを含んだ完全な接続文字列の例を示します。"UID=DBA;PWD=sql;DBF=demo.db;START=c:¥SQLAny17¥bin32¥dbsrv17.exe"
public char *	dbnamebuffer	データベースファイル名バッファへのポインタ。
public char *	lognamebuffer	トランザクションログファイル名バッファへのポインタ。
public char *	mirrornamebuffer	ミラーファイル名バッファへのポインタ。

変更子とタイプ	変数	説明
public char *	charcollationspecbuffer	char 照合文字列バッファへのポインタ。
public char *	charencodingbuffer	char エンコード文字列バッファへのポインタ。
public char *	ncharcollationspecbuffer	nchar 照合文字列バッファへのポインタ。
public char *	ncharencodingbuffer	nchar エンコード文字列バッファへのポインタ。
public unsigned short	dbbufsize	dbnamebuffer のサイズ (たとえば、_MAX_PATH)。
public unsigned short	logbufsize	lognamebuffer のサイズ (たとえば、_MAX_PATH)。
public unsigned short	mirrorbufsize	mirrornamebuffer のサイズ (たとえば、_MAX_PATH)。
public unsigned short	charcollationspecbufsize	charcollationspecbuffer のサイズ (256+1 以上)。
public unsigned short	charencodingbufsize	charencodingbuffer のサイズ (50+1 以上)。
public unsigned short	ncharcollationspecbufsize	ncharcollationspecbuffer のサイズ (256+1 以上)。
public unsigned short	ncharencodingbufsize	ncharencodingbuffer のサイズ (50+1 以上)。
public a_sysinfo	sysinfo	インライン a_sysinfo 構造体。
public a_sql_uint32	file_size	データベースファイルのサイズ (ページ単位)。
public a_sql_uint32	free_pages	空きページ数。
public a_sql_uint32	bit_map_pages	データベース内のビットマップページ数。
public a_sql_uint32	other_pages	テーブルページ、インデックスページ、空きページ、ビットマップページのいずれでもないページの数。
public a_bit_field	quiet	TRUE に設定した場合、操作中に確認メッセージを出力しません。 dbinfo -q オプションによって TRUE に設定されます。
public a_bit_field	page_usage	TRUE に設定した場合はページ使用統計がレポートされ、FALSE に設定した場合はレポートされません。 dbinfo -u オプションによって TRUE に設定されます。
public a_bit_field	checksum	TRUE に設定した場合、グローバルチェックサム (すべてのデータベースページでのチェックサム) が有効になります。

変更子とタイプ	変数	説明
public a_bit_field	encrypted_tables	TRUE に設定した場合、暗号化テーブルがサポートされます。
public a_bit_field	pitr_alternate_timelines	TRUE に設定した場合、ポイントインタイムリカバリおよび代替タイムラインがサポートされます。このフィールドは DB_TOOLS_VERSION_IQSP09 で追加されましたが、構造体の旧バージョンから使用可能なビットを借りています。
public char *	current_timeline_guid	これらのフィールドは DB_TOOLS_VERSION_IQSP09 (16001) で追加されました 現在のタイムラインの GUID。不要な場合は NULL のままにします。
public size_t	current_timeline_guid_buffer_size	current_timeline_guid バッファのサイズ。
public char *	current_timeline_utc_creation_time	現在のタイムラインの UTC 作成時間を文字列として格納するためのバッファ。不要な場合は NULL のままにします。
public size_t	current_timeline_utc_creation_time_buffer_size	current_timeline_utc_creation_time_buffer のサイズ。
public char *	previous_timeline_guid	前のタイムラインの GUID。不要な場合は NULL のままにします。
public size_t	previous_timeline_guid_buffer_size	current_timeline_guid バッファのサイズ。
public char *	previous_timeline_utc_creation_time	前のタイムラインの UTC 作成時間を文字列として格納するためのバッファ。不要な場合は NULL のままにします。
public size_t	previous_timeline_utc_creation_time_buffer_size	previous_timeline_utc_creation_time_buffer のサイズ。
public a_sql_uint64	previous_timeline_branch_log_offset	現在のタイムラインが分岐した前のタイムライン内のログオフセット。
public char *	current_translog_guid	現在のトランザクションログの GUID。不要な場合は NULL のままにします。
public size_t	current_translog_guid_buffer_size	current_translog_guid バッファのサイズ。
public char *	previous_translog_guid	前のトランザクションログの GUID。不要な場合は NULL のままにします。
public size_t	previous_translog_guid_buffer_size	previous_translog_guid バッファのサイズ。

関連情報

a_table_info 構造体 [766 ページ]
a_sysinfo 構造体 [765 ページ]

DBInfo(a_db_info *) メソッド [713 ページ]
DBInfoDump(a_db_info *) メソッド [714 ページ]
DBInfoFree(a_db_info *) メソッド [715 ページ]

1.22.2.37 a_db_version_info 構造体

データベースの作成に使用された SQL Anywhere のバージョンに関する情報を保持します。

構文

```
typedef struct a_db_version_info
```

メンバー

a_db_version_info のすべてのメンバー (継承されたメンバーも含みます) を以下に示します。

変数

変更子とタイプ	変数	説明
public unsigned short	version	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。
public MSG_CALLBACK	errorrtn	エラーメッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	msgrtn	情報メッセージコールバックルーチンのアドレス、または NULL。
public const char *	filename	確認するデータベースファイルの名前。
public char	created_version	VERSION_UNKNOWN、VERSION_PRE_10 などのいずれかの値に設定します。 データベースファイルを作成したサーババージョンを示します。

関連情報

DBCreatedVersion(a_db_version_info *) メソッド [712 ページ]
バージョン列挙 [731 ページ]

1.22.2.38 a_dblic_info 構造体

ライセンス情報などを格納します。

構文

```
typedef struct a_dblic_info
```

メンバー

a_dblic_info のすべてのメンバー (継承されたメンバーも含みます) を以下に示します。

変数

変更子とタイプ	変数	説明
public unsigned short	version	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。
public MSG_CALLBACK	errorrtn	エラーメッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	msgrtn	情報メッセージコールバックルーチンのアドレス、または NULL。
public char *	exename	サーバ実行プログラムまたはライセンスファイルの名前。
public char *	username	ライセンスのユーザ名。
public char *	compname	ライセンスの会社名。
public a_sql_int32	nodecount	ライセンスノード数。
public a_sql_int32	conncount	ライセンスされた接続の最大数。 設定する場合、1000000L をデフォルトに使用します。
public a_license_type	type	値については lictype.h を参照してください。 LICENSE_TYPE_PERSEAT、LICENSE_TYPE_CONCURRENT、LICENSE_TYPE_PERCPU のうちいずれかです。
public char *	installkey	予約済み。NULL に設定します。 dblic -k オプションによって設定されます。
public a_bit_field	quiet	TRUE に設定した場合、操作中にメッセージを出力しません。 dblic -q オプションによって TRUE に設定されます。

変更子とタイプ	変数	説明
public a_bit_field	query_only	TRUE に設定した場合、ライセンス情報を表示するだけです。 FALSE に設定した場合、情報の変更を許可します。

備考

この情報は、ライセンス契約に従って使用してください。

関連情報

DBLicense(const a_dblic_info *) メソッド [716 ページ]

1.22.2.39 a_dbtools_info 構造体

DBTools ライブラリ呼び出しの初期化とファイナライズに使用する DBTools 情報コールバックです。

構文

```
typedef struct a_dbtools_info
```

メンバー

a_dbtools_info のすべてのメンバー (継承されたメンバーも含みます) を以下に示します。

変数

変更子とタイプ	変数	説明
public MSG_CALLBACK	errorrtn	エラーメッセージコールバックルーチンのアドレス、または NULL。

関連情報

DBToolsInit(const a_dbtools_info *) メソッド [719 ページ]

DBToolsFini(const a_dbtools_info *) メソッド [718 ページ]

1.22.2.40 a_log_file_info 構造体

実行中でないデータベースのログファイルとミラーログファイル情報の取得に使用します。

構文

```
typedef struct a_log_file_info
```

メンバー

a_log_file_info のすべてのメンバー (継承されたメンバーも含みます) を以下に示します。

変数

変更子とタイプ	変数	説明
public unsigned short	version	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。
public MSG_CALLBACK	errorrtn	エラーメッセージコールバックルーチンのアドレス、または NULL。
public const char *	dbname	データベースファイル名。
public const char *	encryption_key	データベースファイルの暗号化キー。
public char *	logname	トランザクションログファイル名のバッファ、または NULL。
public size_t	logname_size	トランザクションログファイル名のバッファのサイズ、または 0。
public char *	mirrorname	ミラーログファイル名のバッファ、または NULL。
public size_t	mirrorname_size	ミラーログファイル名のバッファのサイズ、または 0。
public void *	reserved	今後の使用のために予約されており、NULL に設定する必要があります。

関連情報

DBLogFileInfo(const a_log_file_info *) メソッド [716 ページ]

1.22.2.41 a_name 構造体

名前の変数リストを指定します。

構文

```
typedef struct a_name
```

メンバー

a_name のすべてのメンバー (継承されたメンバーも含みます) を以下に示します。

変数

変更子とタイプ	変数	説明
public struct a_name *	next	リスト内の次の名前へのポインタ、または NULL。
public char	name	名前を構成する 1 バイトまたはそれ以上のバイト。

1.22.2.42 a_remote_sql 構造体

DBTools ライブラリを使用する dbremote ユーティリティが必要とする情報を格納します。

構文

```
typedef struct a_remote_sql
```

メンバー

a_remote_sql のすべてのメンバー (継承されたメンバーも含みます) を以下に示します。

変数

変更子とタイプ	変数	説明
public unsigned short	version	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。
public MSG_CALLBACK	confirmrtn	確認要求コールバックルーチンのアドレス、または NULL。

変更子とタイプ	変数	説明
public MSG_CALLBACK	errorrtn	エラーメッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	msgrtn	情報メッセージコールバックルーチンのアドレス、または NULL。
public MSG_QUEUE_CALLBACK	msgqueuertn	DBRemoteSQL がスリープを必要とするときに呼び出す関数。 このパラメータには、スリープ時間をミリ秒単位で指定します。この関数は、dllapi.h に定義されているとおり、以下を返します。 • MSGQ_SLEEP_THROUGH は、要求されたミリ秒だけルーチンがスリープしたことを意味します。通常はこの値を返します。 • MSGQ_SHUTDOWN_REQUESTED は、できるだけ早急に同期を終了することを意味します。
public char *	connectparms	データベース接続に必要なパラメータ。 次のような接続文字列のフォームを取ります。"UID=DBA;PWD=sql;DBF=demo.db". データベースサーバは、接続文字列の START パラメータによって起動されます。たとえば、"START=c:¥SQLAny17¥bin32¥dbeng17.exe" などとなります。 次に START パラメータを含んだ完全な接続文字列の例を示します。"UID=DBA;PWD=sql;DBF=demo.db;START=c:¥SQLAny17¥bin32¥dbeng17.exe"
public char *	transaction_logs	オフライントランザクションログを持つディレクトリを指定します (DBRemoteSQL のみ)。 dbremote の transaction_logs_directory 引数に対応します。
public a_bit_field	receive	TRUE に設定した場合、メッセージを受信します。 receive と send の両方が FALSE の場合、両方とも TRUE であると見なされます。receive と send の両方を FALSE に設定することをお奨めします。dbremote の -r オプションに対応します。

変更子とタイプ	変数	説明
public a_bit_field	send	TRUE に設定した場合、メッセージを送信します。 receive と send の両方が FALSE の場合、両方とも TRUE であると見なされます。receive と send の両方を FALSE に設定することをお奨めします。dbremote の -s オプションに対応します。
public a_bit_field	verbose	設定した場合、追加情報が生成されます。 dbremote の -v オプションに対応します。
public a_bit_field	deleted	通常は TRUE に設定します。 設定しない場合、メッセージは適用後に削除されません。dbremote の -p オプションに対応します。
public a_bit_field	apply	通常は TRUE に設定します。 設定しない場合、メッセージはスキャンされますが、適用されません。dbremote の -a オプションに対応します。
public a_bit_field	batch	TRUE に設定した場合、メッセージを適用してログをスキャン後に強制的に終了します (少なくとも 1 ユーザが「常に」送信時刻を保持している場合と同様です)。 クリアした場合、実行モードはリモートユーザの送信時刻によって決まります。
public a_bit_field	more	TRUE に設定してください。
public a_bit_field	triggers	通常はクリア (FALSE) にしてください。 TRUE に設定した場合、トリガアクションがレプリケートされます。使用には十分注意してください。
public a_bit_field	debug	TRUE に設定した場合、デバッグ出力が含まれます。
public a_bit_field	rename_log	TRUE に設定した場合、ログの名前が変更され、再起動されます (DBRemoteSQL のみ)。
public a_bit_field	latest_backup	TRUE に設定した場合、バックアップされているログのみ処理されます。 ライブログからは操作を送信しません。dbremote の -u オプションに対応します。
public a_bit_field	scan_log	今後の使用のために予約されており、FALSE に設定する必要があります。

変更子とタイプ	変数	説明
public a_bit_field	link_debug	TRUE に設定した場合、リンクのデバッグが有効になります。
public a_bit_field	full_q_scan	今後の使用のために予約されており、FALSE に設定する必要があります。
public a_bit_field	no_user_interaction	TRUE に設定した場合、ユーザの操作を必要としません。
public a_bit_field	unused	今後の使用のために予約されており、FALSE に設定する必要があります。
public a_sql_uint32	max_length	メッセージの最大長をバイト単位で設定します。 この値は送信と受信に影響します。推奨値は 50000 です。dbremote の -l オプションに対応します。
public a_sql_uint32	memory	送信メッセージの作成時に使用するメモリバッファの最大サイズをバイト単位で設定します。 推奨値は 2 * 1024 * 1024 以上です。dbremote の -m オプションに対応します。
public a_sql_uint32	frequency	今後の使用のために予約されており、0 に設定する必要があります。
public a_sql_uint32	threads	メッセージの適用に使用するワーカスレッド数を設定します。 この値は 50 未満にしてください。dbremote の -w オプションに対応します。
public a_sql_uint32	operations	メッセージを適用するときに使用される値。 DBRemoteSQL でコミットされていない操作 (挿入、削除、更新) の数がこの値に達するまで、コミットは無視されます。dbremote の -g オプションに対応します。
public char *	queueparms	今後の使用のために予約されており、NULL に設定する必要があります。
public char *	locale	今後の使用のために予約されており、NULL に設定する必要があります。
public a_sql_uint32	receive_delay	新しいメッセージを受信する際のポーリングの待機間隔を秒単位で設定します。 推奨値は 60 です。dbremote の -rd オプションに対応します。

変更子とタイプ	変数	説明
public a_sql_uint32	patience_retry	受信メッセージが失われたと見なされるまでに DBRemoteSQL が待機する受信メッセージのポーリング回数を設定します。 たとえば、patience_retry が 3 の場合、DBRemoteSQL は見つからないメッセージの受信を 3 回まで試行します。その後、DBRemoteSQL は再送要求を送信します。推奨値は 1 です。dbremote の -rp オプションに対応します。
public MSG_CALLBACK	logrtn	指定されたメッセージをログファイルに出力する関数へのポインタ。 メッセージをユーザに表示する必要はありません。
public a_bit_field	use_hex_offsets	TRUE に設定した場合、ログオフセットが 16 進表記で表示されます。それ以外の場合、小数表記が使用されます。
public a_bit_field	use_relative_offsets	TRUE に設定した場合、ログオフセットが現在のログファイルの開始点への相対値として表示されます。 FALSE に設定した場合、ログオフセットが開始時刻から表示されます。
public a_bit_field	debug_page_offsets	今後の使用のために予約されており、FALSE に設定する必要があります。
public a_sql_uint32	debug_dump_size	今後の使用のために予約されており、0 に設定する必要があります。
public a_sql_uint32	send_delay	新しい操作のログファイルのスキャンを送信するまでの時間を秒単位で設定します。 0 に設定すると、DBRemoteSQL はユーザの送信時刻に基づいて適切な値を選択します。dbremote の -sd オプションに対応します。
public a_sql_uint32	resend_urgency	ユーザが再スキャンを必要としていることが判明してからログのフルスキャンを実行するまでの DBRemoteSQL の待機時間を秒単位で設定します。 0 に設定すると、DBRemoteSQL はユーザの送信時刻と収集したその他の情報に基づいて適切な値を選択します。dbremote の -ru オプションに対応します。
public char *	include_scan_range	今後の使用のために予約されており、NULL に設定する必要があります。

変更子とタイプ	変数	説明
public SET_WINDOW_TITLE_CALLBACK	set_window_title_rtn	ウィンドウのタイトルをリセットする関数へのポインタ (Windows の場合のみ)。 タイトルは "database_name (受信中、スキャン中、送信中) - default_window_title" の形式になります。
public char *	default_window_title	デフォルトのウィンドウタイトルを表す文字列へのポインタ。
public MSG_CALLBACK	progress_msg_rtn	進行状況メッセージを表示する関数へのポインタ。
public SET_PROGRESS_CALLBACK	progress_index_rtn	進行状況バーのステータスを更新する関数へのポインタ。 この関数には符号なしの整数を指定する 2 つの引数 index と max があります。最初の呼び出しでは、2 つの引数の値は最小値と最大値 (例: 0 と 100) になります。2 回目以降の呼び出しでは、最初の引数は現在のインデックス値 (例: 0 ～ 100) になり、2 番目の引数は常に 0 になります。
public char **	argv	解析されたコマンドラインへのポインタ (文字列へのポインタのベクトル)。 NULL 以外の場合、DBRemoteSQL はメッセージルーチンを呼び出して、-c、-cq、-ek で始まる引数を除いた各コマンドラインの引数を表示します。
public a_sql_uint32	log_size	オンライントランザクションログのサイズがこの値より大きくなると、DBRemoteSQL はオンライントランザクションログの名前を変更して再起動します。 dbremote の -x オプションに対応します。
public char *	encryption_key	暗号化キーへのポインタ。dbremote の -ek オプションに対応します。
public const char *	log_file_name	メッセージコールバックによって出力が書き込まれる DBRemoteSQL 出力ログの名前へのポインタ。 send が TRUE の場合、エラーログが統合データベースに送信されます (ただし、このポインタの値が NULL 以外の場合)。
public a_bit_field	truncate_remote_output_file	TRUE に設定した場合、リモート出力ファイルは追加される代わりにトランケートされます。 dbremote の -rt オプションに対応します。

変更子とタイプ	変数	説明
public char *	remote_output_file_name	DBRemoteSQL リモート出力ファイルの名前へのポインタ。 dbremote の -ro または -rt オプションに対応します。
public MSG_CALLBACK	warningrtn	指定された警告メッセージを表示する関数へのポインタ。 NULL の場合、errorrtn 関数が代わりに呼び出されます。
public char *	mirror_logs	オフラインミラートランザクションログを含むディレクトリの名前へのポインタ。 dbremote の -ml オプションに対応します。

備考

dbremote ユーティリティは、次のデフォルト値を設定してからコマンドラインオプションを処理します。

- version = DB_TOOLS_VERSION_NUMBER
- argv = (アプリケーションに渡される引数ベクトル)
- deleted = TRUE
- apply = TRUE
- more = TRUE
- link_debug = FALSE
- max_length = 50000
- memory = 2 * 1024 * 1024
- frequency = 1
- threads = 0
- receive_delay = 60
- send_delay = 0
- log_size = 0
- patience_retry = 1
- resend_urgency = 0
- log_file_name = (コマンドラインから設定)
- truncate_remote_output_file = FALSE
- remote_output_file_name = NULL
- no_user_interaction = TRUE (ユーザインタフェースが使用できない場合)
- errorrtn = (適切なルーチンのアドレス)
- msgrtn = (適切なルーチンのアドレス)
- confirmrtn = (適切なルーチンのアドレス)
- msgqueuertn = (適切なルーチンのアドレス)
- logrtn = (適切なルーチンのアドレス)

- warningrtn = (適切なルーチンのアドレス)
- set_window_title_rtn = (適切なルーチンのアドレス)
- progress_msg_rtn = (適切なルーチンのアドレス)
- progress_index_rtn = (適切なルーチンのアドレス)

関連情報

DBRemoteSQL(a_remote_sql *) メソッド [717 ページ]

1.22.2.43 a_sync_db 構造体

DBTools ライブラリを使用する dbmlsync ユーティリティが必要とする情報を格納します。

構文

```
typedef struct a_sync_db
```

メンバー

a_sync_db のすべてのメンバー (継承されたメンバーも含みます) を以下に示します。

変数

変更子とタイプ	変数	説明
public unsigned short	version	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。
public MSG_CALLBACK	errorrtn	エラーメッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	msgrtn	情報メッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	confirmrtn	確認要求コールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	logrtn	ログファイルのみにメッセージを書き込むロギングコールバックルーチンのアドレス、または NULL。
public SET_WINDOW_TITLE_CALLBACK	set_window_title_rtn	dbmlsync ウィンドウのタイトルを変更するために呼び出す関数 (Windows のみ)。

変更子とタイプ	変数	説明
public MSG_QUEUE_CALLBACK	msgqueuertn	DBMLSync がスリープするときに呼び出す関数。 このパラメータには、スリープ時間をミリ秒単位で指定します。この関数は、dllapi.h に定義されているとおり、以下を返します。 • MSGQ_SLEEP_THROUGH は、要求されたミリ秒だけルーチンがスリープしたことを意味します。通常はこの値を返します。 • MSGQ_SHUTDOWN_REQUESTED は、できるだけ早急に同期を終了することを意味します。 • MSGQ_SYNC_REQUESTED は、ルーチンが要求されたミリ秒数よりも少ない時間スリープしたこと、および、同期が現在実行中でない場合は、次の同期を即座に開始することを意味します。
public MSG_CALLBACK	progress_msg_rtn	進行状況バーの上部のステータスウィンドウのテキストを変更するために呼び出す関数。
public SET_PROGRESS_CALLBACK	progress_index_rtn	進行状況バーのステータスを更新するために呼び出す関数。
public USAGE_CALLBACK	usage_rtn	予約済み。NULL を使用します。
public STATUS_CALLBACK	status_rtn	予約済み。NULL を使用します。
public MSG_CALLBACK	warningrtn	警告メッセージを表示するために呼び出す関数。
public a_syncpub *	upload_defs	同期するパブリケーションまたはサブスクリプションのリンクリスト。
public a_syncpub *	last_upload_def	予約済み。NULL を使用します。
public const char *	offline_dir	トランザクションログディレクトリ。 dbmlsync コマンドラインで指定する最後の項目です。
public const char *	include_scan_range	予約済み。NULL を使用します。
public const char *	raw_file	予約済み。NULL を使用します。
public const char *	log_file_name	データベースサーバメッセージログファイル名。 dbmlsync -o または -ot オプションと同等です。
public const char *	apply_dnld_file	適用するダウンロードファイルの名前。 dbmlsync -ba オプションと同等です。オプションを指定しない場合は NULL です。

変更子とタイプ	変数	説明
public const char *	create_dnld_file	作成するダウンロードファイルの名前。 dbmlsync -bc オプションと同等です。オプションを指定しない場合は NULL です。
public const char *	dnld_file_extra	ダウンロードファイルに含める追加の文字列を指定します。 dbmlsync -be オプションと同等です。
public const char *	encrypted_stream_opts	予約済み。NULL を使用します。
public char **	argv	この実行における argv 配列。配列の最後の要素は NULL です。
public char **	ce_argv	予約済み。NULL を使用します。
public char **	ce_reproc_argv	予約済み。NULL を使用します。
public char *	connectparms	データベース接続に必要なパラメータ。 次のような接続文字列のフォームを取ります。"UID=DBA;PWD=sql;DBF=demo.db". データベースサーバは、接続文字列の START パラメータによって起動されます。たとえば、"START=c:¥SQLAny17¥bin32¥dbsrv17.exe" などとなります。 次に START パラメータを含んだ完全な接続文字列の例を示します。"UID=DBA;PWD=sql;DBF=demo.db;START=c:¥SQLAny17¥bin32¥dbsrv17.exe"
public char *	extended_options	「キーワード=値;...」の形式の拡張オプション。 dbmlsync -e オプションと同等です。
public char *	default_window_title	ウィンドウキャプションに表示するプログラム名 (DBMLSync など)。
public char *	mlpassword	Mobile Link パスワード。オプションを指定しない場合は NULL。 dbmlsync -mp オプションと同等です。
public char *	new_mlpassword	新しい Mobile Link パスワード。オプションを指定しない場合は NULL。 dbmlsync -mn オプションと同等です。
public char *	encryption_key	データベースファイルの暗号化キー。 dbmlsync -ek オプションと同等です。

変更子とタイプ	変数	説明
public char *	user_name	同期する Mobile Link ユーザ (旧式)。 dbmlsync -u オプションと同等です。
public char *	sync_params	ユーザ認証パラメータ。 dbmlsync -ap オプションと同等です。
public char *	preload_dlls	予約済み。NULL を使用します。
public char *	sync_profile	実行する同期プロファイル。 dbmlsync -sp オプションと同等です。
public char *	sync_opt	予約済み。NULL を使用します。
public a_sql_uint32	no_offline_logscan	TRUE に設定した場合、オフラインログスキャンを無効にします (-x オプションと一緒に使用することはできません)。 dbmlsync -do オプションと同等です。
public a_sql_uint32	debug_dump_size	予約済み。0 を使用します。
public a_sql_uint32	dl_insert_width	予約済み。0 を使用します。
public a_sql_uint32	log_size	トランザクションログの名前を変更して再起動するときのログファイルのサイズ (バイト単位)。 サイズを指定しない場合は 0 を指定します。 dbmlsync -x オプションと同等です。
public a_sql_uint32	hovering_frequency	ログスキャンのポーリング期間を秒単位で設定します。 通常は 60 です。dbmlsync -pp オプションと同等です。
public a_sql_uint32	est_upld_row_cnt	推定されるアップロードのロー数を設定します (最適化の場合)。 dbmlsync -urc オプションと同等です。
public a_sql_uint32	dnld_read_size	ダウンロードの読み込みサイズを設定します。 dbmlsync -drs オプションと同等です。
public a_sql_uint32	dnld_fail_len	予約済み。0 を使用します。
public a_sql_uint32	upld_fail_len	予約済み。0 を使用します。
public a_sql_uint32	server_port	サーバモードでの実行時の通信ポートを設定します。 dbmlsync -po オプションと同等です。
public a_sql_uint32	dlg_info_msg	予約済み。0 を使用します。

変更子とタイプ	変数	説明
public a_sql_uint32	min_cache	キャッシュの最小サイズ。 dbmlsync -cl オプションと同等です。
public char	min_cache_suffix	最小キャッシュサイズのサフィックス ('B' はバイト、'P' はパーセンテージ。指定しない場合は 0)。
public a_sql_uint32	max_cache	キャッシュの最大サイズ。 dbmlsync -cm オプションと同等です。
public char	max_cache_suffix	最大キャッシュサイズのサフィックス ('B' はバイト、'P' はパーセンテージ。指定しない場合は 0)。
public a_sql_uint32	init_cache	キャッシュの初期サイズ。 dbmlsync -ci オプションと同等です。
public char	init_cache_suffix	初期キャッシュサイズのサフィックス ('B' はバイト、'P' はパーセンテージ。指定しない場合は 0)。
public a_sql_int32	background_retry	中断されたバックグラウンド同期のリトライ回数。 dbmlsync -bkr オプションと同等です。
public a_bit_field	ping	TRUE に設定した場合、Mobile Link サーバを ping します。 dbmlsync -pi オプションと同等です。
public a_bit_field	dnld_gen_num	TRUE に設定した場合、ダウンロードファイルの適用時に世代番号を更新します。 dbmlsync -bg オプションと同等です。
public a_bit_field	background_sync	TRUE に設定した場合、バックグラウンド同期を実行します。 dbmlsync -bk オプションと同等です。
public a_bit_field	kill_other_connections	TRUE に設定した場合、同期中のテーブルをロックしている接続を削除します。 dbmlsync -d オプションと同等です。
public a_bit_field	continue_download	TRUE に設定した場合、以前に失敗したダウンロードを続行します。 dbmlsync -dc オプションと同等です。
public a_bit_field	download_only	TRUE に設定した場合、ダウンロード専用同期を実行します。 dbmlsync -ds オプションと同等です。

変更子とタイプ	変数	説明
public a_bit_field	ignore_hook_errors	TRUE に設定した場合、フック関数で発生するエラーを無視します。 dbmlsync -eh オプションと同等です。
public a_bit_field	ignore_scheduling	TRUE に設定した場合、スケジュールを無視します。 dbmlsync -is オプションと同等です。
public a_bit_field	autoclose	TRUE に設定した場合、完了時にウィンドウを閉じます。 dbmlsync -qc オプションと同等です。
public a_bit_field	changing_pwd	新しい Mobile Link パスワードを設定する場合、TRUE に設定します。 new_mlpassword フィールドを参照してください。dbmlsync -mn オプションと同等です。
public a_bit_field	ignore_hovering	TRUE に設定した場合、ログスキャンのポーリングを無効にします。 dbmlsync -p オプションと同等です。
public a_bit_field	persist_connection	TRUE に設定した場合、同期間で Mobile Link 接続を持続します。 FALSE に設定した場合、同期間で Mobile Link 接続を閉じます。dbmlsync -pc{+|-} オプションと同等です。
public a_bit_field	allow_schema_change	TRUE に設定した場合、同期間でのスキーマの変更の有無をチェックします。 dbmlsync -sc オプションと同等です。
public a_bit_field	retry_remote_behind	TRUE に設定した場合、リモートオフセットが統合オフセットよりも小さいときに、進行状況が一致しなければ、リモートオフセットを使用してアップロードを再送します。 リモートオフセットが統合オフセット未満の場合。dbmlsync -r または -rb オプションと同等です。
public a_bit_field	server_mode	TRUE に設定した場合、サーバモードで実行します。 dbmlsync -sm オプションと同等です。
public a_bit_field	trans_upload	TRUE に設定した場合、各データベーストランザクションを別々にアップロードします。 dbmlsync -tu オプションと同等です。

変更子とタイプ	変数	説明
public a_bit_field	retry_remote_ahead	TRUE に設定した場合、リモートオフセットが統合オフセットよりも大きいときに、進行状況が一致しなければ、リモートオフセットを使用してアップロードを再送します。 dbmlsync -ra オプションと同等です。
public a_bit_field	upload_only	TRUE に設定した場合、アップロード専用同期を実行します。 dbmlsync -uo オプションと同等です。
public a_bit_field	verbose_minimum	TRUE に設定した場合、冗長性を最小に設定します。 dbmlsync -v オプションと同等です。
public a_bit_field	hide_conn_str	FALSE に設定した場合、接続文字列を表示します。TRUE に設定した場合、接続文字列を表示しません。 dbmlsync -vc オプションと同等です。
public a_bit_field	hide_ml_pwd	FALSE に設定した場合、Mobile Link パスワードを表示します。TRUE に設定した場合、Mobile Link パスワードを表示しません。 dbmlsync -vp オプションと同等です。
public a_bit_field	verbose_row_cnts	TRUE に設定した場合、アップロードまたはダウンロードローカウントを表示します。 dbmlsync -vn オプションと同等です。
public a_bit_field	verbose_option_info	TRUE に設定した場合、コマンドラインと拡張オプションを表示します。 dbmlsync -vo オプションと同等です。
public a_bit_field	verbose_row_data	TRUE に設定した場合、アップロードまたはダウンロードローの値を表示します。 dbmlsync -vr オプションと同等です。
public a_bit_field	verbose_hook	TRUE に設定した場合、フックスクリプト情報を表示します。 dbmlsync -vs オプションと同等です。
public a_bit_field	verbose_upload	TRUE に設定した場合、アップロードストリーム情報を表示します。 dbmlsync -vu オプションと同等です。
public a_bit_field	verbose_msgid	TRUE に設定した場合、メッセージ ID を表示します。 dbmlsync -vi オプションと同等です。

変更子とタイプ	変数	説明
public a_bit_field	rename_log	TRUE に設定した場合、トランザクションログの名前を変更して再起動します。 log_size フィールドを参照してください。 dbmlsync -x オプションと同等です。
public a_bit_field	keep_partial_download	失敗したダウンロードの再開を許可する場合は TRUE に設定します。dbmlsync -kpd オプションと同等です。
public a_bit_field	allow_outside_connect	予約済み。0 を使用します。
public a_bit_field	cache_verbosity	予約済み。0 を使用します。
public a_bit_field	connectparms_allocated	予約済み。0 を使用します。
public a_bit_field	debug	予約済み。0 を使用します。
public a_bit_field	debug_dump_char	予約済み。0 を使用します。
public a_bit_field	debug_dump_hex	予約済み。0 を使用します。
public a_bit_field	debug_page_offsets	予約済み。0 を使用します。
public a_bit_field	dl_use_put	予約済み。0 を使用します。
public a_bit_field	entered_dialog	予約済み。0 を使用します。
public a_bit_field	ignore_debug_interrupt	予約済み。0 を使用します。
public a_bit_field	lite_blob_handling	予約済み。0 を使用します。
public a_bit_field	no_schema_cache	予約済み。0 を使用します。
public a_bit_field	no_stream_compress	予約済み。0 を使用します。
public a_bit_field	output_to_file	予約済み。0 を使用します。
public a_bit_field	output_to_mobile_link	予約済み。1 を使用します。
public a_bit_field	prompt_again	予約済み。0 を使用します。
public a_bit_field	prompt_for_encrypt_key	予約済み。0 を使用します。
public a_bit_field	strictly_ignore_trigger_ops	予約済み。0 を使用します。
public a_bit_field	use_fixed_cache	予約済み。0 を使用します。
public a_bit_field	use_hex_offsets	予約済み。0 を使用します。
public a_bit_field	use_relative_offsets	予約済み。0 を使用します。
public a_bit_field	used_dialog_allocation	予約済み。0 を使用します。
public a_bit_field	verbose	予約済み。0 を使用します。
public a_bit_field	verbose_download	予約済み。0 を使用します。
public a_bit_field	verbose_download_data	予約済み。0 を使用します。
public a_bit_field	verbose_protocol	予約済み。0 を使用します。
public a_bit_field	verbose_server	予約済み。0 を使用します。

変更子とタイプ	変数	説明
public a_bit_field	verbose_upload_data	予約済み。0 を使用します。
public a_bit_field	protocol_add_cli_bit_to_cli_max	予約済み。0 を使用します。
public a_bit_field	protocol_add_cli_bit_to_cli_both	予約済み。0 を使用します。
public a_bit_field	protocol_add_serv_bit_to_cli_max	予約済み。0 を使用します。
public a_bit_field	protocol_add_serv_bit_to_cli_both	予約済み。0 を使用します。
public a_bit_field	protocol_add_serv_bit_to_serv_max	予約済み。0 を使用します。
public a_bit_field	protocol_add_serv_bit_to_serv_both	予約済み。0 を使用します。
public a_bit_field	strictly_free_memory	予約済み。0 を使用します。
public a_bit_field	reserved	予約済み。0 を使用します。

備考

一部のメンバーは、dbmlsync コマンドラインユーティリティからアクセスできる機能に対応しています。未使用のメンバーには、データ型に応じて値 0、FALSE、または NULL を割り当ててください。

関連情報

DBSynchronizeLog(const a_sync_db *) メソッド [718 ページ]

1.22.2.44 a_syncpub 構造体

dbmlsync ユーティリティが必要とする情報を格納します。

構文

```
typedef struct a_syncpub
```

メンバー

a_syncpub のすべてのメンバー (継承されたメンバーも含みます) を以下に示します。

変数

変更子とタイプ	変数	説明
public struct a_syncpub *	next	リスト内の次のノードへのポインタ。最後のノードの場合は NULL。
public char *	pub_name	カンマで区切られたパブリケーション名 (旧式)。 dbmlsync -n オプションの後に続くものと同じ文字列です。pub_name と subscription のいずれか 1 つだけを NULL 以外の値にできます。
public char *	subscription	カンマで区切られたサブスクリプション名。 dbmlsync -s オプションの後に続くものと同じ文字列です。pub_name と subscription のいずれか 1 つだけを NULL 以外の値にできます。
public char *	ext_opt	「キーワード=値;...」の形式の拡張オプション。 dbmlsync -eu オプションの後に続くものと同じオプションです。

関連情報

a_sync_db 構造体 [756 ページ]
DBSynchronizeLog(const a_sync_db *) メソッド [718 ページ]

1.22.2.45 a_sysinfo 構造体

DBTools ライブラリを使用する dbinfo と dbunload ユーティリティが必要とする情報を格納します。

構文

```
typedef struct a_sysinfo
```

メンバー

a_sysinfo のすべてのメンバー (継承されたメンバーも含みます) を以下に示します。

変数

変更子とタイプ	変数	説明
public unsigned short	page_size	データベースのページサイズ。
public char	default_collation	データベースの照合順。
public a_bit_field	valid_data	1 の場合、他のビットフィールドが有効であることを示します。
public a_bit_field	blank_padding	1 の場合、このデータベースではブランクの埋め込みをします。0 の場合はしません。
public a_bit_field	case_sensitivity	1 の場合、データベースは大文字と小文字を区別します。0 の場合はしません。
public a_bit_field	encryption	1 の場合、データベースは暗号化されます。0 の場合はされません。

関連情報

a_db_info 構造体 [742 ページ]
DBInfo(a_db_info *) メソッド [713 ページ]
DBInfoDump(a_db_info *) メソッド [714 ページ]
DBInfoFree(a_db_info *) メソッド [715 ページ]

1.22.2.46 a_table_info 構造体

a_db_info 構造体の一部として必要なテーブルに関する情報を格納します。

構文

```
typedef struct a_table_info
```

メンバー

a_table_info のすべてのメンバー (継承されたメンバーも含みます) を以下に示します。

変数

変更子とタイプ	変数	説明
public struct a_table_info *	next	リスト内の次のテーブル。
public char *	table_name	テーブルの名前。
public a_sql_uint32	table_id	このテーブルの ID 番号。

変更子とタイプ	変数	説明
public a_sql_uint32	table_pages	テーブルページの数。
public a_sql_uint32	index_pages	インデックスページの数。
public a_sql_uint32	table_used	テーブルページに使用されているバイト数。
public a_sql_uint32	index_used	インデックスページに使用されているバイト数。
public a_sql_uint32	table_used_pct	テーブル領域の使用率を示すパーセンテージ。
public a_sql_uint32	index_used_pct	インデックス領域の使用率を示すパーセンテージ。

関連情報

a_db_info 構造体 [742 ページ]
DBInfo(a_db_info *) メソッド [713 ページ]
DBInfoDump(a_db_info *) メソッド [714 ページ]
DBInfoFree(a_db_info *) メソッド [715 ページ]

1.22.2.47 a_translate_log 構造体

DBTools ライブラリを使用してトランザクションログを変換するために必要な情報を格納します。

構文

```
typedef struct a_translate_log
```

メンバー

a_translate_log のすべてのメンバー (継承されたメンバーも含みます) を以下に示します。

変数

変更子とタイプ	変数	説明
public unsigned short	version	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。
public MSG_CALLBACK	errorrtn	エラーメッセージコールバックルーチンのアドレス、または NULL。

変更子とタイプ	変数	説明
public MSG_CALLBACK	msgrtn	情報メッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	confirmrtn	確認要求コールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	statusrtn	ステータスメッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	logrtn	ログファイルのみにメッセージを書き込むロギングコールバックルーチンのアドレス、または NULL。
public const char *	connectparms	データベース接続に必要なパラメータ。 次のような接続文字列のフォームを取ります。"UID=DBA;PWD=sql;DBF=demo.db". データベースサーバは、接続文字列の START パラメータによって起動されます。たとえば、"START=c:¥SQLAny17¥bin32¥dbsrv17.exe" などとなります。 次に START パラメータを含んだ完全な接続文字列の例を示します。"UID=DBA;PWD=sql;DBF=demo.db;START=c:¥SQLAny17¥bin32¥dbsrv17.exe"
public const char *	logname	トランザクションログファイルの名前。NULL の場合、ログは作成されません。
public const char *	sqlname	SQL 出力ファイルの名前。 NULL に設定した場合、トランザクションログファイルの名前に基づいた名前になります。dbtran -n オプションと同等です。
public const char *	encryption_key	データベースファイルの暗号化キー。dbtran -ek オプションと同等です。
public const char *	logs_dir	トランザクションログディレクトリ。 dbtran -m オプションと同等です。sqlname ポインタを設定する必要があり、connectparms を NULL にする必要があります。
public const char *	include_source_sets	予約済み。NULL を使用します。
public const char *	include_destination_sets	予約済み。NULL を使用します。
public const char *	include_scan_range	予約済み。NULL を使用します。
public const char *	repserver_users	予約済み。NULL を使用します。
public const char *	include_tables	予約済み。NULL を使用します。

変更子とタイプ	変数	説明
public const char *	include_publications	予約済み。NULL を使用します。
public const char *	queueparms	予約済み。NULL を使用します。
public const char *	match_pos	予約済み。NULL を使用します。
public const char *	display_timelines_json	-htj オプションで使用します。
public const char *	display_timelines_text	-htt オプションで使用します。
public const char *	force_timeline_guid	-hft オプションで使用します。
public p_name	userlist	リンクされたユーザ名のリスト。 dbtran -u user1,... または -x user1,... と同等です。リストされているユーザのトランザクションを選択または省略します。
public a_sql_uint32	since_time	指定された時刻より前の最新のチェックポイントから出力します。 西暦 1 年 1 月 1 日からの分数です。dbtran -j オプションと同等です。
public a_sql_uint32	debug_dump_size	予約済み。0 を使用します。
public a_sql_uint32	recovery_ops	予約済み。0 を使用します。
public a_sql_uint32	recovery_bytes	予約済み。0 を使用します。
public char	userlisttype	ユーザのリストを含めるか除外する場合を除き、DBTRAN_INCLUDE_ALL に設定します。 -u の場合は DBTRAN_INCLUDE_SOME に設定し、-x の場合は DBTRAN_EXCLUDE_SOME に設定します。
public a_bit_field	quiet	TRUE に設定した場合、操作中にメッセージを出力しません。 dbtran -q オプションによって TRUE に設定されます。
public a_bit_field	remove_rollback	出力結果にロールバックトランザクションを含める場合は FALSE に設定します。 dbtran -a オプションによって FALSE に設定されます。
public a_bit_field	ansi_sql	TRUE に設定した場合、ANSI 標準の SQL トランザクションを生成します。 dbtran -s オプションによって TRUE に設定されます。

変更子とタイプ	変数	説明
public a_bit_field	since_checkpoint	最新のチェックポイントから出力する場合は TRUE に設定します。 dbtran -f オプションによって TRUE に設定されます。
public a_bit_field	replace	TRUE に設定した場合、確認をせずに SQL ファイルを置換します。 dbtran -y オプションによって TRUE に設定されます。
public a_bit_field	include_trigger_trans	TRUE に設定した場合、トリガ生成トランザクションを含めます。 dbtran -t、-g、および -sr オプションによって TRUE に設定されます。
public a_bit_field	comment_trigger_trans	TRUE に設定した場合、トリガ生成トランザクションをコメントとして含めます。 dbtran -z オプションによって TRUE に設定されます。
public a_bit_field	leave_output_on_error	TRUE に設定した場合、ログエラーが検出されると、生成された .SQL ファイルを残します。 dbtran -k オプションによって TRUE に設定されます。
public a_bit_field	debug	予約済み。FALSE に設定します。
public a_bit_field	debug_sql_remote	予約済み。FALSE に設定します。
public a_bit_field	debug_dump_hex	予約済み。FALSE に設定します。
public a_bit_field	debug_dump_char	予約済み。FALSE に設定します。
public a_bit_field	debug_page_offsets	予約済み。FALSE に設定します。
public a_bit_field	omit_comments	予約済み。FALSE に設定します。
public a_bit_field	use_hex_offsets	予約済み。FALSE に設定します。
public a_bit_field	use_relative_offsets	予約済み。FALSE に設定します。
public a_bit_field	include_audit	予約済み。FALSE に設定します。
public a_bit_field	chronological_order	予約済み。FALSE に設定します。
public a_bit_field	force_recovery	予約済み。FALSE に設定します。
public a_bit_field	include_subsets	予約済み。FALSE に設定します。
public a_bit_field	force_chaining	予約済み。FALSE に設定します。
public a_bit_field	generate_reciprocals	予約済み。FALSE に設定します。
public a_bit_field	match_mode	予約済み。FALSE に設定します。

変更子とタイプ	変数	説明
public a_bit_field	show_undo	予約済み。FALSE に設定します。
public a_bit_field	extra_audit	予約済み。FALSE に設定します。

関連情報

a_name 構造体 [749 ページ]
ユーザリスト列挙 [729 ページ]
DBTranslateLog(const a_translate_log *) メソッド [720 ページ]

1.22.2.48 a_truncate_log 構造体

DBTools ライブラリを使用してトランザクションログをトランケートするために必要な情報を格納します。

構文

```
typedef struct a_truncate_log
```

メンバー

a_truncate_log のすべてのメンバー (継承されたメンバーも含みます) を以下に示します。

変数

変更子とタイプ	変数	説明
public unsigned short	version	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。
public MSG_CALLBACK	errorrtn	エラーメッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	msgrtn	情報メッセージコールバックルーチンのアドレス、または NULL。

変更子とタイプ	変数	説明
public const char *	connectparms	データベース接続に必要なパラメータ。 次のような接続文字列のフォームを取ります。"UID=DBA;PWD=sql;DBF=demo.db". データベースサーバは、接続文字列の START パラメータによって起動されます。たとえば、"START=c:¥SQLAny17¥bin32¥dbsrv17.exe" などとなります。 次に START パラメータを含んだ完全な接続文字列の例を示します。"UID=DBA;PWD=sql;DBF=demo.db;START=c:¥SQLAny17¥bin32¥dbsrv17.exe"
public char	truncate_interrupted	0 以外の場合にトランケートが中断されたことを示します。
public a_bit_field	quiet	TRUE に設定した場合、操作中にメッセージを出力しません。 dbbackup -q オプションによって TRUE に設定されます。
public a_bit_field	server_backup	TRUE に設定した場合、BACKUP DATABASE を使用したサーバでのバックアップを示します。 dbbackup -x オプションが指定されたときに、dbbackup -s オプションによって TRUE に設定されます。

関連情報

DBTruncateLog(const a_truncate_log *) メソッド [721 ページ]

1.22.2.49 a_validate_db 構造体

DBTools ライブラリを使用してデータベースを検証するために必要な情報を格納します。

構文

```
typedef struct a_validate_db
```

メンバー

a_validate_db のすべてのメンバー (継承されたメンバーも含みます) を以下に示します。

変数

変更子とタイプ	変数	説明
public unsigned short	version	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。
public MSG_CALLBACK	errorrtn	エラーメッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	msgrtn	情報メッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	statusrtn	ステータスメッセージコールバックルーチンのアドレス、または NULL。
public const char *	connectparms	データベース接続に必要なパラメータ。 次のような接続文字列のフォームを取ります。"UID=DBA;PWD=sql;DBF=demo.db". データベースサーバは、接続文字列の START パラメータによって起動されます。たとえば、"START=c:¥SQLAny17¥bin32¥dbsrv17.exe" などとなります。 次に START パラメータを含んだ完全な接続文字列の例を示します。"UID=DBA;PWD=sql;DBF=demo.db;START=c:¥SQLAny17¥bin32¥dbsrv17.exe"
public p_name	tables	テーブル名またはインデックス名 (index フィールドが TRUE に設定されている場合) のリンクリストへのポインタ。 dbvalid の object-name-list 引数によって設定されます。
public char	type	実行する検証のタイプ。 VALIDATE_NORMAL、VALIDATE_EXPRESS、VALIDATE_CHECKSUM などのいずれか。検証列挙を参照してください。
public a_bit_field	quiet	TRUE に設定した場合、操作中にメッセージを出力しません。 dbvalid -q オプションによって TRUE に設定されます。

変更子とタイプ	変数	説明
public a_bit_field	index	TRUE に設定した場合、インデックスを検証します。 tables フィールドは、インデックスのリストを指し示します。dbvalid -i オプションによって TRUE に設定されます。dbvalid -t オプションによって FALSE に設定されます。
public int	inmem_mode	StartLine 接続パラメータの変更を制御し、自動起動データベースサーバのインメモリモードを選択します。 次のいずれかの IM_ 列挙値に設定します。この機能は、DB_TOOLS_VERSION_17_0_0 以降のバージョンに装備されてます。
public char	live_type	実行するライブ検証のタイプ。 VALIDATE_NO_LIVE_OPTION、VALIDATE_WITH_DATA_LOCK、VALIDATE_WITH_SNAPSHOT のいずれか。ライブ検証列挙を参照してください。この機能は、DB_TOOLS_VERSION_17_0_0 以降のバージョンに装備されてます。

関連情報

a_name 構造体 [749 ページ]
DBValidate(const a_validate_db *) メソッド [723 ページ]
検証列挙 [730 ページ]

1.22.2.50 an_erase_db 構造体

DBTools ライブラリを使用してデータベースを消去するために必要な情報を格納します。

構文

```
typedef struct an_erase_db
```

メンバー

an_erase_db のすべてのメンバー (継承されたメンバーも含みます) を以下に示します。

変数

変更子とタイプ	変数	説明
public unsigned short	version	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。
public MSG_CALLBACK	confirmrtn	確認要求コールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	errorrtn	エラーメッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	msgrtn	情報メッセージコールバックルーチンのアドレス、または NULL。
public const char *	dbname	データベースファイル名。
public const char *	encryption_key	データベースファイルの暗号化キー。 dberase -ek または -ep オプションと同等です。
public a_bit_field	quiet	操作中にメッセージを出力します (0) またはしません (1)。 dberase -q オプションによって TRUE に設定されます。
public a_bit_field	erase	消去するときに確認します (0) またはしません (1)。 dberase -y オプションによって TRUE に設定されます。

関連情報

DBErase(const an_erase_db *) メソッド [713 ページ]

1.22.2.51 an_unload_db 構造体

DBTools ライブラリを使用してデータベースをアンロードするため、または SQL Remote でリモートデータベースを抽出するために必要な情報を格納します。

構文

```
typedef struct an_unload_db
```

メンバー

an_unload_db のすべてのメンバー (継承されたメンバーも含みます) を以下に示します。

変数

変更子とタイプ	変数	説明
public unsigned short	version	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。
public MSG_CALLBACK	errorrtn	エラーメッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	msgrtn	情報メッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	statusrtn	ステータスメッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	confirmrtn	確認要求コールバックルーチンのアドレス、または NULL。
public const char *	connectparms	データベース接続に必要なパラメータ。 次のような接続文字列のフォームを取ります。"UID=DBA;PWD=sql;DBF=demo.db". データベースサーバは、接続文字列の START パラメータによって起動されます。例:"START=c:¥SQLAny17¥bin32¥dbsrv17.exe" 次に START パラメータを含んだ完全な接続文字列の例を示します。"UID=DBA;PWD=sql;DBF=demo.db;START=c:¥SQLAny17¥bin32¥dbsrv17.exe"
public const char *	temp_dir	データファイルのアンロード用ディレクトリ。
public const char *	reload_filename	再ロード SQL スクリプトファイルに使用する名前 (reload.sql など)。 dbunload -r オプションによって設定されます。
public const char *	ms_filename	予約済み。NULL を使用します。
public const char *	remote_dir	temp_dir に似ていますが、サーバ側の内部アンロード用です。
public const char *	subscriber_username	dbxtract で使用されるサブスクライバ名。それ以外の場合は NULL。
public const char *	site_name	dbxtract で使用されるサイト名。それ以外の場合は NULL。
public const char *	template_name	dbxtract で使用されるテンプレート名。それ以外の場合は NULL。

変更子とタイプ	変数	説明
public const char *	encryption_key	データベースファイルの暗号化キー。 dbunload/dbxtract -ek または -ep オプションによって設定されます。
public const char *	encryption_algorithm	暗号化アルゴリズム。"simple"、"aes"、"aes256"、"aes_fips"、"aes256_fips"、または NULL (なしの場合)。 dbunload/dbxtract -ea オプションによって設定されます。
public const char *	locale	予約済み。NULL を使用します。
public const char *	startline	予約済み。NULL を使用します。
public const char *	startline_old	予約済み。NULL を使用します。
public char *	reload_connectparms	接続パラメータ (データベースを再ロードするためのユーザ ID、パスワード、データベースなど)。 dbunload/dbxtract -ac オプションによって設定されます。
public char *	reload_db_filename	作成して再ロードする新しいデータベースファイルの名前。 dbunload/dbxtract -an オプションによって設定されます。
public char *	reload_db_logname	新しいデータベースのトランザクションログのファイル名、または NULL。 dbxtract -al オプションによって設定されます。
public p_name	table_list	選択的なテーブルリスト。 dbunload -e および -t オプションによって設定されます。
public a_sysinfo	sysinfo	予約済み。NULL を使用します。
public long	notemp_size	予約済み。0 を使用します。
public int	ms_reserve	予約済み。0 を使用します。
public int	ms_size	予約済み。0 を使用します。
public unsigned short	isolation_level	操作の独立性レベル。 dbxtract -I オプションによって設定されます。
public unsigned short	reload_page_size	再ロードデータベースページサイズ。 dbunload -ap オプションによって設定されます。

変更子とタイプ	変数	説明
public char	unload_type	アンロード列挙を設定します (UNLOAD_ALL など)。 dbunload/dbxtract -d、-k、-n オプションによって設定されます。
public char	verbose	冗長列挙 (VB_QUIET、VB_NORMAL、VB_VERBOSE) を参照してください。
public char	escape_char	エスケープ文字 (通常は "¥")。 escape_char_present が TRUE の場合に使用します。dbunload/dbxtract -p オプションによって TRUE に設定されます。
public char	unload_interrupted	予約済み。0 に設定します。
public a_bit_field	unordered	順序付けなしのデータの場合、TRUE に設定します。 インデックスはデータのアンロードには使用されません。dbunload/dbxtract -u オプションによって設定されます。
public a_bit_field	no_confirm	TRUE に設定した場合、確認をせずに既存の SQL スクリプトファイルを置換します。 dbunload/dbxtract -y オプションによって設定されます。
public a_bit_field	use_internal_unload	TRUE に設定した場合、内部アンロードを実行します。 dbunload/dbxtract -i? オプションによって TRUE に設定されます。dbunload/dbxtract -x? オプションによって FALSE に設定されます。
public a_bit_field	refresh_mat_view	TRUE に設定した場合、テキストインデックスと有効なマテリアライズドビューを再表示する文を生成します。 dbunload/dbxtract -g オプションによって TRUE に設定されます。
public a_bit_field	table_list_provided	TRUE に設定した場合、テーブルのリストがすでに提供されていることを示します。 table_list フィールドを参照してください。dbunload -e、-t、または -tl オプションによって TRUE に設定されます。

変更子とタイプ	変数	説明
public a_bit_field	exclude_tables	FALSE に設定した場合、追加するテーブルがリストに含まれていることを示します。 TRUE に設定した場合、除外するテーブルがリストに含まれていることを示します。dbunload -e オプションによって TRUE に設定されます。
public a_bit_field	preserve_ids	TRUE に設定した場合、SQL Remote データベース用にユーザ ID を保持します。 これは通常の設定です。dbunload -m オプションによって FALSE に設定されます。
public a_bit_field	replace_db	TRUE に設定した場合、データベースを置換します。 dbunload -ar オプションによって TRUE に設定されます。
public a_bit_field	escape_char_present	TRUE に設定した場合、escape_char のエスケープ文字が定義されていることを示します。 dbunload/dbxtract -p オプションによって TRUE に設定されます。
public a_bit_field	use_internal_reload	TRUE に設定した場合、内部再ロードを実行します。 これは通常の設定です。dbunload/dbxtract -ii および -xi オプションによって TRUE に設定されます。dbunload/dbxtract -ix および -xx オプションによって FALSE に設定されます。
public a_bit_field	recompute	TRUE に設定した場合、計算カラムを再実行します。 dbunload -dc オプションによって TRUE に設定されます。
public a_bit_field	make_auxiliary	TRUE に設定した場合、(診断トレーシングで使用する) 補助カタログを作成します。 dbunload -k オプションによって TRUE に設定されます。
public a_bit_field	profiling_uses_single_dbspace	TRUE に設定した場合、(診断トレーシングで使用する) 1 つの DB 領域ファイルにまとめます。 dbunload -kd オプションによって TRUE に設定されます。

変更子とタイプ	変数	説明
public a_bit_field	encrypted_tables	TRUE に設定した場合、新しいデータベースで暗号化されたテーブルを有効にします (-an または -ar とともに使用)。 dbunload/dbxtract -et オプションによって TRUE に設定されます。
public a_bit_field	remove_encrypted_tables	TRUE に設定した場合、暗号化されたテーブルの暗号化を解除します。 dbunload/dbxtract -er オプションによって TRUE に設定されます。
public a_bit_field	extract	リモートデータベース抽出を実行する場合は TRUE に設定します。 dbunload によって FALSE に設定されます。dbxtract によって TRUE に設定されます。
public a_bit_field	start_subscriptions	TRUE に設定した場合、サブスクリプションを開始します。 これは dbxtract のデフォルトです。dbxtract -b オプションによって FALSE に設定されます。
public a_bit_field	exclude_foreign_keys	TRUE に設定した場合、外部キーを除外します。 dbxtract -xf オプションによって TRUE に設定されます。
public a_bit_field	exclude_procedures	TRUE に設定した場合、ストアドプロシージャを除外します。 dbxtract -xp オプションによって TRUE に設定されます。
public a_bit_field	exclude_triggers	TRUE に設定した場合、トリガを除外します。 dbxtract -xt オプションによって TRUE に設定されます。
public a_bit_field	exclude_views	TRUE に設定した場合、ビューを除外します。 dbxtract -xv オプションによって TRUE に設定されます。
public a_bit_field	isolation_set	TRUE に設定した場合、すべての抽出操作に対して isolation_level が設定されていることを示します。 dbxtract -I オプションによって TRUE に設定されます。

変更子とタイプ	変数	説明
public a_bit_field	include_where_subscribe	TRUE に設定した場合、完全に修飾されたパブリケーションを抽出します。 dbxtract -f オプションによって TRUE に設定されます。
public a_bit_field	exclude_hooks	TRUE に設定した場合、プロシージャフックを除外します。 dbxtract -xh オプションによって TRUE に設定されます。
public a_bit_field	compress_output	TRUE に設定した場合、テーブルデータファイルを圧縮します。 dbunload -cp オプションによって TRUE に設定されます。
public a_bit_field	display_create	TRUE に設定した場合、データベース作成コマンドを表示します (sql または dbinit)。 dbunload -cm sql または -cm dbinit オプションによって TRUE に設定されます。
public a_bit_field	display_create_dbinit	TRUE に設定した場合、dbinit データベース作成コマンドを表示します。 dbunload -cm dbinit オプションによって TRUE に設定されます。
public a_bit_field	preserve_identity_values	TRUE に設定した場合、AUTOINCREMENT カラムの ID 値を保持します。 dbunload -l オプションによって TRUE に設定されます。
public a_bit_field	no_reload_status	TRUE に設定した場合、テーブルとインデックスの再ロードステータスメッセージが非表示になります。 dbunload -qr オプションによって TRUE に設定されます。
public a_bit_field	startline_name	予約済み。FALSE に設定します。
public a_bit_field	debug	予約済み。FALSE に設定します。
public a_bit_field	schema_reload	予約済み。FALSE に設定します。
public a_bit_field	genscript	予約済み。FALSE に設定します。
public a_bit_field	runscript	予約済み。FALSE に設定します。

変更子とタイプ	変数	説明
public a_bit_field	suppress_statistics	TRUE に設定した場合、カラム統計を含めません。 dbunload -ss オプションによって TRUE に設定されます。
public a_bit_field	dbdiff	予約済み。FALSE に設定します。
public a_bit_field	unload_password_hashes	TRUE に設定した場合、アップロード時に実際のパスワードハッシュを出力します。dbunload -up オプションにより、または -ac、-ar、-an オプションが存在する場合、TRUE に設定されます。-no または -k オプションが指定された場合、TRUE になることはありません。
public a_bit_field	user_list_provided	予約済み。FALSE に設定します。
public a_bit_field	table_list_patterns	予約済み。FALSE に設定します。
public p_name	user_list	予約済み。NULL に設定します。
public const char *	null_string	予約済み。NULL に設定します。
public const char *	force_data_format	予約済み。NULL に設定します。
public p_name	table_list_data	予約済み。NULL に設定します。
public a_bit_field	table_list_data_provided	予約済み。FALSE に設定します。
public a_bit_field	online_rebuild	TRUE に設定した場合、オンラインの再構築を実行します。dbunload -ao によって TRUE に設定されます。この機能は、DB_TOOLS_VERSION_17_0_0 以降のバージョンに装備されてます。
public a_bit_field	online_rebuild_from_backup	TRUE に設定した場合、運用データベースのバックアップからオンラインの再構築を実行します。dbunload -aob によって TRUE に設定されます。この機能は、DB_TOOLS_VERSION_17_0_0 以降のバージョンに装備されてます。
public SHOULD_STOP_CALLBACK	shouldstoprtn	コールバックルーチンのアドレス、または NULL。 ルーチンは、処理をすぐに中止する場合に TRUE、それ以外の場合に FALSE を返します。この機能は、DB_TOOLS_VERSION_17_0_0 以降のバージョンに装備されてます。

変更子とタイプ	変数	説明
public a_sql_uint32	online_rebuild_max_apply_sec	オンライン再構築中にインクリメンタルログを一度適用する場合にのみ、0 に設定します。 正の秒数に設定した場合、インクリメンタルバックアップと適用がこの秒数よりも少なくなるまでインクリメントバックアップのループ処理とログの適用を実行します。dbunload -aot によって設定されます。この機能は、DB_TOOLS_VERSION_17_0_0 以降のバージョンに装備されてます。
public const char *	temporary_directory	オンライン再構築テンポラリファイル用のディレクトリ (元のデータベースのフルバックアップも含みます)。 NULL の場合、デフォルトのテンポラリディレクトリが使用されます。dbunload -dt によって設定されます。この機能は、DB_TOOLS_VERSION_17_0_0 以降のバージョンに装備されてます。

備考

SQL Remote 抽出ユーティリティ dbxtract が使用するフィールドが示されます。

関連情報

冗長列挙 [731 ページ]
DBUnload(an_unload_db *) メソッド [722 ページ]

1.22.2.52 an_upgrade_db 構造体

DBTools ライブラリを使用してデータベースをアップグレードするために必要な情報を格納します。

構文

```
typedef struct an_upgrade_db
```

メンバー

an_upgrade_db のすべてのメンバー (継承されたメンバーも含みます) を以下に示します。

変数

変更子とタイプ	変数	説明
public unsigned short	version	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。
public MSG_CALLBACK	errorrtn	エラーメッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	msgrtn	情報メッセージコールバックルーチンのアドレス、または NULL。
public MSG_CALLBACK	statusrtn	ステータスメッセージコールバックルーチンのアドレス、または NULL。
public const char *	connectparms	データベース接続に必要なパラメータ。 次のような接続文字列のフォームを取ります。"UID=DBA;PWD=sql;DBF=demo.db". データベースサーバは、接続文字列の START パラメータによって起動されます。たとえば、"START=c:¥SQLAny17¥bin32¥dbsrv17.exe" などとなります。 次に START パラメータを含んだ完全な接続文字列の例を示します。"UID=DBA;PWD=sql;DBF=demo.db;START=c:¥SQLAny17¥bin32¥dbsrv17.exe"
public a_bit_field	quiet	TRUE に設定した場合、操作中にメッセージを出力しません。 dbupgrad -q オプションによって TRUE に設定されます。
public a_bit_field	jconnect	TRUE に設定した場合、データベースをアップグレードして jConnect プロシージャが含まれるようにします。 dbupgrad -i オプションによって FALSE に設定されます。
public a_bit_field	restart	TRUE に設定した場合、アップグレード後にデータベースを再起動します。 dbupgrad -nrs オプションによって FALSE に設定されます。

変更子とタイプ	変数	説明
public unsigned short	sys_proc_definer	16.0 より前のリリースからアップグレードする場合、0 を割り当ててデータベースをアップグレードし、レガシーシステムストアドプロシージャに 16.0 より前の SQL SECURITY モデルを設定します。 バージョン 16.0 以降のデータベースからアップグレードする場合、現在の SQL SECURITY モデルを保持します (-pd を指定しない場合と同じ)。 1 を割り当ててデータベースをアップグレードし、レガシーシステムストアドプロシージャに 16.0 より前の SQL SECURITY モデルを設定します (-pd y と同じ)。 2 を割り当ててデータベースをアップグレードし、レガシーシステムストアドプロシージャに 16.0 より前の SQL SECURITY モデルを設定します (-pd n と同じ)。

関連情報

DBUpgrade(const an_upgrade_db *) メソッド [723 ページ]

1.22.2.53 DBT_USE_DEFAULT_MIN_PWD_LEN 変数

DBT_USE_DEFAULT_MIN_PWD_LEN は、新しいデータベースでデフォルトの最小パスワード長を使用する (つまり、特定の長さを指定していない) ことを示します。

構文

```
#define DBT_USE_DEFAULT_MIN_PWD_LEN
```

備考

この値を使用して min_pwd_len フィールドを設定します。

1.22.3 ソフトウェアコンポーネントの終了コード

すべてのデータベースツールライブラリのエントリポイントが終了コードを返します。データベースサーバおよびユーティリティ (dbsrv17、dbbackup、dbspawn など) でもこれらの終了コードが使用されます。

コード	ステータス	説明
0	EXIT_OKAY	成功
1	EXIT_FAIL	一般的な失敗
2	EXIT_BAD_DATA	無効なファイルフォーマット
3	EXIT_FILE_ERROR	ファイルが見つからない、開くことができない
4	EXIT_OUT_OF_MEMORY	メモリ不足です
5	EXIT_BREAK	ユーザによる終了
6	EXIT_COMMUNICATIONS_FAIL	通信失敗
7	EXIT_MISSING_DATABASE	必要なデータベース名なし
8	EXIT_PROTOCOL_MISMATCH	クライアントとサーバのプロトコルが一致しない
9	EXIT_UNABLE_TO_CONNECT	データベースサーバと接続できない
10	EXIT_ENGINE_NOT_RUNNING	データベースサーバが起動されない
11	EXIT_SERVER_NOT_FOUND	データベースサーバが見つからない
12	EXIT_BAD_ENCRYPT_KEY	暗号化キーが見つからないか、不正である
13	EXIT_DB_VER_NEWER	データベースを実行するためにサーバをアップグレードする必要がある
14	EXIT_FILE_INVALID_DB	ファイルがデータベースでない
15	EXIT_LOG_FILE_ERROR	ログファイルが見つからないか、その他のエラーが発生した
16	EXIT_FILE_IN_USE	ファイルが使用中
17	EXIT_FATAL_ERROR	致命的なエラーが発生した
18	EXIT_MISSING_LICENSE_FILE	サーバライセンスファイルが見つからない
19	EXIT_BACKGROUND_SYNC_ABORTED	優先度の高い操作を続行するためにバックグラウンド同期がアボートされた
20	EXIT_FILE_ACCESS_DENIED	アクセスが拒否されたためデータベースを起動できない
21	EXIT_SERVER_NAME_IN_USE	同じ名前をもつ別のサーバが現在実行中です。
255	EXIT_USAGE	コマンドラインで無効なパラメータ

これらの終了コードは、*%SQLANY17%*`¥sdk¥include¥sqldef.h` ファイルで定義されています。

1.23 データベースおよびアプリケーションの配備

データベースアプリケーションを完了したら、エンドユーザにアプリケーションを配備します。

アプリケーションの SQL Anywhere の使い方 (クライアント/サーバ形式での組み込みデータベースとしてなど) によっては、SQL Anywhere ソフトウェアのコンポーネントを、アプリケーションとともに配備してください。データソース名などの設定情報も配備し、アプリケーションが SQL Anywhere と通信できるようにします。

> **注記**
>
> ファイルの再配布はライセンス契約に従います。このマニュアル内の記述は、ライセンス契約のどの条項にも優先しません。配備について検討する前にライセンス契約を確認してください。

次のステップについて説明します。

- 選択したアプリケーションプラットフォームとアーキテクチャに基づいて必要なファイルを決定します。
- クライアントアプリケーションを設定します。

この項の大半は、個々のファイルやファイルが配置される場所について説明しています。ただし、SQL Anywhere コンポーネントを配備する方法としては、*Deployment* ウィザードを使用するか、サイレントインストールを使用することをお奨めします。

このセクションの内容:

配備の種類 [788 ページ]
配備に必要なファイルは、選択する配備の種類によって異なります。

ファイルの配布方法 [789 ページ]
SQL Anywhere を配備するには、SQL Anywhere インストーラを使用するか、独自のインストーラを開発します。

インストールディレクトリとファイル名 [790 ページ]
配備されたアプリケーションが正しく動作するためには、データベースサーバとクライアントアプリケーションがそれぞれ必要とするファイルを見つけることができなければなりません。配備ファイルは、使用する SQL Anywhere のインストール環境と互いに同じ形式で置いてください。

Windows 用 Deployment ウィザード [793 ページ]
Deployment ウィザードは、SQL Anywhere for Windows の配備に推奨されるツールです。

Windows へのサイレントインストール [797 ページ]
サイレントインストールは、ユーザの入力を必要とせず、またインストールが発生していることをユーザに知らせることもなく実行されます。Windows オペレーティングシステムで、SQL Anywhere がサイレントにインストールされるように、ユーザ自身のインストールプログラムから SQL Anywhere インストーラを呼び出すことができます。

Linux/UNIX および MAC OS X の Deployment ウィザード [801 ページ]
Deployment ウィザードは、SQL Anywhere for Linux/UNIX および MAC OS X の配備に推奨されるツールです。

Linux/UNIX と Mac OS X でのサイレントインストール [803 ページ]
サイレントインストールは、ユーザの入力を必要とせず、またインストールが発生していることをユーザに知らせることもなく実行されます。Linux および Mac OS X オペレーティングシステムで、SQL Anywhere がサイレントにインストールされるように、ユーザ自身のインストールプログラムから SQL Anywhere インストーラを呼び出すことができます。

クライアントアプリケーションを配備するための要件 [805 ページ]

データベースサーバに対して実行されたクライアントアプリケーションをデプロイする場合、クライアントソフトウェア、クライアントデータベースインタフェースコンポーネント、適切なデータベース接続情報を各エンドユーザに提供する必要があります。

管理ツールの配備 [837 ページ]
ライセンス契約に従って、Interactive SQL および *SQL Central* を含む一連の管理ツールを配備できます。

マニュアルの配備 [860 ページ]
マニュアルを配備する方法には、2 つの選択肢があります。ヘルプファイルが配備されない DocCommentXchange (http://dcx.sap.com) を使用する方法と、HTMLHelp (.chm) ヘルプファイルを配備する方法です。

データベースサーバの配備 [860 ページ]
データベースサーバを稼働するには、一連の適切なファイルをインストールする必要があります。

Windows での DLL の登録 [867 ページ]
DLL ファイルによっては、SQL Anywhere での使用を目的として配備する場合に登録を必要とするものがあります。

外部環境のサポートの配備 [867 ページ]
複数のコンポーネントは、データベースアプリケーションの外部呼び出しをサポートする必要があります。

暗号化の配備 [870 ページ]
SQL Anywhere は、パスワードを使用したセキュアなクライアントログインおよびクライアントとデータベースサーバ間のネットワークトラフィック (TLS、HTTPS) の RSA 暗号化、データベースとデータベーステーブルの AES 暗号化、および暗号化機能をサポートしています。

LDAP の配備 [871 ページ]
SQL Anywhere は、LDAP (Lightweight Directory Access Protocol) ユーザ認証をサポートしています。これにより、クライアントアプリケーションは、LDAP サーバに認証を受けるためにユーザ ID およびパスワード情報をデータベースサーバに送信できるようになります。

組み込みデータベースアプリケーションの配備 [871 ページ]
埋め込みデータベースアプリケーションは、アプリケーションとデータベースが同じコンピュータにあるアプリケーションです。

1.23.1 配備の種類

配備に必要なファイルは、選択する配備の種類によって異なります。

使用可能ないくつかの配備モデルを次に示します。

クライアントの配備

SQL Anywhere のクライアント部分だけをエンドユーザに配備して、集中管理されたネットワークデータベースサーバに接続できるようにすることができます。

ネットワークサーバの配備

ネットワークサーバをオフィスに配備してから、クライアントをそのオフィス内の各ユーザに配備します。

組み込みデータベースの配備

パーソナルデータベースサーバで実行するアプリケーションを配備します。この場合は、クライアントとパーソナルサーバの両方をエンドユーザのコンピュータにインスールする必要があります。

SQL Remote の配備

SQL Remote アプリケーションの配備は、組み込みデータベース配備モデルの拡張モデルです。

Mobile Link の配備

Mobile Link サーバの配備については、Mobile Link マニュアルで説明します。

管理ツールの配備

Interactive SQL、*SQL Central*、その他の管理ツールを配備します。

1.23.2 ファイルの配布方法

SQL Anywhere を配備するには、SQL Anywhere インストーラを使用するか、独自のインストーラを開発します。

SQL Anywhere インストーラの使用

インストーラをエンドユーザが使用できるようにします。適切なオプションを選択することによって、各エンドユーザはそれぞれ必要なファイルを受け取れるようになります。

これは、ほとんどの場合の配備に適用できる最も簡単なソリューションです。この場合は、データベースサーバに接続する方法 (ODBC データソースなど) をエンドユーザに依然として提供する必要があります。

独自のインストーラの開発

SQL Anywhere ファイルを組み込んだ独自のソフトウェアインストーラを開発する理由はいくつかあります。これはより複雑なオプションであり、ここで説明する情報の大半は、独自のインストーラを作成するユーザのニーズに対応するためのものです。

クライアントアプリケーションアーキテクチャによって必要とされるサーバタイプとオペレーティングシステムに SQL Anywhere がすでにインストールされている場合、必要なファイルは SQL Anywhere インストールディレクトリ内の、適切に指定されたサブフォルダに置かれています。たとえば、ソフトウェアインストールディレクトリの `bin32` サブディレクトリには、32 ビット Windows オペレーティングシステムのサーバの実行に必要なファイルが含まれています。

どのオプションを選択する場合でも、ライセンス契約の条項に違反しないでください。

関連情報

Windows 用 Deployment ウィザード [793 ページ]
Linux/UNIX および MAC OS X の Deployment ウィザード [801 ページ]
Windows へのサイレントインストール [797 ページ]
Linux/UNIX と Mac OS X でのサイレントインストール [803 ページ]

1.23.3 インストールディレクトリとファイル名

配備されたアプリケーションが正しく動作するためには、データベースサーバとクライアントアプリケーションがそれぞれ必要とするファイルを見つけることができなければなりません。配備ファイルは、使用する SQL Anywhere のインストール環境と互いに同じ形式で置いてください。

実際には、ほとんどの実行ファイルは Windows 上の 1 つのディレクトリに属しています。たとえば、Windows では、クライアントとデータベースサーバの両方の実行ファイルは同じディレクトリ、つまりこの場合には SQL Anywhere インストールディレクトリの `bin32` または `bin64` サブディレクトリにインストールされます。

このセクションの内容:

Linux、UNIX、Mac OS X の場合の配備 [790 ページ]
UNIX の場合の配備は、Windows の場合の配備とは次のようにいくつかの点で異なります。

ファイルの命名規則 [791 ページ]
SQL Anywhere では、一貫したファイル命名規則を使用して、システムコンポーネントを簡単に識別してグループ分けできるようにしています。

1.23.3.1 Linux、UNIX、Mac OS X の場合の配備

UNIX の場合の配備は、Windows の場合の配備とは次のようにいくつかの点で異なります。

ディレクトリ構造

Linux、UNIX、Mac OS X のインストール環境の場合、デフォルトのディレクトリ構造は次のようになります。

ディレクトリ	目次
/opt/sqlanywhere17`/bin32` および */opt/sqlanywhere17*`/bin64`	実行ファイル、ライセンスファイル
/opt/sqlanywhere17`/lib32` および */opt/sqlanywhere17*`/lib64`	共有オブジェクトと共有ライブラリ
/opt/sqlanywhere17`/res`	文字列ファイル

AIX の場合、デフォルトのルートディレクトリは `/usr/lpp/sqlanywhere17` であり、*/opt/sqlanywhere17* ではありません。

Mac OS X の場合、デフォルトのルートディレクトリは */Applications/SQLAnywhere17/System* であり、*/opt/sqlanywhere17* ではありません。

配備の複雑さに応じて、アプリケーションに必要なすべてのファイルを 1 つのディレクトリに格納することもできます。特に、配備に必要なファイルの数が少ない場合は、この方法を使用することによって、より簡単に配備できることがあります。このディレクトリは、独自のアプリケーションに使用するディレクトリと同じにできます。

ファイルのサフィックス

表では、共有オブジェクトにはサフィックス `.so` または `.so.1` が付いています。更新がリリースされているため、バージョン番号は 1 より大きい場合があります。簡素化するために、バージョン番号が示されていない場合があります。

AIX の場合、サフィックスにはバージョン番号が含まれないため、単に .so です。

シンボリックリンク

各共有オブジェクトは、追加サフィックス .1 (数字の 1) が付いた同じ名前のファイルへのシンボリックリンク (symlink) としてインストールされます。たとえば、libdblib*17*.so は同じディレクトリ内のファイル libdblib*17*.so.1 へのシンボリックリンクです。

更新がリリースされると、シンボリックリンクが適切にリダイレクトされるように、バージョンサフィックスが .1 より大きくなっている場合があります。

Mac OS X では、Java クライアントアプリケーションから直接ロードする dylib については jnilib シンボリックリンクを作成してください。

スレッドアプリケーションと非スレッドアプリケーション

ほとんどの共有オブジェクトは、2 つの形式で提供されます。その一方は、ファイルのサフィックスの前に追加文字 _r が付けられます。たとえば、libdblib*17*.so.1 の他に libdblib*17*_r.so.1 というファイルが存在します。この場合、スレッドアプリケーションは名前のサフィックスが _r である共有オブジェクトにリンクされる必要があるのに対して、非スレッドアプリケーションは名前のサフィックスが _r ではない共有オブジェクトにリンクされる必要があります。ときには、共有オブジェクトの 3 番目の形式として、ファイルのサフィックスの前に _n が付けられています。これは、非スレッドアプリケーションで使用される共有オブジェクトのバージョンです。

文字セット変換

データベースサーバの文字セット変換を使用する場合、次のファイルが必要です。

- libdbicu*17*.so.1
- libdbicu*17*_r.so.1
- libdbicudt*17*.so.1
- sqlany.cvf

環境変数

Linux、UNIX、Mac OS X の場合は、SQL Anywhere アプリケーションとライブラリを配置するために、システムに環境変数を設定してください。必要な環境変数を設定するためのテンプレートとして、sa_config.sh または sa_config.csh のいずれか (*/opt/sqlanywhere17*/bin32 および */opt/sqlanywhere17*/bin64 ディレクトリにあります) のうち、シェルに適した設定スクリプトを使用してください。これらのファイルによって設定される環境変数には PATH、LD_LIBRARY_PATH、SQLANY17 などがあります。

1.23.3.2 ファイルの命名規則

SQL Anywhere では、一貫したファイル命名規則を使用して、システムコンポーネントを簡単に識別してグループ分けできるようにしています。

これらの規則は、次のとおりです。

バージョン番号

SQL Anywhere のバージョン番号は、メインサーバコンポーネント (実行ファイル、ダイナミックリンクライブラリ、共有オブジェクト、ライセンスファイルなど) のファイル名に示されます。

たとえば、ファイル *dbsrv17*.exe は Windows 用のバージョン *17* の実行プログラムです。

言語

言語リソースライブラリで使用される言語は、ファイル名の中の 2 文字のコードで示されます。バージョン番号の前の 2 文字が、ライブラリで使用されている言語を示します。たとえば、dblgen*17*.dll は英語版のメッセージリソースライブラリです。これらの 2 文字のコードは ISO 標準 639-1 に準拠したものです。

言語ラベルの詳細については、言語選択ユーティリティ (dblang) マニュアルを参照してください。

SQL Anywhere の複数のローカライズバージョンがあります。

その他のファイルタイプ

次の表は、ファイルの拡張子に対応する SQL Anywhere ファイルのプラットフォームと機能を示します。SQL Anywhere では、可能なかぎり標準ファイル拡張子の規則に従います。

ファイル拡張子	プラットフォーム	ファイルタイプ
.bat、.cmd	Windows	バッチコマンドファイル
.chm, .chw	Windows	ヘルプシステムファイル
.dll	Windows	ダイナミックリンクライブラリ
.exe	Windows	実行ファイル
.ini	[すべて]	初期化ファイル
.lic	[すべて]	ライセンスファイル
.lib	開発ツールによって異なる	Embedded SQL 実行プログラム作成用の静的ランタイムライブラリ
.res	Linux、UNIX、Mac OS X	非 Windows 環境用の言語リソースファイル
.so	Linux、UNIX	共有オブジェクトまたは共有ライブラリファイル (Windows DLL に相当)
.bundle、.dylib	Mac OS X	共有オブジェクトファイル (Windows DLL に相当)

データベースファイル名

SQL Anywhere データベースは、次の 2 つの要素で構成されます。

データベースファイル

系統立てて管理されたフォーマットで情報を保存するために使用します。デフォルトで、このファイルは拡張子として .db を使います。その他の dbspace ファイルも存在する場合があります。これらのファイルには任意のファイル拡張子が付いているか、または拡張子がない場合があります。

トランザクションログファイル

データベースファイルに保存されているデータに加えられた変更をすべて記録するために使用します。デフォルトでは、ファイル拡張子には .log を使用します。トランザクションログファイルが存在せずログファイルを使用するように指定されている場合は、SQL Anywhere がこのファイルを生成します。トランザクションログミラーには、デフォルトのファイル拡張子 .mlg が使用されます。

これらのファイルは、SQL Anywhere のリレーショナルデータベース管理システムによって、更新、保守、管理が行われます。

1.23.4 Windows 用 *Deployment* ウィザード

Deployment ウィザードは、SQL Anywhere for Windows の配備に推奨されるツールです。

Deployment ウィザードを使用すると、新しいインストールイメージを作成することも、既存のインストールイメージをアップグレードできます。*Deployment* ウィザードでは、次のコンポーネントを一部またはすべて含むインストーラファイルを作成します。

- ODBC などのクライアントインタフェース
- リモートデータアクセス、データベースツール、暗号化を含む SQL Anywhere サーバ
- Ultra Light リレーショナルデータベース
- Mobile Link サーバ、クライアント、暗号化
- Interactive SQL や *SQL Central* などの管理ツール

Deployment ウィザードを使用すると、Microsoft Windows インストーラパッケージファイルまたは Microsoft Windows インストーラマージモジュールファイルを作成できます。

Microsoft Windows インストーラパッケージファイル

アプリケーションのインストールに必要な手順とデータが含まれているファイルです。インストーラパッケージファイルのファイル拡張子は `.msi` です。

Microsoft Windows インストーラマージモジュールファイル

共有コンポーネントのインストールに必要なすべてのファイル、リソース、レジストリエントリ、セットアップロジックが含まれている簡易型の Microsoft Windows インストーラパッケージファイルです。マージモジュールのファイル拡張子は `.msm` です。

マージモジュールには、インストーラパッケージファイルに含まれているいくつかの重要なデータベーステーブルがないため、単独ではインストールできません。マージモジュールにはその他の固有のテーブルも含まれています。マージモジュールによって配信される情報をアプリケーションとともにインストールするには、最初にモジュールをアプリケーションのインストーラパッケージ (`.msi`) ファイルにマージしてください。マージモジュールは、次の部分から構成されます。

- マージモジュールによって配信されるインストールプロパティとセットアップロジックが含まれたマージモジュールデータベース
- モジュールについて記述したマージモジュールのサマリ情報ストリーム
- ストリームとしてマージモジュールの内部に格納された `MergeModule.CAB` キャビネットファイル。このキャビネットファイルには、マージモジュールによって配信されるコンポーネントに必要なすべてのファイルが含まれています。マージモジュールによって配信されるすべてのファイルは、マージモジュールの構造化されたストレージ内にストリームとして埋め込まれたキャビネットファイルの内部に格納されている必要があります。標準のマージモジュールでは、キャビネット名は常に `MergeModule.CAB` です。

Deployment ウィザードにより、SQL Anywhere に含まれるコンポーネントのサブセットを選択できます。各コンポーネントは他のコンポーネントに依存しているため、ウィザードによって選択されたファイルには他のカテゴリのファイルが含まれる場合があります。

それぞれの選択可能コンポーネントに含まれるファイルを特定したい場合は、MSI インストーライメージを作成し、すべてのコンポーネントを選択します。各コンポーネントに含まれるファイルを示す MSI ログファイルが作成されます。このテキストファイルは、テキストエディタで確認できます。これらは *Feature: SERVER32_TOOLS* や *Feature: CLIENT64_TOOLS* などの見出しで、*Deployment* ウィザードのコンポーネントと密接に対応しています。ファイルを調べることにより、各グループに何が含まれているかがわかります。

このセクションの内容:

Windows での Deployment ウィザードの実行 [794 ページ]
Deployment ウィザードは、Windows 用のデータベースおよびアプリケーションの配備を作成するのに使用します。

Windows msiexec インストールユーティリティ [795 ページ]
Microsoft msiexec ユーティリティは、Windows の配備をインストールまたはアンインストールします。

関連情報

Windows へのサイレントインストール [797 ページ]

1.23.4.1 Windows での *Deployment* ウィザードの実行

Deployment ウィザードは、Windows 用のデータベースおよびアプリケーションの配備を作成するのに使用します。

前提条件

ファイルの再配布はライセンス契約に従います。SQL Anywhere ファイルを再配布するためのライセンスがあることを確認してください。ライセンス契約を確認してから、処理を続行してください。

コンテキスト

データベースおよびアプリケーションの配備

手順

1. *Deployment* ウィザードを実行するには、*[スタート]* > *[プログラム]* > *[SQL Anywhere17]* > *[管理ツール]* > *[Windows に展開]* をクリックします。
2. ウィザードの指示に従います。
3. 新しいインストールを作成するか、アップグレードインストールを作成するかを判別します。

結果

選択した *[インストール先パス]* には、配備パッケージ (`sqlany17.msi` など) とログファイル (`sqlany17.msi.log` など) が格納されます。

例

SQL Anywhere インストール環境の `Deployment` サブフォルダから *Deployment* ウィザードを実行することもできます。次に例を示します。

```
%SQLANY17%¥Deployment¥DeploymentWizard.exe
```

次のステップ

配備パッケージの作成が完了したら、インストールしてテストしてください。

インストールイメージの製品コード (ProductCode) を参照するには、Microsoft の Orca ツールを使用して Property テーブルを表示します。製品コードは、アップグレードインストールイメージの作成時に保持されます。ソフトウェアをアンインストールするときに、製品コードを使用できます。

関連情報

Windows msiexec インストールユーティリティ [795 ページ]

1.23.4.2 Windows msiexec インストールユーティリティ

Microsoft msiexec ユーティリティは、Windows の配備をインストールまたはアンインストールします。

構文

```
msiexec [ options ] [ SQLANYDIR=path ]
```

オプション	説明
/log`log-filename`	このパラメータにより、Microsoft Windows インストーラは操作をファイル (`sqlany17.log` など) に記録します。インストーラログファイルは問題の検出に役立ちます。
/package`package-name`	このパラメータにより、Microsoft Windows インストーラは指定されたパッケージ (`sqlany17.msi` など) をインストールします。

オプション	説明
REINSTALL=ALL	このパラメータは、既存のインストールを介してアップグレードインストールの全機能の再インストールを Microsoft Windows インストーラに指示します。
REINSTALLMODE=vomus	このパラメータは、既存のインストールを介してアップグレードインストールのインストールを Microsoft Windows インストーラに指示します。再インストールオプションコードの説明については、Microsoft マニュアルを参照してください。推奨オプションコードは *vomus* です。
/qn	このパラメータにより、Microsoft Windows インストーラはユーザの操作を必要とすることなく、バックグラウンドで実行されます。
/uninstall`package-name` | `product-code`	このパラメータにより、Microsoft Windows インストーラは指定された MSI ファイルまたは製品コードに関連付けられている製品をアンインストールします。`product-code` は、*Deployment ウィザード*を実行して配備パッケージを作成するときに作成されます。
SQLANYDIR=`path`	このパラメータの値によって、ソフトウェアのインストール先のパスが設定されます。このオプションはサイレントインストールに役立ちます。*/uninstall* では無視されます。

備考

/qn オプションを使用すると、ユーザにメッセージを表示せずに操作を実行できます。Windows 7 以降のバージョンの Windows では、ソフトウェアのインストールまたはアンインストールを行う場合に管理者権限が必要です。/qn オプションを使用した場合は、管理者権限が自動的に要求されなくなります。

例

次のコマンドは、配備ファイルをインストールします。

```
msiexec /package sqlany17.msi
```

次のコマンドは、指定されたフォルダへの配備ファイルのサイレントインストールを実行し、結果をファイルに記録します。コンピュータシステムが警告なしでシャットダウンしたり再起動したりする可能性があるため、/qn オプションの使用には注意する必要があります。

```
msiexec /qn /package sqlany17.msi /log sqlany17.log SQLANYDIR=c:¥sa17
```

次のコマンドは、指定されたパッケージを使用してサイレントアンインストールを実行し、結果をファイルに記録します。

```
msiexec /qn /uninstall sqlany17.msi /log sqlany17.log
```

次のコマンドは、指定された製品コードを使用してサイレントアンインストールを実行します。

```
msiexec.exe /qn /uninstall {7C99D24E-AE1B-4770-9015-65B805950E3D}
```

次のコマンドでは、アップグレードインストールを使用して配備を再インストールし、プロセスでログファイルを作成します。

```
msiexec /package sqlany17sp2.msi REINSTALL=ALL REINSTALLMODE=vomus /log
upgrade.log
```

関連情報

Windows での Deployment ウィザードの実行 [794 ページ]

1.23.5 Windows へのサイレントインストール

サイレントインストールは、ユーザの入力を必要とせず、またインストールが発生していることをユーザに知らせることもなく実行されます。Windows オペレーティングシステムで、SQL Anywhere がサイレントにインストールされるように、ユーザ自身のインストールプログラムから SQL Anywhere インストーラを呼び出すことができます。

SQL Anywhere インストールプログラム `setup.exe` の一般的なオプションは次のとおりです。

/l:language_id

言語識別子は、インストールの言語を表すロケール番号です。たとえば、ロケール ID 1033 は米国英語、ロケール ID 1031 はドイツ語、ロケール ID 1036 はフランス語、ロケール ID 1041 は日本語、ロケール ID 2052 は簡体字中国語を表します。

/s

このオプションにより、初期化ウィンドウを非表示にします。このオプションを /v と一緒に使用します。

/v:

Microsoft Windows インストーラツールの MSIEXEC にパラメータを指定します。

/qn

これは MSIEXEC オプションです (ユーザインタラクションなし)。

次のコマンドラインの例では、インストールイメージディレクトリがドライブ `d:` のディスクの `pkg¥SQLAnywhere` ディレクトリにあることを前提としています。

```
d:¥pkg¥sqlanywhere¥setup.exe /l:1033 /s "/v: /qn
    REGKEY=NEEDA-REAL0-KEY12-34567-89012 INSTALLDIR=c:¥sa17 DIR_SAMPLES=c:
¥sa17¥Samples"
```

注記

上記コマンドの `setup.exe` は、SQLANY32.msi ファイルや SQLANY64.msi ファイルと同じディレクトリにあるものです。これらのファイルの親ディレクトリにある setup.exe は、サイレントインストールをサポートしていません。

コンピュータシステムが警告なしでシャットダウンしたり再起動したりする可能性があるため、MSIEXEC /qn オプションの使用には注意する必要があります。

/qn (ユーザインタフェースなし) などのオプションを使用すると、インストール中に警告やエラーメッセージのダイアログが表示されません。インストールプロセスで予期しないことが発生したかどうかを確認するために、インストールログの調査は必要です。

ソフトウェアアップデートは PackageForTheWeb (PFTW) として配布されます。これは、自己完結型の自己解凍ファイルに埋め込まれた InstallShield パッケージアプリケーションです。PackageForTheWeb 自己解凍型の実行ファイルのサイレントインストールには、よく似た構文があります。次に、コマンドラインの例を示します。

```
SA17_Windows_1704_2034_EBF.exe /s /a /l:1033 /s "/v: /qn"
```

最初の /s オプションは自己解凍型の実行ファイルをサイレント化し、/a オプションは、解凍および実行された `setup.exe` ファイルに以降のすべてを渡すことを自己解凍型の実行ファイルに示します。残りのオプション (/l、/s、/v:、/qn) を上記に示します。登録キーおよびインストールディレクトリなどのプロパティは、既存のインストールから判別されるため、指定する必要はありません。

次のプロパティは、SQL Anywhere インストーラに適用されます。

INSTALLDIR

このパラメータの値が、ソフトウェアのインストール先のパスとなります。

DIR_SAMPLES

このパラメータの値が、サンプルプログラムのインストール先のパスとなります。

DIR_SQLANY_MONITOR

このパラメータの値が、SQL Anywhere モニタデータベース (`samonitor.db`) のインストール先のパスとなります。

USERNAME

このパラメータの値が、このインストールに関して記録するユーザ名となります (USERNAME=¥"John Smith¥" など)。

COMPANYNAME

このパラメータの値が、このインストールに関して記録する会社名となります (COMPANYNAME=¥"Smith Holdings¥" など)。

REGKEY

このパラメータの値は、有効なソフトウェア登録キーである必要があります。

REGKEY_ADD_1

このパラメータの値は、FIPS 暗号化などのアドオン機能の有効なソフトウェア登録キーである必要があります。このプロパティは、インストーラの今回の実行中にインストールするオプションのアドオン機能がある場合にのみ適用されます。

REGKEY_ADD_2

このパラメータの値は、FIPS 暗号化などのアドオン機能の有効なソフトウェア登録キーである必要があります。このプロパティは、インストーラの今回の実行中にインストールするオプションのアドオン機能がある場合にのみ適用されます。

REGKEY_ADD_3

このパラメータの値は、FIPS 暗号化などのアドオン機能の有効なソフトウェア登録キーである必要があります。このプロパティは、インストーラの今回の実行中にインストールするオプションのアドオン機能がある場合にのみ適用されます。

次の例は、SQL Anywhere インストールのプロパティを指定する方法を示します。

```
d:¥pkg¥sqlanywhere¥setup.exe /s "/v: /qn
```

```
USERNAME=¥"John Smith¥"
COMPANYNAME=¥"Smith Holdings¥"
REGKEY=NEEDA-REAL0-KEY12-34567-89012
REGKEY_ADD_1=<an add-on software registration key>
REGKEY_ADD_2=<another add-on software registration key>
INSTALLDIR=c:¥sa17
DIR_SAMPLES=c:¥sa17¥Samples
DIR_SQLANY_MONITOR=c:¥sa17¥Monitor"
```

上記のテキストは、長さの都合上、複数の行にわたって表示されていますが、実際は 1 行のテキストとして指定します。円記号を使用して、内部引用符をエスケープしている点に注意してください。

次のプロパティは SQL Anywhere モニタインストーラ (`d:¥pkg¥Monitor¥setup.exe` にあります) に適用されます。

INSTALLDIR

このパラメータの値が、ソフトウェアのインストール先のパスとなります。

DIR_SQLANY_MONITOR

このパラメータの値が、SQL Anywhere モニタデータベース (`samonitor.db`) のインストール先のパスとなります。

REGKEY

このパラメータの値は、有効なソフトウェアインストールキーである必要があります。

REGKEY_ADD_1

このパラメータの値は、FIPS 暗号化などのアドオン機能の有効なソフトウェアインストールキーである必要があります。このプロパティは、インストーラの今回の実行中にインストールするオプションのアドオン機能がある場合にのみ適用されます。

REGKEY_ADD_2

このパラメータの値は、FIPS 暗号化などのアドオン機能の有効なソフトウェアインストールキーである必要があります。このプロパティは、インストーラの今回の実行中にインストールするオプションのアドオン機能がある場合にのみ適用されます。

次の例は、SQL Anywhere モニタインストールのプロパティを指定する方法を示します。

```
d:¥software¥monitor¥setup.exe /s "/v: /qn
REGKEY=<SQL Anywhere Monitor registration key>
REGKEY_ADD_1=<an add-on software registration key>
REGKEY_ADD_2=<another add-on software registration key>
INSTALLDIR=c:¥sa17"
```

上記のプロパティ値の設定以外に、次の SQL Anywhere のコンポーネントや機能をコマンドラインから選択する場合があります。

機能	プロパティ	x86 のデフォルト	x64 のデフォルト
管理ツール (32 ビット)	AT32	1	0
管理ツール (64 ビット)	AT64	0	1
FIPS 認定の暗号化	FIPS *		
高可用性	HA *		
インメモリモード	IM *		
Mobile Link (64 ビット)	ML64	0	1
Mobile Link (32 ビット)	ML32	1	0

機能	プロパティ	x86 のデフォルト	x64 のデフォルト
読み込み専用スケールアウト	SON *		
Relay Server (64 ビット)	RS64	0	1
SACI_SDK	SACI	0	0
サンプル	SAMPLES	1	1
SQL Anywhere クライアント (32 ビット)	CLIENT32	1	1
SQL Anywhere クライアント (64 ビット)	CLIENT64	0	1
SQL Anywhere サーバ (32 ビット)	SERVER32	1	0
SQL Anywhere サーバ (64 ビット)	SERVER64	0	1
SQL Anywhere モニタ (32 ビット)	SM32	1	0
SQL Anywhere モニタ (64 ビット)	SM64	0	1
SQL Remote (32 ビット)	SR32	1	0
SQL Remote (64 ビット)	SR64	0	1
Ultra Light	UL	1	1

* これらの機能には、追加の適切なソフトウェア登録キーが必要です。

プロパティ値を 1 に設定して機能を選択するか、0 に設定して機能を省略します。たとえば、32 ビット Mobile Link を省略するには、ML32=0 と設定します。32 ビット管理ツールを選択するには、AT32=1 と設定します。これらのプロパティは、デフォルト選択を上書きする場合に使用されます。

デフォルトを適用できる場合は、機能の選択状態を指定する必要はありません。たとえば、32 ビットの Mobile Link 機能をインストールし、サンプルをインストールしないで、64 ビットのコンピュータでのデフォルトの選択を上書きするには、次のコマンドラインを使用します。

```
setup.exe "/v ml32=1 samples=0"
```

プロパティ名では大文字と小文字が区別されません。

現在の登録キーでは使用できる機能が制限されることがあります。コマンドラインオプションを使用してこれらの制限を上書きすることはできません。たとえば、登録キーで FIPS 暗号化のインストールが許可されない場合は、コマンドラインで FIPS=1 を使用してもこの機能は選択されません。

MSI ログを生成するには、コマンドラインの `/v:` の後に次のテキストを追加します。

```
/l*v! logfile
```

この例では、`logfile` がメッセージログファイルのフルパスおよびファイル名となります。このパスはあらかじめ用意しておく必要があります。このオプションによって生成されるログはきわめて冗長なものとなり、インストールの実行に必要な時間が大幅に伸びます。

サイレントインストールだけでなく、サイレントアンインストールも実行できます。サイレントアンインストールを実行するためのコマンドラインの例を次に示します。

```
msiexec.exe /qn /uninstall {08787C4E-7210-4FA8-8D09-B393E76CA1A8} /l*v %temp%
¥sauninstall.log
```

上記の例では、Microsoft Windows インストーラツールを直接呼び出します。

/qn

このパラメータにより、Microsoft Windows インストーラはユーザの操作を必要とすることなく、バックグラウンドで実行されます。

/uninstall <製品コード>

このパラメータにより、Microsoft Windows インストーラは指定された製品コードに関連付けられている製品をアンインストールします。上記のコードは SQL Anywhere ソフトウェアのものです。

/l*v %temp%¥sauninstall.log

このパラメータにより、Microsoft Windows インストーラはログファイルを指定された場所に作成します。

SQL Anywhere バージョン 17.0.4 の製品コードは次のとおりです。

{08787C4E-7210-4FA8-8D09-B393E76CA1A8}

SQL Anywhere ソフトウェア
{8AF8FFC2-8C0E-40FE-BFD7-872E65EB235C}

SQL Anywhere クライアント専用ソフトウェア
{CA1B1A3C-9CD6-4BF7-BE9A-0BD4C2601DB9}

SQL Anywhere モニタ

関連情報

コマンドラインオプション

1.23.6 Linux/UNIX および MAC OS X の Deployment ウィザード

Deployment ウィザードは、SQL Anywhere for Linux/UNIX および MAC OS X の配備に推奨されるツールです。

Deployment ウィザードは、次のコンポーネントの一部または全部を選択する場合に使用します。

- ODBC などのクライアントインタフェース
- リモートデータアクセス、データベースツール、暗号化を含む SQL Anywhere サーバ
- Ultra Light リレーショナルデータベース
- Mobile Link サーバ、クライアント、暗号化
- Interactive SQL や *SQL Central* などの管理ツール

Deployment ウィザードにより、SQL Anywhere に含まれるコンポーネントのサブセットを選択できます。各コンポーネントは他のコンポーネントに依存しているため、ウィザードによって選択されたファイルには他のカテゴリのファイルが含まれる場合があります。

このセクションの内容:

Linux/UNIX および MAC OS X での Deployment ウィザードの実行 [802 ページ]
Deployment ウィザードは、Linux/UNIX および MAC OS X 用のデータベースおよびアプリケーションの配備を作成するのに使用します。

1.23.6.1 Linux/UNIX および MAC OS X での Deployment ウィザードの実行

Deployment ウィザードは、Linux/UNIX および MAC OS X 用のデータベースおよびアプリケーションの配備を作成するのに使用します。

前提条件

ファイルの再配布はライセンス契約に従います。SQL Anywhere ファイルを再配布するためのライセンスがあることを確認してください。ライセンス契約を確認してから、処理を続行してください。

コンテキスト

データベースおよびアプリケーションの配備

手順

1. `sa_config.sh` / `sa_config.csh` ファイルを指定して環境を設定します。次に例を示します。

   ```
   . /opt/sqlanywhere17/bin64/sa_config.sh
   ```

2. 配備ウィザードを起動します。

   ```
   $SQLANY17/deployment/deployment_wizard.sh
   ```

3. 指示に従います。インストール場所を指定するように要求されます。デフォルトの場所が正しい場合は、そのまま使用します。
4. 配備するコンポーネントを選択します。
5. ファイルのリストをテキストファイル形式 (`deployment.txt` など) で保存します。このファイルを使用すると、ターボールを作成できます。

6. 配備ウィザードを終了します。
7. テキストファイルを使用してターボールを作成します。次に例を示します。

```
cd $SQLANY17
tar czf deployment.tar -T deployment.txt
```

結果

配備パッケージを tar アーカイブファイル形式で作成しました。

次のステップ

配備パッケージの作成が完了したら、インストールしてテストしてください。

1.23.7 Linux/UNIX と Mac OS X でのサイレントインストール

サイレントインストールは、ユーザの入力を必要とせず、またインストールが発生していることをユーザに知らせることもなく実行されます。Linux および Mac OS X オペレーティングシステムで、SQL Anywhere がサイレントにインストールされるように、ユーザ自身のインストールプログラムから SQL Anywhere インストーラを呼び出すことができます。

SQL Anywhere インストールプログラム *setup* のオプションは次のとおりです。

オプション	説明
-company `company` (-c `company`)	会社名を指定します
-help (-h)	このヘルプ画面を表示します
-l_accept_the_license_agreement	ライセンス契約に同意します
-install `packages`	`packages` は、複数のパッケージをカンマで区切ったリストです
-install_icons `dir` (-ii `dir`)	SQL Anywhere ディレクトリを指定して、アイコンをインストールします
-list_packages	使用可能なパッケージのリストを表示します
-name `username` (-n `username`)	ユーザ名を指定します
-nogui	シェルスクリプトのみを使用します。gui を呼び出しません
-regkey `key` (-k `key`)	`key` をレジストリキーとして使用します
-seat-model `model` (-m `model`)	シートモデルを指定します (下位互換で使用できますが、無視されます)
-seats `num` (-s `num`)	シートまたはプロセッサの数を指定します
-silent (-ss)	サイレントインストール

オプション	説明
-sqlany-dir `dir` (-d `dir`)	SQL Anywhere インストールディレクトリを設定します
-type {CREATE|MODIFY|UPGRADE}	指定したタイプとしてインストールを実行します

これらのオプション名では大文字と小文字を区別するため、次のように指定する必要があります (たとえば、*-SILENT* は受け付けられません)。オプションの値では大文字と小文字は区別されません (たとえば、*modify* を *MODIFY* の代わりに使用できます)。

次の例では、指定したレジストリキーで使用可能なパッケージを示します。

```
setup -list_packages -k NEEDA-REAL0-KEY12-34567-89012
```

次の SQL Anywhere パッケージを選択できます。

機能	パッケージ	x86 のデフォルト	x64 のデフォルト
管理ツール (32 ビット)	admintools32	○	×
管理ツール (64 ビット)	admintools64	×	○
FIPS 認定の強力な暗号化	fips *		
高可用性	high_avail *		
インメモリモード	in_memory *		
Mobile Link (64 ビット)	ml64	×	○
Mobile Link クライアント (32 ビット)	mobilink_sqlany32	○	×
読み込み専用スケールアウト	scaleoutnodes *		
Relay Server (64 ビット)	relayserver64	×	○
サンプル	samples	○	○
SQL Anywhere クライアント (32 ビット)	sqlany_client32	○	○
SQL Anywhere クライアント (64 ビット)	sqlany_client64	×	○
SQL Anywhere サーバ (32 ビット)	sqlany32	○	×
SQL Anywhere サーバ (64 ビット)	sqlany64	×	○
SQL Anywhere モニタ (32 ビット)	samon32	○	×
SQL Anywhere モニタ (64 ビット)	samon64	×	○
SQL Remote (32 ビット)	dbremote32	○	×
SQL Remote (64 ビット)	dbremote64	×	○
Ultra Light エンジン (64 ビット)	ultralite64	○	○

* これらの機能には、追加の適切なソフトウェア登録キーが必要です。

インストールするパッケージを指定していない場合、ライセンス登録キーでのみ制限されたプラットフォームの (「Yes」で示された) デフォルトパッケージがインストールされます。つまり、32 ビットパッケージは x86 プラットフォームに、32 ビットパッケージに従った 64 ビットパッケージは x64 プラットフォームにインストールされます。次の例は、パッケージのデフォルトセットをインストールする方法を示します。

```
setup -k NEEDA-REAL0-KEY12-34567-89012 -ss ¥
    -I_accept_the_license_agreement -name "Sally Smith" -company "SAP AG"
```

インストールするパッケージを (*-install* オプションで) 指定していない場合、インストールしたいすべてのパッケージを指定する必要があります。次の例は、パッケージを指定する方法を示します。

```
setup -k NEEDA-REAL0-KEY12-34567-89012 -install sqlany64,sqlany_client64 -ss ¥
    -I_accept_the_license_agreement -name "Joe Jones" -company "SAP AG"
```

パッケージ名の大文字と小文字は区別されません。

現在の登録キーでは使用できる機能が制限されることがあります。コマンドラインオプションを使用してこれらの制限を上書きすることはできません。たとえば、登録キーで FIPS 暗号化のインストールが許可されない場合は、*install* に *fips* を含めてもこの機能は選択されません。

silent オプションを使用する場合、ライセンス契約を読んでおき、コマンドラインで *l_accept_the_license_agreement* オプションを指定する必要があります。また、*name* オプションと *company* オプションを使用してユーザ名と企業名を指定する必要もあります。

silent オプションを省略した場合、設定が要求されます。その場合、プロンプトごとの新しいデフォルトを設定するために、他のオプションが使用されます。

seat_model オプションを下位互換のために使用できますが、無視されます。シートモデルはライセンスキーで定義されます。

setup でインストールされたソフトウェアを案インストールするには、SQL Anywhere インストールのルートに置かれた *uninstall* スクリプトを使用します。

```
$SQLANY17/uninstall
```

上記の例では、*setup* でインストールされたパッケージのみをアンインストールします。

1.23.8 クライアントアプリケーションを配備するための要件

データベースサーバに対して実行されたクライアントアプリケーションをデプロイする場合、クライアントソフトウェア、クライアントデータベースインタフェースコンポーネント、適切なデータベース接続情報を各エンドユーザに提供する必要があります。

特に、各エンドユーザに次の項目を提供する必要があります。

クライアントアプリケーション

アプリケーションソフトウェア自体はデータベースソフトウェアとは関係がないため、ここでは記述しません。

クライアントデータベースインタフェースファイル

クライアントアプリケーションには、それが使用するクライアントデータベースインタフェース (.NET、ADO、OLE DB、ODBC、JDBC、Embedded SQL、Perl、PHP、Python、Ruby、C/C++ API、または Open Client) 用のファイルが必要です。jConnect または Open Client を使用する TDS クライアントを除き、クライアントアプリケーションでは暗号化サポートライブラリも 1 つ必要です。

接続情報

各クライアントアプリケーションにはデータベース接続情報が必要です。

必要なインタフェースファイルと接続情報は、アプリケーションが使用するインタフェースによって異なります。各インタフェースは個別に説明します。

クライアントを配備する最も簡単な方法は、*Deployment* ウィザードを使用することです。

このセクションの内容:

.NET クライアントの配備 [806 ページ]
SQL Anywhere .NET データプロバイダを使用する ADO.NET アプリケーションは、いくつかのプロバイダコンポーネントを必要とします。

OLE DB と ADO クライアントの開発 [810 ページ]
SQL Anywhere OLE DB プロバイダを使用する OLE DB および ADO アプリケーションは、いくつかのプロバイダコンポーネントを必要とします。

OLE DB プロバイダのカスタマイズ [811 ページ]
プロバイダの独自のバージョンを用意して、OLE DB プロバイダをカスタマイズできます。

ODBC クライアントの配備 [816 ページ]
SQL Anywhere ODBC ドライバを使用するアプリケーションは複数のコンポーネントを必要とします。

Embedded SQL クライアントの配備 [825 ページ]
Embedded SQL を使用するアプリケーションは複数のコンポーネントを必要とします。

JDBC クライアントの配備 [827 ページ]
JDBC を使用するアプリケーションは複数のコンポーネントを必要とします。

PHP クライアントの配備 [829 ページ]
SQL Anywhere PHP 拡張を使用するアプリケーションは複数のコンポーネントを必要とします。

Open Client アプリケーションの配備 [837 ページ]
Open Client アプリケーションを配備するには、各クライアントコンピュータに SAP Open Client 製品が必要です。

関連情報

Windows 用 Deployment ウィザード [793 ページ]
Linux/UNIX および MAC OS X の Deployment ウィザード [801 ページ]
暗号化の配備 [870 ページ]

1.23.8.1 .NET クライアントの配備

SQL Anywhere .NET データプロバイダを使用する ADO.NET アプリケーションは、いくつかのプロバイダコンポーネントを必要とします。

配備する SQL Anywhere .NET データプロバイダコンポーネントを決定する最も単純な方法は、*Deployment* ウィザードを使用することです。

SQL Anywhere .NET データプロバイダコンポーネントを配備する場合、次の情報を使用してクライアントコンピュータにインストールします。

- Microsoft Visual Studio が実行されていないことを確認します。
- SetupVSPackage ツールを使用して SQL Anywhere .NET データプロバイダアセンブリをインストールします。Windows 7 以降のシステムでは、SetupVSPackage の実行に管理者権限が必要です。コマンドプロンプトで SetupVSPackage を実行する場合、コマンドプロンプトに管理者権限があることを確認します。SetupVSPackage は、グローバルアセンブリキャッシュ (GAC)、Windows Microsoft.NET `machine.config` ファイル、Microsoft Visual Studio 統合をアップグレードします。

  > **注記**
  >
  > .NET データプロバイダをインストールすると、16.0 と 12.0 などの主な初期バージョンを含め、プロバイダの全バージョンで Microsoft Visual Studio 統合が影響を受けます。Microsoft Visual Studio 統合で使用できるのは、.NET データプロバイダの 1 バージョンのみです。

- .NET 2.0/3.x のみの場合、*%SQLANY17%*`¥Assembly¥v3.5¥SetupVSPackage.exe /install` を実行します。グローバルアセンブリキャッシュ (GAC) は、.NET プロバイダアセンブリのバージョン 3.5 を含めるように更新されています。Windows Microsoft.NET バージョン 2 `machine.config` ファイルは、`Sap.Data.SQLAnywhere.v3.5` でプロバイダ不変名 Sap.Data.SQLAnywhere に更新されます。
- .NET のその他すべてのバージョンで、*%SQLANY17%*`¥Assembly¥v4.5¥SetupVSPackage.exe /install` を実行します。デフォルトでは、このツールは Entity Framework 5 サポート付きでバージョン 4.5 プロバイダを登録します。`/version` オプションを使用すると、バージョンを指定できます (省略形: `/v`)。Entity Framework 5 サポート付きでバージョン 4.5 プロバイダをインストールする場合は `/v 4.5` を使用します。Entity Framework 6 サポート付きでバージョン 4.5 プロバイダをインストールする場合は `/v EF6` を使用します。グローバルアセンブリキャッシュ (GAC) には、.NET プロバイダアセンブリの全バージョンが含まれます。Windows Microsoft.NET バージョン 4 `machine.config` ファイルは、選択したプロバイダでプロバイダ不変名 Sap.Data.SQLAnywhere に更新されます。
- デフォルトでは、SetupVSPackage は SQL Anywhere のレジストリ設定を使用して .NET アセンブリを検索します。`salocation` オプションを使用すると、SQL Anywhere のインストール場所を指定できます。

  ```
  %SQLANY17%¥Assembly¥v4.5¥SetupVSPackage.exe /install /salocation %SQLANY17%
  ```

  `salocation` の省略形は `sal` です。

Microsoft SQL Server 2008 以降がシステムにインストールされている場合、SetupVSPackage は、また、`MSSqlToSA.xml` および `SAToMSSql10.xml` という 2 つのファイルを Microsoft SQL Server `DTS¥MappingFiles` フォルダにインストールします。

SetupVSPackage では、`SSDLToSA`*17*`.tt` を Microsoft Visual Studio 2010 以降のディレクトリにインストールします。SSDLToSA16.tt は、Entity Data Model のデータベーススキーマ DDL を生成するために使用されます。Entity Data Model のデータベーススキーマ DDL を生成する場合、ユーザはこのファイルに DDL 生成プロパティを設定する必要があります。

独自のインストールを作成するには、一部のファイルをエンドユーザに配備する必要があります。メッセージファイルは複数あり、それぞれが異なる言語をサポートしています。異なる言語のサポートをインストールするには、それらの言語のリソースファイルを追加してください。[LL] を言語コード (**en**、**de**、**ja** など) に置き換えます。

各 .NET クライアントコンピュータには、次のものが必要です。

.NET が動作するインストール環境

Microsoft .NET アセンブリとファイルの再配布に関する指示については、Microsoft から入手できます。ここでは、その詳細は説明しません。

.NET Framework 2.0/3.0/3.5 用 SQL Anywhere .NET Data Provider

SQL Anywhere インストールは、このプロバイダのアセンブリを SQL Anywhere インストールディレクトリの `Assembly¥V3.5` サブディレクトリに配置します。その他のファイルは、SQL Anywhere インストールディレクトリのオペレーティングシステムに対応するバイナリディレクトリに置かれます (例: `bin32`、`bin64`)。次のファイルは必須です。

```
Sap.Data.SQLAnywhere.v3.5.dll
policy.17.0.Sap.Data.SQLAnywhere.v3.5.dll
dblg[LL]17.dll
dbicu17.dll
dbicudt17.dll
dbrsa17.dll
```

Entity Framework 5 付き .NET Framework 4 用 SQL Anywhere .NET Data Provider

SQL Anywhere のインストールでは、Entity Framework 5 サポート付きの .NET Framework バージョン 4.x 用の Windows アセンブリは SQL Anywhere インストールディレクトリの `Assembly¥.5` サブフォルダに置かれます。その他のファイルは、SQL Anywhere インストールディレクトリのオペレーティングシステムに対応するバイナリディレクトリに置かれます (例: `bin32`、`bin64`)。次のファイルは必須です。

```
Sap.Data.SQLAnywhere.v4.5.dll
policy.17.0.Sap.Data.SQLAnywhere.v4.5.dll
dblg[LL]17.dll
dbicu17.dll
dbicudt17.dll
dbrsa17.dll
```

Entity Framework 6 付き .NET Framework 4 用 SQL Anywhere .NET Data Provider

SQL Anywhere のインストールでは、Entity Framework 6 サポート付きの .NET Framework バージョン 4.x 用の Windows アセンブリは SQL Anywhere インストールディレクトリの `Assembly¥V4.5` サブフォルダに置かれます。その他のファイルは、SQL Anywhere インストールディレクトリのオペレーティングシステムに対応するバイナリディレクトリに置かれます (例: `bin32`、`bin64`)。次のファイルは必須です。

```
Sap.Data.SQLAnywhere.EF6.dll
policy.17.0.Sap.Data.SQLAnywhere.EF6.dll
dblg[LL]17.dll
dbicu17.dll
dbicudt17.dll
dbrsa17.dll
```

ポリシーファイルを使用すると、アプリケーションが構築されたプロバイダのバージョンを上書きできます。プロバイダへの更新がリリースされるたびに、ポリシーファイルは更新されます。

ClickOnce を導入すると、.NET アプリケーションを Web サーバまたはネットワークファイル共有にパブリッシュし、簡単にインストールできるようにすることができます。

このセクションの内容:

ClickOnce と .NET データプロバイダのアンマネージコード DLL [809 ページ]
ClickOnce を導入すると、.NET アプリケーションを Web サーバまたはネットワークファイル共有にパブリッシュし、簡単にインストールできるようにすることができます。Microsoft Visual Studio は、ClickOnce テクノロジを使用して配備されたアプリケーションのパブリッシュと更新をサポートします。

SQL Anywhere.NET データプロバイダ削除 [809 ページ]
現在のバージョンの SQL Anywhere .NET データプロバイダをアンインストールするには、実行しなければならない手順がいくつかあります。

関連情報

Windows 用 Deployment ウィザード [793 ページ]

1.23.8.1.1 ClickOnce と .NET データプロバイダのアンマネージコード DLL

ClickOnce を導入すると、.NET アプリケーションを Web サーバまたはネットワークファイル共有にパブリッシュし、簡単にインストールできるようにすることができます。Microsoft Visual Studio は、ClickOnce テクノロジを使用して配備されたアプリケーションのパブリッシュと更新をサポートします。

SQL Anywhere .NET データプロバイダのアンマネージコード部分の配備を支援するために、配備する .NET アプリケーションに DLL をリソースとして追加する方法を説明するサンプル配備アプリケーションが提供されます。このサンプルのソースコードは、*%SQLANYSAMP17%*¥SQLAnywhere¥ADO.NET¥DeployUtility フォルダにあります。

Build.cmd

このバッチファイルは、SQL Anywhere .NET データプロバイダのアンマネージ DLL をリソースとしてアプリケーションの実行ファイルに追加する方法を示します。特に、32 ビットの DLL、dblgen*17*.dll、dbicu*17*.dll、dbicudt*17*.dll、dbrsa*17*.dll をリソースとして .NET アプリケーションに追加する方法を示します。この技術を拡張して、64 ビットバージョンの DLL を含めるようにすることができます。

Program.cs

このサンプルプログラムは、アプリケーションに付加されたリソースとして DLL にアクセスし、SQL Anywhere .NET データプロバイダで使用できるようになるディレクトリに書き込む方法を示します。ここで示す技術は、32 ビットと 64 ビットの DLL を処理するように設定できます。

1.23.8.1.2 SQL Anywhere.NET データプロバイダ削除

現在のバージョンの SQL Anywhere .NET データプロバイダをアンインストールするには、実行しなければならない手順がいくつかあります。

SQL Anywhere .NET データプロバイダアセンブリを案インストールする場合は、次の情報を使用します。

> **注記**
>
> .NET データプロバイダをアンインストールすると、16.0 と 12.0 などの主な初期バージョンを含め、プロバイダの全バージョンで Microsoft Visual Studio 統合が影響を受けます。Microsoft Visual Studio 統合で使用できるのは、.NET データプロバイダの 1 バージョンのみです。

- Microsoft Visual Studio が実行されていないことを確認します。
- SetupVSPackage ツールを使用して SQL Anywhere .NET データプロバイダアセンブリをアンインストールします。Windows 7 以降のシステムでは、SetupVSPackage の実行に管理者権限が必要です。コマンドプロンプトで SetupVSPackage を実行する場合、コマンドプロンプトに管理者権限があることを確認します。
- .NET 2.0/3.x のみの場合、*%SQLANY17%*`¥Assembly¥v3.5¥SetupVSPackage.exe /uninstall` を実行します。
- 他のバージョンの場合、*%SQLANY17%*`¥Assembly¥v4.5¥SetupVSPackage.exe /uninstall` を実行します。

/uninstall の省略形は /u です。

17.0.4 バージョンの SQL Anywhere .NET データプロバイダのすべてのビルドをアンインストールするには、次の作業を実行します。

- Microsoft Visual Studio が実行されていないことを確認します。
- SetupVSPackage ツールを使用して SQL Anywhere .NET データプロバイダアセンブリのすべてのビルドをアンインストールします。Windows 7 以降のシステムでは、SetupVSPackage の実行に管理者権限が必要です。コマンドプロンプトで SetupVSPackage を実行する場合、コマンドプロンプトに管理者権限があることを確認します。
- .NET 2.0/3.x のみの場合、*%SQLANY17%*`¥Assembly¥v3.5¥SetupVSPackage.exe /uninstallall` を実行します。
- 他のバージョンの場合、*%SQLANY17%*`¥Assembly¥v4.5¥SetupVSPackage.exe /uninstallall` を実行します。

`/uninstallall` の省略形は `/ua` です。

1.23.8.2 OLE DB と ADO クライアントの開発

SQL Anywhere OLE DB プロバイダを使用する OLE DB および ADO アプリケーションは、いくつかのプロバイダコンポーネントを必要とします。

OLE DB クライアントライブラリを配備する最も簡単な方法は、*Deployment ウィザード*を使用することです。独自のインストールを作成するには、必要なファイルをエンドユーザに配備する必要があります。

各クライアントシステムは、次の OLE DB コンポーネントを持っている必要があります。

OLE DB が動作するインストール環境

OLE DB ファイルとファイルの再配布に関する指示については、Microsoft から入手できます。Windows クライアントには、Microsoft MDAC 2.7 以降を使用してください。

OLE DB プロバイダ

次の表には、OLE DB プロバイダに必要なファイルを示しています。Windows では、32 ビットバージョンと 64 ビットバージョンのファイルがあります。一部の Windows プラットフォームの場合、プロバイダ DLL が 2 つあります。2 つ目の DLL (`dboledba`*17*`.dll`) は、スキーマサポートの提供に使用される支援 DLL です。

説明	Windows
OLE DB プロバイダライブラリ	`dboledb`*17*`.dll`
OLE DB スキーマアシストライブラリ	`dboledba`*17*`.dll`

説明	Windows
[接続] ウィンドウ	dbcon*17*.dll
文字セット変換	dbicu*17*.dll
文字セット変換データ	dbicudt*17*.dll
言語リソースライブラリ	dblg[LL]*17*.dll
暗号化サポート	dbrsa*17*.dll
昇格操作エージェント	dbelevate*17*.exe (Windows 7 以降のみ)

上記の表には、指定が [LL] であるファイルが示されています。メッセージファイルは複数あり、それぞれが異なる言語をサポートしています。異なる言語のサポートをインストールするには、それらの言語のリソースファイルを追加してください。[LL] を言語コード (**en**、**de**、**ja** など) に置き換えます。

OLE DB プロバイダには、複数のレジストリエントリが必要です。レジストリエントリを作成するには、regsvr32 ユーティリティを使用して dboledb*17*.dll および dboledba*17*.dll の各 DLL を自己登録します。

Windows 7 以降では、DLL を登録または登録解除するときに必要な権限の昇格をサポートする SQL Anywhere 昇格操作エージェントを含める必要があります。このファイルは、OLE DB プロバイダのインストール手順またはアンインストール手順の一部でのみ必要です。

1.23.8.3 OLE DB プロバイダのカスタマイズ

プロバイダの独自のバージョンを用意して、OLE DB プロバイダをカスタマイズできます。

OLE DB プロバイダをインストールする際に、プロバイダに内蔵の自己登録機能を使用して Microsoft Windows レジストリが変更されます。通常、OLE DB プロバイダの登録には regsvr32 ツールが使用されます。

一般的な OLE DB 接続文字列では、コンポーネントの 1 つは Provider 属性になります。OLE DB プロバイダを使用することを示すには、プロバイダの名前を指定します。次に Microsoft Visual Basic の場合の例を示します。

```
connectString = "Provider=SAOLEDB;DSN=SQL Anywhere 17 Demo;PWD="+myPwd
```

ADO か OLE DB または両方を使用する場合には、プロバイダを名前で参照する方法が他にも存在します。次に示すのは C++ の例で、プロバイダ名の他に使用バージョンも指定しています。

```
hr = db.Open(_T("SAOLEDB.17"), &dbinit);
```

プロバイダ名は、レジストリで検索されます。使用コンピュータシステムのレジストリを確認すると、HKEY_CLASSES_ROOT に SAOLEDB のエントリが見つかります。

```
[HKEY_CLASSES_ROOT¥SAOLEDB]
@="SQL Anywhere OLE DB provider"
```

そこには、プロバイダに対して 2 つのサブキーがあり、クラス識別子 (Clsid) と現在のバージョン (CurVer) が指定されています。次に例を示します。

```
[HKEY_CLASSES_ROOT¥SAOLEDB¥Clsid]
@="{41dfe9f7-db91-11d2-8c43-006008d26a6f}"
```

```
[HKEY_CLASSES_ROOT¥SAOLEDB¥CurVer]
@="SAOLEDB.17"
```

同様のエントリが他にもいくつかあります。これらは、OLE DB プロバイダの特定のインスタンスを識別するために使用されます。レジストリで HKEY_CLASSES_ROOT¥CLSID の下の Clsid を探し、そのサブキーを見ると、エントリの 1 つでプロバイダ DLL のロケーションが指定されていることがわかります。

```
[HKEY_CLASSES_ROOT¥CLSID¥
{41dfe9f3-db91-11d2-8c43-006008d26a6f}¥
InprocServer32]
@="c:¥¥sa17¥¥bin64¥¥dboledb17.dll"
"ThreadingModel"="Both"
```

ただし、このモデルにはリスクがあります。SQL Anywhere ソフトウェアをシステムからアンインストールすると、OLE DB プロバイダのエントリがレジストリから削除され、プロバイダ DLL がハードドライブから削除されます。これにより、削除するプロバイダに依存しているアプリケーションが動作しなくなります。

同様に、複数のアプリケーションが同じ OLE DB プロバイダを使用している場合、そのプロバイダをインストールするたびに、共通レジストリ設定が上書きされます。つまり、アプリケーションの動作に必要なプロバイダのバージョンが、別の (新しいまたは古い) バージョンのプロバイダに上書きされる可能性があります。このような理由から、OLE DB プロバイダはカスタマイズ可能となっています。

このセクションの内容:

OLE DB プロバイダのカスタマイズ [812 ページ]
ユニークな GUID のセットを生成し、プロバイダに命名し、ユニークな DLL の名前を選択して、アプリケーションと一緒に配備できるユニークな OLE DB プロバイダを作成します。

1.23.8.3.1 OLE DB プロバイダのカスタマイズ

ユニークな GUID のセットを生成し、プロバイダに命名し、ユニークな DLL の名前を選択して、アプリケーションと一緒に配備できるユニークな OLE DB プロバイダを作成します。

手順

1. 次のデータをテキストエディタにコピーし、.reg ファイルとして保存します。

```
REGEDIT4
; Special registry entries for a private OLE DB provider.
[HKEY_CLASSES_ROOT¥myoledb17]
@="Custom SQL Anywhere OLE DB provider 17"
[HKEY_CLASSES_ROOT¥myoledb17¥Clsid] @="{GUID1}"
; Data1 of the following GUID must be 3 greater than the
; previous, for example, 41dfe9f3 + 3 => 41dfe9ee.
[HKEY_CLASSES_ROOT¥myoledba17]
@="Custom SQL Anywhere OLE DB provider 17"
[HKEY_CLASSES_ROOT¥myoledba17¥Clsid] @="{GUID4}"
; Current version (or version independent prog ID)
; entries (what you get when you have "SQLAny"
; instead of "SQLAny.17")
```

```
[HKEY_CLASSES_ROOT¥SQLAny]
@="SQL Anywhere OLE DB provider"
[HKEY_CLASSES_ROOT¥SQLAny¥Clsid]
@="{GUID1}"
[HKEY_CLASSES_ROOT¥SQLAny¥CurVer]
@="SQLAny.17"
[HKEY_CLASSES_ROOT¥SQLAnyEnum]
@="SQL Anywhere OLE DB provider Enumerator"
[HKEY_CLASSES_ROOT¥SQLAnyEnum¥Clsid]
@="{GUID2}" [HKEY_CLASSES_ROOT¥SQLAnyEnum¥CurVer]
@="SQLAnyEnum.17" [HKEY_CLASSES_ROOT¥SQLAnyErrorLookup]
@="SQL Anywhere OLE DB provider Extended Error Support"
[HKEY_CLASSES_ROOT¥SQLAnyErrorLookup¥Clsid]
@="{GUID3}"
[HKEY_CLASSES_ROOT¥SQLAnyErrorLookup¥CurVer]
@="SQLAnyErrorLookup.17"
[HKEY_CLASSES_ROOT¥SQLAnyA]
@="SQL Anywhere OLE DB provider Assist"
[HKEY_CLASSES_ROOT¥SQLAnyA¥Clsid]
@="{GUID4}"
[HKEY_CLASSES_ROOT¥SQLAnyA¥CurVer]
@="SQLAnyA.17"
; Standard entries (Provider=SQLAny.17)
[HKEY_CLASSES_ROOT¥SQLAny.17]
@="SAP SQL Anywhere OLE DB provider 17"
[HKEY_CLASSES_ROOT¥SQLAny.17¥Clsid]
@="{GUID1}"
[HKEY_CLASSES_ROOT¥SQLAnyEnum.17]
@="SAP SQL Anywhere OLE DB provider Enumerator 17"
[HKEY_CLASSES_ROOT¥SQLAnyEnum.17¥Clsid]
@="{GUID2}"
[HKEY_CLASSES_ROOT¥SQLAnyErrorLookup.17]
@="SAP SQL Anywhere OLE DB provider Extended Error Support 17"
[HKEY_CLASSES_ROOT¥SQLAnyErrorLookup.17¥Clsid]
@="{GUID3}"
[HKEY_CLASSES_ROOT¥SQLAnyA.17]
@="SAP SQL Anywhere OLE DB provider Assist 17"
[HKEY_CLASSES_ROOT¥SQLAnyA.17¥Clsid]
@="{GUID4}"
; SQLAny (Provider=SQLAny.17)
[HKEY_CLASSES_ROOT¥CLSID¥{GUID1}]
@="SQLAny.17"
"OLEDB_SERVICES"=dword:ffffffff
[HKEY_CLASSES_ROOT¥CLSID¥{GUID1}¥ExtendedErrors]
@="Extended Error Service"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID1}¥ExtendedErrors¥{GUID3}]
@="SAP SQL Anywhere OLE DB provider Error Lookup"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID1}¥InprocServer32]
@="d:¥¥mypath¥¥bin32¥¥myoledb17.dll"
"ThreadingModel"="Both"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID1}¥OLE DB Provider]
@="SAP SQL Anywhere OLE DB provider 17"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID1}¥ProgID]
@="SQLAny.17"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID1}¥VersionIndependentProgID]
@="SQLAny"
; SQLAnyErrorLookup
[HKEY_CLASSES_ROOT¥CLSID¥{GUID3}]
@="SAP SQL Anywhere OLE DB provider Error Lookup 17.0"
@="SQLAnyErrorLookup.17"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID3}¥InprocServer32]
@="d:¥¥mypath¥¥bin32¥¥myoledb17.dll"
"ThreadingModel"="Both"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID3}¥ProgID]
@="SQLAnyErrorLookup.17"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID3}¥VersionIndependentProgID]
@="SQLAnyErrorLookup"
```

```
; SQLAnyEnum [HKEY_CLASSES_ROOT¥CLSID¥{GUID2}]
@="SQLAnyEnum.17"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID2}¥InprocServer32]
@="d:¥¥mypath¥¥bin32¥¥myoledb17.dll"
"ThreadingModel"="Both"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID2}¥OLE DB Enumerator]
@="SAP SQL Anywhere OLE DB provider Enumerator"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID2}¥ProgID]
@="SQLAnyEnum.17"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID2}¥VersionIndependentProgID]
@="SQLAnyEnum"
; SQLAnyA [HKEY_CLASSES_ROOT¥CLSID¥{GUID4}]
@="SQLAnyA.17"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID4}¥InprocServer32]
@="d:¥¥mypath¥¥bin32¥¥myoledba17.dll"
"ThreadingModel"="Both"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID4}¥ProgID]
@="SQLAnyA.17"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID4}¥VersionIndependentProgID]
@="SQLAnyA"
```

レジストリ値の名前の大文字と小文字は区別されます。

2. Microsoft Visual Studio の uuidgen ユーティリティを使用して、4 つの連続する UUID (GUID) を作成します。

   たとえば、コマンドプロンプトで次のコマンドを実行します。

   ```
   uuidgen -n4 -s -x >oledbguids.txt
   ```

   4 つの UUID または GUID は、次の順番で割り当てます。

   a. Provider クラス ID (下の表の GUID1)。
   b. Enum クラス ID (下の表の GUID2)。
   c. ErrorLookup クラス ID (下の表の GUID3)。
   d. Provider Assist クラス ID (下の表の GUID4)。

   これらは連続していて、コマンドラインの-x オプションが行うことであることが重要です。

   各 GUID は次のように表示されます。

名称	GUID
GUID1	41dfe9f3-db92-11d2-8c43-006008d26a6f
GUID2	41dfe9f4-db92-11d2-8c43-006008d26a6f
GUID3	41dfe9f5-db92-11d2-8c43-006008d26a6f
GUID4	41dfe9f6-db92-11d2-8c43-006008d26a6f

> **注記**
>
> 増分している GUID の最初の部分 (41dfe9f3 など)。

3. エディタの検索/置換機能を使用して、テキストに出現する GUID1、GUID2、GUID3、GUID4 を対応する GUID に変更します (たとえば、uuidgen によって生成された GUID が上の表のような場合は、GUID1 を 41dfe9f3-db92-11d2-8c43-006008d26a6f に置き換えます)。
4. アプリケーションで接続文字列に使用するプロバイダ名を選択します (例: Provider=SQLAny)。

次の名前は SQL Anywhere によって予約されており、プロバイダ名として使用できません。

バージョン 10 以降	バージョン 9 以前
SAOLEDB	ASAProv
SAErrorLookup	ASAErrorLookup
SAEnum	ASAEnum
SAOLEDBA	ASAProvA

5. エディタの検索/置換機能を使用して、出現するすべての SQLAny という文字列を選択したプロバイダ名に変更します。置換対象には、長い文字列の一部として SQLAny が出現する箇所も含まれます (例: SQLAnyEnum)。

   プロバイダ名を Acme としたとします。この場合に HKEY_CLASSES_ROOT レジストリに表示される名前を、比較しやすいように SQL Anywhere の名前とともに次の表に示します。

SQL Anywhere	カスタムプロバイダ
SAOLEDB	Acme
SAErrorLookup	AcmeErrorLookup
SAEnum	AcmeEnum
SAOLEDBA	AcmeA

6. SQL Anywhere OLE DB プロバイダ DLL (`dboledb17.dll` と `dboledba17.dll`) のコピーを別の名前で作成します。

```
copy dboledb17.dll myoledb17.dll
copy dboledba17.dll myoledba17.dll
```

   スクリプトにより、選択した DLL 名に基づく特別なレジストリキーが作成されます。ここでは、名前が標準の DLL 名 (`dboledb17.dll`、`dboledba17.dll` など) と異なることが重要です。プロバイダ DLL 名を `myoledb17` とすると、プロバイダは HKEY_CLASSES_ROOT でこの名前のレジストリエントリを探します。プロバイダスキーマ支援 DLL の場合も同様です。DLL 名を `myoledba17` とすると、プロバイダは HKEY_CLASSES_ROOT でこの名前のレジストリエントリを探します。ユニークで、他人から選択されにくい名前にすることが重要です。次にその例を示します。

選択された DLL 名	対応する HKEY_CLASSES_ROOT¥name
`myoledb17.dll`	HKEY_CLASSES_ROOT¥myoledb17
`myoledba17.dll`	HKEY_CLASSES_ROOT¥myoledba17
`acmeOledb.dll`	HKEY_CLASSES_ROOT¥acmeOledb
`acmeOledba.dll`	HKEY_CLASSES_ROOT¥acmeOledba
`SAcustom.dll`	HKEY_CLASSES_ROOT¥SAcustom
`SAcustomA.dll`	HKEY_CLASSES_ROOT¥SAcustomA

7. エディタの検索/置換機能を使用して、レジストリスクリプトに出現するすべての `myoledb17` と `myoledba17` を選択した 2 つの DLL 名に変更します。
8. エディタの検索/置換機能を使用して、レジストリスクリプトに出現する `d:¥¥mypath¥¥bin32¥¥` をすべて DLL のインストールロケーションに変更します。1 つのバックスラッシュを表すのにバックスラッシュを 2 つ重ねる必要があることに注意してください。この手順は、アプリケーションのインストール時にカスタマイズが必要です。

9. レジストリスクリプトをディスクに保存して、実行します。
10. 新しいプロバイダを試します。新しいプロバイダ名を使用するよう ADO または OLE DB アプリケーションを必ず変更してください。

結果

カスタム OLE DB プロバイダが設定されました。

次のステップ

カスタム OLE DB プロバイダをアプリケーションと一緒に配備します。

1.23.8.4 ODBC クライアントの配備

SQL Anywhere ODBC ドライバを使用するアプリケーションは複数のコンポーネントを必要とします。

ODBC クライアントを配備する最も簡単な方法は、*Deployment* ウィザードを使用することです。

各 ODBC クライアントコンピュータには、次のものが必要です。

ODBC ドライバマネージャ

Microsoft は、Windows オペレーティングシステム用の ODBC ドライバマネージャを提供しています。SQL Anywhere には、Linux、UNIX、および Mac OS X 用の ODBC ドライバマネージャが含まれています。ODBC アプリケーションはドライバマネージャがなくても実行できますが、ODBC ドライバマネージャを使用できるプラットフォームではこの方法はお奨めできません。

接続情報

クライアントアプリケーションが、サーバの接続に必要な情報にアクセスできるようにしてください。この情報は、通常は ODBC データソースに含まれています。

ODBC ドライバ

SQL Anywhere ODBC ドライバをインストールする必要があります。

このセクションの内容:

ODBC ドライバに必要なファイル [817 ページ]
SQL Anywhere ODBC ドライバは複数のコンポーネントを必要とします。

ODBC ドライバの設定 [820 ページ]
ODBC ドライバファイルをディスクにコピーすることに加え、インストールプログラムで一連のレジストリエントリを作成して ODBC ドライバを適切にインストールする必要もあります。

ODBC アプリケーションのデータベース接続情報 [821 ページ]
ODBC のクライアント接続情報は、通常は ODBC データソースとして配備されます。

関連情報

Windows 用 Deployment ウィザード [793 ページ]
Linux/UNIX および MAC OS X の Deployment ウィザード [801 ページ]

1.23.8.4.1 ODBC ドライバに必要なファイル

SQL Anywhere ODBC ドライバは複数のコンポーネントを必要とします。

次の表は、SQL Anywhere ODBC ドライバを動作させるために必要なファイルを示します。大半のオペレーティングシステムプラットフォームでは、32 ビットバージョンと 64 ビットバージョンのファイルがあります。

Linux、UNIX、および Mac OS X プラットフォーム用のマルチスレッドバージョンの ODBC ドライバは "MT" で示されています。

プラットフォーム	必要なファイル
Windows	dbodbc*17*.dll dbcon*17*.dll dbicu*17*.dll dbicudt*17*.dll dblg[LL]*17*.dll dbrsa*17*.dll dbelevate*17*.exe
Linux、Solaris、HP-UX	libdbodbc*17*.so.1 libdbodbc*17*_n.so.1 libdbodm*17*.so.1 libdbtasks*17*.so.1 libdbicu*17*.so.1 libdbicudt*17*.so.1 libdbrsa*17*.so.1 dblg[LL]*17*.res

プラットフォーム	必要なファイル
Linux、Solaris、HP-UX MT	libdbodbc*17*.so.1 libdbodbc*17*_r.so.1 libdbodm*17*.so.1 libdbtasks*17*_r.so.1 libdbicu*17*_r.so.1 libdbicudt*17*.so.1 libdbrsa*17*_r.so.1 dblg[LL]*17*.res
AIX	libdbodbc*17*.so libdbodbc*17*_n.so libdbodm*17*.so libdbtasks*17*.so libdbicu*17*.so libdbicudt*17*.so libdbrsa*17*.so dblg[LL]*17*.res
AIX MT	libdbodbc*17*.so libdbodbc*17*_r.so libdbodm*17*.so libdbtasks*17*_r.so libdbicu*17*_r.so libdbicudt*17*.so libdbrsa*17*_r.so dblg[LL]*17*.res

プラットフォーム	必要なファイル
Mac OS X	dbodbc*17*.bundle libdbodbc*17*.dylib libdbodbc*17*_n.dylib libdbodm*17*.dylib libdbtasks*17*.dylib libdbicu*17*.dylib libdbicudt*17*.dylib libdbrsa*17*.dylib dblg[LL]*17*.res
Mac OS X MT	dbodbc*17*_r.bundle libdbodbc*17*.dylib libdbodbc*17*_r.dylib libdbodm*17*.dylib libdbtasks*17*_r.dylib libdbicu*17*_r.dylib libdbicudt*17*.dylib libdbrsa*17*_r.dylib dblg[LL]*17*.res

注記

- Linux および Solaris プラットフォームでは、.so.1 ファイルへのリンクを作成します。リンク名はファイル名からバージョンを表すサフィックス ".1" を除いた名前になります。
- Linux、UNIX、および Mac OS X プラットフォームには、マルチスレッド (MT) バージョンの ODBC ドライバがあります。ファイル名には "_r" サフィックスが付きます。アプリケーションで必要な場合は、これらのファイルを配備します。
- Windows の場合、ドライバマネージャはオペレーティングシステムに含まれています。SQL Anywhere には、Linux、UNIX、および Mac OS X 用のドライバマネージャがあります。このファイルには libdbodm*17* で始まる名前が付いています。
- Windows 7 以降では、ODBC ドライバを登録または登録解除するために必要な権限の昇格をサポートする SQL Anywhere 昇格操作エージェント (dbelevate*17*.exe) を含める必要があります。このファイルは、ODBC ドライバのインストール手順またはアンインストール手順の一部でのみ必要です。
- 言語リソースライブラリファイルも含めてください。上記の表には、指定が [LL] であるファイルが示されています。メッセージファイルは複数あり、それぞれが異なる言語をサポートしています。異なる言語のサポートをインストールするには、それらの言語のリソースファイルを追加してください。[LL] を言語コード (**en**、**de**、**ja** など) に置き換えます。

- Windows の場合、エンドユーザが独自のデータソースを作成する場合や、データベースに接続するときにユーザ ID とパスワードを入力する必要がある場合、またはその他の理由で [接続] ウィンドウが表示される場合は、*SQL Anywhere の ODBC 設定*および *SQL Anywhere への接続*ウィンドウのサポートコード (`dbcon17.dll`) が必要です。

1.23.8.4.2 ODBC ドライバの設定

ODBC ドライバファイルをディスクにコピーすることに加え、インストールプログラムで一連のレジストリエントリを作成して ODBC ドライバを適切にインストールする必要もあります。

Windows

SQL Anywhere のインストーラは Windows レジストリを変更し、ODBC ドライバを識別して設定します。インストールプログラムをエンドユーザ用に構築する場合は、同じレジストリ設定を行ってください。

最も簡単な方法は、ODBC ドライバの自己登録機能を使用することです。regsvr32 ユーティリティを Windows で使用します。64 ビットバージョンの Windows では、64 ビットバージョンと 32 ビットバージョンの両方の ODBC ドライバを登録できます。ODBC ドライバの自己登録機能を使用すると、適切なレジストリエントリを確実に作成できます。

インストールする SQL Anywhere ODBC ドライバの 32 ビットおよび 64 ビットバージョンにカスタム名を付けます。これにより、regsvr32 を使用して SQL Anywhere ODBC ドライバの複数の独立したコピーを容易にインストールおよび登録できるようになります。また、別のアプリケーションインストールによって SQL Anywhere ODBC ドライバが登録された場合にレジストリ設定が上書きされることを回避できます。

32 ビットバージョンおよび 64 ビットバージョンの ODBC ドライバの名前をカスタマイズするには、コマンドプロンプトを開き、次のようにドライバファイル名前を変更します。`custom-name` は、会社名などのわかりやすい文字列とします。

```
ren dbodbc17.dll "dbodbc17custom-name.dll"
```

32 ビットおよび 64 ビットバージョンの両方でこれを実行します。ファイル名を変更する場合は、`dbodbc17` プレフィクスを保持します (必須)。

次のようなコマンドを使用して ODBC ドライバを登録します。

```
regsvr32 "dbodbc17custom-name.dll"
```

32 ビットおよび 64 ビットバージョンの両方でこれを実行します。

regedit ユーティリティを使用して、ODBC ドライバで作成されたレジストリエントリを調べることができます。

この SQL Anywhere ODBC ドライバのコピーは、次のレジストリキーの一連のレジストリ値によってシステムで識別されます。

```
HKEY_LOCAL_MACHINE¥Software¥ODBC¥ODBCINST.INI¥SQL Anywhere 17 - custom-name
```

64 ビット Windows の場合のサンプル値を次に示します。

名前	タイプ	データ
Driver	REG_SZ	install-dir ¥bin64¥dbodbc*17*custom-name.dll
設定	REG_SZ	install-dir ¥bin64¥dbodbc*17*custom-name.dll

次のキーにもレジストリエントリがあります。

```
HKEY_LOCAL_MACHINE¥Software¥ODBC¥ODBCINST.INI¥ODBC Drivers
```

エントリを次に示します。

名前	タイプ	データ
SQL Anywhere17 - custom-name	REG_SZ	Installed

64 ビットの Windows

64 ビットの Windows では、32 ビット ODBC ドライバのレジストリエントリ ("*SQL Anywhere17* - custom-name" と "ODBC Drivers") は次のキーに含まれています。

```
HKEY_LOCAL_MACHINE¥Software¥Wow6432Node¥ODBC¥ODBCINST.INI
```

これらのエントリを表示するには、64 ビットバージョンの regedit を使用する必要があります。64 ビットの Windows で Wow6432Node が見つからない場合は、おそらく 32 ビットバージョンの regedit を使用しています。

サードパーティ製 ODBC ドライバ

Windows 以外のオペレーティングシステムでサードパーティ製の ODBC ドライバを使用する場合は、ODBC ドライバの設定方法についてはそのドライバのマニュアルを参照してください。

1.23.8.4.3 ODBC アプリケーションのデータベース接続情報

ODBC のクライアント接続情報は、通常は ODBC データソースとして配備されます。

ODBC データソースは、次のいずれかの方法で配備できます。

プログラムを使用

データソースの記述をエンドユーザのレジストリまたは ODBC 初期化ファイルに追加します。これには dbdsn ユーティリティが便利です。

手動

エンドユーザに手順を示して、各自のコンピュータに適切なデータソースを作成できるようにします。

Windows プラットフォームでは、dbdsn ユーティリティを使用して手動でデータソースを作成できます。または、ODBC データソースアドミニストレータを使用して [ユーザ DSN] タブまたは [システム DSN] タブでデータソースを作成します。SQL Anywhere ODBC ドライバは、設定を入力するための設定ウィンドウを表示します。データソースの設定には、データベースファイルのロケーション、データベースサーバの名前、起動パラメータとその他のオプションが含まれます。コンテキスト別のヘルプは、*[ヘルプ]* ボタンを押すことでアクセスできます。ヘルプテキストは、`sacshelp17.chm` を使ってローカルに提供されます。このファイルが見つからない場合は、システムのデフォルトインターネットブラウザを使用して DocCommentXchange からインターネット経由でヘルプが提供されます。

UNIX プラットフォームでは、dbdsn ユーティリティを使用して手動でデータソースを作成できます。データソースの設定には、データベースファイルのロケーション、データベースサーバの名前、起動パラメータ、その他のオプションが含まれます。

どちらの方法でも知る必要がある情報を次に示します。

データソースのタイプ (Windows)

データソースは 3 種類あります。ユーザデータソース、システムデータソース、ファイルデータソースです。

1. ユーザデータソース定義は、レジストリの一部として保存され、システムに現在ログインしている特定のユーザ用の設定を含んでいます。
2. これに対し、システムデータソースは、すべてのユーザと Windows のサービスで使用でき、ユーザがシステムにログインしているかどうかに関係なく稼働します。MyApp というシステムデータソースが正しく設定されている場合、ODBC 接続文字列に DSN=MyApp と指定することでどのユーザでもその ODBC 設定ソースを使用することができます。
3. ファイルデータソースはレジストリには保存されず、ディスクに格納されます。ファイルデータソースを使用するには、接続文字列に FileDSN 接続パラメータを指定する必要があります。ファイルデータソースのデフォルトロケーションは、*HKEY_CURRENT_USER¥Software¥ODBC¥odbc.ini¥ODBC File DSN¥DefaultDSNDir* レジストリエントリで指定するか、このレジストリが定義されていない場合は *HKEY_LOCAL_MACHINE¥Software¥ODBC¥odbc.ini¥ODBC File DSN¥DefaultDSNDir* レジストリエントリで指定します。ファイルデータソースが他の場所にある場合は、FileDSN 接続パラメータにパスを含めて、検索に役立てることができます。

データソースのレジストリエントリ (Windows)

ユーザデータソースとシステムデータソースの定義は、Windows レジストリに格納されます。データソース定義の正しいレジストリエントリを確実に作成する最も簡単な方法は、dbdsn ユーティリティを使用して作成することです。

64 ビットの Windows でのシステムデータソースの場合、2 セットのレジストリエントリがあります。1 つは 64 ビットのアプリケーション用で、もう 1 つは 32 ビットのアプリケーション用です。問題を避けるために、2 つのセットを作成する必要があります。64 ビットレジストリキーを作成する場合は 64 ビットバージョンの dbdsn を使用し、32 ビットレジストリキーを作成する場合は 32 ビットバージョンの dbdsn を使用します。

このユーティリティを使用しない場合は、一連のレジストリ値を特定のレジストリキーに作成する必要があります。

ユーザデータソースの場合のレジストリキーを次に示します。

```
HKEY_CURRENT_USER¥Software¥ODBC¥ODBC.INI¥user-data-source-name
```

32 ビットの Windows でのシステムデータソースの場合のレジストリキーを次に示します。

```
HKEY_LOCAL_MACHINE¥Software¥ODBC¥ODBC.INI¥system-data-source-name
```

64 ビットの Windows でのシステムデータソースの場合、2 つのレジストリキーがあります。1 つは 64 ビットのアプリケーション用で、もう 1 つは 32 ビットのアプリケーション用です。64 ビットのレジストリキーは次のとおりです。

```
HKEY_LOCAL_MACHINE¥Software¥ODBC¥ODBC.INI¥system-data-source-name
```

32 ビットのレジストリキーは次のとおりです。

```
HKEY_LOCAL_MACHINE¥Software¥Wow6432Node¥ODBC¥ODBC.INI¥system-data-source-name
```

キーには一連のレジストリ値が含まれ、それぞれが 1 つの接続パラメータに対応します (Driver 文字列を除く)。Microsoft ODBC データソースアドミニストレータまたは dbdsn ユーティリティを使用してデータソースが作成されるときに、Driver 文字列は Microsoft ODBC ドライバマネージャによって自動的にレジストリに追加されます。たとえば、32 ビットの Windows では、SQL Anywhere 17 Demo システムのデータソース名 (DSN) に対応する SQL Anywhere 17 Demo キーには次の設定が含まれます。

値の名前	値のタイプ	値データ
Autostop	String	YES
DatabaseFile	String	*C:¥Users¥Public¥Documents¥SQL Anywhere 17¥Samples¥demo.db*
Description	String	SQL Anywhere17 サンプルデータベース
Driver	String	*C:¥Program Files¥SQL Anywhere 17*¥bin32¥dbodbc*17*.dll
ServerName	String	demo17
StartLine	String	*C:¥Program Files¥SQL Anywhere 17*¥bin32¥*dbeng17*.exe
UID	String	DBA

注記

配備されたアプリケーションの接続文字列には、ServerName パラメータを含めます。これにより、コンピュータで複数の SQL Anywhere データベースサーバが実行されている場合に、アプリケーションが確実に正しいサーバに接続できるため、タイミングに依存する接続エラーを防ぐことができます。

これらのエントリにおいて、64 ビットの Windows の場合は、bin32 が bin64 になります。

64 ビットの Windows では、64 ビットアプリケーションと 32 ビットアプリケーションの両方で共有されるユーザデータソースのレジストリエントリは 1 つしかありません。

ODBC データソースリスト (Windows)

さらに、データソース名をレジストリ内のデータソースのリストにも追加する必要があります。

ユーザデータソースの場合は、次のキーを使用します。

```
HKEY_CURRENT_USER¥Software¥ODBC¥ODBC.INI¥ODBC Data Sources
```

32 ビットの Windows でのシステムデータソースの場合は、次のキーを使用します。

```
HKEY_LOCAL_MACHINE¥Software¥ODBC¥ODBC.INI¥ODBC Data Sources
```

64 ビットの Windows でのシステムデータソースの場合、2 つのレジストリキーがあります。1 つは 64 ビットのアプリケーション用で、もう 1 つは 32 ビットのアプリケーション用です。64 ビットのレジストリキーは次のとおりです。

```
HKEY_LOCAL_MACHINE¥Software¥Wow6432Node¥ODBC¥ODBC.INI¥ODBC Data Sources
```

32 ビットのレジストリキーは次のとおりです。

```
HKEY_LOCAL_MACHINE¥Software¥ODBC¥ODBC.INI¥ODBC Data Sources
```

この値によって、各データソースは ODBC ドライバと対応させられます。値の名前はデータソース名で、値データは ODBC ドライバ名です。たとえば、SQL Anywhere によってインストールされたシステムデータソースは、SQL Anywhere 17 Demo という名前で、次のような値を持ちます。

値の名前	値のタイプ	値データ
SQL Anywhere 17 Demo	String	*SQL Anywhere17*

> ⚠ 警告
>
> ODBC データソース名の設定は簡単に表示されてしまいます。ユーザデータソースの設定には、ユーザ ID とパスワードのように機密性のあるデータベース設定を含めることもできます。これらの設定は、レジストリに普通のテキストとして保存され、Windows レジストリエディタ `regedit.exe` または `regedt32.exe` を使用して表示できます。これらのエディタは、Microsoft からオペレーティングシステムとともに提供されています。パスワードを暗号化したり、ユーザが接続するときにパスワードの入力を要求するように選択することもできます。

必須およびオプションの接続パラメータ

ODBC 接続文字列内のデータソース名は次のようにして調べることができます。

```
DSN=ADataSourceName
```

Windows では、DSN パラメータを接続文字列に指定すると、Windows レジストリ内のユーザデータソース定義が検索された後にシステムデータソースが検索されます。ファイルデータソースは、FileDSN 接続パラメータが ODBC 接続文字列に指定された場合にだけ検索されます。

次の表は、データソースが存在し、そのデータソースが DSN パラメータや FileDSN パラメータとしてアプリケーションの接続文字列に含まれている場合の、ユーザと開発者に対する影響を示します。

データソースの状態	接続文字列に指定する追加情報	ユーザが指定する情報
ODBC ドライバの名前とロケーション、データベースファイルまたはデータベースサーバの名前、起動パラメータ、ユーザ ID とパスワードを含む	追加情報なし	追加情報なし
ODBC ドライバの名前とロケーション、データベースファイルまたはデータベースサーバの名前、起動パラメータを含む	追加情報なし	ユーザ ID とパスワード (ODBC データソースに指定がない場合)
ODBC ドライバの名前とロケーションのみを含む	データベースファイル名 (DBF=) とデータベースサーバ名 (SERVER=) またはそのいずれか。オプションで、Userid (UID=) や Password (PWD=) などのその他の接続パラメータを含む場合もあります。	ユーザ ID とパスワード (ODBC データソースまたは ODBC 接続文字列に指定がない場合)
存在しない	使用する ODBC ドライバ名 (Driver=)、データベース名 (DBN=)、データベースファイル名 (DBF=) とデータベースサーバ名 (SERVER=) またはそのいずれか。オプションで、Userid (UID=) や Password (PWD=) などのその他の接続パラメータを含む場合もあります。	ユーザ ID とパスワード (ODBC 接続文字列に指定がない場合)

1.23.8.5 Embedded SQL クライアントの配備

Embedded SQL を使用するアプリケーションは複数のコンポーネントを必要とします。

Embedded SQL クライアントを配備する最も簡単な方法は、*Deployment* ウィザードを使用することです。

Embedded SQL クライアントの配備には、次のものが含まれます。

インストールされるファイル

各クライアントコンピュータには、Embedded SQL クライアントアプリケーションに必要なファイルを準備します。

接続情報

クライアントアプリケーションが、サーバの接続に必要な情報にアクセスできるようにしてください。この情報は、ODBC データソースに含めることができます。

このセクションの内容:

Embedded SQL クライアントに必要なファイル [826 ページ]
Embedded SQL アプリケーションはいくつかのコンポーネントを必要とします。

Embedded SQL アプリケーションのデータベース接続情報 [827 ページ]
Embedded SQL 接続情報は、さまざまな方法で配備できます。

関連情報

Windows 用 Deployment ウィザード [793 ページ]
Linux/UNIX および MAC OS X の Deployment ウィザード [801 ページ]

1.23.8.5.1 Embedded SQL クライアントに必要なファイル

Embedded SQL アプリケーションはいくつかのコンポーネントを必要とします。

次の表は、Embedded SQL クライアントに必要なファイルを示します。

説明	Windows	Linux/UNIX	Mac OS X
インタフェースライブラリ	dblib*17*.dll	libdblib*17*_r.so	libdblib*17*_r.dylib
標準の暗号化サポート	dbrsa*17*.dll	libdbrsa*17*_r.so	libdbrsa*17*_r.dylib
別途ライセンスの暗号化サポート	dbfips*17*.dll、libeay32.dll、ssleay32.dll、msvcr90.dll (64 ビットバージョンは msvcr100.dll)	libdbfips*17*_r.so、libcrypto.so、libssl.so	N/A
スレッドサポートライブラリ	N/A	libdbtasks*17*_r.so	libdbtasks*17*_r.dylib
言語リソースライブラリ	dblg[LL]*17*.dll	dblg[LL]*17*.res	dblg[LL]*17*.res
[接続] ウィンドウ	dbcon*17*.dll	N/A	N/A

注記

- 言語リソースライブラリファイルも含めてください。上記の表には、指定が [LL] であるファイルが示されています。メッセージファイルは複数あり、それぞれが異なる言語をサポートしています。異なる言語のサポートをインストールするには、それらの言語のリソースファイルを追加してください。[LL] を言語コード (**en**、**de**、**ja** など) に置き換えます。
- Linux/UNIX 上で動作する非マルチスレッドアプリケーションについては、libdblib*17*.so と libdbtasks*17*.so を使用してください。
- Mac OS X 上で動作する非マルチスレッドアプリケーションには、libdblib*17*.dylib と libdbtasks*17*.dylib を使用してください。
- クライアントアプリケーションで暗号化を使用する場合は、適切な暗号化サポートも含めてください。
- クライアントアプリケーションが ODBC データソースを使用して接続パラメータを保持する場合は、エンドユーザは動作している ODBC インストール環境を必要とします。ODBC を配備するための手順は、Microsoft の『ODBC SDK』に含まれています。
- Windows の場合、エンドユーザが独自のデータソースを作成する場合や、データベースに接続するときにユーザ ID とパスワードを入力する必要がある場合、またはその他の理由で [接続] ウィンドウが必要な場合は、*[SQL Anywhere の ODBC 設定]* および *[SQL Anywhere への接続]* ウィンドウのサポートコード (dbcon*17*.dll) が必要です。

関連情報

ODBC クライアントの配備 [816 ページ]

1.23.8.5.2 Embedded SQL アプリケーションのデータベース接続情報

Embedded SQL 接続情報は、さまざまな方法で配備できます。

次のいずれかの方法を使用できます。

手動

エンドユーザに手順を示して、各自のコンピュータに適切なデータソースを作成できるようにします。

ファイル

アプリケーションで読むことのできるフォーマットで接続情報を含むファイルを配布します。

ODBC データソース

ODBC データソースを使用して接続情報を保持できます。

1.23.8.6 JDBC クライアントの配備

JDBC を使用するアプリケーションは複数のコンポーネントを必要とします。

JDBC を使用するには、Java Runtime Environment (JRE) をインストールしてください。バージョン 1.8.0 以降 (JRE 8) を推奨します。

JRE に加えて、各 JDBC クライアントには SQL Anywhere JDBC ドライバまたは jConnect ドライバが必要です。

SQL Anywhere JDBC ドライバ

SQL Anywhere JDBC ドライバを配備するには、SQL Anywhere がインストールされる `java` フォルダにある `sajdbc4.jar` を配備する必要があります。このファイルはアプリケーションのクラスパス内に表示されます。

さらに、JDBC ドライバのサポートファイルをいくつか配備します。次の表は、さまざまなプラットフォームで必要なファイルを示します。これらのファイルは単一のディレクトリに置いてください。SQL Anywhere のインストールでは、これらのファイルすべてが SQL Anywhere インストールディレクトリのバイナリサブディレクトリに置かれます (例: `bin32`、`bin64`、`lib32`、`lib64`)。32 ビットまたは 64 ビットのファイルの選択は、インストールされている Java VM のビット数によって決まります。

JDBC ドライバサポートファイルは、すべてのプラットフォームでマルチスレッド対応です。

プラットフォーム	必要なファイル
Windows	dbjdbc*17*.dll dbicu*17*.dll dbicudt*17*.dll dblg[LL]*17*.dll dbrsa*17*.dll
Linux、Solaris、HP-UX	libdbjdbc*17*.so.1 libdbtasks*17*_r.so.1 libdbicu*17*_r.so.1 libdbicudt*17*.so.1 libdbrsa*17*_r.so.1 dblg[LL]*17*.res
AIX	libdbjdbc*17*.so libdbtasks*17*_r.so libdbicu*17*_r.so libdbicudt*17*.so libdbrsa*17*_r.so dblg[LL]*17*.res
Mac OS X	libdbjdbc*17*.dylib libdbtasks*17*_r.dylib libdbicu*17*_r.dylib libdbicudt*17*.dylib libdbrsa*17*_r.dylib dblg[LL]*17*.res

- Linux および Solaris プラットフォームでは、.so.1 ファイルへのリンクを作成します。リンク名はファイル名からバージョンを表すサフィックス ".1" を除いた名前にしてください。
- 言語リソースライブラリファイルも含めてください。上記の表には、指定が [LL] であるファイルが示されています。メッセージファイルは複数あり、それぞれが異なる言語をサポートしています。異なる言語のサポートをインストールするには、それらの言語のリソースファイルを追加してください。[LL] を言語コード (**en**、**de**、**ja** など) に置き換えます。

jConnect JDBC ドライバ

jConnect JDBC ドライバを配備するには、次のファイルを配備します。

- jConnect ドライバのファイル。
- Open Client または jConnect ベースの TDS クライアントを使用する場合は、接続パスワードをクリアテキストとして送信するか、暗号化された形式で送信するかを選択できます。暗号化された形式の場合は、TDS のパスワード暗号化のハンドシェイクを実行することで送信されます。ハンドシェイクでは、プライベートキー/パブリックキーの暗号化を使用します。RSA プライベートキー/パブリックキーのペアを生成し、暗号化されたパスワードの復号化をサポートする機能は、特別なライブラリに含まれています。このライブラリファイルは、SQL Anywhere サーバのシステムパス内に含める必要があります。Windows の場合、これは `dbrsakp17.dll` というファイルです。64 ビットバージョンと 32 ビットバージョンの DLL があります。Linux および UNIX 環境では、このファイルは `libdbrsakp17.so` という共有ライブラリです。Mac OS X では、このファイルは `libdbrsakp17.dylib` という共有ライブラリです。この機能を使用しない場合、このファイルは不要です。

JDBC データベース接続の URL

Java アプリケーションは、データベースに接続するために URL を必要とします。この URL は、ドライバ、使用するコンピュータ、データベースサーバが受信するポートを指定します。

関連情報

jConnect ドライバ接続文字列 [221 ページ]
jConnect for JDBC

1.23.8.7 PHP クライアントの配備

SQL Anywhere PHP 拡張を使用するアプリケーションは複数のコンポーネントを必要とします。

PHP 拡張を配備するには、ターゲットプラットフォームに次のコンポーネントをインストールしてください。

- 使用しているプラットフォーム用の PHP 5 バイナリ。http://scn.sap.com/docs/DOC-40537 から入手できます。Windows プラットフォームの場合、PHP 拡張ではスレッド対応の PHP を使用する必要があります。
- Apache HTTP サーバなどの Web サーバ (PHP スクリプトを Web サーバ内で実行する場合)。
  データベースサーバは、Web サーバと同じコンピュータでも異なるコンピュータでも実行できます。
- 共有オブジェクトまたはライブラリのサポート。

次の表は、PHP クライアントに必要なファイルを示します。

説明	Windows	Linux	Mac OS X
PHP のインストール (サードパーティ)	php.exe	php	php
PHP 拡張	php-5.x.0_sqlanywhere.dll	php-5.x.0_sqlanywhere_r.so	ソースコードからビルド
言語リソースライブラリ	dblg[LL]*17*.dll	dblg[LL]*17*.res	dblg[LL]*17*.res
[接続] ウィンドウ	dbcon*17*.dll	N/A	N/A
SQL Anywhere C API のランタイム	dbcapi.dll	libdbcapi_r.so	libdbcapi_r.dylib
DBLIB (スレッド)	dblib*17*.dll	libdblib*17*_r.so	libdblib*17*_r.dylib
スレッドサポートライブラリ	N/A	libdbtasks*17*_r.so	libdbtasks*17*_r.dylib
ICU ライブラリ	dbicu*17*.dll	libdbicu*17*_r.so[2]	libdbicu*17*_r.dylib[2]
ICU データライブラリ	dbicudt*17*.dll	libdbicudt*17*.so[2]	libdbicudt*17*.dylib[2]
標準の暗号化サポート	dbrsa*17*.dll	libdbrsa*17*_r.so	libdbrsa*17*_r.dylib
別途ライセンスの暗号化サポート	dbfips*17*.dll、libeay32.dll、ssleay32.dll、msvcr90.dll (64 ビットバージョンは msvcr100.dll)	libdbfips*17*_r.so、libcrypto.so、libssl.so	N/A

注記

- 言語リソースライブラリファイルも含めてください。上記の表には、指定が [LL] であるファイルが示されています。メッセージファイルは複数あり、それぞれが異なる言語をサポートしています。異なる言語のサポートをインストールするには、それらの言語のリソースファイルを追加してください。[LL] を言語コード (**en**、**de**、**ja** など) に置き換えます。
- クライアントアプリケーションで暗号化を使用する場合は、適切な暗号化サポートも含めてください。
- クライアントアプリケーションが ODBC データソースを使用して接続パラメータを保持する場合は、エンドユーザは動作している ODBC インストール環境を必要とします。ODBC を配備するための手順は、Microsoft の ODBC SDK に含まれています。
- Windows の場合、エンドユーザが独自のデータソースを作成する場合や、データベースに接続するときにユーザ ID とパスワードを入力する必要がある場合、またはその他の理由で [接続] ウィンドウが必要な場合は、*SQL Anywhere の ODBC 設定*および *SQL Anywhere* への接続ウィンドウのサポートコード (dbcon*17*.dll) が必要です。
- PHP 拡張のインストールの詳細については、次のトピックを参照してください。

このセクションの内容:

PHP 拡張 [831 ページ]
PHP アプリケーションによっては、スレッド対応の PHP 拡張が必要なものも、スレッド非対応の PHP 拡張が必要なものもあります。

Windows での PHP 拡張のインストール [832 ページ]

Windows で SQL Anywhere PHP 拡張を使用するには、拡張 DLL を PHP インストールにコピーします。オプションとして、拡張を自動的にロードするためのエントリを PHP 初期化ファイルに追加すると、各スクリプトで拡張を手動でロードする必要がなくなります。

Linux での PHP 拡張のインストール [833 ページ]
Linux で SQL Anywhere PHP 拡張を使用するには、拡張共有オブジェクトを PHP インストールにコピーします。オプションとして、拡張を自動的にロードするためのエントリを PHP 初期化ファイルに追加すると、各スクリプトで拡張を手動でロードする必要がなくなります。

UNIX と Mac OS X での PHP 拡張 [835 ページ]
その他のバージョンの UNIX または Mac OS X で SQL Anywhere PHP 拡張を使用するには、SQL Anywhere のインストール環境の `sdk/php` サブディレクトリにインストールされたソースコードから PHP 拡張をビルドする必要があります。

SQL Anywhere PHP 拡張の設定 [835 ページ]
SQL Anywhere PHP 拡張の動作は、PHP 初期化ファイル `php.ini` で値を設定することによって制御します。

関連情報

ODBC クライアントの配備 [816 ページ]
SQL Anywhere and PHP について
SAP SQL Anywhere PHP モジュール
PHP

1.23.8.7.1 PHP 拡張

PHP アプリケーションによっては、スレッド対応の PHP 拡張が必要なものも、スレッド非対応の PHP 拡張が必要なものもあります。

Windows では、スレッド対応 PHP ではスレッド対応の PHP 拡張を使用する必要がありますスレッド対応の拡張ファイル名は次のようになります。ここで、5.x は PHP のバージョンを表します。

```
php-5.x.0_sqlanywhere.dll
```

Windows では、非スレッド対応の PHP は、マルチプロセスで非マルチスレッドの Web サーバ (Apache 1.x など) で使用できます。非スレッド対応の PHP 拡張ファイル名は次のようになります。ここで、5.x は PHP のバージョンを表します。

```
php-5.x.0_sqlanywhere_nts.dll
```

Linux では、スレッド対応および非スレッド対応の PHP 拡張があります。

Apache 2.x を使用する場合は、スレッド対応の拡張を使用します。スレッド対応の拡張ファイル名は次のようになります。ここで、5.x は PHP のバージョンを表します。

```
php-5.x.0_sqlanywhere_r.so
```

CGI バージョンの PHP を使用している場合、または Apache 1.x を使用している場合は、非スレッド対応の拡張を使用します。非スレッド対応の拡張ファイル名は次のようになります。ここで、5.x は PHP のバージョンを表します。

```
php-5.x.0_sqlanywhere.so
```

関連情報

SQL Anywhere PHP モジュール

1.23.8.7.2 Windows での PHP 拡張のインストール

Windows で SQL Anywhere PHP 拡張を使用するには、拡張 DLL を PHP インストールにコピーします。オプションとして、拡張を自動的にロードするためのエントリを PHP 初期化ファイルに追加すると、各スクリプトで拡張を手動でロードする必要がなくなります。

前提条件

PHP 5.4 以降をインストールしている必要があります。

手順

1. Windows 用のビルド済みバージョンの PHP 拡張は、http://scn.sap.com/docs/DOC-40537 からダウンロードします。
2. PHP インストールディレクトリにある `php.ini` ファイルを探して、テキストエディタで開きます。
3. extension_dir エントリのロケーションを指定する行を探します。
4. エントリが存在しない場合、extension_dir エントリを作成し、より強固なシステムセキュリティのために分離ディレクトリをポイントします。
5. `php-5.x.0_sqlanywhere.dll` ファイルを `php.ini` ファイルの extension_dir エントリで指定したディレクトリにコピーします。

   文字列 `5.x` は、インストールした PHP バージョンに対応します。

   選択する DLL は、インストールされている PHP のバージョンとそのビットバージョン (32 ビットまたは 64 ビット) で決まります。スレッド非対応のバージョン (`php-5.x.0_sqlanywhere_nts.dll`) も含まれます。

   使用している PHP のバージョンがビルド済み拡張よりも新しい場合は、拡張をビルドする必要があります。PHP 拡張のソースコードは、ソフトウェアインストール環境の `sdk¥php` サブディレクトリにインストールされています。

6. PHP 拡張を自動的にロードするために、次のような行を `php.ini` ファイルの Dynamic Extensions セクションに追加します。

   ```
   extension=php-5.x.0_sqlanywhere.dll
   ```

   この手順を実行しないと、スクリプトに要求されるたびに PHP 拡張を手動でロードする必要があります。
7. `php.ini` を保存して閉じます。
8. 32 ビットバージョンの PHP 拡張では、`Bin32` ディレクトリがパスに含まれている必要があります。64 ビットバージョンの PHP 拡張では、`Bin64` ディレクトリがパスに含まれている必要があります。

結果

PHP 拡張は Windows 環境でサポートされており、使用可能です。

関連情報

SQL Anywhere PHP 拡張の設定 [835 ページ]
PHP テストページの作成と実行 [453 ページ]

1.23.8.7.3 Linux での PHP 拡張のインストール

Linux で SQL Anywhere PHP 拡張を使用するには、拡張共有オブジェクトを PHP インストールにコピーします。オプションとして、拡張を自動的にロードするためのエントリを PHP 初期化ファイルに追加すると、各スクリプトで拡張を手動でロードする必要がなくなります。

前提条件

PHP 5.4 以降をインストールしている必要があります。

手順

1. Linux 用のビルド済みバージョンの PHP 拡張は、http://scn.sap.com/docs/DOC-40537 からダウンロードします。
2. PHP インストールディレクトリにある `php.ini` ファイルを探して、テキストエディタで開きます。
3. extension_dir エントリのロケーションを指定する行を探します。

4. エントリが存在しない場合、extension_dir エントリを作成し、より強固なシステムセキュリティのために分離ディレクトリをポイントします。
5. `php-5.x.0_sqlanywhere_r.so` ファイルを `php.ini` ファイルの extension_dir エントリで指定したディレクトリにコピーします。

   文字列 `5.x` は、インストールした PHP のバージョン番号を表します。`_r` 接尾辞はスレッド対応バージョンの PHP 拡張を示します。

   選択する共有オブジェクトは、インストールされている PHP のバージョンとそのビットバージョン (32 ビットまたは 64 ビット) で決まります。スレッド非対応のバージョン (`php-5.x.0_sqlanywhere.so`) も含まれます。

   使用している PHP のバージョンがビルド済み拡張よりも新しい場合は、拡張をビルドする必要があります。PHP 拡張のソースコードは、ソフトウェアインストール環境の `sdk/php` サブディレクトリにインストールされています。
6. PHP 拡張を自動的にロードするために、次のような行を `php.ini` ファイルの Dynamic Extensions セクションに追加します。

   ```
   extension=php-5.x.0_sqlanywhere_r.so
   ```

   この手順を実行しないと、スクリプトに要求されるたびに PHP 拡張を手動でロードする必要があります。
7. `php.ini` を保存して閉じます。
8. PHP の実行環境が SQL Anywhere のために設定されているかを確認します。

   使用しているシェルに応じて、Web サーバ環境の設定スクリプトを編集し、適切なコマンドを追加することで、SQL Anywhere のインストールディレクトリから SQL Anywhere の設定スクリプトのコマンドを実行します。

オプション	アクション
sh、ksh、または bash	`. /bin32/sa_config.sh`
csh または tcsh	`source /bin32/sa_config.csh`

   32 ビットバージョンの PHP 拡張では、`bin32` ディレクトリがパスに含まれている必要があります。64 ビットバージョンの PHP 拡張では、`bin64` ディレクトリがパスに含まれている必要があります。

   この行を挿入する設定ファイルは、他の Web サーバや Linux ディストリビューションでは異なります。特定のディストリビューションにおける Apache サーバの例を次に示します。

   RedHat/Fedora/CentOS

   /etc/sysconfig/httpd

   Debian/Ubuntu

   /etc/apache2/envvars

> **注記**
>
> Web サーバは、環境設定を編集した後に再起動する必要があります。

結果

PHP 拡張は Linux 環境でサポートされており、使用可能です。

関連情報

PHP 拡張 [831 ページ]
SQL Anywhere PHP 拡張の設定 [835 ページ]
PHP テストページの作成と実行 [453 ページ]

1.23.8.7.4 UNIX と Mac OS X での PHP 拡張

その他のバージョンの UNIX または Mac OS X で SQL Anywhere PHP 拡張を使用するには、SQL Anywhere のインストール環境の `sdk/php` サブディレクトリにインストールされたソースコードから PHP 拡張をビルドする必要があります。

> **注記**
>
> Mac OS X 10.11 上で、SQLANY_API_DLL 環境変数を `libdbcapi_r.dylib` へのフルパスに設定します。

関連情報

UNIX と Mac OS X での PHP 拡張のビルド [461 ページ]

1.23.8.7.5 SQL Anywhere PHP 拡張の設定

SQL Anywhere PHP 拡張の動作は、PHP 初期化ファイル `php.ini` で値を設定することによって制御します。

次のエントリがサポートされます。

extension

PHP が起動するたびに、PHP が SQL Anywhere PHP 拡張を自動的にロードします。このエントリの PHP 初期化ファイルへの追加は任意ですが、追加しない場合、作成する各スクリプトは、この拡張がロードされるように数行のコードから開始する必要があります。Windows プラットフォームの場合、使用するエントリは次のとおりです。

```
extension=php-5.x.0_sqlanywhere.dll
```

Linux プラットフォームの場合、次のいずれかのエントリを使用します。2 番目のエントリはスレッドに対応しています。

```
extension=php-5.x.0_sqlanywhere.so
```

```
extension=php-5.x.0_sqlanywhere_r.so
```

これらのエントリで、5.x は使用している PHP のバージョンを表します。

PHP の起動時に SQL Anywhere PHP 拡張を自動的にロードしない場合は、記述する各スクリプトの先頭に次のコードを追加してください。このコードによって、SQL Anywhere PHP 拡張がロードされるようになります。

```
# Ensure that the SQL Anywhere PHP extension is loaded
if( !extension_loaded('sqlanywhere') ) {
    # Find out which version of PHP is running
    $version = explode('.',phpversion());
    $xy = $version[0].'.'.$version[1].'.0';
    $extension_name = 'php-'.$xy.'_sqlanywhere';
    if( strtoupper(substr(PHP_OS, 0, 3) == 'WIN' )) {
        $extension_ext = '.dll';
    } else {
        $extension_ext = '.so';
    }
    dl( $extension_name.$extension_ext );
}
```

allow_persistent

On に設定すると永続的接続が有効になります。Off に設定すると永続的接続が無効になります。デフォルト値は On です。

```
sqlanywhere.allow_persistent=On
```

max_persistent

永続的接続の最大数を設定します。デフォルト値は -1 で、制限がありません。

```
sqlanywhere.max_persistent=-1
```

max_connections

SQL Anywhere PHP 拡張によって同時に開くことができる接続の最大数を設定します。デフォルト値は -1 で、制限がありません。

```
sqlanywhere.max_connections=-1
```

auto_commit

データベースサーバで自動的にコミット操作を実行するかどうかを指定します。On に設定すると、各文を実行した直後にコミットを実行します。Off に設定すると、必要に応じて sasql_commit または sasql_rollback 関数によってトランザクションを手動で終了する必要があります。デフォルト値は On です。

```
sqlanywhere.auto_commit=On
```

row_counts

On に設定すると、操作によって影響を受けるローの正確な数を返します。Off に設定すると、予測値を返します。デフォルト値は Off です。

```
sqlanywhere.row_counts=Off
```

verbose_errors

On に設定すると、冗長エラーおよび警告を返します。それ以外の場合、詳細なエラー情報を取得するには、sasql_error または sasql_errorcode 関数を呼び出す必要があります。デフォルト値は On です。

```
sqlanywhere.verbose_errors=On
```

関連情報

sasql_set_option [494 ページ]

1.23.8.8 Open Client アプリケーションの配備

Open Client アプリケーションを配備するには、各クライアントコンピュータに SAP Open Client 製品が必要です。

Open Client または jConnect ベースの TDS クライアントを使用する場合は、接続パスワードをクリアテキストとして送信するか、暗号化された形式で送信するかを選択できます。暗号化された形式の場合は、TDS のパスワード暗号化のハンドシェイクを実行することで送信されます。ハンドシェイクでは、プライベートキー/パブリックキーの暗号化を使用します。RSA プライベートキー/パブリックキーのペアを生成し、暗号化されたパスワードの復号化をサポートする機能は、特別なライブラリに含まれています。このライブラリファイルは、SQL Anywhere サーバのシステムパス内に含める必要があります。Windows の場合、これは `dbrsakp17.dll` というファイルです。64 ビットバージョンと 32 ビットバージョンの DLL があります。Linux および UNIX 環境では、このファイルは `libdbrsakp17.so` という共有ライブラリです。Mac OS X では、このファイルは `libdbrsakp17_r.dylib` という共有ライブラリです。この機能を使用しない場合、このファイルは不要です。

Open Client クライアント用の接続情報は interfaces ファイルに保持されます。

1.23.9 管理ツールの配備

ライセンス契約に従って、Interactive SQL および *SQL Central* を含む一連の管理ツールを配備できます。

管理ツールを配備する最も簡単な方法は、*Deployment* ウィザードを使用することです。

初期化ファイルによって管理ツールの配備を簡略化できます。管理ツール (*SQL Central* および Interactive SQL) のランチャ実行プログラムごとに、対応する `.ini` ファイルがあります。初期化ファイルを使用すると、JAR ファイルのロケーションに関するレジストリエントリや固定ディレクトリ構造が不要になります。これらの `ini` ファイルは、実行プログラムファイルと同じファイル名で同じディレクトリ内にあります。

dbisql.ini

これは、Interactive SQL の初期化ファイルの名前です。

scjview.ini

これは、*SQL Central* の初期化ファイルの名前です。

初期化ファイルには、データベース管理ツールをロードする方法の詳細が含まれます。たとえば、初期化ファイルには次の行を含めることができます。

JRE_DIRECTORY=path

必要な JRE のロケーション。JRE_DIRECTORY の指定は必須です。
VM_ARGUMENTS=any-required-VM-arguments

VM 引数はセミコロン (;) で区切って指定します。空白が含まれているパス値は、引用符で囲んでください。VM 引数を確認するには、管理ツールの `-batch` オプションを使用し、対応するバッチファイルを作成して調べます。たとえば Windows の場合、コマンドプロンプトから、*SQL Central* を `scjview -batch` オプションで起動すると `scjview.bat` が生成され、Interactive SQL を `dbisql -batch` オプションで起動すると `dbisql.bat` が生成されます。
VM_ARGUMENTS の指定はオプションです。
JAR_PATHS=path1;path2;...

プログラムの JAR ファイルを含むディレクトリのリスト。値はセミコロン (;) で区切って指定します。JAR_PATHS の指定はオプションです。
ADDITIONAL_CLASSPATH=path1;path2;...

クラスパス値はセミコロン (;) で区切って指定します。ADDITIONAL_CLASSPATH の指定はオプションです。
LIBRARY_PATHS=path1;path2;...

DLL/共有オブジェクトのパス。値はセミコロン (;) で区切って指定します。LIBRARY_PATHS の指定はオプションです。
APPLICATION_ARGUMENTS=arg1;arg2;...

アプリケーションの引数。値はセミコロン (;) で区切って指定します。アプリケーション引数を確認するには、管理ツールの `-batch` オプションを使用し、対応するバッチファイルを作成して調べます。たとえば Windows の場合、コマンドプロンプトから、*SQL Central* を `scjview -batch` オプションで起動すると `scjview.bat` が生成され、Interactive SQL を `dbisql -batch` オプションで起動すると `dbisql.bat` が生成されます。APPLICATION_ARGUMENTS の指定はオプションです。

次に示すのは、*SQL Central* のサンプルの初期化ファイルの内容です。

```
JRE_DIRECTORY=c:¥jdk1.8.0¥jre
VM_ARGUMENTS=
JAR_PATHS=c:¥scj¥jars
ADDITIONAL_CLASSPATH=
LIBRARY_PATHS=c:¥scj¥bin
APPLICATION_ARGUMENTS=-screpository=C:¥Users¥Public¥Documents¥SQL Central 17;-
installdir=c:¥scj
```

この例では、JRE のコピーが `c:¥jdk1.8.0¥jre` にあることを前提としています。また、`jsyblib1700` などの *SQL Central* の実行プログラムと共有ライブラリ (DLL) が `c:¥scj¥bin` に格納されています。JAR ファイルは `c:¥scj¥jars` に格納されています。

> **注記**
>
> アプリケーションを配備するときは、dbinit ユーティリティを使用してデータベースを作成するために、パーソナルデータベースサーバ (dbeng17) が必要です。パーソナルデータベースサーバは、その他のデータベースサーバが実行されていない場合にローカルコンピュータで *SQL Central* からデータベースを作成する場合にも必要です。

1. Windows での管理ツールの配備 [839 ページ]
   Deployment ウィザードを使用せずに Windows に Interactive SQL (dbisql) と *SQL Central* (SQL Anywhere、Mobile Link、Ultra Light プラグインを含みます) をインストールするには、一定の手順に従う必要があります。この項は、これらの管理ツールのインストーラの作成を望むユーザを対象としています。
2. Linux、Solaris、Mac OS X における管理ツールの配備 [847 ページ]

Linux、Solaris、Mac OS X のコンピュータで Interactive SQL (dbisql) と *SQL Central* (SQL Anywhere と Mobile Link のプラグインを含む) をインストールするには、一定の手順に従う必要があります。この項は、これらの管理ツールのインストーラの作成を望むユーザを対象としています。

3. 管理ツールの設定 [855 ページ]
   `OEM.ini` という名前の初期化ファイルを使用すると、どの機能が表示されるか、有効になるかを指定できます。
4. dbisqlc の配備 [857 ページ]
   配備したアプリケーションがクエリの実行やツールのスクリプト記述を必要とし、リソースに制限のあるコンピュータで実行されている場合には、Interactive SQL (dbisql) ではなく dbisqlc 実行プログラムを配備できます。
5. SQL Anywhere モニタ Production Edition について [858 ページ]
   Production Edition は、配備と運用の目的で使用されます。別個にインストールされ、サービスとして実行されます。Production Edition には、すべての機能が搭載された SQL Anywhere インストールが含まれます。

関連情報

Windows 用 Deployment ウィザード [793 ページ]
Linux/UNIX および MAC OS X の Deployment ウィザード [801 ページ]

1.23.9.1 Windows での管理ツールの配備

Deployment ウィザードを使用せずに Windows に Interactive SQL (dbisql) と *SQL Central* (SQL Anywhere、Mobile Link、Ultra Light プラグインを含みます) をインストールするには、一定の手順に従う必要があります。この項は、これらの管理ツールのインストーラの作成を望むユーザを対象としています。

この情報は、サポートされているすべての Windows プラットフォームに適用されます。ここで説明する手順は、バージョン 17.0.4 固有のもので、前後のバージョンには適用できません。

> **注記**
>
> ファイルの再配布はライセンス契約に従います。このマニュアル内の記述は、ライセンス契約のどの条項にも優先しません。配備について検討する前にライセンス契約を確認してください。

始める前に

REGEDIT アプリケーションを含む Windows レジストリに関する知識が必要です。レジストリ値の名前は大文字と小文字を区別します。

> **警告**
>
> Windows レジストリの変更は危険です。変更はユーザ自身の責任で行ってください。レジストリを変更する前に、システムをバックアップしてください。

このセクションの内容:

手順 1: 配備するソフトウェアを決定します [840 ページ]
要件に応じて、管理ツールをさまざまな方法でバンドルできます。

手順 2:必要なファイルをコピーします [841 ページ]
管理ツールには、特定のディレクトリ構造が必要です。

手順 3: 管理ツールを Windows に登録します [845 ページ]
管理ツールを使用するには、レジストリキーをいくつか設定する必要があります。レジストリ値の名前は大文字と小文字を区別します。

手順 4: システムパスを更新します。[846 ページ]
管理ツールを実行するには、`.exe` ファイルと `.dll` ファイルが格納されているディレクトリをパスに含めます。`c:¥sa17¥Bin32` ディレクトリまたは `c:¥sa17¥Bin64` ディレクトリをシステムパスに追加する必要があります。

手順 5: SQL Central の接続プロファイルを作成します [846 ページ]
この手順では *SQL Central* を設定します。*SQL Central* をインストールしない場合は、省略できます。

手順 6: SQL Anywhere ODBC ドライバを登録します [847 ページ]
SQL Anywhere ODBC ドライバをインストールしてから、管理ツールの JDBC ドライバでこれを使用します。

親トピック 管理ツールの配備 [837 ページ]

次のトピック: Linux、Solaris、Mac OS X における管理ツールの配備 [847 ページ]

1.23.9.1.1 手順 1: 配備するソフトウェアを決定します

要件に応じて、管理ツールをさまざまな方法でバンドルできます。

以下のソフトウェアバンドルを任意に組み合わせてインストールできます。

- Interactive SQL
- SQL Anywhere プラグインを含む *SQL Central*
- Mobile Link プラグインを含む *SQL Central*
- Ultra Light プラグインを含む *SQL Central*

上記のどのソフトウェアバンドルをインストールする場合にも、次のコンポーネントが必要です。

- SQL Anywhere ODBC ドライバ
- Java SE Runtime Environment (JRE) 8。ターゲットプラットフォームに応じて、32 ビットバージョンの JRE、64 ビットバージョンの JRE、または両方をインストールできます。
- Microsoft Windows オペレーティングシステム用の Java Access Bridge (Windows プラットフォームで管理ツールのアクセシビリティ機能を有効にする場合)。この機能を有効にすると、画面リーダーテクノロジで管理ツールを使用できるようになります。

ここで説明する手順は、これらのバンドルをどれでも (またはすべて) 競合なしでインストールできるように構成されています。

1.23.9.1.2 手順 2:必要なファイルをコピーします

管理ツールには、特定のディレクトリ構造が必要です。

ディレクトリツリーは任意のドライブのどのディレクトリにも自由に含めることができます。次の説明では、インストールフォルダの例として `c:¥sa17` を使用します。ソフトウェアは、次のレイアウトのディレクトリツリー構造にインストールしてください。

ディレクトリ	説明
`sa17`	ルートフォルダ。以下の手順では、`c:¥sa17` へのインストールを想定していますが、ディレクトリはどこに設定してもかまいません (たとえば `C:¥Program Files¥SQLAny17`)。
`sa17¥Bin32`	プログラムで使用するネイティブの 32 ビット Windows コンポーネントが保存されています。これにはアプリケーションを起動するプログラムも含まれます。
`sa17¥Bin32¥jre180`	32 ビット Java Runtime Environment
`sa17¥Bin64`	プログラムで使用するネイティブの 64 ビット Windows コンポーネントが保存されています。これにはアプリケーションを起動するプログラムも含まれます。
`sa17¥Bin64¥jre180`	64 ビット Java Runtime Environment
`sa17¥Java`	Java プログラム JAR ファイルが保存されています。

次の表は、各管理ツールと *SQL Central* プラグインに必須のファイルを示します。必要なファイルのリストを作成してから、前述したディレクトリ構造にコピーします。

テーブルには、フォルダ指定が BinXX であるファイルが示されています。これらのファイルには、32 ビットバージョンと 64 ビットバージョンが、それぞれ `Bin32` フォルダと `Bin64` フォルダにあります。32 ビットと 64 ビットの管理ツールをインストールしている場合は、それぞれのフォルダに両方の一連のファイルをインストールする必要があります。

テーブルには、指定が [LL] であるファイルが示されています。メッセージファイルは複数あり、それぞれが異なる言語をサポートしています。異なる言語のサポートをインストールするには、それらの言語のリソースファイルを含めてください。

管理ツールは JRE 1.8.0 を必要とします。特に必要のないかぎり、これより新しいパッチバージョンの JRE を代用しないでください。

32 ビットの管理ツールを配備する場合は、32 ビットバージョンの JRE ファイルを *%SQLANY17%*`¥Bin32¥jre180` ディレクトリからコピーします。サブディレクトリも含めて `jre180` ツリー全体をコピーします。

64 ビットの管理ツールを配備する場合は、64 ビットバージョンの JRE ファイルを *%SQLANY17%*`¥Bin64¥jre180` ディレクトリからコピーします。サブディレクトリも含めて `jre180` ツリー全体をコピーします。

Interactive SQL

Interactive SQL (dbisql) には次のファイルが必要です。

```
BinXX¥dbdsn.exe
BinXX¥dbelevate17.exe
BinXX¥dbicu17.dll
BinXX¥dbicudt17.dll
```

```
BinXX¥dbisql.com
BinXX¥dbisql.exe
BinXX¥dbjodbc17.dll
BinXX¥dblg[LL]17.dll
BinXX¥dblib17.dll
BinXX¥dbodbc17.dll
BinXX¥dbrsa17.dll
BinXX¥jsyblib1700.dll
BinXX¥jre180¥...
Java¥batik-all.jar
Java¥isql.jar
Java¥JComponents1700.jar
Java¥jodbc4.jar
Java¥ngdbc.jar
Java¥jsyblib1700.jar
Java¥saip17.jar
Java¥SCEditor1700.jar
Java¥xml-apis-ext.jar
```

上記のファイルパスには、"..." で終わっているものもあります。これは、サブフォルダも含めてツリー全体をコピーする必要があることを表します。32 ビットバージョンの Interactive SQL には、32 ビットバージョンの JRE ファイル (`Bin32¥jre180`) が必要です。64 ビットバージョンの Interactive SQL には、64 ビットバージョンの JRE ファイル (`Bin64¥jre180`) が必要です。

SQL Central

SQL Central (scjview) には次のファイルが必要です。

```
BinXX¥jsyblib1700.dll
BinXX¥scjview.exe
BinXX¥jre180¥...
Java¥jsyblib1700.jar
Java¥SCEditor1700.jar
Java¥sqlcentral1700.jar
```

上記のファイルパスには、"..." で終わっているものもあります。これは、サブフォルダも含めてツリー全体をコピーする必要があることを表します。32 ビットバージョンの *SQL Central* には、32 ビットバージョンの JRE ファイル (`Bin32¥jre180`) が必要です。64 ビットバージョンの *SQL Central* には、64 ビットバージョンの JRE ファイル (`Bin64¥jre180`) が必要です。

SQL Anywhere プラグインを含む *SQL Central*

SQL Central の SQL Anywhere プラグインでは、パーソナルサーバ (*dbeng17*`.exe` とサポートファイル) と次のファイルが必要です。

```
BinXX¥dbdsn.exe
BinXX¥dbicu17.dll
BinXX¥dbicudt17.dll
BinXX¥dbjodbc17.dll
BinXX¥dblg[LL]17.dll
BinXX¥dblib17.dll
BinXX¥dbodbc17.dll
BinXX¥dbput17.dll
BinXX¥dbrsa17.dll
```

```
BinXX¥dbtool17.dll
Java¥batik-all.jar
Java¥dbdiff.jar
Java¥debugger.jar
Java¥diffutils-1.2.1.jar
Java¥isql.jar
Java¥JComponents1700.jar
Java¥jodbc4.jar
Java¥ngdbc.jar
Java¥saip17.jar
Java¥saplugin.jar
Java¥salib.jar
Java¥SQLAnywhere.jpr
Java¥xml-apis-ext.jar
```

Mobile Link プラグインを含む *SQL Central*

SQL Central の Mobile Link プラグインには次のファイルが必要です。

```
BinXX¥dbdsn.exe
BinXX¥dbicu17.dll
BinXX¥dbicudt17.dll
BinXX¥dblg[LL]17.dll
BinXX¥dblib17.dll
BinXX¥dbput17.dll
BinXX¥dbrsa17.dll
BinXX¥dbtool17.dll
BinXX¥mljodbc17.dll
Java¥commons-collections-3.2.1.jar
Java¥commons-lang-2.6.jar
Java¥commons-logging-1.2.jar
Java¥isql.jar
Java¥JComponents1700.jar
Java¥jodbc4.jar
Java¥mldesign.jar
Java¥mlplugin.jar
Java¥MobiLink.jpr
Java¥ngdbc.jar
Java¥saip17.jar
Java¥salib.jar
Java¥velocity-1.7-dep.jar
```

Ultra Light プラグインを含む *SQL Central*

SQL Central の Ultra Light プラグインには次のファイルが必要です。

```
BinXX¥dbdsn.exe
BinXX¥dbicu17.dll
BinXX¥dbicudt17.dll
BinXX¥dblg[LL]17.dll
BinXX¥dblib17.dll
BinXX¥dbput17.dll
BinXX¥dbrsa17.dll
BinXX¥dbtool17.dll
BinXX¥mlcrsa17.dll
BinXX¥ulscutil17.dll
```

```
BinXX¥ulutils17.dll
Java¥batik-all.jar
Java¥isql.jar
Java¥JComponents1700.jar
Java¥jodbc4.jar
Java¥ngdbc.jar
Java¥saip17.jar
Java¥salib.jar
Java¥ulplugin.jar
Java¥UltraLite.jpr
Java¥xml-apis-ext.jar
```

外国語のメッセージとコンテキスト別のヘルプファイル

管理ツールのすべての表示テキストとコンテキスト別のヘルプは、英語からフランス語、ドイツ語、日本語、簡体字中国語に翻訳されています。各言語のリソースは別々のファイルに保存されています。英語のファイルの場合は、ファイル名に en が含まれています。フランス語のファイルも同様な名前ですが、en の代わりに fr を使用します。ドイツ語のファイル名には de、日本語のファイル名には ja、中国語のファイル名には zh がそれぞれ含まれています。

異なる言語のサポートをインストールするには、他の言語のメッセージファイルを含めてください。これらのファイルには、32 ビットバージョンと 64 ビットバージョンが、それぞれ Bin32 フォルダと Bin64 フォルダにあります。32 ビットと 64 ビットの管理ツールをインストールしている場合は、それぞれのフォルダに両方の一連のファイルをインストールする必要があります。翻訳済みのファイルは次のとおりです。

dblgen17.dll	英語
dblgde17.dll	ドイツ語
dblgfr17.dll	フランス語
dblgja17.dll	日本語
dblgzh17.dll	中国語 (簡体字)

これらのファイルは、SQL Anywhere のローカライズ版に含まれます。

アクセシビリティコンポーネント

Java Access Bridge は、Java ベースの管理ツールへのアクセシビリティを提供します (Microsoft Windows オペレーティングシステム用の Java Access Bridge は Java Web サイトからダウンロードできます)。次にアクセシビリティコンポーネントのインストールロケーションを示します。

32 ビット管理ツール

accessibility.properties	Bin32¥jre180¥lib
jaccess.jar	Bin32¥jre180¥lib¥ext
access-bridge.jar	Bin32¥jre180¥lib¥ext
JavaAccessBridge.dll	Bin32¥jre180¥bin

JAWTAccessBridge.dll	Bin32¥jre180¥bin
WindowsAccessBridge.dll	%windir%¥system32 (32-bit Windows) または %windir%¥SysWOW64 (64-bit Windows)

64 ビット管理ツール

accessibility.properties	Bin64¥jre180¥lib
jaccess.jar	Bin64¥jre180¥lib¥ext
access-bridge.jar	Bin64¥jre180¥lib¥ext
JavaAccessBridge-64.dll	Bin64¥jre180¥bin
JAWTAccessBridge-64.dll	Bin64¥jre180¥bin
WindowsAccessBridge-64.dll	%windir%¥system32

インストーラによって BinXX¥jre180¥lib フォルダに accessibility.properties ファイルが作成され、かつ次に示す *assistive_technologies* オプションがこのファイルに含まれている必要があります。

```
#
# Load the Java Access Bridge class into the JVM
#
assistive_technologies=com.sun.java.accessibility.AccessBridge
#screen_magnifier_present=true
```

SQL Anywhere インストーラはこのファイルを作成しますが、すべての行にコメントが付けられます。支援テクノロジを有効にするには、*assistive_technologies* 行をコメント解除します (# を削除すします)。

1.23.9.1.3 手順 3: 管理ツールを Windows に登録します

管理ツールを使用するには、レジストリキーをいくつか設定する必要があります。レジストリ値の名前は大文字と小文字を区別します。

管理ツールに対して以下のレジストリキーを設定します。レジストリ値の名前は大文字と小文字を区別します。

- *HKEY_LOCAL_MACHINE¥Software¥SAP¥SQL Anywhere¥17.0* で

  Location

  インストールフォルダのルートへの完全に修飾されたパス (*C:¥Program Files¥SQL Anywhere 17* など)。

管理ツールに対して以下のレジストリキーを設定する場合があります。レジストリキーは省略可能です。ただし、OS の言語が管理ツールで使用される言語と異なる場合にかぎり、設定する必要があります。レジストリキーの名前では、大文字と小文字が区別されます。

- *HKEY_LOCAL_MACHINE¥Software¥SAP¥SQL Anywhere¥17.0* で

  言語

  SQL Anywhere と管理ツールで使用する言語を表す 2 文字のコード。これは、英語 (EN)、ドイツ語 (DE)、フランス語 (FR)、日本語 (JA)、簡体中国語 (ZH) のいずれかです。

64 ビットの Windows では、32 ビットのソフトウェアに対応するレジストリエントリは *Software¥Wow6432Node¥SAP* にあります。

円記号で終わるパスは無効です。

インストーラは、.reg ファイルを作成し、実行することにより、この情報をすべてカプセル化できます。以下はインストールフォルダに c:¥sa17 を使用した場合の .reg ファイルの例です。

```
REGEDIT4
[HKEY_LOCAL_MACHINE¥Software¥SAP¥SQL Anywhere¥17.0]
"Location"="c:¥¥sa17"
"Language"="FR"
```

.reg ファイルでは、ファイルパスの円記号は別の円記号でエスケープします。

1.23.9.1.4 手順 4: システムパスを更新します。

管理ツールを実行するには、.exe ファイルと .dll ファイルが格納されているディレクトリをパスに含めます。c:¥sa17¥Bin32 ディレクトリまたは c:¥sa17¥Bin64 ディレクトリをシステムパスに追加する必要があります。

Windows では、システムパスは次のレジストリキーに保存されます。

```
HKEY_LOCAL_MACHINE¥
  SYSTEM¥
    CurrentControlSet¥
      Control¥
        Session Manager¥
          Environment¥
            Path
```

1.23.9.1.5 手順 5: *SQL Central* の接続プロファイルを作成します

この手順では *SQL Central* を設定します。*SQL Central* をインストールしない場合は、省略できます。

SQL Central がシステムにインストールされている場合は、SQL Anywhere 17 Demo の接続プロファイルがレポジトリファイルに作成されます。1 つ以上の接続プロファイルを作成しない場合は、この手順を省略できます。

次に示すのは、SQL Anywhere 17 Demo 接続プロファイルを作成するために使用されたコマンドです。独自の接続プロファイルを作成するときのモデルとして使用してください。

```
scjview -write "ConnectionProfiles/SQL Anywhere 17 Demo/Name" "SQL Anywhere 17 Demo"
scjview -write "ConnectionProfiles/SQL Anywhere 17 Demo/FirstTimeStart" "false"
scjview -write "ConnectionProfiles/SQL Anywhere 17 Demo/Description" "Suitable
Description"
scjview -write "ConnectionProfiles/SQL Anywhere 17 Demo/ProviderId"
"sqlanywhere1700"
scjview -write "ConnectionProfiles/SQL Anywhere 17 Demo/Provider" "SQL Anywhere 17"
scjview -write "ConnectionProfiles/SQL Anywhere 17 Demo/Data/
ConnectionProfileSettings" "DSN¥eSQL^0020Anywhere^002016^0020Demo;UID¥eDBA;PWD
¥e35c624d517fb"
scjview -write "ConnectionProfiles/SQL Anywhere 17 Demo/Data/ConnectionProfileName"
"SQL Anywhere 17 Demo"
scjview -write "ConnectionProfiles/SQL Anywhere 17 Demo/Data/ConnectionProfileType"
"SQL Anywhere"
```

接続プロファイルの文字列と値は、レポジトリファイルから抽出できます。*SQL Central* で接続プロファイルを定義し、レポジトリファイルの対応する行を確認します。

上で説明した処理で作成されたレポジトリファイルの一部を次に示します。エントリの一部は、表示のために複数行に分割されています。ファイルでは、各エントリは 1 行に表示されます。

```
#  Version: 17.0.1154
#  Thu Oct 04 12:07:53 EDT 2012
#
ConnectionProfiles/SQL Anywhere 17 Demo/Name=SQL Anywhere 17 Demo
ConnectionProfiles/SQL Anywhere 17 Demo/FirstTimeStart=false
ConnectionProfiles/SQL Anywhere 17 Demo/Description=Suitable Description
ConnectionProfiles/SQL Anywhere 17 Demo/ProviderId=sqlanywhere1700
ConnectionProfiles/SQL Anywhere 17 Demo/Provider=SQL Anywhere 17
ConnectionProfiles/SQL Anywhere 17 Demo/Data/ConnectionProfileSettings=
    DSN¥eSQL^0020Anywhere^002016^0020Demo;
    UID¥eDBA;
    PWD¥e35c624d517fb
ConnectionProfiles/SQL Anywhere 17 Demo/Data/ConnectionProfileName=
    SQL Anywhere 17 Demo
ConnectionProfiles/SQL Anywhere 17 Demo/Data/ConnectionProfileType=
     SQL Anywhere
```

1.23.9.1.6 手順 6: SQL Anywhere ODBC ドライバを登録します

SQL Anywhere ODBC ドライバをインストールしてから、管理ツールの JDBC ドライバでこれを使用します。

関連情報

ODBC ドライバの設定 [820 ページ]

1.23.9.2 Linux、Solaris、Mac OS X における管理ツールの配備

Linux、Solaris、Mac OS X のコンピュータで Interactive SQL (dbisql) と *SQL Central* (SQL Anywhere と Mobile Link のプラグインを含む) をインストールするには、一定の手順に従う必要があります。この項は、これらの管理ツールのインストーラの作成を望むユーザを対象としています。

ここで説明する手順は、バージョン 17.0.4 固有のもので、前後のバージョンには適用できない場合があります。

dbisqlc コマンドラインユーティリティは、Linux、Solaris、Mac OS X、HP-UX、AIX でサポートされています。

> **注記**
>
> ファイルの再配布はライセンス契約に従います。このマニュアル内の記述は、ライセンス契約のどの条項にも優先しません。配備について検討する前にライセンス契約を確認してください。

始める前に

始める前に、プログラムファイルのソースとして SQL Anywhere を 1 台のコンピュータにインストールしてください。これが配備の参照用インストールとなります。

一般的な手順は次のとおりです。

1. 配備するプログラムを決定します。
2. 必要なファイルをコピーします。
3. 環境変数を設定します。
4. *SQL Central* プラグインを登録します。

これらの各手順について詳しく説明します。

このセクションの内容:

手順 1: 配備するソフトウェアを決定します [848 ページ]
要件に応じて、管理ツールをさまざまな方法でバンドルできます。

手順 2: 必要なファイルをコピーします [849 ページ]
インストーラは、SQL Anywhere インストーラによってインストールされたファイルのサブセットをコピーします。同じディレクトリ構造を維持します。

手順 3: 環境変数を設定します。[853 ページ]
管理ツールを実行するには、いくつかの環境変数を定義または変更する必要があります。通常、これは、SQL Anywhere インストーラによって作成される `sa_config.sh` ファイルで行います。`sa_config.sh` ファイルを使用するには、このファイルをコピーし、SQLANY17 が配備ロケーションを指すように設定します。

手順 4: SQL Central の接続プロファイルを作成します [854 ページ]
この手順では *SQL Central* を設定します。*SQL Central* をインストールしない場合は、省略できます。

親トピック 管理ツールの配備 [837 ページ]

前のトピック: Windows での管理ツールの配備 [839 ページ]

次のトピック: 管理ツールの設定 [855 ページ]

関連情報

dbisqlc の配備 [857 ページ]

1.23.9.2.1 手順 1: 配備するソフトウェアを決定します

要件に応じて、管理ツールをさまざまな方法でバンドルできます。

以下のソフトウェアバンドルを任意に組み合わせてインストールできます。

- Interactive SQL
- SQL Anywhere プラグインを含む *SQL Central*
- Mobile Link プラグインを含む *SQL Central*

上記のどのソフトウェアバンドルをインストールする場合にも、次のコンポーネントが必要です。

- SQL Anywhere ODBC ドライバ
- Java SE Runtime Environment (JRE) 8。32 ビットバージョンと 64 ビットバージョンの JRE があります。

> **注記**
>
> Mac OS X における JRE バージョンを確認するには、*[アップル]* > *[システム環境設定]* > *[ソフトウェアアップデート]* をクリックします。*[インストールされたアップデート]* をクリックし、適用済みのアップデートのリストを表示します。

以下の項の手順は、これら 5 つのバンドルをどれでも (またはすべて) 競合なしでインストールできるように構成されています。

1.23.9.2.2 手順 2: 必要なファイルをコピーします

インストーラは、SQL Anywhere インストーラによってインストールされたファイルのサブセットをコピーします。同じディレクトリ構造を維持します。

参照用の SQL Anywhere インストール環境からファイルをコピーする場合は、ファイルのパーミッションも保持します。一般に、すべてのユーザとグループがすべてのファイルを読み取り、実行できます。

管理ツールは JRE 1.8.0 を必要とします。特に必要のないかぎり、これより新しいパッチバージョンの JRE を代用しないでください。

Linux/Solaris の場合、管理ツールには JRE の 64 ビットバージョンまたは 32 ビットバージョン (対象アーキテクチャによる) が必要です。Mobile Link サーバには JRE の 64 ビットバージョンが必要です。Mac OS X の場合、管理ツールには JRE の 64 ビットバージョンが必要です。JRE のすべてのプラットフォームバージョンが SQL Anywhere に付属しています。SQL Anywhere に含まれているプラットフォームでは、x86/x64 の Linux と Solaris SPARC をサポートしています。その他のプラットフォームバージョンは、適切なベンダーから入手する必要があります。たとえば、Linux のインストールを行っている場合は、サブフォルダを含めて `jre180` ツリー全体をコピーします。

必要なプラットフォームが SQL Anywhere に含まれている場合、SQL Anywhere のインストールされたコピーから JRE ファイルをコピーします。サブフォルダも含めてツリー全体をコピーします。

次の表は、各管理ツールと *SQL Central* プラグインに必須のファイルを示します。必要なファイルのリストを作成してから、前述したディレクトリ構造にコピーします。

テーブルには、フォルダ指定が binXX であるファイルが示されています。プラットフォームに応じて、これらのファイルには、32 ビットバージョンと 64 ビットバージョンが、それぞれ `bin32` フォルダと `bin64` フォルダにあります。32 ビットと 64 ビットの管理ツールをインストールしている場合は、それぞれのフォルダに両方の一連のファイルをインストールします。

テーブルには、フォルダ指定が libXX であるファイルが示されています。プラットフォームに応じて、これらのファイルには、32 ビットバージョンと 64 ビットバージョンが、それぞれ `lib32` フォルダと `lib64` フォルダにあります。32 ビットと 64 ビットの管理ツールをインストールしている場合は、それぞれのフォルダに両方の一連のファイルをインストールします。

管理ツールと *SQL Central* プラグインについて、複数のリンクを作成する必要があります。

Linux と Solaris の場合、配備するすべての共有オブジェクトについてシンボリックリンクを作成します。また、*$SQLANY17*/bin64 または *$SQLANY17*/bin32 にシンボリックリンクを作成します。Linux と他のシステムのシンボリックリンクは jre180 です。例を示します。

```
libdblib17_r.so -> $SQLANY17/lib32/libdblib17_r.so.1
jre180 -> $SQLANY17/bin32/jre180
```

64 ビット Linux の Mobile Link プラグインの場合、*$SQLANY17*/bin64 に追加のシンボリックリンクを作成します。Linux のシンボリックリンクは、64 ビット JRE 用の jre180 です。

```
jre180 -> $SQLANY17/bin64/jre180 (Linux)
```

Mac OS X では、共有オブジェクトの拡張子は .dylib です。次の dylib では Symlink (シンボリックリンク) の作成が必要になります。

```
libdbjodbc17.jnilib -> libdbjodbc17.dylib
libdblib17_r.jnilib -> libdblib17_r.dylib
libdbput17_r.jnilib -> libdbput17_r.dylib
libmljodbc17.jnilib -> libmljodbc17.dylib
```

テーブルには、指定が [LL] であるファイルが示されています。メッセージファイルは複数あり、それぞれが異なる言語をサポートしています。異なる言語のサポートをインストールするには、それらの言語のリソースファイルを含めてください。

Interactive SQL

Interactive SQL (dbisql) には次のファイルが必要です。

```
binXX/dbisql
binXX/jre180/...
libXX/libdbicu17_r.so (.dylib)
libXX/libdbicudt17.so (.dylib)
libXX/libdbjodbc17.so (.dylib)
libXX/libdblib17_r.so (.dylib)
libXX/libdbodbc17_r.so (.dylib)
libXX/libdbodm17.so (.dylib)
libXX/libjsyblib1700_r.so (.dylib)
res/dblg[LL]17.res
java/batik-all.jar
java/isql.jar
java/JComponents1700.jar
java/jodbc4.jar
java/jsyblib1700.jar
java/ngdbc.jar
java/saip17.jar
java/SCEditor1700.jar
java/xml-apis-ext.jar
```

上記のファイルパスには、「...」で終わっているものもあります。これは、サブフォルダも含めてツリー全体をコピーする必要があることを表します。32 ビットバージョンの Interactive SQL には、32 ビットバージョンの JRE ファイル (bin32/jre180) が必要です。64 ビットバージョンの Interactive SQL には、64 ビットバージョンの JRE ファイル (bin64/jre180) が必要です。

SQL Central

SQL Central (scjview) には次のファイルが必要です。

```
binXX/scjview
binXX/jre180/...
libXX/libjsyblib1700_r.so (.dylib)
java/jsyblib1700.jar
java/SCEditor1700.jar
java/sqlcentral1700.jar
```

上記のファイルパスには、「...」で終わっているものもあります。これは、サブディレクトリを含めたツリー全体をコピーする必要があることを示します。32 ビットバージョンの *SQL Central* には、32 ビットバージョンの JRE ファイル (`bin32/jre180`) が必要です。64 ビットバージョンの *SQL Central* には、64 ビットバージョンの JRE ファイル (`bin64/jre180`) が必要です。

SQL Anywhere プラグインを含む *SQL Central*

SQL Central の SQL Anywhere プラグインでは、パーソナルサーバ (*dbeng17* とサポートファイル) と次のファイルが必要です。

```
libXX/libdbicu17_r.so (.dylib)
libXX/libdbicudt17.so (.dylib)
libXX/libdbjodbc17.so (.dylib)
libXX/libdblib17_r.so (.dylib)
libXX/libdbodbc17_r.so (.dylib)
libXX/libdbodm17.so (.dylib)
libXX/libdbput17_r.so (.dylib)
libXX/libdbtasks17_r.so (.dylib)
libXX/libdbtool17_r.so (.dylib)
res/dblg[LL]17.res
java/batik-all.jar
java/dbdiff.jar
java/debugger.jar
java/diffutils-1.2.1.jar
java/isql.jar
java/JComponents1700.jar
java/jodbc4.jar
java/ngdbc.jar
java/saip17.jar
java/salib.jar
java/saplugin.jar
java/SQLAnywhere.jpr
java/xml-apis-ext.jar
```

Mobile Link プラグインを含む *SQL Central*

SQL Central の Mobile Link プラグインには次のファイルが必要です。

```
libXX/libdbicu17_r.so (.dylib)
libXX/libdbicudt17.so (.dylib)
libXX/libdblib17_r.so (.dylib)
libXX/libdbput17_r.so (.dylib)
```

```
libXX/libdbtasks17_r.so (.dylib)
libXX/libdbtool17_r.so (.dylib)
libXX/libmljodbc17.so (.dylib)
res/dblg[LL]17.res
java/commons-collections-3.2.1.jar
java/commons-lang-2.6.jar
java/commons-logging-1.2.jar
java/isql.jar
java/JComponents1700.jar
java/jodbc4.jar
java/mldesign.jar
java/mlplugin.jar
java/MobiLink.jpr
java/ngdbc.jar
java/saip17.jar
java/salib.jar
java/velocity-1.7-dep.jar
```

Ultra Light プラグインを含む *SQL Central*

SQL Central の Ultra Light プラグインには次のファイルが必要です。

```
libXX/libdbicu17_r.so (.dylib)
libXX/libdbicudt17.so (.dylib)
libXX/libdblib17_r.so (.dylib)
libXX/libdbput17_r.so (.dylib)
libXX/libdbtasks17_r.so (.dylib)
libXX/libdbtool17_r.so (.dylib)
libXX/libmlcrsa17.so (.dylib)
libXX/libulscutil17.so (.dylib)
libXX/libulutils17.so (.dylib)
res/dblg[LL]17.res
java/batik-all.jar
java/isql.jar
java/JComponents1700.jar
java/jodbc4.jar
java/ngdbc.jar
java/saip17.jar
java/salib.jar
java/ulplugin.jar
java/UltraLite.jpr
java/xml-apis-ext.jar
```

外国語のメッセージとコンテキスト別のヘルプファイル

Linux システムのみ、管理ツールのすべての表示テキストとコンテキスト別のヘルプは、英語からドイツ語、フランス語、日本語、簡体字中国語に翻訳されています。各言語のリソースは別々のファイルに保存されています。英語のファイルの場合は、ファイル名に en が含まれています。ドイツ語のファイル名には de、フランス語のファイル名には fr、日本語のファイル名には ja、中国語のファイル名には zh がそれぞれ含まれています。

異なる言語のサポートをインストールするには、他の言語のメッセージファイルを含めてください。翻訳済みのファイルは次のとおりです。

dblgen*17*.res	英語
dblgde*17*_iso_1.res、dblgde*17*_utf8.res	ドイツ語 (Linux のみ)
dblgja*17*_eucjis.res、dblgja*17*_sjis.res、dblgja*17*_utf8.res	日本語 (Linux のみ)
dblgzh*17*_cp936.res、dblgzh*17*_eucgb.res、dblgzh*17*_utf8.res	簡体字中国語 (Linux のみ)

これらのファイルは、SQL Anywhere のローカライズ版に含まれます。

1.23.9.2.3 手順 3: 環境変数を設定します。

管理ツールを実行するには、いくつかの環境変数を定義または変更する必要があります。通常、これは、SQL Anywhere インストーラによって作成される sa_config.sh ファイルで行います。sa_config.sh ファイルを使用するには、このファイルをコピーし、SQLANY17 が配備ロケーションを指すように設定します。

または、環境変数を設定するには、次の作業を実行します。

1. 以下の環境変数を設定します。

```
SQLANY17=SQL-Anywhere-install-dir
```

2. 以下のいずれかを含むように PATH を設定します。

32 ビット Linux

```
$SQLANY17/bin32
$SQLANY17/bin32/jre180/bin
```

64 ビット Linux

```
$SQLANY17/bin64
$SQLANY17/bin32
$SQLANY17/bin64/jre180/bin
```

64 ビット Solaris

```
$SQLANY17/bin64
$SQLANY17/bin32
$SQLANY17/bin64/jre180/bin/sparcv9
```

3. 以下のフォルダを含むように LD_LIBRARY_PATH を設定します。

32 ビット Linux

```
$SQLANY17/lib32
$SQLANY17/lib64
$SQLANY17/bin32/jre180/lib/i386/server
$SQLANY17/bin32/jre180/lib/i386
```

64 ビット Linux および Solaris

```
$SQLANY17/lib64
$SQLANY17/lib32
$SQLANY17/bin64/jre180/lib/amd64/server
```

```
$SQLANY17/bin64/jre180/lib/amd64
```

Mac OS X では、管理ツールは、インストール時に生成され、各 Java アプリケーションバンドルの `Contents/MacOS` フォルダ内に格納される `sa_java_stub_launcher.sh` というシェルスクリプトスタブランチャを利用します。生成されると、スクリプトは `System/bin64/sa_config.sh` ファイルをソースとして環境を設定し、JavaApplicationStub バイナリを実行します。実際の Java アプリケーションはこのバイナリによって起動されます。配備のため、必要に応じて `sa_java_stub_launcher.sh` を変更し、環境を設定できます。Java アプリケーションバンドル内の `Info.plist` ファイルを変更し、キー CFBundleExecutable の文字列値を変更して、スクリプトの名前を変更できます。

1.23.9.2.4 手順 4: *SQL Central* の接続プロファイルを作成します

この手順では *SQL Central* を設定します。*SQL Central* をインストールしない場合は、省略できます。

SQL Central がシステムにインストールされている場合は、SQL Anywhere 17 Demo の接続プロファイルがレポジトリファイルに作成されます。1 つ以上の接続プロファイルを作成しない場合は、この手順を省略できます。

次に示すのは、SQL Anywhere 17 Demo 接続プロファイルを作成するために使用されたコマンドです。独自の接続プロファイルを作成するときのモデルとして使用してください。

```
scjview -write "ConnectionProfiles/SQL Anywhere 17 Demo/Name" "SQL Anywhere 17 Demo"
scjview -write "ConnectionProfiles/SQL Anywhere 17 Demo/FirstTimeStart" "false"
scjview -write "ConnectionProfiles/SQL Anywhere 17 Demo/Description" "Suitable
Description"
scjview -write "ConnectionProfiles/SQL Anywhere 17 Demo/ProviderId"
"sqlanywhere1700"
scjview -write "ConnectionProfiles/SQL Anywhere 17 Demo/Provider" "SQL Anywhere 17"
scjview -write "ConnectionProfiles/SQL Anywhere 17 Demo/Data/
ConnectionProfileSettings" "DSN¥eSQL^0020Anywhere^002016^0020Demo;UID¥eDBA;PWD
¥e35c624d517fb"
scjview -write "ConnectionProfiles/SQL Anywhere 17 Demo/Data/ConnectionProfileName"
"SQL Anywhere 17 Demo"
scjview -write "ConnectionProfiles/SQL Anywhere 17 Demo/Data/ConnectionProfileType"
"SQL Anywhere"
```

接続プロファイルの文字列と値は、レポジトリファイルから抽出できます。*SQL Central* で接続プロファイルを定義し、レポジトリファイルの対応する行を確認します。

上で説明した処理で作成されたレポジトリファイルの一部を次に示します。エントリの一部は、表示のために複数行に分割されています。ファイルでは、各エントリは 1 行に表示されます。

```
#  Version: 17.0.1154
#  Thu Oct 04 12:07:53 EDT 2012
#
ConnectionProfiles/SQL Anywhere 17 Demo/Name=SQL Anywhere 17 Demo
ConnectionProfiles/SQL Anywhere 17 Demo/FirstTimeStart=false
ConnectionProfiles/SQL Anywhere 17 Demo/Description=Suitable Description
ConnectionProfiles/SQL Anywhere 17 Demo/ProviderId=sqlanywhere1700
ConnectionProfiles/SQL Anywhere 17 Demo/Provider=SQL Anywhere 17
ConnectionProfiles/SQL Anywhere 17 Demo/Data/ConnectionProfileSettings=
    DSN¥eSQL^0020Anywhere^002016^0020Demo;
    UID¥eDBA;
    PWD¥e35c624d517fb
ConnectionProfiles/SQL Anywhere 17 Demo/Data/ConnectionProfileName=
    SQL Anywhere 17 Demo
ConnectionProfiles/SQL Anywhere 17 Demo/Data/ConnectionProfileType=
     SQL Anywhere
```

1.23.9.3 管理ツールの設定

OEM.ini という名前の初期化ファイルを使用すると、どの機能が表示されるか、有効になるかを指定できます。

このファイルは管理ツールが使用する JAR ファイルと同じディレクトリにある必要があります (例 : *C:¥Program Files¥SQL Anywhere 17*¥Java)。このファイルが見つからなかった場合は、デフォルト値が使用されます。また、OEM.ini に指定がない値についてもデフォルトが使用されます。

> **注記**
>
> 管理ツールを再配備するときに、ツールは SQL Anywhere ソフトウェア更新をチェックすることはできません。*[更新のチェック]* メニュー項目とオプションは、再配備されたバージョンには表示されません。

サンプルの OEM.ini ファイルを次に示します。

```
[errors]
# reportErrors type is boolean, default = true
reportErrors=true
[updates]
# checkForUpdates type is boolean, default = true
checkForUpdates=true
[preferences]
directory=preferences_files_directory
[dbisql]
allowPasswordsInFavorites=true
disableExecuteAll=false
# lockedPreferences is assigned a comma-separated
# list of one or more of the following option names:
#    autoCommit
#    autoRefetch
#    commitOnExit
#    disableResultsEditing
#    executeToolbarButtonSemantics
#    maximumDisplayedRows
```

文字 # で始まる行はコメント行なので無視されます。指定されるオプション名と値では、大文字と小文字が区別されます。

OEM.ini ファイルは、次のサブセクションに分かれています。

[errors]

このサブセクションで設定されるオプションは、すべての管理ツールに適用されます。

reportErrors が false の場合、管理ツールはソフトウェアがクラッシュしたときにユーザがテクニカルサポートにエラー情報を送信できるウィンドウを表示しません。代わりに、標準のウィンドウが表示されます。

[updates]

このサブセクションで設定されるオプションは、すべての管理ツールに適用されます。

checkForUpdates が false の場合、管理ツールは SQL Anywhere ソフトウェア更新のチェックを自動的には行わず、ユーザに選択オプションを提示することもしません。

[preferences]

このサブセクションで設定されるオプションは、すべての管理ツールに適用されます。

directory オプションを設定して、管理ツールがユーザ固有の設定ファイルの保存に使用するディレクトリを指定します。これらのファイルには管理ツールの設定や履歴に関する情報が格納されます。たとえば、Interactive SQL 文の履歴、最近開いたファイル、保存されたウィンドウ位置などが格納されます。

パスの区切り文字 (Windows では円記号。Linux、UNIX、Mac OS X ではスラッシュ) で終わらない、完全に修飾されたディレクトリ名 (`c:¥work¥prefs` など) を指定する必要があります。

Windows では、ユーザ設定ディレクトリのデフォルトの設定は `%appdata%¥SAP` です。この設定により、複数のユーザが 1 つのシステムでそれぞれ自分の設定保存用ディレクトリを持つことができるようになります。`OEM.ini` ファイルでこの設定を上書きすると、この機能が無効になります。

[performancedata]

このサブセクションで設定されるオプションは、Interactive SQL と *SQL Central* だけに適用されます。

showPerformanceDataUI

showPerformanceDataUI が false の場合、管理ツールはソフトウェア開発チームへのパフォーマンスデータの自動送信をユーザが有効または無効にするオプションを提供しません。

[dbisql]

このサブセクションで設定されるオプションは、Interactive SQL だけに適用されます。

allowPasswordsInFavorites

allowPasswordsInFavorites が false の場合、Interactive SQL は *[お気に入りに追加]* ウィンドウから *[接続パスワードの保存]* チェックボックスを削除します。デフォルトの設定は true であり、これはチェックボックスが存在することを意味します。

disableExecuteAll

disableExecuteAll が true に設定されている場合、*[SQL]* > *[実行]* メニュー項目とアクセラレータキー [F5] は Interactive SQL で無効になります。ツールバーの *[実行]* ボタンが *[実行]* に設定されている場合は、このボタンも無効になります。このため、Interactive SQL でツールバーの [実行] ボタンを *[選択の実行]* に設定し、さらに `OEM.ini` ファイルで executeToolbarButtonSemantics オプションを設定して、ツールバーの *[実行]* ボタンをユーザが変更できないようにする必要がある場合があります。

lockedPreferences

Interactive SQL オプションの設定をロックして、ユーザが変更できないようにすることがでいます。オプション名では、大文字と小文字が区別されます。次はその例です。

```
[dbisql]
lockedPreferences=autoCommit
```

次の Interactive SQL オプションの設定は、ユーザが変更できないように指定できます。

autoCommit

ユーザが *[各文の後にコミット]* オプションをカスタマイズできないようにします。

autoRefetch

ユーザが *[結果の自動再フェッチ]* オプションをカスタマイズできないようにします。

commitOnExit

ユーザが *[終了時または切断時にコミット]* オプションをカスタマイズできないようにします。

disableResultsEditing

ユーザが *[編集の無効化]* オプションをカスタマイズできないようにします。

executeToolbarButtonSemantics

ユーザがツールバーの *[実行]* ボタンの動作をカスタマイズできないようにします。

maximumDisplayedRows

ユーザが *[表示できるローの最大数]* オプションをカスタマイズできないようにします。

親トピック 管理ツールの配備 [837 ページ]

前のトピック: Linux、Solaris、Mac OS X における管理ツールの配備 [847 ページ]

次のトピック: dbisqlc の配備 [857 ページ]

1.23.9.4 dbisqlc の配備

配備したアプリケーションがクエリの実行やツールのスクリプト記述を必要とし、リソースに制限のあるコンピュータで実行されている場合には、Interactive SQL (dbisql) ではなく dbisqlc 実行プログラムを配備できます。

ただし、dbisqlc は推奨されておらず、新しい機能は追加されません。また、dbisqlc は Interactive SQL のすべての機能は備えておらず、両者間の互換性も保証されていません。

dbisqlc 実行プログラムには、標準 Embedded SQL クライアント側ライブラリが必要です。

親トピック 管理ツールの配備 [837 ページ]

前のトピック: 管理ツールの設定 [855 ページ]

次のトピック: SQL Anywhere モニタ Production Edition について [858 ページ]

1.23.9.5 SQL Anywhere モニタ Production Edition について

Production Edition は、配備と運用の目的で使用されます。別個にインストールされ、サービスとして実行されます。Production Edition には、すべての機能が搭載された SQL Anywhere インストールが含まれます。

SQL Anywhere のアップグレードや更新によって、モニタ Production Edition が上書きされたり、影響を受けたりすることはありません。これに対し、モニタ Developer Edition は、アップグレードや更新によって影響を受ける場合があります。これは、Developer Edition では、インストールされた SQL Anywhere をバックエンドで使用するためです。

リソースが実行されているコンピュータとは別のコンピュータにモニタをインストールする必要があります。これには次に挙げる 2 つの利点があります。

- データベースサーバ、Mobile Link サーバやその他のアプリケーションに対する影響が最小限に抑えられます。
- リソースがインストールされているコンピュータに問題が発生しても、モニタリングは影響を受けません。

このセクションの内容:

モニタ Production Edition のインストール (Windows の場合) [858 ページ]
モニタ Production Edition を Windows にインストールするには、次の手順に従います。

モニタ Production Edition のインストール (Linux の場合) [859 ページ]
モニタ Production Edition を Linux にインストールするには、次の手順に従います。

親トピック 管理ツールの配備 [837 ページ]

前のトピック: dbisqlc の配備 [857 ページ]

1.23.9.5.1 モニタ Production Edition のインストール (Windows の場合)

モニタ Production Edition を Windows にインストールするには、次の手順に従います。

コンテキスト

モニタはサービスとして実行されます。モニタには、すべての機能が搭載された SQL Anywhere インストールが含まれます。

手順

1. インストールメディアの `Monitor` ディレクトリにある `setup.exe` を実行し、表示される指示に従います。

   Windows では、モニタサービスはインストールによって起動されます。

2. モニタにログインするためのデフォルトの URL を開きます。`http://localhost:4950`

   > **注記**
   >
   > モニタにリモートでログインしている場合、http://`computer-name`:4950 をブラウズします。`computer-name` はモニタが稼働しているコンピュータの名前です。

3. ログインします。

   プロンプトが表示されたら、モニタのユーザ名とパスワードを入力します。デフォルトのユーザは、名前 *admin* とパスワード *admin* を持つモニタの管理者です。

結果

モニタ Production Edition がインストールされます。

1.23.9.5.2 モニタ Production Edition のインストール (Linux の場合)

モニタ Production Edition を Linux にインストールするには、次の手順に従います。

前提条件

root ユーザであることが必要です。

コンテキスト

モニタはサービスとして実行されます。モニタには、すべての機能が搭載された SQL Anywhere インストールが含まれます。

手順

1. root ユーザでインストールメディアの `Monitor` ディレクトリから `setup.tar` を実行し、表示される指示に従います。

   Linux では、デフォルトで、モニタ Production Edition はモニタサービスを自動的に起動します。

2. モニタにログインするためのデフォルトの URL を開きます。`http://localhost:4950`

   > **注記**
   >
   > モニタにリモートでログインしている場合、http://`computer-name`:4950 をブラウズします。`computer-name` はモニタが稼働しているコンピュータの名前です。

3. ログインします。

   プロンプトが表示されたら、モニタのユーザ名とパスワードを入力します。デフォルトのユーザは、名前 *admin* とパスワード *admin* を持つモニタの管理者です。

結果

モニタ Production Edition がインストールされます。

次のステップ

モニタを起動できます。

1.23.10 マニュアルの配備

マニュアルを配備する方法には、2 つの選択肢があります。ヘルプファイルが配備されない DocCommentXchange (http://dcx.sap.com) を使用する方法と、HTMLHelp (.chm) ヘルプファイルを配備する方法です。

Windows の場合、HTML ベースのヘルプマニュアルは、*%SQLANY17%*¥Documentation ディレクトリツリーにインストールされます。

1.23.11 データベースサーバの配備

データベースサーバを稼働するには、一連の適切なファイルをインストールする必要があります。

データベースサーバは SQL Anywhere のインストーラをエンドユーザが使用できるようにすることによって配備できます。適切なオプションを選択することによって、各エンドユーザが必要とするファイルを取得できます。

パーソナルデータベースサーバやネットワークデータベースサーバを配備する最も簡単な方法は、*Deployment* ウィザードを使用することです。

データベースサーバで必要なファイルを次の表に示します。これらのファイルの再配布はすべてライセンス契約の条項に従う必要があります。データベースサーバファイルを再配布する権利があるかどうかを事前に確認する必要があります。

Windows	Linux/UNIX	Mac OS X
dbeng17.exe	*dbeng17*	*dbeng17*
dbeng17.lic	*dbeng17*.lic	*dbeng17*.lic
dbsrv17.exe	*dbsrv17*	*dbsrv17*
dbsrv17.lic	*dbsrv17*.lic	*dbsrv17*.lic

Windows	Linux/UNIX	Mac OS X
dbserv*17*.dll	libdbserv*17*_r.so、 libdbtasks*17*_r.so	libdbserv*17*_r.dylib、 libdbtasks*17*_r.dylib
dbscript*17*.dll	libdbscript*17*_r.so	libdbscript*17*_r.dylib
dblg[LL]*17*.dll	dblg[LL]*17*.res	dblgen*17*.res
dbghelp.dll	N/A	N/A
dbctrs*17*.dll	N/A	N/A
dbextf.dll[1]	libdbextf.so[1]	libdbextf.dylib[1]
dbicu*17*.dll[2]	libdbicu*17*_r.so[2]	libdbicu*17*_r.dylib[2]
dbicudt*17*.dll[2]	libdbicudt*17*.so[2]	libdbicudt*17*.dylib[2]
sqlany.cvf	sqlany.cvf	sqlany.cvf
dbrsa*17*.dll	libdbrsa*17*_r.so	libdbrsa*17*_r.dylib
dbrsakp*17*.dll[3]	libdbrsakp*17*_r.so[3]	libdbrsakp*17*_r.dylib[3]
dbodbc*17*.dll[4]	libdbodbc*17*.so[4]	libdbodbc*17*.dylib[4]
dbjodbc*17*.dll[4]	libdbjodbc*17*.so[4]	libdbjodbc*17*.dylib[4]
N/A	libdbodbc*17*_n.so[4]	libdbodbc*17*_n.dylib[4]
N/A	libdbodbc*17*_r.so[4]	libdbodbc*17*_r.dylib[4]
dbjdbc*17*.dll[5]	libdbjdbc*17*.so[5]	libdbjdbc*17*.dylib[5]
java¥sajdbc4.jar[5]	java/sajdbc4.jar[5]	java/sajdbc4.jar[5]
java¥sajvm.jar[5]	java/sajvm.jar[5]	java/sajvm.jar[5]
dbcis*17*.dll[6]	libdbcis*17*.so[6]	libdbcis*17*.dylib[6]
libsybbr.dll[7]	libsybbr.so[7]	libsybbr.dylib[7]
sqlacockpit.template[8]	sqlacockpit.template[8]	sqlacockpit.template[8]
dbsupport.exe[9]	dbsupport[9]	dbsupport[9]

[1] システム拡張ストアドプロシージャとファンクションの廃止されたバージョン (xp_*) を使用する場合のみ必要です。 use_old_dbextf.sql スクリプトは、廃止されたバージョンを使用するデータベースで使用します。

[2] データベースの文字セットがマルチバイトの場合、または UCA 照合順が使用される場合のみ必要です。

[3] 暗号化された TDS 接続にのみ必要です。

[4] バージョン 10 のデータベースで Java を使用する場合のみ必要です。

[5] データベースで Java を使用する場合のみ必要です。

[6] リモートデータアクセスを使用する場合のみ必要です。

[7] アーカイブバックアップにのみ必要です。

[8] SQL Anywhere コックピット を使用する場合のみ必要です。

[9] SAP にクラッシュレポートを自動的に送信する場合にのみ必要です (ターゲットシステムで次のコマンドを一度実行してください:`dbsupport -cc autosubmit`)。

注記

- 場合によって、パーソナルデータベースサーバ (dbeng17) またはネットワークデータベースサーバ (dbsrv17) を配備するかどうか選択してください。
- データベースサーバを配備するときは、対応するライセンスファイル (*dbeng17*`.lic` または *dbsrv17*`.lic`) を含める必要があります。ライセンスファイルは、サーバの実行プログラムと同じフォルダにあります。
- 言語リソースライブラリファイルも含めてください。上記の表には、指定が [LL] であるファイルが示されています。メッセージファイルは複数あり、それぞれが異なる言語をサポートしています。異なる言語のサポートをインストールするには、それらの言語のリソースファイルを追加してください。[LL] を言語コード (**`en`**、**`de`**、**`ja`** など) に置き換えます。
- Java VM jar ファイル (`sajvm.jar`) は、データベースサーバがデータベース機能で Java を使用する場合のみ必要です。
- 表には、dbbackup などのユーティリティの実行に必要なファイルは含まれていません。
- データベースサーバの -xd オプションはデータベースサーバがデフォルトのデータベースサーバにならないようにするため、配備したアプリケーションで役立ちます。接続文字列にサーバ名が含まれている場合にのみ、接続が受け入れられます。

このセクションの内容:

Intel ストレージドライバの堅牢性の向上 [863 ページ]
: レジストリの EnableFlush パラメータを指定して、特定の Intel ストレージドライバを使用するシステムの停電に対する堅牢性を向上させます。このパラメータがないと、停電発生時にデータが失われたり、データベースが破損したりする可能性があります。

イベントログメッセージのフォーマット [864 ページ]
: レジストリキーを作成し、サーバによって Windows のイベントログに書き込まれたメッセージの形式が正しいことを確認します。

Mac OS X の考慮事項 [865 ページ]
: Mac OS X では、DBLauncher と SyncConsole は、ユーザのデフォルトシステムを使用して、SQL Anywhere インストールの場所を特定します。

データベースの配備 [866 ページ]
: データベースファイルは、エンドユーザのディスクにインストールすることによって配備します。

関連情報

Windows 用 Deployment ウィザード [793 ページ]

Linux/UNIX および MAC OS X の Deployment ウィザード [801 ページ]

管理ツールの配備 [837 ページ]

1.23.11.1 Intel ストレージドライバの堅牢性の向上

レジストリの EnableFlush パラメータを指定して、特定の Intel ストレージドライバを使用するシステムの停電に対する堅牢性を向上させます。このパラメータがないと、停電発生時にデータが失われたり、データベースが破損したりする可能性があります。

手順

1. 次のレジストリエントリが存在するかどうかを確認してください。

   ```
   HKEY_LOCAL_MACHINE¥System¥CurrentControlSet¥Services¥iastor¥Parameters¥
   ```

2. レジストリエントリが存在する場合、キー内の EnableFlush という REG_DWORD の値を追加し、1 のデータ値を割り当てます。
3. 次のレジストリエントリが存在するかどうかを確認してください。

   ```
   HKEY_LOCAL_MACHINE¥System¥CurrentControlSet¥Services¥iastorv¥Parameters¥
   ```

4. レジストリエントリが存在する場合、キー内の EnableFlush という名前の REG_DWORD 値を追加し、1 のデータ値を割り当てます。

結果

EnableFlush パラメータが指定され、停電が発生した場合のデータ損失を防ぎます。

例

次の例は、両方の可能な場所の EnableFlush パラメータを指定するレジストリファイルを示します。

```
REGEDIT4
[HKEY_LOCAL_MACHINE¥SYSTEM¥CurrentControlSet¥Services¥iastor¥Parameters]
"EnableFlush"=dword:00000001
[HKEY_LOCAL_MACHINE¥SYSTEM¥CurrentControlSet¥Services¥iastorv¥Parameters]
"EnableFlush"=dword:00000001
```

1.23.11.2 イベントログメッセージのフォーマット

レジストリキーを作成し、サーバによって Windows のイベントログに書き込まれたメッセージの形式が正しいことを確認します。

手順

1. レジストリキーを作成します。

   32 ビットバージョンのサーバの場合、次のキーを作成します。

   ```
   HKEY_LOCAL_MACHINE¥SYSTEM¥CurrentControlSet¥Services¥Eventlog¥Application¥SQLANY
   17.0
   ```

   64 ビットバージョンのサーバの場合、次のキーを作成します。

   ```
   HKEY_LOCAL_MACHINE¥SYSTEM¥CurrentControlSet¥Services¥Eventlog¥Application
   ¥SQLANY64 17.0
   ```

2. このキー内で、EventMessageFile という REG_SZ 値を追加し、`dblgen17.dll` の完全に修飾されたロケーションのデータ値を割り当てます (例:*C:¥Program Files¥SQL Anywhere 17*`¥Bin32¥dblgen17.dll`)。

   次の例では、`c:¥sa17¥bin32¥dblgen17.dll` ディレクトリに対して EventMessageFile キーを指定します。

   ```
   "EventMessageFile"="c:¥¥sa17¥¥bin32¥¥dblgen17.dll"
   ```

   英語バージョンの DLL である `dblgen17.dll` は、配備の言語にかかわらず指定できます。

3. レジストリキーを作成し、MESSAGE...TO EVENT LOG 文によってイベントログに書き込まれたメッセージの形式が正しいことを確認します。

   32 ビットバージョンのサーバの場合、次のキーを作成します。

   ```
   HKEY_LOCAL_MACHINE¥SYSTEM¥CurrentControlSet¥Services¥Eventlog¥Application¥SQLANY
   17.0 Admin
   ```

   64 ビットバージョンのサーバの場合、次のキーを作成します。

   ```
   HKEY_LOCAL_MACHINE¥SYSTEM¥CurrentControlSet¥Services¥Eventlog¥Application
   ¥SQLANY64 17.0 Admin
   ```

4. このキー内で、EventMessageFile という REG_SZ 値を追加し、`dblgen17.dll` の完全に修飾されたロケーションのデータ値を割り当てます (例:*C:¥Program Files¥SQL Anywhere 17*`¥Bin32¥dblgen17.dll`)。

   次の例では、`c:¥sa17¥bin32¥dblgen17.dll` ディレクトリに対して EventMessageFile キーを指定します。

   ```
   "EventMessageFile"="c:¥¥sa17¥¥bin32¥¥dblgen17.dll"
   ```

   英語バージョンの DLL である `dblgen17.dll` は、配備の言語にかかわらず指定できます。

5. 次のレジストリキーを作成し、ログエントリの抑制を制御します。

   ```
   Software¥SAP¥SQL Anywhere¥17.0¥EventLogMask
   ```

キーは HKEY_CURRENT_USER または HKEY_LOCAL_MACHINE ハイブのいずれかに配置できます。

6. EventLogMask という名前の REG_DWORD 値を作成し、Windows の別のイベントタイプの内部ビットが含まれたビットマスク値を割り当てます。

   次のビットタイプは、SQL Anywhere データベースサーバによって使用されます。

   ```
   EVENTLOG_ERROR_TYPE         0x0001
   EVENTLOG_WARNING_TYPE       0x0002
   EVENTLOG_INFORMATION_TYPE   0x0004
   ```

   たとえば、EventLogMask キーを 0 に設定すると、メッセージはまったく出力されなくなります。推奨される設定は 1 です。情報メッセージと警告メッセージは出力されませんが、エラーメッセージは出力されます。デフォルト設定 (エントリが存在しない場合) では、すべてのメッセージタイプが出力されます。サンプルのレジストリ設定を以下に示します。

   ```
   "EventLogMask"=dword:00000007
   ```

結果

イベントログメッセージは対象の言語でフォーマットされ、EventLogMask レジストリキーに割り当てられたビットマスク値に応じてレポートされます。

例

次の例は、32 ビットバージョンのサーバによって送信されるイベントログメッセージをフォーマットするサンプルレジストリファイルを示します。

```
REGEDIT4
[HKEY_LOCAL_MACHINE¥SYSTEM¥CurrentControlSet¥Services¥Eventlog¥Application¥SQLANY
17.0]
"EventMessageFile"="c:¥¥sa17¥¥bin32¥¥dblgen17.dll"
[HKEY_LOCAL_MACHINE¥SYSTEM¥CurrentControlSet¥Services¥Eventlog¥Application¥SQLANY
17.0 Admin]
"EventMessageFile"="c:¥¥sa17¥¥bin32¥¥dblgen17.dll"
[HKEY_LOCAL_MACHINE¥SOFTWARE¥SAP¥SQL Anywhere¥17.0]
"EventLogMask"=dword:00000003
```

この例では、`c:¥sa17¥bin32¥dblgen17.dll` にあると仮定されている英語バージョンの DLL を使用して、イベントログメッセージをフォーマットします。EventLogMask のビットマスク値は、情報メッセージではなく、エラーと警告のみが記録されることを示します。

1.23.11.3 Mac OS X の考慮事項

Mac OS X では、DBLauncher と SyncConsole は、ユーザのデフォルトシステムを使用して、SQL Anywhere インストールの場所を特定します。

SQL Anywhere インストーラでは、インストールを実行しているユーザのユーザデフォルトレポジトリに、インストール先ロケーションを書き込みます。その他のユーザは、DBLauncher を初めて使用するとき、インストールディレクトリの入力を要求されます。また、[ユーザ設定] パネルを使用して、この設定を変更することもできます。

1.23.11.4 データベースの配備

データベースファイルは、エンドユーザのディスクにインストールすることによって配備します。

データベースサーバが正常に停止するかぎりは、データベースファイルとともにトランザクションログファイルを配備する必要はありません。エンドユーザがデータベースの実行を開始するときに、新しいトランザクションログが作成されます。

SQL Remote アプリケーションでは、データベースが正しく同期された状態で作成してください。そうすれば、トランザクションログは必要ありません。この目的で、抽出ユーティリティを使用することができます。

このセクションの内容:

グローバル配備に関する考慮事項 [866 ページ]
データベースをグローバルに配備するときは、データベースが使用される場所 (ロケール) について考慮してください。ロケールにより、ソート順やテキスト比較ルールが異なる場合があるためです。

読み込み専用メディアでのデータベースの配備 [867 ページ]
読み込み専用モードで実行するかぎり、CD-ROM などの読み込み専用メディアでデータベースを配布および実行できます。

1.23.11.4.1 グローバル配備に関する考慮事項

データベースをグローバルに配備するときは、データベースが使用される場所 (ロケール) について考慮してください。ロケールにより、ソート順やテキスト比較ルールが異なる場合があるためです。

たとえば、配備するデータベースが 1252LATIN1 照合で作成されているとすると、使用環境によっては不適切な場合があります。

データベースの照合は作成後に変更できないため、インストールの段階でデータベースを作成して、必要なスキーマやデータをデータベースに後で移植することを検討してください。データベースをインストール中に作成するには、dbinit ユーティリティを使用するか、またはユーティリティデータベースを指定してデータベースサーバを起動し、CREATE DATABASE 文を発行します。次に SQL 文を使用してスキーマを作成し、必要な操作を行って初期状態のデータベースを設定します。

UCA 照合を使用する場合は、dbinit ユーティリティまたは CREATE DATABASE 文を使用して、文字列のソートや比較を詳細に制御するために照合の適合化オプションを指定することができます。これらのオプションは、"keyword=value" のペアの形式で、カッコで囲んで指定して、その後ろに照合名を記述します。たとえば CREATE DATABASE 文を使用した場合は、次のような構文を使用して照合の適合化を指定できます。

```
CHAR COLLATION 'UCA( locale=es;case=respect;accent=respect )'
```

または、データベースが使用されるロケールごとに 1 つずつ、複数のデータベーステンプレートを作成することもできます。データベースを配備するロケールが比較的少ない場合は、この方法が適しています。インストールするデータベースをインストーラで選択することができます。

1.23.11.4.2 読み込み専用メディアでのデータベースの配備

読み込み専用モードで実行するかぎり、CD-ROM などの読み込み専用メディアでデータベースを配布および実行できます。

読み取り専用モードでデータベースを実行するには、-r データベースサーバオプションを使用します。

データベースの変更が必要な場合は、CD-ROM から変更作業ができるハードドライブなどの場所にデータベースをコピーしてください。

1.23.12 Windows での DLL の登録

DLL ファイルによっては、SQL Anywhere での使用を目的として配備する場合に登録を必要とするものがあります。

Windows 7 以降では、DLL を登録または登録解除するときに必要な権限の昇格をサポートする昇格操作エージェント (`dbelevate17.exe`) を含める必要があります。

インストールスクリプトまたは regsvr32 ユーティリティの使用を含め、これらの DLL を登録できるさまざまな方法があります。

次の表は、Windows に配備する場合に登録を必要とする DLL を示します。

ファイル	説明
`dbctrs17.dll`	Windows パフォーマンスモニタのカウンタ
`dbodbc17.dll`	SQL Anywhere ODBC ドライバ
`dboledb17.dll`	SQL Anywhere OLE DB プロバイダ
`dboledba17.dll`	SQL Anywhere OLE DB プロバイダスキーマ支援モジュール (Windows のみ)

例

次のコマンドを実行し、regsvr32 ユーティリティを使用する ODBC ドライバを Windows に登録します。

```
regsvr32 dbodbc17.dll
```

1.23.13 外部環境のサポートの配備

複数のコンポーネントは、データベースアプリケーションの外部呼び出しをサポートする必要があります。

次の表では、配備する必要があるコンポーネントを示します。

ESQL/ODBC の外部呼び出し

コンポーネント	Windows	Linux/UNIX	Mac OS X
ESQL と ODBC のランチャ	dbexternc*17*.exe	dbexternc*17*	dbexternc*17*
ブリッジ	dbextenv*17*.dll	libdbextenv*17*_r.so	libdbextenv*17*_r.dylib
SQL Anywhere C API のランタイム	dbcapi.dll	libdbcapi_r.so	libdbcapi_r.dylib
DBLIB	dblib*17*.dll	libdblib*17*_r.so	libdblib*17*_r.dylib

Embedded SQL アプリケーションで必要な追加ファイルの場合、Embedded SQL クライアント配備に関するマニュアルを参照してください。

ODBC アプリケーションで必要な追加ファイルの場合、ODBC クライアント配備に関するマニュアルを参照してください。

Java の外部呼び出し

コンポーネント	Windows	Linux/UNIX	Mac OS X
Java のインストール (サードパーティ)	java.exe	java	java
ランチャ	sajvm.jar	sajvm.jar	sajvm.jar
SQL Anywhere JDBC ドライバ (サーバ側呼び出し)	dbjdbc*17*.dll	libdbjdbc*17*.so	libdbjdbc*17*.dylib

JDBC アプリケーションで必要な追加ファイルの場合、JDBC クライアント配備に関するマニュアルを参照してください。

.NET CLR の外部呼び出し

コンポーネント	Windows	Linux/UNIX	Mac OS X
.NET 3.5 以降	(Microsoft 提供)	N/A	N/A
SQL Anywhere .NET 3.5 または 4.5 データプロバイダ	(SAP 提供)	N/A	N/A
.NET CLR のブリッジ	dbextclr*17*.exe	N/A	N/A
ブリッジ	dbextenv*17*.dll	N/A	N/A
.NET CLR のサポート	dbclrenv*17*.dll	N/A	N/A
DBLIB	dblib*17*.dll	N/A	N/A

Perl の外部呼び出し

コンポーネント	Windows	Linux/UNIX	Mac OS X
Perl のインストール (サードパーティ)	perl.exe	perl	perl
Perl のランチャ	perlenv.pl	perlenv.pl	perlenv.pl
ブリッジ	dbextenv*17*.dll	libdbextenv*17*_r.so	libdbextenv*17*_r.dylib
SQL Anywhere C API のランタイム	dbcapi.dll	libdbcapi_r.so	libdbcapi_r.dylib
DBLIB	dblib*17*.dll	libdblib*17*_r.so	libdblib*17*_r.dylib

Perl アプリケーションで必要な追加ファイルの場合、DBD::SQLAnywhere に関するマニュアルを参照してください。

PHP の外部呼び出し

コンポーネント	Windows	Linux	Mac OS X
PHP のインストール (サードパーティ)	php.exe	php	php
PHP のランチャ	phpenv.php	phpenv.php	phpenv.php
ブリッジ	dbextenv*17*.dll	libdbextenv*17*_r.so	libdbextenv*17*_r.dylib
PHP 外部環境モジュール	php-5.x.0_sqlanywhere_extenv*17*.dll	php-5.x.0_sqlanywhere_extenv*17*_r.so	ソースコードからビルド
DBLIB (スレッド)	dblib*17*.dll	libdblib*17*_r.so	libdblib*17*_r.dylib
スレッドサポートライブラリ	N/A	libdbtasks*17*_r.so	libdbtasks*17*_r.dylib

ビルド済みバージョンの PHP 拡張および外部環境モジュールは、http://scn.sap.com/docs/DOC-40537 からダウンロードします。

SQL Anywhere PHP 拡張の使用に必要なファイルが他にもあります。SQL Anywhere PHP 拡張に関するマニュアルを参照してください。

関連情報

Embedded SQL クライアントの配備 [825 ページ]
ODBC クライアントの配備 [816 ページ]
JDBC クライアントの配備 [827 ページ]

DBD::SQLAnywhere [431 ページ]
SQL Anywhere PHP 拡張 [451 ページ]
ESQL 外部環境と ODBC 外部環境 [403 ページ]
Java 外部環境 [411 ページ]
JavaScript 外部環境 [416 ページ]
CLR 外部環境 [399 ページ]
Perl 外部環境 [421 ページ]
PHP 外部環境 [425 ページ]

1.23.14 暗号化の配備

SQL Anywhere は、パスワードを使用したセキュアなクライアントログインおよびクライアントとデータベースサーバ間のネットワークトラフィック (TLS、HTTPS) の RSA 暗号化、データベースとデータベーステーブルの AES 暗号化、および暗号化機能をサポートしています。

次の表に、暗号化をサポートするコンポーネントをまとめます。

暗号化オプション	暗号化タイプ	モジュールに付属するファイル	ライセンス設定の可否
データベースの暗号化	AES	dbrsa*17*.dll libdbrsa*17*.so libdbrsa*17*_r.so libdbrsa*17*.dylib libdbrsa*17*_r.dylib	付属 [1]
データベースの暗号化	FIPS 認定の AES	dbfips*17*.dll、libeay32.dll、ssleay32.dll、msvcr90.dll (64 ビットバージョンは msvcr100.dll) libdbfips*17*_r.so、libcrypto.so、libssl.so (Linux のみ)	別途ライセンスが必要 [2]
トランスポートレイヤセキュリティ (TLS)、HTTPS、セキュアログイン、暗号化機能	RSA	dbrsa*17*.dll libdbrsa*17*.so libdbrsa*17*_r.so libdbrsa*17*.dylib libdbrsa*17*_r.dylib	付属 [1]

暗号化オプション	暗号化タイプ	モジュールに付属するファイル	ライセンス設定の可否
トランスポートレイヤセキュリティ (TLS)、HTTPS、セキュアログイン、暗号化機能	FIPS 認定の RSA	dbfips*17*.dll、libeay32.dll、ssleay32.dll、msvcr90.dll (64 ビットバージョンは msvcr100.dll) libdbfips*17*_r.so、libcrypto.so、libssl.so (Linux のみ)	別途ライセンスが必要 [2]

[1] AES および RSA による暗号化はソフトウェアに付属しており、別途ライセンスは不要ですが、ライブラリは FIPS 認定ではありません。

[2] FIPS 認定 AES および RSA 暗号化ライブラリは個別に注文する必要があります。

1.23.15 LDAP の配備

SQL Anywhere は、LDAP (Lightweight Directory Access Protocol) ユーザ認証をサポートしています。これにより、クライアントアプリケーションは、LDAP サーバに認証を受けるためにユーザ ID およびパスワード情報をデータベースサーバに送信できるようになります。

次の表に、LDAP ユーザ認証の配備に必要なドライバファイルを示します。

オペレーティングシステム	32 ビット	64 ビット
Windows	dbldap*17*.dll	dbldap*17*.dll
FIPS サポート付き Windows	dbldapfips*17*.dll	dbldapfips*17*.dll
Linux、Unix (スレッド)	libdbldap*17*_r.so (Linux のみ)	libdbldap*17*_r.so
Linux、Unix (FIPS サポート付きスレッド)	libdbldapfips*17*_r.so (Linux のみ)	libdbldapfips*17*_r.so
Linux、Unix (非スレッド)	libdbldap*17*.so (Linux のみ)	libdbldap*17*.so

LDAP ユーザ認証は、Mac OS X ではサポートされていません。

> **注記**
>
> ソフトウェアの旧リリースでは、配備に必要な Linux および UNIX ファイルは、15.7.0.11 ではなく 15.7.0.4 で終わります。

1.23.16 組み込みデータベースアプリケーションの配備

埋め込みデータベースアプリケーションは、アプリケーションとデータベースが同じコンピュータにあるアプリケーションです。

次に、埋め込みデータベースアプリケーションのコンポーネントを示します。

クライアントアプリケーション

SQL Anywhere クライアントコンポーネントが含まれています。

データベースサーバ

SQL Anywhere パーソナルデータベースサーバ

データベースファイル

アプリケーションが使用するデータを保管するデータベースファイルを配備します。

SQL Remote

アプリケーションで SQL Remote レプリケーションを使用する場合は、SQL Remote Message Agent を配備します。

このセクションの内容:

パーソナルサーバの配備 [872 ページ]
パーソナルサーバを使用するアプリケーションを配備する場合は、クライアントアプリケーションコンポーネントとデータベースサーバコンポーネントの両方を配備する必要があります。

データベースユーティリティの配備 [873 ページ]
データベースユーティリティ (dbbackup など) を配備する場合、一部の追加ファイルが必要になります。

バージョン 9 以前のデータベースのアンロードサポートの配備 [874 ページ]
アプリケーションによっては、バージョン 9.0 以前のデータベースを現在の形式に変換できる機能が必要です。

SQL Remote の配備 [875 ページ]
アプリケーションによっては、SQL Remote Message Agent を必要とするものもあります。

関連情報

データベースサーバの配備 [860 ページ]
クライアントアプリケーションを配備するための要件 [805 ページ]

1.23.16.1 パーソナルサーバの配備

パーソナルサーバを使用するアプリケーションを配備する場合は、クライアントアプリケーションコンポーネントとデータベースサーバコンポーネントの両方を配備する必要があります。

言語リソースライブラリ (`dblgen`*17*`.dll`) は、クライアントとサーバ間で共有されます。このファイルのコピーは 1 つしか必要ありません。

SQL Anywhere のインストール動作に従い、クライアントファイルとサーバファイルを同じディレクトリにインストールします。

1.23.16.2 データベースユーティリティの配備

データベースユーティリティ (dbbackup など) を配備する場合、一部の追加ファイルが必要になります。

アプリケーションでデータベースユーティリティを配備する場合、次のサポートファイルを含めます。

説明	Windows	Linux/UNIX	Mac OS X
データベースツールライブラリ	dbtool*17*.dll	libdbtool*17*_r.so、libdbtasks*17*_r.so	libdbtool*17*_r.dylib、libdbtasks*17*_r.dylib
インタフェースライブラリ	dblib*17*.dll	libdblib*17*_r.so	libdblib*17*_r.dylib
言語リソースライブラリ	dblg[LL]*17*.dll	dblg[LL]*17*.res	dblgen*17*.res
[接続] ウィンドウ	dbcon*17*.dll		
バージョン 9 以前の物理ストアライブラリ	dboftsp.dll	libdboftsp_r.so	N/A

注記

- 言語リソースライブラリファイルも含めてください。上記の表には、指定が [LL] であるファイルが示されています。メッセージファイルは複数あり、それぞれが異なる言語をサポートしています。異なる言語のサポートをインストールするには、それらの言語のリソースファイルを追加してください。[LL] を言語コード (**en**、**de**、**ja** など) に置き換えます。
- Linux/UNIX 上で動作する非マルチスレッドアプリケーションには、libdbtasks*17*.so と libdblib*17*.so を使用できます。
- Mac OS X 上で動作する非マルチスレッドアプリケーションには、libdbtasks*17*.dylib と libdblib*17*.dylib を使用できます。
- Windows の場合、エンドユーザが独自のデータソースを作成する場合や、データベースに接続するときにユーザ ID とパスワードを入力する必要がある場合、またはその他の理由で [接続] ウィンドウが必要な場合は、*[SQL Anywhere の ODBC 設定]* および *[SQL Anywhere への接続]* ウィンドウのサポートコード (dbcon*17*.dll) が必要です。
- 一部のユーティリティ (dblog、dbtran、dberase) では、バージョン 10.0.0 以前のソフトウェアで作成されたログファイルにアクセスするために、バージョン 9 以前の物理ストアライブラリが必要です。これらのユーティリティを配備しない場合は、このライブラリは不要です。
- バージョン 9 以前のデータベースのアンロードをサポートする必要がある場合は、データベースアンロードユーティリティ dbunload と他の一部のコンポーネントが必要になります。これらのコンポーネントについては、別の場所で記述されています。
- *dbinit* ユーティリティを使用してデータベースを作成するには、パーソナルデータベースサーバ (dbeng17) が必要です。パーソナルデータベースサーバは、その他のデータベースサーバが実行されていない場合にローカルコンピュータで *SQL Central* からデータベースを作成する場合にも必要です。

関連情報

データベースサーバの配備 [860 ページ]
バージョン 9 以前のデータベースのアンロードサポートの配備 [874 ページ]

1.23.16.3 バージョン 9 以前のデータベースのアンロードサポートの配備

アプリケーションによっては、バージョン 9.0 以前のデータベースを現在の形式に変換できる機能が必要です。

アプリケーションはバージョン 9.0 以前のデータベースの変換を必要とする場合、データベースアンロードユーティリティ dbunload を以下のファイルと合わせて含める必要があります。

説明	Windows	Linux/UNIX	Mac OS X
10.0 以前のデータベースのアンロードサポート	dbunlspt.exe	dbunlspt	dbunlspt
メッセージリソースライブラリ	dbus[LL].dll	dbus[LL].res	dbus[LL].res
アンロードスクリプトファイル	optdeflt.sql	optdeflt.sql	optdeflt.sql
アンロードスクリプトファイル	opttemp.sql	opttemp.sql	opttemp.sql
アンロードスクリプトファイル	unloadold.sql	unloadold.sql	unloadold.sql

上記の表には、指定が [LL] であるファイルが示されています。メッセージファイルは複数あり、それぞれが異なる言語をサポートしています。異なる言語のサポートをインストールするには、それらの言語のリソースファイルを含めてください。メッセージファイルは次のとおりです。

Windows メッセージファイル

dbusde.dll	ドイツ語
dbusen.dll	英語
dbuses.dll	スペイン語
dbusfr.dll	フランス語
dbusit.dll	イタリア語
dbusja.dll	日本語
dbusko.dll	韓国語
dbuslt.dll	リトアニア語
dbuspl.dll	ポーランド語
dbuspt.dll	ポルトガル語
dbusru.dll	ロシア語
dbustw.dll	中国語 (繁体文字)
dbusuk.dll	ウクライナ語
dbuszh.dll	中国語 (簡体文字)

Linux メッセージファイル

dbusde_iso_1.res、dbusde_utf8.res、dbusen.res	ドイツ語
dbusen.res	英語
dbusja_eucjis.res、dbusja_sjis.res、dbusja_utf8.res	日本語
dbuszh_cp936.res、dbuszh_eucgb.res、dbuszh_utf8.res	中国語

これらのファイルは、SQL Anywhere のローカライズ版に含まれます。

これらのファイルに加え、データベースユーティリティ配備に要求されるファイルも必要です。

関連情報

データベースユーティリティの配備 [873 ページ]

1.23.16.4 SQL Remote の配備

アプリケーションによっては、SQL Remote Message Agent を必要とするものもあります。

SQL Remote Message Agent を配備する場合は、次のファイルを含める必要があります。

説明	Windows	Linux/Solaris	Mac OS X
Message Agent	dbremote.exe	dbremote	dbremote
エンコード/デコードライブラリ	dbencod*17*.dll	libdbencod*17*_r.so.1	libdbencod*17*_r.dylib
FILE メッセージリンクライブラリ[1]	dbfile*17*.dll	libdbfile*17*_r.so.1	libdbfile*17*_r.dylib
FTP メッセージリンクライブラリ[1]	dbftp*17*.dll	libdbftp*17*_r.so.1	libdbftp*17*_r.dylib
言語リソースライブラリ	dblg[LL]*17*.dll	dblg[LL]*17*.res	dblg[LL]*17*.res
インタフェースライブラリ	dblib*17*.dll	libdblib*17*_r.so.1	libdblib*17*_r.dylib
SMTP メッセージリンクライブラリ[1]	dbsmtp*17*.dll	libdbsmtp*17*_r.so.1	libdbsmtp*17*_r.dylib
データベースツールライブラリ	dbtool*17*.dll	libdbtool*17*_r.so.1	libdbtool*17*_r.dylib
スレッドサポートライブラリ	N/A	libdbtasks*17*_r.so.1	libdbtasks*17*_r.dylib

[1] 使用するメッセージリンク用のライブラリだけを配備します。

言語リソースライブラリファイルも含めてください。上記の表には、指定が [LL] であるファイルが示されています。メッセージファイルは複数あり、それぞれが異なる言語をサポートしています。異なる言語のサポートをインストールするには、それらの言語のリソースファイルを追加してください。[LL] を言語コード (**en**、**de**、**ja** など) に置き換えます。

SQL Anywhere のインストール動作に従い、SQL Remote ファイルを SQL Anywhere ファイルと同じディレクトリにインストールしてください。

1.24 このマニュアルの印刷、再生、および再配布

次の条件に従うかぎり、このマニュアルの全部または一部を使用、印刷、再生、配布することができます。

1. ここに示したものとそれ以外のすべての版権と商標の表示をすべてのコピーに含めること。
2. マニュアルに変更を加えないこと。
3. SAP 以外の人間がマニュアルの著者または情報源であるかのように示す一切の行為をしないこと。

ここに記載された情報は事前の通知なしに変更されることがあります。

重要免責事項および法的情報

コードサンプル

この文書に含まれるソフトウェアコード及び / 又はコードライン / 文字列 (「コード」) はすべてサンプルとしてのみ提供されるものであり、本稼動システム環境で使用することが目的ではありません。「コード」は、特定のコードの構文及び表現規則を分かりやすく説明及び視覚化することのみを目的としています。SAP は、この文書に記載される「コード」の正確性及び完全性の保証を行いません。更に、SAP は、「コード」の使用により発生したエラー又は損害が SAP の故意又は重大な過失が原因で発生させたものでない限り、そのエラー又は損害に対して一切責任を負いません。

アクセシビリティ

この SAP 文書に含まれる情報は、公開日現在のアクセシビリティ基準に関する SAP の最新の見解を表明するものであり、ソフトウェア製品のアクセシビリティ機能の確実な提供方法に関する拘束力のあるガイドラインとして意図されるものではありません。SAP は、この文書に関する一切の責任を明確に放棄するものです。ただし、この免責事項は、SAP の意図的な違法行為または重大な過失による場合は、適用されません。さらに、この文書により SAP の直接的または間接的な契約上の義務が発生することは一切ありません。

ジェンダーニュートラルな表現

SAP 文書では、可能な限りジェンダーニュートラルな表現を使用しています。文脈により、文書の読者は「あなた」と直接的な呼ばれ方をされたり、ジェンダーニュートラルな名詞 (例: 「販売員」又は「勤務日数」) で表現されます。ただし、男女両方を指すとき、三人称単数形の使用が避けられない又はジェンダーニュートラルな名詞が存在しない場合、SAP はその名詞又は代名詞の男性形を使用する権利を有します。これは、文書を分かりやすくするためです。

インターネットハイパーリンク

SAP 文書にはインターネットへのハイパーリンクが含まれる場合があります。これらのハイパーリンクは、関連情報を見いだすヒントを提供することが目的です。SAP は、この関連情報の可用性や正確性又はこの情報が特定の目的に役立つことの保証は行いません。SAP は、関連情報の使用により発生した損害が、SAP の重大な過失又は意図的な違法行為が原因で発生したものでない限り、その損害に対して一切責任を負いません。すべてのリンクは、透明性を目的に分類されています (http://help.sap.com/disclaimer を参照)。

go.sap.com/registration/contact.html