SQL Anywhere® サーバ
プログラミング

バージョン 16.0

2013 年 2 月

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

バージョン 16.0
2013 年 2 月

© 2013 SAP AG or an SAP affiliate company.All rights reserved.

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) マニュアルの全部または一部にかかわらず、すべてのコピーにこの情報またはマニュアル内のその他の版権と商標の表示を含めること。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。

SAP AG およびディストリビュータが販売しているソフトウェア製品には、他のソフトウェアベンダー独自のソフトウェアコンポーネントが含まれているものがあります。国内製品の仕様は変わることがあります。

これらの資料は SAP AG および関連会社 (SAP グループ) が情報のみを目的として提供するものであり、いかなる種類の表明または保証も行うものではなく、SAP グループはこの資料に関する誤りまたは脱落について責任を負わないものとします。SAP グループの製品およびサービスに関する保証は、かかる製品およびサービスに付属している明確な保証文書がある場合、そこで明記されている保証に限定されます。ここに記載されているいかなる内容も、追加保証を構成するものとして解釈されるものではありません。

ここに記載された SAP および他の SAP 製品とサービス、ならびに対応するロゴは、ドイツおよび他の国における SAP AG の商標または登録商標です。http://www.sap.com/corporate-en/legal/copyright/index.epx#trademark を参照してください。

目次

はじめに ix

SQL を使用したアプリケーション開発 1

アプリケーションでの SQL 文の実行 1
準備文 2
カーソルの使用法 5
カーソルの原則 8
カーソルタイプ 14
SQL Anywhere のカーソル 16
結果セット記述子 34
アプリケーション内のトランザクション 36

.NET アプリケーションプログラミング 41

SQL Anywhere .NET データプロバイダ 41
.NET データプロバイダチュートリアル 78
SQL Anywhere ASP.NET プロバイダ 97
SQL Anywhere .NET API リファレンス 105

OLE DB と ADO の開発 343

OLE DB 343
SQL Anywhere を使用した ADO プログラミング 344
OLE DB 接続パラメータ 351
OLE DB 接続プーリング 353
Microsoft リンクサーバ 354
サポートされる OLE DB インタフェース 358
OLE DB プロバイダの登録 363

ODBC サポート 365

ODBC アプリケーションの開発要件 365
ODBC アプリケーションの開発 367

ODBC のサンプル 372
ODBC ハンドル 374
ODBC 接続関数 376
ODBC によって変更されるサーバオプション 379
SQLSetConnectAttr 拡張接続属性 380
Windows DllMain 関数の考慮事項 382
SQL 文の実行方法 383
64 ビット ODBC での考慮事項 387
データアラインメントの要件 391
ODBC アプリケーションの結果セット 393
ストアドプロシージャの考慮事項 397
ODBC エスケープ構文 399
ODBC のエラー処理 401

データベースにおける Java 405

データベース内の Java について 405
データベース内の Java の FAQ 405
Java のエラー処理 407
チュートリアル：データベース内の Java の使用 407
Java クラスをデータベースにインストールする方法 413
データベース内の Java クラスの特殊な機能 417
Java VM を起動し、停止する方法 420
Java VM でのシャットダウンフック 420

JDBC サポート 423

JDBC アプリケーション 423
JDBC ドライバ 424
JDBC プログラムの構造 426
クライアント側 JDBC 接続とサーバ側 JDBC 接続の違い 426
SQL Anywhere JDBC ドライバ 427
jConnect JDBC ドライバ 428
JDBC クライアントアプリケーションからの接続 431
サーバ側 JDBC クラスから接続を確立する方法 435
JDBC 接続についての注意 438

JDBC を使用したデータへのアクセス 439
JDBC コールバック 448
JDBC エスケープ構文 452
JDBC 4.0 API のサポート 455

OData のサポート 457

OData サーバのアーキテクチャ 457
OData サーバの制限事項 458
OData サーバのセキュリティの考慮事項 458
OData サーバを設定する方法 459
OData の操作のために HTTP サーバを設定する方法 463
OData プロデューサのサービスモデルを作成する方法 465
OData サーバの実行とクライアントの設定 466
OData サーバユーティリティ (*dbosrv16.exe*) 466
OData サーバ停止ユーティリティ (*dbostop.exe*) 467

Embedded SQL 469

開発プロセスの概要 470
SQL プリプロセッサ 471
対応コンパイラ 476
Embedded SQL ヘッダファイル 477
インポートライブラリ 477
サンプル Embedded SQL プログラム 478
Embedded SQL プログラムの構造 479
Windows での DBLIB の動的ロード 479
サンプル Embedded SQL プログラム 481
Embedded SQL のデータ型 485
Embedded SQL のホスト変数 489
SQLCA (SQL Communication Area) 498
静的 SQL と動的 SQL 503
SQLDA (SQL descriptor area) 506
Embedded SQL を使用してデータをフェッチする方法 515
Embedded SQL を使用して long 値を送信し、取得する方法 523
Embedded SQL での簡単なストアドプロシージャ 528

Embedded SQL を使用した要求管理 531
Embedded SQL を使用したデータベースのバックアップ 531
ライブラリ関数のリファレンス 532
Embedded SQL 文のまとめ 561

C/C++ 用の SQL Anywhere データベース API 565

SQL Anywhere C API のサポート 565
SQL Anywhere C API リファレンス 566

SQL Anywhere 外部呼び出しインタフェース 601

外部呼び出しを使ったプロシージャと関数 601
外部関数のプロトタイプ 603
外部関数呼び出しインタフェースのメソッド 612
データ型の処理 616
外部ライブラリをアンロードする方法 618

SQL Anywhere 外部環境のサポート 619

CLR 外部環境 620
ESQL 外部環境と ODBC 外部環境 623
Java 外部環境 632
PERL 外部環境 638
PHP 外部環境 642

Perl DBI サポート 649

DBD::SQLAnywhere 649
Windows での DBD::SQLAnywhere のインストール 649
UNIX と Mac OS X での DBD::SQLAnywhere のインストール 651
DBD::SQLAnywhere を使用する Perl スクリプト 652

Python サポート 659

sqlanydb 659
Windows での Python サポートのインストール 660

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

UNIX と Mac OS X での Python サポートのインストール 660
sqlanydb を使用する Python スクリプト 661

PHP サポート 667

SQL Anywhere PHP 拡張 667
SQL Anywhere PHP API リファレンス 682

Ruby サポート 713

SQL Anywhere の Ruby API サポート 713
SQL Anywhere Ruby API リファレンス 721

Sybase Open Client のサポート 741

Open Client アーキテクチャ 742
Open Client アプリケーション作成に必要なもの 742
Open Client データ型マッピング 743
Open Client アプリケーションでの SQL 744
SQL Anywhere における Open Client の既知の制限 747

HTTP Web サービス 749

HTTP Web サーバとしての SQL Anywhere 749
Web クライアントを使用した Web サービスへのアクセス 786
Web サービスエラーコードリファレンス 823
HTTP Web サービスの例 825

3 層コンピューティングと分散トランザクション 855

3 層コンピューティングのアーキテクチャ 855
分散トランザクション 859

データベースツールインタフェース (DBTools) 861

DBTools インポートライブラリ 862
DBTools ライブラリの初期化とファイナライズ 862

DBTools 関数呼び出し 863
コールバック関数 863
バージョン番号と互換性 865
ビットフィールド 865
DBTools の例 866
SQL Anywhere データベースツール C API リファレンス 868
ソフトウェアコンポーネントの終了コード 956

データベースおよびアプリケーションの配備 959

配備の種類 959
ファイルの配布方法 960
インストールディレクトリとファイル名 960
Deployment ウィザード 964
SQL Anywhere インストーラを使用したサイレントインストール 968
クライアントアプリケーションを配備するための要件 972
管理ツールの配備 1004
マニュアルの配備 1026
データベースサーバの配備 1027
Windows での DLL の登録 1034
外部環境のサポートの配備 1034
暗号化の配備 1037
組み込みデータベースアプリケーションの配備 1038

索引 1043

はじめに

このマニュアルでは、C、C++、Java、Perl、PHP、Python、Ruby、および Visual Basic や Visual C# などの .NET プログラミング言語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。ADO.NET、OLEDB、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。

SQL を使用したアプリケーション開発

この項では、アプリケーションでの SQL の使用について説明します。

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';
```

- **Sybase Open Client**　Sybase 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。この項では JDBC についても一部説明します。

参照

- 「SQL Anywhere .NET データプロバイダ」41 ページ
- 「ODBC サポート」365 ページ
- 「JDBC サポート」423 ページ
- 「Embedded SQL」469 ページ
- 「Sybase Open Client のサポート」741 ページ
- 「ストアドプロシージャ、トリガ、バッチ、ユーザ定義関数」『SQL Anywhere サーバ SQL の使用法』
- 「JDBC サポート」423 ページ

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次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する。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する。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する。

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する (たとえば、1 つのテーブルに多くのローを挿入する) 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。

1. **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。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

2. **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。

3. **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だけ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。SQL Anywhere の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。

参照

● 「カーソルの使用法」5 ページ

準備文の概要

この項では準備文の使用法についての簡単な概要を説明します。一般的な手順は同じですが、詳細はインタフェースによって異なります。異なるインタフェースで準備文の使い方を比較すると、違いがはっきりします。

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、*%SQLANYSAMP16%¥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、*%SQLANYSAMP16%¥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 のマニュアルと「準備文の実行」385 ページ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 *%SQLANYSAMP16% ¥SQLAnywhere¥JDBC¥JDBCExample.java* を参照してください。

JDBC での準備文の使用については、「より効率的なアクセスのために準備文を使用する方法」442 ページ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。

Embedded SQL 準備文の詳細については、「PREPARE 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』を参照してください。

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 型パラメータで ct_dynamic を使用して文を実行します。
4. CS_DEALLOC 型パラメータで 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 での準備文の使用については、「Open Client アプリケーションでの SQL」744 ページを参照してください。

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。各インタフェースで使用可能なカーソルの種類のリストについては、「カーソルの可用性」15 ページを参照してください。

SQL Anywhere には、接続用に使用す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」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_describe_cursor システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_copy_cursor_to_temp_table システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

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する。

●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、削除を実行する。

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。

異なるプログラミングインタフェースで使用可能なカーソルの種類の詳細については、「カーソルの可用性」15 ページを参照してください。

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。カーソルを使うと 1 回につき 1 つのローのデータを調べて操作できます。SQL Anywhere のカーソルは、クエリ結果内で前方や後方への移動をサポートします。

カーソル位置

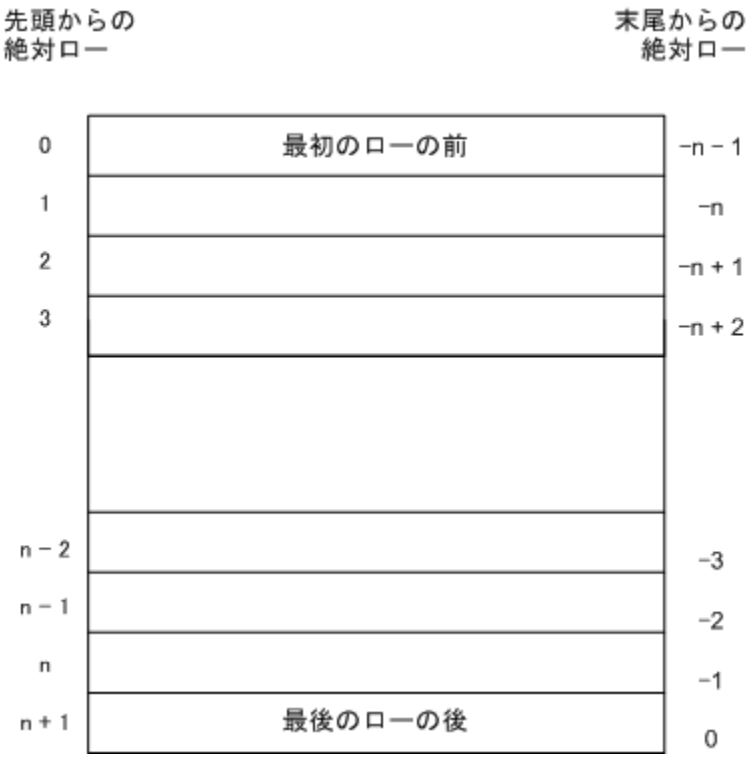
カーソルは、次の場所に置くことができます。

●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

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0



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。

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。

カーソルの原則

ADO.NET、ODBC、JDBC、または Open Client 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。

   簡単なフェッチを行うと、結果セット内の次のローへカーソルが移動しますが、SQL Anywhere では結果セットでより複雑な移動が可能です。

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、文を準備する必要があります。

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。

   簡単なフェッチを行うと、結果セット内の次のローへカーソルが移動しますが、SQL Anywhere では結果セットでより複雑な移動が可能です。どのフェッチ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。

参照

- 「準備文」2 ページ
- 「DECLARE CURSOR 文 [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』
- 「OPEN 文 [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』
- 「FETCH 文 [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』
- 「Embedded SQL を使用してデータをフェッチする方法」515 ページ
- 「CLOSE 文 [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』
- 「DROP STATEMENT 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』

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。

注意
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、SQL Anywhere はカーソル内の予測可能な位置に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、挿入したローがまったく表示されないことがあります。SQL Anywhere では、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こうしたことが起こります。

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。

参照

- 「ヒント：クエリ処理におけるワークテーブルの使用 (All-rows 最適化ゴールの使用)」『SQL Anywhere サーバ SQL の使用法』

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、カーソルの動作について次のように設定できます。

- **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。

- **保持**　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 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。

参照

- 「isolation_level オプション」『SQL Anywhere サーバ データベース管理』

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. 「**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。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

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。SADataReader.Read メソッド [SQL Anywhere .NET]248 ページを参照してください。

- **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。

  ODBC でのカーソルの使用については、「ODBC アプリケーションの結果セット」393 ページを参照してください。

- **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。

  JDBC での ResultSet オブジェクトの使用については、「Java から結果セットを返す方法」446 ページを参照してください。

- **Embedded SQL**　FETCH 文が同じ操作を実行します。

  Embedded SQL でカーソルを使用する方法の詳細については、「Embedded SQL でのカーソル」517 ページを参照してください。

- **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。

  Open Client アプリケーションでのカーソルの使用については、「Open Client カーソルの管理」745 ページを参照してください。

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。

複数ローのフェッチ

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」とも呼びます。

複数ローフェッチの使用

- ODBC では、SQL_ATTR_ROW_ARRAY_SIZE 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。

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。この方法だと、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。

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、この値は、FETCH ABSOLUTE -1 cursor-name を使用して取得できます。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。

参照

- 「FROM 句」『SQL Anywhere サーバ SQL リファレンス』

更新可能な文

この項では、SELECT 文の句が更新可能な文とカーソルに与える影響について説明します。

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、SQL Anywhere には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 SQL Anywhere では、FOR UPDATE 構文は同時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、SQL Anywhere では次の 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。

参照

- 「DECLARE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「FOR 文」『SQL Anywhere サーバ SQL リファレンス』

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。

カーソルタイプ

この項では、SQL Anywhere のカーソルと、SQL Anywhere がサポートしているプログラミングインタフェースから利用できるオプションの間で行うマッピングについて説明します。

参照

- 「SQL Anywhere のカーソル」16 ページ

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、読み込み専用、前方専用のカーソルのみを提供します。
- 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、JDBC 4.0 仕様がサポートされており、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 ドライバと同じく 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サポートされています。
- Sybase 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。

参照

- 「ODBC アプリケーションの結果セット」393 ページ
- 「SQL Anywhere のカーソルの要求」32 ページ

カーソルのプロパティ

カーソルタイプは、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。

- **一意性**　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。

SQL Anywhere では、さまざまな特性を持つカーソルを使用できます。特定のタイプのカーソルを要求すると、SQL Anywhere は、その特性を一致させるよう試みます。

特性を全部指定できない場合もあります。たとえば、SQL Anywhere の insensitive カーソルは読み込み専用です。それは、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) が指定されるからです。

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。SQL Anywhere 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 ではブックマークをサポートしますが、SQL_CURSOR_DYNAMIC と SQL_CURSOR_FORWARD_ONLY ではブックマークをサポートしていないということです。

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 ESQL ARRAY フェッチと同じ動作をします。

SQL Anywhere のカーソル

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。

SQL Anywhere では、感知性に関するさまざまな特性をカーソルに定義しています。この項では、まず感知性について説明し、次にカーソル感知性の特性について説明します。

また、「カーソル」6 ページ を読み終えていることが前提となります。

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。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

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
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。

カーソル感知性

SQL Anywhere のカーソルは、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ています。いいかえると、カーソル感知性は、表示される変更内容によって定義されるものです。

- **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。「asensitive カーソル」24 ページを参照してください。
- **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。「value-sensitive カーソル」25 ページを参照してください。

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。

参照

- 「insensitive カーソル」22 ページ
- 「sensitive カーソル」23 ページ
- 「カーソルの感知性とパフォーマンス」27 ページ

カーソル感知性の例：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く。

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

EmployeeID	Surname
102	Whitney
105	Cobb
160	Breault
...	...

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する (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する (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する。

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。

動作	結果
前のローをフェッチする	ローのオリジナルコピーを返す (102)

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

動作	結果
最初のローをフェッチする (絶対フェッチ)	ローのオリジナルコピーを返す (102)
2 番目のローをフェッチする (絶対フェッチ)	未変更のローを返す (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。

動作	結果
前のローをフェッチする	「**ローが見つかりません。**」というエラーを返す。前のローが存在しない。
最初のローをフェッチする (絶対フェッチ)	ロー 105 を返す
2 番目のローをフェッチする (絶対フェッチ)	ロー 160 を返す

- **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。

動作	結果
前のローをフェッチする	「**カーソルの現在のローがありません。**」というエラーを返す。最初のローが以前存在したカーソルにホールがある。
最初のローをフェッチする (絶対フェッチ)	「**カーソルの現在のローがありません。**」というエラーを返す。最初のローが以前存在したカーソルにホールがある。
2 番目のローをフェッチする (絶対フェッチ)	ロー 105 を返す

- **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。

カーソル感知性の例：更新されるロー

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く。

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

EmployeeID	Surname
102	Whitney
105	Cobb
160	Breault
...	...

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する (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する (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する。

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。

動作	結果
前のローをフェッチする	ローのオリジナルコピーを返す (102)
最初のローをフェッチする (絶対フェッチ)	ローのオリジナルコピーを返す (102)
2 番目のローをフェッチする (絶対フェッチ)	未変更のローを返す (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。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

動作	結果
前のローをフェッチする	SQLCODE 100 を返す。結果セットのメンバーシップは変更されたため、105 が最初のローになる。カーソルが最初のローの前の位置に移動する。
最初のローをフェッチする (絶対フェッチ)	ロー 105 を返す
2 番目のローをフェッチする (絶対フェッチ)	ロー 160 を返す

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。

動作	結果
前のローをフェッチする	SQLCODE 100 を返す。結果セットのメンバーシップは変更されたため、105 が最初のローになる。カーソルはホール上にある。つまり、ロー 105 の前にある。
最初のローをフェッチする (絶対フェッチ)	SQLCODE -197 を返す。結果セットのメンバーシップは変更されたため、105 が最初のローになる。カーソルはホール上にある。つまり、ロー 105 の前にある。
2 番目のローをフェッチする (絶対フェッチ)	ロー 105 を返す

- **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。

注意
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。

参照

- 「ローが見つかりません。」『エラーメッセージ』
- 「最後に読み込まれた後で、ローは更新されています。」『エラーメッセージ』
- 「最後に読み込まれた後で、ローは更新されています。操作はキャンセルされました。」『エラーメッセージ』
- 「カーソルの現在のローがありません。」『エラーメッセージ』

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	insensitive セマンティックは、SQL Anywhere JDBC ドライバでのみサポートされる
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。

- 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ない。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ことがある。
- 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、処理が早くなる。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する。
- insensitive カーソルは、ROLLBACK または ROLLBACK TO SAVEPOINT には影響を受けない。

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 カーソルの要求にも応じて提供される
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ない。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。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ない。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ない。

  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。ただし、ORDER BY カラムにはインデックスが存在しません。
  - ハッシュジョインを使っ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、SQL Anywhere は、アプリケーションにエラーを返すか、カーソルタイプを asensitive に変更して警告を返します。

参照

- 「ヒント：クエリ処理におけるワークテーブルの使用 (All-rows 最適化ゴールの使用)」『SQL Anywhere サーバ SQL の使用法』

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

SQL Anywhere が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高の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。

SQL Anywhere では、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。表示されるものと、されないものがあります。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が開いたときは固定ですが、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 句や ORDER BY 句に一致しないローは返される場合があります。

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	

インタフェース	カーソルタイプ	コメント
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る
Open Client と jConnect	サポートされていない	

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、SQL Anywhere が、結果セットを構成する各ローの識別情報をワークテーブルに入力します。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。「カーソル感知性の例：削除されたロー」18 ページを参照してください。

- 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、SQL Anywhere は、ローがフェッチされた時点で SQLE_ROW_UPDATED_WARNING を返します。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。「バルクオペレーションのパフォーマンスの側面」『SQL Anywhere サーバ SQL の使用法』と「最後に読み込まれた後で、ローは更新されています。」『エラーメッセージ』を参照してください。

- 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、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。「最後に読み込まれた後で、ローは更新されています。操作はキャンセルされました。」『エラーメッセージ』を参照してください。

- 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。「カーソルの現在のローがありません。」『エラーメッセージ』を参照してください。

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。「ローの修正に使用するカーソル」12 ページを参照してください。

カーソルの感知性とパフォーマンス

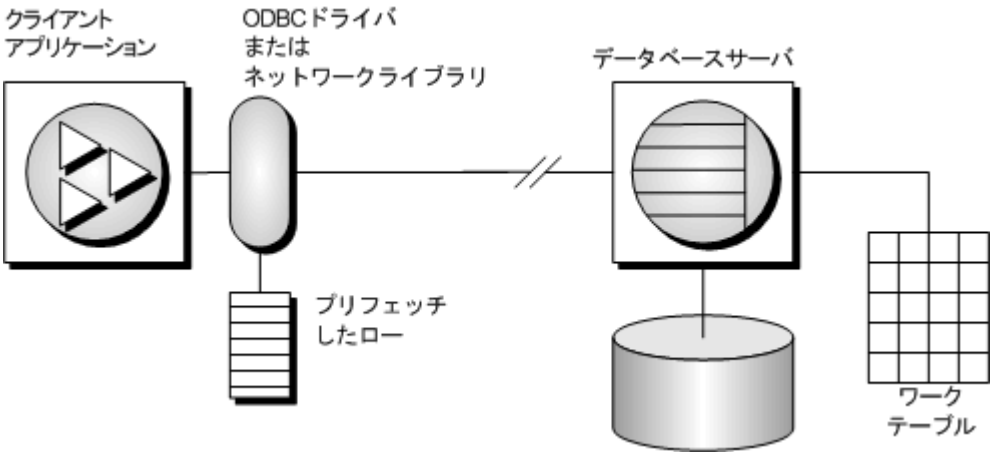
カーソルのパフォーマンスとその他のプロパティの間には、トレードオフ関係があります。特に、カーソルを更新できるようにした場合は、カーソルによるクエリの処理と配信で、パフォーマンスを制約する制限事項が課されます。また、カーソル感知性に稼働条件を設けると、カーソルのパフォーマンスが制約されることがあります。

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。

プリフェッチ

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、SQL Anywhere のクライアントライブラリが複数のローをプリフェッチします。SQL Anywhere の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。

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。

プリフェッチ制御の詳細については、「prefetch オプション」『SQL Anywhere サーバ データベース管理』を参照してください。

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、BLOCK 5 を使

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

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、常に FETCH RELATIVE 0 が同じローを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。「prefetch オプション」『SQL Anywhere サーバ データベース管理』を参照してください。

- 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る。
  - **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 操作のみ実行する (絶対フェッチ、相対フェッチ、後方フェッチは実行しない)。
- 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ない (GET DATA 文を 1 つ使用して、値を取得することはできる)。

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く。

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

ID	Quantity
300	28
301	54

ID	Quantity
302	75
...	...

2. アプリケーションが、カーソルを介して ID = 300 のローをフェッチする。

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する。

```
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) の値に更新する。

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。SQL Anywhere 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 4.0 仕様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。SQL Anywhere の場合、このロックは、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る長時間のロックです。

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。完全に直列化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。「スナップショットアイソレーションのレベルの選択」『SQL Anywhere サーバ SQL の使用法』を参照してください。

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。

SQL Anywhere のカーソルの要求

クライアントアプリケーションでカーソルタイプを要求すると、SQL Anywhere はカーソルを 1 つ返します。SQL Anywhere のカーソルは、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設定した結果の感知性によって定義されます。SQL Anywhere は、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をカーソルに指定します。

クライアントがカーソルタイプを要求すると、SQL Anywhere はそれに答えてカーソル感知性を設定します。

ADO.NET

前方専用、読み込み専用のカーソルは、SACommand.ExecuteReader を利用して使用できます。SADataAdapter オブジェクトは、カーソルの代わりにクライアント側の結果セットを使用します。SACommand クラス [SQL Anywhere .NET]129 ページを参照してください。

ADO/OLE DB と ODBC

次の表は、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	SQL Anywhere のカーソル
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。

SQL Anywhere のカーソルと動作については、「SQL Anywhere のカーソル」16 ページを参照してください。

ODBC でのカーソルタイプの要求方法については、「ODBC カーソル特性」394 ページを参照してください。

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。

JDBC

JDBC 4.0 仕様では、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 ドライバはこれらの JDBC 仕様に準拠しており、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、データベースサーバが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。「sensitive カーソル」23 ページを参照してください。

jConnect の場合は、JDBC 2.0 仕様に従って別のタイプのカーソルを作成する場合は API をサポートしていますが、基本のプロトコル (TDS) ではデータベースサーバ上でサポートしているの

は forward-only と read-only asensitive カーソルのみです。TDS プロトコルでは文の結果セットをブロック単位でバッファに格納するため、すべての jConnect カーソルは asensitive です。バッファに格納された結果のブロックは、スクロール動作がサポートされている insensitive または sensitive カーソルタイプを使用してアプリケーションでスクロールする必要がある場合に、スクロールされます。アプリケーションがキャッシュされた結果セットの先頭を越えて後方にスクロールすると、文は再実行されます。この場合、次の実行までにデータが変更されていると、データに矛盾が生じる可能性があります。

Embedded SQL

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。次の表は、各要求に応じて設定されるカーソル感知性を示しています。

カーソルタイプ	SQL Anywhere のカーソル
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。

Open Client

jConnect の場合と同様、Open Client の基本のプロトコル (TDS) は、forward-only、read-only、asensitive カーソルのみサポートしています。

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。ユーザが表示するカラムを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、アプリケーションが検索する情報を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。「SQLDA (SQL descriptor area)」506 ページを参照してください。
- 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 を使ってこ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。

- 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。

  DESCRIBE 文の入力パラメータと出力パラメータの詳細については、「DESCRIBE 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』を参照してください。

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。この項ではアプリケーションのトランザクションの一面について説明します。

トランザクションの詳細については、「トランザクションと独立性レベル」『SQL Anywhere サーバ SQL の使用法』を参照してください。

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。

パフォーマンスに与えるオートコミットモードの影響については、「ヒント : オートコミットモードをオフにする」『SQL Anywhere サーバ SQL の使用法』 を参照してください。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

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。「オートコミット実装の詳細」38 ページを参照してください。

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。「Transaction 処理」64 ページを参照してください。

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 (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。

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 (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 ON にするには、次の文を実行して chained データベースオプション (サーバ側オプション) を Off に設定します。

```
SET OPTION chained='Off';
```

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 (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。オートコミットモードを OFF にするには、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 (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 ON にするには、次の文を実行して chained データベースオプション (サーバ側オプション) を Off に設定します。

```
SET OPTION chained='Off';
```

クライアント側でコミットを制御するインタフェースを使用している場合、chained データベースオプション (サーバ側オプション) がアプリケーションのパフォーマンスや動作に影響する場合があります。サーバの連鎖モードを設定することはおすすめしません。

参照

- 「オートコミットまたは手動コミットモード」36 ページ

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。

オートコミットモードは、次のいずれかの方法で実装できます。

- **クライアント側オートコミット**　アプリケーションがオートコミットを使用すると、各 SQL 文の実行後、クライアントライブラリが COMMIT 文を送信します。

  ADO.NET、ADO/OLE DB、ODBC、PHP、SQL Anywhere JDBC のドライバアプリケーションでは、クライアント側からコミットの動作を制御します。

- **サーバ側オートコミット**　アプリケーションで連鎖モードを OFF にすると、データベースサーバは各 SQL 文の結果をコミットします。Sybase jConnect JDBC ドライバの場合、この動作は chained データベースオプションによって制御されます。

  Embedded SQL、jConnect ドライバ、Open Client のアプリケーションでは、サーバ側でのコミット動作を操作します (たとえば、chained オプションを設定します)。

ストアドプロシージャやトリガなどの複雑な文では、クライアント側オートコミットとサーバ側オートコミットには違いがあります。クライアント側では、ストアドプロシージャは単一文であるため、オートコミットはプロシージャがすべて実行された後に単一のコミット文を送信します。データベースサーバ側から見た場合、ストアドプロシージャは複数の SQL 文で構成されているため、サーバ側オートコミットはプロシージャ内の各 SQL 文の結果をコミットします。

注意
クライアント側の実装とサーバ側の実装を混在させないでください SQL Anywhere ADO.NET、OLE DB、ODBC、PHP、JDBC のアプリケーションでは、chained オプションとオートコミットオプションの設定を併用しないでください。

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。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

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。「isolation_level オプション」『SQL Anywhere サーバ データベース管理』を参照してください。

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。

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 に設定されている。

●カーソルが WITH HOLD で開かれている。Open Client と JDBC ではデフォルト。

この 2 つのどちらかが真の場合、カーソルは COMMIT 時に開いたままです。

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。しかし、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。サポートされている独立性レベルの詳細については、「独立性レベルと一貫性」『SQL Anywhere サーバ SQL の使用法』と「isolation_level オプション」『SQL Anywhere サーバ データベース管理』を参照してください。

.NET アプリケーションプログラミング

この項では、.NET で SQL Anywhere を使用し、SQL Anywhere .NET データプロバイダの API を含める方法について説明します。

SQL Anywhere .NET データプロバイダ

この項では、Visual Studio プロジェクトでの SQL Anywhere .NET データプロバイダの使用、データベースとの接続、データベーステーブルからのローのフェッチ、挿入、更新、削除、ストアドプロシージャの呼び出し、トランザクションの使用、基本的なエラー処理に関するヒントなど、.NET サポートについて説明します。

SQL Anywhere の .NET サポート

ADO.NET は、ODBC、OLE DB、ADO について Microsoft の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。

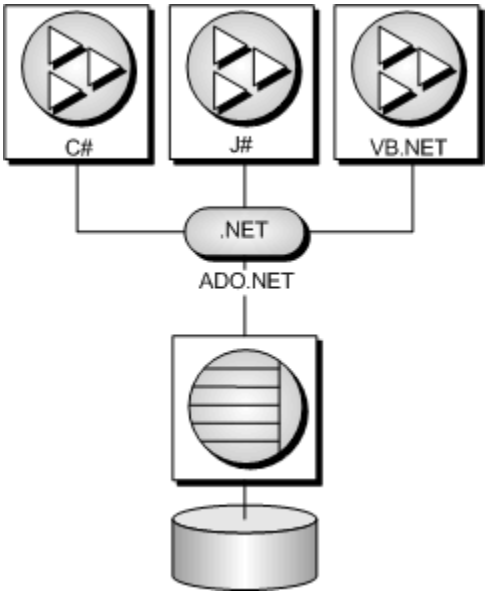
SQL Anywhere .NET データプロバイダは、iAnywhere.Data.SQLAnywhere ネームスペースを実装しており、.NET でサポートされている任意の言語 (C# や Visual Basic .NET など) でプログラムを作成したり、SQL Anywhere データベースからデータにアクセスしたりできます。

.NET データアクセスの概要については、http://msdn.microsoft.com/ja-jp/library/Ee817654%28pandp.10%29.aspx の Microsoft の「.NET データ アクセス アーキテクチャ ガイド」を参照してください。

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、ADO.NET データプロバイダを使用してアプリケーションを SQL Anywhere に接続できます。

ADO.NET データプロバイダに、組み込みの XML と Web サービス機能、Mobile Link 同期用の .NET スクリプト機能、ハンドヘルドデータベースアプリケーション開発用の Ultra Light.NET コンポーネントを組み合わせることで、SQL Anywhere を .NET フレームワークに統合できるようになります。



参照

- 「SQL Anywhere .NET データプロバイダの機能」42 ページ
- 「ネームスペース」105 ページ
- 「.NET データプロバイダチュートリアル」78 ページ

SQL Anywhere .NET データプロバイダの機能

SQL Anywhere は、3 つの異なるネームスペースを使用して Microsoft .NET Framework バージョン 2.0、3.0、3.5、4.0、4.5 をサポートしています。

- **iAnywhere.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。SQL Anywhere には、OLE DB または ODBC のオーバヘッドを加えずに SQL Anywhere 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。SQL Anywhere .NET データプロバイダは、.NET ネームスペースでは iAnywhere.Data.SQLAnywhere として表現されます。

  Microsoft .NET Compact Framework は、Microsoft .NET 用のスマートデバイス開発フレームワークです。SQL Anywhere .NET Compact Framework データプロバイダは、Windows Mobile が稼働しているデバイスをサポートしています。Compact Framework 2.0 および 3.5 がサポートされています。

  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。

ADO.NET オブジェクトモデルを使用して、特に、エンティティ手法への統合言語クエリ (LINQ) を介して、SQL Anywhere データベースの内部に格納されたデータにアクセスする方法の詳細については、www.sybase.com/detail?id=1060541 の「**SQL Anywhere ADO.NET Entity Framework のチュートリアル**」を参照してください。

- **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 SQL Anywhere OLE DB プロバイダの SAOLEDB とともに使用して、SQL Anywhere 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 SQL Anywhere ODBC ドライバとともに使用して、SQL Anywhere データベースにアクセスできます。

SQL Anywhere .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、SQL Anywhere .NET データプロバイダは、SQL Anywhere 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 SQL Anywhere 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、SQL Anywhere .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。SQL Anywhere データベースへのアクセスには SQL Anywhere .NET データプロバイダを使用することをおすすめします。

.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 の配備での 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。詳細については、「ClickOnce と .NET データプロバイダのアンマネージコード DLL」974 ページを参照してください。

- **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。

- **SimpleCE**　**[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、Windows Mobile 用の .NET Compact 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、Windows 用の .NET Framework サンプルプロジェクト。詳細については、「チュートリアル：Simple コードサンプルの使用」78 ページを参照してください。

- **SimpleXML**　ADO.NET. を介して SQL Anywhere から XML データを取得する方法を示す、Windows 用の .NET Framework サンプルプロジェクト。C#、Visual Basic、Visual C++ のサンプルが用意されています。

- **SimpleViewer**　Windows 用の .NET Framework サンプルプロジェクト。詳細については、「チュートリアル：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」87 ページを参照してください。

- **TableViewer**　SQL 文を入力および実行可能な、Windows 用の .NET Framework サンプルプロジェクト。詳細については、「チュートリアル：Table Viewer コードサンプルの使用」82 ページを参照してください。

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

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。

前提条件

この作業を実行するための前提条件はありません。

♦ タスク

1. Visual Studio を起動し、プロジェクトを開きます。

2. **[ソリューション エクスプローラー]** 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、リストをスクロールして、次のいずれかを見つけます。

   - iAnywhere.Data.SQLAnywhere for .NET 2
   - iAnywhere.Data.SQLAnywhere for .NET 3.5
   - iAnywhere.Data.SQLAnywhere for .NET 4

   注意
   Windows Mobile プラットフォームには、個別のバージョンのプロバイダがあります。Windows Mobile SQL Anywhere .NET データプロバイダの場合は、**[参照]** タブをクリックし、Windows Mobile バージョンのプロバイダを指定します。デフォルトのロケーションは、*%SQLANY16%¥CE¥Assembly¥V2* です。

4. 必要なプロバイダをクリックし、**[OK]** をクリックします。

   プロジェクトの **[ソリューション エクスプローラー]** 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 iAnywhere.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 iAnywhere.Data.SQLAnywhere
```

結果

SQL Anywhere .NET データプロバイダが、.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。

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

次の 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 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 iAnywhere.Data.SQLAnywhere.SAConnection()
```

参照

- 「SQL Anywhere .NET データプロバイダの配備」72 ページ

.NET データベースの接続例

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。

C# の SAConnection の例

次の C# コードは、SQL Anywhere の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 16 Demo");
   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");
   }
}
```

Visual Basic SAConnection の例

次の Visual Basic コードは、SQL Anywhere の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 16 Demo")
   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")
   End Try
End Sub
```

参照

- SAConnection クラス [SQL Anywhere .NET]169 ページ
- SAConnection.ConnectionString プロパティ [SQL Anywhere .NET]186 ページ
- SAConnectionStringBuilder クラス [SQL Anywhere .NET]190 ページ
- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』

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。デフォルトでは、接続プーリングは有効になっています。

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。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 16 Demo;Pooling=true;Max Pool Size=50;Min Pool Size=5"
```

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。

SQL Anywhere 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)。

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;
}
```

参照

- SAConnection.State プロパティ [SQL Anywhere .NET]189 ページ

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、次の 2 つの方法で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すすめします。

  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、SQL Anywhere 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。

詳細については、SACommand クラス [SQL Anywhere .NET]129 ページと SADataReader クラス [SQL Anywhere .NET]224 ページを参照してください。

- **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論理をアプリケーションに構築する必要があります。

  競合の詳細については、「SADataAdapter を使用するときの競合の解消」54 ページを参照してください。

  SADataAdapter オブジェクトの詳細については、SADataAdapter クラス [SQL Anywhere .NET]215 ページを参照してください。

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。

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、SQL Anywhere 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。次のいずれかの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。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

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。

C# の ExecuteReader の例

次の C# コードは、SQL Anywhere 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 16 Demo");
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();
```

Visual Basic の ExecuteReader の例

次の Visual Basic コードは、SQL Anywhere 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 16 Demo")
Dim cmd As New SACommand("SELECT Surname FROM Employees", conn)
Dim reader As SADataReader
conn.Open()
reader = cmd.ExecuteReader()
ListEmployees.BeginUpdate()
Do While (reader.Read())
   ListEmployees.Items.Add(reader.GetString(0))
Loop
ListEmployees.EndUpdate()
conn.Close()
```

C# の ExecuteScalar の例

次の C# コードは、SQL Anywhere 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 16 Demo");
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();
```

参照

- SACommand クラス [SQL Anywhere .NET]129 ページ
- SADataReader クラス [SQL Anywhere .NET]224 ページ
- SACommand.ExecuteReader メソッド [SQL Anywhere .NET]146 ページ
- SACommand.ExecuteScalar メソッド [SQL Anywhere .NET]147 ページ

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

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 16 Demo" );
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;
```

参照

- SADataReader.GetSchemaTable メソッド [SQL Anywhere .NET]240 ページ

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 関数を使用します。ExecuteNonQuery 関数は、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、「SACommand：新しく挿入したローのプライマリキー値の取得」52 ページ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。「Transaction 処理」64 ページを参照してください。

C# の ExecuteNonQuery DELETE および INSERT の例

次の例は、SQL Anywhere 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 16 Demo");
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、SQL Anywhere 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 16 Demo");
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();
```

参照

- SACommand クラス [SQL Anywhere .NET]129 ページ
- SACommand.ExecuteNonQuery メソッド [SQL Anywhere .NET]145 ページ
- SAParameter クラス [SQL Anywhere .NET]278 ページ
- SATransaction クラス [SQL Anywhere .NET]324 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

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 16 Demo" );
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();
```

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

```
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;
```

参照

- SACommand クラス [SQL Anywhere .NET]129 ページ
- SAParameter クラス [SQL Anywhere .NET]278 ページ

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 の変更内容を使用してデータベースを更新するためにデータベースに接続されている必要があります。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。これによって、後で ReadXml メソッドを使用して DataSet をロードして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変更を DataSet からデータベースに適用すると、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。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 を変更できるのは、接続が切断されている場合のみです。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論理を採用する必要があります。

アプリケーション論理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。

  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、「SADataAdapter：新しく挿入したローのプライマリキー値の取得」60 ページを参照してください。

- **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論理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すすめします。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。

トランザクションを使用して変更をデータに適用する方法の詳細については、「SACommand：ExecuteNonQuery を使用したローの挿入、削除、更新」50 ページを参照してください。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

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、エラー処理論理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。

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 16 Demo" );
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 16 Demo" );
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"];
```

参照

- SADataAdapter クラス [SQL Anywhere .NET]215 ページ

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 16 Demo" );
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 16 Demo" );
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"];
```

参照

- SADataAdapter クラス [SQL Anywhere .NET]215 ページ

SADataAdapter：Update を使用したローの挿入

C# の SADataAdapter Insert の例

次の例は、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 16 Demo" );
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;
```

参照

- SADataAdapter クラス [SQL Anywhere .NET]215 ページ

SADataAdapter：Update を使用したローの削除

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 16 Demo" );
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;
```

参照

- SADataAdapter クラス [SQL Anywhere .NET]215 ページ

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

SADataAdapter：Update を使用したローの更新

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 16 Demo" );
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;
```

参照

- SADataAdapter クラス [SQL Anywhere .NET]215 ページ

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 16 Demo" );
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"];
```

参照

- SADataAdapter クラス [SQL Anywhere .NET]215 ページ
- SAParameter クラス [SQL Anywhere .NET]278 ページ

BLOB

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 16 Demo" );
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();
```

参照

- SADataReader.GetBytes メソッド [SQL Anywhere .NET]230 ページ
- SADataReader.GetChars メソッド [SQL Anywhere .NET]231 ページ

時間値

.NET Framework には Time 構造体はありません。SQL Anywhere 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 16 Demo" );
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();
```

参照

- SADataReader.GetTimeSpan メソッド [SQL Anywhere .NET]243 ページ

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 16 Demo" );
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();
```

参照

- SACommand.ExecuteReader メソッド [SQL Anywhere .NET]146 ページ
- SACommand.ExecuteNonQuery メソッド [SQL Anywhere .NET]145 ページ
- SACommand.ExecuteScalar メソッド [SQL Anywhere .NET]147 ページ
- SAParameter クラス [SQL Anywhere .NET]278 ページ
- 「SACommand：ExecuteNonQuery を使用したローの挿入、削除、更新」50 ページ
- 「SACommand：ExecuteReader と ExecuteScalar を使用したデータのフェッチ」48 ページ

Transaction 処理

SQL Anywhere .NET データプロバイダでは、SATransaction オブジェクトを使用して文をグループ化できます。各トランザクションは COMMIT または ROLLBACK で終了します。これらは、データベースの変更内容を確定したり、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したりします。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 や 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、SQL Anywhere .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。SQL Anywhere .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 SQL Anywhere 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 SQL Anywhere .NET データプロバイダでサポートされています。分散トランザクションには、大きなパフォーマンスのオーバヘッドがあります。非分散トランザクションにデータベーストランザクションを使用することをおすすめします。

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、SQL Anywhere .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 16 Demo" );
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();
```

参照

- SACommand クラス [SQL Anywhere .NET]129 ページ
- SATransaction クラス [SQL Anywhere .NET]324 ページ
- SAIsolationLevel 列挙体 [SQL Anywhere .NET]338 ページ
- 「独立性レベルと一貫性」『SQL Anywhere サーバ SQL の使用法』
- 「クエリ時のロック」『SQL Anywhere サーバ SQL の使用法』

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。

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。

C# のエラー処理の例

次の C# コードは、SQL Anywhere の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 16 Demo");
    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");
        }
    }
}
```

Visual Basic のエラー処理の例

次の Visual Basic コードは、SQL Anywhere の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 16 Demo")
    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
    End Try
End Sub
```

参照

- SAError クラス [SQL Anywhere .NET]253 ページ
- SAException クラス [SQL Anywhere .NET]258 ページ
- SAFactory クラス [SQL Anywhere .NET]261 ページ
- 「SADataAdapter を使用するときの競合の解消」54 ページ
- 「Simple サンプルプロジェクトの説明」80 ページ
- 「Table Viewer サンプルプロジェクトの説明」85 ページ

Entity Framework のサポート

SQL Anywhere .NET データプロバイダでは Entity Framework 5.0 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 を使用するには、Microsoft の NuGet Package Manager を使用して 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。C# または VB.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、データの注釈や 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したり既存のデータベースにマッピングしたりできます。

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 iAnywhere.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 16 Demo" ) )
            {
                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
iAnywhere.Data.SQLAnywhere.v4.0
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 iAnywhere.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 16 Demo" ) )
            {
                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 SQL Server (SqlClient) と SQL Anywhere .NET データプロバイダとでは、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 で意味を持たないコード規則もあります。SQL Server と SQL Anywhere の間には大きな違いがあります。SQL Server のインスタンスはすべて複数のデータベースを保持しますが、SQL Anywhere のデータベースはすべて単一のファイルです。

   ●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 SQL Server Express に接続します。SQL Anywhere プロバイダでは、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、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。

   ●SqlClient では、Entity Framework から DbDeleteDatabase または DbCreateDatabase を呼び出したときに、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 (SQL Server Express Edition のみ)。SQL Anywhere プロバイダではデータベースの削除や作成をすることは決してなく、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 (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 では、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. SQL Server SqlClient API では、データ注釈属性 TimeStamp を持つカラムが、SQL Server のデータ型 timestamp/rowversion にマッピングされます。SQL Server の timestamp/rowversion については開発者のあいだで誤解されている面があります。SQL Server の timestamp/rowversion データ型は、SQL Anywhere や他のほとんどのデータベースベンダーとは異なります。

   ●SQL Server の timestamp/rowversion はバイナリ (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、SQL Server の datetime データ型と同等です。

   ●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。

   ●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 SQL Anywhere 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 SQL Server のデフォルトのスキーマです。ただし、**dbo** は SQL Anywhere では適切ではありません。SQL Anywhere では、データ注釈か 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 (**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" );
      }
   }
}
```

SQL Anywhere .NET データプロバイダの配備

SQL Anywhere .NET データプロバイダを使用するには、コンピュータまたはハンドヘルドデバイスに以下をインストールしてください。

- .NET Framework および .NET Compact Framework バージョン 2.0 以降 (あるいはそのいずれか)
- Visual Studio 2005 以降、または C# などの .NET 言語コンパイラ (開発用としてのみ必要)

SQL Anywhere .NET データプロバイダのコードは、各プラットフォームの DLL にあります。

Windows に必要なファイル

Windows (Windows Mobile を除く) の場合、次のいずれかの DLL が必要です。

- *%SQLANY16%¥V2¥Assembly¥V2¥iAnywhere.Data.SQLAnywhere.dll*
- *%SQLANY16%¥V2¥Assembly¥V3.5¥iAnywhere.Data.SQLAnywhere.v3.5.dll*
- *%SQLANY16%¥V2¥Assembly¥V4¥iAnywhere.Data.SQLAnywhere.v4.0.dll*

DLL の選択は、対象とする .NET のバージョンによって異なります。

Windows バージョンのプロバイダには、次の DLL も必要です。

- ***policy.16.0.iAnywhere.Data.SQLAnywhere.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 Sybase によって更新されます。また、バージョン 3.5 プロバイダ用 (*policy.16.0.iAnywhere.Data.SQLAnywhere.v3.5.dll*) とバージョン 4.0 プロバイダ用 (*policy.16.0.iAnywhere.Data.SQLAnywhere.v4.0.dll*) のポリシーファイルもあります。
- ***dblgen16.dll*** この言語 DLL には、プロバイダによって発行された英語 (en) のメッセージが含まれています。この言語 DLL は、中国語 (zh)、フランス語 (fr)、ドイツ語 (de)、日本語 (jp) などの他の多くの言語で使用できます。
- ***dbcon16.dll*** この DLL には、**[SQL Anywhere への接続]** ウィンドウのサポートコードが含まれています。

Windows Mobile に必要なファイル

Windows Mobile の場合、次の DLL が必要です。

- ***%SQLANY16%¥CE¥Assembly¥V2¥iAnywhere.Data.SQLAnywhere.dll*** これは、Windows Mobile 用 Compact Framework バージョン 2 をサポートするプロバイダのバージョンです。
- ***%SQLANY16%¥CE¥Assembly¥V3.5¥iAnywhere.Data.SQLAnywhere.dll*** これは、Windows Mobile 用 Compact Framework バージョン 3.5 をサポートするプロバイダのバージョンです。
- ***%SQLANY16%¥CE¥Arm.50¥dblgen16.dll*** この言語 DLL には、プロバイダによって発行された英語 (en) のメッセージが含まれています。この言語 DLL は、中国語 (zh)、フランス語 (fr)、ドイツ語 (de)、日本語 (jp) などの他の多くの言語で使用できます。

- ***%SQLANY16%¥CE¥Arm.50¥dbcon16.dll*** この DLL には、**[SQL Anywhere への接続]** ウィンドウのサポートコードが含まれています。

Visual Studio は、.NET データプロバイダ DLL (*iAnywhere.Data.SQLAnywhere.dll* または *iAnywhere.Data.SQLAnywhere.v3.5.dll*) をプログラムとともにデバイスに配備します。Visual Studio を使用していない場合は、データプロバイダ DLL をアプリケーションとともにデバイスにコピーする必要があります。この DLL は、アプリケーションと同じディレクトリまたは *¥Windows* ディレクトリに配置できます。

SQL Anywhere .NET データプロバイダ dbdata DLL

SQL Anywhere .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 *dbdata16.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、DLL のビット数、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、サブディレクトリ名の例です。

```
{16AA8FB8-4A98-4757-B7A5-0FF22C0A6E33}_160.x64_1
```

SQL Anywhere .NET データプロバイダ DLL の登録

Windows バージョンの SQL Anywhere .NET データプロバイダ DLL (*%SQLANY16%¥Assembly ¥V2¥iAnywhere.Data.SQLAnywhere.dll*) は、SQL Anywhere ソフトウェアのインストールをインストールするとき、グローバルアセンブリキャッシュに登録されます。Windows Mobile の場合、この DLL を登録する必要はありません。

SQL Anywhere .NET データプロバイダを配備する場合は、Microsoft SDK に含まれている **gacutil** ユーティリティを使用して、この DLL を登録できます。

SQL Anywhere .NET データプロバイダを配備するときに、プロバイダを DbProviderFactory インスタンスとして登録するには、.NET *machine.config* ファイルにエントリを追加する必要があります。次に似たエントリを <DbProviderFactories> セクションに配置する必要があります。

```
<add invariant="iAnywhere.Data.SQLAnywhere"
name="SQL Anywhere 16 Data Provider"
description=".Net Framework Data Provider for SQL Anywhere 16"
type="iAnywhere.Data.SQLAnywhere.SAFactory, iAnywhere.Data.SQLAnywhere.v3.5,
   Version=16.0.0.14033, Culture=neutral, PublicKeyToken=f222fc4333e0d400"/>
```

バージョン番号は、インストールしているプロバイダのバージョンに一致する必要があります。設定ファイルは *¥WINDOWS¥Microsoft.NET¥Framework¥v2.0.50727¥CONFIG* にあります。64 ビット Windows システムの場合、同様に変更する必要のある *Framework64* ツリーの下に 2 番目の設定ファイルがあります。

.NET でのトレースのサポート

SQL Anywhere .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。トレースは、Windows Mobile ではサポートされていません。

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="iAnywhere.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* という名前で、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、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**　すべての関数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#
```

詳細については、http://msdn.microsoft.com/ja-jp/library/ms971550.aspx の「Tracing Data Access」を参照してください。

Windows 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

Visual Studio がインストールされている必要があります。

内容と備考

Windows Mobile ではトレースはサポートされていません。

♦ タスク

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

   Visual Studio を起動し、*%SQLANYSAMP16%¥SQLAnywhere¥ADO.NET¥TableViewer¥TableViewer.sln* を開きます。

2. *App.config* という名前のアプリケーションファイルを作成し、次の設定内容をコピーします。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.diagnostics>
<sources>
 <source name="iAnywhere.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。

次の手順

Visual Studio の **[出力]** ウィンドウでトレースログを表示します。

参照

- http://msdn.microsoft.com/ja-jp/library/ms971550.aspx

.NET データプロバイダチュートリアル

この項では、SQL Anywhere .NET データプロバイダに用意されている Simple サンプルプロジェクトと Table Viewer サンプルプロジェクトの使用方法について説明します。サンプルプロジェクトは Visual Studio 2005 以降のバージョンで使用できます。サンプルプロジェクトは Visual Studio 2005 を使用して開発されています。2005 よりも新しいバージョンをお使いの場合は、Visual Studio の**アップグレードウィザード**を実行する必要がある場合があります。また、この項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。

チュートリアル：Simple コードサンプルの使用

Simple プロジェクトでは、.NET データプロバイダを使用してデータベースサーバから結果セットを取得します。

前提条件

コンピュータに 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。

SELECT ANY TABLE システム権限が必要です。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

内容と備考

Simple プロジェクトは SQL Anywhere の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。

♦ タスク

1. 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. *%SQLANYSAMP16%¥SQLAnywhere¥ADO.NET¥SimpleWin32* を参照し、*Simple.sln* プロジェクトを開きます。

4. プロジェクトで SQL Anywhere .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 (*iAnywhere.Data.SQLAnywhere*) への参照を表示するには、**[ソリューション エクスプローラー]** 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 エクスプローラー]** 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。

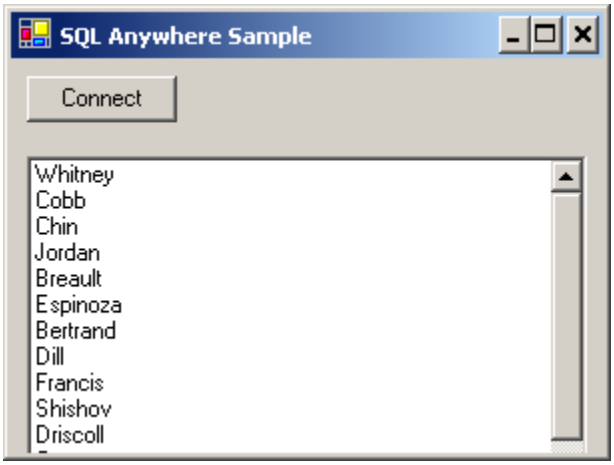
   ```
   using iAnywhere.Data.SQLAnywhere;
   ```

   この行は C# プロジェクトに必要です。Visual Basic .NET を使用している場合、ソースコードに 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 **[Connect]** をクリックします。

   アプリケーションが SQL Anywhere 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、SQL Anywhere .NET データプロバイダを使用して SQL Anywhere データベースサーバ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。

次の手順

なし。

例

完全なアプリケーションは、SQL Anywhere サンプルディレクトリの *%SQLANYSAMP16%¥SQLAnywhere¥ADO.NET¥SimpleWin32* にあります。

参照

- 「Visual Studio プロジェクトでの .NET データプロバイダの使用」44 ページ
- 「Simple サンプルプロジェクトの説明」80 ページ

Simple サンプルプロジェクトの説明

この項では、Simple コードサンプルのコードを使用して 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 コードサンプルは、SQL Anywhere サンプルディレクトリに格納されている SQL Anywhere サンプルデータベース *demo.db* を使用します。

SQL Anywhere サンプルディレクトリのロケーションについては、「サンプルディレクトリ」『SQL Anywhere サーバ データベース管理』を参照してください。

データベース内のテーブルとテーブル間の関係を含むサンプルデータベースの詳細については、「SQL Anywhere サンプルデータベース」『SQL Anywhere 16 紹介』を参照してください。

この項では、1 回に示すコードは数行です。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、*%SQLANYSAMP16%¥SQLAnywhere¥ADO.NET ¥SimpleWin32* のサンプルプロジェクトを開きます。

制御の宣言

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

```
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 16
Demo;UID=DBA;PWD=sql" );
```

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 SQL Anywhere サンプルデータベースに接続します。

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

SAConnection オブジェクトの詳細については、SAConnection クラス [SQL Anywhere .NET]169 ページを参照してください。

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てコマンド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、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 );
```

SACommand オブジェクトの詳細については、SACommand クラス [SQL Anywhere .NET]129 ページを参照してください。

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();
```

SADataReader オブジェクトの詳細については、SADataReader クラス [SQL Anywhere .NET]224 ページを参照してください。

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 SQL Anywhere .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 );
}
```

SAException オブジェクトの詳細については、SAException クラス [SQL Anywhere .NET]258 ページを参照してください。

チュートリアル：Table Viewer コードサンプルの使用

TableViewer プロジェクトでは、.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 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。

SELECT ANY TABLE システム権限が必要です。

内容と備考

TableViewer プロジェクトは SQL Anywhere の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。

♦ タスク

1. 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. *%SQLANYSAMP16%¥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 SQL Anywhere .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 (*iAnywhere.Data.SQLAnywhere*) への参照を表示するには、**[ソリューション エクスプローラー]** 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 エクスプローラー]** 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 iAnywhere.Data.SQLAnywhere;
     ```

     この行は C# プロジェクトに必要です。Visual Basic を使用している場合、ソースコードに Imports 行を追加する必要があります。

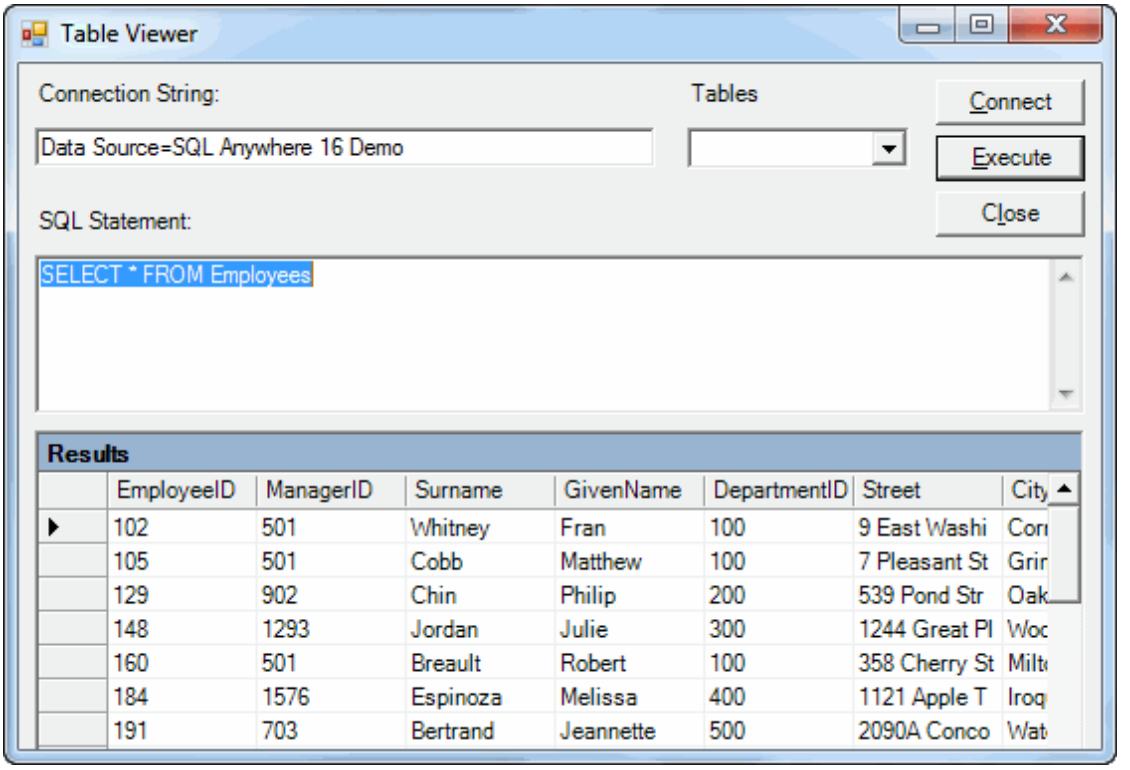
6. **[デバッグ]** » **[デバッグなしで開始]** をクリックするか、[Ctrl + F5] を押して、Table Viewer サンプルを実行します。

   アプリケーションが SQL Anywhere 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 **[Connect]** 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、.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。

次の手順

なし。

例

完全なアプリケーションは、SQL Anywhere サンプルディレクトリの *%SQLANYSAMP16%¥SQLAnywhere¥ADO.NET¥TableViewer* にあります。

参照

- 「Visual Studio プロジェクトでの .NET データプロバイダの使用」44 ページ
- 「Table Viewer サンプルプロジェクトの説明」85 ページ

Table Viewer サンプルプロジェクトの説明

この項では、Table Viewer コードサンプルのコードを使用して 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 プロジェクトは、SQL Anywhere サンプルディレクトリに格納されている SQL Anywhere サンプルデータベース *demo.db* を使用します。

SQL Anywhere サンプルディレクトリのロケーションについては、「サンプルディレクトリ」『SQL Anywhere サーバ データベース管理』を参照してください。

データベース内のテーブルとテーブル間の関係を含むサンプルデータベースの詳細については、「SQL Anywhere サンプルデータベース」『SQL Anywhere 16 紹介』を参照してください。

この項では、1 回に示すコードは数行です。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、*%SQLANYSAMP16%¥SQLAnywhere¥ADO.NET ¥TableViewer* のサンプルプロジェクトを開きます。

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 SQL Anywhere 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、SQL Anywhere 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;
```

SAConnection クラスの詳細については、SAConnection クラス [SQL Anywhere .NET]169 ページを参照してください。

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 16 Demo" 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ているかは、*TableViewer.cs* で「Windows Form Designer Generated Code」という記述のあるリージョンまたはセクションを開くことで確認できます。このセクションでは、次のコード行を探します。

```
this.txtConnectString.Text = "Data Source=SQL Anywhere 16 Demo";
```

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();
```

SAConnection コンストラクタの詳細については、SAConnection.SAConnection コンストラクタ [SQL Anywhere .NET]171 ページを参照してください。

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てコマンド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、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 );
```

SACommand オブジェクトの詳細については、SACommand クラス [SQL Anywhere .NET]129 ページを参照してください。

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();
```

SADataReader オブジェクトの詳細については、SADataReader クラス [SQL Anywhere .NET]224 ページを参照してください。

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 Tables 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" );
  }
```

SAException オブジェクトの詳細については、SAException クラス [SQL Anywhere .NET]258 ページを参照してください。

チュートリアル : 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

この項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。

権限

SELECT ANY TABLE システム権限が必要です。

レッスン 1 : Table Viewer の作成

このレッスンでは、Microsoft Visual Studio、サーバエクスプローラー、および SQL Anywhere .NET データプロバイダを使用して、SQLAnywhere 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 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。

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。「チュートリアル : 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」

内容と備考

このチュートリアルは、Visual Studio と .NET Framework に基づきます。完全なアプリケーションは、ADO.NET プロジェクトの *%SQLANYSAMP16%¥SQLAnywhere¥ADO.NET¥SimpleViewer¥SimpleViewer.sln* にあります。

♦ タスク

1. 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 2008/2010 の場合) をクリックします。

   c. プロジェクトの **[名前]** フィールドに、**MySimpleViewer** と入力します。

   d. **[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 エクスプローラー]** をクリックします。

4. **[サーバ エクスプローラー]** 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. 他のプロジェクトで **[接続の追加]** を一度も使用したことがない場合は、データソースのリストが表示されます。データソースのリストから **[SQL Anywhere]** をクリックします。

      [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 **[SQL Anywhere]** に変更します。

   b. **[データソース]** で、**[ODBC データソース名]** をクリックし、**SQL Anywhere 16 Demo** と入力します。

      注意
      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。Visual Studio の 32 ビット設計環境で [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。

   c. **[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。

   d. **[OK]** をクリックします。

   SQL Anywhere.demo16 という新しい接続が **[サーバ エクスプローラー]** ウィンドウに表示されます。

6. **[サーバ エクスプローラー]** ウィンドウで、テーブル名が表示されるまで **SQL Anywhere.demo16** 接続を展開します。

   (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。

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. (Visual Studio 2010 のみ) **[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. **[データ接続の選択]** ページで **SQL Anywhere.demo16** をクリックし、**[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。

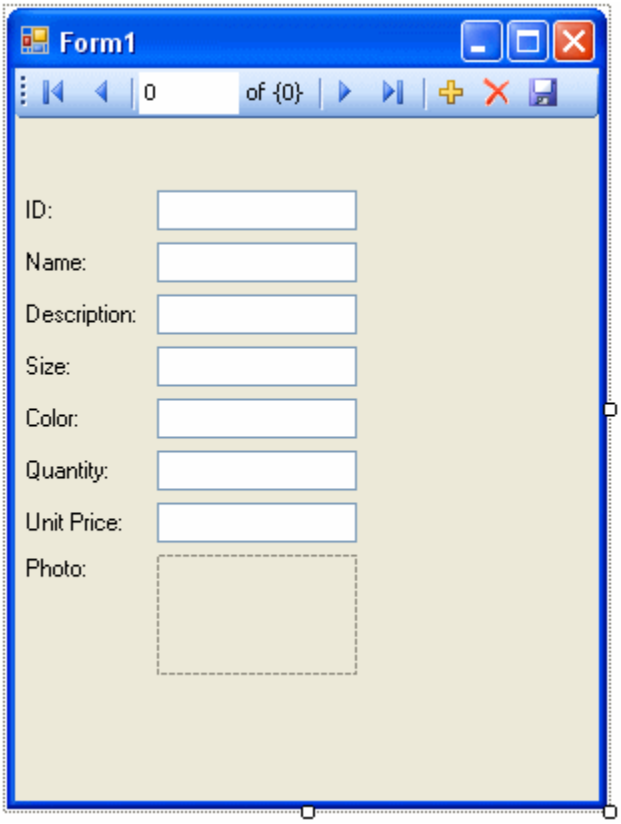
   [データソース] ウィンドウが表示されます。

   [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 SQL Anywhere 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、Visual Studio、サーバエクスプローラー、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 .NET アプリケーションが作成されました。

次の手順

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。 「レッスン 2：同期データコントロールの追加」92 ページに進みます。

レッスン 2：同期データコントロールの追加

このレッスンでは、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。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。

前提条件

このレッスンは、「レッスン 1：Table Viewer の作成」87 ページのレッスンの続きです。

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：データベース内の Java の使用」

内容と備考

完全なアプリケーションは、ADO.NET プロジェクトの *%SQLANYSAMP16%¥SQLAnywhere¥ADO.NET¥SimpleViewer¥SimpleViewer.sln* にあります。

♦ タスク

1. Visual Studio を起動し、MySimpleViewer プロジェクトをロードします。このプロジェクトは「レッスン 1：Table Viewer の作成」87 ページで作成したものです。

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。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

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. **[完了]** 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。

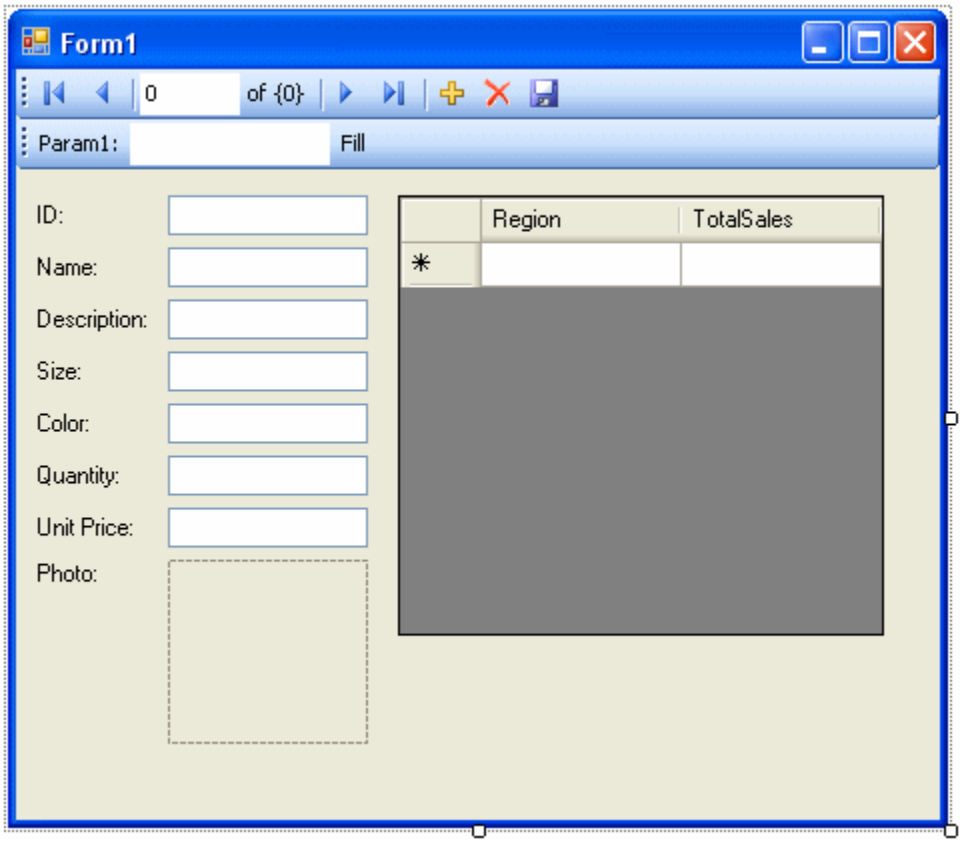
9. フォームデザインタブ (Form1) をクリックします。

   ●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。

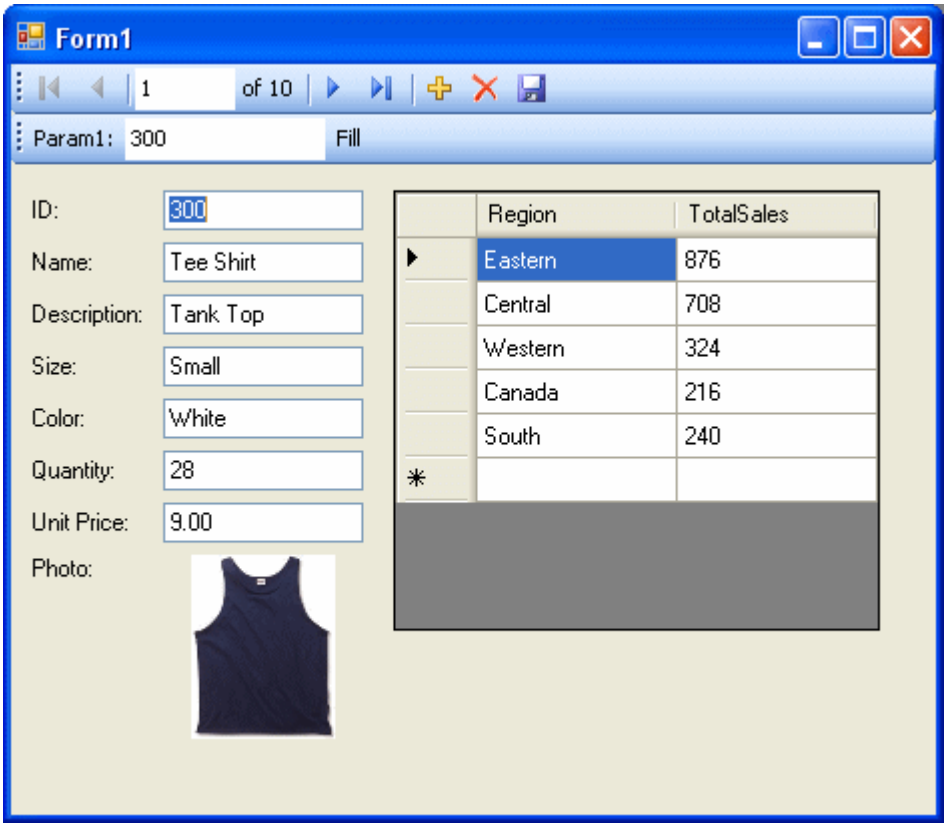


データグリッドビューのコントロールがフォームに表示されます。

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]** (VS 2010) 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。Visual Basic を使用している場合は、**[FillToolStripButton_Click]** というイベントです。

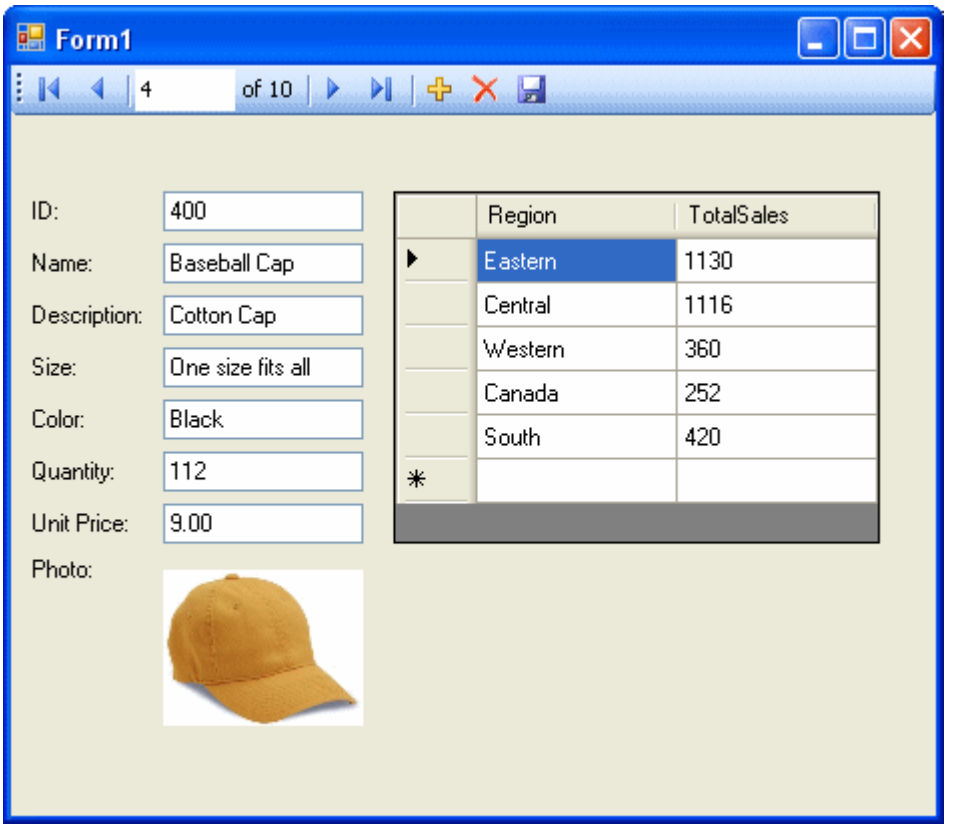
e. **[fillToolStripButton_Click]** をダブルクリックして、fillToolStripButton_Click イベントハンドラに関するフォームのコードウィンドウを開きます。

f. param1ToolStripTextBox または productIDToolStripTextBox (VS 2010) への参照を検索し、これを iDTextBox に変更します。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ー、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。

SQL Anywhere ASP.NET プロバイダ

SQL Anywhere ASP.NET プロバイダは、SQL Server の標準の ASP.NET プロバイダに代わり、SQL Anywhere データベースに基づいて 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 SQL Anywhere 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、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、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。

注意
『**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 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。www.sybase.com/detail?id=1080238 を参照してください。

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。

SQL Anywhere 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。*SASetupAspNet.exe* は SQL Anywhere 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 プロバイダのリソースはすべて *aspnet_* で始まります。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 の実行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切な権限をもつユーザとの接続文字列を指定することをおすすめします。次の最小の特権セットが必要です。

- 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ことをおすすめします。メンバーシッププロバイダとロールプロバイダの両方がインストールされていない場合、Visual Studio ASP.NET の Web サイト管理ツールの有効性が軽減されます。

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 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。www.sybase.com/detail?id=1080238 を参照してください。

接続文字列の登録

接続文字列は次の 2 通りの方法で登録できます。

- 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。

- SQL Anywhere の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=sql"
```

<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="iAnywhere.Data.SQLAnywhere"/>
</connectionStrings>
```

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、プロバイダごとに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、SQL Anywhere ASP.NET の各プロバイダで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>
```

カラムの詳細については、「メンバーシッププロバイダの XML 属性」102 ページを参照してください。

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>
```

カラムの詳細については、「ロールプロバイダのテーブルスキーマ」103 ページを参照してください。

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>
```

カラムの詳細については、「プロファイルプロバイダのテーブルスキーマ」103 ページを参照してください。

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>
```

カラムの詳細については、「Web パーツパーソナル化プロバイダのテーブルスキーマ」104 ページを参照してください。

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

ヘルスモニタリングの設定の詳細については、http://msdn.microsoft.com/ja-jp/library/ms998306.aspx にある「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」を参照してください。.

```
<healthMonitoring enabled="true">
          ...
  <providers>
    <add name="SAWebEventProvider"
      type="iAnywhere.Web.Security.SAWebEventProvider"
      connectionStringName="MyConnectionString"
      commandTimeout="30"
      bufferMode="Notification"
      maxEventDetailsLength="Infinite" /
  </providers>
          ...
</healthMonitoring>
```

カラムの詳細については、「ヘルスモニタリングプロバイダのテーブルスキーマ」104 ページを参照してください。

メンバーシッププロバイダの XML 属性

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。
connectionString	接続文字列 (省略可能)。connectionStringName を指定しない場合は必須。
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。

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)。connectionStringName を指定しない場合は必須。
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)。

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 人のユーザに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)。connectionStringName を指定しない場合は必須。

カラム名	説明
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)。

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)。connectionStringName を指定しない場合は必須。
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)。

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。

ヘルスモニタリングの設定の詳細については、http://msdn.microsoft.com/ja-jp/library/ms998306.aspx にある「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」を参照してください。.

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)。connectionStringName を指定しない場合は必須。
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。

SQL Anywhere .NET API リファレンス

ネームスペース

- `iAnywhere.Data.SQLAnywhere`
- `iAnywhere.SQLAnywhere.Server`

SABulkCopy クラス

別のソースのデータを使用して、SQL Anywhere テーブルを効率的にバルクロードします。

Visual Basic の構文

```
Public NotInheritable Class SABulkCopy Implements System.IDisposable
```

C# の構文

```
public sealed class SABulkCopy : System.IDisposable
```

基本クラス

- System.IDisposable

メンバー

継承されたメンバーを含む SABulkCopy クラスのすべてのメンバー。

名前	説明
SABulkCopy コンストラクタ	SABulkCopy オブジェクトを初期化します。
Close メソッド	SABulkCopy インスタンスを閉じます。
Dispose メソッド	SABulkCopy インスタンスを破棄します。
WriteToServer メソッド	指定された System.Data.DataRow オブジェクトの配列内のすべてのローを、SABulkCopy オブジェクトの DestinationTableName プロパティで指定される送信先テーブルにコピーします。
BatchSize プロパティ	各バッチの中のローの数を取得または設定します。
BulkCopyTimeOut プロパティ	タイムアウトするまでに完了するオペレーションの秒数を取得または設定します。
ColumnMappings プロパティ	SABulkCopyColumnMapping 項目のコレクションを返します。
DestinationTableName プロパティ	サーバ上の送信先テーブルの名前を取得または設定します。
NotifyAfter プロパティ	通知イベントが生成されるまでに処理されるローの数を取得または設定します。
SARowsCopied イベント	NotifyAfter プロパティで指定される数のローが処理されるたびに、このイベントが発生します。

備考

SABulkCopy クラスは、.NET Compact Framework 2.0 では使用できません。

Implements: System.IDisposable

SABulkCopy コンストラクタ

SABulkCopy オブジェクトを初期化します。

オーバロードリスト

名前	説明
SABulkCopy(SAConnection) コンストラクタ	SABulkCopy オブジェクトを初期化します。

名前	説明
SABulkCopy(SAConnection, SABulkCopyOptions, SATransaction) コンストラクタ	SABulkCopy オブジェクトを初期化します。
SABulkCopy(string) コンストラクタ	SABulkCopy オブジェクトを初期化します。
SABulkCopy(string, SABulkCopyOptions) コンストラクタ	SABulkCopy オブジェクトを初期化します。

SABulkCopy(SAConnection) コンストラクタ

SABulkCopy オブジェクトを初期化します。

Visual Basic の構文

```
Public Sub New(ByVal connection As SAConnection)
```

C# の構文

```
public SABulkCopy(SAConnection connection)
```

パラメータ

- **connection** バルクコピーオペレーションの実行に使用する、すでに開いている SAConnection。接続が開いていない場合は、WriteToServer に例外がスローされます。

備考

SABulkCopy クラスは、.NET Compact Framework 2.0 では使用できません。

SABulkCopy(SAConnection, SABulkCopyOptions, SATransaction) コンストラクタ

SABulkCopy オブジェクトを初期化します。

Visual Basic の構文

```
Public Sub New(
    ByVal connection As SAConnection,
    ByVal copyOptions As SABulkCopyOptions,
    ByVal externalTransaction As SATransaction
)
```

C# の構文

```
public SABulkCopy(
    SAConnection connection,
    SABulkCopyOptions copyOptions,
    SATransaction externalTransaction
)
```

パラメータ

- **connection**　バルクコピーオペレーションの実行に使用する、すでに開いている SAConnection。接続が開いていない場合は、WriteToServer に例外がスローされます。
- **copyOptions**　目的のテーブルにコピーされるデータソースローを決定する、SABulkCopyOptions 列挙の値の組み合わせ。
- **externalTransaction**　バルクコピーが発生する、既存の SATransaction インスタンス。externalTransaction が NULL でない場合、バルクコピーオペレーションはその中で行われます。外部トランザクションと UseInternalTransaction オプションの両方を指定すると、エラーになります。

備考

SABulkCopy クラスは、.NET Compact Framework 2.0 では使用できません。

SABulkCopy(string) コンストラクタ

SABulkCopy オブジェクトを初期化します。

Visual Basic の構文

```
Public Sub New(ByVal connectionString As String)
```

C# の構文

```
public SABulkCopy(string connectionString)
```

パラメータ

- **connectionString**　SABulkCopy インスタンスによって使用されるために開かれる接続を定義する文字列。接続文字列は keyword=value のペアがセミコロンで区切られたリストです。

備考

この構文は、connectionString を使用して WriteToServer の実行中に接続を開きます。WriteToServer が終了すると、接続が閉じます。

SABulkCopy クラスは、.NET Compact Framework 2.0 では使用できません。

SABulkCopy(string, SABulkCopyOptions) コンストラクタ

SABulkCopy オブジェクトを初期化します。

Visual Basic の構文

```
Public Sub New(
    ByVal connectionString As String,
    ByVal copyOptions As SABulkCopyOptions
)
```

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

C# の構文

```
public SABulkCopy(
    string connectionString,
    SABulkCopyOptions copyOptions
)
```

パラメータ

- **connectionString**　SABulkCopy インスタンスによって使用されるために開かれる接続を定義する文字列。接続文字列は keyword=value のペアがセミコロンで区切られたリストです。
- **copyOptions**　目的のテーブルにコピーされるデータソースローを決定する、SABulkCopyOptions 列挙の値の組み合わせ。

備考

この構文は、connectionString を使用して WriteToServer の実行中に接続を開きます。WriteToServer が終了すると、接続が閉じます。copyOptions パラメータには、上記の効果があります。

SABulkCopy クラスは、.NET Compact Framework 2.0 では使用できません。

Close メソッド

SABulkCopy インスタンスを閉じます。

Visual Basic の構文

```
Public Sub Close()
```

C# の構文

```
public void Close()
```

Dispose メソッド

SABulkCopy インスタンスを破棄します。

Visual Basic の構文

```
Public Sub Dispose()
```

C# の構文

```
public void Dispose()
```

WriteToServer メソッド

指定された System.Data.DataRow オブジェクトの配列内のすべてのローを、SABulkCopy オブジェクトの DestinationTableName プロパティで指定される送信先テーブルにコピーします。

オーバロードリスト

名前	説明
WriteToServer(DataRow[]) メソッド	指定された System.Data.DataRow オブジェクトの配列内のすべてのローを、SABulkCopy オブジェクトの DestinationTableName プロパティで指定される送信先テーブルにコピーします。
WriteToServer(DataTable) メソッド	指定された System.Data.DataTable のすべてのローを、SABulkCopy オブジェクトの DestinationTableName プロパティで指定される送信先テーブルにコピーします。
WriteToServer(DataTable, DataRowState) メソッド	指定されたローの状態を持つ指定された System.Data.DataTable のすべてのローを、SABulkCopy オブジェクトの DestinationTableName プロパティで指定される送信先テーブルにコピーします。
WriteToServer(IDataReader) メソッド	指定された System.Data.IDataReader のすべてのローを、SABulkCopy オブジェクトの DestinationTableName プロパティで指定される送信先テーブルにコピーします。

WriteToServer(DataRow[]) メソッド

指定された System.Data.DataRow オブジェクトの配列内のすべてのローを、SABulkCopy オブジェクトの DestinationTableName プロパティで指定される送信先テーブルにコピーします。

Visual Basic の構文

```
Public Sub WriteToServer(ByVal rows As DataRow())
```

C# の構文

```
public void WriteToServer(DataRow[] rows)
```

パラメータ

- **rows**　送信先テーブルにコピーされる System.Data.DataRow オブジェクトの配列。

備考

SABulkCopy クラスは、.NET Compact Framework 2.0 では使用できません。

参照

- SABulkCopy.DestinationTableName プロパティ [SQL Anywhere .NET]113 ページ

WriteToServer(DataTable) メソッド

指定された System.Data.DataTable のすべてのローを、SABulkCopy オブジェクトの DestinationTableName プロパティで指定される送信先テーブルにコピーします。

Visual Basic の構文

```
Public Sub WriteToServer(ByVal table As DataTable)
```

C# の構文

```
public void WriteToServer(DataTable table)
```

パラメータ

- **table** ローが送信先テーブルにコピーされる System.Data.DataTable。

備考

SABulkCopy クラスは、.NET Compact Framework 2.0 では使用できません。

参照

- SABulkCopy.DestinationTableName プロパティ [SQL Anywhere .NET]113 ページ

WriteToServer(DataTable, DataRowState) メソッド

指定されたローの状態を持つ指定された System.Data.DataTable のすべてのローを、SABulkCopy オブジェクトの DestinationTableName プロパティで指定される送信先テーブルにコピーします。

Visual Basic の構文

```
Public Sub WriteToServer(
    ByVal table As DataTable,
    ByVal rowState As DataRowState
)
```

C# の構文

```
public void WriteToServer(DataTable table, DataRowState rowState)
```

パラメータ

- **table** ローが送信先テーブルにコピーされる System.Data.DataTable。
- **rowState** System.Data.DataRowState 列挙の値。ローステータスに一致するローのみ、送信先にコピーされます。

備考

ローステータスに一致するローのみ、コピーされます。

SABulkCopy クラスは、.NET Compact Framework 2.0 では使用できません。

参照

- SABulkCopy.DestinationTableName プロパティ [SQL Anywhere .NET]113 ページ

WriteToServer(IDataReader) メソッド

指定された System.Data.IDataReader のすべてのローを、SABulkCopy オブジェクトの DestinationTableName プロパティで指定される送信先テーブルにコピーします。

Visual Basic の構文

```
Public Sub WriteToServer(ByVal reader As IDataReader)
```

C# の構文

```
public void WriteToServer(IDataReader reader)
```

パラメータ

- **reader** ローが送信先テーブルにコピーされる System.Data.IDataReader。

備考

SABulkCopy クラスは、.NET Compact Framework 2.0 では使用できません。

参照

- SABulkCopy.DestinationTableName プロパティ [SQL Anywhere .NET]113 ページ

BatchSize プロパティ

各バッチの中のローの数を取得または設定します。

Visual Basic の構文

```
Public Property BatchSize As Integer
```

C# の構文

```
public int BatchSize {get;set;}
```

備考

各バッチが終了すると、バッチ内のローがサーバに送信されます。

各バッチの中のロー数。デフォルトは 0 です。

このプロパティを 0 に設定すると、すべてのローが 1 つのバッチで送信されます。

このプロパティに 0 未満の値を設定すると、エラーになります。

バッチの進行中にこの値が変更された場合、現在のバッチはそのまま完了し、それ以降のバッチが新しい値を使用します。

BulkCopyTimeOut プロパティ

タイムアウトするまでに完了するオペレーションの秒数を取得または設定します。

Visual Basic の構文

```
Public Property BulkCopyTimeout As Integer
```

C# の構文

```
public int BulkCopyTimeout {get;set;}
```

備考

デフォルト値は 30 秒です。

値が 0 の場合、制限はありません。この場合、待機時間が無限になる可能性があるため、値は 0 にしないでください。

オペレーションがタイムアウトすると、現在のトランザクション内のすべてのローがロールバックされ、SAException が発生します。

このプロパティに 0 未満の値を設定すると、エラーになります。

ColumnMappings プロパティ

SABulkCopyColumnMapping 項目のコレクションを返します。

Visual Basic の構文

```
Public ReadOnly Property ColumnMappings As
SABulkCopyColumnMappingCollection
```

C# の構文

```
public SABulkCopyColumnMappingCollection ColumnMappings {get;}
```

備考

カラムマッピングは、データソース内のカラムと、送信先のカラムの間の関係を定義します。

デフォルトでは、空のコレクションです。

WriteToServer の実行中は、プロパティを変更できません。

WriteToServer の実行時に ColumnMappings が空の場合、ソース内の先頭のカラムが送信先の先頭のカラムにマッピングされ、2 番目は 2 番目にマッピングされます。以降についても同様です。この処理は、カラムの型が変換可能な場合、ソースカラム以上の送信先カラムがある場合、余分な送信先カラムが NULL 入力可のカラムである場合に行われます。

DestinationTableName プロパティ

サーバ上の送信先テーブルの名前を取得または設定します。

Visual Basic の構文

```
Public Property DestinationTableName As String
```

C# の構文

```
public string DestinationTableName {get;set;}
```

備考

デフォルト値は NULL 参照です。Visual Basic では、これは Nothing です。

WriteToServer の実行時に値が変更されても、変更は反映されません。

WriteToServer への呼び出しの前に値が設定されていない場合、InvalidOperationException が発生します。

値を NULL または空の文字列に設定すると、エラーになります。

NotifyAfter プロパティ

通知イベントが生成されるまでに処理されるローの数を取得または設定します。

Visual Basic の構文

```
Public Property NotifyAfter As Integer
```

C# の構文

```
public int NotifyAfter {get;set;}
```

備考

プロパティが設定されていない場合は、0 が返されます。

WriteToServer の実行時に加えられる NotifyAfter への変更は、次の通知まで反映されません。

このプロパティに 0 未満の値を設定すると、エラーになります。

NotifyAfter と BulkCopyTimeout の値は相互に排他的であるため、データベースにローが送信されない場合や、コミットされない場合も、イベントは起動します。

参照

- SABulkCopy.BulkCopyTimeout プロパティ [SQL Anywhere .NET]112 ページ

SARowsCopied イベント

NotifyAfter プロパティで指定される数のローが処理されるたびに、このイベントが発生します。

Visual Basic の構文

```
Public Event SARowsCopied As SARowsCopiedEventHandler
```

C# の構文

```
public event SARowsCopiedEventHandler SARowsCopied;
```

備考

SARowsCopied イベントの受信は、ローがデータベースサーバに送信されたことやコミットされたことを意味するわけではありません。このイベントから Close メソッドを呼び出すことはできません。

参照

- SABulkCopy.NotifyAfter プロパティ [SQL Anywhere .NET]114 ページ

SABulkCopyColumnMapping クラス

SABulkCopy インスタンスのデータソース内のカラムと、インスタンスの送信先テーブル内のカラムの間のマッピングを定義します。

Visual Basic の構文

```
Public NotInheritable Class SABulkCopyColumnMapping
```

C# の構文

```
public sealed class SABulkCopyColumnMapping
```

メンバー

継承されたメンバーを含む SABulkCopyColumnMapping クラスのすべてのメンバー。

名前	説明
SABulkCopyColumnMapping コンストラクタ	カラムの順序または名前を使用してソースカラムと送信先カラムを参照する、新しいカラムマッピングを作成します。
DestinationColumn プロパティ	マッピングされる送信先データベーステーブルのカラムの名前を取得または設定します。
DestinationOrdinal プロパティ	マッピングされる送信先テーブルにおけるカラムの順序を取得または設定します。
SourceColumn プロパティ	データソースにマッピングされるカラムの名前を取得または設定します。
SourceOrdinal プロパティ	データソース内のソースカラムの順序位置を取得または設定します。

備考

SABulkCopyColumnMapping クラスは、.NET Compact Framework 2.0 では使用できません。

SABulkCopyColumnMapping コンストラクタ

カラムの順序または名前を使用してソースカラムと送信先カラムを参照する、新しいカラムマッピングを作成します。

オーバロードリスト

名前	説明
SABulkCopyColumnMapping() コンストラクタ	カラムの順序または名前を使用してソースカラムと送信先カラムを参照する、新しいカラムマッピングを作成します。
SABulkCopyColumnMapping(int, int) コンストラクタ	カラムの順序を使用してソースカラムと送信先カラムを参照する、新しいカラムマッピングを作成します。
SABulkCopyColumnMapping(int, string) コンストラクタ	ソースカラムを参照するカラムの序数と、送信先カラムを参照するカラム名を使用して、新しいカラムマッピングを作成します。
SABulkCopyColumnMapping(string, int) コンストラクタ	カラム名を使用してソースカラムを参照し、カラムの順序を使用して送信先カラムを参照する、新しいカラムマッピングを作成します。
SABulkCopyColumnMapping(string, string) コンストラクタ	カラムの名前を使用してソースカラムと送信先カラムを参照する、新しいカラムマッピングを作成します。

SABulkCopyColumnMapping() コンストラクタ

カラムの順序または名前を使用してソースカラムと送信先カラムを参照する、新しいカラムマッピングを作成します。

Visual Basic の構文

```
Public Sub New()
```

C# の構文

```
public SABulkCopyColumnMapping()
```

備考

SABulkCopyColumnMapping クラスは、.NET Compact Framework 2.0 では使用できません。

SABulkCopyColumnMapping(int, int) コンストラクタ

カラムの順序を使用してソースカラムと送信先カラムを参照する、新しいカラムマッピングを作成します。

Visual Basic の構文

```
Public Sub New(
    ByVal sourceColumnOrdinal As Integer,
    ByVal destinationColumnOrdinal As Integer
)
```

C# の構文

```
public SABulkCopyColumnMapping(
    int sourceColumnOrdinal,
    int destinationColumnOrdinal
)
```

パラメータ

- **sourceColumnOrdinal**　データソース内のソースカラムの序数位置。データソースの最初のカラムの序数位置は 0 です。
- **destinationColumnOrdinal**　送信先テーブル内の送信先カラムの序数位置。テーブルの最初のカラムの序数位置は 0 です。

備考

SABulkCopyColumnMapping クラスは、.NET Compact Framework 2.0 では使用できません。

SABulkCopyColumnMapping(int, string) コンストラクタ

ソースカラムを参照するカラムの序数と、送信先カラムを参照するカラム名を使用して、新しいカラムマッピングを作成します。

Visual Basic の構文

```
Public Sub New(
    ByVal sourceColumnOrdinal As Integer,
    ByVal destinationColumn As String
)
```

C# の構文

```
public SABulkCopyColumnMapping(
    int sourceColumnOrdinal,
    string destinationColumn
)
```

パラメータ

- **sourceColumnOrdinal**　データソース内のソースカラムの序数位置。データソースの最初のカラムの序数位置は 0 です。
- **destinationColumn**　送信先テーブル内の送信先カラムの名前。

備考

SABulkCopyColumnMapping クラスは、.NET Compact Framework 2.0 では使用できません。

SABulkCopyColumnMapping(string, int) コンストラクタ

カラム名を使用してソースカラムを参照し、カラムの順序を使用して送信先カラムを参照する、新しいカラムマッピングを作成します。

Visual Basic の構文

```
Public Sub New(
    ByVal sourceColumn As String,
    ByVal destinationColumnOrdinal As Integer
)
```

C# の構文

```
public SABulkCopyColumnMapping(
    string sourceColumn,
    int destinationColumnOrdinal
)
```

パラメータ

- **sourceColumn** データソース内のソースカラムの名前。
- **destinationColumnOrdinal** 送信先テーブル内の送信先カラムの序数位置。テーブルの最初のカラムの序数位置は 0 です。

備考

SABulkCopyColumnMapping クラスは、.NET Compact Framework 2.0 では使用できません。

SABulkCopyColumnMapping(string, string) コンストラクタ

カラムの名前を使用してソースカラムと送信先カラムを参照する、新しいカラムマッピングを作成します。

Visual Basic の構文

```
Public Sub New(
    ByVal sourceColumn As String,
    ByVal destinationColumn As String
)
```

C# の構文

```
public SABulkCopyColumnMapping(
    string sourceColumn,
    string destinationColumn
)
```

パラメータ

- **sourceColumn** データソース内のソースカラムの名前。
- **destinationColumn** 送信先テーブル内の送信先カラムの名前。

 Copyright © 2013, SAP AG or an SAP affiliate company.

備考

SABulkCopyColumnMapping クラスは、.NET Compact Framework 2.0 では使用できません。

DestinationColumn プロパティ

マッピングされる送信先データベーステーブルのカラムの名前を取得または設定します。

Visual Basic の構文

```
Public Property DestinationColumn As String
```

C# の構文

```
public string DestinationColumn {get;set;}
```

備考

送信先テーブルのカラムの名前を指定する文字列。DestinationOrdinal プロパティに優先度がない場合は NULL 参照 (Visual Basic の Nothing)。

DestinationColumn プロパティと DestinationOrdinal プロパティは、相互に排他的です。直前に設定された値が優先されます。

DestinationColumn プロパティを設定すると、DestinationOrdinal プロパティは -1 に設定されます。DestinationOrdinal プロパティを設定すると、DestinationColumn プロパティは null 参照 (Visual Basic の Nothing) に設定されます。

DestinationColumn を NULL または空の文字列に設定すると、エラーになります。

参照

- SABulkCopyColumnMapping.DestinationOrdinal プロパティ [SQL Anywhere .NET]119 ページ

DestinationOrdinal プロパティ

マッピングされる送信先テーブルにおけるカラムの順序を取得または設定します。

Visual Basic の構文

```
Public Property DestinationOrdinal As Integer
```

C# の構文

```
public int DestinationOrdinal {get;set;}
```

備考

マッピングされるカラムの送信先テーブルにおける順序を指定する整数。プロパティが設定されていない場合は -1。

DestinationColumn プロパティと DestinationOrdinal プロパティは、相互に排他的です。直前に設定された値が優先されます。

DestinationColumn プロパティを設定すると、DestinationOrdinal プロパティは -1 に設定されます。DestinationOrdinal プロパティを設定すると、DestinationColumn プロパティは null 参照 (Visual Basic の Nothing) に設定されます。

参照

- SABulkCopyColumnMapping.DestinationColumn プロパティ [SQL Anywhere .NET]119 ページ

SourceColumn プロパティ

データソースにマッピングされるカラムの名前を取得または設定します。

Visual Basic の構文

```
Public Property SourceColumn As String
```

C# の構文

```
public string SourceColumn {get;set;}
```

備考

データソースのカラムの名前を指定する文字列。SourceOrdinal プロパティに優先度がない場合は NULL 参照 (Visual Basic の Nothing)。

SourceColumn プロパティと SourceOrdinal プロパティは、相互に排他的です。直前に設定された値が優先されます。

SourceColumn プロパティを設定すると、SourceOrdinal プロパティは -1 に設定されます。SourceOrdinal プロパティを設定すると、SourceColumn プロパティは null 参照 (Visual Basic の Nothing) に設定されます。

SourceColumn を NULL または空の文字列に設定すると、エラーになります。

参照

- SABulkCopyColumnMapping.SourceOrdinal プロパティ [SQL Anywhere .NET]120 ページ

SourceOrdinal プロパティ

データソース内のソースカラムの順序位置を取得または設定します。

Visual Basic の構文

```
Public Property SourceOrdinal As Integer
```

C# の構文

```
public int SourceOrdinal {get;set;}
```

備考

データソースのカラムの順序を指定する整数。プロパティが設定されていない場合は -1。

SourceColumn プロパティと SourceOrdinal プロパティは、相互に排他的です。直前に設定された値が優先されます。

SourceColumn プロパティを設定すると、SourceOrdinal プロパティは -1 に設定されます。SourceOrdinal プロパティを設定すると、SourceColumn プロパティは null 参照 (Visual Basic の Nothing) に設定されます。

参照

- SABulkCopyColumnMapping.SourceColumn プロパティ [SQL Anywhere .NET]120 ページ

SABulkCopyColumnMappingCollection クラス

System.Collections.CollectionBase から継承した SABulkCopyColumnMapping オブジェクトのコレクションです。

Visual Basic の構文

```
Public NotInheritable Class SABulkCopyColumnMappingCollection
    Inherits System.Collections.CollectionBase
```

C# の構文

```
public sealed class SABulkCopyColumnMappingCollection :
    System.Collections.CollectionBase
```

基本クラス

- System.Collections.CollectionBase

メンバー

継承されたメンバーを含む SABulkCopyColumnMappingCollection クラスのすべてのメンバー。

名前	説明
Add メソッド	指定された SABulkCopyColumnMapping オブジェクトをコレクションに追加します。
Clear method (System.Collections.CollectionBase から継承)	System.Collections.CollectionBase インスタンスからすべてのオブジェクトを削除します。
Contains メソッド	指定された SABulkCopyColumnMapping オブジェクトがコレクション内にあるかどうかを示す値を取得します。
CopyTo メソッド	SABulkCopyColumnMappingCollection の要素を、特定のインデックスを先頭に、SABulkCopyColumnMapping 項目の配列にコピーします。

名前	説明
GetEnumerator method (System.Collections.CollectionBase から継承)	System.Collections.CollectionBase インスタンスで反復処理する列挙子を返します。
IndexOf メソッド	コレクション内の指定された SABulkCopyColumnMapping オブジェクトのインデックスを取得または設定します。
OnClear method (System.Collections.CollectionBase から継承)	System.Collections.CollectionBase インスタンスの内容をクリアするときに、追加のカスタム処理を実行します。
OnClearComplete method (System.Collections.CollectionBase から継承)	System.Collections.CollectionBase インスタンスの内容をクリアした後に、追加のカスタム処理を実行します。
OnInsert method (System.Collections.CollectionBase から継承)	System.Collections.CollectionBase インスタンスに新しい要素を挿入する前に、追加のカスタム処理を実行します。
OnInsertComplete method (System.Collections.CollectionBase から継承)	System.Collections.CollectionBase インスタンスに新しい要素を挿入した後に、追加のカスタム処理を実行します。
OnRemove method (System.Collections.CollectionBase から継承)	System.Collections.CollectionBase インスタンスから要素を削除するときに、追加のカスタム処理を実行します。
OnRemoveComplete method (System.Collections.CollectionBase から継承)	System.Collections.CollectionBase インスタンスから要素を削除した後に、追加のカスタム処理を実行します。
OnSet method (System.Collections.CollectionBase から継承)	System.Collections.CollectionBase インスタンスに値を設定する前に、追加のカスタム処理を実行します。
OnSetComplete method (System.Collections.CollectionBase から継承)	System.Collections.CollectionBase インスタンスに値を設定した後に、追加のカスタム処理を実行します。
OnValidate method (System.Collections.CollectionBase から継承)	値を検証するときに、追加のカスタム処理を実行します。
Remove メソッド	指定された SABulkCopyColumnMapping 要素を SABulkCopyColumnMappingCollection から削除します。

名前	説明
RemoveAt メソッド	指定されたインデックス位置のマッピングをコレクションから削除します。
Capacity property (System.Collections.CollectionBase から継承)	System.Collections.CollectionBase に含めることのできる要素数を取得または設定します。
Count property (System.Collections.CollectionBase から継承)	System.Collections.CollectionBase インスタンスに含まれている要素数を取得します。
InnerList property (System.Collections.CollectionBase から継承)	System.Collections.CollectionBase インスタンスの要素リストに含まれている System.Collections.ArrayList を取得します。
List property (System.Collections.CollectionBase から継承)	System.Collections.CollectionBase インスタンスの要素リストに含まれている System.Collections.IList を取得します。
this プロパティ	指定されたインデックス位置の SABulkCopyColumnMapping オブジェクトを取得します。

備考

SABulkCopyColumnMappingCollection クラスは、.NET Compact Framework 2.0 では使用できません。

Implements: ICollection、IEnumerable、IList

Add メソッド

指定された SABulkCopyColumnMapping オブジェクトをコレクションに追加します。

オーバロードリスト

名前	説明
Add(int, int) メソッド	ソースカラムと送信先カラムの両方を指定する順序を使用して、新しい SABulkCopyColumnMapping オブジェクトを作成し、このマッピングをコレクションに追加します。

名前	説明
Add(int, string) メソッド	カラムの順序を使用してソースカラムを参照し、カラム名を使用して送信先カラムを参照する、新しい SABulkCopyColumnMapping オブジェクトを作成し、このマッピングをコレクションに追加します。
Add(SABulkCopyColumnMapping) メソッド	指定された SABulkCopyColumnMapping オブジェクトをコレクションに追加します。
Add(string, int) メソッド	カラム名を使用してソースカラムを参照し、カラムの順序を使用して送信先カラムを参照する、新しい SABulkCopyColumnMapping オブジェクトを作成し、このマッピングをコレクションに追加します。
Add(string, string) メソッド	ソースカラムと送信先カラムの両方を指定するカラム名を使用して、新しい SABulkCopyColumnMapping オブジェクトを作成し、このマッピングをコレクションに追加します。

Add(int, int) メソッド

ソースカラムと送信先カラムの両方を指定する順序を使用して、新しい SABulkCopyColumnMapping オブジェクトを作成し、このマッピングをコレクションに追加します。

Visual Basic の構文

```
Public Function Add(
    ByVal sourceColumnOrdinal As Integer,
    ByVal destinationColumnOrdinal As Integer
) As SABulkCopyColumnMapping
```

C# の構文

```
public SABulkCopyColumnMapping Add(
    int sourceColumnOrdinal,
    int destinationColumnOrdinal
)
```

パラメータ

- **sourceColumnOrdinal**　データソース内のソースカラムの序数位置。
- **destinationColumnOrdinal**　送信先テーブル内の送信先カラムの序数位置。

備考

SABulkCopyColumnMappingCollection クラスは、.NET Compact Framework 2.0 では使用できません。

Add(int, string) メソッド

カラムの順序を使用してソースカラムを参照し、カラム名を使用して送信先カラムを参照する、新しい SABulkCopyColumnMapping オブジェクトを作成し、このマッピングをコレクションに追加します。

Visual Basic の構文

```
Public Function Add(
    ByVal sourceColumnOrdinal As Integer,
    ByVal destinationColumn As String
) As SABulkCopyColumnMapping
```

C# の構文

```
public SABulkCopyColumnMapping Add(
    int sourceColumnOrdinal,
    string destinationColumn
)
```

パラメータ

- **sourceColumnOrdinal**　データソース内のソースカラムの序数位置。
- **destinationColumn**　送信先テーブル内の送信先カラムの名前。

備考

SABulkCopyColumnMappingCollection クラスは、.NET Compact Framework 2.0 では使用できません。

Add(SABulkCopyColumnMapping) メソッド

指定された SABulkCopyColumnMapping オブジェクトをコレクションに追加します。

Visual Basic の構文

```
Public Function Add(
    ByVal bulkCopyColumnMapping As SABulkCopyColumnMapping
) As SABulkCopyColumnMapping
```

C# の構文

```
public SABulkCopyColumnMapping Add(
    SABulkCopyColumnMapping bulkCopyColumnMapping
)
```

パラメータ

- **bulkCopyColumnMapping**　コレクションに追加される、マッピングを記述する SABulkCopyColumnMapping オブジェクト。

備考

SABulkCopyColumnMappingCollection クラスは、.NET Compact Framework 2.0 では使用できません。

参照

- SABulkCopyColumnMapping クラス [SQL Anywhere .NET]115 ページ

Add(string, int) メソッド

カラム名を使用してソースカラムを参照し、カラムの順序を使用して送信先カラムを参照する、新しい SABulkCopyColumnMapping オブジェクトを作成し、このマッピングをコレクションに追加します。

Visual Basic の構文

```
Public Function Add(
    ByVal sourceColumn As String,
    ByVal destinationColumnOrdinal As Integer
) As SABulkCopyColumnMapping
```

C# の構文

```
public SABulkCopyColumnMapping Add(
    string sourceColumn,
    int destinationColumnOrdinal
)
```

パラメータ

- **sourceColumn**　データソース内のソースカラムの名前。
- **destinationColumnOrdinal**　送信先テーブル内の送信先カラムの序数位置。

備考

カラムの順序または名前を使用してソースカラムと送信先カラムを参照する、新しいカラムマッピングを作成します。

SABulkCopyColumnMappingCollection クラスは、.NET Compact Framework 2.0 では使用できません。

Add(string, string) メソッド

ソースカラムと送信先カラムの両方を指定するカラム名を使用して、新しい SABulkCopyColumnMapping オブジェクトを作成し、このマッピングをコレクションに追加します。

Visual Basic の構文

```
Public Function Add(
    ByVal sourceColumn As String,
    ByVal destinationColumn As String
) As SABulkCopyColumnMapping
```

C# の構文

```
public SABulkCopyColumnMapping Add(
    string sourceColumn,
    string destinationColumn
)
```

パラメータ

- **sourceColumn**　データソース内のソースカラムの名前。
- **destinationColumn**　送信先テーブル内の送信先カラムの名前。

備考

SABulkCopyColumnMappingCollection クラスは、.NET Compact Framework 2.0 では使用できません。

Contains メソッド

指定された SABulkCopyColumnMapping オブジェクトがコレクション内にあるかどうかを示す値を取得します。

Visual Basic の構文

```
Public Function Contains(
    ByVal value As SABulkCopyColumnMapping
) As Boolean
```

C# の構文

```
public bool Contains(SABulkCopyColumnMapping value)
```

パラメータ

- **value**　有効な SABulkCopyColumnMapping オブジェクト。

戻り値

指定のマッピングがコレクション内にある場合は true、ない場合は false。

CopyTo メソッド

SABulkCopyColumnMappingCollection の要素を、特定のインデックスを先頭に、SABulkCopyColumnMapping 項目の配列にコピーします。

Visual Basic の構文

```
Public Sub CopyTo(
    ByVal array As SABulkCopyColumnMapping(),
    ByVal index As Integer
)
```

C# の構文

```
public void CopyTo(SABulkCopyColumnMapping[] array, int index)
```

パラメータ

- **array**　SABulkCopyColumnMappingCollection の要素のコピー先である、1 次元の SABulkCopyColumnMapping 配列。配列のインデックスは 0 から始まります。
- **index**　コピーが開始される、配列内の 0 から始まるインデックス。

IndexOf メソッド

コレクション内の指定された SABulkCopyColumnMapping オブジェクトのインデックスを取得または設定します。

Visual Basic の構文

```
Public Function IndexOf(
    ByVal value As SABulkCopyColumnMapping
) As Integer
```

C# の構文

```
public int IndexOf(SABulkCopyColumnMapping value)
```

パラメータ

- **value**　検索対象の SABulkCopyColumnMapping オブジェクト。

戻り値

カラムマッピングの 0 から始まるインデックス、またはコレクション内にカラムマッピングが見つからない場合は -1 が返されます。

Remove メソッド

指定された SABulkCopyColumnMapping 要素を SABulkCopyColumnMappingCollection から削除します。

Visual Basic の構文

```
Public Sub Remove(ByVal value As SABulkCopyColumnMapping)
```

C# の構文

```
public void Remove(SABulkCopyColumnMapping value)
```

 Copyright © 2013, SAP AG or an SAP affiliate company.

パラメータ

- **value**　コレクションから削除する SABulkCopyColumnMapping オブジェクト。

RemoveAt メソッド

指定されたインデックス位置のマッピングをコレクションから削除します。

Visual Basic の構文

```
Public Shadows Sub RemoveAt(ByVal index As Integer)
```

C# の構文

```
public new void RemoveAt(int index)
```

パラメータ

- **index**　コレクションから削除する SABulkCopyColumnMapping オブジェクトの、0 から始まるインデックス。

this プロパティ

指定されたインデックス位置の SABulkCopyColumnMapping オブジェクトを取得します。

Visual Basic の構文

```
Public ReadOnly Property Item(
    ByVal index As Integer
) As SABulkCopyColumnMapping
```

C# の構文

```
public SABulkCopyColumnMapping this[int index] {get;}
```

パラメータ

- **index**　検索する SABulkCopyColumnMapping オブジェクトの、0 から始まるインデックス。

戻り値

SABulkCopyColumnMapping オブジェクトが返されます。

SACommand クラス

SQL Anywhere データベースに対して実行される SQL 文またはストアドプロシージャです。

Visual Basic の構文

```
Public NotInheritable Class SACommand
    Inherits System.Data.Common.DbCommand
    Implements System.ICloneable
```

C# の構文

```
public sealed class SACommand :
    System.Data.Common.DbCommand,
    System.ICloneable
```

基本クラス

- System.Data.Common.DbCommand
- System.ICloneable

メンバー

継承されたメンバーを含む SACommand クラスのすべてのメンバー。

名前	説明
SACommand コンストラクタ	SACommand オブジェクトを初期化します。
BeginExecuteNonQuery メソッド	この SACommand で記述される SQL 文またはストアドプロシージャの非同期実行を開始します。
BeginExecuteReader メソッド	この SACommand で記述される SQL 文またはストアドプロシージャの非同期実行を開始し、データベースサーバから 1 つまたは複数の結果セットを取得します。
Cancel メソッド	SACommand オブジェクトの実行をキャンセルします。
CreateParameter メソッド	SACommand オブジェクトにパラメータを指定するために SAParameter オブジェクトを提供します。
EndExecuteNonQuery メソッド	SQL 文またはストアドプロシージャの非同期実行を終了します。
EndExecuteReader メソッド	SQL 文またはストアドプロシージャの非同期実行を終了し、要求された SADataReader を返します。
ExecuteDbDataReader method (System.Data.Common.DbCommand から継承)	接続に対してコマンドテキストを実行します。

名前	説明
ExecuteDbDataReaderAsync method (System.Data.Common.DbCommand から継承)	プロバイダはこのメソッドを実装し、Overload:System.Data.Common.DbCommand.ExecuteReader オーバロードの非デフォルト実装を提供する必要があります。デフォルトの実装は、同期 System.Data.Common.DbCommand.ExecuteReader メソッドを呼び出し、呼び出しスレッドをブロックして完了タスクを返します。
ExecuteNonQuery メソッド	結果セットを返さない文 (INSERT、UPDATE、DELETE など) や、データ定義文を実行します。
ExecuteNonQueryAsync method (System.Data.Common.DbCommand から継承)	接続オブジェクトで SQL 文を実行する、System.Data.Common.DbCommand.ExecuteNonQuery の非同期バージョン。
ExecuteReader メソッド	結果セットを返す SQL 文を実行します。
ExecuteReaderAsync method (System.Data.Common.DbCommand から継承)	System.Data.Common.DbCommand.Connection に対して System.Data.Common.DbCommand.CommandText を実行し、CancellationToken.None で System.Data.Common.DbDataReader.Invokes System.Data.Common.DbCommand.ExecuteDbDataReaderAsync(System.Data.CommandBehavior,System.Threading.CancellationToken)を返す Overload:System.Data.Common.DbCommand.ExecuteReader の非同期バージョン。
ExecuteScalar メソッド	単一の値を返す文を実行します。
ExecuteScalarAsync method (System.Data.Common.DbCommand から継承)	クエリを実行し、クエリによって返された結果セットの最初のローの最初のカラムを返す、System.Data.Common.DbCommand.ExecuteScalar の非同期バージョン。
Prepare メソッド	データソース上で SACommand を準備またはコンパイルします。
ResetCommandTimeout メソッド	CommandTimeout プロパティをデフォルト値の 30 秒にリセットします。
CommandText プロパティ	SQL 文またはストアドプロシージャのテキストを取得または設定します。

名前	説明
CommandTimeout プロパティ	この機能は SQL Anywhere .NET データプロバイダでサポートされていません。
CommandType プロパティ	SACommand によって示されるコマンドのタイプを取得または設定します。
Connection プロパティ	SACommand オブジェクトが適用される接続オブジェクトを取得または設定します。
DesignTimeVisible プロパティ	Windows Form Designer 制御で SACommand を参照できるようにするかどうかを指定する値を取得または設定します。
Parameters プロパティ	現在の文のパラメータのコレクションです。
Transaction プロパティ	SACommand が実行される SATransaction オブジェクトを指定します。
UpdatedRowSource プロパティ	SADataAdapter の Update メソッドによって使用されるときにコマンドの結果が DataRow に適用される方法を取得または設定します。

備考

Implements: IDbCommand、ICloneable

詳細については、「データへのアクセスとデータの操作」47 ページを参照してください。

SACommand コンストラクタ

SACommand オブジェクトを初期化します。

オーバロードリスト

名前	説明
SACommand() コンストラクタ	SACommand オブジェクトを初期化します。
SACommand(string) コンストラクタ	SACommand オブジェクトを初期化します。
SACommand(string, SAConnection) コンストラクタ	SQL Anywhere データベースに対して実行される SQL 文またはストアドプロシージャです。

名前	説明
SACommand(string, SAConnection, SATransaction) コンストラクタ	SQL Anywhere データベースに対して実行される SQL 文またはストアドプロシージャです。

SACommand() コンストラクタ

SACommand オブジェクトを初期化します。

Visual Basic の構文

```
Public Sub New()
```

C# の構文

```
public SACommand()
```

SACommand(string) コンストラクタ

SACommand オブジェクトを初期化します。

Visual Basic の構文

```
Public Sub New(ByVal cmdText As String)
```

C# の構文

```
public SACommand(string cmdText)
```

パラメータ

- **cmdText**　SQL 文またはストアドプロシージャのテキスト。パラメータ化された文の場合、疑問符 (?) プレースホルダを使用してパラメータを渡します。

SACommand(string, SAConnection) コンストラクタ

SQL Anywhere データベースに対して実行される SQL 文またはストアドプロシージャです。

Visual Basic の構文

```
Public Sub New(
    ByVal cmdText As String,
    ByVal connection As SAConnection
)
```

C# の構文

```
public SACommand(string cmdText, SAConnection connection)
```

パラメータ

- **cmdText**　SQL 文またはストアドプロシージャのテキスト。パラメータ化された文の場合、疑問符 (?) プレースホルダを使用してパラメータを渡します。
- **connection**　現在の接続。

SACommand(string, SAConnection, SATransaction) コンストラクタ

SQL Anywhere データベースに対して実行される SQL 文またはストアドプロシージャです。

Visual Basic の構文

```
Public Sub New(
    ByVal cmdText As String,
    ByVal connection As SAConnection,
    ByVal transaction As SATransaction
)
```

C# の構文

```
public SACommand(
    string cmdText,
    SAConnection connection,
    SATransaction transaction
)
```

パラメータ

- **cmdText**　SQL 文またはストアドプロシージャのテキスト。パラメータ化された文の場合、疑問符 (?) プレースホルダを使用してパラメータを渡します。
- **connection**　現在の接続。
- **transaction**　SAConnection が実行される SATransaction オブジェクト。

参照

- SATransaction クラス [SQL Anywhere .NET]324 ページ

BeginExecuteNonQuery メソッド

この SACommand で記述される SQL 文またはストアドプロシージャの非同期実行を開始します。

オーバロードリスト

名前	説明
BeginExecuteNonQuery() メソッド	この SACommand で記述される SQL 文またはストアドプロシージャの非同期実行を開始します。

名前	説明
BeginExecuteNonQuery(AsyncCallback, object) メソッド	コールバックプロシージャとステータス情報を指定し、この SACommand で記述される SQL 文またはストアドプロシージャの非同期実行を開始します。

BeginExecuteNonQuery() メソッド

この SACommand で記述される SQL 文またはストアドプロシージャの非同期実行を開始します。

Visual Basic の構文

```
Public Function BeginExecuteNonQuery() As IAsyncResult
```

C# の構文

```
public IAsyncResult BeginExecuteNonQuery()
```

戻り値

ポーリング、結果の待機、または両方に使用できる System.IAsyncResult。この値は、影響を受けたローの数を返す EndExecuteNonQuery(IAsyncResult) を起動する場合にも必要です。

例外

- **SAException クラス**　コマンドテキストの実行中に発生したエラー。

備考

非同期コマンドでは、パラメータの順序は CommandText と一致している必要があります。

参照

- SACommand.EndExecuteNonQuery メソッド [SQL Anywhere .NET]141 ページ

BeginExecuteNonQuery(AsyncCallback, object) メソッド

コールバックプロシージャとステータス情報を指定し、この SACommand で記述される SQL 文またはストアドプロシージャの非同期実行を開始します。

Visual Basic の構文

```
Public Function BeginExecuteNonQuery(
    ByVal callback As AsyncCallback,
    ByVal stateObject As Object
) As IAsyncResult
```

C# の構文

```
public IAsyncResult BeginExecuteNonQuery(
    AsyncCallback callback,
```

```
    object stateObject
)
```

パラメータ

- **callback**　コマンドの実行が終了すると起動される System.AsyncCallback デリゲート。コールバックが必要ないことを示すには、NULL (Microsoft Visual Basic の場合は Nothing) を渡します。

- **stateObject**　コールバックプロシージャに渡される、ユーザ定義のステータスオブジェクト。コールバックプロシージャからこのオブジェクトを取得するには、System.IAsyncResult.AsyncState プロパティを使用します。

戻り値

ポーリング、結果の待機、または両方に使用できる System.IAsyncResult。この値は、影響を受けたローの数を返す EndExecuteNonQuery(IAsyncResult) を起動する場合にも必要です。

例外

- **SAException クラス**　コマンドテキストの実行中に発生したエラー。

備考

非同期コマンドでは、パラメータの順序は CommandText と一致している必要があります。

参照

- SACommand.EndExecuteNonQuery メソッド [SQL Anywhere .NET]141 ページ

BeginExecuteReader メソッド

この SACommand で記述される SQL 文またはストアドプロシージャの非同期実行を開始し、データベースサーバから 1 つまたは複数の結果セットを取得します。

オーバロードリスト

名前	説明
BeginExecuteReader() メソッド	この SACommand で記述される SQL 文またはストアドプロシージャの非同期実行を開始し、データベースサーバから 1 つまたは複数の結果セットを取得します。
BeginExecuteReader(AsyncCallback, object) メソッド	コールバックプロシージャとステータス情報を指定し、この SACommand オブジェクトで記述される SQL 文の非同期実行を開始し、結果セットを取得します。

 Copyright © 2013, SAP AG or an SAP affiliate company.

名前	説明
BeginExecuteReader(AsyncCallback, object, CommandBehavior) メソッド	この SACommand で記述される SQL 文またはストアドプロシージャの非同期実行を開始し、サーバから 1 つまたは複数の結果セットを取得します。
BeginExecuteReader(CommandBehavior) メソッド	この SACommand で記述される SQL 文またはストアドプロシージャの非同期実行を開始し、サーバから 1 つまたは複数の結果セットを取得します。

BeginExecuteReader() メソッド

この SACommand で記述される SQL 文またはストアドプロシージャの非同期実行を開始し、データベースサーバから 1 つまたは複数の結果セットを取得します。

Visual Basic の構文

```
Public Function BeginExecuteReader() As IAsyncResult
```

C# の構文

```
public IAsyncResult BeginExecuteReader()
```

戻り値

ポーリング、結果の待機、または両方に使用できる System.IAsyncResult。この値は、返されたローを取得するために使用する SADataReader オブジェクトを返す、EndExecuteReader(IAsyncResult) を起動する場合にも必要です。

例外

- **SAException クラス**　コマンドテキストの実行中に発生したエラー。

備考

非同期コマンドでは、パラメータの順序は CommandText と一致している必要があります。

参照

- SACommand.EndExecuteReader メソッド [SQL Anywhere .NET]143 ページ
- SADataReader クラス [SQL Anywhere .NET]224 ページ

BeginExecuteReader(AsyncCallback, object) メソッド

コールバックプロシージャとステータス情報を指定し、この SACommand オブジェクトで記述される SQL 文の非同期実行を開始し、結果セットを取得します。

Visual Basic の構文

```
Public Function BeginExecuteReader(
    ByVal callback As AsyncCallback,
    ByVal stateObject As Object
) As IAsyncResult
```

C# の構文

```
public IAsyncResult BeginExecuteReader(
    AsyncCallback callback,
    object stateObject
)
```

パラメータ

- **callback**　コマンドの実行が終了すると起動される System.AsyncCallback デリゲート。コールバックが必要ないことを示すには、NULL (Microsoft Visual Basic の場合は Nothing) を渡します。

- **stateObject**　コールバックプロシージャに渡される、ユーザ定義のステータスオブジェクト。コールバックプロシージャからこのオブジェクトを取得するには、System.IAsyncResult.AsyncState プロパティを使用します。

戻り値

ポーリング、結果の待機、または両方に使用できる System.IAsyncResult。この値は、返されたローを取得するために使用する SADataReader オブジェクトを返す、EndExecuteReader(IAsyncResult) を起動する場合にも必要です。

例外

- **SAException クラス**　コマンドテキストの実行中に発生したエラー。

備考

非同期コマンドでは、パラメータの順序は CommandText と一致している必要があります。

参照

- SACommand.EndExecuteReader メソッド [SQL Anywhere .NET]143 ページ
- SADataReader クラス [SQL Anywhere .NET]224 ページ

BeginExecuteReader(AsyncCallback, object, CommandBehavior) メソッド

この SACommand で記述される SQL 文またはストアドプロシージャの非同期実行を開始し、サーバから 1 つまたは複数の結果セットを取得します。

Visual Basic の構文

```
Public Function BeginExecuteReader(
    ByVal callback As AsyncCallback,
    ByVal stateObject As Object,
    ByVal behavior As CommandBehavior
) As IAsyncResult
```

C# の構文

```
public IAsyncResult BeginExecuteReader(
    AsyncCallback callback,
    object stateObject,
    CommandBehavior behavior
)
```

パラメータ

- **callback**　コマンドの実行が終了すると起動される System.AsyncCallback デリゲート。コールバックが必要ないことを示すには、NULL (Microsoft Visual Basic の場合は Nothing) を渡します。
- **stateObject**　コールバックプロシージャに渡される、ユーザ定義のステータスオブジェクト。コールバックプロシージャからこのオブジェクトを取得するには、System.IAsyncResult.AsyncState プロパティを使用します。
- **behavior**　クエリの結果の記述と、接続への影響の記述の System.Data.CommandBehavior フラグのビット単位の組み合わせ。

戻り値

ポーリング、結果の待機、または両方に使用できる System.IAsyncResult。この値は、返されたローを取得するために使用する SADataReader オブジェクトを返す、EndExecuteReader(IAsyncResult) を起動する場合にも必要です。

例外

- **SAException クラス**　コマンドテキストの実行中に発生したエラー。

備考

非同期コマンドでは、パラメータの順序は CommandText と一致している必要があります。

参照

- SACommand.EndExecuteReader メソッド [SQL Anywhere .NET]143 ページ
- SADataReader クラス [SQL Anywhere .NET]224 ページ

BeginExecuteReader(CommandBehavior) メソッド

この SACommand で記述される SQL 文またはストアドプロシージャの非同期実行を開始し、サーバから 1 つまたは複数の結果セットを取得します。

Visual Basic の構文

```
Public Function BeginExecuteReader(
    ByVal behavior As CommandBehavior
) As IAsyncResult
```

C# の構文

```
public IAsyncResult BeginExecuteReader(CommandBehavior behavior)
```

パラメータ

- **behavior**　クエリの結果の記述と、接続への影響の記述の System.Data.CommandBehavior フラグのビット単位の組み合わせ。

戻り値

ポーリング、結果の待機、または両方に使用できる System.IAsyncResult。この値は、返されたローを取得するために使用する SADataReader オブジェクトを返す、EndExecuteReader(IAsyncResult) を起動する場合にも必要です。

例外

- **SAException クラス**　コマンドテキストの実行中に発生したエラー。

備考

非同期コマンドでは、パラメータの順序は CommandText と一致している必要があります。

参照

- SACommand.EndExecuteReader メソッド [SQL Anywhere .NET]143 ページ
- SADataReader クラス [SQL Anywhere .NET]224 ページ

Cancel メソッド

SACommand オブジェクトの実行をキャンセルします。

Visual Basic の構文

```
Public Overrides Sub Cancel()
```

C# の構文

```
public override void Cancel()
```

備考

キャンセル対象がない場合は、何も行われません。実行中のコマンドがある場合は、「文の実行がユーザによって中断させられました。」という例外がスローされます。

CreateParameter メソッド

SACommand オブジェクトにパラメータを指定するために SAParameter オブジェクトを提供します。

Visual Basic の構文

```
Public Shadows Function CreateParameter() As SAParameter
```

C# の構文

```
public new SAParameter CreateParameter()
```

 Copyright © 2013, SAP AG or an SAP affiliate company.

戻り値

SAParameter オブジェクトとして返される新しいパラメータ。

備考

ストアドプロシージャとその他一部の SQL 文は、疑問符 (?) によって文のテキストに示されているパラメータを使用できます。

CreateParameter メソッドは、SAParameter オブジェクトを提供します。SAParameter にプロパティを設定し、パラメータの値やデータ型などを指定できます。

参照

- SAParameter クラス [SQL Anywhere .NET]278 ページ

EndExecuteNonQuery メソッド

SQL 文またはストアドプロシージャの非同期実行を終了します。

Visual Basic の構文

```
Public Function EndExecuteNonQuery(
    ByVal asyncResult As IAsyncResult
) As Integer
```

C# の構文

```
public int EndExecuteNonQuery(IAsyncResult asyncResult)
```

パラメータ

- **asyncResult**　SACommand.BeginExecuteNonQuery への呼び出しによって返される IAsyncResult。

戻り値

影響されるローの数 (SACommand.ExecuteNonQuery と同じ動作)。

例外

- **ArgumentException**　asyncResult パラメータが null (Microsoft Visual Basic の Nothing) です。
- **InvalidOperationException**　1 回のコマンド実行に対して SACommand.EndExecuteNonQuery(IAsyncResult) が複数回呼び出されたか、メソッドがその実行メソッドと一致しませんでした。

備考

BeginExecuteNonQuery を呼び出すごとに、EndExecuteNonQuery を呼び出す必要があります。呼び出しは、BeginExecuteNonQuery が返されてから行います。ADO.NET はスレッドに対応していないため、各自で BeginExecuteNonQuery が返されたことを確認する必要があります。EndExecuteNonQuery に渡される IAsyncResult は、完了する BeginExecuteNonQuery 呼び出しから返される IAsyncResult と同じです。EndExecuteNonQuery を呼び出して、BeginExecuteReader への呼び出しを終了すると、エラーになります。逆についても同様です。

コマンドの実行中にエラーが発生すると、EndExecuteNonQuery が呼び出されるときに例外がスローされます。

実行の完了を待機するには、4 とおりの方法があります。

(1) EndExecuteNonQuery を呼び出す。

EndExecuteNonQuery を呼び出すと、コマンドが完了するまでブロックします。次に例を示します。

```
SAConnection conn = new SAConnection("DSN=SQL Anywhere 16 Demo");
conn.Open();
SACommand cmd = new SACommand(
   "UPDATE Departments"
      + " SET DepartmentName = 'Engineering'"
      + " WHERE DepartmentID=100",
   conn );
IAsyncResult res = cmd.BeginExecuteNonQuery();
// perform other work
// this will block until the command completes
int rowCount = cmd.EndExecuteNonQuery( res );
```

(2) IAsyncResult の IsCompleted プロパティをポーリングする。

IAsyncResult の IsCompleted プロパティをポーリングできます。次に例を示します。

```
SAConnection conn = new SAConnection("DSN=SQL Anywhere 16 Demo");
conn.Open();
SACommand cmd = new SACommand(
   "UPDATE Departments"
      + " SET DepartmentName = 'Engineering'"
      + " WHERE DepartmentID=100",
   conn );
IAsyncResult res = cmd.BeginExecuteNonQuery();
while( !res.IsCompleted ) {
   // do other work
}
// this will not block because the command is finished
int rowCount = cmd.EndExecuteNonQuery( res );
```

(3) IAsyncResult.AsyncWaitHandle プロパティを使用して同期オブジェクトを取得する。

IAsyncResult.AsyncWaitHandle プロパティを使用して同期オブジェクトを取得し、その状態で待機できます。次に例を示します。

```
SAConnection conn = new SAConnection("DSN=SQL Anywhere 16 Demo");
conn.Open();
SACommand cmd = new SACommand(
   "UPDATE Departments"
      + " SET DepartmentName = 'Engineering'"
      + " WHERE DepartmentID=100",
   conn );
IAsyncResult res = cmd.BeginExecuteNonQuery();
// perform other work
WaitHandle wh = res.AsyncWaitHandle;
wh.WaitOne();
// this will not block because the command is finished
int rowCount = cmd.EndExecuteNonQuery( res );
```

(4) BeginExecuteNonQuery の呼び出し時にコールバック関数を指定する。

BeginExecuteNonQuery の呼び出し時にコールバック関数を指定できます。次に例を示します。

```
private void callbackFunction( IAsyncResult ar ) {
   SACommand cmd = (SACommand) ar.AsyncState;
   // this won't block since the command has completed
   int rowCount = cmd.EndExecuteNonQuery( ar );
}

// elsewhere in the code
private void DoStuff() {
   SAConnection conn = new SAConnection("DSN=SQL Anywhere 16 Demo");
   conn.Open();
   SACommand cmd = new SACommand(
      "UPDATE Departments"
         + " SET DepartmentName = 'Engineering'"
         + " WHERE DepartmentID=100",
      conn );
   IAsyncResult res = cmd.BeginExecuteNonQuery( callbackFunction, cmd );
   // perform other work.  The callback function will be
   // called when the command completes
}
```

コールバック関数は別のスレッドで実行するため、スレッド化されたプログラム内でのユーザインタフェースの更新に関する通常の注意が適用されます。

参照

- SACommand.BeginExecuteNonQuery メソッド [SQL Anywhere .NET]134 ページ

EndExecuteReader メソッド

SQL 文またはストアドプロシージャの非同期実行を終了し、要求された SADataReader を返します。

Visual Basic の構文

```
Public Function EndExecuteReader(
    ByVal asyncResult As IAsyncResult
) As SADataReader
```

C# の構文

```
public SADataReader EndExecuteReader(IAsyncResult asyncResult)
```

パラメータ

- **asyncResult**　SACommand.BeginExecuteReader への呼び出しによって返される IAsyncResult。

戻り値

要求されたローの取り出しに使用する SADataReader オブジェクト (SACommand.ExecuteReader と同じ動作)。

例外

- **ArgumentException**　asyncResult パラメータが null (Microsoft Visual Basic の Nothing) です。

- **InvalidOperationException**　1 回のコマンド実行に対して SACommand.EndExecuteReader(IAsyncResult) が複数回呼び出されたか、メソッドがその実行メソッドと一致しませんでした。

備考

BeginExecuteReader を呼び出すごとに、EndExecuteReader を呼び出す必要があります。呼び出しは、BeginExecuteReader が返されてから行います。ADO.NET はスレッドに対応していないため、各自で BeginExecuteReader が返されたことを確認する必要があります。EndExecuteReader に渡される IAsyncResult は、完了する BeginExecuteReader 呼び出しから返される IAsyncResult と同じです。EndExecuteReade を呼び出して、BeginExecuteNonQuery への呼び出しを終了すると、エラーになります。逆についても同様です。

コマンドの実行中にエラーが発生すると、EndExecuteReader が呼び出されるときに例外がスローされます。

実行の完了を待機するには、4 とおりの方法があります。

(1) EndExecuteReader を呼び出す。

EndExecuteReader を呼び出すと、コマンドが完了するまでブロックします。次に例を示します。

```
SAConnection conn = new SAConnection("DSN=SQL Anywhere 16 Demo");
conn.Open();
SACommand cmd = new SACommand( "SELECT * FROM Departments", conn );
IAsyncResult res = cmd.BeginExecuteReader();
// perform other work
// this blocks until the command completes
SADataReader reader = cmd.EndExecuteReader( res );
```

(2) IAsyncResult の IsCompleted プロパティをポーリングする。

IAsyncResult の IsCompleted プロパティをポーリングできます。次に例を示します。

```
SAConnection conn = new SAConnection("DSN=SQL Anywhere 16 Demo");
conn.Open();
SACommand cmd = new SACommand( "SELECT * FROM Departments", conn );
IAsyncResult res = cmd.BeginExecuteReader();
while( !res.IsCompleted ) {
    // do other work
}
// this does not block because the command is finished
SADataReader reader = cmd.EndExecuteReader( res );
```

(3) IAsyncResult.AsyncWaitHandle プロパティを使用して同期オブジェクトを取得する。

IAsyncResult.AsyncWaitHandle プロパティを使用して同期オブジェクトを取得し、その状態で待機できます。次に例を示します。

```
SAConnection conn = new SAConnection("DSN=SQL Anywhere 16 Demo");
conn.Open();
SACommand cmd = new SACommand( "SELECT * FROM Departments", conn );
IAsyncResult res = cmd.BeginExecuteReader();
// perform other work
WaitHandle wh = res.AsyncWaitHandle;
wh.WaitOne();
```

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

```
// this does not block because the command is finished
SADataReader reader = cmd.EndExecuteReader( res );
```

(4) BeginExecuteReader の呼び出し時にコールバック関数を指定する。

BeginExecuteReader の呼び出し時にコールバック関数を指定できます。次に例を示します。

```
private void callbackFunction( IAsyncResult ar ) {
    SACommand cmd = (SACommand) ar.AsyncState;
    // this does not block since the command has completed
    SADataReader reader = cmd.EndExecuteReader();
}

// elsewhere in the code
private void DoStuff() {
    SAConnection conn = new SAConnection("DSN=SQL Anywhere 16 Demo");
    conn.Open();
    SACommand cmd = new SACommand( "SELECT * FROM Departments", conn );
    IAsyncResult res = cmd.BeginExecuteReader( callbackFunction, cmd );
    // perform other work.  The callback function will be
    // called when the command completes
}
```

コールバック関数は別のスレッドで実行するため、スレッド化されたプログラム内でのユーザインタフェースの更新に関する通常の注意が適用されます。

参照

- SACommand.BeginExecuteReader メソッド [SQL Anywhere .NET]136 ページ
- SADataReader クラス [SQL Anywhere .NET]224 ページ

ExecuteNonQuery メソッド

結果セットを返さない文 (INSERT、UPDATE、DELETE など) や、データ定義文を実行します。

Visual Basic の構文

```
Public Overrides Function ExecuteNonQuery() As Integer
```

C# の構文

```
public override int ExecuteNonQuery()
```

戻り値

影響を受けたローの数。

備考

ExecuteNonQuery を使用して、DataSet を使用しないでデータベースのデータを変更します。これを行うには、UPDATE、INSERT、または DELETE 文を実行します。

ExecuteNonQuery はローを返しませんが、パラメータにマッピングされる出力パラメータまたは戻り値にはデータが入力されます。

UPDATE、INSERT、DELETE 文の場合、戻り値はコマンドの影響を受けるローの数です。その他すべての文のタイプおよびロールバックの場合、戻り値は -1 です。

参照

- SACommand.ExecuteReader メソッド [SQL Anywhere .NET]146 ページ

ExecuteReader メソッド

結果セットを返す SQL 文を実行します。

オーバロードリスト

名前	説明
ExecuteReader() メソッド	結果セットを返す SQL 文を実行します。
ExecuteReader(CommandBehavior) メソッド	結果セットを返す SQL 文を実行します。

ExecuteReader() メソッド

結果セットを返す SQL 文を実行します。

Visual Basic の構文

```
Public Shadows Function ExecuteReader() As SADataReader
```

C# の構文

```
public new SADataReader ExecuteReader()
```

戻り値

SADataReader オブジェクトとして返される結果セット。

備考

この文は、必要に応じて CommandText と Parameters を持つ現在の SACommand オブジェクトです。SADataReader オブジェクトは、読み込み専用、前方専用の結果セットです。修正可能な結果セットの場合、SADataAdapter を使用します。

参照

- SACommand.ExecuteNonQuery メソッド [SQL Anywhere .NET]145 ページ
- SADataReader クラス [SQL Anywhere .NET]224 ページ
- SADataAdapter クラス [SQL Anywhere .NET]215 ページ
- SACommand.CommandText プロパティ [SQL Anywhere .NET]149 ページ
- SACommand.Parameters プロパティ [SQL Anywhere .NET]151 ページ

ExecuteReader(CommandBehavior) メソッド

結果セットを返す SQL 文を実行します。

Visual Basic の構文

```
Public Shadows Function ExecuteReader(
    ByVal behavior As CommandBehavior
) As SADataReader
```

C# の構文

```
public new SADataReader ExecuteReader(CommandBehavior behavior)
```

パラメータ

- **behavior** CloseConnection、Default、KeyInfo、SchemaOnly、SequentialAccess、SingleResult、SingleRow のいずれか 1 つ。このパラメータの詳細については、.NET Framework のマニュアルの CommandBehavior 列挙を参照してください。

戻り値

SADataReader オブジェクトとして返される結果セット。

備考

この文は、必要に応じて CommandText と Parameters を持つ現在の SACommand オブジェクトです。SADataReader オブジェクトは、読み込み専用、前方専用の結果セットです。修正可能な結果セットの場合、SADataAdapter を使用します。

参照

- SACommand.ExecuteNonQuery メソッド [SQL Anywhere .NET]145 ページ
- SADataReader クラス [SQL Anywhere .NET]224 ページ
- SADataAdapter クラス [SQL Anywhere .NET]215 ページ
- SACommand.CommandText プロパティ [SQL Anywhere .NET]149 ページ
- SACommand.Parameters プロパティ [SQL Anywhere .NET]151 ページ

ExecuteScalar メソッド

単一の値を返す文を実行します。

Visual Basic の構文

```
Public Overrides Function ExecuteScalar() As Object
```

C# の構文

```
public override object ExecuteScalar()
```

戻り値

結果セットの最初のローの最初のカラム。結果セットが空の場合は NULL 参照。

備考

複数のローとカラムを返すクエリでこのメソッドが呼び出されると、最初のローの最初のカラムのみが返されます。

Prepare メソッド

データソース上で SACommand を準備またはコンパイルします。

Visual Basic の構文

```
Public Overrides Sub Prepare()
```

C# の構文

```
public override void Prepare()
```

備考

Prepare を呼び出してから ExecuteNonQuery、ExecuteReader、ExecuteScalar のいずれかのメソッドを呼び出すと、Size プロパティによって指定されている値よりも大きいパラメータ値は、元々指定されているパラメータのサイズに自動的にトランケートされ、トランケーションエラーは返されません。

トランケーションは次のデータ型に対してのみ実行されます。

- CHAR
- VARCHAR
- LONG VARCHAR
- TEXT
- NCHAR
- NVARCHAR
- LONG NVARCHAR
- NTEXT
- BINARY
- LONG BINARY
- VARBINARY
- IMAGE

Size プロパティの指定がなく、デフォルト値が使用されている場合、データはトランケートされません。

参照

- SACommand.ExecuteNonQuery メソッド [SQL Anywhere .NET]145 ページ
- SACommand.ExecuteReader メソッド [SQL Anywhere .NET]146 ページ
- SACommand.ExecuteScalar メソッド [SQL Anywhere .NET]147 ページ

ResetCommandTimeout メソッド

CommandTimeout プロパティをデフォルト値の 30 秒にリセットします。

Visual Basic の構文

```
Public Sub ResetCommandTimeout()
```

C# の構文

```
public void ResetCommandTimeout()
```

CommandText プロパティ

SQL 文またはストアドプロシージャのテキストを取得または設定します。

Visual Basic の構文

```
Public Overrides Property CommandText As String
```

C# の構文

```
public override string CommandText {get;set;}
```

備考

実行する SQL 文またはストアドプロシージャの名前。デフォルトは、空の文字列です。

参照

- SACommand.SACommand コンストラクタ [SQL Anywhere .NET]132 ページ

CommandTimeout プロパティ

この機能は SQL Anywhere .NET データプロバイダでサポートされていません。

Visual Basic の構文

```
Public Overrides Property CommandTimeout As Integer
```

C# の構文

```
public override int CommandTimeout {get;set;}
```

備考

要求タイムアウトを設定するには、次の例を使用します。

```
cmd.CommandText = "SET OPTION request_timeout = 30";
  cmd.ExecuteNonQuery();
```

CommandType プロパティ

SACommand によって示されるコマンドのタイプを取得または設定します。

Visual Basic の構文

```
Public Overrides Property CommandType As CommandType
```

C# の構文

```
public override CommandType CommandType {get;set;}
```

備考

System.Data.CommandType 値の 1 つ。デフォルトは System.Data.CommandType.Text です。

サポートされているコマンドタイプは、次のとおりです。

- System.Data.CommandType.StoredProcedure。この CommandType を指定する場合、コマンドテキストはストアドプロシージャの名前にし、引数はすべて SAParameter オブジェクトとして指定してください。
- System.Data.CommandType.Text。これはデフォルト値です。

CommandType プロパティを StoredProcedure に設定する場合、CommandText プロパティをストアドプロシージャの名前に設定してください。Execute メソッドの 1 つを呼び出すと、このストアドプロシージャが実行されます。

疑問符 (?) プレースホルダを使用してパラメータを渡します。次に例を示します。

```
SELECT * FROM Customers WHERE ID = ?
```

SAParameter オブジェクトが SAParameterCollection に追加される順序は、パラメータの疑問符の位置にそのまま対応している必要があります。

Connection プロパティ

SACommand オブジェクトが適用される接続オブジェクトを取得または設定します。

Visual Basic の構文

```
Public Shadows Property Connection As SAConnection
```

C# の構文

```
public new SAConnection Connection {get;set;}
```

備考

デフォルト値は NULL 参照です。Visual Basic では、これは Nothing です。

DesignTimeVisible プロパティ

Windows Form Designer 制御で SACommand を参照できるようにするかどうかを指定する値を取得または設定します。

Visual Basic の構文

```
Public Overrides Property DesignTimeVisible As Boolean
```

C# の構文

```
public override bool DesignTimeVisible {get;set;}
```

備考

デフォルトは true です。

SACommand インスタンスを参照できるようにする場合は true、このインスタンスを参照できないようにする場合は false です。デフォルトは false です。

Parameters プロパティ

現在の文のパラメータのコレクションです。

Visual Basic の構文

```
Public ReadOnly Shadows Property Parameters As SAParameterCollection
```

C# の構文

```
public new SAParameterCollection Parameters {get;}
```

備考

CommandText で疑問符を使用してパラメータを示します。

SQL 文またはストアドプロシージャのパラメータ。デフォルト値は空のコレクションです。

CommandType を Text に設定する場合、疑問符プレースホルダを使用してパラメータを渡します。次に例を示します。

```
SELECT * FROM Customers WHERE ID = ?
```

SAParameter オブジェクトが SAParameterCollection に追加される順序は、コマンドテキストのパラメータの疑問符の位置にそのまま対応している必要があります。

コレクション内のパラメータが、実行されるクエリの要件と一致しない場合、エラーが発生したり例外がスローされることがあります。

参照

- SAParameterCollection クラス [SQL Anywhere .NET]289 ページ

Transaction プロパティ

SACommand が実行される SATransaction オブジェクトを指定します。

Visual Basic の構文

```
Public Shadows Property Transaction As SATransaction
```

C# の構文

```
public new SATransaction Transaction {get;set;}
```

備考

デフォルト値は NULL 参照です。Visual Basic では、これは Nothing です。

Transaction プロパティがすでに特定の値に設定されていて、コマンドが実行されている場合、このプロパティは設定できません。SACommand オブジェクトと同じ SAConnection オブジェクトに接続されていない SATransaction オブジェクトに対して Transaction プロパティを設定した場合、次に文を実行しようとすると例外がスローされます。

詳細については、「Transaction 処理」64 ページを参照してください。

参照

- SATransaction クラス [SQL Anywhere .NET]324 ページ

UpdatedRowSource プロパティ

SADataAdapter の Update メソッドによって使用されるときにコマンドの結果が DataRow に適用される方法を取得または設定します。

Visual Basic の構文

```
Public Overrides Property UpdatedRowSource As UpdateRowSource
```

C# の構文

```
public override UpdateRowSource UpdatedRowSource {get;set;}
```

備考

UpdatedRowSource 値の 1 つ。デフォルト値は UpdateRowSource.OutputParameters です。コマンドが自動的に生成される場合、このプロパティは UpdateRowSource.None です。

結果セットと出力パラメータの両方を返す UpdatedRowSource.Both は、サポートされていません。

SACommandBuilder クラス

DataSet の変更内容を関連するデータベース内のデータに一致させる単一テーブルの SQL 文を生成する方法です。

Visual Basic の構文

```
Public NotInheritable Class SACommandBuilder
    Inherits System.Data.Common.DbCommandBuilder
```

C# の構文

```
public sealed class SACommandBuilder :
    System.Data.Common.DbCommandBuilder
```

基本クラス

- System.Data.Common.DbCommandBuilder

メンバー

継承されたメンバーを含む SACommandBuilder クラスのすべてのメンバー。

名前	説明
SACommandBuilder コンストラクタ	SACommandBuilder オブジェクトを初期化します。
ApplyParameterInfo method (System.Data.Common.DbCommandBuilder から継承)	System.Data.Common.DbCommandBuilder クラスのプロバイダ実装が追加のパラメータプロパティを処理することを許可します。
DeriveParameters メソッド	指定された SACommand オブジェクトの Parameters コレクションに入力します。
Dispose method (System.Data.Common.DbCommandBuilder から継承)	System.Data.Common.DbCommandBuilder によって使用されるアンマネージリソースを解放するほか、必要に応じてマネージリソースを解放します。
GetDeleteCommand メソッド	SADataAdapter.Update が呼び出されたときにデータベース上で DELETE オペレーションを実行する、生成された SACommand オブジェクトを返します。
GetInsertCommand メソッド	Update が呼び出されたときにデータベース上で INSERT オペレーションを実行する、生成された SACommand オブジェクトを返します。
GetParameterName method (System.Data.Common.DbCommandBuilder から継承)	指定されたパラメータの名前を @p# のフォーマットで返します。
GetParameterPlaceholder method (System.Data.Common.DbCommandBuilder から継承)	関連する SQL 文のパラメータのプレースホルダを返します。
GetSchemaTable method (System.Data.Common.DbCommandBuilder から継承)	System.Data.Common.DbCommandBuilder のスキーマテーブルを返します。
GetUpdateCommand メソッド	Update が呼び出されたときにデータベース上で UPDATE オペレーションを実行する、生成された SACommand オブジェクトを返します。

名前	説明
InitializeCommand method (System.Data.Common.DbCommandBuilder から継承)	System.Data.Common.DbCommand の System.Data.Common.DbCommand.CommandTimeout、System.Data.Common.DbCommand.Transaction、System.Data.Common.DbCommand.CommandType、および System.Data.UpdateRowSource のプロパティをリセットします。
QuoteIdentifier メソッド	識別子に埋め込まれた引用符が適切にエスケープされた、引用符なし識別子の正しい引用付きの形式を返します。
RefreshSchema method (System.Data.Common.DbCommandBuilder から継承)	この System.Data.Common.DbCommandBuilder に関連付けられたコマンドをクリアします。
RowUpdatingHandler method (System.Data.Common.DbCommandBuilder から継承)	System.Data.OleDb.OleDbDataAdapter.RowUpdating のイベントのイベントハンドラを追加します。
SetRowUpdatingHandler method (System.Data.Common.DbCommandBuilder から継承)	System.Data.Common.DbDataAdapter の System.Data.OleDb.OleDbDataAdapter.RowUpdating イベントを処理するために、System.Data.Common.DbCommandBuilder を登録します。
UnquoteIdentifier メソッド	識別子に埋め込まれた引用符が適切にアンエスケープされた、引用符付き識別子の正しい引用符なしの形式を返します。
CatalogLocation property (System.Data.Common.DbCommandBuilder から継承)	System.Data.Common.DbCommandBuilder クラスのインスタンス用に System.Data.Common.CatalogLocation を設定または取得します。
CatalogSeparator property (System.Data.Common.DbCommandBuilder から継承)	System.Data.Common.DbCommandBuilder クラスのインスタンスのカタログセパレータとして使用する文字列を設定または取得します。
ConflictOption property (System.Data.Common.DbCommandBuilder から継承)	System.Data.Common.DbCommandBuilder によって使用される System.Data.ConflictOption を指定します。
DataAdapter プロパティ	文を生成する SADataAdapter を指定します。

名前	説明
QuotePrefix property (System.Data.Common.DbCommandBuilder から継承)	スペースや予約語などの文字列が名前に含まれているデータベースオブジェクト (テーブルやカラムなど) を指定するときに使用する先頭の文字または文字列を取得または設定します。
QuoteSuffix property (System.Data.Common.DbCommandBuilder から継承)	スペースや予約語などの文字列が名前に含まれているデータベースオブジェクト (テーブルやカラムなど) を指定するときに使用する末尾の文字または文字列を取得または設定します。
SchemaSeparator property (System.Data.Common.DbCommandBuilder から継承)	スキーマ識別子とその他の識別子とを区切るセパレータに使用する文字を取得または設定します。
SetAllValues property (System.Data.Common.DbCommandBuilder から継承)	UPDATE 文のすべてのカラム値が含まれるか、変更された値のみが含まれるかを指定します。

SACommandBuilder コンストラクタ

SACommandBuilder オブジェクトを初期化します。

オーバロードリスト

名前	説明
SACommandBuilder() コンストラクタ	SACommandBuilder オブジェクトを初期化します。
SACommandBuilder(SADataAdapter) コンストラクタ	SACommandBuilder オブジェクトを初期化します。

SACommandBuilder() コンストラクタ

SACommandBuilder オブジェクトを初期化します。

Visual Basic の構文

```
Public Sub New()
```

C# の構文

```
public SACommandBuilder()
```

SACommandBuilder(SADataAdapter) コンストラクタ

SACommandBuilder オブジェクトを初期化します。

Visual Basic の構文

```
Public Sub New(ByVal adapter As SADataAdapter)
```

C# の構文

```
public SACommandBuilder(SADataAdapter adapter)
```

パラメータ

- **adapter**　調整文を生成する SADataAdapter オブジェクト。

DeriveParameters メソッド

指定された SACommand オブジェクトの Parameters コレクションに入力します。

Visual Basic の構文

```
Public Shared Sub DeriveParameters(ByVal command As SACommand)
```

C# の構文

```
public static void DeriveParameters(SACommand command)
```

パラメータ

- **command**　パラメータを抽出する SACommand オブジェクト。

備考

これは、SACommand に指定されたストアドプロシージャに対して使用されます。

DeriveParameters は、SACommand の既存のパラメータ情報を上書きします。

DeriveParameters には、データベースサーバへの追加呼び出しが必要です。パラメータ情報が事前に分かっている場合、情報を明示的に設定して Parameters コレクションに入力する方が効率的です。

GetDeleteCommand メソッド

SADataAdapter.Update が呼び出されたときにデータベース上で DELETE オペレーションを実行する、生成された SACommand オブジェクトを返します。

オーバロードリスト

名前	説明
GetDeleteCommand() メソッド	SADataAdapter.Update が呼び出されたときにデータベース上で DELETE オペレーションを実行する、生成された SACommand オブジェクトを返します。
GetDeleteCommand(bool) メソッド	SADataAdapter.Update が呼び出されたときにデータベース上で DELETE オペレーションを実行する、生成された SACommand オブジェクトを返します。

GetDeleteCommand() メソッド

SADataAdapter.Update が呼び出されたときにデータベース上で DELETE オペレーションを実行する、生成された SACommand オブジェクトを返します。

Visual Basic の構文

```
Public Shadows Function GetDeleteCommand() As SACommand
```

C# の構文

```
public new SACommand GetDeleteCommand()
```

戻り値

削除の実行に必要な、自動的に生成された SACommand オブジェクト。

備考

GetDeleteCommand メソッドは、実行対象の SACommand オブジェクトを返すため、情報やトラブルシューティング用として役に立ちます。

また、GetDeleteCommand は、修正されたコマンドの基礎としても使用できます。たとえば、GetDeleteCommand を呼び出して CommandTimeout 値を修正してから、SADataAdapter で値を明示的に設定できます。

アプリケーションが Update または GetDeleteCommand を呼び出すと、SQL 文が最初に生成されます。SQL 文が最初に生成された後で、アプリケーションが文を変更する場合、RefreshSchema を明示的に呼び出す必要があります。この処理を行わないと、GetDeleteCommand は古い文の情報を使用し続けます。

参照

- System.Data.Common.DbCommandBuilder.RefreshSchema

GetDeleteCommand(bool) メソッド

SADataAdapter.Update が呼び出されたときにデータベース上で DELETE オペレーションを実行する、生成された SACommand オブジェクトを返します。

Visual Basic の構文

```
Public Shadows Function GetDeleteCommand(
    ByVal useColumnsForParameterNames As Boolean
) As SACommand
```

C# の構文

```
public new SACommand GetDeleteCommand(bool useColumnsForParameterNames)
```

パラメータ

- **useColumnsForParameterNames** true の場合、可能であれば、カラム名に一致するパラメータ名を生成します。false の場合、@p1、@p2 などを生成します。

戻り値

削除の実行に必要な、自動的に生成された SACommand オブジェクト。

備考

GetDeleteCommand メソッドは、実行対象の SACommand オブジェクトを返すため、情報やトラブルシューティング用として役に立ちます。

また、GetDeleteCommand は、修正されたコマンドの基礎としても使用できます。たとえば、GetDeleteCommand を呼び出して CommandTimeout 値を修正してから、SADataAdapter で値を明示的に設定できます。

アプリケーションが Update または GetDeleteCommand を呼び出すと、SQL 文が最初に生成されます。SQL 文が最初に生成された後で、アプリケーションが文を変更する場合、RefreshSchema を明示的に呼び出す必要があります。この処理を行わないと、GetDeleteCommand は古い文の情報を使用し続けます。

参照

- System.Data.Common.DbCommandBuilder.RefreshSchema

GetInsertCommand メソッド

Update が呼び出されたときにデータベース上で INSERT オペレーションを実行する、生成された SACommand オブジェクトを返します。

オーバロードリスト

名前	説明
GetInsertCommand() メソッド	Update が呼び出されたときにデータベース上で INSERT オペレーションを実行する、生成された SACommand オブジェクトを返します。
GetInsertCommand(bool) メソッド	Update が呼び出されたときにデータベース上で INSERT オペレーションを実行する、生成された SACommand オブジェクトを返します。

GetInsertCommand() メソッド

Update が呼び出されたときにデータベース上で INSERT オペレーションを実行する、生成された SACommand オブジェクトを返します。

Visual Basic の構文

```
Public Shadows Function GetInsertCommand() As SACommand
```

C# の構文

```
public new SACommand GetInsertCommand()
```

戻り値

挿入の実行に必要な、自動的に生成された SACommand オブジェクト。

備考

GetInsertCommand メソッドは、実行対象の SACommand オブジェクトを返すため、情報やトラブルシューティング用として役に立ちます。

また、GetInsertCommand は、修正されたコマンドの基礎としても使用できます。たとえば、GetInsertCommand を呼び出して CommandTimeout 値を修正してから、SADataAdapter で値を明示的に設定できます。

アプリケーションが Update または GetInsertCommand を呼び出すと、SQL 文が最初に生成されます。SQL 文が最初に生成された後で、アプリケーションが文を変更する場合、RefreshSchema を明示的に呼び出す必要があります。この処理を行わないと、GetInsertCommand は、正しくない可能性がある古い文の情報を使用し続けます。

参照

- SACommandBuilder.GetDeleteCommand メソッド [SQL Anywhere .NET]156 ページ

GetInsertCommand(bool) メソッド

Update が呼び出されたときにデータベース上で INSERT オペレーションを実行する、生成された SACommand オブジェクトを返します。

Visual Basic の構文

```
Public Shadows Function GetInsertCommand(
    ByVal useColumnsForParameterNames As Boolean
) As SACommand
```

C# の構文

```
public new SACommand GetInsertCommand(bool useColumnsForParameterNames)
```

パラメータ

- **useColumnsForParameterNames** true の場合、可能であれば、カラム名に一致するパラメータ名を生成します。false の場合、@p1、@p2 などを生成します。

戻り値

挿入の実行に必要な、自動的に生成された SACommand オブジェクト。

備考

GetInsertCommand メソッドは、実行対象の SACommand オブジェクトを返すため、情報やトラブルシューティング用として役に立ちます。

また、GetInsertCommand は、修正されたコマンドの基礎としても使用できます。たとえば、GetInsertCommand を呼び出して CommandTimeout 値を修正してから、SADataAdapter で値を明示的に設定できます。

アプリケーションが Update または GetInsertCommand を呼び出すと、SQL 文が最初に生成されます。SQL 文が最初に生成された後で、アプリケーションが文を変更する場合、RefreshSchema を明示的に呼び出す必要があります。この処理を行わないと、GetInsertCommand は、正しくない可能性がある古い文の情報を使用し続けます。

参照

- SACommandBuilder.GetDeleteCommand メソッド [SQL Anywhere .NET]156 ページ

GetUpdateCommand メソッド

Update が呼び出されたときにデータベース上で UPDATE オペレーションを実行する、生成された SACommand オブジェクトを返します。

オーバロードリスト

名前	説明
GetUpdateCommand() メソッド	Update が呼び出されたときにデータベース上で UPDATE オペレーションを実行する、生成された SACommand オブジェクトを返します。
GetUpdateCommand(bool) メソッド	Update が呼び出されたときにデータベース上で UPDATE オペレーションを実行する、生成された SACommand オブジェクトを返します。

GetUpdateCommand() メソッド

Update が呼び出されたときにデータベース上で UPDATE オペレーションを実行する、生成された SACommand オブジェクトを返します。

Visual Basic の構文

```
Public Shadows Function GetUpdateCommand() As SACommand
```

C# の構文

```
public new SACommand GetUpdateCommand()
```

戻り値

更新の実行に必要な、自動的に生成された SACommand オブジェクト。

備考

GetUpdateCommand メソッドは、実行対象の SACommand オブジェクトを返すため、情報やトラブルシューティング用として役に立ちます。

また、GetUpdateCommand は、修正されたコマンドの基礎としても使用できます。たとえば、GetUpdateCommand を呼び出して CommandTimeout 値を修正してから、SADataAdapter で値を明示的に設定できます。

アプリケーションが Update または GetUpdateCommand を呼び出すと、SQL 文が最初に生成されます。SQL 文が最初に生成された後で、アプリケーションが文を変更する場合、RefreshSchema を明示的に呼び出す必要があります。この処理を行わないと、GetUpdateCommand は、正しくない可能性がある古い文の情報を使用し続けます。

参照

- System.Data.Common.DbCommandBuilder.RefreshSchema

GetUpdateCommand(bool) メソッド

Update が呼び出されたときにデータベース上で UPDATE オペレーションを実行する、生成された SACommand オブジェクトを返します。

Visual Basic の構文

```
Public Shadows Function GetUpdateCommand(
    ByVal useColumnsForParameterNames As Boolean
) As SACommand
```

C# の構文

```
public new SACommand GetUpdateCommand(bool useColumnsForParameterNames)
```

パラメータ

- **useColumnsForParameterNames** true の場合、可能であれば、カラム名に一致するパラメータ名を生成します。false の場合、@p1、@p2 などを生成します。

戻り値

更新の実行に必要な、自動的に生成された SACommand オブジェクト。

備考

GetUpdateCommand メソッドは、実行対象の SACommand オブジェクトを返すため、情報やトラブルシューティング用として役に立ちます。

また、GetUpdateCommand は、修正されたコマンドの基礎としても使用できます。たとえば、GetUpdateCommand を呼び出して CommandTimeout 値を修正してから、SADataAdapter で値を明示的に設定できます。

アプリケーションが Update または GetUpdateCommand を呼び出すと、SQL 文が最初に生成されます。SQL 文が最初に生成された後で、アプリケーションが文を変更する場合、RefreshSchema を明示的に呼び出す必要があります。この処理を行わないと、GetUpdateCommand は、正しくない可能性がある古い文の情報を使用し続けます。

参照

- System.Data.Common.DbCommandBuilder.RefreshSchema

QuoteIdentifier メソッド

識別子に埋め込まれた引用符が適切にエスケープされた、引用符なし識別子の正しい引用付きの形式を返します。

Visual Basic の構文

```
Public Overrides Function QuoteIdentifier(
    ByVal unquotedIdentifier As String
) As String
```

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

C# の構文

```
public override string QuoteIdentifier(string unquotedIdentifier)
```

パラメータ

- **unquotedIdentifier**　引用符を付ける必要のある引用符なしの識別子を表す文字列。

戻り値

埋め込まれた引用符が適切にエスケープされた、引用符なしの識別子の引用符付きの形式を表す文字列を返します。

UnquoteIdentifier メソッド

識別子に埋め込まれた引用符が適切にアンエスケープされた、引用符付き識別子の正しい引用符なしの形式を返します。

Visual Basic の構文

```
Public Overrides Function UnquoteIdentifier(
    ByVal quotedIdentifier As String
) As String
```

C# の構文

```
public override string UnquoteIdentifier(string quotedIdentifier)
```

パラメータ

- **quotedIdentifier**　埋め込まれた引用符が削除される、引用符付きの識別子を表す文字列。

戻り値

埋め込まれた引用符が適切にアンエスケープされた、引用符付き識別子の引用符なしの形式を表す文字列を返します。

DataAdapter プロパティ

文を生成する SADataAdapter を指定します。

Visual Basic の構文

```
Public Shadows Property DataAdapter As SADataAdapter
```

C# の構文

```
public new SADataAdapter DataAdapter {get;set;}
```

備考

SADataAdapter オブジェクト。

SACommandBuilder の新しいインスタンスを作成すると、この SADataAdapter に関連付けられている既存の SACommandBuilder が解放されます。

SACommLinksOptionsBuilder クラス

SAConnection クラスが使用する接続文字列の、CommLinks オプション部分を作成および管理する単純な方法を提供します。

Visual Basic の構文

```
Public NotInheritable Class SACommLinksOptionsBuilder
```

C# の構文

```
public sealed class SACommLinksOptionsBuilder
```

メンバー

継承されたメンバーを含む SACommLinksOptionsBuilder クラスのすべてのメンバー。

名前	説明
SACommLinksOptionsBuilder コンストラクタ	SACommLinksOptionsBuilder オブジェクトを初期化します。
GetUseLongNameAsKeyword メソッド	長い接続パラメータ名を接続文字列で使用するかどうかを示す boolean 値を取得します。
SetUseLongNameAsKeyword メソッド	長い接続パラメータ名を接続文字列で使用するかどうかを示す boolean 値を設定します。
ToString メソッド	SACommLinksOptionsBuilder オブジェクトを文字列表現に変換します。
All プロパティ	ALL CommLinks オプションを取得または設定します。
ConnectionString プロパティ	構築される接続文字列を取得または設定します。
SharedMemory プロパティ	SharedMemory プロトコルを取得または設定します。
TcpOptionsBuilder プロパティ	TCP オプション文字列の作成に使用する TcpOptionsBuilder オブジェクトを取得または設定します。
TcpOptionsString プロパティ	TCP オプションの文字列を取得または設定します。

備考

SACommLinksOptionsBuilder クラスは、.NET Compact Framework 2.0 では使用できません。

接続パラメータのリストについては、「接続パラメータ」『SQL Anywhere サーバ データベース管理』を参照してください。

SACommLinksOptionsBuilder コンストラクタ

SACommLinksOptionsBuilder オブジェクトを初期化します。

オーバロードリスト

名前	説明
SACommLinksOptionsBuilder() コンストラクタ	SACommLinksOptionsBuilder オブジェクトを初期化します。
SACommLinksOptionsBuilder(string) コンストラクタ	SACommLinksOptionsBuilder オブジェクトを初期化します。

SACommLinksOptionsBuilder() コンストラクタ

SACommLinksOptionsBuilder オブジェクトを初期化します。

Visual Basic の構文

```
Public Sub New()
```

C# の構文

```
public SACommLinksOptionsBuilder()
```

備考

SACommLinksOptionsBuilder クラスは、.NET Compact Framework 2.0 では使用できません。

例

次の文は、SACommLinksOptionsBuilder オブジェクトを初期化します。

```
SACommLinksOptionsBuilder commLinks = new SACommLinksOptionsBuilder( );
```

SACommLinksOptionsBuilder(string) コンストラクタ

SACommLinksOptionsBuilder オブジェクトを初期化します。

Visual Basic の構文

```
Public Sub New(ByVal options As String)
```

C# の構文

```
public SACommLinksOptionsBuilder(string options)
```

パラメータ

- **options**　SQL Anywhere CommLinks 接続パラメータ文字列。接続パラメータのリストについては、「接続パラメータ」『SQL Anywhere サーバ データベース管理』を参照してください。

備考

SACommLinksOptionsBuilder クラスは、.NET Compact Framework 2.0 では使用できません。

例

次の文は、SACommLinksOptionsBuilder オブジェクトを初期化します。

```
SACommLinksOptionsBuilder commLinks =
  new SACommLinksOptionsBuilder("TCPIP(DoBroadcast=ALL;Timeout=20)");
```

GetUseLongNameAsKeyword メソッド

長い接続パラメータ名を接続文字列で使用するかどうかを示す boolean 値を取得します。

Visual Basic の構文

```
Public Function GetUseLongNameAsKeyword() As Boolean
```

C# の構文

```
public bool GetUseLongNameAsKeyword()
```

戻り値

長い接続パラメータ名を使用して接続文字列を作成する場合は true、それ以外の場合は false。

備考

SQL Anywhere 接続パラメータの名前には、長い名前と短い名前の 2 種類があります。たとえば、接続文字列に ODBC データソースの名前を指定する場合は、DataSourceName と DSN のいずれかを使用できます。DataSourceName または DSN。デフォルトでは、長い接続パラメータ名を使用して接続文字列を作成します。

参照

- SACommLinksOptionsBuilder.SetUseLongNameAsKeyword メソッド [SQL Anywhere .NET]166 ページ

SetUseLongNameAsKeyword メソッド

長い接続パラメータ名を接続文字列で使用するかどうかを示す boolean 値を設定します。

Visual Basic の構文

```
Public Sub SetUseLongNameAsKeyword(
    ByVal useLongNameAsKeyword As Boolean
)
```

 Copyright © 2013, SAP AG or an SAP affiliate company.

C# の構文

```
public void SetUseLongNameAsKeyword(bool useLongNameAsKeyword)
```

パラメータ

- **useLongNameAsKeyword**　長い接続パラメータ名を接続文字列で使用するかどうかを示す boolean 値。

備考

デフォルトでは、長い接続パラメータ名が使用されます。

参照

- SACommLinksOptionsBuilder.GetUseLongNameAsKeyword メソッド [SQL Anywhere .NET]166 ページ

ToString メソッド

SACommLinksOptionsBuilder オブジェクトを文字列表現に変換します。

Visual Basic の構文

```
Public Overrides Function ToString() As String
```

C# の構文

```
public override string ToString()
```

戻り値

構築されるオプション文字列。

All プロパティ

ALL CommLinks オプションを取得または設定します。

Visual Basic の構文

```
Public Property All As Boolean
```

C# の構文

```
public bool All {get;set;}
```

備考

最初に共有メモリプロトコルを使用して接続を試行し、次に使用可能なすべての通信プロトコルを使用します。使用する通信プロトコルが不明の場合は、この設定を使用してください。

SACommLinksOptionsBuilder クラスは、.NET Compact Framework 2.0 では使用できません。

ConnectionString プロパティ

構築される接続文字列を取得または設定します。

Visual Basic の構文

```
Public Property ConnectionString As String
```

C# の構文

```
public string ConnectionString {get;set;}
```

備考

SACommLinksOptionsBuilder クラスは、.NET Compact Framework 2.0 では使用できません。

SharedMemory プロパティ

SharedMemory プロトコルを取得または設定します。

Visual Basic の構文

```
Public Property SharedMemory As Boolean
```

C# の構文

```
public bool SharedMemory {get;set;}
```

備考

SACommLinksOptionsBuilder クラスは、.NET Compact Framework 2.0 では使用できません。

TcpOptionsBuilder プロパティ

TCP オプション文字列の作成に使用する TcpOptionsBuilder オブジェクトを取得または設定します。

Visual Basic の構文

```
Public Property TcpOptionsBuilder As SATcpOptionsBuilder
```

C# の構文

```
public SATcpOptionsBuilder TcpOptionsBuilder {get;set;}
```

TcpOptionsString プロパティ

TCP オプションの文字列を取得または設定します。

Visual Basic の構文

```
Public Property TcpOptionsString As String
```

C# の構文

```
public string TcpOptionsString {get;set;}
```

SAConnection クラス

SQL Anywhere データベースへの接続を表します。

Visual Basic の構文

```
Public NotInheritable Class SAConnection
    Inherits System.Data.Common.DbConnection
```

C# の構文

```
public sealed class SAConnection : System.Data.Common.DbConnection
```

基本クラス

- System.Data.Common.DbConnection

メンバー

継承されたメンバーを含む SAConnection クラスのすべてのメンバー。

名前	説明
SAConnection コンストラクタ	SAConnection オブジェクトを初期化します。
BeginDbTransaction method (System.Data.Common.DbConnection から継承)	データベーストランザクションを起動します。
BeginTransaction メソッド	トランザクションオブジェクトを返します。
ChangeDatabase メソッド	開いている SAConnection の現在のデータベースを変更します。
ChangePassword メソッド	接続文字列に示されるユーザのパスワードを、指定される新しいパスワードに変更します。
ClearAllPools メソッド	すべての接続プールを空にします。
ClearPool メソッド	指定の接続に関連付けられている接続プールを空にします。
Close メソッド	データベース接続を閉じます。
CreateCommand メソッド	SACommand オブジェクトを初期化します。
EnlistDistributedTransaction メソッド	指定されたトランザクションに分散トランザクションとしてエンリストします。

名前	説明
EnlistTransaction メソッド	指定されたトランザクションに分散トランザクションとしてエンリストします。
GetSchema メソッド	サポートされているスキーマコレクションのリストを返します。
OnStateChange method (System.Data.Common.DbConnection から継承)	System.Data.Common.DbConnection.StateChange イベントを発生させます。
Open メソッド	SAConnection.ConnectionString で指定されたプロパティ設定を使用してデータベース接続を開きます。
OpenAsync method (System.Data.Common.DbConnection から継承)	データベース接続を開き、System.Data.Common.DbConnection.ConnectionString で指定された設定を持つ System.Data.Common.DbConnection.Open の非同期バージョン
ConnectionString プロパティ	データベース接続文字列を指定します。
ConnectionTimeout プロパティ	接続試行がエラーでタイムアウトするまでの秒数を取得します。
Database プロパティ	現在のデータベースの名前を取得します。
DataSource プロパティ	データベースサーバの名前を取得します。
DbProviderFactory property (System.Data.Common.DbConnection から継承)	この System.Data.Common.DbConnection の System.Data.Common.DbProviderFactory を取得します。
InitString プロパティ	接続の確立直後に実行するコマンドです。
ServerVersion プロパティ	クライアントが接続する SQL Anywhere のインスタンスのバージョンが含まれている文字列を取得します。
State プロパティ	SAConnection オブジェクトのステータスを示します。
InfoMessage イベント	SQL Anywhere データベースサーバが警告または情報メッセージを返すときに発生します。
StateChange イベント	SAConnection オブジェクトのステータスが変わると発生します。

備考

接続パラメータのリストについては、「接続パラメータ」『SQL Anywhere サーバ データベース管理』を参照してください。

SAConnection コンストラクタ

SAConnection オブジェクトを初期化します。

オーバロードリスト

名前	説明
SAConnection() コンストラクタ	SAConnection オブジェクトを初期化します。
SAConnection(string) コンストラクタ	SAConnection オブジェクトを初期化します。

SAConnection() コンストラクタ

SAConnection オブジェクトを初期化します。

Visual Basic の構文

```
Public Sub New()
```

C# の構文

```
public SAConnection()
```

備考

接続を開いてから、データベース操作を実行してください。

SAConnection(string) コンストラクタ

SAConnection オブジェクトを初期化します。

Visual Basic の構文

```
Public Sub New(ByVal connectionString As String)
```

C# の構文

```
public SAConnection(string connectionString)
```

パラメータ

- **connectionString**　SQL Anywhere 接続文字列。接続文字列は keyword=value のペアがセミコロンで区切られたリストです。接続パラメータのリストについては、「接続パラメータ」『SQL Anywhere サーバ データベース管理』を参照してください。

備考

接続を開いてから、データベース操作を実行してください。

参照

- SAConnection クラス [SQL Anywhere .NET]169 ページ

例

次の文は、hr という名前の SQL Anywhere データベースサーバ上で動作している policies という名前のデータベースへの接続について SAConnection オブジェクトを初期化します。接続では、ユーザ ID admin とパスワード money を使用します。

```
SAConnection conn = new SAConnection(
   "UID=admin;PWD=money;SERVER=hr;DBN=policies" );
conn.Open();
```

BeginTransaction メソッド

トランザクションオブジェクトを返します。

オーバロードリスト

名前	説明
BeginTransaction() メソッド	トランザクションオブジェクトを返します。
BeginTransaction(IsolationLevel) メソッド	トランザクションオブジェクトを返します。
BeginTransaction(SAIsolationLevel) メソッド	トランザクションオブジェクトを返します。

BeginTransaction() メソッド

トランザクションオブジェクトを返します。

Visual Basic の構文

```
Public Shadows Function BeginTransaction() As SATransaction
```

C# の構文

```
public new SATransaction BeginTransaction()
```

戻り値

新しいトランザクションを表す SATransaction オブジェクト。

備考

トランザクションオブジェクトに関連付けられているコマンドは、単一のトランザクションとして実行されます。トランザクションは、Commit メソッドまたは Rollback メソッドへの呼び出しで終了します。

コマンドをトランザクションオブジェクトに関連付けるには、SACommand.Transaction プロパティを使用します。

参照

- SATransaction クラス [SQL Anywhere .NET]324 ページ
- SACommand.Transaction プロパティ [SQL Anywhere .NET]151 ページ

BeginTransaction(IsolationLevel) メソッド

トランザクションオブジェクトを返します。

Visual Basic の構文

```
Public Shadows Function BeginTransaction(
    ByVal isolationLevel As IsolationLevel
) As SATransaction
```

C# の構文

```
public new SATransaction BeginTransaction(IsolationLevel isolationLevel)
```

パラメータ

- **isolationLevel**　SAIsolationLevel 列挙のメンバー。デフォルト値は ReadCommitted です。

戻り値

新しいトランザクションを表す SATransaction オブジェクト。

備考

トランザクションオブジェクトに関連付けられているコマンドは、単一のトランザクションとして実行されます。トランザクションは、Commit メソッドまたは Rollback メソッドへの呼び出しで終了します。

コマンドをトランザクションオブジェクトに関連付けるには、SACommand.Transaction プロパティを使用します。

参照

- SATransaction クラス [SQL Anywhere .NET]324 ページ
- SACommand.Transaction プロパティ [SQL Anywhere .NET]151 ページ
- SAIsolationLevel 列挙体 [SQL Anywhere .NET]338 ページ

例

```
SATransaction tx =
  conn.BeginTransaction( SAIsolationLevel.ReadUncommitted );
```

BeginTransaction(SAIsolationLevel) メソッド

トランザクションオブジェクトを返します。

Visual Basic の構文

```
Public Function BeginTransaction(
    ByVal isolationLevel As SAIsolationLevel
) As SATransaction
```

C# の構文

```
public SATransaction BeginTransaction(SAIsolationLevel isolationLevel)
```

パラメータ

- **isolationLevel** SAIsolationLevel 列挙のメンバー。デフォルト値は ReadCommitted です。

戻り値

新しいトランザクションを表す SATransaction オブジェクト。

備考

トランザクションオブジェクトに関連付けられているコマンドは、単一のトランザクションとして実行されます。トランザクションは、Commit メソッドまたは Rollback メソッドへの呼び出しで終了します。

コマンドをトランザクションオブジェクトに関連付けるには、SACommand.Transaction プロパティを使用します。

詳細については、「Transaction 処理」64 ページを参照してください。

詳細については、「典型的な矛盾のケース」『SQL Anywhere サーバ SQL の使用法』を参照してください。

参照

- SATransaction クラス [SQL Anywhere .NET]324 ページ
- SACommand.Transaction プロパティ [SQL Anywhere .NET]151 ページ
- SAIsolationLevel 列挙体 [SQL Anywhere .NET]338 ページ
- SATransaction.Commit メソッド [SQL Anywhere .NET]325 ページ
- SATransaction.Rollback メソッド [SQL Anywhere .NET]326 ページ

ChangeDatabase メソッド

開いている SAConnection の現在のデータベースを変更します。

Visual Basic の構文

```
Public Overrides Sub ChangeDatabase(ByVal database As String)
```

C# の構文

```
public override void ChangeDatabase(string database)
```

パラメータ

- **database** 現在のデータベースの代わりに使用するデータベースの名前。

ChangePassword メソッド

接続文字列に示されるユーザのパスワードを、指定される新しいパスワードに変更します。

Visual Basic の構文

```
Public Shared Sub ChangePassword(
    ByVal connectionString As String,
    ByVal newPassword As String
)
```

C# の構文

```
public static void ChangePassword(
    string connectionString,
    string newPassword
)
```

パラメータ

- **connectionString**　目的のデータベースサーバに接続できるだけの情報が含まれる接続文字列。接続文字列には、ユーザ ID と現在のパスワードが含まれる可能性があります。

- **newPassword**　設定する新しいパスワード。このパスワードは、最小文字数や特殊文字の要件など、サーバ上で設定されているパスワードセキュリティポリシーに準拠する必要があります。

例外

- **ArgumentNullException**　connectionString または newPassword parameter のいずれかが null です。

- **ArgumentException**　接続文字列に、統合化セキュリティを使用するオプションが含まれています。

ClearAllPools メソッド

すべての接続プールを空にします。

Visual Basic の構文

```
Public Shared Sub ClearAllPools()
```

C# の構文

```
public static void ClearAllPools()
```

ClearPool メソッド

指定の接続に関連付けられている接続プールを空にします。

Visual Basic の構文

```
Public Shared Sub ClearPool(ByVal connection As SAConnection)
```

C# の構文

```
public static void ClearPool(SAConnection connection)
```

パラメータ

- **connection**　プールからクリアする SAConnection オブジェクト。

参照

- SAConnection クラス [SQL Anywhere .NET]169 ページ

Close メソッド

データベース接続を閉じます。

Visual Basic の構文

```
Public Overrides Sub Close()
```

C# の構文

```
public override void Close()
```

備考

Close メソッドは、保留中のトランザクションをロールバックします。次に、接続プールへの接続を解放します。また、接続プーリングが無効の場合は、接続を閉じます。StateChange イベントの処理中に Close が呼び出されても、追加の StateChange イベントは実行されません。アプリケーションは、Close を複数回呼び出すことができます。

CreateCommand メソッド

SACommand オブジェクトを初期化します。

Visual Basic の構文

```
Public Shadows Function CreateCommand() As SACommand
```

C# の構文

```
public new SACommand CreateCommand()
```

戻り値

SACommand オブジェクト。

備考

コマンドオブジェクトは SAConnection オブジェクトに関連付けられます。

参照

- SACommand クラス [SQL Anywhere .NET]129 ページ
- SAConnection クラス [SQL Anywhere .NET]169 ページ

EnlistDistributedTransaction メソッド

指定されたトランザクションに分散トランザクションとしてエンリストします。

Visual Basic の構文

```
Public Sub EnlistDistributedTransaction(
    ByVal transaction As System.EnterpriseServices.ITransaction
)
```

C# の構文

```
public void EnlistDistributedTransaction(
    System.EnterpriseServices.ITransaction transaction
)
```

パラメータ

- **transaction**　エンリストする既存の System.EnterpriseServices.ITransaction への参照。

EnlistTransaction メソッド

指定されたトランザクションに分散トランザクションとしてエンリストします。

Visual Basic の構文

```
Public Overrides Sub EnlistTransaction(
    ByVal transaction As System.Transactions.Transaction
)
```

C# の構文

```
public override void EnlistTransaction(
    System.Transactions.Transaction transaction
)
```

パラメータ

- **transaction**　エンリストする既存の System.Transactions.Transaction への参照。

GetSchema メソッド

サポートされているスキーマコレクションのリストを返します。

オーバロードリスト

名前	説明
GetSchema() メソッド	サポートされているスキーマコレクションのリストを返します。

名前	説明
GetSchema(string) メソッド	この SAConnection オブジェクトについて指定されたメタデータコレクションに関する情報を返します。
GetSchema(string, string[]) メソッド	この SAConnection オブジェクトのデータソースのスキーマ情報を返します。このとき、文字列が指定されている場合はスキーマ名として使用し、文字列配列が指定されている場合は制限値として使用します。

GetSchema() メソッド

サポートされているスキーマコレクションのリストを返します。

Visual Basic の構文

```
Public Overrides Function GetSchema() As DataTable
```

C# の構文

```
public override DataTable GetSchema()
```

備考

使用可能なメタデータの詳細については、GetSchema(string,string[]) を参照してください。

GetSchema(string) メソッド

この SAConnection オブジェクトについて指定されたメタデータコレクションに関する情報を返します。

Visual Basic の構文

```
Public Overrides Function GetSchema(
    ByVal collection As String
) As DataTable
```

C# の構文

```
public override DataTable GetSchema(string collection)
```

パラメータ

- **collection**　メタデータコレクションの名前。名前が提供されない場合は、MetaDataCollections が使用されます。

備考

使用可能なメタデータの詳細については、GetSchema(string,string[]) を参照してください。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

参照

- SAConnection クラス [SQL Anywhere .NET]169 ページ

GetSchema(string, string[]) メソッド

この SAConnection オブジェクトのデータソースのスキーマ情報を返します。このとき、文字列が指定されている場合はスキーマ名として使用し、文字列配列が指定されている場合は制限値として使用します。

Visual Basic の構文

```
Public Overrides Function GetSchema(
    ByVal collection As String,
    ByVal restrictions As String()
) As DataTable
```

C# の構文

```
public override DataTable GetSchema(
    string collection,
    string[] restrictions
)
```

戻り値

スキーマ情報が格納されている DataTable。

備考

これらのメソッドを使用すると、データベースサーバに各種のメタデータを問い合わせることができます。メタデータの各型にはコレクション名が指定されており、そのデータを受け取るにはコレクション名を渡す必要があります。デフォルトのコレクション名は MetaDataCollections です。

引数を指定せずに、または **MetaDataCollections** というスキーマコレクション名を指定して GetSchema メソッドを呼び出すことによって、SQL Anywhere .NET データプロバイダに問い合わせをして、サポートされているスキーマコレクションのリストを判断できます。これによって、サポートされているスキーマコレクション (CollectionName)、それぞれがサポートする制限の数 (NumberOfRestrictions)、使用する識別子部分の数のリストから成る DataTable が返されます。

コレクション	メタデータ
Columns	データベース内のすべてのカラムに関する情報を返します。
DataSourceInformation	データベースサーバに関する情報を返します。
DataType	サポートされているデータ型のリストを返します。

コレクション	メタデータ
ForeignKeys	データベース内のすべての外部キーに関する情報を返します。
IndexColumns	データベース内のすべてのインデックスカラムに関する情報を返します。
Indexes	データベース内のすべてのインデックスに関する情報を返します。
MetaDataCollections	すべてのコレクション名のリストを返します。
ProcedureParameters	データベース内のすべてのプロシージャパラメータに関する情報を返します。
Procedures	データベース内のすべてのプロシージャに関する情報を返します。
ReservedWords	SQL Anywhere が使用する予約語のリストを返します。
Restrictions	GetSchema で使用される制限に関する情報を返します。
Tables	データベース内のすべてのテーブルに関する情報を返します。
UserDefinedTypes	データベース内のすべてのユーザ定義データ型に関する情報を返します。
Users	データベース内のすべてのユーザに関する情報を返します。
ViewColumns	データベース内のすべてのビュー内のカラムに関する情報を返します。
Views	データベース内のすべてのビューに関する情報を返します。

これらのコレクション名は、SAMetaDataCollectionNames クラスの読み込み専用プロパティとしても使用できます。

返される結果は、GetSchema への呼び出しの中で制限の配列を指定することによってフィルタできます。

各コレクションで有効な制限は、次の文を呼び出すことで問い合わせることができます。

```
GetSchema( "Restrictions" )
```

コレクションが 4 つの制限を必要とする場合、制限パラメータは、NULL、または 4 つの値から成る文字列です。

特定の制限をフィルタするには、フィルタする文字列を配列内の所定の位置に指定し、使用しない位置には NULL を指定します。たとえば、Tables コレクションには 3 つの制限があります。Owner、Table、TableType です。

Table コレクションを table_name でフィルタするには、次の手順に従います。

```
GetSchema( "Tables", new string[ ] { NULL, "my_table", NULL } )
```

my_table という名前のすべてのテーブルに関する情報を返します。

```
GetSchema( "Tables", new string[ ] { "DBA", "my_table", NULL } )
```

DBA ユーザが所有する my_table という名前のすべてのテーブルに関する情報を返します。

次に、各コレクションが返すカラムの概要を示します。カラムに制限を指定することで、コレクションが返すローの数を減らすことができる場合は、そのカラムの制限名をカッコ内に表示します。制限は、以下のリストで表示される順序で指定されます。

Columns コレクション

- table_schema (Owner)
- table_name (Table)
- column_name (Column)
- ordinal_position
- column_default
- is_nullable
- data_type
- precision
- scale
- column_size

DataSourceInformation コレクション

- CompositeIdentifierSeparatorPattern
- DataSourceProductName
- DataSourceProductVersion
- DataSourceProductVersionNormalized

- GroupByBehavior
- IdentifierPattern
- IdentifierCase
- OrderByColumnsInSelect
- ParameterMarkerFormat
- ParameterMarkerPattern
- ParameterNameMaxLength
- ParameterNamePattern
- QuotedIdentifierPattern
- QuotedIdentifierCase
- StatementSeparatorPattern
- StringLiteralPattern
- SupportedJoinOperators

DataTypes コレクション

- TypeName
- ProviderDbType
- ColumnSize
- CreateFormat
- CreateParameters
- DataType
- IsAutoIncrementable
- IsBestMatch
- IsCaseSensitive
- IsFixedLength
- IsFixedPrecisionScale
- IsLong

- IsNullable
- IsSearchable
- IsSearchableWithLike
- IsUnsigned
- MaximumScale
- MinimumScale
- IsConcurrencyType
- IsLiteralSupported
- LiteralPrefix
- LiteralSuffix

ForeignKeys コレクション

- table_schema (Owner)
- table_name (Table)
- column_name (Column)

IndexColumns コレクション

- table_schema (Owner)
- table_name (Table)
- index_name (Name)
- column_name (Column)
- order

Indexes コレクション

- table_schema (Owner)
- table_name (Table)
- index_name (Name)
- primary_key
- is_unique

MetaDataCollections コレクション

- CollectionName
- NumberOfRestrictions
- NumberOfIdentifierParts

ProcedureParameters コレクション

- procedure_schema (Owner)
- procedure_name (Name)
- parameter_name (Parameter)
- data_type
- parameter_type
- is_input
- is_output

Procedures コレクション

- procedure_schema (Owner)
- procedure_name (Name)

ReservedWords コレクション

- reserved_word

Restrictions コレクション

- CollectionName
- RestrictionName
- RestrictionDefault
- RestrictionNumber

Tables コレクション

- table_schema (Owner)
- table_name (Table)
- table_type (TableType)。"BASE"、"VIEW"、"MAT VIEW"、"LCL TEMP"、"GBL TEMP"、"TEXT"、"TEXT GBL TEMP" のいずれか。

UserDefinedTypes コレクション

- data_type
- default
- precision
- scale

Users コレクション

- user_name (UserName)
- resource_auth
- database_auth
- schedule_auth
- user_group

ViewColumns コレクション

- view_schema (Owner)
- view_name (Name)
- column_name (Column)

Views コレクション

- view_schema (Owner)
- view_name (Name)

参照

- SAConnection クラス [SQL Anywhere .NET]169 ページ

Open メソッド

SAConnection.ConnectionString で指定されたプロパティ設定を使用してデータベース接続を開きます。

Visual Basic の構文

```
Public Overrides Sub Open()
```

C# の構文

```
public override void Open()
```

参照

- SAConnection.ConnectionString プロパティ [SQL Anywhere .NET]186 ページ

ConnectionString プロパティ

データベース接続文字列を指定します。

Visual Basic の構文

```
Public Overrides Property ConnectionString As String
```

C# の構文

```
public override string ConnectionString {get;set;}
```

備考

ConnectionString は、SQL Anywhere 接続文字列のフォーマットとできるかぎり同じになるよう定義されています。ただし、Persist Security Info 値が false (デフォルト) に設定されている場合、返される接続文字列は、ユーザ定義の ConnectionString からセキュリティ情報を除いたものと同じになります。Persist Security Info 値を true に設定しないかぎり、SQL Anywhere .NET データプロバイダは返される接続文字列にパスワードを保持しません。

ConnectionString プロパティを使用して、さまざまなデータソースに接続できます。

ConnectionString プロパティを設定できるのは、接続が閉じられている場合のみです。接続文字列値の多くには、対応する読み込み専用プロパティがあります。接続文字列が設定されると、エラーが検出されないかぎり、これらのプロパティのすべてが更新されます。エラーが検出されると、いずれのプロパティも更新されません。SAConnection プロパティは、ConnectionString に含まれているこれらの設定のみを返します。

閉じられた接続上で ConnectionString をリセットすると、パスワードを含むすべての接続文字列値と関連プロパティがリセットされます。

プロパティが設定されると、接続文字列の事前検証が実行されます。アプリケーションが Open メソッドを呼び出すと、接続文字列が完全に検証されます。接続文字列に無効なプロパティやサポートされていないプロパティが含まれる場合、ランタイム例外が生成されます。

値は、一重または二重引用符で区切ることができます。また、接続文字列内では一重または二重引用符を交互に使用できます。たとえば、name="value's" や name= 'value"s' は使用できますが、name='value's' や name= ""value"" は使用できません。値または引用符内に配置されていないブランク文字は無視されます。keyword=value ペアはセミコロンで区切ってください。セミコロンが値の一部である場合、セミコロンも引用符で区切ってください。エスケープシーケンスはサポートされておらず、値タイプは関係ありません。名前の大文字と小文字は区別されません。接続文字列内でプロパティ名が複数回使用されている場合、最後のプロパティ名に関連付けられている値が使用されます。

ウィンドウからユーザ ID やパスワードを取得し、これらを接続文字列に付加する場合のように、ユーザ入力に基づいて接続文字列を構成するときは注意してください。アプリケーションでは、ユーザがこれらの値に余分な接続文字列パラメータを埋め込めないようにする必要があります。

接続プーリングのデフォルト値は、true (pooling=true) です。

参照

- SAConnection クラス [SQL Anywhere .NET]169 ページ
- SAConnection.Open メソッド [SQL Anywhere .NET]185 ページ

例

次の文は、SQL Anywhere 16 Demo という名前の ODBC データソースに接続文字列を設定し、接続を開きます。

```
SAConnection conn = new SAConnection();
conn.ConnectionString = "DSN=SQL Anywhere 16 Demo";
conn.Open();
```

ConnectionTimeout プロパティ

接続試行がエラーでタイムアウトするまでの秒数を取得します。

Visual Basic の構文

```
Public ReadOnly Overrides Property ConnectionTimeout As Integer
```

C# の構文

```
public override int ConnectionTimeout {get;}
```

備考

ConnectionTimeout のデフォルト値は 15 秒です。

例

次の文は、ConnectionTimeout の値を表示します。

```
MessageBox.Show( conn.ConnectionTimeout.ToString( ) );
```

Database プロパティ

現在のデータベースの名前を取得します。

Visual Basic の構文

```
Public ReadOnly Overrides Property Database As String
```

C# の構文

```
public override string Database {get;}
```

備考

接続が開かれると、SAConnection は現在のデータベースの名前を返します。それ以外の場合、SAConnection は DatabaseName、DBN、DataSourceName、Data Source、DSN、DatabaseFile、DBF、FileDataSourceName、FileDSN の順に接続文字列を参照します。

DataSource プロパティ

データベースサーバの名前を取得します。

Visual Basic の構文

```
Public ReadOnly Overrides Property DataSource As String
```

C# の構文

```
public override string DataSource {get;}
```

備考

接続が開かれると、SAConnection オブジェクトが ServerName サーバプロパティを返します。それ以外の場合、SAConnection オブジェクトは ServerName 接続パラメータの接続文字列を参照します。

参照

- SAConnection クラス [SQL Anywhere .NET]169 ページ

InitString プロパティ

接続の確立直後に実行するコマンドです。

Visual Basic の構文

```
Public Property InitString As String
```

C# の構文

```
public string InitString {get;set;}
```

備考

InitString は、接続が開かれた直後に実行されます。

ServerVersion プロパティ

クライアントが接続する SQL Anywhere のインスタンスのバージョンが含まれている文字列を取得します。

Visual Basic の構文

```
Public ReadOnly Overrides Property ServerVersion As String
```

C# の構文

```
public override string ServerVersion {get;}
```

戻り値

SQL Anywhere のインスタンスのバージョン。

備考

バージョンのフォームは ##.##.#### です。最初の 2 桁はメジャーバージョン、次の 2 桁はマイナーバージョン、最後の 4 桁はリリースバージョンを示します。付加されている文字列のフォームは major.minor.build です。major と minor は 2 桁、build は 4 桁です。

State プロパティ

SAConnection オブジェクトのステータスを示します。

Visual Basic の構文

```
Public ReadOnly Overrides Property State As ConnectionState
```

C# の構文

```
public override ConnectionState State {get;}
```

戻り値

System.Data.ConnectionState 列挙。

InfoMessage イベント

SQL Anywhere データベースサーバが警告または情報メッセージを返すときに発生します。

Visual Basic の構文

```
Public Event InfoMessage As SAInfoMessageEventHandler
```

C# の構文

```
public event SAInfoMessageEventHandler InfoMessage;
```

備考

イベントハンドラは、このイベントに関するデータが含まれるタイプ SAInfoMessageEventArgs の引数を受け取ります。次の SAInfoMessageEventArgs プロパティは、このイベントに固有の情報 (NativeError、Errors、Message、MessageType、Source) を提供します。

詳細については、.NET Framework のマニュアルの OleDbConnection.InfoMessage イベントを参照してください。

StateChange イベント

SAConnection オブジェクトのステータスが変わると発生します。

Visual Basic の構文

```
Public Event StateChange As StateChangeEventHandler
```

C# の構文

```
public event override StateChangeEventHandler StateChange;
```

備考

イベントハンドラは、このイベントに関するデータが含まれるタイプ StateChangeEventArgs の引数を受け取ります。次の StateChangeEventArgs プロパティは、このイベントに固有の情報 (CurrentState および OriginalState) を提供します。

詳細については、.NET Framework のマニュアルの OleDbConnection.StateChange イベントを参照してください。

SAConnectionStringBuilder クラス

SAConnection クラスが使用する接続文字列の内容を作成および管理する単純な方法を提供します。

Visual Basic の構文

```
Public NotInheritable Class SAConnectionStringBuilder
    Inherits SAConnectionStringBuilderBase
```

C# の構文

```
public sealed class SAConnectionStringBuilder :
    SAConnectionStringBuilderBase
```

基本クラス

- SAConnectionStringBuilderBase クラス [SQL Anywhere .NET]208 ページ

メンバー

継承されたメンバーを含む SAConnectionStringBuilder クラスのすべてのメンバー。

名前	説明
SAConnectionStringBuilder コンストラクタ	SAConnectionStringBuilder クラスの新しいインスタンスを初期化します。
Add method (System.Data.Common.DbConnectionStringBuilder から継承)	指定されたキーと値を持つエントリを System.Data.Common.DbConnectionStringBuilder に追加します。
AppendKeyValuePair method (System.Data.Common.DbConnectionStringBuilder から継承)	キーと値を既存の System.Text.StringBuilder オブジェクトに追加する十分で安全な方法を提供します。
Clear method (System.Data.Common.DbConnectionStringBuilder から継承)	System.Data.Common.DbConnectionStringBuilder インスタンスの内容をクリアします。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

名前	説明
ClearPropertyDescriptors method (System.Data.Common.DbConnectionStringBuilder から継承)	関連する System.Data.Common.DbConnectionStringBuilder の System.ComponentModel.PropertyDescriptor オブジェクトのコレクションをクリアします。
ContainsKey メソッド	SAConnectionStringBuilder オブジェクトに特定のキーワードが含まれているかどうかを判断します。
EquivalentTo method (System.Data.Common.DbConnectionStringBuilder から継承)	この System.Data.Common.DbConnectionStringBuilder オブジェクト内の接続情報と指定されたオブジェクト内の接続情報を比較します。
GetKeyword メソッド	指定された SAConnectionStringBuilder プロパティのキーワードを取得します。
GetProperties method (System.Data.Common.DbConnectionStringBuilder から継承)	提供される System.Collections.Hashtable にこの System.Data.Common.DbConnectionStringBuilder のすべてのプロパティに関する情報を格納します。
GetUseLongNameAsKeyword メソッド	長い接続パラメータ名を接続文字列で使用するかどうかを示す boolean 値を取得します。
Remove メソッド	指定されたキーが設定されたエントリを SAConnectionStringBuilder インスタンスから削除します。
SetUseLongNameAsKeyword メソッド	長い接続パラメータ名を接続文字列で使用するかどうかを示す boolean 値を設定します。
ShouldSerialize メソッド	指定されたキーが、この SAConnectionStringBuilder インスタンスに存在するかどうかを示します。
ToString method (System.Data.Common.DbConnectionStringBuilder から継承)	この System.Data.Common.DbConnectionStringBuilder に関連付けられた接続文字列を返します。
TryGetValue メソッド	入力されたキーに対応する値を、この SAConnectionStringBuilder から取り出します。

名前	説明
AppInfo プロパティ	AppInfo 接続プロパティを取得または設定します。
AutoStart プロパティ	AutoStart 接続プロパティを取得または設定します。
AutoStop プロパティ	AutoStop 接続プロパティを取得または設定します。
BrowsableConnectionString property (System.Data.Common.DbConnectionStringBuilder から継承)	Visual Studio デザイナで System.Data.Common.DbConnectionStringBuilder.ConnectionString プロパティを表示するかどうかを示す値を取得または設定します。
Charset プロパティ	Charset 接続プロパティを取得または設定します。
CommBufferSize プロパティ	CommBufferSize 接続プロパティを取得または設定します。
CommLinks プロパティ	CommLinks プロパティを取得または設定します。
Compress プロパティ	Compress 接続プロパティを取得または設定します。
CompressionThreshold プロパティ	CompressionThreshold 接続プロパティを取得または設定します。
ConnectionLifetime プロパティ	ConnectionLifetime 接続プロパティを取得または設定します。
ConnectionName プロパティ	ConnectionName 接続プロパティを取得または設定します。
ConnectionPool プロパティ	ConnectionPool プロパティを取得または設定します。
ConnectionReset プロパティ	ConnectionReset 接続プロパティを取得または設定します。
ConnectionString property (System.Data.Common.DbConnectionStringBuilder から継承)	System.Data.Common.DbConnectionStringBuilder に関連付けられた接続文字列を取得または設定します。
ConnectionTimeout プロパティ	ConnectionTimeout 接続プロパティを取得または設定します。

名前	説明
Count property (System.Data.Common.DbConnectionStringBuilder から継承)	System.Data.Common.DbConnectionStringBuilder.ConnectionString プロパティに含まれているキーの現在の数を取得します。
DatabaseFile プロパティ	DatabaseFile 接続プロパティを取得または設定します。
DatabaseKey プロパティ	DatabaseKey 接続プロパティを取得または設定します。
DatabaseName プロパティ	DatabaseName 接続プロパティを取得または設定します。
DatabaseSwitches プロパティ	DatabaseSwitches 接続プロパティを取得または設定します。
DataSourceName プロパティ	DataSourceName 接続プロパティを取得または設定します。
DisableMultiRowFetch プロパティ	DisableMultiRowFetch 接続プロパティを取得または設定します。
Elevate プロパティ	Elevate 接続プロパティを取得または設定します。
EncryptedPassword プロパティ	EncryptedPassword 接続プロパティを取得または設定します。
Encryption プロパティ	Encryption 接続プロパティを取得または設定します。
Enlist プロパティ	Enlist 接続プロパティを取得または設定します。
FileDataSourceName プロパティ	FileDataSourceName 接続プロパティを取得または設定します。
ForceStart プロパティ	ForceStart 接続プロパティを取得または設定します。
Host プロパティ	Host プロパティを取得または設定します。
IdleTimeout プロパティ	IdleTimeout 接続プロパティを取得または設定します。
InitString プロパティ	InitString 接続プロパティを取得または設定します。

名前	説明
Integrated プロパティ	Integrated 接続プロパティを取得または設定します。
IsFixedSize property (System.Data.Common.DbConnectionStringBuilder から継承)	System.Data.Common.DbConnectionStringBuilder のサイズが固定かどうかを示す値を取得します。
IsReadOnly property (System.Data.Common.DbConnectionStringBuilder から継承)	System.Data.Common.DbConnectionStringBuilder が読み込み専用かどうかを示す値を取得します。
Kerberos プロパティ	Kerberos 接続プロパティを取得または設定します。
Keys プロパティ	SAConnectionStringBuilder のキーが含まれた System.Collections.ICollection を取得します。
Language プロパティ	Language 接続プロパティを取得または設定します。
LazyClose プロパティ	LazyClose 接続プロパティを取得または設定します。
LivenessTimeout プロパティ	LivenessTimeout 接続プロパティを取得または設定します。
LogFile プロパティ	LogFile 接続プロパティを取得または設定します。
MaxPoolSize プロパティ	MaxPoolSize 接続プロパティを取得または設定します。
MinPoolSize プロパティ	MinPoolSize 接続プロパティを取得または設定します。
NewPassword プロパティ	NewPassword 接続プロパティを取得または設定します。
NodeType プロパティ	NodeType プロパティを取得または設定します。
Password プロパティ	Password 接続プロパティを取得または設定します。
PersistSecurityInfo プロパティ	PersistSecurityInfo 接続プロパティを取得または設定します。

名前	説明
Pooling プロパティ	Pooling 接続プロパティを取得または設定します。
PrefetchBuffer プロパティ	PrefetchBuffer 接続プロパティを取得または設定します。
PrefetchRows プロパティ	PrefetchRows 接続プロパティを取得または設定します。
RetryConnectionTimeout プロパティ	RetryConnectionTimeout プロパティを取得または設定します。
ServerName プロパティ	ServerName 接続プロパティを取得または設定します。
StartLine プロパティ	StartLine 接続プロパティを取得または設定します。
this プロパティ	接続キーワードの値を取得または設定します。
Unconditional プロパティ	Unconditional 接続プロパティを取得または設定します。
UserID プロパティ	UserID 接続プロパティを取得または設定します。
Values property (System.Data.Common.DbConnectionStringBuilder から継承)	System.Data.Common.DbConnectionStringBuilder に値が含まれている System.Collections.ICollection を取得します。

備考

SAConnectionStringBuilder クラスは SAConnectionStringBuilderBase を継承し、SAConnectionStringBuilderBase は DbConnectionStringBuilder を継承しています。

SAConnectionStringBuilder クラスは、.NET Compact Framework 2.0 では使用できません。

接続パラメータのリストについては、「接続パラメータ」『SQL Anywhere サーバ データベース管理』を参照してください。

SAConnectionStringBuilder コンストラクタ

SAConnectionStringBuilder クラスの新しいインスタンスを初期化します。

オーバロードリスト

名前	説明
SAConnectionStringBuilder() コンストラクタ	SAConnectionStringBuilder クラスの新しいインスタンスを初期化します。
SAConnectionStringBuilder(string) コンストラクタ	SAConnectionStringBuilder クラスの新しいインスタンスを初期化します。

SAConnectionStringBuilder() コンストラクタ

SAConnectionStringBuilder クラスの新しいインスタンスを初期化します。

Visual Basic の構文

```
Public Sub New()
```

C# の構文

```
public SAConnectionStringBuilder()
```

備考

SAConnectionStringBuilder クラスは、.NET Compact Framework 2.0 では使用できません。

SAConnectionStringBuilder(string) コンストラクタ

SAConnectionStringBuilder クラスの新しいインスタンスを初期化します。

Visual Basic の構文

```
Public Sub New(ByVal connectionString As String)
```

C# の構文

```
public SAConnectionStringBuilder(string connectionString)
```

パラメータ

- **connectionString**　オブジェクトの内部接続情報の基礎。keyword=value のペアに解析された情報です。接続パラメータのリストについては、「接続パラメータ」『SQL Anywhere サーバ データベース管理』を参照してください。

備考

SAConnectionStringBuilder クラスは、.NET Compact Framework 2.0 では使用できません。

例

次の文は、hr という名前の SQL Anywhere データベースサーバ上で動作している policies という名前のデータベースへの接続について SAConnection オブジェクトを初期化します。接続では、ユーザ ID admin とパスワード money を使用します。

```
SAConnectionStringBuilder conn = new SAConnectionStringBuilder(
   "UID=admin;PWD=money;SERVER=hr;DBN=policies" );
```

AppInfo プロパティ

AppInfo 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property AppInfo As String
```

C# の構文

```
public string AppInfo {get;set;}
```

AutoStart プロパティ

AutoStart 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property AutoStart As String
```

C# の構文

```
public string AutoStart {get;set;}
```

AutoStop プロパティ

AutoStop 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property AutoStop As String
```

C# の構文

```
public string AutoStop {get;set;}
```

Charset プロパティ

Charset 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property Charset As String
```

C# の構文

```
public string Charset {get;set;}
```

CommBufferSize プロパティ

CommBufferSize 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property CommBufferSize As Integer
```

C# の構文

```
public int CommBufferSize {get;set;}
```

CommLinks プロパティ

CommLinks プロパティを取得または設定します。

Visual Basic の構文

```
Public Property CommLinks As String
```

C# の構文

```
public string CommLinks {get;set;}
```

Compress プロパティ

Compress 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property Compress As String
```

C# の構文

```
public string Compress {get;set;}
```

CompressionThreshold プロパティ

CompressionThreshold 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property CompressionThreshold As Integer
```

C# の構文

```
public int CompressionThreshold {get;set;}
```

ConnectionLifetime プロパティ

ConnectionLifetime 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property ConnectionLifetime As Integer
```

C# の構文

```
public int ConnectionLifetime {get;set;}
```

ConnectionName プロパティ

ConnectionName 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property ConnectionName As String
```

C# の構文

```
public string ConnectionName {get;set;}
```

ConnectionPool プロパティ

ConnectionPool プロパティを取得または設定します。

Visual Basic の構文

```
Public Property ConnectionPool As String
```

C# の構文

```
public string ConnectionPool {get;set;}
```

ConnectionReset プロパティ

ConnectionReset 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property ConnectionReset As Boolean
```

C# の構文

```
public bool ConnectionReset {get;set;}
```

戻り値

スキーマ情報が格納されている DataTable。

ConnectionTimeout プロパティ

ConnectionTimeout 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property ConnectionTimeout As Integer
```

C# の構文

```
public int ConnectionTimeout {get;set;}
```

例

次の文は、ConnectionTimeout プロパティの値を表示します。

```
MessageBox.Show( connString.ConnectionTimeout.ToString() );
```

DatabaseFile プロパティ

DatabaseFile 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property DatabaseFile As String
```

C# の構文

```
public string DatabaseFile {get;set;}
```

DatabaseKey プロパティ

DatabaseKey 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property DatabaseKey As String
```

C# の構文

```
public string DatabaseKey {get;set;}
```

DatabaseName プロパティ

DatabaseName 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property DatabaseName As String
```

C# の構文

```
public string DatabaseName {get;set;}
```

DatabaseSwitches プロパティ

DatabaseSwitches 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property DatabaseSwitches As String
```

C# の構文

```
public string DatabaseSwitches {get;set;}
```

DataSourceName プロパティ

DataSourceName 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property DataSourceName As String
```

C# の構文

```
public string DataSourceName {get;set;}
```

DisableMultiRowFetch プロパティ

DisableMultiRowFetch 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property DisableMultiRowFetch As String
```

C# の構文

```
public string DisableMultiRowFetch {get;set;}
```

Elevate プロパティ

Elevate 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property Elevate As String
```

C# の構文

```
public string Elevate {get;set;}
```

EncryptedPassword プロパティ

EncryptedPassword 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property EncryptedPassword As String
```

C# の構文

```
public string EncryptedPassword {get;set;}
```

Encryption プロパティ

Encryption 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property Encryption As String
```

C# の構文

```
public string Encryption {get;set;}
```

Enlist プロパティ

Enlist 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property Enlist As Boolean
```

C# の構文

```
public bool Enlist {get;set;}
```

FileDataSourceName プロパティ

FileDataSourceName 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property FileDataSourceName As String
```

C# の構文

```
public string FileDataSourceName {get;set;}
```

ForceStart プロパティ

ForceStart 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property ForceStart As String
```

C# の構文

```
public string ForceStart {get;set;}
```

Host プロパティ

Host プロパティを取得または設定します。

Visual Basic の構文

```
Public Property Host As String
```

C# の構文

```
public string Host {get;set;}
```

IdleTimeout プロパティ

IdleTimeout 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property IdleTimeout As Integer
```

C# の構文

```
public int IdleTimeout {get;set;}
```

InitString プロパティ

InitString 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property InitString As String
```

C# の構文

```
public string InitString {get;set;}
```

Integrated プロパティ

Integrated 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property Integrated As String
```

C# の構文

```
public string Integrated {get;set;}
```

Kerberos プロパティ

Kerberos 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property Kerberos As String
```

C# の構文

```
public string Kerberos {get;set;}
```

Language プロパティ

Language 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property Language As String
```

C# の構文

```
public string Language {get;set;}
```

LazyClose プロパティ

LazyClose 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property LazyClose As String
```

C# の構文

```
public string LazyClose {get;set;}
```

LivenessTimeout プロパティ

LivenessTimeout 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property LivenessTimeout As Integer
```

C# の構文

```
public int LivenessTimeout {get;set;}
```

LogFile プロパティ

LogFile 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property LogFile As String
```

C# の構文

```
public string LogFile {get;set;}
```

MaxPoolSize プロパティ

MaxPoolSize 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property MaxPoolSize As Integer
```

C# の構文

```
public int MaxPoolSize {get;set;}
```

MinPoolSize プロパティ

MinPoolSize 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property MinPoolSize As Integer
```

C# の構文

```
public int MinPoolSize {get;set;}
```

NewPassword プロパティ

NewPassword 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property NewPassword As String
```

C# の構文

```
public string NewPassword {get;set;}
```

NodeType プロパティ

NodeType プロパティを取得または設定します。

Visual Basic の構文

```
Public Property NodeType As String
```

C# の構文

```
public string NodeType {get;set;}
```

Password プロパティ

Password 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property Password As String
```

C# の構文

```
public string Password {get;set;}
```

PersistSecurityInfo プロパティ

PersistSecurityInfo 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property PersistSecurityInfo As Boolean
```

C# の構文

```
public bool PersistSecurityInfo {get;set;}
```

Pooling プロパティ

Pooling 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property Pooling As Boolean
```

C# の構文

```
public bool Pooling {get;set;}
```

PrefetchBuffer プロパティ

PrefetchBuffer 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property PrefetchBuffer As Integer
```

C# の構文

```
public int PrefetchBuffer {get;set;}
```

PrefetchRows プロパティ

PrefetchRows 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property PrefetchRows As Integer
```

C# の構文

```
public int PrefetchRows {get;set;}
```

備考

デフォルト値は 200 です。

RetryConnectionTimeout プロパティ

RetryConnectionTimeout プロパティを取得または設定します。

Visual Basic の構文

```
Public Property RetryConnectionTimeout As Integer
```

C# の構文

```
public int RetryConnectionTimeout {get;set;}
```

ServerName プロパティ

ServerName 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property ServerName As String
```

C# の構文

```
public string ServerName {get;set;}
```

StartLine プロパティ

StartLine 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property StartLine As String
```

C# の構文

```
public string StartLine {get;set;}
```

Unconditional プロパティ

Unconditional 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property Unconditional As String
```

C# の構文

```
public string Unconditional {get;set;}
```

UserID プロパティ

UserID 接続プロパティを取得または設定します。

Visual Basic の構文

```
Public Property UserID As String
```

C# の構文

```
public string UserID {get;set;}
```

SAConnectionStringBuilderBase クラス

SAConnectionStringBuilder クラスの基本クラスです。

Visual Basic の構文

```
Public MustInherit Class SAConnectionStringBuilderBase
    Inherits System.Data.Common.DbConnectionStringBuilder
```

C# の構文

```
public abstract class SAConnectionStringBuilderBase :
    System.Data.Common.DbConnectionStringBuilder
```

基本クラス

- System.Data.Common.DbConnectionStringBuilder

派生クラス

- SAConnectionStringBuilder クラス [SQL Anywhere .NET]190 ページ
- SATcpOptionsBuilder クラス [SQL Anywhere .NET]316 ページ

メンバー

継承されたメンバーを含む SAConnectionStringBuilderBase クラスのすべてのメンバー。

名前	説明
Add method (System.Data.Common.DbConnectionStringBuilder から継承)	指定されたキーと値を持つエントリを System.Data.Common.DbConnectionStringBuilder に追加します。

名前	説明
AppendKeyValuePair method (System.Data.Common.DbConnectionStringBuilder から継承)	キーと値を既存の System.Text.StringBuilder オブジェクトに追加する十分で安全な方法を提供します。
Clear method (System.Data.Common.DbConnectionStringBuilder から継承)	System.Data.Common.DbConnectionStringBuilder インスタンスの内容をクリアします。
ClearPropertyDescriptors method (System.Data.Common.DbConnectionStringBuilder から継承)	関連する System.Data.Common.DbConnectionStringBuilder の System.ComponentModel.PropertyDescriptor オブジェクトのコレクションをクリアします。
ContainsKey メソッド	SAConnectionStringBuilder オブジェクトに特定のキーワードが含まれているかどうかを判断します。
EquivalentTo method (System.Data.Common.DbConnectionStringBuilder から継承)	この System.Data.Common.DbConnectionStringBuilder オブジェクト内の接続情報と指定されたオブジェクト内の接続情報を比較します。
GetKeyword メソッド	指定された SAConnectionStringBuilder プロパティのキーワードを取得します。
GetProperties method (System.Data.Common.DbConnectionStringBuilder から継承)	提供される System.Collections.Hashtable にこの System.Data.Common.DbConnectionStringBuilder のすべてのプロパティに関する情報を格納します。
GetUseLongNameAsKeyword メソッド	長い接続パラメータ名を接続文字列で使用するかどうかを示す boolean 値を取得します。
Remove メソッド	指定されたキーが設定されたエントリを SAConnectionStringBuilder インスタンスから削除します。
SetUseLongNameAsKeyword メソッド	長い接続パラメータ名を接続文字列で使用するかどうかを示す boolean 値を設定します。
ShouldSerialize メソッド	指定されたキーが、この SAConnectionStringBuilder インスタンスに存在するかどうかを示します。

名前	説明
ToString method (System.Data.Common.DbConnectionStringBuilder から継承)	この System.Data.Common.DbConnectionStringBuilder に関連付けられた接続文字列を返します。
TryGetValue メソッド	入力されたキーに対応する値を、この SAConnectionStringBuilder から取り出します。
BrowsableConnectionString property (System.Data.Common.DbConnectionStringBuilder から継承)	Visual Studio デザイナで System.Data.Common.DbConnectionStringBuilder.ConnectionString プロパティを表示するかどうかを示す値を取得または設定します。
ConnectionString property (System.Data.Common.DbConnectionStringBuilder から継承)	System.Data.Common.DbConnectionStringBuilder に関連付けられた接続文字列を取得または設定します。
Count property (System.Data.Common.DbConnectionStringBuilder から継承)	System.Data.Common.DbConnectionStringBuilder.ConnectionString プロパティに含まれているキーの現在の数を取得します。
IsFixedSize property (System.Data.Common.DbConnectionStringBuilder から継承)	System.Data.Common.DbConnectionStringBuilder のサイズが固定かどうかを示す値を取得します。
IsReadOnly property (System.Data.Common.DbConnectionStringBuilder から継承)	System.Data.Common.DbConnectionStringBuilder が読み込み専用かどうかを示す値を取得します。
Keys プロパティ	SAConnectionStringBuilder のキーが含まれた System.Collections.ICollection を取得します。
this プロパティ	接続キーワードの値を取得または設定します。
Values property (System.Data.Common.DbConnectionStringBuilder から継承)	System.Data.Common.DbConnectionStringBuilder に値が含まれている System.Collections.ICollection を取得します。

ContainsKey メソッド

SAConnectionStringBuilder オブジェクトに特定のキーワードが含まれているかどうかを判断します。

Visual Basic の構文

```
Public Overrides Function ContainsKey(
    ByVal keyword As String
) As Boolean
```

C# の構文

```
public override bool ContainsKey(string keyword)
```

パラメータ

- **keyword** SAConnectionStringBuilder 内で検索するキーワード。

戻り値

keyword に関連する値が設定されている場合は true、設定されていない場合は false。

例

次の文は、SAConnectionStringBuilder オブジェクトに UserID キーワードが含まれているかどうかを判断します。

```
connectString.ContainsKey("UserID")
```

GetKeyword メソッド

指定された SAConnectionStringBuilder プロパティのキーワードを取得します。

Visual Basic の構文

```
Public Function GetKeyword(ByVal propName As String) As String
```

C# の構文

```
public string GetKeyword(string propName)
```

パラメータ

- **propName** SAConnectionStringBuilder プロパティの名前。

戻り値

指定された SAConnectionStringBuilder プロパティのキーワード。

GetUseLongNameAsKeyword メソッド

長い接続パラメータ名を接続文字列で使用するかどうかを示す boolean 値を取得します。

Visual Basic の構文

```
Public Function GetUseLongNameAsKeyword() As Boolean
```

C# の構文

```
public bool GetUseLongNameAsKeyword()
```

戻り値

長い接続パラメータ名を使用して接続文字列を作成する場合は true、それ以外の場合は false。

備考

SQL Anywhere 接続パラメータの名前には、長い名前と短い名前の 2 種類があります。たとえば、接続文字列に ODBC データソースの名前を指定する場合は、DataSourceName と DSN のいずれかを使用できます。DataSourceName または DSN。デフォルトでは、長い接続パラメータ名を使用して接続文字列を作成します。

参照

- SAConnectionStringBuilderBase.SetUseLongNameAsKeyword メソッド [SQL Anywhere .NET]212 ページ

Remove メソッド

指定されたキーが設定されたエントリを SAConnectionStringBuilder インスタンスから削除します。

Visual Basic の構文

```
Public Overrides Function Remove(ByVal keyword As String) As Boolean
```

C# の構文

```
public override bool Remove(string keyword)
```

パラメータ

- **keyword**　この SAConnectionStringBuilder 内の接続文字列から削除するキー／値のペアのキー。

戻り値

接続文字列内のキーが削除された場合は true、キーが存在しなかった場合は false。

SetUseLongNameAsKeyword メソッド

長い接続パラメータ名を接続文字列で使用するかどうかを示す boolean 値を設定します。

Visual Basic の構文

```
Public Sub SetUseLongNameAsKeyword(
    ByVal useLongNameAsKeyword As Boolean
)
```

C# の構文

```
public void SetUseLongNameAsKeyword(bool useLongNameAsKeyword)
```

パラメータ

- **useLongNameAsKeyword**　長い接続パラメータ名を接続文字列で使用するかどうかを示す boolean 値。

備考

デフォルトでは、長い接続パラメータ名が使用されます。

参照

- SAConnectionStringBuilderBase.GetUseLongNameAsKeyword メソッド [SQL Anywhere .NET]211 ページ

ShouldSerialize メソッド

指定されたキーが、この SAConnectionStringBuilder インスタンスに存在するかどうかを示します。

Visual Basic の構文

```
Public Overrides Function ShouldSerialize(
    ByVal keyword As String
) As Boolean
```

C# の構文

```
public override bool ShouldSerialize(string keyword)
```

パラメータ

- **keyword**　SAConnectionStringBuilder 内で検索するキー。

戻り値

SAConnectionStringBuilder に指定されたキーが設定されたエントリが含まれている場合は true、それ以外の場合は false。

TryGetValue メソッド

入力されたキーに対応する値を、この SAConnectionStringBuilder から取り出します。

Visual Basic の構文

```
Public Overrides Function TryGetValue(
    ByVal keyword As String,
    ByVal value As Object
) As Boolean
```

C# の構文

```
public override bool TryGetValue(string keyword, out object value)
```

パラメータ

- **keyword**　取り出す項目のキー。
- **value**　キーワードに対応する値。

戻り値

キーワードが接続文字列内にある場合は true、それ以外は false。

Keys プロパティ

SAConnectionStringBuilder のキーが含まれた System.Collections.ICollection を取得します。

Visual Basic の構文

```
Public ReadOnly Overrides Property Keys As ICollection
```

C# の構文

```
public override ICollection Keys {get;}
```

戻り値

SAConnectionStringBuilder のキーが含まれた System.Collections.ICollection。

this プロパティ

接続キーワードの値を取得または設定します。

Visual Basic の構文

```
Public Overrides Property Item(ByVal keyword As String) As Object
```

C# の構文

```
public override object this[string keyword] {get;set;}
```

パラメータ

- **keyword**　接続キーワードの名前。

備考

指定された接続キーワードの値を表すオブジェクト。

キーワードまたは型が無効の場合は、例外が発生します。キーワードは大文字と小文字を区別します。

値の設定で NULL が渡されると、値がクリアされます。

SADataAdapter クラス

System.Data.DataSet に入力したりデータベースを更新したりするために使用する一連のコマンドとデータベース接続を表します。

Visual Basic の構文

```
Public NotInheritable Class SADataAdapter
    Inherits System.Data.Common.DbDataAdapter
    Implements System.ICloneable
```

C# の構文

```
public sealed class SADataAdapter :
    System.Data.Common.DbDataAdapter,
    System.ICloneable
```

基本クラス

- System.Data.Common.DbDataAdapter
- System.ICloneable

メンバー

継承されたメンバーを含む SADataAdapter クラスのすべてのメンバー。

名前	説明
SADataAdapter コンストラクタ	SADataAdapter オブジェクトを初期化します。
AddToBatch method (System.Data.Common.DbDataAdapter から継承)	System.Data.IDbCommand を現在のバッチに追加します。
CreateRowUpdatedEvent method (System.Data.Common.DbDataAdapter から継承)	System.Data.Common.RowUpdatedEventArgs クラスの新しいインスタンスを初期化します。
CreateRowUpdatingEvent method (System.Data.Common.DbDataAdapter から継承)	System.Data.Common.RowUpdatingEventArgs クラスの新しいインスタンスを初期化します。
Dispose method (System.Data.Common.DbDataAdapter から継承)	System.Data.Common.DbDataAdapter によって使用されるアンマネージリソースを解放するほか、必要に応じてマネージリソースを解放します。
ExecuteBatch method (System.Data.Common.DbDataAdapter から継承)	現在のバッチを実行します。

名前	説明
Fill method (System.Data.Common.DbDataAdapter から継承)	System.Data.DataSet で、ローを追加または再表示します。
FillSchema method (System.Data.Common.DbDataAdapter から継承)	"Table" という名前の System.Data.DataTable を指定された System.Data.DataSet に追加し、指定された System.Data.SchemaType に基づいてスキーマがデータソースのスキーマと一致するように設定します。
GetBatchedParameter method (System.Data.Common.DbDataAdapter から継承)	現在のバッチにあるコマンドの 1 つから System.Data.IDataParameter を返します。
GetBatchedRecordsAffected method (System.Data.Common.DbDataAdapter から継承)	大規模なバッチ更新内の個別の更新試行に関する情報を返します。
GetFillParameters メソッド	SELECT 文の実行時に自分が設定したパラメータを返します。
OnRowUpdated method (System.Data.Common.DbDataAdapter から継承)	.NET Framework データプロバイダの RowUpdated イベントを生成します。
OnRowUpdating method (System.Data.Common.DbDataAdapter から継承)	.NET Framework データプロバイダの RowUpdating イベントを生成します。
Update method (System.Data.Common.DbDataAdapter から継承)	System.Data.DataRow オブジェクトの指定された配列で挿入、更新、削除されたローに対して、それぞれ INSERT、UPDATE、DELETE 文を呼び出します。
DeleteCommand プロパティ	DataSet で削除されたローに該当するデータベース内のローを削除するために、Update メソッドが呼び出されたときにデータベースに対して実行される SACommand オブジェクトを指定します。
FillCommandBehavior property (System.Data.Common.DbDataAdapter から継承)	データアダプタの入力に使用されるコマンドの動作を取得または設定します。

名前	説明
InsertCommand プロパティ	DataSet に挿入されたローに該当するローをデータベースに追加するために、Update メソッドが呼び出されたときにデータベースに対して実行される SACommand オブジェクトを指定します。
SelectCommand プロパティ	Fill または FillSchema の実行時に、DataSet にコピーするための結果セットをデータベースから取得するために使用される SACommand を指定します。
TableMappings プロパティ	ソーステーブルと DataTable 間のマスタマッピングを提供するコレクションを指定します。
UpdateBatchSize プロパティ	サーバとの各往復の中で処理されるローの数を取得または設定します。
UpdateCommand プロパティ	DataSet で更新されたローに該当するデータベース内のローを更新するために、Update メソッドが呼び出されたときにデータベースに対して実行される SACommand オブジェクトを指定します。
RowUpdated イベント	データソースに対してコマンドが実行された後の更新時に発生します。
RowUpdating イベント	データソースに対してコマンドが実行される前の更新時に発生します。

備考

System.Data.DataSet は、データをオフラインで処理するための方法を提供します。
SADataAdapter は、DataSet を一連の SQL 文に関連付けるためのメソッドを提供します。

Implements: IDbDataAdapter、IDataAdapter、ICloneable

詳細については、「SADataAdapter：概要」53 ページと「データへのアクセスとデータの操作」47 ページを参照してください。

SADataAdapter コンストラクタ

SADataAdapter オブジェクトを初期化します。

オーバロードリスト

名前	説明
SADataAdapter() コンストラクタ	SADataAdapter オブジェクトを初期化します。
SADataAdapter(SACommand) コンストラクタ	SADataAdapter オブジェクトを、指定した SELECT 文で初期化します。
SADataAdapter(string, SAConnection) コンストラクタ	SADataAdapter オブジェクトを、指定した SELECT 文および接続で初期化します。
SADataAdapter(string, string) コンストラクタ	SADataAdapter オブジェクトを、指定した SELECT 文および接続文字列で初期化します。

SADataAdapter() コンストラクタ

SADataAdapter オブジェクトを初期化します。

Visual Basic の構文

```
Public Sub New()
```

C# の構文

```
public SADataAdapter()
```

参照

- SADataAdapter.SADataAdapter コンストラクタ [SQL Anywhere .NET]217 ページ

SADataAdapter(SACommand) コンストラクタ

SADataAdapter オブジェクトを、指定した SELECT 文で初期化します。

Visual Basic の構文

```
Public Sub New(ByVal selectCommand As SACommand)
```

C# の構文

```
public SADataAdapter(SACommand selectCommand)
```

パラメータ

- **selectCommand** System.Data.DataSet に配置するためのレコードをデータソースから選択するために System.Data.Common.DbDataAdapter.Fill(System.Data.DataSet) の実行時に使用される SACommand オブジェクト。

参照

- SADataAdapter.SADataAdapter コンストラクタ [SQL Anywhere .NET]217 ページ

SADataAdapter(string, SAConnection) コンストラクタ

SADataAdapter オブジェクトを、指定した SELECT 文および接続で初期化します。

Visual Basic の構文

```
Public Sub New(
    ByVal selectCommandText As String,
    ByVal selectConnection As SAConnection
)
```

C# の構文

```
public SADataAdapter(
    string selectCommandText,
    SAConnection selectConnection
)
```

パラメータ

- **selectCommandText**　SADataAdapter オブジェクトの SADataAdapter.SelectCommand プロパティを設定するために使用される SELECT 文。
- **selectConnection**　データベースへの接続を定義する SAConnection オブジェクト。

参照

- SADataAdapter.SADataAdapter コンストラクタ [SQL Anywhere .NET]217 ページ
- SADataAdapter.SelectCommand プロパティ [SQL Anywhere .NET]221 ページ
- SAConnection クラス [SQL Anywhere .NET]169 ページ

SADataAdapter(string, string) コンストラクタ

SADataAdapter オブジェクトを、指定した SELECT 文および接続文字列で初期化します。

Visual Basic の構文

```
Public Sub New(
    ByVal selectCommandText As String,
    ByVal selectConnectionString As String
)
```

C# の構文

```
public SADataAdapter(
    string selectCommandText,
    string selectConnectionString
)
```

パラメータ

- **selectCommandText**　SADataAdapter オブジェクトの SADataAdapter.SelectCommand プロパティを設定するために使用される SELECT 文。
- **selectConnectionString**　SQL Anywhere データベースの接続文字列。

参照

- SADataAdapter.SADataAdapter コンストラクタ [SQL Anywhere .NET]217 ページ
- SADataAdapter.SelectCommand プロパティ [SQL Anywhere .NET]221 ページ

GetFillParameters メソッド

SELECT 文の実行時に自分が設定したパラメータを返します。

Visual Basic の構文

```
Public Shadows Function GetFillParameters() As SAParameter()
```

C# の構文

```
public new SAParameter[] GetFillParameters()
```

戻り値

ユーザによって設定されたパラメータが含まれる IDataParameter オブジェクトの配列。

DeleteCommand プロパティ

DataSet で削除されたローに該当するデータベース内のローを削除するために、Update メソッドが呼び出されたときにデータベースに対して実行される SACommand オブジェクトを指定します。

Visual Basic の構文

```
Public Shadows Property DeleteCommand As SACommand
```

C# の構文

```
public new SACommand DeleteCommand {get;set;}
```

備考

Update の実行時にこのプロパティが設定されておらず、DataSet にプライマリキー情報がある場合、SelectCommand を設定して SACommandBuilder を使用すると、DeleteCommand を自動的に生成できます。この場合、SACommandBuilder は、設定されていない追加コマンドを生成します。この生成論理には、SelectCommand に表示されるキーカラム情報が必要です。

DeleteCommand が既存の SACommand オブジェクトに割り当てられる場合、SACommand オブジェクトのクローンは作成されません。DeleteCommand は、既存の SACommand への参照を保持します。

参照

- SADataAdapter.SelectCommand プロパティ [SQL Anywhere .NET]221 ページ

InsertCommand プロパティ

DataSet に挿入されたローに該当するローをデータベースに追加するために、Update メソッドが呼び出されたときにデータベースに対して実行される SACommand オブジェクトを指定します。

Visual Basic の構文

```
Public Shadows Property InsertCommand As SACommand
```

C# の構文

```
public new SACommand InsertCommand {get;set;}
```

備考

SACommandBuilder には、InsertCommand を生成するためのキーカラムは必要ありません。

InsertCommand が既存の SACommand オブジェクトに割り当てられる場合、SACommand オブジェクトのクローンは作成されません。InsertCommand は、既存の SACommand への参照を保持します。

このコマンドがローを返す場合、SACommand オブジェクトの UpdatedRowSource プロパティの設定方法によっては、これらのローが DataSet に追加されることがあります。

SelectCommand プロパティ

Fill または FillSchema の実行時に、DataSet にコピーするための結果セットをデータベースから取得するために使用される SACommand を指定します。

Visual Basic の構文

```
Public Shadows Property SelectCommand As SACommand
```

C# の構文

```
public new SACommand SelectCommand {get;set;}
```

備考

SelectCommand が以前に作成された SACommand オブジェクトに割り当てられる場合、SACommand オブジェクトのクローンは作成されません。SelectCommand は、以前に作成された SACommand オブジェクトへの参照を保持します。

SelectCommand がローを返さない場合、DataSet にテーブルが追加されず、例外も発生しません。

SELECT 文は、SADataAdapter コンストラクタにも指定できます。

TableMappings プロパティ

ソーステーブルと DataTable 間のマスタマッピングを提供するコレクションを指定します。

Visual Basic の構文

```
Public ReadOnly Shadows Property TableMappings As
DataTableMappingCollection
```

C# の構文

```
public new DataTableMappingCollection TableMappings {get;}
```

備考

デフォルト値は空のコレクションです。

変更を調整する場合、SADataAdapter は DataTableMappingCollection コレクションを使用して、データソースによって使用されるカラム名を、DataSet によって使用されるカラム名に関連付けます。

制限：TableMappings プロパティは、.NET Compact Framework 2.0 では使用できません。

UpdateBatchSize プロパティ

サーバとの各往復の中で処理されるローの数を取得または設定します。

Visual Basic の構文

```
Public Overrides Property UpdateBatchSize As Integer
```

C# の構文

```
public override int UpdateBatchSize {get;set;}
```

備考

デフォルト値は 1 です。

1 より大きい値を設定すると、SADataAdapter.Update は、すべての挿入文をバッチで実行します。削除と更新はこれまでのように順次実行されますが、挿入は、UpdateBatchSize の値のサイズのバッチで、後で実行されます。0 を設定すると、Update は挿入文を 1 つのバッチで送信します。

1 より大きい値を設定すると、SADataAdapter.Fill は、すべての挿入文をバッチで実行します。削除と更新はこれまでのように順次実行されますが、挿入は、UpdateBatchSize の値のサイズのバッチで、後で実行されます。

0 を設定すると、Fill は挿入文を 1 つのバッチで送信します。

0 未満の値を設定すると、エラーになります。

UpdateBatchSize が 1 以外の値に設定され、InsertCommand プロパティが INSERT 文以外のものに設定されている場合、Fill を呼び出すときに例外がスローされます。

この動作は、SqlDataAdapter とは異なります。SqlDataAdapter は、すべての種類のコマンドをバッチ処理します。

UpdateCommand プロパティ

DataSet で更新されたローに該当するデータベース内のローを更新するために、Update メソッドが呼び出されたときにデータベースに対して実行される SACommand オブジェクトを指定します。

Visual Basic の構文

```
Public Shadows Property UpdateCommand As SACommand
```

C# の構文

```
public new SACommand UpdateCommand {get;set;}
```

備考

Update の実行時に、このプロパティが設定されておらず、SelectCommand にプライマリキー情報がある場合、SelectCommand プロパティを設定して SACommandBuilder を使用すると、UpdateCommand を自動的に生成できます。次に、設定していない追加コマンドが SACommandBuilder によって生成されます。この生成論理には、SelectCommand に表示されるキーカラム情報が必要です。

UpdateCommand が以前に作成された SACommand オブジェクトに割り当てられる場合、SACommand オブジェクトのクローンは作成されません。UpdateCommand は、以前に作成された SACommand オブジェクトへの参照を保持します。

このコマンドを実行するとローが返される場合、SACommand オブジェクトの UpdatedRowSource プロパティの設定方法によっては、これらのローが DataSet とマージされることがあります。

RowUpdated イベント

データソースに対してコマンドが実行された後の更新時に発生します。

Visual Basic の構文

```
Public Event RowUpdated As SARowUpdatedEventHandler
```

C# の構文

```
public event SARowUpdatedEventHandler RowUpdated;
```

備考

更新が試みられると、イベントが発生します。

イベントハンドラは、このイベントに関するデータが含まれるタイプ SARowUpdatedEventArgs の引数を受け取ります。

詳細については、.NET Framework のマニュアルの OleDbDataAdapter.RowUpdated イベントを参照してください。

RowUpdating イベント

データソースに対してコマンドが実行される前の更新時に発生します。

Visual Basic の構文

```
Public Event RowUpdating As SARowUpdatingEventHandler
```

C# の構文

```
public event SARowUpdatingEventHandler RowUpdating;
```

備考

更新が試みられると、イベントが発生します。

イベントハンドラは、このイベントに関するデータが含まれるタイプ SARowUpdatingEventArgs の引数を受け取ります。

詳細については、.NET Framework のマニュアルの OleDbDataAdapter.RowUpdating イベントを参照してください。

SADataReader クラス

クエリまたはストアドプロシージャからの読み込み専用、前方専用の結果セットです。

Visual Basic の構文

```
Public NotInheritable Class SADataReader
    Inherits System.Data.Common.DbDataReader
    Implements System.ComponentModel.IListSource
```

C# の構文

```
public sealed class SADataReader :
    System.Data.Common.DbDataReader,
    System.ComponentModel.IListSource
```

基本クラス

- System.Data.Common.DbDataReader
- System.ComponentModel.IListSource

メンバー

継承されたメンバーを含む SADataReader クラスのすべてのメンバー。

名前	説明
Close メソッド	SADataReader を閉じます。
Dispose method (System.Data.Common.DbDataReader から継承)	System.Data.Common.DbDataReader クラスの現在のインスタンスで使用しているすべてのリソースを解放します。

名前	説明
GetBoolean メソッド	指定されたカラムの値をブール値として返します。
GetByte メソッド	指定されたカラムの値をバイトとして返します。
GetBytes メソッド	指定されたカラムオフセットからバイトのストリームを特定のバッファオフセットから始まる配列としてバッファに読み込みます。
GetChar メソッド	指定されたカラムの値を文字として返します。
GetChars メソッド	指定されたカラムオフセットから文字のストリームを特定のバッファオフセットから始まる配列としてバッファに読み込みます。
GetData メソッド	このメソッドはサポートされていません。
GetDataTypeName メソッド	ソースデータ型の名前を返します。
GetDateTime メソッド	指定されたカラムの値を DateTime オブジェクトとして返します。
GetDateTimeOffset メソッド	指定されたカラムの値を DateTimeOffset オブジェクトとして返します。
GetDbDataReader method (System.Data.Common.DbDataReader から継承)	プロバイダ固有の実装によって無効になる可能性のある、要求されたカラム序数の System.Data.Common.DbDataReader オブジェクトを返します。
GetDecimal メソッド	指定されたカラムの値を Decimal オブジェクトとして返します。
GetDouble メソッド	指定されたカラムの値を倍精度の浮動小数点数として返します。
GetEnumerator メソッド	SADataReader オブジェクトで反復処理する System.Collections.IEnumerator を返します。
GetFieldType メソッド	オブジェクトのデータ型である Type を返します。
GetFieldValue(T) method (System.Data.Common.DbDataReader から継承)	指定されたカラムの値をタイプとして同期的に取得します。

名前	説明
GetFieldValueAsync method (System.Data.Common.DbDataReader から継承)	指定されたカラムの値をタイプとして非同期的に取得します。
GetFloat メソッド	指定されたカラムの値を単精度の浮動小数点数として返します。
GetGuid メソッド	指定されたカラムの値をグローバル一意識別子 (GUID) として返します。
GetInt16 メソッド	指定されたカラムの値を 16 ビット符号付き整数として返します。
GetInt32 メソッド	指定されたカラムの値を 32 ビット符号付き整数として返します。
GetInt64 メソッド	指定されたカラムの値を 64 ビット符号付き整数として返します。
GetName メソッド	指定されたカラムの名前を返します。
GetOrdinal メソッド	指定されたカラム名に対応するカラムの順序を返します。
GetProviderSpecificFieldType method (System.Data.Common.DbDataReader から継承)	指定されたカラムのプロバイダ固有のフィールドタイプを返します。
GetProviderSpecificValue method (System.Data.Common.DbDataReader から継承)	指定されたカラムの値を System.Object のインスタンスとして取得します。
GetProviderSpecificValues method (System.Data.Common.DbDataReader から継承)	現在のローのコレクション内のプロバイダ固有のすべての属性カラムを取得します。
GetSchemaTable メソッド	SADataReader のカラムメタデータが記述された DataTable を返します。
GetStream method (System.Data.Common.DbDataReader から継承)	データを System.IO.Stream として取得します。
GetString メソッド	指定されたカラムの値を文字列として返します。
GetTextReader method (System.Data.Common.DbDataReader から継承)	データを System.IO.TextReader として取得します。
GetTimeSpan メソッド	指定されたカラムの値を TimeSpan オブジェクトとして返します。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

名前	説明
GetUInt16 メソッド	指定されたカラムの値を 16 ビット符号なし整数として返します。
GetUInt32 メソッド	指定されたカラムの値を 32 ビット符号なし整数として返します。
GetUInt64 メソッド	指定されたカラムの値を 64 ビット符号なし整数として返します。
GetValue メソッド	指定されたカラムの値を Object として返します。
GetValues メソッド	現在のローのすべてのカラムを取得します。
IsDBNull メソッド	カラムに NULL 値が含まれるかどうかを示す値を返します。
IsDBNullAsync method (System.Data.Common.DbDataReader から継承)	カラムに存在のないまたは見つからない値が含まれているかどうかを示す値を取得する、System.Data.Common.DbDataReader.IsDBNull(System.Int32) の非同期バージョン。
myDispose メソッド	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。
NextResult メソッド	複数の結果セットを返すクエリを処理するとき、SADataReader を次の結果セットに進めます。
NextResultAsync method (System.Data.Common.DbDataReader から継承)	CancellationToken.None で statements.Invokes System.Data.Common.DbDataReader.NextResultAsync(System.Threading.CancellationToken) のバッチの結果を読むとき、読者を次の結果に進める System.Data.Common.DbDataReader.NextResult の非同期バージョン。
Read メソッド	結果セットの次のローを読み込み、SADataReader をこのローに移動します。
ReadAsync method (System.Data.Common.DbDataReader から継承)	読者を結果セットの次のレコードに進める、System.Data.Common.DbDataReader.Read の非同期バージョン。
Depth プロパティ	現在のローのネストの深さを示す値を取得します。

名前	説明
FieldCount プロパティ	結果セット内のカラム数を取得します。
HasRows プロパティ	SADataReader に 1 つ以上のローが含まれるかどうかを示す値を取得します。
IsClosed プロパティ	SADataReader が閉じているかどうかを示す値を取得します。
RecordsAffected プロパティ	SQL 文の実行によって変更、挿入、または削除されたローの数。
this プロパティ	カラムの値をネイティブフォーマットで返します。
VisibleFieldCount property (System.Data.Common.DbDataReader から継承)	System.Data.Common.DbDataReader の非表示でないフィールドの数を取得します。

備考

SADataReader にはコンストラクタがありません。SADataReader オブジェクトを取得するには、SACommand を実行します。

```
SACommand cmd = new SACommand(
   "SELECT EmployeeID FROM Employees", conn );
SADataReader reader = cmd.ExecuteReader();
```

SADataReader では前方へのみ移動できます。結果を操作するためにより柔軟なオブジェクトが必要な場合は、SADataAdapter を使用します。

SADataReader は必要に応じてローを取得しますが、SADataAdapter の場合、結果セットのすべてのローを取得しないと、オブジェクトに対してアクションを実行できません。結果セットのサイズが大きい場合、この違いのために SADataReader の方が応答時間が速くなります。

Implements: IDataReader、IDisposable、IDataRecord、IListSource

詳細については、「データへのアクセスとデータの操作」47 ページを参照してください。

参照

● SACommand.ExecuteReader メソッド [SQL Anywhere .NET]146 ページ

Close メソッド

SADataReader を閉じます。

Visual Basic の構文

```
Public Overrides Sub Close()
```

C# の構文

```
public override void Close()
```

備考

SADataReader を使用し終わったら、Close メソッドを明示的に呼び出してください。

オートコミットモードで実行している場合、SADataReader を閉じる関連動作として、COMMIT が発行されます。

GetBoolean メソッド

指定されたカラムの値をブール値として返します。

Visual Basic の構文

```
Public Overrides Function GetBoolean(
    ByVal ordinal As Integer
) As Boolean
```

C# の構文

```
public override bool GetBoolean(int ordinal)
```

パラメータ

- **ordinal**　値の取得元のカラムを示す順序数。番号は 0 から始まります。

戻り値

カラムの値。

備考

変換は行われないため、取り出されるデータはすでにブール値である必要があります。

参照

- SADataReader.GetOrdinal メソッド [SQL Anywhere .NET]239 ページ
- SADataReader.GetFieldType メソッド [SQL Anywhere .NET]236 ページ

GetByte メソッド

指定されたカラムの値をバイトとして返します。

Visual Basic の構文

```
Public Overrides Function GetByte(ByVal ordinal As Integer) As Byte
```

C# の構文

```
public override byte GetByte(int ordinal)
```

パラメータ

- **ordinal**　値の取得元のカラムを示す順序数。番号は 0 から始まります。

戻り値

カラムの値。

備考

変換は行われないため、取り出されるデータはすでにバイトである必要があります。

GetBytes メソッド

指定されたカラムオフセットからバイトのストリームを特定のバッファオフセットから始まる配列としてバッファに読み込みます。

Visual Basic の構文

```
Public Overrides Function GetBytes(
    ByVal ordinal As Integer,
    ByVal dataIndex As Long,
    ByVal buffer As Byte(),
    ByVal bufferIndex As Integer,
    ByVal length As Integer
) As Long
```

C# の構文

```
public override long GetBytes(
    int ordinal,
    long dataIndex,
    byte[] buffer,
    int bufferIndex,
    int length
)
```

パラメータ

- **ordinal**　値の取得元のカラムを示す順序数。番号は 0 から始まります。
- **dataIndex**　バイトの読み込み元のカラム値内のインデックス。
- **buffer**　データを格納する配列。
- **bufferIndex**　データのコピーを開始する配列内のインデックス。
- **length**　指定されたバッファにコピーするデータの最大長。

戻り値

読み込まれたバイト数。

備考

GetBytes は、フィールド内で使用可能なバイト数を返します。ほとんどの場合、これは正確なフィールド長です。ただし、GetBytes を使用してフィールドからバイトがすでに取得されている

場合、返される数値が実際の長さより小さくなる可能性があります。これはたとえば、SADataReader がサイズの大きいデータ構造体をバッファに読み込む場合などです。

NULL 参照 (Visual Basic の場合は Nothing) であるバッファを渡すと、GetBytes はフィールドの長さをバイト数で返します。

変換は行われないため、取り出されるデータはすでにバイト配列である必要があります。

GetChar メソッド

指定されたカラムの値を文字として返します。

Visual Basic の構文

```
Public Overrides Function GetChar(ByVal ordinal As Integer) As Char
```

C# の構文

```
public override char GetChar(int ordinal)
```

パラメータ

- **ordinal**　値の取得元のカラムを示す順序数。番号は 0 から始まります。

戻り値

カラムの値。

備考

変換は行われないため、取り出されるデータはすでに文字である必要があります。

SADataReader.IsDBNull メソッドを呼び出して NULL 値を確認してから、このメソッドを呼び出します。

参照

- SADataReader.IsDBNull メソッド [SQL Anywhere .NET]247 ページ

GetChars メソッド

指定されたカラムオフセットから文字のストリームを特定のバッファオフセットから始まる配列としてバッファに読み込みます。

Visual Basic の構文

```
Public Overrides Function GetChars(
    ByVal ordinal As Integer,
    ByVal dataIndex As Long,
    ByVal buffer As Char(),
    ByVal bufferIndex As Integer,
    ByVal length As Integer
) As Long
```

C# の構文

```
public override long GetChars(
    int ordinal,
    long dataIndex,
    char[] buffer,
    int bufferIndex,
    int length
)
```

パラメータ

- **ordinal**　0 から始まるカラムの順序。
- **dataIndex**　読み込みオペレーションを開始するロー内のインデックス。
- **buffer**　データのコピー先のバッファ。
- **bufferIndex**　読み込みオペレーションを開始するバッファのインデックス。
- **length**　読み込まれる文字数。

戻り値

実際に読み込まれた文字数。

備考

GetChars は、フィールド内で使用可能な文字数を返します。ほとんどの場合、これは正確なフィールド長です。ただし、GetChars を使用してフィールドから文字がすでに取得されている場合、返される数値が実際の長さより小さくなる可能性があります。これはたとえば、SADataReader がサイズの大きいデータ構造体をバッファに読み込む場合などです。

NULL 参照 (Visual Basic の場合は Nothing) であるバッファを渡すと、GetChars はフィールドの長さを文字数として返します。

変換は行われないため、取り出されるデータはすでに文字配列である必要があります。

BLOB の処理については、「BLOB」61 ページを参照してください。

GetData メソッド

このメソッドはサポートされていません。

Visual Basic の構文

```
Public Shadows Function GetData(ByVal i As Integer) As IDataReader
```

C# の構文

```
public new IDataReader GetData(int i)
```

備考

このメソッドを呼び出すと、InvalidOperationException がスローされます。

参照

- System.InvalidOperationException

GetDataTypeName メソッド

ソースデータ型の名前を返します。

Visual Basic の構文

```
Public Overrides Function GetDataTypeName(
    ByVal index As Integer
) As String
```

C# の構文

```
public override string GetDataTypeName(int index)
```

パラメータ

- **index**　0 から始まるカラムの順序。

戻り値

バックエンドデータ型の名前。

GetDateTime メソッド

指定されたカラムの値を DateTime オブジェクトとして返します。

Visual Basic の構文

```
Public Overrides Function GetDateTime(ByVal ordinal As Integer) As Date
```

C# の構文

```
public override DateTime GetDateTime(int ordinal)
```

パラメータ

- **ordinal**　0 から始まるカラムの順序。

戻り値

指定されたカラムの値。

備考

変換は行われないため、取り出されるデータはすでに DateTime オブジェクトである必要があります。

SADataReader.IsDBNull メソッドを呼び出して NULL 値を確認してから、このメソッドを呼び出します。

参照

- SADataReader.IsDBNull メソッド [SQL Anywhere .NET]247 ページ

GetDateTimeOffset メソッド

指定されたカラムの値を DateTimeOffset オブジェクトとして返します。

Visual Basic の構文

```
Public Function GetDateTimeOffset(
    ByVal ordinal As Integer
) As DateTimeOffset
```

C# の構文

```
public DateTimeOffset GetDateTimeOffset(int ordinal)
```

パラメータ

- **ordinal**　0 から始まるカラムの順序。

戻り値

指定されたカラムの値。

備考

変換は行われないため、取り出されるデータはすでに DateTimeOffset オブジェクトである必要があります。

SADataReader.IsDBNull メソッドを呼び出して NULL 値を確認してから、このメソッドを呼び出します。

参照

- SADataReader.IsDBNull メソッド [SQL Anywhere .NET]247 ページ

GetDecimal メソッド

指定されたカラムの値を Decimal オブジェクトとして返します。

Visual Basic の構文

```
Public Overrides Function GetDecimal(
    ByVal ordinal As Integer
) As Decimal
```

C# の構文

```
public override decimal GetDecimal(int ordinal)
```

パラメータ

- **ordinal**　値の取得元のカラムを示す順序数。番号は 0 から始まります。

戻り値

指定されたカラムの値。

備考

変換は行われないため、取り出されるデータはすでに Decimal オブジェクトである必要があります。

SADataReader.IsDBNull メソッドを呼び出して NULL 値を確認してから、このメソッドを呼び出します。

参照

- SADataReader.IsDBNull メソッド [SQL Anywhere .NET]247 ページ

GetDouble メソッド

指定されたカラムの値を倍精度の浮動小数点数として返します。

Visual Basic の構文

```
Public Overrides Function GetDouble(ByVal ordinal As Integer) As Double
```

C# の構文

```
public override double GetDouble(int ordinal)
```

パラメータ

- **ordinal**　値の取得元のカラムを示す順序数。番号は 0 から始まります。

戻り値

指定されたカラムの値。

備考

変換は行われないため、取り出されるデータはすでに倍精度の浮動小数点数である必要があります。

SADataReader.IsDBNull メソッドを呼び出して NULL 値を確認してから、このメソッドを呼び出します。

参照

- SADataReader.IsDBNull メソッド [SQL Anywhere .NET]247 ページ

GetEnumerator メソッド

SADataReader オブジェクトで反復処理する System.Collections.IEnumerator を返します。

Visual Basic の構文

```
Public Overrides Function GetEnumerator()
    As System.Collections.IEnumerator
```

C# の構文

```
public override IEnumerator GetEnumerator()
```

戻り値

SADataReader オブジェクトの System.Collections.IEnumerator。

参照

- SADataReader クラス [SQL Anywhere .NET]224 ページ

GetFieldType メソッド

オブジェクトのデータ型である Type を返します。

Visual Basic の構文

```
Public Overrides Function GetFieldType(ByVal index As Integer) As Type
```

C# の構文

```
public override Type GetFieldType(int index)
```

パラメータ

- **index**　0 から始まるカラムの順序。

戻り値

オブジェクトのデータ型である Type。

GetFloat メソッド

指定されたカラムの値を単精度の浮動小数点数として返します。

Visual Basic の構文

```
Public Overrides Function GetFloat(ByVal ordinal As Integer) As Single
```

C# の構文

```
public override float GetFloat(int ordinal)
```

パラメータ

- **ordinal**　値の取得元のカラムを示す順序数。番号は 0 から始まります。

戻り値

指定されたカラムの値。

備考

変換は行われないため、取り出されるデータはすでに単精度の浮動小数点数である必要があります。

SADataReader.IsDBNull メソッドを呼び出して NULL 値を確認してから、このメソッドを呼び出します。

参照

- SADataReader.IsDBNull メソッド [SQL Anywhere .NET]247 ページ

GetGuid メソッド

指定されたカラムの値をグローバル一意識別子 (GUID) として返します。

Visual Basic の構文

```
Public Overrides Function GetGuid(ByVal ordinal As Integer) As Guid
```

C# の構文

```
public override Guid GetGuid(int ordinal)
```

パラメータ

- **ordinal**　値の取得元のカラムを示す順序数。番号は 0 から始まります。

戻り値

指定されたカラムの値。

備考

取り出されるデータは、すでにグローバル一意識別子またはバイナリ (16) である必要があります。

SADataReader.IsDBNull メソッドを呼び出して NULL 値を確認してから、このメソッドを呼び出します。

参照

- SADataReader.IsDBNull メソッド [SQL Anywhere .NET]247 ページ

GetInt16 メソッド

指定されたカラムの値を 16 ビット符号付き整数として返します。

Visual Basic の構文

```
Public Overrides Function GetInt16(ByVal ordinal As Integer) As Short
```

C# の構文

```
public override short GetInt16(int ordinal)
```

パラメータ

- **ordinal**　値の取得元のカラムを示す順序数。番号は 0 から始まります。

戻り値

指定されたカラムの値。

備考

変換は行われないため、取り出されるデータはすでに 16 ビット符号付き整数である必要があります。

GetInt32 メソッド

指定されたカラムの値を 32 ビット符号付き整数として返します。

Visual Basic の構文

```
Public Overrides Function GetInt32(ByVal ordinal As Integer) As Integer
```

C# の構文

```
public override int GetInt32(int ordinal)
```

パラメータ

- **ordinal**　値の取得元のカラムを示す順序数。番号は 0 から始まります。

戻り値

指定されたカラムの値。

備考

変換は行われないため、取り出されるデータはすでに 32 ビット符号付き整数である必要があります。

GetInt64 メソッド

指定されたカラムの値を 64 ビット符号付き整数として返します。

Visual Basic の構文

```
Public Overrides Function GetInt64(ByVal ordinal As Integer) As Long
```

C# の構文

```
public override long GetInt64(int ordinal)
```

パラメータ

- **ordinal**　値の取得元のカラムを示す順序数。番号は 0 から始まります。

戻り値

指定されたカラムの値。

備考

変換は行われないため、取り出されるデータはすでに 64 ビット符号付き整数である必要があります。

GetName メソッド

指定されたカラムの名前を返します。

Visual Basic の構文

```
Public Overrides Function GetName(ByVal index As Integer) As String
```

C# の構文

```
public override string GetName(int index)
```

パラメータ

- **index**　カラムの 0 から始まるインデックス。

戻り値

指定されたカラムの名前。

GetOrdinal メソッド

指定されたカラム名に対応するカラムの順序を返します。

Visual Basic の構文

```
Public Overrides Function GetOrdinal(ByVal name As String) As Integer
```

C# の構文

```
public override int GetOrdinal(string name)
```

パラメータ

- **name**　カラムの名前。

戻り値

0 から始まるカラムの順序。

備考

GetOrdinal は、最初に大文字と小文字を区別したルックアップを実行します。このルックアップが失敗した場合、2 回目は大文字と小文字を区別しないでルックアップを実行します。

GetOrdinal は、日本語のかな幅を区別しません。

順序ベースのルックアップの方が名前ベースのルックアップより効率的であるため、ループ内で GetOrdinal を呼び出すのは非効率です。GetOrdinal を 1 回呼び出し、結果を整数変数に割り当ててループ内で使用すると、時間を節約できます。

GetSchemaTable メソッド

SADataReader のカラムメタデータが記述された DataTable を返します。

Visual Basic の構文

```
Public Overrides Function GetSchemaTable() As DataTable
```

C# の構文

```
public override DataTable GetSchemaTable()
```

戻り値

カラムメタデータが記述された DataTable。

備考

このメソッドは、各カラムに関するメタデータを次の順で返します。

DataTable カラム	説明
ColumnName	カラムの名前。カラムに名前がない場合は NULL 参照 (Visual Basic の Nothing)。SQL クエリでカラムのエイリアスが使用されている場合は、そのエイリアスが返されます。結果セットでは、すべてのカラムに名前があるとは限らないほか、すべてのカラム名がユニークであるとは限りません。
ColumnOrdinal	カラムの ID。値の範囲は、[0, FieldCount -1] です。
ColumnSize	サイズ指定されたカラムの場合、カラムの値の最大長。その他のカラムの場合、これは、そのデータ型のバイト単位のサイズです。
NumericPrecision	数値カラムの精度。カラムが数値型ではない場合は DBNull。
NumericScale	数値カラムの位取り。カラムが数値型ではない場合は DBNull。
IsUnique	カラムが取得元のテーブル (BaseTableName) でユニークな非計算カラムである場合は true。

DataTable カラム	説明
IsKey	カラムが、結果セットのユニークキーからともに取得された結果セットの一連のカラムのいずれかである場合は true。IsKey が true に設定されたカラムのセットは、結果セット内のローをユニークに識別する最小限のセットである必要はありません。
BaseServerName	SADataReader が使用する SQL Anywhere データベースサーバの名前。
BaseCatalogName	カラムが含まれているデータベース内のカタログの名前。値は常に DBNull です。
BaseColumnName	データベースのテーブル BaseTableName にあるカラムの元の名前。カラムが計算される場合、およびこの情報を特定できない場合は DBNull です。
BaseSchemaName	カラムが含まれているデータベース内のスキーマの名前。
BaseTableName	カラムが含まれているデータベースのテーブルの名前。カラムが計算される場合、およびこの情報を特定できない場合は DBNull です。
DataType	この型のカラムに最適な .NET データ型。
AllowDBNull	カラムが NULL 入力可である場合は true、NULL 入力不可である場合またはこの情報を特定できない場合は false。
ProviderType	カラム型
IsAliased	カラム名がエイリアスの場合は true、エイリアスでない場合は false。
IsExpression	カラムが式の場合は true、カラム値の場合は false。
IsIdentity	カラムが identity カラムである場合は true、identity カラムでない場合は false。

DataTable カラム	説明
IsAutoIncrement	カラムがオートインクリメントカラムまたはグローバルオートインクリメントカラムである場合は true、それ以外のカラムである場合 (または、この情報を特定できない場合) は false。
IsRowVersion	書き込みできない永続的なロー識別子がカラムに含まれており、ローを識別する以外は意味を持たない値がカラムにある場合は true。
IsHidden	カラムが非表示の場合は true、表示される場合は false。
IsLong	カラムが long varchar、long nvarchar、または long binary の場合は true、それ以外の場合は false。
IsReadOnly	カラムが読み込み専用である場合は true、カラムが修正可能であるか、カラムのアクセス権を特定できない場合は false。

これらのカラムの詳細については、.NET Framework のマニュアルの SqlDataReader.GetSchemaTable を参照してください。

詳細については、「SACommand：GetSchemaTable を使用した結果セットのスキーマのフェッチ」50 ページを参照してください。

GetString メソッド

指定されたカラムの値を文字列として返します。

Visual Basic の構文

```
Public Overrides Function GetString(ByVal ordinal As Integer) As String
```

C# の構文

```
public override string GetString(int ordinal)
```

パラメータ

- **ordinal**　値の取得元のカラムを示す順序数。番号は 0 から始まります。

戻り値

指定されたカラムの値。

備考

変換は行われないため、取り出されるデータはすでに文字列である必要があります。

SADataReader.IsDBNull メソッドを呼び出して NULL 値を確認してから、このメソッドを呼び出します。

参照

- SADataReader.IsDBNull メソッド [SQL Anywhere .NET]247 ページ

GetTimeSpan メソッド

指定されたカラムの値を TimeSpan オブジェクトとして返します。

Visual Basic の構文

```
Public Function GetTimeSpan(ByVal ordinal As Integer) As TimeSpan
```

C# の構文

```
public TimeSpan GetTimeSpan(int ordinal)
```

パラメータ

- **ordinal**　値の取得元のカラムを示す順序数。番号は 0 から始まります。

戻り値

指定されたカラムの値。

備考

カラムは、SQL Anywhere TIME データ型である必要があります。データは、TimeSpan に変換されます。TimeSpan の Days プロパティは常に 0 に設定されます。

SADataReader.IsDBNull メソッドを呼び出して NULL 値を確認してから、このメソッドを呼び出します。

詳細については、「時間値」62 ページを参照してください。

参照

- SADataReader.IsDBNull メソッド [SQL Anywhere .NET]247 ページ

GetUInt16 メソッド

指定されたカラムの値を 16 ビット符号なし整数として返します。

Visual Basic の構文

```
Public Function GetUInt16(ByVal ordinal As Integer) As UShort
```

C# の構文

```
public ushort GetUInt16(int ordinal)
```

パラメータ

- **ordinal**　値の取得元のカラムを示す順序数。番号は 0 から始まります。

戻り値

指定されたカラムの値。

備考

変換は行われないため、取り出されるデータはすでに 16 ビット符号なし整数である必要があります。

GetUInt32 メソッド

指定されたカラムの値を 32 ビット符号なし整数として返します。

Visual Basic の構文

```
Public Function GetUInt32(ByVal ordinal As Integer) As UInteger
```

C# の構文

```
public uint GetUInt32(int ordinal)
```

パラメータ

- **ordinal**　値の取得元のカラムを示す順序数。番号は 0 から始まります。

戻り値

指定されたカラムの値。

備考

変換は行われないため、取り出されるデータはすでに 32 ビット符号なし整数である必要があります。

GetUInt64 メソッド

指定されたカラムの値を 64 ビット符号なし整数として返します。

Visual Basic の構文

```
Public Function GetUInt64(ByVal ordinal As Integer) As ULong
```

C# の構文

```
public ulong GetUInt64(int ordinal)
```

パラメータ

- **ordinal**　値の取得元のカラムを示す順序数。番号は 0 から始まります。

戻り値

指定されたカラムの値。

備考

変換は行われないため、取り出されるデータはすでに 64 ビット符号なし整数である必要があります。

GetValue メソッド

指定されたカラムの値を Object として返します。

オーバロードリスト

名前	説明
GetValue(int) メソッド	指定されたカラムの値を Object として返します。
GetValue(int, long, int) メソッド	指定したカラムの値の部分文字列を Object として返します。

GetValue(int) メソッド

指定されたカラムの値を Object として返します。

Visual Basic の構文

```
Public Overrides Function GetValue(ByVal ordinal As Integer) As Object
```

C# の構文

```
public override object GetValue(int ordinal)
```

パラメータ

- **ordinal**　値の取得元のカラムを示す順序数。番号は 0 から始まります。

戻り値

オブジェクトとして返される指定されたカラムの値。

備考

このメソッドは、NULL データベースカラムに対して DBNull を返します。

GetValue(int, long, int) メソッド

指定したカラムの値の部分文字列を Object として返します。

Visual Basic の構文

```
Public Function GetValue(
    ByVal ordinal As Integer,
    ByVal index As Long,
    ByVal length As Integer
) As Object
```

C# の構文

```
public object GetValue(int ordinal, long index, int length)
```

パラメータ

- **ordinal**　値の取得元のカラムを示す順序数。番号は 0 から始まります。
- **index**　取得される値の部分文字列の、0 から始まるインデックス。
- **length**　取得される値の部分文字列の長さ。

戻り値

部分文字列の値はオブジェクトとして返されます。

備考

このメソッドは、NULL データベースカラムに対して DBNull を返します。

GetValues メソッド

現在のローのすべてのカラムを取得します。

Visual Basic の構文

```
Public Overrides Function GetValues(ByVal values As Object()) As Integer
```

C# の構文

```
public override int GetValues(object[] values)
```

パラメータ

- **values**　結果セットのロー全体を保持するオブジェクトの配列。

戻り値

配列内のオブジェクトの数。

備考

ほとんどのアプリケーションについて、GetValues メソッドは、各カラムを個々に取り出すのではなく、すべてのカラムを取り出す効率的な方法を提供します。

結果のローに含まれるカラムの数より少ないカラムが含まれる Object 配列を渡すことができます。Object 配列が保持するデータ量のみが配列にコピーされます。また、結果のローに含まれるカラムの数より長い Object 配列を渡すこともできます。

このメソッドは、NULL データベースカラムに対して DBNull を返します。

IsDBNull メソッド

カラムに NULL 値が含まれるかどうかを示す値を返します。

Visual Basic の構文

```
Public Overrides Function IsDBNull(ByVal ordinal As Integer) As Boolean
```

C# の構文

```
public override bool IsDBNull(int ordinal)
```

パラメータ

- **ordinal**　0 から始まるカラムの順序。

戻り値

指定されたカラム値が DBNull と等しい場合は true を返します。ない場合、false を返します。

備考

入力された取得メソッド (GetByte、GetChar など) を呼び出す前に、このメソッドを呼び出して NULL カラム値を確認し、例外が発生しないようにします。

myDispose メソッド

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。

Visual Basic の構文

```
Public Sub myDispose()
```

C# の構文

```
public void myDispose()
```

NextResult メソッド

複数の結果セットを返すクエリを処理するとき、SADataReader を次の結果セットに進めます。

Visual Basic の構文

```
Public Overrides Function NextResult() As Boolean
```

C# の構文

```
public override bool NextResult()
```

戻り値

さらに結果セットがある場合は true を返します。ない場合、false を返します。

備考

バッチ SQL 文またはストアドプロシージャを実行して生成できる複数の結果セットを処理するために使用されます。

デフォルトでは、データリーダは最初の結果セットの位置にあります。

Read メソッド

結果セットの次のローを読み込み、SADataReader をこのローに移動します。

Visual Basic の構文

```
Public Overrides Function Read() As Boolean
```

C# の構文

```
public override bool Read()
```

戻り値

さらにローがある場合は true を返します。ない場合、false を返します。

備考

SADataReader のデフォルト位置は、最初のレコードより前です。このため、任意のデータにアクセスするには Read を呼び出す必要があります。

例

次のコードは、結果の単一カラムの値をリストボックスに設定します。

```
while( reader.Read() ) {
   listResults.Items.Add( reader.GetValue( 0 ).ToString() );
}
listResults.EndUpdate();
reader.Close();
```

Depth プロパティ

現在のローのネストの深さを示す値を取得します。

Visual Basic の構文

```
Public ReadOnly Overrides Property Depth As Integer
```

C# の構文

```
public override int Depth {get;}
```

 Copyright © 2013, SAP AG or an SAP affiliate company.

備考

最も外側のテーブルの深さは 0 です。

現在のローのネストの深さ。

FieldCount プロパティ

結果セット内のカラム数を取得します。

Visual Basic の構文

```
Public ReadOnly Overrides Property FieldCount As Integer
```

C# の構文

```
public override int FieldCount {get;}
```

備考

現在のレコード内のカラム数。

HasRows プロパティ

SADataReader に 1 つ以上のローが含まれるかどうかを示す値を取得します。

Visual Basic の構文

```
Public ReadOnly Overrides Property HasRows As Boolean
```

C# の構文

```
public override bool HasRows {get;}
```

備考

SADataReader に 1 つ以上のローが含まれる場合は true、含まれない場合は false です。

IsClosed プロパティ

SADataReader が閉じているかどうかを示す値を取得します。

Visual Basic の構文

```
Public ReadOnly Overrides Property IsClosed As Boolean
```

C# の構文

```
public override bool IsClosed {get;}
```

備考

SADataReader が閉じられている場合は true、閉じられていない場合は false です。

SADataReader が閉じられた後に呼び出すことができるプロパティは、IsClosed と RecordsAffected のみです。

RecordsAffected プロパティ

SQL 文の実行によって変更、挿入、または削除されたローの数。

Visual Basic の構文

```
Public ReadOnly Overrides Property RecordsAffected As Integer
```

C# の構文

```
public override int RecordsAffected {get;}
```

備考

変更、挿入、または削除されたローの数。この値は、ローが影響されなかったり文が失敗した場合は 0、SELECT 文の場合は -1 です。

変更、挿入、または削除されたローの数。この値は、ローが影響されなかったり文が失敗した場合は 0、SELECT 文の場合は -1 です。

このプロパティの値は累積されます。たとえば、2 つのレコードがバッチモードで挿入された場合、RecordsAffected の値は 2 になります。

SADataReader が閉じられた後に呼び出すことができるプロパティは、IsClosed と RecordsAffected のみです。

this プロパティ

カラムの値をネイティブフォーマットで返します。

オーバロードリスト

名前	説明
this[int] プロパティ	カラムの値をネイティブフォーマットで返します。
this[string] プロパティ	カラムの値をネイティブフォーマットで返します。

this[int] プロパティ

カラムの値をネイティブフォーマットで返します。

Visual Basic の構文

```
Public ReadOnly Overrides Property Item(
    ByVal index As Integer
) As Object
```

C# の構文

```
public override object this[int index] {get;}
```

パラメータ

- **index**　カラムの順序。

備考

C# では、このプロパティは SADataReader クラスのインデクサです。

this[string] プロパティ

カラムの値をネイティブフォーマットで返します。

Visual Basic の構文

```
Public ReadOnly Overrides Property Item(ByVal name As String) As Object
```

C# の構文

```
public override object this[string name] {get;}
```

パラメータ

- **name**　カラムの名前。

備考

C# では、このプロパティは SADataReader クラスのインデクサです。

SADataSourceEnumerator クラス

ローカルネットワーク内で有効な SQL Anywhere データベースサーバのすべてのインスタンスを列挙するメカニズムを提供します。

Visual Basic の構文

```
Public NotInheritable Class SADataSourceEnumerator
    Inherits System.Data.Common.DbDataSourceEnumerator
```

C# の構文

```
public sealed class SADataSourceEnumerator :
    System.Data.Common.DbDataSourceEnumerator
```

基本クラス

- System.Data.Common.DbDataSourceEnumerator

メンバー

継承されたメンバーを含む SADataSourceEnumerator クラスのすべてのメンバー。

名前	説明
GetDataSources メソッド	参照可能なすべての SQL Anywhere データベースサーバに関する情報が含まれる DataTable を取得します。
Instance プロパティ	SADataSourceEnumerator のインスタンスを取得します。これを使用すると、すべての表示可能な SQL Anywhere データベースサーバを取得できます。

備考

SADataSourceEnumerator にはコンストラクタがありません。

SADataSourceEnumerator クラスは、.NET Compact Framework 2.0 では使用できません。

GetDataSources メソッド

参照可能なすべての SQL Anywhere データベースサーバに関する情報が含まれる DataTable を取得します。

Visual Basic の構文

```
Public Overrides Function GetDataSources() As DataTable
```

C# の構文

```
public override DataTable GetDataSources()
```

備考

返されるテーブルは、ServerName、IPAddress、PortNumber、DataBaseNames という 4 つのカラムで構成されます。テーブルには、有効なデータベースサーバごとに 1 つのローがあります。

例

次のコードは、有効な各データベースサーバに関する情報を DataTable に設定します。

```
DataTable servers = SADataSourceEnumerator.Instance.GetDataSources();
```

Instance プロパティ

SADataSourceEnumerator のインスタンスを取得します。これを使用すると、すべての表示可能な SQL Anywhere データベースサーバを取得できます。

Visual Basic の構文

```
Public Shared ReadOnly Property Instance As SADataSourceEnumerator
```

C# の構文

```
public SADataSourceEnumerator Instance {get;}
```

SADefault クラス

デフォルト値が設定されたパラメータを表します。

Visual Basic の構文

```
Public NotInheritable Class SADefault
```

C# の構文

```
public sealed class SADefault
```

メンバー

継承されたメンバーを含む SADefault クラスのすべてのメンバー。

名前	説明
Value フィールド	デフォルトパラメータの値を取得します。

備考

SADefault にはコンストラクタがありません。

```
SAParameter parm = new SAParameter();
parm.Value = SADefault.Value;
```

Value フィールド

デフォルトパラメータの値を取得します。

Visual Basic の構文

```
Public Shared ReadOnly Value As SADefault
```

C# の構文

```
public static readonly SADefault Value;
```

備考

このフィールドは読み込み専用で静的です。

SAError クラス

データソースによって返された警告またはエラーに関する情報を収集します。

Visual Basic の構文

```
Public NotInheritable Class SAError
```

C# の構文

```
public sealed class SAError
```

メンバー

継承されたメンバーを含む SAError クラスのすべてのメンバー。

名前	説明
ToString メソッド	エラーメッセージの完全なテキストです。
Message プロパティ	エラーの簡単な説明を返します。
NativeError プロパティ	データベース固有のエラー情報を返します。
Source プロパティ	エラーを生成したプロバイダの名前を返します。
SqlState プロパティ	ANSI SQL 規格に準拠する SQL Anywhere の 5 文字の SQLSTATE です。

備考

SAError にはコンストラクタがありません。

エラー処理の詳細については、「エラー処理」65 ページを参照してください。

ToString メソッド

エラーメッセージの完全なテキストです。

Visual Basic の構文

```
Public Overrides Function ToString() As String
```

C# の構文

```
public override string ToString()
```

例

戻り値は、**SAError:** の形式の文字列で、後ろにメッセージが続きます。次に例を示します。

```
SAError:UserId or Password not valid.
```

Message プロパティ

エラーの簡単な説明を返します。

Visual Basic の構文

```
Public ReadOnly Property Message As String
```

C# の構文

```
public string Message {get;}
```

NativeError プロパティ

データベース固有のエラー情報を返します。

Visual Basic の構文

```
Public ReadOnly Property NativeError As Integer
```

C# の構文

```
public int NativeError {get;}
```

Source プロパティ

エラーを生成したプロバイダの名前を返します。

Visual Basic の構文

```
Public ReadOnly Property Source As String
```

C# の構文

```
public string Source {get;}
```

SqlState プロパティ

ANSI SQL 規格に準拠する SQL Anywhere の 5 文字の SQLSTATE です。

Visual Basic の構文

```
Public ReadOnly Property SqlState As String
```

C# の構文

```
public string SqlState {get;}
```

SAErrorCollection クラス

SQL Anywhere .NET データプロバイダによって生成されたすべてのエラーを収集します。

Visual Basic の構文

```
Public NotInheritable Class SAErrorCollection
    Implements System.Collections.ICollection
    Implements System.Collections.IEnumerable
```

C# の構文

```
public sealed class SAErrorCollection :
    System.Collections.ICollection,
    System.Collections.IEnumerable
```

基本クラス

- System.Collections.ICollection
- System.Collections.IEnumerable

メンバー

継承されたメンバーを含む SAErrorCollection クラスのすべてのメンバー。

名前	説明
CopyTo メソッド	配列内の特定のインデックスから開始して SAErrorCollection の要素を配列にコピーします。
GetEnumerator メソッド	SAErrorCollection で反復処理する列挙子を返します。
Count プロパティ	コレクション内のエラーの数を返します。
this プロパティ	指定されたインデックス位置のエラーを返します。

備考

SAErrorCollection にはコンストラクタがありません。通常、SAErrorCollection は SAException.Errors プロパティから取得されます。

Implements: ICollection、IEnumerable

エラー処理の詳細については、「エラー処理」65 ページを参照してください。

参照

- SAException.Errors プロパティ [SQL Anywhere .NET]260 ページ

CopyTo メソッド

配列内の特定のインデックスから開始して SAErrorCollection の要素を配列にコピーします。

Visual Basic の構文

```
Public Sub CopyTo(ByVal array As Array, ByVal index As Integer)
```

C# の構文

```
public void CopyTo(Array array, int index)
```

パラメータ

- **array**　要素のコピー先の配列。
- **index**　配列の開始インデックス。

GetEnumerator メソッド

SAErrorCollection で反復処理する列挙子を返します。

Visual Basic の構文

```
Public Function GetEnumerator() As System.Collections.IEnumerator
```

C# の構文

```
public System.Collections.IEnumerator GetEnumerator()
```

戻り値

SAErrorCollection の System.Collections.IEnumerator。

Count プロパティ

コレクション内のエラーの数を返します。

Visual Basic の構文

```
Public ReadOnly Property Count As Integer
```

C# の構文

```
public int Count {get;}
```

this プロパティ

指定されたインデックス位置のエラーを返します。

Visual Basic の構文

```
Public ReadOnly Property Item(ByVal index As Integer) As SAError
```

C# の構文

```
public SAError this[int index] {get;}
```

パラメータ

- **index**　取り出すエラーの 0 から始まるインデックス。

備考

指定されたインデックス位置のエラーが含まれる SAError オブジェクト。

参照

- SAError クラス [SQL Anywhere .NET]253 ページ

SAException クラス

SQL Anywhere が警告またはエラーを返したときにスローされる例外です。

Visual Basic の構文

```
Public Class SAException Inherits System.Exception
```

C# の構文

```
public class SAException : System.Exception
```

基本クラス

- System.Exception

メンバー

継承されたメンバーを含む SAException クラスのすべてのメンバー。

名前	説明
GetBaseException method(System.Exception から継承)	派生クラスで上書きされる場合は、後続の 1 つ以上の例外の根本原因となる System.Exception を返します。
GetObjectData メソッド	SerializationInfo に例外に関する情報を設定します。
GetType method(System.Exception から継承)	現在のインスタンスのランタイムタイプを取得します。
ToString method(System.Exception から継承)	現在の例外を表す文字列を作成して返します。
Data property(System.Exception から継承)	例外に関するユーザ定義の追加情報を提供するキー／値のペアのコレクションを取得します。
Errors プロパティ	1 つまたは複数の SAError オブジェクトのコレクションを返します。
HelpLink property(System.Exception から継承)	この例外に関連付けられたヘルプファイルへのリンクを取得または設定します。
HResult property(System.Exception から継承)	特定の例外に割り当てられたコード化された数値 HRESULT を取得または設定します。

名前	説明
InnerException property(System.Exception から継承)	現在の例外を発生させた System.Exception インスタンスを取得します。
Message プロパティ	エラーが記述されたテキストを返します。
NativeError プロパティ	データベース固有のエラー情報を返します。
Source プロパティ	エラーを生成したプロバイダの名前を返します。
StackTrace property(System.Exception から継承)	呼び出しスタックで即時フレームの文字列表現を取得します。
TargetSite property(System.Exception から継承)	現在の例外をスローしたメソッドを取得します。

備考

SAException にはコンストラクタがありません。通常、SAException オブジェクトは catch 内で宣言されます。次に例を示します。

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

エラー処理の詳細については、「エラー処理」65 ページを参照してください。

GetObjectData メソッド

SerializationInfo に例外に関する情報を設定します。

Visual Basic の構文

```
Public Overrides Sub GetObjectData(
    ByVal info As SerializationInfo,
    ByVal context As StreamingContext
)
```

C# の構文

```
public override void GetObjectData(
    SerializationInfo info,
    StreamingContext context
)
```

パラメータ

- **info**　スローされた例外に関する直列化形式のオブジェクトデータを保持する SerializationInfo。

- **context**　ソースまたは送信先に関するコンテキスト情報が含まれる StreamingContext。

備考

Exception.GetObjectData を上書きします。

Errors プロパティ

1 つまたは複数の SAError オブジェクトのコレクションを返します。

Visual Basic の構文

```
Public ReadOnly Property Errors As SAErrorCollection
```

C# の構文

```
public SAErrorCollection Errors {get;}
```

備考

SAErrorCollection オブジェクトには常に少なくとも 1 つの SAError オブジェクトのインスタンスがあります。

参照

- SAErrorCollection クラス [SQL Anywhere .NET]255 ページ
- SAError クラス [SQL Anywhere .NET]253 ページ

Message プロパティ

エラーが記述されたテキストを返します。

Visual Basic の構文

```
Public ReadOnly Overrides Property Message As String
```

C# の構文

```
public override string Message {get;}
```

備考

このメソッドは、Errors コレクション内のすべての SAError オブジェクトのすべての Message プロパティの連結が含まれる単一文字列を返します。最後のメッセージを除く各メッセージの後ろには復帰文字があります。

参照

- SAError クラス [SQL Anywhere .NET]253 ページ

NativeError プロパティ

データベース固有のエラー情報を返します。

Visual Basic の構文

```
Public ReadOnly Property NativeError As Integer
```

C# の構文

```
public int NativeError {get;}
```

Source プロパティ

エラーを生成したプロバイダの名前を返します。

Visual Basic の構文

```
Public ReadOnly Overrides Property Source As String
```

C# の構文

```
public override string Source {get;}
```

SAFactory クラス

データソースクラスの iAnywhere.Data.SQLAnywhere プロバイダの実装のインスタンスを作成する、メソッドのセットを表します。

Visual Basic の構文

```
Public NotInheritable Class SAFactory
    Inherits System.Data.Common.DbProviderFactory
```

C# の構文

```
public sealed class SAFactory : System.Data.Common.DbProviderFactory
```

基本クラス

- System.Data.Common.DbProviderFactory

メンバー

継承されたメンバーを含む SAFactory クラスのすべてのメンバー。

名前	説明
CreateCommand メソッド	厳密に型指定された System.Data.Common.DbCommand インスタンスを返します。
CreateCommandBuilder メソッド	厳密に型指定された System.Data.Common.DbCommandBuilder インスタンスを返します。

名前	説明
CreateConnection メソッド	厳密に型指定された System.Data.Common.DbConnection インスタンスを返します。
CreateConnectionStringBuilder メソッド	厳密に型指定された System.Data.Common.DbConnectionStringBuilder インスタンスを返します。
CreateDataAdapter メソッド	厳密に型指定された System.Data.Common.DbDataAdapter インスタンスを返します。
CreateDataSourceEnumerator メソッド	厳密に型指定された System.Data.Common.DbDataSourceEnumerator インスタンスを返します。
CreateParameter メソッド	厳密に型指定された System.Data.Common.DbParameter インスタンスを返します。
CreatePermission メソッド	厳密に型指定された CodeAccessPermission インスタンスを返します。
CanCreateDataSourceEnumerator プロパティ	常に true を返します。これは、SADataSourceEnumerator オブジェクトを作成できることを示しています。
Instance フィールド	SAFactory クラスのシングルトンインスタンスを表します。

備考

SAFactory にはコンストラクタがありません。

ADO.NET 2.0 には DbProviderFactories および DbProviderFactory という 2 つのクラスが新しく追加され、プロバイダに依存しないコードを簡単に作成できるようになりました。これらを SQL Anywhere で使用するには、GetFactory に渡されるプロバイダの不変名として iAnywhere.Data.SQLAnywhere を指定します。次に例を示します。

```
' Visual Basic
Dim factory As DbProviderFactory = _
  DbProviderFactories.GetFactory( "iAnywhere.Data.SQLAnywhere" )
Dim conn As DbConnection = _
  factory.CreateConnection()

// C#
DbProviderFactory factory =
      DbProviderFactories.GetFactory("iAnywhere.Data.SQLAnywhere" );
DbConnection conn = factory.CreateConnection();
```

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

この例の中の conn は、SAConnection オブジェクトとして作成されます。

ADO.NET 2.0 におけるプロバイダファクトリと汎用プログラミングについては、Generic Coding with the ADO.NET 2.0 Base Classes and Factories を参照してください。

SAFactory クラスは、.NET Compact Framework 2.0 では使用できません。

CreateCommand メソッド

厳密に型指定された System.Data.Common.DbCommand インスタンスを返します。

Visual Basic の構文

```
Public Overrides Function CreateCommand() As DbCommand
```

C# の構文

```
public override DbCommand CreateCommand()
```

戻り値

DbCommand として型指定された新しい SACommand オブジェクト。

参照

- SACommand クラス [SQL Anywhere .NET]129 ページ

CreateCommandBuilder メソッド

厳密に型指定された System.Data.Common.DbCommandBuilder インスタンスを返します。

Visual Basic の構文

```
Public Overrides Function CreateCommandBuilder() As DbCommandBuilder
```

C# の構文

```
public override DbCommandBuilder CreateCommandBuilder()
```

戻り値

DbCommand として型指定された新しい SACommand オブジェクト。

参照

- SACommand クラス [SQL Anywhere .NET]129 ページ

CreateConnection メソッド

厳密に型指定された System.Data.Common.DbConnection インスタンスを返します。

Visual Basic の構文

```
Public Overrides Function CreateConnection() As DbConnection
```

C# の構文

```
public override DbConnection CreateConnection()
```

戻り値

DbCommand として型指定された新しい SACommand オブジェクト。

参照

- SACommand クラス [SQL Anywhere .NET]129 ページ

CreateConnectionStringBuilder メソッド

厳密に型指定された System.Data.Common.DbConnectionStringBuilder インスタンスを返します。

Visual Basic の構文

```
Public Overrides Function CreateConnectionStringBuilder()
    As DbConnectionStringBuilder
```

C# の構文

```
public override DbConnectionStringBuilder CreateConnectionStringBuilder()
```

戻り値

DbCommand として型指定された新しい SACommand オブジェクト。

参照

- SACommand クラス [SQL Anywhere .NET]129 ページ

CreateDataAdapter メソッド

厳密に型指定された System.Data.Common.DbDataAdapter インスタンスを返します。

Visual Basic の構文

```
Public Overrides Function CreateDataAdapter() As DbDataAdapter
```

C# の構文

```
public override DbDataAdapter CreateDataAdapter()
```

戻り値

DbCommand として型指定された新しい SACommand オブジェクト。

参照

- SACommand クラス [SQL Anywhere .NET]129 ページ

CreateDataSourceEnumerator メソッド

厳密に型指定された System.Data.Common.DbDataSourceEnumerator インスタンスを返します。

Visual Basic の構文

```
Public Overrides Function CreateDataSourceEnumerator()
    As DbDataSourceEnumerator
```

C# の構文

```
public override DbDataSourceEnumerator CreateDataSourceEnumerator()
```

戻り値

DbCommand として型指定された新しい SACommand オブジェクト。

参照

- SACommand クラス [SQL Anywhere .NET]129 ページ

CreateParameter メソッド

厳密に型指定された System.Data.Common.DbParameter インスタンスを返します。

Visual Basic の構文

```
Public Overrides Function CreateParameter() As DbParameter
```

C# の構文

```
public override DbParameter CreateParameter()
```

戻り値

DbCommand として型指定された新しい SACommand オブジェクト。

参照

- SACommand クラス [SQL Anywhere .NET]129 ページ

CreatePermission メソッド

厳密に型指定された CodeAccessPermission インスタンスを返します。

Visual Basic の構文

```
Public Overrides Function CreatePermission(
    ByVal state As PermissionState
) As CodeAccessPermission
```

C# の構文

```
public override CodeAccessPermission CreatePermission(
    PermissionState state
)
```

パラメータ

- **state** System.Security.Permissions.PermissionState 列挙のメンバー。

戻り値

DbCommand として型指定された新しい SACommand オブジェクト。

参照

- SACommand クラス [SQL Anywhere .NET]129 ページ

CanCreateDataSourceEnumerator プロパティ

常に true を返します。これは、SADataSourceEnumerator オブジェクトを作成できることを示しています。

Visual Basic の構文

```
Public ReadOnly Overrides Property CanCreateDataSourceEnumerator As
Boolean
```

C# の構文

```
public override bool CanCreateDataSourceEnumerator {get;}
```

戻り値

DbCommand として型指定された新しい SACommand オブジェクト。

参照

- SADataSourceEnumerator クラス [SQL Anywhere .NET]251 ページ
- SACommand クラス [SQL Anywhere .NET]129 ページ

Instance フィールド

SAFactory クラスのシングルトンインスタンスを表します。

Visual Basic の構文

```
Public Shared ReadOnly Instance As SAFactory
```

C# の構文

```
public static readonly SAFactory Instance;
```

備考

SAFactory はシングルトンクラスです。つまり、このクラスのインスタンスとして存在できるのは、このインスタンスのみです。

通常、このフィールドを直接使用することはありません。代わりに、SAFactory のこのインスタンスへの参照を取得するには、System.Data.Common.DbProviderFactories.GetFactory(String) を使用します。例については、SAFactory の説明を参照してください。

SAFactory クラスは、.NET Compact Framework 2.0 では使用できません。

参照

- SAFactory クラス [SQL Anywhere .NET]261 ページ

SAInfoMessageEventArgs クラス

InfoMessage イベントのデータを提供します。

Visual Basic の構文

```
Public NotInheritable Class SAInfoMessageEventArgs
    Inherits System.EventArgs
```

C# の構文

```
public sealed class SAInfoMessageEventArgs : System.EventArgs
```

基本クラス

- System.EventArgs

メンバー

継承されたメンバーを含む SAInfoMessageEventArgs クラスのすべてのメンバー。

名前	説明
ToString メソッド	InfoMessage イベントの文字列表現を取り出します。
Errors プロパティ	データソースから送信されたメッセージのコレクションを返します。
Message プロパティ	データソースから送信されたエラーの完全なテキストを返します。
MessageType プロパティ	メッセージのタイプを返します。
NativeError プロパティ	データベースによって返される SQLCODE を返します。
Source プロパティ	SQL Anywhere .NET データプロバイダの名前を返します。
Empty field (System.EventArgs から継承)	イベントデータのないイベントを表します。

備考

SAInfoMessageEventArgs にはコンストラクタがありません。

ToString メソッド

InfoMessage イベントの文字列表現を取り出します。

Visual Basic の構文

```
Public Overrides Function ToString() As String
```

C# の構文

```
public override string ToString()
```

戻り値

InfoMessage イベントを表す文字列。

Errors プロパティ

データソースから送信されたメッセージのコレクションを返します。

Visual Basic の構文

```
Public ReadOnly Property Errors As SAErrorCollection
```

C# の構文

```
public SAErrorCollection Errors {get;}
```

Message プロパティ

データソースから送信されたエラーの完全なテキストを返します。

Visual Basic の構文

```
Public ReadOnly Property Message As String
```

C# の構文

```
public string Message {get;}
```

MessageType プロパティ

メッセージのタイプを返します。

Visual Basic の構文

```
Public ReadOnly Property MessageType As SAMessageType
```

C# の構文

```
public SAMessageType MessageType {get;}
```

備考

次のいずれかを指定してください。Action、Info、Status、または Warning。

NativeError プロパティ

データベースによって返される SQLCODE を返します。

Visual Basic の構文

```
Public ReadOnly Property NativeError As Integer
```

C# の構文

```
public int NativeError {get;}
```

Source プロパティ

SQL Anywhere .NET データプロバイダの名前を返します。

Visual Basic の構文

```
Public ReadOnly Property Source As String
```

C# の構文

```
public string Source {get;}
```

SAMetaDataCollectionNames クラス

メタデータコレクションを取得する SAConnection.GetSchema(string) メソッドで使用する定数のリストを提供します。

Visual Basic の構文

```
Public NotInheritable Class SAMetaDataCollectionNames
```

C# の構文

```
public sealed class SAMetaDataCollectionNames
```

メンバー

継承されたメンバーを含む SAMetaDataCollectionNames クラスのすべてのメンバー。

名前	説明
Columns フィールド	Columns コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。
DataSourceInformation フィールド	DataSourceInformation コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。

名前	説明
DataTypes フィールド	DataTypes コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。
ForeignKeys フィールド	ForeignKeys コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。
IndexColumns フィールド	IndexColumns コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。
Indexes フィールド	Indexes コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。
MetaDataCollections フィールド	MetaDataCollections コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。
ProcedureParameters フィールド	ProcedureParameters コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。
Procedures フィールド	Procedures コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。
ReservedWords フィールド	ReservedWords コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。
Restrictions フィールド	Restrictions コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。
Tables フィールド	Tables コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。
UserDefinedTypes フィールド	UserDefinedTypes コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。

名前	説明
Users フィールド	Users コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。
ViewColumns フィールド	ViewColumns コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。
Views フィールド	Views コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。

備考

このフィールドは定数で、読み込み専用です。

参照

- SAConnection.GetSchema メソッド [SQL Anywhere .NET]177 ページ

Columns フィールド

Columns コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。

Visual Basic の構文

```
Public Shared ReadOnly Columns As String
```

C# の構文

```
public static readonly string Columns;
```

参照

- SAConnection.GetSchema メソッド [SQL Anywhere .NET]177 ページ

例

次のコードは、Columns コレクションを使用して DataTable を設定します。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Columns );
```

DataSourceInformation フィールド

DataSourceInformation コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。

Visual Basic の構文

```
Public Shared ReadOnly DataSourceInformation As String
```

C# の構文

```
public static readonly string DataSourceInformation;
```

参照

- SAConnection.GetSchema メソッド [SQL Anywhere .NET]177 ページ

例

次のコードは、DataSourceInformation コレクションを使用して DataTable を設定します。

```
DataTable schema =
  GetSchema( SAMetaDataCollectionNames.DataSourceInformation );
```

DataTypes フィールド

DataTypes コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。

Visual Basic の構文

```
Public Shared ReadOnly DataTypes As String
```

C# の構文

```
public static readonly string DataTypes;
```

参照

- SAConnection.GetSchema メソッド [SQL Anywhere .NET]177 ページ

例

次のコードは、DataTypes コレクションを使用して DataTable を設定します。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.DataTypes );
```

ForeignKeys フィールド

ForeignKeys コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。

Visual Basic の構文

```
Public Shared ReadOnly ForeignKeys As String
```

C# の構文

```
public static readonly string ForeignKeys;
```

参照

- SAConnection.GetSchema メソッド [SQL Anywhere .NET]177 ページ

例

次のコードは、ForeignKeys コレクションを使用して DataTable を設定します。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.ForeignKeys );
```

IndexColumns フィールド

IndexColumns コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。

Visual Basic の構文

```
Public Shared ReadOnly IndexColumns As String
```

C# の構文

```
public static readonly string IndexColumns;
```

参照

- SAConnection.GetSchema メソッド [SQL Anywhere .NET]177 ページ

例

次のコードは、IndexColumns コレクションを使用して DataTable を設定します。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.IndexColumns );
```

Indexes フィールド

Indexes コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。

Visual Basic の構文

```
Public Shared ReadOnly Indexes As String
```

C# の構文

```
public static readonly string Indexes;
```

参照

- SAConnection.GetSchema メソッド [SQL Anywhere .NET]177 ページ

例

次のコードは、Indexes コレクションを使用して DataTable を設定します。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Indexes );
```

MetaDataCollections フィールド

MetaDataCollections コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。

Visual Basic の構文

```
Public Shared ReadOnly MetaDataCollections As String
```

C# の構文

```
public static readonly string MetaDataCollections;
```

参照

- SAConnection.GetSchema メソッド [SQL Anywhere .NET]177 ページ

例

次のコードは、MetaDataCollections コレクションを使用して DataTable を設定します。

```
DataTable schema =
  GetSchema( SAMetaDataCollectionNames.MetaDataCollections );
```

ProcedureParameters フィールド

ProcedureParameters コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。

Visual Basic の構文

```
Public Shared ReadOnly ProcedureParameters As String
```

C# の構文

```
public static readonly string ProcedureParameters;
```

参照

- SAConnection.GetSchema メソッド [SQL Anywhere .NET]177 ページ

例

次のコードは、ProcedureParameters コレクションを使用して DataTable を設定します。

```
DataTable schema =
  GetSchema( SAMetaDataCollectionNames.ProcedureParameters );
```

Procedures フィールド

Procedures コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。

Visual Basic の構文

```
Public Shared ReadOnly Procedures As String
```

C# の構文

```
public static readonly string Procedures;
```

参照

- SAConnection.GetSchema メソッド [SQL Anywhere .NET]177 ページ

例

次のコードは、Procedures コレクションを使用して DataTable を設定します。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Procedures );
```

ReservedWords フィールド

ReservedWords コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。

Visual Basic の構文

```
Public Shared ReadOnly ReservedWords As String
```

C# の構文

```
public static readonly string ReservedWords;
```

参照

- SAConnection.GetSchema メソッド [SQL Anywhere .NET]177 ページ

例

次のコードは、ReservedWords コレクションを使用して DataTable を設定します。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.ReservedWords );
```

Restrictions フィールド

Restrictions コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。

Visual Basic の構文

```
Public Shared ReadOnly Restrictions As String
```

C# の構文

```
public static readonly string Restrictions;
```

参照

- SAConnection.GetSchema メソッド [SQL Anywhere .NET]177 ページ

例

次のコードは、Restrictions コレクションを使用して DataTable を設定します。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Restrictions );
```

Tables フィールド

Tables コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。

Visual Basic の構文

```
Public Shared ReadOnly Tables As String
```

C# の構文

```
public static readonly string Tables;
```

参照

- SAConnection.GetSchema メソッド [SQL Anywhere .NET]177 ページ

例

次のコードは、Tables コレクションを使用して DataTable を設定します。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Tables );
```

UserDefinedTypes フィールド

UserDefinedTypes コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。

Visual Basic の構文

```
Public Shared ReadOnly UserDefinedTypes As String
```

C# の構文

```
public static readonly string UserDefinedTypes;
```

参照

- SAConnection.GetSchema メソッド [SQL Anywhere .NET]177 ページ

例

次のコードは、UserDefinedTypes コレクションを使用して DataTable を設定します。

```
DataTable schema =
   GetSchema( SAMetaDataCollectionNames.UserDefinedTypes );
```

Users フィールド

Users コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。

Visual Basic の構文

```
Public Shared ReadOnly Users As String
```

C# の構文

```
public static readonly string Users;
```

参照

- SAConnection.GetSchema メソッド [SQL Anywhere .NET]177 ページ

例

次のコードは、Users コレクションを使用して DataTable を設定します。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Users );
```

ViewColumns フィールド

ViewColumns コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。

Visual Basic の構文

```
Public Shared ReadOnly ViewColumns As String
```

C# の構文

```
public static readonly string ViewColumns;
```

参照

- SAConnection.GetSchema メソッド [SQL Anywhere .NET]177 ページ

例

次のコードは、ViewColumns コレクションを使用して DataTable を設定します。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.ViewColumns );
```

Views フィールド

Views コレクションを示す SAConnection.GetSchema(string) メソッドで使用する定数を提供します。

Visual Basic の構文

```
Public Shared ReadOnly Views As String
```

C# の構文

```
public static readonly string Views;
```

参照

- SAConnection.GetSchema メソッド [SQL Anywhere .NET]177 ページ

例

次のコードは、Views コレクションを使用して DataTable を設定します。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Views );
```

SAParameter クラス

SACommand のパラメータと、必要に応じて DataSet カラムへのマッピングを表します。

Visual Basic の構文

```
Public NotInheritable Class SAParameter
    Inherits System.Data.Common.DbParameter
    Implements System.ICloneable
```

C# の構文

```
public sealed class SAParameter :
    System.Data.Common.DbParameter,
    System.ICloneable
```

基本クラス

- System.Data.Common.DbParameter
- System.ICloneable

メンバー

継承されたメンバーを含む SAParameter クラスのすべてのメンバー。

名前	説明
SAParameter コンストラクタ	値として NULL (Visual Basic の場合は Nothing) を使用して、SAParameter オブジェクトを初期化します。
ResetDbType メソッド	この SAParameter に関連付けられている型 (DbType および SADbType の値) をリセットします。
ToString メソッド	ParameterName が含まれる文字列を返します。
DbType プロパティ	パラメータの DbType を取得または設定します。

名前	説明
Direction プロパティ	パラメータが入力専用、出力専用、双方向性、またはストアドプロシージャ戻り値パラメータのいずれであるかを示す値を取得または設定します。
IsNullable プロパティ	パラメータが NULL 値を受け入れるかどうかを示す値を取得または設定します。
Offset プロパティ	Value プロパティのオフセットを取得または設定します。
ParameterName プロパティ	SAParameter の名前を取得または設定します。
Precision プロパティ	Value プロパティを表すために使用される最大桁数を取得または設定します。
SADbType プロパティ	パラメータの SADbType です。
Scale プロパティ	Value が解析される小数点の桁の数を取得または設定します。
Size プロパティ	カラム内のデータの最大サイズ (バイト単位) を取得または設定します。
SourceColumn プロパティ	DataSet にマッピングされ、値をロードしたり返したりするときに使用するソースカラムの名前を取得または設定します。
SourceColumnNullMapping プロパティ	ソースカラムが NULL 入力可かどうかを示す値を取得または設定します。
SourceVersion プロパティ	Value をロードするときに使用する DataRowVersion を取得または設定します。
Value プロパティ	パラメータの値を取得または設定します。

備考

Implements: IDbDataParameter、IDataParameter、ICloneable

SAParameter コンストラクタ

値として NULL (Visual Basic の場合は Nothing) を使用して、SAParameter オブジェクトを初期化します。

オーバロードリスト

名前	説明
SAParameter() コンストラクタ	値として NULL (Visual Basic の場合は Nothing) を使用して、SAParameter オブジェクトを初期化します。
SAParameter(string, object) コンストラクタ	SAParameter オブジェクトを、指定したパラメータ名と値で初期化します。
SAParameter(string, SADbType) コンストラクタ	SAParameter オブジェクトを、指定したパラメータ名、データ型と長さで初期化します。
SAParameter(string, SADbType, int) コンストラクタ	SAParameter オブジェクトを、指定したパラメータ名、データ型と長さで初期化します。
SAParameter(string, SADbType, int, ParameterDirection, bool, byte, byte, string, DataRowVersion, object) コンストラクタ	指定されたパラメータ名、データ型、長さ、方向、null 入力属性、数値精度、数値の位取り、ソースカラム、ソースのバージョン、値で、SAParameter オブジェクトを初期化します。
SAParameter(string, SADbType, int, string) コンストラクタ	SAParameter オブジェクトを、指定されたパラメータ名、データ型、長さで初期化します。

SAParameter() コンストラクタ

値として NULL (Visual Basic の場合は Nothing) を使用して、SAParameter オブジェクトを初期化します。

Visual Basic の構文

```
Public Sub New()
```

C# の構文

```
public SAParameter()
```

SAParameter(string, object) コンストラクタ

SAParameter オブジェクトを、指定したパラメータ名と値で初期化します。

Visual Basic の構文

```
Public Sub New(ByVal parameterName As String, ByVal value As Object)
```

C# の構文

```
public SAParameter(string parameterName, object value)
```

パラメータ

- **parameterName**　パラメータの名前。
- **value**　パラメータの値である Object。

備考

このコンストラクタの使用はおすすめしません。これは、他のデータプロバイダとの互換性のために用意されています。

SAParameter(string, SADbType) コンストラクタ

SAParameter オブジェクトを、指定したパラメータ名、データ型と長さで初期化します。

Visual Basic の構文

```
Public Sub New(ByVal parameterName As String, ByVal dbType As SADbType)
```

C# の構文

```
public SAParameter(string parameterName, SADbType dbType)
```

パラメータ

- **parameterName**　パラメータの名前。
- **dbType**　SADbType 値の 1 つ。

参照

- SAParameter.SADbType プロパティ [SQL Anywhere .NET]286 ページ

SAParameter(string, SADbType, int) コンストラクタ

SAParameter オブジェクトを、指定したパラメータ名、データ型と長さで初期化します。

Visual Basic の構文

```
Public Sub New(
    ByVal parameterName As String,
    ByVal dbType As SADbType,
    ByVal size As Integer
)
```

C# の構文

```
public SAParameter(string parameterName, SADbType dbType, int size)
```

パラメータ

- **parameterName**　パラメータの名前。
- **dbType**　SADbType 値の 1 つ。

- **size**　パラメータの長さ。

SAParameter(string, SADbType, int, ParameterDirection, bool, byte, byte, string, DataRowVersion, object) コンストラクタ

指定されたパラメータ名、データ型、長さ、方向、null 入力属性、数値精度、数値の位取り、ソースカラム、ソースのバージョン、値で、SAParameter オブジェクトを初期化します。

Visual Basic の構文

```
Public Sub New(
    ByVal parameterName As String,
    ByVal dbType As SADbType,
    ByVal size As Integer,
    ByVal direction As ParameterDirection,
    ByVal isNullable As Boolean,
    ByVal precision As Byte,
    ByVal scale As Byte,
    ByVal sourceColumn As String,
    ByVal sourceVersion As DataRowVersion,
    ByVal value As Object
)
```

C# の構文

```
public SAParameter(
    string parameterName,
    SADbType dbType,
    int size,
    ParameterDirection direction,
    bool isNullable,
    byte precision,
    byte scale,
    string sourceColumn,
    DataRowVersion sourceVersion,
    object value
)
```

パラメータ

- **parameterName**　パラメータの名前。
- **dbType**　SADbType 値の 1 つ。
- **size**　パラメータの長さ。
- **direction**　ParameterDirection 値の 1 つ。
- **isNullable**　フィールドの値を NULL にできる場合は true、できない場合は false。
- **precision**　Value が解決される小数点の左右の桁の合計数。
- **scale**　Value が解決される小数点までの桁の合計数。

- **sourceColumn**　マッピングするソースカラムの名前。
- **sourceVersion**　DataRowVersion 値の 1 つ。
- **value**　パラメータの値である Object。

SAParameter(string, SADbType, int, string) コンストラクタ

SAParameter オブジェクトを、指定されたパラメータ名、データ型、長さで初期化します。

Visual Basic の構文

```
Public Sub New(
    ByVal parameterName As String,
    ByVal dbType As SADbType,
    ByVal size As Integer,
    ByVal sourceColumn As String
)
```

C# の構文

```
public SAParameter(
    string parameterName,
    SADbType dbType,
    int size,
    string sourceColumn
)
```

パラメータ

- **parameterName**　パラメータの名前。
- **dbType**　SADbType 値の 1 つ。
- **size**　パラメータの長さ。
- **sourceColumn**　マッピングするソースカラムの名前。

ResetDbType メソッド

この SAParameter に関連付けられている型 (DbType および SADbType の値) をリセットします。

Visual Basic の構文

```
Public Overrides Sub ResetDbType()
```

C# の構文

```
public override void ResetDbType()
```

ToString メソッド

ParameterName が含まれる文字列を返します。

Visual Basic の構文

```
Public Overrides Function ToString() As String
```

C# の構文

```
public override string ToString()
```

戻り値

パラメータの名前。

DbType プロパティ

パラメータの DbType を取得または設定します。

Visual Basic の構文

```
Public Overrides Property DbType As DbType
```

C# の構文

```
public override DbType DbType {get;set;}
```

備考

SADbType と DbType はリンクされます。このため、DbType を設定すると、サポートされている SADbType に SADbType を変更します。

この値は、SADbType 列挙のメンバーにする必要があります。

Direction プロパティ

パラメータが入力専用、出力専用、双方向性、またはストアドプロシージャ戻り値パラメータのいずれであるかを示す値を取得または設定します。

Visual Basic の構文

```
Public Overrides Property Direction As ParameterDirection
```

C# の構文

```
public override ParameterDirection Direction {get;set;}
```

備考

ParameterDirection 値の 1 つ。

ParameterDirection が出力である場合、関連付けられている SACommand を実行しても値は返らず、SAParameter には NULL 値が含まれます。最後の結果セットの最後のローが読み込まれると、Output、InputOut、ReturnValue パラメータが更新されます。

IsNullable プロパティ

パラメータが NULL 値を受け入れるかどうかを示す値を取得または設定します。

Visual Basic の構文

```
Public Overrides Property IsNullable As Boolean
```

C# の構文

```
public override bool IsNullable {get;set;}
```

備考

NULL 値が受け入れられる場合、この値は true です。受け入れられない場合は false です。デフォルトは false です。NULL 値は DBNull クラスを使用して処理されます。

Offset プロパティ

Value プロパティのオフセットを取得または設定します。

Visual Basic の構文

```
Public Property Offset As Integer
```

C# の構文

```
public int Offset {get;set;}
```

備考

値のオフセット。デフォルトは 0 です。

ParameterName プロパティ

SAParameter の名前を取得または設定します。

Visual Basic の構文

```
Public Overrides Property ParameterName As String
```

C# の構文

```
public override string ParameterName {get;set;}
```

備考

デフォルトは、空の文字列です。

SQL Anywhere .NET データプロバイダは、名前付きのパラメータの代わりに疑問符 (?) が付けられた位置パラメータを使用します。

Precision プロパティ

Value プロパティを表すために使用される最大桁数を取得または設定します。

Visual Basic の構文

```
Public Property Precision As Byte
```

C# の構文

```
public byte Precision {get;set;}
```

備考

このプロパティの値は、Value プロパティを表すために使用される最大桁数です。デフォルト値は 0 です。これは、データプロバイダが Value プロパティの精度を設定することを示します。

Precision プロパティは、10 進数および数値入力パラメータに対してのみ使用されます。

SADbType プロパティ

パラメータの SADbType です。

Visual Basic の構文

```
Public Property SADbType As SADbType
```

C# の構文

```
public SADbType SADbType {get;set;}
```

備考

SADbType と DbType はリンクされます。このため、SADbType を設定すると、サポートされている DbType に DbType を変更します。

この値は、SADbType 列挙のメンバーにする必要があります。

Scale プロパティ

Value が解析される小数点の桁の数を取得または設定します。

Visual Basic の構文

```
Public Property Scale As Byte
```

C# の構文

```
public byte Scale {get;set;}
```

備考

Value が解析される小数点までの桁の数。デフォルトは 0 です。

Scale プロパティは、10 進数および数値入力パラメータに対してのみ使用されます。

Size プロパティ

カラム内のデータの最大サイズ (バイト単位) を取得または設定します。

Visual Basic の構文

```
Public Overrides Property Size As Integer
```

C# の構文

```
public override int Size {get;set;}
```

備考

このプロパティの値は、カラム内のデータの最大サイズ (バイト単位) です。デフォルト値はパラメータ値から推測されます。

このプロパティの値は、カラム内のデータの最大サイズ (バイト単位) です。デフォルト値はパラメータ値から推測されます。

Size プロパティは、バイナリおよび文字列型に対して使用されます。

可変長のデータ型の場合、Size プロパティは、サーバに送信するデータの最大量を示します。たとえば、Size プロパティを使用して、サーバに送信されるデータ量を文字列値の最初の 100 バイトに制限できます。

このプロパティを明示的に設定しない場合、サイズは、指定されたパラメータ値の実際のサイズから推測されます。固定幅のデータ型の場合、Size の値は無視されます。この値は情報用として取り出すことができ、プロバイダがパラメータの値をサーバに送信するときに使用する最大量を返します。

SourceColumn プロパティ

DataSet にマッピングされ、値をロードしたり返したりするときに使用するソースカラムの名前を取得または設定します。

Visual Basic の構文

```
Public Overrides Property SourceColumn As String
```

C# の構文

```
public override string SourceColumn {get;set;}
```

備考

DataSet にマッピングされ、値をロードしたり返したりするときに使用するソースカラムの名前を指定する文字列。

SourceColumn を空の文字列以外の値に設定すると、パラメータの値は SourceColumn 名を持つカラムから取り出されます。Direction を Input に設定すると、値は DataSet から取得されます。Direction を Output に設定すると、値はデータソースから取得されます。Direction が InputOutput の場合は Input と Output の両方です。

SourceColumnNullMapping プロパティ

ソースカラムが NULL 入力可かどうかを示す値を取得または設定します。

Visual Basic の構文

```
Public Overrides Property SourceColumnNullMapping As Boolean
```

C# の構文

```
public override bool SourceColumnNullMapping {get;set;}
```

備考

これによって、SACommandBuilder は、NULL 入力可のカラムに対して適切に Update 文を生成できます。

ソースカラムが NULL 入力可の場合は true、NULL 入力不可の場合は false が返されます。

SourceVersion プロパティ

Value をロードするときに使用する DataRowVersion を取得または設定します。

Visual Basic の構文

```
Public Overrides Property SourceVersion As DataRowVersion
```

C# の構文

```
public override DataRowVersion SourceVersion {get;set;}
```

備考

Update オペレーション時に UpdateCommand によって使用され、パラメータ値を Current と Original のどちらに設定するかを決定します。これを使用してプライマリキーを更新できます。このプロパティは、InsertCommand と DeleteCommand によって無視されます。このプロパティは、Item プロパティによって使用される DataRow のバージョン、または DataRow オブジェクトの GetChildRows メソッドに設定されます。

Value プロパティ

パラメータの値を取得または設定します。

Visual Basic の構文

```
Public Overrides Property Value As Object
```

C# の構文

```
public override object Value {get;set;}
```

備考

パラメータの値を指定する Object。

入力パラメータの場合、この値は、サーバに送信される SACommand のバウンド値です。取得および戻り値パラメータの場合、この値は、SADataReader が閉じられてから SACommand が完了したときに設定されます。

サーバに NULL パラメータを送信する場合、NULL ではなく DBNull を指定してください。システム内では、NULL 値は値を持たない空のオブジェクトです。DBNull を使用して NULL 値を表します。

アプリケーションでデータベースタイプを指定する場合、SQL Anywhere .NET データプロバイダがデータをサーバに送信するときにバウンド値はこのタイプに変換されます。プロバイダは、IConvertible インタフェースをサポートしている場合、あらゆるタイプの値を変換しようとします。指定されたタイプと値の間に互換性がない場合、変換エラーが発生する可能性があります。

Value を設定して、DbType および SADbType プロパティの両方を推測できます。

Value プロパティは Update によって上書きされます。

SAParameterCollection クラス

SACommand オブジェクトのすべてのパラメータと、必要に応じて DataSet カラムへのマッピングを表します。

Visual Basic の構文

```
Public NotInheritable Class SAParameterCollection
    Inherits System.Data.Common.DbParameterCollection
```

C# の構文

```
public sealed class SAParameterCollection :
    System.Data.Common.DbParameterCollection
```

基本クラス

- System.Data.Common.DbParameterCollection

メンバー

継承されたメンバーを含む SAParameterCollection クラスのすべてのメンバー。

名前	説明
Add メソッド	SAParameter オブジェクトをこのコレクションに追加します。
AddRange メソッド	SAParameterCollection の末尾に値の配列を追加します。
AddWithValue メソッド	このコレクションの末尾に値を追加します。

名前	説明
Clear メソッド	コレクションからすべての項目を削除します。
Contains メソッド	コレクション内に SAParameter オブジェクトがあるかどうかを示します。
CopyTo メソッド	SAParameter オブジェクトを SAParameterCollection から指定された配列にコピーします。
GetEnumerator メソッド	SAParameterCollection で反復処理する列挙子を返します。
GetParameter method (System.Data.Common.DbParameterCollection から継承)	コレクションで指定されたインデックスに System.Data.Common.DbParameter オブジェクトを返します。
IndexOf メソッド	コレクション内の SAParameter オブジェクトのロケーションを返します。
Insert メソッド	コレクション内の指定されたインデックス位置に SAParameter オブジェクトを挿入します。
Remove メソッド	指定された SAParameter オブジェクトをコレクションから削除します。
RemoveAt メソッド	指定された SAParameter オブジェクトをコレクションから削除します。
SetParameter method (System.Data.Common.DbParameterCollection から継承)	指定されたインデックスにある System.Data.Common.DbParameter オブジェクトに新しい値を設定します。
Count プロパティ	コレクション内の SAParameter オブジェクトの数を返します。
IsFixedSize プロパティ	SAParameterCollection のサイズが固定かどうかを示す値を取得します。
IsReadOnly プロパティ	SAParameterCollection が読み込み専用かどうかを示す値を取得します。
IsSynchronized プロパティ	SAParameterCollection オブジェクトが同期しているかどうかを示す値を取得します。

名前	説明
SyncRoot プロパティ	SAParameterCollection へのアクセスを同期するために使用するオブジェクトを取得します。
this プロパティ	指定されたインデックス位置の SAParameter オブジェクトを取得または設定します。

備考

SAParameterCollection にはコンストラクタがありません。SAParameterCollection オブジェクトは、SACommand オブジェクトの SACommand.Parameters プロパティから取得します。

参照

- SACommand クラス [SQL Anywhere .NET]129 ページ
- SACommand.Parameters プロパティ [SQL Anywhere .NET]151 ページ
- SAParameter クラス [SQL Anywhere .NET]278 ページ
- SAParameterCollection クラス [SQL Anywhere .NET]289 ページ

Add メソッド

SAParameter オブジェクトをこのコレクションに追加します。

オーバロードリスト

名前	説明
Add(object) メソッド	SAParameter オブジェクトをこのコレクションに追加します。
Add(SAParameter) メソッド	SAParameter オブジェクトをこのコレクションに追加します。
Add(string, object) メソッド	コレクションに対して指定されたパラメータ名と値を使用して作成された SAParameter オブジェクトをこのコレクションに追加します。
Add(string, SADbType) メソッド	コレクションに対して指定されたパラメータ名とデータ型を使用して作成された SAParameter オブジェクトをこのコレクションに追加します。

名前	説明
Add(string, SADbType, int) メソッド	コレクションに対して指定されたパラメータ名、データ型、長さを使用して作成された SAParameter オブジェクトをこのコレクションに追加します。
Add(string, SADbType, int, string) メソッド	コレクションに対して指定されたパラメータ名、データ型、長さ、ソースカラム名を使用して作成された SAParameter オブジェクトをコレクションに追加します。

Add(object) メソッド

SAParameter オブジェクトをこのコレクションに追加します。

Visual Basic の構文

```
Public Overrides Function Add(ByVal value As Object) As Integer
```

C# の構文

```
public override int Add(object value)
```

パラメータ

- **value**　コレクションに追加される SAParameter オブジェクト。

戻り値

新しい SAParameter オブジェクトのインデックス。

参照

- SAParameter クラス [SQL Anywhere .NET]278 ページ

Add(SAParameter) メソッド

SAParameter オブジェクトをこのコレクションに追加します。

Visual Basic の構文

```
Public Function Add(ByVal value As SAParameter) As SAParameter
```

C# の構文

```
public SAParameter Add(SAParameter value)
```

パラメータ

- **value**　コレクションに追加される SAParameter オブジェクト。

戻り値

新しい SAParameter オブジェクト。

Add(string, object) メソッド

コレクションに対して指定されたパラメータ名と値を使用して作成された SAParameter オブジェクトをこのコレクションに追加します。

Visual Basic の構文

```
Public Function Add(
    ByVal parameterName As String,
    ByVal value As Object
) As SAParameter
```

C# の構文

```
public SAParameter Add(string parameterName, object value)
```

パラメータ

- **parameterName** パラメータの名前。
- **value** コレクションに追加するパラメータの値。

戻り値

新しい SAParameter オブジェクト。

備考

定数 0 および 0.0 の特別な処理と、オーバロードされたメソッドの解決方法のため、このメソッドを使用するときは、定数値を型オブジェクトに明示的にキャストすることを強くおすすめします。

参照

- SAParameter クラス [SQL Anywhere .NET]278 ページ

Add(string, SADbType) メソッド

コレクションに対して指定されたパラメータ名とデータ型を使用して作成された SAParameter オブジェクトをこのコレクションに追加します。

Visual Basic の構文

```
Public Function Add(
    ByVal parameterName As String,
    ByVal saDbType As SADbType
) As SAParameter
```

C# の構文

```
public SAParameter Add(string parameterName, SADbType saDbType)
```

パラメータ

- **parameterName**　パラメータの名前。
- **saDbType**　SADbType 値の 1 つ。

戻り値

新しい SAParameter オブジェクト。

参照

- SADbType 列挙体 [SQL Anywhere .NET]332 ページ
- SAParameterCollection.Add メソッド [SQL Anywhere .NET]291 ページ

Add(string, SADbType, int) メソッド

コレクションに対して指定されたパラメータ名、データ型、長さを使用して作成された SAParameter オブジェクトをこのコレクションに追加します。

Visual Basic の構文

```
Public Function Add(
    ByVal parameterName As String,
    ByVal saDbType As SADbType,
    ByVal size As Integer
) As SAParameter
```

C# の構文

```
public SAParameter Add(
    string parameterName,
    SADbType saDbType,
    int size
)
```

パラメータ

- **parameterName**　パラメータの名前。
- **saDbType**　SADbType 値の 1 つ。
- **size**　パラメータの長さ。

戻り値

新しい SAParameter オブジェクト。

参照

- SADbType 列挙体 [SQL Anywhere .NET]332 ページ
- SAParameterCollection.Add メソッド [SQL Anywhere .NET]291 ページ

Add(string, SADbType, int, string) メソッド

コレクションに対して指定されたパラメータ名、データ型、長さ、ソースカラム名を使用して作成された SAParameter オブジェクトをコレクションに追加します。

Visual Basic の構文

```
Public Function Add(
    ByVal parameterName As String,
    ByVal saDbType As SADbType,
    ByVal size As Integer,
    ByVal sourceColumn As String
) As SAParameter
```

C# の構文

```
public SAParameter Add(
    string parameterName,
    SADbType saDbType,
    int size,
    string sourceColumn
)
```

パラメータ

- **parameterName**　パラメータの名前。
- **saDbType**　SADbType 値の 1 つ。
- **size**　カラムの長さ。
- **sourceColumn**　マッピングするソースカラムの名前。

戻り値

新しい SAParameter オブジェクト。

参照

- SADbType 列挙体 [SQL Anywhere .NET]332 ページ
- SAParameterCollection.Add メソッド [SQL Anywhere .NET]291 ページ

AddRange メソッド

SAParameterCollection の末尾に値の配列を追加します。

オーバロードリスト

名前	説明
AddRange(Array) メソッド	SAParameterCollection の末尾に値の配列を追加します。

名前	説明
AddRange(SAParameter[]) メソッド	SAParameterCollection の末尾に値の配列を追加します。

AddRange(Array) メソッド

SAParameterCollection の末尾に値の配列を追加します。

Visual Basic の構文

```
Public Overrides Sub AddRange(ByVal values As Array)
```

C# の構文

```
public override void AddRange(Array values)
```

パラメータ

- **values**　追加する値。

AddRange(SAParameter[]) メソッド

SAParameterCollection の末尾に値の配列を追加します。

Visual Basic の構文

```
Public Sub AddRange(ByVal values As SAParameter())
```

C# の構文

```
public void AddRange(SAParameter[] values)
```

パラメータ

- **values**　このコレクションの末尾に追加する SAParameter オブジェクトの配列。

AddWithValue メソッド

このコレクションの末尾に値を追加します。

Visual Basic の構文

```
Public Function AddWithValue(
    ByVal parameterName As String,
    ByVal value As Object
) As SAParameter
```

C# の構文

```
public SAParameter AddWithValue(string parameterName, object value)
```

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

パラメータ

- **parameterName**　パラメータの名前。
- **value**　追加される値。

戻り値

新しい SAParameter オブジェクト。

Clear メソッド

コレクションからすべての項目を削除します。

Visual Basic の構文

```
Public Overrides Sub Clear()
```

C# の構文

```
public override void Clear()
```

Contains メソッド

コレクション内に SAParameter オブジェクトがあるかどうかを示します。

オーバロードリスト

名前	説明
Contains(object) メソッド	コレクション内に SAParameter オブジェクトがあるかどうかを示します。
Contains(string) メソッド	コレクション内に SAParameter オブジェクトがあるかどうかを示します。

Contains(object) メソッド

コレクション内に SAParameter オブジェクトがあるかどうかを示します。

Visual Basic の構文

```
Public Overrides Function Contains(ByVal value As Object) As Boolean
```

C# の構文

```
public override bool Contains(object value)
```

パラメータ

- **value**　検索する SAParameter オブジェクト。

戻り値

コレクションに SAParameter オブジェクトが含まれる場合は true。それ以外の場合は false。

参照

- SAParameter クラス [SQL Anywhere .NET]278 ページ
- SAParameterCollection.Contains メソッド [SQL Anywhere .NET]297 ページ

Contains(string) メソッド

コレクション内に SAParameter オブジェクトがあるかどうかを示します。

Visual Basic の構文

```
Public Overrides Function Contains(ByVal value As String) As Boolean
```

C# の構文

```
public override bool Contains(string value)
```

パラメータ

- **value**　検索対象のパラメータの名前。

戻り値

コレクションに SAParameter オブジェクトが含まれる場合は true。それ以外の場合は false。

参照

- SAParameter クラス [SQL Anywhere .NET]278 ページ
- SAParameterCollection.Contains メソッド [SQL Anywhere .NET]297 ページ

CopyTo メソッド

SAParameter オブジェクトを SAParameterCollection から指定された配列にコピーします。

Visual Basic の構文

```
Public Overrides Sub CopyTo(
    ByVal array As Array,
    ByVal index As Integer
)
```

C# の構文

```
public override void CopyTo(Array array, int index)
```

パラメータ

- **array**　SAParameter オブジェクトのコピー先の配列。
- **index**　配列の開始インデックス。

参照

- SAParameter クラス [SQL Anywhere .NET]278 ページ
- SAParameterCollection クラス [SQL Anywhere .NET]289 ページ

GetEnumerator メソッド

SAParameterCollection で反復処理する列挙子を返します。

Visual Basic の構文

```
Public Overrides Function GetEnumerator()
    As System.Collections.IEnumerator
```

C# の構文

```
public override IEnumerator GetEnumerator()
```

戻り値

SAParameterCollection オブジェクトの System.Collections.IEnumerator。

参照

- SAParameterCollection クラス [SQL Anywhere .NET]289 ページ

IndexOf メソッド

コレクション内の SAParameter オブジェクトのロケーションを返します。

オーバロードリスト

名前	説明
IndexOf(object) メソッド	コレクション内の SAParameter オブジェクトのロケーションを返します。
IndexOf(string) メソッド	コレクション内の SAParameter オブジェクトのロケーションを返します。

IndexOf(object) メソッド

コレクション内の SAParameter オブジェクトのロケーションを返します。

Visual Basic の構文

```
Public Overrides Function IndexOf(ByVal value As Object) As Integer
```

C# の構文

```
public override int IndexOf(object value)
```

パラメータ

- **value**　検索する SAParameter オブジェクト。

戻り値

コレクション内の SAParameter オブジェクトの 0 から始まるロケーション。

参照

- SAParameter クラス [SQL Anywhere .NET]278 ページ
- SAParameterCollection.IndexOf メソッド [SQL Anywhere .NET]299 ページ

IndexOf(string) メソッド

コレクション内の SAParameter オブジェクトのロケーションを返します。

Visual Basic の構文

```
Public Overrides Function IndexOf(
    ByVal parameterName As String
) As Integer
```

C# の構文

```
public override int IndexOf(string parameterName)
```

パラメータ

- **parameterName**　検索するパラメータの名前。

戻り値

コレクション内の SAParameter オブジェクトの 0 から始まるインデックス。

参照

- SAParameter クラス [SQL Anywhere .NET]278 ページ
- SAParameterCollection.IndexOf メソッド [SQL Anywhere .NET]299 ページ

Insert メソッド

コレクション内の指定されたインデックス位置に SAParameter オブジェクトを挿入します。

Visual Basic の構文

```
Public Overrides Sub Insert(
    ByVal index As Integer,
    ByVal value As Object
)
```

C# の構文

```
public override void Insert(int index, object value)
```

パラメータ

- **index** コレクション内にパラメータを挿入するロケーションの 0 から始まるインデックス。
- **value** コレクションに追加される SAParameter オブジェクト。

Remove メソッド

指定された SAParameter オブジェクトをコレクションから削除します。

Visual Basic の構文

```
Public Overrides Sub Remove(ByVal value As Object)
```

C# の構文

```
public override void Remove(object value)
```

パラメータ

- **value** コレクションから削除する SAParameter オブジェクト。

RemoveAt メソッド

指定された SAParameter オブジェクトをコレクションから削除します。

オーバロードリスト

名前	説明
RemoveAt(int) メソッド	指定された SAParameter オブジェクトをコレクションから削除します。
RemoveAt(string) メソッド	指定された SAParameter オブジェクトをコレクションから削除します。

RemoveAt(int) メソッド

指定された SAParameter オブジェクトをコレクションから削除します。

Visual Basic の構文

```
Public Overrides Sub RemoveAt(ByVal index As Integer)
```

C# の構文

```
public override void RemoveAt(int index)
```

パラメータ

- **index** 削除するパラメータの 0 から始まるインデックス。

参照

- SAParameterCollection.RemoveAt メソッド [SQL Anywhere .NET]301 ページ

RemoveAt(string) メソッド

指定された SAParameter オブジェクトをコレクションから削除します。

Visual Basic の構文

```
Public Overrides Sub RemoveAt(ByVal parameterName As String)
```

C# の構文

```
public override void RemoveAt(string parameterName)
```

パラメータ

- **parameterName**　削除する SAParameter オブジェクトの名前。

参照

- SAParameterCollection.RemoveAt メソッド [SQL Anywhere .NET]301 ページ

Count プロパティ

コレクション内の SAParameter オブジェクトの数を返します。

Visual Basic の構文

```
Public ReadOnly Overrides Property Count As Integer
```

C# の構文

```
public override int Count {get;}
```

備考

コレクション内の SAParameter オブジェクトの数。

参照

- SAParameter クラス [SQL Anywhere .NET]278 ページ
- SAParameterCollection クラス [SQL Anywhere .NET]289 ページ

IsFixedSize プロパティ

SAParameterCollection のサイズが固定かどうかを示す値を取得します。

Visual Basic の構文

```
Public ReadOnly Overrides Property IsFixedSize As Boolean
```

C# の構文

```
public override bool IsFixedSize {get;}
```

備考

コレクションのサイズが固定の場合は true、そうでない場合は false。

IsReadOnly プロパティ

SAParameterCollection が読み込み専用かどうかを示す値を取得します。

Visual Basic の構文

```
Public ReadOnly Overrides Property IsReadOnly As Boolean
```

C# の構文

```
public override bool IsReadOnly {get;}
```

備考

コレクションが読み込み専用の場合は true、そうでない場合は false。

IsSynchronized プロパティ

SAParameterCollection オブジェクトが同期しているかどうかを示す値を取得します。

Visual Basic の構文

```
Public ReadOnly Overrides Property IsSynchronized As Boolean
```

C# の構文

```
public override bool IsSynchronized {get;}
```

備考

コレクションが同期している場合は true、そうでない場合は false。

SyncRoot プロパティ

SAParameterCollection へのアクセスを同期するために使用するオブジェクトを取得します。

Visual Basic の構文

```
Public ReadOnly Overrides Property SyncRoot As Object
```

C# の構文

```
public override object SyncRoot {get;}
```

this プロパティ

指定されたインデックス位置の SAParameter オブジェクトを取得または設定します。

オーバロードリスト

名前	説明
this[int] プロパティ	指定されたインデックス位置の SAParameter オブジェクトを取得または設定します。
this[string] プロパティ	指定されたインデックス位置の SAParameter オブジェクトを取得または設定します。

this[int] プロパティ

指定されたインデックス位置の SAParameter オブジェクトを取得または設定します。

Visual Basic の構文

```
Public Shadows Property Item(ByVal index As Integer) As SAParameter
```

C# の構文

```
public new SAParameter this[int index] {get;set;}
```

パラメータ

- **index**　取り出すパラメータの 0 から始まるインデックス。

戻り値

指定されたインデックス位置の SAParameter。

備考

SAParameter オブジェクト。

C# では、このプロパティは SAParameterCollection オブジェクトのインデクサです。

参照

- SAParameter クラス [SQL Anywhere .NET]278 ページ
- SAParameterCollection クラス [SQL Anywhere .NET]289 ページ

this[string] プロパティ

指定されたインデックス位置の SAParameter オブジェクトを取得または設定します。

Visual Basic の構文

```
Public Shadows Property Item(
    ByVal parameterName As String
) As SAParameter
```

C# の構文

```
public new SAParameter this[string parameterName] {get;set;}
```

パラメータ

- **parameterName**　取り出すパラメータの名前。

戻り値

指定された名前の SAParameter オブジェクト。

備考

SAParameter オブジェクト。

C# では、このプロパティは SAParameterCollection オブジェクトのインデクサです。

参照

- SAParameter クラス [SQL Anywhere .NET]278 ページ
- SAParameterCollection クラス [SQL Anywhere .NET]289 ページ
- SADataReader.GetOrdinal メソッド [SQL Anywhere .NET]239 ページ
- SADataReader.GetValue メソッド [SQL Anywhere .NET]245 ページ
- SADataReader.GetFieldType メソッド [SQL Anywhere .NET]236 ページ

SAPermission クラス

ユーザが SQL Anywhere データソースへのアクセスに適したセキュリティレベルを持っていることを、SQL Anywhere .NET データプロバイダが確認できるようにします。

Visual Basic の構文

```
Public NotInheritable Class SAPermission
    Inherits System.Data.Common.DBDataPermission
```

C# の構文

```
public sealed class SAPermission : System.Data.Common.DBDataPermission
```

基本クラス

- System.Data.Common.DBDataPermission

メンバー

継承されたメンバーを含む SAPermission クラスのすべてのメンバー。

名前	説明
SAPermission コンストラクタ	SAPermission クラスの新しいインスタンスを初期化します。

名前	説明
Add method (System.Data.Common.DBDataPermission から継承)	指定された接続文字列のアクセスを DBDataPermission の既存のステータスに追加します。
Clear method (System.Data.Common.DBDataPermission から継承)	以前に System.Data.Common.DBDataPermission.Add(System.String,System.String,System.Data.KeyRestrictionBehavior) メソッドを使用して追加したすべてのパーミッションを削除します。
Copy method (System.Data.Common.DBDataPermission から継承)	現在のパーミッションオブジェクトと同じコピーを作成して返します。
FromXml method (System.Data.Common.DBDataPermission から継承)	指定されたステータスのセキュリティオブジェクトを XML エンコードから再構築します。
Intersect method (System.Data.Common.DBDataPermission から継承)	現在のパーミッションオブジェクトと指定されたパーミッションオブジェクトの共通部分を表す新しいパーミッションオブジェクトを返します。
IsSubsetOf method (System.Data.Common.DBDataPermission から継承)	現在のパーミッションオブジェクトが指定されたパーミッションオブジェクトのサブセットであるかどうかを示す値を返します。
IsUnrestricted method (System.Data.Common.DBDataPermission から継承)	パーミッションセマンティックがわからなくてもパーミッションを無制限にできるかどうかを示す値を返します。
ToXml method (System.Data.Common.DBDataPermission から継承)	セキュリティオブジェクトの XML エンコードとその現在のステータスを作成します。
Union method (System.Data.Common.DBDataPermission から継承)	現在のパーミッションオブジェクトと指定されたパーミッションオブジェクトを統合した新しいパーミッションオブジェクトを返します。
AllowBlankPassword property (System.Data.Common.DBDataPermission から継承)	ブランクのパスワードを使用できるかどうかを示す値を取得します。

SAPermission コンストラクタ

SAPermission クラスの新しいインスタンスを初期化します。

Visual Basic の構文

```
Public Sub New(ByVal state As PermissionState)
```

C# の構文

```
public SAPermission(PermissionState state)
```

パラメータ

- **state**　PermissionState 値の 1 つ。

SAPermissionAttribute クラス

セキュリティアクションをカスタムセキュリティ属性に関連付けます。

Visual Basic の構文

```
Public NotInheritable Class SAPermissionAttribute
    Inherits System.Data.Common.DBDataPermissionAttribute
```

C# の構文

```
public sealed class SAPermissionAttribute :
    System.Data.Common.DBDataPermissionAttribute
```

基本クラス

- System.Data.Common.DBDataPermissionAttribute

メンバー

継承されたメンバーを含む SAPermissionAttribute クラスのすべてのメンバー。

名前	説明
SAPermissionAttribute コンストラクタ	SAPermissionAttribute クラスの新しいインスタンスを初期化します。
CreatePermission メソッド	属性プロパティに応じて設定された SAPermission オブジェクトを返します。
ShouldSerializeConnectionString method (System.Data.Common.DBDataPermissionAttribute から継承)	属性で接続文字列を直列化するかどうかを識別します。
ShouldSerializeKeyRestrictions method (System.Data.Common.DBDataPermissionAttribute から継承)	属性でキー制限のセットを直列化するかどうかを識別します。

名前	説明
AllowBlankPassword property (System.Data.Common.DBDataPermissionAttribute から継承)	ブランクのパスワードを使用できるかどうかを示す値を取得または設定します。
ConnectionString property (System.Data.Common.DBDataPermissionAttribute から継承)	許可される接続文字列を取得または設定します。
KeyRestrictionBehavior property (System.Data.Common.DBDataPermissionAttribute から継承)	System.Data.Common.DBDataPermissionAttribute.KeyRestrictions プロパティによって特定される接続文字列パラメータのリストが許可されている唯一の接続文字列パラメータかどうかを特定します。
KeyRestrictions property (System.Data.Common.DBDataPermissionAttribute から継承)	許可される、または許可されない接続文字列パラメータを取得または設定します。

SAPermissionAttribute コンストラクタ

SAPermissionAttribute クラスの新しいインスタンスを初期化します。

Visual Basic の構文

```
Public Sub New(ByVal action As SecurityAction)
```

C# の構文

```
public SAPermissionAttribute(SecurityAction action)
```

パラメータ

- **action** 宣言型セキュリティを使用して実行できるアクションを表す SecurityAction 値の 1 つ。

戻り値

An SAPermissionAttribute オブジェクト。

CreatePermission メソッド

属性プロパティに応じて設定された SAPermission オブジェクトを返します。

Visual Basic の構文

```
Public Overrides Function CreatePermission() As IPermission
```

C# の構文

```
public override IPermission CreatePermission()
```

SARowsCopiedEventArgs クラス

SARowsCopiedEventHandler に渡される引数のセットを表します。

Visual Basic の構文

```
Public NotInheritable Class SARowsCopiedEventArgs
```

C# の構文

```
public sealed class SARowsCopiedEventArgs
```

メンバー

継承されたメンバーを含む SARowsCopiedEventArgs クラスのすべてのメンバー。

名前	説明
SARowsCopiedEventArgs コンストラクタ	SARowsCopiedEventArgs オブジェクトの新しいインスタンスを作成します。
Abort プロパティ	バルクコピーオペレーションをアボートするかどうかを示す値を取得または設定します。
RowsCopied プロパティ	現在のバルクコピーオペレーションでコピーされるローの数を取得します。

備考

SARowsCopiedEventArgs クラスは、.NET Compact Framework 2.0 では使用できません。

SARowsCopiedEventArgs コンストラクタ

SARowsCopiedEventArgs オブジェクトの新しいインスタンスを作成します。

Visual Basic の構文

```
Public Sub New(ByVal rowsCopied As Long)
```

C# の構文

```
public SARowsCopiedEventArgs(long rowsCopied)
```

パラメータ

- **rowsCopied** 現在のバルクコピーオペレーションでコピーされるローの数を示す 64 ビット整数値。

備考

SARowsCopiedEventArgs クラスは、.NET Compact Framework 2.0 では使用できません。

Abort プロパティ

バルクコピーオペレーションをアボートするかどうかを示す値を取得または設定します。

Visual Basic の構文

```
Public Property Abort As Boolean
```

C# の構文

```
public bool Abort {get;set;}
```

備考

SARowsCopiedEventArgs クラスは、.NET Compact Framework 2.0 では使用できません。

RowsCopied プロパティ

現在のバルクコピーオペレーションでコピーされるローの数を取得します。

Visual Basic の構文

```
Public ReadOnly Property RowsCopied As Long
```

C# の構文

```
public long RowsCopied {get;}
```

備考

SARowsCopiedEventArgs クラスは、.NET Compact Framework 2.0 では使用できません。

SARowUpdatedEventArgs クラス

RowUpdated イベントのデータを提供します。

Visual Basic の構文

```
Public NotInheritable Class SARowUpdatedEventArgs
    Inherits System.Data.Common.RowUpdatedEventArgs
```

C# の構文

```
public sealed class SARowUpdatedEventArgs :
    System.Data.Common.RowUpdatedEventArgs
```

基本クラス

- System.Data.Common.RowUpdatedEventArgs

メンバー

継承されたメンバーを含む SARowUpdatedEventArgs クラスのすべてのメンバー。

名前	説明
SARowUpdatedEventArgs コンストラクタ	SARowUpdatedEventArgs クラスの新しいインスタンスを初期化します。
CopyToRows method (System.Data.Common.RowUpdatedEventArgs から継承)	変更されたローへの参照を指定された配列にコピーします。
Command プロパティ	DataAdapter.Update の呼び出し時に実行する SACommand を取得します。
Errors property (System.Data.Common.RowUpdatedEventArgs から継承)	System.Data.Common.RowUpdatedEventArgs.Command が実行されたときに .NET Framework データプロバイダによって生成されたエラーがあれば取得します。
RecordsAffected プロパティ	SQL 文の実行によって変更、挿入、または削除されたローの数を返します。
Row property (System.Data.Common.RowUpdatedEventArgs から継承)	System.Data.Common.DbDataAdapter.Update(System.Data.DataSet) 経由で送信された System.Data.DataRow を取得します。
RowCount property (System.Data.Common.RowUpdatedEventArgs から継承)	更新済みレコードのバッチで処理されたローの数を取得します。
StatementType property (System.Data.Common.RowUpdatedEventArgs から継承)	実行された SQL 文のタイプを取得します。
Status property (System.Data.Common.RowUpdatedEventArgs から継承)	System.Data.Common.RowUpdatedEventArgs.Command プロパティの System.Data.UpdateStatus を取得します。
TableMapping property (System.Data.Common.RowUpdatedEventArgs から継承)	System.Data.Common.DbDataAdapter.Update(System.Data.DataSet) 経由で送信された System.Data.Common.DataTableMapping を取得します。

SARowUpdatedEventArgs コンストラクタ

SARowUpdatedEventArgs クラスの新しいインスタンスを初期化します。

Visual Basic の構文

```
Public Sub New(
    ByVal row As DataRow,
    ByVal command As IDbCommand,
    ByVal statementType As StatementType,
    ByVal tableMapping As DataTableMapping
)
```

C# の構文

```
public SARowUpdatedEventArgs(
    DataRow row,
    IDbCommand command,
    StatementType statementType,
    DataTableMapping tableMapping
)
```

パラメータ

- **row** Update を介して送信された DataRow。
- **command** Update が呼び出されたときに実行された IDbCommand。
- **statementType** 実行されたクエリのタイプを指定する StatementType 値の 1 つ。
- **tableMapping** Update を介して送信された DataTableMapping。

Command プロパティ

DataAdapter.Update の呼び出し時に実行する SACommand を取得します。

Visual Basic の構文

```
Public ReadOnly Shadows Property Command As SACommand
```

C# の構文

```
public new SACommand Command {get;}
```

RecordsAffected プロパティ

SQL 文の実行によって変更、挿入、または削除されたローの数を返します。

Visual Basic の構文

```
Public ReadOnly Shadows Property RecordsAffected As Integer
```

C# の構文

```
public new int RecordsAffected {get;}
```

備考

変更、挿入、または削除されたローの数。文が失敗したときにローが影響されなかった場合は 0、SELECT 文の場合は -1。

SARowUpdatingEventArgs クラス

RowUpdating イベントのデータを提供します。

Visual Basic の構文

```
Public NotInheritable Class SARowUpdatingEventArgs
    Inherits System.Data.Common.RowUpdatingEventArgs
```

C# の構文

```
public sealed class SARowUpdatingEventArgs :
    System.Data.Common.RowUpdatingEventArgs
```

基本クラス

- System.Data.Common.RowUpdatingEventArgs

メンバー

継承されたメンバーを含む SARowUpdatingEventArgs クラスのすべてのメンバー。

名前	説明
SARowUpdatingEventArgs コンストラクタ	SARowUpdatingEventArgs クラスの新しいインスタンスを初期化します。
BaseCommand property (System.Data.Common.RowUpdatingEventArgs から継承)	このクラスのインスタンス用に System.Data.IDbCommand オブジェクトを取得または設定します。
Command プロパティ	Update の実行時に実行する SACommand を指定します。
Errors property (System.Data.Common.RowUpdatingEventArgs から継承)	System.Data.Common.RowUpdatedEventArgs.Command が実行するときに、.NET Framework データプロバイダによって生成されるエラーがあれば取得します。
Row property (System.Data.Common.RowUpdatingEventArgs から継承)	挿入、更新、または削除操作の一部としてサーバに送信される System.Data.DataRow を取得します。
StatementType property (System.Data.Common.RowUpdatingEventArgs から継承)	実行する SQL 文のタイプを取得します。
Status property (System.Data.Common.RowUpdatingEventArgs から継承)	System.Data.Common.RowUpdatedEventArgs.Command プロパティの System.Data.UpdateStatus を取得または設定します。

名前	説明
TableMapping property (System.Data.Common.RowUpdatingEventArgs から継承)	System.Data.Common.DbDataAdapter.Update(System.Data.DataSet) 経由で送信されるように System.Data.Common.DataTableMapping を取得します。

SARowUpdatingEventArgs コンストラクタ

SARowUpdatingEventArgs クラスの新しいインスタンスを初期化します。

Visual Basic の構文

```
Public Sub New(
    ByVal row As DataRow,
    ByVal command As IDbCommand,
    ByVal statementType As StatementType,
    ByVal tableMapping As DataTableMapping
)
```

C# の構文

```
public SARowUpdatingEventArgs(
    DataRow row,
    IDbCommand command,
    StatementType statementType,
    DataTableMapping tableMapping
)
```

パラメータ

- **row**　更新する DataRow。
- **command**　更新時に実行する IDbCommand。
- **statementType**　実行されたクエリのタイプを指定する StatementType 値の 1 つ。
- **tableMapping**　Update を介して送信された DataTableMapping。

Command プロパティ

Update の実行時に実行する SACommand を指定します。

Visual Basic の構文

```
Public Shadows Property Command As SACommand
```

C# の構文

```
public new SACommand Command {get;set;}
```

SAServerSideConnection クラス

SQL Anywhere データベースへのサーバ側の接続を表します。

Visual Basic の構文

```
Public NotInheritable Class SAServerSideConnection
```

C# の構文

```
public sealed class SAServerSideConnection
```

メンバー

継承されたメンバーを含む SAServerSideConnection クラスのすべてのメンバー。

名前	説明
Connection プロパティ	データベースサーバで SQL を実行するために必要な SAConnection オブジェクトを提供します。

備考

このクラスは CLR 外部環境からのサーバ側 SQL の実行をサポートします。データベースサーバから呼び出した CLR 関数またはメソッドは、サーバにコールバックできます。サーバとの接続はすでに確立されています。接続を確立するために SAConnection を使用する必要はありません。

Connection プロパティ

データベースサーバで SQL を実行するために必要な SAConnection オブジェクトを提供します。

Visual Basic の構文

```
Public Shared ReadOnly Property Connection As SAConnection
```

C# の構文

```
public SAConnection Connection {get;}
```

備考

サーバとの接続はすでに確立されています。接続を確立するために SAConnection を使用する必要はありません。次のような文を使用して、接続オブジェクトを取得します。

SAConnection _conn = SAServerSideConnection.Connection;

参照

- SAConnection クラス [SQL Anywhere .NET]169 ページ

例

次の例は、SAConnection オブジェクトを取得し、SQL 文を実行してテーブルを作成します。

```
private static void GetConnection() {
SAConnection _conn;
_conn = SAServerSideConnection.Connection;
SACommand cmd = _conn.CreateCommand();
cmd.CommandText = "CREATE TABLE Tab( c1 int, c2 char(128), c3 smallint, c4 double, c5
numeric(30,6) )";
cmd.ExecuteNonQuery();
cmd.Dispose();
```

SATcpOptionsBuilder クラス

SAConnection オブジェクトが使用する接続文字列の、TCP オプション部分を作成および管理する単純な方法を提供します。

Visual Basic の構文

```
Public NotInheritable Class SATcpOptionsBuilder
    Inherits SAConnectionStringBuilderBase
```

C# の構文

```
public sealed class SATcpOptionsBuilder : SAConnectionStringBuilderBase
```

基本クラス

- SAConnectionStringBuilderBase クラス [SQL Anywhere .NET]208 ページ

メンバー

継承されたメンバーを含む SATcpOptionsBuilder クラスのすべてのメンバー。

名前	説明
SATcpOptionsBuilder コンストラクタ	SATcpOptionsBuilder オブジェクトを初期化します。
Add method (System.Data.Common.DbConnectionStringBuilder から継承)	指定されたキーと値を持つエントリを System.Data.Common.DbConnectionStringBuilder に追加します。
AppendKeyValuePair method (System.Data.Common.DbConnectionStringBuilder から継承)	キーと値を既存の System.Text.StringBuilder オブジェクトに追加する十分で安全な方法を提供します。
Clear method (System.Data.Common.DbConnectionStringBuilder から継承)	System.Data.Common.DbConnectionStringBuilder インスタンスの内容をクリアします。

名前	説明
ClearPropertyDescriptors method (System.Data.Common.DbConnectionStringBuilder から継承)	関連する System.Data.Common.DbConnectionStringBuilder の System.ComponentModel.PropertyDescriptor オブジェクトのコレクションをクリアします。
ContainsKey メソッド	SAConnectionStringBuilder オブジェクトに特定のキーワードが含まれているかどうかを判断します。
EquivalentTo method (System.Data.Common.DbConnectionStringBuilder から継承)	この System.Data.Common.DbConnectionStringBuilder オブジェクト内の接続情報と指定されたオブジェクト内の接続情報を比較します。
GetKeyword メソッド	指定された SAConnectionStringBuilder プロパティのキーワードを取得します。
GetProperties method (System.Data.Common.DbConnectionStringBuilder から継承)	提供される System.Collections.Hashtable にこの System.Data.Common.DbConnectionStringBuilder のすべてのプロパティに関する情報を格納します。
GetUseLongNameAsKeyword メソッド	長い接続パラメータ名を接続文字列で使用するかどうかを示す boolean 値を取得します。
Remove メソッド	指定されたキーが設定されたエントリを SAConnectionStringBuilder インスタンスから削除します。
SetUseLongNameAsKeyword メソッド	長い接続パラメータ名を接続文字列で使用するかどうかを示す boolean 値を設定します。
ShouldSerialize メソッド	指定されたキーが、この SAConnectionStringBuilder インスタンスに存在するかどうかを示します。
ToString メソッド	SATcpOptionsBuilder オブジェクトを文字列表現に変換します。
TryGetValue メソッド	入力されたキーに対応する値を、この SAConnectionStringBuilder から取り出します。
Broadcast プロパティ	Broadcast オプションを取得または設定します。

名前	説明
BroadcastListener プロパティ	BroadcastListener オプションを取得または設定します。
BrowsableConnectionString property (System.Data.Common.DbConnectionStringBuilder から継承)	Visual Studio デザイナで System.Data.Common.DbConnectionStringBuilder.ConnectionString プロパティを表示するかどうかを示す値を取得または設定します。
ClientPort プロパティ	ClientPort オプションを取得または設定します。
ConnectionString property (System.Data.Common.DbConnectionStringBuilder から継承)	System.Data.Common.DbConnectionStringBuilder に関連付けられた接続文字列を取得または設定します。
Count property (System.Data.Common.DbConnectionStringBuilder から継承)	System.Data.Common.DbConnectionStringBuilder.ConnectionString プロパティに含まれているキーの現在の数を取得します。
DoBroadcast プロパティ	DoBroadcast オプションを取得または設定します。
Host プロパティ	Host オプションを取得または設定します。
IPV6 プロパティ	IPV6 オプションを取得または設定します。
IsFixedSize property (System.Data.Common.DbConnectionStringBuilder から継承)	System.Data.Common.DbConnectionStringBuilder のサイズが固定かどうかを示す値を取得します。
IsReadOnly property (System.Data.Common.DbConnectionStringBuilder から継承)	System.Data.Common.DbConnectionStringBuilder が読み込み専用かどうかを示す値を取得します。
Keys プロパティ	SAConnectionStringBuilder のキーが含まれた System.Collections.ICollection を取得します。
LDAP プロパティ	LDAP オプションを取得または設定します。
LocalOnly プロパティ	LocalOnly オプションを取得または設定します。
MyIP プロパティ	MyIP オプションを取得または設定します。
ReceiveBufferSize プロパティ	ReceiveBufferSize オプションを取得または設定します。

名前	説明
SendBufferSize プロパティ	Send BufferSize オプションを取得または設定します。
ServerPort プロパティ	ServerPort オプションを取得または設定します。
TDS プロパティ	TDS オプションを取得または設定します。
this プロパティ	接続キーワードの値を取得または設定します。
Timeout プロパティ	Timeout オプションを取得または設定します。
Values property (System.Data.Common.DbConnectionStringBuilder から継承)	System.Data.Common.DbConnectionStringBuilder に値が含まれている System.Collections.ICollection を取得します。
VerifyServerName プロパティ	VerifyServerName オプションを取得または設定します。

備考

SATcpOptionsBuilder クラスは、.NET Compact Framework 2.0 では使用できません。

参照

- SAConnection クラス [SQL Anywhere .NET]169 ページ

SATcpOptionsBuilder コンストラクタ

SATcpOptionsBuilder オブジェクトを初期化します。

オーバロードリスト

名前	説明
SATcpOptionsBuilder() コンストラクタ	SATcpOptionsBuilder オブジェクトを初期化します。
SATcpOptionsBuilder(string) コンストラクタ	SATcpOptionsBuilder オブジェクトを初期化します。

SATcpOptionsBuilder() コンストラクタ

SATcpOptionsBuilder オブジェクトを初期化します。

Visual Basic の構文

```
Public Sub New()
```

C# の構文

```
public SATcpOptionsBuilder()
```

備考

SATcpOptionsBuilder クラスは、.NET Compact Framework 2.0 では使用できません。

例

次の文は、SATcpOptionsBuilder オブジェクトを初期化します。

```
SATcpOptionsBuilder options = new SATcpOptionsBuilder( );
```

SATcpOptionsBuilder(string) コンストラクタ

SATcpOptionsBuilder オブジェクトを初期化します。

Visual Basic の構文

```
Public Sub New(ByVal options As String)
```

C# の構文

```
public SATcpOptionsBuilder(string options)
```

パラメータ

- **options**　SQL Anywhere TCP 接続パラメータオプション文字列。接続パラメータのリストについては、「接続パラメータ」『SQL Anywhere サーバ データベース管理』を参照してください。

備考

SATcpOptionsBuilder クラスは、.NET Compact Framework 2.0 では使用できません。

例

次の文は、SATcpOptionsBuilder オブジェクトを初期化します。

```
SATcpOptionsBuilder options = new SATcpOptionsBuilder("localonly=yes;port=6873");
```

ToString メソッド

SATcpOptionsBuilder オブジェクトを文字列表現に変換します。

Visual Basic の構文

```
Public Overrides Function ToString() As String
```

C# の構文

```
public override string ToString()
```

戻り値

構築されるオプション文字列。

Broadcast プロパティ

Broadcast オプションを取得または設定します。

Visual Basic の構文

```
Public Property Broadcast As String
```

C# の構文

```
public string Broadcast {get;set;}
```

BroadcastListener プロパティ

BroadcastListener オプションを取得または設定します。

Visual Basic の構文

```
Public Property BroadcastListener As String
```

C# の構文

```
public string BroadcastListener {get;set;}
```

ClientPort プロパティ

ClientPort オプションを取得または設定します。

Visual Basic の構文

```
Public Property ClientPort As String
```

C# の構文

```
public string ClientPort {get;set;}
```

DoBroadcast プロパティ

DoBroadcast オプションを取得または設定します。

Visual Basic の構文

```
Public Property DoBroadcast As String
```

C# の構文

```
public string DoBroadcast {get;set;}
```

Host プロパティ

Host オプションを取得または設定します。

Visual Basic の構文

```
Public Property Host As String
```

C# の構文

```
public string Host {get;set;}
```

IPV6 プロパティ

IPV6 オプションを取得または設定します。

Visual Basic の構文

```
Public Property IPV6 As String
```

C# の構文

```
public string IPV6 {get;set;}
```

LDAP プロパティ

LDAP オプションを取得または設定します。

Visual Basic の構文

```
Public Property LDAP As String
```

C# の構文

```
public string LDAP {get;set;}
```

LocalOnly プロパティ

LocalOnly オプションを取得または設定します。

Visual Basic の構文

```
Public Property LocalOnly As String
```

C# の構文

```
public string LocalOnly {get;set;}
```

MyIP プロパティ

MyIP オプションを取得または設定します。

Visual Basic の構文

```
Public Property MyIP As String
```

C# の構文

```
public string MyIP {get;set;}
```

ReceiveBufferSize プロパティ

ReceiveBufferSize オプションを取得または設定します。

Visual Basic の構文

```
Public Property ReceiveBufferSize As Integer
```

C# の構文

```
public int ReceiveBufferSize {get;set;}
```

SendBufferSize プロパティ

Send BufferSize オプションを取得または設定します。

Visual Basic の構文

```
Public Property SendBufferSize As Integer
```

C# の構文

```
public int SendBufferSize {get;set;}
```

ServerPort プロパティ

ServerPort オプションを取得または設定します。

Visual Basic の構文

```
Public Property ServerPort As String
```

C# の構文

```
public string ServerPort {get;set;}
```

TDS プロパティ

TDS オプションを取得または設定します。

Visual Basic の構文

```
Public Property TDS As String
```

C# の構文

```
public string TDS {get;set;}
```

Timeout プロパティ

Timeout オプションを取得または設定します。

Visual Basic の構文

```
Public Property Timeout As Integer
```

C# の構文

```
public int Timeout {get;set;}
```

VerifyServerName プロパティ

VerifyServerName オプションを取得または設定します。

Visual Basic の構文

```
Public Property VerifyServerName As String
```

C# の構文

```
public string VerifyServerName {get;set;}
```

SATransaction クラス

SQL トランザクションを表します。

Visual Basic の構文

```
Public NotInheritable Class SATransaction
    Inherits System.Data.Common.DbTransaction
```

C# の構文

```
public sealed class SATransaction : System.Data.Common.DbTransaction
```

基本クラス

- System.Data.Common.DbTransaction

メンバー

継承されたメンバーを含む SATransaction クラスのすべてのメンバー。

名前	説明
Commit メソッド	データベーストランザクションをコミットします。

名前	説明
Dispose method (System.Data.Common.DbTransaction から継承)	System.Data.Common.DbTransaction により使用されるアンマネージリソースを解放します。
Rollback メソッド	トランザクションを保留状態からロールバックします。
Save メソッド	トランザクションの一部をロールバックするために使用できるトランザクションのセーブポイントを作成し、セーブポイント名を指定します。
Connection プロパティ	トランザクションに関連付けられている SAConnection オブジェクトです。トランザクションが無効な場合は NULL 参照 (Visual Basic の場合は Nothing) です。
IsolationLevel プロパティ	このトランザクションの独立性レベルを指定します。
SAIsolationLevel プロパティ	このトランザクションの拡張された独立性レベルを指定します。

備考

SATransaction にはコンストラクタがありません。SATransaction オブジェクトを取得するには、いずれかの BeginTransaction メソッドを使用します。コマンドをトランザクションに関連付けるには、SACommand.Transaction プロパティを使用します。

詳細については、「Transaction 処理」64 ページと 「SACommand：ExecuteNonQuery を使用したローの挿入、削除、更新」50 ページを参照してください。

参照

- SAConnection.BeginTransaction メソッド [SQL Anywhere .NET]172 ページ
- SACommand.Transaction プロパティ [SQL Anywhere .NET]151 ページ

Commit メソッド

データベーストランザクションをコミットします。

Visual Basic の構文

```
Public Overrides Sub Commit()
```

C# の構文

```
public override void Commit()
```

Rollback メソッド

トランザクションを保留状態からロールバックします。

オーバロードリスト

名前	説明
Rollback() メソッド	トランザクションを保留状態からロールバックします。
Rollback(string) メソッド	トランザクションを保留状態からロールバックします。

Rollback() メソッド

トランザクションを保留状態からロールバックします。

Visual Basic の構文

```
Public Overrides Sub Rollback()
```

C# の構文

```
public override void Rollback()
```

備考

このトランザクションがロールバックできるのは、保留状態 (BeginTransaction が呼び出された後だが、Commit が呼び出される前) からのみです。

Rollback(string) メソッド

トランザクションを保留状態からロールバックします。

Visual Basic の構文

```
Public Sub Rollback(ByVal savePoint As String)
```

C# の構文

```
public void Rollback(string savePoint)
```

パラメータ

- **savePoint**　ロールバック先のセーブポイントの名前。

備考

このトランザクションがロールバックできるのは、保留状態 (BeginTransaction が呼び出された後だが、Commit が呼び出される前) からのみです。

Save メソッド

トランザクションの一部をロールバックするために使用できるトランザクションのセーブポイントを作成し、セーブポイント名を指定します。

Visual Basic の構文

```
Public Sub Save(ByVal savePoint As String)
```

C# の構文

```
public void Save(string savePoint)
```

パラメータ

- **savePoint** ロールバック先のセーブポイントの名前。

Connection プロパティ

トランザクションに関連付けられている SAConnection オブジェクトです。トランザクションが無効な場合は NULL 参照 (Visual Basic の場合は Nothing) です。

Visual Basic の構文

```
Public ReadOnly Shadows Property Connection As SAConnection
```

C# の構文

```
public new SAConnection Connection {get;}
```

備考

1 つのアプリケーションに複数のデータベース接続があり、各接続に 0 以上のトランザクションがある場合があります。このプロパティを使用して、BeginTransaction によって作成された特定のトランザクションに関連付けられた接続オブジェクトを確認できます。

IsolationLevel プロパティ

このトランザクションの独立性レベルを指定します。

Visual Basic の構文

```
Public ReadOnly Overrides Property IsolationLevel As
System.Data.IsolationLevel
```

C# の構文

```
public override System.Data.IsolationLevel IsolationLevel {get;}
```

備考

このトランザクションの IsolationLevel。次のいずれかを指定してください。

- Unspecified

- Chaos
- ReadUncommitted
- ReadCommitted
- RepeatableRead
- Serializable
- Snapshot

デフォルトは ReadCommitted です。

SAIsolationLevel プロパティ

このトランザクションの拡張された独立性レベルを指定します。

Visual Basic の構文

```
Public ReadOnly Property SAIsolationLevel As SAIsolationLevel
```

C# の構文

```
public SAIsolationLevel SAIsolationLevel {get;}
```

備考

このトランザクションの SAIsolationLevel。次のいずれかを指定してください。

- Unspecified
- Chaos
- ReadUncommitted
- ReadCommitted
- RepeatableRead
- Serializable
- Snapshot
- StatementSnapshot
- ReadOnlySnapshot

デフォルトは ReadCommitted です。

並列トランザクションはサポートされていません。このため、SAIsolationLevel はトランザクション全体に適用されます。

SAInfoMessageEventHandler デリゲート

SAConnection オブジェクトの SAConnection.InfoMessage イベントを処理するメソッドを表します。

Visual Basic の構文

```
Public Delegate Sub SAInfoMessageEventHandler(
    ByVal obj As Object,
    ByVal args As SAInfoMessageEventArgs
)
```

C# の構文

```
public delegate void SAInfoMessageEventHandler(
    object obj,
    SAInfoMessageEventArgs args
);
```

参照

- SAConnection クラス [SQL Anywhere .NET]169 ページ
- SAConnection.InfoMessage イベント [SQL Anywhere .NET]189 ページ

SARowUpdatedEventHandler デリゲート

SADataAdapter の RowUpdated イベントを処理するメソッドを表します。

Visual Basic の構文

```
Public Delegate Sub SARowUpdatedEventHandler(
    ByVal sender As Object,
    ByVal e As SARowUpdatedEventArgs
)
```

C# の構文

```
public delegate void SARowUpdatedEventHandler(
    object sender,
    SARowUpdatedEventArgs e
);
```

SARowUpdatingEventHandler デリゲート

SADataAdapter の RowUpdating イベントを処理するメソッドを表します。

Visual Basic の構文

```
Public Delegate Sub SARowUpdatingEventHandler(
    ByVal sender As Object,
    ByVal e As SARowUpdatingEventArgs
)
```

C# の構文

```
public delegate void SARowUpdatingEventHandler(
    object sender,
    SARowUpdatingEventArgs e
);
```

SARowsCopiedEventHandler デリゲート

SABulkCopy の SABulkCopy.SARowsCopied イベントを処理するメソッドを表します。

Visual Basic の構文

```
Public Delegate Sub SARowsCopiedEventHandler(
    ByVal sender As Object,
    ByVal rowsCopiedEventArgs As SARowsCopiedEventArgs
)
```

C# の構文

```
public delegate void SARowsCopiedEventHandler(
    object sender,
    SARowsCopiedEventArgs rowsCopiedEventArgs
);
```

備考

SARowsCopiedEventHandler デリゲートは、.NET Compact Framework 2.0 では使用できません。

参照

- SABulkCopy クラス [SQL Anywhere .NET]105 ページ

SABulkCopyOptions 列挙

SABulkCopy のインスタンスで使用する、1 つ以上のオプションを指定するビット単位フラグです。

Visual Basic の構文

```
Public Enum SABulkCopyOptions
```

C# の構文

```
public enum SABulkCopyOptions
```

メンバー

メンバー名	説明	値
Default	この値だけを指定すると、デフォルトの動作が使用されます。 デフォルトでは、トリガは有効です。	0x0
DoNotFireTriggers	これを指定すると、トリガが起動しません。 トリガを無効にするには、DBA パーミッションが必要です。トリガは、WriteToServer の開始時に接続に対して無効となり、メソッドの終了時に値がリストアされます。	0x1
KeepIdentity	これを指定すると、IDENTITY カラムにコピーされる元の値が保持されます。 デフォルトでは、送信先テーブルでは新しい ID 番号が生成されます。	0x2
TableLock	これを指定すると、コマンド LOCK TABLE table_name WITH HOLD IN SHARE MODE を使用してテーブルがロックされます。 このロックは、接続が閉じるまで続きます。	0x4

メンバー名	説明	値
UseInternalTransaction	これを指定すると、バルクコピーオペレーションの各バッチがトランザクションの中で実行されます。 これが指定されない場合、トランザクションは使用されません。このオプションを指定し、コンストラクタに SATransaction オブジェクトも指定すると、System.ArgumentException が発生します。	0x8

備考

SABulkCopyOptions 列挙は、SABulkCopy オブジェクトを構築し、WriteToServer メソッドの動作を指定する場合に使用します。

SABulkCopyOptions クラスは、.NET Compact Framework 2.0 では使用できません。

CheckConstraints オプションと KeepNulls オプションはサポートされません。

参照

- SABulkCopy クラス [SQL Anywhere .NET]105 ページ

SADbType 列挙

SQL Anywhere .NET データベースデータ型を列挙します。

Visual Basic の構文

```
Public Enum SADbType
```

C# の構文

```
public enum SADbType
```

メンバー

メンバー名	説明
BigInt	符号付き 64 ビット整数値

メンバー名	説明
Binary	指定された最大長のバイナリデータ。 列挙値 Binary と VarBinary は同等のエイリアスです。
Bit	1 ビットフラグ
Char	指定された長さの文字データ。 このタイプは常に Unicode 文字をサポートします。Char 型と VarChar 型は、完全に互換性があります。
日付	日付情報
DateTime	タイムスタンプ情報 (日付、時刻)。 列挙値 DateTime と TimeStamp は同等のエイリアスです。
DateTimeOffset	タイムスタンプ情報 (日付、時刻) オフセット。
Decimal	精度と桁数が指定された正確な数値データ。 列挙値 Decimal と Numeric は同等のエイリアスです。
Double	倍精度浮動小数点数 (8 バイト)
Float	単精度浮動小数点数 (4 バイト)。 列挙値 Float と Real は同等のエイリアスです。
Image	任意の長さのバイナリデータを格納します。
Integer	符号なし 32 ビット整数値。
LongBinary	可変長のバイナリデータ
LongNVarchar	NCHAR 文字セットの可変長文字データ。 このタイプは常に Unicode 文字をサポートします。
LongVarbit	可変長のビット配列

メンバー名	説明
LongVarchar	可変長の文字データ。 このタイプは常に Unicode 文字をサポートします。
Money	通貨データ
NChar	32767 文字までの Unicode 文字データを格納します。
NText	任意の長さの Unicode 文字データを格納します。
Numeric	精度と桁数が指定された正確な数値データ。 列挙値 Decimal と Numeric は同等のエイリアスです。
NVarChar	32767 文字までの Unicode 文字データを格納します。
Real	単精度浮動小数点数 (4 バイト)。 列挙値 Float と Real は同等のエイリアスです。
SmallDateTime	TIMESTAMP として実装されたドメイン
SmallInt	符号付き 16 ビット整数値
SmallMoney	100 万通貨単位未満の通貨データを格納します。
SysName	任意の長さの文字データを格納します。
Text	任意の長さの文字データを格納します。
Time	時刻情報
TimeStamp	タイムスタンプ情報 (日付、時刻)。 列挙値 DateTime と TimeStamp は同等のエイリアスです。

メンバー名	説明
TimeStampWithTimeZone	タイムスタンプ情報 (日付、時刻、タイムゾーン)。 列挙値 DateTime と TimeStamp は同等のエイリアスです。
TinyInt	符号なし 8 ビット整数値
UniqueIdentifier	ユニバーサルユニーク識別子 (UUID/GUID)
UniqueIdentifierStr	CHAR(36) として実装されたドメイン。 UniqueIdentifierStr は、Microsoft SQL Server の uniqueidentifier カラムをマッピングするとき、リモートデータアクセスに使用されます。
UnsignedBigInt	符号なし 64 ビット整数値
UnsignedInt	符号なし 32 ビット整数値。
UnsignedSmallInt	符号なし 16 ビット整数値
VarBinary	指定された最大長のバイナリデータ。 列挙値 **Binary** と **VarBinary** は同等のエイリアスです。
VarBit	長さが 1 ～ 32767 ビットのビット配列
VarChar	指定された最大長の文字データ。 このタイプは常に Unicode 文字をサポートします。Char 型と VarChar 型は、完全に互換性があります。
Xml	XML データ。 この型は、任意の長さの文字データを格納し、XML 文書を格納するために使用されます。

備考

下の表には、各 SADbType との互換性がある .NET 型がリストされています。整数型の場合、テーブルのカラムは、常により小さい整数型を使用して設定できるほか、実際の値がその型の範囲内にあるかぎり、より大きい型を使用して設定することも可能です。

SADbType	互換性のある .NET 型	C# 組み込みタイプ	Visual Basic 組み込みタイプ
BigInt	System.Int64	long	Long
Binary, **VarBinary**	System.Byte[]、または System.Guid (サイズが 16 の場合)	byte[]	Byte()
Bit	System.Boolean	bool	Boolean
Char, **VarChar**	System.String	String	String
Date	System.DateTime	DateTime (組み込みタイプなし)	Date
DateTime, **TimeStamp**	System.DateTime	DateTime (組み込みタイプなし)	DateTime
DateTimeOffset	System.DateTimeOffset	DateTimeOffset (組み込みタイプなし)	DateTimeOffset
Decimal, **Numeric**	System.String	decimal	Decimal
Double	System.Double	double	Double
Float, **Real**	System.Single	float	Single
Image	System.Byte[]	byte[]	Byte()
Integer	System.Int32	int	Integer
LongBinary	System.Byte[]	byte[]	Byte()
LongNVarChar	System.String	String	String
LongVarChar	System.String	String	String
Money	System.String	decimal	Decimal
NChar	System.String	String	String
NText	System.String	String	String
Numeric	System.String	decimal	Decimal
NVarChar	System.String	String	String

SADbType	互換性のある .NET 型	C# 組み込みタイプ	Visual Basic 組み込みタイプ
SmallDateTime	System.DateTime	DateTime (組み込みタイプなし)	Date
SmallInt	System.Int16	short	Short
SmallMoney	System.String	decimal	Decimal
SysName	System.String	String	String
Text	System.String	String	String
Time	System.TimeSpan	TimeSpan (組み込みタイプなし)	TimeSpan (組み込みタイプなし)
TimeStamp	System.DateTime	DateTime (組み込みタイプなし)	Date
TimeStampWithTimeZone	System.DateTimeOffset	DateTimeOffset (組み込みタイプなし)	DateTimeOffset
TinyInt	System.Byte	byte	Byte
UniqueIdentifier	System.Guid	Guid (組み込みタイプなし)	Guid (組み込みタイプなし)
UniqueIdentifierStr	System.String	String	String
UnsignedBigInt	System.UInt64	ulong	UInt64 (組み込みタイプなし)
UnsignedInt	System.UInt32	uint	UInt64 (組み込みタイプなし)
UnsignedSmallInt	System.UInt16	ushort	UInt64 (組み込みタイプなし)
Xml	System.Xml	String	String

長さが 16 のバイナリカラムには、UniqueIdentifier 型との完全な互換性があります。

参照

- SADataReader.GetFieldType メソッド [SQL Anywhere .NET]236 ページ
- SADataReader.GetDataTypeName メソッド [SQL Anywhere .NET]233 ページ

SAIsolationLevel 列挙

SQL Anywhere の独立性レベルを指定します。

Visual Basic の構文

```
Public Enum SAIsolationLevel
```

C# の構文

```
public enum SAIsolationLevel
```

メンバー

メンバー名	説明
Chaos	この独立性レベルはサポートされていません。 この独立性レベルで SAConnection.BeginTransaction を呼び出すと、例外がスローされます。 詳細については、SAConnection.BeginTransaction メソッド [SQL Anywhere .NET]172 ページを参照してください。
ReadCommitted	独立性レベル 1 と同等な動作を設定します。 この独立性レベルはダーティリードを防ぎますが、繰り返し不可能読み出しと幻ローを許可します。 詳細については、「独立性レベルと一貫性」『SQL Anywhere サーバ SQL の使用法』を参照してください。
ReadUncommitted	独立性レベル 0 と同等な動作を設定します。 この独立性レベルはダーティリード、繰り返し不可能読み出し、幻ローを許可します。 詳細については、「独立性レベルと一貫性」『SQL Anywhere サーバ SQL の使用法』を参照してください。

 Copyright © 2013, SAP AG or an SAP affiliate company.

メンバー名	説明
RepeatableRead	独立性レベル 2 と同等な動作を設定します。 この独立性レベルはダーティリードを防ぎ、繰り返し可能読み出しを保証します。ただし、幻ローを許可します。 詳細については、「独立性レベルと一貫性」『SQL Anywhere サーバ SQL の使用法』を参照してください。
Serializable	独立性レベル 3 と同等な動作を設定します。 この独立性レベルはダーティリードを防ぎ、繰り返し可能読み出しを保証し、幻ローを防ぎます。 詳細については、「独立性レベルと一貫性」『SQL Anywhere サーバ SQL の使用法』を参照してください。
Snapshot	トランザクションが最初のローの読み込み、挿入、更新、または削除を行った時点から、コミットされたデータのスナップショットを使用します。 この独立性レベルでは、トランザクションが最初のローの読み込みまたは更新を行った時点から、コミットされたデータのスナップショットを使用します。 詳細については、「独立性レベルと一貫性」『SQL Anywhere サーバ SQL の使用法』を参照してください。
Unspecified	この独立性レベルはサポートされていません。 この独立性レベルで SAConnection.BeginTransaction を呼び出すと、例外がスローされます。 詳細については、SAConnection.BeginTransaction メソッド [SQL Anywhere .NET]172 ページを参照してください。

メンバー名	説明
ReadOnlySnapshot	読み込み専用の文についてのみ、データベースから最初のローが読み込まれた時点から、コミットされたデータのスナップショットを使用します。 繰り返し不可能読み出しと幻ローは、1 つのトランザクション内で発生することはありますが、1 つの文の中で発生することはありません。更新可能な文については、updatable_statement_isolation オプションに指定された独立性レベル (0 (デフォルト)、1、2、3 のいずれか) を使用します。 詳細については、「独立性レベルと一貫性」『SQL Anywhere サーバ SQL の使用法』を参照してください。
StatementSnapshot	文で最初のローが読み込まれた時点から、コミットされたデータのスナップショットを使用します。 トランザクション内の各文で参照されるデータのスナップショットはそれぞれ異なる時点のものになります。 個々の文について、データベースから最初のローが読み込まれた時点から、コミットされたデータのスナップショットを使用します。繰り返し不可能読み出しと幻ローは、1 つのトランザクション内で発生することはありますが、1 つの文の中で発生することはありません。 詳細については、「独立性レベルと一貫性」『SQL Anywhere サーバ SQL の使用法』を参照してください。

備考

このクラス引数は System.Data.IsolationLevel クラスです。

SQL Anywhere .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のレベルなど、すべての SQL Anywhere 独立性レベルをサポートします。スナップショットアイソレーションを使用するには、SAIsolationLevel.Snapshot、SAIsolationLevel.ReadOnlySnapshot、SAIsolationLevel.StatementSnapshot のいずれかを、BeginTransaction へのパラメータとして指定します。BeginTransaction はオーバロードされているため、IsolationLevel または SAIsolationLevel を指定できます。2 つの列挙内の値は同じですが、ReadOnlySnapshot と StatementSnapshot は例外

で、SAIsolationLevel にのみ存在します。SATransaction には、SAIsolationLevel を取得する SAIsolationLevel という名前の新しいプロパティがあります。

詳細については、「スナップショットアイソレーション」『SQL Anywhere サーバ SQL の使用法』を参照してください。

SAMessageType 列挙

メッセージのタイプを示します。

Visual Basic の構文

```
Public Enum SAMessageType
```

C# の構文

```
public enum SAMessageType
```

メンバー

メンバー名	説明	値
Action	タイプ ACTION のメッセージ	2
Info	タイプ INFO のメッセージ	0
Status	タイプ STATUS のメッセージ	3
Warning	タイプ WARNING のメッセージ	1

備考

次のいずれかを指定してください。Action、Info、Status、または Warning。

OLE DB と ADO の開発

SQL Anywhere には、OLE DB と ADO の OLE DB プロバイダが含まれています。

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 SQL Anywhere に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。

参照

● 「SQL Anywhere の .NET サポート」41 ページ

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。

SQL Anywhere には **SAOLEDB** という名前の「OLE DB プロバイダ」が含まれています。このプロバイダは現在の Windows プラットフォームで使用できます。このプロバイダは Windows Mobile プラットフォームでは使用できません。

また、Microsoft OLE DB Provider for ODBC (MSDASQL) を使用すると、SQL Anywhere の ODBC ドライバで SQL Anywhere にアクセスすることもできます。

SQL Anywhere の OLE DB プロバイダを使用すると、いくつかの利点が得られます。

- 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。
- SQL Anywhere の OLE DB プロバイダを使用する場合、配備に 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。

SQL Anywhere プロバイダを使用すると、OLE DB プログラミング環境から SQL Anywhere のすべての機能を利用できます。

サポートするプラットフォーム

SQL Anywhere の 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。

サポートされるプラットフォームのリストについては、http://www.sybase.com/detail?id=1061806 を参照してください。

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。

参照

- 「3 層コンピューティングと分散トランザクション」855 ページ

SQL Anywhere を使用した ADO プログラミング

ADO (ActiveX Data Objects) は Automation インタフェースを通じて公開されてい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、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。

SQL Anywhere OLE DB プロバイダを使用して、ADO プログラミング環境から SQL Anywhere のすべての機能を利用できます。

この項では、Visual Basic から ADO を使用するときに基本的な作業を実行する方法について説明します。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。

この項のコードサンプルは、*%SQLANYSAMP16%¥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。

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

この項では、データベースに接続する簡単な Visual Basic ルーチンについて説明します。

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 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 16 Demo"
   myConn.Open()
   MsgBox("Connection succeeded")
   myConn.Close()
   Exit Sub

HandleError:
   MsgBox(ErrorToString(Err.Number))
   Exit Sub
End Sub
```

注意

この例は、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。
- SQL Anywhere の 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。

参照

- 「OLE DB 接続パラメータ」351 ページ

Command オブジェクトを使用して文を実行する方法

この項では、データベースに簡単な SQL 文を送る簡単なルーチンについて説明します。

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 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 16 Demo"
   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。

参照

- 「ADO のトランザクション」350 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」349 ページ

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 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 16 Demo"
   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。

これは、ADO のカーソルを使用した簡単な例です。

ADO からカーソルを使用する詳細な例については、「Recordset オブジェクト」348 ページを参照してください。

Recordset オブジェクト

ADO の Recordset は、SQL Anywhere で処理する場合、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。SQL Anywhere がサポートするカーソルタイプのセットについては、「カーソルのプロパティ」15 ページを参照してください。

使用できるカーソルタイプと、対応するカーソルタイプの定数と、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	SQL Anywhere 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	insensitive 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
```

参照

- 「カーソルタイプ」14 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

SQL Anywhere の OLE DB プロバイダで、カーソルによる結果セットの更新ができます。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 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 16 Demo"
   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。

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	SQL Anywhere 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
スナップショット	2097152	4
文のスナップショット	4194304	5

ADO 独立性レベル	定数	SQL Anywhere レベル
読み込み専用文のスナップショット	8388608	6

参照

- 「独立性レベルと一貫性」『SQL Anywhere サーバ SQL の使用法』

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。SQL Anywhere 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。ときには、OLE DB 接続パラメータが SQL Anywhere 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の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、SQL Anywhere 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、SQL Anywhere UserID (UID) 接続パラメータに直接マップします(例：User ID=DBA)。
- **Password**　この接続パラメータは、SQL Anywhere Password (PWD) 接続パラメータに直接マップします(例：Password=sql)。
- **Data Source**　この接続パラメータは、SQL Anywhere DataSourceName (DSN) 接続パラメータに直接マップします(例：Data Source=SQL Anywhere 16 Demo)。
- **Initial Catalog**　この接続パラメータは、SQL Anywhere DatabaseName (DBN) 接続パラメータに直接マップします(例：Initial Catalog=demo)。
- **Location**　この接続パラメータは、SQL Anywhere Host 接続パラメータに直接マップします。パラメータ値は Host パラメータ値と同じ形式になります(例：Location=localhost:4444)。
- **Extended Properties**　この接続パラメータは、SQL Anywhere 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(例：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、SQL Anywhere 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に存在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。

SQL Anywhere 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'
```

SQL Anywhere 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;UserID=DBA;Location=localhost:4444;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。SQL Anywhere 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。

参照

- 「Userid (UID) 接続パラメータ」『SQL Anywhere サーバ データベース管理』
- 「Password (PWD) 接続パラメータ」『SQL Anywhere サーバ データベース管理』
- 「DataSourceName (DSN) 接続パラメータ」『SQL Anywhere サーバ データベース管理』
- 「DatabaseName (DBN) 接続パラメータ」『SQL Anywhere サーバ データベース管理』
- 「Host 接続パラメータ」『SQL Anywhere サーバ データベース管理』
- 「OLE DB 接続プーリング」 353 ページ

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 (Sybase 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。

参照

- SQL Server Connection Pooling (ADO.NET)
- Overriding Provider Service Defaults
- OLE DB, ODBC, and Oracle Connection Pooling (ADO.NET)

Microsoft リンクサーバ

SQL Anywhere OLE DB プロバイダを使用して SQL Anywhere 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 部分構成のテーブル参照構文か 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** はカタログまたはデータベース名であり、**GROUPO** は SQL Anywhere データベースのテーブル所有者であり、また、**Customers** は SQL Anywhere 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 SQL Anywhere 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 SQL Anywhere 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 部分構成の構文では、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評価できるようになります (たとえば、WHERE、JOIN、ネストされたクエリによるクエリなど)。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 部分構成のクエリの例では、SQL Server は OLE DB プロバイダ経由で、テーブル全体での単純な SELECT (WHERE 句なし) を SQL Anywhere データベースサーバに渡します。

```
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、SQL Anywhere 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。

注意
リンクサーバの設定の前に、Windows Vista 以降の Windows を使用するときに、2、3 考慮すべきことがあります。SQL Server は、システムのサービスとして実行します。Windows Vista 以降のバージョンでサービスが設定される方法に従い、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、SQL Anywhere 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。

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、SQL Anywhere OLE DB プロバイダを使用して SQL Anywhere 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

SQL Server 2000 以降。

♦ タスク

1. Microsoft SQL Server 2005/2008 の場合は、SQL Server Management Studio を起動します。他のバージョンの 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 エクスプローラー] 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 16]** 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 (SQL Anywhere または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、SQL Server の 32 ビットバージョン用の 32 ビット DSN であるか、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 16 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、SQL Anywhere 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。

   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。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。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 SQL Anywhere 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。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。SQL Server 2005/2008 の場合、**[リンク サーバ]** » **[プロバイダ]** の下で SAOLEDB.16 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 SQL Server の下位互換性に関連しており、SQL Server が SQL Anywhere 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。

スクリプトを使用したリンクサーバの設定

リンクサーバの定義は、SQL Server スクリプトを使用して設定できます。

前提条件

SQL Server 2005 以降。

内容と備考

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、SQL Server で実行してください。

```
USE [master]
GO
EXEC master.dbo.sp_addlinkedserver @server=N'SADATABASE',
    @srvproduct=N'SQL Anywhere', @provider=N'SAOLEDB.16',
    @datasrc=N'SQL Anywhere 16 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.16', N'AllowInProcess', 1
```

♦ タスク

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 16 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。

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 SQL Anywhere の OLE DB ドライバにあ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する。	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、カラムを変更する。	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る。	サポートされていません。SQL Anywhere では、区分化されたローセットはサポートされていません。
IColumnsInfo	ローセットのカラムについての簡単な情報を得る。	サポートされています。
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する。	サポートされています。
ICommand	SQL 文を実行する。	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。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

インタフェース	内容	制限事項
ICommandPersist	command オブジェクトの状態を保持する (アクティブなローセットは保持しない)。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	コマンドを準備する。	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 プロパティを設定する。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 文を設定する。	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、設定または取得する。	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する。	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する。	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する。	サポートされています。

インタフェース	内容	制限事項
IDBDataSourceAdmin	データソースオブジェクトを作成／破壊／修正する。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。
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する (非標準の SQL キーワードを検索する)。 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する。	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する。	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する。	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する。	サポートされています。
IErrorInfo IErrorLookup IErrorRecords	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する。	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) を取り出す。	サポートされています。

インタフェース	内容	制限事項
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する。	サポートされていません。
IRowset	ローセットにアクセスする。	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る。 BLOB に対する InsertRow/SetData は実装されていない。	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する。	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する。	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する。	サポートされていません。
IRowsetIdentity	ローのハンドルを比較する。	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する。	サポートされていません。
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する。	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する。	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する。	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する。	サポートされていません。

インタフェース	内容	制限事項
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する。	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る。	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する。	サポートされていません。
ISequentialStream	BLOB カラムを取り出す。	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する。	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する。	サポートされています。
ISQLErrorInfo ISupportErrorInfo	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、変更する。	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する。	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する。	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。

インタフェース	内容	制限事項
ITransactionLocal	セッションでトランザクションを処理する。 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する。	サポートされています。
IViewChapter	既存のローセットでビューを使用する。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。
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する。	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する。	サポートされていません。
IViewSort	ソート順をビューに適用する。	サポートされていません。

OLE DB プロバイダの登録

SQL Anywhere 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体を登録します。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、それ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、SQL Anywhere OLE DB プロバイダが登録されます。

```
regsvr32 dboledb16.dll
regsvr32 dboledba16.dll
```

ODBC サポート

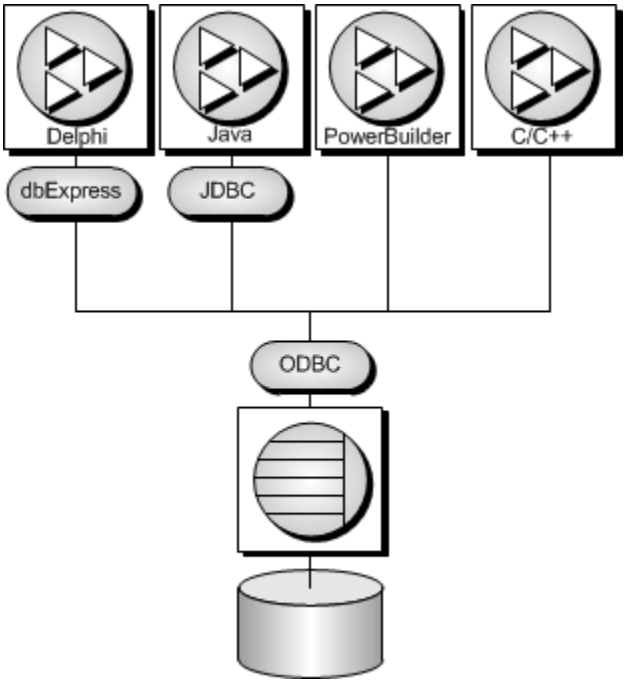
ODBC (Open Database Connectivity) は、Microsoft が開発した標準 CLI (コールレベルインタフェース) です。SQL Access Group CLI 仕様に基づいています。ODBC アプリケーションは、ODBC ドライバを提供するあらゆるデータソースに使用できます。ODBC ドライバを持つ他のデータソースにアプリケーションを移植できるようにしたい場合は、プログラミングインタフェースとして ODBC を使用することをおすすめします。

ODBC は低レベルインタフェースです。SQL Anywhere 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。

ODBC API の中核となるマニュアルは、http://msdn.microsoft.com/ja-jp/library/ms710252.aspx にある Microsoft Developer Network の Web サイトから入手できます。

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 SQL Anywhere データベースサーバにアクセスできます。



SQL Anywhere 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。

●SQL Anywhere。

●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。

注意
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。SQL Anywhere のマニュアルでは、これらのツールの使い方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。Windows Mobile 用のドライバマネージャはありません。UNIX の場合、ドライバマネージャは SQL Anywhere に付属で提供されます。

ODBC 準拠

SQL Anywhere 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。これらの機能は、http://msdn.microsoft.com/ja-jp/library/ms714177.aspx にある Microsoft の『**ODBC Programmer's Reference**』にリストされています。

SQL Anywhere がサポートする機能

SQL Anywhere での ODBC 3.5 仕様のサポートは、次のとおりです。

● **コア準拠**　SQL Anywhere は、コアレベルの機能をすべてサポートします。

● **レベル 1 準拠**　SQL Anywhere は、ODBC 関数の非同期実行を除いてレベル 1 の機能をすべてサポートします。

SQL Anywhere は、単一の接続を共有するマルチスレッドをサポートします。各スレッドからの要求は、SQL Anywhere によって直列化されます。

● **レベル 2 準拠**　SQL Anywhere は、次の機能を除いてレベル 2 の機能をすべてサポートします。

○3 語で構成されるビュー名とテーブル名。この名前は SQL Anywhere では使用できません。

○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。

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関数、データ型、定数定義がすべて含まれています。

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"
Windows Mobile	#include "ntodbc.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 SQL Anywhere インストールディレクトリの *SDK¥Include* サブフォルダ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。次の項では、実行アプリケーションを作成する方法について説明します。

Windows での ODBC アプリケーション

この項は、Windows Mobile には該当しません。

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、SQL Anywhere の ODBC ドライバ *dbodbc16.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:¥sa16¥SDK¥Include odbc32.lib
```

参照

- 「Windows Mobile での ODBC アプリケーション」368 ページ

Windows Mobile での ODBC アプリケーション

Windows Mobile オペレーティングシステムには、ODBC ドライバマネージャはありません。インポートライブラリ (*dbodbc16.lib*) では、SQL Anywhere の ODBC ドライバ *dbodbc16.dll* への直接のエントリポイントが定義されています。このファイルは、SQL Anywhere インストール環境の *SDK¥Lib¥CE¥Arm.50* サブフォルダにあります。

Windows Mobile には ODBC ドライバマネージャがないため、SQLDriverConnect 関数に提供される接続文字列内の DRIVER= パラメータに SQL Anywhere ODBC ドライバ DLL のロケーションを指定する必要があります。次はその例です。

```
szConnStrIn = "driver=¥¥Windows¥¥dbodbc16.dll;dbf=¥¥My Documents¥¥demo.db"
```

Windows Mobile の ODBC アプリケーションにリンクするには、プラットフォーム固有のインポートライブラリが含まれるディレクトリをライブラリディレクトリのリストに追加します。

サポートされる Windows Mobile のバージョンのリストについては、http://www.sybase.com/detail?id=1002288 の SQL Anywhere PC プラットフォームを参照してください。

サンプルプログラム (*odbc_sample.cpp*) は、ファイルデータソース (FileDSN 接続パラメータ) である *SQL Anywhere 16 Demo.dsn* を使用します。このファイルは、SQL Anywhere for Windows Mobile をデバイスにインストールするときに、Windows Mobile デバイスのルートディレクトリに置かれます。ODBC データソースアドミニストレータを使用してファイルデータソースをデスクトップシステムに作成できますが、作成したファイルデータソースはデスクトップ環境用に設定して、Windows Mobile 環境に合わせて編集してください。編集してからファイルデータソースを Windows Mobile デバイスにコピーします。

Windows Mobile と Unicode

SQL Anywhere は、Unicode のエンコードに使用できるマルチバイト文字コード UTF-8 を使用します。

SQL Anywhere ODBC ドライバは、ASCII (8 ビット) 文字列または Unicode (ワイド) 文字列をサポートします。UNICODE マクロは、ODBC 関数が ASCII または Unicode のどちらの文字列を期待するかを制御します。UNICODE マクロを定義してアプリケーションを構築する必要があり、ASCII の ODBC 関数も使用したい場合は、SQL_NOUNICODEMAP マクロも同時に定義してください。

Unicode ODBC の機能については、*%SQLANYSAMP16%¥SQLAnywhere¥C¥odbc.c* サンプルファイルを参照してください。

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 SQL Anywhere 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。この項では、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、SQL Anywhere 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 SQL Anywhere 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	*libdbodbc16.so (libdbodbc16.so.1)*
(Mac OS X と HP-UX 以外のすべての UNIX)	シングルスレッド	*libdbodbc16_n.so (libdbodbc16_n.so.1)*
(Mac OS X と HP-UX 以外のすべての UNIX)	マルチスレッド	*libdbodbc16_r.so (libdbodbc16_r.so.1)*
HP-UX	汎用	*libdbodbc16.sl (libdbodbc16.sl.1)*
HP-UX	シングルスレッド	*libdbodbc16_n.sl (libdbodbc16_n.sl.1)*
HP-UX	マルチスレッド	*libdbodbc16_r.sl (libdbodbc16_r.sl.1)*
Mac OS X	汎用	*libdbodbc16.dylib*
Mac OS X	シングルスレッド	*libdbodbc16_n.dylib*

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	マルチスレッド	*libdbodbc16_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	*dbodbc16.bundle*
Mac OS X	マルチスレッド	*dbodbc16_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 *libdbodbc16* 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

SQL Anywhere で ODBC ドライバマネージャの存在が検出されない場合は、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。

参照

- 「UNIX での ODBC データソース」『SQL Anywhere サーバ データベース管理』

UNIX 用 SQL Anywhere ODBC ドライバマネージャ

SQL Anywhere には、UNIX 用の ODBC ドライバマネージャが用意されています。*libdbodm16* 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。

参照

- 「unixODBC ドライバマネージャ」371 ページ
- 「UNIX 用 UTF-32ODBC ドライバマネージャ」372 ページ

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 SQL Anywhere 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する必要があります。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、SQL Anywhere 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、SQL Anywhere 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代わりに、SQL Anywhere ODBC ヘッダファイルの *unixodbc.h* 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、SQL Anywhere 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、SQL Anywhere 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 SQL Anywhere ドライバにリンクします。

   ```
   libodbc.so.1 -> libdbodbc16_r.so.1
   ```

   プラットフォームによっては、SQL Anywhere ドライバマネージャを代わりに使用できます。

参照

- 「UNIX 用 SQL Anywhere ODBC ドライバマネージャ」370 ページ
- 「UNIX での ODBC アプリケーション」369 ページ
- 「64 ビット ODBC での考慮事項」387 ページ

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 SQL Anywhere 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、SQL Anywhere 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 SQL Anywhere 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 *libdbodbcansi16_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 (*dbodbcansi16_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。

ODBC のサンプル

SQL Anywhere には、ODBC のサンプルがいくつか用意されています。サンプルは、*%SQLANYSAMP16%¥SQLAnywhere* サブディレクトリにあります。

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 *%SQLANYSAMP16%¥SQLAnywhere¥C¥odbc.c* 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。

参照

- 「サンプル Embedded SQL プログラム」481 ページ

Windows 用の ODBC サンプルプログラムの構築

サンプル ODBC プログラムを構築することでプログラムを実行でき、データベースへの接続や文の実行など、ODBC タスクがどのように実行されるのかを確認できます。

前提条件

最新バージョンの Microsoft Visual Studio が必要です。

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、Visual Studio の *vcvars32.bat* または *vcvars64.bat* (Visual Studio の旧バージョンでは *vcvarsamd64.bat*) を使用して行います。

内容と備考

%SQLANYSAMP16%¥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。

♦ タスク

1. コマンドプロンプトを開き、*%SQLANYSAMP16%¥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。

UNIX 用の ODBC サンプルプログラムの構築

サンプル ODBC プログラムを構築することでプログラムを実行でき、データベースへの接続や文の実行など、ODBC タスクがどのように実行されるのかを確認できます。

前提条件

この作業を実行するための前提条件はありません。

内容と備考

$SQLANYSAMP16/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。

♦ タスク

1. コマンドシェルを開き、*$SQLANYSAMP16/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。

ODBC サンプルプログラム

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、*%SQLANYSAMP16%¥SQLAnywhere¥C¥odbcwin.exe* を実行します。

●64 ビットの Windows では、*%SQLANYSAMP16%¥SQLAnywhere¥C¥odbcx64.exe* を実行します。

●UNIX では、*$SQLANYSAMP16/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。

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、データベース接続や SQL 文などの基本的な機能を定義します。ハンドルは、32 ビット値です。

次のハンドルは、事実上すべての 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、ODBC ドライバとデータソースによって指定されます。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、接続の確立時に使用します。

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って、SQL 文と、結果セットやパラメータなどの関連情報へアクセスできます。接続ごとに複数の文を使用できます。文は、カーソル処理 (データのフェッチ) と単一の文の実行 (INSERT、UPDATE、DELETE など) の両方に使用されます。

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 );
```

ODBC ハンドルを割り付ける方法

ODBC プログラムに必要なハンドルの型は、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

詳細については、http://msdn.microsoft.com/ja-jp/library/ms712455.aspx にある Microsoft の『**ODBC API Reference**』の「SQLAllocHandle」を参照してください。

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

詳細については、http://msdn.microsoft.com/ja-jp/library/ms710123.aspx にある Microsoft の『**ODBC API Reference**』の「SQLFreeHandle」を参照してください。

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 );
```

参照

- 「ODBC のエラー処理」401 ページ

ODBC のサンプル

%SQLANYSAMP16%¥SQLAnywhere¥ODBCConnect¥odbcconnect.cpp にある次の簡単な ODBC プログラムは、SQL Anywhere 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。

ODBC 接続関数

ODBC には、一連の接続関数が用意されています。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。

  詳細については、http://msdn.microsoft.com/ja-jp/library/ms711810.aspx にある Microsoft の『**ODBC API Reference**』の「SQLConnect」を参照してください。

- **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の外部にある SQL Anywhere 固有の接続情報を使用できます。また、SQL Anywhere 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、SQL Anywhere 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 16;UID=DBA;PWD=sql", SQL_NTS,
  scso, sizeof(scso)-1,
  &cso, SQL_DRIVER_NOPROMPT );
```

詳細については、http://msdn.microsoft.com/ja-jp/library/ms715433.aspx にある Microsoft の『**ODBC API Reference**』の「SQLDriverConnect」を参照してください。

- **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 (この場合は SQL Anywhere ドライバ) で使用するデータソースを参照したりできます。

  詳細については、http://msdn.microsoft.com/ja-jp/library/ms714565.aspx にある Microsoft の『**ODBC API Reference**』の「SQLBrowseConnect」を参照してください。

参照

- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』

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。

前提条件

この作業を実行するための前提条件はありません。

内容と備考

完全なサンプルは、*%SQLANYSAMP16%¥SQLAnywhere¥ODBCConnect¥odbcconnect.cpp* にあります。

♦ タスク

1. ODBC 環境を割り付けます。

   例：

   ```
   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。

   アプリケーションが ODBC バージョン 3 に準拠するように宣言すると、SQLSTATE 値と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。次に例を示します。

   ```
   rc = SQLSetEnvAttr( env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0 );
   ```

3. ODBC 接続項目を割り付けます。

   例：

   ```
   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 接続関数を呼び出します。

   例：

   ```
   if (rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO)
   {
     printf( "dbc allocated¥n" );
     rc = SQLConnect( dbc,
       (SQLCHAR *) "SQL Anywhere 16 Demo", SQL_NTS,
       (SQLCHAR *) "DBA", SQL_NTS,
       (SQLCHAR *) "sql", 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、終端を NULL 文字 (¥0) でマークした「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。

参照

- 「接続属性を設定する方法」378 ページ

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 のオートコミット動作をオフにします。

```
rc = SQLSetConnectAttr( dbc, SQL_AUTOCOMMIT, (SQLPOINTER)SQL_AUTOCOMMIT_OFF, 0 );
```

詳細については、http://msdn.microsoft.com/ja-jp/library/ms713605.aspx にある Microsoft の『**ODBC API Reference**』の「SQLSetConnectAttr」を参照してください。

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。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

参照

- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』

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。

接続属性のリストなどの詳細については、http://msdn.microsoft.com/ja-jp/library/ms710297.aspx にある Microsoft の『**ODBC API Reference**』の「SQLGetConnectAttr」を参照してください。

ODBC アプリケーションでのスレッドと接続

SQL Anywhere 用にマルチスレッド ODBC アプリケーションを開発できます。スレッドごとに別々の接続を使用することをおすすめします。

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。

ODBC によって変更されるサーバオプション

SQL Anywhere ODBC ドライバは、SQL Anywhere データベース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

デフォルトのオプション設定に戻すには、SET 文を実行します。次に、timestamp_format オプションをリセットする文の例を示します。

```
set temporary option timestamp_format =
```

参照

- 「ODBC トランザクションの独立性レベル」393 ページ

SQLSetConnectAttr 拡張接続属性

SQL Anywhere 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。

- **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 );
```

参照

- 「MESSAGE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「progress_messages オプション」『SQL Anywhere サーバ データベース管理』
- 「ODBC トランザクションの独立性レベル」393 ページ

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、SQL Anywhere 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。SQL Anywhere 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。SQL Anywhere 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 */
}
```

詳細については、http://msdn.microsoft.com/ja-jp/windows/hardware/gg487379.aspx にある Microsoft のホワイトペーパー**「DLL を作成する際のベストプラクティス」**を参照してください。

SQL 文の実行方法

ODBC には、SQL 文を実行するための複数の関数があります。

- **直接実行**　SQL Anywhere は SQL 文を解析し、アクセスプランを準備して、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。

参照

- 「準備文の実行」385 ページ

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)。より柔軟に文を構成する方法については、「バウンドパラメータを使用した文の実行」384 ページを参照してください。

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。

SQLExecDirect の詳細については、http://msdn.microsoft.com/ja-jp/library/ms713611.aspx にある Microsoft の『**ODBC API Reference**』を参照してください。

エラーチェックを含む完全なサンプルについては、*%SQLANYSAMP16%¥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。

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。また、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。

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

♦ タスク

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 stmt の insertstmt 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。

次の手順

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、*%SQLANYSAMP16%¥SQLAnywhere¥ODBCExecute¥odbcexecute.cpp* を参照してください。

SQLExecDirect の詳細については、http://msdn.microsoft.com/ja-jp/library/ms713611.aspx にある Microsoft の『**ODBC API Reference**』を参照してください。

参照

● 「準備文の実行」385 ページ

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

♦ タスク

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。

次の手順

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、*%SQLANYSAMP16%¥SQLAnywhere¥ODBCPrepare¥odbcprepare.cpp* を参照してください。

SQLPrepare の詳細については、http://msdn.microsoft.com/ja-jp/library/ms710926.aspx にある Microsoft の『**ODBC API Reference**』の「SQLPrepare」を参照してください。

参照

- 「準備文」2 ページ

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、SQL Anywhere 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は、*ntodbc.h* ヘッダファイルをインクルードしてください。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てください。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 SQL Anywhere 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
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

ODBC API	パラメータ (マニュアル記載のパラメータ名)
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
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

ODBC API	Value/ValuePtr 変数の型
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

詳細については、http://support.microsoft.com/kb/298678 にある Microsoft のアーティクル『ODBC 64-Bit API Changes in MDAC 2.7』を参照してください。

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

C データ型	必要なアラインメント
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。

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、結果セットの操作と更新にカーソルを使用します。SQL Anywhere 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。

参照

- 「カーソルの原則」8 ページ

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。SQL Anywhere 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。

詳細については、http://msdn.microsoft.com/ja-jp/library/ms713605.aspx にある Microsoft の『**ODBC API Reference**』の「SQLSetConnectAttr」を参照してください。

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 );
```

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 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、SQL Anywhere 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。

前方専用カーソルの簡単な例については、「データの取得」395 ページを参照してください。

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必要のあるアプリケーションの場合、カーソルの動作はもっと複雑です。アプリケーションが、他のアプリケーションによって更新されたローに戻るときの動作を考えてみます。ODBC 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。SQL Anywhere 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 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。SQL Anywhere 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**　SQLBase ROWID または Sybase TIMESTAMP などのローバージョンを比較して、最適の同時制御を使用します。

○ **SQL_CONCUR_VALUES**　値を比較して、最適の同時制御を使用します。

詳細については、http://msdn.microsoft.com/ja-jp/library/ms712631.aspx にある Microsoft の『**ODBC API Reference**』の「SQLSetStmtAttr」を参照してください。

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 );
```

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。これで文の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。

SQL Anywhere 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)。SQL Anywhere ODBC ドライバでは、この実装をサポートしていません。

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。サンプルは、*%SQLANYSAMP16%¥SQLAnywhere¥ODBCSelect¥odbcselect.cpp* にあります。

```
SQLINTEGER cbDeptID = 0, cbDeptName = SQL_NTS, cbManagerID = 0;
SQLCHAR deptName[ DEPT_NAME_LEN + 1 ];
SQLSMALLINT deptID, managerID;
SQLHENV  env;
SQLHDBC  dbc;
SQLHSTMT stmt;
SQLRETURN rc;

SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env );
SQLSetEnvAttr( env,
        SQL_ATTR_ODBC_VERSION,
        (void *)SQL_OV_ODBC3, 0 );
SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
SQLConnect( dbc,
        (SQLCHAR *) "SQL Anywhere 16 Demo", SQL_NTS,
        (SQLCHAR *) "DBA", SQL_NTS,
        (SQLCHAR *) "sql", SQL_NTS );
SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
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 );
SQLExecDirect( stmt, (SQLCHAR * )
   "SELECT DepartmentID, DepartmentName, DepartmentHeadID "
   "FROM Departments "
   "ORDER BY DepartmentID", SQL_NTS );
while( ( rc = 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 );
```

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。

カーソルを使用したローの更新と削除

Microsoft の『**ODBC Programmer's Reference**』では、クエリが位置付けオペレーションを使用して更新可能であることを示すために、SELECT...FOR UPDATE を使用するように提案しています。SQL Anywhere では、FOR UPDATE 句を使用する必要はありません。次の条件が満たされている場合は、SELECT 文が自動的に更新可能になります。

- 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、SQL Anywhere 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、SQL Anywhere では使用しないでください。

参照

- 「ansi_update_constraints オプション」『SQL Anywhere サーバ データベース管理』

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。SQL Anywhere 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 ではブックマークをサポートしますが、SQL_CURSOR_DYNAMIC と SQL_CURSOR_FORWARD_ONLY ではブックマークをサポートしていないということです。

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、SQL Anywhere 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、これは実行しないでください。

ストアドプロシージャの考慮事項

この項では、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方法について説明します。

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 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_FOUND )
    {
      rc = SQLMoreResults( stmt );
      if( rc == SQL_NO_DATA_FOUND ) break;
    }
    else
    {
      do_something( ID, Surname );
    }
  }
```

参照

- 「ストアドプロシージャ、トリガ、バッチ、ユーザ定義関数」『SQL Anywhere サーバ SQL の使用法』

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。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、SQL Anywhere 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、SQL Anywhere 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-string time-string は、SQL Anywhere 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、SQL Anywhere が受け取ることのできる有効な OUTER JOIN 式です。
- **{? = call func(p1,...)}**　func は、SQL Anywhere 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、SQL Anywhere 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、SQL Anywhere ODBC によってサポートされている関数を示します。

SQL Anywhere ODBC ドライバでサポートされている関数

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
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		WEEK
ROUND	RTRIM		YEAR
SIGN	SOUNDEX		

数値関数	文字列関数	システム関数	日付／時刻関数
SIN	SPACE		
SQRT	SUBSTRING		
TAN	UCASE		
TRUNCATE			

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、大カッコ ({) は**必ず**二重にしてください。カッコ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、SQL エスケープ構文を使用して sa_db_info プロシージャを持つデータベースプロパティを取得するには、InteractiveSQL で次のコマンドを実行します。

```
{{CALL sa_db_info( 0 ) }}
```

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、次のステータスコードのいずれかの 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 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。

```
SQLRETURN rc;
SQLHDBC dbc;
SQLHSTMT stmt;
UCHAR errmsg[100];

rc = SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
if( rc == SQL_ERROR )
{
  SQLError( env, dbc, SQL_NULL_HSTMT, NULL, NULL,
```

```
        errmsg, sizeof(errmsg), NULL );
    print_error( "Allocation failed", errmsg );
    return;
}

/* Delete items for order 2015 */
rc = SQLExecDirect( stmt,
      "DELETE FROM SalesOrderItems WHERE ID=2015",
      SQL_NTS );
if( rc == SQL_ERROR )
{
    SQLError( env, dbc, stmt, NULL, NULL,
          errmsg, sizeof(errmsg), NULL );
    print_error( "Failed to delete items", errmsg );
    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。

```
SQLRETURN rc;
SQLHDBC dbc;
SQLHSTMT stmt;
SQLSMALLINT errmsglen;
SQLINTEGER errnative;
SQLCHAR errmsg[255];
SQLCHAR errstate[5];

rc = SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
if( rc == SQL_ERROR )
{
    SQLGetDiagRec( SQL_HANDLE_DBC, dbc, 1, errstate,
        &errnative, errmsg, sizeof(errmsg), &errmsglen );
    print_error( "Allocation failed", errstate, errnative, errmsg );
    return;
}

rc = SQLExecDirect( stmt,
      "DELETE FROM SalesOrderItems WHERE ID=2015",
      SQL_NTS );
if( rc == SQL_ERROR )
{
    SQLGetDiagRec( SQL_HANDLE_STMT, stmt, 1, errstate,
        &errnative, errmsg, sizeof(errmsg), &errmsglen );
    print_error( "Failed to delete items", errstate, errnative, errmsg );
    return;
}
```

データベースにおける Java

SQL Anywhere では、データベースサーバ環境内から Java クラスを実行するメカニズムが用意されています。データベースサーバで 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。

- クライアント、中間層、またはサーバなどアプリケーションの異なるレイヤで 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。SQL Anywhere が、分散コンピューティング用のプラットフォームになります。
- 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。

データベース内の Java について

次の表は、読者の興味やバックグラウンドに応じた、Java マニュアルの参照箇所を示します。

対象読者	参照先
Java の使用を開始する Java 開発者	「チュートリアル : データベース内の Java の使用」
データベースでの Java の主な特徴を知りたい方	「データベース内の Java の FAQ」
Java からデータにアクセスする方法を知りたい方	「JDBC サポート」

データベース内の Java の FAQ

この項では、データベースにおける Java の主な特徴について説明します。

データベースにおける Java の主な特徴は？

次の各項目については、このあとの項で詳しく説明します。

- **データベースで 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**　SQL Anywhere では、Java 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 がよい理由は？

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。

- コンパイル時の徹底的なエラーチェック
- 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

データベースで独自の Java クラスを使用する方法は？

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;
```

SQL Anywhere 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。

参照

- 「CREATE PROCEDURE 文 [外部呼び出し]」『SQL Anywhere サーバ SQL リファレンス』
- 「Java クラスをデータベースにインストールする方法」413 ページ

Java はデータベースでどのように実行されるか？

SQL Anywhere 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。

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」」と呼ばれる)。スローされた例外がキャッチされ、アプリケーションのあるレベルで正しく処理されない限り、その例外は Java プログラムを終了します。

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 Anywhere では、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。

チュートリアル：データベース内の Java の使用

次の項では、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 クラスをコンパイルし、データベースにインストールすることで、SQL Anywhere で使用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。

次の項では、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、*%SQLANYSAMP16%¥SQLAnywhere¥JavaInvoice* にあります。

権限

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

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 を使用するための最初の手順は、Java コードの記述とコンパイルです。

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。

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：データベース内の Java の使用」

内容と備考

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。

♦ タスク

1. コマンドプロンプトを開き、*%SQLANYSAMP16%¥SQLAnywhere¥JavaInvoice* フォルダに移動します。

   ```
   cd %SQLANYSAMP16%¥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=%SQLANYSAMP16%¥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。

次の手順

次のレッスンで、Java コードを実行するために使用される Java VM を選択します。「レッスン 2：Java VM の選択」409 ページに進みます。

レッスン 2：Java VM の選択

Java 仮想マシン (VM) の場所を特定でき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

レッスン 2 を試みる前に、レッスン 1 のステップを完了します。

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：データベース内の Java の使用」

内容と備考

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。

♦ タスク

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 16 Demo"
   ```

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:¥¥jdk1.7.0¥¥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ない場合、またはその場所が正しくない場合、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。

```
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。

次の手順

次のレッスンでは、Java クラスをデータベースにインストールします。「レッスン 3：Java クラスのインストール」410 ページに進みます。

参照

- 「ALTER EXTERNAL ENVIRONMENT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「START JAVA 文」『SQL Anywhere サーバ SQL リファレンス』
- 「java_vm_options オプション」『SQL Anywhere サーバ データベース管理』

レッスン 3：Java クラスのインストール

Java クラスはデータベースにインストールしてから、使用してください。クラスは、Sybase Central または Interactive SQL からインストールできます。

前提条件

レッスン 3 を試みる前に、レッスン 1 および 2 のステップを完了します。

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：データベース内の Java の使用」

内容と備考

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。

♦ タスク

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(CAST(xp_getenv('CLASSPATH') AS LONG VARCHAR), ';');
```

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パスは、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。パスは、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';
```

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。

次の手順

次のレッスンでは、SQL から Java クラスのメソッドを呼び出します。「レッスン 4：Java クラスのメソッドの呼び出し」411 ページに進みます。

参照

- 「INSTALL JAVA 文」『SQL Anywhere サーバ SQL リファレンス』
- 「Java クラスをデータベースにインストールする方法」413 ページ

レッスン 4：Java クラスのメソッドの呼び出し

クラスの Java メソッドにアクセスするには、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ファンクションを作成します。

前提条件

レッスン 4 を試みる前に、レッスン 1、2、3 のステップを完了します。

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：データベース内の Java の使用」

内容と備考

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。

♦ タスク

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。

参照

- 「CREATE PROCEDURE 文 [外部呼び出し]」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE FUNCTION 文 [外部呼び出し]」『SQL Anywhere サーバ SQL リファレンス』

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、次のもの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。SQL Anywhere 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。

クラスファイルの作成

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、Utility というクラスは、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。

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。

♦ タスク

1. Sybase Central **SQL Anywhere 16** 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。

参照

- 「INSTALL JAVA 文」『SQL Anywhere サーバ SQL リファレンス』

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。

内容と備考

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。

♦ タスク

1. Sybase Central **SQL Anywhere 16** 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]** » **[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。

参照

- 「INSTALL JAVA 文」『SQL Anywhere サーバ SQL リファレンス』

クラスと JAR ファイルの更新

Sybase 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。

内容と備考

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。

♦ タスク

1. Sybase Central **SQL Anywhere 16** 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。

次の手順

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。

参照

- 「INSTALL JAVA 文」『SQL Anywhere サーバ SQL リファレンス』

データベース内の Java クラスの特殊な機能

この項では、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。

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 *%SQLANYSAMP16%¥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、渡せるのはのは 1 つの文字列のみです。

参照

- 「CREATE PROCEDURE 文 [外部呼び出し]」『SQL Anywhere サーバ SQL リファレンス』

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。

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。引数の数とタイプを確認します。

参照

- 「レッスン 4：Java クラスのメソッドの呼び出し」411 ページ

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 の EXTERNAL NAME であると宣言された 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。

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 LANGUAGE JAVA 型の 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。

   Java メソッドのラッパーであるストアドプロシージャの構文の詳細については、「CREATE PROCEDURE 文 [外部呼び出し]」『SQL Anywhere サーバ SQL リファレンス』を参照してください。

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 Anywhere 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、パラメータと戻り値の数や型を簡潔に文字で表現したものです。

参照

- 「CREATE PROCEDURE 文 [外部呼び出し]」『SQL Anywhere サーバ SQL リファレンス』
- 「Java から結果セットを返す方法」446 ページ

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 LANGUAGE JAVA 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。この項では、ストアドプロシージャ内で OUT または INOUT パラメータを利用する Java メソッドの記述方法について説明します。

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。

メソッドのシグネチャを含む構文の詳細については、「CREATE PROCEDURE 文 [外部呼び出し]」『SQL Anywhere サーバ SQL リファレンス』を参照してください。

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。Java VM でサポートされているセキュリティ管理機能を利用できます。

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 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;
```

Java VM でのシャットダウンフック

データベース内の Java サポートを提供するときに使用する SQL Anywhere Java VM クラスローダは、アプリケーションがシャットダウンフックをインストールすることを許可しています。これらのシャットダウンフックは、アプリケーションが JVM ランタイムでインストールするシャットダウンフックに非常によく似ています。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、または切断した場合、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通常のデータベース内の Java アプリケーションに対しては、シャットダウンフックのインストールが必須ではないことに注目してください。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 を停止中の特定の接続用に割り付け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にのみ使用すべきです。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。

SQL Anywhere 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、アプリケーションが Java コンパイラクラスパス内に *sajvm.jar* を含める必要があり、また、次に似た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、どれも正しい SQL Anywhere Java VM クラスローダによって自動的に実行されます。

SQL Anywhere 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に似た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。

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。

SQL Anywhere には、Type 2 ドライバである 4.0 のドライバが含まれています。

また、SQL Anywhere では、Sybase 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。

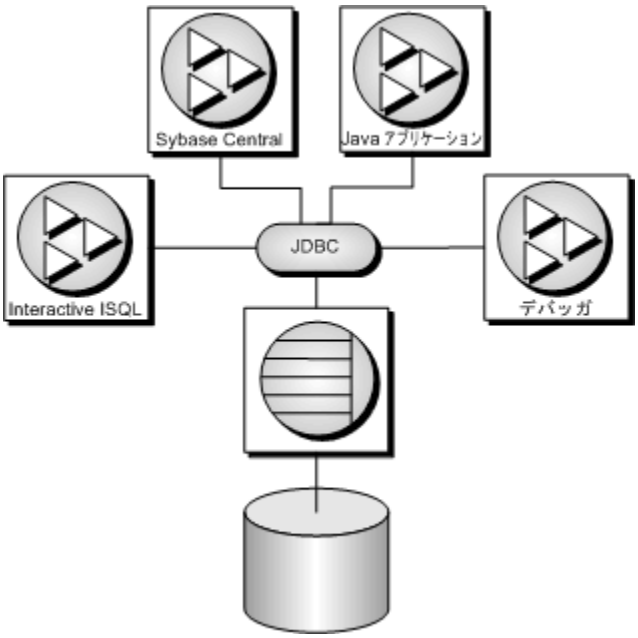
JDBC は、クライアント側のアプリケーションプログラミングインタフェースとして使用することもできますし、データベースサーバ内で使用して Java でデータベースのデータにアクセスすることもできます。

参照

- 「JDBC ドライバ」424 ページ

JDBC アプリケーション

JDBC API を使用して SQL Anywhere に接続する Java アプリケーションを開発できます。デバッガ、Sybase Central、Interactive SQL など、SQL Anywhere に付属のアプリケーションのいくつかは JDBC を使用しています。



また、Java と JDBC は Ultra Light アプリケーションを開発するための重要なプログラミング言語です。

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」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。

それぞれの例では、SQL Anywhere で JDBC を使用する特徴的な機能を示しています。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。

SQL Anywhere では、JDBC を次のように使用します。

- **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 SQL Anywhere 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 には JDBC 4.0 ドライバ (Type 2 JDBC ドライバ) が含まれており、pure Java アプリケーション用の jConnect ドライバ (Type 4 JDBC ドライバ) もサポートしています。

- **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**　この項で示したサンプルのソースコードは、*%SQLANYSAMP16%¥SQLAnywhere¥JDBC* ディレクトリにあります。

- **JDBC 仕様**　JDBC データアクセス API の詳細については、http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html を参照してください。

- **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。

  jConnect ドライバは、http://www.sybase.com/products/allproductsa-z/softwaredeveloperkit/jconnect から入手できます。

参照

- 「jConnect JDBC ドライバ」428 ページ

JDBC ドライバ

SQL Anywhere がサポートする JDBC ドライバは次のとおりです。

- **SQL Anywhere JDBC 4.0 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 SQL Anywhere と通信します。ODBC、Embedded SQL、OLE DB

アプリケーションと一貫性のある動作をします。SQL Anywhere JDBC 4.0 ドライバは、SQL Anywhere データベースに接続する場合の推奨 JDBC ドライバです。JDBC 4.0 ドライバは、JRE 1.6 以降でのみ使用できます。

JDBC 4.0 ドライバは新しい自動 JDBC ドライバ登録を利用します。したがって、アプリケーションで JDBC 4.0 ドライバを使用すると、Class.forName 呼び出しを実行して JDBC ドライバをロードする必要がなくなります。代わりに、*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() を単に呼び出すのみとなります。

JDBC 4.0 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。

JDBC 4.0 ドライバの場合、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 SQL Anywhere と通信します。

  jConnect と jConnect のマニュアルは、http://www.sybase.com/products/allproductsa-z/softwaredeveloperkit/jconnect から入手できます。

使用するドライバを選択するときは、次の要因を考慮します。

- **機能**　SQL Anywhere JDBC 4.0 ドライバと jConnect は JDBC 4.0 に準拠しています。SQL Anywhere JDBC ドライバでは、SQL Anywhere 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。jConnect JDBC ドライバでは、SQL Anywhere データベースサーバに接続したときは、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、スクロール可能なカーソルを使用できます。

  JDBC 4.0 API のマニュアルは、http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html から入手できます。SQL Anywhere JDBC ドライバがサポートする JDBC API メソッドの概要については、「JDBC 4.0 API のサポート」455 ページを参照してください。

- **Pure Java**　jConnect ドライバは pure Java ソリューションです。SQL Anywhere JDBC ドライバは、pure Java ソリューションではなく、SQL Anywhere ODBC ドライバに基づいています。

- **パフォーマンス**　ほとんどの用途で、SQL Anywhere JDBC ドライバのパフォーマンスが jConnect ドライバを上回ります。

- **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。

SQL Anywhere JDBC ドライバと jConnect のプラットフォームの対応状況については、http://www.sybase.com/detail?id=1061806 を参照してください。

参照

- 「Windows Mobile の jConnect」『SQL Anywhere サーバ データベース管理』

JDBC プログラムの構造

JDBC アプリケーションでは、一般的に次のような一連のイベントが発生します。

1. **Connection オブジェクトの作成**　DriverManager クラスの getConnection クラスメソッドを呼び出すと 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。

2. **Statement オブジェクトの生成**　Connection オブジェクトによって Statement オブジェクトが生成されます。

3. **SQL 文の引き渡し**　データベース環境で実行する SQL 文が、Statement オブジェクトに渡されます。この SQL 文がクエリの場合、これにより ResultSet オブジェクトが返されます。

   ResultSet オブジェクトには SQL 文から返されたデータが格納されていますが、一度に 1 つのローしか公開されません (カーソルの動きと同じです)。

4. **結果セットのローのループ**　ResultSet オブジェクトの next メソッドが次に示す 2 つの動作を実行します。

   ○現在のロー (ResultSet オブジェクトによって公開されている結果セット内のロー) が、1 つ前に送られます。

   ○ブール値が返され、前に送るローが存在するかどうかが示されます。

5. **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。getData 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。

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。

クライアント側 JDBC 接続とサーバ側 JDBC 接続の違い

クライアントの JDBC とデータベースサーバ内の JDBC の違いは、データベース環境との接続の確立にあります。

- **クライアント側**　クライアント側の JDBC では、接続を確立するために SQL Anywhere JDBC ドライバまたは jConnect JDBC ドライバが必要です。DriverManager.getConnection に引数を渡すと、接続が確立されます。データベース環境は、クライアントアプリケーションから見て外部アプリケーションとなります。

- **サーバ側**　JDBC がデータベースサーバ内で使用されている場合、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。それは、接続を確立するためにクライアントアプリケーションがデータベースセキュリティをすでに渡しているためです。ユーザ ID とパスワード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。サーバ側の JDBC ドライバが接続できるのは、現在の接続のデータベースのみです。

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 クラスを作成します。内部接続には "jdbc:default:connection" が必要ですが、外部接続にはホスト名とポート番号が必要です。

SQL Anywhere JDBC ドライバ

SQL Anywhere JDBC 4.0 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点で利点があります。ただし、このドライバでは pure Java ソリューションは提供されません。SQL Anywhere JDBC 4.0 ドライバをおすすめします。

参照

● 「JDBC ドライバ」 424 ページ

SQL Anywhere JDBC 4.0 ドライバをロードする方法

SQL Anywhere JDBC 4.0 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=%SQLANY16%¥java¥sajdbc4.jar;%classpath%
```

JDBC 4.0 ドライバは新しい自動 JDBC 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自動的にロードされます。

必要なファイル

SQL Anywhere JDBC 4.0 ドライバの Java コンポーネントは、SQL Anywhere インストール環境の *Java* サブフォルダ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 SQL Anywhere インストール環境の *bin32* または *bin64* サブフォルダの *dbjdbc16.dll* です。UNIX の場合、ネイティブコンポーネントは *libdbjdbc16.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。

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 を指定する必要があります。次に例を示します。

```
Connection con = DriverManager.getConnection(
    "jdbc:sqlanywhere:DSN=SQL Anywhere 16 Demo" );
```

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、JDBC 4.0 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=sql;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。

参照

- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』

jConnect JDBC ドライバ

jConnect ドライバは、http://www.sybase.com/products/allproductsa-z/softwaredeveloperkit/jconnect から別途ダウンロードして入手できます。jConnect のマニュアルも同じページから入手できます。

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 SQL Anywhere データベースに接続してください。

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。*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.sybase.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.sybase.jdbc4.jdbc.*
```

パスワードの暗号化

SQL Anywhere では、jConnect 接続でのパスワードの暗号化をサポートしています。

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 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) にアクセスする場合は、jConnect システムオブジェクトをデータベースに追加してください。

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。

内容と備考

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 SQL Anywhere データベースにインストールされます。また、jConnect システムオブジェクトは、Windows Mobile データベースには必要ありません。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。データベースは、Sybase Central や dbupgrad ユーティリティでアップグレードできます。

♦ タスク

1. Sybase Central **SQL Anywhere 16** プラグインを使用して、データベースに接続します。
2. **[ツール]** » **[SQL Anywhere 16]** » **[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。

参照

- 「アップグレードユーティリティ (dbupgrad)」『SQL Anywhere サーバ データベース管理』
- 「Windows Mobile の jConnect」『SQL Anywhere サーバ データベース管理』
- 「バックアップとデータリカバリ」『SQL Anywhere サーバ データベース管理』

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%
```

jConnect ドライバは新しい自動 JDBC 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自動的にロードされます。

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 を指定する必要があります。次に例を示します。

```
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。同じ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。SQL Anywhere に割り当てられているポート番号は 2638 です。特別な理由がないかぎり、この番号を使用してください。

接続文字列の長さは、253 文字未満にしてください。

SQL Anywhere 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。

jConnect 接続文字列でデータベースを指定する方法

各 SQL Anywhere 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 を提供することによって、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。次に例を示します。

```
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", "sql" );
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) や 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は自動的に起動します。

参照

- 「ユーティリティデータベース」『SQL Anywhere サーバ データベース管理』

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。

参照

- 「Sybase Open Client と jConnect 接続の特性」『SQL Anywhere サーバ データベース管理』
- 「sp_tsql_environment システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

JDBC クライアントアプリケーションからの接続

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。

jConnect システムオブジェクトをデータベースに追加する方法の詳細については、「jConnect JDBC ドライバ」428 ページを参照してください。

次に示す完全な Java アプリケーションはコマンドラインプログラムであり、稼働中のデータベースに接続して、一連の情報をコマンドラインに出力し、終了します。

すべての JDBC アプリケーションは、データベースのデータを処理する際、最初に接続を確立します。

次の例は、通常のクライアント／サーバ接続である外部接続を示しています。データベースサーバ内で動作している Java クラスから内部接続を作成する方法については、「サーバ側 JDBC クラスから接続を確立する方法」435 ページを参照してください。

接続サンプルのコード

この例では、SQL Anywhere JDBC ドライバの JDBC 4.0 バージョンを使用して、デフォルトでデータベースに接続します。異なるドライバを使用するには、コマンドラインでドライバ名 (jdbc4、jConnect) を渡します。JDBC 4.0 ドライバと jConnect の使用例は、コードに含まれています。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。ソースコードは、*%SQLANYSAMP16%¥SQLAnywhere¥JDBC* ディレクトリの *JDBCConnect.java* ファイルにあります。

```
import java.io.*;
import java.sql.*;

public class JDBCConnect
{
   public static void main( String args[] )
   {
      try
      {
         String arg;
         Connection con;

         // Select the JDBC driver and create a connection.
         // May throw a SQLException.
         // Choices are:
         // 1. jConnect driver
         // 2. SQL Anywhere JDBC 4.0 driver
         arg = "jdbc4";
         if( args.length > 0 ) arg = args[0];
         if( arg.compareToIgnoreCase( "jconnect" ) == 0 )
         {
            con = DriverManager.getConnection(
               "jdbc:sybase:Tds:localhost:2638", "DBA", "sql");
         }
         else
         {
            con = DriverManager.getConnection(
               "jdbc:sqlanywhere:uid=DBA;pwd=sql" );
         }

         System.out.println("Using "+arg+" driver");

         // Create a statement object, the container for the SQL
         // statement. May throw a SQLException.
         Statement stmt = con.createStatement();

         // Create a result set object by executing the query.
         // May throw a SQLException.
         ResultSet rs = stmt.executeQuery(
            "SELECT ID, GivenName, Surname FROM Customers");
```

```
            // Process the result set.
            while (rs.next())
            {
                int value = rs.getInt(1);
                String FirstName = rs.getString(2);
                String LastName = rs.getString(3);
                System.out.println(value+" "+FirstName+" "+LastName);
            }
            rs.close();
            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);
    }
}
```

接続サンプルの動作

この外部接続サンプルは Java コマンドラインプログラムです。

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。

main メソッド

それぞれの Java アプリケーションでは main という名前のメソッドを持つクラスが必要です。このメソッドはプログラムの起動時に呼び出されます。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. コマンドラインの引数に基づいて、ロードするドライバを決定します。SQL Anywhere JDBC 4.0 と jConnect 7.0 のドライバがクラスファイルパスにあると、起動時に自動的にロードされます。

2. 選択された JDBC ドライバ URL を使用して、実行しているデフォルト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. 結果セット、文、接続の各オブジェクトを閉じます。

接続サンプルの実行

JDBC アプリケーションを作成して実行する手順については、例を使用して示します。

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。

内容と備考

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。

♦ タスク

1. コマンドプロンプトで、*%SQLANYSAMP16%¥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。

   ```
   dbsrv16 "%SQLANYSAMP16%¥demo.db"
   ```

3. CLASSPATH 環境変数を設定します。この例では、*sajdbc4.jar* にある SQL Anywhere JDBC 4.0 ドライバを使用します。

   ```
   set classpath=.;%SQLANY16%¥java¥sajdbc4.jar
   ```

   jConnect ドライバを使用している場合は、次のコマンドを使用します (*path* は jConnect インストールディレクトリ)。

   ```
   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
   ```

jconnect などのコマンドライン引数を追加して、異なる JDBC ドライバをロードします。

```
java JDBCConnect jconnect
```

6. ID 番号と顧客名のリストがコマンドプロンプトに表示されることを確認します。

   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。クラスファイルパスが正しいことを確認します。設定が間違っていると、クラスを検索できません。

結果

ID 番号と顧客名のリストが表示されます。

サーバ側 JDBC クラスから接続を確立する方法

JDBC の SQL 文は、Connection オブジェクトの createStatement メソッドを使用して作成されます。サーバ内で動作するクラスも、Connection オブジェクトを作成するために接続を確立する必要があります。

サーバ側 JDBC クラス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、クラスでは現在の接続を使用できるからです。

サーバ側接続サンプルのコード

次はサーバ側の接続サンプルのソースコードです。これは、*JDBCConnect.java* の例に変更を加えたもので、*%SQLANYSAMP16%¥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
    {
      // Open the connection. May throw a SQLException.
      Connection con = DriverManager.getConnection(
          "jdbc:default:connection" );

      // Create a statement object, the container for the SQL
      // statement. May throw a SQLException.
      Statement stmt = con.createStatement();
      // Create a result set object by executing the query.
      // May throw a SQLException.
      ResultSet rs = stmt.executeQuery(
        "SELECT ID, GivenName, Surname FROM Customers");

      // Process the result set.
      while (rs.next())
      {
        int value = rs.getInt(1);
        String FirstName = rs.getString(2);
```

```
        String LastName = rs.getString(3);
        System.out.println(value+" "+FirstName+" "+LastName);
      }
      rs.close();
      stmt.close();
      con.close();
    }
    catch (SQLException sqe)
    {
      System.out.println("Unexpected exception : " +
              sqe.toString() + ", sqlstate = " +
              sqe.getSQLState());
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

サーバ側接続サンプルの動作の違い

サーバ側の接続サンプルは、次のことを除いてクライアント側の接続サンプルとほぼ同じです。

1. JDBC ドライバをプリロードする必要はありません。
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. System.exit() 文は削除されています。

サーバ側接続サンプルの実行

JDBC サーバ側アプリケーションを作成して実行する手順については、例を使用して示します。

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

内容と備考

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。

♦ タスク

1. コマンドプロンプトで、*%SQLANYSAMP16%¥SQLAnywhere¥JDBC* ディレクトリに移動します。

   ```
   cd %SQLANYSAMP16%¥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=.;%SQLANYSAMP16%¥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。

   ```
   dbsrv16 "%SQLANYSAMP16%¥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';
   ```

   Sybase 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 PROCEDURE JDBCConnect(OUT args 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. ID 番号と顧客名のリスト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ることを確認します。

   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。

結果

ID 番号と顧客名のリスト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。

JDBC 接続についての注意

- **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 4.0 ドライバの場合は、次を使用します。

  - TRANSACTION_NONE
  - TRANSACTION_READ_COMMITTED
  - TRANSACTION_READ_UNCOMMITTED
  - TRANSACTION_REPEATABLE_READ
  - TRANSACTION_SERIALIZABLE
  - sybase.jdbc4.sqlanywhere.IConnection.SA_TRANSACTION_SNAPSHOT
  - sybase.jdbc4.sqlanywhere.IConnection.SA_TRANSACTION_STATEMENT_SNAPSHOT
  - sybase.jdbc4.sqlanywhere.IConnection.SA_TRANSACTION_STATEMENT_READONLY_SNAPSHOT

  次の例は、JDBC 4.0 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(
        sybase.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 );
  }
  ```

  getTransactionIsolation と setTransactionIsolation メソッドの詳細については、http://docs.oracle.com/javase/6/docs/technotes/guides/jdbc/ にある java.sql.Connection インタフェースのマニュアルを参照してください。

- **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たい場合があります。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。

getTransactionIsolation、setTransactionIsolation、isReadOnly の各メソッドの詳細については、http://docs.oracle.com/javase/6/docs/technotes/guides/jdbc/ にある java.sql.Connection インタフェースのマニュアルを参照してください。

JDBC を使用したデータへのアクセス

データベース内にクラスの一部、またはすべてを格納している Java アプリケーションは、従来の SQL ストアドプロシージャよりもはるかに有利です。ただし、導入段階では、SQL ストアドプロシージャに相当するものを使用して、JDBC の機能を確認した方が便利な場合もあります。次の例では、ローを Departments テーブルに挿入する Java クラスを記述しています。

その他の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。また、実行プランの解析と選択を文の「準備」と呼びます。

同じ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、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。動的 SQL については、「より効率的なアクセスのために準備文を使用する方法」442 ページで説明します。

JDBC サンプルの準備

次の項に記載されているコードフラグメントは、完全なクラス *%SQLANYSAMP16%¥SQLAnywhere¥JDBC¥JDBCExample.java* から引用しています。これらの項の準備として、サンプル Java アプリケーションはコンパイルされ、データベースにインストールされます。

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。

♦ タスク

1. *JDBCExample.java* ソースコードをコンパイルします。
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。

   Sybase 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。

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();
  }
}
```

注意

- このコードフラグメントは、*%SQLANYSAMP16%¥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メッセージウィンドウに表示されます。

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

Java Development Kit (JDK) をインストールしている必要があります。

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。

♦ タスク

1. Interactive SQL から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。

   Java のサンプルクラスをインストールする方法の詳細については、「JDBC サンプルの準備」439 ページを参照してください。

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 CHAR(50))
  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;
```

   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。

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。

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();
  }
}
```

注意

- このコードフラグメントは、*%SQLANYSAMP16%¥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メッセージウィンドウに表示されます。

JDBC からの準備 INSERT 文と DELETE 文の使用

サンプル JDBC アプリケーションがデータベースサーバから呼び出され、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

Java Development Kit (JDK) をインストールしている必要があります。

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。

♦ タスク

1. Interactive SQL から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。

   Java のサンプルクラスをインストールする方法の詳細については、「JDBC サンプルの準備」439 ページを参照してください。

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 CHAR(50))
  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;
```

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。

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。

参照

- 「準備文」2 ページ

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、INSERT 文を準備します。

```
// Build the INSERT statement
String sqlStr = "INSERT INTO Departments " +
          "( DepartmentID, DepartmentName ) " +
          "VALUES ( ? , ? )";
// Prepare the statement
PreparedStatement stmt =
    con.prepareStatement( sqlStr );
```

2. 次のようにして、準備された INSERT 文のパラメータを設定してバッチ処理します。

```
// loop to batch "n" sets of parameters
for( i=0; i < n; i++ )
{
   // "stmt" is the original prepared insert statement from step 1.
   stmt.setSomeType( 1, param_1 );
   stmt.setSomeType( 2, param_2 );
   .
   .
   .
   // There are "m" parameters in the statement.
   stmt.setSomeType( m , param_m );
```

```
        // Add the set of parameters to the batch and
        // move to the next row of parameters.
        stmt.addBatch();
    }
```

例：

```
    for( i=0; i < 5; i++ )
    {
        stmt.setInt( 1, idValue );
        stmt.setString( 2, name );
        stmt.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。

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、小さなカラムサイズを使用することを推奨します。バッチ挿入を使用して、大きなバイナリデータや文字データを long binary カラムや 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 SQL Anywhere 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。これ以外の場合では、バッチ挿入を使用することで個別に挿入するよりも高いパフォーマンスを維持できます。

Java から結果セットを返す方法

この項では、Java メソッドから 1 つ以上の結果セットを取得する方法について説明します。

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();
}
```

注意

- このサーバ側 JDBC の例は、*%SQLANYSAMP16%¥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。
- 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。

JDBC から結果セットを返す

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、複数の結果セットを返します。

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。

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。

♦ タスク

1. Interactive SQL から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。

   Java のサンプルクラスをインストールする方法の詳細については、「JDBC サンプルの準備」439 ページを参照してください。
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。

   例：

   ```
   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。

JDBC に関する注意事項

- **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)。

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。これは、SQL Anywhere 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 sybase.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";
```

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

```
        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 sybase.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 4.0 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 *%SQLANY16%¥java¥sajdbc4.jar* を入れる必要があります。

```
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();
         ((sybase.jdbc4.sqlanywhere.IConnection)con).setASAMessageHandler( message_worker );

         // create and register validate file transfer callback
         T_filetrans_callback filetran_worker = new T_filetrans_callback();

((sybase.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();
        }
    }
}
```

参照

- 「MESSAGE 文」『SQL Anywhere サーバ SQL リファレンス』

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、SQL Anywhere 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、SQL Anywhere 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-string time-string は、SQL Anywhere 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、SQL Anywhere が受け取ることのできる有効な OUTER JOIN 式です。
- **{? = call func(p1,...)}** func は、SQL Anywhere 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、SQL Anywhere 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 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

数値関数	文字列関数	システム関数	日付／時刻関数
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
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			

エスケープ構文を使用している文は、SQL Anywhere、Adaptive Server Enterprise、Oracle、SQL Server、または接続されている他のデータベース管理システムで動作します。

Interactive SQL では、大カッコ ({) は必ず二重にしてください。カッコ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、SQL エスケープ構文を使用して sa_db_info プロシージャを持つデータベースプロパティを取得するには、InteractiveSQL で次のコマンドを実行します。

```
{{CALL sa_db_info( 0 ) }}
```

JDBC 4.0 API のサポート

SQL Anywhere JDBC ドライバでは、JDBC 4.0 仕様のすべての必須クラスとメソッドがサポートされています。java.sql.Blob インタフェースの一部のオプションメソッド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);

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 (Universal 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。

この項では、SQL Anywhere に対する OData のサポート (OData サーバ) の概要について示します。開発者が OData プロトコルの概念に精通していることを前提としています。

参照

- http://www.odata.org/documentation/IndexV2

OData サーバのアーキテクチャ

OData サーバは次のコンポーネントで構成されています。

- **OData プロデューサ**　「OData プロデューサ」 は、JDBC API を使用して SQL Anywhere 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 の要求と応答を処理し、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
エンティティタイプ	テーブルまたはビュー
エンティティタイプインスタンス	ロー
キー	プライマリキー
リンク	外部キー
プロパティ	カラム

- **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。

  OData サーバは Jetty WebServer をその HTTP サーバとして使用します。この 「Embedded 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 コンテナとしても機能します。

  ソリューションが Java Servlets のホストとしても機能する場合、独自の HTTP サーバを使用して、OData の要求を Embedded HTTP サーバの代替として処理できます。たとえば、IIS または Apache サーバが要求を Tomcat 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。

事前に設定された接続文字列または HTTP 基本認証のいずれかを使用して、クライアントにデータベースへのアクセスを付与できます。

参照

- 「OData サーバの実行とクライアントの設定」466 ページ
- http://www.odata.org/documentation/IndexV2

OData サーバの制限事項

OData サーバは、OData プロトコルの仕様によって明示的に定義されない次の制限事項がある OData プロトコルバージョン 2 の仕様に準拠しています。

- **スキーマの変更**　データベーススキーマを変更したら、変更が有効になり、OData クライアントにわかるようにするために、OData サーバユーティリティを再起動します。

- **$orderby クエリ**　エンティティのプロパティによってのみソートできます。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。

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。

- ディープ挿入と更新
- サービス操作
- 動的プロパティ
- 複合型
- メディア型
- HTTP Etag

参照

- 「OData サーバのセキュリティの考慮事項」458 ページ
- http://www.odata.org/documentation/IndexV2

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。

HTTPS 証明書

OData サーバ設定ファイルで指定された HTTPS 証明書の詳細は、Embedded HTTP サーバにのみ適用されます。HTTPS 証明書が代替 HTTP サーバで処理されるしくみに関する詳細については、HTTP サーバのマニュアルを参照してください。

SSL 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常に SSL を使用することをおすすめします。

SSLKeyStore オプションが OData サーバ設定ファイルで指定されていない場合、ユーザ ID とパスワードを含む、OData プロデューサとクライアントの間のすべてのトラフィックは、プレーンテキストで送信されます。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。

SQL Anywhere バージョン 12 以前のデータベース

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。

参照

- 「OData サーバを設定する方法」459 ページ
- 「OData サーバユーティリティ (dbosrv16.exe)」466 ページ

- http://www.odata.org/documentation/IndexV2
- http://wiki.eclipse.org/Jetty
- http://docs.codehaus.org/display/JETTY/Jetty+Wiki

OData サーバを設定する方法

データベース接続パラメータ、Embedded HTTP サーバオプション、OData プロデューサオプションを設定ファイルに設定します。OData サーバを起動したら、コマンドラインでこのファイルを指定します。

Embedded HTTP サーバオプション

次の Embedded HTTP サーバオプションを設定ファイルで指定できます。

オプション	説明
LogFile = *path-and-filename*	Embedded HTTP サーバが OData プロデューサの出力をロギングするパスとファイル名を指定します。 デフォルトの動作では、ロギングが無効になります。 パスは、サーバの実行プログラムの場所に対する相対パスです。

オプション	説明
LogVerbosity = { **1** | **2** | **3** | **4** }	ログの冗長性レベルを指定します。冗長性レベルが高い場合、追加情報がロギングされ、すべてのレベルで提供される情報が含まれます。 冗長性レベル 1 は、予期しないエラーに関する情報を出力します。 冗長性レベル 2 は、一般的な情報と設定メッセージを出力します。 冗長性レベル 3 は、HTTP 要求に関する詳細な情報を出力します。 冗長性レベル 4 は、デバッグメッセージを出力します。
ServerPort = *port-number*	Embedded HTTP サーバが受信するポート番号を指定します。 デフォルト設定値は 80 です。
ShutdownListenerPort = *port-number*	Embedded HTTP サーバがシャットダウン要求を受信するポート番号を指定します。 デフォルト設定値は 2449 です。
SSLKeyStore = *path-and-filename*	Embedded HTTP サーバがトラフィックを暗号化するために使用する SSL 証明書が含まれる Java Key Store キーストアに、パスとファイル名を指定します。 このオプションが指定されている場合、SSL が有効になり、暗号化されていない HTTP トラフィックがブロックされます。 パスは、サーバの実行プログラムの場所に対する相対パスです。
SSLKeyStorePassword = *SSLKeyStore-password*	SSLKeyStore オプションによって識別された Java キーストアに対して認証するために Embedded HTTP サーバが使用するパスワードを指定します。

OData プロデューサオプションとデータベース接続パラメータ設定

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。

オプション	説明
Authentication = { **none** | **database** }	データベースに接続するための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。
ConnectionPoolMaximum = *num-max-connections*	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によって許可される同時接続の最大数によって制限されています。
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。
DbProduct = sqlanywhere	OData プロデューサが接続するデータベースサーバのタイプを示します。
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。 デフォルト設定値は 100 です。

オプション	説明
Model = *path-and-filename*	OData メタデータ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。 デフォルトの動作では、テーブルとビューをユーザ権限に基づいて公開します。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。 パスは、サーバの実行プログラムの場所に対する相対パスです。
ModelConnectionString = *connection-string*	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、エンリストされている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** パラメータが含まれている必要があります。 デフォルトの動作では、OSDL ファイルが有効であることを前提にしています。
ReadOnly = { **true** | **false** }	変更要求を無視するかどうかを示します。 デフォルト設定は false です。
ServiceRoot = / *path-prefix*	OData サーバで OData サービスのルートを指定します。 デフォルト設定は **/odata** です。 この OData プロデューサのすべてのリソースには、次のフォーマットの URI を使用してアクセスします。 *scheme*:*host*:*port*/*path-prefix*/*resource-path*[*query-options*] 例：http://services.odata.org/OData/OData.svc/Category(1)/Products?$top=2&$orderby=name

参照

- 「OData サーバユーティリティ (dbosrv16.exe)」466 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」465 ページ

例

次の例は、OData サーバで使用する設定ファイルをフォーマットする方法について説明します。

```
# Embedded HTTP server options
# ----------------------------
LogFile = ../../odata.log
LogVerbosity = 1
ServerPort = 8000
ShutdownListenerPort = 8083
SSLKeyStore = ../../samplekeystore.jks
SSLKeyStorePassword = pwd

# OData Producer options
# ----------------------
Authentication = none
ConnectionPoolMaximum = 10
DbProduct = sqlanywhere
Model = ../../model.osdl
ModelConnectionString = uid=dba;pwd=sql;eng=orderentry;dbf=orderentry.db
PageSize = 100
ReadOnly = false
ServiceRoot = /odata/MyProducer

# Database connection parameters
# ------------------------------
DbConnectionString = uid=dba;pwd=sql;eng=orderentry;dbf=orderentry.db
```

ファイルを *server.properties* で保存し、次のコマンドを実行して、その設定ファイルで指定されたオプションで OData サーバを起動します。

```
dbosrv16 server.properties
```

OData の操作のために HTTP サーバを設定する方法

Embedded HTTP サーバまたは独自の HTTP サーバのいずれかを設定できます。

> **注意**
> セキュリティ対策は、HTTP サーバを設定する場合に考慮する必要があります。 詳細については、「OData サーバのセキュリティの考慮事項」458 ページを参照してください。

Embedded HTTP サーバの設定

OData サーバユーティリティは Embedded HTTP サーバのインスタンスを開始し、OData プロデューサを Java Servlet として自動的にロードします。

Embedded HTTP サーバは、HTML ファイルなどの他の OData 以外の内容のために手動で設定または使用できません。ただし、一部の HTTP サーバオプションは、OData サーバ設定ファイルで指定できます。

Embedded HTTP サーバを設定して起動するには、次の手順が必要です。

1. OData プロデューサと Embedded HTTP サーバの両方の設定が含まれる OData サーバ設定ファイルを作成します。

2. HTTP サーバとして動作するように設定ファイルをコンピュータに保存し、コマンドプロンプトで OData サーバユーティリティを実行するときに設定ファイルをロードします。

> **注意**
> OData の終了ポイントは、Embedded HTTP サーバを使用する場合に、**odata** コンテキストパスに配置されます。
>
> たとえば、ポート 8080 で実行されているローカルホスト HTTP サーバに対して OData 要求を実行するには、http://localhost:8080/odata を指定します。

代替 HTTP サーバの設定

OData プロデューサを代替 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、HTTP サーバにロードできる Java Servlet として実行する必要があります。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。

> **注意**
> IIS は Java Servlets 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 プロデューサを正常に配備するには、次の手順が必要です。

1. OData サーバ設定ファイルを作成します。

   セキュリティの考慮事項などの Embedded HTTP サーバに固有の設定オプションは、代替 HTTP サーバには適用されません。

   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. 次のファイルとともに、*%SQLANY16%¥Java¥* ディレクトリにある OData サーバ設定ファイルを配備します。

   ●*dbodata.jar*

3. JDBC クライアントを配備します。これには SQL Anywhere JDBC 4.0 ドライバとサポートファイルが含まれます。

4. Java EE HttpServlet インタフェースを実装する SQLAnywhereODataServlet クラスを HTTP サーバにロードします。

参照

- 「JDBC クライアントの配備」994 ページ
- 「OData サーバユーティリティ (dbosrv16.exe)」466 ページ
- 「OData サーバを設定する方法」459 ページ

- http://docs.oracle.com/javaee/1.3/api/javax/servlet/http/package-summary.html

OData プロデューサのサービスモデルを作成する方法

OData プロデューサのサービスモデルを作成し、OData Service Definition Language (OSDL) ファイルを指定して、特定のテーブルとビューを公開します。

OSDL 構文

service [**namespace** "*namespace-name*"] **{**
  "*owner*"."{*table-name* | *view-name*}" [**keys(** "*column-name*", ... **)**];
  ...
}

パラメータ

- ***namespace-name*** 定義するネームスペース。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。
- ***table-name*** OData メタデータで公開するテーブルの名前。
- ***view-name*** OData メタデータで公開するビューの名前。
- ***column-name*** *table-name* または *view-name* によって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。

  複数のカラム名を参照できます。

備考

複数のテーブルまたはビューの参照を含むサービスを定義できます。

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ません。

OData サーバ設定ファイルで **Model** オプションを使用するテキストファイルを参照する場合、モデルは OData プロデューサに適用されます。

参照

- 「OData サーバを設定する方法」459 ページ
- 「OData サーバユーティリティ (dbosrv16.exe)」466 ページ
- 「OData サーバの実行とクライアントの設定」466 ページ

例

次は、テーブルとビューを公開するサービスモデルの例です。

```
service namespace "DBServer" {
"dba"."TableWithPrimaryKey";
"dba"."ViewWithoutPrimaryKey" keys("primary_key1", "primary_key2");
}
```

OData サーバの実行とクライアントの設定

OData サーバのサンプルは、*%SQLANYSAMP16%¥SQLAnywhere¥* ディレクトリで使用できます。サンプルは OData サーバを実行し、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。

.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、SQL Anywhere サンプルデータベースに接続される OData サーバに OData 要求を送信する方法を説明します。

詳細については、*%SQLANYSAMP16%¥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、SQL Anywhere サンプルデータベースに接続される OData 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、*%SQLANYSAMP16%¥SQLAnywhere¥ODataSecurity¥readme.txt* を参照してください。

参照

- 「OData サーバユーティリティ (dbosrv16.exe)」466 ページ
- 「Windows 用サービスユーティリティ (dbsvc)」『SQL Anywhere サーバ データベース管理』

OData サーバユーティリティ (*dbosrv16.exe*)

Embedded HTTP サーバを起動し、OData プロデューサのホストとして機能します。

構文

dbosrv16 *properties-filename*

オプション	説明
properties-filename	設定ファイルで指定されたオプションを使用して、データベース接続を設定します。

備考

OData サーバユーティリティには、HTTP サーバを起動するのに必要なサーブレット実装とコンポーネントが含まれています。

OData サーバ停止ユーティリティを使用して OData サーバを再起動し、変更が有効になり、OData クライアントにわかるようにするために、データベーススキーマを変更します。

-t OData オプションが指定された状態で dbsvc ユーティリティを実行し、OData サーバユーティリティをサービスとして起動できます。

参照

- 「OData の操作のために HTTP サーバを設定する方法」463 ページ
- 「OData サーバを設定する方法」459 ページ
- 「OData サーバ停止ユーティリティ (dbostop.exe)」467 ページ
- 「Windows 用サービスユーティリティ (dbsvc)」『SQL Anywhere サーバ データベース管理』

OData サーバ停止ユーティリティ (*dbostop.exe*)

Embedded HTTP サーバと OData プロデューサを停止します。

構文

dbostop [**-q**] { **-f** *properties-filename* | **-p** *port-number* }

オプション	説明
-f *properties-filename*	ShutdownListenerPort オプションによって指定されたポート番号を使用して、シャットダウン要求を送信します。
-p *port-number*	シャットダウン要求を送信するポート番号を指定します。デフォルト値は 2449 です。
-q	サーバのクワイエットモードでのシャットダウンを試みます。メッセージを表示しません。

備考

-p または **-f** オプションが提供されていない場合、このユーティリティはポート 2449 で Embedded HTTP サーバと OData プロデューサをシャットダウンしようとします。

シャットダウンポート番号は、OData サーバを起動するために使用された OData サーバ設定ファイルで指定された ShutdownListenerPort オプションと同じである必要があります。

変更を有効にして OData クライアントにわかるようにするためにデータベーススキーマを変更する場合、OData サーバ停止ユーティリティを使用して、OData サーバを再起動します。

参照

- 「OData サーバを設定する方法」459 ページ
- 「OData サーバユーティリティ (dbosrv16.exe)」466 ページ

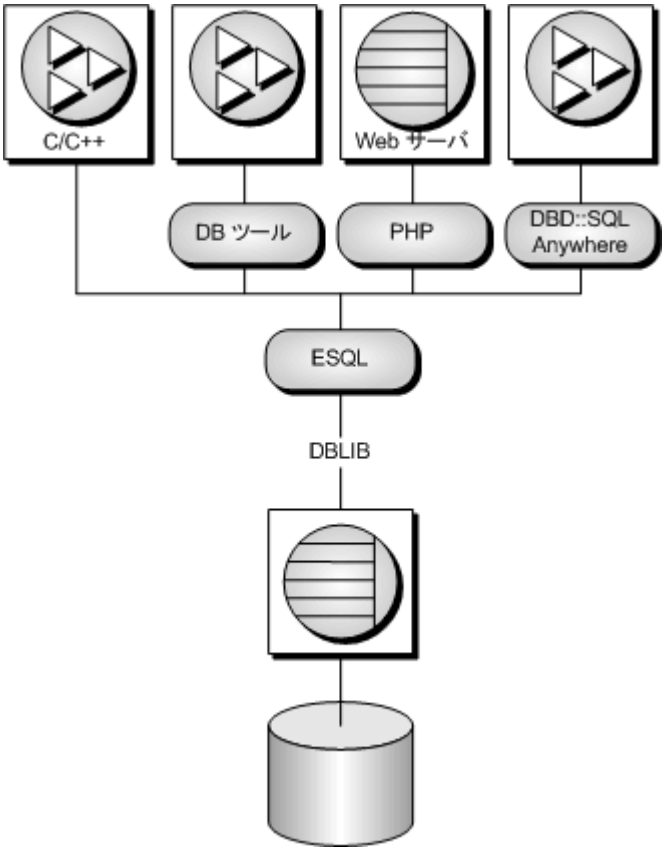
Embedded SQL

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

SQL Anywhere Embedded SQL インタフェースを使用して SQL Anywhere 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「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 SQL Anywhere 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。

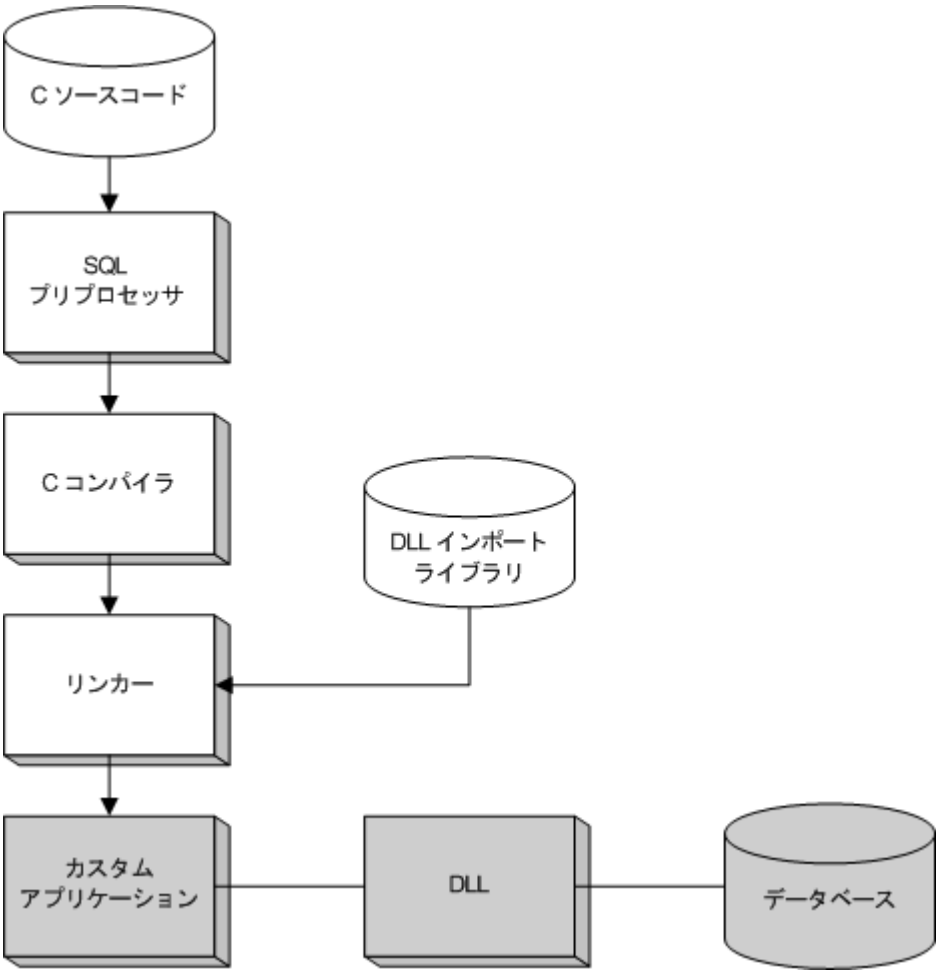
●Windows オペレーティングシステムでは、インタフェースライブラリは *dblib16.dll* です。

●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、*libdblib16.so*、*libdblib16.sl*、または *libdblib16.a* です。

●Mac OS X では、インタフェースライブラリは *libdblib16.dylib.1* です。

SQL Anywhere には、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。

開発プロセスの概要



プリプロセッサ処理とコンパイルが成功すると、プログラムは DBLIB 用の「インポートライブラリ」とリンクされ、実行ファイルになります。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やりとりをします。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実行され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して、インポートライブラリの使用を避けることをおすすめします。

参照

- 「Windows での DBLIB の動的ロード」479 ページ

SQL プリプロセッサ

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拡張子 *.c* が付いた *sql-filename* です。*sql-filename* にすでに *.c* 拡張子が付いている場合、出力ファイルのデフォルトの拡張子は *.cc* に変更されます。

注意
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。

オプション	説明
-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、使用する標準を表します。たとえば、sqlpp -e c03 ... は、コア SQL/2008 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 SQL Anywhere 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* は次のいずれかである可能性があります。 ● **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。

オプション	説明
-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のとは対照的です。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** Windows Mobile を含む 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ない)
-r-	再入力不可コードを生成します。
-s *len*	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。

オプション	説明
-w *level*	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、使用する標準を表します。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 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 SQL Anywhere 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
-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。
-z *cs*	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	SQL プリプロセッサが作成した C 言語のソースファイル。

参照

- 「Embedded SQL」469 ページ
- 「sql_flagger_error_level オプション」『SQL Anywhere サーバ データベース管理』
- 「sql_flagger_warning_level オプション」『SQL Anywhere サーバ データベース管理』
- 「SQLFLAGGER 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_ansi_standard_packages システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「SQL プリプロセッサのエラーメッセージ」『エラーメッセージ』
- 「マルチスレッドまたは再入可能コードでの SQLCA 管理」500 ページ
- 「Ultra Light Embedded SQL API リファレンス」『Ultra Light C/C++ プログラミング』
- 「SACHARSET 環境変数」『SQL Anywhere サーバ データベース管理』
- 「SALANG 環境変数」『SQL Anywhere サーバ データベース管理』

対応コンパイラ

C 言語 SQL プリプロセッサは、これまでに次のコンパイラで使用されてきました。

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
Windows Mobile	Microsoft Visual C++	2005
UNIX	GNU またはネイティブコンパイラ	

Embedded SQL ヘッダファイル

ヘッダファイルはすべて、SQL Anywhere インストールディレクトリの *SDK¥Include* サブフォルダ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。
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。

インポートライブラリ

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、SQL Anywhere インストールディレクトリの *SDK¥Lib* サブフォルダ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* サブフォルダに格納されています。Windows Mobile 用のインストールライブラリは、*SDK¥Lib¥CE¥Arm.50* サブフォルダにインストールされています。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、SQL Anywhere インストールディレクトリの *lib32* および *lib64* サブフォルダ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、SQL Anywhere インストールディレクトリの *System/lib32* および *System/lib64* サブフォルダ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*
Windows Mobile	Microsoft Visual C++ 2005	*dblib16.lib*
UNIX (非スレッドアプリケーション)	全コンパイラ	*libdblib16.so*、*libdbtasks16.so*、*libdblib16.sl*、*libdbtasks16.sl*
UNIX (スレッドアプリケーション)	全コンパイラ	*libdblib16_r.so*、*libdbtasks16_r.so*、*libdblib16_r.sl*、*libdbtasks16_r.sl*
Mac OS X (スレッドアプリケーション)	全コンパイラ	*libdblib16.dylib*、*libdbtasks16.dylib*
Mac OS X (スレッドアプリケーション)	全コンパイラ	*libdblib16_r.dylib*、*libdbtasks16_r.dylib*

libdbtasks16 ライブラリは、*libdblib16* ライブラリに呼び出されます。コンパイラの中には、*libdbtasks16* 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。

サンプル Embedded SQL プログラム

Embedded SQL プログラムの非常に簡単な例を次に示します。

```
#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、終了しています。この例では、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。

参照

- 「Embedded SQL を使用してデータをフェッチする方法」515 ページ

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。ESQL 文の途中に、通常の C 言語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 文の認可に使用されます。

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 );
```

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、インストールディレクトリの *SDK¥C* サブフォルダにある *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。

前提条件

この作業を実行するための前提条件はありません。

内容と備考

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。

♦ タスク

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、適切なマクロを特定して、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。Windows で定義が必要となるマクロを次に示します。

マクロ	プラットフォーム
_SQL_OS_WINDOWS	すべての Windows オペレーティングシステム

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、*%SQLANYSAMP16%¥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 %SQLANY16%¥sdk¥c¥esqldll.c /I%SQLANY16%¥sdk¥include Advapi32.lib
```

サンプル Embedded SQL プログラム

SQL Anywhere をインストールすると、Embedded SQL のサンプルプログラムがインストールされます。サンプルプログラムは *%SQLANYSAMP16%¥SQLAnywhere¥C* ディレクトリにあります。Windows Mobile の場合は、追加サンプルが *¥SQLAnywhere¥CE¥esql_sample* 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、カーソルを開く。
- **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する。
- **WSQLEX_Finish**　カーソルを閉じ、データベースとの接続を切断する。

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、ループする。
3. WSQLEX_Finish ルーチンを呼び出す。

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、SQL Anywhere 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。

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。

参照

- 「静的カーソルのサンプルプログラムの実行」482 ページ
- 「動的カーソルのサンプル」483 ページ

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。

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、Visual Studio の *vcvars32.bat* または *vcvars64.bat* (Visual Studio の旧バージョンでは *vcvarsamd64.bat*) を使用して行います。

内容と備考

実行ファイルと対応するソースコードは *%SQLANYSAMP16%¥SQLAnywhere¥C* ディレクトリにあります。Windows Mobile の場合は、追加サンプルが *¥SQLAnywhere¥CE¥esql_sample* ディレクトリにあります。

♦ **タスク**

1. SQL Anywhere 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
   ```

   Windows Mobile の場合は、Microsoft Visual C++ 用の *esql_sample.sln* プロジェクトファイルを使用してください。このファイルは *¥SQLAnywhere¥CE¥esql_sample* にあります。

   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。

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、*%SQLANYSAMP16%¥SQLAnywhere¥C* ディレクトリにあります。Windows Mobile の場合は、動的カーソルのサンプルが *¥SQLAnywhere¥CE ¥esql_sample* 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。

注意
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。

参照

- 「動的カーソルのサンプルプログラムの実行」484 ページ
- 「静的カーソルのサンプル」482 ページ

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。

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、Visual Studio の *vcvars32.bat* または *vcvars64.bat* (Visual Studio の旧バージョンでは *vcvarsamd64.bat*) を使用して行います。

内容と備考

実行ファイルと対応するソースコードは *%SQLANYSAMP16%¥SQLAnywhere¥C* ディレクトリにあります。Windows Mobile の場合は、追加サンプルが *¥SQLAnywhere¥CE¥esql_sample* ディレクトリにあります。

♦ タスク

1. SQL Anywhere 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
   ```

   Windows Mobile の場合は、Microsoft Visual C++ 用の *esql_sample.sln* プロジェクトファイルを使用してください。このファイルは *¥SQLAnywhere¥CE¥esql_sample* にあります。

   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。

   Windows Mobile の例では、Windows Mobile デバイスでファイル *esql_sample.exe* を配備して実行します。

   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 16 Demo
   ```
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。

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
- **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
- **DT_INT**　32 ビット符号付き整数
- **DT_UNSINT**　32 ビット符号なし整数
- **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。最大長は 32765 バイトです。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。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。データは NULL で終了せず、ブランクも埋め込まれません。

```
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。最大長は 32765 バイトです。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。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。データは NULL で終了せず、ブランクも埋め込まれません。

```
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。最大長は 32765 バイトです。データベースサーバに情報を渡す場合は、長さフィールドを設定してください。データベースサーバから情報をフェッチする場合は、サーバが長さフィールドを設定します。

```
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が、この構造体を使ってデータをフェッチすると、以後の操作が簡単になります。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。

参照

- 「GET DATA 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「Embedded SQL を使用して long 値を送信し、取得する方法」523 ページ
- 「date_format オプション」『SQL Anywhere サーバ データベース管理』
- 「date_order オプション」『SQL Anywhere サーバ データベース管理』
- 「time_format オプション」『SQL Anywhere サーバ データベース管理』
- 「timestamp_format オプション」『SQL Anywhere サーバ データベース管理』

Embedded SQL のホスト変数

ホスト変数とは、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。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。

参照

- 「静的 SQL と動的 SQL」503 ページ

ホスト変数の宣言

ホスト変数は、「宣言セクション」で定義します。ANSI Embedded SQL 標準では、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、どの SQL 文でも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。ホスト変数は、SQL 文の中では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。

SQL プリプロセッサは、DECLARE SECTION 内でのみ C 言語コードをスキャンします。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 );
```

参照

● 「静的カーソルのサンプル」482 ページ

C ホスト変数型

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) 型を使用しないことをおすすめします。

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
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
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
float f;	DT_FLOAT	4 バイトの単精度浮動小数点値
double d;	DT_DOUBLE	8 バイトの倍精度浮動小数点値
char a[n]; /*n>=1*/	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。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。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。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。

C データ型	Embedded SQL のインタフェースのデータ型	説明
DECL_NCHAR *a;	DT_NSTRING	NCHAR 文字セット内の NULL で終了する文字列。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。文字列は NULL で終了せず、ブランクも埋め込まれない。n の最大値は 32765 (バイト単位) です。
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。文字列は NULL で終了せず、ブランクも埋め込まれない。n の最大値は 32765 (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。文字列は NULL で終了せず、ブランクも埋め込まれない。
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。文字列は NULL で終了せず、ブランクも埋め込まれない。
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。n の最大値は 32765 (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。

C データ型	Embedded SQL のインタフェースのデータ型	説明
char a; /*n=1*/ DECL_FIXCHAR(n) a;	DT_FIXCHAR	CHAR 文字セット内の固定長文字列。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。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

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、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。

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。

参照

- 「ロケール」『SQL Anywhere サーバ データベース管理』
- 「CharSet (CS) 接続パラメータ」『SQL Anywhere サーバ データベース管理』
- 「db_change_char_charset 関数」539 ページ
- 「db_change_nchar_charset 関数」539 ページ
- 「GET DATA 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』

ホスト変数の使用法

ホスト変数は次の場合に使用できます。

- SELECT、INSERT、UPDATE、DELETE 文で数値定数または文字列定数を書ける場所。
- SELECT、FETCH 文の INTO 句。
- ホスト変数は、文名、カーソル名、Embedded SQL 固有の文のオプション名としても使用できます。
- 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、接続文字列として使用できます。
- 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 標準を使用することで、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];
```

使用する場合、これらの変数が設定されるのは、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) の後になります。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 ESQL 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...
}
```

SQL プリプロセッサの -k オプションでは、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。

参照

- 「SQL プリプロセッサ」471 ページ

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 値を扱えるようにする。
- **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おさ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する。

- **変換エラー**　エラー情報を保持する。

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。

> **注意**
> 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 での short int の使用は推奨されなくなりました。a_sql_len を代わりに使用します。

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 言語の定数は、ポイント先がメモリのロケーションではないポインタ値を表します。

SQL Anywhere のマニュアルで使用されている NULL の場合は、上記のような SQL データベースを指します。C 言語の定数を指す場合は、null ポインタ (小文字) のように表記されます。

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)。

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。

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。

インジケータ変数値のまとめ

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る。インジケータ変数は実際の長さを示す。
0	ホスト変数値	フェッチが成功、または conversion_error が On に設定されている。
-1	NULL 値	NULL 結果。

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	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 警告を示す。
< -2	NULL 値	NULL 結果。

参照

- 「GET DATA 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』

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、ESQL 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。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、インストールディレクトリの *SDK¥Include* サブフォルダにある *sqlca.h* ヘッダファイルで定義されています。

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。

SQLCA のフィールド

SQLCA のフィールドの意味を次に示します。

- **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、「**テーブルが見つかりません**」の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。
- **sqlerrd**　32 ビット整数の汎用配列。
- **sqlwarn**　予約。
- **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によって計算されたローの数は、ローの実際の数です。ローを数える必要はありません。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。

参照

- 「SQL Anywhere のエラーメッセージ」『エラーメッセージ』
- 「SQL Anywhere のエラーメッセージ (SQLCODE 順)」『エラーメッセージ』
- 「SQL Anywhere のエラーメッセージ (SQLSTATE 順)」『エラーメッセージ』
- 「ワイドフェッチまたは配列フェッチ」519 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

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、スレッドでは db_cancel_request 関数を使用して、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 );
}
```

参照

- 「Embedded SQL を使用した要求管理」531 ページ

複数の SQLCA

再入力不可コードを生成する 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、少しだけサイズが大きく、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、SQL プリプロセッサにデータベース要求で別の SQLCA を使用することを伝えます。通常は、EXEC SQL SET SQLCA 'task_data->sqlca'; のような文をプログラムの先頭かヘッダファイルに置いて、SQLCA 参照がそのタスクに特定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。この条件は、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 SQLCA のサポートは、サポートされるどの Embedded SQL 環境でも使用できますが、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。

注意
異なる接続に対する操作では通常のレコードロックメカニズムが使用され、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。

参照

- 「SET SQLCA 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「SET CONNECTION 文 [Interactive SQL] [ESQL]」『SQL Anywhere サーバ SQL リファレンス』

静的 SQL と動的 SQL

SQL 文を C プログラムに埋め込むには次の 2 つの方法があります。

- 静的文
- 動的文

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。

参照

- 「Embedded SQL のホスト変数」489 ページ

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。動的文は C 言語の文字列で構成されます。この文は PREPARE 文と EXECUTE 文を使用して実行できます。この SQL 文は静的文と同じようにして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」) 構造体が使用されます。EXECUTE 文の USING 句を使ってホスト変数のリストを指定すると、SQL プリプロセッサが自動的にこの構造体を用意します。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出し入れするロケーションを記述します。

- データ型
- *型*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。

参照

- 「SQLDA (SQL descriptor area)」506 ページ
- 「EXECUTE 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「インジケータ変数」495 ページ

動的 SELECT 文

シングルローだけを返す 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、値用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;
```

注意
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。

参照

- 「PREPARE 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「DESCRIBE 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「動的カーソルのサンプル」483 ページ
- 「ライブラリ関数のリファレンス」532 ページ

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。

参照

- 「ライブラリ関数のリファレンス」532 ページ

SQLDA ヘッダファイル

sqlda.h の内容は次のとおりです。

```
#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
```

SQLDA のフィールド

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。

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。

参照

- 「Embedded SQL のデータ型」485 ページ
- 「SQLDA の sqllen フィールドの値」509 ページ
- 「conversion_error オプション」『SQL Anywhere サーバ データベース管理』
- 「インジケータ変数」495 ページ

SQLDA の sqllen フィールドの値

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。
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 に設定される。sqlda.h の PRECISION および SCALE マクロを参照してください。
DOUBLE	DT_DOUBLE	8

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)
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

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)	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。
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。

参照

- 「Embedded SQL のデータ型」485 ページ
- 「CHAR データ型」『SQL Anywhere サーバ SQL リファレンス』
- 「NCHAR データ型」『SQL Anywhere サーバ SQL リファレンス』
- 「NVARCHAR データ型」『SQL Anywhere サーバ SQL リファレンス』
- 「VARCHAR データ型」『SQL Anywhere サーバ SQL リファレンス』
- 「db_change_nchar_charset 関数」539 ページ

値を送信するときの SQLDA の sqllen フィールドの値

次の表は、SQLDA においてデータベースサーバにデータを渡すとき、値の長さをどう指定するかを示します。

この場合は、表で示したデータ型だけを使用できます。DT_DATE、DT_TIME、DT_TIMESTAMP 型は、データベースに情報を渡すときは、DT_STRING 型と同じものとして扱われます。値は、NULL で終了する適切な日付または時刻形式の文字列にしてください。

Embedded SQL のデータ型	長さを設定するプログラム動作
DT_BIGINT	動作不要
DT_BINARY(n)	BINARY 構造体の長さフィールドから取る
DT_BIT	動作不要
DT_DATE	末尾の null 文字によって長さが決まる
DT_DOUBLE	動作不要
DT_FIXCHAR(n)	SQLDA の長さフィールドが文字列の長さを決定する
DT_FLOAT	動作不要
DT_INT	動作不要
DT_LONGBINARY	長さフィールドが無視される。
DT_LONGNVARCHAR	長さフィールドが無視される。
DT_LONGVARCHAR	長さフィールドが無視される。
DT_NFIXCHAR(n)	SQLDA の長さフィールドが文字列の長さを決定する
DT_NSTRING	末尾の ¥0 によって長さが決まる。ansi_blanks オプションが On に設定されていて、データベースでブランクが埋め込まれる場合、SQLDA の長さフィールドは、値が含まれているバッファの長さ (少なくても値の長さと末尾の NULL 文字の分を足した長さ) に設定される。
DT_NVARCHAR	NVARCHAR 構造体の長さフィールドから取る
DT_SMALLINT	動作不要
DT_STRING	末尾の ¥0 によって長さが決まる。ansi_blanks オプションが On に設定されていて、データベースでブランクが埋め込まれる場合、SQLDA の長さフィールドは、値が含まれているバッファの長さ (少なくても値の長さと末尾の NULL 文字の分を足した長さ) に設定される。
DT_TIME	末尾の null 文字によって長さが決まる
DT_TIMESTAMP	末尾の null 文字によって長さが決まる

Embedded SQL のデータ型	長さを設定するプログラム動作
DT_TIMESTAMP_STRUCT	動作不要
DT_UNSBIGINT	動作不要
DT_UNSINT	動作不要
DT_UNSSMALLINT	動作不要
DT_VARCHAR(n)	VARCHAR 構造体の長さフィールドから取る
DT_VARIABLE	末尾の ¥0 によって長さが決まる

参照

- 「動的 SQL を使用した LONG データの送信」527 ページ

データを取得するときの SQLDA の sqllen フィールドの値

次の表は、SQLDA を使用してデータベースからデータを取り出すときの、長さフィールドの値を示します。データを取り出すときには、sqllen フィールドは変更されません。

この場合に使用できるのは、表で示したインタフェースデータ型だけです。DT_DATE、DT_TIME、DT_TIMESTAMP 型はデータベースから情報を取り出すときは 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したが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	データを受け取るときにプログラムが長さフィールドに設定する値	値をフェッチした後、データベースが長さ情報を返す方法
DT_BIGINT	動作不要	動作不要
DT_BINARY(n)	BINARY 構造体の最大長 (n +2)。n の最大値は 32765 です。	BINARY 構造体の len フィールドに実際の長さをバイト単位で設定
DT_BIT	動作不要	動作不要
DT_DATE	バッファの長さ	文字列の末尾の null 文字
DT_DOUBLE	動作不要	動作不要
DT_FIXCHAR(n)	バッファの長さ (バイト)。n の最大値は 32767 です。	バッファの長さまでブランクを埋め込む
DT_FLOAT	動作不要	動作不要
DT_INT	動作不要	動作不要

Embedded SQL のデータ型	データを受け取るときにプログラムが長さフィールドに設定する値	値をフェッチした後、データベースが長さ情報を返す方法
DT_LONGBINARY	長さフィールドが無視される。	長さフィールドが無視される。
DT_LONGNVARCHAR	長さフィールドが無視される。	長さフィールドが無視される。
DT_LONGVARCHAR	長さフィールドが無視される。	長さフィールドが無視される。
DT_NFIXCHAR(n)	バッファの長さ (バイト)。n の最大値は 32767 です。	バッファの長さまでブランクを埋め込む
DT_NSTRING	バッファの長さ	文字列の末尾の null 文字
DT_NVARCHAR(n)	NVARCHAR 構造体の最大長 (n+2)。n の最大値は 32765 です。	NVARCHAR 構造体の len フィールドに実際の長さをバイト単位で設定
DT_SMALLINT	動作不要	動作不要
DT_STRING	バッファの長さ	文字列の末尾の null 文字
DT_TIME	バッファの長さ	文字列の末尾の null 文字
DT_TIMESTAMP	バッファの長さ	文字列の末尾の null 文字
DT_TIMESTAMP_ STRUCT	動作不要	動作不要
DT_UNSBIGINT	動作不要	動作不要
DT_UNSINT	動作不要	動作不要
DT_UNSSMALLINT	動作不要	動作不要
DT_VARCHAR(n)	VARCHAR 構造体の最大長 (n+2)。n の最大値は 32765 です。	VARCHAR 構造体の len フィールドに実際の長さをバイト単位で設定

参照

- 「動的 SQL を使用した LONG データの取得」525 ページ

Embedded SQL を使用してデータをフェッチする方法

ESQL 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。

参照

- 「FETCH 文 [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』

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 */
  }
}
```

参照

- 「SQLCA (SQL Communication Area)」498 ページ

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. 「**ローが見つかりません**」という警告が返されるまで、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;
}
```

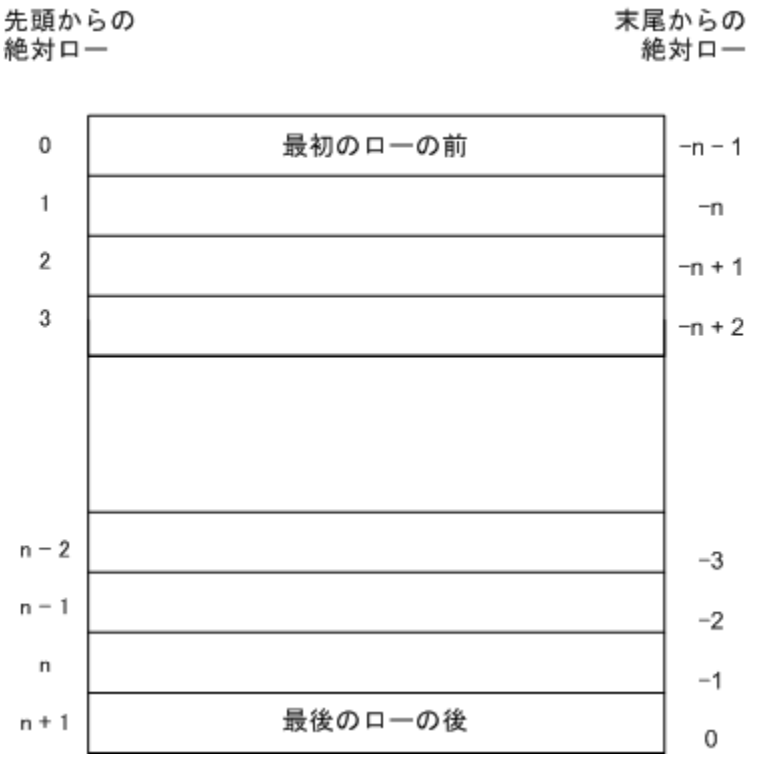
カーソル位置

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。

SQL Anywhere では、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)。

参照

- 「CLOSE 文 [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』
- 「DECLARE CURSOR 文 [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』
- 「DELETE 文 (位置付け) [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』
- 「FETCH 文 [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』
- 「OPEN 文 [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』
- 「PUT 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「UPDATE (位置付け) 文 [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』
- 「カーソルの原則」8 ページ
- 「静的カーソルのサンプル」482 ページ
- 「動的カーソルのサンプル」483 ページ
- 「SQLCA (SQL Communication Area)」498 ページ
- 「ヒント：クエリ処理におけるワークテーブルの使用 (All-rows 最適化ゴールの使用)」『SQL Anywhere サーバ SQL の使用法』

ワイドフェッチまたは配列フェッチ

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。

SQL Anywhere は、ワイドプットとワイド挿入もサポートします。

Embedded SQL でワイド・フェッチを使用するには、コードに次のような FETCH 文を含めます。

```
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、エラーではなくて 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。このコードは *%SQLANYSAMP16%¥SQLAnywhere ¥esqlwidefetch¥widefetch.sqc* にもあります。

```
#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 ) ) );
}
static SQLDA * PrepareSQLDA(
  a_sql_statement_number stat0,
  unsigned  width,
  unsigned  *cols_per_row )

/* Allocate a SQLDA to be used for fetching from
   the statement identified by "stat0". "width"
   rows are retrieved on each FETCH request.
   The number of columns per row is assigned to
   "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 )
{
  /* Print rows already wide fetched in the SQLDA */
  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 )
{
  /* Wide Fetch "query_str0" select statement
   * using a width of "fetch_width0" rows" */
  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 );
}
void main( int argc, char *argv[] )
{
  /* Optional first argument is a select statement,
   * optional second argument is the fetch width */
  char *query_str =
    "SELECT GivenName, Surname FROM Employees";
  unsigned fetch_width = 10;

  if( argc > 1 )
  {
    query_str = argv[1];
    if( argc > 2 )
    {
      fetch_width = atoi( argv[2] );
      if( fetch_width < 2 )
      {
        fetch_width = 2;
      }
    }
  }
  db_init( &sqlca );
  EXEC SQL CONNECT "DBA" IDENTIFIED BY "sql";

  DoQuery( query_str, fetch_width );

  EXEC SQL DISCONNECT;
err:
  db_fini( &sqlca );
}
```

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 関数とは違って、インジケータ変数用の領域が確保できます。

- 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指定されていない場合は、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。残りの SQLDA 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みません。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。

参照

- 「EXECUTE 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「FETCH 文 [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』
- 「PUT 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「alloc_sqlda 関数」532 ページ
- 「alloc_sqlda_noind 関数」533 ページ

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には、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]; ¥
  }
```

動的 SQL 構造体

動的 SQL の場合は、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。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。

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。

前提条件

この作業を実行するための前提条件はありません。

♦ タスク

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 メンバー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。SQL Anywhere によって次の情報が設定されます。
   - **インジケータ変数**　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。

- **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。

結果

LONG データは静的 SQL を使用して取得されます。

参照

- 「Embedded SQL を使用して long 値を送信し、取得する方法」523 ページ
- 「インジケータ変数」495 ページ

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。

前提条件

この作業を実行するための前提条件はありません。

♦ タスク

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。SQL Anywhere によって次の情報が設定されます。
   - *** sqlind**　この sqlda フィールドは、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。
   - **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。

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 );
}
```

参照

- 「Embedded SQL を使用して long 値を送信し、取得する方法」523 ページ
- 「インジケータ変数」495 ページ

静的 SQL を使用した LONG データの送信

ESQL 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。

前提条件

この作業を実行するための前提条件はありません。

♦ タスク

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 DATA_LEN 12800
EXEC SQL BEGIN DECLARE SECTION;
// SQLPP initializes longdata.array_len
DECL_LONGVARCHAR(128000) longdata;
EXEC SQL END DECLARE SECTION;

void set_test_var()
{
  // init longdata for sending data
  memset( longdata.array, 'a', DATA_LEN );
  longdata.stored_len = DATA_LEN;

  printf( "Setting test_var to %d a's¥n", DATA_LEN );
  EXEC SQL SET test_var = :longdata;
}
```

参照

- 「Embedded SQL を使用して long 値を送信し、取得する方法」523 ページ
- 「インジケータ変数」495 ページ

動的 SQL を使用した LONG データの送信

ESQL 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。

前提条件

この作業を実行するための前提条件はありません。

♦ タスク

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。

参照

- 「Embedded SQL を使用して long 値を送信し、取得する方法」523 ページ
- 「インジケータ変数」495 ページ

Embedded SQL での簡単なストアドプロシージャ

Embedded SQL でストアドプロシージャを作成して呼び出すことができます。

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したい場合、または出力変数を取り出したい場合は、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=sql';

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;
```

参照

- 「CREATE PROCEDURE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「EXECUTE 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「PREPARE 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』

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 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 カラムはどちらのビットも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記述するには、文ではなくカーソルを記述する必要があります。

参照

- 「CREATE PROCEDURE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「動的 SELECT 文」505 ページ
- 「DESCRIBE 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』

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。

参照

- 「db_register_a_callback 関数」546 ページ
- 「db_cancel_request 関数」538 ページ
- 「db_is_working 関数」543 ページ

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 文の使用をおすすめします。

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。SQL Anywhere 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、SQL Anywhere Backup ユーティリティと直接やり取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。

参照

- 「BACKUP 文」『SQL Anywhere サーバ SQL リファレンス』
- 「バックアップユーティリティ (dbbackup)」『SQL Anywhere サーバ データベース管理』
- DBBackup メソッド [データベースツール]869 ページ
- 「db_backup 関数」533 ページ

ライブラリ関数のリファレンス

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。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。

この項では、これらの関数のリファレンスについて説明します。

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。これは、__stdcall の値に解析されます。

alloc_sqlda 関数

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。alloc_sqlda_noind 関数の代わりに、この関数を使用することをおすすめします。

参照

- 「SQLDA (SQL descriptor area)」506 ページ
- 「fill_s_sqlda 関数」556 ページ
- 「fill_sqlda 関数」556 ページ
- 「fill_sqlda_ex 関数」557 ページ
- 「free_sqlda 関数」558 ページ

alloc_sqlda_noind 関数

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。

参照

- 「SQLDA (SQL descriptor area)」506 ページ
- 「fill_s_sqlda 関数」556 ページ
- 「fill_sqlda 関数」556 ページ
- 「fill_sqlda_ex 関数」557 ページ
- 「free_sqlda_noind 関数」559 ページ

db_backup 関数

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データベースチェックポイントは無効にされます (db_backup は、DB_BACKUP_END の *op* 値で呼び出されます)。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*、*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 と *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。

注意
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。古いほう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、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 と *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*、*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データベースチェックポイントは無効にされます (db_backup は、DB_BACKUP_END の *op* 値で呼び出されます)。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、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。

  注意
  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_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_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_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* と *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_PARALLEL_START 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_NO_ERROR
    /* The file exists */
    num_pages = SQLCOUNT
    file_time = SQLE_IO_ESTIMATE
    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_NO_ERROR
     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
```

参照

- 「SQLCA (SQL Communication Area)」498 ページ
- 「SQLDA (SQL descriptor area)」506 ページ
- 「BACKUP 文」『SQL Anywhere サーバ SQL リファレンス』
- 「Embedded SQL のデータ型」485 ページ

db_cancel_request 関数

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。

参照

- 「SQLCA (SQL Communication Area)」498 ページ

db_change_char_charset 関数

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

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。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。

参照

- 「SQLCA (SQL Communication Area)」498 ページ
- 「推奨文字セットと照合」『SQL Anywhere サーバ データベース管理』

db_change_nchar_charset 関数

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

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。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。

参照

- 「SQLCA (SQL Communication Area)」498 ページ
- 「推奨文字セットと照合」『SQL Anywhere サーバ データベース管理』

db_find_engine 関数

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。

参照

- 「SQLCA (SQL Communication Area)」498 ページ

db_fini 関数

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。

UltraLite アプリケーションでは、同等の db_fini メソッドがあります。

参照

- 「SQLCA (SQL Communication Area)」498 ページ
- 「db_fini メソッド」『Ultra Light C/C++ プログラミング』

db_get_property 関数

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 ("16.0.1297" など)。

参照

- 「SQLCA (SQL Communication Area)」498 ページ

db_init 関数

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。

UltraLite アプリケーションでは、同等の db_init メソッドがあります。

参照

- 「SQLCA (SQL Communication Area)」498 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」500 ページ
- 「db_init メソッド」『Ultra Light C/C++ プログラミング』

db_is_working 関数

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 を返します。*sqlca* を使用する要求が処理中でない場合は 0 を返します。

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。

参照

- 「SQLCA (SQL Communication Area)」498 ページ

db_locate_servers 関数

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 SQL Anywhere 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 *sqlca.h* で次のように定義されています。

```
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* 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。

参照

- 「SQLCA (SQL Communication Area)」498 ページ
- 「サーバ列挙ユーティリティ (dblocate)」『SQL Anywhere サーバ データベース管理』

db_locate_servers_ex 関数

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 SQL Anywhere 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 *sqlca.h* で次のように定義されています。

```
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* 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。

参照

- 「SQLCA (SQL Communication Area)」498 ページ
- 「サーバ列挙ユーティリティ (dblocate)」『SQL Anywhere サーバ データベース管理』

db_register_a_callback 関数

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**　後述するように、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。

  このコールバックは次のように登録します。

  ```
  db_register_a_callback( &sqlca,
    DB_CALLBACK_WAIT,
    (SQL_CALLBACK_PARM)&db_wait_request );
  ```

- **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 フィールド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* フィールドはメッセージの長さを示します。メッセージは、NULL で終了しません。SQL Anywhere DBLIB および ODBC クライアントは、DB_CALLBACK_MESSAGE パラメータを使用して、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 [メッセージ] 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。

参照

- 「SQLCA (SQL Communication Area)」498 ページ
- 「LogFile (LOG) 接続パラメータ」『SQL Anywhere サーバ データベース管理』
- 「MESSAGE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「progress_messages オプション」『SQL Anywhere サーバ データベース管理』

db_start_database 関数

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 で終了された文字列で、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=sql;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。

参照

- 「SQLCA (SQL Communication Area)」498 ページ
- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』
- 「トラブルシューティング：データベースサーバの検索方法」『SQL Anywhere サーバ データベース管理』
- 「-gd データベースサーバオプション」『SQL Anywhere サーバ データベース管理』

db_start_engine 関数

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 で終了された文字列で、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=sql;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=dbsrv16" );
```

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。

```
dbsrv16 -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=dbsrv16 -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。

参照

- 「SQLCA (SQL Communication Area)」498 ページ
- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』
- 「トラブルシューティング：データベースサーバの検索方法」『SQL Anywhere サーバ データベース管理』

db_stop_database 関数

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 で終了された文字列で、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=sql;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。

参照

- 「SQLCA (SQL Communication Area)」498 ページ
- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』
- 「-gd データベースサーバオプション」『SQL Anywhere サーバ データベース管理』

db_stop_engine 関数

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 で終了された文字列で、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=sql;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。

 Copyright © 2013, SAP AG or an SAP affiliate company.

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。

参照

- 「SQLCA (SQL Communication Area)」498 ページ
- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』
- 「-gk データベースサーバオプション」『SQL Anywhere サーバ データベース管理』

db_string_connect 関数

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 で終了された文字列で、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=sql;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、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。

参照

- 「SQLCA (SQL Communication Area)」498 ページ
- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』
- 「トラブルシューティング：接続」『SQL Anywhere サーバ データベース管理』

db_string_disconnect 関数

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 で終了された文字列で、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=sql;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

この関数は、ConnectionName パラメータで識別される接続を解除します。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。

参照

- 「SQLCA (SQL Communication Area)」498 ページ
- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』

db_string_ping_server 関数

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、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。

参照

- 「SQLCA (SQL Communication Area)」498 ページ
- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』

db_time_change 関数

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ようにすることをおすすめします。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。

参照

- 「SQLCA (SQL Communication Area)」498 ページ

fill_s_sqlda 関数

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

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* バイトまでです。SQLDA 内の長さフィールド (sqllen) は適切に修正されます。

SQLDA は、free_filled_sqlda 関数を使用して解放する必要があります。

参照

- 「SQLCA (SQL Communication Area)」498 ページ

fill_sqlda 関数

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。

SQLDA は、free_filled_sqlda 関数を使用して解放する必要があります。

参照

- 「SQLCA (SQL Communication Area)」498 ページ
- 「fill_sqlda_ex 関数」557 ページ
- 「free_filled_sqlda 関数」558 ページ

fill_sqlda_ex 関数

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。

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*xyz* 型を使用すると、DT_VARCHAR、DT_NVARCHAR、DT_BINARY が制限されている 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。

参照

- 「SQLCA (SQL Communication Area)」498 ページ
- 「fill_sqlda 関数」556 ページ
- 「free_filled_sqlda 関数」558 ページ

free_filled_sqlda 関数

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。

参照

- 「SQLCA (SQL Communication Area)」498 ページ
- 「alloc_sqlda 関数」532 ページ
- 「fill_s_sqlda 関数」556 ページ
- 「fill_sqlda 関数」556 ページ
- 「fill_sqlda_ex 関数」557 ページ
- 「free_sqlda 関数」558 ページ

free_sqlda 関数

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。

参照

- 「SQLDA (SQL descriptor area)」506 ページ
- 「alloc_sqlda 関数」532 ページ
- 「fill_s_sqlda 関数」556 ページ
- 「fill_sqlda 関数」556 ページ
- 「fill_sqlda_ex 関数」557 ページ

free_sqlda_noind 関数

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。

参照

- 「SQLDA (SQL descriptor area)」506 ページ
- 「alloc_sqlda 関数」532 ページ
- 「fill_s_sqlda 関数」556 ページ
- 「fill_sqlda 関数」556 ページ
- 「fill_sqlda_ex 関数」557 ページ

sql_needs_quotes 関数

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。

参照

- 「SQLCA (SQL Communication Area)」498 ページ

sqlda_storage 関数

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**　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。

参照

- 「SQLDA (SQL descriptor area)」506 ページ

sqlda_string_length 関数

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。

参照

- 「SQLDA (SQL descriptor area)」506 ページ

sqlerror_message 関数

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。

参照

- 「SQLCA (SQL Communication Area)」498 ページ

Embedded SQL 文のまとめ

必ず、ESQL 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、ESQL 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。

標準 SQL 文の詳細については、「SQL 文」『SQL Anywhere サーバ SQL リファレンス』を参照してください。

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。「SQL 言語の要素」『SQL Anywhere サーバ SQL リファレンス』を参照してください。

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]」『SQL Anywhere サーバ SQL リファレンス』**　記述子にメモリを割り付ける。
- **「CLOSE 文 [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』**　カーソルを閉じる。

- **「CONNECT 文 [ESQL] [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』** データベースに接続する。
- **「DEALLOCATE DESCRIPTOR 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』** 記述子のメモリを再使用する。
- **「宣言セクション [ESQL]」『SQL Anywhere サーバ SQL リファレンス』** 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する。
- **「DECLARE CURSOR 文 [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』** カーソルを宣言する。
- **「DELETE 文 (位置付け) [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』** カーソルの現在位置のローを削除する。
- **「DESCRIBE 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』** 特定の SQL 文用のホスト変数を記述する。
- **「DISCONNECT 文 [ESQL] [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』** データベースサーバとの接続を切断する。
- **「DROP STATEMENT 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』** 準備文が使用したリソースを解放する。
- **「EXECUTE 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』** 特定の SQL 文を実行する。
- **「EXPLAIN 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』** 特定のカーソルの最適化方式を説明する。
- **「FETCH 文 [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』** カーソルからローをフェッチする。
- **「GET DATA 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』** カーソルから長い値をフェッチする。
- **「GET DESCRIPTOR 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』** SQLDA 内の変数に関する情報を取り出す。
- **「GET OPTION 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』** 特定のデータベースオプションの設定を取得する。
- **「INCLUDE 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』** SQL 前処理用のファイルをインクルードする。
- **「OPEN 文 [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』** カーソルを開く。
- **「PREPARE 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』** 特定の SQL 文を準備する。

- **「PUT 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』** カーソルにローを挿入する。
- **「SET CONNECTION 文 [Interactive SQL] [ESQL]」『SQL Anywhere サーバ SQL リファレンス』** アクティブな接続を変更する。
- **「SET DESCRIPTOR 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』** SQLDA 内で変数を記述し、SQLDA にデータを置く。
- **「SET SQLCA 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』** デフォルトのグローバル SQLCA 以外の SQLCA を使用する。
- **「UPDATE (位置付け) 文 [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』** カーソルの現在位置のローを更新する。
- **「WHENEVER 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』** SQL 文でエラーが発生した場合の動作を指定する。

C/C++ 用の SQL Anywhere データベース API

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 を使用すると、C/C++ アプリケーションから 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。

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。SQL Anywhere C API を使用すると、C/C++ アプリケーションから 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。

SQL Anywhere 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。SQL Anywhere 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。SQL Anywhere 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。

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 SQL Anywhere 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。

SQL Anywhere 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 SQL Anywhere の *sdk¥dbcapi* ディレクトリにあります。

スレッドサポート

SQL Anywhere 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。

例

SQL Anywhere C API の使用例については、インストールされている SQL Anywhere の *sdk¥dbcapi¥examples* ディレクトリにある例を参照してください。

- ***connecting.cpp*** これは、接続オブジェクトを作成し、SQL Anywhere 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。

インタフェースライブラリの動的ロード

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 SQL Anywhere の *sdk¥dbcapi* 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 SQL Anywhere に付属の例を参照してください。

SQL Anywhere C API リファレンス

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。

sqlany_affected_rows メソッド

準備文の実行の影響を受けるローの数を返します。

構文

```
public sacapi_i32 sqlany_affected_rows(a_sqlany_stmt * sqlany_stmt)
```

パラメータ

- **sqlany_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。失敗した場合は -1。

参照

- sqlany_execute メソッド [SQL Anywhere C]572 ページ
- sqlany_execute_direct メソッド [SQL Anywhere C]573 ページ

sqlany_bind_param メソッド

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

```
public sacapi_bool sqlany_bind_param(
    a_sqlany_stmt * sqlany_stmt,
    sacapi_u32 index,
    a_sqlany_bind_param * param
)
```

パラメータ

- **sqlany_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。この数値は、0 ～ sqlany_num_params() - 1 の間である必要があります。
- **param**　バインドされるパラメータの a_sqlany_bind_param 構造体の記述。

戻り値

成功した場合は 1、失敗した場合は 0。

参照

- sqlany_describe_bind_param メソッド [SQL Anywhere C]570 ページ

sqlany_cancel メソッド

接続の未処理の要求をキャンセルします。

構文

```
public void sqlany_cancel(a_sqlany_connection * sqlany_conn)
```

パラメータ

- **sqlany_conn** sqlany_connect() を使用して確立された接続の接続オブジェクト。

sqlany_clear_error メソッド

最後に格納されたエラーコードをクリアします。

構文

```
public void sqlany_clear_error(a_sqlany_connection * sqlany_conn)
```

パラメータ

- **sqlany_conn** sqlany_new_connection() から返された接続オブジェクト。

参照

- sqlany_new_connection メソッド [SQL Anywhere C]584 ページ

sqlany_client_version メソッド

現在のクライアントバージョンを返します。

構文

```
public sacapi_bool sqlany_client_version(char * buffer, size_t len)
```

パラメータ

- **buffer** クライアントバージョン文字列を格納するバッファ。
- **len** 指定されたバッファの長さ。

戻り値

成功した場合は 1、失敗した場合は 0。

備考

このメソッドは、渡されたバッファに、クライアントライブラリのメジャー番号、マイナー番号、パッチ番号、ビルド番号を入れます。バッファは NULL で終了します。

sqlany_client_version_ex メソッド

現在のクライアントバージョンを返します。

構文

```
public sacapi_bool sqlany_client_version_ex(
    a_sqlany_interface_context * context,
    char * buffer,
```

```
    size_t len
)
```

パラメータ

- **context**　sqlany_init_ex() を使用して作成されたオブジェクト。
- **buffer**　クライアントバージョン文字列を格納するバッファ。
- **len**　指定されたバッファの長さ。

戻り値

成功した場合は 1、失敗した場合は 0。

備考

このメソッドは、渡されたバッファに、クライアントライブラリのメジャー番号、マイナー番号、パッチ番号、ビルド番号を入れます。バッファは NULL で終了します。

参照

- sqlany_init_ex メソッド [SQL Anywhere C]582 ページ

sqlany_commit メソッド

現在のトランザクションをコミットします。

構文

```
public sacapi_bool sqlany_commit(a_sqlany_connection * sqlany_conn)
```

パラメータ

- **sqlany_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 1、失敗した場合は 0。

参照

- sqlany_rollback メソッド [SQL Anywhere C]588 ページ

sqlany_connect メソッド

指定された接続オブジェクトと接続文字列を使用して、SQL Anywhere データベースサーバへの接続を作成します。

構文

```
public sacapi_bool sqlany_connect(
    a_sqlany_connection * sqlany_conn,
```

```
    const char * str
)
```

パラメータ

- **sqlany_conn** sqlany_new_connection() によって作成された接続オブジェクト。
- **str** SQL Anywhere 接続文字列。

戻り値

接続が確立された場合は 1、接続が失敗した場合は 0。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。

備考

まず、sqlany_new_connection() を使用して、指定された接続オブジェクトを割り付ける必要があります。

次の例は、失敗した接続試行のエラーコードを取得する方法を示します。

```
a_sqlany_connection * sqlany_conn;
sqlany_conn = sqlany_new_connection();
if( !sqlany_connect( sqlany_conn, "uid=dba;pwd=sql" ) ) {
   char reason[SACAPI_ERROR_SIZE];
   sacapi_i32 code;
   code = sqlany_error( sqlany_conn, reason, sizeof(reason) );
   printf( "Connection failed. Code: %d Reason: %s¥n", code, reason );
} else {
   printf( "Connected successfully!¥n" );
   sqlany_disconnect( sqlany_conn );
}
sqlany_free_connection( sqlany_conn );
```

SQL Anywhere データベースサーバに接続する方法の詳細については、「接続パラメータ」『SQL Anywhere サーバ データベース管理』と「データベース接続」『SQL Anywhere サーバ データベース管理』を参照してください。

参照

- sqlany_new_connection メソッド [SQL Anywhere C]584 ページ
- sqlany_error メソッド [SQL Anywhere C]571 ページ

sqlany_describe_bind_param メソッド

準備文のバインドパラメータを記述します。

構文

```
public sacapi_bool sqlany_describe_bind_param(
    a_sqlany_stmt * sqlany_stmt,
    sacapi_u32 index,
    a_sqlany_bind_param * param
)
```

パラメータ

- **sqlany_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。この数値は、0 ～ sqlany_num_params() - 1 の間である必要があります。
- **param** 情報が格納された a_sqlany_bind_param 構造体。

戻り値

成功した場合は 1、失敗した場合は 0。

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。

参照

- sqlany_bind_param メソッド [SQL Anywhere C]567 ページ
- sqlany_prepare メソッド [SQL Anywhere C]587 ページ

sqlany_disconnect メソッド

すでに確立された SQL Anywhere 接続を切断します。

構文

```
public sacapi_bool sqlany_disconnect(a_sqlany_connection * sqlany_conn)
```

パラメータ

- **sqlany_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 1、失敗した場合は 0。

備考

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。

参照

- sqlany_connect メソッド [SQL Anywhere C]569 ページ
- sqlany_new_connection メソッド [SQL Anywhere C]584 ページ

sqlany_error メソッド

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取得します。

構文

```
public sacapi_i32 sqlany_error(
    a_sqlany_connection * sqlany_conn,
    char * buffer,
    size_t size
)
```

パラメータ

- **sqlany_conn** sqlany_new_connection() から返された接続オブジェクト。
- **buffer** エラーメッセージを格納するバッファ。
- **size** 指定されたバッファのサイズ。

戻り値

最後のエラーコード。正の値は警告、負の値はエラー、0 は成功を示します。

備考

SQLCODE エラーメッセージの詳細については、「SQL Anywhere のエラーメッセージ (SQLCODE 順)」『エラーメッセージ』を参照してください。

参照

- sqlany_connect メソッド [SQL Anywhere C]569 ページ

sqlany_execute メソッド

準備文を実行します。

構文

```
public sacapi_bool sqlany_execute(a_sqlany_stmt * sqlany_stmt)
```

パラメータ

- **sqlany_stmt** sqlany_prepare() を使用して準備された文。

戻り値

文が正しく実行された場合は 1、失敗した場合は 0。

備考

sqlany_num_cols() を使用して、実行された文が結果セットを返したかどうか確認できます。

次の例は、結果セットを返さない文を実行する方法を示します。

```
a_sqlany_stmt *      stmt;
int                  i;
a_sqlany_bind_param  param;

stmt = sqlany_prepare( sqlany_conn, "insert into moe(id,value) values( ?,? )" );
if( stmt ) {
```

```
    sqlany_describe_bind_param( stmt, 0, &param );
    param.value.buffer = (char *)&i;
    param.value.type   = A_VAL32;
    sqlany_bind_param( stmt, 0, &param );

    sqlany_describe_bind_param( stmt, 1, &param );
    param.value.buffer = (char *)&i;
    param.value.type   = A_VAL32;
    sqlany_bind_param( stmt, 1, &param );

    for( i = 0; i < 10; i++ ) {
      if( !sqlany_execute( stmt ) ) {
          // call sqlany_error()
        }
    }
    sqlany_free_stmt( stmt );
  }
```

参照

- sqlany_prepare メソッド [SQL Anywhere C]587 ページ

sqlany_execute_direct メソッド

文字列引数で指定された SQL 文を実行し、結果セットを返す可能性があります。

構文

```
public a_sqlany_stmt * sqlany_execute_direct(
    a_sqlany_connection * sqlany_conn,
    const char * sql_str
)
```

パラメータ

- **sqlany_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_str**　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

関数の実行が成功した場合はステートメントハンドル、失敗した場合は NULL。

備考

このメソッドを使用して、文を準備および実行できます。また、sqlany_prepare() に続けて sqlany_execute() を呼び出す代わりにも使用できます。

次の例は、結果セットを返す文を実行する方法を示します。

```
a_sqlany_stmt *   stmt;

stmt = sqlany_execute_direct( sqlany_conn, "select * from employees" );
if( stmt && sqlany_num_cols( stmt ) > 0 ) {
  while( sqlany_fetch_next( stmt ) ) {
    int i;
      for( i = 0; i < sqlany_num_cols( stmt ); i++ ) {
       // Get column i data
    }
```

```
    }
    sqlany_free_stmt( stmt  );
}
```

注意
この関数は、パラメータを持つ SQL 文の実行には使用できません。

参照

- sqlany_fetch_absolute メソッド [SQL Anywhere C]574 ページ
- sqlany_fetch_next メソッド [SQL Anywhere C]575 ページ
- sqlany_num_cols メソッド [SQL Anywhere C]585 ページ
- sqlany_get_column メソッド [SQL Anywhere C]578 ページ

sqlany_execute_immediate メソッド

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。

構文

```
public sacapi_bool sqlany_execute_immediate(
    a_sqlany_connection * sqlany_conn,
    const char * sql
)
```

パラメータ

- **sqlany_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**　実行される SQL 文を表す文字列。

戻り値

成功した場合は 1、失敗した場合は 0。

備考

この関数は、結果セットを返さない SQL 文に役立ちます。

sqlany_fetch_absolute メソッド

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

```
public sacapi_bool sqlany_fetch_absolute(
    a_sqlany_stmt * sqlany_stmt,
    sacapi_i32 row_num
)
```

パラメータ

- **sqlany_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

フェッチに成功した場合は 1、失敗した場合は 0。

参照

- sqlany_execute_direct メソッド [SQL Anywhere C]573 ページ
- sqlany_execute メソッド [SQL Anywhere C]572 ページ
- sqlany_error メソッド [SQL Anywhere C]571 ページ
- sqlany_fetch_next メソッド [SQL Anywhere C]575 ページ

sqlany_fetch_next メソッド

結果セットから次のローを返します。

構文

```
public sacapi_bool sqlany_fetch_next(a_sqlany_stmt * sqlany_stmt)
```

パラメータ

- **sqlany_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

フェッチに成功した場合は 1、失敗した場合は 0。

備考

この関数は、結果セットから次のローを返します。結果オブジェクトを初めて作成した場合、現在のローポインタは最初のローであるロー 0 の前に設定されます。この関数は、ローポインタを進めてから新しいローでデータをフェッチします。

参照

- sqlany_fetch_absolute メソッド [SQL Anywhere C]574 ページ
- sqlany_execute_direct メソッド [SQL Anywhere C]573 ページ
- sqlany_execute メソッド [SQL Anywhere C]572 ページ
- sqlany_error メソッド [SQL Anywhere C]571 ページ

sqlany_finalize_interface メソッド

C API DLL ライブラリをアンロードし、SQLAnywhereInterface 構造体をリセットします。

構文

```
public void sqlany_finalize_interface(SQLAnywhereInterface * api)
```

パラメータ

- **api** ファイナライズする初期化された構造体。

備考

次の文を使用して関数プロトタイプをインクルードします。

```
#include "sacapidll.h"
```

このメソッドを使用して、SQL Anywhere C API DLL に関連付けられているリソースのファイナライズおよび解放を実行します。

sqlany_finalize_interface メソッドの使用例については、インストールされている SQL Anywhere の *sdk¥dbcapi¥examples* ディレクトリにある C API の例を参照してください。

sqlany_fini メソッド

インタフェースをファイナライズします。

構文

```
public void sqlany_fini()
```

備考

API によって割り付けられたリソースをすべて解放します。

参照

- sqlany_init メソッド [SQL Anywhere C]581 ページ

sqlany_fini_ex メソッド

指定したコンテキストを使用して作成されたインタフェースをファイナライズします。

構文

```
public void sqlany_fini_ex(a_sqlany_interface_context * context)
```

パラメータ

- **context** sqlany_init_ex() から返されたコンテキストオブジェクト。

参照

- sqlany_init_ex メソッド [SQL Anywhere C]582 ページ

sqlany_free_connection メソッド

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

```
public void sqlany_free_connection(a_sqlany_connection * sqlany_conn)
```

パラメータ

- **sqlany_conn** sqlany_new_connection によって作成された接続オブジェクト。

参照

- sqlany_new_connection メソッド [SQL Anywhere C]584 ページ

sqlany_free_stmt メソッド

準備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

```
public void sqlany_free_stmt(a_sqlany_stmt * sqlany_stmt)
```

パラメータ

- **sqlany_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。

参照

- sqlany_prepare メソッド [SQL Anywhere C]587 ページ
- sqlany_execute_direct メソッド [SQL Anywhere C]573 ページ

sqlany_get_bind_param_info メソッド

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

```
public sacapi_bool sqlany_get_bind_param_info(
    a_sqlany_stmt * sqlany_stmt,
    sacapi_u32 index,
    a_sqlany_bind_param_info * info
)
```

パラメータ

- **sqlany_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。この数値は、0 ～ sqlany_num_params() - 1 の間である必要があります。

- **info**　バインドパラメータの情報を格納する sqlany_bind_param_info バッファ。

戻り値

成功した場合は 1、失敗した場合は 0。

参照

- sqlany_bind_param メソッド [SQL Anywhere C]567 ページ
- sqlany_describe_bind_param メソッド [SQL Anywhere C]570 ページ
- sqlany_prepare メソッド [SQL Anywhere C]587 ページ

sqlany_get_column メソッド

指定されたカラムのためにフェッチされた値を、指定されたバッファに格納します。

構文

```
public sacapi_bool sqlany_get_column(
    a_sqlany_stmt * sqlany_stmt,
    sacapi_u32 col_index,
    a_sqlany_data_value * buffer
)
```

パラメータ

- **sqlany_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。
- **buffer**　カラム col_index のためにフェッチされたデータを格納する a_sqlany_data_value オブジェクト。

戻り値

成功した場合は 1、失敗した場合は 0。パラメータのいずれかが無効であった場合、または SQL Anywhere データベースサーバから完全な値を取り出すために必要なメモリがない場合は、失敗する可能性があります。

備考

A_BINARY および A_STRING * データ型では、value->buffer は、結果セットに関連付けられている内部バッファをポイントします。ポインタバッファの内容は、新しいローがフェッチされたとき、または結果セットオブジェクトが解放されたときに変更されます。したがって、ポインタの内容を変更したり、内容に依存したりしないでください。ユーザはこれらのポインタから各自のバッファにデータをコピーする必要があります。

value->length フィールドは、value->buffer がポイントする有効な文字の数を示します。value->buffer で返されるデータは、NULL で終了しません。この関数は、SQL Anywhere データベースサーバからのすべての戻り値をフェッチします。たとえば、カラムに BLOB が含まれている場合、その値を保持するのに必要なメモリの割り付けはこの関数が行います。メモリの割り付けを行わない場合は、代わりに sqlany_get_data() を使用してください。

参照

- sqlany_execute_direct メソッド [SQL Anywhere C]573 ページ
- sqlany_execute メソッド [SQL Anywhere C]572 ページ
- sqlany_fetch_absolute メソッド [SQL Anywhere C]574 ページ
- sqlany_fetch_next メソッド [SQL Anywhere C]575 ページ

sqlany_get_column_info メソッド

カラムのメタデータ情報を取得し、a_sqlany_column_info 構造体にカラムに関する情報を格納します。

構文

```
public sacapi_bool sqlany_get_column_info(
    a_sqlany_stmt * sqlany_stmt,
    sacapi_u32 col_index,
    a_sqlany_column_info * buffer
)
```

パラメータ

- **sqlany_stmt**　sqlany_prepare() または sqlany_execute_direct() によって作成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。
- **buffer**　カラム情報を格納するカラム info 構造体。

戻り値

成功した場合は 1。カラムのインデックスが範囲外の場合、または文が結果セットを返さない場合は 0。

参照

- sqlany_execute メソッド [SQL Anywhere C]572 ページ
- sqlany_execute_direct メソッド [SQL Anywhere C]573 ページ
- sqlany_prepare メソッド [SQL Anywhere C]587 ページ

sqlany_get_data メソッド

指定されたカラムのためにフェッチしたデータを取り出し、指定されたバッファのメモリに格納します。

構文

```
public sacapi_i32 sqlany_get_data(
    a_sqlany_stmt * sqlany_stmt,
    sacapi_u32 col_index,
    size_t offset,
    void * buffer,
```

```
    size_t size
)
```

パラメータ

- **sqlany_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。
- **offset** 取得するデータの開始オフセット。
- **buffer** カラムの内容を格納するバッファ。コピーされるデータ型に応じてバッファポインタのアラインメントを適切に行う必要があります。
- **size** バッファのサイズ (バイト単位)。指定したサイズが 2^31 - 1 より大きいと、関数は失敗します。

戻り値

指定されたバッファにコピーされたバイト数。この値は 2^31 - 1 を超えることはできません。0 は、コピーすべきデータが残っていないことを意味します。-1 は失敗を示します。

参照

- sqlany_execute メソッド [SQL Anywhere C]572 ページ
- sqlany_execute_direct メソッド [SQL Anywhere C]573 ページ
- sqlany_fetch_absolute メソッド [SQL Anywhere C]574 ページ
- sqlany_fetch_next メソッド [SQL Anywhere C]575 ページ

sqlany_get_data_info メソッド

最後のフェッチ操作によってフェッチされたデータに関する情報を取り出します。

構文

```
public sacapi_bool sqlany_get_data_info(
    a_sqlany_stmt * sqlany_stmt,
    sacapi_u32 col_index,
    a_sqlany_data_info * buffer
)
```

パラメータ

- **sqlany_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。
- **buffer** フェッチされたデータのメタデータを格納するデータ info バッファ。

戻り値

成功した場合は 1、失敗した場合は 0。指定したパラメータのいずれかが無効であった場合、失敗が返されます。

参照

- sqlany_execute メソッド [SQL Anywhere C]572 ページ
- sqlany_execute_direct メソッド [SQL Anywhere C]573 ページ
- sqlany_fetch_absolute メソッド [SQL Anywhere C]574 ページ
- sqlany_fetch_next メソッド [SQL Anywhere C]575 ページ

sqlany_get_next_result メソッド

複数の結果セットクエリのうちの次の結果セットに進みます。

構文

```
public sacapi_bool sqlany_get_next_result(a_sqlany_stmt * sqlany_stmt)
```

パラメータ

- **sqlany_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

文が次の結果セットに進んだ場合は 1、それ以外の場合は 0。

備考

クエリ (ストアドプロシージャの呼び出しなど) が複数の結果セットを返す場合、この関数は現在の結果セットから次へ進みます。

次の例は、複数の結果セットクエリのうちの次の結果セットに進む方法を示します。

```
stmt = sqlany_execute_direct( sqlany_conn, "call my_multiple_results_procedure()" );
if( result ) {
   do {
      while( sqlany_fetch_next( stmt ) ) {
         // get column data
      }
   } while( sqlany_get_next_result( stmt ) );
   sqlany_free_stmt( stmt );
}
```

参照

- sqlany_execute_direct メソッド [SQL Anywhere C]573 ページ
- sqlany_execute メソッド [SQL Anywhere C]572 ページ

sqlany_init メソッド

インタフェースを初期化します。

構文

```
public sacapi_bool sqlany_init(
    const char * app_name,
    sacapi_u32 api_version,
    sacapi_u32 * version_available
)
```

パラメータ

- **app_name**　API を使用しているアプリケーションに名前を付ける文字列。たとえば、"PHP"、"PERL"、"RUBY" など。
- **api_version**　コンパイルされたアプリケーションのバージョン。
- **version_available**　サポートされている API の最大バージョンを返すオプションの引数。

戻り値

成功した場合は 1、失敗した場合は 0。

備考

次の例は、SQL Anywhere C API DLL を初期化する方法を示します。

```
sacapi_u32 api_version;
if( sqlany_init( "PHP", SQLANY_API_VERSION_1, &api_version ) ) {
   printf( "Interface initialized successfully!¥n" );
} else {
   printf( "Failed to initialize the interface! Supported version=%d¥n", api_version );
}
```

参照

- sqlany_fini メソッド [SQL Anywhere C]576 ページ

sqlany_init_ex メソッド

コンテキストを使用してインタフェースを初期化します。

構文

```
public a_sqlany_interface_context * sqlany_init_ex(
    const char * app_name,
    sacapi_u32 api_version,
    sacapi_u32 * version_available
)
```

パラメータ

- **app_name**　使用される API に "PHP"、"PERL"、"RUBY" などの名前を付ける文字列。
- **api_version**　アプリケーションで使用されている現在の API バージョン。通常、SQLANY_API_VERSION_* マクロの 1 つである必要があります。
- **version_available**　サポートされている API の最大バージョンを返すオプションの引数。

戻り値

成功した場合はコンテキストオブジェクト、失敗した場合は NULL。

参照

- sqlany_fini_ex メソッド [SQL Anywhere C]576 ページ

sqlany_initialize_interface メソッド

SQLAnywhereInterface オブジェクトを初期化し、DLL を動的にロードします。

構文

```
public int sqlany_initialize_interface(
    SQLAnywhereInterface * api,
    const char * optional_path_to_dll
)
```

パラメータ

- **api**　初期化する API structure 構造体。
- **optional_path_to_dll**　SQL Anywhere C API DLL へのパスを指定するオプションの引数。

戻り値

初期化に成功した場合は 1、失敗した場合は 0。

備考

次の文を使用して関数プロトタイプをインクルードします。

```
#include "sacapidll.h"
```

この関数は SQL Anywhere C API DLL を動的にロードし、DLL のすべてのエントリポイントを検索します。SQLAnywhereInterface 構造体のフィールドに入力されたデータは、DLL の該当する関数をポイントします。オプションのパス引数が NULL の場合は、SQLANY_DLL_PATH 環境変数がチェックされます。SQLANY_DLL_PATH 環境変数が設定されている場合、ライブラリは環境変数が指定する DLL をロードします。失敗した場合、インタフェースが DLL を直接ロードします (環境が正しく設定されているかによって決まります)。

sqlany_initialize_interface メソッドの使用例については、インストールされている SQL Anywhere の *sdk¥dbcapi¥examples* ディレクトリにある C API の例を参照してください。

sqlany_make_connection メソッド

指定された DBLIB SQLCA ポインタに基づいて接続オブジェクトを作成します。

構文

```
public a_sqlany_connection * sqlany_make_connection(void * arg)
```

パラメータ

- **arg** DBLIB SQLCA オブジェクトへの void * ポインタ。

戻り値

接続オブジェクト。

参照

- sqlany_new_connection メソッド [SQL Anywhere C]584 ページ
- sqlany_execute メソッド [SQL Anywhere C]572 ページ
- sqlany_execute_direct メソッド [SQL Anywhere C]573 ページ
- sqlany_execute_immediate メソッド [SQL Anywhere C]574 ページ
- sqlany_prepare メソッド [SQL Anywhere C]587 ページ

sqlany_make_connection_ex メソッド

指定された DBLIB SQLCA ポインタとコンテキストに基づいて接続オブジェクトを作成します。

構文

```
public a_sqlany_connection * sqlany_make_connection_ex(
    a_sqlany_interface_context * context,
    void * arg
)
```

パラメータ

- **context** sqlany_init_ex() によって作成された有効なコンテキストオブジェクト。
- **arg** DBLIB SQLCA オブジェクトへの void * ポインタ。

戻り値

接続オブジェクト。

参照

- sqlany_init_ex メソッド [SQL Anywhere C]582 ページ
- sqlany_execute メソッド [SQL Anywhere C]572 ページ
- sqlany_execute_direct メソッド [SQL Anywhere C]573 ページ
- sqlany_execute_immediate メソッド [SQL Anywhere C]574 ページ
- sqlany_prepare メソッド [SQL Anywhere C]587 ページ

sqlany_new_connection メソッド

接続オブジェクトを作成します。

構文

```
public a_sqlany_connection * sqlany_new_connection(void)
```

戻り値

接続オブジェクト

備考

API 接続オブジェクトを作成してからデータベース接続を確立する必要があります。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。また、接続オブジェクトに一度にアクセスできるスレッドは 1 つに限られます。複数のスレッドが同時に接続オブジェクトへのアクセスを試みると、不確定な動作や障害が発生します。

参照

- sqlany_connect メソッド [SQL Anywhere C]569 ページ
- sqlany_disconnect メソッド [SQL Anywhere C]571 ページ

sqlany_new_connection_ex メソッド

コンテキストを使用して接続オブジェクトを作成します。

構文

```
public a_sqlany_connection * sqlany_new_connection_ex(
    a_sqlany_interface_context * context
)
```

パラメータ

- **context** sqlany_init_ex() から返されたコンテキストオブジェクト。

戻り値

接続オブジェクト

参照

- sqlany_connect メソッド [SQL Anywhere C]569 ページ
- sqlany_disconnect メソッド [SQL Anywhere C]571 ページ
- sqlany_init_ex メソッド [SQL Anywhere C]582 ページ

sqlany_num_cols メソッド

結果セット内のカラム数を返します。

構文

```
public sacapi_i32 sqlany_num_cols(a_sqlany_stmt * sqlany_stmt)
```

パラメータ

- **sqlany_stmt** sqlany_prepare() または sqlany_execute_direct() によって作成された文オブジェクト。

戻り値

結果セット内のカラム数。失敗した場合は -1。

参照

- sqlany_execute メソッド [SQL Anywhere C]572 ページ
- sqlany_execute_direct メソッド [SQL Anywhere C]573 ページ
- sqlany_prepare メソッド [SQL Anywhere C]587 ページ

sqlany_num_params メソッド

準備文で必要とされるパラメータ数を返します。

構文

```
public sacapi_i32 sqlany_num_params(a_sqlany_stmt * sqlany_stmt)
```

パラメータ

- **sqlany_stmt** sqlany_prepare() の実行によって返された文オブジェクト。

戻り値

必要とされるパラメータ数。文オブジェクトが有効でない場合は -1。

参照

- sqlany_prepare メソッド [SQL Anywhere C]587 ページ

sqlany_num_rows メソッド

結果セット内のロー数を返します。

構文

```
public sacapi_i32 sqlany_num_rows(a_sqlany_stmt * sqlany_stmt)
```

パラメータ

- **sqlany_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。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。row_counts オプションの詳細については、「row_counts オプション」『SQL Anywhere サーバ データベース管理』を参照してください。

参照

- sqlany_execute_direct メソッド [SQL Anywhere C]573 ページ
- sqlany_execute メソッド [SQL Anywhere C]572 ページ

sqlany_prepare メソッド

指定された SQL 文字列を準備します。

構文

```
public a_sqlany_stmt * sqlany_prepare(
    a_sqlany_connection * sqlany_conn,
    const char * sql_str
)
```

パラメータ

- **sqlany_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_str** 準備される SQL 文。

戻り値

SQL Anywhere 文オブジェクトのハンドル。sqlany_execute() で文オブジェクトを使用して、文を実行できます。

備考

sqlany_execute() が呼び出されるまで実行は行われません。返された文オブジェクトは、sqlany_free_stmt() を使用して解放する必要があります。

次の例は、SELECT SQL 文字列を準備する方法を示します。

```
char * str;
a_sqlany_stmt * stmt;

str = "select * from employees where salary >= ?";
stmt = sqlany_prepare( sqlany_conn, str );
if( stmt == NULL ) {
   // Failed to prepare statement, call sqlany_error() for more info
}
```

参照

- sqlany_free_stmt メソッド [SQL Anywhere C]577 ページ
- sqlany_connect メソッド [SQL Anywhere C]569 ページ
- sqlany_execute メソッド [SQL Anywhere C]572 ページ
- sqlany_num_params メソッド [SQL Anywhere C]586 ページ
- sqlany_describe_bind_param メソッド [SQL Anywhere C]570 ページ
- sqlany_bind_param メソッド [SQL Anywhere C]567 ページ

sqlany_reset メソッド

文を準備されたステータス状態にリセットします。

構文

```
public sacapi_bool sqlany_reset(a_sqlany_stmt * sqlany_stmt)
```

パラメータ

- **sqlany_stmt** sqlany_prepare() を使用して準備された文。

戻り値

成功した場合は 1、失敗した場合は 0。

参照

- sqlany_prepare メソッド [SQL Anywhere C]587 ページ

sqlany_rollback メソッド

現在のトランザクションをロールバックします。

構文

```
public sacapi_bool sqlany_rollback(a_sqlany_connection * sqlany_conn)
```

パラメータ

- **sqlany_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 1、失敗した場合は 0。

参照

- sqlany_commit メソッド [SQL Anywhere C]569 ページ

sqlany_send_param_data メソッド

データをバインドパラメータの一部として送信します。

構文

```
public sacapi_bool sqlany_send_param_data(
    a_sqlany_stmt * sqlany_stmt,
    sacapi_u32 index,
    char * buffer,
    size_t size
)
```

パラメータ

- **sqlany_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。これは 0 ～ sqlany_num_params() - 1 の間の数です。
- **buffer** 送信されるデータ。
- **size** 送信するバイト数。

戻り値

成功した場合は 1、失敗した場合は 0。

備考

このメソッドを使用して、バインドパラメータの大量のデータをチャンク単位で送信できます。

参照

- sqlany_prepare メソッド [SQL Anywhere C]587 ページ

sqlany_sqlstate メソッド

現在の SQLSTATE を取得します。

構文

```
public size_t sqlany_sqlstate(
    a_sqlany_connection * sqlany_conn,
    char * buffer,
    size_t size
)
```

パラメータ

- **sqlany_conn** sqlany_new_connection() から返された接続オブジェクト。
- **buffer** 現在の 5 文字 SQLSTATE を格納するバッファ。
- **size** バッファのサイズ。

戻り値

バッファにコピーされたバイト数。

備考

SQLSTATE エラーメッセージの詳細については、「SQL Anywhere のエラーメッセージ (SQLSTATE 順)」『エラーメッセージ』を参照してください。

参照

- sqlany_error メソッド [SQL Anywhere C]571 ページ

a_sqlany_data_direction 列挙

データ方向の列挙です。

構文

```
public enum a_sqlany_data_direction
```

メンバー

メンバー名	説明	値
DD_INVALID	無効なデータ方向。	0x0
DD_INPUT	入力専用のホスト変数。	0x1
DD_OUTPUT	出力専用のホスト変数。	0x2
DD_INPUT_OUTPUT	入出力するホスト変数。	0x3

a_sqlany_data_type 列挙

渡されている、または取り出されているデータ型を指定します。

構文

```
public enum a_sqlany_data_type
```

メンバー

メンバー名	説明
A_INVALID_TYPE	無効なデータ型。
A_BINARY	バイナリデータ。バイナリデータは現状のままで処理され、文字セット変換は実行されません。
A_STRING	文字列データ。文字セット変換が実行されるデータ。
A_DOUBLE	Double データ。float 値を含みます。
A_VAL64	64 ビット整数値。
A_UVAL64	64 ビット符号なし整数
A_VAL32	32 ビット整数値。
A_UVAL32	32 ビット符号なし整数

メンバー名	説明
A_VAL16	16 ビット整数値。
A_UVAL16	16 ビット符号なし整数
A_VAL8	8 ビット整数値。
A_UVAL8	8 ビット符号なし整数。

a_sqlany_native_type 列挙

サーバによって記述された値のネイティブ型の列挙。

構文

```
public enum a_sqlany_native_type
```

メンバー

メンバー名	説明
DT_NOTYPE	データ型なし
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_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。最大長は 32765 バイトです。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。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。データは NULL で終了せず、ブランクも埋め込まれません。
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_LONGVARCHAR	CHAR 文字セット内の長い可変長文字列

メンバー名	説明
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_DOUBLE	8 バイト浮動小数点数。
DT_FLOAT	4 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)
DT_INT	32 ビット符号付き整数。
DT_SMALLINT	16 ビット符号付き整数。
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。最大長は 32765 バイトです。データベースサーバに情報を渡す場合は、長さフィールドを設定してください。データベースサーバから情報をフェッチする場合は、サーバが長さフィールドを設定します。
DT_LONGBINARY	長いバイナリデータ。
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_UNSINT	32 ビット符号なし整数。
DT_UNSSMALLINT	16 ビット符号なし整数。
DT_UNSBIGINT	64 ビット符号なし整数。
DT_BIT	8 ビット符号付き整数。
DT_LONGNVARCHAR	NCHAR 文字セット内の長い可変長文字列。

備考

Embedded SQL データ型に対応する値タイプ。Embedded SQL のデータ型の詳細については、「Embedded SQL のデータ型」485 ページを参照してください。

参照

- sqlany_get_column_info メソッド [SQL Anywhere C]579 ページ
- a_sqlany_column_info 構造体 [SQL Anywhere C]594 ページ

a_sqlany_bind_param 構造体

パラメータと準備文をバインドするために使用されるバインドパラメータ構造体。

構文

```
public typedef struct a_sqlany_bind_param
```

メンバー

メンバー名	型	説明
direction	a_sqlany_data_direction	データの方向 (入力、出力、入出力)。
name	char *	バインドパラメータの名前。sqlany_describe_bind_param() によってのみ使用されます。
value	a_sqlany_data_value	データの実際の値。

備考

a_sqlany_bind_param 構造体の使用例については、インストールされている SQL Anywhere の *sdk¥dbcapi¥examples* ディレクトリにあるサンプルファイルを参照してください。

参照

- sqlany_execute メソッド [SQL Anywhere C]572 ページ

a_sqlany_bind_param_info 構造体

現在のバインドパラメータに関する情報を取得します。

構文

```
public typedef struct a_sqlany_bind_param_info
```

メンバー

メンバー名	タイプ	説明
direction	a_sqlany_data_direction	パラメータの方向。
input_value	a_sqlany_data_value	バインドされる入力値に関する情報。
name	char *	パラメータの名前へのポインタ。

メンバー名	タイプ	説明
output_value	a_sqlany_data_value	バインドされる出力値に関する情報。

備考

sqlany_get_bind_param_info() を使用して、この構造体にデータを移植できます。

a_sqlany_bind_param_info 構造体の使用例については、インストールされている SQL Anywhere の *sdk¥dbcapi¥examples* ディレクトリにあるサンプルファイルを参照してください。

参照

- sqlany_execute メソッド [SQL Anywhere C]572 ページ

a_sqlany_column_info 構造体

カラムのメタデータ情報を返します。

構文

```
public typedef struct a_sqlany_column_info
```

メンバー

メンバー名	タイプ	説明
max_size	size_t	このカラムに格納できるデータ値の最大サイズ。
name	char *	カラムの名前 (NULL で終了)。 結果セットのオブジェクトが解放されていない場合は、文字列を参照できます。
native_type	a_sqlany_native_type	データベースのカラムのネイティブタイプ。
nullable	sacapi_bool	カラムの値に NULL を指定できるかどうかを示します。
precision	unsigned short	精度。
scale	unsigned short	位取り。
type	a_sqlany_data_type	カラムのデータ型。

備考

sqlany_get_column_info() を使用して、この構造体にデータを移植できます。

a_sqlany_column_info 構造体の使用例については、インストールされている SQL Anywhere の *sdk¥dbcapi¥examples* ディレクトリにある次のサンプルファイルを参照してください。

- dbcapi_isql.cpp

a_sqlany_data_info 構造体

結果セット内のカラム値に関するメタデータ情報を返します。

構文

```
public typedef struct a_sqlany_data_info
```

メンバー

メンバー名	タイプ	説明
data_size	size_t	フェッチ可能なバイトの総数。 このフィールドは、フェッチ操作が成功した後のみ有効です。
is_null	sacapi_bool	最後にフェッチされたデータが NULL かどうかを示します。 このフィールドは、フェッチ操作が成功した後のみ有効です。
type	a_sqlany_data_type	カラムに格納されているデータ型

備考

sqlany_get_data_info() を使用して、フェッチ操作によって最後に取得されたデータに関する情報をこの構造体に移植できます。

a_sqlany_data_info 構造体の使用例については、インストールされている SQL Anywhere の *sdk¥dbcapi¥examples* ディレクトリにある次のサンプルファイルを参照してください。

参照

- sqlany_get_data_info メソッド [SQL Anywhere C]580 ページ

a_sqlany_data_value 構造体

データ値の属性に関する説明を返します。

構文

```
public typedef struct a_sqlany_data_value
```

メンバー

メンバー名	タイプ	説明
buffer	char *	ユーザが指定するデータバッファへのポインタ。
buffer_size	size_t	バッファのサイズ。
is_null	sacapi_bool *	最後にフェッチされたデータが NULL かどうかを示すポインタ。
length	size_t *	バッファ内の有効なバイト数へのポインタ。この値は buffer_size 未満である必要があります。
type	a_sqlany_data_type	データ型。

備考

a_sqlany_data_value 構造体の使用例については、インストールされている SQL Anywhere の *sdk¥dbcapi¥examples* ディレクトリにある次のサンプルファイルを参照してください。

- dbcapi_isql.cpp
- fetching_a_result_set.cpp
- send_retrieve_full_blob.cpp
- preparing_statements.cpp

SQLAnywhereInterface 構造体

SQL Anywhere C API のインタフェース構造体です。

構文

```
public typedef struct SQLAnywhereInterface
```

メンバー

メンバー名	タイプ	説明
dll_handle	void *	DLL ハンドル
initialized	int	初期化されているかどうかを識別するためのフラグ。
sqlany_affected_rows	void *	sqlany_affected_rows() 関数へのポインタ
sqlany_bind_param	void *	sqlany_bind_param() 関数へのポインタ
sqlany_cancel	void *	sqlany_cancel() 関数へのポインタ
sqlany_clear_error	void *	sqlany_clear_error() 関数へのポインタ
sqlany_client_version	void *	sqlany_client_version() 関数へのポインタ
sqlany_client_version_ex	void *	sqlany_client_version_ex() 関数へのポインタ
sqlany_commit	void *	sqlany_commit() 関数へのポインタ
sqlany_connect	void *	sqlany_connect() 関数へのポインタ
sqlany_describe_bind_param	void *	sqlany_describe_bind_param() 関数へのポインタ
sqlany_disconnect	void *	sqlany_disconnect() 関数へのポインタ
sqlany_error	void *	sqlany_error() 関数へのポインタ
sqlany_execute	void *	sqlany_execute() 関数へのポインタ
sqlany_execute_direct	void *	sqlany_execute_direct() 関数へのポインタ

メンバー名	タイプ	説明
sqlany_execute_immediate	void *	sqlany_execute_immediate() 関数へのポインタ
sqlany_fetch_absolute	void *	sqlany_fetch_absolute() 関数へのポインタ
sqlany_fetch_next	void *	sqlany_fetch_next() 関数へのポインタ
sqlany_fini	void *	sqlany_fini() 関数へのポインタ
sqlany_fini_ex	void *	sqlany_fini_ex() 関数へのポインタ
sqlany_free_connection	void *	sqlany_free_connection() 関数へのポインタ
sqlany_free_stmt	void *	sqlany_free_stmt() 関数へのポインタ
sqlany_get_bind_param_info	void *	sqlany_get_bind_param_info() 関数へのポインタ
sqlany_get_column	void *	sqlany_get_column() 関数へのポインタ
sqlany_get_column_info	void *	sqlany_get_column_info() 関数へのポインタ
sqlany_get_data	void *	sqlany_get_data() 関数へのポインタ
sqlany_get_data_info	void *	sqlany_get_data_info() 関数へのポインタ
sqlany_get_next_result	void *	sqlany_get_next_result() 関数へのポインタ
sqlany_init	void *	sqlany_init() 関数へのポインタ
sqlany_init_ex	void *	sqlany_init_ex() 関数へのポインタ
sqlany_make_connection	void *	sqlany_make_connection() 関数へのポインタ

メンバー名	タイプ	説明
sqlany_make_connection_ex	void *	sqlany_make_connection_ex() 関数へのポインタ
sqlany_new_connection	void *	sqlany_new_connection() 関数へのポインタ
sqlany_new_connection_ex	void *	sqlany_new_connection_ex() 関数へのポインタ
sqlany_num_cols	void *	sqlany_num_cols() 関数へのポインタ
sqlany_num_params	void *	sqlany_num_params() 関数へのポインタ
sqlany_num_rows	void *	sqlany_num_rows() 関数へのポインタ
sqlany_prepare	void *	sqlany_prepare() 関数へのポインタ
sqlany_reset	void *	sqlany_reset() 関数へのポインタ
sqlany_rollback	void *	sqlany_rollback() 関数へのポインタ
sqlany_send_param_data	void *	sqlany_send_param_data() 関数へのポインタ
sqlany_sqlstate	void *	sqlany_sqlstate() 関数へのポインタ

参照

- sqlany_initialize_interface メソッド [SQL Anywhere C]583 ページ

SACAPI_ERROR_SIZE 変数

最小エラーバッファサイズを返します。

構文

```
#define SACAPI_ERROR_SIZE
```

SQLANY_API_VERSION_1 変数

API バージョンを示すために定義します。

構文

```
#define SQLANY_API_VERSION_1
```

SQLANY_API_VERSION_2 変数

バージョン 2 では "_ex" 関数、および要求をキャンセルする機能が追加されました。

構文

```
#define SQLANY_API_VERSION_2
```

SQL Anywhere 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ファンクション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。Windows Mobile では、外部関数を呼び出すことができません。

この項では、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、フォルダ *%SQLANYSAMP16%¥SQLAnywhere¥ExternalProcedures* に格納されています。

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。

この項で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。

SQL Anywhere 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。「MAPI と SMTP のシステムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』を参照してください。

外部呼び出しを使ったプロシージャと関数

ここでは、外部呼び出しを使ったプロシージャと関数の例を示します。

注意
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。

構文

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;
```

この方法で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、SQL Anywhere 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、関数が原因で障害が発生すると、データベースサーバは終了されます。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。「外部関数のプロトタイプ」603 ページを参照してください。

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ファンクションによって、呼び出し元の環境に返すことができます。

他の文は使用不可

外部関数を参照するストアドプロシージャまたはファンクションに、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ファンクションの目的は、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。

オペレーティングシステムに依存する呼び出し

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。

参照

- 「CREATE FUNCTION 文 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE FUNCTION 文 [外部呼び出し]」『SQL Anywhere サーバ SQL リファレンス』
- 「SQL Anywhere 外部環境のサポート」619 ページ

外部関数のプロトタイプ

この項では、C や C++ で記述された関数で使用するインタフェースについて説明します。

インタフェースは、SQL Anywhere インストールディレクトリの *SDK¥Include* サブフォルダのヘッダファイル *extfnapi.h* によって定義されます。こ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'
  LANGUAGE C_ESQL32;
```

C/C++ 関数宣言は、次のようにします。

void *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。

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、外部ライブラリで次の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 );
}
```

参照

- 「an_extfn_api 構造体」606 ページ

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、外部ライブラリで次の関数をエクスポートします。

構文

extern "C" void SQL_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。

```
extern "C" void SQL_CALLBACK extfn_cancel( void *cancel_handle )
{
   *(short *)cancel_handle = 1;
}
```

参照

- 「an_extfn_api 構造体」606 ページ

extfn_post_load_library メソッド

この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 SQL_CALLBACK 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

```
extern "C" __declspec( dllexport )
void SQL_CALLBACK extfn_post_load_library( void )
{
   MessageBox( NULL, "Library loaded",
         "SQL Anywhere",
         MB_OK | MB_TASKMODAL );
}
```

参照

● 「an_extfn_api 構造体」606 ページ

extfn_pre_unload_library メソッド

この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 SQL_CALLBACK 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

```
extern "C" __declspec( dllexport )
void SQL_CALLBACK extfn_pre_unload_library( void )
{
   MessageBox( NULL, "Library unloading",
```

```
        "SQL Anywhere",
        MB_OK | MB_TASKMODAL );
}
```

参照

- 「an_extfn_api 構造体」606 ページ

an_extfn_api 構造体

呼び出し元の SQL 環境と通信するために使用します。

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 = extapi->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 = extapi->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 );
extapi->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;
extapi->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。

```
extern "C" __declspec( dllexport )
void my_external_proc( an_extfn_api *extapi, void *arg_handle )
{
    short               result;
    short               canceled;
    an_extfn_value      arg;

    canceled = 0;
    extapi->set_cancel( arg_handle, &canceled );

    result = extapi->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。

参照

- 「an_extfn_value 構造体」607 ページ
- 「extfn_cancel メソッド」604 ページ

an_extfn_value 構造体

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。「Embedded SQL のデータ型」485 ページを参照してください。

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@c:¥¥project¥¥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 = extapi->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 = extapi->get_piece( arg_handle, 1, &arg, offset );
    }
```

参照

- 「an_extfn_api 構造体」606 ページ

an_extfn_result_set_info 構造体

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。「an_extfn_result_set_column_info 構造体」610 ページを参照してください。
- **column_data_values**　結果セットカラムデータの記述へのリンク。「an_extfn_result_set_column_data 構造体」611 ページを参照してください。

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;

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;
```

参照

- 「an_extfn_result_set_column_info 構造体」610 ページ
- 「an_extfn_result_set_column_data 構造体」611 ページ

an_extfn_result_set_column_info 構造体

結果セットを記述するために使用されます。

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。「Embedded SQL のデータ型」485 ページを参照してください。
- **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
// 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;

extapi->set_value( arg_handle,
            EXTFN_RESULT_SET_ARG_NUM,
```

```
            (an_extfn_value *)&rs_info,
            EXTFN_RESULT_SET_DESCRIBE );
```

参照

- 「an_extfn_result_set_info 構造体」609 ページ
- 「an_extfn_result_set_column_data 構造体」611 ページ

an_extfn_result_set_column_data 構造体

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";

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;

extapi->set_value(  arg_handle,
            EXTFN_RESULT_SET_ARG_NUM,
            (an_extfn_value *)&rs_info,
            EXTFN_RESULT_SET_NEW_ROW_FLUSH );
```

参照

- 「an_extfn_result_set_info 構造体」609 ページ
- 「an_extfn_result_set_column_info 構造体」610 ページ

外部関数呼び出しインタフェースのメソッド

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ファンクション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 "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
     )
{
   return TRUE;
}

extern "C" __declspec( dllexport )
a_sql_uint32 extfn_use_new_api( void )
{
   return( EXTFN_API_VERSION );
}

extern "C" __declspec( dllexport )
void mystring( an_extfn_api *extapi, void *arg_handle )
{
   short              result;
   an_extfn_value      arg;
   unsigned           offset;
   char               *string;

   result = extapi->get_value( arg_handle, 1, &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( &string[offset], arg.data, arg.piece_len );
       offset += arg.piece_len;
       string[offset] = '¥0';
       if( arg.piece_len == 0 ) break;
       result = extapi->get_piece( arg_handle, 1, &arg, offset );
   }
   MessageBoxA( NULL, string,
        "SQL Anywhere",
        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 );
extapi->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。次のコードフラグメントがその例です。

```
extern "C" __declspec( dllexport )
void extfn_cancel( void *cancel_handle )
{
    *(short *)cancel_handle = 1;
}

extern "C" __declspec( dllexport )
void mystring( an_extfn_api *api, void *arg_handle )
{
.
.
.
```

```
    short  canceled = 0;

    extapi->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;
}
```

注意

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。

SQL Anywhere インストールディレクトリの *SDK¥Include* フォルダのヘッダファイル *extfnapi.h* に、詳細な注意事項が記載されています。

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。

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
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。

データ型の処理

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

SQL データ型	sqldef.h	C データ型
[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;

extapi->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;
}
```

データ型の詳細については、「Embedded SQL のホスト変数」489 ページを参照してください。

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 );
extapi->set_value( arg_handle, 0, &retval, 0 );
```

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。このパラメータでアンロードするライブラリの名前を指定します。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。

SQL Anywhere 外部環境のサポート

SQL Anywhere では、6 つの外部ランタイム環境をサポートしています。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、Perl、PHP、または Microsoft .NET Framework Common Language Runtime (CLR) に基づく C# や 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。

これまで SQL Anywhere では、C または C++ で記述されたコンパイル済みネイティブ関数を呼び出すことができました。ただし、これらのプロシージャがデータベースサーバで実行されるとき、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が常にデータベースサーバによってロードされ、ネイティブ関数への呼び出しがデータベースサーバによって行われていました。この方法には、ネイティブ関数が原因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というリスクがあります。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、SQL Anywhere での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、Perl スクリプトなどの Perl または PHP 外部オブジェクト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 Perl または 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。

参照

- 「ALTER EXTERNAL ENVIRONMENT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「COMMENT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE FUNCTION 文 [外部呼び出し]」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE PROCEDURE 文 [外部呼び出し]」『SQL Anywhere サーバ SQL リファレンス』
- 「INSTALL EXTERNAL OBJECT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「REMOVE EXTERNAL OBJECT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「START EXTERNAL ENVIRONMENT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「STOP EXTERNAL ENVIRONMENT 文」『SQL Anywhere サーバ SQL リファレンス』

CLR 外部環境

SQL Anywhere では、CLR ストアドプロシージャおよび関数をサポートしています。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 C# または Visual Basic などの .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 .NET 実行ファイル内) で行われます。この .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 .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。.NET バージョン 2.0 のみがサポートされています。

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が、その実行時に 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。.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 .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、*%SQLANYSAMP16%¥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 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。データベースサーバが CLR 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 実行ファイルを検出できない可能性があります。CLR 実行ファイルは *dbextclr16.exe* です。このファイルが、使

用しているデータベースサーバのバージョンに応じて *%SQLANY16%¥Bin32* または *%SQLANY16%¥Bin64* にあることを確認してください。

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。ただし、CLR 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 CLR 文を使用して接続のための CLR インスタンスを解放します。

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。*dbextclr16.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、さまざまなバージョンが SQL Anywhere に含まれています。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 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、.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();
```

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、*%SQLANYSAMP16%¥SQLAnywhere¥ExternalEnvironments¥CLR* ディレクトリのサンプルを参照してください。

参照

- 「SQL 関数」『SQL Anywhere サーバ SQL リファレンス』
- 「ユーザ定義関数」『SQL Anywhere サーバ SQL の使用法』
- 「システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「プロシージャ」『SQL Anywhere サーバ SQL の使用法』
- 「名前付きパラメータ」『SQL Anywhere サーバ SQL リファレンス』

ESQL 外部環境と ODBC 外部環境

これまで SQL Anywhere では、C または C++ で記述されたコンパイル済みネイティブ関数を呼び出すことができました。ただし、これらのプロシージャがデータベースサーバで実行されるとき、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が常にデータベースサーバによってロードされ、ネイティブ関数への呼び出しがデータベースサーバによって行われていました。データベースサーバでこれらのネイティブ関数の呼び出しを行うと効率が最も良くなる一方で、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外部環境でコンパイル済みネイティブ関数を実行するオプションが導入されました。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。

1. 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ない。
2. ODBC、Embedded SQL (ESQL)、または SQL Anywhere 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る。
3. 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る。
4. 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る。また、その逆も可能である。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ネイティブ関数呼び出しインタフェースを使用する必要があります。このインタフェースについては、「SQL Anywhere 外部呼び出しインタフェース」601 ページを参照してください。

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ファンクションを 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、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。*dbexternc16* 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、さまざまなバージョンが SQL Anywhere に含まれています。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 *dbexternc16* 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 *dbexternc16* 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、*dbexternc16* 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 SQL Anywhere 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、SQL Anywhere 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 (SQL Anywhere ソフトウェアがフォルダ *c:¥sa16* 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:¥sa16¥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 (SQL Anywhere ソフトウェアがフォルダ *c:¥sa16* 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:¥sa16¥sdk¥include c:¥sa16¥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 SQL Anywhere 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 SQL Anywhere 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 (SQL Anywhere ソフトウェアがフォルダ *c:¥sa16* 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:¥sa16¥SDK¥C¥sacapidll.c
    /Ic:¥sa16¥SDK¥Include c:¥sa16¥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。SQL Anywhere 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、*dbexternc16* 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 *dbexternc16* 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
```

dbexternc16 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 *dbexternc16* 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 *dbexternc16* 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、*dbexternc16* 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ネイティブ関数呼び出しインタフェースを使用する必要があります。このインタフェースの詳細については、「SQL Anywhere 外部呼び出しインタフェース」601 ページを参照してください。

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、SQL Anywhere 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。SQL Anywhere 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。SQL Anywhere 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。SQL Anywhere 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。SQL Anywhere 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、*%SQLANYSAMP16%¥SQLAnywhere¥ExternalEnvironments¥ExternC* のサンプルを参照してください。

参照

- 「SQL Anywhere 外部呼び出しインタフェース」601 ページ

Java 外部環境

SQL Anywhere では、Java ストアドプロシージャおよび関数をサポートしています。Java 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 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。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。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。

1. 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。

2. SQL Anywhere 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';
```

次に例を示します。

```
ALTER EXTERNAL ENVIRONMENT JAVA
  LOCATION 'c:¥¥jdk1.7.0¥¥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 と *return-descriptor* :
Z
| **B**
| **S**
| **I**
| **J**
| **F**
| **D**
| **C**
| **V**
| **[***descriptor*
| **L***class-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 と *return-descriptor* の意味は次のとおりです。

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、ドットを / に置き換えたものとします。例 : java/lang/String
S	short
V	void
Z	Boolean

フィールドタイプ	Java データ型
[	配列の各次元ごとに 1 つ使用

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 SQL Anywhere." );
    }
}
```

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ています。Interactive SQL で次の SQL 文を実行することでインタフェースをテストします。

```
CALL HelloDemo('SQL Anywhere');
```

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;
```

関数 whoAreYou は java.lang.String を返すものとして記述されています。Interactive SQL で次の SQL 文を実行することでインタフェースをテストします。

```
SELECT WhoAreYou();
```

Interactive SQL の **[結果]** ウィンドウの応答を確認してください。

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らない」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 *dbjdbc16/libdbjdbc16* 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。

参照

- 「レッスン 4 : Java クラスのメソッドの呼び出し」411 ページ
- 「Java メソッドから結果セットを返す方法」418 ページ
- 「データベースにおける Java」405 ページ

PERL 外部環境

SQL Anywhere では、Perl ストアドプロシージャおよび関数をサポートしています。Perl 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 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。

データベースでの Perl 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。

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、SQL Anywhere 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。

   DBD::SQLAnywhere ドライバのインストールの詳細については、「Perl DBI サポート」649 ページを参照してください。

上記の前提条件に加え、データベース管理者は SQL Anywhere 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。

Windows の場合、次のコマンドを SQL Anywhere インストール環境の *SDK¥PerlEnv* のサブフォルダから実行し、外部環境モジュールをインストールします。

```
perl Makefile.PL
nmake
nmake install
```

UNIX の場合、次のコマンドを SQL Anywhere インストール環境の *sdk/perlenv* のサブフォルダから実行し、外部環境モジュールをインストールします。

```
perl Makefile.PL
make
make install
```

Perl 外部環境モジュールが構築されてインストールされると、データベースでの Perl のサポートを使用できるようになります。データベースでの Perl のサポートを使用できるのは、SQL Anywhere バージョン 11 以降のデータベースのみです。SQL Anywhere 10 データベースがロードされている場合、データベースでの Perl のサポートを使用しようとすると、外部環境がサポートされていないことを示すエラーが返されます。

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。これは、次の文を実行して確認できます。

```
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';
```

次に例を示します。

```
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;
```

Perl 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 'perl-script'
  NEW
  FROM VALUE 'perl-statements'
  ENVIRONMENT PERL;
```

Perl 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 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。

- **$sa_perl_return**　これは、関数呼び出しの戻り値を設定するために使用します。
- **$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、*%SQLANYSAMP16%¥SQLAnywhere¥ExternalEnvironments¥Perl* ディレクトリのサンプルを参照してください。

PHP 外部環境

SQL Anywhere では、PHP ストアドプロシージャおよび関数をサポートしています。PHP 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 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。

PHP をデータベースで使用するには、PHP をインストールし、SQL Anywhere PHP ドライバをインストールし、SQL Anywhere PHP 外部環境モジュールをインストールし、PHP がこれらのコンポーネントを検索してロードできるように設定する必要があります。次の項では、これを行う方法を説明します。

PHP 外部環境モジュールの Windows または UNIX 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 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。また、SQL Anywhere 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 ドライバ (SQL Anywhere に付属) 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。「PHP クライアントの配備」996 ページを参照してください。

内容と備考

上記 2 つの前提条件に加え、データベース管理者は SQL Anywhere 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。SQL Anywhere 配布には、いくつかのバージョンの PHP のビルド済みモジュールが含まれています。ビルド済みモジュールをインストールするには、適切なドライバモジュールを 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。

♦ タスク

1. 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。

2. 目的の外部環境 PHP モジュールを、SQL Anywhere のインストールディレクトリから PHP のインストールまたは拡張ディレクトリにコピーします。選択したバージョンを反映するために x.y を変更します。

   Windows の場合、次のようなコマンドを使用します。

   ```
   copy "%SQLANY16%¥Bin32¥php-5.x.y_sqlanywhere_extenv16.dll" php-ext-dir
   ```

   UNIX の場合、次のようなコマンドを使用します。

   ```
   cp $SQLANY16/bin32/php-5.x.y_sqlanywhere_extenv16.so php-ext-dir
   ```

   上記の例では、*php-ext-dir* は PHP 拡張ディレクトリ (*c:¥php¥ext* など) へのパスです。

3. 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 x.y を変更します。

   Windows の場合、次の行を追加します。

   ```
   extension=php-5.x.y_sqlanywhere_extenv16.dll
   ```

   UNIX の場合、次の行を追加します。

   ```
   extension=php-5.x.y_sqlanywhere_extenv16.so
   ```

4. *php.ini* を保存して閉じます。

5. SQL Anywhere PHP ドライバも、SQL Anywhere のインストールディレクトリから 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。ファイル名のプレフィクスは **php-5.x.y_sqlanywhere** のようになります (x および y はバージョン番号を表します)。ステップ 2 でコピーしたファイルのバージョン番号と一致する必要があります。

結果

外部環境モジュールがインストールされます。

PHP をデータベースから使用する方法

データベースでの PHP のサポートを使用できるのは、SQL Anywhere バージョン 11 以降のデータベースのみです。SQL Anywhere 10 データベースがロードされている場合、データベースでの PHP のサポートを使用しようとすると、外部環境がサポートされていないことを示すエラーが返されます。

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する必要があります。

```
ALTER EXTERNAL ENVIRONMENT PHP
  LOCATION 'php-path';
```

次に例を示します。

```
ALTER EXTERNAL ENVIRONMENT PHP
  LOCATION 'c:¥¥php¥¥php-5.4.8-win32¥¥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.y_sqlanywhere* および *php-5.x.y_sqlanywhere_extenv16* 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。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.y_sqlanywhere* および *php-5.x.y_sqlanywhere_extenv16* の両モジュールが、*php.ini* ファイルの Dynamic Extensions セクションに存在す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。SQL Anywhere *bin32* フォルダが PATH に含まれていることを確認します。
- Windows では、32 ビット DLLs (*dbcapi.dll*、*dblib16.dll*、*dbicu16.dll*、*dbicudt16.dll*、*dblgen16.dll*、*dbextenv16.dll*) が検出できることを確認します。SQL Anywhere *bin32* フォルダが PATH に含まれていることを確認します。
- Linux、UNIX、Mac OS X では、32 ビットの共有オブジェクト (*libdbcapi_r*、*libdblib16_r*、*libdbicu16_r*、*libdbicudt16*、*dblgen16.res*、*libdbextenv16_r*) が検出できることを確認します。SQL Anywhere *bin32* 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、次に必要な 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 SQL Anywhere 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、*%SQLANYSAMP16%¥SQLAnywhere¥ExternalEnvironments¥PHP* ディレクトリのサンプルを参照してください。

Perl DBI サポート

DBD::SQLAnywhere は DBI 用の SQL Anywhere 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。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。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。

DBD::SQLAnywhere

DBD::SQLAnywhere は、Tim Bunce によって作成された Database Independent Interface for Perl (DBI) モジュールのドライバです。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。

稼働条件

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。
- 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。

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 SQL Anywhere データベースにアクセスします。

前提条件

- ActivePerl 5.6.0 以降をインストールします。ActivePerl インストーラを使用して、Perl をインストールし、コンピュータを設定できます。Perl を再コンパイルする必要はありません。
- Microsoft Visual Studio をインストールして環境を設定します。

  インストール時に環境を設定しなかった場合は、作業を行う前に PATH、LIB、INCLUDE 環境変数を正しく設定します。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、Visual Studio 2005 または 2008 のインストール環境の *vc¥bin* サブフォルダには *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、*vcvarsamd64.bat* など、このバッチファイルの 64 ビットバージョンを探します。作業を続ける前に、新しいシステムコマンドプロンプトを開き、このバッチファイルを実行してください。

64 ビットの Visual C++ ビルド環境の設定の詳細については、http://msdn.microsoft.com/ja-jp/library/x4d2c09s.aspx を参照してください。

♦ タスク

1. 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* サブフォルダに移動します。

   別のシェルからは次の手順を使用できないことがあるため、システムコマンドプロンプトを使用することを強くおすすめします。

2. Perl Module Manager を使用して、次のコマンドを入力します。

   ```
   ppm query dbi
   ```

   ppm を実行できない場合は、Perl が正しくインストールされていることを確認してください。

   このコマンドを実行すると、次のような 2 行のテキストが生成されます。この場合、この情報は、ActivePerl バージョン 5.8.1 ビルド 807 が動作しており、DBI バージョン 1.38 がインストールされていることを示します。

   ```
   Querying target 1 (ActivePerl 5.8.1.807)
     1. DBI [1.38] Database independent interface for Perl
   ```

   それ以降のバージョンの Perl では、次のようなテーブルが表示される場合があります。この場合は、DBI バージョン 1.58 がインストールされていることを示します。

name	version	abstract	area
DBI	1.58	Database independent interface for Perl	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。インストールするには、ppm プロンプトで次のコマンドを入力します。

   ```
   ppm install dbi
   ```

3. コマンドプロンプトで、SQL Anywhere インストール環境の *SDK¥Perl* サブフォルダに移動します。

4. 次のコマンドを入力し、DBD::SQLAnywhere を構築してテストします。

   ```
   perl Makefile.PL
   nmake
   ```

   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** を実行し、部分的に構築されたターゲットを削除できます。

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 *SDK¥Perl* ディレクトリにコピーして、テストを実行します。

   ```
   copy "%SQLANYSAMP16%¥demo.db" .
   ```

```
dbsrv16 demo
nmake test
```

テストが行われない場合は、SQL Anywhere インストール環境の *bin32* または *bin64* サブフォルダ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。

```
nmake install
```

結果

DBI Perl 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。

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 SQL Anywhere データベースにアクセスします。

前提条件

ActivePerl 5.6.0 ビルド 616 以降と C コンパイラがインストールされている必要があります。

♦ タスク

1. DBI モジュールソースを、http://www.cpan.org からダウンロードします。
2. 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。
3. 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 DBI モジュールを構築します。

```
perl Makefile.PL
make
```

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** を使用し、部分的に構築されたターゲットを削除できます。

4. 次のコマンドを使用して、DBI モジュールをテストします。

```
make test
```

5. 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
```

6. SQL Anywhere の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、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。

シェル	使用するコマンド
sh、ksh、または bash	. *bin/sa_config.sh*
csh または tcsh	source *bin/sa_config.csh*

7. シェルプロンプトで、SQL Anywhere インストール環境の *sdk/perl* サブフォルダに移動します。

8. コマンドプロンプトで、次のコマンドを実行して DBD::SQLAnywhere を構築します。

```
perl Makefile.PL
make
```

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** を使用し、部分的に構築されたターゲットを削除できます。

9. 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 *sdk/perl* ディレクトリにコピーして、テストを実行します。

```
cp $SQLANYSAMP16/demo.db .
dbsrv16 demo
make test
```

テストが行われない場合は、SQL Anywhere インストール環境の *bin32* または *bin64* サブフォルダがパスに含まれていることを確認してください。

10. 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
```

結果

DBI Perl 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。

次の手順

オプションとして、ここで DBI ソースツリーを削除できます。このツリーは必要なくなりました。

DBD::SQLAnywhere を使用する Perl スクリプト

この項では、DBD::SQLAnywhere インタフェースを使用する Perl スクリプトを作成する方法の概要を説明します。DBD::SQLAnywhere は、DBI モジュールのドライバです。DBI モジュールの完全なドキュメントについては、オンラインで http://dbi.perl.org を参照してください。

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、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。たとえば、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;
```

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。

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。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、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、SQL Anywhere サンプルデータベースへの接続を開いて閉じます。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起動します。

```
#!/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 %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";
```

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、SQL Anywhere 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。

参照

- 「max_cursor_count オプション」『SQL Anywhere サーバ データベース管理』
- 「max_statement_count オプション」『SQL Anywhere サーバ データベース管理』

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__
```

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__
```

Python サポート

SQL Anywhere Python データベースインタフェース sqlanydb は、Python 言語のデータアクセス API です。この項では、SQL Anywhere を Python と一緒に使用する方法について説明します。

sqlanydb

SQL Anywhere Python データベースインタフェース sqlanydb は、Python 言語のデータアクセス API です。Python データベース 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メソッドを定義します。sqlanydb モジュールを使用すると、Python 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。

sqlanydb モジュールは、Marc-André Lemburg が作成した Python データベース API 仕様 v2.0 を拡張して実装しています。sqlanydb モジュールをインストールすると、Python 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。

Python データベース API 仕様 v2.0 の詳細については、http://www.python.org/dev/peps/pep-0249/を参照してください。

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。

稼働条件

sqlanydb モジュールには次のコンポーネントが必要です。

- Python。サポートされるバージョンのリストについては、http://www.sybase.com/detail?id=1068981 を参照してください。

- ctypes モジュール。ctypes モジュールがインストールされているかどうかをテストするには、コマンドプロンプトウィンドウを開いて 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。インストールについては、http://sourceforge.net/project/showfiles.php?group_id=71702 で SourceForge.net ファイルのセクションを参照してください。

  ctypes は Peak EasyInstall でもインストールされます。Peak EasyInstall をダウンロードするには、http://peak.telecommunity.com/DevCenter/EasyInstall にアクセスしてください。

Windows での Python サポートのインストール

Python サポートは、SQL Anywhere インストール環境の *SDK¥Python* サブフォルダから該当する設定 Python スクリプトを実行し、Windows に設定できます。

前提条件

Python および ctypes モジュールがインストールされていることを確認します。サポートされる Python のバージョンのリストについては、http://www.sybase.com/detail?id=1068981 を参照してください。

♦ タスク

1. システムのコマンドプロンプトで、SQL Anywhere インストール環境の *SDK¥Python* サブフォルダに移動します。

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. sqlanydb をテストするには、サンプルデータベースのコピーを現在のディレクトリに作成し、テストを実行します。

   ```
   newdemo
   dbsrv16 demo
   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。正常に動作すると、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、SQL Anywhere インストール環境の *bin32* または *bin64* サブフォルダがパスに含まれていることを確認してください。

結果

これで、sqlanydb モジュールが使用可能になりました。

UNIX と Mac OS X での Python サポートのインストール

Python サポートは、SQL Anywhere インストール環境の *sdk/python* サブフォルダから該当する設定 Python スクリプトを実行し、Windows に設定できます。

前提条件

Python および ctypes モジュールがインストールされていることを確認します。サポートされる Python のバージョンのリストについては、http://www.sybase.com/detail?id=1068981 を参照してください。

♦ タスク

1. SQL Anywhere の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、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 (64 ビットのソフトウェアがインストールされている場合は、*bin32* の代わりに *bin64* を使用できます)。

シェル	使用するコマンド
sh、ksh、または bash	. *bin32/sa_config.sh*
csh または tcsh	source *bin32/sa_config.csh*

2. シェルプロンプトで、SQL Anywhere インストール環境の *sdk/python* サブフォルダに移動します。

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. sqlanydb をテストするには、サンプルデータベースのコピーを現在のディレクトリに作成し、テストを実行します。

   ```
   newdemo
   dbsrv16 demo
   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。正常に動作すると、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、SQL Anywhere インストール環境の *bin32* または *bin64* サブフォルダがパスに含まれていることを確認してください。

結果

これで、sqlanydb モジュールが使用可能になりました。

sqlanydb を使用する Python スクリプト

この項では、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。API の完全なドキュメントについては、オンラインで http://www.python.org/dev/peps/pep-0249/を参照してください。

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、ファイルの先頭に次の行を含めて、sqlanydb モジュールをロードします。

```
import sqlanydb
```

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。

```
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 16 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、SQL Anywhere サンプルデータベースへの接続を開いて閉じます。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起動します。

```
import sqlanydb

# Create a connection object
con = sqlanydb.connect( userid="DBA",
                        password="sql" )
# 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、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

# Create a connection object
con = sqlanydb.connect( DSN="SQL Anywhere 16 Demo" )

# Close the connection
con.close()
```

参照

- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』

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

# Create a connection object, then use it to create a cursor
con = sqlanydb.connect( userid="DBA",
                        password="sql" )
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()
```

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

# Create a connection object, then use it to create a cursor
con = sqlanydb.connect( userid="DBA", pwd="sql" )
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

# Create a connection object, then use it to create a cursor
con = sqlanydb.connect( userid="DBA", pwd="sql" )
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。

データベースタイプの変換

データベースサーバから結果がフェッチされたときに、データベースタイプを Python オブジェクトにマッピングする方法を制御するには、変換コールバックを登録します。

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)
```

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_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)

# Create a connection object, then use it to create a cursor
con = sqlanydb.connect( userid="DBA",
                        password="sql" )
cursor = con.cursor()

# Execute a SQL string
sql = "SELECT * FROM Employees WHERE EmployeeID=105"
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()
```

PHP サポート

SQL Anywhere PHP 拡張

PHP (**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に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に立つ言語として使用できるように、PHP には SQL Anywhere を含む多くの一般的なデータベースから情報を取得する機能が含まれています。SQL Anywhere 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。SQL Anywhere 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、SQL Anywhere 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。

SQL Anywhere 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。

ビルド済みバージョンの PHP 拡張は Windows、Linux、および Solaris 用に用意されており、SQL Anywhere インストール環境のバイナリサブフォルダにインストールされます。SQL Anywhere PHP 拡張のソースコードは、SQL Anywhere インストール環境の *sdk¥php* サブフォルダにインストールされています。

最新の SQL Anywhere PHP ドライバについては、http://www.sybase.com/detail?id=1019698 を参照してください。

SQL Anywhere PHP 拡張のインストールについては、「PHP クライアントの配備」996 ページを参照してください。

PHP 拡張のテスト

簡単なチェックを行って、SQL Anywhere 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。

♦ タスク

1. SQL Anywhere インストール環境の *bin32* サブフォルダがパスに含まれていることを確認してください。SQL Anywhere 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 SQL Anywhere サンプルデータベースを起動します。

   ```
   dbsrv16 "%SQLANYSAMP16%¥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、SQL Anywhere インストール環境の *SDK¥PHP¥Examples* サブフォルダに移動します。実行プログラムディレクトリの **php** 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
   ```

   SQL Anywhere 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、SQL Anywhere データベースサーバを停止します。

結果

テストが成功し、SQL Anywhere PHP 拡張が正常に動作していることを示します。

参照

- 「PHP テストページの作成と実行」668 ページ

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。PHP のインストールの詳細については、http://us2.php.net/install を参照してください。

内容と備考

このプロシージャはすべての設定に適用されます。

♦ タスク

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 のユーザの場合は、ファイルを */Users/your-user-name/Sites/* に置きます。

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 と SQL Anywhere 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* サブフォルダから次のコマンドを実行します。

   ```
   apachectl start
   ```

6. Linux または Mac OS X では、提供されているスクリプトのいずれかを使用して SQL Anywhere の環境変数を設定します。

   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、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。

シェル	使用するコマンド
sh、ksh、または bash	. /bin32/sa_config.sh
csh または tcsh	source /bin32/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 SQL Anywhere 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)。

```
dbsrv16 "%SQLANYSAMP16%¥demo.db"
```

8. PHP と Web サーバが SQL Anywhere 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	使用する 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。

PHP スクリプトの開発

この項では、SQL Anywhere PHP 拡張を使用して SQL Anywhere データベースにアクセスする PHP スクリプトの作成方法について説明します。

ここで使用される例のソースコードは他の例と同様に、SQL Anywhere インストール環境の *SDK ¥PHP¥Examples* サブフォルダにあります。

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 SQL Anywhere 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 ?> のタグは、この間のコードを PHP が実行し、そのコードを PHP 出力に置き換えることを Web サーバに指定します。

この例のソースコードは、SQL Anywhere インストール環境の *connect.php* というファイルにあります。

```
<?php
 # Connect using the default user ID and password
```

```
$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、ローカルサーバで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、SQL Anywhere 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実行されている必要があります。

PHP を使用してデータベースからデータを取得する方法

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、SQL Anywhere 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。

この例のソースコードは、SQL Anywhere インストール環境の *query.php* というファイルにあります。

```
<?php
 # Connect using the default user ID and password
 $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。

この例のソースコードは、SQL Anywhere インストール環境の *fetch.php* というファイルにあります。

```
<?php
 # Connect using the default user ID and password
 $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。

この例のソースコードは、SQL Anywhere インストール環境の *fetch_assoc.php* というファイルにあります。

```
<?php
 # Connect using the default user ID and password
 $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。

この例のソースコードは、SQL Anywhere インストール環境の *nested.php* という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。

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。

この例のソースコードは、SQL Anywhere インストール環境の *webisql.php* という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、ユーザによって入力される値に基づいてカスタマイズされた 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。

PHP アプリケーション内の BLOB

SQL Anywhere データベースでは、あらゆる種類のデータもバイナリラージオブジェクト (BLOB) として格納できます。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、SQL Anywhere 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('ianywhere_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ので、イメージが正しく表示されます。

UNIX と Mac OS X で SQL Anywhere PHP 拡張を構築する方法

UNIX および Mac OS X で、SQL Anywhere PHP 拡張を使用して PHP を SQL Anywhere に接続するには、SQL Anywhere PHP 拡張のファイルを PHP のソースツリーに追加し、PHP を再コンパイルする必要があります。

UNIX または Mac OS X での SQL Anywhere PHP 拡張ファイルの PHP ソースツリーへの追加

このトピックでは、SQL Anywhere PHP 拡張のファイルを PHP のソースツリーに追加するために必要な手順について説明します。

前提条件

UNIX と Mac OS X で SQL Anywhere PHP 拡張を使用する場合、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 インストール環境。Apache 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。
- SQL Anywhere PHP 拡張のソースコード。http://download.sybase.com/ianywhere/php/2.0.3/src/sasql_php.zip からダウンロードできます。

  sqlpp および *libdblib16.so* (UNIX の場合) または *libdblib16.dylib* (Mac OS X の場合) がインストールされている必要があります (SQL Anywhere *lib32* ディレクトリを確認してください)。
- PHP ソースコード。http://www.php.net からダウンロードできます。

  サポートされるバージョンのリストについては、http://www.sybase.com/detail?id=1068981 を参照してください。
- Apache Web サーバのソースコード。http://httpd.apache.org からダウンロードできます。

  ビルド済みバージョンの Apache を使用する場合は、**apache** と **apache-devel** がインストールされていることを確認してください。
- Unified ODBC PHP 拡張を使用する場合は、*libdbodbc16.so* (UNIX の場合) または *libdbodbc16.dylib* (Mac OS X の場合) がインストールされている必要があります (SQL Anywhere *lib32* ディレクトリを確認してください)。

UNIX インストールディスクから次のバイナリをインストールする必要があります (すでにインストールされていない場合)。これらのバイナリは RPM パッケージとして提供されています。

- make
- automake
- autoconf
- libtool (Mac OS X の場合は glibtool)
- makeinfo
- bison
- gcc
- cpp
- glibc-devel
- kernel-headers
- flex

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。

内容と備考

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。

♦ タスク

1. SQL Anywhere PHP 拡張のソースコードを、http://www.sybase.com/detail?id=1019698 からダウンロードします。「**Building the Driver from Source**」というタイトルの項を探してください。

2. SQL Anywhere PHP 拡張を保存したディレクトリから PHP ソースツリーの *ext* サブフォルダにファイルを抽出します (Mac OS X ユーザの場合は **tar** を **gnutar** と置換します)。

   ```
   $ tar -xzf sasql_php.zip -C PHP-source-directory/ext/
   ```

   次の例は、PHP バージョン 5.2.11 用のコマンドです。以下の php-5.2.11 を、使用している PHP のバージョンに変更する必要があります。

   ```
   $ tar -xzf sqlanywhere_php-1.0.8.tar.gz -C ~/php-5.2.11/ext
   ```

3. PHP に拡張を認識させます。

   ```
   $ cd PHP-source-directory/ext/sqlanywhere
   $ touch *
   $ cd ~/PHP-source-directory
   $ ./buildconf
   ```

   次の例は、PHP バージョン 5.2.11 用のコマンドです。以下の php-5.2.11 を、使用している PHP のバージョンに変更する必要があります。

   ```
   $ cd ~/php-5.2.11/ext/sqlanywhere
   $ touch *
   ```

```
$ cd ~/php-5.2.11
$ ./buildconf
```

4. PHP が拡張を認識していることを確認します。

```
$ ./configure -help | egrep sqlanywhere
```

結果

PHP による SQL Anywhere 拡張の認識が成功した場合は、次のテキストが表示されます。

```
--with-sqlanywhere=[DIR]
```

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 (http://sqlanywhere-forum.sybase.com) に投稿してサポートを受けてください。

Apache と PHP をコンパイルする方法

PHP は、Web サーバ (Apache など) の共有モジュールとして、または CGI 実行プログラムとしてコンパイルできます。PHP でサポートされていない Web サーバを使用している場合、または PHP スクリプトを Web ページではなくコマンドシェルで実行したい場合は、PHP を CGI 実行プログラムとしてコンパイルします。それ以外の場合、PHP をインストールして Apache で動作するようにするには、Apache モジュールとしてコンパイルします。

Apache モジュールとしての PHP のコンパイル

PHP をインストールして Apache で動作するようにするには、Apache モジュールとしてコンパイルします。

前提条件

次の手順を使用して、Apache が共有モジュールを認識するように設定します。

> **注意**
> Mac OS X には Apache Web サーバがプリインストールされています。

1. Apache を設定して共有モジュールを認識させます。

   Apache ファイルが抽出されたディレクトリから、次のようなコマンドを実行します。

   ```
   $ cd Apache-source-directory
   $ ./configure --enabled-shared=max --enable-module=most --prefix=/Apache-installation-directory
   ```

   次の例は、Apache バージョン 2.2.9 用のコマンドです。**apache_2.2.9** を、使用している Apache のバージョンに変更する必要があります。

   ```
   $ cd ~/apache_2.2.9
   $ ./configure --enabled-shared=max --enable-module=most --prefix=/usr/local/web/apache
   ```

2. 関連するコンポーネントを再コンパイルしてインストールします。

```
$ make
$ make install
```

これで、Apache モジュールとして動作するように PHP をコンパイルできます。

♦ タスク

1. SQL Anywhere の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、SQL Anywhere がインストールされているディレクトリ (デフォルトでは */opt/sqlanywhere16*) から適切なコマンドを入力します。Mac OS X では、デフォルトディレクトリは */Applications/SQLAnywhere16/System* です。

シェル	使用するコマンド
sh, ksh, bash	. ./bin32/sa_config.sh
csh, tcsh	source ./bin32/sa_config.csh

2. PHP を Apache モジュールとして設定して、SQL Anywhere PHP 拡張を含めます。

   次のコマンドを実行します。

```
$ cd PHP-source-directory
$ ./configure --with-sqlanywhere --with- apxs=/Apache-installation-directory/bin/apxs
```

   次の例は、PHP バージョン 5.2.11 用のコマンドです。php-5.2.11 を、使用している PHP のバージョンに変更する必要があります。

```
$ cd ~/php-5.2.11
$ ./configure --with-sqlanywhere --with- apxs=/usr/local/web/apache/bin/apxs
```

   configure スクリプトによって、インストールされている SQL Anywhere のバージョンとロケーションの特定が試行されます。コマンドの出力に次のような行が表示されます。

```
checking for SQL Anywhere support... yes
checking    SQL Anywhere install dir... /opt/sqlanywhere16
checking    SQL Anywhere version... 16
```

3. 関連するコンポーネントを再コンパイルします。

```
$ make
```

4. ライブラリが正常にリンクされていることを確認します。

   ●Linux ユーザの場合 (次の例は、PHP バージョン 5 を使用していると仮定)

```
ldd ./.libs/libphp5.so
```

   ●Mac OS X ユーザの場合：

   httpd.conf 設定ファイルを参照して、*libphp5.so* の場所を特定します。次のコマンドによりチェックを実行します。

```
otool -L $LIBPHP5_DIR/libphp5.so
```

サーバ設定によると、*libphp5.so* は *$LIBPHP5_DIR* フォルダにあります。

このコマンドは、*libphp5.so* が使用するライブラリのリストを出力します。*libdblib16.so* がリストにあることを確認してください。

5. PHP バイナリを Apache の *lib* ディレクトリにインストールします。

```
$ make install
```

6. 検証を実行します。検証は PHP により自動的に行われます。ユーザによる確認が必要なのは、*httpd.conf* 設定ファイルが検証され、Apache が *.php* ファイルを PHP スクリプトとして認識されるかどうかの確認だけです。

   httpd.conf は、*Apache* ディレクトリの *conf* サブフォルダに格納されています。

```
$ cd Apache-installation-directory/conf
```

   次に例を示します。

```
$ cd /usr/local/web/apache/conf
```

   httpd.conf のバックアップコピーを作成してからファイルを編集します (**pico** を好みのテキストエディタと置き換えることができます)。

```
$ cp httpd.conf httpd.conf.backup
$ pico httpd.conf
```

   次の行を *httpd.conf* に追加するか、コメント解除します (ファイル内の同じ場所にはありません)。

```
LoadModule php5_module   libexec/libphp5.so
AddModule mod_php5.c
AddType application/x-httpd-php .php
AddType application/x-httpd-php-source .phps
```

> **注意**
> Mac OS X の場合は、最後の 2 行を *httpd_macosxserver.conf* に追加するか、コメント解除する必要があります。

   最初の 2 行により、PHP コードの解釈に使用されるファイルに Apache がポイントされます。残りの 2 行により、拡張子が *.php* または *.phps* のファイルのファイルタイプが宣言されます。これにより、Apache はファイルを適切に認識および処理できるようになります。

結果

PHP は共有モジュールとして正常にコンパイルされます。

参照

- 「CGI 実行プログラムとしての PHP のコンパイル」681 ページ
- 「PHP テストページの作成と実行」668 ページ

CGI 実行プログラムとしての PHP のコンパイル

PHP でサポートされていない Web サーバを使用している場合、または PHP スクリプトを Web ページではなくコマンドシェルで実行したい場合は、PHP を CGI 実行プログラムとしてコンパイルします。

前提条件

この作業を実行するための前提条件はありません。

♦ タスク

1. SQL Anywhere の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、SQL Anywhere がインストールされているディレクトリ (デフォルトでは */opt/sqlanywhere16*) から適切なコマンドを入力します。Mac OS X では、デフォルトディレクトリは */Applications/SQLAnywhere16/System* です。

シェル	使用するコマンド
sh, ksh, bash	. ./bin32/sa_config.sh
csh, tcsh	source ./bin32/sa_config.csh

2. PHP を CGI 実行プログラムとして設定して、SQL Anywhere PHP 拡張を含めます。

   PHP ファイルが抽出されたディレクトリから、次のコマンドを実行します。

```
$ cd PHP-source-directory
$ ./configure --with-sqlanywhere
```

   次の例は、PHP バージョン 5.2.11 用のコマンドです。php-5.2.11 を、使用している PHP のバージョンに変更する必要があります。

```
$ cd ~/php-5.2.11
$ ./configure --with-sqlanywhere
```

   設定スクリプトによって、インストールされている SQL Anywhere のバージョンとロケーションの特定が試行されます。コマンドの出力に次のような行が表示されます。

```
checking for SQL Anywhere support... yes
checking    SQL Anywhere install dir... /opt/sqlanywhere16
checking    SQL Anywhere version... 16
```

3. 実行プログラムをコンパイルします。

```
$ make
```

4. コンポーネントをインストールします。

```
$ make install
```

結果

PHP は CGI 実行プログラムとして正常にコンパイルされます。

参照

- 「Apache モジュールとしての PHP のコンパイル」678 ページ
- 「PHP テストページの作成と実行」668 ページ

SQL Anywhere PHP API リファレンス

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_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」

sasql_affected_rows

プロトタイプ

int **sasql_affected_rows(** sasql_conn *$conn* **)**

説明

最後の SQL 文の影響を受けるローの数を返します。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。

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。

参照

- 「sasql_num_rows」

sasql_commit

プロトタイプ

bool **sasql_commit(** sasql_conn *$conn* **)**

説明

SQL Anywhere サーバ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。auto_commit オプションが Off である場合にのみ有効です。

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。

参照

- 「sasql_rollback」
- 「sasql_set_option」
- 「sasql_pconnect」
- 「sasql_disconnect」

sasql_close

プロトタイプ

bool **sasql_close(** sasql_conn *$conn* **)**

説明

開いていたデータベース接続を閉じます。

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。

sasql_connect

プロトタイプ

sasql_conn **sasql_connect(** string *$con_str* **)**

説明

SQL Anywhere データベースへの接続を確立します。

パラメータ

$con_str　SQL Anywhere によって認識される接続文字列。

戻り値

成功の場合は正の SQL Anywhere 接続リソース、失敗の場合はエラーまたは 0。

参照

- 「sasql_pconnect」
- 「sasql_disconnect」
- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』
- 「データベース接続」『SQL Anywhere サーバ データベース管理』

sasql_data_seek

プロトタイプ

bool **sasql_data_seek(** sasql_result *$result*, int *row_num* **)**

説明

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。

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。

参照

- 「sasql_fetch_field」
- 「sasql_fetch_array」
- 「sasql_fetch_assoc」
- 「sasql_fetch_row」
- 「sasql_fetch_object」
- 「sasql_query」

sasql_disconnect

プロトタイプ

bool **sasql_disconnect(** sasql_conn *$conn* **)**

説明

sasql_connect または sasql_pconnect によって開かれている接続を閉じます。

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。

参照

- 「sasql_connect」
- 「sasql_pconnect」

sasql_error

プロトタイプ

string **sasql_error(** [sasql_conn *$conn*] **)**

説明

最後に実行された SQL Anywhere PHP 関数のエラーテキストを返します。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 SQL Anywhere 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。

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。エラーメッセージのリストについては、「SQL Anywhere のエラーメッセージ」『エラーメッセージ』を参照してください。

参照

- 「sasql_errorcode」
- 「sasql_sqlstate」
- 「sasql_set_option」
- 「sasql_stmt_errno」
- 「sasql_stmt_error」

sasql_errorcode

プロトタイプ

int **sasql_errorcode(** [sasql_conn *$conn*] **)**

説明

最後に実行された SQL Anywhere PHP 関数のエラーコードを返します。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。

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

SQL Anywhere のエラーコードを表す整数。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。エラーコードのリストについては、「SQL Anywhere のエラーメッセージ (SQLCODE 順)」『エラーメッセージ』を参照してください。

参照

- 「sasql_connect」
- 「sasql_pconnect」
- 「sasql_error」
- 「sasql_sqlstate」
- 「sasql_set_option」
- 「sasql_stmt_errno」
- 「sasql_stmt_error」

sasql_escape_string

プロトタイプ

string **sasql_escape_string(** sasql_conn *$conn*, string *$str* **)**

説明

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。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。

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。

参照

- 「sasql_real_escape_string」
- 「sasql_connect」

sasql_fetch_array

プロトタイプ

array **sasql_fetch_array(** sasql_result *$result* **[,** int *$result_type* **])**

説明

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。

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。

参照

- 「sasql_data_seek」
- 「sasql_fetch_assoc」
- 「sasql_fetch_field」
- 「sasql_fetch_row」
- 「sasql_fetch_object」

sasql_fetch_assoc

プロトタイプ

array **sasql_fetch_assoc(** sasql_result *$result* **)**

説明

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。

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。

参照

- 「sasql_data_seek」
- 「sasql_fetch_field」
- 「sasql_fetch_field」
- 「sasql_fetch_row」
- 「sasql_fetch_object」

sasql_fetch_field

プロトタイプ

object **sasql_fetch_field(** sasql_result *$result* [, int *$field_offset*] **)**

説明

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。

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の記憶サイズを返します。
- **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。「Embedded SQL のデータ型」485 ページを参照してください。
- **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。

参照

- 「sasql_data_seek」
- 「sasql_fetch_array」
- 「sasql_fetch_assoc」
- 「sasql_fetch_row」
- 「sasql_fetch_object」

sasql_fetch_object

プロトタイプ

object **sasql_fetch_object(** sasql_result *$result* **)**

説明

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。

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。

参照

- 「sasql_data_seek」
- 「sasql_fetch_field」
- 「sasql_fetch_array」
- 「sasql_fetch_assoc」
- 「sasql_fetch_row」

sasql_fetch_row

プロトタイプ

array **sasql_fetch_row(** sasql_result *$result* **)**

説明

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。

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。

参照

- 「sasql_data_seek」
- 「sasql_fetch_field」
- 「sasql_fetch_array」
- 「sasql_fetch_assoc」
- 「sasql_fetch_object」

sasql_field_count

プロトタイプ

int **sasql_field_count(** sasql_conn *$conn* **)**

説明

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。

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。

sasql_field_seek

プロトタイプ

bool **sasql_field_seek(** sasql_result *$result*, int *$field_offset* **)**

説明

フィールドカーソルを特定のオフセットに設定します。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。

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。

sasql_free_result

プロトタイプ

bool **sasql_free_result(** sasql_result *$result* **)**

説明

sasql_query 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。

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。

参照

- 「sasql_query」

sasql_get_client_info

プロトタイプ

string **sasql_get_client_info( )**

説明

クライアントのバージョン情報を返します。

パラメータ

なし

戻り値

SQL Anywhere 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。

sasql_insert_id

プロトタイプ

int **sasql_insert_id(** sasql_conn *$conn* **)**

説明

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_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。

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。

sasql_message

プロトタイプ

bool **sasql_message(** sasql_conn *$conn*, string *$message* **)**

説明

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。

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。

sasql_multi_query

プロトタイプ

bool **sasql_multi_query(** sasql_conn *$conn*, string *$sql_str* **)**

説明

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。

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。

SQL 文の詳細については、「SQL 文」『SQL Anywhere サーバ SQL リファレンス』を参照してください。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

参照

- 「sasql_store_result」
- 「sasql_use_result」
- 「sasql_field_count」

sasql_next_result

プロトタイプ

bool **sasql_next_result(** sasql_conn *$conn* **)**

説明

$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。

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。

参照

- 「sasql_use_result」
- 「sasql_store_result」

sasql_num_fields

プロトタイプ

int **sasql_num_fields(** sasql_result *$result* **)**

説明

$result 内のローに含まれるフィールドの数を返します。

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。

参照

- 「sasql_num_rows」
- 「sasql_query」

sasql_num_rows

プロトタイプ

int **sasql_num_rows(** sasql_result *$result* **)**

説明

$result に含まれるローの数を返します。

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。「sasql_set_option」700 ページを参照してください。

参照

- 「sasql_num_fields」
- 「sasql_query」

sasql_pconnect

プロトタイプ

sasql_conn **sasql_pconnect(** string *$con_str* **)**

説明

SQL Anywhere データベースへの永続的な接続を確立します。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。

パラメータ

$con_str　SQL Anywhere によって認識される接続文字列。

戻り値

成功の場合は正の SQL Anywhere 永続接続リソース、失敗の場合はエラーまたは 0。

参照

- 「sasql_connect」
- 「sasql_disconnect」
- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』
- 「データベース接続」『SQL Anywhere サーバ データベース管理』

sasql_prepare

プロトタイプ

sasql_stmt **sasql_prepare(** sasql_conn *$conn*, string *$sql_str* **)**

説明

指定された SQL 文字列を準備します。

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。

SQL 文の詳細については、「SQL 文」『SQL Anywhere サーバ SQL リファレンス』を参照してください。

戻り値

文のオブジェクト。失敗した場合は FALSE。

参照

- 「sasql_stmt_param_count」
- 「sasql_stmt_bind_param」
- 「sasql_stmt_bind_param_ex」
- 「sasql_prepare」
- 「sasql_stmt_execute」
- 「sasql_connect」
- 「sasql_pconnect」

sasql_query

プロトタイプ

mixed **sasql_query(** sasql_conn *$conn*, string *$sql_str* [, int *$result_mode*] **)**

説明

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* を準備して実行します。

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。

パラメータ

$conn　接続関数から返される接続リソース。

$sql_str　SQL Anywhere 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。

SQL 文の詳細については、「SQL 文」『SQL Anywhere サーバ SQL リファレンス』を参照してください。

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。

参照

- 「sasql_real_query」
- 「sasql_free_result」
- 「sasql_fetch_array」
- 「sasql_fetch_field」
- 「sasql_fetch_object」
- 「sasql_fetch_row」

sasql_real_escape_string

プロトタイプ

string **sasql_real_escape_string(** sasql_conn *$conn*, string *$str* **)**

説明

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。エスケープされる特殊文字は、¥r、¥n、'、"、;、¥、および NULL 文字です。

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。

参照

- 「sasql_escape_string」
- 「sasql_connect」

sasql_real_query

プロトタイプ

bool **sasql_real_query(** sasql_conn *$conn*, string *$sql_str* **)**

説明

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。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。

パラメータ

$conn　接続関数から返される接続リソース。

$sql_str　SQL Anywhere 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。

参照

- 「sasql_query」
- 「sasql_store_result」
- 「sasql_use_result」
- 「sasql_field_count」

sasql_result_all

プロトタイプ

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*
]]]] **)**

説明

$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。

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"** のように指定します。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"** のように指定します。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')'"** のように指定します。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')'"** のように指定します。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。

参照

- 「sasql_query」

sasql_rollback

プロトタイプ

bool **sasql_rollback(** sasql_conn *$conn* **)**

説明

SQL Anywhere サーバ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。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。

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。

参照

- 「sasql_commit」
- 「sasql_set_option」

sasql_set_option

プロトタイプ

bool **sasql_set_option(** sasql_conn *$conn*, string *$option*, mixed *$value* **)**

説明

指定した接続で、指定したオプションの値を設定します。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
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
```

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。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

参照

- 「sasql_commit」
- 「sasql_error」
- 「sasql_errorcode」
- 「sasql_num_rows」
- 「sasql_rollback」

sasql_stmt_affected_rows

プロトタイプ

int **sasql_stmt_affected_rows(** sasql_stmt *$stmt* **)**

説明

文の実行の影響を受けるローの数を返します。

パラメータ

$stmt　sasql_stmt_execute によって実行された文リソース。

戻り値

影響を受けたローの数。失敗した場合は FALSE。

参照

- 「sasql_stmt_execute」

sasql_stmt_bind_param

プロトタイプ

bool **sasql_stmt_bind_param(** sasql_stmt *$stmt*, string *$types*, mixed **&***$var_1* [, mixed **&***$var_2* ..] **)**

説明

PHP 変数を文のパラメータにバインドします。

パラメータ

$stmt　sasql_prepare 関数によって返された準備文リソース。

$types　対応するバインドの種類を指定する 1 つ以上の文字を含む文字列。次のいずれかを指定してください。文字列には **s**、整数には **i**、二重には **d**、blob には **b** です。$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。

参照

- 「sasql_prepare」
- 「sasql_stmt_param_count」
- 「sasql_stmt_bind_param_ex」
- 「sasql_stmt_execute」

sasql_stmt_bind_param_ex

プロトタイプ

bool **sasql_stmt_bind_param_ex(** sasql_stmt *$stmt*, int *$param_number*, mixed **&***$var*, string *$type* [, bool *$is_null* [, int *$direction*]] **)**

説明

PHP 変数を文のパラメータにバインドします。

パラメータ

$stmt　sasql_prepare 関数によって返された準備文リソース。

$param_number　パラメータ番号。これは 0 ～ (sasql_stmt_param_count($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。文字列には **s**、整数には **i**、二重には **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。

参照

- 「sasql_prepare」
- 「sasql_stmt_param_count」
- 「sasql_stmt_bind_param」
- 「sasql_stmt_execute」

sasql_stmt_bind_result

プロトタイプ

bool **sasql_stmt_bind_result(** sasql_stmt *$stmt*, mixed **&***$var1* [, mixed **&***$var2* ..] **)**

説明

実行された文の結果カラムに 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。

パラメータ

$stmt sasql_stmt_execute によって実行された文リソース。

$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。

参照

- 「sasql_stmt_execute」
- 「sasql_stmt_fetch」

sasql_stmt_close

プロトタイプ

bool **sasql_stmt_close(** sasql_stmt *$stmt* **)**

説明

指定された文リソースを閉じて、関連付けられているリソースを解放します。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。

パラメータ

$stmt sasql_prepare 関数によって返された準備文リソース。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

参照

- 「sasql_stmt_result_metadata」
- 「sasql_prepare」

sasql_stmt_data_seek

プロトタイプ

bool **sasql_stmt_data_seek(** sasql_stmt *$stmt*, int *$offset* **)**

説明

この関数は、結果セット内で指定されたオフセットを検索します。

パラメータ

$stmt　文リソース。

$offset　結果セット内のオフセット。これは、0 ～ (sasql_stmt_num_rows($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。

参照

- 「sasql_stmt_num_rows」

sasql_stmt_errno

プロトタイプ

int **sasql_stmt_errno(** sasql_stmt *$stmt* **)**

説明

指定された文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。

パラメータ

$stmt　sasql_prepare 関数によって返された準備文リソース。

戻り値

整数のエラーコード。エラーコードのリストについては、「SQL Anywhere のエラーメッセージ (SQLCODE 順)」『エラーメッセージ』を参照してください。

参照

- 「sasql_stmt_error」
- 「sasql_error」
- 「sasql_errorcode」
- 「sasql_prepare」
- 「sasql_stmt_result_metadata」

sasql_stmt_error

プロトタイプ

string **sasql_stmt_error(** sasql_stmt *$stmt* **)**

説明

指定された文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。

パラメータ

$stmt　sasql_prepare 関数によって返された準備文リソース。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

戻り値

エラーが記述された文字列。エラーメッセージのリストについては、「SQL Anywhere のエラーメッセージ」『エラーメッセージ』を参照してください。

参照

- 「sasql_stmt_errno」
- 「sasql_error」
- 「sasql_errorcode」
- 「sasql_prepare」
- 「sasql_stmt_result_metadata」

sasql_stmt_execute

プロトタイプ

bool **sasql_stmt_execute(** sasql_stmt *$stmt* **)**

説明

準備文を実行します。sasql_stmt_result_metadata を使用して、文が結果セットを返すかどうかを確認できます。

パラメータ

$stmt sasql_prepare 関数によって返された準備文リソース。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。

参照

- 「sasql_prepare」
- 「sasql_stmt_param_count」
- 「sasql_stmt_bind_param」
- 「sasql_stmt_bind_param_ex」
- 「sasql_stmt_result_metadata」
- 「sasql_stmt_bind_result」

sasql_stmt_fetch

プロトタイプ

bool **sasql_stmt_fetch(** sasql_stmt *$stmt* **)**

説明

この関数は、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。

パラメータ

$stmt　文リソース。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

参照

- 「sasql_stmt_bind_result」

sasql_stmt_field_count

プロトタイプ

int **sasql_stmt_field_count(** sasql_stmt *$stmt* **)**

説明

この関数は、文の結果セット内のカラム数を返します。

パラメータ

$stmt　文リソース。

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。

参照

- 「sasql_stmt_result_metadata」

sasql_stmt_free_result

プロトタイプ

bool **sasql_stmt_free_result(** sasql_stmt *$stmt* **)**

説明

この関数は、キャッシュされた文の結果セットを解放します。

パラメータ

$stmt　sasql_stmt_execute を使用して実行された文リソース。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

参照

- 「sasql_stmt_execute」
- 「sasql_stmt_store_result」

sasql_stmt_insert_id

プロトタイプ

int **sasql_stmt_insert_id(** sasql_stmt *$stmt* **)**

説明

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。

パラメータ

$stmt sasql_stmt_execute によって実行された文リソース。

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。

参照

- 「sasql_stmt_execute」

sasql_stmt_next_result

プロトタイプ

bool **sasql_stmt_next_result(** sasql_stmt *$stmt* **)**

説明

この関数は、文の次の結果に進みます。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。

パラメータ

$stmt 文リソース。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

参照

- 「sasql_stmt_result_metadata」

sasql_stmt_num_rows

プロトタイプ

int **sasql_stmt_num_rows(** sasql_stmt *$stmt* **)**

説明

結果セット内のロー数を返します。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。

パラメータ

$stmt sasql_stmt_execute によって実行され、sasql_stmt_store_result が呼び出された文リソース。

戻り値

結果で使用できるロー数。失敗した場合は 0。

参照

- 「sasql_stmt_execute」
- 「sasql_stmt_store_result」

sasql_stmt_param_count

プロトタイプ

int **sasql_stmt_param_count(** sasql_stmt *$stmt* **)**

説明

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。

パラメータ

$stmt sasql_prepare 関数によって返される文リソース。

戻り値

パラメータ数。エラーの場合は FALSE。

参照

- 「sasql_prepare」
- 「sasql_stmt_bind_param」
- 「sasql_stmt_bind_param_ex」

sasql_stmt_reset

プロトタイプ

bool **sasql_stmt_reset(** sasql_stmt *$stmt* **)**

説明

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。

パラメータ

$stmt　文リソース。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

参照

- 「sasql_stmt_send_long_data」

sasql_stmt_result_metadata

プロトタイプ

sasql_result **sasql_stmt_result_metadata(** sasql_stmt *$stmt* **)**

説明

指定された文の結果セットオブジェクトを返します。

パラメータ

$stmt　準備され、実行された文リソース。

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。

sasql_stmt_send_long_data

プロトタイプ

bool **sasql_stmt_send_long_data(** sasql_stmt *$stmt*, int *$param_number*, string *$data* **)**

説明

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。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。

パラメータ

$stmt　sasql_prepare を使用して準備された文リソース。

$param_number　パラメータ番号。これは 0 ～ (sasql_stmt_param_count($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。

参照

- 「sasql_stmt_bind_param」
- 「sasql_stmt_bind_param_ex」
- 「sasql_prepare」
- 「sasql_stmt_param_count」

sasql_stmt_store_result

プロトタイプ

bool **sasql_stmt_store_result(** sasql_stmt *$stmt* **)**

説明

この関数を使用すると、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。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。

パラメータ

$stmt　sasql_stmt_execute を使用して実行された文リソース。

戻り値

成功した場合は TRUE、失敗した場合は FALSE。

参照

- 「sasql_stmt_free_result」
- 「sasql_stmt_execute」

sasql_store_result

プロトタイプ

sasql_result **sasql_store_result(** sasql_conn *$conn* **)**

説明

データベース接続 *$conn* での最後のクエリの結果セットを、sasql_data_seek 関数で使用できるように転送します。

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。

参照

- 「sasql_data_seek」
- 「sasql_stmt_execute」

sasql_sqlstate

プロトタイプ

string **sasql_sqlstate(** sasql_conn *$conn* **)**

説明

最新の SQLSTATE 文字列を返します。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。

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。SQLSTATE コードのリストについては、「SQL Anywhere のエラーメッセージ (SQLSTATE 順)」『エラーメッセージ』を参照してください。

参照

- 「sasql_error」
- 「sasql_errorcode」

sasql_use_result

プロトタイプ

sasql_result **sasql_use_result(** sasql_conn *$conn* **)**

説明

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。

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。

参照

- 「sasql_data_seek」
- 「sasql_stmt_execute」

Ruby サポート

SQL Anywhere の Ruby API サポート

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 SQL Anywhere 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。SQL Anywhere は、DBI とともに使用できる Ruby データベースドライバ (DBD) を提供しています。

SQL Anywhere for Ruby プロジェクトには 3 つの別個のパッケージがあります。すべて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。RubyGems を入手するには、http://rubyforge.org/projects/rubygems/にアクセスしてください。

SQL Anywhere Ruby プロジェクトのホームは、http://sqlanywhere.rubyforge.org/です。

SQL Anywhere のネイティブ 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
```

SQL Anywhere のこれ以外の Ruby パッケージを使用するには、このパッケージが必要です。

Rails

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。http://www.rubyonrails.org/を参照してください。

SQL Anywhere の ActiveRecord アダプタ

activerecord-sqlanywhere-adapter　このパッケージは、ActiveRecord と SQL Anywhere 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 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
```

SQL Anywhere の 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 SQL Anywhere 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
```

これらのパッケージに関するフィードバックがある場合は、sqlanywhere-users@rubyforge.com のメーリングリストを使用してください。

参照

- Ruby on Rails： http://www.rubyonrails.org/
- 「SQL Anywhere Ruby API リファレンス」
- 「SQL Anywhere 用 Ruby-DBI ドライバ」
- Ruby プログラミング言語： http://www.ruby-lang.org
- RubyForge/Ruby Central： http://rubyforge.org/
- Ruby/DBI - Ruby のダイレクトデータベースアクセスレイヤ： http://ruby-dbi.rubyforge.org/
- 「SQL Anywhere での Rails サポートの設定」
- ソースのダウンロード：http://rubyforge.org/projects/sqlanywhere
- RDocs：http://sqlanywhere.rubyforge.org/sqlanywhere/
- RDocs： http://sqlanywhere.rubyforge.org/dbd-sqlanywhere
- RDocs： http://sqlanywhere.rubyforge.org/activerecord-sqlanywhere-adapter

SQL Anywhere での Rails サポートの設定

SQL Anywhere に Ruby on Rails サポートを設定できます。

前提条件

この作業を実行するための前提条件はありません。

♦ タスク

1. 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。インストールする適切な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。http://www.rubyonrails.org/を参照してください。

2. システムに Ruby のインタプリターをインストールします。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。http://www.rubyonrails.org/ を参照してください。

3. 次のコマンドを実行し、Ruby Rails とその依存性をインストールします。

   ```
   gem install rails
   ```

4. Ruby Development Kit (**DevKit**) をインストールします。**DevKit** を http://rubyinstaller.org/downloads/ からダウンロードして、http://github.com/oneclick/rubyinstaller/wiki/Development-Kit の手順に従ってください。

5. 次のコマンドを実行し、SQL Anywhere 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 3.1.3 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* ディレクトリに Ruby をインストールし、Rails のバージョン 3.1.3 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-3.1.3¥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: sql

# 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: sql

production:
  adapter: sqlanywhere
  server: <%= app_name %>
  database: <%= app_name %>_production
  username: DBA
  password: sql
```

   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)。SQL Anywhere 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** (オプション、デフォルトは 'DBA')。このオプションは、接続文字列の "UserID" に対応しています。
- **password** (オプション、デフォルトは 'sql')。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-3.1.3¥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 )
```

7. 次の SQL Anywhere 固有の注意事項に従って、Ruby on Rails の Web サイト (http://guides.rails.info/getting_started.html) にあるチュートリアルに従います。

- 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
```

- **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
```

- *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'
```

- *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しているように **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,sql blog_development
dbinit -dba DBA,sql blog_test
dbinit -dba DBA,sql blog_production
cd ..
```

結果

SQL Anywhere で Rails サポートが設定されました。

次の手順

次のように、データベースサーバと 3 つのデータベースを起動します。

```
dbsrv16 -n blog blog_development.db blog_production.db blog_test.db
```

コマンドラインのデータベースサーバ名 (*blog*) は、*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。

参照

- http://www.rubyonrails.org/

SQL Anywhere 用 Ruby-DBI ドライバ

この項では、SQL Anywhere DBI ドライバを使用する Ruby アプリケーションを作成する方法の概要を説明します。DBI モジュールの完全なドキュメントについては、オンラインで http://ruby-dbi.rubyforge.org/を参照してください。

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 SQL Anywhere 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。この文字列が空白でない場合、接続文字列に ";UID=value" が付加されます。

- ***password*** ユーザ ID に対応するパスワードです。この文字列が空白でない場合、接続文字列に ";PWD=value" が付加されます。

- ***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。

```
dbsrv16 -n myserver "%SQLANYSAMP16%¥demo.db"
```

次のコードサンプルは、SQL Anywhere サンプルデータベースへの接続を開いて閉じます。次の例では文字列 "myserver" がサーバ名です。

```
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、追加接続パラメータをキーと値のペアのハッシュとして 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、SQL Anywhere 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
```

参照

- 「max_cursor_count オプション」『SQL Anywhere サーバ データベース管理』
- 「max_statement_count オプション」『SQL Anywhere サーバ データベース管理』

SQL Anywhere Ruby API リファレンス

SQL Anywhere には、SQL Anywhere C API への低レベルのインタフェースがあります。以降の各項で説明する 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。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、SQL Anywhere 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()
api.sqlany_connect( conn, "DSN=SQL Anywhere 16 Demo" )
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
```

以降の各項では、サポートされている各関数について説明します。

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。

参照

- 「sqlany_execute 関数」726 ページ

例

```
affected = api.sqlany_affected( stmt )
```

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。

参照

- 「sqlany_describe_bind_param 関数」725 ページ

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 )
```

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。

参照

- 「sqlany_new_connection 関数」735 ページ

例

```
api.sqlany_clear_error( conn )
```

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()
```

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。

参照

- 「sqlany_rollback 関数」738 ページ

例

```
rc = api.sqlany_commit( conn )
```

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、SQL Anywhere 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** SQL Anywhere 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。

参照

- 「sqlany_new_connection 関数」735 ページ
- 「sqlany_error 関数」726 ページ
- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』
- 「データベース接続」『SQL Anywhere サーバ データベース管理』

例

```
# Create a connection
conn = api.sqlany_new_connection()
```

```
# Establish a connection
status = api.sqlany_connect( conn, "UID=DBA;PWD=sql" )
print "Connection status = #{status}¥n"
```

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。

参照

- 「sqlany_bind_param 関数」722 ページ
- 「sqlany_prepare 関数」737 ページ

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 )
```

sqlany_disconnect 関数

SQL Anywhere 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。

参照

- 「sqlany_connect 関数」724 ページ
- 「sqlany_new_connection 関数」735 ページ

例

```
# Disconnect from the database
status = api.sqlany_disconnect( conn )
print "Disconnect status = #{status}¥n"
```

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。

参照

- 「sqlany_connect 関数」724 ページ
- 「SQL Anywhere のエラーメッセージ (SQLCODE 順)」『エラーメッセージ』

例

```
code, msg = api.sqlany_error( conn )
print "Code=#{code} Message=#{msg}¥n"
```

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。

参照

- 「sqlany_prepare 関数」737 ページ

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 )
```

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。

参照

- 「sqlany_fetch_absolute 関数」728 ページ
- 「sqlany_fetch_next 関数」729 ページ
- 「sqlany_num_cols 関数」735 ページ
- 「sqlany_get_column 関数」732 ページ

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"
```

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。

参照

- 「sqlany_error 関数」726 ページ

例

```
rc = api.sqlany_execute_immediate(conn, "UPDATE Contacts
    SET Contacts.ID = Contacts.ID + 1000
    WHERE Contacts.ID >= 50" )
```

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。

参照

- 「sqlany_error 関数」726 ページ
- 「sqlany_execute 関数」726 ページ
- 「sqlany_execute_direct 関数」727 ページ
- 「sqlany_fetch_next 関数」729 ページ

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"
```

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。

参照

- 「sqlany_error 関数」726 ページ
- 「sqlany_execute 関数」726 ページ
- 「sqlany_execute_direct 関数」727 ページ
- 「sqlany_fetch_absolute 関数」728 ページ

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"
```

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。

参照

- 「sqlany_init 関数」734 ページ

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 )
```

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。

参照

- 「sqlany_new_connection 関数」735 ページ

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 )
```

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。

参照

- 「sqlany_prepare 関数」 737 ページ
- 「sqlany_execute_direct 関数」 727 ページ

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 )
```

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。

参照

- 「sqlany_bind_param 関数」722 ページ
- 「sqlany_describe_bind_param 関数」725 ページ
- 「sqlany_prepare 関数」737 ページ

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"
```

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。

参照

- 「sqlany_execute 関数」726 ページ
- 「sqlany_execute_direct 関数」727 ページ
- 「sqlany_fetch_absolute 関数」728 ページ
- 「sqlany_fetch_next 関数」729 ページ

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"
```

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	型	説明
0	整数	成功した場合は 1、失敗した場合は 0。
1	整数	カラムのインデックス (0 ～ sqlany_num_cols() - 1)。
2	文字列	カラム名。
3	整数	カラム型。「カラムの型」739 ページを参照してください。
4	整数	カラムのネイティブ型。「ネイティブのカラム型」739 ページを参照してください。
5	整数	カラム精度 (数値型の場合)。
6	整数	カラムの位取り (数値型の場合)。
7	整数	カラムサイズ。
8	整数	カラムが NULL 入力可かどうか (1 = NULL 入力可、0 = NULL 入力不可)。

参照

- 「sqlany_execute 関数」726 ページ
- 「sqlany_execute_direct 関数」727 ページ
- 「sqlany_prepare 関数」737 ページ

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 )
```

sqlany_get_next_result 関数

複数の結果セットクエリのうち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。

参照

- 「sqlany_execute 関数」726 ページ
- 「sqlany_execute_direct 関数」727 ページ

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 )
```

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。

参照

- 「sqlany_fini 関数」730 ページ

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()
```

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。

参照

- 「sqlany_connect 関数」724 ページ
- 「sqlany_disconnect 関数」725 ページ

例

```
# Create a connection
conn = api.sqlany_new_connection()

# Establish a connection
status = api.sqlany_connect( conn, "UID=DBA;PWD=sql" )
print "Status=#{status}¥n"
```

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。

参照

- 「sqlany_execute 関数」726 ページ
- 「sqlany_execute_direct 関数」727 ページ
- 「sqlany_prepare 関数」737 ページ

例

```
stmt = api.sqlany_execute_direct( conn, "SELECT * FROM Employees" )
# Get number of result set columns
num_cols = api.sqlany_num_cols( stmt )
```

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。

参照

- 「sqlany_prepare 関数」737 ページ

例

```
stmt = api.sqlany_prepare(conn, "UPDATE Contacts
    SET Contacts.ID = Contacts.ID + 1000
    WHERE Contacts.ID >= ?" )
num_params = api.sqlany_num_params( stmt )
```

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。詳細については、「row_counts オプション」『SQL Anywhere サーバ データベース管理』を参照してください。

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。

参照

- 「sqlany_execute 関数」726 ページ
- 「sqlany_execute_direct 関数」727 ページ

例

```
stmt = api.sqlany_execute_direct( conn, "SELECT * FROM Employees" )
# Get number of rows in result set
num_rows = api.sqlany_num_rows( stmt )
```

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。

参照

- 「sqlany_execute 関数」726 ページ
- 「sqlany_free_stmt 関数」731 ページ
- 「sqlany_num_params 関数」736 ページ
- 「sqlany_describe_bind_param 関数」725 ページ
- 「sqlany_bind_param 関数」722 ページ

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 )
```

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。

参照

- 「sqlany_commit 関数」724 ページ

例

```
rc = api.sqlany_rollback( conn )
```

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。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

戻り値

現在の SQLSTATE を表す 5 文字のスカラー値を返します。

参照

- 「sqlany_error 関数」726 ページ
- 「SQL Anywhere のエラーメッセージ (SQLSTATE 順)」『エラーメッセージ』

例

```
sql_state = api.sqlany_sqlstate( conn )
```

カラムの型

次の Ruby クラスで、一部の SQL Anywhere Ruby 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
```

ネイティブのカラム型

次の表に、一部の SQL Anywhere 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

ネイティブ型の値	ネイティブ型
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

Sybase Open Client のサポート

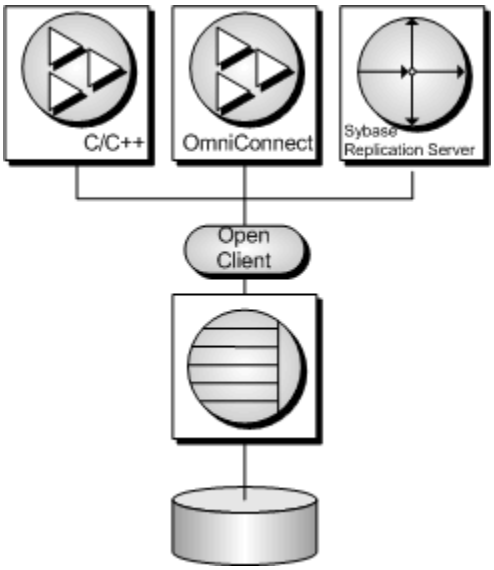
Sybase Open Client は、カスタマアプリケーション、サードパーティ製品、その他の Sybase 製品に、SQL Anywhere およびその他の Open Server と通信するために必要なインタフェースを提供します。

どのようなときに Open Client を使用するか

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 Sybase 製品を使用しているときに、Open Client インタフェースの使用が考えられます。

Open Client アプリケーション

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 SQL Anywhere に接続できます。その他の Sybase アプリケーションの中にも OmniConnect のように Open Client を使用しているものがあります。Open Client API は Sybase Adaptive Server Enterprise でもサポートされています。



参照

- 「Open Server としての SQL Anywhere」『SQL Anywhere サーバ データベース管理』

Open Client アーキテクチャ

> **注意**
> この項では、SQL Anywhere 用の Sybase Open Client プログラミングインタフェースについて説明します。Sybase Open Client アプリケーション開発の基本のマニュアルは、Sybase から入手できる Open Client マニュアルです。この項は、SQL Anywhere 特有の機能について説明していますが、Sybase Open Client アプリケーションプログラミングの包括的なガイドではありません。

Sybase Open Client には、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

Sybase Open Client はクライアントアプリケーションを記述する 2 つの主要なプログラミングインタフェースを提供します。それは 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 については、Sybase Open Client 製品に付属する『**Open Client DB-Library/C リファレンスマニュアル**』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が使用す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。

CS-Library と Bulk-Library はどちらも Sybase 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 Sybase 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 Sybase 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/Server 設定ガイド**』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、『**Open Client/Server プログラマーズガイド補足**』を参照してください。

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しているコンピュータに Sybase 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 Sybase 製品の一部として入手するか、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、Sybase 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。

参照

- 「Open Server としての SQL Anywhere」『SQL Anywhere サーバ データベース管理』

Open Client データ型マッピング

Sybase 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 アプリケーションと SQL Anywhere で使用されるそれぞれの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。

Open Client とは異なる名前を持つ SQL Anywhere のデータ型

次の表は、SQL Anywhere でサポートされるデータ型と Open Client のデータ型のマッピング リストです。これらは同じデータ型名ではないデータ型です。

SQL Anywhere データ型	Open Client データ型
unsigned short	int
unsigned int	bigint
unsigned bigint	numeric(20,0)
string	varchar
timestamp	datetime

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、取り得る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する SQL Anywhere データ型の方が取り得る値の範囲が大きくなっています。その結果、SQL Anywhere に値を渡してデータベースに格納できます。ただし、大きすぎて Open Client アプリケーションがフェッチできない値は除きます。

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 SQL Anywhere のカラムに設定できます。クライアントが SQL Anywhere 経由で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、SQL Anywhere に挿入した TIMESTAMP 値または SQL Anywhere 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。

Open Client アプリケーションでの SQL

この項では、SQL Anywhere 特有の問題に特に注目しながら、Open Client アプリケーションで SQL を使用する方法を簡潔に説明します。

詳細については、http://www.sybase.com/products/databasemanagement/openserver にある Open Client のマニュアルを参照してください。

参照

● 「SQL を使用したアプリケーション開発」1 ページ

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);
```

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、実行したいアクションを *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。

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、実行したいアクションを *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 がサポートする全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。

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。

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。

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 SQL Anywhere 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 SQL Anywhere に送信した場合、カラムのデータ型は返されません。

HTTP Web サービス

HTTP Web サーバとしての SQL Anywhere

SQL Anywhere には、SQL Anywhere データベースでのオンライン Web サービスの作成を可能にする組み込み HTTP Web サーバが含まれています。SQL Anywhere Web サーバでは、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 がサポートされています。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。

SQL Anywhere 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わりになる方法を提供します。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。

HTTP Web サーバでの Web サービスの提供に加え、インターネットや他の SQL Anywhere HTTP Web サーバで使用できる標準の Web サービスにアクセスするため、SQL Anywhere は SOAP または HTTP クライアントアプリケーションとして機能することができます。

参照

- 「Web クライアントを使用した Web サービスへのアクセス」786 ページ

HTTP Web サーバとして SQL Anywhere を使用するためのクイックスタート

この項では、SQL Anywhere HTTP Web サーバの起動方法、Web サービスの作成方法、Web ブラウザからのアクセス方法について説明します。HTTP を介した SOAP のサポートやアプリケーション開発など、SQL Anywhere Web サービスの機能については詳しく説明していません。このマニュアルでは説明しない数多くの SQL Anywhere Web サービスの機能を使用できます。

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. SQL Anywhere HTTP Web サーバを起動し、SQL Anywhere データベースをロードします。

   コマンドプロンプトで次のコマンドを実行します。

   ```
   dbsrv16 -xs http(port=8082) "%SQLANYSAMP16%¥demo.db"
   ```

   注意
   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、dbsrv16 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、dbeng16 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 要求を受信することをサーバに指示します。Web サーバがポート 8082 ですでに稼働中の場合は、異なるポート番号を使用します。

2. CREATE SERVICE 文を使用して、着信 Web ブラウザ要求に応答する Web サービスを作成します。

   a. 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=%SQLANYSAMP16%¥demo.db;uid=DBA;pwd=sql"
   ```

   b. 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、テーブルまたは関数からサービスを選択したり、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、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** から "Hello world" と表示されます。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。Web ブラウザでの結果セットの表示方法については、「Web サービスタイプ」754 ページを参照してください。

他のサンプルリソース

サンプルプログラムは *%SQLANYSAMP16%¥SQLAnywhere¥http* ディレクトリにあります。

その他の例は、http://www.sybase.com/developer/codexchange にある CodeXchange から入手できる場合があります。

参照

- 「HTTP Web サーバを起動する方法」751 ページ
- 「Web サービスタイプ」754 ページ
- 「CREATE SERVICE 文 [HTTP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』

HTTP Web サーバを起動する方法

-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、SQL Anywhere HTTP Web サーバが自動的に起動します。このオプションでは、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

コマンドの全般的なフォーマットを次に示します。

```
dbsrv16 -xs protocol-type(protocol-options) your-database-name.db
```

protocol-type および *protocol-options* 引数を、サポートされている次のいずれかのプロトコルおよび適切なプロトコル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。次に例を示します。

  ```
  dbsrv16 -xs HTTP(PORT=8082) services.db
  ```

- **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。SSL バージョン 3.0 と TLS バージョン 1.0/1.1 がサポートされています。次に例を示します。

  ```
  dbsrv16 -xs "HTTPS(FIPS=N;PORT=8082;IDENTITY=C:¥Users¥Public¥Documents¥SQL
  Anywhere 16¥Samples¥Certificates¥rsaserver.id;IDENTITY_PASSWORD=test)" services.db
  ```

注意
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。これらのオプションを使用すると、プロトコルの動作を制御でき、データベースサーバの起動時にコマンドラインで設定できます。

参照

- 「-xs データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「ネットワークプロトコルオプションの設定」751 ページ
- 「Web サービスとは?」754 ページ

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。「-xs データベースサーバオプション」『SQL Anywhere サーバ データベース管理』を参照してください。

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。

```
dbsrv16 -xs https(PORT=544;FIPS=YES;
   IDENTITY=certificate.id;IDENTITY_PASSWORD=password) 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。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 で受信する。
- FIPS 認定セキュリティアルゴリズムを有効にして、通信を暗号化する。
- 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 *certificate.id* を検索する。
- 指定された ID パスワード **password** に対してプライベートキーを検証する。

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) プロトコルオプション」『SQL Anywhere サーバ データベース管理』	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 プロトコルオプション」『SQL Anywhere サーバ データベース管理』	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 プロトコルオプション」『SQL Anywhere サーバ データベース管理』	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 プロトコルオプション」『SQL Anywhere サーバ データベース管理』	HTTPS	暗号化証明書のパスワードを指定します。

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
「LocalOnly (LO) プロトコルオプション」『SQL Anywhere サーバ データベース管理』	HTTP、HTTPS	クライアントがローカルコンピュータ上のサーバ (存在する場合) にのみ接続できるようにします。
「LogFile (LOG) プロトコルオプション」『SQL Anywhere サーバ データベース管理』	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) プロトコルオプション」『SQL Anywhere サーバ データベース管理』	HTTP、HTTPS	データベースサーバが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) プロトコルオプション」『SQL Anywhere サーバ データベース管理』	HTTP、HTTPS	データベースサーバがログに記録する Web 要求に関する情報メッセージのタイプを指定します。
「ServerPort (PORT) プロトコルオプション」『SQL Anywhere サーバ データベース管理』	HTTP、HTTPS	データベースサーバが受信しているポートを指定します。

参照

- 「ネットワークプロトコルオプション」『SQL Anywhere サーバ データベース管理』

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。

```
dbsrv16 -xs http(port=80;dbn=your-first-database),http(port=8800;dbn=your-second-database)
   your-first-database.db your-second-database.db
```

参照

- 「DatabaseName (DBN) プロトコルオプション」『SQL Anywhere サーバ データベース管理』
- 「-xs データベースサーバオプション」『SQL Anywhere サーバ データベース管理』

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、SQL Anywhere データベースで作成し、格納できます。

Web サービスタイプ

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 Web サービスに対して 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。SQL Anywhere では、結果セットのフォーマットとそれを返す方法を制御するいくつかの Web サービスタイプが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。

RAW Web サービスタイプと併用して動作するストアドプロシージャの例については、「Web ページをカスタマイズする方法」764 ページを参照してください。

- **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 には、次の 2 つの構造化型もあります。オブジェクトと配列です。JSON の詳細については、http://www.json.org/を参照してください。

  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、*%SQLANYSAMP16%¥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、SQL Anywhere の 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 (SQL Anywhere 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 (SQL Anywhere 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 );
```

参照

- 「Web サービスを作成または変更する方法」756 ページ
- 「Web サービスシステムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「HTTP 要求と SOAP 要求の構造」822 ページ
- 「CREATE SERVICE 文 [HTTP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「DISH サービスを作成する方法」758 ページ
- 「SOAP over HTTP サービスを作成する方法」757 ページ

Web サービスの管理

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。汎用 HTTP Web サービスの作成例については、「HTTP Web サーバとして SQL Anywhere を使用するためのクイックスタート」749 ページを参照してください。

- **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) 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。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。「CREATE SERVICE 文 [HTTP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』 を参照してください。

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。この項では、これらの文を Interactive SQL で実行して異なるタイプの Web サービスを作成する方法について説明します。この項の例では、次のコマンドを使用して、Interactive SQL 経由で SQL Anywhere データベース *your-sql-anywhere-database.db* に接続済みであることが前提となります。

```
dbisql -c "dbf=your-sql-anywhere-database.db;uid=your-userid;pwd=your-password"
```

HTTP Web サービスを作成する方法

HTTP Web サービスは、HTML、XML、または RAW 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 文と同じ構文を使用して作成、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 サービス用のストアド関数またはストアドプロシージャの作成の詳細については、「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」764 ページを参照してください。

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。

参照

- 「Web サービスタイプ」754 ページ
- 「CREATE SERVICE 文 [HTTP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER SERVICE 文 [HTTP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「SQL Anywhere HTTP Web サーバを参照する方法」782 ページ

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。SOAP 標準の詳細については、http://www.w3.org/TR/2000/NOTE-SOAP-20000508/を参照してください。

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。クライアントアプリケーションのデータ型のフォーマット方法については、「結果セットからの変数へのアクセス」806 ページを参照してください。

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。SOAP サービスでサポートされる他のフォーマットの例については、「HTTP Web サービスの例」825 ページを参照してください。

注意
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;
```

参照

- 「Web サービスタイプ」754 ページ
- 「CREATE SERVICE 文 [SOAP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER SERVICE 文 [SOAP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「DISH サービスを作成する方法」758 ページ

DISH サービスを作成する方法

SQL Anywhere では、SOAP サービスのグループに対して SOAP 終了ポイントとして動作する DISH サービスを作成できます。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 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、各 DISH 終了ポイントが異なる機能のサービスを 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。

参照

- 「Web サービスタイプ」754 ページ
- 「SQL Anywhere HTTP Web サーバを参照する方法」782 ページ
- 「HTTP Web サービスの例」825 ページ
- 「CREATE SERVICE 文 [SOAP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER SERVICE 文 [SOAP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「SOAP over HTTP サービスを作成する方法」757 ページ

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) 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;
```

参照

- 「DROP SERVICE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「root Web サービスを作成し、カスタマイズする方法」760 ページ

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.";
```

参照

- 「COMMENT 文」『SQL Anywhere サーバ SQL リファレンス』

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。詳細については、「sa_set_http_header システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』を参照してください。

参照

- 「Web ページをカスタマイズする方法」764 ページ
- 「SQL Anywhere HTTP Web サーバを参照する方法」782 ページ
- 「HTML_ENCODE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE SERVICE 文 [HTTP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』

Web サービスの SQL 文

次の SQL 文を使用して、Web サービスの開発を支援できます。

Web サーバ関連の SQL 文	説明
「CREATE SERVICE 文 [HTTP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』	新しい HTTP Web サービスを作成します。
「CREATE SERVICE 文 [SOAP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』	HTTP または DISH サービス経由の新しい SOAP を作成します。
「ALTER SERVICE 文 [HTTP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』	既存の HTTP Web サービスを変更します。

Web サーバ関連の SQL 文	説明
「ALTER SERVICE 文 [SOAP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』	HTTP または DISH サービス経由の既存の SOAP を変更します。
「COMMENT 文」『SQL Anywhere サーバ SQL リファレンス』	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 文」『SQL Anywhere サーバ SQL リファレンス』	Web サービスを削除します。

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設定の詳細については、「max_plans_cached オプション」『SQL Anywhere サーバ データベース管理』を参照してください。

接続プールでは、特定のサービスに対する HTTP 要求は、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。詳細については、「CREATE SERVICE 文 [HTTP Web サービス]」『SQL Anywhere

サーバ SQL リファレンス』 と 「CREATE SERVICE 文 [SOAP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』 を参照してください。

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。

参照

- http_connection_pool_basesize 接続プロパティ『SQL Anywhere サーバ データベース管理』
- http_connection_pool_timeout 接続プロパティ『SQL Anywhere サーバ データベース管理』
- 「Web サービス接続プロパティ」781 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

この項では、Web ページの作成とカスタマイズの概要について説明します。HTTP Web サーバ用のストアドプロシージャを開発する方法について説明します。SQL Anywhere HTTP Web サーバの起動方法と、ストアドプロシージャを呼び出す Web サービスの作成方法に関する知識があることが前提となります。

Web サービスアプリケーションの例の詳細については、*%SQLANYSAMP16%¥SQLAnywhere ¥HTTP* ディレクトリを参照してください。

参照

- 「HTTP Web サーバを起動する方法」751 ページ
- 「Web サービスとは?」754 ページ

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 サービスタイプでは、HTML や XML などの必要なマークアップを提供するために Web サービスのプロシージャと関数を明示的にコーディング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** タイプを使用する場合に Web ページをカスタマイズするには、次のタスクを実行してください。

- HTTP コンテンツタイプヘッダフィールドを、呼び出されるストアドプロシージャで、text/html などの適切な MIME タイプに設定する。

- 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する。

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。SQL Anywhere には、タグを作成するための関数がいくつか備えられています。

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 <)。

カスタム Web ページを生成する **RAW** Web サービスの詳しい例については、「root Web サービスを作成し、カスタマイズする方法」760 ページを参照してください。

参照

- 「Web サービスタイプ」754 ページ
- 「sa_set_http_header システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「Web サービス関数」780 ページ
- 「XMLCONCAT 関数 [文字列]」『SQL Anywhere サーバ SQL リファレンス』
- 「XMLELEMENT 関数 [文字列]」『SQL Anywhere サーバ SQL リファレンス』
- 「SQL Anywhere HTTP Web サーバを参照する方法」782 ページ
- 「関数」『SQL Anywhere サーバ SQL リファレンス』

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、次のいずれかの方法でアクセスできます。

- ストアド関数またはストアドプロシージャ呼び出しのホストパラメータとして変数とヘッダを渡す、Web サービス文の宣言。
- ストアド関数またはストアドプロシージャでの HTTP_VARIABLE、NEXT_HTTP_VARIABLE、HTTP_HEADER、NEXT_HTTP_HEADER 関数の呼び出し。

参照

- 「Web サービスへの変数の指定」803 ページ

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

関数またはプロシージャ呼び出しのホストパラメータとしてクライアント指定の変数を渡すと、クライアント指定の変数を参照できます。

例

次の例は、**ShowTable** という Web サービスで使用されるホスト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 tblname VARCHAR(128))
BEGIN
   -- write SQL code utilizing the username and tblname 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** パラメータにそれぞれマッピングされます。

ホストパラメータとして URL を渡す場合の詳細については、「SQL Anywhere HTTP Web サーバを参照する方法」782 ページを参照してください。

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 関数は、クライアントが指定した変数とヘッダに対して反復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 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 SERVICE ShowDetail
   TYPE 'HTML'
   URL PATH OFF
   AUTHORIZATION OFF
   USER DBA
   AS CALL ShowDetail();

CREATE 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;
```

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。

HTTP_HEADER と NEXT_HTTP_HEADER を使用したヘッダのアクセス

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、前のヘッダ名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。

次の表は、いくつかの共通の HTTP 要求ヘッダと典型的な値を示します。

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、特殊なヘッダと典型的な値を示します。

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 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 関数を使用してその値を取得します。

```
CREATE PROCEDURE HTTPHeaderExample()
RESULT ( html_string LONG VARCHAR )
BEGIN
   declare header_name LONG VARCHAR;
   declare header_value LONG VARCHAR;
   declare header_query LONG VARCHAR;
   declare table_rows XML;
   set header_name = NULL;
   set table_rows = NULL;
header_loop:
   LOOP
      SET header_name = NEXT_HTTP_HEADER( header_name );
      IF header_name IS NULL THEN
         LEAVE header_loop
      END IF;
      SET header_value = HTTP_HEADER( header_name );
      SET header_query = HTTP_HEADER( '@HttpQueryString' );
      -- Format header name and value 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", header_query ) );

   END LOOP;
   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" ),
                           'HTTP Query String' ),
            table_rows );
END;
```

Web ブラウザで **ShowHTTPHeaders** 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。

参照

- 「Web サービスへの変数の指定」803 ページ
- 「HTTP_VARIABLE 関数 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「NEXT_HTTP_VARIABLE 関数 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「HTTP_HEADER 関数 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「NEXT_HTTP_HEADER 関数 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』

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 関数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、int 型の **amount** と文字列型の **item** の 2 つのパラメータを取る **addItem** SOAP 操作を実行します。**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で SOAP 応答検証ヘッダを移植するために使用されます。応答は、それぞれ INT、LONG VARCHAR、LONG VARCHAR 型の **quantity**、**item**、**status** の 3 つのカラムの結果です。

```
// 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;
```

この例のクライアント側の実装については、「SOAP 要求ヘッダの管理」797 ページを参照してください。

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。SQL Anywhere には、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追加機能があります。

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 要求がキューに追加され、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、その 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、効率的に新しいセッションコンテキストに移動され、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、*%SQLANYSAMP16%¥SQLAnywhere¥HTTP ¥session.sql* を参照してください。

> **注意**
> 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。
>
> SQL Anywhere でのライセンスの詳細については、http://www.sybase.com/detail?id=1056242 を参照してください。

参照

- 「sa_set_http_option システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_set_http_header システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「CONNECTION_PROPERTY 関数 [システム]」『SQL Anywhere サーバ SQL リファレンス』
- 「接続プロパティの一覧」『SQL Anywhere サーバ データベース管理』

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。

セッションステータス管理は、URL と cookie によってサポートされています。HTTP セッションには、HTTP cookie または GET 要求の URL を使用してアクセスでき、POST (x-www-form-urlencoded) 要求の本文からもアクセスできます。たとえば、次の URL では、実行時に **XYZ** データベース接続が使用されます。

```
http://localhost/sa_svc?SESSIONID=XYZ
```

XYZ データベース接続が存在しない場合は、標準のセッションレス要求として処理されます。

例

次のコードは、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。

```
CREATE SERVICE mysession
   TYPE 'RAW'
   AUTHORIZATION OFF
   USER DBA
   AS CALL mysession_proc();

CREATE PROCEDURE mysession_proc()
BEGIN
   DECLARE body LONG VARCHAR;
   DECLARE hostname LONG VARCHAR;
   DECLARE svcname LONG VARCHAR;
   DECLARE sesid LONG VARCHAR;

   CALL sa_set_http_header ( 'Content-Type', 'text/html' );
```

```
    SELECT CONNECTION_PROPERTY('SessionID') INTO sesid;
    SELECT CONNECTION_PROPERTY('HttpServiceName') INTO svcname;
    SELECT HTTP_HEADER( 'Host' ) INTO hostname;
    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>';
       SELECT body;
    END IF;
    IF sesid = '' THEN
       SET sesid = set_session_url();
       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;

    SELECT body;
 END;
```

参照

- 「URL を使用してセッションを管理する方法」774 ページ

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 ブラウザにより URL 内でセッション ID が指定されます。

例

次の例は、セッション ID を URL でのみ指定できる場合に、HTTP Web サーバの SQL 関数内でユニークなセッション ID を作成する方法を示しています。

```
CREATE FUNCTION set_session_url()
RETURNS LONG VARCHAR
BEGIN
   DECLARE session_id LONG VARCHAR;
   DECLARE tm TIMESTAMP;
   SET tm = NOW(*);
   SET session_id = 'session_' ||
      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;
```

接続に対して session_id が定義されていないと、接続がセッションレスとなり、SessionID は空の文字列で示されます。

別の HTTP 要求が session_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ます。

参照

- 「sa_set_http_option システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_set_http_header システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「CONNECTION_PROPERTY 関数 [システム]」『SQL Anywhere サーバ SQL リファレンス』
- 「接続プロパティの一覧」『SQL Anywhere サーバ データベース管理』
- 「非アクティブな HTTP セッションを検出する方法」776 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」776 ページ
- 「Web サービスエラーコードリファレンス」823 ページ

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。

注意
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。URL で指定されたセッション ID は、セッション ID が URL と cookie の両方で指定された場合に使用されます。

例

次の例は、セッション ID を URL または cookie で指定できる場合に、HTTP Web サーバの SQL 関数内でユニークなセッション ID を作成する方法を示しています。

```
CREATE FUNCTION set_session_cookie()
RETURNS LONG VARCHAR
BEGIN
   DECLARE session_id LONG VARCHAR;
   DECLARE tm TIMESTAMP;
   SET tm = NOW(*);
   SET session_id = 'session_' ||
      CONVERT( VARCHAR, SECONDS(tm) * 1000 + DATEPART( MILLISECOND, tm ) );
   CALL sa_set_http_option( 'SessionID', session_id );
   CALL sa_set_http_header( 'Set-Cookie',
      'sessionid=' || session_id || ';' ||
      'max-age=60;' ||
      'path=/session;' );
   SELECT CONNECTION_PROPERTY( 'SessionID' ) INTO session_id;
   RETURN( session_id );
END;
```

参照

- 「sa_set_http_option システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_set_http_header システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「CONNECTION_PROPERTY 関数 [システム]」『SQL Anywhere サーバ SQL リファレンス』
- 「接続プロパティの一覧」『SQL Anywhere サーバ データベース管理』
- 「非アクティブな HTTP セッションを検出する方法」776 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」776 ページ

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。

注意
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;
```

参照

- 「CONNECTION_PROPERTY 関数 [システム]」『SQL Anywhere サーバ SQL リファレンス』
- 「接続プロパティの一覧」『SQL Anywhere サーバ データベース管理』
- 「HTTP セッションを作成する方法」773 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」776 ページ
- 「http_session_timeout オプション」『SQL Anywhere サーバ データベース管理』

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

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。

参照

- 「CONNECTION_PROPERTY 関数 [システム]」『SQL Anywhere サーバ SQL リファレンス』
- 「接続プロパティの一覧」『SQL Anywhere サーバ データベース管理』
- 「sa_set_http_option システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「非アクティブな HTTP セッションを検出する方法」776 ページ

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ので、このセッションコンテキストを必要とする後続の HTTP 要求は作成された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。

参照

- 「sa_set_http_option システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「HTTP セッションを作成する方法」773 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」776 ページ
- 「NEXT_CONNECTION 関数 [システム]」『SQL Anywhere サーバ SQL リファレンス』

HTTP セッションのエラーコード

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 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。

参照

- 「Web サービスエラーコードリファレンス」823 ページ

文字セットの変換に関する考慮事項

文字セット変換は、デフォルトによりテキストタイプの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。

参照

- 「sa_set_http_option システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

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する必要があります。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すすめします。Open Web Application Security Project (https://www.owasp.org) には、Web アプリケーションをセキュリティ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

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 サービスとともに使用します。

- 「sa_http_header_info システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_http_php_page システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_http_php_page_interpreted システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_http_variable_info システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_set_http_header システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_set_http_option システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_set_soap_header システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

参照

- 「Web サービス関数」『SQL Anywhere サーバ SQL リファレンス』
- 「HTTP Web サーバとしての SQL Anywhere」749 ページ
- 「-xs データベースサーバオプション」『SQL Anywhere サーバ データベース管理』

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。

次の関数を使用できます。

- 「HTML_DECODE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「HTML_ENCODE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「HTTP_BODY 関数 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「HTTP_DECODE 関数 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「HTTP_ENCODE 関数 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「HTTP_HEADER 関数 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「HTTP_RESPONSE_HEADER 関数 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「HTTP_VARIABLE 関数 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「NEXT_HTTP_HEADER 関数 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「NEXT_HTTP_RESPONSE_HEADER 関数 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「NEXT_HTTP_VARIABLE 関数 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「NEXT_SOAP_HEADER 関数 [SOAP]」『SQL Anywhere サーバ SQL リファレンス』
- 「SOAP_HEADER 関数 [SOAP]」『SQL Anywhere サーバ SQL リファレンス』

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。

参照

- 「HTTP Web サーバとしての SQL Anywhere」749 ページ
- 「-xs データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「Web サービスシステムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

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できるデータベースプロパティを指定できます。

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。

参照

- 「接続プロパティの一覧」『SQL Anywhere サーバ データベース管理』
- HttpServiceName 接続プロパティ『SQL Anywhere サーバ データベース管理』
- AuthType 接続プロパティ『SQL Anywhere サーバ データベース管理』
- ServerPort 接続プロパティ『SQL Anywhere サーバ データベース管理』
- ClientNodeAddress 接続プロパティ『SQL Anywhere サーバ データベース管理』
- ServerNodeAddress 接続プロパティ『SQL Anywhere サーバ データベース管理』
- BytesReceived 接続プロパティ『SQL Anywhere サーバ データベース管理』

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。

参照

- 「データベースオプション」『SQL Anywhere サーバ データベース管理』
- 「http_connection_pool_basesize オプション」『SQL Anywhere サーバ データベース管理』
- 「http_connection_pool_timeout オプション」『SQL Anywhere サーバ データベース管理』
- 「http_session_timeout オプション」『SQL Anywhere サーバ データベース管理』
- 「request_timeout オプション」『SQL Anywhere サーバ データベース管理』
- 「webservice_namespace_host オプション」『SQL Anywhere サーバ データベース管理』

SQL Anywhere HTTP Web サーバを参照する方法

Web サービスの命名と設計の方法によって、使用可能な URL 名が定義されます。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。

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。この項では、SQL Anywhere HTTP Web サーバで定義された Web サービスにアクセスできるようにするため、Web ブラウザの URL 構文をフォーマットする方法について説明します。

注意
この項で記載する情報は、RAW、XML、HTML、DISH サービスなどの汎用 HTTP Web サービスタイプを使用する HTTP Web サーバに適用されます。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。

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。

  「-xs データベースサーバオプション」『SQL Anywhere サーバ データベース管理』と「ServerPort (PORT) プロトコルオプション」『SQL Anywhere サーバ データベース管理』を参照してください。

- **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。

  「-xs データベースサーバオプション」『SQL Anywhere サーバ データベース管理』と「DatabaseName (DBN) プロトコルオプション」『SQL Anywhere サーバ データベース管理』を参照してください。

- **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。SQL Anywhere は、URL の残りの部分と定義されたサービスを一致させます。

  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、**TestProcedure** 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 ELEMENTS と URL ベースの変数の詳細については、「CREATE SERVICE 文 [HTTP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』 を参照してください。

- **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** からデモデータベースで **ShowSalesOrderDetail** が実行されていると仮定した場合、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

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、その後で *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

参照

- 「CREATE SERVICE 文 [HTTP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「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」766 ページ
- 「HTTP_VARIABLE 関数 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「root Web サービスを作成し、カスタマイズする方法」760 ページ

例

次の URL 構文は、**gallery_image** サービスが **URL ON** で定義されていると仮定した場合に、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。

```
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、*%SQLANYSAMP16%¥SQLAnywhere¥HTTP¥gallery.sql* を参照してください。

Web クライアントを使用した Web サービスへのアクセス

SQL Anywhere を Web クライアントとして使用して、SQL Anywhere 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。

SQL Anywhere を Web クライアントとして使用できるだけでなく、SQL Anywhere 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。

SQL Anywhere を Web クライアントとして使用するためのクイックスタート

この項では、SQL Anywhere 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 HTTP サーバに接続し、一般的な HTTP Web サービスにアクセスする方法について説明します。SQL Anywhere Web クライアントの機能全般については説明しません。このトピックで説明するもの以外にも、数多くの SQL Anywhere Web クライアントの機能を使用できます。

あらゆるタイプのオンライン Web サーバに接続する SQL Anywhere Web クライアントアプリケーションを開発できますが、この項では、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 **SampleHTMLService** という Web サービスに接続することを前提としています。

```
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;
```

SQL Anywhere 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。

1. SQL Anywhere クライアントデータベースが存在しない場合、次のコマンドを実行して作成します。

   ```
   dbinit -dba DBA,sql 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。

   ```
   dbsrv16 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=sql;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 2010 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
```

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** プロシージャで結果セットが作成されて 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。

Attribute	Value
Status	HTTP /1.1 200 OK
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>
Date	Wed, 09 Jun 2010 21:55:01 GMT
Connection	close
Expires	Wed, 09 Jun 2010 21:55:01 GMT
Content-Type	text/html; charset=windows-1252
Server	SQLAnywhere/16.0

参照

- 「HTTP Web サーバとして SQL Anywhere を使用するためのクイックスタート」749 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」764 ページ
- 「CREATE PROCEDURE 文 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE FUNCTION 文 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』

SQL Anywhere HTTP Web サーバへのアクセスのクイックスタート

この項では、Python と C# の 2 つの異なるタイプ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について説明します。SQL Anywhere Web サービスアプリケーションの機能全般については説明しません。このトピックで説明するもの以外にも、数多くの SQL Anywhere Web サービスの機能を使用できます。

あらゆるタイプのオンライン Web サーバに接続する SQL Anywhere Web クライアントアプリケーションを開発できますが、このマニュアルでは、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ています。

```
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。

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。

   ●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。

●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
```

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
```

参照

- 「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」764 ページ
- 「HTTP Web サーバとして SQL Anywhere を使用するためのクイックスタート」749 ページ

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 によって実行される 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制限

されています。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。

Web サービスアプリケーションの例の詳細については、*%SQLANYSAMP16%¥SQLAnywhere ¥HTTP* ディレクトリを参照してください。

参照

● 「Web クライアント SQL 文」 802 ページ

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。

次のリストに、Web クライアント関数とプロシージャの作成または変更の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)

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ファンクションが 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。

次の文は、ポート 8082 の **localhost** にある HTTP Web サーバによって実行される **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ホスト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。

これらの指定例を使用して HTTP サーバを起動する方法については、「HTTP Web サーバを起動する方法」751 ページを参照してください。

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ファンクションが SSL (セキュアソケットレイヤ) 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、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できる必要があります。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提供できます。両方を行うことはできません。

次の文は、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 16¥¥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 16¥Samples¥Certificates¥rsaroot.crt* ファイルにあることを示します。

注意
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。

これらの指定例を使用して HTTP Web サーバを起動する方法については、「HTTP Web サーバを起動する方法」751 ページを参照してください。

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、SQL Anywhere へそれを返します。

参照

- 「Web クライアント SQL 文」802 ページ

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/16.0.0.1403
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/16.0.0.1403
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場合は、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することをおすすめします。SOAP 関数では、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。

参照

- 「Web クライアント SQL 文」802 ページ
- 「OPENXML 演算子」『SQL Anywhere サーバ SQL リファレンス』
- 「Web サービスへの変数の指定」803 ページ
- 「HTTP 要求と SOAP 要求の構造」822 ページ

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ことをおすすめします。

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することをおすすめします。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。

参照

- 「Web クライアント SQL 文」 802 ページ
- 「ClientPort (CPORT) プロトコルオプション (クライアント側のみ)」『SQL Anywhere サーバ データベース管理』

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した URL に要求を送信する **SomeOperation2** 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 SomeOperation2()
   URL 'HTTP://localhost:8082/dbname/SampleWebService'
   TYPE 'HTTP:GET'
   HEADER 'SOAPAction¥nDate¥nFrom:¥nCustomAlias:John Doe';
```

この例では、SQL Anywhere によって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、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/16.0.0.1403
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/16.0.0.1403
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
```

注意
SOAP 関数またはプロシージャの作成時に WSDL に指定された 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。
Authorization	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。
Connection	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。
Host	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。

HTTP ヘッダ	説明
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。

参照

- 「Web クライアント SQL 文」802 ページ

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ファンクションによって、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したり、代入パラメータメカニズムを使用して動的に設定したりできる 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** と **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:ianywhere-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' );
```

この例のサーバ側の実装については、「クライアント提供の SOAP 要求ヘッダにアクセスする方法」770 ページを参照してください。

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、SQL Anywhere 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。

SOAP ネームスペース URI の要件

ネームスペース URI では、指定された 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に示されていますが、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 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 ドキュメントの最初で指定された **targetNamespace** 属性からコピーできます。後続の '/' は重要であるため、これら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。

参照

- 「Web サービスの要求タイプ」793 ページ
- 「Web クライアント SQL 文」802 ページ
- 「DISH サービスを作成する方法」758 ページ

Web クライアント SQL 文

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 文 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』	HTTP 要求を使用して HTTP 要求または SOAP 要求を行う Web クライアント関数を作成します。
「ALTER FUNCTION 文」『SQL Anywhere サーバ SQL リファレンス』	関数を変更します。
「CREATE PROCEDURE 文 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』	HTTP サーバへの HTTP 要求または SOAP 要求を作成するユーザ定義の Web クライアントプロシージャを作成します。
「ALTER PROCEDURE 文」『SQL Anywhere サーバ SQL リファレンス』	プロシージャを変更します。

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。

参照

- 「Web サービスタイプ」754 ページ

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。URL 句の指定方法、およびサーバによる URL パラメータの解釈の詳細については、「SQL Anywhere HTTP Web サーバを参照する方法」782 ページを参照してください。

参照

- 「CREATE SERVICE 文 [HTTP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「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」766 ページ
- 「root Web サービスを作成し、カスタマイズする方法」760 ページ

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。

参照

- 「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」766 ページ
- 「CREATE PROCEDURE 文 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE FUNCTION 文 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』

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>
```

前の方法の代わりに、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' );
```

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すすめします。

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 を必要としてい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

参照

- 「結果セットからの変数へのアクセス」806 ページ
- 「データベースにおける XML」『SQL Anywhere サーバ SQL の使用法』
- 「OPENXML 演算子」『SQL Anywhere サーバ SQL リファレンス』

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 2 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**、2 番目のカラムには **Value** という名前が付けられています。どちらも LONG VARCHAR 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。

次の例は、一般的な応答を示します。

Attribute	Value
Status	HTTP /1.0 200 OK
Body	<!DOCTYPE HTML ... ><HTML> ... </HTML>
Content-Type	text/html
Server	GWS/2.1
Content-Length	2234
Date	Mon, 18 Oct 2004, 16:00:00 GMT

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 SybaseWebPage()
URL 'http://www.sybase.com/mobilize'
TYPE 'HTTP';
```

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 SybaseWebPage()
   WITH (Attribute LONG VARCHAR, Value LONG VARCHAR);
```

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、テンポラリビューを定義するには WITH 句が必要です。

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
);
```

結果セットは、次のように **StoredResults** テーブルに挿入します。

```
INSERT INTO StoredResults
   SELECT * FROM SybaseWebPage()
   WITH (Attribute LONG VARCHAR, Value LONG VARCHAR);
```

SELECT 文の通常の構文に従い、句を追加できます。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 SybaseWebPage()
   WITH (Attribute LONG VARCHAR, Value LONG VARCHAR)
   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。

参照

- 「Web クライアント SQL 文」802 ページ
- 「結果セットからの変数へのアクセス」806 ページ

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 SQL Anywhere 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 SQL Anywhere 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

SQL Anywhere の型	XML スキーマの型	XML の例
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」 と解釈される)
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
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

SQL Anywhere の型	XML スキーマの型	XML の例
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

SOAP の構造化されたデータ型

XML 戻り値

Web サービスクライアントとしての SQL Anywhere 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。

単純な戻り値のデータ型には、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

SQL Anywhere 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。しかし、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ことが必要です。次に示すプレフィクスは、SQL Anywhere 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ます。

SQL Anywhere の 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

複雑なデータ型の例

配列、構造体、構造体の配列をそれぞれ表すパラメータを取る Web サービスクライアント関数を作成する方法を次の 3 つの例で示します。Web サービス関数は、それぞれ echoFloatArray、echoStruct、echoStructArray という SOAP 操作 (または RPC 関数名) と通信します。相互運用性テストで共通で使用されるネームスペースは http://soapinterop.org/ で、URL 句を目的の SOAP 終了ポイントに変更するだけで、関数を代替相互運用サーバに対してテストすることができます。

これらの例は、http://mssoapinterop.org/stkV3 の Microsoft SOAP ToolKit 3.0 Round 2 相互運用性テストサーバに対して要求を発行するように設計されています。

これらの例では、テーブルを使用して XML データを生成します。このテーブルを設定する方法は次のとおりです。

```
CREATE LOCAL TEMPORARY TABLE SoapData
(
   seqno INT DEFAULT AUTOINCREMENT,
   i INT,
   f FLOAT,
   s LONG VARCHAR
) ON COMMIT PRESERVE ROWS;
```

```
INSERT INTO SoapData (i,f,s)
VALUES (99,99.999,'Ninety-Nine');

INSERT INTO SoapData (i,f,s)
VALUES (199,199.999,'Hundred and Ninety-Nine');
```

次の 3 つの関数は、SOAP 要求を相互運用サーバに送信します。このサンプルでは、Microsoft の Interop サーバに対して要求を発行します。

```
CREATE FUNCTION echoFloatArray(inputFloatArray XML)
RETURNS XML
URL 'http://mssoapinterop.org/stkV3/Interop.wsdl'
HEADER 'SOAPAction:"http://soapinterop.org/"'
NAMESPACE 'http://soapinterop.org/';

CREATE FUNCTION echoStruct(inputStruct XML)
RETURNS XML
URL 'http://mssoapinterop.org/stkV3/Interop.wsdl'
HEADER 'SOAPAction:"http://soapinterop.org/"'
NAMESPACE 'http://soapinterop.org/';

CREATE FUNCTION echoStructArray(inputStructArray XML)
RETURNS XML
URL 'http://mssoapinterop.org/stkV3/Interop.wsdl'
HEADER 'SOAPAction:"http://soapinterop.org/"'
NAMESPACE 'http://soapinterop.org/';
```

最後に、例の文を 3 つ示します。それぞれのパラメータは XML 表現で表されています。

1. 次の例のパラメータは、配列を表します。

```
SELECT echoFloatArray(
   XMLELEMENT( 'inputFloatArray',
     XMLATTRIBUTES( 'xsd:float[2]' as "SOAP-ENC:arrayType" ),
      (
      SELECT XMLAGG( XMLELEMENT( 'number', f ) ORDER BY seqno )
      FROM SoapData
      )
   )
);
```

ストアドプロシージャ echoFloatArray は、次の XML を相互運用サーバに送信します。

```
<inputFloatArray SOAP-ENC:arrayType="xsd:float[2]">
<number>99.9990005493164</number>
<number>199.998992919922</number>
</inputFloatArray>
```

相互運用サーバからの応答は次のようになります。

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<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
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAPSDK4:echoFloatArrayResponse
      xmlns:SOAPSDK4="http://soapinterop.org/">
```

```
      <Result SOAPSDK3:arrayType="SOAPSDK1:float[2]"
        SOAPSDK3:offset="[0]"
        SOAPSDK2:type="SOAPSDK3:Array">
        <SOAPSDK3:float>99.9990005493164</SOAPSDK3:float>
        <SOAPSDK3:float>199.998992919922</SOAPSDK3:float>
      </Result>
    </SOAPSDK4:echoFloatArrayResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

応答が変数に格納された場合は、OPENXML を使用して解析できます。

```
SELECT * FROM OPENXML( resp,'//*:Result/*' )
WITH ( varFloat FLOAT 'text()' );
```

varFloat
99.9990005493
199.9989929199

2. 次の例のパラメータは、構造体を表します。

```
SELECT echoStruct(
   XMLELEMENT('inputStruct',
      (
      SELECT XMLFOREST( s as varString,
                        i as varInt,
                        f as varFloat )
      FROM SoapData
      WHERE seqno=1
      )
   )
);
```

ストアドプロシージャ echoStruct は、次の XML を相互運用サーバに送信します。

```
<inputStruct>
  <varString>Ninety-Nine</varString>
  <varInt>99</varInt>
  <varFloat>99.9990005493164</varFloat>
</inputStruct>
```

相互運用サーバからの応答は次のようになります。

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<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
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAPSDK4:echoStructResponse
      xmlns:SOAPSDK4="http://soapinterop.org/">
      <Result href="#id1"/>
    </SOAPSDK4:echoStructResponse>
    <SOAPSDK5:SOAPStruct
      xmlns:SOAPSDK5="http://soapinterop.org/xsd"
      id="id1"
```

```
      SOAPSDK3:root="0"
      SOAPSDK2:type="SOAPSDK5:SOAPStruct">
      <varString>Ninety-Nine</varString>
      <varInt>99</varInt>
      <varFloat>99.9990005493164</varFloat>
    </SOAPSDK5:SOAPStruct>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

応答が変数に格納された場合は、OPENXML を使用して解析できます。

```
SELECT * FROM OPENXML( resp,'//*:Body/*:SOAPStruct' )
WITH (
varString LONG VARCHAR 'varString',
varInt INT 'varInt',
varFloat FLOAT 'varFloat' );
```

varString	varInt	varFloat
Ninety-Nine	99	99.9990005493

3. 次の例のパラメータは、構造体の配列を表します。

```
SELECT echoStructArray(
   XMLELEMENT( 'inputStructArray',
      XMLATTRIBUTES( 'http://soapinterop.org/xsd' AS "xmlns:q2",
                     'q2:SOAPStruct[2]' AS "SOAP-ENC:arrayType" ),
         (
         SELECT XMLAGG(
            XMLElement('q2:SOAPStruct',
            XMLFOREST( s as varString,
                       i as varInt,
                       f as varFloat )
            )
         ORDER BY seqno
         )
      FROM SoapData
      )
   )
);
```

ストアドプロシージャ echoFloatArray は、次の XML を相互運用サーバに送信します。

```
<inputStructArray xmlns:q2="http://soapinterop.org/xsd"
  SOAP-ENC:arrayType="q2:SOAPStruct[2]">
  <q2:SOAPStruct>
    <varString>Ninety-Nine</varString>
    <varInt>99</varInt>
    <varFloat>99.9990005493164</varFloat>
  </q2:SOAPStruct>
  <q2:SOAPStruct>
    <varString>Hundred and Ninety-Nine</varString>
    <varInt>199</varInt>
    <varFloat>199.998992919922</varFloat>
  </q2:SOAPStruct>
</inputStructArray>
```

相互運用サーバからの応答は次のようになります。

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<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
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAPSDK4:echoStructArrayResponse
    xmlns:SOAPSDK4="http://soapinterop.org/">
    <Result xmlns:SOAPSDK5="http://soapinterop.org/xsd"
      SOAPSDK3:arrayType="SOAPSDK5:SOAPStruct[2]"
      SOAPSDK3:offset="[0]" SOAPSDK2:type="SOAPSDK3:Array">
      <SOAPSDK5:SOAPStruct href="#id1"/>
      <SOAPSDK5:SOAPStruct href="#id2"/>
    </Result>
  </SOAPSDK4:echoStructArrayResponse>
  <SOAPSDK6:SOAPStruct
    xmlns:SOAPSDK6="http://soapinterop.org/xsd"
    id="id1"
    SOAPSDK3:root="0"
    SOAPSDK2:type="SOAPSDK6:SOAPStruct">
    <varString>Ninety-Nine</varString>
    <varInt>99</varInt>
    <varFloat>99.9990005493164</varFloat>
  </SOAPSDK6:SOAPStruct>
  <SOAPSDK7:SOAPStruct
    xmlns:SOAPSDK7="http://soapinterop.org/xsd"
    id="id2"
    SOAPSDK3:root="0"
    SOAPSDK2:type="SOAPSDK7:SOAPStruct">
    <varString>Hundred and Ninety-Nine</varString>
    <varInt>199</varInt>
    <varFloat>199.998992919922</varFloat>
  </SOAPSDK7:SOAPStruct>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

応答が変数に格納された場合は、OPENXML を使用して解析できます。

```
SELECT * FROM OPENXML( resp,'//*:Body/*:SOAPStruct' )
WITH (
varString LONG VARCHAR 'varString',
varInt INT 'varInt',
varFloat FLOAT 'varFloat' );
```

varString	varInt	varFloat
Ninety-Nine	99	99.9990005493
Hundred and Ninety-Nine	199	199.9989929199

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ファンクションの宣言済みパラメータは、そのプロシージャまたはファンクション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 16¥¥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。

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>
```

参照

- 「句の値に使用する代入パラメータ」820 ページ

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' );
```

参照

- 「-zoc データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「sa_server_option システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

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。

番号	名前	SOAP フォールト	説明
400	Bad Request	Client.BadRequest	HTTP 要求が正しくないか不正です。
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.RequestTimeout	要求の受信中に最大接続アイドル時間が超過しました。
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。

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、*%SQLANYSAMP16%¥SQLAnywhere¥HTTP* フォルダにあります。サンプルの詳細については、*%SQLANYSAMP16%¥SQLAnywhere¥HTTP¥readme.txt* を参照してください。

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、SQL Anywhere データベースサーバを使用して 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。

必要なソフトウェア

- SQL Anywhere 16

前提知識と経験

- XML の知識

- MIME (Multipurpose Internet Mail Extensions) タイプの知識

- SQL Anywhere Web サービスの基本的な知識

目的

- 新しい SQL Anywhere Web サーバデータベースを作成し、起動します。

- 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 SQL Anywhere Web クライアントデータベースを作成し、起動します。

- HTTP:POST 要求を、Web クライアントからデータベース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。

権限

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

関連項目

- 「HTTP Web サーバとしての SQL Anywhere」 749 ページ

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

このレッスンの目標は、Web サービスが実行されている SQL Anywhere Web サーバを設定することです。

前提条件

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「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。」

♦ タスク

1. Web サービス定義を含めるために使用する SQL Anywhere データベースを作成します。

   ```
   dbinit -dba DBA,sql 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。

   ```
   dbsrv16 -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=sql;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。

次の手順

「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」827 ページに進みます。

参照

- 「CREATE SERVICE 文 [HTTP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE PROCEDURE 文 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE FUNCTION 文 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「HTTP_VARIABLE 関数 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「HTTP Web サーバとしての SQL Anywhere」749 ページ

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

このレッスンでは、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

このレッスンは、レッスン 1 で説明したように Web サーバが設定されていることを前提としています。このレッスンで説明している、Web クライアントからの要求を受信するデータベースサーバの設定の詳細については、「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」826 ページを参照してください。

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「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。」

内容と備考

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。

♦ タスク

1. Web クライアントのプロシージャを含めるために使用する SQL Anywhere データベースを作成します。

```
dbinit -dba DBA,sql echo_client
```

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 クライアントとして動作します。

```
dbsrv16 echo_client.db
```

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=sql;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。

Attribute	Value
Status	HTTP/1.1 200 OK
Body	<hello>this is xml</hello>
Server	SQLAnywhere/16.0.1234
Expires	Tue, 09 Oct 2012 21:06:01 GMT
Date	Tue, 09 Oct 2012 21:06:01 GMT
Content-Type	text/xml; charset=windows-1252
Connection	close

次に、Web サーバに送信される HTTP パケットの例を示します。

```
POST /EchoService HTTP/1.0
ASA-Id: 46758096650a44088c77237cc8719d5c
User-Agent: SQLAnywhere/16.0.1234
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>
```

Web サーバからの応答は次のようになります。

```
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/16.0.1234

<hello>this is xml</hello>
```

参照

- 「CREATE PROCEDURE 文 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE FUNCTION 文 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』

チュートリアル : SQL Anywhere 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。

必要なソフトウェア

●SQL Anywhere 16

前提知識と経験

●SOAP の知識

●SQL Anywhere Web サービスの基本的な知識

目的

●新しい SQL Anywhere Web サーバデータベースを作成し、起動します。

●SOAP Web サービスを作成する。

●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 SQL Anywhere Web クライアントデータベースを作成し、起動します。

●SOAP 要求を、Web クライアントからデータベースサーバに送信します。

●SOAP 応答を、データベースサーバから Web クライアントに送信します。

権限

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

関連項目

●「HTTP Web サーバとしての SQL Anywhere」749 ページ

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

このレッスンでは、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。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。

前提条件

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：SQL Anywhere を使用した SOAP/DISH サービスへのアクセス」

♦ タスク

1. Web サービス定義を含めるために使用する SQL Anywhere データベースを作成します。

```
dbinit -dba DBA,sql 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。

```
dbsrv16 -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=sql;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。

注意
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 SQL Anywhere 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

この時点で、華氏から摂氏への温度変換を行うサービスを提供する SQL Anywhere 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。

次の手順

次のレッスンでは、SOAP 要求を Web サーバに送信し、Web サーバから SOAP 応答を受信する、クライアントの例を開発します。「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」834 ページに進みます。

参照

- 「DISH サービスを作成する方法」758 ページ
- 「CREATE SERVICE 文 [SOAP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE PROCEDURE 文 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE FUNCTION 文 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「SOAP_HEADER 関数 [SOAP]」『SQL Anywhere サーバ SQL リファレンス』
- 「NEXT_SOAP_HEADER 関数 [SOAP]」『SQL Anywhere サーバ SQL リファレンス』
- 「SOAP ネームスペース URI の要件」801 ページ
- 「OPENXML 演算子」『SQL Anywhere サーバ SQL リファレンス』

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

このレッスンでは、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

このレッスンは、前のレッスンで説明したように Web サーバが設定されていることを前提としています。「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」830 ページを参照してください。

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：SQL Anywhere を使用した SOAP/DISH サービスへのアクセス」

内容と備考

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。

♦ タスク

1. 次のコマンドを実行して、SQL Anywhere データベースを作成します。

```
dbinit -dba DBA,sql 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。

```
dbsrv16 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=sql;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。

次の手順

次のレッスンでは、SOAP 要求を Web サーバに送信して SOAP 応答を調べます。「レッスン 3：SOAP 要求の送信と SOAP 応答の受信」836 ページに進みます。

参照

- 「CREATE PROCEDURE 文 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE FUNCTION 文 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』

レッスン 3：SOAP 要求の送信と SOAP 応答の受信

このレッスンでは、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

このレッスンは、レッスン 1 で説明したように Web サーバが設定されていることを前提としています。「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」830 ページを参照してください。

このレッスンは、レッスン 2 で説明したように Web クライアントが設定されていることを前提としています。「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」834 ページを参照してください。

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：SQL Anywhere を使用した SOAP/DISH サービスへのアクセス」

♦ タスク

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=sql;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 関数を使用して 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
```

参照

- 「sa_server_option システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

チュートリアル : 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、SQL Anywhere Web サーバ上の SOAP/DISH サービスにアクセスする 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。

必要なソフトウェア

- SQL Anywhere 16
- Visual Studio

前提知識と経験

- SOAP の知識
- .NET フレームワークの知識
- SQL Anywhere Web サービスの基本的な知識

目的

- 新しい SQL Anywhere Web サーバデータベースを作成し、起動します。
- SOAP Web サービスを作成する。
- 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 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。

権限

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

関連項目

- 「HTTP Web サーバとしての SQL Anywhere」 749 ページ

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

このレッスンでは、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/DISH Web サービスが実行されている SQL Anywhere Web サーバを設定します。

前提条件

最新バージョンの Visual Studio が必要です。

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：Visual C# を使用した SOAP/DISH Web サービスへのアクセス」

♦ タスク

1. 次のコマンドを使用して SQL Anywhere デモデータベースを起動します。

   ```
   dbsrv16 -xs http(port=8082) "%SQLANYSAMP16%¥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

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 SQL Anywhere Web サーバが設定されました。

次の手順

次のレッスンでは、Web サーバと通信するための Visual C# アプリケーションを作成します。「レッスン 2 : Web サーバと通信するための Visual C# アプリケーションの作成」841 ページに進みます。

参照

- 「DISH サービスを作成する方法」758 ページ
- 「CREATE SERVICE 文 [SOAP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』

レッスン 2 : Web サーバと通信するための Visual C# アプリケーションの作成

このレッスンでは、Web サーバと通信するための Visual C# アプリケーションを作成します。

前提条件

このレッスンは、レッスン 1 で説明したように Web サーバが設定されていることを前提としています。「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」840 ページを参照してください。

このレッスンを終了するには、最新バージョンの Visual Studio が必要です。

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：Visual C# を使用した SOAP/DISH Web サービスへのアクセス」

内容と備考

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。

この例では、.NET Framework 2.0 の機能を使用しています。

♦ タスク

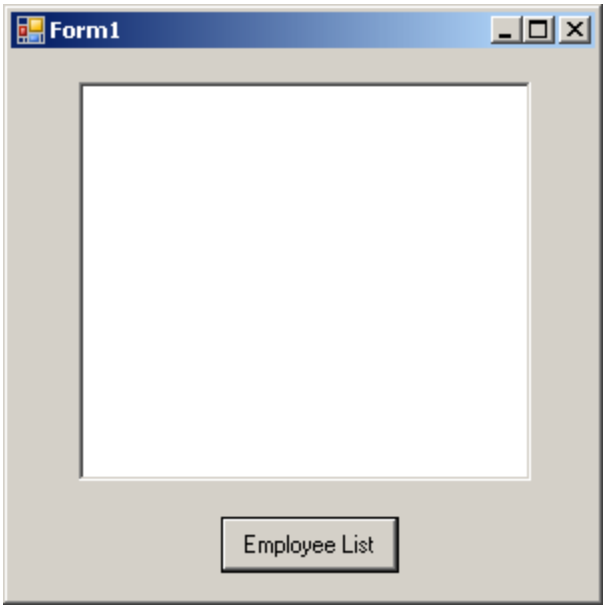
1. Visual Studio を起動します。
2. 新しい Visual C# **Windows フォームのアプリケーション**プロジェクトを作成します。

   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。

      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。

      Visual Studio は、**localhost** を **[ソリューション エクスプローラー]** 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る)。

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 日と仮定する)。

```
(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' />
```

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、SQL Anywhere Web 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。

必要なソフトウェア

- SQL Anywhere 16
- JDK 1.7.0

●JAX-WS 2.2.7 以降のバージョン

前提知識と経験

●SOAP の知識

●Java および JAX-WS の知識

●SQL Anywhere Web サービスの基本的な知識

目的

●新しい SQL Anywhere Web サーバデータベースを作成し、起動します。

●SOAP Web サービスを作成する。

●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。

権限

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

関連項目

●「HTTP Web サーバとしての SQL Anywhere」749 ページ

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

このレッスンでは、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/DISH Web サービスが実行されている SQL Anywhere Web サーバを設定します。

前提条件

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「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」

内容と備考

このレッスンでは、次のレッスンで使用する 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。

♦ タスク

1. 次のコマンドを使用して SQL Anywhere デモデータベースを起動します。

   ```
   dbsrv16 -xs http(port=8082) "%SQLANYSAMP16%¥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。**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 SQL Anywhere Web サーバが設定されました。

次の手順

次のレッスンでは、Web サーバと通信するための Java アプリケーションを作成します。「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」850 ページに進みます。

参照

- 「DISH サービスを作成する方法」758 ページ
- 「CREATE SERVICE 文 [SOAP Web サービス]」『SQL Anywhere サーバ SQL リファレンス』

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

このレッスンでは、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

このレッスンの内容は、レッスン 1 で行われたステップによって異なります。 「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」847 ページを参照してください。

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「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」

内容と備考

本書作成時点では、JAX-WS は JDK 1.7.0 に含まれており、JAX-WS の最新バージョンは 2.2.7 でした。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。

♦ タスク

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 "http://localhost: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* 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。

   wsimport アプリケーションは、現在の作業ディレクトリに *localhost¥_8082¥demo¥ws* という新しいサブフォルダを作成します。次に、ディレクトリ *ws* の内容のリストを示します。

   - *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 *SASoapDemo.java* として保存します。現在の作業ディレクトリには、*localhost* サブフォルダが含まれている必要があります。

```
// SASoapDemo.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 localhost._8082.demo.ws.*;

public class SASoapDemo
{
  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();
        }
      }
    }
```

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。

> **注意**
> このアプリケーションでは、レッスン 1 で指示するように、SQL Anywhere Web サーバがポート 8082 で受信することを前提としています。**import localhost._8082.demo.ws.*** コード行の「8082」 部分を、SQL Anywhere Web サーバを起動したときに指定したポート番号に置き換えます。

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、http://docs.oracle.com/javase/にある javax.xml.bind.JAXBElement クラス API のマニュアルを参照してください。

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。

   ```
   javac SASoapDemo.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 SASoapDemo
   ```

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。

結果

次に、実行中の SASoapDemo 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。

3 層コンピューティングと分散トランザクション

SQL Anywhere は、データベースとして使用するほかに、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。

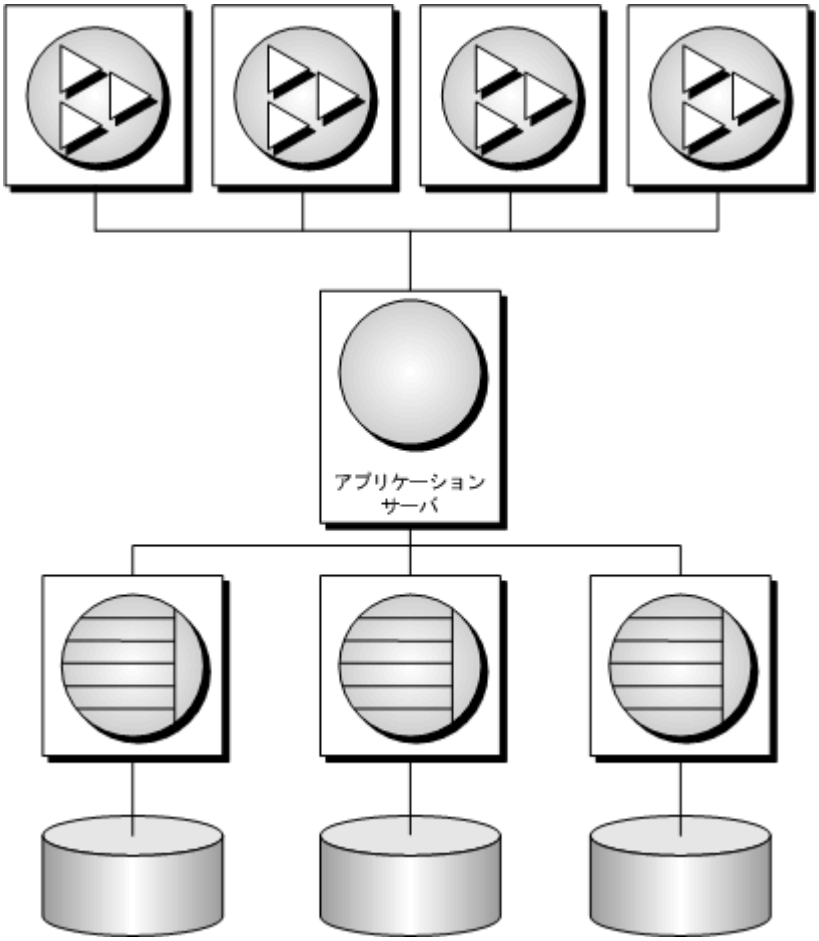
3 層環境では、クライアントアプリケーションと一連のリソースマネージャの間にアプリケーションサーバを置きますが、これが一般的な分散トランザクション環境です。Sybase EAServer と他のアプリケーションサーバの一部もトランザクションサーバです。

Sybase 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。SQL Anywhere 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、SQL Anywhere を使用できます。

SQL Anywhere 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。この項では、3 層コンピューティングの概念とアーキテクチャ、SQL Anywhere 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 SQL Anywhere 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。

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 Sybase EAServer 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。



Sybase 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、こ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、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、EAServer のマニュアルを参照してください。

3 層コンピューティングにおける分散トランザクション

クライアントアプリケーションまたはアプリケーションサーバが SQL Anywhere などの単一のトランザクション処理データベースとともに動作するときは、データベース自体の外部にトランザクション論理は必要ありません。しかし、複数のリソースマネージャとともに動作するときは、トランザクションで使用される複数のリソースにわたってトランザクション制御を行う必要があります。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。

Sybase EAServer をはじめとする多くのトランザクションサーバは、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、クライアントアプリケーションにトランザクション

サービスを提供します。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。この項で説明する機能を使用するには、DTC がインストールされている必要があります。

分散トランザクションにおける SQL Anywhere

DTC が調整するトランザクションに SQL Anywhere を追加できます。つまり、Sybase EAServer や Microsoft Transaction Server などのトランザクションサーバを使用して、SQL Anywhere データベースを分散トランザクションの中で使用できます。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。

分散トランザクションに関する用語

この項は、分散トランザクションについて基本的な知識を持っている方を対象としています。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。この項では、よく使用される用語をいくつか説明します。

- 「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。

  分散トランザクションの中で ADO.NET、OLE DB、または ODBC を通してアクセスする場合、SQL Anywhere データベースサーバはリソースマネージャとして動作します。SQL Anywhere .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。SQL Anywhere .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。

  SQL Anywhere 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。

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。

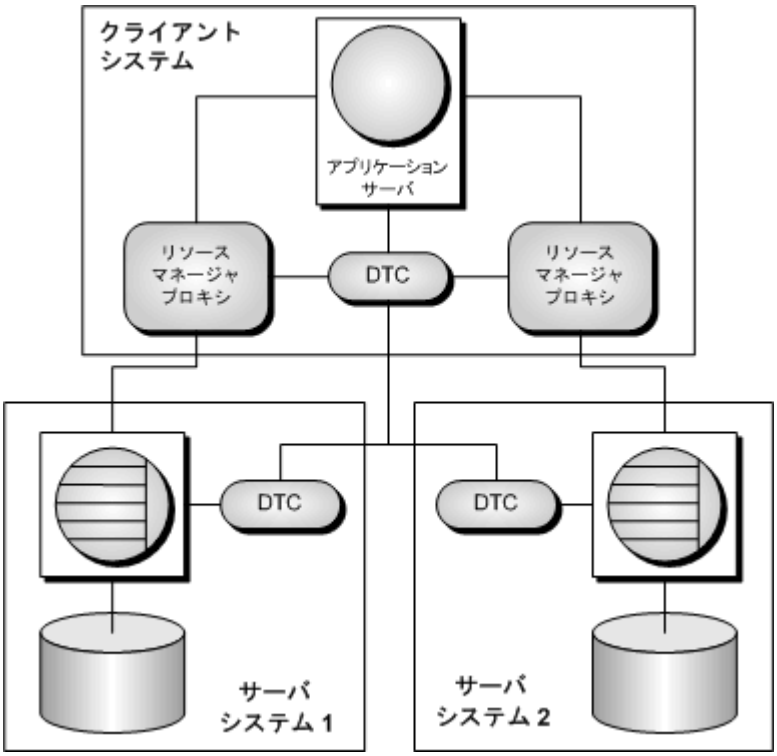
アプリケーションサーバが DTC を使用する方法

Sybase EAServer と Microsoft Transaction Server は、どちらもコンポーネントサーバです。アプリケーション論理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。

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) サービスを実行中にしてください。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 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。

分散トランザクション

SQL Anywhere は、分散トランザクションにエンリストされている間は、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、SQL Anywhere 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。SQL Anywhere 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設定されます。

- 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る) 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ずに直接 SQL Anywhere に発行すると、エラーが発生します。ただし、トランザクションはアボートしません。
- 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。

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 SQL Anywhere 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	SQL Anywhere の独立性レベル
ISOLATIONLEVEL_UNSPECIFIED	0
ISOLATIONLEVEL_CHAOS	0
ISOLATIONLEVEL_READUNCOMMITTED	0
ISOLATIONLEVEL_BROWSE	0
ISOLATIONLEVEL_CURSORSTABILITY	1

DTC の独立性レベル	SQL Anywhere の独立性レベル
ISOLATIONLEVEL_READCOMMITTED	1
ISOLATIONLEVEL_REPEATABLEREAD	2
ISOLATIONLEVEL_SERIALIZABLE	3
ISOLATIONLEVEL_ISOLATED	3

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、SQL Anywhere 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**　指定した時間内に再エンリストが完了しないときは、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。

参照

- 「-tmf データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「-tmt データベースサーバオプション」『SQL Anywhere サーバ データベース管理』

データベースツールインタフェース (DBTools)

SQL Anywhere は Sybase Central とデータベース管理用のユーティリティのセットを含みます。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。共有ライブラリは、Windows オペレーティングシステム、Linux、UNIX、および Mac OS X 向けに提供されています。Windows 向けのライブラリ名は *dbtool16.dll* です。Linux および UNIX 向けのライブラリ名は *libdbtool16_r.so* です。Mac OS X 向けのライブラリ名は *libdbtool16_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。この項では、データベースツールライブラリに対するインタフェースについて説明します。この項の説明は、使用中の開発環境からライブラリルーチンを呼び出す方法に精通しているユーザを対象にしています。

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。

Windows Mobile

Windows Mobile には *dbtool16.dll* ライブラリが用意されていますが、DBToolsInit、DBToolsFini、DBRemoteSQL、DBSynchronizeLog のエントリポイントだけが含まれています。その他のエントリポイントは、Windows Mobile 用に提供されていません。

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、SQL Anywhere インストールディレクトリの *SDK¥Include* サブフォルダ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 ヘッダファイルには、エラー戻り値が含まれています。

dbrmt.h ヘッダファイル

SQL Anywhere に含まれる *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、SQL Anywhere インストールディレクトリの *SDK¥Include* サブフォルダ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。

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。*libdbtool16.so* (非スレッド) または *libdbtool16_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 インタフェース用のインポートライブラリは、Windows および Windows Mobile 用の SQL Anywhere に用意されています。Windows の場合、インポートライブラリは SQL Anywhere インストールディレクトリの *SDK¥Lib¥x86* サブフォルダと *SDK¥Lib¥x64* サブフォルダにあります。Windows Mobile の場合は、SQL Anywhere インストールディレクトリの *SDK¥Lib¥CE¥Arm.50* サブフォルダにあります。提供される DBTools インポートライブラリは次のとおりです。

コンパイラ	ライブラリ
Microsoft Windows	*dbtlstm.lib*
Microsoft Windows Mobile	*dbtool16.lib*

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 SQL Anywhere メッセージライブラリ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 );
```

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 );
```

参照

- 「DBTools 構造体」868 ページ

コールバック関数

DBTools 構造体には MSG_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 ライブラリはコールバック関数を使用して、DBTools 関数から、次の 4 つの目的を持ってユーザに送られたメッセージを処理します。

- **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。

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
```

MSG_CALLBACK は、SQL Anywhere に付属する *dllapi.h* ヘッダファイルに定義されています。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を付けて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 );
}
```

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。

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。

DBTools の例

このサンプルとコンパイル手順は、*%SQLANYSAMP16%¥SQLAnywhere¥DBTools* ディレクトリにあります。サンプルプログラム自体は *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。

```
#define WIN32

#include <stdio.h>
#include <string.h>
#include "windows.h"
#include "sqldef.h"
#include "dbtools.h"
extern short _callback ConfirmCallBack( char * str )
{
   if( MessageBox( NULL, str, "Backup",
       MB_YESNO|MB_ICONQUESTION ) == IDYES )
   {
      return 1;
   }
   return 0;
}
extern short _callback MessageCallBack( char * str )
{
   if( str != NULL )
   {
      fprintf( stdout, "%s¥n", str );
   }
   return 0;
}
extern short _callback StatusCallBack( char * str )
{
   if( str != NULL )
   {
      fprintf( stdout, "%s¥n", str );
   }
   return 0;
}
extern short _callback ErrorCallBack( char * str )
{
   if( str != NULL )
   {
      fprintf( stdout, "%s¥n", str );
   }
   return 0;
}
typedef void (CALLBACK *DBTOOLSPROC)( void * );
typedef short (CALLBACK *DBTOOLSFUNC)( void * );
```

```
// Main entry point into the program.
int main( int argc, char * argv[] )
{
   a_dbtools_info  dbt_info;
   a_backup_db     backup_info;
   char            dir_name[ _MAX_PATH + 1];
   char            connect[ 256 ];
   HINSTANCE       hinst;
   DBTOOLSFUNC     dbbackup;
   DBTOOLSFUNC     dbtoolsinit;
   DBTOOLSPROC     dbtoolsfini;
   short           ret_code;

   // Always initialize to 0 so new versions
   // of the structure will be compatible.
   memset( &dbt_info, 0, sizeof( a_dbtools_info ) );
   dbt_info.errorrtn = (MSG_CALLBACK)MessageCallBack;;

   memset( &backup_info, 0, sizeof( a_backup_db ) );
   backup_info.version = DB_TOOLS_VERSION_NUMBER;
   backup_info.quiet = 0;
   backup_info.no_confirm = 0;
   backup_info.confirmrtn = (MSG_CALLBACK)ConfirmCallBack;
   backup_info.errorrtn = (MSG_CALLBACK)ErrorCallBack;
   backup_info.msgrtn = (MSG_CALLBACK)MessageCallBack;
   backup_info.statusrtn = (MSG_CALLBACK)StatusCallBack;
   if( argc > 1 )
   {
      strncpy( dir_name, argv[1], _MAX_PATH );
   }
   else
   {
      // DBTools does not expect (or like) a trailing slash
      strcpy( dir_name, "c:¥¥temp" );
   }
   backup_info.output_dir = dir_name;
   if( argc > 2 )
   {
      strncpy( connect, argv[2], 255 );
   }
   else
   {
      strcpy( connect, "DSN=SQL Anywhere 16 Demo" );
   }
   backup_info.connectparms = connect;
   backup_info.quiet = 0;
   backup_info.no_confirm = 0;
   backup_info.backup_database = 1;
   backup_info.backup_logfile = 1;
   backup_info.rename_log = 0;
   backup_info.truncate_log = 0;
   hinst = LoadLibrary( "dbtool16.dll" );
   if( hinst == NULL )
   {
      // Failed
      return EXIT_FAIL;
   }
   dbbackup = (DBTOOLSFUNC) GetProcAddress( (HMODULE)hinst,
      "_DBBackup@4" );
   dbtoolsinit = (DBTOOLSFUNC) GetProcAddress( (HMODULE)hinst,
      "_DBToolsInit@4" );
   dbtoolsfini = (DBTOOLSPROC) GetProcAddress( (HMODULE)hinst,
      "_DBToolsFini@4" );
```

```
    ret_code = (*dbtoolsinit)( &dbt_info );
    if( ret_code != EXIT_OKAY ) {
        return ret_code;
    }
    ret_code = (*dbbackup)( &backup_info );
    (*dbtoolsfini)( &dbt_info );
    FreeLibrary( hinst );
    return ret_code;
}
```

SQL Anywhere データベースツール C API リファレンス

ヘッダファイル

- dbtools.h
- dbrmt.h

DBTools 構造体

この項では、DBTools ライブラリとの間で情報を交換するために使用する構造体について説明します。構造体はアルファベット順に示します。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。a_remote_sql 構造体は、*dbrmt.h* に定義されています。

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も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 10.0.0 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、*dbtool9.dll* や *dbtool11.dll* にアクセスできなくなります。

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するように、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なくなります。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 "version" 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。

DBBackup 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

「ソフトウェアコンポーネントの終了コード」956 ページにリストされているリターンコード。

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。各タスクの詳細については、「バックアップユーティリティ (dbbackup)」『SQL Anywhere サーバ データベース管理』を参照してください。

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。

参照

- a_backup_db 構造体 [データベースツール]885 ページ
- 「BACKUP 文」『SQL Anywhere サーバ SQL リファレンス』

DBChangeLogName 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(const 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

「ソフトウェアコンポーネントの終了コード」956 ページにリストされているリターンコード。

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。

参照

- a_change_log 構造体 [データベースツール]889 ページ
- 「トランザクションログユーティリティ (dblog)」『SQL Anywhere サーバ データベース管理』
- 「ALTER DATABASE 文」『SQL Anywhere サーバ SQL リファレンス』

DBCreate 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

「ソフトウェアコンポーネントの終了コード」956 ページにリストされているリターンコード。

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。

参照

- a_create_db 構造体 [データベースツール]892 ページ
- 「初期化ユーティリティ (dbinit)」『SQL Anywhere サーバ データベース管理』

DBCreatedVersion 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

「ソフトウェアコンポーネントの終了コード」956 ページにリストされているリターンコード。

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。

参照

- a_db_version_info 構造体 [データベースツール]901 ページ
- バージョン列挙 [データベースツール]884 ページ
- 「CREATE DATABASE 文」『SQL Anywhere サーバ SQL リファレンス』

DBErase 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

「ソフトウェアコンポーネントの終了コード」956 ページにリストされているリターンコード。

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。

参照

- an_erase_db 構造体 [データベースツール]942 ページ
- 「消去ユーティリティ (dberase)」『SQL Anywhere サーバ データベース管理』

DBInfo 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

「ソフトウェアコンポーネントの終了コード」956 ページにリストされているリターンコード。

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。

参照

- a_db_info 構造体 [データベースツール]897 ページ
- DBInfoDump メソッド [データベースツール]872 ページ
- DBInfoFree メソッド [データベースツール]872 ページ
- 「情報ユーティリティ (dbinfo)」『SQL Anywhere サーバ データベース管理』

DBInfoDump 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

「ソフトウェアコンポーネントの終了コード」956 ページにリストされているリターンコード。

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。

参照

- a_db_info 構造体 [データベースツール]897 ページ
- DBInfo メソッド [データベースツール]871 ページ
- DBInfoFree メソッド [データベースツール]872 ページ

DBInfoFree 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

「ソフトウェアコンポーネントの終了コード」956 ページにリストされているリターンコード。

参照

- a_db_info 構造体 [データベースツール]897 ページ
- DBInfo メソッド [データベースツール]871 ページ
- DBInfoDump メソッド [データベースツール]872 ページ

DBLicense 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

「ソフトウェアコンポーネントの終了コード」956 ページにリストされているリターンコード。

参照

- a_dblic_info 構造体 [データベースツール]902 ページ
- 「サーバライセンス取得ユーティリティ (dblic)」『SQL Anywhere サーバ データベース管理』

DBLogFileInfo 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

「ソフトウェアコンポーネントの終了コード」956 ページにリストされているリターンコード。

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。

参照

- a_log_file_info 構造体 [データベースツール]904 ページ

DBRemoteSQL 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

「ソフトウェアコンポーネントの終了コード」956 ページにリストされているリターンコード。

備考

アクセスできる機能の詳細については、「SQL Remote Message Agent ユーティリティ (dbremote)」『SQL Remote』を参照してください。

参照

- a_remote_sql 構造体 [データベースツール]905 ページ
- 「SQL Remote システム」『SQL Remote』

DBSynchronizeLog 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

「ソフトウェアコンポーネントの終了コード」956 ページにリストされているリターンコード。

参照

- a_sync_db 構造体 [データベースツール]916 ページ
- 「dbmlsync の DBTools インタフェース」『Mobile Link クライアント管理』

DBToolsFini 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

「ソフトウェアコンポーネントの終了コード」956 ページにリストされているリターンコード。

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。

参照

- a_dbtools_info 構造体 [データベースツール]904 ページ
- DBToolsInit メソッド [データベースツール]875 ページ

DBToolsInit 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

「ソフトウェアコンポーネントの終了コード」956 ページにリストされているリターンコード。

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。

参照

- a_dbtools_info 構造体 [データベースツール]904 ページ
- DBToolsFini メソッド [データベースツール]874 ページ
- 「DBTools の例」866 ページ

DBToolsVersion 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。

参照

- 「バージョン番号と互換性」865 ページ

DBTranslateLog 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

「ソフトウェアコンポーネントの終了コード」956 ページにリストされているリターンコード。

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。

参照

- a_translate_log 構造体 [データベースツール]932 ページ
- 「ログ変換ユーティリティ (dbtran)」『SQL Anywhere サーバ データベース管理』

DBTruncateLog 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

「ソフトウェアコンポーネントの終了コード」956 ページにリストされているリターンコード。

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。

参照

- a_truncate_log 構造体 [データベースツール]938 ページ
- 「バックアップユーティリティ (dbbackup)」『SQL Anywhere サーバ データベース管理』
- 「BACKUP 文」『SQL Anywhere サーバ SQL リファレンス』

DBUnload 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

「ソフトウェアコンポーネントの終了コード」956 ページにリストされているリターンコード。

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。

参照

- an_unload_db 構造体 [データベースツール]943 ページ
- 「アンロードユーティリティ (dbunload)」『SQL Anywhere サーバ データベース管理』
- 「抽出ユーティリティ (dbxtract)」『SQL Remote』

DBUpgrade 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

「ソフトウェアコンポーネントの終了コード」956 ページにリストされているリターンコード。

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。

参照

- an_upgrade_db 構造体 [データベースツール]953 ページ
- 「アップグレードユーティリティ (dbupgrad)」『SQL Anywhere サーバ データベース管理』
- 「ALTER DATABASE 文」『SQL Anywhere サーバ SQL リファレンス』

DBValidate 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

「ソフトウェアコンポーネントの終了コード」956 ページにリストされているリターンコード。

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。

参照

- a_validate_db 構造体 [データベースツール]940 ページ
- 「検証ユーティリティ (dbvalid)」『SQL Anywhere サーバ データベース管理』
- 「VALIDATE 文」『SQL Anywhere サーバ SQL リファレンス』

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	値
BACKUP_AUTO_TUNE_UNSPECIFIED	AUTO TUNE WRITERS 句を未指定にするために使用します。	0

メンバー名	説明	値
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。	

参照

- a_backup_db 構造体 [データベースツール]885 ページ
- DBBackup メソッド [データベースツール]869 ページ

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	値
BACKUP_CHKPT_LOG_COPY	WITH CHECKPOINT LOG COPY 句の生成に使用します。	0
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。	

参照

- a_backup_db 構造体 [データベースツール]885 ページ
- DBBackup メソッド [データベースツール]869 ページ

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	値
BACKUP_HISTORY_UNSPECIFIED	HISTORY 句を未指定にするために使用します。	0
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。	

参照

- a_backup_db 構造体 [データベースツール]885 ページ
- DBBackup メソッド [データベースツール]869 ページ

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。

参照

- a_create_db 構造体 [データベースツール]892 ページ
- DBCreate メソッド [データベースツール]870 ページ

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。

参照

- a_create_db 構造体 [データベースツール]892 ページ
- DBCreate メソッド [データベースツール]870 ページ

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。

メンバー名	説明
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。

参照

- an_unload_db 構造体 [データベースツール]943 ページ
- DBUnload メソッド [データベースツール]877 ページ

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。

参照

- a_translate_log 構造体 [データベースツール]932 ページ
- DBTranslateLog メソッド [データベースツール]876 ページ

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	値
VALIDATE_NORMAL	デフォルトのチェックのみで検証します。	0
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。	

参照

- a_validate_db 構造体 [データベースツール]940 ページ
- DBValidate メソッド [データベースツール]878 ページ
- 「チェックサムを使用した破損の検出」『SQL Anywhere サーバ データベース管理』

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。

メンバー名	説明
VB_VERBOSE	冗長出力。デバッグ用。

参照

- a_create_db 構造体 [データベースツール]892 ページ
- DBCreate メソッド [データベースツール]870 ページ

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

参照

- a_db_version_info 構造体 [データベースツール]901 ページ
- DBCreatedVersion メソッド [データベースツール]870 ページ

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

メンバー名	タイプ	説明
auto_tune_writers	char	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。
backup_comment	const char *	WITH COMMENT 句に使用するコメント
backup_database	a_bit_field	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。

メンバー名	タイプ	説明
backup_history	char	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。
backup_interrupted	char	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。
backup_logfile	a_bit_field	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。
chkpt_log_type	char	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。
confirmrtn	MSG_CALLBACK	確認要求コールバックルーチンのアドレス、または NULL。

メンバー名	タイプ	説明
connectparms	const char *	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とる。"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:¥SQLAny16¥bin32¥dbsrv16.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:¥SQLAny16¥bin32¥dbsrv16.exe".
errorrtn	MSG_CALLBACK	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。
hotlog_filename	const char *	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。
msgrtn	MSG_CALLBACK	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。
no_confirm	a_bit_field	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。
output_dir	const char *	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。

メンバー名	タイプ	説明
page_blocksize	a_sql_uint32	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。
progress_messages	a_bit_field	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。
quiet	a_bit_field	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。
rename_local_log	a_bit_field	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。
rename_log	a_bit_field	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。
server_backup	a_bit_field	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。
statusrtn	MSG_CALLBACK	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。

メンバー名	タイプ	説明
truncate_log	a_bit_field	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。
version	unsigned short	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。
wait_after_end	a_bit_field	終了後に待ちます。 dbbackup -wa オプションによって TRUE に設定されます。
wait_before_start	a_bit_field	開始前に待ちます。 dbbackup -wb オプションによって TRUE に設定されます。

参照

- チェックポイント列挙 [データベースツール]879 ページ
- DBBackup メソッド [データベースツール]869 ページ
- 「コールバック関数」863 ページ
- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』

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

メンバー名	タイプ	説明
change_logname	a_bit_field	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。
change_mirrorname	a_bit_field	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。
dbname	const char *	データベースファイル名。
encryption_key	char *	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。
errorrtn	MSG_CALLBACK	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。
generation_number	unsigned short	新しい世代番号。予約済み。0 を使用します。
ignore_dbsync_trunc	a_bit_field	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。
ignore_ltm_trunc	a_bit_field	予約済み。FALSE を使用します。

メンバー名	タイプ	説明
ignore_remote_trunc	a_bit_field	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。
logname	const char *	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。
mirrorname	const char *	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。
msgrtn	MSG_CALLBACK	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。
query_only	a_bit_field	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。
quiet	a_bit_field	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。
set_generation_number	a_bit_field	予約済み。FALSE を使用します。
version	unsigned short	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。

メンバー名	タイプ	説明
zap_current_offset	char *	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。
zap_starting_offset	char *	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。

参照

- DBChangeLogName メソッド [データベースツール]869 ページ

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

メンバー名	タイプ	説明
accent_sensitivity	char	'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。

メンバー名	タイプ	説明
avoid_view_collisions	a_bit_field	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。
blank_pad	a_bit_field	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。
case_sensitivity_use_default	a_bit_field	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。
checksum	a_bit_field	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。
data_store_type	const char *	予約済み。NULL を使用します。

メンバー名	タイプ	説明
db_size	unsigned int	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。
db_size_unit	int	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。
dba_pwd	char *	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。
dba_uid	char *	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。
dbname	const char *	データベースファイル名。
default_collation	const char *	データベースの照合。dbinit -z オプションと同等です。
encoding	const char *	文字セットエンコード。dbinit -ze オプションと同等です。

メンバー名	タイプ	説明
encrypt	a_bit_field	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。
encrypted_tables	a_bit_field	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。
encryption_algorithm	const char *	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。
encryption_key	const char *	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。
errorrtn	MSG_CALLBACK	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。
iq_params	void *	予約済み。NULL を使用します。

メンバー名	タイプ	説明
jconnect	a_bit_field	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。
logname	const char *	新しいトランザクションログ名。dbinit -t オプションと同等です。
mirrorname	const char *	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。
msgrtn	MSG_CALLBACK	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。
nchar_collation	const char *	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。
page_size	unsigned short	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。
respect_case	a_bit_field	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。

メンバー名	タイプ	説明
startline	const char *	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。 例："c:¥SQLAny16¥bin32¥dbsrv16.exe".NULL の場合、SQL Anywhere のデフォルトの START パラメータは "dbeng16 -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。
sys_proc_definer	a_bit_field	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。
verbose	char	冗長列挙 (VB_QUIET、VB_NORMAL、VB_VERBOSE) を参照してください。
version	unsigned short	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。

参照

- 埋め込み列挙 [データベースツール]880 ページ
- 単位列挙 [データベースツール]880 ページ
- 冗長列挙 [データベースツール]883 ページ
- DBCreate メソッド [データベースツール]870 ページ
- 「チェックサムを使用した破損の検出」『SQL Anywhere サーバ データベース管理』

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

メンバー名	タイプ	説明
bit_map_pages	a_sql_uint32	データベース内のビットマップページ数。
charcollationspecbuffer	char *	char 照合文字列バッファへのポインタ。
charcollationspecbufsize	unsigned short	charcollationspecbuffer のサイズ (256+1 以上)。
charencodingbuffer	char *	char エンコード文字列バッファへのポインタ。
charencodingbufsize	unsigned short	charencodingbuffer のサイズ (50+1 以上)。
checksum	a_bit_field	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。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

メンバー名	タイプ	説明
connectparms	const char *	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とる。"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:¥SQLAny16¥bin32¥dbsrv16.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:¥SQLAny16¥bin32¥dbsrv16.exe".
dbbufsize	unsigned short	dbnamebuffer のサイズ (たとえば、_MAX_PATH)。
dbnamebuffer	char *	データベースファイル名バッファへのポインタ。
encrypted_tables	a_bit_field	TRUE に設定した場合、暗号化テーブルがサポートされます。
errorrtn	MSG_CALLBACK	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。
file_size	a_sql_uint32	データベースファイルのサイズ (ページ単位)。
free_pages	a_sql_uint32	空きページ数。
logbufsize	unsigned short	lognamebuffer のサイズ (たとえば、_MAX_PATH)。

メンバー名	タイプ	説明
lognamebuffer	char *	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。
mirrorbufsize	unsigned short	mirrornamebuffer のサイズ (たとえば、_MAX_PATH)。
mirrornamebuffer	char *	ミラーファイル名バッファへのポインタ。
msgrtn	MSG_CALLBACK	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。
ncharcollationspecbuffer	char *	nchar 照合文字列バッファへのポインタ。
ncharcollationspecbufsize	unsigned short	ncharcollationspecbuffer のサイズ (256+1 以上)。
ncharencodingbuffer	char *	nchar エンコード文字列バッファへのポインタ。
ncharencodingbufsize	unsigned short	ncharencodingbuffer のサイズ (50+1 以上)。
other_pages	a_sql_uint32	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。
page_usage	a_bit_field	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。
quiet	a_bit_field	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。

メンバー名	タイプ	説明
statusrtn	MSG_CALLBACK	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。
sysinfo	a_sysinfo	インライン a_sysinfo 構造体。
totals	a_table_info *	a_table_info 構造体へのポインタ。
version	unsigned short	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。

参照

- a_table_info 構造体 [データベースツール]931 ページ
- a_sysinfo 構造体 [データベースツール]930 ページ
- DBInfo メソッド [データベースツール]871 ページ
- DBInfoDump メソッド [データベースツール]872 ページ
- DBInfoFree メソッド [データベースツール]872 ページ
- 「コールバック関数」863 ページ
- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』
- 「チェックサムを使用した破損の検出」『SQL Anywhere サーバ データベース管理』

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

メンバー名	タイプ	説明
created_version	char	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。

メンバー名	タイプ	説明
errorrtn	MSG_CALLBACK	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。
filename	const char *	確認するデータベースファイルの名前。
msgrtn	MSG_CALLBACK	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。
version	unsigned short	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。

参照

- DBCreatedVersion メソッド [データベースツール]870 ページ
- バージョン列挙 [データベースツール]884 ページ

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

メンバー名	タイプ	説明
compname	char *	ライセンスの会社名。
conncount	a_sql_int32	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。
errorrtn	MSG_CALLBACK	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。
exename	char *	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。

メンバー名	タイプ	説明
installkey	char *	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。
msgrtn	MSG_CALLBACK	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。
nodecount	a_sql_int32	ライセンスノード数。
query_only	a_bit_field	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。
quiet	a_bit_field	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。
type	a_license_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。
username	char *	ライセンスのユーザ名。
version	unsigned short	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。

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。

参照

- DBLicense メソッド [データベースツール]873 ページ

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

メンバー名	タイプ	説明
errorrtn	MSG_CALLBACK	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。

参照

- DBToolsInit メソッド [データベースツール]875 ページ
- DBToolsFini メソッド [データベースツール]874 ページ

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

メンバー名	タイプ	説明
dbname	const char *	データベースファイル名。
encryption_key	const char *	データベースファイルの暗号化キー。
errorrtn	MSG_CALLBACK	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。
logname	char *	トランザクションログファイル名のバッファ、または NULL。
logname_size	size_t	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。

メンバー名	タイプ	説明
mirrorname	char *	ミラーログファイル名のバッファ、または NULL。
mirrorname_size	size_t	ミラーログファイル名のバッファのサイズ、または 0。
reserved	void *	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。
version	unsigned short	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。

参照

- DBLogFileInfo メソッド [データベースツール]873 ページ

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

メンバー名	タイプ	説明
name	char	名前を構成する 1 バイトまたはそれ以上のバイト。
next	struct a_name *	リスト内の次の名前へのポインタ、または NULL。

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

メンバー名	タイプ	説明
apply	a_bit_field	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。
argv	char **	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。
batch	a_bit_field	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。
confirmrtn	MSG_CALLBACK	確認要求コールバックルーチンのアドレス、または NULL。

メンバー名	タイプ	説明
connectparms	char *	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とる。"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:¥SQLAny16¥bin32¥dbeng16.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:¥SQLAny16¥bin32¥dbeng16.exe"
debug	a_bit_field	TRUE に設定した場合、デバッグ出力が含まれます。
debug_dump_size	a_sql_uint32	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。
debug_page_offsets	a_bit_field	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。
default_window_title	char *	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。
deleted	a_bit_field	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。

メンバー名	タイプ	説明
encryption_key	char *	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。
errorrtn	MSG_CALLBACK	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。
frequency	a_sql_uint32	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。
full_q_scan	a_bit_field	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。
include_scan_range	char *	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。
latest_backup	a_bit_field	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。
link_debug	a_bit_field	TRUE に設定した場合、リンクのデバッグが有効になります。
locale	char *	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。
log_file_name	const char *	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)。

メンバー名	タイプ	説明
log_size	a_sql_uint32	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。
logrtn	MSG_CALLBACK	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。
max_length	a_sql_uint32	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。
memory	a_sql_uint32	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。
mirror_logs	char *	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。
more	a_bit_field	TRUE に設定してください。

メンバー名	タイプ	説明
msgqueuertn	MSG_QUEUE_CALLBACK	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。
msgrtn	MSG_CALLBACK	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。
no_user_interaction	a_bit_field	TRUE に設定した場合、ユーザの操作を必要としません。
operations	a_sql_uint32	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。

メンバー名	タイプ	説明
patience_retry	a_sql_uint32	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。
progress_index_rtn	SET_PROGRESS_CALLBACK	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。
progress_msg_rtn	MSG_CALLBACK	進行状況メッセージを表示する関数へのポインタ。
queueparms	char *	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。
receive	a_bit_field	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すすめします。dbremote の -r オプションに対応します。

メンバー名	タイプ	説明
receive_delay	a_sql_uint32	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。
remote_output_file_name	char *	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。
rename_log	a_bit_field	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)。
resend_urgency	a_sql_uint32	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。
scan_log	a_bit_field	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。

メンバー名	タイプ	説明
send	a_bit_field	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すすめします。dbremote の -s オプションに対応します。
send_delay	a_sql_uint32	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。
set_window_title_rtn	SET_WINDOW_TITLE_CALLBACK	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。
threads	a_sql_uint32	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。
transaction_logs	char *	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。

メンバー名	タイプ	説明
triggers	a_bit_field	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。
truncate_remote_output_file	a_bit_field	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。
unused	a_bit_field	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。
use_hex_offsets	a_bit_field	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。
use_relative_offsets	a_bit_field	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。
verbose	a_bit_field	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。
version	unsigned short	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。

メンバー名	タイプ	説明
warningrtn	MSG_CALLBACK	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。

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)

参照

- DBRemoteSQL メソッド [データベースツール]873 ページ

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

メンバー名	タイプ	説明
allow_outside_connect	a_bit_field	予約済み。0 を使用します。
allow_schema_change	a_bit_field	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。
apply_dnld_file	const char *	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。

メンバー名	タイプ	説明
argv	char **	この実行における argv 配列。配列の最後の要素は NULL です。
autoclose	a_bit_field	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。
background_retry	a_sql_int32	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。
background_sync	a_bit_field	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。
cache_verbosity	a_bit_field	予約済み。0 を使用します。
ce_argv	char **	予約済み。NULL を使用します。
ce_reproc_argv	char **	予約済み。NULL を使用します。
changing_pwd	a_bit_field	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。
confirmrtn	MSG_CALLBACK	確認要求コールバックルーチンのアドレス、または NULL。

メンバー名	タイプ	説明
connectparms	char *	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とる。"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:¥SQLAny16¥bin32¥dbsrv16.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:¥SQLAny16¥bin32¥dbsrv16.exe".
connectparms_allocated	a_bit_field	予約済み。0 を使用します。
continue_download	a_bit_field	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。
create_dnld_file	const char *	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。
debug	a_bit_field	予約済み。0 を使用します。
debug_dump_char	a_bit_field	予約済み。0 を使用します。
debug_dump_hex	a_bit_field	予約済み。0 を使用します。
debug_dump_size	a_sql_uint32	予約済み。0 を使用します。

メンバー名	タイプ	説明
debug_page_offsets	a_bit_field	予約済み。0 を使用します。
default_window_title	char *	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)。
dl_insert_width	a_sql_uint32	予約済み。0 を使用します。
dl_use_put	a_bit_field	予約済み。0 を使用します。
dlg_info_msg	a_sql_uint32	予約済み。0 を使用します。
dnld_fail_len	a_sql_uint32	予約済み。0 を使用します。
dnld_file_extra	const char *	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。
dnld_gen_num	a_bit_field	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。
dnld_read_size	a_sql_uint32	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。
download_only	a_bit_field	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。
encrypted_stream_opts	const char *	予約済み。NULL を使用します。

メンバー名	タイプ	説明
encryption_key	char *	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。
entered_dialog	a_bit_field	予約済み。0 を使用します。
errorrtn	MSG_CALLBACK	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。
est_upld_row_cnt	a_sql_uint32	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。
extended_options	char *	「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。
hide_conn_str	a_bit_field	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。
hide_ml_pwd	a_bit_field	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。
hovering_frequency	a_sql_uint32	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。

メンバー名	タイプ	説明
ignore_debug_interrupt	a_bit_field	予約済み。0 を使用します。
ignore_hook_errors	a_bit_field	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。
ignore_hovering	a_bit_field	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。
ignore_scheduling	a_bit_field	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。
include_scan_range	const char *	予約済み。NULL を使用します。
init_cache	a_sql_uint32	キャッシュの初期サイズ。 dbmlsync -ci オプションと同等です。
init_cache_suffix	char	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)。
kill_other_connections	a_bit_field	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。
last_upload_def	a_syncpub *	予約済み。NULL を使用します。
lite_blob_handling	a_bit_field	予約済み。0 を使用します。

メンバー名	タイプ	説明
log_file_name	const char *	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。
log_size	a_sql_uint32	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。
logrtn	MSG_CALLBACK	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。
max_cache	a_sql_uint32	キャッシュの最大サイズ。 dbmlsync -cm オプションと同等です。
max_cache_suffix	char	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)。
min_cache	a_sql_uint32	キャッシュの最小サイズ。 dbmlsync -cl オプションと同等です。
min_cache_suffix	char	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)。
mlpassword	char *	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。

メンバー名	タイプ	説明
msgqueuertn	MSG_QUEUE_CALLBACK	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。
msgrtn	MSG_CALLBACK	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。
new_mlpassword	char *	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。

メンバー名	タイプ	説明
no_offline_logscan	a_sql_uint32	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。
no_schema_cache	a_bit_field	予約済み。0 を使用します。
no_stream_compress	a_bit_field	予約済み。0 を使用します。
offline_dir	const char *	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。
output_to_file	a_bit_field	予約済み。0 を使用します。
output_to_mobile_link	a_bit_field	予約済み。1 を使用します。
persist_connection	a_bit_field	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。
ping	a_bit_field	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。
preload_dlls	char *	予約済み。NULL を使用します。
progress_index_rtn	SET_PROGRESS_CALLBACK	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。

メンバー名	タイプ	説明
progress_msg_rtn	MSG_CALLBACK	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。
prompt_again	a_bit_field	予約済み。0 を使用します。
prompt_for_encrypt_key	a_bit_field	予約済み。0 を使用します。
protocol_add_cli_bit_to_cli_both	a_bit_field	予約済み。0 を使用します。
protocol_add_cli_bit_to_cli_max	a_bit_field	予約済み。0 を使用します。
protocol_add_serv_bit_to_cli_both	a_bit_field	予約済み。0 を使用します。
protocol_add_serv_bit_to_cli_max	a_bit_field	予約済み。0 を使用します。
protocol_add_serv_bit_to_serv_both	a_bit_field	予約済み。0 を使用します。
protocol_add_serv_bit_to_serv_max	a_bit_field	予約済み。0 を使用します。
raw_file	const char *	予約済み。NULL を使用します。
rename_log	a_bit_field	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。
reserved	a_bit_field	予約済み。0 を使用します。

メンバー名	タイプ	説明
retry_remote_ahead	a_bit_field	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。
retry_remote_behind	a_bit_field	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。
server_mode	a_bit_field	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。
server_port	a_sql_uint32	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。
set_window_title_rtn	SET_WINDOW_TITLE_CALLBACK	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)。
status_rtn	STATUS_CALLBACK	予約済み。NULL を使用します。
strictly_free_memory	a_bit_field	予約済み。0 を使用します。
strictly_ignore_trigger_ops	a_bit_field	予約済み。0 を使用します。

メンバー名	タイプ	説明
sync_opt	char *	予約済み。NULL を使用します。
sync_params	char *	ユーザ認証パラメータ。 dbmlsync -ap オプションと同等です。
sync_profile	char *	実行する同期プロファイル。 dbmlsync -sp オプションと同等です。
trans_upload	a_bit_field	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。
upld_fail_len	a_sql_uint32	予約済み。0 を使用します。
upload_defs	a_syncpub *	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。
upload_only	a_bit_field	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。
usage_rtn	USAGE_CALLBACK	予約済み。NULL を使用します。
use_fixed_cache	a_bit_field	予約済み。0 を使用します。
use_hex_offsets	a_bit_field	予約済み。0 を使用します。
use_relative_offsets	a_bit_field	予約済み。0 を使用します。
used_dialog_allocation	a_bit_field	予約済み。0 を使用します。

メンバー名	タイプ	説明
user_name	char *	同期する Mobile Link ユーザ (旧式)。 dbmlsync -u オプションと同等です。
verbose	a_bit_field	予約済み。0 を使用します。
verbose_download	a_bit_field	予約済み。0 を使用します。
verbose_download_data	a_bit_field	予約済み。0 を使用します。
verbose_hook	a_bit_field	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。
verbose_minimum	a_bit_field	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。
verbose_msgid	a_bit_field	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。
verbose_option_info	a_bit_field	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。
verbose_protocol	a_bit_field	予約済み。0 を使用します。
verbose_row_cnts	a_bit_field	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。

メンバー名	タイプ	説明
verbose_row_data	a_bit_field	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。
verbose_server	a_bit_field	予約済み。0 を使用します。
verbose_upload	a_bit_field	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。
verbose_upload_data	a_bit_field	予約済み。0 を使用します。
version	unsigned short	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。
warningrtn	MSG_CALLBACK	警告メッセージを表示するために呼び出す関数。

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。

参照

- DBSynchronizeLog メソッド [データベースツール]874 ページ
- 「dbmlsync 構文」『Mobile Link クライアント管理』
- 「dbmlsync の DBTools インタフェース」『Mobile Link クライアント管理』
- 「コールバック関数」863 ページ
- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』

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

メンバー名	タイプ	説明
ext_opt	char *	「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。
next	struct a_syncpub *	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。
pub_name	char *	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。
subscription	char *	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。

参照

- a_sync_db 構造体 [データベースツール]916 ページ
- DBSynchronizeLog メソッド [データベースツール]874 ページ
- 「dbmlsync の DBTools インタフェース」『Mobile Link クライアント管理』

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

メンバー名	タイプ	説明
blank_padding	a_bit_field	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。
case_sensitivity	a_bit_field	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。
default_collation	char	データベースの照合順。
encryption	a_bit_field	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。
page_size	unsigned short	データベースのページサイズ。
valid_data	a_bit_field	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。

参照

- a_db_info 構造体 [データベースツール]897 ページ
- DBInfo メソッド [データベースツール]871 ページ
- DBInfoDump メソッド [データベースツール]872 ページ
- DBInfoFree メソッド [データベースツール]872 ページ

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

メンバー名	タイプ	説明
index_pages	a_sql_uint32	インデックスページの数。
index_used	a_sql_uint32	インデックスページに使用されているバイト数。

メンバー名	タイプ	説明
index_used_pct	a_sql_uint32	インデックス領域の使用率を示すパーセンテージ。
next	struct a_table_info *	リスト内の次のテーブル。
table_id	a_sql_uint32	このテーブルの ID 番号。
table_name	char *	テーブルの名前。
table_pages	a_sql_uint32	テーブルページの数。
table_used	a_sql_uint32	テーブルページに使用されているバイト数。
table_used_pct	a_sql_uint32	テーブル領域の使用率を示すパーセンテージ。

参照

- a_db_info 構造体 [データベースツール]897 ページ
- DBInfo メソッド [データベースツール]871 ページ
- DBInfoDump メソッド [データベースツール]872 ページ
- DBInfoFree メソッド [データベースツール]872 ページ

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

メンバー名	タイプ	説明
ansi_sql	a_bit_field	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。
chronological_order	a_bit_field	予約済み。FALSE に設定します。

メンバー名	タイプ	説明
comment_trigger_trans	a_bit_field	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。
confirmrtn	MSG_CALLBACK	確認要求コールバックルーチンのアドレス、または NULL。
connectparms	const char *	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とる。"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:¥SQLAny16¥bin32¥dbsrv16.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:¥SQLAny16¥bin32¥dbsrv16.exe".
debug	a_bit_field	予約済み。FALSE に設定します。
debug_dump_char	a_bit_field	予約済み。FALSE に設定します。
debug_dump_hex	a_bit_field	予約済み。FALSE に設定します。
debug_dump_size	a_sql_uint32	予約済み。0 を使用します。
debug_page_offsets	a_bit_field	予約済み。FALSE に設定します。

メンバー名	タイプ	説明
debug_sql_remote	a_bit_field	予約済み。FALSE に設定します。
encryption_key	const char *	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。
errorrtn	MSG_CALLBACK	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。
extra_audit	a_bit_field	予約済み。FALSE に設定します。
force_chaining	a_bit_field	予約済み。FALSE に設定します。
force_recovery	a_bit_field	予約済み。FALSE に設定します。
generate_reciprocals	a_bit_field	予約済み。FALSE に設定します。
include_audit	a_bit_field	予約済み。FALSE に設定します。
include_destination_sets	const char *	予約済み。NULL を使用します。
include_publications	const char *	予約済み。NULL を使用します。
include_scan_range	const char *	予約済み。NULL を使用します。
include_source_sets	const char *	予約済み。NULL を使用します。
include_subsets	a_bit_field	予約済み。FALSE に設定します。
include_tables	const char *	予約済み。NULL を使用します。

メンバー名	タイプ	説明
include_trigger_trans	a_bit_field	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。
leave_output_on_error	a_bit_field	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。
logname	const char *	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。
logrtn	MSG_CALLBACK	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。
logs_dir	const char *	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。
match_mode	a_bit_field	予約済み。FALSE に設定します。
match_pos	const char *	予約済み。NULL を使用します。
msgrtn	MSG_CALLBACK	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。
omit_comments	a_bit_field	予約済み。FALSE に設定します。

メンバー名	タイプ	説明
queueparms	const char *	予約済み。NULL を使用します。
quiet	a_bit_field	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。
recovery_bytes	a_sql_uint32	予約済み。0 を使用します。
recovery_ops	a_sql_uint32	予約済み。0 を使用します。
remove_rollback	a_bit_field	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。
replace	a_bit_field	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。
repserver_users	const char *	予約済み。NULL を使用します。
show_undo	a_bit_field	予約済み。FALSE に設定します。
since_checkpoint	a_bit_field	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。

メンバー名	タイプ	説明
since_time	a_sql_uint32	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。
sqlname	const char *	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。
statusrtn	MSG_CALLBACK	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。
use_hex_offsets	a_bit_field	予約済み。FALSE に設定します。
use_relative_offsets	a_bit_field	予約済み。FALSE に設定します。
userlist	p_name	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。
userlisttype	char	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。

メンバー名	タイプ	説明
version	unsigned short	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。

参照

- a_name 構造体 [データベースツール]905 ページ
- ユーザリスト列挙 [データベースツール]882 ページ
- DBTranslateLog メソッド [データベースツール]876 ページ
- 「コールバック関数」 863 ページ
- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』

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

メンバー名	タイプ	説明
connectparms	const char *	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とる。"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:¥SQLAny16¥bin32¥dbsrv16.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:¥SQLAny16¥bin32¥dbsrv16.exe".
errorrtn	MSG_CALLBACK	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。
msgrtn	MSG_CALLBACK	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。
quiet	a_bit_field	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。

メンバー名	タイプ	説明
server_backup	a_bit_field	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。
truncate_interrupted	char	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。
version	unsigned short	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。

参照

- DBTruncateLog メソッド [データベースツール]876 ページ
- 「コールバック関数」863 ページ
- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』

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

メンバー名	タイプ	説明
connectparms	const char *	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とる。"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:¥SQLAny16¥bin32¥dbsrv16.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:¥SQLAny16¥bin32¥dbsrv16.exe".
errorrtn	MSG_CALLBACK	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。
index	a_bit_field	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。
msgrtn	MSG_CALLBACK	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。

メンバー名	タイプ	説明
quiet	a_bit_field	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。
statusrtn	MSG_CALLBACK	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。
tables	p_name	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。
type	char	実行する検証のタイプ。 VALIDATE_NORMAL、VALIDATE_EXPRESS、VALIDATE_CHECKSUM などのうちいずれかです。検証列挙を参照してください。
version	unsigned short	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。

参照

- a_name 構造体 [データベースツール]905 ページ
- DBValidate メソッド [データベースツール]878 ページ
- 検証列挙 [データベースツール]882 ページ
- 「コールバック関数」 863 ページ
- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』

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

メンバー名	タイプ	説明
confirmrtn	MSG_CALLBACK	確認要求コールバックルーチンのアドレス、または NULL。
dbname	const char *	データベースファイル名。
encryption_key	const char *	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。
erase	a_bit_field	消去するときに確認する (0) またはしない (1)。 dberase -y オプションによって TRUE に設定されます。
errorrtn	MSG_CALLBACK	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。
msgrtn	MSG_CALLBACK	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。
quiet	a_bit_field	操作中にメッセージを出力する (0) またはしない (1)。 dberase -q オプションによって TRUE に設定されます。
version	unsigned short	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。

参照

- DBErase メソッド [データベースツール]871 ページ
- 「コールバック関数」863 ページ

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

メンバー名	タイプ	説明
compress_output	a_bit_field	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。
confirmrtn	MSG_CALLBACK	確認要求コールバックルーチンのアドレス、または NULL。
connectparms	const char *	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とる。"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:¥SQLAny16¥bin32¥dbsrv16.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:¥SQLAny16¥bin32¥dbsrv16.exe".
debug	a_bit_field	予約済み。FALSE に設定します。

メンバー名	タイプ	説明
display_create	a_bit_field	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。
display_create_dbinit	a_bit_field	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。
encrypted_tables	a_bit_field	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。
encryption_algorithm	const char *	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。
encryption_key	const char *	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。
errorrtn	MSG_CALLBACK	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。

メンバー名	タイプ	説明
escape_char	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。
escape_char_present	a_bit_field	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。
exclude_foreign_keys	a_bit_field	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。
exclude_hooks	a_bit_field	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。
exclude_procedures	a_bit_field	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。
exclude_tables	a_bit_field	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。

メンバー名	タイプ	説明
exclude_triggers	a_bit_field	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。
exclude_views	a_bit_field	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。
extract	a_bit_field	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。
genscript	a_bit_field	予約済み。FALSE に設定します。
include_where_subscribe	a_bit_field	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。
isolation_level	unsigned short	操作の独立性レベル。 dbxtract -l オプションによって設定されます。
isolation_set	a_bit_field	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 -l オプションによって TRUE に設定されます。
locale	const char *	予約済み。NULL を使用します。

メンバー名	タイプ	説明
make_auxiliary	a_bit_field	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。
ms_filename	const char *	予約済み。NULL を使用します。
ms_reserve	int	予約済み。0 を使用します。
ms_size	int	予約済み。0 を使用します。
msgrtn	MSG_CALLBACK	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。
no_confirm	a_bit_field	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。
no_reload_status	a_bit_field	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。
notemp_size	long	予約済み。0 を使用します。
preserve_identity_values	a_bit_field	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。

メンバー名	タイプ	説明
preserve_ids	a_bit_field	TRUE に設定した場合、ユーザ ID を保持します。 これは通常の設定です。dbunload -e オプションによって FALSE に設定されます。
profiling_uses_single_dbspace	a_bit_field	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。
recompute	a_bit_field	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。
refresh_mat_view	a_bit_field	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。
reload_connectparms	char *	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。
reload_db_filename	char *	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。

メンバー名	タイプ	説明
reload_db_logname	char *	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。
reload_filename	const char *	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。
reload_page_size	unsigned short	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。
remote_dir	const char *	temp_dir に似ていますが、サーバ側の内部アンロード用です。
remove_encrypted_tables	a_bit_field	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。
replace_db	a_bit_field	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。
runscript	a_bit_field	予約済み。FALSE に設定します。
schema_reload	a_bit_field	予約済み。FALSE に設定します。

メンバー名	タイプ	説明
site_name	const char *	dbxtract で使用されるサイト名。それ以外の場合は NULL。
start_subscriptions	a_bit_field	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。
startline	const char *	予約済み。NULL を使用します。
startline_name	a_bit_field	予約済み。FALSE に設定します。
startline_old	const char *	予約済み。NULL を使用します。
statusrtn	MSG_CALLBACK	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。
subscriber_username	const char *	dbxtract で使用されるサブスクライバ名。それ以外の場合は NULL。
suppress_statistics	a_bit_field	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。
sysinfo	a_sysinfo	予約済み。NULL を使用します。
table_list	p_name	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。

メンバー名	タイプ	説明
table_list_provided	a_bit_field	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 または -t オプションによって TRUE に設定されます。
temp_dir	const char *	データファイルのアンロード用ディレクトリ。
template_name	const char *	dbxtract で使用されるテンプレート名。それ以外の場合は NULL。
unload_interrupted	char	予約済み。0 に設定します。
unload_type	char	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。
unordered	a_bit_field	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。
use_internal_reload	a_bit_field	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。

メンバー名	タイプ	説明
use_internal_unload	a_bit_field	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。
verbose	char	冗長列挙 (VB_QUIET、VB_NORMAL、VB_VERBOSE) を参照してください。
version	unsigned short	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。

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。

参照

- 冗長列挙 [データベースツール]883 ページ
- DBUnload メソッド [データベースツール]877 ページ
- 「コールバック関数」863 ページ
- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』

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

メンバー名	タイプ	説明
connectparms	const char *	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とる。"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:¥SQLAny16¥bin32¥dbsrv16.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:¥SQLAny16¥bin32¥dbsrv16.exe".
errorrtn	MSG_CALLBACK	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。
jconnect	a_bit_field	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。
msgrtn	MSG_CALLBACK	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。
quiet	a_bit_field	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。

メンバー名	タイプ	説明
restart	a_bit_field	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。
statusrtn	MSG_CALLBACK	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。
sys_proc_definer	unsigned short	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)。
version	unsigned short	DBTools のバージョン番号 (DB_TOOLS_VERSION_NUMBER)。

参照

- DBUpgrade メソッド [データベースツール]877 ページ
- 「コールバック関数」863 ページ
- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』

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で次の終了コードが使用されます。SQL Anywhere のユーティリティ (dbbackup、dbspawn、dbsrv16 など) でもこれらの終了コードを使用します。

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	メモリがない
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

コード	ステータス	説明
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
255	EXIT_USAGE	コマンドラインで無効なパラメータ
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。

これらの終了コードは、*%SQLANY16%¥sdk¥include¥sqldef.h* ファイルに定義されています。

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。

注意
ファイルの再配布は Sybase との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。

この項では、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すすめします。詳細については、「Deployment ウィザード」964 ページと「SQL Anywhere インストーラを使用したサイレントインストール」968 ページを参照してください。

配備の種類

配備する必要があるファイルは、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 アプリケーションの配備」『Mobile Link サーバ管理』を参照してください。

- **管理ツールの配備**　Interactive SQL、Sybase Central、その他の管理ツールを配備します。

ファイルの配布方法

SQL Anywhere を配備するには、次の 2 つの方法があります。

- **SQL Anywhere インストーラを使用する**　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。

  詳細については、「Deployment ウィザード」964 ページまたは「SQL Anywhere インストーラを使用したサイレントインストール」968 ページを参照してください。

- **独自のインストール環境を開発する**　SQL Anywhere ファイルを組み込んだ独自のインストールプログラム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、この項の大半で独自のインストール環境を作成するユーザの必要性について取り上げます。

  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、インストールディレクトリの *bin32* サブフォルダには、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。

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が必要とするファイルは単一のディレクトリ、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* サブフォルダにインストールされます。

ソフトウェアがファイルを探す場所の詳細については、「SQL Anywhere のファイル検索方法」『SQL Anywhere サーバ データベース管理』を参照してください。

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	内容
/opt/sqlanywhere16/bin32 と */opt/sqlanywhere16/bin64*	実行ファイル、ライセンスファイル
/opt/sqlanywhere16/lib32 と */opt/sqlanywhere16/lib64*	共有オブジェクトと共有ライブラリ
/opt/sqlanywhere16/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/sqlanywhere16* であり、*/opt/sqlanywhere16* 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/SQLAnywhere16/System* であり、*/opt/sqlanywhere16* 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**　この項の表では、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、*libdblib16.so* は同じディレクトリ内のファイル *libdblib16.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する必要があります。

- **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文字 *_r* が追加された形式になります。たとえば、*libdblib16.so.1* の他に *libdblib16_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にサフィックス *_r* が付く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にサフィックス *_r* が付かない共有オブジェクトにリンクされる必要があります。さらに、ファイルのサフィックスの前に *_n* が付いた共有オブジェクトの第 3 の形式も存在する場合があります。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。

  - *libdbicu16.so.1*
  - *libdbicu16_r.so.1*
  - *libdbicudt16.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できるように、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* と *sa_config.csh* (*/opt/sqlanywhere16/bin32* と */opt/sqlanywhere16/bin64* ディレクトリ内) のいずれかのうち、シェルに適したファイルを使用することをおすすめします。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、SQLANY16 などがあります。

SQL Anywhere がファイルを探す場所の詳細については、「SQL Anywhere のファイル検索方法」『SQL Anywhere サーバ データベース管理』を参照してください。

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 *dbsrv16.exe* は Windows 用のバージョン 16 の実行プログラムです。

- **Language**　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、*dblgen16.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)」『SQL Anywhere サーバ データベース管理』を参照してください。

  SQL Anywhere で使用できる言語リストについては、「SQL Anywhere のローカライズ版」『SQL Anywhere サーバ データベース管理』を参照してください。

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、このファイルの拡張子には *.db* が使用されます。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。

Deployment ウィザード

SQL Anywhere の **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、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できます。

●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 Sybase 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。

Deployment ウィザードを実行するには、**[スタート]** » **[プログラム]** » **[SQL Anywhere 16]** » **[管理ツール]** » **[Windows に展開]** をクリックします。

配備パッケージの作成に使用した製品コードを必ず控えておいてください。後でパッケージをアンインストールする場合にこの製品コードを使用できます。

注意
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。

参照

- 「Deployment ウィザードの実行」965 ページ
- 「配備パッケージインストールユーティリティ」966 ページ
- 「SQL Anywhere インストーラを使用したサイレントインストール」968 ページ

Deployment ウィザードの実行

SQL Anywhere **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

Windows オペレーティングシステム

内容と備考

データベースおよびアプリケーションの配備

♦ タスク

1. **Deployment ウィザード**を実行するには、**[スタート]** » **[プログラム]** » **[SQL Anywhere 16]** » **[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. 配備パッケージの作成に使用した製品コードを控えておいてください。製品コードは、たとえば {7C99D24E-AE1B-4770-9015-65B805950E3D} のようになっています。後でパッケージをアンインストールする場合にこの製品コードを使用できます。

結果

選択した **[インストール先パス]** には、配備パッケージ (*sqlany16.msi* など) とログファイル (*sqlany16.msi.log* など) が格納されます。

次の手順

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。

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。例：

```
%SQLANY16%¥Deployment¥DeploymentWizard.exe
```

参照

- 「Deployment ウィザード」964 ページ
- 「配備パッケージインストールユーティリティ」966 ページ

配備パッケージインストールユーティリティ

WIndows で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 (*sqlany16.log* など) に記録します。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 (*sqlany16.msi* など) をインストールします。
/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。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

オプション	説明
/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 Vista 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。

参照

- 「Deployment ウィザード」964 ページ
- 「Deployment ウィザードの実行」965 ページ

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 sqlany16.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 sqlany16.msi /log sqlany16.log SQLANYDIR=c:¥sa16
```

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 sqlany16.msi /log sqlany16.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}
```

SQL Anywhere インストーラを使用した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。

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=QEDEV-B888A-6L123-45678-90123 INSTALLDIR=c:¥sa16 DIR_SAMPLES=c:
¥sa16¥Samples"
```

注意
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。

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=QEDEV-B888A-6L123-45678-90123
REGKEY_ADD_1=<an add-on software registration key>
REGKEY_ADD_2=<another add-on software registration key>
INSTALLDIR=c:¥sa16
DIR_SAMPLES=c:¥sa16¥Samples
DIR_SQLANY_MONITOR=c:¥sa16¥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 Monitor インストーラ (d:¥pkg¥Monitor¥setup.exe にある) 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=<SQLAnywhere Monitor registration key>
```

```
REGKEY_ADD_1=<an add-on software registration key>
REGKEY_ADD_2=<another add-on software registration key>
INSTALLDIR=c:¥sa16"
```

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
CAC 認証	CAC *		
FIPS 認定の暗号化	FIPS *		
高可用性	HA *		
インメモリモード	IM *		
Mobile Link (32 ビット)	ML32	1	0
Mobile Link (64 ビット)	ML64	0	1
読み込み専用のスケールアウト	SON *		
Relay Server (32 ビット)	RS32	1	0
Relay Server (64 ビット)	RS64	0	1
サンプル	SAMPLES	1	1
SQL Anywhere (32 ビット)	SA32	1	0
SQL Anywhere (64 ビット)	SA64	0	1
SQL Anywhere for Windows Mobile	MOBILE	1	1

機能	プロパティ	x86 のデフォルト	x64 のデフォルト
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 ビットの Mobile Link を省略するには、ML32=0 に設定します。32 ビットの管理ツールを選択するには、AT32=1 に設定します。デフォルトの選択を上書きするには、これらのプロパティを使用します。

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。ログファイルへの出力を減らす方法については、http://msdn.microsoft.com/ja-jp/library/aa367988.aspx を参照してください。

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 {3D7AB509-4FC2-4570-9913-A8F1CAD8C743} /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 16.0 の製品コードは次のとおりです。

- **{3D7AB509-4FC2-4570-9913-A8F1CAD8C743}**　SQL Anywhere ソフトウェア
- **{37B83CDB-F75E-4428-A9C2-995D92105EEC}**　SQL Anywhere クライアント専用ソフトウェア
- **{1AF08ED0-804C-4AFF-BB6C-D980D81AD594}**　SQL Anywhere モニタ

クライアントアプリケーションを配備するための要件

ネットワークデータベースサーバに対して実行されるクライアントアプリケーションを配備するには、次に示すものを各エンドユーザに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。
- **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、それが使用するデータベースインタフェース (.NET、ADO、OLE DB、ODBC、JDBC、Embedded SQL、または Open Client) 用のファイルが必要です。
- **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。各インタフェースについては、次の項で個別に説明します。

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。詳細については、「Deployment ウィザード」964 ページを参照してください。

.NET クライアントの配備

.NET アセンブ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。詳細については、「Deployment ウィザード」964 ページを参照してください。

クライアントコンピュータへの .NET アセンブリのインストールを完了するには、次を実行する必要があります。

- 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 .NET アセンブリをインストールします。Windows Vista 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を実行する場合は、必ず管理者権限で実行してください。

●.NET 2.0/3.x の場合、*%SQLANY16%¥Assembly¥v2¥SetupVSPackage.exe /install* を実行します。

●.NET 4.x の場合、*%SQLANY16%¥Assembly¥v4¥SetupVSPackage.exe /install* を実行します。

●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。

```
%SQLANY16%¥Assembly¥v2¥SetupVSPackage.exe /install /salocation %SQLANY16%
```

salocation の省略形は **sal** です。

SetupVSPackage アプリケーションは、グローバルアセンブリキャッシュと Windows Microsoft.NET の *machine.config* ファイルを更新します。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 SQL Server *DTS¥MappingFiles* フォルダにインストールします。

SetupVSPackage アプリケーションで、*SSDLToSA16.tt* を Visual Studio 2010 ディレクトリにコピーします。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環境を構築するために、この項ではエンドユーザに配備するファイルについて説明します。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**、**jp** 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。

● **正常に機能する .NET 2.0、3.0、3.5、4.0、または 4.5 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 の SQL Anywhere プロバイダ**　SQL Anywhere のインストールでは、.NET Framework バージョン 2.0 および 3.0 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¥V2* 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。

```
iAnywhere.Data.SQLAnywhere.dll
policy.16.0.iAnywhere.Data.SQLAnywhere.dll
dblg[LL]16.dll
dbicu16.dll
dbicudt16.dll
```

● **.NET Framework 3.5 の SQL Anywhere プロバイダ**　SQL Anywhere のインストールでは、.NET Framework バージョン 3.5 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¥V3.5* 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。

iAnywhere.Data.SQLAnywhere.v3.5.dll
policy.16.0.iAnywhere.Data.SQLAnywhere.v3.5.dll
dblg[LL]16.dll
dbicu16.dll
dbicudt16.dll

- **.NET Framework 4.0/4.5 の SQL Anywhere プロバイダ**　SQL Anywhere のインストールでは、.NET Framework バージョン 4.0 および 4.5 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* 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。

  iAnywhere.Data.SQLAnywhere.v4.0.dll
  policy.16.0.iAnywhere.Data.SQLAnywhere.v4.0.dll
  dblg[LL]16.dll
  dbicu16.dll
  dbicudt16.dll

- **.NET Compact Framework 2.0 の SQL Anywhere プロバイダ**　SQL Anywhere のインストールでは、.NET Compact Framework バージョン 2.0 用の Windows Mobile アセンブリは *CE¥Assembly¥V2* 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 Windows Mobile バイナリサブフォルダに置かれます (例：*CE¥Arm.50*)。次のファイルは必須です。

  iAnywhere.Data.SQLAnywhere.dll
  iAnywhere.Data.SQLAnywhere.gac
  dblg[LL]16.dll
  dbicu16.dll

- **.NET Compact Framework 3.5 の SQL Anywhere プロバイダ**　SQL Anywhere のインストールでは、.NET Compact Framework バージョン 3.5 用の Windows Mobile アセンブリは *CE¥Assembly¥V3.5* 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 Windows Mobile バイナリサブフォルダに置かれます (例：*CE¥Arm.50*)。次のファイルは必須です。

  iAnywhere.Data.SQLAnywhere.v3.5.dll
  dblg[LL]16.dll
  dbicu16.dll

SQL Anywhere .NET データプロバイダの配備の詳細については、「SQL Anywhere .NET データプロバイダの配備」72 ページと「ClickOnce と .NET データプロバイダのアンマネージコード DLL」974 ページを参照してください。

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、配備する必要がある .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、*%SQLANYSAMP16%¥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、*dblgen16.dll*、*dbicu16.dll*、*dbicudt16.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。

SQL Anywhere .NET データプロバイダの削除

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、次の作業を実行します。

- 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 .NET アセンブリをアンインストールします。Windows Vista 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を実行する場合は、必ず管理者権限で実行してください。

- .NET 2.0/3.x の場合、*%SQLANY16%¥Assembly¥v2¥SetupVSPackage.exe /uninstall* を実行します。

- .NET 4.x の場合、*%SQLANY16%¥Assembly¥v4¥SetupVSPackage.exe /uninstall* を実行します。

/uninstall の省略形は **/u** です。

16.0 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。

- 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 .NET 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 Vista 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を実行する場合は、必ず管理者権限で実行してください。

- .NET 2.0/3.x の場合、*%SQLANY16%¥Assembly¥v2¥SetupVSPackage.exe /uninstallall* を実行します。

- .NET 4.x の場合、*%SQLANY16%¥Assembly¥v4¥SetupVSPackage.exe /uninstallall* を実行します。

/uninstallall の省略形は **/ua** です。

OLE DB と ADO クライアントの開発

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環境を構築するために、この項ではエンドユーザに配備するファイルについて説明します。

各 OLE DB クライアントコンピュータには、次のものが必要です。

- **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。ここでは、その詳細は説明しません。

- **SQL Anywhere OLE DB プロバイダ**　次の表には、SQL Anywhere OLE DB が動作するプロバイダに必要なファイルを示しています。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オペレーティングシステムに対応するサブフォルダに置かれます (例：*bin32*、*bin64*)。Windows の場合、プロバイダ DLL が 2 つあります。2 つ目の DLL (*dboledba16.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 ドライバファイル	*dboledb16.dll*
OLE DB ドライバファイル	*dboledba16.dll*
言語リソースライブラリ	*dblg[LL]16.dll*
[接続] ウィンドウ	*dbcon16.dll*
昇格操作エージェント	*dbelevate16.exe* (Windows Vista 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**、**jp** 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 *dboledb16.dll* および *dboledba16.dll* の各 DLL を自己登録します。

Windows Vista 以降のバージョンの Windows 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。

Windows クライアントには、Microsoft MDAC 2.7 以降を使用することをおすすめします。

OLE DB プロバイダのカスタマイズ

OLE DB プロバイダをインストールする場合は、Windows レジストリの変更が必要です。通常、変更には OLE DB プロバイダに組み込まれている自己登録機能を使用します。たとえば、Windows regsvr32 ツールを使用することができます。レジストリエントリの標準セットは、プロバイダによって作成されます。

一般的な接続文字列では、コンポーネントの 1 つは Provider 属性になります。SQL Anywhere 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 Visual Basic の場合の例を示します。

```
connectString = "Provider=SAOLEDB;DSN=SQL Anywhere 16 Demo"
```

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数多く存在します。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.16"), &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.16"
```

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:¥¥sa16¥¥bin64¥¥dboledb16.dll"
"ThreadingModel"="Both"
```

ここで問題になるのは、これが融通の利かない構造であることです。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、さまざまなベンダー製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。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置き換わる可能性があります。

このような状況によって不安定な状態が生じるのは望ましくありません。この問題に対応するため、SQL Anywhere OLE DB プロバイダはカスタマイズが可能になっています。

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。

前提条件

この作業を実行するための前提条件はありません。

♦ タスク

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¥myoledb16]
@="Custom SQL Anywhere OLE DB Provider 16.0"
[HKEY_CLASSES_ROOT¥myoledb16¥Clsid] @="{GUID1}"
; Data1 of the following GUID must be 3 greater than the
; previous, for example, 41dfe9f3 + 3 => 41dfe9ee.
[HKEY_CLASSES_ROOT¥myoledba16]
@="Custom SQL Anywhere OLE DB Provider 16.0"
[HKEY_CLASSES_ROOT¥myoledba16¥Clsid] @="{GUID4}"
; Current version (or version independent prog ID)
; entries (what you get when you have "SQLAny"
; instead of "SQLAny.16")
[HKEY_CLASSES_ROOT¥SQLAny]
@="SQL Anywhere OLE DB Provider"
[HKEY_CLASSES_ROOT¥SQLAny¥Clsid]
@="{GUID1}"
[HKEY_CLASSES_ROOT¥SQLAny¥CurVer]
@="SQLAny.16"
[HKEY_CLASSES_ROOT¥SQLAnyEnum]
@="SQL Anywhere OLE DB Provider Enumerator"
[HKEY_CLASSES_ROOT¥SQLAnyEnum¥Clsid]
@="{GUID2}" [HKEY_CLASSES_ROOT¥SQLAnyEnum¥CurVer]
@="SQLAnyEnum.16" [HKEY_CLASSES_ROOT¥SQLAnyErrorLookup]
@="SQL Anywhere OLE DB Provider Extended Error Support"
[HKEY_CLASSES_ROOT¥SQLAnyErrorLookup¥Clsid]
@="{GUID3}"
[HKEY_CLASSES_ROOT¥SQLAnyErrorLookup¥CurVer]
@="SQLAnyErrorLookup.16"
[HKEY_CLASSES_ROOT¥SQLAnyA]
@="SQL Anywhere OLE DB Provider Assist"
[HKEY_CLASSES_ROOT¥SQLAnyA¥Clsid]
@="{GUID4}"
[HKEY_CLASSES_ROOT¥SQLAnyA¥CurVer]
@="SQLAnyA.16"
; Standard entries (Provider=SQLAny.16)
[HKEY_CLASSES_ROOT¥SQLAny.16]
@="Sybase SQL Anywhere OLE DB Provider 16.0"
[HKEY_CLASSES_ROOT¥SQLAny.16¥Clsid]
@="{GUID1}"
[HKEY_CLASSES_ROOT¥SQLAnyEnum.16]
@="Sybase SQL Anywhere OLE DB Provider Enumerator 16.0"
[HKEY_CLASSES_ROOT¥SQLAnyEnum.16¥Clsid]
```

```
@="{GUID2}"
[HKEY_CLASSES_ROOT¥SQLAnyErrorLookup.16]
@="Sybase SQL Anywhere OLE DB Provider Extended Error Support 16.0"
[HKEY_CLASSES_ROOT¥SQLAnyErrorLookup.16¥Clsid]
@="{GUID3}"
[HKEY_CLASSES_ROOT¥SQLAnyA.16]
@="Sybase SQL Anywhere OLE DB Provider Assist 16.0"
[HKEY_CLASSES_ROOT¥SQLAnyA.16¥Clsid]
@="{GUID4}"
; SQLAny (Provider=SQLAny.16)
[HKEY_CLASSES_ROOT¥CLSID¥{GUID1}]
@="SQLAny.16"
"OLEDB_SERVICES"=dword:ffffffff
[HKEY_CLASSES_ROOT¥CLSID¥{GUID1}¥ExtendedErrors]
@="Extended Error Service"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID1}¥ExtendedErrors¥{GUID3}]
@="Sybase SQL Anywhere OLE DB Provider Error Lookup"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID1}¥InprocServer32]
@="d:¥¥mypath¥¥bin32¥¥myoledb16.dll"
"ThreadingModel"="Both"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID1}¥OLE DB Provider]
@="Sybase SQL Anywhere OLE DB Provider 16.0"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID1}¥ProgID]
@="SQLAny.16"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID1}¥VersionIndependentProgID]
@="SQLAny"
; SQLAnyErrorLookup
[HKEY_CLASSES_ROOT¥CLSID¥{GUID3}]
@="Sybase SQL Anywhere OLE DB Provider Error Lookup 16.0"
@="SQLAnyErrorLookup.16"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID3}¥InprocServer32]
@="d:¥¥mypath¥¥bin32¥¥myoledb16.dll"
"ThreadingModel"="Both"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID3}¥ProgID]
@="SQLAnyErrorLookup.16"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID3}¥VersionIndependentProgID]
@="SQLAnyErrorLookup"
; SQLAnyEnum [HKEY_CLASSES_ROOT¥CLSID¥{GUID2}]
@="SQLAnyEnum.16"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID2}¥InprocServer32]
@="d:¥¥mypath¥¥bin32¥¥myoledb16.dll"
"ThreadingModel"="Both"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID2}¥OLE DB Enumerator]
@="Sybase SQL Anywhere OLE DB Provider Enumerator"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID2}¥ProgID]
@="SQLAnyEnum.16"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID2}¥VersionIndependentProgID]
@="SQLAnyEnum"
; SQLAnyA [HKEY_CLASSES_ROOT¥CLSID¥{GUID4}]
@="SQLAnyA.16"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID4}¥InprocServer32]
@="d:¥¥mypath¥¥bin32¥¥myoledba16.dll"
"ThreadingModel"="Both"
[HKEY_CLASSES_ROOT¥CLSID¥{GUID4}¥ProgID]
@="SQLAnyA.16"
[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 >oledbs.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)。最後の GUID は、Windows Mobile の場合の配備では使用されません。

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。

Name	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

注意
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 プロバイダ	カスタムプロバイダ
SAOLEDB	Acme
SAErrorLookup	AcmeErrorLookup
SAEnum	AcmeEnum
SAOLEDBA	AcmeA

6. SQL Anywhere プロバイダ DLL (*dboledb16.dll* と *dboledba16.dll*) のコピーを別の名前で作成します。

```
copy dboledb16.dll myoledb16.dll
copy dboledba16.dll myoledba16.dll
```

   Windows Mobile の場合、*dboledba16.dll* はありません。

   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、標準の DLL 名 (*dboledb16.dll*、*dboledba16.dll* など) と名前が異なっていることが重要です。プロバイダ DLL 名を *myoledb16* 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 *myoledba16* 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
myoledb16.dll	HKEY_CLASSES_ROOT¥myoledb16
myoledba16.dll	HKEY_CLASSES_ROOT¥myoledba16
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 *myoledb16* と *myoledba16* 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すべての *d:¥¥mypath¥¥bin32¥¥* を DLL のインストールロケーションに変更します。1 つのスラッシュを表すのにスラッシュを 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。

次の手順

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。

ODBC クライアントの配備

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。Windows Mobile 用の ODBC ドライバマネージャはありません。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すすめできません。

- **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。

参照

- 「Deployment ウィザード」964 ページ
- 「ODBC ドライバに必要なファイル」982 ページ

ODBC ドライバに必要なファイル

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。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オペレーティングシステムに対応するサブフォルダに置かれます (例：*bin32*、*bin64*)。

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	*dbodbc16.dll* *dbcon16.dll* *sacshelp16.chm* (オプション) *dbicu16.dll* *dbicudt16.dll* *dblg[LL]16.dll* *dbelevate16.exe*
Windows Mobile	*dbodbc16.dll* *dbicu16.dll* (オプション) *dbicudt16.dat* (オプション) *dblg[LL]16.dll*
Linux、Solaris、HP-UX	*libdbodbc16.so.1* *libdbodbc16_n.so.1* *libdbodm16.so.1* *libdbtasks16.so.1* *libdbicu16.so.1* *libdbicudt16.so.1* *dblg[LL]16.res*

プラットフォーム	必要なファイル
Linux、Solaris、HP-UX MT	*libdbodbc16.so.1* *libdbodbc16_r.so.1* *libdbodm16.so.1* *libdbtasks16_r.so.1* *libdbicu16_r.so.1* *libdbicudt16.so.1* *dblg[LL]16.res*
AIX	*libdbodbc16.so* *libdbodbc16_n.so* *libdbodm16.so* *libdbtasks16.so* *libdbicu16.so* *libdbicudt16.so* *dblg[LL]16.res*
AIX MT	*libdbodbc16.so* *libdbodbc16_r.so* *libdbodm16.so* *libdbtasks16_r.so* *libdbicu16_r.so* *libdbicudt16.so* *dblg[LL]16.res*

プラットフォーム	必要なファイル
Mac OS X	*dbodbc16.bundle* *libdbodbc16.dylib* *libdbodbc16_n.dylib* *libdbodm16.dylib* *libdbtasks16.dylib* *libdbicu16.dylib* *libdbicudt16.dylib* *dblg[LL]16.res*
Mac OS X MT	*dbodbc16_r.bundle* *libdbodbc16.dylib* *libdbodbc16_r.dylib* *libdbodm16.dylib* *libdbtasks16_r.dylib* *libdbicu16_r.dylib* *libdbicudt16.dylib* *dblg[LL]16.res*

注意

- 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する必要があります。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。

- 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 *libdbodm16* で始まる名前が付いています。

- Windows の場合、*sacshelp16.chm* ヘルプファイルをインストールして、Microsoft ODBC データソースアドミニストレータを使用するときに、ローカルのコンテキスト別ヘルプを提供できま

す。ヘルプファイルが見つからない場合には、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 への移動が試行されます。

- Windows Vista 以降のバージョンの Windows 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 (*dbelevate16.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追加してください。**[LL]** を言語コード (**en**、**de**、**jp** 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] ウィンドウを表示する必要がある場合は、**[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 (*dbcon16.dll*) が必要です。

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。Windows Mobile 用の ODBC ドライバでは登録が不要です。

32 ビットバージョンおよび 64 ビットバージョンの SQL Anywhere ODBC ドライバにカスタム名を付けてください。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 "%SQLANY16%¥bin32¥dbodbc16.dll" dbodbc16custom-name.dll
ren "%SQLANY16%¥bin64¥dbodbc16.dll" dbodbc16custom-name.dll
```

ファイル名を変更するときは、必ず dbodbc16 プレフィクスを維持してください。

カスタム名を使用して 32 ビットバージョンおよび 64 ビットバージョンの ODBC ドライバを登録するには、次のコマンドを発行します。

```
regsvr32 "%SQLANY16%¥bin32¥dbodbc16custom-name.dll"
regsvr32 "%SQLANY16%¥bin64¥dbodbc16custom-name.dll"
```

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。

SQL Anywhere ODBC ドライバは、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 16
```

32 ビット Windows の場合のサンプル値を次に示します。

値の名前	値のタイプ	値データ
Driver	文字列	*C:¥Program Files¥SQL Anywhere 16¥bin32¥dbodbc16custom-name.dll*
Setup	文字列	*C:¥Program Files¥SQL Anywhere 16¥bin32¥dbodbc16custom-name.dll*

次のキーにもレジストリ値があります。

```
HKEY_LOCAL_MACHINE¥
  SOFTWARE¥
    ODBC¥
      ODBCINST.INI¥
        ODBC Drivers
```

値は次のとおりです。

値の名前	値のタイプ	値データ
SQL Anywhere 16 - *custom-name*	文字列	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 Anywhere 16 - *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、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。

接続情報の配備

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。

- **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 では、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手動で作成します。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、*sacshelp16.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、SQL Anywhere の 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、起動パラメータとその他のオプションが含まれます。

この項では、どちらの方法であっても知る必要がある情報について説明します。

データソースのタイプ (Windows)

データソースには 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のロケーションは、**HKEY_CURRENT_USER ¥Software¥ODBC¥odbc.ini¥ODBC File DSN¥DefaultDSNDir** レジストリエントリまたは **HKEY_LOCAL_MACHINE¥Software¥ODBC¥odbc.ini¥ODBC File DSN¥DefaultDSNDir** レジストリエントリ (フォーマットが定義されていない場合) で指定されます。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。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
```

64 ビット Windows では、dbdsn の 64 ビットバージョンを使用して 64 ビットレジストリキーを作成します。

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
```

64 ビット Windows では、dbdsn の 32 ビットバージョンを使用して 32 ビットレジストリキーを作成します。

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 16 Demo システムのデータソース名 (DSN) に対応する SQL Anywhere 16 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	文字列	YES
DatabaseFile	文字列	*C:¥Users¥Public¥Documents¥SQL Anywhere 16¥Samples¥demo.db*
Description	文字列	SQL Anywhere 16 サンプルデータベース
Driver	文字列	*C:¥Program Files¥SQL Anywhere 16¥bin32¥dbodbc16.dll*
PWD	文字列	sql
ServerName	文字列	demo16
StartLine	文字列	*C:¥Program Files¥SQL Anywhere 16¥bin32¥dbeng16.exe*
UID	文字列	DBA

注意
配備されたアプリケーションの接続文字列には、ServerName パラメータを含めることをおすすめします。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。

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
```

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
```

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 16 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 16 Demo	文字列	SQL Anywhere 16

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、レジストリにプレーン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)

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)

参照

- 「データベース接続」『SQL Anywhere サーバ データベース管理』
- 『ODBC SDK』(Microsoft から入手可能)

Embedded SQL クライアントの配備

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、SQL Anywhere の 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。

参照

- 「Deployment ウィザード」964 ページ

Embedded SQL クライアントに必要なファイル

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	*dblib16.dll*	*libdblib16_r.so*	*libdblib16_r.dylib*
オプションの暗号化サポート	*dbfips16.dll*	*libdbfips16_r.so*	*libdbfips16_r.dylib*
オプションの暗号化サポート	*dbrsa16.dll*	*libdbrsa16_r.so*	*libdbrsa16_r.dylib*
スレッドサポートライブラリ	なし	*libdbtasks16_r.so*	*libdbtasks16_r.dylib*
言語リソースライブラリ	*dblg[LL]16.dll*	*dblg[LL]16.res*	*dblg[LL]16.res*
[接続] ウィンドウ	*dbcon16.dll*	なし	なし

説明

- 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**、**jp** 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、*libdblib16.so* と *libdbtasks16.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、*libdblib16.dylib* と *libdbtasks16.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 (*dbfips16.dll* または *dbrsa16.dll*) 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] ウィンドウを表示する必要がある場合は、**[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 (*dbcon16.dll*) が必要です。

参照

- 「ODBC クライアントの配備」982 ページ

接続情報

Embedded SQL 接続情報は、次のいずれかの方法で配備できます。

- **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。

JDBC クライアントの配備

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 1.7.0 以降を推奨します。

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 が必要です。

SQL Anywhere JDBC 4.0 ドライバ

SQL Anywhere JDBC 4.0 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オペレーティングシステムに対応するサブフォルダに置かれます (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	*dbjdbc16.dll* *dbicu16.dll* *dbicudt16.dll* *dblg[LL]16.dll*

プラットフォーム	必要なファイル
Linux、Solaris、HP-UX	*libdbjdbc16.so.1* *libdbtasks16_r.so.1* *libdbicu16_r.so.1* *libdbicudt16.so.1* *dblg[LL]16.res*
AIX	*libdbjdbc16.so* *libdbtasks16_r.so* *libdbicu16_r.so* *libdbicudt16.so* *dblg[LL]16.res*
Mac OS X	*libdbjdbc16.dylib* *libdbtasks16_r.dylib* *libdbicu16_r.dylib* *libdbicudt16.dylib* *dblg[LL]16.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する必要があります。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**、**jp** 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。jConnect ソフトウェアのバージョンと jConnect のマニュアルについては、http://www.sybase.com/products/allproductsa-z/softwaredeveloperkit/jconnect を参照してください。

- 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 *dbrsakp16.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 *libdbrsakp16.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 *libdbrsakp16.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。

参照

- 「jConnect ドライバ接続文字列」430 ページ

PHP クライアントの配備

SQL Anywhere 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://www.php.net からダウンロードできます。Windows プラットフォームの場合、SQL Anywhere 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)。

  SQL Anywhere 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。
- SQL Anywhere には、5.1.1 ～ 5.4.8 の多くの PHP バージョン用の構築済み PHP 拡張機能が用意されています。
- SQL Anywhere 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/UNIX	Mac OS X
PHP のインストール (サードパーティ)	*php.exe*	*php*	*php*

説明	Windows	Linux/UNIX	Mac OS X
PHP 5.1.x の呼び出し	*php-5.1.[1-*]_sqlanywhere.dll*	*php-5.1.[1-*]_sqlanywhere_r.so* またはソースコードからビルド	ソースコードからビルド
PHP 5.2.x の呼び出し	*php-5.2.[0-*]_sqlanywhere.dll* *php-5.2.1_sqlanywhere_nts.dll*	*php-5.2.[0-*]_sqlanywhere_r.so* またはソースコードからビルド	ソースコードからビルド
PHP 5.3.x の呼び出し	*php-5.3.[0-*]_sqlanywhere.dll* *php-5.3.[0-*]_sqlanywhere_nts.dll*	*php-5.3.[0-*]_sqlanywhere_r.so* またはソースコードからビルド	ソースコードからビルド
PHP 5.4.x の呼び出し	*php-5.4.[0-*]_sqlanywhere.dll* *php-5.4.[0-*]_sqlanywhere_nts.dll*	*php-5.4.[0-*]_sqlanywhere_r.so* またはソースコードからビルド	ソースコードからビルド
言語リソースライブラリ	*dblg[LL]16.dll*	*dblg[LL]16.res*	*dblg[LL]16.res*
[接続] ウィンドウ	*dbcon16.dll*	なし	なし
SQL Anywhere C API のランタイム	*dbcapi.dll*	*libdbcapi_r.so*	*libdbcapi_r.dylib*
DBLIB (スレッド)	*dblib16.dll*	*libdblib16_r.so*	*libdblib16_r.dylib*
スレッドサポートライブラリ	なし	*libdbtasks16_r.so*	*libdbtasks16_r.dylib*
ICU ライブラリ	*dbicu16.dll*	*libdbicu16_r.so*[2]	*libdbicu16_r.dylib*[2]
ICU データライブラリ	*dbicudt16.dll*	*libdbicudt16.so*[2]	*libdbicudt16.dylib*[2]
オプションの暗号化サポート	*dbfips16.dll*	*libdbfips16_r.so*	*libdbfips16_r.dylib*
オプションの暗号化サポート	*dbrsa16.dll*	*libdbrsa16_r.so*	*libdbrsa16_r.dylib*

注意

- 最新の SQL Anywhere PHP ドライバについては、http://www.sybase.com/detail?id=1019698 を参照してください。
- *php-5.x.y_sqlanywhere_nts.dll* ファイルはスレッド非対応の PHP 拡張です。
- 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**、**jp** 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] ウィンドウを表示する必要がある場合は、**[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 (*dbcon16.dll*) が必要です。
- PHP の Windows と Linux/Solaris へ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。
- PHP に関するホワイトペーパーについては、http://www.sybase.com/detail?id=1057714 を参照してください。

参照

- 「ODBC クライアントの配備」982 ページ

PHP 拡張

Windows の場合、SQL Anywhere には 5.1.1 ～ 5.4.8 の多くの PHP バージョンのスレッド対応拡張が含まれます。SQL Anywhere 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。サポートされる PHP バージョンの拡張ファイル名は、次のパターンに従います。

```
php-5.x.y_sqlanywhere.dll
```

Windows の場合、SQL Anywhere には 5.2.11 ～ 5.4.8 の多くの PHP バージョンのスレッド非対応拡張も含まれます。スレッド非対応の PHP は、マルチプロセスでは使用できますが、Apache や IIS などのマルチスレッド Web サーバでは使用できません。サポートされる PHP バージョンの拡張ファイル名は、次のパターンに従います。

```
php-5.x.y_sqlanywhere_nts.dll
```

Linux および Solaris の場合、SQL Anywhere には 5.1.1 ～ 5.4.8 の多くの PHP バージョンの 64 ビットバージョンと 32 ビットバージョンの拡張が含まれます。また、スレッド型モジュールと

非スレッド型拡張も含まれています。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、非スレッド型拡張を使用します。Apache 2.x を使用する場合は、スレッド型拡張を使用します。サポートされる PHP バージョンの拡張ファイル名は、次のパターンに従います。

```
php-5.x.y_sqlanywhere[_r].so
```

5.x.y は、5.2.11 などのように PHP バージョンを表します。Linux と Solaris の場合、スレッド型バージョンの PHP 拡張はファイル名の最後に *_r* が付きます。Windows バージョンはダイナミックリンクライブラリとして実装され、Linux/Solaris バージョンは共有オブジェクトとして実装されます。以降のバージョンの PHP を使用している場合、最新の SQL Anywhere PHP ドライバについてはインターネットで http://www.sybase.com/detail?id=1019698 を参照してください。

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、PHP 拡張 DLL を SQL Anywhere のインストールディレクトリからコピーして PHP インストールに追加します。オプションとして、拡張を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 5.0 以降をインストールしている必要があります。

♦ タスク

1. 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。

2. **extension_dir** エントリのロケーションを指定する行を探します。

3. 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することをおすすめします。

4. *php-5.x.y_sqlanywhere.dll* ファイルのいずれかを、SQL Anywhere インストール環境の *Bin32* サブフォルダから *php.ini* ファイルの **extension_dir** エントリで指定されたディレクトリにコピーします。

   注意
   文字列 *5.x.y* は、インストールした PHP のバージョン番号を表します。

   使用している PHP のバージョンが SQL Anywhere で提供される SQL Anywhere PHP 拡張よりも新しい場合は、SQL Anywhere で提供される最新の拡張を使用してください。バージョン 5.2.x の SQL Anywhere PHP 拡張は、バージョン 5.3.x の PHP では動作しません。

5. SQL Anywhere PHP ドライバを自動的にロードするために、次の行を *php.ini* ファイルの Dynamic Extensions セクションに追加します。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 ドライバを手動でロードする必要があります。

次のエントリをコピーします。

```
extension=php-5.x.y_sqlanywhere.dll
```

5.x.y は、前の手順でコピーした SQL Anywhere PHP 拡張のバージョン番号を表します。

6. *php.ini* を保存して閉じます。

7. SQL Anywhere インストール環境の *Bin32* サブフォルダがパスに含まれていることを確認してください。

結果

SQL Anywhere PHP 拡張は Windows 環境でサポートされており、使用可能です。

参照

- 「PHP テストページの作成と実行」668 ページ
- 「SQL Anywhere PHP 拡張の設定」1002 ページ

Linux/Solaris での PHP 拡張のインストール

Linux または Solaris で SQL Anywhere PHP 拡張を使用するには、共有オブジェクトを SQL Anywhere のインストールディレクトリからコピーして PHP インストールに追加します。オプションとして、拡張を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 5.0 以降をインストールしている必要があります。

♦ タスク

1. 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 エントリのロケーションを指定する行を探します。

   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することをおすすめします。

2. *php-5.x.y_sqlanywhere* 共有オブジェクトのいずれかを、SQL Anywhere インストール環境の *lib32* または *lib64* サブフォルダから *php.ini* ファイルの **extension_dir** 変数で指定されたディレクトリにコピーします。

   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。

Copyright © 2013, SAP AG or an SAP affiliate company.

> **注意**
> 文字列 *5.x.y* は、インストールした PHP のバージョン番号を表します。
>
> お使いの PHP のバージョンが SQL Anywhere で提供される共有オブジェクトよりも新しい場合は、SQL Anywhere で提供される最新の共有オブジェクトを使用してください。バージョン 5.2.x の SQL Anywhere PHP 拡張は、バージョン 5.3.x の PHP では動作しません。

使用する共有オブジェクトのバージョンについては、「PHP 拡張」998 ページを参照してください。

3. SQL Anywhere PHP ドライバを自動的にロードするために、次の行を *php.ini* ファイルの Dynamic Extensions セクションに追加します。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 ドライバを手動でロードする必要があります。

   エントリは、コピーした共有オブジェクトを特定する必要があり、次のいずれかになります。

   ```
   extension=php-5.x.y_sqlanywhere.so
   ```

   スレッド対応共有オブジェクトの場合は、次のとおりです。

   ```
   extension=php-5.x.y_sqlanywhere_r.so
   ```

   5.x.y は、前の手順でコピーした PHP 共有オブジェクトのバージョン番号を表します。

4. *php.ini* を保存して閉じます。

5. 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。

シェル	使用するコマンド
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 SQL Anywhere 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 SQL Anywhere 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

> **注意**
> 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。

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01001

結果

SQL Anywhere PHP 拡張は Linux/Solaris 環境でサポートされており、使用可能です。

参照

- 「PHP テストページの作成と実行」668 ページ
- 「SQL Anywhere PHP 拡張の設定」1002 ページ

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 インストール環境の *sdk/php* サブフォルダにインストールされているソースコードから PHP 拡張をビルドする必要があります。

参照

- 「UNIX と Mac OS X で SQL Anywhere PHP 拡張を構築する方法」676 ページ

SQL Anywhere PHP 拡張の設定

SQL Anywhere PHP ドライバの動作は、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.y_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.y_sqlanywhere.so
  ```

  ```
  extension=php-5.x.y_sqlanywhere_r.so
  ```

  これらのエントリで、*5.x.y* は PHP のバージョンを表します。

  PHP の起動時に SQL Anywhere 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 = phpversion();
     $extension_name = 'php-'.$version.'_sqlanywhere';
     if( strtoupper(substr(PHP_OS, 0, 3) == 'WIN' )) {
        $extension_ext = '.dll';
     } else {
        $extension_ext = '.so';
  ```

Copyright © 2013, SAP AG or an SAP affiliate company.

```
    }
    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

参照

- 「sasql_set_option」700 ページ

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 Sybase Open Client 製品が必要です。Open Client ソフトウェアは Sybase から別途購入する必要があります。これには、独自のインストール方法があります。

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

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01003

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 *dbrsakp16.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 *libdbrsakp16.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 *libdbrsakp16_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。

管理ツールの配備

ライセンス契約に従って、Interactive SQL、Sybase Central、SQL Anywhere コンソールユーティリティ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。

管理ツールのシステム稼働条件については、http://www.sybase.com/detail?id=1002288 を参照してください。

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 (Sybase Central、Interactive SQL、SQL Anywhere コンソールユーティリティ) 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。

- ***dbconsole.ini*** これは、コンソールユーティリティの初期化ファイルの名前です。
- ***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、Sybase 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、Sybase 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。

Copyright © 2013, SAP AG or an SAP affiliate company.

- **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、Sybase 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、Sybase Central のサンプルの初期化ファイルの内容です。

```
JRE_DIRECTORY=c:¥jdk1.7.0¥jre
VM_ARGUMENTS=-Xmx200m
JAR_PATHS=c:¥scj¥jars
ADDITIONAL_CLASSPATH=
LIBRARY_PATHS=c:¥scj¥bin
APPLICATION_ARGUMENTS=-screpository=c:¥Users¥Public¥Documents¥Sybase Central 16;-
installdir=c:¥scj
```

この例では、JRE のコピーが *c:¥jdk1.7.0¥jre* にあることを前提としています。また、*jsyblib1600* などの Sybase 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。SQL Anywhere の JAR ファイルは *c:¥scj¥jars* に格納されています。

注意
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 (dbeng16) 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 Sybase Central からデータベースを作成する場合にも必要です。

参照

- 「Deployment ウィザード」964 ページ

Windows での管理ツールの配備

この項では、Windows コンピュータで Deployment ウィザードを使用せずに Interactive SQL (dbisql)、Sybase Central (SQL Anywhere、Mobile Link、Ultra Light のプラグインを含む)、および SQL Anywhere コンソールユーティリティ (dbconsole) をインストールする方法について説明します。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、Windows Mobile を除くすべての 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 16.0 固有のもので、前後のバージョンには適用できません。

注意
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。

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01005

始める前に

この項を読む前に、REGEDIT アプリケーションを含む Windows レジストリに関して理解しておいてください。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。システムをバックアップしてからレジストリを変更することをおすすめします。

管理ツールの配備には以下の手順が必要です。

1. 配備の対象を決定します。
2. 必要なファイルをコピーします。
3. 管理ツールを Windows に登録します。
4. システムパスを更新します。
5. Sybase Central にプラグインを登録します。
6. Windows に SQL Anywhere の ODBC ドライバを登録します。
7. Windows にオンラインヘルプのファイルを登録します。

これらの各手順については以下の項で詳しく説明します。

手順 1：配備するソフトウェアを決定します

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 Sybase Central
- Mobile Link プラグインを含む Sybase Central
- Relay Server プラグインを含む Sybase Central
- Ultra Light プラグインを含む Sybase Central
- SQL Anywhere コンソールユーティリティ (dbconsole)

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 Runtime Environment (JRE) バージョン 1.7.0。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。

Copyright © 2013, SAP AG or an SAP affiliate company.

●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。詳細については、http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136191.html を参照してください。

以下の項の手順は、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。

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:¥sa16* 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
sa16	ルートフォルダ。以下の手順では、*c:¥sa16* 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¥SQLAny16*)。
sa16¥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。
sa16¥Bin32¥jre170	32 ビット Java Runtime Environment
sa16¥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。
sa16¥Bin64¥jre170	64 ビット Java Runtime Environment
sa16¥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 Sybase 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。

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01007

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。詳細については、「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」1011 ページを参照してください。

管理ツールには JRE 1.7.0 が必要です。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 *%SQLANY16%¥Bin32¥jre170* ディレクトリからコピーします。サブフォルダも含めて *jre170* 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 *%SQLANY16%¥Bin64¥jre170* ディレクトリからコピーします。サブフォルダも含めて *jre170* 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¥dbelevate16.exe
BinXX¥dbicu16.dll
BinXX¥dbicudt16.dll
BinXX¥dbisql.com
BinXX¥dbisql.exe
BinXX¥dbjodbc16.dll
BinXX¥dblg[LL]16.dll
BinXX¥dblib16.dll
BinXX¥dbodbc16.dll
BinXX¥jsyblib1600.dll
BinXX¥jre170¥...
Java¥batik*.jar
Java¥isql.jar
Java¥JComponents1600.jar
Java¥jlogon.jar
Java¥jodbc4.jar
Java¥js.jar
Java¥ngdbc.jar
Java¥jsyblib1600.jar
Java¥pdf-transcoder.jar
Java¥saip16.jar
Java¥SCEditor1600.jar
Java¥xerces_2_5_0.jar
Java¥xml-apis-ext.jar
Java¥xml-apis.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¥jre170*) 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¥jre170*) が必要です。

Sybase Central

Sybase Central (scjview) には次のファイルが必要です。

BinXX¥jsyblib1600.dll
BinXX¥scjview.exe

Copyright © 2013, SAP AG or an SAP affiliate company.

BinXX¥jre170¥...
Java¥jsyblib1600.jar
Java¥salib.jar
Java¥SCEditor1600.jar
Java¥sybasecentral1600.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 Sybase Central には、32 ビットバージョンの JRE ファイル (*Bin32¥jre170*) が必要です。64 ビットバージョンの Sybase Central には、64 ビットバージョンの JRE ファイル (*Bin64¥jre170*) が必要です。

SQL Anywhere プラグインを含む Sybase Central

Sybase Central の SQL Anywhere プラグインでは、パーソナルサーバ (*dbeng16.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¥dbfips16.dll
BinXX¥dbicu16.dll
BinXX¥dbicudt16.dll
BinXX¥dbjodbc16.dll
BinXX¥dblg[LL]16.dll
BinXX¥dblib16.dll
BinXX¥dbodbc16.dll
BinXX¥dbput16.dll
BinXX¥dbtool16.dll
BinXX¥sbgse2.dll
Java¥batik*.jar
Java¥debugger.jar
Java¥isql.jar
Java¥JComponents1600.jar
Java¥jlogon.jar
Java¥jodbc4.jar
Java¥js.jar
Java¥pdf-transcoder.jar
Java¥saplugin.jar
Java¥SQLAnywhere.jpr
Java¥xerces_2_5_0.jar
Java¥xml-apis-ext.jar
Java¥xml-apis.jar

Mobile Link プラグインを含む Sybase Central

Sybase 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¥dbicu16.dll
BinXX¥dbicudt16.dll
BinXX¥dblg[LL]16.dll
BinXX¥dblib16.dll
BinXX¥dbput16.dll
BinXX¥dbtool16.dll
BinXX¥mljodbc16.dll
Java¥isql.jar
Java¥JComponents1600.jar
Java¥jlogon.jar
Java¥jodbc4.jar
Java¥mldesign.jar
Java¥mlplugin.jar
Java¥MobiLink.jpr
Java¥stax-api-1.0.jar

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01009

Java¥velocity-dep.jar
Java¥velocity.jar
Java¥wstx-asl-3.2.6.jar

Relay Server プラグインを含む Sybase Central

Sybase Central の Relay Server プラグインには次のファイルが必要です。

Java¥JComponents1600.jar
Java¥jlogon.jar
Java¥RelayServer.jpr
Java¥rsplugin.jar
Java¥rstool.jar

Ultra Light プラグインを含む Sybase Central

Sybase 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¥dbicu16.dll
BinXX¥dbicudt16.dll
BinXX¥dblg[LL]16.dll
BinXX¥dblib16.dll
BinXX¥dbput16.dll
BinXX¥dbtool16.dll
BinXX¥mlcrsa16.dll
BinXX¥mlcrsafips16.dll
BinXX¥ulfips16.dll
BinXX¥ulscutil16.dll
BinXX¥ulutils16.dll
Java¥batik*.jar
Java¥isql.jar
Java¥JComponents1600.jar
Java¥jlogon.jar
Java¥jodbc4.jar
Java¥js.jar
Java¥pdf-transcoder.jar
Java¥ulplugin.jar
Java¥UltraLite.jpr
Java¥xerces_2_5_0.jar
Java¥xml-apis-ext.jar
Java¥xml-apis.jar

DBConsole

DBConsole アプリケーションには次のファイルが必要です。

BinXX¥dbconsole.exe
BinXX¥dbdsn.exe
BinXX¥dbelevate16.exe
BinXX¥dbicu16.dll
BinXX¥dbicudt16.dll
BinXX¥dbjodbc16.dll
BinXX¥dblg[LL]16.dll
BinXX¥dblib16.dll
BinXX¥dbodbc16.dll
BinXX¥jsyblib1600.dll
BinXX¥jre170¥...
Java¥DBConsole.jar
Java¥JComponents1600.jar
Java¥jlogon.jar

Copyright © 2013, SAP AG or an SAP affiliate company.

Java¥jodbc4.jar
Java¥jsyblib1600.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 DBConsole には、32 ビットバージョンの JRE ファイル (*Bin32¥jre170*) が必要です。64 ビットバージョンの DBConsole には、64 ビットバージョンの JRE ファイル (*Bin64¥jre170*) が必要です。

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名も似ていますが、**en** が **fr** になります。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。

dblgen16.dll	英語
dblgde16.dll	ドイツ語
dblgfr16.dll	フランス語
dblgja16.dll	日本語
dblgzh16.dll	中国語 (簡体文字)

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¥jre170¥lib*
jaccess.jar	*Bin32¥jre170¥lib¥ext*
access-bridge.jar	*Bin32¥jre170¥lib¥ext*
JavaAccessBridge.dll	*Bin32¥jre170¥bin*

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01011

JAWTAccessBridge.dll	*Bin32¥jre170¥bin*
WindowsAccessBridge.dll	*%windir%¥system32* (32 ビットの Windows) または *%windir%¥SysWOW64* (64 ビットの Windows)

● **64 ビット管理ツール**

accessibility.properties	*Bin64¥jre170¥lib*
jaccess.jar	*Bin64¥jre170¥lib¥ext*
access-bridge.jar	*Bin64¥jre170¥lib¥ext*
JavaAccessBridge-64.dll	*Bin64¥jre170¥bin*
JAWTAccessBridge-64.dll	*Bin64¥jre170¥bin*
WindowsAccessBridge-64.dll	*%windir%¥system32*

インストーラによって *BinXX¥jre170¥lib* フォルダに *accessibility.properties* ファイルが作成され、かつ、このファイルに、次に示す **assistive_technologies** オプションが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 (# を削除する)。

手順 3：管理ツールを Windows に登録します。

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¥Sybase¥SQL Anywhere¥16.0** では次のように設定します。

○ **Location**　Sybase Central ファイルを格納する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 16* 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。

Copyright © 2013, SAP AG or an SAP affiliate company.

- **HKEY_LOCAL_MACHINE¥SOFTWARE¥Sybase¥Sybase Central¥16.0** では次のように設定します。
  - **Language**　Sybase Central で使用する言語を表す 2 文字のコード。これは次のいずれかにしてください。英語、ドイツ語、フランス語、日本語、簡体中国語に対して、それぞれ **EN**、**DE**、**FR**、**JA**、**ZH**
- **HKEY_LOCAL_MACHINE¥SOFTWARE¥Sybase¥SQL Anywhere¥16.0** では次のように設定します。
  - **Language**　SQL Anywhere とその他の管理ツールで使用する言語を表す 2 文字のコード。これは次のいずれかにしてください。英語、ドイツ語、フランス語、日本語、簡体中国語に対して、それぞれ **EN**、**DE**、**FR**、**JA**、**ZH**

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¥Sybase** 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:¥sa16* を使用した場合の *.reg* ファイルの例です。

```
REGEDIT4

[HKEY_LOCAL_MACHINE¥SOFTWARE¥Sybase¥Sybase Central¥16.0]
"Language"="FR"

[HKEY_LOCAL_MACHINE¥SOFTWARE¥Sybase¥SQL Anywhere¥16.0]
"Location"="c:¥¥sa16"
"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。

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。システムパスに *c:¥sa16¥Bin32* ディレクトリか *c:¥sa16¥Bin64* ディレクトリのいずれかを追加してください。

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
```

手順 5：Sybase Central の接続プロファイルを作成します

この手順では Sybase Central を設定します。Sybase Central をインストールしない場合は、省略できます。

Sybase Central がシステムにインストールされている場合は、**SQL Anywhere 16 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 16 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 16 Demo/Name" "SQL Anywhere 16 Demo"
scjview -write "ConnectionProfiles/SQL Anywhere 16 Demo/FirstTimeStart" "false"
scjview -write "ConnectionProfiles/SQL Anywhere 16 Demo/Description" "Suitable Description"
scjview -write "ConnectionProfiles/SQL Anywhere 16 Demo/ProviderId" "sqlanywhere1600"
scjview -write "ConnectionProfiles/SQL Anywhere 16 Demo/Provider" "SQL Anywhere 16"
scjview -write "ConnectionProfiles/SQL Anywhere 16 Demo/Data/ConnectionProfileSettings" "DSN
¥eSQL^0020Anywhere^002016^0020Demo;UID¥eDBA;PWD¥e35c624d517fb"
scjview -write "ConnectionProfiles/SQL Anywhere 16 Demo/Data/ConnectionProfileName" "SQL
Anywhere 16 Demo"
scjview -write "ConnectionProfiles/SQL Anywhere 16 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。Sybase 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: 16.0.1154
#  Thu Oct 04 12:07:53 EDT 2012
#
ConnectionProfiles/SQL Anywhere 16 Demo/Name=SQL Anywhere 16 Demo
ConnectionProfiles/SQL Anywhere 16 Demo/FirstTimeStart=false
ConnectionProfiles/SQL Anywhere 16 Demo/Description=Suitable Description
ConnectionProfiles/SQL Anywhere 16 Demo/ProviderId=sqlanywhere1600
ConnectionProfiles/SQL Anywhere 16 Demo/Provider=SQL Anywhere 16
ConnectionProfiles/SQL Anywhere 16 Demo/Data/ConnectionProfileSettings=
   DSN¥eSQL^0020Anywhere^002016^0020Demo;
   UID¥eDBA;
   PWD¥e35c624d517fb
ConnectionProfiles/SQL Anywhere 16 Demo/Data/ConnectionProfileName=
   SQL Anywhere 16 Demo
ConnectionProfiles/SQL Anywhere 16 Demo/Data/ConnectionProfileType=
    SQL Anywhere
```

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。

参照

- 「ODBC ドライバの設定」986 ページ

Linux、Solaris、Mac OS X における管理ツールの配備

この項では、Linux、Solaris、Mac OS X のコンピュータで Interactive SQL (dbisql)、Sybase Central (SQL Anywhere と Mobile Link のプラグインを含む)、SQL Anywhere コンソールユーティリティ

(dbconsole) をインストールする方法を説明します。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 16.0 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。「dbisqlc の配備」1026 ページを参照してください。

注意
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. Sybase Central プラグインを登録します。

これらの各手順については以下の項で詳しく説明します。

手順 1：配備するソフトウェアを決定します

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 Sybase Central
- Mobile Link プラグインを含む Sybase Central
- SQL Anywhere コンソールユーティリティ (dbconsole)

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 Runtime Environment (JRE) バージョン 1.7.0。考慮すべき 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。

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01015

> **注意**
> 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。Java 1.7.0 がリストに表示されていない場合は、http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1637583.html にアクセスします。

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。

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てください。

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。

管理ツールには JRE 1.7.0 が必要です。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 *jre170* 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 Sybase 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する必要があります。

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する必要があります。

管理ツールと Sybase 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。

Copyright © 2013, SAP AG or an SAP affiliate company.

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、*$SQLANY16/bin64* または *$SQLANY16/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 *jre170* です。次にその例を示します。

```
libdblib16_r.so -> $SQLANY16/lib32/libdblib16_r.so.1
jre170 -> $SQLANY16/bin32/jre170
```

64 ビット Linux の Mobile Link プラグインの場合、*$SQLANY16/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 *jre170* です。

```
jre170 -> $SQLANY16/bin64/jre170 (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。

```
libdbjodbc16.jnilib -> libdbjodbc16.dylib
libdblib16_r.jnilib -> libdblib16_r.dylib
libdbput16_r.jnilib -> libdbput16_r.dylib
libmljodbc16.jnilib -> libmljodbc16.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。詳細については、「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」1011 ページを参照してください。

Interactive SQL

Interactive SQL (dbisql) には次のファイルが必要です。

```
binXX/dbisql
binXX/jre170/...
libXX/libdbicu16_r.so (.dylib)
libXX/libdbicudt16.so (.dylib)
libXX/libdbjodbc16.so (.dylib)
libXX/libdblib16_r.so (.dylib)
libXX/libdbodbc16_r.so (.dylib)
libXX/libdbodm16.so (.dylib)
libXX/libjsyblib1600_r.so (.dylib)
res/dblg[LL]16.res
java/batik*.jar
java/isql.jar
java/JComponents1600.jar
java/jlogon.jar
java/jodbc4.jar
java/js.jar
java/jsyblib1600.jar
java/ngdbc.jar
java/pdf-transcoder.jar
java/saip16.jar
java/SCEditor1600.jar
java/xerces_2_5_0.jar
java/xml-apis-ext.jar
java/xml-apis.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/jre170*) 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/jre170*) が必要です。

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01017

Sybase Central

Sybase Central (scjview) には次のファイルが必要です。

binXX/scjview
binXX/jre170/...
libXX/libjsyblib1600_r.so (.dylib)
java/jsyblib1600.jar
java/salib.jar
java/SCEditor1600.jar
java/sybasecentral1600.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 Sybase Central には、32 ビットバージョンの JRE ファイル (*bin32/jre170*) が必要です。64 ビットバージョンの Sybase Central には、64 ビットバージョンの JRE ファイル (*bin64/jre170*) が必要です。

SQL Anywhere プラグインを含む Sybase Central

Sybase Central の SQL Anywhere プラグインでは、パーソナルサーバ (*dbeng16* とサポートファイル) と次のファイルが必要です。

libXX/libdbfips16.so (.dylib)
libXX/libdbicu16_r.so (.dylib)
libXX/libdbicudt16.so (.dylib)
libXX/libdbjodbc16.so (.dylib)
libXX/libdblib16_r.so (.dylib)
libXX/libdbodbc16_r.so (.dylib)
libXX/libdbodm16.so (.dylib)
libXX/libdbput16_r.so (.dylib)
libXX/libdbtasks16_r.so (.dylib)
libXX/libdbtool16_r.so (.dylib)
libXX/sbgse2.so
res/dblg[LL]16.res
java/batik*.jar
java/debugger.jar
java/isql.jar
java/JComponents1600.jar
java/jlogon.jar
java/jodbc4.jar
java/js.jar
java/pdf-transcoder.jar
java/saplugin.jar
java/SQLAnywhere.jpr
java/xerces_2_5_0.jar
java/xml-apis-ext.jar
java/xml-apis.jar

Mobile Link プラグインを含む Sybase Central

Sybase Central の Mobile Link プラグインには次のファイルが必要です。

libXX/libdbicu16_r.so (.dylib)
libXX/libdbicudt16.so (.dylib)
libXX/libdblib16_r.so (.dylib)
libXX/libdbput16_r.so (.dylib)
libXX/libdbtasks16_r.so (.dylib)
libXX/libdbtool16_r.so (.dylib)
libXX/libmljodbc16.so (.dylib)
res/dblg[LL]16.res
java/isql.jar

Copyright © 2013, SAP AG or an SAP affiliate company.

java/JComponents1600.jar
java/jlogon.jar
java/jodbc4.jar
java/mldesign.jar
java/mlplugin.jar
java/MobiLink.jpr
java/stax-api-1.0.jar
java/velocity-dep.jar
java/velocity.jar
java/wstx-asl-3.2.6.jar

Relay Server プラグインを含む Sybase Central

Sybase Central の Relay Server プラグインには次のファイルが必要です。

java/JComponents1600.jar
java/jlogon.jar
java/RelayServer.jpr
java/rsplugin.jar
java/rstool.jar

Ultra Light プラグインを含む Sybase Central

Sybase Central の Ultra Light プラグインは、Windows と Linux でのみ使用できます。Sybase Central の Ultra Light プラグインには次のファイルが必要です。

libXX/libdbicu16_r.so (.dylib)
libXX/libdbicudt16.so (.dylib)
libXX/libdblib16_r.so (.dylib)
libXX/libdbput16_r.so (.dylib)
libXX/libdbtasks16_r.so (.dylib)
libXX/libdbtool16_r.so (.dylib)
libXX/libmlcrsa16.so (.dylib)
libXX/libmlcrsafips16.so (.dylib)
libXX/libulfips16.so (.dylib)
libXX/libulscutil16.so (.dylib)
libXX/libulutils16.so (.dylib)
res/dblg[LL]16.res
java/batik*.jar
java/isql.jar
java/JComponents1600.jar
java/jlogon.jar
java/jodbc4.jar
java/js.jar
java/pdf-transcoder.jar
java/ulplugin.jar
java/UltraLite.jpr
java/xerces_2_5_0.jar
java/xml-apis-ext.jar
java/xml-apis.jar

DBConsole

DBConsole アプリケーションには次のファイルが必要です。

binXX/dbconsole
binXX/jre170/...
libXX/libdbicu16_r.so (.dylib)
libXX/libdbicudt16.so (.dylib)
libXX/libdbjodbc16.so (.dylib)
libXX/libdblib16_r.so (.dylib)
libXX/libdbodbc16_r.so (.dylib)

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01019

```
libXX/libdbodm16.so (.dylib)
libXX/libjsyblib1600.so (.dylib)
res/dblg[LL]16.res
java/DBConsole.jar
java/JComponents1600.jar
java/jlogon.jar
java/jodbc4.jar
java/jsyblib1600.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 DBConsole には、32 ビットバージョンの JRE ファイル (*bin32/jre170*) が必要です。64 ビットバージョンの DBConsole には、64 ビットバージョンの JRE ファイル (*bin64/jre170*) が必要です。

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。

dblgen16.res	英語
dblgde16_iso_1.res、*dblgde16_utf8.res*	ドイツ語 (Linux のみ)
dblgja16_eucjis.res、*dblgja16_sjis.res*、*dblgja16_utf8.res*	日本語 (Linux のみ)
dblgzh16_cp936.res、*dblgzh16_eucgb.res*、*dblgzh16_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。

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、SQLANY16 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。

   ```
   SQLANY16="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。

Copyright © 2013, SAP AG or an SAP affiliate company.

- **32 ビット Linux および Solaris**

  $SQLANY16/bin32:

- **64 ビット Linux および Solaris**

  $SQLANY16/bin64:

3. 以下を含むように LD_LIBRARY_PATH を設定します。

- **32 ビット Linux および Solaris**

  $SQLANY16/lib32:
  $SQLANY16/lib64:
  $SQLANY16/bin32/jre170/lib/i386/client:
  $SQLANY16/bin32/jre170/lib/i386/server:
  $SQLANY16/bin32/jre170/lib/i386:
  $SQLANY16/bin32/jre170/lib/i386/native_threads

- **64 ビット Linux および Solaris**

  $SQLANY16/lib32:
  $SQLANY16/lib64:
  $SQLANY16/bin64/jre170/lib/amd64/client:
  $SQLANY16/bin64/jre170/lib/amd64/server:
  $SQLANY16/bin64/jre170/lib/amd64:
  $SQLANY16/bin64/jre170/lib/amd64/native_threads

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。

手順 4：Sybase Central の接続プロファイルを作成します

この手順では Sybase Central を設定します。Sybase Central をインストールしない場合は、省略できます。

Sybase Central がシステムにインストールされている場合は、**SQL Anywhere 16 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 16 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 16 Demo/Name" "SQL Anywhere 16 Demo"
scjview -write "ConnectionProfiles/SQL Anywhere 16 Demo/FirstTimeStart" "false"
scjview -write "ConnectionProfiles/SQL Anywhere 16 Demo/Description" "Suitable Description"
scjview -write "ConnectionProfiles/SQL Anywhere 16 Demo/ProviderId" "sqlanywhere1600"
scjview -write "ConnectionProfiles/SQL Anywhere 16 Demo/Provider" "SQL Anywhere 16"
scjview -write "ConnectionProfiles/SQL Anywhere 16 Demo/Data/ConnectionProfileSettings" "DSN
```

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01021

```
¥eSQL^0020Anywhere^002016^0020Demo;UID¥eDBA;PWD¥e35c624d517fb"
scjview -write "ConnectionProfiles/SQL Anywhere 16 Demo/Data/ConnectionProfileName" "SQL
Anywhere 16 Demo"
scjview -write "ConnectionProfiles/SQL Anywhere 16 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。Sybase 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: 16.0.1154
#  Thu Oct 04 12:07:53 EDT 2012
#
ConnectionProfiles/SQL Anywhere 16 Demo/Name=SQL Anywhere 16 Demo
ConnectionProfiles/SQL Anywhere 16 Demo/FirstTimeStart=false
ConnectionProfiles/SQL Anywhere 16 Demo/Description=Suitable Description
ConnectionProfiles/SQL Anywhere 16 Demo/ProviderId=sqlanywhere1600
ConnectionProfiles/SQL Anywhere 16 Demo/Provider=SQL Anywhere 16
ConnectionProfiles/SQL Anywhere 16 Demo/Data/ConnectionProfileSettings=
   DSN¥eSQL^0020Anywhere^002016^0020Demo;
   UID¥eDBA;
   PWD¥e35c624d517fb
ConnectionProfiles/SQL Anywhere 16 Demo/Data/ConnectionProfileName=
   SQL Anywhere 16 Demo
ConnectionProfiles/SQL Anywhere 16 Demo/Data/ConnectionProfileType=
    SQL Anywhere
```

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 16¥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。

注意
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]
```

Copyright © 2013, SAP AG or an SAP affiliate company.

```
allowPasswordsInFavorites=true
defaultShowResultsForAllStatements=false
defaultShowMultipleResultSets=false
disableExecuteAll=false
fastLauncherEnabled=true
# lockedPreferences is assigned a comma-separated
# list of one or more of the following option names:
#    autoCommit
#    autoRefetch
#    commitOnExit
#    disableResultsEditing
#    executeToolbarButtonSemantics
#    fastLauncherEnabled
#    maximumDisplayedRows
#    showMultipleResultSets
#    showResultsForAllStatements
lockedPreferences=showMultipleResultSets,commitOnExit
```

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、次のセクションに分かれています。

[errors]

この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%¥sybase* 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。

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01023

[performancedata]

このセクションで設定されるオプションは、Interactive SQL と Sybase 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。「パフォーマンスデータの定期送信」『SQL Anywhere サーバ データベース管理』を参照してください。

[dbisql]

この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。「SQL スクリプトファイル、SQL 文、および接続をお気に入りに追加する」『SQL Anywhere サーバ データベース管理』を参照してください。

- **defaultShowResultsForAllStatements**　このオプションにより、Interactive SQL に結果が表示される方法のデフォルト設定が変更されます。**defaultShowResultsForAllStatements** が true に設定されている場合、すべてのデータベースタイプのデフォルトは、すべての文の結果を表示することです。このオプションが false に設定されている場合、すべてのデータベースタイプのデフォルトは、最後の文の結果のみを表示することです。

  オプションでプレフィクスを、特定のデータベースタイプへの影響を制限するために使用できます。プレフィクスが存在しない場合、すべてのデータベースタイプにディレクティブが適用されます。プレフィクスは次のとおりです。

  - **SQLAnywhere.**
  - **UltraLite.**
  - **SybaseIQ.**
  - **HANA.**

  たとえば、次のオプション設定は、SQL Anywhere データベースのデフォルト設定にのみ影響を与えます。

  ```
  SQLAnywhere.defaultShowResultsForAllStatements=true
  ```

- **defaultShowMultipleResultSets**　このオプションにより、Interactive SQL に結果が表示される方法のデフォルト設定が変更されます。**defaultShowMultipleResultSets** が true に設定されている場合、すべてのデータベースタイプのデフォルトは、複数の結果セットを表示することです。このオプションが false に設定されている場合、すべてのデータベースタイプのデフォルトは、最初の結果セットのみを表示することです。

Copyright © 2013, SAP AG or an SAP affiliate company.

オプションでプレフィクスを、特定のデータベースタイプへの影響を制限するために使用できます。プレフィクスが存在しない場合、すべてのデータベースタイプにディレクティブが適用されます。プレフィクスは次のとおりです。

- **SQLAnywhere.**
- **SybaseIQ.**

- **disableExecuteAll**　**disableExecuteAll** が true に設定されている場合、**[SQL] » の [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。「SQL 文の実行 (Interactive SQL)」『SQL Anywhere サーバ データベース管理』を参照してください。

- **fastLauncherEnabled**　**fastLauncherEnabled** が true に設定されると、高速ランチャ機能は有効になります。高速ランチャ機能を無効にするには、オプションを false に設定します。「高速ランチャオプション」『SQL Anywhere サーバ データベース管理』を参照してください。

- **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。「auto_commit オプション [Interactive SQL]」『SQL Anywhere サーバ データベース管理』を参照してください。

- **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。「auto_refetch オプション [Interactive SQL]」『SQL Anywhere サーバ データベース管理』を参照してください。

- **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。「commit_on_exit オプション [Interactive SQL]」『SQL Anywhere サーバ データベース管理』を参照してください。

- **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。「SQL 文の実行 (Interactive SQL)」『SQL Anywhere サーバ データベース管理』を参照してください。

- **fastLauncherEnabled**　ユーザが高速ランチャのオプションをカスタマイズできないようにします。「高速ランチャオプション」『SQL Anywhere サーバ データベース管理』を参照してください。

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01025

- **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。「isql_maximum_displayed_rows オプション [Interactive SQL]」『SQL Anywhere サーバ データベース管理』を参照してください。

- **showMultipleResultSets**　ユーザが **[最初の結果セットだけを表示]** オプションまたは **[すべての結果セットを表示]** オプションをカスタマイズできないようにします。この設定は、また、ユーザが isql_show_multiple_result_sets オプションを設定できないようにします。「isql_show_multiple_result_sets オプション [Interactive SQL]」『SQL Anywhere サーバ データベース管理』を参照してください。

- **showResultsForAllStatements**　ユーザが **[最後の文の結果を表示]** オプションまたは **[各文の結果を表示]** オプションをカスタマイズできないようにします。「複数の結果セットを返す」『SQL Anywhere サーバ SQL の使用法』を参照してください。

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 実行プログラムの配備が必要になることもあります。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。

参照

- 「dbisqlc ユーティリティ (旧式)」『SQL Anywhere サーバ データベース管理』
- 「Interactive SQL ユーティリティ (dbisql)」『SQL Anywhere サーバ データベース管理』

マニュアルの配備

次の説明では **[LL]** 指定を使用します。マニュアルは各種言語で提供されています。**[LL]** を配備するマニュアルの言語コード (**en**、**de**、**jp** など) に置き換えます。

Windows の場合、マニュアルは 2 通りの方法で使用できます。DCX ヘルプを使用する (この場合、dcx.sybase.com Web サイトからヘルプを入手できるため、ヘルプファイルを配備する必要はありません) か、または HTML ベースのヘルプを配備できます。

HTML ベースのヘルプマニュアルは、*%SQLANY16%¥Documentation* ディレクトリツリーにインストールされます。

Documentation¥sqlanywhere_[LL]16.map
Documentation¥[LL]¥htmlhelp¥dbadmin16.chm
Documentation¥[LL]¥htmlhelp¥dbprogramming16.chm
Documentation¥[LL]¥htmlhelp¥dbreference16.chm
Documentation¥[LL]¥htmlhelp¥dbspatial16.chm
Documentation¥[LL]¥htmlhelp¥dbusage16.chm
Documentation¥[LL]¥htmlhelp¥mlclient16.chm
Documentation¥[LL]¥htmlhelp¥mlserver16.chm
Documentation¥[LL]¥htmlhelp¥mlsisync16.chm

Copyright © 2013, SAP AG or an SAP affiliate company.

```
Documentation¥[LL]¥htmlhelp¥mlstart16.chm
Documentation¥[LL]¥htmlhelp¥relayserver16.chm
Documentation¥[LL]¥htmlhelp¥sachanges16.chm
Documentation¥[LL]¥htmlhelp¥sacshelp16.chm
Documentation¥[LL]¥htmlhelp¥saerrors16.chm
Documentation¥[LL]¥htmlhelp¥saintro16.chm
Documentation¥[LL]¥htmlhelp¥sausinghelp16.chm
Documentation¥[LL]¥htmlhelp¥sqlanywhere_[LL]16.chm
Documentation¥[LL]¥htmlhelp¥sqlremote16.chm
Documentation¥[LL]¥htmlhelp¥uladmin16.chm
Documentation¥[LL]¥htmlhelp¥ulc16.chm
Documentation¥[LL]¥htmlhelp¥uldotnet16.chm
Documentation¥[LL]¥htmlhelp¥ulj16.chm
```

Linux、UNIX、Mac OS X の場合、マニュアルは 2 通りの方法で使用できます。DCX ヘルプを使用する (この場合、dcx.sybase.com Web サイトからヘルプを入手できるため、ヘルプファイルを配備する必要はありません) か、または Eclipse ヘルプを配備できます。

Eclipse ヘルプは、マニュアルインストーラ (*sa16_doc_[LL]_linux_x86+x64.[version].[build].tar.gz*) を使用してローカルにインストールできます。さまざまな言語に使用できるバージョンがあります。

たとえば、*sa16_doc_en_linux_x86+x64.1600.2406.tar.gz* をダウンロードした場合、次のコマンドを使用してマニュアルをインストールできます。

```
/setup -ss -sqlany-dir $SQLANY16
```

これは、環境変数で指定されたロケーションにヘルプをサイレントインストールします。

データベースサーバの配備

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。詳細については、「Deployment ウィザード」964 ページを参照してください。

データベースサーバを稼働するには、一連のファイルをインストールする必要があります。これらの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
dbeng16.exe	*dbeng16*	*dbeng16*
dbeng16.lic	*dbeng16.lic*	*dbeng16.lic*
dbsrv16.exe	*dbsrv16*	*dbsrv16*

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01027

Windows	Linux/UNIX	Mac OS X
dbsrv16.lic	*dbsrv16.lic*	*dbsrv16.lic*
dbserv16.dll	*libdbserv16_r.so*、*libdbtasks16_r.so*	*libdbserv16_r.dylib*、*libdbtasks16_r.dylib*
dbscript16.dll	*libdbscript16_r.so*	*libdbscript16_r.dylib*
dblg[LL]16.dll	*dblg[LL]16.res*	*dblgen16.res*
dbghelp.dll	なし	なし
dbctrs16.dll	なし	なし
dbextf.dll[1]	*libdbextf.so*[1]	*libdbextf.dylib*[1]
dbicu16.dll[2]	*libdbicu16_r.so*[2]	*libdbicu16_r.dylib*[2]
dbicudt16.dll[2、3]	*libdbicudt16.so*[2]	*libdbicudt16.dylib*[2]
sqlany.cvf	*sqlany.cvf*	*sqlany.cvf*
dbrsakp16.dll[4]	*libdbrsakp16_r.so*[4]	*libdbrsakp16_r.dylib*[4]
dbodbc16.dll[5]	*libdbodbc16.so*[5]	*libdbodbc16.dylib*[5]
dbjodbc16.dll[5]	*libdbjodbc16.so*[5]	*libdbjodbc16.dylib*[5]
なし	*libdbodbc16_n.so*[5]	*libdbodbc16_n.dylib*[5]
なし	*libdbodbc16_r.so*[5]	*libdbodbc16_r.dylib*[5]
dbjdbc16.dll[6]	*libdbjdbc16.so*[6]	*libdbjdbc16.dylib*[6]
java¥sajdbc4.jar[6]	*java/sajdbc4.jar*[6]	*java/sajdbc4.jar*[6]
java¥sajvm.jar[6]	*java/sajvm.jar*[6]	*java/sajvm.jar*[6]
dbcis16.dll[7]	*libdbcis16.so*[7]	*libdbcis16.dylib*[7]
libsybbr.dll[8]	*libsybbr.so*[8]	*libsybbr.dylib*[8]

[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。

Copyright © 2013, SAP AG or an SAP affiliate company.

[3] Windows Mobile の場合、配備するファイル名は *dbicudt16.dat* です。

[4] 暗号化された TDS 接続にのみ必要です。

[5] 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。

[6] データベースで Java を使用する場合のみ必要です。

[7] リモートデータアクセスを使用する場合のみ必要です。

[8] アーカイブバックアップにのみ必要です。

説明

- 場合によって、パーソナルデータベースサーバ (dbeng16) またはネットワークデータベースサーバ (dbsrv16) 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 (*dbeng16.lic* または *dbsrv16.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**、**jp** 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内の Java 機能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。

  ユーティリティの配備については、「管理ツールの配備」1004 ページを参照してください。
- 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。

  xd オプションの詳細については、「-xd データベースサーバオプション」『SQL Anywhere サーバ データベース管理』を参照してください。

  サーバ名の指定については、「ServerName (Server) 接続パラメータ」『SQL Anywhere サーバ データベース管理』を参照してください。

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。

前提条件

この作業を実行するための前提条件はありません。

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01029

♦ タスク

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
```

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。

前提条件

この作業を実行するための前提条件はありません。

♦ タスク

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 16.0

   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 16.0

Copyright © 2013, SAP AG or an SAP affiliate company.

2. このキー内で、EventMessageFile という REG_SZ 値を追加し、*dblgen16.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 16¥Bin32¥dblgen16.dll*)。

   次の例は、EventMessageFile キーを *c:¥¥sa16¥¥bin32¥¥dblgen16.dll* ディレクトリに指定します。

   ```
   "EventMessageFile"="c:¥¥sa16¥¥bin32¥¥dblgen16.dll"
   ```

   英語バージョンの DLL である *dblgen16.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
   16.0 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 16.0 Admin
   ```

4. このキー内で、EventMessageFile という REG_SZ 値を追加し、*dblgen16.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 16¥Bin32¥dblgen16.dll*)。

   たとえば、次の例は、EventMessageFile キーを *c:¥¥sa16¥¥bin32¥¥dblgen16.dll* ディレクトリに指定します。

   ```
   "EventMessageFile"="c:¥¥sa16¥¥bin32¥¥dblgen16.dll"
   ```

   英語バージョンの DLL である *dblgen16.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¥Sybase¥SQL Anywhere¥16.0¥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。

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01031

```
"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
16.0]
"EventMessageFile"="c:¥¥sa16¥¥bin32¥¥dblgen16.dll"

[HKEY_LOCAL_MACHINE¥SYSTEM¥CurrentControlSet¥Services¥Eventlog¥Application¥SQLANY
16.0 Admin]
"EventMessageFile"="c:¥¥sa16¥¥bin32¥¥dblgen16.dll"

[HKEY_LOCAL_MACHINE¥SOFTWARE¥Sybase¥SQL Anywhere¥16.0]
"EventLogMask"=dword:00000003
```

この例では、*c:¥¥sa16¥¥bin32¥¥dblgen16.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。

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。

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。

Copyright © 2013, SAP AG or an SAP affiliate company.

参照

- 「リモートデータベースの抽出」『SQL Remote』

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、「*キーワード=値*」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。

参照

- 「CREATE DATABASE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「初期化ユーティリティ (dbinit)」『SQL Anywhere サーバ データベース管理』
- 「照合の考慮事項」『SQL Anywhere サーバ データベース管理』

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できます。

読み込み専用モードによるデータベース実行の詳細については、「-r データベースサーバオプション」『SQL Anywhere サーバ データベース管理』を参照してください。

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。

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01033

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 Vista 以降のバージョンの Windows 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 (*dbelevate16.exe*) を含める必要があります。また、ODBC ドライバは Windows Mobile に登録する必要はありません。

DLL は、インストールスクリプトに含める方法、Windows の regsvr32 ユーティリティまたは Windows Mobile の regsvrce ユーティリティを使用する方法など、さまざまな方法で登録できます。バッチファイルにもコマンドを含めることができます。

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
dbctrs16.dll	SQL Anywhere パフォーマンスモニタのカウンタ
dbodbc16.dll	SQL Anywhere ODBC ドライバ
dboledb16.dll	SQL Anywhere OLE DB プロバイダ
dboledba16.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 dbodbc16.dll
```

外部環境のサポートの配備

次の各表は、SQL Anywhere で外部呼び出しをサポートするために配備する必要があるコンポーネントをまとめたものです。

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	*dbexternc16.exe*	*dbexternc16*	*dbexternc16*
ブリッジ	*dbextenv16.dll*	*libdbextenv16_r.so*	*libdbextenv16_r.dylib*
SQL Anywhere C API のランタイム	*dbcapi.dll*	*libdbcapi_r.so*	*libdbcapi_r.dylib*

Copyright © 2013, SAP AG or an SAP affiliate company.

コンポーネント	Windows	Linux/UNIX	Mac OS X
DBLIB	*dblib16.dll*	*libdblib16_r.so*	*libdblib16_r.dylib*

Embedded SQL アプリケーションに必要な追加ファイルについては、「Embedded SQL クライアントの配備」992 ページを参照してください。

ODBC アプリケーションに必要な追加ファイルについては、「ODBC クライアントの配備」982 ページ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)	*dbjdbc16.dll*	*libdbjdbc16.so*	*libdbjdbc16.dylib*

JDBC アプリケーションに必要な追加ファイルについては、「JDBC クライアントの配備」994 ページ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 より)	なし	なし
SQL Anywhere .NET 3.5 または 4.0 プロバイダ	(Sybase より)	なし	なし
.NET CLR のブリッジ	*dbextclr16.exe*	なし	なし
ブリッジ	*dbextenv16.dll*	なし	なし
.NET CLR のサポート	*dbclrenv16.dll*	なし	なし
DBLIB	*dblib16.dll*	なし	なし

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*

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01035

コンポーネント	Windows	Linux/UNIX	Mac OS X
Perl のランチャ	*perlenv.pl*	*perlenv.pl*	*perlenv.pl*
ブリッジ	*dbextenv16.dll*	*libdbextenv16_r.so*	*libdbextenv16_r.dylib*
SQL Anywhere C API のランタイム	*dbcapi.dll*	*libdbcapi_r.so*	*libdbcapi_r.dylib*
DBLIB	*dblib16.dll*	*libdblib16_r.so*	*libdblib16_r.dylib*

Perl アプリケーションに必要な追加ファイルについては、「DBD::SQLAnywhere」649 ページ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/UNIX	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	*dbextenv16.dll*	*libdbextenv16_r.so*	*libdbextenv16_r.dylib*
PHP 5.1.x の外部呼び出し	*php-5.1.[1-*]_sqlanywhere_extenv16.dll*	*php-5.1.[1-*]_sqlanywhere_extenv16_r.so* またはソースコードからビルド	ソースコードからビルド
PHP 5.2.x の外部呼び出し	*php-5.2.[0-*]_sqlanywhere_extenv16.dll*	*php-5.2.[0-*]_sqlanywhere_extenv16_r.so* またはソースコードからビルド	ソースコードからビルド
PHP 5.3.x の外部呼び出し	*php-5.3.[0-*]_sqlanywhere_extenv16.dll*	*php-5.3.[0-*]_sqlanywhere_extenv16_r.so* またはソースコードからビルド	ソースコードからビルド
PHP 5.4.x の外部呼び出し	*php-5.4.[0-*]_sqlanywhere_extenv16.dll*	*php-5.4.[0-*]_sqlanywhere_extenv16_r.so* またはソースコードからビルド	ソースコードからビルド
DBLIB (スレッド)	*dblib16.dll*	*libdblib16_r.so*	*libdblib16_r.dylib*

Copyright © 2013, SAP AG or an SAP affiliate company.

コンポーネント	Windows	Linux/UNIX	Mac OS X
スレッドサポートライブラリ	なし	*libdbtasks16_r.so*	*libdbtasks16_r.dylib*

Windows バージョンの PHP 外部環境サポートファイルは、32 ビットのみです。これらのファイルを Windows コンピュータにインストールするには、SQL Anywhere インストールの一部として 32 ビットサーバコンポーネントを選択します。

PHP アプリケーションに必要な追加ファイルについては、「SQL Anywhere PHP 拡張」667 ページを参照してください。

参照

- 「ESQL 外部環境と ODBC 外部環境」623 ページ
- 「Java 外部環境」632 ページ
- 「CLR 外部環境」620 ページ
- 「PERL 外部環境」638 ページ
- 「PHP 外部環境」642 ページ

暗号化の配備

次の表に、SQL Anywhere で暗号化機能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	*dbserv16.dll* *libdbserv16_r.so*	付属 [1]
データベースの暗号化	FIPS 認定の AES	*dbfips16.dll*	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	RSA	*dbrsa16.dll* *libdbrsa16.so* *libdbrsa16.dylib* *libdbrsa16_r.dylib*	付属 [1]
トランスポートレイヤセキュリティ	FIPS 認定の RSA	*dbfips16.dll*、*sbgse2.dll*	別途ライセンスが必要 [2]

[1] AES および RSA による暗号化は SQL Anywhere 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。

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01037

[2] FIPS 認定テクノロジを使用した暗号化ソフトウェアは、別途注文する必要があります。

組み込みデータベースアプリケーションの配備

この項では、アプリケーションとデータベースの両方が同じコンピュータ上に置かれる組み込みデータベースアプリケーションの配備について説明します。

組み込みデータベースアプリケーションには、次に示すものが含まれます。

- **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を指します。
- **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。

参照

- 「データベースサーバの配備」1027 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」972 ページ

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 (*dblgen16.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することをおすすめします。

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) をアプリケーションとともに配備する必要がある場合は、ユーティリティの実行プログラムとともに次の追加ファイルも必要です。

説明	Windows	Linux/UNIX	Mac OS X
データベースツールライブラリ	*dbtool16.dll*	*libdbtool16_r.so*、*libdbtasks16_r.so*	*libdbtool16_r.dylib*、*libdbtasks16_r.dylib*

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

説明	Windows	Linux/UNIX	Mac OS X
インタフェースライブラリ	*dblib16.dll*	*libdblib16_r.so*	*libdblib16_r.dylib*
言語リソースライブラリ	*dblg[LL]16.dll*	*dblg[LL]16.res*	*dblgen16.res*
[接続] ウィンドウ	*dbcon16.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*	なし

説明

- 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**、**jp** 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、*libdbtasks16.so* と *libdblib16.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、*libdbtasks16.dylib* と *libdblib16.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] ウィンドウを表示する必要がある場合は、**[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 (*dbcon16.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、「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」で説明しているファイルが必要です。

- dbeng16 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 (dbinit) が必要です。これは、他のデータベースサーバが実行されていないときに、ローカルコンピュータの Sybase Central からデータベースを作成する場合にも必要です。

参照

- 「データベースサーバの配備」1027 ページ

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01039

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

バージョン 9.0 以前のデータベースを現在のフォーマットに変換できる機能がアプリケーションに必要な場合は、データベースアンロードユーティリティ (dbunload) とともに次の追加ファイルも必要です。

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[en].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

Copyright © 2013, SAP AG or an SAP affiliate company.

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の配備」1038 ページで説明しているファイルも必要です。

SQL Remote の配備

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	*dbencod16.dll*	*libdbencod16_r.so.1*	*libdbencod16_r.dylib*
FILE メッセージリンクライブラリ [1]	*dbfile16.dll*	*libdbfile16_r.so.1*	*libdbfile16_r.dylib*
FTP メッセージリンクライブラリ [1]	*dbftp16.dll*	*libdbftp16_r.so.1*	*libdbftp16_r.dylib*
言語リソースライブラリ	*dblg[LL]16.dll*	*dblg[LL]16.res*	*dblg[LL]16.res*

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01041

説明	Windows	Linux/Solaris	Mac OS X
インタフェースライブラリ	*dblib16.dll*	*libdblib16_r.so.1*	*libdblib16_r.dylib*
SMTP メッセージリンクライブラリ [1]	*dbsmtp16.dll*	*libdbsmtp16_r.so.1*	*libdbsmtp16_r.dylib*
データベースツールライブラリ	*dbtool16.dll*	*libdbtool16_r.so.1*	*libdbtool16_r.dylib*
スレッドサポートライブラリ	なし	*libdbtasks16_r.so.1*	*libdbtasks16_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**、**jp** 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することをおすすめします。

Copyright © 2013, SAP AG or an SAP affiliate company.

索引

記号

dbsrv16.lic
データベースサーバの配備, 1027
.NET
SQL Anywhere .NET データプロバイダの使用, 41
データ制御, 87
配備, 972
.NET API
説明, 41
.NET データプロバイダ
dbdata dll, 73
Entity Framework のサポート, 66
iAnywhere.Data.SQLAnywhere プロバイダ, 42
POOLING オプション, 46
Simple コードサンプルの使用, 78
Table Viewer コードサンプルの使用, 82
エラー処理, 65
機能, 42
サポートされている言語, 41
サポートされるバージョン, 41
参照の追加, 44
サンプルプロジェクトの実行, 43
時間値の取得, 62
システムの稼働条件, 72
ストアドプロシージャの実行, 62
接続プーリング, 46
説明, 41
ソースコードのプロバイダクラスを参照する, 44
データの更新, 47
データの削除, 47
データの挿入, 47
データへのアクセス, 47
データベースへの接続, 45
登録, 73
トランザクション処理, 64
トレースのサポート, 74
配備, 72
配備に必要なファイル, 72
例外処理, 65
.NET データプロバイダを使用したアプリケーションの開発
説明, 41
.NET データベースのプログラミングインタフェース
チュートリアル, 87
@HttpMethod
HTTP ヘッダへのアクセス, 768
@HttpQueryString
HTTP ヘッダへのアクセス, 768
@HttpStatus
HTTP ヘッダへのアクセス, 768
@HttpURI
HTTP ヘッダへのアクセス, 768
@HttpVersion
HTTP ヘッダへのアクセス, 768
2 フェーズコミット
3 層コンピューティング, 856
Open Client, 747
分散トランザクションの管理, 857
3 層コンピューティング
EAServer, 858
Microsoft Transaction Server, 858
アーキテクチャ, 855
説明, 855
分散トランザクション, 856
分散トランザクションコーディネーター, 858
リソースディスペンサー, 857
リソースマネージャ, 857
64 ビット
ODBC, 387
-d オプション
SQL プリプロセッサユーティリティ (sqlpp), 471
-e オプション
SQL プリプロセッサユーティリティ (sqlpp), 471
-gn オプション
スレッド, 417
-h オプション
SQL プリプロセッサユーティリティ (sqlpp), 471
-install オプション
SetupVSPackage, 972
-k オプション
SQL プリプロセッサユーティリティ (sqlpp), 471
-m オプション
SQL プリプロセッサユーティリティ (sqlpp), 471
-n オプション

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01043

SQL プリプロセッサユーティリティ (sqlpp), 471
-o オプション
SQL プリプロセッサユーティリティ (sqlpp), 471
-q オプション
SQL プリプロセッサユーティリティ (sqlpp), 471
-r オプション
SQL プリプロセッサユーティリティ (sqlpp), 471
-salocation オプション
SetupVSPackage, 972
-s オプション
SQL プリプロセッサユーティリティ (sqlpp), 471
-uninstall オプション
SetupVSPackage, 975
-u オプション
SQL プリプロセッサユーティリティ (sqlpp), 471
-w オプション
SQL プリプロセッサユーティリティ (sqlpp), 471
-x オプション
SQL プリプロセッサユーティリティ (sqlpp), 471
-z オプション
SQL プリプロセッサユーティリティ (sqlpp), 471

A

a_backup_db 構造体 [データベースツール API]
説明, 885
a_change_log 構造体 [データベースツール API]
説明, 889
a_create_db 構造体 [データベースツール API]
説明, 892
a_db_info 構造体 [データベースツール API]
説明, 897
a_db_version_info 構造体 [データベースツール API]
説明, 901
a_dblic_info 構造体 [データベースツール API]
説明, 902
a_dbtools_info 構造体 [データベースツール API]
説明, 904
a_log_file_info 構造体 [データベースツール API]
説明, 904
a_name 構造体 [データベースツール API]
説明, 905
a_remote_sql 構造体 [データベースツール API]
説明, 905
a_sqlany_bind_param_info 構造体 [SQL Anywhere C API]
説明, 593
a_sqlany_bind_param 構造体 [SQL Anywhere C API]
説明, 593
a_sqlany_column_info 構造体 [SQL Anywhere C API]
説明, 594
a_sqlany_data_direction 列挙体 [SQL Anywhere C API]
説明, 590
a_sqlany_data_info 構造体 [SQL Anywhere C API]
説明, 595
a_sqlany_data_type 列挙体 [SQL Anywhere C API]
説明, 590
a_sqlany_data_value 構造体 [SQL Anywhere C API]
説明, 596
a_sqlany_native_type 列挙体 [SQL Anywhere C API]
説明, 591
a_sync_db 構造体 [データベースツール API]
説明, 916
a_syncpub 構造体 [データベースツール API]
説明, 929
a_sysinfo 構造体 [データベースツール API]
説明, 930
a_table_info 構造体 [データベースツール API]
説明, 931
a_translate_log 構造体 [データベースツール API]
説明, 932
a_truncate_log 構造体 [データベースツール API]
説明, 938
a_validate_db 構造体 [データベースツール API]
説明, 940
Abort プロパティ
SARowsCopiedEventArgs クラス [SQL Anywhere .NET API], 310
Accept
HTTP ヘッダへのアクセス, 768
Accept-Charset
HTTP ヘッダへのアクセス, 768

Copyright © 2013, SAP AG or an SAP affiliate company.

AcceptCharset オプション
例, 779
Accept-Encoding
HTTP ヘッダへのアクセス, 768
Accept-Language
HTTP ヘッダへのアクセス, 768
access-bridge.jar
Java ベースの管理ツールの配備, 1011
accessibility.properties
Java ベースの管理ツールの配備, 1011
ActiveX Data Objects
説明, 344
addBatch
PreparedStatement クラス, 445
Statement クラス, 440
AddRange メソッド
SAParameterCollection クラス [SQL Anywhere .NET API], 295
addShutdownHook
Java VM シャットダウンフック, 420
AddWithValue メソッド
SAParameterCollection クラス [SQL Anywhere .NET API], 296
Add メソッド
SABulkCopyColumnMappingCollection クラス [SQL Anywhere .NET API], 123
SAParameterCollection クラス [SQL Anywhere .NET API], 291
ADO
Command オブジェクト, 345
Connection オブジェクト, 344
Recordset オブジェクト, 347
Recordset オブジェクトとカーソルタイプ, 348
アプリケーションでの SQL 文の使用, 1
カーソル, 33
カーソルタイプ, 15
カーソルによるデータの更新, 349
クエリ, 347
更新, 349
コマンド, 345
接続, 344
説明, 344
トランザクション, 350
プログラミングの概要, 343
ADO.NET
アプリケーションでの SQL 文の使用, 1
オートコミットの動作の制御, 37
オートコミットモード, 36
カーソルのサポート, 32
準備文, 3
説明, 41
配備, 972
ADO.NET API
説明, 41
alloc_sqlda_noind 関数
説明, 533
alloc_sqlda 関数
説明, 532
allowPasswordsInFavorites
設定可能なオプション, 1022
All プロパティ
SACommLinksOptionsBuilder クラス [SQL Anywhere .NET API], 167
ALTER EXTERNAL ENVIRONMENT JAVA
例, 409
an_erase_db 構造体 [データベースツール API]
説明, 942
an_extfn_api
外部関数呼び出しインタフェース, 606
an_extfn_result_set_column_data
外部関数呼び出しインタフェース, 611
an_extfn_result_set_column_info
外部関数呼び出しインタフェース, 610
an_extfn_result_set_info
外部関数呼び出しインタフェース, 609
an_extfn_value
外部関数呼び出しインタフェース, 607
an_unload_db 構造体 [データベースツール API]
説明, 943
an_upgrade_db 構造体 [データベースツール API]
説明, 953
ANSI ODBC ドライバ
UNIX, 372
Apache
PHP 拡張の選択, 998
インストール, 996
apache_files.txt
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
apache_license_1.1.txt
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
apache_license_2.0.txt
Mac OS X での配備, 1016

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01045

UNIX での配備, 1016
Windows での配備, 1007
API
ADO.NET, 41
ADO API, 343
C API, 565
JDBC, 423
ODBC API, 365
OLE DB API, 343
Perl DBD::SQLAnywhere API, 649
PHP, 682
Python データベース API, 659
Ruby API, 713
Sybase Open Client API, 741
AppInfo プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 197
ARRAY 句
FETCH 文の使用, 519
asensitive カーソル
概要, 17
更新の例, 20
説明, 24
例の削除, 18
ASP.NET
SQL Anywhere ASP.NET プロバイダの使用, 97
SQL Anywhere ASP.NET プロバイダの登録, 99
Sybase.com で入手できるホワイトペーパー, 97
接続文字列の登録, 99
データベースへのプロバイダスキーマの追加, 98
プロバイダ、説明, 97
ASP.NET プロバイダを使用したアプリケーション開発
説明, 97
autoCommit オプション
設定可能なオプション, 1022
AUTOINCREMENT
挿入された最新のローの検索, 12
autoRefetch
設定可能なオプション, 1022
AutoStart プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 197
AutoStop プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 197

B

BatchSize プロパティ
SABulkCopy クラス [SQL Anywhere .NET API], 112
BatchUpdateException
JDBC, 440
batik-awt-util.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
batik-bridge.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
batik-codec.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
batik-css.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
batik-dom.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
batik-ext.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
batik-extension.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
batik-gui-util.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
batik-gvt.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
batik-parser.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
batik-script.jar

Copyright © 2013, SAP AG or an SAP affiliate company.

Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
batik-svg-dom.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
batik-svggen.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
batik-swing.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
batik-transcoder.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
batik-util.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
batik-xml.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
BeginExecuteNonQuery メソッド
SACommand クラス [SQL Anywhere .NET API], 134
BeginExecuteReader メソッド
SACommand クラス [SQL Anywhere .NET API], 136
BeginTransaction メソッド
SAConnection クラス [SQL Anywhere .NET API], 172
BIGINT データ型
Embedded SQL, 490
BIT データ型
Embedded SQL, 490
BLOB
Embedded SQL, 523
ESQL での取得, 524, 525
ESQL での送信, 526, 527
BroadcastListener プロパティ
SATcpOptionsBuilder クラス [SQL Anywhere .NET API], 321
Broadcast プロパティ
SATcpOptionsBuilder クラス [SQL Anywhere .NET API], 321
BulkCopyTimeout プロパティ
SABulkCopy クラス [SQL Anywhere .NET API], 112
Bulk-Library
説明, 742

C

C_ESQL32 キーワード
外部環境, 624
C_ESQL64 キーワード
外部環境, 624
C_ODBC32 キーワード
外部環境, 624
C_ODBC64 キーワード
外部環境, 624
C#
.NET データプロバイダでのサポート, 41
C++ アプリケーション
dbtools, 861
Embedded SQL, 469
CALL 文
Embedded SQL, 528
Cancel メソッド
SACommand クラス [SQL Anywhere .NET API], 140
CanCreateDataSourceEnumerator プロパティ
SAFactory クラス [SQL Anywhere .NET API], 266
C API
プログラミングの概要, 565
CD-ROM
データベースの配備, 1033
chained オプション
JDBC, 438
ChangeDatabase メソッド
SAConnection クラス [SQL Anywhere .NET API], 174
ChangePassword メソッド
SAConnection クラス [SQL Anywhere .NET API], 175
CharsetConversion オプション
例, 779
Charset プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 197

checkForUpdates
設定可能なオプション, 1022
Class.forName メソッド
SQL Anywhere JDBC 4.0 ドライバのロード, 427
CLASSPATH 環境変数
jConnect, 428
設定, 434
データベース内の Java の使用, 408
ClearAllPools メソッド
SAConnection クラス [SQL Anywhere .NET API], 175
clearBatch
Statement クラス, 440
ClearPool メソッド
SAConnection クラス [SQL Anywhere .NET API], 175
Clear メソッド
SAParameterCollection クラス [SQL Anywhere .NET API], 297
ClickOnce
.NET データプロバイダのアンマネージ DLL のインストール, 974
Client-Library
Sybase Open Client, 742
CLIENTPORT 句
指定, 794
ClientPort プロパティ
SATcpOptionsBuilder クラス [SQL Anywhere .NET API], 321
CLOSE 文
Embedded SQL でのカーソルの使用, 517
close メソッド
Python, 662
Close メソッド
SABulkCopy クラス [SQL Anywhere .NET API], 109
SAConnection クラス [SQL Anywhere .NET API], 176
SADataReader クラス [SQL Anywhere .NET API], 228
CLR
外部環境, 620
CLR キーワード
外部環境, 620
CodeXchange
サンプル, 751
ColumnMappings プロパティ
SABulkCopy クラス [SQL Anywhere .NET API], 113
Columns フィールド
SAMetaDataCollectionNames クラス [SQL Anywhere .NET API], 271
Command ADO オブジェクト
ADO, 345
CommandText プロパティ
SACommand クラス [SQL Anywhere .NET API], 149
CommandTimeout プロパティ
SACommand クラス [SQL Anywhere .NET API], 149
CommandType プロパティ
SACommand クラス [SQL Anywhere .NET API], 149
Command プロパティ
SARowUpdatedEventArgs クラス [SQL Anywhere .NET API], 312
SARowUpdatingEventArgs クラス [SQL Anywhere .NET API], 314
CommBufferSize プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 198
commitOnExit
設定可能なオプション, 1022
CommitTrans ADO メソッド
ADO プログラミング, 350
データの更新, 350
COMMIT 文
JDBC, 438
カーソル, 39
commit メソッド
Python, 663
Commit メソッド
SATransaction クラス [SQL Anywhere .NET API], 325
CommLinks プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 198
CompressionThreshold プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 198
Compress プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 198
Connection
HTTP ヘッダへのアクセス, 768

Copyright © 2013, SAP AG or an SAP affiliate company.

CONNECTION_PROPERTY 関数
例, 778
Connection ADO オブジェクト
ADO, 344
ADO プログラミング, 350
ConnectionLifetime プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 198
ConnectionName プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 199
ConnectionPool プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 199
ConnectionReset プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 199
ConnectionString プロパティ
SACommLinksOptionsBuilder クラス [SQL Anywhere .NET API], 168
SAConnection クラス [SQL Anywhere .NET API], 186
ConnectionTimeout プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 199
SAConnection クラス [SQL Anywhere .NET API], 187
Connection プロパティ
SACommand クラス [SQL Anywhere .NET API], 150
SAServerSideConnection クラス [SQL Anywhere .NET API], 315
SATransaction クラス [SQL Anywhere .NET API], 327
connect メソッド
Python, 662
ContainsKey メソッド
SAConnectionStringBuilderBase クラス [SQL Anywhere .NET API], 210
Contains メソッド
SABulkCopyColumnMappingCollection クラス [SQL Anywhere .NET API], 127
SAParameterCollection クラス [SQL Anywhere .NET API], 297
cookie
作成, 773
セッション管理, 775
CopyTo メソッド
SABulkCopyColumnMappingCollection クラス [SQL Anywhere .NET API], 127
SAErrorCollection クラス [SQL Anywhere .NET API], 256
SAParameterCollection クラス [SQL Anywhere .NET API], 298
Count プロパティ
SAErrorCollection クラス [SQL Anywhere .NET API], 257
SAParameterCollection クラス [SQL Anywhere .NET API], 302
CreateCommandBuilder メソッド
SAFactory クラス [SQL Anywhere .NET API], 263
CreateCommand メソッド
SAConnection クラス [SQL Anywhere .NET API], 176
SAFactory クラス [SQL Anywhere .NET API], 263
CreateConnectionStringBuilder メソッド
SAFactory クラス [SQL Anywhere .NET API], 264
CreateConnection メソッド
SAFactory クラス [SQL Anywhere .NET API], 263
CreateDataAdapter メソッド
SAFactory クラス [SQL Anywhere .NET API], 264
CreateDataSourceEnumerator メソッド
SAFactory クラス [SQL Anywhere .NET API], 265
CreateParameter メソッド
SACommand クラス [SQL Anywhere .NET API], 140
SAFactory クラス [SQL Anywhere .NET API], 265
使用, 3
CreatePermission メソッド
SAFactory クラス [SQL Anywhere .NET API], 265
SAPermissionAttribute クラス [SQL Anywhere .NET API], 308
CREATE PROCEDURE 文
Embedded SQL, 528
CS_CSR_ABS
Open Client でサポートされていない, 747
CS_CSR_FIRST
Open Client でサポートされていない, 747

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01049

CS_CSR_LAST
Open Client でサポートされていない, 747
CS_CSR_PREV
Open Client でサポートされていない, 747
CS_CSR_REL
Open Client でサポートされていない, 747
CS_DATA_BOUNDARY
Open Client でサポートされていない, 747
CS_DATA_SENSITIVITY
Open Client でサポートされていない, 747
CS_PROTO_DYNPROC
Open Client でサポートされていない, 747
CS_REG_BCP
Open Client でサポートされていない, 747
CS_REG_NOTIF
Open Client でサポートされていない, 747
CS-Library
説明, 742
ct_command 関数
Open Client での結果の記述, 747
Open Client での文の実行, 745
ct_cursor 関数
Open Client, 745
ct_dynamic 関数
Open Client, 745
ct_results 関数
Open Client, 747
ct_send 関数
Open Client, 747
C プログラミング言語
Embedded SQL アプリケーション, 469
データ型, 490

D

DataAdapter
説明, 47
データの更新, 53
データの削除, 53
データの取得, 55, 56
データの挿入, 53
プライマリキー値の取得, 60
DataAdapter プロパティ
SACommandBuilder クラス [SQL Anywhere .NET API], 163
DatabaseFile プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 200
DatabaseKey プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 200
DatabaseName プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 200
DatabaseSwitches プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 200
Database プロパティ
SAConnection クラス [SQL Anywhere .NET API], 187
DataSet
SQL Anywhere .NET データプロバイダ, 53
DataSourceInformation フィールド
SAMetaDataCollectionNames クラス [SQL Anywhere .NET API], 271
DataSourceName プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 201
DataSource プロパティ
SAConnection クラス [SQL Anywhere .NET API], 188
DataTypes フィールド
SAMetaDataCollectionNames クラス [SQL Anywhere .NET API], 272
DATETIME データ型
Embedded SQL, 490
Open Client の変換, 744
DB_ACTIVE_CONNECTION
db_find_engine 関数, 541
DB_BACKUP_CLOSE_FILE パラメータ
説明, 533
DB_BACKUP_END パラメータ
説明, 533
DB_BACKUP_INFO_CHKPT_LOG パラメータ
説明, 533
DB_BACKUP_INFO_PAGES_IN_BLOCK パラメータ
説明, 533
DB_BACKUP_INFO パラメータ
説明, 533
DB_BACKUP_OPEN_FILE パラメータ
説明, 533
DB_BACKUP_PARALLEL_READ パラメータ
説明, 533
DB_BACKUP_PARALLEL_START パラメータ
説明, 533

DB_BACKUP_READ_PAGE パラメータ
説明, 533
DB_BACKUP_READ_RENAME_LOG パラメータ
説明, 533
DB_BACKUP_START パラメータ
説明, 533
db_backup 関数
dbbackup ユーティリティ, 531
説明, 533
DB_CALLBACK_CONN_DROPPED コールバックパラメータ
説明, 547
DB_CALLBACK_DEBUG_MESSAGE コールバックパラメータ
説明, 547
DB_CALLBACK_FINISH コールバックパラメータ
説明, 547
DB_CALLBACK_MESSAGE コールバックパラメータ
説明, 548
DB_CALLBACK_START コールバックパラメータ
説明, 547
DB_CALLBACK_VALIDATE_FILE_TRANSFER コールバックパラメータ
説明, 549
DB_CALLBACK_WAIT コールバックパラメータ
説明, 548
DB_CAN_MULTI_CONNECT
db_find_engine 関数, 541
DB_CAN_MULTI_DB_NAME
db_find_engine 関数, 541
db_cancel_request 関数
説明, 538
要求管理, 531
db_change_char_charset 関数
説明, 539
db_change_nchar_charset 関数
説明, 539
DB_CLIENT
db_find_engine 関数, 541
DB_CONNECTION_DIRTY
db_find_engine 関数, 541
DB_DATABASE_SPECIFIED
db_find_engine 関数, 541
DB_ENGINE
db_find_engine 関数, 541
db_find_engine 関数
説明, 540
db_fini_dll
呼び出し, 479
db_fini 関数
説明, 541
db_get_property 関数
説明, 542
db_init_dll
呼び出し, 479
db_init 関数
説明, 543
db_is_working 関数
説明, 543
要求管理, 531
db_locate_servers_ex 関数
説明, 545
db_locate_servers 関数
説明, 544
DB_LOOKUP_FLAG_ADDRESS_INCLUDES_PORT
説明, 545
DB_LOOKUP_FLAG_DATABASES
説明, 545
DB_LOOKUP_FLAG_NUMERIC
説明, 545
DB_NO_DATABASES
db_find_engine 関数, 541
DB_PROP_CLIENT_CHARSET
使用法, 542
DB_PROP_DBLIB_VERSION
使用法, 542
DB_PROP_SERVER_ADDRESS
使用法, 542
db_register_a_callback 関数
説明, 546
要求管理, 531
db_start_database 関数
説明, 550
db_start_engine 関数
説明, 550
db_stop_database 関数
説明, 552
db_stop_engine 関数
説明, 552
db_string_connect 関数
説明, 553
db_string_disconnect 関数

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01051

説明, 554
db_string_ping_server 関数
説明, 555
db_time_change 関数
説明, 555
DBBackup メソッド [データベースツール API]
説明, 869
dbcapi.dll
PHP クライアントの配備, 996
PHP サポートモジュール, 644
配備, 1034
DBChangeLogName メソッド [データベースツール API]
説明, 869
dbcis16.dll
データベースサーバの配備, 1027
dbclrenv16.dll
配備, 1035
dbcon16.dll
Embedded SQL クライアントの配備, 992
ODBC クライアントの配備, 983
OLE DB クライアントの配備, 976
PHP クライアントの配備, 996
Windows での配備, 1007
データベースユーティリティの配備, 1038
dbconsole.exe
Windows での配備, 1007
dbconsole.ini
管理ツールの配備, 1004
DBConsole.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
dbconsole ユーティリティ
Linux、UNIX、Mac OS X での管理ツールの配備, 1019
Linux と UNIX での配備, 1014
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での管理ツールの配備, 1010
Windows での配備, 1005
配備, 1004
DBCreatedVersion メソッド [データベースツール API]
説明, 870
DBCreate メソッド [データベースツール API]
説明, 870
dbctrs16.dll
SQL Anywhere の配備, 1034
データベースサーバの配備, 1027
DBD::SQLAnywhere
Perl スクリプトの作成, 652
SQL 文の実行, 654
データベースへの接続, 653
ローの挿入, 656
説明, 649
複数の結果セットの処理, 655
dbdata.dll
SQL Anywhere .NET データプロバイダ, 73
dbelevate16.exe
DLL の登録, 1034
Windows での配備, 1007
dbencod16.dll
SQL Remote の配備, 1041
dbeng16
データベースサーバの配備, 1027
dbeng16.lic
データベースサーバの配備, 1027
DBErase メソッド [データベースツール API]
説明, 871
dbextclr16.exe
配備, 1035
dbextenv16.dll
PHP サポートモジュール, 644
配備, 1034
dbexternc16
外部呼び出し, 625
dbexternc16
配備, 1034
dbextf.dll
データベースサーバの配備, 1027
dbfile16.dll
SQL Remote の配備, 1041
dbfips16.dll
Embedded SQL クライアントの配備, 992
FIPS 認定の AES 暗号化, 1037
FIPS 認定の RSA 暗号化, 1037
PHP クライアントの配備, 996
dbftp16.dll
SQL Remote の配備, 1041
dbghelp.dll
Windows での配備, 1007
データベースサーバの配備, 1027
dbicu16.dll
JDBC クライアントの配備, 994
ODBC クライアントの配備, 983

PHP クライアントの配備, 996
Windows での配備, 1007
データベースサーバの配備, 1027
dbicudt16.dat
JDBC クライアントの配備, 994
ODBC クライアントの配備, 983
データベースサーバの配備, 1027
dbicudt16.dll
JDBC クライアントの配備, 994
ODBC クライアントの配備, 983
PHP クライアントの配備, 996
Windows での配備, 1007
データベースサーバの配備, 1027
DBInfoDump メソッド [データベースツール API]
説明, 872
DBInfoFree メソッド [データベースツール API]
説明, 872
DBInfo メソッド [データベースツール API]
説明, 871
dbinit.exe
Windows での配備, 1007
dbinit ユーティリティ
Mac OS X での配備, 1016
UNIX での配備, 1016
配備に関する考慮事項, 1039
dbisql.com
Windows での配備, 1007
dbisql.exe
Windows での配備, 1007
dbisql.ini
管理ツールの配備, 1004
dbisqlc ユーティリティ
制限付き機能, 1026
dbisqlc ユーティリティ
UNIX でサポートされる配備プラットフォーム, 1015
配備, 1026
dbisql ユーティリティ
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での管理ツールの配備, 1008
dbjdbc16.dll
JDBC クライアントの配備, 994
データベースサーバの配備, 1027
配備, 1035
dbjodbc16.dll
Windows での配備, 1007
データベースサーバの配備, 1027
dblgen16.dll
Embedded SQL クライアントの配備, 992
JDBC クライアントの配備, 994
ODBC クライアントの配備, 983
OLE DB クライアントの配備, 976
PHP クライアントの配備, 996
SQL Remote の配備, 1041
Windows での配備, 1007
データベースサーバの配備, 1027
データベースユーティリティの配備, 1038
dblgen16.dll レジストリエントリ
説明, 1030
dblgen16.res
JDBC クライアントの配備, 994
Mac OS X での配備, 1016
ODBC クライアントの配備, 983
SQL Remote の配備, 1041
UNIX での配備, 1016
データベースサーバの配備, 1027
データベースユーティリティの配備, 1038
DBLIB
インタフェースライブラリ, 469
動的ロード, 479
dblib16.dll
Embedded SQL クライアントの配備, 992
PHP クライアントの配備, 996
SQL Remote の配備, 1041
Windows での配備, 1007
データベースユーティリティの配備, 1038
DB-Library
説明, 742
DBLicense メソッド [データベースツール API]
説明, 873
DBLogFileInfo メソッド [データベースツール API]
説明, 873
dbodbc16_r.bundle
ODBC クライアントの配備, 983
dbodbc16.bundle
Mac OS X ODBC ドライバ, 370
ODBC クライアントの配備, 983
dbodbc16.dll
ODBC クライアントの配備, 983
SQL Anywhere の配備, 1034
Windows での配備, 1007
データベースサーバの配備, 1027
リンク, 367
dbodbc16.lib

Windows Mobile ODBC インポートライブラリ, 368
dboftsp.dll
データベースユーティリティの配備, 1038
dboledb16.dll
OLE DB クライアントの配備, 976
SQL Anywhere の配備, 1034
dboledba16.dll
OLE DB クライアントの配備, 976
SQL Anywhere の配備, 1034
dbosrv16.exe ユーティリティ
構文, 466
dbostop.exe ユーティリティ
構文, 467
DbProviderFactory
登録, 73
dbput16.dll
Windows での配備, 1007
DBRemoteSQL メソッド [データベースツール API]
説明, 873
dbremote ユーティリティ
SQL Remote の配備, 1041
dbrmt.h
説明, 861
dbrsa16.dll
PHP クライアントの配備, 996
RSA 暗号化, 1037
dbrsakp16.dll
JDBC クライアントの配備, 995
Open Client の配備, 1003
データベースサーバの配備, 1027
dbscript16.dll
データベースサーバの配備, 1027
dbserv16.dll
AES 暗号化, 1037
データベースサーバの配備, 1027
dbsmtp16.dll
SQL Remote の配備, 1041
dbsrv16
データベースサーバの配備, 1027
DBSynchronizeLog メソッド [データベースツール API]
説明, 874
dbtool16.dll
SQL Remote の配備, 1041
Windows Mobile, 861
Windows での配備, 1007
説明, 861
データベースユーティリティの配備, 1038
dbtools.h
説明, 861
dbtools.h ヘッダファイル
SQL Anywhere データベースツール C API リファレンス, 868
DBToolsFini メソッド [データベースツール API]
説明, 874
DBToolsInit メソッド [データベースツール API]
説明, 875
DBToolsVersion メソッド [データベースツール API]
説明, 875
DBTools インタフェース
DBTools 関数の呼び出し, 863
概要, 861
起動, 862
終了, 862
使用, 861
初期化, 862
説明, 861
ファイナライズ, 862
プログラム例, 866
リターンコード, 956
DBTranslateLog メソッド [データベースツール API]
説明, 876
DBTruncateLog メソッド [データベースツール API]
説明, 876
DbType プロパティ
SAParameter クラス [SQL Anywhere .NET API], 284
DBUnload メソッド [データベースツール API]
説明, 877
dbunload ユーティリティ
配備に関する考慮事項, 1039
バージョン 9 以前のデータベースの配備, 1040
dbunlspt
バージョン 9 以前のデータベースの配備, 1040
DBUpgrade メソッド [データベースツール API]
説明, 877
dbupgrad ユーティリティ
jConnect メタデータサポートのインストール, 429
dbusde.dll
バージョン 9 以前のデータベースの配備, 1040

dbusde.res
バージョン 9 以前のデータベースの配備, 1040
dbusen.dll
バージョン 9 以前のデータベースの配備, 1040
dbusen.res
バージョン 9 以前のデータベースの配備, 1040
dbuses.dll
バージョン 9 以前のデータベースの配備, 1040
dbusfr.dll
バージョン 9 以前のデータベースの配備, 1040
dbusfr.res
バージョン 9 以前のデータベースの配備, 1040
dbusit.dll
バージョン 9 以前のデータベースの配備, 1040
dbusja.dll
バージョン 9 以前のデータベースの配備, 1040
dbusja.res
バージョン 9 以前のデータベースの配備, 1040
dbusko.dll
バージョン 9 以前のデータベースの配備, 1040
dbuslt.dll
バージョン 9 以前のデータベースの配備, 1040
dbuspl.dll
バージョン 9 以前のデータベースの配備, 1040
dbuspt.dll
バージョン 9 以前のデータベースの配備, 1040
dbusru.dll
バージョン 9 以前のデータベースの配備, 1040
dbustw.dll
バージョン 9 以前のデータベースの配備, 1040
dbusuk.dll
バージョン 9 以前のデータベースの配備, 1040
dbuszh.res
バージョン 9 以前のデータベースの配備, 1040
DBValidate メソッド [データベースツール API]
説明, 878
debugger.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
DECIMAL データ型
ESQL, 490
DECL_BIGINT マクロ
説明, 490
DECL_BINARY マクロ
説明, 490
DECL_BIT マクロ
説明, 490
DECL_DATETIME マクロ
説明, 490
DECL_DECIMAL マクロ
説明, 490
DECL_FIXCHAR マクロ
説明, 490
DECL_LONGBINARY マクロ
説明, 490
DECL_LONGNVARCHAR マクロ
説明, 490
DECL_LONGVARCHAR マクロ
説明, 490
DECL_NCHAR マクロ
説明, 490
DECL_NFIXCHAR マクロ
説明, 490
DECL_NVARCHAR マクロ
説明, 490
DECL_UNSIGNED_BIGINT マクロ
説明, 490
DECL_VARCHAR マクロ
説明, 490
DECLARE セクション
説明, 489
DECLARE 文
Embedded SQL でのカーソルの使用, 517
DefaultDSNDir
ODBC FileDSN, 988
defaultShowMultipleResultSets
設定可能なオプション, 1022
defaultShowResultsForAllStatements
設定可能なオプション, 1022
DeleteCommand プロパティ
SADataAdapter クラス [SQL Anywhere .NET API], 220
DeleteDynamic メソッド
JDBCExample, 444
DeleteStatic メソッド
JDBCExample, 441
DELETE 文
JDBC, 440
位置付け, 12
Deployment ウィザード
実行, 965
説明, 964
DeployUtility
.NET データプロバイダサンプルプロジェクト, 43

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01055

.NET データプロバイダのアンマネージ DLL のインストール, 974
Depth プロパティ
SADataReader クラス [SQL Anywhere .NET API], 248
DeriveParameters メソッド
SACommandBuilder クラス [SQL Anywhere .NET API], 156
DESCRIBE SELECT LIST 文
動的 SELECT 文, 505
DESCRIBE 文
DT_HAS_USERTYPE_INFO, 509
SQLDA フィールド, 508
sqllen フィールド, 509
sqltype フィールド, 509
動的 SELECT 文での使用, 505
複数の結果セット, 531
DesignTimeVisible プロパティ
SACommand クラス [SQL Anywhere .NET API], 150
DestinationColumn プロパティ
SABulkCopyColumnMapping クラス [SQL Anywhere .NET API], 119
DestinationOrdinal プロパティ
SABulkCopyColumnMapping クラス [SQL Anywhere .NET API], 119
DestinationTableName プロパティ
SABulkCopy クラス [SQL Anywhere .NET API], 113
Direction プロパティ
SAParameter クラス [SQL Anywhere .NET API], 284
disableExecuteAll
設定可能なオプション, 1022
DisableMultiRowFetch プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 201
disableResultsEditing
設定可能なオプション, 1022
DISH サービス
.NET チュートリアル, 839
JAX-WS チュートリアル, 846
SQL Anywhere Web クライアントチュートリアル, 830
コメント付け, 760
削除, 760
作成, 758
説明, 754
同種, 759
Dispose メソッド
SABulkCopy クラス [SQL Anywhere .NET API], 109
DLL
外部プロシージャの呼び出し, 602
ストアドプロシージャからの呼び出し, 601
配備, 1034
配備用に登録, 1034
複数の SQLCA, 502
DllMain
db_fini の呼び出し, 541
ODBC 関数の呼び出し, 382
DLL のエントリポイント
説明, 532
DLL の自己登録
SQL Anywhere の配備, 1034
DoBroadcast プロパティ
SATcpOptionsBuilder クラス [SQL Anywhere .NET API], 321
DSN-less 接続
ODBC の使用, 991
DT_BIGINT ESQL データ型
説明, 486
DT_BINARY ESQL データ型
説明, 487
DT_BIT ESQL データ型
説明, 486
DT_DATE ESQL データ型
説明, 486
DT_DECIMAL ESQL データ型
説明, 486
DT_DOUBLE ESQL データ型
説明, 486
DT_FIXCHAR ESQL データ型
説明, 486
DT_FLOAT ESQL データ型
説明, 486
DT_HAS_USERTYPE_INFO
DESCRIBE 文, 509
DT_INT ESQL データ型
説明, 486
DT_LONGBINARY ESQL データ型
説明, 488
DT_LONGNVARCHAR ESQL データ型
説明, 487
DT_LONGVARCHAR ESQL データ型
説明, 487

Copyright © 2013, SAP AG or an SAP affiliate company.

DT_NFIXCHAR ESQL データ型
説明, 486
DT_NSTRING ESQL データ型
説明, 486
DT_NVARCHAR ESQL データ型
説明, 487
DT_PROCEDURE_IN
使用, 530
DT_PROCEDURE_OUT
使用, 530
DT_SMALLINT ESQL データ型
説明, 486
DT_STRING ESQL データ型
説明, 486
DT_TIME ESQL データ型
説明, 486
DT_TIMESTAMP_STRUCT ESQL データ型
説明, 488
DT_TIMESTAMP ESQL データ型
説明, 486
DT_TINYINT ESQL データ型
説明, 486
DT_UNSBIGINT Embedded SQL データ型
説明, 486
DT_UNSINT ESQL データ型
, 486
DT_UNSSMALLINT ESQL データ型
説明, 486
DT_VARCHAR ESQL データ型
説明, 486
DT_VARIABLE ESQL データ型
説明, 488
DTC
3 層コンピューティング, 858
独立性レベル, 859
DTC 独立性レベル
説明, 859
DYNAMIC SCROLL カーソル
asensitive カーソル, 24
Embedded SQL, 34

E

EAServer
3 層コンピューティング, 858
Elevate プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 201
Embedded HTTP サーバ
説明, 457
Embedded SQL
DllMain からの db_fini の呼び出し, 541
FETCH FOR UPDATE, 31
SQL 文, 1
インポートライブラリ, 477
オートコミットの動作の制御, 37
オートコミットモード, 36
開発, 469
外部環境, 623
関数, 532
カーソル, 34
カーソルタイプ, 15
カーソルの使用, 517
カーソル例, 481
行番号, 471
クライアントの配備, 992
権限, 471
コンパイルとリンクの処理, 470
サンプルプログラム, 478
説明, 469
データのフェッチ, 515
動的カーソル, 483
文のまとめ, 561
ヘッダファイル, 477
ホスト変数, 489
文字列, 471
EnableFlush レジストリエントリ
データベースの破損の回避, 1029
EncryptedPassword プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 201
Encryption プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 202
EndExecuteNonQuery メソッド
SACommand クラス [SQL Anywhere .NET API], 141
EndExecuteReader メソッド
SACommand クラス [SQL Anywhere .NET API], 143
EnlistDistributedTransaction メソッド
SAConnection クラス [SQL Anywhere .NET API], 177
EnlistTransaction メソッド
SAConnection クラス [SQL Anywhere .NET API], 177

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01057

Enlist プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 202
Entity Framework
使用, 66
Entity Framework サポート
iAnywhere.Data.SQLAnywhere プロバイダ, 42
Errors プロパティ
SAException クラス [SQL Anywhere .NET API], 260
SAInfoMessageEventArgs クラス [SQL Anywhere .NET API], 268
ESQL
静的文, 503
動的文, 503
esqldll.c
説明, 479
EXEC SQL
Embedded SQL の開発, 479
executeBatch
PreparedStatement クラス, 445
Statement クラス, 440
executemany メソッド
Python, 664
ExecuteNonQuery メソッド
SACommand クラス [SQL Anywhere .NET API], 145
SACommand の使用, 50
ExecuteReader メソッド
ADO.NET 準備文, 3
SACommand クラス [SQL Anywhere .NET API], 146
SACommand クラス Simple サンプル, 81
SACommand クラス TableView サンプル, 86
SACommand の使用, 48
ExecuteScalar メソッド
SACommand クラス [SQL Anywhere .NET API], 147
SACommand の使用, 48
executeToolbarButtonSemantics
設定可能なオプション, 1022
executeUpdate
Statement クラス, 440
executeUpdate JDBC メソッド
使用, 4
EXECUTE 文
Embedded SQL のストアドプロシージャ, 528
使用, 503
execute メソッド
Python, 663
extfn_cancel
外部関数呼び出しインタフェース, 604
extfn_post_load_library
外部関数呼び出しインタフェース, 605
extfn_pre_unload_library
外部関数呼び出しインタフェース, 605
extfn_use_new_api
外部関数呼び出しインタフェース, 603

F

fastLauncherEnabled
設定可能なオプション, 1022
fetchall メソッド
Python, 663
FETCH FOR UPDATE
Embedded SQL, 31
ODBC, 31
FETCH 文
Embedded SQL でのカーソルの使用, 517
使用, 515
動的クエリ, 505
複数のロー, 519
ワイド, 519
FieldCount プロパティ
SADataReader クラス [SQL Anywhere .NET API], 249
FileDataSourceName プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 202
fill_s_sqlda 関数
説明, 556
fill_sqlda_ex 関数
説明, 557
fill_sqlda 関数
説明, 556
ForceStart 接続パラメータ
db_start_engine, 551
ForceStart プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 202
ForeignKeys フィールド
SAMetaDataCollectionNames クラス [SQL Anywhere .NET API], 272
free_filled_sqlda 関数
説明, 558

Copyright © 2013, SAP AG or an SAP affiliate company.

free_sqlda_noind 関数
説明, 559
free_sqlda 関数
説明, 558

G

get_piece
説明, 612
get_value 関数
使用, 617
説明, 612
getAutoCommit メソッド
JDBC, 438
GetBoolean メソッド
SADataReader クラス [SQL Anywhere .NET API], 229
GetBytes メソッド
SADataReader クラス [SQL Anywhere .NET API], 230
使用, 61
GetByte メソッド
SADataReader クラス [SQL Anywhere .NET API], 229
GetChars メソッド
SADataReader クラス [SQL Anywhere .NET API], 231
使用, 61
GetChar メソッド
SADataReader クラス [SQL Anywhere .NET API], 231
GetConnection メソッド
インスタンス, 438
GetDataSources メソッド
SADataSourceEnumerator クラス [SQL Anywhere .NET API], 252
GetDataTypeName メソッド
SADataReader クラス [SQL Anywhere .NET API], 233
GetData メソッド
SADataReader クラス [SQL Anywhere .NET API], 232
GetDateTimeOffset メソッド
SADataReader クラス [SQL Anywhere .NET API], 234
GetDateTime メソッド
SADataReader クラス [SQL Anywhere .NET API], 233
GetDecimal メソッド
SADataReader クラス [SQL Anywhere .NET API], 234
GetDeleteCommand メソッド
SACommandBuilder クラス [SQL Anywhere .NET API], 156
GetDouble メソッド
SADataReader クラス [SQL Anywhere .NET API], 235
GetEnumerator メソッド
SADataReader クラス [SQL Anywhere .NET API], 235
SAErrorCollection クラス [SQL Anywhere .NET API], 257
SAParameterCollection クラス [SQL Anywhere .NET API], 299
GetFieldType メソッド
SADataReader クラス [SQL Anywhere .NET API], 236
GetFillParameters メソッド
SADataAdapter クラス [SQL Anywhere .NET API], 220
GetFloat メソッド
SADataReader クラス [SQL Anywhere .NET API], 236
GetGuid メソッド
SADataReader クラス [SQL Anywhere .NET API], 237
GetInsertCommand メソッド
SACommandBuilder クラス [SQL Anywhere .NET API], 158
GetInt16 メソッド
SADataReader クラス [SQL Anywhere .NET API], 237
GetInt32 メソッド
SADataReader クラス [SQL Anywhere .NET API], 238
GetInt64 メソッド
SADataReader クラス [SQL Anywhere .NET API], 238
GetKeyword メソッド
SAConnectionStringBuilderBase クラス [SQL Anywhere .NET API], 211
GetName メソッド
SADataReader クラス [SQL Anywhere .NET API], 239
GetObjectData メソッド

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01059

SAException クラス [SQL Anywhere .NET API], 259
GetOrdinal メソッド
SADataReader クラス [SQL Anywhere .NET API], 239
GetSchemaTable メソッド
SADataReader クラス [SQL Anywhere .NET API], 240
SADataReader の使用, 50
GetSchema メソッド
SAConnection クラス [SQL Anywhere .NET API], 177
GetString メソッド
SADataReader クラス, 82
SADataReader クラス [SQL Anywhere .NET API], 242
GetTimeSpan メソッド
SADataReader クラス [SQL Anywhere .NET API], 243
使用, 62
GetUInt16 メソッド
SADataReader クラス [SQL Anywhere .NET API], 243
GetUInt32 メソッド
SADataReader クラス [SQL Anywhere .NET API], 244
GetUInt64 メソッド
SADataReader クラス [SQL Anywhere .NET API], 244
GetUpdateCommand メソッド
SACommandBuilder クラス [SQL Anywhere .NET API], 160
getUpdateCounts
BatchUpdateException, 440
GetUseLongNameAsKeyword メソッド
SACommLinksOptionsBuilder クラス [SQL Anywhere .NET API], 166
SAConnectionStringBuilderBase クラス [SQL Anywhere .NET API], 211
GetValues メソッド
SADataReader クラス [SQL Anywhere .NET API], 246
GetValue メソッド
SADataReader クラス [SQL Anywhere .NET API], 245
GNU コンパイラ
Embedded SQL のサポート, 476
GRANT 文
JDBC, 448

H

HasRows プロパティ
SADataReader クラス [SQL Anywhere .NET API], 249
HEADER 句
管理, 795
HKEY_CURRENT_USER
ユーザデータソース, 989
HKEY_LOCAL_MACHINE
Intel ストレージドライバの設定, 1029
ODBC ドライバ, 986
イベントログ, 1030
管理ツール, 1012
システムデータソース, 989
システムパス, 1013
Host
HTTP ヘッダへのアクセス, 768
Host プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 203
SATcpOptionsBuilder クラス [SQL Anywhere .NET API], 322
HTML サービス
Web クライアントのクイックスタート, 787
Web サーバのクイックスタート, 749
クイックスタート, 789
コメント付け, 760
削除, 760
作成, 756
説明, 754
HTTP_HEADER 関数
例, 767
HTTP_VARIABLE 関数
例, 767
HttpMethod
HTTP ヘッダへのアクセス, 768
HttpQueryString
HTTP ヘッダへのアクセス, 768
HTTPS
有効化, 751
HttpStatus
HTTP ヘッダへのアクセス, 768
HTTPS プロトコル
設定, 751
HttpURI

HTTP ヘッダへのアクセス, 768
HttpVersion
HTTP ヘッダへのアクセス, 768
HTTP システムプロシージャ
アルファベット順リスト, 780
HTTP プロトコル
設定, 751
有効化, 751
HTTP ヘッダ
アクセス, 768
HTTP 要求ヘッダ
アクセス, 768
管理, 795
HTTP 要求
構造, 822

I

iAnywhere.Data.SQLAnywhere
Entity Framework サポート, 42
iAnywhere.Data.SQLAnywhere.dll
.NET クライアントの配備, 972
iAnywhere.Data.SQLAnywhere.dll.config
.NET クライアントの配備, 972
iAnywhere.Data.SQLAnywhere.gac
.NET クライアントの配備, 972
iAnywhere.Data.SQLAnywhere.v3.5.dll
.NET クライアントの配備, 972
iAnywhere.Data.SQLAnywhere.v4.0.dll
.NET クライアントの配備, 972
iAnywhere.Data.SQLAnywhere ネームスペース
SQL Anywhere .NET API リファレンス, 105
iAnywhere.SQLAnywhere.Server ネームスペース
SQL Anywhere .NET API リファレンス, 105
iAnywhere ODBC ドライバマネージャ
UNIX, 370
iastorv レジストリエントリ
データベースの破損の回避, 1029
iastor レジストリエントリ
データベースの破損の回避, 1029
IdleTimeout プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 203
IMPORT 文
jConnect, 428
INCLUDE 文
SQLCA, 498
IndexColumns フィールド
SAMetaDataCollectionNames クラス [SQL Anywhere .NET API], 273
Indexes フィールド
SAMetaDataCollectionNames クラス [SQL Anywhere .NET API], 273
IndexOf メソッド
SABulkCopyColumnMappingCollection クラス [SQL Anywhere .NET API], 128
SAParameterCollection クラス [SQL Anywhere .NET API], 299
InfoMessage イベント
SAConnection クラス [SQL Anywhere .NET API], 189
InitString プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 203
SAConnection クラス [SQL Anywhere .NET API], 188
INOUT パラメータ
データベース内の Java, 419
Inprocess オプション
リンクサーバ, 355, 357
insensitive カーソル
Embedded SQL, 34
概要, 17
カーソルのプロパティ, 15
更新の例, 20
説明, 22
例の削除, 18
InsertCommand プロパティ
SADataAdapter クラス [SQL Anywhere .NET API], 221
InsertDynamic メソッド
JDBCExample, 443, 444
InsertStatic メソッド
JDBCExample, 441
INSERT 文
JDBC, 440
Python スクリプトの作成, 663
パフォーマンス, 2
Insert メソッド
SAParameterCollection クラス [SQL Anywhere .NET API], 300
INSTALL JAVA 文
JAR のインストール時に使用, 415
クラスのインストール時に使用, 414
Instance フィールド

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01061

SAFactory クラス [SQL Anywhere .NET API], 266
Instance プロパティ
SADataSourceEnumerator クラス [SQL Anywhere .NET API], 252
Integrated プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 203
Intel ストレージドライバ
データベースの破損の回避, 1029
Interactive SQL
dbisqlc の配備, 1026
配備, 1004, 1022
Interactive SQL
JDBC エスケープ構文, 452
Linux と UNIX での配備, 1014
OEM の設定, 1022
UNIX でサポートされる配備プラットフォーム, 1015
Windows での配備, 1005
配備されたアプリケーションのオプション設定, 1025
Interactive SQL のオプション
配備されたアプリケーションの設定をロックする, 1025
Interactive SQL ユーティリティ (dbisql)
UNIX でサポートされる配備プラットフォーム, 1015
Interop
Web サービス, 816
iODBC
ドライバマネージャ, 372
IPV6 プロパティ
SATcpOptionsBuilder クラス [SQL Anywhere .NET API], 322
IsClosed プロパティ
SADataReader クラス [SQL Anywhere .NET API], 249
IsDBNull メソッド
SADataReader クラス [SQL Anywhere .NET API], 247
IsFixedSize プロパティ
SAParameterCollection クラス [SQL Anywhere .NET API], 302
IsNullable プロパティ
SAParameter クラス [SQL Anywhere .NET API], 285
ISOLATIONLEVEL_BROWSE
説明, 859
ISOLATIONLEVEL_CHAOS
説明, 859
ISOLATIONLEVEL_CURSORSTABILITY
説明, 859
ISOLATIONLEVEL_ISOLATED
説明, 859
ISOLATIONLEVEL_READCOMMITTED
説明, 859
ISOLATIONLEVEL_READUNCOMMITTED
説明, 859
ISOLATIONLEVEL_REPEATABLEREAD
説明, 859
ISOLATIONLEVEL_SERIALIZABLE
説明, 859
ISOLATIONLEVEL_UNSPECIFIED
説明, 859
IsolationLevel プロパティ
SATransaction クラス [SQL Anywhere .NET API], 327
isql.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
IsReadOnly プロパティ
SAParameterCollection クラス [SQL Anywhere .NET API], 303
IsSynchronized プロパティ
SAParameterCollection クラス [SQL Anywhere .NET API], 303

J

jaccess.jar
Java ベースの管理ツールの配備, 1011
JAR ファイル
SQL を使用したインストール, 415
Sybase Central からのインストール, 415
インストール, 413
更新, 415
追加, 415
バージョン, 415
JAR ファイルおよび ZIP ファイル作成ウィザード
使用, 415
Java
JDBC, 423
外部環境, 632

データベース内の, 405
JAVA_HOME 環境変数
Java VM の起動, 409
java_vm_options オプション
使用, 409
JavaAccessBridge.dll
Java ベースの管理ツールの配備, 1011
JAVAHOME 環境変数
Java VM の起動, 409
Java Runtime Environment
データベースにおける Java の使用, 409
Java VM
環境変数, 409
起動, 420
シャットダウンフック, 420
選択, 409
停止, 420
JAVA キーワード
外部環境, 632
Java クラス
インストール, 414
インストール例, 410
追加, 414
Java クラス作成ウィザード
使用, 414, 436
Java シグネチャ
CREATE PROCEDURE 文 [ユーザ定義], 634
Java ストアドプロシージャ
説明, 418
例, 418
Java メソッドの呼び出し
チュートリアル, 411
Java メソッドへのアクセス
チュートリアル, 411
JAWTAccessBridge.dll
Java ベースの管理ツールの配備, 1011
JAX-WS
インストール, 850
チュートリアル, 846
バージョン, 850
JComponents1600.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
jconn4.jar
jConnect JDBC ドライバのロード, 434
jConnect ロード, 429
データベースサーバの配備, 1027
jConnect
CLASSPATH 環境変数, 428
JDBC クライアントの配備, 994
JDBC ドライバの選択, 424
URL, 430
外部接続, 431
サーバ側の接続, 435
システムオブジェクト, 429
接続, 430
説明, 428
ダウンロード, 428
提供されるバージョン, 428
データベースの設定, 429
パッケージ, 428
メタデータサポートのインストール, 429
ロード, 429
JDBC
DELETE 文, 440
INSERT 文, 440
Interactive SQL のエスケープ構文, 452
jConnect, 428
JDBC クライアントの配備, 994
SQL Anywhere JDBC ドライバ, 427
SQL 文, 1
UPDATE 文, 440
アプリケーションの概要, 426
オートコミット, 438
オートコミットの動作の制御, 37
オートコミットモード, 36
カーソルタイプ, 15
クライアント側, 426
クライアントの接続, 431
結果セット, 446
権限, 448
サンプル接続, 431
サンプルソースコード, 424
サーバ側, 426
サーバ側の接続, 435
準備文, 442
使用方法, 424
接続, 426
接続コード, 432
接続のデフォルト, 438
説明, 423
データアクセス, 439
データベースへの接続, 430
トランザクションの独立性レベル, 438
バッチ挿入, 441

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01063

バッチ挿入の例, 445
プログラミングの概要, 423
要件, 424
JDBCExample.java
説明, 439
JDBCExamples クラス
説明, 439
JDBC-ODBC ブリッジ
SQL Anywhere JDBC ドライバ, 424
JDBC エスケープ構文
Interactive SQL で使用, 452
JDBC コールバック
例, 448
JDBC ドライバ
OSGi バンドル, 424
互換性, 424
選択, 424
パフォーマンス, 424
jlogon.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
jodbc4.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
JRE
Mac OS X でのバージョンの確認, 1016
データベースにおける Java の使用, 409
jre170
Java Runtime Environment, 1007
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
js.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
JSON サービス
Web クライアントのクイックスタート, 787
Web サーバのクイックスタート, 749
クイックスタート, 789
コメント付け, 760
削除, 760
作成, 756
説明, 754
jsyblib1600.dll
Windows での配備, 1007
jsyblib1600.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007

K

Kerberos プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 203
Keys プロパティ
SAConnectionStringBuilderBase クラス [SQL Anywhere .NET API], 214

L

Language プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 204
LazyClose プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 204
LD_LIBRARY_PATH 環境変数
展開設定, 1020
配備, 962
LDAP プロパティ
SATcpOptionsBuilder クラス [SQL Anywhere .NET API], 322
length SQLDA フィールド
値, 509
説明, 508
libdbcapi_r.dylib
PHP クライアントの配備, 996
PHP サポートモジュール, 644
配備, 1034
libdbcapi_r.so
PHP クライアントの配備, 996
PHP サポートモジュール, 644
配備, 1034
libdbcapi.dylib
PHP サポートモジュール, 644
libdbcapi.so
PHP サポートモジュール, 644
libdbcis16.dylib
データベースサーバの配備, 1027
libdbcis16.so
データベースサーバの配備, 1027
libdbencod16_r
SQL Remote の配備, 1041

libdbextenv16_r.dylib
PHP サポートモジュール, 644
配備, 1034
libdbextenv16_r.so
PHP サポートモジュール, 644
配備, 1034
libdbextf.dylib
データベースサーバの配備, 1027
libdbextf.so
データベースサーバの配備, 1027
libdbfile16_r
SQL Remote の配備, 1041
libdbfips16_r.dylib
PHP クライアントの配備, 996
libdbfips16_r.so
PHP クライアントの配備, 996
libdbftp16_r
SQL Remote の配備, 1041
libdbicu16_r.dylib
JDBC クライアントの配備, 994
ODBC クライアントの配備, 983
PHP クライアントの配備, 996
データベースサーバの配備, 1027
libdbicu16_r.so
JDBC クライアントの配備, 994
Mac OS X での配備, 1016
ODBC クライアントの配備, 983
PHP クライアントの配備, 996
UNIX での配備, 1016
データベースサーバの配備, 1027
libdbicu16.dylib
JDBC クライアントの配備, 983
ODBC クライアントの配備, 983
libdbicu16.so
ODBC クライアントの配備, 983
libdbicudt16.dylib
JDBC クライアントの配備, 983, 994
ODBC クライアントの配備, 983
PHP クライアントの配備, 996
データベースサーバの配備, 1027
libdbicudt16.so
JDBC クライアントの配備, 994
Mac OS X での配備, 1016
ODBC クライアントの配備, 983
PHP クライアントの配備, 996
UNIX での配備, 1016
データベースサーバの配備, 1027
libdbjdbc16.dylib
JDBC クライアントの配備, 994
データベースサーバの配備, 1027
配備, 1035
libdbjdbc16.so
JDBC クライアントの配備, 994
データベースサーバの配備, 1027
配備, 1035
libdbjodbc16.dylib
Mac OS X での配備, 1016
データベースサーバの配備, 1027
libdbjodbc16.so
データベースサーバの配備, 1027
libdbjodbc16.so.1
UNIX での配備, 1016
libdblib16_r.so
データベースユーティリティの配備, 1038
libdblib16_r
SQL Remote の配備, 1041
libdblib16_r.dylib
Mac OS X での配備, 1016
データベースユーティリティの配備, 1038
libdblib16_r.so
PHP クライアントの配備, 996
libdblib16_r.so.1
UNIX での配備, 1016
libdblib16.dylib
PHP クライアントの配備, 996
データベースユーティリティの配備, 1038
libdblib16.so
Embedded SQL クライアントの配備, 992
データベースユーティリティの配備, 1038
libdbodbc16
UNIX ODBC ドライバ, 369
libdbodbc16_n.dylib
ODBC クライアントの配備, 983
データベースサーバの配備, 1027
libdbodbc16_n.so
ODBC クライアントの配備, 983
データベースサーバの配備, 1027
libdbodbc16_r.dylib
Mac OS X での配備, 1016
ODBC クライアントの配備, 983
データベースサーバの配備, 1027
libdbodbc16_r.so
ODBC クライアントの配備, 983
データベースサーバの配備, 1027
libdbodbc16_r.so.1
UNIX での配備, 1016

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01065

libdbodbc16.dylib
ODBC クライアントの配備, 983
データベースサーバの配備, 1027
libdbodbc16.so
ODBC クライアントの配備, 983
データベースサーバの配備, 1027
libdbodbcansi16_r
ANSI ODBC ドライバ, 372
libdbodm16
UNIX ODBC ドライバマネージャ, 370
説明, 370
libdbodm16.dylib
Mac OS X での配備, 1016
ODBC クライアントの配備, 983
libdbodm16.so
ODBC クライアントの配備, 983
libdbodm16.so.1
UNIX での配備, 1016
libdboftsp_r.so
データベースユーティリティの配備, 1038
libdbput16_r.dylib
Mac OS X での配備, 1016
libdbput16_r.so.1
UNIX での配備, 1016
libdbrsa16_r.dylib
PHP クライアントの配備, 996
RSA 暗号化, 1037
libdbrsa16_r.so
PHP クライアントの配備, 996
libdbrsa16.dylib
RSA 暗号化, 1037
libdbrsa16.so
RSA 暗号化, 1037
libdbrsakp16_r.dll
データベースサーバの配備, 1027
libdbrsakp16_r.dylib
データベースサーバの配備, 1027
libdbrsakp16.dylib
JDBC クライアントの配備, 995
Open Client の配備, 1003
libdbrsakp16.so
JDBC クライアントの配備, 995
Open Client の配備, 1003
libdbscript16_r.dylib
データベースサーバの配備, 1027
libdbscript16_r.so
データベースサーバの配備, 1027
libdbserv16_r.dylib
データベースサーバの配備, 1027
libdbserv16_r.so
AES 暗号化, 1037
データベースサーバの配備, 1027
libdbsmtp16_r
SQL Remote の配備, 1041
libdbtasks16_r
SQL Remote の配備, 1041
libdbtasks16_r.dylib
JDBC クライアントの配備, 994
Mac OS X での配備, 1016
ODBC クライアントの配備, 983
PHP クライアントの配備, 996
データベースサーバの配備, 1027
データベースユーティリティの配備, 1038
libdbtasks16_r.so
JDBC クライアントの配備, 994
ODBC クライアントの配備, 983
データベースサーバの配備, 1027
データベースユーティリティの配備, 1038
libdbtasks16_r.so.1
UNIX での配備, 1016
libdbtasks16.dylib
JDBC クライアントの配備, 983
データベースユーティリティの配備, 1038
libdbtasks16.so
Embedded SQL クライアントの配備, 992
ODBC クライアントの配備, 983
PHP クライアントの配備, 996
データベースユーティリティの配備, 1038
libdbtool16_r
SQL Remote の配備, 1041
説明, 861
libdbtool16_r.dylib
Mac OS X での配備, 1016
データベースユーティリティの配備, 1038
libdbtool16_r.so
データベースユーティリティの配備, 1038
libdbtool16_r.so.1
UNIX での配備, 1016
libjsyblib1600_r.dylib
Mac OS X での配備, 1016
libjsyblib1600_r.so.1
UNIX での配備, 1016
libmljodbc16.dylib
Mac OS X での配備, 1016
libmljodbc16.so.1
UNIX での配備, 1016

Copyright © 2013, SAP AG or an SAP affiliate company.

libsybbr.dll
データベースサーバの配備, 1027
libsybbr.dylib
データベースサーバの配備, 1027
libsybbr.so
データベースサーバの配備, 1027
libulscutil16.so.1
UNIX での配備, 1016
LinqSample
.NET データプロバイダサンプルプロジェクト, 43
LINQ サポート
.NET データプロバイダの機能, 42
LinqSample、.NET データプロバイダサンプルプロジェクト, 43
Linux
ディレクトリ構造, 960
配備, 960
Linux と UNIX での配備
SQL Anywhere コンソールユーティリティ (dbconsole), 1014
LivenessTimeout プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 204
LocalOnly プロパティ
SATcpOptionsBuilder クラス [SQL Anywhere .NET API], 322
lockedPreferences
設定可能なオプション, 1022
LogFile プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 204
LONGBINARY データ型
Embedded SQL, 490
LONG BINARY データ型
ESQL, 523
ESQL での取得, 524, 525
ESQL での送信, 526, 527
LONGNVARCHAR データ型
Embedded SQL, 490
LONG NVARCHAR データ型
ESQL, 523
ESQL での取得, 524, 525
ESQL での送信, 526, 527
LONGVARCHAR データ型
Embedded SQL, 490
LONG VARCHAR データ型
ESQL, 523
ESQL での取得, 524, 525
ESQL での送信, 526, 527

M

Mac OS X
JRE バージョンの確認, 1016
ディレクトリ構造, 960
配備, 960
main メソッド
データベース内の Java, 417
maximumDisplayedRows
設定可能なオプション, 1022
MaxPoolSize プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 205
MDAC
配備, 976
バージョン, 976
MergeModule.CABinet
Deployment ウィザード, 964
Message Agent (dbremote)
SQL Remote の配備, 1041
MessageType プロパティ
SAInfoMessageEventArgs クラス [SQL Anywhere .NET API], 268
Message プロパティ
SAError クラス [SQL Anywhere .NET API], 254
SAException クラス [SQL Anywhere .NET API], 260
SAInfoMessageEventArgs クラス [SQL Anywhere .NET API], 268
MetaDataCollections フィールド
SAMetaDataCollectionNames クラス [SQL Anywhere .NET API], 274
Microsoft SQL Server Management Studio
リンクサーバ, 355, 357
Microsoft Transaction Server
3 層コンピューティング, 858
Microsoft Visual C++
Embedded SQL のサポート, 476
MIME
タイプの設定, 765
MIME タイプ
Web サービスチュートリアル, 825
MinPoolSize プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 205

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01067

mldesign.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
mljodbc16.dll
Windows での配備, 1007
mlmon.ini
管理ツールの配備, 1004
mlplugin.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
mlstream.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
mobilink.jpr
Linux、UNIX、Mac OS X での管理ツールの配備, 1018
Windows での管理ツールの配備, 1009
MONEY データ型
Open Client の変換, 744
MSDASQL
OLE DB プロバイダ, 343
msiexec
Deployment ウィザード, 964
サイレントインストール, 968
説明, 966
MSSqlToSA.xml
.NET クライアントの配備, 972
myDispose メソッド
SADataReader クラス [SQL Anywhere .NET API], 247
MyIP プロパティ
SATcpOptionsBuilder クラス [SQL Anywhere .NET API], 322

N

NAMESPACE 句
管理, 801
name SQLDA フィールド
説明, 508
NativeError プロパティ
SAError クラス [SQL Anywhere .NET API], 255
SAException クラス [SQL Anywhere .NET API], 260
SAInfoMessageEventArgs クラス [SQL Anywhere .NET API], 269
NCHAR データ型
ESQL, 490
NewPassword プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 205
NEXT_CONNECTION 関数
例, 778
NEXT_HTTP_HEADER 関数
例, 767
NEXT_HTTP_VARIABLE 関数
例, 767
NextResult メソッド
SADataReader クラス [SQL Anywhere .NET API], 247
ngdbc.jar
UNIX での配備, 1016
Windows での配備, 1007
NodeType プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 205
NO SCROLL カーソル
Embedded SQL, 34
説明, 22
NotifyAfter プロパティ
SABulkCopy クラス [SQL Anywhere .NET API], 114
ntodbc.h
コンパイルプラットフォーム, 387
説明, 367
NuGet
Entity Framework 4, 66
NULL
インジケータ変数, 495
動的 SQL, 506
NULL で終了する文字列
ESQL データ型, 486
NVARCHAR データ型
ESQL, 490

O

OData
説明, 457
OData サーバ
オプション, 459
制限事項, 458

セキュリティの考慮事項, 458
説明, 457
OData サーバ (dbosrv16.exe)
構文, 466
OData サーバ停止 (dbostop.exe)
構文, 467
OData プロデューサ
オプション, 459
サービスモデル, 465
説明, 457
ODBC
64 ビットでの考慮事項, 387
DllMain からの ODBC 関数の呼び出し, 382
FETCH FOR UPDATE, 31
SQL Anywhere のドライバ名, 376
SQLSetConnectAttr, 380
SQL 文, 1
UNIX での開発, 369
Windows Mobile, 368
インポートライブラリ, 367
エスケープ構文, 399
エラーチェック, 401
オプション, 379
オートコミットの動作の制御, 37
オートコミットモード, 36
外部環境, 623
カーソル, 33
カーソルサポートと結果セット, 393
カーソルタイプ, 15
結果セット, 397
サポートされるバージョン, 366
サンプルアプリケーション, 376
サンプルプログラム, 372
システムデータソース, 989
準拠, 366
準備文, 385
ストアドプロシージャ, 397
説明, 365
データアラインメント, 391
データソースタイプ, 989
データソースなしでの接続, 991
ドライバの配備, 982
ドライバマネージャ, 366
ハンドル, 374
複数の結果セット, 397
ヘッダファイル, 367
マルチスレッドアプリケーション, 366
マルチスレッドアプリケーションの接続, 379
ユーザデータソース, 989
要件, 365
リンク, 367
レジストリエントリ, 989
odbc.h
説明, 367
ODBC クライアント
インストール, 982
配備, 982
ODBC データソース
配備, 988
ODBC ドライバ
UNIX, 369
ODBC ドライバマネージャ
iODBC, 372
UNIX, 370
unixODBC, 371
UTF-32, 372
Windows のパフォーマンス, 989
OEM.ini
管理ツール, 1022
Offset プロパティ
SAParameter クラス [SQL Anywhere .NET API], 285
OLE DB
Microsoft リンクサーバの設定, 354
ODBC との組み合わせ, 343
オートコミットの動作の制御, 37
オートコミットモード, 36
カーソル, 33
カーソルタイプ, 15
カーソルによるデータの更新, 349
更新, 349
サポートインタフェース, 358
サポートするプラットフォーム, 344
接続パラメータ, 351
接続プーリング, 353
説明, 343
配備, 976
プロバイダの配備, 976
OLE DB と ADO の開発
説明, 343
OLE DB と ADO のプログラミングインタフェース
概要, 343
説明, 343
OLE トランザクション
3 層コンピューティング, 856

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01069

3 層コンピューティング用語, 857
Open Client
Open Client アプリケーションの配備, 1003
SQL, 744
SQL Anywhere の制限, 747
SQL 文, 1
アーキテクチャ, 742
インタフェース, 741
オートコミットの動作の制御, 37
オートコミットモード, 36
概要, 741
カーソルタイプ, 15
制限, 747
データ型の互換性, 743
データ型の範囲, 744
パスワードの暗号化, 747
要件, 742
openquery
リンクサーバ, 354
OPEN 文
Embedded SQL でのカーソルの使用, 517
Open メソッド
SAConnection クラス [SQL Anywhere .NET API], 185
OSGi 配備バンドル
JDBC 4.0 ドライバ, 424
OUT パラメータ
データベース内の Java, 419
OWASP
Web サービス, 780

P

ParameterName プロパティ
SAParameter クラス [SQL Anywhere .NET API], 285
Parameters プロパティ
SACommand クラス [SQL Anywhere .NET API], 151
Password プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 206
pdf-transcoder.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
Perl
DBD::SQLAnywhere, 649, 652
SQL 文の実行, 654
UNIX と Mac OS X での Perl/DBI サポートのインストール, 651
Windows での Perl/DBI サポートのインストール, 649
外部環境, 638
データベースへの接続, 653
複数の結果セットの処理, 655
ローの挿入, 656
Perl DBD::SQLAnywhere
プログラミングの概要, 649
説明, 649
Perl DBI モジュール
説明, 649
perlenv.pl
配備, 1035
PERL キーワード
外部環境, 638
PersistSecurityInfo プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 206
PHP
API リファレンス, 682
Linux/Solaris での PHP サポートのインストール, 1000
PHP クライアントの配備, 996
PHP の設定, 1002
Web ページでの PHP スクリプトの実行, 668
Windows での PHP サポートのインストール, 999
外部環境, 642
外部環境サポートのインストール, 642
外部環境のトラブルシューティング, 643
拡張, 667
スクリプトの作成, 670
ストアドプロシージャからの PHP スクリプトの呼び出し, 643
説明, 667
データアクセス, 667
データベース内の, 642
バージョン, 998
php.ini
PHP の設定, 1002
php-5.1.[1-*]_sqlanywhere_extenv16.dll
配備, 1036
php-5.1.[1-*]_sqlanywhere_extenv16.so
配備, 1036
php-5.1.[1-*]_sqlanywhere.dll

PHP クライアントの配備, 996
php-5.1.[1-*]_sqlanywhere.so
PHP クライアントの配備, 996
php-5.2.[0-*]_sqlanywhere_extenv16.dll
配備, 1036
php-5.2.[0-*]_sqlanywhere_extenv16.so
配備, 1036
php-5.2.[0-*]_sqlanywhere.dll
PHP クライアントの配備, 996
php-5.2.[0-*]_sqlanywhere.so
PHP クライアントの配備, 996
php-5.3.[0-*]_sqlanywhere_extenv16.dll
配備, 1036
php-5.3.[0-*]_sqlanywhere_extenv16.so
配備, 1036
php-5.3.[0-*]_sqlanywhere.dll
PHP クライアントの配備, 996
php-5.3.[0-*]_sqlanywhere.so
PHP クライアントの配備, 996
php-5.4.[0-*]_sqlanywhere_extenv16.dll
配備, 1036
php-5.4.[0-*]_sqlanywhere_extenv16.so
配備, 1036
php-5.4.[0-*]_sqlanywhere.dll
PHP クライアントの配備, 996
php-5.4.[0-*]_sqlanywhere.so
PHP クライアントの配備, 996
phpenv.php
PHP サポートモジュール, 644
配備, 1036
PHP Hypertext Preprocessor
説明, 667
PHPRC
環境変数, 644
PHP 拡張
テスト, 667
PHP 拡張
Linux/Solaris, 1000
Windows, 999
使用する PHP 拡張の選択, 998
バージョン, 998
プログラミングの概要, 667
PHP 関数
sasql_affected_rows, 683
sasql_close, 684
sasql_commit, 684
sasql_connect, 684
sasql_data_seek, 685
sasql_disconnect, 686
sasql_error, 686
sasql_errorcode, 687
sasql_escape_string, 687
sasql_fetch_array, 688
sasql_fetch_assoc, 688
sasql_fetch_field, 689
sasql_fetch_object, 690
sasql_fetch_row, 690
sasql_field_count, 691
sasql_field_seek, 691
sasql_free_result, 692
sasql_get_client_info, 692
sasql_insert_id, 692
sasql_message, 693
sasql_multi_query, 693
sasql_next_result, 694
sasql_num_fields, 694
sasql_num_rows, 695
sasql_pconnect, 695
sasql_prepare, 696
sasql_query, 696
sasql_real_escape_string, 697
sasql_real_query, 698
sasql_result_all, 698
sasql_rollback, 699
sasql_set_option, 700
sasql_sqlstate, 711
sasql_stmt_affected_rows, 701
sasql_stmt_bind_param, 701
sasql_stmt_bind_param_ex, 702
sasql_stmt_bind_result, 702
sasql_stmt_close, 703
sasql_stmt_data_seek, 703
sasql_stmt_errno, 704
sasql_stmt_error, 704
sasql_stmt_execute, 705
sasql_stmt_fetch, 705
sasql_stmt_field_count, 706
sasql_stmt_free_result, 706
sasql_stmt_insert_id, 707
sasql_stmt_next_result, 707
sasql_stmt_num_rows, 707
sasql_stmt_param_count, 708
sasql_stmt_reset, 708
sasql_stmt_result_metadata, 709
sasql_stmt_send_long_data, 709
sasql_stmt_store_result, 710

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01071

sasql_store_result, 710
sasql_use_result, 711
PHP キーワード
外部環境, 643
policy.16.0.iAnywhere.Data.SQLAnywhere.dll
.NET クライアントの配備, 972
policy.16.0.iAnywhere.Data.SQLAnywhere.v3.5.dll
.NET クライアントの配備, 972
policy.16.0.iAnywhere.Data.SQLAnywhere.v4.0.dll
.NET クライアントの配備, 972
POOLING オプション
.NET データプロバイダ, 46
Pooling プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 206
Precision プロパティ
SAParameter クラス [SQL Anywhere .NET API], 286
PrefetchBuffer プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 206
PrefetchRows プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 206
prefetch オプション
カーソル, 28
PreparedStatement インタフェース
説明, 442
prepareStatement メソッド
JDBC, 4
PREPARE TRANSACTION 文
Open Client, 747
PREPARE 文
使用, 503
Prepare メソッド
SACommand クラス [SQL Anywhere .NET API], 148
使用, 3
ProcedureParameters フィールド
SAMetaDataCollectionNames クラス [SQL Anywhere .NET API], 274
Procedures フィールド
SAMetaDataCollectionNames クラス [SQL Anywhere .NET API], 274
PUT 文
カーソルによるローの変更, 12
複数のロー, 519
ワイド, 519
Python
commit メソッド, 663
sqlanydb, 659
sqlanydb スクリプトの作成, 661
SQL 文の実行, 663
UNIX と Mac OS X での Python サポートのインストール, 660
Windows での Python サポートのインストール, 660
カーソルの作成, 663
接続の作成, 662
接続の切断, 662
タイプ変換の制御, 664
データベースタイプ, 665
テーブルへ挿入, 663
複数の挿入, 664
Python データベース API
プログラミングの概要, 659
Python データベースサポート
説明, 659

Q

QuoteIdentifier メソッド
SACommandBuilder クラス [SQL Anywhere .NET API], 162

R

Rails
ActiveRecord アダプタのインストール, 713
説明, 713
チュートリアル, 716
RAW サービス
Web クライアントのクイックスタート, 787
Web サーバのクイックスタート, 749
クイックスタート, 789
コメント付け, 760
削除, 760
作成, 756
説明, 754
READ_CLIENT_FILE 関数
ESQL クライアント API コールバック関数, 546
ODBC クライアント API コールバック関数, 380
Read メソッド
SADataReader クラス [SQL Anywhere .NET API], 248

ReceiveBufferSize プロパティ
SATcpOptionsBuilder クラス [SQL Anywhere .NET API], 323
RecordsAffected プロパティ
SADataReader クラス [SQL Anywhere .NET API], 250
SARowUpdatedEventArgs クラス [SQL Anywhere .NET API], 312
Recordset ADO オブジェクト
ADO, 347
ADO プログラミング, 350
データの更新, 349
Recordset オブジェクト
ADO, 348
relayserver.jpr
Linux、UNIX、Mac OS X での管理ツールの配備, 1019
Windows での管理ツールの配備, 1010
REMOTEPWD
使用, 430
RemoveAt メソッド
SABulkCopyColumnMappingCollection クラス [SQL Anywhere .NET API], 129
SAParameterCollection クラス [SQL Anywhere .NET API], 301
removeShutdownHook
Java VM シャットダウンフック, 420
Remove メソッド
SABulkCopyColumnMappingCollection クラス [SQL Anywhere .NET API], 128
SAConnectionStringBuilderBase クラス [SQL Anywhere .NET API], 212
SAParameterCollection クラス [SQL Anywhere .NET API], 301
reportErrors
設定可能なオプション, 1022
ReservedWords フィールド
SAMetaDataCollectionNames クラス [SQL Anywhere .NET API], 275
ResetCommandTimeout メソッド
SACommand クラス [SQL Anywhere .NET API], 149
ResetDbType メソッド
SAParameter クラス [SQL Anywhere .NET API], 283
Restrictions フィールド
SAMetaDataCollectionNames クラス [SQL Anywhere .NET API], 275
Results メソッド
JDBCExample, 446, 447
RetryConnectionTimeout プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 207
ROLLBACK TO SAVEPOINT 文
カーソル, 39
RollbackTrans ADO メソッド
ADO プログラミング, 350
データの更新, 350
ROLLBACK 文
カーソル, 39
Rollback メソッド
SATransaction クラス [SQL Anywhere .NET API], 326
root Web サービス
Web 参照, 782
説明, 760
RowsCopied プロパティ
SARowsCopiedEventArgs クラス [SQL Anywhere .NET API], 310
RowUpdated イベント
SADataAdapter クラス [SQL Anywhere .NET API], 223
RowUpdating イベント
SADataAdapter クラス [SQL Anywhere .NET API], 224
RPC オプション
リンクサーバ, 355, 357
RPC 出力オプション
リンクサーバ, 355, 357
Ruby
ActiveRecord アダプタのインストール, 713
Ruby/DBI サポートのインストール, 713
説明, 713
ネイティブ Ruby ドライバのインストール, 713
Ruby API
sqlany_affected_rows 関数, 722
sqlany_bind_param 関数, 722
sqlany_clear_error 関数, 723
sqlany_client_version 関数, 723
sqlany_commit 関数, 724
sqlany_connect 関数, 724
sqlany_describe_bind_param 関数, 725
sqlany_disconnect 関数, 725
sqlany_error 関数, 726
sqlany_execute_direct 関数, 727
sqlany_execute_immediate 関数, 728

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01073

sqlany_execute 関数, 726
sqlany_fetch_absolute 関数, 728
sqlany_fetch_next 関数, 729
sqlany_fini 関数, 730
sqlany_free_connection 関数, 730
sqlany_free_stmt 関数, 731
sqlany_get_bind_param_info 関数, 731
sqlany_get_column_info 関数, 733
sqlany_get_column 関数, 732
sqlany_get_next_result 関数, 734
sqlany_init 関数, 734
sqlany_new_connection 関数, 735
sqlany_num_cols 関数, 735
sqlany_num_params 関数, 736
sqlany_num_rows 関数, 736
sqlany_prepare 関数, 737
sqlany_rollback 関数, 738
sqlany_sqlstate 関数, 738
説明, 721
プログラミングの概要, 713
Ruby DBI
dbd-sqlanywhere のインストール, 713
接続の例, 717
説明, 717
RubyGems
インストール, 713
Ruby on Rails
ActiveRecord アダプタのインストール, 713
説明, 713
チュートリアル, 716

S

sa_config.csh
配備, 962
sa_config.sh
配備, 962
sa_external_library_unload システムプロシージャ
使用, 618
SA_GET_MESSAGE_CALLBACK_PARM
SQLSetConnectAttr, 380
SA_REGISTER_MESSAGE_CALLBACK
SQLSetConnectAttr, 380
SA_REGISTER_VALIDATE_FILE_TRANSFER_CALLBACK
SQLSetConnectAttr, 380
sa_set_http_header システムプロシージャ
例, 765
SA_SQL_ATTR_TXN_ISOLATION
SQLSetConnectAttr, 380
SA_SQL_TXN_READONLY_STATEMENT_SNAPSHOT
独立性レベル, 393
SA_SQL_TXN_SNAPSHOT
独立性レベル, 393
SA_SQL_TXN_STATEMENT_SNAPSHOT
独立性レベル, 393
SA_TRANSACTION_SNAPSHOT
トランザクションの独立性レベル, 438
SA_TRANSACTION_STATEMENT_READONLY_SNAPSHOT
トランザクションの独立性レベル, 438
SA_TRANSACTION_STATEMENT_SNAPSHOT
トランザクションの独立性レベル, 438
SABulkCopyColumnMappingCollection クラス [SQL Anywhere .NET API]
Add メソッド, 123
Contains メソッド, 127
CopyTo メソッド, 127
IndexOf メソッド, 128
RemoveAt メソッド, 129
Remove メソッド, 128
this プロパティ, 129
説明, 121
SABulkCopyColumnMapping クラス [SQL Anywhere .NET API]
DestinationColumn プロパティ, 119
DestinationOrdinal プロパティ, 119
SABulkCopyColumnMapping コンストラクタ, 116
SourceColumn プロパティ, 120
SourceOrdinal プロパティ, 120
説明, 115
SABulkCopyColumnMapping コンストラクタ
SABulkCopyColumnMapping クラス [SQL Anywhere .NET API], 116
SABulkCopyOptions 列挙体 [SQL Anywhere .NET API]
説明, 330
SABulkCopy クラス [SQL Anywhere .NET API]
BatchSize プロパティ, 112
BulkCopyTimeout プロパティ, 112
Close メソッド, 109
ColumnMappings プロパティ, 113
DestinationTableName プロパティ, 113
Dispose メソッド, 109

Copyright © 2013, SAP AG or an SAP affiliate company.

NotifyAfter プロパティ, 114
SABulkCopy コンストラクタ, 106
SARowsCopied イベント, 114
WriteToServer メソッド, 109
説明, 105
SABulkCopy コンストラクタ
SABulkCopy クラス [SQL Anywhere .NET API], 106
sacapi.h ヘッダファイル
SQL Anywhere C API リファレンス, 566
sacapidll.h
インタフェースライブラリ, 566
sacapidll.h ヘッダファイル
SQL Anywhere C API リファレンス, 566
SACommand
プライマリキー値の取得, 52
SACommandBuilder クラス [SQL Anywhere .NET API]
DataAdapter プロパティ, 163
DeriveParameters メソッド, 156
GetDeleteCommand メソッド, 156
GetInsertCommand メソッド, 158
GetUpdateCommand メソッド, 160
QuoteIdentifier メソッド, 162
SACommandBuilder コンストラクタ, 155
UnquoteIdentifier メソッド, 163
説明, 152
SACommandBuilder コンストラクタ
SACommandBuilder クラス [SQL Anywhere .NET API], 155
SACommand クラス
Visual Studio プロジェクトでの使用, 81
準備文の使用, 3
説明, 47
データの更新, 50
データの削除, 50
データの取得, 48
データの挿入, 50
例, 86
SACommand クラス [SQL Anywhere .NET API]
BeginExecuteNonQuery メソッド, 134
BeginExecuteReader メソッド, 136
Cancel メソッド, 140
CommandText プロパティ, 149
CommandTimeout プロパティ, 149
CommandType プロパティ, 149
Connection プロパティ, 150
CreateParameter メソッド, 140
DesignTimeVisible プロパティ, 150
EndExecuteNonQuery メソッド, 141
EndExecuteReader メソッド, 143
ExecuteNonQuery メソッド, 145
ExecuteReader メソッド, 146
ExecuteScalar メソッド, 147
Parameters プロパティ, 151
Prepare メソッド, 148
ResetCommandTimeout メソッド, 149
SACommand コンストラクタ, 132
Transaction プロパティ, 151
UpdatedRowSource プロパティ, 152
説明, 129
SACommand コンストラクタ
SACommand クラス [SQL Anywhere .NET API], 132
SACommLinksOptionsBuilder クラス [SQL Anywhere .NET API]
All プロパティ, 167
ConnectionString プロパティ, 168
GetUseLongNameAsKeyword メソッド, 166
SACommLinksOptionsBuilder コンストラクタ, 165
SetUseLongNameAsKeyword メソッド, 166
SharedMemory プロパティ, 168
TcpOptionsBuilder プロパティ, 168
TcpOptionsString プロパティ, 168
ToString メソッド, 167
説明, 164
SACommLinksOptionsBuilder コンストラクタ
SACommLinksOptionsBuilder クラス [SQL Anywhere .NET API], 165
SAConnectionStringBuilderBase クラス [SQL Anywhere .NET API]
ContainsKey メソッド, 210
GetKeyword メソッド, 211
GetUseLongNameAsKeyword メソッド, 211
Keys プロパティ, 214
Remove メソッド, 212
SetUseLongNameAsKeyword メソッド, 212
ShouldSerialize メソッド, 213
this プロパティ, 214
TryGetValue メソッド, 213
説明, 208
SAConnectionStringBuilder クラス [SQL Anywhere .NET API]
AppInfo プロパティ, 197
AutoStart プロパティ, 197

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01075

AutoStop プロパティ, 197
Charset プロパティ, 197
CommBufferSize プロパティ, 198
CommLinks プロパティ, 198
CompressionThreshold プロパティ, 198
Compress プロパティ, 198
ConnectionLifetime プロパティ, 198
ConnectionName プロパティ, 199
ConnectionPool プロパティ, 199
ConnectionReset プロパティ, 199
ConnectionTimeout プロパティ, 199
DatabaseFile プロパティ, 200
DatabaseKey プロパティ, 200
DatabaseName プロパティ, 200
DatabaseSwitches プロパティ, 200
DataSourceName プロパティ, 201
DisableMultiRowFetch プロパティ, 201
Elevate プロパティ, 201
EncryptedPassword プロパティ, 201
Encryption プロパティ, 202
Enlist プロパティ, 202
FileDataSourceName プロパティ, 202
ForceStart プロパティ, 202
Host プロパティ, 203
IdleTimeout プロパティ, 203
InitString プロパティ, 203
Integrated プロパティ, 203
Kerberos プロパティ, 203
Language プロパティ, 204
LazyClose プロパティ, 204
LivenessTimeout プロパティ, 204
LogFile プロパティ, 204
MaxPoolSize プロパティ, 205
MinPoolSize プロパティ, 205
NewPassword プロパティ, 205
NodeType プロパティ, 205
Password プロパティ, 206
PersistSecurityInfo プロパティ, 206
Pooling プロパティ, 206
PrefetchBuffer プロパティ, 206
PrefetchRows プロパティ, 206
RetryConnectionTimeout プロパティ, 207
SAConnectionStringBuilder コンストラクタ, 195
ServerName プロパティ, 207
StartLine プロパティ, 207
Unconditional プロパティ, 207
UserID プロパティ, 208
説明, 190
SAConnectionStringBuilder コンストラクタ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 195
SAConnection 関数
Visual Studio プロジェクトでの使用, 85
SAConnection クラス
Visual Studio プロジェクトでの使用, 81
データベースへの接続, 45
例, 86
SAConnection クラス [SQL Anywhere .NET API]
BeginTransaction メソッド, 172
ChangeDatabase メソッド, 174
ChangePassword メソッド, 175
ClearAllPools メソッド, 175
ClearPool メソッド, 175
Close メソッド, 176
ConnectionString プロパティ, 186
ConnectionTimeout プロパティ, 187
CreateCommand メソッド, 176
Database プロパティ, 187
DataSource プロパティ, 188
EnlistDistributedTransaction メソッド, 177
EnlistTransaction メソッド, 177
GetSchema メソッド, 177
InfoMessage イベント, 189
InitString プロパティ, 188
Open メソッド, 185
SAConnection コンストラクタ, 171
ServerVersion プロパティ, 188
StateChange イベント, 189
State プロパティ, 189
説明, 169
SAConnection コンストラクタ
SAConnection クラス [SQL Anywhere .NET API], 171
sacshelp16.chm
ODBC クライアントの配備, 983
SADataAdapter
プライマリキー値の取得, 60
SADataAdapter クラス
説明, 47
データの更新, 53
データの削除, 53
データの取得, 55, 56
データの挿入, 53
SADataAdapter クラス [SQL Anywhere .NET API]
DeleteCommand プロパティ, 220
GetFillParameters メソッド, 220

Copyright © 2013, SAP AG or an SAP affiliate company.

InsertCommand プロパティ, 221
RowUpdated イベント, 223
RowUpdating イベント, 224
SADataAdapter コンストラクタ, 217
SelectCommand プロパティ, 221
TableMappings プロパティ, 221
UpdateBatchSize プロパティ, 222
UpdateCommand プロパティ, 223
説明, 215
SADataAdapter コンストラクタ
SADataAdapter クラス [SQL Anywhere .NET API], 217
SADataReader クラス
Visual Studio プロジェクトでの使用, 81
使用, 48
例, 86
SADataReader クラス [SQL Anywhere .NET API]
Close メソッド, 228
Depth プロパティ, 248
FieldCount プロパティ, 249
GetBoolean メソッド, 229
GetBytes メソッド, 230
GetByte メソッド, 229
GetChars メソッド, 231
GetChar メソッド, 231
GetDataTypeName メソッド, 233
GetData メソッド, 232
GetDateTimeOffset メソッド, 234
GetDateTime メソッド, 233
GetDecimal メソッド, 234
GetDouble メソッド, 235
GetEnumerator メソッド, 235
GetFieldType メソッド, 236
GetFloat メソッド, 236
GetGuid メソッド, 237
GetInt16 メソッド, 237
GetInt32 メソッド, 238
GetInt64 メソッド, 238
GetName メソッド, 239
GetOrdinal メソッド, 239
GetSchemaTable メソッド, 240
GetString メソッド, 242
GetTimeSpan メソッド, 243
GetUInt16 メソッド, 243
GetUInt32 メソッド, 244
GetUInt64 メソッド, 244
GetValues メソッド, 246
GetValue メソッド, 245
HasRows プロパティ, 249
IsClosed プロパティ, 249
IsDBNull メソッド, 247
myDispose メソッド, 247
NextResult メソッド, 247
Read メソッド, 248
RecordsAffected プロパティ, 250
this プロパティ, 250
説明, 224
SADataSourceEnumerator クラス [SQL Anywhere .NET API]
GetDataSources メソッド, 252
Instance プロパティ, 252
説明, 251
SADbType プロパティ
SAParameter クラス [SQL Anywhere .NET API], 286
SADbType 列挙体 [SQL Anywhere .NET API]
説明, 332
SADefault クラス [SQL Anywhere .NET API]
Value フィールド, 253
説明, 253
SAErrorCollection クラス [SQL Anywhere .NET API]
CopyTo メソッド, 256
Count プロパティ, 257
GetEnumerator メソッド, 257
this プロパティ, 257
説明, 255
SAError クラス [SQL Anywhere .NET API]
Message プロパティ, 254
NativeError プロパティ, 255
Source プロパティ, 255
SqlState プロパティ, 255
ToString メソッド, 254
説明, 253
SAException クラス
Visual Studio プロジェクトでの使用, 82
例, 86
SAException クラス [SQL Anywhere .NET API]
Errors プロパティ, 260
GetObjectData メソッド, 259
Message プロパティ, 260
NativeError プロパティ, 260
Source プロパティ, 261
説明, 258
SAFactory クラス [SQL Anywhere .NET API]

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01077

CanCreateDataSourceEnumerator プロパティ, 266
CreateCommandBuilder メソッド, 263
CreateCommand メソッド, 263
CreateConnectionStringBuilder メソッド, 264
CreateConnection メソッド, 263
CreateDataAdapter メソッド, 264
CreateDataSourceEnumerator メソッド, 265
CreateParameter メソッド, 265
CreatePermission メソッド, 265
Instance フィールド, 266
説明, 261
SAInfoMessageEventArgs クラス [SQL Anywhere .NET API]
Errors プロパティ, 268
MessageType プロパティ, 268
Message プロパティ, 268
NativeError プロパティ, 269
Source プロパティ, 269
ToString メソッド, 268
説明, 267
SAInfoMessageEventHandler デリゲート [SQL Anywhere .NET API]
説明, 329
saip16.jar
Windows での配備, 1007
SAIsolationLevel プロパティ
SATransaction クラス [SQL Anywhere .NET API], 328
SAIsolationLevel 列挙体 [SQL Anywhere .NET API]
説明, 338
sajdbc4.jar
JDBC クライアントの配備, 994
SQL Anywhere JDBC ドライバのロード, 434
データベースサーバの配備, 1027
sajvm.jar
データベースサーバの配備, 1027
salib.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
SAMessageType 列挙体 [SQL Anywhere .NET API]
説明, 341
SAMetaDataCollectionNames クラス [SQL Anywhere .NET API]
Columns フィールド, 271
DataSourceInformation フィールド, 271
DataTypes フィールド, 272
ForeignKeys フィールド, 272
IndexColumns フィールド, 273
Indexes フィールド, 273
MetaDataCollections フィールド, 274
ProcedureParameters フィールド, 274
Procedures フィールド, 274
ReservedWords フィールド, 275
Restrictions フィールド, 275
Tables フィールド, 276
UserDefinedTypes フィールド, 276
Users フィールド, 276
ViewColumns フィールド, 277
Views フィールド, 277
説明, 269
SAOLEDB
OLE DB プロバイダ, 343
SAParameterCollection クラス [SQL Anywhere .NET API]
AddRange メソッド, 295
AddWithValue メソッド, 296
Add メソッド, 291
Clear メソッド, 297
Contains メソッド, 297
CopyTo メソッド, 298
Count プロパティ, 302
GetEnumerator メソッド, 299
IndexOf メソッド, 299
Insert メソッド, 300
IsFixedSize プロパティ, 302
IsReadOnly プロパティ, 303
IsSynchronized プロパティ, 303
RemoveAt メソッド, 301
Remove メソッド, 301
SyncRoot プロパティ, 303
this プロパティ, 304
説明, 289
SAParameter クラス [SQL Anywhere .NET API]
DbType プロパティ, 284
Direction プロパティ, 284
IsNullable プロパティ, 285
Offset プロパティ, 285
ParameterName プロパティ, 285
Precision プロパティ, 286
ResetDbType メソッド, 283
SADbType プロパティ, 286
SAParameter コンストラクタ, 279
Scale プロパティ, 286
Size プロパティ, 287

Copyright © 2013, SAP AG or an SAP affiliate company.

SourceColumnNullMapping プロパティ, 288
SourceColumn プロパティ, 287
SourceVersion プロパティ, 288
ToString メソッド, 283
Value プロパティ, 288
説明, 278
SAParameter コンストラクタ
SAParameter クラス [SQL Anywhere .NET API], 279
SAPermissionAttribute クラス [SQL Anywhere .NET API]
CreatePermission メソッド, 308
SAPermissionAttribute コンストラクタ, 308
説明, 307
SAPermissionAttribute コンストラクタ
SAPermissionAttribute クラス [SQL Anywhere .NET API], 308
SAPermission クラス [SQL Anywhere .NET API]
SAPermission コンストラクタ, 307
説明, 305
SAPermission コンストラクタ
SAPermission クラス [SQL Anywhere .NET API], 307
saplugin.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
SARowsCopiedEventArgs クラス [SQL Anywhere .NET API]
Abort プロパティ, 310
RowsCopied プロパティ, 310
SARowsCopiedEventArgs コンストラクタ, 309
説明, 309
SARowsCopiedEventArgs コンストラクタ
SARowsCopiedEventArgs クラス [SQL Anywhere .NET API], 309
SARowsCopiedEventHandler デリゲート [SQL Anywhere .NET API]
説明, 330
SARowsCopied イベント
SABulkCopy クラス [SQL Anywhere .NET API], 114
SARowUpdatedEventArgs クラス [SQL Anywhere .NET API]
Command プロパティ, 312
RecordsAffected プロパティ, 312
SARowUpdatedEventArgs コンストラクタ, 311
説明, 310
SARowUpdatedEventArgs コンストラクタ
SARowUpdatedEventArgs クラス [SQL Anywhere .NET API], 311
SARowUpdatedEventHandler デリゲート [SQL Anywhere .NET API]
説明, 329
SARowUpdatingEventArgs クラス [SQL Anywhere .NET API]
Command プロパティ, 314
SARowUpdatingEventArgs コンストラクタ, 314
説明, 313
SARowUpdatingEventArgs コンストラクタ
SARowUpdatingEventArgs クラス [SQL Anywhere .NET API], 314
SARowUpdatingEventHandler デリゲート [SQL Anywhere .NET API]
説明, 329
SAServerSideConnection クラス [SQL Anywhere .NET API]
Connection プロパティ, 315
説明, 315
sasql_affected_rows 関数 (PHP)
構文, 683
sasql_close 関数 (PHP)
構文, 684
sasql_commit 関数 (PHP)
構文, 684
sasql_connect 関数 (PHP)
構文, 684
sasql_data_seek 関数 (PHP)
構文, 685
sasql_disconnect 関数 (PHP)
構文, 686
sasql_errorcode 関数 (PHP)
構文, 687
sasql_error 関数 (PHP)
構文, 686
sasql_escape_string 関数 (PHP)
構文, 687
sasql_fetch_array 関数 (PHP)
構文, 688
sasql_fetch_assoc 関数 (PHP)
構文, 688
sasql_fetch_field 関数 (PHP)
構文, 689
sasql_fetch_object 関数 (PHP)
構文, 690
sasql_fetch_row 関数 (PHP)

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01079

構文, 690
sasql_field_count 関数 (PHP)
構文, 691
sasql_field_seek 関数 (PHP)
構文, 691
sasql_free_result 関数 (PHP)
構文, 692
sasql_get_client_info 関数 (PHP)
構文, 692
sasql_insert_id 関数 (PHP)
構文, 692
sasql_message 関数 (PHP)
構文, 693
sasql_multi_query 関数 (PHP)
構文, 693
sasql_next_result 関数 (PHP)
構文, 694
sasql_num_fields 関数 (PHP)
構文, 694
sasql_num_rows 関数 (PHP)
構文, 695
sasql_pconnect 関数 (PHP)
構文, 695
sasql_prepare 関数 (PHP)
構文, 696
sasql_query 関数 (PHP)
構文, 696
sasql_real_escape_string 関数 (PHP)
構文, 697
sasql_real_query 関数 (PHP)
構文, 698
sasql_result_all 関数 (PHP)
構文, 698
sasql_rollback 関数 (PHP)
構文, 699
sasql_set_option 関数 (PHP)
構文, 700
sasql_sqlstate 関数 (PHP)
構文, 711
sasql_stmt_affected_rows 関数 (PHP)
構文, 701
sasql_stmt_bind_param_ex 関数 (PHP)
構文, 702
sasql_stmt_bind_param 関数 (PHP)
構文, 701
sasql_stmt_bind_result 関数 (PHP)
構文, 702
sasql_stmt_close 関数 (PHP)
構文, 703
sasql_stmt_data_seek 関数 (PHP)
構文, 703
sasql_stmt_errno 関数 (PHP)
構文, 704
sasql_stmt_error 関数 (PHP)
構文, 704
sasql_stmt_execute 関数 (PHP)
構文, 705
sasql_stmt_fetch 関数 (PHP)
構文, 705
sasql_stmt_field_count 関数 (PHP)
構文, 706
sasql_stmt_free_result 関数 (PHP)
構文, 706
sasql_stmt_insert_id 関数 (PHP)
構文, 707
sasql_stmt_next_result 関数 (PHP)
構文, 707
sasql_stmt_num_rows 関数 (PHP)
構文, 707
sasql_stmt_param_count 関数 (PHP)
構文, 708
sasql_stmt_reset 関数 (PHP)
構文, 708
sasql_stmt_result_metadata 関数 (PHP)
構文, 709
sasql_stmt_send_long_data 関数 (PHP)
構文, 709
sasql_stmt_store_result 関数 (PHP)
構文, 710
sasql_store_result 関数 (PHP)
構文, 710
sasql_use_result 関数 (PHP)
構文, 711
SATcpOptionsBuilder クラス [SQL Anywhere .NET API]
BroadcastListener プロパティ, 321
Broadcast プロパティ, 321
ClientPort プロパティ, 321
DoBroadcast プロパティ, 321
Host プロパティ, 322
IPV6 プロパティ, 322
LDAP プロパティ, 322
LocalOnly プロパティ, 322
MyIP プロパティ, 322
ReceiveBufferSize プロパティ, 323
SATcpOptionsBuilder コンストラクタ, 319

SendBufferSize プロパティ, 323
ServerPort プロパティ, 323
TDS プロパティ, 323
Timeout プロパティ, 324
ToString メソッド, 320
VerifyServerName プロパティ, 324
説明, 316
SATcpOptionsBuilder コンストラクタ
SATcpOptionsBuilder クラス [SQL Anywhere .NET API], 319
SAToMSSql10.xml
.NET クライアントの配備, 972
SATransaction クラス
使用, 64
SATransaction クラス [SQL Anywhere .NET API]
Commit メソッド, 325
Connection プロパティ, 327
IsolationLevel プロパティ, 327
Rollback メソッド, 326
SAIsolationLevel プロパティ, 328
Save メソッド, 327
説明, 324
Save メソッド
SATransaction クラス [SQL Anywhere .NET API], 327
SA ロケーションオプション
SetupVSPackage, 972
sbgse2.dll
FIPS 認定の RSA 暗号化, 1037
Scale プロパティ
SAParameter クラス [SQL Anywhere .NET API], 286
SCEditor1600.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
scjview
Mac OS X での配備, 1016
UNIX での配備, 1016
scjview.exe
Windows での配備, 1007
scjview.ini
管理ツールの配備, 1004
SCROLL カーソル
Embedded SQL, 34
value-sensitive, 25
SelectCommand プロパティ
SADataAdapter クラス [SQL Anywhere .NET API], 221
SELECT 文
シングルロー, 516
動的 SELECT 文の使用, 505
SendBufferSize プロパティ
SATcpOptionsBuilder クラス [SQL Anywhere .NET API], 323
sensitive カーソル
Embedded SQL, 34
概要, 17
カーソルのプロパティ, 15
更新の例, 20
説明, 23
例の削除, 18
ServerName プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 207
ServerPort プロパティ
SATcpOptionsBuilder クラス [SQL Anywhere .NET API], 323
ServerVersion プロパティ
SAConnection クラス [SQL Anywhere .NET API], 188
SessionCreateTime プロパティ
説明, 776
SessionID プロパティ
例, 773
SessionLastTime プロパティ
説明, 776
set_cancel
説明, 614
set_value 関数
使用, 617
説明, 613
setAutoCommit メソッド
説明, 438
setTransactionIsolation メソッド
JDBC, 438
SetupVSPackage
.NET クライアントの配備, 972
.NET データプロバイダのアンインストール, 975
SetUseLongNameAsKeyword メソッド
SACommLinksOptionsBuilder クラス [SQL Anywhere .NET API], 166
SAConnectionStringBuilderBase クラス [SQL Anywhere .NET API], 212

SharedMemory プロパティ
SACommLinksOptionsBuilder クラス [SQL Anywhere .NET API], 168
ShouldSerialize メソッド
SAConnectionStringBuilderBase クラス [SQL Anywhere .NET API], 213
showMultipleResultSets
設定可能なオプション, 1022
showPerformanceDataUI
設定オプション, 1022
showResultsForAllStatements
設定可能なオプション, 1022
SimpleCE
.NET データプロバイダサンプルプロジェクト, 43
SimpleViewer
.NET データプロバイダサンプルプロジェクト, 43
.NET プロジェクト, 87
SimpleWin32
.NET データプロバイダサンプルプロジェクト, 43
SimpleXML
.NET データプロバイダサンプルプロジェクト, 43
Size プロパティ
SAParameter クラス [SQL Anywhere .NET API], 287
SMALLDATETIME データ型
Open Client の変換, 744
SMALLMONEY データ型
Open Client の変換, 744
SOAP
エンベロープでの変数の指定, 805
SOAPHEADER 句
管理, 797
SOAP サービス
.NET チュートリアル, 839
JAX-WS チュートリアル, 846
SQL Anywhere Web クライアントチュートリアル, 830
コメント付け, 760
削除, 760
作成, 757
説明, 754
データ型, 809
フォールト, 823
SOAP システムプロシージャ
アルファベット順リスト, 780
SOAP ネームスペース
管理, 801
SOAP ヘッダ
管理, 797
SOAP 要求
構造, 822
SourceColumnNullMapping プロパティ
SAParameter クラス [SQL Anywhere .NET API], 288
SourceColumn プロパティ
SABulkCopyColumnMapping クラス [SQL Anywhere .NET API], 120
SAParameter クラス [SQL Anywhere .NET API], 287
SourceOrdinal プロパティ
SABulkCopyColumnMapping クラス [SQL Anywhere .NET API], 120
SourceVersion プロパティ
SAParameter クラス [SQL Anywhere .NET API], 288
Source プロパティ
SAError クラス [SQL Anywhere .NET API], 255
SAException クラス [SQL Anywhere .NET API], 261
SAInfoMessageEventArgs クラス [SQL Anywhere .NET API], 269
sp_tsql_environment システムプロシージャ
jConnect でのオプションの設定, 431
SQL
ADO アプリケーション, 1
Embedded SQL アプリケーション, 1
JDBC アプリケーション, 1
ODBC アプリケーション, 1
Open Client アプリケーション, 1
アプリケーション, 1
SQL_ATTR_CONCURRENCY 属性
説明, 394
SQL_ATTR_CONNECTION_DEAD
SQLGetConnectAttr, 379
SQL_ATTR_CURSOR_SCROLLABLE 属性
説明, 394
SQL_ATTR_KEYSET_SIZE
ODBC 属性, 33
SQL_ATTR_MAX_LENGTH 属性
説明, 395
SQL_ATTR_ROW_ARRAY_SIZE
ODBC 属性, 33

Copyright © 2013, SAP AG or an SAP affiliate company.

複数ローのフェッチ, 11
SQL_CALLBACK_PARM 型宣言
説明, 546
SQL_CALLBACK 型宣言
説明, 546
SQL_CONCUR_LOCK
同時実行性の値, 394
SQL_CONCUR_READ_ONLY
同時実行性の値, 394
SQL_CONCUR_ROWVER
同時実行性の値, 394
SQL_CONCUR_VALUES
同時実行性の値, 394
SQL_CURSOR_KEYSET_DRIVEN
ODBC カーソル属性, 33
SQL_ERROR
ODBC リターンコード, 401
SQL_INVALID_HANDLE
ODBC リターンコード, 401
SQL_NEED_DATA
ODBC リターンコード, 401
sql_needs_quotes 関数
説明, 559
SQL_NO_DATA_FOUND
ODBC リターンコード, 401
SQL_ROWSET_SIZE
複数ローのフェッチ, 11
SQL_SUCCESS
ODBC リターンコード, 401
SQL_SUCCESS_WITH_INFO
ODBC リターンコード, 401
SQL_TXN_READ_COMMITTED
独立性レベル, 393
SQL_TXN_READ_UNCOMMITTED
独立性レベル, 393
SQL_TXN_REPEATABLE_READ
独立性レベル, 393
SQL_TXN_SERIALIZABLE
独立性レベル, 393
SQL/1992
SQL プリプロセッサユーティリティ (sqlpp), 471
SQL/1999
SQL プリプロセッサユーティリティ (sqlpp), 471
SQL/2003
SQL プリプロセッサユーティリティ (sqlpp), 471
SQL/2008
SQL プリプロセッサユーティリティ (sqlpp), 471
SQLAllocHandle ODBC 関数
使用, 375
説明, 374
パラメータのバインド, 384
文の実行, 383
sqlany_affected_rows 関数 [Ruby API]
説明, 722
sqlany_affected_rows メソッド [SQL Anywhere C API]
説明, 566
sqlany_bind_param 関数 [Ruby API]
説明, 722
sqlany_bind_param メソッド [SQL Anywhere C API]
説明, 567
sqlany_cancel メソッド [SQL Anywhere C API]
説明, 567
sqlany_clear_error 関数 [Ruby API]
説明, 723
sqlany_clear_error メソッド [SQL Anywhere C API]
説明, 568
sqlany_client_version_ex メソッド [SQL Anywhere C API]
説明, 568
sqlany_client_version 関数 [Ruby API]
説明, 723
sqlany_client_version メソッド [SQL Anywhere C API]
説明, 568
sqlany_commit 関数 [Ruby API]
説明, 724
sqlany_commit メソッド [SQL Anywhere C API]
説明, 569
sqlany_connect 関数 [Ruby API]
説明, 724
sqlany_connect メソッド [SQL Anywhere C API]
説明, 569
sqlany_describe_bind_param 関数 [Ruby API]
説明, 725
sqlany_describe_bind_param メソッド [SQL Anywhere C API]
説明, 570
sqlany_disconnect 関数 [Ruby API]
説明, 725
sqlany_disconnect メソッド [SQL Anywhere C API]

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01083

説明, 571
sqlany_error 関数 [Ruby API]
説明, 726
sqlany_error メソッド [SQL Anywhere C API]
説明, 571
sqlany_execute_direct 関数 [Ruby API]
説明, 727
sqlany_execute_direct メソッド [SQL Anywhere C API]
説明, 573
sqlany_execute_immediate 関数 [Ruby API]
説明, 728
sqlany_execute_immediate メソッド [SQL Anywhere C API]
説明, 574
sqlany_execute 関数 [Ruby API]
説明, 726
sqlany_execute メソッド [SQL Anywhere C API]
説明, 572
sqlany_fetch_absolute 関数 [Ruby API]
説明, 728
sqlany_fetch_absolute メソッド [SQL Anywhere C API]
説明, 574
sqlany_fetch_next 関数
説明, 729
sqlany_fetch_next メソッド [SQL Anywhere C API]
説明, 575
sqlany_finalize_interface メソッド [SQL Anywhere C API]
説明, 575
sqlany_fini_ex メソッド [SQL Anywhere C API]
説明, 576
sqlany_fini 関数 [Ruby API]
説明, 730
sqlany_fini メソッド [SQL Anywhere C API]
説明, 576
sqlany_free_connection 関数 [Ruby API]
説明, 730
sqlany_free_connection メソッド [SQL Anywhere C API]
説明, 577
sqlany_free_stmt 関数 [Ruby API]
説明, 731
sqlany_free_stmt メソッド [SQL Anywhere C API]
説明, 577
sqlany_get_bind_param_info 関数 [Ruby API]
説明, 731
sqlany_get_bind_param_info メソッド [SQL Anywhere C API]
説明, 577
sqlany_get_column_info 関数 [Ruby API]
説明, 733
sqlany_get_column_info メソッド [SQL Anywhere C API]
説明, 579
sqlany_get_column 関数 [Ruby API]
説明, 732
sqlany_get_column メソッド [SQL Anywhere C API]
説明, 578
sqlany_get_data_info メソッド [SQL Anywhere C API]
説明, 580
sqlany_get_data メソッド [SQL Anywhere C API]
説明, 579
sqlany_get_next_result 関数 [Ruby API]
説明, 734
sqlany_get_next_result メソッド [SQL Anywhere C API]
説明, 581
sqlany_init_ex メソッド [SQL Anywhere C API]
説明, 582
sqlany_initialize_interface メソッド [SQL Anywhere C API]
説明, 583
sqlany_init 関数 [Ruby API]
説明, 734
sqlany_init メソッド [SQL Anywhere C API]
説明, 581
sqlany_make_connection_ex メソッド [SQL Anywhere C API]
説明, 584
sqlany_make_connection メソッド [SQL Anywhere C API]
説明, 583
sqlany_new_connection_ex メソッド [SQL Anywhere C API]
説明, 585
sqlany_new_connection 関数 [Ruby API]
説明, 735
sqlany_new_connection メソッド [SQL Anywhere C API]
説明, 584
sqlany_num_cols 関数 [Ruby API]
説明, 735

Copyright © 2013, SAP AG or an SAP affiliate company.

sqlany_num_cols メソッド [SQL Anywhere C API]
説明, 585
sqlany_num_params 関数 [Ruby API]
説明, 736
sqlany_num_params メソッド [SQL Anywhere C API]
説明, 586
sqlany_num_rows 関数 [Ruby API]
説明, 736
sqlany_num_rows メソッド [SQL Anywhere C API]
説明, 586
sqlany_prepare 関数 [Ruby API]
説明, 737
sqlany_prepare メソッド [SQL Anywhere C API]
説明, 587
sqlany_reset メソッド [SQL Anywhere C API]
説明, 588
sqlany_rollback 関数 [Ruby API]
説明, 738
sqlany_rollback メソッド [SQL Anywhere C API]
説明, 588
sqlany_send_param_data メソッド [SQL Anywhere C API]
説明, 588
sqlany_sqlstate 関数 [Ruby API]
説明, 738
sqlany_sqlstate メソッド [SQL Anywhere C API]
説明, 589
sqlany.cvf
データベースサーバの配備, 1027
SQLANY16 環境変数
展開設定, 1020
配備, 962
sqlanydb
Python スクリプトの作成, 661
Python データベース API, 659
UNIX と Mac OS X でのインストール, 660
Windows でのインストール, 660
説明, 659
SQLANYDIR
msiexec, 966
SQL Anywhere
C API, 565
説明, 565
sqlanywhere_en16.chm
Windows での配備, 1026
sqlanywhere_en16.map
Windows での配備, 1026
sqlanywhere.jpr
Linux、UNIX、Mac OS X での管理ツールの配備, 1018
Windows での管理ツールの配備, 1009
SQL Anywhere .NET API
SABulkCopyColumnMappingCollection クラス, 121
SABulkCopyColumnMapping クラス, 115
SABulkCopyOptions 列挙体, 330
SABulkCopy クラス, 105
SACommandBuilder クラス, 152
SACommand クラス, 129
SACommLinksOptionsBuilder クラス, 164
SAConnectionStringBuilderBase クラス, 208
SAConnectionStringBuilder クラス, 190
SAConnection クラス, 169
SADataAdapter クラス, 215
SADataReader クラス, 224
SADataSourceEnumerator クラス, 251
SADbType 列挙体, 332
SADefault クラス, 253
SAErrorCollection クラス, 255
SAError クラス, 253
SAException クラス, 258
SAFactory クラス, 261
SAInfoMessageEventArgs クラス, 267
SAInfoMessageEventHandler デリゲート, 329
SAIsolationLevel 列挙体, 338
SAMessageType 列挙体, 341
SAMetaDataCollectionNames クラス, 269
SAParameterCollection クラス, 289
SAParameter クラス, 278
SAPermissionAttribute クラス, 307
SAPermission クラス, 305
SARowsCopiedEventArgs クラス, 309
SARowsCopiedEventHandler デリゲート, 330
SARowUpdatedEventArgs クラス, 310
SARowUpdatedEventHandler デリゲート, 329
SARowUpdatingEventArgs クラス, 313
SARowUpdatingEventHandler デリゲート, 329
SAServerSideConnection クラス, 315
SATcpOptionsBuilder クラス, 316
SATransaction クラス, 324
説明, 41
SQL Anywhere .NET API リファレンス
説明, 105
SQL Anywhere .NET データプロバイダ
サンプル, 78

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01085

説明, 41
SQL Anywhere .NET データプロバイダの配備
説明, 72
SQL Anywhere 16 Demo.dsn
Windows Mobile ODBC, 368
SQL Anywhere ASP.NET データプロバイダ
説明, 97
SQL Anywhere C API
a_sqlany_bind_param_info 構造体, 593
a_sqlany_bind_param 構造体, 593
a_sqlany_column_info 構造体, 594
a_sqlany_data_direction 列挙, 590
a_sqlany_data_info 構造体, 595
a_sqlany_data_type 列挙, 590
a_sqlany_data_value 構造体, 596
a_sqlany_native_type 列挙, 591
C API, 565
sqlany_affected_rows メソッド, 566
sqlany_bind_param メソッド, 567
sqlany_cancel メソッド, 567
sqlany_clear_error メソッド, 568
sqlany_client_version_ex メソッド, 568
sqlany_client_version メソッド, 568
sqlany_commit メソッド, 569
sqlany_connect メソッド, 569
sqlany_describe_bind_param メソッド, 570
sqlany_disconnect メソッド, 571
sqlany_error メソッド, 571
sqlany_execute_direct メソッド, 573
sqlany_execute_immediate メソッド, 574
sqlany_execute メソッド, 572
sqlany_fetch_absolute メソッド, 574
sqlany_fetch_next メソッド, 575
sqlany_finalize_interface メソッド, 575
sqlany_fini_ex メソッド, 576
sqlany_fini メソッド, 576
sqlany_free_connection メソッド, 577
sqlany_free_stmt メソッド, 577
sqlany_get_bind_param_info メソッド, 577
sqlany_get_column_info メソッド, 579
sqlany_get_column メソッド, 578
sqlany_get_data_info メソッド, 580
sqlany_get_data メソッド, 579
sqlany_get_next_result メソッド, 581
sqlany_init_ex メソッド, 582
sqlany_initialize_interface メソッド, 583
sqlany_init メソッド, 581
sqlany_make_connection_ex メソッド, 584
sqlany_make_connection メソッド, 583
sqlany_new_connection_ex メソッド, 585
sqlany_new_connection メソッド, 584
sqlany_num_cols メソッド, 585
sqlany_num_params メソッド, 586
sqlany_num_rows メソッド, 586
sqlany_prepare メソッド, 587
sqlany_reset メソッド, 588
sqlany_rollback メソッド, 588
sqlany_send_param_data メソッド, 588
sqlany_sqlstate メソッド, 589
SQLAnywhereInterface 構造体, 596
インタフェースライブラリ, 566
外部環境, 623
SQL Anywhere C API リファレンス
sacapi.h ヘッダファイル, 566
sacapidll.h ヘッダファイル, 566
SQLAnywhereInterface 構造体 [SQL Anywhere C API]
説明, 596
SQL Anywhere JDBC ドライバ
4.0 ドライバのロード, 427
JDBC クライアントの配備, 994
JDBC ドライバの選択, 424
URL, 427
コンポーネント, 994
使用, 427
接続, 427
説明, 423
必要なファイル, 427
SQL Anywhere ODBC ドライバ
ANSI, 372
Windows でのリンク, 367
インストール, 982
コンポーネント, 982
データソースの定義, 988
ドライバ名, 376
配備, 982
配備のための名前のカスタマイズ, 986
レジストリの設定, 986
SQL Anywhere Perl DBD::SQLAnywhere DBI モジュール
説明, 649
SQL Anywhere PHP API
説明, 682
SQL Anywhere PHP 拡張
設定, 1002
説明, 667

Copyright © 2013, SAP AG or an SAP affiliate company.

SQL Anywhere PHP モジュール
API リファレンス, 682
SQL Anywhere Python データベースサポート
説明, 659
SQL Anywhere Ruby API
関数, 721
SQL Anywhere Web サービス
説明, 749
SQL Anywhere データベースツール C API リファレンス
dbtools.h ヘッダファイル, 868
SQL Anywhere プラグイン
配備に関する考慮事項, 1005
SQLBindCol ODBC 関数
ストレージのアラインメント, 391
説明, 395
パラメータサイズ, 387
SQLBindParameter ODBC 関数
ストアドプロシージャ, 397
ストレージのアラインメント, 391
説明, 384
パラメータサイズ, 387
SQLBindParameter 関数
ODBC 準備文, 4
準備文, 385
SQLBindParam ODBC 関数
パラメータサイズ, 387
SQLBrowseConnect ODBC 関数
説明, 376
SQLBulkOperations
ODBC 関数, 12
SQLCA
スレッド, 500
説明, 498
長さ, 498
フィールド, 498
複数の管理, 502
変更, 500
sqlcabc SQLCA フィールド
説明, 498
sqlcaid SQLCA フィールド
説明, 498
sqlcode SQLCA フィールド
説明, 498
SQLColAttribute ODBC 関数
パラメータサイズ, 387
SQLColAttributes ODBC 関数
パラメータサイズ, 387
SQL Communications Area
説明, 498
SQLConnect ODBC 関数
Windows のパフォーマンス, 989
説明, 376
SQLCOUNT
sqlerror SQLCA フィールドの要素, 499
SQLDA
fill_sqlda_ex を使った割り付け, 557
fill_sqlda を使った割り付け, 556
sqllen フィールド, 509
解放, 556
記述子, 35
説明, 506
動的 SQL, 503
フィールド, 507
ホスト変数, 508
文字列と fill_sqlda, 556
文字列と fill_sqlda_ex, 557
割り付け, 532
sqlda_storage 関数
説明, 560
sqlda_string_length 関数
説明, 560
sqldabc SQLDA フィールド
説明, 507
sqldaid SQLDA フィールド
説明, 507
sqldata SQLDA フィールド
説明, 508
SQLDATETIME データ型
Embedded SQL, 490
sqldef.h
ソフトウェアの終了コードの検索, 957
データ型, 485
SQLDescribeCol ODBC 関数
パラメータサイズ, 387
SQLDescribeParam ODBC 関数
パラメータサイズ, 387
SQLDriverConnect ODBC 関数
説明, 376
sqld SQLDA フィールド
説明, 507
sqlerrd SQLCA フィールド
説明, 499
sqlerrmc SQLCA フィールド
説明, 499
sqlerrml SQLCA フィールド

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01087

説明, 499
sqlerror_message 関数
説明, 561
SQLError ODBC 関数
説明, 401
sqlerror SQLCA フィールド
SQLCOUNT, 499
SQLIOCOUNT, 499
SQLIOESTIMATE, 500
要素, 499
sqlerrp SQLCA フィールド
説明, 499
SQLExecDirect ODBC 関数
説明, 383
バウンドパラメータ, 384
SQLExecute 関数
ODBC 準備文, 4
SQLExtendedFetch
ODBC 関数, 10
複数ローのフェッチ, 11
SQLExtendedFetch ODBC 関数
ストアドプロシージャ, 397
説明, 395
パラメータサイズ, 387
SQLFetch
ODBC 関数, 10
SQLFetch ODBC 関数
ストアドプロシージャ, 397
説明, 395
SQLFetchScroll
ODBC 関数, 10
複数ローのフェッチ, 11
SQLFetchScroll ODBC 関数
説明, 395
パラメータサイズ, 387
SQLFreeHandle ODBC 関数
使用, 375
SQLFreeStmt 関数
ODBC 準備文, 4
SQLGetConnectAttr ODBC 関数
説明, 379
SQLGetData ODBC 関数
ストレージのアラインメント, 391
説明, 395
パラメータサイズ, 387
SQLGetDescRec ODBC 関数
パラメータサイズ, 387
sqlind SQLDA フィールド
説明, 508
SQLIOCOUNT
sqlerror SQLCA フィールドの要素, 499
SQLIOESTIMATE
sqlerror SQLCA フィールドの要素, 500
SQLJ 標準
説明, 405
sqllen SQLDA フィールド
DESCRIBE 文, 509
値, 509
値の記述, 509
値の取得, 514
値の送信, 512
説明, 508
SQLLEN と SQLINTEGER
ODBC, 387
sqlname SQLDA フィールド
説明, 508
SQLNumResultCols ODBC 関数
ストアドプロシージャ, 397
SQLParamOptions ODBC 関数
パラメータサイズ, 387
sqlpp ユーティリティ
構文, 471
サポートされているコンパイラ, 476
配備, 471
プリプロセッサのオプション, 471
SQLPrepare 関数
ODBC 準備文, 4
説明, 385
SQLPutData ODBC 関数
パラメータサイズ, 387
SQL Remote
配備, 1041
SQLRETURN
ODBC リターンコードのタイプ, 401
SQLRowCount ODBC 関数
パラメータサイズ, 387
SQL Server
マッピングファイルの配備, 972
SQLSetConnectAttr
ODBC アプリケーション, 380
SQLSetConnectAttr ODBC 関数
説明, 378
トランザクションの独立性レベル, 393
SQLSetConnectOption ODBC 関数
パラメータサイズ, 387
SQLSetDescRec ODBC 関数

Copyright © 2013, SAP AG or an SAP affiliate company.

パラメータサイズ, 387
SQLSetParam ODBC 関数
パラメータサイズ, 387
SQLSetPos ODBC 関数
説明, 396
パラメータサイズ, 387
SQLSetScrollOptions ODBC 関数
パラメータサイズ, 387
SQLSetStmtAttr ODBC 関数
カーソル特性, 394
SQLSetStmtOption ODBC 関数
パラメータサイズ, 387
sqlstate SQLCA フィールド
説明, 499
SqlState プロパティ
SAError クラス [SQL Anywhere .NET API], 255
SQLTransact ODBC 関数
説明, 376
sqltype SQLDA フィールド
DESCRIBE 文, 509
説明, 508
SQLULEN と SQLUINTEGER
ODBC, 387
sqlvar SQLDA フィールド
説明, 507
内容, 508
sqlwarn SQLCA フィールド
説明, 499
SQL アプリケーション
SQL 文の実行, 1
SQL プリプロセッサユーティリティ (sqlpp)
構文, 471
実行, 471
説明, 471
SQL 文
Web クライアント, 802
Web サービス, 762
実行, 745
SQL 文の実行
アプリケーション, 1
SSDLToSA16.tt
.NET 4.x クライアントの配備, 972
SSIS
マッピングファイルの配備, 972
StartLine プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 207
StateChange イベント
SAConnection クラス [SQL Anywhere .NET API], 189
State プロパティ
.NET データプロバイダ, 47
SAConnection クラス [SQL Anywhere .NET API], 189
stax-api-1.0.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
strlen_or_ind
ODBC, 387
Sybase Central
JAR ファイルの追加, 415
Java クラスの追加, 414
jConnect メタデータサポートのインストール, 429
Linux、UNIX、Mac OS X での管理ツールの配備, 1018
Linux と UNIX での配備, 1014
Windows での管理ツールの配備, 1008
Windows での配備, 1005
ZIP ファイルの追加, 415
配備, 1004
配備に関する考慮事項, 1005
配備用の設定, 1022
sybasecentral1600.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
Sybase Open Client のサポート
説明, 741
symlink
UNIX での配備, 961
SyncRoot プロパティ
SAParameterCollection クラス [SQL Anywhere .NET API], 303
System.Transactions
使用, 64

T

TableMappings プロパティ
SADataAdapter クラス [SQL Anywhere .NET API], 221
Tables フィールド
SAMetaDataCollectionNames クラス [SQL Anywhere .NET API], 276

TableViewer
.NET データプロバイダサンプルプロジェクト, 43
TcpOptionsBuilder プロパティ
SACommLinksOptionsBuilder クラス [SQL Anywhere .NET API], 168
TcpOptionsString プロパティ
SACommLinksOptionsBuilder クラス [SQL Anywhere .NET API], 168
TDS プロパティ
SATcpOptionsBuilder クラス [SQL Anywhere .NET API], 323
this プロパティ
SABulkCopyColumnMappingCollection クラス [SQL Anywhere .NET API], 129
SAConnectionStringBuilderBase クラス [SQL Anywhere .NET API], 214
SADataReader クラス [SQL Anywhere .NET API], 250
SAErrorCollection クラス [SQL Anywhere .NET API], 257
SAParameterCollection クラス [SQL Anywhere .NET API], 304
Timeout プロパティ
SATcpOptionsBuilder クラス [SQL Anywhere .NET API], 324
TimeSpan
SQL Anywhere .NET データプロバイダ, 62
TIMESTAMP データ型
Open Client の変換, 744
Time 構造体
.NET データプロバイダの時間値, 62
ToString メソッド
SACommLinksOptionsBuilder クラス [SQL Anywhere .NET API], 167
SAError クラス [SQL Anywhere .NET API], 254
SAInfoMessageEventArgs クラス [SQL Anywhere .NET API], 268
SAParameter クラス [SQL Anywhere .NET API], 283
SATcpOptionsBuilder クラス [SQL Anywhere .NET API], 320
TransactionScope クラス
使用, 64
Transaction プロパティ
SACommand クラス [SQL Anywhere .NET API], 151
TryGetValue メソッド
SAConnectionStringBuilderBase クラス [SQL Anywhere .NET API], 213
TYPE 句
指定, 793
例, 804

U

ulplugin.jar
Windows での配備, 1007
ulscutil16.dll
Windows での配備, 1007
ultralite.jpr
Linux、UNIX、Mac OS X での管理ツールの配備, 1019
Windows での管理ツールの配備, 1010
Unconditional プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 207
Unicode
Windows Mobile 用の ODBC アプリケーションのリンク, 369
UNIX
ODBC, 369
ODBC ドライバマネージャ, 370
ディレクトリ構造, 960
配備, 960
マルチスレッドアプリケーション, 961
unixODBC
ドライバマネージャ, 371
unixodbc.h
コンパイルプラットフォーム, 387
説明, 367
unload ユーティリティ (dbunload)
バージョン 9 以前のデータベースの配備, 1040
UnquoteIdentifier メソッド
SACommandBuilder クラス [SQL Anywhere .NET API], 163
UNSIGNED BIGINT データ型
Embedded SQL, 490
UpdateBatch ADO メソッド
ADO プログラミング, 350
データの更新, 350
UpdateBatchSize プロパティ
SADataAdapter クラス [SQL Anywhere .NET API], 222
UpdateCommand プロパティ

SADataAdapter クラス [SQL Anywhere .NET API], 223
UpdatedRowSource プロパティ
SACommand クラス [SQL Anywhere .NET API], 152
UPDATE 文
JDBC, 440
位置付け, 12
URL
jConnect, 430
SQL Anywhere JDBC ドライバ, 427
解釈, 782
セッション管理, 774
データベース, 430
変数の指定, 803
URL 句
指定, 791
User-Agent
HTTP ヘッダへのアクセス, 768
UserDefinedTypes フィールド
SAMetaDataCollectionNames クラス [SQL Anywhere .NET API], 276
UserID プロパティ
SAConnectionStringBuilder クラス [SQL Anywhere .NET API], 208
Users フィールド
SAMetaDataCollectionNames クラス [SQL Anywhere .NET API], 276
UTF-32
ODBC ドライバマネージャ, 372

V

value-sensitive カーソル
概要, 17
更新の例, 20
説明, 25
例の削除, 18
Value フィールド
SADefault クラス [SQL Anywhere .NET API], 253
Value プロパティ
SAParameter クラス [SQL Anywhere .NET API], 288
VARCHAR データ型
ESQL, 490
velocity.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
velocity-dep.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007
VerifyServerName プロパティ
SATcpOptionsBuilder クラス [SQL Anywhere .NET API], 324
ViewColumns フィールド
SAMetaDataCollectionNames クラス [SQL Anywhere .NET API], 277
Views フィールド
SAMetaDataCollectionNames クラス [SQL Anywhere .NET API], 277
Visual Basic
.NET データプロバイダでのサポート, 41
チュートリアル, 87
Visual C#
チュートリアル, 87
Visual C++
Embedded SQL のサポート, 476
VM
Java VM, 407
Java の起動, 420
Java の停止, 420
シャットダウンフック, 420

W

WebClientLogFile プロパティ
サーバオプション, 823
WebClientLogging プロパティ
サーバオプション, 823
Web クライアント
HTTP 要求ヘッダの管理, 795
SOAP ネームスペースの管理, 801
SOAP ヘッダの管理, 797
SQL 文, 802
URL 句, 791
関数とプロシージャ, 790
関数とプロシージャの要件, 791
クイックスタート, 787
結果セットの取得, 808
結果セットへのアクセス, 806
説明, 786
代入パラメータ, 820
変数の指定, 803

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01091

ポートの指定, 794
要求タイプの指定, 793
Web サーバ
PHP API, 667
アプリケーションの開発, 764
クイックスタート, 749
説明, 749
複数の起動, 753
プロトコルの設定, 751
プロトコルの有効化, 751
Web サービス
HTTP_HEADER の例, 767
HTTP_VARIABLE の例, 767
HTTP 変数とヘッダのアクセス, 766
MIME タイプチュートリアル, 825
NEXT_HTTP_HEADER の例, 767
NEXT_HTTP_VARIABLE の例, 767
root, 760
SOAP/DISH チュートリアル, 830
SOAP のデータ型, 815
SOAP ヘッダのアクセス, 770
SQL 文, 762
URL の解釈, 782
Web サービス関連システムプロシージャのリスト, 780
アクセス, 786
アルファベット順のシステムプロシージャ, 780
エラー, 823
オプション, 782
開発, 764
管理, 754, 756
クライアントログファイル、説明, 823
クロスサイトスクリプティング, 780
結果セットの取得, 808
結果セットへのアクセス, 806
構造体型, 816
コメント付け, 760
削除, 760
作成, 756
セッションの管理, 772
接続プロパティ, 781
説明, 749
タイプ, 754
データ型, 809
配列型, 816
プロトコルの設定, 751
プロトコルの有効化, 751
プーリング, 763
変更, 756
ホスト変数, 766
文字セット, 779
ロギング, 823
Web ページ
カスタマイズ, 764
スクリプトの実行, 668
WITH HOLD 句
カーソル, 10
Windows
OLE DB サポート, 343
管理ツールの配備, 1005
サイレントインストール, 966
WindowsAccessBridge.dll
Java ベースの管理ツールの配備, 1011
Windows Mobile
ODBC, 368
OLE DB サポート, 343
Wow6432Node
レジストリエントリ, 989
WRITE_CLIENT_FILE 関数
ESQL クライアント API コールバック関数, 546
ODBC クライアント API コールバック関数, 380
WriteToServer メソッド
SABulkCopy クラス [SQL Anywhere .NET API], 109
wsimport
JAX-WS と Web サービス, 850
wstx-asl-3.2.6.jar
Mac OS X での配備, 1016
UNIX での配備, 1016
Windows での配備, 1007

X

XMLCONCAT 関数
例, 765
XMLELEMENT 関数
例, 765
XML サービス
Web クライアントのクイックスタート, 787
Web サーバのクイックスタート, 749
クイックスタート, 789
コメント付け, 760
削除, 760

作成, 756
説明, 754
XSS
Web サービス, 780

あ

アクセシビリティ
Java ベースの管理ツールの配備, 1005
アップグレードユーティリティ (dbupgrad)
jConnect メタデータサポートのインストール, 429
アプリケーション
.NET クライアントの配備, 972
Embedded SQL の配備, 992
JDBC クライアントの配備, 994
ODBC の配備, 982
OLE DB の配備, 976
Open Client の配備, 1003
PHP クライアントの配備, 996
SQL, 1
クライアントアプリケーションの配備, 972
配備, 959
アンインストール
.NET データプロバイダ, 975
アンインストールオプション
SetupVSPackage, 975
アンインストールオールオプション
SetupVSPackage, 975
アンマネージコード
.NET データプロバイダ, 974
dbdata.dll, 73
アンロードユーティリティ (dbunload)
配備に関する考慮事項, 1039
アンロード列挙 [データベースツール API]
説明, 881
アーキテクチャ
OData サーバ, 457

い

位置付け DELETE 文
説明, 12
位置付け UPDATE 文
説明, 12
位置付け更新
説明, 10
イベントログ
レジストリエントリ, 1030
インジケータ
ワイドフェッチ, 519
インジケータ変数
NULL, 496
ODBC, 387
SQLDA, 508
値のまとめ, 497
説明, 495
データ型変換, 497
トランケーション, 497
インストーラ
サイレントインストール, 968
インストール
JAR ファイルをデータベースに, 415
Java クラスをデータベースに, 413
jConnect メタデータサポート, 429
SQL を使用して Java クラスをデータベースに, 414
Sybase Central を使用して Java クラスをデータベースに, 414
サイレントインストール, 968
製品コード, 972
登録キー, 968
インストールオプション
SetupVSPackage, 972
インストールキー
サイレントインストール, 968
インストールプログラム
配備, 960
インタフェース
SQL Anywhere Embedded SQL, 469
インタフェースライブラリ
DBLIB, 469
SQL Anywhere C API, 566
動的ロード, 479
インポートライブラリ
DBTools, 862
Embedded SQL, 477
ODBC, 367
Windows Mobile ODBC, 368
基本説明, 470
代替方法, 479
引用符付き識別子
sql_needs_quotes 関数, 559

う

埋め込み列挙 [データベースツール API]

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01093

説明, 880

え

エクスポートファイル
dblib.def, 477
エスケープ構文
Interactive SQL, 452
ODBC, 399
エラーコード
SQL Anywhere の終了コード, 956
エラー
HTTP コード, 823
SOAP フォールト, 823
sqlcode SQLCA フィールド, 498
エラー処理
Java, 407
ODBC, 401
SQL Anywhere .NET データプロバイダ, 65
エラーメッセージ
ESQL 関数, 561
エントリポイント
DBTools 関数の呼び出し, 863
エンリスト
分散トランザクション, 857

お

お気に入りリスト
設定可能なオプション, 1022
オブジェクト
記憶形式, 415
オプション
OData, 459
ODBC アプリケーション, 379
Web サービス, 782
オペレーティングシステム
ファイル名, 962
オンラインバックアップ
Embedded SQL, 531
オートコミット
JDBC, 438
ODBC, 376
実装, 38
制御, 37
トランザクション用の設定, 36
オートチューン列挙 [データベースツール API]
説明, 878
オーバフローエラー
Open Client データ型の変換, 744

か

開始
jConnect を使用したデータベース, 430
外部オブジェクト
コメント, 619
説明, 619
外部環境
C API, 623
CLR, 620
Embedded SQL, 623
JAVA, 632
ODBC, 623
PERL, 638
PHP, 642
説明, 619
外部関数
取り消し, 614
パラメータを渡す, 612
プロトタイプ, 603
ライブラリのアンロード, 618
リターンタイプ, 618
外部関数呼び出し
外部環境のサポート, 619
説明, 601
外部関数呼び出しインタフェース
an_extfn_api, 606
an_extfn_result_set_column_data, 611
an_extfn_result_set_column_info, 610
an_extfn_result_set_info, 609
an_extfn_value, 607
extfn_cancel メソッド, 604
extfn_post_load_library メソッド, 605
extfn_pre_unload_library メソッド, 605
extfn_use_new_api メソッド, 603
get_piece コールバック, 612
get_value コールバック, 612
set_cancel コールバック, 614
set_value コールバック, 613
プロトタイプ, 603
外部プロシージャコール
説明, 601
外部プロシージャの呼び出し
外部環境のサポート, 619
外部呼び出し
プロシージャとファンクションの作成, 601

Copyright © 2013, SAP AG or an SAP affiliate company.

各種のカーソル
ODBC, 33
活性
タイムアウトコールバック, 547
環境ハンドル
ODBC, 374
環境変数
CLASSPATH, 408
JAVA_HOME, 409
JAVAHOME, 409
データベース内の Java, 409
関数
CLR, 620
DBTools 関数の呼び出し, 863
Embedded SQL, 532
SQL Anywhere PHP モジュール, 682
Web クライアント, 790
Web クライアントの要件, 791
外部, 601
感知性
SQL Anywhere のカーソル, 16
カーソル, 17
更新の例, 20
独立性レベル, 31
例の削除, 18
管理ツール
dbtools, 861
配備, 1004
カーソル
ADO, 33
ADO.NET, 32
asensitive, 24
C コード例, 481
db_cancel_request 関数, 538
DYNAMIC SCROLL と asensitive カーソル, 24
DYNAMIC SCROLL とカーソル位置, 10
Embedded SQL での使用, 517
Embedded SQL のサポートされる型, 34
insensitive, 22
ODBC, 393
ODBC カーソル特性の選択, 394
ODBC 結果セット, 395
ODBC と SQL Anywhere のタイプ, 33
ODBC の更新, 396
ODBC の削除, 396
ODBC ブックマーク, 397
OLE DB, 33
Open Client, 745
Python, 663
SCROLL, 25
sensitive, 23
SQL Anywhere での感知性, 16
value-sensitive, 25
値, 16
位置, 9
一意性, 15
可用性, 15
感知性, 15
感知性と更新の例, 20
感知性と削除の例, 18
感知性と独立性レベル, 31
感知性とパフォーマンス, 27
感知性の概要, 17
キャンセル, 14
キーセット駆動型, 25
結果セット, 5
結果セットの記述, 34
更新, 349, 746
更新可能性, 15
削除, 746
順序, 16
準備文, 8
使用, 6, 8
スクロール可能, 12
スクロール動作, 15
ストアドプロシージャ, 529
制限, 9
静的, 22
接続用に存在するカーソルの特定, 5
接続用のカーソルの内容の表示, 5
説明, 6
セーブポイント, 39
動的, 23
独立性レベル, 10
トランザクション, 39
内部, 16
表示可能な変更, 17
ファット, 11
複数のローの挿入, 12
複数ローのフェッチ, 11
プラットフォーム, 15
プリフェッチのパフォーマンス, 28
ブロックカーソル, 16
プロパティ, 15
未指定の感知性, 24
メンバーシップ, 16

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01095

要求, 32
読み込み専用, 22
利点, 7
ローの更新と削除, 12
ローの挿入, 12
ローのフェッチ, 10
ワークテーブル, 27
カーソル位置
トラブルシューティング, 10
カーソルとブックマーク
説明, 16
カーソルの感知性とパフォーマンス
説明, 27

き

記述
Embedded SQL の NCHAR カラム, 509
結果セット, 34
記述子
結果セットの記述, 34
規則
ファイル名, 962
機能
サポートされる, 747
行の長さ
SQL プリプロセッサ出力, 471
行番号
SQL プリプロセッサユーティリティ (sqlpp), 471
共有オブジェクト
外部プロシージャの呼び出し, 602
ストアドプロシージャからの呼び出し, 601
キーセット駆動型カーソル
ODBC, 33
value-sensitive, 25

く

句
WITH HOLD, 10
クイックスタート
SQL Anywhere Web クライアント, 787
SQL Anywhere Web サーバ, 749
SQL Anywhere Web サーバへのアクセス, 789
クエリ
ADO Recordset オブジェクト, 347
ADO Recordset オブジェクトとカーソル, 348
シングルロー, 516
組み込みデータベース
配備, 1038
クライアント
Web, 786
時刻の変更, 555
クライアント側オートコミット
説明, 38
クライアントファイル
ESQL クライアント API コールバック関数, 546
ODBC クライアント API コールバック関数, 380
クラス
インストール, 413
更新, 415
作成, 414
バージョン, 415
クラスファイル
データベースへのインストール, 414
クロスサイトスクリプティング
Web サービス, 780

け

結果セット
ADO Recordset オブジェクトの使用, 348
ADO Recordset オブジェクトの説明, 347
JDBC, 446
ODBC, 393
ODBC ストアドプロシージャ, 397
ODBC の取り出し, 395
Open Client, 747
Web クライアントからのアクセス, 806
Web サービスからの取得, 808
カーソル, 5
使用, 8
ストアドプロシージャ, 529
データベース内の Java ストアドプロシージャ, 418
複数の ODBC, 397
メタデータ, 34
権限
JDBC, 448
言語
ファイル名, 962
検証列挙 [データベースツール API]
説明, 882

Copyright © 2013, SAP AG or an SAP affiliate company.

こ

更新
　カーソル, 349
更新内容の消失
　説明, 29
構造のパック
　ヘッダファイル, 477
高速ランチャ
　OEM の設定可能なオプション, 1022
考慮事項
　OData サーバ, 458
コマンド
　ADO Command オブジェクト, 345
コマンドラインユーティリティ
　SQL プリプロセッサ (sqlpp) の構文, 471
　配備, 1038
コミット
　ODBC からのトランザクション, 376
コメント付け
　Web サービス, 760
コロンホスト変数名
　SQLBindParameter 関数, 385
コンソールユーティリティ (dbconsole)
　Linux と UNIX での配備, 1014
　Windows での配備, 1005
　配備, 1004
コンテナ
　OData メタデータ, 465
コンパイラ
　sqlpp での使用, 476
コンパイルとリンクの処理
　説明, 470
コールバック
　DB_CALLBACK_CONN_DROPPED, 547
　DB_CALLBACK_DEBUG_MESSAGE, 547
　DB_CALLBACK_FINISH, 547
　DB_CALLBACK_MESSAGE, 548
　DB_CALLBACK_START, 547
　DB_CALLBACK_VALIDATE_FILE_TRANSFE
　R, 549
　DB_CALLBACK_WAIT, 548
　JDBC, 448
コールバック関数
　Embedded SQL, 531
　登録, 546

さ

再入可能コード
　マルチスレッド Embedded SQL の例, 500
サイレントアンインストール
　msiexec, 966
サイレントインストール
　msiexec, 966
　説明, 968
削除
　.NET データプロバイダ, 975
　Web サービス, 760
作成
　Web サービス, 756
　外部呼び出しを使ったプロシージャと関数,
　601
サポートするプラットフォーム
　OLE DB, 344
サンプル
　.NET データプロバイダ, 78
　Embedded SQL, 482, 484
　Embedded SQL アプリケーション, 481
　Embedded SQL 内の静的カーソル, 482
　Embedded SQL 内の動的カーソル, 483
　OData サーバとクライアント, 466
　SimpleViewer, 87
サーバ
　ESQL からの検索, 555
　Web, 749
サーバアドレス
　ESQL 関数, 542
サーバエクスプローラー
　Visual Studio, 87
サーバ側オートコミット
　説明, 38
サービス
　OData プロデューサモデル, 465
　Web, 749
　データ型, 809

し

時間
　.NET データプロバイダを使用した取得, 62
時間値の取得
　説明, 62
識別子
　引用符が必要な識別子, 559
シグネチャ

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01097

Java メソッド, 634
システムデータソース
Windows, 988
システムの稼働条件
SQL Anywhere .NET データプロバイダ, 72
システムプロシージャ
HTTP, 780
SOAP, 780
終了コード
説明, 956
手動コミットモード
実装, 38
制御, 37
トランザクション, 36
取得
SQLDA, 514
準拠
ODBC, 366
準備
コミット, 857
文, 2
準備文
ADO.NET の概要, 3
JDBC, 442
ODBC, 385
Open Client, 745
カーソル, 8
削除, 3
使用, 2
バインドパラメータ, 3
冗長列挙 [データベースツール API]
説明, 883
初期化ユーティリティ (dbinit)
配備に関する考慮事項, 1039
シングルスレッドアプリケーション
UNIX, 369
進行メッセージ
ODBC, 380

す

スキーマ
SQL Anywhere ASP.NET プロバイダの追加, 98
Web パーツパーソナル化プロバイダ, 104
プロファイルプロバイダ, 103
ヘルスモニタリングプロバイダ, 104
メンバーシッププロバイダ, 102
ロールプロバイダ, 103
スクロール可能カーソル
JDBC サポート, 424
スクロール可能なカーソル
説明, 12
ステートメントハンドル
ODBC, 374
ストアド関数
外部関数の呼び出し, 601
ストアドプロシージャ
CLR, 620
Embedded SQL での作成, 528
Embedded SQL での実行, 528
ESQL の結果セット, 529
INOUT パラメータと Java, 419
OUT パラメータと Java, 419
SQL Anywhere .NET データプロバイダ, 62
外部関数の呼び出し, 601
データベース内の Java, 418
スナップショットアイソレーション
SQL Anywhere .NET データプロバイダ, 64
更新内容の消失, 30
スレッド
Embedded SQL での複数スレッドの管理, 500
ODBC, 366
ODBC アプリケーション, 379
UNIX での開発, 369
データベース内の Java, 417
複数の SQLCA, 502
スレッドアプリケーション
UNIX, 961

せ

制限事項
OData サーバ, 458
静的 SQL
説明, 503
静的カーソル
ODBC, 33
説明, 22
製品コード
SQL Anywhere, 972
セキュリティ
Interactive SQL, 1022
OData サーバ, 458
データベース内の Java, 420
セキュリティコンテキスト
リンクサーバ, 355, 357

Copyright © 2013, SAP AG or an SAP affiliate company.

セキュリティマネージャ
説明, 420
セッション
エラー, 779
管理, 778
検出, 776
削除, 776
作成, 773
説明, 772
接続
.NET データプロバイダを使用してデータベースに接続する, 45
ADO Connection オブジェクト, 344
jConnect, 430
jConnect URL, 430
JDBC, 426
JDBC クライアントアプリケーション, 431
JDBC サーバ側の例, 435
JDBC デフォルト, 438
JDBC トランザクションの独立性レベル, 438
JDBC の例, 431
ODBC 関数, 376
ODBC 属性の取得, 379
ODBC 属性の設定, 378
ODBC プログラミング, 377
SQL Anywhere JDBC ドライバ URL, 427
Web アプリケーションのライセンス, 772
関数, 553
サーバの JDBC, 435
接続状態
.NET データプロバイダ, 47
接続パラメータ
OLE DB, 351
接続ハンドル
ODBC, 374
接続プロパテ
Web サービス, 781
接続プロファイル
Linux、Solaris、Mac OS X における管理ツールの配備, 1021
Windows での管理ツールの配備, 1013
接続プーリング
.NET データプロバイダ, 46
OLE DB, 353
Web サービス, 763
設定
Interactive SQL、配備用, 1022
SQLDA を使った値, 512
Sybase Central、配備用, 1022
管理ツール、配備用, 1022
設定ファイル
OData サーバ, 459
宣言
ESQL データ型, 485
ホスト変数, 489
宣言セクション
説明, 489
セーブポイント
カーソル, 39

そ

挿入
JDBC, 445
ソフトウェア
リターンコード, 956

た

単位列挙 [データベースツール API]
説明, 880

ち

チェックポイント列挙 [データベースツール API]
説明, 879
チュートリアル
.NET データプロバイダの Simple コードサンプルの使用, 78
.NET データプロバイダの Table Viewer コードサンプルの使用, 82
JAX-WS を使用した SOAP/DISH Web サービスへのアクセス, 846
SQL Anywhere を使用した SOAP サービスへのアクセス, 830
Visual C# を使用した SOAP/DISH Web サービスへのアクセス, 839
Web サーバを作成し、Web クライアントからアクセスします, 825
単純な .NET データベースアプリケーションの開発, 87
データベース内の Java の使用, 407
直列化
テーブル内のオブジェクト, 415

つ

追加
JAR ファイル, 415

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01099

データベースの Java クラス, 414

て

停電
EnableFlush レジストリエントリ, 1029
ディレクトリ構造
UNIX, 960
テンポラリオプション
ODBC アプリケーションでの設定, 379
データ
.NET データプロバイダを使用したアクセス, 47
.NET データプロバイダを使用した操作, 47
データ型
C データ型, 490
Embedded SQL, 485
Open Client の範囲, 744
Open Client マッピング, 743
SQLDA, 508
SQL と C, 616
Web サービスハンドラ内, 809
動的 SQL, 506
ホスト変数, 490
データ型変換
インジケータ変数, 497
データグリッドコントロール
Visual Studio, 92
データ接続
Visual Studio, 87
データソース
FileDSN 用の DefaultDSNDir, 988
システム DSN, 988
配備, 988
ファイル DSN, 988
ユーザ DSN, 988
データのアラインメント
ODBC, 391
データの挿入
マルチロー, 519
ワイド挿入, 519
データベース
Java クラスの格納, 406
jConnect メタデータサポートのインストール, 429
URL, 430
配備, 1032
データベースアップグレードウィザード
jConnect メタデータサポートのインストール, 429
データベースオプション
jConnect での設定, 431
データベース管理
dbtools, 861
データベース作成ウィザード
配備に関する考慮事項, 1005
データベースサーバ
関数, 553
配備, 1027
データベースツール API
a_backup_db 構造体, 885
a_change_log 構造体, 889
a_create_db 構造体, 892
a_db_info 構造体, 897
a_db_version_info 構造体, 901
a_dblic_info 構造体, 902
a_dbtools_info 構造体, 904
a_log_file_info 構造体, 904
a_name 構造体, 905
a_remote_sql 構造体, 905
a_sync_db 構造体, 916
a_syncpub 構造体, 929
a_sysinfo 構造体, 930
a_table_info 構造体, 931
a_translate_log 構造体, 932
a_truncate_log 構造体, 938
a_validate_db 構造体, 940
an_erase_db 構造体, 942
an_unload_db 構造体, 943
an_upgrade_db 構造体, 953
DBBackup メソッド, 869
DBChangeLogName メソッド, 869
DBCreatedVersion メソッド, 870
DBCreate メソッド, 870
DBErase メソッド, 871
DBInfoDump メソッド, 872
DBInfoFree メソッド, 872
DBInfo メソッド, 871
DBLicense メソッド, 873
DBLogFileInfo メソッド, 873
DBRemoteSQL メソッド, 873
DBSynchronizeLog メソッド, 874
DBToolsFini メソッド, 874
DBToolsInit メソッド, 875
DBToolsVersion メソッド, 875
DBTranslateLog メソッド, 876

DBTruncateLog メソッド, 876
DBUnload メソッド, 877
DBUpgrade メソッド, 877
DBValidate メソッド, 878
アンロード列挙, 881
埋め込み列挙, 880
オートチューン列挙, 878
検証列挙, 882
冗長列挙, 883
単位列挙, 880
チェックポイント列挙, 879
バージョン列挙, 884
ユーザリスト列挙, 882
履歴列挙, 880
データベースツールインタフェース
説明, 861
データベースツールライブラリ
説明, 861
データベース内の Java
Java VM, 407
Java VM の選択, 409
main メソッド, 417
NoSuchMethodException, 418
VM シャットダウンフック, 420
VM の起動, 420
VM の停止, 420
エラー処理, 407
主な特徴, 405
クラスのインストール, 413
クラスの格納, 406
結果セットを返す, 418
セキュリティ管理, 420
説明, 405
配備, 1027
データベースプロパティ
db_get_property 関数, 542
テーブルアダプタ
Visual Studio, 92

と

同時実行性の値
SQL_CONCUR_LOCK, 394
SQL_CONCUR_READ_ONLY, 394
SQL_CONCUR_ROWVER, 394
SQL_CONCUR_VALUES, 394
動的 SELECT 文
DESCRIBE SELECT LIST 文, 505
動的 SQL
SQLDA, 506
説明, 503
動的カーソル
ODBC, 33
サンプル, 483
説明, 23
動的スクロールカーソル
トラブルシューティング, 10
登録
SQL Anywhere .NET データプロバイダ, 73
SQL Anywhere ASP.NET 接続文字列, 99
SQL Anywhere ASP.NET プロバイダ, 99
配備用の DLL, 1034
登録キー
サイレントインストール, 968
独立性レベル
ADO プログラミング, 350
DTC, 859
JDBC, 438
readonly-statement-snapshot, 39
SA_SQL_TXN_READONLY_STATEMENT_SNAPSHOT, 393
SA_SQL_TXN_SNAPSHOT, 393
SA_SQL_TXN_STATEMENT_SNAPSHOT, 393
SATransaction オブジェクトの設定, 64
snapshot, 39
SQL_TXN_READ_COMMITTED, 393
SQL_TXN_READ_UNCOMMITTED, 393
SQL_TXN_REPEATABLE_READ, 393
SQL_TXN_SERIALIZABLE, 393
statement-snapshot, 39
アプリケーション, 38
カーソル, 10
カーソルの感知性, 31
更新内容の消失, 30
ドライバ
jConnect JDBC ドライバ, 424
SQL Anywhere JDBC ドライバ, 424
SQL Anywhere ODBC ドライバの配備, 982
Windows での SQL Anywhere ODBC ドライバのリンク, 367
ドライバのレジストリエントリ
ODBC, 989
トラブルシューティング
カーソル位置, 10
データベース内の Java メソッド, 418
トランケーション

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01101

FETCH の場合, 497
FETCH 文, 497
インジケータ変数, 497
トランザクション
ADO, 350
ODBC, 376
ODBC トランザクションの独立性レベルの選択, 393
OLE DB, 350
アプリケーションの開発, 36
オートコミットの動作の制御, 37
オートコミットモード, 36
カーソル, 39
独立性レベル, 38
分散, 855
分散の使用, 859
トランザクション処理
SQL Anywhere .NET データプロバイダの使用, 64
取り消し処理
外部関数, 614
取り出し
ODBC, 395
トレース
.NET でのサポート, 74

ね

ネットワークサービス
OLE DB, 354
ネームスペース
Web サービス, 816

は

バイトコード
Java クラス, 406
バイナリデータ型
Embedded SQL, 490
配備
.NET データプロバイダ, 972
ADO.NET データプロバイダ, 972
CD-ROM でのデータベース, 1033
CLR 外部環境, 1035
Deployment ウィザード, 964
DLL の登録, 1034
Embedded SQL, 992
ESQL 外部環境, 1034
Interactive SQL, 1004
Interactive SQL (dbisqlc), 1026
Java 外部環境, 1035
jConnect, 994
JDBC クライアント, 994
Linux と UNIX でのコンソールユーティリティ (dbconsole), 1014
Linux と UNIX における Interactive SQL, 1014
Linux と UNIX における Sybase Central, 1014
Linux と UNIX における管理ツール, 1014
Mobile Link プラグイン, 1005
OData プロデューサ, 463
ODBC, 982
ODBC 外部環境, 1034
ODBC データソース, 988
OLE DB プロバイダ, 976
Open Client, 1003
Perl 外部環境, 1035
PHP 外部環境, 1036
PHP 拡張, 996
PHP クライアント, 996
SQL Anywhere, 959
SQL Anywhere .NET データプロバイダアプリケーション, 72
SQL Anywhere JDBC ドライバ, 994
SQL Anywhere プラグイン, 1005
SQL Remote, 1041
Sybase Central, 1004
Ultra Light プラグイン, 1005
UNIX の場合, 960
Windows の Interactive SQL, 1005
Windows の Sybase Central, 1005
Windows の管理ツール, 1005
Windows のコンソールユーティリティ (dbconsole), 1005
Windows への SQL Anywhere コンポーネントの配備, 964
アプリケーションとデータベース, 959
ウィザード, 964
外部環境, 1034
概要, 959
管理ツール, 1004
組み込みデータベース, 1038
クライアントアプリケーション, 972
グローバル配備に関する考慮事項, 1033
コンソールユーティリティ (dbconsole), 1004
サイレントインストール, 968
説明, 959
データベース, 1032

Copyright © 2013, SAP AG or an SAP affiliate company.

データベースサーバ, 1027
データベース内の Java, 1027
データベースのローカライズ, 1033
配備, 1034
パーソナルデータベースサーバ, 1038
ファイルロケーション, 960
読み込み専用データベース, 1033
レジストリの設定, 1029
配列フェッチ
ESQL, 519
説明, 519
バインドされたカラム
ODBC, 395
データアラインメント, 391
プロシージャの結果セット, 397
バインドされたパラメータ
ODBC, 384
SQLBindParameter 関数, 385
準備文, 3
データアラインメント, 391
プロシージャパラメータ, 397
バインドパラメータ
準備文, 3
バインド変数
説明, 503
パスワード
Interactive SQL, 1022
jConnect での暗号化, 428
Open Client での暗号化, 747
破損したデータベース
EnableFlush レジストリエントリ, 1029
バックアップ
DBTools の例, 866
ESQL 関数, 531
バックグラウンド処理
コールバック関数, 531
パッケージ
jConnect, 428
インストール, 415
バッチ挿入
JDBC, 445
バッファの長さ
ODBC, 387
パフォーマンス
JDBC, 442
JDBC ドライバ, 424
カーソル, 27
カーソルとプリフェッチされたロー, 28
準備文, 2
パラメータ
代入, 820
パラメータサイズの考慮事項
ODBC, 387
パラメータの受け渡し
外部関数, 612
パラメータのバインド
ODBC, 384
SQLBindParameter 関数, 385
ハンドル
ODBC の説明, 374
ODBC の割り付け, 375
バージョン番号
ファイル名, 962
バージョン列挙 [データベースツール API]
説明, 884
パーソナルサーバ
配備, 1038
パーミッション
外部関数を呼び出すプロシージャ, 601

ひ

ビットフィールド
使用, 865
表示可能な変更
カーソル, 17
標準
SQLJ, 405
非連鎖モード
実装, 38
制御, 37
トランザクション, 36

ふ

ファイル
配備ロケーション, 960
命名規則, 962
ファイル転送
コールバック, 549
ファイルデータソース
Windows, 988
ファイル名
.db ファイル拡張子, 963
.log ファイル拡張子, 963
SQL Anywhere, 963
規則, 962

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01103

言語, 962
バージョン番号, 962
ファイル命名規則
説明, 962
ファットカーソル
説明, 11
フェッチ
ESQL, 515
ODBC, 395
制限, 9
配列フェッチ, 519
ワイドフェッチ, 519
フェッチオペレーション
カーソル, 10
スクロール可能カーソル, 12
複数ロー, 11
複数の結果セット
DESCRIBE 文, 531
ODBC, 397
OEM の設定可能なオプション, 1022
複数のローのフェッチ
ESQL, 519
複数のローのプット
ESQL, 519
ブックマーク
ODBC カーソル, 397
説明, 16
プライマリキー
SACommand による取得, 52
SADataAdapter による取得, 60
プラグイン
配備, 1005
プラットフォーム
カーソル, 15
ブランク埋め込み
ESQL の文字列, 486
プリフェッチ
カーソル, 28
カーソルのパフォーマンス, 27
複数ローのフェッチ, 11
プリプロセッサ
実行, 471
説明, 469
プレースホルダ
動的 SQL, 503
プログラミングインタフェース
C API, 565
JDBC API, 423
ODBC, 365
Perl DBD::SQLAnywhere API, 649
Python データベース API, 659
Ruby API, 713
SQL Anywhere .NET API, 41
SQL Anywhere Embedded SQL, 469
SQL Anywhere OLE DB と ADO API, 343
SQL Anywhere PHP DBI, 667
Sybase Open Client API, 741
プログラム構造
Embedded SQL, 479
プロシージャ
CLR, 620
Embedded SQL, 528
ESQL の結果セット, 529
ODBC, 397
Web クライアント, 790
Web クライアントの要件, 791
プロシージャからの外部ライブラリの呼び出し
説明, 601
ブロックカーソル
ODBC, 16
説明, 11
プロトコル
Web サービスの設定, 751
Web サービスの有効化, 751
プロトタイプ
外部関数, 603
プロバイダ
.NET でサポートされている, 42
ASP.NET でサポートされている, 97
SQL Anywhere ASP.NET Web パーツパーソナル化プロバイダ, 97
SQL Anywhere ASP.NET プロファイルプロバイダ, 97
SQL Anywhere ASP.NET ヘルスモニタリングプロバイダ, 97
SQL Anywhere ASP.NET メンバーシッププロバイダ, 97
SQL Anywhere ASP.NET ロールプロバイダ, 97
プロパティ
db_get_property 関数, 542
文
INSERT, 2
分散トランザクション
3 層コンピューティング, 856
アーキテクチャ, 858
エンリスト, 857

Copyright © 2013, SAP AG or an SAP affiliate company.

制限, 859
説明, 855
リカバリ, 860
分散トランザクションコーディネーター
3 層コンピューティング, 858
分散トランザクション処理
SQL Anywhere .NET データプロバイダの使用, 64
プーリング
Web サービス, 763
プーリング
.NET データプロバイダを使用した接続, 46

へ

並列バックアップ
db_backup 関数, 533
ヘッダ
HTTP Web サービスでのアクセス, 766
SOAP Web サービス, 770
ヘッダファイル
Embedded SQL, 477
ODBC, 367
変換
データ型, 497
変更
Web サービス, 756
変数
HTTP Web サービスでのアクセス, 766
HTTP Web サービスへの指定, 803
SOAP Web サービス, 770

ほ

ホスト変数
SQLDA, 508
使用法, 494
説明, 489
宣言, 489
データ型, 490
バッチでサポートされない, 489
例, 766

ま

マクロ
_SQL_OS_WINDOWS, 479
マッピングファイル
SQL Server への配備, 972
マニュアル
Linux での配備, 1027
Mac OS X での配備, 1027
UNIX での配備, 1027
Windows での配備, 1026
マルチスレッドアプリケーション
Embedded SQL, 500
Embedded SQL における複数の SQLCA, 502
ODBC 機能サポート, 366
ODBC 接続, 379
UNIX, 369
データベース内の Java, 417
マルチロークエリ
カーソル, 517
マルチロー挿入
ESQL, 519

む

無効な文字列またはバッファ長
データアラインメント, 391

め

メソッドシグネチャ
Java, 634
メタデータサポート
jConnect に対するインストール, 429
メッセージ
コールバック, 548
サーバ, 548
メンバーシップ
結果セット, 16

も

文字セット
CHAR 文字セットの設定, 539
NCHAR 文字セットの設定, 539
Web サービス, 779
文字データ
Embedded SQL 内の長さ, 493
Embedded SQL 内の文字セット, 493
文字列
DT_NSTRING のブランク埋め込み, 486
DT_STRING のブランク埋め込み, 486
Embedded SQL, 471
データ型, 560
モデル
OData プロデューサ, 465
戻り値

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01105

Web クライアント, 806

ゆ

ユニークカーソル
説明, 15
ユーザ定義関数
CLR, 620
ユーザデータソース
Windows, 988
ユーザリスト列挙 [データベースツール API]
説明, 882
ユーティリティ
OData サーバ (dbosrv16.exe) の構文, 466
OData サーバ停止 (dbostop.exe) の構文, 467
SQL プリプロセッサ (sqlpp) の構文, 471
データベースユーティリティの配備, 1038
リターンコード, 956

よ

要求
アボート, 538
要求処理
Embedded SQL, 531
要求のキャンセル
Embedded SQL, 531
要件
Open Client アプリケーション, 742
読み込み専用
データベース配備, 1033
読み込み専用カーソル
説明, 15

ら

ライセンス
Web クライアント, 772
ライブラリ
dblib16.lib, 477
dblibtm.lib, 477
dbtlstm.lib, 862
dbtool16.lib, 862
Embedded SQL, 477
libdblib16_r.so, 477
libdblib16.so, 477
libdbtasks16_r.so, 477
libdbtasks16.so, 477
インポートライブラリの使い方, 862
ストアドプロシージャまたはファンクションからの外部ライブラリの呼び出し, 601
ライブラリ関数
Embedded SQL, 532

り

リカバリ
分散トランザクション, 860
リソースディスペンサー
3 層コンピューティング, 857
リソースマネージャ
3 層コンピューティング, 857
説明, 855
リダイレクション
URL の検証, 780
リターンコード
ODBC, 401
説明, 956
リターンタイプ
外部関数, 618
履歴列挙 [データベースツール API]
説明, 880
リンクサーバ
4 部分構成の構文, 354
Inprocess オプション, 355, 357
OLE DB, 354
openquery, 354
RPC オプション, 355, 357
RPC 出力オプション, 355, 357
セキュリティコンテキスト, 355, 357

れ

例
DBTools プログラム, 866
ODBC, 372
例外
Java, 407
SQL Anywhere .NET データプロバイダ, 65
レコードセット
ADO プログラミング, 349
レジストリ
ODBC, 989
Windows での管理ツールの配備, 1013
Wow6432Node, 1013
配備, 1029
レジストリの設定
SQL Anywhere ODBC ドライバ, 986

Copyright © 2013, SAP AG or an SAP affiliate company.

連鎖モード
実装, 38
制御, 37
トランザクション, 36

ろ

ロギング
Web サービスクライアントの情報, 823

わ

ワイド挿入
ESQL, 519
JDBC, 445
ワイドフェッチ
ESQL, 519
説明, 11, 519
ワイドプット
ESQL, 519
ワークテーブル
カーソルのパフォーマンス, 27

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01107