SYBASE iAnywhere

SQL Anywhere サーバ プログラミング

改訂 2007 年 3 月

版権と商標

目次

はじめに

このマニュアルの内容

このマニュアルでは、C、C++、Java プログラミング言語、Visual Studio .NET 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。Visual Basic や PowerBuilder などのツールのユーザは、それらのツールのプログラミング・インタフェースを使用できます。

対象読者

このマニュアルは SQL Anywhere の各インタフェースに直接アクセスするプログラムを作成するアプリケーション開発者を対象としています。

PowerBuilder や Visual Basic など、ODBC に加えて独自のデータベース・インタフェースを備えた開発ツールを使用している場合は、このマニュアルを読む必要はありません。

SQL Anywhere のマニュアル

このマニュアルは、SQL Anywhere のマニュアル・セットの一部です。この項では、マニュアル・セットに含まれる各マニュアルと使用法について説明します。

SQL Anywhere のマニュアル

SQL Anywhere の完全なマニュアルは、各マニュアルをまとめたオンライン形式とマニュアル別の PDF ファイルで提供されます。いずれの形式のマニュアルも、同じ情報が含まれ、次のマニュアルから構成されます。

- **『SQL Anywhere 10 - 紹介』** このマニュアルでは、データの管理および交換機能を提供する包括的なパッケージである SQL Anywhere 10 について説明します。SQL Anywhere を使用すると、サーバ環境、デスクトップ環境、モバイル環境、リモート・オフィス環境に適したデータベース・ベースのアプリケーションを迅速に開発できるようになります。
- **『SQL Anywhere 10 - 変更点とアップグレード』** このマニュアルでは、SQL Anywhere 10 とそれ以前のバージョンに含まれる新機能について説明します。
- **『SQL Anywhere サーバ - データベース管理』** このマニュアルでは、SQL Anywhere データベースの実行、管理、設定について説明します。管理ユーティリティとオプションのほか、データベース接続、データベース・サーバ、データベース・ファイル、バックアップ・プロシージャ、セキュリティ、高可用性、Replication Server を使用したレプリケーションについて説明します。
- **『SQL Anywhere サーバ - SQL の使用法』** このマニュアルでは、データベースの設計と作成の方法、データのインポート・エクスポート・変更の方法、データの検索方法、ストアド・プロシージャとトリガの構築方法について説明します。
- **『SQL Anywhere サーバ - SQL リファレンス』** このマニュアルは、SQL Anywhere で使用する SQL 言語の完全なリファレンスです。また、SQL Anywhere のシステム・ビューとシステム・プロシージャについても説明しています。
- **『SQL Anywhere サーバ - プログラミング』** このマニュアルでは、C、C++、Java プログラミング言語、Visual Studio .NET 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。Visual Basic や PowerBuilder などのツールのユーザは、それらのツールのプログラミング・インタフェースを使用できます。
- **『SQL Anywhere 10 - エラー・メッセージ』** このマニュアルでは、SQL Anywhere エラー・メッセージの完全なリストを、その診断情報とともに説明します。
- **『Mobile Link - クイック・スタート』** このマニュアルでは、セッションベースのリレーショナル・データベース同期システムである Mobile Link について説明します。Mobile Link テクノロジは、双方向レプリケーションを可能にし、モバイル・コンピューティング環境に非常に適しています。
- **『Mobile Link - サーバ管理』** このマニュアルでは、Mobile Link アプリケーションを設定して管理する方法について説明します。

- **『Mobile Link - クライアント管理』** このマニュアルでは、Mobile Link クライアントを設定、構成、同期する方法について説明します。Mobile Link クライアントには、SQL Anywhere または Ultra Light のいずれかのデータベースを使用できます。
- **『Mobile Link - サーバ起動同期』** このマニュアルでは、Mobile Link のサーバによって開始される同期について説明します。サーバによって開始される同期とは、統合データベースから同期またはその他のリモート・アクションの開始を可能にする Mobile Link の機能です。
- **『QAnywhere』** このマニュアルでは QAnywhere について説明します。QAnywhere は、従来のデスクトップ・クライアントやラップトップ・クライアント用のメッセージング・プラットフォームであるほか、モバイル・クライアントや無線クライアント用のメッセージング・プラットフォームでもあります。
- **『SQL Remote』** このマニュアルでは、モバイル・コンピューティング用の SQL Remote データ・レプリケーション・システムについて説明します。このシステムによって、SQL Anywhere の統合データベースと複数の SQL Anywhere リモート・データベースの間で、電子メールやファイル転送などの間接的リンクを使用したデータ共有が可能になります。
- **『SQL Anywhere 10 - コンテキスト別ヘルプ』** このマニュアルには、[接続] ダイアログ、クエリ・エディタ、Mobile Link モニタ、SQL Anywhere コンソール・ユーティリティ、インデックス・コンサルタント、Interactive SQL のコンテキスト別のヘルプが収録されています。
- **『Ultra Light - データベース管理とリファレンス』** このマニュアルでは、小型デバイス用 Ultra Light データベース・システムの概要を説明します。
- **『Ultra Light - AppForge プログラミング』** このマニュアルでは、Ultra Light for AppForge について説明します。Ultra Light for AppForge を使用すると、Palm OS、Symbian OS、または Windows CE を搭載しているハンドヘルド、モバイル、または埋め込みデバイスに対してデータベース・アプリケーションを開発、配備できます。
- **『Ultra Light - .NET プログラミング』** このマニュアルでは、Ultra Light.NET について説明します。Ultra Light.NET を使用すると、PC、ハンドヘルド、モバイル、埋め込みデバイスのデータベース・アプリケーションを開発し、これらのデバイスに配備できます。
- **『Ultra Light - M-Business Anywhere プログラミング』** このマニュアルは、Ultra Light for M-Business Anywhere について説明します。Ultra Light for M-Business Anywhere を使用すると、Palm OS、Windows CE、または Windows XP を搭載しているハンドヘルド、モバイル、または埋め込みデバイスに対して Web ベースのデータベース・アプリケーションを開発、配備できます。
- **『Ultra Light - C/C++ プログラミング』** このマニュアルでは、Ultra Light C および Ultra Light C++ のプログラミング・インタフェースについて説明します。Ultra Light を使用すると、ハンドヘルド、モバイル、埋め込みデバイスに対してデータベース・アプリケーションを開発、配備できます。

マニュアルの形式

SQL Anywhere のマニュアルは、次の形式で提供されています。

- **オンライン・マニュアル** オンライン・マニュアルには、SQL Anywhere の完全なマニュアルがあり、SQL Anywhere ツールに関する印刷マニュアルとコンテキスト別のヘルプの両方が含

まれています。オンライン・マニュアルは、製品のメンテナンス・リリースごとに更新されます。これは、最新の情報を含む最も完全なマニュアルです。

Windows オペレーティング・システムでオンライン・マニュアルにアクセスするには、[スタート] − [プログラム] − [SQL Anywhere 10] − [オンライン・マニュアル] を選択します。オンライン・マニュアルをナビゲートするには、左ウィンドウ枠で HTML ヘルプの目次、索引、検索機能を使用し、右ウィンドウ枠でリンク情報とメニューを使用します。

UNIX オペレーティング・システムでオンライン・マニュアルにアクセスするには、SQL Anywhere のインストール・ディレクトリまたはインストール CD に保存されている HTML マニュアルを参照してください。

- **PDF ファイル** SQL Anywhere の完全なマニュアル・セットは、Adobe Reader で表示できる Adobe Portable Document Format (pdf) 形式のファイルとして提供されています。

  Windows では、PDF 形式のマニュアルはオンライン・マニュアルの各ページ上部にある PDF のリンクから、または Windows の [スタート] メニュー ([スタート] − [プログラム] − [SQL Anywhere 10] − [オンライン・マニュアル - PDF フォーマット]) からアクセスできます。

  UNIX では、PDF 形式のマニュアルはインストール CD にあります。

表記の規則

この項では、このマニュアルで使用されている書体およびグラフィック表現の規則について説明します。

SQL 構文の表記規則

SQL 構文の表記には、次の規則が適用されます。

- **キーワード**　SQL キーワードはすべて次の例に示す ALTER TABLE のように大文字で表記します。

  ALTER TABLE [*owner.*]*table-name*

- **プレースホルダ**　適切な識別子または式で置き換えられる項目は、次の例に示す *owner* や *table-name* のように表記します。

  ALTER TABLE [*owner.*]*table-name*

- **繰り返し項目**　繰り返し項目のリストは、次の例に示す *column-constraint* のように、リストの要素の後ろに省略記号 (ピリオド 3 つ …) を付けて表します。

  ADD *column-definition* [*column-constraint*, …]

  複数の要素を指定できます。複数の要素を指定する場合は、各要素間をカンマで区切る必要があります。

- **オプション部分**　文のオプション部分は角カッコで囲みます。

  RELEASE SAVEPOINT [*savepoint-name*]

  この例では、角カッコで囲まれた *savepoint-name* がオプション部分です。角カッコは入力しないでください。

- **オプション**　項目リストから 1 つだけ選択する場合や、何も選択しなくてもよい場合は、項目間を縦線で区切り、リスト全体を角カッコで囲みます。

  [**ASC** | **DESC**]

  この例では、ASC と DESC のどちらか 1 つを選択しても、選択しなくてもかまいません。角カッコは入力しないでください。

- **選択肢**　オプションの中の 1 つを必ず選択しなければならない場合は、選択肢を中カッコで囲み、縦棒で区切ります。

  [**QUOTES** { **ON** | **OFF** }]

  QUOTES オプションを使用する場合は、ON または OFF のどちらかを選択する必要があります。角カッコと中カッコは入力しないでください。

オペレーティング・システムの表記規則

- **Windows**　デスクトップおよびラップトップ・コンピュータ用の Microsoft Windows オペレーティング・システムのファミリのことです。Windows ファミリには Windows Vista や Windows XP も含まれます。
- **Windows CE**　Microsoft Windows CE モジュラ・オペレーティング・システムに基づいて構築されたプラットフォームです。Windows Mobile や Windows Embedded CE などのプラットフォームが含まれます。

  Windows Mobile は Windows CE 上に構築されています。これにより、Windows のユーザ・インタフェースや、Word や Excel といったアプリケーションの小規模バージョンなどの追加機能が実現されています。Windows Mobile は、モバイル・デバイスで最も広く使用されています。

  SQL Anywhere の制限事項や相違点は、基盤となっているオペレーティング・システム (Windows CE) に由来しており、使用しているプラットフォーム (Windows Mobile など) に依存していることはほとんどありません。
- **UNIX**　特に記述がないかぎり、UNIX は Linux プラットフォームと UNIX プラットフォームの両方のことです。

ファイルの命名規則

マニュアルでは、パス名やファイル名などのオペレーティング・システムに依存するタスクと機能を表すときは、通常 Windows の表記規則が使用されます。ほとんどの場合、他のオペレーティング・システムで使用される構文に簡単に変換できます。

- **ディレクトリ名とパス名**　マニュアルでは、ドライブを示すコロンや、ディレクトリの区切り文字として使用する円記号など、Windows の表記規則を使用して、ディレクトリ・パスのリストを示します。次に例を示します。

  ```
  MobiLink¥redirector
  ```

  UNIX、Linux、Mac OS X では、代わりにスラッシュを使用してください。次に例を示します。

  ```
  MobiLink/redirector
  ```

  SQL Anywhere がマルチプラットフォーム環境で使用されている場合、プラットフォーム間でのパス名の違いに注意する必要があります。
- **実行ファイル**　マニュアルでは、実行ファイルの名前は、Windows の表記規則が使用され、拡張子 *.exe* が付きます。UNIX、Linux、Mac OS X では、実行ファイルの名前には拡張子は付きません。NetWare では、実行ファイルの名前には、拡張子 *.nlm* が付きます。

  たとえば、Windows では、ネットワーク・データベース・サーバは *dbsrv10.exe* です。UNIX、Linux、Mac OS X では、*dbsrv10* になります。NetWare では、*dbsrv10.nlm* になります。
- **install-dir**　インストール・プロセスでは、SQL Anywhere をインストールするロケーションを選択できます。マニュアルでは、このロケーションは *install-dir* という表記で示されます。

インストールが完了すると、環境変数 SQLANY10 によって SQL Anywhere コンポーネントがあるインストール・ディレクトリのロケーション (*install-dir*) が指定されます。SQLANYSH10 は、SQL Anywhere が他の Sybase アプリケーションと共有しているコンポーネントがあるディレクトリのロケーションを指定します。

オペレーティング・システム別の *install-dir* のデフォルト・ロケーションの詳細については、「SQLANY10 環境変数」『SQL Anywhere サーバ - データベース管理』を参照してください。

- **samples-dir** インストール・プロセスでは、SQL Anywhere に含まれるサンプルをインストールするロケーションを選択できます。マニュアルでは、このロケーションは *samples-dir* という表記で示されます。

  インストールが完了すると、環境変数 SQLANYSAMP10 によってサンプルがあるディレクトリのロケーション (*samples-dir*) が指定されます。Windows の [スタート] メニューから、[プログラム] – [SQL Anywhere 10] – [サンプル・アプリケーションおよびプロジェクト] を選択すると、このディレクトリで [Windows エクスプローラ] ウィンドウが表示されます。

  オペレーティング・システム別の *samples-dir* のデフォルト・ロケーションの詳細については、「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。

- **環境変数** マニュアルでは、環境変数設定が引用されます。Windows では、環境変数を参照するのに、構文 *%envvar%* が使用されます。UNIX、Linux、Mac OS X では、環境変数を参照するのに、構文 *$envvar* または *${envvar}* が使用されます。

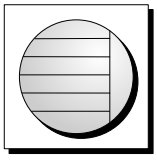
  UNIX、Linux、Mac OS X 環境変数は、*.cshrc* や *.tcshrc* などのシェルとログイン・スタートアップ・ファイルに格納されます。

グラフィック・アイコン

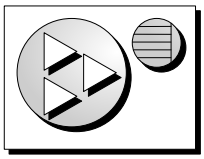
このマニュアルでは、次のアイコンを使用します。

- クライアント・アプリケーション

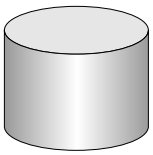
- SQL Anywhere などのデータベース・サーバ



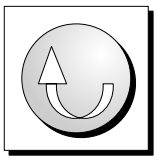
- Ultra Light アプリケーション



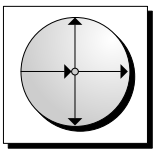
- データベース。高度な図では、データベースとデータベースを管理するデータ・サーバの両方をこのアイコンで表します。



- レプリケーションまたは同期のミドルウェア。ソフトウェアのこれらの部分は、データベース間のデータ共有を支援します。たとえば、Mobile Link サーバ、SQL Remote Message Agent などが挙げられます。



- Sybase Replication Server



- プログラミング・インタフェース

詳細情報の検索／フィードバックの提供

詳細情報の検索

詳しい情報やリソース (コード交換など) については、iAnywhere Developer Network (http://www.ianywhere.com/developer/) を参照してください。

ご質問がある場合や支援が必要な場合は、次に示す Sybase iAnywhere ニュースグループのいずれかにメッセージをお寄せください。

ニュースグループにメッセージをお送りいただく際には、ご使用の SQL Anywhere バージョンのビルド番号を明記し、現在発生している問題について詳しくお知らせくださいますようお願いいたします。バージョン情報は、コマンド・プロンプトで **dbeng10 -v** と入力して確認できます。

ニュースグループは、ニュース・サーバ *forums.sybase.com* にあります (ニュースグループにおけるサービスは英語でのみの提供となります)。以下のニュースグループがあります。

- sybase.public.sqlanywhere.general
- sybase.public.sqlanywhere.linux
- sybase.public.sqlanywhere.mobilink
- sybase.public.sqlanywhere.product_futures_discussion
- sybase.public.sqlanywhere.replication
- sybase.public.sqlanywhere.ultralite
- ianywhere.public.sqlanywhere.qanywhere

ニュースグループに関するお断り
iAnywhere Solutions は、ニュースグループ上に解決策、情報、または意見を提供する義務を負うものではありません。また、システム・オペレータ以外のスタッフにこのサービスを監視させて、操作状況や可用性を保証する義務もありません。
iAnywhere のテクニカル・アドバイザとその他のスタッフは、時間のある場合にかぎりニュースグループでの支援を行います。こうした支援は基本的にボランティアで行われるため、解決策や情報を定期的に提供できるとはかぎりません。支援できるかどうかは、スタッフの仕事量に左右されます。

フィードバック

このマニュアルに関するご意見、ご提案、フィードバックをお寄せください。

マニュアルに関するご意見、ご提案は、SQL Anywhere ドキュメンテーション・チームの iasdoc@ianywhere.com 宛てに電子メールでお寄せください。このアドレスに送信された電子メールに返信はいたしませんが、お寄せいただいたご意見、ご提案は必ず読ませていただきます。

マニュアルまたはソフトウェアについてのフィードバックは、上記のニュースグループを通してお寄せいただいてもかまいません。

パート I. SQL Anywhere でのプログラミングの概要

パート I では、SQL Anywhere でのプログラミングについて説明します。

第 1 章

SQL Anywhere データ・アクセス・プログラミング・インタフェース

目次

SQL Anywhere .NET API

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。

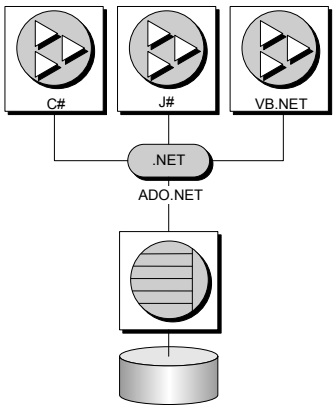
SQL Anywhere .NET データ・プロバイダは、iAnywhere.Data.SQLAnywhere ネームスペースを実装しており、.NET でサポートされている任意の言語 (C# や Visual Basic .NET など) でプログラムを作成したり、SQL Anywhere データベースからデータにアクセスしたりできます。

.NET データ・アクセスの概要については、http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/daag.asp 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 フレームワークに完全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」 113 ページ
- 「SQL Anywhere .NET 2.0 API リファレンス」 163 ページ
- 「チュートリアル：SQL Anywhere .NET データ・プロバイダの使用」 151 ページ

SQL Anywhere OLE DB と ADO API

SQL Anywhere には、OLE DB と ADO のプログラマ向けの 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。詳細については、「SQL Anywhere .NET API」 4 ページを参照してください。

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、「SQL Anywhere OLE DB と ADO API」 443 ページを参照してください。

ODBC API

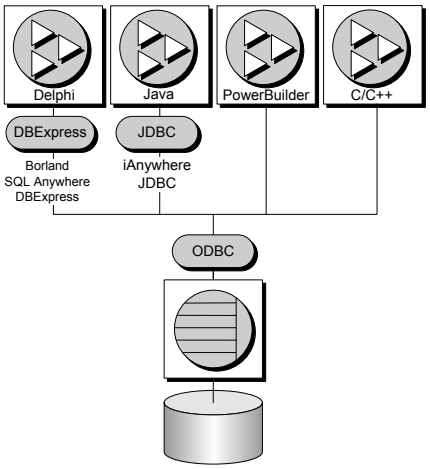
ODBC (Open Database Connectivity) は、Microsoft が開発した標準 CLI (コール・レベル・インタフェース) です。SQL Access Group CLI 仕様に基づいています。ODBC アプリケーションは、ODBC ドライバを提供するあらゆるデータ・ソースに使用できます。ODBC ドライバを持つ他のデータ・ソースにアプリケーションを移植できるようにしたい場合は、プログラミング・インタフェースとして ODBC を使用することをおすすめします。

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。ODBC は、Windows CE を除く Windows オペレーティング・システム上で DLL として提供されます。UNIX 用にはライブラリとして提供されます。

ODBC の基本のマニュアルは、Microsoft ODBC Software Development Kit です。

ODBC アプリケーション

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、ODBC API を使用して SQL Anywhere データベース・サーバにアクセスすることでさまざまなアプリケーションを開発できます。



たとえば、SQLAnywhere に付属のアプリケーションのうち、InfoMaker と PowerDesigner Physical Data Model は ODBC を使用してデータベースに接続します。

参照

- 「SQL Anywhere ODBC API」 459 ページ

JDBC API

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。Sun Microsystems が開発したこの JDBC を使用すると、Java プログラマは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。

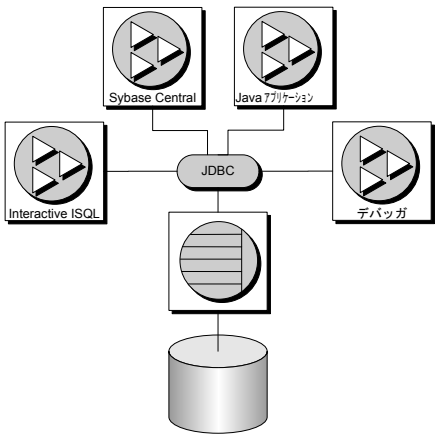
SQL Anywhere には、Sybase jConnect という pure Java の JDBC ドライバが用意されています。また、タイプ 2 ドライバである iAnywhere JDBC ドライバも用意されています。いずれも「SQL Anywhere JDBC API」 491 ページで説明されています。

JDBC ドライバの選択については、「JDBC ドライバの選択」 492 ページを参照してください。

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。

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。

参照

- 「JDBC ドライバの選択」 492 ページ
- 「SQL Anywhere JDBC API」 491 ページ

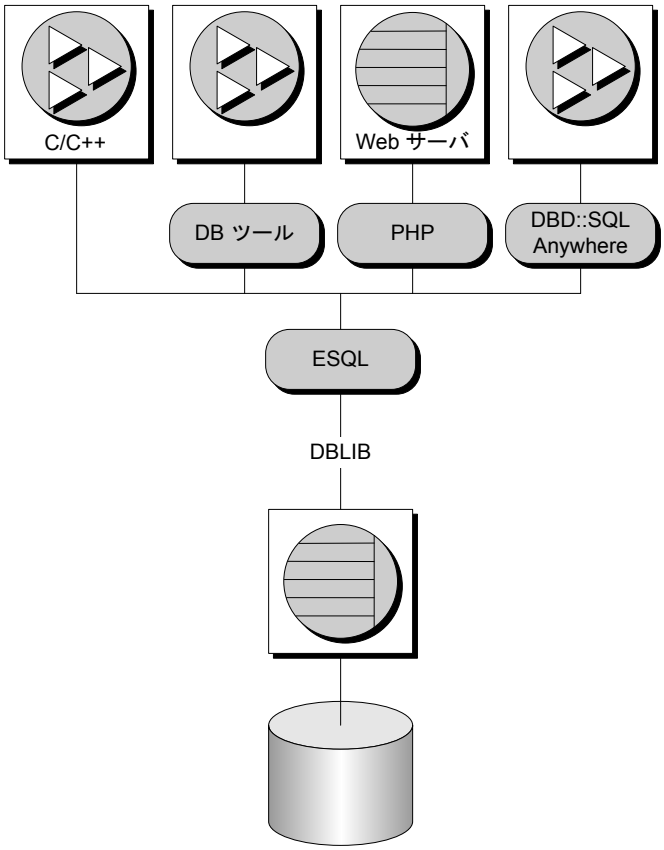
SQL Anywhere Embedded SQL

Embedded SQL は、SQL コマンドを C または C++ ソース・ファイルに直接組み込むシステムです。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。



参照

- 「SQL Anywhere Embedded SQL」 519 ページ

Sybase Open Client API

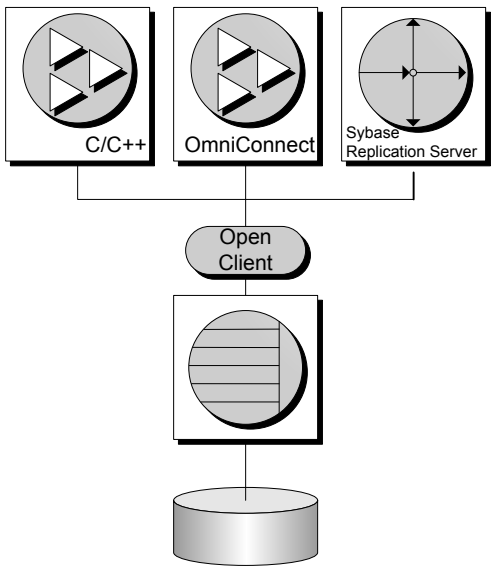
Sybase Open Client は、カスタマ・アプリケーション、サードパーティ製品、その他の Sybase 製品に、SQL Anywhere およびその他の Open Server と通信するために必要なインタフェースを提供します。

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 Replication Server など、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 や Replication Server 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』

Perl DBD::SQLAnywhere API

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 スクリプトが Sybase 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。

参照

- 「SQL Anywhere Perl DBD::SQLAnywhere API」 609 ページ

SQL Anywhere PHP API

PHP には一般的なデータベースから情報を取得する機能があります。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モジュールが用意されています。PHP 言語を使用すると、SQL Anywhere データベースから情報を取得し、独自の Web サイトで動的な Web コンテンツを提供できます。

SQL Anywhere PHP モジュールを使用すると、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。このモジュールは、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。

参照

- 「SQL Anywhere PHP API」 619 ページ

SQL Anywhere Web サービス

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 などのデータ・アクセス API の代わりになる方法を提供します。Web サービスへは、各種の言語で記述され、各種のプラットフォームで実行されるクライアント・アプリケーションからアクセスできます。Perl や Python などの一般的なスクリプト言語でも Web サービスにアクセスできます。

参照

- 「SQL Anywhere Web サービス」 661 ページ

第 2 章

SQL Anywhere Explorer

目次

SQL Anywhere Explorer の概要

SQL Anywhere Explorer は、Visual Studio .NET から SQL Anywhere と Ultra Light に接続できるコンポーネントです。

Ultra Light 用 SQL Anywhere Explorer の使用については、「Ultra Light 用 SQL Anywhere Explorer」『Ultra Light - .NET プログラミング』を参照してください。

SQL Anywhere Explorer の使用

Visual Studio .NET 2003 と 2005 では、SQL Anywhere Explorer を使用して SQL Anywhere データベースへの接続を作成できます。データベースに接続すると、次の操作を実行できます。

- データベース内のテーブル、ビュー、プロシージャを表示する。
- テーブルやビューに保存されたデータを表示する。
- SQL Anywhere データベースへの接続を開くプログラムやデータの取り出しと操作を行うプログラムを設計する。
- データベース・オブジェクトを C# や Visual Basic のコードやフォームにドラッグ・アンド・ドロップすることで、選択されたオブジェクトを参照するコードを IDE で自動的に生成する。

[ツール] メニューから対応するコマンドを選択することで、Sybase Central と Interactive SQL を Visual Studio .NET から開くこともできます。

インストールの注意

SQL Anywhere ソフトウェアをインストールした Windows コンピュータに Visual Studio がインストール済みの場合、インストール・プロセスは Visual Studio の存在を検出し、必要な統合手順を実行します。Visual Studio を SQL Anywhere の後にインストールする場合、または Visual Studio の新しいバージョンをインストールする場合、SQL Anywhere を Visual Studio と統合するプロセスを次の手順に従って手動で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。
- Visual Studio .NET 2003 の場合は、*install-dir¥Assembly¥v1¥setupVSPackage.exe* を実行します。

  Visual Studio 2005 の場合は、*install-dir¥Assembly¥v2¥setupVSPackage.exe* を実行します。

Visual Studio .NET でのデータベース接続の使用

SQL Anywhere Explorer を使用して、データ接続ノードで SQL Anywhere データベース接続を表示します。テーブルやビュー内のデータを表示するには、データ接続を作成してください。

SQL Anywhere Explorer でデータベース・テーブル、ビュー、ストアド・プロシージャ、関数を表示し、個々のテーブルを展開してカラムを表示できます。SQL Anywhere Explorer ウィンドウで選択されたオブジェクトのプロパティは、Visual Studio の [プロパティ] ウィンドウ枠に表示されます。

♦ Visual Studio .NET で SQL Anywhere データベース接続を追加するには、次の手順に従います。

1. [表示] – [SQL Anywhere Explorer] を選択して、SQL Anywhere Explorer を開きます。

2. SQL Anywhere Explorer ウィンドウで、[データ接続] を右クリックし、[接続の追加] を選択します。

   [接続の追加] ダイアログが表示されます。

3. [SQL Anywhere] を選択し、[OK] をクリックします。

   [接続プロパティ] ダイアログが表示されます。

4. 適切な値を入力して、データベースに接続します。

5. [OK] をクリックします。

   データベースへの接続が確立され、接続が [データ接続] リストに追加されます。

♦ Visual Studio .NET から SQL Anywhere データベース接続を削除するには、次の手順に従います。

1. [表示] − [SQL Anywhere Explorer] を選択して、SQL Anywhere Explorer を開きます。

2. SQL Anywhere Explorer ウィンドウで、削除するデータベース接続を右クリックし、[削除] を選択します。

   SQL Anywhere Explorer ウィンドウから接続が削除されます。

SQL Anywhere Explorer の設定

Visual Studio の [オプション] ダイアログには、SQL Anywhere Explorer を設定するのに使用できる設定が含まれています。

♦ SQL Anywhere Explorer オプションを使用するには、次の手順に従います。

1. Visual Studio の [ツール] メニューから [オプション] を選択します。

   [オプション] ダイアログが表示されます。

2. [オプション] ダイアログの左ウィンドウ枠で、[SQL Anywhere] を展開します。

3. [一般] をクリックして、必要に応じて SQL Anywhere Explorer の一般オプションを設定します。

   [出力ウィンドウに送信するクエリ結果を制限する] [出力] ウィンドウに表示するローの数を指定します。デフォルト値は 500 です。

   [オブジェクトのソート] SQL Anywhere Explorer ウィンドウで、オブジェクト名またはオブジェクトの所有者名順にオブジェクトをソートするときに選択します。

   [テーブルまたはビューをデザイナにドラッグ・アンド・ドロップするときに UI コードを生成する] Windows Forms Designer にドラッグ・アンド・ドロップするテーブルまたはビューのコードを生成します。

[データ・アダプタ用に INSERT、UPDATE、DELETE コマンドを生成する] C# または Visual Basic ドキュメントにテーブルまたはビューをドラッグ・アンド・ドロップするときに、データ・アダプタ用の INSERT、UPDATE、DELETE コマンドを生成します。

[データ・アダプタ用にテーブル・マッピングを生成する] C# または Visual Basic ドキュメントにテーブルをドラッグ・アンド・ドロップするときに、データ・アダプタ用のテーブル・マッピングを生成します。

SQL Anywhere Explorer を使用したデータベース・オブジェクトの追加

Visual Studio .NET では、SQL Anywhere Explorer から Visual Studio .NET デザイナに特定のデータベース・オブジェクトをドラッグ・アンド・ドロップすると、選択されたオブジェクトを参照する新しいコンポーネントが IDE によって自動的に作成されます。ドラッグ・アンド・ドロップ操作の設定を指定するには、Visual Studio .NET の [ツール] – [オプション] を選択します。

たとえば、SQL Anywhere Explorer から Windows フォームにストアド・プロシージャをドラッグすると、ストアド・プロシージャを呼び出すよう設定された Command オブジェクトが IDE によって自動的に作成されます。

次の表に、SQL Anywhere Explorer からドラッグできるオブジェクトと、Visual Studio .NET Forms Designer またはコード・エディタにドロップしたときに作成されるコンポーネントを示します。

項目	結果
データ接続	データ接続を作成します。
テーブル	アダプタを作成します。
ビュー	アダプタを作成します。
ストアド・プロシージャまたは関数	コマンドを作成します。

♦ SQL Anywhere Explorer を使用して、新しいデータ・コンポーネントを作成するには、次の手順に従います。

1. データ・コンポーネントを追加するフォームまたはクラスを開きます。
2. SQL Anywhere Explorer で、使用するオブジェクトを選択します。
3. SQL Anywhere Explorer から Forms Designer またはコード・エディタにオブジェクトをドラッグします。

SQL Anywhere Explorer を使用したテーブルの操作

SQL Anywhere Explorer では、Visual Studio .NET から SQL Anywhere データベースのテーブルやビューのプロパティとデータを表示できます。

♦ **Visual Studio .NET でテーブルまたはビューのデータを表示するには、次の手順に従います。**

1. SQL Anywhere Explorer を使用して、SQL Anywhere データベースに接続します。
2. [SQL Anywhere Explorer] ダイアログで、データベースを展開するか、表示するオブジェクトによって [テーブル] または [ビュー] を展開します。
3. テーブルまたはビューを右クリックして、[データの取得] を選択します。

   選択したテーブルまたはビューのデータが Visual Studio .NET の [出力] ウィンドウに表示されます。

SQL Anywhere Explorer を使用したプロシージャと関数の操作

ストアド・プロシージャを変更した場合は、SQL Anywhere Explorer でプロシージャを再表示して、カラムやパラメータへの最新の変更内容を取得できます。

♦ **Visual Studio .NET でプロシージャを再表示するには、次の手順に従います。**

1. SQL Anywhere データベースに接続します。
2. プロシージャを右クリックして、[再表示] を選択します。

   データベース内のプロシージャが変更されている場合は、パラメータとカラムが更新されます。

第 3 章

アプリケーションでの SQL の使用

目次

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();
```

「SQL Anywhere .NET データ・プロバイダ」 113 ページを参照してください。

- **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 );
```

「SQL Anywhere ODBC API」 459 ページを参照してください。

- **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" );
```

「SQL Anywhere JDBC API」 491 ページを参照してください。

- **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';
```

「SQL Anywhere Embedded SQL」 519 ページを参照してください。

- **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);
```

「Sybase Open Client API」 651 ページを参照してください。

- **アプリケーション開発ツール**　Sybase Enterprise Application Studio ファミリのメンバのようなアプリケーション開発ツールは独自の SQL オブジェクトを提供し、ODBC (PowerBuilder) または JDBC (Power J) を見えない所で使用します。

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。

ストアド・プロシージャとトリガの詳細については、「プロシージャ、トリガ、バッチの使用」『SQL Anywhere サーバ - SQL の使用法』を参照してください。

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。この章では JDBC についても一部説明します。JDBC の使用の詳細については、「SQL Anywhere JDBC API」 491 ページを参照してください。

文の準備

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。これは文の「**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・プランを生成する。
- 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、文全体を何度も準備しなくても、プレースホルダに値を割り当てるだけで済みます。

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。

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。
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。

カーソルについては、「カーソルの概要」 31 ページを参照してください。

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。

準備文の使用方法

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、次の手順に従います (一般)。**

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。Java のガーベジ・コレクション・メカニズムが処理するので、この手順は JDBC では必要ありません。

♦ **準備文を使用するには、次の手順に従います (ADO.NET)。**

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。

3. Prepare メソッドを使って文を準備します。

   ```
   cmd.Prepare();
   ```

4. 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、*samples-dir¥SQLAnywhere¥ADO.NET ¥SimpleWin32* にあるソース・コードを参照してください。

♦ **準備文を使用するには、次の手順に従います (ODBC)。**

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、*samples-dir¥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、「準備文の実行」 477 ページと ODBC SDK のマニュアルを参照してください。

♦ **準備文を使用するには、次の手順に従います (JDBC)。**

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 *samples-dir¥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」 511 ページを参照してください。

♦ **準備文を使用するには、次の手順に従います (Embedded SQL)。**

1. EXEC SQL PREPARE コマンド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 コマンドを使用して文を実行します。
4. EXEC SQL DROP コマンドを使用して、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。

♦ **準備文を使用するには、次の手順に従います (Open Client)。**

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 の使用」 656 ページを参照してください。

カーソルの概要

アプリケーションでクエリを実行すると、結果セットは多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タイプのリストについては、「カーソルの可用性」 41 ページを参照してください。

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」 41 ページを参照してください。

カーソルとは？

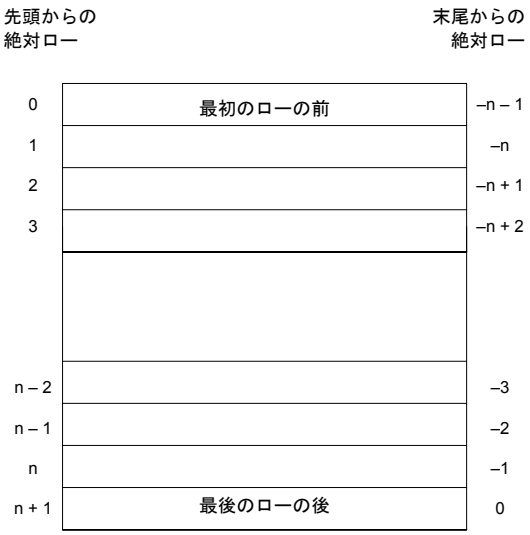
「**カーソル**」とは、結果セットに関連付けられた名前です。結果セットは、SELECT 文かストアド・プロシージャ呼び出しによって取得されます。

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



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

データベース・アプリケーションでは、カーソルを使用する必要はありませんが、カーソルには多くの利点があります。たとえば、カーソルを使用しない場合は、処理や表示のために結果セット全体をクライアントに送信する必要があることからも、カーソルの利点は明らかです。

- **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に格納するには、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。
- **応答時間**　カーソルは、結果セット全体をアセンブルする前に、最初の数行分のローを表示することができます。カーソルを使わない場合は、アプリケーションがどのローを表示するにも、まず結果セット全体が送信されている必要があります。
- **同時性の制御**　アプリケーションでデータを更新する場合にカーソルを使用しない場合、変更を適用するために別の SQL 文をデータベース・サーバに送信します。この方法では、クライアントがクエリを実行した後で結果セットが変更された場合には、同時性の問題が生じる可能性があります。その結果、更新情報が失われる可能性もあります。

  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、加えた変更には適切な同時性制約が課されます。

カーソルを使用した操作

この項では、カーソルを使ったさまざまな種類の操作について説明します。

カーソルの使い方

Embedded SQL でのカーソルの使用法は他のインタフェースとは異なります。

♦ **カーソルを使用するには、次の手順に従います (ADO.NET、ODBC、JDBC、Open Client)。**

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 コマン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. 文を開放します。

   準備した文を使った場合は、それを開放してメモリを再利用します。

♦ **カーソルを使用するには、次の手順に従います (Embedded SQL)。**

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。

   文の準備方法については、「文の準備」 28 ページを参照してください。

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。

   詳細については、「DECLARE CURSOR 文 [ESQL] [SP]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

3. カーソルを開きます。

   詳細については、「OPEN 文 [ESQL] [SP]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

   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。

   詳細については、「FETCH 文 [ESQL] [SP]」『SQL Anywhere サーバ - SQL リファレンス』と「データのフェッチ」 563 ページを参照してください。

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。

   詳細については、「CLOSE 文 [ESQL] [SP]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

   詳細については、「DROP STATEMENT 文 [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Embedded SQL でカーソルを使用する方法の詳細については、「データのフェッチ」 563 ページを参照してください。

ローのプリフェッチ

場合によっては、インタフェース・ライブラリがパフォーマンスの最適化を (結果のプリフェッチのように) 見えない所で実行するので、クライアント・アプリケーションの手順はソフトウェアの操作と完全には一致していません。

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。このカーソルが最初のローの前、または最後のローの後にある場合、「**カーソルの現在のローがありません。**」というエラーが返されます。

カーソル位置に関する問題
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、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、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ときにこうしたことが起こります (「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』を参照)。
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。

開くときのカーソルの設定

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。

  詳細については、「isolation_level オプション [互換性]」 『SQL Anywhere サーバ - データベース管理』を参照してください。

- **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。

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。

♦ **結果セットのローをループするには、次の手順に従います。**

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。

手順 2 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.NextResult メソッドを使用します。「NextResult メソッド」 323 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 でカーソルを使用する詳細については、「結果セットの処理」 479 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 ResultSet オブジェクトを使用する方法の詳細については、「結果セットを返す」 513 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、「ESQL でのカーソルの使用」 564 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カーソルを使用する方法の詳細については、「カーソルの使い方」 656 ページを参照してください。

複数ローのフェッチ

この項では、一度に複数のローをフェッチしてパフォーマンスを向上させる方法について説明します。

複数ローのフェッチと、次の項で説明するローのプリフェッチと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。

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によるフェッチ

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。

カーソルによるローの変更

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、挿入動作の場合は、これを「**入力**」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オートインクリメント・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修正が許可されますが、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。

   詳細については、「FROM 句」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
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。

更新可能な文の概要

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 READ ONLY になります。SQL の SELECT 文で FOR XML を指定すると、カーソルは READ ONLY 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 の指定はカーソルが更新可能かどうかに影響しますが、SQL Anywhere では FOR UPDATE 構文自体は同時制御には影響しません。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制御は、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。

詳細については、「DECLARE 文」『SQL Anywhere サーバ - SQL リファレンス』と「FOR 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

カーソル操作のキャンセル

インタフェース機能で要求をキャンセルできます。Interactive SQL から、ツールバーの [SQL 文の中断] をクリックして、(または [SQL] – [停止] を選択して) 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・タイプの選択

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。

SQL Anywhere のカーソルについては、「SQL Anywhere のカーソル」 43 ページを参照してください。

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。

  詳細については、「結果セットの処理」 479 ページを参照してください。
- 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：
  - iAnywhere JDBC ドライバでは、JDBC 2.0 と JDBC 3.0 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 5.5 と 6.0.5 では、iAnywhere 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。

    JDBC カーソルの宣言の詳細については、「SQL Anywhere のカーソルの要求」 58 ページを参照してください。
- 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。

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。

どのカーソル・タイプも、複数の特性によって定義されています。

- **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、その特性をできるだけ一致させます。

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。これはつまり、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。このように、基本となるデータの変更に関するカーソルのさまざまな動作を、カーソルの「**感知性**」といいます。

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、「カーソルとは ?」 31 ページ を読み終えていることが前提となります。

メンバシップ、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。

- **メンバシップ**　結果セットのローのセットです。プライマリ・キー値で指定されています。
- **順序**　結果セットにあるローの順序です。
- **値**　結果セットにあるローの値です。

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

EmployeeID	Surname
1	Whitney
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。

カーソル感知性の概要

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。特に、カーソル感知性は、変更内容が表示されるかどうかという観点から定義されています。

- **insensitive カーソル**　カーソルが開いているとき、結果セットは固定です。基本となるデータに加えられた変更はすべて非表示です。「insensitive カーソル」 48 ページを参照してください。
- **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。「sensitive カーソル」 49 ページを参照してください。
- **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シップ、順序、または値に反映されます。「asensitive カーソル」 50 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シップは固定です。「value-sensitive カーソル」 51 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、実行とパフォーマンスの両面でさまざまな制約があります。詳細については、「カーソルの感知性とパフォーマンス」 53 ページを参照してください。

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

EmployeeID	Surname
…	…

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)
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。

動作	結果
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**　結果セットのメンバシップと値を変更したかどうか確定できません。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、結果セットの順序を変更した方法で、現在更新中の結果セットに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。結果セットのメンバ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、警告 SQLE_ROW_UPDATED_WARNING 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りした場合には、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	「**ローが見つかりません。**」というエラーを返す。結果セットのメンバ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、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)	「**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。結果セットのメンバ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、つまりロー 105 の前にある。

動作	結果
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**　結果セットのメンバシップと値を変更したかどうか確定できません。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。

バルク・オペレーション・モードでは警告またはエラーが発生しない
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。

insensitive カーソル

insensitive カーソルには、insensitive メンバシップ、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、iAnywhere 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、iAnywhere 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い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。

  最適化とワーク・テーブル使用の詳細については、「クエリの最適化と実行」 『SQL Anywhere サーバ - SQL の使用法』を参照してください。

asensitive カーソル

asensitive カーソルには、メンバシップ、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、メンバシップについては insensitive です。また、順序と結果セットの値については sensitive です。

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	iAnywhere 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。詳細については、「カーソル感知性の例：削除されたロー」 44 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』を参照してください。

詳細については、「最後に読み込まれた後で、ローは更新されています。」『SQL Anywhere 10 - 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。

  詳細については、「最後に読み込まれた後で、ローは更新されています。オペレーションはキャンセルされました。」『SQL Anywhere 10 - 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「**カーソルの現在のローがありません。**」というエラーが返され、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 カーソルのメンバシップは値とともに変化するからです。

  詳細については、「カーソルの現在のローがありません。」『SQL Anywhere 10 - 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。この稼働条件は、パフォーマンスに影響を与える場合があります。

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。詳細については、「カーソルによるローの変更」 38 ページを参照してください。

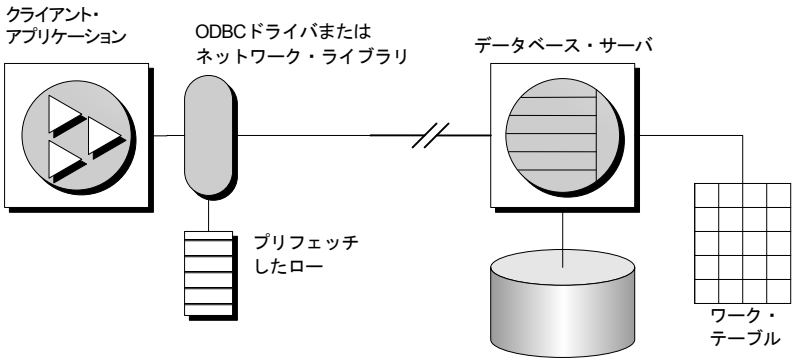
カーソルの感知性とパフォーマンス

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。

ローのプリフェッチ

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・トラフィック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 [データベース]」『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 On または Off に設定できます。デフォルトでは On 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。

詳細については、「prefetch オプション [データベース]」『SQL Anywhere サーバ - データベース管理』を参照してください。

- 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。

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、その結果、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
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) という値に更新する。
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、基本の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。iAnywhere 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 3.0 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。更新内容の消失を回避するには、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 文では、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。

詳細については、「SACommand クラス」 195 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

ODBC のスクロール可能なカーソル・タイプ	SQL Anywhere のカーソル
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」 43 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 カーソル特性の選択」 480 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 2.0 と 3.0 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。iAnywhere 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」 49 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。

結果セットの記述

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。たとえば、ユーザが表示するカラムを選択できるレポート・アプリケーションのように、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は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。

  詳細については、「SQLDA (SQL descriptor area)」 554 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の制御

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・モードの設定

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、各コマンドの最後に 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の影響については、「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。

オートコミットの動作を制御する

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。

詳細については、「オートコミット実装の詳細」 65 ページ を参照してください。

- **オートコミット・モードを制御するには、次の手順に従います (ADO.NET)。**

1. 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。

   詳細については、「Transaction 処理」 142 ページを参照してください。

♦ **オートコミット・モードを制御するには、次の手順に従います (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・モードを制御するには、次の手順に従います (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・モードを制御するには、次の手順に従います (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・モードを制御するには、次の手順に従います (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・モードを制御するには、次の手順に従います (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・モードを制御するには、次の手順に従います (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、sqlanywhere_set_option 関数を使用します。

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

♦ **オートコミット・モードを制御するには、次の手順に従います (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。

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 の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。JDBC の場合、この動作は chained データベース・オプションによって暗黙的に制御されます。

  Embedded SQL、JDBC、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。

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
ADO.NET、ADO/OLE DB、ODBC、PHP 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。

独立性レベルの制御

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

ODBC など、インタフェースによっては、接続時に接続の独立性レベルを設定できます。このレベルは isolation_level データベース・オプションを使って、後でリセットできます。「isolation_level オプション [互換性]」 『SQL Anywhere サーバ - データベース管理』を参照してください。

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。

第 4 章

3 層コンピューティングと分散トランザクション

目次

3 層コンピューティングと分散トランザクションの概要

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。

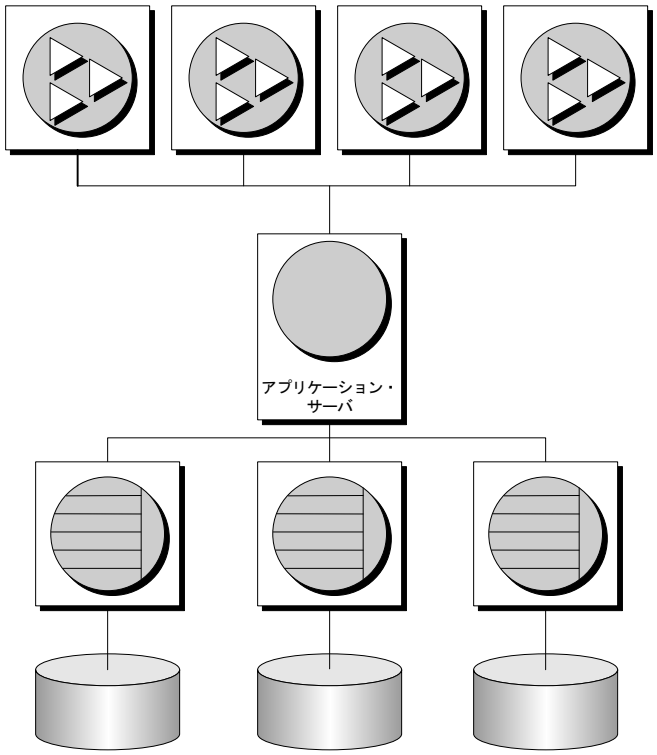
3 層環境では、クライアント・アプリケーションと一連のリソース・マネージャの間にアプリケーション・サーバを置きますが、これが一般的な分散トランザクション環境です。Sybase EAServer と他のアプリケーション・サーバの一部もトランザクション・サーバです。

Sybase EAServer と Microsoft Transaction Server はともに、Microsoft Distributed Transaction Coordinator (DTC) 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。この章では、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 Distributed Transaction Coordinator (DTC) 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。この章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

この章は、分散トランザクションについてある程度の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 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。ODBC ドライバと OLE DB プロバイダは、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。

- 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**」と通信できます。リソース・ディスペンサは、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。

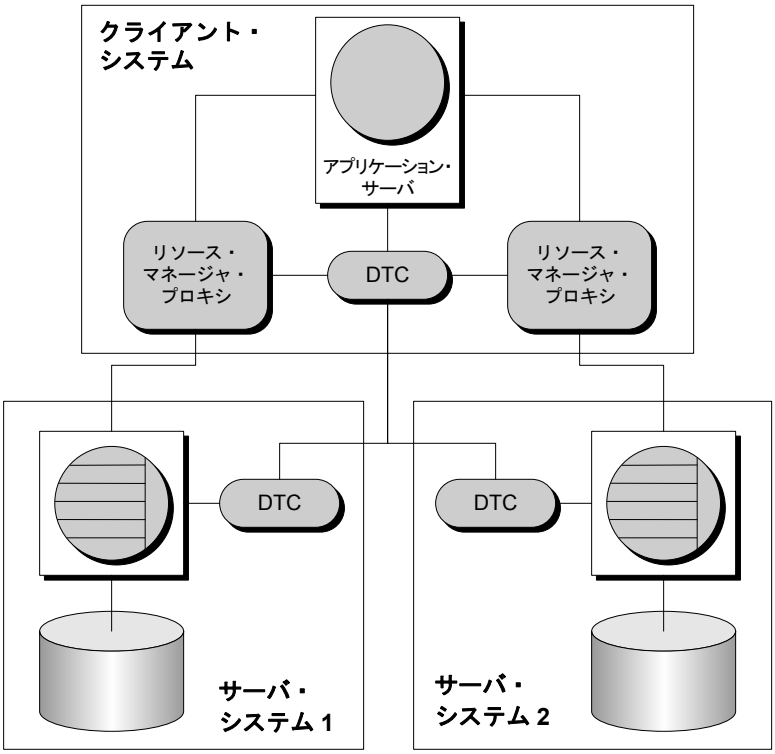
アプリケーション・サーバが DTC を使用する方法

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アプリケーション開発者は、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。しかし、アプリケーション開発者は、トランザクション管理の論理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 ODBC または OLE DB です。



この場合、単一のリソース・ディスペンサ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・ディスペンサは、トランザクション内の各接続をエンリストします。各リソース・マネージャを DTC とデータベースの両方に接続してください。これで、作業を実行したり、要求次第でトランザクション・ステータスを DTC に通知したりできます。

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 DTC サービスを実行中にしてください。Windows の [コントロール パネル] – [サービス] を選択し、DTC サービスを制御できます。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の使用

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プされます。

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
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。「-tmf 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。
- **-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。「-tmt 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。

EAServer と SQL Anywhere の併用

この項では、EAServer 3.0 以降を SQL Anywhere とともに動作させるために必要な作業の概要について説明します。詳細については、EAServer のマニュアルを参照してください。

EAServer の設定

Sybase EAServer システムにインストールされたすべてのコンポーネントは、同一のトランザクション・コーディネータを共有します。

EAServer 3.0 以降では、トランザクション・コーディネータを選択できます。トランザクションに SQL Anywhere を追加する場合は、トランザクション・コーディネータに DTC を使用してください。この項では、EAServer 3.0 を設定して DTC をトランザクション・コーディネータとして使用する方法について説明します。

EAServer のコンポーネント・サーバ名は、Jaguar です。

♦ EAServer を設定して Microsoft DTC トランザクション・モデルを使用するには、次の手順に従います。

1. Jaguar サーバが実行中であることを確認します。

   通常、Windows 上で、Jaguar サーバはサービスとして実行されます。EAServer 3.0 に付属のインストール済み Jaguar サーバを手動で起動するには、[スタート] – [プログラム] – [Sybase] – [EAServer] を選択します。

2. Jaguar Manager を起動します。

   Windows デスクトップから、[スタート] – [プログラム] – [Sybase] – [EAServer] – [Jaguar Manager] を選択します。

3. Jaguar Manager から Jaguar サーバに接続します。

   [Sybase Central] から、[ツール] – [接続] – [Jaguar Manager] を選択します。接続ダイアログで、ユーザ名に **jagadmin**、パスワードには何も指定しないで、ホスト名に **localhost** を入力します。[OK] をクリックして接続します。

4. Jaguar サーバにトランザクション・モデルを設定します。

   左ウィンドウ枠で、[Servers] フォルダを開きます。右ウィンドウ枠で、設定するサーバを右クリックし、ポップアップ・メニューから [Server Properties] を選択します。[Transaction] タブをクリックしてから、トランザクション・モデルとして Microsoft DTC を選択します。[OK] をクリックし、操作を完了します。

コンポーネントのトランザクション属性の設定

EAServer の場合、2 つ以上のデータベース上で操作を実行するコンポーネントを実装できます。その場合、このコンポーネントに「**トランザクション属性**」を割り当てて、トランザクションとの関係を定義します。トランザクション属性には、次の値を指定できます。

- **Not Supported**　コンポーネントのメソッドがトランザクションの一部として実行されることはありません。トランザクション内で実行中の別のコンポーネントによってコンポーネントがアクティブにされると、新しいインスタンスの作業は既存のトランザクションの外で実行されます。これはデフォルトです。
- **Supports Transaction**　コンポーネントをトランザクションのコンテキストで実行できますが、コンポーネントのメソッドを実行するための接続は必要ありません。コンポーネントがベース・クライアントによって直接インスタンス化された場合、EAServer はトランザクションを開始しません。コンポーネント A がコンポーネント B によってインスタンス化され、コンポーネント B がトランザクション内で実行されている場合、コンポーネント A は同じトランザクション内で実行されます。
- **Requires Transaction**　コンポーネントは常にトランザクションの中で実行されます。コンポーネントがベース・クライアントによって直接インスタンス化された場合、新しいトランザクションが開始されます。コンポーネント A がコンポーネント B によってアクティブにされ、B がトランザクション内で実行されている場合、A は同じトランザクション内で実行されます。B がトランザクション内で実行されていない場合、A は新しいトランザクション内で実行されます。
- **Requires New Transaction**　コンポーネントがインスタンス化されると、新しいトランザクションが開始されます。コンポーネント A がコンポーネント B によってアクティブにされ、B がトランザクション内で実行されている場合、A は B のトランザクションの結果に影響されない新しいトランザクションを開始します。B がトランザクション内で実行されていない場合、A は新しいトランザクション内で実行されます。

たとえば、EAServer に SVU パッケージとして含まれている Sybase Virtual University サンプル・アプリケーションの中で、SVUEnrollment コンポーネントの enroll メソッドは、2 つの独立した操作 (講座の座席の予約、学生への受講費の請求) を実行します。これらの 2 つの操作は、1 つのトランザクションとして処理される必要があります。

Microsoft Transaction Server の場合も、Enterprise Application Server の場合と同様に、属性値のセットが提供されます。

♦ コンポーネントのトランザクション属性を設定するには、次の手順に従います。

1. Jaguar Manager 内でコンポーネントを指定します。

   Jaguar サンプル・アプリケーションの SVUEnrollment コンポーネントを検索するには、Jaguar サーバに接続し、[Packages] フォルダを開いて、SVU パッケージを開きます。パッケージに含まれるコンポーネントが、右ウィンドウ枠にリストされます。

2. 目的のコンポーネントに対して、トランザクション属性を設定します。

コンポーネントを右クリックし、ポップアップ・メニューから [Component Properties] を選択します。[Transaction] タブをクリックし、リストからトランザクション属性を選択します。[OK] をクリックし、操作を完了します。

SVUEnrollment コンポーネントには、すでに [Requires Transaction] のマークが付いています。

コンポーネントのトランザクション属性が設定されると、そのコンポーネントから SQL Anywhere のデータベース操作を実行できます。また、トランザクション処理が指定したレベルで行われることが保証されます。

パート II. データベースにおける Java

パート II では、Java とデータベースにおける Java について説明します。

第 5 章

データベースにおける Java

目次

データベースにおける Java の概要

SQL Anywhere では、「**Java クラスのランタイム環境**」を用意しています。これは、Java クラスをデータベース・サーバで実行できるようにします。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 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できます。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。
- Java は、データベースの整合性、セキュリティ、堅牢性を保ちながらデータベースで使用できます。

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。

データベースにおける Java に対する理解

次の表は、データベースでの Java の使用に関するマニュアルの概要を示します。

タイトル	内容
「データベースにおける Java」 81 ページ (この章)	Java の概念と SQL Anywhere への適用方法
「チュートリアル：データベースにおける Java の使用」 93 ページ	データベースで Java を使用する場合の手順
「SQL Anywhere JDBC API」 491 ページ	分散コンピューティングを含む Java クラスからのデータのアクセス

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	参照箇所
Java の使用を開始する Java 開発者	「データベースにおける Java のランタイム環境」 89 ページ 「データベースにおける Java のチュートリアルの概要」 94 ページ

対象読者	参照箇所
データベースでの Java の主な特徴を知りたい方	「データベースにおける Java の Q & A」 84 ページ
Java からデータにアクセスする方法を知りたい方	「SQL Anywhere JDBC API」 491 ページ

データベースにおける Java の Q & A

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・サーバで Java を実行できる**　外部 Java 仮想マシン (VM) は、データベース・サーバで Java コードを実行します。
- **Java からデータにアクセスできる**　内部 JDBC ドライバによって、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。

データベースに Java クラスを格納する方法は？

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。

次に、これらのコンパイル済みクラスをデータベースにインストールします。これらのクラスは、インストール後、ストアド・プロシージャとしてデータベース・サーバで実行できます。たとえば、次の文は、Java ストアド・プロシージャを作成します。

```
CREATE PROCEDURE insertfix()
EXTERNAL NAME 'JDBCExample.InsertFixed ()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です。Java の記述やコンパイルには、Sun Microsystems Java Development Kit などの Java 開発環境が必要です。

詳細については、「Java クラスをデータベースにインストールする」 101 ページを参照してください。

Java はデータベースでどのように実行されるか？

SQL Anywhere では、「**Java 仮想マシン (VM)**」を使用します。Java VM はコンパイル済みの Java 命令を解釈し、データベース・サーバで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 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に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
- Java コードが解釈され、Java VM に受け入れられるオペレーションだけが実行される。

データベースにおける Java をサポートするプラットフォームは？

データベースおける Java は、すべての Windows オペレーティング・システム (Windows CE を除く)、UNIX、NetWare でサポートされます。

Java と SQL を一緒に使用する方法は？

Java メソッドは、ストアド・プロシージャとして宣言されるため、SQL ストアド・プロシージャのように呼び出すことができます。

Sun Microsystems 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。

SQL から Java にアクセスする方法は？

Java メソッドは、SQL から呼び出すことができるストアド・プロシージャとして処理できます。

メソッドを実行するストアド・プロシージャを作成します。次に例を示します。

```
CREATE PROCEDURE javaproc()
EXTERNAL NAME 'JDBCExample.MyMethod ()V'
LANGUAGE JAVA;
```

詳細については、「CREATE PROCEDURE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

たとえば、SQL 関数 PI(*) は、pi の値を返します。Java API クラス java.lang.Math は同じ値を返す PI という名前のパラレル・フィールドを持っています。しかし、java.lang.Math はまた自然対

数の底と IEEE 754 標準に規定された 2 つの引数の剰余演算を計算するメソッドを返す E という名前のフィールドも持っています。

Java API の他のメンバはもっと特殊な機能を提供します。たとえば、java.util.Stack は指定順でリストを保管できる後入れ先出しキューを生成し、java.util.HashTable はキーに値をマップし、java.util.StringTokenizer は文字列を個々のワード単位に分割します。

サポートされるのはどの Java クラスか？

データベースは、すべての Java API クラスをサポートするわけではありません。アプリケーションのユーザ・インタフェース・コンポーネントを含む *java.awt* パッケージのような一部のクラスは、データベース・サーバ内部での使用に適していません。また、ディスクに情報を書き込む *java.io* の一部を含むその他のクラスは、データベース・サーバ環境でサポートされません。

データベースで独自の Java クラスを使用する方法は？

データベースに独自の Java クラスをインストールできます。たとえば、ユーザが作成した Employee クラスまたは Package クラスを Java で設計して記述し、Java コンパイラでコンパイルできます。

ユーザが作成した 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むことができます。クラスをデータベースにインストールすると、SQL Anywhere によってこれらのクラスをデータベースのすべての部分や演算で使用し、それらの機能 (クラスまたはインスタンス・メソッドの形式) をストアド・プロシージャの呼び出しと同じように簡単に実行できます。

Java クラスとストアド・プロシージャは異なる
Java クラスは、ストアド・プロシージャとは異なります。ストアド・プロシージャは SQL で記述されますが、Java クラスではより強力な言語を提供します。また、ストアド・プロシージャと同じように簡単に、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。

詳細については、「Java クラスをデータベースにインストールする」 101 ページを参照してください。

Java を使用してデータにアクセスできるか？

JDBC インタフェースは、データベース・システムにアクセスするために設計された業界標準です。JDBC クラスは、データベースへの接続、SQL 文を使用したデータの要求、クライアント・アプリケーションで処理可能な結果セットの返送を実行するように設計されています。

通常、クライアント・アプリケーションは JDBC クラスを使用し、データベース・システム・ベンダが JDBC ドライバを提供します。このドライバによって、JDBC クラスは接続を確立できます。

jConnect または iAnywhere JDBC ドライバを使用して、JDBC を介してクライアント・アプリケーションを SQL Anywhere に接続できます。SQL Anywhere は内部 JDBC ドライバも提供している

ため、これによって、データベースにインストールされた Java クラスは SQL 文を実行する JDBC クラスを使用できます。「SQL Anywhere JDBC API」 491 ページを参照してください。

クライアントからサーバへクラスを移動できるか？

エンタープライズ・アプリケーションのレベル間を移動できる Java クラスを作成することができます。また、同じ Java クラスを、クライアント・アプリケーション、中間層、またはデータベースなど、最も適切な場所に統合できます。

ビジネス論理を含むクラスは、データベース・サーバなど、どのレベルのエンタープライズ・システムにも移動できます。これによって、リソースを最も適切に使用できる柔軟性が得られます。また、エンタープライズ・カスタマは、非常に高い柔軟性を持つ多層アーキテクチャで、単一の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ます。

データベースで Java を使用してできないことは何か？

SQL Anywhere は、Java 開発環境ではなく、Java クラスのランタイム環境です。

データベースで実行できないタスクを次に示します。

- クラス・ソース・ファイル (**.java* ファイル) の編集
- Java クラス・ソース・ファイル (**.java* ファイル) のコンパイル
- アプレットやビジュアル・クラスなどサポートされていない Java API の実行
- ネイティブ・メソッドの実行が必要な Java メソッドの実行。データベースにインストールされたすべてのユーザ・クラスは、100% Java である必要があります。

SQL Anywhere で使用する Java クラスは、Java アプリケーション開発ツールを使用して記述およびコンパイルしてから、データベースにインストールして使用してください。

Java のエラー処理

Java のエラー処理コードは、通常の処理コードとは分離されています。

エラーによって、エラーを示す例外オブジェクトが生成されます。これは、「**例外のスロー**」と呼ばれます。スローされた例外がキャッチされ、アプリケーションの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。

データベースにおける Java のランタイム環境

この項では、Java のための SQL Anywhere ランタイム環境と、標準の Java Runtime Environment との違いについて説明します。

ランタイム Java クラス

ランタイム Java クラスは、作成時または Java 実行可能となったときにデータベースで使用可能になる低レベルのクラスです。これらのクラスには、Java API のサブセットが含まれています。また、各クラスは Sun Java Development Kit の一部です。

ランタイム・クラスは、アプリケーションを構築するための基本的な機能を提供します。ランタイム・クラスは、常にデータベースのクラスで使用できます。

ランタイム Java クラスを、ユーザが作成したクラスに統合できます。統合するには、その機能を継承するか、メソッド内での計算またはオペレーションで使用します。

例

ランタイム Java クラスに組み込まれる Java API クラスは、次のとおりです。

- **プリミティブ Java データ型**　Java のプリミティブ (ネイティブ) データ型はすべて、対応するクラスを持っています。これらの型のオブジェクトを作成できる以外に、クラスはさらに次のような便利な機能を備えています。

  Java int データ型は、java.lang.Integer に対応するクラスを持ちます。

- **ユーティリティ・パッケージ**　java.util.* パッケージには、SQL Anywhere で使用可能な SQL 関数で利用できない機能を持つ非常に便利なクラスが含まれています。

  その主なクラスを次に示します。

  - **Hashtable**　キーを値にマップします。
  - **StringTokenizer**　文字列を個々のワードに分割します。
  - **Vector**　サイズを動的に変更できるオブジェクトの配列を保持します。
  - **Stack**　後入れ先出しオブジェクト・スタックを保持します。

- **SQL オペレーション用 JDBC**　java.SQL.* パッケージには、SQL 文を使用してデータベースからデータを抽出するために Java オブジェクトが必要とするクラスが含まれています。

ユーザ定義クラスとは違って、ランタイム・クラスはデータベースに保管されません。その代わり、SQL Anywhere インストール・ディレクトリの *java* サブディレクトリにあるファイルに保持されます。

Java は大文字と小文字を区別

Java 構文は予想したとおりに実行され、SQL 構文は Java クラスが存在しても変更されません。同じ SQL 文に Java 構文と SQL 構文の両方が含まれている場合でも同じです。これは単純な文ですが、広い意味を持ちます。

Java では大文字と小文字を区別します。Java FindOut クラスは、Findout クラスとはまったく異なります。SQL は、キーワードと識別子に対して大文字と小文字を区別しません。

Java の大文字と小文字の区別は、大文字と小文字を区別しない SQL 文に埋め込まれるときにも保持されます。Java 構文の前後の部分が大文字でも小文字でも、Java の文の部分は大文字と小文字を区別します。

たとえば、次の SQL 文は、文の残りの SQL 部分に大文字と小文字が混在している場合でも、Java のオブジェクト、クラス、演算子の大文字と小文字が尊重されるため、正しく実行されます。

```
SeLeCt java.lang.Math.random();
```

Java と SQL の文字列

次の Java コード・フラグメントに示すように、Java の文字列リテラルを一対の二重引用符で識別します。

```
String str = "This is a string";
```

ただし、SQL では、次の SQL 文に示すように、文字列には一重引用符を付け、二重引用符は識別子を示します。

```
INSERT INTO TABLE DBA.t1
VALUES( 'Hello' );
```

Java ソース・コードでは二重引用符、SQL 文では一重引用符を必ず使用してください。

次の Java コード・フラグメントは、Java クラス内で使用する場合に有効です。

```
String str = new java.lang.String(
        "Brand new object" );
```

コマンド・ラインへの出力

標準出力に出力すると、コード実行の各種ポイントで変数値や実行結果を迅速にチェックできます。次の Java コード・フラグメントの 2 行目のメソッドが検出されると、それを受け入れる文字列引数が標準出力に出力されます。

```
String str = "Hello world";
System.out.println( str );
```

SQL Anywhere では標準出力がサーバ・メッセージ・ウィンドウであるため、文字列はそこに表示されます。データベース内で上記の Java コードを実行することは、次の SQL 文を実行することと同じです。

```
MESSAGE 'Hello world';
```

main メソッドの使用

次の宣言に一致する main メソッドがクラスに含まれる場合、そのメソッドは Sun Java インタプリタなどほとんどの Java Runtime Environment で自動的に実行されます。通常、この静的メソッドは、そのメソッドが Java インタプリタによって呼び出されたクラスである場合にかぎり実行されます。

```
public static void main( String args[ ] ) { }
```

Sun Java ランタイム・システムが起動すると、このメソッドが最初に呼び出されることが常に保証されています。

SQL Anywhere では、Java ランタイム・システムを常に使用できます。オブジェクトとメソッドの機能は、SQL 文を使用して、特定の動的方法でテストできます。これは、Java クラスの機能をテストするための柔軟な方法です。

持続性

Java クラスをデータベースに追加すると、REMOVE JAVA 文で明示的に削除するまでデータベースに保持されます。

Java クラスの変数は、SQL 変数と同様に接続している間だけ持続されます。

クラスの削除の詳細については、「REMOVE JAVA 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

SQL 文の Java エスケープ文字

Java コードでは、特定の特殊文字を文字列に挿入するために、エスケープ文字を使用します。次のコードでは、アポストロフィを含む文の前に新しい行とタブを挿入します。

```
String str = "¥n¥t¥This is an object¥'s string literal";
```

SQL Anywhere では、Java クラスで使用する場合にかぎり、Java エスケープ文字の使用が許可されます。ただし、SQL で使用する場合は、SQL の文字列に適用される規則に従ってください。

たとえば、SQL 文を使用して文字列値をフィールドに渡すには、次の文 (SQL エスケープ文字を含む) を使用できますが、Java エスケープ文字は使用できません。

```
SET obj.str = '¥nThis is the object''s string field';
```

SQL 文字列処理規則の詳細については、「文字列」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

IMPORT 文の使用

Java クラス宣言では、import 文を含めて、他のパッケージにあるクラスにアクセスすることが一般的です。修飾されていないクラス名を使用して、インポートされたクラスを参照できます。

たとえば、java.util パッケージの Stack クラスを、次の 2 つの方法で参照できます。

- java.util.Stack という名前を明示的に使用する
- 名前 Stack を使用し、次の import 文を含む

```
import java.util.*;
```

階層内のさらに上にあるクラスもインストールする必要がある

別のクラスによって、完全に修飾された名前で明示的に参照されたクラス、または import 文を暗黙的に使用して参照されたクラスも、データベースにインストールする必要があります。

import 文は、コンパイルされたクラス内では意図したように実行されます。ただし、SQL Anywhere ランタイム環境内では、import 文に相当するものがありません。ストアド・プロシージャで使用するすべてのクラス名を完全に修飾してください。たとえば、String 型の変数を作成する場合は、完全に修飾された名前 java.lang.String を使用してクラスを参照します。

public フィールド

オブジェクト指向型プログラミングでは、クラス・フィールドを private に定義し、それらの値を public メソッドを介してのみ使用可能にするのが一般的です。

このマニュアルで使用しているほとんどの例では、より簡潔で読みやすくするために、フィールドを public として定義しています。SQL Anywhere で public フィールドを使用すると、public メソッドにアクセスするよりもパフォーマンス面で有利です。

このマニュアルで採用されている一般規則では、SQL Anywhere で使用するように設計されたユーザ作成の Java クラスは、そのフィールドの主要な値を公開します。メソッドには、これらのフィールドに対して実行される計算オートメーションと論理が含まれます。

第 6 章

チュートリアル：データベースにおける Java の使用

目次

データベースにおける Java のチュートリアルの概要

この章では、データベースで Java を使用してタスクを実行する方法について説明します。最初に必要なのは、データベースで Java クラスをコンパイルおよびインストールし、SQL Anywhere で使用できるようにすることです。

次に Java メソッドの作成と SQL からの呼び出しに関する手順について簡単に説明します。Java クラスをコンパイルしてデータベースにインストールする方法について説明します。また、SQL 文からクラスと、そのメンバとメソッドにアクセスする方法についても説明します。

稼働条件

このチュートリアルでは、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。

リソース

このサンプルで使用するソース・コードとバッチ・ファイルは、*samples-dir¥SQLAnywhere¥JavaInvoice* にあります。

サンプル Java クラスの作成とコンパイル

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。この作業はデータベースの外部で行います。

♦ **クラスを作成してコンパイルするには、次の手順に従います。**

1. サンプル Java クラスのソース・ファイルを作成します。

   便宜上、ここではサンプル・コードが含まれています。以下のコードをコピーして *Invoice.java* に貼り付けるか、*samples-dir¥SQLAnywhere¥JavaInvoice* からファイルを取得します。

```
import java.io.*;

public class Invoice
{
   public static String lineItem1Description;
   public static double lineItem1Cost;

   public static String lineItem2Description;
   public static double lineItem2Cost;

   public static double totalSum() {
      double runningsum;
      double taxfactor = 1 + Invoice.rateOfTaxation();

         runningsum = lineItem1Cost + lineItem2Cost;
         runningsum = runningsum * taxfactor;

         return runningsum;
   }
```

```
    public static double rateOfTaxation()
    {
        double rate;
        rate = .15;

        return rate;
    }

    public static void init(
      String item1desc, double item1cost,
      String item2desc, double item2cost )
    {
        lineItem1Description = item1desc;
        lineItem1Cost = item1cost;
        lineItem2Description = item2desc;
        lineItem2Cost = item2cost;
    }


    public static String getLineItem1Description()
    {
       return lineItem1Description;
    }

    public static double getLineItem1Cost()
    {
       return lineItem1Cost;
    }

    public static String getLineItem2Description()
    {
       return lineItem2Description;
    }

    public static double getLineItem2Cost()
    {
       return lineItem2Cost;
    }

    public static boolean testOut( int[] param )
    {
       param[0] = 123;
         return true;
    }


    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();
    }
}
```

2. このファイルをコンパイルしてファイル *Invoice.class* を作成します。

   ```
   javac Invoice.java
   ```

   クラスがコンパイルされ、データベースにインストールできるようになります。

Java VM の選択

データベース・サーバは Java VM を検索するように設定する必要があります。各データベースに別の Java VM を指定できるので、java_location オプションを使用して Java VM のロケーション (パス) を指定します。「java_location オプション [データベース]」『SQL Anywhere サーバ - データベース管理』を参照してください。

このオプションを設定しないと、データベース・サーバは次のように Java VM のロケーションを検索します。

- JAVA_HOME 環境変数を確認します。
- JAVAHOME 環境変数を確認します。
- パスを確認します。
- パスに情報が設定されていない場合、エラーが返されます。

注意
JAVA_HOME および JAVAHOME 環境変数は、Java VM のインストール時に作成されます。どちらもない場合は、これらの環境変数を手動で作成し、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、java_location オプションを使用している場合は必要ありません。「java_location オプション [データベース]」『SQL Anywhere サーバ - データベース管理』を参照してください。

♦ Java VM (Interactive SQL) のロケーションを指定するには、次の手順に従います。

1. Interactive SQL を起動して、データベースに接続します。
2. [SQL 文] ウィンドウ枠で、次のコマンドを入力します。

   ```
   SET OPTION PUBLIC.java_location='path¥java.exe';
   ```

   path は Java VM のロケーション (*c:¥jdk1.5.0_06¥jre¥bin* など) を示します。

java_main_userid オプションを設定して、クラスをインストールしたりその他の Java 関連の管理タスクを実行したりするために使用する接続用のデータベース・ユーザを指定することもできます。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 オプションを使用します。「java_main_userid オプション [データベース]」『SQL Anywhere サーバ - データベース管理』と「java_vm_options オプション [データベース]」『SQL Anywhere サーバ - データベース管理』を参照してください。

データベースで Java を使用したいのに Java Runtime Environment (JRE) がインストールされていない場合は、希望する任意の Java JRE をインストールして使用することができます。JRE をインストールしたら、その JRE のルートを指すように JAVA_HOME または JAVAHOME 環境変数を設定することをおすすめします。ほとんどの Java インストーラでは、どちらか 1 つの環境変数をデフォルトで設定します。JRE がインストールされ、JAVA_HOME または JAVAHOME が適切に設定されれば、追加の手順を実行しなくても、データベースで Java を使用できるようになります。

NetWare では環境変数を使用しないため、JRE をこのパスにインストールするか、java_location オプションを適切に設定します。「java_location オプション [データベース]」 『SQL Anywhere サーバ - データベース管理』を参照してください。

サンプル 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。

♦ クラスを SQL Anywhere サンプル・データベースにインストールするには、次の手順に従います (Sybase Central の場合)。

1. Sybase Central を起動し、サンプル・データベースに接続します。
2. [Java オブジェクト] フォルダを開き、[ファイル] – [新規] – [Java クラス] を選択します。

   [Java クラス作成] ウィザードが表示されます。
3. [参照] ボタンを使用して、*Invoice.class* を検索します。
4. [完了] をクリックしてウィザードを閉じます。

♦ クラスを SQL Anywhere サンプル・データベースにインストールするには、次の手順に従います (Interactive SQL の場合)。

1. InteractiveSQL を起動して、サンプル・データベースに接続します。
2. InteractiveSQL の [SQL 文] ウィンドウ枠に、次のコマンドを入力します。

   ```
   INSTALL JAVA NEW
   FROM FILE 'path¥¥Invoice.class';
   ```

   path は、コンパイル済みクラス・ファイルのロケーションです。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。
3. [F5] キーを押して、文を実行します。

注意

- この時点では、データベース内で Java オペレーションは実行されません。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。
- 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のコピーに自動的に反映されるわけではありません。変更を反映するには、データベース内のクラスを更新する必要があります。

クラスのインストールとインストールしたクラスの更新の詳細については、「Java クラスをデータベースにインストールする」 101 ページを参照してください。

CLASSPATH 変数の使用

Sun Java Runtime Environment と Sun JDK Java コンパイラは、Java コード内で参照されるクラスを探すときに CLASSPATH 環境変数を使用します。CLASSPATH 環境変数は、Java コードと、参照されるクラスの実際のファイル・パスまたは URL ロケーション間のリンクを提供します。たとえば、import java.io.* は java.io パッケージ内のすべてのクラスを、完全に修飾された名前を必要としないで参照できるようにします。java.io パッケージのクラスを使用するために次の Java コードで必要なのはクラス名だけです。Java クラス宣言をコンパイルするシステムの CLASSPATH 環境変数には、Java ディレクトリのロケーション (java.io パッケージのルート) が含まれている必要があります。

CLASSPATH はクラスのインストールに使用される

CLASSPATH 環境変数は、クラスのインストールのときにファイルを検索するために使用できます。たとえば、次の文はユーザが作成した Java クラスをデータベースにインストールしますが、フル・パス名ではなくファイル名だけを指定しています。この文は、Java オペレーションを使用しません。

```
INSTALL JAVA NEW
FROM FILE 'Invoice.class';
```

指定したファイルが CLASSPATH 環境変数で指定されるディレクトリまたは ZIP ファイル内にある場合、SQL Anywhere はファイルを見つけることに成功し、そのクラスをインストールします。

Java クラスのメソッドへのアクセス

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として動作するストアド・プロシージャまたは関数を作成します。

♦ **Interactive SQL を使用して Java メソッドを呼び出すには、次の手順に従います。**

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 メソッドのラッパとして動作します。

   このコマンドの構文の詳細については、「CREATE PROCEDURE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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・コンソールまたはサーバ・メッセージ・ウィンドウに "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。

Java オブジェクトのフィールドとメソッドへのアクセス

Java メソッドを呼び出して、引数を渡し、値を返す方法の例をさらに示します。

♦ Invoice クラスのメソッドのストアド・プロシージャまたは関数を作成するには、次の手順に従います。

1. Invoice クラスの Java メソッドに引数を渡して、戻り値を取得する次の SQL ストアド・プロシージャを作成します。

```
-- Invoice.init takes a string argument (Ljava/lang/String;)
-- a double (D), a string argument (Ljava/lang/String;), and
-- another double (D), and returns nothing (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;

-- Invoice.rateOfTaxation take no arguments ()
-- and returns a double (D)
CREATE FUNCTION rateOfTaxation()
RETURNS DOUBLE
EXTERNAL NAME
  'Invoice.rateOfTaxation()D'
LANGUAGE JAVA;

-- Invoice.rateOfTaxation take no arguments ()
-- and returns a double (D)
CREATE FUNCTION totalSum()
RETURNS DOUBLE
EXTERNAL NAME
  'Invoice.totalSum()D'
LANGUAGE JAVA;

-- Invoice.getLineItem1Description take no arguments ()
-- and returns a string (Ljava/lang/String;)
CREATE FUNCTION getLineItem1Description()
RETURNS CHAR(50)
EXTERNAL NAME
  'Invoice.getLineItem1Description()Ljava/lang/String;'
LANGUAGE JAVA;

-- Invoice.getLineItem1Cost take no arguments ()
-- and returns a double (D)
CREATE FUNCTION getLineItem1Cost()
RETURNS DOUBLE
EXTERNAL NAME
  'Invoice.getLineItem1Cost()D'
LANGUAGE JAVA;

-- Invoice.getLineItem2Description take no arguments ()
-- and returns a string (Ljava/lang/String;)
CREATE FUNCTION getLineItem2Description()
RETURNS CHAR(50)
EXTERNAL NAME
  'Invoice.getLineItem2Description()Ljava/lang/String;'
LANGUAGE JAVA;
```

```
-- Invoice.getLineItem2Cost take no arguments ()
-- and returns a double (D)
CREATE FUNCTION getLineItem2Cost()
RETURNS DOUBLE
EXTERNAL NAME
  'Invoice.getLineItem2Cost()D'
LANGUAGE JAVA;
```

Java メソッドの引数と戻り値の記述子には次の意味があります。

フィールド・タイプ	Java データ型
B	byte
C	char
D	double
F	float
I	int
J	long
L *class-name*;	クラス *class-name* のインスタンス。クラス名は、完全に修飾された名前で、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	ブール式
[	配列の各次元ごとに 1 つ使用

これらのコマンドの構文の詳細については、「CREATE PROCEDURE 文」『SQL Anywhere サーバ - SQL リファレンス』と「CREATE FUNCTION 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

2. ラッパとして動作するストアド・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 init('Shirt',10.00,'Jacket',25.00);
SELECT getLineItem1Description() as Item1,
      getLineItem1Cost() as Item1Cost,
      getLineItem2Description() as Item2,
      getLineItem2Cost() as Item2Cost,
      rateOfTaxation() as TaxRate,
      totalSum() as Cost;
```

このクエリは、次のような値を持つ 6 つのカラムを返します。

Item1	Item1Cost	Item2	Item2Cost	TaxRate	Cost
Shirt	10	Jacket	25	0.15	40.25

Java クラスをデータベースにインストールする

Java クラスは、次に示す方法で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、Sun の 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。

- **クラスを作成するには、次の手順に従います。**

1. **クラスを定義する**　クラスを定義する Java コードを記述します。Sun Java SDK を使用している場合は、テキスト・エディタを使用できます。開発ツールを使用している場合は、その開発ツールが指示を出します。

   サポートされるクラスだけを使用する
   ユーザ・クラスは、100% Java にしてください。ネイティブ・メソッドは使用できません。

2. **クラスに名前を付けて保存する**　クラス宣言 (Java コード) を拡張子 *.java* が付いたファイルに保存します。ファイル名とクラス名が同じで、大文字と小文字の使い分けが一致していることを確認します。

   たとえば、クラス Utility は、ファイル *Utility.java* に保存されます。

3. **クラスをコンパイルする**　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。

   Sun JDK には、Java コンパイラである *javac* が含まれています。

クラスのインストール

作成した Java クラスをデータベースで使用できるようにするには、Sybase Central を使用するか、または Interactive SQL や他のアプリケーションから INSTALL JAVA 文を使用して、そのク

ラスをデータベースにインストールします。インストールするクラスのパスとファイル名を確認します。

クラスをインストールするには、DBA 権限が必要です。

♦ **クラスをインストールするには、次の手順に従います (Sybase Central の場合)。**

1. DBA ユーザとしてデータベースに接続します。
2. データベースの [Java オブジェクト] フォルダを開きます。
3. 右ウィンドウ枠を右クリックし、ポップアップ・メニューから [新規] – [Java クラス] を選択します。
4. ウィザードの指示に従います。

♦ **クラスをインストールするには、次の手順に従います (SQL の場合)。**

1. DBA ユーザとしてデータベースに接続します。
2. 次の文を実行します。

```
INSTALL JAVA NEW
FROM FILE 'path¥¥ClassName.class';
```

path はクラス・ファイルが保持されるディレクトリ、*ClassName.class* はクラス・ファイル名を表します。

二重円記号は、円記号がエスケープ文字として処理されるのを防ぐために使用します。

たとえば、*c:¥source* ディレクトリに保持されている *Utility.class* ファイルにクラスをインストールするには、次の文を実行します。

```
INSTALL JAVA NEW
FROM FILE 'c:¥¥source¥¥Utility.class';
```

相対パスを使用する場合は、データベース・サーバの現在の作業ディレクトリからの相対パスとします。

詳細については、「INSTALL JAVA 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

JAR のインストール

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。

JAR ファイルのインストール方法は、クラス・ファイルのインストール方法と同じです。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。

詳細については、「INSTALL JAVA 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

♦ **JAR をインストールするには、次の手順に従います (Sybase Central の場合)。**

1. DBA ユーザとしてデータベースに接続します。
2. データベースの [Java オブジェクト] フォルダを開きます。
3. 右ウィンドウ枠を右クリックし、ポップアップ・メニューから [新規] – [JAR ファイル] を選択します。
4. ウィザードの指示に従います。

♦ **JAR をインストールするには、次の手順に従います (SQL の場合)。**

1. DBA ユーザとしてデータベースに接続します。
2. 次の文を実行します。

```
INSTALL JAVA NEW
JAR 'jarname'
FROM FILE 'path¥¥JarName.jar';
```

クラスと JAR ファイルの更新

Sybase Central を使用するか、Interactive SQL または他のクライアント・アプリケーションで INSTALL JAVA 文を実行すると、クラスと JAR ファイルを更新できます。

クラスまたは JAR を更新するには、DBA 権限と、ディスク上のファイルで使用できる新バージョンのコンパイルされたクラス・ファイルまたは JAR ファイルが必要です。

更新されたクラスはいつ有効となるか

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、その後再接続する必要があります。

♦ **クラスまたは JAR を更新するには、次の手順に従います (Sybase Central の場合)。**

1. DBA ユーザとしてデータベースに接続します。
2. [Java オブジェクト] フォルダを開きます。
3. 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。
4. そのクラスまたは JAR ファイルを選択し、[ファイル] – [更新] を選択します。

   [更新] ダイアログが表示されます。

5. [更新] ダイアログで、更新するクラスまたは JAR ファイルの名前とロケーションを指定します。[参照] をクリックして検索することもできます。

> **ヒント**
> Java クラスや JAR ファイルのプロパティ・シートの [一般] タブで [すぐに更新] をクリックしても、更新ができます。

♦ クラスまたは JAR を更新するには、次の手順に従います (SQL の場合)。

1. DBA ユーザとしてデータベースに接続します。
2. 次の文を実行します。

```
INSTALL JAVA UPDATE
[ JAR 'jarname' ]
FROM FILE 'filename'
```

JAR を更新する場合、データベースで認識される JAR 名を入力します。「INSTALL JAVA 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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 メソッドの呼び出し

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 *samples-dir¥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
```

♦ **クラスの main メソッドを SQL から呼び出すには、次の手順に従います。**

1. 引数として文字列配列を指定し、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;
   ```

   詳細については、「CREATE PROCEDURE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

3. 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。

Java アプリケーションでのスレッドの使用

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りした場合、Java VM からは java.lang.NoSuchMethodException エラーが返されます。引数の数と型を確認してください。

詳細については、「Java オブジェクトのフィールドとメソッドへのアクセス」 99 ページを参照してください。

Java メソッドから返される結果セット

この項では、Java メソッドから結果セットを得られるようにする方法について説明します。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、次の手順に従います。**

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」 『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 メソッドのシグニチャを指定する 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()
 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 メソッドのシグニチャで、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。

Java メソッドのシグニチャの詳細については、「CREATE PROCEDURE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

返される結果セットの詳細については、「結果セットを返す」 513 ページを参照してください。

Java からストアド・プロシージャを経由して値を返す

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 メソッドのシグニチャで、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。

メソッドのシグニチャを含む構文の詳細については、「CREATE PROCEDURE 文」『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 の起動と停止

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。この文を実行できるのは、DBA 権限を持つユーザのみです。構文は次のとおりです。

```
STOP JAVA
```

サポートされていない Java クラス

JDK のクラスをすべて使用できるわけではありません。データベース・サーバで使用できるランタイム Java クラスは、Java API のサブセットに属しています。次のパッケージのクラスは SQL Anywhere ではサポートしていません。

- java.applet
- java.awt
- java.awt.datatransfer
- java.awt.event
- java.awt.image
- プレフィクスが sun の全パッケージ (たとえば sun.audio)

パート III. SQL Anywhere データ・アクセス API

パート III では、SQL Anywhere のプログラミング・インタフェースについて説明します。

第 7 章

SQL Anywhere .NET データ・プロバイダ

目次

SQL Anywhere .NET データ・プロバイダの機能

注意
SQL Anywhere マニュアルでは、ADO.NET 用 SQL Anywhere .NET データ・プロバイダの API について説明しています。
ADO.NET 1.*x* を使用してアプリケーションを開発している場合は、http://www.ianywhere.com/downloads/products/sqlanywhere/sql_10_dotnet_api_reference.pdf の SQL Anywhere .NET データ・プロバイダの API リファレンスを参照してください。

SQL Anywhere は、3 つの異なるネームスペースを使用して Microsoft .NET Framework 1.*x* と 2.0 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 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 CE が稼働しているデバイスをサポートしています。

  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。

- **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。

Windows CE では、SQL Anywhere .NET データ・プロバイダのみがサポートされています。

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。

サンプル・プロジェクトの実行

SQL Anywhere .NET データ・プロバイダには、4 つのサンプル・プロジェクトが用意されています。次のものがあります。

- **SimpleCE** [接続] ボタン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 CE 用の .NET Compact Framework サンプル・プロジェクト。
- **SimpleWin32** [接続] ボタン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。
- **SimpleXML** ADO.NET を使用して 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。
- **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。

サンプル・プロジェクトについて説明したチュートリアルについては、「チュートリアル：SQL Anywhere .NET データ・プロバイダの使用」 151 ページを参照してください。

注意
SQL Anywhere をデフォルトのインストール・ディレクトリ (*C:¥Program Files¥SQL Anywhere 10*) 以外の場所にインストールした場合、サンプル・プロジェクトをロードするときにデータ・プロバイダ DLL の参照エラーが発生する可能性があります。このような場合は、新しい参照を *iAnywhere.Data.SQLAnywhere.dll* に追加してください。データ・プロバイダには、.NET Framework 1.*x* をサポートするバージョンと、.NET Framework 2.0 をサポートするバージョンの 2 つのバージョンがあります。Windows 用の 1.*x* データ・プロバイダは *install-dir¥Assembly¥v1¥iAnywhere.Data.SQLAnywhere.dll* にあります。Windows 用の 2.0 データ・プロバイダは *install-dir¥Assembly¥v2¥iAnywhere.Data.SQLAnywhere.dll* にあります。プロバイダの Windows CE バージョンは、サポートされる各 Windows CE ハードウェア・プラットフォームの *install-dir¥ce¥Assembly* にあります。
参照を DLL に追加する方法の詳細については、「プロジェクトにデータ・プロバイダ DLL への参照を追加する」 117 ページを参照してください。

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

SQL Anywhere .NET データ・プロバイダを使用するには、Visual Studio .NET プロジェクトに次の 2 つの項目を含めてください。

- SQL Anywhere .NET データ・プロバイダ DLL への参照
- SQL Anywhere .NET データ・プロバイダ・クラスを参照するソース・コード内の行

次に手順を説明します。

SQL Anywhere .NET データ・プロバイダのインストールと登録については、「SQL Anywhere .NET データ・プロバイダの配備」 145 ページを参照してください。

プロジェクトにデータ・プロバイダ DLL への参照を追加する

参照を追加して、SQL Anywhere .NET データ・プロバイダのコードを検索するために必要な DLL を Visual Studio .NET に伝えます。

♦ Visual Studio .NET プロジェクトに SQL Anywhere .NET データ・プロバイダへの参照を追加するには、次の手順に従います。

1. Visual Studio .NET 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] ウィンドウで、[参照設定] を右クリックし、ポップアップ・メニューから [参照の追加] を選択します。

   [参照の追加] ダイアログが表示されます。
3. [.NET] タブで、[参照] をクリックして *iAnywhere.Data.SQLAnywhere.dll* を見つけます。.NET 1.*x* と 2.0、および Windows と Windows CE にはそれぞれ個別の DLL バージョンがあります。

   - Windows .NET 2.0 データ・プロバイダの場合、デフォルトのロケーションは *install-dir¥Assembly¥v2* です。
   - Windows CE .NET 2.0 データ・プロバイダの場合、デフォルトのロケーションはサポートされる Windows CE ハードウェア・プラットフォームごとに *install-dir¥ce¥Assembly¥v2* (たとえば *ce¥Assembly¥v2¥arm.30*) です。
   - Windows .NET 1.*x* データ・プロバイダの場合、デフォルトのロケーションは *install-dir¥Assembly¥v1* です。
   - Windows CE .NET 1.*x* データ・プロバイダの場合、デフォルトのロケーションはサポートされる Windows CE ハードウェア・プラットフォームごとに *install-dir¥ce¥Assembly¥v1* (たとえば *ce¥Assembly¥v1¥arm.30*) です。
4. DLL を選択して [開く] をクリックします。

   インストールされている DLL の詳細リストについては、「SQL Anywhere .NET データ・プロバイダに必要なファイル」 145 ページを参照してください。

5. DLL がプロジェクトに追加されているかどうかを確認できます。[参照の追加] ダイアログを開き、[.NET] タブをクリックします。[選択されたコンポーネント] リストに *iAnywhere.Data.SQLAnywhere.dll* が表示されます。[OK] をクリックしてダイアログを閉じます。

   プロジェクトの [ソリューション エクスプローラ] ウィンドウの [参照設定] フォルダに DLL が追加されます。

ソース・コードのデータ・プロバイダ・クラスを使用する

SQL Anywhere .NET データ・プロバイダのネームスペースとネームスペースで定義したタイプの使用を容易にするには、ソース・コードにディレクティブを追加してください。

♦ データ・プロバイダのネームスペースをコードで容易に使用できるようにするには、次の手順に従います。

1. Visual Studio .NET を起動し、プロジェクトを開きます。
2. 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は、プロジェクトの先頭にある 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 .NET を使用している場合は、プロジェクトの先頭で行 Public Class Form1 の前に次の行を追加します。

     ```
     Imports iAnywhere.Data.SQLAnywhere
     ```

このディレクティブは必要ありませんが、これによって SQL Anywhere ADO.NET クラスの省略形を使用できます。次に例を示します。

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

ディレクティブがなくても、次のソース・コードを使用できます。

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

データベースへの接続

データを操作するには、アプリケーションをデータベースに接続してください。この項では、SQL Anywhere データベースに接続するためのコードを作成する方法について説明します。

詳細については、「SAConnectionStringBuilder クラス」 253 ページと「ConnectionName プロパティ」 261 ページを参照してください。

♦ **SQL Anywhere データベースに接続するには、次の手順に従います。**

1. SAConnection オブジェクトを割り付けます。

   次のコードは、conn という名前の SAConnection オブジェクトを作成します。

   ```
   SAConnection conn = new SAConnection(connection-string)
   ```

   アプリケーションから 1 つのデータベースに対して複数の接続を確立できます。一部のアプリケーションは、SQL Anywhere データベースに対して 1 つの接続を使用し、この接続を常時開いておきます。これを行うには、接続に対してグローバル変数を宣言します。

   ```
   private SAConnection       _conn;
   ```

   詳細については、*samples-dir¥Samples¥SQLAnywhere¥ADO.NET¥TableViewer* のサンプル・コードと「Table Viewer サンプル・プロジェクトの知識」 159 ページを参照してください。

2. 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。

   次に例を示します。

   ```
   "Data Source=SQL Anywhere 10 Demo;UID=DBA;PWD=sql"
   ```

   接続パラメータの詳細リストについては、「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。

   必要に応じて、接続文字列を指定する代わりに、ユーザ ID とパスワードを入力するようユーザに求めることができます。

3. データベースへの接続を開きます。

   次のコードは、データベースに接続しようとします。必要に応じて、データベース・サーバが自動スタートします。

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

4. 接続エラーを取得します。

   アプリケーションは、データベースに接続しようとしたときに発生するエラーを取得できるよう設計してください。次のコードは、エラーを取得してそのメッセージを表示する方法を示します。

   ```
   try {
      _conn = new SAConnection( txtConnectString.Text );
      _conn.Open();
    } catch( SAException ex ) {
      MessageBox.Show( ex.Errors[0].Source + " : "
   ```



```
        + ex.Errors[0].Message + " (" +
        ex.Errors[0].NativeError.ToString() + ")",
           "Failed to connect" );
```

また、SAConnection オブジェクトが作成されるときに接続文字列を渡す代わりに、ConnectionString プロパティを使用して接続文字列を設定できます。

```
SAConnection    _conn;
   _conn = new SAConnection();
   _conn.ConnectionString =
   "Data Source=SQL Anywhere 10 Demo;UID=DBA;PWD=sql";
   _conn.Open();
```

5. データベースとの接続を閉じます。データベースとの接続は、conn.Close() メソッドを使用して明示的に閉じるまで開いたままです。

Visual Basic .NET の接続例

次の Visual Basic .NET コードは、SQL Anywhere サンプル・データベースへの接続を開きます。

```
Private Sub Button1_Click(ByVal sender As _
   System.Object, ByVal e As System.EventArgs) _
   Handles Button1.Click
   ' Declare the connection object
   Dim myConn As New _
     iAnywhere.Data.SQLAnywhere.SAConnection()
   myConn.ConnectionString = _
     "Data Source=SQL Anywhere 10 Demo;UID=DBA;PWD=sql"
   myConn.Open()
   myConn.Close()
End Sub
```

接続プーリング

SQL Anywhere .NET データ・プロバイダは、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、接続プーリングはオンになっています。

プール・サイズは、POOLING 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、最小プール・サイズは 0 です。プールの最小と最大サイズは指定できます。次に例を示します。

```
"Data Source=SQL Anywhere 10 Demo;UID=DBA;PWD=sql;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。

接続プーリングの詳細については、「ConnectionName プロパティ」 261 ページを参照してください。

接続状態の確認

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、接続が開かれているかについて接続状態を確認してから、データベースのデータをフェッチして更新できます。接続が失われたりビジー状態であったり、別のコマンドが処理されている場合は、適切なメッセージをユーザに返すことができます。

SAConnection クラスには、接続の状態を確認する状態プロパティがあります。取り得る状態値は Open と Closed です。

次のコードは、Connection 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;
```

詳細については、「State プロパティ」 243 ページを参照してください。

データのアクセスと操作

SQL Anywhere .NET データ・プロバイダでは、SACommand オブジェクトを使用する方法と SADataAdapter オブジェクトを使用する方法の 2 つの方法を使用してデータにアクセスできます。

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

  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」 195 ページと「SADataReader クラス」 294 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」 130 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」 283 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 オブジェクトを使用したデータの検索と操作

次の各項では、SADataReader を使用してデータを取り出す方法とローを挿入、更新、または削除する方法について説明します。

SACommand オブジェクトを使用したデータの取得

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 文を発行したりストアド・プロシージャを呼び出したりできます。次のタイプのコマンドを発行して、データベースからデータを取り出すことができます。

- **ExecuteReader** 結果セットを返すコマンドを発行します。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。

  詳細については、「ExecuteReader メソッド」 214 ページを参照してください。

- **ExecuteScalar** 単一の値を返すコマンドを発行します。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。

  詳細については、「ExecuteScalar メソッド」 216 ページを参照してください。

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。

次の手順では、.NET データ・プロバイダに用意されている Simple コード・サンプルを使用します。

Simple コード・サンプルの詳細については、「Simple サンプル・プロジェクトの知識」 154 ページを参照してください。

♦ 詳細な結果セットを返すコマンドを発行するには、次の手順に従います。

1. Connection 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 10 Demo;UID=DBA;PWD=sql" );
   ```

2. 接続を開きます。

   ```
   try {
      conn.Open();
   ```

3. SQL 文を定義して実行する Command オブジェクトを追加します。

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

   ストアド・プロシージャを呼び出す場合は、ストアド・プロシージャのパラメータを指定してください。

   詳細については、「ストアド・プロシージャの使用」 140 ページと「SAParameter クラス」 370 ページを参照してください。

4. DataReader オブジェクトを返す ExecuteReader メソッドを呼び出します。

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


5. 結果を表示します。

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

6. DataReader オブジェクトと Connection オブジェクトを閉じます。

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

♦ **単一値のみを返すコマンドを発行するには、次の手順に従います。**

1. SAConnection 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 10 Demo" );
```

2. 接続を開きます。

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

3. SQL 文を定義して実行する SACommand オブジェクトを追加します。

```
SACommand cmd = new SACommand(
   "SELECT COUNT(*) FROM Employees WHERE Sex = 'M'",
   conn );
```

ストアド・プロシージャを呼び出す場合は、ストアド・プロシージャのパラメータを指定してください。

詳細については、「ストアド・プロシージャの使用」 140 ページを参照してください。

4. ExecuteScalar メソッドを呼び出して、値が含まれるオブジェクトを返します。

```
int count = (int) cmd.ExecuteScalar();
```

5. SAConnection オブジェクトを閉じます。

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

SADataReader を使用する場合、結果を目的のデータ型で返すための Get メソッドが複数あります。

詳細については、「SADataReader クラス」 294 ページを参照してください。

Visual Basic .NET DataReader の例

次の Visual Basic .NET コードは、SQL Anywhere サンプル・データベースへの接続を開き、DataReader を使用して結果セット内の最初の 5 人の従業員の姓を返します。

```
Dim myConn As New .SAConnection()
Dim myCmd As _
  New .SACommand _
  ("SELECT Surname FROM Employees", myConn)
Dim myReader As SADataReader
```



```
Dim counter As Integer
myConn.ConnectionString = _
  "Data Source=SQL Anywhere 10 Demo;UID=DBA;PWD=sql"
myConn.Open()
myReader = myCmd.ExecuteReader()
counter = 0
Do While (myReader.Read())
  MsgBox(myReader.GetString(0))
  counter = counter + 1
  If counter >= 5 Then Exit Do
Loop
myConn.Close()
```

SACommand オブジェクトを使用したローの挿入、更新、削除

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コマンド (SQL 文またはストアド・プロシージャ) を発行します。「ExecuteNonQuery メソッド」 213 ページを参照してください。

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。

オートインクリメント・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、「プライマリ・キー値の取得」 135 ページを参照してください。

コマンド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、.NET データ・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」 142 ページを参照してください。

♦ ローを挿入するコマンドを発行するには、次の手順に従います。

1. SAConnection オブジェクトを宣言して初期化します。

   ```
   SAConnection conn = new SAConnection(
      c_connStr );
   conn.Open();
   ```

2. 接続を開きます。

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

3. INSERT 文を定義して実行する SACommand オブジェクトを追加します。

   INSERT、UPDATE、または DELETE 文とともに ExecuteNonQuery メソッドを使用できます。

   ```
   SACommand insertCmd = new SACommand(
      "INSERT INTO Departments( DepartmentID, DepartmentName )
      VALUES( ?, ? )", conn );
   ```

   ストアド・プロシージャを呼び出す場合は、ストアド・プロシージャのパラメータを指定してください。

詳細については、「ストアド・プロシージャの使用」 140 ページと「SAParameter クラス」 370 ページを参照してください。

4. SACommand オブジェクトのパラメータを設定します。

   次のコードは、DepartmentID カラムと DepartmentName カラムそれぞれのパラメータを定義します。

```
SAParameter parm = new SAParameter();
parm.SADbType = SADbType.Integer;
insertCmd.Parameters.Add( parm );
parm = new SAParameter();
parm.SADbType = SADbType.Char;
insertCmd.Parameters.Add( parm );
```

5. 新しい値を挿入し、ExecuteNonQuery メソッドを呼び出して、変更内容をデータベースに適用します。

```
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();
```

6. 結果を表示し、これらを画面上のグリッドにバインドします。

```
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(10, 10);
dataGrid.Size = new Size(275, 200);
dataGrid.CaptionText = "iAnywhere SACommand Example";
this.Controls.Add(dataGrid);

dataGrid.DataSource = dr;
dataGrid.Show();
```

7. SADataReader オブジェクトと SAConnection オブジェクトを閉じます。

```
dr.Close();
conn.Close();
```

♦ ローを更新するコマンドを発行するには、次の手順に従います。

1. SAConnection オブジェクトを宣言して初期化します。

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

2. 接続を開きます。

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

3. UPDATE 文を定義して実行する SACommand オブジェクトを追加します。

   INSERT、UPDATE、または DELETE 文とともに ExecuteNonQuery メソッドを使用できます。


```
SACommand updateCmd = new SACommand(
   "UPDATE Departments SET DepartmentName = 'Engineering'
   WHERE DepartmentID=100", conn );
```

ストアド・プロシージャを呼び出す場合は、ストアド・プロシージャのパラメータを指定してください。

詳細については、「ストアド・プロシージャの使用」 140 ページと「SAParameter クラス」 370 ページを参照してください。

4. ExecuteNonQuery メソッドを呼び出して、変更内容をデータベースに適用します。

```
int recordsAffected = updateCmd.ExecuteNonQuery();
```

5. 結果を表示し、これらを画面上のグリッドにバインドします。

```
SACommand selectCmd = new SACommand(
   "SELECT * FROM Departments", conn );
SADataReader   dr = selectCmd.ExecuteReader();
dataGrid.DataSource = dr;
```

6. SADataReader オブジェクトと SAConnection オブジェクトを閉じます。

```
dr.Close();
conn.Close();
```

♦ ローを削除するコマンドを発行するには、次の手順に従います。

1. SAConnection オブジェクトを宣言して初期化します。

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

2. 接続を開きます。

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

3. DELETE 文を定義して実行する SACommand オブジェクトを作成します。

INSERT、UPDATE、または DELETE 文とともに ExecuteNonQuery メソッドを使用できます。

```
SACommand deleteCmd = new SACommand(
   "DELETE FROM Departments WHERE ( DepartmentID > 500 )", conn );
```

ストアド・プロシージャを呼び出す場合は、ストアド・プロシージャのパラメータを指定してください。

詳細については、「ストアド・プロシージャの使用」 140 ページと「SAParameter クラス」 370 ページを参照してください。

4. ExecuteNonQuery メソッドを呼び出して、変更内容をデータベースに適用します。

```
int recordsAffected = deleteCmd.ExecuteNonQuery();
```

5. SAConnection オブジェクトを閉じます。

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


DataReader スキーマ情報の取得

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 を使用している場合、GetSchemaTable メソッドを使用して結果セットに関する情報を取得できます。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。

GetSchemaTable メソッドの詳細については、「GetSchemaTable メソッド」 314 ページを参照してください。

♦ GetSchemaTable メソッドを使用して結果セットに関する情報を取得するには、次の手順に従います。

1. Connection オブジェクトを宣言して初期化します。

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

2. 接続を開きます。

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

3. 使用する SELECT 文によって SACommand オブジェクトを作成します。このクエリの結果セットに対してスキーマが返されます。

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

4. SADataReader オブジェクトを作成し、作成した Command オブジェクトを実行します。

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

5. DataTable にデータ・ソースのスキーマを設定します。

```
DataTable      schema = dr.GetSchemaTable();
```

6. SADataReader オブジェクトと SAConnection オブジェクトを閉じます。

```
dr.Close();
conn.Close();
```

7. DataTable を画面上のグリッドにバインドします。

```
dataGrid.DataSource = schema;
```

SADataAdapter オブジェクトを使用したデータのアクセスと操作

次の各項では、SADataAdapter を使用してデータを取り出す方法とローを挿入、更新、または削除する方法について説明します。

SADataAdapter オブジェクトを使用したデータの取得

SADataAdapter を使用すると、Fill メソッドを使用して DataSet 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 DataSet に設定し、結果セット全体を表示できます。

SADataAdapter を使用すると、結果セットを返す文字列 (SQL 文またはストアド・プロシージャ) を渡すことができます。SADataAdapter を使用する場合、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。SADataAdapter を使用すると、DataSet を変更できます。変更が完了したら、データベースに再接続して変更を適用してください。

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
DataSet を変更できるのは、データベースへの接続が切断されている場合のみです。つまり、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 の詳細については、「SADataAdapter クラス」 283 ページを参照してください。

SADataAdapter の例

次の例は、SADataAdapter を使用して DataSet を設定する方法を示します。

♦ **SADataAdapter オブジェクトを使用してデータを取り出すには、次の手順に従います。**

1. データベースに接続します。
2. 新しい DataSet を作成します。この場合、DataSet は Results と呼ばれます。

```
DataSet ds =new DataSet ();
```

3. SQL 文を実行して DataSet を設定する新しい SADataAdapter オブジェクトを作成します。

```
SADataAdapter da=new SADataAdapter(
    txtSQLStatement.Text, _conn);
da.Fill(ds, "Results")
```

4. DataSet を画面上のグリッドにバインドします。

```
dgResults.DataSource = ds.Tables["Results"]
```

SADataAdapter オブジェクトを使用したローの挿入、更新、削除

SADataAdapter オブジェクトは、結果セットを DataSet に取り出します。DataSet は、テーブルのコレクションと、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、DataSet を設定し、DataSet 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、DataSet を一度設定すれば、データベースと切断されていても DataSet 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 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。

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コマンド、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。つまり、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。オートインクリメント・プライマリ・キーの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。

  オートインクリメント・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、「プライマリ・キー値の取得」 135 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 オブジェクトを使用したローの挿入、更新、削除」 125 ページを参照してください。

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 を使用してローをテーブルに挿入するには、次の手順に従います。

1. SAConnection オブジェクトを宣言して初期化します。

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

2. 接続を開きます。

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

3. 新しい SADataAdapter オブジェクトを作成します。

```
SADataAdapter adapter = new SADataAdapter();
adapter.MissingMappingAction =
   MissingMappingAction.Passthrough;
adapter.MissingSchemaAction =
   MissingSchemaAction.Add;
```

4. 必要な SACommand オブジェクトを作成し、必要なパラメータを定義します。

   次のコードは、SELECT コマンドと INSERT コマンドを作成し、INSERT コマンドのパラメータを定義します。

```
adapter.SelectCommand = new SACommand(
   "SELECT * FROM Departments", conn );
adapter.InsertCommand = new SACommand(
   "INSERT INTO Departments( DepartmentID, DepartmentName )
   VALUES( ?, ? )", conn );
adapter.InsertCommand.UpdatedRowSource =
   UpdateRowSource.None;
SAParameter parm = new SAParameter();
parm.SADbType = SADbType.Integer;
parm.SourceColumn = "DepartmentID";
parm.SourceVersion = DataRowVersion.Current;
adapter.InsertCommand.Parameters.Add(
   parm );
parm = new SAParameter();
parm.SADbType = SADbType.Char;
parm.SourceColumn = "DepartmentName";
```



```
parm.SourceVersion = DataRowVersion.Current;
adapter.InsertCommand.Parameters.Add( parm );
```

5. DataTable に SELECT 文の結果を設定します。

```
DataTable dataTable = new DataTable( "Departments" );
int rowCount = adapter.Fill( dataTable );
```

6. 新しいローを DataTable に挿入し、変更内容をデータベースに適用します。

```
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 );
recordsAffected = adapter.Update( dataTable );
```

7. 更新の結果を表示します。

```
dataTable.Clear();
rowCount = adapter.Fill( dataTable );
dataGrid.DataSource = dataTable;
```

8. 接続を閉じます。

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

♦ **SADataAdapter オブジェクトを使用してローを更新するには、次の手順に従います。**

1. SAConnection オブジェクトを宣言して初期化します。

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

2. 接続を開きます。

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

3. 新しい SADataAdapter オブジェクトを作成します。

```
SADataAdapter adapter = new SADataAdapter();
adapter.MissingMappingAction =
   MissingMappingAction.Passthrough;
adapter.MissingSchemaAction =
   MissingSchemaAction.Add;
```

4. SACommand オブジェクトを作成し、そのパラメータを定義します。

   次のコードは、SELECT コマンドと UPDATE コマンドを作成し、UPDATE コマンドのパラメータを定義します。

```
adapter.SelectCommand = new SACommand(
   "SELECT * FROM Departments WHERE DepartmentID > 500",
   conn );
adapter.UpdateCommand = new SACommand(
   "UPDATE Departments SET DepartmentName = ?
   WHERE DepartmentID = ?", conn );
adapter.UpdateCommand.UpdatedRowSource =
   UpdateRowSource.None;
```

```
SAParameter parm = new SAParameter();
parm.SADbType = SADbType.Char;
parm.SourceColumn = "DepartmentName";
parm.SourceVersion = DataRowVersion.Current;
adapter.UpdateCommand.Parameters.Add( parm );
parm = new SAParameter();
parm.SADbType = SADbType.Integer;
parm.SourceColumn = "DepartmentID";
parm.SourceVersion = DataRowVersion.Original;
adapter.UpdateCommand.Parameters.Add( parm );
```

5. DataTable に SELECT 文の結果を設定します。

```
DataTable dataTable = new DataTable( "Departments" );
int rowCount = adapter.Fill( dataTable );
```

6. ローの更新値を使用して DataTable を更新し、変更内容をデータベースに適用します。

```
foreach ( DataRow row in dataTable.Rows )
{
row[1] = ( string ) row[1] + "_Updated";
}
recordsAffected = adapter.Update( dataTable );
```

7. 結果を画面上のグリッドにバインドします。

```
dataTable.Clear();
adapter.SelectCommand.CommandText =
   "SELECT * FROM Departments";
rowCount = adapter.Fill( dataTable );
dataGrid.DataSource = dataTable;
```

8. 接続を閉じます。

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

♦ SADataAdapter オブジェクトを使用してローをテーブルから削除するには、次の手順に従います。

1. SAConnection オブジェクトを宣言して初期化します。

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

2. 接続を開きます。

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

3. SADataAdapter オブジェクトを作成します。

```
SADataAdapter adapter = new SADataAdapter();
adapter.MissingMappingAction =
   MissingMappingAction.Passthrough;
adapter.MissingSchemaAction =
   MissingSchemaAction.AddWithKey;
```

4. 必要な SACommand オブジェクトを作成し、必要なパラメータを定義します。

   次のコードは、SELECT コマンドと DELETE コマンドを作成し、DELETE コマンドのパラメータを定義します。

```
adapter.SelectCommand = new SACommand(
   "SELECT * FROM Departments WHERE DepartmentID > 500",
```



```
    conn );
adapter.DeleteCommand = new SACommand(
    "DELETE FROM Departments WHERE DepartmentID = ?",
    conn );
adapter.DeleteCommand.UpdatedRowSource =
    UpdateRowSource.None;
SAParameter parm = new SAParameter();
parm.SADbType = SADbType.Integer;
parm.SourceColumn = "DepartmentID";
parm.SourceVersion = DataRowVersion.Original;
adapter.DeleteCommand.Parameters.Add( parm );
```

5. DataTable に SELECT 文の結果を設定します。

```
DataTable dataTable = new DataTable( "Departments" );
int rowCount = adapter.Fill( dataTable );
```

6. DataTable を修正し、変更内容をデータベースに適用します。

```
for each ( DataRow in dataTable.Rows )
{
    row.Delete();
}
recordsAffected = adapter.Update( dataTable )
```

7. 結果を画面上のグリッドにバインドします。

```
dataTable.Clear();
rowCount = adapter.Fill( dataTable );
dataGrid.DataSource = dataTable;
```

8. 接続を閉じます。

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

SADataAdapter スキーマ情報の取得

SADataAdapter を使用する場合、FillSchema メソッドを使用して DataSet の結果セットに関するスキーマ情報を取得できます。FillSchema 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の名前を提供します。

♦ FillSchema メソッドを使用して DataSet のスキーマ情報を取得するには、次の手順に従います。

1. SAConnection オブジェクトを宣言して初期化します。

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

2. 接続を開きます。

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

3. 使用する SELECT 文によって SADataAdapter を作成します。このクエリの結果セットに対してスキーマが返されます。

```
SADataAdapter  adapter = new SADataAdapter(
    "SELECT * FROM Employees", conn );
```

4. スキーマを設定する新しい DataTable オブジェクト (この場合は Table と呼ばれます) を作成します。

```
DataTable      dataTable = new DataTable(
   "Table" );
```

5. DataTable にデータ・ソースのスキーマを設定します。

```
adapter.FillSchema( dataTable, SchemaType.Source );
```

6. SAConnection オブジェクトを閉じます。

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

7. DataSet を画面上のグリッドにバインドします。

```
dataGrid.DataSource = dataTable;
```

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値を取得できます。

SADataAdapter を使用する場合、この方法を使用して、データ・ソースによって生成されたプライマリ・キー値を DataSet のカラムに設定できます。SACommand オブジェクトに対してこの方法を使用する場合は、パラメータからキー・カラムを取得するか、DataReader を再度開くことができます。

例

次の例は、ID と Name という 2 つのカラムが含まれる adodotnet_primarykey と呼ばれるテーブルを使用します。テーブルのプライマリ・キーは ID です。これは INTEGER であり、オートインクリメント値が含まれます。Name カラムは CHAR(40) です。

これらの例は、次のストアド・プロシージャを呼び出してデータベースからオートインクリメント・プライマリ・キー値を取り出します。

```
CREATE 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
```

♦ SACommand オブジェクトを使用してオートインクリメント・プライマリ・キーがある新しいローを挿入するには、次の手順に従います。

1. データベースに接続します。

```
SAConnection   conn = OpenConnection();
```

2. 新しいローを DataTable に挿入する SACommand オブジェクトを作成します。次のコードでは、行 int id1 = (int) parmId.Value; はローのプライマリ・キー値を確認します。


```
SACommand      cmd = conn.CreateCommand();
cmd.CommandText = "sp_adodotnet_primarykey";
cmd.CommandType = CommandType.StoredProcedure;
SAParameter parmId = new SAParameter();
parmId.SADbType = SADbType.Integer;
parmId.Direction = ParameterDirection.Output;
cmd.Parameters.Add( parmId );
SAParameter parmName = new SAParameter();
parmName.SADbType = SADbType.Char;
parmName.Direction = ParameterDirection.Input;
cmd.Parameters.Add( parmName );
parmName.Value = "R & D --- Command";
cmd.ExecuteNonQuery();
int id1 = ( int ) parmId.Value;
parmName.Value = "Marketing --- Command";
cmd.ExecuteNonQuery();
int id2 = ( int ) parmId.Value;
parmName.Value = "Sales --- Command";
cmd.ExecuteNonQuery();
int id3 = ( int ) parmId.Value;
parmName.Value = "Shipping --- Command";
cmd.ExecuteNonQuery();
int id4 = ( int ) parmId.Value;
```

3. 結果を画面上のグリッドにバインドし、変更内容をデータベースに適用します。

```
cmd.CommandText = "SELECT * FROM " +
   adodotnet_primarykey";
cmd.CommandType = CommandType.Text;
SADataReader dr = cmd.ExecuteReader();
dataGrid.DataSource = dr;
```

4. 接続を閉じます。

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

♦ SADataAdapter オブジェクトを使用してオートインクリメント・プライマリ・キーがある新しいローを挿入するには、次の手順に従います。

1. 新しい SADataAdapter を作成します。

```
DataSet        dataSet = new DataSet();
SAConnection   conn = OpenConnection();
SADataAdapter  adapter = new SADataAdapter();
adapter.MissingMappingAction =
   MissingMappingAction.Passthrough;
adapter.MissingSchemaAction =
   MissingSchemaAction.AddWithKey;
```

2. DataSet のデータとスキーマを設定します。これを行うために、SADataAdapter.Fill メソッドによって SelectCommand が呼び出されます。また、既存のレコードが必要ない場合は、Fill メソッドと SelectCommand を使用しないで DataSet を手動でも作成できます。

```
adapter.SelectCommand = new SACommand( "select * from  + adodotnet_primarykey", conn );
```

3. データベースからプライマリ・キー値を取得する新しい SACommand オブジェクトを作成します。

```
adapter.InsertCommand = new SACommand(
   "sp_adodotnet_primarykey", conn );
adapter.InsertCommand.CommandType =
```

```
    CommandType.StoredProcedure;
adapter.InsertCommand.UpdatedRowSource =
    UpdateRowSource.OutputParameters;
SAParameter parmId = new SAParameter();
parmId.SADbType = SADbType.Integer;
parmId.Direction = ParameterDirection.Output;
parmId.SourceColumn = "ID";
parmId.SourceVersion = DataRowVersion.Current;
adapter.InsertCommand.Parameters.Add( parmId );
SAParameter parmName = new SAParameter();
parmName.SADbType = SADbType.Char;
parmName.Direction = ParameterDirection.Input;
parmName.SourceColumn = "name";
parmName.SourceVersion = DataRowVersion.Current;
adapter.InsertCommand.Parameters.Add( parmName );
```

4. DataSet を設定します。

   ```
   adapter.Fill( dataSet );
   ```

5. DataSet に新しいローを挿入します。

   ```
   DataRow row = dataSet.Tables[0].NewRow();
   row[0] = -1;
   row[1] = "R & D --- Adapter";
   dataSet.Tables[0].Rows.Add( row );
   row = dataSet.Tables[0].NewRow();
   row[0] = -2;
   row[1] = "Marketing --- Adapter";
   dataSet.Tables[0].Rows.Add( row );
   row = dataSet.Tables[0].NewRow();
   row[0] = -3;
   row[1] = "Sales --- Adapter";
   dataSet.Tables[0].Rows.Add( row );
   row = dataSet.Tables[0].NewRow();
   row[0] = -4;
   row[1] = "Shipping --- Adapter";
   dataSet.Tables[0].Rows.Add( row );
   ```

6. DataSet の変更内容をデータベースに適用します。Update メソッドが呼び出されると、プライマリ・キー値はデータベースから取得された値に変更されます。

   ```
   adapter.Update( dataSet );
   dataGrid.DataSource = dataSet.Tables[0];
   ```

   新しいローを DataTable に追加して Update メソッドを呼び出すと、SADataAdapter は InsertCommand を呼び出し、出力パラメータを新しい各ローのキー・カラムに対してマッピングします。Update メソッドが呼び出されるのは 1 回だけですが、InsertCommand は Update によって、追加される新しいローごとに必要な回数だけ呼び出されます。

7. データベースとの接続を閉じます。

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

BLOB の処理

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

合は GetChars メソッドを使用します。それ以外の場合、データベースからフェッチする他のデータと同じ方法で BLOB データが処理されます。

詳細については、「GetBytes メソッド」 302 ページと「GetChars メソッド」 304 ページを参照してください。

♦ **GetChars メソッドを使用して文字列を返すコマンドを発行するには、次の手順に従います。**

1. Connection オブジェクトを宣言して初期化します。
2. 接続を開きます。
3. SQL 文を定義して実行する Command オブジェクトを追加します。

```
SACommand cmd = new SACommand(
   "SELECT int_col, blob_col FROM test", conn );
```

4. DataReader オブジェクトを返す ExecuteReader メソッドを呼び出します。

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

次のコードは、結果セットから 2 つのカラムを読み込みます。最初のカラムは整数 (GetInt32 (0)) で、2 番目のカラムは LONG VARCHAR です。GetChars を使用して、LONG VARCHAR カラムから 100 文字が 1 回で読み込まれます。

```
int length = 100;
char[] buf = new char[ length ];
int intValue;
long dataIndex = 0;
long charsRead = 0;
long blobLength = 0;
while( reader.Read() ) {
   intValue = reader.GetInt32( 0 );
   while ( ( charsRead = reader.GetChars(
      1, dataIndex, buf, 0, length ) ) == ( long )
         length ) {
   dataIndex += length;
}
blobLength = dataIndex + charsRead;
}
```

5. DataReader オブジェクトと Connection オブジェクトを閉じます。

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

時間値の取得

.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。このメソッドを使用すると、データが .NET Framework TimeSpan オブジェクトとして返されます。

GetTimeSpan メソッドの詳細については、「GetTimeSpan メソッド」 317 ページを参照してください。

♦ **GetTimeSpan メソッドを使用して時間値を変換するには、次の手順に従います。**

1. Connection オブジェクトを宣言して初期化します。

```
SAConnection conn = new SAConnection(
   "Data Source=dsn-time-test;UID=DBA;PWD=sql" );
```

2. 接続を開きます。

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

3. SQL 文を定義して実行する Command オブジェクトを追加します。

```
SACommand cmd = new SACommand(
   "SELECT ID, time_col FROM time_test", conn )
```

4. DataReader オブジェクトを返す ExecuteReader メソッドを呼び出します。

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

次のコードは、時間を TimeSpan として返す GetTimeSpan メソッドを使用します。

```
while ( reader.Read() )
{
   int ID = reader.GetInt32();
   TimeSpan time = reader.GetTimeSpan();
}
```

5. DataReader オブジェクトと Connection オブジェクトを閉じます。

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

ストアド・プロシージャの使用

.NET データ・プロバイダとともにストアド・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。

ストアド・プロシージャを呼び出すには、SAParameter オブジェクトを作成します。次のように、疑問符をパラメータのプレースホルダとして使用します。

```
sp_producttype( ?, ? )
```

Parameter オブジェクトの詳細については、「SAParameter クラス」 370 ページを参照してください。

♦ ストアド・プロシージャを実行するには、次の手順に従います。

1. SAConnection 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 10 Demo" );
   ```

2. 接続を開きます。

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

3. SQL 文を定義して実行する SACommand オブジェクトを追加します。次のコードは、コマンドをストアド・プロシージャとして識別する CommandType プロパティを使用します。

   ```
   SACommand cmd = new SACommand( "ShowProductInfo",
      conn );
   cmd.CommandType = CommandType.StoredProcedure;
   ```

   CommandType を指定しない場合、次のように、疑問符をパラメータのプレースホルダとして使用します。

   ```
   SACommand cmd = new SACommand(
      "call ShowProductInfo(?)", conn );
   cmd.CommandType = CommandType.Text;
   ```

4. ストアド・プロシージャのパラメータを定義する SAParameter オブジェクトを追加します。ストアド・プロシージャに必要なパラメータごとに新しい SAParameter オブジェクトを作成してください。

   ```
   SAParameter param = cmd.CreateParameter();
   param.SADbType = SADbType.Int32;
   param.Direction = ParameterDirection.Input;
   param.Value = 301;
   cmd.Parameters.Add( param );
   ```

   Parameter オブジェクトの詳細については、「SAParameter クラス」 370 ページを参照してください。

5. DataReader オブジェクトを返す ExecuteReader メソッドを呼び出します。Get メソッドを使用して、結果を目的のデータ型で返します。


```
SADataReader reader = cmd.ExecuteReader();
reader.Read();
int ID = reader.GetInt32(0);
string name = reader.GetString(1);
string descrip = reader.GetString(2);
decimal price = reader.GetDecimal(6);
```

6. SADataReader オブジェクトと SAConnection オブジェクトを閉じます。

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

ストアド・プロシージャを呼び出す別の方法

前述の手順 3 の説明は、ストアド・プロシージャを呼び出すための 2 つの方法を示します。Parameter オブジェクトを使用しないでストアド・プロシージャを呼び出すためのもう 1 つの方法は、次のように、ソース・コードからストアド・プロシージャを呼び出す方法です。

```
SACommand cmd = new SACommand(
   "call ShowProductInfo( 301 )", conn );
```

結果セットまたは単一値を返すストアド・プロシージャを呼び出す方法の詳細については、「SACommand オブジェクトを使用したデータの取得」 123 ページを参照してください。

結果セットを返さないストアド・プロシージャを呼び出す方法の詳細については、「SACommand オブジェクトを使用したローの挿入、更新、削除」 125 ページを参照してください。

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。

SATransaction オブジェクトの詳細については、「SATransaction クラス」 437 ページを参照してください。

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。ただし、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コマンド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。

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』を参照してください。

SELECT 文を入力するときに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。

ロックと独立性レベルの詳細については、「クエリ時のロック」 『SQL Anywhere サーバ - SQL の使用法』を参照してください。

次の例では、SQL 文を発行してロールバックする SATransaction オブジェクトを使用します。このトランザクションは独立性レベル 2 (RepeatableRead) を使用します。この場合、修正対象のローに対して書き込みロックをかけて、他のデータベース・ユーザがこのローを更新できないようにします。

♦ **SATransaction オブジェクトを使用してコマンドを発行するには、次の手順に従います。**

1. SAConnection 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 10 Demo" );
```

2. 接続を開きます。

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

3. Tee shirts の価格を変更する SQL 文を発行します。


```
string stmt = "UPDATE Products SET UnitPrice =
   2000.00 WHERE name = 'Tee shirt'";
```

4. Command オブジェクトを使用して SQL 文を発行する SATransaction オブジェクトを作成します。

   トランザクションを使用して独立性レベルを指定できます。この例では、独立性レベル 2 (RepeatableRead) を使用して、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。

   ```
   SATransaction trans = conn.BeginTransaction(
      IsolationLevel.RepeatableRead );
   SACommand cmd = new SACommand( stmt, conn,
      trans );
   int rows = cmd.ExecuteNonQuery();
   ```

5. 変更内容をロールバックします。

   ```
   trans.Rollback();
   ```

   SATransaction オブジェクトを使用して、データベースの変更内容をコミットまたはロールバックできます。トランザクションを使用しない場合、.NET データ・プロバイダはオートコミット・モードで動作し、データベースの変更内容をロールバックできません。変更内容を確定するには、次のコードを使用します。

   ```
   trans.Commit();
   ```

6. SAConnection オブジェクトを閉じます。

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


エラー処理と SQL Anywhere .NET データ・プロバイダ

アプリケーションは、ADO.NET エラーを含む任意のエラーを処理できるよう設計してください。ADO.NET エラーはコード内で、アプリケーション内の他のエラーを処理する場合と同じ方法で処理されます。

SQL Anywhere .NET データ・プロバイダは、実行時にエラーが発生した場合はいつでも SAException オブジェクトをスローします。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。

競合の処理の詳細については、「SADataAdapter を使用するときの競合の解消」 130 ページを参照してください。

.NET データ・プロバイダのエラー処理の例

次に示すのは Simple サンプル・プロジェクトの例です。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、メッセージ・ボックス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 );
}
```

接続エラーの処理の例

次に示すのは Table Viewer サンプル・プロジェクトの例です。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にエラーが発生した場合、次のコードは、トライ・アンド・キャッチ・ブロックを使用してエラーとそのメッセージを取得します。

```
try {
    _conn = new SAConnection( txtConnectString.Text );
    _conn.Open();
  } catch( SAException ex ) {
    MessageBox.Show( ex.Errors[0].Source + " : "
     + ex.Errors[0].Message + " (" +
     ex.Errors[0].NativeError.ToString() + ")",
        "Failed to connect" );
```

エラー処理例の詳細については、「Simple サンプル・プロジェクトの知識」 154 ページと 「Table Viewer サンプル・プロジェクトの知識」 159 ページを参照してください。。

エラー処理の詳細については、「SAFactory クラス」 345 ページと「SAError クラス」 334 ページを参照してください。

SQL Anywhere .NET データ・プロバイダの配備

次の各項では、SQL Anywhere .NET データ・プロバイダを配備する方法について説明します。

SQL Anywhere .NET データ・プロバイダ・システムの稼働条件

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 1.*x* または 2.0 (あるいはそのいずれか)
- Visual Studio .NET 2003 または 2005、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・プロバイダに必要なファイル

SQL Anywhere .NET データ・プロバイダは、プラットフォームごとに 2 つの DLL から成ります。

Windows に必要なファイル

Windows (Windows CE を除く) の場合、次の DLL が必要です。

- *install-dir¥Assembly¥v2¥iAnywhere.Data.SQLAnywhere.dll*
- *install-dir¥Assembly¥v1¥iAnywhere.Data.SQLAnywhere.dll*

ファイル *iAnywhere.Data.SQLAnywhere.dll* は、Visual Studio プロジェクトによって参照される DLL です。1 番目の DLL は .NET Framework および .NET Compact Framework のバージョン 2.0 のアプリケーション、またはそのいずれかで必要です。2 番目の DLL は .NET Framework および .NET Compact Framework のバージョン 1.*x* のアプリケーション、またはそのいずれかで必要です。

Windows CE に必要なファイル

以下のファイルは、SQL Anywhere のインストール・ディレクトリに配置される言語 DLL を必要とするため、同様に SQL Anywhere のインストール・ディレクトリ (デフォルト・ロケーションは *C:¥Program Files¥SQL Anywhere 10¥ce¥Assembly*) にインストールしてください。

Windows CE の場合、*iAnywhere.Data.SQLAnywhere.dll* は、Visual Studio プロジェクトによって参照される DLL です。サポートされる Windows CE ハードウェア・プラットフォームごとに、.NET Compact Framework のバージョン 2.0 または 1.*x* 用のバージョンがあります。DLL は *install-dir¥ce¥Assembly* の下の次のロケーションに格納されます。

- *v2¥arm.30¥iAnywhere.Data.SQLAnywhere.dll*
- *v2¥arm.50¥iAnywhere.Data.SQLAnywhere.dll*
- *v2¥armt.40¥iAnywhere.Data.SQLAnywhere.dll*

- *v2¥x86.30¥iAnywhere.Data.SQLAnywhere.dll*
- *v1¥arm.30¥iAnywhere.Data.SQLAnywhere.dll*
- *v1¥armt.40¥iAnywhere.Data.SQLAnywhere.dll*
- *v1¥arm.50¥iAnywhere.Data.SQLAnywhere.dll*
- *v1¥x86.30¥iAnywhere.Data.SQLAnywhere.dll*

Visual Studio .NET は、.NET データ・プロバイダ DLL (*iAnywhere.Data.SQLAnywhere.dll*) をプログラムとともにデバイスに配備します。Visual Studio .NET を使用していない場合は、データ・プロバイダ DLL をプログラム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 DLL の登録

SQL Anywhere .NET データ・プロバイダ DLL (*install-dir¥Assembly¥v2 ¥iAnywhere.Data.SQLAnywhere.dll*) は、Windows (Windows CE を除く) 上の Global Assembly Cache に登録する必要があります。Global Assembly Cache には、マシンに登録されているすべてのプログラムがリストされています。.NET データ・プロバイダをインストールすると、.NET データ・プロバイダのインストール・プログラムによって DLL が登録されます。Windows CE の場合、この DLL を登録する必要はありません。

.NET データ・プロバイダを配備する場合は、.NET Framework に含まれている gacutil ユーティリティを使用して、.NET データ・プロバイダ DLL (*install-dir¥Assembly¥v2 ¥iAnywhere.Data.SQLAnywhere.dll*) を登録してください。

.NET 2.0 のトレース・サポート

SQL Anywhere ADO.NET 2.0 プロバイダでは、.NET 2.0 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 CE 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。次に、設定ファイルの例を示します。

```
<?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="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>
```

トレースの設定情報は、*app.exe.config* という名前で、アプリケーションの *bin¥debug* フォルダに配置されます。

指定できる traceOutputOptions には、次の項目などがあります。

- **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。

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。

- **SATraceAllSwitch**　すべてをトレースするスイッチです。指定すると、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#
  ```

アプリケーション・トレースの例として、ここでは TableViewer サンプルを使用します。

- **トレース用にアプリケーションを設定するには、次の手順に従います。**

1. .NET 2.0 を使用する必要があります。トレースは、.NET 1.*x* ではサポートされていません。

Visual Studio 2005 を起動し、*samples-dir¥Samples¥SQLAnywhere¥ADO.NET¥TableViewer* にある TableViewer プロジェクト・ファイル (*TableViewer.sln*) を開きます。

2. このアプリケーションは、配布された状態では .NET 1.0 を使用します。.NET 2.0 に変換するには、[ソリューション エクスプローラ] ウィンドウで [参照設定] フォルダを開きます。
3. iAnywhere.Data.SQLAnywhere を右クリックして、[削除] を選択します。これにより、プロジェクトから .NET 1.0 プロバイダが削除されます。
4. [参照設定] を右クリックして、[追加] を選択します。
5. [参照の追加] ダイアログの [.NET] タブから、[ランタイム] のバージョンが 2.0.xxxxxx である iAnywhere.Data.SQLAnywhere を選択します。
6. [OK] をクリックします。
7. 上述の設定ファイルのコピーを *TableViewer.exe.config* という名前でアプリケーションの *bin¥debug* フォルダに配置します。
8. [デバッグ] メニューから [デバッグの開始] を選択します。

アプリケーションの実行が完了すると、トレース出力ファイルが *samples-dir¥Samples¥SQLAnywhere¥ADO.NET¥TableViewer¥bin¥Debug¥myTrace.log* に作成されています。

トレースは、Windows CE ではサポートされていません。

詳細については、http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnadonet/html/tracingdataaccess.asp の「Tracing Data Access」を参照してください。

第 8 章

チュートリアル：SQL Anywhere .NET データ・プロバイダの使用

目次

.NET データ・プロバイダのチュートリアルの概要

この章では、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。

SQL Anywhere のインストール・ディレクトリがデフォルト (*C:¥Program Files¥SQL Anywhere 10*) ではない場合、サンプル・プロジェクトをロードするときにデータ・プロバイダ DLL の参照エラーが発生する可能性があります。このような場合は、新しい参照を *iAnywhere.Data.SQLAnywhere.dll* に追加してください。

参照を DLL に追加する方法の詳細については、「プロジェクトにデータ・プロバイダ DLL への参照を追加する」 117 ページを参照してください。

注意
SQL Anywhere マニュアルでは、ADO.NET 用 SQL Anywhere .NET データ・プロバイダの API について説明しています。
ADO.NET 1.0 を使用してアプリケーションを開発している場合は、http://www.ianywhere.com/downloads/products/sqlanywhere/sql_10_dotnet_api_reference.pdf の SQL Anywhere .NET データ・プロバイダの API リファレンスを参照してください。

Simple コード・サンプルの使用

このチュートリアルは、SQL Anywhere 10 に用意されている Simple プロジェクトに基づいています。

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 *samples-dir¥SQLAnywhere¥ADO.NET¥SimpleWin32* で参照できます。

samples-dir のデフォルト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。

Simple プロジェクトには、次の機能があります。

- SAConnection オブジェクトを使用したデータベースへの接続
- SACommand オブジェクトを使用したクエリの実行
- SADataReader オブジェクトを使用した結果の取得
- 基本的なエラー処理

サンプルの動作に関する詳細については、「Simple サンプル・プロジェクトの知識」 154 ページを参照してください。

♦ Visual Studio .NET で Simple コード・サンプルを実行するには、次の手順に従います。

1. Visual Studio .NET を起動します。
2. [ファイル] – [開く] – [プロジェクト] を選択します。
3. *samples-dir¥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 DLL に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 DLL への参照は次のロケーションで確認できます。

   - [ソリューション エクスプローラ] ウィンドウで、[参照設定] フォルダを開きます。
   - リストに iAnywhere.Data.SQLAnywhere が表示されます。

   データ・プロバイダ DLL に参照を追加する方法の詳細については、「プロジェクトにデータ・プロバイダ DLL への参照を追加する」 117 ページを参照してください。
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 ディレクティブを参照するには、次の手順に従います。

   - プロジェクトのソース・コマンドを開きます。[ソリューション エクスプローラ] ウィンドウで、*Form1.cs* を右クリックし、ポップアップ・メニューから [コードの表示] を選択します。

- 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;
  ```

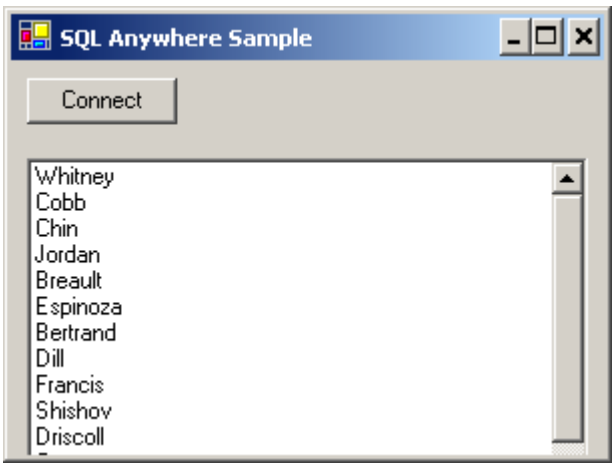
  この行は C# プロジェクトに必要です。Visual Basic .NET を使用している場合、ソース・コードに Imports 行を追加する必要があります。

6. [デバッグ] – [デバッグなしで開始] を選択するか、[Ctrl + F5] を押して、Simple サンプルを実行します。

   [SQL Anywhere サンプル] ダイアログが表示されます。

   - [SQL Anywhere サンプル] ダイアログで [接続] をクリックします。

     アプリケーションは、SQL Anywhere サンプル・データベースに接続し、次のように各従業員の姓をダイアログに入力します。

7. 画面右上の X をクリックし、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。

ここではアプリケーションを実行しました。次の項では、アプリケーション・コードについて説明します。

Simple サンプル・プロジェクトの知識

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 10 - 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、*samples-dir¥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。

```
private void btnConnect_Click(object sender,
  System.EventArgs e)
  SAConnection conn = new SAConnection(
   "Data Source=SQL Anywhere 10 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」 236 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」 195 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、事前に宣言されている SACommand オブジェクトである cmd のメンバです。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」 294 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・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、メッセージ・ボックス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」 341 ページを参照してください。

Table Viewer コード・サンプルの使用

このチュートリアルは、SQL Anywhere .NET データ・プロバイダに用意されている Table Viewer プロジェクトに基づいています。

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 *samples-dir¥SQLAnywhere¥ADO.NET¥TableViewer* で参照できます。

samples-dir のデフォルト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。

Table Viewer プロジェクトは Simple プロジェクトよりも複雑です。このプロジェクトには、次の機能があります。

- SAConnection オブジェクトを使用したデータベースへの接続
- SACommand オブジェクトを使用したクエリの実行
- SADataReader オブジェクトを使用した結果の取得
- DataGrid オブジェクトを使用したグリッドによる結果の表示
- より高度なエラー処理と結果の確認

サンプルの動作に関する詳細については、「Table Viewer サンプル・プロジェクトの知識」 159 ページを参照してください。

♦ Visual Studio .NET で Table Viewer コード・サンプルを実行するには、次の手順に従います。

1. Visual Studio .NET を起動します。
2. [ファイル] – [開く] – [プロジェクト] を選択します。
3. *samples-dir¥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場合は、データ・プロバイダ DLL に参照を追加してください。これはすでに Table Viewer コード・サンプルで行われています。データ・プロバイダ DLL への参照は次のロケーションで確認できます。
   - [ソリューション エクスプローラ] ウィンドウで、[参照設定] フォルダを開きます。
   - リストに iAnywhere.Data.SQLAnywhere が表示されます。

   データ・プロバイダ DLL に参照を追加する方法の詳細については、「プロジェクトにデータ・プロバイダ DLL への参照を追加する」 117 ページを参照してください。
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 ディレクティブを参照するには、次の手順に従います。

- プロジェクトのソース・コマンドを開きます。[ソリューション エクスプローラ] ウィンドウで、*TableViewer.cs* を右クリックし、ポップアップ・メニューから [コードの表示] を選択します。
- 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] を選択するか、[Ctrl + F5] を押して、Table Viewer サンプルを実行します。

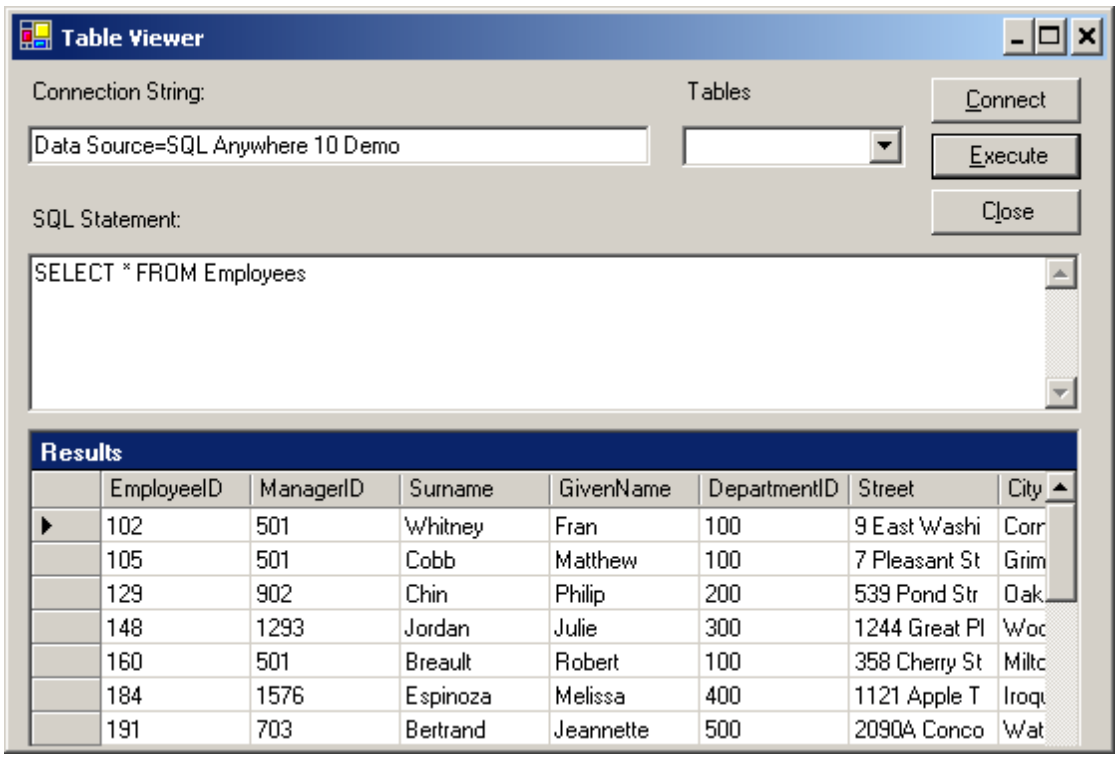
   [Table Viewer] ダイアログが表示されます。

   - [Table Viewer] ダイアログで [接続] 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します。

   - [Table Viewer] ダイアログで [実行] 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 DataList に入力します。



- また、このアプリケーションから別の SQL 文も実行できます。これを行うには、[SQL Statement] ウィンドウ枠に SQL 文を入力し、[実行] をクリックします。

7. 画面右上の [X] をクリックし、アプリケーションを終了して SQL Anywhere 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。

ここではアプリケーションを実行しました。次の項では、アプリケーション・コードについて説明します。

Table Viewer サンプル・プロジェクトの知識

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 10 - 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、*samples-dir¥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」 236 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 10 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 10 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 メンバ」 236 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」 195 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」 294 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 systable where creator = 101", _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」 341 ページを参照してください。

第 9 章

SQL Anywhere .NET 2.0 API リファレンス

目次

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。このクラスは継承できません。

構文

Visual Basic

```
Public NotInheritable Class SABulkCopy
  Implements IDisposable
```

C#

```
public sealed class SABulkCopy : IDisposable
```

備考

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

実装：IDisposable

参照

- 「SABulkCopy メンバ」 165 ページ

SABulkCopy メンバ

パブリック・コンストラクタ

メンバ名	説明
SABulkCopy コンストラクタ	SABulkCopy オブジェクトを初期化します。

パブリック・プロパティ

メンバ名	説明
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。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。
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。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。
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。

パブリック・メソッド

メンバ名	説明
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 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。

パブリック・イベント

メンバ名	説明
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。

参照

- 「SABulkCopy クラス」 165 ページ

SABulkCopy 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

構文

Visual Basic

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

C#

```
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

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

参照

- 「SABulkCopy クラス」 165 ページ
- 「SABulkCopy メンバ」 165 ページ
- 「SABulkCopy コンストラクタ」 166 ページ

SABulkCopy(String) コンストラクタ

SABulkCopy オブジェクトを初期化します。

構文

Visual Basic

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

C#

```
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。

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

参照

- 「SABulkCopy クラス」 165 ページ
- 「SABulkCopy メンバ」 165 ページ
- 「SABulkCopy コンストラクタ」 166 ページ

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

SABulkCopy オブジェクトを初期化します。

構文

Visual Basic

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


C#

```
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。

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

参照

- 「SABulkCopy クラス」 165 ページ
- 「SABulkCopy メンバ」 165 ページ
- 「SABulkCopy コンストラクタ」 166 ページ

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。

構文

Visual Basic

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

C#

```
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

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

参照

- 「SABulkCopy クラス」 165 ページ
- 「SABulkCopy メンバ」 165 ページ
- 「SABulkCopy コンストラクタ」 166 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。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。

構文

Visual Basic

```
Public Property BatchSize As Integer
```

C#

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

プロパティ値

各バッチの中のロー数。デフォルトは 0 です。

備考

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。

参照

- 「SABulkCopy クラス」 165 ページ
- 「SABulkCopy メンバ」 165 ページ

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。

構文

Visual Basic

Public Property **BulkCopyTimeout** As Integer

C#

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

プロパティ値

デフォルト値は 30 秒です。

備考

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。

参照

- 「SABulkCopy クラス」 165 ページ
- 「SABulkCopy メンバ」 165 ページ

ColumnMappings プロパティ

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

構文

Visual Basic

Public Readonly Property **ColumnMappings** As SABulkCopyColumnMappingCollection

C#

public SABulkCopyColumnMappingCollection **ColumnMappings** { get;}

プロパティ値

デフォルトでは、空のコレクションです。

備考

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プされ、2 番目は 2 番目にマップされます。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 入力可能なカラムである場合に行われます。

参照

- 「SABulkCopy クラス」 165 ページ
- 「SABulkCopy メンバ」 165 ページ

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。

構文

Visual Basic

Public Property **DestinationTableName** As String

C#

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

プロパティ値

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

備考

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。

参照

- 「SABulkCopy クラス」 165 ページ
- 「SABulkCopy メンバ」 165 ページ

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。

構文

Visual Basic

Public Property **NotifyAfter** As Integer

C#

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

プロパティ値

プロパティが設定されていない場合は、0 が返されます。

備考

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なため、データベースにローが送信されなかったり、コミットされない場合であっても、イベントは起動します。

参照

- 「SABulkCopy クラス」 165 ページ
- 「SABulkCopy メンバ」 165 ページ
- 「BulkCopyTimeOut プロパティ」 170 ページ

Close メソッド

SABulkCopy インスタンスを閉じます。

構文

Visual Basic

Public Sub **Close()**

C#

public void **Close();**

参照

- 「SABulkCopy クラス」 165 ページ
- 「SABulkCopy メンバ」 165 ページ

Dispose メソッド

SABulkCopy インスタンスを破棄します。

構文

Visual Basic

NotOverridable Public Sub **Dispose()**

C#

```
public void Dispose();
```

参照

- 「SABulkCopy クラス」 165 ページ
- 「SABulkCopy メンバ」 165 ページ

WriteToServer メソッド

指定された 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(DataRow[]) メソッド

指定された 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。

構文

Visual Basic

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

C#

```
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

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

参照

- 「SABulkCopy クラス」 165 ページ
- 「SABulkCopy メンバ」 165 ページ
- 「WriteToServer メソッド」 173 ページ
- 「DestinationTableName プロパティ」 171 ページ

WriteToServer(DataTable) メソッド

指定された 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。

構文

Visual Basic

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

C#

```
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

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

参照

- 「SABulkCopy クラス」 165 ページ
- 「SABulkCopy メンバ」 165 ページ
- 「WriteToServer メソッド」 173 ページ
- 「DestinationTableName プロパティ」 171 ページ

WriteToServer(IDataReader) メソッド

指定された 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。

構文

Visual Basic

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

C#

```
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

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

参照

- 「SABulkCopy クラス」 165 ページ
- 「SABulkCopy メンバ」 165 ページ
- 「WriteToServer メソッド」 173 ページ
- 「DestinationTableName プロパティ」 171 ページ

WriteToServer(DataTable, DataRowState) メソッド

指定されたロー・ステータスの指定された 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。

構文

Visual Basic

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

C#

```
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。

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

参照

- 「SABulkCopy クラス」 165 ページ
- 「SABulkCopy メンバ」 165 ページ
- 「WriteToServer メソッド」 173 ページ
- 「DestinationTableName プロパティ」 171 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。

構文

Visual Basic

Public Event **SARowsCopied** As SARowsCopiedEventHandler

C#

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 クラス」 165 ページ
- 「SABulkCopy メンバ」 165 ページ
- 「NotifyAfter プロパティ」 171 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。このクラスは継承できません。

構文

Visual Basic

Public NotInheritable Class **SABulkCopyColumnMapping**

C#

public sealed class **SABulkCopyColumnMapping**

備考

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

参照

- 「SABulkCopyColumnMapping メンバ」 177 ページ

SABulkCopyColumnMapping メンバ

パブリック・コンストラクタ

メンバ名	説明
SABulkCopyColumnMapping コンストラクタ	「SABulkCopyColumnMapping クラス」 177 ページの新しいインスタンスを初期化します。

パブリック・プロパティ

メンバ名	説明
DestinationColumn プロパティ	マップ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	マップされる送信先テーブルのカラムの序数を取得または設定します。
SourceColumn プロパティ	データ・ソースにマップ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・カラムの序数位置を取得または設定します。

参照

- 「SABulkCopyColumnMapping クラス」 177 ページ

SABulkCopyColumnMapping コンストラクタ

「SABulkCopyColumnMapping クラス」 177 ページの新しいインスタンスを初期化します。

SABulkCopyColumnMapping() コンストラクタ

カラムの序数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。

構文

Visual Basic

```
Public Sub New()
```

C#

```
public  SABulkCopyColumnMapping();
```

備考

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

参照

- 「SABulkCopyColumnMapping クラス」 177 ページ
- 「SABulkCopyColumnMapping メンバ」 177 ページ
- 「SABulkCopyColumnMapping コンストラクタ」 178 ページ

SABulkCopyColumnMapping(Int32, Int32) コンストラクタ

カラム序数を使用してソース・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。

構文

Visual Basic

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

C#

```
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・ソースの先頭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。テーブルの先頭カラムの序数位置は 0 です。

備考

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

参照

- 「SABulkCopyColumnMapping クラス」 177 ページ
- 「SABulkCopyColumnMapping メンバ」 177 ページ
- 「SABulkCopyColumnMapping コンストラクタ」 178 ページ

SABulkCopyColumnMapping(Int32, String) コンストラクタ

カラム序数を使用してソース・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、新しいカラム・マッピングを作成します。

構文

Visual Basic

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

C#

```
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・ソースの先頭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

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

参照

- 「SABulkCopyColumnMapping クラス」 177 ページ
- 「SABulkCopyColumnMapping メンバ」 177 ページ
- 「SABulkCopyColumnMapping コンストラクタ」 178 ページ

SABulkCopyColumnMapping(String, Int32) 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、カラム序数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。

構文

Visual Basic

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

C#

```
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。テーブルの先頭カラムの序数位置は 0 です。

備考

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

参照

- 「SABulkCopyColumnMapping クラス」 177 ページ
- 「SABulkCopyColumnMapping メンバ」 177 ページ
- 「SABulkCopyColumnMapping コンストラクタ」 178 ページ

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

カラム名を使用してソース・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。

構文

Visual Basic

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

C#

```
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。

備考

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

参照

- 「SABulkCopyColumnMapping クラス」 177 ページ
- 「SABulkCopyColumnMapping メンバ」 177 ページ
- 「SABulkCopyColumnMapping コンストラクタ」 178 ページ

DestinationColumn プロパティ

マップ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。

構文

Visual Basic

```
Public Property DestinationColumn As String
```

C#

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

プロパティ値

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)。

備考

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 クラス」 177 ページ
- 「SABulkCopyColumnMapping メンバ」 177 ページ
- 「DestinationOrdinal プロパティ」 182 ページ

DestinationOrdinal プロパティ

マップされる送信先テーブルのカラムの序数を取得または設定します。

構文

Visual Basic

Public Property **DestinationOrdinal** As Integer

C#

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

プロパティ値

マップされるカラムの送信先テーブルにおける序数を指定する整数。プロパティが設定されていない場合は -1。

備考

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 クラス」 177 ページ
- 「SABulkCopyColumnMapping メンバ」 177 ページ
- 「DestinationColumn プロパティ」 181 ページ

SourceColumn プロパティ

データ・ソースにマップされるカラムの名前を取得または設定します。

構文

Visual Basic

Public Property **SourceColumn** As String

C#

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

プロパティ値

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)。

備考

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 クラス」 177 ページ
- 「SABulkCopyColumnMapping メンバ」 177 ページ
- 「SourceOrdinal プロパティ」 183 ページ

SourceOrdinal プロパティ

データ・ソース内のソース・カラムの序数位置を取得または設定します。

構文

Visual Basic

Public Property **SourceOrdinal** As Integer

C#

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

プロパティ値

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

備考

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 クラス」 177 ページ
- 「SABulkCopyColumnMapping メンバ」 177 ページ
- 「SourceColumn プロパティ」 182 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。このクラスは継承できません。

構文

Visual Basic

Public NotInheritable Class **SABulkCopyColumnMappingCollection**
  Inherits CollectionBase

C#

public sealed class **SABulkCopyColumnMappingCollection** : CollectionBase

備考

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

参照

- 「SABulkCopyColumnMappingCollection メンバ」 184 ページ

SABulkCopyColumnMappingCollection メンバ

パブリック・プロパティ

メンバ名	説明
Capacity (CollectionBase から継承)	
Count (CollectionBase から継承)	
Item 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。

パブリック・メソッド

メンバ名	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 (CollectionBase から継承)	
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。

メンバ名	説明
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。
GetEnumerator (CollectionBase から継承)	
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。
Remove メソッド	指定された SABulkCopyColumnMapping 要素を SABulkCopyColumnMappingCollection から削除します。
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。

参照

- 「SABulkCopyColumnMappingCollection クラス」 184 ページ

Item 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。

構文

Visual Basic

```
Public Readonly Property Item ( _
   ByVal index As Integer _
) As SABulkCopyColumnMapping
```

C#

```
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。

プロパティ値

SABulkCopyColumnMapping オブジェクトが返されます。

参照

- 「SABulkCopyColumnMappingCollection クラス」 184 ページ
- 「SABulkCopyColumnMappingCollection メンバ」 184 ページ

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。

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。

構文

Visual Basic

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

C#

```
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

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

参照

- 「SABulkCopyColumnMappingCollection クラス」 184 ページ
- 「SABulkCopyColumnMappingCollection メンバ」 184 ページ
- 「Add メソッド」 186 ページ
- 「SABulkCopyColumnMapping クラス」 177 ページ

Add(Int32, Int32) 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序数を使用して、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。

構文

Visual Basic

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


C#

```
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

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

参照

- 「SABulkCopyColumnMappingCollection クラス」 184 ページ
- 「SABulkCopyColumnMappingCollection メンバ」 184 ページ
- 「Add メソッド」 186 ページ

Add(Int32, String) メソッド

カラム序数を使用してソース・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。

構文

Visual Basic

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

C#

```
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

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

参照

- 「SABulkCopyColumnMappingCollection クラス」 184 ページ
- 「SABulkCopyColumnMappingCollection メンバ」 184 ページ
- 「Add メソッド」 186 ページ

Add(String, Int32) メソッド

カラム名を使用してソース・カラムを参照し、カラム序数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。

カラムの序数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。

構文

Visual Basic

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

C#

```
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

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

参照

- 「SABulkCopyColumnMappingCollection クラス」 184 ページ
- 「SABulkCopyColumnMappingCollection メンバ」 184 ページ
- 「Add メソッド」 186 ページ

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。

構文

Visual Basic

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

C#

```
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

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

参照

- 「SABulkCopyColumnMappingCollection クラス」 184 ページ
- 「SABulkCopyColumnMappingCollection メンバ」 184 ページ
- 「Add メソッド」 186 ページ

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。

構文

Visual Basic

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

C#

```
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。

参照

- 「SABulkCopyColumnMappingCollection クラス」 184 ページ
- 「SABulkCopyColumnMappingCollection メンバ」 184 ページ

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。

構文

Visual Basic

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

C#

```
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。

参照

- 「SABulkCopyColumnMappingCollection クラス」 184 ページ
- 「SABulkCopyColumnMappingCollection メンバ」 184 ページ

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。

構文

Visual Basic

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

C#

```
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。

参照

- 「SABulkCopyColumnMappingCollection クラス」 184 ページ
- 「SABulkCopyColumnMappingCollection メンバ」 184 ページ

Remove メソッド

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

構文

Visual Basic

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

C#

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

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。

参照

- 「SABulkCopyColumnMappingCollection クラス」 184 ページ
- 「SABulkCopyColumnMappingCollection メンバ」 184 ページ

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。

構文

Visual Basic

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

C#

```
public 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。

参照

- 「SABulkCopyColumnMappingCollection クラス」 184 ページ
- 「SABulkCopyColumnMappingCollection メンバ」 184 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処理フラグです。

構文

Visual Basic

Public Enum **SABulkCopyOptions**

C#

public enum **SABulkCopyOptions**

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。

制限：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。

メンバ

メンバ名	説明	値
デフォルト	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。	0
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。	1
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。	2
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。	4
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。	8

参照

- 「SABulkCopy クラス」 165 ページ

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。このクラスは継承できません。

構文

Visual Basic

```
Public NotInheritable Class SACommand
  Inherits DbCommand
  Implements ICloneable
```

C#

```
public sealed class SACommand : DbCommand,
  ICloneable
```

備考

実装：ICloneable

詳細については、「データのアクセスと操作」 122 ページを参照してください。

参照

- 「SACommand メンバ」 195 ページ

SACommand メンバ

パブリック・コンストラクタ

メンバ名	説明
SACommand コンストラクタ	「SACommand クラス」 195 ページの新しいインスタンスを初期化します。

パブリック・プロパティ

メンバ名	説明
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。
CommandTimeout プロパティ	コマンドを実行しようとするのを中止し、エラーを生成するまでの待機時間 (秒単位) を取得または設定します。
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。

メンバ名	説明
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。デフォルトは true 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。CommandText で疑問符を使用してパラメータを示します。
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。

パブリック・メソッド

メンバ名	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。
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。
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。
ExecuteReader メソッド	結果セットを返す SQL 文を実行します。
ExecuteScalar メソッド	単一の値を返す文を実行します。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。
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。

参照

- 「SACommand クラス」 195 ページ

SACommand コンストラクタ

「SACommand クラス」 195 ページの新しいインスタンスを初期化します。

SACommand() コンストラクタ

SACommand オブジェクトを初期化します。

構文

Visual Basic

```
Public Sub New()
```

C#

```
public  SACommand();
```

参照

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 ページ
- 「SACommand コンストラクタ」 197 ページ

SACommand(String) コンストラクタ

SACommand オブジェクトを初期化します。

構文

Visual Basic

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

C#

```
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。

参照

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 ページ
- 「SACommand コンストラクタ」 197 ページ

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。

構文

Visual Basic

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

C#

```
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。

参照

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 ページ
- 「SACommand コンストラクタ」 197 ページ

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。

構文

Visual Basic

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

C#

```
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

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 ページ
- 「SACommand コンストラクタ」 197 ページ
- 「SATransaction クラス」 437 ページ

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。

構文

Visual Basic

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

C#

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

プロパティ値

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 クラス」 195 ページ
- 「SACommand メンバ」 195 ページ
- 「SACommand() コンストラクタ」 197 ページ

CommandTimeout プロパティ

コマンドを実行しようとするのを中止し、エラーを生成するまでの待機時間 (秒単位) を取得または設定します。

構文

Visual Basic

Public Overrides Property **CommandTimeout** As Integer

C#

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

プロパティ値

デフォルト値は 30 秒です。

備考

値が 0 の場合、制限はありません。値を 0 にすると、コマンドを無期限に実行しようとしてしまうため、値は 0 にしないでください。

参照

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 ページ

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。

構文

Visual Basic

Public Overrides Property **CommandType** As CommandType

C#

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

プロパティ値

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

備考

サポートされているコマンド・タイプは、次のとおりです。

- 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、任意の引数を SAParameter オブジェクトとして指定してください。
- 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。

参照

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 ページ

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。

構文

Visual Basic

Public Property **Connection** As SAConnection

C#

public SAConnection **Connection** { get; set; }

プロパティ値

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

参照

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 ページ

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。デフォルトは true です。

構文

Visual Basic

Public Overrides Property **DesignTimeVisible** As Boolean

C#

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

プロパティ値

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。

参照

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 ページ

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。CommandText で疑問符を使用してパラメータを示します。

構文

Visual Basic

Public Readonly Property **Parameters** As SAParameterCollection

C#

public SAParameterCollection **Parameters** { get;}

プロパティ値

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。

備考

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

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 ページ
- 「SAParameterCollection クラス」 384 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。

構文

Visual Basic

Public Property **Transaction** As SATransaction

C#

public SATransaction **Transaction** { get; set; }

プロパティ値

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

備考

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」 142 ページを参照してください。

参照

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 ページ
- 「SATransaction クラス」 437 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。

構文

Visual Basic

Public Overrides Property **UpdatedRowSource** As UpdateRowSource

C#

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

プロパティ値

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。

備考

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。

参照

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 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。

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。

構文

Visual Basic

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

C#

```
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 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。

参照

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 ページ
- 「BeginExecuteNonQuery メソッド」 204 ページ
- 「EndExecuteNonQuery メソッド」 209 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。

構文

Visual Basic

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

C#

```
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 AsyncCallback 委任。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、ユーザ定義の状態オブジェクト。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、IAsyncResult.AsyncState 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 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。

参照

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 ページ
- 「BeginExecuteNonQuery メソッド」 204 ページ
- 「EndExecuteNonQuery メソッド」 209 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。

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。

構文

Visual Basic

Public Function **BeginExecuteReader()** As IAsyncResult

C#

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 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。

参照

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 ページ
- 「BeginExecuteReader メソッド」 205 ページ

- 「EndExecuteReader メソッド」 211 ページ
- 「SADataReader クラス」 294 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。

構文

Visual Basic

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

C#

```
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 CommandBehavior フラグのビット処理の組み合わせ。

戻り値

ポーリング、結果の待機、または両方に使用できる 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。

参照

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 ページ
- 「BeginExecuteReader メソッド」 205 ページ
- 「EndExecuteReader メソッド」 211 ページ
- 「SADataReader クラス」 294 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 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。

構文

Visual Basic

```
Public Function BeginExecuteReader( _
  ByVal callback As AsyncCallback, _
```

```
  ByVal stateObject As Object _
) As IAsyncResult
```

C#

```
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 AsyncCallback 委任。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、IAsyncResult.AsyncState 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 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。

参照

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 ページ
- 「BeginExecuteReader メソッド」 205 ページ
- 「EndExecuteReader メソッド」 211 ページ
- 「SADataReader クラス」 294 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。

構文

Visual Basic

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

C#

```
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 AsyncCallback 委任。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、ユーザ定義の状態オブジェクト。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、IAsyncResult.AsyncState 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 CommandBehavior フラグのビット処理の組み合わせ。

戻り値

ポーリング、結果の待機、または両方に使用できる 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。

参照

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 ページ
- 「BeginExecuteReader メソッド」 205 ページ
- 「EndExecuteReader メソッド」 211 ページ
- 「SADataReader クラス」 294 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。

構文

Visual Basic

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

C#

```
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ときにキャンセル試行が失敗した場合、例外は生成されません。

参照

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 ページ

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。

構文

Visual Basic

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

C#

```
public SAParameter CreateParameter();
```

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

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 ページ
- 「SAParameter クラス」 370 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。

構文

Visual Basic

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

C#

```
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)。

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 通りの方法があります。

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 10 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 reader = cmd.EndExecuteNonQuery( res );
```

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 10 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 );
```

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 10 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 );
```

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();
}

// elsewhere in the code
private void DoStuff()
{
    SAConnection conn = new SAConnection("DSN=SQL Anywhere 10 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 クラス」 195 ページ
- 「SACommand メンバ」 195 ページ
- 「BeginExecuteNonQuery() メソッド」 204 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。

構文

Visual Basic

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

C#

```
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)。

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 通りの方法があります。

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 10 Demo");
conn.Open();
SACommand cmd = new SACommand( "SELECT * FROM Departments",
  conn );
IAsyncResult res = cmd.BeginExecuteReader();
// perform other work
// this will block until the command completes
SADataReader reader = cmd.EndExecuteReader( res );
```

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

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 10 Demo");
conn.Open();
SACommand cmd = new SACommand( "SELECT * FROM Departments",
  conn );
IAsyncResult res = cmd.BeginExecuteReader();
// perform other work
WaitHandle wh = res.AsyncWaitHandle;
wh.WaitOne();
// this will not block because the command is finished
SADataReader reader = cmd.EndExecuteReader( res );
```

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 wona ?t block since the command has completed
          SADataReader reader = cmd.EndExecuteReader();
}

// elsewhere in the code
private void DoStuff()
{
     SAConnection conn = new SAConnection("DSN=SQL Anywhere 10 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 クラス」 195 ページ
- 「SACommand メンバ」 195 ページ
- 「BeginExecuteReader() メソッド」 205 ページ
- 「SADataReader クラス」 294 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。

構文

Visual Basic

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

C#

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 クラス」 195 ページ
- 「SACommand メンバ」 195 ページ
- 「ExecuteReader() メソッド」 214 ページ

ExecuteReader 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。

構文

Visual Basic

Public Function **ExecuteReader()** As SADataReader

C#

public 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 および 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 クラス」 195 ページ
- 「SACommand メンバ」 195 ページ

- 「ExecuteReader メソッド」 214 ページ
- 「ExecuteNonQuery メソッド」 213 ページ
- 「SADataReader クラス」 294 ページ
- 「SADataAdapter クラス」 283 ページ
- 「CommandText プロパティ」 199 ページ
- 「Parameters プロパティ」 202 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。

構文

Visual Basic

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

C#

```
public 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 クラス」 195 ページ
- 「SACommand メンバ」 195 ページ
- 「ExecuteReader メソッド」 214 ページ
- 「ExecuteNonQuery メソッド」 213 ページ
- 「SADataReader クラス」 294 ページ
- 「SADataAdapter クラス」 283 ページ
- 「CommandText プロパティ」 199 ページ
- 「Parameters プロパティ」 202 ページ

ExecuteScalar メソッド

単一の値を返す文を実行します。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。

構文

Visual Basic

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

C#

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。

参照

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 ページ

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。

構文

Visual Basic

Public Overrides Sub **Prepare()**

C#

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 クラス」 195 ページ
- 「SACommand メンバ」 195 ページ
- 「ExecuteNonQuery メソッド」 213 ページ
- 「ExecuteReader() メソッド」 214 ページ
- 「ExecuteScalar メソッド」 216 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。

構文

Visual Basic

Public Sub **ResetCommandTimeout()**

C#

public void **ResetCommandTimeout();**

参照

- 「SACommand クラス」 195 ページ
- 「SACommand メンバ」 195 ページ

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。このクラスは継承できません。

構文

Visual Basic

```
Public NotInheritable Class SACommandBuilder
  Inherits DbCommandBuilder
```

C#

```
public sealed class SACommandBuilder : DbCommandBuilder
```

参照

- 「SACommandBuilder メンバ」 218 ページ

SACommandBuilder メンバ

パブリック・コンストラクタ

メンバ名	説明
SACommandBuilder コンストラクタ	「SACommandBuilder クラス」 218 ページの新しいインスタンスを初期化します。

パブリック・プロパティ

メンバ名	説明
CatalogLocation (DbCommandBuilder から継承)	
CatalogSeparator (DbCommandBuilder から継承)	
ConflictOption (DbCommandBuilder から継承)	
DataAdapter プロパティ	文を生成する SADataAdapter を指定します。
QuotePrefix (DbCommandBuilder から継承)	

メンバ名	説明
QuoteSuffix (DbCommandBuilder から継承)	
SchemaSeparator (DbCommandBuilder から継承)	
SetAllValues (DbCommandBuilder から継承)	

パブリック・メソッド

メンバ名	説明
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。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。
GetDeleteCommand メソッド	データ・ソースでの削除に必要な自動生成された DbCommand オブジェクトを取得します。
GetInsertCommand メソッド	データ・ソースでの挿入に必要な自動生成された DbCommand オブジェクトを取得します。
GetUpdateCommand メソッド	データ・ソースでの更新に必要な自動生成された DbCommand オブジェクトを取得します。
QuoteIdentifier (DbCommandBuilder から継承)	
RefreshSchema (DbCommandBuilder から継承)	
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。

参照

- 「SACommandBuilder クラス」 218 ページ

SACommandBuilder コンストラクタ

「SACommandBuilder クラス」 218 ページの新しいインスタンスを初期化します。

SACommandBuilder() コンストラクタ

SACommandBuilder オブジェクトを初期化します。

構文

Visual Basic

```
Public Sub New()
```

C#

```
public  SACommandBuilder();
```

参照

- 「SACommandBuilder クラス」 218 ページ
- 「SACommandBuilder メンバ」 218 ページ
- 「SACommandBuilder コンストラクタ」 219 ページ

SACommandBuilder(SADataAdapter) コンストラクタ

SACommandBuilder オブジェクトを初期化します。

構文

Visual Basic

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

C#

```
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。

参照

- 「SACommandBuilder クラス」 218 ページ
- 「SACommandBuilder メンバ」 218 ページ
- 「SACommandBuilder コンストラクタ」 219 ページ

DataAdapter プロパティ

文を生成する SADataAdapter を指定します。

構文

Visual Basic

Public Property **DataAdapter** As SADataAdapter

C#

public SADataAdapter **DataAdapter** { get; set; }

プロパティ値

SADataAdapter オブジェクト。

備考

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。

参照

- 「SACommandBuilder クラス」 218 ページ
- 「SACommandBuilder メンバ」 218 ページ

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。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。

構文

Visual Basic

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

C#

```
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

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。

参照

- 「SACommandBuilder クラス」 218 ページ
- 「SACommandBuilder メンバ」 218 ページ

GetDeleteCommand メソッド

データ・ソースでの削除に必要な自動生成された DbCommand オブジェクトを取得します。

GetDeleteCommand(Boolean) 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。

構文

Visual Basic

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

C#

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

パラメータ

- **useColumnsForParameterNames** true の場合、カラム名に一致するパラメータ名を生成します (可能な場合)。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

- 「SACommandBuilder クラス」 218 ページ
- 「SACommandBuilder メンバ」 218 ページ
- 「GetDeleteCommand メソッド」 222 ページ
- DbCommandBuilder.RefreshSchema

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。

構文

Visual Basic

Public Function **GetDeleteCommand()** As SACommand

C#

public 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

- 「SACommandBuilder クラス」 218 ページ
- 「SACommandBuilder メンバ」 218 ページ
- 「GetDeleteCommand メソッド」 222 ページ
- DbCommandBuilder.RefreshSchema

GetInsertCommand メソッド

データ・ソースでの挿入に必要な自動生成された DbCommand オブジェクトを取得します。

GetInsertCommand(Boolean) 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。

構文

Visual Basic

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

C#

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

パラメータ

- **useColumnsForParameterNames** true の場合、カラム名に一致するパラメータ名を生成します (可能な場合)。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 クラス」 218 ページ
- 「SACommandBuilder メンバ」 218 ページ
- 「GetInsertCommand メソッド」 223 ページ
- 「GetDeleteCommand() メソッド」 223 ページ

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。

構文

Visual Basic

Public Function **GetInsertCommand()** As SACommand

C#

public 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 クラス」 218 ページ
- 「SACommandBuilder メンバ」 218 ページ
- 「GetInsertCommand メソッド」 223 ページ
- 「GetDeleteCommand() メソッド」 223 ページ

GetUpdateCommand メソッド

データ・ソースでの更新に必要な自動生成された DbCommand オブジェクトを取得します。

GetUpdateCommand(Boolean) 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。

構文

Visual Basic

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

C#

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

パラメータ

- **useColumnsForParameterNames** true の場合、カラム名に一致するパラメータ名を生成します (可能な場合)。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

- 「SACommandBuilder クラス」 218 ページ
- 「SACommandBuilder メンバ」 218 ページ
- 「GetUpdateCommand メソッド」 225 ページ
- DbCommandBuilder.RefreshSchema

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。

構文

Visual Basic

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

C#

```
public 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

- 「SACommandBuilder クラス」 218 ページ
- 「SACommandBuilder メンバ」 218 ページ
- 「GetUpdateCommand メソッド」 225 ページ
- DbCommandBuilder.RefreshSchema

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。

構文

Visual Basic

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

C#

```
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。

参照

- 「SACommandBuilder クラス」 218 ページ
- 「SACommandBuilder メンバ」 218 ページ

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。このクラスは継承できません。

構文

Visual Basic

Public NotInheritable Class **SACommLinksOptionsBuilder**

C#

public sealed class **SACommLinksOptionsBuilder**

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。

参照

- 「SACommLinksOptionsBuilder メンバ」 228 ページ

SACommLinksOptionsBuilder メンバ

パブリック・コンストラクタ

メンバ名	説明
SACommLinksOptionsBuilder コンストラクタ	「SACommLinksOptionsBuilder クラス」 228 ページの新しいインスタンスを初期化します。

パブリック・プロパティ

メンバ名	説明
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。
SpxOptionsBuilder プロパティ	SPX オプション文字列の作成に使用する SpxOptionsBuilder オブジェクトを取得または設定します。
SpxOptionsString プロパティ	SPX オプションの文字列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。

メンバ名	説明
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。

パブリック・メソッド

メンバ名	説明
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。デフォルトでは、長い接続パラメータ名が使用されます。
ToString メソッド	SACommLinksOptionsBuilder オブジェクトを文字列表現に変換します。

参照

- 「SACommLinksOptionsBuilder クラス」 228 ページ

SACommLinksOptionsBuilder コンストラクタ

「SACommLinksOptionsBuilder クラス」 228 ページの新しいインスタンスを初期化します。

SACommLinksOptionsBuilder() コンストラクタ

SACommLinksOptionsBuilder オブジェクトを初期化します。

構文

Visual Basic

Public Sub **New()**

C#

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( );
```


参照

- 「SACommLinksOptionsBuilder クラス」 228 ページ
- 「SACommLinksOptionsBuilder メンバ」 228 ページ
- 「SACommLinksOptionsBuilder コンストラクタ」 229 ページ

SACommLinksOptionsBuilder(String) コンストラクタ

SACommLinksOptionsBuilder オブジェクトを初期化します。

構文

Visual Basic

```
Public Sub New( _
  ByVal options As String _
)
```

C#

```
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)" );
```

参照

- 「SACommLinksOptionsBuilder クラス」 228 ページ
- 「SACommLinksOptionsBuilder メンバ」 228 ページ
- 「SACommLinksOptionsBuilder コンストラクタ」 229 ページ

All プロパティ

ALL CommLinks オプションを取得または設定します。

構文

Visual Basic

Public Property **All** As Boolean

C#

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。

参照

- 「SACommLinksOptionsBuilder クラス」 228 ページ
- 「SACommLinksOptionsBuilder メンバ」 228 ページ

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。

構文

Visual Basic

Public Property **ConnectionString** As String

C#

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。

参照

- 「SACommLinksOptionsBuilder クラス」 228 ページ
- 「SACommLinksOptionsBuilder メンバ」 228 ページ

SharedMemory プロパティ

SharedMemory プロトコルを取得または設定します。

構文

Visual Basic

Public Property **SharedMemory** As Boolean

C#

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。

参照

- 「SACommLinksOptionsBuilder クラス」 228 ページ
- 「SACommLinksOptionsBuilder メンバ」 228 ページ

SpxOptionsBuilder プロパティ

SPX オプション文字列の作成に使用する SpxOptionsBuilder オブジェクトを取得または設定します。

構文

Visual Basic

Public Property **SpxOptionsBuilder** As SASpxOptionsBuilder

C#

public SASpxOptionsBuilder **SpxOptionsBuilder** { get; set; }

参照

- 「SACommLinksOptionsBuilder クラス」 228 ページ
- 「SACommLinksOptionsBuilder メンバ」 228 ページ

SpxOptionsString プロパティ

SPX オプションの文字列を取得または設定します。

構文

Visual Basic

Public Property **SpxOptionsString** As String

C#

public string **SpxOptionsString** { get; set; }

参照

- 「SACommLinksOptionsBuilder クラス」 228 ページ
- 「SACommLinksOptionsBuilder メンバ」 228 ページ

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。

構文

Visual Basic

Public Property **TcpOptionsBuilder** As SATcpOptionsBuilder

C#

public SATcpOptionsBuilder **TcpOptionsBuilder** { get; set; }

参照

- 「SACommLinksOptionsBuilder クラス」 228 ページ
- 「SACommLinksOptionsBuilder メンバ」 228 ページ

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。

構文

Visual Basic

Public Property **TcpOptionsString** As String

C#

public string **TcpOptionsString** { get; set; }

参照

- 「SACommLinksOptionsBuilder クラス」 228 ページ
- 「SACommLinksOptionsBuilder メンバ」 228 ページ

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。

構文

Visual Basic

Public Function **GetUseLongNameAsKeyword()** As Boolean

C#

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。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 クラス」 228 ページ
- 「SACommLinksOptionsBuilder メンバ」 228 ページ
- 「SetUseLongNameAsKeyword メソッド」 234 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。デフォルトでは、長い接続パラメータ名が使用されます。

構文

Visual Basic

```
Public Sub SetUseLongNameAsKeyword( _
  ByVal useLongNameAsKeyword As Boolean _
)
```

C#

```
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。

参照

- 「SACommLinksOptionsBuilder クラス」 228 ページ
- 「SACommLinksOptionsBuilder メンバ」 228 ページ
- 「GetUseLongNameAsKeyword メソッド」 233 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。

構文

Visual Basic

Public Overrides Function **ToString()** As String

C#

public override string **ToString();**

戻り値

構築されるオプション文字列。

参照

- 「SACommLinksOptionsBuilder クラス」 228 ページ
- 「SACommLinksOptionsBuilder メンバ」 228 ページ

SAConnection クラス

SQL Anywhere データベースへの接続を表します。このクラスは継承できません。

構文

Visual Basic

```
Public NotInheritable Class SAConnection
  Inherits DbConnection
```

C#

```
public sealed class SAConnection : DbConnection
```

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。

参照

- 「SAConnection メンバ」 236 ページ

SAConnection メンバ

パブリック・コンストラクタ

メンバ名	説明
SAConnection コンストラクタ	「SAConnection クラス」 236 ページの新しいインスタンスを初期化します。

パブリック・プロパティ

メンバ名	説明
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。
DataSource プロパティ	データベース・サーバの名前を取得します。
Database プロパティ	現在のデータベースの名前を取得します。
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。

パブリック・メソッド

メンバ名	説明
BeginTransaction メソッド	データベース・トランザクションを起動します。
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	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。
EnlistTransaction (DbConnection から継承)	
GetSchema メソッド	この DbConnection のデータ・ソースに関するスキーマ情報を返します。
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。

パブリック・イベント

メンバ名	説明
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。

参照

- 「SAConnection クラス」 236 ページ

SAConnection コンストラクタ

「SAConnection クラス」 236 ページの新しいインスタンスを初期化します。

SAConnection() コンストラクタ

SAConnection オブジェクトを初期化します。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。

構文

Visual Basic

```
Public Sub New()
```

C#

```
public  SAConnection();
```

参照

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ
- 「SAConnection コンストラクタ」 237 ページ

SAConnection(String) コンストラクタ

SAConnection オブジェクトを初期化します。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。

構文

Visual Basic

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

C#

```
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。

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 およびパスワード money を使用します。

```
SAConnection conn = new SAConnection(
"UID=admin;PWD=money;ENG=hr;DBN=policies" );
conn.Open();
```

参照

- 「SAConnection クラス」 236 ページ

- 「SAConnection メンバ」 236 ページ
- 「SAConnection コンストラクタ」 237 ページ
- 「SAConnection クラス」 236 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。

構文

Visual Basic

Public Overrides Property **ConnectionString** As String

C#

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。

ダイアログからユーザ 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。

例

次の文は、SQL Anywhere 10 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 10 Demo";
conn.Open();
```

参照

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ
- 「SAConnection クラス」 236 ページ
- 「Open メソッド」 250 ページ

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。

構文

Visual Basic

Public Overrides Readonly Property **ConnectionTimeout** As Integer

C#

public override int **ConnectionTimeout** { get;}

プロパティ値

15 秒

例

次の文は、ConnectionTimeout の値を表示します。

```
MessageBox.Show( conn.ConnectionTimeout.ToString( ) );
```

参照

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ

DataSource プロパティ

データベース・サーバの名前を取得します。

構文

Visual Basic

Public Overrides Readonly Property **DataSource** As String

C#

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 EngineName、ServerName、ENG の順に接続文字列を参照します。

参照

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ
- 「SAConnection クラス」 236 ページ

Database プロパティ

現在のデータベースの名前を取得します。

構文

Visual Basic

Public Overrides Readonly Property **Database** As String

C#

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、DataSource、DSN、DatabaseFile、DBF の順に接続文字列を参照します。

参照

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 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。

構文

Visual Basic

Public Property **InitString** As String

C#

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。

参照

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ

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。

構文

Visual Basic

Public Overrides Readonly Property **ServerVersion** As String

C#

public override string **ServerVersion** { get;}

プロパティ値

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。

参照

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ

State プロパティ

SAConnection オブジェクトのステータスを示します。

構文

Visual Basic

Public Overrides Readonly Property **State** As ConnectionState

C#

public override ConnectionState **State** { get;}

プロパティ値

ConnectionState 列挙。

参照

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ

BeginTransaction メソッド

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。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。

構文

Visual Basic

Public Function **BeginTransaction()** As SATransaction

C#

public 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

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

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ
- 「BeginTransaction メソッド」 243 ページ
- 「SATransaction クラス」 437 ページ
- 「Transaction プロパティ」 202 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。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。

構文

Visual Basic

```
Public Function BeginTransaction( _
   ByVal isolationLevel As IsolationLevel _
) As SATransaction
```

C#

```
public SATransaction BeginTransaction(
   IsolationLevel isolationLevel
);
```

パラメータ

- **isolationLevel** SAIsolationLevel 列挙のメンバ。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

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。

例

```
SATransaction tx = conn.BeginTransaction(
   SAIsolationLevel.ReadUncommitted );
```

参照

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ
- 「BeginTransaction メソッド」 243 ページ
- 「SATransaction クラス」 437 ページ
- 「Transaction プロパティ」 202 ページ
- 「SAIsolationLevel 列挙」 357 ページ

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。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。

構文

Visual Basic

```
Public Function BeginTransaction( _
   ByVal isolationLevel As SAIsolationLevel _
) As SATransaction
```

C#

```
public SATransaction BeginTransaction(
   SAIsolationLevel isolationLevel
);
```

パラメータ

- **isolationLevel** SAIsolationLevel 列挙のメンバ。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。

詳細については、「Transaction 処理」 142 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。

備考

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

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ
- 「BeginTransaction メソッド」 243 ページ
- 「SATransaction クラス」 437 ページ
- 「Transaction プロパティ」 202 ページ
- 「SAIsolationLevel 列挙」 357 ページ
- 「Commit メソッド」 440 ページ
- 「Rollback() メソッド」 440 ページ
- 「Rollback(String) メソッド」 441 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。

構文

Visual Basic

```
Public Overrides Sub ChangeDatabase( _
  ByVal database As String _
)
```

C#

```
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。

参照

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ

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。

構文

Visual Basic

```
Public Shared Sub ChangePassword( _
  ByVal connectionString As String, _
  ByVal newPassword As String _
)
```

C#

```
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。

参照

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ

ClearAllPools メソッド

接続プールを空にします。

構文

Visual Basic

Public Shared Sub **ClearAllPools()**

C#

public static void **ClearAllPools();**

参照

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ

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。

構文

Visual Basic

```
Public Shared Sub ClearPool( _
  ByVal connection As SAConnection _
)
```

C#

```
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」 236 ページ
- 「SAConnection メンバ」 236 ページ
- 「SAConnection クラス」 236 ページ

Close メソッド

データベース接続を閉じます。

構文

Visual Basic

Public Overrides Sub **Close()**

C#

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。

参照

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ

CreateCommand メソッド

SACommand オブジェクトを初期化します。

構文

Visual Basic

Public Function **CreateCommand()** As SACommand

C#

public 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

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ
- 「SACommand クラス」 195 ページ
- 「SAConnection クラス」 236 ページ

GetSchema メソッド

この DbConnection のデータ・ソースに関するスキーマ情報を返します。

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。

構文

Visual Basic

```
Public Overrides Function GetSchema() As DataTable
```

C#

```
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。

参照

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ
- 「GetSchema メソッド」 248 ページ
- 「GetSchema(String, String[]) メソッド」 250 ページ

GetSchema(String) 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。

構文

Visual Basic

```
Public Overrides Function GetSchema( _
  ByVal collection As String _
) As DataTable
```

C#

```
public override DataTable GetSchema(
  string  collection
);
```

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。

参照

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ
- 「GetSchema メソッド」 248 ページ
- 「GetSchema(String, String[]) メソッド」 250 ページ

GetSchema(String, String[]) 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。

構文

Visual Basic

```
Public Overrides Function GetSchema( _
  ByVal collection As String, _
  ByVal restrictions As String() _
) As DataTable
```

C#

```
public override DataTable GetSchema(
  string  collection,
  string [] restrictions
);
```

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。

参照

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ
- 「GetSchema メソッド」 248 ページ
- 「GetSchema(String, String[]) メソッド」 250 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。

構文

Visual Basic

```
Public Overrides Sub Open()
```

C#

```
public override void Open();
```

参照

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ
- 「ConnectionString プロパティ」 239 ページ

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。

構文

Visual Basic

Public Event **InfoMessage** As SAInfoMessageEventHandler

C#

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 SAaInfoMessageEventArgs 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。

イベント・データ

- **MessageType** メッセージのタイプを返します。タイプは Action、Info、Status、Warning のいずれかです。
- **Errors** 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** 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。
- **Source** SQL Anywhere .NET データ・プロバイダの名前を返します。
- **NativeError** データベースによって返される SQL コードを返します。

参照

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ

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。

構文

Visual Basic

Public Overrides Event **StateChange** As StateChangeEventHandler

C#

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。

イベント・データ

- **CurrentState**
- **OriginalState**

参照

- 「SAConnection クラス」 236 ページ
- 「SAConnection メンバ」 236 ページ

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。このクラスは継承できません。

構文

Visual Basic

Public NotInheritable Class **SAConnectionStringBuilder**
  Inherits SAConnectionStringBuilderBase

C#

public sealed class **SAConnectionStringBuilder** : SAConnectionstring BuilderBase

備考

SAConnectionStringBuilder クラスは SAConnectionStringBuilderBase を継承し、SAConnectionStringBuilderBase は DbConnectionStringBuilder を継承しています。

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

継承：「SAConnectionStringBuilderBase クラス」 275 ページ

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。

参照

- 「SAConnectionStringBuilder メンバ」 253 ページ

SAConnectionStringBuilder メンバ

パブリック・コンストラクタ

メンバ名	説明
SAConnectionStringBuilder コンストラクタ	「SAConnectionStringBuilder クラス」 253 ページの新しいインスタンスを初期化します。

パブリック・プロパティ

メンバ名	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。

メンバ名	説明
BrowsableConnectionString (DbConnectionStringBuilder から継承)	
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。
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 (DbConnectionStringBuilder から継承)	
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。
Count (DbConnectionStringBuilder から継承)	
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。
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。
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。
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。

メンバ名	説明
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。
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。
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 (DbConnectionStringBuilder から継承)	
IsReadOnly (DbConnectionStringBuilder から継承)	
Item プロパティ (SAConnectionStringBuilderBase から継承)	接続キーワードの値を取得または設定します。
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 (SAConnectionStringBuilderBase から継承)	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。
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。
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。デフォルト値は 200 です。
RetryConnectionTimeout プロパティ	RetryConnectionTimeout 接続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。

メンバ名	説明
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。
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 (DbConnectionStringBuilder から継承)	

パブリック・メソッド

メンバ名	説明
Add (DbConnectionStringBuilder から継承)	
Clear (DbConnectionStringBuilder から継承)	
ContainsKey メソッド (SAConnectionStringBuilderBase から継承)	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 (DbConnectionStringBuilder から継承)	
GetKeyword メソッド (SAConnectionStringBuilderBase から継承)	指定された SAConnectionStringBuilder プロパティのキーワードを取得します。
GetUseLongNameAsKeyword メソッド (SAConnectionStringBuilderBase から継承)	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 (SAConnectionStringBuilderBase から継承)	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 (SAConnectionStringBuilderBase から継承)	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。デフォルトでは、長い接続パラメータ名が使用されます。
ShouldSerialize メソッド (SAConnectionStringBuilderBase から継承)	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。

メンバ名	説明
ToString (DbConnectionStringBuilder から継承)	
TryGetValue メソッド (SAConnectionStringBuilderBase から継承)	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。

参照

- 「SAConnectionStringBuilder クラス」 253 ページ

SAConnectionStringBuilder コンストラクタ

「SAConnectionStringBuilder クラス」 253 ページ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。

構文

Visual Basic

Public Sub **New()**

C#

public **SAConnectionStringBuilder();**

備考

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

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ
- 「SAConnectionStringBuilder コンストラクタ」 257 ページ

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。

構文

Visual Basic

Public Sub **New(_**


```
  ByVal connectionString As String _
)
```

C#

```
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

制限：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;ENG=hr;DBN=policies" );
```

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ
- 「SAConnectionStringBuilder コンストラクタ」 257 ページ

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。

構文

Visual Basic

```
Public Property AppInfo As String
```

C#

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

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **AutoStart** As String

C#

public string **AutoStart** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **AutoStop** As String

C#

public string **AutoStop** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **Charset** As String

C#

public string **Charset** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **CommBufferSize** As Integer

C#

public int **CommBufferSize** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

CommLinks プロパティ

CommLinks プロパティを取得または設定します。

構文

Visual Basic

Public Property **CommLinks** As String

C#

public string **CommLinks** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **Compress** As String

C#

public string **Compress** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **CompressionThreshold** As Integer

C#

public int **CompressionThreshold** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **ConnectionLifetime** As Integer

C#

public int **ConnectionLifetime** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **ConnectionName** As String

C#

public string **ConnectionName** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **ConnectionReset** As Boolean

C#

public bool **ConnectionReset** { get; set; }

プロパティ値

スキーマ情報が格納されている DataTable。

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **ConnectionTimeout** As Integer

C#

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() );
```

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **DataSourceName** As String

C#

public string **DataSourceName** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **DatabaseFile** As String

C#

public string **DatabaseFile** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **DatabaseKey** As String

C#

public string **DatabaseKey** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **DatabaseName** As String

C#

public string **DatabaseName** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **DatabaseSwitches** As String

C#

public string **DatabaseSwitches** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **DisableMultiRowFetch** As String

C#

public string **DisableMultiRowFetch** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **EncryptedPassword** As String

C#

public string **EncryptedPassword** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **Encryption** As String

C#

public string **Encryption** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **Enlist** As Boolean

C#

public bool **Enlist** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **FileDataSourceName** As String

C#

public string **FileDataSourceName** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **ForceStart** As String

C#

public string **ForceStart** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **IdleTimeout** As Integer

C#

public int **IdleTimeout** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **Integrated** As String

C#

public string **Integrated** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **Kerberos** As String

C#

public string **Kerberos** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **Language** As String

C#

public string **Language** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **LazyClose** As String

C#

public string **LazyClose** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **LivenessTimeout** As Integer

C#

public int **LivenessTimeout** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **LogFile** As String

C#

public string **LogFile** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **MaxPoolSize** As Integer

C#

public int **MaxPoolSize** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **MinPoolSize** As Integer

C#

public int **MinPoolSize** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **Password** As String

C#

public string **Password** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **PersistSecurityInfo** As Boolean

C#

public bool **PersistSecurityInfo** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **Pooling** As Boolean

C#

public bool **Pooling** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **PrefetchBuffer** As Integer

C#

public int **PrefetchBuffer** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。デフォルト値は 200 です。

構文

Visual Basic

Public Property **PrefetchRows** As Integer

C#

public int **PrefetchRows** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

RetryConnectionTimeout プロパティ

RetryConnectionTimeout プロパティを取得または設定します。

構文

Visual Basic

Public Property **RetryConnectionTimeout** As Integer

C#

public int **RetryConnectionTimeout** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **ServerName** As String

C#

public string **ServerName** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **StartLine** As String

C#

public string **StartLine** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **Unconditional** As String

C#

public string **Unconditional** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

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。

構文

Visual Basic

Public Property **UserID** As String

C#

public string **UserID** { get; set; }

参照

- 「SAConnectionStringBuilder クラス」 253 ページ
- 「SAConnectionStringBuilder メンバ」 253 ページ

SAConnectionStringBuilderBase クラス

SAConnectionStringBuilder クラスの基本クラスです。これは抽象クラスであるため、インスタンス化はできません。

構文

Visual Basic

MustInherit Public Class **SAConnectionStringBuilderBase**
  Inherits DbConnectionStringBuilder

C#

public abstract class **SAConnectionStringBuilderBase** : DbConnectionstring Builder

参照

- 「SAConnectionStringBuilderBase メンバ」 275 ページ

SAConnectionStringBuilderBase メンバ

パブリック・プロパティ

メンバ名	説明
BrowsableConnectionString (DbConnectionStringBuilder から継承)	
ConnectionString (DbConnectionStringBuilder から継承)	
Count (DbConnectionStringBuilder から継承)	
IsFixedSize (DbConnectionStringBuilder から継承)	
IsReadOnly (DbConnectionStringBuilder から継承)	
Item 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。
Keys プロパティ	SAConnectionStringBuilder のキーが含まれた System.Collections.ICollection を取得します。

メンバ名	説明
Values (DbConnectionStringBuilder から継承)	

パブリック・メソッド

メンバ名	説明
Add (DbConnectionStringBuilder から継承)	
Clear (DbConnectionStringBuilder から継承)	
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 (DbConnectionStringBuilder から継承)	
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。
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。デフォルトでは、長い接続パラメータ名が使用されます。
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 (DbConnectionStringBuilder から継承)	
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。

参照

- 「SAConnectionStringBuilderBase クラス」 275 ページ

Item 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。

構文

Visual Basic

```
Public Overrides Default Property Item ( _
  ByVal keyword As String _
) As Object
```

C#

```
public override object this [
  string  keyword
] { get; set; }
```

パラメータ

- **keyword**　接続キーワードの名前。

プロパティ値

指定された接続キーワードの値を表すオブジェクト。

備考

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。

参照

- 「SAConnectionStringBuilderBase クラス」 275 ページ
- 「SAConnectionStringBuilderBase メンバ」 275 ページ

Keys プロパティ

SAConnectionStringBuilder のキーが含まれた System.Collections.ICollection を取得します。

構文

Visual Basic

```
Public Overrides Readonly Property Keys As ICollection
```

C#

```
public override ICollection Keys  { get;}
```

プロパティ値

SAConnectionStringBuilder のキーが含まれた System.Collections.ICollection。

参照

- 「SAConnectionStringBuilderBase クラス」 275 ページ
- 「SAConnectionStringBuilderBase メンバ」 275 ページ

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。

構文

Visual Basic

```
Public Overrides Function ContainsKey( _
  ByVal keyword As String _
) As Boolean
```

C#

```
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")
```

参照

- 「SAConnectionStringBuilderBase クラス」 275 ページ
- 「SAConnectionStringBuilderBase メンバ」 275 ページ

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。

構文

Visual Basic

```
Public Function GetKeyword( _
  ByVal propName As String _
) As String
```

C#

```
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。

参照

- 「SAConnectionStringBuilderBase クラス」 275 ページ
- 「SAConnectionStringBuilderBase メンバ」 275 ページ

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。

構文

Visual Basic

```
Public Function GetUseLongNameAsKeyword() As Boolean
```

C#

```
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。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 クラス」 275 ページ
- 「SAConnectionStringBuilderBase メンバ」 275 ページ
- 「SetUseLongNameAsKeyword メソッド」 280 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。

構文

Visual Basic

```
Public Overrides Function Remove( _
  ByVal keyword As String _
) As Boolean
```

C#

```
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。

参照

- 「SAConnectionStringBuilderBase クラス」 275 ページ
- 「SAConnectionStringBuilderBase メンバ」 275 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。デフォルトでは、長い接続パラメータ名が使用されます。

構文

Visual Basic

```
Public Sub SetUseLongNameAsKeyword( _
  ByVal useLongNameAsKeyword As Boolean _
)
```

C#

```
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。

参照

- 「SAConnectionStringBuilderBase クラス」 275 ページ
- 「SAConnectionStringBuilderBase メンバ」 275 ページ
- 「GetUseLongNameAsKeyword メソッド」 279 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。

構文

Visual Basic

```
Public Overrides Function ShouldSerialize( _
  ByVal keyword As String _
) As Boolean
```

C#

```
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。

参照

- 「SAConnectionStringBuilderBase クラス」 275 ページ
- 「SAConnectionStringBuilderBase メンバ」 275 ページ

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。

構文

Visual Basic

```
Public Overrides Function TryGetValue( _
  ByVal keyword As String, _
```



```
  ByVal value As Object _
) As Boolean
```

C#

```
public override bool TryGetValue(
  string  keyword,
  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。

参照

- 「SAConnectionStringBuilderBase クラス」 275 ページ
- 「SAConnectionStringBuilderBase メンバ」 275 ページ

SADataAdapter クラス

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示します。このクラスは継承できません。

構文

Visual Basic

Public NotInheritable Class **SADataAdapter**
  Inherits DbDataAdapter

C#

public sealed class **SADataAdapter** : DbDataAdapter

備考

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。

実装： IDbDataAdapter、IDataAdapter、ICloneable

詳細については、「SADataAdapter オブジェクトを使用したデータのアクセスと操作」 128 ページと「データのアクセスと操作」 122 ページを参照してください。

参照

- 「SADataAdapter メンバ」 283 ページ

SADataAdapter メンバ

パブリック・コンストラクタ

メンバ名	説明
SADataAdapter コンストラクタ	「SADataAdapter クラス」 283 ページの新しいインスタンスを初期化します。

パブリック・プロパティ

メンバ名	説明
AcceptChangesDuringFill (DataAdapter から継承)	
AcceptChangesDuringUpdate (DataAdapter から継承)	
ContinueUpdateOnError (DataAdapter から継承)	

メンバ名	説明
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。
FillLoadOption (DataAdapter から継承)	
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。
MissingMappingAction (DataAdapter から継承)	
MissingSchemaAction (DataAdapter から継承)	
ReturnProviderSpecificTypes (DataAdapter から継承)	
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。

パブリック・メソッド

メンバ名	説明
Fill (DbDataAdapter から継承)	DataSet または DataTable に入力します。
FillSchema (DbDataAdapter から継承)	DataTable を DataSet に追加し、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。
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。
ResetFillLoadOption (DataAdapter から継承)	

メンバ名	説明
ShouldSerializeAcceptChanges DuringFill (DataAdapter から継承)	
ShouldSerializeFillLoadOption (DataAdapter から継承)	
Update (DbDataAdapter から継承)	DataSet 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。

パブリック・イベント

メンバ名	説明
FillError (DataAdapter から継承)	
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。更新が試みられると、イベントが発生します。
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。更新が試みられると、イベントが発生します。

参照

- 「SADataAdapter クラス」 283 ページ

SADataAdapter コンストラクタ

「SADataAdapter クラス」 283 ページの新しいインスタンスを初期化します。

SADataAdapter() コンストラクタ

SADataAdapter オブジェクトを初期化します。

構文

Visual Basic

Public Sub **New()**

C#

public **SADataAdapter();**

参照

- 「SADataAdapter クラス」 283 ページ
- 「SADataAdapter メンバ」 283 ページ

- 「SADataAdapter コンストラクタ」 285 ページ
- 「SADataAdapter(SACommand) コンストラクタ」 286 ページ
- 「SADataAdapter(String, SAConnection) コンストラクタ」 286 ページ
- 「SADataAdapter(String, String) コンストラクタ」 287 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。

構文

Visual Basic

```
Public Sub New( _
  ByVal selectCommand As SACommand _
)
```

C#

```
public  SADataAdapter(
  SACommand selectCommand
);
```

パラメータ

- **selectCommand** 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 DbDataAdapter.Fill の実行時に使用される SACommand オブジェクト。

参照

- 「SADataAdapter クラス」 283 ページ
- 「SADataAdapter メンバ」 283 ページ
- 「SADataAdapter コンストラクタ」 285 ページ
- 「SADataAdapter() コンストラクタ」 285 ページ
- 「SADataAdapter(String, SAConnection) コンストラクタ」 286 ページ
- 「SADataAdapter(String, String) コンストラクタ」 287 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。

構文

Visual Basic

```
Public Sub New( _
  ByVal selectCommandText As String, _
  ByVal selectConnection As SAConnection _
)
```

C#

```
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 クラス」 283 ページ
- 「SADataAdapter メンバ」 283 ページ
- 「SADataAdapter コンストラクタ」 285 ページ
- 「SADataAdapter() コンストラクタ」 285 ページ
- 「SADataAdapter(SACommand) コンストラクタ」 286 ページ
- 「SADataAdapter(String, String) コンストラクタ」 287 ページ
- 「SelectCommand プロパティ」 289 ページ
- 「SAConnection クラス」 236 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。

構文

Visual Basic

```
Public Sub New( _
  ByVal selectCommandText As String, _
  ByVal selectConnectionString As String _
)
```

C#

```
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 クラス」 283 ページ
- 「SADataAdapter メンバ」 283 ページ

- 「SADataAdapter コンストラクタ」 285 ページ
- 「SADataAdapter() コンストラクタ」 285 ページ
- 「SADataAdapter(SACommand) コンストラクタ」 286 ページ
- 「SADataAdapter(String, SAConnection) コンストラクタ」 286 ページ
- 「SelectCommand プロパティ」 289 ページ

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。

構文

Visual Basic

Public Property **DeleteCommand** As SACommand

C#

public 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 クラス」 283 ページ
- 「SADataAdapter メンバ」 283 ページ
- 「SelectCommand プロパティ」 289 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。

構文

Visual Basic

Public Property **InsertCommand** As SACommand

C#

public 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。

参照

- 「SADataAdapter クラス」 283 ページ
- 「SADataAdapter メンバ」 283 ページ

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。

構文

Visual Basic

Public Property **SelectCommand** As SACommand

C#

public 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。

参照

- 「SADataAdapter クラス」 283 ページ
- 「SADataAdapter メンバ」 283 ページ

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。

構文

Visual Basic

Public Readonly Property **TableMappings** As DataTableMappingCollection

C#

public 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。

参照

- 「SADataAdapter クラス」 283 ページ
- 「SADataAdapter メンバ」 283 ページ

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。

構文

Visual Basic

Public Overrides Property **UpdateBatchSize** As Integer

C#

public override int **UpdateBatchSize** { get; set; }

備考

UpdateBatchSize の値を 1 に設定すると、SADataAdapter.Fill は、.NET データ・プロバイダのバージョン 1.0 のようにそのローを送信します。つまり、入力セット内の各ローは一度に 1 つずつ、順番に更新されます。デフォルト値は 1 です。

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的に実行されますが、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。

参照

- 「SADataAdapter クラス」 283 ページ
- 「SADataAdapter メンバ」 283 ページ

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。

構文

Visual Basic

Public Property **UpdateCommand** As SACommand

C#

public 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。

参照

- 「SADataAdapter クラス」 283 ページ
- 「SADataAdapter メンバ」 283 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。

構文

Visual Basic

Public Function **GetFillParameters()** As SAParameter

C#

public 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。

参照

- 「SADataAdapter クラス」 283 ページ
- 「SADataAdapter メンバ」 283 ページ

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。更新が試みられると、イベントが発生します。

構文

Visual Basic

Public Event **RowUpdated** As SARowUpdatedEventHandler

C#

public event SARowUpdatedEventHandler **RowUpdated** ;

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 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。

イベント・データ

- **Command** DataAdapter.Update の呼び出し時に実行する SACommand を取得します。
- **RecordsAffected** 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。
- **Command**
- **Errors**
- **Row**
- **RowCount**

- **StatementType**
- **Status**
- **TableMapping**

参照

- 「SADataAdapter クラス」 283 ページ
- 「SADataAdapter メンバ」 283 ページ

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。更新が試みられると、イベントが発生します。

構文

Visual Basic

Public Event **RowUpdating** As SARowUpdatingEventHandler

C#

public event SARowUpdatingEventHandler **RowUpdating** ;

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 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。

イベント・データ

- **Command** Update の実行時に実行する SACommand を指定します。
- **Command**
- **Errors**
- **Row**
- **StatementType**
- **Status**
- **TableMapping**

参照

- 「SADataAdapter クラス」 283 ページ
- 「SADataAdapter メンバ」 283 ページ

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。このクラスは継承できません。

構文

Visual Basic

```
Public NotInheritable Class SADataReader
  Inherits DbDataReader
  Implements IListSource
```

C#

```
public sealed class SADataReader : DbDataReader,
  IListSource
```

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。

実装：IDataReader、IDisposable、IDataRecord、IListSource

詳細については、「データのアクセスと操作」 122 ページを参照してください。

参照

- 「SADataReader メンバ」 294 ページ
- 「ExecuteReader() メソッド」 214 ページ

SADataReader メンバ

パブリック・プロパティ

メンバ名	説明
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。最も外側のテーブルの深さは 0 です。
FieldCount プロパティ	結果セット内のカラム数を取得する。

メンバ名	説明
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。
Item プロパティ	指定されたカラムの値を Object のインスタンスとして取得します。
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です。
VisibleFieldCount (DbDataReader から継承)	

パブリック・メソッド

メンバ名	説明
Close メソッド	SADataReader を閉じます。
Dispose (DbDataReader から継承)	
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。このメソッドを呼び出すと、InvalidOperationException がスローされます。
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。
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。

メンバ名	説明
GetEnumerator メソッド	SADataReader オブジェクトで反復処理する 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。
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序数を返します。
GetProviderSpecificFieldType (DbDataReader から継承)	
GetProviderSpecificValue (DbDataReader から継承)	
GetProviderSpecificValues (DbDataReader から継承)	
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。
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。
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。
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	指定したカラムの値を 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。

メンバ名	説明
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。
NextResult メソッド	バッチ SQL 文の結果を読み込むときに SADataReader を次の結果に進めます。
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。
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。

参照

- 「SADataReader クラス」 294 ページ
- 「ExecuteReader() メソッド」 214 ページ

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。最も外側のテーブルの深さは 0 です。

構文

Visual Basic

Public Overrides Readonly Property **Depth** As Integer

C#

public override int **Depth** { get;}

プロパティ値

現在のローのネストの深さ。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

FieldCount プロパティ

結果セット内のカラム数を取得する。

構文

Visual Basic

Public Overrides Readonly Property **FieldCount** As Integer

C#

public override int **FieldCount** { get;}

プロパティ値

現在のレコード内のカラム数。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

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。

構文

Visual Basic

Public Overrides Readonly Property **HasRows** As Boolean

C#

public override bool **HasRows** { get;}

プロパティ値

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。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

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。

構文

Visual Basic

Public Overrides Readonly Property **IsClosed** As Boolean

C#

public override bool **IsClosed** { get;}

プロパティ値

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。

備考

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。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

Item プロパティ

指定されたカラムの値を Object のインスタンスとして取得します。

Item(Int32) 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。C# では、このプロパティは SADataReader クラスのインデクサです。

構文

Visual Basic

```
Public Overrides Default Readonly Property Item ( _
  ByVal index As Integer _
) As Object
```

C#

```
public override object this [
  int index
] { get;}
```

パラメータ

- **index**　カラム序数。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ
- 「Item プロパティ」 299 ページ

Item(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。C# では、このプロパティは SADataReader クラスのインデクサです。

構文

Visual Basic

```
Public Overrides Default Readonly Property Item ( _
  ByVal name As String _
) As Object
```


C#

```
public override object this [
   string  name
] { get;}
```

パラメータ

- **name**　カラムの名前。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ
- 「Item プロパティ」 299 ページ

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です。

構文

Visual Basic

```
Public Overrides Readonly Property RecordsAffected As Integer
```

C#

```
public override int RecordsAffected  { get;}
```

プロパティ値

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。

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。

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。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

Close メソッド

SADataReader を閉じます。

構文

Visual Basic

Public Overrides Sub **Close()**

C#

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。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

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。

構文

Visual Basic

```
Public Overrides Function GetBoolean( _
  ByVal ordinal As Integer _
) As Boolean
```

C#

```
public override bool GetBoolean(
  int ordinal
);
```

パラメータ

- **ordinal**　値の取得元のカラムを示す序数。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 クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ
- 「GetOrdinal メソッド」 313 ページ
- 「GetFieldType メソッド」 309 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。

構文

Visual Basic

```
Public Overrides Function GetByte( _
   ByVal ordinal As Integer _
) As Byte
```

C#

```
public override byte GetByte(
   int ordinal
);
```

パラメータ

- **ordinal**　値の取得元のカラムを示す序数。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。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

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。

構文

Visual Basic

```
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#

```
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。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

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。

構文

Visual Basic


```
Public Overrides Function GetChar( _
  ByVal ordinal As Integer _
) As Char
```

C#

```
public override char GetChar(
  int ordinal
);
```

パラメータ

- **ordinal**　値の取得元のカラムを示す序数。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 クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ
- 「IsDBNull メソッド」 322 ページ
- 「IsDBNull メソッド」 322 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。

構文

Visual Basic

```
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#

```
public override long GetChars(
  int ordinal,
  long dataIndex,
  char[] buffer,
```

```
  int bufferIndex,
  int length
);
```

パラメータ

- **ordinal**　0 から始まるカラム序数。
- **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 の処理」 137 ページを参照してください。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

GetData メソッド

このメソッドはサポートされていません。このメソッドを呼び出すと、InvalidOperationException がスローされます。

構文

Visual Basic

```
Public Function GetData( _
  ByVal i As Integer _
) As IDataReader
```

C#

```
public IDataReader GetData(
```

```
  int i
);
```

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ
- InvalidOperationException

GetDataTypeName メソッド

ソース・データ型の名前を返します。

構文

Visual Basic

```
Public Overrides Function GetDataTypeName( _
  ByVal index As Integer _
) As String
```

C#

```
public override string  GetDataTypeName(
  int index
);
```

パラメータ

- **index**　0 から始まるカラム序数。

戻り値

バックエンド・データ型の名前。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

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。

構文

Visual Basic

```
Public Overrides Function GetDateTime( _
  ByVal ordinal As Integer _
) As Date
```

C#

```
public override DateTime GetDateTime(
  int ordinal
);
```

パラメータ

- **ordinal**　0 から始まるカラム序数。

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 クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ
- 「IsDBNull メソッド」 322 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。

構文

Visual Basic

```
Public Overrides Function GetDecimal( _
  ByVal ordinal As Integer _
) As Decimal
```

C#

```
public override decimal GetDecimal(
  int ordinal
);
```

パラメータ

- **ordinal**　値の取得元のカラムを示す序数。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 クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ
- 「IsDBNull メソッド」 322 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。

構文

Visual Basic

```
Public Overrides Function GetDouble( _
   ByVal ordinal As Integer _
) As Double
```

C#

```
public override double GetDouble(
   int ordinal
);
```

パラメータ

- **ordinal**　値の取得元のカラムを示す序数。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 クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ
- 「IsDBNull メソッド」 322 ページ

GetEnumerator メソッド

SADataReader オブジェクトで反復処理する IEnumerator を返します。

構文

Visual Basic

```
Public Overrides Function GetEnumerator() As IEnumerator
```

C#

```
public override IEnumerator GetEnumerator();
```

戻り値

SADataReader オブジェクトの IEnumerator。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ
- 「SADataReader クラス」 294 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。

構文

Visual Basic

```
Public Overrides Function GetFieldType( _
  ByVal index As Integer _
) As Type
```

C#

```
public override Type GetFieldType(
  int index
);
```

パラメータ

- **index**　0 から始まるカラム序数。

戻り値

オブジェクトのデータ型である Type。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

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。

構文

Visual Basic

```
Public Overrides Function GetFloat( _
   ByVal ordinal As Integer _
) As Single
```

C#

```
public override float  GetFloat(
   int ordinal
);
```

パラメータ

- **ordinal**　値の取得元のカラムを示す序数。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 クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ
- 「IsDBNull メソッド」 322 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。

構文

Visual Basic

```
Public Overrides Function GetGuid( _
   ByVal ordinal As Integer _
) As Guid
```

C#

```
public override Guid GetGuid(
   int ordinal
);
```

パラメータ

- **ordinal**　値の取得元のカラムを示す序数。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 クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ
- 「IsDBNull メソッド」 322 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。

構文

Visual Basic

```
Public Overrides Function GetInt16( _
   ByVal ordinal As Integer _
) As Short
```

C#

```
public override short GetInt16(
   int ordinal
);
```

パラメータ

- **ordinal**　値の取得元のカラムを示す序数。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。

参照

- 「SADataReader クラス」 294 ページ

- 「SADataReader メンバ」 294 ページ

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。

構文

Visual Basic

```
Public Overrides Function GetInt32( _
   ByVal ordinal As Integer _
) As Integer
```

C#

```
public override int GetInt32(
   int ordinal
);
```

パラメータ

- **ordinal**　値の取得元のカラムを示す序数。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。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

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。

構文

Visual Basic

```
Public Overrides Function GetInt64( _
   ByVal ordinal As Integer _
) As Long
```

C#

```
public override long GetInt64(
```

```
  int ordinal
);
```

パラメータ

- **ordinal**　値の取得元のカラムを示す序数。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。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

GetName メソッド

指定されたカラムの名前を返します。

構文

Visual Basic

```
Public Overrides Function GetName( _
  ByVal index As Integer _
) As String
```

C#

```
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。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

GetOrdinal メソッド

指定されたカラム名に対応するカラム序数を返します。

構文

Visual Basic

```
Public Overrides Function GetOrdinal( _
   ByVal name As String _
) As Integer
```

C#

```
public override int GetOrdinal(
   string  name
);
```

パラメータ

- **name**　カラムの名前。

戻り値

0 から始まるカラム序数。

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。

序数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。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

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。

構文

Visual Basic

```
Public Overrides Function GetSchemaTable() As DataTable
```

C#

```
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。
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 入力可能である場合は true、null 入力不可能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

DataTable カラム	説明
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。
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、「DataReader スキーマ情報の取得」 128 ページを参照してください。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

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。

構文

Visual Basic

```
Public Overrides Function GetString( _
   ByVal ordinal As Integer _
) As String
```

C#

```
public override string  GetString(
   int ordinal
);
```


パラメータ

- **ordinal**　値の取得元のカラムを示す序数。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 クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ
- 「IsDBNull メソッド」 322 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。

構文

Visual Basic

```
Public Function GetTimeSpan( _
   ByVal ordinal As Integer _
) As TimeSpan
```

C#

```
public TimeSpan GetTimeSpan(
   int ordinal
);
```

パラメータ

- **ordinal**　値の取得元のカラムを示す序数。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、「時間値の取得」 138 ページを参照してください。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ
- 「IsDBNull メソッド」 322 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。

構文

Visual Basic

```
Public Function GetUInt16( _
   ByVal ordinal As Integer _
) As UInt16
```

C#

```
public ushort GetUInt16(
   int ordinal
);
```

パラメータ

- **ordinal**　値の取得元のカラムを示す序数。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。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

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。

構文

Visual Basic

```
Public Function GetUInt32( _
   ByVal ordinal As Integer _
) As UInt32
```

C#

```
public uint GetUInt32(
   int ordinal
);
```

パラメータ

- **ordinal**　値の取得元のカラムを示す序数。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。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

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。

構文

Visual Basic

```
Public Function GetUInt64( _
   ByVal ordinal As Integer _
) As UInt64
```

C#

```
public ulong GetUInt64(
   int ordinal
);
```

パラメータ

- **ordinal**　値の取得元のカラムを示す序数。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。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

GetValue メソッド

指定したカラムの値を Object として返します。

GetValue(Int32) メソッド

指定したカラムの値を Object として返します。

構文

Visual Basic

```
Public Overrides Function GetValue( _
   ByVal ordinal As Integer _
) As Object
```

C#

```
public override object GetValue(
   int ordinal
);
```

パラメータ

- **ordinal**　値の取得元のカラムを示す序数。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したカラムの値。

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。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ
- 「GetValue メソッド」 320 ページ

GetValue(Int32, Int64, Int32) 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。

構文

Visual Basic

```
Public Function GetValue( _
  ByVal ordinal As Integer, _
  ByVal index As Long, _
  ByVal length As Integer _
) As Object
```

C#

```
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。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ
- 「GetValue メソッド」 320 ページ

GetValues メソッド

現在のローのすべてのカラムを取得します。

構文

Visual Basic

```
Public Overrides Function GetValues( _
  ByVal values As Object() _
) As Integer
```

C#

```
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。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

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。

構文

Visual Basic

```
Public Overrides Function IsDBNull( _
   ByVal ordinal As Integer _
) As Boolean
```

C#

```
public override bool IsDBNull(
   int ordinal
);
```

パラメータ

- **ordinal**　0 から始まるカラム序数。

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。そうでない場合、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。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

NextResult メソッド

バッチ SQL 文の結果を読み込むときに SADataReader を次の結果に進めます。

構文

Visual Basic

Public Overrides Function **NextResult()** As Boolean

C#

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 文を実行して生成できる複数の結果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の位置にあります。

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

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。

構文

Visual Basic

Public Overrides Function **Read()** As Boolean

C#

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();
```

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

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。

構文

Visual Basic

Public Sub **myDispose()**

C#

public void **myDispose();**

参照

- 「SADataReader クラス」 294 ページ
- 「SADataReader メンバ」 294 ページ

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。このクラスは継承できません。

構文

Visual Basic

Public NotInheritable Class **SADataSourceEnumerator**
  Inherits DbDataSourceEnumerator

C#

public sealed class **SADataSourceEnumerator** : DbDataSourceEnumerator

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。

参照

- 「SADataSourceEnumerator メンバ」 325 ページ

SADataSourceEnumerator メンバ

パブリック・プロパティ

メンバ名	説明
Instance プロパティ	SADataSourceEnumberator 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。

パブリック・メソッド

メンバ名	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。

参照

- 「SADataSourceEnumerator クラス」 325 ページ

Instance プロパティ

SADataSourceEnumberator 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。

構文

Visual Basic

Public Shared Readonly Property **Instance** As SADataSourceEnumerator

C#

public const SADataSourceEnumerator **Instance** { get;}

参照

- 「SADataSourceEnumerator クラス」 325 ページ
- 「SADataSourceEnumerator メンバ」 325 ページ

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。

構文

Visual Basic

Public Overrides Function **GetDataSources()** As DataTable

C#

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();
```

参照

- 「SADataSourceEnumerator クラス」 325 ページ
- 「SADataSourceEnumerator メンバ」 325 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。

構文

Visual Basic

Public Enum **SADbType**

C#

public enum **SADbType**

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)	Date
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()

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
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
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
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.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。

メンバ

メンバ名	説明	値
BigInt	符号付き 64 ビット整数値	1
Binary	指定した最大長のバイナリ・データ。列挙値 Binary と VarBinary は互いのエイリアスです。	2
Bit	1 ビット・フラグ	3
Char	指定した長さの文字データ。このタイプは常にユニコード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。	4
Date	日付情報	5
DateTime	タイムスタンプ情報 (日付、時刻)。列挙値 DateTime と TimeStamp は互いのエイリアスです。	6
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 は互いのエイリアスです。	7
Double	倍精度浮動小数点数 (8 バイト)	8
Float	単精度浮動小数点数 (4 バイト)。列挙値 Float と Real は互いのエイリアスです。	9
Image	任意の長さのバイナリ・データを格納します。	10
Integer	符号なし 32 ビット整数値	11
LongBinary	可変長のバイナリ・データ	12
LongNVarchar	NCHAR 文字セットの可変長文字データ。このタイプは常にユニコード文字をサポートします。	13
LongVarbit	可変長のビット配列	14
LongVarchar	可変長の文字データ。このタイプは常にユニコード文字をサポートします。	15
Money	通貨データ	16
NChar	8191 文字までのユニコード文字データを格納します。	17
NText	任意の長さのユニコード文字データを格納します。	18

メンバ名	説明	値
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 は互いのエイリアスです。	19
NVarChar	8191 文字までのユニコード文字データを格納します。	20
Real	単精度浮動小数点数 (4 バイト)。列挙値 Float と Real は互いのエイリアスです。	21
SmallDateTime	TIMESTAMP として実装されたドメイン	22
SmallInt	符号付き 16 ビット整数値	23
SmallMoney	100 万通貨単位未満の通貨データを格納します。	24
SysName	任意の長さの文字データを格納します。	25
Text	任意の長さの文字データを格納します。	26
Time	時刻情報	27
TimeStamp	タイムスタンプ情報 (日付、時刻)。列挙値 DateTime と TimeStamp は互いのエイリアスです。	28
TinyInt	符号なし 8 ビット整数値	29
UniqueIdentifier	ユニバーサル・ユニーク識別子 (UUID/GUID)	30
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。	31
UnsignedBigInt	符号なし 64 ビット整数値	32
UnsignedInt	符号なし 32 ビット整数値	33
UnsignedSmallInt	符号なし 16 ビット整数値	34
VarBinary	指定した最大長のバイナリ・データ。列挙値 **Binary** と **VarBinary** は互いのエイリアスです。	35
VarBit	長さが 1 ー 32767 ビットのビット配列	36
VarChar	指定した最大長の文字データ。このタイプは常にユニコード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。	37

メンバ名	説明	値
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。	38

参照

- 「GetFieldType メソッド」 309 ページ
- 「GetDataTypeName メソッド」 306 ページ

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。これは静的クラスであるため、継承またはインスタンス化はできません。

構文

Visual Basic

Public NotInheritable Class **SADefault**

C#

public sealed class **SADefault**

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;
```

参照

- 「SADefault メンバ」 332 ページ

SADefault メンバ

パブリック・フィールド

メンバ名	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。このフィールドは読み込み専用で静的です。このフィールドは読み込み専用です。

参照

- 「SADefault クラス」 332 ページ

Value フィールド

デフォルト・パラメータの値を取得します。このフィールドは読み込み専用で静的です。このフィールドは読み込み専用です。

構文

Visual Basic

Public Shared Readonly **Value** As SADefault

C#

public const SADefault **Value** ;

参照

- 「SADefault クラス」 332 ページ
- 「SADefault メンバ」 332 ページ

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。このクラスは継承できません。

構文

Visual Basic

Public NotInheritable Class **SAError**

C#

public sealed class **SAError**

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、「エラー処理と SQL Anywhere .NET データ・プロバイダ」 144 ページを参照してください。

参照

- 「SAError メンバ」 334 ページ

SAError メンバ

パブリック・プロパティ

メンバ名	説明
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。

パブリック・メソッド

メンバ名	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。

参照

- 「SAError クラス」 334 ページ

Message プロパティ

エラーの簡単な説明を返します。

構文

Visual Basic

Public Readonly Property **Message** As String

C#

public string **Message** { get;}

参照

- 「SAError クラス」 334 ページ
- 「SAError メンバ」 334 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。

構文

Visual Basic

Public Readonly Property **NativeError** As Integer

C#

public int **NativeError** { get;}

参照

- 「SAError クラス」 334 ページ
- 「SAError メンバ」 334 ページ

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。

構文

Visual Basic

Public Readonly Property **Source** As String

C#

public string **Source** { get;}

参照

- 「SAError クラス」 334 ページ
- 「SAError メンバ」 334 ページ

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。

構文

Visual Basic

Public Readonly Property **SqlState** As String

C#

public string **SqlState** { get;}

参照

- 「SAError クラス」 334 ページ
- 「SAError メンバ」 334 ページ

ToString メソッド

エラー・メッセージの完全なテキストです。

構文

Visual Basic

Public Overrides Function **ToString()** As String

C#

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.
```

参照

- 「SAError クラス」 334 ページ
- 「SAError メンバ」 334 ページ

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。このクラスは継承できません。

構文

Visual Basic

```
Public NotInheritable Class SAErrorCollection
  Implements ICollection, IEnumerable
```

C#

```
public sealed class SAErrorCollection  : ICollection, IEnumerable
```

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。

実装： ICollection、IEnumerable

エラー処理の詳細については、「エラー処理と SQL Anywhere .NET データ・プロバイダ」 144 ページを参照してください。

参照

- 「SAErrorCollection メンバ」 337 ページ
- 「Errors プロパティ」 342 ページ
- SqlClientFactory.CanCreateDataSourceEnumerator

SAErrorCollection メンバ

パブリック・プロパティ

メンバ名	説明
Count プロパティ	コレクション内のエラーの数を返します。
Item 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。

パブリック・メソッド

メンバ名	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。

参照

- 「SAErrorCollection クラス」 337 ページ
- 「Errors プロパティ」 342 ページ
- SqlClientFactory.CanCreateDataSourceEnumerator

Count プロパティ

コレクション内のエラーの数を返します。

構文

Visual Basic

```
NotOverridable Public Readonly Property Count As Integer
```

C#

```
public int Count  { get;}
```

参照

- 「SAErrorCollection クラス」 337 ページ
- 「SAErrorCollection メンバ」 337 ページ

Item 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。

構文

Visual Basic

```
Public Readonly Property Item ( _
   ByVal index As Integer _
) As SAError
```

C#

```
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。

プロパティ値

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

- 「SAErrorCollection クラス」 337 ページ

- 「SAErrorCollection メンバ」 337 ページ
- 「SAError クラス」 334 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。

構文

Visual Basic

```
NotOverridable Public Sub CopyTo( _
  ByVal array As Array, _
  ByVal index As Integer _
)
```

C#

```
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。

参照

- 「SAErrorCollection クラス」 337 ページ
- 「SAErrorCollection メンバ」 337 ページ

GetEnumerator メソッド

SAErrorCollection で反復処理する列挙子を返します。

構文

Visual Basic

```
NotOverridable Public Function GetEnumerator() As IEnumerator
```

C#

```
public IEnumerator GetEnumerator();
```

戻り値

SAErrorCollection の IEnumerator。

参照

- 「SAErrorCollection クラス」 337 ページ
- 「SAErrorCollection メンバ」 337 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。

構文

Visual Basic

Public Class **SAException**
  Inherits DbException

C#

public class **SAException** : DbException

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、「エラー処理と SQL Anywhere .NET データ・プロバイダ」 144 ページを参照してください。

参照

- 「SAException メンバ」 341 ページ

SAException メンバ

パブリック・プロパティ

メンバ名	説明
Data (Exception から継承)	
ErrorCode (ExternalException から継承)	
Errors プロパティ	1 つまたは複数の「SAError クラス」 334 ページオブジェクトのコレクションを返します。
HelpLink (Exception から継承)	
InnerException (Exception から継承)	
Message プロパティ	エラーが記述されたテキストを返します。

メンバ名	説明
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 (Exception から継承)	
TargetSite (Exception から継承)	

パブリック・メソッド

メンバ名	説明
GetBaseException (Exception から継承)	
GetObjectData メソッド	SerializationInfo に例外に関する情報を設定します。Exception.GetObjectData を上書きします。
GetType (Exception から継承)	
ToString (Exception から継承)	

参照

- 「SAException クラス」 341 ページ

Errors プロパティ

1 つまたは複数の「SAError クラス」 334 ページオブジェクトのコレクションを返します。

構文

Visual Basic

```
Public Readonly Property Errors As SAErrorCollection
```

C#

```
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

- 「SAException クラス」 341 ページ
- 「SAException メンバ」 341 ページ

- 「SAErrorCollection クラス」 337 ページ
- 「SAError クラス」 334 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。

構文

Visual Basic

Public Overrides Readonly Property **Message** As String

C#

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

- 「SAException クラス」 341 ページ
- 「SAException メンバ」 341 ページ
- 「SAError クラス」 334 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。

構文

Visual Basic

Public Readonly Property **NativeError** As Integer

C#

public int **NativeError** { get;}

参照

- 「SAException クラス」 341 ページ
- 「SAException メンバ」 341 ページ

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。

構文

Visual Basic

```
Public Overrides Readonly Property Source As String
```

C#

```
public override string  Source  { get;}
```

参照

- 「SAException クラス」 341 ページ
- 「SAException メンバ」 341 ページ

GetObjectData メソッド

SerializationInfo に例外に関する情報を設定します。Exception.GetObjectData を上書きします。

構文

Visual Basic

```
Public Overrides Sub GetObjectData( _
  ByVal info As SerializationInfo, _
  ByVal context As StreamingContext _
)
```

C#

```
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。

参照

- 「SAException クラス」 341 ページ
- 「SAException メンバ」 341 ページ

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示します。これは静的クラスであるため、継承またはインスタンス化はできません。

構文

Visual Basic

Public NotInheritable Class **SAFactory**
  Inherits DbProviderFactory

C#

public sealed class **SAFactory** : DbProviderFactory

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();
```

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、http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs05/html/vsgenerics.asp を参照してください。

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

継承：DbProviderFactory

参照

- 「SAFactory メンバ」 346 ページ

SAFactory メンバ

パブリック・フィールド

メンバ名	説明
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。このフィールドは読み込み専用です。

パブリック・プロパティ

メンバ名	説明
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。

パブリック・メソッド

メンバ名	説明
CreateCommand メソッド	厳密に型指定された 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 DbCommandBuilder インスタンスを返します。
CreateConnection メソッド	厳密に型指定された 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 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 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 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 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。

参照

- 「SAFactory クラス」 345 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。このフィールドは読み込み専用です。

構文

Visual Basic

Public Shared Readonly **Instance** As SAFactory

C#

public const 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、DbProviderFactories.GetFactory を使用して SAFactory のこのインスタンスを参照します。例については、SAFactory の説明を参照してください。

制限：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」 345 ページ
- 「SAFactory メンバ」 346 ページ
- 「SAFactory クラス」 345 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。

構文

Visual Basic

Public Overrides Readonly Property **CanCreateDataSourceEnumerator** As Boolean

C#

public override bool **CanCreateDataSourceEnumerator** { get;}

プロパティ値

DbCommand として型指定された新しい SACommand オブジェクト。

参照

- 「SAFactory クラス」 345 ページ
- 「SAFactory メンバ」 346 ページ
- 「SADataSourceEnumerator クラス」 325 ページ
- 「SACommand クラス」 195 ページ

CreateCommand メソッド

厳密に型指定された DbCommand インスタンスを返します。

構文

Visual Basic

Public Overrides Function **CreateCommand()** As DbCommand

C#

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

- 「SAFactory クラス」 345 ページ
- 「SAFactory メンバ」 346 ページ
- 「SACommand クラス」 195 ページ

CreateCommandBuilder メソッド

厳密に型指定された DbCommandBuilder インスタンスを返します。

構文

Visual Basic

Public Overrides Function **CreateCommandBuilder()** As DbCommandBuilder

C#

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

- 「SAFactory クラス」 345 ページ
- 「SAFactory メンバ」 346 ページ
- 「SACommand クラス」 195 ページ

CreateConnection メソッド

厳密に型指定された DbConnection インスタンスを返します。

構文

Visual Basic

Public Overrides Function **CreateConnection()** As DbConnection

C#

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

- 「SAFactory クラス」 345 ページ
- 「SAFactory メンバ」 346 ページ
- 「SACommand クラス」 195 ページ

CreateConnectionStringBuilder メソッド

厳密に型指定された DbConnectionStringBuilder インスタンスを返します。

構文

Visual Basic

Public Overrides Function **CreateConnectionStringBuilder()** As DbConnectionStringBuilder

C#

public override DbConnectionstring Builder **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

- 「SAFactory クラス」 345 ページ
- 「SAFactory メンバ」 346 ページ
- 「SACommand クラス」 195 ページ

CreateDataAdapter メソッド

厳密に型指定された DbDataAdapter インスタンスを返します。

構文

Visual Basic

Public Overrides Function **CreateDataAdapter()** As DbDataAdapter

C#

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

- 「SAFactory クラス」 345 ページ
- 「SAFactory メンバ」 346 ページ
- 「SACommand クラス」 195 ページ

CreateDataSourceEnumerator メソッド

厳密に型指定された DbDataSourceEnumerator インスタンスを返します。

構文

Visual Basic

Public Overrides Function **CreateDataSourceEnumerator()** As DbDataSourceEnumerator

C#

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

- 「SAFactory クラス」 345 ページ
- 「SAFactory メンバ」 346 ページ
- 「SACommand クラス」 195 ページ

CreateParameter メソッド

厳密に型指定された DbParameter インスタンスを返します。

構文

Visual Basic

Public Overrides Function **CreateParameter()** As DbParameter

C#

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

- 「SAFactory クラス」 345 ページ
- 「SAFactory メンバ」 346 ページ
- 「SACommand クラス」 195 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。

構文

Visual Basic

```
Public Overrides Function CreatePermission( _
   ByVal state As PermissionState _
) As CodeAccessPermission
```

C#

```
public override CodeAccessPermission CreatePermission(
   PermissionState state
);
```

パラメータ

- **state**　PermissionState 列挙のメンバ。

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

- 「SAFactory クラス」 345 ページ
- 「SAFactory メンバ」 346 ページ
- 「SACommand クラス」 195 ページ

SAInfoMessageEventArgs クラス

InfoMessage イベントのデータを提供します。このクラスは継承できません。

構文

Visual Basic

Public NotInheritable Class **SAInfoMessageEventArgs**
  Inherits EventArgs

C#

public sealed class **SAInfoMessageEventArgs** : EventArgs

備考

SAInfoMessageEventArgs にはコンストラクタがありません。

参照

- 「SAInfoMessageEventArgs メンバ」 352 ページ

SAInfoMessageEventArgs メンバ

パブリック・プロパティ

メンバ名	説明
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。タイプは Action、Info、Status、Warning のいずれかです。
NativeError プロパティ	データベースによって返される SQL コードを返します。
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。

パブリック・メソッド

メンバ名	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。

参照

- 「SAInfoMessageEventArgs クラス」 352 ページ

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。

構文

Visual Basic

Public Readonly Property **Errors** As SAErrorCollection

C#

public SAErrorCollection **Errors** { get;}

参照

- 「SAInfoMessageEventArgs クラス」 352 ページ
- 「SAInfoMessageEventArgs メンバ」 352 ページ

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。

構文

Visual Basic

Public Readonly Property **Message** As String

C#

public string **Message** { get;}

参照

- 「SAInfoMessageEventArgs クラス」 352 ページ
- 「SAInfoMessageEventArgs メンバ」 352 ページ

MessageType プロパティ

メッセージのタイプを返します。タイプは Action、Info、Status、Warning のいずれかです。

構文

Visual Basic

Public Readonly Property **MessageType** As SAMessageType

C#

public SAMessageType **MessageType** { get;}

参照

- 「SAInfoMessageEventArgs クラス」 352 ページ
- 「SAInfoMessageEventArgs メンバ」 352 ページ

NativeError プロパティ

データベースによって返される SQL コードを返します。

構文

Visual Basic

Public Readonly Property **NativeError** As Integer

C#

public int **NativeError** { get;}

参照

- 「SAInfoMessageEventArgs クラス」 352 ページ
- 「SAInfoMessageEventArgs メンバ」 352 ページ

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。

構文

Visual Basic

Public Readonly Property **Source** As String

C#

public string **Source** { get;}

参照

- 「SAInfoMessageEventArgs クラス」 352 ページ
- 「SAInfoMessageEventArgs メンバ」 352 ページ

ToString メソッド

InfoMessage イベントの文字列表現を取り出します。

構文

Visual Basic

Public Overrides Function **ToString()** As String

C#

public override string **ToString();**

戻り値

InfoMessage イベントを示す文字列。

参照

- 「SAInfoMessageEventArgs クラス」 352 ページ
- 「SAInfoMessageEventArgs メンバ」 352 ページ

SAInfoMessageEventHandler 委任

SAConnection オブジェクトの SAConnection.InfoMessage イベントを処理するメソッドを示します。

構文

Visual Basic

```
Public Delegate Sub SAInfoMessageEventHandler( _
  ByVal obj As Object, _
  ByVal args As SAInfoMessageEventArgs _
)
```

C#

```
public delegate void SAInfoMessageEventHandler(
  object obj,
  SAInfoMessageEventArgs args
);
```

参照

- 「SAConnection クラス」 236 ページ
- 「InfoMessage イベント」 251 ページ

SAIsolationLevel 列挙

SQL Anywhere の独立性レベルを指定します。このクラスの引数は IsolationLevel です。

構文

Visual Basic

Public Enum **SAIsolationLevel**

C#

public enum **SAIsolationLevel**

備考

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。

メンバ

メンバ名	説明	値
Chaos	この独立性レベルはサポートされていません。	16
ReadCommitted	独立性レベル 1 と同等な動作を設定します。	4096
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。	16777217
ReadUncommitted	独立性レベル 0 と同等な動作を設定します。	256
RepeatableRead	独立性レベル 2 と同等な動作を設定します。	65536
Serializable	独立性レベル 3 と同等な動作を設定します。	1048576
Snapshot	トランザクションが最初のローの読み込み、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。	16777216

メンバ名	説明	値
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。	16777218
Unspecified	この独立性レベルはサポートされていません。	-1

SAMessageType 列挙

メッセージのタイプを示します。タイプは Action、Info、Status、Warning のいずれかです。

構文

Visual Basic

Public Enum **SAMessageType**

C#

public enum **SAMessageType**

メンバ

メンバ名	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

SAMetaDataCollectionNames クラス

メタデータ・コレクションを取得する SAConnection.GetSchema(String,String[]) メソッドで使用する定数のリストを提供します。このクラスは継承できません。

構文

Visual Basic

Public NotInheritable Class **SAMetaDataCollectionNames**

C#

public sealed class **SAMetaDataCollectionNames**

備考

このフィールドは定数で、読み込み専用です。

参照

- 「SAMetaDataCollectionNames メンバ」 360 ページ
- 「GetSchema(String, String[]) メソッド」 250 ページ

SAMetaDataCollectionNames メンバ

パブリック・フィールド

メンバ名	説明
Columns フィールド	Columns コレクションを示す SAConnection.GetSchema (String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。
DataSourceInformation フィールド	DataSourceInformation コレクションを示す SAConnection.GetSchema(String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。
DataTypes フィールド	DataTypes コレクションを示す SAConnection.GetSchema (String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。
ForeignKeys フィールド	ForeignKeys コレクションを示す SAConnection.GetSchema (String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。
IndexColumns フィールド	IndexColumns コレクションを示す SAConnection.GetSchema (String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。
Indexes フィールド	Indexes コレクションを示す SAConnection.GetSchema (String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。

メンバ名	説明
MetaDataCollections フィールド	MetaDataCollections コレクションを示す SAConnection.GetSchema(String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。
ProcedureParameters フィールド	ProcedureParameters コレクションを示す SAConnection.GetSchema(String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。
Procedures フィールド	Procedures コレクションを示す SAConnection.GetSchema (String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。
ReservedWords フィールド	ReservedWords コレクションを示す SAConnection.GetSchema (String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。
Restrictions フィールド	Restrictions コレクションを示す SAConnection.GetSchema (String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。
Tables フィールド	Tables コレクションを示す SAConnection.GetSchema (String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。
UserDefinedTypes フィールド	UserDefinedTypes コレクションを示す SAConnection.GetSchema (String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。
Users フィールド	Users コレクションを示す SAConnection.GetSchema (String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。
ViewColumns フィールド	ViewColumns コレクションを示す SAConnection.GetSchema (String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。
Views フィールド	Views コレクションを示す SAConnection.GetSchema (String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。

参照

- 「SAMetaDataCollectionNames クラス」 360 ページ
- 「GetSchema(String, String[]) メソッド」 250 ページ

Columns フィールド

Columns コレクションを示す SAConnection.GetSchema(String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。

構文

Visual Basic

Public Shared Readonly **Columns** As String

C#

public const string **Columns** ;

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 );
```

参照

- 「SAMetaDataCollectionNames クラス」 360 ページ
- 「SAMetaDataCollectionNames メンバ」 360 ページ
- 「GetSchema(String, String[]) メソッド」 250 ページ

DataSourceInformation フィールド

DataSourceInformation コレクションを示す SAConnection.GetSchema(String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。

構文

Visual Basic

Public Shared Readonly **DataSourceInformation** As String

C#

public const string **DataSourceInformation** ;

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 );
```

参照

- 「SAMetaDataCollectionNames クラス」 360 ページ
- 「SAMetaDataCollectionNames メンバ」 360 ページ
- 「GetSchema(String, String[]) メソッド」 250 ページ

DataTypes フィールド

DataTypes コレクションを示す SAConnection.GetSchema(String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。

構文

Visual Basic

Public Shared Readonly **DataTypes** As String

C#

public const string **DataTypes** ;

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 );
```

参照

- 「SAMetaDataCollectionNames クラス」 360 ページ
- 「SAMetaDataCollectionNames メンバ」 360 ページ
- 「GetSchema(String, String[]) メソッド」 250 ページ

ForeignKeys フィールド

ForeignKeys コレクションを示す SAConnection.GetSchema(String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。

構文

Visual Basic

Public Shared Readonly **ForeignKeys** As String

C#

public const string **ForeignKeys** ;

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 );
```

参照

- 「SAMetaDataCollectionNames クラス」 360 ページ
- 「SAMetaDataCollectionNames メンバ」 360 ページ
- 「GetSchema(String, String[]) メソッド」 250 ページ

IndexColumns フィールド

IndexColumns コレクションを示す SAConnection.GetSchema(String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。

構文

Visual Basic

Public Shared Readonly **IndexColumns** As String

C#

public const string **IndexColumns** ;

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 );
```

参照

- 「SAMetaDataCollectionNames クラス」 360 ページ
- 「SAMetaDataCollectionNames メンバ」 360 ページ
- 「GetSchema(String, String[]) メソッド」 250 ページ

Indexes フィールド

Indexes コレクションを示す SAConnection.GetSchema(String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。

構文

Visual Basic

Public Shared Readonly **Indexes** As String

C#

public const string **Indexes** ;

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 );
```

参照

- 「SAMetaDataCollectionNames クラス」 360 ページ
- 「SAMetaDataCollectionNames メンバ」 360 ページ
- 「GetSchema(String, String[]) メソッド」 250 ページ

MetaDataCollections フィールド

MetaDataCollections コレクションを示す SAConnection.GetSchema(String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。

構文

Visual Basic

Public Shared Readonly **MetaDataCollections** As String

C#

public const string **MetaDataCollections** ;

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 );
```

参照

- 「SAMetaDataCollectionNames クラス」 360 ページ
- 「SAMetaDataCollectionNames メンバ」 360 ページ
- 「GetSchema(String, String[]) メソッド」 250 ページ

ProcedureParameters フィールド

ProcedureParameters コレクションを示す SAConnection.GetSchema(String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。

構文

Visual Basic

Public Shared Readonly **ProcedureParameters** As String

C#

public const string **ProcedureParameters** ;

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 );
```

参照

- 「SAMetaDataCollectionNames クラス」 360 ページ
- 「SAMetaDataCollectionNames メンバ」 360 ページ
- 「GetSchema(String, String[]) メソッド」 250 ページ

Procedures フィールド

Procedures コレクションを示す SAConnection.GetSchema(String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。

構文

Visual Basic

Public Shared Readonly **Procedures** As String

C#

public const string **Procedures** ;

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 );
```

参照

- 「SAMetaDataCollectionNames クラス」 360 ページ
- 「SAMetaDataCollectionNames メンバ」 360 ページ
- 「GetSchema(String, String[]) メソッド」 250 ページ

ReservedWords フィールド

ReservedWords コレクションを示す SAConnection.GetSchema(String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。

構文

Visual Basic

Public Shared Readonly **ReservedWords** As String

C#

public const string **ReservedWords** ;

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 );
```

参照

- 「SAMetaDataCollectionNames クラス」 360 ページ
- 「SAMetaDataCollectionNames メンバ」 360 ページ
- 「GetSchema(String, String[]) メソッド」 250 ページ

Restrictions フィールド

Restrictions コレクションを示す SAConnection.GetSchema(String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。

構文

Visual Basic

Public Shared Readonly **Restrictions** As String

C#

public const string **Restrictions** ;

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 );
```

参照

- 「SAMetaDataCollectionNames クラス」 360 ページ
- 「SAMetaDataCollectionNames メンバ」 360 ページ
- 「GetSchema(String, String[]) メソッド」 250 ページ

Tables フィールド

Tables コレクションを示す SAConnection.GetSchema(String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。

構文

Visual Basic

Public Shared Readonly **Tables** As String

C#

public const string **Tables** ;

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 );
```

参照

- 「SAMetaDataCollectionNames クラス」 360 ページ
- 「SAMetaDataCollectionNames メンバ」 360 ページ
- 「GetSchema(String, String[]) メソッド」 250 ページ

UserDefinedTypes フィールド

UserDefinedTypes コレクションを示す SAConnection.GetSchema(String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。

構文

Visual Basic

Public Shared Readonly **UserDefinedTypes** As String

C#

public const string **UserDefinedTypes** ;

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.UserDefinedTypes );
```

参照

- 「SAMetaDataCollectionNames クラス」 360 ページ
- 「SAMetaDataCollectionNames メンバ」 360 ページ
- 「GetSchema(String, String[]) メソッド」 250 ページ

Users フィールド

Users コレクションを示す SAConnection.GetSchema(String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。

構文

Visual Basic

Public Shared Readonly **Users** As String

C#

public const string **Users** ;

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 );
```

参照

- 「SAMetaDataCollectionNames クラス」 360 ページ
- 「SAMetaDataCollectionNames メンバ」 360 ページ
- 「GetSchema(String, String[]) メソッド」 250 ページ

ViewColumns フィールド

ViewColumns コレクションを示す SAConnection.GetSchema(String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。

構文

Visual Basic

Public Shared Readonly **ViewColumns** As String

C#

public const string **ViewColumns** ;

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 );
```

参照

- 「SAMetaDataCollectionNames クラス」 360 ページ
- 「SAMetaDataCollectionNames メンバ」 360 ページ
- 「GetSchema(String, String[]) メソッド」 250 ページ

Views フィールド

Views コレクションを示す SAConnection.GetSchema(String,String[]) メソッドで使用する定数を提供します。このフィールドは読み込み専用です。

構文

Visual Basic

Public Shared Readonly **Views** As String

C#

public const string **Views** ;

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 );
```

参照

- 「SAMetaDataCollectionNames クラス」 360 ページ
- 「SAMetaDataCollectionNames メンバ」 360 ページ
- 「GetSchema(String, String[]) メソッド」 250 ページ

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示します。このクラスは継承できません。

構文

Visual Basic

```
Public NotInheritable Class SAParameter
  Inherits DbParameter
  Implements ICloneable
```

C#

```
public sealed class SAParameter : DbParameter,
  ICloneable
```

備考

実装： IDbDataParameter、IDataParameter、ICloneable

参照

- 「SAParameter メンバ」 370 ページ

SAParameter メンバ

パブリック・コンストラクタ

メンバ名	説明
SAParameter コンストラクタ	「SAParameter クラス」 370 ページの新しいインスタンスを初期化します。

パブリック・プロパティ

メンバ名	説明
DbType プロパティ	パラメータの DbType を取得または設定します。
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。
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。

メンバ名	説明
Precision プロパティ	Value プロパティを示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 入力可能かどうかを示す値を取得または設定します。これによって、SACommandBuilder は、null 入力可能なカラムに対して適切に Update 文を生成できます。
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。

パブリック・メソッド

メンバ名	説明
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。

参照

- 「SAParameter クラス」 370 ページ

SAParameter コンストラクタ

「SAParameter クラス」 370 ページの新しいインスタンスを初期化します。

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。

構文

Visual Basic

Public Sub **New()**

C#

```
public SAParameter();
```

参照

- 「SAParameter クラス」 370 ページ
- 「SAParameter メンバ」 370 ページ
- 「SAParameter コンストラクタ」 371 ページ

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。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。

構文

Visual Basic

```
Public Sub New( _
  ByVal parameterName As String, _
  ByVal value As Object _
)
```

C#

```
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。

参照

- 「SAParameter クラス」 370 ページ
- 「SAParameter メンバ」 370 ページ
- 「SAParameter コンストラクタ」 371 ページ

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とデータ型で初期化します。

構文

Visual Basic

```
Public Sub New( _
  ByVal parameterName As String, _
```

```
  ByVal dbType As SADbType _
)
```

C#

```
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 クラス」 370 ページ
- 「SAParameter メンバ」 370 ページ
- 「SAParameter コンストラクタ」 371 ページ
- 「SADbType プロパティ」 379 ページ

SAParameter(String, SADbType, Int32) 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。

構文

Visual Basic

```
Public Sub New( _
  ByVal parameterName As String, _
  ByVal dbType As SADbType, _
  ByVal size As Integer _
)
```

C#

```
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。

参照

- 「SAParameter クラス」 370 ページ

- 「SAParameter メンバ」 370 ページ
- 「SAParameter コンストラクタ」 371 ページ

SAParameter(String, SADbType, Int32, String) 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、長さとソース・カラムで初期化します。

構文

Visual Basic

```
Public Sub New( _
  ByVal parameterName As String, _
  ByVal dbType As SADbType, _
  ByVal size As Integer, _
  ByVal sourceColumn As String _
)
```

C#

```
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。

参照

- 「SAParameter クラス」 370 ページ
- 「SAParameter メンバ」 370 ページ
- 「SAParameter コンストラクタ」 371 ページ

SAParameter(String, SADbType, Int32, ParameterDirection, Boolean, Byte, Byte, String, DataRowVersion, Object) コンストラクタ

指定したパラメータ名、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。

構文

Visual Basic

```
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#

```
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。

参照

- 「SAParameter クラス」 370 ページ
- 「SAParameter メンバ」 370 ページ

- 「SAParameter コンストラクタ」 371 ページ

DbType プロパティ

パラメータの DbType を取得または設定します。

構文

Visual Basic

Public Overrides Property **DbType** As DbType

C#

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 列挙のメンバにする必要があります。

参照

- 「SAParameter クラス」 370 ページ
- 「SAParameter メンバ」 370 ページ

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。

構文

Visual Basic

Public Overrides Property **Direction** As ParameterDirection

C#

public override ParameterDirection **Direction** { get; set; }

プロパティ値

ParameterDirection 値の 1 つ。

備考

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。

参照

- 「SAParameter クラス」 370 ページ
- 「SAParameter メンバ」 370 ページ

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。

構文

Visual Basic

Public Overrides Property **IsNullable** As Boolean

C#

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。

参照

- 「SAParameter クラス」 370 ページ
- 「SAParameter メンバ」 370 ページ

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。

構文

Visual Basic

Public Property **Offset** As Integer

C#

public int **Offset** { get; set; }

プロパティ値

値のオフセット。デフォルトは 0 です。

参照

- 「SAParameter クラス」 370 ページ
- 「SAParameter メンバ」 370 ページ

ParameterName プロパティ

SAParameter の名前を取得または設定します。

構文

Visual Basic

Public Overrides Property **ParameterName** As String

C#

public override string **ParameterName** { get; set; }

プロパティ値

デフォルトは、空の文字列です。

備考

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。

参照

- 「SAParameter クラス」 370 ページ
- 「SAParameter メンバ」 370 ページ

Precision プロパティ

Value プロパティを示すために使用される最大桁数を取得または設定します。

構文

Visual Basic

Public Property **Precision** As Byte

C#

public byte **Precision** { get; set; }

プロパティ値

このプロパティの値は、Value プロパティを示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。

備考

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。

参照

- 「SAParameter クラス」 370 ページ
- 「SAParameter メンバ」 370 ページ

SADbType プロパティ

パラメータの SADbType です。

構文

Visual Basic

Public Property **SADbType** As SADbType

C#

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 列挙のメンバにする必要があります。

参照

- 「SAParameter クラス」 370 ページ
- 「SAParameter メンバ」 370 ページ

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。

構文

Visual Basic

Public Property **Scale** As Byte

C#

public byte **Scale** { get; set; }

プロパティ値

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。

備考

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。

参照

- 「SAParameter クラス」 370 ページ
- 「SAParameter メンバ」 370 ページ

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。

構文

Visual Basic

Public Overrides Property **Size** As Integer

C#

public override int **Size** { get; set; }

プロパティ値

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。

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。

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。

参照

- 「SAParameter クラス」 370 ページ
- 「SAParameter メンバ」 370 ページ

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。

構文

Visual Basic

Public Overrides Property **SourceColumn** As String

C#

public override string **SourceColumn** { get; set; }

プロパティ値

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。

備考

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。

参照

- 「SAParameter クラス」 370 ページ
- 「SAParameter メンバ」 370 ページ

SourceColumnNullMapping プロパティ

ソース・カラムが null 入力可能かどうかを示す値を取得または設定します。これによって、SACommandBuilder は、null 入力可能なカラムに対して適切に Update 文を生成できます。

構文

Visual Basic

Public Overrides Property **SourceColumnNullMapping** As Boolean

C#

public override bool **SourceColumnNullMapping** { get; set; }

備考

ソース・カラムが null 入力可能な場合は true、null 入力可能でない場合は false が返されます。

参照

- 「SAParameter クラス」 370 ページ
- 「SAParameter メンバ」 370 ページ

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。

構文

Visual Basic

Public Overrides Property **SourceVersion** As DataRowVersion

C#

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。

参照

- 「SAParameter クラス」 370 ページ
- 「SAParameter メンバ」 370 ページ

Value プロパティ

パラメータの値を取得または設定します。

構文

Visual Basic

Public Overrides Property **Value** As Object

C#

public override object **Value** { get; set; }

プロパティ値

パラメータの値を指定する Object。

備考

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 値を示します。

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。

参照

- 「SAParameter クラス」 370 ページ
- 「SAParameter メンバ」 370 ページ

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。

構文

Visual Basic

Public Overrides Sub **ResetDbType()**

C#

public override void **ResetDbType();**

参照

- 「SAParameter クラス」 370 ページ
- 「SAParameter メンバ」 370 ページ

ToString メソッド

ParameterName が含まれる文字列を返します。

構文

Visual Basic

Public Overrides Function **ToString()** As String

C#

public override string **ToString();**

戻り値

パラメータの名前。

参照

- 「SAParameter クラス」 370 ページ
- 「SAParameter メンバ」 370 ページ

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示します。このクラスは継承できません。

構文

Visual Basic

```
Public NotInheritable Class SAParameterCollection
  Inherits DbParameterCollection
```

C#

```
public sealed class SAParameterCollection : DbParameterCollection
```

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

- 「SAParameterCollection メンバ」 384 ページ
- 「SACommand クラス」 195 ページ
- 「Parameters プロパティ」 202 ページ
- 「SAParameter クラス」 370 ページ
- 「SAParameterCollection クラス」 384 ページ

SAParameterCollection メンバ

パブリック・プロパティ

メンバ名	説明
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。
Item プロパティ	コレクション内の DbParameter を取得または設定します。
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。

パブリック・メソッド

メンバ名	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。
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	特定のプロパティが設定された DbParameter がコレクション内に存在す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。
IndexOf メソッド	指定された DbParameter オブジェクトのインデックスを返します。
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 DbParameter オブジェクトをコレクションから削除します。

参照

- 「SAParameterCollection クラス」 384 ページ
- 「SACommand クラス」 195 ページ
- 「Parameters プロパティ」 202 ページ
- 「SAParameter クラス」 370 ページ
- 「SAParameterCollection クラス」 384 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。

構文

Visual Basic

Public Overrides Readonly Property **Count** As Integer

C#

public override int **Count** { get;}

プロパティ値

コレクション内の SAParameter オブジェクトの数。

参照

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ
- 「SAParameter クラス」 370 ページ
- 「SAParameterCollection クラス」 384 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。

構文

Visual Basic

Public Overrides Readonly Property **IsFixedSize** As Boolean

C#

public override bool **IsFixedSize** { get;}

プロパティ値

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。

参照

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ

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。

構文

Visual Basic

Public Overrides Readonly Property **IsReadOnly** As Boolean

C#

public override bool **IsReadOnly** { get;}

プロパティ値

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。

参照

- 「SAParameterCollection クラス」 384 ページ

- 「SAParameterCollection メンバ」 384 ページ

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。

構文

Visual Basic

```
Public Overrides Readonly Property IsSynchronized As Boolean
```

C#

```
public override bool IsSynchronized  { get;}
```

プロパティ値

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。

参照

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ

Item プロパティ

コレクション内の DbParameter を取得または設定します。

Item(Int32) 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設定します。

構文

Visual Basic

```
Public Property Item ( _
   ByVal index As Integer _
) As SAParameter
```

C#

```
public 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。

プロパティ値

指定されたインデックス位置の SAParameter。

備考

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

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ
- 「Item プロパティ」 387 ページ
- 「SAParameter クラス」 370 ページ
- 「SAParameterCollection クラス」 384 ページ

Item(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設定します。

構文

Visual Basic

```
Public Property Item ( _
   ByVal parameterName As String _
) As SAParameter
```

C#

```
public SAParameter this [
   string  parameterName
] { get; set; }
```

パラメータ

- **parameterName**　取り出すパラメータの名前。

プロパティ値

指定された名前の SAParameter オブジェクト。

備考

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

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ
- 「Item プロパティ」 387 ページ
- 「SAParameter クラス」 370 ページ
- 「SAParameterCollection クラス」 384 ページ
- 「Item(Int32) プロパティ」 299 ページ
- 「GetOrdinal メソッド」 313 ページ

- 「GetValue(Int32) メソッド」 320 ページ
- 「GetFieldType メソッド」 309 ページ

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。

構文

Visual Basic

```
Public Overrides Readonly Property SyncRoot As Object
```

C#

```
public override object SyncRoot  { get;}
```

参照

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 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。

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。

構文

Visual Basic

```
Public Overrides Function Add( _
   ByVal value As Object _
) As Integer
```

C#

```
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

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ
- 「Add メソッド」 389 ページ
- 「SAParameter クラス」 370 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。

構文

Visual Basic

```
Public Function Add( _
   ByVal value As SAParameter _
) As SAParameter
```

C#

```
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。

参照

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ
- 「Add メソッド」 389 ページ

Add(String, Object) メソッド

コレクションに対して指定し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。

構文

Visual Basic

```
Public Function Add( _
   ByVal parameterName As String, _
   ByVal value As Object _
) As SAParameter
```

C#

```
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

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ
- 「Add メソッド」 389 ページ
- 「SAParameter クラス」 370 ページ

Add(String, SADbType) メソッド

コレクションに対して指定し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。

構文

Visual Basic

```
Public Function Add( _
   ByVal parameterName As String, _
   ByVal saDbType As SADbType _
) As SAParameter
```

C#

```
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

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ
- 「Add メソッド」 389 ページ
- 「SADbType 列挙」 327 ページ
- 「Add(SAParameter) メソッド」 390 ページ
- 「Add(String, Object) メソッド」 390 ページ

Add(String, SADbType, Int32) メソッド

コレクションに対して指定したパラメータ名、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。

構文

Visual Basic

```
Public Function Add( _
   ByVal parameterName As String, _
   ByVal saDbType As SADbType, _
   ByVal size As Integer _
) As SAParameter
```

C#

```
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

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ
- 「Add メソッド」 389 ページ
- 「SADbType 列挙」 327 ページ
- 「Add(SAParameter) メソッド」 390 ページ

- 「Add(String, Object) メソッド」 390 ページ

Add(String, SADbType, Int32, String) メソッド

コレクションに対して指定したパラメータ名、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。

構文

Visual Basic

```
Public Function Add( _
   ByVal parameterName As String, _
   ByVal saDbType As SADbType, _
   ByVal size As Integer, _
   ByVal sourceColumn As String _
) As SAParameter
```

C#

```
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

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ
- 「Add メソッド」 389 ページ
- 「SADbType 列挙」 327 ページ
- 「Add(SAParameter) メソッド」 390 ページ
- 「Add(String, Object) メソッド」 390 ページ

AddRange 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。

構文

Visual Basic

```
Public Overrides Sub AddRange( _
  ByVal values As Array _
)
```

C#

```
public override void AddRange(
  Array values
);
```

パラメータ

- **values** 追加する値。

参照

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ
- 「AddRange メソッド」 393 ページ

AddRange(SAParameter[]) メソッド

SAParameterCollection の末尾に値の配列を追加します。

構文

Visual Basic

```
Public Sub AddRange( _
  ByVal values As SAParameter() _
)
```

C#

```
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。

参照

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ
- 「AddRange メソッド」 393 ページ

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。

構文

Visual Basic

```
Public Overrides Sub Clear()
```

C#

```
public override void Clear();
```

参照

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ

Contains メソッド

特定のプロパティが設定された DbParameter がコレクション内に存在す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。

構文

Visual Basic

```
Public Overrides Function Contains( _
   ByVal value As Object _
) As Boolean
```

C#

```
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

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ
- 「Contains メソッド」 395 ページ

- 「SAParameter クラス」 370 ページ
- 「Contains(String) メソッド」 396 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。

構文

Visual Basic

```
Public Overrides Function Contains( _
   ByVal value As String _
) As Boolean
```

C#

```
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

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ
- 「Contains メソッド」 395 ページ
- 「SAParameter クラス」 370 ページ
- 「Contains(Object) メソッド」 395 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。

構文

Visual Basic

```
Public Overrides Sub CopyTo( _
   ByVal array As Array, _
   ByVal index As Integer _
)
```

C#

```
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

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ
- 「SAParameter クラス」 370 ページ
- 「SAParameterCollection クラス」 384 ページ

GetEnumerator メソッド

SAParameterCollection で反復処理する列挙子を返します。

構文

Visual Basic

Public Overrides Function **GetEnumerator()** As IEnumerator

C#

public override IEnumerator **GetEnumerator();**

戻り値

SAParameterCollection オブジェクトの IEnumerator。

参照

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ
- 「SAParameterCollection クラス」 384 ページ

IndexOf メソッド

指定された DbParameter オブジェクトのインデックスを返します。

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。

構文

Visual Basic

```
Public Overrides Function IndexOf( _
   ByVal value As Object _
) As Integer
```

C#

```
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

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ
- 「IndexOf メソッド」 397 ページ
- 「SAParameter クラス」 370 ページ
- 「IndexOf(String) メソッド」 398 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。

構文

Visual Basic

```
Public Overrides Function IndexOf( _
   ByVal parameterName As String _
) As Integer
```

C#

```
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

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ
- 「IndexOf メソッド」 397 ページ
- 「SAParameter クラス」 370 ページ
- 「IndexOf(Object) メソッド」 397 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。

構文

Visual Basic

```
Public Overrides Sub Insert( _
  ByVal index As Integer, _
  ByVal value As Object _
)
```

C#

```
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。

参照

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ

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。

構文

Visual Basic

```
Public Overrides Sub Remove( _
  ByVal value As Object _
)
```

C#

```
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。

参照

- 「SAParameterCollection クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ

RemoveAt メソッド

指定された DbParameter オブジェクトをコレクションから削除します。

RemoveAt(Int32) 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。

構文

Visual Basic

```
Public Overrides Sub RemoveAt( _
  ByVal index As Integer _
)
```

C#

```
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 クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ
- 「RemoveAt メソッド」 400 ページ
- 「RemoveAt(String) メソッド」 400 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。

構文

Visual Basic

```
Public Overrides Sub RemoveAt( _
  ByVal parameterName As String _
)
```

C#

```
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 クラス」 384 ページ
- 「SAParameterCollection メンバ」 384 ページ
- 「RemoveAt メソッド」 400 ページ
- 「RemoveAt(Int32) メソッド」 400 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。このクラスは継承できません。

構文

Visual Basic

```
Public NotInheritable Class SAPermission
  Inherits DBDataPermission
```

C#

```
public sealed class SAPermission : DBDataPermission
```

備考

基本クラス DBDataPermission

参照

- 「SAPermission メンバ」 402 ページ

SAPermission メンバ

パブリック・コンストラクタ

メンバ名	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。

パブリック・プロパティ

メンバ名	説明
AllowBlankPassword (DBDataPermission から継承)	

パブリック・メソッド

メンバ名	説明
Add (DBDataPermission から継承)	
Assert (CodeAccessPermission から継承)	
Copy (DBDataPermission から継承)	

メンバ名	説明
Demand (CodeAccessPermission から継承)	
Deny (CodeAccessPermission から継承)	
Equals (CodeAccessPermission から継承)	
FromXml (DBDataPermission から継承)	
GetHashCode (CodeAccessPermission から継承)	
Intersect (DBDataPermission から継承)	
IsSubsetOf (DBDataPermission から継承)	
IsUnrestricted (DBDataPermission から継承)	
PermitOnly (CodeAccessPermission から継承)	
ToString (CodeAccessPermission から継承)	
ToXml (DBDataPermission から継承)	
Union (DBDataPermission から継承)	

参照

- 「SAPermission クラス」 402 ページ

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。

構文

Visual Basic


```
Public Sub New( _
  ByVal state As PermissionState _
)
```

C#

```
public  SAPermission(
  PermissionState state
);
```

パラメータ

- **state**　PermissionState 値の 1 つ。

参照

- 「SAPermission クラス」 402 ページ
- 「SAPermission メンバ」 402 ページ

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。このクラスは継承できません。

構文

Visual Basic

```
Public NotInheritable Class SAPermissionAttribute
  Inherits DBDataPermissionAttribute
```

C#

```
public sealed class SAPermissionAttribute : DBDataPermissionAttribute
```

参照

- 「SAPermissionAttribute メンバ」 405 ページ

SAPermissionAttribute メンバ

パブリック・コンストラクタ

メンバ名	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。

パブリック・プロパティ

メンバ名	説明
Action (SecurityAttribute から継承)	
AllowBlankPassword (DBDataPermissionAttribute から継承)	
ConnectionString (DBDataPermissionAttribute から継承)	
KeyRestrictionBehavior (DBDataPermissionAttribute から継承)	
KeyRestrictions (DBDataPermissionAttribute から継承)	

メンバ名	説明
TypeId (Attribute から継承)	
Unrestricted (SecurityAttribute から継承)	

パブリック・メソッド

メンバ名	説明
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。
Equals (Attribute から継承)	
GetHashCode (Attribute から継承)	
IsDefaultAttribute (Attribute から継承)	
Match (Attribute から継承)	
ShouldSerializeConnectionString (DBDataPermissionAttribute から継承)	
ShouldSerializeKeyRestrictions (DBDataPermissionAttribute から継承)	

参照

- 「SAPermissionAttribute クラス」 405 ページ

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。

構文

Visual Basic

```
Public Sub New( _
   ByVal action As SecurityAction _
)
```

C#

```
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示す SecurityAction 値の 1 つ。

参照

- 「SAPermissionAttribute クラス」 405 ページ
- 「SAPermissionAttribute メンバ」 405 ページ

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。

構文

Visual Basic

Public Overrides Function **CreatePermission()** As IPermission

C#

public override IPermission **CreatePermission();**

参照

- 「SAPermissionAttribute クラス」 405 ページ
- 「SAPermissionAttribute メンバ」 405 ページ

SARowsCopiedEventArgs クラス

SARowsCopiedEventHandler に渡される引数のセットを示します。このクラスは継承できません。

構文

Visual Basic

Public NotInheritable Class **SARowsCopiedEventArgs**

C#

public sealed class **SARowsCopiedEventArgs**

備考

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

参照

- 「SARowsCopiedEventArgs メンバ」 408 ページ

SARowsCopiedEventArgs メンバ

パブリック・コンストラクタ

メンバ名	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。

パブリック・プロパティ

メンバ名	説明
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。

参照

- 「SARowsCopiedEventArgs クラス」 408 ページ

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。

構文

Visual Basic

```
Public Sub New( _
  ByVal rowsCopied As Long _
)
```

C#

```
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

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

参照

- 「SARowsCopiedEventArgs クラス」 408 ページ
- 「SARowsCopiedEventArgs メンバ」 408 ページ

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。

構文

Visual Basic

```
Public Property Abort As Boolean
```

C#

```
public bool Abort  { get; set; }
```

備考

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

参照

- 「SARowsCopiedEventArgs クラス」 408 ページ
- 「SARowsCopiedEventArgs メンバ」 408 ページ

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。

構文

Visual Basic

Public Readonly Property **RowsCopied** As Long

C#

public long **RowsCopied** { get;}

備考

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

参照

- 「SARowsCopiedEventArgs クラス」 408 ページ
- 「SARowsCopiedEventArgs メンバ」 408 ページ

SARowsCopiedEventHandler 委任

SABulkCopy の SABulkCopy.SARowsCopied イベントを処理するメソッドを示します。

構文

Visual Basic

```
Public Delegate Sub SARowsCopiedEventHandler( _
  ByVal sender As Object, _
  ByVal rowsCopiedEventArgs As SARowsCopiedEventArgs _
)
```

C#

```
public delegate void SARowsCopiedEventHandler(
  object sender,
  SARowsCopiedEventArgs rowsCopiedEventArgs
);
```

備考

制限：SARowsCopiedEventHandler 委任は、.NET Compact Framework 2.0 では使用できません。

参照

- 「SABulkCopy クラス」 165 ページ

SARowUpdatedEventArgs クラス

RowUpdated イベントのデータを提供します。このクラスは継承できません。

構文

Visual Basic

```
Public NotInheritable Class SARowUpdatedEventArgs
  Inherits RowUpdatedEventArgs
```

C#

```
public sealed class SARowUpdatedEventArgs : RowUpdatedEventArgs
```

参照

- 「SARowUpdatedEventArgs メンバ」 412 ページ

SARowUpdatedEventArgs メンバ

パブリック・コンストラクタ

メンバ名	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。

パブリック・プロパティ

メンバ名	説明
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 (RowUpdatedEventArgs から継承)	
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 (RowUpdatedEventArgs から継承)	
RowCount (RowUpdatedEventArgs から継承)	
StatementType (RowUpdatedEventArgs から継承)	

メンバ名	説明
Status (RowUpdatedEventArgs から継承)	
TableMapping (RowUpdatedEventArgs から継承)	

パブリック・メソッド

メンバ名	説明
CopyToRows (RowUpdatedEventArgs から継承)	バッチ更新操作の実行中に処理されるローにアクセスできます。

参照

- 「SARowUpdatedEventArgs クラス」 412 ページ

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。

構文

Visual Basic

```
Public Sub New( _
  ByVal row As DataRow, _
  ByVal command As IDbCommand, _
  ByVal statementType As StatementType, _
  ByVal tableMapping As DataTableMapping _
)
```

C#

```
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。

参照

- 「SARowUpdatedEventArgs クラス」 412 ページ
- 「SARowUpdatedEventArgs メンバ」 412 ページ

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。

構文

Visual Basic

Public Readonly Property **Command** As SACommand

C#

public SACommand **Command** { get;}

参照

- 「SARowUpdatedEventArgs クラス」 412 ページ
- 「SARowUpdatedEventArgs メンバ」 412 ページ

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。

構文

Visual Basic

Public Readonly Property **RecordsAffected** As Integer

C#

public int **RecordsAffected** { get;}

プロパティ値

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。

参照

- 「SARowUpdatedEventArgs クラス」 412 ページ
- 「SARowUpdatedEventArgs メンバ」 412 ページ

SARowUpdatedEventHandler 委任

SADataAdapter の RowUpdated イベントを処理するメソッドを示します。

構文

Visual Basic

```
Public Delegate Sub SARowUpdatedEventHandler( _
   ByVal sender As Object, _
   ByVal e As SARowUpdatedEventArgs _
)
```

C#

```
public delegate void SARowUpdatedEventHandler(
   object sender,
   SARowUpdatedEventArgs e
);
```

SARowUpdatingEventArgs クラス

RowUpdating イベントのデータを提供します。このクラスは継承できません。

構文

Visual Basic

Public NotInheritable Class **SARowUpdatingEventArgs**
  Inherits RowUpdatingEventArgs

C#

public sealed class **SARowUpdatingEventArgs** : RowUpdatingEventArgs

参照

- 「SARowUpdatingEventArgs メンバ」 416 ページ

SARowUpdatingEventArgs メンバ

パブリック・コンストラクタ

メンバ名	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。

パブリック・プロパティ

メンバ名	説明
Command プロパティ	Update の実行時に実行する SACommand を指定します。
Errors (RowUpdatingEventArgs から継承)	
Row (RowUpdatingEventArgs から継承)	
StatementType (RowUpdatingEventArgs から継承)	
Status (RowUpdatingEventArgs から継承)	
TableMapping (RowUpdatingEventArgs から継承)	

参照

- 「SARowUpdatingEventArgs クラス」 416 ページ

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。

構文

Visual Basic

```
Public Sub New( _
  ByVal row As DataRow, _
  ByVal command As IDbCommand, _
  ByVal statementType As StatementType, _
  ByVal tableMapping As DataTableMapping _
)
```

C#

```
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。

参照

- 「SARowUpdatingEventArgs クラス」 416 ページ
- 「SARowUpdatingEventArgs メンバ」 416 ページ

Command プロパティ

Update の実行時に実行する SACommand を指定します。

構文

Visual Basic

```
Public Property Command As SACommand
```

C#

public SACommand **Command** { get; set; }

参照

- 「SARowUpdatingEventArgs クラス」 416 ページ
- 「SARowUpdatingEventArgs メンバ」 416 ページ

SARowUpdatingEventHandler 委任

SADataAdapter の RowUpdating イベントを処理するメソッドを示します。

構文

Visual Basic

```
Public Delegate Sub SARowUpdatingEventHandler( _
   ByVal sender As Object, _
   ByVal e As SARowUpdatingEventArgs _
)
```

C#

```
public delegate void SARowUpdatingEventHandler(
   object sender,
   SARowUpdatingEventArgs e
);
```

SASpxOptionsBuilder クラス

SAConnection オブジェクトが使用する接続文字列の、SPX 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。このクラスは継承できません。

構文

Visual Basic

Public NotInheritable Class **SASpxOptionsBuilder**
  Inherits SAConnectionStringBuilderBase

C#

public sealed class **SASpxOptionsBuilder** : SAConnectionstring BuilderBase

参照

- 「SASpxOptionsBuilder メンバ」 420 ページ
- 「SAConnection クラス」 236 ページ

SASpxOptionsBuilder メンバ

パブリック・コンストラクタ

メンバ名	説明
SASpxOptionsBuilder コンストラクタ	「SASpxOptionsBuilder クラス」 420 ページの新しいインスタンスを初期化します。

パブリック・プロパティ

メンバ名	説明
BrowsableConnectionString (DbConnectionStringBuilder から継承)	
ConnectionString (DbConnectionStringBuilder から継承)	
Count (DbConnectionStringBuilder から継承)	
DLL プロパティ	DLL オプションを取得または設定します。
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。

メンバ名	説明
IsFixedSize (DbConnectionStringBuilder から継承)	
IsReadOnly (DbConnectionStringBuilder から継承)	
Item プロパティ (SAConnectionStringBuilderBase から継承)	接続キーワードの値を取得または設定します。
Keys プロパティ (SAConnectionStringBuilderBase から継承)	SAConnectionStringBuilder のキーが含まれた System.Collections.ICollection を取得します。
SearchBindery プロパティ	SearchBindery オプション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 (DbConnectionStringBuilder から継承)	

パブリック・メソッド

メンバ名	説明
Add (DbConnectionStringBuilder から継承)	
Clear (DbConnectionStringBuilder から継承)	
ContainsKey メソッド (SAConnectionStringBuilderBase から継承)	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 (DbConnectionStringBuilder から継承)	
GetKeyword メソッド (SAConnectionStringBuilderBase から継承)	指定された SAConnectionStringBuilder プロパティのキーワードを取得します。
GetUseLongNameAsKeyword メソッド (SAConnectionStringBuilderBase から継承)	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。

メンバ名	説明
Remove メソッド (SAConnectionStringBuilderBase から継承)	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 (SAConnectionStringBuilderBase から継承)	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。デフォルトでは、長い接続パラメータ名が使用されます。
ShouldSerialize メソッド (SAConnectionStringBuilderBase から継承)	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	SpxOptionsBuilder オブジェクトを文字列表現に変換します。
TryGetValue メソッド (SAConnectionStringBuilderBase から継承)	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。

参照

- 「SASpxOptionsBuilder クラス」 420 ページ
- 「SAConnection クラス」 236 ページ

SASpxOptionsBuilder コンストラクタ

「SASpxOptionsBuilder クラス」 420 ページの新しいインスタンスを初期化します。

SASpxOptionsBuilder() コンストラクタ

SASpxOptionsBuilder オブジェクトを初期化します。

構文

Visual Basic

Public Sub **New()**

C#

public **SASpxOptionsBuilder();**

例

次の文は、SASpxOptionsBuilder オブジェクトを初期化します。

```
SASpxOptionsBuilder options = new SASpxOptionsBuilder( );
```

参照

- 「SASpxOptionsBuilder クラス」 420 ページ

- 「SASpxOptionsBuilder メンバ」 420 ページ
- 「SASpxOptionsBuilder コンストラクタ」 422 ページ

SASpxOptionsBuilder(String) コンストラクタ

SASpxOptionsBuilder オブジェクトを初期化します。

構文

Visual Basic

```
Public Sub New( _
  ByVal options As String _
)
```

C#

```
public  SASpxOptionsBuilder(
  string  options
);
```

パラメータ

- **options**　SQL Anywhere SPX 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。

例

次の文は、SASpxOptionsBuilder オブジェクトを初期化します。

```
SASpxOptionsBuilder options = new SASpxOptionsBuilder( );
```

参照

- 「SASpxOptionsBuilder クラス」 420 ページ
- 「SASpxOptionsBuilder メンバ」 420 ページ
- 「SASpxOptionsBuilder コンストラクタ」 422 ページ

DLL プロパティ

DLL オプションを取得または設定します。

構文

Visual Basic

```
Public Property DLL As String
```

C#

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


参照

- 「SASpxOptionsBuilder クラス」 420 ページ
- 「SASpxOptionsBuilder メンバ」 420 ページ

DoBroadcast プロパティ

DoBroadcast オプションを取得または設定します。

構文

Visual Basic

Public Property **DoBroadcast** As String

C#

public string **DoBroadcast** { get; set; }

参照

- 「SASpxOptionsBuilder クラス」 420 ページ
- 「SASpxOptionsBuilder メンバ」 420 ページ

Host プロパティ

Host オプションを取得または設定します。

構文

Visual Basic

Public Property **Host** As String

C#

public string **Host** { get; set; }

参照

- 「SASpxOptionsBuilder クラス」 420 ページ
- 「SASpxOptionsBuilder メンバ」 420 ページ

SearchBindery プロパティ

SearchBindery オプションを取得または設定します。

構文

Visual Basic

Public Property **SearchBindery** As String

C#

public string **SearchBindery** { get; set; }

参照

- 「SASpxOptionsBuilder クラス」 420 ページ
- 「SASpxOptionsBuilder メンバ」 420 ページ

Timeout プロパティ

Timeout オプションを取得または設定します。

構文

Visual Basic

Public Property **Timeout** As Integer

C#

public int **Timeout** { get; set; }

参照

- 「SASpxOptionsBuilder クラス」 420 ページ
- 「SASpxOptionsBuilder メンバ」 420 ページ

ToString メソッド

SpxOptionsBuilder オブジェクトを文字列表現に変換します。

構文

Visual Basic

Public Overrides Function **ToString()** As String

C#

public override string **ToString();**

戻り値

構築されるオプション文字列。

参照

- 「SASpxOptionsBuilder クラス」 420 ページ
- 「SASpxOptionsBuilder メンバ」 420 ページ

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。このクラスは継承できません。

構文

Visual Basic

```
Public NotInheritable Class SATcpOptionsBuilder
  Inherits SAConnectionStringBuilderBase
```

C#

```
public sealed class SATcpOptionsBuilder : SAConnectionstring BuilderBase
```

備考

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

参照

- 「SATcpOptionsBuilder メンバ」 426 ページ
- 「SAConnection クラス」 236 ページ

SATcpOptionsBuilder メンバ

パブリック・コンストラクタ

メンバ名	説明
SATcpOptionsBuilder コンストラクタ	「SATcpOptionsBuilder クラス」 426 ページの新しいインスタンスを初期化します。

パブリック・プロパティ

メンバ名	説明
Broadcast プロパティ	Broadcast オプションを取得または設定します。
BroadcastListener プロパティ	BroadcastListener オプションを取得または設定します。
BrowsableConnectionString (DbConnectionStringBuilder から継承)	
ClientPort プロパティ	ClientPort オプションを取得または設定します。
ConnectionString (DbConnectionStringBuilder から継承)	

メンバ名	説明
Count (DbConnectionStringBuilder から継承)	
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 (DbConnectionStringBuilder から継承)	
IsReadOnly (DbConnectionStringBuilder から継承)	
Item プロパティ (SAConnectionStringBuilderBase から継承)	接続キーワードの値を取得または設定します。
Keys プロパティ (SAConnectionStringBuilderBase から継承)	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。
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。
Timeout プロパティ	Timeout オプションを取得または設定します。
Values (DbConnectionStringBuilder から継承)	
VerifyServerName プロパティ	VerifyServerName オプションを取得または設定します。

パブリック・メソッド

メンバ名	説明
Add (DbConnectionStringBuilder から継承)	
Clear (DbConnectionStringBuilder から継承)	
ContainsKey メソッド (SAConnectionStringBuilderBase から継承)	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 (DbConnectionStringBuilder から継承)	
GetKeyword メソッド (SAConnectionStringBuilderBase から継承)	指定された SAConnectionStringBuilder プロパティのキーワードを取得します。
GetUseLongNameAsKeyword メソッド (SAConnectionStringBuilderBase から継承)	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 (SAConnectionStringBuilderBase から継承)	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 (SAConnectionStringBuilderBase から継承)	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。デフォルトでは、長い接続パラメータ名が使用されます。
ShouldSerialize メソッド (SAConnectionStringBuilderBase から継承)	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	TcpOptionsBuilder オブジェクトを文字列表現に変換します。
TryGetValue メソッド (SAConnectionStringBuilderBase から継承)	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。

参照

- 「SATcpOptionsBuilder クラス」 426 ページ
- 「SAConnection クラス」 236 ページ

SATcpOptionsBuilder コンストラクタ

「SATcpOptionsBuilder クラス」 426 ページの新しいインスタンスを初期化します。

SATcpOptionsBuilder() コンストラクタ

SATcpOptionsBuilder オブジェクトを初期化します。

構文

Visual Basic

```
Public Sub New()
```

C#

```
public  SATcpOptionsBuilder();
```

備考

制限：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( );
```

参照

- 「SATcpOptionsBuilder クラス」 426 ページ
- 「SATcpOptionsBuilder メンバ」 426 ページ
- 「SATcpOptionsBuilder コンストラクタ」 429 ページ

SATcpOptionsBuilder(String) コンストラクタ

SATcpOptionsBuilder オブジェクトを初期化します。

構文

Visual Basic

```
Public Sub New( _
  ByVal options As String _
)
```

C#

```
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

制限：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( );
```

参照

- 「SATcpOptionsBuilder クラス」 426 ページ
- 「SATcpOptionsBuilder メンバ」 426 ページ
- 「SATcpOptionsBuilder コンストラクタ」 429 ページ

Broadcast プロパティ

Broadcast オプションを取得または設定します。

構文

Visual Basic

```
Public Property Broadcast As String
```

C#

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

参照

- 「SATcpOptionsBuilder クラス」 426 ページ
- 「SATcpOptionsBuilder メンバ」 426 ページ

BroadcastListener プロパティ

BroadcastListener オプションを取得または設定します。

構文

Visual Basic

```
Public Property BroadcastListener As String
```

C#

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

参照

- 「SATcpOptionsBuilder クラス」 426 ページ
- 「SATcpOptionsBuilder メンバ」 426 ページ

ClientPort プロパティ

ClientPort オプションを取得または設定します。

構文

Visual Basic

Public Property **ClientPort** As String

C#

public string **ClientPort** { get; set; }

参照

- 「SATcpOptionsBuilder クラス」 426 ページ
- 「SATcpOptionsBuilder メンバ」 426 ページ

DoBroadcast プロパティ

DoBroadcast オプションを取得または設定します。

構文

Visual Basic

Public Property **DoBroadcast** As String

C#

public string **DoBroadcast** { get; set; }

参照

- 「SATcpOptionsBuilder クラス」 426 ページ
- 「SATcpOptionsBuilder メンバ」 426 ページ

Host プロパティ

Host オプションを取得または設定します。

構文

Visual Basic

Public Property **Host** As String

C#

public string **Host** { get; set; }

参照

- 「SATcpOptionsBuilder クラス」 426 ページ
- 「SATcpOptionsBuilder メンバ」 426 ページ

IPV6 プロパティ

IPV6 オプションを取得または設定します。

構文

Visual Basic

Public Property **IPV6** As String

C#

public string **IPV6** { get; set; }

参照

- 「SATcpOptionsBuilder クラス」 426 ページ
- 「SATcpOptionsBuilder メンバ」 426 ページ

LDAP プロパティ

LDAP オプションを取得または設定します。

構文

Visual Basic

Public Property **LDAP** As String

C#

public string **LDAP** { get; set; }

参照

- 「SATcpOptionsBuilder クラス」 426 ページ
- 「SATcpOptionsBuilder メンバ」 426 ページ

LocalOnly プロパティ

LocalOnly オプションを取得または設定します。

構文

Visual Basic

Public Property **LocalOnly** As String

C#

public string **LocalOnly** { get; set; }

参照

- 「SATcpOptionsBuilder クラス」 426 ページ
- 「SATcpOptionsBuilder メンバ」 426 ページ

MyIP プロパティ

MyIP オプションを取得または設定します。

構文

Visual Basic

Public Property **MyIP** As String

C#

public string **MyIP** { get; set; }

参照

- 「SATcpOptionsBuilder クラス」 426 ページ
- 「SATcpOptionsBuilder メンバ」 426 ページ

ReceiveBufferSize プロパティ

ReceiveBufferSize オプションを取得または設定します。

構文

Visual Basic

Public Property **ReceiveBufferSize** As Integer

C#

public int **ReceiveBufferSize** { get; set; }

参照

- 「SATcpOptionsBuilder クラス」 426 ページ
- 「SATcpOptionsBuilder メンバ」 426 ページ

SendBufferSize プロパティ

Send BufferSize オプションを取得または設定します。

構文

Visual Basic

Public Property **SendBufferSize** As Integer

C#

public int **SendBufferSize** { get; set; }

参照

- 「SATcpOptionsBuilder クラス」 426 ページ
- 「SATcpOptionsBuilder メンバ」 426 ページ

ServerPort プロパティ

ServerPort オプションを取得または設定します。

構文

Visual Basic

Public Property **ServerPort** As String

C#

public string **ServerPort** { get; set; }

参照

- 「SATcpOptionsBuilder クラス」 426 ページ
- 「SATcpOptionsBuilder メンバ」 426 ページ

TDS プロパティ

TDS オプションを取得または設定します。

構文

Visual Basic

Public Property **TDS** As String

C#

public string **TDS** { get; set; }

参照

- 「SATcpOptionsBuilder クラス」 426 ページ
- 「SATcpOptionsBuilder メンバ」 426 ページ

Timeout プロパティ

Timeout オプションを取得または設定します。

構文

Visual Basic

```
Public Property Timeout As Integer
```

C#

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

参照

- 「SATcpOptionsBuilder クラス」 426 ページ
- 「SATcpOptionsBuilder メンバ」 426 ページ

VerifyServerName プロパティ

VerifyServerName オプションを取得または設定します。

構文

Visual Basic

```
Public Property VerifyServerName As String
```

C#

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

参照

- 「SATcpOptionsBuilder クラス」 426 ページ
- 「SATcpOptionsBuilder メンバ」 426 ページ

ToString メソッド

TcpOptionsBuilder オブジェクトを文字列表現に変換します。

構文

Visual Basic

```
Public Overrides Function ToString() As String
```


C#

public override string **ToString();**

戻り値

構築されるオプション文字列。

参照

- 「SATcpOptionsBuilder クラス」 426 ページ
- 「SATcpOptionsBuilder メンバ」 426 ページ

SATransaction クラス

SQL トランザクションを示します。このクラスは継承できません。

構文

Visual Basic

Public NotInheritable Class **SATransaction**
  Inherits DbTransaction

C#

public sealed class **SATransaction** : DbTransaction

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」 142 ページと「SACommand オブジェクトを使用したローの挿入、更新、削除」 125 ページを参照してください。

参照

- 「SATransaction メンバ」 437 ページ
- 「BeginTransaction() メソッド」 243 ページ
- 「BeginTransaction(SAIsolationLevel) メソッド」 245 ページ
- 「Transaction プロパティ」 202 ページ

SATransaction メンバ

パブリック・プロパティ

メンバ名	説明
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独立性レベルを指定します。

パブリック・メソッド

メンバ名	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。

メンバ名	説明
Dispose (DbTransaction から継承)	
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。

参照

- 「SATransaction クラス」 437 ページ
- 「BeginTransaction() メソッド」 243 ページ
- 「BeginTransaction(SAIsolationLevel) メソッド」 245 ページ
- 「Transaction プロパティ」 202 ページ

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。

構文

Visual Basic

```
Public Readonly Property Connection As SAConnection
```

C#

```
public 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。

参照

- 「SATransaction クラス」 437 ページ
- 「SATransaction メンバ」 437 ページ

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。

構文

Visual Basic

Public Overrides Readonly Property **IsolationLevel** As IsolationLevel

C#

public override IsolationLevel **IsolationLevel** { get;}

プロパティ値

このトランザクションの独立性レベル。次のいずれかを指定してください。

- ReadCommitted
- ReadUncommitted
- RepeatableRead
- Serializable
- Snapshot
- ReadOnlySnapshot
- StatementSnapshot

デフォルトは ReadCommitted です。

参照

- 「SATransaction クラス」 437 ページ
- 「SATransaction メンバ」 437 ページ

SAIsolationLevel 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。

構文

Visual Basic

Public Readonly Property **SAIsolationLevel** As SAIsolationLevel

C#

public SAIsolationLevel **SAIsolationLevel** { get;}

プロパティ値

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。

- Chaos
- ReadCommitted
- ReadOnlySnapshot
- ReadUncommitted
- RepeatableRead
- Serializable

- Snapshot
- StatementSnapshot
- Unspecified

デフォルトは ReadCommitted です。

備考

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、IsolationLevel はトランザクション全体に適用されます。

参照

- 「SATransaction クラス」 437 ページ
- 「SATransaction メンバ」 437 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。

構文

Visual Basic

Public Overrides Sub **Commit()**

C#

public override void **Commit();**

参照

- 「SATransaction クラス」 437 ページ
- 「SATransaction メンバ」 437 ページ

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() 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。

構文

Visual Basic

Public Overrides Sub **Rollback()**

C#

```
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。

参照

- 「SATransaction クラス」 437 ページ
- 「SATransaction メンバ」 437 ページ
- 「Rollback メソッド」 440 ページ

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。

構文

Visual Basic

```
Public Sub Rollback( _
  ByVal savePoint As String _
)
```

C#

```
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。

参照

- 「SATransaction クラス」 437 ページ
- 「SATransaction メンバ」 437 ページ
- 「Rollback メソッド」 440 ページ

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。

構文

Visual Basic

```
Public Sub Save( _
  ByVal savePoint As String _
)
```

C#

```
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。

参照

- 「SATransaction クラス」 437 ページ
- 「SATransaction メンバ」 437 ページ

第 10 章

SQL Anywhere OLE DB と ADO API

目次

OLE DB の概要

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、Windows と Windows CE の現在のプラットフォームで利用できます。

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、OLE DB 2.5 以降で動作するように設計されています。Windows CE およびその後継プラットフォームに関しては、OLE DB プロバイダは ADOCE 3.0 以降で動作するように設計されています。

ADOCE は Microsoft による Windows CE SDK の ADO で、Windows CE Toolkits for Visual Basic 5.0 と Windows CE Toolkits for Visual Basic 6.0 で開発されたアプリケーションに、データベース機能を提供します。

サポートされているプラットフォームのリストについては、「SQL Anywhere がサポートするプラットフォームおよびエンジニアリング・サポート状況」の SQL Anywhere PC プラットフォーム版の表を参照してください。

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。

詳細については、「3 層コンピューティングと分散トランザクション」 67 ページを参照してください。

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、以下のファイルにあります。

開発ツール	サンプル
Microsoft Visual Basic 6.0	*samples-dir¥SQLAnywhere¥VBSampler¥vbsampler.vbp*
Microsoft eMbedded Visual Basic 3.0	*samples-dir¥SQLAnywhere¥ADOCE¥OLEDB_PocketPC.ebp*

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 オブジェクトでデータベースに接続する

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 Command1 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()
    ' Declare variables
    Dim myConn As New ADODB.Connection
    Dim myCommand As New ADODB.Command
    Dim cAffected As Long

    On Error GoTo HandleError

    ' Establish the connection
    myConn.Provider = "SAOLEDB"
    myConn.ConnectionString = _
      "Data Source=SQL Anywhere 10 Demo"
    myConn.Open
    MsgBox "Connection succeeded"
    myConn.Close
    Exit Sub
```

```
HandleError:
   MsgBox "Connection failed"
   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、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。

SAOLEDB プロバイダは、インストールされると自動的に登録を行います。この登録プロセスには、レジストリの COM セクションへのレジストリ・エントリの作成も含まれます。このため、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、それを登録する必要があります。

♦ **OLE DB プロバイダを登録するには、次の手順に従います。**

1. コマンド・プロンプトを開きます。
2. OLE DB プロバイダがインストールされているディレクトリに移動します。
3. 次のコマンドを入力して、プロバイダを登録します。

   ```
   regsvr32 dboledb10.dll
   ```

OLE DB を使用したデータベースへの接続の詳細については、「OLE DB を使用したデータベースへの接続」 『SQL Anywhere サーバ - データベース管理』を参照してください。

Command オブジェクトを使用した文の実行

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 Command2 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、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()
   ' Declare variables
   Dim myConn As New ADODB.Connection
   Dim myCommand As New ADODB.Command
   Dim cAffected As Long
   ' Establish the connection
   myConn.Provider = "SAOLEDB"
   myConn.ConnectionString = _
```



```
    "Data Source=SQL Anywhere 10 Demo"
  myConn.Open

  'Execute a command
  myCommand.CommandText = _
  "update Customers set GivenName='Liz' where ID=102"
  Set myCommand.ActiveConnection = myConn
  myCommand.Execute cAffected
  MsgBox CStr(cAffected) +
  " rows affected.", vbInformation

  myConn.Close
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メッセージ・ボックスに表示します。

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。

ADO でのトランザクションの使用については、「トランザクションの使用」 450 ページを参照してください。

カーソルを使用して更新を実行することもできます。

詳細については、「カーソルによるデータの更新」 449 ページを参照してください。

Recordset オブジェクトを使用したデータベースのクエリ

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、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 のローのメッセージ・ボックスへの表示、切断を行います。

```
Private Sub cmdQuery_Click()
' Declare variables
  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 10 Demo"
  myConn.CursorLocation = adUseServer
  myConn.Mode = adModeReadWrite
  myConn.IsolationLevel = adXactCursorStability
  myConn.Open
```

```
    'Execute a query
    Set myRS = New Recordset
    myRS.CacheSize = 50
    myRS.Source = "SELECT * FROM Customers"
    myRS.ActiveConnection = myConn
    myRS.CursorType = adOpenKeyset
    myRS.LockType = adLockOptimistic
    myRS.Open

    'Scroll through the first few results
    myRS.MoveFirst
    For i = 1 To 5
      MsgBox myRS.Fields("CompanyName"), vbInformation
      myRS.MoveNext
    Next

    myRS.Close
    myConn.Close
    Exit Sub
ErrorHandler:
    MsgBox Error(Err)
    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 オブジェクトの処理」 448 ページを参照してください。

Recordset オブジェクトの処理

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」 41 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

ADO カーソル・タイプ	ADO 定数	SQL Anywhere のタイプ
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

アプリケーションに適したカーソル・タイプの選択方法については、「カーソル・タイプの選択」 41 ページを参照してください。

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 = adOpenDynamic
```

カーソルによるデータの更新

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 Command6_Click()
    Dim myConn As New ADODB.Connection
    Dim myRS As New ADODB.Recordset
    Dim SQLString As String
    ' Connect
    myConn.Provider = "SAOLEDB"
    myConn.ConnectionString = _
        "Data Source=SQL Anywhere 10 Demo"
    myConn.Open
    myConn.BeginTrans
    SQLString = "SELECT * FROM Customers"
    myRS.Open SQLString, _
        myConn, adOpenDynamic, adLockBatchOptimistic

    If myRS.BOF And myRS.EOF Then
        MsgBox "Recordset is empty!", _
        16, "Empty Recordset"
    Else
        MsgBox "Cursor type: " + _
        CStr(myRS.CursorType), vbInformation
        myRS.MoveFirst
        For i = 1 To 3
            MsgBox "Row: " + CStr(myRS.Fields("ID")), _
             vbInformation
            If i = 2 Then
                myRS.Update "City", "Toronto"
                myRS.UpdateBatch
            End If
            myRS.MoveNext
        Next i
'       myRS.MovePrevious
```



```
        myRS.Close
    End If
    myConn.CommitTrans
    myConn.Close
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。

トランザクションの使用

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
コミットしない読み出し	adXactReadUncommitted	0
カーソル安定性	adXactCursorStability	1
コミットする読み出し	adXactReadCommitted	1
繰り返し可能な読み出し	adXactRepeatableRead	2
独立	adXactIsolated	3
逐次化可能	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

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』を参照してください。

OLE DB を使用する Microsoft リンク・サーバの設定

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..GROUPO.Customers
```

この例で、SADATABASE はリンク・サーバの名前、GROUPO は SQL Anywhere データベースのテーブル所有者、Customers は SQL Anywhere データベースのテーブル名です。カタログ名が省略されていますが (連続した 2 つのドットがある箇所)、これは SQL Anywhere データベースではカタログ名がサポートされていないからです。

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。

SQL Anywhere OLE DB プロバイダを使用するリンク・サーバを設定するには、必要な手順があります。

♦ リンク・サーバを設定するには、次の手順に従います。

1. [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 サーバー] 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、リストから [SQL Anywhere OLE DB Provider] を選択します。[製品名] フィールドには、ODBC データ・ソース名を指定します (SQL Anywhere 10 Demo など)。[プロバイダ文字列] フィールドには、ユーザ ID やパスワードなど、追加の接続パラメータを指定できます (uid=DBA;pwd=sql など)。[全般] ページの [データ ソース] などのその他のフィールドは空欄のままにします。

2. [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 の場合、[リンク サーバー] – [プロバイダ] ツリー・ビューで [SAOLEDB] を右クリックし、[プロパティ] を選択すると、グローバルの [InProcess 許可] チェックボックスが表示されます。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。

3. [RPC] オプションと [RPC 出力] 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、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。チェック・ボックスは、[サーバー オプション] ページにあります。SQL Server 2005 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関数の呼び出しを 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。

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。	Windows CE ではサポートされません。
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。	Windows CE ではサポートされません。
ICommand	SQL コマンドを実行する。	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。
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。	Windows CE ではサポートされません。
ICommandPrepare	コマンドを準備する。	Windows CE ではサポートされません。
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。

インタフェース	内容	制限事項
ICommandText	ICommand に SQL コマンドを設定する。	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。 CE ではサポートされていません。
IConvertType		サポートされています。 CE では制限があります。
IDBAsynchNotify IDBAsyncStatus	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。
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。	Windows CE ではサポートされません。
IDBInitialize	データ・ソース・オブジェクトと列挙子を初期化する。	Windows CE ではサポートされません。
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。	Windows CE ではサポートされません。

インタフェース	内容	制限事項
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。	Windows CE ではサポートされません。
IErrorInfo IErrorLookup IErrorRecords	ActiveX エラー・オブジェクト・サポート。	Windows CE ではサポートされません。
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。
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。	Windows CE ではサポートされません。
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。

インタフェース	内容	制限事項
IRowsetInfo	ローセット・プロパティについての情報を検索する、または、ローセットを作成したオブジェクトを検索する。	Windows CE ではサポートされません。
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。	Windows CE ではサポートされません。
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。
IRowsetResynch	以前の OLEDB 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。 CE ではサポートされていません。
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。 CE で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。	Windows CE ではサポートされません。
ISQLErrorInfo ISupportErrorInfo	ActiveX エラー・オブジェクト・サポート。	CE ではオプション。
ITableDefinition ITableDefinitionWithConstraints	制約を使用して、テーブルを作成、削除、変更する。	Windows CE ではサポートされません。

インタフェース	内容	制限事項
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。 CE ではサポートされていません。
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。 CE ではサポートされていません。
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。	Windows CE ではサポートされません。
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。	Windows CE ではサポートされません。
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。

第 11 章

SQL Anywhere ODBC API

目次

ODBC の概要

ODBC (Open Database Connectivity) インタフェースは、Windows オペレーティング・システムにおけるデータベース管理システムへの標準インタフェースとして Microsoft が規定した、アプリケーション・プログラミング・インタフェースです。ODBC は呼び出しベースのインタフェースです。

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 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。

サポートするプラットフォーム

SQL Anywhere は、Windows の他にも、UNIX 上と Windows CE 上の ODBC API をサポートしています。マルチプラットフォーム ODBC をサポートすることにより、移植可能なデータベース・アプリケーションの開発が非常に簡単になります。

分散トランザクションにおける ODBC ドライバのエンリストの詳細については、「3 層コンピューティングと分散トランザクション」 67 ページを参照してください。

参照

- ODBC SDK のマニュアル

> **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しないですむ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 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。これらの機能のリストについては、『*ODBC Programmer's Reference*』を参照してください。Microsoft から ODBC ソフトウェア開発キットの一部として入手するか、Microsoft Web サイト http://msdn.microsoft.com/library/en-us/odbc/htm/odbcabout_this_manual.asp から入手できます。

SQL Anywhere がサポートする機能

SQL Anywhere は ODBC 3.5 仕様をサポートします。

- **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 の下位互換性

以前のバージョンの ODBC を使用して開発されたアプリケーションは、SQL Anywhere と新しい ODBC ドライバ・マネージャでも引き続き動作します。ただし、ODBC の新しい機能は以前のアプリケーションでは使用できません。

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。UNIX の場合、ODBC ドライバ・マネージャは SQL Anywhere に付属で提供されます。

ODBC アプリケーションの構築

この項では、簡単な ODBC アプリケーションをコンパイルしてリンクする方法について説明します。

ODBC ヘッダ・ファイルのインクルード

ODBC 関数を呼び出す 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 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、次の手順に従います。

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 CE	#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 *h* サブディレクトリにインストールされます。

Windows での ODBC アプリケーションのリンク

この項は、Windows CE には該当しません。

Windows CE については、「Windows CE での ODBC アプリケーションのリンク」 463 ページを参照してください。

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 *dbodbc10.dll* をロードします。

Microsoft と Watcom の各コンパイラ用に、別々のインポート・ライブラリが用意されています。

♦ **ODBC アプリケーションをリンクするには、次の手順に従います (Windows の場合)。**

- 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、ライブラリ・ディレクトリのリストに追加します。

  インポート・ライブラリは、SQL Anywhere 実行プログラムがあるディレクトリの *lib* サブディレクトリに格納されています。ライブラリ名は、次のとおりです。

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	*odbc32.lib*
Windows	Watcom C/C++	*wodbc32.lib*
Windows	Borland	*bodbc32.lib*
Windows CE	Microsoft	*dbodbc10.lib*

Windows CE での ODBC アプリケーションのリンク

Windows CE 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 (*dbodbc10.lib*) では、SQL Anywhere の ODBC ドライバ *dbodbc10.dll* への直接のエントリ・ポイントが定義されています。

Windows CE 用のチップごとに、この DLL の別々のバージョンが用意されています。各ファイルは、SQL Anywhere のインストール・ディレクトリにある *ce* ディレクトリ内のオペレーティング・システム固有のサブディレクトリにあります。たとえば、ARM チップを使用する Windows CE 用の ODBC ドライバは、次のディレクトリにあります。

```
C:¥Program Files¥SQL Anywhere 10¥ce¥arm.30
```

サポートされる Windows CE のバージョンのリストについては、「SQL Anywhere がサポートするプラットフォームおよびエンジニアリング・サポート状況」の SQL Anywhere PC プラットフォーム版の表を参照してください。

♦ **ODBC アプリケーションをリンクするには、次の手順に従います (Windows CE の場合)。**

1. 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、ライブラリ・ディレクトリのリストに追加します。

   インポート・ライブラリ名は *dbodbc10.lib* で、SQL Anywhere インストール・ディレクトリの *ce* ディレクトリにあるオペレーティング・システム固有のサブディレクトリに格納されています。たとえば、ARM チップを使用する Windows CE 用のインポート・ライブラリは、次のディレクトリにあります。

   ```
   C:¥Program Files¥SQL Anywhere 10¥ce¥arm.30¥lib
   ```

2. SQLDriverConnect 関数に与える接続文字列内で DRIVER= パラメータを指定します。

   ```
   szConnStrIn = "driver=ospath¥¥dbodbc10.dll;dbf=¥¥samples-dir¥¥demo.db"
   ```


ospath は Windows CE デバイス上の Windows ディレクトリへのフル・パスです。次に例を示します。

```
¥¥Windows
```

samples-dir のデフォルト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。

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 10 Demo.dsn* を使用します。このファイルのコピーは、SQL Anywhere インストール・ディレクトリの *ce* ディレクトリにあります。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・システム用に設定して、Windows CE 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 CE デバイスにコピーします。

Windows CE とユニコード

SQL Anywhere は、ユニコード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) 文字列またはユニコード (ワイド) 文字列をサポートします。UNICODE マクロは、ODBC 関数が ASCII またはユニコード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。

ユニコード ODBC の機能については、*samples-dir¥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 アプリケーションのリンク

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	*libdbodbc10.so (libdbodbc10.so.1)*

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)	シングルスレッド	*libdbodbc10_n.so (libdbodbc10_n.so.1)*
(Mac OS X と HP-UX 以外のすべての UNIX)	マルチスレッド	*libdbodbc10_r.so (libdbodbc10_r.so.1)*
HP-UX	汎用	*libdbodbc10.sl (libdbodbc10.sl.1)*
HP-UX	シングルスレッド	*libdbodbc10_n.sl (libdbodbc10_n.sl.1)*
HP-UX	マルチスレッド	*libdbodbc10_r.sl (libdbodbc10_r.sl.1)*
Mac OS X	汎用	*libdbodbc10.dylib*
Mac OS X	シングルスレッド	*libdbodbc10_n.dylib*
Mac OS X	マルチスレッド	*libdbodbc10_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	*dbodbc10.bundle*
Mac OS X	マルチスレッド	*dbodbc10_r.bundle*

♦ **ODBC アプリケーションをリンクするには、次の手順に従います (UNIX の場合)。**

1. アプリケーションを汎用 ODBC ドライバ *libdbodbc10* にリンクします。
2. 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。「UNIX での ODBC データ・ソースの使用」 『SQL Anywhere サーバ - データベース管理』を参照してください。

UNIX 上での ODBC ドライバ・マネージャの使用

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。*libdbodm10* 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。iAnywhere ODBC 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、iAnywhere ODBC 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。iAnywhere ODBC 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。

iAnywhere ODBC 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 は接続に関してトレースされた出力を記録するログ・ファイルで、TraceLevel はトレーシング情報の量を表します。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。

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、*samples-dir¥SQLAnywhere* サブディレクトリにあります。

ディレクトリ内の ODBC 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、*samples-dir¥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。

関連する Embedded SQL プログラムの詳細については、「サンプル Embedded SQL プログラム」 527 ページを参照してください。

サンプル ODBC プログラムの構築

samples-dir¥SQLAnywhere¥C にある ODBC サンプル・プログラムにはバッチ・ファイル (UNIX 用のシェル・スクリプト) が含まれています。このバッチ・ファイルは、サンプル・アプリケーションのコンパイルとリンクに使用できます。

♦ サンプル ODBC プログラムを構築するには、次の手順に従います。

1. コマンド・プロンプトを開き、*samples-dir¥SQLAnywhere¥C* ディレクトリに移動します。
2. *makeall* バッチ・ファイルまたはシェル・スクリプトを実行します。

   コマンドのフォーマットを次に示します。

   ```
   makeall api platform compiler
   ```

   パラメータは次のとおりです。

   - **API**　アプリケーションの Embedded SQL バージョンではなく ODBC のサンプルをコンパイルするように、**odbc** を指定します。
   - **Platform**　Windows オペレーティング・システム用にコンパイルするように、**WINDOWS** を指定します。
   - **Compiler**　プログラムのコンパイルに使用するコンパイラを指定します。次のいずれかを指定できます。
     - **WC**　Watcom C/C++ を使用
     - **MC**　Microsoft Visual C++ を使用
     - **BC**　Borland C++ Builder を使用

サンプル ODBC プログラムの実行

♦ **ODBC のサンプルを実行するには、次の手順に従います。**

1. プログラムを起動します。
   - ファイル *samples-dir¥SQLAnywhere¥C¥odbcwnt.exe* を実行します。
2. テーブルを選択します。
   - 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** と入力します。

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。

  ```
  SQLHENV env;
  SQLRETURN rc;
  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。

  ```
  SQLHDBC  dbc;
  SQLRETURN rc;
  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。

  ```
  SQLHSTMT stmt;
  SQLRETURN rc;
  rc = SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
  ```

ODBC ハンドルの割り付け

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、次の手順に従います。**

1. SQLAllocHandle 関数を呼び出します。

   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、Microsoft の『*ODBC Programmer's Reference*』の「SQLAllocHandle」を参照してください。
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。

   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、Microsoft の『ODBC Programmer's 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。

```
SQLHENV env;
SQLRETURN retcode;
retcode = SQLAllocHandle(
      SQL_HANDLE_ENV,
      SQL_NULL_HANDLE,
      &env );
if( retcode == SQL_SUCCESS
   || retcode == SQL_SUCCESS_WITH_INFO ) {
  // success: application Code here
}
SQLFreeHandle( SQL_HANDLE_ENV, env );
```

リターン・コードとエラー処理の詳細については、「エラー処理」 488 ページを参照してください。

ODBC の例 1

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。

このサンプルは *samples-dir¥SQLAnywhere¥ODBCConnect¥odbcconnect.cpp* にあります。


```
#include <stdio.h>
#include "ntodbc.h"

int main(int argc, char* argv[])
{
  SQLHENV  env;
  SQLHDBC  dbc;
  SQLRETURN retcode;

  retcode = SQLAllocHandle( SQL_HANDLE_ENV,
            SQL_NULL_HANDLE,
            &env );
  if (retcode == SQL_SUCCESS
     || retcode == SQL_SUCCESS_WITH_INFO) {
    printf( "env allocated¥n" );
    /* Set the ODBC version environment attribute */
    retcode = SQLSetEnvAttr( env,
                SQL_ATTR_ODBC_VERSION,
                (void*)SQL_OV_ODBC3, 0);
      retcode = SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
      if (retcode == SQL_SUCCESS
          || retcode == SQL_SUCCESS_WITH_INFO) {
        printf( "dbc allocated¥n" );
        retcode = SQLConnect( dbc,
          (SQLCHAR*) "SQL Anywhere 10 Demo", SQL_NTS,
          (SQLCHAR* ) "DBA", SQL_NTS,
          (SQLCHAR*) "sql", SQL_NTS );
        if (retcode == SQL_SUCCESS
           || retcode == SQL_SUCCESS_WITH_INFO) {
          printf( "Successfully connected¥n" );
        }
        SQLDisconnect( dbc );
      }
      SQLFreeHandle( SQL_HANDLE_DBC, dbc );
    }
    SQLFreeHandle( SQL_HANDLE_ENV, env );
  return 0;
}
```

ODBC 接続関数の選択

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、Microsoft の『*ODBC Programmer's 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。

  詳細については、Microsoft の『*ODBC Programmer's 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ダイアログ・ボックスを構築して、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、Microsoft の『*ODBC Programmer's Reference*』の「SQLBrowseConnect」を参照してください。

接続文字列に使用できる接続パラメータの詳細リストについては、「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。

接続の確立

アプリケーションでデータベース操作を実行するには、接続を確立します。

- **ODBC 接続を確立するには、次の手順に従います。**

1. ODBC 環境を割り付けます。

   例：

```
SQLHENV   env;
SQLRETURN retcode;
retcode = 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。

   ```
   retcode = SQLSetEnvAttr( env,
     SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);
   ```

3. 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。

4. ODBC 接続項目を割り付けます。

   例：

   ```
   retcode = SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
   ```

5. 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。

   ```
   retcode = SQLSetConnectAttr( dbc,
     SQL_AUTOCOMMIT,
     (SQLPOINTER)SQL_AUTOCOMMIT_OFF, 0 );
   ```

   詳細については、「接続属性の設定」 474 ページを参照してください。

6. ODBC 接続関数を呼び出します。

   例：

   ```
   if (retcode == SQL_SUCCESS
       || retcode == SQL_SUCCESS_WITH_INFO) {
     printf( "dbc allocated¥n" );
     retcode = SQLConnect( dbc,
       (SQLCHAR*) "SQL Anywhere 10 Demo", SQL_NTS,
       (SQLCHAR* ) "DBA", SQL_NTS,
       (SQLCHAR*) "sql", SQL_NTS );
     if (retcode == SQL_SUCCESS
         || retcode == SQL_SUCCESS_WITH_INFO){
     // successfully connected.
   ```

完全なサンプルは、*samples-dir¥SQLAnywhere¥ODBCConnect¥odbcconnect.cpp* にあります。

注意

- **SQL_NTS**　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。

- **SQLSetConnectAttr**　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。

  詳細については、「接続属性の設定」 474 ページを参照してください。

接続属性の設定

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。

```
retcode = SQLSetConnectAttr( dbc, SQL_AUTOCOMMIT,
  (SQLPOINTER)SQL_AUTOCOMMIT_OFF, 0 );
```

接続属性のリストなどの詳細については、Microsoft の『*ODBC Programmer's 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。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』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。

SQL 文の実行

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。

  詳細については、「準備文の実行」 477 ページを参照してください。

文の直接実行

SQLExecDirect 関数は、SQL 文を準備して実行します。文には、パラメータを指定することもできます。

次のコード・フラグメントは、パラメータを指定しないで文を実行する方法を示します。SQLExecDirect 関数は、ステートメント・ハンドル、SQL 文字列、長さまたは終了インジケータをパラメータに取ります。この場合、終了インジケータは NULL で終了された文字列インジケータです。

この項で説明する手順は単純ですが、柔軟性がありません。アプリケーションでは、ユーザの入力によってこの文を修正できません。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」 476 ページを参照してください。

♦ ODBC アプリケーションで SQL 文を実行するには、次の手順に従います。

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、名前が stmt の SQL_HANDLE_STMT 型のハンドルを割り付けます。

   ```
   SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
   ```

2. SQLExecDirect 関数を呼び出して文を実行します。

   たとえば、次の行は文を宣言して実行します。通常、deletestmt の宣言は関数の先頭にあります。

   ```
   SQLCHAR deletestmt[ STMT_LEN ] =
     "DELETE FROM Departments WHERE DepartmentID = 201";
   SQLExecDirect( stmt, deletestmt, SQL_NTS) ;
   ```

エラー・チェックを含む完全なサンプルについては、*samples-dir¥SQLAnywhere¥ODBCExecute¥odbcexecute.cpp* を参照してください。

SQLExecDirect の詳細については、Microsoft の『*ODBC Programmer's Reference*』の「SQLExecDirect」を参照してください。

バウンド・パラメータを使用した文の実行

この項では、SQL 文を構成し、バウンド・パラメータを使用して実行時に文のパラメータ値を設定して実行する方法について説明します。

♦ ODBC アプリケーションでバウンド・パラメータを使用して SQL 文を実行するには、次の手順に従います。

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、名前が stmt 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、文の文字列自体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
   SQLINTEGER cbDeptID = 0,
     cbDeptName = SQL_NTS, cbManagerID = 0;
   SQLCHAR deptName[ DEPT_NAME_LEN + 1 ];
   SQLSMALLINT deptID, managerID;
   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) ;
   ```


また、バインド・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。詳細については、「準備文の実行」 477 ページを参照してください。

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、*samples-dir¥SQLAnywhere¥ODBCExecute¥odbcexecute.cpp* を参照してください。

SQLExecDirect の詳細については、Microsoft の『*ODBC Programmer's Reference*』の「SQLExecDirect」を参照してください。

準備文の実行

準備文を使用すると、繰り返し使用する文のパフォーマンスが向上します。ODBC は、準備文を使用するための関数をすべて提供しています。

準備文の概要については、「文の準備」 28 ページを参照してください。

- **SQL 準備文を実行するには、次の手順に従います。**

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。

```
SQLRETURN   retcode;
SQLHSTMT    stmt;
retcode = 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。

   - **retcode**　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**　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。

2. SQLBindParameter を使用して文のパラメータ値を設定します。

   たとえば、次の関数呼び出しは DepartmentID 変数の値を設定します。

```
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。

- **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。

   ```
   retcode = SQLExecute( stmt);
   ```

   手順 2 ー 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。

エラー・チェックを含む完全なサンプルについては、*samples-dir¥SQLAnywhere¥ODBCPrepare¥odbcprepare.cpp* を参照してください。

SQLPrepare の詳細については、Microsoft の『*ODBC Programmer's Reference*』の「SQLPrepare」を参照してください。

結果セットの処理

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。

カーソルの概要については、「カーソルを使用した操作」 34 ページを参照してください。

ODBC トランザクションの独立性レベルの選択

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、スナップショット・アイソレーション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、文のスナップショット・アイソレーションよりデータの整合性は若干低くなりますが、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。

詳細については、Microsoft の『*ODBC Programmer's Reference*』の「SQLSetConnectAttr」を参照してください。

例

次のフラグメントは、snapshot 独立性レベルを使用します。

```
SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
SQLSetConnectAttr( dbc, SQL_ATTR_TXN_ISOLATION,
    SA_SQL_TXN_SNAPSHOT, SQL_IS_UINTEGER );
```

ODBC カーソル特性の選択

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、「データの取り出し」 481 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、Microsoft の『*ODBC Programmer's 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 );
```

データの取り出し

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 ドライバでは、ODBC 仕様での意図とは別の方法で SQL_ATTR_MAX_LENGTH を実装していることに注意してください。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。これは、*samples-dir¥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 retcode;
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 10 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( ( retcode = 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。

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。次の表に、各プロセッサ (x86、x64、PowerPC プラットフォームを除く) のメモリ・アラインメント要件を示します。Itanium-IA64 などのプロセッサや ARM ベースのデバイスでは、メモリ・アライメントが必要です。

C のデータ型	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

C のデータ型	必要なアラインメント
SQL_C_STINYINT	なし
SQL_C_UTINYINT	なし
SQL_C_TINYINT	なし
SQL_C_NUMERIC	なし
SQL_C_DEFAULT	なし
SQL_C_SSHORT	2
SQL_C_USHORT	2
SQL_C_SHORT	2
SQL_C_DATE	2
SQL_C_TIME	2
SQL_C_TIMESTAMP	2
SQL_C_TYPE_DATE	2
SQL_C_TYPE_TIME	2
SQL_C_TYPE_TIMESTAMP	2
SQL_C_WCHAR	2 (すべてのプラットフォームでバッファ・サイズは 2 の倍数であることが必要)
SQL_C_SLONG	4
SQL_C_ULONG	4
SQL_C_LONG	4
SQL_C_FLOAT	4
SQL_C_DOUBLE	8
SQL_C_SBIGINT	8
SQL_C_UBIGINT	8

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。

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変更文が有効であるかぎり、位置付けデータ変更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。

  詳細については、「ansi_update_constraints オプション [互換性]」『SQL Anywhere サーバ - データベース管理』を参照してください。
- 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。

ブックマークの使用

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。これはつまり、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・プロシージャの呼び出し

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。

ストアド・プロシージャとトリガの詳細については、「プロシージャ、トリガ、バッチの使用」『SQL Anywhere サーバ - SQL の使用法』を参照してください。

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

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_col 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;
HSTMT stmt;
long I;
SWORD num_col;

/* Create a procedure */
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_col );
do_something( I );
```

例

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_col** 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。

```
HDBC dbc;
HSTMT stmt;
SWORD num_col;
RETCODE retcode;
char ID[ 10 ];
char Surname[ 20 ];

/* Create the procedure */
SQLExecDirect( stmt,
     "CREATE PROCEDURE employees()" ¥
     " 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 employees()", SQL_NTS );
SQLNumResultCols( stmt, &num_col );
SQLBindCol( stmt, 1, SQL_C_CHAR, &ID,
          sizeof(ID), NULL );
SQLBindCol( stmt, 2, SQL_C_CHAR, &Surname,
          sizeof(Surname), NULL );

for( ;; ) {
  retcode = SQLFetch( stmt );
  if( retcode == SQL_NO_DATA_FOUND ) {
    retcode = SQLMoreResults( stmt );
  if( retcode == SQL_NO_DATA_FOUND ) break;
}  else {
     do_something( ID, Surname );
  }
}
```

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。
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。

```
/* Declare required variables */
SQLHDBC dbc;
SQLHSTMT stmt;
SQLRETURN retcode;
UCHAR errmsg[100];
/* Code omitted here */
retcode = SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt );
if( retcode == SQL_ERROR ){
   SQLError( env, dbc, SQL_NULL_HSTMT, NULL, NULL,
         errmsg, sizeof(errmsg), NULL );
   /* Assume that print_error is defined */
   print_error( "Allocation failed", errmsg );
   return;
}

/* Delete items for order 2015 */
retcode = SQLExecDirect( stmt,
      "DELETE FROM SalesOrderItems WHERE ID=2015",
      SQL_NTS );
if( retcode == SQL_ERROR ) {
   SQLError( env, dbc, stmt, NULL, NULL,
          errmsg, sizeof(errmsg), NULL );
   /* Assume that print_error is defined */
   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。

```
/* Declare required variables */
SQLHDBC dbc;
SQLHSTMT stmt;
SQLRETURN retcode;
SQLSMALLINT errmsglen;
SQLINTEGER errnative;
UCHAR errmsg[255];
UCHAR errstate[5];
/* Code omitted here */
retcode = SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt );
if( retcode == SQL_ERROR ){
   SQLGetDiagRec(SQL_HANDLE_DBC, dbc, 1, errstate,
      &errnative, errmsg, sizeof(errmsg), &errmsglen);
    /* Assume that print_error is defined */
   print_error( "Allocation failed",
errstate, errnative, errmsg );
   return;

}
```

```
/* Delete items for order 2015 */
retcode = SQLExecDirect( stmt,
      "DELETE FROM SalesOrderItems WHERE ID=2015",
      SQL_NTS );
if( retcode == SQL_ERROR ) {
   SQLGetDiagRec(SQL_HANDLE_STMT, stmt,
      recnum, errstate,
      &errnative, errmsg, sizeof(errmsg), &errmsglen);
   /* Assume that print_error is defined */
   print_error("Failed to delete items",
      errstate, errnative, errmsg );
   return;
}
```

第 12 章

SQL Anywhere JDBC API

目次

JDBC の概要

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 は、iAnywhere JDBC ドライバ (Type 2 JDBC ドライバ) と pure Java アプリケーション用の jConnect ドライバ (Type 4 JDBC ドライバ) の 2 つの JDBC ドライバをサポートし、同梱しています。

- **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**　この章で示したサンプルのソース・コードは、*samples-dir¥SQL Anywhere¥JDBC* ディレクトリにあります。

- **必要なソフトウェア**　jConnect ドライバを使用するには、TCP/IP が必要です。

  jConnect ドライバは、http://www.sybase.com/products/informationmanagement/softwaredeveloperkit/jconnect から入手できます。

  jConnect ドライバとそのロケーションの詳細については、「jConnect ドライバのファイル」 497 ページを参照してください。

JDBC ドライバの選択

SQL Anywhere がサポートする JDBC ドライバは次のとおりです。

- **iAnywhere JDBC ドライバ**　このドライバは、Command Sequence クライアント／サーバ・プロトコルを使用して 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。iAnywhere JDBC 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。

- **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 ドライバと jConnect のマニュアルは、http://www.sybase.com/products/informationmanagement/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**　iAnywhere JDBC ドライバは、JDBC 2.0 と 3.0 の両方に準拠しています。jConnect 5.5 は JDBC 2.0 に、jConnect 6.0.5 は JDBC 3.0 に準拠しています。iAnywhere 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、Adaptive Server Enterprise データベースに接続したときにのみスクロール可能なカーソルを使用できます。
- **Pure Java**　jConnect ドライバは pure Java ソリューションです。iAnywhere JDBC ドライバは、SQL Anywhere ODBC ドライバを必要とし、pure Java ソリューションではありません。
- **パフォーマンス**　ほとんどの用途で、iAnywhere 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。

iAnywhere 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、「プラットフォーム別 SQL Anywhere 10.0.0 コンポーネント」にある SQL Anywhere の表を参照してください。

jConnect の Windows CE での使用の詳細については、「Windows CE での jConnect の使用」『SQL Anywhere サーバ - データベース管理』を参照してください。

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、接続の確立に iAnywhere 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。

iAnywhere JDBC ドライバの使用

iAnywhere JDBC ドライバでは、pure Java である jConnect JDBC ドライバに比べて何らかの有利なパフォーマンスや機能を備えた JDBC ドライバが提供されます。ただし、このドライバは pure Java ソリューションではありません。iAnywhere JDBC ドライバは一般に推奨されるドライバです。

使用する JDBC ドライバの選択方法については、「JDBC ドライバの選択」 492 ページを参照してください。

iAnywhere JDBC ドライバのロード

アプリケーションで iAnywhere JDBC ドライバを使用する前に、適切なドライバをロードする必要があります。JDBC 3.0 バージョンの iAnywhere JDBC ドライバをロードするには、次の文を使用します。

```
DriverManager.registerDriver( (Driver)
   Class.forName(
   "ianywhere.ml.jdbcodbc.jdbc3.IDriver").newInstance()
   );
```

JDBC 2.0 バージョンの iAnywhere JDBC ドライバをロードするには、次の文を使用します。

```
DriverManager.registerDriver( (Driver)
   Class.forName(
   "ianywhere.ml.jdbcodbc.IDriver").newInstance()
   );
```

newInstance メソッドを使用して、一部のブラウザの問題を処理します。

- クラスが Class.forName を使用してロードされるため、import 文を使用して、iAnywhere JDBC ドライバを含むパッケージをインポートする必要はありません。
- アプリケーションを実行するとき、*jodbc.jar* がクラスパスにあることが必要です。

  ```
  set classpath=%classpath%;install-dir¥java¥jodbc.jar
  ```

必要なファイル

iAnywhere JDBC 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* サブディレクトリにインストールされている *jodbc.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 *win32* サブディレクトリの *dbjodbc10.dll* です。UNIX の場合、ネイティブ・コンポーネントは *dbjodbc10.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。このドライバを使用してアプリケーションを配備するときは、ODBC ドライバ・ファイルも配備します。

ドライバへの URL の指定

iAnywhere 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:ianywhere:DSN=SQL Anywhere 10 Demo" );
```

URL には、**jdbc:ianywhere:** の後に標準 ODBC 接続文字列が含まれています。接続文字列は通常は ODBC データ・ソースですが、データ・ソースの他に、またはデータ・ソースの代わりに、接続パラメータをセミコロンで区切って明示的に指定できます。接続文字列で使用できるパラメータの詳細については、「接続パラメータ」『SQL Anywhere サーバ - データベース管理』を参照してください。

データ・ソースを使用しない場合は、次のように接続文字列に DRIVER パラメータを指定することで、使用する ODBC ドライバを指定してください。

```
Connection con = DriverManager.getConnection(
    "jdbc:ianywhere:DRIVER=SQL Anywhere 10;..." );
```

jConnect JDBC ドライバの使用

SQL Anywhere では、jConnect 5.5 と jConnect 6.0.5 の 2 つのバージョンの jConnect をサポートしています。jConnect ドライバは、http://www.sybase.com/products/informationmanagement/softwaredeveloperkit/jconnect から別途ダウンロードできます。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

SQL Anywhere では次のバージョンの jConnect をサポートしています。

- **jConnect 5.5**
  このバージョンの jConnect は、JDK 1.2 アプリケーションの開発用です。jConnect 5.5 は JDBC 2.0 に準拠しており、*jconn2.jar* という JAR ファイルとして提供されます。
- **jConnect 6.0.5**
  このバージョンの jConnect は、JDK 1.3 以降のアプリケーションの開発用です。jConnect 6.0.5 は JDBC 3.0 に準拠しており、*jconn3.jar* という JAR ファイルとして提供されます。

注意
このマニュアルの目的上、ここに記載されている説明とコード・サンプルでは、JDK 1.5 アプリケーションの開発と jConnect 6.0.5 ドライバの使用を想定しています。

jConnect 用クラスパスの設定

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 6.0.5 ドライバを追加します。*path* は SQL Anywhere のインストール・ディレクトリです。

```
set classpath=%classpath%;path¥jConnect-6_0¥classes¥jconn3.jar
```

jConnect クラスのインポート

jConnect のクラスはすべて com.sybase パッケージにあります。

jConnect 6.0.5 を使用する場合、各クラスは com.sybase.jdbc3.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.jdbc3.jdbc.*
```

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。

jConnect システム・オブジェクトは、作成時、更新時、またはその後でもデータベースに追加できます。後で jConnect を新しいバージョンに更新する場合は、最新バージョンの *jcatalog.sql* を実行してください。

Sybase Central または Interactive SQL から、jConnect システム・オブジェクトをインストールします。

♦ データベースに jConnect システム・オブジェクトを追加するには、次の手順に従います (Sybase Central の場合)。

1. DBA ユーザとして Sybase Central からデータベースに接続します。
2. 左ウィンドウ枠でデータベースを選択し、[ファイル] メニューから [データベースのアップグレード] を選択します。

   [データベース・アップグレード] ウィザードが表示されます。
3. ウィザードの指示に従い、jConnect サポートをデータベースに追加します。

♦ データベースに jConnect システム・オブジェクトを追加するには、次の手順に従います (InteractiveSQL の場合)。

- DBA ユーザとして Interactive SQL からデータベースに接続し、次の文を実行します (*path* は SQL Anywhere インストール・ディレクトリです)。

```
READ path¥scripts¥jcatalog.sql
```

ヒント
次のコマンドを実行することで、コマンド・プロンプトから jConnect システム・オブジェクトをデータベースに追加することもできます。

```
dbisql -c "connection-string" install-dir¥scripts¥jcatalog.sql
```

このコマンドでは、*connection-string* は DBA ユーザがデータベースとサーバにアクセスするための接続文字列で、*install-dir* は SQL Anywhere のインストール・ディレクトリです。

jConnect ドライバのロード

アプリケーションで jConnect を使用する前に、次の文を入力してドライバをロードしてください。

```
DriverManager.registerDriver( (Driver)
   Class.forName(
   "com.sybase.jdbc3.jdbc.SybDriver").newInstance()
   );
```

newInstance メソッドを使用して、一部のブラウザの問題を処理します。

- クラスが Class.forName を使用してロードされるため、import 文を使用して、jConnect ドライバを含むパッケージをインポートする必要はありません。
- jConnect 5.5 を使用する場合は、アプリケーションを実行するときに *jconn2.jar* がクラスパスにあることが必要です。*jconn2.jar* は、jConnect 5.5 インストール環境の *classes* サブディレクトリにあります。
- jConnect 6.0.5 を使用する場合は、アプリケーションを実行するときに *jconn3.jar* がクラスパスにあることが必要です。*jconn3.jar* は、jConnect 6.0.5 インストール環境の *classes* サブディレクトリにあります。

ドライバへの URL の指定

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、Sybase 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。

サーバ上でのデータベースの指定

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;
.
.
.
DriverManager.registerDriver( (Driver)
   Class.forName(
   "com.sybase.jdbc3.jdbc.SybDriver").newInstance()
   );

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。

ユーティリティ・データベースの詳細については、「ユーティリティ・データベースの使用」『SQL Anywhere サーバ - データベース管理』を参照してください。

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、次の 2 つのストアド・プロシージャが呼び出されます。

1. 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。

2. sp_mda プロシージャが次に呼び出され、その他のオプションが設定されます。特に、sp_mda プロシージャによって quoted_identifier 設定が決定されます。デフォルトの動作を変更するには、*jcatalog.sql* の insert dbo.spt_mda values ... 文を変更してください。

参照

- 「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』
- 「quoted_identifier オプション [互換性]」『SQL Anywhere サーバ - データベース管理』

JDBC クライアント・アプリケーションからの接続

iAnywhere 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 ドライバの使用」 497 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・アプリケーションであり、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 クラスからの接続の確立」 505 ページを参照してください。

接続サンプルのコード

次に示すのは、接続の確立に使用するメソッドのソース・コードです。ソース・コードは、*samples-dir¥SQLAnywhere¥JDBC* ディレクトリの *JDBCConnect.java* ファイルにあります。この例では、iAnywhere JDBC ドライバの JDBC 2.0 バージョンを使用してデータベースに接続します (iAnywhere JDBC ドライバの JDBC 3.0 バージョンを使用する場合は、ianywhere.ml.jdbcodbc.IDriver を ianywhere.ml.jdbcodbc.jdbc3.IDriver に置き換えます)。jConnect 6.0.5 ドライバを使用する場合のコードは、ソース・コードにコメントとして記載されています。

```
import java.io.*;
import java.sql.*;

public class JDBCConnect
{
  public static void main( String args[] )
  {
    try
    {
      // Open the connection. May throw a SQLException.
      DriverManager.registerDriver( (Driver)
        Class.forName(
        // "com.sybase.jdbc3.jdbc.SybDriver").newInstance()
        "ianywhere.ml.jdbcodbc.IDriver").newInstance()
        );

      Connection con = DriverManager.getConnection(
          // "jdbc:sybase:Tds:localhost:2638", "DBA", "sql");
          "jdbc:ianywhere:driver=SQL Anywhere 10;uid=DBA;pwd=sql" );

      // 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・アプリケーションです。

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 Sun Microsystems io クラスが含まれていて、このクラスはコンソールへの出力に必要です。
- java.sql パッケージには Sun Microsystems 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. iAnywhere JDBC ドライバをロードします。jConnect 6.0.5 JDBC ドライバをロードするには、ドライバ文字列 "com.sybase.jdbc3.jdbc.SybDriver" (コメントを参照) を使用します。

   Class.forName がドライバをロードします。newInstance メソッドを使用して、一部のブラウザの問題を処理します。

2. iAnywhere 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。jConnect ドライバを使用している場合は、URL "jdbc:sybase:Tds:localhost:2638" (コメントを参照) を使用し、ユーザ ID とパスワードにそれぞれ "DBA" と "sql" を指定します。

   DriverManager.getConnection が指定された URL を使用して接続を確立します。

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

♦ **外部接続サンプルのアプリケーションを作成して実行するには、次の手順に従います。**

1. コマンド・プロンプトで、*samples-dir¥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。

   ```
   dbeng10 samples-dir¥demo.db
   ```

3. CLASSPATH 環境変数を設定します。

   ```
   set classpath=%classpath%;install-dir¥java¥jodbc.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=%classpath%;path¥jconnect-6_0¥classes¥jconn3.jar;
   ```

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 JDBCConnect.java
   ```

5. 次のコマンドを入力してサンプルを実行します。

   ```
   java JDBCConnect
   ```

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。

サーバ側 JDBC クラスからの接続の確立

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、*samples-dir¥SQLAnywhere¥JDBC¥JDBCConnect.java* にあるソース・コードの変更版です。

```
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. ドライバ・マネージャはロードする必要はありません。次のコードはサンプルから削除されています。

   ```
   DriverManager.registerDriver( (Driver)
     Class.forName(
     // "com.sybase.jdbc3.jdbc.SybDriver").newInstance()
     "ianywhere.ml.jdbcodbc.IDriver").newInstance()
     );
   ```

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

♦ **内部接続サンプルのアプリケーションを作成して実行するには、次の手順に従います。**

1. コマンド・プロンプトで、*samples-dir¥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。

   ```
   dbeng10 samples-dir¥demo.db
   ```

3. サーバ側の JDBC の場合、CLASSPATH 環境変数を設定する必要はありません。
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。

   ```
   INSTALL JAVA NEW
   FROM FILE 'samples-dir¥SQLAnywhere¥JDBC¥JDBCConnect2.class'
   ```

   Sybase Central を使用してもクラスをインストールできます。サンプル・データベースとの接続中に、[Java オブジェクト] フォルダを開いて [ファイル] – [新規] – [Java クラス] を選択します。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()
  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。

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修正文が実行されると、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。

- **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、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」 511 ページで説明します。

サンプルの準備

サンプル・コード

こ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 *samples-dir¥SQLAnywhere¥JDBC¥JDBCExample.java* から引用しています。

♦ **JDBCExamples クラスをインストールするには、次の手順に従います。**

1. *JDBCExample.java* ソース・コードをコンパイルします。
2. Interactive SQL を使用して、DBA としてサンプル・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 (*samples-dir* は SQL Anywhere サンプル・ディレクトリ)。

```
INSTALL JAVA NEW
FROM FILE 'samples-dir¥SQLAnywhere¥JDBC¥JDBCExample.class'
```

Sybase Central を使用してもクラスをインストールできます。サンプル・データベースとの接続中に、[Java オブジェクト] フォルダを開いて [ファイル] – [新規] – [Java クラス] を選択します。ウィザードの指示に従います。

JDBC を使用した挿入、更新、削除

Statement オブジェクトは、静的 SQL 文を実行します。INSERT、UPDATE、DELETE など結果セットを返さない SQL 文の実行には、Statement オブジェクトの 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。

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();
  }
}
```

利用可能なソース・コード
このコード・フラグメントは、*samples-dir¥SQLAnywhere¥JDBC* ディレクトリに含まれている JDBCExample クラスの一部です。

注意

- 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 の出力結果はサーバ・メッセージ・ウィンドウに表示されます。

♦ JDBC Insert のサンプルを実行するには、次の手順に従います。

1. Interactive SQL を使用して、DBA としてサンプル・データベースに接続します。
2. JDBCExample クラスがインストールされていることを確認します。

   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、「サンプルの準備」 509 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サーバ・メッセージ・ウィンドウに表示します。

より効率的なアクセスのために準備文を使用する

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。実際に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。

準備文の詳細については、「文の準備」 28 ページを参照してください。

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
    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();
      }
    }
```

利用可能なソース・コード
このコード・フラグメントは、*samples-dir¥SQLAnywhere¥JDBC* ディレクトリに含まれている JDBCExample クラスの一部です。

注意

- 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 の出力結果はサーバ・メッセージ・ウィンドウに表示されます。

♦ JDBC Insert のサンプルを実行するには、次の手順に従います。

1. Interactive SQL を使用して、DBA としてサンプル・データベースに接続します。
2. JDBCExample クラスがインストールされていることを確認します。

   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、「サンプルの準備」 509 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サーバ・メッセージ・ウィンドウに表示します。

結果セットを返す

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呼び出し元の SQL コード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();
    }
```

利用可能なソース・コード
このコード・フラグメントは、*samples-dir¥SQLAnywhere¥JDBC* ディレクトリに含まれている JDBCExample クラスの一部です。

注意

- このサーバ側の JDBC サンプルでは、getConnection を使用して、現在の接続を使用して実行されているデフォルトのデータベースに接続します。
- executeQuery メソッドによって結果セットが返されます。

♦ JDBC 結果セットのサンプルを実行するには、次の手順に従います。

1. Interactive SQL を使用して、DBA としてサンプル・データベースに接続します。
2. JDBCExample クラスがインストールされていることを確認します。

   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、「サンプルの準備」 509 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。

   ```
   CREATE PROCEDURE JDBCResults()
     DYNAMIC RESULT SETS 3
     EXTERNAL NAME 'JDBCExample.Results([Ljava/sql/ResultSet;)V'
     LANGUAGE JAVA;
   ```
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] メニューから [オプション] を選択します。

      [オプション] ダイアログが表示されます。

   b. [結果] をクリックします。

   c. [表示できるローの最大数] の値を **5000** に設定します。

d. [複数の結果セットを表示] を選択します。

e. [OK] をクリックします。

5. 次のように 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。

JDBC に関する各種注意事項

- **アクセス・パーミッション** 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 メソッドのラッパとして動作するストアド・プロシージャを実行するパーミッションが GRANT EXECUTE 文で付与されているどのユーザもアクセスできます。
- **実行パーミッション** Java クラスは、そのクラスを実行する接続のパーミッションによって実行されます。この動作は、所有者のパーミッションによって実行されるストアド・プロシージャの動作とは異なります。

JDBC エスケープ構文の使用

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* }}

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 で実行されないため、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。

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ユーザの名前を取得できます。

```
SELECT {{ FN USER() }}
```

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、iAnywhere JDBC と jConnect ドライバによってサポートされている関数のリストです。

iAnywhere 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	IFNULL	CURDATE
ACOS	CHAR	USERNAME	CURTIME
ASIN	CONCAT		DAYNAME
ATAN	DIFFERENCE		DAYOFMONTH
ATAN2	INSERT		DAYOFWEEK
CEILING	LCASE		DAYOFYEAR
COS	LEFT		HOUR
COT	LENGTH		MINUTE
DEGREES	LOCATE		MONTH
EXP	LOCATE_2		MONTHNAME
FLOOR	LTRIM		NOW
LOG	REPEAT		QUARTER
LOG10	RIGHT		SECOND

数値関数	文字列関数	システム関数	日付／時刻関数
MOD	RTRIM		WEEK
PI	SOUNDEX		YEAR
POWER	SPACE		
RADIANS	SUBSTRING		
RAND	UCASE		
ROUND			
SIGN			
SIN			
SQRT			
TAN			
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

数値関数	文字列関数	システム関数	日付／時刻関数
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。

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 ) }}
```

第 13 章

SQL Anywhere Embedded SQL

目次

Embedded SQL の概要

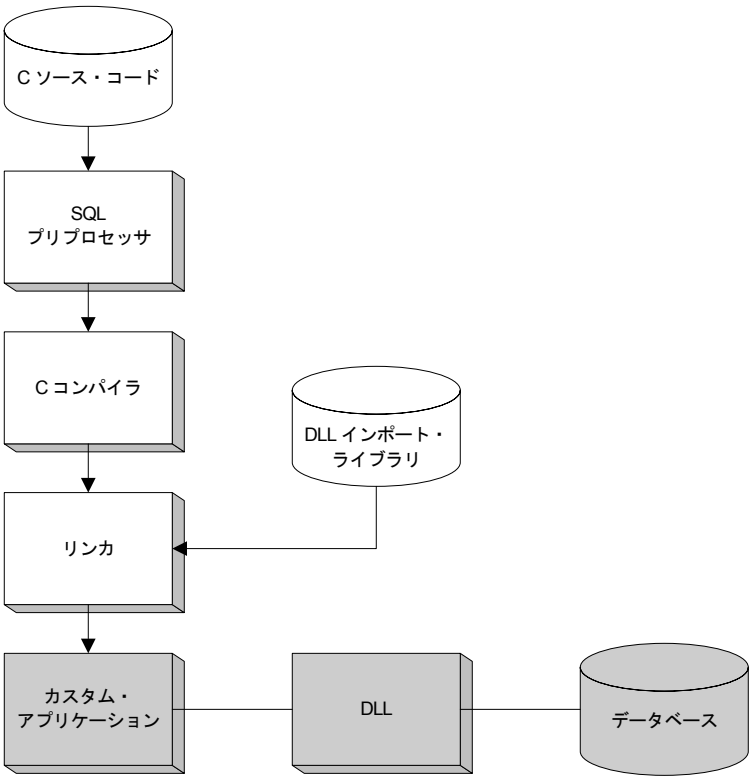
Embedded SQL は、C と C++ プログラミング言語用のデータベース・プログラミング・インタフェースです。Embedded SQL は、C と C++ のソース・コードが混合された (埋め込まれた) SQL 文で構成されます。この SQL 文は「**SQL プリプロセッサ**」によって C または C++ のソース・コードに翻訳され、その後ユーザによってコンパイルされます。

実行時に、Embedded SQL アプリケーションは 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。インタフェース・ライブラリは、ほとんどのプラットフォームで、ダイナミック・リンク・ライブラリ (**DLL**) または共有ライブラリです。

- 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 *dblib10.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、*libdblib10.so*、*libdblib10.sl,*、または *libdblib10.a* です。
- Mac OS X では、インタフェース・ライブラリは *libdblib10.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 SQL Anywhere インタフェース・ライブラリ用の「**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 SQL Anywhere の DLL 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、Watcom C/C++、Microsoft Visual C++、Borland C++ の各コンパイラ用にインポート・ライブラリが用意されています。

DLL 内の関数を呼び出すアプリケーションはインポート・ライブラリを使用して開発するのが標準的な方法です。しかし、SQL Anywhere には、インポート・ライブラリを使用しない開発方法も用意されており、こちらの方がおすすめです。詳細については、「インタフェース・ライブラリの動的ロード」 525 ページを参照してください。

SQL プリプロセッサの実行

SQL プリプロセッサの実行プログラム名は *sqlpp.exe* です。

SQLPP のコマンド・ラインを次に示します。

sqlpp [*options*] *sql-filename* [*output-filename*]

SQL プリプロセッサが ESQL を含んだ C プログラムの処理を行ってから、C または C++ コンパイラが実行されます。プリプロセッサは SQL 文を C/C++ 言語のソースに翻訳し、ファイルに出力します。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、出力ファイル拡張子はデフォルトで *.cc* になります。

コマンド・ライン・オプションの一覧については、「SQL プリプロセッサ」 581 ページを参照してください。

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	Watcom C/C++	9.5 以上
Windows	Microsoft Visual C++	6.0 以上
Windows	Borland C++	4.5
Windows CE	Microsoft Visual C++	2005
Windows CE	Microsoft eMbedded Visual C++	3.0, 4.0
UNIX	GNU またはネイティブ・コンパイラ	
NetWare	Watcom C/C++	10.6, 11

NetWare NLM の構築については、「NetWare Loadable Module の構築」 526 ページを参照してください。

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 *h* サブディレクトリにインストールされています。

ファイル名	説明
sqlca.h	メイン・ヘッダ・ファイル。すべての Embedded SQL プログラムにインクルードされる。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む。
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る。

ファイル名	説明
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・コードも含む。
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, pshpk2.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

インポート・ライブラリはすべて、SQL Anywhere インストール・ディレクトリのオペレーティング・システム別サブディレクトリにある *lib* サブディレクトリにインストールされています。たとえば、Windows 用のインポート・ライブラリは *win32¥lib*、*x64¥lib*、*ia64¥lib* の各サブディレクトリに格納されています。Windows CE 用のインポート・ライブラリは、各プラットフォーム依存のディレクトリ (たとえば *ce¥arm.50*) の *lib* サブディレクトリにインストールされます。

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	Watcom C/C++ (32 ビットのみ)	*dblibtw.lib*
Windows	Microsoft Visual C++	*dblibtm.lib*
Windows	Borland Delphi (32 ビットのみ)	*dblibtb.lib*
Windows CE	Microsoft Visual C++ 2005	*dblib10.lib*
Windows CE	Microsoft eMbedded Visual C++	*dblib10.lib*
NetWare	Watcom C/C++	*dblib10.lib*
Solaris (非スレッド・アプリケーション)	全コンパイラ	*libdblib10.so, libdbtasks10.so*
Solaris (スレッド・アプリケーション)	全コンパイラ	*libdblib10_r.so, libdbtasks10_r.so*

libdbtasks10 ライブラリは、*libdblib10* ライブラリに呼び出されます。コンパイラによっては、*libdbtasks10* を自動的に見つけるものもありますが、ユーザによる明示的な指定が必要なものもあります。

簡単な例

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。SQL と C コードの間では事実上やりとりはありません。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。

データのフェッチについては、「データのフェッチ」 563 ページを参照してください。

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;
```

C プログラムで実行する最初の ESQL 文は CONNECT 文にします。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。

ESQL コマンドには 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。このようなコマンドは 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。

インタフェース・ライブラリの動的ロード

DLL 内の関数を使用するアプリケーションの開発では、必要な関数定義のはいった「**インポート・ライブラリ**」とアプリケーションをリンクするのが一般的な方法です。

この項ではインポート・ライブラリを使わないで SQL Anywhere アプリケーションを開発する方法を説明します。SQL Anywhere インタフェース・ライブラリは、インポート・ライブラリとリンクしなくても、動的にロードできます。これには、インストール・ディレクトリの *src* サブディレクトリにある *esqldll.c* モジュールを使用します。

♦ **インタフェース DLL を動的にロードするには、次の手順に従います。**

1. プログラムでは、db_init_dll を呼び出して DLL をロードし、db_fini_dll を呼び出して 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 #include 指定するか、#include <*sqlca.h*> 行を追加してください。

3. SQL の OS マクロを定義します。*esqdll.c* にインクルードされるヘッダ・ファイル *sqlca.h* は、適切なマクロを判断して、定義しようとします。しかし、プラットフォームとコンパイラの組み合わせによっては、定義に失敗することがあります。その場合は、このヘッダ・ファイルの先頭に #define を追加するか、コンパイラ・オプションを使用してマクロを定義するかをしてください。

マクロ	プラットフォーム
_SQL_OS_NETWARE	NetWare
_SQL_OS_UNIX	UNIX
_SQL_OS_UNIX64	64 ビット UNIX
_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。

サンプル

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、*samples-dir¥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。

NetWare Loadable Module の構築

Embedded SQL プログラムを NetWare Loadable Module (NLM) としてコンパイルするには、Watcom C/C++ コンパイラのバージョン 10.6 または 11.0 を使用してください。

♦ **Embedded SQL NLM を作成するには、次の手順に従います。**

1. Windows では、次のコマンドを使用して Embedded SQL ファイルを前処理します。

   ```
   sqlpp -o NETWARE srcfile.sqc
   ```

   この命令は、拡張子 *.c* の付いたファイルを作成します。

2. *file.c* を Watcom コンパイラ (10.6 または 11.0) で /bt=netware オプションを使用してコンパイルします。

3. その結果できたオブジェクト・ファイルを Watcom リンカで以下のオプションを使用してリンクさせます。

   ```
   FORMAT NOVELL
   MODULE dblib10
   OPTION CASEEXACT
   IMPORT @dblib10.imp
   LIBRARY dblib10.lib
   ```

   dblib10.imp ファイルと *dblib10.lib* ファイルは、SQL Anywhere に付属しており、*nlm¥lib* ディレクトリに入っています。IMPORT 行と LIBRARY 行にはフル・パスが必要な場合があります。

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 *samples-dir¥SQLAnywhere¥C* ディレクトリにあります。Windows CE の場合は、追加サンプルが *samples-dir¥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 ESQL の CONNECT コマンドによって適切なユーザ ID とパスワードを与えて実行します。

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。

サンプル・プログラムの構築

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 オペレーティング・システムと Windows オペレーティング・システムで実行されている NetWare オペレーティング・システムの場合は、*makeall.bat* を使用してサンプル・プログラムをコンパイルします。

- UNIX 環境では、シェル・スクリプトの *makeall* を使用してください。
- Windows CE 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 *samples-dir¥SQLAnywhere¥CE¥esql_sample* にあります。

コマンドのフォーマットを次に示します。

```
makeall {Example} {Platform} {Compiler}
```

最初のパラメータはコンパイルするサンプル・プログラムの名前です。次のいずれかを指定してください。

- **CUR** 静的カーソルのサンプル
- **DCUR** 動的カーソルのサンプル
- **ODBC** ODBC のサンプル

2 番目のパラメータはターゲット・プラットフォームです。次のいずれかを指定してください。

- **WINDOWS** Windows 用にコンパイル
- **WINIA64** Windows 用にコンパイル
- **WINX64** Windows 用にコンパイル
- **NETWARE** NetWare NLM 用にコンパイル
- **UNIX** UNIX 用にコンパイル
- **UNIX64** 64 ビット UNIX 用にコンパイル

3 番目のパラメータは、プログラムのコンパイルに使用するコンパイラです。コンパイラは次のいずれかを指定してください。

- **WC** Watcom C/C++ を使用
- **MC** Microsoft C/C++ を使用
- **BC** Borland C++ を使用

x64 と IA64 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。x64 プラットフォーム用の動的カーソルのサンプルをビルドするコマンド例を次に示します。

```
c:¥x64sdk¥SetEnv /XP64
makeall dcur winx64 mc
```

サンプル・プログラムの実行

実行ファイルと対応するソース・コードは *samples-dir¥SQLAnywhere¥C* ディレクトリにあります。Windows CE の場合は、追加サンプルが *samples-dir¥SQLAnywhere¥CE¥esql_sample* ディレクトリにあります。

♦ **静的カーソルのサンプル・プログラムを実行するには、次の手順に従います。**

1. SQL Anywhere サンプル・データベース *demo.db* を起動します。
2. ファイル *curwnt.exe* を実行します。
3. 画面に表示される指示に従います。

   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・プログラムを実行するには、次の手順に従います。**

1. ファイル *dcurwnt.exe* を実行します。
2. 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 10 Demo
   ```

3. 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** と入力します。
4. 画面に表示される指示に従います。

   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。

Windows のサンプル

Windows 版のサンプル・プログラムでは、Windows のグラフィカル・ユーザ・インタフェースを使用します。しかし、ユーザ・インタフェース用のコードを単純にするために、いくつか処理を簡略化しています。特に、これらのプログラムは、プロンプトを再表示するとき以外、WM_PAINT メッセージで自分のウィンドウを再描画しません。

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。「動的カーソルのサンプル」 530 ページを参照してください。

ソース・コードのある場所とサンプル・プログラムの作成方法については、「サンプル Embedded SQL プログラム」 527 ページを参照してください。

open_cursor ルーチンは、指定の SQL コマンド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・ルーチンが警告条件 (「**ローが見つかりません**」など) を検査し、適切なメッセージを表示することに注意してください。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ないことに注意してください。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。

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。これは静的カーソルのサンプルに少し手を加えたものです。静的カーソルのサンプルをまだ見ていない場合は、先にそれを確認すると、このサンプルの理解に役立つでしょう。「静的カーソルのサンプル」 529 ページを参照してください。

ソース・コードのある場所とサンプル・プログラムの作成方法については、「サンプル Embedded SQL プログラム」 527 ページを参照してください。

dcur プログラムでは、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形式の接続文字列を入力してください。

```
UID=DBA;PWD=sql;DBF=samples-dir¥demo.db
```

Embedded SQL を使用する C プログラムは、*samples-dir¥SQLAnywhere¥C* ディレクトリにあります。Windows CE の場合は、動的カーソルのサンプルが *samples-dir¥SQLAnywhere¥CE¥esql_sample* ディレクトリにあります。プログラムは、connect、open_cursor、print 関数を除いて、静的カーソルのサンプルとほぼ同じです。

connect 関数は 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。

ESQL の DESCRIBE コマンドは、SELECT 文の結果を SQLDA 構造体に設定するために使用されます。

SQLDA のサイズ
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 ルーチンは少し違います。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。

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。ESQL 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**
  16 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る。

- **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る。
- **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 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 {
    unsigned short int 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 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 {
    unsigned short int 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。

詳細については、「長い値の送信と取り出し」 572 ページを参照してください。

- **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。

  詳細については、「長い値の送信と取り出し」 572 ページを参照してください。

- **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 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ます。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 {
  unsigned short int 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。

詳細については、「長い値の送信と取り出し」 572 ページを参照してください。

- **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、プログラマは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 メンバは無視されます。

  次の項を参照してください。

  - 「date_format オプション [互換性]」 『SQL Anywhere サーバ - データベース管理』
  - 「date_order オプション [互換性]」 『SQL Anywhere サーバ - データベース管理』
  - 「time_format オプション [互換性]」 『SQL Anywhere サーバ - データベース管理』
  - 「timestamp_format オプション [互換性]」 『SQL Anywhere サーバ - データベース管理』

- **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が、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用です。

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。

詳細については、「GET DATA 文 [ESQL]」 『SQL Anywhere サーバ - SQL リファレンス』と「SET 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

ホスト変数の使用

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。

バッチにホスト変数を使用することはできません。

動的 SQL の詳細については、「静的 SQL と動的 SQL」 550 ページを参照してください。

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・サーバがコマンドを実行する場合は、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ないことに注意してください。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 の外では C 言語コードをスキャンしません。したがって、DECLARE SECTION 内では TYPEDEF 型と構造体は使用できません。変数の初期化は DECLARE SECTION 内でもできます。

例

INSERT コマンド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 );
```

さらに複雑な例については、「静的カーソルのサンプル」 529 ページを参照してください。

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、FIXCHAR、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 バイト浮動小数点数
double d;	DT_DOUBLE	8 バイト浮動小数点数

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]; /*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 文字と 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 文字と 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 文字と NULL ターミネータが保持されます。
DECL_NCHAR *a;	DT_NSTRING	NCHAR 文字セット内の NULL で終了する文字列。この変数は、最大 32766 文字と 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 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 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。

C データ型	Embedded SQL のインタフェースのデータ型	説明
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 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。
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 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 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。

ロケールと文字セットの詳細については、「ロケールの知識」 『SQL Anywhere サーバ - データベース管理』を参照してください。

CHAR 文字セットの変更の詳細については、「CharSet 接続パラメータ [CS]」 『SQL Anywhere サーバ - データベース管理』または「db_change_char_charset 関数」 590 ページを参照してください。

NCHAR 文字セットの変更の詳細については、「db_change_nchar_charset 関数」 591 ページを参照してください。

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みなされます。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。

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、ESQL に特有のコマンド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、ホスト変数はユーザ ID、オプション名、オプション値として使用できます。

- 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。

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。

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。

詳細については、「SQL プリプロセッサ」 581 ページの sqlpp -k オプションの説明を参照してください。

次に示すのは、有効な ESQL です。

```
EXEC SQL INCLUDE SQLCA;
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 ...
}
```

次に示す ESQL は、有効ではありません。

```
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...
   // No SQLSTATE in scope of this statement
}
```

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 short int 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 );
```

NULL を扱うためのインジケータ変数

SQL データでは、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。同じ呼び方 (NULL) であるからといって、SQL での NULL を C 言語の定数と混同しないでください。C の定数の場合は、初期化されていないか不正なポインタを表すために使用されます。

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];
short int ind_phone;
EXEC SQL END DECLARE SECTION;

/*
program fills in empnum, empname,
initials and homephone
*/
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)。

トランケートされた値に対するインジケータ変数

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。値の長さが 32767 を超える場合は、インジケータ変数は 32767 になります。

変換エラーの場合のインジケータ変数

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
−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

長い値の取得の詳細については、「GET DATA 文 [ESQL]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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 *h* サブディレクトリにある *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役立ちます。
- **sqlcabc**
  long integer。SQLCA 構造体の長さ (136 バイト) が入ります。
- **sqlcode**
  long integer。データベースが検出した要求エラーのエラー・コードが入ります。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。

  エラー・コードの一覧については、SQL Anywhere 10 - エラー・メッセージ『SQL Anywhere 10 - エラー・メッセージ』を参照してください。
- **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。

エラー・メッセージの一覧については、SQL Anywhere 10 - エラー・メッセージ『SQL Anywhere 10 - エラー・メッセージ』を参照してください。

- **sqlerrp**
  予約。
- **sqlerrd**
  long integer 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 ー 9 の数字、または、A ー Z の英大文字です。

  0 ー 4 または A ー H の文字で始まるクラス、サブクラスはすべて SQL 規格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。

  SQLSTATE 値の詳細については、SQL Anywhere 10 - エラー・メッセージ『SQL Anywhere 10 - エラー・メッセージ』を参照してください。

sqlerror 配列

sqlerror フィールドの配列要素を次に示します。

- **sqlerrd[1] (SQLIOCOUNT)**　コマンド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・サーバは、コマンド実行ごとにこの値を 0 に設定することはしません。プログラムでこの変数を 0 に設定してから、一連のコマンドを実行してもかまいません。最後のコマンドが実行された後、この値は一連のコマンド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。

    カーソルを開いたとき、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。

ワイド・フェッチの詳細については、「一度に複数のローをフェッチする」 567 ページを参照してください。

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すると、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呼び出しを再度記述してください。
  - **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。

  構文エラーの SQLE_SYNTAX_ERROR の場合、このフィールドにはコマンド文字列内のおおよそのエラー検出位置が入ります。

- **sqlerrd[3] (SQLIOESTIMATE)** コマンド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 または EXPLAIN コマンドによって値が設定されます。

マルチスレッドまたは再入可能コードでの SQLCA 管理

ESQL 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 を用意してください。

1 つのデータベース接続には 1 つの SQLCA からのみアクセスできます。キャンセル命令が出た場合は例外ですが、この命令は別のスレッドから発行します。

キャンセル要求については、「要求管理の実装」 580 ページを参照してください。

次はマルチスレッド Embedded SQL の再入可能コードの例です。

```
#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 );
}
```

複数の SQLCA の使用

♦ **アプリケーションで複数の SQLCA を管理するには、次の手順に従います。**

1. SQL プリプロセッサで、再入力不可コード (-r-) を生成するオプション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。
2. 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。

   警告
   NetWare 上では各 db_init に対応する db_fini を呼び出さないと、データベース・サーバと NetWare ファイル・サーバが失敗することがあります。

3. 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、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 参照がタスク独自のデータを指すようにします。この文はコードをまったく生成しないので、パフォーマンスに

影響を与えません。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。

SQLCA の作成については、「SET SQLCA 文 [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

複数の SQLCA を使用する場合

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 SQLCA を使用する必要があるのは次のような環境の場合です。

- **マルチスレッド・アプリケーション** 各スレッドには専用の SQLCA が必要です。これは、ESQL を使用する DLL があっ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場合にも発生することがあります。
- **ダイナミック・リンク・ライブラリと共有ライブラリ** 1 つの DLL に与えられるデータ・セグメントは 1 つだけです。データベース・サーバが 1 つのアプリケーションからの要求を処理している間に、データベース・サーバに要求する別のアプリケーションに渡すことがあります。DLL がグローバル SQLCA を使用する場合は、両方のアプリケーションがその SQLCA を同時に使用します。各 Windows アプリケーションは専用の SQLCA を使用できる必要があります。
- **1 つのデータ・セグメントを持つ DLL** DLL はデータ・セグメントを 1 つだけ持つように作成したり、アプリケーションごとに 1 つのデータ・セグメントを持つように作成したりできます。使用する DLL のデータ・セグメントが 1 つだけの場合は、1 つの DLL がグローバル SQLCA を使用することはできないという同じ理由によってグローバル SQLCA を使用することはできません。各アプリケーションには専用の SQLCA が必要です。

複数の SQLCA を使用する接続管理

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。「SET CONNECTION statement [Interactive SQL] [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

レコード・ロック
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。ロックの詳細については、「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。

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

ここまでは、静的 SQL について説明してきました。この項では、静的 SQL と動的 SQL を比較します。

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、前に EXEC SQL を付け、コマンドの後ろにセミコロン (;) を付けて、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。ここまでの例はすべて静的 ESQL 文を使用しています。「ホスト変数の使用」 536 ページを参照してください。

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。

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 コマンドの 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、準備されたコマンド文字列内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。

SQLDA については、「SQLDA (SQL descriptor area)」 554 ページ を参照してください。

「**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

次に例を示します。

```
EXEC SQL BEGIN DECLARE SECTION;
char comm[200];
```



```
char Street[30];
char City[20];
short int cityind;
long empnum;
EXEC SQL END DECLARE SECTION;


. . .
sprintf( comm,
   "UPDATE %s SET Street = :?, City = :?"
            "WHERE employee_number = :?",
            tablename );
EXEC SQL PREPARE S1 FROM :comm;
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 コマンドの 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 EmployeeNumber = :empnum",
         tablename );
EXEC SQL PREPARE S1 FROM :comm;
/* 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 USING DESCRIPTOR 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

SQLDA 構造体の詳細については、「SQLDA (SQL descriptor area)」 554 ページを参照してください。

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。

詳細については、「インジケータ変数」 541 ページを参照してください。

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 C プログラマは select リスト項目の数を知らないので、たいていは 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
  USING DESCRIPTOR 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
        USING DESCRIPTOR 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;
```

使用後に文を削除する
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 select 文でのカーソルの完全な使用例については、「動的カーソルのサンプル」 530 ページを参照してください。

この例で取り上げた関数の詳細については、「ライブラリ関数のリファレンス」 585 ページを参照してください。

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。この構造体で、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やりとりします。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・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。詳細については、「ライブラリ関数のリファレンス」 585 ページを参照してください。

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。

SQLDA ヘッダ・ファイル

sqlda.h の内容を、次に示します。

```
#ifndef _SQLDA_H_INCLUDED
#define _SQLDA_H_INCLUDED
#define II_SQLDA

#include "sqlca.h"

#if defined( _SQL_PACK_STRUCTURES )
#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 */
  short int  sqllen;  /* length of host variable */
  void       *sqldata; /* address of variable */
  short int *sqlind;  /* indicator variable pointer */
  struct sqlname  sqlname;
};

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 */
};


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"
#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役立ちます。
sqldabc	long integer。SQLDA 構造体の長さが入る。
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 文によって設定される。データベース・サーバにデータを渡すときにプログラマが設定することもある。
sqlvar	struct sqlvar 型の記述子の配列。各要素がホスト変数を記述する。

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。「Embedded SQL のデータ型」 532 ページを参照してください。

  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。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。

長さフィールドの詳細については、「SQLDA の sqllen フィールドの値」 557 ページを参照してください。

- **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。

  格納フォーマットについては、「Embedded SQL のデータ型」 532 ページを参照してください。

  UPDATE、INSERT コマンドでは、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 short int 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。「conversion_error オプション [互換性]」 『SQL Anywhere サーバ - データベース管理』を参照してください。

  詳細については、「インジケータ変数」 541 ページを参照してください。

  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 コマンドでは、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。

SQLDA の sqllen フィールドの値

SQLDA における sqlvar 構造体の sqllen フィールドの長さは、データベース・サーバとの次のやりとりで使用されます。

- **値の記述**　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。「値の記述」 557 ページを参照。
- **値の取り出し**　データベースから値を取り出します。「値の取り出し」 560 ページを参照。
- **値の送信**　情報をデータベースに送信します。「値の送信」 559 ページを参照。
- この項ではこれらのやりとりについて説明します。

次の 3 つの表でそれぞれのやりとりの詳細を示します。これらの表は、*sqldef.h* ヘッダ・ファイルにあるインタフェース定数型 (**DT_** 型) を一覧にしています。この定数は SQLDA の sqltype フィールドで指定します。

sqltype フィールドの値については、「Embedded SQL のデータ型」 532 ページを参照してください。

静的 SQL でも SQLDA は使用されますが、この場合、SQL プリプロセッサが SQLDA を生成し、完全に設定します。静的 SQL の場合、これらの表は、静的 C ホスト変数型とインタフェース定数の対応を示します。

値の記述

次の表は、データベースのさまざまな型に対して DESCRIBE コマンド (SELECT LIST と BIND VARIABLE の両方) が返す sqllen と sqltype 構造体のメンバの値を示します。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。

ESQL データ型については、「Embedded SQL のデータ型」 532 ページを参照してください。

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
BIGINT	DT_BIGINT	8
BINARY(n)	DT_BINARY	n
BIT	DT_BIT	1
CHAR(n)	DT_FIXCHAR	n
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高位バイトが p に、低位バイトが s に設定される
DOUBLE	DT_DOUBLE	8
FLOAT	DT_FLOAT	4
INT	DT_INT	4
LONG BINARY	DT_LONGBINARY	32767
LONG NVARCHAR	DT_LONGNVARCHAR[1]	32767
LONG VARCHAR	DT_LONGVARCHAR	32767
NCHAR(n)	DT_NFIXCHAR[1]	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
NVARCHAR(n)	DT_NVARCHAR[1]	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
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

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
UNSIGNED SMALLINT	DT_UNSSMALLINT	2
VARCHAR(n)	DT_VARCHAR	n

[1] 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。「db_change_nchar_charset 関数」 591 ページを参照してください。

値の送信

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フォーマット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 ¥0 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。「LONG データの送信」 575 ページを参照。
DT_LONGNVARCHAR	長さフィールドが無視される。「LONG データの送信」 575 ページを参照。
DT_LONGVARCHAR	長さフィールドが無視される。「LONG データの送信」 575 ページを参照。
DT_NFIXCHAR(n)	SQLDA の長さフィールドが文字列の長さを決定する

Embedded SQL のデータ型	長さを設定するプログラム動作
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 ¥0 によって長さが決まる
DT_TIMESTAMP	末尾の ¥0 によって長さが決まる
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

値の取り出し

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。値は現在の日付フォーマット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

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_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設定
DT_BIT	動作不要	動作不要
DT_DATE	バッファの長さ	文字列末尾に ¥0
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
DT_LONGBINARY	長さフィールドが無視される。「LONG データの取り出し」 573 ページを参照。	長さフィールドが無視される。「LONG データの取り出し」 573 ページを参照してください。
DT_LONGNVARCHAR	長さフィールドが無視される。「LONG データの取り出し」 573 ページを参照してください。	長さフィールドが無視される。「LONG データの取り出し」 573 ページを参照してください。
DT_LONGVARCHAR	長さフィールドが無視される。「LONG データの取り出し」 573 ページを参照してください。	長さフィールドが無視される。「LONG データの取り出し」 573 ページを参照してください。
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	文字列末尾に ¥0
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設定
DT_SMALLINT	動作不要	動作不要
DT_STRING	バッファの長さ	文字列末尾に ¥0
DT_TIME	バッファの長さ	文字列末尾に ¥0
DT_TIMESTAMP	バッファの長さ	文字列末尾に ¥0
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

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_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設定

データのフェッチ

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。「ローを返さないか、1 つだけ返す SELECT 文」 563 ページを参照してください。
- **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。「ESQL でのカーソルの使用」 564 ページを参照してください。

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は、警告 (**ローが見つかりません**) が返ります。エラーと警告は、SQLCA 構造体で返されます。「SQLCA (SQL Communication Area)」 544 ページを参照してください。

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];
short int     ind_birthdate;
EXEC SQL END DECLARE SECTION;
. . .
int find_employee( long Employees )
{
    ID = Employees;

    EXEC SQL    SELECT GivenName ||
                ' ' || Surname, Sex, BirthDate
                INTO :name, :Sex,
                     :birthdate:ind_birthdate
                FROM Employees
                WHERE EmployeeID = :ID;
    if( SQLCODE == SQLE_NOTFOUND )
  {
        return( 0 ); /* Employees not found */
    }
  else if( SQLCODE < 0 )
```

```
    {
        return( -1 ); /* error */
    }
  else
  {
        return( 1 ); /* found */
    }
}
```

ESQL でのカーソルの使用

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。

カーソルの概要については、「カーソルを使用した操作」 34 ページを参照してください。

♦ **Embedded SQL でカーソルを管理するには、次の手順に従います。**

1. DECLARE 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。「SQLCA (SQL Communication Area)」 544 ページを参照してください。
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];
  short int 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 )
```



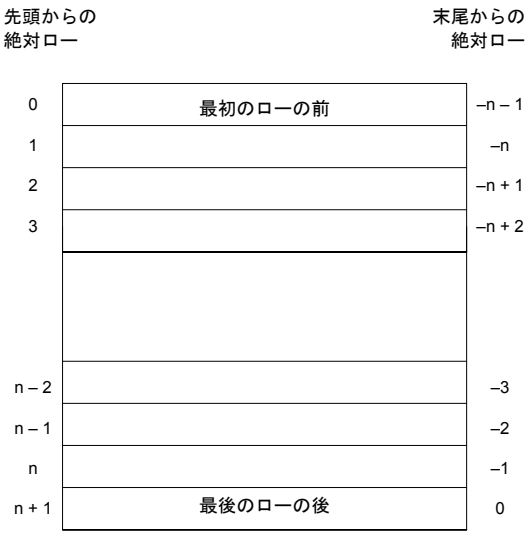
```
        {
                break;
            }
        else if( SQLCODE < 0 )
        {
                break;
            }

        if( ind_birthdate < 0 )
        {
                strcpy( birthdate, "UNKNOWN" );
            }
            printf( "Name: %s Sex: %c Birthdate:
                        %s.n",name, Sex, birthdate );
        }
      EXEC SQL CLOSE C1;
    }
```

カーソル使用の完全な例については、「静的カーソルのサンプル」 529 ページと「動的カーソルのサンプル」 530 ページを参照してください。

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 コマンド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。「FETCH 文 [ESQL] [SP]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。このカーソルが最初のローの前、または、最後のローの後にある場合、「**カーソルの現在のローがありません**」というエラーが返ります。

PUT 文で、カーソルにローを挿入できます。「PUT 文 [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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、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。

詳細については、「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』を参照してください。

UPDATE 文は、カーソル中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)。

一度に複数のローをフェッチする

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。「PUT 文 [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』と「EXECUTE 文 [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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 *samples-dir¥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。

長い値の送信と取り出し

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 (sqldata、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。

値の取り出し方法と送信方法については、「LONG データの取り出し」 573 ページと 「LONG データの送信」 575 ページを参照してください。

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。

LONG データの取り出し

この項では、データベースから LONG 値を取り出す方法について説明します。詳細については、「長い値の送信と取り出し」 572 ページを参照してください。

手順は、静的 SQL と動的 SQL のどちらを使用するかに応じて異なります。

♦ LONG VARCHAR、LONG NVARCHAR、LONG BINARY の値を受信するには、次の手順に従います (静的 SQL の場合)。

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。

     詳細については、「インジケータ変数」 541 ページを参照してください。
   - **stored_len**　配列に格納されるバイト数。常に array_len および untrunc_len 以下になります。
   - **untrunc_len**　値がトランケートされなかった場合に配列に格納されるバイト数。常に stored_len 以上になります。トランケートが発生すると、値は array_len より大きくなります。

♦ LONGVARCHAR、LONGNVARCHAR、LONGBINARY 構造体に値を受信するには、次の手順に従います (動的 SQL の場合)。

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) マクロを使用すると、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。

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 );
}
```

LONG データの送信

この項では、Embedded SQL アプリケーションからデータベースに LONG 値を送信する方法について説明します。詳細については、「長い値の送信と取り出し」 572 ページを参照してください。

手順は、静的 SQL と動的 SQL のどちらを使用するかに応じて異なります。

♦ **LONG 値を送信するには、次の手順に従います (静的 SQL の場合)。**

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。

   詳細については、「インジケータ変数」 541 ページを参照してください。
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。

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;
}
```

♦ **LONG 値を送信するには、次の手順に従います (動的 SQL の場合)。**

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) マクロを使用すると、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。

単純なストアド・プロシージャの使用

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;

// Code here
EXEC SQL CREATE 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;
```

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

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 コマンドのすべての形式 (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 コマンドを使用してください。RESUME コマンドは警告 (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。

動的カーソルについては、「動的 SELECT 文」 552 ページを参照してください。

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。

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。

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。

Embedded SQL のプログラミング・テクニック

この項では、Embedded SQL プログラムの開発者に役立つ一連のヒントについて説明します。

要求管理の実装

インタフェース DLL のデフォルトの動作では、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他の関数を実行します。この動作は、要求管理関数を使用して変更することができます。たとえば、Interactive SQL を使用している場合、Interactive SQL がデータベースからの応答を待っている間もオペレーティング・システムは依然としてアクティブであり、Interactive SQL はそのときに何らかのタスクを実行できます。

コールバック関数を使用すると、データベース要求の処理中もアプリケーションのアクティビティを実行できます。このコールバック関数の内部では、他のデータベース要求はしないでください (db_cancel_request を除く)。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。

db_register_a_callback 関数は、アプリケーションのコールバック関数を登録するために使用します。

参照

- 「db_register_a_callback 関数」 596 ページ
- 「db_cancel_request 関数」 590 ページ
- 「db_is_working 関数」 594 ページ

バックアップ関数

db_backup 関数は Embedded SQL アプリケーションにオンライン・バックアップ機能を提供します。バックアップ・ユーティリティは、この関数を使用しています。この関数を使用するプログラムを記述する必要があるのは、SQL Anywhere のバックアップ・ユーティリティでは希望どおりのバックアップができない場合だけです。

BACKUP 文を推奨
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提供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。「BACKUP 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Database Tools の DBBackup 関数を使用して、バックアップ・ユーティリティに直接アクセスすることもできます。「DBBackup 関数」 765 ページを参照してください。

参照

- 「db_backup 関数」 585 ページ

SQL プリプロセッサ

SQL プリプロセッサは、コンパイラを実行する前に、Embedded SQL を含んだ C または C++ プログラムを処理します。

構文

sqlpp [*options*] *input-file* [*output-file*]

オプション	説明
-d	データ領域サイズを減らすコードを生成します。データ構造体を再利用し、実行時に初期化してから使用します。これはコード・サイズを増加させます。
-e *level*	指定した規格に含まれない静的 Embedded SQL をエラーとして通知します。*level* 値は、使用する規格を表します。たとえば sqlpp -e c03 ... は、コア SQL/2003 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。 ◆ **c03** コア SQL/2003 構文でない構文を通知します。 ◆ **p03** 上級 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 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
-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。
-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** Microsoft Windows ◆ **NETWARE** Novell NetWare ◆ **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する。再入可能コードの詳細については、「マルチスレッドまたは再入可能コードでの SQLCA 管理」 546 ページを参照してください。

オプション	説明
-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。詳細については、「Embedded SQL API リファレンス」『Ultra Light - C/C++ プログラミング』を参照してください。
-w *level*	指定した規格に含まれない静的 Embedded SQL を警告として通知します。*level* 値は、使用する規格を表します。たとえば sqlpp -w c03 ... は、コア SQL/2003 構文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。 ◆ **c03** コア SQL/2003 構文でない構文を通知します。 ◆ **p03** 上級 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 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*	照合順を指定する。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 と入力してください。 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。「SACHARSET 環境変数」『SQL Anywhere サーバ - データベース管理』と「SALANG 環境変数」『SQL Anywhere サーバ - データベース管理』を参照してください。
input-file	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-file	SQL プリプロセッサが作成した C 言語のソース・ファイル

説明

SQL プリプロセッサは *input-file* に記述されている SQL 文を C 言語ソースに変換し、*output-file* に出力します。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 *input-file* です。*input-file* が *.c* 拡張子を持つ場合、デフォルトの出力ファイル拡張子は *.cc* になります。

参照

- 「Embedded SQL の概要」 520 ページ
- 「sql_flagger_error_level オプション [互換性]」『SQL Anywhere サーバ - データベース管理』
- 「sql_flagger_warning_level オプション [互換性]」『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・エラー・メッセージ」『SQL Anywhere 10 - エラー・メッセージ』

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 コマンドで含めます。

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

プロトタイプ

struct sqlda * **alloc_sqlda(** unsigned *numvar* **);**

説明

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。

alloc_sqlda_noind 関数

プロトタイプ

struct sqlda * **alloc_sqlda_noind(** unsigned *numvar* **);**

説明

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。

db_backup 関数

プロトタイプ

void **db_backup(**
SQLCA * *sqlca*,
int *op*,
int file_num,

unsigned long *page_num*,
struct sqlda * *sqlda*);

権限

DBA 権限、REMOTE DBA 権限 (SQL Remote の場合)、または BACKUP 権限を持つユーザとして接続してください。

説明

BACKUP 文を推奨
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提供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。「BACKUP 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

バッファを保存するのはアプリケーションです
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。値がゼロ

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わしたクライアント・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。*N* の値を渡すと、サーバはクライアントが最大 *N* 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、*N* 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、*N* 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・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。バックアップするすべてのファイルを DB_BACKUP_OPEN_FILE オペレーションを使用して開いてから、このオペレーションを実行してください。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、*N* 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。このデータ型の詳細については、「Embedded SQL のデータ型」 532 ページの DT_LONGBINARY を参照してください。

  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。retry 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 *ではありません*また、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
```

db_cancel_request 関数

プロトタイプ

int **db_cancel_request(** SQLCA * *sqlca* **);**

説明

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。関数が 1 を返した場合は、キャンセル要求は送信されています。0 を返した場合は、送信されていません。

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。

db_change_char_charset 関数

プロトタイプ

unsigned int **db_change_char_charset(**
SQLCA * *sqlca*,
char * charset **);**

説明

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。FIXCHAR、VARCHAR、LONGVARCHAR、STRING 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 CHAR です。

変更処理が正常終了すると 1 を返し、それ以外は 0 を返します。

推奨される文字セットのリストについては、「推奨文字セットと照合」 『SQL Anywhere サーバ - データベース管理』を参照してください。

db_change_nchar_charset 関数

プロトタイプ

unsigned int **db_change_nchar_charset(**
SQLCA * *sqlca*,
char * charset **);**

説明

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。NFIXCHAR、NVARCHAR、LONGNVARCHAR、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、ユニコード・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。

変更処理が正常終了すると 1 を返し、それ以外は 0 を返します。

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。

推奨される文字セットのリストについては、「推奨文字セットと照合」 『SQL Anywhere サーバ - データベース管理』を参照してください。

db_delete_file 関数

プロトタイプ

void **db_delete_file(**
SQLCA * *sqlca*,
char * *filename* **);**

権限

DBA 権限または REMOTE DBA 権限 (SQL Remote の場合) のあるユーザ ID で接続します。

説明

db_delete_file 関数は、データベース・サーバに *filename* を削除するように要求します。この関数は、トランザクション・ログをバックアップして名前を変更した後で、古い方のトランザクション・ログを削除するために使用することができます。「db_backup 関数」 585 ページの DB_BACKUP_READ_RENAME_LOG に関する説明を参照してください。

DBA 権限のあるユーザ ID で接続します。

db_find_engine 関数

プロトタイプ

unsigned short **db_find_engine(**
SQLCA * *sqlca*,
char * *name* **);**

説明

name という名前のデータベース・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見つからない場合、戻り値は 0 です。0 以外の値は、サーバが現在稼働中であることを示します。

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。*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。

db_fini 関数

プロトタイプ

int **db_fini(** SQLCA * *sqlca* **);**

説明

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 コマンド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。

警告
NetWare 上では各 db_init に対応する db_fini を呼び出さないと、データベース・サーバと NetWare ファイル・サーバが失敗することがあります。

Ultra Light アプリケーションで db_init を使用する方法については、「db_fini 関数」 『Ultra Light - C/C++ プログラミング』を参照してください。

db_get_property 関数

プロトタイプ

unsigned int **db_get_property(**
SQLCA * *sqlca*,
a_db_property *property*,
char * *value_buffer*,
int *value_buffer_size* **);**

説明

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。

引数は次のとおりです。

- **a_db_property**　要求されるプロパティ。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 文字を含む、文字列 value_buffer の最大長。

次のプロパティがサポートされます。

- **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 と SPX の各プロトコルは、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 ("10.0.1.3309" など)。

正常終了すると 1 を返し、それ以外は 0 を返します。

db_init 関数

プロトタイプ

int **db_init(** SQLCA * *sqlca* **);**

説明

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に、そして ESQL コマンド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。

正常終了すると 1 を返し、それ以外は 0 を返します。

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、ESQL コマンド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。

詳細については、「マルチスレッドまたは再入可能コードでの SQLCA 管理」 546 ページを参照してください。

警告
NetWare 上では各 db_init に対応する db_fini を呼び出さないと、データベース・サーバと NetWare ファイル・サーバが失敗することがあります。

Ultra Light アプリケーションで db_init を使用する方法については、「db_init 関数」 『Ultra Light - C/C++ プログラミング』を参照してください。

db_is_working 関数

プロトタイプ

unsigned short **db_is_working(** SQLCA * *sqlca* **);**

説明

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。

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。

db_locate_servers 関数

プロトタイプ

unsigned int**db_locate_servers(**
SQLCA * *sqlca*,
SQL_CALLBACK_PARM *callback_address*,
void * *callback_user_data* **);**

説明

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 は次の内容を *sqlca.h* で定義します。

```
typedef struct a_server_address {
    a_sql_uint32    port_type;
    a_sql_uint32    port_num;
    char            *name;
    char            *address;
} a_server_address;
```

- **port_type**　この時点では常に PORT_TYPE_TCP です (*sqlca.h* 内では 6 に定義されています)。
- **port_num**　このサーバが受信している TCP ポート番号です。
- **name**　サーバ名があるバッファを指します。
- **address**　サーバの IP アドレスがあるバッファを指します。

正常終了すると 1 を返し、それ以外は 0 を返します。

参照

- 「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

プロトタイプ

unsigned int **db_locate_servers_ex(**
SQLCA * *sqlca*,
SQL_CALLBACK_PARM *callback_address*,
void * *callback_user_data*,
unsigned int *bitmask*);

説明

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 は次の内容を *sqlca.h* で定義します。

```
typedef struct a_server_address {
    a_sql_uint32   port_type;
    a_sql_uint32   port_num;
    char           *name;
    char           *address;
    char           *dbname;
} a_server_address;
```

- **port_type**　この時点では常に PORT_TYPE_TCP です (*sqlca.h* 内では 6 に定義されています)。
- **port_num**　このサーバが受信している TCP ポート番号です。
- **name**　サーバ名があるバッファを指します。
- **address**　サーバの IP アドレスがあるバッファを指します。
- **dbname**　データベース名があるバッファを指します。

3 つのビットマスク・フラグがサポートされています。

- DB_LOOKUP_FLAG_NUMERIC
- DB_LOOKUP_FLAG_ADDRESS_INCLUDES_PORT
- DB_LOOKUP_FLAG_DATABASES

これらのフラグは *sqlca.h* で定義されており、OR を使用して併用できます。

DB_LOOKUP_FLAG_NUMERIC は、コールバック関数に渡されたアドレスがホスト名ではなく IP アドレスであることを確認します。

DB_LOOKUP_FLAG_ADDRESS_INCLUDES_PORT では、コールバック関数に渡された a_server_address 構造体内の TCP/IP ポート番号がアドレスに含まれていることを示します。

DB_LOOKUP_FLAG_DATABASES は、検出されたデータベースごと、または検出されたデータベースごと (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。

正常終了すると 1 を返し、それ以外は 0 を返します。

詳細については、「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』を参照してください。

db_register_a_callback 関数

プロトタイプ

```
void db_register_a_callback(
SQLCA * sqlca,
a_db_callback_index index,
( SQL_CALLBACK_PARM ) callback );
```

説明

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。「MESSAGE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。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。

  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する場合に使用できます。使用可能なメッセージ・タイプは、MESSAGE_TYPE_INFO、MESSAGE_TYPE_WARNING、MESSAGE_TYPE_ACTION、MESSAGE_TYPE_STATUS です。これらの定数は、*sqldef.h* で定義されています。*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*

  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 メッセージはダイアログ・ボックスに表示されます。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 on 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。

db_start_database 関数

プロトタイプ

unsigned int **db_start_database(** SQLCA * *sqlca*, char * *parms* **);**

引数

sqlca　SQLCA 構造体へのポインタ。詳細については、「SQLCA (SQL Communication Area)」 544 ページを参照してください。

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"
```

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。

説明

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。データベースを起動するために実行されるステップについては、「データベース・サーバの検出」『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パーミッションは、サーバ・コマンド・ラインで設定します。詳細については、「-gd 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。

db_start_engine 関数

プロトタイプ

unsigned int **db_start_engine(** SQLCA * *sqlca*, char * *parms* **);**

引数

sqlca SQLCA 構造体へのポインタ。詳細については、「SQLCA (SQL Communication Area)」 544 ページを参照してください。

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"
```

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。

説明

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。

この関数によって実行されるステップの説明については、「データベース・サーバの検出」『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・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。

次に示す db_start_engine の呼び出しは、データベース・サーバを起動して demo という名前を付けます。ただし、DBF 接続パラメータの指定にかかわらずデータベースはロードしません。

```
db_start_engine( &sqlca,
   "DBF=samples-dir¥¥demo.db;START=dbeng10" );
```

サーバとともにデータベースも起動したい場合は、次に示すように StartLine (START) 接続パラメータにデータベース・ファイルを含めてください。


```
db_start_engine( &sqlca,
  "ENG=eng_name;START=dbeng10 samples-dir¥¥demo.db" );
```

この呼び出しは、サーバを起動して eng_name という名前を付け、そのサーバで SQL Anywhere サンプル・データベースを起動します。

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 (FORCE) 接続パラメータは、db_start_engine 関数のみが使用します。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。

   ```
   start dbeng10 -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=dbeng10 -n server_2 mydb.db;ForceStart=YES" )
   ```

ForceStart (FORCE) を使用せず、EngineName (ENG) パラメータも指定されていない場合、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。

db_stop_database 関数

プロトタイプ

unsigned int **db_stop_database(** SQLCA * *sqlca*, char * *parms* **);**

引数

sqlca　SQLCA 構造体へのポインタ。詳細については、「SQLCA (SQL Communication Area)」 544 ページを参照してください。

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"
```

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。

説明

EngineName (ENG) 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。EngineName 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 が yes の場合は、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パーミッションは、サーバ・コマンド・ラインで設定します。詳細については、「-gd 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。

db_stop_engine 関数

プロトタイプ

unsigned int **db_stop_engine(** SQLCA * *sqlca*, char * *parms* **);**

引数

sqlca SQLCA 構造体へのポインタ。詳細については、「SQLCA (SQL Communication Area)」 544 ページを参照してください。

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"
```

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。

説明

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。

- EngineName (ENG) 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。EngineName 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 が yes の場合は、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パーミッションが適用されます。「-gk 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。

db_string_connect 関数

プロトタイプ

unsigned int **db_string_connect(** SQLCA * *sqlca*, char * *parms* **);**

引数

sqlca　SQLCA 構造体へのポインタ。詳細については、「SQLCA (SQL Communication Area)」 544 ページを参照してください。

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"
```

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。

説明

Embedded SQL CONNECT コマンドに対する拡張機能を提供します。

この関数によって使用されるアルゴリズムについては、「接続のトラブルシューティング」 『SQL Anywhere サーバ - データベース管理』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。

db_string_disconnect 関数

プロトタイプ

unsigned int **db_string_disconnect(**
SQLCA * *sqlca*,
char * *parms* **);**

引数

sqlca　SQLCA 構造体へのポインタ。詳細については、「SQLCA (SQL Communication Area)」 544 ページを参照してください。

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"
```

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。

説明

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 コマンドと同等の機能です。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 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。

db_string_ping_server 関数

プロトタイプ

unsigned int **db_string_ping_server(**
SQLCA * *sqlca*,
char * *connect_string*,
unsigned int *connect_to_db* **);**

説明

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りますし、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。

db_time_change 関数

プロトタイプ

unsigned int **db_time_change(**
SQLCA * *sqlca*);

説明

sqlca SQLCA 構造体へのポインタ。詳細については、「SQLCA (SQL Communication Area)」 544 ページを参照してください。
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。

正常終了すると TRUE を返し、それ以外は FALSE を返します。

fill_s_sqlda 関数

プロトタイプ

struct sqlda * **fill_s_sqlda(**
struct sqlda * *sqlda*,
unsigned int *maxlen* **);**

説明

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。成功した場合は *sqlda* が返され、十分なメモリがない場合は NULL ポインタが返されます。

SQLDA は、free_filled_sqlda 関数を使用して解放する必要があります。

fill_sqlda 関数

プロトタイプ

struct sqlda * **fill_sqlda(** struct sqlda * *sqlda* **);**

説明

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。成功した場合は *sqlda* が返され、十分なメモリがない場合は NULL ポインタが返されます。

SQLDA は、free_filled_sqlda 関数を使用して解放する必要があります。

free_filled_sqlda 関数

プロトタイプ

void **free_filled_sqlda(** struct sqlda * *sqlda* **);**

説明

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 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。

free_sqlda 関数

プロトタイプ

void **free_sqlda(** struct sqlda * *sqlda* **);**

説明

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。

free_sqlda_noind 関数

プロトタイプ

void **free_sqlda_noind(** struct sqlda * *sqlda* **);**

説明

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。

sql_needs_quotes 関数

プロトタイプ

unsigned int **sql_needs_quotes(** SQLCA **sqlca*, char * *str* **);**

説明

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 値を返します。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 SQLE_WARNING 以外の場合は、このテストでは確定できません。

sqlda_storage 関数

プロトタイプ

unsigned int **sqlda_storage(** struct sqlda * *sqlda*, int *varno* **);**

説明

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。

sqlda_string_length 関数

プロトタイプ

unsigned int **sqlda_string_length(** struct sqlda * *sqlda*, int *varno* **);**

説明

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。

sqlerror_message 関数

プロトタイプ

char * **sqlerror_message(** SQLCA * *sqlca*, char * *buffer*, int *max* **);**

説明

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。

ESQL コマンドのまとめ

EXEC SQL
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。

ESQL コマンドは大きく 2 つに分類できます。標準の SQL コマンドは、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、ESQL 特有のコマンドになります。

標準 SQL コマンドの詳細については、「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 コマンドは ESQL 特有であり、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 ESQL プログラミング専用です。

- **ALLOCATE DESCRIPTOR** 記述子にメモリを割り付ける。「ALLOCATE DESCRIPTOR 文 [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **CLOSE** カーソルを閉じる。「CLOSE 文 [ESQL] [SP]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **CONNECT** データベースに接続する。「CONNECT 文 [ESQL] [Interactive SQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **DEALLOCATE DESCRIPTOR** 記述子のメモリを再使用する。「DEALLOCATE DESCRIPTOR 文 [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **宣言セクション** 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。「宣言セクション [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **DECLARE CURSOR** カーソルを宣言する。「DECLARE CURSOR 文 [ESQL] [SP]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **DELETE (位置付け)** カーソルの現在位置のローを削除する。「DELETE (位置付け) 文 [ESQL] [SP]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **DESCRIBE** 特定の SQL 文用のホスト変数を記述する。「DESCRIBE 文 [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **DISCONNECT** データベース・サーバとの接続を切断する。「DISCONNECT 文 [ESQL] [Interactive SQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **DROP STATEMENT** 準備文が使用したリソースを解放する。「DROP STATEMENT 文 [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照。

- **EXECUTE** 特定の SQL 文を実行する。「EXECUTE 文 [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **EXPLAIN** 特定のカーソルの最適化方式を説明する。「EXPLAIN 文 [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **FETCH** カーソルからローをフェッチする。「FETCH 文 [ESQL] [SP]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **GET DATA** カーソルから長い値をフェッチする。「GET DATA 文 [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **GET DESCRIPTOR** SQLDA 内の変数に関する情報を取り出す。「GET DESCRIPTOR 文 [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **GET OPTION** 特定のデータベース・オプションの設定を取得する。「GET OPTION 文 [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **INCLUDE** SQL 前処理用のファイルをインクルードする。「INCLUDE 文 [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **OPEN** カーソルを開く。「OPEN 文 [ESQL] [SP]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **PREPARE** 特定の SQL 文を準備する。「PREPARE 文 [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **PUT** カーソルにローを挿入する。「PUT 文 [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **SET CONNECTION** アクティブな接続を変更する。「SET CONNECTION statement [Interactive SQL] [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **SET DESCRIPTOR** SQLDA 内で変数を記述し、SQLDA にデータを置く。「SET DESCRIPTOR 文 [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **SET SQLCA** デフォルトのグローバル SQLCA 以外の SQLCA を使用する。「SET SQLCA 文 [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **UPDATE (位置付け)** カーソルの現在位置のローを更新する。「UPDATE (位置付け) 文 [ESQL] [SP]」『SQL Anywhere サーバ - SQL リファレンス』を参照。
- **WHENEVER** SQL 文でエラーが発生した場合の動作を指定する。「WHENEVER 文 [ESQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照。

第 14 章

SQL Anywhere Perl DBD::SQLAnywhere API

目次

DBD::SQLAnywhere の概要

DBD::SQLAnywhere インタフェースを使用すると、Perl で作成されたスクリプトから SQL Anywhere データベースにアクセスできるようになります。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

♦ **コンピュータを準備するには、次の手順に従います。**

1. 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。
2. Microsoft Visual C++ または Microsoft Visual Studio .NET をインストールし、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。たとえば、Visual Studio .NET 2003 インストール環境の *Vc7¥bin* サブディレクトリには *vcvars32.bat* というバッチ・ファイルが格納されています。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。

♦ **Windows で DBI Perl モジュールをインストールするには、次の手順に従います。**

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* サブディレクトリに移動します。

   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. 次のコマンドを実行して Perl Module Manager を起動します。

   ```
   ppm
   ```

   ppm を起動できない場合は、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。
3. ppm プロンプトで次のコマンドを入力します。

   ```
   query dbi
   ```

   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
   ```

   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。

   ```
   install dbi
   ```
4. 次のコマンドを実行して Perl Module Manager を終了します。

   ```
   exit
   ```


♦ **Windows で DBD::SQLAnywhere をインストールするには、次の手順に従います。**

1. システムのコマンド・プロンプトで、SQL Anywhere のインストール環境の *src¥perl* サブディレクトリに移動します。
2. 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 **make 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。
3. 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 *src¥perl* ディレクトリにコピーして、テストを実行します。

   ```
   copy "samples-dir¥demo.db" .
   ```

   samples-dir のデフォルト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。

   ```
   dbeng10 demo
   nmake test
   ```

   テストが行われない場合は、SQL Anywhere のインストール環境の *win32* サブディレクトリがパスに含まれていることを確認してください。
4. 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
   ```

これで、DBD::SQLAnywhere インタフェースが使用可能になりました。

UNIX での DBD::SQLAnywhere のインストール

次の手順は、Mac OS X を含む、サポートされている UNIX プラットフォームで DBD::SQLAnywhere インタフェースをインストールする方法を示します。

♦ **コンピュータを準備するには、次の手順に従います。**

1. ActivePerl 5.6.0 ビルド 616 以降をインストールします。
2. C コンパイラをインストールします。

♦ **UNIX で DBI Perl モジュールをインストールするには、次の手順に従います。**

1. DBI モジュール・ソースを 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. 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。

♦ **UNIX で DBD::SQLAnywhere をインストールするには、次の手順に従います。**

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。

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

2. シェル・プロンプトで、SQL Anywhere のインストール環境の *src/perl* サブディレクトリに移動します。

3. システム・コマンド・プロンプトで、次のコマンドを入力して 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。

4. 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 DBD::SQLAnywhere ディレクトリにコピーして、テストを実行します。

   ```
   cp /opt/sqlanywhere10/demo.db .
   dbeng10 demo
   make test
   ```

   テストが行われない場合は、SQL Anywhere のインストール環境の *bin* サブディレクトリがパスに含まれていることを確認してください。

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
   ```

これで、DBD::SQLAnywhere インタフェースが使用可能になりました。

DBD::SQLAnywhere を使用する Perl スクリプトの作成

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について説明します。DBD::SQLAnywhere は、DBI モジュールのドライバです。DBI モジュールの詳細については、オンラインで dbi.perl.org を参照してください。

DBI モジュールのロード

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 に通知してください。Perl への通知には、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。

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 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 $data_src = "DBI:SQLAnywhere:ENG=$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 "Can't connect to $data_source: $DBI::errstr¥n";
```

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。おそらく最も簡単な操作方法は、一部のローを取得して印刷する方法です。

ローのセットを返す SQL 文は、実行する前に準備してください。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、結果セットと結果セットのローに関するメタ情報を取得します。

```
#!/usr/local/bin/perl -w
#
use DBI;
use strict;
my $database = "demo";
my $data_src = "DBI:SQLAnywhere:ENG=$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 "Can't 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。「max_cursor_count オプション [データベース]」『SQL Anywhere サーバ - データベース管理』と「max_statement_count オプション [データベース]」『SQL Anywhere サーバ - データベース管理』を参照してください。

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。最も簡単な方法は、パラメータ化された SELECT 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 $data_src = "DBI:SQLAnywhere:ENG=$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__
```

第 15 章

SQL Anywhere PHP API

目次

SQL Anywhere PHP モジュールの概要

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モジュールが 2 つ用意されています。これらのモジュールと 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。

PHP の機能は、SQL Anywhere の 32 ビット Windows バージョンでは DLL によって、x86 用 Linux では共有オブジェクトによって提供されます。SQLAnywhere PHP モジュールのソース・コードは、SQL Anywhere インストール環境の *src¥php* サブディレクトリにインストールされています。ビルド済みバージョンは、SQL Anywhere インストール環境の使用オペレーティング・システムに対応するバイナリ・サブディレクトリにインストールされます。

稼働条件

SQL Anywhere PHP モジュールを使用する前に、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 バイナリ (http://www.php.net からダウンロード)
- 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 クライアント・ソフトウェア (Windows では *dblib10.dll*、Linux/UNIX では *libdblib10.so*、Mac OS X では *libdblib10.dylib*)

SQL Anywhere PHP のインストールと設定

次の各項では、SQL Anywhere PHP モジュールのインストール方法と設定方法について説明します。

使用する PHP モジュールの選択

Windows の場合、SQL Anywhere には PHP バージョン 4 のモジュールと PHP バージョン 5 のモジュールが含まれます。

Linux の場合、SQL Anywhere には PHP バージョン 4 と PHP バージョン 5 のスレッド型モジュールと非スレッド型モジュールの両方が含まれます。PHP の CGI バージョン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、非スレッド型モジュールを使用します。Apache 2.x を使用する場合は、スレッド型モジュールを使用します。

ファイル名は次のとおりです。

ファイル名	説明
php-4.3.8_sqlanywhere10.dll	Windows 用の PHP バージョン 4.3.8 ライブラリ
php-4.3.11_sqlanywhere10.dll	Windows 用の PHP バージョン 4.3.11 ライブラリ
php-4.4.0_sqlanywhere10.dll	Windows 用の PHP バージョン 4.4.0 ライブラリ
php-5.0.2_sqlanywhere10.dll	Windows 用の PHP バージョン 5.0.2 ライブラリ
php-5.0.4_sqlanywhere10.dll	Windows 用の PHP バージョン 5.0.4 ライブラリ
php-5.1.1_sqlanywhere10.dll	Windows 用の PHP バージョン 5.1.1 ライブラリ
php-5.1.2_sqlanywhere10.dll	Windows 用の PHP バージョン 5.1.2 ライブラリ
php-4.3.8_sqlanywhere10.so	Linux 用の非スレッド型 PHP バージョン 4.3.8 ライブラリ
php-4.3.11_sqlanywhere10.so	Linux 用の非スレッド型 PHP バージョン 4.3.11 ライブラリ
php-4.4.0_sqlanywhere10.so	Linux 用の非スレッド型 PHP バージョン 4.4.0 ライブラリ
php-5.0.2_sqlanywhere10.so	Linux 用の非スレッド型 PHP バージョン 5.0.2 ライブラリ
php-5.0.4_sqlanywhere10.so	Linux 用の非スレッド型 PHP バージョン 5.0.4 ライブラリ
php-5.1.1_sqlanywhere10.so	Linux 用の非スレッド型 PHP バージョン 5.1.1 ライブラリ

ファイル名	説明
php-5.1.2_sqlanywhere10.so	Linux 用の非スレッド型 PHP バージョン 5.1.2 ライブラリ
php-4.3.8_sqlanywhere10_r.so	Linux 用のスレッド型 PHP バージョン 4.3.8 ライブラリ
php-4.3.11_sqlanywhere10_r.so	Linux 用のスレッド型 PHP バージョン 4.3.11 ライブラリ
php-4.4.0_sqlanywhere10_r.so	Linux 用のスレッド型 PHP バージョン 4.4.0 ライブラリ
php-5.0.2_sqlanywhere10_r.so	Linux 用のスレッド型 PHP バージョン 5.0.2 ライブラリ
php-5.0.4_sqlanywhere10_r.so	Linux 用のスレッド型 PHP バージョン 5.0.4 ライブラリ
php-5.1.1_sqlanywhere10_r.so	Linux 用のスレッド型 PHP バージョン 5.1.1 ライブラリ
php-5.1.2_sqlanywhere10_r.so	Linux 用のスレッド型 PHP バージョン 5.1.2 ライブラリ

Windows での PHP モジュールのインストール

Windows で SQL Anywhere PHP モジュールを使用するには、SQL Anywhere のインストール・ディレクトリから DLL をコピーして PHP インストールに追加する必要があります。オプションとして、モジュール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、各スクリプトでモジュールを手動でロードする必要がなくなります。

♦ **Windows で PHP モジュールをインストールするには、次の手順に従います。**

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。
2. *phpX_sqlanywhere10.dll* ファイルを *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。*X* は SQL Anywhere インストール環境の *win32* サブディレクトリからの PHP バージョン番号です。

注意
お使いの PHP のバージョンが SQL Anywhere で提供される SQL Anywhere PHP モジュールよりも新しい場合は、SQL Anywhere で提供される最新のモジュールを使用してください。たとえば PHP 5.1.6 がインストールされていて、最新の SQL Anywhere PHP モジュールが *php-5.1.2_sqlanywhere10.dll* であれば、*php-5.1.2_sqlanywhere10.dll* を使用します。

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。

   ```
   extension=php-X_sqlanywhere10.dll
   ```

   X は、前の手順でコピーした SQL Anywhere PHP モジュールのバージョン番号を表します。

   PHP ドライバを自動的にロードする代わりに、それを必要とする各スクリプトで手動でロードすることもできます。「SQL Anywhere PHP モジュールの設定」 625 ページを参照してください。

4. コマンド・プロンプトで、次のコマンドを入力して SQL Anywhere サンプル・データベースを起動します。

   ```
   dbeng10 demo.db
   ```

   SQL Anywhere サンプル・データベースのロケーションをコマンド・ラインで指定なければならない場合もあります。次に例を示します。

   ```
   dbeng10 "C:¥Documents and Settings¥All Users¥Documents¥SQL Anywhere 10¥Samples
   ¥demo.db"
   ```

   コマンドによって、サーバ名 **demo** のデータベース・サーバが起動されます。Windows の [スタート] メニューからサーバを起動する場合、サーバ名は **demo10** であり、SQL Anywhere マニュアルの PHP のサンプルと一致しません。

5. コマンド・プロンプトで、SQL Anywhere のインストール環境の *src¥php¥examples* サブディレクトリに移動し、次のコマンドを入力します。

   ```
   php connect.php
   ```

   メッセージ「Connected successfully」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。

6. ここまで終了したら、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。

詳細については、「PHP テスト・ページの作成」 627 ページを参照してください。

Linux での PHP モジュールのインストール

Linux で SQL Anywhere PHP モジュールを使用するには、SQL Anywhere のインストール・ディレクトリから共有オブジェクトをコピーして PHP インストールに追加する必要があります。オプションとして、モジュールをロードするためのエントリを PHP 初期化ファイル *php.ini* に追加すると、各スクリプトでモジュールを手動でロードする必要がなくなります。

♦ Linux で PHP モジュールをインストールするには、次の手順に従います。

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。
2. 共有オブジェクトを、SQL Anywhere インストール環境の *lib* サブディレクトリから *php.ini* ファイルの extension_dir エントリによって指定されるディレクトリにコピーします。

注意
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。たとえば PHP 5.1.6 がインストールされていて、最新の SQL Anywhere 共有オブジェクトが *php-5.1.2_sqlanywhere10_r.so* であれば、*php-5.1.2_sqlanywhere10_r.so* を使用します。

使用する共有オブジェクトのバージョンについては、「使用する PHP モジュールの選択」 621 ページを参照してください。

3. オプションとして、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* ファイルに追加します。別の方法として、このドライバは、それを必要とする各スクリプトの最初に数行のコードを追加することによって、手動でロードすることもできます。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=phpX_sqlanywhere10.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=phpX_sqlanywhere10_r.so
   ```

   X 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 モジュールを使用する前に、SQL Anywhere のために環境が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

5. コマンド・プロンプトで、次のコマンドを入力して SQL Anywhere サンプル・データベースを起動します。

   ```
   dbeng10 demo.db
   ```

   SQL Anywhere サンプル・データベースのロケーションをコマンド・ラインで指定なければならない場合もあります。

6. コマンド・プロンプトで、SQL Anywhere のインストール環境の *src/php/examples* サブディレクトリに移動します。次のコマンドを入力します。

   ```
   php connect.php
   ```

   メッセージ「Connected successfully」が表示されます。php コマンドが認識されない場合は、それがパスにあるかを確認します。

7. ここまで終了したら、データベース・サーバが実行されているコンソール・ウィンドウで [Q] を押して、データベース・サーバを停止します。

詳細については、「PHP テスト・ページの作成」 627 ページを参照してください。

UNIX と Mac OS X での PHP モジュールのビルド

その他のバージョンの UNIX または Mac OS X で SQL Anywhere PHP モジュールを使用するには、SQL Anywhere のインストール環境の *src¥php* サブディレクトリにインストールされたソース・コードから PHP モジュールをビルドする必要があります。詳細については、「Apache と PHP を使用した Adaptive Server Anywhere データからのコンテンツ配信」を参照してください。

SQL Anywhere PHP モジュールの設定

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 モジュール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、このモジュール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=phpX_sqlanywhere10.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を使用します。

  ```
  extension=phpX_sqlanywhere10.so

  extension=phpX_sqlanywhere10_r.so
  ```

  これらのエントリで、*X* は PHP のバージョンを表します。バージョン 4 または 5 を使用できます。

  PHP の起動時に SQL Anywhere モジュール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 モジュールがロードされるようになります。

  ```
  # Ensure that the SQL Anywhere PHP module is loaded
  if( !extension_loaded('sqlanywhere') ) {
    # Find out which version of PHP is running
    $version=phpversion;
    if( strtoupper(substr(PHP_OS, 0, 3) == 'WIN') ) {
        dl( $module_name.'.dll' );
    } else {
        dl( $module_name.'.so' );
    }
  }
  ```

- **allow_persistent**　1 に設定すると永続的接続を許可し、0 に設定すると許可しません。デフォルト値は 1 です。

  ```
  sqlanywhere.allow_persistent=1
  ```

- **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 モジュール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。1 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。0 に設定すると、必要に応じて sqlanywhere_commit または sqlanywhere_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 1 です。

  sqlanywhere.auto_commit=1

- **row_counts** 1 に設定すると、操作によって影響されるローの正確な数を返します。0 に設定すると、予測値を返します。

  sqlanywhere.row_counts=0

- **verbose_errors** 1 に設定すると、冗長エラーを返します。それ以外の場合、詳細なエラー情報を取得するには、sqlanywhere_error または sqlanywhere_errorcode 関数を呼び出す必要があります。デフォルト値は 1 です。

  sqlanywhere.verbose_errors=1

詳細については、「sqlanywhere_set_option」 649 ページ を参照してください。

Web ページでの PHP テスト・スクリプトの実行

この項では、サンプル・データベースを問い合わせて PHP に関する情報を表示する PHP テスト・スクリプトの作成方法について説明します。

PHP テスト・ページの作成

次の説明は、すべての設定に該当します。

PHP が適切に設定されているかをテストするには、次の手順に従って、phpinfo を呼び出す Web ページを作成して実行します。phpinfo は、システム設定情報のページを生成する PHP 関数です。その出力によって、PHP が適切に機能しているかを確認できます。

PHP のインストールについては、http://us2.php.net/install を参照してください。

♦ PHP 情報テスト・ページを作成するには、次の手順に従います。

1. ルートの Web コンテンツ・ディレクトリに *info.php* という名前のファイルを作成します。

   使用するディレクトリがわからない場合は、Web サーバの設定ファイルを確認してください。Apache のインストール環境では、多くの場合、そのコンテンツ・ディレクトリの名前は *htdocs* です。Mac OS X では、Web コンテンツ・ディレクトリの名前は使用しているアカウントによって異なります。

   - Mac OS X システムのシステム管理者である場合は、*/Library/WebServer/Documents* を使用します。
   - Mac OS X ユーザである場合は、*/Users/your-user-name/Sites/* にファイルを置きます。

2. ファイルに次のコードを挿入します。

   ```
   <? phpinfo() ?>
   ```

   別の方法として、PHP を適切にインストールおよび設定してから、コマンド・プロンプトで次のコマンドを発行することによって、テスト Web ページを作成することもできます。

   ```
   php -I > info.html
   ```

   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. 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、次の手順に従います。

   a. ファイル *connect.php* を PHP サンプルの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。

   b. Web ブラウザから、*connect.php* ページにアクセスします。

      メッセージ「Connected successfully」が表示されます。

♦ **SQL Anywhere PHP モジュールを使用するクエリ・ページを作成するには、次の手順に従います。**

1. ルートの Web コンテンツ・ディレクトリに、次の PHP コードを含む *sa_test.php* という名前のファイルを作成します。
2. ファイルに次の PHP コードを挿入します。

```
<?
 $conn = sqlanywhere_connect( "UID=DBA;PWD=sql" );
 $result = sqlanywhere_query( $conn, "SELECT * FROM Employees" );
 sqlanywhere_result_all( $result );
 sqlanywhere_free_result( $result );
 sqlanywhere_disconnect( $conn );
?>
```

テスト Web ページへのアクセス

次の手順に従って、PHP と SQL Anywhere PHP モジュールをインストールおよび設定した後で、Web ブラウザでテスト・ページを表示します。

♦ **Web ページを表示するには、次の手順に従います。**

1. Web サーバを再起動します。

   たとえば、Apache Web サーバを起動するには、Apache のインストール環境の *bin* サブディレクトリから次のコマンドを実行します。

   ```
   apachectl start
   ```

2. 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

3. SQL Anywhere データベース・サーバを起動します。

   たとえば、前述のテスト Web ページにアクセスするには、次のコマンドを使用して SQL Anywhere サンプル・データベースを起動します。

   ```
   dbeng10 demo.db
   ```

   SQL Anywhere サンプル・データベースのロケーションをコマンド・ラインで指定なければならない場合もあります。

4. 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・ページにアクセスするには、次の URL を入力します。

テスト・ページ …	… 使用する URL
info.php	http://localhost/info.php
sa_test.php	http://localhost/sa_test.php

すべてが正しく設定されている場合、sa_test ページに Employees テーブルの内容が表示されます。

PHP スクリプトの作成

この項では、SQL Anywhere PHP モジュールを使用して 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 のインストール環境の *src/php/examples* サブディレクトリにあります。

データベースへの接続

データベースに接続するには、標準の SQL Anywhere 接続文字列を sqlanywhere_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。<? と ?> のタグは、この間のコードを Web サーバで PHP が実行して PHP 出力に置き換えることを示します。

この例のソース・コードは、SQL Anywhere インストール環境の *connect.php* という名前のファイルにあります。

```
<?
 # Ensure that the SQL Anywhere PHP module is loaded
 if( !extension_loaded('sqlanywhere') ) {
   # Find out which version of PHP is running
   list($version, $rest) = explode('.', phpversion(), 2);
   $module_name = 'php'.$version.'_sqlanywhere10';
   if( strtoupper(substr(PHP_OS, 0, 3) == 'WIN' )) {
      dl( $module_name.'.dll' );
   } else {
      dl( $module_name.'.so' );
   }
 }

 # Connect using the default user ID and password
 $conn = sqlanywhere_connect( "UID=DBA;PWD=sql" );
 if( ! $conn ) {
    die ("Connection failed");
 } else {
    echo "Connected successfully¥n";
    sqlanywhere_disconnect( $conn );
 }
?>
```

コードの最初のブロックは、PHP モジュールがロードされていることを確認します。モジュールを自動的にロードするための行を PHP 初期化ファイルに追加した場合は、コードのこのブロックは必要ありません。起動時に SQL Anywhere PHP モジュールを自動的にロードするように PHP を設定していない場合は、このコードを他のサンプル・スクリプトに追加する必要があります。

2 番目のブロックで接続を試みます。このコードを正常に実行するには、SQL Anywhere サンプル・データベースが実行されている必要があります。

データベースからのデータの取り出し

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* という名前のファイルにあります。

```
<?
 # Connect using the default user ID and password
 $conn = sqlanywhere_connect( "UID=DBA;PWD=sql" );
 if( ! $conn ) {
    die ("Connection failed");
 } else {
    # Connected successfully.
 }

 # Execute a SELECT statement
 $result = sqlanywhere_query( $conn, "SELECT * FROM Customers" );
 if( ! $result ) {
    echo "sqlanywhere_query failed!";
    return 0;
 } else {
    echo "query completed successfully¥n";
 }

 # Generate HTML from the result set
 sqlanywhere_result_all( $result );
 sqlanywhere_free_result( $result );

 # Disconnect
 sqlanywhere_disconnect( $conn );
?>
```

sqlanywhere_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。sqlanywhere_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* という名前のファイルにあります。

```
<?
 # Connect using the default user ID and password
 $conn = sqlanywhere_connect( "UID=DBA;PWD=sql" );
 if( ! $conn ) {
    die ("Connection failed");
 } else {
    # Connected successfully.
 }
```



```
    # Execute a SELECT statement
    $result = sqlanywhere_query( $conn, "SELECT * FROM Customers" );
    if( ! $result ) {
       echo "sqlanywhere_query failed!";
       return 0;
    } else {
       echo "query completed successfully¥n";
    }

    # Retrieve meta information about the results
    $num_cols = sqlanywhere_num_fields( $result );
    $num_rows = sqlanywhere_num_rows( $result );
    echo "Num of rows = $num_rows¥n";
    echo "Num of cols = $num_cols¥n";

    while( ($field = sqlanywhere_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 = sqlanywhere_fetch_row( $result ))) {
       $curr_row++;
       $curr_col = 0;
       while( $curr_col < $num_cols ) {
          echo "$row[$curr_col]¥t|";
          $curr_col++;
       }
       echo "¥n";
    }

    # Clean up.
    sqlanywhere_free_result( $result );
    sqlanywhere_disconnect( $conn );
  ?>
```

sqlanywhere_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。この他に 2 つの同様な方法が PHP インタフェースにあり、sqlanywhere_fetch_row はカラム・インデックスのみを基準に検索し、sqlanywhere_fetch_object はカラム名のみを基準に検索します。

sqlanywhere_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。sqlanywhere_fetch_row 関数と sqlanywhere_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* という名前のファイルにあります。

```
  <?
    # Connect using the default user ID and password
    $conn = sqlanywhere_connect( "UID=DBA;PWD=sql" );
    if( ! $conn ) {
       die ("Connection failed");
```



```
} else {
    # Connected successfully.
}

# Retrieve the data and output HTML
echo "<BR>¥n";
$query1 = "SELECT table_id, table_name FROM SYSTAB";
$result = sqlanywhere_query( $conn, $query1 );
if( $result ) {

  $num_rows = sqlanywhere_num_rows( $result );
  echo "Returned : $num_rows <BR>¥n";
  $I = 1;
  while( ($row = sqlanywhere_fetch_array( $result ))) {
    echo "$I:  table_id:$row[table_id]" .
         " --- table_name:$row[table_name] <br>¥n";

    $query2 = "SELECT table_id, column_name " .
    "FROM SYSTABCOL" .
    "WHERE table_id = '$row[table_id]'";
    echo " $query2 <br>¥n";
    echo "  Columns: ";
    $result2 = sqlanywhere_query( $conn, $query2 );
    if( $result2 ) {
   while(($detailed = sqlanywhere_fetch_array($result2))) {
      echo " $detailed[column_name]";

    }
    sqlanywhere_free_result( $result2 );
     } else {
    echo "******FAILED********";
     }
     echo "<br>¥n";
     $I++;
   }
}
 echo "<BR>¥n";
 sqlanywhere_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。sqlanywhere_query 関数が、ローの配列を返します。スクリプトは、sqlanywhere_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* という名前のファイルにあります。

```
<?
 echo "<HTML>¥n";
 $qname = $_POST["qname"];
 $qname = str_replace( "¥¥", "", $qname );
```



```
    echo "<form method=post action=webisql.php>¥n";
    echo "<br>Query: <input 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;
    }

    # Connect to the database
    $con_str = "UID=DBA;PWD=sql;ENG=demo;LINKS=tcpip";
    $conn = sqlanywhere_connect( $con_str );
    if( ! $conn ) {
        echo "sqlanywhere_connect failed¥n";
        echo "</html>¥n";
        return 0;
    }

    $qname = str_replace( "¥¥", "", $qname );
    $result = sqlanywhere_query( $conn, $qname );

    if( ! $result ) {
          echo "sqlanywhere_query failed!";
    } else {
          // echo "query completed successfully¥n";
          sqlanywhere_result_all( $result, "border=1" );
        sqlanywhere_free_result( $result );
    }

    sqlanywhere_disconnect( $conn );
    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。

BLOB の使用

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・ディレクトリ内のファイル *image_insert.php* と *image_retrieve.php* のサンプル・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。

```
  <?
    $conn = sqlanywhere_connect( "UID=DBA;PWD=sql" )
          or die("Can not connect to database");
```



```
    $create_table = "CREATE TABLE images (ID INTEGER PRIMARY KEY, img IMAGE)";
    sqlanywhere_query( $conn, $create_table);

    $insert = "INSERT INTO images VALUES (99, xp_read_file('ianywhere_logo.gif'))";
    sqlanywhere_query( $conn, $insert );

    $query = "SELECT img FROM images WHERE ID = 99";
    $result = sqlanywhere_query($conn, $query);

    $data = sqlanywhere_fetch_row($result);
    $img = $data[0];

    header("Content-type: image/gif");
    echo $img;

    sqlanywhere_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。

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

- 「sqlanywhere_connect」 637 ページ
- 「sqlanywhere_disconnect」 638 ページ
- 「sqlanywhere_identity」 644 ページ
- 「sqlanywhere_pconnect」 646 ページ
- 「sqlanywhere_set_option」 649 ページ

クエリ

- 「sqlanywhere_execute」 640 ページ
- 「sqlanywhere_query」 646 ページ

結果セット

- 「sqlanywhere_data_seek」 637 ページ
- 「sqlanywhere_fetch_array」 641 ページ
- 「sqlanywhere_fetch_field」 641 ページ
- 「sqlanywhere_fetch_object」 642 ページ
- 「sqlanywhere_fetch_row」 643 ページ
- 「sqlanywhere_free_result」 644 ページ
- 「sqlanywhere_num_rows」 645 ページ
- 「sqlanywhere_num_fields」 645 ページ
- 「sqlanywhere_result_all」 647 ページ

トランザクション

- 「sqlanywhere_commit」 636 ページ
- 「sqlanywhere_rollback」 648 ページ

sqlanywhere_commit

プロトタイプ

bool **sqlanywhere_commit(** resource *link_identifier* **)**

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

link_identifier sqlanywhere_connect 関数によって返されるリンク識別子

戻り値

成功した場合は true、失敗した場合は false

例

次の例では、sqlanywhere_commit を使用して特定の接続でコミットを発生させる方法を示しています。

```
$result = sqlanywhere_commit( $conn );
```

関連する関数

- 「sqlanywhere_rollback」 648 ページ
- 「sqlanywhere_set_option」 649 ページ

sqlanywhere_connect

プロトタイプ

resource **sqlanywhere_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 リンク識別子、失敗の場合はエラーまたは 0

例

次の例では、接続文字列内に指定された SQL Anywhere データベースのユーザ ID とパスワードを渡します。

```
$conn = sqlanywhere_connect( "UID=DBA;PWD=sql" );
```

関連する関数

- 「sqlanywhere_pconnect」 646 ページ
- 「sqlanywhere_disconnect」 638 ページ

sqlanywhere_data_seek

プロトタイプ

bool **sqlanywhere_data_seek(** resource *result_identifier*, resource *row_num* **)**

説明

sqlanywhere_query を使用して開かれた *result_identifier* のロー *row_num* にカーソルを配置します。

パラメータ

result_identifier sqlanywhere_query 関数によって返される結果識別子

row_num result_identifier 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

例

次の例では、結果セット内の 6 番目のレコードを検索する方法を示しています。

```
sqlanywhere_data_seek( $result, 5 );
```

関連する関数

- 「sqlanywhere_fetch_field」 641 ページ
- 「sqlanywhere_fetch_array」 641 ページ
- 「sqlanywhere_fetch_row」 643 ページ
- 「sqlanywhere_fetch_object」 642 ページ
- 「sqlanywhere_query」 646 ページ

sqlanywhere_disconnect

プロトタイプ

bool **sqlanywhere_disconnect(** resource *link_identifier* **)**

説明

sqlanywhere_connect によってすでに開かれている接続を閉じます。

パラメータ

link_identifier sqlanywhere_connect 関数によって返されるリンク識別子

戻り値

成功の場合は true、エラーの場合は false

例

次の例では、データベースとの接続を閉じます。

```
sqlanywhere_disconnect( $conn );
```

関連する関数

- 「sqlanywhere_connect」 637 ページ
- 「sqlanywhere_pconnect」 646 ページ

sqlanywhere_error

プロトタイプ

string **sqlanywhere_error(** [resource *link_identifier*] **)**

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。*link_identifier* を指定しないと、sqlanywhere_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、sqlanywhere_connect を呼び出して接続が失敗した場合、*link_identifier* のパラメータを設定せずに sqlanywhere_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、sqlanywhere_errorcode 関数を使用します。

パラメータ

link_identifier　sqlanywhere_connect または sqlanywhere_pconnect によって返されるリンク識別子

戻り値

エラーが記述された文字列

例

次の例では、存在しないテーブルからの選択を試みます。sqlanywhere_query 関数は false を返し、sqlanywhere_error 関数はエラー・メッセージを返します。

```
$result = sqlanywhere_query( $conn, "SELECT * FROM table_that_does_not_exist" );
if( ! $result ) {
    $error_msg = sqlanywhere_error( $conn );
    echo "Query failed. Reason: $error_msg";
}
```

関連する関数

- 「sqlanywhere_errorcode」 639 ページ
- 「sqlanywhere_set_option」 649 ページ

sqlanywhere_errorcode

プロトタイプ

integer **sqlanywhere_errorcode(** [resource *link_identifier*] **)**

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。*link_identifier* を指定しないと、sqlanywhere_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、sqlanywhere_connect を呼び出して接続が失敗した場合、*link_identifier* のパラメータを設定せずに sqlanywhere_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、sqlanywhere_error 関数を使用します。

パラメータ

link_identifier　sqlanywhere_connect または sqlanywhere_pconnect によって返されるリンク識別子

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。

例

次の例では、失敗に終わった SQL Anywhere PHP 呼び出しから最後のエラー・コードを取得する方法を示しています。

```
$result = sqlanywhere_query( $conn, "SELECT * from table_that_does_not_exist" );
  if( ! $result ) {
     $error_code = sqlanywhere_errorcode( $conn );
     echo "Query failed: Error code: $error_code";
  }
```

関連する関数

- 「sqlanywhere_error」 639 ページ
- 「sqlanywhere_set_option」 649 ページ

sqlanywhere_execute

プロトタイプ

bool **sqlanywhere_execute(** resource *link_identifier*, string *sql_str* **)**

説明

sqlanywhere_connect または sqlanywhere_pconnect を使用してすでに開かれている、*link_identifier* 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。クエリの実行結果によって true または false を返します。この関数は、結果セットを返さないクエリに適しています。結果セットを取得する必要がある場合は、sqlanywhere_query 関数を使用してください。

パラメータ

link_identifier　sqlanywhere_connect または sqlanywhere_pconnect によって返されるリンク識別子

sql_str　SQL クエリ

戻り値

クエリが正常に実行されると true、それ以外の場合は false とエラー・メッセージ

例

次の例では、sqlanywhere_execute 関数を使用して DDL 文を実行する方法を示しています。

```
if( sqlanywhere_execute( $conn, "CREATE TABLE my_test_table( INT id )" ) ) {
  // handle success
} else {
  // handle failure
}
```

関連する関数

- 「sqlanywhere_query」 646 ページ

sqlanywhere_fetch_array

プロトタイプ

array **sqlanywhere_fetch_array(** resource *result_identifier* **)**

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_identifier sqlanywhere_query 関数によって返される結果識別子

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

例

次の例では、結果セットのすべてのローを取得する方法を示します。各ローは配列として返されます。

```
$result = sqlanywhere_query( $conn, "SELECT GivenName, Surname FROM Employees" );
 While( ($row = sqlanywhere_fetch_array( $result )) ) {
      echo " GivenName = $row["GivenName"] ¥n" ;
      echo " Surname = $row[1] ¥n";
 }
```

関連する関数

- 「sqlanywhere_data_seek」 637 ページ
- 「sqlanywhere_fetch_field」 641 ページ
- 「sqlanywhere_fetch_row」 643 ページ
- 「sqlanywhere_fetch_object」 642 ページ

sqlanywhere_fetch_field

プロトタイプ

object **sqlanywhere_fetch_field(** resource *result_identifier* [, *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_identifier sqlanywhere_query 関数によって返される結果識別子

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 (カラム) 番号を示す
- **name**　フィールド (カラム) 名を示す
- **numeric**　フィールドが数値であるかを示す
- **length**　フィールド長を返す
- **type**　フィールド・タイプを返す

例

次の例では、sqlanywhere_fetch_field を使用して結果セットのすべてのカラム情報を取得する方法を示します。

```
$result = sqlanywhere_query($conn, "SELECT GivenName, Surname FROM Employees");
while( ($field = sqlanywhere_fetch_field( $result )) ) {
      echo " Field ID = $field->ID ¥n";
      echo " Field name = $field->name ¥n";
 }
```

関連する関数

- 「sqlanywhere_data_seek」 637 ページ
- 「sqlanywhere_fetch_array」 641 ページ
- 「sqlanywhere_fetch_row」 643 ページ
- 「sqlanywhere_fetch_object」 642 ページ

sqlanywhere_fetch_object

プロトタイプ

array **sqlanywhere_fetch_object(** resource *result_identifier* **)**

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_identifier　sqlanywhere_query 関数によって返される結果識別子

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

例

次の例では、結果セットからローをオブジェクトとして 1 つずつ取得する方法を示します。カラム名をオブジェクト・メンバとして使用して、カラム値にアクセスできます。

```
$result = sqlanywhere_query( $conn, "SELECT GivenName, Surname FROM Employees" );
 While( ($row = sqlanywhere_fetch_object( $result )) ) {
      echo "$row->GivenName ¥n";   # output the data in the first column only.
 }
```

関連する関数

- 「sqlanywhere_data_seek」 637 ページ
- 「sqlanywhere_fetch_field」 641 ページ
- 「sqlanywhere_fetch_array」 641 ページ
- 「sqlanywhere_fetch_row」 643 ページ

sqlanywhere_fetch_row

プロトタイプ

array **sqlanywhere_fetch_row(** resource *result_identifier* **)**

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_identifier sqlanywhere_query 関数によって返される結果識別子

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

例

次の例では、結果セットからローを 1 つずつ取得する方法を示します。

```
while( ($row = sqlanywhere_fetch_row( $result )) ) {
      echo "$row[0] ¥n";     # output the data in the first column only.
 }
```

関連する関数

- 「sqlanywhere_data_seek」 637 ページ
- 「sqlanywhere_fetch_field」 641 ページ
- 「sqlanywhere_fetch_array」 641 ページ
- 「sqlanywhere_fetch_object」 642 ページ

sqlanywhere_free_result

プロトタイプ

bool **sqlanywhere_free_result(** resource *result_identifier* **)**

説明

sqlanywhere_query から返される結果識別子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_identifier sqlanywhere_query 関数によって返される結果識別子

戻り値

成功の場合は true、エラーの場合は false

例

次の例では、結果識別子のリソースを解放する方法を示します。

```
sqlanywhere_free_result( $result );
```

関連する関数

- 「sqlanywhere_query」 646 ページ

sqlanywhere_identity

プロトタイプ

integer **sqlanywhere_identity(** resource *link_identifier* **)**

integer **sqlanywhere_insert_id(** resource *link_identifier* **)**

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、ゼロを返します。

sqlanywhere_insert_id 関数は、MySQL データベースとの互換性を提供します。

パラメータ

link_identifier sqlanywhere_connect または sqlanywhere_pconnect によって返されるリンク識別子

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ゼロ。*link_identifier* が有効でない場合は false。

例

次の例では、sqlanywhere_identity 関数を使用して、指定した接続によって最後にテーブルに挿入された autoincrement 値を取得する方法を示します。

```
if( sqlanywhere_execute( $conn, "INSERT INTO my_auto_increment_table VALUES ( 1 ) " ) ) {
    $insert_id = sqlanywhere_insert_id( $conn );
    echo "Last insert id = $insert_id";
}
```

関連する関数

- 「sqlanywhere_execute」 640 ページ

sqlanywhere_num_fields

プロトタイプ

integer **sqlanywhere_num_fields(** resource *result_identifier* **)**

説明

result_identifier に含まれるカラム (フィールド) の数を返します。

パラメータ

result_identifier sqlanywhere_query 関数によって返される結果識別子

戻り値

カラムの正数、*result_identifier* が有効でない場合はエラー

例

次の例では、結果セット内のカラム数を表す値を返します。

```
$num_columns = sqlanywhere_num_fields( $result );
```

sqlanywhere_num_rows

プロトタイプ

integer **sqlanywhere_num_rows(** resource *result_identifier* **)**

説明

result_identifier に含まれるローの数を返します。

パラメータ

result_identifier sqlanywhere_query 関数によって返される結果識別子

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。「sqlanywhere_set_option」 649 ページを参照してください。

例

次の例では、結果セットに返される推定ロー数を取得する方法を示します。

```
$num_rows = sqlanywhere_num_rows( $result );
  if( $num_rows < 0 ) {
     $num_rows = abs( $num_rows );    # take the absolute value as an estimate
  }
```

関連する関数

- 「sqlanywhere_query」 646 ページ

sqlanywhere_pconnect

プロトタイプ

resource **sqlanywhere_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。sqlanywhere_connect の代わりに sqlanywhere_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 永続リンク識別子、失敗の場合はエラーまたは 0

例

次の例では、結果セットのすべてのローを取得する方法を示します。各ローは配列として返されます。

```
$conn = sqlanywhere_pconnect( "UID=DBA;PWD=sql" );
```

関連する関数

- 「sqlanywhere_connect」 637 ページ
- 「sqlanywhere_disconnect」 638 ページ

sqlanywhere_query

プロトタイプ

resource **sqlanywhere_query(** resource *link_identifier*, string *sql_str* **)**

説明

sqlanywhere_connect または sqlanywhere_pconnect を使用してすでに開かれている、*link_identifier* 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。結果セットを返さないクエリの場合は、sqlanywhere_execute 関数を使用できます。

パラメータ

link_identifier　sqlanywhere_connect 関数によって返されるリンク識別子

sql_str　SQL Anywhere によってサポートされている 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

成功の場合は結果 ID を表す正の値、失敗の場合は 0 とエラー・メッセージ

例

次の例では、SQL Anywhere データベースに対してクエリ "SELECT * FROM SYSTAB" を実行します。

```
$result = sqlanywhere_query( $conn, "SELECT * FROM SYSTAB" );
```

関連する関数

- 「sqlanywhere_execute」 640 ページ
- 「sqlanywhere_free_result」 644 ページ
- 「sqlanywhere_fetch_array」 641 ページ
- 「sqlanywhere_fetch_field」 641 ページ
- 「sqlanywhere_fetch_object」 642 ページ
- 「sqlanywhere_fetch_row」 643 ページ

sqlanywhere_result_all

プロトタイプ

bool **sqlanywhere_result_all(** resource *result_identifier* **[,** *html_table_format_string* **[,** *html_table_header_format_string* **[,** *html_table_row_format_string* **[,** *html_table_cell_format_string* **]]]])**

説明

result_identifier 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_identifier　sqlanywhere_query 関数によって返される結果識別子

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_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_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

例

次の例では、sqlanywhere_result_all を使用して、結果セットのすべてのローを含む HTML テーブルを生成する方法を示します。

```
$result = sqlanywhere_query( $conn, "SELECT GivenName, Surname FROM Employees" );
  sqlanywhere_result_all( $result );
```

この例は、スタイル・シートを使用して異なるフォーマットをローに交互に使用する方法を示しています。

```
$result = sqlanywhere_query( $conn, "SELECT GivenName, Surname FROM Employees");
sqlanywhere_result_all( $result, "border=2", "bordercolor=#3F3986", "bgcolor=#3F3986 style=
¥"color=#FF9533¥"", 'class="even"><class="odd"');
```

関連する関数

- 「sqlanywhere_query」 646 ページ

sqlanywhere_rollback

プロトタイプ

bool **sqlanywhere_rollback(** resource *link_identifier* **)**

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

link_identifier　sqlanywhere_connect 関数によって返されるリンク識別子

戻り値

成功した場合は true、失敗した場合は false

例

次の例では、sqlanywhere_rollback を使用して接続をロールバックします。

```
$result = sqlanywhere_rollback( $conn );
```

関連する関数

- 「sqlanywhere_commit」 636 ページ
- 「sqlanywhere_set_option」 649 ページ

sqlanywhere_set_option

プロトタイプ

bool **sqlanywhere_set_option(** resource *link_identifier*, string *option*, string *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、sqlanywhere_num_rows 関数は影響され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、sqlanywhere_error または sqlanywhere_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

link_identifier sqlanywhere_connect 関数によって返されるリンク識別子

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

例

次の例では、auto_commit オプションの値を設定するいくつかの方法を示します。

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

$result = sqlanywhere_set_option( $conn, "auto_commit", 0 );

$result = sqlanywhere_set_option( $conn, "auto_commit", False );
```

関連する関数

- 「sqlanywhere_commit」 636 ページ
- 「sqlanywhere_error」 639 ページ
- 「sqlanywhere_errorcode」 639 ページ
- 「sqlanywhere_num_rows」 645 ページ
- 「sqlanywhere_rollback」 648 ページ

第 16 章

Sybase Open Client API

目次

Open Client アーキテクチャ

> **注意**
> この章では、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。この章は、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・プログラマからは見えません。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。

SQL Anywhere を使った Open Client アプリケーションの実行については、「Open Server としての SQL Anywhere」『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)
date	smalldatetime
time	smalldatetime
string	varchar
timestamp	datetime

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。

ほとんどの場合、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	−1e15 + 0.0001	1e15 − 0.0001
SMALLMONEY	−214 748.3648	214 748.3647	−214 748.3648	214 748.3647
DATETIME	Jan 1, 1753	Dec 31, 9999	Jan 1, 0001	Dec 31, 9999
SMALLDATETIME	Jan 1, 1900	June 6, 2079	March 1, 1600	Dec 31, 7910

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

SQL Anywhere にタイムスタンプ値が渡された場合、Open Client の TIMESTAMP データ型の SQL Anywhere 実装は、Adaptive Server Enterprise の場合と異なります。SQL Anywhere の場合、その値は SQL Anywhere の DATETIME データ型にマッピングされます。SQL Anywhere では、デフォルト値は NULL であり、ユニークであることは保証されません。一方、Adaptive Server Enterprise では、単調に増加するユニークな値であることが保証されます。

一方、SQL Anywhere の TIMESTAMP データ型には、年、月、日、時、分、秒、秒未満が入ります。さらに、SQL Anywhere の DATETIME データ型は、SQL Anywhere によってマッピングされる Open Client データ型よりも、取り得る値の範囲が大きくなっています。

Open Client アプリケーションでの SQL の使用

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。

この項の概念については、「アプリケーションでの SQL の使用」 25 ページを参照してください。詳細については、Open Client のマニュアルを参照してください。

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);
```

ct_command 関数はさまざまな目的に使用されます。

準備文の使用

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、次の手順に従います。**

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。

カーソルの使い方

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、次の手順に従います。**

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。

♦ **カーソルからローを修正するには、次の手順に従います。**

- 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 でクエリ結果を記述する

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。

- **コミット・サービス**　SQL Anywhere は Adaptive Server Enterprise のコミット・サービスをサポートしません。

- **機能**　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 や暗号化パスワードなどのセキュリティ・オプションは、サポートされません。

- 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。

第 17 章

SQL Anywhere Web サービス

目次

Web サービスの概要

SQL Anywhere には、Web サービスを提供したり、その他の SQL Anywhere データベースの Web サービスやインターネットを介して使用できる標準の Web サービスにアクセスするための、組み込みの HTTP サーバが含まれています。SOAP はこの目的に使用される標準ですが、SQL Anywhere の組み込みの HTTP サーバでは、クライアント・アプリケーションからの標準の HTTP 要求や HTTPS 要求も処理できます。

「Web サービス」という用語は、さまざまな意味で使用されています。通常は、コンピュータ間のデータの転送と相互運用性を実現するソフトウェアを指します。本質的に、Web サービスはビジネス論理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。「**Simple Object Access Protocol** (SOAP)」は XML ベースの単純なプロトコルで、SOAP を使用するとアプリケーションは HTTP を介して情報をやりとりできるようになります。

SOAP は、Java や Visual Studio .NET で記述されたアプリケーション同士がインターネットを介して通信する方法を提供します。SOAP メッセージは、サーバが提供するサービスを定義します。実際のデータ転送は、関連情報を効果的にエンコードするよう構造化された XML ドキュメントを交換できるよう、通常は HTTP を使用して行われます。SOAP 通信に参加するクライアントまたはサーバなどのアプリケーションはすべて SOAP ノードまたは SOAP 終了ポイントと呼ばれます。このようなアプリケーションは、SOAP メッセージを送信、受信、処理できます。SOAP ノードは、SQL Anywhere を使用して作成できます。

SOAP 規格の詳細については、www.w3.org/TR/soap を参照してください。

Web サービスと SQL Anywhere

SQL Anywhere のコンテキストにおいて、Web サービスという用語は、SQL Anywhere に標準の SOAP 要求を受信して処理する機能があることを意味しています。SQL Anywhere の Web サービスは、クライアント・アプリケーションに対して、JDBC や ODBC などの従来のインタフェースの代用を提供します。Web サービスへは、各種の言語で記述され、各種のプラットフォームで実行されるクライアント・アプリケーションからアクセスできます。Perl や Python などの一般的なスクリプト言語でも Web サービスにアクセスできます。Web サービスは、CREATE SERVICE 文を使用してデータベースに作成できます。

SQL Anywhere は、SOAP または HTTP クライアントとしても機能し、データベース内で実行中のアプリケーションが、インターネットで使用できる標準の Web サービスまたは SQL Anywhere データベースで提供されている Web サービスにアクセスできるようにします。このクライアント機能は、ストアド関数またはストアド・プロシージャを使ってアクセスされます。

さらに、Web サービスとは、組み込みの Web サーバを使用してクライアントからの HTTP 要求を処理するアプリケーションのことも指します。通常これらのアプリケーションは、従来のデータベースに基づく Web アプリケーションのように機能しますが、よりコンパクトで、データとアプリケーション全体がデータベース内に存在できるため、書き込みが簡単です。このようなアプリケーションでは、Web サービスは通常、HTML フォーマットのドキュメントを返します。GET、HEAD、POST メソッドをサポートしています。

データベース内の Web サービスの集合が、使用可能な URL を定義します。各サービスには Web ページのセットがあります。通常、これらのページの内容はデータベース内に記述および格納されたプロシージャによって生成され、単一文の場合もあれば、オプションでユーザ自身が文を実

行することもできます。これらの Web サービスは、HTTP 要求の受信を可能にするオプションのあるデータベース・サーバを起動すると使用可能になります。

Web サービス要求を処理する HTTP サーバがデータベースに組み込まれているため、パフォーマンスに優れています。Web サービスを使用するアプリケーションは、データベースとデータベース・サーバ以外の追加のコンポーネントが必要ないため、簡単に配備されます。

Web サービスのクイック・スタート

ここでは、新しいデータベースの作成、HTTP サーバが有効の状態での SQL Anywhere データベースの起動、一般的な Web ブラウザを使用したデータベースへのアクセス方法について説明します。

♦ **簡単な XML Web サービスを作成しアクセスするには、次の手順に従います。**

1. SQL Anywhere サンプル・データベースを *samples-dir* から *c:¥webserver¥demo.db* などの別のロケーションにコピーします。
2. 次の文を実行して、パーソナル Web サーバを起動します。-xs http(port=80) オプションは、HTTP 要求を受信するようにデータベース・サーバに指示します。ポート 80 ですでに Web サーバが実行されている場合、このデモには 8080 などの別のポート番号を使用します。

   ```
   dbeng10 -xs http(port=80) c:¥webserver¥demo.db
   ```

   HTTP 通信リンクの多くのプロパティは、-xs オプションのパラメータで制御できます。

   詳細については、「-xs 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。
3. 適切な -xs オプション・パラメータでデータベース・サーバを起動し、着信要求に応答するための Web サービスを作成してください。これらは、CREATE SERVICE 文を使用して定義します。

   Interactive SQL を起動します。DBA として SQL Anywhere サンプル・データベースに接続します。次の文を実行します。

   ```
   CREATE SERVICE XMLtables
   TYPE 'XML'
   AUTHORIZATION OFF
   USER DBA
   AS SELECT * FROM Customers
   ```

   この文は、XMLtables という Web サービスを作成します。この単純な Web サービスは SELECT * FROM Customers 文の結果を返し、出力は自動的に XML フォーマットに変換されます。認証がオフになっているので、Web ブラウザからテーブルへのアクセスには許可が不要です。

   詳細については、「Web サービスの作成」 667 ページと「CREATE SERVICE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
4. Web ブラウザを起動します。
5. URL http://localhost:80/demo/XMLtables を検索します。データベース・サーバの起動時に指定したポート番号を使用します。

   - **localhost:80** 使用する Web ホスト名とポート番号を定義します。
   - **demo** 使用するデータベース名を定義します。ここでは *c:¥webserver¥demo.db* を使用します。
   - **XMLtables** 使用するサービス名を定義します。

Web ブラウザは、データベース・サーバによって返された XML ドキュメントの本文を表示します。フォーマット情報は含まれないため、表示されるのはタグや属性を含んだ未加工 XML です。

6. また、一般的なプログラミング言語からも XMLtables ルーチンにアクセスできます。たとえば、次の短い C# プログラムでは XMLtables Web サービスを使用します。

```
using System.Xml;

static void Main(string[] args)
{
  XmlTextReader reader =
    new XmlTextReader( "http://localhost:80/demo/XMLtables" );

  while( reader.Read() )
  {

    switch( reader.NodeType )
    {
    case XmlNodeType.Element:
      if( reader.Name == "row" )
      {
        Console.Write(reader.GetAttribute("ID")+" ");
        Console.WriteLine(reader.GetAttribute("Surname"));
      }
      break;
    }
  }
}
```

7. さらに、次の例のように Python からも同じ Web サービスにアクセスできます。

```
import xml.sax

class DocHandler( xml.sax.ContentHandler ):
  def startElement( self, name, attrs ):
    if name == 'row':
      table_id = attrs.getValue( 'ID' )
      table_name = attrs.getValue( 'Surname' )
      print '%s %s' % ( table_id, table_name )

parser = xml.sax.make_parser()
parser.setContentHandler( DocHandler() )
parser.parse( 'http://localhost:80/demo/XMLtables' )
```

このコードを *DocHandler.py* というファイルに保存します。アプリケーションを実行するには、次のようなコマンドを入力します。

```
python DocHandler.py
```

♦ **簡単な HTML Web サービスを作成しアクセスするには、次の手順に従います。**

1. 次の文を実行して、パーソナル Web サーバを起動します。-xs http(port=80) オプションは、HTTP 要求を受信するようにデータベース・サーバに指示します。ポート 80 ですでに Web サーバが実行されている場合、このデモには 8080 などの別のポート番号を使用します。

```
dbeng10 -xs http(port=80) c:¥webserver¥demo.db
```

HTTP 通信リンクの多くのプロパティは、-xs オプションのパラメータで制御できます。

詳細については、「-xs 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。

2. 適切な -xs オプション・パラメータでデータベース・サーバを起動し、着信要求に応答するための Web サービスを作成してください。これらは、CREATE SERVICE 文を使用して定義します。

   Interactive SQL を起動します。DBA として SQL Anywhere サンプル・データベースに接続します。次の文を実行します。

   ```
   CREATE SERVICE HTMLtables
   TYPE 'HTML'
   AUTHORIZATION OFF
   USER DBA
   AS SELECT * FROM Customers
   ```

   この文は、HTMLtables という Web サービスを作成します。この単純な Web サービスは SELECT * FROM Customers 文の結果を返し、出力は自動的に HTML フォーマットに変換されます。認証がオフになっているので、Web ブラウザからテーブルへのアクセスには許可が不要です。

   詳細については、「Web サービスの作成」 667 ページと「CREATE SERVICE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

3. Web ブラウザを起動します。

4. URL http://localhost:80/demo/HTMLtables を検索します。データベース・サーバの起動時に指定したポート番号を使用します。

   Web ブラウザは、データベース・サーバによって返された HTML ドキュメントの本文を表示します。デフォルトでは、結果セットは HTML テーブル形式にフォーマットされます。

クイック・スタートのためのその他の資料

サンプル・プログラムは *samples-dir¥SQLAnywhere¥HTTP* ディレクトリにあります。

その他の例は、CodeXchange (http://ianywhere.codexchange.sybase.com/) から入手できる場合があります。

Web サービスの作成

データベース内に作成および格納されている Web サービスが、有効な URL と実行する内容を定義します。単一のデータベースに複数の Web サービスを定義できます。異なるデータベースにある複数の Web サービスを、単一の Web サイトの一部として表示するように定義できます。

次の文は、Web サービスの作成、変更、削除を許可します。

- CREATE SERVICE
- ALTER SERVICE
- DROP SERVICE
- COMMENT ON SERVICE

CREATE SERVICE 文の一般的な構文は、次のとおりです。

CREATE SERVICE *service-name* **TYPE '***service-type***'** [*attributes*] [**AS** *statement*]

サービス名

サービス名がサービスへのアクセスに使用する URL の一部となるので、URI を構成する文字については柔軟性があります。標準的な英数字に加えて、"-"、"_"、"."、"!"、"*"、"'"、"("、")" が使用できます。

また、DISH サービスの名前に使用されているもの以外のサービス名にはスラッシュ (/) を含めることができますが、この文字は標準の URL デリミタであり、SQL Anywhere による URL の解釈方法に影響するため、使用にはいくつかの制限があります。この文字はサービス名の先頭文字としては使えません。また、サービス名に 2 つのスラッシュを連続して使用することはできません。

サービス名の命名に使用できる文字はグループ名にも使用できます。これは、DISH サービスについてのみ該当します。

サービス・タイプ

サポートされるサービス・タイプは次のとおりです。

- **SOAP**　結果セットは、SOAP 応答として返されます。データのフォーマットは、FORMAT 句によって決定されます。SOAP サービスへの要求は、単純な HTTP 要求ではなく有効な SOAP 要求である必要があります。
- **DISH**　DISH サービス (SOAP ハンドラを決定) は、GROUP 句で識別される SOAP サービスのプロキシとして機能し、これらの各 SOAP サービスに対して WSDL (Web Services Description Language) ドキュメントを生成します。
- **XML**　結果セットは XML として返されます。結果セットがすでに XML の場合、それ以上フォーマットは適用されません。XML 以外の場合は、自動的に XML としてフォーマットされます。効果は、SELECT 文で FOR XML RAW 句を使用した場合と同様です。

- **HTML**　文またはプロシージャの結果セットは、テーブルを格納する HTML ドキュメントに自動的にフォーマットされます。
- **RAW**　結果セットがフォーマットされずにクライアントに送られます。要求されたタグをプロシージャ内で明示的に生成することによって、フォーマットされた文書を作成できます。

すべてのサービス・タイプのうち、出力を制御しやすいのは RAW です。ただし、必要なタグをすべて明示的に出力しなければならないので必要な作業が増えます。XML サービスの出力は、サービスの文に FOR XML 句を適用することにより調節できます。SOAP サービスの出力は、CREATE SERVICE 文または ALTER SERVICE 文の FORMAT 属性を適用することにより調節できます。

詳細については、「CREATE SERVICE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

文

文とはコマンドのことであり、通常はストアド・プロシージャです。文は、ユーザがサービスにアクセスしたときに呼び出されます。特定の文を定義した場合、それがこのサービスで実行可能な唯一の文となります。文は SOAP サービスでは必須で、DISH サービスでは無視されます。デフォルトは NULL で、文がないことを示します。

文を含まないサービスを作成できます。文は URL から取得されます。このように設定されたサービスは、サービスをテストしたり、情報への一般的なアクセスが必要であったりする場合に便利です。文を含まないサービスを作成するには、文を完全に省略するか、文の箇所に AS NULL というフレーズを使用します。

文を持たないサービスでは、深刻なセキュリティ上の問題が発生します。というのも、Web クライアントによる任意のコマンドの実行が可能となるからです。そのようなサービスを作成した場合、認証を有効にし、有効なユーザ名とパスワードの入力をすべてのクライアントに要求してください。その場合でも、運用システムでは、文が定義されたサービスだけが実行されるようにしてください。

属性

使用可能な属性は、次のとおりです。一般的に、すべてオプションです。ただし、相互依存型のものもあります。

- **AUTHORIZATION**　この属性は、サービスを使用できるユーザを制御します。デフォルト設定は ON です。文がない場合、認証は ON にしてください。また、認証設定は、USER 属性で定義されたユーザ名の解釈方法に影響します。
- **SECURE**　ON に設定すると、安全な接続のみが許可されます。HTTP ポートが受信するすべての接続が自動的に HTTPS ポートにリダイレクトされます。デフォルトは OFF で、データベース・サーバを起動するときにこれらのポートが適切なオプションを使用して有効になっていれば、HTTP 要求と HTTPS 要求の両方が有効です。

  詳細については、「-xs 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。
- **USER**　USER 句は、サービス要求を処理するのに使用できるデータベース・ユーザ・アカウントを制御します。ただし、この設定の解釈は、認証が ON か OFF かに依存します。

認証が ON に設定されている場合、すべてのクライアントは接続時に有効なユーザ名とパスワードを入力する必要があります。認証が ON の場合、USER オプションは NULL、データベース・ユーザ名、データベース・グループ名のいずれかです。NULL の場合、どのデータベース・ユーザも接続して要求できます。要求は、そのユーザのアカウントとパーミッションを使用して実行されます。グループ名が指定されている場合、グループに属するユーザのみが要求を実行できます。ほかのデータベース・ユーザはすべて、サービスを使用するためのパーミッションが拒否されます。

認証が OFF の場合は、文を指定する必要があります。また、ユーザ名も指定してください。そのユーザのアカウントとパーミッションを使用して、すべての要求が実行されます。したがって、サーバがパブリックなネットワークに接続されている場合、悪意の使用による損傷を制限するために、指定されたユーザ・アカウントのパーミッションを最小限にしてください。

- **GROUP** DISH サービスのみに適用する GROUP 句は、DISH サービスで公開される SOAP サービスを決定します。DISH サービスによって公開される SOAP サービスは、その DISH サービスのグループ名で始まる名前を持つもののみです。したがって、グループ名が、公開された SOAP サービスに共通するプレフィクスとなります。たとえば、GROUP *xyz* を指定すると、SOAP サービス *xyz*/*aaaa*、*xyz*/*bbbb*、または *xyz*/*cccc* のみ公開され、*abc/aaaa* または *xyzaaaa* は公開されません。グループ名が指定されていない場合、DISH サービスはデータベース内のすべての SOAP サービスを公開します。サービス名で使用できる文字と同じ文字をグループ名に使用できます。

  SOAP サービスは、複数の DISH サービスによって公開される場合があります。具体的には、この機能によって、単一の SOAP サービスが複数のフォーマットでデータを提供することが可能になります。SOAP サービスで指定がないかぎり、サービス・タイプは DISH サービスから継承されます。したがって、フォーマット・タイプを宣言しない SOAP サービスを作成してから、それぞれ異なるフォーマットを指定する複数の DISH サービスに含めることができます。

- **FORMAT** DISH サービスと SOAP サービスに適用する FORMAT 句は、SOAP または DISH の応答の出力フォーマットを制御します。.NET または Java JAX-RPC など、さまざまな種類の SOAP クライアントと互換性のある出力フォーマットが使用可能です。SOAP サービスのフォーマットが指定されていない場合、フォーマットはサービスの DISH サービス宣言から継承されます。DISH サービスでもフォーマットが宣言されていない場合、デフォルトは、.NET クライアントと互換性のある DNET です。フォーマットを宣言しない SOAP サービスは、複数の DISH サービスを定義することにより、それぞれ異なる FORMAT タイプを持つさまざまな種類の SOAP クライアントで使用できます。

- **URL [PATH]** URL 句または URL PATH 句は、URL の相互運用を制御し、XML、HTML、および RAW サービス・タイプのみ適用します。特に、URL パスを受け入れるか否か、また受け入れる場合はどのように処理するかを決定します。サービス名が文字 "/" で終了している場合は、URL を OFF に設定してください。

  詳細については、「CREATE SERVICE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Web 要求を受信するデータベース・サーバの起動

データベース・サーバに HTTP または HTTPS で Web サービス要求を受信させたい場合は、サーバの起動時に受信する Web 要求のタイプをコマンド・ラインで指定する必要があります。デフォルトでは、データベース・サーバは Web サービス要求を受信しないため、クライアントはデータベースに定義されているサービスへアクセスする方法がありません。

また、どのポートで受信するかなど、HTTP または HTTPS サービスのさまざまなプロパティを、コマンド・ラインで指定することもできます。

また、データベース内に Web サービスを作成することも必要です。詳細については、「Web サービスの作成」 667 ページを参照してください。

-xs オプションを使用してプロトコルを有効にできます。使用可能な Web サービス・プロトコルには、HTTP と HTTPS の 2 つがあります。プロトコル名の後にオプションのパラメータをカッコに入れて追加すると、各タイプの Web サービスへのアクセスをカスタマイズできます。

オプションの一般的な構文は、次のとおりです。

-xs { *protocol* [**(***option***=***value*; ⋯**)**], ⋯ }

複数の Web サーバの起動
複数の Web サーバを同時に起動する場合、どちらも同じデフォルト・ポートを使用するため、追加の Web サーバのポートを変更する必要があります。

プロトコル

次の Web サービス・プロトコル値を使用できます。

- **http** HTTP 接続を受信します。
- **https** HTTPS 接続を受信します。
- **none** Web サービス要求を受信しません。これがデフォルト設定です。

オプション

使用可能なオプションは次のとおりです。

- **FIPS** **FIPS=Y** と指定すると、HTTPS FIPS 接続を受信します。
- **ServerPort [PORT]** Web 要求を受信するポート。デフォルトで、SQL Anywhere はポート 80 で HTTP 要求を受信し、ポート 443 でセキュア HTTP (HTTPS) 要求を受信します。FIPS 承認の HTTPS 接続のデフォルト・ポートは、HTTPS の場合と同じです。

  たとえば、すでにポート 80 で動作中の Web サーバがある場合、次のオプションを使用してポート 8080 で Web 要求を受信するデータベース・サーバを起動できます。

```
dbeng10 mywebapp.db -xs http(port=8080)
```

別の例として、次のコマンドは、SQL Anywhere に含まれているサンプル証明書を使用して安全な Web サーバを起動します (ファイルは、RSA または FIPS 承認の RSA 暗号化がインストールされていると存在します)。このコマンドは、1 行に入力してください。

```
dbeng10 -xs https(certificate=rsaserver.crt;
        certificate_password=test)
```

警告
サンプル証明書は、テスト作業と開発作業にのみ使用します。この証明書は SQL Anywhere の標準部分であるため、保護機能は備えていません。アプリケーションを配備する前に独自の証明書で置換してください。

- **DatabaseName [DBN]** 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。

  このパラメータが REQUIRED に設定されている場合は、URL がデータベース名を指定します。

  このパラメータが AUTO に設定されている場合は、URL がデータベース名を指定できますが、必須ではありません。URL にデータベース名が含まれていない場合は、サーバでのデフォルトのデータベースを Web 要求の処理に使用します。

  このパラメータにデータベースが設定されている場合は、このデータベースを使用してすべての Web 要求を処理します。URL にはデータベース名を含めないでください。

- **LocalOnly [LOCAL]** このパラメータを YES に設定すると、ネットワーク・データベース・サーバは異なるコンピュータで実行中のクライアントからの通信をすべて拒否します。このオプションは、他のコンピュータからの Web サービス要求を受け入れることのないパーソナル・データベース・サーバには影響しません。デフォルト値は NO で、どの場所にあるクライアントの要求も受け入れます。

- **LogFile [LOG]** データベース・サーバが Web サービス要求に関する情報を書き込むファイル名。

- **LogFormat [LF]** ログ・ファイルに書き込まれるメッセージのフォーマットと、表示されるフィールドを制御します。文字列に表示される場合、各メッセージが書き込まれると現在の値が @T などのコードに置き換えられます。

  デフォルト値は **@T - @W - @I - @P - "@M @U @V" - @R - @L - @E** で、次のようなメッセージを生成します。

  ```
  06/15 01:30:08.114 - 0.686 - 127.0.0.1 - 80
   - "GET /web/ShowTable HTTP/1.1" - 200 OK - 55133 -
  ```

  ログ・ファイルのフォーマットは Apache との互換性があるため、その分析に同じツールを使用できます。

  フィールド・コードの詳細については、「LogFormat プロトコル・オプション [LF]」 『SQL Anywhere サーバ - データベース管理』を参照してください。

- **LogOptions [LOPT]**　ログ・ファイルに書き込まれるメッセージまたはメッセージのタイプを制御するキーワードとエラー番号を指定できます。

  詳細については、「LogOptions プロトコル・オプション [LOPT]」『SQL Anywhere サーバ - データベース管理』を参照してください。

使用可能なオプションの完全なリストと詳細については、「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』を参照してください。

URL の解釈方法の概要

URL (Universal Resource Locators) は、SOAP または HTTP Web サービスから取得できる HTML ページなどのドキュメントを指定します。SQL Anywhere で使用される URL は、Web のブラウズで見慣れた形式に従っています。データベース・サーバを介してユーザがブラウズする場合、それらの要求が従来のスタンドアロン Web サーバで処理されていないことを意識する必要はありません。

SQL Anywhere データベース・サーバは、フォーマットは標準ですが、URL の解釈方法は標準の Web サーバと異なります。データベース・サーバの起動時に指定するオプションも、その解釈方法に影響します。

URL の一般的な構文は、次のとおりです。

{ http | https }:*//*[*user*:*password*@]*host*[:*port*][*/dbn*]*/service-name*[*path* | *?searchpart*]

次に示すのも URL の例です。http://localhost:80/demo/XMLtables

ユーザとパスワード

Web サービスに認証が必要な場合、ユーザ名とパスワードは、コロンで区切って電子メール・アドレスのようにホスト名の前に挿入することにより、URL の一部として直接渡すことができます。

ホストとポート

すべての標準的な HTTP 要求と同様に、URL はホスト名や IP 番号から始まり、オプションでポート番号になります。IP アドレスやホスト名とポートは、サーバが受信しているうちの 1 つにしてください。IP アドレスは、SQL Anywhere を実行しているコンピュータのネットワーク・カードのアドレスです。ポート番号は、データベース・サーバを起動したときに -xs オプションで指定したポート番号です。ポート番号を指定しない場合、そのサービス・タイプでデフォルトのポート番号が使用されます。たとえば、デフォルトでサーバはポート 80 で HTTP 要求を受信します。

詳細については、「-xs 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。

データベース名

スラッシュの間にある次のトークンは、通常データベース名です。このデータベースは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。

URL にデータベース名が表示されず、データベース名が -xs サーバ・オプションの DBS 接続パラメータを使用して指定されていない場合は、デフォルトのデータベースが使用されます。

データベース・サーバで実行中のデータベースが 1 つだけか、またはデータベース名が -xs オプションの DBS 接続パラメータを使用して指定されている場合のみ、データベース名を省略できます。

サービス名

次の URL の部分はサービス名です。このサービスは、指定したデータベースに存在している必要があります。Web サービス名にスラッシュ文字が含まれていることもあるため、サービス名

は次のスラッシュ文字にまたがる場合もあります。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。

URL にサービス名の指定がない場合は、データベース・サーバがサービスの指定されたルートを検索します。指定したサービスまたはルート・サービスが定義されていないと、サーバは 404 Not Found エラーを返します。

パラメータ

対象となるサービスの種類によって、パラメータを指定する方法が異なります。HTML、XML、RAW サービスに対するパラメータは、次のいずれかの方法で渡すことができます。

- スラッシュを使用して URL に追加
- 明示的な URL パラメータ・リストとして指定
- POST 要求の POST データとして指定

SOAP サービスに対するパラメータは、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。

URL パス

パラメータ値にアクセスするには、パラメータに名前を指定します。これらのホスト変数名にはプレフィクスとしてコロン (:) が付き、Web サービス定義の一部を形成する文に含めることができます。

たとえば、次のストアド・プロシージャを定義したとします。

```
CREATE PROCEDURE Display (IN ident INT )
BEGIN
   SELECT ID, GivenName, Surname FROM Customers
   WHERE ID = ident;
END
```

このストアド・プロシージャを呼び出す文には、顧客 ID 番号が必要です。サービスを次のように定義します。

```
CREATE SERVICE DisplayCustomer
TYPE 'HTML'
URL PATH ELEMENTS
AUTHORIZATION OFF
USER DBA
AS CALL Display( :url1 );
```

この場合、URL は http://localhost/demo/DisplayCustomer/105 のようになります。

パラメータ 105 が url1 としてサービスに渡されます。URL PATH ELEMENTS 句は、スラッシュで区切られたパラメータがそれぞれパラメータ url1、url2、url3 などとして渡されることを示します。この方法では、パラメータを 10 個まで渡すことができます。

Display プロシージャのパラメータは 1 つなので、サービスは次のように定義することもできます。

```
CREATE SERVICE DisplayCustomer
TYPE 'HTML'
```

```
URL PATH ON
AUTHORIZATION OFF
USER DBA
AS CALL Display( :url );
```

この場合、パラメータ 105 が url としてサービスに渡されます。URL PATH ON 句は、サービス名の後に続くものすべてが url という 1 つのパラメータとして渡されることを示します。したがって、次の URL では、文字列 105/106 が url として渡されます (Display ストアド・プロシージャでは整数値が必要とされるので、これは SQL エラーになります)。

http://localhost:80/demo/DisplayCustomer/105/106

変数の詳細については、「変数の使用」 719 ページを参照してください。

パラメータには、HTTP_VARIABLE 関数を使用してもアクセスできます。詳細については、「HTTP_VARIABLE 関数 [HTTP]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

URL searchpart

パラメータを渡すもう 1 つの方法は、URL searchpart (検索部分) メカニズムを使用することです。URL searchpart は、疑問符 (?) と、それに続くアンパサンド (&) 区切りの *name=value* ペアとで構成されます。searchpart は、URL の末尾に追加されます。次に、一般的なフォーマットの例を示します。

```
http://server/path/document?name1=value1&name2=value2
```

GET 要求はこのような形でフォーマットされます。名前付き変数がある場合、対応する値が定義され、割り当てられます。

たとえば、ストアド・プロシージャ ShowSalesOrderDetail を呼び出す文には、顧客 ID 番号と製品 ID 番号がどちらも必要です。

```
CREATE SERVICE ShowSalesOrderDetail
TYPE 'HTML'
URL PATH OFF
AUTHORIZATION OFF
USER DBA
AS CALL ShowSalesOrderDetail( :customer_id, :product_id );
```

この場合、URL は http://localhost:80/demo/ShowSalesOrderDetail?customer_id=101&product_id=300 のようになります。

URL PATH を ON または ELEMENTS に設定すると、追加の変数が定義されます。ただし、この 2 つに通常は関連性はありません。URL PATH を ON または ELEMENTS に設定すると、要求された URL で変数を使用できます。次の例は、これら 2 つを組み合わせて使用する方法を示します。

```
CREATE SERVICE ShowSalesOrderDetail2
TYPE 'HTML'
URL PATH ON
AUTHORIZATION OFF
USER DBA
AS CALL ShowSalesOrderDetail( :customer_id, :url );
```


次の例では、searchpart と URL パスがどちらも使用されています。値 300 は url に代入され、101 は customer_id に代入されます。

http://localhost:80/demo/ShowSalesOrderDetail2/300?customer_id=101

また、次のようにすると searchpart だけで表現できます。

http://localhost:80/demo/ShowSalesOrderDetail2/?customer_id=101&url=300

ここで、同じ変数に対してこれら両方の方法で指定があった場合にどうなるかが問題になります。次の例では、url には 300、302 の順序で代入され、有効になるのは最後に代入された値です。

http://localhost:80/demo/ShowSalesOrderDetail2/300?customer_id=101&url=302

変数の詳細については、「変数の使用」 719 ページを参照してください。

パラメータには、HTTP_VARIABLE 関数を使用してもアクセスできます。詳細については、「HTTP_VARIABLE 関数 [HTTP]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

SOAP および DISH Web サービスの作成

SOAP および DISH Web サービスは、標準の SOAP クライアント (Microsoft .NET や Java JAX-RPC で記述された SOAP クライアントなど) がアクセスできる標準の SOAP Web サービスを作成するための手段です。

SOAP サービス

SOAP サービスは、標準の SOAP 要求を受け入れ、処理する Web サービスを SQL Anywhere で構築するためのメカニズムです。

SOAP サービスを宣言するには、そのサービスが SOAP のタイプであることを指定します。標準の SOAP 要求の本文は、SOAP エンベロープで、特定のフォーマットの XML ドキュメントを意味します。SQL Anywhere は、指定したプロシージャを使用してこれらの要求を解析し、処理します。応答は、SOAP エンベロープでもある標準の SOAP 応答の形式で自動的にフォーマットされ、クライアントに返されます。

SOAP サービスの作成に使用する文の構文は、次のとおりです。

CREATE SERVICE *service-name*
TYPE 'SOAP'
[**FORMAT** { **'DNET'** | **'CONCRETE'** | **'XML'** | **NULL** }]
[*common-attributes*]
AS *statement*

DISH サービス

DISH サービスは、SOAP サービスのグループのプロキシとして機能するほか、現在公開している SOAP サービスを記述する WSDL (Web Services Description Language) ドキュメントをクライアント向けに自動構築します。

DISH サービスを作成する場合、GROUP 句で指定された名前によって、その DISH サービスが公開する SOAP サービスが決定されます。DISH サービスの名前がプレフィクスとなっている名前を持つ SOAP サービスがすべて公開されます。たとえば、GROUP xyz を指定すると、SOAP サービス xyz/aaaa、xyz/bbbb、または xyz/cccc が公開されます。abc/aaaa や xyzaaaa という名前の SOAP サービスは公開されません。SOAP サービスは、複数の DISH サービスによって公開される場合があります。グループ名が指定されていない場合、DISH サービスはデータベース内のすべての SOAP サービスを公開します。SOAP サービス名で使用できる文字と同じ文字を DISH グループ名に使用できます。

DISH サービスの作成に使用する文の構文は、次のとおりです。

CREATE SERVICE *service-name*
TYPE 'DISH'
[**GROUP** { *group-name* | **NULL** }]
[**FORMAT** { **'DNET'** | **'CONCRETE'** | **'XML'** | **NULL** }]
[*common-attributes*]

SOAP および DISH サービスのフォーマット

CREATE SERVICE 文の FORMAT 句により、.NET や Java JAX-RPC など、さまざまな種類の SOAP クライアントに合わせて SOAP サービス・データ・ペイロードがカスタマイズされます。

FORMAT 句は、DISH サービスによって返される WSDL ドキュメントの内容と、SOAP 応答によって返されるデータ・ペイロードのフォーマットに影響します。

デフォルト・フォーマットの DNET は、.NET DataSet フォーマットを必要とする .NET SOAP クライアント・アプリケーションで使用するネイティブ・フォーマットです。

CONCRETE フォーマットは、返されたデータ構造のフォーマットに基づいてインタフェースを自動的に生成する Java JAX-RPC や .NET などのクライアント用です。このフォーマットを指定すると、SQL Anywhere によって返された WSDL ドキュメントは、結果セットを具体的に説明する SimpleDataset 要素を公開します。この要素は、ローの配列から構成されるローセットの包含階層で、それぞれにカラム要素の配列が含まれます。

XML フォーマットは、SOAP 応答を 1 つの大きな文字列として受け入れ、XML パーサによって必要な要素と値を検索して抽出する SOAP クライアントで使用します。通常このフォーマットは、さまざまな種類の SOAP クライアント間で最も手軽です。

SOAP サービスのフォーマットが指定されていない場合は、フォーマットは DISH サービス宣言から継承されます。DISH サービスでもフォーマットが宣言されていない場合、デフォルトは、.NET クライアントと互換性のある DNET です。フォーマットを宣言しない SOAP サービスは、複数の DISH サービスを定義することにより、それぞれ異なる FORMAT タイプを持つさまざまな種類の SOAP クライアントで使用できます。

同種の DISH サービスの作成

SOAP サービスでは、フォーマット・タイプを指定する必要はなく、フォーマット・タイプに NULL を設定できます。この場合は、SOAP サービスのプロキシとして機能する DISH サービスからフォーマットが継承されます。各 SOAP サービスに対して複数の DISH サービスがプロキシの役割をすることができ、それらの DISH サービスは同じ種類でなくてもかまいません。このことは、.NET や Java JAX-RPC など、さまざまな種類の SOAP クライアントで単一の SOAP サービスを使用できることを意味します。DISH サービスは、同じ SOAP サービスに対して、フォーマットは異なっても同じデータ・ペイロードを公開するので、「**同種**」であると見なされます。

たとえば、以下の 2 つの SOAP サービスを考えてみます。どちらもフォーマットを指定していません。

```
CREATE SERVICE "abc/hello"
TYPE 'SOAP'
AS CALL hello(:student);

CREATE SERVICE "abc/goodbye"
TYPE 'SOAP'
AS CALL goodbye(:student);
```

これらのサービスはどちらも FORMAT 句を含んでいないため、フォーマットはデフォルトで NULL になります。したがって、プロキシとして機能している DISH サービスからフォーマットが継承されます。ここで、次のような 2 つの DISH サービスを考えてみます。

```
CREATE SERVICE "abc_xml"
TYPE 'DISH'
GROUP "abc"
FORMAT 'XML';

CREATE SERVICE "abc_concrete"
TYPE 'DISH'
```

```
GROUP "abc"
FORMAT 'CONCRETE';
```

どちらの DISH サービスも同じグループ名 abc を指定しているため、両方が同じ SOAP サービス (主にプレフィクス "abc/" が付いている名前を持つ SOAP サービスすべて) のプロキシとして機能します。

ただし、abc_xml DISH サービスを介して 2 つの SOAP サービスのいずれかがアクセスされた場合、SOAP サービスは XML フォーマットを継承し、abc_concrete SOAP サービスを介してアクセスした場合は、CONCRETE フォーマットを継承します。

同種の DISH サービスは、作成した SOAP Web サービスにさまざまなタイプの SOAP クライアントがアクセスできるようにしたい場合、重複するサービスを避ける手段を提供しています。

チュートリアル：Microsoft .NET からの Web サービスへのアクセス

このチュートリアルでは、Visual C# を使用して Microsoft .NET から Web サービスにアクセスする方法を示します。

♦ **SOAP および DISH サービスを作成するには、次の手順に従います。**

1. SQL Anywhere サンプル・データベースを *samples-dir* から *c:¥webserver¥demo.db* などの別のロケーションにコピーします。
2. コマンド・プロンプトで、次の文を実行して、パーソナル Web サーバを起動します。-xs http(port=80) オプションは、HTTP 要求を受け入れるようにデータベース・サーバに指示します。ポート 80 ですでに Web サーバが実行されている場合、このチュートリアルには 8080 などの別のポート番号を使用します。

   ```
   dbeng10 -xs http(port=80) c:¥webserver¥demo.db
   ```

3. Interactive SQL を起動します。DBA として SQL Anywhere サンプル・データベースに接続します。次の文を実行します。

   a. Employees テーブルをリストする SOAP サービスを定義します。

      ```
      CREATE SERVICE "SASoapTest/EmployeeList"
      TYPE 'SOAP'
      AUTHORIZATION OFF
      SECURE OFF
      USER DBA
      AS SELECT * FROM Employees;
      ```

      認証はオフになっているため、ユーザ名とパスワードを入力せずにだれでもこのサービスを利用できます。このコマンドは、ユーザが DBA の場合に実行できます。この方法は簡単ですが、安全性に優れていません。

   b. 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 DISH サービスを作成します。

      ```
      CREATE SERVICE "SASoapTest_DNET"
      TYPE 'DISH'
      GROUP "SASoapTest"
      FORMAT 'DNET'
      AUTHORIZATION OFF
      SECURE OFF
      USER DBA;
      ```

      SOAP サービスと DISH サービスは、DNET フォーマットである必要があります。この例では、SOAP サービスの作成時に FORMAT 句が省略されています。その結果、SOAP サービスは DISH サービスから DNET フォーマットを継承します。

4. Microsoft Visual C# を起動します。この例では、.NET Framework 2.0 の機能を使用しています。

   a. 新しい Windows 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。

b. [プロジェクト] メニューで [Web 参照の追加] を選択します。

c. [Web 参照の追加] ページの [URL] フィールドに、URL **http://localhost:80/demo/SASoapTest_DNET** を入力します。

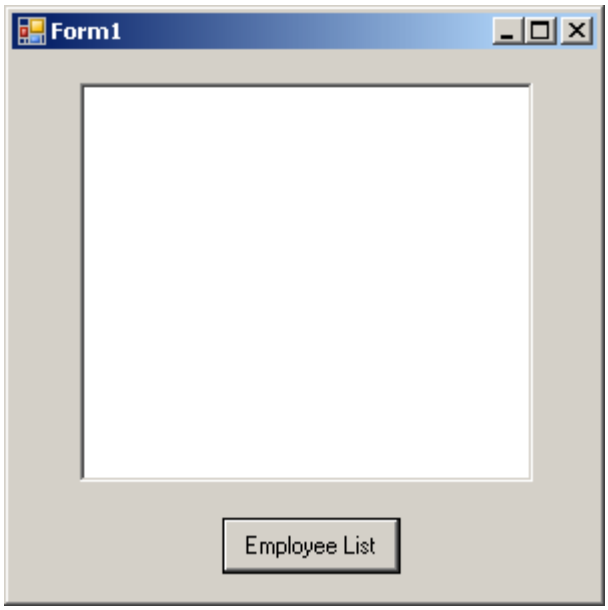
d. [移動] をクリックします。

SASoapTest_DNET で使用可能なメソッドのリストが表示されます。この中に EmployeeList メソッドがあります。

e. [参照の追加] をクリックして完了します。

[ソリューション エクスプローラ] ウィンドウに新しい Web 参照が表示されます。

f. 下の図のように、ListBox と Button をフォームに追加します。



g. ボタンのテキストを **Employee List** に変更します。

h. [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();
while (dr.Read())
{
  for (int i = 0; i < dr.FieldCount; i++)
```



```
        {
          string columnName = dr.GetName(i);
          string value = dr.GetString(i);
          listBox1.Items.Add(columnName+"="+value);
        }
        listBox1.Items.Add("");
      }
      dr.Close();
```

i. プログラムをビルドし、実行します。

   リストボックスに EmployeeList 結果セットが、カラム名=値のペアで表示されます。

チュートリアル：Java JAX-RPC からの Web サービスへのアクセス

次のチュートリアルでは、Java JAX-RPC から Web サービスにアクセスする方法を示します。

JAX-RPC からアクセスできる SQL Anywhere SOAP Web サービスは、CONCRETE フォーマットとして宣言する必要があります。

♦ **SOAP および DISH サービスを作成するには、次の手順に従います。**

1. SQL Anywhere サンプル・データベースを *samples-dir* から *c:¥webserver¥demo.db* などの別のロケーションにコピーします。
2. コマンド・プロンプトで、次の文を実行して、パーソナル Web サーバを起動します。-xs http(port=80) オプションは、HTTP 要求を受け入れるようにデータベース・サーバに指示します。ポート 80 ですでに Web サーバが実行されている場合、このチュートリアルには 8080 などの別のポート番号を使用します。

   ```
   dbeng10 -xs http(port=80) c:¥webserver¥demo.db
   ```

3. Interactive SQL を起動し、DBA として SQL Anywhere サンプル・データベースに接続します。次の文を実行します。

   a. Employees テーブルをリストする SOAP サービスを定義します。「チュートリアル：Microsoft .NET からの Web サービスへのアクセス」 680 ページの手順を実行している場合は、定義済みです。

   ```
   CREATE SERVICE "SASoapTest/EmployeeList"
   TYPE 'SOAP'
   AUTHORIZATION OFF
   SECURE OFF
   USER DBA
   AS SELECT * FROM Employees;
   ```

   認証はオフになっているため、ユーザ名とパスワードを入力せずにだれでもこのサービスを利用できます。このコマンドは、ユーザが DBA の場合に実行できます。この方法は簡単ですが、安全性に優れていません。

   b. 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 DISH サービスを作成します。

   ```
   CREATE SERVICE "SASoapTest_CONCRETE"
   TYPE 'DISH'
   GROUP "SASoapTest"
   FORMAT 'CONCRETE'
   AUTHORIZATION OFF
   SECURE OFF
   USER DBA;
   ```

   SOAP サービスと DISH サービスは、CONCRETE フォーマットである必要があります。この例では、SOAP サービスの作成時に FORMAT 句が省略されています。その結果、SOAP サービスは DISH サービスから CONCRETE フォーマットを継承します。

4. DISH サービスにより自動的に生成される WSDL を見てみます。そのためには、Web ブラウザを開き、URL http://localhost:80/demo/SASoapTest_CONCRETE を参照します。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。

   このサービスのフォーマットは CONCRETE であるため、公開された SimpleDataset オブジェクトに特に注目してください。この後の手順で、wscompile 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。

```
<types>
<s:schema attributeFormDefault="qualified"
  elementFormDefault="qualified"
  targetNamespace=
  "http://localhost/demo/SASoapTest_CONCRETE">
<s:import namespace="http://www.w3.org/2001/XMLSchema" />
<s:complexType name="SimpleDataset">
<s:sequence>

<s:element name="rowset">
<s:complexType>
<s:sequence>
<s:element name="row" minOccurs="0" maxOccurs="unbounded">
<s:complexType>
<s:sequence>
<s:any minOccurs="0" maxOccurs="unbounded" />

</s:sequence>
</s:complexType>
</s:element>
</s:sequence>
</s:complexType>
</s:element>
</s:sequence>
</s:complexType>
<s:element name="error" type="s:string" />
<s:element name="EmployeeList">
<s:complexType>

<s:sequence />
</s:complexType>
</s:element>
<s:element name="EmployeeListResponse">
<s:complexType>
<s:sequence>
<s:element minOccurs="1" maxOccurs="1"
  name="EmployeeListResult"
  type="s3:SimpleDataset" />
<s:element name="sqlcode" type="s:int" />
</s:sequence>
</s:complexType>
</s:element>
</s:schema>
</types>
```

このチュートリアルの次の項では、Java からこれらのサービスにアクセスします。これを行うには、Sun から入手できる Java Web Services Developer Pack を使用する必要があります。

Java JAX-RPC ツールをダウンロードするには、http://java.sun.com/webservices/ を参照してください。この例は、Java Web Services Developer Pack 2.0 for Windows を使用して開発されたものです。

♦ **JAX-RPC インタフェースを生成し Web サービスで使用するには、次の手順に従います。**

1. CLASSPATH 環境変数を設定します。この例では、Java Web Services Developer Pack 2.0 と Sun Java 1.5 JDK がドライブ *C:* にインストールされています。

   ```
   set classpath=.;c:¥jdk1.5.0_06¥jre¥lib¥rt.jar;
   c:¥Sun¥jwsdp-2.0¥jaxrpc¥lib¥jaxrpc-api.jar;
   c:¥Sun¥jwsdp-2.0¥jaxrpc¥lib¥jaxrpc-impl.jar;
   c:¥Sun¥jwsdp-2.0¥jwsdp-shared¥lib¥activation.jar;
   c:¥Sun¥jwsdp-2.0¥jwsdp-shared¥lib¥mail.jar;
   c:¥Sun¥jwsdp-2.0¥saaj¥lib¥saaj-api.jar;
   c:¥Sun¥jwsdp-2.0¥saaj¥lib¥saaj-impl.jar;
   c:¥Sun¥jwsdp-2.0¥fastinfoset¥lib¥FastInfoset.jar;
   c:¥Sun¥jwsdp-2.0¥sjsxp¥lib¥jsr173_api.jar
   ```

2. Java Web Services バイナリと JDK がパスに含まれるように PATH 環境変数を設定します。この例では、Java Web Services Developer Pack 2.0 と Sun Java 1.5 JDK がドライブ *C:* にインストールされています。バイナリは次のディレクトリにあります。

   ```
   c:¥Sun¥jwsdp-2.0¥jaxrpc¥bin
   c:¥Sun¥jwsdp-2.0¥jwsdp-shared¥bin
   c:¥jdk1.5.0_06¥bin
   ```

3. 次に、単純な XML ドキュメントを作成し、wscompile を使用してコンパイルします。この構成ドキュメントのサンプルは、Sun から入手できます。それに対して、**wsdl** 要素のロケーション属性を DISH サービスの URL で置換するだけです。オプションで、パッケージ・ファイル名も指定できます。パッケージ名により、生成された Java ファイルと Class ファイルは同じ名前のサブディレクトリに格納されます。

   テキスト・エディタを使用して、次の内容の *config.xml* という XML ドキュメントを作成します。

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <configuration
     xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
     <wsdl
       location="http://localhost:80/demo/SASoapTest_CONCRETE"
       packageName="sqlanywhere" />
   </configuration>
   ```

   この場合、指定のロケーションは DISH サービスの URL です。さらに、生成されたすべてのファイルが *sqlanywhere* という新しいサブディレクトリに格納されるように、オプションの packageName 属性が追加されました。これが生成されたクラス・ファイルのパッケージ名です。

4. コマンド・プロンプトで次のコマンドを実行します。

   ```
   wscompile -gen -keep config.xml
   ```

   -gen オプションは、指定の URL から WSDL ドキュメントを取り出し、そのためのインタフェースをコンパイルするよう wscompile に指示します。-keep オプションは、*.java* ファイルを削除しないよう wscompile に指示します。このオプションが指定されない場合は、対応する *.class* ファイルの生成後にこれらのファイルは削除されます。これらのファイルを保存すると、インタフェースの構成の検査が簡単になります。

   このコマンドが完了すると、以下の Java ファイルと、各 Java ファイルがコンパイルされた *.class* ファイルを含む *sqlanywhere* というサブディレクトリが作成されています。


```
EmployeeList.java
EmployeeListResponse.java
EmployeeListResponse_LiteralSerializer.java
EmployeeList_LiteralSerializer.java
FaultMessage.java
Row.java
Rowset.java
Rowset_LiteralSerializer.java
Row_LiteralSerializer.java

EmployeeList_LiteralSerializer.java
SASoapTest_CONCRETE.java
SASoapTest_CONCRETESoapPort.java
SASoapTest_CONCRETESoapPort_EmployeeList_Fault_SOAPBuilder.java
SASoapTest_CONCRETESoapPort_EmployeeList_Fault_SOAPSerializer.java
SASoapTest_CONCRETESoapPort_Stub.java
SASoapTest_CONCRETE_Impl.java
SASoapTest_CONCRETE_SerializerRegistry.java
SimpleDataset.java
SimpleDataset_LiteralSerializer.java
```

5. 次の Java ソース・コードを *SASoapDemo.java* として保存します。

```
// SASoapDemo.java illustrates a web service client that
// calls the SASoapTest_CONCRETE service and prints out
// the data.

import java.util.*;
import sqlanywhere.*;

public class SASoapDemo
{
  public static void main( String[] args )
  {
    try {
      SASoapTest_CONCRETE_Impl service =
                    new SASoapTest_CONCRETE_Impl();
      SASoapTest_CONCRETESoapPort port =
                    service.getSASoapTest_CONCRETESoap();

      // This is the SOAP service call to EmployeeList
      EmployeeListResponse response = port.employeeList();
      SimpleDataset result = response.getEmployeeListResult();
      Rowset rowset = result.getRowset();
      Row[] row = rowset.getRow();


      for ( int i = 0; i < row.length; i++ ) {
        // Column data is contained as a SOAPElement
        javax.xml.soap.SOAPElement[] col = row[i].get_any();
        for ( int j = 0; j < col.length; j++ ) {
          System.out.print(col[j].getLocalName() + "=" +
            col[j].getValue() + " " );
        }
        System.out.println();
        System.out.println();
      }

    }
    catch (Exception x) {
      x.printStackTrace();
    }
```

```
    }
}
```

6. *SASoapDemo.java* をコンパイルします。

   ```
   javac SASoapDemo.java
   ```

7. コンパイル済みのクラス・ファイルを実行します。

   ```
   java SASoapDemo
   ```

   EmployeeList 結果セットが、カラム名=値のペアで表示されます。生成される出力は次のようになります。

   ```
   EmployeeID=102 ManagerID=501
   Surname=Whitney GivenName=Fran DepartmentID=100
   Street=9 East Washington Street City=Cornwall
   State=NY Country=USA PostalCode=02192
   Phone=6175553985 Status=A SocialSecurityNumber=017349033
   Salary=45700.000 StartDate=1984-08-28 TerminationDate=null
   BirthDate=1958-06-05 BenefitHealthInsurance=1
   BenefitLifeInsurance=1 BenefitDayCare=0 Sex=F

   EmployeeID=105 ManagerID=501
   Surname=Cobb GivenName=Matthew DepartmentID=100
   Street=7 Pleasant Street City=Grimsby
   State=UT Country=USA PostalCode=02154
   Phone=6175553840 Status=A SocialSecurityNumber=052345739
   Salary=62000.000 StartDate=1985-01-01 TerminationDate=null
   BirthDate=1960-12-04 BenefitHealthInsurance=1
   BenefitLifeInsurance=1 BenefitDayCare=0 Sex=M
   ```

HTML ドキュメントを提供するプロシージャの使用

一般的に、特定のサービスに送信される要求を処理するプロシージャを記述するのが最も簡単です。このようなプロシージャは Web ページを返します。オプションで、プロシージャは出力をカスタマイズするために、URL の一部として渡される引数を受け入れることができます。

しかし、次の例はさらに単純です。これはサービスの単純さを表しています。この Web サービスは "Hello world!" というフレーズを返すだけです。

```
CREATE SERVICE hello
TYPE 'RAW'
AUTHORIZATION OFF
USER DBA
AS SELECT 'Hello world!';
```

Web 要求の処理を可能にするために -xs オプションを指定してデータベース・サーバを起動し、任意の Web ブラウザから URL http://localhost/hello を要求します。Hello world! という言葉が無地のページに表示されます。

HTML ページ

上記のページは、使用しているブラウザにプレーン・テキストで表示されます。これは、デフォルトの HTTP コンテンツタイプが text/plain であるためです。HTML でフォーマットされたごく普通の Web ページを作成するには、次の 2 つの作業を行ってください。

- HTTP コンテンツタイプ・ヘッダ・フィールドを text/html に設定して、ブラウザが HTML を予期するようにします。
- 出力に HTML タグを含めます。

出力にタグを書き込むには 2 つの方法があります。1 つは、CREATE SERVICE 文で TYPE 'HTML' フレーズを使用する方法です。この方法では、SQL Anywhere データベース・サーバは、HTML タグを追加するよう指示されます。これは、たとえばテーブルを返す場合などにうまく機能します。

もう 1 つは、TYPE 'RAW' を使用して必要なタグをすべて自分で書き出す方法です。この 2 番目の方法は出力を最も制御できます。RAW タイプを指定しても、出力が必ずしも HTML または XML フォーマットではないという意味ではありません。これは、タグ自身を追加せずにクライアントに直接戻り値を渡せるということを SQL Anywhere に通知するだけです。

次のプロシージャでは、より凝ったバージョンの Hello world を生成します。便宜上、本文については次のプロシージャで扱いますが、これは Web ページをフォーマットするものです。

組み込みプロシージャ sa_set_http_header を使用して HTTP ヘッダ・タイプを指定するので、ブラウザは適切に結果を解釈します。この文を省略すると、ブラウザは、HTML コードをドキュメントのフォーマットに使用せず、すべての HTML コードを表示します。

```
CREATE PROCEDURE hello_pretty_world ()
RESULT (html_doc XML)
BEGIN
  CALL dbo.sa_set_http_header( 'Content-Type', 'text/html' );
  SELECT HTML_DECODE(
    XMLCONCAT(
```

```
        '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">',
        XMLELEMENT('HTML',

        XMLELEMENT('HEAD',
           XMLELEMENT('TITLE', 'Hello Pretty World')
         ),
         XMLELEMENT('BODY',
           XMLELEMENT('H1', 'Hello Pretty World!'),
           XMLELEMENT('P',
             '(If you see the tags in your browser, check that '
             || 'the Content-Type header is set to text/html.)'
           )
         )
        )
      )
    );
END
```

次の文は、このプロシージャを使用するサービスを作成します。この文は、Hello Pretty World の Web ページを生成する上記のプロシージャを呼び出します。

```
CREATE SERVICE hello_pretty_world
TYPE 'RAW'
AUTHORIZATION OFF
USER DBA
AS CALL hello_pretty_world();
```

いったんプロシージャとサービスを作成したら、Web ページへのアクセスが可能になります。正しい -xs オプション値を指定してデータベース・サーバを起動していることを確認してください。確認したら、URL http://localhost/hello_pretty_world を Web ブラウザで開きます。

Hello Pretty World というタイトルの、単純な HTML ページでフォーマットされた結果が表示されます。Web ページをより凝ったものにするには、コンテンツを増やす、より多くのタグやスタイル・シートを使用する、ブラウザで実行するスクリプトを使用する、などを行ってください。どのような場合でも、ブラウザの要求を処理するためには必要なサービスを作成する必要があります。

組み込みのストアド・プロシージャの詳細については、「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。

ルート・サービス

URL にサービス名が含まれていない場合、SQL Anywhere は **root** という名前の Web サービスを検索します。ルート・ページの役割は、従来の多くの Web サーバにおける *index.html* ページの役割に似ています。

ルート・サービスは、Web サイトのアドレスのみを含む URL 要求を処理できるので、ホーム・ページの作成に便利です。たとえば、次のプロシージャとサービスは、URL http://localhost をブラウズした場合に表示される簡単な Web ページを実装しています。

```
CREATE PROCEDURE HomePage()
RESULT (html_doc XML)
BEGIN
  CALL dbo.sa_set_http_header( 'Content-Type', 'text/html' );
  SELECT HTML_DECODE(
    XMLCONCAT(
      '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">',
      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!'
          )
        )
      )
    )
  );
END
```

次に、このプロシージャを使用するサービスを作成します。

```
CREATE SERVICE root
TYPE 'RAW'
AUTHORIZATION OFF
USER DBA
AS CALL HomePage()
```

データベース・サーバを起動するときにデータベース名が必須であると指定しないかぎり、URL http://localhost をブラウズすることでこの Web ページにアクセスできます。

詳細については、「Web 要求を受信するデータベース・サーバの起動」 670 ページを参照してください。

例

samples-dir¥SQLAnywhere¥HTTP ディレクトリには、より多くのサンプルが用意されています。

データ型の使用

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。

DATATYPE { OFF | ON | IN | OUT }

- **OFF**　DATATYPE オプションが使用されないときのデフォルトの動作です。DNET 出力フォーマットでは、SQL Anywhere データ型は XML スキーマの String 型との間で変換されます。CONCRETE フォーマットと XML フォーマットの場合は、データ型情報が生成されません。
- **ON**　データ型情報は、入力パラメータと結果セットの応答の両方に対して生成されます。SQL Anywhere データ型は XML スキーマのデータ型との間で変換されます。
- **IN**　データ型情報は、入力パラメータのみに対して生成されます。
- **OUT**　データ型情報は、結果セット応答のみに対して生成されます。

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復号化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 (データ型は 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
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)
UNIQUEIDENTIFIERSTR	string	12345678-1234-5678-9012-123456789012
BIGINT	long	-9223372036854775807
UNSIGNED BIGINT	unsignedLong	18446744073709551615
BIT	boolean	1
DECIMAL	decimal	-1.756000
DOUBLE	double	.555555555555555582

SQL Anywhere の型	XML スキーマの型	XML の例
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
VARBIT	string	11111111
LONG VARBIT	string	00000000000000001000000000000000
DATE	date	2006-11-21-05:00
DATETIME	dateTime	2006-05-21T09:00:00.000-08:00
SMALLDATETIME	dateTime	2007-01-15T09:00:00.000-08:00
TIME	time	14:14:48.980-05:00
TIMESTAMP	dateTime	2007-01-12T21:02:14.420-06:00
BINARY	base64Binary	AAAAZg==
IMAGE	base64Binary	AAAAZg==
LONG BINARY	base64Binary	AAAAZg==
UNIQUEIDENTIFIER	string	12345678-1234-5678-9012-123456789012
VARBINARY	base64Binary	AAAAZg==

チュートリアル：Microsoft .NET でのデータ型の使用

このチュートリアルでは、Visual C# を使用して Microsoft .NET 内から SQL Anywhere Web サービス・データ型のサポートを使用する方法を示します。

♦ **SOAP および DISH サービスを作成するには、次の手順に従います。**

1. SQL Anywhere サンプル・データベースを *samples-dir* から *c:¥webserver¥demo.db* などの別のロケーションにコピーします。
2. コマンド・プロンプトで、次の文を実行して、パーソナル Web サーバを起動します。-xs http(port=80) オプションは、HTTP 要求を受け入れるようにデータベース・サーバに指示します。ポート 80 ですでに Web サーバが実行されている場合、このチュートリアルには 8080 などの別のポート番号を使用します。

   ```
   dbeng10 -xs http(port=80) c:¥webserver¥demo.db
   ```

3. Interactive SQL を起動します。DBA として SQL Anywhere サンプル・データベースに接続します。次の文を実行します。

   a. Employees テーブルをリストする SOAP サービスを定義します。

      ```
      CREATE SERVICE "SASoapTest/EmployeeList"
      TYPE 'SOAP'
      AUTHORIZATION OFF
      SECURE OFF
      USER DBA
      DATATYPE OUT
      AS SELECT * FROM Employees;
      ```

      この例では、DATATYPE OUT が指定されているため、XML 応答でデータ型情報が有効になります。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' />
      ```

b. 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 DISH サービスを作成します。

```
CREATE SERVICE "SASoapTest_DNET"
TYPE 'DISH'
GROUP "SASoapTest"
FORMAT 'DNET'
AUTHORIZATION OFF
SECURE OFF
USER DBA;
```

SOAP サービスと DISH サービスは、DNET フォーマットである必要があります。この例では、SOAP サービスの作成時に FORMAT 句が省略されています。その結果、SOAP サービスは DISH サービスから DNET フォーマットを継承します。

4. Microsoft Visual C# を起動します。この例では、.NET Framework 2.0 の機能を使用しています。

   a. 新しい Windows 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。

   b. [プロジェクト] メニューで [Web 参照の追加] を選択します。

   c. [Web 参照の追加] ページの [URL] フィールドに、URL **http://localhost:80/demo/SASoapTest_DNET** を入力します。

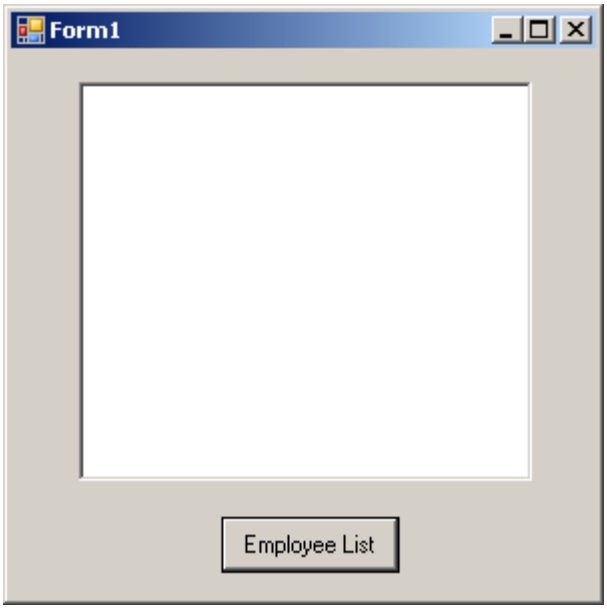
   d. [移動] をクリックします。

   SASoapTest_DNET で使用可能なメソッドのリストが表示されます。この中に EmployeeList メソッドがあります。

   e. [参照の追加] をクリックして完了します。

   [ソリューション エクスプローラ] ウィンドウに新しい Web 参照が表示されます。

   f. 下の図のように、ListBox と Button をフォームに追加します。



g. ボタンのテキストを **Employee List** に変更します。

h. [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();
while (dr.Read())
{
  for (int i = 0; i < dr.FieldCount; i++)
  {
    string columnName = dr.GetName(i);
    string typeName = dr.GetDataTypeName(i);
    columnName = columnName + "(" + typeName + ")";
    if (dr.IsDBNull(i))
    {
      listBox1.Items.Add(columnName + "=(null)");
    }
    else {
      System.TypeCode typeCode =
          System.Type.GetTypeCode(dr.GetFieldType(i));
      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;
            case System.TypeCode.DBNull:
              listBox1.Items.Add(columnName
                + "=(null)");
              break;
            default:
              listBox1.Items.Add(columnName
                + "=(unsupported)");
              break;
          }
        }
      }
      listBox1.Items.Add("");
    }
    dr.Close();
```

この例は、アプリケーション開発者が利用可能なデータ型情報に対して詳細に制御する様子を示すためのものです。

i. プログラムをビルドし、実行します。

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。フォーマットされた結果セットの最初のローは、次のとおりです。

```
<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>1984-08-28-05:00</StartDate>
 <TerminationDate xsi:nil="true" />
 <BirthDate>1958-06-05-05:00</BirthDate>
 <BenefitHealthInsurance>1</BenefitHealthInsurance>
 <BenefitLifeInsurance>1</BenefitLifeInsurance>
 <BenefitDayCare>0</BenefitDayCare>
```



```
 <Sex>F</Sex>
</row>
```

XML 応答について、注意しなければならない点がいくつかあります。

- 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。
- 日付や時刻の情報が格納されたカラムには、Web サーバの UTC からのオフセットが含まれます。この例では、オフセットは -05:00 であり、これは UTC から西に 5 時間 (この場合はアメリカ東部標準時) であることを意味します。
- 日付だけが格納されたカラムは、yyyy-mm-dd-HH:MM または yyyy-mm-dd+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。
- 時刻だけが格納されたカラムは、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。
- 日付と時刻が格納されたカラムは、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。

リストボックスに EmployeeList 結果セットが、カラム名(型)=値のペアで表示されます。結果セットの最初のローを処理した結果は、次のとおりです。

```
EmployeeID(Int32)=102
ManagerID(Int32)=501
Surname(String)=Whitney
GivenName(String)=Fran
DepartmentID(Int32)=100
Street(String)=9 East Washington Street
City(String)=Cornwall
State(String)=New York
Country(String)=USA
PostalCode(String)=02192
Phone(String)=6175553985
Status(String)=A
SocialSecurityNumber(String)=017349033
Salary(String)=$45,700.00
StartDate(DateTime)=28/08/1984 0:00:00 AM
TerminationDate(DateTime)=(null)
BirthDate(DateTime)=05/06/1958 0:00:00 AM
BenefitHealthInsurance(Boolean)=True
BenefitLifeInsurance(Boolean)=True
BenefitDayCare(Boolean)=False
Sex(String)=F
```

結果について、注意しなければならない点がいくつかあります。

- null が格納されたカラムは、型 DBNull として返されます。
- 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。
- 日付が格納されているが時刻の値が含まれないカラムは、時刻を 00:00:00 つまり午前 0 時と想定します。

Web サービス・クライアント関数とプロシージャの作成

SQL Anywhere データベースは、Web サービスを提供すると同時に、Web サービスを消費します。Web サービスは、それがクライアント・プロシージャまたは関数と同じデータベースに存在していないかぎり、インターネットで利用できる標準の Web サービスである場合や、SQL Anywhere データベースによって提供される Web サービスである場合があります。

SQL Anywhere は、HTTP と SOAP の両方の Web サービス・クライアントとして機能できます。この機能は、ストアド関数またはストアド・プロシージャにより提供されます。

クライアント関数とプロシージャは、以下の SQL 文を使用して作成し、操作します。

- 「CREATE FUNCTION 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「CREATE PROCEDURE 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「ALTER FUNCTION 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「ALTER PROCEDURE 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「DROP 文」『SQL Anywhere サーバ - SQL リファレンス』

たとえば、Web サービス・クライアント関数の作成に使用される CREATE FUNCTION 文と CREATE PROCEDURE 文の構文は、次のとおりです。

CREATE FUNCTION [*owner.*]*procedure-name* **(** [*parameter*, …] **)**
RETURNS *data-type*
URL *url-string*
[*proc-attributes*]

CREATE PROCEDURE [*owner.*]*procedure-name* **(** [*parameter*, …] **)**
URL *url-string*
[*proc-attributes*]

この構文の鍵となるのは、URL 句です。この句は、プロシージャでアクセスする Web サービスの URL を提供するために使用されます。URL 句の基本的な構文は、次のとおりです。

url-string :
'{ HTTP | HTTPS | HTTPS_FIPS }://[*user*:*password*@]*hostname*[:*port*][/*path*]**'**

オプションのユーザおよびパスワード情報により、認証を必要とする Web サービスにアクセスできます。ホスト名には、Web サービスを提供しているコンピュータの名前または IP アドレスを使用できます。

ポート番号は、サーバがデフォルト以外のポート番号で受信する場合のみ必要です。デフォルトのポート番号は、HTTP サーバの場合は 80 で、HTTPS サービスの場合は 443 です。

パスは、サーバ上のリソースまたは Web サービスを識別します。

要求は、別の SQL Anywhere データベースによって提供されたものか、インターネットで利用可能なものかにかかわらず、どの Web サービスにも送信できます。Web サービスは、同じデータベース・サーバによって提供されていてもかまいませんが、クライアント関数と同じデータベースにあるものは使用できません。同じデータベース内の Web サービスにアクセスしようとすると、「403 Forbidden」のエラーが返されます。

感嘆符は代入パラメータに使用されるため、プロシージャ定義の文字列のどこかに含まれる感嘆符はエスケープする必要があります。詳細については、「! 文字のエスケープ」 712 ページを参照してください。

一般的な句

プロシージャ・コールに関する追加の詳細情報を提供するためのその他の句には、以下があります。

proc-attributes :
[**TYPE** { **'HTTP**[:{ **GET** | **POST**[:*MIME-type*] }]' |
'SOAP[:{**RPC**|**DOC**}]' }]
[**NAMESPACE** *namespace-string*]
[**CERTIFICATE** *certificate-string*]
[**CLIENTPORT** *clientport-string*]
[**PROXY** *proxy-string*]

TYPE 句は、SQL Anywhere に Web サービス・プロバイダへの要求のフォーマット方法を指定するために重要です。標準の SOAP タイプの RPC と DOC を使用できます。GET や POST などの標準の HTTP メソッドも使用可能で、それぞれ HTTP:GET および HTTP:POST と指定されます。HTTP を指定すると、HTTP:POST を暗黙で指定したことになります。

タイプ SOAP が選択されると、SQL Anywhere は自動的に要求を SOAP 要求に必要な標準フォーマットの XML ドキュメントとしてフォーマットします。SOAP 要求は常に XML ドキュメントであるため、タイプ SOAP が選択されるたびに、常に HTTP POST 要求を暗黙的に使用して SOAP 要求ドキュメントがサーバに送信されます。タイプ SOAP は、SOAP:RPC を暗黙で指定します。

Web サービス・クライアント関数とプロシージャの名前

出力 SOAP 要求の構築時には、プロシージャ名が SOAP 操作名として使用されます。さらに、パラメータの名前も SOAP 要求エンベロープのタグ名に表示されます。したがって、これらの名前を SOAP サーバで必要なとおりに正しく指定することは、SOAP ストアド・プロシージャの定義において重要なポイントです。これは、SOAP プロシージャと関数の名前には、SQL Anywhere のプロシージャ名と関数名に適用されるもの以外にも、制約が加えられることを意味します。

次のプロシージャ定義は、これを具体的に示しています。

```
CREATE PROCEDURE MyOperation ( a INTEGER, b CHAR(128) )
URL 'HTTP://localhost'
TYPE 'SOAP:DOC';
```

次の文などにより、このプロシージャが呼び出されると、SOAP 要求が生成されます。

```
CALL MyOperation( 123, 'abc' )
```

プロシージャ名 (この例では MyOperation) は、要求本文内の <m:MyOperation> タグに表示されます。さらに、プロシージャに対する 2 つのパラメータ a と b は、それぞれ <m:a> と <m:b> になります。

```
<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">
  <SOAP-ENV:Body>
```



```
    <m:MyOperation>
      <m:a>123</m:a>
      <m:b>abc</m:b>
    </m:MyOperation>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

ネームスペース URI

すべての SOAP 要求は、メソッド・ネームスペース URI を必要とします。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。

SOAP:DOC または SOAP:RPC の SOAP 関数またはプロシージャを作成する場合、ネームスペース URI を指定しなければ呼び出しが成功しない可能性があります。必要なネームスペース値は、WSDL 記述ドキュメントやサービスのマニュアルから取得できます。NAMESPACE 句は、SOAP 関数とプロシージャのみに適用します。デフォルトのネームスペース値はプロシージャの URI で、オプションのパス・コンポーネントとユーザおよびパスワードの値は含みません。

HTTPS 要求

セキュア HTTP 要求を発行するためには、クライアントはサーバの証明書、またはサーバの証明書の署名に使用する証明書にアクセスします。この証明書によって、SQL Anywhere で要求を暗号化する方法が指定されます。証明書の値は、セキュアでないサーバ宛ての要求をセキュア・サーバへリダイレクトする場合にも必要です。

証明書情報を提供するには、2 つの方法があります。証明書をファイルに保存してファイル名を指定する方法と、証明書全体を文字列値として提供する方法です。両方を行うことはできません。

証明書属性は、次のようにセミコロンで区切られた key=value ペアとして構成された文字列値として提供されます。

certificate-string :
{ **file=***filename* | **certificate=***string* } ; **company=***company* ; **unit=***company-unit* ; **name=** *common-name*

次のキーを使用できます。

キー	省略形	説明
file		証明書のファイル名
certificate	cert	証明書そのもの。Base64 形式でエンコードされています。
company	co	証明書で指定された会社
unit		証明書で指定された会社の部門
name		証明書で指定された共通名

たとえば、次の文は、クライアントと同じコンピュータ上にある Web サービスへの安全な要求を行うプロシージャを作成します。


```
CREATE PROCEDURE test()
URL 'HTTPS://localhost/myservice'
CERTIFICATE 'file=C:¥srv_cert.crt;co=iAnywhere;
        unit=SA;name=JohnSmith';
```

TYPE 句が含まれていないため、要求は SOAP:RPC タイプであると見なされます。サーバのパブリック証明書は、*C:¥srv_cert.crt* ファイルにあります。

クライアント・ポート

ファイアウォールを介して Web サービスにアクセスする場合、サーバへの接続を確立するときに使用するポートを SQL Anywhere に対して指定する必要があることがよくあります。通常、ポート番号は動的に取得されるため、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、デフォルトの動作を使用してください。

ClientPort オプションは、クライアント・アプリケーションが TCP/IP を使って通信するポート番号を指定します。次の例で示すように、単一のポート番号、または個々のポート番号の組み合わせやポート番号の範囲を指定することができます。

```
CREATE PROCEDURE test ()
URL 'HTTPS://localhost/myservice'
CLIENTPORT '5040,5050-5060,5070';
```

ポート番号のリストまたは範囲を指定することをおすすめします。ポート番号を 1 つだけ指定すると、アプリケーションが維持できるのは、一度に 1 つの接続のみとなります。また、1 つの接続を閉じた後は、短いタイムアウト時間が発生します。その間、同じリモート・サーバとポートを使って新しい接続は作成できません。ポート番号のリストや範囲を指定すると、アプリケーションは、いずれかのポート番号との接続が確立するまで、試行を続けます。

この機能は、ClientPort ネットワーク・プロトコル・オプションと類似しています。詳細については、「ClientPort プロトコル・オプション [CPORT]」 『SQL Anywhere サーバ - データベース管理』を参照してください。

プロキシの使用

プロキシ・サーバを使用して実行しなければならない Web サービス要求もあります。そのような場合は、PROXY 句を使用してプロキシ・サーバの URL を指定します。

値のフォーマットは、URL 句と同じですが、ユーザ、パスワード、パス値などは無視されます。

proxy-string :
'{ HTTP | **HTTPS }://**[*user*:*password*@]*hostname*[:*port*][*/path*]'

プロキシ・サーバを指定すると、SQL Anywhere は要求をフォーマットし、指定されたプロキシ URL を使用してそれをプロキシ・サーバに送ります。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。

HTTP ヘッダの修正

CREATE PROCEDURE 文を使用して Web サービスのプロシージャを作成する場合、HTTP HEADER 名をコロン (:) と値のどちらも含めずに指定すると、HTTP クライアント・アプリケーションではヘッダを表示しなくなります。コロンは含めても値を指定しないと、ヘッダ名は含まれますが、値は含まれません。次に例を示します。


```
CREATE PROCEDURE suds(...)
TYPE 'SOAP:RPC'
URL '...'
HEADER 'SOAPAction¥nDate¥nFrom:';
```

この例では、Action と Data HTTP ヘッダ (SQL Anywhere で自動生成されるヘッダ) は表示されず、From ヘッダは含まれますが値はありません。

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 ヘッダは、通常は自動的に生成されるため、不注意で修正しないようにしてください。

Accept-Charset	常に自動的に生成されます。変更や削除によって、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。デッドロックの原因になる可能性があるため、クライアントが自分自身 (同じサーバ) 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。変更や削除によって、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。クライアントは、永続的接続をサポートしません。接続がハングする可能性があるため、変更しないでください。
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。
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。

戻り値と結果セットの使用

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ストアド・プロシージャのどちらでも実行できます。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、この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 は以下のデータ・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、組み込みの Open XML 処理機能を使用してもアクセスできます。

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ストアド関数は、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、結果セットの 2 つのカラムを OPENXML プロシージャを使用して抽出する方法を示しています。それらは *service_name* カラムと *secure_required* カラムで、SOAP サービスがセキュアで、HTTPS を必要とすることを示しています。

```
SELECT *
FROM openxml( get_webservices(), '//row' )
WITH ("Name"    char(128) 'service_name',
     "Secure?" char(1)   'secure_required' );
```

この文は、*ロー・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* 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

SQL Anywhere で使用できる XML 処理機能の詳細については、「データベースにおける XML の使用」 『SQL Anywhere サーバ - SQL の使用法』を参照してください。

その他のタイプのプロシージャ

その他のタイプのプロシージャは、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

Attribute	Value
Server	GWS/2.1
Content-Length	2234
Date	Mon, 18 Oct 2004, 16:00:00 GMT

結果セットからの選択

SELECT 文を使用して、結果セットから値を取り出します。取り出した値はテーブルに保存したり、変数を設定したりするために使用します。

```
CREATE PROCEDURE test( INOUT parm CHAR(128) )
URL 'HTTP://localhost/test'
TYPE 'HTTP';
```

このプロシージャは、HTTP タイプであるため、前の項で説明した 2 つのカラムから成る結果セットを返します。最初のカラムは属性名、2 番目のカラムは属性値です。キーワードは、HTTP 応答ヘッダ・フィールドにあるものと同様です。Body 属性には、メッセージの本文が含まれ、これは通常 HTML ドキュメントです。

以下のように結果セットをテーブルに挿入する方法があります。

```
CREATE TABLE StoredResults(
    Attribute LONG VARCHAR,
    Value     LONG VARCHAR
);
```

結果セットは、次のようにこのテーブルに挿入します。

```
INSERT INTO StoredResults SELECT *
FROM test('Storing into a table')
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 Value
FROM test('Calling test for the Status Code')
WITH (Attribute LONG VARCHAR, Value LONG VARCHAR)
WHERE Attribute = 'Status';
```

この文は、結果セットからステータス情報のみを選択します。このようにして、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。

パラメータの使用

Web サービス・クライアントとして機能するストアド関数とストアド・プロシージャは、その他の関数やプロシージャと同様にパラメータを使用して宣言できます。パラメータの代入中に使用する場合を除き、これらのパラメータ値は HTTP 要求または SOAP 要求の一部として渡されます。

加えて、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、Web サービス要求の一部として渡されません。

渡されるパラメータ

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。

HTTP 要求

HTTP:GET タイプの要求のパラメータは、URL でエンコードされます。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が要求の本文の一部になります。2 つのパラメータと値の場合、次のテキストがヘッダの後、HTTP 要求の本文に表示されます。

```
a=123&b=xyz
```

SOAP 要求

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">
  <SOAP-ENV:Body>
    <m:test>
      <m:a>123</m:a>
      <m:b>abc</m:b>
    </m:test>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

代入パラメータ

ストアド・プロシージャまたはストアド関数の宣言済みパラメータは、そのプロシージャまたは関数が実行されるたびに、ストアド関数またはストアド・プロシージャ定義内のプレースホルダを自動的に置き換えます。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。

たとえば、次のプロシージャ定義では、URL 全体がパラメータとして渡されます。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。

```
CREATE PROCEDURE test ( url CHAR(128) )
URL '!url'
TYPE 'HTTP:POST';
```

たとえば、次のようなプロシージャを使用できます。

```
CALL test ( 'HTTP://localhost/myservice' );
```

ユーザとパスワード値の非表示

代入パラメータの有効な適用例として、ユーザ名やパスワードなどの機密の値を Web サービス・クライアント関数やプロシージャの定義の一部にすることを避けることがあります。そのような値がプロシージャまたは関数定義でリテラルとして指定されていると、それらはシステム・テーブルに保存され、データベースのすべてのユーザによって簡単にアクセスされてしまいます。このような値をパラメータとして渡すと、この問題を回避できます。

たとえば、次のプロシージャ定義には、ユーザ名とパスワードがプロシージャ定義の一部としてプレーン・テキストで含まれています。

```
CREATE PROCEDURE test
URL 'HTTP://dba:sql@localhost/myservice';
```

ユーザとパスワードをパラメータとして宣言するとこの問題を避けることができます。これにより、ユーザとパスワードの値はプロシージャが呼び出されたときにしか提供されなくなります。次に例を示します。

```
CREATE PROCEDURE test ( uid CHAR(128), pwd CHAR(128) )
URL 'HTTP://!uid:!pwd@localhost/myservice';
```

このプロシージャは次のように呼び出されます。

```
CALL test ( 'dba', 'sql' );
```

別の例として、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。

```
CREATE PROCEDURE secure( cert LONG VARCHAR )
URL 'https://localhost/secure'
TYPE 'HTTP:GET'
CERTIFICATE 'cert=!cert;company=test;unit=test;name=RSA Server';
```

このプロシージャを呼び出すときに証明書を文字列として提供します。次の呼び出し例では、証明書をファイルから読み出します。証明書は、CERTIFICATE 句の **file=** キーワードを使用して、ファイルから直接読み出すことができるため、これは説明のためにのみ行います。

```
CALL secure( xp_read_file('install-dir¥win32¥rsaserver.crt') );
```

! 文字のエスケープ

感嘆符 (!) は Web サービス・クライアントのストアド関数とストアド・プロシージャのコンテキストで、代入パラメータで使用するプレースホルダを識別するために使用するため、プロシージャの属性文字列の一部としてこの文字を含める場合は、エスケープします。そのためには、感嘆符にプレフィクスとしてもう 1 つの感嘆符を付けます。これにより、Web サービス・クライアントまたは Web サービス関数定義内のすべての !! が、! で置換されます。

プレースホルダとして使用されたパラメータ名には、アルファベット文字のみを含めます。さらに、あいまいにならないように、プレースホルダの後にはアルファベット以外の文字を挿入します。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://salesserver/order?size=!sizeXL'
TYPE 'SOAP:RPC'
```

!sizeXL は、一致するパラメータのない有効なプレースホルダであるため、常に削除されます。

パラメータのデータ型変換

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。

構造化されたデータ型の使用

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。このような場合、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 型のパラメータの場合、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。これらの例は、Microsoft SOAP ToolKit 3.0 Round

2 相互運用性テスト・サーバ (http://mssoapinterop.org/stkV3) に対して要求を発行するように設計されています。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 3 つの例では、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。

```
CALL sa_make_object('function', 'echoFloatArray');
ALTER FUNCTION echoFloatArray( inputFloatArray XML )
RETURNS XML
URL 'http://mssoapinterop.org/stkV3/Interop.wsdl'
HEADER 'SOAPAction:"http://soapinterop.org/"'
NAMESPACE 'http://soapinterop.org/';

CALL sa_make_object('function', 'echoStruct');
ALTER FUNCTION echoStruct( inputStruct XML )
RETURNS XML
URL 'http://mssoapinterop.org/stkV3/Interop.wsdl'
HEADER 'SOAPAction:"http://soapinterop.org/"'
NAMESPACE 'http://soapinterop.org/';

CALL sa_make_object('function', 'echoStructArray');
ALTER 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[]' 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

変数の使用

HTTP 要求の変数は、2 種類のソースのいずれかによって指定されます。最初の方法は、さまざまな名前=値のペアが含まれたクエリ文字列を URL で指定することです。HTTP GET 要求はこのような形でフォーマットされます。クエリ文字列を含む URL の例を次に示します。

```
http://localhost/gallery?picture=sunset.jpg
```

2 番目は、URL パスを介した方法です。URL PATH を ON または ELEMENTS に設定すると、サービス名に続くパスの部分が変数値として解釈されます。このオプションにより、データベース内に格納されている何かを示す代わりに、従来のファイルベースの Web サイトのように URL が特定のディレクトリ内のファイルを要求するように指定できます。次はその例です。

```
http://localhost/gallery/sunset.jpg
```

この URL は、gallery というディレクトリからファイル *sunset.jpg* を要求するように見えます。しかし実際は、gallery サービスがこの文字列をパラメータとして受け取ります (このパラメータは、データベース・テーブルから画像を取得するためなどに使用されます)。

HTTP 要求で渡されるパラメータは、URL PATH の設定によって決まります。

- **OFF**　サービス名の後にパス・パラメータを許可しません。
- **ON**　サービス名の後のすべてのパス要素が、変数 URL に割り当てられます。
- **ELEMENTS**　URL パスの残りの部分をスラッシュ文字で区切り、最大で 10 要素をリストできます。これらの値は、変数 URL1、URL2、URL3、…、URL10 と割り当てられます。値が 10 個より少ない場合、残りの変数は NULL に設定されます。11 個以上の変数を指定するとエラーになります。

定義されたロケーション以外は、変数に違いはありません。すべての HTTP 変数に同じようにアクセスし、使用します。たとえば、url1 などの変数値は、**?picture=sunset.jpg** のようなクエリの一部として指定されるパラメータと同じようにアクセスされます。

変数へのアクセス

変数にアクセスする主な方法がいくつかあります。1 つは、サービス宣言の文にある変数を使用する方法です。たとえば、次の文は複数の変数値を ShowTable ストアド・プロシージャに渡します。

```
CREATE SERVICE ShowTable
TYPE 'RAW'
AUTHORIZATION ON
AS CALL ShowTable( :user_name, :table_name, :limit, :start );
```

他の方法としては、要求を処理するストアド・プロシージャ内で組み込み関数 NEXT_HTTP_VARIABLE と HTTP_VARIABLE を使用することです。どの変数が定義されているかがわからない場合は、NEXT_HTTP_VARIABLE を使用して検索できます。HTTP_VARIABLE 関数によって変数値が返されます。

NEXT_HTTP_VARIABLE 関数により、定義された変数の名前で繰り返すことができます。最初にこれを呼び出すときには、NULL 値を渡します。すると、この関数が 1 つの変数名を返しま

す。次にこれを呼び出すときから、前の変数の名前を渡すたびに、次の変数名を返すようになります。最後の変数名がこの関数に渡されると、NULL を返します。

この方法で変数名を繰り返し渡す場合、各変数名が正確に 1 回だけ返されることになります。ただし、変数が返される順番は、要求で指定された順番と同じでない場合もあります。さらに、これを繰り返した場合、2 回目は違う順番で返されます。

各変数の値を取得するには、HTTP_VARIABLE 関数を使用します。最初のパラメータが変数の名前です。追加のパラメータはオプションです。1 つの変数に対して複数の値が指定される場合、1 つのパラメータのみが指定されると関数は最初の値を返します。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、変数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。たとえば、次の SQL 文は 3 つの変数値を検索し、次にイメージ変数のヘッダフィールド値を検索します。

```
SET v_id = HTTP_VARIABLE( 'ID' );
SET v_title = HTTP_VARIABLE( 'Title' );
SET v_descr = HTTP_VARIABLE( 'descr' );

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' );
```

HTTP_VARIABLE 関数を使用して変数に関連付けられている値を取得する例を次に示します。これは、前の項で説明した ShowSalesOrderDetail サービスを変更したものです。

```
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;
```

このストアド・プロシージャを呼び出すサービスは、次のとおりです。

```
CREATE SERVICE ShowDetail
TYPE 'HTML'
URL PATH OFF
AUTHORIZATION OFF
USER DBA
AS CALL ShowDetail();
```

サービスをテストするには、Web ブラウザを開き、次の URL を指定します。

http://localhost:80/demo/ShowDetail?product_id=300&customer_id=101

パラメータの受け渡しの詳細については、「URL の解釈方法の概要」 673 ページと「HTTP 関数と SOAP 関数」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

HTTP ヘッダの使用

HTTP 要求のヘッダは、NEXT_HTTP_HEADER 関数と HTTP_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_HTTP_HEADER 関数は、要求に含まれる 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、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。

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 HTTP 要求での表示順に表示されるとはかぎりません。

HTTP_HEADER 関数は、名前付きの HTTP ヘッダ・フィールドの値を返します。HTTP 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 HTTP 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。

次の表に、典型的な HTTP ヘッダと値の例を示します。

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
UA-CPU	x86
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
Connection	Keep-Alive
@HttpMethod	GET
@HttpURI	/demo/ShowHTTPHeaders
@HttpVersion	HTTP/1.1

各ヘッダの値を取得するには、まず NEXT_HTTP_HEADER 関数を使用してヘッダ名を取得し、HTTP_HEADER 関数を使用してその値を取得します。次の例は、これを行う方法を示します。

```
CREATE PROCEDURE HTTPHeaderExample()
RESULT ( html_string LONG VARCHAR )
BEGIN
   declare header_name long varchar;
   declare header_value 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 );
      -- 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 ) );

   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' ),
            table_rows );
END;
```

この例では、ヘッダの名前と値を HTML テーブルとして出力します。このサンプル・プロシージャの動作を確認するには、次のサービスを定義します。

```
CREATE SERVICE ShowHTTPHeaders
TYPE 'RAW'
AUTHORIZATION OFF
USER DBA
AS CALL HTTPHeaderExample();
```

サービスをテストするには、Web ブラウザを開き、次の URL を指定します。

http://localhost:80/demo/ShowHTTPHeaders

ヘッダ処理の詳細については、「HTTP 関数と SOAP 関数」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

SOAP サービスの使用

Web サービスの数多くの機能を説明するため、まず華氏を摂氏に変換する単純なサンプル・サービスから見ていきます。

♦ **単純な Web サービス・サーバを設定するには、次の手順に従います。**

1. データベースを作成します。

```
dbinit ftc
```

2. このデータベースを使用してサーバを起動します。

```
dbsrv10 -xs http(port=8082) -n ftc ftc.db
```

3. Interactive SQL を使用してサーバに接続します。

```
dbisql -c "UID=DBA;PWD=sql;ENG=ftc"
```

4. Interactive SQL を使用して Web サービスを作成します。

```
CREATE SERVICE FtoCService
TYPE 'SOAP'
FORMAT 'XML'
AUTHORIZATION OFF
USER DBA
AS CALL FToCConvertor( :temperature );
```

5. このサービスが呼び出す、華氏表現の温度を摂氏に変換するために必要な計算を実行するストアド・プロシージャを定義します。

```
CREATE PROCEDURE FToCConvertor( temperature FLOAT )
BEGIN
   SELECT ROUND((temperature - 32.0) * 5.0 / 9.0, 5)
   AS answer;
END;
```

この時点で、SQL Anywhere Web サービス・サーバは実行されており、要求処理の準備ができています。サーバは SOAP 要求をポート 8082 で受信しています。

この SOAP 要求サーバをテストするための最も簡単な方法は、別の SQL Anywhere データベース・サーバで SOAP 要求を送信して応答を取得することです。

♦ **SOAP 要求を送受信するには、次の手順に従います。**

1. 第 2 のサーバで使用するデータベースを新たに作成します。

```
dbinit ftc_client
```

2. このデータベースを使用してパーソナル・サーバを起動します。

```
dbeng10 ftc_client.db
```

3. Interactive SQL の別のインスタンスを使用してパーソナル・サーバに接続します。

```
dbisql -c "UID=DBA;PWD=sql;ENG=ftc_client"
```

4. Interactive SQL を使用してストアド・プロシージャを作成します。

```
CREATE PROCEDURE FtoC( temperature FLOAT )
  URL 'http://localhost:8082/FtoCService'
  TYPE 'SOAP:DOC';
```

URL 句は、SOAP Web サービスを参照するために使用されます。文字列 'http://localhost:8082/FtoCService' は、使用される Web サービスの URI を指定しています。これは、ポート 8082 で受信する Web サーバを指しています。

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。

5. FtoC ストアド・プロシージャのラッパが必要なので、2 つ目のストアド・プロシージャを作成します。

```
CREATE PROCEDURE FahrenheitToCelsius( temperature FLOAT )
BEGIN
   DECLARE result LONG VARCHAR;
   DECLARE err INTEGER;
   DECLARE crsr CURSOR FOR
      CALL FtoC( temperature );

   OPEN crsr;
   FETCH crsr INTO result, err;
   CLOSE crsr;

   SELECT temperature, Celsius
   FROM OPENXML(result, '//tns:answer', 1, result)
      WITH ("Celsius" FLOAT 'text()');
END;
```

このストアド・プロシージャは、Web サービスの呼び出しを隠すプロシージャとして機能します。FtoC ストアド・プロシージャは、このストアド・プロシージャによって処理される結果セットを返します。結果セットは、次のような単一の XML 文字列です。

```
<tns:rowset xmlns:tns="http://localhost/ftc/FtoCService">
 <tns:row>
  <tns:answer>100</tns:answer>
 </tns:row>
</tns:rowset>
```

OPENXML 関数を使用して、返された XML を解析し、摂氏単位の温度を示す値を抽出します。

6. ストアド・プロシージャを呼び出して、要求を送信し、応答を取得します。

```
CALL FahrenheitToCelsius(212);
```

華氏温度 (temperature) とその摂氏 (Celsius) 換算値が表示されます。

temperature	Celsius
212	100

ここまでで、SQL Anywhere Web サーバで実行されている単純な Web サービスのデモを行いました。上記のように、別の SQL Anywhere サーバがこの Web サーバと通信できます。サーバ間でやりとりされる SOAP 要求と応答の内容に対しては、ほとんど制御できません。次の項では、独自の SOAP ヘッダを追加することでこの単純な Web サービスを拡張する方法について見ていきます。

注意
Web サービスは、同じデータベース・サーバによって提供されていてもかまいませんが、クライアント関数と同じデータベースにあるものは使用できません。同じデータベース内の Web サービスにアクセスしようとすると、403 Forbidden エラーが返されます。

SOAP ヘッダ処理の詳細については、「SOAP ヘッダの使用」 726 ページを参照してください。

SOAP ヘッダの使用

この項では、「SOAP サービスの使用」 723 ページで説明した単純な Web サービスを拡張して、SOAP ヘッダを処理します。

前項の手順を実行済みの場合は、手順 1 ー 4 を省略して、手順 5 に進んでください。

♦ **Web サービス・サーバを作成するには、次の手順に従います。**

1. データベースを作成します。

```
dbinit ftc
```

2. このデータベースを使用してサーバを起動します。

```
dbsrv10 -xs http(port=8082) -n ftc ftc.db
```

3. Interactive SQL を使用してサーバに接続します。

```
dbisql -c "UID=DBA;PWD=sql;ENG=ftc"
```

4. Interactive SQL を使用して Web サービスを作成します。

```
CREATE SERVICE FtoCService
TYPE 'SOAP'
FORMAT 'XML'
AUTHORIZATION OFF
USER DBA
AS CALL FToCConvertor( :temperature );
```

5. このサービスが呼び出す、華氏表現の温度を摂氏に変換するために必要な計算を実行するストアド・プロシージャを定義します。前項の例とは異なり、このストアド・プロシージャには特別な SOAP ヘッダを処理するための文が追加されています。前項の例を使用してきた場合は、ストアド・プロシージャを変更することになるので、次の中の CREATE を ALTER に変更します。

```
CREATE PROCEDURE FToCConvertor( 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;
```

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、要求に含まれる 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。この例で SOAP ヘッダを取得する SQL コードは次の部分です。NULL が返されるとループを抜けます。

```
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するとともに、@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="SecretAgent" mustUnderstand="1">
  <userName alias="99">
    <first>Susan</first>
    <last>Hilton</last>
  </userName>
</Authentication>
```

このヘッダの場合、@namespace 属性値は次のとおりです。

```
SecretAgent
```

また、mustUnderstand 属性値は次のとおりです。

```
1
```

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 ( alias LONG VARCHAR '@*:alias',
  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。

alias	first_name	last_name
99	Susan	Hilton

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。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。この情報を使用して何ができるでしょうか。

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。

ストアド・プロシージャでの次の処理は、SOAP フォーマットで応答を作成することです。XML 応答は、次のようにして構築します。

```
SET authinfo =
  XMLELEMENT( 'Authentication',
    XMLATTRIBUTES(
      namespace as xmlns,
      alias,
      mustUnderstand ),
      XMLELEMENT( 'first', first_name ),
      XMLELEMENT( 'last', last_name ) );
```

これにより、次の XML 文字列が構築されます。

```
<Authentication xmlns="SecretAgent" alias="99"
                mustUnderstand="1">
  <first>Susan</first>
  <last>Hilton</last>
</Authentication>
```

最後に、SA_SET_SOAP_HEADER ストアド・プロシージャを使用して、SOAP 応答を呼び出し側に返します。

```
CALL SA_SET_SOAP_HEADER( 'authinfo', authinfo );
```

前項の例のように、最後の手順は華氏から摂氏への変換計算です。

この時点で、前項と同様に、華氏から摂氏への温度変換を行う SQL Anywhere Web サービスが実行されています。ここでの大きな違いは、Web サービスが呼び出し側からの SOAP ヘッダを処理して、SOAP 応答を呼び出し側に返送できることです。

これはまだ全体像の半分にしか達していません。次のステップは、SOAP 要求を送信し、SOAP 応答を受信する、サンプル・クライアントの開発です。

前項の手順を実行済みの場合は、手順 1 ー 3 を省略して、手順 4 に進んでください。

♦ SOAP ヘッダを送受信するには、次の手順に従います。

1. 第 2 のサーバで使用するデータベースを新たに作成します。

   ```
   dbinit ftc_client
   ```

2. このデータベースを使用してパーソナル・サーバを起動します。

   ```
   dbeng10 ftc_client.db
   ```

3. Interactive SQL の別のインスタンスを使用してパーソナル・サーバに接続します。

   ```
   dbisql -c "UID=DBA;PWD=sql;ENG=ftc_client"
   ```

4. Interactive SQL を使用してストアド・プロシージャを作成します。

   ```
   CREATE PROCEDURE FtoC( temperature FLOAT,
      INOUT inoutheader LONG VARCHAR,
      IN inheader LONG VARCHAR )
    URL 'http://localhost:8082/FtoCService'
    TYPE 'SOAP:DOC'
    SOAPHEADER '!inoutheader!inheader';
   ```

   URL 句は、SOAP Web サービスを参照するために使用されます。文字列 'http://localhost:8082/FtoCService' は、使用される Web サービスの URI を指定しています。これは、ポート 8082 で受信する Web サーバを指しています。

   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。

   SQL Anywhere クライアント・プロシージャ (inoutheader、inheader) の代入変数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. FtoC ストアド・プロシージャのラッパが必要なので、次のような 2 つ目のストアド・プロシージャを作成します。前項の例とは異なり、このストアド・プロシージャには、特別な SOAP ヘッダを作成し、それを Web サービス呼び出しに含めて送信し、Web サーバからの応答を処理する文が追加されています。前項の例を使用してきた場合は、ストアド・プロシージャを変更することになるので、次の中の CREATE を ALTER に変更します。

```
CREATE PROCEDURE FahrenheitToCelsius( temperature 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( temperature, 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 temperature, Celsius
  FROM OPENXML(result, '//tns:answer', 1, result)
       WITH ("Celsius" FLOAT 'text()');
END;
```

このストアド・プロシージャは、Web サービスの呼び出しを隠すプロシージャとして機能します。このストアド・プロシージャは、前項の例から拡張されており、2 つの SOAP ヘッダを作成します。最初のヘッダは次のとおりです。

```
<Authentication xmlns="SecretAgent"
                    mustUnderstand="1">
  <userName alias="99">
    <first>Susan</first>
    <last>Hilton</last>
  </userName></Authentication>
```

2 番目のヘッダは次のとおりです。

```
<Session xmlns="SomeSession">123456789</Session>
```

カーソルが開かれると、SOAP 要求は Web サービスに送信されます。

```
<Authentication xmlns="SecretAgent" alias="99"
                    mustUnderstand="1">
<first>Susan</first>
```

```
<last>Hilton</last>
</Authentication>
```

FtoC ストアド・プロシージャは、このストアド・プロシージャによって処理される結果セットを返します。結果セットは次のようになります。

```
<tns:rowset xmlns:tns="http://localhost/ftc/FtoCService">
 <tns:row>
  <tns:answer>100</tns:answer>
 </tns:row>
</tns:rowset>
```

OPENXML 関数を使用して、返された XML を解析し、摂氏単位の温度を示す値を抽出します。

6. ストアド・プロシージャを呼び出して、要求を送信し、応答を取得します。

```
CALL FahrenheitToCelsius(212);
```

華氏温度 (temperature) とその摂氏 (Celsius) 換算値が表示されます。

temperature	Celsius
212.0	100.0

SQL Anywhere Web サービス・クライアントは、関数またはプロシージャとして宣言できます。SQL Anywhere クライアント関数宣言は、実質的に、すべてのパラメータを in モードのみに制限します (パラメータは呼び出された側の関数で代入できません)。SQL Anywhere Web サービスの関数を呼び出すと未加工の SOAP エンベロープ応答が返され、プロシージャを呼び出すと結果セットが返されます。

CREATE/ALTER PROCEDURE/FUNCTION 文には SOAPHEADER 句が追加されています。SOAP ヘッダは、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ます。Web サービス・クライアント関数は in モード代入パラメータを 1 つまたは複数定義でき、Web サービス・クライアント・プロシージャも inout または out 代入パラメータを 1 つ定義できます。したがって、Web サービス・クライアント・プロシージャは、応答 SOAP ヘッダを out (または inout) 代入パラメータに含めて返すことができます。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。

次の例は、クライアントが、いくつかのヘッダ・エントリをパラメータを使用して送信し、応答 SOAP ヘッダ・データを受信するよう指定する方法を示しています。

```
CREATE PROCEDURE SoapClient(
   INOUT hd1 VARCHAR,
   IN hd2 VARCHAR,
   IN hd3 VARCHAR )
 URL 'localhost/some_endpoint'
 SOAPHEADER '!hd1!hd2!hd3';
```

注意

- hd1、hd2、および hd3 はどれも、要求ヘッダ・エントリを指定しています。hd1 はまた、すべての応答ヘッダ・エントリの集合を返します。

- SOAP ヘッダを使用して呼び出された SOAP クライアントは、内包する SOAP ヘッダ要素を生成します。SOAPHEADER 値が NULL の場合、SOAP ヘッダ要素は生成されません。
- SOAP ヘッダが受信されなかった場合、hd1 は NULL に設定されます。
- パラメータのデフォルト・モードは INOUT なので、hd1 の INOUT モード指定は冗長です。
- 複数の代入パラメータが OUT (または INOUT) タイプとして指定されると、ランタイム・エラー '式にサポートされていないデータ型があります。' SQLCODE=-624, ODBC 3 State-"HY000" が発生します。
- SOAPHEADER に対して明示的に使用される代入パラメータ 1 つだけを OUT と宣言できます。

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。つまり、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。

MIME タイプの使用

SQL Anywhere Web サービス・クライアントのプロシージャや関数の定義における TYPE 句では、MIME タイプを指定できます。MIME タイプ指定の値は、Content-Type 要求ヘッダや操作モードの設定に使用することで、1 つの呼び出しパラメータのみで要求の本文を設定することができます。パラメータの置換後に Web サービスのストアド・プロシージャ (または関数) を呼び出す場合は、パラメータがまったくなくなるか、1 つだけ残される場合があります。Web サービスのプロシージャを (置換後に) null パラメータまたはパラメータなしで呼び出すと、本文がなくコンテンツ長が 0 の要求になります。MIME タイプを指定しない場合、動作は変更されません。パラメータの名前と値 (複数のパラメータが可能) は、HTTP 要求の本文内で URL コード化されます。

一般的な MIME タイプの例を次に示します。

- text/plain
- text/html
- text/xml

MIME タイプを設定する手順を次に示します。前半は、MIME タイプの設定をテストするために使用できる Web サービスを設定します。後半は、MIME タイプを設定する方法を示します。

♦ Web サービス・サーバを作成します。

1. データベースを作成します。

   ```
   dbinit echo
   ```

2. このデータベースを使用してサーバを起動します。

   ```
   dbsrv10 -xs http(port=8082) -n echo echo.db
   ```

3. Interactive SQL を使用してサーバに接続します。

   ```
   dbisql -c "UID=DBA;PWD=sql;ENG=echo"
   ```

4. Interactive SQL を使用して Web サービスを作成します。

   ```
   CREATE SERVICE EchoService
   TYPE 'RAW'
   USER DBA
   AUTHORIZATION OFF
   SECURE OFF
   AS CALL Echo(:valueAsXML);
   ```

5. このサービスで呼び出すストアド・プロシージャを定義します。

   ```
   CREATE PROCEDURE Echo( parm LONG VARCHAR )
   BEGIN
      SELECT parm;
   END;
   ```

この時点で、SQL Anywhere Web サービス・サーバは実行されており、要求処理の準備ができています。サーバは HTTP 要求をポート 8082 で受信しています。

この Web サーバをテストに使用するには、別の SQL Anywhere データベースを作成し、起動して、接続します。次の手順は、これを行う方法を示します。

♦ **HTTP 要求を送信するには、次の手順に従います。**

1. データベース作成ユーティリティを使用して、Web サービス・クライアントで使用する別のデータベースを作成します。

```
dbinit echo_client
```

2. Interactive SQL で、次の文を使用してこのデータベースを起動します。

```
START DATABASE 'echo_client.db'
AS echo_client;
```

3. 次の文を使用して、サーバ・エコーで起動したデータベースに接続します。

```
CONNECT TO 'echo'
DATABASE 'echo_client'
USER 'DBA'
IDENTIFIED BY 'sql';
```

4. EchoService Web サービスと通信するストアド・プロシージャを作成します。

```
CREATE PROCEDURE setMIME(
  value LONG VARCHAR,
  mimeType LONG VARCHAR,
  urlSpec LONG VARCHAR
  )
URL '!urlSpec'
HEADER 'ASA-Id'
TYPE 'HTTP:POST:!mimeType';
```

   URL 句は、Web サービスを参照するために使用されます。説明するために、URL は パラメータとして setMIME プロシージャに渡されます。

   TYPE 句は、MIME タイプがパラメータとして setMIME プロシージャに渡されることを示しています。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 Web サービス要求を行うために選択したフォーマットは 'HTTP:POST' です。

5. ストアド・プロシージャを呼び出して、要求を送信し、応答を取得します。渡される value パラメータは、<hello>this is xml</hello> の URL コード化形式です。form-urlencoded は、SQL Anywhere Web サーバによって認識されるため、メディア・タイプは application/x-www-form-urlencoded です。Web サービスの URL は、呼び出しの最後のパラメータとして含まれます。

```
CALL setMIME('valueAsXML=%3Chello%3Ethis%20is%20xml%3C/hello%3E',
   'application/x-www-form-urlencoded',
   'http://localhost:8082/EchoService');
```

   最後のパラメータでは、ポート 8082 で受信している Web サービスの URI を指定します。

Web サーバに送信される HTTP パケットの例は次のようになります。

```
POST /EchoService HTTP/1.0
Date: Sun, 28 Jan 2007 04:04:44 GMT
Host: localhost
```

```
Accept-Charset: windows-1252, UTF-8, *
User-Agent: SQLAnywhere/10.0.1.3349
Content-Type: application/x-www-form-urlencoded; charset=windows-1252
Content-Length: 49
ASA-Id: 1055532613:echo_client:echo:968000
Connection: close

valueAsXML=%3Chello%3Ethis%20is%20xml%3C/hello%3E
```

Web サーバからの応答は次のようになります。

```
HTTP/1.1 200 OK
Server: SQLAnywhere/10.0.1.3349
Date: Sun, 28 Jan 2007 04:04:44 GMT
Expires: Sun, 28 Jan 2007 04:04:44 GMT
Content-Type: text/plain; charset=windows-1252
Connection: close

<hello>this is xml</hello>
```

Interactive SQL で表示される結果セットは次のようになります。

属性	値
Status	HTTP /1.1 200 OK
Body	<hello>this is xml</hello>
Server	SQLAnywhere/10.0.1.3349
Date	Sun, 28 Jan 2007 04:04:44 GMT
Expires	Sun, 28 Jan 2007 04:04:44 GMT
Content-Type	text/plain; charset=windows-1252
Connection	close

HTTP セッションの使用

HTTP 接続は、HTTP 要求間のステータスを管理する HTTP セッションを作成できます。

「**HTTP セッション**」は、最低限の SQL アプリケーション・コードを使用してクライアント (Web ブラウザなど) のステータスを保持する手段を提供します。セッション・コンテキストにおけるデータベース接続は、セッションの存続期間の間、保持されます。セッション ID でマーク付けされた新しい HTTP 要求は直列化 (キューに追加) され、セッション ID が同じ要求は同じデータベース接続を使用して順番に処理されます。データベース接続の再利用は、HTTP 要求間でステータス情報を保持する手段になります。一方、セッションレス HTTP 要求は、要求ごとに新しいデータベース接続を作成するので、テンポラリ・テーブルのデータや接続変数を要求間で共有することができません。

HTTP セッション管理により、URL と cookie の両方のステータス管理方法がサポートされます。

HTTP セッション機能の実例は、*samples-dir¥SQLAnywhere¥HTTP¥session.sql* に用意されています。

HTTP セッションの作成

セッションは、Web アプリケーション内で sa_set_http_option システム・プロシージャを呼び出し、HTTP オプション SessionID を使用して作成します。セッション ID になり得るのは、null 以外の任意の文字列です。内部的には、セッション・キーはセッション ID とデータベース名という構成で生成されるので、複数のデータベースがロードされた場合でも、セッション・キーはデータベース間でユニークになります。セッション・キー全体の長さの上限は 128 文字です。次の例では、ユニークなセッション ID が生成され、sa_set_http_option に渡されています。

```
DECLARE session_id VARCHAR(64);
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);
```

Web アプリケーションは、SessionID 接続プロパティを使用してセッション ID を取得できます。接続に対してセッション ID が定義されていない場合 (接続がセッションレスの場合)、このプロパティは空の文字列になります。

```
DECLARE session_id VARCHAR(64);
SELECT CONNECTION_PROPERTY( 'SessionID' ) INTO session_id;
```

sa_set_http_option プロシージャを使用してセッションが作成されると、localhost 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 など) 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
```

接続されているデータベースが 1 つだけの場合は、データベース名を省略できます。

```
http://localhost/session_service?sessionid=session_63315422814117
```

cookie を使用したセッション管理

cookie のステータス管理は、sa_set_http_header システム・プロシージャにフィールド名として 'Set-Cookie' を指定することでサポートされます。ステータス管理に cookie を使用することで、URL にセッション ID を含める必要がなくなります。代わりに、クライアントは HTTP cookie ヘッダでセッション ID を提供します。ステータス管理に cookie を使用することの欠点は、クライアントが cookie を無効にする可能性がある統制のない環境では、cookie がサポートされているかどうかが確実ではないことです。したがって、Web アプリケーションは URL および cookie によるセッション・ステータス管理をどちらもサポートする必要があります。前項で説明した URL セッション ID は、クライアントが URL および cookie セッション ID をどちらも提供した場合に優先的に使用されます。Web アプリケーションは、セッションの有効期限が切れたり、セッションが明示的に削除されたりしたときに (sa_set_http_option('SessionID', NULL) など)、SessionID cookie を削除する必要があります。

```
DECLARE session_id VARCHAR(64);
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;' );
```

古いセッションの削除

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するには、SessionID および SessionCreateTime 接続プロパティが便利です。いずれかの接続プロパティに対するクエリで空の文字列が返された場合、そのセッションは存在していません。SessionCreateTime 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、sa_set_http_option が呼び出された時点でただちに定義されます。SessionLastTime プロパティは、セッションが最後に使用された時刻を提供します。具体的には、そのセッションで最後に処理された要求が、その要求の終了時にデータベース接続を解放した時刻です。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、SessionLastTime 接続プロパティには空の文字列が返されます。

セッション ID の削除または変更

セッション ID は、新しい SessionID 値を指定した sa_set_http_option システム・プロシージャを呼び出すことで、別の値に設定し直すことができます。セッション ID の変更は、古いセッションを削除して新しいセッションを作成することに相当しますが、現在のデータベース接続を再利用するので、ステータス情報は失われません。SessionID に NULL (または空の文字列) を設定すると、そのセッションが削除されます。SessionID に既存するセッションの ID を設定することはできません (セッション自身のセッション ID は設定できますが、その場合は何も起こりません)。既存するセッションのセッション ID を SessionID として設定しようとすると、エラー「HTTP オプション 'SessionID' の設定が無効です。SQLCODE=-939」が発生します。

サーバは、同じセッション・コンテキストを指定する複数の HTTP 要求を集中的に受信すると、そうした要求をセッション・キューに追加 (直列化) します。SessionID が 1 つ (または複数の) 要求によって変更または削除された場合、セッション・キューにある保留中の要求はすべて、独立した HTTP 要求としてキューに再度追加されます。各 HTTP 要求は、該当セッションが存在しないので、セッションを取得できません。セッションを取得できなかった HTTP 要求は、デフォルトでセッションレス動作になり、新しいデータベース接続を作成します。Web アプリケーションは、SessionID または SessionCreateTime 接続プロパティの文字列値が空でないかどうかを確認することで、要求がセッション・コンテキスト内で動作しているかどうかを確認できます。当然のことながら、Web アプリケーションは、使用しているアプリケーション固有の変数やテンポラリ・テーブルのステータスを確認できます。次に例を示します。

```
IF VAREXISTS( 'state' ) = 0 THEN
   // first invocation by this connection
   CREATE VARIABLE state LONG VARCHAR;
END IF;
```

セッション・セマンティック

HTTP 要求は、HTTP セッション・コンテキストを作成できます。要求によって作成されたセッションは必ず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。

セッションなしで開始し、セッション・コンテキストを作成した HTTP 要求が、そのセッションの作成者になります。作成者の要求は、セッション・コンテキストを変更または削除でき、変更または削除はただちに実行されます。セッション・コンテキスト内で開始された HTTP 要求も、自身のセッションを変更または削除できます。自身のセッションを変更すると、完全に動作可能な保留中のセッションがただちに作成されます。ただし、他の HTTP 要求は所有権を取得できません (保留中のセッションを必要とする要求は、受信されるとそのセッションのキューに追加されます)。要約すると、作成者要求のセッションを変更または削除すると、現在のセッション・コンテキストがただちに変更されますが、自身のセッションを変更するだけの要求は、自身の保留中のセッションを変更します。HTTP 要求は、完了すると、保留中のセッションがあるかどうかを確認します。保留中のセッションがある場合は、現在のセッションを削除し、保留中のセッションに置き換えます。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。

どのような場合にも、HTTP セッションが削除されると、キュー内にあった要求はすべて解放され、セッション・コンテキストなしで実行可能になります。要求がセッション・コンテキスト内で実行されることを想定しているアプリケーション・コードは、CONNECTION_PROPERTY ('SessionID') を呼び出して、有効なセッション・コンテキストを取得する必要があります。

```
DECLARE ses_id LONG VARCHAR;
SELECT CONNECTION_PROPERTY( 'SessionID' ) INTO ses_id;
```

HTTP 要求が、故意またはネットワーク障害によって取り消された場合、既存の保留中のセッションは削除され、元のセッション・コンテキストが保持されます。作成者の HTTP 要求は、取り消されても通常に終了された場合でも、セッションのステータスをただちに変更します。

接続の切断とサーバのシャットダウン

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 HTTP 要求はすべてキャンセル・ステータスになります。これにより、HTTP 要求は迅速に終了され、ユーザにそのことが通知されます。

同様に、サーバまたはデータベースをシャットダウンすることで適切なデータベース接続がキャンセルされると、HTTP 要求がキャンセルされる可能性があります。

セッション・タイムアウト

http_session_timeout パブリック・データベース・オプションを使用して、さまざまなセッション・タイムアウトを制御できます。このオプションは分単位で指定します。デフォルトのパブリック設定は 30 分です。最小値は 1 分で、最大値は 525600 分 (365 日) です。Web アプリケーションは、セッションを所有する任意の要求を使用してタイムアウト基準を変更できます。新しいタイムアウト基準は、そのセッションがタイムアウトした場合に、キューに追加される後続の要求に影響を与える可能性があります。存在しないセッションにクライアントがアクセスしようとしていることを検出する論理の提供は、Web アプリケーションで行う必要があります。検出するには、SessionCreateTime 接続プロパティに有効なタイムスタンプがあるかどうかを判断します。HTTP 要求が既存のセッションに関連付けられていない場合、SessionCreateTime 値は空の文字列になります。

セッションのスコープ

セッションは、サーバの再起動後は存続しません。

ライセンス

セッションに関連付けられている接続は、接続の存続期間中はデータベース接続を保持し続けるので、ライセンス・シートも 1 つ保持し続けます。このことを考慮して、Web アプリケーションでは、古いセッションを適切に削除するか、適切なタイムアウト値を設定する必要があります。

SQL Anywhere のライセンス取得の詳細については、http://www.ianywhere.com/products/sa_licensing.html を参照してください。

セッション・エラー

エラー 503 Service Unavailable は、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に発生します。

エラー 403 Forbidden は、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に発生します。

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。

セッションの接続プロパティおよびオプションの一覧

接続プロパティ

- **SessionID**

  ```
  SELECT CONNECTION_PROPERTY('SessionID') INTO ses_id;
  ```

  現在のデータベース・コンテキストでの現在のセッション ID を提供します。

- **SessionCreateTime**

  ```
  SELECT CONNECTION_PROPERTY('SessionCreateTime') INTO ses_create;
  ```

  セッションが作成された時刻を示すタイムスタンプを提供します。

- **SessionLastTime**

  ```
  SELECT CONNECTION_PROPERTY('SessionLastTime') INTO ses_last;
  ```

  セッションが最後の要求により解放された時刻を示すタイムスタンプを提供します。

- **http_session_timeout**

  ```
  SELECT CONNECTION_PROPERTY('http_session_timeout') INTO ses_timeout;
  ```

  現在のセッション・タイムアウトを分単位でフェッチします。

HTTP オプション

- **'SessionID','value'**

  ```
  CALL sa_set_http_option( 'SessionID', 'my_app_session_1' )
  ```

  現在の HTTP 要求のセッション・コンテキストを作成または変更します。my_app_session_1 が別の HTTP 要求に所有されている場合は、エラーを返します。

- **'SessionID', NULL**

  ```
  CALL sa_set_http_option('SessionID', NULL )
  ```

  要求がセッション作成者から送信された場合は、現在のセッションはただちに削除されます。それ以外の場合は、セッションは削除対象としてマーク付けされます。要求にセッションがない場合にセッションを削除することは、エラーではなく、影響は何もありません。

  セッションを現在のセッションの SessionID に変更 (保留中のセッションなし) - エラーではない。何も起こらない。

  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 - エラー。

変更がすでに保留中のときにセッションを変更 - 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る。

保留中のセッションがあるセッションを元の SessionID に戻す - 保留中のセッションが削除される。

HTTP セッション・タイムアウト

- **http_session_timeout**

  ```
  SET TEMPORARY OPTION PUBLIC.http_session_timeout=100;
  ```

  現在の HTTP セッション・タイムアウトを分単位で設定します。デフォルトは 30 分で、範囲は 1 ー 525600 分 (365 日) です。

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

管理

Web アプリケーションでは、データベース・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 コードは、そのための方法を示しています。

```
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';
```

サーバ・メッセージ・ウィンドウには、次のような出力が表示されます。

```
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、「セッション ID の削除または変更」 737 ページで説明したように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。

データベースを無条件に停止すると、各データベース接続が切断され、そのデータベース・コンテキスト内のすべてのセッション (「HTTP セッションの作成」 736 ページのセッション・キーの説明を参照) が削除されます。この動作が保証されるのは、セッション・コンテキスト 1 つにつき有効なデータベース接続が 1 つ必要であり、データベース接続は一度に 1 つのセッションにしか関連付けることができないからです。セッションとデータベース接続は、どちらも同じデータベース・コンテキスト内に存在する必要があります。

自動文字セット変換の使用

デフォルトで、文字セット変換はテキスト・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では変換されません。要求の文字セットはデータベースの文字セットに変換され、必要に応じて結果セットはデータベースの文字セットからクライアントの文字セットに変換されます。結果セットのバイナリ・カラムは変換されません。要求に処理可能な文字セットが複数リストされている場合、サーバがリストの中から最初に検出した最適なものを使用します。

文字セット変換は、HTTP オプションの CharsetConversion を設定することで有効または無効にできます。使用できる値は ON と OFF です。デフォルト値は ON です。次の文は、文字セットの変換をオフにします。

```
CALL sa_set_http_option( 'CharsetConversion', 'OFF' )
```

組み込みのストアド・プロシージャの詳細については、「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。

エラー処理

Web サービス要求に失敗した場合、データベース・サーバが標準エラーを生成してブラウザに表示します。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。

サービスが SOAP サービスである場合、SOAP バージョン 1.1 標準で定義されているように、フォールトは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。

発生する可能性のある一般的なエラーは次のとおりです。

番号	名前	SOAP フォールト	説明
301	Moved permanently	Server	要求されたページは永続的に移動されました。サーバは、自動的に新しいロケーションに要求をリダイレクトします。
304	Not Modified	Server	サーバは、要求の情報に基づき、要求されたデータは前回の要求の後変更されていないため、再度送信する必要はないと判断しました。
307	Temporary Redirect	Server	要求されたページは移動されましたが、この変更は永続的なものではない可能性があります。サーバは、自動的に新しいロケーションに要求をリダイレクトします。
400	Bad Request	Client.BadRequest	HTTP 要求が正しくないか不正です。
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。

番号	名前	SOAP フォールト	説明
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。

パート IV. SQL Anywhere での ADO と Visual Basic の使用

パート IV では、Visual Basic を使用した ADO インタフェースの簡単な例を提供します。

第 18 章

チュートリアル：Visual Basic での簡易アプリケーションの開発

目次

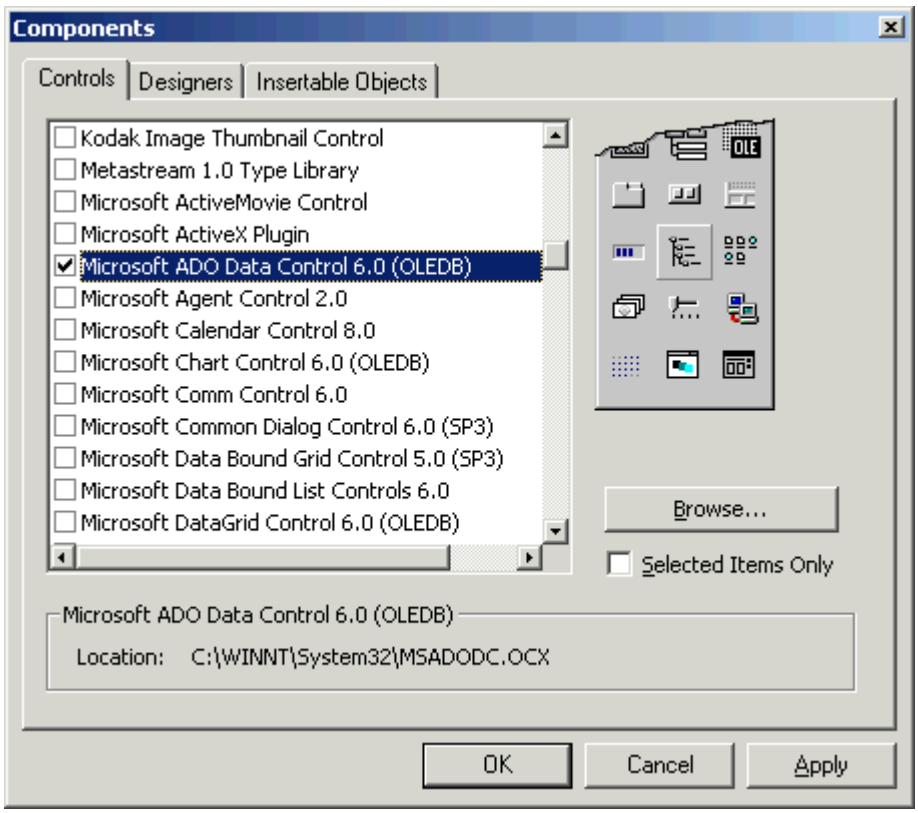
Visual Basic チュートリアルの概要

このチュートリアルは、Visual Basic 6.0 を基にしています。完全なアプリケーションは、Visual Basic プロジェクトの *samples-dir¥SQLAnywhere¥VBStarter¥Starter.vbp* で参照できます。

Visual Basic は、複数のデータ・アクセス技術を提供しています。このチュートリアルでは、Microsoft ADO Data Control を SQL Anywhere の OLE DB プロバイダと一緒に使用して、Visual Basic から SQL Anywhere のサンプル・データベースにアクセスします。

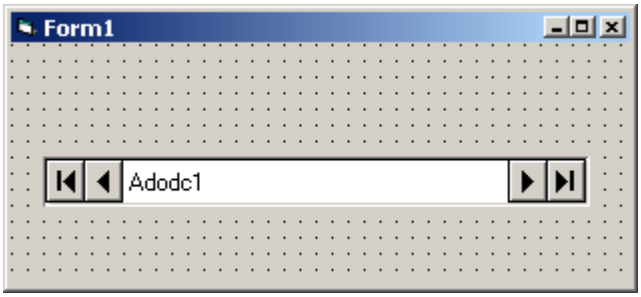
♦ **Visual Basic でデータベース・アプリケーションを開発するには、次の手順に従います。**

1. Visual Basic を起動して、[標準 EXE] プロジェクトを選択します。
2. Microsoft ADO Data Control 6.0 をツール・パレットに追加します。
   - ♦ [プロジェクト] メニューから、[コンポーネント] を選択します。
   - ♦ リストから、[Microsoft ADO Data Control 6.0] コンポーネントを選択します。



   - ♦ [OK] をクリックして、コントロールをツールボックス・パレットに追加します。
3. 次のように、ADO Data Control をフォームに追加します。

- [表示] メニューから、[ツールボックス] を選択します。
- ツールボックス・パレットで Adodc アイコンをクリックします。
- 設計フォームに長方形を描きます。

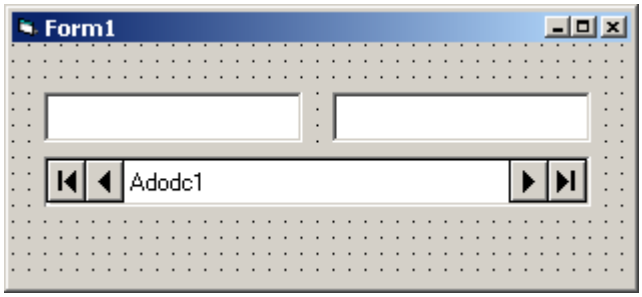


4. ADO Data Control の設定

プロパティ	値
CommandType	2 - adCmdTable
ConnectionString	Provider=SAOLEDB;DSN=SQL Anywhere 10 Demo
CursorLocation	2 - adUseServer
CursorType	2 - adOpenDynamic
RecordSource	Employees

ConnectionString は SQL Anywhere OLE DB プロバイダ (SAOLEDB) を使用して "SQL Anywhere 10 Demo" データ・ソースに接続します。カーソルの設定では、クライアント側のカーソルではなく、SQL Anywhere のサーバ側のカーソルを利用します。

5. 次のように、2 つのテキスト・ボックスをフォームに追加します。
   - ツールボックス・パレットで TextBox アイコンをクリックします。
   - 下の図のように、設計フォーム内で Adodc1 コントロールのすぐ上に長方形を描きます。
   - 下の図のように、設計フォーム内で最初の TextBox の右側に 2 番目の長方形を描きます。



6. テキスト・ボックスを ADO Data Control にバインドします。

   - それぞれの DataSource プロパティを「**Adodc1**」に設定します。
   - 左側のテキスト・ボックスの DataField プロパティを **GivenName** に設定します。これは、従業員の名を格納しているカラムです。
   - 右側のテキスト・ボックスの DataField プロパティを **Surname** に設定します。これは、従業員の姓を格納しているカラムです。

7. プロジェクトを保存します。

8. サンプルを実行します。

   - [実行] – [開始] を選択してアプリケーションを実行します。

     アプリケーションが SQL Anywhere サンプル・データベースに接続され、次のように最初の従業員の名前がテキスト・ボックスに表示されます。

   - ADO Data Control のボタンを使用して、結果セットのローをスクロールします。

これで ADO と SQL Anywhere OLEDB プロバイダで動作する Visual Basic の簡単なアプリケーションが作成されました。

パート V. SQL Anywhere データベース・ツール・インタフェース

パート V では、SQL Anywhere のデータベース・ツール・プログラミング・インタフェースについて説明します。

第 19 章

データベース・ツール・インタフェース

目次

データベース・ツール・インタフェースの概要

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。このライブラリは、Windows オペレーティング・システムと UNIX 向けに提供されています。ライブラリの名前は、Windows の場合は *dbtool10.dll* で、UNIX の場合は *libdbtool10.so* (非スレッド) または *libdbtool10_r.so* (スレッド) 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。この章では、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。この章の説明は、使用中の開発環境から DLL 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 CE

Windows CE には *dbtool10.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。その他のツールは Windows CE では使用できません。

dbtools.h ヘッダ・ファイル

SQL Anywhere に含まれる dbtools ヘッダ・ファイルは、DBTools ライブラリへのエントリ・ポイントと、DBTools ライブラリとの間で情報をやりとりするために使用する構造体をリストします。*dbtools.h* ファイルはすべて、SQL Anywhere インストール・ディレクトリの *h* サブディレクトリにインストールされています。エントリ・ポイントと構造体メンバ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・ファイルには他の 3 つの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・ファイルはエラー戻り値も含みます。

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* ファイルはすべて、SQL Anywhere インストール・ディレクトリの *h* サブディレクトリにインストールされています。DBRemoteSQL エントリ・ポイントと構造体メンバ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。

データベース・ツール・インタフェースの使い方

この項では、データベースの管理に DBTools インタフェースを使用するアプリケーションの開発方法の概要について説明します。

インポート・ライブラリの使い方

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。

サポートするプラットフォーム

インポート・ライブラリはコンパイラ固有であり、Windows CE 以外の Windows オペレーティング・システム用に用意されています。DBTools インタフェース用のインポート・ライブラリは SQL Anywhere に付属しており、SQL Anywhere インストール・ディレクトリの下の、各オペレーティング・システム・ディレクトリの *lib* サブディレクトリに収められています。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
Borland	*lib¥dbtlstb.lib*
Microsoft	*lib¥dbtlstm.lib*
Microsoft (Windows CE)	*lib¥dbtools10.lib*
Watcom	*lib¥dbtlstw.lib*

DBTools ライブラリの開始と終了

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 DLL 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 DLL が SQL Anywhere 言語 DLL をロードできるようにすることです。言語 DLL 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を含んでいます。DBToolsFini を呼び出さないと、言語 DLL 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/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 を初期化してクリーンアップ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 DBTools
```

```
ret = DBToolsInit( &info );
if( ret != EXIT_OKAY ) {
    // DLL initialization failed
    ...
}
// call some DBTools routines . . .
...
// cleanup the DBTools dll
DBToolsFini( &info );
```

DBTools 関数の呼び出し

すべてのツールは、まず構造体に値を設定し、次に DBTools DLL 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 );
```

DBTools 構造体のメンバについては、「DBTools 構造体」 775 ページを参照してください。

コールバック関数の使い方

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 DLL 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領域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・メッセージをメッセージ・ボックスに表示します。

```
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があります。このバージョン・メンバに、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を格納しておきます。DBTools ライブラリの現在のバージョンは、*dbtools.h* ヘッダ・ファイルにシンボリック定数として設定されています。

♦ 現在のバージョン番号を構造体に割り当てるには、次の手順に従ってください。

1. バージョン定数を構造体のバージョン・メンバに割り当ててから、DBTools 関数を呼び出します。次の行は、現在のバージョンをバックアップ構造体に割り当てています。

   ```
   backup_info.version = DB_TOOLS_VERSION_NUMBER;
   ```

互換性

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 DLL を配備してください。アプリケーションの機能が変更されない場合は、DLL バージョンの不一致が起こらないように *dbtlsvers.h* で定義されたバージョン固有のマクロを使用してください。

ビット・フィールドの使い方

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、*samples-dir¥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 10 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( "dbtool10.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;
}
```

DBTools 関数

DBBackup 関数

データベースをバックアップします。この関数は、dbbackup ユーティリティによって使用されます。

プロトタイプ

short **DBBackup (** const a_backup_db * **);**

パラメータ

構造体へのポインタ。「a_backup_db 構造体」 775 ページを参照してください。

戻り値

「ソフトウェア・コンポーネントの終了コード」 820 ページにリストされているリターン・コード

備考

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。「BACKUP 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

DBChangeLogName 関数

トランザクション・ログ・ファイルの名前を変更します。この関数は、dblog ユーティリティによって使用されます。

プロトタイプ

short **DBChangeLogName (** const a_change_log * **);**

パラメータ

構造体へのポインタ。「a_change_log 構造体」 777 ページを参照してください。

戻り値

「ソフトウェア・コンポーネントの終了コード」 820 ページにリストされているリターン・コード

備考

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。

dblog ユーティリティの説明については、「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』を参照してください。

参照

- 「ALTER DATABASE 文」『SQL Anywhere サーバ - SQL リファレンス』

DBCreate 関数

データベースを作成します。この関数は、dbinit ユーティリティによって使用されます。

プロトタイプ

short **DBCreate (** const a_create_db * **);**

パラメータ

構造体へのポインタ。「a_create_db 構造体」 779 ページを参照してください。

戻り値

「ソフトウェア・コンポーネントの終了コード」 820 ページにリストされているリターン・コード

備考

dbinit ユーティリティの詳細については、「初期化ユーティリティ (dbinit)」『SQL Anywhere サーバ - データベース管理』を参照してください。

参照

- 「CREATE DATABASE 文」『SQL Anywhere サーバ - SQL リファレンス』

DBCreatedVersion 関数

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。現在、この関数はバージョン 10 と 10 以前のデータベースを区別するだけです。

プロトタイプ

short **DBCreatedVersion (** a_db_version_info * **);**

パラメータ

構造体へのポインタ。「a_db_version_info 構造体」 784 ページを参照してください。

戻り値

「ソフトウェア・コンポーネントの終了コード」 820 ページにリストされているリターン・コード

備考

成功したことを示すリターン・コードが返された場合、a_db_version_info 構造体の created_version フィールドには、データベースを作成した SQL Anywhere のバージョンを示す a_db_version の値が含まれます。可能な値の定義については、「a_db_version 列挙」 815 ページを参照してください。

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

- 「CREATE DATABASE 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「a_db_version_info 構造体」 784 ページ
- 「a_db_version 列挙」 815 ページ

DBErase 関数

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。この関数は、dberase ユーティリティによって使用されます。

プロトタイプ

short **DBErase (** const an_erase_db * **);**

パラメータ

構造体へのポインタ。「an_erase_db 構造体」 786 ページを参照してください。

戻り値

「ソフトウェア・コンポーネントの終了コード」 820 ページにリストされているリターン・コード

備考

消去ユーティリティとその機能の詳細については、「消去ユーティリティ (dberase)」 『SQL Anywhere サーバ - データベース管理』を参照してください。

DBInfo 関数

データベース・ファイルに関する情報を戻します。この関数は、dbinfo ユーティリティによって使用されます。

プロトタイプ

short **DBInfo (** const a_db_info * **);**

パラメータ

構造体へのポインタ。「a_db_info 構造体」 781 ページを参照してください。

戻り値

「ソフトウェア・コンポーネントの終了コード」 820 ページにリストされているリターン・コード

備考

情報ユーティリティとその機能の詳細については、「情報ユーティリティ (dbinfo)」 『SQL Anywhere サーバ - データベース管理』を参照してください。

参照

- 「DBInfoDump 関数」 768 ページ
- 「DBInfoFree 関数」 768 ページ
- 「DB_PROPERTY 関数 [システム]」 『SQL Anywhere サーバ - SQL リファレンス』

DBInfoDump 関数

データベース・ファイルに関する情報を戻します。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。

プロトタイプ

short **DBInfoDump (** const a_db_info * **);**

パラメータ

構造体へのポインタ。「a_db_info 構造体」 781 ページを参照してください。

戻り値

「ソフトウェア・コンポーネントの終了コード」 820 ページにリストされているリターン・コード

備考

情報ユーティリティとその機能の詳細については、「情報ユーティリティ (dbinfo)」 『SQL Anywhere サーバ - データベース管理』を参照してください。

参照

- 「DBInfo 関数」 767 ページ
- 「DBInfoFree 関数」 768 ページ
- 「sa_table_page_usage システム・プロシージャ」 『SQL Anywhere サーバ - SQL リファレンス』

DBInfoFree 関数

DBInfoDump 関数の呼び出し後に、リソースを解放します。

プロトタイプ

short **DBInfoFree (** const a_db_info * **);**

パラメータ

構造体へのポインタ。「a_db_info 構造体」 781 ページを参照してください。

戻り値

「ソフトウェア・コンポーネントの終了コード」 820 ページにリストされているリターン・コード

備考

情報ユーティリティとその機能の詳細については、「情報ユーティリティ (dbinfo)」『SQL Anywhere サーバ - データベース管理』を参照してください。

参照

- 「DBInfo 関数」 767 ページ
- 「DBInfoDump 関数」 768 ページ

DBLicense 関数

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。

プロトタイプ

short **DBLicense (** const a_db_lic_info * **);**

パラメータ

構造体へのポインタ。「a_dblic_info 構造体」 784 ページを参照してください。

戻り値

「ソフトウェア・コンポーネントの終了コード」 820 ページにリストされているリターン・コード

備考

サーバ・ライセンス取得ユーティリティとその機能の詳細については、「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』を参照してください。

DBRemoteSQL 関数

SQLRemote Message Agent にアクセスします。

プロトタイプ

short **DBRemoteSQL(** const a _remote_sql * **);**

パラメータ

構造体へのポインタ。「a_remote_sql 構造体」 787 ページを参照してください。

戻り値

「ソフトウェア・コンポーネントの終了コード」 820 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、「Message Agent」 『SQL Remote』を参照してください。

参照

- 「SQLRemote の概念」 『SQL Remote』

DBSynchronizeLog 関数

データベースを Mobile Link サーバと同期させます。

プロトタイプ

short **DBSynchronizeLog(** const a _sync_db * **);**

パラメータ

構造体へのポインタ。「a_sync_db 構造体」 793 ページを参照してください。

戻り値

「ソフトウェア・コンポーネントの終了コード」 820 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、「同期の開始」 『Mobile Link - クライアント管理』を参照してください。

参照

- 「dbmlsync の DBTools インタフェース」 『Mobile Link - クライアント管理』

DBToolsFini 関数

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、カウンタを減分して、リソースを解放します。

プロトタイプ

short **DBToolsFini (** const a_dbtools_info * **);**

パラメータ

構造体へのポインタ。「a_dbtools_info 構造体」 785 ページを参照してください。

戻り値

「ソフトウェア・コンポーネントの終了コード」 820 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

- 「DBToolsInit 関数」 771 ページ

DBToolsInit 関数

DBTools ライブラリを使用できるよう準備します。

プロトタイプ

short **DBToolsInit(** const a_dbtools_info * **);**

パラメータ

構造体へのポインタ。「a_dbtools_info 構造体」 785 ページを参照してください。

戻り値

「ソフトウェア・コンポーネントの終了コード」 820 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 言語 DLL をロードすることです。言語 DLL 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。例については、「DBTools の例」 762 ページを参照してください。

参照

- 「DBToolsFini 関数」 770 ページ

DBToolsVersion 関数

DBTools ライブラリのバージョン番号を戻します。

プロトタイプ

short **DBToolsVersion (** void **);**

戻り値

DBTools ライブラリのバージョン番号を示す short integer

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」 761 ページ

DBTranslateLog 関数

トランザクション・ログ・ファイルを SQL に変換します。この関数は、dbtran ユーティリティによって使用されます。

プロトタイプ

short **DBTranslateLog (** const a_translate_log * **);**

パラメータ

構造体へのポインタ。「a_translate_log 構造体」 802 ページを参照してください。

戻り値

「ソフトウェア・コンポーネントの終了コード」 820 ページにリストされているリターン・コード

備考

ログ変換ユーティリティの詳細については、「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』を参照してください。

DBTruncateLog 関数

トランザクション・ログ・ファイルをトランケートします。この関数は、dbbackup ユーティリティによって使用されます。

プロトタイプ

short **DBTruncateLog (** const a_truncate_log * **);**

パラメータ

構造体へのポインタ。「a_truncate_log 構造体」 806 ページを参照してください。

戻り値

「ソフトウェア・コンポーネントの終了コード」 820 ページにリストされているリターン・コード

備考

バックアップ・ユーティリティの詳細については、「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。

参照

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

DBUnload 関数

データベースをアンロードします。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。

プロトタイプ

short **DBUnload (** const an_unload_db * **);**

パラメータ

構造体へのポインタ。「an_unload_db 構造体」 807 ページを参照してください。

戻り値

「ソフトウェア・コンポーネントの終了コード」 820 ページにリストされているリターン・コード

備考

アンロード・ユーティリティの詳細については、「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』を参照してください。

抽出ユーティリティの詳細については、「抽出ユーティリティ」『SQL Remote』を参照してください。

DBUpgrade 関数

データベース・ファイルをアップグレードします。この関数は、dbupgrad ユーティリティによって使用されます。

プロトタイプ

short **DBUpgrade (** const an_upgrade_db * **);**

パラメータ

構造体へのポインタ。「an_upgrade_db 構造体」 811 ページを参照してください。

戻り値

「ソフトウェア・コンポーネントの終了コード」 820 ページにリストされているリターン・コード

備考

アップグレード・ユーティリティの詳細については、「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』を参照してください。

参照

- 「ALTER DATABASE 文」『SQL Anywhere サーバ - SQL リファレンス』

DBValidate 関数

データベースの全部または一部を検証します。この関数は、dbvalid ユーティリティによって使用されます。

プロトタイプ

short **DBValidate (** const a_validate_db * **);**

パラメータ

構造体へのポインタ。「a_validate_db 構造体」 812 ページを参照してください。

戻り値

「ソフトウェア・コンポーネントの終了コード」 820 ページにリストされているリターン・コード

備考

検証ユーティリティの詳細については、「検証ユーティリティ (dbvalid)」『SQL Anywhere サーバ - データベース管理』を参照してください。

> **警告**
> テーブルまたはデータベース全体の検証は、どの接続においてもデータベースを変更していない場合に実行してください。そうでない場合、実際に破損がなくても、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

- 「VALIDATE 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「sa_validate システム・プロシージャ」『SQL Anywhere サーバ - SQL リファレンス』

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) と呼ばれるメンバをもつ構造体があります。このメンバは、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。

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 {
    unsigned short      version;
    const char *        output_dir;
    const char *        connectparms;
    const char *        _unused0;
    MSG_CALLBACK        confirmrtn;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        statusrtn;
    a_bit_field         backup_database : 1;
    a_bit_field         backup_logfile  : 1;
    a_bit_field         _unused1        : 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;
    const char *        hotlog_filename;
    char                backup_interrupted;
    a_bit_field         server_backup   : 1;
    a_chkpt_log_type    chkpt_log_type;
    a_sql_uint32        page_blocksize;
} a_backup_db;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号
output_dir	出力ディレクトリのパス。次に例を示します。 "c:¥backup"

メンバ	説明
connectparms	データベース接続に必要なパラメータ。次のような接続文字列のフォームをとる。 "UID=DBA;PWD=sql;DBF=samples-dir¥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。次に例を示します。 "START=d:¥sqlany10¥win32¥dbeng10.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=samples-dir¥demo.db;START=d:¥sqlany10¥win32¥dbeng10.exe" 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。
confirmrtn	動作確認コールバック・ルーチン
errorrtn	エラー・メッセージ処理コールバック・ルーチン
msgrtn	情報メッセージ処理コールバック・ルーチン
statusrtn	ステータス・メッセージ処理コールバック・ルーチン
backup_database	データベース・ファイルをバックアップする (1) またはしない (0)
backup_logfile	トランザクション・ログ・ファイルをバックアップする (1) またはしない (0)
_unused1	(使われていない)
no_confirm	操作の確認をする (0) またはしない (1)
quiet	操作中にメッセージを出力する (0) またはしない (1)
rename_log	トランザクション・ログの名前変更
truncate_log	トランザクション・ログの削除
rename_local_log	トランザクション・ログのローカル・バックアップの名前変更
hotlog_filename	ライブ・バックアップ・ファイルのファイル名
backup_interrupted	オペレーションが中断されたことを示す。
server_backup	1 に設定すると、バックアップ・データベースを使用したサーバでのバックアップを示す。dbbackup の -s オプションと同等。

メンバ	説明
chkpt_log_type	チェックポイント・ログのコピーを制御する。 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。
page_blocksize	データ・ブロック内のページ数。dbbackup の -b オプションと同等。0 に設定すると、デフォルトは 128 になります。

参照

- 「DBBackup 関数」 765 ページ
- 「a_db_version 列挙」 815 ページ
- 「コールバック関数の使い方」 759 ページ.

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 {
    unsigned short      version;
    const char *        dbname;
    const char *        logname;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;


    a_bit_field         query_only              : 1;
    a_bit_field         quiet                   : 1;
    a_bit_field         _unused1                : 1;
    a_bit_field         change_mirrorname       : 1;
    a_bit_field         change_logname          : 1;
    a_bit_field         ignore_ltm_trunc        : 1;
    a_bit_field         ignore_remote_trunc     : 1;
    a_bit_field         set_generation_number   : 1;
    a_bit_field         ignore_dbsync_trunc     : 1;


    const char *        mirrorname;
    unsigned short      generation_number;
    const char *        _unused2;
    char *              zap_current_offset;
    char *              zap_starting_offset;
    char *              encryption_key;
} a_change_log;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号

メンバ	説明
dbname	データベース・ファイル名
logname	トランザクション・ログの名前。NULL を設定すると、ログは取られない。
errorrtn	エラー・メッセージ処理コールバック・ルーチン
msgrtn	情報メッセージ処理コールバック・ルーチン
query_only	1 の場合、トランザクション・ログの名前は表示のみ。0 の場合、ログ名を変更可能。
quiet	操作中にメッセージを出力する (0) またはしない (1)
change_mirrorname	1 の場合、ログ・ミラー名を変更可能
change_logname	1 の場合、トランザクション・ログ名を変更可能
ignore_ltm_trunc	Log Transfer Manager を使用している場合、dbcc settrunc('ltm', 'gen_id', n) Replication Server 関数と同じ関数を実行する。 dbcc については、Replication Server マニュアルを参照してください。
ignore_remote_trunc	SQL Remote 用。delete_old_logs オプションのための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する。
set_generation_number	Log Transfer Manager を使用している場合、バックアップをリストアして世代番号を設定した後に使用される。
ignore_dbsync_trunc	dbmlsync を使用している場合、delete_old_logs オプションのための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する。
mirrorname	トランザクション・ログ・ミラー・ファイルの新しい名前
generation_number	新しい世代番号。set_generation_number とともに使用される。
zap_current_offset	現在のオフセットを指定の値に変更する。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する。
zap_starting_offset	開始オフセットを指定の値に変更する。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する。
encryption_key	データベース・ファイルの暗号化キー

参照

- 「DBChangeLogName 関数」 765 ページ
- 「コールバック関数の使い方」 759 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 {
    unsigned short      version;
    const char          *dbname;
    const char          *logname;
    const char          *startline;
    unsigned short      page_size;
    const char          *default_collation;
    const char          *encoding;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    short               _unused1;
    char                verbose;


    a_bit_field         blank_pad               : 2;
    a_bit_field         respect_case            : 1;
    a_bit_field         encrypt                 : 1;
    a_bit_field         _unused2                : 1;
    a_bit_field         _unused3                : 1;
    a_bit_field         _unused4                : 1;
    a_bit_field         avoid_view_collisions   : 1;
    short               _unused5;
    const char          *_unused6;
    const char          *_mirrorname;
    const char          *_unused7;
    a_bit_field         _unused8                : 1;
    a_bit_field         jconnect                : 1;
    const char          *data_store_type;
    const char          *encryption_key;
    const char          *encryption_algorithm;
    const char          *_unused9;
    a_bit_field         _unused10               : 1;
    a_bit_field         checksum                : 1;
    a_bit_field         encrypted_tables        : 1;
    char                accent_sensitivity;
    const char          *nchar_collation;
    char                *dba_uid;
    char                *dba_pwd;
    unsigned int        db_size;
    int                 db_size_unit;
} a_create_db;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号

メンバ	説明
dbname	データベース・ファイル名
logname	新しいトランザクション・ログ名
startline	データベース・サーバを開始するときに使用するコマンド・ライン。次に例を示します。 "d:¥sqlany10¥win32¥dbeng10.exe" このメンバが NULL のときは、デフォルトの startline が使用される。 デフォルトの START パラメータ： "dbeng10 -gp page_size -c 10M"
page_size	データベースのページ・サイズ
default_collation	データベースの照合
errorrtn	エラー・メッセージ処理コールバック・ルーチン
msgrtn	情報メッセージ処理コールバック・ルーチン
verbose	「冗長列挙」 818 ページを参照。
blank_pad	NO_BLANK_PADDING または BLANK_PADDING のいずれかを設定する。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する。「ブランク埋め込み列挙」 814 ページを参照。
respect_case	文字列の比較のときに大文字と小文字を区別するようにし、これを反映するインデックス情報を保持する。
encrypt	設定すると、ENCRYPTED ON 句が生成され、encrypted_tables も設定されている場合は、ENCRYPTED TABLES ON 句が生成される。
avoid_view_collisions	Watcom SQL 互換ビュー SYS.SYSCOLUMNS と SYS.SYSINDEXES の世代を除外する。
mirrorname	トランザクション・ログ・ミラー名
jconnect	jConnect に必要なシステム・プロシージャを含める。
data_store_type	予約。NULL を使用する。
encryption_key	データベース・ファイルの暗号化キー。encrypt とともに使用すると、KEY 句を生成する。
encryption_algorithm	暗号化アルゴリズム (AES または AES_FIPS)。encrypt と encryption_key とともに使用すると、ALGORITHM 句を生成する。

メンバ	説明
checksum	ON の場合は 1 に設定し、OFF の場合は 0 に設定。CHECKSUM ON または CHECKSUM OFF 句のいずれかを生成する。
encrypted_tables	暗号化されたテーブルの場合は 1 を設定する。encrypt とともに使用すると、ENCRYPTED ON 句の代わりに ENCRYPTED TABLE ON 句を生成する。
accent_sensitivity	y (はい)、n (いいえ)、または f (フランス語) のいずれか。ACCENT RESPECT、ACCENT IGNORE、ACCENT FRENCH 句のいずれかを生成する。
nchar_collation	NULL でない場合、指定された文字列を使用して NCHAR COLLATION 句を生成するのに使用する。
dba_uid	NULL でない場合、DBA USER xxx 句を生成する。
dba_pwd	NULL でない場合、DBA PASSWORD xxx 句を生成する。
db_size	0 でない場合、DATABASE SIZE 句を生成する。
db_size_unit	db_size とともに使用し、DBSP_UNIT_NONE、DBSP_UNIT_PAGES、DBSP_UNIT_BYTES、DBSP_UNIT_KILOBYTES、DBSP_UNIT_MEGABYTES、DBSP_UNIT_GIGABYTES、DBSP_UNIT_TERABYTES のうちいずれかを指定する。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」 815 ページを参照。

参照

- 「DBCreate 関数」 766 ページ
- 「コールバック関数の使い方」 759 ページ.

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 {
    unsigned short      version;
    MSG_CALLBACK        errorrtn;
    const char *        _unused0;
    unsigned short      dbbufsize;
    char *              dbnamebuffer;
    unsigned short      logbufsize;
    char *              lognamebuffer;
    unsigned short      _unused1;
    char *              _unused2;
    a_bit_field         quiet    : 1;
    a_bit_field         _unused3 : 1;
```



```
    a_sysinfo           sysinfo;
    a_sql_uint32        free_pages;
    a_bit_field         _unused4 : 1;


    const char *        connectparms;
    const char *        _unused5;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        statusrtn;
    a_bit_field         page_usage : 1;
    a_table_info *      totals;
    a_sql_uint32        file_size;
    a_sql_uint32        unused_pages;
    a_sql_uint32        other_pages;
    unsigned short      mirrorbufsize;
    char *              mirrornamebuffer;
    const char *        _unused6;
    char *              collationnamebuffer;
    unsigned short      collationnamebufsize;
    char *              _unused7;
    unsigned short      _unused8;
    a_bit_field         checksum : 1;
    a_bit_field         encrypted_tables : 1;
    a_sql_uint32        bit_map_pages;
} a_db_info;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号
errortrn	エラー・メッセージ処理コールバック・ルーチン
dbbufsize	dbnamebuffer メンバの長さ
dbnamebuffer	データベース・ファイル名
logbufsize	lognamebuffer メンバの長さ
lognamebuffer	トランザクション・ログ・ファイル名
_unused1	(使われていない)
_unused2	(使われていない)
quiet	確認メッセージを出さずに操作する。
sysinfo	a_sysinfo 構造体へのポインタ
free_pages	空きページ数

メンバ	説明
connectparms	データベース接続に必要なパラメータ。次のような接続文字列のフォームをとる。 "UID=DBA;PWD=sql;DBF=samples-dir¥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。次に例を示します。 "START=d:¥sqlany10¥win32¥dbeng10.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=samples-dir¥demo.db;START=d:¥sqlany10¥win32¥dbeng10.exe" 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。
msgrtn	情報メッセージ処理コールバック・ルーチン
statusrtn	ステータス・メッセージ処理コールバック・ルーチン
page_usage	1 の場合、ページ使用統計をレポートする。レポートが必要なければ 0。
totals	a_table_info 構造体へのポインタ
file_size	データベース・ファイルのサイズ
unused_pages	未使用空きページ数
other_pages	テーブル・ページ、インデックス・ページ、空きページ、ビットマップ・ページのいずれでもないページの数
mirrorbufsize	mirrornamebuffer メンバの長さ
mirrornamebuffer	トランザクション・ログ・ミラー名
collationnamebuffer	データベースの照合名と照合ラベル (最大サイズは 128+1)
collationnamebufsize	collationnamebuffer メンバの長さ
checksum	1 の場合はページ・チェックサムが有効で、0 の場合は無効
encrypted_tables	1 の場合は暗号化されたテーブルはサポートされ、0 の場合は無効
bit_map_pages	データベース内のビットマップ・ページ数

参照

- 「DBInfo 関数」 767 ページ
- 「コールバック関数の使い方」 759 ページ.

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 {
        unsigned short  version;
        const char      *filename;
        a_db_version    created_version;
        MSG_CALLBACK    errorrtn;
        MSG_CALLBACK    msgrtn;
  } a_db_version_info;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号
filename	確認するデータベース・ファイルの名前
created_version	データベース・ファイルを作成したサーバ・バージョンを示す a_db_version 型の値に設定される。「a_db_version 列挙」 815 ページを参照。
errorrtn	エラー・メッセージ処理コールバック・ルーチン
msgrtn	情報メッセージ処理コールバック・ルーチン

参照

- 「DBCreatedVersion 関数」 766 ページ
- 「a_db_version 列挙」 815 ページ
- 「コールバック関数の使い方」 759 ページ

a_dblic_info 構造体

ライセンス情報などを格納します。この情報は、ライセンス契約に従って使用してください。

構文

```
typedef struct a_dblic_info {
    unsigned short      version;
    char                *exename;
    char                *username;
    char                *compname;
    char                *_unused1;
    a_sql_int32         nodecount;
    a_sql_int32         conncount;
    a_license_type      type;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    a_bit_field         quiet           : 1;
    a_bit_field         query_only      : 1;
    char                *installkey;
} a_dblic_info;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号
exename	サーバ実行プログラムまたはライセンス・ファイルの名前
username	ライセンスのユーザ名
compname	ライセンスの会社名
nodecount	ライセンス・ノード数
conncount	1000000L に設定する。
type	値については *lictype.h* を参照。
errorrtn	エラー・メッセージ処理コールバック・ルーチン
msgrtn	情報メッセージ処理コールバック・ルーチン
quiet	操作中にメッセージを出力する (0) またはしない (1)
query_only	1 の場合、単にライセンス情報が表示される。0 の場合は情報を変更可能。
installkey	内部でのみ使用。NULL に設定されます。

a_dbtools_info 構造体

DBTools ライブラリの使用を開始および終了するために必要な情報を格納します。

構文

```
typedef struct a_dbtools_info {
    MSG_CALLBACK        errorrtn;
} a_dbtools_info;
```

メンバ

メンバ	説明
errorrtn	エラー・メッセージ処理コールバック・ルーチン

参照

- 「DBToolsFini 関数」 770 ページ
- 「DBToolsInit 関数」 771 ページ
- 「コールバック関数の使い方」 759 ページ.

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 {
    unsigned short      version;
    const char *        dbname;
    MSG_CALLBACK        confirmrtn;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    a_bit_field         quiet : 1;
    a_bit_field         erase : 1;
    const char *        encryption_key;
} an_erase_db;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号
dbname	消去するデータベース・ファイル名
confirmrtn	動作確認コールバック・ルーチン
errorrtn	エラー・メッセージ処理コールバック・ルーチン
msgrtn	情報メッセージ処理コールバック・ルーチン
quiet	操作中にメッセージを出力する (0) またはしない (1)
erase	消去するときに確認する (0) またはしない (1)
encryption_key	データベース・ファイルの暗号化キー

参照

- 「DBErase 関数」 767 ページ
- 「コールバック関数の使い方」 759 ページ.

a_name 構造体

名前のリンク・リストを格納します。名前のリストを必要とする他の構造体が使用します。

構文

```
typedef struct a_name {
    struct a_name *next;
    char          name[1];
} a_name, * p_name;
```

メンバ

メンバ	説明
next	リスト内の次の a_name 構造体へのポインタ
name	名前

参照

- 「a_translate_log 構造体」 802 ページ
- 「a_validate_db 構造体」 812 ページ
- 「an_unload_db 構造体」 807 ページ

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 {
    short               version;
    MSG_CALLBACK        confirmrtn;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_QUEUE_CALLBACK  msgqueuertn;
    char *              connectparms;
    char *              transaction_logs;

    a_bit_field         receive : 1;
    a_bit_field         send : 1;
    a_bit_field         verbose : 1;
    a_bit_field         deleted : 1;
    a_bit_field         apply : 1;
    a_bit_field         batch : 1;
    a_bit_field         more : 1;
    a_bit_field         triggers : 1;
    a_bit_field         debug : 1;
    a_bit_field         rename_log : 1;
    a_bit_field         latest_backup : 1;
    a_bit_field         scan_log : 1;
    a_bit_field         link_debug : 1;
    a_bit_field         full_q_scan : 1;
    a_bit_field         no_user_interaction : 1;
    a_bit_field         _unused1 : 1;

    a_sql_uint32        max_length;
    a_sql_uint32        memory;
    a_sql_uint32        frequency;
    a_sql_uint32        threads;
    a_sql_uint32        operations;
    char *              queueparms;
    char *              locale;
    a_sql_uint32        receive_delay;
    a_sql_uint32        patience_retry;
    MSG_CALLBACK        logrtn;

    a_bit_field         use_hex_offsets : 1;
    a_bit_field         use_relative_offsets : 1;
```

```
    a_bit_field        debug_page_offsets : 1;
    a_sql_uint32       debug_dump_size;
    a_sql_uint32       send_delay;
    a_sql_uint32       resend_urgency;


    char *             include_scan_range;
    SET_WINDOW_TITLE_CALLBACK set_window_title_rtn;
    char *             default_window_title;
    MSG_CALLBACK       progress_msg_rtn;
    SET_PROGRESS_CALLBACK progress_index_rtn;
    char **            argv;
    a_sql_uint32       log_size;
    char *             encryption_key;
    char *             log_file_name;
    a_bit_field        truncate_remote_output_file:1;
    char *             remote_output_file_name;
    MSG_CALLBACK       warningrtn;
    char *             mirror_logs;
} a_remote_sql;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号
confirmrtn	指定されたメッセージを表示する関数へのポインタ。yes または no による応答を受け付ける。yes の場合は TRUE を返し、no の場合は FALSE を返します。
errorrtn	指定されたエラー・メッセージを表示する関数へのポインタ
msgrtn	指定された (エラー以外の) 情報メッセージを表示する関数へのポインタ
msgqueuertn	指定された時間 (ミリ秒) が経過したらスリープ状態になる関数へのポインタ。この関数は、0 に設定され、DBRemoteSQL がビジー状態だが上位レイヤでメッセージが処理されるようにする場合に呼び出されます。このルーチンは、通常、MSGQ_SLEEP_THROUGH を返し、SQL Remote 処理を停止する場合は MSGQ_SHUTDOWN_REQUESTED を返します。

メンバ	説明
connectparms	データベース接続に必要なパラメータ。dbremote の -c オプションに対応。次のような接続文字列のフォームをとる。 "UID=DBA;PWD=sql;DBF=samples-dir¥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。次に例を示します。 "START=d:¥sqlany10¥win32¥dbeng10.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=samples-dir¥demo.db;START=d:¥sqlany10¥win32¥dbeng10.exe" 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。
transaction_logs	オフライン・トランザクション・ログでディレクトリの名前を表わす文字列へのポインタ。dbremote の transaction_logs_directory 引数に対応。
receive	true の場合、メッセージを受信する。dbremote の -r オプションに対応。 receive と send の両方が false の場合、両方とも true であるとみなされます。receive と send の両方を false に設定することをおすすめします。
send	true の場合、メッセージを送信する。dbremote の -s オプションに対応。 receive と send の両方が false の場合、両方とも true であるとみなされます。receive と send の両方を false に設定することをおすすめします。
verbose	true の場合、追加情報を表示する。dbremote の -v オプションに対応。
deleted	true に設定。false に設定した場合、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。
apply	true に設定。false に設定した場合、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。
batch	true の場合、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する。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。false の場合、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。
more	true に設定。

メンバ	説明
triggers	通常は false に設定。true に設定すると、DBRemoteSQL によってトリガ動作がレプリケートされます。dbremote の -t オプションに対応します。
debug	true に設定すると、デバッグの出力結果を含む。
rename_log	true に設定すると、ログの名前が変更され、再起動される。
latest_backup	true に設定すると、バックアップされているログのみ処理する。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。
scan_log	予約。false に設定。
link_debug	true に設定すると、リンクのデバッグが有効になる。
full_q_scan	予約。false に設定。
no_user_interaction	true に設定すると、ユーザの操作を必要としない。
max_length	メッセージの最大長をバイト単位で設定する。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	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する。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。
frequency	受信メッセージのポーリング頻度を設定する。この値は max (1, receive_delay/60) にしてください。以下の receive_delay を参照。
threads	メッセージの適用に使用するワーカ・スレッド数を設定する。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。
operations	メッセージを適用するときに使用される値。DBRemoteSQL でコミットされていない操作 (挿入、削除、更新) の数がこの値に達するまで、コミットは無視される。dbremote の -g オプションに対応します。
queueparms	予約。NULL を設定。
locale	予約。NULL を設定。
receive_delay	新しいメッセージ受信するポーリングの待機間隔を秒単位で設定する。推奨値は 60 です。dbremote の -rd オプションに対応します。

メンバ	説明
patience_retry	受信メッセージが失われたとみなされるまでに DBRemoteSQL が待機する受信メッセージのポーリング回数を設定する。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。
logrtn	指定されたメッセージをログ・ファイルに出力する関数へのポインタ。メッセージをユーザに表示する必要はありません。
use_hex_offsets	ログ・オフセットを 16 進表記で表示する場合は、true に設定する。そうでない場合は、小数表記が使用されます。
use_relative_offsets	ログ・オフセットを現在のログ・ファイルの開始点への相対値として表示する場合は、true に設定する。ログ・オフセットを開始時刻から表示する場合は、false に設定します。
debug_page_offsets	予約。false に設定。
debug_dump_size	予約。0 に設定。
send_delay	新しい操作のログ・ファイルのスキャンを送信するまでの時間を秒単位で設定する。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。
resend_urgency	ユーザが再スキャンを必要としていることが判明してからログのフル・スキャンを実行するまでの DBRemoteSQL の待機時間を秒単位で設定する。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。
include_scan_range	予約。NULL を設定。
set_window_title_rtn	ウィンドウのタイトルをリセットする関数へのポインタ (Windows の場合のみ)。タイトルは "*database_name* (受信中、スキャン中、送信中) - *default_window_title*" の形式になります。
default_window_title	デフォルトのウィンドウ・タイトルを表わす文字列へのポインタ
progress_msg_rtn	進行状況メッセージを表示する関数へのポインタ
progress_index_rtn	進行状況バーのステータスを更新する関数へのポインタ。この関数には符号なしの整数を指定する 2 つの引数 *index* と *max* があります。最初の呼び出しでは、2 つの引数の値は最小値と最大値 (例：0 と 100) になります。2 回目以降の呼び出しでは、最初の引数は現在のインデックス値 (例：0 ー 100) になり、2 番目の引数は常に 0 になります。

メンバ	説明
argv	解析されたコマンド・ラインへのポインタ (文字列へのポインタのベクトル)。NULL 以外の場合、DBRemoteSQL はメッセージ・ルーチンを呼び出して、-c、-cq、-ek で始まる引数を除いた各コマンド・ラインの引数を表示します。
log_size	オンライン・トランザクション・ログのサイズがこの値より大きくなると、DBRemoteSQL はオンライン・トランザクション・ログの名前を変更して再起動する。dbremote の -x オプションに対応します。
encryption_key	暗号化キーへのポインタ。dbremote の -ek オプションに対応。
log_file_name	メッセージ・コールバックによって出力が書き込まれる DBRemoteSQL 出力ログの名前へのポインタ。*send* が true の場合、エラー・ログが統合データベースに送信されます (ただし、このポインタの値が NULL 以外の場合)。
truncate_remote_output_file	true に設定すると、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る。以下を参照。dbremote の -rt オプションに対応します。
remote_output_file_name	DBRemoteSQL リモート出力ファイルの名前へのポインタ。dbremote の -ro または -rt オプションに対応。
warningrtn	指定された警告メッセージを表示する関数へのポインタ。NULL の場合、*errorrtn* 関数が代わりに呼び出されます。
mirror_logs	オフライン・ミラー・トランザクション・ログを含むディレクトリの名前へのポインタ。dbremote の -ml オプションに対応。

dbremote ツールは、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 関数」 769 ページ
- 「dbmlsync の DBTools インタフェース」 『Mobile Link - クライアント管理』

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 {
    unsigned short      version;
    char *              connectparms;
    char *              publication;
    const char *        offline_dir;
    char *              extended_options;
    char *              script_full_path;
    const char *        include_scan_range;
    const char *        raw_file;
    MSG_CALLBACK        confirmrtn;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        logrtn;

    a_sql_uint32        debug_dump_size;
    a_sql_uint32        dl_insert_width;
    a_bit_field         verbose             : 1;
    a_bit_field         debug               : 1;
    a_bit_field         debug_dump_hex      : 1;
    a_bit_field         debug_dump_char     : 1;
    a_bit_field         debug_page_offsets  : 1;
    a_bit_field         use_hex_offsets     : 1;
    a_bit_field         use_relative_offsets : 1;
    a_bit_field         output_to_file      : 1;
    a_bit_field         output_to_mobile_link : 1;
    a_bit_field         dl_use_put          : 1;
    a_bit_field         dl_use_upsert       : 1;
    a_bit_field         kill_other_connections : 1;
    a_bit_field         retry_remote_behind : 1;
    a_bit_field         ignore_debug_interrupt : 1;

    SET_WINDOW_TITLE_CALLBACK set_window_title_rtn;
    char *              default_window_title;
```

```
MSG_QUEUE_CALLBACK  msgqueuertn;
MSG_CALLBACK        progress_msg_rtn;
SET_PROGRESS_CALLBACK progress_index_rtn;
char **             argv;
char **             ce_argv;
a_bit_field         connectparms_allocated  : 1;
a_bit_field         entered_dialog          : 1;
a_bit_field         used_dialog_allocation  : 1;
a_bit_field         ignore_scheduling       : 1;
a_bit_field         ignore_hook_errors      : 1;
a_bit_field         changing_pwd            : 1;
a_bit_field         prompt_again            : 1;
a_bit_field         retry_remote_ahead      : 1;
a_bit_field         rename_log              : 1;
a_bit_field         hide_conn_str           : 1;
a_bit_field         hide_ml_pwd             : 1;
a_sql_uint32        dlg_launch_focus;
char *              mlpassword;
char *              new_mlpassword;
char *              verify_mlpassword;
a_sql_uint32        pub_name_cnt;
char **             pub_name_list;
USAGE_CALLBACK      usage_rtn;

a_sql_uint32        log_size;
a_sql_uint32        hovering_frequency;
a_bit_short         ignore_hovering         : 1;
a_bit_short         verbose_upload          : 1;
a_bit_short         verbose_upload_data     : 1;
a_bit_short         verbose_download        : 1;
a_bit_short         verbose_download_data   : 1;
a_bit_short         autoclose               : 1;
a_bit_short         ping                    : 1;
a_bit_short         _unused                 : 9;
char *              encryption_key;
a_syncpub *         upload_defs;
char *              log_file_name;
char *              user_name;
a_bit_short         verbose_minimum         : 1;
a_bit_short         verbose_hook            : 1;
a_bit_short         verbose_row_data        : 1;
a_bit_short         verbose_row_cnts        : 1;
a_bit_short         verbose_option_info     : 1;
a_bit_short         strictly_ignore_trigger_ops : 1;
a_bit_short         _unused2                : 10;
a_sql_uint32        est_upld_row_cnt;
STATUS_CALLBACK     status_rtn;
MSG_CALLBACK        warningrtn;
char **             ce_reproc_argv;

a_bit_short         upload_only             : 1;
a_bit_short         download_only           : 1;
a_bit_short         allow_schema_change     : 1;
a_bit_short         dnld_gen_num            : 1;
a_bit_short         _unused3                :12;
const char *        apply_dnld_file;
const char *        create_dnld_file;
char *              sync_params;
const char *        dnld_file_extra;
COMServer *         com_server;
a_bit_short         trans_upload            : 1;
a_bit_short         continue_download       : 1;
a_bit_short         lite_blob_handling      : 1;
a_sql_uint32        dnld_read_size;
```

```
    a_sql_uint32      dnld_fail_len;
    a_sql_uint32      upld_fail_len;
    a_bit_short       persist_connection     :1;
    a_bit_short       verbose_protocol       :1;
    a_bit_short       no_stream_compress     :1;
    a_bit_short       _unused4               :13;
    char              _unused5;
} a_sync_db;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号
connectparms	データベース接続に必要なパラメータ。次のような接続文字列のフォームをとる。 "UID=DBA;PWD=sql;DBF=samples-dir¥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。次に例を示します。 "START=d:¥sqlany10¥win32¥dbeng10.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=samples-dir¥demo.db;START=d:¥sqlany10¥win32¥dbeng10.exe" 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。
publication	旧式。NULL を使用。
offline_dir	ログ・ディレクトリ。オプションに続いてコマンド・ラインで指定。
extended_options	拡張オプション。-e を使用して指定。
script_full_path	旧式。NULL を使用。
include_scan_range	予約。NULL を使用。
raw_file	予約。NULL を使用。
confirmrtn	予約。NULL を使用。
errorrtn	エラー・メッセージを表示する関数
msgrtn	ユーザ・インタフェース、およびオプションとしてログ・ファイルにメッセージを書き込む関数
logrtn	ログ・ファイルのみにメッセージを書き込む関数

メンバ	説明
debug_dump_size	予約。0 を使用。
dl_insert_width	予約。0 を使用。
verbose	旧式。0 を使用。
debug	予約。0 を使用。
debug_dump_hex	予約。0 を使用。
debug_dump_char	予約。0 を使用。
debug_page_offsets	予約。0 を使用。
use_hex_offsets	予約。0 を使用。
use_relative_offsets	予約。0 を使用。
output_to_file	予約。0 を使用。
output_to_mobile_link	予約。1 を使用。
dl_use_put	予約。0 を使用。
dl_use_upsert	予約。0 を使用。
kill_other_connections	-d オプションが指定されている場合は TRUE
retry_remote_behind	-r または -rb オプションが指定されている場合は TRUE
ignore_debug_interrupt	予約。0 を使用。
set_window_title_rtn	dbmlsync ウィンドウのタイトルを変更するために呼び出す関数 (Windows のみ)
default_window_title	ウィンドウ・キャプションに表示するプログラム名 (DBMLSync など)

メンバ	説明
msgqueuertn	DBMLSync がスリープするときに呼び出す関数。このパラメータには、目的の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。ほとんどの場合、この値が返されます。 ♦ 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。
progress_msg_rtn	進行状況バーの上部のステータス・ウィンドウのテキストを変更する関数
progress_index_rtn	進行状況バーのステータスを更新する関数
argv	この実行における argv 配列。配列の最後の要素は NULL です。
ce_argv	予約。NULL を使用。
connectparms_allocated	予約。0 を使用。
entered_dialog	予約。0 を使用。
used_dialog_allocation	予約。0 を使用。
ignore_scheduling	-is が指定されている場合は TRUE
ignore_hook_errors	-eh が指定されている場合は TRUE
changing_pwd	-mn が指定されている場合は TRUE
prompt_again	予約。0 を使用。
retry_remote_ahead	-ra が指定されている場合は TRUE
rename_log	-x が指定されている場合は TRUE。この場合、ログ・ファイルは名前が変更され、再起動されます。
hide_conn_str	-vc が指定されていない場合は TRUE
hide_ml_pwd	-vp が指定されていない場合は TRUE
dlg_launch_focus	予約。0 を使用。

メンバ	説明
mlpassword	-mp を使用して指定した Mobile Link のパスワード。そうでない場合は、NULL。
new_mlpassword	-mn を使用して指定した新しい Mobile Link のパスワード。そうでない場合は、NULL。
verify_mlpassword	予約。NULL を使用。
pub_name_cnt	旧式。0 を使用。
pub_name_list	旧式。NULL を使用。
usage_rtn	予約。NULL を使用。
log_size	-x を使用して指定したバイト単位のログ・サイズ。そうでない場合は 0。
hovering_frequency	-pp を使用して設定した秒単位の停止頻度
ignore_hovering	-p が指定されている場合は TRUE
verbose_upload	-vu が指定されている場合は TRUE
verbose_upload_data	予約。0 を使用。
verbose_download	予約。0 を使用。
verbose_download_data	予約。0 を使用。
autoclose	-k が指定されている場合は TRUE
ping	-pi が指定されている場合は TRUE
encryption_key	-ek を使用して指定したデータベース・キー
upload_defs	まとめてアップロードされるパブリケーションのリンク・リスト。a_syncpub を参照。
log_file_name	-o または -ot を使用して指定した出力ログ・ファイルの名前
user_name	-u を使用して指定した Mobile Link のユーザ名
verbose_minimum	-v が指定されている場合は TRUE
verbose_hook	-vs が指定されている場合は TRUE
verbose_row_data	-vr が指定されている場合は TRUE
verbose_row_cnts	-vn が指定されている場合は TRUE
verbose_option_info	-vo が指定されている場合は TRUE

メンバ	説明
strictly_ignore_trigger_ops	予約。0 を使用。
est_upld_row_cnt	-urc を使用して指定した、アップロードするローの推定数
status_rtn	予約。NULL を使用。
warningrtn	警告メッセージを表示する関数
ce_reproc_argv	予約。NULL を使用。
upload_only	-uo が指定されている場合は TRUE
download_only	-ds が指定されている場合は TRUE
allow_schema_change	-sc が指定されている場合は TRUE
dnld_gen_num	-bg が指定されている場合は TRUE
apply_dnld_file	-ba を使用して指定したファイル。そうでない場合は NULL。
create_dnld_file	-bc を使用して指定したファイル。そうでない場合は NULL。
sync_params	ユーザ認証パラメータ。-ap で指定。
dnld_file_extra	-be を使用して指定した文字列
com_server	予約。NULL を使用。
trans_upload	-tu が指定されている場合は TRUE
continue_download	-dc が指定されている場合は TRUE
dnld_read_size	-drs オプションで指定した値
dnld_fail_len	予約。0 を使用。
upld_fail_len	予約。0 を使用。
persist_connection	-pp がコマンド・ラインで指定された場合は TRUE
verbose_protocol	予約。0 を使用。
no_stream_compress	予約。0 を使用。

一部のメンバは、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。

詳細については、*dbtools.h* ヘッダ・ファイルを参照してください。

詳細については、「dbmlsync 構文」 『Mobile Link - クライアント管理』を参照してください。

参照

- 「dbmlsync の DBTools インタフェース」 『Mobile Link - クライアント管理』
- 「DBSynchronizeLog 関数」 770 ページ

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 {
    struct a_syncpub *  next;
    char *              pub_name;
    char *              ext_opt;
    a_bit_field         alloced_by_dbsync: 1;
} a_syncpub;
```

メンバ

メンバ	説明
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	この -n オプションに指定するパブリケーション名。コマンド・ラインで -n の後に指定する正確な文字列。
ext_opt	-eu オプションを使用して指定する拡張オプション
alloced_by_dbsync	予約。FALSE を使用。

参照

- 「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。

```
typedef struct a_sysinfo {
    a_bit_field     valid_data          : 1;
    a_bit_field     blank_padding       : 1;
    a_bit_field     case_sensitivity    : 1;
    a_bit_field     encryption          : 1;
    char            default_collation[11];
    unsigned short  page_size;
} a_sysinfo;
```

メンバ

メンバ	説明
valid_date	後続の値が設定されているかどうかを示すビットフィールド
blank_padding	1 の場合、このデータベースではブランクの埋め込みをする。0 の場合はしない。
case_sensitivity	1 の場合、データベースは大文字と小文字を区別。0 の場合はしない。
encryption	1 の場合、データベースは暗号化されている。0 の場合はされていない。
default_collation	データベースの照合順
page_size	データベースのページ・サイズ

参照

- 「a_db_info 構造体」 781 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 {
    struct a_table_info *next;
    a_sql_uint32        table_id;
    a_sql_uint32        table_pages;
    a_sql_uint32        index_pages;
    a_sql_uint32        table_used;
    a_sql_uint32        index_used;
    char *              table_name;
    a_sql_uint32        table_used_pct;
    a_sql_uint32        index_used_pct;
} a_table_info;
```

メンバ

メンバ	説明
next	リスト内の次のテーブル
table_id	このテーブルの ID 番号
table_pages	テーブル・ページの数
index_pages	インデックス・ページの数
table_used	テーブル・ページに使用されているバイト数
index_used	インデックス・ページに使用されているバイト数

メンバ	説明
table_name	テーブルの名前
table_used_pct	テーブル領域の使用率を示すパーセンテージ
index_used_pct	インデックス領域の使用率を示すパーセンテージ

参照

- 「a_db_info 構造体」 781 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 {
    unsigned short      version;
    const char *        logname;
    const char *        sqlname;
    p_name              userlist;
    MSG_CALLBACK        confirmrtn;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    char                userlisttype;


    a_bit_field         remove_rollback : 1;
    a_bit_field         ansi_sql        : 1;
    a_bit_field         since_checkpoint: 1;
    a_bit_field         omit_comments   : 1;
    a_bit_field         replace         : 1;
    a_bit_field         debug           : 1;
    a_bit_field         include_trigger_trans : 1;
    a_bit_field         comment_trigger_trans : 1;
    a_sql_uint32        since_time;
    const char *        connectparms;
    MSG_CALLBACK        logrtn;


    const char *        _unused1;
    const char *        _unused2;
    a_sql_uint32        debug_dump_size;
    a_bit_field         debug_sql_remote        : 1;
    a_bit_field         debug_dump_hex          : 1;
    a_bit_field         debug_dump_char         : 1;
    a_bit_field         debug_page_offsets      : 1;
    a_bit_field         _unused3                : 1;
    a_bit_field         use_hex_offsets         : 1;
    a_bit_field         use_relative_offsets    : 1;
    a_bit_field         include_audit           : 1;
    a_bit_field         chronological_order     : 1;
    a_bit_field         force_recovery          : 1;
    a_bit_field         include_subsets         : 1;
    a_bit_field         force_chaining          : 1;
```

```
    a_sql_uint32        recovery_ops;
    a_sql_uint32        recovery_bytes;


    const char *        include_source_sets;
    const char *        include_destination_sets;
    const char *        include_scan_range;
    const char *        repserver_users;
    const char *        include_tables;
    const char *        include_publications;
    const char *        queueparms;
    a_bit_field         generate_reciprocals    :1;
    a_bit_field         match_mode              :1;
    const char *        match_pos;
    MSG_CALLBACK        statusrtn;
    const char *        encryption_key;
    a_bit_field         show_undo               :1;
    a_bit_field         quiet                   :1;
    const char *        logs_dir;
} a_translate_log;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号
logname	トランザクション・ログ・ファイルの名前。NULL を設定すると、ログは取られない。
sqlname	SQL 出力ファイルの名前。NULL を設定すると、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る (-n で文字列を設定)。
userlist	リンクされたユーザ名のリスト。-u user1,... または -x user1,... と同等。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。
confirmrtn	動作確認コールバック・ルーチン
errorrtn	エラー・メッセージ処理コールバック・ルーチン
msgrtn	情報メッセージ処理コールバック・ルーチン
userlisttype	ユーザのリストを含めるか除外する場合を除き、DBTRAN_INCLUDE_ALL に設定する。-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。
remove_rollback	通常は TRUE に設定する。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 (-a と同等)。
ansi_sql	ANSI 標準の SQL トランザクションを生成する場合は、TRUE に設定する (-s と同等)。
since_checkpoint	最新のチェックポイントから出力する場合は、TRUE に設定する (-f と同等)。
omit_comments	予約。FALSE に設定。

メンバ	説明
replace	確認メッセージを表示せずに既存の SQL ファイルを置換する (-y と同等)。
debug	予約。FALSE に設定。
include_trigger_trans	TRUE に設定すると、トリガ生成トランザクションを含める (-g、-sr、-t と同等)。
comment_trigger_trans	TRUE に設定すると、トリガ生成トランザクションをコメントとして含める (-z と同等)。
since_time	指定された時刻 (-j <time> で設定) より前の最新のチェックポイントから出力する。西暦 1 年 1 月 1 日からの分数。
connectparms	データベース接続に必要なパラメータ。次のような接続文字列のフォームをとる。 "UID=DBA;PWD=sql;DBF=samples-dir¥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。次に例を示します。 "START=d:¥sqlany10¥win32¥dbeng10.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=samples-dir¥demo.db;START=d:¥sqlany10¥win32¥dbeng10.exe" 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。
logrtn	ログ・ファイルのみにメッセージを書き込むコールバック・ルーチン
debug_dump_size	予約。0 を使用。
debug_sql_remote	予約。FALSE を使用。
debug_dump_hex	予約。FALSE を使用。
debug_dump_char	予約。FALSE を使用。
debug_page_offsets	予約。FALSE を使用。
use_hex_offsets	予約。FALSE を使用。
use_relative_offsets	予約。FALSE を使用。
include_audit	予約。FALSE を使用。
chronological_order	予約。FALSE を使用。
force_recovery	予約。FALSE を使用。

メンバ	説明
include_subsets	予約。FALSE を使用。
force_chaining	予約。FALSE を使用。
recovery_ops	予約。0 を使用。
recovery_bytes	予約。0 を使用。
include_source_sets	予約。NULL を使用。
include_destination_sets	予約。NULL を使用。
include_scan_range	予約。NULL を使用。
repserver_users	予約。NULL を使用。
include_tables	予約。NULL を使用。
include_publications	予約。NULL を使用。
queueparms	予約。NULL を使用。
generate_reciprocals	予約。FALSE を使用。
match_mode	予約。FALSE を使用。
match_pos	予約。NULL を使用。
statusrtn	ステータス・メッセージ処理コールバック・ルーチン
encryption_key	データベース暗号化キーを指定する (-ek で文字列を設定)。
show_undo	予約。FALSE を使用。
quiet	TRUE に設定すると、操作中にメッセージを出力しない (-y)。
logs_dir	トランザクション・ログ・ディレクトリ (-m *dir* で文字列を設定)。 sqlname を設定し、connect_parms に NULL を設定してください。

各メンバは、dbtran 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。

詳細については、*dbtools.h* ヘッダ・ファイルを参照してください。

参照

- 「DBTranslateLog 関数」 772 ページ
- 「a_name 構造体」 786 ページ
- 「dbtran_userlist_type 列挙」 816 ページ
- 「コールバック関数の使い方」 759 ページ.

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 {
    unsigned short      version;
    const char *        connectparms;
    const char *        _unused1;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    a_bit_field         quiet           : 1;
    a_bit_field         server_backup   : 1;
    char                truncate_interrupted;
} a_truncate_log;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号
connectparms	データベース接続に必要なパラメータ。次のような接続文字列のフォームをとる。 "UID=DBA;PWD=sql;DBF=samples-dir¥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。次に例を示します。 "START=d:¥sqlany10¥win32¥dbeng10.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=samples-dir¥demo.db;START=d:¥sqlany10¥win32¥dbeng10.exe" 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。
errorrtn	エラー・メッセージ処理コールバック・ルーチン
msgrtn	情報メッセージ処理コールバック・ルーチン
quiet	操作中にメッセージを出力する (0) またはしない (1)
server_backup	1 に設定すると、バックアップ・データベースを使用したサーバでのバックアップを示す。dbbackup の -s オプションと同等。
truncate_interrupted	オペレーションが中断されたことを示す。

参照

- 「DBTruncateLog 関数」 772 ページ
- 「コールバック関数の使い方」 759 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。dbxtract SQLRemote 抽出ユーティリティが使用するフィールドが示されます。

構文

```
typedef struct an_unload_db {
    unsigned short      version;
    const char *        connectparms;
    const char *        startline;
    const char *        temp_dir;
    const char *        reload_filename;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        statusrtn;
    MSG_CALLBACK        confirmrtn;
    char                unload_type;
    char                verbose;


    a_bit_field         unordered               : 1;
    a_bit_field         no_confirm              : 1;
    a_bit_field         use_internal_unload     : 1;
    a_bit_field         _unused1                : 1;
    a_bit_field         extract                 : 1;
    a_bit_field         table_list_provided     : 1;
    a_bit_field         exclude_tables          : 1;
    a_bit_field         more_flag_bits_present  : 1;
    a_sysinfo           sysinfo;


    const char *        remote_dir;
    const char *        _unused2;
    const char *        subscriber_username;
    const char *        _unused3;
    const char *        _unused4;
    unsigned short      isolation_level;
    a_bit_field         start_subscriptions     : 1;
    a_bit_field         exclude_foreign_keys    : 1;
    a_bit_field         exclude_procedures      : 1;
    a_bit_field         exclude_triggers        : 1;
    a_bit_field         exclude_views           : 1;
    a_bit_field         isolation_set           : 1;
    a_bit_field         include_where_subscribe : 1;
    a_bit_field         debug                   : 1;
    p_name              table_list;
    a_bit_short         escape_char_present     : 1;
    a_bit_short         _unused5                : 1;
    a_bit_short         use_internal_reload     : 1;
    unsigned short      _unused6;


    char                escape_char;
    char *              reload_connectparms;
    char *              reload_db_filename;
    a_bit_field         _unused7                : 1;
    char                unload_interrupted;
    a_bit_field         replace_db              : 1;
    const char *        locale;
    const char *        site_name;
```

```
    const char *       template_name;
    a_bit_field        preserve_ids           : 1;
    a_bit_field        exclude_hooks          : 1;
    char *             reload_db_logname;
    const char *       encryption_key;
    const char *       encryption_algorithm;
    a_bit_field        _unused8               : 1;
    a_bit_field        _unused9               : 1;
    unsigned short     reload_page_size;
    a_bit_field        recompute              : 1;
    a_bit_field        make_auxiliary         : 1;
    a_bit_field        encrypted_tables       : 1;
    a_bit_field        remove_encrypted_tables : 1;
} an_unload_db;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号
connectparms	データベース接続に必要なパラメータ。次のような接続文字列のフォームをとる。 "UID=DBA;PWD=sql;DBF=samples-dir¥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。次に例を示します。 "START=d:¥sqlany10¥win32¥dbeng10.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=samples-dir¥demo.db;START=d:¥sqlany10¥win32¥dbeng10.exe" 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。
startline	旧式。startline フィールドは使用されなくなりました。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。詳細については connectparms を参照してください。
temp_dir	データ・ファイルのアンロード用ディレクトリ
reload_filename	dbunload -r オプション。"reload.sql" など。
errorrtn	エラー・メッセージ処理コールバック・ルーチン
msgrtn	情報メッセージ処理コールバック・ルーチン
statusrtn	ステータス・メッセージ処理コールバック・ルーチン
confirmrtn	動作確認コールバック・ルーチン
unload_type	「dbunload type 列挙」 817 ページを参照。

メンバ	説明
verbose	「冗長列挙」 818 ページを参照。
unordered	dbunload -u は TRUE を設定。
no_confirm	dbunload -y は TRUE を設定。
use_internal_unload	dbunload -i? は TRUE を設定。
extract	dbxtract の場合は TRUE、それ以外の場合は FALSE
table_list_provided	dbunload -e *list*、または -i は TRUE を設定。
exclude_tables	dbunload -e は TRUE を設定。dbunload -i (マニュアルに記載されていない) は FALSE を設定。
more_flag_bits_present	通常は TRUE に設定。
sysinfo	(内部使用)
remote_dir	temp_dir に類似するが、内部のアンロード用 // サーバ側
subscriber_username	dbxtract の引数
isolation_level	dbxtract -l は値を設定。
start_subscriptions	デフォルトでは dbxtract TRUE。-b は FALSE を設定。
exclude_foreign_keys	dbxtract -xf は TRUE を設定。
exclude_procedures	dbxtract -xp は TRUE を設定。
exclude_triggers	dbxtract -xt は TRUE を設定。
exclude_views	dbxtract -xv は TRUE を設定。
isolation_set	dbxtract -l は TRUE を設定。
include_where_subscribe	dbxtract -f は TRUE を設定。
debug	(内部使用)
table_list	選択的なテーブル・リスト
escape_char_present	-p は TRUE を設定し、escape_char の設定を必須にする。
use_internal_reload	通常 TRUE に設定。-ix/-xx は FALSE を設定。-ii/-xi は TRUE を設定。
escape_char	escape_char_present が TRUE の場合に使用。
reload_connectparms	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

メンバ	説明
reload_db_filename	データベースを再ロードして作成するファイル名
unload_interrupted	アンロードが中断されると設定される。
replace_db	dbunload -ar は TRUE を設定。
locale	(内部使用) ロケール (言語と文字セット)
site_name	dbxtract でサイト名を指定。
template_name	dbxtract でテンプレート名を指定。
preserve_ids	dbunload は TRUE を設定。-m は FALSE を設定。
exclude_hooks	dbxtract -hx は TRUE を設定。
reload_db_logname	再ロード・データベースのログ・ファイル名
encryption_key	-ek は文字列を設定。
encryption_algorithm	-ea は "aes" または "aes_fips" を設定。
reload_page_size	dbunload -ap は値を設定。再構築したデータベースのページ・サイズを設定します。
recompute	dbunload -dc は TRUE を設定。すべての計算カラムを再計算します。
make_auxiliary	dbunload -k は TRUE を設定。(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。
encrypted_tables	dbunload -et は TRUE を設定。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)。
remove_encrypted_tables	dbunload -er は TRUE を設定。暗号化されたテーブルから暗号化を削除します。

各メンバは、dbunload ユーティリティと dbxtract 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。

詳細については、*dbtools.h* ヘッダ・ファイルを参照してください。

参照

- 「DBUnload 関数」 773 ページ
- 「a_name 構造体」 786 ページ
- 「dbunload type 列挙」 817 ページ
- 「冗長列挙」 818 ページ
- 「コールバック関数の使い方」 759 ページ.

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 {
    unsigned short      version;
    const char *        connectparms;
    const char *        _unused1;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        statusrtn;
    a_bit_field         quiet          : 1;
    a_bit_field         _unused2       : 1;
    const char *        _unused3;
    a_bit_field         _unused4       : 1;
    a_bit_field         jconnect       : 1;
    a_bit_field         _unused5       : 1;
    a_bit_field         _unused6       : 1;
    const char *        _unused7;
} an_upgrade_db;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号
connectparms	データベース接続に必要なパラメータ。次のような接続文字列のフォームをとる。 "UID=DBA;PWD=sql;DBF=samples-dir¥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。次に例を示します。 "START=d:¥sqlany10¥win32¥dbeng10.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=samples-dir¥demo.db;START=d:¥sqlany10¥win32¥dbeng10.exe" 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。
errorrtn	エラー・メッセージ処理コールバック・ルーチン
msgrtn	情報メッセージ処理コールバック・ルーチン
statusrtn	ステータス・メッセージ処理コールバック・ルーチン
quiet	操作中にメッセージを出力する (0) またはしない (1)
jconnect	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する。

参照

- 「DBUpgrade 関数」 773 ページ
- 「コールバック関数の使い方」 759 ページ.

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 {
    unsigned short      version;
    const char *        connectparms;
    const char *        _unused1;
    p_name              tables;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        statusrtn;
    a_bit_field         quiet : 1;
    a_bit_field         index : 1;
    a_validate_type     type;
} a_validate_db;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号
connectparms	データベース接続に必要なパラメータ。次のような接続文字列のフォームをとる。 "UID=DBA;PWD=sql;DBF=samples-dir¥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。次に例を示します。 "START=d:¥sqlany10¥win32¥dbeng10.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=samples-dir¥demo.db;START=d:¥sqlany10¥win32¥dbeng10.exe" 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。
tables	テーブル名のリンク・リストへのポインタ
errorrtn	エラー・メッセージ処理コールバック・ルーチン
msgrtn	情報メッセージ処理コールバック・ルーチン
statusrtn	ステータス・メッセージ処理コールバック・ルーチン
quiet	操作中にメッセージを出力する (0) またはしない (1)

メンバ	説明
index	インデックスを検証
type	「a_validate_type 列挙」 817 ページを参照。

参照

- 「DBValidate 関数」 774 ページ
- 「a_name 構造体」 786 ページ
- 「a_validate_type 列挙」 817 ページ
- コールバック関数の詳細については、「コールバック関数の使い方」 759 ページ を参照してください。

DBTools 列挙型

この項では、DBTools ライブラリが使用する列挙型について説明します。列挙はアルファベット順に示します。

ブランク埋め込み列挙

blank_pad の値を指定するために 「a_create_db 構造体」 779 ページで使用されます。

構文

```
enum {
    NO_BLANK_PADDING,
    BLANK_PADDING
};
```

パラメータ

値	説明
NO_BLANK_PADDING	ブランク埋め込みを使用しない。
BLANK_PADDING	ブランク埋め込みを使用する。

参照

- 「a_create_db 構造体」 779 ページ

a_chkpt_log_type 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」 775 ページで使用されます。

構文

```
typedef enum {
    BACKUP_CHKPT_LOG_COPY = 0,
    BACKUP_CHKPT_LOG_NOCOPY,
    BACKUP_CHKPT_LOG_RECOVER,
    BACKUP_CHKPT_LOG_AUTO,
    BACKUP_CHKPT_LOG_DEFAULT
} a_chkpt_log_type;
```

パラメータ

値	説明
BACKUP_CHKPT_LOG_COPY	WITH CHECKPOINT LOG COPY 句の生成に使用する。
BACKUP_CHKPT_LOG_NOCOPY	WITH CHECKPOINT LOG COPY 句の生成に使用する。

値	説明
BACKUP_CHKPT_LOG_RECOVER	WITH CHECKPOINT LOG RECOVER 句の生成に使用する。
BACKUP_CHKPT_LOG_AUTO	WITH CHECKPOINT LOG AUTO 句の生成に使用する。
BACKUP_CHKPT_LOG_DEFAULT	WITH CHECKPOINT 句を省略するのに使用する。

参照

- 「a_backup_db 構造体」 775 ページ

a_db_version 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」 784 ページで使用されます。

構文

```
enum {
      VERSION_UNKNOWN,
      VERSION_PRE_10,
      VERSION_10
  };
```

パラメータ

値	説明
VERSION_UNKNOWN	データベースを最初に作成した SQL Anywhere のバージョンを決定できない。
VERSION_PRE_10	データベースは、バージョン 10 より前の SQL Anywhere を使用して作成された。
VERSION_10	データベースは、SQL Anywhere 10 を使用して作成された。

参照

- 「DBCreatedVersion 関数」 766 ページ
- 「a_db_version_info 構造体」 784 ページ

データベース・サイズ単位列挙

db_size_unit の値を指定するために、「a_create_db 構造体」 779 ページで使用されます。

構文

```
enum {
   DBSP_UNIT_NONE,
```



```
    DBSP_UNIT_PAGES,
    DBSP_UNIT_BYTES,
    DBSP_UNIT_KILOBYTES,
    DBSP_UNIT_MEGABYTES,
    DBSP_UNIT_GIGABYTES,
    DBSP_UNIT_TERABYTES
};
```

パラメータ

値	説明
DBSP_UNIT_NONE	単位を指定しない。
DBSP_UNIT_PAGES	サイズをページ単位で指定する。
DBSP_UNIT_BYTES	サイズをバイト単位で指定する。
DBSP_UNIT_KILOBYTES	サイズをキロバイト単位で指定する。
DBSP_UNIT_MEGABYTES	サイズをメガバイト単位で指定する。
DBSP_UNIT_GIGAYTES	サイズをギガバイト単位で指定する。
DBSP_UNIT_TERABYTES	サイズをテラバイト単位で指定する。

参照

- 「a_create_db 構造体」 779 ページ

dbtran_userlist_type 列挙

「a_translate_log 構造体」 802 ページで使用される、ユーザ・リストのタイプ。

構文

```
typedef enum dbtran_userlist_type {
    DBTRAN_INCLUDE_ALL,
    DBTRAN_INCLUDE_SOME,
    DBTRAN_EXCLUDE_SOME
} dbtran_userlist_type;
```

パラメータ

値	説明
DBTRAN_INCLUDE_ALL	全ユーザの操作を含む。
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む。
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する。

参照

- 「a_translate_log 構造体」 802 ページ

dbunload type 列挙

「an_unload_db 構造体」 807 ページで使用される、実行中のアンロードのタイプ。

構文

```
enum {
    UNLOAD_ALL,
    UNLOAD_DATA_ONLY,
    UNLOAD_NO_DATA,
    UNLOAD_NO_DATA_FULL_SCRIPT
};
```

パラメータ

値	説明
UNLOAD_ALL	データとスキーマの両方をアンロードする。
UNLOAD_DATA_ONLY	データをアンロード。スキーマはアンロードしない。dbunload -d オプションと同等。
UNLOAD_NO_DATA	データなし。スキーマのみをアンロードする。dbunload -n オプションと同等。
UNLOAD_NO_DATA_FULL_SCRIPT	データなし。再ロード・スクリプトに LOAD/INPUT 文を追加する。dbunload -nl オプションと同等。

参照

- 「an_unload_db 構造体」 807 ページ

a_validate_type 列挙

「a_validate_db 構造体」 812 ページで使用される、実行中の検証のタイプ。

構文

```
typedef enum {
    VALIDATE_NORMAL = 0,
    VALIDATE_DATA,
    VALIDATE_INDEX,
    VALIDATE_EXPRESS,
    VALIDATE_FULL,
    VALIDATE_CHECKSUM,
    VALIDATE_DATABASE
} a_validate_type;
```

パラメータ

値	説明
VALIDATE_NORMAL	デフォルトのチェックのみで検証
VALIDATE_DATA	(旧式)
VALIDATE_INDEX	(旧式)
VALIDATE_EXPRESS	エクスプレス・チェックで検証する。dbvalid -fx オプションと同等。
VALIDATE_FULL	(旧式)
VALIDATE_CHECKSUM	データベース・チェックサムを検証する。dbvalid -s オプションと同等。
VALIDATE_DATABASE	データベースを検証する。dbvalid -d オプションと同等。

参照

- 「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』

冗長列挙

出力のボリュームを指定します。

構文

```
enum {
    VB_QUIET,
    VB_NORMAL,
    VB_VERBOSE
};
```

パラメータ

値	説明
VB_QUIET	出力なし
VB_NORMAL	通常の出力量
VB_VERBOSE	冗長出力。デバッグ用。

参照

- 「a_create_db 構造体」 779 ページ
- 「an_unload_db 構造体」 807 ページ

第 20 章

終了コード

目次

ソフトウェア・コンポーネントの終了コード

データベース・ツールはすべて DLL のエントリ・ポイントとして提供されます。エントリ・ポイントで使用する終了コードは、次のとおりです。SQL Anywhere のユーティリティ (dbbackup、dbspawn、dbeng10 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
16	EXIT_FILE_IN_USE	ファイルが使用中
17	EXIT_FATAL_ERROR	致命的なエラーまたはアサーションが発生した
255	EXIT_USAGE	コマンド・ラインで無効なパラメータ

これらの終了コードは、*install-dir¥h¥sqldef.h* ファイルに含まれています。

パート VI. SQL Anywhere の配備

パート VI では、SQL Anywhere での配備方法について説明します。

第 21 章

データベースとアプリケーションの配備

目次

配備の概要

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。

ライセンス契約の確認
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。

この章では、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。

この章の大半は、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、配備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、「配備ウィザードの使用」 830 ページと「サイレント・インストールを使用した配備」 832 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、「サイレント・インストールを使用した配備」 832 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、この章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ディレクトリに置かれています。たとえば、インストール・ディレクトリの *win32* サブディレクトリには、32 ビット Windows オペレーティング・システムのサーバの実行に必要なファイルが含まれています。

どのオプションを選択する場合でも、ライセンス契約の条項に違反しないでください。

インストール・ディレクトリとファイル名の知識

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。配備されるファイルは、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。

これは、実際には、各 Windows 上の単一のディレクトリにほとんどのファイルを置くということです。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 *win32* サブディレクトリ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、次のようなディレクトリ構造になります。

ディレクトリ	内容
/opt/sqlanywhere10/bin	実行ファイル、ライセンス・ファイル
/opt/sqlanywhere10/lib	共有オブジェクトと共有ライブラリ
/opt/sqlanywhere10/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/sqlanywhere10* であり、*/opt/sqlanywhere10* 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/SQLAnywhere10/System* であり、*/opt/sqlanywhere10* ではありません。

- **ファイルのサフィックス**　この章の表では、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。

  HP-UX の場合、サフィックスは *.sl.1* または *.so.1* です。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、*libdblib10.so* は同じディレクトリ内のファイル *libdblib10.so.1* へのシンボリック・リンクです。HP-UX の場合、シンボリック・リンクのサフィックスは *.sl* です。

更新がリリースされているため、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* より大きい場合があります。

- **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追加文字 *_r* が付けられます。たとえば、*libdblib10.so.1* の他に *libdblib10_r.so.1* というファイルが存在します。この場合、スレッド・アプリケーションは名前のサフィックスが *_r* である共有オブジェクトにリンクされる必要があるのに対して、非スレッド・アプリケーションは名前のサフィックスが *_r* ではない共有オブジェクトにリンクされる必要があります。ときには、共有オブジェクトの 3 番目の形式として、ファイルのサフィックスの前に *_n* が付けられています。これは、非スレッド・アプリケーションで使用される共有オブジェクトのバージョンです。
- **文字セット変換** データベース・サーバの文字セット変換を使用する場合、次のファイルが必要です。
  - *libdbicu10.so.1*
  - *libdbicu10_r.so.1*
  - *libdbicudt10.so.1*
  - *sqlany.cvf*
- **環境変数**
  Linux や UNIX 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/sqlanywhere10/bin* 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、SQLANY10、SQLANYSH10 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 *dbeng10.exe* は Windows 用のバージョン 10 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、*dblgen10.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 規格 639 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	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
.nlm	Novell NetWare	NetWare ロード・モジュール
.res	NetWare、Linux/UNIX、Mac OS X	非 Windows 環境用の言語リソース・ファイル
.so, .sl,.a	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 の同等品。
.bundle, .dylib	Mac OS X	共有オブジェクト・ファイル。Windows DLL の同等品。

データベース・ファイル名

SQL Anywhere データベースは、次の 2 つの要素で構成されます。

- **データベース・ファイル**　系統立てて管理されたフォーマットで情報を保存するために使用します。デフォルトで、このファイルは拡張子として *.db* を使います。その他の dbspace ファイルも存在する場合があります。これらのファイルには任意のファイル拡張子が付いているか、または拡張子がない場合があります。
- **トランザクション・ログ・ファイル**　データベース・ファイルに保存されているデータに加えられた変更をすべて記録するために使用します。デフォルトでは、ファイル拡張子には *.log*

を使用します。トランザクション・ログ・ファイルが存在せずログ・ファイルを使用するように指定されている場合は、SQL Anywhere がこのファイルを生成します。ミラーリングされたトランザクション・ログには、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。

配備ウィザードの使用

SQL Anywhere の配備ウィザードは、SQL Anywhere for Windows の配備に推奨されるツールです。配備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
- Mobile Link サーバ、クライアント、モニタ、暗号化
- Interactive SQL や Sybase Central などの管理ツール

配備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ストレージ・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。また、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。

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。

♦ **配備を作成するには、次の手順に従います。**

1. 配備ウィザードを起動します。

   - [スタート] − [プログラム ファイル] − [SQL Anywhere 10] − [SQL Anywhere Deployment ウィザード] を選択します。

     または

   - SQL Anywhere インストール環境の *deployment* サブディレクトリから、*setup.exe* を実行します。

2. ウィザードの指示に従います。

サイレント・インストールを使用した配備

サイレント・インストールは、ユーザの入力は必要とせず、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 のインストールがサイレントになるように、ユーザ自身のセットアップ・プログラムから SQL Anywhere InstallShield インストーラを呼び出すことができます。

「配備の種類」 824 ページで説明されているどの配備モデルに対しても、サイレント・インストールを使用できます。Mobile Link サーバを配備する場合にもサイレント・インストールを使用できます。

サイレント・インストールの作成

サイレント・インストールで使用されるインストール・オプションは、「**応答ファイル**」から取得されます。-r オプションを使用して、インストール・ディスクから SQL Anywhere の *setup* プログラムを実行すると、応答ファイルが作成されます。-s オプションを使用して *setup* プログラムを実行すると、サイレント・インストールが実行されます。

[参照] ボタンは使用しない
サイレント・インストールを作成する場合には、[参照] ボタンを使用しないでください。[参照] ボタンの記録は確実ではありません。

♦ **サイレント・インストールを作成するには、次の手順に従います。**

1. 既存の SQL Anywhere インストールを削除します。インストールを削除しないと、応答ファイルに記録される内容に影響する場合があります。
2. システム・コマンド・プロンプトを開き、インストール・イメージ (*setup.exe*、*setup.ins* など) が入っているディレクトリに移動します。
3. Record モードを使って、ソフトウェアをインストールします。

   次のコマンドを入力します。

   ```
   setup -r
   ```

   このコマンドは、SQL Anywhere のインストーラを実行し、ユーザの選択に基づいて応答ファイルを作成します。応答ファイルは *setup.iss* という名前で、*Windows* ディレクトリに置かれます。このファイルには、ユーザがインストール中にダイアログ・ボックスで入力した応答が入っています。

   Record モードで実行した場合、再起動が必要であってもインストール・プログラムはオペレーティング・システムの再起動を要求してきません。
4. ユーザ・アプリケーションといっしょに使うために、エンド・ユーザのコンピュータに SQL Anywhere を配備する場合には、適切なオプションや設定を使って SQL Anywhere をインストールしてください。以降のサイレント・インストールでは、選択したパスを上書きできます。

サイレント・インストールの実行

-s オプションを使って、ユーザ自身のインストール・プログラムで SQL Anywhere のサイレント・インストールを呼び出してください。この項では、サイレント・インストールの使用方法について説明します。

♦ **サイレント・インストールを使用するには、次の手順に従います。**

1. インストール・プロシージャに、SQL Anywhere のサイレント・インストールを呼び出すコマンドを追加します。

   応答ファイルは通常、書き込み禁止のインストール・イメージ・ディレクトリ (CD に収録されたインストール・イメージ・ディレクトリなど) にあります。応答ファイルとログ・ファイルの場所を指定して、サイレント・インストールを実行できます。ログ・ファイルの場所には、書き込み可能なメディアを指定してください。応答ファイルの場所は **-f1** オプションを使用して指定します。次のコマンド・ラインでは、**f1** と引用符の間にスペースを入れないようにします。**-f2** オプションを使用して、ログ・ファイルの場所を指定することもできます。この場合も、**f2** と引用符との間にはスペースを入れないようにします。InstallShield を使用する場合は、ファイルへのフル・パスを指定する必要があります。

   次の例では、インストール・イメージ・ディレクトリがドライブ *d:* の CD の *install* ディレクトリにあることを前提としています。ログ・ファイルは *Windows* ディレクトリに置かれます。

   ```
   d:¥install¥setup -s -f1"d:¥install¥setup.iss" -f2"c:¥windows¥setup.log"
   ```

   別の InstallShield スクリプトからインストールを呼び出すには、次のように行います。

   ```
   DoInstall( "sql_anywhere_install_image_path¥setup.inx",
     "-s -f1"d:¥install¥setup.iss" -f2"c:¥windows¥setup.log"", WAIT );
   ```

   オプションを使って、SQL Anywhere インストール・ディレクトリと共有ディレクトリのパスの選択を両方とも上書きできます。

   ```
   setup TARGET_DIR=dirname SHARED_DIR=shared_dir
   -s -f1"d:¥install¥setup.iss" -f2"c:¥windows¥setup.log"
   ```

   TARGET_DIR 引数と SHARED_DIR 引数は、他のオプションの前にくるようにしてください。

2. 対象となるコンピュータの再起動が必要かどうか確認します。

   セットアップによって、対象ディレクトリに *silent.log* というファイルが作成されます。このファイルには、次の行が入っている **ResponseResult** という単一のセクションがあります。

   ```
   Reboot=value
   ```

   この行は、インストールを完了するために対象コンピュータの再起動が必要かどうかを示す、0 または 1 の値を持っています。値の示す内容は、次のとおりです。

   - **Reboot=0**　再起動は必要ありません。
   - **Reboot=1**　インストール中に BATCH_INSTALL オプションが設定されました。対象コンピュータの再起動が必要です。サイレント・インストールを呼び出したインストー

ル・プロシージャで Reboot エントリをチェックし、必要に応じて対象コンピュータを再起動します。

3. セットアップが正常に完了したことをチェックします。

セットアップでは、*setup.log* という名前のログ・ファイルがデフォルトで作成されます。ログ・ファイルには、サイレント・インストールのレポートが入っています。このファイルのセクションは **ResponseResult** と呼ばれ、次の行が入っています。

```
ResultCode=value
```

この行は、インストールが正常に終了したかどうかを示します。ResultCode が 0 以外の場合、インストール中にエラーが発生したことを示します。以下は結果コードの一部です。

ResultCode 値	説明
0	成功
-1	一般的なエラー
-2	無効なモード
-3	必要なデータが Setup.iss ファイルにない
-4	メモリ不足
-5	ファイルが存在しない
-6	応答ファイルに書き込みできない
-7	ログ・ファイルに書き込みできない
-8	InstallShield サイレント・インストールの応答ファイルへのパスが無効
-9	リスト・タイプが無効 (文字列または数字)
-10	データ型が無効
-11	セットアップ時に未知のエラーが発生
-12	ダイアログ・ボックスが機能しない
-51	指定されたフォルダを作成できない
-52	指定されたファイルまたはフォルダにアクセスできない
-53	無効なオプションが選択された

結果コードの詳細については、InstallShield のマニュアルを参照してください。

クライアント・アプリケーションの配備

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、配備ウィザードを使用することです。詳細については、「配備ウィザードの使用」 830 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、配備ウィザードを使用することです。詳細については、「配備ウィザードの使用」 830 ページを参照してください。

独自のインストール環境を構築する場合を考慮して、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 .NET クライアント・コンピュータには、次のものが必要です。

- **.NET が動作するインストール環境**　Microsoft .NET アセンブリとファイルの再配布に関する指示については、Microsoft から入手できます。ここでは、その詳細は説明しません。
- **SQL Anywhere .NET データ・プロバイダ**　次の表には、SQL Anywhere .NET データ・プロバイダが動作するの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、Windows アセンブリは SQL Anywhere インストール・ディレクトリの *Assembly* サブディレクトリに置かれます。たとえば、*Assembly¥v1* (.NET 1.0 の場合) や *Assembly¥v2* (.NET 2.0 の場合) です。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サブディレクトリに置かれます (例：*win32*、*x64*)。

  .NET Compact Framework バージョン 2.0 用 Windows CE アセンブリは、サポートされる Windows CE ハードウェア・プラットフォームごとに *ce¥Assembly¥v2* に配置されます。

  .NET Compact Framework バージョン 1.*x* 用 Windows CE アセンブリは、サポートされる Windows CE ハードウェア・プラットフォームごとに *ce¥Assembly¥v1* に配置されます。

説明	Windows	Windows CE
.NET ドライバ・ファイル	*iAnywhere.Data.SQLAnywhere.dll*	*iAnywhere.Data.SQLAnywhere.dll*
.NET Global Assembly Cache	なし	*iAnywhere.Data.SQLAnywhere.gac*
言語リソース・ライブラリ	*dblg[en]10.dll*	*dblg[en]10.dll*
[接続] ダイアログ	*dbcon10.dll*	*N/A*

SQL Anywhere .NET 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」 145 ページを参照してください。

OLE DB クライアントと ADO クライアントの配備

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、配備ウィザードを使用することです。詳細については、「配備ウィザードの使用」 830 ページを参照してください。

独自のインストール環境を構築する場合を考慮して、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・サブディレクトリに置かれます (例：*win32*、*x64*)。Windows の場合、プロバイダ DLL が 2 つあります。2 つ目の DLL (*dboledba10*) は、スキーマ・サポートの提供に使用される支援 DLL です。Windows CE の場合、2 つ目の DLL はありません。

説明	Windows	Windows CE
OLE DB ドライバ・ファイル	*dboledb10.dll*	*dboledb10.dll*
OLE DB ドライバ・ファイル	*dboledba10.dll*	なし
言語リソース・ライブラリ	*dblg[en]10.dll*	*dblg[en]10.dll*
[接続] ダイアログ	*dbcon10.dll*	なし

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。これらのエントリは、Windows 上では regsvr32 ユーティリティ、Windows CE 上では regsvrce ユーティリティで、DLL を自己登録することで作成できます。

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 アプリケーションを Windows CE デバイスに配備する場合は、Microsoft の ADOCE バージョン 3.1 以降も含めるようにしてください。最低限必要なファイルは次のとおりです。

```
adoce31.dll
adocedb31.dll
adoceoledb31.dll
msdadc.dll
msdaer.dll
msdaerXX.dll (where XX is the 2 letter country code, EN for English)
```

Microsoft Data Access Components (MDAC) ファイル *msdadc.dll* と *msdaer.dll* をデバイスに登録してください。

詳細については、「Windows CE データベースの作成」 『SQL Anywhere サーバ - データベース管理』を参照してください。

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 10 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.10"), &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]
@="{41dfe9ef-db91-11d2-8c43-006008d26a6f}"

[HKEY_CLASSES_ROOT¥SAOLEDB¥CurVer]
@="SAOLEDB.10"
```

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¥
{41dfe9ef-db91-11d2-8c43-006008d26a6f}¥
InprocServer32]

@="c:¥¥sa10¥¥x64¥¥dboledb10.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う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。

次の演習では、3 つのことを行います。まず、ユニークな GUID セットを生成します。次に、ユニークなプロバイダ名を指定します。最後に、ユニークな DLL 名を指定します。この 3 つの手順を踏むことで、アプリケーションとともに配備できるユニークな OLE DB プロバイダが作成されます。

ここでは、OLE DB プロバイダのカスタム・バージョンを作成する手順を説明します。

♦ OLE DB プロバイダをカスタマイズするには、次の手順に従います。

1. 後に示すサンプル登録ファイルのコピーを作成します。登録ファイルは非常に長いので、手順の後に示しています。ファイル名には、サフィックスとして *.reg* を付けます。レジストリ値の名前は大文字と小文字を区別します。
2. 4 つの連続する UUID (GUID) を作成します。

   Microsoft Visual Studio の uuidgen ツールを使用します。

   ```
   uuidgen -n4 -s -x >oledbguids.txt
   ```
3. 4 つの UUID または GUID は、順番に次のように割り当てます。
   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 CE の場合の配備では使用されません。

この 4 つが連番であることが重要です (uuidgen コマンド・ラインで -x を指定することでそのようになります)。各 GUID は次のようになります。

名前	GUID
GUID1	41dfe9eb-db92-11d2-8c43-006008d26a6f
GUID2	41dfe9ec-db92-11d2-8c43-006008d26a6f
GUID3	41dfe9ed-db92-11d2-8c43-006008d26a6f
GUID4	41dfe9ee-db92-11d2-8c43-006008d26a6f

増分されているのは GUID の最初の部分です (この例では 41dfe9eb)。

4. 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 をすべて対応する 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 41dfe9eb-db92-11d2-8c43-006008d26a6f に置き換えます)。

5. Provider 名を決定します。ここで決定する名前を、アプリケーションで接続文字列に使用します (例：Provider=SQLAny)。次の名前は使用しないでください。これらは、SQL Anywhere によって使用されています。

バージョン 10	バージョン 9 以前
SAOLEDB	ASAProv
SAErrorLookup	ASAErrorLookup
SAEnum	ASAEnum
SAOLEDBA	ASAProvA

6. エディタの検索／置換機能を使用して、出現する文字列 SQLAny をすべて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

7. SQL Anywhere プロバイダ DLL (*dboledb10.dll* と *dboledba10.dll*) のコピーを別の名前で作成します。Windows CE の場合、*dboledba10.dll* はありません。

```
copy dboledb10.dll myoledb10.dll
copy dboledba10.dll myoledba10.dll
```

スクリプトにより、選択した DLL 名に基づく特別なレジストリ・キーが作成されます。ここでは、名前が標準の DLL 名 (*dboledb10.dll*、*dboledba10.dll* など) と異なることが重要です。プロバイダ DLL 名を *myoledb10* 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。プロバイダ支援 DLL の場合も同様です。DLL 名を *myoledba10* 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
myoledb10.dll	HKEY_CLASSES_ROOT¥myoledb10
myoledba10.dll	HKEY_CLASSES_ROOT¥myoledba10
acmeOledb.dll	HKEY_CLASSES_ROOT¥acmeOledb
acmeOledba.dll	HKEY_CLASSES_ROOT¥acmeOledba
SAcustom.dll	HKEY_CLASSES_ROOT¥SAcustom
SAcustomA.dll	HKEY_CLASSES_ROOT¥SAcustomA

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 *myoledb10* や *myoledba10* をすべて選択した 2 つの DLL 名に変更します。

9. エディタの検索／置換機能を使用して、レジストリ・スクリプトに出現する *d:¥¥mypath¥¥win32¥¥* をすべて DLL のインストール・ロケーションに変更します。1 つの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。

10. 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。

11. 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。

変更するレジストリ・スクリプトを次に示します。

```
REGEDIT4
; Special registry entries for a private OLE DB provider.

[HKEY_CLASSES_ROOT¥myoledb10]
@="Custom SQL Anywhere OLE DB Provider 10.0"

[HKEY_CLASSES_ROOT¥myoledb10¥Clsid]
@="{GUID1}"

; Data1 of the following GUID must be 3 greater than the
; previous, for example, 41dfe9eb + 3 => 41dfe9ee.
```

```
[HKEY_CLASSES_ROOT¥myoledba10]
@="Custom SQL Anywhere OLE DB Provider 10.0"

[HKEY_CLASSES_ROOT¥myoledba10¥Clsid]
@="{GUID4}"

; Current version (or version independent prog ID)
; entries (what you get when you have "SQLAny"
; instead of "SQLAny.10")

[HKEY_CLASSES_ROOT¥SQLAny]
@="SQL Anywhere OLE DB Provider"

[HKEY_CLASSES_ROOT¥SQLAny¥Clsid]
@="{GUID1}"

[HKEY_CLASSES_ROOT¥SQLAny¥CurVer]
@="SQLAny.10"

[HKEY_CLASSES_ROOT¥SQLAnyEnum]
@="SQL Anywhere OLE DB Provider Enumerator"

[HKEY_CLASSES_ROOT¥SQLAnyEnum¥Clsid]
@="{GUID2}"

[HKEY_CLASSES_ROOT¥SQLAnyEnum¥CurVer]
@="SQLAnyEnum.10"

[HKEY_CLASSES_ROOT¥SQLAnyErrorLookup]
@="SQL Anywhere OLE DB Provider Extended Error Support"

[HKEY_CLASSES_ROOT¥SQLAnyErrorLookup¥Clsid]
@="{GUID3}"

[HKEY_CLASSES_ROOT¥SQLAnyErrorLookup¥CurVer]
@="SQLAnyErrorLookup.10"

[HKEY_CLASSES_ROOT¥SQLAnyA]
@="SQL Anywhere OLE DB Provider Assist"

[HKEY_CLASSES_ROOT¥SQLAnyA¥Clsid]
@="{GUID4}"

[HKEY_CLASSES_ROOT¥SQLAnyA¥CurVer]
@="SQLAnyA.10"

; Standard entries (Provider=SQLAny.10)

[HKEY_CLASSES_ROOT¥SQLAny.10]
@="Sybase SQL Anywhere OLE DB Provider 10.0"

[HKEY_CLASSES_ROOT¥SQLAny.10¥Clsid]
@="{GUID1}"

[HKEY_CLASSES_ROOT¥SQLAnyEnum.10]
@="Sybase SQL Anywhere OLE DB Provider Enumerator 10.0"

[HKEY_CLASSES_ROOT¥SQLAnyEnum.10¥Clsid]
@="{GUID2}"

[HKEY_CLASSES_ROOT¥SQLAnyErrorLookup.10]
@="Sybase SQL Anywhere OLE DB Provider Extended Error Support 10.0"

[HKEY_CLASSES_ROOT¥SQLAnyErrorLookup.10¥Clsid]
```

```
@="{GUID3}"

[HKEY_CLASSES_ROOT¥SQLAnyA.10]
@="Sybase SQL Anywhere OLE DB Provider Assist 10.0"

[HKEY_CLASSES_ROOT¥SQLAnyA.10¥Clsid]
@="{GUID4}"

; SQLAny (Provider=SQLAny.10)

[HKEY_CLASSES_ROOT¥CLSID¥{GUID1}]
@="SQLAny.10"
"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¥¥win32¥¥myoledb10.dll"
"ThreadingModel"="Both"

[HKEY_CLASSES_ROOT¥CLSID¥{GUID1}¥OLE DB Provider]
@="Sybase SQL Anywhere OLE DB Provider 10.0"

[HKEY_CLASSES_ROOT¥CLSID¥{GUID1}¥ProgID]
@="SQLAny.10"

[HKEY_CLASSES_ROOT¥CLSID¥{GUID1}¥VersionIndependentProgID]
@="SQLAny"

; SQLAnyErrorLookup

[HKEY_CLASSES_ROOT¥CLSID¥{GUID3}]
@="Sybase SQL Anywhere OLE DB Provider Error Lookup 10.0"
@="SQLAnyErrorLookup.10"

[HKEY_CLASSES_ROOT¥CLSID¥{GUID3}¥InprocServer32]
@="d:¥¥mypath¥¥win32¥¥myoledb10.dll"
"ThreadingModel"="Both"

[HKEY_CLASSES_ROOT¥CLSID¥{GUID3}¥ProgID]
@="SQLAnyErrorLookup.10"

[HKEY_CLASSES_ROOT¥CLSID¥{GUID3}¥VersionIndependentProgID]
@="SQLAnyErrorLookup"

; SQLAnyEnum

[HKEY_CLASSES_ROOT¥CLSID¥{GUID2}]
@="SQLAnyEnum.10"

[HKEY_CLASSES_ROOT¥CLSID¥{GUID2}¥InprocServer32]
@="d:¥¥mypath¥¥win32¥¥myoledb10.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.10"
```

```
[HKEY_CLASSES_ROOT¥CLSID¥{GUID2}¥VersionIndependentProgID]
@="SQLAnyEnum"

; SQLAnyA

[HKEY_CLASSES_ROOT¥CLSID¥{GUID4}]
@="SQLAnyA.10"

[HKEY_CLASSES_ROOT¥CLSID¥{GUID4}¥InprocServer32]
@="d:¥¥mypath¥¥win32¥¥myoledba10.dll"
"ThreadingModel"="Both"

[HKEY_CLASSES_ROOT¥CLSID¥{GUID4}¥ProgID]
@="SQLAnyA.10"

[HKEY_CLASSES_ROOT¥CLSID¥{GUID4}¥VersionIndependentProgID]
@="SQLAnyA"
```

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、配備ウィザードを使用することです。詳細については、「配備ウィザードの使用」 830 ページを参照してください。

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。

- 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、UNIX 用の ODBC ドライバ・マネージャが用意されています。Windows CE 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。
- **SQL Anywhere の ODBC ドライバ**　ODBC クライアント・アプリケーションの配備に必要なファイルについては、次の ODBC ドライバに必要なファイルで説明します。

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・サブディレクトリに置かれます (例：*win32*、*x64*)。

UNIX 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	*dbodbc10.dll* *dbcon10.dll* *dbicu10.dll* *dbicudt10.dll* *dblg[en]10.dll*
Windows CE	*dbodbc10.dll* *dblg[en]10.dll*
Linux/Solaris	*libdbodbc10.so.1* *libdbodbc10_n.so.1* *libdbodm10.so.1* *libdbtasks10.so.1* *libdbicu10.so.1* *libdbicudt10.so.1* *dblg[en]10.res*
Linux/Solaris MT	*libdbodbc10.so.1* *libdbodbc10_r.so.1* *libdbodm10.so.1* *libdbtasks10_r.so.1* *libdbicu10_r.so.1* *libdbicudt10.so.1* *dblg[en]10.res*
HP-UX	*libdbodbc10.sl.1* *libdbodbc10_n.sl.1* *libdbodm10.sl.1* *libdbtasks10.sl.1* *libdbicu10.sl.1* *libdbicudt10.sl.1* *dblg[en]10.res*

プラットフォーム	必要なファイル
HP-UX MT	*libdbodbc10.sl.1* *libdbodbc10_r.sl.1* *libdbodm10.sl.1* *libdbtasks10_r.sl.1* *libdbicu10_r.sl.1* *libdbicudt10.sl.1* *dblg[en]10.res*
AIX	*libdbodbc10.so* *libdbodbc10_n.so* *libdbodm10.so* *libdbtasks10.so* *libdbicu10.so* *libdbicudt10.so* *dblg[en]10.res*
AIX MT	*libdbodbc10.so* *libdbodbc10_r.so* *libdbodm10.so* *libdbtasks10_r.so* *libdbicu10_r.so* *libdbicudt10..so* *dblg[en]10.res*
Mac OS X	*dbodbc10.bundle* *libdbodbc10.dylib* *libdbodbc10_n.dylib* *libdbodm10.dylib* *libdbtasks10.dylib* *libdbicu10.dylib* *libdbicudt10.dylib* *dblg[en]10.res*

プラットフォーム	必要なファイル
Mac OS X MT	*dbodbc10_r.bundle* *libdbodbc10.dylib* *libdbodbc10_r.dylib* *libdbodm10.dylib* *libdbtasks10_r.dylib* *libdbicu10_r.dylib* *libdbicudt10.dylib* *dblg[en]10.res*

注意

- UNIX 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。また、UNIX プラットフォームの場合は、これらのファイルへのリンクを作成してください。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。
- 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。UNIX の場合は、SQL Anywhere によってドライバ・マネージャが提供されます。このファイルには *libdbodm10* で始まる名前が付いています。
- 言語リソース・ライブラリ・ファイルも含めてください。上の表では、英語バージョン (en) になっています。サポートする言語に対応する言語リソース・ライブラリを配備してください。
- 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] ダイアログを表示する必要がある場合は、[接続] ダイアログのサポート・コード (*dbcon10.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、セットアップ・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。セットアップ・プログラムをエンド・ユーザ用に構築する場合は、同じ設定を行ってください。レジストリ値の名前は大文字と小文字を区別します。

レジストリ・エントリを調べる場合は、regedit ユーティリティを使用できます。

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 10
```

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	*install-dir¥win32 ¥dbodbc10.dll*
Setup	文字列	*install-dir¥win32 ¥dbodbc10.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 10	文字列	Installed

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。

  ODBC アドミニストレータを使用して [ユーザー 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

定ダイアログを表示します。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。

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

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。

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。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 接続を使用することができます。

ファイル・データ・ソースはレジストリには保持されないで、特別なディレクトリに保持されます。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。

データ・ソースのレジストリ・エントリ

各ユーザ・データ・ソースは、レジストリ・エントリによってシステムに識別されます。

一連のレジストリ値を特定のレジストリ・キーに入力する必要があります。ユーザ・データ・ソースの場合のキーを次に示します。

```
HKEY_CURRENT_USER¥
  SOFTWARE¥
    ODBC¥
      ODBC.INI¥
        userdatasourcename
```

システム・データ・ソースの場合のキーを次に示します。

```
HKEY_LOCAL_MACHINE¥
  SOFTWARE¥
    ODBC¥
      ODBC.INI¥
        systemdatasourcename
```

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。たとえば、SQL Anywhere 10 Demo ユーザのデータ・ソース名 (DSN) に対応する SQL Anywhere 10 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	*samples-dir¥demo.db*
Description	文字列	SQL Anywhere 10 サンプル・データベース
Driver	文字列	*install-dir¥win32¥dbodbc10.dll*

値の名前	値のタイプ	値データ
EngineName	文字列	demo10
PWD	文字列	sql
Start	文字列	*install-dir¥win32¥dbeng10.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、EngineName 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することができるため、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。

これらのエントリの *install-dir* は SQL Anywhere のインストール・ディレクトリです。Windows x64 ベース・システムでは、*win32* は *x64* になります。

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ユーザ・データ・ソースは、SQL Anywhere 10 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 10 Demo	文字列	SQL Anywhere 10

警告：ODBC の設定は簡単に表示されてしまう
ユーザ・データ・ソースの設定には、ユーザ ID とパスワードのように機密性のあるデータベース設定を含めることもできます。これらの設定は、レジストリに普通のテキストとして保存され、Windows レジストリ・エディタ *regedit.exe* または *regedt32.exe,* を使用して表示できます。これらのエディタは、Microsoft からオペレーティング・システムとともに提供されています。パスワードを暗号化したり、ユーザが接続するときにパスワードの入力を要求するように選択することもできます。

必須およびオプションの接続パラメータ

ODBC 設定文字列内のデータ・ソース名は次のようにして調べることができます。

```
DSN=UserDataSourceName
```

DSN パラメータを接続文字列に指定すると、Windows レジストリ内の現在のユーザ・データ・ソース定義が検索されたあとにシステム・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 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 (DSN に指定がない場合)
ODBC ドライバの名前とロケーションのみを含む	データベース・ファイル名 (DBF=) とデータベース・サーバ名 (ENG=) 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 (DSN または ODBC 接続文字列に指定がない場合)

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
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 (ENG=) 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)

ODBC の接続と設定の詳細については、次を参照してください。

- 「データベースへの接続」 『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、配備ウィザードを使用することです。詳細については、「配備ウィザードの使用」 830 ページを参照してください。

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。

Embedded SQL クライアント用ファイルのインストール

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	UNIX
インタフェース・ライブラリ	*dblib10.dll*	*libdblib10.so, libdbtasks10.so*
言語リソース・ライブラリ	*dblg[en]10.dll*	*dblg[en]10.res*
[接続] ダイアログ	*dbcon10.dll*	なし

注意

- 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 (*dbecc10.dll*、*dbfips10.dll*、または *dbrsa10.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。

  ODBC 情報の配備の詳細については、「ODBC クライアントの配備」 843 ページを参照してください。
- 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] ダイアログを表示する必要がある場合は、[接続] ダイアログのサポート (*dbcon10.dll*) が必要です。
- UNIX 上で動作するマルチスレッド・アプリケーションについては、*libdblib10_r.so* と *libdbtasks10_r.so* を使用してください。

接続情報

Embedded SQL 接続情報は、次のいず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 をインストールしてください。

JAVA_HOME 環境変数が定義されていない場合は、定義が必要になる場合があります。これにより、データベース・サーバは Java VM を見つけられるようになります。JAVA_HOME または JAVAHOME 環境変数は、Java VM のインストール時に作成されます。この環境変数は Java VM のルート・ディレクトリを示すようにしてください。

データベース・サーバは、次の順序で Java VM を検索します。

1. java_location データベース・オプションを確認します。
2. JAVA_HOME 環境変数を確認します。
3. JAVAHOME 環境変数を確認します。
4. パスを確認します。
5. 上記の検索が失敗すると、データベース・サーバは Java VM をロードできません。

JAVAHOME 環境変数を定義した場合、クライアント・コンピュータで JAVA_HOME がすでに定義されていると、サーバは JAVA_HOME を使用します。

Java Runtime Environment に加えて、各 JDBC クライアントには iAnywhere JDBC ドライバまたは jConnect が必要です。

iAnywhere JDBC ドライバを配備するには、次のファイルを配備します。

- *jodbc.jar* アプリケーションのクラスパス内に含めます。
- *dbjodbc10.dll* システム・パス内に含めます。UNIX 環境では、このファイルは共用ライブラリ (*libdbjodbc10.so*) です。
- ODBC ドライバ・ファイル。詳細については、「ODBC ドライバに必要なファイル」 843 ページを参照してください。

jConnect ソフトウェアと jConnect マニュアルのバージョンについては、http://www.sybase.com/products/informationmanagement/softwaredeveloperkit/jconnect を参照してください。

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。

URL の詳細については、「ドライバへの URL の指定」 499 ページを参照してください。

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 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。interfaces ファイルの詳細については、Open Client のマニュアルと 「Open Server の設定」 『SQL Anywhere サーバ - データベース管理』を参照してください。

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、配備ウィザードを使用することです。詳細については、「配備ウィザードの使用」 830 ページを参照してください。

管理ツールのシステム稼働条件については、「プラットフォーム別 SQL Anywhere 10.0.1 コンポーネント」の管理ツールの表を参照してください。

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 (dbeng10) 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。

Windows における InstallShield を使用しない管理ツールの配備

この項では、Windows コンピュータで InstallShield を使用せずに Interactive SQL (dbisql)、Sybase Central (SQL Anywhere、Mobile Link、QAnywhere、UltraLite 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。

この情報は、Windows CE 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 10.0.1 固有のもので、前後のバージョンには適用できません。

ライセンス契約の確認
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

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。

レジストリの変更の危険性
レジストリは、ユーザ自身の責任で変更してください。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する

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
- QAnywhere 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 1.5.0

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] メニュー – [システム環境設定] – [ソフトウェア・アップデート] を選択します。[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 1.5.0 がリストにない場合は、www.developer.apple.com/java/download/ を参照してください。

以下の項の手順は、これら 6 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する

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:¥SA10* 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
SA10	ルート・フォルダ。以下の手順では、*C:¥SA10* 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¥SQLAny10*)。
SA10¥java	Java プログラム JAR ファイルが保存されています。
SA10¥win32	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。
SA10¥Sun¥JavaHelp-1_1	JavaHelp ランタイム・ライブラリ
SA10¥Sun¥JAXB1.0	Java Architecture for XML Binding
SA10¥Sun¥jre150	Java Runtime Environment

x64
Java ベースの管理ツールは、32 ビット・アプリケーションです。64 ビット・バージョンはありません。32 ビット管理ツールは、Windows x64 ベース・プラットフォームに配備できます。

Itanium 64
Itanium (ia64) プラットフォームに配備できる Java ベースの管理ツールはありません。ただし、Java バージョンよりも機能が限定された Interactve SQL のネイティブ・バージョンがあります。「dbisqlc の配備」 874 ページを参照してください。

次の表は、各ソフトウェア・バンドルに必要な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、上述したディレクトリ構造にコピーします。通常は、すでにインストールされている SQL Anywhere のコピーからファイルを使用するようにしてください。

ファイル	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	QAnywhere 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
docs¥en¥htmlhelp¥dbma[en]10.chm	X	X	X	X	X	X
c:¥windows¥system32¥keyHH.exe [1]	X	X	X	X	X	X

ファイル	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	QAnywhere 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
java¥dbma[en]10.jar	X	X	X	X	X	X
java¥jodbc.jar	X	X	X	X	X	X
java¥JComponents1001.jar	X	X	X	X	X	X
java¥jlogon.jar	X	X	X	X	X	X
java¥SCEditor500.jar	X	X	X	X	X	X
java¥jsyblib500.jar	X	X	X	X	X	X
win32¥jsyblib500.dll	X	X	X	X	X	X
Sun¥JavaHelp-1_1¥jh.jar	X	X	X	X	X	X
Sun¥JAXB1.0¥...	X	X	X	X	X	X
Sun¥jre150¥...	X	X	X	X	X	X
win32¥dblib10.dll	X	X	X	X	X	X
win32¥dbjodbc10.dll	X	X	X	X	X	X
win32¥dbodbc10.dll	X	X	X	X	X	X
win32¥dbcon10.dll	X	X	X	X	X	X
win32¥dblg[en]10.dll	X	X	X	X	X	X
win32¥dbtool10.dll	X	X	X			
win32¥dbisql.exe	X					
win32¥dbisqlg.exe	X					
win32¥dbisql.cls	X					
java¥isql.jar	X	X	X		X	
Sybase Central 5.0.0¥win32¥scjview.exe		X	X	X	X	
Sybase Central 5.0.0¥win32¥scjview.cls		X	X	X	X	

1 Windows システム・ディレクトリの正確な名前は、ご使用のオペレーティング・システムによって異なります。

ファイル	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	QAnywhere 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
Sybase Central 5.0.0¥win32¥scjlg[en].dll		X	X	X	X	
Sybase Central 5.0.0¥scvw[en]500.chm		X	X	X	X	
Sybase Central 5.0.0¥scvw[en]500.jar		X	X	X	X	
Sybase Central 5.0.0¥sybasecentral500.jar		X	X	X	X	
java¥salib.jar		X	X	X	X	
java¥saplugin.jar		X				
java¥debugger.jar		X				
win32¥dbput10.dll		X	X			
java¥apache_files.txt			X	X		
java¥apache_license_1.1.txt			X	X		
java¥apache_license_2.0.txt			X	X		
java¥log4j.jar			X	X		
java¥mlplugin.jar			X			
java¥mldesign.jar			X			
java¥stax-api-1.0.jar			X			
java¥wstx-asl-2.0.5.jar			X			
java¥velocity.jar			X			
java¥velocity-dep.jar			X			
java¥qaplugin.jar				X		
java¥qaconnector.jar				X		
java¥mlstream.jar				X		
win32¥qaagent.exe				X		

ファイル	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	QAnywhere 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
win32¥dbicu10.dll				X		
win32¥dbicudt10.dll				X		
win32¥dbghelp.dll				X		
win32¥dbinit.exe				X		
java¥ulplugin.jar					X	
win32¥dbconsole.exe						X
java¥DBConsole.jar						X

上記のテーブルには、指定が **[en]** であるファイルの数が示されています。これらのファイルは、さまざまな言語版が用意されており、英語 (**en**) はそのうちの 1 つにすぎません。詳細については、外国語のメッセージとヘルプ・ファイルを参照してください。

上記のファイル・パスには、「...」で終わっているものもあります。これは、サブディレクトリ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。

管理ツールには JRE 1.5.0_10 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。インストールされた SQL Anywhere 10 の JRE ファイルを、*C:¥Program Files¥SQL Anywhere 10¥Sun¥jre150* ディレクトリからコピーしてください。サブディレクトリも含めて *jre150* ツリー全体をコピーします。

参照のために、*SQLAnywhere.jpr* ファイルには、Sybase Central の SQL Anywhere プラグインに必要なコンポーネント・ファイルのリストが含まれています。

MobiLink.jpr ファイルには、Sybase Central の Mobile Link プラグインに必要なコンポーネント・ファイルのリストが含まれています。

QAnywhere.jpr ファイルには、Sybase Central の QAnywhere プラグインに必要なコンポーネント・ファイルのリストが含まれています。QAnywhere プラグインを配備するには、dbinit が必要です。データベース・ツールの配備については、「データベース・ユーティリティの配備」 882 ページを参照してください。

Ultralite.jpr ファイルには、Sybase Central の Ultra Light プラグインに必要なコンポーネント・ファイルのリストが含まれています。

外国語のメッセージとヘルプ・ファイル

管理ツールのすべての表示テキストとオンライン・ヘルプは、英語からフランス語、ドイツ語、日本語、簡体字中国語に翻訳されています。各言語のリソースは別々のファイルに保存されています。英語のファイルの場合は、ファイル名に **en** が含まれています。フランス語のファイルも

同様な名前ですが、**en** の代わりに **fr** を使用します。ドイツ語のファイル名には **de**、日本語のファイル名には **ja**、中国語のファイル名には **zh** がそれぞれ含まれています。

異なる言語のサポートをインストールするには、それらの言語のリソースとヘルプ・ファイルを追加してください。翻訳済みのファイルは次のとおりです。

dbma??10.jar - SQL Anywhere Help Files
dbmaen10.jar English
dbmafr10.jar French
dbmade10.jar German
dbmaja10.jar Japanese
dbmazh10.jar Chinese

dblg??10.res - SQL Anywhere Messages Files
dblgen10.res English
dblgfr10.res French
dblgde10.res German
dblgja10.res Japanese
dblgzh10.res Chinese

scvw??500.jar - Sybase Central Help Files
scvwen500.jar English
scvwfr500.jar French
scvwde500.jar German
scvwja500.jar Japanese
scvwzh500.jar Chinese

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：管理ツールを Windows に登録する

管理ツールに対して以下のレジストリ値を設定します。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¥Sybase Central¥5.0.0** では次のように設定します。
  - **Location** Sybase Central ファイルを格納するフォルダへの完全に修飾されたパス (デフォルトで *C:¥Program Files¥Sybase Central 5.0.0*)。
  - **Shared Location** Sybase Central フォルダを格納するフォルダへの完全に修飾されたパス (デフォルトで *C:¥Program Files¥SQL Anywhere 10*)。
  - **Language** Sybase Central で使用する言語を表す 2 文字のコード。これは **en** (英語)、**fr** (フランス語)、**de** (ドイツ語)、**ja** (日本語)、**zh** (簡体字中国語) のいずれかです。
- **HKEY_LOCAL_MACHINE¥SOFTWARE¥Sybase¥SQL Anywhere¥10.0** 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 (デフォルトで *C:¥Program Files¥SQL Anywhere 10*)。
  - **Shared Location** Sybase Central フォルダを格納するフォルダへの完全に修飾されたパス (デフォルトで *C:¥Program Files¥SQL Anywhere 10*)。

- **Language**　SQL Anywhere で使用する言語を表す 2 文字のコード。これは **en** (英語)、**fr** (フランス語)、**de** (ドイツ語)、**ja** (日本語)、**zh** (簡体字中国語) のいずれかです。

Windows x64 ベース・システムでは、これらのレジストリ・エントリは、32 ビット・レジストリ (**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:¥SA10* 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¥5.0.0]
"Location"="c:¥¥sa10¥¥Sybase Central 5.0.0"
"Shared Location"="c:¥¥sa10"
"Language"="EN"

[HKEY_LOCAL_MACHINE¥SOFTWARE¥Sybase¥SQL Anywhere¥10.0]
"Location"="c:¥¥sa10"
"Shared Location"="c:¥¥sa10"
"Language"="EN"
```

.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・パスを更新する

管理ツールを実行するには、*.exe* ファイルと *.dll* ファイルが格納されているディレクトリをパスに含めます。*C:¥SA10¥win32* ディレクトリと *C:¥SA10¥Sybase Central 5.0.0¥win32* ディレクトリをシステム・パスに追加する必要があります。

Windows では、システム・パスは次のレジストリ・キーに保存されます。

```
HKEY_LOCAL_MACHINE¥
  SYSTEM¥
    CurrentControlSet¥
      Control¥
        Session Manager¥
          Environment¥
            Path
```

Interactive SQL または Sybase Central を配備する場合は、既存のパスの末尾に以下のディレクトリを追加します。

```
c:¥sa10¥win32;c:¥sa10¥Sybase Central 5.0.0¥win32
```

手順 5：Sybase Central プラグインを登録する

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 では、インストールされているプラグインをリストした設定ファイルが必要です。このファイルはインストーラによって作成されます。このファイルには、複数の JAR ファ

イルへのフル・パスが含まれますが、それらのパスはソフトウェアのインストール場所によって変わる可能性があることに注意してください。

ファイルは *.scRepository* と呼ばれます。Windows XP/2000 では、このファイルは *%allusersprofile%¥Sybase Central 5.0.0* フォルダにあります。Windows Vista では、このファイルは *%ProgramData%¥Sybase Central 5.0.0* フォルダにあります。これはプレーン・テキスト・ファイルで、Sybase Central でロードするプラグインに関するいくつかの基本情報が含まれています。

Windows Vista では、*.scRepository* ファイルが含まれるディレクトリに対して、すべてのユーザに読み込みアクセス権が必要です。これを行うには次のコマンドを実行します。手動で行う場合は、管理者のコマンド・プロンプト・ウィンドウを開きます ([コマンド プロンプト] を右クリックし、[管理者として実行] をクリックします)。

```
icacls "%ProgramData%¥Sybase Central 5.0.0" /grant everyone:F
```

SQL Anywhere のプロバイダ情報は、次のコマンドを使用してリポジトリ・ファイルに作成されます。

```
scjview.exe -register "C:¥Program Files¥SQL Anywhere 10¥java¥SQLAnywhere.jpr"
```

SQLAnywhere.jpr ファイルの内容は、次のようになります (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)。**AdditionalClasspath** の行は、*.jpr* ファイルでは 1 行に入力してください。

```
PluginName=SQL Anywhere 10
PluginId=sqlanywhere1000
PluginClass=com.sybase.asa.plugin.SAPlugin
PluginFile=C:¥Program Files¥SQL Anywhere 10¥java¥saplugin.jar
AdditionalClasspath=
    C:¥Program Files¥SQL Anywhere 10¥java¥isql.jar;
    C:¥Program Files¥SQL Anywhere 10¥java¥salib.jar;
    C:¥Program Files¥SQL Anywhere 10¥java¥JComponents1001.jar;
    C:¥Program Files¥SQL Anywhere 10¥java¥jlogon.jar;
    C:¥Program Files¥SQL Anywhere 10¥java¥debugger.jar;
    C:¥Program Files¥SQL Anywhere 10¥java¥jodbc.jar
ClassloaderId=SA1000
InitialLoadOrder=0
```

SQLAnywhere.jpr ファイルは、SQL Anywhere を最初にインストールしたときに SQL Anywhere インストール環境の *java* フォルダに作成されています。インストール処理の一部として作成が必要な *.jpr* ファイルのモデルとしてこのファイルを使用します。Mobile Link、QAnywhere、Ultra Light 用には、それぞれ *MobiLink.jpr*、*QAnywhere.jpr*、*UltraLite.jpr* という名前のファイルがあります。これらのファイルも *java* フォルダにあります。

上で説明した処理で作成された *.scRepository* 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: 5.0.0.3245
#  Fri Feb 23 13:09:14 EST 2007
#
SCRepositoryInfo/Version=4
#
Providers/sqlanywhere1000/Version=10.0.1.3390
Providers/sqlanywhere1000/UseClassloader=true
Providers/sqlanywhere1000/ClassloaderId=SA1000
```



```
Providers/sqlanywhere1000/Classpath=
   C:¥¥Program Files¥¥SQL Anywhere 10¥¥java¥¥saplugin.jar
Providers/sqlanywhere1000/Name=SQL Anywhere 10
Providers/sqlanywhere1000/AdditionalClasspath=
   C:¥¥Program Files¥¥SQL Anywhere 10¥¥java¥¥isql.jar;
   C:¥¥Program Files¥¥SQL Anywhere 10¥¥java¥¥salib.jar;
   C:¥¥Program Files¥¥SQL Anywhere 10¥¥java¥¥JComponents1001.jar;
   C:¥¥Program Files¥¥SQL Anywhere 10¥¥java¥¥jlogon.jar;
   C:¥¥Program Files¥¥SQL Anywhere 10¥¥java¥¥debugger.jar;
   C:¥¥Program Files¥¥SQL Anywhere 10¥¥java¥¥jodbc.jar
Providers/sqlanywhere1000/Provider=com.sybase.asa.plugin.SAPlugin
Providers/sqlanywhere1000/ProviderId=sqlanywhere1000
Providers/sqlanywhere1000/InitialLoadOrder=0
#
```

注意

- インストーラでは、上記の手法を使用して、これに類似したファイルを書き出します。必要な唯一の変更は、Classpath および AdditionalClasspath 行の JAR ファイルへの完全に修飾されたパスのみです。
- 上記の AdditionalClasspath 行は、右端で折り返し複数行になっています。*.scRepository* ファイルでは 1 行にしてください。
- *.scRepository* ファイルでは、円記号 (**¥**) は **¥¥** のエスケープ・シーケンスで示します。
- 最初の行は、*.scRepository* ファイルのバージョンを示します。
- 先頭に # がある行はコメントです。

手順 6 ：Sybase Central 用の接続プロファイルを作成する

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 10 Demo** の接続プロファイルが *.scRepository* 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 10 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 10 Demo/Name" "SQL Anywhere 10 Demo"
scjview -write "ConnectionProfiles/SQL Anywhere 10 Demo/FirstTimeStart" "false"
scjview -write "ConnectionProfiles/SQL Anywhere 10 Demo/Description" "Suitable Description"
scjview -write "ConnectionProfiles/SQL Anywhere 10 Demo/ProviderId" "sqlanywhere1000"
scjview -write "ConnectionProfiles/SQL Anywhere 10 Demo/Provider" "SQL Anywhere 10"
scjview -write "ConnectionProfiles/SQL Anywhere 10 Demo/Data/ConnectionProfileSettings" "DSN
¥eSQL^0020Anywhere^002010^0020Demo;UID¥eDBA;PWD¥e35c624d517fb"
scjview -write "ConnectionProfiles/SQL Anywhere 10 Demo/Data/ConnectionProfileName" "SQL
Anywhere 10 Demo"
```

接続プロファイルの文字列と値は、*.scRepository* 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、*.scRepository* 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 *.scRepository* 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: 5.0.0.3245
# Fri Feb 23 13:09:14 EST 2007
#
ConnectionProfiles/SQL Anywhere 10 Demo/Name=SQL Anywhere 10 Demo
ConnectionProfiles/SQL Anywhere 10 Demo/FirstTimeStart=false
ConnectionProfiles/SQL Anywhere 10 Demo/Description=Suitable Description
ConnectionProfiles/SQL Anywhere 10 Demo/ProviderId=sqlanywhere1000
ConnectionProfiles/SQL Anywhere 10 Demo/Provider=SQL Anywhere 10
ConnectionProfiles/SQL Anywhere 10 Demo/Data/ConnectionProfileSettings=
   DSN¥eSQL^0020Anywhere^002010^0020Demo;UID¥eDBA;PWD¥e35c624d517fb;
   UID¥eDBA;
   PWD¥e35c624d517fb;
ConnectionProfiles/SQL Anywhere 10 Demo/Data/ConnectionProfileName=
   SQL Anywhere 10 Demo
```

手順 7：SQL Anywhere ODBC ドライバを登録する

SQL Anywhere ODBC ドライバをインストールしてから、管理ツールの iAnywhere JDBC ドライバでこれを使用します。

詳細については、「ODBC ドライバの設定」 846 ページを参照してください。

手順 8：オンライン・ヘルプ・ファイルを登録する

管理ツールには、オンライン・ヘルプの HTML ヘルプ・ファイルが付属しています。これらのファイルは Windows に登録してから、使用してください。登録には、ヘルプ・ファイルの名前を示し、完全に修飾されたパスを指定するレジストリ文字列が必要です。

英語版の SQL Anywhere ヘルプ・ファイルの場合、**HKEY_LOCAL_MACHINE¥SOFTWARE¥Microsoft¥Windows¥HTML Help¥dbmaen10.chm** という文字列値を定義する必要があります。値は *install-dir¥docs¥en¥htmlhelp* です。サポートされているその他の言語版のヘルプ・ファイルは、同じような方法で登録されます。

英語版の Sybase Central ヘルプ・ファイルの場合は、**HKEY_LOCAL_MACHINE¥SOFTWARE¥Microsoft¥Windows¥HTML Help¥scvwen500.chm** という文字列値も定義してください。値は *install-dir¥Sybase Central 5.0.0* です。サポートされているその他の言語版のヘルプ・ファイルは、同じような方法で登録されます。

フランス語、ドイツ語、日本語、簡体字中国語のヘルプ・ファイルを配備する場合は、それらに対しても同様の設定を行う必要があります。

Windows x64 ベース・システムでは、これらのレジストリ・エントリは、32 ビット・レジストリ (**SOFTWARE¥Wow6432Node¥Microsoft**) にあります。

コンピュータにはあらかじめ HTML ヘルプ・ビューワをインストールしておいてください。HTML ヘルプ・ビューワは Windows の最新バージョンには含まれていますが、古いコンピュータにはない場合もあります。このビューワが存在するかどうかは、*Windows* ディレクトリにインストールされる *hh.exe* ファイルを検索することで確認できます。

データベースとデータベース・アプリケーションの配備の詳細については、「データベースとアプリケーションの配備」 823 ページを参照してください。

Linux、UNIX、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、Mobile Link、QAnywhere 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 10.0.1 固有のもので、前後のバージョンには適用できません。

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」 874 ページを参照してください。

ライセンス契約の確認
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：配備するプログラムを決定する

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
- QAnywhere 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 1.5.0

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する

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。すべてのファイルは */opt/sqlanywhere10/* ディレクトリの下にインストールされる必要があります。

参照用の SQL Anywhere インストール環境からファイルをコピーする場合は、ファイルのパーミッションも保持します。一般に、すべてのユーザとグループがすべてのファイルを読み取り、実行できます。

次の表は、各ソフトウェア・バンドルに必要な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、上述したディレクトリ構造にコピーします。通常は、すでにインストールされている SQL Anywhere のコピーからファイルを使用するようにしてください。

ファイル	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	QAnywhere プラグインを含む Sybase Central	SQL Anywhere コンソール
java/dbma[en]10.jar	X	X	X	X	X
java/jodbc.jar	X	X	X	X	X
java/JComponents1001.jar	X	X	X	X	X
java/jlogon.jar	X	X	X	X	X
java/SCEditor500.jar	X	X	X	X	X
java/jsyblib500.jar	X	X	X	X	X
lib32/libjsyblib500_r.so.1	X	X	X	X	X
sun/javahelp-1_1/jh.jar	X	X	X	X	X
sun/JAXB1.0/...	X	X	X	X	X
jre_1.5.0_linux_sun_i586/... (Linux only)	X	X	X	X	X

ファイル	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	QAnywhere プラグインを含む Sybase Central	SQL Anywhere コンソール
jre_1.5.0_solaris_sun_sparc/... (Solaris only)	X	X	X	X	X
lib32/libdblib10_r.so.1	X	X	X	X	X
lib32/libdbjodbc10.so.1	X	X	X	X	X
lib32/libdbodbc10_r.so.1	X	X	X	X	X
lib32/libdbodm10.so.1	X	X	X	X	X
lib32/libdbtasks10_r.so.1	X	X	X	X	X
res/dblg[en]10.res	X	X	X	X	X
lib32/libdbtool10_r.so.1	X	X	X		
bin32/dbisql	X	X			
java/isql.jar	X	X	X		
sybcentral500/scjview		X	X	X	
sybcentral500/scvw[en]500.jar		X	X	X	
sybcentral500/sybasecentral500.jar		X	X	X	
java/salib.jar		X	X	X	
java/saplugin.jar		X			
java/debugger.jar		X			
lib32/libdbput10_r.so.1		X			
java/apache_files.txt			X	X	
java/apache_license_1.1.txt			X	X	
java/apache_license_2.0.txt			X	X	
java/log4j.jar			X	X	
java/mlplugin.jar			X		
java/mldesign.jar			X		

ファイル	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	QAnywhere プラグインを含む Sybase Central	SQL Anywhere コンソール
java/stax-api-1.0.jar			X		
java/wstx-asl-2.0.5.jar			X		
java/velocity.jar			X		
java/velocity-dep.jar			X		
drivers/...			X		
java/qaplugin.jar				X	
java/qaconnector.jar				X	
java/mlstream.jar				X	
bin32/qaagent				X	
lib32/libdbicu10_r.so				X	
lib32/libdbicudt10.so				X	
bin32/dbinit				X	
bin32/dbconsole					X
java/DBConsole.jar					X

前掲のテーブルには、指定が **[en]** であるファイルの数が示されています。Linux の場合、これらのファイルは、さまざまな言語版が用意されており、英語 (**en**) はそのうちの 1 つにすぎません。詳細については、「外国語のメッセージとヘルプ・ファイル」 870 ページを参照してください。

上記のファイル・パスには、「...」で終わっているものもあります。これは、サブディレクトリ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。

Mac OS X では、共有オブジェクトの拡張子は *.dylib* であり、symlink (シンボリック・リンク) の作成は必要ありません。

管理ツールには JRE 1.5.0_10 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。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、適切なベンダから入手する必要があります。サブディレクトリも含めて *jre150* 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は、SQL Anywhere 10 のインストール・コピーから JRE ファイルをコピーします。サブディレクトリも含めてツリー全体をコピーします。

sqlanywhere.jpr ファイルには、Sybase Central の SQL Anywhere プラグインに必要なコンポーネント・ファイルのリストが含まれています。

mobilink.jpr ファイルには、Sybase Central の Mobile Link プラグインに必要なコンポーネント・ファイルのリストが含まれています。

qanywhere.jpr ファイルには、Sybase Central の QAnywhere プラグインに必要なコンポーネント・ファイルのリストが含まれています。QAnywhere プラグインを配備するには、dbinit が必要です。データベース・ツールの配備については、「データベース・ユーティリティの配備」 882 ページを参照してください。

上記の表にある 5 つのバンドルすべてで、リンクをいくつか作成する必要があります。次の項で詳細を説明します。

基本コンポーネント・ファイル

5 つのすべてのバンドルはこの項にリストされたリンクを必要とします。

/opt/sqlanywhere10/lib に次のシンボリック・リンクを作成します。

```
libdbicu10_r.so -> libdbicu10_r.so.1
libdbicudt10.so -> libdbicudt10.so.1
libdbjodbc10.so -> libdbjodbc10.so.1
libjsyblib500_r.so -> libjsyblib500_r.so.1
libdbodbc10_r.so -> libdbodbc10_r.so.1
libdbodm10.so -> libdbodm10.so.1
libdbtasks10_r.so -> libdbtasks10_r.so.1
```

/opt/sqlanywhere10 に次のシンボリック・リンクを作成します。

```
jre150 -> /opt/sqlanywhere10/jre_1.5.0_linux_sun_i586 (Linux)
jre150 -> /opt/sqlanywhere10/jre_1.5.0_solaris_sun_sparc (Solaris)
shared -> /opt/sqlanywhere10
```

Interactive SQL ファイル

/opt/sqlanywhere10/lib に次のシンボリック・リンクを作成します。

```
libdblib10_r.so -> libdblib10_r.so.1
libdbtool10_r.so -> libdbtool10_r.so.1
```

SQL Anywhere プラグインを含む Sybase Central

/opt/sqlanywhere10/lib に次のシンボリック・リンクを作成します。

```
libdblib10_r.so -> libdblib10_r.so.1
libdbput10_r.so -> libdbput10_r.so.1
libdbtool10_r.so -> libdbtool10_r.so.1
```

/opt/sqlanywhere10/sybcentral500 に次のリンクを作成します。

```
jre150 -> /opt/sqlanywhere10/jre_1.5.0_linux_sun_i586 (Linux)
jre150 -> /opt/sqlanywhere10/jre_1.5.0_solaris_sun_sparc (Solaris)
```


Mobile Link プラグインを含む Sybase Central

/opt/sqlanywhere10/lib に次のシンボリック・リンクを作成します。

```
libdblib10_r.so -> libdblib10_r.so.1
libdbmlput10_r.so -> libdbmlput10_r.so.1
libdbtool10_r.so -> libdbtool10_r.so.1
```

/opt/sqlanywhere10/sybcentral500 に次のシンボリック・リンクを作成します。

```
jre150 -> /opt/sqlanywhere10/jre_1.5.0_linux_sun_i586 (Linux)
jre150 -> /opt/sqlanywhere10/jre_1.5.0_solaris_sun_sparc (Solaris)
```

QAnywhere プラグインを含む Sybase Central

/opt/sqlanywhere10/lib に次のシンボリック・リンクを作成します。

```
libdblib10_r.so -> libdblib10_r.so.1
```

/opt/sqlanywhere10/sybcentral500 に次のシンボリック・リンクを作成します。

```
jre150 -> /opt/sqlanywhere10/jre_1.5.0_linux_sun_i586 (Linux)
jre150 -> /opt/sqlanywhere10/jre_1.5.0_solaris_sun_sparc (Solaris)
```

SQL Anywhere コンソール

/opt/sqlanywhere10/lib に次のシンボリック・リンクを作成します。

```
libdblib10_r.so   -> libdblib10_r.so.1
```

外国語のメッセージとヘルプ・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オンライン・ヘルプは、英語からドイツ語、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。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、それらの言語のリソースとヘルプ・ファイルを追加してください。翻訳済みのファイルは次のとおりです。

dbma??10.jar - SQL Anywhere Help Files
dbmaen10.jar English
dbmade10.jar German
dbmaja10.jar Japanese
dbmazh10.jar Chinese

dblg??10.res - SQL Anywhere Messages Files
dblgen10.res English
dblgde10.res German
dblgja10.res Japanese
dblgzh10.res Chinese

scvw??500.jar - Sybase Central Help Files
scvwen500.jar English
scvwde500.jar German
scvwja500.jar Japanese
scvwzh500.jar Chinese

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：環境変数を設定する

管理ツールを実行するには、環境変数のいくつかを定義または変更します。これは通常、SQL Anywhere インストーラによって作成された *sa_config.sh* ファイルで行いますが、ご使用のアプリケーションに最適な方法で柔軟に実行できます。

1. 以下を含むように PATH を設定します。

   ```
   /opt/sqlanywhere10/bin32
   ```

   または

   ```
   /opt/sqlanywhere10/bin32s
   ```

   (どちらか適切なほう)

   Sybase Central の配備では、以下も追加してください。

   ```
   /opt/sqlanywhere10/sybcentral500
   ```

2. 以下を含むように LD_LIBRARY_PATH を設定します。

   ```
   /opt/sqlanywhere10/jre150/lib/i386/client
   /opt/sqlanywhere10/jre150/lib/i386
   /opt/sqlanywhere10/jre150/lib/i386/native_threads
   ```

3. 以下の環境変数を設定します。

   ```
   SQLANY10="/opt/sqlanywhere10"
   SQLANYSH10="/opt/sqlanywhere10"
   ```

手順 4：Sybase Central プラグインを登録する

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 では、インストールされているプラグインをリストした設定ファイルが必要です。このファイルはインストーラによって作成されます。このファイルには、複数の JAR ファイルへのフル・パスが含まれますが、それらのパスはソフトウェアのインストール場所によって変わる可能性があることに注意してください。

ファイルは *.scRepository* と呼ばれます。このファイルは */opt/sqlanywhere10/sybcentral500* ディレクトリにあります。これはプレーン・テキスト・ファイルで、Sybase Central でロードするプラグインに関するいくつかの基本情報が含まれています。

SQL Anywhere のプロバイダ情報は、次のコマンドを使用してリポジトリ・ファイルに作成されます。

```
scjview -register "/opt/sa10/java/sqlanywhere.jpr"
```

sqlanywhere.jpr ファイルの内容は、次のようになります (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)。**AdditionalClasspath** の行は、*.jpr* ファイルでは 1 行に入力してください。

```
PluginName=SQL Anywhere 10
PluginId=sqlanywhere1000
```



```
PluginClass=com.sybase.asa.plugin.SAPlugin
PluginFile=¥_opt¥_sa10¥_java¥_saplugin.jar
AdditionalClasspath=¥_opt¥_sa10¥_java¥_isql.jar:
   ¥_opt¥_sa10¥_java¥_salib.jar:
   ¥_opt¥_sa10¥_java¥_JComponents1001.jar:
   ¥_opt¥_sa10¥_java¥_jlogon.jar:
   ¥_opt¥_sa10¥_java¥_debugger.jar:
   ¥_opt¥_sa10¥_java¥_jodbc.jar
ClassloaderId=SA1000
```

sqlanywhere.jpr ファイルは、SQL Anywhere を最初にインストールしたときに SQL Anywhere インストール環境の *java* フォルダに作成されています。インストール処理の一部として作成が必要な *.jpr* ファイルのモデルとしてこのファイルを使用します。Mobile Link と QAnywhere 用には、それぞれ *mobilink.jpr*、*qanywhere.jpr* という名前のファイルがあります。これらのファイルも *java* フォルダにあります。

上で説明した処理で作成されたサンプルの *.scRepository* ファイルを次に示します。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: 5.0.0.3245
#  Fri Feb 23 13:09:14 EST 2007
#
SCRepositoryInfo/Version=4
#
Providers/sqlanywhere1000/Version=10.0.1.3390
Providers/sqlanywhere1000/UseClassloader=true
Providers/sqlanywhere1000/ClassloaderId=SA1000
Providers/sqlanywhere1000/Classpath=
   ¥_opt¥_sa10¥_java¥_saplugin.jar
Providers/sqlanywhere1000/Name=SQL Anywhere 10
Providers/sqlanywhere1000/AdditionalClasspath=
   ¥_opt¥_sa10¥_java¥_isql.jar:
   ¥_opt¥_sa10¥_java¥_salib.jar:
   ¥_opt¥_sa10¥_java¥_JComponents1001.jar:
   ¥_opt¥_sa10¥_java¥_jlogon.jar:
   ¥_opt¥_sa10¥_java¥_debugger.jar:
   ¥_opt¥_sa10¥_java¥_jodbc.jar
Providers/sqlanywhere1000/Provider=com.sybase.asa.plugin.SAPlugin
Providers/sqlanywhere1000/ProviderId=sqlanywhere1000
Providers/sqlanywhere1000/InitialLoadOrder=0
#
```

注意

- インストーラでは、上記の手法を使用して、これに類似したファイルを書き出します。必要な唯一の変更は、Classpath および AdditionalClasspath 行の JAR ファイルへの完全に修飾されたパスのみです。
- 上記の AdditionalClasspath 行は、右端で折り返し複数行になっています。*.scRepository* ファイルでは 1 行にしてください。
- *.scRepository* ファイルでは、通常のスラッシュ文字 (*/*) は **¥_** のエスケープ・シーケンスで示します。
- 最初の行は、*.scRepository* ファイルのバージョンを示します。
- 先頭に # がある行はコメントです。

データベースとデータベース・アプリケーションの配備の詳細については、「データベースとアプリケーションの配備」 823 ページを参照してください。

手順 5 ：Sybase Central 用の接続プロファイルを作成する

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 10 Demo** の接続プロファイルが *.scRepository* 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 10 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 10 Demo/Name" "SQL Anywhere 10 Demo"
scjview -write "ConnectionProfiles/SQL Anywhere 10 Demo/FirstTimeStart" "false"
scjview -write "ConnectionProfiles/SQL Anywhere 10 Demo/Description" "Suitable Description"
scjview -write "ConnectionProfiles/SQL Anywhere 10 Demo/ProviderId" "sqlanywhere1000"
scjview -write "ConnectionProfiles/SQL Anywhere 10 Demo/Provider" "SQL Anywhere 10"
scjview -write "ConnectionProfiles/SQL Anywhere 10 Demo/Data/ConnectionProfileSettings" "DSN
¥eSQL^0020Anywhere^002010^0020Demo;UID¥eDBA;PWD¥e35c624d517fb"
scjview -write "ConnectionProfiles/SQL Anywhere 10 Demo/Data/ConnectionProfileName" "SQL
Anywhere 10 Demo"
```

接続プロファイルの文字列と値は、*.scRepository* 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、*.scRepository* 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 *.scRepository* 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: 5.0.0.3245
#  Fri Feb 23 13:09:14 EST 2007
#
ConnectionProfiles/SQL Anywhere 10 Demo/Name=SQL Anywhere 10 Demo
ConnectionProfiles/SQL Anywhere 10 Demo/FirstTimeStart=false
ConnectionProfiles/SQL Anywhere 10 Demo/Description=Suitable Description
ConnectionProfiles/SQL Anywhere 10 Demo/ProviderId=sqlanywhere1000
ConnectionProfiles/SQL Anywhere 10 Demo/Provider=SQL Anywhere 10
ConnectionProfiles/SQL Anywhere 10 Demo/Data/ConnectionProfileSettings=
   DSN¥eSQL^0020Anywhere^002010^0020Demo;UID¥eDBA;PWD¥e35c624d517fb;
   UID¥eDBA;
   PWD¥e35c624d517fb;
ConnectionProfiles/SQL Anywhere 10 Demo/Data/ConnectionProfileName=
   SQL Anywhere 10 Demo
```

管理ツールの設定

管理ツールを使用すると、表示または有効化する機能を制御できます。これには、*OEM.ini* という初期化ファイルが使用されます。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 10¥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。

OEM.ini の構造は次のとおりです。

```
[errors]
# reportErrors type is boolean, default = true
reportErrors=false

[updates]
# checkForUpdates type is boolean, default = true
checkForUpdates=false

[help]
# The help file name is made up of
# path + file separator + prefix + current language + suffix + ".jar".
# Here is an example: c:¥sa10¥java¥dbmaen10.jar
# The file separator is the appropriate file separator for the current
# operating system, for example, "¥" for Windows or "/" for UNIX.
# The current language is the two letter ISO code for language.
# For example, en for English.
# directory type is string, default is an empty string
directory='c:¥sa10¥java¥'
# prefix type is string, default is an empty string
prefix='dbma'
# suffix type is string, default is an empty string
suffix='10'
```

reportErrors が false の場合、ソフトウェアがクラッシュしたときに管理ツールはユーザが iAnywhere にエラー情報を送信できるダイアログを表示しません。その代わりに、従来のシンプルなダイアログが表示されます。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。

directory、prefix、suffix を使用すると、ヘルプ・ファイルのロケーションと名前を上書きできます。

dbisqlc の配備

リソースに制限のあるコンピュータでカスタマ・アプリケーションを実行している場合には、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 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。

SQL スクリプト・ファイルの配備

データベースをアプリケーションの一部として作成する場合は、*scripts* ディレクトリ (例：*C:¥Program Files¥SQL Anywhere 10¥scripts*) のコンポーネントを含める必要があります。新しいデータベースの作成に必要なファイルのリストを次に示します。

boottabs.sql
dboviews.sql
migrat.sql
mkexclud.sql
mksadb.sql
oleschem.sql
rstab.sql
sadebug.sql
saopts.sql
sybprocs.sql
syscap.sql
systabviews.sql
sysviews.sql

認証済みバージョンのデータベース・サーバを使用するアプリケーションを配備する場合は、次のファイルも含める必要があります。

authenticate.sql

認証済みアプリケーションの詳細については、「SQL Anywhere の認証アプリケーションの実行」『SQL Anywhere サーバ - データベース管理』を参照してください。

作成するデータベースで jConnect サポートが必要な場合は、次のファイルを含めてください。

jcatalog.sql

次のファイルは SQL Anywhere 10 データベースのアンロードに必要です。

unload.sql
optdeflt.sql
opttemp.sql

次のファイルは、以前のバージョンの SQL Anywhere で作成したデータベースのアンロードに必要です。

unloadold.sql

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、配備ウィザードを使用することです。詳細については、「配備ウィザードの使用」 830 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	NetWare
dbeng10.exe	*dbeng10*	*dbeng10*	なし
dbeng10.lic	*dbeng10.lic*	*dbeng10.lic*	なし
dbsrv10.exe	*dbsrv10*	*dbsrv10*	*dbsrv10.nlm*
dbsrv10.lic	*dbsrv10.lic*	*dbsrv10.lic*	*dbsrv10.lic*
dbserv10.dll	*libdbserv10_r.so, libdbtasks10_r.so*	*libdbserv10_r.dylib, libdbtasks10_r.dylib*	なし
dblg[en]10.dll	*dblg[en]10.res*	*dblg[en]10.res*	*dblg[en]10.res*
dbctrs10.dll	なし	なし	なし
dbextf.dll [1]	*libdbextf.so* [1]	*libdbextf.dylib* [1]	*dbextf.nlm* [1]
dbicu10.dll[2]	*libdbicu10_r.so*[2]	*libdbicu10_r.dylib*[2]	*dbicu10.nlm*[2]
dbicudt10.dll[2]	*libdbicudt10.so*[2]	*libdbicudt10.dylib*[2]	*dbicud10.nlm*[2]
sqlany.cvf	*sqlany.cvf*	*sqlany.cvf*	*sqlany.cvf*
dbodbc10.dll [3]	*libdbodbc10.so* [3]	*libdbodbc10.dylib* [3]	なし
なし	*libdbodbc10_n.so* [3]	*libdbodbc10_n.dylib* [3]	なし
なし	*libdbodbc10_r.so* [3]	*libdbodbc10_r.dylib* [3]	なし
dbjodbc10.dll [3]	*libdbjodbc10.so* [3]	*libdbjodbc10.dylib* [3]	なし
java¥jconn2.jar [3]	*java/jconn2.jar* [3]	*java/jconn2.jar* [3]	*java¥jconn2.jar* [3]
java¥jodbc.jar [3]	*java/jodbc.jar* [3]	*java/jodbc.jar* [3]	*java¥jdbcdrv.zip* [3]
java¥sajvm.jar [3]	*java/sajvm.jar* [3]	*java/sajvm.jar* [3]	*java¥sajvm.jar* [3]

Windows	Linux/UNIX	Mac OS X	NetWare
java¥cis.zip [4]	*java/cis.zip* [4]	*java/cis.zip* [4]	*java¥cis.zip* [4]

[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 (xp_) を使用する場合のみ必要です。

[2] データベースの文字セットがマルチバイトの場合、または UCA 照合順が使用される場合のみ必要です。

[3] データベースで Java を使用する場合のみ必要です。

[4] データベースとリモート・データ・アクセスで Java を使用する場合のみ必要です。

注意

- 場合によって、パーソナル・データベース・サーバ (dbeng10) またはネットワーク・データベース・サーバ (dbsrv10) 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 (*dbeng10.lic* または *dbsrv10.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ディレクトリにあります。
- Java VM jar ファイル (*sajvm.jar*) は、データベース・サーバがデータベース機能で Java を使用する場合のみ必要です。
- 表には、**dbbackup** などのユーティリティの実行に必要なファイルは含まれていません。

  ユーティリティの配備については、「管理ツールの配備」 854 ページを参照してください。

Windows レジストリ・エントリ

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するには、次のレジストリ・キーを作成します。

```
HKEY_LOCAL_MACHINE¥
  SYSTEM¥
    CurrentControlSet¥
      Services¥
    Eventlog¥
   application¥
    SQLANY 10.0
```

このキー内で、EventMessageFile という REG_SZ 値を追加し、*dblgen10.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 10¥win32¥dblgen10.dll*)。英語の DLL である *dblgen10.dll* は、配備の言語にかかわらず指定できます。サンプルのレジストリ変更ファイルを以下に示します。

```
REGEDIT4
[HKEY_LOCAL_MACHINE¥SYSTEM¥CurrentControlSet¥Services¥Eventlog¥Application¥SQLANY
10.0]
"EventMessageFile"="c:¥¥sa10¥¥win32¥¥dblgen10.dll"
```

MESSAGE ...TO EVENT LOG 文によって 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するには、次のレジストリ・キーを作成します。


```
HKEY_LOCAL_MACHINE¥
  SYSTEM¥
    CurrentControlSet¥
      Services¥
    Eventlog¥
   application¥
    SQLANY 10.0 Admin
```

このキー内で、EventMessageFile という REG_SZ 値を追加し、*dblgen10.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 10¥win32¥dblgen10.dll*)。英語の DLL である *dblgen10.dll* は、配備の言語にかかわらず指定できます。サンプルのレジストリ変更ファイルを以下に示します。

```
REGEDIT4
[HKEY_LOCAL_MACHINE¥SYSTEM¥CurrentControlSet¥Services¥Eventlog¥Application¥SQLANY
10.0 Admin]
"EventMessageFile"="c:¥¥sa10¥¥win32¥¥dblgen10.dll"
```

レジストリ・キーを設定することによって、Windows イベント・ログのエントリを抑制できます。レジストリ・キーは、次のとおりです。

```
Software¥Sybase¥SQL Anywhere¥10.0¥EventLogMask
```

これは、HKEY_CURRENT_USER または HKEY_LOCAL_MACHINE ハイブのいずれかに配置できます。イベント・ログのエントリを制御するには、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値が含まれたビット・マスクに割り当てます。SQL Anywhere データベース・サーバでは、次の 3 つのタイプがサポートされています。

```
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変更ファイルを以下に示します。

```
REGEDIT4
[HKEY_LOCAL_MACHINE¥SOFTWARE¥Sybase¥SQL Anywhere¥10.0]
"EventLogMask"=dword:00000007
```

Windows での DLL の登録

SQL Anywhere を配備する場合、SQL Anywhere が正しく機能するためには DLL ファイルを登録してください。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 CE 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。また、次の手順で示されているように、コマンドをバッチ・ファイルに含めることもできます。

♦ **DLL を登録するには、次の手順に従います。**

1. コマンド・プロンプトを開きます。
2. DLL プロバイダがインストールされているディレクトリに移動します。
3. 次のコマンドを入力してプロバイダを登録します (この例では、OLE DB プロバイダが登録されます)。

```
regsvr32 dboledb10.dll
```

次の表は、SQL Anywhere を配備するときに登録が必要な DLL を示します。

ファイル	説明
dbctrs10.dll	SQL Anywhere パフォーマンス・モニタのカウンタ
dbmlsynccom.dll	dbmlsync 統合コンポーネント (非ビジュアル・コンポーネント)
dbmlsynccomg.dll	dbmlsync 統合コンポーネント (ビジュアル・コンポーネント)
dbodbc10.dll	SQL Anywhere ODBC ドライバ
dboledb10.dll	SQL Anywhere OLE DB プロバイダ
dboledba10.dll	SQL Anywhere OLE DB プロバイダ支援
Windows¥system32¥msxml4.dll	Microsoft XML パーサ

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。

データベースの抽出については、「データベース抽出ユーティリティ」 『SQL Remote』を参照してください。

データベースを読み込み専用メディアで配備する

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 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。

セキュリティの配備

次の表に、SQL Anywhere でセキュリティ機能をサポートするコンポーネントをまとめます。

セキュリティ・オプション	セキュリティのタイプ	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	*dbserv10.dll* *libdbserv10_r.so*	付属 [1]
データベースの暗号化	FIPS 認定の AES	*dbfips10.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	*dbrsa10.dll* *libdbrsa10.so* *libdbrsa10.dylib* *libdbrsa10_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	*dbfips10.dll*, *sbgse2.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	ECC	*dbecc10.dll* *libdbecc10.so*	別途ライセンスが必要 [2]

[1] AES および RSA による強力な暗号化は 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 承認ではありません。

[2] ECC テクノロジまたは FIPS 承認テクノロジを使用した強力な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 クライアントの稼働条件が含まれています。

  クライアント・アプリケーションの配備については、「クライアント・アプリケーションの配備」 835 ページを参照してください。

- **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・サーバの配備については、「データベース・サーバの配備」 876 ページを参照してください。

- **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。

- **データベース**　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。

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 (*dblgen10.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。

アプリケーションがデータベースで Java を使用する場合は、Java zip ファイルと Java DLL を提供してください。

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	*dbtool10.dll*	*libdbtool10.so, libdbtasks10.so*	*libdbtool10.dylib, libdbtasks10.dylib*

説明	Windows	Linux/UNIX	Mac OS X
インタフェース・ライブラリ	*dblib10.dll*	*libdblib10.so*	*libdblib10.dylib*
言語リソース・ライブラリ	*dblg[en]10.dll*	*dblg[en]10.res*	*dblg[en]10.res*
[接続] ダイアログ	*dbcon10.dll*		

注意

- Linux/UNIX 上で動作するマルチスレッド・アプリケーションについては、*libdbtool10_r.so*、*libdbtasks10_r.so*、*libdblib10_r.so* を使用してください。
- Mac OS X 上で動作するマルチスレッド・アプリケーションについては、*libdbtool10_r.dylib*、*libdbtasks10_r.dylib*、*libdblib10_r.dylib* を使用してください。
- *dbinit* ユーティリティを使用してデータベースを作成するには、パーソナル・データベース・サーバ (*dbeng10*) 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。
- *dbinit* や *dbunload* などのデータベース・ユーティリティでは、*scripts* ディレクトリの内容が存在している必要があります。「SQL スクリプト・ファイルの配備」 875 ページを参照してください。

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/UNIX
Message Agent	*dbremote.exe*	*dbremote*
データベース・ツール・ライブラリ	*dbtool10.dll*	*libdbtools10.so, libdbtasks10.so*
暗号化／復号モジュール	*dbencod10.dll*	*libdbencod10_r.so*
言語リソース・ライブラリ	*dblg[en]10.dll*	*dblg[en]10.res*
VIM メッセージ・リンク・ライブラリ [1]	*dbvim10.dll*	
SMTP メッセージ・リンク・ライブラリ [1]	*dbsmtp10.dll*	
FILE メッセージ・リンク・ライブラリ [1]	*dbfile10.dll*	*libdbfile10.so*

説明	Windows	Linux/UNIX
FTP メッセージ・リンク・ライブラリ [1]	*dbftp10.dll*	
MAPI メッセージ・リンク・ライブラリ [1]	*dbmapi10.dll*	
インタフェース・ライブラリ	*dblib10.dll*	

[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。

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。

Linux/UNIX 上で動作するマルチスレッド・アプリケーションについては、*libdbtools10_r.so* と *libdbtasks10_r.so* を使用してください。

索引

記号

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

あ

い

う

え

お

か

き

く

け

こ

さ

し

す

せ

そ

ち

つ

て

と

に

は

ひ

ふ

へ

ほ

ま

め

も

ゆ

よ

ら

り

れ

わ

