iAnywhere SOLUTIONS™
A SYBASE COMPANY

Adaptive Server® Anywhere
プログラミング・ガイド

パート番号：DC03947-01-0902-01
改訂：2005 年 3 月

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 Adaptive Server 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 Studio 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 Studio のマニュアル

SQL Anywhere Studio のマニュアルは、各マニュアルを 1 つの大きなヘルプ・ファイルにまとめたオンライン形式、マニュアル別の PDF ファイル、および有料の製本版マニュアルで提供されます。SQL Anywhere Studio のマニュアルは、次の分冊マニュアルで構成されています。

- **『SQL Anywhere Studio の紹介』** このマニュアルでは、SQL Anywhere Studio のデータベース管理と同期テクノロジの概要について説明します。また、SQL Anywhere Studio を構成する各部分について説明するチュートリアルも含まれています。
- **『SQL Anywhere Studio 新機能ガイド』** このマニュアルは、SQL Anywhere Studio のこれまでのリリースのユーザを対象としています。ここでは、製品の今回のリリースと以前のリリースで導入された新機能をリストし、アップグレード手順を説明しています。
- **『Adaptive Server Anywhere データベース管理ガイド』** このマニュアルでは、データベースおよびデータベース・サーバの実行、管理、設定について説明しています。
- **『Adaptive Server Anywhere SQL ユーザーズ・ガイド』** 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。
- **『Adaptive Server Anywhere SQL リファレンス・マニュアル』** このマニュアルは、Adaptive Server Anywhere で使用する SQL 言語の完全なリファレンスです。また、Adaptive Server Anywhere のシステム・テーブルとシステム・プロシージャについても説明しています。
- **『Adaptive Server Anywhere プログラミング・ガイド』** このマニュアルでは、C、C++、Java プログラミング言語を使用してデータベース・アプリケーションを構築、配備する方法につい

て説明します。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。また、Adaptive Server Anywhere ADO.NET データ・プロバイダについても説明します。

- **『Adaptive Server Anywhere エラー・メッセージ』** このマニュアルでは、Adaptive Server 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。
- **『SQL Anywhere Studio セキュリティ・ガイド』** このマニュアルでは、Adaptive Server Anywhere データベースのセキュリティ機能について説明します。Adaptive Server Anywhere 7.0 は、米国政府から TCSEC (Trusted Computer System Evaluation Criteria) の C2 セキュリティ評価を授与されています。このマニュアルには、Adaptive Server Anywhere の現在のバージョンを、C2 基準を満たした環境と同等の方法で実行することを望んでいるユーザにとって役に立つ情報が含まれています。
- **『Mobile Link 管理ガイド』** このマニュアルでは、モバイル・コンピューティング用の Mobile Link データ同期システムについてあらゆる角度から説明します。このシステムによって、Oracle、Sybase、Microsoft、IBM の単一データベースと、Adaptive Server Anywhere や Ultra Light の複数データベースの間でのデータ共有が可能になります。
- **『Mobile Link クライアント』** このマニュアルでは、Adaptive Server Anywhere リモート・データベースと Ultra Light リモート・データベースの設定を行い、これらを同期させる方法について説明します。
- **『Mobile Link サーバ起動同期ユーザーズ・ガイド』** 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の開始を可能にする Mobile Link の機能です。
- **『Mobile Link チュートリアル』** このマニュアルには、Mobile Link アプリケーションの設定と実行を行う方法を説明するチュートリアルがいくつか用意されています。

- **『QAnywhere ユーザーズ・ガイド』** このマニュアルでは、Mobile Link QAnywhere について説明します。Mobile Link QAnywhere は、従来のデスクトップ・クライアントやラップトップ・クライアントだけでなく、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・アプリケーションの開発と展開を可能にするメッセージング・プラットフォームです。
- **『Mobile Link およびリモート・データ・アクセスの ODBC ドライバ』** このマニュアルでは、Oracle、DB2、Sybase Adaptive Server Enterprise で iAnywhere Solutions ODBC ドライバを使用する方法について説明します。これらの ODBC ドライバは、SQL Anywhere Studio に付属しており、Mobile Link 同期と Adaptive Server Anywhere リモート・データ・アクセスで使用します。
- **『SQL Remote ユーザーズ・ガイド』** 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、あらゆる角度から説明します。このシステムによって、Adaptive Server Anywhere または Adaptive Server Enterprise の単一データベースと Adaptive Server Anywhere の複数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。
- **『Sybase Central ヘルプ』** このマニュアルには、Sybase Central や Interactive SQL、その他のグラフィカル・ツールに関するコンテキスト別のヘルプが含まれています。これは、製本版マニュアル・セットには含まれていません。
- **『Adaptive Server Anywhere SNMP Extension Agen ユーザーズ・ガイド』** このマニュアルでは、Adaptive Server Anywhere SNMP Extension Agent の実行と設定について説明しています。
- **『Ultra Light データベース・ユーザーズ・ガイド』** このマニュアルは、Ultra Light 開発者を対象としています。ここでは、Ultra Light データベース・システムの概要について説明します。また、すべての Ultra Light プログラミング・インタフェースに共通する情報を提供します。

- **『Native Ultra Light for Java ユーザーズ・ガイド』** このマニュアルでは、Native Ultra Light for Java について説明します。Native Ultra Light for Java を使用すると、Jeode VM または CrEme VM を実行している Windows CE 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 静的型 Java ユーザーズ・ガイド』** このマニュアルでは、Ultra Light 静的型 Java API について説明します。『*Ultra Light データベース・ユーザーズ・ガイド*』の内容を補足説明します。

- **『Ultra Light.NET ユーザーズ・ガイド』** 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 for MobileVB ユーザーズ・ガイド』** このマニュアルでは、Ultra Light for MobileVB について説明します。Ultra Light for MobileVB を使用すると、Palm 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 ActiveX ユーザーズ・ガイド』** このマニュアルでは、Ultra Light ActiveX について説明します。Ultra Light ActiveX を使用すると、Windows CE が稼働しているハンドヘルド、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 C/C++ ユーザーズ・ガイド』** 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。

- **『Ultra Light for M-Business Anywhere User's Guide』** 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。

このマニュアル・セットの他に、PowerDesigner と InfoMaker には、独自のオンライン・マニュアル (英語版) がそれぞれ用意されています。

マニュアルの形式

SQL Anywhere Studio 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 Studio 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] ー [プログラム] ー [SQL Anywhere 9] ー [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・ディレクトリに保存されている HTML マニュアルを参照してください。

- **PDF 版マニュアル** SQL Anywhere の各マニュアルは、Adobe Acrobat Reader で表示できる PDF ファイルで提供されています。

  PDF 版マニュアルは、オンライン・マニュアルまたは Windows の [スタート] メニューから利用できます。

- **製本版マニュアル** 製本版マニュアルをご希望の方は、ご購入いただいた販売代理店または弊社営業担当までご連絡ください。

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 つだけ選択するか、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、どちら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** }]

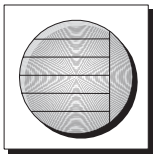
  QUOTES オプションを使用する場合は、ON または OFF のどちらかを選択する必要があります。角カッコと中カッコは入力しないでください。

グラフィック・アイコン

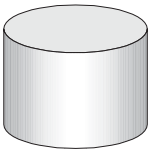
このマニュアルでは、次のアイコンを使用します。

- クライアント・アプリケーション

- Sybase Adaptive Server Anywhere などのデータベース・サーバ



- 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 同期サーバ、SQL Remote Message Agent などがあげられます。

- プログラミング・インタフェース

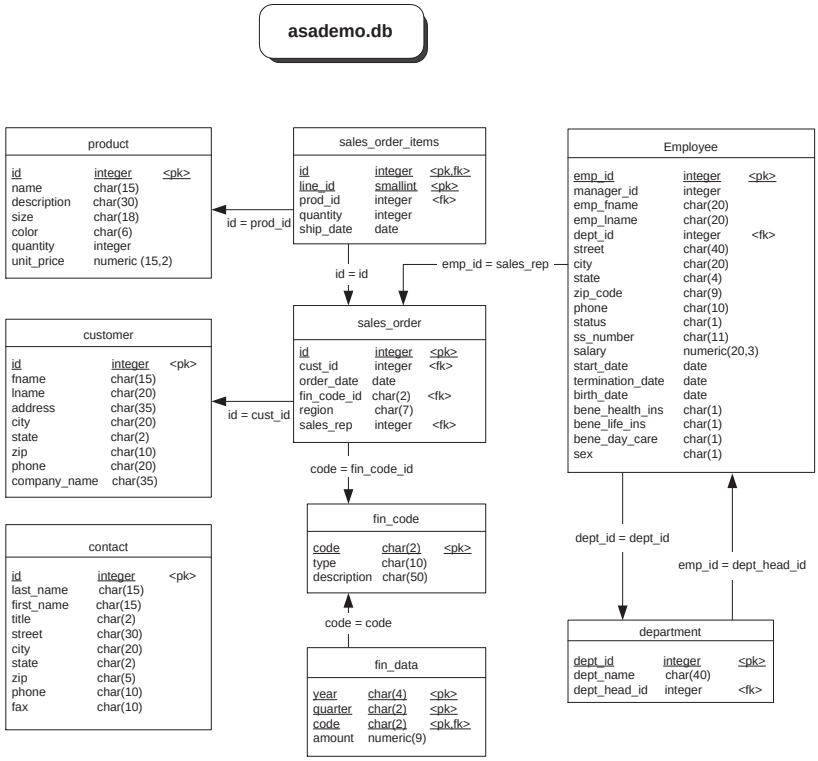
Adaptive Server Anywhere サンプル・データベース

このマニュアルでは、多くの例で Adaptive Server Anywhere サンプル・データベースが使用されています。

サンプル・データベースは、*asademo.db* という名前のファイルに保存され、SQL Anywhere ディレクトリに置かれています。

サンプル・データベースは小規模の企業の例です。データベースには、この企業の内部情報 (従業員、部署、経理) とともに、製品情報や販売情報 (受注、顧客、連絡先) が入っています。データベースに含まれる情報はすべて架空のものです。

次の図は、サンプル・データベース内のテーブルと各テーブル間の関係を示しています。



詳細情報の検索／フィードバックの提供

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。

マニュアルおよびソフトウェアに関するフィードバックは、SQL Anywhere のテクノロジについて議論するニュースグループを介してお送りいただけます。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

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 Solutions 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。

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する責任は負いませんが、お寄せ頂いたご意見、ご提案は必ず読ませて頂きます。

第 1 章

プログラミング・インタフェースの概要

この章の内容

この章では、Adaptive Server Anywhere の各プログラミング・インタフェースについて説明します。クライアント・アプリケーションは、必ずこれらのインタフェースの 1 つを使用してデータベースと通信します。

ODBC プログラミング・インタフェース

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。Adaptive Server 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。現在のマニュアルには、ODBC 開発者のために Adaptive Server Anywhere 特有の注意が付加されています。

ODBC については、「ODBC プログラミング」279 ページを参照してください。

ADO.NET プログラミング・インタフェース

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。

Adaptive Server Anywhere .NET データ・プロバイダは、iAnywhere.Data.AsaClient 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、Adaptive Server Anywhere からデータにアクセスしたりできます。

このマニュアル以外にも、.NET データ・アクセスに関する他の資料を参照し、開発作業に役立てることができます。これにはたとえば、http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/daag.asp などがあります。

ADO.NET プログラミング・インタフェースについては、「Adaptive Server Anywhere .NET データ・プロバイダの概要」421 ページ、「Adaptive Server Anywhere .NET データ・プロバイダのサンプル・アプリケーションの使用」425 ページ、「.NET データ・プロバイダを使用したアプリケーションの開発」437 ページ、「Adaptive Server Anywhere .NET データ・プロバイダ API リファレンス」483 ページを参照してください。

OLE DB と ADO プログラミング・インタフェース

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。

Adaptive Server 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 と ADO のプログラミングに関する主要なマニュアルは、Microsoft Developer Network から入手できます。現在のマニュアルには、OLE DB と ADO の開発者のために Adaptive Server Anywhere 特有の注意が付加されています。

OLE DB プロバイダについては、「OLE DB と ADO プログラミング・インタフェース」401 ページを参照してください。

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、「ADO.NET プログラミング・インタフェース」3 ページを参照してください。

Embedded SQL プログラミング・インタフェース

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 については、「Embedded SQL のプログラミング」169 ページを参照してください。

JDBC プログラミング・インタフェース

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 Studio には、Sybase jConnect という純粋な Java 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 「JDBC プログラミング」129 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」131 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 から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 インタフェースについては、「JDBC プログラミング」129 ページを参照してください。

Open Client プログラミング・インタフェース

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、Adaptive Server 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 インタフェースの詳細については、『ASA データベース管理ガイド』>「Open Server としての Adaptive Server Anywhere」を参照してください。

Open Client アーキテクチャ

Open Client は 2 つのコンポーネントから構成されています。プログラミング・インタフェースとネットワーク・サービスです。

Client Library と DB-Library

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。

DBI モジュールの Perl DBD::ASAny ドライバ

DBD::ASAny 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 Adaptive Server Anywhere データベースにアクセスできるようになります。ASAny 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::ASAny をインストールすると、Perl から Adaptive Server 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。

これらのコンポーネントのインストール方法と、これらを使用する Perl スクリプトの作成方法については、「DBD::ASAny Perl インタフェース」585 ページを参照してください。

コード・サンプルとその他のプログラミング・インタフェース

Adaptive Server Anywhere に対してその他のインタフェースを提供するサポート対象外のコードをダウンロードできます。

- **PHP モジュール** Adaptive Server Anywhere PHP モジュールを使用して、Adaptive Server Anywhere データベースからデータを取り出すことができます。PHP モジュールを使用して Adaptive Server Anywhere に PHP 接続を行うには、Adaptive Server Anywhere モジュールのファイルを PHP のソース・ツリーに追加し、PHP を再コンパイルします。

  PHP モジュールは、個別のダウンロードとして使用できます。詳細については、http://www.ianywhere.com/developer/code_samples/sqlany_php_module.html を参照してください。

- **perl DBI ドライバ** DBD::ASAny は、DBI 用の Adaptive Server Anywhere データベース・ドライバです。これは、perl 言語用のデータベース・アクセスのアプリケーション・プログラミング・インタフェース (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::ASAny を使用すると、perl スクリプトが Sybase Adaptive Server 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。

  詳細については、http://www.ianywhere.com/developer/code_samples/dbd_asa_perl.html を参照してください。

コード・サンプル

アプリケーションのコード・サンプルは、アプリケーション開発者にとって最も役に立つツールの 1 つです。コード・サンプル、ユーティリティ、ソリューション・サンプルは、http://www.ianywhere.com/downloads の iAnywhere Web サイトから入手できます。

第 2 章

アプリケーションでの SQL の使用

この章の内容

データベース・アプリケーションの開発は、多くの面で、使用するアプリケーション開発ツール、データベース・インタフェース、プログラミング言語に依存しますが、開発に影響を及ぼす共通の問題と原則があります。

この章では、ほとんどすべてのインタフェースで共通するいくつかの原則やテクニックについて説明し、詳細な情報の参照先を示します。ここでは、任意のインタフェースを使用したプログラミングについては、詳しく説明しません。

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。

- **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 employee
      WHERE emp_id = 105",
     SQL_NTS );
```

- **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。AsaCommand オブジェクトはその 1 つの例です。

```
AsaCommand cmd = new AsaCommand(
      "select emp_lname from employee", conn );
 AsaDataReader reader = cmd.ExecuteReader();
```

- **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 employee
      WHERE emp_id = 105" );
```

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

```
EXEC SQL EXECUTE IMMEDIATE
  'DELETE FROM employee
   WHERE emp_id = 105';
```

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

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

- **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。

Embedded SQL プログラミングの詳細については、「Embedded SQL のプログラミング」169 ページを参照してください。

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、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・プロシージャでは Embedded SQL 文に非常によく似た文が使われます。

ストアド・プロシージャとトリガの詳細については、『ASA SQL ユーザーズ・ガイド』>「プロシージャ、トリガ、バッチの使用」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、「JDBC プログラミング」129 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、サーバはまず、文を「**準備**」します。文の準備では次のような処理が行われます。

- 文を解析し、内部フォームに変換する。
- 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・オプティマイザにアクセス・プランを生成させる。
- 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」18 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。Adaptive Server 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. 文中に値を保持するために使われる「**バウンド・パラメータ**」を設定します。
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 (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 EXE 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。

❖ 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。

詳細については、「準備文の実行」302 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 (ADO.NET)。

1 文を保持する AsaCommand オブジェクトを作成します。

```
AsaCommand cmd = new AsaCommand(
      "select emp_lname from employee", conn );
```

2 文中のパラメータのデータ型を宣言します。

AsaCommand.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。

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

❖ 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 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」161 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 (Open Client)。

1　CS_PREPARE 型パラメータで **ct_dynamic** 関数を使って文を準備します。

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** を使って文を実行します。

4　CS_DEALLOC 型パラメータで **ct_dynamic** を使っ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」580 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」29 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」29 ページを参照してください。

カーソルとは？

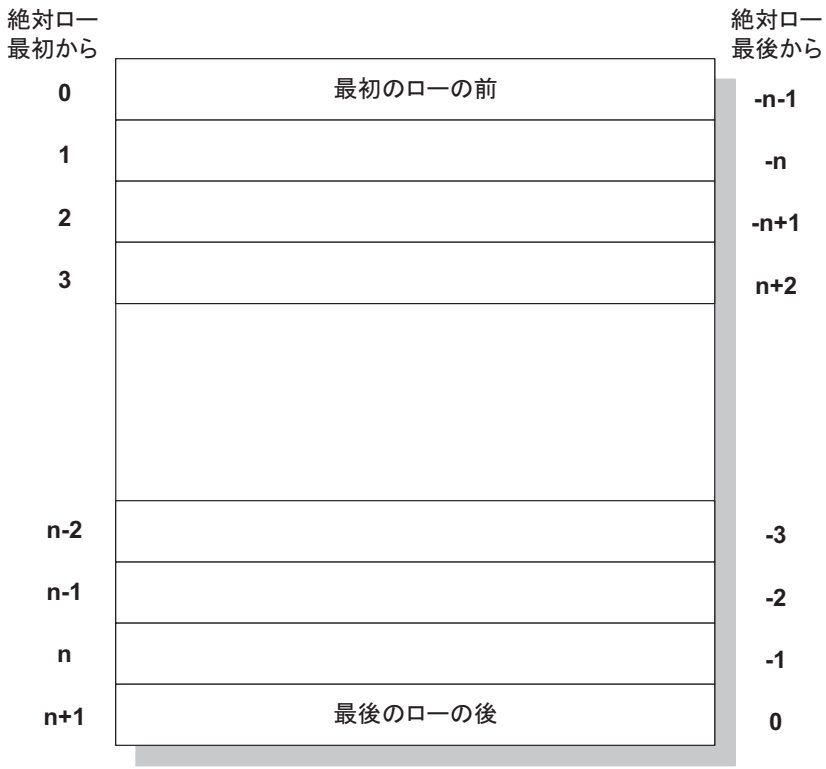
「**カーソル**」とは、結果セットに関連付けられた名前です。結果セットは、SELECT 文かストアド・プロシージャ呼び出しによって取得されます。

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。Adaptive Server 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。

カーソルの使用手順

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 (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」14 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、『ASA SQL リファレンス・マニュアル』>「DECLARE CURSOR 文 [ESQL] [SP]」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、『ASA SQL リファレンス・マニュアル』>「OPEN 文 [ESQL] [SP]」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クエリが実行されます。

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、Adaptive Server 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、『ASA SQL リファレンス・マニュアル』>「FETCH 文 [ESQL] [SP]」と「データのフェッチ」208 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。基本となるデータに設定されていたロックが解除されます。

詳細については、『ASA SQL リファレンス・マニュアル』>「CLOSE 文 [ESQL] [SP]」を参照してください。

6　文を削除します。

カーソルに関連付けられたメモリと文を解放するには、文を解放します。

詳細については、『ASA SQL リファレンス・マニュアル』>「DROP STATEMENT 文 [ESQL]」を参照してください。

❖ カーソルを使用するには、次の手順に従います (ODBC、ADO.NET、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、AsaCommand.ExecuteReader コマンドのみがカーソルを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、Adaptive Server 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。

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、Adaptive Server 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。

Adaptive Server 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 (『ASA SQL ユーザーズ・ガイド』> 「クエリ処理中のワーク・テーブルの使用」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、『ASA データベース管理ガイド』> 「ISOLATION_LEVEL オプション [互換性]」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開いたままにできます。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 AsaDataReader 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。

- **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」305 ページを参照してください。

- **ADO.NET**　AsaDataReader.NextResult メソッドを使用します。「NextResult メソッド」535 ページ を参照。

- **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」209 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、「JDBC を使用したクエリ」160 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」581 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_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、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、JDBC 1.1 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、『ASA SQL リファレンス・マニュアル』>「FROM 句」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。

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] ボタンを押して、(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、Adaptive Server Anywhere のカーソルと、Adaptive Server 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。

Adaptive Server Anywhere のカーソルについては、「Adaptive Server Anywhere のカーソル」34 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。

- ODBC と OLE DB (ADO) 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」305 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。
- 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。
- JDBC の場合：
  - jConnect 4.x では、asensitive カーソルのみが提供されます。
  - jConnect 5.x ではすべてのカーソル・タイプがサポートされていますが、スクロール可能カーソルではパフォーマンスが著しく低下します。
  - iAnywhere JDBC ドライバではすべてのカーソル・タイプがサポートされています。
- Sybase Open Client でサポートされているのは asensitive カーソルだけです。また、ユニークではない更新可能なカーソルを使用すると、パフォーマンスが著しく低下します。

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。

Adaptive Server 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、Adaptive Server Anywhere は、その特性をできるだけ一致させます。Adaptive Server Anywhere のカーソルが、プログラミング・インタフェースで指定されたカーソル・タイプに一致する仕組みについては、次の項で詳しく述べます。

特性を全部指定できない場合もあります。たとえば、Adaptive Server 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。

Adaptive Server 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、Adaptive Server Anywhere はカーソルを 1 つ返します。Adaptive Server 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。Adaptive Server 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、Adaptive Server Anywhere はそれに答えてカーソル感知性を設定します。

ODBC と OLE DB

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	Adaptive Server Anywhere のカーソル
STATIC	Insensitive
KEYSET	Value-sensitive
DYNAMIC	Sensitive
MIXED	Value-sensitive

Adaptive Server Anywhere のカーソルと動作については、「Adaptive Server Anywhere のカーソル」34 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、「カーソル特性の選択」305 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。

ADO.NET

前方専用、読み込み専用のカーソルは、AsaCommand.ExecuteReader を使用して使用できます。AsaDataAdapter 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。

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	**Adaptive Server 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。

JDBC

Open Client アプリケーションで使用できるカーソル・タイプは 1 つだけです。このカーソル・タイプは、asensitive です。JDBC では、**ExecuteQuery** 文を実行してカーソルを開きます。

Open Client

Open Client アプリケーションで使用できるカーソル・タイプは 1 つだけです。このカーソル・タイプは、asensitive 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。Adaptive Server Anywhere は DYNAMIC カーソルを除くすべての種類のカーソルにブックマークをサポートします。

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。

Adaptive Server 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。

Adaptive Server 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 ?」18 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 (emp_id はプライマリ・キー・カラムです)。

emp_id	emp_lname
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 emp_id, emp_lname
 FROM employee
 ORDER BY emp_id
```

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

Adaptive Server 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。

  詳細については、「insensitive カーソル」41 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。

  詳細については、「sensitive カーソル」43 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。

  詳細については、「asensitive カーソル」45 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を変更できます。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。

  詳細については、「Value-sensitive カーソル」46 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」49 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 emp_id, emp_lname
 FROM employee
 ORDER BY emp_id
```

emp_id	emp_lname
102	Whitney
105	Cobb
160	Breault
…	…

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

この場合、カーソル・アクションの結果は、カーソルの感知性によって異なります。

- **insensitive カーソル**　DELETE は、カーソルを使用して表示される結果セットのメンバ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。
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 emp_id, emp_lname
 FROM employee
```

emp_id	emp_lname
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、emp_lname 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)	「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。
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、OLE DB、ADO	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 または NO SCROLL	
JDBC	サポート対象外	
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

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。

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

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、OLE DB、ADO	動的	
Embedded SQL	SENSITIVE	要求されているワーク・テーブルがなく、PREFETCH がオフに設定されている場合は、DYNAMIC SCROLL の要求にも応じて提供される

説明

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 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、Adaptive Server 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、『ASA SQL ユーザーズ・ガイド』> 「クエリの最適化と実行」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、OLE DB、ADO	感知性は未指定
Embedded SQL	DYNAMIC SCROLL

説明

Adaptive Server 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。

Adaptive Server 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
ODBC、OLE DB、ADO	キーセット駆動型
Embedded SQL	SCROLL
JDBC	キーセット駆動型
Open Client	キーセット駆動型

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、Adaptive Server 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」36 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、Adaptive Server 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、emp_lname と emp_fname 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。詳細については、『ASA SQL ユーザーズ・ガイド』> 「バルク・オペレーションのパフォーマンスの側面」を参照してください。

詳細については、『ASA エラー・メッセージ』> 「最後に読み込まれた後で、ローは更新されています。」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、『ASA エラー・メッセージ』> 「最後に読み込まれた後で、ローは更新されています。オペレーションはキャンセルされました。」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 (SQL 状態 24503) 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、『ASA エラー・メッセージ』> 「カーソルの現在のローがありません。」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」26 ページを参照してください。

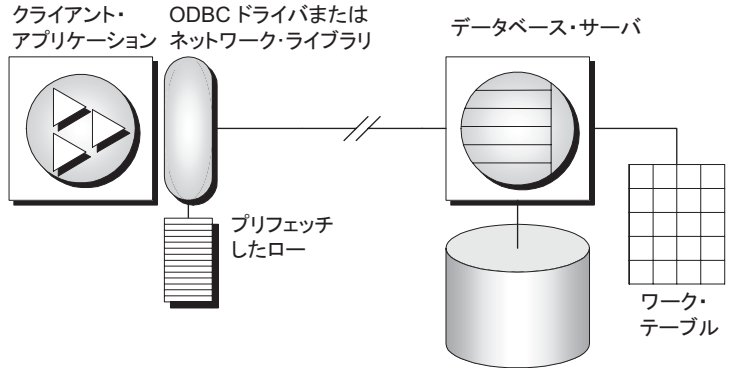
カーソルの感知性とパフォーマンス

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。

更新可能なカーソルでは、ワーク・テーブルを使用したり、結果をプリフェッチしたりできません。どちらか 1 つでも使った場合、カーソルは更新情報の損失に対して無防備になります。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 product
```

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. 次の文を使用して更新されたローを別のトランザクションが更新する。

```
UPDATE product
 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。

感知性も、同様の制限によって制御されます。詳細については、distinct カーソル・タイプの説明を参照してください。

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、Adaptive Server 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。Adaptive Server 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、『ASA データベース管理ガイド』>「PREFETCH オプション [データベース]」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 OFF 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 と設定する方が効果的です。

詳細については、『ASA データベース管理ガイド』>
「PREFETCH オプション [データベース]」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。

カーソルの感知性と独立性レベル

カーソルの感知性とトランザクションの独立性レベルはどちらも同時実行性の問題を処理しますが、それぞれ方法が異なります。

トランザクションの独立性レベルを選択する場合は (通常は接続レベルを選択)、データベースのローをロックする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値にアクセスしたり修正したりできなくなります。

カーソルの感知性を選択する場合は、カーソルを使用してアプリケーションに表示する変更を決定します。カーソルの感知性を設定しただけでは、データベースのローをロックするタイミングを指定したことにはなりません。また、データベース自体に加えることのできる変更を制限したことにもなりません。

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)」224 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することもでき、**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、記述子を使用してエンジン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、出力パラメータの詳細については、『ASA SQL リファレンス・マニュアル』> 「DESCRIBE 文 [ESQL]」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、『ASA SQL ユーザーズ・ガイド』>「トランザクションと独立性レベル」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、『ASA SQL ユーザーズ・ガイド』>「オートコミット・モードをオフにする」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」57 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 (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 (ADO.NET)。

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

  詳細については、「Transaction 処理」475 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 (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 (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 (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・オプションを OFF に設定します。

```
SET OPTION CHAINED='OFF'
```

オートコミット実装の詳細

前項「オートコミットの動作を制御する」56 ページでは、オートコミット動作が Adaptive Server Anywhere programming の各プログラミング・インタフェースから制御できることを説明しました。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。

  Adaptive Server Anywhere では、ODBC と OLE DB の各アプリケーションに対してクライアント側オートコミットが使用されます。

- **サーバ側オートコミット** アプリケーションがオートコミットを使用すると、データベース・サーバは各 SQL 文の後に COMMIT 文を発行します。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。

  Adaptive Server Anywhere では、サーバ側オートコミットが Embedded SQL、JDBC、Open Client の各アプリケーションに使用されます。

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 文の後に COMMIT 文を発行します。

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
ODBC アプリケーションまたは OLE DB アプリケーションではオートコミットと CHAINED オプションの使用を組み合わせないでください。

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。

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・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・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_AT_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_AT_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・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、この変更が反映するのは閉じられたカーソルに限られます。

第 3 章

データベースにおける Java の概要

この章の内容

この章では、データベースで Java を使用する目的と概念について説明します。

Adaptive Server Anywhere は、Java Runtime Environment です。Java は、SQL ストアド・プロシージャ言語の代わりとなります。

概要

Adaptive Server Anywhere は、**Java Runtime Environment** です。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。Java Runtime Environment をデータベース・サーバに構築すれば、データベースにプログラミング論理を追加する有効な手段となります。

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。Adaptive Server 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には、ストアド・プロシージャ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。

別途ライセンスを入手可能なコンポーネント

データベース内の Java に関するコンポーネントには別途ライセンスが必要です。インストールする場合はご注文ください。このコンポーネントのご注文については、ご購入いただいた販売代理店または弊社営業担当までご連絡ください。

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ユーザ定義データ型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 の概要」61 ページ (この章)	Java の概念と Adaptive Server Anywhere への適用方法
「データベースにおける Java の使用」97 ページ	データベースで Java を使用する実際の手順
「JDBC プログラミング」129 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
『ASA SQL ユーザーズ・ガイド』>「データベースでのデバッグ論理」	データベースで実行される Java コードのテストとデバッグ

Java マニュアルの使用

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
オブジェクト指向型プログラミングに関して経験の浅い方	「Java セミナー」71 ページ
「インスタンス化」「フィールド」「クラス・メソッド」などの用語の意味を知りたい方	「Java セミナー」71 ページ
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」83 ページ 「チュートリアル：データベースにおける Java の演習」91 ページ
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」65 ページ

対象読者	参照箇所
Java からデータにアクセスする方法を知りたい方	「JDBC プログラミング」129 ページ
Java 用のデータベースを準備したい方	「データベースを Java 実行可能にする」102 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**　内部 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。
- **SQL から Java を呼び出すことができる**　SQL 文から Java 関数 (メソッド) を呼び出すことができます。Java メソッドは、データベースに論理を追加する SQL ストアド・プロシージャよりも高機能な言語を提供します。
- **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。
- **データベースの Java をデバッグできる**　Adaptive Server Anywhere デバッガを使用して、データベースの Java クラスをテストしたりデバッグしたりできます。
- **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 命令を格納する方法は？

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;
```

Adaptive Server 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」 109 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？

Adaptive Server Anywhere には、データベース環境で実行される「**Java 仮想マシン（VM)**」が搭載されています。Adaptive Server Anywhere 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。

VM の他に、データベース・サーバの SQL 要求プロセッサは、VM に呼び出され、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 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。

スタンドアロン VM との違い

Sun Java VM *java.exe* などの標準 VM を使用して Java コードを実行する場合と、データベースで Java コードを実行する場合とでは、違いが 1 つあります。Sun VM がコマンド・ラインから実行されるのに対し、Adaptive Server Anywhere Java VM は、SQL 文の実行の一部として要求されると、Java オペレーションを実行するためにいつでも使用できます。

Java VM は外部からアクセスできません。このインタプリタは、SQL 文の実行に Java オペレーションが必要な場合にだけ使用されます。データベース・サーバは必要に応じて VM を自動的に起動するので、ユーザがわざわざ VM を起動したり、停止したりする必要はありません。

Java がよい理由は？

Java はデータベースで効果的に使用できるよう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、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？

データベースにおける Java は、Windows CE ではサポートされていません。その他の Windows オペレーション・システム、UNIX、NetWare ではサポートされています。

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、『ASA SQL リファレンス・マニュアル』>「CREATE PROCEDURE 文」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** は指定した方法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。たとえば、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、開発者は、ユーザが作成した Employee クラスまたは Package クラスを、Java で設計、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、Adaptive Server 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」109 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 Adaptive Server Anywhere に接続できます。Adaptive Server 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。

詳細については、「JDBC プログラミング」129 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、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？

Adaptive Server 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。

Adaptive Server Anywhere で使用される 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、コンパイルされ、次に使用、テスト、デバッグのためにデータベースにインストールされます。

Java セミナー

この項では、主要な Java の概念について説明します。この項を読み終えたら、簡単なクラス定義やメソッドの実行などの Java コードを調べたり、現在何が起こっているかを理解できたりするようになります。

Java samples ディレクトリ
このマニュアルで例として使用されるクラスのいくつかは、Java samples ディレクトリ *Samples¥ASA¥Java* に配置されています。このディレクトリは、SQL Anywhere ディレクトリのサブディレクトリです。

2 つのファイルが、各 Java クラスの例 (Java ソースとコンパイル済みクラス) を示しています。コンパイル済みの Java クラス例は、修正しないですぐにデータベースにインストールできます。

Java クラスの知識

「Java クラス」はデータと、情報を保持したり計算を実行したりできる機能を結合します。クラスの概念を理解する 1 つの方法として、クラスをエンティティ、すなわち、ある事柄の抽象表現として見る方法があります。

たとえば、Invoice クラスを毎日の業務で使用される紙の送り状を模倣して設計します。紙の送り状と同様に、Invoice クラスのインスタンスにも特定の情報 (行項目詳細、送り状作成者名、送り状発行日、支払額と支払い日) が含まれます。クラスは、「フィールド」内に情報を保持します。

クラスはデータを記述するだけでなく、計算や論理演算を実行できます。たとえば、Invoice クラスは、ユーザが介入しなくても、各 Invoice オブジェクトの行項目リストに対して税金を計算し、それを小計に追加して最終合計を出すことができます。また、このようなクラスは、重要な情報が確実に Invoice に存在し、支払いがいつ期限を過ぎるか、またはいつ分割払いするかを示すこともできます。計算や他の論理演算は、クラスの「*メソッド*」によって実行されます。

例

次の Java コードは、Invoice というクラスを宣言します。このクラス宣言は、*Invoice.java* ファイルに格納されます。その後、Java コンパイラを使用して Java クラスにコンパイルされます。

Java クラスのコンパイル

Java クラスのソースをコンパイルすると、ソース・ファイルと同じ名前で拡張子が異なる新しいファイルが作成されます。*Invoice.java* をコンパイルすると、*Invoice.class* ファイルが作成されます。このファイルは、Java アプリケーションで使用され、Java VM で実行できます。

クラス宣言をコンパイルするための Sun JDK ツールは、*javac.exe* です。

```
public class Invoice {
   // So far, this class does nothing and knows nothing
 }
```

class キーワードの後にクラス名が続きます。中カッコ (開くと閉じる) があります。フィールドやメソッドなど、これらの中カッコの間で宣言されたものはすべてクラスの一部になります。

実際、クラス宣言以外に Java コードは存在しません。Java インタプリタが他のオブジェクトを作成して管理するために自動的に実行する Java プロシージャ (アプリケーションの開始点であることが多い **main** メソッド) でさえ、それ自体はクラス宣言内にあります。

データベースの Java ランタイム・クラス (*rt.jar* または *classes.zip*) は、JDK によってインストールされる 1.1 ターゲットを使用して構築されます。1.1 以外のターゲットの場合、ClassFormatErrors が発生します。JDK バージョンを使用してデータベースにインストールするためのクラスをコンパイルする場合、Java コンパイラのオプション ″target″ を 1.1 に設定してください。次に例を示します。

```
javac -target 1.1 Invoice.java
```

JDK 1.3 以前のバージョンを使用する場合、コンパイラのデフォルト・ターゲットは 1.1 になっています。

Java におけるサブクラス

他のクラスの「**サブクラス**」として、クラスを定義できます。あるクラスのサブクラスは、親クラスのフィールドやメソッドを使用できます。これは、「**継承**」と呼ばれます。サブクラスにだけ適用される追加のメソッドやフィールドを定義したり、継承されたフィールドやメソッドの意味を再定義したりできます。

Java は単一階層言語です。すなわち、実際に作成または使用するすべてのクラスは 1 つのクラスから継承します。このことは、高レベルのクラスを使用する前に、低レベルのクラス (階層の上位クラス) が存在しなければならないことを意味します。Java アプリケーションを実行するために必要なクラスの基本セットは、「**ランタイム Java クラス**」または「**Java API**」と呼ばれます。

Java オブジェクトの知識

送り状のフォームが送り状に含める情報を定義するテンプレートであるのと同じように、「**クラス**」はオブジェクトの機能を定義するテンプレートです。

クラスには、オブジェクトについての特定の情報は含まれません。アプリケーションがクラス (テンプレート) に基づいてオブジェクトを作成または「**インスタンス化**」し、オブジェクトがデータを保持したり、計算を実行したりします。このインスタンス化されたオブジェクトが、クラスの「**インスタンス**」です。たとえば、Invoice オブジェクトは Invoice クラスのインスタンスです。クラスはオブジェクトの機能を定義しますが、オブジェクトはクラスを具体化したものでり、クラスを意味のある有用なものにします。

この送り状の例では、送り状のフォームは、そのフォームを基にしたすべての送り状が達成できることを定義します。フォームが 1 つあり、そのフォームに基づく送り状が、ゼロまたは多数存在します。フォームには定義が含まれますが、送り状は実用性をカプセル化します。

Invoice オブジェクトは、作成、情報の格納、格納、検索、編集、更新などが行われます。

1 つの送り状のテンプレートで多くの送り状をそれぞれ詳細内容で分類し作成するのとまったく同じように、1 つのクラスから多くのオブジェクトを生成できます。

メソッドとフィールド

「**メソッド**」は、何らかの操作を行うクラスの一部、すなわちクラスのために計算を実行したり、他のオブジェクトと対話したりする機能です。メソッドは、引数を指定でき、呼び出し関数に値を返すことができます。戻り値が不要の場合、メソッドは **void** を返すことができます。クラスは、任意の数のメソッドを持つことができます。

「**フィールド**」は、情報を保持するクラスの一部です。*JavaClass* 型のオブジェクトを作成すると、*JavaClass* 内のフィールドは、そのオブジェクトにとってユニークな状態を保持します。

クラス・コンストラクタ

クラス・コンストラクタを呼び出して、オブジェクトを作成します。「**コンストラクタ**」は、次のプロパティを持つメソッドです。

- コンストラクタ・メソッドはクラスと同じ名前を持ち、宣言されたデータ型を持ちません。たとえば、Product クラスの簡単なコンストラクタは次のように宣言されます。

```
Product () {
 ...constructor code here...
 }
```

- クラス定義にコンストラクタ・メソッドを含めない場合、Java ベース・オブジェクトで提供されるデフォルトのメソッドが使用されます。
- 各クラスに、異なる数とタイプの引数を持つ複数のコンストラクタを提供できます。コンストラクタを呼び出すと、適切な数とタイプの引数を持つコンストラクタが使用されます。

フィールドの知識

Java フィールドには、2 つのカテゴリがあります。

- **インスタンス・フィールド**　各オブジェクトは、オブジェクト作成時に作成された独自のインスタンス・フィールドのセットを持っています。これらは、そのインスタンスに固有の情報を保持します。たとえば、Invoice クラスの **lineItem1Description** フィールドは、特定の送り状にある行項目の説明を保持します。インスタンス・フィールドには、オブジェクト参照を介してのみアクセスできます。

- **クラス・フィールド**　クラス・フィールドは、特定のインスタンスから独立した情報を保持します。クラス・フィールドは、クラスが最初にロードされるときに作成されます。オブジェクトをいくつ作成しても、それ以上のインスタンスは作成されません。クラス・フィールドは、クラス名またはオブジェクト参照を介してアクセスできます。

クラスでフィールドを宣言するには、その型と名前の次にセミコロンを付けて指定します。クラス・フィールドを宣言するには、宣言内で**静的**な Java キーワードを使用します。フィールドは、メソッド内ではなくクラスで宣言します。メソッド内で変数を宣言すると、フィールドは、クラスではなくメソッドの一部となります。

例

次の Invoice クラスの宣言には、送り状の 2 行の項目に含まれる情報に対応して、4 つのフィールドが含まれています。

```
public class Invoice {

    // Fields of an invoice contain the invoice data
    public String lineItem1Description;
    public int lineItem1Cost;

    public String lineItem2Description;
    public int lineItem2Cost;

}
```

メソッドの知識

Java メソッドには、2 つのカテゴリがあります。

- **インスタンス・メソッド**　Invoice クラスの **totalSum** メソッドは、計算と税金の加算を行い、合計額を返すことができます。ただし、その行項目コスト値を持つ **Invoice** オブジェクトと一緒に呼

び出される場合にだけ役立ちます。送り状の行項目はクラスではなくオブジェクトに含まれているため、計算はオブジェクトに対してのみ実行できます。

- **クラス・メソッド** クラス・メソッド (「**静的メソッド**」とも呼ばれる) は、はじめにオブジェクトを作成しなくても呼び出すことができます。クラス・メソッドを呼び出すために必要なのは、クラスとメソッドの名前だけです。

  インスタンス・メソッドと同様に、クラス・メソッドは引数と戻り値を受け入れます。通常、クラス・メソッドはある種のユーティリティまたはクラス全体の機能に関する情報関数を実行します。

  クラス・メソッドは、インスタンス・フィールドにアクセスできません。

メソッドを宣言するには、戻り型、名前、指定するパラメータを入力します。クラス宣言と同様に、メソッドも一対の大カッコを使用して、コードが実行されるメソッド本体を示します。

```
public class Invoice {

    // Fields
    public String lineItem1Description;
    public double lineItem1Cost;

    public String lineItem2Description;
    public double lineItem2Cost;

    // A method
    public double totalSum() {
        double runningsum;

        runningsum = lineItem1Cost + lineItem2Cost;
        runningsum = runningsum * 1.15;

        return runningsum;
    }
}
```

totalSum メソッド内で、**runningsum** という名前の変数が宣言されます。この変数はまず、最初と 2 番目の行項目コストの小計を保持します。この小計に 15 % (税率) を乗算して合計値を出します。

ローカル変数 (メソッド内の変数) は、次に呼び出し関数に返されます。**totalSum** メソッドを呼び出すと、2 行の項目コスト・フィールドの値とそれぞれの税額を足した合計額を返してきます。

例

Adaptive Server Anywhere で提供される **java.lang.Integer** クラスの **parseInt** メソッドは、クラス・メソッドの一例です。文字列引数が与えられると、**parseInt** メソッドは、その文字列を整数にして返します。

たとえば、文字列値 "1" の場合、**parseInt** メソッドはこの Java コード・フラグメントで示すように、最初に **java.lang.Integer** クラスのインスタンスの作成を要求しないで整数値 1 を返します。

```
String num = "1";
 int i = java.lang.Integer.parseInt( num );
```

例

次の Invoice クラスのバージョンには、インスタンス・メソッドとクラス・メソッドの両方が含まれています。**rateOfTaxation** という名前のクラス・メソッドは、送り状の合計計算にクラスが使用する税率を返します。

rateOfTaxation メソッドをクラス・メソッド (インスタンス・メソッドやフィールドではない) にする利点は、他のクラスやプロシージャが最初にクラスのインスタンスを作成しなくても、このメソッドで返される値を使用できる点です。このクラスで使用される税率を返すには、クラスとメソッドの名前だけが必要です。

rateofTaxation をフィールドではなく、メソッドにすると、アプリケーション開発者はその戻り値を使用するオブジェクト、アプリケーション、またはプロシージャに影響を与えないで税率の計算方法を変更できます。Invoice の将来のバージョンでは、戻り値を使用する他のメソッドに影響を与えないで、さらに複雑な計算を基に **rateOfTaxation** クラス・メソッドの戻り値を作成できる予定です。

```
public class Invoice {
     // Fields
     public String lineItem1Description;
     public double lineItem1Cost;
     public String lineItem2Description;
     public double lineItem2Cost;
     // An instance method
     public double totalSum() {
         double runningsum;
```

```
            double taxfactor = 1 +
    Invoice.rateOfTaxation();

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

            return runningsum;
        }
        // A class method
        public static double rateOfTaxation() {
            double rate;
            rate = .15;

            return rate;
        }
    }
```

オブジェクト指向型言語と手続き型言語

この項では、オブジェクト指向型言語よりも C や SQL ストアド・プロシージャ言語などの手続き型言語の方に詳しい方を対象に、手続き型言語とオブジェクト指向型言語の主な類似点と相違点について説明します。

Java はクラスを基にしている

Java のコードの主要な構造単位は、「**クラス**」です。

Java クラスは、特定の識別可能なカテゴリに関連しているため、グループ化されたプロシージャや変数の集まりと見なすことができます。

ただし、クラスの使用方法によって、オブジェクト指向型言語と手続き型言語は区別されます。手続き型言語で記述したアプリケーションを実行すると、通常はいったんメモリにロードされ、ユーザを定義済みの実行コースに導きます。

Java などのオブジェクト指向言語の場合、クラスはテンプレートのように使用されます (テンプレートとは実行される可能性のあるプログラムの定義をいいます)。必要に応じて、クラスの複数のコピーを作成して、独自のデータ、値、実行コースを含むことができるクラスの各インスタンスとともに動的にロードできます。ロードされた各クラスは、メモリにロードされた他のクラスとは無関係に動作または実行できます。

実行用にメモリにロードされたクラスは、「インスタンス化された」と呼ばれます。インスタンス化されたクラスは、オブジェクトと呼ばれます。オブジェクトとはクラスから派生したアプリケーションであり、ユニークな値を保持するか、または他のクラス・インスタンスとは無関係な方法でそのメソッドを実行します。

Java の用語解説

ここでは、Java クラスに関する詳細をいくつかまとめて説明します。Java 言語を徹底的に解説するわけではありませんが、Adaptive Server Anywhere で Java クラスを使用するときに役立つことがあります。

Java 言語の詳細については、*Samples¥ASA¥Java¥Tjava.pdf* ファイルの Adaptive Server Anywhere に同梱されているオンライン・ブック『*Thinking in Java*』(Bruce Eckel 著) を参照してください。

パッケージ

「**パッケージ**」は、共通の目的やカテゴリを共有するクラス・グループです。パッケージのメンバは、別のメンバのデータやメソッドにアクセスできる特殊な権限を持つので、**protected** アクセス変更子と呼ばれます。

Java では、パッケージはライブラリに相当します。パッケージはクラスを集めたもので、**import** 文を使用して利用できます。次の Java 文は、Java API からユーティリティ・ライブラリをインポートします。

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

パッケージは、通常、拡張子 *.jar* または *.zip* が付いた JAR ファイルに入っています。

public 対 private

アクセス変更子は、他の Java オブジェクトに対するフィールド、メソッド、またはクラスの可視性 (本来、宣言の先頭に使用される **public**、**private**、または **protected** キーワード) を決定します。

- **public** クラス、メソッド、フィールドは、どこでも参照できます。
- **private** クラス、メソッド、フィールドは、そのクラス内で定義されているメソッド内でのみ参照できます。

- **protected** メソッドまたはフィールドは、そのクラス内、クラスのサブクラス内、または同じパッケージの他のクラス内で定義されているメソッドでのみ参照できます。
- デフォルトの可視性は、パッケージとも呼ばれ、そのクラス内や同じパッケージの他のクラス内で参照できるメソッドまたはフィールドを示します。

コンストラクタ

コンストラクタは、クラスのインスタンスが作成されたときに呼び出される Java クラスの特殊メソッドです。

クラスは、複数の上書きコンストラクタを含む独自のコンストラクタを定義できます。オブジェクトの作成にどの引数を使用したかによって、どのコンストラクタを使用するかが決まります。クラスのインスタンスの作成に使用する引数の型、数、順序がクラスのコンストラクタの 1 つと一致する場合、そのコンストラクタを使用してオブジェクトが作成されます。

ガーベジ・コレクション

「**ガーベジ・コレクション**」は、参照を持たないオブジェクトを自動的に削除します。ただし、テーブル内に値として格納されているオブジェクトは例外です。

Java にはデストラクタ・メソッドのようなものがありません (C++ にはある)。Java クラスは、ガーベジ・コレクション中にオブジェクトを廃棄するとき、オペレーションをクリーンアップするために独自の **finalize** メソッドを定義できます。

インタフェース

Java クラスは、1 つのクラスからだけ継承できます。Java は、複数継承の代わりにインタフェースを使用します。クラスは複数のインタフェースを実装でき、各「**インタフェース**」は、クラスのコンパイル用にクラスで実装しなければならないメソッドとメソッド・プロファイルのセットを定義します。

インタフェースは、クラスが宣言しなければならないメソッドと静的フィールドを定義します。インタフェースで宣言されるメソッドとフィールドの実装はインタフェースを使用するクラス内に配置されます。インタフェースはクラスが宣言するものを定義し、実装方法はクラスが決定します。

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捕獲され、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。

Adaptive Server 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。

Java のエラー・タイプ

Java のエラーはすべて、2 種類のエラー・クラス **Exception** と **Error** から派生しています。通常、Exception ベースのエラーは、メソッドのエラー処理コードで処理されます。Error タイプのエラーは、Java ランタイム・システム内の内部エラーおよびリソース不足エラー用に確保されています。

Exception クラス・エラーは、スロー、捕獲されます。例外処理コードの特徴として、**try**、**catch**、**finally** の各コード・ブロックがあります。

try ブロックは、エラーを生成するコードを実行します。**catch** ブロックは、**try** ブロックの実行によってエラーが生成 (スロー) されると実行されるコードです。

finally ブロックは、エラーが生成、捕獲されたかどうかに関係なく実行されるコード・ブロックを定義し、通常、オペレーションのクリーンアップに使用されます。このブロックは、いかなる状況においても省略できないコードに使用されます。

Exception クラス・エラーには、ランタイム例外とそれ以外の例外の 2 種類があります。

ランタイム・システムによって生成されるエラーは、暗黙例外と呼ばれ、すべてのクラスまたはメソッド宣言の一部として明示的に処理する必要はありません。

たとえば、境界外配列例外は、配列を使用するが、エラーがその配列を使用するクラスまたはメソッドの宣言の一部とならなくてもよい場合に発生します。

それ以外のすべての例外は、明示例外です。呼び出されたメソッドがエラーをスローできる場合、例外スローのメソッドを使用してクラスがこのエラーを明示的に捕獲するか、このクラスがクラス宣言で生成する例外を識別し明示的にエラーをスローする必要があります。本質的に、明示例外は明示的に処理される必要があります。メソッドはスローする明示的エラーをすべて宣言するか、スローされる可能性がある明示的エラーをすべて捕獲する必要があります。

ランタイム以外の例外は、コンパイル時にチェックされます。Java は、コンパイル時にそうしたエラーをほとんど捕獲してから、コードを実行します。

すべての Java メソッドには代替実行パスが与えられるため、通常の方法では完了できない場合でも処理を完了できます。スローされているエラー・タイプが捕獲されない場合は、スタックの次のコード・ブロックまたはメソッドに渡されます。

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 Adaptive Server 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。

サポートされる Java と JDBC のバージョン

Java VM では、使用するプログラミング・インタフェースとして JDK 1.1、JDK 1.2、または JDK 1.3 を選択できます。提供されるバージョンは、JDK バージョン 1.1.8 と 1.3 です。

JDK のリリース 1.0 とリリース 1.1 の間で、いくつかの新しい API が導入されています。また、いくつかの API の使用が推奨されなくなりました。それらの API は将来のリリースではサポートされなくなる可能性があります。

推奨されていない API を使用する Java クラス・ファイルは、コンパイル時に警告を生成しますが、Adaptive Server Anywhere VM などリリース 1.1 標準で構築された Java 仮想マシンではまだ実行されます。

内部 JDBC ドライバは、JDBC のバージョン 2 をサポートします。

Java をサポートするデータベースの作成方法については、「データベースを Java 実行可能にする」102 ページを参照してください。

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、Adaptive Server 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、Adaptive Server 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。

ユーザ定義クラス

ユーザ定義クラスは、INSTALL JAVA 文を使用してデータベースにインストールされます。インストールすると、データベースの他のクラスで使用できるようになります。パブリック・クラスである場合は、ドメインとして SQL から使用できます。

クラスのインストールの詳細については、「Java クラスをデータベースにインストールする」109 ページを参照してください。

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。

たとえば、次の SQL 文は有効です。

```
CREATE VARIABLE str char(20);
 SET str = NEW java.lang.String( 'Brand new object' )
```

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

```
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 );
```

Adaptive Server Anywhere では標準出力がサーバ・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 ランタイム環境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[ ] ) { }
```

Java オブジェクトの機能のテストに役立つように、Sun Java ランタイム・システムの起動時に、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。

Adaptive Server 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。ほとんどの場合、この方法を使用すると、Java クラスの機能を柔軟にテストできます。

スコープと持続性

SQL 変数は、接続されている間だけ持続します。これは、旧バージョンの Adaptive Server Anywhere から変更されておらず、変数が Java クラスまたはネイティブ SQL データ型のどれであるかに影響を受けません。

Java クラスの持続性は、データベースのテーブルに似ています。テーブルは、データを保持しているかまたは使用されているかに関係なく、削除されるまでデータベースに存在します。データベースにインストールされた Java クラスも同様です。Java クラスは、REMOVE JAVA 文で明示的に削除するまで使用できます。

クラスの削除の詳細については、『ASA SQL リファレンス・マニュアル』>「REMOVE JAVA 文」を参照してください。

インストールされた Java クラスのクラス・メソッドは SQL 文からいつでも呼び出すことができます。次の文は、SQL 文を実行できる場所ならどこでも実行できます。

```
SELECT java.lang.Math.abs(-342)
```

Java オブジェクトは、2 つの形式、つまり変数値またはテーブルの値としてのみ使用できます。

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";
```

Adaptive Server Anywhere では、Java クラスで使用する場合に限り、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、『ASA SQL リファレンス・マニュアル』> 「文字列」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。しかし、Adaptive Server 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。SQL 文またはストアド・プロシージャで使用されるクラス名をすべて完全に修飾する必要があります。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。

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 はランタイムに無視される

classpath 環境変数は、Java オペレーションの実行中に Adaptive Server Anywhere の Java ランタイム環境に影響を与えません。これは、外部ファイルまたはアーカイブではなく、データベースにクラスが格納されるためです。

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、Adaptive Server 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。

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。Adaptive Server 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、Adaptive Server 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。

チュートリアル：データベースにおける Java の演習

このチュートリアルでは、SQL 文を使用して Java のクラスとオブジェクトに Java オペレーションを呼び出すための基本的な事柄について説明します。Java クラスをデータベースにインストールする方法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。このチュートリアルでは、「Java セミナー」71 ページで作成した Invoice クラスを使用します。

稼働条件

このチュートリアルは、データベース・ソフトウェアに Java をインストール済みであることを前提としています。また、Java コンパイラ (javac) など、Java Development Kit (JDK) のインストールも完了しているものとします。

Java をデータベースにインストールしていない場合は、次の SQL を使用できます。

```
ALTER DATABASE UPGRADE JAVA ON;
```

この操作を有効にするため、データベース・サーバを再起動してください。

データベースがない場合は、Sybase Central のウィザードや dbinit などを使用して、Java が有効なデータベースを初期化できます。

```
dbinit -ja mydb.db
```

リソース

この例のソース・コードとバッチ・ファイルは、SQL Anywhere フォルダの *samples¥ASA¥Java* フォルダにあります。

サンプル Java クラスの作成とコンパイル

最初の手順では、Java コードを記述して、それをコンパイルします。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 クラスを作成します。

サンプル・コードがインストールされます。便宜上、ここではサンプル・コードが含まれています。次のコードを *Invoice.java* に貼り付けます。

```
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;
    }
}
```

このクラスのソース・コードは、SQL Anywhere ディレクトリに *Samples¥ASA¥JavaInvoice¥Invoice.java* ファイルとして用意されています。

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。

コマンド・プロンプトで、*samples¥ASA¥Java* ディレクトリに変更し、次のコマンドを実行します。

```
javac -target 1.1 *.java
```

データベースの Java ランタイム・クラス (*rt.jar* または *classes.zip*) は、JDK によってインストールされる 1.1 ターゲットを使用して構築されます。1.1 以外のターゲットの場合、ClassFormatErrors が発生します。JDK バージョンを使用してデータベースにインストールするためのクラスをコンパイルする場合、Java コンパイラのオプション ″target″ を 1.1 に設定してください。

JDK 1.3 以前のバージョンを使用する場合、コンパイラのデフォルト・ターゲットは 1.1 になっています。

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。

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。この項では、両方の手順について説明します。該当する手順に従ってください。

❖ クラスをサンプル・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)。

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] フォルダを開いて [Java クラスの追加] をダブルクリックします。

   [Java クラスの作成] 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* を選択します。これは、SQL Anywhere インストール・フォルダの *Samples¥ASA¥Java* サブフォルダにあります。
4. [完了] をクリックしてウィザードを終了します。

❖ クラスをサンプル・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)。

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/samples/ASA/Java/Invoice.class'
   ```

   この場合、パスは SQL Anywhere フォルダです。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。

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、クラスを再インストール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」109 ページを参照してください。

Java オブジェクトのフィールドと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ユーザ定義の関数を作成します。

❖ 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. Java メソッドに引数を渡すには、次の SQL ラッパを作成します。

```
CREATE PROCEDURE init( IN arg1 CHAR(10),
                       IN arg2 DOUBLE,
                       IN arg3 CHAR(10),
                       IN arg4 DOUBLE)
 EXTERNAL NAME 'Invoice.init (Ljava/lang/
String;DLjava/lang/String;D)V'
 LANGUAGE JAVA;
```

   このコマンドの構文の詳細については、『ASA SQL リファレンス・マニュアル』>「CREATE PROCEDURE 文」を参照してください。

2. Java メソッドのリターン・コードを検索するには、次の SQL ラッパを作成します。

```
CREATE FUNCTION taxationrate()
 RETURNS DOUBLE
 EXTERNAL NAME        'Invoice.rateOfTaxation ()D'
 LANGUAGE JAVA;
 CREATE FUNCTION totalSum()
 RETURNS DOUBLE
 EXTERNAL NAME 'Invoice.totalSum ()D'
 LANGUAGE JAVA;
 CREATE FUNCTION linedesc1()
 RETURNS CHAR(10)
 EXTERNAL NAME
         'Invoice.getLineItem1Description ()Ljava/
lang/String;'
 LANGUAGE JAVA;
```

これらのコマンドの構文の詳細については、『ASA SQL リファレンス・マニュアル』> 「CREATE FUNCTION 文」を参照してください。

3 Java 属性のリターン・コードを検索するには、次の SQL ラッパを作成します。

```
CREATE FUNCTION linecost2()
 RETURNS DOUBLE
 EXTERNAL NAME
         'Invoice.getLineItem2Cost ()D'
 LANGUAGE JAVA;
```

メソッドまたは属性のシグニチャと名前の間にはスペースがあります。また、Java メソッドのシグニチャでは大文字と小文字が区別されます。

Java 機能の使用

ストアド・プロシージャと関数を使用してメソッドを呼び出し、SQL から属性にアクセスできるようになりました。

```
CALL init('prod1',10.00,'prod2',25.00);
 SELECT
taxationrate(),totalSum(),linedesc1(),linecost2();
```

このクエリは、次のような値がある 4 つのカラムを返します。

Taxationrate(*)	totalSum(*)	linedesc1(*)	linecost2(*)
.15	40.25	prod1	25

第 4 章

データベースにおける Java の使用

この章の内容

この章では、データベースに Java クラスを追加する方法とこれらのクラスをリレーショナル・データベースで使用する方法について説明します。

概要

この章では、データベースで Java を使用して次のタスクを実行する方法について説明します。

- **データベースを Java 実行可能にする** データベースで Java を使用できるようにするには特定の手順を踏む必要があります。
- **Java クラスをインストールする** Java クラスを Adaptive Server Anywhere で使用できるようにするには、データベースにインストールする必要があります。

Java のサンプルの設定

この章で掲載されている例の一部では、サンプル・データベースに JDBCExamples クラスを追加する必要があります。

次の 2 つの手順で Java のサンプルを設定します。

1. サンプル・データベースを Java 実行可能にします。Adaptive Server Anywhere データベースはデフォルトでは Java 実行可能ではありません。
2. JDBCExamples クラスをデータベースに追加します。

❖ サンプル・データベースを Java 実行可能にするには、次の手順に従います。

1 Sybase Central を起動し、サンプル・データベース (ASA 9.0 Sample ODBC データ・ソース) に接続します。asademo9 データベース・サーバが asademo データベースとともに表示されます。

2 Sybase Central の左ウィンドウ枠で asademo データベースを右クリックし、ポップアップ・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　[データベース・アップグレード] ウィザードの指示に従います。JDK バージョン 1.3 で [Java サポートをインストール] オプションを選択します。

4　サンプル・データベースを再起動します。

[データベースのアップグレード] ウィザードが完了したら、接続を切断し、サンプル・データベースが停止していることを確認します。データベースを停止して再起動してから Java サポートを使用してください。

5　Java サポートが追加されていることを確認します。

- Sybase Central で、サンプル・データベースに接続します。
- Sybase Central の左ウィンドウ枠で asademo データベースを右クリックし、ポップアップ・メニューから [プロパティ] を選択します。
- Java JDK バージョンが 1.3 に設定されていることを確認します。

❖ JDBCExamples クラスをサンプル・データベースするには、次の手順に従います。

1　Sybase Central を起動し、サンプル・データベース (ASA 9.0 Sample ODBC データ・ソース) に接続します。asademo9 データベース・サーバが asademo データベースとともに表示されます。

2　Sybase Central の左ウィンドウ枠で、[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] ー [Java クラス] を選択します。[新しい Java クラスの作成] ウィザードが表示されます。

4　[参照] をクリックし、SQL Anywhere インストールの *Samples¥ASA¥Java* サブディレクトリにある JDBCExamples.class を見つけます。

5 [OK] をクリックし、[完了] をクリックしてインストールを完了します。

Java Runtime Environment の管理

Java Runtime Environment は、次のもので構成されています。

- **Adaptive Server Anywhere Java 仮想マシン (VM)** Adaptive Server Anywhere Java 仮想マシンをデータベース・サーバ内で実行すると、コンパイル済みの Java クラス・ファイルを翻訳して、実行します。
- **ランタイム Java クラス** データベースを作成すると、一連の Java クラスがデータベースで使用可能になります。データベースで Java アプリケーションを使用するには、ランタイム・クラスが正常に動作している必要があります。

Java の管理タスク

Java Runtime Environment を使用可能にするには、次のタスクを実行する必要があります。

- **データベースを Java 実行可能にする** このタスクでは、組み込みクラスが使用可能なことと、データベースがバージョン 9 にアップグレードされていることを確認します。

  詳細については、「データベースを Java 実行可能にする」102 ページを参照してください。

- **ユーザが必要とする他のクラスをインストールする** このタスクでは、ランタイム・クラス以外のクラスがインストールされていて、それが最新のものであることを確認します。

  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」109 ページを参照してください。

- **サーバを設定する** Java タスクを実行するために必要なメモリを使用できるように、サーバを設定します。

  詳細については、「Java 用のメモリ設定」123 ページを参照してください。

Java を管理するためのツール

Sybase Central または Interactive SQL から、これらすべてのタスクを実行できます。

データベースを Java 実行可能にする

Adaptive Server Anywhere の Java Runtime Environment では、Java VM と「**Adaptive Server Anywhere ランタイム Java クラス**」が必要です。ランタイム Java クラスを使用できるように、データベースを Java 実行可能にする必要があります。

データベース内の Java は、SQL Anywhere Studio のコンポーネントですが、別途ライセンスが必要です。

新しいデータベースはデフォルトで Java 実行可能にならない
Adaptive Server Anywhere で作成したデータベースは、デフォルトでは Java 実行可能になりません。

Java は単一階層言語です。すなわち、実際に作成または使用するすべてのクラスは 1 つのクラスから継承します。このことは、低レベルのクラス (階層内の上位クラス) が存在して初めて、高レベルのクラスが使用可能になることを意味します。Java アプリケーションを実行するために必要なクラスの基本セットは、ランタイム Java クラスまたは Java API です。

データベースを Java 実行可能にしない場合

Java 実行可能なデータベースは、多数のエントリをシステム・テーブルに追加します。これにより、Java 機能をまったく使用しなくても、データベース・サイズが追加され、さらに、データベースを稼働するためのメモリの必要量が約 200 K 増えます。

Java を使用しない場合、また限られたメモリ環境で作業している場合は、データベースを Java 実行可能にしないこともあります。

Adaptive Server Anywhere ランタイム Java クラス

Adaptive Server Anywhere ランタイム Java クラスは、他のクラスのようにデータベースに保管されるのではなく、ディスク上に保持されます。以下のファイルには、Adaptive Server Anywhere ランタイム Java クラスが入っています。各ファイルは、SQL Anywhere ディレクトリの *Java* サブディレクトリにあります。

- **1.1\classes.zip**　このファイルは、Sun Microsystems からライセンスを受けたもので、JDK 1.1.8 用の Sun Microsystems Java ランタイム・クラスのサブセットを含んでいます。
- **1.3\rt.jar**　このファイルは、Sun Microsystems からライセンスを受けたもので、JDK 1.3 用の Sun Microsystems Java ランタイム・クラスのサブセットを含んでいます。
- **asajdbc.zip**　このファイルは、JDK 1.1 用の Adaptive Server Anywhere の内部 JDBC ドライバ・クラスを含んでいます。
- **asajrt12.zip**　このファイルは、JDK 1.2 と JDK 1.3 用の Adaptive Server Anywhere の内部 JDBC ドライバ・クラスを含んでいます。

データベースを Java 使用可能にする際は、システム JAR ファイルの使用可能なクラス一覧を使用してシステム・テーブルの更新も行います。その後、Sybase Central からクラス階層を検索できますが、クラス自体はデータベースに存在しません。

JAR ファイル

データベースは、次の JAR ファイルにランタイム・クラス名を保管します。

- **ASACIS**　リモート・データ・アクセスに必要なクラスを保持します。
- **ASAJDBCDRV**　*jdbcdrv.zip* のクラス名を保持します (com.sybase.jdbc パッケージ)。
- **ASAJIO**
- **ASAJRT**　*asajdbc.zip* のクラス名を保持します。
- **ASASystem**　*classes.zip* のクラス名を保持します。
- **ASASystemUNIX**　*classes.zip* のクラス名を保持します。

インストールされたパッケージ

ランタイム・クラスには、次のパッケージがあります。

- **java**　ここに保管されているパッケージには、サポートされている Sun Microsystems の Java ランタイム・クラスが含まれます。

- **com.sybase**　ここに保管されているパッケージからは、サーバ側の JDBC サポートが提供されます。
- **sun**　ここに保管されているパッケージは、Sun Microsystems から提供されます。
- **sybase.sql**　ここに保管されているパッケージは、サーバ側の JDBC サポートの一部です。

警告：別のバージョンの Sun JDK からクラスをインストールしないでください。

Sun の JDK にあるクラスは、Java オペレーションを実行するデータベースにインストールされる Adaptive Server Anywhere ランタイム Java クラスと名前を共有します。

Adaptive Server Anywhere に含まれている *classes.zip* ファイルを置き換えないでください。これらのクラスの別バージョンを使用すると、Adaptive Server Anywhere Java VM との互換性の問題が生じます。

この項で説明するメソッドでは、データベースを Java 実行可能にするだけにしてください。

データベースを Java 実行可能にする方法

データベースは、作成時でもアップグレード時でも、または後の別の操作でも Java 実行可能にできます。

データベースの作成

Java 実行可能なデータベースを作成するには、次のものを使用します。

- CREATE DATABASE 文

  構文の詳細については、『ASA SQL リファレンス・マニュアル』>「CREATE DATABASE 文」を参照してください。

- *dbinit* ユーティリティ

詳細については、『ASA データベース管理ガイド』> 「dbinit コマンド・ライン・ユーティリティを使用したデータベースの作成」を参照してください。

- Sybase Central

  詳細については、『ASA SQL ユーザーズ・ガイド』> 「データベースの作成」を参照してください。

データベースのアップグレード

データベースを Java 実行可能なバージョン 9 のデータベースにアップグレードするには、次のものを使用します。

- ALTER DATABASE 文

  構文の詳細については、『ASA SQL リファレンス・マニュアル』> 「ALTER DATABASE 文」を参照してください。

- *dbupgrad.exe* アップグレード・ユーティリティ

  詳細については、『ASA データベース管理ガイド』> 「dbupgrad コマンド・ライン・ユーティリティを使用したデータベースのアップグレード」を参照してください。

- Sybase Central

  詳細については、「データベースを Java 実行可能にする」107 ページを参照してください。

データベースに Java をインストールしなくても、Java オペレーションを含まないデータベース・オペレーションはすべて完全に機能し、動作します。

新しいデータベースと Java

デフォルトでは、データベースを作成するたびに Adaptive Server Anywhere が Adaptive Server Anywhere ランタイム Java クラスをインストールすることはありません。この別途ライセンスが入手可能なコンポーネントをインストールするかどうかはオプションであり、データベースの作成方法によって決まります。

CREATE DATABASE オプション

CREATE DATABASE SQL 文には、JAVA というオプションがあります。データベースで Java を有効にするには、このオプションを ON に設定します。Java を無効にするには、このオプションを OFF に設定します。このオプションのデフォルト設定は OFF です。

たとえば、次の文は、Java 実行可能なデータベース・ファイル *temp.db* を作成します。

```
CREATE DATABASE 'c:¥¥sybase¥¥asa9¥¥temp' JAVA ON
```

次の文は、Java をサポートしないデータベース・ファイル *temp2.db* を作成します。

```
CREATE DATABASE 'c:¥¥sybase¥¥asa9¥¥temp2'
```

データベース初期化ユーティリティ

データベースは、*dbinit.exe* データベース初期化ユーティリティを使用して作成できます。このユーティリティには、新しく作成したデータベースにランタイム Java クラスをインストールするかどうかを制御するオプションがあります。デフォルトでは、これらのクラスはインストールされません。

Sybase Central を使用してデータベースを作成するときも、同じオプションを使用できます。

データベースと Java のアップグレード

このソフトウェアの以前のバージョンで作成した既存のデータベースは、アップグレード・ユーティリティまたは ALTER DATABASE 文を使用してアップグレードできます。

データベース・アップグレード・ユーティリティ

データベースは、*dbupgrad.exe* ユーティリティを使用して、Adaptive Server Anywhere バージョン 9 標準版にアップグレードできます。`-jr` アップグレード・ユーティリティ・オプションを使用すると、Adaptive Server Anywhere ランタイム Java クラスはインストールされません。

データベース内の Java がアップグレード後のデータベースに組み込まれるための条件については、『ASA データベース管理ガイド』>「dbupgrad コマンド・ライン・ユーティリティを使用したデータベースのアップグレード」を参照してください。

データベースを Java 実行可能にする

データベースの作成時、またはデータベースを標準版にアップグレードしたときに、データベースを Java 実行可能にするように選択しなかった場合は、Sybase Central または Interactive SQL を使用して、後から必要な Java クラスを追加できます。

❖ Java ランタイム・クラスをデータベースに追加するには、次の手順に従います (Sybase Central の場合)。

1. DBA 権限を持つユーザとして、Sybase Central からデータベースに接続します。
2. データベースを右クリックして、[データベースのアップグレード] を選択します。
3. ウィザードの概要ページで [次へ] をクリックします。
4. アップグレードするデータベースをリストから選択します。
5. 必要であれば、データベースのバックアップ作成も選択できます。[次へ] をクリックします。
6. 必要であれば、jConnect メタ情報のサポートのインストールも選択できます。[次へ] をクリックします。
7. [Java サポートをインストール] オプションを選択します。インストールする JDK のバージョンも選択します。デフォルトのクラスは JDK 1.3 のクラスです。
8. ウィザードの指示に従います。

❖ Java ランタイム・クラスをデータベースに追加するには、次の手順に従います (SQL の場合)。

1. DBA 権限を持つユーザとして Interactive SQL からデータベースに接続します。
2. 次の文を実行します。

```
ALTER DATABASE UPGRADE JAVA ON
```

詳細については、『ASA SQL リファレンス・マニュアル』>「ALTER DATABASE 文」を参照してください。

3 Java サポートが有効になるように、データベースを再起動します。

Sybase Central を使用してデータベースを Java 実行可能にする

Sybase Central では、ウィザードを使用してデータベースを作成できます。データベースを作成またはアップグレードすると、Adaptive Server Anywhere ランタイム Java クラスをインストールするかどうか選択するようウィザードで要求されます。デフォルトでは、このオプションはデータベースを Java 実行可能にします。

Sybase Central を使用して、次の方法でデータベースを作成またはアップグレードできます。

- [ファイル] ー [データベースの作成] を選択し、新しいデータベースを作成します。
- 左ウィンドウ枠でデータベース・サーバをクリックし、右ウィンドウ枠で [ユーティリティ] タブをクリック、[データベースのアップグレード] をダブルクリックして、旧バージョンのソフトウェアのデータベースから Java 機能を持つデータベースにアップグレードします。

Java クラスをデータベースにインストールする

Java クラスをデータベースにインストールする前に、必ずコンパイルしてください。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* がついています。Adaptive Server 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。

この項では、コンパイルした Java クラスをインストールする方法について説明します。クラスまたは JAR をインストールするには、DBA 権限が必要です。

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.exe* が含まれています。

Java 実行可能なデータベースのみ
コンパイルされた Java クラス・ファイルはすべてデータベースにインストールできます。ただし、インストールされたクラスを使用する Java オペレーションを実行できるのは、「データベースを Java 実行可能にする」102 ページで述べたようにデータベースを Java 実行可能にしたときのみです。

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 権限を使用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] ー [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 権限を使用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 には、クラス・ファイルが保持されるディレクトリ、*ClassName.class* にはクラス・ファイル名を指定します。

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、次の文を入力します。

```
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、『ASA SQL リファレンス・マニュアル』>「INSTALL JAVA 文」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、『ASA SQL リファレンス・マニュアル』> 「INSTALL JAVA 文」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 権限を使用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 権限を使用してデータベースに接続します。

2 次の文を入力します。

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

クラスと Jar 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 文を入力すると、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。一度仮想マシン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ています。

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。

なぜ更新したクラスがこの方法で有効となるかを理解するには、VM の働きについて少し理解する必要があります。詳細については、「Java 用のメモリ設定」123 ページを参照してください。

❖ 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 権限を使用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 ファイルを探します。
4. そのクラスまたは JAR ファイルを右クリックし、ポップアップ・メニューから [更新] を選択します。
5. 表示されるダイアログで、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 権限を使用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。

詳細については、『ASA SQL リファレンス・マニュアル』>「INSTALL JAVA 文」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。

サポートされるクラス

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。

サポートされるパッケージの詳細については、「サポートされている Java パッケージ」126 ページを参照してください。

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、SQL Anywhere ディレクトリの *Samples¥ASA¥Java¥JDBCExamples.java* ファイルにある **JDBCExamples** クラスは、main メソッドを持ちます。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 JDBCExamples
```

JDBCExamples クラスの実行方法の詳細については、「JDBC 接続の確立」146 ページを参照してください。

❖ 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。

詳細については、『ASA SQL リファレンス・マニュアル』> 「CREATE PROCEDURE 文」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。

文字列配列の各メンバは、CHAR または VARCHAR データ型にするか、リテラル文字列にします。

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。各 Java スレッドはエンジン・スレッドで、スレッドの数は -gn データベース・サーバ・オプションによって許可された数になります。

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。

データベース・サーバ・スレッドの詳細については、『ASA データベース管理ガイド』> 「-gn サーバ・オプション」を参照してください。

JDBC 呼び出しの直列化

サーバ側 JDBC ドライバへの呼び出しはすべて直列化されているため、一度にアクティブに JDBC を実行するスレッドは 1 つのみです。

「プロシージャは見つかりません」エラー

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、サーバからは「プロシージャは見つかりません」というエラーが返されます。引数の数と型を確認してください。

Java メソッドから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[]** パラメータのひとつに割り当てます。
4. EXTERNAL NAME LANGUAGE JAVA 型の SQL ストアド・プロシージャを作成します。この型のプロシージャは、Java メソッドのラッパーです。結果セットを返す他のプロシージャと同じ方法で、SQL プロシージャの結果セット上でカーソルを使用することができます。

   Java メソッドのラッパーであるストアド・プロシージャの構文の詳細については、『ASA SQL リファレンス・マニュアル』>「CREATE PROCEDURE 文」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 CAST( JName.lastName " +
                           "AS CHAR( 50 ) )" +
                           "FROM jdba.contact " );
        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 ASA 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、『ASA SQL リファレンス・マニュアル』> 「CREATE PROCEDURE 文」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 メソッドのラッパーとして使用できます。この項では、ストアド・プロシージャ内で OUT または INOUT パラメータを利用する Java メソッドの記述方法について説明します。

Java は、INOUT または OUT パラメータの明示的なサポートはしていません。ただし、パラメータの配列は使用できます。たとえば、整数の OUT パラメータを使用するには、1 つの整数だけの配列を作成します。

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

次のプロシージャでは、**testOut** メソッドを使用しています。

```
CREATE PROCEDURE sp_testOut ( OUT p INTEGER )
 EXTERNAL NAME 'TestClass/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。

メソッドのシグニチャを含む構文の詳細については、『ASA SQL リファレンス・マニュアル』> 「CREATE PROCEDURE 文」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。Adaptive Server Anywhere には、データベース内の Java セキュリティ・マネージャ用に次のサポートが用意されています。

- Adaptive Server Anywhere によるデフォルトのセキュリティ・マネージャ。

- 独自のセキュリティ・マネージャを使用できる。

  詳細については、「独自のセキュリティ・マネージャの実装」121 ページを参照してください。

デフォルトのセキュリティ・マネージャ

デフォルトのセキュリティ・マネージャは、**com.sybase.asa.jrt.SAGenericSecurityManager** クラスです。このクラスでは次のタスクが実行されます。

1. データベース・オプション JAVA_INPUT_OUTPUT の値を確認します。
2. データベース・サーバが -sc データベース・サーバ・オプションを使用して C2 セキュリティ・モードで起動されたかどうかを確認します。
3. 接続プロパティが OFF の場合は、Java ファイル I/O 機能へのアクセスを禁止します。
4. データベース・サーバが C2 セキュリティ・モードで動作している場合は、**java.net** パッケージへのアクセスを禁止します。
5. セキュリティ・マネージャは、ユーザに対して機能へのアクセスを禁止すると、**java.lang.SecurityException** を返します。

詳細については、『ASA データベース管理ガイド』> 「JAVA_INPUT_OUTPUT オプション [データベース]」と『ASA データベース管理ガイド』> 「-sc サーバ・オプション」を参照してください。

デフォルトのセキュリティ・マネージャを使用した Java ファイル I/O の制御

Java ファイル I/O は、JAVA_INPUT_OUTPUT データベース・オプションを通じて制御されます。デフォルトでは、このオプションは OFF に設定され、ファイル I/O は禁止されます。

❖ デフォルトのセキュリティ・マネージャを使用してファイル・アクセスを許可するには、次の手順に従います。

- JAVA_INPUT_OUTPUT オプションを ON に設定します。

```
SET OPTION JAVA_INPUT_OUTPUT='ON'
```

独自のセキュリティ・マネージャの実装

独自のセキュリティ・マネージャを実装するには、いくつかの手順があります。

❖ 独自のセキュリティ・マネージャを使用するには、次の手順に従います。

1　**java.lang.SecurityManager** を拡張するクラスを実装します。

SecurityManager クラスには、特定のアクションが許可されているかどうかを確認するための多数のメソッドがあります。アクションが許可されている場合、メソッドは何も返しません。メソッドが値を返すと、**SecurityException** がスローされます。

許可するアクションを制御するメソッドを、値を返さないメソッドで上書きしてください。そのためには、`public void` メソッドを実装します。

2　セキュリティ・マネージャに適切なユーザを割り当てます。

add_user_security_manager、update_user_security_manager、delete_user_security_manager の各システム・ストアド・プロシージャを使用して、セキュリティ・マネージャをユーザに割り当てます。たとえば、MySecurityManager クラスをユーザのセキュリティ・マネージャとして割り当てるには、次のコマンドを実行します。

```
call dbo.add_user_security_manager(
   user_name, 'MySecurityManager', NULL )
```

例

次のクラスでは、ファイルからの読み込みは許可されますが、書き込みは禁止されます。

```
public class MySecurityManager extends SecurityManager
 { public void checkRead(FileDescriptor) {}
   public void checkRead(String) {}
   public void checkRead(String, Object) {}
 }
```

SecurityManager.checkWrite メソッドは上書きされず、ファイルの書き込み操作は禁止されます。**checkRead** メソッドは値を返さず、アクションは許可されます。

Java 用のメモリ設定

この項では、データベースで Java を実行するためのメモリ要件と、その要件を満たすサーバのセット・アップ方法について説明します。

Java VM では、大容量のキャッシュ・メモリを必要とします。

キャッシュのチューニングの詳細については、『ASA SQL ユーザーズ・ガイド』>「パフォーマンス向上へのキャッシュの使用」を参照してください。

データベースと接続レベルの要件

Java VM は、データベースごとと、接続ごとにメモリを使用します。

- データベースごとの要件は「**再配置**」できません。つまり、ディスクにページアウトできません。この要件は、サーバ・キャッシュに配置します。このメモリのタイプはサーバ用ではなく、各データベース用です。キャッシュ要件を見積もる場合、サーバで実行する各データベースのメモリ要件を合計します。
- 接続ごとの要件は、ユニットとしてのみ、再配置できます。1 つの接続の要件は、すべてキャッシュ内にあるか、テンポラリ・ファイル内にあります。

メモリの使用法

データベースの Java は次の用途でメモリを必要とします。

- サーバを実行するときに最初に Java を使用すると、VM がメモリにロードされますが、このときに 1.5 MB 以上のメモリが必要となります。これは、データベース全体の要件の一部です。Java を使用するデータベースごとに追加の VM がロードされます。
- Java を使用する接続ごとに、VM の新しいインスタンスがロードされます。新しいインスタンスは、1 つの接続につき約 200 K のメモリを必要とします。

- Java アプリケーションで使用される各クラス定義は、メモリにロードされます。これは、データベース全体のメモリに保持されるので、個々の接続に対して個別にコピーする必要はありません。
- 各接続は、Java 変数とアプリケーション・スタック領域 (メソッドの引数などに使用される) の作業セットを必要とします。

メモリの管理

メモリの使用について、次の方法で管理できます。

- **全体のキャッシュ・サイズを設定する** 再配置不可能メモリのすべての要件を十分に満たすキャッシュ・サイズを使用します。

  キャッシュ・サイズは、サーバ起動時に、-c オプションを使用して設定されます。

  ほとんどの場合、キャッシュ・サイズは 8 MB で十分です。

- **ネームスペース・サイズを設定する** Java のネームスペース・サイズは、データベースごとのメモリ割り付けの最大サイズをバイトで表したものです。

  これは、JAVA_NAMESPACE_SIZE オプションを使用して設定できます。このオプションはグローバルで、DBA 権限を持つユーザだけが設定できます。

- **ヒープ・サイズを設定する** この JAVA_HEAP_SIZE オプションは、接続ごとのメモリの最大サイズをバイトで設定します。

  このオプションは、個々の接続ごとに設定できますが、他のユーザが利用可能なメモリに影響を及ぼすため、DBA 権限を持つユーザだけが設定できます。

VM の起動と停止

Java にメモリ・パラメータを設定する以外に、Java を使用しないときに STOP JAVA 文を使用して 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。この構文は次のように簡単なものです。

```
STOP JAVA
```

VM は、Java オペレーションが実行されるときに必ずロードされます。Java オペレーションを実行する準備として、明示的に VM をロードする場合は、次の文を実行します。

```
START JAVA
```

Java クラスのリファレンス

この項では、Adaptive Server Anywhere でサポートされている JDK クラスおよびパッケージに関するリファレンス項目について説明します。ユーザ定義のクラスとパッケージは、DBA 権限を持つユーザによってデータベースにインストールされてから、使用できます。

サポートされている Java パッケージ

この項では、Java 実行可能のデータベースで使用できる組み込みクラスのパッケージについて説明します。サポートされていない、または部分的にサポートされているパッケージ内のクラスについては、「サポートされていない Java パッケージとクラス」127 ページと「部分的にサポートされるパッケージとクラス」128 ページを参照してください。

ここに記載されていないパッケージ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。

- java.beans
- java.io ― ファイル・アクセスを制御するクラスは、特定の Windows オペレーション・システムにおいてのみ、JAVA_INPUT_OUTPUT オプションが ON に設定されている場合にかぎりサポートされます。『ASA データベース管理ガイド』>「JAVA_INPUT_OUTPUT オプション [データベース]」を参照。
- java.lang
- java.lang.reflect
- java.lang.Thread
- java.math
- java.net
- java.net.PlainDatagramSocketImpl
- java.rmi

- java.rmi.dgc
- java.rmi.registry
- java.rmi.server
- java.security
- java.security.acl
- java.security.interfaces
- java.SQL。JDBC 2.0 機能の詳細については、『ASA プログラミング・ガイド』> 「データベース内の JDBC の機能」を参照してください。
- java.text
- java.util
- java.util.zip

サポートされていない Java パッケージとクラス

次のパッケージのクラスは Adaptive Server 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` など。

部分的にサポートされるパッケージとクラス

次のクラスは*部分的にサポート*されています。これらのクラスにはサポートされていないネイティブ・メソッドがいくつか含まれています。

- java.lang.ClassLoader
- java.lang.Compiler
- java.lang.Runtime (exec/load/loadlibrary)
- java.io.File
- java.io.FileDescriptor
- java.io.FileInputStream
- java.io.FileOutputStream
- java.io.RandomAccessFile
- java.util.zip.Deflater
- java.util.zip.Inflater

第 5 章

JDBC プログラミング

この章の内容

この章では、JDBC を使用してデータにアクセスする方法について説明します。

JDBC はクライアント・アプリケーションからとデータベース内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 の概要

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。

JDBC データベース・インタフェースについて詳しくは解説しませんが、この章では簡単な例を挙げて JDBC の概要を説明し、クライアント側とデータベース側で JDBC を使用する方法を説明します。

それぞれの例では、Adaptive Server 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。

JDBC と Adaptive Server Anywhere

Adaptive Server 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 Adaptive Server 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 Studio には、pure Java アプリケーション対応 jConnect ドライバと、タイプ 2 JDBC ドライバである iAnywhere JDBC ドライバの 2 つの JDBC ドライバが用意されています。

  この章では、「**クライアント・アプリケーション**」というフレーズは、ユーザのマシン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。

- **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

- **必要なソフトウェア** Sybase jConnect ドライバを使用するには、TCP/IP が必要です。

  Sybase jConnect ドライバがすでに使用可能になっているかどうかは Adaptive Server Anywhere のインストール環境によって異なります。

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」138 ページを参照してください。

- **ソース・コードのサンプル**　この章で示したソース・コードのサンプルは、SQL Anywhere ディレクトリの *Samples¥ASA¥Java¥JDBCExamples.java* ファイルにあります。

  JDBCExamples クラスなどの Java のサンプルを設定する方法の詳細については、「Java のサンプルの設定」98 ページを参照してください。

JDBC ドライバの選択

Adaptive Server Anywhere では、次の 2 種類の JDBC ドライバが提供されています。

- **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 Adaptive Server Anywhere と通信します。

  jConnect のマニュアルについては、http://sybooks.sybase.com/jc.html を参照してください。

- **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 Adaptive Server 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。

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**　どちらのドライバも JDK 2 に準拠しています。iAnywhere JDBC ドライバは完全なスクロール可能カーソルを提供していますが、これは jConnect にはありません。

- **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、Adaptive Server 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。

どちらのドライバも、Windows 95/98/Me と Windows NT/2000/XP で使用でき、UNIX と Linux オペレーティング・システムでもサポートされています。NetWare または Windows CE では使用できません。

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される 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 (true/false) が返され、前に送るローが実際に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。**getDate** 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たり、操作やその他のクエリなどで独自に利用するデータを取得したりすることができます。

データベース内の JDBC の機能

データベース内で Java から使用できる JDBC のバージョンは、データベース用に設定されている JDK バージョンによって決まります。

- データベースが JDK 1.2 または JDK 1.3 を使用して初期化されている場合は、JDBC 2.0 API を使用できます。

  データベースを JDK 1.2 または JDK 1.3 にアップグレードする方法については、『ASA SQL リファレンス・マニュアル』> 「ALTER DATABASE 文」または『ASA データベース管理ガイド』> 「dbupgrad コマンド・ライン・ユーティリティを使用したデータベースのアップグレード」を参照してください。

- データベースが JDK 1.1 を使用して初期化されている場合は、JDBC 1.2 の機能を使用できます。JDK 1.1 用の内部 JDBC ドライバ (asajdbc) によって JDBC 2.0 の機能の一部がサーバ側 Java アプリケーションから使用可能になりますが、JDBC 2.0 への完全対応はしていません。

  詳細については、「JDK 1.1 データベースから JDBC 2.0 の機能を使用する」133 ページを参照してください。

JDK 1.1 データベースから JDBC 2.0 の機能を使用する

この項では、JDK 1.1 のサポートを使用して初期化されたデータベースから JDBC 2.0 の機能にアクセスする方法について説明します。さまざまな用途を考慮して、データベース内の Java のバージョンを 1.3 にアップグレードすることをおすすめします。

JDK 1.1 のサポートを使用して初期化されたデータベースの場合は、**sybase.sql.ASA** パッケージに JDBC 2.0 の一部の機能があります。これらの JDBC 2.0 の機能を使用するには、**java.sql** パッケージではなく **sybase.sql.ASA** パッケージの対応クラスに JDBC オブジェクトをキャストしてください。**java.sql** と宣言されたクラスは、JDBC 1.2 の機能のみに制限されます。

sybase.sql.ASA のクラスは、次のとおりです。

JDBC クラス	Subase 内部ドライバのクラス
java.sql.Connection	sybase.sql.ASA.SAConnection
java.sql.Statement	sybase.sql.ASA.SAStatement
java.sql.PreparedStatement	sybase.sql.ASA.SAPreparedStatement
java.sql.CallableStatement	sybase.sql.ASA.SACallableStatement
java.sql.ResultSetMetaData	sybase.sql.ASA.SAResultSetMetaData
java.sql.ResultSet	sybase.sql.SAResultSet
java.sql.DatabaseMetaData	sybase.sql.SADatabaseMetaData

次の機能は準備文の **ResultSetMetaData** オブジェクトですが、このとき **ResultSet** は要求されず、文も実行されません。これは、この機能が JDBC 1.2 標準の一部でないためです。

```
ResultSetMetaData
sybase.sql.ASA.SAPreparedStatement.describe()
```

次のコードは、結果セット内の前のローをフェッチしますが、これは JDBC 1.2 ではサポートされない機能です。

```
import java.sql.*;
 import sybase.sql.asa.*;
 ResultSet rs;
 // more code here
 ( ( sybase.sql.asa.SAResultSet)rs ).previous();
```

JDBC 2.0 の制限

次のクラスは JDBC 2.0 コア・インタフェースの一部ですが、**sybase.sql.ASA** パッケージでは使用できません。

- java.sql.Blob
- java.sql.Clob
- java.sql.Ref
- java.sql.Struct
- java.sql.Array
- java.sql.Map

次の JDBC 2.0 コア・ファンクションは、**sybase.sql.ASA** パッケージでは使用できません。

sybase.sql.ASA のクラス	使用不可能なファンクション
SAConnection	java.util.Map getTypeMap()
	void setTypeMap(java.util.Map map)
SAPreparedStatement	void setRef(int pidx, java.sql.Ref r)
	void setBlob(int pidx, java.sql.Blob b)
	void setClob(int pidx, java.sql.Clob c)
	void setArray(int pidx, java.sql.Array a)
SACallableStatement	Object getObject(pidx, java.util.Map map)
	java.sql.Ref getRef(int pidx)
	java.sql.Blob getBlob(int pidx)
	java.sql.Clob getClob(int pidx)
	java.sql.Array getArray(int pidx)

sybase.sql.ASA のクラス	使用不可能なファンクション
SAResultSet	Object getObject(int cidx, java.util.Map map)
	java.sql.Ref getRef(int cidx)
	java.sql.Blob getBlob(int cidx)
	java.sql.Clob getClob(int cidx)
	java.sql.Array getArray(int cidx)
	Object getObject(String cName, java.util.Map map)
	java.sql.Ref getRef(String cName)
	java.sql.Blob getBlob(String cName)
	java.sql.Clob getClob(String cName)
	java.sql.Array getArray(String cName)

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 Sybase jConnect JDBC ドライバまたは iAnywhere 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** の値が **DriverManager.getConnection** に渡され、その結果、JDBC アプリケーションは現在のユーザ接続内で動作できるようになります。これは簡単で効率が良く、安全な操作です。それは、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。内部 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。

このような方法で、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、外部接続にはマシン名とポート番号が必要です。

jConnect JDBC ドライバの使用

クライアント・アプリケーションまたはアプレットから JDBC を使用する場合は、jConnect JDBC ドライバを介して Adaptive Server Anywhere データベースに接続してください。

jConnect は SQL Anywhere Studio に付属しています。Adaptive Server Anywhere が入っていたパッケージによって、jConnect が含まれている場合と含まれていない場合があります。クライアント・アプリケーションから JDBC を使用するには、jConnect が必要です。データベース内では、jConnect がなくても JDBC を使用できます。

jConnect のマニュアルについては、http://sybooks.sybase.com/jc.html を参照してください。

jConnect ドライバのファイル

jConnect JDBC ドライバは、*Sybase¥Shared* ディレクトリの一連のサブディレクトリにインストールされます。提供される jConnect には、次の 2 つのバージョンがあります。

- **jConnect 4.5** このバージョンの jConnect は、JDK 1.1 アプリケーションの開発用です。jConnect 4.5 は *Sybase¥Shared¥jConnect-4_5* ディレクトリにインストールされます。

  jConnect 4.5 は一連のクラスとして提供されます。

- **jConnect 5.5** このバージョンの jConnect は、JDK 1.2 以降のアプリケーションの開発用です。jConnect 5.5 は *Sybase¥Shared¥jConnect-5_5* ディレクトリにインストールされます。

  jConnect 5.5 は、JAR ファイル *jconn2.jar* として提供されます。

この章の例では、jConnect 5.5 を使用しています。jConnect 4.5 を使用している場合は、適切な置き換えが必要です。

jConnect 用 CLASSPATH 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 5.5 ドライバを追加します。*path* は *Sybase¥Shared* ディレクトリです。

```
set classpath=%classpath%;path¥jConnect-
5_5¥classes¥jconn2.jar
```

次のコマンドは、既存の CLASSPATH 環境変数に jConnect 4.5 ドライバを追加します。

```
set classpath=%classpath%;path¥jConnect-4_5¥classes
```

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 5.5 を使用する場合、アプリケーションは必ず **com.sybase.jdbc2.jdbc** 内のクラスにアクセスします。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.jdbc2.jdbc.*
```

jConnect 4.5 を使用する場合、各クラスは **com.sybase.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. 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、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。

デフォルトでは、jConnect システム・オブジェクトは新しいデータベースに追加されます。jConnect オブジェクトのデータベースへの追加は、作成／更新時やその後でも選択できます。

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 権限を持つユーザとして、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右クリックし、ポップアップ・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。

   [データベースのアップグレード] 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 (Interactive SQL の場合)。

- DBA 権限を持つユーザとして Interactive SQL からデータベースに接続し、[SQL 文] ウィンドウ枠に次のコマンドを入力します。

  ```
  read path¥scripts¥jcatalog.sql
  ```

  この場合、*path* は SQL Anywhere ディレクトリです。

ヒント

コマンド・プロンプトを使用して、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。コマンド・プロンプトで次のように入力します。

```
dbisql -c "uid=user;pwd=pwd" path¥scripts¥jcatalog.sql
```

user と *pwd* は DBA 権限を持つユーザです。*path* は SQL Anywhere 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。

```
Class.forName("com.sybase.jdbc2.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。

サーバの 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 (Uniform Resource Locator) を指定する必要があります。「jConnect を使用した JDBC クライアント・アプリケーションからの接続」146 ページで述べた例では、次のようになります。

```
StringBuffer temp = new StringBuffer();
 // Use the jConnect driver...
 temp.append("jdbc:sybase:Tds:");
 // to connect to the supplied machine name...
 temp.append(_coninfo);
 // on the default port number for ASA...
 temp.append(":2638");
 // and connect.
 System.out.println(temp.toString());
 conn = DriverManager.getConnection(temp.toString() ,
_props );
```

URL は次のように構成されます。

```
jdbc:sybase:Tds:machine-name:port-number
```

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。
- **machine-name**　サーバが動作しているマシンの IP アドレスまたは名前。同じマシンと接続を確立している場合、現在のマシンを意味する **localhost** を使用できます。

- **port number** データベース・サーバが受信しているポート番号。Adaptive Server 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 Adaptive Server Anywhere 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。前述の URL はサーバを指定しますが、データベースは指定しません。接続は、サーバ上のデフォルト・データベースに対して試行されます。

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:machine-name:port-
number?ServiceName=DBN
```

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。= 記号の前後にはスペースを入れないでください。*DBN* パラメータはデータベース名です。

RemotePWD パラメータの使用

より一般的な方法は、**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することです。**setRemotePassword()** メソッドを使用して、[プロパティ] フィールドに **RemotePWD** を設定します。

フィールドの使い方を示すサンプル・コードは、次のとおりです。

```
sybDrvr = (SybDriver)Class.forName(
     "com.sybase.jdbc2.jdbc.SybDriver" ).newInstance();
 props = new Properties();
 props.put( "User", "DBA" );
 props.put( "Password", "SQL" );
 sybDrvr.setRemotePassword(
     null, "dbf=asademo.db", props );
 Connection con = DriverManager.getConnection(
     "jdbc:sybase:Tds:localhost", props );
```

データベース・ファイル・パラメータ **DBF** を使用することによって、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** の DBF または DBN を指定すると、ユーティリティ・データベースが自動的に起動されます。

ユーティリティ・データベースの詳細については、『ASA データベース管理ガイド』>「ユーティリティ・データベースの使用」を参照してください。

jConnect のマニュアルについては、http://sybooks.sybase.com/jc.html 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 によって、Adaptive Server Enterprise の動作と互換性を保つためのデータベース・オプションが設定されます。
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、sp_mda プロシージャを変更してください。

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。

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」131 ページを参照してください。

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 *dbjodbc9.dll* です。UNIX と Linux の場合、ネイティブ・コンポーネントは *dbjodbc9.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。

接続の確立

次のコードは、iAnywhere JDBC ドライバを使用して接続を確立する方法を示しています。

```
String     driver, url;
 Connection  conn;
 driver="ianywhere.ml.jdbcodbc.IDriver";
 url = "jdbc:odbc:dsn=ASA 9.0 Sample";
 Class.forName( driver );
 conn = DriverManager.getConnection( url );
```

このコードでは次の点に注意してください。

- 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。

- URL には、**jdbc:odbc:** 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、『ASA データベース管理ガイド』> 「接続パラメータ」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ドライバ・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。

```
url = "jdbc:odbc:";
 url += "driver=Adaptive Server Anywhere 9.0;...";
```

文字セット

UNIX では、iAnywhere JDBC ドライバは、ODBC のユニコード・バインドまたは呼び出しは使用 *しません*。また、文字変換も実行しません。iAnywhere JDBC ドライバを介して ASCII 以外のデータを送信すると、データが破損する可能性があります。

Windows では、iAnywhere JDBC ドライバは、ODBC のユニコード・バインドまたは呼び出しを使用 *します*。また、文字セットの変換も実行します。

JDBC 接続の確立

この項では、Java アプリケーションから JDBC データベース接続を確立するクラスについて説明します。この項の例では、jConnect (クライアント側) またはデータベース内の Java (サーバ側) を使用しています。iAnywhere JDBC ドライバを使用した接続確立の詳細については、「iAnywhere JDBC ドライバの使用」144 ページを参照してください。

jConnect を使用した JDBC クライアント・アプリケーションからの接続

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、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。内部 JDBC ドライバ・クラスと jConnect は、データベース・メタデータのサポート用にストアド・プロシージャを共有します。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」138 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」151 ページを参照してください。

外部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。このソース・コードは、SQL Anywhere ディレクトリの *Samples¥ASA¥Java* ディレクトリにある *JDBCExamples.java* ファイルの **main** メソッドと **ASAConnect** メソッドに含まれています。

```
import java.sql.*;              // JDBC
 import com.sybase.jdbc2.jdbc.*;      // Sybase jConnect
 import java.util.Properties;   // Properties
 import sybase.sql.*;           // Sybase utilities
 import asademo.*;              // Example classes
 public class JDBCExamples{
   private static Connection conn;

public static void main( String args[] ){
     // Establish a connection
     conn = null;
     String machineName =
        ( args.length == 1 ? args[0] : "localhost" );
     ASAConnect( "DBA", "SQL", machineName );
     if( conn!=null ) {
         System.out.println( "Connection successful" );
     }else{
         System.out.println( "Connection failed" );
     }

     try{
        getObjectColumn();
        getObjectColumnCastClass();
        insertObject();
     }
     catch( Exception e ){
       System.out.println( "Error: " + e.getMessage() );
        e.printStackTrace();
     }
   }

private static void ASAConnect( String userID,
                      String password,
                      String machineName ) {
     // Connect to an Adaptive Server Anywhere
     String coninfo = new String( machineName );
```

```
        Properties props = new Properties();
        props.put( "user", userID );
        props.put( "password", password );
        props.put("DYNAMIC_PREPARE", "true");

    // Load jConnect
       try {
         Class.forName(
         "com.sybase.jdbc2.jdbc.SybDriver" ).newInstance();
          String dbURL = "jdbc:sybase:Tds:" + machineName  +
                          ":2638/?JCONNECT_VERSION=5";
       System.out.println( dbURL );
        conn = DriverManager.getConnection( dbURL , props );
         }
         catch ( Exception e ) {
          System.out.println( "Error: " + e.getMessage() );
           e.printStackTrace();
         }
       }
     }
```

外部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はいくつかのライブラリを必要とし、そのライブラリは *JDBCExamples.java* の 1 行目からインポートされます。

- **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となります。これは Java サブディレクトリの *classes.zip* ファイルにあります。
- **com.sybase.jdbc2.jdbc** からインポートされる Sybase jConnect JDBC ドライバは、jConnect を使用して接続するすべてのアプリケーションで必要です。
- このアプリケーションは「**プロパティ・リスト**」を使用します。プロパティ・リストを処理するには **java.util.Properties** クラスが必要です。これは *Java* サブディレクトリの *classes.zip* ファイルにあります。

- **asademo** パッケージには一部の例で使用されるクラスが入っています。これは *Samples¥ASA¥Java¥asademo.jar* ファイルにあります。

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、**JDBCExamples.main** がアプリケーションで唯一のメソッドです。

JDBCExamples.main メソッドは次の作業を実行します。

1. マシン名が入力されている場合はそれを使用してコマンド・ライン引数を処理します。デフォルトでは、マシン名は *localhost* です。これはパーソナル・データベース・サーバに適した名前です。
2. **ASAConnect** メソッドを呼び出して接続を確立します。
3. データをコマンド・ラインにスクロールするメソッドをいくつか実行します。

ASAConnect メソッド

JDBCExamples.ASAConnect メソッドは次の作業を実行します。

1. Sybase jConnect を使用して、稼動中のデフォルト・データベースに接続します。
   - **Class.forName** が jConnect 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。
   - **StringBuffer** 文が、リテラル文字列とコマンド・ラインで指定されたマシン名から接続文字列を構築します。
   - **DriverManager.getConnection** が接続文字列を使用して接続を確立します。
2. 呼び出し元メソッドに制御を返します。

外部接続サンプルの実行

この項では、外部接続サンプルを実行する方法について説明します。

❖ **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・プロンプトを開きます。

2 SQL Anywhere ディレクトリに移動します。

3 *Samples¥ASA¥Java* サブディレクトリに移動します。

4 TCP/IP が動作しているデータベース・サーバにデータベースがロードされていることを確認します。そのようなサーバをローカル・マシンで起動するには、(*Samples¥ASA¥Java* サブディレクトリから) 次のコマンドを使用します。

```
start dbeng9 ..¥..¥..¥asademo
```

5 コマンド・プロンプトで次のように入力して、サンプルを起動します。

```
java JDBCExamples
```

別のマシン上で動作しているサーバに対してこれを行う場合、そのマシンの正しい名前を入力してください。デフォルトは、現在のマシン名のエイリアスである **localhost** になっています。

6 コマンド・プロンプトにユーザと製品のリストが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。CLASSPATH が正しいことを確認します。CLASSPATH が不正だと、クラスの配置に失敗します。

jConnect の使用方法の詳細については、「jConnect JDBC ドライバの使用」138 ページと jConnect のオンライン・マニュアルを参照してください。

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。サーバ側クラスを実行するのはすでに接続されているユーザなので、クラスは現在の接続を使用するだけです。

サーバ側接続サンプルのコード

次に示すのはソース・コードのサンプルです。ソース・コードは、SQL Anywhere ディレクトリにある *Samples¥ASA¥Java¥JDBCExamples.java* の **InternalConnect** メソッドに含まれています。

```
public static void InternalConnect() {
   try {
       conn =
DriverManager.getConnection("jdbc:default:connection");
       System.out.println("Hello World");
     }
     catch ( Exception e ) {
       System.out.println("Error: " + e.getMessage());
       e.printStackTrace();
     }
   }
 }
```

サーバ側接続サンプルの動作

この簡単な例では、**InternalConnect()** がアプリケーションで使用される唯一のメソッドです。

このアプリケーションで必要となるのは、*JDBCExamples.java* クラスの 1 行目にインポートされたライブラリ (JDBC) のうちの 1 つだけです。その他のライブラリは外部接続で必要になります。**java.sql** というパッケージには、JDBC クラスが入っています。

InternalConnect() メソッドは次の作業を実行します。

1. 現在の接続を使用して、稼動中のデフォルト・データベースに接続します。

   - **DriverManager.getConnection** が **jdbc:default:connection** の接続文字列を使用して接続を確立します。

2. 現在の標準出力であるサーバ・ウィンドウに、**Hello World** と出力します。**System.out.println** が出力を実行します。

3. 接続中にエラーが発生すると、サーバ・ウィンドウに、エラー・メッセージとエラーが発生した箇所が表示されます。

   try 命令と **catch** 命令がエラー処理を指示します。

4. クラスを終了します。

サーバ側接続サンプルの実行

この項では、サーバ側接続サンプルの実行方法について説明します。

❖ 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 まだ *JDBCExamples.java* ファイルをコンパイルしていない場合は、コンパイルします。JDK を使用している場合は、*Samples¥ASA¥Java* ディレクトリ内でコマンド・プロンプトから次のコマンドを実行します。

```
javac JDBCExamples.java
```

2 サンプル・データベースを使用してデータベース・サーバを起動します。そのようなサーバをローカル・マシンで起動するには、(*Samples¥ASA¥Java* サブディレクトリから) 次のコマンドを使用します。

```
start dbeng9 ..¥..¥..¥asademo
```

jConnect を使用していないので、このケースでは TCP/IP ネットワーク・プロトコルは不要です。

3　クラスをサンプル・データベースにインストールします。サンプル・データベースに一度接続すれば、Interactive SQL から次のコマンドを使用して接続できます。

```
INSTALL JAVA NEW
 FROM FILE
'path¥Samples¥ASA¥Java¥JDBCExamples.class'
```

path はインストール・ディレクトリです。

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] ー [新規] ー [Java クラス] を選択します。ウィザードの指示に従います。

4　ストアド・プロシージャを呼び出すのと同じ方法で、このクラスの **InternalConnect** メソッドを呼び出します。

```
CALL JDBCExamples>>InternalConnect()
```

セッション中に Java クラスが最初に呼び出されると、内部 Java 仮想マシンがロードされます。これには数秒を要します。

5　メッセージ **Hello World** が、サーバ画面に出力されることを確認します。

JDBC 接続についての注意

- **Autocommit 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、サーバ側 JDBC では COMMIT が実行されています。この動作を制御するには、次の文を使用します。

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

  ここで、**conn** は現在の接続オブジェクトです。

- **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 AutoCommit を OFF に設定すると、同じ Java コード内の後続の **getConnection** 呼び出しは、AutoCommit が OFF 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設定し直して、後続の接続が標準の JDBC 値を取得できるようにしたい場合があります。これは、次のようなコードによって実現します。

```
Connection conn = DriverManager.getConnection("");
 boolean oldAutoCommit = conn.getAutoCommit();
 try {
      // do code here
 }
 finally {
     conn.setAutoCommit( oldAutoCommit );
 }
```

これは AutoCommit だけではなく、TransactionIsolation や isReadOnly などの他の接続プロパティにも適用されます。

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 の機能を確認したほうが便利な場合もあります。次の例では、ローを Department 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。

この項では、静的 SQL を使用しています。動的 SQL については以降の項で説明します。

サンプルの準備

この項では、これ以降使用するサンプルを準備する方法について説明します。

サンプル・コード

この項のコード・フラグメントは、完全なクラス *Samples¥ASA¥Java¥JDBCExamples.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。**

- *JDBCExamples.class* ファイルをサンプル・データベースにインストールしていない場合は、インストールします。Interactive SQL からサンプル・データベースに接続したら、[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¥Samples¥ASA¥Java¥JDBCExamples.class'
```

path はインストール・ディレクトリです。

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] ー [新規] ー [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・フラグメントでは、JDBC が INSERT 文を実行する方法が示されています。ここでは、**conn** という名前の Connection オブジェクトに保持される、内部接続を使用します。JDBC を使用して外部アプリケーションから値を挿入するコードには、異なる接続を使用する必要がありますが、コードを使用しない場合は変更されません。

```
public static void InsertFixed()  {
     // returns current connection
     conn = DriverManager.getConnection(
         "jdbc:default:connection");
     // Disable autocommit
     conn.setAutoCommit( false );

     Statement stmt = conn.createStatement();
```

```
        Integer IRows = new Integer( stmt.executeUpdate
           ("INSERT INTO Department (dept_id, dept_name )"
            + "VALUES (201, 'Eastern Sales')"
             ) );
        // Print the number of rows updated
        System.out.println(IRows.toString() + " row
inserted" );
     }
```

利用可能なソース・コード
このコード・フラグメントは **JDBCExamples** クラスの **InsertFixed** メソッドの一部であり、インストール・ディレクトリの *Samples¥ASA¥Java* サブディレクトリにあります。

注意

- **setAutoCommit** メソッドは AutoCommit の動作をオフにするため、明示的な COMMIT 命令が実行された場合にのみ、変更がコミットされます。
- **executeUpdate** メソッドは整数を返します。この整数は、操作の影響を受けるローの番号を表しています。この場合、INSERT が成功すると、値 1 が返されます。
- 整数のリターン・タイプが、**Integer** オブジェクトに変換されます。整数クラスは基本の **int** データ型のラッパーであり、**toString()** など、いくつかの便利なメソッドを持っています。
- 整数 **IRows** が、文字列に変換されて出力されます。出力先はサーバ・ウィンドウです。

❖ 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 から、ユーザ ID **DBA** でサンプル・データベースに接続します。
2. JDBCExamples クラスがインストールされていることを確認します。これは、他のクラスの Java のサンプルとともにインストールされます。

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、「Java のサンプルの設定」98 ページを参照してください。

3 次のようにメソッドを呼び出します。

```
CALL JDBCExamples>>InsertFixed()
```

4 ローが department テーブルに追加されたことを確認します。

```
SELECT *
 FROM department
```

ID 201 のローはコミットされません。ROLLBACK 文を実行すれば、このローを削除できます。

この例では、非常に単純な JDBC クラスの作成方法について説明しました。以降の例では、これをさらに詳しく解説します。

Java メソッドへの引数の引き渡し

InsertFixed メソッドをさらに展開して、引数が Java メソッドに渡される過程について説明します。

次のメソッドは、呼び出し内でメソッドに渡された引数を、挿入する値として使用しています。

```
public static void InsertArguments(
                  String id, String name)  {
 try {
       conn = DriverManager.getConnection(
                     "jdbc:default:connection" );

 String sqlStr =  "INSERT INTO Department "
     + " ( dept_id, dept_name )"
     + " VALUES (" +  id +  ", '"  + name + "')";

       // Execute the statement
       Statement stmt = conn.createStatement();
       Integer IRows = new Integer(
          stmt.executeUpdate( sqlStr.toString() ) );

       // Print the number of rows updated
```

```
        System.out.println(IRows.toString() + " row
inserted" );
      }
      catch ( Exception e ) {
        System.out.println("Error: " + e.getMessage());
        e.printStackTrace();
      }
    }
```

注意

- 2 つの引数は部門 ID (整数) と部門名 (文字列) です。ここでは、両方の引数が文字列としてメソッドに渡されます。これは、これらの引数が SQL 文の文字列の一部であるためです。
- INSERT は静的文なので、SQL そのもの以外のパラメータはありません。
- 引数の数やタイプを誤って指定すると、「プロシージャは見つかりません」というエラー・メッセージが表示されます。

❖ 引数を指定して Java メソッドを使用するには、次の手順に従います。

1. *JDBCExamples.class* ファイルをまだサンプル・データベースにインストールしていない場合は、インストールします。
2. Interactive SQL からサンプル・データベースに接続し、次のコマンドを入力します。

```
call JDBCExamples>>InsertArguments(
     '203', 'Northern Sales' )
```

3. ローが Department テーブルに追加されたことを確認します。

```
SELECT *
 FROM Department
```

4. 変更をロールバックし、データベースを未変更のままにします。

```
ROLLBACK
```

JDBC を使用したクエリ

Statement オブジェクトは、静的クエリや、結果セットを返さない文を実行します。クエリでは、**Statement** オブジェクトの **executeQuery** メソッドを使用します。これにより、**ResultSet** オブジェクトに結果セットが返されます。

次のコード・フラグメントは、JDBC 内でクエリが処理される方法を示しています。このコード・フラグメントは、ある製品の在庫の合計値を **inventory** という名前の変数に配置します。製品名が、**String** 変数 **prodname** に格納されます。この例は、**JDBCExamples** クラスの **Query** メソッドとして利用できます。

次の例では、内部接続または外部接続が取得され、Connection オブジェクト **conn** に格納されていることが前提です。

```
public static int Query () {
 int max_price = 0;
     try{
       conn = DriverManager.getConnection(
                    "jdbc:default:connection" );

       // Build the query
       String sqlStr =  "SELECT id, unit_price "
           + "FROM product" ;

       // Execute the statement
       Statement stmt = conn.createStatement();
       ResultSet result = stmt.executeQuery( sqlStr );

       while( result.next() ) {
           int price = result.getInt(2);
           System.out.println( "Price is "  + price );
           if( price > max_price ) {
              max_price = price ;
           }
       }
     }
     catch( Exception e ) {
       System.out.println("Error: " + e.getMessage());
       e.printStackTrace();
     }
       return max_price;
   }
```

サンプルの実行

JDBCExamples クラスをサンプル・データベースに一度インストールすれば、Interactive SQL で次の文を使用して、このメソッドを実行できます。

```
CALL JDBCExamples>>Query()
```

注意

- このクエリは、**prodname** という名前の製品の数量と単価を選択します。これらの結果は、**result** という名前の **ResultSet** オブジェクトに返されます。
- 結果セットのそれぞれのローにはループがあります。ループは **next** メソッドを使用します。
- ローごとに、**getInt** メソッドを使用して各カラムの値が整数変数に取り出されます。**ResultSet** には、**getString**、**getDate**、**getBinaryString** などの他のデータ型のメソッドもあります。

  getInt メソッドの引数は、カラムのインデックス番号です。この番号は 1 から始まります。
- Adaptive Server Anywhere は、双方向にスクロールするカーソルをサポートしています。ただし、JDBC は、結果セット内を前方にスクロールする **next** メソッドしか提供していません。
- このメソッドは **max_price** の値を呼び出しを行った環境に返し、Interactive SQL がこれを [結果] ウィンドウ枠の [結果] タブに表示します。

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」14 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 クラスの次のメソッドによって、準備文を実行します。

```
public static void JInsertPrepared(int id, String name)
 try {
       conn = DriverManager.getConnection(
                    "jdbc:default:connection");

       // Build the INSERT statement
       // ? is a placeholder character
       String sqlStr = "INSERT INTO Department "
                + "( dept_id, dept_name ) "
                + "VALUES ( ? , ? )" ;

       // Prepare the statement
       PreparedStatement stmt =
           conn.prepareStatement( sqlStr );

       stmt.setInt(1, id);
       stmt.setString(2, name );
       Integer IRows = new Integer(
                           stmt.executeUpdate() );

       // Print the number of rows updated
       System.out.println(
            IRows.toString() + " row inserted" );
     }
     catch ( Exception e ) {
       System.out.println("Error: " + e.getMessage());
       e.printStackTrace();
     }
   }
```

サンプルの実行

JDBCExamples クラスをサンプル・データベースに一度インストールすれば、次の文を入力することによってこのサンプルを実行できます。

```
call JDBCExamples>>InsertPrepared(
                               202, 'Eastern Sales' )
```

文字列の引数が一重引用符で囲まれていますが、これは SQL 特有のものです。このメソッドを Java アプリケーションから呼び出す場合は、二重引用符で文字列を区切ります。

JDBC に関する各種注意事項

- **アクセス・パーミッション**　データベース内のすべての Java クラスと同じように、ユーザは JDBC 文を含んでいるクラスにアクセスできます。プロシージャを実行するパーミッションを付与する、GRANT EXECUTE 文に相当するものはなく、クラス名をその所有者名で修飾する必要はありません。
- **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。この動きは、所有者のパーミッションによって実行されるストアド・プロシージャの動きとは異なります。

JDBC エスケープ構文の使用

JDBC エスケープ構文は、Interactive SQL 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* }}

大カッコ ({) は*必ず*二重にしてください。この二重カッコの使用は、Interactive SQL に固有です。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 で実行しないため、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、次のように入力すると、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、jConnect と iAnywhere JDBC 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。

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

数値関数	文字列関数	システム関数	日付／時刻関数
CEILING	LENGTH		HOUR
COS	REPEAT		MINUTE
COT	RIGHT		MONTH
DEGREES	SOUNDEX		MONTHNAME
EXP	SPACE		NOW
FLOOR	SUBSTRING		QUARTER
LOG	UCASE		SECOND
LOG10			TIMESTAMPADD
PI			TIMESTAMPDIFF
POWER			YEAR
RADIANS			
RAND			
ROUND			
SIGN			
SIN			
SQRT			
TAN			

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

数値関数	文字列関数	システム関数	日付／時刻関数
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
MOD	RTRIM		WEEK
PI	SOUNDEX		YEAR
POWER	SPACE		
RADIANS	SUBSTRING		
RAND	UCASE		
ROUND			
SIGN			
SIN			
SQRT			
TAN			
TRUNCATE			

エスケープ構文を使用している文は、Adaptive Server 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、Interactive SQL の [SQL 文] ウィンドウ枠で次のように入力します。

```
{{CALL sa_db_info( 1 ) }}
```

第 6 章

Embedded SQL のプログラミング

この章の内容

この章では、Adaptive Server Anywhere に対して Embedded SQL プログラミング・インタフェースを使用する方法について説明します。

概要

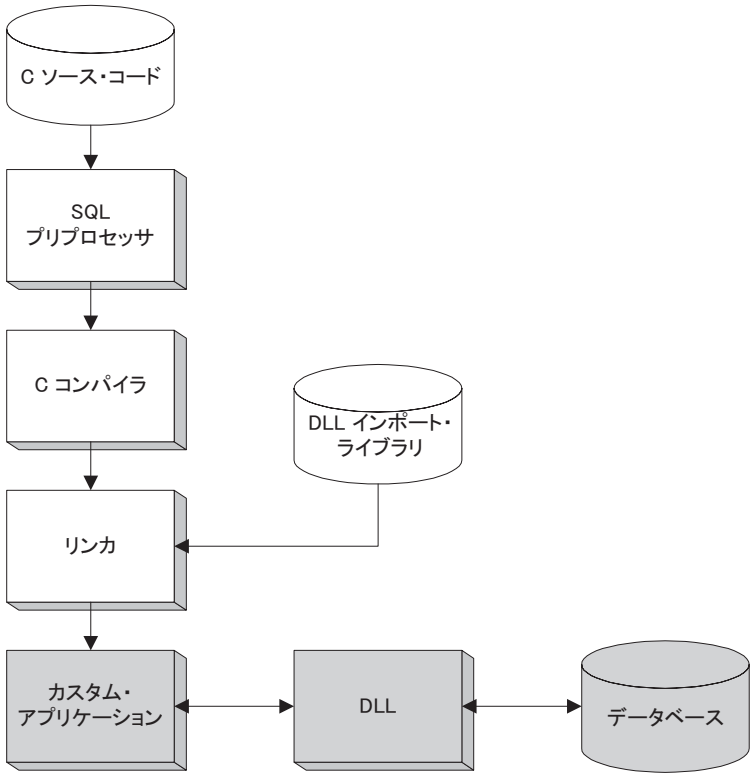
Embedded SQL は、C と C++ プログラミング言語用のデータベース・プログラミング・インタフェースです。Embedded SQL は、C と C++ のソース・コードが混合された (埋め込まれた) SQL 文で構成されます。この SQL 文は「**SQL プリプロセッサ**」によって C または C++ のソース・コードに翻訳され、その後ユーザによってコンパイルされます。

実行時に、Embedded SQL アプリケーションは Adaptive Server 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 *dblib9.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、*libdblib9.so*、*libdblib9.sl*、または *libdblib9.a* です。

Adaptive Server Anywhere には、2 種類の Embedded SQL が用意されています。静的な ESQL は、動的な ESQL に比べて使用方法は単純ですが、柔軟性は乏しくなります。この章では両者について説明します。

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 Adaptive Server 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。データベースが起動中のとき、この実行ファイルは Adaptive Server Anywhere の DLL を使用してデータベース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が起動し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、Adaptive Server 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、こちらのほうがおすすめです。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」176 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。

コマンド・ライン

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」248 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/C++	1.0 以上
Windows	Borland C++	4.5

オペレーティング・システム	コンパイラ	バージョン
Windows CE	Microsoft Visual C/C++	5.0
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」177 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。
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。

ファイル名	説明
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。対応コンパイラは Watcom C/C++、Microsoft Visual C++、IBM Visual Age、Borland C/C++。

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の *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* サブディレクトリに格納されています。

オペレーティング・システム	コンパイラ	インポート・ライブラリ
Windows	Watcom C/C++	*dblibtw.lib*
Windows	Microsoft Visual C++	*dblibtm.lib*
Windows CE	Microsoft Visual C++	*dblib9.lib*
NetWare	Watcom C/C++	*dblib9.lib*
Solaris (非スレッド・アプリケーション)	全コンパイラ	*libdblib9.so*、*libdbtasks9.so*
Solaris (スレッド・アプリケーション)	全コンパイラ	*libdblib9_r.so*、*libdbtasks9_r.so*

libdbtasks9 ライブラリは、*libdblib9* ライブラリに呼び出されます。コンパイラによっては、*libdbtasks9* 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 employee
         SET emp_lname =     'Plankton'
         WHERE emp_id = 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」208 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。

CONNECT 文は、最初に実行する ESQL 文にしてください。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 Adaptive Server Anywhere アプリケーションを開発する方法を説明します。Adaptive Server 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。プログラム開発には *esqldll.c* モジュールを使用することをおすすめします。こちらのほうが使いやすく、また、インタフェース DLL が見つからないというエラーの発生する可能性も少なくなるためです。

❖ 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** するか、**#include** 〈*sqlca.h*> 行を入れるかをしてください。

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_WINNT	すべての Windows オペレーティング・システム
_SQL_OS_UNIX	UNIX
_SQL_OS_NETWARE	NetWare

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、SQL Anywhere ディレクトリの *Samples¥ASA¥ESQLDynamicLoad* サブディレクトリにあります。ソース・コードは、*Samples¥ASA¥ESQLDynamicLoad¥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 dblib9
 OPTION CASEEXACT
 IMPORT @dblib9.imp
 LIBRARY dblib9.lib
```

dblib9.imp ファイルと *dblib9.lib* ファイルは、Adaptive Server 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

Adaptive Server 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。各サンプル・プログラムは、SQL Anywhere ディレクトリの *Samples¥ASA¥C* サブ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 *Samples¥ASA¥C¥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 *Samples¥ASA¥C¥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 Studio 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・システムでホストされている 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、Microsoft Visual C++ 用の *dcur.dsp* プロジェクト・ファイルを使用してください。

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。

- **WINNT** Windows 用にコンパイル
- **NETWARE** NetWare NLM 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 を使用
- **BC**　Borland C を使用

サンプル・プログラムの実行

実行ファイルは、ソース・コードとともに *Samples¥ASA¥C* 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　プログラムを起動します。

- Adaptive Server Anywhere パーソナル・サーバのサンプル・データベースを起動します。
- ファイル *Samples¥ASA¥C¥curwnt.exe* を実行します。

2　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　プログラムを起動します。

- ファイル *Samples¥ASA¥C¥dcurwnt.exe* を実行します。

2　データベースに接続します。

- 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=ASA 9.0 Sample
  ```

3 テーブルを選択します。

- 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、**Customer** または **Employee** 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

サンプル・プログラムの Windows バージョンは本物の Windows プログラムです。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 **employee** 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。次のサンプル (「動的カーソルのサンプル」183 ページ) 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。

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」179 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。まだ「静的カーソルのサンプル」182 ページを読んでいない場合は、目を通しておくことをおすすめします。このサンプルの理解に役立ちます。

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」179 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=c:¥asa¥asademo.db
```

ESQL を含んだ C プログラムは、SQL Anywhere ディレクトリの *Samples¥ASA¥C* サブ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 tablename
```

この *tablename* 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。

サービスのサンプル

サンプル・プログラムの *cur.sqc* と *dcur.sqc* は、サービスをサポートするバージョンの Windows 用にコンパイルすると、サービスとしても実行できます。

Windows サービスのサンプル用のファイルは、ソース・ファイル *ntsvc.c* とヘッダ・ファイル *ntsvc.h* の 2 つです。リンクされた実行プログラムは、通常の実行プログラムまたは Windows サービスとして実行できます。

❖ コンパイルしたサンプルを Windows サービスとして実行するには、次の手順に従います。

1. Sybase Central を起動します。
2. 左ウィンドウ枠で、Adaptive Server Anywhere 9 を選択します。
3. 右ウィンドウ枠で、[サービス] タブを選択します。
4. [ファイル] メニューから [新規] ー [サービス] を選択します。

   サービス作成ウィザードが表示されます。
5. 最初のページで、サービスの名前を入力します。
6. 2 番目のページで、サンプル・プログラムを選択します。
7. 3 番目のページで、SQL Anywhere ディレクトリの *Samples¥ASA¥C* サブディレクトリからサンプル・プログラム (*curwnt.exe* または *dcurwnt.exe*) を選択します。
8. ウィザードを完了して、サービスをインストールします。
9. メイン・ウィンドウの [開始] をクリックして、サービスを起動します。

サービスとして実行されるとき、このプログラムは可能ならば通常のユーザ・インタフェースを表示します。また、出力をアプリケーション・イベント・ログに書き込みます。ユーザ・インタフェースを起動できない場合、プログラムはデータの 1 ページ分をアプリケーション・イベント・ログに出力して停止します。

これらのサンプルは Watcom C/C++ 10.5 コンパイラと Microsoft Visual C++ コンパイラでテスト済みです。

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_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

```
typedef struct DECIMAL {
   char array[1];
 } DECIMAL;
```

- **DT_STRING** 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** ブランクが埋め込まれた固定長文字列
- **DT_VARCHAR** 2 バイトの長さフィールドを持つ可変長文字列。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 (埋め込みは行われません)。

```
typedef struct VARCHAR {
    unsigned short int len;
    char array[1];
 } VARCHAR;
```

- **DT_LONGVARCHAR** 長い可変長文字データ。マクロによって、構造体が次のように定義されます。

```
#define DECL_LONGVARCHAR( size )          ¥
   struct { a_sql_uint32    array_len;     ¥
            a_sql_uint32    stored_len;    ¥
            a_sql_uint32    untrunc_len;   ¥
            char            array[size+1];¥
          }
```

32 K を超えるデータには、DECL_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 で終了しません。

詳細については、「長い値の送信と取り出し」235 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。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。

```
#define DECL_LONGBINARY( size )          ¥
   struct { a_sql_uint32    array_len;    ¥
            a_sql_uint32    stored_len;   ¥
            a_sql_uint32    untrunc_len;  ¥
            char            array[size];  ¥
          }
```

32 K を超えるデータには、DECL_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」235 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; /* e.g. 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。

詳細については、『ASA データベース管理ガイド』> 「データベース・オプション」の DATE_FORMAT、TIME_FORMAT、TIMESTAMP_FORMAT、DATE_ORDER の各データベース・オプションを参照してください。

- **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 型、BINARY 型、DECIMAL 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を何度も割り当てるのには有用です。

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、『ASA SQL リファレンス・マニュアル』> 「GET DATA 文 [ESQL]」と 『ASA SQL リファレンス・マニュアル』> 「SET 文」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。

動的 SQL については、「静的 SQL と動的 SQL」219 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。IBM SAA と ANSI ESQL 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。

標準の SQL プリプロセッサは DECLARE SECTION 内以外では、C 言語コードはスキャンしません。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 Employee
 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」182 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、VARCHAR、FIXCHAR、BINARY、PACKED DECIMAL、LONG VARCHAR、LONG BINARY、または SQLDATETIME 型の構造体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_VARCHAR( 10 ) v_varchar;
 DECL_FIXCHAR( 10 ) v_fixchar;
 DECL_LONGVARCHAR( 32678 ) v_longvarchar;
 DECL_BINARY( 4000 ) v_binary;
 DECL_LONGBINARY( 128000 ) v_longbinary;
 DECL_DECIMAL( 10, 2 ) v_packed_decimal;
 DECL_DATETIME v_datetime;
 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。

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 のデータ型	Embedded SQL のインタフェースのデータ型	説明

`short i;`
`short int i;`
`unsigned short int i;`
	DT_SMALLINT	16 ビット符号付き整数

`long l;`
`long int l;`
`unsigned long int l;`
	DT_INT	32 ビット符号付き整数

`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
`DECL_DECIMAL(p,s)`	DT_DECIMAL(p,s)	パック 10 進数

`char a; /*n=1*/`
`DECL_FIXCHAR(n) a;`
`DECL_FIXCHAR a[n];`
	DT_FIXCHAR(n)	ブランクが埋め込まれた固定長文字列

`char a[n]; /*n>=1*/`	DT_STRING(n)	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。
`char *a;`	DT_STRING(32767)	NULL で終了する文字列
`DECL_VARCHAR(n) a;`	DT_VARCHAR(n)	2 バイトの長さフィールドを持つ可変長文字列。ブランクは埋め込まれない。

C のデータ型	Embedded SQL のインタフェースのデータ型	説明
`DECL_BINARY(n) a;`	DT_BINARY(n)	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
`DECL_DATETIME a;`	DT_TIMESTAMP_STRUCT	SQLDATETIME 構造体
`DECL_LONGVARCHAR( n ) a;`	DT_LONGVARCHAR	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。ブランクは埋め込まれず、NULL で終了しない。
`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。ブランクは埋め込まれない。

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・インタフェースからは 32 767 バイトの長さとみなされます。**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ときは、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受け取れるよう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。16 ビット・コンパイラの場合、単純に 32 767 バイトを確保するとプログラムのスタック・オーバフローを引き起こすこともあります。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ほうが安全です。こうすれば、PREPARE 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 では、ホスト変数はユーザ ID、パスワード、接続名、接続文字列、データベース環境名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。

例

- 次に示すのは、有効な ESQL です。

```
INCLUDE SQLCA;
 long SQLCODE;
 sub1() {
     char SQLSTATE[6];
     exec SQL CREATE TABLE ...
 }
```

- 次に示す ESQL は、有効ではありません。

```
INCLUDE SQLCA;
 sub1() {
     char SQLSTATE[6];
     exec SQL CREATE TABLE...
 }
 sub2() {
     exec SQL DROP TABLE...
     // No SQLSTATE in scope of this statement
 }
```

- SQLSTATE と SQLCODE はすべて大文字で書いてください。また、ISO/ANSI 規格ではこれらの定義は正確に次のようにする必要があります。

```
long SQLCODE;
 char SQLSTATE[6];
```

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 Employee
     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。

Adaptive Server 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 Employee
     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)。エラーについては次の項で説明します。

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 32 767 を超える場合は、インジケータ変数は 32 767 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

詳細については、『ASA SQL リファレンス・マニュアル』> 「GET DATA 文 [ESQL]」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がエラーを起こすと、**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 (次の項目を参照)。**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、『ASA エラー・メッセージ』> 「ASA 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、『ASA エラー・メッセージ』> 「ASA 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。これは ANSI SQL 規格 (SQL-92) であり、以前の規格の SQLCODE 値とは別に新しく定義された SQL 文の戻り値です。SQLSTATE 値は NULL で終了する長さ 5 の文字列で、前半 2 バイトがクラス、後半 3 バイトがサブクラスを表します。各バイトは 0 ～ 9 の数字、または、A ～ Z の英大文字です。

  0 ～ 4 または A ～ H の文字で始まるクラス、サブクラスはすべて SQL 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、『ASA エラー・メッセージ』> 「ASA 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。

  データベースはコマンド実行ごとにこの値を 0 に設定しなおすことはしません。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_COUNT 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・フェッチ中は、SQLE_NOTFOUND は*設定されません*。

    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」213 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」246 ページを参照してください。

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 プリプロセッサのオプションを使用して、再入可能コード (`-r`) を生成してください。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。

3. Embedded SQL 文の SET SQLCA (『ASA SQL リファレンス・マニュアル』> 「SET SQLCA 文 [ESQL]」) 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、『ASA SQL リファレンス・マニュアル』>「SET SQLCA 文 [ESQL]」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** 複数のスレッドが同じ SQLCA を使用すると、コンテキストの切り替えによって複数のスレッドがその SQLCA を同時に使用しようとすることがあります。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。『ASA SQL リファレンス・マニュアル』> 「SET CONNECTION 文 [Interactive SQL] [ESQL]」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、『ASA SQL ユーザーズ・ガイド』>「トランザクションと独立性レベル」の章を参照してください。

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。

  詳細については、「ローを返さないか、1 つだけ返す SELECT 文」208 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。

  詳細については、「ESQL でのカーソルの使用」209 ページを参照してください。

データ型 LONG VARCHAR と LONG BINARY の処理は、他のデータ型とは異なります。詳細については、「LONG データの取り出し」236 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 (SQL Communication Area)」201 ページ で説明しているように、SQLCA 構造体で返されます。

例

たとえば、次のコードは従業員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            emp_id;
     char            name[41];
     char            sex;
     char            birthdate[15];
     short int    ind_birthdate;
 EXEC SQL END DECLARE SECTION;
 . . .
 int find_employee( long employee )
 {
     emp_id = employee;
     EXEC SQL    SELECT emp_fname ||
                 ' ' || emp_lname, sex, birth_date
                 INTO :name, :sex,
                         :birthdate:ind_birthdate
                 FROM "DBA".employee
                 WHERE emp_id = :emp_id;
     if( SQLCODE == SQLE_NOTFOUND ) {
         return( 0 ); /* employee 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」23 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 (SQL Communication Area)」201 ページ で説明している SQLCA 構造体で返されます。

5 CLOSE 文を使ってカーソルを閉じます。

デフォルトによって、カーソルはトランザクション終了時 (COMMIT または ROLLBACK 時) に自動的に閉じられます。WITH HOLD 句を指定して開いたカーソルは、明示的に閉じるまで以降のトランザクション中も開いたままになります。

次は、簡単なカーソル使用の例です。

```
void print_employees( void )
 {
     EXEC SQL BEGIN DECLARE SECTION;
     char name[50];
     char sex;
     char birthdate[15];
     short int ind_birthdate;
     EXEC SQL END DECLARE SECTION;
     EXEC SQL DECLARE C1 CURSOR FOR
         SELECT    emp_fname || ' ' || emp_lname,
                     sex, birth_date
         FROM "DBA".employee;
     EXEC SQL OPEN C1;
     for( ;; ) {
         EXEC SQL FETCH C1 INTO :name, :sex,
         :birthdate:ind_birthdate;
         if( SQLCODE == SQLE_NOTFOUND ) {
             break;
```

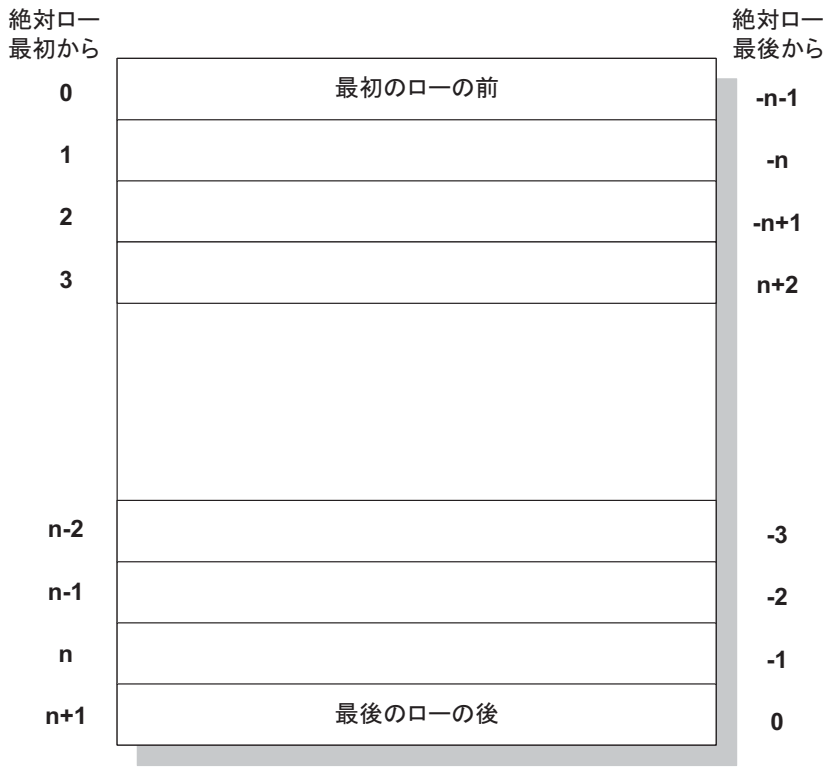
```
            } else if( SQLCODE < 0 ) {
                break;
            }
            if( ind_birthdate < 0 ) {
                strcpy( birthdate, "UNKNOWN" );
            }
            printf( "Name: %s Sex: %c Birthdate:
                        %s.n",name, sex, birthdate );
        }
    EXEC 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」182 ページと「動的カーソルのサンプル」183 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。FETCH コマンドについては、『ASA SQL リファレンス・マニュアル』> 「FETCH 文 [ESQL] [SP]」を参照してください。カーソルはクエリ結果の先頭または末尾を基点にした絶対位置に位置付けできます。カーソルの現在位置を基準にした相対位置にも移動できます。

カーソルの現在位置のローを更新または削除するために、UPDATE (*位置付け*) 文と DELETE (*位置付け*) 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。

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。

Adaptive Server 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、『ASA SQL ユーザーズ・ガイド』>「クエリ処理中のワーク・テーブルの使用」を参照してください。

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 (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。

Adaptive Server Anywhere は、ワイド・プットとワイド挿入もサポートします。これらの詳細については、『ASA SQL リファレンス・マニュアル』>「PUT 文 [ESQL]」と『ASA SQL リファレンス・マニュアル』>「EXECUTE 文 [ESQL]」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、SQL Anywhere ディレクトリの *samples¥ASA¥esqlwide-fetch¥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 will be 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 emp_fname, emp_lname from employee";
    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。

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

これまでは、静的 SQL の説明をしてきました。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 を置き、コマンドの後ろにセミコロン (;) を置いて、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。

静的文では「ホスト変数の使用」192 ページで説明したホスト変数を参照できます。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)」224 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 address[30];
     char city[20];
     short int cityind;
     long empnum;
 EXEC SQL END DECLARE SECTION;
 . . .
     sprintf( comm, "update %s set address = :?,
                              city = :?"
                  " where employee_number = :?",
                  tablename );
 EXEC SQL PREPARE S1 FROM :comm;
 EXEC SQL EXECUTE S1 USING :address, :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プログラマが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 address = :address,
              city = :city"
             " where employee_number = :empnum",
             tablename );
 EXEC SQL PREPARE S1 FROM :comm;
 /* Assume that there are no more than 10 host
variables. See next example if you can't 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、長さ
- **型**が数値型の場合、精度と位取り
- メモリ・アドレス
- インジケータ変数

SQLDA 構造体については、「SQLDA (SQL descriptor area)」224 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」197 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」183 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」252 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」252 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 */
 };

#define SCALE(sqllen)            ((sqllen)/256)
 #define PRECISION(sqllen)        ((sqllen)&0xff)
 #define SET_PRECISION_SCALE(sqllen,precision,scale)
¥
                             sqllen = (scale)*256 +
(precision)
 #define DECIMALSTORAGE(sqllen)  (PRECISION(sqllen)/2 +
1)

 typedef struct sqlda    SQLDA;
 typedef struct sqlvar   SQLVAR, SQLDA_VARIABLE;
 typedef struct sqlname  SQLNAME, SQLDA_NAME;

#ifndef SQLDASIZE
 #define SQLDASIZE(n)  ( sizeof( struct sqlda ) +  ¥
                       (n-1) * sizeof( struct sqlvar) )
 #endif
 #if defined( _SQL_PACK_STRUCTURES )
 #include "poppk.h"
 #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** 配列内の変数記述子の数。
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」187 ページを参照)。

  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。

DECIMAL 型では、このフィールドは 1 バイトのフィールド 2 つに分割されます。高位バイトは精度で、低位バイトは位取りです。精度とは総桁数のことです。位取りとは小数点以下の桁数のことです。

データ型 LONG VARCHAR と 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_LONGBINARY と DT_LONGVARCHAR 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」229 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 4 バイト・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」187 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。

詳細については、「インジケータ変数」197 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」229 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」232 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」231 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」187 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。

ESQL データ型については、「Embedded SQL のデータ型」187 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 VARCHAR	DT_LONGVARCHAR	32767
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

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
TINYINT	DT_TINYINT	1
UNSIGNED BIGINT	DT_UNSBIGINT	8
UNSIGNED INT	DT_UNSINT	4
UNSIGNED SMALLINT	DT_UNSSMALLINT	2
VARCHAR(n)	DT_VARCHAR	n

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_DECIMAL(p,s)	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
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

Embedded SQL のデータ型	長さを設定するプログラム動作
DT_LONGBINARY	長さフィールドが無視される。「LONG データの送信」239 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」239 ページを参照。
DT_SMALLINT	動作不要
DT_STRING	末尾の \0 によって長さが決まる
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
DT_BINARY(n)	BINARY 構造体の最大長 (n+2)	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_DECIMAL(p,s)	高位バイトを p に、低位バイトを s に設定	動作不要
DT_DOUBLE	動作不要	動作不要
DT_FIXCHAR(n)	バッファの長さ	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」236 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」236 ページを参照。
DT_LONGVARCHAR	長さフィールドが無視される。「LONG データの取り出し」236 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」236 ページを参照。
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

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_UNSBIGINT	動作不要	動作不要
DT_UNSINT	動作不要	動作不要
DT_UNSSMALLINT	動作不要	動作不要
DT_VARCHAR(n)	VARCHAR 構造体の最大長 (n+2)	VARCHAR 構造体の **len** フィールドに実際の長さを設定

長い値の送信と取り出し

Embedded SQL アプリケーションで LONG VARCHAR 値と LONG BINARY 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 フィールドを使用できますが、32 KB のデータに制限されています。これは、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 値と LONG BINARY 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」236 ページと 「LONG データの送信」239 ページを参照してください。

静的 SQL の使用法

データ型 LONG BINARY と LONG VARCHAR 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、別々の構造体が使用されます。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_LONGBINARY( size )           ¥
   struct { a_sql_uint32    array_len;     ¥
            a_sql_uint32    stored_len;    ¥
            a_sql_uint32    untrunc_len;   ¥
            char            array[size];   ¥
          }
```

動的 SQL の使用法

動的 SQL の場合は、**sqltype** フィールドを必要に応じて DT_LONGVARCHAR または DT_LONGBINARY に設定します。対応する LONGBINARY と LONGVARCHAR の構造体は、次のとおりです。

```
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, LONGBINARY;
```

この機能をアプリケーションに実装する方法の詳細については、「LONG データの取り出し」236 ページと 「LONG データの送信」239 ページを参照してください。

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」235 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 値または LONG BINARY 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 型または DECL_LONGBINARY 型のホスト変数を宣言します。
2. FETCH、GET DATA、または EXECUTE INTO を使用してデータを取り出します。Adaptive Server Anywhere によって次の情報が設定されます。
   - **インジケータ変数** インジケータ変数は、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。

詳細については、「インジケータ変数」197 ページを参照してください。

- **stored_len**　この DECL_LONGVARCHAR または DECL_LONGBINARY フィールドには、配列に取り出されたバイト数が入ります。**array_len** より大きくなることはありません。
- **untrunc_len**　この DECL_LONGVARCHAR または DECL_LONGBINARY フィールドには、データベース・サーバが保持するバイト数が入ります。**stored_len** 以上の値です。値がトランケートされない場合にも設定されます。

❖ LONGVARCHAR または 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 または DT_LONGBINARY に設定します。
2. **sqldata** フィールドを、LONGVARCHAR または LONGBINARY 構造体を指すように設定します。

   `LONGVARCHARSIZE( 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. LONGVARCHAR または LONGBINARY 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。Adaptive Server 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** この LONGVARCHAR または LONGBINARY フィールドには、配列に取り出されたバイト数が入ります。**array_len** より大きくなることはありません。

- **untrunc_len** この LONGVARCHAR または LONGBINARY フィールドには、データベース・サーバが保持するバイト数が入ります。stored_len 以上の値です。値がトランケートされない場合にも設定されます。

次のコードの抜粋は、動的 ESQL 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」235 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 値または LONG BINARY 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 型または DECL_LONGBINARY 型のホスト変数を宣言します。
2. NULL を送信し、インジケータ変数を使用している場合は、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」197 ページを参照してください。
3. DECL_LONGVARCHAR または DECL_LONGBINARY 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。

次のコードの抜粋は、静的 ESQL 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;
 }
```

❖ LONGVARCHAR または LONGBINARY 構造体を使用して値を送信するには、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 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 **sqldata** フィールドを、LONGVARCHAR または LONGBINARY 構造体を指すように設定します。

LONGVARCHARSIZE(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 LONGVARCHAR または LONGBINARY 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 LONGVARCHAR または LONGBINARY 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。

ストアド・プロシージャの使用

この項では、Embedded SQL における SQL プロシージャの使用方法を説明します。

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、『ASA SQL リファレンス・マニュアル』>「EXECUTE 文 [ESQL]」と『ASA SQL リファレンス・マニュアル』>「PREPARE 文 [ESQL]」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 emp_fname || emp_lname FROM employee
         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 emp_fname || emp_lname
     FROM employee;

     SELECT fname || lname
     FROM customer;
 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」222 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、『ASA SQL リファレンス・マニュアル』> 「DESCRIBE 文 [ESQL]」 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**」を用意すると、データベース要求の処理中もアプリケーションをアクティブにできます。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。

詳細については、次の項目を参照してください。

- 「db_register_a_callback 関数」263 ページ
- 「db_cancel_request 関数」256 ページ
- 「db_is_working 関数」260 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、Adaptive Server 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。詳細については、『ASA SQL リファレンス・マニュアル』> 「BACKUP 文」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。この関数の詳細については、「DBBackup 関数」332 ページ を参照してください。

詳細については、「db_backup 関数」253 ページを参照してください。

SQL プリプロセッサ

SQL プリプロセッサは、コンパイラが実行される前に、Embedded SQL を含む 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
-c *"keyword=value;..."*	リファレンス・データベース接続パラメータを指定する (Ultra Light)
-d	データ・サイズを適切なサイズにする
-e *level*	SQL 構文に準拠しないものをエラーとして通知する
-f	生成した静的データに far キーワードを付ける
-g	Ultra Light の警告を表示しない
-h *line-width*	出力する行の長さの最大値を制限する
-k	SQLCODE のユーザ宣言をインクルードする
-m *version*	生成される同期スクリプトのバージョン名を指定する
-n	行番号
-o *operating-sys*	ターゲット・オペレーティング・システム
-p *project*	Ultra Light プロジェクト名
-q	クワイエット・モード (バナーを表示しない)
-r	再入可能コードを生成する
-s *string-len*	コンパイラに与える最大文字列の長さ
-w *level*	SQL 構文に準拠しないものを警告として通知する
-x	複数バイト SQL 文字列をエスケープ・シーケンスに変更する
-z *sequence*	照合順を指定する

参照

「概要」170 ページ

説明

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。SQLPP 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。

オプション

-c　Ultra Light アプリケーションの一部であるファイルの前処理を実行するときに必要です。ここで接続文字列を指定することにより、SQL プリプロセッサは読み込みと修正のためにリファレンス・データベースにアクセスできるようになります。

-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　指定した SQL/92 のセットに含まれない Embedded SQL をエラーとして通知します。

次に、設定できる *level* の値とその意味を示します。

- **e**　初級レベルの SQL/92 構文ではない構文を通知します。
- **i**　中級レベルの SQL/92 構文ではない構文を通知します。
- **f**　上級レベルの SQL/92 構文では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。
- **w**　サポートされているすべての構文を許容します。

-g　Ultra Light のコード生成に固有の警告を表示しません。

-h　*sqlpp* によって出力される行の最大長を *num* 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、*num* 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。

-m 生成される同期スクリプトのバージョン名を指定します。生成された同期スクリプトを Mobile Link 統合データベースに使用すると簡単に同期できます。

-n C ファイルに行番号情報を生成します。これは、生成された C コード内の適切な場所にある *#line* 指令で構成されます。使用しているコンパイラが *#line* 指令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* 指令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 ターゲット・オペレーティング・システムを指定します。このオプションが、プログラムを実行するオペレーティング・システムと一致するように注意してください。プログラム内に、特殊記号への参照が生成されます。この記号はインタフェース・ライブラリで定義されます。適切でないオペレーティング・システムを指定したり、適切でないライブラリを使用したりすると、リンカがエラーを検知します。サポートされているオペレーティング・システムは次のとおりです。

- **WINDOWS** Windows 95/98/Me、Windows CE
- **WINNT** Microsoft Windows NT/2000/XP
- **NETWARE** Novell NetWare
- **UNIX** UNIX

-p Embedded SQL ファイルが属する Ultra Light プロジェクトを識別します。Ultra Light アプリケーションの一部であるファイルを処理する場合にのみ適用します。

-q バナーを表示しません。

-r 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」204 ページを参照してください。

-s　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。

-w　指定した SQL/92 のセットに含まれない Embedded SQL を、警告として通知します。

次に、設定できる *level* の値とその意味を示します。

- **e**　初級レベルの SQL/92 構文ではない構文を通知します。
- **i**　中級レベルの SQL/92 構文ではない構文を通知します。
- **f**　上級レベルの SQL/92 構文では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。
- **w**　サポートされているすべての構文を許容します。

-x　マルチバイト文字列をエスケープ・シーケンスに変更して、コンパイラをパススルーできるようにします。

-z　照合順を指定します。推奨する照合順のリストを表示するには、コマンド・プロンプトで **dbinit −l** 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 SQLLOCALE 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。

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

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

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,
    SQLDA * sqlda );
```

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

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。詳細については、『ASA SQL リファレンス・マニュアル』> 「BACKUP 文」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 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 フィールドには各データベース・ページのサイズが設定されます。バックアップは一度に 1 ページずつ処理されます。

  file_num、*page_num*、*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 まで、トランザクション・ログ・ファイルの場合は DB_BACKUP_TRANS_LOG_FILE まで、データベース・ライ

ト・ファイルが存在する場合は DB_BACKUP_WRITE_FILE 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 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.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。

このあと、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。

dbbackup プログラムは、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。

```
db_backup( ... DB_BACKUP_START ... )
 allocate page buffer based on page size in SQLCODE
 sqlda = alloc_sqlda( 1 )
 sqlda->sqld = 1;
 sqlda->sqlvar[0].sqltype = DT_BINARY
 sqlda->sqlvar[0].sqldata = allocated buffer
 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
     for page_num = 0 to num_pages - 1
       db_backup( ... DB_BACKUP_READ_PAGE,
                  file_num, page_num, sqlda )
       write page buffer out to backup file
     next page_num
     close backup file
    db_backup( ... DB_BACKUP_CLOSE_FILE, file_num ... )
   end if
 next file_num
 backup up file DB_BACKUP_WRITE_FILE as above
 backup up file DB_BACKUP_TRANS_LOG_FILE as above
 free page buffer
 db_backup( ... DB_BACKUP_END ... )
```

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_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、古いほう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」253 ページの DB_BACKUP_READ_RENAME_LOG 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・データベース環境について情報が返されます。

db_fini 関数

プロトタイプ

unsigned short **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。

参照

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、『Ultra Light C/C++ ユーザーズ・ガイド』> 「db_fini 関数」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、現在接続しているサーバのアドレスを取得するために使用します。*dbping* ユーティリティでは、サーバ・アドレスを出力するために使用されます。

また、この関数を使用すると、データベース・プロパティの値を取得できます。データベース・プロパティは、『ASA データベース管理ガイド』>「データベース・プロパティ」で説明しているように、SELECT 文を実行してインタフェースに依存しない方法で取得することもできます。

引数は次のとおりです。

- **a_db_property**　値 DB_PROP_SERVER_ADDRESS を持つ **enum**。DB_PROP_SERVER_ADDRESS 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。共有メモリと NamedPipes の各プロトコルは、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。
- **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 文字を含め、文字列 **value_buffer** の最大長です。

参照

『ASA データベース管理ガイド』>「データベース・プロパティ」

db_init 関数

プロトタイプ

unsigned short **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。

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 ESQL 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、使用される各 SQLCA につき 1 回 **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」204 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。

参照

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、『Ultra Light C/C++ ユーザーズ・ガイド』> 「db_init 関数」を参照してください。

db_is_working 関数

プロトタイプ

unsigned **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 Adaptive Server Anywhere データベース・サーバをリストして、*dblocate* ユーティリティによって表示される情報にプログラムからアクセスできるようにします。

コールバック関数には、次のプロトタイプが必要です。

int (*)(SQLCA **sqlca*,
a_server_address **server_addr*,
void **callback_user_data*);

コールバック関数は、検出されたサーバごとに呼び出されます。コールバック関数が 0 を返すと、**db_locate_servers** はサーバ間の反復を中止します。

コールバック関数に渡される **sqlca** と **callback_user_data** は、**db_locate_servers** に渡されるものと同じです。2 番目のパラメータは、**a_server_address** 構造体へのポインタです。**a_server_address** は *sqlca.h* 内で次のように定義されています。

```
typedef struct a_server_address {
     a_SQL_uint32    port_type;
     a_SQL_uint32    port_num;
     char           *name;
     char           *address;
 } a_server_address;
```

- **port_type**　この時点では常に PORT_TYPE_TCP です (*sqlca.h* 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。

詳細については、『ASA データベース管理ガイド』> 「サーバ検索ユーティリティ」を参照してください。

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 Adaptive Server 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が、コールバック・ルーチン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、ビット・マスクを満足させているサーバごとに呼び出されます。コールバック関数が 0 を返すと、**db_locate_servers_ex** はサーバ間の反復を中止します。

コールバック関数に渡される **sqlca** と **callback_user_data** は、**db_locate_servers** に渡されるものと同じです。2 番目のパラメータは、**a_server_address** 構造体へのポインタです。**a_server_address** は *sqlca.h* 内で次のように定義されています。

```
typedef struct a_server_address {
     a_SQL_uint32    port_type;
     a_SQL_uint32    port_num;
     char           *name;
     char           *address;
 } a_server_address;
```

- **port_type** この時点では常に PORT_TYPE_TCP です (*sqlca.h* 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。

現在サポートされているビット・マスク・フラグは DB_LOOKUP_FLAG_NUMERIC のみです。このフラグは、sqlca.h で定義され、コールバック・ルーチンに渡されるアドレスがホスト名ではなく IP アドレスであることを保証します。

詳細については、『ASA データベース管理ガイド』> 「サーバ検索ユーティリティ」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データベースの応答を待ってブロックし、Windows のカーソルは砂時計の形に変わります。

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,
    DBCALLBACK_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** フィールドは識別子です。**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] ウィンドウ枠に表示しますが、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)」201 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** フォーム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"
```

使用可能な接続パラメータのリストについては、『ASA データベース管理ガイド』> 「接続パラメータ」を参照してください。

説明

データベースがまだ稼働していない場合、可能であれば、既存のサーバでデータベースを起動します。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、『ASA データベース管理ガイド』> 「サーバの検出」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 (非ゼロ値) 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、『ASA データベース管理ガイド』> 「データベース・サーバ」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)」201 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** フォーム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"
```

使用可能な接続パラメータのリストについては、『ASA データベース管理ガイド』> 「接続パラメータ」を参照してください。

説明

データベース・サーバが稼働しておらず、DBF パラメータが指定されていない場合は起動します。この関数によって実行されるステップについては、『ASA データベース管理ガイド』> 「サーバの検出」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 (非ゼロ値) 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 **asademo** 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=c:¥¥asa9¥¥asademo.db; Start=dbeng9" );
```

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 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=dbeng9 c:¥¥asa¥¥asademo.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 **asademo** 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 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 dbeng9 -n server_1 asademo.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=dbeng9 -n server_2
asademo.db;ForceStart=YES" )
```

FORCESTART が使用されず、ENG パラメータがなかった場合、2 番目のコマンドは server_1 に接続しようとします。**db_start_engine** 関数は、START パラメータの -n オプションからサーバ名を取り出しません。

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)」201 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** フォーム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"
```

使用可能な接続パラメータのリストについては、『ASA データベース管理ガイド』> 「接続パラメータ」を参照してください。

説明

EngineName で識別されるサーバ上の **DatabaseName** 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、『ASA データベース管理ガイド』> 「データベース・サーバ」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)」201 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** フォーム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"
```

使用可能な接続パラメータのリストについては、『ASA データベース管理ガイド』>「接続パラメータ」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** 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、この関数は失敗します。
- 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。

詳細については、『ASA データベース管理ガイド』>「-gk サーバ・オプション」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)」201 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** フォーム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"
```

使用可能な接続パラメータのリストについては、『ASA データベース管理ガイド』> 「接続パラメータ」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。この関数は、『ASA データベース管理ガイド』> 「接続のトラブルシューティング」で説明されているアルゴリズムを使用して接続を実行します。

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)」201 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** フォーム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"
```

使用可能な接続パラメータのリストについては、『ASA データベース管理ガイド』> 「接続パラメータ」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、戻り値はブール値の true になります。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。**AutoStop=yes** パラメータを使用して起動され、稼働中のデータベースが他に存在しない場合は、サーバも停止します。

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 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 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。0 以外 (true) の値を返すのは、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。0 以外の値を返すのは、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。

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。

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。

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。

『ASA データベース管理ガイド』> 「データベース・プロパティ」

『ASA データベース管理ガイド』> 「Ping ユーティリティ」

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ブール値を返します。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**　この場合は、文字列に引用符は必要ありません。
- **return = TRUE**　この場合は、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 long **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返します。

sqlda_string_length 関数

プロトタイプ unsigned long **sqlda_string_length(** SQLDA *_sqlda_**,** int _varno_ **);**

説明 C の文字列 (DT_STRING データ型) の長さを返します。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、『ASA SQL リファレンス・マニュアル』>「SQL 文」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。

これらの Embedded SQL コマンドの詳細については、『ASA SQL リファレンス・マニュアル』>「SQL 言語の要素」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

  『ASA SQL リファレンス・マニュアル』>「ALLOCATE DESCRIPTOR 文 [ESQL]」を参照。

- **CLOSE**　カーソルを閉じる

  『ASA SQL リファレンス・マニュアル』>「CLOSE 文 [ESQL] [SP]」を参照。

- **CONNECT**　データベースに接続する

  『ASA SQL リファレンス・マニュアル』>「CONNECT 文 [ESQL] [Interactive SQL]」を参照。

- **DEALLOCATE DESCRIPTOR** 記述子のメモリを再使用する

  『ASA SQL リファレンス・マニュアル』> 「DEALLOCATE DESCRIPTOR 文 [ESQL]」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

  『ASA SQL リファレンス・マニュアル』> 「宣言セクション [ESQL]」を参照。

- **DECLARE CURSOR** カーソルを宣言する

  『ASA SQL リファレンス・マニュアル』> 「DECLARE CURSOR 文 [ESQL] [SP]」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

  『ASA SQL リファレンス・マニュアル』> 「DELETE (位置付け) 文 [ESQL] [SP]」を参照。

- **DESCRIBE** 特定の SQL 文用のホスト変数を記述する

  『ASA SQL リファレンス・マニュアル』> 「DESCRIBE 文 [ESQL]」を参照。

- **DISCONNECT** データベース・サーバとの接続を切断する

  『ASA SQL リファレンス・マニュアル』> 「DISCONNECT 文 [ESQL] [Interactive SQL]」を参照。

- **DROP STATEMENT** 準備文が使用したリソースを解放する

  『ASA SQL リファレンス・マニュアル』> 「DROP STATEMENT 文 [ESQL]」を参照。

- **EXECUTE** 特定の SQL 文を実行する

  『ASA SQL リファレンス・マニュアル』> 「EXECUTE 文 [ESQL]」を参照。

- **EXPLAIN** 特定のカーソルの最適化方式を説明する

『ASA SQL リファレンス・マニュアル』>「EXPLAIN 文 [ESQL]」を参照。

- **FETCH**　カーソルからローをフェッチする

  『ASA SQL リファレンス・マニュアル』>「FETCH 文 [ESQL] [SP]」を参照。

- **GET DATA**　カーソルから長い値をフェッチする

  『ASA SQL リファレンス・マニュアル』>「GET DATA 文 [ESQL]」を参照。

- **GET DESCRIPTOR**　SQLDA 内の変数に関する情報を取り出す

  『ASA SQL リファレンス・マニュアル』>「GET DESCRIPTOR 文 [ESQL]」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

  『ASA SQL リファレンス・マニュアル』>「GET OPTION 文 [ESQL]」を参照。

- **INCLUDE**　SQL 前処理用のファイルをインクルードする

  『ASA SQL リファレンス・マニュアル』>「INCLUDE 文 [ESQL]」を参照。

- **OPEN**　カーソルを開く

  『ASA SQL リファレンス・マニュアル』>「OPEN 文 [ESQL] [SP]」を参照。

- **PREPARE**　特定の SQL 文を準備する

  『ASA SQL リファレンス・マニュアル』>「PREPARE 文 [ESQL]」を参照。

- **PUT**　カーソルにローを挿入する

  『ASA SQL リファレンス・マニュアル』>「PUT 文 [ESQL]」を参照。

- **SET CONNECTION** アクティブな接続を変更する

  『ASA SQL リファレンス・マニュアル』> 「SET CONNECTION 文 [Interactive SQL] [ESQL]」を参照。

- **SET DESCRIPTOR** SQLDA 内で変数を記述し、SQLDA にデータを置く

  『ASA SQL リファレンス・マニュアル』> 「SET DESCRIPTOR 文 [ESQL]」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

  『ASA SQL リファレンス・マニュアル』> 「SET SQLCA 文 [ESQL]」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

  『ASA SQL リファレンス・マニュアル』> 「UPDATE (位置付け) 文 [ESQL] [SP]」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

  『ASA SQL リファレンス・マニュアル』> 「WHENEVER 文 [ESQL]」を参照。

第 7 章

ODBC プログラミング

この章の内容

この章では、ODBC プログラミング・インタフェースを直接呼び出すアプリケーションの開発について説明します。

ODBC アプリケーション開発に関する主要なマニュアルは、Microsoft の ODBC SDK マニュアルで、Microsoft Data Access Components (MDAC) SDK の一部として入手できます。この章は、Adaptive Server Anywhere 特有の機能について説明している入門編であり、ODBC 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。

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。この章は、この種のツールのユーザを対象としていません。

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。

Adaptive Server 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。

- Adaptive Server 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

Adaptive Server 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」633 ページを参照してください。

ODBC 準拠

Adaptive Server 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。

Adaptive Server Anywhere がサポートする機能

Adaptive Server Anywhere は ODBC 3.5 仕様をサポートします。

- **コア準拠**　Adaptive Server 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**　Adaptive Server 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。

  Adaptive Server 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、Adaptive Server Anywhere によって直列化されます。
- **レベル 2 準拠**　Adaptive Server 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 Adaptive Server 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、Adaptive Server 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

ODBC ドライバ・マネージャは、Adaptive Server Anywhere に付属する ODBC ソフトウェアの一部です。ODBC バージョン 3 ドライバ・マネージャは、ODBC データ・ソースを設定するための新しいインタフェースを備えています。

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。*odbc.h* 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 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。詳細については、「Windows CE での ODBC アプリケーションのリンク」284 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、Adaptive Server Anywhere の ODBC ドライバの *dbodbc9.dll* をロードします。

Microsoft、Watcom、Borland 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、Adaptive Server 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	*dbodbc9.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 (*dbodbc9.lib*) では、Adaptive Server Anywhere の ODBC ドライバ *dbodbc9.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 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¥Sybase¥SQL Anywhere 9¥ce¥arm.30
```

サポートしている Windows CE のバージョンについては、『SQL Anywhere Studio の紹介』> 「Windows オペレーティング・システムと NetWare オペレーティング・システム」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 *dbodbc9.lib* で、SQL Anywhere 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¥Sybase¥SQL Anywhere 9¥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¥dbodbc9.dll;dbf=c:¥asademo.db"
```

ospath は、Windows CE デバイス上の SQL Anywhere ディレクトリのチップ固有のサブディレクトリへのフル・パスです。次に例を示します。

```
¥Program Files¥Sybase¥SQL Anywhere 9¥ce¥arm.30¥lib
```

サンプル・プログラム (*odbc.c*) は、ファイル・データ・ソース (FileDSN 接続パラメータ) である *ASA 9.0 Sample.dsn* を使用します。お使いのデスクトップ・システムで ODBC ドライバ・マネージャからファイル・データ・ソースを作成し、それを Windows CE デバイスにコピーできます。

Windows CE とユニコード

Adaptive Server 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。

Adaptive Server 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。

Unicode ODBC の機能については、*Samples¥ASA¥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 Adaptive Server Anywhere には同梱されていませんが、サード・パーティのドライバ・マネージャを利用できます。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、Adaptive Server Anywhere ODBC ドライバの別々のバージョンが用意されています。

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
Solaris/Sparc	シングルスレッド	*dbodbc9.so* (*dbodbc9.so.1*)
Solaris/Sparc	マルチスレッド	*saodbc_r.so* (*saodbc_r.so.1*)

ライブラリは、バージョン番号 (カッコ内) 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。

❖ 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適切な ODBC ドライバに直接リンクします。
2. アプリケーションを配備するときに、ユーザのライブラリ・パス内で適切な ODBC ドライバが使用可能であることを確認します。

データ・ソース情報 ODBC ドライバ・マネージャの存在を検出しなかった場合、Adaptive Server Anywhere は *~/.odbc.ini* をデータ・ソース情報に使用します。

UNIX 上で ODBC ドライバ・マネージャを使用する

サードパーティの UNIX 用 ODBC ドライバ・マネージャを使用できます。ODBC ドライバ・マネージャには次のファイルが含まれています。

オペレーティング・システム	ファイル
Solaris/Sparc	*libodbc.so* (*libodbc.so.1*) *libodbcinst.so* (*libodbcinst.so.1*)

ODBC ドライバ・マネージャを必要とするアプリケーションを配備するときに、サードパーティのドライバ・マネージャを使用しない場合は、*libodbc* と *libodbcinst* の各共有ライブラリ用に、Adaptive Server Anywhere ODBC ドライバへのシンボリック・リンクを作成します。

ODBC ドライバ・マネージャがある場合、Adaptive Server Anywhere は *~/.odbc.ini* ではなくドライバ・マネージャにデータ・ソース情報を問い合わせます。

標準的な ODBC アプリケーションは、ODBC ドライバに直接リンクしません。代わりに、ODBC 関数呼び出しで ODBC ドライバ・マネージャにアクセスします。UNIX と Windows CE の各オペレーティング・システムでは、Adaptive Server Anywhere に ODBC ドライバ・マネージャは付属していません。Adaptive Server Anywhere ODBC ドライバに直接リンクして ODBC アプリケーションを作成することもできますが、その場合、アクセスできるのは Adaptive Server Anywhere のデータ・ソースのみとなります。

ODBC のサンプル

Adaptive Server Anywhere には、ODBC のサンプルがいくつか用意されています。各サンプルは、SQL Anywhere ディレクトリの *Samples¥ASA* サブディレクトリにあります。デフォルトのロケーションは、次のとおりです。

```
C:¥Program Files¥Sybase¥SQL Anywhere 9¥Samples¥ASA
```

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。詳細なサンプル ODBC プログラムは、*Samples¥ASA¥C¥odbc.c* として提供されます。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」179 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¥ASA¥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、SQL Anywhere ディレクトリの *Samples¥ASA¥C* サブ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 ESQL 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、**WINNT** 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

サンプル・プログラム *odbc.c* は、サービスをサポートするバージョンの Windows 用にコンパイルすると、オプションでサービスとして動作します。

Windows サービスのサンプル用のファイルは、ソース・ファイル *ntsvc.c* とヘッダ・ファイル *ntsvc.h* の 2 つです。このコードを使用すると、リンク済みの実行プログラムを、通常の実行プログラムまたは Windows サービスとして実行できます。

❖ 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¥ASA¥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、**Customer** または **Employee** と入力します。

❖ ODBC のサンプルを Windows サービスとして実行するには、次の手順に従います。

1. Sybase Central を起動します。
2. 左ウィンドウ枠で、Adaptive Server Anywhere 9 を選択します。
3. 右ウィンドウ枠で、[サービス] タブを選択します。
4. [ファイル] メニューから [新規] ー [サービス] を選択します。

   サービス作成ウィザードが表示されます。
5. 最初のページで、サービスの名前を入力します。
6. 2 番目のページで、サンプル・プログラムを選択します。
7. 3 番目のページで、SQL Anywhere ディレクトリの *Samples¥ASA¥C* サブディレクトリからサンプル・プログラム (*odbcwnt.exe*) を選択します。
8. ウィザードを完了して、サービスをインストールします。
9. メイン・ウィンドウの [開始] をクリックして、サービスを起動します。

サービスとして実行すると、このプログラムは可能であれば通常のユーザ・インタフェースを表示します。さらに、出力をアプリケーション・イベント・ログに書き込みます。ユーザ・インタフェースを表示できないときは、1 ページ分のデータをアプリケーション・イベント・ログに出力して停止します。

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」314 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、Adaptive Server 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、SQL Anywhere ディレクトリに *Samples¥ASA¥ODBCConnect¥odbcconnect.cpp* として格納されています。

```
#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*) "ASA 9.0 Sample", 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;
  }
```

データ・ソースへの接続

この項では、ODBC 関数を使用して Adaptive Server Anywhere データベースへの接続を確立する方法について説明します。

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 とパスワードを取ります。データ・ソース名をアプリケーションにハードエンコードする場合は、**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 Adaptive Server Anywhere 固有の接続情報を使用できます。また、Adaptive Server 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 Adaptive Server 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。

この章の例では、主に **SQLDriverConnect** を使用します。

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、『ASA データベース管理ガイド』> 「接続パラメータ」を参照してください。

接続の確立

アプリケーションが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。

次に例を示します。

```
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エンコードしたり、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。

次に例を示します。

```
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。次に例を示します。

```
retcode = SQLSetConnectAttr( dbc,
    SQL_AUTOCOMMIT,
    (SQLPOINTER)SQL_AUTOCOMMIT_OFF, 0 );
```

詳細については、「接続属性の設定」298 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。

次に例を示します。

```
if (retcode == SQL_SUCCESS
     || retcode == SQL_SUCCESS_WITH_INFO) {
     printf( "dbc allocated¥n" );
     retcode = SQLConnect( dbc,
          (SQLCHAR*) "ASA 9.0 Sample", SQL_NTS,
          (SQLCHAR* ) "DBA", SQL_NTS,
          (SQLCHAR*) "SQL", SQL_NTS );
     if (retcode == SQL_SUCCESS
               || retcode == SQL_SUCCESS_WITH_INFO){
     // successfully connected.
```

詳細例は、SQL Anywhere ディレクトリに *Samples¥ASA¥ODBCConnect¥odbcconnect.cpp* として格納されています。

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」298 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、『ASA データベース管理ガイド』> 「接続パラメータ」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

Adaptive Server 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**　Adaptive Server 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。

「準備文の実行」302 ページ を参照。

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、オプションでパラメータを指定できます。

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」300 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 department WHERE dept_id = 201";
SQLExecDirect( stmt, deletestmt, SQL_NTS) ;
```

エラー・チェックを含む詳細例については、*Samples¥ASA¥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最初、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 20
 SQLINTEGER cbDeptID = 0,
    cbDeptName = SQL_NTS, cbManagerID = 0;
 SQLCHAR deptname[ DEPT_NAME_LEN ];
 SQLSMALLINT deptID, managerID;
 SQLCHAR insertstmt[ STMT_LEN ] =
   "INSERT INTO department "
   "( dept_id, dept_name, dept_head_id )"
   "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」302 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・チェックを含む詳細例については、*Samples¥ASA¥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」14 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 department
              ( dept_id, dept_name, dept_head_id )
              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 dept_id 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 ～ 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・チェックを含む詳細例については、*Samples¥ASA¥ODBCPrepare¥odbcprepare.cpp* を参照してください。

SQLEPrepare 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。Adaptive Server 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」23 ページを参照してください。

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、Adaptive Server 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」307 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。Adaptive Server 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。Adaptive Server 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, 0 );
```

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 **SQLExtendedFetch** を使用し、カーソルを介してローをフェッチします。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_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。

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、詳細例から抜粋したものです。詳細例は、*Samples¥ASA¥ODBCSelect¥odbcselect.cpp* にあります。

```
  SQLINTEGER cbDeptID = 0, cbDeptName = SQL_NTS,
  cbManagerID = 0;
   SQLCHAR deptname[ DEPT_NAME_LEN ];
   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*) "ASA 9.0 Sample", 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 dept_id, dept_name, dept_head_id FROM
DEPARTMENT "
                      "ORDER BY dept_id", 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。integer に格納できる値より 1 小さい 2147483646 までの番号が付けられたローをフェッチできます。ローの位置番号に、クエリ結果の最後を基準として負の数を使用している場合、integer に格納できる負の最大値より 1 大きい数までの番号のローをフェッチできます。

カーソルを使用したローの更新と削除

Microsoft の『ODBC Programmer's Reference』では、クエリが位置付けオペレーションを使用して更新可能であることを示すために、SELECT… FOR UPDATE を使用するように提案しています。Adaptive Server 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、『ASA データベース管理ガイド』> 「ANSI_UPDATE_CONSTRAINTS オプション [互換性]」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、Adaptive Server 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、Adaptive Server 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。Adaptive Server Anywhere は、動的カーソルを除くすべての種類のカーソルにブックマークをサポートします。

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、Adaptive Server 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、『ASA SQL ユーザーズ・ガイド』>「プロシージャ、トリガ、バッチの使用」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 emp_id[ 10 ];
 char emp_lname[ 20 ];

 /* Create the procedure */
 SQLExecDirect( stmt,
         "CREATE PROCEDURE employees()" ¥
         " RESULT( emp_id CHAR(10), emp_lname
CHAR(20))"¥
         " BEGIN" ¥
         " SELECT emp_id, emp_lname FROM employee" ¥
         " 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, &emp_id,
                  sizeof(emp_id), NULL );
 SQLBindCol( stmt, 2, SQL_C_CHAR, &emp_lname,
                  sizeof(emp_lname), NULL );

 for( ;; ) {
     retcode = SQLFetch( stmt );
     if( retcode == SQL_NO_DATA_FOUND ) {
```

```
        retcode = SQLMoreResults( stmt );
    if( retcode == SQL_NO_DATA_FOUND ) break;
}     else {
        do_something( emp_id, emp_lname );
    }
}
```

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。

ステータス・コード	説明
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 sales_order_items 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 sales_order_items 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 8 章

データベース・ツール・インタフェース

この章の内容

この章では、Adaptive Server Anywhere に付属するデータベース・ツール・ライブラリを使用して、データベース管理機能を C または C++ アプリケーションに追加する方法について説明します。

データベース・ツール・インタフェースの概要

Sybase Adaptive Server 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行うことができます。

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 *dbtool9.dll* で、UNIX の場合は *libdbtool9.so* (非スレッド) または *libdbtool9_r.so* (スレッド) です。

UNIX の非スレッド・ライブラリ (旧式)

UNIX の非スレッド・ライブラリは推奨されていません。現在のソフトウェアでは完全にサポートされていますが、SQL Anywhere Studio の次のメジャー・リリースではサポートされません。

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 *dbtool9.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

Adaptive Server 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、インストール・ディレクトリの下の *h* サブディレクトリにインストールされます。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・ファイルは他の 2 つのファイルをインクルードします。

- **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。

また、*sqldef.h* ヘッダ・ファイルはエラー戻り値も含みます。

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 Adaptive Server Anywhere に付属しており、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
Watcom	*win32¥dbtlstw.lib*
Microsoft	*win32¥dbtlstM.lib*
Borland	*win32¥dbtlstB.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 Adaptive Server 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=asademo.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」347 ページを参照してください。

ソフトウェア・コンポーネントのリターン・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リターン・コードは、次のとおりです。

コード	説明
0	成功
1	一般的な失敗
2	無効なファイル・フォーマット
3	ファイルが見つからない、開くことができない
4	メモリがない
5	ユーザによる終了
6	通信失敗
7	必要なデータベース名なし
8	クライアントとサーバのプロトコルが一致しない
9	データベース・サーバと接続できない
10	データベース・サーバが起動されない

コード	説明
11	データベース・サーバが見つからない
254	タイムアウト
255	コマンド・ラインで無効なパラメータ

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 (バックアップ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、Adaptive Server 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;
     if( question != NULL ) {
         ret = MessageBox( HwndParent, question,
         "Confirm", MB_ICONEXCLAMTION|MB_YESNO );
     }
     return( 0 );
 }
```

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 ) {
         ret = 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 * errorstr )
 {
     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。この場合も、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 ) {
         return FALSE;
     }
     OutputMessageToWindow( statustr );
     return TRUE;
 }
```

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。

- 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。

アプリケーションのバージョンより DBTools ライブラリのバージョンのほうが古いときは、アプリケーションは実行できません。

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        backup_writefile: 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 の整数値をビット・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については、SQL Anywhere ディレクトリの *Samples¥ASA¥DBTools* サブディレクトリを参照してください。サンプル・プログラム自体は *Samples¥ASA¥DBTools¥main.cpp* です。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 WINNT

 #include <stdio.h>
 #include "windows.h"
 #include "string.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", str );
         fprintf( stdout, "¥n" );
         fflush( stdout );
     }
     return 0;
 }

extern short _callback StatusCallBack( char * str )
 {
     if( str != NULL )
     {
         fprintf( stdout, "%s", str );
         fprintf( stdout, "¥n" );
         fflush( stdout );
     }
     return 0;
 }

extern short _callback ErrorCallBack( char * str )
 {
     if( str != NULL )
     {
         fprintf( stdout, "%s", str );
         fprintf( stdout, "¥n" );
         fflush( stdout );
     }
     return 0;
 }
```

```
typedef void (CALLBACK *DBTOOLSPROC)( 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;
     DBTOOLSPROC         dbbackup;
     DBTOOLSPROC         dbtoolsinit;
     DBTOOLSPROC         dbtoolsfini;


// Always initialize to 0 so new versions
     // of the structure will be compatible.
     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=ASA 9.0 Sample" );
     }
     backup_info.connectparms = connect;
     backup_info.startline = "";
     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( "dbtool9.dll" );
     if( hinst == NULL )
     {
         // Failed
         return 0;
     }
     dbt_info.errorrtn = (MSG_CALLBACK)ErrorCallBack;
     dbbackup = (DBTOOLSPROC) GetProcAddress(
(HMODULE)hinst,
         "_DBBackup@4" );
     dbtoolsinit = (DBTOOLSPROC) GetProcAddress(
(HMODULE)hinst,
         "_DBToolsInit@4" );
     dbtoolsfini = (DBTOOLSPROC) GetProcAddress(
(HMODULE)hinst,
         "_DBToolsFini@4" );
     (*dbtoolsinit)( &dbt_info );
     (*dbbackup)( &backup_info );
     (*dbtoolsfini)( &dbt_info );
     FreeLibrary( hinst );
     return 0;
 }
```

DBTools 関数

この項では、DBTools ライブラリ内の使用可能な関数について説明します。関数はアルファベット順に示します。

DBBackup 関数

機能

データベース・バックアップ関数。この関数を使用するのは、*dbbackup* コマンド・ライン・ユーティリティです。

プロトタイプ

short **DBBackup (** const a_backup_db * *backup-db* **);**

パラメータ

パラメータ	説明
backup-db	「a_backup_db 構造体」347 ページへのポインタ

戻り値

「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法

DBBackup 関数は、すべてのデータベース・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、『ASA データベース管理ガイド』>「バックアップ・ユーティリティ」を参照してください。

参照

「a_backup_db 構造体」347 ページ

DBChangeLogName 関数

機能

トランザクション・ログ・ファイルの名前を変更します。この関数を使用するのは、*dblog* コマンド・ライン・ユーティリティです。

プロトタイプ

short **DBChangeLogName (** const a_change_log * *change-log* **);**

パラメータ

パラメータ	説明
change-log	「a_change_log 構造体」349 ページへのポインタ

戻り値

「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法

dblog コマンド・ライン・ユーティリティに -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、『ASA データベース管理ガイド』>「トランザクション・ログ・ユーティリティ」を参照してください。

参照

「a_change_log 構造体」349 ページ

DBChangeWriteFile 関数

機能

他のデータベース・ファイルを参照するようライト・ファイルを変更します。この関数を使用するのは、-d オプションが指定された *dbwrite* コマンド・ライン・ユーティリティです。

プロトタイプ

short **DBChangeWriteFile (** const a_writefile * *writefile* **);**

パラメータ

パラメータ	説明
writefile	「a_writefile 構造体」395 ページへのポインタ

戻り値

「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法

ライト・ファイル・ユーティリティとその機能の詳細については、『ASA データベース管理ガイド』>「ライト・ファイル・ユーティリティ (旧式)」を参照してください。

参照

「DBCreateWriteFile 関数」335 ページ

「DBStatusWriteFile 関数」340 ページ

「a_writefile 構造体」395 ページ

DBCollate 関数

機能

データベースから照合順序を抽出します。

プロトタイプ

short **DBCollate (** const a_db_collation * *db-collation* **);**

パラメータ

パラメータ	説明
db-collation	「a_db_collation 構造体」357 ページへのポインタ

戻り値

「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法

照合ユーティリティとその機能の詳細については、『ASA データベース管理ガイド』> 「照合ユーティリティ」を参照してください。

参照

「a_db_collation 構造体」357 ページ

DBCompress 関数

機能

データベース・ファイルを圧縮します。この関数を使用するのは、*dbshrink* コマンド・ライン・ユーティリティです。

プロトタイプ

short **DBCompress (** const a_compress_db * *compress-db* **);**

パラメータ

パラメータ	説明
compress-db	「a_compress_db 構造体」352 ページへのポインタ

戻り値　「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法　圧縮ユーティリティとその機能の詳細については、『ASA データベース管理ガイド』>「圧縮ユーティリティ (旧式)」を参照してください。

参照　「a_compress_db 構造体」352 ページ

DBCreate 関数

機能　データベースを作成します。この関数を使用するのは、*dbinit* コマンド・ライン・ユーティリティです。

プロトタイプ　short **DBCreate (** const a_create_db * *create-db* **);**

パラメータ

パラメータ	説明
create-db	「a_create_db 構造体」354 ページへのポインタ

戻り値　「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法　初期化ユーティリティの詳細については、『ASA データベース管理ガイド』>「初期化ユーティリティ」を参照してください。

参照　「a_create_db 構造体」354 ページ

DBCreateWriteFile 関数

機能　ライト・ファイルを作成します。この関数を使用するのは、`-c` オプションが指定された *dbwrite* コマンド・ライン・ユーティリティです。

プロトタイプ　short **DBCreateWriteFile (** const a_writefile * *writefile* **);**

パラメータ

パラメータ	説明
writefile	「a_writefile 構造体」395 ページへのポインタ

戻り値 「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法 ライト・ファイル・ユーティリティとその機能の詳細については、『ASA データベース管理ガイド』>「ライト・ファイル・ユーティリティ (旧式)」を参照してください。

参照 「DBChangeWriteFile 関数」333 ページ

「DBStatusWriteFile 関数」340 ページ

「a_writefile 構造体」395 ページ

DBErase 関数

機能 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。この関数を使用するのは、*dberase* コマンド・ライン・ユーティリティです。

プロトタイプ short **DBErase (** const an_erase_db * *erase-db* **);**

パラメータ

パラメータ	説明
erase-db	「an_erase_db 構造体」364 ページへのポインタ

戻り値 「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法 消去ユーティリティとその機能の詳細については、『ASA データベース管理ガイド』>「消去ユーティリティ」を参照してください。

参照 「an_erase_db 構造体」364 ページ

DBExpand 関数

機能

データベース・ファイルを展開します。この関数を使用するのは、*dbexpand* コマンド・ライン・ユーティリティです。

プロトタイプ

short **DBExpand (** const an_expand_db * *expand-db* **);**

パラメータ

パラメータ	説明
expand_db	「an_expand_db 構造体」365 ページへのポインタ

戻り値

「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法

展開ユーティリティとその機能の詳細については、『ASA データベース管理ガイド』>「展開ユーティリティ (旧式)」を参照してください。

参照

「an_expand_db 構造体」365 ページ

DBInfo 関数

機能

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、*dbinfo* コマンド・ライン・ユーティリティです。

プロトタイプ

short **DBInfo (** const a_db_info * *db-info* **);**

パラメータ

パラメータ	説明
db-info	「a_db_info 構造体」359 ページへのポインタ

戻り値

「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法

情報ユーティリティとその機能の詳細については、『ASA データベース管理ガイド』>「情報ユーティリティ」を参照してください。

参照

「DBInfoDump 関数」338 ページ

「DBInfoFree 関数」338 ページ

「a_db_info 構造体」359 ページ

DBInfoDump 関数

機能

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* コマンド・ライン・ユーティリティです。

プロトタイプ

short **DBInfoDump (** const a_db_info * *db-info* **);**

パラメータ

パラメータ	説明
db-info	「a_db_info 構造体」359 ページへのポインタ

戻り値

「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法

情報ユーティリティとその機能の詳細については、『ASA データベース管理ガイド』> 「情報ユーティリティ」を参照してください。

参照

「DBInfo 関数」337 ページ

「DBInfoFree 関数」338 ページ

「a_db_info 構造体」359 ページ

DBInfoFree 関数

機能

DBInfoDump 関数の呼び出し後に、リソースを解放するために呼び出されます。

プロトタイプ

short **DBInfoFree (** const a_db_info * *db-info* **);**

パラメータ

パラメータ	説明
db-info	「a_db_info 構造体」359 ページへのポインタ

戻り値

「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法

情報ユーティリティとその機能の詳細については、『ASA データベース管理ガイド』>「情報ユーティリティ」を参照してください。

参照

「DBInfo 関数」337 ページ

「DBInfoDump 関数」338 ページ

「a_db_info 構造体」359 ページ

DBLicense 関数

機能

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するために呼び出されます。

プロトタイプ

short **DBLicense (** const a_db_lic_info * *db-lic-info* **);**

パラメータ

パラメータ	説明
db-lic-info	「a_dblic_info 構造体」362 ページへのポインタ

戻り値

「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法

情報ユーティリティとその機能の詳細については、『ASA データベース管理ガイド』>「情報ユーティリティ」を参照してください。

参照

「a_dblic_info 構造体」362 ページ

DBStatusWriteFile 関数

機能

ライト・ファイルのステータスを取得します。この関数を使用するのは、`-s` オプションが指定された *dbwrite* コマンド・ライン・ユーティリティです。

プロトタイプ

short **DBStatusWriteFile (** const a_writefile * *writefile* **);**

パラメータ

パラメータ	説明
writefile	「a_writefile 構造体」395 ページへのポインタ

戻り値

「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法

ライト・ファイル・ユーティリティとその機能の詳細については、『ASA データベース管理ガイド』>「ライト・ファイル・ユーティリティ (旧式)」を参照してください。

参照

「DBChangeWriteFile 関数」333 ページ

「DBCreateWriteFile 関数」335 ページ

「a_writefile 構造体」395 ページ

DBSynchronizeLog 関数

機能

データベースを Mobile Link 同期サーバと同期させます。

プロトタイプ

short **DBSynchronizeLog(** const a _sync_db * *sync-db* **);**

パラメータ

パラメータ	説明
sync-db	「a_sync_db 構造体」367 ページへのポインタ

戻り値　「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法　アクセスできる機能の詳細については、『Mobile Link クライアント』> 「同期の開始」 を参照してください。

参照　『Mobile Link クライアント』> 「dbmlsync の DBTools インタフェース」

DBToolsFini 関数

機能　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 * *dbtools-info* **);**

パラメータ

パラメータ	説明
dbtools-info	「a_dbtools_info 構造体」363 ページへのポインタ

戻り値　「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法　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」341 ページ

「a_dbtools_info 構造体」363 ページ

DBToolsInit 関数

機能　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 * *dbtools-info* **);**

パラメータ

パラメータ	説明
dbtools-info	「a_dbtools_info 構造体」363 ページへのポインタ

戻り値

「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法

DBToolsInit 関数の主な目的は、Adaptive Server 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

- 次のコード例は、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。

```
a_dbtools_info    info;
 short            ret;

 memset( &info, 0, sizeof( a_dbtools_info) );
info.errorrtn = (MSG_CALLBACK)MakeProcInstance(
(FARPROC)MyErrorRtn, hInst );

 // initialize DBTools
 ret = DBToolsInit( &info );
 if( ret != EXIT_OKAY ) {
         // DLL initialization failed
         ...
     }
     // call some DBTools routines . . .
     ...
     // cleanup the DBTools dll
     DBToolsFini( &info );
```

参照

「DBToolsFini 関数」341 ページ

「a_dbtools_info 構造体」363 ページ

DBToolsVersion 関数

機能　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

使用法　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」327 ページ

DBTranslateLog 関数

機能　トランザクション・ログ・ファイルを SQL に変換します。この関数を使用するのは、*dbtran* コマンド・ライン・ユーティリティです。

プロトタイプ　short **DBTranslateLog (** const a_translate_log * *translate-log* **);**

パラメータ

パラメータ	説明
translate-log	「a_translate_log 構造体」378 ページへのポインタ

戻り値　「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法　ログ変換ユーティリティの詳細については、『ASA データベース管理ガイド』>「ログ変換ユーティリティ」を参照してください。

参照　「a_translate_log 構造体」378 ページ

DBTruncateLog 関数

機能

トランザクション・ログ・ファイルをトランケートします。この関数を使用するのは、*dbbackup* コマンド・ライン・ユーティリティです。

プロトタイプ

short **DBTruncateLog (** const a_truncate_log * *truncate-log* **);**

パラメータ

パラメータ	説明
truncate-log	「a_truncate_log 構造体」383 ページへのポインタ

戻り値

「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法

バックアップ・ユーティリティの詳細については、『ASA データベース管理ガイド』>「バックアップ・ユーティリティ」を参照してください。

参照

「a_truncate_log 構造体」383 ページ

DBUnload 関数

機能

データベースをアンロードします。この関数を使用するのは、*dbunload* コマンド・ライン・ユーティリティと、SQL Remote 用の *dbxtract* ユーティリティです。

プロトタイプ

short **DBUnload (** const an_unload_db * *unload-db* **);**

パラメータ

パラメータ	説明
unload-db	「an_unload_db 構造体」385 ページへのポインタ

戻り値

「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法　アンロード・ユーティリティの詳細については、『ASA データベース管理ガイド』>「アンロード・ユーティリティ」を参照してください。

参照　「an_unload_db 構造体」385 ページ

DBUpgrade 関数

機能　データベース・ファイルをアップグレードします。この関数を使用するのは、*dbupgrade* コマンド・ライン・ユーティリティです。

プロトタイプ　short **DBUpgrade (** const an_upgrade_db * *upgrade-db* **);**

パラメータ

パラメータ	説明
upgrade-db	「an_upgrade_db 構造体」391 ページへのポインタ

戻り値　「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法　アップグレード・ユーティリティの詳細については、『ASA データベース管理ガイド』>「アップグレード・ユーティリティ」を参照してください。

参照　「an_upgrade_db 構造体」391 ページ

DBValidate 関数

機能　データベースの全部または一部を検証します。この関数を使用するのは、*dbvalid* コマンド・ライン・ユーティリティです。

プロトタイプ　short **DBValidate (** const a_validate_db * *validate-db* **);**

パラメータ

パラメータ	説明
validate-db	「a_validate_db 構造体」393 ページへのポインタ

戻り値

「ソフトウェア・コンポーネントのリターン・コード」324 ページにリストされているリターン・コード

使用法

アップグレード・ユーティリティの詳細については、『ASA データベース管理ガイド』> 「検証ユーティリティ」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

「a_validate_db 構造体」393 ページ

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。

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) コマンド・ライン・オプションに対応しています。

a_backup_db 構造体

機能

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 *         startline;
    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         backup_writefile  : 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_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"`

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=c:¥asa¥asademo.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。次はその例です。
	
	`"START=d:¥asa90¥win32¥dbeng9.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。
	
	`"userid=dba;password=sql;dbf=d:¥asa90¥asademo.db;start=d:¥asa90¥win32¥dbeng9.exe"`
	
	接続文字列オプションの全種類については、『ASA データベース管理ガイド』> 「接続パラメータ」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 connectparams 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

メンバ	説明
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)
backup_writefile	ライト・ファイルを使用している場合、データベース・ライト・ファイルをバックアップする (1) またはしない (0)
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	

参照

「DBBackup 関数」332 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」325 ページ を参照してください。

a_change_log 構造体

機能

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        mirrorname_present   : 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;
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
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)
mirrorname_present	1 に設定。DBTools のバージョンが、mirrorname フィールドをサポートするのに十分新しいことを示す。
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

メンバ	説明
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。
key_file	暗号化キーを保持するファイル
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」332 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」325 ページ を参照してください。

a_compress_db 構造体

機能

DBTools ライブラリを使用してデータベースの圧縮を実行するために必要な情報を格納します。

構文

```
typedef struct a_compress_db {
    unsigned short      version;
    const char *        dbname;
    const char *        compress_name;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        statusrtn;
    a_bit_field         display_free_pages   : 1;
    a_bit_field         quiet          : 1;
    a_bit_field         record_unchanged   : 1;
    a_compress_stats *      stats;
    MSG_CALLBACK        confirmrtn;
    a_bit_field         noconfirm      : 1;
    const char *        encryption_key;
    } a_compress_db;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号
dbname	圧縮するデータベースのファイル名
compress_name	圧縮されたデータベースのファイル名
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
display_free_pages	空きページ情報を表示する
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
record_unchanged	1 に設定。a_compress_stats 構造体が充分に新しく、**unchanged** メンバを持つことを示す。
a_compress_stats	a_compress_stats 型の構造体へのポインタ。メンバが NULL でなく、display_free_pages が 0 でない場合に設定される。
confirmrtn	動作確認コールバック・ルーチン
noconfirm	操作の確認をする (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

「DBCompress 関数」334 ページ

「a_compress_stats 構造体」353 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」325 ページ を参照してください。

a_compress_stats 構造体

機能

圧縮されたデータベース・ファイルの統計に関する情報を格納します。

構文

```
typedef struct a_compress_stats {
    a_stats_line    tables;
    a_stats_line    indices;
    a_stats_line    other;
    a_stats_line    free;
    a_stats_line    total;
    a_sql_int32     free_pages;
    a_sql_int32     unchanged;
    } a_compress_stats;
```

メンバ

メンバ	説明
tables	テーブルに関する圧縮情報

メンバ	説明
indices	インデックスに関する圧縮情報
other	その他の圧縮情報
free	空き領域に関する情報
total	全体的な圧縮情報
free_pages	空きページに関する情報
unchanged	圧縮アルゴリズムによって縮小できなかったページ数

参照

「DBCompress 関数」334 ページ

「a_compress_db 構造体」352 ページ

a_create_db 構造体

機能

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;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    short           database_version;
    char            verbose;
    a_bit_field         blank_pad       : 2;
    a_bit_field         respect_case    : 1;
    a_bit_field         encrypt         : 1;
    a_bit_field         debug           : 1;
    a_bit_field     dbo_avail       : 1;
    a_bit_field         mirrorname_present  : 1;
    a_bit_field         avoid_view_collisions   : 1;
    short           collation_id;
    const char *    dbo_username;
```

```
const char *          mirrorname;
a_bit_field          java_classes : 1;
a_bit_field          jconnect : 1;
const char *          data_store_type;
const char *          encryption_key;
const char *          encryption_algorithm;
const char *          jdk_version;
a_bit_field       respect_password_case    : 1;
a_bit_field  checksum : 1;
} a_create_db;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号
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。startline の例 :
	
	`"c:¥asa¥win32¥dbeng9.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 :
	
	`"dbeng9 -gp <PAGE_SIZE> -C 10M"`
	
	(-c 10M はページ・サイズが >=2048 の場合にのみ追加されます)

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
database_version	データベースのバージョン番号
verbose	冗長モードで実行

メンバ	説明
blank_pad	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
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	データベースの暗号化
debug	予約
dbo_avail	1 に設定。このデータベースで dbo ユーザが使用可能。
mirrorname_present	1 に設定。DBTools のバージョンが、mirrorname フィールドをサポートするのに十分新しいことを示す
avoid_view_collisions	Watcom SQL 互換ビュー SYS.SYSCOLUMNS と SYS.SYSINDEXES の世代を除外する
collation_id	照合識別子
dbo_username	旧式
mirrorname	トランザクション・ログ・ミラー名
encryption_dllname	データベースの暗号化に使用する DLL
java_classes	Java 実行可能のデータベースを作成する
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
encryption_algorithm	暗号化アルゴリズム (**AES**)
jdk_version	*dbinit* `-jdk` オプションの値の 1 つ
respect_password_case	
checksum	

参照 「DBCreate 関数」335 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」325 ページ を参照してください。

a_db_collation 構造体

機能

DBTools ライブラリを使用してデータベースから照合順を抽出するために必要な情報を格納します。

構文

```
typedef struct a_db_collation {
    unsigned short      version;
    const char *        connectparms;
    const char *        startline;
    const char *        collation_label;
    const char *        filename;
    MSG_CALLBACK        confirmrtn;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    a_bit_field         include_empty       : 1;
    a_bit_field         hex_for_extended    : 1;
    a_bit_field         replace             : 1;
    a_bit_field         quiet               : 1;
    const char *        input_filename;
    } a_db_collation;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号

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=c:¥asa¥asademo.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。次はその例です。
	
	`"START=d:¥asa90¥win32¥dbeng9.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。
	
	`"userid=dba;password=sql;dbf=d:¥asa90¥asademo.db;start=d:¥asa90¥win32¥dbeng9.exe"`
	
	接続文字列オプションの全種類については、『ASA データベース管理ガイド』>「接続パラメータ」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 connectparams 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
include_empty	照合順のギャップに対して空のマッピングを書き込む
hex_for_extended	2 桁の 16 進数を使用して、ハイバリュー文字を表す
replace	動作を確認しないで操作する
quiet	メッセージを出さずに操作する
input_filename	入力の照合定義

参照

「DBCollate 関数」334 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」325 ページ を参照してください。

a_db_info 構造体

機能

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 *    dbname;
    unsigned short    dbbufsize;
    char *    dbnamebuffer;
    unsigned short    logbufsize;
    char *    lognamebuffer;
    unsigned short    wrtbufsize;
    char *    wrtnamebuffer;
    a_bit_field    quiet : 1;
    a_bit_field    mirrorname_present : 1;
    a_sysinfo    sysinfo;
    a_sql_uint32    free_pages;
    a_bit_field    compressed : 1;
    const char *    connectparms;
    const char *    startline;
    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;
    char *    collationnamebuffer;
    unsigned short    collationnamebufsize;
    char *    classesversionbuffer;
    unsigned short    classesversionbufsize;
    a_bit_field    checksum : 1;
     } 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
dbname	データベース・ファイル名
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
wrtbufsize	wrtnamebuffer メンバの長さ
wrtnamebuffer	ライト・ファイル名
quiet	確認メッセージを出さずに操作する
mirrorname_present	1 に設定。DBTools のバージョンが、mirrorname フィールドをサポートするのに十分新しいことを示す。
sysinfo	a_sysinfo 構造体へのポインタ
free_pages	空きページ数
compressed	圧縮されている場合は 1、圧縮されていなければ 0

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=c:¥asa¥asademo.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。次はその例です。
	
	`"START=d:¥asa90¥win32¥dbeng9.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。
	
	`"userid=dba;password=sql;dbf=d:¥asa90¥asademo.db;start=d:¥asa90¥win32¥dbeng9.exe"`
	
	接続文字列オプションの全種類については、『ASA データベース管理ガイド』>「接続パラメータ」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 connectparams 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	未使用ページ数
other_pages	テーブル・ページでもインデックス・ページでもないページの数

メンバ	説明
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
classesversionbuffer	インストールされている Java クラスの JDK バージョン (1.1.3、1.1.8、1.3 など)、またはデータベースに Java クラスがインストールされていない場合は空の文字列 (最大サイズは 10+1)
classesversionbufsize	classesversionbuffer 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

参照

「DBInfo 関数」337 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」325 ページ を参照してください。

a_dblic_info 構造体

機能

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        *       platform_str;
    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;
    } a_dblic_info;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号
exename	実行プログラム名
username	ライセンスのユーザ名
compname	ライセンスの会社名
platform_str	オペレーティング・システム。WinNT、NLM、または UNIX。
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。

a_dbtools_info 構造体

機能

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」341 ページ

「DBToolsInit 関数」341 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」325 ページ を参照してください。

an_erase_db 構造体

機能

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)

メンバ	説明
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」336 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」325 ページ を参照してください。

an_expand_db 構造体

機能

DBTools ライブラリを使用してデータベースを拡張するために必要な情報を格納します。

構文

```
typedef struct an_expand_db {
    unsigned short      version;
    const char *        compress_name;
    const char *        dbname;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        statusrtn;
    a_bit_field         quiet : 1;
    MSG_CALLBACK        confirmrtn;
    a_bit_field         noconfirm   : 1;
    const char *          unused_field;
    const char *        encryption_key;
    } an_expand_db;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号
compress_name	圧縮されたデータベース・ファイルの名前
dbname	データベース・ファイル名
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

メンバ	説明
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)
confirmrtn	動作確認コールバック・ルーチン
noconfirm	操作の確認をする (0) またはしない (1)
key_file	暗号化キーを保持するファイル
encryption_key	データベース・ファイルの暗号化キー

参照

「DBExpand 関数」337 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」325 ページ を参照してください。

a_name 構造体

機能

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
p_name	直前の a_name 構造体へのポインタ

参照

「a_translate_log 構造体」378 ページ

「a_validate_db 構造体」393 ページ

「an_unload_db 構造体」385 ページ

a_stats_line 構造体

機能

DBTools ライブラリを使用してデータベースを圧縮および拡張するために必要な情報を格納します。

構文

```
typedef struct a_stats_line {
    a_sql_int32     pages;
    a_sql_int32     bytes;
    a_sql_int32     compressed_bytes;
    } a_stats_line;
```

メンバ

メンバ	説明
pages	ページ数
bytes	展開されたデータベースのバイト数
compressed_bytes	圧縮されたデータベースのバイト数

参照

「a_compress_stats 構造体」353 ページ

a_sync_db 構造体

機能

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_sql_uint32     dnld_read_size;
```

```
    a_sql_uint32    dnld_fail_len;
    a_sql_uint32    upld_fail_len;
} 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=c:¥asa¥asademo.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。次はその例です。
	
	`"START=d:¥asa90¥win32¥dbeng9.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。
	
	`"userid=dba;password=sql;dbf=d:¥asa90¥asademo.db;start=d:¥asa90¥win32¥dbeng9.exe"`
	
	接続文字列オプションの全種類については、『ASA データベース管理ガイド』> 「接続パラメータ」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。切り替え後にコマンド・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。

メンバ	説明
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。
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	旧式。0 を使用。
debug_dump_hex	旧式。0 を使用。
debug_dump_char	旧式。0 を使用。
debug_page_offsets	旧式。0 を使用。
use_hex_offsets	旧式。0 を使用。
use_relative_offsets	旧式。0 を使用。
output_to_file	旧式。0 を使用。
output_to_mobile_link	旧式。1 を使用。
dl_use_put	旧式。0 を使用。
dl_use_upsert	旧式。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 NT/2000/XP のみ)。

メンバ	説明
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)。
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	進行状況を示す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	進行状況を示す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。

メンバ	説明
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。
delay_ml_disconn	-x が指定されている場合は TRUE。
dlg_launch_focus	予約。0 を使用。
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。

メンバ	説明
ping	-pi が指定されている場合は TRUE。
_unused	予約。0 を使用。
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。
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。
strictly_ignore_trigger_ops	予約。0 を使用。
_unused2	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。
_unused3	予約。0 を使用。

メンバ	説明
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 スイッチで指定した値。
dnld_fail_len	予約。0 を使用。
upld_fail_len	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、『Mobile Link クライアント』> 「Mobile Link 同期クライアント」を参照してください。

参照

『Mobile Link クライアント』> 「dbmlsync の DBTools インタフェース」

「DBSynchronizeLog 関数」340 ページ

a_syncpub 構造体

機能

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

『Mobile Link クライアント』> 「dbmlsync の DBTools インタフェース」

a_sysinfo 構造体

機能

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;
    a_bit_field     pwd_case_sensitivity: 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。
pwd_case_sensitivity	
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」359 ページ

a_table_info 構造体

機能

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
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」359 ページ

a_translate_log 構造体

機能 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 *       reserved_1;
        const char *       reserved_2;
        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          reserved_3             : 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
バージョン	DBTools のバージョン番号

メンバ	説明
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。
sqlname	
userlist	
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	
remove_rollback	
ansi_sql	
since_checkpoint	
omit_comments	
replace	動作を確認しないで操作する
debug	予約
include_trigger_trans	
comment_trigger_trans	
since_time	

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=c:¥asa¥asademo.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。次はその例です。
	
	`"START=d:¥asa90¥win32¥dbeng9.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。
	
	`"userid=dba;password=sql;dbf=d:¥asa90¥asademo.db;start=d:¥asa90¥win32¥dbeng9.exe"`
	
	接続文字列オプションの全種類については、『ASA データベース管理ガイド』> 「接続パラメータ」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関数。
reserved_1	予約。NULL を使用。
reserved_2	予約。NULL を使用。
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。

メンバ	説明
reserved_3	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。
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	sstran 専用に予約。
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	-ek は文字列を設定。
show_undo	予約。FALSE を使用。

メンバ	説明
quiet	操作中にメッセージを出力する (0) またはしない (1)
logs_dir	

各メンバは、*dbtran* 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。

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」343 ページ

「a_name 構造体」366 ページ

「dbtran_userlist_type 列挙」398 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」325 ページ を参照してください。

a_truncate_log 構造体

機能

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 *        startline;
    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

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=c:¥asa¥asademo.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。次はその例です。
	
	`"START=d:¥asa90¥win32¥dbeng9.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。
	
	`"userid=dba;password=sql;dbf=d:¥asa90¥asademo.db;start=d:¥asa90¥win32¥dbeng9.exe"`
	
	接続文字列オプションの全種類については、『ASA データベース管理ガイド』>「接続パラメータ」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 connectparams 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	
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」344 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」325 ページ を参照してください。

an_unload_db 構造体

機能

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* SQL Remote 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         dbo_avail : 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 *        dbo_username;
    const char *        subscriber_username;
    const char *        publisher_address_type;
    const char *        publisher_address;
    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          view_iterations_present : 1;
        a_bit_short       use_internal_reload    : 1;

unsigned short       view_iterations;
        char             escape_char;
        char *           reload_connectparms;
        char *           reload_db_filename;
        a_bit_field          output_connections: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          syntax_version_7:1;
        a_bit_field          remove_java:1;
        unsigned short     reload_page_size;
        } an_unload_db;
```

メンバ

メンバ	説明
バージョン	DBTools のバージョン番号

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=c:¥asa¥asademo.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。次はその例です。
	
	`"START=d:¥asa90¥win32¥dbeng9.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。
	
	`"userid=dba;password=sql;dbf=d:¥asa90¥asademo.db;start=d:¥asa90¥win32¥dbeng9.exe"`
	
	接続文字列オプションの全種類については、『ASA データベース管理ガイド』> 「接続パラメータ」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 connectparams 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	オプションのユーザ・コールバック。

メンバ	説明
msgrtn	オプションのユーザ・コールバック。
statusrtn	オプションのユーザ・コールバック。
confirmrtn	オプションのユーザ・コールバック。
unload_type	UNLOAD_xxx (上述)。
verbose	VB_xxx (上述)。
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。
dbo_avail	旧式。
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
dbo_username	dbo_username 機能は 6.0 以降廃止 // NULL のままに設定。
subscriber_username	dbxtract の引数。
publisher_address_type	(使われていない)
publisher_address	(使われていない)

メンバ	説明
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／ mlxtract 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 を設定する。
view_iterations_present	(使われていない)
use_internal_reload	通常 TRUE に設定 // -ix/-xx は FALSE を設定 // -ii/-xi は TRUE を設定。
view_iterations	(使われていない)
escape_char	escape_char_present が TRUE の場合に使用。
reload_connectparms	データベースの再ロードのためのユーザ ID、パスワード、データベース。
reload_db_filename	データベースの再ロードで作成するファイル名。
output_connections	(内部使用)
unload_interrupted	(内部使用)

メンバ	説明
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 を指定／ -m は FALSE を指定。
exclude_hooks	dbxtract -hx は TRUE を設定。
reload_db_logname	データベースの再ロードのログ・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。
syntax_version_7	mlxtract -s7 は TRUE を設定。
remove_java	-jr は TRUE を設定。
reload_page_size	dbunload -ap は値を設定。

各メンバは、*dbunload*、*dbxtract*、*mlxtract* の各コマンド・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。

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」344 ページ

「a_name 構造体」366 ページ

「dbunload type 列挙」398 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」325 ページ を参照してください。

an_upgrade_db 構造体

機能

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 *    startline;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        statusrtn;
    a_bit_field         quiet : 1;
    a_bit_field         dbo_avail : 1;
    const char *    dbo_username;
    a_bit_field         java_classes    : 1;
    a_bit_field         jconnect    : 1;
    a_bit_field         remove_java    : 1;
    a_bit_field         java_switch_specified : 1;
    const char *        jdk_version;
    } an_upgrade_db;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号

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=c:¥asa¥asademo.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。次はその例です。
	
	`"START=d:¥asa90¥win32¥dbeng9.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。
	
	`"userid=dba;password=sql;dbf=d:¥asa90¥asademo.db;start=d:¥asa90¥win32¥dbeng9.exe"`
	
	接続文字列オプションの全種類については、『ASA データベース管理ガイド』> 「接続パラメータ」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 connectparams 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)
dbo_avail	旧式
dbo_username	dbo_username 機能は 6.0 以降廃止。NULL のままに設定。
java_classes	データベースをアップグレードして Java 実行可能にする
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
remove_java	データベースをアップグレードして Java 機能を削除する

メンバ	説明
java_switch_specified	
jdk_version	*dbinit* -jdk オプションの値の 1 つ

参照

「DBUpgrade 関数」345 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」325 ページ を参照してください。

a_validate_db 構造体

機能

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 *        startline;
    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

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=c:¥asa¥asademo.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。次はその例です。
	
	`"START=d:¥asa90¥win32¥dbeng9.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。
	
	`"userid=dba;password=sql;dbf=d:¥asa90¥asademo.db;start=d:¥asa90¥win32¥dbeng9.exe"`
	
	接続文字列オプションの全種類については、『ASA データベース管理ガイド』>「接続パラメータ」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 connectparams 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)
index	インデックスの検証
type	「a_validate_type 列挙」399 ページを参照。

参照

「DBValidate 関数」345 ページ

「a_name 構造体」366 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」325 ページ を参照してください。

a_writefile 構造体

機能

DBTools ライブラリを使用してデータベース・ライト・ファイルを管理するために必要な情報を格納します。

構文

```
typedef struct a_writefile {
    unsigned short      version;
    const char *        writename;
    const char *        wlogname;
    const char *        dbname;
    const char *        forcename;
    MSG_CALLBACK        confirmrtn;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    char            action;
    a_bit_field         quiet   : 1;
    a_bit_field         erase   : 1;
    a_bit_field         force   : 1;
    a_bit_field         mirrorname_present    : 1;
    const char *          wlogmirrorname;
    a_bit_field         make_log_and_mirror_names: 1;
    const char *          encryption_key;
    } a_writefile;
```

メンバ

メンバ	説明
version	DBTools のバージョン番号
writename	ライト・ファイル名
wlogname	ライト・ファイルの作成時のみ使用。
dbname	ライト・ファイルの変更および作成時に使用
forcename	強制ファイル名参照
confirmrtn	動作確認コールバック・ルーチン。ライト・ファイルの作成時のみ使用

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
msgrtn	情報メッセージ処理コールバック・ルーチン
action	Sybase 用に予約
quiet	操作中にメッセージを出力する (0) またはしない (1)
erase	ライト・ファイルの作成にのみ使用。消去するときに確認する (0) またはしない (1)。
force	1 の場合、ライト・ファイルが指定のファイルを強制的に指すようにする
mirrorname_present	作成時のみ使用。1 に設定。DBTools のバージョンが、mirrorname フィールドをサポートするのに十分新しいことを示す。
wlogmirrorname	トランザクション・ログ・ミラーの名前
make_log_and_mirror_names	TRUE の場合、wlogname と wlogmirrorname の値を使用してファイル名を決定
encryption_key	データベース・ファイルの暗号化キー

参照

「DBChangeWriteFile 関数」333 ページ

「DBCreateWriteFile 関数」335 ページ

「DBStatusWriteFile 関数」340 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」325 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。

冗長列挙

機能

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」354 ページ

「an_unload_db 構造体」385 ページ

ブランク埋め込み列挙

機能

blank_pad の値を指定するために 「a_create_db 構造体」354 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」354 ページ

dbtran_userlist_type 列挙

機能 「a_translate_log 構造体」378 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」378 ページ

dbunload type 列挙

機能 「an_unload_db 構造体」385 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
    };
```

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
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。
UNLOAD_NO_DATA	スキーマだけをアンロード

参照

「an_unload_db 構造体」385 ページ

a_validate_type 列挙

機能

「a_validate_db 構造体」393 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
    } 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	デフォルトのチェックとデータ・チェックで検証
VALIDATE_INDEX	デフォルトのチェックとインデックス・チェックで検証

値	説明
VALIDATE_EXPRESS	デフォルトのチェック、データ・チェック、エクスプレス・チェックで検証
VALIDATE_FULL	デフォルトのチェック、データ・チェック、インデックス・チェックで検証

参照

『ASA データベース管理ガイド』> 「dbvalid コマンド・ライン・ユーティリティを使用したデータベースの検証」

『ASA SQL リファレンス・マニュアル』> 「VALIDATE TABLE 文」

第 9 章

OLE DB と ADO プログラミング・インタフェース

この章の内容

この章では、Adaptive Server Anywhere に対して OLE DB インタフェースを使用する方法について説明します。

OLE DB インタフェースを使用するアプリケーションの多くは、直接ではなく、Microsoft ActiveX Data Objects (ADO) プログラミング・モデルを通じて、それを使用しています。この章では、Adaptive Server Anywhere を使用した ADO プログラミングについても説明します。

OLE DB の概要

OLE DB は Microsoft から提供されているデータ・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。

Adaptive Server Anywhere には **ASAProv** という「**OLE DB プロバイダ**」が入っています。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、Adaptive Server Anywhere の ODBC ドライバで Adaptive Server Anywhere にアクセスすることもできます。

Adaptive Server 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。
- Adaptive Server 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。Adaptive Server 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 Adaptive Server 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

Adaptive Server 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 で開発されるアプリケーションに、データベース機能を提供します。

サポートされるプラットフォームのリストについては、『SQL Anywhere Studio の紹介』> 「オペレーティング・システムのバージョン」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」633 ページを参照してください。

Adaptive Server 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。

Adaptive Server Anywhere OLE DB を使用して、ADO プログラミング環境から Adaptive Server 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¥ASA¥VBSampler¥vbsampler.vbp*
Microsoft eMbedded Visual Basic 3.0	*Samples¥ASA¥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 = "ASAProv"
     myConn.ConnectionString = _
       "Data Source=ASA 9.0 Sample"
     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。
- Adaptive Server Anywhere の OLE DB プロバイダを使用して、サンプル・データベースへの接続を確立します。
- Command オブジェクトを使用して簡単な文を実行し、データベース・サーバ・ウィンドウにメッセージを表示します。
- 接続を閉じます。

ASAProv 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 **ASAProv**

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 dboledb9.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、『ASA データベース管理ガイド』> 「OLE DB を使用したデータベースへの接続」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、データベース・サーバ・ウィンドウへのメッセージの表示、および切断を行います。

```
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 = "ASAProv"
     myConn.ConnectionString = _
       "Data Source=ASA 9.0 Sample"
     myConn.Open
```

```
        'Execute a command
        myCommand.CommandText = _
        "update customer set fname='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」411 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」410 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、フォームにコマンド・ボタン **cmdQuery** を配置し、その **Click** イベントに次のルーチンをペーストして試用できます。プログラムを実行し、ボタンをクリックして接続、データベース・サーバ・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、および切断を行います。

```
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 = "ASAProv"
     myConn.ConnectionString = _
       "Data Source=ASA 9.0 Sample"
     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 customer"
     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("company_name"), 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、Customer 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 company_name 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」409 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、Adaptive Server 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

Adaptive Server 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」30 ページを参照してください。ADO には、カーソル・タイプに対する固有の命名規則があります。

使用できるカーソル・タイプと、対応するカーソル・タイプの定数と、それらと同等の Adaptive Server Anywhere のタイプは、次のとおりです。

ADO カーソル・タイプ	ADO 定数	Adaptive Server Anywhere のタイプ
動的カーソル	adOpenDynamic	動的スクロール・カーソル
キーセット・カーソル	adOpenKeyset	スクロール・カーソル
静的カーソル	adOpenStatic	insensitive カーソル
前方向カーソル	adOpenForwardOnly	非スクロール・カーソル

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」29 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

Adaptive Server 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

データベースはレコード・セット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 = "ASAProv"
     myConn.ConnectionString = _
         "Data Source=ASA 9.0 Sample"
     myConn.Open
     myConn.BeginTrans
     SQLString = "Select * from customer"
     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

レコード・セットで 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	ASA 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

ADO 独立性レベル	定数	ASA レベル
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

独立性レベルの詳細については、『ASA SQL ユーザーズ・ガイド』>「独立性レベルと一貫性」を参照してください。

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 Adaptive Server 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。Adaptive Server 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。	CE ではサポートされていません。
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。	CE ではサポートされていません。

インタフェース	内容	制限事項
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。	CE ではサポートされていません。
ICommandPrepare	コマンドを準備する。	CE ではサポートされていません。
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。
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。

インタフェース	内容	制限事項
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	非同期処理。	サポートされていません。
IDBAsyncStatus	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。	
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。

インタフェース	内容	制限事項
IDBInfo	このプロバイダにとってユニークなキーワードについての情報を検索する (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。	CE ではサポートされていません。
IDBInitialize	データ・ソース・オブジェクトと列挙子を初期化する。	CE ではサポートされていません。
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。	CE ではサポートされていません。
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。	CE ではサポートされていません。
IErrorInfo IErrorLookup IErrorRecords	ActiveX エラー・オブジェクト・サポート。	CE ではサポートされていません。
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。

インタフェース	内容	制限事項
IOpenRowset	名前でデータベース・テーブルにアクセスする非 SQL 的な方法。	サポートされています。 名前でテーブルを開くのはサポートされていますが、GUID で開くのはサポートされていません。
IParentRowset	区分化／階層ローセットにアクセスする。	サポートされていません。
IRowset	ローセットにアクセスする。	サポートされています。
IRowsetChange	ローセット・データへの変更を許し、変更をデータ・ストアに反映させる。 blob に対する InsertRow/SetData はまだ実装されていない。	CE ではサポートされていません。
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。
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。	CE ではサポートされていません。

インタフェース	内容	制限事項
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。	CE ではサポートされていません。
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。

インタフェース	内容	制限事項
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。	CE ではサポートされていません。
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。	CE ではサポートされていません。
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。	CE ではサポートされていません。
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。	CE ではサポートされていません。
IViewChapter	既存のローセットでビューを使用する。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。

インタフェース	内容	制限事項
IViewFilter	ローセットの内容を、一連の条件と一致するローに制限する。	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 10 章

Adaptive Server Anywhere .NET データ・プロバイダの概要

この章の内容

この章では、Adaptive Server Anywhere .NET データ・プロバイダについて説明します。

Adaptive Server Anywhere .NET データ・プロバイダの機能

Windows NT/2000/XP で Visual Studio .NET を使用している場合、Adaptive Server Anywhere にアクセスするために次のデータ・プロバイダがサポートされています。

- **iAnywhere.Data.AsaClient** このマニュアルに記載されている Adaptive Server Anywhere .NET データ・プロバイダを使用します。
- **System.Data.Oledb** OLE DB データ・ソースを対象とした汎用データ・プロバイダです。これは、Microsoft .NET Framework の一部です。System.Data.Oledb を Adaptive Server Anywhere OLE DB ドライバとともに使用して、Adaptive Server Anywhere データベースにアクセスできます。
- **System.Data.Odbc** ODBC データ・ソースを対象とした汎用データ・プロバイダです。これは、Microsoft .NET Framework の一部です。System.Data.Odbc を Adaptive Server Anywhere ODBC ドライバとともに使用して、Adaptive Server Anywhere データベースにアクセスできます。

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

Adaptive Server 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。

- Adaptive Server Anywhere .NET データ・プロバイダは、OLE DB プロバイダより速く動作します。
- .NET 環境では、Adaptive Server Anywhere .NET データ・プロバイダは、Adaptive Server Anywhere 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 Adaptive Server Anywhere 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。

サンプル・プロジェクトの実行

Adaptive Server Anywhere .NET データ・プロバイダには、次の 3 つのサンプル・プロジェクトが用意されています。

- **SimpleCE**　[接続] ボタンをクリックしたときに従業員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 Compact Framework Windows CE サンプル。
- **SimpleWin32**　[接続] ボタンをクリックしたときに従業員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 サンプル。
- **TableViewer**　SQL 文を入力して実行するための Windows プログラム。

win32 および Table Viewer サンプルについて説明したチュートリアルについては、「Adaptive Server Anywhere .NET データ・プロバイダのサンプル・アプリケーションの使用」425 ページを参照してください。

注意

SQL Anywhere のインストール・ディレクトリがデフォルト (*C:¥Program Files¥Sybase¥SQL Anywhere 9*) 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 に関するエラーを受け取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.AsaClient.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」438 ページを参照してください。

第 11 章

Adaptive Server Anywhere .NET データ・プロバイダのサンプル・アプリケーションの使用

この章の内容

この章では、Adaptive Server Anywhere .NET データ・プロバイダに用意されている Simple および 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¥Sybase¥SQL Anywhere 9*) 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 に関するエラーを受け取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.AsaClient.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」438 ページを参照してください。

チュートリアル：Simple コード・サンプルの使用

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

アプリケーションの詳細については、*Samples¥ASA¥ADO.NET¥SimpleWin32¥Simple.csproj* の SQL Anywhere のインストール・ディレクトリ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。

- データベースへの接続
- AsaCommand オブジェクトを使用したクエリの実行
- AsaDataReader オブジェクトの使用
- 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」428 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] ー [開く] ー [プロジェクト] を選択します。
3. SQL Anywhere のインストール・ディレクトリの *Samples¥ASA¥ADO.NET¥SimpleWin32* を参照し、*Simple.csproj* プロジェクトを開きます。
4. プロジェクトで Adaptive Server 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.AsaClient 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」438 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 指令もソース・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 指令を参照するには、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* を選択します。
  - [表示] ー [コード] を選択します。
- 上部セクションの using 指令に次の行が表示されます。

  ```
  using iAnywhere.Data.AsaClient;
  ```

  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・コードに別の行を追加する必要があります。

  詳細については、「ソース・コードのデータ・プロバイダ・クラスを参照する」439 ページを参照してください。

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

[AsaSample] ダイアログが表示されます。

- [AsaSample] 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、Adaptive Server 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 Adaptive Server Anywhere .NET データ・プロバイダのいくつかの主な機能について説明します。Simple コード・サンプルは、SQL Anywhere インストール・ディレクトリに格納されている Adaptive Server Anywhere サンプル・データベース、*asademo.db* を使用します。

データベース内のテーブルとこれら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 Studio の紹介』>「サンプル・データベースについて」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。コード全体を参照するには、*Samples¥ASA¥ADO.NET¥SimpleWin32¥Simple.csproj* にあるサンプル・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 ListBox 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、接続オブジェクト (新しい AsaConnection) を宣言して初期化します。

```
private void btnConnect_Click(object sender,
    System.EventArgs e)
    AsaConnection conn = new AsaConnection(
     "Data Source=ASA 9.0 Sample;UID=DBA;PWD=SQL" );
```

AsaConnection オブジェクトは、接続文字列を使用してサンプル・データベースに接続します。

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

AsaConnection オブジェクトの詳細については、「AsaConnection クラス」498 ページを参照してください。

クエリの実行　次のコードは、Command オブジェクト (AsaCommand) を使用し SQL 文 (`SELECT emp_lname FROM employee`) を定義して実行します。次に、このコードは DataReader オブジェクト (AsaDataReader) を返します。

```
AsaCommand cmd = new AsaCommand(
      "select emp_lname from employee", conn );
 AsaDataReader reader = cmd.ExecuteReader();
```

Command オブジェクトの詳細については、「AsaCommand クラス」484 ページを参照してください。

結果の表示　次のコードは、AsaDataReader 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 ListBox 制御に追加します。DataReader は、`GetString( 0 )` を使用してローの最初の値を取得します。

Read メソッドが呼び出されるたびに、DataReader 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 ListBox に追加されます。

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

AsaDataReader オブジェクトの詳細については、「AsaDataReader クラス」518 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、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 Adaptive Server 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( AsaException ex ) {
     MessageBox.Show( ex.Errors[0].Message );
 }
```

AsaException オブジェクトの詳細については、「AsaException クラス」543 ページを参照してください。

チュートリアル：Table Viewer コード・サンプルの使用

このチュートリアルは、Adaptive Server 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。

アプリケーションの詳細については、*Samples¥ASA¥ado.net¥TableViewer¥TableViewer.csproj* の SQL Anywhere のインストール・ディレクトリ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。

- データベースへの接続
- AsaDataAdapter オブジェクトの処理
- 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」433 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] ー [開く] ー [プロジェクト] を選択します。
3. SQL Anywhere のインストール・ディレクトリの *SamplesASA¥ado.net¥TableViewer* を参照し、*TableViewer.csproj* プロジェクトを開きます。
4. プロジェクトで Adaptive Server 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 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.AsaClient 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」438 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 指令もソース・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 指令を参照するには、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* を選択します。
  - [表示] ー [コード] を選択します。
- 上部セクションの using 指令に次の行が表示されます。

  ```
  using iAnywhere.Data.AsaClient;
  ```

  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。

  詳細については、「ソース・コードのデータ・プロバイダ・クラスを参照する」439 ページを参照してください。

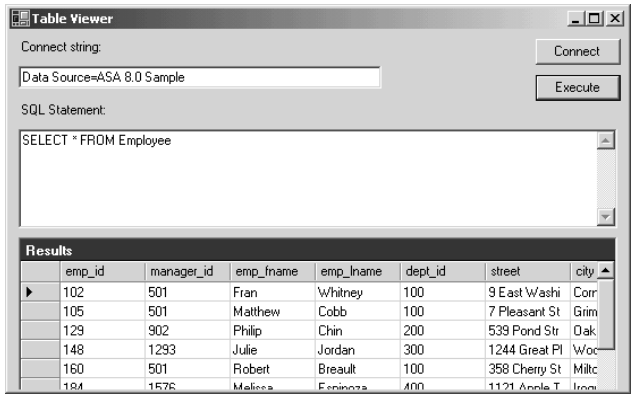
6 [デバッグ] ー [デバッグなしで開始] を選択し、Table Viewer プロジェクトを実行します。

[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 Adaptive Server 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・データベースの従業員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。



最初にデータベースに接続しないでクエリを実行しようとすると、データベースに接続するよう求めるメッセージが表示されます。

- 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 Adaptive Server Anywhere .NET データ・プロバイダのいくつかの主な機能について説明します。Table Viewer プロジェクトは、SQL Anywhere インストール・ディレクトリに格納されている Adaptive Server Anywhere サンプル・データベース *asademo.db* を使用します。

データベース内のテーブルとこれら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 Studio の紹介』>「サンプル・データベースについて」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。コード全体を参照するには、*Samples¥ASA¥ado.net¥Tableviewer¥TableViewer.csproj* にあるサンプル・プロジェクトを開きます。

制御の宣言　次のコードは、Connection String というラベルが付いた TextBox、btnConnect という名前のボタン、txtSQLStatement というラベルが付いた TextBox、btnExecute という名前のボタン、dgResults というラベルが付いた 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;
```

グローバル変数の宣言　AsaConnection 関数を使用して、グローバル変数を宣言します。この接続は、データベースへの最初の接続時に使用されるほか、[実行] をクリックしてデータベースから結果セットを取り出すときにも使用します。

```
private AsaConnection    _conn;
```

AsaConnection 関数の詳細については、「AsaConnection コンストラクタ」498 ページを参照してください。

データベースへの接続　次のコードは、デフォルトで [接続文字列] フィールドに表示される接続文字列のデフォルト値を指定します。

```
this.txtConnectString.Text =
   "Data Source=ASA 9.0 Sample";
```

Connection オブジェクトは、後で接続文字列 (`"Data Source=ASA 9.0 Sample"`) を使用してサンプル・データベースに接続します。

```
_conn = new AsaConnection( txtConnectString.Text );
 _conn.Open();
```

Connection オブジェクトの詳細については、「AsaConnection クラス」498 ページを参照してください。

クエリの定義　次のコードは、[SQL Statement] フィールドに表示されるデフォルト・クエリを定義します。

```
this.txtSQLStatement.Text = "SELECT * FROM employee";
```

結果の表示　結果がフェッチされる前に、アプリケーションは Connection オブジェクトが初期化されているかどうかを確認します。初期化されている場合、接続状態がオープンであることを確認します。

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

データベースに接続されると、次のコードは、新しい DataSet を作成し、DataAdapter オブジェクト (AsaDataAdapter) を使用して SQL 文 (`SELECT * FROM employee`) を実行し、DataSet を設定します。最後の 2 行は、DataSet を画面上のグリッドにバインドします。

```
DataSet ds =new DataSet ();
 AsaDataAdapter da=new AsaDataAdapter(
      txtSQLStatement.Text, _conn);
 da.Fill(ds, "Results")
 dgResults.DataSource = ds.Tables["Results"];
```

グローバル変数を使用して接続が宣言されるため、先に開かれた接続を再使用して SQL 文が実行されます。

DataAdapter オブジェクトの詳細については、「AsaDataAdapter クラス」506 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 AsaConnection( txtConnectString.Text );
      _conn.Open();
   } catch( AsaException ex ) {
     MessageBox.Show( ex.Errors[0].Source + " : "
      + ex.Errors[0].Message + " (" +
      ex.Errors[0].NativeError.ToString() + ")",
          "Failed to connect" );
 }
```

第 12 章

.NET データ・プロバイダを使用したアプリケーションの開発

この章の内容

この章では、Adaptive Server Anywhere .NET データ・プロバイダを使用してアプリケーションを開発して配備する方法について説明します。

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

Adaptive Server Anywhere .NET データ・プロバイダをインストールしたら、Visual Studio .NET プロジェクトを使用するために次の 2 つの変更を行ってください。

- Adaptive Server Anywhere .NET データ・プロバイダ DLL に参照を追加する。
- Adaptive Server Anywhere .NET データ・プロバイダ・クラスを参照する行をソース・コードに追加する。

Adaptive Server 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、「Adaptive Server Anywhere .NET データ・プロバイダの配置」480 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、Adaptive Server 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 に Adaptive Server 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] フォルダを右クリックし、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.AsaClient.dll* を見つけます (デフォルト・ロケーションは、*¥Program Files¥Sybase¥SQL Anywhere 9¥win32* です)。DLL を選択して [開く] をクリックします。

Windows と Windows CE にはそれぞれ異なるバージョンの DLL があるので注意してください。

インストールされている DLL の詳細リストについては、「Adaptive Server Anywhere .NET データ・プロバイダに必要なファイル」480 ページを参照してください。

4　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.AsaClient.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・クラスを参照する

Adaptive Server Anywhere .NET データ・プロバイダを使用するには、データ・プロバイダを参照するための行もソース・コードに追加してください。C# には Visual Basic .NET とは異なる行を追加してください。

❖ コードのデータ・プロバイダ・クラスを参照するには、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　C# を使用している場合は、プロジェクトの先頭にある `using` 指令のリストに次の行を追加します。

```
using iAnywhere.Data.AsaClient;
```

3　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.AsaClient
```

この行は厳密には必要ありません。しかし、この行を追加すると、Adaptive Server Anywhere クラスの省略形を使用できるようになります。この行がなくても、

```
iAnywhere.Data.AsaClient.AsaConnection
  conn = new
iAnywhere.Data.AsaClient.AsaConnection()
```

を次の行の代わりに

```
AsaConnection    conn = new AsaConnection()
```

コード内で使用できます。

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する必要があります。この項では、Adaptive Server 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、「AsaConnection クラス」498 ページと「ConnectionString プロパティ」500 ページを参照してください。

❖ Adaptive Server Anywhere データベースに接続するには、次の手順に従います。

1　AsaConnection 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 AsaConnection オブジェクトを作成します。

```
AsaConnection conn = new AsaConnection(
```

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、Adaptive Server 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 AsaConnection _conn;
```

詳細については、*Samples¥ASA¥ado.net¥Table-Viewer¥TableViewer.csproj* と 「Table Viewer サンプル・プロジェクトの知識」433 ページ のサンプル・コード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=ASA 9.0 Sample;UID=DBA;PWD=SQL" );
```

接続パラメータの詳細リストについては、『ASA データベース管理ガイド』> 「接続パラメータ」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 AsaConnection(
txtConnectString.Text );
     _conn.Open();
   } catch( AsaException ex ) {
     MessageBox.Show( ex.Errors[0].Source + " : "
      + ex.Errors[0].Message + " (" +
      ex.Errors[0].NativeError.ToString() + ")",
          "Failed to connect" );
```

また、AsaConnection 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。

```
AsaConnection    _conn;
     _conn = new AsaConnection();
     _conn.ConnectionString =
      "Data Source=ASA 9.0 Sample;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、Adaptive Server 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.AsaClient.AsaConnection()
     myConn.ConnectionString = _
       "Data Source=ASA 9.0 Sample;UID=DBA;PWD=SQL"
     myConn.Open()
     myConn.Close()
 End Sub
```

接続プーリング

Adaptive Server Anywhere .NET 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、プール内の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。また、プールの最大サイズと最小サイズも指定できます。次に例を示します。

```
"Data Source=ASA 9.0
Sample;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、「ConnectionString プロパティ」500 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。

AsaConnection 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」504 ページを参照してください。

データのアクセスと操作

Adaptive Server Anywhere .NET データ・プロバイダでは、AsaCommand オブジェクトを使用する方法と AsaDataAdapter 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。

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

  AsaCommand 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。AsaCommand 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。

  AsaCommand オブジェクトでは、AsaDataReader 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。AsaDataReader は 1 回に 1 つのローのみを返しますが、Adaptive Server 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。

  AsaCommand 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。AsaTransaction 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、「AsaCommand クラス」484 ページと「AsaDataReader クラス」518 ページを参照してください。

- **AsaDataAdapter オブジェクト**　AsaDataAdapter 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、AsaDataAdapter 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。AsaDataAdapter 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、「AsaDataAdapter を使用するときの競合の解消」457 ページを参照してください。

AsaDataAdapter オブジェクトの詳細については、「AsaDataAdapter クラス」506 ページを参照してください。

AsaDataAdapter オブジェクトとは異なり、AsaCommand オブジェクト内で AsaDataReader 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。

AsaCommand 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、AsaDataReader 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。

AsaCommand オブジェクトを使用したデータの取得

AsaCommand オブジェクトを使用して、Adaptive Server 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するために使用します。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 方向のみで簡単にループできます。

  詳細については、「ExecuteReader メソッド」488 ページを参照してください。

- **ExecuteScalar** 単一値を返すコマンドを発行するために使用します。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」489 ページを参照してください。

AsaCommand オブジェクトを使用する場合、AsaDataReader 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」428 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。

```
AsaConnection conn = new AsaConnection(
     "Data Source=ASA 9.0 Sample;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。

```
AsaCommand cmd = new AsaCommand(
      "select emp_lname from employee", 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」472 ページと 「AsaParameter クラス」548 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。

```
AsaDataReader 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 AsaConnection オブジェクトを宣言して初期化します。

```
AsaConnection conn = new AsaConnection(
     "Data Source=ASA 9.0 Sample" );
```

2 接続を開きます。

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

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

```
AsaCommand cmd = new AsaCommand(
     "select count(*) from employee 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」472 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　AsaConnection オブジェクトを閉じます。

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

AsaDataReader 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、「AsaDataReader クラス」518 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、Adaptive Server 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 .AsaConnection()
 Dim myCmd As _
   New .AsaCommand _
   ("select emp_lname from employee", myConn)
 Dim myReader As AsaDataReader
 Dim counter As Integer
 myConn.ConnectionString = _
   "Data Source=ASA 9.0 Sample;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()
```

AsaCommand オブジェクトを使用したローの挿入、更新、削除

AsaCommand 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。

詳細については、「ExecuteNonQuery メソッド」488 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。AsaCommand オブジェクトを使用するには、データベースに接続されている必要があります。

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」464 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、AsaCommand オブジェクトを AsaTransaction オブジェクトの一部として使用します。AsaTransaction オブジェクトを使用しないでデータを修正すると、.NET 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。

詳細については、「Transaction 処理」475 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 AsaConnection オブジェクトを宣言して初期化します。

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

2 接続を開きます。

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

3 INSERT 文を定義して実行する AsaCommand 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。

```
AsaCommand      insertCmd = new AsaCommand(
     "INSERT INTO department( dept_id, dept_name )
     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」472 ページと 「AsaParameter クラス」548 ページを参照してください。

4　AsaCommand オブジェクトのパラメータを設定します。

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

```
AsaParameter parm = new AsaParameter();
 parm.AsaDbType = AsaDbType.Integer;
 insertCmd.Parameters.Add( parm );
 parm = new AsaParameter();
 parm.AsaDbType = AsaDbType.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";
 int 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。

```
AsaCommand      selectCmd = new AsaCommand(
     "SELECT * FROM department", conn );
 AsaDataReader   dr = selectCmd.ExecuteReader();
 dataGrid.DataSource = dr;
```

7　AsaDataReader オブジェクトと AsaConnection 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　AsaConnection オブジェクトを宣言して初期化します。

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

2 接続を開きます。

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

3 UPDATE 文を定義して実行する AsaCommand 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。

```
AsaCommand      updateCmd = new AsaCommand(
     "UPDATE department SET dept_name =
'Engineering'
     WHERE dept_id=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」472 ページと 「AsaParameter クラス」548 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。

```
AsaCommand      selectCmd = new AsaCommand(
     "SELECT * FROM department", conn );
 AsaDataReader   dr = selectCmd.ExecuteReader();
 dataGrid.DataSource = dr;
```

6 AsaDataReader オブジェクトと AsaConnection 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　AsaConnection オブジェクトを宣言して初期化します。

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

2　接続を開きます。

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

3　DELETE 文を定義して実行する AsaCommand 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。

```
AsaCommand      deleteCmd = new AsaCommand(
   "DELETE FROM department WHERE ( dept_id > 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」472 ページと 「AsaParameter クラス」548 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　AsaConnection 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。

AsaDataReader 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」529 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。

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

2 接続を開きます。

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

3 使用する SELECT 文によって AsaCommand 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。

```
AsaCommand cmd = new AsaCommand(
     "SELECT * FROM employee", conn );
```

4 AsaDataReader 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。

```
AsaDataReader   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 AsaDataReader オブジェクトと AsaConnection 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;
```

AsaDataAdapter 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、AsaDataAdapter 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。

AsaDataAdapter オブジェクトを使用したデータの取得

AsaDataAdapter 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。

AsaDataAdapter 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。AsaDataAdapter 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。
AsaDataAdapter 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。

AsaDataAdapter 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、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。

AsaDataAdapter の詳細については、「AsaDataAdapter クラス」506 ページを参照してください。

AsaDataAdapter の例

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

❖ **AsaDataAdapter 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 AsaDataAdapter オブジェクトを作成します。

```
AsaDataAdapter da=new AsaDataAdapter(
      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"]
```

AsaDataAdapter オブジェクトを使用したローの挿入、更新、削除

AsaDataAdapter 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。

AsaDataAdapter 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、AsaDataAdapter 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。

AsaDataAdapter を使用するときの競合の解消

AsaDataAdapter 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」464 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。AsaDataAdapter 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、AsaCommand、AsaDataReader、AsaTransaction 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、AsaTransaction オブジェクトを使用することをおすすめします。これは、AsaTransaction 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、「AsaCommand オブジェクトを使用したローの挿入、更新、削除」449 ページを参照してください。

競合の解消プロセスを簡単にするために、挿入、更新、または削除文をストアド・プロシージャ呼び出しとして設計できます。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。

❖ AsaDataAdapter 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　AsaConnection オブジェクトを宣言して初期化します。

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

2　接続を開きます。

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

3　新しい AsaDataAdapter を作成します。

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

4　必要な AsaCommand 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 AsaCommand(
     "SELECT * FROM department", conn );
 adapter.InsertCommand = new AsaCommand(
     "INSERT INTO department( dept_id, dept_name )
     VALUES( ?, ? )", conn );
 adapter.InsertCommand.UpdatedRowSource =
     UpdateRowSource.None;
 AsaParameter parm = new AsaParameter();
 parm.AsaDbType = AsaDbType.Integer;
 parm.SourceColumn = "dept_id";
 parm.SourceVersion = DataRowVersion.Current;
 adapter.InsertCommand.Parameters.Add(
     parm );
 parm = new AsaParameter();
 parm.AsaDbType = AsaDbType.Char;
 parm.SourceColumn = "dept_name";
 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( "department" );
 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();
```

❖ AsaDataAdapter 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 AsaConnection オブジェクトを宣言して初期化します。

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

2 接続を開きます。

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

3 新しい AsaDataAdapter を作成します。

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

4　AsaCommand 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 AsaCommand(
     "SELECT * FROM department WHERE dept_id > 500",
     conn );
 adapter.UpdateCommand = new AsaCommand(
     "UPDATE department SET dept_name = ?
     WHERE dept_id = ?", conn );
 adapter.UpdateCommand.UpdatedRowSource =
     UpdateRowSource.None;
 AsaParameter parm = new AsaParameter();
 parm.AsaDbType = AsaDbType.Char;
 parm.SourceColumn = "dept_name";
 parm.SourceVersion = DataRowVersion.Current;
 adapter.UpdateCommand.Parameters.Add( parm );
 parm = new AsaParameter();
 parm.AsaDbType = AsaDbType.Integer;
 parm.SourceColumn = "dept_id";
 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( "department" );
 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 department";
 rowCount = adapter.Fill( dataTable );
 dataGrid.DataSource = dataTable;
```

8 接続を閉じます。

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

❖ AsaDataAdapter 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 AsaConnection オブジェクトを宣言して初期化します。

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

2 接続を開きます。

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

3 AsaDataAdapter オブジェクトを作成します。

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

4 必要な AsaCommand 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 AsaCommand(
     "SELECT * FROM department WHERE dept_id > 500",
     conn );
 adapter.DeleteCommand = new AsaCommand(
     "DELETE FROM department WHERE dept_id = ?",
     conn );
 adapter.DeleteCommand.UpdatedRowSource =
     UpdateRowSource.None;
```

```
AsaParameter parm = new AsaParameter();
parm.AsaDbType = AsaDbType.Integer;
parm.SourceColumn = "dept_id";
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( "department" );
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();
```

AsaDataAdapter スキーマ情報の取得

AsaDataAdapter 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 メソッドの詳細については、「FillSchema メソッド」510 ページを参照してください。

❖ 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 AsaConnection オブジェクトを宣言して初期化します。

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

2 接続を開きます。

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

3 使用する SELECT 文によって AsaDataAdapter 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。

```
AsaDataAdapter  adapter = new AsaDataAdapter(
     "SELECT * FROM employee", 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 AsaConnection 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。

AsaDataAdapter 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。AsaCommand 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
```

❖ AsaCommand 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。

```
AsaConnection   conn = OpenConnection();
```

2　新しいローを DataTable に挿入する AsaCommand 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。

```
AsaCommand      cmd = conn.CreateCommand();
 cmd.CommandText = "sp_adodotnet_primarykey";
 cmd.CommandType = CommandType.StoredProcedure;
 AsaParameter parmId = new AsaParameter();
 parmId.AsaDbType = AsaDbType.Integer;
 parmId.Direction = ParameterDirection.Output;
 cmd.Parameters.Add( parmId );
 AsaParameter parmName = new AsaParameter();
 parmName.AsaDbType = AsaDbType.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;
AsaDataReader dr = cmd.ExecuteReader();
dataGrid.DataSource = dr;
```

4 接続を閉じます。

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

❖ **AsaDataAdapter 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 AsaDataAdapter を作成します。

```
DataSet         dataSet = new DataSet();
AsaConnection   conn = OpenConnection();
AsaDataAdapter  adapter = new AsaDataAdapter();
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、AsaDataAdapter.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 AsaCommand( "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 AsaCommand オブジェクトを作成します。

```
adapter.InsertCommand = new AsaCommand(
     "sp_adodotnet_primarykey", conn );
 adapter.InsertCommand.CommandType =
     CommandType.StoredProcedure;
 adapter.InsertCommand.UpdatedRowSource =
     UpdateRowSource.OutputParameters;
 AsaParameter parmId = new AsaParameter();
 parmId.AsaDbType = AsaDbType.Integer;
 parmId.Direction = ParameterDirection.Output;
 parmId.SourceColumn = "id";
 parmId.SourceVersion = DataRowVersion.Current;
 adapter.InsertCommand.Parameters.Add( parmId );
 AsaParameter parmName = new AsaParameter();
 parmName.AsaDbType = AsaDbType.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、AsaDataAdapter 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」521 ページと 「GetChars メソッド」522 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。

```
AsaCommand cmd = new AsaCommand(
     "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。

```
AsaDataReader 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。Adaptive Server 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」 531 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。

```
AsaConnection conn = new AsaConnection(
     "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。

```
AsaCommand cmd = new AsaCommand(
     "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。

```
AsaDataReader 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、AsaParameter 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、「AsaParameter クラス」548 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. AsaConnection オブジェクトを宣言して初期化します。

   ```
   AsaConnection conn = new AsaConnection(
        "Data Source=ASA 9.0 Sample" );
   ```

2. 接続を開きます。

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

3. SQL 文を定義して実行する AsaCommand 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。

   ```
   AsaCommand cmd = new AsaCommand( "sp_product_info",
        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。

```
AsaCommand cmd = new AsaCommand(
     "call sp_product_info(?)", 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 AsaParameter 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 AsaParameter オブジェクトを作成してください。

```
AsaParameter param = cmd.CreateParameter();
 param.AsaDbType = AsaDbType.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、「AsaParameter クラス」548 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。

```
AsaDataReader 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　AsaDataReader オブジェクトと AsaConnection 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。

```
AsaCommand cmd = new AsaCommand(
     "call sp_product_info( 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、「AsaCommand オブジェクトを使用したデータの取得」446 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、「AsaCommand オブジェクトを使用したローの挿入、更新、削除」449 ページを参照してください。

Transaction 処理

Adaptive Server Anywhere .NET プロバイダでは、AsaTransaction 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 AsaTransaction 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、Adaptive Server Anywhere .NET 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。

AsaTransaction オブジェクトの詳細については、「AsaTransaction クラス」571 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。

独立性レベルの詳細については、『ASA SQL ユーザーズ・ガイド』>「独立性レベルと一貫性」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、『ASA SQL ユーザーズ・ガイド』>「クエリ時のロック」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 AsaTransaction 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。

❖ AsaTransaction 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 AsaConnection オブジェクトを宣言して初期化します。

```
AsaConnection conn = new AsaConnection(
     "Data Source=ASA 9.0 Sample" );
```

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 product set unit_price =
     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 AsaTransaction 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。

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

5 変更内容をロールバックします。

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

AsaTransaction 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　AsaConnection オブジェクトを閉じます。

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

エラー処理と Adaptive Server 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。

Adaptive Server 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 AsaException オブジェクトをスローします。各 AsaException オブジェクトは AsaError 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、「AsaDataAdapter を使用するときの競合の解消」457 ページを参照してください。

.NET 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 Adaptive Server 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( AsaException 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 AsaConnection( txtConnectString.Text );
      _conn.Open();
   } catch( AsaException 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」428 ページと 「Table Viewer サンプル・プロジェクトの知識」433 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、「AsaException クラス」543 ページと「AsaError クラス」539 ページを参照してください。

Adaptive Server Anywhere .NET データ・プロバイダの配置

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

Adaptive Server Anywhere .NET データ・プロバイダ・システムの稼働条件

Adaptive Server Anywhere .NET データ・プロバイダを使用するには、マシンまたはハンドヘルド・デバイスに以下がインストールされている必要があります。

- .NET Framework か .NET Compact Framework またはその両方
- Visual Studio .NET 1.0、Visual Studio .NET 2003、または C# などの .NET 言語コンパイラ (開発用としてのみ必要)

Adaptive Server Anywhere .NET データ・プロバイダに必要なファイル

Adaptive Server 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。

- *win32¥iAnywhere.Data.AsaClient.dll*
- *win32¥dbdata9.dll*

最初の DLL (*iAnywhere.Data.AsaClient.dll*) は、Visual Studio プロジェクトによって参照されるメイン DLL です。2 番目の DLL (*dbdata9.dll*) にはユーティリティ・コードが含まれます。

Windows CE に必要なファイル

これらのファイルには、SQL Anywhere のインストール・ディレクトリに配置される言語 DLL が必要であるため、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¥Sybase¥SQL Anywhere 9¥win32* です) にインストールしてください。

Windows CE の場合、*iAnywhere.Data.AsaClient.dll* は、Visual Studio プロジェクトによって参照されるメイン DLL です。サポートされている Windows CE プラットフォームごとに個別の *dbdata9.dll* ファイルがあります。Windows CE DLL は、次のとおりです。

- *ce¥iAnywhere.Data.AsaClient.dll*
- *ce¥arm.30¥dbdata9.dll*
- *ce¥emulator.30¥dbdata9.dll*
- *ce¥mips.30¥dbdata9.dll*
- *ce¥x86¥dbdata9.dll*

ユーティリティ DLL (*dbdata9.dll*) は、デバイス上の Windows ディレクトリに配置してください。Visual Studio .NET は、.NET データ・プロバイダ DLL (*iAnywhere.Data.Asaclient.dll*) をプログラムとともにデバイスに配置します。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。

Adaptive Server Anywhere .NET データ・プロバイダ DLL の登録

Adaptive Server Anywhere .NET データ・プロバイダ DLL (*win32¥iAnywhere.Data.AsaClient.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・プロバイダを配置する場合、.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 (*win32¥iAnywhere.Data.AsaClient.dll*) を登録してください。

第 13 章

Adaptive Server Anywhere .NET データ・プロバイダ API リファレンス

この章の内容

この章では、Adaptive Server Anywhere .NET データ・プロバイダの API について説明します。

注意: この章の多くのプロパティとメソッドは、OLE DB .NET データ・プロバイダとよく似ています。詳細と例については、Microsoft .NET Framework のマニュアルを参照してください。

AsaCommand クラス

説明 Adaptive Server 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。

基本クラス Component

実装されたインタフェース IDbCommand, IDisposable

参照 「AsaCommand オブジェクトを使用したデータの検索と操作」446 ページ

「データのアクセスと操作」445 ページ

AsaCommand コンストラクタ

説明 AsaCommand オブジェクトを初期化します。

構文 1 void **AsaCommand( )**

構文 2 void **AsaCommand(** string *cmdText* **)**

構文 3
void **AsaCommand(**
string *cmdText*,
AsaConnection *connection*
)

構文 4
void **AsaCommand(**
string *cmdText*,
AsaConnection *connection*,
AsaTransaction *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　AsaConnection が実行される AsaTransaction。

Cancel メソッド

説明　AsaCommand オブジェクトの実行をキャンセルします。

構文　void **Cancel( )**

使用法　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。

実装　IDbCommand.Cancel

CommandText プロパティ

説明　SQL 文またはストアド・プロシージャのテキスト。

構文　string **CommandText**

アクセス　読み込み／書き込み

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。

実装　IDbCommand.CommandText

参照　「AsaCommand コンストラクタ」484 ページ

CommandTimeout プロパティ

説明　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)。

構文　int **CommandTimeout**

アクセス　読み込み／書き込み

実装 IDbCommand.CommandTimeout

デフォルト 30 秒

使用法 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。

CommandType プロパティ

説明 AsaCommand によって示されるコマンドのタイプ。

構文 CommandType **CommandType**

アクセス 読み込み／書き込み

実装 IDbCommand.CommandType

使用法 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 AsaParameter 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 CustomerID = ?
```

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

Connection プロパティ

説明　AsaCommand オブジェクトが適用される接続オブジェクト。

構文　AsaConnection **Connection**

アクセス　読み込み／書き込み

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

CreateParameter メソッド

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

構文　AsaParameter **CreateParameter( )**

戻り値　AsaParameter オブジェクトとしての新しいパラメータ。

使用法　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、AsaParameter オブジェクトを提供します。AsaParameter 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　「AsaParameter クラス」548 ページ

DesignTimeVisible プロパティ

説明　カスタマイズされた Windows Form Designer 制御で AsaCommand を参照できるようにするかどうかを指定する値。デフォルトは false です。

構文　bool **DesignTimeVisible**

アクセス　読み込み／書き込み

ExecuteNonQuery メソッド

説明　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　int **ExecuteNonQuery( )**

戻り値　影響を受けたローの数。

実装　IDbCommand.ExecuteNonQuery

使用法　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。

ExecuteReader メソッド

説明　結果セットを返す SQL 文を実行します。

構文 1　AsaDataReader **ExecuteReader( )**

構文 2　AsaDataReader **ExecuteReader(CommandBehavior** *behavior* **)**

パラメータ　**behavior　CloseConnection**、**Default**、**KeyInfo**、**SchemaOnly**、**SequentialAccess**、**SingleResult**、または **SingleRow** の 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　AsaDataReader オブジェクトとしての結果セット。

使用法　この文は、必要に応じて CommandText および Parameters を持つ現在の AsaCommand オブジェクトです。AsaDataReader 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、AsaDataAdapter を使用します。

参照　「AsaDataReader クラス」518 ページ

「AsaDataAdapter クラス」506 ページ

ExecuteScalar メソッド

説明　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　object **ExecuteScalar( )**

戻り値　結果セットの最初のローの最初のカラム、または NULL 参照 (結果セットが空の場合)。

実装　IDbCommand.ExecuteScalar

Parameters プロパティ

説明　現在の文のパラメータのコレクション。CommandText で疑問符を使用してパラメータを示します。

構文　AsaParameterCollection **Parameters**

アクセス　読み込み専用

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。

使用法　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 CustomerID = ?
```

AsaParameter オブジェクトが AsaParameterCollection 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 「AsaParameterCollection クラス」556 ページ

Prepare メソッド

説明 データ・ソース上で AsaCommand を準備またはコンパイルします。

構文 void **Prepare( )**

実装 IDbCommand.Prepare

使用法 準備する文の各パラメータのデータ型を指定してから、Prepare を呼び出します。

Prepare を呼び出してから Execute 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。

準備されているかどうかにかかわらず、出力パラメータにはユーザ指定のデータ型が必要です。

ResetCommandTimeout メソッド

説明 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 void **ResetCommandTimeout( )**

Transaction プロパティ

説明 現在のコマンドをトランザクションに関連付けます。

構文　AsaTransaction **Transaction**

アクセス　読み込み／書き込み

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

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

参照　「AsaTransaction クラス」571 ページ

「Transaction 処理」475 ページ

UpdatedRowSource プロパティ

説明　AsaDataAdapter 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。

構文　UpdateRowSource **UpdatedRowSource**

アクセス　読み込み／書き込み

実装　IDbCommand.UpdatedRowSource

プロパティ値　UpdatedRowSource 値の 1 つ。コマンドが自動的に生成される場合、デフォルトは None です。それ以外の場合、デフォルトは Both です。

AsaCommandBuilder クラス

説明 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。

基本クラス Component

実装されたインタフェース IDisposable

AsaCommandBuilder コンストラクタ

説明 AsaCommandBuilder オブジェクトを初期化します。

構文 1 void **AsaCommandBuilder( )**

構文 2 void **AsaCommandBuilder(** AsaDataAdapter *adapter* **)**

パラメータ **adapter** 調整文を生成する AsaDataAdapter オブジェクト。

DataAdapter プロパティ

説明 文を生成する AsaDataAdapter。

構文 AsaDataAdapter **DataAdapter**

アクセス 読み込み／書き込み

プロパティ値 AsaDataAdapter オブジェクト。

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

DeriveParameters メソッド

説明　指定された AsaCommand オブジェクトの Parameters コレクションに入力します。これは、AsaCommand 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　void **DeriveParameters(** AsaCommand *command* **)**

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

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

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

GetDeleteCommand メソッド

説明　AsaDataAdapter.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 AsaCommand オブジェクト。

構文　AsaCommand **GetDeleteCommand( )**

戻り値　削除の実行に必要な、自動的に生成された AsaCommand オブジェクト。

使用法　GetDeleteCommand メソッドは、実行対象の AsaCommand 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、AsaDataAdapter 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ら、アプリケーションは文を変更する場合、RefreshSchema を明示的に呼び出す必要があります。それ以外の場合、GetDeleteCommand は前の文の情報を使用し続けます。

GetInsertCommand メソッド

説明

AsaDataAdapter.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 AsaCommand オブジェクト。

構文

AsaCommand **GetInsertCommand( )**

戻り値

挿入の実行に必要な、自動的に生成された AsaCommand オブジェクト。

使用法

GetInsertCommand メソッドは、実行対象の AsaCommand 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、AsaDataAdapter 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ら、アプリケーションは文を変更する場合、RefreshSchema を明示的に呼び出す必要があります。それ以外の場合、GetInsertCommand は、正しくない可能性がある前の文の情報を使用し続けます。

GetUpdateCommand メソッド

説明

AsaDataAdapter.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 AsaCommand オブジェクト。

構文　AsaCommand **GetUpdateCommand( )**

戻り値　更新の実行に必要な、自動的に生成された AsaCommand オブジェクト。

使用法　GetUpdateCommand メソッドは、実行対象の AsaCommand 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、AsaDataAdapter 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ら、アプリケーションは文を変更する場合、RefreshSchema を明示的に呼び出す必要があります。それ以外の場合、GetUpdateCommand は、正しくない可能性がある前の文の情報を使用し続けます。

QuotePrefix プロパティ

説明　スペースなどの文字が含まれるデータベース・オブジェクト名を指定するときに使用する 1 つまたは複数の先頭文字。

構文　string **QuotePrefix**

アクセス　読み込み／書き込み

プロパティ値　使用する 1 つまたは複数の先頭文字。これには、角カッコを使用できます。また、Adaptive Server Anywhere QUOTED_IDENTIFIER オプションがオフに設定されている場合は、二重引用符も使用できます。デフォルトは、空の文字列です。

使用法　Adaptive Server Anywhere オブジェクトには、スペース、カンマ、セミコロンなどの文字を使用できます。QuotePrefix および QuoteSuffix プロパティは、オブジェクト名をカプセル化するためのデリミタを指定します。

INSERT、UPDATE、または DELETE オペレーションの後は QuotePrefix または QuoteSuffix プロパティを変更できませんが、DataAdapter の Update メソッドを呼び出した後はこれらの設定を変更できます。

参照

『ASA SQL リファレンス・マニュアル』> 「識別子」

『ASA データベース管理ガイド』> 「QUOTED_IDENTIFIER オプション [互換性]」

QuoteSuffix プロパティ

説明

名前にスペースなどの文字が含まれるデータベース・オブジェクトを指定するときに使用する 1 つまたは複数の末尾文字。

構文

string **QuoteSuffix**

アクセス

読み込み／書き込み

プロパティ値

使用する 1 つまたは複数の末尾文字。これには、角カッコを使用できます。また、Adaptive Server Anywhere QUOTED_IDENTIFIER オプションがオフに設定されている場合は、二重引用符も使用できます。デフォルトは、空の文字列です。

使用法

Adaptive Server Anywhere オブジェクトには、スペース、カンマ、セミコロンなどの文字を使用できます。QuotePrefix および QuoteSuffix プロパティは、オブジェクト名をカプセル化するためのデリミタを指定します。

INSERT、UPDATE、または DELETE オペレーションの後は QuotePrefix または QuoteSuffix プロパティを変更できませんが、DataAdapter の Update メソッドを呼び出した後はこれらの設定を変更できます。

参照

『ASA SQL リファレンス・マニュアル』> 「識別子」

『ASA データベース管理ガイド』> 「QUOTED_IDENTIFIER オプション [互換性]」

RefreshSchema メソッド

説明　INSERT、UPDATE、または DELETE 文の生成に使用されるデータベース・スキーマ情報を再表示します。

構文　void **RefreshSchema( )**

使用法　関連付けられている AsaDataAdapter の SelectCommand 値が変更されるたびに RefreshSchema を呼び出します。

AsaConnection クラス

説明 Adaptive Server Anywhere データベースへの接続を示します。

基本クラス Component

実装されたインタフェース IDbConnection, IDisposable

参照 「データベースへの接続」441 ページ

AsaConnection コンストラクタ

説明 AsaConnection 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 1 void **AsaConnection( )**

構文 2 void **AsaConnection(** string *connectionString* **)**

パラメータ **connectionString** Adaptive Server Anywhere 接続文字列。接続文字列は、キーワード値の組がセミコロンで区切られたリストです。

パラメータのリストについては、『ASA データベース管理ガイド』>「接続パラメータ」を参照してください。

例 次の文は、hr という名前の Adaptive Server 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 AsaConnection オブジェクトを初期化します。この接続は、パスワード money を持つ admin というユーザ ID を使用します。

```
AsaConnection conn = new AsaConnection(
 "uid=admin;pwd=money;eng=hr;dbn=policies" );
 conn.Open();
```

BeginTransaction メソッド

説明　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() または Rollback() で終了します。

構文 1　AsaTransaction **BeginTransaction( )**

構文 2　AsaTransaction **BeginTransaction(** IsolationLevel *isolationLevel* **)**

パラメータ　**isolationLevel**　IsolationLevel 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示すオブジェクト。

使用法　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、AsaCommand.Transaction プロパティを使用します。

例

```
AsaTransaction tx = conn.BeginTransaction(
     IsolationLevel.ReadUncommitted );
```

参照

「Commit メソッド」571 ページ

「Rollback メソッド」572 ページ

「Transaction 処理」475 ページ

『ASA SQL ユーザーズ・ガイド』>「典型的な矛盾のケース」

ChangeDatabase メソッド

説明　オープン AsaConnection の現在のデータベースを変更します。

構文　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。

実装 IDbConnection.ChangeDatabase

Close メソッド

説明 データベース接続を閉じます。

構文 void **Close( )**

実装 IDbConnection.Close

使用法 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。

ConnectionString プロパティ

説明 データベース接続文字列。

構文 string **ConnectionString**

アクセス 読み込み／書き込み

実装 IDbConnection.ConnectionString

使用法 接続プーリングのデフォルト値は、true (pooling=true) です。

ConnectionString は、Adaptive Server Anywhere 接続文字列フォーマットを次の例外にできるだけ近づけるように設計されています。

- 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、ユーザ設定の ConnectionString からセキュリティ情報を除いたものと同じです。ユーザが Persist Security Info 値を true に設定しないかぎり、Adaptive Server Anywhere .NET データ・プロバイダは、接続文字列でパスワードを保持したり返したりしません。

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。AsaConnection 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によって区切られることがあります。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。キーワード値の組は、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・ボックスからユーザ 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。

例

次の文は、**ASA 9.0 Sample** 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。

```
AsaConnection conn = new AsaConnection();
 conn.ConnectionString = "dsn=ASA 9.0 Sample";
 conn.Open();
```

ConnectionTimeout プロパティ

説明 接続試行がエラーでタイムアウトするまでの秒数。

構文 int **ConnectionTimeout**

アクセス 読み込み専用

デフォルト 15 秒

実装 IDbConnection.ConnectionTimeout

例 次の文は、ConnectionTimeout の値を表示します。

```
MessageBox.Show( conn.ConnectionTimeout.ToString() );
```

CreateCommand メソッド

説明 AsaCommand オブジェクトを初期化します。AsaCommand のプロパティを使用して、その動作を制御できます。

構文 AsaCommand **CreateCommand( )**

戻り値 AsaCommand オブジェクト。

使用法 コマンド・オブジェクトは AsaConnection に関連付けられます。

Database プロパティ

説明 接続を開いた後に使用する現在のデータベースの名前。

構文 string **Database**

アクセス 読み込み専用

実装 IDbConnection.Database

使用法　AsaConnection は、DatabaseName、dbn、DataSourceName、DataSource、dsn、DatabaseFile、dbf の順に接続文字列を参照します。

DataSource プロパティ

説明　接続先の動作中のデータベース・サーバの名前。

構文　string **DataSource**

アクセス　読み込み専用

使用法　AsaConnection は、Enginename、Servername、Eng の順に接続文字列を参照します。

InfoMessage イベント

説明　プロバイダが警告または情報メッセージを送信するときに発生します。

構文　event AsaInfoMessageEventHandler **InfoMessage**

使用法　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 AsaInfoMessageEventArgs の引数を受け取ります。次の AsaInfoMessageEventArgs プロパティは、このイベントに固有の情報 (ErrorCode、Errors、Message、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。

Open メソッド

説明　以前に指定されている接続文字列を使用してデータベースへの接続を開きます。

構文　void **Open( )**

実装　IDbConnection.Open

使用法

AsaConnection は、接続プールからオープン接続 (使用可能な場合) を取得します。使用不可能な場合、データ・ソースに対して新しい接続を確立します。

AsaConnection が範囲外になってしまうと、接続は閉じられません。このため、Close または Dispose を呼び出して接続を明示的に閉じてください。

ServerVersion プロパティ

説明

Adaptive Server Anywhere データベース・サーバのソフトウェア・バージョン。

構文

string **ServerVersion**

アクセス

読み込み専用

使用法

バージョンのフォームは *##.##.####* です。最初の 2 桁はメジャー・バージョン、次の 2 桁はマイナー・バージョン、最後の 4 桁はリリース・バージョンを示します。付加されている文字列のフォームは *major.minor.build* です。*major* と *minor* は 2 桁、*build* は 4 桁です。

State プロパティ

説明

接続の現在の状態。

構文

ConnectionState **State**

アクセス

読み込み専用

デフォルト

デフォルト値は Closed です。

実装

IDbConnection.State

参照

「接続状態の確認」444 ページ

StateChange イベント

説明　接続状態が変更されると発生します。

構文　event StateChangeEventHandler **StateChange**

使用法　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。

AsaDataAdapter クラス

説明

DataSet に設定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。

基本クラス

Component

実装されたインタフェース

IDbDataAdapter, IDisposable

使用法

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。AsaDataAdapter 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。

参照

「AsaDataAdapter オブジェクトを使用したデータのアクセスと操作」455 ページ

「データのアクセスと操作」445 ページ

AsaDataAdapter コンストラクタ

説明

AsaDataAdapter オブジェクトを初期化します。

構文 1

void **AsaDataAdapter( )**

構文 2

void **AsaDataAdapter(** AsaCommand *selectCommand* **)**

構文 3

void **AsaDataAdapter(**
string *selectCommandText*,
AsaConnection *selectConnection*
)

構文 4

void **AsaDataAdapter(**
string *selectCommandText*,
string *selectConnectionString*
)

パラメータ

selectCommand　placement 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 Fill の実行時に使用される AsaCommand オブジェクト。

selectCommandText　AsaDataAdapter の SelectCommand プロパティによって使用される SELECT 文またはストアド・プロシージャ。

selectConnection　データベースへの接続を定義する AsaConnection オブジェクト。

selectConnectionString　Adaptive Server Anywhere データベースの接続文字列。

例

次のコードは、AsaDataAdapter オブジェクトを初期化します。

```
AsaDataAdapter da = new AsaDataAdapter(
 "SELECT emp_id, emp_lname FROM employee, conn );
```

AcceptChangesDuringFill プロパティ

説明

AcceptChanges が DataTable に追加された後に DataRow で呼び出されるかどうかを示す値。

構文

bool **AcceptChangesDuringFill**

アクセス

読み込み／書き込み

使用法

このプロパティが true の場合、DataAdapter は DataRow で AcceptChanges 関数を呼び出します。false の場合、AcceptChanges は呼び出されず、新しく追加されたローが、挿入されたローとして処理されます。デフォルトは true です。

ContinueUpdateOnError プロパティ

説明

ローの更新時にエラーが検出された場合、例外を生成するかどうかを指定する値。

構文

bool **ContinueUpdateOnError**

アクセス

読み込み／書き込み

使用法

デフォルトは false です。例外を生成しないで更新を継続するには、このプロパティを true に設定します。

ContinueUpdateOnError が true の場合、ローの更新時にエラーが発生しても例外はスローされません。ローの更新はスキップされ、エラー情報はローの RowError プロパティに格納されます。DataAdapter は後続のローの更新を継続します。

ContinueUpdateOnError が false の場合、エラーが発生すると例外がスローされます。

DeleteCommand プロパティ

説明

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() 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 AsaCommand オブジェクト。

構文

AsaCommand **DeleteCommand**

アクセス

読み込み／書き込み

使用法

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 AsaCommandBuilder を使用すると、DeleteCommand を自動的に生成できます。この場合、AsaCommandBuilder 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 AsaCommand オブジェクトに割り当てられる場合、AsaCommand オブジェクトのクローンは作成されません。DeleteCommand は、既存の AsaCommand への参照を保持します。

参照

「Update メソッド」515 ページ

「SelectCommand プロパティ」514 ページ

Fill メソッド

説明　データベースのデータを使用して DataSet または DataTable オブジェクトにローを追加したりこれらのローを再表示したりします。

構文 1

int **Fill(** DataSet *dataSet* **)**

構文 2

int **Fill(**
DataSet *dataSet*,
string *srcTable*
)

構文 3

int **Fill(**
DataSet *dataSet*,
int *startRecord*,
int *maxRecords*,
string *srcTable*
)

構文 4

int **Fill(** DataTable *dataTable* **)**

パラメータ

dataSet　レコード、および必要に応じてスキーマを使用して設定する DataSet。

srcTable　テーブルのマッピングに使用するソース・テーブルの名前。

startRecord　先頭のゼロ・ベースのレコード番号。

maxRecords　DataSet に読み込むレコードの最大数。

dataTable　レコード、および必要に応じてスキーマを使用して設定する DataTable。

戻り値　DataSet で正常に追加または再表示されたローの数。

使用法　startRecord 引数を使用して、DataSet にコピーされるレコードの数を制限しても、AsaDataAdapter クエリ内のすべてのレコードがデータベースからクライアントにフェッチされます。結果セットのサイズが大きい場合、パフォーマンスに重大な影響を及ぼす可能性があります。

別の方法として、読み込み専用、前方専用の結果セットで十分な場合、場合によっては SQL 文 (ExecuteNonQuery) とともに AsaDataReader を使用して修正を行う方法があります。さらにもう 1 つの方法として、必要な結果のみを返すストアド・プロシージャを作成する方法もあります。

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。

参照 「AsaDataAdapter オブジェクトを使用したデータの取得」455 ページ

FillError イベント

説明 設定オペレーションの実行時にエラーが発生したときに返されます。

構文 event FillErrorEventHandler **FillError**

使用法 FillError イベントを使用して、エラーが発生した後に設定オペレーションを継続するかどうかを決定できます。FillError イベントが発生する例は、次のとおりです。

- DataSet に追加されるデータを、精度を落とさないで共通言語ランタイム・タイプに変換できない。
- DataSet の DataColumn に適用する必要がある制約に違反するデータが、追加されるローに含まれている。

FillSchema メソッド

説明 DataTables 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。

構文 1

```
DataTable[ ] FillSchema(
    DataSet dataSet,
     SchemaType schemaType
    )
```

構文 2

DataTable[ ] **FillSchema(**
DataSet *dataSet*,
SchemaType *schemaType*,
string *srcTable*
)

構文 3

DataTable **FillSchema(**
DataTable *dataTable*,
SchemaType *schemaType*
)

パラメータ

dataSet　レコード、および必要に応じてスキーマを使用して設定する DataSet。

schemaType　スキーマを挿入する方法を指定する SchemaType 値の 1 つ。

srcTable　テーブルのマッピングに使用するソース・テーブルの名前。

dataTable　DataTable。

戻り値

構文 1 と 2 の場合、戻り値は、DataSet に追加された DataTable オブジェクトのコレクションへの参照です。構文 3 の場合、戻り値は DataTable への参照です。

参照

「AsaDataAdapter スキーマ情報の取得」463 ページ

GetFillParameters メソッド

説明

SELECT 文の実行時にユーザによって設定されるパラメータ。

構文

AsaParameter[ ] **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。

実装

IDataAdapter.GetFillParameters

InsertCommand プロパティ

説明

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() 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 AsaCommand オブジェクト。

構文

AsaCommand **InsertCommand**

アクセス

読み込み／書き込み

使用法

AsaCommandBuilder 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 AsaCommand オブジェクトに割り当てられる場合、AsaCommand オブジェクトのクローンは作成されません。InsertCommand は、既存の AsaCommand への参照を保持します。

このコマンドがローを返す場合、AsaCommand 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

「Update メソッド」515 ページ

「AsaCommand オブジェクトを使用したローの挿入、更新、削除」449 ページ

「AsaDataAdapter オブジェクトを使用したローの挿入、更新、削除」456 ページ

MissingMappingAction プロパティ

説明

受信データと一致するテーブルまたはカラムがない場合に実行するアクションを決定します。

構文

MissingMappingAction **MissingMappingAction**

アクセス

読み込み／書き込み

プロパティ値

MissingMappingAction 値の 1 つ。デフォルトは Passthrough です。

実装　IDataAdapter.MissingMappingAction

MissingSchemaAction プロパティ

説明　既存の DataSet スキーマが受信データと一致しない場合に実行するアクションを決定します。

構文　MissingSchemaAction **MissingSchemaAction**

アクセス　読み込み／書き込み

プロパティ値　MissingSchemaAction 値の 1 つ。デフォルトは Add です。

実装　IDataAdapter.MissingSchemaAction

RowUpdated イベント

説明　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　event AsaRowUpdatedEventHandler **RowUpdated**

使用法　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 AsaRowUpdatedEventArgs の引数を受け取ります。次の AsaRowUpdatedEventArgs プロパティは、このイベントに固有の情報を提供します。

- Command
- Errors
- RecordsAffected
- Row
- StatementType
- Status

- TableMapping

詳細については、.NET Framework のマニュアルの OleDbDataAdapter.RowUpdated イベントを参照してください。

RowUpdating イベント

説明 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 event AsaRowUpdatingEventHandler **RowUpdating**

使用法 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 AsaRowUpdatingEventArgs の引数を受け取ります。次の AsaRowUpdatingEventArgs プロパティは、このイベントに固有の情報を提供します。

- Command
- Errors
- Row
- StatementType
- Status
- TableMapping

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。

SelectCommand プロパティ

説明 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取得するために使用される AsaCommand。

構文 AsaCommand **SelectCommand**

アクセス　読み込み／書き込み

使用法　SelectCommand が以前に作成された AsaCommand オブジェクトに割り当てられる場合、AsaCommand 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 AsaCommand 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、AsaDataAdapter コンストラクタにも指定できます。

TableMappings プロパティ

説明　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。

構文　DataTableMappingCollection **TableMappings**

アクセス　読み込み専用

使用法　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、AsaDataAdapter 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。

Update メソッド

説明　DataSet の変更内容を使用してデータベース内のテーブルを更新します。

構文 1

int **Update(** DataSet *dataSet* **)**

構文 2

int **Update(**
    DataSet *dataSet*,
    string *srcTable*
)

構文 3 int **Update(** DataTable *dataTable* **)**

構文 4 int **Update(** DataRow[ ] *dataRows* **)**

パラメータ

dataSet レコード、および必要に応じてスキーマを使用して更新する DataSet。

srcTable テーブルのマッピングに使用するソース・テーブルの名前。

dataTable レコード、および必要に応じてスキーマを使用して更新する DataTable。

dataRows データ・ソースの更新に使用される DataRow オブジェクトの配列。

戻り値 DataSet から正常に更新されたローの数。

使用法 Update は、InsertCommand、UpdateCommand、DeleteCommand を使用して、挿入、更新、または削除されたデータ・セット内の各ローに対して実行されます。

参照

「DeleteCommand プロパティ」508 ページ

「InsertCommand プロパティ」512 ページ

「UpdateCommand プロパティ」516 ページ

「AsaDataAdapter オブジェクトを使用したローの挿入、更新、削除」456 ページ

UpdateCommand プロパティ

説明 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() 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 AsaCommand オブジェクト。

構文 AsaCommand **UpdateCommand**

アクセス 読み込み／書き込み

使用法

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 AsaCommandBuilder 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 AsaCommandBuilder 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 AsaCommand オブジェクトに割り当てられる場合、AsaCommand 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 AsaCommand 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、AsaCommand 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

「Update メソッド」515 ページ

AsaDataReader クラス

説明

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。

基本クラス

MarshalByRefObject

実装されたインタフェース

IDataReader, IDisposable, IDataRecord

使用法

AsaDataReader にはコンストラクタがありません。AsaDataReader オブジェクトを取得するには、AsaCommand を実行します。

```
AsaCommand cmd = new AsaCommand(
     "Select emp_id from employee", conn );
 AsaDataReader reader = cmd.ExecuteReader();
```

AsaDataReader 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、AsaDataAdapter を使用します。

AsaDataReader は必要に応じてローを取得しますが、AsaDataAdapter 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 AsaDataReader の方が応答時間が速くなります。

参照

「ExecuteReader メソッド」488 ページ

「データのアクセスと操作」445 ページ

Close メソッド

説明

AsaDataReader を閉じます。

構文

void **Close( )**

実装

IDataReader.Close

使用法　AsaDataReader を使用し終わったら、Close メソッドを明示的に呼び出す必要があります。

Depth プロパティ

説明　現在のローのネストの深さを示す値。最も深いテーブルの深さは 0 です。

構文　int **Depth**

アクセス　読み込み専用

プロパティ値　現在のローのネストの深さ。

実装　IDataReader.Depth

Dispose メソッド

説明　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　void **Dispose( )**

FieldCount プロパティ

説明　結果セット内のカラムの数。

構文　int **FieldCount**

アクセス　読み込み専用

プロパティ値　有効なレコード・セットに位置付けられていない場合は 0。それ以外の場合は、現在のレコード内のカラムの数。デフォルトは -1 です。

実装　IDataRecord.FieldCount

使用法 有効なレコード・セットに位置付けられていない場合、このプロパティの値は 0 です。それ以外の場合は、現在のレコード内のカラムの数です。デフォルトは -1 です。ローを返さないクエリを実行すると、FieldCount は 0 を返します。

GetBoolean メソッド

説明 ブール値として指定されているカラムの値。

構文 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。

実装 IDataRecord.GetBoolean

使用法 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。

GetByte メソッド

説明 バイトとして指定されているカラムの値。

構文 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。

実装 IDataRecord.GetByte

使用法 変換は行われないため、取り出されるデータはすでにバイトである必要があります。

GetBytes メソッド

説明

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

```
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。

実装

IDataRecord.GetBytes

使用法

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、AsaDataReader 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 Nothing) であるバッファを渡すと、GetBytes はフィールドの長さをバイト数で返します。

変換は行われないため、取り出されるデータはすでにバイト配列である必要があります。

GetChar メソッド

説明　文字として指定されているカラムの値。

構文　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。

実装　IDataRecord.GetChar

使用法　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。

AsaDataReader.IsDBNull 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　「IsDBNull メソッド」534 ページ

GetChars メソッド

説明　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

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。

実装　IDataRecord.GetChars

使用法　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、AsaDataReader 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 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。

参照　「BLOB の処理」468 ページ

GetDataTypeName メソッド

説明　ソース・データ型の名前。

構文　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。

実装　IDataRecord.GetDataTypeName

GetDateTime メソッド

説明　DateTime オブジェクトとして指定されているカラムの値。

構文 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。

実装 IDataRecord.GetDateTime

使用法 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。

AsaDataReader.IsDBNull 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 「IsDBNull メソッド」534 ページ

GetDecimal メソッド

説明 Decimal オブジェクトとして指定されているカラムの値。

構文 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。

実装 IDataRecord.GetDecimal

使用法 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。

AsaDataReader.IsDBNull 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 「IsDBNull メソッド」534 ページ

GetDouble メソッド

説明　倍精度の浮動小数点数として指定されているカラムの値。

構文　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。

実装　IDataRecord.GetDouble

使用法　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。

AsaDataReader.IsDBNull 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　「IsDBNull メソッド」534 ページ

GetFieldType メソッド

説明　オブジェクトのデータ型である Type。

構文　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。

実装　IDataRecord.GetFieldType

GetFloat メソッド

説明　単精度の浮動小数点数として指定されているカラムの値。

構文　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。

実装 IDataRecord.GetFloat

使用法 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。

AsaDataReader.IsDBNull 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 「IsDBNull メソッド」534 ページ

GetGuid メソッド

説明 グローバル一意識別子 (GUID) として指定されているカラムの値。

構文 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。

実装 IDataRecord.GetGuid

使用法 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。

AsaDataReader.IsDBNull 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 「IsDBNull メソッド」534 ページ

GetInt16 メソッド

説明 16 ビット符号付き整数として指定されているカラムの値。

構文　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。

実装　IDataRecord.GetInt16

使用法　変換は行われないため、取り出されるデータはすでに 16 ビット符号付き整数である必要があります。

GetInt32 メソッド

説明　32 ビット符号付き整数として指定されているカラムの値。

構文　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。

実装　IDataRecord.GetInt32

使用法　変換は行われないため、取り出されるデータはすでに 32 ビット符号付き整数である必要があります。

GetInt64 メソッド

説明　64 ビット符号付き整数として指定されているカラムの値。

構文　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。

実装　IDataRecord.GetInt64

使用法 変換は行われないため、取り出されるデータはすでに 64 ビット符号付き整数である必要があります。

GetName メソッド

説明 指定されたカラムの名前。

構文 string **GetName(** int *index* **)**

パラメータ **index** ゼロ・ベースのカラムのインデックス。

戻り値 指定されたカラムの名前。

実装 IDataRecord.GetName

GetOrdinal メソッド

説明 カラム名が与えられた場合のカラム序数。

構文 int **GetOrdinal(** string *name* **)**

パラメータ **name** カラム名。

戻り値 0 から始まるカラム序数。

実装 IDataRecord.GetOrdinal

使用法 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、結果をループ内で使用する整数変数に割り当てて、時間を節約してください。

GetSchemaTable メソッド

説明　AsaDataReader のカラム・メタデータが記述された DataTable を返します。

構文　DataTable **GetSchemaTable( )**

戻り値　カラム・メタデータが記述された DataTable。

実装　IDataReader.GetSchemaTable

使用法　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。

- ColumnName
- ColumnOrdinal
- ColumnSize
- NumericPrecision
- NumericScale
- IsUnique
- IsKey
- BaseCatalogName
- BaseColumnName
- BaseSchemaName
- BaseTableName
- DataType
- AllowDBNull
- ProviderType
- IsAliased

- IsExpression
- IsIdentity
- IsAutoIncrement
- IsRowVersion
- Is Hidden
- IsLong
- IsReadOnly

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。

参照 「DataReader スキーマ情報の取得」453 ページ

GetString メソッド

説明 文字列として指定されているカラムの値。

構文 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。

実装 IDataRecord.GetString

使用法 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。

AsaDataReader.IsDBNull 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 「IsDBNull メソッド」534 ページ

GetTimeSpan メソッド

説明　TimeSpan オブジェクトとして指定されているカラムの値。

構文　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。

使用法　カラムは、ASA 時間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。

AsaDataReader.IsDBNull 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　「時間値の取得」470 ページ

GetUInt16 メソッド

説明　16 ビット符号なし整数として指定されているカラムの値。

構文　UInt16 **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。

使用法　変換は行われないため、取り出されるデータはすでに 16 ビット符号なし整数である必要があります。

GetUInt32 メソッド

説明　32 ビット符号なし整数として指定されているカラムの値。

構文　UInt32 **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。

使用法 変換は行われないため、取り出されるデータはすでに 32 ビット符号なし整数である必要があります。

GetUInt64 メソッド

説明 64 ビット符号なし整数として指定されているカラムの値。

構文 UInt64 **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。

使用法 変換は行われないため、取り出されるデータはすでに 64 ビット符号なし整数である必要があります。

GetValue メソッド

説明 指定された順序にあるネイティブ・フォーマットのカラムの値。

構文 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 返される値。

実装 IDataRecord.GetValue

使用法 このメソッドは、NULL データベース・カラムに対して DBNull を返します。

GetValues メソッド

説明　現在のローのすべての属性カラム。

構文　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。

実装　IDataRecord GetValues

使用法

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。

指定された順序にあるネイティブ・フォーマットのカラムの値を取得します。

IsClosed プロパティ

説明　AsaDataReader が閉じられている場合、true を返します。
AsaDataReader が閉じられていない場合、false を返します。

構文　bool **IsClosed**

アクセス　読み込み専用

プロパティ値　AsaDataReader が閉じられている場合は true、閉じられていない場合は false。

実装 IDataReader.IsClosed

使用法 AsaDataReader 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。

IsDBNull メソッド

説明 カラムに NULL 値が含まれるかどうかを示す値。

構文 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。等しくない場合は false。

実装 IDataRecord.IsDbNull

使用法 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。

Item プロパティ

説明 ネイティブ・フォーマットのカラムの値。C# では、このプロパティは AsaDataReader クラスのインデクサです。

構文 1 object **this[** int *index* **]**

構文 2 object **this[** string *name* **]**

パラメータ **index** カラム序数。

name カラム名。

アクセス 読み込み専用

実装 IDataRecord.Item

NextResult メソッド

説明　バッチ SQL 文の結果を読み込むときに AsaDataReader を次の結果に進めます。

構文　bool **NextResult( )**

戻り値　さらに結果セットがある場合は true。ない場合は false。

実装　IDataReader.NextResult

使用法　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。

Read メソッド

説明　結果セットの次のローを読み込み、AsaDataReader をこのローに移動します。

構文　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。

実装　IDataReader.Read

使用法　AsaDataReader 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();
```

RecordsAffected プロパティ

説明

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。

構文

int **RecordsAffected**

アクセス

読み込み専用

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。

実装

IDataReader.RecordsAffected

使用法

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。

AsaDataReader 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。

AsaDbType 一覧

Adaptive Server Anywhere のデータ型を指定します。

メンバ

BigInt

Binary

Bit

Char

Date

Decimal

Double

Float

Integer

LongBinary

LongVarchar

Numeric

SmallInt

Time

TimeStamp

TinyInt

UnsignedBigInt

UnsignedInt

UnsignedSmallInt

VarBinary

VarChar

UniqueIdentifier

AsaError クラス

説明　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。

基本クラス　Object

AsaError にはコンストラクタがありません。

参照　「エラー処理と Adaptive Server Anywhere .NET データ・プロバイダ」 478 ページ

Message プロパティ

説明　エラーの簡単な説明。

構文　string **Message**

アクセス　読み込み専用

NativeError プロパティ

説明　データベース固有のエラー情報。

構文　int **NativeError**

アクセス　読み込み専用

Source プロパティ

説明　エラーを生成したプロバイダの名前。

構文　string **Source**

アクセス　読み込み専用

SqlState プロパティ

説明 ANSI SQL 規格に続く Adaptive Server Anywhere の 5 文字の SQL 状態。エラーが複数の場所から発行される場合、5 文字のエラー・コードはエラーのソースを示します。

構文 string **SqlState**

アクセス 読み込み専用

ToString メソッド

説明 エラー・メッセージの完全なテキスト。

構文 string **ToString( )**

使用法 戻り値は、"AsaError:" 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。

```
AsaError:UserId or Password not valid.
```

AsaErrorCollection クラス

説明　Adaptive Server Anywhere ADO.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。

基本クラス　Object

実装されたインタフェース　ICollection, IEnumerable

AsaErrorCollection にはコンストラクタがありません。通常、AsaErrorCollection は AsaException.Errors プロパティから取得されます。

参照　「Errors プロパティ」 543 ページ

「エラー処理と Adaptive Server Anywhere .NET データ・プロバイダ」 478 ページ

CopyTo メソッド

説明　配列内の特定のインデックスから開始して AsaErrorCollection の要素を配列にコピーします。

構文

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。

実装　ICollection.CopyTo

Count プロパティ

説明　コレクション内のエラーの数。

構文 int **Count**

アクセス 読み込み専用

実装 ICollection.Count

Item プロパティ

説明 指定されたインデックスのエラー。

構文 AsaError **this[** int *index* **]**

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のエラーが含まれる AsaError。

アクセス 読み込み専用

AsaException クラス

説明　Adaptive Server 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。

基本クラス　SystemException

AsaException にはコンストラクタがありません。通常、AsaException 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( AsaException ex )
 {
     MessageBox.Show( ex.Errors[0].Message, "Error" );
 }
```

参照　「エラー処理と Adaptive Server Anywhere .NET データ・プロバイダ」 478 ページ

Errors プロパティ

説明　1 つまたは複数の AsaError オブジェクトのコレクション。

構文　AsaErrorCollection **Errors**

アクセス　読み込み専用

使用法　AsaErrorCollection クラスには常に少なくとも 1 つの AsaError クラスのインスタンスがあります。

GetObjectData メソッド

説明　このメンバは Exception.GetObjectData を上書きします。

構文

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。

Message プロパティ

説明

エラーが記述されたテキスト。

構文

string **Message**

アクセス

読み込み専用

使用法

このメソッドは、Errors コレクション内のすべての AsaError 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。

Source プロパティ

説明

エラーを生成したプロバイダの名前。

構文

string **Source**

アクセス

読み込み専用

AsaInfoMessageEventArgs クラス

説明　InfoMessage イベントのデータを提供します。

基本クラス　EventArgs

AsaInfoMessageEventArgs にはコンストラクタがありません。

Errors プロパティ

説明　データ・ソースから送信された警告のコレクション。

構文　AsaErrorCollection **Errors**

アクセス　読み込み専用

Message プロパティ

説明　データ・ソースから送信されたエラーの完全なテキスト。

構文　string **Message**

アクセス　読み込み専用

Source プロパティ

説明　エラーを生成したオブジェクトの名前。

構文　string **Source**

アクセス　読み込み専用

ToString メソッド

説明　InfoMessage イベントの文字列表現を取り出します。

構文 string **ToString( )**

戻り値 InfoMessage イベントを示す文字列。

AsaInfoMessageEventHandler 委任

説明

AsaConnection の InfoMessage イベントを処理するメソッドを示します。

構文

void **AsaInfoMessageEventHandler (**
object *sender*,
AsaInfoMessageEventArgs *e*
)

パラメータ

sender　イベントのソース。

e　イベント・データが含まれる AsaInfoMessageEventArgs オブジェクト。

AsaParameter クラス

説明 AsaCommand 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。

基本クラス MarshalByRefObject

実装されたインタフェース IDbDataParameter, IDataParameter

AsaParameter コンストラクタ

構文 1

void **AsaParameter( )**

構文 2

void **AsaParameter(**
string *parameterName*,
object *value*
)

構文 3

void **AsaParameter(**
string *parameterName*,
AsaDbType *dbType*
)

構文 4

void **AsaParameter(**
string *parameterName*,
AsaDbType *dbType*,
int *size*
)

構文 5

void **AsaParameter(**
string *parameterName*,
AsaDbType *dbType*,
int *size*,
string *sourceColumn*
)

構文 6

void **AsaParameter(**
string *parameterName*,
AsaDbType *dbType*,
int *size*,

ParameterDirection *direction*,
bool *isNullable*,
byte *precision*,
byte *scale*,
string *sourceColumn*,
DataRowVersion *sourceVersion*,
object *value*
)

パラメータ

value　パラメータの値である Object。

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。

parameterName　パラメータの名前。

dbType　AsaDbType 値の 1 つ。

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 が解析される小数点の左右の桁の合計数。

scale　Value が解析される小数点までの桁の合計数。

sourceVersion　DataRowVersion 値の 1 つ。

AsaDbType プロパティ

説明

パラメータの AsaDbType。

構文

AsaDbType **AsaDbType**

アクセス

読み込み／書き込み

使用法

AsaDbType と DbType はリンクされます。このため、DbType を設定すると、AsaDbType がサポートされている AsaDbType に変更されます。

この値は、AsaDbType 列挙のメンバにする必要があります。

DbType プロパティ

説明 パラメータの DbType。

構文 DbType **DbType**

アクセス 読み込み／書き込み

使用法 AsaDbType と DbType はリンクされます。このため、DbType を設定すると、AsaDbType がサポートされている AsaDbType に変更されます。

この値は、AsaDbType 列挙のメンバにする必要があります。

Direction プロパティ

説明 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。

構文 ParameterDirection **Direction**

アクセス 読み込み／書き込み

使用法 ParameterDirection が出力である場合、関連付けられている AsaCommand を実行しても値は返らず、AsaParameter には NULL 値が含まれます。最後の結果セットの最後のローが読み込まれると、Output、InputOut、ReturnValue パラメータが更新されます。

IsNullable プロパティ

説明 パラメータが NULL 値を受け入れるかどうかを示す値。

構文 bool **IsNullable**

アクセス 読み込み／書き込み

使用法 NULL 値が受け入れられる場合、この値は true です。受け入れられない場合は false です。デフォルトは false です。NULL 値は DBNull クラスを使用して処理されます。

Offset プロパティ

説明　Value プロパティのオフセット。

構文　int **Offset**

アクセス　読み込み／書き込み

プロパティ値　値のオフセット。デフォルトは 0 です。

ParameterName プロパティ

説明　AsaParameter の名前。

構文　string **ParameterName**

アクセス　読み込み／書き込み

実装　IDataParameter.ParameterName

使用法　Adaptive Server 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。

デフォルトは、空の文字列です。

Precision プロパティ

説明　Value プロパティを示すために使用される最大桁数。

構文　byte **Precision**

アクセス　読み込み／書き込み

実装　IDbDataParameter.Precision

使用法

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。

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。

Scale プロパティ

説明

Value が解析される小数点までの桁の数。

構文

byte **Scale**

アクセス

読み込み／書き込み

実装

IDbDataParameter.Scale

使用法

デフォルト値は 0 です。Scale プロパティは、10 進数および数値入力パラメータに対してのみ使用されます。

Size プロパティ

説明

カラム内のデータの最大サイズ (バイト単位)。

構文

int **Size**

アクセス

読み込み／書き込み

実装

IDbDataParameter.Size

使用法

このプロパティの値は、カラム内のデータの最大サイズ (バイト単位) です。デフォルト値はパラメータ値から推測されます。

Size プロパティは、バイナリおよび文字列型に対して使用されます。

可変長のデータ型の場合、Size プロパティは、サーバに送信するデータの最大量を示します。たとえば、Size プロパティを使用して、サーバに送信されるデータ量を文字列値の最初の 100 バイトに制限できます。

このプロパティを明示的に設定しない場合、サイズは、指定されたパラメータ値の実際のサイズから推測されます。固定幅のデータ型の場合、Size の値は無視されます。この値は情報用として取り出すことができ、プロバイダがパラメータの値をサーバに送信するときに使用する最大量を返します。

SourceColumn プロパティ

説明　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。

構文　string **SourceColumn**

アクセス　読み込み／書き込み

実装　IDbDataParameter.SourceColumn

使用法　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。

SourceVersion プロパティ

説明　Value をロードするときに使用する DataRowVersion。

構文　DataRowVersion **SourceVersion**

アクセス　読み込み／書き込み

実装　IDbDataParameter.SourceVersion

使用法

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。

ToString メソッド

説明

ParameterName が含まれる文字列。

構文

string **ToString( )**

アクセス

読み込み／書き込み

Value プロパティ

説明

パラメータの値。

構文

object **Value**

アクセス

読み込み／書き込み

実装

IDataParameter.Value

使用法

入力パラメータの場合、この値は、サーバに送信される AsaCommand 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、AsaDataReader が閉じられてから AsaCommand 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、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 AsaDbType 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。

AsaParameterCollection クラス

説明

AsaCommand 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。

基本クラス

Object

実装されたインタフェース

ICollection, IEnumerable, IDataParameterCollection

使用法

AsaParameterCollection にはコンストラクタがありません。AsaParameterCollection は、AsaCommand.Parameters プロパティから取得します。

参照

「Parameters プロパティ」489 ページ

Add メソッド

説明

AsaParameter を AsaCommand に追加します。

構文 1

int **Add(** object *value* **)**

構文 2

int **Add(** AsaParameter *value* **)**

構文 3

int **Add(**
string *parameterName*,
object *value*
)

構文 4

int **Add(**
string *parameterName*,
AsaDbType *asaDbType*
)

構文 5

int **Add(**
string *parameterName*,
AsaDbType *asaDbType*,
int *size*
)

構文 6

int **Add(**
string *parameterName*,
AsaDbType *asaDbType*,
int *size*,
string *sourceColumn*
)

パラメータ

value　構文 1 および 2 の場合、値は、コレクションに追加する AsaParameter オブジェクトです。構文 3 の場合、値は、コレクションに追加するパラメータの値です。

parameterName　パラメータの名前。

asaDbType　AsaDbType 値の 1 つ。

size　カラムの長さ。

sourceColumn　ソース・カラムの名前。

戻り値

新しい AsaParameter オブジェクトのインデックス。

Clear メソッド

説明

コレクションからすべての項目を削除します。

構文

void **Clear( )**

実装

IList.Clear

Contains メソッド

説明

コレクション内に AsaParameter があるかどうかを示す値。

構文 1

bool **Contains(** object *value* **)**

構文 2

bool **Contains(** string *value* **)**

パラメータ

value　検索する AsaParameter オブジェクトの値。構文 2 の場合、これは名前です。

戻り値 コレクションに AsaParameter が含まれる場合は true。含まれない場合は false。

実装 構文 1 は IList.Contains を実装します。

構文 2 は IDataParameterCollection.Contains を実装します。

CopyTo メソッド

説明 AsaParameter オブジェクトを AsaParameterCollection から指定された配列にコピーします。

構文

void **CopyTo(**
array *array*
int *index*
)

パラメータ **array** AsaParameter 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。

実装 ICollection.CopyTo

Count プロパティ

説明 コレクション内の AsaParameter オブジェクトの数。

構文 int **Count**

アクセス 読み込み専用

実装 ICollection.Count

IndexOf メソッド

説明 コレクション内の AsaParameter のロケーション。

構文 1　int **IndexOf(** object *value* **)**

構文 2　int **IndexOf(** string *parameterName* **)**

パラメータ　**value**　検索する AsaParameter オブジェクト。

parameterName　検索する AsaParameter オブジェクトの名前。

戻り値　コレクション内の AsaParameter の 0 から始まるロケーション。

実装　構文 1 は IList.IndexOf を実装します。

構文 2 は IDataParameterCollection.IndexOf を実装します。

Insert メソッド

説明　指定されたインデックスでコレクションに AsaParameter を挿入します。

構文

```
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する AsaParameter。

実装　IList.Insert

Item プロパティ

説明　指定されたインデックスまたは名前の AsaParameter。

構文 1　AsaParameter **this[** int *index* **]**

構文 2　AsaParameter **this[** string *parameterName* **]**

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。

parameterName 取り出すパラメータの名前。

プロパティ値 AsaParameter。

アクセス 読み込み／書き込み

使用法 C# では、このプロパティは AsaParameterCollection クラスのインデクサです。

Remove メソッド

説明 指定された AsaParameter をコレクションから削除します。

構文 void **Remove(** object *value* **)**

パラメータ **value** コレクションから削除する AsaParameter オブジェクト。

実装 IList.Remove

RemoveAt メソッド

説明 指定された AsaParameter をコレクションから削除します。

構文 1 void **RemoveAt(** int *index* **)**

構文 2 void **RemoveAt(** string *parameterName* **)**

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。

parameterName 削除する AsaParameter オブジェクトの名前。

実装 構文 1 は IList.RemoveAt を実装します。

構文 2 は IDataParameterCollection.RemoveAt を実装します。

AsaPermission クラス

説明　ユーザが Adaptive Server 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、Adaptive Server Anywhere .NET データ・プロバイダが確認できるようにします。

基本クラス　DBDataPermission

AsaPermission コンストラクタ

説明　AsaPermission クラスの新しいインスタンスを初期化します。

構文 1　void **AsaPermission( )**

構文 2　void **AsaPermission(** PermissionState *state* **)**

構文 3

void **AsaPermission(**
    PermissionState *state*,
    bool *allowBlankPassword*
)

パラメータ

state　PermissionState 値の 1 つ。

allowBlankPassword　ブランク・パスワードを使用できるかどうかを示します。

AsaPermissionAttribute クラス

説明 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。

基本クラス DBDataPermissionAttribute

AsaPermissionAttribute コンストラクタ

説明 AsaPermissionAttribute クラスの新しいインスタンスを初期化します。

構文 void **AsaPermissionAttribute(** 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。

戻り値 AsaPermissionAttribute オブジェクト。

CreatePermission メソッド

説明 属性プロパティに応じて設定された AsaPermission オブジェクトを返します。

構文 IPermission **CreatePermission( )**

AsaRowUpdatedEventArgs クラス

説明　RowUpdated イベントのデータを提供します。

基本クラス　RowUpdatedEventArgs

AsaRowUpdatedEventArgs コンストラクタ

説明　AsaRowUpdatedEventArgs クラスの新しいインスタンスを初期化します。

構文

void **AsaRowUpdatedEventArgs(**
DataRow *dataRow*,
IDbCommand *command*,
StatementType *statementType*,
DataTableMapping *tableMapping*
)

パラメータ

dataRow　Update を介して送信された DataRow。

command　Update が呼び出されたときに実行された IDbCommand。

statementType　実行されたクエリのタイプを指定する StatementType 値の 1 つ。

tableMapping　Update を介して送信された DataTableMapping。

Command プロパティ

説明　Update が呼び出されたときに実行された AsaCommand。

構文　AsaCommand **Command**

アクセス　読み込み専用

Errors プロパティ

説明 Command が実行されたときに Adaptive Server Anywhere によって生成されたエラー。RowUpdatedEventArgs から継承されます。

構文 Exception **Errors**

プロパティ値 Command が実行されたときに Adaptive Server Anywhere によって生成されたエラー。

アクセス 読み込み／書き込み

RecordsAffected プロパティ

説明 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。RowUpdatedEventArgs から継承されます。

構文 int **RecordsAffected**

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。

アクセス 読み込み専用

Row プロパティ

説明 Update を介して送信された DataRow。RowUpdatedEventArgs から継承されます。

構文 DataRow **Row**

アクセス 読み込み専用

StatementType プロパティ

説明　実行された SQL 文のタイプ。RowUpdatedEventArgs から継承されます。

構文　StatementType **StatementType**

アクセス　読み込み専用

使用法　StatementType は、**Select**、**Insert**、**Update**、または **Delete** の 1 つです。

Status プロパティ

説明　Command プロパティの UpdateStatus。RowUpdatedEventArgs から継承されます。

構文　UpdateStatus **Status**

プロパティ値　UpdateStatus の値である **Continue**、**ErrorsOccurred**、**SkipAllRemainingRows**、または **SkipCurrentRow** の 1 つ。デフォルトは **Continue** です。

アクセス　読み込み／書き込み

TableMapping プロパティ

説明　Update を介して送信された DataTableMapping。RowUpdatedEventArgs から継承されます。

構文　DataTableMapping **TableMapping**

アクセス　読み込み専用

AsaRowUpdatingEventArgs クラス

説明 RowUpdating イベントのデータを提供します。

基本クラス RowUpdatingEventArgs

AsaRowUpdatingEventArgs コンストラクタ

説明 AsaRowUpdatingEventArgs クラスの新しいインスタンスを初期化します。

構文

void **AsaRowUpdatingEventArgs(**
DataRow *row*,
IDbCommand *command*,
StatementType *statementType*,
DataTableMapping *tableMapping*
)

パラメータ

row 更新する DataRow。

command 更新時に実行する IDbCommand。

statementType 実行されたクエリのタイプを指定する StatementType 値の 1 つ。

tableMapping Update を介して送信された DataTableMapping。

Command プロパティ

説明 Update の実行時に実行する AsaCommand。

構文 AsaCommand **Command**

アクセス 読み込み／書き込み

Errors プロパティ

説明　Command が実行されたときに Adaptive Server Anywhere によって生成されたエラー。RowUpdatingEventArgs から継承。

構文　Exception **Errors**

プロパティ値　Command が実行されたときに Adaptive Server Anywhere によって生成されたエラー。

アクセス　読み込み／書き込み

Row プロパティ

説明　Update を介して送信された DataRow。RowUpdatingEventArgs から継承。

構文　DataRow **Row**

アクセス　読み込み専用

StatementType プロパティ

説明　実行された SQL 文のタイプ。RowUpdatingEventArgs から継承。

構文　StatementType **StatementType**

アクセス　読み込み専用

使用法　StatementType は、**Select**、**Insert**、**Update**、または **Delete** の 1 つです。

Status プロパティ

説明　Command プロパティの UpdateStatus。RowUpdatingEventArgs から継承。

構文 UpdateStatus **Status**

プロパティ値 UpdateStatus の値である **Continue**、**ErrorsOccurred**、**SkipAllRemainingRows**、または **SkipCurrentRow** の 1 つ。デフォルトは **Continue** です。

アクセス 読み込み／書き込み

TableMapping プロパティ

説明 Update を介して送信された DataTableMapping。RowUpdatingEventArgs から継承。

構文 DataTableMapping **TableMapping**

アクセス 読み込み専用

AsaRowUpdatedEventHandler 委任

説明

AsaDataAdapter の RowUpdated イベントを処理するメソッドを示します。

構文

void **AsaRowUpdatedEventHandler (**
object *sender*,
AsaRowUpdatedEventArgs *e*
)

パラメータ

sender　イベントのソース。

e　イベント・データが含まれる AsaRowUpdatedEventArgs。

AsaRowUpdatingEventHandler 委任

説明

AsaDataAdapter の RowUpdating イベントを処理するメソッドを示します。

構文

void **AsaRowUpdatingEventHandler (**
object *sender*,
AsaRowUpdatingEventArgs *e*
)

パラメータ

sender イベントのソース。

e イベント・データが含まれる AsaRowUpdatingEventArgs。

AsaTransaction クラス

説明　SQL トランザクションを示します。

基本クラス　Object

実装されたインタフェース　IDbTransaction

使用法　AsaTransaction にはコンストラクタがありません。AsaTransaction オブジェクトを取得するには、AsaConnection.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、AsaCommand.Transaction プロパティを使用します。

参照

「BeginTransaction メソッド」499 ページ

「Transaction プロパティ」490 ページ

「Transaction 処理」475 ページ

「AsaCommand オブジェクトを使用したローの挿入、更新、削除」449 ページ

Commit メソッド

説明　データベース・トランザクションをコミットします。

構文　void **Commit( )**

実装　IDbTransaction.Commit

Connection プロパティ

説明　トランザクションに関連付けられている AsaConnection オブジェクト。トランザクションが無効な場合は NULL 参照 (Visual Basic の Nothing)。

構文 AsaConnection **Connection**

アクセス 読み込み専用

使用法 1 つのアプリケーションに複数のデータベース接続があり、各接続に 0 以上のトランザクションがある場合があります。このプロパティを使用して、BeginTransaction によって作成された特定のトランザクションに関連付けられた接続オブジェクトを確認できます。

IsolationLevel プロパティ

説明 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 IsolationLevel *IsolationLevel*

アクセス 読み込み専用

プロパティ値 このトランザクションの IsolationLevel。これは、**ReadCommitted**、**ReadUncommitted**、**RepeatableRead**、または **Serializable.** の 1 つです。デフォルトは **ReadCommitted** です。

実装 IDbTransaction.IsolationLevel

使用法 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。

Rollback メソッド

説明 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 1 void **Rollback( )**

構文 2 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。

使用法　このトランザクションがロールバックできるのは、保留状態 (BeginTransaction が呼び出された後だが、Commit が呼び出される前) からのみです。

Save メソッド

説明　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　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。

第 14 章

Open Client インタフェース

この章の内容

この章では、Adaptive Server Anywhere 用の 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。

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、Adaptive Server Anywhere 特有の機能について説明していますが、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。

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、アプリケーションが動いているマシンに 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、Adaptive Server 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・サーバが動いているマシンでは、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、Adaptive Server 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。

Adaptive Server Anywhere を使った Open Client アプリケーションの実行については、『ASA データベース管理ガイド』> 「Open Server としての Adaptive Server Anywhere」を参照してください。

データ型マッピング

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 Adaptive Server Anywhere で使用されるものと細部が多少異なります。このため、Adaptive Server Anywhere は Open Client アプリケーションと Adaptive Server 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。

Adaptive Server 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 Adaptive Server 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 Adaptive Server Anywhere のデータ型

次の表は、Adaptive Server 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。

ASA データ型	Open Client データ型
unsigned short	int
unsigned int	bigint
unsigned bigint	bigint
date	smalldatetime
time	smalldatetime
serialization	longbinary
string	varchar
timestamp struct	datetime

データ型マッピングの範囲制限

データ型によっては、Adaptive Server 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、Adaptive Server 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 Adaptive Server Anywhere データ型のほうが取り得る値の範囲が大きくなっています。その結果、Adaptive Server 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	ASA の最小値	ASA 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 Adaptive Server 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 Adaptive Server Anywhere のカラムに設定できます。クライアントが Adaptive Server 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

Adaptive Server 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 Adaptive Server Anywhere 実装は、Adaptive Server Enterprise の場合と異なります。Adaptive Server

Anywhere の場合、その値は Adaptive Server Anywhere の DATETIME データ型にマッピングされます。Adaptive Server 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、Adaptive Server 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、Adaptive Server Anywhere の DATETIME データ型は、Adaptive Server 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、Adaptive Server 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」11 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 employee
                     WHERE emp_id=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 *type* 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 *type* パラメータで ct_dynamic を使用して文を実行します。

4　CS_DEALLOC *type* 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

Adaptive Server 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 (下記参照) 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 でカーソルを開くには、CS_CURSOR_OPEN を *type* パラメータに指定して 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 Adaptive Server 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面の処理に `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。

Adaptive Server 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、Adaptive Server 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** Adaptive Server 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_REG_NOTIF
  - CS_CSR_ABS
  - CS_CSR_FIRST
  - CS_CSR_LAST
  - CS_CSR_PREV
  - CS_CSR_REL
  - CS_DATA_BOUNDARY
  - CS_DATA_SENSITIVITY
  - CS_PROTO_DYNPROC
  - 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 またはローカル・マシンの NamedPipes プロトコルが利用可能であれば、これらを利用して Adaptive Server 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。

第 15 章

DBD::ASAny Perl インタフェース

この章の内容

この章では、DBD::ASAny インタフェースをインストールして使用する方法について説明します。この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が、Adaptive Server Anywhere データベースに格納された情報にアクセスして操作できるようになります。

DBD::ASAny について

DBD::ASAny 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 Adaptive Server Anywhere データベースにアクセスできるようになります。ASAny 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::ASAny をインストールすると、Perl から Adaptive Server 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::ASAny 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::ASAny 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::ASAny のインストール

次の手順は、Windows で DBD::ASAny インタフェースをインストールする方法を示します。

❖ マシンを準備するには、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、マシンを設定できます。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 *Common7¥Tools* サブディレクトリには *vsvars32.bat* と呼ばれるバッチ・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
```

❖ Windows で DBD::ASAny 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・コマンド・プロンプトで、SQL Anywhere Studio 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、ASAny 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::ASAny をテストするには、Adaptive Server Anywhere Sample Database ファイルを DBD::ASAny ディレクトリにコピーし、テストを行います。

```
copy "C:¥Program Files¥Sybase¥SQL Anywhere
9¥asademo.db"

dbeng9 asademo
```

```
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::ASAny インタフェースが使用可能になりました。

UNIX での DBD::ASAny 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::ASAny インタフェースをインストールする方法を示します。

❖ マシンを準備するには、次の手順に従います。

1 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。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。

2 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。

3 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
```

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。

```
make install
```

5 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::ASAny をインストールするには、次の手順に従います。

1　Adaptive Server 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、Adaptive Server Anywhere の設定スクリプトを SQL Anywhere Studio のインストール・ディレクトリから取得します。

シェル	使用するコマンド
sh、ksh、または bash	`. /bin/asa_config.sh`
csh または tcsh	`source /bin/asa_config.csh`

2　シェル・プロンプトで、SQL Anywhere Studio 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 ASAny 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::ASAny をテストするには、Adaptive Server Anywhere Sample Database ファイルを DBD::ASAny ディレクトリにコピーし、テストを行います。

```
cp /opt/sybase/SYBSsa9/asademo.db .

dbeng9 asademo

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 *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。

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::ASAny インタフェースが使用可能になりました。

DBD::ASAny を使用する Perl スクリプトの作成

この項では、DBD::ASAny 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::ASAny 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::ASAny 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、ASAny もその 1 つである DBD ドライバ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:ASAny:" 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、Adaptive Server 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 = "asademo";
 my $data_src =
"DBI:ASAny:ENG=$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 "Can't connect to $data_source:
$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 = "asademo";
 my $data_src =
"DBI:ASAny:ENG=$database;DBN=$database";
 my $uid      = "dba";
 my $pwd      = "sql";
 my $sel_stmt = "SELECT id, fname, lname
                 FROM customer
                 ORDER BY fname, lname";
 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、Adaptive Server 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。

詳細については、『ASA データベース管理ガイド』> 「MAX_CURSOR_COUNT オプション [データベース]」と 『ASA データベース管理ガイド』> 「MAX_STATEMENT_COUNT オプション [データベース]」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 = "asademo";
 my $data_src =
"DBI:ASAny:ENG=$database;DBN=$database";
 my $uid      = "dba";
 my $pwd      = "sql";
 my $ins_stmt = "INSERT INTO customer (id, fname, lname,
                         address, city, state, zip,
                         phone, company_name)
                  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,78312,43234,BXM",
```

```
          "802,Zach,Zed,82 Fair St,New
York,NY,12333,72234,Zap"
         );
        $sth = $dbh->prepare($ins);
        my $row = undef;
        foreach $row ( @rows ) {
          my @values = split(/,/, $row);
          $sth->execute(@values);
        }
 }
 __END__
```

第 16 章

SQLAnywhere PHP モジュールの使用

この章の内容

この章では、PHP スクリプトから SQL Anywhere データベースにアクセスする SQLAnywhere PHP モジュールのインストール方法および使用方法について説明します。

SQLAnywhere PHP モジュールについて

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 Adaptive Server 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 Adaptive Server 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、Adaptive Server 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 Windows バージョンでは DLL によって、UNIX、Linux、Mac OS X のすべてのバージョンでは共有オブジェクトによって提供されます。

稼働条件

必要なソフトウェアは次のとおりです。

- Adaptive Server Anywhere 9.0.2 以上
- 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)。Adaptive Server Anywhere は、Web サーバと同じマシンでも異なるマシンでも実行できます。
- Adaptive Server Anywhere クライアント・ソフトウェア *dblib9.dll* (Windows)、*libdblib9.so* (Linux および UNIX)、*libdblib9.dylib.1* (Mac OS X)

SQLAnywhere PHP のインストールと設定

次の各項では、SQLAnywhere 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。

UNIX の場合、SQL Anywhere には PHP バージョン 4 および 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。UNIX の場合の PHP は、コマンド・ラインで使用できる CGI バージョンとして、また Apache Web サーバの一部として提供されます。CGI バージョンの PHP を使用する場合、または Apache 1.x を使用する場合は、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
php4_sqlanywhere9.dll	Windows 用の PHP バージョン 4 ライブラリ
php5_sqlanywhere9.dll	Windows 用の PHP バージョン 4 ライブラリ
php4_sqlanywhere9.so	UNIX 用の非スレッド型 PHP バージョン 4 ライブラリ
php4_sqlanywhere9_r.so	UNIX 用のスレッド型 PHP バージョン 4 ライブラリ
php5_sqlanywhere9.so	UNIX 用の非スレッド型 PHP バージョン 5 ライブラリ
php5_sqlanywhere9_r.so	UNIX 用のスレッド型 PHP バージョン 5 ライブラリ

UNIX と Mac OS X での PHP モジュールのインストール

UNIX、Linux、Mac OS X で SQLAnywhere 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。このあとにオプションとして、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。

❖ UNIX または Mac OS X 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 インストールにある *php.ini* ファイルを探して、それをテキスト・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。

   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」601 ページを参照してください。

3. オプションとして、SQLAnywhere 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_sqlanywhere9.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_sqlanywhere9_r.so
   ```

   ここで、*X* は PHP バージョン番号です。

4　PHP モジュールを使用する前に、Adaptive Server 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 Studio のインストール・ディレクトリから Adaptive Server 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/asa_config.sh`
csh または tcsh	`source /bin/asa_config.csh`

5　コマンド・プロンプトで、SQL Anywhere のインストール環境のルート・ディレクトリに移動します。次のコマンドを入力して Adaptive Server Anywhere サンプル・データベースを起動します。

```
dbeng9 asademo.db
```

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」608 ページを参照してください。

Windows での PHP モジュールのインストール

Windows で SQLAnywhere 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。このあとにオプションとして、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. ファイル *php_sqlanywhere9.dll* を、SQL Anywhere インストール環境の *win32* 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。
3. SQLAnywhere 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。

   ```
   extension=phpX_sqlanywhere9.dll
   ```

   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。詳細については、「SQLAnywhere PHP の設定」605 ページを参照してください。
4. コマンド・プロンプトで、SQL Anywhere のインストール環境のルート・ディレクトリに移動します。次のコマンドを入力して Adaptive Server Anywhere サンプル・データベースを起動します。

   ```
   dbeng9 asademo.db
   ```

コマンドによって、この章のサンプルで必要な、サーバ名 **asademo** のデータベース・サーバが起動されます。Windows のメニューからサーバを起動する場合、サーバ名は **asademo9** であり、この章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。

6　ここまで終了したら、コンソール・ウィンドウで [シャットダウン] をクリックして、Adaptive Server 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」608 ページを参照してください。

SQLAnywhere PHP 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。現在、次のエントリがサポートされています。

- **extension** PHP が起動するたびに、PHP が SQLAnywhere 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_sqlanywhere9.dll
  ```

  UNIX プラットフォームおよび Mac OS X 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_sqlanywhere9.so

extension=phpX_sqlanywhere9_r.so
```

SQLAnywhere PHP モジュールがロードされるようにするために、作成する各スクリプトの最初に次の行のコードを記述することもできます。この記述が必要なのは、PHP が起動されるたびに SQLAnywhere PHP モジュールが自動的にロードされる必要がない場合のみです。

```
# Ensure that the SQLAnywhere PHP module is loaded
   if( !extension_loaded('sqlanywhere')) {
    if( strtoupper(substr(PHYP_OS, 0, 3) == 'WIN' ))
{
        dl('phpX_sqlanywhere9.dll');
     } else {
         dl('phpX_sqlanywhere9.so');
     }
   }
```

- **allow_persistent** 1 に設定すると PHP モジュールは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** SQLAnywhere 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** 1 に設定すると、各文の実行直後にデータベース・サーバがコミット操作を自動的に実行します。0 に設定すると、必要に応じて sqlanywhere_commit() または sqlanywhere_rollback() プロシージャ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
  ```

詳細については、「sqlanywhere_set_option」631 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、Adaptive Anywhere 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 username]/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 Apache のインストールが共に適切に機能していることが確認されます。

3　PHP と Apache が Adaptive Server Anywhere と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。

❖ SQLAnywhere PHP モジュールを使用するクエリ・ページの作成

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 *asa_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 employee" );
   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 SQLAnywhere PHP モジュールをインストールおよび設定したあとで、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 UNIX 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 Adaptive Server 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 Studio のインストール・ディレクトリから Adaptive Server 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/asa_config.sh`
csh または tcsh	`source /bin/asa_config.csh`

3 Adaptive Server 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 Adaptive Server Anywhere サンプル・データベースを起動します。

```
dbeng9 asademo.db
```

4 サーバと同じマシン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`
asa_test.php	`http://localhost/asa_test.php`

すべてが正しく設定されている場合、asa_test ページに従業員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、SQLAnywhere PHP モジュールを使用して Adaptive Server 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。

これに関する例のソース・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 Adaptive Server 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 SQLAnywhere PHP module is loaded
   if( !extension_loaded('sqlanywhere')) {
     if( strtoupper(substr(PHYP_OS, 0, 3) == 'WIN' )) {
        dl('phpX_sqlanywhere9.dll');
     } else {
        dl('phpX_sqlanywhere9.so');
     }
   }

  # Connect using the default userid and password
   $conn = sqlanywhere_connect( "uid=DBA;pwd=SQL" );
   if( ! $conn ) {
       echo "Connection failed¥n";
       return 0;
   } 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 SQLAnywhere 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、Adaptive Server 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、Adaptive Server 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 userid and password
    $conn = sqlanywhere_connect( "uid=DBA;pwd=SQL" );
    if( ! $conn ) {
        echo "Connection failed¥n";
        return 0;
    } else {
        echo "Connected successfully¥n";
        sqlanywhere_disconnect( $conn );
    }
```

```
  # Execute a SELECT statement
   $result = sqlanywhere_query( $conn, "SELECT * FROM
CUSTOMER" );
   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 userid and password
   $conn = sqlanywhere_connect( "uid=DBA;pwd=SQL" );
   if( ! $conn ) {
       echo "Connection failed¥n";
       return 0;
   } else {
       echo "Connected successfully¥n";
       sqlanywhere_disconnect( $conn );
   }
```

```
  # Execute a SELECT statement
   $result = sqlanywhere_query( $conn, "SELECT * FROM
CUSTOMER" );
   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 の例は、*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 userid and password
   $conn = sqlanywhere_connect( "uid=DBA;pwd=SQL" );
   if( ! $conn ) {
       echo "Connection failed¥n";
       return 0;
   } else {
       echo "Connected successfully¥n";
       sqlanywhere_disconnect( $conn );
   }

  # Retrieve the data and output HTML
   echo "<BR>¥n";
   $query1 = "select table_id, table_name from
systable";
   $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";
       $query2 = "select table_id, column_name " +
         "from syscolumn " +
         "where table_id = '$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";
  ?>
```

上の例では、SYSTABLE から各テーブルのテーブル ID とテーブル名が選択されます。sqlanywhere_query 関数が、ローの配列を返します。スクリプトは、sqlanywhere_fetch_array を使用してそれらのローを反復して、ローのカラムを配列として取り出します。内部反復がそのローのカラムで行われ、それらの名前が出力されます。

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 asademo 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。

```
<?
   # Connect using the default userid and password
   $conn = sqlanywhere_connect( "uid=DBA;pwd=SQL" )
            or die("Can not connect to database");
```

```
  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=asademo;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

Adaptive Server Anywhere データベースには、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) として格納できる機能があります。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、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。

この例では、大きなサイズのテキストを格納するための BLOB カラムの使用方法を示しています。

この例のソース・コードは、SQL Anywhere インストール環境の *image_insert.php* という名前のファイルにあります。

```
<?
   $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ために、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。

トラブルシューティング

- 問題解決のためには、次に示す情報も参照してください。
  - **ニュースグループ sybase.public.sqlanywhere.general** と **sybase.public.sqlanywhere.linux** を中心に、Web ブラウザまたはニュースグループ・リーダで参照してください。すでに問題が解決されている場合があるので、古いスレッドを確認してください。

    ニュース・リーダを使用する場合、ニュースグループは **forums.sybase.com** から購読できます。Web の場合の URL は、http://www.sybase.com/support/newsgroups です。
  - **FAQ** UNIX バージョンの Adaptive Server Anywhere については、http://www.sybase.com/detail?id=1011965 で参照できます。
- バグ・フィックスまたは更新が公開されているかどうかについては、http://downloads.sybase.com/swx/sdmain.stm を確認してください。

SQLAnywhere PHP モジュールのインタフェース・リファレンス

この項では、PHP モジュールによって提供されるプロシージャに関するリファレンス情報を示します。

接続

- 「sqlanywhere_connect」623 ページ
- 「sqlanywhere_pconnect」628 ページ
- 「sqlanywhere_disconnect」624 ページ

クエリ

- 「sqlanywhere_query」629 ページ
- 「sqlanywhere_free_result」627 ページ

結果セット

- 「sqlanywhere_data_seek」623 ページ
- 「sqlanywhere_num_rows」628 ページ
- 「sqlanywhere_num_fields」628 ページ
- 「sqlanywhere_fetch_field」625 ページ
- 「sqlanywhere_fetch_array」624 ページ
- 「sqlanywhere_fetch_row」627 ページ
- 「sqlanywhere_fetch_object」626 ページ
- 「sqlanywhere_result_all」630 ページ

トランザクション

- 「sqlanywhere_commit」622 ページ
- 「sqlanywhere_rollback」630 ページ
- 「sqlanywhere_set_option」631 ページ

sqlanywhere_commit

プロトタイプ

bool **sqlanywhere_commit(** int *link_identifier* **)**

説明

Adaptive Server 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

```
$result = sqlanywhere_commit( $conn );
```

関連する関数

- 「sqlanywhere_rollback」630 ページ
- 「sqlanywhere_set_option」631 ページ

sqlanywhere_connect

プロトタイプ　int **sqlanywhere_connect(** string *con_str* **)**

説明　Adaptive Server Anywhere データベースとの接続を確立します。

パラメータ

- **con_str** Adaptive Server 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 Adaptive Server Anywhere リンク識別子、失敗の場合はエラーまたは 0

例

```
$conn = sqlanywhere_connect( "UID=dba;PWD=sql" );
```

関連する関数

- 「sqlanywhere_pconnect」628 ページ
- 「sqlanywhere_disconnect」624 ページ

sqlanywhere_data_seek

プロトタイプ　bool **sqlanywhere_data_seek(** int *result_identifier*, *int 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 内で移動したい位置を表す整数。たとえば、結果セットの 5 番目のローに移動するには、5 を指定します。負の数は、現在の位置よりも前に結果セットを戻ることを表します。-1 を指定すると、結果セットの最後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

```
sqlanywhere_data_seek( $result, 5 );
```

関連する関数

- 「sqlanywhere_fetch_field」625 ページ
- 「sqlanywhere_fetch_array」624 ページ
- 「sqlanywhere_fetch_row」627 ページ
- 「sqlanywhere_fetch_object」626 ページ
- 「sqlanywhere_query」629 ページ

sqlanywhere_disconnect

プロトタイプ bool **sqlanywhere_disconnect(** int *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

```
sqlanywhere_disconnect( $conn );
```

関連する関数

- 「sqlanywhere_connect」623 ページ
- 「sqlanywhere_pconnect」628 ページ

sqlanywhere_fetch_array

プロトタイプ array **sqlanywhere_fetch_array(** int *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セット

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

```
$result = sqlanywhere_query( $conn, "SELECT fname,
lname FROM employee" );
   While( ($row = sqlanywhere_fetch_array( $result )) )
{
         echo " fname = $row["fname"] ¥n" ;
         echo " lname = $row[1] ¥n";
   }
```

関連する関数

- 「sqlanywhere_data_seek」623 ページ
- 「sqlanywhere_fetch_field」625 ページ
- 「sqlanywhere_fetch_row」627 ページ
- 「sqlanywhere_fetch_object」626 ページ

sqlanywhere_fetch_field

プロトタイプ　object **sqlanywhere_fetch_field(** int *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** はフィールド長を返し、**type** はフィールドの型を返します。

例

```
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」623 ページ
- 「sqlanywhere_fetch_array」624 ページ
- 「sqlanywhere_fetch_row」627 ページ
- 「sqlanywhere_fetch_object」626 ページ

sqlanywhere_fetch_object

プロトタイプ

array **sqlanywhere_fetch_object(** int *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

```
$result = sqlanywhere_query( $conn, "SELECT fname,
lname FROM employee" );
   While( ($row = sqlanywhere_fetch_array( $result )) )
{
        echo "$row["fname"] ¥n";     # output the data
in the first column only.
   }
```

関連する関数

- 「sqlanywhere_data_seek」623 ページ
- 「sqlanywhere_fetch_field」625 ページ
- 「sqlanywhere_fetch_array」624 ページ
- 「sqlanywhere_fetch_row」627 ページ

sqlanywhere_fetch_row

プロトタイプ　array **sqlanywhere_fetch_row(** int *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

```
while( ($row = sqlanywhere_fetch_array( $result )) ) {
        echo "$row[0] ¥n";     # output the data in the
first col only.
   }
```

関連する関数

- 「sqlanywhere_data_seek」623 ページ
- 「sqlanywhere_fetch_field」625 ページ
- 「sqlanywhere_fetch_array」624 ページ
- 「sqlanywhere_fetch_object」626 ページ

sqlanywhere_free_result

プロトタイプ　bool **sqlanywhere_free_result(** int *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

```
sqlanywhere_free_result( $result );
```

関連する関数

- 「sqlanywhere_query」629 ページ

sqlanywhere_num_fields

プロトタイプ int **sqlanywhere_num_fields(** int *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

```
sqlanywhere_num_fields( $result );
```

sqlanywhere_num_rows

プロトタイプ int **sqlanywhere_num_rows(** int *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。

例

```
sqlanywhere_num_rows( $result );
```

関連する関数

- 「sqlanywhere_query」629 ページ

sqlanywhere_pconnect

プロトタイプ int **sqlanywhere_pconnect(** string *con_str* **)**

説明

Adaptive Server Anywhere データベースと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** Adaptive Server 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 ASQLAnywhere リンク識別子、失敗の場合はエラーまたは 0

例

```
$conn = sqlanywhere_pconnect( "UID=dba;PWD=sql" );
```

関連する関数

- 「sqlanywhere_connect」623 ページ
- 「sqlanywhere_disconnect」624 ページ

sqlanywhere_query

プロトタイプ

int **sqlanywhere_query(** int *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。

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** Adaptive Server 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、『Adaptive Server Anywhere SQL リファレンス・マニュアル』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

```
$result = sqlanywhere_query( $conn, "SELECT * FROM
systable" );
```

関連する関数

- 「sqlanywhere_free_result」627 ページ

sqlanywhere_result_all

プロトタイプ bool **sqlanywhere_result_all(** int *result_identifier* [, *html_table_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。

戻り値 成功した場合は TRUE、失敗した場合は FALSE

例

```
$result = sqlanywhere_query( $conn, "SELECT fname,
lname FROM employee" );
   sqlanywhere_result_all( $result );
```

関連する関数

- 「sqlanywhere_query」629 ページ

sqlanywhere_rollback

プロトタイプ bool **sqlanywhere_rollback(** int *link_identifier* **)**

説明 Adaptive Server 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。
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

```
$result = sqlanywhere_rollback( $conn );
```

関連する関数

- 「sqlanywhere_commit」622 ページ
- 「sqlanywhere_set_option」631 ページ

sqlanywhere_set_option

プロトタイプ

bool **sqlanywhere_rollback(** int *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。現在サポートされているオプションは、AUTO_COMMIT のみです。このオプションのデフォルト値は ON であり、これによってデータベース・サーバは各文の実行後にコミットします。

php.ini ファイルに次の行を追加することによって、デフォルト値を OFF に変更できます。

```
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

```
$result = sqlanywhere_set_option( $conn, "AUTO_COMMIT",
"OFF" );
```

関連する関数

- 「sqlanywhere_commit」622 ページ
- 「sqlanywhere_rollback」630 ページ

第 17 章

3 層コンピューティングと分散トランザクション

この章の内容

この章では、3 層環境においてアプリケーション・サーバとともに Adaptive Server Anywhere を使用する方法を説明します。特に、分散トランザクションの中で Adaptive Server Anywhere をエンリストする方法について説明します。

概要

Adaptive Server 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。

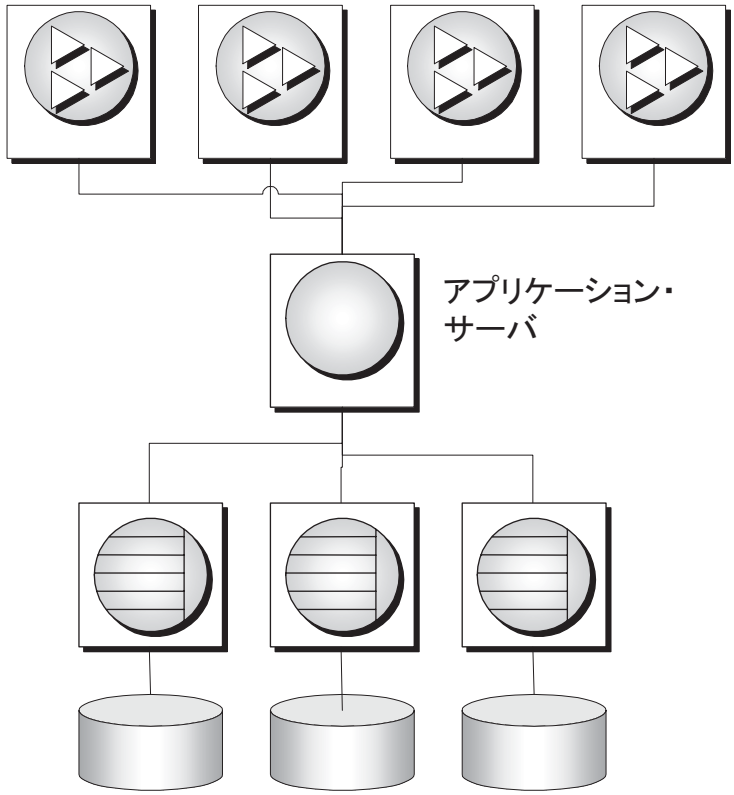
3 層環境では、クライアント・アプリケーションと一連のリソース・マネージャの間にアプリケーション・サーバを置きますが、これが一般的な分散トランザクション環境です。Sybase EAServer と他のアプリケーション・サーバの一部もトランザクション・サーバです。

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。Adaptive Server 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、Adaptive Server Anywhere を使用できます。

Adaptive Server 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、Adaptive Server 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 Adaptive Server 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 Adaptive Server 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 Adaptive Server Anywhere

DTC が調整するトランザクションに Adaptive Server 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、Adaptive Server 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、Adaptive Server 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・マシン上のリソース・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**」と通信することがあります。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。

  Adaptive Server 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。

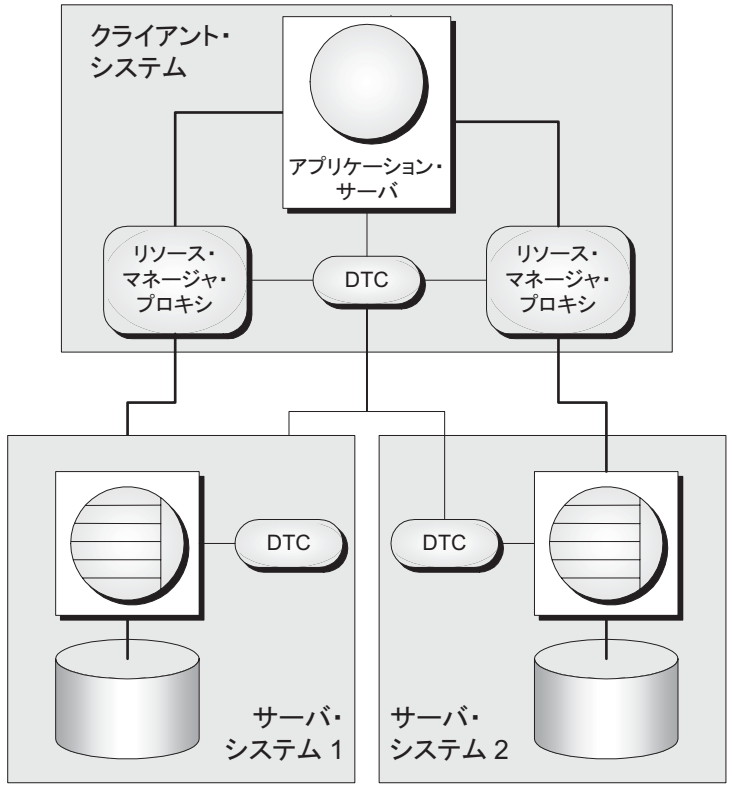
アプリケーション・サーバが DTC を使用する方法

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、各マシン上で 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] ー [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

Adaptive Server 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、Adaptive Server 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。Adaptive Server 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 Adaptive Server 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 Adaptive Server Anywhere の独立性レベルにマップされます。

DTC の独立性レベル	Adaptive Server Anywhere の独立性レベル
ISOLATIONLEVEL_UNSPECIFIED	0
ISOLATIONLEVEL_CHAOS	0
ISOLATIONLEVEL_READUNCOMMITTED	0

DTC の独立性レベル	Adaptive Server Anywhere の独立性レベル
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、Adaptive Server 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。

  詳細については、『ASA データベース管理ガイド』>「-tmf サーバ・オプション」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。

詳細については、『ASA データベース管理ガイド』> 「-tmt サーバ・オプション」を参照してください。

EAServer と Adaptive Server Anywhere の併用

この項では、EAServer 3.0 以降を Adaptive Server 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 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 Adaptive Server 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] － [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] メニューから、[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、ドロップ・ダウン・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することがあります。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、[Package] 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 Adaptive Server Anywhere の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。

第 18 章

データベースとアプリケーションの配備

この章の内容

この章では、Adaptive Server Anywhere コンポーネントを配備する方法について説明します。配備に必要なファイルを示し、接続の設定などの関連する問題を取り上げます。

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。

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 Adaptive Server 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、Adaptive Server 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 Adaptive Server 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、Adaptive Server Anywhere コンポーネントを配備する方法としては、InstallShield オブジェクトを使用するか、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、「InstallShield を使用した配備」656 ページと「サイレント・インストールを使用した配備」658 ページを参照してください。

配備モデル

配備する必要があるファイルは、選択する配備モデル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** Adaptive Server 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・ユーザのマシン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 同期サーバの配備については、『Mobile Link 管理ガイド』>「Mobile Link アプリケーションの配備」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

Adaptive Server Anywhere を配備するには、次の 2 つの方法があります。

- **Adaptive Server Anywhere インストール環境を使用する**　セットアップ・プログラムをエンド・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・ユーザが必要とするファイルを取得できます。

  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」658 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** Adaptive Server 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 Adaptive Server 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 Adaptive Server 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、デフォルトのインストール・ディレクトリが選択されている場合は、インストール・ディレクトリの *win32* サブディレクトリに、Windows オペレーティング システムのサーバを稼働するために必要なファイルが置かれています。

  また、InstallShield Professional 5.5 以降を使用している場合は、SQL Anywhere Studio InstallShield Template Projects を使用して、作成したアプリケーションを配備できます。これによって、テンプレート・プロジェクトの全部またはインストールに適用する部分だけを使用して、アプリケーションのインストール・プログラムを迅速に構築できます。

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・ライブラリ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 Adaptive Server 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、実際上は、各 PC 上の単一のディレクトリにほとんどのファイルを置いてくださいということです。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 Adaptive Server 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、『ASA データベース管理ガイド』>「Adaptive Server Anywhere のファイル検索方法」を参照してください。

UNIX の場合の配備

UNIX の場合の配備は、PC 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**　UNIX 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/sybase/SYBSsa9/bin	実行ファイル
/opt/sybase/SYBSsa9/lib	共有オブジェクトと共有ライブラリ
/opt/sybase/SYBSsa9/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/sybase/SYBSsa9* であり、*/opt/sybase/SYBSsa9* ではありません。

- **ファイル拡張子**　この章の表では、共有オブジェクトには拡張子 *.so* が付いています。HP-UX の場合、拡張子は *.sl* です。

  AIX オペレーティング・システムでは、アプリケーションのリンク先になる共有オブジェクトには、拡張子 *.a* が付けられます。

- **シンボリック・リンク**　各共有オブジェクトは、追加拡張子 *.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・リンクとしてインストールされます。たとえば、*libdblib9.so* は同じディレクトリ内のファイル *libdblib9.so.1* へのシンボリック・リンクです。

  Adaptive Server Anywhere のインストール環境にパッチが必要な場合は、それらのファイルに拡張子 *.2* を付けて、シンボリック・リンクをリダイレクトする必要があります。

- **スレッド・アプリケーションと非スレッド・アプリケーション**　ほとんどの共有オブジェクトは、2 つの形式で提供されます。その一方は、ファイル拡張子の前に追加文字 *_r* が付けられます。たとえば、*libdblib9.so* のほかに *libdblib9_r.so* 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 *_r* 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 *_r* が付いていない共有オブジェクトにリンクされる必要があります。

- **文字セット変換**　データベース・サーバの文字セット変換 (`-ct` サーバ・オプション) を使用する場合、次のファイルが必要です。

  - *libunic.so*
  - *charsets/* ディレクトリのサブツリー
  - *asa.cvf*
  - *asa.cvu*

- **環境変数**　UNIX では、システムが SQL Anywhere Studio アプリケーションとライブラリを検出できるように環境変数を設定します。必要な環境変数を設定するためのテンプレートとして、*asa_config.sh* と *asa_config.csh*（*/opt/sybase/SYBSsa9/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、ASANY9、ASANYSH9 などがあります。

Adaptive Server Anywhere がファイルを探す場所の詳細については、『ASA データベース管理ガイド』>「Adaptive Server Anywhere のファイル検索方法」を参照してください。

ファイルの命名規則

Adaptive Server 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**　Adaptive Server Anywhere のバージョン番号は、メイン・サーバ・コンポーネント (*.exe* ファイルと *.dll* ファイル) のファイル名に示されます。

  たとえば、ファイル *dbeng9.exe* はバージョン 9 の実行プログラムです。

- **言語**　言語リソース・ライブラリで使用される言語は、ファイル名の中の 2 文字のコードで示されます。バージョン番号の前の 2 文字が、ライブラリで使用されている言語を示します。たとえば、*dblgen9.dll* は英語版の言語リソース・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、『ASA データベース管理ガイド』>「ロケール言語の知識」を参照してください。

言語リソース配備 DLL を含む多言語リソース配備キットは、Sybase の Web サイトから無償でダウンロードできます。

❖ Sybase の Web サイトから多言語リソース配備キットを無償でダウンロードするには、次の手順に従います。

1　Web ブラウザで次の URL を開きます。

http://www.ianywhere.com/developer/

2 ページの左側にある [Downloads] の見出しで、[EBFs/Patches] をクリックします。

3 Sybase Web アカウントにログインします。

まだアカウントを作成していない場合には、[Create a New Account] をクリックして Sybase Web アカウントを作成します。

4 有効なダウンロード情報のリストから、現在使用しているプラットフォームと Adaptive Server Anywhere のバージョンに合う多言語リソース配備キットを選択します。

Adaptive Server Anywhere で使用できる言語リストについては、『ASA データベース管理ガイド』> 「照合の選択」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 Adaptive Server Anywhere ファイルのプラットフォームと機能を示します。Adaptive Server 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
.nlm	Novell NetWare	NetWare ロード・モジュール
.chm、*.chw*	Windows	ヘルプ・システム・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
.cfg、*.cpr*、*.dat*、*.loc*、*.spr*、*.srt*、*.xlt*	Windows	Sybase Adaptive Server Enterprise コンポーネント
.cmd、*.bat*	Windows	コマンド・ファイル
.res	NetWare、UNIX	非 Windows 環境用の言語リソース・ファイル

ファイル拡張子	プラットフォーム	ファイル・タイプ
.dll	Windows	ダイナミック・リンク・ライブラリ
.so、*.sl*、*.a*	UNIX	共有オブジェクト (Sun Solaris と IBM AIX) または共有ライブラリ (HP-UX) ファイル。Windows DLL の同等品

データベース・ファイル名

Adaptive Server 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。ファイル拡張子には *.db* を使います。
- **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。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、Adaptive Server 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。
- **ライト・ファイル**　アプリケーションがライト・ファイルを使用する場合は、通常はファイル拡張子として *.wrt* が付きます。
- **圧縮データベース・ファイル**　読み込み専用の圧縮データベース・ファイルを提供する場合は、通常は拡張子として *.cdb* を使用します。

これらのファイルは、Adaptive Server 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。

InstallShield を使用した配備

InstallShield では、インストール・コンポーネントを InstallShield プロジェクトに組み込むためにさまざまな方法が使用されています。

- InstallShield 7 以降を使用している場合には、SQL Anywhere Studio の InstallShield Merge Modules Projects を使用できます。これらのプロジェクトは、SQL Anywhere Studio のインストール環境の *deployment¥MergeModules* サブディレクトリにあります。Merge Module Projects を使用すると、現在マシンにインストールされているソフトウェアを配備するためのマージ・モジュールを生成できます。

- InstallShield 6 以降を使用している場合には、SQL Anywhere Studio の InstallShield Object Projects を使用できます。クライアント、パーソナル・データベース・サーバ、ネットワーク・サーバ、管理ツールを配備するプロジェクトは、SQL Anywhere ディレクトリの下にある *deployment¥Object* ディレクトリにあります。Object Projects を使用すると、現在マシンにインストールされているソフトウェアを配備するためのオブジェクトを生成できます。

- InstallShield Professional 5.5 以降を使用している場合は、SQL Anywhere Studio InstallShield Template Projects を使用して簡単に配備できます。ネットワーク・サーバ、パーソナル・サーバ、クライアント・インタフェース、管理ツールを配備するためのテンプレートは、*SQL Anywhere 9¥deployment¥Templates* フォルダにあります。

  メディアを作成すると、空のファイル・グループに関する警告がいくつか表示されます。これらの警告は、テンプレートにアプリケーション・ファイルのプレースホルダとして空のファイル・グループが追加されたために表示されるものです。これらの警告を削除するには、アプリケーションのファイルをそのファイル・グループに追加するか、ファイル・グループを削除するか、ファイル・グループの名前を変更します。

InstallShield 7 以降を使用している場合は、Merge Modules Projects のご使用をおすすめします。InstallShield 6 を使用している場合には、Templates ではなく Object Projects のご使用をおすすめします。Object Projects は、他のコンポーネントとともに簡単にインストールに組み込むことができます。

これらの InstallShield Projects と Templates をインストール・プログラムに組み込む方法については、InstallShield のマニュアルを参照してください。

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、Adaptive Server 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 Adaptive Server Anywhere InstallShield セットアップ・プログラムを呼び出すことができます。サイレント・インストールは Microsoft System Management Server でも使用されます (「SMS のインストール」661 ページを参照してください)。

「配備モデル」648 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 同期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 Adaptive Server 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 Adaptive Server Anywhere インストールを削除します (オプション)。
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、Adaptive Server Anywhere のセットアップ・プログラムを実行し、ユーザの選択にもと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、リブート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・システムのリブート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・ユーザのマシンに Adaptive Server 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 Adaptive Server Anywhere をインストールしてください。サイレント・インストール中に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 Adaptive Server 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、Adaptive Server 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。

インストール・イメージのディレクトリに応答ファイルがある場合、インストール・イメージの入っているディレクトリから次のコマンドを入力して、サイレント・インストールを実行できます。

```
setup -s
```

応答ファイルが別の場所に置かれている場合には、-f1 オプションを使って応答ファイルのロケーションを指定してください。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。

```
setup -s -f1"c:¥winnt¥setup.iss"
```

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( "ASA_install_image_path¥SETUP.INS",
     "-s", WAIT );
```

オプションを使って、Adaptive Server Anywhere 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
```

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リブート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リブート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** リブートは必要ありません。

- **Reboot=1**　インストール中に BATCH_INSTALL フラグが設定されました。対象コンピュータのリブートが必要です。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リブートします。

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。

セットアップによって、応答ファイルが保管されているディレクトリに *setup.log* というファイルが作成されます。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。このファイルの最後のセクションは **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。エラー・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。

SMS のインストール

Microsoft System Management Server (SMS) では、対象コンピュータをリブートしないサイレント・インストールが必要です。Adaptive Server Anywhere のサイレント・インストールは、コンピュータをリブートしません。

SMS 配布パッケージには、応答ファイル、インストール・イメージ、*asa9.pdf* パッケージ定義ファイルが入っています (Adaptive Server Anywhere の CD-ROM の *¥extras* フォルダに入っています)。PDF ファイルのセットアップ・コマンドには、次のオプションが入っています。

- `-s` オプション。サイレント・インストールで使用します。
- `-SMS` オプション。SMS に呼び出されていることを示します。
- `-m` オプション。MIF ファイルを生成します。MIF ファイルを使って、SMS はインストールが正常に終了したかどうかを判断します。

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 (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、提供された InstallShield のオブジェクトを使用することです。詳細については、「InstallShield を使用した配備」656 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、InstallShield のオブジェクトまたはテンプレートを使用することです。詳細については、「InstallShield を使用した配備」656 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・マシンには、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 の再配布情報から入手できます。ここでは、その詳細は説明しません。

- **Adaptive Server Anywhere OLE DB プロバイダ**　次の表には、Adaptive Server 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。Adaptive Server 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*)。

説明	Windows	Windows CE
OLE DB ドライバ・ファイル	*dboledb9.dll*	*dboledb9.dll*
OLE DB ドライバ・ファイル	*dboledba9.dll*	*dboledba9.dll*
言語リソース・ライブラリ	*dblgen9.dll*	*dblgen9.dll*
[接続] ダイアログ	*dbcon9.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。

詳細については、『ASA データベース管理ガイド』>「Windows CE 用データベースの作成」と「Windows CE での ODBC アプリケーションのリンク」284 ページを参照してください。

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、InstallShield のオブジェクトまたはテンプレートを使用することです。詳細については、「InstallShield を使用した配備」656 ページを参照してください。

各 ODBC クライアント・マシンには、次のものが必要です。

- **ODBC が動作するインストール環境**　ODBC 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 の再配布情報から入手できます。ここでは、その詳細は説明しません。

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 Studio 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。ただし、ODBC ドライバ・マネージャを利用できるプラットフォームでは、なるべく ODBC ドライバ・マネージャを使用してください。

必要に応じた ODBC の更新

SQL Anywhere セットアップ・プログラムは、ODBC を含む Microsoft Data Access Components の古いインストール環境を更新します。独自のアプリケーションを配備する場合には、ODBC のインストール環境が自分のアプリケーションに十分対応するかどうか確認してください。

- **Adaptive Server Anywhere の ODBC ドライバ** ドライバは、*dbodbc9.dll* ファイルといくつかの追加ファイルで構成されています。

  詳細については、「ODBC ドライバに必要なファイル」664 ページを参照してください。

- **接続情報** クライアント・アプリケーションが、サーバの接続に必要な情報にアクセスできるようにしてください。この情報は、通常は ODBC データ・ソースに含まれています。

ODBC ドライバに必要なファイル

次の表は、Adaptive Server 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。Adaptive Server 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*)。

説明	Windows	Windows CE	UNIX
ODBC ドライバ	*dbodbc9.dll*	*dbodbc9.dll*	*libdbodbc9.so* *libdbtasks9.so*
言語リソース・ライブラリ	*dblgen9.dll*	*dblgen9.dll*	*dblgen9.res*
[接続] ダイアログ	*dbcon9.dll*	*なし*	*なし*

注意

- エンド・ユーザは、ドライバ・マネージャを含む、動作している ODBC インストール環境が必要です。ODBC を配備するための手順は、Microsoft の『ODBC SDK』に含まれています。
- エンド・ユーザが独自のデータ・ソースを作成する必要がある場合や、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、*libdbodbc9_r.so* および *libdbtasks9_r.so* を参照してください。

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

Adaptive Server Anywhere のセットアップ・プログラムは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・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。

Adaptive Server 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¥
                    Adaptive Server Anywhere 9.0
```

値は次のとおりです。

値の名前	値のタイプ	値データ
ドライバ	文字列	*path¥dbodbc9.dll*
Setup	文字列	*path¥dbodbc9.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
Adaptive Server Anywhere 9.0	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、各自のマシンに適切なデータ・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。Adaptive Server 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、ASA 9.0 Sample データ・ソースに対応する ASA 9.0 Sample 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	*Path*¥asademo.db
Description	文字列	Adaptive Server Anywhere Sample Database
Driver	文字列	*Path*¥win32¥dbodbc9.dll
PWD	文字列	sql
Start	文字列	*Path*¥win32¥dbeng9.exe -c 8m
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、マシンで複数の Adaptive Server 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。

これらのエントリ内の *path* は Adaptive Server Anywhere のインストール・ディレクトリです。

さらに、データ・ソース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、Adaptive Server Anywhere によってインストールされたユーザ・データ・ソースは、ASA 9.0 Sample 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
ASA 9.0 Sample	文字列	Adaptive Server Anywhere 9.0

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、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のみを含む	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、オプションとしてユーザ ID とパスワード	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)
存在しない	使用する ODBC ドライバの名前を次の書式で指定 `Driver={ODBC-drivername}` さらに、データベースの名前、データベース・ファイルまたはデータベース・サーバ、オプションとしてユーザ ID とパスワードなどの接続パラメータを指定	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。

- 『ASA データベース管理ガイド』> 「データベースへの接続」

- 『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、InstallShield のオブジェクトまたはテンプレートを使用することです。詳細については、「InstallShield を使用した配備」656 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・マシンには、Adaptive Server 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	*dblib9.dll*	*libdblib9.so*、*libdbtasks9.so*
言語リソース・ライブラリ	*dblgen9.dll*	*dblgen9.res*
[接続] ダイアログ	*dbcon9.dll*	*なし*

注意

- クライアントがパーソナル・データベース・サーバだけで動作する場合は、ネットワーク・ポート DLL は必要ありません。

- 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」663 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・ソースを作成する予定がある場合や、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、*libdblib9_r.so* および *libdbtasks9_r.so* を参照してください。

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、各自のマシンに適切なデータ・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。この場合は、Microsoft から入手できる再配布可能な ODBC ファイルのサブセットが必要です。詳細については、「ODBC クライアントの配備」663 ページを参照してください。

- **ハード・エンコード**　接続情報をアプリケーション内にハード・エンコードできます。これは柔軟性のない方法であり、たとえばデータベースを更新する場合に制限になることがあります。

JDBC クライアントの配備

Java Runtime Environment に加えて、各 JDBC クライアントには jConnect または iAnywhere JDBC ドライバが必要です。

jConnect のマニュアルについては、http://sybooks.sybase.com/jc.html を参照してください。

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。
- *dbjodbc9.dll* システム・パス内に含めます。UNIX または Linux 環境では、このファイルは共用ライブラリ (*dbjodbc9.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」664 ページ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、使用するマシン、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、「サーバの URL の指定」141 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・マシンに 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 『ASA データベース管理ガイド』> 「Open Server の設定」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、Adaptive Server 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、提供された InstallShield マージ・モジュールまたはオブジェクトを使用することです。詳細については、「InstallShield を使用した配備」656 ページを参照してください。

Interactive SQL の配備

リソースに制限のあるマシンでカスタマ・アプリケーションを実行している場合には、Interactive SQL ではなく *dbisqlc.exe* の配備が必要になることもあります。しかし、*dbisqlc* は制限付きアプリケーションです。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。

管理ツールのシステム稼働条件については、『SQL Anywhere Studio の紹介』>「SQL Anywhere Studio がサポートしているプラットフォーム」を参照してください。

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 マシンで InstallShield を使用せずに Interactive SQL (dbisql)、Sybase Central (Adaptive Server Anywhere と Mobile Link のプラグインを含む)、および Adaptive Server 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 9.0.2 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 Adaptive Server 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
- Adaptive Server Anywhere プラグインを含む Sybase Central

- Mobile Link プラグインを含む Sybase Central
- Adaptive Server Anywhere コンソール・ユーティリティ (DBConsole)

どのソフトウェア・バンドルをインストールする場合も以下のコンポーネントが必要です。

- Java Runtime Environment (JRE) バージョン 1.4.2_02
- jConnect ランタイム・ライブラリ
- iAnywhere JDBC ドライバと Adaptive Server Anywhere ODBC ドライバ

以下の項の手順は、これら 4 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が、以下のレイアウトにしてください。

ディレクトリ	説明
SA9	ルート・フォルダ。このドキュメントでは、*C:¥SA9* 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¥SA9*)。
SA9¥java	プログラム JAR ファイルが保存されています。
SA9¥win32	プログラムで使用するネイティブ (Win32) 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。
SA9¥Shared	このツリーには、異なるアプリケーション間で共有するファイルが含まれます。

ディレクトリ	説明
SA9¥Shared¥java	共有ユーティリティ・ライブラリ。
SA9¥Shared¥jConnect-5_5	jConnect 5_5 ランタイム・ライブラリ。
SA9¥Shared¥Sun¥JavaHelp-1_1	JavaHelp ランタイム・ライブラリ。
SA9¥Shared¥win32	共有ユーティリティ・ライブラリ。

Itanium 64
このドキュメントでは、32 ビットの Windows プラットフォームでのインストールを想定しています。管理ツールを 64 ビットの Windows XP Itanium マシンにインストールする場合は、64 ビット・バージョンのソフトウェアをインストールしてください。具体的には、Java Runtime Environment の 64 ビット・バージョンと、すべての *.exe* および *.dll* ファイルの 64 ビット・バージョンをインストールしてください。通常、これらのファイルの 32 ビット・バージョンは、*win32* というフォルダに、また 64 ビット・バージョンは *ia64* というフォルダにあります。*.jar* 拡張子の付いたファイルは、32 ビットと 64 ビットの両方のマシンで使用できます。

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 Studio 9.0.2 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	Adaptive Server Anywhere プラグイン	Mobile Link プラグイン	Interactive SQL	Adaptive Server Anywhere コンソール
java¥jcchart400k.jar	X	X		
java¥jlogon.jar	X	X	X	X
java¥jodbc.jar	X	X	X	X

ファイル	Adaptive Server Anywhere プラグイン	Mobile Link プラグイン	Interactive SQL	Adaptive Server Anywhere コンソール
java¥sybasecentral.jar	X	X		
java¥ASA.jar	X			
java¥asaplugin.jar	X			
java¥debugger.jar	X			
java¥dbconsole.jar				X
java¥isql.jar	X		X	
java¥mlplugin.jar		X		
java¥xerces.jar	X		X	
java¥xml4j.jar	X		X	
java¥apache_files.txt	X		X	
java¥apache_license.txt	X		X	
java¥dbmaen9.jar	X	X	X	X
java¥dbmaen9.chm	X	X	X	X
java¥scjview.exe	X	X		
java¥scjview.cls	X	X		
java¥scjlgen.dll	X	X		
java¥scvwen43.chm	X	X		
java¥scvwen43.jar	X	X		
Shared¥java¥HelpManager11.jar	X	X	X	X
Shared¥java¥JComponents142.jar	X		X	X

ファイル	Adaptive Server Anywhere プラグイン	Mobile Link プラグイン	Interactive SQL	Adaptive Server Anywhere コンソール
Shared¥java¥jsyblib142.jar	X	X	X	X
Shared¥java¥SCEditor142.jar	X	X	X	
Shared¥jConnect-5_5¥classes¥jconn2.jar	X	X	X	X
Shared¥Sun¥jre142¥.** [1]	X	X	X	X
Shared¥Sun¥JavaHelp-1_1¥jh.jar	X	X	X	X
Shared¥win32¥jsyblib142.dll	X	X	X	X
win32¥dbcon9.dll	X	X	X	X
win32¥dbconsole.exe				X
win32¥dbisql.exe			X	
win32¥dbisqlg.exe			X	
win32¥dbisql.cls			X	
win32¥dbjodbc9.dll	X	X	X	X
win32¥dblgen9.dll	X		X	X
win32¥dblib9.dll	X	X	X	X
win32¥dbmlput9.dll		X		
win32¥dbodbc9.dll	X	X	X	X
win32¥dbput9.dll	X			
win32¥dbtool9.dll	X			
c:¥windows¥system32¥keyhh.exe [2]	X	X	X	X

1. 管理ツールには、JRE 1.4.2_02 が必要です。特に必要のない限り、これより新しいパッチ・バージョンの JRE (1.4.2_03 など) を代用しないでください。インストールされた Adaptive Server Anywhere 9.0.2 のコピーの JRE ファイルを、*C:¥Program Files¥Sybase¥Shared¥Sun¥jre142* 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 *jre142* ツリー全体をコピーします。
2. 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。

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。

前述した手順では、英語のファイルのみがリストされています。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。

dbmaxx9.jar - SQL Anywhere Help Files

```
dbmaen9.jar   English
dbmafr9.jar   French
dbmade9.jar   German
dbmaja9.jar   Japanese
dbmazh9.jar   Chinese
```

dblgxx9.res - Adaptive Server Anywhere Messages Files

```
dblgen9.res   English
dblgfr9.res   French
dblgde9.res   German
dblgja9.res   Japanese
dblgzh9.res   Chinese
```

scvwen43.jar - Sybase Central Help Files

```
scvwen43.jar  English
scvwfr43.jar  French
scvwde43.jar  German
scvwja43.jar  Japanese
scvwzh43.jar  Chinese
```

これらのファイルは、SQL Anywhere Studio 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。

- *HKLM¥SOFTWARE¥Sybase¥Sybase Central¥4.3* 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。
    - **Shared Location** *Shared* フォルダへの完全に修飾されたパス。
    - **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。
- *HKLM¥SOFTWARE¥Sybase¥Adaptive Server Anywhere¥9.0* では次のように設定します。
    - **Location** インストール・フォルダのルートへの完全に修飾されたパス。
    - **Shared Location** *shared* フォルダへの完全に修飾されたパス。

    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。次に、*.reg* ファイルの例を示します。

```
Windows Registry Editor Version 5.00

 [HKEY_LOCAL_MACHINE¥¥SOFTWARE¥¥Sybase¥¥Sybase
Central¥¥4.3]
 Location=C:¥¥SA9¥¥java
 Shared Location=C:¥¥SA9¥¥Shared
 Language=EN
```

```
 [HKEY_LOCAL_MACHINE¥SOFTWARE¥Sybase¥Adaptive Server
Anywhere¥¥9.0]
 Location=c:¥¥SA9
 Shared Location=c:¥¥SA9¥¥shared
```

.reg ファイルでは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。*SA9¥win32*、*SA9¥java*、および *SA9¥Shared¥win32* ディレクトリをシステム・パスに含めてください。

Windows NT/2000/XP 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。

```
HKLM¥
   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:¥SA9¥win32;c:¥SA9¥java;C:¥SA9¥Shared¥win32
```

手順 5：プラグインを登録する

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。

ファイルは *.scRepository* と呼ばれます。このファイルは、*scjview.exe* と同じ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。

Adaptive Server Anywhere と Mobile Link のプラグインの情報を含む *.scRepository* ファイルの例を次に示します。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 行にしてください。

```
SCRepositoryInfo/Version=4
 #
 Providers/asa90/Name=Adaptive Server Anywhere 9
Providers/asa90/Classpath=C:¥¥sa9¥¥java¥¥asaplugin.jar
 Providers/asa90/UseClassloader=true
 Providers/asa90/AdditionalClasspath=
 C:¥¥sa9¥¥Shared¥¥jConnect-5_5¥¥classes¥¥jconn2.jar;
 C:¥¥sa9¥¥java¥¥isql.jar;
 C:¥¥SA9¥¥java¥¥asa.jar;
 C:¥¥SA9¥¥shared¥¥java¥¥JComponents142.jar;
 C:¥¥SA9¥¥java¥¥xerces.jar;
 C:¥¥SA9¥¥java¥¥xml4j.jar;
 C:¥¥SA9¥¥java¥¥jlogon.jar;
 C:¥¥SA9¥¥java¥¥debugger.jar;
 C:¥¥SA9¥¥java¥¥jodbc.jar
 Providers/asa90/ProviderId=asa90
 Providers/asa90/Version=9.0.2.0000
 Providers/asa90/
Provider=com.sybase.asa.plugin.ASAPlugin

#
 Providers/mobilink90/Name=MobiLink Synchronization 9
 Providers/mobilink90/
Classpath=C:¥¥SA9¥¥java¥¥mlplugin.jar
 Providers/mobilink90/UseClassloader=true
 Providers/mobilink90/AdditionalClasspath=
 C:¥¥SA9¥¥Shared¥¥jConnect-5_5¥¥classes¥¥jconn2.jar;
 C:¥¥SA9¥¥java¥¥asa.jar;
 C:¥¥SA9¥¥java¥¥jlogon.jar;
 C:¥¥SA9¥¥java¥¥jodbc.jar
 Providers/mobilink90/Version=9.0.2.0000
 Providers/mobilink90/ProviderId=mobilink90
 Providers/mobilink90/
Provider=com.sybase.mobilink.plugin.MLPlugin
```

注意

- AdditionalClasspath の行は、*.scRepository* ファイルでは 1 行に入力してください。
- 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。
- 次の 7 行は、Adaptive Server Anywhere プラグインの情報です。これらの行の後に別のコメント行が続き、その後 Mobile Link プラグインの情報があります。インストーラによってこれに類似したファイルが書き出されます。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。
- ファイルには、配備するプラグインについての情報のみを含めます。たとえば、Mobile Link プラグインを配備しない場合は、ファイルの最後のセクション (`Providers/mobilink90` で始まる行) は含めません。

手順 6：Adaptive Server Anywhere ODBC ドライバを登録する

Adaptive Server 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」665 ページを参照してください。

手順 7：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。

すべての管理ツールに対して、*HKLM¥SOFTWARE¥Microsoft¥Windows¥HTML Help¥dbmaen9.chm* という文字列値を定義する必要があります。値は *C:¥SA9¥java¥dbmaen9.chm* です。

Sybase Central ヘルプ・ファイルについては、*HKLM¥SOFTWARE¥Microsoft¥Windows¥HTML Help¥scvwen43.chm* という文字列値も定義してください。値は *C:¥SA9¥java¥scvwen43.chm* です。

翻訳されたオンライン・ヘルプ・ファイルをインストールする場合は、それらも登録してください。英語のヘルプ・ファイルについて前述した手順に従いますが、翻訳されたファイルの名前を使用します。

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、古いマシン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、*C:¥Windows* または *C:¥WINNT* フォルダ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」647 ページを参照してください。

Linux と Solaris における管理ツールの配備

この項では、Linux と Solaris のマシンで Interactive SQL (dbisql)、Sybase Central (Adaptive Server Anywhere と Mobile Link のプラグインを含む)、および Adaptive Server 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 9.0.2 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

始める前に、プログラム・ファイルのソースとして SQL Anywhere Studio を 1 台のマシン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
- Adaptive Server Anywhere プラグインを含む Sybase Central
- Mobile Link プラグインを含む Sybase Central
- Adaptive Server Anywhere コンソール・ユーティリティ (DBConsole)

これらのすべてのバンドルには以下のコンポーネントが必要です。

- Java Runtime Environment (JRE) バージョン 1.4.2_02
- jConnect ランタイム・ライブラリ
- iAnywhere JDBC ドライバと Adaptive Server Anywhere ODBC ドライバ

以下の項の手順は、これら 4 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 Studio インストーラによってインストールされたファイルのサブセットをコピーします。同じディレクトリ構造を維持してください。

参照用の SQL Anywhere Studio インストール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。

基本コンポーネント・ファイル

4 つのすべてのバンドルはこの項にリストされたファイルとリンクを必要とします。

参照インストールから以下のファイルをコピーします。

```
/opt/sybase/SYBSsa9/java/jodbc.jar
 /opt/sybase/SYBSsa9/lib/libdbjodbc9.so.1
 /opt/sybase/SYBSsa9/lib/libdbodbc9_r.so.1
 /opt/sybase/SYBSsa9/lib/libdbodm9_r.so.1
 /opt/sybase/SYBSsa9/lib/libdbtasks9_r.so.1
```

以下のディレクトリをコピーします。

```
/opt/sybase/shared/jre_1.4.2_linux_sun_i386 (Linux)
 /opt/sybase/shared/jre_1.4.2_solaris_sun_sparc
(Solaris)
 /opt/sybase/shared/jConnect-5_5
```

/opt/SYBSsa9/lib に次のシンボリック・リンクを作成します。

```
libdbjodbc9.so -> libdbjodbc9.so.1
 libdbodbc9_r.so -> libdbodbc9_r.so.1
 libdbodm9_r.so -> libdbodm9_r.so.1
 libdbtasks9_r.so -> libdbtasks9_r.so.1
```

/opt/Sybase/SYBSsa9 に次のシンボリック・リンクを作成します。

```
jre142 -> /opt/sybase/shared/jre_1.4.2_linux_sun_i386
(Linux)
 jre142 -> /opt/sybase/shared/
jre_1.4.2_solaris_sun_sparc (Solaris)
 shared -> /opt/sybase/shared
```

Interactive SQL ファイル

参照インストールから以下のファイルをコピーします。

```
/opt/sybase/SYBSsa9/bin/dbisql
 /opt/sybase/SYBSsa9/java/dbmaen9.jar
 /opt/sybase/SYBSsa9/java/isql.jar
 /opt/sybase/SYBSsa9/java/jlogon.jar
 /opt/sybase/SYBSsa9/java/xerces.jar
 /opt/sybase/SYBSsa9/java/xml4j.jar

/opt/sybase/SYBSsa9/lib/libdblib9_r.so.1
 /opt/sybase/SYBSsa9/lib/libdbtool9_r.so.1
 /opt/sybase/SYBSsa9/res/asa.cvf
 /opt/sybase/SYBSsa9/res/dblgen9.res
 /opt/sybase/shared/java/HelpManager11.jar
 /opt/sybase/shared/java/JComponents142.jar
 /opt/sybase/shared/java/SCEditor142.jar
 /opt/sybase/shared/java/jsyblib142.jar
 /opt/sybase/shared/sun/javahelp-1_1/jh.jar
```

/opt/SYBSsa9/lib に次のシンボリック・リンクを作成します。

```
libdblib9_r.so -> libdblib9_r.so.1
 libdbtool9_r.so -> libdbtool9_r.so.1
```

Adaptive Server Anywhere プラグインを含む Sybase Central

参照インストールから以下のファイルをコピーします。

```
/opt/sybase/SYBSsa9/bin/dbisql
 /opt/sybase/SYBSsa9/java/asa.jar
 /opt/sybase/SYBSsa9/java/asaplugin.jar
 /opt/sybase/SYBSsa9/java/dbmaen9.jar
 /opt/sybase/SYBSsa9/java/debugger.jar
 /opt/sybase/SYBSsa9/java/isql.jar
 /opt/sybase/SYBSsa9/java/jlogon.jar
 /opt/sybase/SYBSsa9/java/jodbc.jar

/opt/sybase/SYBSsa9/java/xerces.jar
 /opt/sybase/SYBSsa9/java/xml4j.jar
 /opt/sybase/SYBSsa9/lib/libdblib9_r.so.1
 /opt/sybase/SYBSsa9/lib/libdbput9_r.so.1
 /opt/sybase/SYBSsa9/lib/libdbtool9_r.so.1
 /opt/sybase/SYBSsa9/res/asa.cvf
 /opt/sybase/SYBSsa9/res/dblgen9.res
 /opt/sybase/shared/java/HelpManager11.jar
```

```
/opt/sybase/shared/java/JComponents142.jar
 /opt/sybase/shared/java/SCEditor142.jar
 /opt/sybase/shared/java/jsyblib142.jar
 /opt/sybase/shared/sun/javahelp-1_1/jh.jar
 /opt/sybase/shared/sybcentral43/jcchart400K.jar
 /opt/sybase/shared/sybcentral43/scjview
 /opt/sybase/shared/sybcentral43/scvwen43.jar
 /opt/sybase/shared/sybcentral43/sybasecentral.jar
```

/opt/SYBSsa9/lib に次のシンボリック・リンクを作成します。

```
libdblib9_r.so -> libdblib9_r.so.1
 libdbput9_r.so -> libdbput9_r.so.1
 libdbtool9_r.so -> libdbtool9_r.so.1
```

/opt/sybase/shared/sybcentral43 に以下のリンクを作成します。

```
jre142 -> /opt/sybase/shared/jre_1.4.2_linux_sun_i386
(Linux)
 jre142 -> /opt/sybase/shared/
jre_1.4.2_solaris_sun_sparc (Solaris)
```

Mobile Link プラグインを含む Sybase Central

参照インストールから以下のファイルをコピーします。

```
/opt/sybase/SYBSsa9/java/asa.jar
 /opt/sybase/SYBSsa9/java/dbmaen9.jar
 /opt/sybase/SYBSsa9/java/jlogon.jar
 /opt/sybase/SYBSsa9/java/jodbc.jar
 /opt/sybase/SYBSsa9/java/mlplugin.jar
 /opt/sybase/SYBSsa9/lib/libdblib9_r.so.1
 /opt/sybase/SYBSsa9/lib/libdbmlput9_r.so.1
 /opt/sybase/SYBSsa9/lib/libdbtool9_r.so.1

/opt/sybase/shared/java/HelpManager11.jar
 /opt/sybase/shared/java/JComponents142.jar
 /opt/sybase/shared/java/SCEditor142.jar
 /opt/sybase/shared/java/jsyblib142.jar
 /opt/sybase/shared/sun/javahelp-1_1/jh.jar
 /opt/sybase/shared/sybcentral43/jcchart400K.jar
 /opt/sybase/shared/sybcentral43/scjview
 /opt/sybase/shared/sybcentral43/scvwen43.jar
 /opt/sybase/shared/sybcentral43/sybasecentral.jar
```

このディレクトリ全体をコピーします。

```
/opt/sybase/SYBSsa9/drivers
```

/opt/sybase/SYBSsa9/lib に次のシンボリック・リンクを作成します。

```
libdblib9_r.so -> libdblib9_r.so.1
 libdbmlput9_r.so -> libdbmlput9_r.so.1
 libdbtool9_r.so -> libdbtool9_r.so.1
```

/opt/sybase/shared/sybcentral43 に次のシンボリック・リンクを作成します。

```
jre142 -> /opt/sybase/shared/jre_1.4.2_linux_sun_i386
(Linux)
 jre142 -> /opt/sybase/shared/
jre_1.4.2_solaris_sun_sparc (Solaris)
```

Adaptive Server Anywhere コンソール

参照インストールから以下のファイルをコピーします。

```
/opt/sybase/SYBSsa9/bin/dbconsole
 /opt/sybase/SYBSsa9/java/DBConsole.jar
 /opt/sybase/SYBSsa9/java/dbmaen9.jar
 /opt/sybase/SYBSsa9/java/jlogon.jar
 /opt/sybase/SYBSsa9/java/jodbc.jar
 /opt/sybase/SYBSsa9/lib/libdblib9_r.so.1
 /opt/sybase/SYBSsa9/lib/libdbtool9_r.so.1

/opt/sybase/SYBSsa9/res/asa.cvf
 /opt/sybase/SYBSsa9/res/dblgen9.res
 /opt/sybase/shared/java/HelpManager11.jar
 /opt/sybase/shared/java/JComponents142.jar
 /opt/sybase/shared/java/SCEditor142.jar
 /opt/sybase/shared/java/jsyblib142.jar
 /opt/sybase/shared/sun/javahelp-1_1/jh.jar
```

/opt/sybase/SYBSsa9/lib に次のシンボリック・リンクを作成します。

```
libdblib9_r.so   -> libdblib9_r.so.1
 libdbtool9_r.so  -> libdbtool9_r.so.1
```

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。

前述した手順では、英語のファイルのみがリストされています。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。

dbmaxx9.jar - SQL Anywhere Help Files

```
dbmaen9.jar   English
dbmafr9.jar   French
dbmade9.jar   German
dbmaja9.jar   Japanese
dbmazh9.jar   Chinese
```

dblgxx9.res - Adaptive Server Anywhere Messages Files

```
dblgen9.res   English
dblgfr9.res   French
dblgde9.res   German
dblgja9.res   Japanese
dblgzh9.res   Chinese
```

scvwen43.jar - Sybase Central Help Files

```
scvwen43.jar  English
scvwfr43.jar  French
scvwde43.jar  German
scvwja43.jar  Japanese
scvwzh43.jar  Chinese
```

これらのファイルは、SQL Anywhere Studio 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 Studio インストーラによって作成された *asa_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/Sybase/SYBsa9/bin
   ```

   Sybase Central の配備では、以下も追加してください。

   ```
   /opt/sybase/shared/sybcentral43
   ```

2. 以下を含むように LD_LIBRARY_PATH を設定します。

   ```
   /opt/sybase/SYBsa9/jre142/lib/i386/client
    /opt/sybase/SYBsa9/jre142/lib/i386
    /opt/sybase/SYBsa9/jre142/lib/i386/native_threads
   ```

   Mobile Link プラグインの配備では、以下も含めてください。

   ```
   /opt/sybase/SYBsa9/drivers
   ```

3. 以下の環境変数を設定します。

   ```
   ASANY9="/opt/sybase/SYBsa9"
    ASANYSH="/opt/sybase/shared"
   ```

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を使用します。ファイルは *.scRepository* と呼ばれます。このファイルは *scjview*（*/opt/sybase/shared/sybcentral43*) と同じディレクトリにあります。

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。

Adaptive Server Anywhere と Mobile Link のプラグインの情報を含む *.scRepository* ファイルの例を次に示します。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。*.scRepository* ファイルを作成する場合、各エントリは 1 行に入力してください。

```
SCRepositoryInfo/Version=4
 #
 Providers/asa90/Version=9.0.2.0000
 Providers/asa90/UseClassloader=true
 Providers/asa90/
Classpath=¥_opt¥_sybase¥_SYBSsa9¥_java¥_asaplugin.jar
 Providers/asa90/Name=Adaptive Server Anywhere 9
 Providers/asa90/AdditionalClasspath=
   ¥_opt¥_sybase¥_SYBSsa9¥_java¥_isql.jar:
   ¥_opt¥_sybase¥_SYBSsa9¥_java¥_asa.jar:
   ¥_opt¥_sybase¥_SYBSsa9¥_java¥_xerces.jar:
   ¥_opt¥_sybase¥_SYBSsa9¥_java¥_xml4j.jar:
   ¥_opt¥_sybase¥_SYBSsa9¥_java¥_jlong.jar:
   ¥_opt¥_sybase¥_SYBSsa9¥_java¥_debugger.jar:
   ¥_opt¥_sybase¥_SYBSsa9¥_java¥_jlogon.jar:
   ¥_opt¥_sybase¥_SYBSsa9¥_java¥_jodbc.jar:
   ¥_opt¥_sybase¥_shared¥_java¥_JComponents142.jar:
   ¥_opt¥_sybase¥_shared¥_jConnect-
5_5¥_classes¥_jconn2.jar
 Providers/asa90/
Provider=com.sybase.asa.plugin.ASAPlugin
 Providers/asa90/ProviderId=asa90
 Providers/asa90/InitialLoadOrder=0

#
 Providers/mobilink90/Version=9.0.2.0000
 Providers/mobilink90/UseClassloader=true
 Providers/mobilink90/
Classpath=¥_opt¥_sybase¥_SYBSsa9¥_java¥_mlplugin.jar
 Providers/mobilink90/Name=MobiLink Synchronization 9
 Providers/mobilink90/AdditionalClasspath=
   ¥_opt¥_sybase¥_SYBSsa9¥_java¥_asa.jar:
   ¥_opt¥_sybase¥_SYBSsa9¥_java¥_jlogon.jar:
   ¥_opt¥_sybase¥_SYBSsa9¥_java¥_jodbc.jar:
   ¥_opt¥_sybase¥_shared¥_jConnect-
```

```
5_5¥_classes¥_jconn2.jar
 Providers/mobilink90/
Provider=com.sybase.mobilink.plugin.MLPlugin
 Providers/mobilink90/ProviderId=mobilink90
 Providers/mobilink90/InitialLoadOrder=0
```

注意

- インストーラによってこれに類似したファイルが書き出されます。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。
- 最初のコメント行の後の 8 行は、Adaptive Server Anywhere プラグインの情報です。これらの行の後に別のコメント行が続き、その後 Mobile Link プラグインの情報があります。
- ファイルには、配備するプラグインのみに関する情報を含めます。たとえば、Mobile Link プラグインを配備しない場合は、ファイルの最後のセクション (`Providers/mobilink90` で始まる行) は含めません。

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」647 ページを参照してください。

データベース・サーバの配備

データベース・サーバは SQL Anywhere Studio のセットアップ・プログラムをエンド・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、提供されている InstallShield のオブジェクトを使用することです。詳細については、「InstallShield を使用した配備」656 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	UNIX	NetWare
dbeng9.exe	*dbeng9*	なし
dbsrv9.exe	*dbsrv9*	*dbsrv9.nlm*
dbserv9.dll	*libdbserv9.so*、*libdbtasks9_r.so*	なし
dblgen9.dll	*dblgen9.res*	*dblgen9.res*
dbjava9.dll (1)	*libdbjava9.so* (1)	*dbjava9.nlm* (1)
dbctrs9.dll	なし	なし
dbextf.dll (2)	*libdbextf.so* (2)	*dbextf.nlm* (2)
asajdbc.zip (1,3)	*asajdbc.zip* (1,3)	*asajdbc.zip* (1,3)
asajrt12.zip (1,3)	*asajrt12.zip* (1,3)	*asajrt12.zip* (1,3)
classes.zip (1,3)	*classes.zip* (1,3)	*classes.zip* (1,3)
dbmem.vxd (4)	なし	なし
dbunic9.dll	*libunic.so*	なし

Windows	UNIX	NetWare
asa.cvf	*asa.cvf*	*asa.cvf*
charsets¥ directory	*charsets/ directory*	なし

1. データベースで Java を使用する場合のみ必要です。JDK 1.1 を使用して初期化したデータベースには asajdbc.zip を配布してください。JDK 1.2 または JDK 1.3 を使用して初期化したデータベースには asajrt13.zip を配布してください。

2. 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。

3. CLASSPATH 環境変数がこのファイル内のクラスを見つけることができるようにインストールします。

4. Windows 95/98/Me で動的キャッシュ・サイズ決定を使用している場合に必要です。

レジストリ・エントリ

Windows NT/2000/XP でデータベース・サーバを配備する場合は、イベント・ログに書き込まれるメッセージが正しくフォーマットされるように、次のレジストリ・キーを追加します。

```
HKEY_LOCAL_MACHINE¥
     SYSTEM¥
         CurrentControlSet¥
             Services¥
           Eventlog¥
       Application¥
        ASA 9.0¥
            ODBC Data Sources
```

このキー内で、EventMessageFile という REG_SZ 値を追加し、*dblgen.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¥Sybase¥SQL Anywhere 9¥win32¥dblgen9.dll*)。英語の DLL である *dblgen.dll* は、配備の言語にかかわらず指定できます。

注意

- 場合によって、パーソナル・データベース・サーバ (*dbeng9*) またはネットワーク・データベース・サーバ (*dbsrv9*) を配備するかどうか選択してください。

- Java DLL (*dbjava9.dll*) が必要になるのは、データベース・サーバがデータベース機能で Java を使用する場合だけです。
- 表には、*dbbackup* など、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」674 ページを参照してください。
- zip ファイルはデータベースで Java を使用するアプリケーションについてのみ必要であり、ユーザの CLASSPATH 環境変数で指定されるロケーションにインストールする必要があります。

データベースの配備

データベース・ファイルは、エンド・ユーザのディスクにインストールすることによって配備します。

データベース・サーバが正常に停止するかぎりは、データベース・ファイルとともにトランザクション・ログ・ファイルを配備する必要はありません。エンド・ユーザがデータベースの実行を開始するときに、新しいトランザクション・ログが作成されます。

SQL Remote アプリケーションでは、データベースが正しく同期された状態で作成してください。そうすれば、トランザクション・ログは必要ありません。この目的で、抽出ユーティリティを使用することができます。

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したり、ライト・ファイルを使用したりする限りにおいては、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、『ASA データベース管理ガイド』> 「-r サーバ・オプション」を参照してください。

CD-ROM のような読み込み専用メディアで配布された Adaptive Server Anywhere データベースへの変更を可能にするには、「**ライト・ファイル**」を使用します。ライト・ファイルは読み込み専用データベース・ファイルに対する変更を記録し、ハード・ディスクなどの読み込み／書き込み記憶メディア上に置かれます。

この場合、データベース・ファイルが CD-ROM 上に置かれるのに対して、ライト・ファイルはディスク上に置かれます。接続はライト・ファイルに対して行われ、ライト・ファイルはトランザクション・ログ・ファイルをディスク上で管理します。

ライト・ファイルの詳細については、『ASA データベース管理ガイド』> 「ライト・ファイルの処理 (旧式)」を参照してください。

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マシン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**　Adaptive Server 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」662 ページを参照してください。

- **データベース・サーバ**　Adaptive Server 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」695 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 (*dblgen9.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。

Adaptive Server 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.exe* 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ファイルとともに次の追加ファイルも必要です。

説明	Windows	UNIX
データベース・ツール・ライブラリ	*dbtool9.dll*	*libdbtools9.so*、*libdbtasks9.so*
言語リソース・ライブラリ	*dblgen9.dll*	*dblgen9.res*
[接続] ダイアログ (dbisqlc のみ)	*dbcon9.dll*	

注意

- データベース・ツールは Embedded SQL アプリケーションです。「Embedded SQL クライアントの配備」671 ページにリストされているアプリケーションに必要なファイルを提供してください。
- 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、*libdbtools9_r.so* および *libdbtasks9_r.so* 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	UNIX
Message Agent	*dbremote.exe*	*dbremote*
データベース・ツール・ライブラリ	*dbtool9.dll*	*libdbtools9.so*、*libdbtasks9.so*
言語リソース・ライブラリ	*dblgen9.dll*	*dblgen9.res*

説明	Windows	UNIX
VIM メッセージ・リンク・ライブラリ [1]	*dbvim9.dll*	
SMTP メッセージ・リンク・ライブラリ [1]	*dbsmtp9.dll*	
FILE メッセージ・リンク・ライブラリ [1]	*dbfile9.dll*	*libdbfile9.so*
FTP メッセージ・リンク・ライブラリ [1]	*dbftp9.dll*	
MAPI メッセージ・リンク・ライブラリ [1]	*dbmapi9.dll*	
インタフェース・ライブラリ	*dblib9.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。

Adaptive Server Anywhere のインストール動作に従い、SQL Remote ファイルを Adaptive Server 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。

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、*libdbtools9_r.so* および *libdbtasks9_r.so* を参照してください。

第 19 章

SQL プリプロセッサ・エラー・メッセージ

この章の内容

この章では、SQL プリプロセッサのすべてのエラーと警告について説明します。

SQL プリプロセッサ・エラー・メッセージ (エラー・メッセージ値順)

メッセージ値	メッセージ
2601	「サブスクリプト値 %1 が大きすぎます。」725 ページ
2602	「ホスト・タイプにはポインタ配列を使用できません。」715 ページ
2603	「char 型では 1 次元配列のみサポートされます。」724 ページ
2604	「VARCHAR 型には長さの指定が必要です。」713 ページ
2605	「VARCHAR 型の配列はサポートされません。」714 ページ
2606	「VARCHAR 型のホスト変数をポインタとすることはできません。」713 ページ
2607	「VARCHAR 型のホスト変数は初期化できません。」719 ページ
2608	「FIXCHAR 型には長さの指定が必要です。」710 ページ
2609	「FIXCHAR 型の配列はサポートされません。」713 ページ
2610	「この型の配列はサポートされません。」714 ページ
2611	「decimal 型には精度の指定が必要です。」724 ページ
2612	「decimal 型の配列はサポートされません。」714 ページ

メッセージ値	メッセージ
2613	「未知の hostvar 型です。」713 ページ
2614	「無効な整数」721 ページ
2615	「'%1' ホスト変数は C 文字列型でなければなりません。」709 ページ
2617	「'%1' シンボルはすでに定義されています。」709 ページ
2618	「SQL 文の変数の型が無効です。」721 ページ
2619	「インクルード・ファイル '%1' がみつかりません。」710 ページ
2620	「'%1' は未知のホスト変数です。」717 ページ
2621	「'%1' は未知のインジケータ変数です。」719 ページ
2622	「インジケータ変数 '%1' として無効な型です。」721 ページ
2623	「'%1' に無効なホスト変数の型が使われました。」720 ページ
2625	「ホスト変数 '%1' には 2 つの異なった定義があります。」717 ページ
2626	「'%1' 文はあらかじめ準備されていませんでした。」724 ページ
2627	「カーソル '%1' は前もって宣言されていませんでした。」715 ページ
2628	「文 '%1' は認識できません。」726 ページ
2629	「このカーソルではホスト変数を使用できません。」718 ページ
2630	「ホスト変数が宣言部とオープン部の両方で指定されています。」718 ページ

メッセージ値	メッセージ
2631	「%1 のホスト・リストか、using 句を指定しなければなりません。」722 ページ
2633	「SELECT 文に INTO 句がありません。」723 ページ
2634	「SQL の使い方が正しくありません。拡張 '%1' です。」719 ページ
2635	「Embedded SQL の使い方が正しくありません。拡張 '%1' です。」718 ページ
2636	「不正な Embedded SQL 構文です。」718 ページ
2637	「文字列の終わりに引用符がありません。」722 ページ
2639	「トークンが長すぎます。」725 ページ
2640	「'%1' ホスト変数は整数型でなければなりません。」709 ページ
2641	「DESCRIBE には必ず SQLDA を指定してください。」723 ページ
2642	「同じ種類 (INTO または USING) の SQLDA が 2 つ指定されています。」712 ページ
2646	「静的カーソルは記述できません。」715 ページ
2647	「マクロは再定義できません。」712 ページ
2648	「配列の次元が正しくありません。」712 ページ
2649	「記述子のインデックスが正しくありません。」720 ページ
2650	「SET DESCRIPTOR に対する不正なフィールドです。」720 ページ
2651	「SET DESCRIPTOR 文ですでに使用されたフィールドです。」716 ページ

メッセージ値	メッセージ
2652	「データの値はホスト変数でなければなりません。」715 ページ
2660	「カーソル宣言文で Into 句は使用できません。Into 句は無視されます。」712 ページ
2661	「認識できない SQL 構文です。」727 ページ
2662	「'%1' は未知の SQL 関数です。」726 ページ
2663	「SQL 関数 '%1' のパラメータ数が正しくありません。」727 ページ
2664	「静的文の名前は、2 つのスレッドで使用する場合、正しく機能しません。」725 ページ
2665	「ホスト変数 '%1' が再定義されています。」717 ページ
2666	「ベンダの拡張機能」727 ページ
2667	「SQL の中間機能」719 ページ
2668	「SQL の全機能」716 ページ
2669	「transact SQL の拡張機能」726 ページ
2680	「宣言の部分と INCLUDE SQLCA 文がありません。」724 ページ
2681	「テンポラリ・ファイルを開けません。」726 ページ
2682	「テンポラリ・ファイルの読み込み中にエラーが発生しました。」716 ページ
2683	「出力ファイルへの書き込み中にエラーが発生しました。」716 ページ
2690	「このカーソルのホスト変数の数が不一致です。」711 ページ

メッセージ値	メッセージ
2691	「このカーソルのホスト変数のタイプが不一致です。」711 ページ
2692	「このカーソルのインジケータ変数が不一致です。」711 ページ
2693	「Ultra Light では使用できない機能です。」710 ページ
2694	「カーソル '%1' に対する OPEN がありません。」723 ページ
2695	「カーソル '%1' に対する FETCH または PUT はありません。」723 ページ
2696	「ホスト変数 '%1' は異なるインジケータですでに使用されています。」710 ページ
2697	「Ultra Light の long binary または long varchar のサイズ制限は 65,535 です。」722 ページ

SQLPP エラー

この項では、SQL プリプロセッサが生成するメッセージをリストします。メッセージにはエラーと警告があります。コマンド・ライン・オプションによってメッセージの意味が異なる場合もあります。

SQL プリプロセッサとコマンド・ライン・オプションの詳細については、「SQL プリプロセッサ」248 ページを参照してください。

'%1' ホスト変数は C 文字列型でなければなりません。

メッセージ値	メッセージ・タイプ
2615	エラー

考えられる原因　Embedded SQL 文 (カーソル名、オプション名など) 中で、C 文字列が要求されましたが、違う型の値が渡されました。

'%1' ホスト変数は整数型でなければなりません。

メッセージ値	メッセージ・タイプ
2640	エラー

考えられる原因　整数型のホスト変数だけが許可される文中で、整数型ではないホスト変数を使いました。

'%1' シンボルはすでに定義されています。

メッセージ値	メッセージ・タイプ
2617	エラー

考えられる原因　ホスト変数を 2 回定義しました。

インクルード・ファイル '%1' がみつかりません。

メッセージ値	メッセージ・タイプ
2619	エラー

考えられる原因 指定されたインクルード・ファイルが見つかりませんでした。プリプロセッサはインクルード・ファイルを探すのに、INCLUDE 環境変数を使用することに注意してください。

FIXCHAR 型には長さの指定が必要です。

メッセージ値	メッセージ・タイプ
2608	エラー

考えられる原因 DECL_FIXCHAR マクロを使用して FIXCHAR 型のホスト変数を宣言しましたが、長さを指定しませんでした。

Ultra Light では使用できない機能です。

メッセージ値	メッセージ・タイプ
2693	フラグ (警告またはエラー)

考えられる原因 Ultra Light でサポートされていない機能を使用しました。

ホスト変数 '%1' は異なるインジケータですでに使用されています。

メッセージ値	メッセージ・タイプ
2696	エラー

考えられる原因　同じ文の中で、異なるインジケータ変数を使用して同じホスト変数が複数回使用されています。これはサポートされていません。

このカーソルのホスト変数のタイプが不一致です。

メッセージ値	メッセージ・タイプ
2691	エラー

考えられる原因　使用したホスト変数の型や長さが、以前にカーソルで使用したものとは異なります。カーソルに対して一貫したホスト変数の型を使用してください。

このカーソルのインジケータ変数が不一致です。

メッセージ値	メッセージ・タイプ
2692	エラー

考えられる原因　以前にカーソルで使用していないインジケータ変数を使用しました。または、以前にカーソルで使用したインジケータ変数を使用していません。カーソルに対して一貫したインジケータ変数を使用してください。

このカーソルのホスト変数の数が不一致です。

メッセージ値	メッセージ・タイプ
2690	エラー

考えられる原因　ホスト変数の数が以前にカーソルで使用した数と異なっています。カーソルに対して同じ数のホスト変数を使用してください。

カーソル宣言文で Into 句は使用できません。Into 句は無視されます。

メッセージ値	メッセージ・タイプ
2660	警告

考えられる原因 DECLARE CURSOR 文で INTO 句を使用しました。INTO 句は無視されます。

配列の次元が正しくありません。

メッセージ値	メッセージ・タイプ
2648	エラー

考えられる原因 配列の次元が負の値になっています。

マクロは再定義できません。

メッセージ値	メッセージ・タイプ
2647	エラー

考えられる原因 ヘッダ・ファイルで、プリプロセッサ・マクロが 2 回定義されていることが考えられます。

同じ種類 (INTO または USING) の SQLDA が 2 つ指定されています。

メッセージ値	メッセージ・タイプ
2642	エラー

考えられる原因 この文に 2 つの INTO DESCRIPTOR 句または 2 つの USING DESCRIPTOR 句を指定しました。

未知の hostvar 型です。

メッセージ値	メッセージ・タイプ
2613	エラー

考えられる原因　SQL プリプロセッサが理解できないデータ型で、ホスト変数を宣言しました。

VARCHAR 型のホスト変数をポインタとすることはできません。

メッセージ値	メッセージ・タイプ
2606	エラー

考えられる原因　ホスト変数を VARCHAR または BINARY のポインタとして宣言しようとしました。これはホスト変数の型としては許可されません。

VARCHAR 型には長さの指定が必要です。

メッセージ値	メッセージ・タイプ
2604	エラー

考えられる原因　DECL_VARCHAR または DECL_BINARY マクロを使用して、VARCHAR または BINARY ホスト変数を宣言しましたが、配列のサイズを指定しませんでした。

FIXCHAR 型の配列はサポートされません。

メッセージ値	メッセージ・タイプ
2609	エラー

考えられる原因 ホスト変数を FIXCHAR の配列として宣言しようとしました。これはホスト変数の型としては許可されません。

VARCHAR 型の配列はサポートされません。

メッセージ値	メッセージ・タイプ
2605	エラー

考えられる原因 ホスト変数を VARCHAR または BINARY の配列として宣言しようとしました。これはホスト変数の型としては許可されません。

decimal 型の配列はサポートされません。

メッセージ値	メッセージ・タイプ
2612	エラー

考えられる原因 ホスト変数を DECIMAL の配列として宣言しようとしました。DECIMAL 型の配列は、ホスト変数の型としては許可されません。

この型の配列はサポートされません。

メッセージ値	メッセージ・タイプ
2610	エラー

考えられる原因 サポートされていない型のホスト変数の配列を宣言しようとしました。

静的カーソルは記述できません。

メッセージ値	メッセージ・タイプ
2646	エラー

考えられる原因　静的カーソルを記述しました。カーソルを記述するときは、ホスト変数にカーソル名を指定してください。

ホスト・タイプにはポインタ配列を使用できません。

メッセージ値	メッセージ・タイプ
2602	エラー

考えられる原因　ポインタの配列をホスト変数として使用しました。これは許可されません。

カーソル '%1' は前もって宣言されていませんでした。

メッセージ値	メッセージ・タイプ
2627	エラー

考えられる原因　Embedded SQL カーソル名を最初に宣言 (DECLARE) せずに使用 (FETCH、OPEN、CLOSE など) しました。

データの値はホスト変数でなければなりません。

メッセージ値	メッセージ・タイプ
2652	エラー

考えられる原因 ホスト変数として宣言されていない変数を SET DESCRIPTOR 文で使用しようとしました。

テンポラリ・ファイルの読み込み中にエラーが発生しました。

メッセージ値	メッセージ・タイプ
2682	エラー

考えられる原因 テンポラリ・ファイルの読み込み中にエラーが発生しました。

出力ファイルへの書き込み中にエラーが発生しました。

メッセージ値	メッセージ・タイプ
2683	エラー

考えられる原因 出力ファイルへの書き込み中にエラーが発生しました。

SET DESCRIPTOR 文ですでに使用されたフィールドです。

メッセージ値	メッセージ・タイプ
2651	エラー

考えられる原因 単一の SET DESCRIPTOR 文で、同じキーワードが 2 回以上使用されました。

SQL の全機能

メッセージ値	メッセージ・タイプ
2668	フラグ (警告またはエラー)

考えられる原因　SQL/92 の全機能を使用し、-ee、-ei、-we、または -wi のフラグ・スイッチを指定して前処理を実行しました。

ホスト変数 '%1' が再定義されています。

メッセージ値	メッセージ・タイプ
2665	警告

考えられる原因　同一のホスト変数を、異なるホストの型で再定義しています。プリプロセッサに関しては、ホスト変数はグローバルです。型が異なる 2 つのホスト変数には同じ名前を使用できません。

ホスト変数 '%1' には 2 つの異なった定義があります。

メッセージ値	メッセージ・タイプ
2625	エラー

考えられる原因　同じモジュール中で、同じホスト変数名に対して 2 つの異なった型が定義されています。ホスト変数名は C モジュールに対してグローバルであることに注意してください。

'%1' は未知のホスト変数です。

メッセージ値	メッセージ・タイプ
2620	エラー

考えられる原因　宣言セクションで宣言されなかったホスト変数を、文の中で使用しました。

このカーソルではホスト変数を使用できません。

メッセージ値	メッセージ・タイプ
2629	エラー

考えられる原因

指定されたカーソルの宣言では、ホスト変数は許可されません。ホスト変数を通してカーソル名を与える場合は、完全な動的 SQL を使用して文を作成してください。作成された文中にはホスト変数があってもかまいません。

ホスト変数が宣言部とオープン部の両方で指定されています。

メッセージ値	メッセージ・タイプ
2630	エラー

考えられる原因

declare と open 文の両方のカーソルにホスト変数を指定しました。この場合、declare 文でホスト変数を指定してください。動的の場合、指定してから開いてください。

Embedded SQL の使い方が正しくありません。拡張 '%1' です。

メッセージ値	メッセージ・タイプ
2635	エラー

不正な Embedded SQL 構文です。

メッセージ値	メッセージ・タイプ
2636	エラー

考えられる原因　Embedded SQL 文 (OPEN、DECLARE、FETCH など) に構文エラーがあります。

SQL の使い方が正しくありません。拡張 '%1' です。

メッセージ値	メッセージ・タイプ
2634	エラー

'%1' は未知のインジケータ変数です。

メッセージ値	メッセージ・タイプ
2621	エラー

考えられる原因　宣言セクションで宣言されなかったインジケータ変数を、文の中で使用しました。

VARCHAR 型のホスト変数は初期化できません。

メッセージ値	メッセージ・タイプ
2607	エラー

考えられる原因　VARCHAR または BINARY データ型のホスト変数に、C 変数イニシャライザは指定できません。この変数は、通常の C プログラム・コードで初期化する必要があります。

SQL の中間機能

メッセージ値	メッセージ・タイプ
2667	フラグ (警告またはエラー)

考えられる原因 SQL/92 の中間機能を使用し、-ee または -we のフラグ・スイッチを指定して前処理を実行しました。

記述子のインデックスが正しくありません。

メッセージ値	メッセージ・タイプ
2649	エラー

考えられる原因 ALLOCATE DESCRIPTOR 文で変数が 1 つも割り付けられていません。

SET DESCRIPTOR に対する不正なフィールドです。

メッセージ値	メッセージ・タイプ
2650	エラー

考えられる原因 SET DESCRIPTOR 文に不正なキーワードまたは未知のキーワードがあります。使用できるキーワードは、TYPE、PRECISION、SCALE、LENGTH、INDICATOR、DATA のいずれかです。

'%1' に無効なホスト変数の型が使われました。

メッセージ値	メッセージ・タイプ
2623	エラー

考えられる原因 プリプロセッサが文字列型のホスト変数を期待している場所に、文字列型ではないホスト変数が使われました。

無効な整数

メッセージ値	メッセージ・タイプ
2614	エラー

考えられる原因　Embedded SQL 文 (フェッチ・オフセット、ホスト変数配列インデックスなど) 中で、整数が要求されましたが、プリプロセッサは入力された内容を整数に変換できませんでした。

インジケータ変数 '%1' として無効な型です。

メッセージ値	メッセージ・タイプ
2622	エラー

考えられる原因　インジケータ変数の型は short int である必要があります。これとは違う型が使われました。

SQL 文の変数の型が無効です。

メッセージ値	メッセージ・タイプ
2618	エラー

考えられる原因　文識別子として使われるホスト変数の型は a_sql_statement_number になります。他の型のホスト変数を使用しようとしました。

Ultra Light の long binary または long varchar のサイズ制限は 65,535 です。

メッセージ値	メッセージ・タイプ
2697	エラー

考えられる原因 Ultra Light で DECL_LONGBINARY または DECL_LONGVARCHAR を使用する場合、配列の最大サイズは 64 KB です。

文字列の終わりに引用符がありません。

メッセージ値	メッセージ・タイプ
2637	エラー

考えられる原因 Embedded SQL 文中で文字列定数を指定しましたが、文字列を囲む引用符の後ろの方が行の最後またはファイルの最後までに見つかりません。

%1 のホスト・リストか、using 句を指定しなければなりません。

メッセージ値	メッセージ・タイプ
2631	エラー

考えられる原因 指定された文には、ホスト変数がホスト変数リストまたは SQLDA から指定される必要があります。

DESCRIBE には必ず SQLDA を指定してください。

メッセージ値	メッセージ・タイプ
2641	エラー

カーソル '%1' に対する FETCH または PUT はありません。

メッセージ値	メッセージ・タイプ
2695	エラー

考えられる原因　カーソルが宣言されて開かれましたが、使用されませんでした。

SELECT 文に INTO 句がありません。

メッセージ値	メッセージ・タイプ
2633	エラー

考えられる原因　Embedded Static SELECT 文は指定しましたが、結果のための INTO 句は指定しませんでした。

カーソル '%1' に対する OPEN がありません。

メッセージ値	メッセージ・タイプ
2694	エラー

考えられる原因　カーソルが宣言され、開かれずに使用されたことが考えられます。

宣言の部分と INCLUDE SQLCA 文がありません。

メッセージ値	メッセージ・タイプ
2680	エラー

考えられる原因 EXEC SQL INCLUDE SQLCA 文がソース・ファイルにありません。

char 型では 1 次元配列のみサポートされます。

メッセージ値	メッセージ・タイプ
2603	エラー

考えられる原因 ホスト変数を文字の配列として宣言しようとしました。これはホスト変数の型としては許可されません。

decimal 型には精度の指定が必要です。

メッセージ値	メッセージ・タイプ
2611	エラー

考えられる原因 DECL_DECIMAL マクロを使用して、パックされた 10 進数のホスト変数を宣言するときは、精度を指定する必要があります。小数点以下の桁数はオプションです。

'%1' 文はあらかじめ準備されていませんでした。

メッセージ値	メッセージ・タイプ
2626	エラー

考えられる原因　Embedded SQL 文名を最初に準備 (PREPARE) せずに実行 (EXECUTE) しました。

静的文の名前は、2 つのスレッドで使用する場合、正しく機能しません。

メッセージ値	メッセージ・タイプ
2664	警告

考えられる原因　静的文の名前を使って、-r リエントランシ・スイッチで事前に処理しました。静的文の名前は静的変数を生成し、それはデータベースによって入力されます。2 つのスレッドが同じ文を使うと、この変数の競合が生じます。文識別子には静的な名前ではなく、ローカルなホスト変数を使用してください。

サブスクリプト値 %1 が大きすぎます。

メッセージ値	メッセージ・タイプ
2601	エラー

考えられる原因　インデックスを付けようとしたホスト変数が、大きすぎる値を持つ配列でした。

トークンが長すぎます。

メッセージ値	メッセージ・タイプ
2639	エラー

考えられる原因　SQL プリプロセッサの最大トークン長は 2 K です。これより長いトークンはエラーになります。Embedded SQL コマンドの定数文字列 (このエラーが主に発生する場所) では、長い文字列を作成するには文字列の連結を使用してください。

transact SQL の拡張機能

メッセージ値	メッセージ・タイプ
2669	フラグ (警告またはエラー)

考えられる原因 SQL/92 で定義されていない Sybase Transact-SQL 機能を使用し、-ee、-ei、-ef、-we、-wi、または -wf のフラグ・スイッチを指定して前処理を実行しました。

テンポラリ・ファイルを開けません。

メッセージ値	メッセージ・タイプ
2681	エラー

考えられる原因 テンポラリ・ファイルを開こうとしたときにエラーが発生しました。

'%1' は未知の SQL 関数です。

メッセージ値	メッセージ・タイプ
2662	警告

考えられる原因 プリプロセッサが認識できない SQL 関数を使いました。これがデータベース・エンジンに送信されると、エラーになる可能性があります。

文 '%1' は認識できません。

メッセージ値	メッセージ・タイプ
2628	エラー

考えられる原因　存在しない Embedded SQL 文を削除しようとしました。

認識できない SQL 構文です。

メッセージ値	メッセージ・タイプ
2661	警告

考えられる原因　データベース・エンジンに送信されるとエラーになる可能性のある SQL 文を使いました。

ベンダの拡張機能

メッセージ値	メッセージ・タイプ
2666	フラグ (警告またはエラー)

考えられる原因　SQL/92 で定義されていない Adaptive Server Anywhere 機能を使用し、-ee、-ei、-ef、-we、-wi、または -wf のフラグ・スイッチを指定して前処理を実行しました。

SQL 関数 '%1' のパラメータ数が正しくありません。

メッセージ値	メッセージ・タイプ
2663	警告

考えられる原因　SQL 関数のパラメータ数が正しくありません。これがデータベース・エンジンに送信されると、エラーになる可能性があります。

索引

記号

A

B

C

D

E

F

G

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Z

あ

い

き

く

け

こ

さ

し

す

せ

そ

た

ち

つ

て

と

に

ね

は

ひ

ふ

め

も

ゆ

よ

ら

り

れ

わ