SYBASE®

Embedded SQL™/C プログラマーズ・ガイド

Open Client™

15.5

ドキュメント ID：DC37697-01-1550-01
改訂：2009 年 10 月

マニュアルの注文

マニュアルの注文を承ります。ご希望の方は、サイベース株式会社営業部または代理店までご連絡ください。マニュアルの変更は、弊社の定期的なソフトウェア・リリース時にのみ提供されます。

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

目次

はじめに

このマニュアルでは、C 言語のアプリケーションで Embedded SQL™ と Embedded SQL プリコンパイラを使う方法について説明します。Sybase® Embedded SQL は、Transact-SQL® のスーパーセットです。このセットを使うと、COBOL や C 言語で開発されたアプリケーション・プログラム内に Transact-SQL 文を埋め込むことができます。

このマニュアルの情報は、各プラットフォームに共通です。プラットフォーム特有の Embedded SQL の使い方については、使用しているプラットフォーム用の『Open Client/Server プログラマーズ・ガイド補足』を参照してください。

対象読者

このマニュアルは、Embedded SQL の概念と使用に関心のあるアプリケーション開発者を対象としています。このマニュアルの対象読者には、次の知識が必要です。

- 『ASE リファレンス・マニュアル』の内容を理解していること
- C のプログラミング経験があること

このマニュアルの内容

このマニュアルには、以下の章があります。

- 「第 1 章 概要」では、Embedded SQL の簡単な概要を示し、Embedded SQL の利点と機能について説明します。
- 「第 2 章 一般情報」では、Embedded SQL プログラムおよび Embedded SQL を使用したプログラミングの一般的な規則について説明します。
- 「第 3 章 Adaptive Server Enterprise との通信」では、SQLCA、SQLCODE、SQLSTATE による通信領域の確立と使用方法について説明します。また、通信領域で使うシステム変数についても説明します。
- 「第 4 章 変数の使い方」では、Embedded SQL 内でのホスト変数やインジケータ変数の宣言および使用方法について説明します。また、配列およびデータ型の変換についても説明します。
- 「第 5 章 Adaptive Server Enterprise への接続」では、Embedded SQL を使ってアプリケーション・プログラムを Adaptive Server® Enterprise および通常のデータ・サーバに接続する方法について説明します。
- 「第 6 章 Transact-SQL 文の使い方」では、Embedded SQL アプリケーション・プログラム内で Transact-SQL を使う方法について説明します。ここでは、配列およびバッチを使用したローの選択方法と、Transact-SQL 文をグループ化する方法を説明します。

- 「第 7 章 動的 SQL の使い方」では、アプリケーションのユーザが、実行時に対話形式で入力できる Embedded SQL 文の作成方法について説明します。
- 「第 8 章 エラーの処理」では、リターン・コードと、エラーの検出および対処のための Embedded SQL プリコンパイラ機能について説明します。
- 「第 9 章 継続バインドによるパフォーマンスの向上」では、継続バインドを使用したパフォーマンスの向上とその実装方法について説明します。
- 「第 10 章 Embedded SQL 文:リファレンス・ページ」では、個々の Embedded SQL 文のリファレンス・ページを提供します。
- 「第 11 章 Open Client/Server 設定ファイル」では、Embedded SQL での外部設定ファイルの使用方法について説明します。
- 「付録 A プリコンパイラの警告とエラー・メッセージ」では、プリコンパイラおよび実行時メッセージのリストを掲載します。
- 「付録 B サイズの大きな text データと image データを処理するサンプル・コード」では、大きな text および image データを処理するホスト変数の使用方法を示す Embedded SQL のサンプル・プログラムについて説明します。

関連マニュアル

詳細については、これらのマニュアルを参照できます。

- 『Open Server リリース・ノート Microsoft Windows 版』には、Open Server に関する重要な最新情報が記載されています。
- 『Software Developer's Kit リリース・ノート Microsoft Windows 版』には、Open Client™ および SDK に関する重要な最新情報が記載されています。
- 『jConnect™ for JDBC™ リリース・ノート バージョン 6.05 および 7.0』には、jConnect に関する重要な最新情報が記載されています。
- この『Open Client/Server 設定ガイド Microsoft Windows 版』では、システムを設定して Open Client/Server 製品を実行する方法について説明します。
- 『Open Client Client-Library/C リファレンス・マニュアル』では、Open Client Client-Library™ のリファレンス情報について説明しています。
- 『Open Client Client-Library/C プログラマーズ・ガイド』では、Client-Library アプリケーションの設計方法および実装方法について説明しています。
- 『Open Server Server-Library/C リファレンス・マニュアル』では、Open Server Server-Library のリファレンス情報について説明しています。
- 『Open Client および Open Server Common Libraries リファレンス・マニュアル』では、CS-Library のリファレンス情報について説明しています。CS-Library は、Client-Library と Server-Library の両方のアプリケーションで役に立つユーティリティ・ルーチンの集まりです。

- 『Open Client/Server プログラマーズ・ガイド補足 Microsoft Windows 版』では、Open Client/Server を使用するプログラマのために、プラットフォーム固有の情報について説明しています。このマニュアルには、次の情報が含まれています。
  - アプリケーションのコンパイルおよびリンク
  - Open Client/Server に含まれているサンプル・プログラム
  - プラットフォーム固有の動作をするルーチン
- 『jConnect for JDBC インストール・ガイド バージョン 6.05』では、jConnect for JDBC のインストール方法について説明しています。
- 『jConnect for JDBC プログラマーズ・リファレンス』では、jConnect for JDBC 製品について説明し、リレーショナル・データベース管理システムに保管されているデータにアクセスする方法について説明しています。
- 『Adaptive Server® Enterprise ADO.NET Data Provider ユーザーズ・ガイド』では、C#、Visual Basic .NET、マネージ拡張を備えた C++、J# など、.NET でサポートされる任意の言語を使用して Adaptive Server 内のデータにアクセスする方法について説明しています。
- Sybase 製 Adaptive Server Enterprise ODBC ドライバの『ユーザーズ・ガイド』(Windows および Linux 版) では、Windows、Linux、および Apple Mac OS X プラットフォームの Adaptive Server から、Open Database Connectivity (ODBC) ドライバ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。
- Sybase 製 Adaptive Server Enterprise OLE DB プロバイダの『ユーザーズ・ガイド』(Microsoft Windows 版) では、Microsoft Windows プラットフォームの Adaptive Server から、OLE DB プロバイダ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。

その他の情報

Sybase® Getting Started CD、SyBooks™ CD、Sybase Product Manuals Web サイトを利用すると、製品について詳しく知ることができます。

- Getting Started CD には、PDF 形式のリリース・ノートとインストール・ガイド、SyBooks CD に含まれていないその他のマニュアルや更新情報が収録されています。この CD は製品のソフトウェアに同梱されています。Getting Started CD に収録されているマニュアルを参照または印刷するには、Adobe Acrobat Reader が必要です (CD 内のリンクを使用して Adobe の Web サイトから無料でダウンロードできます)。
- SyBooks CD には製品マニュアルが収録されています。この CD は製品のソフトウェアに同梱されています。Eclipse ベースの SyBooks ブラウザを使用すれば、使いやすい HTML 形式のマニュアルにアクセスできます。

  一部のマニュアルは PDF 形式で提供されています。これらのマニュアルは SyBooks CD の PDF ディレクトリに収録されています。PDF ファイルを開いたり印刷したりするには、Adobe Acrobat Reader が必要です。

SyBooks をインストールして起動するまでの手順については、Getting Started CD の『SyBooks インストール・ガイド』、または SyBooks CD の *README.txt* ファイルを参照してください。

- Sybase Product Manuals Web サイトは、SyBooks CD のオンライン版であり、標準の Web ブラウザを使用してアクセスできます。また、製品マニュアルのほか、EBFs/Maintenance、Technical Documents、Case Management、Solved Cases、ニュース・グループ、Sybase Developer Network へのリンクもあります。

  Technical Library Product Manuals Web サイトにアクセスするには、Product Manuals (http://www.sybase.com/support/manuals/) にアクセスしてください。

Web 上の Sybase 製品の動作確認情報

Sybase Web サイトの技術的な資料は頻繁に更新されます。

❖ **製品認定の最新情報にアクセスする**

1. Web ブラウザで Technical Documents を指定します。(http://www.sybase.com/support/techdocs/)
2. [Partner Certification Report] をクリックします。
3. [Partner Certification Report] フィルタで製品、プラットフォーム、時間枠を指定して [Go] をクリックします。
4. [Partner Certification Report] のタイトルをクリックして、レポートを表示します。

❖ **コンポーネント認定の最新情報にアクセスする**

1. Web ブラウザで Availability and Certification Reports を指定します。(http://certification.sybase.com/)
2. [Search By Base Product] で製品ファミリとベース製品を選択するか、[Search by Platform] でプラットフォームとベース製品を選択します。
3. [Search] をクリックして、入手状況と認定レポートを表示します。

❖ **Sybase Web サイト (サポート・ページを含む) の自分専用のビューを作成する**

MySybase プロファイルを設定します。MySybase は無料サービスです。このサービスを使用すると、Sybase Web ページの表示方法を自分専用にカスタマイズできます。

1. Web ブラウザで Technical Documents を指定します。(http://www.sybase.com/support/techdocs/)
2. [MySybase] をクリックし、MySybase プロファイルを作成します。

Sybase EBF とソフトウェア・メンテナンス

❖ EBF とソフトウェア・メンテナンスの最新情報にアクセスする

1 Web ブラウザで Sybase Support Page を指定します。(http://www.sybase.com/support)

2 [EBFs/Maintenance] を選択します。MySybase のユーザ名とパスワードを入力します。

3 製品を選択します。

4 時間枠を指定して [Go] をクリックします。EBF/Maintenance リリースの一覧が表示されます。

鍵のアイコンは、「Technical Support Contact」として登録されていないため、一部の EBF/Maintenance リリースをダウンロードする権限がないことを示しています。未登録でも、Sybase 担当者またはサポート・コンタクトから有効な情報を得ている場合は、[Edit Roles] をクリックして、「Technical Support Contact」の役割を MySybase プロファイルに追加します。

5 EBF/Maintenance レポートを表示するには [Info] アイコンをクリックします。ソフトウェアをダウンロードするには製品の説明をクリックします。

表記規則

表 *1:* 構文の表記規則

キー	定義
command	コマンド名、コマンドのオプション名、ユーティリティ名、ユーティリティのフラグ、キーワードは sans serif で示す。
variable	変数 (ユーザが入力する値を表す語) は斜体で表記する。
{ }	中カッコは、その中から必ず 1 つ以上のオプションを選択しなければならないことを意味する。コマンドには中カッコは入力しない。
[ ]	角カッコは、オプションを選択しても省略してもよいことを意味する。コマンドには中カッコは入力しない。
( )	このカッコはコマンドの一部として入力する。
|	中カッコまたは角カッコの中の縦線で区切られたオプションのうち 1 つだけを選択できることを意味する。
,	中カッコまたは角カッコの中のカンマで区切られたオプションをいくつでも選択できることを意味する。複数のオプションを選択する場合には、オプションをカンマで区切る。

アクセシビリティ機能

このマニュアルには、アクセシビリティを重視した HTML 版もあります。この HTML 版マニュアルは、スクリーン・リーダーで読み上げる、または画面を拡大表示するなどの方法により、その内容を理解できるよう配慮されています。

Open Client および Open Server のマニュアルは、連邦リハビリテーション法第 508 条のアクセシビリティ規定に準拠していることがテストにより確認されています。第 508 条に準拠しているマニュアルは通常、World Wide Web Consortium (W3C) の Web サイト用ガイドラインなど、米国以外のアクセシビリティ・ガイドラインにも準拠しています。

注意 アクセシビリティ・ツールを効率的に使用するには、設定が必要な場合もあります。一部のスクリーン・リーダーは、テキストの大文字と小文字を区別して発音します。たとえば、すべて大文字のテキスト (ALL UPPERCASE TEXT など) はイニシャルで発音し、大文字と小文字の混在したテキスト (Mixed Case Text など) は単語として発音します。構文規則を発音するようにツールを設定すると便利かもしれません。詳細については、ツールのマニュアルを参照してください。

Sybase のアクセシビリティに対する取り組みについては、Sybase Accessibility (http://www.sybase.com/accessibility) を参照してください。Sybase Accessibility サイトには、第 508 条と W3C 標準に関する情報へのリンクもあります。

不明な点があるときは

Sybase ソフトウェアがインストールされているサイトには、Sybase 製品の保守契約を結んでいるサポート・センタとの連絡担当の方 (コンタクト・パーソン) を決めてあります。マニュアルだけでは解決できない問題があった場合には、担当の方を通して Sybase のサポート・センタまでご連絡ください。

第 1 章

概要

この章では、Embedded SQL および Embedded SQL プリコンパイラの概要について説明します。

Embedded SQL の概要

Embedded SQL は Transact-SQL のスーパーセットで、C および COBOL で記述されたアプリケーション・プログラムに Transact-SQL 文を埋め込みます。

Open Client™ Embedded SQL を使用して、Adaptive Server Enterprise のデータをアクセスしたり更新したりするプログラムを作成できます。Embedded SQL のプログラマは、C や COBOL のような一般的なプログラミング言語で記述されたアプリケーション・プログラムの中に直接 SQL 文を記述します。プリプロセッシング・プログラムである Embedded SQL プリコンパイラは、完成したアプリケーション・プログラムを処理して、ホスト言語コンパイラがコンパイル可能なプログラムにします。このプログラムは実行される前に Open Client Client-Library™ にリンクされます。

Embedded SQL は、Adaptive Server Enterprise へアクセスするために Sybase が提供する 2 つのプログラミング方法のうちの 1 つです。もう 1 つのプログラミング方法は、呼び出しレベルのインタフェースです。呼び出しレベルのインタフェースでは、Client-Library 呼び出しを直接アプリケーション・プログラムに記述してから Client-Library とリンクします。

Embedded SQL 文は「ホスト・プログラム」の任意の場所に、ホスト言語の文と混在して記述することができます。すべての Embedded SQL 文は exec sql というキーワードで開始し、セミコロン (;) で終了しなければなりません。

Adaptive Server Enterprise から取得したデータを保管したり、select 文の where 句のように Embedded SQL 文中のパラメータとして、「ホスト変数」を Embedded SQL 文中に使用できます。動的 SQL では、ホスト変数は Embedded SQL 文に対するテキストも含むことができます。

Embedded SQL プログラムを記述したあとは、プリコンパイラを実行してください。プリコンパイラは、Embedded SQL 文を Client-Library 関数呼び出しに変換します。

Embedded SQL の特長

Embedded SQL は、呼び出しレベル・インタフェースに比べ、次のようないくつかの有利な点があります。

- Embedded SQL は、Transact-SQL に、アプリケーション内で使いやすくするための機能を単に追加したものなので、操作が簡単です。
- Embedded SQL は、ANSI/ISO 標準のプログラミング言語です。
- 呼び出しレベルの場合と比べて、少ないコーディングで同じ結果が得られます。
- Embedded SQL は、異なるホスト言語でも本質的に同じです。プログラム規約や構文の変更はそれほどありません。したがって、異なる言語でアプリケーションを作成する場合にも、新たに構文を覚える必要はありません。
- プリコンパイラは、Embedded SQL 文のストアド・プロシージャを作成して実行時間を最適化できます。

Embedded SQL での Transact-SQL のサポート

Transact-SQL とは、『ASE リファレンス・マニュアル』で説明している SQL コマンドのセットです。print、readtext、writetext といった例外を除いて、すべての Transact-SQL 文、関数、フロー制御言語が Embedded SQL において有効です。アプリケーションのデバッグを容易にするため、Transact-SQL で Embedded SQL アプリケーションの対話的なプロトタイプを開発し、それをアプリケーションに組み込むことができます。

Adaptive Server Enterprise データ型のほとんどは、同等のデータ型が Embedded SQL にもあります。また、ホスト言語のデータ型も Embedded SQL の中で使用できます。ホスト言語のデータ型が Adaptive Server Enterprise のデータ型に完全に一致しない場合には、自動的にデータ型が変換されます。

Transact-SQL でリテラル引用符を使用できる場所であれば、Embedded SQL 文にホスト言語の変数を使用できます。一重引用符 (‘) または二重引用符 (“) のどちらかでリテラルを囲みます。引用符が含まれるリテラルの区切り方については、『ASE リファレンス・マニュアル』を参照してください。

Embedded SQL には、Transact-SQL には含まれていない次のような機能があります。

- ホスト言語のデータ型と Adaptive Server Enterprise のデータ型の間の自動データ型変換
- 実行時に SQL 文を定義できる動的 SQL
- Adaptive Server Enterprise とアプリケーション・プログラム間の通信を可能にする *SQLCA*、*SQLCODE*、*SQLSTATE*。この 3 つのエンティティには、Adaptive Server Enterprise が生成するエラー、警告、情報メッセージのコードが含まれています。
- 実行中のエラーを検出するリターン・コード・テスト・ルーチン。

はじめに

プリコンパイラはランタイム・ライブラリとして Client-Library™ を使うので、Client-Library バージョン 12.5 以降がインストールされていることを確認してから、プリコンパイラを起動してください。また、Adaptive Server Enterprise 12.5 以降がインストールされていることも確認してください。不足している製品がある場合には、「システム管理者」に連絡してください。

オペレーティング・システムのプロンプトで、適切な「コマンド」を発行してプリコンパイラを起動します。詳細については、使用しているプラットフォームの『Open Client/Server プログラマーズ・ガイド補足』を参照してください。

プリコンパイラ・コマンドには、プリコンパイラのオプションを決定するフラグがあります。これらのオプションを使用すると、入力ファイル、ログイン・ユーザ名とパスワード、HA フェールオーバの起動、プリコンパイラのモードなどが指定できます。Embedded SQL アプリケーションのプリコンパイル、コンパイル、リンクに関するオペレーティング・システムごとの情報については、『Open Client/Server プログラマーズ・ガイド補足』を参照してください。

プログラム例の利用法

このマニュアルのプログラム例では、pubs2 データベースを使用しています。プログラム例を実行するには、Transact-SQL の use 文で pubs2 データベースを指定します。

この製品には、いくつかのオンライン・プログラム例が添付されています。これらのプログラム例の実行方法については、使用しているプラットフォームの『Open Client/Server プログラマーズ・ガイド補足』を参照してください。

旧バージョンとの互換性

System 11 以降のプリコンパイラは、SQL-89 準拠のプリコンパイラと互換性があります。しかし、システムには ANSI に準拠していない古いリリースの Embedded SQL で作成されたアプリケーションも存在するかもしれません。現在のプリコンパイラは以前のバージョンとほぼ同じ Embedded SQL 文を使いますが、その処理方法は異なります。

以前のリリースのプリコンパイラ用に作成されたアプリケーションを移行するには、次の手順を実行します。

1 次の SQL 文とキーワードは System 11 ではサポートしていないので、アプリケーションから取り除きます。

- release *connection_name*
- recompile
- noparse
- noproc
- pcoptions
- cancel

release 文はプリコンパイラ・エラーの原因となります。つまり、プリコンパイラは他のキーワードを無視します。cancel 文はランタイム・エラーの原因となります。

2 System 11 以降のプリコンパイラを使用して、アプリケーションをもう一度プリコンパイルします。

Embedded SQL プログラムの作成と実行

次に、Embedded SQL アプリケーション・プログラムを作成、実行する手順を示します。

1　アプリケーション・プログラムを記述し、Embedded SQL 文と変数宣言を埋め込みます。

2　アプリケーションを拡張子に *.cp* と付けたファイルに保存します。

3　アプリケーションをプリコンパイルします。重大なエラーが発生しなければ、プリコンパイラはアプリケーション・プログラムを含むファイルを生成します。このファイルには、拡張子だけを変えて元のソース・ファイルと同じ名前が付けられます。この拡張子は、使用している C コンパイラの要件に応じて異なります。詳細については、使用しているプラットフォームの『Open Client/Server プログラマーズ・ガイド補足』を参照してください。

4　新しく生成されたソース・コードを標準の C プログラムと同じようにコンパイルします。

5　コンパイルされたコードを Client-Library とリンクします。

6　ストアド・プロシージャを生成するためのプリコンパイラ・オプションを指定した場合は、isql を使用して生成されたスクリプトを実行して Adaptive Server Enterprise にロードします。

7　標準の C プログラムと同じようにアプリケーション・プログラムを実行します。

プリコンパイラによるアプリケーションの処理

Embedded SQL プリコンパイラは、Embedded SQL 文を C のデータ宣言文と呼び出し文に変換します。プリコンパイル後に作成されるソース・プログラムは、普通の C プログラムと同様にコンパイルできます。

プリコンパイラは、アプリケーションを 2 段階で処理します。1 段階目の処理では、Embedded SQL 文と変数宣言を解析し、構文をチェックして、エラーを検出した場合にはメッセージを表示します。重大なエラーが検出されなければ、次のような 2 段階目の処理に移ります。

- “_sql” で始まるプリコンパイラ変数の宣言を追加します。混乱を避けるため、自分で作成する変数には “_sql” で始まる名前を付けないでください。
- 元の Embedded SQL 文のテキストをコメントに変換します。
- プリコンパイル・コマンドにストアド・プロシージャの生成および呼び出しオプションを付けた場合は、ストアド・プロシージャを生成し、呼び出します。

- Embedded SQL 文を Client-Library 呼び出しに変換します。Embedded SQL は、Client-Library をランタイム・ライブラリとして使っています。
- プリコンパイラは 3 種類のファイルを作成します。「ターゲット・ファイル」、オプションの「リスティング・ファイル」、オプションの「isql スクリプト・ファイル」の 3 つです。

注意 プリコンパイラのコマンド・ライン・オプションの詳細については、使用しているプラットフォームの『Open Client/Server プログラマーズ・ガイド補足』を参照してください。

複数の Embedded SQL ソース・ファイル

Embedded SQL を使用したアプリケーションが、複数のソース・ファイルで構成される場合には、次の規則が適用されます。

- 接続名はユニークで、アプリケーション全体でグローバル
- カーソル名は接続ごとでユニーク
- 準備文名は接続に対してグローバル
- 動的記述子はアプリケーションに対してグローバル

プリコンパイラの互換性

Embedded SQL バージョン 12.5 以降は、ANSI SQL-89 標準に完全に準拠しています。したがって、他の ANSI-89 標準準拠のプリコンパイラと互換性があります。

プリコンパイラの生成ファイル

ターゲット・ファイルは、元の入力ファイルに似ていますが、すべての SQL 文が Client-Library の実行時呼び出しに変換されています。

リスティング・ファイルには、入力ファイルの元の文に加えて、付加的な情報、警告、またはエラー・メッセージなどが含まれています。

isql スクリプト・ファイルには、プリコンパイラが生成したストアド・プロシージャが含まれています。ストアド・プロシージャは、Transact-SQL で記述されています。

第 2 章

一般情報

この章では、Embedded SQL の一般的な情報について説明します。

Embedded SQL プログラムの 5 つのタスク

Embedded SQL プログラムはホスト言語のコードを含めることに加え、さらに 5 つのタスクを実行します。正しくプリコンパイル、コンパイル、実行するには、Embedded SQL プログラムはこれら 5 つのタスクをすべて実行しなければなりません。これら 5 つのタスクについては、あとの章で説明します。

1 SQLCA、SQLCODE、または SQLSTATE を使用して SQL 通信を確立します。

SQL 通信領域 (SQLCA、SQLCODE、または SQLSTATE) を設定して、アプリケーション・プログラムと Adaptive Server Enterprise 間の通信パスを提供します。これらの構造体には、Adaptive Server Enterprise と Client-Library が生成するエラー・メッセージ、警告メッセージ、情報メッセージのコードが入っています。「第 3 章 Adaptive Server Enterprise との通信」を参照してください。

2 変数を宣言します。

- Embedded SQL 文で使用されるホスト変数をプリコンパイラに示します。「第 4 章 変数の使い方」を参照してください。

3 Adaptive Server Enterprise に接続します。

アプリケーションを Adaptive Server Enterprise に接続します。「第 5 章 Adaptive Server Enterprise への接続」を参照してください。

4 Transact-SQL 文を Adaptive Server Enterprise に送信します。

- Transact-SQL 文を Adaptive Server Enterprise に送信して、データの定義と操作を行います。「第 6 章 Transact-SQL 文の使い方」を参照してください。

5 エラーを処理してコードを返します。

- SQLCA、SQLCODE、または SQLSTATE を使用して Client-Library と Adaptive Server Enterprise が返したエラーの処理とレポートを行います。「第 8 章 エラーの処理」を参照してください。

簡単な Embedded SQL プログラム

次に、簡単な Embedded SQL プログラムを示します。ここでは、このプログラムをすべて理解する必要はありません。プログラムを示した目的は、Embedded SQL プログラムの構成を説明するためです。詳細は、あとの章で説明します。

```
/* Establishing a communication area - Chapter 3 */

 exec sql include sqlca;

 main()
 {

 /* Declaring variables - Chapter 4 */

 exec sql begin declare section;
 CS_CHAR  user[31], passwd[31];
 exec sql end declare section;

 /*Initializing error-handling routines - Chapter 8 */

 exec sql whenever sqlerror call err_p();

 /*Establishing Adaptive Server Enterprise connections -
Chapter 5 */

 printf("¥nplease enter your userid ");
 gets(user);
 printf("¥npassword ");
 gets(passwd);
 exec sql connect :user identified by :passwd;

 /* Issuing Transact-SQL statements  -  Chapter 6 */

 exec sql update titles set price = price * 1.10;
exec sql commit work;

 /* Closing server connections - Chapter 5 */

 exec sql disconnect all;
 }

/* Error-handling routines - Chapter 8 */
```

```
err_p()
{
    /* Print the error code and error message */

 printf("¥nError occurred: code %d.¥n%s",
    sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
}
```

Embedded SQL の一般的な規則

C プログラム内の Embedded SQL 文には、次のような規則が適用されます。

- Embedded SQL 文は、次のキーワードで始めます。

  ```
  exec sql
  ```

- Embedded SQL 文は、セミコロンで終了します。

  ```
  exec sql sql_statement;
  ```

- C のラベルが先行する場合を除き、exec sql は行の先頭に置きます。

  ```
  [label:] exec sql sql_statement;
  ```

- Embedded SQL のキーワードには、大文字と小文字の区別がありません。exec sql、EXEC SQL、Exec Sql などのほか、大文字と小文字を自由に混在させることができます。このマニュアルでは、Embedded SQL のキーワードは次のように小文字で統一して表記しています。次に例を示します。

  ```
  exec sql commit work;
  ```

文の位置

アプリケーション・プログラム内で C の文が有効な場所であれば、どこにでも Embedded SQL 文が使えます。

コメント

Embedded SQL や C の文の内部にコメントを入れる場合、次の 2 つの規約のうちのいずれかに従います。

Transact-SQL の規約は、次のとおりです。

```
/* comments */
```

ANSI の規約は、次のとおりです。

```
-- comments
```

SQL 文の外部で書かれるコメントは、C の規約に準拠します。

識別子

識別子は、アプリケーション内で関数名または変数名として使用されます。

引用符

Embedded SQL 文中のリテラル文字列は、一重引用符または二重引用符で囲みます。文字列を二重引用符で始めたときは、二重引用符で終了します。文字列を一重引用符で始めたときは、一重引用符で終了します。

予約語

C、Transact-SQL、または Embedded SQL の予約語は、言語本来の意味以外に使用しないでください。

Embedded SQL のキーワードは、大文字でも小文字でも、両者が混在していてもかまいません。このマニュアルでは、Embedded SQL のキーワードを小文字で示します。

変数の命名規約

Embedded SQL の変数は、C の命名規約に準拠します。一対の引用符内には、変数名を入れないでください。変数名が実際の値で置き換えられる際に、自動的に適切な引用符が挿入されます。

プリコンパイラがアプリケーションを解析する際に、プリコンパイラ変数に宣言が追加されます。これらの変数は、“_sql” で始まります。混乱を避けるために、“_sql” で始まる変数名は使用しないようにしてください。

スコープの規則

Embedded SQL 文と、プリコンパイラが生成する文のスコープは、「ホスト言語」のスコープ規則に準拠します。ただし、whenever 文とカーソル名は例外です。

文のバッチ処理

Transact-SQL と同じように、複数の SQL 文は、1 つの exec sql 文にまとめてバッチ処理ができます。アプリケーションが実行されるたびに同じ Transact-SQL を実行しなければならないときは、バッチ処理を行えば便利で効率もよくなります。

たとえば、あるアプリケーションでは、起動時にテンポラリ・テーブルとテンポラリ・インデックスを作成するとします。これらの文は、1 つのバッチで送信できます。文のバッチ処理の規則については、『ASE リファレンス・マニュアル』を参照してください。

次のような制約が、文のバッチ処理に適用されます。

- バッチ内の文からプログラムに結果を返すことはできません。つまり、バッチに select 文は使えません。
- バッチ内のすべての文は、有効な Transact-SQL 文でなければなりません。declare cursor や prepare といった Embedded SQL 文は、バッチ内では使えません。
- Transact-SQL バッチとまったく同じ規則が Embedded SQL バッチにも適用されます。たとえば、use database 文は Embedded SQL バッチでは使えません。

Embedded SQL の構文

表 2-1 には、Embedded SQL 文の有効な構文が示されています。

表 *2-1: Embedded SQL* の構文

begin declare section	dump database
begin tran	dump tran
begin work	end declare section
checkpoint	exec *procedure_name*
close *cursor_name*	execute *name*
commit tran	execute immediate
commit work	fetch *cursor_name*
connect	grant
create database	include sqlca または include *filename*
create default	insert
create table	open *cursor_name*
create index	prepare *statement_name*
create unique index	revoke
create clustered index	rollback tran
create nonclustered index	rollback work
create unique clustered index	select
create unique nonclustered index	set
create proc	truncate
create rule	update
create trigger	use
create view	whenever *condition action*
declare cursor	
delete	
disconnect	
drop table | default | index | proc | rule | trigger | view	

第 3 章

Adaptive Server Enterprise との通信

この章では、アプリケーション・プログラムがどのようにして Adaptive Server Enterprise からステータス情報を受け取るかについて説明します。

通信パスを作成し、Adaptive Server Enterprise からアプリケーションへの通信で使われるシステム変数を宣言するには、次の 3 つのうちのどれかを作成してください。

- SQLCODE を持つ、SQL 通信領域 (SQLCA)
- スタンドアロンの SQLCODE long integer
- SQLSTATE 文字配列

SQLCODE、SQLCA、SQLSTATE は、Adaptive Server Enterprise からアプリケーションへの通信時に使用する変数です。

Adaptive Server Enterprise は、Embedded SQL 文を実行するたびに、リターン・コードを SQLCA、SQLCODE、または SQLSTATE 内に格納します。アプリケーション・プログラムは、その変数にアクセスすると、実行した SQL 文が成功したか失敗したかを調べることができます。

注意 プリコンパイラは、自動的に SQLCA、SQLCODE、SQLSTATE 変数を設定します。これらの変数はデータベースへのランタイム・アクセスに非常に重要であり、ユーザがこれを初期化または変更する必要はありません。

エラーの検出と処理、複数のエラー・メッセージやその他のリターン・コードに関する詳細については、「第 8 章 エラーの処理」を参照してください。

スコープの規則：SQLCA、SQLCODE、および SQLSTATE

アプリケーション・プログラムで C の変数を宣言できる場所であればどこでも、SQLCA を宣言できます。この構造体のスコープは、C のスコープ規則に従います。

使用しているファイル内で SQLCA、SQLCODE、または SQLSTATE を宣言している場合、各変数はそのファイルにあるすべての実行可能な Embedded SQL 文のスコープ内になければなりません。プリコンパイラは各 Embedded SQL 文に対して、これらのステータス変数のそれぞれを設定するコードを生成します。そのため、変数がスコープ内にない場合は、生成されたコードはコンパイルされません。

プリコンパイラに渡されるファイル内で SQLCA、SQLCODE、または SQLSTATE が宣言されていない場合は、参照先ファイル内で SQLCODE を宣言しなければなりません。プリコンパイラは SQLCODE の宣言を前提としてコードを作成します。

SQLCA の宣言

警告！ SQLCODE や SQLCA よりも SQLSTATE を使用する方が望ましいのですが、このバージョンで完全にサポートしているのは SQLCODE のみです。今後のバージョンでは SQLSTATE をサポートする予定です。

次に、SQLCA の宣言の構文を示します。

```
exec sql include sqlca;
```

Embedded SQL の include 文は、C のプリプロセッサの #include コマンドを使用した場合と同様に、アプリケーションに他のファイルをインクルードできます。プリコンパイラのコマンド・オプションを設定して、*include* ファイルのディレクトリを指定できます。プリコンパイラは、プリコンパイル時に C のコンパイル・コマンドで指定されたパスを検索します。また、プリコンパイラはこのファイルを検索するのに *include* ファイルのパスを使用します。これは、インクルード・ファイルを main ファイルの一部としてオープンし、読み込みます。インクルード・ファイルが見つからない場合、プリコンパイラは失敗します。

複数の SQLCA

SQLCA が複数ある場合でも、それらはすべてホスト変数に対する C のスコープ規則に従います。各 SQLCA は、別々のスコープ内にある必要はありません。

SQLCA 変数

プリコンパイラは include sqlca 文を検出すると、アプリケーション・プログラムに SQLCA 構造体の宣言を挿入します。SQLCA とはプリコンパイラが定める「システム変数」を持つデータ構造体であり、各システム変数は独立してアクセスされます。ユーザのアプリケーション・プログラムは、これらの変数を直接変更してはいけません。

SQLCA 変数は、直前に実行した Embedded SQL 文のステータスに関する情報をアプリケーション・プログラムへ渡します。

表 3-1 では、Adaptive Server Enterprise が生成するステータス情報、リターン・コード、エラー・コード、エラー・メッセージを保持する SQLCA 変数について説明します。

表 *3-1: Adaptive Server Enterprise の SQLCA* 変数

変数	データ型	説明
sqlcaid	char	“sqlca” を含む文字列
sqlcabc	long	SQLCA の長さ
sqlcode	long	直前に実行した SQL 文のリターン・コードを含む。 リターン・コードの定義については、SQLCODE 値を参照。
sqlwarn[0] to *sqlwarn[7]*	char	警告フラグ。個々のフラグは警告が発行されたかどうかを示す。‘W’ は警告あり、スペースは警告なしを意味する。 *sqlwarn* フラグについては第 8 章を参照。
sqlerrm.sqlerrmc[ ]	char	エラー・メッセージ
sqlerrm.sqlerrml	long	エラー・メッセージの長さ
sqlerrp	char	エラー／警告を検出したプロシージャ
sqlerrd[6]	long	エラー／警告の詳細。[2] は影響を受けたローの数。

SQLCA 変数へのアクセス

SQLCA 変数は、include sqlca 文が宣言する C 構造体 sqlca のメンバです。SQLCA 変数にアクセスするには、C の構造体のメンバ演算子 (.) を使用してください。次に、その例を示します。

```
if (sqlca.sqlwarn[1] == 'W')
 {
    printf("¥nData truncated");
 return;
 }
```

sqlca 構造体のアドレスを関数に渡し、その関数の内部で -> 演算子を使うと SQLCA 変数にアクセスできます。次に、そのように動作する関数の例を示します。

```
warning(p)
 struct sqlca *p;
 {
 if (p->sqlwarn[3] == 'W')
 {
    printf("¥nIncorrect number of variables in
 fetch.¥n");
 }
 return;

 }
```

SQLCA 変数は Embedded SQL 文の実行が成功したかどうかを判断するのに便利です。前の項で挙げた SQLCA 変数は、エラーとリターン・コードについてより多くの情報を提供し、アプリケーションのデバッグと通常の処理に役立ちます。

SQLCA 内の SQLCODE

Adaptive Server Enterprise は実行ごとに sqlcode を更新するので、アプリケーションは文の実行ごとに sqlcode をテストしなければなりません。これには、たいていの場合、「第 8 章 エラーの処理」で説明する whenever 文を使用します。

スタンドアロン領域としての SQLCODE の宣言

警告！ SQLCODE や SQLCA よりも SQLSTATE を使用する方が望ましいのですが、このバージョンで完全にサポートしているのは SQLCODE のみです。今後のバージョンでは SQLSTATE を完全にサポートする予定です。

SQLCA を作成する代わりに、SQLCODE を独立して使用する方法があります。これには、直前に実行した SQL 文のリターン・コードが含まれています。SQLCODE をスタンドアロン領域として宣言する利点は、コードを実行する速度が向上する点です。SQLCA が持つ他の情報を見る必要がなく、リターン・コードだけを知りたい場合は、SQLCODE を使用することもできます。

しかし、SQLCODE は Embedded SQL の前のバージョンとの互換性を保つためにサポートされているので、SQLCODE　の実行速度の方が高速ですが、SQLCODE ではなく、SQLSTATE の使用をおすすめします。

注意　今後のバージョンでは、ステータス結果を受け取るのに SQLCODE ではなく SQLSTATE を使用することをおすすめします。

次は、スタンドアロン領域として SQLCODE を宣言する例です。

```
long SQLCODE;
exec sql open cursor pub_id;
     while (SQLCODE == 0)
     {
     exec sql fetch pub_id into :pub_name;
```

SQLCODE が示すエラーのデバッグの詳細については、「第 8 章 エラーの処理」を参照してください。

表 3-2 に SQLCODE の値を示します。

表 *3-2: SQLCODE* の値

値	説明
0	文の実行が成功した。
-n	エラーが発生した。サーバまたは Client-Library のエラー・メッセージを参照。“*-n*” は、エラーまたは例外の数を表す。
+100	データが存在しない、fetch のあとにローが残っていない、または update、delete、insert の探索条件に合うローが存在しない。

SQLSTATE の使用

警告！　SQLCODE や SQLCA よりも SQLSTATE を使用する方が望ましいのですが、このバージョンで完全にサポートしているのは SQLCODE のみです。今後のバージョンでは SQLCA および SQLSTATE の両方を完全にサポートする予定です。

SQLSTATE はステータス・パラメータです。そのコードは、直前に実行されたプロシージャのステータスを示します。つまり、そのプロシージャが正常に完了したか、またはそのプロシージャの実行中にエラーが発生したかを示します。

SQLSTATE は文字列パラメータです。その例外値を表 3-3 に示します。

表 *3-3: SQLSTATE* の値

値	説明
00XXX	正常に実行された
01XXX	警告
02XXX	データが存在しない。影響を受けたローがない。
その他の値	エラー

SQLSTATE コードおよびエラー・メッセージの取得

SQLSTATE メッセージには、情報、警告、重大なエラー、および致命的エラーのメッセージがあります。Adaptive Server Enterprise および Open Client Client-Library は、SQLSTATE メッセージを生成します。SQLSTATE コードおよびエラー・メッセージの全リストについては、適切なマニュアルを参照してください。

プリコンパイラが生成する SQLSTATE メッセージの表については、「付録 A プリコンパイラの警告とエラー・メッセージ」を参照してください。

まとめ

この章では、SQLCA、SQLCODE、SQLSTATE について説明しました。文の実行後、Adaptive Server Enterprise はリターン・コードと情報を SQLCA 変数、スタンドアロン SQLCODE 領域、または SQLSTATE に格納します。これらのリターン・コードは、直前に実行した文の成功または失敗を表します。

第 4 章

変数の使い方

この章では、アプリケーションと Adaptive Server Enterprise 間でデータを渡す、次の 2 種類の変数について説明します。

- ホスト変数：Embedded SQL 文で使用する C 言語の変数で、Adaptive Server Enterprise から取得したデータや Adaptive Server に送信するデータを格納します。
- インジケータ変数：ホスト変数に関連する null データやデータのトランケートを示す変数です。

変数の宣言

第 3 章で説明したように、アプリケーション・プログラムに SQLCA、SQLCODE、SQLSTATE が含まれていると、プリコンパイラが自動的にシステム変数を設定します。ただし、Embedded SQL 文を使う前に、declare セクションでホスト変数とインジケータ変数を明示的に宣言してください。

警告！ プリコンパイラによって作成される一部の変数は、すべて “_sql” で始まっています。自分で作成する変数には、先頭に “_sql” を付けないでください。エラー・メッセージや不正確なデータを受け取ることがあります。

プリコンパイラは、declare セクション内のマクロと #include 文を無視します。include 文は、プリコンパイルされているファイルにインクルード・ファイルの内容が直接コピーされたかのように処理します。include 文を使った declare セクションの構文は、次のとおりです。

```
exec sql begin declare section;
 exec sql include "filename";
 ...
 exec sql end declare section;
```

ホスト変数の宣言は、C 言語の変数宣言の規則に従ってください。プログラム内の declare セクションの数は無制限なので、すべての変数を同じ declare セクションで宣言する必要はありません。

変数を宣言するときは、「データ型」を指定してください。有効なデータ型については、「データ型と Adaptive Server Enterprise」(33 ページ) を参照してください。または、declare セクションの *cspublic.h* ファイルで宣言された CS_CHAR など、Client-Library の typedef を使用することもできます。

次の例では、declare セクションに定義された 2 つの文字列を示します。

```
exec sql begin declare section;
 CS_CHAR name[20];
 CS_CHAR type[3];
 exec sql end declare section;
```

ホスト変数を宣言するときは、次のようなスカラ変数の場合にのみ、そのホスト変数を初期化できます。

```
exec sql begin declare section;
     int total = 0;
 exec sql end declare section;
```

配列を宣言の中で初期化することはできません。

データ型の使い方

Embedded SQL では、C のデータ型 char、int、float、double、void を使用できます。キーワード const と volatile を使用できますが、構造体とともに使用することはできません。また、キーワード unsigned、long、short も使用できます。記憶クラス指定子として、auto、extern、register、static を使用できます。

注意 64 ビット・アプリケーションを構築する場合は、long int は使用しないでください。

```
exec sql begin declare section;
     register int frequently_used_host_variable;
     extern char
     shared_string_host_variable[STRING_SIZE];
     /*
```

```
        ** The const restriction is not enforced by
        ** the precompiler; only the compiler makes use
        ** of it.
        */
        const float
        input_only_host_variable = 3.1415926;
        /*
        ** Be careful.  You can declare unsigned
        ** integers, but if you select a negative
        ** number into one, you will get an incorrect
        ** result and no error message.
        */
        unsigned long int unsigned_host_variable;
    exec sql end declare section;
```

declare セクションでポインタを宣言できますが、Embedded SQL 文でホスト変数としてポインタを使用することはできません。

```
exec sql begin declare section;
 int number;
 /*
 ** It's convenient to declare this here,
 ** but we won't be using it as a host variable.
 */
 int *next_number;
 exec sql end declare section;
```

次のような Sybase データ型を使用できます。

CS_BINARY、CS_BIT、CS_BIGINT、CS_BOOL、CS_CHAR、CS_DATE、CS_DATETIME、CS_DATETIME4、CS_DECIMAL、CS_FLOAT、CS_REAL、CS_IMAGE、CS_INT、CS_MONEY、CS_MONEY4、CS_NUMERIC、CS_RETCODE、CS_SMALLINT、CS_TEXT、CS_TIME CS_TINYINT、CS_UBIGINT、CS_UINT、CS_UNICHAR、CS_UNITEXT、CS_USMALLINT、CS_VOID、CS_XML。

CS_CHAR は char とは処理方法が異なります。CS_CHAR の場合は null で終了しますが、ブランクは埋め込まれません。それに対して char の場合は null で終了しますが、その配列の長さまでブランクが埋め込まれます。

```
/*
 ** Your #define for the array size doesn't
 ** have to be in the declare section,
 ** though it would be legal if it were.
 */
 #define MAX_NAME 40;

 exec sql begin declare section;
     CS_MONEY salary;
     CS_CHAR print_this[MAX_NAME];
     char print_this_also[MAX_NAME];
 exec sql end declare section;
```

```
exec sql select salary into :salary from salaries
    where employee_ID = '01234';
/*
** The CS_MONEY type is not directly printable.
** Here's an easy way to do a conversion.
*/
exec sql select :salary into :print_this;

/*
** This will not be blank-padded.
*/
printf("Salary for employee 01234 is %s.¥n",
    print_this);

/*
** This will be blank-padded.
*/
exec sql select :salary into :print_this_also;
printf("Salary for employee 01234 is %s.¥n",
    print_this_also);
```

タイプ定義の使い方

declare セクション内でタイプ定義 (typedef) を使うと、変数を宣言できます。次に例を示します。

```
exec sql begin declare section;
    /*
    ** The typedef and the use of the typedef
    ** can be in separate declare sections
    ** if the typedef comes first.
    ** The typedef can even be in an “exec
    ** sql include file”.
    */
    typedef int STORE_ID;
    STORE_ID current_ID;
exec sql end declare section;

exec sql select store_ID into :current_ID
    from sales_table where
    store_name = 'Furniture Kingdom';
```

データ型定義および制限

表 4-1 には、Embedded SQL の有効なデータ型定義が示されています。

表 *4-1: 有効な typedef*

typedef	説明
CS_BINARY	バイナリ型
CS_BIT	ビット型
CS_CHAR	文字型
CS_DATE	日付型
CS_TIME	時刻型
CS_DATETIME	日時型
CS_FLT8	8 バイトの浮動小数点型
SQLINDICATOR	インジケータ変数 (2 バイト整数) に使用
CS_INT	4 バイト整数
CS_BIGINT	8 バイト整数
CS_MONEY	通貨型
CS_SMALLINT	2 バイト整数
CS_TINYINT	1 バイトの符号なし整数
CS_SMALLINT	2 バイト整数
CS_USMALLINT	2 バイトの符号なし整数
CS_UINT	4 バイトの符号なし整数
CS_UBIGINT	8 バイトの符号なし整数
CS_TEXT	文字列型
CS_IMAGE	イメージ型
CS_UNICHAR	UTF16 Unicode 文字型
CS_UNITEXT	UTF16 Unicode 文字列型
CS_XML	xml データ

Embedded SQL では、すべての基本的な ANSI データ型定義も有効です。

実装制限

exec sql include *filename* のネストの深さの制限は 32 です。

#define の使い方

declare セクションで #define 値を使うと、配列にしたり、変数を初期化したりすることができます。ホスト変数の宣言で #define を使うときは、これを使うホスト変数の宣言より前に置きます。次に、有効な 2 つの例を示します。

```
#define PLEN  26
 CS_CHAR name[PLEN];
```

および

```
exec sql begin declare section;
 #define  PLEN  26
 exec sql end declare section;
     ...
 exec sql begin declare section;
 CS_CHAR  name[PLEN];
 exec sql end declare section;
```

#define を使用して、記号名を宣言できます。これは、アプリケーションで使う前に宣言します。たとえば、“10” を記号的に定義するには、次のような名前を使用します。

```
exec sql begin declare section;
 #define count_1 10
 CS_CHAR var1[count_1];
 exec sql end declare section;
```

配列の宣言

プリコンパイラは、「複雑な定義」、つまり構造体と配列をサポートしています。構造体はネストできますが、構造体の「配列」はできません。

プリコンパイラは、あらゆるデータ型の 1 次元配列を認識します。

また、次の例のように、プリコンパイラは文字の 2 次元配列も認識します。

```
#define maxrows 25
 int numsales [maxrows];
exec sql begin declare section;
 #define DATELEN 30
 #define DAYS_PER_WEEK 7
 CS_CHAR days_of_the_week[DAYS_PER_WEEK][DATELEN+1];
 exec sql end declare section;
```

配列は、どのようなデータ型でも宣言できます。ただし、ひとつの配列要素に SELECT INTO するには、そのデータ型がスカラ、つまり整数、文字、浮動小数点、またはポインタである必要があります。また、次のように、複数の配列要素に ELECT INTO する場合は、スカラ配列だけでなく、構造体の配列も可能です。

```
exec sql begin declare section;
     int sales_totals[100];
     struct sales_record{
         int total_sales;
         char store_name[40];
         } sales_records [100];
 exec sql end declare section;

 /*
 ** If there are fewer than 100 stores,
 ** this will get the sales totals for all
 ** of them.  If there are more than
 ** 100, it will cause an error at runtime.
 */
 exec sql select total_sales into :sales_totals
     from sales_table;
 /*
 ** This gets the sales for just one store.
 */
 exec sql select total_sales into :sales_totals[0]
     from sales_table where store_ID = 'xyz';
 /*
 ** This gets two pieces of information on a single **
store.
 */
 exec sql select total_sales, store_name
     into :sales_records[i]
     from sales_table where store_ID = 'abc';
```

文字配列の宣言

文字配列のタイプは CS_CHAR でも char[] でもかまいませんが、これら 2 つのデータ型を制御する規則は異なります。入力としてタイプ char[] の配列を使用した場合、プリコンパイラはその配列が null 文字で終了しているかどうかチェックします。配列が null で終了しない場合は、プリコンパイラのランタイム機能によってエラーが返されます。これに対してタイプ CS_CHAR の配列の場合は、null で終了するかどうかはチェックしません。反対にその入力の長さは、null 文字がある場合にはその null 文字まで、または配列の宣言された長さまでのどちらか短い方になります。

出力としてタイプ char[] の配列を使用した場合は、スペース文字 (ブランク) が埋め込まれて null で終了します。タイプ CS_CHAR の配列の場合は、null で終了するだけでブランクは埋め込まれません。

文字配列はスカラです。これは単一の文字列を表すからです。したがって、ある文字配列を選択して単一の文字列だけを取得することができます。またその他のデータ型の配列とは異なり、文字配列はホスト入力変数の場合もあります。

「配列の使い方」(32 ページ) を参照してください。

共用体と構造体の宣言

共用体と構造体は直接宣言するか、またはタイプ定義 (typedef) を使用して宣言できます。共用体の 1 つの要素をホスト変数として使用することはできますが、共用体を全体として使用することはできません。これに対してホスト変数はある構造体全体の場合や、その構造体の要素のうちの 1 つだけの場合もあります。次は、共用体と構造体の宣言の例です。

```
exec sql begin declare section;
     typedef int PAYMENT_METHOD;
     PAYMENT_METHOD method;
     union salary_or_percentage {
         CS_MONEY salary;
         CS_NUMERIC percentage;
         } amount;
      struct employee_record {
         char first_name[30];
         char last_name[30];
         char employee_ID[30];
         } this_employee;
       char *employee_of_the_month_ID = "01234567";
 exec sql end declare section;

 exec sql select first_name, last_name, employee_ID
     into :this_employee
     from employee_table
     where employee_ID = :employee_of_the_month_ID;
 exec sql select payment_type into :method
     from remuneration_table where employee_ID =
     :this_employee.employee_ID;
 switch (method) {
     case SALARIED:
         exec sql select salary into
         :amount.salary
         from remuneration_table
         where employee_ID =
         this_employee.employee_ID;
         break;
     case VOLUNTEER:
         exec sql select 0 into
         :amount.salary
         break;
     case COMMISSION:
         exec sql select commission_percentage into
         :amount.percentage
         from remuneration_table
         where employee_ID =
         this_employee.employee_ID;
         break;
     }
```

ホスト変数の使い方

ホスト変数を使うと、Adaptive Server Enterprise とアプリケーション・プログラム間で値を転送できます。

ホスト変数は、アプリケーション・プログラムの Embedded SQL の declare セクションで宣言します。宣言しないと SQL 文で変数を使うことはできません。

Embedded SQL 文の中でホスト変数を使うときは、変数の前にコロンを付けます。プログラム内のその他の箇所でこの変数を使うときは、コロンは使わないでください。Embedded SQL 文の中で複数のホスト変数を続けて使うときは、それぞれの変数の間をカンマで区切るか、SQL 文の文法規則に従ってください。

次の例は、変数の使い方を示しています。declare セクションでは *user* が文字変数として宣言されています。次にこれを、select 文の中でホスト変数として使っています。

```
exec sql begin declare section;
 CS_CHAR  user[32];
 exec sql end declare section;

 exec sql select user_name() into :user;
 printf("You are logged in as %s.¥n", user);
```

ホスト変数には、次のような 4 とおりの使い方があります。

- SQL 文およびプロシージャのための入力変数
- 結果変数
- SQL プロシージャの呼び出しによるステータス変数
- SQL 文およびプロシージャの出力変数

関数に関係なく、「変数の宣言」(19 ページ) に記載されているあらゆるホスト変数を宣言してください。次に、ホスト変数の使い方を説明します。

ホスト入力変数

これらの変数は、情報を Adaptive Server Enterprise に渡します。アプリケーション・プログラムは、これらの変数に値を代入します。ここに格納されたデータは、ストアド・プロシージャ、where 句のある select 文、values 句のある insert 文、set 句のある update 文などの実行文に使われます。

次の例では、*id* と *publisher* 変数を入力変数として使っています。

```
exec sql begin declare section;
     CS_CHAR id[7];
     CS_CHAR publisher[5];
 exec sql end declare section;
     ...
 exec sql delete from titles where title_id = :id;
 exec sql update titles set pub_id = :publisher
     where title_id = :id;
```

ホスト結果変数

これらの変数は、select 文と fetch 文の結果を受け取ります。

次の例では、変数 *id* を「結果変数」として使っています。

```
exec sql begin declare section;
 CS_CHAR  id[5];
 exec sql end declare section;

 exec sql select title_id into :id from titles
     where pub_id = "0736" and type = "business";
```

ホスト・ステータス変数

これらの変数は、ストアド・プロシージャのリターン・ステータス値を受け取ります。ステータス変数は、ストアド・プロシージャが正常に終了したかどうか、また失敗した場合はその理由を示します。

ステータス変数を 2 バイトの整数 (CS_SMALLINT) として宣言します。

次の例では、変数 *retcode* を「ステータス変数」として使っています。

```
exec sql begin declare section;
 CS_SMALLINT  retcode;
 exec sql end declare section;

 exec sql begin transaction;
 exec sql exec :retcode = update_proc;
 if (retcode != 0)
 {
     exec sql rollback transaction;
 }
```

ホスト出力変数

ホスト出力の変数は、ストアド・プロシージャから受け取ったデータをアプリケーション・プログラムへ渡します。ストアド・プロシージャが out を使用して宣言したパラメータ値を返すときには、ホスト出力変数を使います。

次の例では、変数 *par1* と *par2* を出力変数として使っています。

```
exec sql exec a_proc :par1 out, :par2 out;
```

インジケータ変数の使い方

インジケータ変数をホスト変数と対応付けると、データベースの値が null であることを示すことができます。スペースと、オプションの indicator キーワードを使って、各インジケータ変数を対応するホスト変数と区切ります。「インジケータ変数」は、それぞれ対応するホスト変数の直後に置いてください。

インジケータ変数がないと、Embedded SQL は NULL 値を示すことができません。

インジケータ変数およびサーバの制約

Embedded SQL は一般的なインタフェースであり、Adaptive Server Enterprise の他にも、さまざまなサーバで実行できます。

このため、Embedded SQL は特定のサーバの制約を適用または反映することはありません。

Embedded SQL のアプリケーションを記述する場合、アプリケーションの対象となる「サーバ」の制約に注意してください。あるサーバについてどのような制約があるか不明なときは、そのサーバのマニュアルを参照してください。

インジケータ変数を使用したホスト変数

Embedded SQL を含むアプリケーション・プログラムでインジケータ変数を使う前に、declare セクションでホスト変数とインジケータ変数を宣言します。インジケータ変数は、使う前に declare で 2 バイト整数 (short または CS_SMALLINT) として宣言します。

Embedded SQL 文でこの変数を使うときは、インジケータ変数の前にコロンを付けてください。

インジケータ変数をホスト変数と対応付ける構文は、次のとおりです。

```
:host_variable [[indicator] :indicator_variable]
```

インジケータ変数とホスト変数の対応関係は、文中、つまり 1 つの exec sql 文の中、または open カーソル文から close カーソル文までの間でのみ有効です。ホスト変数に値が代入されると同時に、インジケータ変数にも値が代入されます。

Adaptive Server Enterprise がインジケータ変数を設定するのは、ホスト変数に値を代入したときだけです。したがって、インジケータ変数を宣言したあとで、別の文で別のホスト変数とともに再使用できます。

インジケータ変数は、出力変数、結果変数、および入力変数とともに使用できます。出力変数や結果変数とともに使用する場合、Embedded SQL は、この変数を対応するホスト変数の NULL ステータスを示すように設定します。入力変数とともに使用する場合は、インジケータ変数の値を Adaptive Server Enterprise に送出される前の「入力変数」の null ステータスを示すように設定してください。

ホスト出力変数と結果変数に対応するインジケータ変数の使い方

インジケータ変数を出力変数または結果変数に対応付けると、Client-Library は、自動的にこの変数を表 4-2 に示す次のいずれかの値に設定します。

表 *4-2:* 出力変数または結果変数と併用したインジケータ変数の値

値	意味
-1	対応する Adaptive Server Enterprise のデータベース・カラムに null 値がある。
0	ホスト変数に null 以外の値が代入された。
>0	ホスト変数のデータ変換中にオーバフローが発生した。ホスト変数には、トランケートされたデータが代入される。正の数は、トランケートされる前の値の長さをバイト単位で表している。

次の例は、インジケータ変数 *indic* と結果変数 *id* の対応関係を示しています。

```
exec sql begin declare section;
 CS_CHAR          id[6];
 CS_SMALLINT      indic;
 CS_CHAR          pub_name[41];
 exec sql end declare section;

 exec sql select pub_id into :id indicator :indic
     from titles where title
     like "%Stress%";

 if (indic == -1)
 {
     printf("¥npub_id is null");
 }
 else
 {
     exec sql select pub_name into :pub_name
         from publishers where pub_id = :id;
     printf("¥nPublisher: %s", pub_name);
 }
```

インジケータ変数を使用したホスト入力変数

インジケータ変数を入力変数に対応付けるときは、表 4-3 の値を参考にインジケータ変数を明示的に設定してください。

表 *4-3:* 入力変数と併用したインジケータ変数の値

値	意味
-1	対応する入力を NULL 値として扱う。
0	ホスト変数の値をカラムに代入する。

入力値が null かどうかを調べてインジケータ変数に -1 を代入することは、ホスト言語コード側で行う必要があります。これによって、null 値があることが Client-Library に通知されます。インジケータ変数を -1 に設定すると、ホスト変数の実際の値に関係なく null が使われます。

次の例は、インジケータ変数と入力変数の対応関係を示しています。*indic* が -1 に設定されているため、データベースの royalty カラムは null 値に設定されます。*indic* の値を変更すると、royalty の値も変化します。

```
exec sql begin declare section;
 CS_SMALLINT     indic;
 CS_INT          royalty;
 exec sql end declare section;

 indic = -1;
 exec sql update titles set royalty = :royalty
     :indic where pub_id = "0736";
```

ホスト変数の規約

「ホスト変数名」は、C の命名規則に準拠していなければなりません。

同じロケーションで Transact-SQL 文の Transact-SQL のリテラルを使用できる場所であればどこでも、Embedded SQL 文でホスト変数を使用できます。

ホスト変数は、プリコンパイラの有効なデータ型にしてください。ホスト変数のデータ型は、返されるデータベース・カラム値のデータ型と互換性があるものにします。詳細については、表 4-5 (35 ページ) と表 4-6 (36 ページ) を参照してください。ホスト言語の予約語と Embedded SQL のキーワードは、変数名として使用できません。

ホスト変数は、「動的 **SQL**」で指定される場合を除いて、Embedded SQL のキーワードやデータベース・オブジェクトを表すことはできません。「第 7 章 動的 SQL の使い方」 を参照してください。

ホスト変数が SQL 文の文字列を表す場合は、引用符で囲まないでください。

次に示す例は、プリコンパイラが必要に応じて値の前後に引用符を挿入するので、正しくありません。引用符は記述しないでください。

```
strcpy (p_id, "12345");
 exec sql select pub_id into :p_id from publishers
 where pub_id like ":p_id";
```

次の例は、正しい例です。

```
strcpy (p_id, "12345");
 exec sql select pub_id into :p_id from publishers
 where pub_id like :p_id;
```

配列の使い方

配列は、関連するデータの集まりを 1 つの変数にしたものです。配列は、select 文や fetch 文の into 句の出力変数として使用できます。次に例を示します。

```
exec sql begin declare section;
 CS_CHAR  au_array [100] [30];
exec sql end declare section;
exec sql
 select au_lname
 into :au_array
 from authors;
```

注意 単一の項目は、配列のどこにでもフェッチできます。しかし、複数ローは、配列の先頭にしかフェッチできません。

select と fetch into の使い方については、「配列を使用した複数ローの選択」(45 ページ) を参照してください。

複数の配列

1 つの SQL 文で複数の配列を使う場合、配列はすべてサイズが同じでなければなりません。サイズが異なっていると、エラー・メッセージが表示されます。

スコープの規則

プリコンパイラは、C 言語における変数のスコープ規則をサポートしています。ネストされたプログラム内で定義されたホスト変数では、変数名のほかに external 句を使用できます。次に例を示します。

```
FILE 1:
CS_CHAR  username[31]
 main()
 {
     sub1();
     printf("%s¥n", username);
 }
FILE 2:
void sub1()
 {
     exec sql begin declare section;
     extern char username[31];
     exec sql end declare section;
    exec sql select USER() into :username;
     return;
 }
```

データ型と Adaptive Server Enterprise

ホスト変数データ型は対応するデータベース・カラムのデータ型と互換性をもたせてください。したがって、アプリケーション・プログラムを記述する前に、データベース・カラムのデータ型を確認してください。ホスト変数と Adaptive Server Enterprise のデータ型の互換性を保つには、Sybase が提供するタイプ定義を使います。

表 4-4 は、等価のデータ型の簡単な説明です。Adaptive Server Enterprise の各データ型の詳細については、『ASE リファレンス・マニュアル』を参照してください。

表 *4-4: C 言語と Adaptive Server Enterprise のデータ型の互換性の比較*

Sybase 提供の typedef	説明	C 言語のデータ型	Adaptive Server Enterprise のデータ型
CS_BIGINT	8 バイトの整数型	long long	bigint
CS_BINARY	バイナリ型	unsigned char	binary, varbinary
CS_BIT	ビット型	unsigned char	boolean
CS_CHAR	文字型	char[n]	char、varchar
CS_DATE	4 バイトの日付型	なし	date
CS_TIME	4 バイトの時刻型	なし	time

Sybase 提供の typedef	説明	C 言語のデータ型	Adaptive Server Enterprise のデータ型
CS_DATETIME	8 バイトの日時型	なし	datetime
CS_DATETIME4	4 バイトの日時型	なし	smalldatetime
CS_BIGDATETIME	8 バイトのバイナリ型	なし	bigdatetime
CS_BIGTIME	8 バイトのバイナリ型	なし	bigtime
CS_TINYINT	1 バイトの符号なし整数型	unsigned char	tinyint
CS_SMALLINT	2 バイトの整数型	short	smallint
CS_INT	4 バイトの整数型	long	int
CS_DECIMAL	10 進数型	なし	decimal
CS_NUMERIC	数値型	なし	numeric
CS_FLOAT	8 バイトの浮動小数点型	double	float
CS_REAL	4 バイトの浮動小数点型	float	real
CS_MONEY	8 バイトの通貨型	なし	money
CS_MONEY4	4 バイトの通貨型	なし	smallmoney
CS_TEXT	文字列型 -y オプションが必要	unsigned char	text
CS_IMAGE	イメージ型 -y オプションが必要	unsigned char	image
CS_UBIGINT	8 バイトの符号なし整数型	unsigned long long	ubigint
CS_UINT	4 バイトの符号なし整数型	unsigned int	uint
CS_UNICHAR	2 バイトの UTF-16 Unicode 文字型	unsigned short	unichar
CS_UNITEXT	2 バイトの UTF-16 Unicode 文字列型	unsigned short	unitext
CS_USMALLINT	2 バイトの符号なし整数型	unsigned short	usmallint
CS_XML	XML 型	unsigned char	xml

データ型の変換

プリコンパイラは、自動的にホスト変数のデータ型を Adaptive Server Enterprise のテーブル・カラムのデータ型と比較します。Adaptive Server Enterprise のデータ型が、ホスト言語のデータ型と互換性はあるが同じ型ではない場合、プリコンパイラは、一方をもう一方と同じデータ型に変換します。プリコンパイラがデータを別のデータ型に変換できる場合、データ型には互換性があります。データ型に互換性がない場合は、実行時に変換エラーが発生し、sqlcode が <0 に設定されます。

4 バイトから 2 バイトへの変換など、長いデータ型を短いデータ型に変換するときは、データがトランケートされる可能性があるため、十分に注意してください。データがトランケートされた場合、sqlwarn1 は “W” に設定されます。

結果変数のデータ型の変換

表 4-5 は、結果変数に対するデータ変換が有効かどうかを示します。黒点は変換できることを示していますが、ホスト変数のデータ型を選択するときに注意しないと、エラーが発生します。

表 *4-5:* 結果変数のデータ型の変換

	変換先：Sybase データ型定義														
変換元：Adaptive Server Enterprise のデータ型	CS_TINYINT	CS_SMALLINT	CS_USMALLINT	CS_INT	CS_UINT	CS_BIGINT	CS_UBIGINT	CS_REAL	CS_CHAR	CS_UNICHAR	CS_MONEY	CS_DATE	CS_TIME	CS_DATETIME	CS_TEXT	CS_XML
char	•	•	•	•	•	•	•	•	•	•	•			•	•	•
unichar	•	•	•	•	•	•	•	•	•	•	•			•	•	•
varchar	•	•	•	•	•	•	•	•	•	•	•			•	•	•
bit	•	•	•	•	•	•	•	•	•	•	•				•	•
binary	•	•	•	•	•	•	•	•	•	•	•				•	•
tinyint	•	•	•	•	•	•	•	•	•	•	•				•	•
smallint	•	•	•	•	•	•	•	•	•	•	•				•	•
int	•	•	•	•	•	•	•	•	•	•	•				•	•
bigint	•	•	•	•	•	•	•	•	•	•	•				•	•
ubigint	•	•	•	•	•	•	•	•	•	•	•				•	•
uint	•	•	•	•	•	•	•	•	•	•	•				•	•
usmallint	•	•	•	•	•	•	•	•	•	•	•				•	•
float	•	•	•	•	•	•	•	•	•	•	•				•	•
money	•	•	•	•	•	•	•	•	•	•	•				•	•
date									•	•		•				
time									•	•			•			
datetime									•	•				•		
decimal	•	•	•	•	•	•	•	•	•	•	•				•	•
numeric	•	•	•	•	•	•	•	•	•	•	•				•	•
text	•	•	•	•	•	•	•	•	•	•	•				•	•
xml	•	•	•	•	•	•	•	•	•	•	•				•	•

入力変数のデータ型の変換

表 4-6 は、入力変数の有効なデータ変換を示しています。黒点は変換できることを示し、“X” は変換が必要なことを示しています。変換できないホスト変数のデータ型を選択すると、トランケートなどのエラーが発生します。

表 *4-6:* 入力変数のデータ型の変換

	変換先：Adaptive Server Enterprise のデータ型																		
変換元：C 言語のデータ型	**tinyint**	**bit**	**smallint**	**usmallint**	**int**	**uint**	**bigint**	**ubigint**	**float**	**char**	**unichar**	**money**	**date**	**time**	**datetime**	**decimal**	**numeric**	**text**	**xml**
unsigned char	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•
unichar	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•
short int	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•
long int	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•
bigint	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•
ubigint	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•
uint	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•
usmallint	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•
double float	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•
char	X	X	X	X	X	X	X	X	X	•	•	X	•	•	•	X	X	X	X
money	•	•	•	•	•	•	•	•	•	•	•	•				•	•	•	•
date													•						
time														•					
datetime										X	X				•				
text	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•
xml	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•

X – 明示的な変換が必要なことを示しています。

第 5 章

Adaptive Server Enterprise への接続

この章では、Embedded SQL プログラムを Adaptive Server Enterprise と接続する方法、およびサーバ、ユーザ名、パスワードの指定方法について説明します。

サーバへの接続

サーバに接続することによって、Embedded SQL プログラムがデータベースにアクセスでき、SQL オペレーションの実行が可能になります。

アプリケーション・プログラムと Adaptive Server Enterprise を接続するには、connect 文を使用します。アプリケーションが C と COBOL の両方の言語を使用している場合、最初の connect 文は COBOL プログラムから発行されなければなりません。詳細については、『Embedded SQL/COBOL プログラマーズ・ガイド』を参照してください。

connect 文の構文は次のとおりです。

```
exec sql connect :user [identified by :password]
  [at :connection_name] [using :server]
```

以降の各項では、connect 文の各引数について、1 つずつ説明します。connect 文に必須なのは *user* 引数のみです。それ以外の引数はオプションです。

user

user はホスト変数または引用符付き文字列で、Adaptive Server Enterprise ユーザ名を示します。ユーザ名は、指定したサーバに対して有効でなければなりません。

password

password はホスト変数または引用符付き文字列で、指定されたユーザ名に対応するパスワードを示します。この引数は、Adaptive Server Enterprise へアクセスするときにパスワードが要求される場合にのみ必要です。password 引数が null 値に設定されている場合は、ユーザがパスワードを指定する必要はありません。

connection_name

connection_name は、Adaptive Server Enterprise 接続をユニークに識別します。引用符付きリテラルを使用することもできます。アプリケーション・プログラム内では接続をいくつでも作成でき、そのうちの 1 つは名前を付けなくてもかまいません。*connection_name* の最大サイズは、255 文字です。

connect 文中で *connection_name* を使用すると、それ以降同じ接続を指定した Embedded SQL 文は、connect 文で指定したサーバを自動的に使用します。connect 文でサーバを指定していない場合には、デフォルト・サーバが使用されます。サーバの決定方法については、『Open Client/Server プログラマーズ・ガイド補足』を参照してください。

注意 現在のサーバ接続を変更するには、「現在の接続の変更」(39 ページ) で説明されている set connection 文を使用してください。

Embedded SQL 文は、connect 文で指定された *connection_name* だけを参照してください。アプリケーション・プログラムが使用するサーバ 1 つにつき、少なくとも 1 つの connect が必要です。

server

server はホスト変数または引用符付き文字列で、サーバ名を示します。*server* には、サーバをユニークにかつ完全に識別する文字列を使用してください。

connect の使用例

次の例は、“passes” というパスワードを使用して、SYBASE というサーバに接続します。

```
exec sql begin declare section;
CS_CHAR user[16];
 CS_CHAR passwd[16];
 CS_CHAR server[BUFSIZ];
exec sql end declare section;

 strcpy(server,"SYBASE");
 strcpy(passwd,"passes");
```

```
strcpy(user, "my_id");

exec sql connect :user identified by :passwd using
:server;
```

現在の接続の変更

現在の接続を変更するには、set connection 文を使用します。文の構文は、次のとおりです。

```
exec sql set connection {connection_name | default}
```

default は、名前のついていない接続があればそれを指します。

次は、現在の接続を変更する例を示します。

```
exec sql connect "ME" at connect1 using "SERVER1";
exec sql connect "ME" at connect2 using "SERVER2";
exec sql set connection connect1;
exec sql select user_id() into :myid;
```

複数の接続の確立

効率を考えると、Embedded SQL アプリケーションには、複数のアクティブな Adaptive Server Enterprise 接続が必要な場合があります。次に例を示します。

- 複数の Adaptive Server Enterprise のログイン名を必要とするアプリケーションでは、各ログイン・アカウントにつき 1 つずつ接続を持つとよいでしょう。
- 複数のサーバに接続することによって、アプリケーションは複数のサーバ上にあるデータへ同時にアクセスできます。

単一のアプリケーションが、1 つのサーバに対して複数の接続を持つことも、複数のサーバに対して複数の接続を持つことも可能です。アプリケーションに複数の接続を指定するには、connect 文の at*connection_name* 句を使用してください。

1 つの接続をオープンして、次にもう 1 つ別の新しい接続 (名前のある接続または名前のない接続) をオープンする場合は、新しい接続が現在の接続になります。

注意 プリコンパイラを使用して適切な SQL 文のストアド・プロシージャを作成する場合、プリコンパイラは、すべてのサーバ上のすべてのストアド・プロシージャ用のファイルを、各 Embedded SQL ファイルにつき 1 つずつ作成します。このファイルは、ユーザが適切なサーバにロードできます。サーバは他のサーバ用のプロシージャが読み込めないという警告やエラーを出しますが、これは無視してください。各サーバにとって適切なストアド・プロシージャが、そのサーバ上に正しくロードされます。使用するすべてのサーバにストアド・プロシージャをロードしてください。そうしないと、クエリは失敗します。

接続名の指定

表 5-1 は、接続名の付け方を示します。

表 *5-1:* 接続名の付け方

使用されている句	使用されていない句	接続名
at *connection_name*		*connection_name*
using *server_name*	at	*server_name*
なし		“DEFAULT” 接続の実際の名前

at 句では無効な文

次の文は、at 句では無効になります。

- connect
- begin declare section
- end declare section
- include file
- include sqlca
- set connection
- whenever

Adaptive Server Enterprise 接続の使い方

名前の付いていないデフォルトの接続を使用するとき以外は、Embedded SQL 文を実行するには、常に接続名を指定します。アプリケーション・プログラムが接続を 1 つしか使用しないときは、接続に名前を付けなくてもかまいません。この場合、at 句は必要ありません。

複数の接続を使用する場合の構文は、次のとおりです。

```
exec sql [at connection_name] sql_statement;
```

sql_statement は Transact-SQL 文です。

次の例は、2 つの接続を異なるサーバに対して確立し、使用する方法を示します。

```
    ...

exec sql begin declare section;
CS_CHAR user[16];
CS_CHAR passwd[16];
CS_CHAR name;
CS_INT  value, test;
CS_CHAR server_1[BUFSIZ];
CS_CHAR server_2[BUFSIZ];
exec sql end declare section;
...
strcpy (server_1, "sybase1");
strcpy (server_2, "sybase2");
strcpy(user, "my_id");
strcpy(passwd, "mypass");

exec sql connect :user identified by :passwd
at connection_2 using :server_2;

exec sql connect :user identified by :passwd using
:server_1;

/* This statement uses the current "server_1"
connection */
exec sql select royalty into :value from authors
where author = :name;

if (value == test)
{
/* This statement uses connection "connection_2" */
exec sql at connection_2 update authors
set column = :value*2
where author = :name;
}
```

サーバとの接続の切断

アプリケーション・プログラムが確立した接続は、明示的にクローズするか、プログラムが終了するまでオープンされたままになります。disconnect 文を使用すると、アプリケーション・プログラムと Adaptive Server Enterprise との接続がクローズされます。

disconnect 文の構文は、次のとおりです。

```
exec sql disconnect {connection_name | current | DEFAULT | all}
```

各パラメータの意味は、次のとおりです。

- current は、現在の接続を指定します。
- DEFAULT は、名前の付いていないデフォルト接続を指定します。
- all は、現在使用しているすべての接続を指定します。

disconnect 文は次の順序で処理を行います。

1 現在のトランザクションをロールバックします。このとき、セーブポイントが設定されていても無視します。

2 接続をクローズします。

3 テーブルなどのテンポラリ・オブジェクトをすべて削除します。

4 オープンしているカーソルをすべてクローズします。

5 現在のトランザクション用に設定されたロックを解放します。

6 サーバのデータベースへのアクセスを終了します。

disconnect は、現在のトランザクションを暗黙的には commit しません。

警告！ プログラムを終了する前に、オープンしている各接続に対して exec sql disconnect または exec sql disconnect all 文を実行するようにしてください。設定の仕様によっては、切断せずに「クライアント」を終了した場合、Adaptive Server Enterprise には通知されないことがあります。この場合は、アプリケーションによって保持されていたリソースが解放されません。

第 6 章

Transact-SQL 文の使い方

この章では、Embedded SQL とホスト変数による Transact-SQL 文の使い方について説明します。また、「ストアド・プロシージャ」の使い方についても説明します。ストアド・プロシージャとは、Adaptive Server Enterprise に保管されている SQL 文の集まりです。ストアド・プロシージャは、コンパイルしてから「データベース」に保存されるので、起動時に再コンパイルされずに高速に実行されます。

Embedded SQL における Transact-SQL 文

ここでは、Transact-SQL 文と、Embedded SQL における文の相違点について説明します。

exec sql 構文

Embedded SQL 文は、exec sql というキーワードで始めてください。Embedded SQL 文の構文は、次に示すとおりです。

```
exec sql [at connection_name] sql_statement
```

各パラメータの意味は、次のとおりです。

- *connection_name* は、文に対する接続を指定します。接続の詳細については、「第 5 章 Adaptive Server Enterprise への接続」を参照してください。at というキーワードは Transact-SQL 文および disconnect 文について有効です。
- *sql_statement* は、1 つ以上の Transact-SQL 文です。

無効な文

次に挙げる Transact-SQL 文を除いて、Embedded SQL 中では、すべての Transact-SQL 文が有効です。

- print
- readtext
- writetext

Embedded SQL における Transact-SQL 文の相違点

ほとんどの Transact-SQL 文は、Embedded SQL で使用される場合にその機能性や構文を保持しますが、select 文、update 文、delete 文などのデータ操作言語文 (DML 文) は、Embedded SQL においては多少異なります。

- 次の 4 項目は、select 文の into 句に特有のものです。
  - into 句は、スカラ・ホスト変数に単一のデータ・ローを割り当てることができます。この句は、単一のデータ・ローを返す select 文に対してのみ有効です。select を使って複数のローを選択した場合、負の SQLCODE となり、最初のローだけが返されます。
  - into 句内の変数が配列の場合は、select を使って複数のローを選択できます。配列が保持できる数よりも多くのローを select した場合には、SQLCODE <0 の例外が発生し、余分なローは失われます。
  - select は、カーソルや配列を使用する場合を除き、ホスト変数を使用して複数のデータ・ローを返すことはできません。
- update 文と delete 文では、探索条件 where current of *cursor_name* を使用できます。

ローの選択

select 文で使用できる最大カラム数は 1024 です。select 文の構文の完全なリストについては、『ASE リファレンス・マニュアル』を参照してください。

1 つのローの選択

カーソルや配列を使用しない場合、select はデータ中の単一のローだけを返すことができます。Embedded SQL で複数のデータ・ローを返すには、カーソルまたは配列を使う必要があります。

Embedded SQL では、select 文には into 句が必要です。into 句は値を割り当てるホスト変数のリストを指定します。

注意　Embedded SQL プリコンパイラの現在のバージョンでは、テーブルを指定する into 句をサポートしていません。

Embedded SQL の select 文の構文は、次のとおりです。

```
exec sql [at connect_name ]
    select [all | distinct] select_list into
    :host_variable[[indicator]:indicator_variable]
    [, :host_variable
    [[indicator]:indicator_variable]...];
```

select 文の句の詳細については、『ASE リファレンス・マニュアル』を参照してください。

pubs2 データベース中の authors テーブルにアクセスし、au_id の値をホスト変数 *id* に割り当てる select 文を発行する構文の例は、次のようになります。

```
exec sql select au_id into :id from authors
     where au_lname = "Stringer";
```

配列を使用した複数ローの選択

配列を使うと、複数のローを返すことができます。配列に対しては、配列への選択とフェッチという 2 つの動作があります。

配列への *select into* の使用

返されるローの最大数がわかっている場合は、「select into 配列」という方法を使います。select into 文が配列の保持できるローよりも多くのローを返した場合、その文は、最小の配列が保持可能な最も多くのローを返します。

例　次に、配列を選択する例を示します。

```
exec sql begin declare section;
     CS_CHAR titleid_array [100] [6];
 exec sql end declare section;
 ...
 exec sql select title_id into :titleid_array
     from titles;
```

配列フェッチとの使用

配列フェッチとともにインジケータを使用するには、*host_variable* 配列と同じ長さを持つインジケータの配列を宣言し、そのインジケータをホスト変数に関連付けるための構文を使用してください。

例

次に、配列フェッチとともにインジケータを使用する例を示します。

```
exec sql begin declare section;
     int item_numbers [100];
     short i_item_numbers [100];
 exec sql end declare section;
 ...
 exec sql select it_n from item.info
     into :item_numbers :i_item_numbers;
 ...
```

インジケータ変数としての配列と構造体

多数のカラムを持つテーブルの場合は、SQL 文で参照されるホスト変数のセットとして、配列と構造体を使用できます。インジケータ変数は、常に 2 バイトの integer (short) です。

例

例 1 インジケータ配列を宣言する例を示します。

```
EXEC SQL BEGIN DECLARE SECTION;

   /* Destination variables for fetches, using an */
   /* array.*/
   struct _hostvar {
      int m_titleid;
      char m_title[65];
      char m_pubname[41];
      char m_pubcity[21];
      char m_pubstate[3];
      char m_notes[201];
      float m_purchase;
   } host_var1;

   /* An indicator array for all variables.  */
   short indic_var[7];

EXEC SQL END DECLARE SECTION;
```

例 2 インジケータ構造体を宣言する例を示します。

```
EXEC SQL BEGIN DECLARE SECTION;
   /* Destination variables for fetches, using a */
   /* struct.*/
   struct _hostvar {
      int m_titleid;
      char m_title[65];
      char m_pubname[41];
```

```
        char m_pubcity[21];
        char m_pubstate[3];
        char m_notes[201];
        float m_purchase;
    } host_var1;

    /* An indicator structure for above variables.  */
    struct _indicvar {
        short i_titleid;
        short i_title;
        short i_pubname;
        short i_pubcity;
        short i_pubstate;
        short i_notes;
        short i_purchase;
    } indic_var1;

EXEC SQL END DECLARE SECTION;
```

例 3 インジケータ配列またはインジケータ構造体にクエリを実行する例を示します。

```
EXEC SQL
SELECT titleid, title, pubname, city, state, notes,
        purchases
    INTO :host_var1 INDICATOR :indic_var1
    FROM T1, T2
    WHERE ....
```

使用法

配列と構造体をインジケータ変数として使用する場合は、次の点に留意してください。

- インジケータ配列またはインジケータ構造体の要素数は、ホスト変数構造体の要素数と一致する必要があります。両者が一致しないと cpre または cpre64 の処理が停止し、コードが生成されません。
- SELECT リストのカラムが、INTO リストの順序、データ型、選択した構造体名と一致している必要があります。一致しないと、ct_bind() ランタイム・エラーが発生し、処理が停止します。
- INDICATOR はオプションのキーワードであり、省略できます。ただし、ホスト変数構造体とインジケータ変数またはインジケータ構造体の前の ":" は必須です。

エラー・メッセージ

表 6-1 に、この機能のホスト変数とインジケータ変数の各項目の不一致によるエラーを処理する際に作成される Embedded SQL の内部エラー・メッセージを示します。

表 *6-1:* 新しい内部エラー・メッセージ

メッセージ ID	メッセージ	重大度	対処法
M_INVTYPE_V	インジケータ変数の不正な型が、構造体で見つかりました。	致命的なエラー	ホスト変数宣言とインジケータ宣言で同じインジケータ変数が使用されていることを確認してください。
M_INVTYPE_VI	インジケータ構造体とホスト変数構造体の構造体要素の数が一致しません。	致命的なエラー	インジケータ構造体とホスト変数構造体に同じ数の要素を宣言してください。
M_INVTYPE_VII	インジケータ配列とホスト変数構造体の要素の数が一致しません。	致命的なエラー	インジケータ配列とホスト変数構造体に同じ数の要素を宣言してください。

制限事項

ホスト変数構造体を持つ単一ホスト変数または単一インジケータ変数を、インジケータ配列またはインジケータ構造体と混合することはできません。

fetch into バッチ配列

fetch は、指定された数のローを現在アクティブな集合から返します。フェッチは、実行されるたびにあとに続くバッチのローを返します。たとえば、現在アクティブな集合に 150 のローがあり、60 のローを選択してフェッチする場合、最初のフェッチは、最初の 60 ローを返します。次のフェッチは、それに続く 60 ローを返します。3 番目のフェッチは、最後の 30 ローを返します。

注意 フェッチされたローの総数を知る方法については、「SQLCA 変数」(15 ページ) で説明しているように、SQLCA の *SQLERRD* 変数を参照してください。

カーソルと配列

配列に返されるローの数がわからない場合は、「fetch into 配列」という方法を使います。カーソルを宣言し、オープンしてから、fetch を使ってローのグループを取得します。fetch into で配列が保持できるロー数よりも多くのローを返そうとした場合、その文は最小の配列が保持可能な最も多くのローを返し、SQLCODE はエラーまたは例外を示す負の値を表示します。

カーソルを使用した複数ローの選択

カーソルを使用して、複数のローを返すこともできます。「カーソル」は、複数のデータ・ローをホスト・プログラムに渡すときに、一度に 1 つのローを選択します。カーソルは、データの最初のローである「現在のロー」を示し、そのローをホスト・プログラムに渡します。次の fetch 文によってカーソルは次のローへ進み、今度はこのローが現在のローになります。この処理は、要求されたすべてのローがホスト・プログラムに渡されるまで続きます。

カーソルは、select 文が複数のデータ・ローを返す場合に使用してください。Client-Library は Adaptive Server Enterprise の返した複数のローを追跡し、アプリケーション用にバッファリングします。カーソルを使ってデータを取得するには fetch 文を使用します。

カーソルのメカニズムは、次に挙げる文から構成されます。

- declare
- open
- fetch
- update と delete where current of
- close

カーソルのスコープ規則

カーソルの初期スコープを制御する規則は、そのカーソルが静的カーソルか動的カーソルかによって異なります。ただし静的カーソルのオープン後または動的カーソルの宣言後は、両方のタイプのカーソルのスコープ規則は同じになります。カーソルのスコープ規則は次のとおりです。

- 静的カーソルがオープンされるまでは、スコープはそのカーソルが宣言されたファイルに限定されます。静的カーソルをオープンする文は、このファイルで指定される必要があります。静的カーソルがオープンされたあとは、スコープはそのカーソルがオープンされた接続に限定されます。
- 動的カーソルが宣言されるまでは、そのカーソルは存在していません。動的カーソルが宣言されたあとは、スコープはそのカーソルが宣言された接続に限定されます。
- 1 つのカーソル名を同時に複数の接続でオープンできます。
- カーソルをフェッチ、更新、削除、またはクローズする文を、そのカーソルが宣言される以外のファイルに指定することもできます。ただしこのような文は、そのカーソルがオープンされた接続に対して実行する必要があります。

同一の名前の静的カーソル

- Embedded SQL/C で静的カーソルを宣言してオープンし、データのフェッチに使用してからクローズした後に、カーソルの割り付けを解除しない場合、その後で同じ名前と DML でカーソルを宣言しても、エラーが発生しません。2 回目の宣言では、Embedded SQL/C プログラムが同じ名前と DML のカーソルがすでに存在することを単純に認識し、2 回目の宣言を無視して既存のカーソルを再度オープンします。ただし、同じ名前と異なる DML を使用してカーソルを再宣言すると、エラーが表示される場合があります。また、クローズされていない既存の静的カーソルをオープンしようとすると、次のエラーが表示されます。

  ```
  SQLCODE=(-16843032)
  Adaptive Server Error
  ct_cursor(OPEN): user api layer: external error: The cursor
  on this command structure has already been opened.
  ```

- isql ユーティリティを使用して静的カーソルを宣言してオープンし、データのフェッチに使用してからクローズした後に、カーソルの割り付けを解除しない場合、その後で同じ名前と DML でカーソルを宣言すると、isql エラーが表示されます。isql では、既存のカーソルの割り付けを解除してから、再宣言する必要があります。

同一の名前の動的カーソル

- Embedded SQL/C プログラムか isql ユーティリティを使用して動的カーソルを宣言してオープンし、データのフェッチに使用してからクローズした後に、カーソルの割り付けを解除しない場合、その後で同じ名前と DML でカーソルを宣言すると、エラーが表示されます。Embedded SQL/C プログラムか isql で最初に既存の動的カーソルの割り付けを解除してから、再宣言する必要があります。すでに宣言されている動的カーソルを宣言しようとしたときに、その割り付けが解除されていなければ、次のエラーが表示されます。

  ```
  SQLCODE=(-16843030)
  Adaptive Server Error
  ct_dynamic(CURSOR_DECLARE): user api layer: external
  error: A cursor has already been declared on this command
  structure.
  ```

カーソルの宣言

カーソルは、データから複数のローを返す各 select 文に対して宣言します。カーソルは、使用する前に宣言しておいてください。declare セクション内では、カーソルは宣言できません。

注意　declare cursor 文は、実行文ではなく宣言です。そのため、ファイルのどこにでも置くことができますが、SQLCODE、SQLSTATE、SQLCA は、この文の後では設定されません。

カーソルを宣言する構文は、次のとおりです。

```
exec sql declare  cursor_name  cursor
     for  select_statement ;
```

各パラメータの意味は、次のとおりです。

- *cursor_name* には、カーソルの名前を指定します。この名前はユニークで、最大 255 文字にしてください。また、英字または記号 “#” か “_” で始めてください。
- select_statement は、データから複数のローを返す select 文です。select の構文の詳細については、『ASE リファレンス・マニュアル』を参照してください。ただし、into 句や compute 句を使うことはできません。

例

次に、カーソルを宣言する例を示します。

```
exec sql declare c1 cursor for
 select type, price from titles
 where type like :wk-type;
```

上の例で、*c1* は、type カラムと price カラムで返されるローに対するカーソルとして宣言されています。プリコンパイラは、declare cursor 文に対するコードは生成しません。プリコンパイラは、単にカーソルに対応する select 文を格納します。

カーソルをオープンすると、select 文または declare cursor 文中のプロシージャが実行されます。そこでデータがフェッチされると、結果がホスト変数にコピーされます。

注意　各カーソルの open 文と declare 文は、同じファイルで指定してください。declare 文中のホスト変数は、open 文を定義したスコープと同じスコープになるようにしてください。ただし、一度カーソルをオープンすれば、どのファイルの中でもそのカーソルの fetch および update、または delete where current of を実行できます。

スクロール可能カーソルの宣言

スクロール可能カーソルを宣言する構文は、次のとおりです。

```
exec sql declare  cursor_name  [cursor sensitivity] [cursor
scrollability] cursor
     for  select_statement ;
```

各パラメータの意味は、次のとおりです。

- *cursor_name* には、カーソルの名前を指定します。この名前はユニークで、最大 255 文字にしてください。また、英字または記号 "#" か "_" で始めてください。
- *cursor sensitivity* では、カーソルに変更が反映されるかどうかを指定します。オプションは次のとおりです。
  - semi_sensitive − declare 文で semi_sensitive を指定すると、スクロール可能であることが暗黙的に設定されます。カーソルは、semi_sensitive (半反映型) のスクロール可能な読み取り専用カーソルになります。
  - insensitive − declare 文で insensitive を指定すると、カーソルは非反映型になります。スクロールの可能性は、declare 部分に SCROLL を指定することにより決定されます。SCROLL を省略するか、NOSCROLL を指定すると、カーソルには insensitive だけが設定され、非スクロール可能カーソルになります。このカーソルは読み込み専用でもあります。

  cursor sensitivity を指定しない場合、カーソルは非スクロール可能な読み込み専用カーソルになります。
- *cursor scrollability* には、カーソルがスクロール可能かどうかを指定します。オプションは次のとおりです。
  - scroll − declare 文で scroll を指定し、変更反映の可否を指定しない場合、カーソルは非反映型でスクロール可能になります。このカーソルは読み込み専用でもあります。
  - no scroll − SCROLL オプションを省略するか、NOSCROLL を指定した場合は、読み込み専用の非スクロール可能カーソルになります。カーソルの動作については、上の *cursor sensitivity* の説明を参照してください。

  cursor scrollability を指定しない場合、カーソルは非スクロール可能な読み込み専用カーソルになります。
- select_statement は、データから複数のローを返す select 文です。select の構文の詳細については、『ASE リファレンス・マニュアル』を参照してください。ただし、into 句や compute 句を使うことはできません。

カーソルのオープン

選択されたローの内容を取得するには、最初にカーソルをオープンする必要があります。open 文は、declare 文のカーソルに対応する select 文を実行します。

カーソルをオープンするときの open 文の構文は次のとおりです。

```
exec sql open cursor_name;
```

スクロール可能カーソルをオープンするときの open 文の構文は次のとおりです。

```
exec sql open cursor_name [ROW_COUNT = size];
```

注意　ROW_COUNT は、ホスト変数として配列を使用する場合にのみ指定するようにしてください。

カーソルを宣言すれば、select 文を発行したいときにいつでもカーソルを開くことができます。open 文を実行すると、Embedded SQL が、declare cursor 文の where 句で参照されたすべてのホスト変数の値を置き換えます。

オープンできるカーソルの数は、現在のセッションのリソース要求に依存します。Adaptive Server Enterprise は、オープンできるカーソルの数を制限しません。ただし、現在オープンしているカーソルをオープンすることはできません。このような場合には、エラー・メッセージが表示されます。

アプリケーションの実行中に、必要なだけカーソルをオープンできます。しかし、再びオープンする場合には、その前にクローズしなければなりません。現在のカーソルの結果セットからすべてのローを取得しなくても、別のカーソルの結果セットからローを取得できます。

カーソルを使用したデータのフェッチ

カーソルを使ってデータを取得し、そのデータをホスト変数に割り当てるには fetch 文を使用します。fetch 文の構文は次のとおりです。

```
exec sql [at connect_name] fetch cursor_name
 into : host_variable
 [[ indicator]: indicator_variable ]
 [,: host_variable
 [[ indicator]: indicator_variable ]...];
```

ただし、結果ローの各カラムに対して 1 つの *host_variable* が必要です。

各ホスト変数の前にはコロンを付け、次のホスト変数との間にカンマを入れてください。fetch 文にリストされたホスト変数は、select 文が取得する Adaptive Server Enterprise の値と一致している必要があります。つまり、変数の数は戻り値の数と同じで、同じ順序で並び、互換性のあるデータ型でなければなりません。

indicator_variable は、前の declare セクションで宣言された 2 バイトの符号付き整数です。Adaptive Server Enterprise から取得された値が null の場合、実行時のシステムは対応するインジケータ変数を -1 に設定します。また、null でなければ、インジケータを 0 に設定します。

fetch 文が取得するデータは、カーソルの位置に依存します。カーソルは、「現在のロー」を指します。fetch 文は、常に現在のローを返します。最初の fetch は最初のローを取得し、その値を指定されたホスト変数にコピーします。fetch を行うごとにカーソルは、次の結果ローに進みます。

通常、fetch 文は、select 文で返されたすべての値がホスト変数に割り当てられるように、ループの中に置きます。

次のループでは、whenever not found 文を使っています。

```
/* Initialize error-handling routines */
exec sql whenever sqlerror call err_handle();
 exec sql whenever not found goto end_label;
 for (;;)
 {
     exec sql fetch cursor_name
     into :host_variable [, host_variable];
     ...
 }
 end_label:
```

上記のループは、すべてのローが返されるか、エラーが発生するまで続きます。どちらの場合も、sqlcode または sqlstate がループの終了した理由を示します。これらは、フェッチのたびに whenever 文によってチェックされます。エラー処理ルーチンを使用すれば、どちらかの条件が発生したとき何らかのアクションが実行されるようにできます。詳細については、「第 8 章 エラーの処理」を参照してください。

スクロール可能カーソルを使用したデータのフェッチ

カーソルを使ってデータを取得し、そのデータをホスト変数に割り当てるには fetch 文を使用します。fetch 文の構文は次のとおりです。

```
exec sql [at connect_name] fetch [fetch
orientation]cursor_name
 into : host_variable
 [[ indicator]: indicator_variable ]
 [,: host_variable
 [[ indicator]: indicator_variable ]...];
```

結果ローの各カラムに対して 1 つの *host_variable* が必要です。

各ホスト変数の前にはコロンを付け、次のホスト変数との間にカンマを入れてください。fetch 文にリストされたホスト変数は、select 文が取得する Adaptive Server Enterprise の値と一致している必要があります。つまり、変数の数は戻り値の数と同じで、同じ順序で並び、互換性のあるデータ型でなければなりません。

fetch orientation には、カーソルがスクロール可能である場合のローのフェッチ方向を指定します。オプションは次のとおりです。NEXT、PRIOR、FIRST、LAST、ABSOLUTE *fetch_offset*、RELATIVE *fetch_offset* です。フェッチ方向を指定しない場合のデフォルトは next です。フェッチ方向を指定するには、カーソルがスクロール可能でなければなりません。

fetch 文が取得するデータは、カーソルの位置に依存します。fetch 文は、通常、カーソルをオープンするときに指定された ROW_COUNT に応じて、カーソル結果セットから単一または複数のローを取得します。カーソルがスクロール可能でない場合、fetch は、結果セットの次のローを取得します。カーソルがスクロール可能な場合、フェッチされるローの位置は、fetch 文に含まれるコマンドによって指定されます。

スクロール可能カーソルの宣言とローのフェッチの例

スクロール可能カーソルを宣言し、無作為にローをフェッチするには、カーソルの宣言で変更反映の可否とスクロール可能性を指定した後、フェッチ時にフェッチ方向を指定します。次の例は、insensitive (非反映型) スクロール可能カーソルを宣言して無作為にローをフェッチする方法を示しています。

```
exec sql declare c1 insensitive scroll cursor for
 select title_id, royalty, ytd_sales from authors
 where royalty < 25;
exec sql open c1;
```

この例では、カーソル宣言に scroll と insensitive が指定されています。フェッチ方向は fetch 時に指定可能であり、結果セットからフェッチする必要があるローを示します。

スクロール可能カーソルを宣言してオープンした後で、フェッチ時に FETCH 方向を指定できます。これによって、結果セットからどのローを取得するかを指定します。

次の fetch 例では、結果セットから先頭ローの指定されたカラムをフェッチします。

```
exec sql fetch first from c1 into :title,:roy,:sale;
```

次の fetch 例では、結果セットから前のローの指定されたカラムをフェッチします。

```
exec sql fetch prior from c1 into :title,:roy,:sale;
```

次の fetch 例では、結果セットからロー 20 の指定されたカラムをフェッチします。

```
exec sql fetch absolute 20 from c1 into :title,:roy,:sale;
```

フェッチ文で有効なローが返されたかどうかを判別するには、*sqlcode* または *sqlstate* を使用します。スクロール可能カーソルでは、カーソルが結果セット境界の外 (先頭ローの前や最終ローの後ろなど) に位置している場合、フェッチされるローが 0 行になる可能性があります。このような状況では、フェッチされるロー数が 0 であるのが本来の動作です。

カーソルを使用したローの更新と削除

カーソルの現在のローを更新または削除するには、update または delete 文内に探索条件として where current of *cursor_name* を指定します。

カーソルを使ってローを更新するには、更新に使われる結果カラムが、更新可能になっていなければなりません。max(colname) のような SQL 式の結果は使えません。つまり、結果カラムと更新されるデータベース・カラムに有効な対応がなければなりません。

次は、カーソルを使用してローを更新する方法の例です。

```
exec sql declare c1 cursor for
     select title_id, royalty, ytd_sales
     from titles
     where royalty < 25;

 exec sql open c1;

 for (;;)
 {
     exec sql fetch c1 into :title, :roy, :sales;
     if (SQLCODE == 100) break;
       if (sales > 10000)
         exec sql update titles
             set royalty = :roy + 2
             where current of c1;
 }
 exec sql close c1;
```

Embedded SQL での update と delete 文の構文は、where current of *cursor_name* による探索条件を含めて、Transact-SQL と同じです。

テーブルの更新プロトコルの決定とロックの詳細については、『ASE Transact-SQL ユーザーズ・ガイド』を参照してください。

カーソルのクローズ

オープンされたカーソルをクローズするには、close 文を使用します。close 文の構文は次のとおりです。

```
exec sql [at connection] close cursor_name;
```

クローズしたカーソルを再使用するには、open 文を発行します。再びカーソルをオープンしたとき、そのカーソルは最初のローを指します。オープンしていないカーソルに対して、close 文を発行しないでください。発行した場合は、エラーが起こります。

カーソルの例

次の例では、ネストした 2 つのカーソルを示します。カーソル c2 は、カーソル c1 から *title-id* にフェッチされた値に依存します。

このプログラムでは、*title-id* の値を、宣言時ではなくオープン時に取得します。

```
exec sql include sqlca;
main()
{
     exec sql begin declare section;
        CS_CHARtitle_id[7];
        CS_CHARtitle[81];
        CS_INT  totalsales;
        CS_SMALLINTsalesind;
        CS_CHAR au_lname[41];
        CS_CHAR au_fname[21];
     exec sql end declare section;
     exec sql whenever sqlerror call error_handler();
     exec sql whenever sqlwarning call error_handler();
     exec sql whenever not found continue;
     exec sql connect "sa" identified by "";
     exec sql declare c1 cursor for
        select title_id, title, total_sales from pubs2..titles;
     exec sql declare c2 cursor for
        select au_lname, au_fname from pubs2..authors
        where au_id in (select au_id from pubs2..titleauthor
            where title_id = :title_id);
     exec sql open c1;
     for (;;)
     {
       exec sql fetch c1 into :title_id, :title,
          :totalsales :salesind;
        if (sqlca.sqlcode ==100)
          break;
        printf("¥nTitle ID: %s, Total Sales: %d", title_id, totalsales);
        printf("¥n%s", title);
     if (totalsales > 10)
     {
        exec sql open c2;
        for (;;)
        {
            exec sql fetch c2 into :au_lname, :au_fname;
                if (sqlca.sqlcode == 100)
             break;
            printf("¥n¥tauthor: %s, %s", au_lname, au_fname);
          }
          exec sql close c2;
       }
    }
    exec sql close c1;
    exec sql disconnect all;
```

```
}
error_handler()
{
printf("%d¥n%s¥n",sqlca.sqlcode,sqlca.sqlerrm.sqlerrmc);
exec sql disconnect all;
exit(0);
}
```

次に示すのは、非反映型のスクロール可能カーソルの例です。

```
/*
**    example4.cp
**
**      This example is a non-interactive query program that
**      shows the user some actions executed by a scrollable,
**      insensitive cursor. This serves as a demo for usage
**      of scrollable cursors in ESQL/C.
*/
#include <stdio.h>
#include "sybsqlex.h"

EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;
#define TITLE_STRING    65
EXEC SQL END DECLARE SECTION;

void    error_handler();
void    warning_handler();
void    notfound_handler();

int
main(int argc, char *argv[])
{
        EXEC SQL BEGIN DECLARE SECTION;
        char    username[30];
        char    password[30];
        char    a_type[TITLE_STRING+1];
        EXEC SQL END DECLARE SECTION;

        EXEC SQL WHENEVER SQLERROR CALL error_handler();
        EXEC SQL WHENEVER SQLWARNING CALL warning_handler();
        EXEC SQL WHENEVER NOT FOUND CALL notfound_handler();

        strcpy(username, USER);
        strcpy(password, PASSWORD);

        EXEC SQL CONNECT :username IDENTIFIED BY :password;

        EXEC SQL USE pubs2;

    /*
```

```
** Declare an insensitive scrollable cursor against the
** titles table.
*/

EXEC SQL DECLARE typelist INSENSITIVE SCROLL CURSOR FOR
SELECT DISTINCT title FROM titles;

EXEC SQL OPEN typelist;

printf("¥n==> Selecting the FIRST book Title:¥n");

/*
** Fetch the first row in cursor resultset
*/
EXEC SQL FETCH FIRST FROM typelist INTO :a_type;

printf("¥n%s¥n", a_type);

/*
** Fetch the last row in cursor resultset
*/
printf("¥n==> Selecting the LAST book Title:¥n");

EXEC SQL FETCH LAST FROM typelist INTO :a_type;

printf("¥n%s¥n", a_type);

/*
** Fetch the previous (PRIOR) row based on current
** cursor position
*/
printf("¥n==> Selecting the PREVIOUS book Title:¥n");

EXEC SQL FETCH PRIOR FROM typelist INTO :a_type;

printf("¥n%s¥n", a_type);

/*
** Jump 5 rows back from current cursor position
*/
printf("¥n==> Rewinding 5 STEPS through the Book
selection...:¥n");

EXEC SQL FETCH RELATIVE -5 FROM typelist INTO :a_type;

printf("¥n%s¥n", a_type);

/*
** Fetch the next row based on current cursor position
*/
printf("¥n==> Selecting the NEXT book Title:¥n");
```

```
        EXEC SQL FETCH NEXT FROM typelist INTO :a_type;

        printf("¥n%s¥n", a_type);

        /*
        ** Jump out of the cursor result set. Note that this will
        ** lead to a "no rows found" condition. There are only 18
        ** rows in 'titles'.
        */

        a_type[0] = '¥0';

        printf("¥n==> Jumping out of the resultset.¥n");

        EXEC SQL FETCH ABSOLUTE 100 FROM typelist INTO :a_type;

        printf("¥n%s¥n", a_type);

        /* Close shop */
        EXEC SQL CLOSE typelist;

        printf("¥n==> That's it for now.¥n");

        EXEC SQL DISCONNECT DEFAULT;

        return(STDEXIT);
}

        /* Error handlers deleted */
```

カーソルを使用したこの他の例については、オンラインのサンプル・プログラムを参照してください。オンライン・サンプルへのアクセスの詳細については、使用しているプラットフォームの『Open Client/Server プログラマーズ・ガイド補足』を参照してください。

ストアド・プロシージャの使用

「ストアド・プロシージャ」には、*2* つのタイプがあります。ユーザ定義のストアド・プロシージャとプリコンパイラが生成するストアド・プロシージャです。どちらも Adaptive Server Enterprise がクエリをあらかじめ最適化するので、個々に独立した文よりも高速に実行されます。ユーザがユーザ定義のストアド・プロシージャを生成し、プリコンパイラがストアド・プロシージャを生成します。

ユーザ定義のストアド・プロシージャ

Embedded SQL を使用して、データ・ローを返す select 文を含むストアド・プロシージャを実行できます。ストアド・プロシージャは、プログラムに出力パラメータとリターン・ステータス変数を使って結果を返します。

ストアド・プロシージャのパラメータは、入力、出力、あるいは入出力の両方として使うこともできます。ストアド・プロシージャの詳細については、『ASE Transact-SQL ユーザーズ・ガイド』を参照してください。

構文

有効なストアド・プロシージャ名として含むことができるのは、大文字または小文字の英字、および “$”、“_”、“#” です。

ストアド・プロシージャに use 文を含めることはできません。

ストアド・プロシージャを実行するには、次のような構文を使います。

```
exec [[:status_variable =]status_value] procedure_name
[([[@parameter_name=]parameter_value [out[put]]],...)]
 [into :hostvar_1 [:indicator_1]
 [, hostvar_n [indicator_n, ...]]]
 [with recompile];
```

各パラメータの意味は、次のとおりです。

- *status_variable* では、Adaptive Server Enterprise リターン・ステータス値またはリターン・コードを返すことができます。これらは、ストアド・プロシージャが正常に終了したこと、または失敗の理由を示します。負のステータス値は、Adaptive Server Enterprise 用に予約されています。ストアド・プロシージャのリターン・ステータス値のリストについては、『ASE Transact-SQL ユーザーズ・ガイド』を参照してください。
- *status_value* は、ストアド・プロシージャのリターン・ステータス変数 *status_variable* の値です。
- *procedure_name* は、実行するストアド・プロシージャの名前です。
- *parameter_name* は、ストアド・プロシージャの変数の名前です。位置または名前のどちらかを使用して、パラメータを渡すことができます。1 つのパラメータが名前を指定された場合、すべてに名前を指定してください。詳細については、『ASE Transact-SQL ユーザーズ・ガイド』を参照してください。
- *parameter _value* はリテラル定数で、この値がストアド・プロシージャに渡されます。
- output は、ストアド・プロシージャがパラメータ値を返すことを示します。ストアド・プロシージャ内の対応するパラメータも、output キーワードを使用して作成されている必要があります。

- into*:hostvar_1* を指定すると、ストアド・プロシージャから返されるロー・データは、指定されたホスト変数 (*hostvar_1* ～ *hostvar_n*) に保管されます。それぞれのホスト変数にはインジケータ変数を指定できます。
- *indicator_n* は、事前に declare セクションで宣言された 2 バイトのホスト変数です。対応する *hostvar_n* の値が null の場合、ロー・データの取得時にインジケータ変数は -1 に設定されます。トランケーションが発生すると、インジケータ変数は結果カラムの実際の長さに設定されます。トランケーションがなければ、インジケータ変数は 0 になります。
- with recompile を指定すると、ストアド・プロシージャが実行されるたびに Adaptive Server Enterprise はこのプロシージャのための新しいクエリ・プランを作成します。

注意 Embedded SQL 内でストアド・プロシージャを実行するには、exec キーワードが必要です。exec の代わりに execute は使えません。

ストアド・プロシージャの例

次に、「ストアド・プロシージャ」への呼び出しの例を示します。ここで、*retcode* はステータス変数、a_proc はストアド・プロシージャ、*par1* は入力パラメータ、*par2* は出力パラメータです。

```
exec sql begin declare section;
 CS_INT  par1;
 CS_INT  par2;
 CS_SMALLINT  retcode;
 exec sql end declare section;
     ...
 exec sql exec :retcode = a_proc :par1, :par2 out;
```

次の例は、データ・ローを取得するストアド・プロシージャの使用方法を示します。ストアド・プロシージャの名前は *get_publishers* です。

```
exec sql begin declare section;
 CS_CHAR  pub_id(4);
 CS_CHAR  name(45);
 CS_CHAR  city(25);
 CS_CHAR  state(2);
 CS_SMALLINT  retcode;
 exec sql end declare section;
    . . .
 exec sql exec :retcode = get_publishers :pub_id
                    into :name :city :state;
```

exec 文の詳細については、「第 10 章 Embedded SQL 文：リファレンス・ページ」を参照してください。

規約

ストアド・プロシージャ中のパラメータのデータ型は、C ホスト変数と互換性がなければなりません。Client-Library は、ある決まった組み合わせだけしか変換しません。互換データ型の互換性については、「第 4 章 変数の使い方」にある表を参照してください。

プリコンパイラが生成するストアド・プロシージャ

オプションのコマンド・ライン・スイッチを設定すると、プリコンパイラは、プログラム中の Transact-SQL 文の実行を最適化できるストアド・プロシージャを自動的に生成します。

プリコンパイラのコマンド・ライン・オプション・スイッチのリストについては、『Open Client/Server プログラマーズ・ガイド補足』を参照してください。

プリコンパイラが生成するストアド・プロシージャを起動するには、次の手順に従ってください。

1　適切なコマンド・ライン・スイッチを設定して、プリコンパイラが自動的に最適化する Transact-SQL 文に対するストアド・プロシージャを生成します。

   プリコンパイラは、isql ファイルを生成します。このファイルには、ストアド・プロシージャを生成する文が含まれています。

2　対話型 SQL (isql プログラム) を使って、そのファイルを実行します。

   Adaptive Server Enterprise にストアド・プロシージャをロードします。プリコンパイラは、ストアド・プロシージャの呼び出しも出力ファイル内に生成します。

デフォルトでは、プリコンパイラが生成するストアド・プロシージャの名前は、ソース・プログラムの名前から拡張子を除いた名前になります。ストアド・プロシージャは連続した番号を持ち、ファイル名と番号はセミコロン (";") によって区切られます。

たとえば、ソース・プログラムの名前が *test1.pc* のとき、そのストアド・プロシージャの名前は test1;1 から順に test1;*n* となります。ここで *n* は、ソース・プログラムで最後のストアド・プロシージャの番号です。

オプションとして、ストアド・プロシージャの名前を変更するコマンド・ライン・フラグがあります。このフラグを使うと、変更したアプリケーションをテストする際に、生成済みのストアド・プロシージャを削除する必要がありません。そのアプリケーションのテストが終わったあとにフラグを設定せずにプリコンパイルすれば、ストアド・プロシージャをインストールできます。

注意　declare cursor 文を発行する場合は、select 句だけがストアド・プロシージャとして保存されます。アプリケーションに構文エラーがあると、プリコンパイラはファイルとストアド・プロシージャのどちらも作成しません。

文のグループ化

文を実行する場合には、バッチまたはトランザクションによってその文をグループ化できます。

バッチによる文のグループ化

バッチとは、ひとまとまりで実行させる文の集合です。プリコンパイラは、exec sql と ; キーワードの間にあるすべての Transact-SQL 文をバッチ・モードで実行します。

プリコンパイラはストアド・プロシージャを保存しますが、再度実行できるようにバッチを保存することはできません。バッチは、現在の実行にのみ有効です。

プリコンパイラは、結果セットを返さないバッチ・モード文だけをサポートしています。

```
exec sql insert into TABLE1 values (:val1)
     insert into TABLE2 values (:val2)
     insert into TABLE3 values (:val3);
```

上記の 3 つの insert 文は 1 つのグループとして処理されます。これで個別に処理するよりも効率的になります。バッチ内でエラー処理を行うには、get diagnostics を使います。詳細については、「get diagnostics の使い方」(89 ページ) を参照してください。

これらの文は結果を返さないので、バッチ内で使用できます。詳細については、『ASE Transact-SQL ユーザーズ・ガイド』を参照してください。

トランザクションによる文のグループ化

「トランザクション」とは、1 文または複数文から構成される、実行の 1 つの単位をいいます。トランザクション内の文は 1 つのグループとして実行されるので、それらの文すべてが実行されるか、1 つも実行されないかのどちらかになります。

プリコンパイラは、デフォルトの ANSI/ISO とオプションの Transact-SQL という 2 つのトランザクション・モードをサポートします。Transact-SQL トランザクション・モードでは、文の前に begin transaction 文がなければ、その文は暗黙的にコミットされます。

Transact-SQL モードはシステム・リソースを少ししか使用しないのに対して、デフォルトの ANSI/ISO トランザクション・モードはシステム応答時間に著しく影響します。使用するアプリケーションにとって適切なモード選択の詳細については、『ASE Transact-SQL ユーザーズ・ガイド』を参照してください。

プリコンパイラ・オプションを使用して、アプリケーションがオープンする接続に対するトランザクション・モードを決定できます。詳細については、『Open Client/Server プログラマーズ・ガイド補足』を参照してください。

Transact-SQL トランザクション・モード

このオプションのトランザクション・モードでは、Embedded SQL の構文は Transact-SQL で使用される構文と同じです。begin transaction 文が明示的にトランザクションを開始します。

Embedded SQL のトランザクション文の構文は、次のとおりです。

```
exec sql [at  connect_name ]
     begin transaction [ transaction_name ];

exec sql [at  connect_name]
     save transaction [ savepoint_name];

exec sql [at  connect_name] commit transaction
     [ transaction_name ];

exec sql [at  connect_name] rollback transaction
     [ savepoint_name  |  transaction_name ];
```

注意　disconnect コマンドは、すべてのオープン・トランザクションをロールバックします。disconnect 文の詳細については、「第 5 章 Adaptive Server Enterprise への接続」を参照してください。

ある接続で begin transaction を発行した場合は、同じ接続で save、commit、または roll back transaction のいずれかを発行してください。そうしないと、エラーが発生します。

デフォルトの ANSI/ISO トランザクション・モード

ANSI/ISO SQL では、save transaction または begin transaction 文がサポートされていません。その代わりに、アプリケーション・プログラムが次に挙げる文のどれかを実行すると、トランザクションは暗黙的に開始されます。

- delete
- insert
- select
- update
- open
- exec

トランザクションを明示的に終了させるには、commit work 文または rollback work 文を発行します。ANSI/ISO フォーマットの commit 文および rollback 文を使ってください。構文は次のとおりです。

```
exec sql commit [work] end-exec
exec sql rollback [work] end-exec
```

拡張トランザクション

「拡張トランザクション」は、複数の Embedded SQL 文を含む作業の単位です。Transact-SQL「トランザクション・モード」では、拡張トランザクション文を begin transaction および commit transaction 文で囲んでください。

デフォルトの ANSI モードでは、常に拡張トランザクションの中で処理が行われます。commit work 文を発行した場合は、現在の拡張トランザクションが終了し、別の拡張トランザクションが開始されます。詳細については、『ASE Transact-SQL ユーザーズ・ガイド』を参照してください。

注意 allow ddl in tran データベース・オプションが設定されている場合を除き、ANSI モードの拡張トランザクションの中では、次に示す Transact-SQL 文を使用しないでください。alter database、create database、create index、create table、create view、disk init、grant、load database、load transaction、revoke、truncate table、 update statistics

---

第 7 章

動的 SQL の使い方

この章では、動的 SQL について説明します。動的 SQL とは、Embedded SQL アプリケーションのユーザがアプリケーションの実行中に SQL 文を入力できるようにする高度な手法です。通常は静的 SQL で十分ですが、動的 SQL を使用すると、実行時にさまざまな SQL 文を柔軟に構築できます。

動的 SQL は、一連の Embedded SQL 文からなり、これを使用してオンライン・アプリケーションのユーザはアプリケーションの実行時に対話的にデータベースへアクセスできます。

次の条件のいずれかが実行時までわからないときに、動的 SQL を使用します。

- ユーザが実行する SQL 文
- カラム、インデックス、テーブルの参照
- ホスト変数の数またはデータ型

動的 SQL の概要

動的 SQL は ANSI および ISO SQL2 標準の一部です。動的 SQL は対話型アプリケーションの実行に便利です。アプリケーションが少数の SQL 文のみを実行する場合は、プログラム内に埋め込むことができます。しかし、アプリケーションがさまざまなタイプの SQL 文を実行する場合には、SQL 文を構築し、それらを動的にバインドして実行すると便利です。

次のような状況では、動的 SQL の利点を活用できます。たとえば、アプリケーション・プログラムが書店のデータベースから販売中の本を検索するとします。見込み顧客は、価格、主題、装丁の種類、ページ数、発行日付、言語などのいろいろな基準を適用できます。

ある顧客が「10 ～ 20 ドルの、ビジネスに関するノンフィクションの本が欲しい」と要求しているとします。この要求は、次の Transact-SQL 文で簡単に表現できます。

```
select * from titles where
 type = "business"
 and price between $10 and $20
```

すべての顧客の検索条件を予想することは不可能です。そのため、動的 SQL を使わないと、Embedded SQL プログラムでは、1 つのクエリを使用して該当する本のリストを生成するのは困難です。

動的 SQL を使えば、書店はそれぞれの顧客に対して別々の where 句の探索条件で「クエリ」を入力できます。書店は、発行日付、本のカテゴリー、その他のデータに基づいて要求を変えたり、表示されるカラムを変えたりできます。次に例を示します。

```
select * from titles
 where type = ?
 and price between ? and ?
```

疑問符 (“?”) は、動的パラメータのマーカで、ユーザが検索値を入力できる場所を表します。

動的 SQL プロトコル

注意　実行時まで文が完成されないので、プリコンパイラは、動的 SQL 文に対するストアド・プロシージャを生成しません。実行時に、Adaptive Server Enterprise が、一時的なストアド・プロシージャとして tempdb データベースに記録します。tempdb データベースにはユーザ名 “guest” が指定されている必要があり、この “guest” アカウントは create procedure パーミッションを持っている必要があります。そうでない場合は、これらのテンポラリ・ストアド・プロシージャの 1 つを実行しようとすると、「サーバのユーザ ID *user_id* は、データベース *database_name* の中では有効なユーザではありません。」というエラー・メッセージが表示されます。*user_id* はユーザのユーザ ID であり、*database_name* はユーザのデータベースの名前です。

動的 SQL の prepare 文は、サーバに実際の SQL 文を送ります。この文には、任意のデータ定義言語 (DDL)、データ操作言語 (DML) の文、または create procedure 以外の Transact-SQL 文のいずれかを使用できます。

動的 SQL は、次のように動作します。

1　入力データを SQL 文に変換します。

2　SQL 文が動的に実行できるかどうかを確認します。

3　SQL 文を Adaptive Server Enterprise に送り、実行の準備をします。Adaptive Server は、受け取った文をコンパイルし、一時的なストアド・プロシージャとして保存します (メソッド 2、3、4 のとき)。

4　すべての入力パラメータまたは記述子をバインドします (メソッド 2、3、4 のとき)。

5　文を実行します。

select リストが変化する select 文に対しては、返されるデータ項目とローの参照に記述子が使われます (メソッド 4 のとき)。

6　出力パラメータまたは記述子をバインドします (メソッド 2、3、4 のとき)。

7　結果を取得します。

8　Adaptive Server Enterprise のストアド・プロシージャを非アクティブにして、文を削除します (メソッド 2、3、4 のとき)。

9　Adaptive Server Enterprise と Client-Library からのすべてのエラーと警告条件を処理します。

メソッド 1：*execute immediate* の使い方

ホスト変数またはリテラル文字列に格納された Transact-SQL 文を Adaptive Server Enterprise に送るには、execute immediate を使用してください。文は、結果を返しません。このメソッドでは、select 文は実行できません。

動的に入力された文は、セッション中にユーザが起動した回数だけ実行されます。このメソッドでは、次の動作が実行されます。

1 Embedded SQL プログラムが Adaptive Server Enterprise にテキストを渡します。

2 Adaptive Server Enterprise は、文が select 文でないことを確認します。

3 Adaptive Server Enterprise が文をコンパイルして実行します。

execute immediate を使うと、ユーザは Transact-SQL 文のすべてまたは一部を入力できます。

execute immediate の構文は、次のとおりです。

```
exec sql [at connection_name] execute immediate
 {:host_variable | string};
```

各パラメータの意味は、次のとおりです。

- *host_variable* は、declare セクションで定義された文字列変数です。execute immediate を呼び出す前に、ホスト変数は、文法的に正しい完全な Transact-SQL 文に設定されていなければなりません。
- *string* は、*host_variable* の代わりに使われるリテラル Transact-SQL 文の文字列です。

Embedded SQL は、*host_variable* または文字列を処理したり、チェックしたりせずに Adaptive Server Enterprise に送ります。文が結果を返すか、または失敗した場合は、エラーが発生します。文の実行後に SQLCODE の値をテストしたり、エラー・ハンドラを設定するのに、whenever 文を使用できます。Embedded SQL プログラムでのエラー処理の詳細については、第 8 章を参照してください。

メソッド 1：例

次の 2 つの例では、メソッド 1 の execute immediate の実際の使い方を説明します。最初の例では、ユーザに文の入力を求めて、その文を実行します。

```
exec sql begin declare section;
 CS_CHAR statement_buffer[linesize];
 exec sql end declare section;
 ...
 printf("¥nEnter statement¥n");
 gets(statement_buffer);

 exec sql [at connection] execute immediate :statement_buffer;
```

次の例は、ユーザに探索条件の入力を求めて、更新する titles テーブルのローを指定します。そして、その探索条件を update 文に連結して、完全な文を Adaptive Server Enterprise に送ります。

```
exec sql begin declare section;
 CS_CHAR sqlstring[200];
 exec sql end declare section;

 char     cond[150];

 exec sql whenever sqlerror call err_p();
 exec sql whenever sqlwarning call warn_p();

 strcpy(sqlstring,
 "update titles set price=price*1.10 where ");

 printf("Enter search condition:");
 scanf("%s", cond);
 strcat(sqlstring, cond);

 exec sql execute immediate :sqlstring;

 exec sql commit work;
```

メソッド 2：*prepare* および *execute* の使い方

メソッド 2 の prepare および execute は、次のいずれかの場合に使います。

- 取得するデータがないことが確かで、文を複数回実行する場合。
- select 文が単一のローを返す場合。このメソッドでは、カーソルと select 文を対応させることはできません。

この処理は、単一ロー select とも呼ばれます。複数のローを取得する場合は、メソッド 3 またはメソッド 4 を使ってください。

このメソッドは、prepare と execute を使用して、文を Adaptive Server Enterprise に送る前に C 変数から Transact-SQL 文にデータを置換します。Transact-SQL 文は、文字バッファに C 変数のどの値を置換すべきかを表す動的パラメータ・マーカとともに格納されます。

このような文が準備されるため、Adaptive Server Enterprise は文をコンパイルして、一時的なストアド・プロシージャとして保存します。そして、セッションの間、必要なだけ文を繰り返し実行します。

prepare 文は、バッファを文の名前と対応させて、文を実行する準備をします。execute 文は、C 変数または SQL 記述子のリストからバッファに値を置換し、完全な文を Adaptive Server Enterprise に送ります。このメソッドで、すべての文を実行できます。

prepare

prepare 文の構文は次のとおりです。

```
exec sql [at connection] prepare statement_name from
{:host_variable | string};
```

各パラメータの意味は、次のとおりです。

- *statement_name* は、最大 255 文字までの名前で文を識別します。これは、プリコンパイラが、execute 文を prepare 文に対応させるのに使用する文の名前を含んでいる記号名または C の文字配列です。
- *host_variable* は、文字配列ホスト変数です。

  標準 Embedded SQL 文と同じように、ホスト変数の前にはコロンを付けてください。
- *string* は、リテラル文字列で、*host_variable* の代わりに使われます。

host_variable または *string* には、動的パラメータ・マーカ (“?”) を含めることができます。動的パラメータ・マーカは、動的クエリ内で文が実行されるときに値が置換される場所を表します。

execute

execute 文の構文は次のとおりです。

```
exec sql [at connection] execute statement_name
 [into host_var_list | sql descriptor
      descriptor_name | descriptor sqlda_name]
 [using host_var_list | sql descriptor
      descriptor_name | descriptor sqlda_name];
```

各パラメータの意味は、次のとおりです。

- *statement_name* は prepare 文で割り当てられた名前です。
- into は、単一ロー select に使われます。
- using は、*host_variable* の動的パラメータ・マーカと置換される C 変数または記述子を指定します。変数は、declare セクションで定義する必要があり、リストされている順番で置換されます。この句は文が動的パラメータ・マーカを含んでいるときにだけ必要です。
- *descriptor_name* は、動的 SQL 文の動的パラメータ・マーカの記述を保持しているメモリの領域を表します。
- *host_var_list* は、クエリ内でパラメータ・マーカ (“?”) に代わるホスト変数のリストです。
- *sqlda_name* は、SQLDA の名前です。

メソッド 2：例

次の例では、メソッド 2 の prepare および execute の実際の使い方を示します。この例では、titles テーブルのどのローを更新するかを決める where 句と価格を変更する乗数の入力が要求されます。選択したものによって、ホスト変数 "*sqlstring*" 内に格納されている update 文に、適切な文字列が連結されます。

```
exec sql begin declare section;


    CS_CHAR sqlstring[200];
    CS_FLOAT multiplier;


exec sql end declare section;


  char    cond[150];

  exec sql whenever sqlerror perform err_p();
  exec sql whenever sqlwarning perform warn_p();
   printf("Enter search condition:");
   scanf("%s", cond);
   printf("Enter price multiplier:  ");
   scanf("%f", &multiplier);
  strcpy(sqlstring,
    "update titles set price = price * ? where ");
  strcat(sqlstring, cond);
  exec sql prepare update_statement from :sqlstring;
  exec sql execute update_statement using
    :multiplier;
  exec sql commit;
```

メソッド 3：カーソルによる *prepare* および *fetch* の使い方

メソッド 3 は、カーソル文とともに prepare 文を使って select 文の結果を返します。このメソッドは、select リストが固定しており、複数のローを返す select 文に対して使います。つまり、前もってアプリケーションが返される select カラム・リスト属性の数とデータ型を確認している場合に使います。結果を取り込むためにホスト変数を予測して定義する必要があります。

メソッド 3 の場合、文を実行するには、declare、open、fetch、close cursor 文を含めてください。このメソッドは、文が複数のローを返すので必要です。準備文の識別子と指定したカーソル名は対応しています。update および delete where current of というカーソル文も使用できます。

メソッド 2 の prepare と execute のように、Transact-SQL select 文は、最初に文字ホスト変数または文字列に格納されます。これには、入力変数からの値をどこに置換するかを表す動的パラメータ・マーカを含めることができます。文には、prepare、declare、および open 文中でその文を識別するための名前が与えられています。

メソッド 3 には、5 つの手順が必要です。

1 prepare

2 declare

3 open

4 fetch (オプションとして update および delete)

5 close

次の項ではこれらの手順について説明します。

prepare

prepare 文は、メソッド 2 で使われたものと同じです。詳細については、「prepare」(72 ページ) を参照してください。

declare

declare 文は、カーソルに対する標準の declare 文と似ています。ただし動的 SQL では、カーソルは select 文に対してではなく、準備された *statement_name* に対して宣言され、すべての入力ホスト変数は、declare 文ではなく open 文で参照されます。

動的 declare 文は、宣言というより実行文です。そのため、コード内で実行文が有効である場所に置かれなければならず、アプリケーションは宣言を実行したあとで、ステータス・コード (SQLCODE、SQLCA、または SQLSTATE) を確認する必要があります。

declare 文の動的 SQL の構文は、次のとおりです。

```
exec sql [at connection_name] declare cursor_name
     cursor for statement_name;
```

各パラメータの意味は、次のとおりです。

- at *connection_name* は、カーソルが使用する Adaptive Server Enterprise 接続を指定します。
- *cursor_name* は、open、fetch、close 文で使われるカーソルを指定します。
- *statement_name* は、prepare 文で指定される名前で、実行される select 文を表します。

open

open 文は、文バッファのすべての入力変数を置換し、その結果を実行するために Adaptive Server Enterprise に送ります。open 文の構文は次のとおりです。

```
exec sql [at connection_name] open cursor_name [using
{host_var_list | sql descriptor descriptor_name |
descriptor sqlda_name}];
```

各パラメータの意味は、次のとおりです。

- *cursor_name* は、declare 文でカーソルに与えられた名前です。
- *host_var_list* は、動的パラメータ・マーカに対する値を含むホスト変数の名前です。
- *descriptor_name* は、動的パラメータ・マーカに対する値を含む記述子の名前です。
- *sqlda_name* は、SQLDA の名前です。

fetch および *close*

カーソルがオープンしたあとに、結果セットがアプリケーションに返されます。データは、アプリケーション・プログラムのホスト変数にフェッチおよびロードされます。オプションとして、データの更新や削除もできます。fetch と close 文は、静的 Embedded SQL と同じです。

fetch 文の構文は次のとおりです。

```
exec sql [at connection_name] fetch cursor_nameinto
:host_variable [[indicator]:indicator_variable]
 [,:host_variable
 [[indicator]:indicator_variable]...];
```

各パラメータの意味は、次のとおりです。

- *cursor_name* は、declare 文でカーソルに与えられた名前です。
- 結果ローのそれぞれのカラムには、1 つの C *host_variable* があります。変数は、declare セクションで定義しておいてください。データ型はカーソルで返される結果と互換性をもたせてください。

close 文の構文は次のとおりです。

```
exec sql [at connection_name] close cursor_name;
```

cursor_name は、declare 文でカーソルに割り当てられた名前です。

メソッド 3：例

次は、prepare および fetch を使用する例で、ユーザに select 文の order by 句の入力を求めます。

```
exec sql begin declare section;
 CS_CHAR     sqlstring[200];
 CS_FLOAT    bookprice,condprice;
 CS_CHAR     booktitle[200];
 exec sql end declare section;

 char    orderby[150];

 exec sql whenever sqlerror call err_p();
 exec sql whenever sqlwarning call warn_p();

 strcpy(sqlstring,
 "select title,price from titles¥
 where price>? order by ");

 printf("Enter the order by clause:");
 scanf("%s", orderby);
 strcat(sqlstring, orderby);

 exec sql prepare select_state from :sqlstring;
 exec sql declare select_cur cursor for select_state;

 condprice = 10;  /* the user can be prompted
                  ** for this value */

 exec sql open select_cur using :condprice;
 exec sql whenever not found goto end;

 for (;;)
 {
     exec sql fetch select_cur
       into :booktitle,:bookprice;
     printf("%20s  %bookprice=%6.2f¥n",
         booktitle, bookprice);
 }

 end:

 exec sql close select_cur;
 exec sql commit work;
```

メソッド 4：動的記述子による *prepare* および *fetch* の使い方

メソッド 4 では、select リストが変化する select 文を使用できます。つまり、アプリケーションを作成するときに、select 文が返す項目の型や数がわからなくてもよいということです。どれだけの数の変数が必要であるか、また変数はどのようなデータ型でなければならないかが不明なため、前もってホスト変数を定義できないときにメソッド 4 を使用します。

メソッド 4：動的記述子

「動的記述子」とは、動的 SQL 文で使用される変数の記述を保持するデータ構造体です。動的記述子には、SQL 記述子と SQLDA 構造体の 2 種類があります。この 2 種類の動的記述子については、このあとの項を参照してください。

カーソルは、オープンされるときにそのカーソルに対応する入力記述子を持つことができます。入力記述子には、動的 SQL 文のパラメータ・マーカに代入される値が含まれます。

ユーザがパラメータの数と、それぞれのパラメータについてのデータ型、長さ、精度、位取り、インジケータ、データなどを含む適切な情報を使用して入力記述子を指定したあとに、カーソルがオープンされます。

出力記述子は fetch 文と対応していて、fetch 文を実行した結果生成されるデータを保持しています。Adaptive Server Enterprise は、データ型と返される実際のデータを含むデータ項目の属性を読み込みます。SQL 記述子を使用する場合は、get descriptor 文を使用してホスト変数にデータをコピーしてください。

動的 SQL のメソッド 4 は、次の手順に従って実行されます。

1　実行するための文を準備します。

2　カーソルと文を対応付けます。

3　入力パラメータまたは記述子を定義およびバインドします。

- 入力記述子を使っているときに、その記述子に割り付けます。
- 入力パラメータを使っているときに、そのパラメータに文またはカーソルを対応付けます。

4　適当な入力パラメータまたは記述子を使用してカーソルをオープンします。

5　入力記述子と異なる場合は出力記述子を割り付け、出力記述子を文へバインドします。

6　fetch cursor と出力記述子を使用してデータを取得します。

7　動的記述子からホスト・プログラム変数へデータをコピーします。SQLDA を使用している場合、この手順は適用されません。データは手順 6 でコピーされます。

8　カーソルをクローズします。

9　動的記述子の割り付けを解除します。

10　文 (最終的には、ストアド・プロシージャ) を削除します。

動的記述子の文

SQL 文に記述子を対応させる文と、その SQL 文と対応するカーソルに記述子を対応させる文があります。次の表は、メソッド 4 の場合の動的 SQL 文を示します。

文	説明
allocate descriptor	SQL 記述子を割り付けるように Client-Library に通知する。
describe input	prepare 文の動的パラメータ・マーカについての情報を得る。
set descriptor	動的記述子のデータの挿入または更新を行う。
get descriptor	記述子に保管されたロー情報またはパラメータ情報をホスト変数に移動し、これによってアプリケーション・プログラムがその情報を使用できるようにする。
execute	準備された文を実行する。
open cursor	記述子をカーソルと対応させて、そのカーソルをオープンする。
describe output	準備された動的 SQL 文の select リスト・カラムについての情報を得る。
fetch cursor	動的に宣言されたカーソルに対してデータのローを取得する。
deallocate descriptor	動的記述子の割り付けを解除する。

上記の文の詳細については、「第 10 章 Embedded SQL 文：リファレンス・ページ」を参照してください。

SQL 記述子の概要

SQL 記述子とは、prepare 文で準備された動的 SQL 文で使用される変数の記述を保管するメモリの領域です。SQL 記述子には、データ属性についての次のような情報が含まれる場合があります (詳細については、「第 10 章 Embedded SQL 文：リファレンス・ページ」の set descriptor と get descriptor コマンドについての説明を参照してください)。

- precision ― 整数。
- scale ― 整数。
- nullable ― カラムが null を含むことができる場合は 1 (cs_true)、含むことができない場合は 0 (cs_false)。get descriptor 文でのみ有効。

- indicator − 動的パラメータ・マーカに対応したインジケータの値。get descriptor 文でのみ有効。
- name − 動的パラメータ・マーカの名前。get descriptor 文でのみ有効。
- data − 項目番号で指定された動的パラメータ・マーカの値。*indicator* の値が -1 の場合、*data* の値は未定義である。
- count − 記述子で記述された動的パラメータ・マーカの数。
- type − 動的パラメータ・マーカまたはホスト変数のデータ型。
- returned_length − 出力カラム内のデータの実際の長さ。

メソッド 4：SQL 記述子の使用例

次の例は、動的パラメータ・マーカおよび SQL 記述子とともに prepare および fetch を使う例です。

```
exec sql begin declare section
     int  index_colcnt, coltype;
     int  int_buff;
     char     char_buff[255], void_buff[255];
     char type[255], title[255];
     char colname[255];
     int sales;
     int descnt, occur, cnt;
     int condcnt, diag_cnt, num_msgs;
     char user_id[30], pass_id[30], server_name[30];
     char str1[1024], str2[1024], str3[1024],
         str4[1024];
exec sql end declare section;
...
 void dyn_m4()
 {
  printf("¥n¥nDynamic sql Method 4¥n");
  printf("Enter in a Select statement to retrieve
     any kind of ");
  printf("information from the pubs database:");
  scanf("%s", &str4);

  printf("¥nEnter the largest number of columns to
     be retrieved or the number ");
  printf("of ? in the sql statement:¥n");
  scanf("%d", &occur);

 exec sql allocate descriptor dinout with max
     :occur;
 exec sql prepare s4 from :str4;
exec sql declare c2 cursor for s4;

exec sql describe input s4 using sql descriptor
```

```
     dinout;

fill_descriptor();

exec sql open c2 using sql descriptor dinout;


while (sqlca.sqlcode == 0)
 {
     exec sql fetch c2 into sql descriptor dinout;
     if(sqlca.sqlcode == 0) {
         print_descriptor();
     }
 }

 exec sql close c2;

exec sql deallocate descriptor dinout;

exec sql deallocate prepare s4;

printf("Dynamic SQL Method 4 completed¥n¥n");

 }

 void
 print_descriptor()
 {

exec sql get descriptor dinout :descnt = count;
 printf("Column name ¥t¥tColumn data¥n");
  printf("------------ ¥t¥t-------------------------
 ----------------¥n");

  for (index_colcnt = 1; index_colcnt <= descnt;
      index_colcnt++)
  { /* get each column attribute */
     exec sql get descriptor dinout value
        :index_colcnt :coltype = TYPE;

     switch(coltype)
     {
      ...
      case 4:/* integer type */
         exec sql get descriptor dinout value
             :index_colcnt
                 :colname = NAME, :int_buff = DATA;
         printf("%s ¥t¥t %d¥n", colname, int_buff);
         break;
      ...
     }
  }

 }
```

```
void
fill_descriptor()
{
exec sql get descriptor dinout :descnt = count;
    for (cnt = 1; cnt <= descnt; cnt++)
 {
    printf("Enter in the data type of the %d ?:",
        cnt);
    scanf("%d", &coltype;);
    switch(coltype)
    {
     ...
     case 4:/* integer type */
        printf("Enter in the value of the data:");
        scanf("%d¥n", ∫_buff);
        exec sql set descriptor dinout VALUE :cnt
            TYPE = :coltype,
                DATA = :int_buff;
        break;

     default:
        printf("non-supported column type.¥n");
        break;
    }
 }
}
```

SQLDA について

SQLDA は、SQL 記述子のように動的 SQL 文の準備文で使用される変数を記述するホスト言語の構造体です。ただし SQL 記述子とは異なり、SQLDA はユーザがアクセスできるフィールドを持つパブリック・データ構造体です。SQLDA を使用する文は、SQL 記述子を使用する同等の文よりも速く実行される場合があります。

SQLDA 構造体は SQL 標準の一部ではありません。Embedded SQL の実装方法が異なると、SQLDA 構造体の定義も異なります。Embedded SQL バージョン 11.1 以降では、Sybase が定義する SQLDA をサポートしていますが、他のベンダが定義する SQLDA データ型はサポートしていません。

Embedded SQL プログラムで SQLDA データ型を定義するには、Embedded SQL コマンド include sqlda を使用します。プログラムの中で SQLDA 構造体を割り付けるには、malloc 関数を使用してください。SQLDA 構造体の割り付けを解除するには、free 関数を使用してください。プログラムによって作成されるすべての SQLDA 構造体の割り付け解除は、そのプログラム内で行います。Embedded SQL では、1 つのプログラムが作成できる SQLDA 構造体の数を制限しません。

表 7-1 は、SQLDA 構造体のフィールドの説明です。

表 *7-1: SQLDA* 構造体のフィールド

フィールド	データ型	説明
sd_sqln	CS_SMALLINT	sd_ 配列のサイズ。
sd_sqld	CS_SMALLINT	記述されるクエリのカラム数。つまり記述される文がクエリでない場合は、0 である。fetch 文、open 文、execute 文の場合、このフィールドは sd_column で記述されるホスト変数の数を示す。また describe input 文の場合、このフィールドは動的パラメータ・マーカの数を示す。
sd_column[].sd_datafmt	CS_DATAFMT	このカラムと対応する Client-Library の CS_DATAFMT 構造体を示す。『Open Client Client-Library/C リファレンス・マニュアル』の ct_bind、ct_param、ct_describe の説明を参照。
sd_column[].sd_sqldata	CS_VOID	fetch 文、open 文、execute 文の場合は、その文のホスト変数のアドレスを保管する。このフィールドは、describe 文または prepare 文では使用されない。
sd_column[].sd_sqlind	CS_SMALLINT	fetch 文、open 文、execute 文の場合、このフィールドは記述されるカラムのインジケータ変数として機能する。そのカラムの値が null である場合、このフィールドは -1 に設定される。このフィールドは、describe 文または prepare 文では使用されない。
sd_column[].sd_sqllen	CS_INT	このカラムと対応する *sd_sqldata* で示されたデータの実際の大きさ。
sd_column[].sd_sqlmore	CS_VOID	Sybase によって予約されている。

Embedded SQL のヘッダ・ファイル *sqlda.h* には、SQLDADECL マクロが入っています。このマクロによって、ユーザはプログラムで SQLDA 構造体を宣言できます。SQLDADECL マクロは次のようになっています。

```
#ifndef SQLDADECL
 #define SQLDADECL(name, size)
     struct {
         CS_INT    sd_sqln;
         CS_INT    sd_sqln;
         struct {
             CS_DATAFMT     sd_datafmt;
             CS_VOID        sd_sqldata;
             CS_SMALLINT    sd_sqlind;
             CS_INT         sd_sqllen;
             CS_VOID        sd_sqlmore;
         } sd_column[ (SIZE) ]
     } name
 #endif /* SQLDADECL */
```

メソッド 4：SQLDA の使用例

次は、動的パラメータ・マーカおよび SQL 記述子とともに prepare および fetch を使う例です。

```
exec sql include sqlca;
 exec sql include sqlda;

...
SQLDA *input_descriptor, *output_descriptor;
 CS_SMALLINT small;
 CS_CHAR     character[20];

input_descriptor = (SQLDA *)malloc(SYB_SQLDA_SIZE(3));
 input_descriptor->sqlda_sqln = 3;
 output_descriptor = (SQLDA *)malloc(SYB_SQLDA_SIZE(3));
 output_descriptor->sqlda_sqln = 3;
*p_retcode = CS_SUCCEED;
exec sql connect "sa" identified by "";
/* setup */
exec sql drop table example;
exec sql create table example (fruit char(30), number int);
exec sql insert example values ('tangerine', 1);
exec sql insert example values ('pomegranate', 2);
exec sql insert example values ('banana', 3);
/* Prepare and describe the select statement */
 exec sql prepare statement from
     "select fruit from example where number = ?";
exec sql describe input statement using descriptor    input_descriptor;
input_descriptor->sqlda_column[0].sqlda_datafmt.datatype =     CS_SMALLINT_TYPE;
input_descriptor->sqlda_column[0].sqlda_sqldata = &small;
input_descriptor->sqlda_column[0].sqlda_sqllen = sizeof(small);
small = 2;
exec sql describe output statement using descriptor
     output_descriptor;
if (output_descriptor->sqlda_sqld != 1 ||
     output_descriptor->sqlda_column[0].sqlda_datafmt.datatype !=     CS_CHAR_TYPE)
     FAIL;
else
     printf("First describe output ¥n");
output_descriptor->sqlda_column[0].sqlda_sqldata = character;
output_descriptor->sqlda_column[0].sqlda_datafmt.maxlength = 20;
exec sql execute statement into descriptor output_descriptor
     using descriptor input_descriptor;
printf("Expected pomegranate, got %s¥n", character);
exec sql deallocate prepare statement;
/* Prepare and describe second select statement */
exec sql prepare statement from
     "select number from example where fruit = ?";
exec sql declare c cursor for statement;
exec sql describe input statement using descriptor
     input_descriptor;
```

```
input_descriptor->sqlda_column->sqlda_sqldata = character;
input_descriptor->sqlda_column->sqlda_datafmt.maxlength =     CS_NULLTERM;
strcpy(character, "banana");
input_descriptor->sqlda_column->sqlda_sqllen = CS_NULLTERM;
exec sql open c using descriptor input_descriptor;
exec sql describe output statement using descriptor
     output_descriptor;
output_descriptor->sqlda_column->sqlda_sqldata = character;
output_descriptor->sqlda_column->sqlda_datafmt.datatype =     CS_CHAR_TYPE;
output_descriptor->sqlda_column->sqlda_datafmt.maxlength = 20;
output_descriptor->sqlda_column->sqlda_sqllen = 20;
output_descriptor->sqlda_column->sqlda_datafmt.format =
     (CS_FMT_NULLTERM | CS_FMT_PADBLANK);
exec sql fetch c into descriptor output_descriptor;
printf("Expected pomegranate, got %s¥n", character);
exec sql commit work;
```

まとめ

この章では、動的 SQL、つまりオンライン・アプリケーションが対話的にデータベースにアクセスできる一連の Embedded SQL 文について説明しました。このデータベースとの対話によって、実行時に SQL 文を定義して実行できます。

動的 SQL には 4 つのメソッドがあります。

- メソッド 1：execute immediate
- メソッド 2：prepare および execute
- メソッド 3：prepare および fetch
- メソッド 4：動的記述子を使用した prepare と fetch

次の章では、Embedded SQL のエラーの検出および修正方法を説明します。

第 8 章

エラーの処理

この章では、Embedded SQL プログラムの実行中に発生するエラーを検出し、修正する方法について説明します。また、警告およびエラーの処理に使用する whenever および get diagnostics 文についてや、警告およびエラーと関係する SQLCA 変数についても説明します。

Embedded SQL アプリケーションの稼働中、アプリケーションの操作を妨げるイベントが起こることもあります。次にそのようなイベントの例を示します。

- Adaptive Server Enterprise がアクセスできなくなる
- ユーザが間違ったパスワードを入力する
- ユーザがデータベース・オブジェクトへのアクセス権を持っていない
- データベース・オブジェクトが削除されている
- カラムのデータ型が変更されている
- クエリが予期しない null 値を返す
- 動的 SQL 文に構文エラーがある

ユーザが、警告およびエラー処理コードを記述しておくと、これらのイベントを前もって予測でき、上記の状況をうまく復旧できます。

エラーのテスト

Adaptive Server Enterprise に送られた SQL 文の実行の成否を示すために、Embedded SQL は、*SQLCODE* 変数にリターン・コードを返します。Embedded SQL 文のあとで *SQLCODE* の値をテストするか、whenever 文を使ってユーザ用のテスト・コードを生成するようにプリコンパイラに指示できます。whenever 文については、この章で後述します。

SQLCODE の使用

表 8-1 に、*SQLCODE* に入る可能性のある値を示します。

表 *8-1: SQLCODE* の戻り値

値	意味
0	警告またはエラーの発生なし。
<0	エラーが発生した。SQLCA 変数には、エラーを診断するための情報が入る。
100	文は正常に実行されたが、最後の文からローが返されない。これは、カーソルからローをフェッチするループを使用しているときに役立つ。SQLCODE が 100 になると、ループおよびフェッチされたすべてのローが終了する。この手法については、「第 6 章 Transact-SQL 文の使い方」を参照。

警告状態のテスト

SQLCODE が文の実行が成功したことを示していても、警告状態が発生する場合があります。8 つの文字配列である sqlca.sqlwarn は、このような警告状態を示します。各 sqlwarn 配列要素またはフラグには、スペース文字または文字 “W” が格納されます。

表 8-2 は、各フラグでのスペース文字または文字 “W” の意味を示します。

表 8-2: sqlwarn フラグ

フラグ	説明
sqlwarn[0]	ブランクの場合、いかなるときでも警告状態ではなく、他のすべての sqlwarn フラグもブランクになる。sqlwarn[0] に “W” が設定されている場合、1 つ以上の警告状態が発生し、他に少なくとも 1 つのフラグに “W” が設定されている。
sqlwarn[1]	“W” が設定されている場合、fetch 文で指定した文字列変数が文の結果データを格納するには短すぎ、そのため結果データがトランケートされた。トランケートされたデータの元の長さを受信するインジケータ変数を指定していない。
sqlwarn[2]	“W” が設定されている場合、Adaptive Server Enterprise に送信した入力が、式や null 値を使用できないテーブルへの入力値など、無効なコンテキスト内に null 値を含んでいた。
sqlwarn[3]	select 文の結果セット内のカラム数が、文の into 句内のホスト変数の数を超えていた。
sqlwarn[4]	Sybase によって予約されている。
sqlwarn[5]	この文を実行しようとしたとき、Adaptive Server が変換エラーを生成した。
sqlwarn[6]	Sybase によって予約されている。
sqlwarn[7]	Sybase によって予約されている。

警告のテストは、SQL 文の実行が成功したことを確認してから行ってください。ユーザ用のテスト・コードを生成するようにプリコンパイラに指示するには、whenever 文を使用してください。これについては、次の項で説明します。

whenever を使用したエラーのトラップ

Embedded SQL の whenever 文を使って、エラーおよび警告状態をトラップします。この文には、Adaptive Server Enterprise に送られた Embedded SQL 文の結果に対応した動作を指定します。

whenever 文は、実行文ではありません。代わりに、プリコンパイラに C コードを生成するように指示して、プログラムの各 Embedded SQL 実行文のあとで指定した条件をテストします。

whenever 文の構文は、次のとおりです。

```
exec sql whenever {sqlwarning | sqlerror | not found}
 {continue | goto label |
 call function_name ([param [, param]...]) | stop};
```

whenever の条件のテスト

whenever 文では、次のような 3 つの条件をテストできます。

- sqlwarning
- sqlerror
- not found

各条件に対する whenever 文を記述していない場合、プリコンパイラは警告メッセージを作成します。ユーザが、自分でコードを記述してエラーおよび警告をチェックする場合は、各条件に対応する whenever...continue 句を記述してプリコンパイラの警告を出さないようにします。こうすると、プリコンパイラはエラーおよび警告を無視します。

verbose オプションを使ってプリコンパイルすると、プリコンパイラは、connect 文の一部として ct_debug() 関数呼び出しを生成します。これによって、Client-Library はアプリケーションが実行している画面に情報、警告、およびエラー・メッセージを表示します。whenever は、これらのメッセージを無効にしません。『Open Client/Server プログラマーズ・ガイド補足』を参照してください。

Embedded SQL 文の実行後、*sqlcode* および *sqlwarn0* の値によって条件のいずれかが成立しているかどうかを判定します。表 8-3 に、whenever が条件の検出に使用する基準を示します。

表 *8-3: whenever* 文の基準

条件	基準
sqlwarning	sqlcode = 0 および sqlwarn[0] = W
sqlerror	sqlcode < 0
not found	sqlcode = 100

whenever 文の動作を変更するには、その条件について新しく whenever 文を記述します。whenever は、その同じ条件に対する whenever 文が次に現れるまで、それ以降の Embedded SQL 文のすべてに適用されます。

whenever 文は、アプリケーション・プログラムの論理を無視します。たとえば、ループの最後に whenever を置いた場合は、whenever 文より前の部分のコードに対しては、その後の loop で実行されたとしても、何の影響も与えません。

whenever の動作

whenever 文は、次の 4 つの動作のうちの 1 つを指定します。

表 *8-4: whenever* の動作

動作	説明
continue	SQL 文が指定した条件を戻しても、特別の動作は実行しない。通常の処理を続ける。
goto	アプリケーション・プログラム中のエラー処理プロシージャへ分岐する。goto を記述するときは goto または go to のどちらでも使用できる。後ろには、有効なラベル名を置かなければならない。ラベル名がプログラムで定義されていない場合、プリコンパイラはエラーを検出しないが、C コンパイラが検出する。
call	別の C ルーチンを呼び出す。オプションで変数を渡すこともできる。
stop	SQL 文が指定条件を戻したとき、プログラムを終了する。

get diagnostics の使い方

get diagnostics 文は、Client-Library からエラー・メッセージ、警告メッセージ、および情報メッセージを取得します。この文は whenever 文と似ていますが、検出したエラーの詳細を取得できるよう拡張されているため、さらに強力なものとなっています。

whenever 文内でアプリケーションを指定して別のルーチンへジャンプしたり呼び出したりする場合は、関数コード内で次のように get diagnostics 文を指定します。

```
void
 error_handler()
 {
     exec sql begin declare section;
         int num_msgs;
         int condcnt;
     exec sql include sqlca;
     exec sql end declare section;
     exec sql get diagnostics :num_msgs = number;
     for (condcnt=1; condcnt <= num_msgs; condcnt++)
     {
         exec sql get diagnostics exception :condcnt
             :sqlca = sqlca_info;
         printf("sqlcode is :%d¥n¥ message text:
             %s¥n", sqlca.sqlcode,
             sqlca.sqlerrm.sqlerrmc);
     }
 }
```

警告およびエラーを処理するルーチンの書き方

Embedded SQL アプリケーションでエラーおよび警告を処理するのによい手法は、エラーや警告を処理するカスタム・プロシージャを記述し、whenever...call 文を使ってそのプロシージャをインストールすることです。

次に、警告およびエラー処理ルーチンの例を示します。記述量を少なくするため、どちらのルーチンでも、通常は指定する必要のある特定の条件を省略しています。warning_hndl では *sqlwarn*[1] のコードを省略し、error_hndl では Client-Library のエラーとオペレーティング・システムのエラーを処理するコードを省略しています。

```
/* Declare the sqlca.   */
 exec sql include sqlca;
 exec sql whenever sqlerror call error_handler();
 exec sql whenever sqlwarning call     warning_handler();
 exec sql whenever not found continue;
 /*
 ** void error_handler()
 **
 ** Displays error codes and numbers from the sqlca
 */
 void error_handler()
 {
     fprint(stderr,
     "¥n**sqlcode=(%d)",sqlca.sqlcode);

 /*
 ** void warning_handler()
 **
 **  Displays warning messages.
 */
 void warning_handler()
 {

     if (sqlca. sqlwarn[1] == 'W')
     {
         fprintf(stderr, "¥n** Data truncated.¥n");
     }

     if (sqlca.sqlwarn[3] == 'W')
     {
         fprintf(stderr, "¥n** Insufficient
             host variables to store results.¥n");
     }
     return;
 }
```

プリコンパイラが検出するエラー

Embedded SQL プリコンパイラは、プリコンパイル時に Embedded SQL のエラーを検出します。プリコンパイラは、セミコロンが見つからなかったり、宣言されていないホスト変数が SQL 文で使われているなどの構文エラーを検出します。これらは、重大なエラーであるため、適切なエラー・メッセージが生成されます。

プリコンパイラで Transact-SQL 構文エラーをチェックすることもできます。Adaptive Server Enterprise は、適切なプリコンパイラ・コマンド・オプションが設定されている場合、プリコンパイル時に Transact-SQL 文を解析します。詳細については、使用しているプラットフォームの『Open Client/Server プログラマーズ・ガイド補足』のプリコンパイラのリファレンス・ページを参照してください。

プリコンパイラは、次の例のようにテーブルを作成してそこからデータを選択する場合にはエラーを検出しません。エラーは、ホスト変数のデータ型が取得したカラムに合わないということです。プリコンパイラが文を解析する時点では、テーブルはまだ存在しないため、プリコンパイラはエラーを検出しません。

```
exec sql begin declare section;
 CS_INT   var1;
 CS_CHAR  var2[20];
 exec sql end declare section;

 exec sql create table
     T1 (col1 int, col2 varchar(20));
 ....

 exec sql select * from T1 into :var2, :var1;
```

エラーは実行時に検出され、レポートされることに注意してください。

第 9 章

継続バインドによるパフォーマンスの向上

この章では、継続バインドとパフォーマンスを向上させる方法について説明します。継続バインドは Client-Library の機能であり、Embedded SQL 文を実行するルーチンの集まりです。継続バインドを使用すると Embedded SQL プリコンパイラがより効率的なコードを作成できるので、プログラムのパフォーマンスが向上します。

継続バインドはオプションの機能です。つまり、プログラムをプリコンパイルするときにこの機能を要求すれば有効になります。継続バインドは、特定のタイプの Embedded SQL プログラムでのみ効果があります。

この章の内容を理解するには、ホスト変数、カーソル、動的 SQL、プリコンパイラ・オプションについて理解している必要があります。それぞれの詳細については、次の章またはマニュアルを参照してください。

- ホスト変数の詳細については、「第 4 章 変数の使い方」を参照してください。
- カーソルの詳細については、「第 6 章 Transact-SQL 文の使い方」を参照してください。
- 動的 SQL の詳細については、「第 7 章 動的 SQL の使い方」を参照してください。
- プリコンパイラ・オプションとプリコンパイラの起動方法については、『Open Client/Server プログラマーズ・ガイド補足』を参照してください。

Embedded SQL で継続バインドを使用するために Client-Library について理解する必要はありませんが、Client-Library のコマンド構造体、ct_bind ルーチン、ct_fetch ルーチンについて理解していれば、Embedded SQL で継続バインドがなぜそのように機能するかを理解するのに役立ちます。

コマンド構造体 ct_bind、ct_fetch の一般的な機能については、この章で簡単に説明します。詳細については、『Open Client Client-Library/C プログラマーズ・ガイド』と『Open Client Client-Library/C リファレンス・マニュアル』を参照してください。

継続バインドの概要

Adaptive Server に値を渡したり、Adaptive Server からの値を保管したりするために、Embedded SQL プログラムはホスト変数、つまり Embedded SQL が認識する C 変数を使用します。プログラムは、これらの変数を Adaptive Server での値と対応させます。たとえば次の select 文は、ホスト出力変数 *last* を Adaptive Server から取得したロー値と対応させます。

```
id = "998-72-3567";
 exec sql select au_lname into :last
 from authors where au_id = :id;
```

この文はそのホスト入力変数 *id* を Adaptive Server に渡して、その変数をサーバの *au_id* カラムと対応させます。

ある文のホスト変数を Adaptive Server の値に対応させる作業は、「バインド」と呼ばれます。対応付け自体もバインドと呼ばれます。ホスト入力変数は入力バインドだけを使用し、ホスト出力変数は出力バインドだけを使用します。

バインドは、ある文がサーバからどのデータを取得するかを制御します。ある文がホスト変数を不正なサーバ・データにバインドした場合、その文はそのホスト変数に対して不正な値を取得します。ただし、必要でないバインドを行うとプログラムのパフォーマンスが低下する可能性があります。

Embedded SQL では、バインドがどのくらいの間有効になるか、つまりどのくらい「継続する」かをプログラマが制御できます。1 つの文を複数回実行する間継続するバインドを、「継続バインド」と呼びます。継続バインドを実行するといくつかの Embedded SQL 文をより速く実行できるので、プログラムのパフォーマンスが向上します。

Embedded SQL では、それぞれのバインドは Client-Library の「コマンド構造体」、つまり特に Embedded SQL 文のバインドを定義するデータ構造体によって実行されます。実行する Embedded SQL 文ごとに対応するコマンド構造体があります。ただし、単一のコマンド構造体を複数の文で使用することは可能です。実際、ある Embedded SQL 文から別の文へとバインドが継続する場合は、これらの文が単一のコマンド構造体を共有するので、単一のコマンド構造体を複数の文で使用します。

Embedded SQL プログラムのソース・コードは、コマンド構造体を明示的に宣言したり割り付けたりしません。反対にコマンド構造体は、プログラムの生成されたコードによって宣言されて割り付けられます。

バインドが行われるタイミング

デフォルトでは、ホスト変数を使用して Embedded SQL 文が実行されるたびにバインドが行われます。ループ内のように 1 つの Embedded SQL 文が 2 度以上実行される場合は、文が実行されるたびにバインドが行われます。たとえば次のようなループでは、insert 文が実行されるたびにそのホスト変数は同じ Adaptive Server 値と対応付けられます。さらにデフォルトで、Embedded SQL 文が実行されるたびにバインドが行われます。

```
for (i = 1; i <= 3; i++)
 {
  exec sql insert into titles (title_id, title)
  values (:bk_id, :bk_title);
  /*
  ** Binding occurs here at each execution.
  ** When a statement undergoes binding, all
  ** its host variables get bound.
  */
 }
```

ほとんどの文では、ユーザが継続バインドを要求しても、バインドはある文から次の文まで継続しません。たとえば次の insert 文はすべて同じで連続していますが、バインドを共有しません。

```
exec sql insert into titles (title_id, title)
     values (:bk_id, :bk_title);
 /* Binding occurs for the first statement.  */

 exec sql insert into titles (title_id, title)
     values (:bk_id, :bk_title);
 /* Binding occurs for the second statement.  */

 exec sql insert into titles (title_id, title)
     values (:bk_id, :bk_title);
 /* Binding occurs for the third statement.  */
```

先頭に for が指定されたループでの insert 文など、2 度以上実行される Embedded SQL 文の場合、文が最初に実行されるときだけバインドを行うのか、それとも後続のそれぞれの文の実行でもバインドを行うかを指定できます。

継続バインドを制御するには、プリコンパイラ・オプションを使用して、ファイル内のすべての文のバインド動作を指定します。プリコンパイラ・オプションを使用してもユーザは個々の文のバインド動作を制御できません。バインドを制御するプリコンパイラ・オプションについては、このあとの項を参照してください。

継続バインドを使用すると便利なプログラム

すべての Embedded SQL プログラムで継続バインドを使用しても、必ずしも便利とはかぎりません。次の質問によって、どのようなユーザのプログラムで継続バインドを使用すると便利かを判断できます。

1 2 度以上実行される Embedded SQL 文がプログラムに 1 つ以上ありますか。

2 1 で「はい」と答えた場合、その文は同じホスト変数を繰り返し使用して、Adaptive Server と値を交換しますか。

上記の質問の答えが両方とも「はい」の場合は、たいていそのプログラムで継続バインドを使用すると便利です。どちらかの答えが「いいえ」の場合は、両方の質問に「はい」と答えることができるようにプログラムを修正しないかぎり、継続バインドを使用してもプログラムのパフォーマンスは向上しません。

継続バインドを使用して最大限の効果を得るには、プログラムで 2 つ以上の同一の文を実行する代わりに、単一の Embedded SQL 文を繰り返し実行してください。たとえば次の insert 文は繰り返し実行されます。

```
for (i = 1; i <= 3; i++)
 {
  exec sql insert into titles (title_id, title)
  values (:bk_id, :bk_title);
 }
```

この例の insert 文は 3 度実行されますが、その変数がバインドされるのは一度だけです。バインドが繰り返されないので、この例の文は、一度だけ実行される同一の insert 文が連続している場合よりも速く実行されます。

継続バインドのスコープ

継続バインドのスコープ、つまり継続バインドがどのくらい継続するかは、このあとの項で説明するように、文のタイプと有効になっているプリコンパイラ・オプションによって異なります。ただし、接続の継続期間を超えてバインドが継続することはありません。プログラムによって接続がクローズされた場合、実行される文に対するすべてのバインドと、その接続にわたって割り付けられたすべてのコマンド構造体がキャンセルされます。

継続バインドのプリコンパイラ・オプション

バインドは 2 つのプリコンパイラ・オプション、-p オプションと -b オプションによって制御されます。これらのオプションの影響を受けるのは、継続バインドを使用できる Embedded SQL 文だけです (継続バインドを使用できない文のリストについては、表 9-1 を参照してください)。

-p オプションの概要

-p オプションは、それぞれの文が「継続コマンド構造体」、つまり特定の文のすべての実行の間継続する構造体を持っているかどうかを制御します。継続コマンド構造体を持つ文だけが、ホスト変数の継続バインドを行うことができます。このように、-p オプションは、ホスト入力変数のバインドを制御し、その変数の値は Adaptive Server Enterprise に渡されます (この章の「ホスト入力変数」についての説明は、値が Adaptive Server に渡されるその他の変数にも当てはまります。例外事項については本文中に示します)。

-b オプションの概要

-b オプションは、Adaptive Server Enterprise から結果データを取得する文で使用されるホスト変数のバインドを制御します。-b オプションを -p オプションとともに使用した場合は、select 文と exec 文のホスト変数のバインドを制御します。-b オプションを単独で使用した場合は、カーソルを使用してフェッチする文だけを制御できます。

このように、-b オプションは、出力変数、結果変数、ステータス変数、インジケータ変数など、値を Adaptive Server Enterprise との間で受け渡しするホスト変数のバインドを制御します (「ホスト出力変数」についての説明は、値が Adaptive Server から出力される他のすべての変数にも当てはまります)。正確には、Client-Library の ct_fetch ルーチンが実行されるたびにバインドが行われるかどうかを -b オプションによって制御します (ct_fetch ルーチンは、Adaptive Server からデータの単一のローを取得します)。

使用するオプション：*-p*、*-b*、またはその両方

入力変数に対して継続バインドを使用すると便利な大部分のプログラムでは、出力変数に対しても継続バインドを使用すると便利です。一般に、-p オプションと -b オプションは、両方とも使用するか、どちらも使用しないかのいずれかにしてください。

プリコンパリラ・オプション *-p* と *-b* のスコープ

そのファイルでカーソルが宣言されない場合、-p オプションと -b オプションはプリコンパイルされるファイルだけに影響を与えます。ファイルでカーソルが宣言された場合、そのカーソルを使用する文がプログラムのさまざまなソース・ファイル内にあるときでも、-p オプションと -b オプションはそのすべての文に影響を与えます。カーソルを使用するファイルでの -p オプションと -b オプションの影響の詳細については、このあとの項を参照してください。

継続バインドの規則の概要

継続バインドの規則は、Embedded SQL 文のタイプによって異なります。規則は特に、文が次の条件に当てはまるかどうかによって異なります。

- 継続バインドを使用する
- カーソルを使用する
- 動的 SQL 文である
- rebind 句または norebind 句を使用した fetch 文である

継続バインドを使用できる文

大部分の Embedded SQL 文では継続バインドを使用できます。継続バインドを使用できない Embedded SQL コマンドについては、表 9-1 と表 9-2 を参照してください。Transact-SQL コマンドを含むその他のすべての Embedded SQL コマンドでは、ホスト変数のいくつかまたはすべてに対して継続バインドを使用できます。

文のバインドが継続するかどうかとバインドの継続時間は、文のタイプ、特にその文がカーソルを使用するかどうかによって決まります。

表 *9-1:* 継続バインドを使用できない *Embedded SQL* コマンド

allocate descriptor	begin transaction
close	commit
connect	deallocate cursor
deallocate descriptor	deallocate prepare
describe input	describe output
disconnect	end transaction
execute	execute immediate
get descriptor	get diagnostics
open using descriptor	prepare
prepare transaction	rollback
set descriptor	set connection
set transaction diagnostics	

表 *9-2:* 継続バインドを使用できない *Embedded SQL* コマンドのタイプ

-y オプションを使用して text データまたは image データを Adaptive Server に送信するコマンド	Adaptive Server Enterprise への入力に SQL 記述子または SQLDA を使用する動的 SQL コマンド

カーソルを使用しない文での継続バインド

継続バインドを使用できるけれどもカーソルを使用しない Embedded SQL 文の場合は、文のプリコンパイル時に -p オプションと -b オプションを使用して文のバインドを制御します。表 9-3 は、カーソルを使用しない文にこれらのオプションがどのように影響を与えるかをまとめたものです。

表 *9-3:* カーソルを使用しない文に対する *-p* と *-b* オプションの影響

文のプリコンパイルに使用するオプション	文のバインドへの影響
-p も -b も使用しない	どのようなバインドも継続しない
-p だけを使用	入力バインドだけが継続する
-b だけを使用	どのようなバインドも継続しない
-p と -b の両方を使用	すべてのバインドが継続する

文のバインドが継続する場合、その文が実行される接続がプログラムによってクローズされるまでバインドは継続します。バインドはその文のすべての実行が終わるまでの間、その他の文が実行されても継続します。文のバインドが継続しない場合は、その文が実行されるたびにバインドが行われます。

カーソルを使用する文での継続バインド

プログラムでカーソルを使用するには、declare cursor コマンドを使用してカーソルを宣言する必要があります。カーソルの宣言は、プログラムのすべてのソース・ファイルで、そのカーソルを使用するすべての文のバインド動作を制御します。このように制御されるのは、カーソルの宣言のためのコマンド構造体をそのカーソルを使用するすべての文が共有しているからです。

文がカーソルを使用する場合、そのカーソルを使用する文ではなく、カーソルの宣言がその文のバインドの継続時間を制御します。カーソルを宣言するファイルのプリコンパイル時に -b オプションと -p オプションを使用した場合にだけ、バインドが継続します。これらのオプションを使用した場合、カーソルを使用するすべての文に対して、オプションで指定された継続バインドが行われます。

厳密に言うと、カーソルの宣言はそのカーソルが「動的カーソル」、つまり動的 SQL 文のカーソルである場合にだけバインド動作を制御します。その他のすべての SQL 文のカーソル (「静的カーソル」) では、そのカーソルを宣言する文ではなく、カーソルを最後にオープンした文 (open cursor) がバインド動作を制御します。

注意　静的カーソルの場合、open cursor を実行した結果生成されたコードはカーソルの宣言とオープンの両方を行います。動的カーソルの場合、open cursor を実行した結果生成されたコードはカーソルのオープンだけを行います。

静的カーソルと動的カーソルのバインド規則は、上記の相違点以外は同じです。プログラムの複数のソース・ファイルで特定のカーソルを使用しないのであれば、静的カーソルと動的カーソルのバインド動作は同じです。

カーソルを使用する文では、ユーザが継続バインドを使用していても、そのカーソルの割り付けが解除されるとバインドは継続しません。また、割り付けが解除されたカーソルは再度オープンできません。割り付けを解除されたカーソルの名前を使用して新しいカーソルを宣言しても、割り付けを解除されたカーソルは再度オープンされず、そのカーソルと対応するバインドも保持されません。「第 10 章 Embedded SQL 文: リファレンス・ページ」の deallocate cursor コマンドの説明を参照してください。

次の例は、-b オプションと -p オプションがカーソル *curs1* にどのように影響を与えるかを示します。この例の fetch 文にはホスト変数が指定されています。この例のあとで、-b オプションと -p オプションがこれらのホスト変数のバインドにどのように影響を与えるかについて説明します。

```
#include <stdio.h>
 int SQLCODE;

 void
 main()
 {
    exec sql begin declare section;
      char title[100], pub_id[8];
    exec sql end declare section;

    exec sql connect "sa";

    exec sql use pubs2;
         /*
         ** The options used to precompile a cursor's declaration
         ** control whether host variables persist in statements,
         ** such as FETCH, that use the cursor.
         */
     exec sql declare curs1 cursor for select title, pub_id from
         titles;
     exec sql open curs1;

     while (SQLCODE == 0)
         {
         /* If the declaration of curs1 was precompiled without
         ** the -b option, rebind the FETCH statement's variables
         ** each time the statement repeats. Otherwise, bind only
         ** the first time, and let the bindings persist for
         ** subsequent repetitions.
         */
         exec sql fetch curs1 into :title, :pub_id;
             printf("%s, %s¥n", title, pub_id);
         }
         /* If the declaration of curs1 was precompiled without
```

```
** the -p option, cancel the bindings of the FETCH
** statement's variables when curs1 is closed.
** Otherwise, let the bindings persist until the
** program deallocates curs1 or, as here, until the
** program ends.
*/
exec sql close curs1;
exec sql disconnect CURRENT;

exit(0);
```

すべてのカーソル・ホスト変数の継続バインドの防止

上記の例のプリコンパイル時に -b オプションと -p オプションの両方を省略した場合は、どのようなバインドも継続しません。その代わり、ホスト変数が Adaptive Server Enterprise に入力されるか Adaptive Server Enterprise から出力されるかにかかわらず、fetch 文が実行されるたびに、生成されたコードによってその変数がバインドされます。

すべてのカーソル・ホスト変数の継続バインドの要求

前述の例のプリコンパイル時に -b オプションと -p オプションの両方を使用した場合、生成されたコードによって、fetch 文のホスト変数はその文が最初に実行されるときにだけバインドされます。他の Embedded SQL 文 (「バインドが行われるタイミング」(95 ページ) で説明) とは異なり、同一の fetch 文が連続しているか、1 つの fetch 文をループで実行するかは問題ではありません。これらのオプションを両方とも使用すると、プログラムによってカーソルがクローズされてもバインドは継続するようになります。そのカーソルが再度オープンされたときに、ホスト変数を再バインドする必要はありません。プログラムでカーソルの割り付けが解除されるまで、バインドは継続します。これには、多くの場合 deallocate cursor 文または disconnect 文が使用されます。

カーソル出力変数だけに対する継続バインドの要求

前述の例のプリコンパイル時に -b を使用して -p を省略した場合、生成されたコードによって、fetch 文のホスト出力変数はその文が最初に実行されるときに一度だけバインドされます (正確には、ホスト変数はまだバインドされていない場合にだけバインドされます)。プログラムによって *curs1* がクローズされるまでは、その文の後続のすべての実行が終了するまでの間バインドが継続します。バインドが継続するのは、-b オプションを使用したからです。ただし -p オプションを省略したので、ホスト入力変数のバインドは継続しません。

プログラムによって *curs1* がクローズされてから再度オープンされた場合、*curs1* に関連するホスト変数のすべてのバインドがキャンセルされます。カーソルが再度オープンされたときに、すべてのホスト入力変数とホスト出力変数が再バインドされます。プログラムによってカーソルが再度クローズされるまで、バインドは継続します。

カーソル入力変数だけに対する継続バインドの要求

前述の例では、カーソルとともにホスト変数を使用する文に -b オプションと -p オプションがどのように影響を与えるかを示しました。この例の唯一のホスト変数はホスト出力変数でした。次は、カーソルとともにホスト入力変数を使用した文に -b オプションと -p オプションがどのように影響を与えるかを示すコード例です。ここでは動的カーソルの名前は *dyn_curs1* です。

次の例の open 文には、ホスト入力変数 *min_price* が指定されています。これ以降の項では、このホスト入力変数のバインドに -b オプションと -p オプションがどのように影響を与えるかについて説明します。

```
#include <stdio.h>
 long SQLCODE = 0;

 void main()
 {
     int i = 0;
     exec sql begin declare section;
         CS_CHAR        sql_string[200];
         CS_FLOAT       min_price;
         CS_CHAR        book_title[200];
     exec sql end declare section;
     exec sql connect "sa";
     exec sql use pubs2;
     strcpy(sql_string,
         "select title from titles where price > ?");
         exec sql prepare sel_stmt from :sql_string;

     /* The options used to precompile a cursor's declaration
     ** control whether host variables persist in statements,
     ** such as OPEN, that use the cursor.
     */
     exec sql declare dyn_curs1 cursor for sel_stmt;
     min_price = 10.00;
     /* If the declaration of dyn_curs1 was precompiled
     ** without -p, bind the OPEN statement's input variable
     ** (min_price) each time the statement repeats.  Otherwise,
     ** bind only the first time, letting the binding persist
     ** until dyn_curs1 is deallocated.
     */

     for (i = 10; i <= 21; ++i)
     {     min_price = min_price + 1.00;
           exec sql open dyn_curs1 using :min_price;
           while (SQLCODE != 100)
               {
                 exec sql fetch dyn_curs1 into :book_title;
                 if (SQLCODE != 100) printf("%s¥n", book_title);
               }
           printf("¥n");
           exec sql close dyn_curs1;
```

```
    }
    exec sql deallocate cursor dyn_curs1;
    exec sql disconnect CURRENT;
    exit(0);
}
```

前述の例のプリコンパイル時に -p オプションだけを使用して -b オプションを省略した場合、生成されたコードによって、*min_price* は open 文が最初に実行されるときに一度だけバインドされます。ホスト入力変数を制御する -p オプションを使用したので、バインドは継続します。

プログラムによって *dyn_curs1* の割り付けが解除されるまでは、*min_price* のバインドはその文の後続のすべての実行が終了するまでの間、継続します。プログラムによって *dyn_curs1* がクローズされてから再度オープンされた場合でも、バインドは継続します。

継続バインド、カーソル、複数のソース・ファイル

前述の例では、カーソル *dyn_curs1* の宣言によって、対応するホスト変数が継続するかどうかが制御されます。このため、fetch 文が個別のソース・ファイルでプリコンパイルされた場合でも、fetch 文中のホスト変数は例で説明されているようにバインドされます。

継続バインドとカーソル *fetch* 文

Embedded SQL fetch コマンドでは、特定の fetch 文でバインドが継続するかどうかを制御する rebind 句または norebind 句をオプションで指定できます。あるファイルに指定したプリコンパイラ・オプションを上書きする必要がある場合には、この句を使用すると便利です。rebind 句または norebind 句は、その句が指定されている文だけに影響を与えます。他の fetch 文を含むその他の文のバインドは影響を受けません。

fetch 文で rebind 句または norebind 句を省略した場合、その文は当該のカーソルを使用するその他のタイプの文と同じバインド規則に従います。

fetch 文にキーワード rebind が指定された場合は、その文のホスト変数のバインドは継続しません。その代わり、その文のカーソルの宣言をプリコンパイルするために -b オプションを使用したかどうかにかかわらず、その文が実行されるたびに再バインドされます。

fetch 文にキーワード norebind が指定されていても -b オプションを使用してその文がプリコンパイルされる場合は、そのキーワードはどのような影響も与えません。

継続バインドを使用するためのガイドライン

この項では、継続バインドを正しく使用するためのガイドライン、ヒント、注意点について説明します。

- 継続バインドを使用すると便利なのは、プログラムが次の両方の基準を満たす場合だけです。
  - 2 度以上実行される Embedded SQL 文が 1 つ以上あること
  - その文が同じホスト変数を繰り返し使用して、Adaptive Server と値を交換すること
- そのファイルでカーソルが宣言されない場合、-p オプションと -b オプションはプリコンパイルされるファイルだけに影響を与えます。ファイルでカーソルが宣言された場合、-p オプションと -b オプションはカーソルを使用するすべての文に影響を与えます。一般に -p オプションと -b オプションは、両方とも使用するか、どちらも使用しないかのいずれかにしてください。プログラムが 2 つ以上の Embedded SQL ソース・ファイルから構成される場合は、-p オプションと -b オプションの同じ組み合わせを使用して、そのすべてのファイルをプリコンパイルしてください。

  あるプログラムの 2 つ以上のソース・ファイルで同じカーソルを使用する場合、通常はこれらのファイルのプリコンパイル時に -p オプションと -b オプションの同じ組み合わせを使用してください。そうしない場合は、オプションの組み合わせが異なることによって、文が送信または取得するデータがどのように変化するかを正確に理解する必要があります。
- 継続バインドを使用するプログラムでは、実施できる場合、複数の同一の文を一度ずつ実行するのではなく、単一の Embedded SQL 文を繰り返し実行してください。
- 文のバインドを制御する規則は、その文が次の条件に当てはまるかどうかによって異なります。
  - 継続バインドを使用する
  - カーソルを使用する
  - 動的 SQL 文である
  - rebind 句または norebind 句を使用した fetch 文である
- 接続の継続期間を超えてバインドが継続することはありません。カーソルを使用する文では、カーソルの割り付けが解除されたあとにバインドが継続することはありません。
- 動的カーソルの宣言によって、そのカーソルを使用するすべての文のバインド動作が制御されます。静的カーソルの場合は、そのカーソルを最後にオープンした文がこのような制御を行います。プログラムでは、静的カーソルを宣言するソース・ファイル内だけでそのカーソルをオープンしてください。

ホスト変数のバインドについての注意事項

ここでは、サブスクリプト付きの配列のホスト変数の動作と、繰り返し実行される場合のホスト変数の動作について説明します。

サブスクリプト付きの配列

-p オプションまたは -b オプションを使用して、サブスクリプト付きの配列のホスト変数 (入力または出力) をバインドする場合、その文が最初に実行されたあとにはサブスクリプトは無視されます。これは、指定された配列要素の実際のアドレスがバインドされるからです。次に例を示します。

```
exec sql begin declare section;
int row;
int int_table[3] = {
   10,
   20,
   30,
};
char *string_table[3] = {
   “how”,
   “are”,
   “you”,
};
exec sql end declare section;
for (row=0; row < 3; row++)
{
   EXEC SQL insert into ...  values (:row, :int_table[row],
        :string_table[row]);
     /*
      ** If this statement is precompiled with -p, only
      ** int_table[0] and string_table[0] will be bound and
      ** inserted each time.
      ** The same thing  applies to output variables
      ** At this time, NO warnings are issued to detect this.
      */
}
```

この問題を解決するには、次の解決策のいずれかを選択できます。

- 再バインドを行いたいのであるから、サブスクリプト付きの配列が使用される場合は継続バインドを使用しないでください (*table[0] は次の繰り返しの *table[1] とは同じではありません)。
- 継続バインドを使用する必要がある場合は、現在の値を保持する中間変数を使用してください。この方法を使用すると、エラーもなく継続バインドが行われます。ただし、データをコピーすることによってオーバヘッドが生じます。上記の例を使用すると、次のようになります。

```
exec sql begin declare section;
char    bind_str[80];
int bind_int_variable;
exec sql end declare section;
for (row=0; row < 3; row++)
{
      /*
       ** Must copy the contents- pointer assignment does
       ** not suffice host var 'row' is not a subscripted
       ** array, so it can remain the same.
       */
   memcpy(bind_str, string_table[row],80);
   bind_int_variable = int_table[row];
   EXEC SQL insert into ...  values (:row,
           :bind_int_variable,
         :bind_str);
}
```

注意 継続バインドではレジスタ変数は使用できません。

ホスト変数のスコープ

ホスト変数がある実行から次の実行までバインドされたままになっている場合、その変数がまだスコープ内にあることを確認する必要があります。スタック変数などの自動変数が使用される場合には特に注意してください。

プリコンパイラが問題のありそうな状況を検出できた場合は、警告が表示されます。ホスト変数がまだスコープ内にあるかどうかも、全体のプログラム論理によって決まります。

次に例を示します。

```
 /*
  ** a function called by main()
  */
CS_VOID insert(insert_row)
exec sql begin declare section;
int insert_row;/* row will go out of scope once exit
                ** function*/
exec sql end declare section;
{
   /*
       ** id is a stack variable which will go out of scope
       ** once we exit the function insert()
       ** it is not likely to be at the same address at the
       ** next call to this function, so if it is bound as
       ** an input variable, there will be errors.
       */
   exec sql begin declare section;
```

```
   int id;
   exec sql end declare section;
   exec sql insert values(:row,:id);
}
int fetched_row;/* this variable can be safely bound with
                ** persistence */
main()
{
   exec sql begin declare section;
   /*
       ** This variable will go out of scope when the program
       ** exits main, which is not a problem.
       */
   int row;
   /*
       ** This variable is a pointer, thus it does not
       ** necessarily pose problems, depending on the scope
       ** of the data it is pointing to.
       */
   char *pointer;
   exec sql end declare section;
   for (row = 0; row < 10; row++)
   {
        insert(row);
   }
}
```

第 10 章

Embedded SQL 文：リファレンス・ページ

この章は、Transact-SQL にはない Embedded SQL 文、および Transact-SQL の場合とは機能が異なる Embedded SQL 文のリファレンス・ページから構成されています。Embedded SQL で有効なその他のすべての Transact-SQL 文の詳細については、『ASE Transact-SQL ユーザーズ・ガイド』を参照してください。

print、readtext、writetext 以外のすべての Transact-SQL 文を Embedded SQL で使用できます。ただし、いくつかの文の構文はこの章で説明されているように異なっています。

この章のリファレンス・ページはアルファベット順になっています。それぞれの文のリファレンス・ページの内容は次のとおりです。

- その文の機能についての簡単な説明
- その文の構文の説明
- その文のキーワードとオプションについての説明
- その文の適切な使用方法についてのコメント
- 関連する文がある場合にはそのリスト
- その文の簡単な使用例

allocate DESCRIPTOR

説明

SQL 記述子を割り付けます。

構文

exec sql allocate descriptor *descriptor_name*
[with max [*host_variable* | *integer_literal*]];

パラメータ

descriptor_name
準備文内の動的パラメータ・マーカについての情報が入っている SQL 記述子の名前です。

with max
SQL 記述子内のカラムの最大数です。

host_variable
declare セクション内で定義された整数ホスト変数です。

integer_literal
SQL 記述子をいくつ使用するかを表す数値です。

例

```
exec sql begin declare section;
 CS_INT      type;
 CS_INT      numcols, colnum;
 exec sql end declare section;
 ...
 exec sql allocate descriptor big_desc
     with max 1000;
 exec sql prepare dynstmt from "select * from
     huge_table";
 exec sql execute dynstmt into sql descriptor
     big_desc;
 exec sql get descriptor :numcols = count;
 for (colnum = 1; colnum <= numcols; colnum++)
 {
     exec sql get descriptor big_desc :type = type;
         ...
 }
 exec sql deallocate descriptor big_desc;
         ...
```

使用法

- allocate descriptor コマンドは、Adaptive Server Enterprise が割り付ける項目記述子領域の数を指定します。
- 割り付けできる SQL 記述子の数には制限がありません。
- SQL 記述子が割り付けられるとき、そのフィールドは未定義です。
- すでに割り付けられている SQL 記述子を割り付けようとすると、エラーが発生します。
- with max 句に値を指定していない場合は、1 つの項目記述子が割り当てられます。

- SQL 記述子が割り付けられるとき、そのフィールドのそれぞれの値は未定義です。

参照 deallocate descriptor、get descriptor、set descriptor

begin declare section

説明 Embedded SQL ソース・ファイルで使用されるホスト言語変数を宣言する declare セクションを開始します。

構文

```
exec sql begin declare section;
 host_variable_declaration;
 ...
exec sql end declare section;
```

パラメータ *host_variable_declaration*

1 つ以上のホスト言語変数の宣言です。

例

```
exec sql begin declare section;
     CS_CHAR    name(80);
     CS_INT     value;
 exec sql end declare section;
```

使用法

- declare セクションの終了には、Embedded SQL の end declare section 文を使用します。
- ソース・ファイルに指定できる declare セクションの数には制限がありません。
- 変数を宣言できる場所であればどこにでも declare セクションを指定できます。変数を宣言する declare セクションは、その変数を参照する文よりも前に指定してください。
- declare セクションでの変数の宣言は、ホスト言語の規則に準拠する必要があります。
- declare セクションではネストされた構造体を使用できます。ただし、構造体の配列は使用できません。
- declare セクションで指定できる Embedded SQL の include 文の数には制限がありません。
- Embedded SQL/C ルーチンでは、*cspublic.h* 内で定義された Client-Library データ型を declare セクションで使用できます。
- C ルーチンでは、文字については 2 次元配列を宣言できますが、その他のデータ型については 1 次元配列だけです。

- declare セクションを処理する場合、Embedded SQL プリコンパイラは C プリプロセッサ・マクロと #include 文を無視します。declare セクションにある Embedded SQL include 文を処理する場合、Embedded SQL プリコンパイラは、プリコンパイルされるファイルに直接入力されたようにインクルード・ファイルの内容を処理します。

参照　exec sql include "filename"

begin transaction

説明　非連鎖トランザクションの始めにマークを付けます。

構文　exec sql [at *connection_name*]
begin {transaction | tran} [*transaction_name*];

パラメータ　transaction | tran

キーワードは transaction と tran のどちらでも使用できます。

transaction_name

このトランザクションに割り当てられる名前です。この名前は、Transact-SQL 識別子の規則に準拠しなければなりません。

例

```
/*
 ** Use explicit transactions to
 ** syncronize tables on two servers
 */
 exec sql begin declare section;
     char       title_id[7];
     int        num_sold;
 exec sql end declare section;
     long       sqlcode;
     ...

 exec sql whenever sqlerror goto abort_tran;
 try_update:

exec sql at connect1 begin transaction;

 exec sql at connect2 begin transaction;
 exec sql at connect1 select sum(qty)
     into :num_sold
     from salesdetail
     where title_id = :title_id;
 exec sql at connect2 update current_sales
     set num_sold = :num_sold
     where title_id = :title_id;
 exec sql at connect2 commit transaction;
 exec sql at connect1 commit transaction;
 if (sqlcode != 0)
     printf("oops, should have used 2-phase
```

```
        commit¥n");
    return;
    abort_tran:
    exec sql whenever sqlerror continue:
    exec sql at connect2 rollback transaction;
    exec sql at connect1 rollback transaction;
    goto try_update;
```

使用法

- このリファレンス・ページでは、主に Transact-SQL の begin transaction 文を Embedded SQL で使用した場合の相違点について説明します。『ASE リファレンス・マニュアル』を参照してください。
- begin transaction 文は非連鎖トランザクション・モードでのみ有効です。連鎖トランザクション・モードでは、トランザクションの開始点に明示的にマークを付けることはできません。
- トランザクションをネストする場合、最も外側の begin transaction 文とその文に対応する commit transaction 文または rollback transaction 文だけにトランザクション名を割り当てます。
- データベース・オプション ddl in tran を設定しないかぎり、Adaptive Server Enterprise では、次の各文を非連鎖トランザクションの中で使用することはできません。create database、create table、create index、create view、drop、select into table_name、grant、revoke、alter database、alter table、truncate table、update statistics、load database、load transaction、disk init
- 1 つのトランザクションには、そのトランザクションの開始時に使用している、現在の接続で実行される文だけが含まれています。
- リモート・プロシージャは、それらのプロシージャが含まれているトランザクションとは関係なく実行されます。

参照

commit transaction、 commit work、 rollback transaction、 rollback work

close

説明

オープンしているカーソルをクローズします。

構文

exec sql [at *connection_name*] close *cursor_name*;

パラメータ

cursor_name

クローズするカーソルの名前です。この名前はカーソルを宣言したときに割り当てた名前です。

例

```
long SQLCODE;
exec sql begin declare section;
 CS_CHAR        mlname[40];
 CS_CHAR        mfname[20];
 CS_CHAR        phone[12];
 exec sql end declare section;
```

```
exec sql declare author_list cursor for
         select au_lname, au_fname, phone
         from authors;
exec sql open author_list;
while (SQLCODE == 0) {
         exec sql fetch author_list into
         :mlname, :mfname, :mphone;

       if(SQLCODE != 100)
         printf("%s, %s, %s¥n", mlname, mfname,
             mphone);
 }
exec sql close author_list;
```

使用法

- close 文はオープンしているカーソルをクローズします。フェッチされなかったローはキャンセルされます。
- クローズしたカーソルを再度オープンすると、対応するクエリが再実行されて、結果セットの最初のローの前にカーソル・ポインタが置かれます。
- カーソルは再度オープンする前にクローズしなければなりません。
- オープンしていないカーソルをクローズしようとすると、実行時エラーが発生します。
- commit transaction 文、rollback transaction 文、commit work 文、rollback work 文はカーソルを自動的にクローズしますが、プリコンパイラのオプションを使用してその動作を無効にすることもできます。
- カーソルを一度クローズしてから再度オープンすると、そのカーソルがローを取得するテーブルに対して行われたすべての変更を、プログラム側で認識できます。

参照　declare cursor、fetch、open、prepare

commit

説明　トランザクション中のデータベースへの変更を保存し、トランザクションを終了します。

構文

exec sql [at *connection_name*]
commit [transaction | tran | work]
[*transaction_name*];

パラメータ

transaction | trans | work

rollback 文では、キーワード transaction、trans、work は交換可能ですが、work だけは ANSI 準拠です。

transaction_name

トランザクションに割り当てられた名前です。

例

```
/*
 ** Using chained transaction mode,
 ** synchronize tables on two servers
 */
 exec sql begin declare section;
     char     title_id[7];
     int      num_sold;
 exec sql end declare section;
     long     SQLCODE;
     ...
 try_update:
 exec sql whenever sqlerror goto abort_tran;

 exec sql at connect1 select sum(qty)
     into :num_sold
     from salesdetail
     where title_id = :title_id;
 exec sql at connect2 update current_sales
     set num_sold = :num_sold
     where title_id = :title_id;
exec sql at connect2 commit work;
 exec sql at connect1 commit work;
 return;
abort_tran:
 printf("oops, should have used 2-phase commit¥n");
 exec sql whenever sqlerror continue;
 exec sql at connect2 rollback work;
 exec sql at connect1 rollback work;
 goto try_update;
```

使用法

- このリファレンス・ページでは、主に Transact-SQL の commit 文を Embedded SQL で使用した場合の相違点について説明します。『ASE リファレンス・マニュアル』を参照してください。
- トランザクション名は、識別子についての Transact-SQL の規則に準拠している必要があります。トランザクション名は Transact-SQL の拡張機能です。したがって、ANSI 標準に準拠するキーワード work とともに使用することはできません。
- トランザクションをネストする場合、最も外側の begin transaction 文とその文に対応する commit transaction 文または rollback transaction 文だけにトランザクション名を割り当てます。

参照

begin transaction、commit work、rollback transaction、rollback work

connect

説明

Adaptive Server Enterprise との接続を確立します。

構文

exec sql connect *user_name*
[identified by *password*] [at *connection_name*]
[using *server_name*];

パラメータ

user_name

Adaptive Server Enterprise にログインするときに使用するユーザ名です。

password

Adaptive Server Enterprise にログインするときに使用するパスワードです。

connection_name

Adaptive Server Enterprise 接続をユニークに識別するのに使用する名前です。

server_name

接続している Adaptive Server Enterprise の名前です。

例

```
exec sql begin declare section;
     CS_CHAR       user[32];
     CS_CHAR       password[32];
     CS_CHAR       server[90];
     CS_CHAR       conname[20];
 exec sql end declare section;

 strcpy(user, "mylogin");
 strcpy(password, "mypass");
 strcpy(server, "YOURSERVER");
 strcpy(conname, "con_one");
exec sql connect :user identified by :password
     using :server at :conname;
```

使用法

- すべての Embedded SQL プログラムで、connect 文は allocate descriptor 文以外のすべての実行 SQL 文よりも前に実行される必要があります。
- C 言語と COBOL 言語の両方を使用するプログラムでは、最初の connect 文は COBOL プログラムから発行しなければなりません。
- プログラムが複数の接続を持っている場合、1 つの接続だけが名前を指定しないで使用でき、その場合はこの接続がデフォルトの接続になります。
- 特定の名前付き接続に指示する at *connection_name* 句が Embedded SQL 文に存在しないと、その文は現在の接続上で実行されます。
- null パスワードを指定するには、identified by 句を省略するか、空の文字列を使ってください。
- connect 文が Adaptive Server Enterprise を指定しないと、DSQUERY 環境変数または論理名によって指定されたサーバを使用します。DSQUERY が定義されていない場合のデフォルトのサーバは SYBASE です。
- Client-Library は、SYBASE 環境変数または論理名によって指定されたディレクトリ内にある interfaces ファイル内のサーバ名を探します。

- Embedded SQL プログラムが終了するか、または disconnect 文を発行すると、Adaptive Server Enterprise 接続が終了します。
- 名前付きまたは名前なしに関わらず新しい接続をオープンすると、新しい接続が現在の接続になります。
- 複数の Adaptive Server Enterprise ログイン名を必要とするプログラムは、ログイン・アカウントごとに接続を持つことができます。
- 2 つ以上のサーバに接続することによって、プログラムは、異なるサーバに格納されたデータに同時にアクセスできます。
- 1 つのプログラムは 1 台のサーバへの複数接続、または、異なるサーバへの複数接続を持つことができます。
- 表 10-1 は、接続名の付け方を示します。

表 *10-1:* 接続名の付け方

使用されている句	使用されていない句	接続名
at *connection_name*		*connection_name*
using *server_name*	at	*server_name*
なし		DEFAULT

参照 at connection_name、exec sql、disconnect、set connection

deallocate cursor

説明 静的 SQL 文または動的 SQL 文のカーソルの割り付けを解除します。

構文 exec sql [at *connection_name*] deallocate cursor *cursor_name*;

パラメータ *cursor_name*

割り付けを解除するカーソルの名前です。*cursor_name* は、“*my_cursor*” または *my_cursor* のように、二重引用符で囲まれた文字列か引用符のない文字列にしてください。cursor_name にはホスト変数は指定できません。

例

```
exec sql include sqlca;
main()
{
exec sql begin declare section;
     CS_CHAR title[80];
     CS_SMALLINT i_title;
exec sql end declare section;
exec sql whenever sqlerror call error_handler();
exec sql whenever sqlwarning call error_handler();
exec sql whenever not found continue;
exec sql connect “sa”;
```

```
exec sql use pubs2;
exec sql declare title_list cursor for select title from titles;

exec sql open title_list;
for (;;)
{
     exec sql fetch title_list into :title :i_title;
     if (sqlca.sqlcode == 100)  break;

     if (i_title == -1)  printf("Title is NULL.¥n");

     printf("Title:%s¥n", title);
}
exec sql close title_list;
exec sql deallocate cursor title_list;
exec sql disconnect all;
exit(0);
}
error_handler()
{
printf("%d¥n%s¥n",sqlca.sqlcode,sqlca.sqlerrm.sqlerrmc);
exec sql deallocate cursor title_list;
exec sql disconnect all;
exit(-
}
```

使用法

- カーソルの割り付けを解除すると、そのカーソルに割り付けられたすべてのリソースが解放されます。特に deallocate cursor 文は、そのカーソルに対応する Client-Library コマンド・ハンドルと CS_COMMAND 構造体を削除します。
- 静的カーソルのオープン後はいつでもそのカーソルの割り付けを解除できます。また、動的カーソルの宣言後はいつでもそのカーソルの割り付けを解除できます。
- *cursor_name* がオープンしている場合、deallocate cursor 文はそのカーソルをクローズしてからその割り付けを解除します。
- 割り付けが解除されたカーソルは、参照したり再度オープンしたりすることはできません。このようなカーソルを参照または再度オープンしようとすると、エラーが発生します。
- 割り付けが解除されたカーソルと同じ名前を使用して新しいカーソルを宣言できます。割り付けが解除されたカーソルと同じ名前を使用してカーソルをオープンすることは、割り付けが解除されたカーソルを再度オープンすることと同じではありません。新しいカーソルは、割り付けが解除されたカーソルとは名前以外には何も共有しません。
- 割り付けが解除されたカーソルと同じ名前を使用して新しいカーソルを宣言すると、プリコンパイラが警告メッセージを表示する場合もあります。

- deallocate cursor 文は Sybase の拡張機能であり、SQL 標準では定義されていません。

注意 Embedded SQL プログラムで継続バインドを使用する場合は、注意して deallocate cursor 文を使用してください。カーソルの割り付けの解除を必要もないのに行うと、継続バインドの利点がなくなる可能性があります。

参照 close cursor、declare cursor、open (**静的カーソル**)

deallocate descriptor

説明 SQL 記述子の割り付けを解除します。

構文 exec sql deallocate descriptor *descriptor_name*;

パラメータ *descriptor_name*

準備文内の動的パラメータ・マーカまたは戻り値についての情報が入っている SQL 記述子の名前です。

例

```
exec sql begin declare section;
     CS_INT   numcols, colnum;
 exec sql end declare section;
     ...
 exec sql allocate descriptor big_desc
     with max 1000;
 exec sql prepare dynstmt from "select * from
     huge_table";
 exec sql execute dynstmt into sql descriptor
     big_desc;
 exec sql get descriptor :numcols = count;
 for (colnum = 1; colnum <= numcols; colnum++)
 {
     exec sql get descriptor big_desc
          ...
 }
exec sql deallocate descriptor big_desc;
          ...
```

使用法 まだ割り付けられていない SQL 記述子の割り付けを解除しようとすると、エラーが発生します。

参照 allocate descriptor

deallocate prepare

説明

prepare 文を使用して準備された動的 SQL 文の割り当てを解除します。

構文

exec sql [at *connection_name*]
 deallocate prepare *statement_name*;

パラメータ

statement_name
文が準備されたときに、動的 SQL 文に割り当てられる識別子です。

例

```
exec sql begin declare section;
   CS_CHAR        sqlstmt[100];
 exec sql end declare section;
strcpy(sqlstmt, "select * from publishers");
exec sql prepare make_work from :sqlstmt;
exec sql declare make_work_cursor cursor for
     make_work;
exec sql deallocate prepare make_work;
```

使用法

- prepare 文で文を指定してから、その文の割り付けを解除してください。prepare 文で指定されていない文の割り付けを解除しようとすると、エラーが発生します。
- *statement_name* は文バッファをユニークに識別する必要があり、変数の命名についての SQL 識別子の規則に準拠する必要があります。*statement_name* にはリテラルと文字配列ホスト変数のどちらかを指定できます。
- deallocate prepare 文は、*statement_name* に対して宣言されたすべての動的カーソルのクローズと割り付けの解除を行います。

警告！ Embedded SQL プログラムで継続バインドを使用する場合は、注意して deallocate prepare 文を使用してください。prepare 文で指定されていない文の割り付けの解除を必要もないのに行うと、継続バインドの利点がなくなる可能性があります。

参照

declare cursor (動的)、execute、execute immediate、prepare

declare cursor (動的)

説明

準備した動的 select 文によって返される複数のローを処理するためにカーソルを宣言します。

構文

exec sql [at *connection_name*]
 declare *cursor_name*
 cursor for *prepped_statement_name*;

パラメータ

cursor_name

open 文、fetch 文、close 文の中のカーソルを参照するのに使用するカーソルの名前です。カーソルの名前は各接続上でユニークでなければなりません。また、その名前は 255 文字以下にしてください。

prepped_statement_name

実行する select 文を表す (前の prepare 文内で指定された) 名前です。

例

```
exec sql begin declare section;
CS_CHAR        sqlstmt[100];
exec sql end declare section;
strcpy(sqlstmt, "select * from publishers");
exec sql prepare make_work from :sqlstmt;
exec sql declare make_work_cursor cursor for
     make_work;
exec sql deallocate prepare make_work;
```

使用法

- *prepped_statement_name* には、compute 句を指定できません。
- *prepped_statement_name* が prepare 文で指定された接続で *cursor_name* を宣言してください。
- 静的 declare cursor 文が単に宣言であるのに対して、動的 declare cursor 文は実行文です。ホスト言語で実行文を使用できる場所に動的 declare 文を置いてください。また、そのホスト言語のプログラムは実行文のリターン・コード (SQLCODE、SQLCA、または SQLSTATE) をチェックする必要があります。
- 動的カーソルの for update 句と read only 句は declare cursor 文の一部ではありませんが、prepare 文で指定された文の select クエリ内に含まれている必要があります。

参照

close、connect、fetch、open、prepare

declare cursor (静的)

説明

select 文によって返される複数のローを処理するためにカーソルを宣言します。

構文

exec sql declare *cursor_name*
 cursor for *select_statement*
 [for update [of *col_name_1* [, *col_name_n*]…]|
 for read only];

パラメータ

cursor_name

open 文、fetch 文、close 文の中のカーソルを参照するのに使用するカーソルの名前です。カーソルの名前は各接続上でユニークでなければなりません。また、その名前は 255 文字以下にしてください。

select_statement

カーソルのオープン時に実行される Transact-SQL の select 文です。『ASE リファレンス・マニュアル』の select 文の説明を参照してください。

for update

更新できるカーソルの結果リストを指定します (結果リストを更新するには update 文を使用します)。

of *col_name_1*

更新する最初のカラム名です。

of *col_name_n*

更新する *n* 番目のカラム名です。

for read only

更新できないカーソルの結果リストを指定します。

例

```
main()
 {
  exec sql begin declare section;
         CS_CHAR       b_titleid[TIDSIZE+1];
         CS_CHAR       b_title[65];
         CS_CHAR       b_type[TYPESIZE+1];
  exec sql end declare section;
         long       SQLCODE;
  exec sql connect “sa”;
  exec sql use pubs2;
  exec sql declare titlelist cursor for
     select title_id, substring(title,1,64)
     from titles where type like :b_type;
  strcpy(b_type, "business");
  exec sql open titlelist;
  for (;;)
  {
         exec sql fetch titlelist into :b_titleid,
             :b_title;
         if (SQLCODE == 100)
             break;
         printf("   %-8s %s¥n", b_titleid, b_title);
```

```
    }
    exec sql close titlelist;
    exec sql disconnect all;
}
```

使用法

- Embedded SQL プリコンパイラは、declare cursor 文に対してはコードを生成しません。
- open cursor 文を使用してプログラムがカーソルをオープンするまで、*select_statement* は実行されません。
- Embedded SQL では compute 句を使用できない点を除けば、*select_statement* の構文は『ASE リファレンス・マニュアル』に記載されている説明と同じです。
- *select_statement* にはホスト変数を指定できます。ホスト変数の値は、プログラムがカーソルをオープンするときに置き換えられます。
- for update 句と read only 句のどちらかを省略した場合、Adaptive Server Enterprise はそのカーソルが更新可能かどうかを判断します。

参照

close、connect、deallocate cursor、declare cursor (ストアド・プロシージャ)、declare cursor (動的)、fetch、open、update

declare cursor (ストアド・プロシージャ)

説明

ストアド・プロシージャに対してカーソルを宣言します。

構文

exec sql declare *cursor_name*
cursor for execute *procedure_name*
([[@*param_name* =]:*host_var*]
[,[@*param_name* =]:*host_var*]…)

パラメータ

cursor_name

open 文、fetch 文、close 文の中のカーソルを参照するのに使用するカーソルの名前です。カーソルの名前は各接続上でユニークでなければなりません。また、その名前は 255 文字以下にしてください。

procedure_name

実行するストアド・プロシージャの名前です。

param_name

ストアド・プロシージャ内のパラメータの名前です。

host_var

パラメータ値として渡されるホスト変数の名前です。

例

```
main()
{
  exec sql begin declare section;
     CS_CHAR        b_titleid[7];
     CS_CHAR        b_title[65];
     CS_CHAR        b_type[13];
  exec sql end declare section;
    long        SQLCODE;
  exec sql connect "sa";
  exec sql use pubs2;
  exec sql
    create procedure p_titles
      (@p_type varchar(30)) as
      select title_id, substring(title,1,64)
        from titles
        where type like @p_type;
  exec sql declare titlelist cursor for
    execute p_titles (:b_type);
  strcpy(b_type, "business");
  exec sql open titlelist;
  for (;;)
  {
    exec sql fetch titlelist into :b_titleid,
      :b_title;
    if (SQLCODE == 100)
      break;
      printf("   %-8s %s¥n", b_titleid, b_title);
  }
  exec sql close titlelist;
  exec sql disconnect all;
}
```

使用法

- *procedure_name* は、1 つの select 文だけで構成してください。
- カーソルを使用して実行したストアド・プロシージャの出力パラメータを取得することはできません。
- カーソルを使用して実行したストアド・プロシージャのリターン・ステータスを取得することはできません。

参照

close、deallocate cursor、declare cursor (静的)、declare cursor (動的)、fetch、open、update

declare scrollable cursor

説明

データのローを返す select 文に対するカーソルを宣言します。カーソルは、使用する前に宣言しておいてください。declare セクション内では、カーソルは宣言できません。

構文

exec sql declare *cursor_name* [*cursor sensitivity*]
[*cursor scrollability*] cursor for *select_statement* ;

パラメータ

cursor_name
カーソルを識別する名前です。

cursor sensitivity
カーソルの変更反映の可否を指定します。

cursor scrollability
カーソルがスクロール可能かどうかを指定します。

select_statement
データのローを複数返すことのできる select 文です。select の構文の詳細については、『ASE リファレンス・マニュアル』を参照してください。ただし、into 句や compute 句を使うことはできません。

例

```
EXEC SQL BEGIN DECLARE SECTION;
char    username[30];
char    password[30];
char    a_type[TITLE_STRING+1];
EXEC SQL END DECLARE SECTION;

.....

/*
** Declare an insensitive scrollable cursor against the
** titles table.Open the cursor.
*/
EXEC SQL DECLARE typelist INSENSITIVE SCROLL CURSOR FOR
SELECT DISTINCT title FROM titles;

EXEC SQL OPEN typelist;
```

使用法

- *cursor_name* はユニークでなければなりません。長さは最大 255 文字です。
- *cursor_name* の最初の文字は、英字または記号 “#” または “_” でなければなりません。
- *cursor sensitivity* を semi_sensitive として宣言すると、スクロール可能であることが暗黙的に設定されます。カーソルは、semi_sensitive (半反映型) のスクロール可能な読み取り専用カーソルになります。

- *cursor sensitivity* を insensitive として宣言すると、カーソルは非反映型になります。スクロールの可能性は、declare 部分に SCROLL を指定することにより決定されます。SCROLL を省略するか、NOSCROLL を指定すると、カーソルには insensitive だけが設定され、非スクロール可能カーソルになります。このカーソルは読み込み専用でもあります。
- cursor sensitivity を指定しない場合、カーソルは非スクロール可能な読み込み専用カーソルになります。
- declare 文で *cursor scrollability* が scroll であると指定し、cursor sensitivity を指定しない場合は、カーソルは非反映型でスクロール可能になります。このカーソルは読み込み専用でもあります。
- *cursor scrollability* で SCROLL オプションを省略するか、NOSCROLL を指定した場合は、読み込み専用の非スクロール可能カーソルになります。
- cursor scrollability を指定しない場合、カーソルは非スクロール可能な読み込み専用カーソルになります。

参照　fetch (スクロール可能カーソル)

delete (カーソル位置)

説明　オープンしているカーソルの現在のカーソル位置によって示されたローをテーブルから削除します。

構文

```
exec sql [at connection_name] delete
 [from] table_name
 where current of cursor_name;
```

パラメータ

table_name

ローが削除されるテーブルの名前です。

where current of *cursor_name*

この句を指定すると、カーソル *cursor_name* の現在のカーソル位置によって示されたテーブルのローを Adaptive Server Enterprise が削除します。

例

```
exec sql include sqlca;
main()
{
     char answer[1];
exec sql begin declare section;
     CS_CHAR disc_type[40];
     CS_CHAR store_id[5];
     CS_SMALLINT ind_store_id;
exec sql end declare section;
exec sql connect "sa";
exec sql use pubs2;
exec sql declare purge_cursor cursor for
     select discounttype, stor_id
```

```
     from discounts;
exec sql open purge_cursor;
exec sql whenever not found goto alldone;
while(1)
     {
     exec sql fetch purge_cursor into :disc_type, :store_id
            :ind_store_id;
     if (ind_store_id != -1)
          {
          printf("%s, %s¥n", disc_type, store_id);
          printf("Delete Discount Record?(y/n)  >");
          gets(answer);
          if (strncmp(answer, "y", 1) == 0)
               {
               exec sql delete from discounts where
                    current of purge_cursor;
               }
          }
     }
/*
 **  No changes will be committed to the database because
 **  this program does not contain an "exec sql commit work;"
 **  statement.The changes will be rolled back when the
 **  user disconnects.
 */
alldone:
     exec sql close purge_cursor;
     exec sql disconnect all;
}
```

使用法

- このリファレンス・ページでは、主に Transact-SQL の delete 文を Embedded SQL で使用した場合の相違点について説明します。『ASE リファレンス・マニュアル』を参照してください。
- このフォーマットの delete 文は、そのカーソルの *cursor_name* をオープンした接続で実行してください。delete 文内で at *connection_name* 句を指定する場合、その句は *cursor_name* をオープンした open cursor 文の at *connection_name* 句と一致していなければなりません。
- カーソルを for read only で宣言したり、select 文内に order by 句を指定したりした場合、delete 文は失敗します。

参照

close、declare cursor、fetch、open、update

delete (検索条件)

説明

検索条件によって指定されたローを削除します。

構文

exec sql [at *connection_name*] delete *table_name_1*
 [from *table_name_n*
 [, *table_name_n*]…]
 [where *search_conditions*];

パラメータ

table_name_1

この delete 文によってローが削除されるテーブルの名前です。

from *table_name_n*

table_name_1 のどのローが削除されるかを決定するために、*table_name_1* とジョインされるテーブルの名前です。delete 文は、*table_name_n* からはローを削除しません。

where *search_conditions*

どのローが削除されるかを指定します。where 句を省略した場合、delete 文は *table_name_1* のすべてのローを削除します。

例

```
/*
 ** Function to FAKE a cascade delete of an author **
 **by name -- this function assumes that pubs2 is
 ** the current database.
 ** Returns 1 for success, 0 for failure
 **/
     int        drop_author(fname, lname)
     char       *fname;
     char       *lname;
 {
 exec sql begin declare section;
     CS_CHAR        f_name[41], l_name[41];
     CS_CHAR        titleid[10], auid[10];
 exec sql end declare section;
     long        SQLCODE;
 strcpy(f_name, fname);
 strcpy(l_name, lname);
 exec sql whenever sqlerror goto roll_back;
 exec sql select au_id from authors into :auid
         where au_fname = :f_name
         and au_lname = :l_name;
exec sql delete from au_pix where au_id = :auid;
exec sql delete from blurbs where au_id = :auid;
 exec sql declare cur1 cursor for
     select title_id from titleauthor
     where au_id = :auid;
 exec sql open cur1;
 while (SQLCODE == 0)
 {
     exec sql fetch cur1 into :titleid;
    if(SQLCODE == 100) break;
    exec sql delete from salesdetail
```

```
            where title_id = :titleid;
        exec sql delete from rowsched
            where title_id = :titleid;
        exec sql delete from titles
            where title_id = :titleid;
        exec sql delete from titleauthor
            where current of cur1;
    }
    exec sql close cur1;
    exec sql delete from authors
        where au_id = :auid;
    exec sql commit work;
    return 1;

    roll_back:
        exec sql rollback work;
        return 0;
    }
```

使用法

- このリファレンス・ページでは、主に Transact-SQL の delete 文を Embedded SQL で使用した場合の相違点について説明します。『ASE リファレンス・マニュアル』を参照してください。
- カーソル・ポインタの現在の位置によって指定されるローを削除する必要がある場合は、delete (**カーソル位置**) 文を使用してください。

参照

close、declare cursor、fetch、open、update

describe input (SQL 記述子)

説明

準備された動的 SQL 文の動的パラメータ・マーカについての情報を取得し、その情報を SQL 記述子に保管します。

使用可能な SQL 記述子のデータ型のコードのリストについては、表 10-5 (167 ページ) を参照してください。

構文

exec sql describe input *statement_name*
using sql descriptor *descriptor_name*;

パラメータ

statement_name

情報を得るための準備文で指定された文の名前です。*statement_name* は、準備文を示す必要があります。

sql descriptor

descriptor_name を SQL 記述子として指定します。

descriptor_name

準備文内の動的パラメータ・マーカについての情報を格納できる SQL 記述子の名前です。

例

```
exec sql begin declare section;
char           query[maxstmt];
int            nin, nout, i;
exec sql end declare section;
int    j;
                  ...
exec sql allocate descriptor din with max 256;
exec sql allocate descriptor dout with max 256;
exec sql whenever sqlerror stop;
exec sql prepare dynstmt from :query;
exec sql describe input dynstmt
    using sql descriptor din;
exec sql get descriptor din :nin = count;
for (i = 0; i < nin; i++)
```

使用法

- 文についての情報は、using 句で指定された記述子に書き込まれます。describe input 文の実行後に get descriptor 文を使用して、記述子からホスト変数に情報を抽出してください。
- 記述子を割り付けてからでないと、describe input 文を実行できません。

参照

allocate descriptor、deallocate descriptor、describe output、get descriptor、prepare、set descriptor

describe input (SQLDA)

説明

準備された動的 SQL 文の動的パラメータ・マーカについての情報を取得し、その情報を SQLDA 構造体に保管します。

構文

exec sql describe input *statement_name*
 using descriptor *descriptor_name*;

パラメータ

statement_name

情報を得るための準備文で指定された文の名前です。*statement_name* は、準備文を示す必要があります。

descriptor

descriptor_name を SQLDA 構造体として指定します。

descriptor_name

準備文内の動的パラメータ・マーカについての情報を格納できる SQLDA 構造体の名前です。

例

```
...
exec sql prepare s4 from :str4;
exec sql declare c2 cursor for s4;
exec sql describe input s4 using descriptor dinout;
printf("Number of input parameters is %hd¥n",
     dinout.sd.sqld);
```

使用法

- 文についての情報は、using 句で指定された記述子に書き込まれます。get descriptor 文が実行されたあとに、SQLDA 構造体からの情報の読み込みができるようになります。

参照

allocate descriptor、deallocate descriptor、describe output、get descriptor、prepare、set descriptor

describe output (SQL 記述子)

説明

準備された動的 SQL 文の結果セットについてのロー・フォーマット情報を取得します。

使用可能な SQL 記述子のデータ型のコードのリストについては、表 10-5 (167 ページ) を参照してください。

構文

exec sql describe [output] *statement_name*
using sql descriptor *descriptor_name*;

パラメータ

output

describe output 文では有効でないオプションのキーワードですが、SQL 標準に準拠しています。

statement_name

実行する select 文を表す (前の prepare 文内で指定された) 名前です。

sql descriptor

descriptor_name を SQL 記述子として指定します。

descriptor_name

describe output 文によって返された情報を保管する SQL 記述子の名前です。

例

```
...
exec sql open curs2 using sql descriptor descr_out;
exec sql describe output prep_stmt4
     using sql descriptor descr_out;
while (sqlca.sqlcode != 100 && sqlca.sqlcode >= 0)
 {
         exec sql fetch curs2 into sql descriptor
             descr_out;
         print_descriptor();
 }
 exec sql close curs2;
 exec sql deallocate descriptor descr_out;
 exec sql deallocate prepare prep_stmt4;
 printf("dynamic sql method 4 completed¥n¥n");
 }
         ...
```

使用法

- 得られる情報はタイプ、名前、長さ (数値であれば、精度と位取り)、null が許されるかどうかのステータス、結果セットの項目数です。
- この情報は、select カラム・リストからの結果カラムです。
- describe output 文を実行してから、準備文を実行してください。準備文の実行後に、describe output を実行してから、get descriptor を実行すると、結果は廃棄されます。

参照

allocate descriptor、describe input、execute、get descriptor、prepare

describe output (SQLDA)

説明

準備された動的 SQL 文の結果セットについてのロー・フォーマット情報を取得し、その情報を SQLDA 構造体に保管します。

構文

exec sql describe [output] *statement_name*
 using descriptor *sqlda_name*;

パラメータ

output
: describe output 文では有効でないオプションのキーワードですが、SQL 標準に準拠しています。

statement_name
: 実行する select 文を表す (前の prepare 文内で指定された) 名前です。

descriptor
: *sqlda_name* を SQLDA 構造体として指定します。

sqlda_name
: describe output 文によって返された情報を保管する SQLDA 構造体の名前です。

例

```
...
exec sql open curs2 using descriptor input_descriptor;
exec sql describe output statement using descriptor
     output_descriptor;
output_descriptor->sqlda_column->sqlda_sqldata = character;
output_descriptor->sqlda_column->sqlda_datafmt.datatype =     CS_CHAR_TYPE;
output_descriptor->sqlda_column->sqlda_datafmt.maxlength = 20;
output_descriptor->sqlda_column->sqlda_sqllen = 20;
output_descriptor->sqlda_column->sqlda_datafmt.format =
     (CS_FMT_NULLTERM | CS_FMT_PADBLANK);
exec sql fetch curs2 into descriptor output_descriptor;
```

使用法

- 得られる情報はタイプ、名前、長さ (数値であれば、精度と位取り)、null が許されるかどうかのステータス、結果セットの項目数など、SQLDA フィールド内に入るデータです。
- この情報は、select カラム・リストからの結果カラムです。

参照 describe input、execute、prepare

disconnect

説明 1 つ以上の Adaptive Server Enterprise への接続をクローズします。

構文

exec sql disconnect
{*connection_name* | current | DEFAULT | all};

パラメータ

connection_name
クローズする接続の名前です。

current
現在の接続をクローズすることを指定します。

DEFAULT
デフォルトの接続をクローズすることを指定します。次の例のような文字列変数を使用してデフォルトの *connection_name* を指定する場合は、このキーワードを大文字にしてください。

```
exec sql disconnect :hv;
```

all
アクティブなすべての接続をクローズすることを指定します。

例

```
#include <stdio.h>

 exec sql include sqlca;

 main()
 {
         exec sql begin declare section;
         CS_CHAR servname[31], username[31],
         password[31], conname[129];
         exec sql end declare section;

         exec sql whenever sqlerror call error_handler();
         exec sql whenever sqlwarning call error_handler();
         exec sql whenever not found continue;

         printf ("Username: ");
         gets   (username);
         printf ("Password: ");
```

```
        gets   (password);
        printf ("Adaptive Server Enterprise name: ");
        gets   (servname);
        printf ("Connection name: ");
        gets   (conname);

/*
** Make a named connection.
*/
        exec sql connect :username identified by :password
                at :conname using :servname;
/*
** Make an unnamed (default) connection.
*/
        exec sql connect :username identified by :password
                using :servname;
/*
** The second (default) connection is the current connection.
*/
        exec sql disconnect current;
/*
** We now have neither a default connection nor a current one.
*/
        exec sql disconnect :conname;
/*
** Now there are no open connections.
*/
        exec sql exit;
}

error_handler()
{
        printf("%d¥n%s¥n",sqlca.sqlcode,sqlca.sqlerrm.sqlerrmc);
        exit(0);
}
```

使用法

- disconnect キーワードは、単独では有効な文ではありません。したがって、このキーワードのあとに *connection_name*、current、DEFAULT、または all を指定してください。
- 接続をクローズすると、その接続に対応するすべてのメモリとリソースが解放されます。
- disconnect キーワードは現在のトランザクションを commit しないで、そのトランザクションをロールバックします。その接続で非連鎖トランザクションがアクティブである場合、disconnect キーワードはどのようなセーブポイントも無視して、そのトランザクションをロールバックします。

- 接続をクローズすると、オープンしているカーソルがクローズされて、テンポラリの Adaptive Server Enterprise オブジェクトは削除されます。また、Adaptive Server Enterprise でその接続が持っているすべてのロックが解放されて、Adaptive Server Enterprise へのネットワーク接続はクローズされます。

参照

commit work、commit transaction、connect、rollback transaction、rollback work

exec

説明

システム・プロシージャまたはユーザ定義ストアド・プロシージャを実行します。

構文

```
exec sql [at connection_name]
 exec [[:status_var =]status_value] procedure_name
[([[@parameter_name =]param_value [out[put]]],...)]
 [into :hostvar_1 [:indicator_1]
 [, hostvar_n [indicator_n,…]]]
 [with recompile];
```

注意 exec 文を Embedded SQL の execute 文と混同しないでください。これらの文は関係がありません。ただし、Embedded SQL の exec 文は Transact-SQL の execute 文と同じです。

パラメータ

status_var
ストアド・プロシージャのリターン・ステータスを受信するホスト変数です。

status_value
ストアド・プロシージャのリターン・ステータス変数 *status_var* の値です。

procedure_name
実行するストアド・プロシージャの名前です。

parameter_name
ストアド・プロシージャのパラメータの名前です。

param_value
ホスト変数またはリテラル値です。

output
ストアド・プロシージャがパラメータ値を返すことを示します。ストアド・プロシージャ内の対応するパラメータも、output キーワードを使用して作成されている必要があります。

into *:hostvar_1*
このキーワードを指定すると、ストアド・プロシージャから返されるローはここで指定したホスト変数 (*hostvar_1* ～ *hostvar_n*) に保管されます。それぞれのホスト変数にはインジケータ変数を指定できます。

with recompile

このキーワードを指定すると、プロシージャが実行されるたびに Adaptive Server Enterprise はこのストアド・プロシージャに対する新しいクエリ・プランを作成します。

例

例 *1*

```
exec sql begin declare section;
     char       titleid[10];
     int        total_discounts;
     short      retstat;
 exec sql end declare section exec;
 exec sql create procedure get_sum_discounts
     (@titleid tid, @discount int output) as
 begin
         select @discount = sum( qty * discount)
         from salesdetail
         where title_id = @titleid

 end;
 printf("title id: ");
 gets(titleid);

    exec sql exec
         :retstat = get_sum_discount :titleid,
         :total_discounts out;

printf("total discounts for title_id %s were
         %s¥n", titleid, total_discounts);
exec sql begin declare section;
     CS_INT            status;
     CS_CHAR           city(30);
     CS_INT            result;
 exec sql end declare section;
 LONG                  SQLCODE;

 input "City", city ;
  exec sql exec countcity :city, :result out;
 if (SQLCODE = 0)
         print city + " occurs " + result + "
             times." ;
```

例 *2*

```
EXEC SQL BEGIN DECLARE SECTION;
       /* storage for login name and password */
       CS_CHAR         username[30], password[30];
       CS_CHAR         pub_id[4][5], pub_name[4][40], stmt[100] ;
       CS_CHAR         city[4][15], state[4][3];
       CS_INT          ret_status;
EXEC SQL END DECLARE SECTION ;
        ...
EXEC SQL set chained off;
strcpy(stmt,"create proc get_publishers as select * from publishers
```

```
 return ");
EXEC SQL EXECUTE IMMEDIATE :stmt;

EXEC SQL EXEC :ret_status = get_publishers INTO
                            :pub_id,
                            :pub_name,
                            :city,
                            :state;
printf("Pub Id    Publisher Name         City       State ¥n");
 printf("¥n----- ----------------------  ------------ -----¥n");
for ( i = 0 ; i < sqlca.sqlerrd[2] ; i++ )
 {
         printf("%-8s", pub_id[i])   ;
         printf("%-25s", pub_name[i]) ;
         printf("%-12s", city[i]) ;
         printf("%-6s¥n", state[i]) ;
}
printf("¥n(%d rows affected, return status = %d)¥n", sqlca.sqlerrd[2],
 ret_status);
        ...
}
```

使用法

- クライアント・アプリケーションにローを返すことができるのは、1 つの select 文だけです。
- ロー・データを返すことができる select 文がストアド・プロシージャに含まれている場合は、次の 2 つの方法のどちらかを使用してそのデータを保管してください。つまり、exec 文の into 句を使用する方法とそのプロシージャのカーソルを宣言する方法のどちらかです。into 句を使用した場合、ユーザが指定したホスト変数が配列でないかぎり、ストアド・プロシージャは複数のデータ・ローを返しません。
- 値 *param_value* にはホスト変数またはリテラル値を指定できます。ただし output キーワードを使用する場合は、*param_value* をホスト変数にしてください。
- *parameter_name* に output キーワードを指定できるのは、*procedure_name* を作成した create procedure 文の対応するパラメータにもこのキーワードを使用した場合にかぎります。
- Embedded SQL の exec 文の機能は、Transact-SQL execute 文とほとんど同じです。

参照

declare cursor (ストアド・プロシージャ)、select

exec sql

説明

ホスト言語プログラムに埋め込まれた SQL 文の始めにマークを付けます。

構文

exec sql [at *connection_name*] *sql_statement*;

パラメータ

at

SQL 文 *sql_statement* を Adaptive Server 接続 *connection_name* で実行することを指定します。

connection_name

sql_statement が実行される Adaptive Server 接続を識別する接続名です。*connection_name* は前の connect 文内で定義してください。

sql_statement

Transact-SQL 文または Embedded SQL 文です。

例

```
exec sql
 begin declare section;
     char     site1(20);
     int      sales1;
 exec sql end declare section;

 exec sql connect "user1" identified by "password1"
     using "server1";
 exec sql connect "user2" identified by "password2"
     using "server2"
 /* Remember that a connection that has not been
     explicitly named has the name of its server */
 exec sql at server1 select count(*) from sales
     into :sales1;

 site1 = sitename("server1");
 exec sql at server2 insert into numsales
     values(:site1, :sales1);
```

使用法

- ホスト言語に埋め込まれた SQL 文は exec sql で始めてください。キーワード exec sql は、ホスト言語の文を開始できる場所であればどこにでも指定できます。
- 文 *sql_statement* を 1 つまたは複数のプログラム行に渡って指定できます。ただし、改行と継続行についてのホスト言語の規則に準拠する必要があります。
- at 句の影響が及ぶのは文 *sql_statement* だけです。この句は後続の SQL 文に影響を与えることも、現在の接続をリセットすることもありません。
- *sql_statement* が次に示す SQL 文のうちのどれかである場合は、at 句は有効ではありません。

表 *10-2: exec sql の at* 句を使用できない文

allocate descriptor	begin declare section	connect
deallocate descriptor	declare cursor (動的)	end declare section
exit	get diagnostics	include file
include sqlca	set connection	set diagnostics
whenever		

- *connection_name* は前の connect 文内で定義してください。
- ターミネータを使用してそれぞれの Embedded SQL 文を終了させてください。C 言語の場合、ターミネータはセミコロン (;) です。

参照

begin declare section、connect、disconnect、set connection

execute

説明

準備文から動的 SQL 文を実行します。

execute immediate 文の詳細については、「execute immediate」(142 ページ) を参照してください。

構文

```
exec sql [at connection_name] execute statement_name
 [into {host_var_list |
    descriptor descriptor_name |
    sql descriptor descriptor_name}]
 [using {host_var_list |
     descriptor descriptor_name |
     sql descriptor descriptor_name}];
```

注意 Embedded SQL の execute 文を Embedded SQL の exec 文または Transact-SQL の execute 文と混同しないでください。

パラメータ

statement_name

前の prepare 文内で定義された文に対するユニークな識別子です。

descriptor_name

文の動的パラメータ・マーカ、または select カラム・リストを記述するメモリ領域または SQLDA 構造体を指定します。

into

文が select 文 (単一ローの選択) を実行するときに必要な句です。into 句の対象は SQL 記述子、SQLDA 構造体、あるいは 1 つまたは複数の Embedded SQL ホスト変数のリストです。

host_var_list 中の各ホスト変数は、declare セクションの中で最初に定義しなければなりません。null データ値が取得された場合にそのことを示すために、インジケータ変数をホスト変数に対応付けることができます。

descriptor

descriptor_name を SQLDA 構造体として指定します。

sql descriptor

descriptor_name を SQL 記述子として指定します。

using

host_var_list 内の動的パラメータ・マーカと置き換えられるホスト変数です。declare セクション内で定義しなければならないホスト変数は、リストされた順に置き換えられます。この句は、*statement_name* に動的パラメータ・マーカが含まれているときだけ使います。また、動的パラメータ・マーカの値を動的記述子に入れることもできます。

例

```
exec sql begin declare section;
     CS_CHAR        dymo_buf(128);
     CS_CHAR        title_id(6);
     CS_INT         qty;
     CS_CHAR        order_no(20);
 exec sql end declare section;

 dymo_buf = "INSERT salesdetail
     (ord_num, title_id, qty) VALUES (:?, :?, :?)"

 exec sql prepare ins_com from :dymo_buf;

 print "Recording Book Sales";
 input "Order number?", order_no;
input "Title ID?", title_id;
input "Quantity sold?", qty;
exec sql execute ins_com
     using :order_no, :title_id, :qty;
exec sql disconnect;
```

使用法

- execute は動的 SQL のメソッド 2 の 2 番目の手順です。最初の手順は prepare 文です。
- prepare および execute は、複数ローの select 以外の SQL 文に有効です。複数ローの select の場合には、動的カーソルを使用してください。
- *statement_name* 内の文に動的パラメータ・マーカ（“?”）を記述できます。マーカは、文が実行する前にホスト変数値が置き換えられる位置を示します。
- execute キーワードは、この文と exec 文を区別します。「exec」(136 ページ) を参照してください。

参照

declare section、get descriptor、prepare、set descriptor

execute immediate

説明

文字列ホスト変数または引用符で囲まれた文字列に格納された動的 SQL 文を実行します。

構文

exec sql [at *connection_name*] execute immediate
{:*host_variable* | “string”};

パラメータ

host_variable

declare セクション内で定義された文字列ホスト変数です。execute immediate を呼び出す前に、ホスト変数は、文法的に正しい完全な Transact-SQL 文に設定されていなければなりません。

string

host_variable の代わりに使われる、引用符で囲まれたリテラル Transact-SQL 文の文字列です。

例

```
exec sql begin declare section;
     CS_CHAR         host_var(128);
 exec sql end declare section;

 printf("Enter a non-select SQL statement: ");
 gets(host_var);

 exec sql execute immediate :host_var;
```

使用法

- execute immediate 文を使用することは、動的 SQL のメソッド 1 を使用することを意味します。動的 SQL のメソッドの詳細については、「第 7 章 動的 SQL の使い方」を参照してください。
- *host_variable* 内の文は、メッセージを除いてプログラムに結果を返すことができません。したがって、その文には select 文などは指定できません。
- Embedded SQL プリコンパイラは、*host_variable* 内に保管された文の構文をチェックしないで Adaptive Server Enterprise に送信します。その文の構文が正しくない場合は、Adaptive Server Enterprise がエラー・コードとエラー・メッセージをプログラムに返します。
- ホスト変数から動的 SQL 文に値を置き換えるには、prepare 文と execute 文を使用してください (動的 SQL のメソッド 2)。
- 結果を返す動的 SQL 文とともに select 文を実行するには、prepare 文、open 文、fetch 文を使用してください (動的 SQL のメソッド 3)。

参照

execute、prepare

exit

説明

Client-Library をクローズして、プログラムに割り付けられたすべての Embedded SQL リソースの割り付けを解除します。

構文

exec sql exit;

例

```
exec sql include sqlca;
 main()
 {
 /* The body of the main function goes here,
 ** including various Embedded SQL statements.
 */
 ...
 /* The exit statement must be the last
 ** embedded SQL statement in the program.
 */
 exec sql exit;
 }     /* end of main */
```

使用法

- exit 文は、プログラムがオープンしたすべての接続をクローズします。また exit 文は、プログラムに割り付けられたすべての Embedded SQL リソースと Client-Library リソースの割り付けも解除します。
- exit 文はすべてのプラットフォームで有効ですが、必須なのは一部のプラットフォームのみです。『Open Client/Server プログラマーズ・ガイド補足』を参照してください。
- exit 文の使用後は、Client-Library をもう一度初期化しないと Client-Library 機能を使用できません。Client-Library の初期化の詳細については、『Open Client Client-Library/C プログラマーズ・ガイド』を参照してください。
- exit 文は Sybase の拡張機能であり、SQL 標準では定義されていません。

参照

disconnect

fetch

説明

現在のカーソル・ローからホスト変数または動的記述子にデータ値をコピーします。

構文

exec sql [at *connection_name*] fetch [rebind | norebind] *cursor_name*
into {:*host_variable* [[indicator]:*indicator_variable*]
 [,:*host_variable*
 [[indicator]:*indicator_variable*]]… |
 descriptor *descriptor_name* |
 sql descriptor *descriptor_name*};

パラメータ

rebind | norebind
ホスト変数がこの fetch 文に対して再バインドを要求するかどうかを指定します。rebind 句は、再バインドを制御するプリコンパイラ・オプションを無効にします。

cursor_name
カーソルの名前です。この名前は前の declare cursor 文内で定義します。

host_variable
declare セクション内で定義されたホスト言語変数です。

indicator_variable
前の declare セクションで宣言された 2 バイトのホスト変数です。対応する変数の値が null の場合、fetch 文はインジケータ変数を -1 に設定します。トランケーションが行われる場合、fetch 文はインジケータ変数を結果カラムの実際の長さに設定します。トランケーションが行われない場合は、fetch 文はインジケータ変数を *0* に設定します。

descriptor
descriptor_name を SQLDA 構造体として指定します。

sql descriptor
descriptor_name を SQL 記述子として指定します。

descriptor_name
結果セットを保持する動的記述子の名前です。

例

```
exec sql begin declare section;
     CS_CHAR        title_id[6];
     CS_CHAR        title[80];
     CS_CHAR        type[12];
     CS_SMALLINT    i_title;
     CS_SMALLINT    i_type;
 exec sql end declare section;
 exec sql declare title_list cursor for
     select type, title_id, title from titles
     order by type;

 exec sql open title_list
 while (sqlca.sqlcode != 100) {
exec sql fetch title_list into
         :type :i_type, :title_id, :title :i_title;

         if (i_type != -1) {
             printf("Type:%s¥n", type);
         }
         else {
             printf("Type:undecided¥n");
         }

         printf("Title id:%s¥n", title_id);
```

```
            if (i_title <> -1) {
                print "Title:", title;
            }
            else {
                print "Title:undecided";
            }
    }

    exec sql close title_list;
```

使用法

- fetch 文は、動的 SQL 内のカーソルまたは静的カーソルのいずれにも使用できます。
- open 文を実行してから、fetch 文を実行してください。
- オープン・カーソルでの最初の fetch は、カーソルの結果テーブルから最初のローまたはローのグループを返します。後続の fetch は、それぞれ次のローまたはローのグループを返します。
- 複数のローを配列にフェッチできます。
- 「現在のロー」は最後にフェッチされたローです。ローを更新または削除するには、update 文または delete 文に where current of *cursor_name* 句を使用します。これらの文は、ローがフェッチされるまで無効です。
- カーソルからすべてのローがフェッチされたあと、fetch の呼び出しは SQLCODE を 100 に設定します。select の実行によって結果が生成されない場合、最初のフェッチ時に SQLCODE が 100 に設定されます。
- 結果セットの各カラムには *host_variable* が 1 つだけ必要です。
- rebind も norebind も指定しない場合、バインド動作はプリコンパイラ・オプション -b によって決定されます。継続バインドの詳細については、「継続バインドを使用するためのガイドライン」(104 ページ) を参照してください。プリコンパイラ・オプションの詳細については、使用しているプラットフォームの『Open Client/Server プログラマーズ・ガイド補足』を参照してください。
- *indicator_variable* が null 値を受信できるように *host_variable* を指定する必要があります。インジケータ変数のないホスト変数が null 値をフェッチする場合、実行時エラーが発生します。
- Client-Library は、可能な場合に、結果カラムのデータ型を対応するホスト変数のデータ型に変換します。Client-Library がデータ型を変換できない場合には、エラー・メッセージが表示されます。変換ができない場合にはエラーが発生します。

参照

allocate descriptor、close、declare、delete (**カーソル位置**)、open、prepare、update

fetch (スクロール可能カーソル)

説明

fetch 文を使用して、カーソルによってデータを取得し、そのデータをホスト変数に割り当てます。

構文

exec sql [at *connect_name*] fetch [*fetch orientation*]*cursor_name* into :*host_variable* [[*indicator*]:*indicator_variable*] [,:*host_variable* [[*indicator*]:*indicator_variable*]...];

パラメータ

host_variable

結果ローの各カラムに 1 つずつ *host_variable* が必要です。

fetch orientation

カーソルがスクロール可能である場合に、ローのフェッチ方向を指定します。

例

```
/*
** Fetch the first row in cursor resultset
*/
EXEC SQL FETCH FIRST FROM typelist INTO :a_type;
printf("¥n%s¥n", a_type);

/*
** Fetch the last row in cursor resultset
*/
EXEC SQL FETCH LAST FROM typelist INTO :a_type;
printf("¥n%s¥n", a_type);
```

使用法

- *host_variable* を使用するときは、各ホスト変数の前にコロンを付け、次のホスト変数との間をカンマで区切ります。fetch 文にリストされたホスト変数は、select 文が取得する Adaptive Server Enterprise の値と一致している必要があります。つまり、変数の数は戻り値の数と同じで、同じ順序で並び、互換性のあるデータ型でなければなりません。
- *fetch orientation* のオプションは、NEXT、PRIOR、FIRST、LAST、ABSOLUTE *fetch_offset*、RELATIVE *fetch_offset* です。*fetch orientation* を指定しない場合のデフォルトは next です。*fetch orientation* を指定する場合は、カーソルがスクロール可能でなければなりません。fetch 文が取得するデータは、カーソルの位置に依存します。
  fetch 文は、通常、カーソルをオープンするときに指定された ROW_COUNT に応じて、カーソル結果セットから単一または複数のローを取得します。カーソルがスクロール可能でない場合、fetch は、結果セットの次のローを取得します。カーソルがスクロール可能な場合は、fetch 文に含まれるコマンドで指定された位置のローを取得します。

参照

declare scrollable cursor

get descriptor

説明

動的パラメータ・マーカおよび select カラム・リスト属性の属性情報、および SQL 記述子からのデータを取得します。

SQL 記述子のデータ型コードのリストについては、表 10-5 (167 ページ) を参照してください。

構文

exec sql get descriptor *descriptor_name*
{:*host_variable* = count |
value *item_number* :*host_variable* = *item_name*
[, :*host_variable* = *item_name*]…};

パラメータ

descriptor_name
準備文内の動的パラメータ・マーカまたは戻りカラムについての情報が入っている SQL 記述子の名前です。

host_variable
declare セクション内で定義された変数です。

count
取得される動的パラメータの数です。

item_number
get descriptor 文で *n* 番目の動的パラメータ・マーカまたは select カラムの情報を取得することを指定する数字です。

item_name
取得される属性の名前です。詳細については、表 10-3 を参照してください。

表 *10-3:* 有効な *item_name* 値

値	説明
data	指定した SQL 記述子と対応する動的パラメータ・マーカまたはターゲットの値。インジケータが負の値である場合、このフィールドは未定義である。
indicator	動的パラメータ・マーカまたはターゲットと対応するインジケータ・パラメータの値。
length	指定した SQL 記述子のターゲットの動的パラメータ・マーカの文字単位での長さ。
name	動的パラメータ・マーカについての情報が入っている、指定した SQL 記述子の名前。
nullable	動的パラメータ・マーカで null 値が許可される場合は 0 であり、null 値が許可されない場合は 1 である。
precision	CS_NUMERIC 変数の精度の有効桁数を指定する整数。
returned_length	select カラム・リストからの character データ型の値の長さ。
scale	CS_NUMERIC 変数の小数点桁数を指定する整数。
type	ロー内のこのカラム (項目番号) のデータ型。値については、「SQL 記述子のデータ型のコード」を参照。

例

```
exec sql begin declare section;
     int     numcols, colnum, type, intbuf;
     char    charbuf[100];
 exec sql end declare section;
 ...
 exec sql allocate descriptor big_desc
     with max 1000;
 exec sql prepare dynstmt from "select * from ¥
     huge_table";
 exec sql execute dynstmt into sql descriptor
     big_desc;
exec sql get descriptor big_desc :numcols = count;
for (colnum = 1; colnum <= numcols; colnum++)
 {
exec sql get descriptor big_desc
 value :colnum :type = type;
if (type == 4)
{
exec sql get descriptor big_desc
 value :colnum :intbuf = data;
/* Display intbuf. */
...
}
else if (type == 1)
{
big_desc
 value :colnum :charbuf = data;
/* Display charbuf. */
...
}
 }
 exec sql deallocate descriptor big_desc;
         ...
```

使用法

- get descriptor 文は、指定した動的パラメータの数または属性についての情報、または準備文で指定された文の select リスト・カラムについての情報を返します。
- get descriptor 文は、describe input 文、describe output 文、execute 文、または fetch (動的) 文を実行したあとに実行してください。
- execute 文または fetch 文を実行してその記述子と対応するデータをサーバから取得するまでは、*data*、indicator、または *returned_length* を取得できません。

参照

describe input、describe output、fetch、set descriptor

get diagnostics

説明

Client-Library からエラー、警告、情報メッセージを取得します。

構文

get diagnostics
{:*hv* = *statement_info* [, :*hv* = *statement_info*]…|
exception :*condition_number*
:*hv* = *condition_info* [, :*hv* = *condition_info*]…}

パラメータ

statement_info

キーワード number は、現在サポートされている唯一の *statement_info* タイプです。このキーワードは例外の数の合計を診断キューに返します。

condition_info

sqlca_info、*sqlcode_number*、*returned_sqlstate* のいずれかのキーワードを指定します。

例

```
exec sql begin declare section;
     CS_INT   num_msgs;
     CS_INT   condcnt=1;
     exec sql include sqlca;
 exec sql end declare section;
 exec sql exec sp_password "bass", "foo";
 exec sql get diagnostics :num_msgs = number;

printf("Number of messages is %d.¥n", num_msgs);

 /* Loop through and print the messages. */

 while (condcnt <= num_msgs)
 {
     exec sql get diagnostics exception :condcnt
          :sqlca = sqlca_info;
    printf("SQLCODE = %d  ¥n", sqlca.sqlcode);
     printf("%s ¥n", sqlca.sqlerrm.sqlerrmc);
     condcnt = condcnt + 1;
 }
```

使用法

- 多くの Embedded SQL 文は複数の警告やエラーを発生させる傾向があります。通常、最初のエラーだけが、SQLCODE、SQLCA、または SQLSTATE によって報告されます。すべてのエラーを処理するには get diagnostics を使用してください。
- get diagnostics は、whenever 文の call 句、perform 句、または go to 句のターゲットであり、コード内で使用できます。
- 情報メッセージを取得する文のあとで get diagnostics を使用できます。

参照

whenever

include "*filename*"

説明 Embedded SQL のソース・ファイルに外部ファイルをインクルードします。

構文 exec sql include "*filename*";

パラメータ *filename*
この文が指定されている Embedded SQL のソース・ファイルにインクルードされるファイルの名前です。

例

```
common.h:
    /* This file contains definitions and
     ** declarations used in the file getinfo.c.
     */

 #include <stdio.h>
 #include "./common.h"
 void    err_handler();
 void    warning_handler();
 exec sql include sqlca;
 {
    exec sql begin declare section;
              CS_CHAR username[33], password[33], date[33];
    exec sql end declare section;

    exec sql whenever sqlerror call err_handler();
    exec sql whenever sqlwarning call warning_handler();
    exec sql whenever not found continue;

 /*
 ** Copy the user name and password defined in common.h to
 ** the variables decalred for them in the declare section.
 */
 strcpy (username, USER);
 strcpy(password, PASSWORD);

 printf("Today's date:%s¥n", date);
 ...
 }
 void     err_handler()
 {
 ...
 }
 void     warning_handler()
 {
 ...
 }
 /* common.h */
 #define USER "sa"
 #define PASSWORD ""
```

```
============================================================

exec sql begin declare section;
        char    global_username[100];
        char    global_password[100];
exec sql end declare section;

                        getinfo.c

                                #include <common.h>
                                 printf("uid?¥n");
                                 gets(global_username);
                                 printf("password?¥n");
                                 gets(global_password);

                        do_connect.c
                                exec sql include "common.h";

                                 exec sql connect :global_username
                                   identified by :global_password;
```

使用法

- Embedded SQL プリコンパイラは、すべての declare セクションと SQL 文を認識して、インクルード・ファイルを Embedded SQL のソース・ファイルの一部のように処理します。Embedded SQL プリコンパイラは、この処理の結果生じるホスト言語のソース・コードを生成されたファイルに書き込みます。
- インクルード・パスのプリコンパイラ・コマンド・ライン・オプションを使用して、インクルード・ファイルを検索するディレクトリを指定してください。『Open Client/Server プログラマーズ・ガイド補足』を参照してください。
- インクルード・ファイルは最大 32 個の深さまでネストできます。
- include "*filename*" 文はどこにでも使用できます。

参照　declare section

include sqlca

説明　Embedded SQL プログラムで SQLCA (SQL 通信領域) を定義します。

構文　exec sql include sqlca;

例

```
exec sql include SQLCA;
...
 exec sql update t1 set c1 = 123 where c2 > 47;
 if (sqlca.sqlcode     ==     0)
 {
     printf("%d rows updated/n", sqlca.sqlerrd[2]);
```

```
}
else if (sqlca.sqlcode     ==     100)
{
    printf("No rows matched the query¥n");
} else {
    printf("An error occured¥n%s¥n",
    sqlca.sqlerrm.sqlerrmc);
}
```

使用法

- ホスト言語の宣言が可能な場所であればどこにでも include sqlca 文を使用できます。

参照 begin declare section

include sqlda

説明 Embedded SQL プログラムで SQLDA 構造体を定義します。

構文 exec sql include sqlda;

例

```
exec sql include sqlda;
...
SQLDA *input_descriptor, *output_descriptor;
 CS_SMALLINT small;
 CS_CHAR     character[20];
input_descriptor = (SQLDA *)malloc(SYB_SQLDA_SIZE(3));
 input_descriptor->sqlda_sqln = 3;
 output_descriptor = (SQLDA *)malloc(SYB_SQLDA_SIZE(3));
 output_descriptor->sqlda_sqln = 3;
```

使用法

- ホスト言語の宣言が可能な場所であればどこにでも include sqlda 文を使用できます。

initialize_application

説明 グローバルな CS_CONTEXT ハンドルに、アプリケーション名を設定するための呼び出しを生成します。プリコンパイル時に -x オプションを指定した場合は、cs_config(CS_SET, CS_EXTERNAL_CONFIG, CS_TRUE) プロパティも設定されます。

構文 exec sql initialize_application
[application_name “=” *application_name*];

例

```
exec sql include sqlca;
 main()
 {
 exec sql initialize_application
     application_name = :appname;
 /*
 ** The body of the main function goes here,
 ** including various Embedded SQL statements.
 */
 ...
 /* The init statement must be the first
 ** embedded SQL statement in the program.
 */
 exec sql exit;
 }     /* end of main */
```

使用法

- *application_name* にはリテラル文字列と、アプリケーションの名前が入っている文字変数のどちらかを指定できます。
- initialize_application 文がアプリケーションで実行される最初の Embedded SQL 文である場合に -x オプションを指定すると、ct_init は外部設定オプションを使用して CS_CONTEXT 構造体の Client-Library 部分を初期化します。
- initialize_application が最初の Embedded SQL 文でない場合は、ct_init が外部設定オプションを選択することはありません。
- initialize_application 文がアプリケーションで実行される最初の Embedded SQL 文であるかどうかに関係なく、-x オプションを指定すると、exec sql connect 文は外部設定データを使用します。-e オプションも指定した場合、Sybase では設定データへのキーとしてサーバ名が使用されます。-e オプションを指定しない場合、アプリケーション名 (または DEFAULT) が設定データへのキーとして使用されます。
- -x オプションとアプリケーション名を指定した場合は、次のようになります。
  - ct_init はアプリケーション名を使用して、外部設定ファイルのどのセクションを初期化に使用するかを決定します。
  - アプリケーション名は connect 文の一部として Adaptive Server Enterprise に渡されます。アプリケーション名は、*sysprocesses.program_name* テーブルに入力されます。
- -e を指定して -x を指定しない場合は、ct_init による初期化時に外部設定データが使用されますが、すべての接続で外部設定データへのキーとしてサーバ名が使用されます。コマンド・ライン・オプションの詳細については、『Open Client/Server プログラマーズ・ガイド補足』を参照してください。

参照

exit

open (動的カーソル)

説明

前に宣言されている動的カーソルをオープンします。

構文

exec sql [at *connection_name*] open *cursor_name*
[row_count = *size*] [using {*host_var_list* |
descriptor *descriptor_name* |
sql descriptor *descriptor_name*}];

パラメータ

cursor_name

declare cursor 文を使用して宣言されているカーソルに名前を付けます。

size

ネットワーク経由で 1 度に返されるローの数です。ホスト変数にフェッチされるローの数ではありません。*size* 引数にはリテラルまたは宣言されたホスト変数のどちらかを指定できます。

host_var_list

動的パラメータ・マーカの値が入っているホスト変数に名前を付けます。

descriptor

descriptor_name を SQLDA 構造体として指定します。

sql descriptor

descriptor_name を SQL 記述子として指定します。

descriptor_name

準備文で指定された文内の動的パラメータ・マーカについての情報が入っている動的記述子に名前を付けます。

例

```
exec sql begin declare section;
     CS_CHAR        dyna_buf[128];
     CS_CHAR        title_id[6];
     CS_CHAR        lastname[40];
     CS_CHAR        firstname[20];
     CS_CHAR        phone[12];
 exec sql end declare section;

 dyna_buf = "SELECT a.au_lname, a.au_fname, a.phone"
         + "FROM authors a, titleauthor t "
         + "WHERE a.au_id = t.au_id "
         + "AND t.title_id = ?";

 exec sql prepare dyna_comm from :dyna_buf;

 exec sql declare who_wrote cursor for dyna_comm;

 printf("List authors for what title? ");
 gets(title_id);
exec sql open who_wrote using :title_id;
while (TRUE){         exec sql fetch who_wrote into
             :lastname, :firstname, :phone;
          if (sqlcode == 100) break;
        printf("Last name is %s¥n",lastname,          "First
```

```
name is %s¥n", firstname,
          "Phone number is %s¥n", phone);
 }

 exec sql close who_wrote;
```

使用法

- open 文は、対応する declare cursor 文で指定された文を実行します。その後ユーザは fetch 文を使用して、準備文の結果を取得できます。
- オープンできるカーソルの数に制限はありません。
- using 句は、select 文内の動的パラメータ・マーカ (“?”) をホスト変数または動的記述子の内容に置き換えます。

参照

close、declare、fetch、prepare

open (静的カーソル)

説明

事前に宣言されている静的カーソルをオープンします。この文はストアド・プロシージャを含む任意の静的カーソルをオープンするのに使用できます。

構文

exec sql [at *connection_name*] open *cursor_name*
[row_count = *size*];

パラメータ

cursor_name
オープンするカーソルの名前です。

row_count
ネットワーク経由で一度に返されるローの数です。ホスト変数にフェッチされるローの数ではありません。

size
Adaptive Server Enterprise からクライアントに同時に移動されるローの数です。ローがアプリケーションによってフェッチされるまで、クライアントはこれらのローをバッファします。このパラメータを使用して、ネットワークの効率をチューニングできます。

例

```
exec sql begin declare section;
         char        b_titleid[tidsize+1];
         char        b_title[65];
         char        b_type[typesize+1];
 exec sql end declare section;
         long        sqlcode;
         char        response[10];
                  ...
 exec sql declare titlelist cursor for
     select title_id, substring(title,1,64)
     from titles where type like :b_type;
     strcpy(b_type, "business");
exec sql open titlelist;
```

```
    for (;;)
     exec sql fetch titlelist into :b_titleid,
          :b_title;
          if (sqlcode == 100)
              break;
          printf("   %-8s %s¥n", b_titleid, b_title);
          printf("update/delete? ");
          gets(response);
          if (!strncasecmp(response,"u",1))
          {
              printf("enter the new titleid¥n>");
              gets(b_titleid);
              exec sql update titles
                  set title_id = :b_titleid
                  where current of titlelist;
          }
          else if (!strncasecmp(response,"d",1))
          {
              exec sql delete from titles
                  where current of titlelist;
          }
 }
 exec sql close titlelist;
```

使用法

- open は declare cursor 文によって指定された select 文を実行し、fetch 文に対する結果を準備します。
- オープンするカーソルの数に制限はありません。
- 静的カーソルのオープンは、そのカーソルが宣言されるファイルでのみ実行してください。静的カーソルのクローズはどのファイルでも実行できます。
- declare cursor 文に埋め込まれたホスト変数の値は、オープン時に受け取られます。
- *cursor_name* を指定する場合、割り付けを解除された静的カーソルの名前を使用できます。その場合、プリコンパイラは割り付けが解除されたカーソルと同じ名前の新しいカーソルを宣言してオープンします。つまり、プリコンパイラは割り付けが解除されたカーソルを再度オープンするのではなく、新しいカーソルを作成します。これら 2 つのカーソルの結果セットは異なる可能性があります。

prepare

説明

動的 SQL 文バッファの名前を宣言します。

構文

exec sql [at *connection_name*] prepare *statement_name* from {:*host_variable* | "*string*"};

パラメータ

statement_name

文を参照するときに使用する識別子です。*statement_name* は文バッファをユニークに識別する必要があり、変数名を指定するための SQL 識別子規則に準拠する必要があります。また、*statement_name* には、有効な SQL 識別子を含む *host_variable* 文字列を指定することもできます。*statement_name* には最大 255 文字を指定できます。

host_variable

実行可能な SQL 文を含んでいる文字列ホスト変数です。ホスト変数の値が置き換えられる select 文の任意の場所に動的パラメータ・マーカ (“?”) を記述してください。

string

host_variable の代わりに使用できるリテラル文字列です。

例

```
exec sql begin declare section;
     CS_CHAR        dyn_buffer[128];
     CS_CHAR        state[2];
 exec sql end declare section;

 -- The select into table_name statement returns no
 -- results to the program, so it does not
 -- need a cursor.

 dyn_buffer = "select * into #work from authors"
         + "where state = ?”;

 printf("State? ");
 gets(state);
exec sql prepare make_work from :dyn_buffer;
exec sql execute make_work using :state;
```

使用法

- 現在の実装では、Sybase はリテラル文字列またはホスト変数に保管される動的 SQL 文のテンポラリ・ストアド・プロシージャを作成します。
- prepare は、*host_variable* の内容を Adaptive Server Enterprise へ送信し、テンポラリ・ストアド・プロシージャに変換します。このテンポラリ・ストアド・プロシージャは、文を割り付け解除するか接続を切断するまで Adaptive Server Enterprise の tempdb に存在します。
- *statement_name* のスコープはプログラムに対してはグローバルですが、接続 *connection_name* に対してはローカルです。この文は、プログラムによってその文の割り付けが解除されるか、接続がクローズされるまで継続します。
- prepare は、動的 SQL のメソッド 2、3、および 4 の場合に有効です。

- メソッド 2 (prepare および execute) では、ホスト変数があれば、その値を execute 文が準備した文に置き換え、完成した文を Adaptive Server Enterprise に送信します。置き換えるホスト変数がなく、結果もない場合には、代わりに execute immediate を使用できます。
- メソッド 3 (prepare および fetch) では、declare cursor 文が、保存された select 文とカーソルを対応付けます。ホスト変数があれば、その値を open 文が select 文に置き換え、実行のために結果を Adaptive Server Enterprise に送ります。
- メソッド 2、3、4 (パラメータ記述子を持つ prepare および fetch) では、疑問符 (“?”) で表された動的パラメータ記述子で、ホスト変数で置き換えられる箇所を示します。
- 準備文は、その文が準備された接続上で実行してください。準備文を使用してカーソルを宣言する場合、そのカーソル上でのすべてのオペレーションは、準備文と同じ接続を使用します。
- *host_variable* 内の文は、ホスト変数の値を文に置き換える箇所を示す動的パラメータ・マーカを含むことができます。

参照

declare cursor、execute、execute immediate、deallocate prepare

rollback

説明

データベースの変更を、トランザクション・セーブポイント、またはトランザクションの開始点にロールバックします。

構文

```
exec sql [at connection_name]
 rollback [transaction | tran | work]
 [transaction_name | savepoint_name];
```

パラメータ

transaction | trans | work

rollback 文では、キーワード transaction、trans、work は交換可能ですが work だけは ANSI 準拠です。

transaction_name

ロールバックするトランザクションの名前です。

savepoint_name

save transaction 文でセーブポイントに割り当てた名前です。*savepoint_name* を省略した場合、Adaptive Server はトランザクション全体をロールバックします。

例

```
abort_tran:
 exec sql whenever sqlerror continue:
 exec sql at connect2 rollback transaction;
exec sql at connect1 rollback transaction;
 goto try_update;
```

使用法

- このリファレンス・ページでは、主に Transact-SQL の rollback 文を Embedded SQL で使用した場合の相違点について説明します。『ASE リファレンス・マニュアル』を参照してください。
- トランザクション名とセーブポイント名は、Transact-SQL の識別子規則に準拠する必要があります。
- トランザクション名とセーブポイントは Transact-SQL の拡張機能であり、ANSI 標準には準拠していません。トランザクション名またはセーブポイント名を ANSI 標準に準拠するキーワード work とともに使用しないでください。

参照

begin transaction、commit

select

説明

データベース・オブジェクトからローを取得します。

構文

exec sql [at *connect_name*]
 select *select_list*
 into *destination*
from *table_name*…;

パラメータ

select_list

Embedded SQL での *select_list* は変数の割り当てを実行できませんが、それ以外は Transact-SQL の select 文内の *select_list* と同じです。

destination

テーブルまたは 1 つ以上の Embedded SQL ホスト変数です。前の declare セクションで最初にそれぞれのホスト変数を定義してください。インジケータ変数はホスト変数と対応付けることができます。

例

```
/* This example retrieves columns from a
 ** single row of the authors table and
 ** stores them in host variables.Because the
 ** example's select statement cannot return more
 ** than one row, no cursor is needed.
 */

 exec sql begin declare section;
     character        last[40];
     character        first[20];
     character        phone[12];
     character        id[11];
 exec sql end declare section;

 printf("Enter author id: ");
gets(id);
exec sql select au_lname, au_fname, phone
     into :last, :first, :phone
```

```
        from authors
        where au_id = :id;
    if (sqlcode != 100)
     {
            print "Information for Author ", id, ":";
            print last, first, phone;
     }
     else
     {
            print "Could not locate author ", id;
     };
```

使用法

- このリファレンス・ページでは、主に Transact-SQL の select 文を Embedded SQL で使用した場合の相違点について説明します。『ASE リファレンス・マニュアル』を参照してください。
- Embedded SQL プログラムでは、Transact-SQL の select 文の compute 句を使用できません。
- select 文内のホスト変数は、into 句以外では入力変数のみです。into 句内のホスト変数は出力変数です。
- 事前に宣言された入力ホスト変数は、リテラル値または Transact-SQL 変数が許される select 文内の任意の箇所で使用できます。入力ホスト変数にインジケータ変数を対応付けて、null 値を指定することができます。
- select 文が 2 つ以上のローを返す場合、その文の into 句内にあるそれぞれのホスト変数は、すべてのローを入れるのに十分な領域を持った配列にしてください。そうでない場合は、カーソルを使用してローを一度に 1 つずつ処理する必要があります。

参照 declare cursor

set connection

説明 指定した既存の接続を現在の接続に変更します。

構文 set connection {*connection_name* | DEFAULT};

パラメータ

connection_name
現在の接続に変更する既存の接続の名前です。

default
名前が指定されていないデフォルトの接続を現在の接続に変更することを指定します。

例

```
exec sql connect "ME" at connect1 using "SERVER1";
 exec sql connect "ME" at connect2 using "SERVER2";

 /* The next statement executes on connect2. */
 exec sql select userid() into :myid;

exec sql set connection connect1;

/* The next statement executes on connect1. */
 exec sql select count(*)from t1;
```

使用法

- set connection 文は、exec sql 文の at 句が前に入っている文を除き、後続のすべての SQL 文に対して現在の接続を指定します。
- set connection 文を再使用することによって現在の接続とは別の接続を選択するまで、set connection 文は有効になります。

参照

at connection_name、connect

set descriptor

説明

SQL 記述子へデータを挿入または更新します。

使用可能な SQL 記述子のデータ型のリストについては、表 10-5 (167 ページ)を参照してください。

構文

exec sql set descriptor *descriptor_name*
{count = *host_variable*} |
{value *item_number* {*item_name* =
:host_variable}[,...];

パラメータ

descriptor_name
準備文内の動的パラメータ・マーカについての情報が入っている SQL 記述子の名前です。

count
記述される動的パラメータの指定回数です。

host_variable
declare セクション内で定義されたホスト変数です。

item_number
動的パラメータ・マーカまたは選択カラムの *n* 番目のオカレンスを表します。

item_name

動的パラメータ・マーカまたは select リスト・カラムの属性情報を表します。表 10-4 に、*item_name* の値をリストします。

表 *10-4: item_name* の値

値	説明
data	指定した SQL 記述子と対応する動的パラメータ・マーカまたはターゲットの値。インジケータが負の値である場合、このフィールドは未定義である。
length	指定した SQL 記述子のターゲットの動的パラメータ・マーカの文字単位での長さ。
precision	CS_NUMERIC 変数の精度の有効桁数を指定する整数。
scale	CS_NUMERIC 変数の小数点桁数を指定する整数。
type	ロー内のこのカラム (項目番号) のデータ型。値については、表 10-5 (167 ページ) を参照。

例

```
exec sql prepare get_royalty
     from "select royalty from roysched
     where title_id = ?and lorange <= ?and
     hirange > ?";


exec sql allocate descriptor roy_desc with max 3;
exec sql set descriptor roy_desc
     value 1 data = :tid;
 exec sql set descriptor roy_desc
     value 2 data = :sales;
 exec sql set descriptor roy_desc
     value 3 data = :sales;
exec sql execute get_royalty into :royalty
     using sql descriptor roy_desc;
```

使用法

Embedded SQL プログラムは、属性および値の情報を Client-Library に渡します。Client-Library は、プログラムが文を実行する要求を発行するまで、指定された SQL 記述子にデータを保持します。

参照

allocate descriptor、describe input、describe output、execute、fetch、get descriptor、open (**動的カーソル**)

thread exit

説明

Embedded SQL プログラムで特定のスレッドに割り付けられたメモリを解放できます。

構文

exec sql thread_exit;

例

```
exec sql include sqlca;
 main()
 {
 ...
 for (;;)
 {
 /* A thread connects to Adaptive Server Enterprise,
 ** executes various embedded SQL statements,
 ** and then disconnects from
 ** Adaptive Server Enterprise
 */
 ...
 exec sql thread_exit;
 ...
 }
 /* The exit statement must be the last
 ** embedded SQL statement in the program.
 */
 exec sql exit;
 }     /* end of main */
```

使用法

- thread exit 文は特定のスレッドに割り付けたすべてのメモリ・リソースの割り付けを解除します。
- thread exit 文は Sybase の拡張機能であり、SQL 標準では定義されていません。

参照

exit

update

説明

テーブルのロー内のデータを変更します。

構文

exec sql [at *connection_name*] update *table_name*
 set [*table_name*]
    column_name1 = {*expression1*
       | NULL | (*select_statement*)}
    [, *column_name2* =
    {*expression2* | NULL
       | (*select_statement*)}]……
 [from *table_name*
    [, *table_name*]……
 [where {*search_conditions* | current of *cursor_name*}];

パラメータ

table_name
テーブルまたはビューの名前です。この名前を指定する場合、Transact-SQL の update 文に対して有効であればどのようなフォーマットでも使用できます。

例

```
exec sql begin declare section;
     CS_CHAR       store_name[40];
     CS_CHAR       disc_type[40];
     CS_INT        lowqty;
     CS_INT        highqty;
     CS_FLOAT      discount;
 exec sql end declare section;

 CS_CHAR      answer[1]);

 exec sql declare update_cursor cursor for
     select s.stor_name, d.discounttype,
     d.lowqty, d.highqty, d.discount
     from   stores s, discounts d
     where  d.stor_id = s.stor_id;

 exec sql open update_cursor;

 exec sql whenever not found goto alldone;

 while (TRUE) {
     exec sql fetch update_cursor into
         :store_name, :disc_type, :lowqty,
         :highqty, discount;
     print store_name, disc_type, lowqty,
         highqty, discount;
     printf("New discount? ");
     gets(discount);
    exec sql update discounts
         set discount = :discount
         where current of update_cursor;

 }

 alldone:
 exec sql close update_cursor;
 exec sql disconnect all;
```

使用法

- このリファレンス・ページでは、主に Transact-SQL の update 文を Embedded SQL で使用した場合の相違点について説明します。『ASE リファレンス・マニュアル』を参照してください。
- ホスト変数は、式または where 句内のどこにでも指定できます。

- where 句を使用して、テーブル内の選択したローを更新できます。テーブル内のすべてのローを更新するには、where 句を省略してください。オープンしているカーソルの現在のローを更新するには、where current of *cursor_name* を使用してください。
- where current of *cursor_name* を指定する場合、open cursor 文内で指定した接続でその文を実行してください。at *connection_name* 句を使用する場合、その句は open cursor 文と一致していなければなりません。

参照　close、delete cursor、fetch、open、prepare

whenever

説明　実行可能な SQL 文が、指定した条件になると実行する動作を指定します。

構文

```
exec sql whenever {sqlerror | not found | sqlwarning}
 {continue | go to label | goto label |
 stop | call routine_name [args]};
```

パラメータ

sqlerror
: Adaptive Server から Embedded SQL プログラムに返される構文エラーなど、エラーを検出したときに行う動作を指定します。

not found
: fetch 文または select into 文がデータを取得しない場合や、検索された update 文または delete 文がローに影響を与えない場合に行う動作を指定します。

sqlwarning
: 警告を受け取ったとき (たとえば、文字列がトランケートされたとき) に行う動作を指定します。

continue
: 指定した条件になっても何の動作も行いません。

go to | goto
: 指定した *label* のプログラム文に制御を渡します。

label
: C ラベルなどのホスト言語文ラベルです。

stop
: 指定した条件が発生したときに Embedded SQL プログラムを終了します。

call
: ユーザ定義の関数またはサブルーチンなどのプログラム内の呼び出し可能ルーチンに制御を渡します。

routine_name

呼び出しのできるホスト言語ルーチンです。このルーチンは、whenever 文を含んでいるソース・ファイルから呼び出し可能でなければなりません。Embedded SQL プログラムをコンパイルするには、そのルーチンを external として宣言する必要があります。

args

ホスト言語のパラメータ渡し規約を使用して、呼び出し可能なルーチンに渡す 1 つ以上の引数です。引数は、ホスト言語が許可するホスト変数、リテラル、または式のリストです。個々の引数を区切るには、スペース文字を使用します。

例

```
exec sql whenever sqlerror call err_handler();
exec sql whenever sqlwarning call warn_handler();


 long SQLCODE;
 exec sql begin declare section;
     CS_CHAR        lastname[40];
     CS_CHAR        firstname[20];
     CS_CHAR        phone[12];
exec sql end declare section;

 exec sql declare au_list cursor for
     select au_lname, au_fname, phone
     from authors
     order by au_lname;

exec sql open au_list;

 exec sql whenever not found go to list_done;

 while (TRUE){
        exec sql fetch au_list
             into :lastname, :firstname, :phone;
         printf("Lastname is:%s¥n", lastname,
"Firstname is:%s¥n", firstname,
         "Phone number is:%s¥n", phone;
 }
 list_done:
 exec sql close au_list;
 exec sql disconnect current;
```

使用法

- whenever 文によって、Embedded SQL プログラムのプリコンパイラは実行可能な各 SQL 文のあとにコードを生成します。生成されたコードには、条件に対するテスト、およびホスト言語文または指定された動作を行う文が含まれます。
- Embedded SQL プリコンパイラは、ソース・ファイル内の whenever 文のあとの SQL 文に対して、コードを生成します。これには、同じソース・ファイル内で定義された、サブルーチン内の SQL 文を含みます。
- 前回の whenever 文を取り消すには、whenever …continue を使用します。continue の動作は、Embedded SQL プログラムのプリコンパイラに条件を無視するように命令します。無限ループを防ぐには、エラー・ハンドラ内で whenever …continue を使用してから、Embedded SQL 文を実行してください。
- whenever …go to *label* を使用する場合、*label* は実行の継続可能なロケーションを示さなければなりません。たとえば C では、whenever 文のスコープ内で実行可能な SQL 文を持っているルーチン内で、*label* を宣言しなければなりません。C の場合、goto 文は他の関数内で宣言されたラベルにジャンプすることはできません。
- プログラム内で whenever 文を使用し、SQLCA または SQLSTATE ステータス変数を宣言していない場合、Embedded SQL プログラムのプリコンパイラは SQLCODE 変数が使用されていると想定します。SQLCODE が宣言されていることを確認してください。そうでない場合、生成されたコードはコンパイルされません。

SQL 記述子のコード

表 10-5 は、動的 SQL 文に使用される SQL 記述子を示します。Sybase での動的 SQL 値の使用方法は、ANSI/ISO 185-92 SQL-92 標準に準拠しています。対応する ANSI/ISO のマニュアルを参照してください。

表 *10-5: SQL* 記述子のデータ型のコード

ANSI SQL データ型	コード
bit	14
character	1
character varying	12
date、time	9
decimal	3
double precision	8
float	6
integer	4
numeric	2
real	7
smallint	5

Sybase 定義のデータ型	Client-Library コード
smalldatetime	-9
money	-10
smallmoney	-11
text	-3
image	-4
tinyint	-8
binary	-5
varbinary	-6
long binary	-7
longchar	-2

表 *10-6: SQL* 記述子の識別子値

値	説明
type	ロー内のこのカラム (項目番号) のデータ型。値については、表 10-5 (167 ページ) を参照。
length	指定した SQL 記述子のターゲットの動的パラメータ・マーカの文字単位での長さ。
returned_length	select カラム・リストからの char データ型の値の長さ。
precision	CS_NUMERIC 変数の精度の有効桁数を指定する整数。
scale	CS_NUMERIC 変数の小数点桁数を指定する整数。
nullable	動的パラメータ・マーカで null 値が許可される場合は 0 であり、null 値が許可されない場合は 1 である。
indicator	動的パラメータ・マーカまたはターゲットと対応するインジケータ・パラメータの値。
data	指定した SQL 記述子と対応する動的パラメータ・マーカまたはターゲットの値。インジケータが負の値である場合、このフィールドは未定義である。
name	動的パラメータ・マーカについての情報が入っている、指定した SQL 記述子の名前。

第 11 章

Open Client/Server 設定ファイル

Open Client/Server 設定ファイルを使用すると、Open Client/Server アプリケーションを簡単に設定できます。デフォルトではこの設定ファイルは *ocs.cfg* という名前で、*$SYBASE/$SYBASE_OCS/config* ディレクトリにあります。この章では、Embedded SQL でのこの設定ファイルの使用方法について説明します。

Open Client/Server 設定ファイルの使用目的

Open Client/Server 設定ファイルを使用すると、すべての Open Client/Server アプリケーションの接続を一箇所で設定できます。この設定ファイルを使用すると、標準の設定を確立する作業と設定の変更を管理する作業が簡単になります。

設定機能へのアクセス

この機能は、initialize_application 文の次の 2 つのコマンドライン・オプションを使用してアクセスできます。

- -x – これは外部設定のためのオプションです。アプリケーションはある名前を使用してアプリケーションを初期化する必要があります。Open Client/Server 設定ファイルには、このアプリケーション名を使用するセクションがあります。このセクションには、このアプリケーションに対して設定する必要があるすべてのプロパティを指定してください。-x オプションは、initialize_application 文とともに使用した場合にだけ有効になります。初期化を行わないで -x オプションを使用した場合は、設定ファイルのデフォルト・セクションがアクセスされます。

- -e – このオプションは、サーバ名を使用して設定できるようにします。initialize_application 文への呼び出しは必要ありません。サーバ名は、そのサーバ名によって定義されたセクションに設定されるプロパティを設定ファイル内で検索するためのキーとして使用されます。これによって、ユーザは特定の接続プロパティと接続名を対応させることができます。

注意 実行される最初の Embedded SQL 文が INITIALIZE_APPLICATION 文でない場合は、外部設定プロパティは設定されません。実行される最初の Embedded SQL 文が INITIALIZE_APPLICATION 文である場合は、外部設定オプションが初期化に使用されます。

デフォルト設定

次は、デフォルト設定の Open Client/Server 設定ファイルです。必要に応じてこのファイルをカスタマイズできます。

```
[DEFAULT]
;This is the default section loaded by applications that use the
 ;external configuration feature, but which do not specify their
 ;own application name.Initially this section is empty.Defaults
 ;from all properties will be the same as earlier releases of
 ;Open Client libraries.
[ANSI_ESQL]
;This section defines configuration which an ANSI conforming
 ;Embedded SQL application should use to get ANSI-defined
 ;behavior from Adaptive Server Enterprises and Open Client libraries.This set of
 ;configuration ;properties matches the set which earlier
 ;releases of Embedded SQL (version 10.0.x) automatically set for
 ;applications duringexecution of a CONNECT statement.
CS_CAP_RESPONSE=CS_RES_NOSTRIPBLANKS
 CS_EXTRA_INF=CS_TRUE
 CS_ANSI_BINDS=CS_TRUE
 CS_OPT_ANSINULL=CS_TRUE
 CS_OPT_ANSIPERM=CS_TRUE
 CS_OPT_STR_RTRUNC=CS_TRUE
 CS_OPT_ARITHABORT=CS_FALSE
 CS_OPT_TRUNCIGNORE=CS_TRUE
 CS_OPT_ISOLATION=CS_OPT_LEVEL3
 CS_OPT_CHAINXACTS=CS_TRUE
 CS_OPT_CURCLOSEONXACT=CS_TRUE
 CS_OPT_QUOTED_IDENT=CS_TRUE
 ;End of default sections
```

Open Client/Server 設定ファイルの構文

Open Client/Server 設定ファイルの修正に使用する構文は、多少の違いはありますが、CS-Library でサポートされる Sybase ローカライゼーション・ファイルと設定ファイルの既存の構文とほぼ一致しています。

構文は、次のとおりです。

- ; ― コメント行を示します。
- [section_name] ― セクション名は角カッコ ([]) で囲まれています。Open Client/Server 設定ファイルには DEFAULT と ANSI_ESQL という名前のセクションがあります。-x オプションを使用してコンパイルされたアプリケーションでは、アプリケーション名がセクション名として使用されます。-e オプションを使用してコンパイルされたアプリケーションでは、そのサーバ名がセクション名に使用されます。複数のセクションで使用される設定が入っているセクションの名前は、どのような名前でも使用できます。次の例は、“GENERIC” という任意の名前のセクションと、そのセクションがその他のセクションにどのように指定されているかを示します。

```
[GENERIC]
 CS_OPT_ANSINULL=CS_TRUE
[APP_PAYROLL]
 include=GENERIC
 CS_CAP_RESPONSE=CS_RES_NOSTRIPBLANKS
[APP_HR]
 include=GENERIC
 CS_OPT_QUOTED_IDENT=CS_TRUE
```

- entry_name=entry_value
  - エントリ値には、整数や文字列など、どのような値でも指定できます。エントリ値の行が '¥<newline> で終了する場合、そのエントリ値は次の行に続きます。
  - エントリ値の最初と最後にあるスペースはトリムされます。
  - エントリ値の最初または最後にスペースが必要な場合は、そのスペースを二重引用符で囲んでください。
  - 二重引用符で始まるエントリは、二重引用符で終了する必要があります。引用符で囲まれた文字列の中に連続する 2 つの二重引用符がある場合、それは値文字列の中の 1 つの二重引用符を表します。二重引用符内に改行文字が検出された場合は、文字どおりにその値の一部とみなされます。
  - エントリ名とセクション名は、アルファベット (大文字と小文字の両方)、0 ～ 9 の数字、以下の任意の区切り文字 (デリミタ) から構成できます。! " # $ % & ' ( ) * + , - . / : ; < > ? @ ¥ ^ _ ` { | } ~。

    角カッコ ([ ])、スペース、等号記号 (=) はサポートされません。ただし、最初の文字は必ずアルファベットにしてください。

- エントリ名とセクション名は大文字と小文字が区別されます。
- Include=*earlier_section*

  セクションに include というエントリがある場合、以前に定義されたセクションの内容全体がこのセクション内にコピーされるとみなされます。つまり、以前のセクションで定義されたプロパティはこのセクションによって継承されます。

  インクルードされるセクションは、別のセクションにインクルードされる前に定義されている必要があるので注意してください。これによって、設定ファイルの解析を単一のパスで行うことができ、再帰的にインクルードされたディレクティブ (命令語) を検出する必要がなくなります。

  インクルードされたセクションが順に別のセクションをインクルードする場合、エントリ値の順序は、インクルードされたセクションの「入れ子の深い方から浅い方へ」の検索によって定義されます。

  セクションには、そのセクション自体への参照をインクルードすることはできません。つまり、以前に定義されたセクションをインクルードする必要があるので再帰は不可能です。また、定義されていないセクションをインクルードすることはできません。

  あるセクションで定義されたすべての直接エントリ値は、別のセクションからインクルードされた可能性のある値を置き換えます。次の例では、CS_OPT_ANSINULL は APP.PAYROLL アプリケーションでは false に設定されます。include 文の位置は、この規則には影響を与えません。

  ```
  [GENERIC]
   CS_OPT_ANSINULL=CS_TRUE
  [APP_PAYROLL]
   CS_OPT_ANSINULL=CS_FALSE
   include=GENERIC
  ```

サンプル・プログラム

次のような例があると仮定します。Embedded SQL プログラムではカーソルを定義して、pubs2 データベースにある titles テーブルからローを取得します。WHERE 句は ANSI 標準ではない NULL チェック機能を使用します。IS NULL と IS NOT NULL は、Embedded SQL プログラムで使用されるデフォルトであり、これは ANSI 標準です。それに対して、= NULL または != NULL を使用する Embedded SQL プログラムは、ANSINULL 動作を OFF にして、代わりに Transact-SQL 構文を使用する必要があります。

次の例では Embedded SQL コードへの変更はありませんが、Open Client/Server 設定ファイルに適切なプロパティを設定することによって適切な動作を実行します。

以降の項に同じプログラムの 2 つのバージョンを示します。1 つは -e オプションを使用するバージョンであり、もう 1 つは -x オプションを使用するバージョンです。

Embedded SQL/C の makefile サンプル・ファイル (Windows)

libsybcobct.lib および *mfrts32.lib* ライブラリは、Embedded SQL/C の makefile サンプル・ファイルに含まれている必要はありません。

makefile での CC_INCLUDE 変数を、次のように変更する必要があります。

```
CC_INCLUDES= -I$(SYBASE)¥include
```

注意　Microsoft Windows でサンプル・プログラムをコンパイルするときには、make ではなく nmake コマンドを使用します。

Embedded SQL/C サンプル・プログラム

UNIX プラットフォームで Embedded SQL/C サンプル・プログラムを開発するには、事前に以下の準備が必要です。

- ファイルの所有者に対し、*sybopts.sh* の実行パーミッションを次のように設定する。

  ```
  chmod u+x sybopts.sh
  ```

- 検索パスに現在のディレクトリを追加する (まだ指定していない場合)。

  ```
  setenv PATH .:$PATH
  ```

-x オプションを使用する Embedded SQL プログラム

```
/* Program name:ocs_test.cp
 **
 ** Description  :This program declares a cursor which retireves rows
 **  from the 'titles' table based on condition checking for NULLS
 **  in the NON-ANSI style.
 **  The program will be compiled using the -x option which will
 **  use an external configuration file (ocs.cfg) based on the
 **  name of the application.The name of the application is
 **  defined at the time of INITIALIZING the application.Note that
 **  this is a new 11.x feature too.
 */

 #include <stdio.h>

 /* Declare the SQLCA */
 EXEC SQL INCLUDE sqlca;

 EXEC SQL BEGIN DECLARE SECTION;
         /* storage for login name and password */
         CS_CHARusername[30], password[30];
         CS_CHARtitle_id[7], price[30];
 EXEC SQL END DECLARE SECTION;


 /*
 ** Forward declarations of the error and message handlers and
 ** other subroutines called from main().
 */
 void    error_handler();
 void    warning_handler();

 int main()
 {
         int i=0 ;

         EXEC SQL WHENEVER SQLERROR CALL error_handler();
         EXEC SQL WHENEVER SQLWARNING CALL warning_handler();
         EXEC SQL WHENEVER NOT FOUND CONTINUE ;

         /*
         ** Copy the user name and password defined in sybsqlex.h to
         ** the variables declared for them in the declare section.
         */

         strcpy(username, "sa");
         strcpy(password, "");

         EXEC SQL INITIALIZE_APPLICATION APPLICATION_NAME = "TEST1";
```

```
        EXEC SQL CONNECT :username IDENTIFIED BY :password ;
        EXEC SQL USE pubs2 ;

        EXEC SQL DECLARE title_list CURSOR FOR
        SELECT title_id, price FROM titles
                WHERE price != NULL;

        EXEC SQL OPEN title_list ;
        for ( ;; )
        {
                EXEC SQL FETCH title_list INTO
                          :title_id, :price;
                if (sqlca.sqlcode == 100)
                {
                          printf("End of fetch!¥n");
                          break;
                }
                printf("Title ID :%s¥n", title_id );
                printf("Price    :%s¥n", price) ;
                printf("Please press RETURN to continue .. ");
                getchar();
                printf("¥n¥n");
       }
       EXEC SQL CLOSE title_list;
       exit(0);

}

void error_handler()
{
       . . .}

void warning_handler()
{
      . . .}
```

注意　makefile で設定するプリコンパイラ・オプション：cpre -x

次は、前述のプログラムの設定ファイルの例です。

```
[DEFAULT]
 ;

 [TEST1]
 ;This is name of the application set by INITIALIZE_APPLICATION.;Therefore this
is the section that will be referred to a runtime.

 CS_OPT_ANSINULL=CS_FALSE

 ;The above option will enable comparisons of nulls in the NON-ANSI
 ;style.
```

-e オプションを使用する同じ Embedded SQL プログラム

```
/* Program name:ocs_test.cp
 **
 ** Description  :This program declares a cursor which retireves rows
 ** from the 'titles' table based on condition checking for NULLS
 ** in the NON-ANSI style.
 ** The program will be compiled using the -e option which will
 ** use the server name that the application connects to, as the
 ** corresponding section to look up in the configuration file.
 */

 #include <stdio.h>

 /* Declare the SQLCA */
 EXEC SQL INCLUDE sqlca;

 EXEC SQL BEGIN DECLARE SECTION;
        /* storage for login name and password */
         CS_CHARusername[30], password[30];
         CS_CHARtitle_id[7], price[30];
 EXEC SQL END DECLARE SECTION;


 /*
 ** Forward declarations of the error and message handlers and
 ** other subroutines called from main().
 */
 void    error_handler();
 void    warning_handler();

 int main()
 {
         int i=0 ;

         EXEC SQL WHENEVER SQLERROR CALL error_handler();
         EXEC SQL WHENEVER SQLWARNING CALL warning_handler();
         EXEC SQL WHENEVER NOT FOUND CONTINUE ;

         /*
         ** Copy the user name and password defined in sybsqlex.h to
         ** the variables declared for them in the declare section.
         */

         strcpy(username, "sa");
         strcpy(password, "");

         EXEC SQL CONNECT :username IDENTIFIED BY :password ;
         EXEC SQL USE pubs2 ;

         EXEC SQL DECLARE title_list CURSOR FOR
```

```
                SELECT title_id, price FROM titles
                      WHERE price != NULL;

        EXEC SQL OPEN title_list ;
        for ( ;; )
        {
               EXEC SQL FETCH title_list INTO
                         :title_id, :price;
               if (sqlca.sqlcode == 100)
               {
                       printf("End of fetch!¥n");
                       break;
               }
               printf("Title ID :%s¥n", title_id );
               printf("Price    :%s¥n", price) ;
               printf("Please press RETURN to continue .. ");
               getchar();
               printf("¥n¥n");
        }
        EXEC SQL CLOSE title_list;
        exit(0);

 }

 void error_handler()
 {
         . . .}
```

注意　makefile で設定するプリコンパイラ・オプション：cpre -e

次は、前述のプログラムの設定ファイルの例です。

```
[DEFAULT]
 ;

 [SYBASE]
 ;This is name of the server that the application connect to.Therefore
 ;this is the section that will be referred to a runtime.
 ;
 CS_OPT_ANSINULL=CS_FALSE
 ;The above option will enable comparisons of nulls in the NON-ANSI
 ;style.
```

上記の設定ファイルはかなり簡略化してあります。一般的な Open Client/Server 設定ファイルは次のような形式になります。

```
[DEFAULT]
 ;
 [ANSI_ESQL]
 CS_CAP_RESPONSE=CS_RES_NOSTRIPBLANKS
 CS_EXTRA_INF=CS_TRUE
```

```
CS_ANSI_BINDS=CS_TRUE
CS_OPT_ANSINULL=CS_TRUE
CS_OPT_ANSIPERM=CS_TRUE
CS_OPT_STR_RTRUNC=CS_TRUE
CS_OPT_ARITHABORT=CS_FALSE
CS_OPT_TRUNCIGNORE=CS_TRUE
CS_OPT_ISOLATION=CS_OPT_LEVEL3
CS_OPT_CHAINXACTS=CS_TRUE
CS_OPT_CURCLOSEONXACT=CS_TRUE
CS_OPT_QUOTED_IDENT=CS_TRUE
;
;The following is a sample section showing how to alter standard
;configuration:
;
[RELEVANT_SECION_NAME]
;
;Use most of the ANSI properties defined above,
;
include=ANSI_ESQL

;but override some default properties

CS_OPT_ANSINULL=CS_TRUE    ; enable non-ansi style null comparisons
CS_OPT_CHAINXACTS=CS_FALSE ; run in autocommit mode
```

まとめ

Open Client/Server 設定ファイルを使用すると、複数の Embedded SQL アプリケーションの環境設定を 1 か所で管理できます。この設定ファイルのデフォルト名は *ocs.cfg* であり、*$SYBASE/$SYBASE_OCS/config* ディレクトリにあります。この設定ファイルの使用は、プリコンパイラ・オプション -x と -e を使用して管理されます。Open Client/Server 設定ファイルの修正に使用する構文は、多少の違いはありますが、CS-Library でサポートされる Sybase ローカライゼーション・ファイルと設定ファイルの既存の構文とほぼ一致しています。

付 録 A

プリコンパイラの警告とエラー・メッセージ

Embedded SQL プリコンパイラは、次の表に示す情報メッセージ、警告メッセージ、エラー・メッセージを生成します。

表には、それぞれ 4 つのフィールドがあります。

- 「メッセージ ID」には、表示されるメッセージの識別コードをリストしています。
- 「メッセージ」には、表示されるメッセージのオンライン・テキストをリストしています。
- 「重大度」には、表示されるメッセージの重大度をリストしています。意味は、次のとおりです。
    - 情報メッセージ − エラーも警告も検出されず、プリコンパイラの作業は成功しました。このメッセージは、情報を提供するだけのものです。
    - 警告 − 致命的でないエラーが検出されましたが、プログラムはプリコンパイルされました。
    - 重大なエラー − エラーが発生し、コードの生成は行われませんでした。プリコンパイルは失敗しました。
    - 致命的なエラー − プリコンパイラがリカバリできない重大なエラーが発生しました。これ以上ファイルを処理できません。プリコンパイラは終了します。

- 4 つめのフィールド「対処法」には、エラーや警告の原因になった状態を修正するための方法をリストします。

表 *A-1:* コマンド・ライン・オプション・メッセージ

メッセージ ID	メッセージ	重大度	対処法
M_COMPAT_INFO	互換モードが指定されました。	情報メッセージ	対処不要。
M_DUPOPT	重複したコマンド・ライン・オプションが指定されました。	重大なエラー	コマンド・ラインでは、オプションを重複して指定してはならない。間違って指定した方のオプションを削除する。
M_EXCFG_OVERRIDE	外部設定スイッチ value が指定されていないため、スイッチ value による影響はありません。	警告	外部設定ファイルを使用している場合、コマンド・ラインで設定されているオプションを上書きしている可能性がある。設定オプションの 1 つを選択する。
M_INVALID_COMPAT	認識できない互換モードが指定されました。	情報メッセージ	対処不要。
M_INVALID_FILE_FMT	ファイル value の行 value に正しくない文字があります。	重大なエラー	入力ファイルの文字が正しいか確認する。また、使用する文字セットの設定が正しいことを確認する。
M_INVALID_FIPLEVEL	指定した FIPS レベルが正しくありません。	重大なエラー	正しい値は、SQL92E と SQL89。
M_INVALID_SYNLEVEL	指定した構文チェック・レベルが正しくありません。	重大なエラー	正しい値は、NONE、SYNTAX、SEMANTIC。
M_INVLD_HLANG	指定されたホスト言語が正しくありません。	重大なエラー	有効なオプションは、ANSI_C、KR_C。
M_INVLD_OCLIB_VER	The Open Client Client-Library version is invalid.	重大なエラー	正しいバージョン文字列は "CS_VERSION_xxx"。*xxx* は現在のバージョン。
M_INVOPT	オプションが正しくありません。	重大なエラー	無効なオプション指定。正しい値に置き換える。
M_LABEL_SYNTAX	セキュリティ・ラベルが不適切に指定されています。正しいフォーマットは、'labelname=labelvalue' です。	重大なエラー	許可されている構文を使用する。
M_MSGINIT_FAIL	ローカライズされたエラー・メッセージの初期化のエラーです。	警告	Sybase のインストールが完了しているか、*locales.dat* ファイルに LANG 変数に対する有効なエントリがあるかどうかを確認する。
M_MULTI_IN_USE_DEF_OUT	複数の入力ファイルをプリコンパイルするときは出力 (リスティング、SQL、または言語) ファイル名を指定できません。	重大なエラー	コマンド・ラインから -G、-L、-O フラグをすべて削除するか、1 回にプリコンパイルするファイルを 1 つだけにする。

メッセージ ID	メッセージ	重大度	対処法
M_NO_INPUT_FILE	エラー : プリコンパイルする入力ファイルが指定されていません。	重大なエラー	プリコンパイルする入力ファイルを指定する。 **注意**　このエラーは、オプション引数のフラグ (ストアド・プロシージャを生成する -G など) を入力ファイル名の前に置くと発生することがある。これを解決するには、入力ファイル名の前に別のフラグを付ける。たとえば、cpre -G file.pc を cpre -G -C*compilername* に置き換える。
M_OPEN_INCLUDE	インクルード・ファイル *file* をオープンできません。	重大なエラー	指定したファイルがパスにないか、必要な読み込みパーミッションがない。-I フラグを使用してパスを指定するか、読み込みパーミッションを確認する。
M_OPEN_INPUT	入力ファイル *file* をオープンできません。	重大なエラー	指定したパスとファイル名が有効かどうかを確認する。ファイル名の拡張子を指定していない場合、プリコンパイラはデフォルトの拡張子を検索する。
M_OPEN_ISQL	ISQL ファイル *file* をオープンできません。	重大なエラー	isql ファイル名 (ストアド・プロシージャが記述されているファイル) が有効かどうかを確認する。ファイルを作成しているディレクトリに、書き込みパーミッションがあるかどうかを確認する。
M_OPEN_LIST	リスティング・ファイル *file* をオープンできません。	重大なエラー	リスティング・ファイル名が有効かどうかを確認する。ファイルを作成しているディレクトリに、書き込みパーミッションがあるかどうかを確認する。
M_OPEN_TARGT	ターゲット・ファイル *file* をオープンできません。	重大なエラー	出力ファイル名が有効かどうかを確認する。ファイルを作成しているディレクトリに、書き込みパーミッションがあるかどうかを確認する。
M_OPT_MUST_BE_PROVIDED	オプション *value* を指定しなければいけません。	重大なエラー	オプションの値を設定する。
M_OPT_REINIT	警告：*value* スイッチが複数回初期化されました。	警告	指定したスイッチが複数回初期化されている。2 回目以降の値は無視される。

メッセージ ID	メッセージ	重大度	対処法
M_PATH_OFL	エラー : インクルード・ファイルの最大許容パスは 64 です。(オーバーフロー)	重大なエラー	コマンド・ラインで指定できるパスの最大数を超えた。*"INCLUDE"* ファイルをフェッチするディレクトリの数を減らす。
M_STATIC_HV_CNAME	静的カーソル名はホスト変数ではいけません。*line*	重大なエラー	ホスト変数を SQL 識別子に置き換える。
M_UNBALANCED_DQ	区切られた識別子に対になっていない引用符があります。	重大なエラー	左右の引用符の数を一致させる。

表 *A-2:* 第 *1* パス・パーサ・メッセージ

メッセージ ID	メッセージ	重大度	対処法
M_64BIT_INT	警告：64 ビットの整数ホスト変数はサポートされていません。行 *value*	警告	他のホスト変数データ型 (浮動小数点、数値、32 ビット整数) を使用する。必要なら、ホスト変数と 64 ビット・プログラム変数間で値をコピーする。
M_BLOCK_ERROR	*value* の行 *value* にノンマッチング・ブロック・ターミネータがあります。	重大なエラー	プログラムの構文を修正する。
M_CONST_FETCH	エラー :CONST 記憶クラスの変数 *value* にフェッチしようとしました。	重大なエラー	定数データ型への fetch into はできない。値をフェッチするには、宣言から定数修飾子を削除する。
M_DUP_HV	*file* の行 *line* に重複したホスト変数があります。	重大なエラー	同じブロック内で、すでに同じ名前のホスト変数が宣言されている。表示されたブロックの各変数がユニークであるかどうかを確認する。
M_DUP_STRUNION	*file* の行 *line* に重複した構造体／共用体があります。	重大なエラー	同じブロック内で、すでに同じ名前の構造体が宣言されている。表示されたブロックの各変数がユニークであるかどうかを確認する。
M_IDENT_OR_STRINGVAR	エラー : 項目は、SQL 識別子か文字列型変数でなければなりません。	重大なエラー	接続、カーソル、または文名が、文字列型または SQL 識別子になっていることを確認する。
M_ILL_LITERAL_USAGE	エラー :OUTPUT 修飾子を持つストアド・プロシージャにリテラル・パラメータを使用することは、無効です。	重大なエラー	リテラルを、ストアド・プロシージャに対する OUTPUT パラメータとして使用しないようにする。
M_ILL_PARAM_MODE	エラー: ファイル *file* 行 *line* でのリモート・プロシージャ・コールに呼び出しモードが混在しています。	重大なエラー	名前または位置を使って渡した引数を使用して、ストアド・プロシージャを呼び出す。これらのモードを同じ呼び出しで同時に使用すると無効になる。
M_INDICVAR	エラー : 項目は、インジケータ・タイプ変数でなければなりません。	重大なエラー	short integer を使用する。

メッセージ ID	メッセージ	重大度	対処法
M_INTVAR	エラー : 項目は、整数タイプ変数でなければなりません。	重大なエラー	整数を使用する。
M_MISMATCHED_QUOTES	エラー :16 進リテラル *value* にある引用符が正しくありません。	重大なエラー	左右の引用符の数を一致させる。
M_MULTIDIM_ARRAY	エラー : 行 *line*。多次元配列の変数は、サポートされていません。	重大なエラー	多次元の配列はサポートしていない。*m x n* 配列を分割して、それぞれ *n* 個の要素からなる *m* 配列にする。
M_MULTI_RESULTS	エラー : 行 *line* の Embedded Query が複数の結果セットを返します。	重大なエラー	クエリを複数のクエリに分割し、各クエリが 1 つの結果セットを返すようにする。または、テンポラリ・テーブルをすべての値で満たすようにクエリを書き直し、テンポラリ・テーブルから選択することによって、1 つの結果セットを返す。
M_NODCL_NONANSI	警告：非 ANSI モードで SQLCODE または SQLCA のどちらも宣言されていません。	警告	非 ANSI モードでは、SQLCA と SQLCODE のいずれか、または両方を宣言する。プログラム内の Embedded SQL 文すべてをスコープに適用できるかどうかを確認する。
M_NOLITERAL	エラー : 項目が、引用符で囲まれていない名前の可能性があります。	重大なエラー	引用符付きの名前またはホスト変数を使用する。
M_NOSQUOTE	エラー : 項目は、一重引用符で囲まれていてはいけません。二重引用符を使用してください。	重大なエラー	二重引用符を使用する。
M_NOT_AT_ABLE	“at” 句を使用できない文タイプで “at” 句が使用されています。これは、文 *value* で発生しました。	重大なエラー	指定した文から at 句を削除する。
M_NUMBER_OR_INDICVAR	エラー : 項目は、整数またはインジケータ・タイプ変数でなければなりません。	重大なエラー	リテラル integer、short integer、または CS_SMALLINT を使用する。
M_NUMBER_OR_INTVAR	エラー : 項目は、整定数か整数型変数でなければなりません。	重大なエラー	未使用。動的 SQL 文内のあるフィールド (たとえば、MAX、Value *n*) が、integer 型でも整数定数でもない場合に、このエラーを発生させるために使用することがある。
M_PARAM_RESULTS	エラー : 行 *line* の Embedded Query が予期しないパラメータ結果セットを返します。	重大なエラー	オプションのサーバ構文の確認中にのみ発生する。クエリがパラメータを返している原因を調べ、クエリを書き直す。

メッセージ ID	メッセージ	重大度	対処法
M_PASS1_ERR	ファイル *file*: パス 1 の構文エラーです。パス 2 は、実行されません。	情報メッセージ	パス 1 で発生したエラーのため、プリコンパイルがアボートされた。パス 1 のエラーを修正し、処理を進める。
M_PTR_IN_DEC_SEC	警告：宣言部分でのポインタは、サポートされていません。	警告	
M_QSTRING_OR_STRINGVAR	エラー : 項目は、引用文字列か文字列型変数でなければなりません。	重大なエラー	サーバ名、ユーザ名、パスワードが、二重引用符で囲まれた文字列または文字列データ型になっているかどうかを確認する。
M_SCALAR_CHAR	エラー :*line* 行で、配列のない文字列変数 *value* がホスト変数として不正に使用されています。	重大なエラー	文字配列を使用する。
M_SQLCA_IGNR	警告：SQLCODE と SQLCA の両方が宣言されています。SQLCA は無視されます。	警告	2 つの宣言のうちの 1 つを削除する。
M_SQLCA_WARN	警告：ANSI モードで、INCLUDE SQLCA がありました。SQLCA は無視されます。	警告	
M_SQLCODE_UNDCL	警告：ANSI モードで、SQLCODE が宣言されていません。	警告	SQLCODE を宣言する。
M_STATE_CODE	警告：SQLSTATE と SQLCODE の両方が宣言されています。SQLCODE は無視されます。	警告	2 つの宣言のうちの 1 つを削除する。
M_STATE_SQLCA	警告：SQLSTATE と SQLCA の両方が宣言されています。SQLCA は無視されます。	警告	2 つの宣言のうちの 1 つを削除する。
M_STATUS_RESULTS	エラー : 行 *line* の Embedded Query が予期しないステータス結果セットを返します。	重大なエラー	オプションのサーバ構文の確認中にのみ発生する。クエリがステータス結果を返している原因を調べ、クエリを書き直す。
M_STICKY_AUTOVAR	警告：*line* 行で、スティッキ・バインドとともに自動変数 *value* が使用されています。これにより、実行時に正しくない結果またはエラーが発生する可能性があります。	警告	この場合、使用しているプログラムの論理がエラーを許可しないことを確認する。または、静的変数またはグローバル変数を使用する。
M_STICKY_REGVAR	エラー :*line* 行で、スティッキ・バインドとともにレジスタ変数 *value* を使用できません。	重大なエラー	レジスタ修飾子を削除する。

メッセージ ID	メッセージ	重大度	対処法
M_STRUCT_NOTFOUND	*file* の行 *line* のスコープに構造体／共用体定義がありません。	重大なエラー	構造体や共用体の定義が指定された行のスコープ内であるかどうかを確認する。
M_SYNTAX_PARSE	ファイル *file* の行 *line* の付近に構文エラーがあります。	重大なエラー	表示された行番号の Embedded SQL の文法に構文エラーがないかどうかを確認する。
M_UNBALANCED_DQ	区切られた識別子に対になっていない引用符があります。	重大なエラー	左右の引用符の数を一致させる。
M_UNDEF_ELM	エラー *value*：構造体／共用体の要素が不正です。	重大なエラー	構造体の指定された要素が構造体の定義に入っていない。定義を修正する。
M_UNDEF_HV	ホスト変数 *value* は定義されていません。	重大なエラー	適切な場所でホスト変数を定義する。
M_UNDEF_IV	Indicator variable *value* undefined.	重大なエラー	適切な場所でインジケータ変数を定義する。
M_UNDEF_STR	エラー : 構造体 *value* は定義されていません。	重大なエラー	未定義の構造体が指定した行にある。適切なスコープ内で構造体を定義する。
M_UNSUP	*value* 機能は、このバージョンではサポートされていません。	致命的なエラー	この機能はサポートされていない。

表 *A-3:* 第 *2* パス・パーサ・メッセージ

メッセージ ID	メッセージ	重大度	対処法
M_CURSOR_RD	ファイル *file* の行 *line* で、カーソル *value* が再定義されています。	警告	同じ名前のカーソルがすでに宣言されている。違う名前を使用する。
M_HOSTVAR_ MULTIBIND	警告：ホスト変数 *value* がバインド変数として 1 つの文に 2 度以上使用されました。	警告	1 つのフェッチ文内で複数回ホスト変数を使用してはならない。複数の結果を 1 つのロケーションへフェッチすることはできない。Client-Library は、最後にフェッチした値を変数に代入する。
M_INVTYPE_IV	インジケータ変数のタイプが正しくありません。	重大なエラー	インジケータ変数は、CS_SMALLINT 型または INDICATOR 型にする。
M_PARSE_INTERNAL	行 *line* で内部パーサ・エラーです。サイベースにお問い合わせください。	致命的なエラー	この内部一貫性パーサ・エラーが発生した場合は、ただちに Sybase 製品の保守契約を結んでいるサポート・センタに連絡する。
M_SQLCANF	'INCLUDE SQLCA' 文がありません。	警告	文を追加する。

メッセージ ID	メッセージ	重大度	対処法
M_WHEN_ERROR	SQL 文 'WHENEVER SQLERROR' がありません。	警告	WHENEVER SQLERROR 文を追加するか、コマンド・ライン・オプションを使用して、警告と 'INTO' メッセージを出さないようにする (『Open Client/Server プログラマーズ・ガイド補足』参照)。
M_WHEN_NF	SQL 文 'WHENEVER NOT FOUND' がありません。	警告	WHENEVER NOT FOUND 文を追加するか、コマンド・ライン・オプションを使用して、警告と 'INTO' メッセージを出さないようにする (『Open Client/Server プログラマーズ・ガイド補足』参照)。
M_WHEN_WARN	SQL 文 'WHENEVER WARNING' がありません。	警告	WHENEVER WARNING 文を追加するか、コマンド・ライン・オプションを使用して、警告と 'INTO' メッセージを出さないようにする (『Open Client/Server プログラマーズ・ガイド補足』参照)。

表 *A-4:* コード生成メッセージ

メッセージ ID	メッセージ	重大度	対処法
M_INCLUDE_PATHLEN	インクルードまたはコピーされたパスが長すぎます。生成されたファイル名 *value* からパスを取り除いてください。	警告	リンクを使用するか、ファイルを短いパスに移動する。
M_WRITE_ISQL	isql ファイルに書き込みできません。リターン・コード :*value*	致命的なエラー	isql ファイルとディレクトリに対する作成および書き込みパーミッションを確認する。また、ファイル・システムが満杯でないことを確認する。
M_WRITE_TARGT	ターゲット・ファイルに書き込みできません。リターン・コード :*value*	致命的なエラー	プリコンパイラがターゲット・ファイルを生成するディレクトリに対するファイルの作成および書き込みパーミッションがあるかどうかを確認する。また、ファイル・システムが満杯でないことを確認する。

表 *A-5: FIPS* フラグ・メッセージ

メッセージ ID	メッセージ	重大度	ANSI 拡張機能
M_FIPS_ARRAY	FIPS-FLAGGER 警告 :ANSI 拡張子配列型 (行 *line*)	情報メッセージ	配列。ANSI 標準に準拠する必要がある場合、一切の FIPS メッセージでこの機能を使用してはならない。
M_FIPS_DATAINIT	FIPS-FLAGGER 警告 :ANSI 拡張子データ初期化 (行 *line*)	情報メッセージ	データ初期化。
M_FIPS_HASHDEF	FIPS-FLAGGER 警告 :ANSI 拡張機能 "#DEFINE" *line*。	情報メッセージ	宣言セクション内で #define を使用している。
M_FIPS_LABEL	FIPS-FLAGGER 警告 :WHENEVER 句のラベルでの ANSI 拡張子 ':'	情報メッセージ	WHENEVER 句内でラベルに ":" を許可している。
M_FIPS_POINTER	FIPS-FLAGGER 警告 :ANSI 拡張子ポインタ型 (行 *line*)	情報メッセージ	POINTER 型。
M_FIPS_SQLDA	FIPS-FLAGGER 警告 :ANSI 拡張子 sqlda (行 *line*)	情報メッセージ	SQLDA 構造体。
M_FIPS_STMT	FIPS-FLAGGER 警告 :ANSI 拡張子文 (行 *line*)	情報メッセージ	この行にある文は拡張機能である。
M_FIPS_SYBTYPE	FIPS-FLAGGER 警告 :ANSI 拡張子 Sybase SQL 型 (行 *line*)	情報メッセージ	Sybase 固有のデータ型である。
M_FIPS_TYPE	FIPS-FLAGGER 警告 :ANSI 拡張子データ型 (行 *line*)	情報メッセージ	指定された構文は、ANSI 準拠ではない。
M_FIPS_TYPEDEF	FIPS-FLAGGER 警告 :ANSI 拡張機能 TYPEDEF *line*	情報メッセージ	TYPEDEF。
M_FIPS_VOID	FIPS-FLAGGER 警告 :ANSI 拡張子 VOID 型 (行 *line*)	情報メッセージ	VOID 型。

表 *A-6:* 内部エラー・メッセージ

メッセージ ID	メッセージ	重大度	対処法
M_ALC_MEMORY	メモリのブロックを割り当てできません。	致命的なエラー	システム・リソースを確認する。
M_FILE_STACK_OVFL	ファイル・スタック・オーバーフロー : 最大許容ネスティングは *value* です。	致命的なエラー	ネストした INCLUDE 文の処理中に、ファイル・スタックがオーバフローした。ネストの深さは、最大値 32 を超えてはならない。
M_INTERNAL_ERROR	致命的な内部エラーがファイル *file* の行 *line* で発生しました。引数不一致エラーです。サイベースにお問い合わせください。	致命的なエラー	内部エラー。Sybase の担当者に連絡する。

表 *A-7: Sybase/Client-Library* のメッセージ

メッセージ ID	メッセージ	重大度	対処法
M_COLMCNT	*bind variable count* のバインド・カウントと *result set* のカラム・カウントは矛盾します。	警告	戻りカラムの数が、バインド変数のデータ型や数とともに戻された結果のカラム数と異なっている。
M_COLVARLM	ホスト変数 *name* の長さ *value* が *value* のカラム長より小さいです。	警告	ホスト変数がフェッチされたカラムを保持できない。カラムの長さを確認し、それに従ってホスト変数の長さを調整する。
M_COLVARPS	ホスト変数 *name* の総桁数 *value* と小数点以下桁数 *value* がカラムの総桁数 *value* と小数点以下桁数 *value* とは異なります。	警告	ホスト変数の精度と位取りがフェッチまたは挿入されたカラムと異なる。位取りと精度を適合させる。
M_COLVARTM	Open Client が *value* から *value* にホスト変数 *value* の型を変換できません。	警告	無効なタイプ。Open Client がデフォルトで変換をしないときは、cs_convert を使用する。
M_CTMSG	Client Library メッセージ :*value*		
M_OCAPI	Open Client API の *value* を実行中にエラーが発生しました。エラー :*value*	警告	この警告が発生したコンテキストによっては、先へ進む前に修正が必要になることがある。
M_OPERSYS	オペレーティング・システム・エラー :Open Client API の実行中に *value* が発生しました。	警告	オペレーティング・システムでエラーが発生した。システム管理者に連絡する。
M_PRECLINE	行 *value* のクエリーをチェック中の警告	情報メッセージ	クエリに問題がないかどうかを調べる。
M_SYBSERV	Sybase サーバ エラー。サーバ：*value* メッセージ：name	警告	エラーが発生したサーバへ送った文の構文を確認する。SQL 文を処理するときにサーバ内ですべてのリソースが利用できることを確認する。

表 *A-8:* 実行時メッセージ

SQLCODE 値、SQLSTATE コード	メッセージ	重大度	対処法
-25001 ZZ000	リカバリできないエラーが発生しました。	致命的なエラー	Sybase 製品の保守契約を結んでいるサポート・センタに連絡する。
-25002 ZA000	内部エラーが発生しました。	致命的なエラー	Sybase 製品の保守契約を結んでいるサポート・センタに連絡する。
-25003 ZD000	予期しない CS_COMPUTE_RESULT を受け取りました。	重大なエラー	Embedded SQL が計算結果を取得できない。エラーを返さないようにクエリを書き換える。

SQLCODE 値、SQLSTATE コード	メッセージ	重大度	対処法
-25004 ZE000	予期しない CS_CURSOR_RESULT を受け取りました。	重大なエラー	CS_LIBRARY ルーチンから返された値が有効かどうかを確認する。詳細は、CS-Library のマニュアルを参照。
-25005 ZF000	予期しない CS_PARAM_RESULT を受け取りました。	重大なエラー	CS_LIBRARY ルーチンから返された値が有効かどうかを確認する。詳細は、CS-Library のマニュアルを参照。
-25006 ZG000	予期しない CS_ROW_RESULT を受け取りました。	重大なエラー	CS_LIBRARY ルーチンから返された値が有効かどうかを確認する。詳細は、CS-Library のマニュアルを参照。
-25007 ZB000	SQLCA、SQLCODE または SQLSTATE にメッセージが返されません。	情報メッセージ	情報メッセージ。対処不要。
-25008 ZC000	まだ接続が定義されていません。	重大なエラー	有効な connect 文を入力する。
-25009 ZH000	予想外の CS_STATUS_RESULT を受信しました。	重大なエラー	CS_LIBRARY ルーチンから返された値が有効かどうかを確認する。詳細は、CS-Library のマニュアルを参照。
-25010 ZI000	予期しない CS_DESCRIBE_RESULT を受け取りました。	重大なエラー	CS_LIBRARY ルーチンから返された値が有効かどうかを確認する。詳細は、CS-Library のマニュアルを参照。
-25011 22005	データ例外 -- 項目記述子の割り当てエラー - タイプ	重大なエラー	正しい記述子型を入力する。
-25012 ZJ000	メモリ割り当てに失敗しました。	重大なエラー	この作業を行うにはメモリ不足である。
-25013 ZK000	Adaptive Server Enterprise のバージョンは 10 以上でなければなりません。	重大なエラー	Adaptive Server Enterprise 10.0 以降がインストールされていることを確認する。Adaptive Server Enterprise 10.0 以降がない場合、インストール担当者は Sybase 製品の保守契約を結んでいるサポート・センタに連絡する。
-25014 22024	Data exception — unterminated C string.	重大なエラー	すべての C の文字列が null で終了していることを確認する。
-25015 ZL000	Error retrieving thread identification.	重大なエラー	内部エラーが発生した可能性がある。Sybase 製品の保守契約を結んでいるサポート・センタに連絡する。
-25016 ZM000	クライアント・ライブラリ初期化時のエラー	重大なエラー	*$SYBASE* ディレクトリの設定を確認する。

SQLCODE 値、 SQLSTATE コード	メッセージ	重大度	対処法
-25017 ZN000	mutex の取得時または解放時のエラー	重大なエラー	未使用。
-25018 08002	接続名が使用中です。	重大なエラー	使用しているプログラムの論理を確認する。オープンしている接続を再度オープンしていないか確認する。または、2 番目の接続に新しい名前を使用する。 **注意** “DEFAULT” 接続を 2 つ持つことはできない。

付 録　B

サイズの大きな text データと image データを処理するサンプル・コード

他のサンプルについて

この付録では、大きな text および image データを処理するホスト変数の使用方法を示す Embedded SQL のサンプル・プログラムについて説明します。Web 上の Technical Library の Technical Documents コレクションにもサンプル・プログラムがあります。Technical Library Web サイトにアクセスするには、support.sybase.com にアクセスし、[Support Services] タブから [Technical Documents] へのリンクを選択します。コレクションの中から、次のタイトルの TechNote を検索してください。

- 「Client-Library Sample Programs」
- 「Embedded SQL/C Sample Programs」
- 「Embedded SQL/COBOL Sample Programs」

text_image.sql

次のスクリプトを使用して、テーブル “text_tab” を作成します。次の項のサンプル・プログラムを実行するときにはこのテーブルを使用します。

```
use tempdb
go

if exists (select 1 from sysobjects
where name = 'text_tab' and type = 'U' )
drop table text_tab
go

create table text_tab (
text_col text null,
image_col image null)
go
```

text_image.cp

```
/* Program name:text_image.cp
**
** Description:Inserting text and image data using host
** variables of types CS_TEXT and CS_IMAGE.

** Notes:This is a new feature in 11.x which allows you to use
** host variables of type CS_TEXT and CS_IMAGE in insert
** or update statements to handle text or image data.You don't
** need to use to mixed-mode client-library programming or
** dynamic sql, which had a limit of 64 k bytes.
** The size of the text or image data that can now be sent is
** limited only by memory or the maximum size allowed for
** text and image data by the Adaptive Server
Enterprise.However,
** the larger the data being sent this way, the slower the
** performance.
**
** Script file:text_image.sql
**
** Notes:Make sure you compile the program using the '-y'
** precompiler flag.
**
*/

#include <stdio.h>
#include "sybsqlex.h"

/* Declare the SQLCA */
EXEC SQL INCLUDE sqlca;

/*
** Forward declarations of the error and message handlers and
** other subroutines called from main().
*/
void    error_handler();
void    warning_handler();

int main()
{
int i=0;

EXEC SQL BEGIN DECLARE SECTION;
/* storage for login name and password */
    CS_CHAR        username[30], password[30];
    CS_TEXT        text_var[10000];
    CS_IMAGE        image_var[10000];
EXEC SQL END DECLARE SECTION;

    EXEC SQL WHENEVER SQLERROR CALL error_handler();
```

```
    EXEC SQL WHENEVER SQLWARNING CALL warning_handler();
    EXEC SQL WHENEVER NOT FOUND CONTINUE;

/*
** Copy the user name and password defined in sybsqlex.h to
** the variables declared for them in the declare section.
*/
strcpy(username, USER);
strcpy(password, PASSWORD);

/* Connect to the server and specify the database to use */
EXEC SQL CONNECT :username IDENTIFIED BY :password;

EXEC SQL USE tempdb;

/* Put something interesting in the variables. */
for (i=0; i< 10000; i++ )
    {
    text_var[i]  = 'a';
    image_var[i] = '@';
    }

EXEC SQL INSERT text_tab VALUES(:text_var, :image_var);
if (sqlca.sqlcode == 0)
    {
    printf("Row successfully inserted!¥n");
    EXEC SQL COMMIT WORK ;
    }

EXEC SQL DISCONNECT ALL;
exit(0);
}

/*
** void error_handler()
**
** Displays error codes and numbers from the SQLCA and exits
with
** an ERREXIT status.
*/
void error_handler()
{
fprintf(stderr, "¥n** SQLCODE=(%d)", sqlca.sqlcode);

if (sqlca.sqlerrm.sqlerrml)
    {
    fprintf(stderr, "¥n** Error Message: ");
    fprintf(stderr, "¥n** %s", sqlca.sqlerrm.sqlerrmc);
    }

fprintf(stderr, "¥n¥n");
```

```
exit(ERREXIT);
}

/*
** void warning_handler()
**
** Displays warning messages.
*/
void warning_handler()
{

if (sqlca.sqlwarn[1] == 'W')
    {
    fprintf(stderr,
    "¥n** Data truncated.¥n");
    }

if (sqlca.sqlwarn[3] == 'W')
    {
    fprintf(stderr,
    "¥n** Insufficient host variables to store results.¥n");
    }
return;
}
```

用語解説

Adaptive Server Enterprise
Sybase のクライアント／サーバ・アーキテクチャにおけるサーバ。Adaptive Server は、複数のデータベースと複数のユーザを管理します。ディスク上にあるデータの実際のロケーションを監視し、論理データ記述から物理データ記憶領域へのマッピングを管理します。メモリ内のデータ・キャッシュとプロシージャ・キャッシュの保守も行います。

Client-Library
Open Client の一部で、クライアント・アプリケーションを記述するためのルーチンの集まり。Client-Library は、Sybase 製品ラインのカーソルや高度な機能を取り込んでいます。

CS-Library
Client-Library と Server-Library のアプリケーションの両方で役立つユーティリティ・ルーチンの集まり。Open Client および Open Server の両方に含まれています。

DB-Library
Open Client の一部で、クライアント・アプリケーションを記述するためのルーチンの集まり。

FIPS
Federal Information Processing Standards (連邦情報処理標準) の略。FIPS フラグが有効なとき、Adaptive Server Enterprise および Embedded SQL プリコンパイラは、標準でない拡張された SQL 文を検出すると警告を発行します。

interfaces ファイル (interfaces file)
サーバ名をトランスポート・アドレスにマップするファイル。クライアント・アプリケーションが、サーバに接続するために ct_connect または dbopen を呼び出すと、Client-Library または DB-Library が interfaces ファイルからサーバのアドレスを検索します。ただし、すべてのプラットフォームが interfaces ファイルを使うわけではありません。interfaces ファイルを使わないプラットフォームでは、別のメカニズムでクライアントにサーバ・アドレスを知らせます。

isql スクリプト・ファイル (isql script file)
Embedded SQL において、プリコンパイラが生成できる 3 つのファイルのうちの 1 つ。isql スクリプト・ファイルには、プリコンパイラが生成したストアド・プロシージャが含まれます。ストアド・プロシージャは、Transact-SQL で記述されます。

mutex
相互排他セマフォ。Open Server アプリケーションが、共有オブジェクトへ排他アクセスをするために使用する論理オブジェクトのことです。

NULL
NULL は、明示的に割り当てられた値を持ちません。また、0 でもブランクでもありません。NULL の値は、他の値と比べて大きいとも、小さいとも、同じであるともみなされません。他の NULL と比べるときも同じです。

Open Server
カスタム・サーバを作成するためのツールとインタフェースを提供する Sybase 製品。

Open Server アプリケーション (Open Server application)

Open Server で構築されたカスタム・サーバ。

Server-Library

Open Server アプリケーションの記述に使用するルーチンの集まり。

SQLCA

1. Embedded SQL アプリケーションにおいて、Adaptive Server Enterprise とアプリケーション・プログラム間の通信パスを提供する構造体。Adaptive Server Enterprise は、各 SQL 文を実行したあと、SQLCA 内のリターン・コードを格納します。

2. Client-Library アプリケーションにおいて、アプリケーションが、Client-Library およびサーバのエラー・メッセージと情報メッセージを取得するのに使用する構造体。

SQLCODE

1. Embedded SQL アプリケーションにおいて、Adaptive Server Enterprise とアプリケーション・プログラム間の通信パスを提供する構造体。Adaptive Server Enterprise は、各 SQL 文を実行したあと、SQLCODE 内のリターン・コードを格納します。SQLCODE は、独立して存在することも、SQLCA 構造体内の変数として存在することもできます。

2. Client-Library アプリケーションにおいて、アプリケーションが、Client-Library およびサーバのエラー・メッセージと情報メッセージのコードを取得するのに使用する構造体。

TDS

Sybase のクライアントとサーバが通信に使用するアプリケーション・レベルのプロトコル。TDS は、コマンドと結果を記述します。

Transact-SQL

データベース言語 SQL の機能拡張バージョン。アプリケーションは、Transact-SQL を使用して、Adaptive Server Enterprise と通信できます。

イベント・ハンドラ (event handler)

Open Server におけるイベントを処理するルーチン。Open Server アプリケーションは、Open Server が提供するデフォルト・ハンドラを使用することができます。また、カスタマイズしたイベント・ハンドラをインストールすることもできます。

イベント (event)

Open Server アプリケーションに何らかの動作を行うよう要求するオカレンス。クライアント・コマンドおよび Open Server アプリケーション・コードの特定のコマンドは、イベントをトリガできます。イベントが発生すると、Open Server は、サーバ・アプリケーション・コード内の適切なイベント処理ルーチン、または適切なデフォルト・イベント・ハンドラのどちらかを呼び出します。

インジケータ変数 (indicator variable)

他の変数の値やフェッチしたデータについての特別な条件を示す変数。

Embedded SQL ホスト変数とともに使用すると、インジケータ変数は、データベースの値が null である箇所を示します。

エラー・メッセージ (error message)

Open Client/Server 製品がエラー状態を検出したときに発行するメッセージ。

カーソル (cursor)
SQL 文に関連付けられた記号名。

Embedded SQL において、カーソルは複数ローのデータをホスト・プログラムに渡すデータ・セレクタです。このローの受け渡しは、一度に 1 つずつ行われます。

隠し構造体 (hidden structure)
内部が Open Client/Server プログラマに対して隠されている構造体。Open Client/Server プログラマは、隠し構造体の割り付け、操作、割り付け解除を行うために、Open Client/Server ルーチンを使用しなければなりません。隠し構造体には、CS_CONTEXT 構造体などがあります。

拡張トランザクション (extended transaction)
Embedded SQL における、複数の Embedded SQL 文からなるトランザクション。

キー (key)
ローをユニークに識別するロー・データのサブセット。キー・データは、オープンされたカーソル内の「現在のロー」をユニークに記述します。

キーワード (keyword)
Transact-SQL または Embedded SQL で排他的に利用するように予約されているワードまたはフレーズ。「予約語」ともいいます。

機能 (capabilities)
クライアント／サーバ接続で許可されるクライアントの要求とサーバの応答を決定します。

クエリ (query)
1. データの検索要求。通常は select 文です。

2. データを操作する任意の SQL 文。

クライアント (client)
クライアント／サーバ・システムにおいて、サーバへ要求を送り、この要求に対する結果に対して処理を行う部分。

ゲートウェイ (gateway)
直接通信できないクライアントとサーバとの仲介として動作するアプリケーション。ゲートウェイ・アプリケーションは、クライアントとサーバの両方として動作します。クライアントからの要求をサーバに渡し、サーバからの結果をクライアントに返します。

結果変数 (result variable)
Embedded SQL において、select または fetch 文の結果を受け取る変数。

現在のロー (current row)
カーソルに関連して、カーソルが置かれているロー。フェッチは、カーソルに対して現在のローを取得します。

公開された構造体 (exposed structure)
内部が Open Client/Server プログラマに公開されている構造体。Open Client/Server プログラマは、公開された構造体の宣言、操作、割り付け解除を直接行うことができます。公開された構造体には、CS_DATAFMT 構造体などがあります。

コード・セット (code set)
「文字セット *(character set)*」を参照してください。

コールバック・イベント (callback event)
Open Client と Open Server におけるコールバック・ルーチンをトリガするイベント。

コールバック・ルーチン (callback routine)
コールバック・イベントと呼ばれるトリガ・イベントに応答して、Open Client または Open Server が呼び出すルーチン。

コマンド構造体 (command structure)
Client-Library アプリケーションがコマンドの送信と結果の処理に使用する Client-Library の隠し構造体 (CS_COMMAND)。

コマンド (command)
Client-Library において、コマンドは、ct_command、ct_dynamic、または ct_cursor に対するアプリケーションの呼び出しで始まり、ct_send に対するアプリケーションの呼び出しで終了するサーバ要求。

コンテキスト構造体 (context structure)
Client-Library または Open Server アプリケーション内でアプリケーション「コンテキスト」または操作環境を定義する CS-Library の隠し構造体 (CS_CONTEXT)。CS-Library ルーチンの cs_ctx_alloc と cs_ctx_drop は、それぞれコンテキスト構造体の割り付けと削除を行います。

サーバ (server)
クライアント／サーバ・システムにおけるクライアント要求を処理し、結果をクライアントに返す部分。

システム・プロシージャ (system procedure)
Adaptive Server Enterprise が、システム管理のために提供するストアド・プロシージャ。これらのプロシージャは、システム・テーブルからの情報の取得を簡単にし、データベース管理とシステム・テーブルの更新などを可能にします。

システム・レジスタード・プロシージャ (system registered procedures)
Open Server が、レジスタード・プロシージャのノーティフィケーション (通知) の管理とステータスの監視のために提供する内部レジスタード・プロシージャ。

システム管理者 (System Administrator)
Adaptive Server Enterprise システムにおいて、ユーザ・アカウントの作成、パーミッションの割り当て、新しいデータベースの作成などのシステム管理を担当するユーザ。Adaptive Server Enterprise では、システム管理者のログイン名は “sa” です。

システム記述子 (system descriptor)
Embedded SQL において、動的 SQL 文で使われる変数の記述を保持するメモリの領域。

出力変数 (output variable)
Embedded SQL において、ストアド・プロシージャからアプリケーション・プログラムにデータを渡す変数。

照合順 (collating sequence)
「ソート順 (sort order)」を参照してください。

ステータス変数 (status variable)
Embedded SQL において、ストアド・プロシージャのリターン・ステータス値を受け取ることによって、プロシージャの成功または失敗を示す変数。

ストアド・プロシージャ (stored procedure)
Adaptive Server Enterprise における、名前を付けて保管された SQL 文とオプションのフロー制御文の集まり。Adaptive Server Enterprise が提供するストアド・プロシージャは、「システム・プロシージャ (system procedure)」と呼ばれます。

スレッド (thread)
Open Server アプリケーションからライブラリ・コードまでの実行のパス。また、スタック領域、ステータス情報およびイベント・ハンドラに対応するパス。

接続構造体 (connection structure)
コンテキスト内にクライアント／サーバ接続を定義する Client-Library の隠し構造体 (CS_CONNECTION)。

ソート順 (sort order)
文字データをソートするときの順序の決定に使用されます。「照合順」とも呼ばれます。

ターゲット・ファイル (target file)
Embedded SQL において、プリコンパイラが生成できる 3 つのファイルのうちの 1 つ。ターゲット・ファイルは元の入力ファイルと似ていますが、すべての SQL 文が Client-Library の関数呼び出しに変換されています。

データ型 (datatype)
変数に有効な値と演算を表す定義属性。

データベース (database)
特別な目的のために組織化された、関連するデータ・テーブルとその他のデータベース・オブジェクトの集まり。

デッドロック (deadlock)
データにロックを保持している 2 人のユーザが、互いに他方のデータのロックを獲得しようとしたときに起こる状態。Adaptive Server Enterprise がデッドロックを検出すると、片方のユーザのプロセスを強制終了することによってこの状態を解決します。

デフォルト・データベース (default database)
ユーザがデータベース・サーバにログインしたとき、デフォルトで指定されるデータベース。

デフォルト言語 (default language)
1. アプリケーションに対して明示的にローカライゼーションの指定を行わないとき、Open Client/Server 製品が使用する言語。デフォルト言語は、ロケール・ファイルの “default” エントリにより決定されます。

2. ユーザが明示的に言語を選択しなかったとき、Adaptive Server Enterprise がメッセージとプロンプトに使用する言語。

デフォルト (default)
明示的に何も指定されなかったときに、Open Client/Server 製品が使用する値、オプション、または動作。

動的 SQL (Dynamic SQL)
SQL の一種であり、Embedded SQL または Client-Library アプリケーションが、実行時に値の決まる変数を含む SQL 文を実行できます。

トランザクション・モード (transaction mode)
Adaptive Server Enterprise がトランザクションを管理する方法。Adaptive Server Enterprise は、2 つのトランザクション・モードをサポートしています。Transact-SQL モード (「非連鎖トランザクション」とも呼ばれる) と ANSI モード (「連鎖トランザクション」とも呼ばれる) です。

トランザクション (transaction)
バックアップまたはリカバリのために 1 つの単位として扱われる、1 つ以上のサーバ・コマンド。トランザクション内のコマンドは、1 つのグループとしてコミットされます。したがって、すべてのコマンドがコミットされるか、すべてロールバックされるかのどちらかとなります。

入力変数 (input variable)
情報をルーチン、ストアド・プロシージャ、または Adaptive Server Enterprise に渡すときに使う変数。

配列バインド (array binding)
結果カラムを配列変数にバインドする処理。フェッチのときは、複数のロー分のカラムが変数にコピーされます。

配列 (array)
複数の同じタイプの変数からなる構造体。各変数は、個々にアドレッシングされます。

パススルー・モード (passthrough mode)
クライアントとリモート・データ・ソース間の TDS (Tabular Data Stream™) パケットの内容はアンパックされません。

バッチ (batch)
コマンドまたは文の集まり。

Client-Library のコマンド・バッチは、ct_send へのアプリケーションの呼び出しで終了する、1 つ以上の Client-Library のコマンドです。たとえば、アプリケーションは、カーソルに対する宣言、ローの選択、オープンを実行する複数のコマンドをまとめてバッチ処理できます。

Transact-SQL 文バッチは、1 つの Client-Library コマンドまたは Embedded SQL 文によって Adaptive Server Enterprise に送信される 1 つ以上の Transact-SQL 文です。

パラメータ (parameter)
1. データをルーチンに渡すとき、およびルーチンからデータを取得するときに使用する変数。

2. ストアド・プロシージャへの引数。

バルク・コピー (bulk copy)
データベースからデータをコピーしたり、データベースへデータをコピーしたりするコピー・ユーティリティ。bcp とも呼ばれます。

ブラウズ・モード (browse mode)
DB-Library™ と Client-Library アプリケーションが、一度に 1 つのローの値を更新しながらデータベース・ローをブラウズする方法。同様の機能を果たすカーソルの方が、一般に扱いやすくなっています。

プロパティ (property)
構造体に格納される名前付きの値。コンテキスト構造体、接続構造体、スレッド構造体、コマンド構造体は、プロパティを持ちます。構造体のプロパティは、構造体の動作を決めます。

文 (statement)
Transact-SQL または Embedded SQL におけるキーワードで始まる命令。キーワード名は、基本オペレーションまたは実行するコマンドを表します。

変換 (conversion)
「文字セット変換 (character set conversion)」を参照してください。

ホスト・プログラム (host program)
Embedded SQL における、Embedded SQL コードを含むアプリケーション・プログラム。

ホスト言語 (host language)
アプリケーションを記述するときに使われるプログラミング言語。

ホスト変数 (host variable)
Embedded SQL における、Adaptive Server Enterprise とアプリケーション・プログラム間のデータ転送を可能にする変数。「インジケータ変数 (indicator variable)」「入力変数 (input variable)」「出力変数 (output variable)」「結果変数 (result variable)」「ステータス変数 (status variable)」を参照してください。

マルチバイト文字セット (multi-byte character set)
複数のバイトを使用してコード化された文字を含む文字セット。マルチバイト文字セットには、EUC JIS、シフト JIS などがあります。

メッセージ・キュー (message queue)

Open Server において、スレッドが通信するとき使用するメッセージ・ポインタのリンク・リスト。スレッドは、キューにメッセージを書き込んだり、キューからメッセージを読み込んだりすることができます。

メッセージ番号 (message number)

エラー・メッセージをユニークに識別する番号。

文字セット変換 (character set conversion)

サーバへ入出力するときの文字セットのコード化スキームの変換。サーバとクライアントが異なる文字セットを使って通信するとき、変換が行われます。たとえば、Adaptive Server Enterprise が ISO 8859-1 を使用し、クライアントが Code Page 850 を使用する場合、文字セット変換をオンにして、サーバとクライアントが、受け渡しされるデータを同じように解釈するようにします。

文字セット (character set)

各文字をユニークに定義するコード化スキームを持つ特定の (通常、標準化された) 文字の集まり。ASCII と ISO 8859-1 (Latin 1) は、よく使用される文字セットです。

ユーザ名 (user name)

「ログイン名 (login name)」を参照してください。

リスティング・ファイル (listing file)

Embedded SQL において、プリコンパイラが生成できる 3 つのファイルのうちの 1 つ。リスティング・ファイルには、入力ファイルのソース文と、情報、警告、エラーなどのメッセージが含まれます。

リモート・プロシージャ・コール (RPC：remote procedure call)

1. クライアント・アプリケーションが Adaptive Server Enterprise ストアド・プロシージャを実行する 2 つの方法のうちの 1 つ (もう 1 つの方法では、Transact-SQL の execute 文を使用します)。Client-Library のアプリケーションは、ct_command を呼び出すことによって、リモート・プロシージャ・コール・コマンドを開始します。DB-Library アプリケーションは、dbrpcinit を呼び出すことによって、リモート・プロシージャ・コール・コマンドを開始します。

2. クライアントが Open Server アプリケーションを使って利用できる要求のタイプの 1 つ。これに応答して Open Server は、対応するレジスタード・プロシージャを実行するか、または Open Server アプリケーションの RPC イベント・ハンドラを呼び出します。

3. ユーザが接続しているサーバと異なるサーバで実行されるストアド・プロシージャ。

レジスタード・プロシージャ (registered procedure)

Open Server において、名前を付けて保管される C で記述された文の集まり。Open Server が提供するレジスタード・プロシージャは、「システム・レジスタード・プロシージャ (system registered procedure)」と呼ばれます。

ローカライゼーション (localization)

アプリケーションを特定の言語環境で使用するために設定する処理のこと。ローカライズされたアプリケーションは、通常、各国の言語と文字セットでメッセージを作成し、その国の日時表記フォーマットを使用します。

ログイン名 (login name)

ユーザが、サーバにログインするとき使用する名前。Adaptive Server Enterprise のログイン名が有効になるのは、Adaptive Server Enterprise がシステム・テーブル syslogins にそのユーザのエントリを持つ場合です。

ロケール・ファイル (locales file)

ロケール名を言語と文字セットのペアにマッピングするファイル。Open Client/Server 製品は、ローカライゼーション情報をロードするときにこのロケール・ファイルを調べます。

ロケール構造体 (locale structure)

Client-Library または Open Server アプリケーションのためのカスタム・ローカライゼーション値を定義する CS-Library の隠し構造体 (CS_LOCALE)。アプリケーションは、CS_LOCALE を使用して、使用される言語、文字セット、日付順、ソート順を定義できます。ロケール構造体の割り付けと割り付け解除には、CS-Library ルーチン cs_loc_alloc および cs_loc_drop を使用します。

ロケール名 (locale name)

言語と文字セットのペアを表す文字列。ロケール名は、「ロケール・ファイル」にリストされています。Sybase があらかじめ定義しているロケール名の他に、システム管理者が別のロケール名を定義し、ロケール・ファイルに追加することもできます。

索引

記号

数字

A

B

C

D

E

F

G

I

N

O

P

R

S

T

U

W

あ

い

え

お

か

き

け

こ

さ

し

す

せ

そ

た

て

と

な

に

ね

は

ふ

へ

ほ

ま

む

め

も

よ

ら

り

る

ろ