SYBASE®

Client-Library/C プログラマーズ・ガイド

Open Client™

15.5

ドキュメント ID：DC35395-01-1550-01
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 ..... **ix**

第 1 章 **Client-Library を使用する前に** **1**
Client-Library について 1
Client-Library アプリケーションのタイプ 2
Adaptive Server Enterprise クライアント・アプリケーション 2
Open Server クライアントまたはゲートウェイ・アプリケーション 4
簡単なサンプル・プログラム 4
プログラムの構築 5
サンプル・プログラムの手順 5
ソースのリスト 6
手順 1：Client-Library プログラミング環境を設定する 17
ヘッダ・ファイル 17
コンテキスト構造体の割り付け 17
Setting CS-Library コンテキスト・プロパティの設定 18
Client-Library の初期化 19
Client-Library コンテキスト・プロパティの設定 19
外部設定 20
手順 2：エラー処理を定義する 20
手順 3：サーバに接続する 21
接続構造体の割り付け 21
接続構造体プロパティの設定 22
サーバへのログイン 23
手順 4：サーバにコマンドを送信する 23
コマンド構造体の割り付け 23
コマンド構造体プロパティの設定 24
コマンドの実行 24
手順 5：コマンドの結果を処理する 24
手順 6：終了処理を行う 26
コマンド構造体の割り付け解除 26
接続のクローズと割り付け解除 26
Client-Library の終了 27
コンテキスト構造体の割り付け解除 27

第 2 章 構造体、定数、規則の説明 29
隠し構造体 29
CS_CONTEXT 30
CS_CONNECTION 30
CS_COMMAND 31
制御構造体の階層 31
接続規則とコマンド規則 31
CS_LOGINFO 32
CS_DS_OBJECT 32
CS_BLKDESC 32
CS_LOCALE 33
公開された構造体 33
CS_BROWSEDESC 34
CS_CLIENTMSG 34
CS_DATAFMT 34
CS_DATEREC 35
CS_IODESC 35
CS_PROP_SSL_LOCALID 35
CS_SERVERMSG 36
SQLCA、SQLCODE、および SQLSTATE 36
SQLDA 36
定数 36
型定数 37
フォーマット定数 37
その他の記号定数 38
表記規則 38
NULL と未使用パラメータ 38
入力パラメータ文字列 39
出力パラメータ文字列 39
基本構造体へのポインタ 40
項目番号 40
action、buffer、buflen、outlen 41

第 3 章 Open Client と Open Server のデータ型の使い方 43
データ型と型定数 43
データ型はどこで宣言するか 43
Open Client データ型と Open Server データ型を使用する理由 44
unichar データ型 44
unitext データ型 47
xml データ型 48
型定数とは 49
データ型の概要 50
binary 型 51
bit 型 52
character 型 52
datetime 型 53

numeric 型 55
money 型 55
text 型および image 型 56
null 代入値 57
Open Client のユーザ定義データ型 58

第 4 章 エラーおよびメッセージの処理 59
メッセージについて 59
メッセージを識別する方法 59
2 つのメッセージ処理方法 60
コールバック・ルーチンによるメッセージ処理 61
クライアント・メッセージ・コールバックの定義 62
サーバ・メッセージ・コールバックの定義 63
コールバックのインストール 64
メッセージのインライン処理 64
CS_EXTRA_INF プロパティ 65
CS_DIAG_TIMEOUT_FAIL プロパティ 65
長いメッセージの連続化 66
拡張エラー・データ 67
拡張エラー・データの用途 67
サーバ・トランザクション・ステータス 68

第 5 章 コマンド・タイプの選択 69
コマンドについて 69
コマンド・タイプ 70
コマンドの実行 70
コマンドの起動 71
コマンドのパラメータの定義 71
結果の処理 71
コマンドの再送信 72
言語コマンド 72
言語コマンドの構築 72
言語コマンドの結果処理 73
言語コマンドの使用が適する場合 74
言語コマンドの使用が適さない場合 74
RPC コマンド 74
RPC コマンドの構築 75
RPC コマンド結果の処理 76
RPC コマンドの使用が適する場合 78
RPC と execute 言語コマンドの比較 79
Client-Library カーソル・コマンド 79
Client-Library カーソル・コマンドの構築 80
Client-Library カーソルの使用が適する場合 80
Client-Library カーソルの使用が適さない場合 80

動的 SQL コマンド 80
動的 SQL コマンドの構築 81
動的 SQL コマンドの使用が適する場合 81
動的 SQL の使用が適さない場合 81
メッセージ・コマンド 81
メッセージ・コマンドの使用が適する場合 82
メッセージ・コマンドの使用が適さない場合 82
パッケージ・コマンド 83
データ送信コマンド 83
データ送信コマンドの使用が適する場合 83
データ送信コマンドの使用が適さない場合 83

第 6 章 結果処理コードの書き方 85
結果のタイプ 85
基本ループの構造 86
通常ロー結果の処理 88
カーソル結果の処理 89
スクロール可能なカーソルの結果の処理 91
パラメータ結果の処理 92
リターン・ステータス結果の処理 93
計算結果の処理 94
メッセージ結果の処理 96
記述結果の処理 96
フォーマット結果の処理 97
コマンド・ステータスを表示する result_type の値 98
論理コマンド 99
ct_results の最後のリターン・コード 99

第 7 章 Client-Library カーソルの使い方 101
カーソルの概要 101
言語カーソルと Client-Library カーソルの比較 102
言語カーソル 103
Client-Library カーソル 103
Client-Library カーソルの使用が適する場合 104
Client-Library カーソルの利点 104
Client-Library カーソルを使用する場合のパフォーマンス問題 106
Client-Library カーソルの使い方 106
手順 1：カーソルを宣言する 108
手順 2：カーソル・ロー数を設定する 113
手順 3：カーソルをオープンする 115
手順 4：カーソル・ローを処理する 116
手順 5：カーソルをクローズする 119
手順 6：カーソルの割り付けを解除する 119
Client-Library のカーソル・プロパティ 120

第 8 章 動的 SQL コマンドの使い方 121
動的 SQL の概要 121
動的 SQL の利点 122
動的 SQL の制限事項 122
動的 SQL コマンドのパフォーマンス 123
Adaptive Server Enterprise の制限事項とデータベースの稼働条件 123
動的 SQL に代わる方法 124
即時実行方式の使い方 124
即時実行方式の使用が適する場合 124
即時実行コマンドのコーディング 125
準備実行方式の使い方 125
準備実行方式の使用が適する場合 125
準備実行方式のプログラム構造 126
手順 1：文を準備する 127
手順 2：コマンド入力の記述を取得する 128
手順 3：コマンド出力の記述を取得する 129
手順 4：準備文を実行する 130
手順 5：準備文の割り付けを解除する 131
動的 SQL とストアド・プロシージャの比較 131

第 9 章 ディレクトリ・サービスの使い方 133
ディレクトリ・サービスとは 133
アプリケーションはディレクトリ・サービスをどのように使用するか 134
ディレクトリの検索 134
サンプル・プログラム 134
プログラム構造 134
手順 1：検索を開始する 135
データ構造体の初期化 135
ディレクトリ・サービス・プロパティの設定 136
ディレクトリ・コールバックのインストール 137
ct_ds_lookup の呼び出し 137
ディレクトリ検索を開始するサンプル・プログラム 137
手順 2：ディレクトリ・コールバックで検索結果を収集する 140
ディレクトリ・コールバックの定義 140
ディレクトリ・コールバックの例 142
手順 3：ディレクトリ・オブジェクトを検査する 143
属性データ構造体 145
ディレクトリ・オブジェクトを検査するサンプル・プログラム 146
手順 4：クリーンアップする 157

付録 A　呼び出しの論理シーケンス 159
Client-Library のステータス・マシン 159
呼び出しのコマンド・レベル・シーケンス 160
コマンド・ステータス・テーブル 160
起動されたコマンド・ステータス・テーブル 160
結果タイプ・ステータス・テーブル 161
まとめ 161
コマンド・ステータス 162
コマンド・レベル・ルーチン 163
各コマンド・ステータスで呼び出し可能なルーチン 164
起動されたコマンド 174
起動されたコマンド・ルーチン 176
起動されたコマンドで呼び出し可能なルーチン 176
結果タイプ 178
結果タイプ処理ルーチン 180
各結果タイプで呼び出し可能なルーチン 180
保留中の結果 183

索引 185

はじめに

このマニュアルでは、Open Client™ Client-Library を使用して C 言語のアプリケーションを作成する方法について説明します。

対象読者

このマニュアルは、C プログラミング言語に精通したアプリケーション・プログラマを対象としています。

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 Client-Library を使用する前に」では、基本的な Client-Library プログラムの構築方法について説明します。簡単かつ完全な Client-Library アプリケーションを示します。
- 「第 2 章 構造体、定数、規則の説明」では、Client-Library の構造体、定数、パラメータの規則について説明します。
- 「第 3 章 Open Client と Open Server のデータ型の使い方」では、Client-Library アプリケーションで使用できるデータ型を説明します。
- 「第 4 章 エラーおよびメッセージの処理」では、ユーザのアプリケーションにおいて、Client-Library とサーバのエラーを処理する方法について説明します。
- 「第 5 章 コマンド・タイプの選択」では、個々のコマンド・タイプをアプリケーションで使用する方法、およびいつ使用するかについて説明します。
- 「第 6 章 結果処理コードの書き方」では、Client-Library の結果処理モデルについて説明します。
- 「第 7 章 Client-Library カーソルの使い方」では、Client-Library カーソルを宣言および操作する方法について説明します。
- 「第 8 章 動的 SQL コマンドの使い方」では、ユーザ・アプリケーションにおける動的 SQL クエリの使用方法について説明します。
- 「第 9 章 ディレクトリ・サービスの使い方」では、Client-Library ディレクトリ・サービスを使用する方法について説明します。
- 「付録 A 呼び出しの論理シーケンス」では、Client-Library アプリケーションでの有効な呼び出しシーケンスの構成図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、次のプラットフォームで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 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/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 (http://www.sybase.com/support) を指定します。

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

Client-Library を使用する前に

この章では、Client-Library/C アプリケーションを開発するのに必要な基本的な概念を説明します。

トピック名	ページ
Client-Library について	1
Client-Library アプリケーションのタイプ	2
簡単なサンプル・プログラム	4
手順 1：Client-Library プログラミング環境を設定する	17
手順 2：エラー処理を定義する	20
手順 3：サーバに接続する	21
手順 4：サーバにコマンドを送信する	23
手順 5：コマンドの結果を処理する	24
手順 6：終了処理を行う	26

Client-Library について

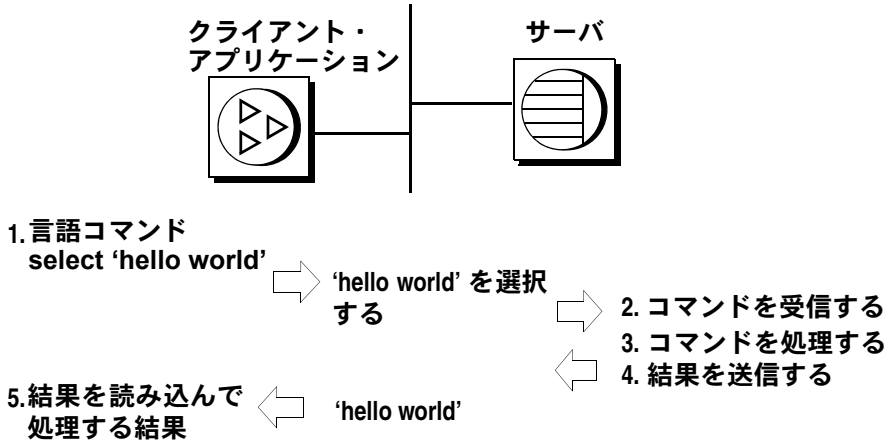
Client-Library は、Sybase サーバにコマンドを送信し、Sybase サーバから結果を取得するルーチンの集まりです。

Sybase のクライアント／サーバ・アーキテクチャと Sybase 製品の概要については、『Open Client Client-Library/C リファレンス・マニュアル』の「第 1 章 Client-Library の概要」を参照してください。

Client-Library アプリケーションのタイプ

Client-Library アプリケーションは、主として、送信するコマンドのタイプによってさまざまな種類があります。クライアント・アプリケーションが、いったんサーバに接続されると、図 1-1 に示すような「コマンド送信 / 結果処理」方式を使用します。

図 *1-1:* コマンド送信 / 結果処理方式



Adaptive Server Enterprise クライアント・アプリケーション

次にいくつかの例を挙げて、Adaptive Server Enterprise クライアント・アプリケーションがどのようなタスクを実行するかを説明します。

- SQL インタプリタ – クライアント・アプリケーションは、クエリの入力を指示し、入力されたクエリを言語コマンドとしてサーバに送信し、Adaptive Server Enterprise から結果を取得して、その結果を表示します。Sybase isql ユーティリティはこのような処理を行うアプリケーションです。このタイプのアプリケーションは、次の Client-Library ルーチンを呼び出します。
  - 言語コマンドとそのテキストを定義する ct_command(CS_LANG_CMD)
  - コマンドをサーバに送信する ct_send
  - 結果を読み込む ct_results
  - カラム・フォーマットを探す ct_res_info と ct_describe
  - ローを取得する ct_bind と ct_fetch

「言語コマンド」(72 ページ) を参照してください。また、「簡単なサンプル・プログラム」(4 ページ) に示すサンプル・アプリケーションも合わせて参照してください。

- データ入力 － 常に同じクエリを実行するアプリケーションです。このタイプのアプリケーションは、挿入、更新、メニュー移植を実行するアプリケーション論理を実装するのに、Adaptive Server Enterprise ストアド・プロシージャを使用します。クライアント・プログラムは、RPC コマンドを送信して、これらのストアド・プロシージャを呼び出します。このタイプのアプリケーションは次のコマンドを呼び出します。
  - RPC コマンドを定義する ct_command(CS_RPC_CMD)
  - プロシージャ呼び出しに使用するパラメータ値を定義する ct_param または ct_setparam
  - コマンドをサーバに送信する ct_send
  - 結果を読み込む ct_results、ct_bind、ct_fetch など

  「RPC コマンド」(74 ページ) を参照してください。
- 対話型クエリ・サンプル － 実行時に入力する値として、疑問符 (?) で指示したマーカを含むことができるクエリの入力を指示するアプリケーションです。このタイプのアプリケーションは、動的 SQL コマンドを使用して、次のことを行います。
  - ct_dynamic(CS_PREPARE) コマンドを送信し、結果を処理することによって、文を準備します。
  - ct_dynamic(CS_DESCRIBE_INPUT) コマンドを送信し、結果を処理することによって、パラメータ・フォーマットを問い合わせます。
  - 入力値の入力を指示してから、ct_dynamic(CS_EXECUTE) コマンドを送信し、結果を処理することによって、文を実行します。

  詳細については、「第 8 章 動的 SQL コマンドの使い方」を参照してください。

Open Server クライアントまたはゲートウェイ・アプリケーション

Open Server Server-Library は、カスタム・サーバ・アプリケーションを作成するときに使用するルーチンの集まりです。Server-Library ルーチンについては、『Open Server Server-Library/C リファレンス・マニュアル』を参照してください。

次にいくつかの例を挙げて、Open Server クライアント・アプリケーションが実行するタスクを説明します。

- カスタム Open Server アプリケーションのクライアント － クライアント・アプリケーションは、RPC コマンドを送信して、Open Server アプリケーション・プログラムに呼び出し可能なサーバ・プロシージャとして登録されているカスタム・サーバ・ルーチンを呼び出します。このような登録ルーチンをレジスタード・プロシージャといいますが、これについては、『Open Server Server-Library/C リファレンス・マニュアル』を参照してください。クライアント・アプリケーションが RPC コマンドをどのように送信するかについては、「RPC コマンド」(74 ページ) を参照してください。
- ノーティフィケーション・クライアント － Open Server には、クライアント・アプリケーションが選択したレジスタード・プロシージャの呼び出しを監視できるようにする「レジスタード・プロシージャ・ノーティフィケーション」という機能が用意されています。たとえば、重要なデータのコピーをキャッシュするクライアント・アプリケーションは、通常、そのデータを更新するレジスタード・プロシージャに関するノーティフィケーション (通知) を監視します。ノーティフィケーションは、キャッシュしたコピーをいつリフレッシュすべきかを知らせてくれます。『Open Client Client-Library/C リファレンス・マニュアル』の「レジスタード・プロシージャ」を参照してください。
- ゲートウェイ・アプリケーション － サーバ・アプリケーションは自身のクライアントと他のサーバとの間の仲介者として働きます。ゲートウェイはクライアント・コマンドを受け付けて、それをリモート・サーバに渡し、その結果を読み込んで自身のクライアントに渡します。リモート・サーバが Sybase サーバである場合、ゲートウェイは Client-Library 呼び出しを行って、リモート・サーバと交信します。

簡単なサンプル・プログラム

ここでは、サーバに接続し、クエリを送信し、結果を処理して終了するプログラムの例を示します。ほとんどの Client-Library アプリケーションは、この例と同じようなプログラム構造になります。

プログラムの構築

使用しているプラットホームで Client-Library アプリケーションを構築する方法と、必要なコンパイル・オプション、リンク・オプション、ライブラリ・ファイル名、ランタイム動作条件については、『Open Client/Server プログラマーズ・ガイド補足 Windows 版』または『UNIX 版』を参照してください。

サンプル・プログラムの手順

簡単な Client-Library アプリケーションは、次の 6 つの手順で構成されます。

1　Client-Library プログラミング環境を設定します。

   a　cs_ctx_alloc を使用して、コンテキスト構造体を割り付けます。

   b　cs_config を使用して、コンテキストの CS-Library プロパティを設定します。

   c　ct_init を使用して、Client-Library を初期化します。

   d　ct_config を使用して、コンテキストの Client-Library プロパティを設定します。

2　エラー処理を定義します。ほとんどのアプリケーションは、コールバック・ルーチンを使用して、エラーを処理します。

   a　cs_config(CS_MESSAGE_CB) を使用して、CS-Library エラー・コールバックをインストールします。

   b　ct_callback を使用して、クライアント・メッセージ・コールバックをインストールします。

   c　ct_callback を使用して、サーバ・メッセージ・コールバックをインストールします。

警告！ エラー処理を定義していないアプリケーションは、プログラム、ネットワーク、サーバで発生したエラーのノーティフィケーション (通知) を受信しません。エラーおよびサーバ・メッセージを処理するようにアプリケーションをコーディングしてください。エラー処理を行わないアプリケーションは、デバッグおよび保守が困難です。

3　サーバに接続します。

   a　ct_con_alloc を使用して、接続構造体を割り付けます。

   b　ct_con_props を使用して、接続構造体のプロパティを設定します。

   c　ct_connect を使用して、サーバとの接続をオープンします。

   d　ct_options を使用して、この接続のサーバ・オプションを設定します。

4 サーバに言語コマンドを送信します。

   a ct_cmd_alloc を使用して、コマンド構造体を割り付けます。

   b ct_command を使用して、言語コマンドを起動します。

   c ct_send を使用して、このコマンドを送信します。

5 コマンドの結果を処理します。

   a ct_results を使用して、(ループで呼び出した) 処理の結果を設定します。

   b ct_res_info を使用して、結果セットに関する情報を取得します。

   c ct_describe を使用して、結果項目に関する情報を取得します。

   d ct_bind を使用して、結果項目をプログラム・データ領域にバインドします。

   e ct_fetch を使用して、(ループで呼び出した) 結果ローをフェッチします。

6 終了処理を行います。

   a ct_cmd_drop を使用して、コマンド構造体の割り付けを解除します。

   b ct_close を使用して、サーバとの接続をクローズします。

   c ct_exit を使用して、Client-Library を終了します。

   d cs_ctx_drop を使用して、コンテキスト構造体の割り付けを解除します。

ソースのリスト

次のサンプル・プログラム *firstapp.c* は、上記で説明した手順のコードです。このコード例に続いて、各手順について説明します (「手順 1：Client-Library プログラミング環境を設定する」(17 ページ) 以降)。

このアプリケーションのソース・コードは、Client-Library のサンプル・プログラムの中に含まれています。サンプル・プログラムの作成方法と実行方法については、『Open Client/Server プログラマーズ・ガイド補足 Windows 版』または『Open Client/Server プログラマーズ・ガイド補足 UNIX 版』の「Client-Library」の章を参照してください。

```
/*
** Language Query Example Program.
*/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctpublic.h>
#include "example.h"
```

```
#define MAXCOLUMNS   2
#define MAXSTRING    40
#define ERR_CH stderr
#define OUT_CH stdout

/*
** Define a macro that exits if a function return code indicates
** failure.
*/
#define EXIT_ON_FAIL(context, ret, str) ¥
  if (ret != CS_SUCCEED) ¥
  { ¥
    fprintf(ERR_CH, "Fatal error:%s¥n", str); ¥
    if (context != (CS_CONTEXT *) NULL) ¥
    { ¥
      (CS_VOID) ct_exit(context, CS_FORCE_EXIT); ¥
      (CS_VOID) cs_ctx_drop(context); ¥
    } ¥
    exit(-1);¥
  }


/*
** Callback routines for library errors and server messages.
*/
CS_RETCODE CS_PUBLIC csmsg_callback PROTOTYPE((
        CS_CONTEXT     *context,
        CS_CLIENTMSG   *clientmsg ));
CS_RETCODE CS_PUBLIC clientmsg_callback PROTOTYPE((
        CS_CONTEXT     *context,
        CS_CONNECTION  *connection,
        CS_CLIENTMSG   *clientmsg ));
CS_RETCODE CS_PUBLIC servermsg_callback PROTOTYPE((
        CS_CONTEXT     *context,
        CS_CONNECTION  *connection,
        CS_CLIENTMSG   *servermsg ));


/*
** Main entry point for the program.
*/
int
main(int argc, char *argv[])
{
    CS_CONTEXT     *context;     /* Context structure     */
    CS_CONNECTION  *connection;  /* Connection structure. */
    CS_COMMAND     *cmd;         /* Command structure.    */

    /* Data format structures for column descriptions: */
    CS_DATAFMT     columns[MAXCOLUMNS];
```

```
CS_INT          datalength[MAXCOLUMNS];
CS_SMALLINT     indicator[MAXCOLUMNS];
CS_INT          count;
CS_RETCODE      ret;
CS_RETCODE      results_ret;
CS_INT          result_type;
CS_CHAR         name[MAXSTRING];
CS_CHAR         city[MAXSTRING];

EX_SCREEN_INIT();

/*
** Step 1:Initialize the application.
*/
```

詳細については、「手順 1：Client-Library プログラミング環境を設定する」(17 ページ) を参照してください。

```
/*
** First allocate a context structure.
*/
context = (CS_CONTEXT *) NULL;
ret = cs_ctx_alloc(EX_CTLIB_VERSION, &context);
EXIT_ON_FAIL(context, ret, "cs_ctx_alloc failed");

/*
** Initialize Client-Library.
*/
ret = ct_init(context, EX_CTLIB_VERSION);
EXIT_ON_FAIL(context, ret, "ct_init failed");


/*
** Step 2:Set up the error handling.Install callback handlers
** for:- CS-Library errors - Client-Library errors - Server
** messages
*/
```

詳細については、「手順 2：エラー処理を定義する」(20 ページ) を参照してください。

```
/*
** Install a callback function to handle CS-Library errors
*/
ret = cs_config(context, CS_SET, CS_MESSAGE_CB,
                (CS_VOID *)csmsg_callback,
                CS_UNUSED, NULL);
EXIT_ON_FAIL(context, ret,
             "cs_config(CS_MESSAGE_CB) failed");

/*
```

```
** Install a callback function to handle Client-Library errors
**
** The client message callback receives error or informational
** messages discovered by Client-Library.
*/
ret = ct_callback(context, NULL, CS_SET, CS_CLIENTMSG_CB,
                  (CS_VOID *) clientmsg_callback);
EXIT_ON_FAIL(context,ret,
            "ct_callback for client messages failed");

/*
** The server message callback receives server messages sent by
** the server.These are error or informational messages.
*/
ret = ct_callback(context, NULL, CS_SET, CS_SERVERMSG_CB,
                  (CS_VOID *) servermsg_callback);
EXIT_ON_FAIL(context, ret,
            "ct_callback for server messages failed");


/*
** Step 3:Connect to the server.We must:- Allocate a connection
** structure.- Set user name and password.- Create the
** connection.
*/
```

詳細については、「手順 3:サーバに接続する」(21 ページ) を参照してください。

```
/*
** First, allocate a connection structure.
*/
ret = ct_con_alloc(context, &connection);
EXIT_ON_FAIL(context, ret, "ct_con_alloc() failed");

/*
** These two calls set the user credentials (username and
** password) for opening the connection.
*/
ret = ct_con_props(connection, CS_SET, CS_USERNAME,
           Ex_username, CS_NULLTERM, NULL);
EXIT_ON_FAIL(context, ret, "Could not set user name");
ret = ct_con_props(connection, CS_SET, CS_PASSWORD,
           Ex_password, CS_NULLTERM, NULL);
EXIT_ON_FAIL(context, ret, "Could not set password");

/*
** Create the connection.
*/
if(EX_SERVER==NULL)
    ret = ct_connect(connection, (CS_CHAR *) NULL, 0);
else
    ret = ct_connect(connection, (CS_CHAR *)EX_SERVER, strlen(EX_SERVER));
EXIT_ON_FAIL(context, ret, "Could not connect!");
```

```
/*
** Step 4:Send a command to the server, as follows:- Allocate a
** CS_COMMAND structure - Build a command to be sent with
** ct_command.- Send the command with ct_send.
*/
```

詳細については、「手順 4：サーバにコマンドを送信する」(23 ページ) を参照してください。

```
/*
** Allocate a command structure.
*/
ret = ct_cmd_alloc(connection, &cmd);
EXIT_ON_FAIL(context, ret, "ct_cmd_alloc() failed");


/*
** Initiate a language command.This call associates a query with
** the command structure.
*/
ret = ct_command(cmd, CS_LANG_CMD,
         "select au_lname, city from pubs2..authors ¥
         where state = 'CA'",
         CS_NULLTERM, CS_UNUSED);
EXIT_ON_FAIL(context, ret, "ct_command() failed");

/*
** Send the command.
*/
ret = ct_send(cmd);
EXIT_ON_FAIL(context, ret, "ct_send() failed");

/*
** Step 5:Process the results of the command.
*/
```

詳細については、「手順 5：コマンドの結果を処理する」(24 ページ) を参照してください。

```
while ((results_ret = ct_results(cmd, &result_type))
       == CS_SUCCEED)
{

    /*
    ** ct_results sets result_type to indicate when data is
    ** available and to indicate command status codes.
    */
    switch ((int)result_type)
    {
    case CS_ROW_RESULT:

       /*
```

```
** This result_type value indicates that the rows
** returned by the query have arrived.We bind and
** fetch the rows.
**
** We're expecting exactly two character columns:
** Column 1 is au_lname, 2 is au_city.
**
** For each column, fill in the relevant fields in
** the column's data format structure, and bind
** the column.
*/
columns[0].datatype = CS_CHAR_TYPE;
columns[0].format = CS_FMT_NULLTERM;
columns[0].maxlength = MAXSTRING;
columns[0].count = 1;
columns[0].locale = NULL;
ret = ct_bind(cmd, 1, &columns[0],
             name, &datalength[0],
             &indicator[0]);
EXIT_ON_FAIL(context, ret,
  "ct_bind() for au_lname failed");

/*
** Same thing for the 'city' column.
*/
columns[1].datatype = CS_CHAR_TYPE;
columns[1].format = CS_FMT_NULLTERM;
columns[1].maxlength = MAXSTRING;
columns[1].count = 1;
columns[1].locale = NULL;

ret = ct_bind(cmd, 2, &columns[1], city,
             &datalength[1],
             &indicator[1]);
EXIT_ON_FAIL(context, ret,
  "ct_bind() for city failed");

/*
** Now fetch and print the rows.

*/
while(((ret = ct_fetch(cmd, CS_UNUSED, CS_UNUSED,
                       CS_UNUSED, &count))
       == CS_SUCCEED)
      || (ret == CS_ROW_FAIL))
{
    /*
    ** Check if we hit a recoverable error.
    */
    if( ret == CS_ROW_FAIL )
    {
        fprintf(ERR_CH,
```

```
                        "Error on row %ld.¥n",
                        (long)(count+1));
            }
            /*
            ** We have a row, let's print it.
            */
            fprintf(OUT_CH, "%s:%s¥n", name, city);
        }

        /*
        ** We're finished processing rows, so check
        ** ct_fetch's final return value to see if an
        ** error occurred.The final return code should be
        ** CS_END_DATA.
        */
        if ( ret == CS_END_DATA )
        {
            fprintf(OUT_CH,
                    "¥nAll done processing rows.¥n");
        }
        else /* Failure occurred. */
        {
            EXIT_ON_FAIL(context, CS_FAIL,
                        "ct_fetch failed");
        }

        /*
        ** All done with this result set.
        */
        break;

    case CS_CMD_SUCCEED:

        /*
        ** We executed a command that never returns rows.
        */
        fprintf(OUT_CH, "No rows returned.¥n");
        break;

      case CS_CMD_FAIL:

        /*
        ** The server encountered an error while
        ** processing our command.These errors will be
        ** displayed by the server-message callback that
        ** we installed earlier.
        */
        break;

      case CS_CMD_DONE:
```

```
        /*
        ** The logical command has been completely
        ** processed.
        */
        break;

    default:

        /*
        ** We got something unexpected.
        */
        EXIT_ON_FAIL(context, CS_FAIL,
           "ct_results returned unexpected result type");
        break;
    }
}

/*
** We've finished processing results.Check the return value
** of ct_results() to see if everything went okay.
*/
switch( (int) results_ret)
{
case CS_END_RESULTS:

        /*
        ** Everything went fine.
        */
        break;

    case CS_FAIL:

        /*
        ** Something terrible happened.
        */
        EXIT_ON_FAIL(context, CS_FAIL,
                    "ct_results() returned CS_FAIL.");
        break;

    default:

        /*
        ** We got an unexpected return value.
        */
        EXIT_ON_FAIL(context, CS_FAIL,
           "ct_results returned unexpected return code");
        break;
}

/*
** Step 6:Clean up and exit.
```

```
    */
```

詳細については、「手順 6：終了処理を行う」(26 ページ) を参照してください。

```
    /*
    ** Drop the command structure.
    */
    ret = ct_cmd_drop(cmd);
    EXIT_ON_FAIL(context, ret, "ct_cmd_drop failed");

    /*
    ** Close the connection and drop its control structure.
    */
    ret = ct_close(connection, CS_UNUSED);
    EXIT_ON_FAIL(context, ret, "ct_close failed");
    ret = ct_con_drop(connection);
    EXIT_ON_FAIL(context, ret, "ct_con_drop failed");

    /*
    ** ct_exit tells Client-Library that we are done.
    */
    ret = ct_exit(context, CS_UNUSED);
    EXIT_ON_FAIL(context, ret, "ct_exit failed");

    /*
    ** Drop the context structure.
    */
    cs_ctx_drop(context)
    EXIT_ON_FAIL(context, ret, "cs_ctx_drop failed");

    /*
    ** Normal exit to the operating system.
    */
    exit(0);
}


/*
** Handler for server messages.Client-Library will call this
** routine when it receives a message from the server.
*/
CS_RETCODE CS_PUBLIC
servermsg_callback(CS_CONTEXT *cp, CS_CONNECTION *chp, CS_SERVERMSG *msgp)
{

    /*
    ** Print the message info.
    */
    fprintf(ERR_CH,
        "Server message:¥n¥t");
    fprintf(ERR_CH,
        "number(%ld) severity(%ld) state(%ld) line(%ld)¥n",
        (long) msgp->msgnumber, (long) msgp->severity,
        (long) msgp->state, (long) msgp->line);
```

```
    /*
    ** Print the server name if one was supplied.
    */
    if (msgp->svrnlen > 0)
        fprintf(ERR_CH, "¥tServer name:%s¥n", msgp->svrname);

    /*
    ** Print the procedure name if one was supplied.
    */
    if (msgp->proclen > 0)
        fprintf(ERR_CH, "¥tProcedure name:%s¥n", msgp->proc);

    /*
    ** Print the null terminated message.
    */
    fprintf(ERR_CH, "¥t%s¥n", msgp->text);

    /*
    ** Server message callbacks must return CS_SUCCEED.
    */
    return(CS_SUCCEED);
}

/*
**  Client-Library error handler.This function will be invoked
**  when a Client-Library has detected an error.Before Client-
**  Library routines return CS_FAIL, this handler will be called
**  with additional error information.
*/
CS_RETCODE CS_PUBLIC
clientmsg_callback(CS_CONTEXT *context, CS_CONNECTION *conn, CS_CLIENTMSG
*emsgp)
{

     /*
     ** Error number:
     ** Print the error's severity, number, origin, and
     ** layer.These four numbers uniquely identify the error.
     */
     fprintf(ERR_CH,
             "Client Library error:¥n¥t");
     fprintf(ERR_CH,
        "severity(%ld) number(%ld) origin(%ld) layer(%ld)¥n",
        (long) CS_SEVERITY(emsgp->severity),
        (long) CS_NUMBER(emsgp->msgnumber),
        (long) CS_ORIGIN(emsgp->msgnumber),
        (long) CS_LAYER(emsgp->msgnumber));

     /*
     ** Error text:
     ** Print the error text.
     */
     fprintf(ERR_CH, "¥t%s¥n", emsgp->msgstring);
```

```
    /*
    ** Operating system error information:Some errors, such as
    ** network errors, may have an operating system error associated
    ** with them.If there was an operating system error, this code
    ** prints the error message text.
    */
    if (emsgp->osstringlen > 0)
    {
         fprintf(ERR_CH,
                 "Operating system error number(%ld):¥n",
                 (long) emsgp->osnumber);
         fprintf(ERR_CH, "¥t%s¥n", emsgp->osstring);
    }

    /*
    ** If we return CS_FAIL, Client-Library marks the connection as
    ** dead.This means that it cannot be used anymore.If we return
    ** CS_SUCCEED, the connection remains alive if it was not already
    ** dead.
    */
    return (CS_SUCCEED);
}


/*
**  CS-Library error handler.This function will be invoked
**  when CS-Library has detected an error.
*/
CS_RETCODE CS_PUBLIC
csmsg_callback(CS_CONTEXT *context, CS_CLIENTMSG *emsgp)
{
 /*
  ** Print the error number and message.
  */
  fprintf(ERR_CH,
          "CS-Library error:¥n");
  fprintf(ERR_CH,
           "¥tseverity(%ld) layer(%ld) origin(%ld) number(%ld)",
           (long) CS_SEVERITY(emsgp->msgnumber),
           (long) CS_LAYER(emsgp->msgnumber),
           (long) CS_ORIGIN(emsgp->msgnumber),
           (long) CS_NUMBER(emsgp->msgnumber));

  fprintf(ERR_CH, "¥t%s¥n", emsgp->msgstring);

  /*
  ** Print any operating system error information.
  */
  if(emsgp->osstringlen > 0)
  {
```

```
    fprintf(ERR_CH, "Operating System Error:%s¥n",
      emsgp->osstring);
  }

  return (CS_SUCCEED);
}
```

手順 1：Client-Library プログラミング環境を設定する

Client-Library プログラミング環境は次の項目で定義します。

- プログラミング・コンテキストを定義する CS_CONTEXT 構造体
- アプリケーションの ct_init 呼び出しで指示される Client-Library バージョン・レベル

ヘッダ・ファイル

すべての Client-Library/C アプリケーションに、ヘッダ・ファイル *ctpublic.h* が必要です。このファイルには、Client-Library ルーチンに必要な型定義と宣言が含まれています。

コンテキスト構造体の割り付け

Client-Library アプリケーションは、CS-Library ルーチン cs_ctx_alloc を呼び出して、コンテキスト構造体を割り付けます。コンテキスト構造体を割り付けてから、Client-Library を初期化してください。

注意　CS-Library ルーチン名はプレフィクス cs で始まり、Client-Library ルーチン名はプレフィクス ct で始まります。すべての Client-Library プログラムに、少なくとも 2 つの CS-Library 呼び出しが含まれています。これは、コンテキスト構造体の割り付けとその解除を行うためのルーチン呼び出しです。

Setting CS-Library コンテキスト・プロパティの設定

Client-Library アプリケーションは、コンテキスト構造体を割り付けた後、cs_config を呼び出して、コンテキスト構造体の CS-Library プロパティを設定できます。

コンテキスト・プロパティは、アプリケーションの動作の種々の局面をコンテキスト・レベルで定義したものです。サンプル・プログラムの *firstapp.c* では、cs_config を呼び出して、CS_MESSAGE_CB プロパティを設定します。このプロパティは、CS-Library メッセージ・コールバック・ルーチンを定義します。コールバック方式で CS-Library エラーを処理する場合には、アプリケーションにこのプロパティを設定してください。「第 4 章 エラーおよびメッセージの処理」を参照してください。

必要に応じて、他の CS-Library コンテキスト・プロパティも設定するように、アプリケーションをコーディングしてください。ほとんどのアプリケーションでは、CS_MESSAGE_CB の他に、次のプロパティを cs_config で設定します。

- CS_LOC_PROP − コンテキストのローカライゼーション情報を記述します。オペレーティング・システム環境に用意されているローカライゼーション情報とは異なるローカライゼーション情報をコンテキストが必要とする場合には、アプリケーションにこのプロパティを設定してください。たとえば、ドイツ語環境で実行されるアプリケーションでフランス語のコンテキストが必要な場合、cs_config を呼び出して、プロパティには CS_LOC_PROP プロパティを設定します。
- CS_EXTERNAL_CONFIG − ct_init が OCS ランタイム設定ファイルからデフォルト・アプリケーション・プロパティ設定値を読み込むかどうかを指定します。「外部設定」(20 ページ) を参照してください。
- CS_APP_NAME − アプリケーションの名前を指定します。外部設定が使用可能である (CS_EXTERNAL_CONFIG が CS_TRUE である) 場合、アプリケーション名には、設定ファイルのどのセクションから設定値を読み込むかを指定します。CS_APP_NAME は、割り付けられた CS_CONNECTION 構造体からも継承します。

『Open Client/Server Common Libraries リファレンス・マニュアル』の cs_config を参照してください。

Client-Library の初期化

アプリケーションは ct_init を呼び出して、Client-Library を初期化します。ct_init は、内部制御構造体を設定して、アプリケーションに必要な Client-Library 動作のバージョンを定義します。ct_init は、アプリケーション内で最初の Client-Library 呼び出しにする必要があります。

ほとんどのアプリケーションは ct_init を 1 回だけ呼び出します。ただし、アプリケーションが ct_init を複数回呼び出してもエラーにはなりません。複数モジュールのどれを最初に実行するかを保証できないアプリケーションもあるため、Client-Library は、複数の ct_init 呼び出しを許可しています。このタイプのアプリケーションはモジュールごとに ct_init を呼び出す必要があります。

ct_init は、アプリケーションが期待する Client-Library 動作のバージョンを記述している記号をパラメータとして持ちます。

Client-Library がこの動作を提供できない場合には、ct_init は CS_FAIL を返します。

Client-Library コンテキスト・プロパティの設定

firstapp.c は、ct_config を呼び出して、CS_MAX_CONNECT コンテキスト・プロパティを設定します。このプロパティには、コンテキストの最大接続数を指定します。

Client-Library コンテキスト・プロパティは、次の 2 つの用途のどちらかに使用します。

- コンテキストの動作を定義します。

  CS_MAX_CONNECT はこのカテゴリーの一例です。

- コンテキストから作成した接続のデフォルト・プロパティを定義します。

  CS_NETIO プロパティはこのカテゴリーの一例です。コンテキスト CS_NETIO プロパティが同期接続を指示する CS_SYNC_IO に設定されていると、コンテキスト内に割り付けられる接続構造体は同期になります。特定の接続の CS_NETIO の値を割り付けてから変更するには、ct_con_props を呼び出します。

Client-Library コンテキスト・プロパティの一覧表については、『Open Client Client-Library/C リファレンス・マニュアル』の「プロパティ」の項を参照してください。

マルチスレッドではないアプリケーションは、ct_config を呼び出して、プログラム実行中のいつでもコンテキスト・プロパティを変更できます。マルチスレッドのアプリケーションは、シングルスレッドのスタートアップ・コード内にコンテキスト・プロパティを設定するか、コンテキストとその子接続に対するすべてのアクセスをシングルスレッドに限定してください。『Open Client Client-Library/C リファレンス・マニュアル』の「マルチスレッド・プログラミング」を参照してください。

アプリケーションが ct_config を呼び出してコンテキスト・プロパティを変更しても、現行の接続のプロパティ値は変更されません。ct_config 呼び出し後に割り付けられた接続からは、新しいプロパティ値を採用することになります。

外部設定

外部設定機能を使用するように設定されているアプリケーションであれば、ハードコードされた ct_config 呼び出しでプロパティを設定する代わりに、プロパティ値の外部設定ができます。『Open Client Client-Library/C リファレンス・マニュアル』の「ランタイム設定ファイルの使い方」の項を参照してください。

手順 2：エラー処理を定義する

エラーは、インラインまたはコールバック機能によって処理できます。サンプル・プログラムはコールバック機能を使用しています。インライン方式については、「2 つのメッセージ処理方法」(60 ページ) を参照してください。

Client-Library コールバック・ルーチンは、ct_callback を使用してインストールします。コールバック・ルーチンは、該当タイプのトリガ・イベントが発生したときに Client-Library が自動的に呼び出すアプリケーション・ルーチンです。

コールバックには、いくつかのタイプがありますが、サンプル・プログラムはこのうち 2 つだけをインストールしています。Client-Library エラー・メッセージと情報メッセージを処理するクライアント・メッセージ・コールバックと、サーバのエラー・メッセージと情報メッセージを処理するサーバ・メッセージ・コールバックの 2 つです。

クライアント・メッセージ・コールバックは、Client-Library がエラー・メッセージまたは情報メッセージを生成すると自動的に呼び出されます。たとえば、アプリケーションが無効なパラメータ値を渡したり、ルーチンを順不同に呼び出したりすると、Client-Library はエラーを生成して、エラーの記述と一緒にクライアント・メッセージ・コールバックを呼び出します。

サーバ・メッセージ・コールバックは、サーバが結果処理中に情報メッセージまたはエラー・メッセージを送信すると呼び出されます。たとえば、アプリケーションが構文エラーのある言語コマンドを送信したり、存在しないテーブルを参照したりすると、サーバはそのエラーを説明したメッセージを送信します。

サンプル・プログラムは、cs_config を呼び出して、CS-Library エラー・ハンドラもインストールしています。CS-Library は、CS-Library 呼び出しでエラーが発生すると、アプリケーションの CS-Library エラー・ハンドラを呼び出します。

以上のほかに、次のようなタイプのコールバックがあります。

- 完了コールバック － 非同期オペレーションの完了を処理するのに非同期接続が使用します。
- ノーティフィケーション・コールバック － Open Server から受信したレジスタード・プロシージャ・ノーティフィケーションを処理するのに使用します。
- シグナル・コールバック － 非 Client-Library シグナルを処理するのに UNIX アプリケーションが使用します。

『Open Client Client-Library/C リファレンス・マニュアル』の「ct_callback」のリファレンス・ページと「コールバック」の項を参照してください。

注意 CS-Library メッセージ・コールバックと Client-Library メッセージ・コールバックのインストール方法は異なります。アプリケーションは、ct_callback ではなく cs_config を呼び出して、CS-Library メッセージ・コールバックをインストールします。インストール後は、どちらのタイプのコールバックも同じように機能します。

手順 3：サーバに接続する

サーバへの接続は、3 つの手順のプロセスです。アプリケーションは次のことを行います。

- 接続構造体を割り付けます。
- 必要に応じて、接続のプロパティを設定します。
- サーバにログインします。

接続構造体の割り付け

アプリケーションは、ct_con_alloc を呼び出して、接続構造体を割り付けます。

接続構造体プロパティの設定

アプリケーションは、ct_con_props を呼び出して、接続構造体プロパティの設定、取得、クリアを行います。

接続プロパティによって、接続の動作のさまざまな局面が定義されます。次に例を示します。

- CS_USERNAME プロパティには、接続がサーバにログインするときに使用するユーザ名を定義します。
- CS_APPNAME プロパティには、接続オープン後に Adaptive Server Enterprise の *sysprocess* テーブルに表示するアプリケーション名を指定します。
- CS_PACKETSIZE プロパティは、Tabular Data Stream™ (TDS) パケット・サイズを定義します。TDS パケット・サイズによって、アプリケーションがこの接続を介して送受信するネットワーク・パケットのサイズが決まります。デフォルトでは、Open Client Server (OCS) では、サーバ側で 512 ～ 65535 バイトのパケット・サイズを選択できます。サーバ側で指定されたパケット・サイズをサポートするサーバ (Adaptive Server Enterprise など) では、パケット・サイズを自由に設定できます。CS_PACKETSIZE で指定されているより小さいパケット・サイズ、または大きいパケット・サイズにすることもできます。

接続構造体は、割り付けられると、一部のデフォルト・プロパティ値を親のコンテキストから取り出します。たとえば、CS_APPNAME プロパティがコンテキスト・レベルで設定されている場合、そのコンテキストから割り付けられるすべての接続構造体がこのアプリケーション名を継承します。CS_PACKETSIZE など、コンテキスト・レベルでは存在しない、その他のプロパティは、デフォルトでは、標準 Client-Library 値に設定されます。

接続プロパティの一覧表については、『Open Client Client-Library/C リファレンス・マニュアル』の「ct_con_props」のリファレンス・ページを参照してください。

必須指定の接続プロパティ

アプリケーションは、最小限、接続のユーザ名を指定する接続プロパティ (CS_USERNAME) を設定して、サーバがユーザの識別を認証できるようにする必要があります。サーバはユーザの識別を次の 2 とおりの方法で確認できます。

- 有効なパスワードを要求する
- ネットワーク・ベースのユーザ認証機能を使用する

サーバがパスワードを要求する場合、アプリケーションは CS_PASSWORD プロパティにユーザのサーバ・パスワードの値を設定する必要があります。

『Open Client Client-Library/C リファレンス・マニュアル』の「セキュリティ機能」トピックのページを参照してください。

サーバへのログイン

アプリケーションは、ct_connect を呼び出して、サーバに接続します。接続を確立するとき、ct_connect がネットワークとの通信を設定し、サーバにログインし、接続固有のプロパティ情報をサーバに送信します。

たとえば、サーバがネットワークベースのユーザ認証をサポートしていて、クライアント・アプリケーションがユーザ認証を要求した場合、Client-Library とサーバは、ネットワークのセキュリティ・システムに問い合わせて、ログインしているユーザの資格 (CS_USERNAME に名前が指定されているユーザかどうか) を確認します。アプリケーションは、CS_SEC_NETWORKAUTH 接続プロパティを設定してネットワークベースのユーザ認証を要求します。

手順 4：サーバにコマンドを送信する

Client-Library では、「コマンド」はクライアント・アプリケーションからサーバに送信するアクション要求です。各コマンドは特定のコマンド・タイプに属しており、タイプによっては入力データが必要です。Client-Library は、この情報を 1 つの記号フォーマットにまとめ、これをネットワークを介して実行場所であるサーバに送信します。

firstapp.c は言語コマンドをサーバに送信します。このコマンドは、ct_command の 3 番目のパラメータ *text* に定義したクエリを解析して実行するように、サーバに指示します。他のコマンド・タイプについては、「第 5 章 コマンド・タイプの選択」を参照してください。

アプリケーションは、CS_COMMAND 構造体を使用して、コマンドを定義し、サーバに送信します。コマンドを定義し、送信するために、アプリケーションは次のことを行います。

- CS_COMMAND 構造体を割り付ける。
- 必要に応じて、コマンド構造体のプロパティを設定する。
- コマンドを起動する。
- コマンドに必要なパラメータを定義する。
- コマンドを送信する。

コマンド構造体の割り付け

アプリケーションは、ct_cmd_alloc を呼び出して、コマンド構造体を割り付けます。複数のコマンド構造体を同じ接続から割り付けることができます。

コマンド構造体プロパティの設定

アプリケーションは、ct_cmd_props を呼び出して、コマンド構造体プロパティの設定、取得、クリアを行います。

コマンド構造体プロパティは、コマンド構造体レベルで Client-Library 動作の種々の局面を決定します。たとえば、CS_HIDDEN_KEYS プロパティは、Client-Library が結果セットの一部として返される隠しキーを公開するかどうかを決定します。

firstapp.c は、コマンド構造体プロパティを設定せず、代わりにデフォルト・コマンドレベル動作を使用します。コマンド構造体は親接続からデフォルト・プロパティ値を継承します。

コマンド構造体プロパティの一覧表については、『Open Client Client-Library/C リファレンス・マニュアル』の「ct_cmd_props」のリファレンス・ページを参照してください。

コマンドの実行

アプリケーションは、ct_command、ct_cursor、または ct_dynamic を呼び出して、コマンドを起動します。ct_send は任意のタイプのコマンドをサーバに送信します。

firstapp.c は、ct_command を呼び出して言語コマンドを起動します。ct_send がサーバにコマンド・テキストを送信し、サーバがそれを解析し、コンパイルして実行します。

「第 5 章 コマンド・タイプの選択」を参照してください。

手順 5：コマンドの結果を処理する

アプリケーションは、ct_results を繰り返し呼び出して、サーバから返された結果を処理します。ほとんどすべての Client-Library プログラムが、ct_results リターン・ステータスによって制御されるループを実行する形で結果を処理します。ループ内で、結果の現在のタイプが切り替えられます。異なるタイプの結果には、異なるタイプの処理が必要です。

例の中で使用する結果処理モデルは、次の疑似コードを基にしています。

```
while ct_results returns CS_SUCCEED
        switch on result_type
            case row results
                for each column:
                    ct_bind
                end for
                while ct_fetch is returning rows
                    process each row
```

```
                end while
                check ct_fetch's final return code
            end case row results
            case command done ....
            case command failed ....
            case other result type....
            ... raise an error ...
        end switch
end while
check ct_results' final return code
```

注意 簡単な言語コマンドの場合でも、このタイプのプログラム構造を使用することをおすすめします。より複雑なプログラムの場合、アプリケーションが 1 つのコマンドに対して受信する結果セットの数とタイプを予測することは不可能です。ct_results をループで呼び出すコードは、結果処理論理がまとまっているため、保守、機能強化、再使用が比較的簡単に行えます。

ct_results は、処理の結果を設定し、処理に使用可能な結果データのタイプを指示するリターン・パラメータ *result_type* を設定します。

サンプル・プログラム *firstapp.c* が送信した select 文がサーバで正常に実行されると、CS_ROW_RESULT と CS_CMD_DONE という結果タイプを、この順序で受信します。select 文がサーバで正常に実行されなかった場合には、CS_CMD_FAIL という結果タイプを受信します。

このサンプル・プログラムは非常に簡単なので、*result_type* 切り替えの場合のような結果タイプはほとんど含まれていません。ただし、*result_type* の予期しない値については、エラーを表示するようにコーディングされています。プログラムの結果ループには、このチェックをコーディングするようにしてください。エラー表示は、プログラム作成の早い時期でのコーディングのバグ修正に役立ちます。

ロー結果については、通常、結果セット内のカラム数が決定され、その値が結果項目をプログラム変数にバインドするループを制御するのに使用されます。アプリケーションは、ct_res_info を呼び出して結果カラム数を取得し、ct_describe を呼び出して各カラムの内容を取得することができます。ただし、*firstapp.c* では、選択するカラム数とそのフォーマットがあらかじめわかっているので、これらの呼び出しは必要ありません。

ct_bind は結果項目をプログラム変数にバインドします。バインドによって、結果項目とプログラム・データ領域が対応付けられます。

ct_fetch は結果データをフェッチします。このサンプル・プログラムでは、バインドが指定されていて、各カラムの CS_DATAFMT 構造体のカウント・フィールドが 1 に設定されているので、ct_fetch 呼び出しごとに、1 つのロー・データがプログラム・データ領域にコピーされます。サンプル・プログラムは、ローがフェッチされるたびにローを出力します。

ct_fetch は、ローがなくなるまで呼び出されます。このあと、サンプル・プログラムは、ct_fetch の最後のリターン・コードを調べて、ループが正常に終了したのか、エラーで終了したのかをチェックします。

アプリケーションが受信する他の結果タイプについては、「第 6 章 結果処理コードの書き方」を参照してください。

手順 6：終了処理を行う

Client-Library アプリケーションは、終了する前に、次のことを行います。

1 各接続のすべてのコマンド構造体の割り付けを解除する。

2 オープンしているすべての接続をクローズし、割り付けを解除する。

3 Client-Library を終了する。

4 すべてのコンテキスト構造体の割り付けを解除する。

「Client-Library の終了」(27 ページ) で説明するように、前述の手順 2 は手順 3 に含めることができます。

コマンド構造体の割り付け解除

アプリケーションは、ct_cmd_drop を呼び出して、コマンド構造体の割り付けを解除します。保留中の結果やオープン・カーソルがあるコマンド構造体の割り付けを解除すると、エラーになります。

接続のクローズと割り付け解除

アプリケーションは、ct_close を呼び出して、接続をクローズし、ct_con_drop を呼び出して、クローズした接続の割り付けを解除します。クローズしていない接続の割り付けを解除すると、エラーになります。

Client-Library の終了

アプリケーションは、ct_exit を呼び出して、特定のコンテキストの Client-Library を終了します。ct_exit は、オープンしているすべての接続をクローズし、割り付けを解除し、内部 Client-Library データ領域をクリーンアップします。ct_exit はコンテキストの最後の Client-Library 呼び出しにしてください。

ct_exit が、オープンしているすべての接続をクローズし、割り付けを解除するので、各アプリケーションで、必ずしも、ct_close を呼び出して接続をクローズし、ct_con_drop を呼び出して割り付けを解除する、という 2 つの手順を行う必要はありません。ct_exit だけを呼び出しても、同じように終了できます。

コンテキスト構造体の割り付け解除

CS-Library ルーチンの cs_ctx_drop はコンテキスト構造体の割り付けを解除します。

第 2 章

構造体、定数、規則の説明

この章では、Client-Library の構造体、定数、規則について説明します。

トピック名	ページ
隠し構造体	29
接続規則とコマンド規則	31
CS_LOGINFO	32
CS_DS_OBJECT	32
CS_BLKDESC	32
CS_LOCALE	33
公開された構造体	33
定数	36
表記規則	38

隠し構造体

「隠し構造体」は、内部が文書化されていない構造体です。たとえば、Client-Library アプリケーションは、隠し構造体の割り付け、検査、修正、割り付け解除には、CS-Library または Client-Library ルーチンを呼び出す必要があります。アプリケーションはこの構造体の内容に直接アクセスできません。隠し構造体には、次のようなものがあります。

- CS_CONTEXT － Client-Library プログラミング・コンテキストを定義します。
- CS_CONNECTION － 個別のクライアント／サーバ接続を定義します。
- CS_COMMAND － コマンドを送信して結果を処理するのに使用します。
- CS_LOGINFO － サーバ・ログイン情報構造体。CS_CONNECTION と関連するこの構造体は、ユーザ名およびパスワードのようなサーバ・ログイン情報で構成されます。
- CS_DS_OBJECT － ディレクトリ・エントリに関する情報を保管します。
- CS_BLKDESC － Bulk-Library ルーチンを呼び出すアプリケーションが使用する制御構造体です。Bulk-Library については、『Open Client/Server Common Libraries リファレンス・マニュアル』を参照してください。
- CS_LOCALE － ローカライゼーション情報を保管するために使用する。

CS_CONTEXT

アプリケーションは、CS_CONTEXT またはコンテキスト構造体を割り付けてから、Client-Library を初期化する必要があります。

CS_CONTEXT 構造体には、一連のサーバ接続に関する特定の「コンテキスト」またはオペレーティング・システム環境を記述した設定情報を保管します。CS_CONTEXT は、CS-Library、Client-Library、Server-Library の間で共有します。CS_CONTEXT 構造体は、CS-Library ルーチンの cs_ctx_alloc によって割り付けられ、cs_ctx_drop によって割り付けが解除されます。

1 つのアプリケーションが複数のコンテキストを使用できますが、簡単なアプリケーションの場合、一般に 1 つのコンテキストを使用するだけです。

注意 IBM ホストの CICS で実行される Open Client アプリケーションの場合、1 つのアプリケーションにつき 1 つのコンテキストに制限されています。

一部のコンテキスト情報は、「プロパティ」の形で保管されます。プロパティの場合、アプリケーションは値を変更してコンテキストをカスタマイズできます。プロパティとしては、コンテキスト内の最大接続数を定義する CS_MAX_CONNECT、コンテキストの接続のデフォルト設定を同期動作または非同期動作のどちらにするかを決定する CS_NETIO などがあります。

接続構造体とコマンド構造体にも、プロパティがあります。接続は、割り付けられるときに、親コンテキストからデフォルト・プロパティ値を継承します。コマンド構造体は、割り付けられるときに、親接続からデフォルト・プロパティ値を継承します。

『Open Client Client-Library/C リファレンス・マニュアル』の「プロパティ」を参照してください。

CS_CONNECTION

CS_CONNECTION 構造体は、接続のユーザ名とパスワード、接続が使用するパケット・サイズ、接続が同期か非同期かなど、特定のクライアント／サーバ接続に関する情報を保管します。

コンテキストの場合と同様に、一部の接続情報はプロパティの形で保管されます。接続が作成される場合、一部のデフォルト・プロパティ値を親コンテキストから取り出します。CS_PACKETSIZE など、コンテキスト・レベルでは存在しない、その他のプロパティは、デフォルトでは、標準 Client-Library 値に設定されます。

1 つのコンテキスト内に、1 つ以上のサーバに対する複数接続が同時に存在することができます。

CS_COMMAND

CS_COMMAND 構造体、つまりコマンド構造体は、コマンドをサーバに送信し、それらのコマンドの結果を処理するのに使用します。

コマンド構造体は、特定の親接続と対応します。1 つの接続に対して、同時に複数のコマンド構造体が存在することができます。

制御構造体の階層

CS_CONTEXT、CS_CONNECTION、CS_COMMAND は、Client-Library 環境を設定し、サーバに接続し、コマンドを送信し、その結果を処理する基本制御構造体です。これら 3 つの構造体はどれも隠し構造体です。

接続規則とコマンド規則

接続構造体とコマンド構造体には、次の規則が適用されます。

- 接続内では、コマンドの結果を完全に処理してから、別のコマンドを送信しなければならない。

  カーソル結果セットを生成する ct_cursor (CS_CURSOR_OPEN) コマンドはこの規則での例外です。ct_results が CS_CURSOR_RESULT を返して、カーソル結果が使用可能であることを指示してから、次のようになります。

  - カーソル・オープン・コマンドを送信したコマンド構造体は、新しくオープンしたカーソルに関係するカーソル更新コマンドまたはカーソル削除コマンドを送信するのに使用できます。
  - 接続内の他のすべてのコマンド構造体は、新しくオープンしたカーソルとは関係のないコマンドを送信するのに使用できます。

- 各 Client-Library カーソルには、それぞれ、別のコマンド構造体を使用しなければなりません。Client-Library カーソルは、ct_cursor で宣言されるカーソルです。「第 7 章 Client-Library カーソルの使い方」を参照してください。

CS_LOGINFO

CS_LOGINFO 構造体、つまりログイン情報構造体は、ユーザ名、パスワードなど、サーバにログインするときに使用する接続構造体情報を保管するのに、内部で使用されます。

この構造体内にある接続プロパティを、「ログイン・プロパティ」といいます。

Client-Library ルーチンの ct_getloginfo と ct_setloginfo は、CS_LOGINFO 構造体を使用します。アプリケーションは、これらのルーチンを使用して、オープンしている接続のログイン・プロパティを新しい接続構造体にコピーできます。

CS_DS_OBJECT

CS_DS_OBJECT 構造体、つまりディレクトリ・オブジェクト構造体は、ディレクトリ・エントリに関する情報を保管します。Client-Library と Server-Library は、ディレクトリを使用して、接続を作成するのに必要なネットワーク・アドレス情報を保管します。このディレクトリの記憶域は、Sybase の interfaces ファイルまたは Windows のレジストリなどのネットワークベース・ディレクトリに用意されます。

アプリケーションは、Client-Library ルーチン ct_ds_lookup によるディレクトリの検索結果として、1 つ以上の CS_DS_OBJECT 構造体に対するポインタを受信します。

「第 9 章 ディレクトリ・サービスの使い方」を参照してください。

CS_BLKDESC

Bulk-Library ルーチンは、CS_BLKDESC 構造体、つまりバルク記述子構造体を使用します。バルク記述子はバルク・コピー・オペレーションの制御構造体です。

アプリケーションは、blk_alloc を呼び出して、CS_BLKDESC 構造体を割り付けます。

バルク・コピー・オペレーションを完了した後、アプリケーションは blk_drop を呼び出して、CS_BLKDESC を解放します。

Bulk-Library ルーチンについては、『Open Client/Server Common Libraries リファレンス・マニュアル』を参照してください。

CS_LOCALE

CS_LOCALE 構造体、つまりロケール構造体は、コンテキスト、接続、コマンド構造体、データ要素の各レベルでローカライゼーション情報を指定します。

CS_LOCALE 構造体には、次の項目を指定します。

- 言語、文字セット、照合順
- 日付、時刻、数値、通貨値の文字フォーマットで表現する方法

アプリケーションは、CS-Library ルーチンの cs_loc_alloc、cs_locale、および cs_loc_drop を呼び出して、CS_LOCALE 構造体を割り付け、その値を設定し、割り付けを解除できます。

『Open Client Client-Library/C リファレンス・マニュアル』の「国際化のサポート」を参照してください。

公開された構造体

「公開された構造体」は、内部が文書化されている構造体です。Client-Library アプリケーションで公開された構造体を使用するには、それを割り付ける必要があります。公開された構造体の型定義は、ヘッダ・ファイル *ctpublic.h* に含まれています。また、『Open Client Client-Library/C リファレンス・マニュアル』の「第 2 章 トピックス」にも、公開された構造体に関する説明があります。

公開された構造体には次のものがあります。

- CS_BROWSEDESC − ブラウズ記述子構造体
- CS_CLIENTMSG − Client-Library メッセージ構造体
- CS_DATAFMT − データ・フォーマット構造体
- CS_DATEREC − 日時記述子構造体
- CS_IODESC − I/O 記述子構造体
- CS_PROP_SSL_LOCALID − 復号化構造体
- CS_SERVERMSG − サーバ・メッセージ構造体
- SQLCA − SQL 通信領域構造体
- SQLCODE − SQL コード構造体
- SQLSTATE − SQL ステータス構造体

CS_BROWSEDESC

ct_br_column は、CS_BROWSEDESC 構造体を使用して、browse モード・カラムに関する情報を返します。Browse モード・カラムは、Transact-SQL select ... for browse 文から返されます。

詳細については、『Open Client Client-Library/C リファレンス・マニュアル』の「ブラウズ・モード」の項を参照してください。

CS_BROWSEDESC 構造体内のフィールドの説明については、『Open Client Client-Library/C リファレンス・マニュアル』の「CS_BROWSEDESC 構造体」を参照してください。

CS_CLIENTMSG

Client-Library は、CS_CLIENTMSG 構造体を使用して、Client-Library のエラー・メッセージまたは情報メッセージを記述します。

Client-Library のメッセージ処理については、「第 4 章 エラーおよびメッセージの処理」を参照してください。

CS_CLIENTMSG 構造体内のフィールドの説明については、『Open Client Client-Library/C リファレンス・マニュアル』の「CS_CLIENTMSG 構造体」を参照してください。

CS_DATAFMT

Client-Library ルーチンは、CS_DATAFMT 構造体を使用して、データ値とプログラム変数を記述します。

一部のルーチンには、入力パラメータとして CS_DATAFMT 構造体が必要です。たとえば、ct_bind には、バインドの送信先変数を記述するデータ・フォーマット構造体が必要であり、ct_param には、渡すパラメータを記述するデータ・フォーマット構造体が必要です。

また、アプリケーションから直接アクセスできるように、CS_DATAFMT フィールドに出力データの記述を含めるルーチンもあります。たとえば、ct_describe は結果データ項目の記述を使用して CS_DATAFMT 構造体を初期化します。

CS_DATAFMT 構造体を使用する Client-Library ルーチンは、ct_bind、ct_describe、ct_param です。CS_DATAFMT 構造体を使用する CS-Library ルーチンは、cs_convert と cs_set_convert です。

CS_DATAFMT 構造体内のフィールドの説明については、『Open Client Client-Library/C リファレンス・マニュアル』の「CS_DATAFMT 構造体」を参照してください。

CS_DATAFMT 構造体がルーチンの入力パラメータである場合、ルーチンは構造体内の使用しないフィールドの内容を無視します。たとえば、ct_bind は、*name*、*namelen*、*status*、*usertype* の各フィールドの内容を無視します。

CS_DATAFMT を使用する各ルーチンを説明しているリファレンス・ページに、使用するフィールドとその有効な値をリストしています。

CS_DATEREC

CS_DATEREC 構造体は、CS-Library ルーチンの cs_dt_crack がサーバから返された日時データを解釈するのに使用します。日時データは、サーバでは、date、time、datetime、datetime4、bigdatetime、または bigtime データ型で表示されます。これらのデータ型もパックされた構造体です。cs_dt_crack はこの日付と時刻のコンポーネントをアンパックして CS_DATEREC フィールドに入れます。

サーバの datetime データ型とこれに対応する Client-Library のデータ型については、「datetime 型」(53 ページ) を参照してください。CS_DATEREC 構造体については、『Open Client/Server Common Libraries リファレンス・マニュアル』の「cs_dt_crack」のリファレンス・ページを参照してください。

CS_IODESC

Client-Library は、CS_IODESC 構造体を使用して、text または image データを記述します。

CS_IODESC を使用して text 値と image 値をどのように処理するかについては、『Open Client Client-Library/C リファレンス・マニュアル』の「text および image データの処理」を参照してください。

CS_IODESC 構造体内のフィールドの説明については、『Open Client Client-Library/C リファレンス・マニュアル』の「CS_IODESC 構造体」の項を参照してください。

CS_PROP_SSL_LOCALID

Client-Library は CS_PROP_SSL_LOCALID 構造体を使用して、ローカル ID (認証) ファイルへのパスを指定します。CS_PROP_SSL_LOCALID 構造体には、ファイル内の情報を復号化するために使用する、ファイル名とパスワードが含まれています。

CS_PROP_SSL_LOCALID の詳細については、『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。

CS_SERVERMSG

Client-Library は、CS_SERVERMSG 構造体を使用して、サーバのエラー・メッセージまたは情報メッセージを記述します。

Client-Library のメッセージ処理については、「第 4 章 エラーおよびメッセージの処理」を参照してください。

CS_SERVERMSG 構造体内のフィールドの説明については、『Open Client Client-Library/C リファレンス・マニュアル』の「CS_SERVERMSG 構造体」を参照してください。

SQLCA、SQLCODE、および SQLSTATE

アプリケーションがエラー・メッセージまたは情報メッセージをインラインで処理している場合、Client-Library ルーチン ct_diag は SQLCA、SQLCODE、または SQLSTATE 構造体でメッセージ情報を返すことができます。

Client-Library のメッセージ処理については、「第 4 章 エラーおよびメッセージの処理」を参照してください。

SQLCA、SQLCODE、SQLSTATE の各構造体の説明については、『Open Client Client-Library/C リファレンス・マニュアル』の「SQLCA 構造体」、「SQLCODE 構造体」、「SQLSTATE 構造体」の項を参照してください。

SQLDA

アプリケーションは、Client-Library ルーチン ct_dynsqlda で SQLDA 構造体を使用して、サーバ・コマンドにパラメータを渡し、サーバ・コマンドからの結果を処理できます。

SQLDA 構造体とアプリケーションでの使い方については、『Open Client Client-Library/C リファレンス・マニュアル』の「ct_dynsqlda」のリファレンス・ページを参照してください。

定数

Client-Library は、型定数、フォーマット定数、その他の記号定数など、さまざまな定数を使用します。

特定のルーチンに関係する定数 (たとえば、戻り値として使用される記号定数) は、『Open Client Client-Library/C リファレンス・マニュアル』のそのルーチンを説明しているリファレンス・ページにリストされています。

型定数

Open Client と Open Server は、型定数を使用して、プログラム変数のデータ型を記述します。たとえば、ct_bind を呼び出して CS_DATETIME 型のバインド変数を記述する場合、アプリケーションは CS_DATAFMT 構造体の datatype フィールドを CS_DATETIME_TYPE に設定します。

型定数を使用する Client-Library ルーチンとしては、ct_bind、ct_describe、ct_param があります。また、CS-Library ルーチン cs_convert も型定数を使用します。

データ型の型定数は、データ型の名前に _TYPE を付加したものです。たとえば、データ型 CS_CHAR の型定数は CS_CHAR_TYPE です。

CS_CHAR を除くすべてのデータ型は、1 つの型定数に対応します。

CS_CHAR は、CS_CHAR_TYPE、CS_BOUNDARY_TYPE、CS_SENSITIVITY_TYPE の 3 つの型定数に対応します。このため、CS_BOUNDARY_TYPE または CS_SENSITIVITY_TYPE として記述された変数は CS_CHAR として宣言してください。

表 3-3 (50 ページ) に、Open Client の型定数が示されています。

フォーマット定数

Open Client と Open Server は、フォーマット定数を使用して、文字データ、バイナリ・データのフォーマットを記述します。特に、CS_DATAFMT 構造体のフォーマット・フィールドは、文字、テキスト、バイナリ・データのフォーマットを指示するフォーマット定数のビットマスクです。

表 2-1 に、Open Client のフォーマット定数をリストします。

表 *2-1:* フォーマット定数

フォーマット定数	有効なデータ型	結果のフォーマット
CS_FMT_NULLTERM	character と text	データは null で終了する。
CS_FMT_PADBLANK	character と text	データは変数の末尾までブランクが埋め込まれる。
CS_FMT_PADNULL	character、text、binary、image	データは変数の末尾まで null が埋め込まれる。
CS_FMT_UNUSED	すべてのデータ型	フォーマットは行われない。

その他の記号定数

Open Client はその他のさまざまな記号定数を使用します。多くの Client-Library ルーチンが記号定数を入出力パラメータ値として使用します。

表 2-2 に、Open Client で使用する記号定数の一部をリストします。

表 2-2: その他の記号定数

記号定数	意味
CS_FAIL	失敗を示すリターン・コード。
CS_FALSE	ブール式の false 値。
CS_MAX_NAME	Adaptive Server Enterprise によって許されるカラム名の最大長。
CS_NULLTERM	バッファの長さとして渡される CS_NULLTERM は、バッファ内の値が null で終了することを示す。
CS_SUCCEED	ライブラリ呼び出しが正常に実行されたことを示すリターン・コード。
CS_TRUE	ブール式の true 値。

注意 記号定数の基本となる値は、バージョンごとに変わる可能性があります。Client-Library アプリケーション・プログラマは、常に、記号定数の基本となる値ではなく、記号定数そのものを使用してコーディングしてください。

表記規則

ここでは、Client-Library のパラメータ規則について説明します。

ここで説明する項目は、NULL と未使用のパラメータ、文字列パラメータ、標準 Client-Library パラメータの *action*、*buffer*、*buflen*、*outlen* です。

NULL と未使用パラメータ

ここでは、NULL と未使用パラメータについて説明します。

ポインタ・パラメータ

ポインタ・パラメータは次のいずれかの形式になります。

- 非 NULL 値
- NULL 値
- 未使用

NULL および未使用ポインタ・パラメータは NULL として渡します。

パラメータ値が NULL である場合、パラメータに対応する長さの変数は、0 または CS_UNUSED にしてください。

パラメータが未使用である場合、パラメータに対応する長さの変数は、CS_UNUSED にしてください。

Client-Library は、現在のプログラミング・コンテキスト情報を使用して、パラメータを NULL または未使用のどちらに解釈するかを決定します。

非ポインタ・パラメータ

未使用の非ポインタ・パラメータは CS_UNUSED として渡します。

入力パラメータ文字列

ほとんどの文字列パラメータは、文字列の長さを指示するパラメータと対応しています。

null で終了する文字列を渡す場合、アプリケーションは長さパラメータを CS_NULLTERM として渡します。

null で終了しない文字列を渡す場合、対応する長さパラメータはその文字列のバイト単位の長さに設定します。

文字列パラメータが NULL である場合、対応する長さパラメータは、0 または CS_UNUSED にしてください。

出力パラメータ文字列

アプリケーションは、対応する長さパラメータを設定して、文字列バッファの長さを指示します。長さパラメータから、バッファが null で終了する出力文字列を保管するのに十分な大きさでないことがわかると、Client-Library ルーチンは CS_FAIL を返します。

基本構造体へのポインタ

すべての Client-Library ルーチンは、CS_CONTEXT 構造体、CS_CONNECTION 構造体、CS_COMMAND 構造体へのポインタをパラメータとして持ちます。

アプリケーションは、これらの構造体を (cs_ctx_alloc、ct_con_alloc、ct_cmd_alloc を使用して) 割り付けてから、パラメータとして使用します。

アプリケーションが無効な制御構造体アドレスを Client-Library ルーチンに渡すと、ルーチンは CS_FAIL を返し、Client-Library はアプリケーションのクライアント・メッセージ・コールバック・ルーチンを呼び出しません。Client-Library がアプリケーションのコールバック・ルーチンのアドレスを取得するには、有効な制御構造体のアドレスが必要です。

項目番号

結果を処理したり、結果に関する情報を返したりする多くの Client-Library ルーチンは、「項目番号」をパラメータとして持ちます。項目番号は、結果セット内の結果項目を識別するための情報であり、カラム番号、計算カラム番号、パラメータ番号、リターン・ステータス番号のいずれかです。

項目番号は、1 から始まり、現在の結果セット内の項目数を超えることはありません。アプリケーションは、*type* を CS_NUMDATA に設定して ct_res_info を呼び出すことにより、現在の結果セット内の項目番号を取得できます。

結果セットにカラムが含まれている場合、*item* はカラム番号です。カラムは、select リスト順にアプリケーションに返されます。

結果セットに計算カラムが含まれている場合、*item* は計算カラムのカラム番号です。計算カラムは compute 句に指定した順序で返されます。

結果セットにパラメータが含まれている場合、*item* はパラメータ番号です。ストアド・プロシージャのリターン・パラメータは、ストアド・プロシージャの create procedure 文にリストした順序と同じ順序で返されます。この順序は、そのストアド・プロシージャを呼び出す RPC (リモート・プロシージャ・コール) コマンドに指定した順序と必ずしも同じではありません。どの数字を *item* として渡すかを決定する場合には、リターン・パラメータ以外は計算に入れません。たとえば、ストアド・プロシージャの 2 番目のパラメータが唯一のリターン・パラメータである場合、*item* は 1 として渡します。

結果セットにリターン・ステータスが含まれている場合、*item* は常に 1 です。リターン・ステータス結果セットには、1 つのステータスしか存在しないからです。

action、*buffer*、*buflen*、*outlen*

多くの Client-Library ルーチンは、*action*、*buffer*、*buflen*、*outlen* の 4 つのパラメータを組み合わせて使用します。

- *action* − 情報を設定するのか取得するのかを指定します。ほとんどのルーチンの場合、*action* は記号値 CS_GET、CS_SET、CS_CLEAR となります。

  action が CS_CLEAR である場合、*buffer* は NULL、*buflen* は CS_UNUSED にしてください。

- *buffer* − 通常は、プログラム・データ領域を指すポインタです。

  情報を設定する場合、*buffer* は、情報を設定するときに使用する値を指します。

  情報を取得する場合、*buffer* は、Client-Library ルーチンが要求された情報を格納する領域を指します。

  情報をクリアする場合、*buffer* は NULL にしてください。

  Client-Library ルーチンが CS_FAIL を返した場合、**buffer* の内容は変わりません。

- *buflen* − *buffer* データ領域のバイト単位の長さです。

  情報を設定する場合で、かつ **buffer* 内の値が null で終了する場合、*buflen* は CS_NULLTERM として渡します。

  **buffer* が固定長値、記号値、関数である場合、*buflen* は CS_UNUSED にしてください。

  buffer が NULL である場合、*buflen* は 0 または CS_UNUSED にしてください。

- *outlen* − integer 変数に対するポインタです。

  情報を設定する場合、*outlen* は NULL にしてください。

  情報を取得する場合、*outlen* はオプション・パラメータになります。指定した場合には、Client-Library はこの変数に、要求された情報の長さをバイト単位で設定します。

  情報が *buflen* のバイト数より長い場合、アプリケーションは **outlen* の値を使用して、情報を保管するのに必要なバイト数を決定します。

表 2-3 は、*action*、*buffer*、*buflen*、*outlen* の相関関係をまとめたものです。

表 *2-3: action*、*buffer*、*buflen*、*outlen* パラメータの相関関係

action	buffer	buflen	outlen	内容
CS_CLEAR	NULL	CS_UNUSED	NULL	Client-Library 情報は、デフォルト値にリセットされ、クリアされる。
CS_SET	null で終了する文字列を指すポインタ	CS_NULLTERM または文字列の長さ。null ターミネータは含まない。	NULL	Client-Library 情報は *buffer 文字列の値に設定される。

action	buffer	buflen	outlen	内容
CS_SET	null で終了しない文字列を指すポインタ	文字列の長さ	NULL	Client-Library 情報は *buffer* 文字列の値に設定される。
CS_SET	可変長非文字値 (たとえば、バイナリ・データ) を指すポインタ	データの長さ	NULL	Client-Library 情報は *buffer* データの値に設定される。
CS_SET	固定長値または記号値を指すポインタ	CS_UNUSED	NULL	Client-Library 情報は整数値または記号値に設定される。
CS_SET	NULL	0 または CS_UNUSED	NULL	Client-Library 情報は NULL に設定される。
CS_GET	リターン文字列と null ターミネータを保管するのに十分な大きさの領域を指すポインタ	*buffer* の長さ	指定または NULL	戻り値が *buffer* にコピーされる。 null ターミネータが付加される。 指定した場合、*outlen* は null ターミネータを含む戻り値の長さに設定される。
CS_GET	リターン文字列と null ターミネータを保管するのに十分な大きさではない領域を指すポインタ	*buffer* の長さ	指定または NULL	データは *buffer にコピーされない。 指定した場合、*outlen* は null ターミネータを含む戻り値の長さに設定される。 ルーチンは CS_FAIL を返す。
CS_GET	リターン可変長非文字データを保管するのに十分な大きさの領域を指すポインタ	*buffer* の長さ	指定または NULL	戻り値が *buffer* にコピーされる。 指定した場合、*outlen* は戻り値の長さに設定される。
CS_GET	リターン可変長非文字データを保管するのに十分な大きさではない領域を指すポインタ	*buffer* の長さ	指定または NULL	データは *buffer* にコピーされない。 指定した場合、*outlen* は戻り値の長さに設定される。 ルーチンは CS_FAIL を返す。
CS_GET	固定長値または記号値を保管するのに十分な大きさと見なされる領域を指すポインタ	CS_UNUSED	指定または NULL	戻り値が *buffer* にコピーされる。 指定した場合、*outlen* は戻り値の長さに設定される。

第 3 章

Open Client と Open Server のデータ型の使い方

この章では、Open Client と Open Server で共有するデータ型について説明します。

トピック名	ページ
データ型と型定数	43
データ型の概要	50
null 代入値	57
Open Client のユーザ定義データ型	58

データ型と型定数

Client-Library はさまざまなデータ型をサポートします。これらのデータ型は CS-Library と Server-Library によって共有されます。ほとんどの場合には、これらは Adaptive Server Enterprise データ型と直接対応します。

データ型はどこで宣言するか

ヘッダ・ファイル *cstypes.h* には、Open Client データ型と Open Server データ型のすべての型定義 (typedef) が含まれています。*cstypes.h* ファイルは、*ctpublic.h* を使用して Client-Library アプリケーションにインクルードされます。明示的にインクルードする必要はありません。

プログラム変数を宣言するアプリケーションは、宣言セクションでこれらの型定義を使用します。次に例を示します。

```
CS_CHAR          buffer[40];
CS_INT           result_type, count;
CS_MONEY         profit;
```

Open Client データ型と Open Server データ型を使用する理由

アプリケーションにネイティブ C のデータ型ではなく Open Client と Open Server のデータ型を使用する理由は、異機種プラットフォーム・アーキテクチャの問題とアプリケーション・コードの移植性の 2 つがあります。

クライアント／サーバ・アプリケーションでは、異なるアーキテクチャのマシン間でデータを共有することができます。

Open Client と Open Server のデータ型は、異なるアーキテクチャのマシン間で転送し合える、プラットフォームに依存しないデータ表現です。たとえば、クライアント・プログラムが、サーバが稼働しているマシンとは異なる順序で、整数値のバイトを保管するマシンでコンパイルされ、実行されるような場合、CS_INT 値が接続を介して転送されるときに、バイトがスワップされます。このため、サーバに送信したり、サーバ・コマンドの結果から読み込んだりするデータを保持する変数は、正しい CS_TYPEDEF を使用して宣言してください。

Open Client と Open Server のデータ型を使用すれば、アプリケーション・ソース・コードをプラットフォーム間で移植することもできます。たとえば、CS_INT は常に、4 バイトの整数に対応するシステム・データ型にマップされます。Client-Library または CS-Library ルーチンの呼び出しに使用する変数は、正しい CS_TYPEDEF を使用して宣言してください。

unichar データ型

unichar は多言語クライアント・アプリケーションで使用される 2 バイト文字をサポートしており、文字セット変換に伴うオーバヘッドが減少します。

C プログラミング・データ型 CS_UNICHAR は、Open Client と Open Server の CS_CHAR データ型と同様に設計されているため、CS_CHAR データ型を使用できる場所であればどこでも使用できます。CS_UNICHAR データ型には、Unicode UCS 変換フォーマット 16 ビット (UTF-16) で、2 バイト文字の文字データが格納されます。

Open Client と Open Server の CS_UNICHAR データ型は、2 バイト文字を Adaptive Server Enterprise データベースに格納する Adaptive Server Enterprise の UNICHAR 固定幅データ型と UNIVARCHAR 可変幅データ型に対応しています。

Open Client アプリケーションはスタンドアロンとして、この機能を使用することにより、2 バイト文字の処理機能が備わっていないサーバの場合でも、クライアント側で他のデータ型と CS_UNICHAR との変換を行うことができます。

データ型と機能

2 バイト文字の送受信を行う場合、接続のログイン・フェーズ中にクライアントは優先するバイト順序を指定します。必要なバイト・スワッピングはすべて、サーバ・サイトで行われます。

Open Client ct_capability() パラメータは、次のとおりです。

- CS_DATA_UCHAR − サーバが 2 バイト文字をサポートしているかどうかを判別するために、サーバへ送信される要求です。
- CS_DATA_NOUCHAR − クライアントからサーバへ送信されるパラメータです。この接続で unichar をサポートしないようにサーバに通知します。

2 バイト文字データにアクセスするために、Open Client と Open Server には次のパラメータが実装されています。

- CS_UNICHAR − データ型
- CS_UNICHAR_TYPE − データのデータ型を識別するためのデータ型定数

CS_DATAFMT パラメータのデータ型を CS_UNICHAR_TYPE に設定すると、ct_bind、ct_describe、ct_param などの既存の API 呼び出しを使用できます。

CS_UNICHAR は、CS_DATAFMT のフォーマット・ビットマスク・フィールドを使用して、送信先のフォーマットを記述します。

たとえば、Client-Library サンプル・プログラム *rpc.c* では、データ型を記述するコード・セクションが BuildRpcCommand() 関数にあります。

```
...
strcpy (datafmt.name, "@charparam");
datafmt.namelen =CS_NULLTERM;
datafmt.datatype = CS_CHAR_TYPE;
datafmt.maxlength = CS_MAX_CHAR;
datafmt.status = CS_RETURN;
datafmt.locale = NULL;
...
```

この例では、*uni_rpc.c* サンプル・プログラムによって、文字型が `datafmt.datatype = CS_CHAR_TYPE` と定義されています。ASCII テキスト・エディタを使用して、datafmt.datatype フィールドを次のように編集してください。

```
...
strcpy (datafmt.name, "@charparam");
datafmt.namelen =CS_NULLTERM;
datafmt.datatype = CS_UNICHAR_TYPE;
datafmt.maxlength = CS_MAX_CHAR;
datafmt.status = CS_RETURN;
datafmt.locale = NULL;
...
```

サンプルは、Windows の場合は *%SYBASE%¥%SYBASE_OCS%¥sample*、UNIX の場合は *$SYBASE/$SYBASE_OCS/sample* にあります。

CS_UNICHAR は、UTF-16 形式でコード化された Unicode 文字データ型として 2 バイトで格納されるため、サーバに送信された CS_UNICHAR 文字列パラメータの最大長は、1 バイト・フォーマットで格納された CS_CHAR の半分に制限されます。

表 3-1 に、CS_DATAFMT ビットマスク・フィールドを示します。

表 *3-1: CS_DATAFMT* 構造体

ビットマスク・フィールド	説明
CS_FMT_NULLTERM	データは、2 バイトの Unicode の null (0x0000) で終了する。
CS_FMT_PADBLANK	データには、送信先変数の末尾まで 2 バイトの Unicode のブランク (0x0020) が埋め込まれる。
CS_FMT_PADNULL	データには、送信先変数の末尾まで 2 バイトの Unicode の null (0x0000) が埋め込まれる。
CS_FMT_UNUSED	フォーマット情報はない。

isql および *bcp* ユーティリティ

サーバが 2 バイト文字データをサポートする場合、isql ユーティリティと bcp ユーティリティはどちらも自動的に unichar データをサポートします。bcp は、4K、8K、16K のページ・サイズをサポートしています。

クライアントのデフォルト文字セットが UTF-8 形式である場合、isql は 2 バイト文字データを表示し、bcp は 2 バイト文字データを UTF-8 形式で保存します。それ以外の場合は、データは、バイナリ・フォーマットの 2 バイトの Unicode データとして表示、保存されます。

isql ユーティリティでクライアント文字セットを設定するには、`isql -Jutf8` を使用してください。bcp ユーティリティでクライアント文字セットを設定するには、`bcp -Jutf8` を使用してください。

制限事項

Open Client と Open Server が接続しているサーバは、2 バイトの Unicode のデータ型をサポートし、UTF-8 をデフォルトの文字セットとして使用する必要があります。

サーバが 2 バイトの Unicode データ型をサポートしていない場合は、次のエラー・メッセージが返されます。"`Type not found.Unichar/univarchar is not supported.`"

CS_UNICHAR は、CS_BOUNDARY と CS_SENSITIVITY について UTF-8 から UTF-16 へのバイト・フォーマットの変換をサポートしていません。他のデータ型フォーマットはすべて変換可能です。

CS_UNICHAR には、Unicode 文字列などの UTF-16 形式でコード化した Unicode データに関する C プログラミング操作は用意されていません。

unitext データ型

CS_UNITEXT は、Open Client と Open Server の C プログラミング・データ型で、サーバの UNITEXT データ型と直接対応しています。また、CS_UNITEXT と CS_TEXT は共通の構文とセマンティックを使用します。違いは、CS_UNITEXT では文字データが Unicode UTF-16 形式でコード化されることです。

データ型と機能

2 バイト文字の送受信を行う場合、接続のログイン・フェーズ中にクライアントは優先するバイト順序を指定します。必要なバイト・スワッピングはすべて、サーバ側で行われます。

Open Client ct_capability() パラメータは、次のとおりです。

- CS_DATA_UNITEXT − サーバが 2 バイトの Unicode データ型をサポートしているかどうかを判別するために、サーバへ送信される要求です。
- CS_DATA_NOUNITEXT − クライアントからサーバへ送信されるパラメータです。この接続で unitext を送信しないようにサーバに通知します。

2 バイト文字データにアクセスするために、Open Client と Open Server には次のパラメータが実装されています。

- CS_UNITEXT − データ型
- CS_UNITEXT_TYPE − データのデータ型を識別するためのデータ型定数

CS_DATAFMT パラメータのデータ型を CS_UNITEXT_TYPE に設定すると、ct_bind、ct_describe、ct_param、ct_setparam、cs_convert などの既存の API 呼び出しを使用できます。

CS_UNITEXT は UTF-16 Unicode データ型としてコード化され、2 バイト・フォーマットで格納されるため、CS_TEXT を使用する場所ならどこでも使用できます。CS_UNITEXT 文字列パラメータの最大長は、CS_TEXT の最大長の半分です。

CS_TEXT と同様に、CS_UNITEXT は CS_DATAFMT を使用して変換後のフォーマットを記述します。次に、format フィールドの値に使用できる記号とその意味を示します。

表 *3-2: CS_DATAFMT* 構造体

ビットマスク・フィールド	説明
CS_FMT_NULLTERM	データは、2 バイトの Unicode の null (0x0000) で終了する。
CS_FMT_PADBLANK	データには、送信先変数の末尾まで 2 バイトの Unicode のブランク (0x0020) が埋め込まれる。
CS_FMT_PADNULL	データには、送信先変数の末尾まで 2 バイトの Unicode の null (0x0000) が埋め込まれる。
CS_FMT_UNUSED	フォーマット情報はない。

isql および *bcp* ユーティリティ

Open Client アプリケーションでは、UNITEXT は常にアクティブになっています。設定パラメータは不要です。UNITEXT は、Open Client と Open Server のライブラリと、製品付属のユーティリティ (isql と bcp) に含まれています。isql はサーバの UNITEXT をバイナリ・フォーマットで表示し、bcp はバイナリ・フォーマットで保存します。

制限事項

Open Client と Open Server が接続しているサーバは、2 バイトの Unicode のデータ型をサポートしている必要があります。

サーバが 2 バイトの Unicode データ型をサポートしていない場合は、エラー・メッセージが返されます。ただしクライアントは、CS_UNITEXT から別のデータ型に、または別のデータ型から CS_UNITEXT に変換を行うことができます。

CS_UNITEXT には、Unicode 文字列などの UTF-16 形式でコード化した Unicode データに関する C プログラミング操作は用意されていません。

xml データ型

CS_XML は、Open Client と Open Server の可変幅の C プログラミング・データ型です。CS_XML は、CS_TEXT データ型と CS_IMAGE データ型に直接対応しています。CS_XML は、XML ドキュメントとそのコンテンツを表し、CS_TEXT と CS_IMAGE を使用できるところであればどこでも使用できます。

データ型と機能

Open Client ct_capability() パラメータは、次のとおりです。

- CS_DATA_XML − サーバが XML をサポートしているかどうかを判別するために、サーバへ送信される要求です。
- CS_DATA_NOXML − クライアントからサーバへ送信されるパラメータです。この接続で xml をサポートしないようにサーバに通知します。

XML データ型にアクセスするために、Open Client と Open Server には次のパラメータが実装されています。

- CS_XML − データ型
- CS_XML_TYPE − データのデータ型を識別するためのデータ型定数

CS_DATAFMT パラメータのデータ型を CS_XML_TYPE に設定すると、ct_bind、ct_describe、ct_param、ct_setparam、cs_convert などの既存の API 呼び出しを使用できます。

isql および *bcp* ユーティリティ

Open Client アプリケーションでは、XML は常にアクティブになっています。設定パラメータは不要です。XML は、Open Client と Open Server のライブラリと、製品付属のユーティリティ (isql と bcp) に含まれています。isql はサーバの XML をバイナリ・フォーマットで表示し、bcp はバイナリ・フォーマットで保存します。

制限事項

XML データをクライアントとサーバ間でやりとりできるのは、サーバが XML をサポートしている場合にかぎります。サポートしていない場合は、サーバからエラー・メッセージが返されます。サーバが XML をサポートしているかどうかを確認するには、cs_capability を使用します。クライアントは、CS_XML から別のデータ型に、または別のデータ型から CS_XML に変換を行うことができます。

次の XML 構文ルールを参照してください。

- XML の終了タグは省略できない。
- XML タグでは大文字と小文字を区別する。
- XML 要素は、正しくネストさせる必要がある。
- XML ドキュメントには、ルート要素が必要である。
- XML 属性値は必ず引用符で囲む。

XML では、スペースは保持され、CR/LF は LF に変換されます。

Open Client と Open Server は、CS_XML のドキュメントやコンテンツのチェックや検証を行いません。

型定数とは

「型定数」は、プログラム変数のデータ型を識別する記号値です。多くの CS-Library、Client-Library、Server-Library ルーチンは、プログラム変数のアドレスを CS_VOID * パラメータとして持ちます。型定数は、CS_VOID * パラメータを渡すときにデータ型を識別するのに必要です。通常、型定数は、CS_DATAFMT 構造体の *datatype* フィールドとしてルーチンに渡されます (「CS_DATAFMT」(34 ページ) を参照してください)。

データ型の概要

表 3-3 に、Open Client と Open Server の型定数、対応する型定義、対応する Adaptive Server Enterprise のデータ型を示します。

Adaptive Server Enterprise のデータ型は、Transact-SQL キーワードによって識別されます。Adaptive Server Enterprise のデータ型については、Adaptive Server Enterprise のマニュアルを参照してください。

表 *3-3:* データ型の概要

型カテゴリー	Open Client と Open Server の型定数	説明	対応する C データ型	対応するサーバ・データ型
binary 型	CS_BINARY_TYPE	バイナリ型	CS_BINARY	binary、varbinary
	CS_LONGBINARY_TYPE	長いバイナリ型	CS_LONGBINARY	なし
	CS_VARBINARY_TYPE	可変長バイナリ型	CS_VARBINARY	なし
bit 型	CS_BIT_TYPE	ビット型	CS_BIT	bit
character 型	CS_CHAR_TYPE	文字型	CS_CHAR	char、varchar
	CS_LONGCHAR_TYPE	長い文字型	CS_LONGCHAR	なし
	CS_VARCHAR_TYPE	可変長文字型	CS_VARCHAR	なし
	CS_UNICHAR_TYPE	固定長または可変長文字型	CS_UNICHAR	unichar univarchar
	CS_XML_TYPE	可変長文字型	CS_XML	xml
datetime 型	CS_DATE_TYPE	4 バイトの日付型	CS_DATE	date
	CS_TIME_TYPE	4 バイトの時刻型	CS_TIME	time
	CS_DATETIME_TYPE	8 バイトの日時型	CS_DATETIME	datetime
	CS_DATETIME4_TYPE	4 バイトの日時型	CS_DATETIME4	smalldatetime
	CS_BIGDATETIME_TYPE	8 バイトのバイナリ型	CS_BIGDATETIME	bigdatetime
	CS_BIGTIME_TYPE	8 バイトのバイナリ型	CS_BIGTIME	bigtime

型カテゴリー	Open Client と Open Server の型定数	説明	対応する C データ型	対応するサーバ・データ型
numeric 型	CS_TINYINT_TYPE	1 バイトの符号なし整数型	CS_TINYINT	tinyint
	CS_SMALLINT_TYPE	2 バイトの整数型	CS_SMALLINT	smallint
	CS_INT_TYPE	4 バイトの整数型	CS_INT	int
	CS_BIGINT_TYPE	8 バイトの整数型	CS_BIGINT	bigint
	CS_USMALLINT_TYPE	2 バイトの符号なし整数型	CS_USMALLINT	usmallint
	CS_UINT_TYPE	4 バイトの符号なし整数型	CS_UINT	uint
	CS_UBIGINT_TYPE	8 バイトの符号なし整数型	CS_UBIGINT	ubigint
	CS_DECIMAL_TYPE	10 進数型	CS_DECIMAL	decimal
	CS_NUMERIC_TYPE	数値型	CS_NUMERIC	numeric
	CS_FLOAT_TYPE	8 バイトの浮動小数点型	CS_FLOAT	float
	CS_REAL_TYPE	4 バイトの浮動小数点型	CS_REAL	real
money 型	CS_MONEY_TYPE	8 バイトの通貨型	CS_MONEY	money
	CS_MONEY4_TYPE	4 バイトの通貨型	CS_MONEY4	smallmoney
text 型および image 型	CS_TEXT_TYPE	テキスト型	CS_TEXT	text
	CS_IMAGE_TYPE	イメージ型	CS_IMAGE	image
	CS_UNITEXT_TYPE	可変長文字型	CS_UNITEXT	unitext

binary 型

Open Client には、CS_BINARY、CS_LONGBINARY、CS_VARBINARY の 3 つの binary 型があります。

- CS_BINARY は、Adaptive Server Enterprise の binary 型と varbinary 型に対応します。つまり、Client-Library はサーバの binary 型と varbinary 型をどちらも CS_BINARY として解釈します。たとえば、ct_describe は、サーバ・データ型 varbinary のある結果カラムを記述している場合、CS_BINARY_TYPE を返します。
- CS_LONGBINARY はどの Adaptive Server Enterprise データ型にも対応していませんが、一部の Open Server アプリケーションは CS_LONGBINARY をサポートします。アプリケーションは、ct_capability を呼び出し、CS_DATA_LBIN 機能をチェックして、Open Server 接続が CS_LONGBINARY をサポートするかどうかを決定します。その結果、サポートする場合には、ct_describe は、結果データ項目を記述している場合、CS_LONGBINARY を返します。CS_LONGBINARY 値の最大長は、2,147,483,647 バイトです。

- CS_VARBINARY は Adaptive Server Enterprise のどのデータ型とも対応しません。このため、Open Client ルーチンは CS_VARBINARY_TYPE を返しません。CS_VARBINARY は、バイト配列とその長さを保管する構造体です。

```
typedef struct _cs_varybin
 {
 CS_SMALLINT      len;
 CS_BYTE          array[CS_MAX_CHAR];
 } CS_VARBINARY;
```

  CS_VARBINARY は、プログラマが C 以外のプログラミング言語で Open Client 用のプログラムを書けるように用意されている構造体です。通常のクライアント・アプリケーションでは、CS_VARBINARY を使用しません。

bit 型

Open Client は、1 つの bit 型、つまり CS_BIT だけをサポートします。このデータ型は、0 または 1 のサーバ・ビット (ブール式の) 値を保持します。他の型を bit 型に変換すると、ゼロ以外のすべての値は 1 に変換されます。

character 型

Open Client には、CS_CHAR、CS_LONGCHAR、CS_VARCHAR、CS_XML の 4 つの文字型があります。

- CS_CHAR は、Adaptive Server Enterprise の char 型と varchar 型に対応します。つまり、Client-Library はサーバの char 型と varchar 型をどちらとも CS_CHAR と解釈します。たとえば、ct_describe は、サーバ・データ型 varchar のある結果カラムを記述している場合、CS_CHAR_TYPE を返します。
- CS_LONGCHAR はどの Adaptive Server Enterprise データ型にも対応していませんが、Open Server アプリケーションによっては CS_LONGCHAR をサポートするものもあります。アプリケーションは、ct_capability を呼び出し、CS_DATA_LCHAR 機能をチェックして、Open Server 接続が CS_LONGCHAR をサポートするかどうかを決定します。その結果、サポートする場合には、結果データ項目を記述するときに、ct_describe は、CS_LONGCHAR を返すことができます。CS_LONGCHAR 値の最大長は、2,147,483,647 バイトです。
- CS_VARCHAR は Adaptive Server Enterprise のどのデータ型とも対応しません。このため、Open Client ルーチンは CS_VARCHAR_TYPE を返しません。CS_VARCHAR は、プログラマが C 以外のプログラミング言語で Open Client 用のプログラムを書けるように用意されている構造体です。このデータ型は、文字列とその長さを保管します。

```
typedef struct_cs_varchar
 {
 CS_SMALLINT      len;
 CS_CHAR          str[CS_MAX_CHAR];
 } CS_VARCHAR;
```

通常のクライアント・アプリケーションは、CS_VARCHAR を使用しません。

- CS_XML は、xml データ型に直接対応しているので、CS_TEXT データ型や CS_IMAGE データ型と同様に XML データを表現できます。CS_XML は、XML データを解析されないフォーマットで表現するので、CS_TEXT と CS_IMAGE を使用できる場所であればどこでも使用できます。たとえば、cs_convert、ct_bind、ct_param などで使用できます。

  CS_XML は、サーバが XML データ型をサポートする場合にかぎりデータをフェッチします。サーバが XML データ型をサポートするかどうかを確認するため、CS_DATA_XML (要求) と CS_DATA_NOXML (応答) が ct_capability に追加されます。

  CS_XML は常にアクティブで、データ型定数は CS_XML_TYPE です。xml データ型は、TDS_XML にマップされます。

datetime 型

Open Client は、6 つの datetime 型、CS_DATE、CS_TIME、CS_DATETIME、CS_DATETIME4、CS_BIGDATETIME、CS_BIGTIME をサポートします。これらのデータ型は、8 バイトまたは 4 バイトの datetime 値を保持します。

CS_BIGDATETIME および CS_BIGTIME データ型は、マイクロ秒の精度の time データを提供します。これらのデータ型には 8 バイトのバイナリ値が格納されます。これらのデータ型はそれぞれ、CS_DATETIME データ型および CS_TIME データ型に似ています。CS_BIGDATETIME データ型は、CS_DATETIME データ型を使用する場所ならどこでも使用可能です。CS_BIGTIME データ型は、CS_TIME データ型を使用する場所ならどこでも使用可能です。CS_DATETIME データ型および CS_TIME データ型に適用できるすべての Open Client および Open Server ルーチンは、CS_BIGDATETIME データ型および CS_BIGTIME データ型にも適用できます。

- CS_DATE は、Adaptive Server Enterprise の date データ型に対応します。値の有効な範囲は 1 年 1 月 1 日～ 9999 年 12 月 31 日です。
- CS_TIME は、Adaptive Server Enterprise の time データ型に対応します。値の有効な範囲は 12:00:00.000 ～ 11:59:59:999、精度は 1/300 秒 (3.33 ミリ秒) です。
- CS_DATETIME は、Adaptive Server Enterprise の datetime データ型に対応します。値の有効な範囲は 1753 年 1 月 1 日～ 9999 年 12 月 31 日、精度は 1/300 秒 (3.33 ミリ秒) です。

- CS_DATETIME4 は、Adaptive Server Enterprise の smalldatetime データ型に対応します。値の有効な範囲は、1900 年 1 月 1 日～ 2079 年 6 月 6 日、精度は 1 分です。
- CS_BIGDATETIME は、Adaptive Server Enterprise のデータ型 bigdatetime に対応し、0000 年 1 月 1 日の 00:00:00.000000 から経過したマイクロ秒数を格納します。有効な CS_BIGDATETIME 値の範囲は、0001 年 1 月 1 日の 00:00:00.000000 から 9999 年 12 月 31 日の 23:59:59.999999 までです。

注意 0000 年 1 月 1 日の 00:00:00.000000 は、マイクロ秒数のカウントが開始される基本の値です。0001 年 1 月 1 日の 00:00:00.000000 より前の値は無効です。

- CS_BIGTIME は、Adaptive Server Enterprise のデータ型 bigtime に対応し、当日の午前 0 時ちょうどから経過したマイクロ秒数を示します。有効な CS_BIGTIME 値の範囲は、00:00:00.000000 から 23:59:59.999999 までです。
- CS_BIGDATETIME データ型および CS_BIGTIME データ型は、基本となるクライアント・プラットフォームのネイティブのバイト順序 (エディアン) のクライアントに示されます。必要であればサーバで行われるバイト・スワッピングは、クライアントにデータが送られる前、またはクライアントからのデータを受け取った後に行われます。

アプリケーションは、CS-Library ルーチン cs_convert を呼び出して、文字列から datetime 型を初期化できます。cs_convert は、Transact-SQL datetime 文字列に有効なすべての日付と時刻のフォーマットを認識します。詳細については、『ASE リファレンス・マニュアル』の「データ型」の項を参照してください。

cs_convert は、CS_DATETIME または CS_DATETIME4 値を文字列に変換することもできます。

次に、datetime 値を扱うのに便利なその他のルーチンを示します。

- cs_cmp － 2 つのデータ値を比較します。
- cs_dt_crack － datetime 値を CS_DATEREC 構造体にマップします。CS_DATEREC では、datetime 値の各部分が個別のフィールドに保管されます。
- cs_dt_info － 各言語固有の日時情報 (曜日など) を取得します。このルーチンは、datetime データ値を文字列に変換するためのフォーマットも設定します。

cs_convert、cs_cmp、cs_dt_crack、cs_dt_info は、CS_CONTEXT 構造体を使用して間接的に、または CS_LOCALE 構造体を使用して直接的に指定されているロケール情報を使用します (「CS_LOCALE」(33 ページ) を参照してください)。アプリケーションは、cs_config を呼び出してコンテキストの CS_LOC_PROP プロパティを設定することによって、CS_CONTEXT のロケール情報を変更できます。

numeric 型

Open Client は、さまざまな数値データ型をサポートします。

- 整数型には、CS_TINYINT (1 バイト整数)、CS_SMALLINT (2 バイト整数)、CS_INT (4 バイト整数)、CS_BIGINT (8 バイト整数)、CS_USMALLINT (符号なし 2 バイト整数)、CS_UINT (符号なし 4 バイト整数)、CS_UBIGINT (符号なし 8 バイト整数) があります。
- CS_REAL は、Adaptive Server Enterprise の real データ型に対応し、C 言語の float 型として実装されます。
- CS_FLOAT は、Adaptive Server Enterprise の float データ型に対応し、C 言語の double 型として実装されます。
- CS_NUMERIC と CS_DECIMAL は、Adaptive Server Enterprise のデータ型 numeric と decimal に対応します。これらは、精度および位取りのある数字をプラットフォームに依存しないでサポートするためのデータ型です。

  Adaptive Server Enterprise の numeric データ型と decimal データ型は等価で、CS_DECIMAL は CS_NUMERIC として定義されます。

money 型

Open Client は、CS_MONEY と CS_MONEY4 の 2 つの money データ型をサポートします。これらは、それぞれ 8 バイトおよび 4 バイトの money 値を保管するためのデータ型です。

- CS_MONEY は、Adaptive Server Enterprise の money データ型に対応します。有効な値の範囲は -$922,337,203,685,477.5807 から +$922,337,203,685,477.5807 までです。
- CS_MONEY4 は、Adaptive Server Enterprise の smallmoney データ型に対応します。有効な値の範囲は -$214,748.3648 ～ +$214,748.3647 です。

アプリケーションは、CS-Library ルーチン cs_convert を呼び出して、文字列から money 型を初期化できます。cs_convert ルーチンは、Transact-SQL の通貨文字列に有効なすべての通貨フォーマットを認識します。詳細については、『ASE リファレンス・マニュアル』の「データ型」の項を参照してください。

cs_convert ルーチンは、CS_MONEY 値または CS_MONEY4 値を文字列に変換することもできます。

money 値は、構造体に保管されるので、標準の C 演算子では操作できません。money 値で算術演算を行うには、アプリケーションは次のどちらかの方法を使用します。

- CS-Library ルーチン cs_calc を呼び出して、算術演算を行う。
- cs_convert を呼び出して、money 型を標準 C と同等のデータ型 (CS_FLOAT など) に変換する。

cs_cmp ルーチンは、money 値を比較するときに呼び出せます。

text 型および image 型

Open Client は、*text* データ型の CS_TEXT、*unitext* データ型の CS_UNITEXT、*image* データ型の CS_IMAGE をサポートします。

- CS_TEXT は、最大 2,147,483,647 バイトの印刷可能文字データを格納する可変長カラムを定義する、サーバのデータ型 text に対応しています。
- CS_UNITEXT は、サーバのデータ型 unitext に対応しています。text と同様、unitext は、最大 2,147,483,647 バイトまでの印刷可能文字データを格納する可変長カラムを記述します。ただし、unitext の文字データは、サーバのデフォルトの文字セットではなく、Unicode の UTF-16 形式でコード化されて保存されます。
- CS_IMAGE は、サーバ・データ型の image に対応し、最大 2,147,483,647 バイトのバイナリ・データからなる可変長カラムを記述します。

text、*unitext*、*image* のデータ値が小さい場合は、特別な処理は必要ありません。結果の値は、プログラム変数にバインドし、後でフェッチできます。また、入力データ値は、Transact-SQL の insert および update コマンドを使用して、データベースに挿入できます。ただし、*text*、*unitext*、*image* の値が大きい場合、一般的に、ルーチンを使用して text、unitext または image データを 1 つにまとめて一括処理する方が便利です。

これらには、次のようなルーチンが使用できます。

- ct_data_info － CS_IODESC 構造体を設定または取得します。CS_IODESC 構造体は、サーバから読み込んだり、サーバに書き込んだりする text、unitext、または image データを記述します。
- ct_get_data － 結果ストリームからデータチャンクを読み込みます。
- ct_send_data － データチャンクをコマンド・ストリームに書き込みます。

詳細については、『Open Client Client-Library/C リファレンス・マニュアル』の「text および image データの処理」の項を参照してください。

null 代入値

NULL 値が入っているローがサーバからフェッチされると、Client-Library は、ロー・データをプログラム変数にコピーするときに、null カラムには、指定された「null 代入値」を代入します。

表 3-4 に、Client-Library のデフォルトの null 代入値を示します。

表 *3-4:* デフォルトの *null* 代入値

送信先の型	null 代入値
CS_BINARY_TYPE	空の配列
CS_VARBINARY_TYPE	空の配列
CS_BIT_TYPE	0
CS_CHAR_TYPE	空の文字列
CS_VARCHAR_TYPE	空の文字列
CS_DATE_TYPE	4 バイトのゼロ
CS_DATETIME_TYPE	8 バイトのゼロ
CS_DATETIME4_TYPE	4 バイトのゼロ
CS_BIGDATETIME	8 バイトのゼロ
CS_BIGTIME	8 バイトのゼロ
CS_TINYINT_TYPE	0
CS_SMALLINT_TYPE	0
CS_BIGINT_TYPE	0
CS_INT_TYPE	0
CS_UINT_TYPE	0
CS_UBIGINT_TYPE	0
CS_USMALLINT_TYPE	0
CS_DECIMAL_TYPE	0.0 (デフォルトの位取りおよび精度)
CS_NUMERIC_TYPE	0.0 (デフォルトの位取りおよび精度)
CS_FLOAT_TYPE	0.0
CS_REAL_TYPE	0.0
CS_MONEY_TYPE	$0.0
CS_MONEY4_TYPE	$0.0
CS_BOUNDARY_TYPE	空の文字列
CS_SENSITIVITY_TYPE	空の文字列
CS_TEXT_TYPE	空の文字列
CS_UNITEXT_TYPE	空の文字列
CS_TIME_TYPE	4 バイトのゼロ
CS_XML_TYPE	空の文字列
CS_IMAGE_TYPE	空の配列

アプリケーションは、CS-Library ルーチン cs_setnull を呼び出して、null 代入値を変更できます。

Open Client のユーザ定義データ型

Open Client の標準データ型に含まれていないデータ型を使用するアプリケーションの場合、ユーザ定義データ型を作成できます。たとえば、暗号化された文字データを表すユーザ定義データ型を作成することができます。ユーザ定義データ型を作成する手順は、次のとおりです。

1 新しいデータ型の名前を作成します。次に例を示します。

```
typedef char ENCRYPTED_CHAR;
```

2 データ型を表す型定数を定義します。次に例を示します。

```
#define ENCRYPTED_TYPE      CS_USERTYPE + 2;
```

Open Client ルーチンの ct_bind と cs_set_convert は、記号型定数を使用してデータ型を識別するので、ユーザ定義データ型ごとに型定数を定義してください。ユーザ定義型定数は、CS_USERTYPE 以上の大きさにしてください。

3 cs_set_convert を呼び出して、Open Client の標準データ型とユーザ定義データ型の間の変換を行うカスタム変換ルーチンをインストールします。上記の例の ENCRYPTED_CHAR ユーザ定義データ型の場合、文字データを暗号化し、その暗号を解読するカスタム変換ルーチンを定義し、インストールすることになります。たとえば、CS_CHAR_TYPE から ENCRYPTED_TYPE に変換する暗号ルーチンと、ENCRYPTED_TYPE から CS_CHAR_TYPE に変換する暗号解読ルーチンをインストールします。

4 cs_setnull を呼び出して、ユーザ定義データ型の null 代入値を定義します。

変換ルーチンをインストールしたあと、アプリケーションはサーバ結果をユーザ定義データ型にバインドできます。

```
mydatafmt.datatype = ENCRYPTED_CHAR;
 ct_bind(cmd, 1, &mydatafmt, mycodename, NULL,
 NULL);
```

カスタム変換ルーチンは、必要なときに、ct_bind と cs_convert から自動的に呼び出されます。

注意 Open Client のユーザ定義データ型を Adaptive Server Enterprise のユーザ定義データ型と混同しないでください。Open Client のユーザ定義データ型は C 言語のデータ型であり、アプリケーション内で宣言します。これに対して、Adaptive Server Enterprise のユーザ定義データ型は、データベース・カラム・データ型であり、システム・ストアド・プロシージャの sp_addtype を使用して作成します。

第 4 章

エラーおよびメッセージの処理

この章では、Client-Library とサーバのエラー・メッセージおよび情報メッセージを処理するためのアプリケーション・プログラミング方法を説明します。

トピック名	ページ
メッセージについて	59
コールバック・ルーチンによるメッセージ処理	61
メッセージのインライン処理	64
長いメッセージの連続化	66
拡張エラー・データ	67
サーバ・トランザクション・ステータス	68

メッセージについて

Client-Library は、エラー状態や情報発生状態に応じてメッセージを生成します。このようなメッセージを「Client-Library メッセージ」または「クライアント・メッセージ」といいます。

サーバも、エラー状態や情報発生状態に応じてメッセージを生成します。このメッセージを「サーバ・メッセージ」といいます。

メッセージを識別する方法

Client-Library メッセージを Client-Library リターン・コードと、またサーバ・メッセージをメッセージ結果と、それぞれ混同しないでください。

Client-Library メッセージと Client-Library リターン・コード

Client-Library メッセージは、Client-Library エラーなど、問題となる状態が発生すると生成されます。各 Client-Library メッセージには、番号、テキスト、重大度レベルが含まれています。

リターン・コードは、実行の成否など、問題となる状態を示す記号値です。すべての Client-Library ルーチンがリターン・コードを使用します。

一般的に、Client-Library は、Client-Library ルーチンが CS_FAIL を返した場合、メッセージを生成しますが、その他の場合でも、メッセージを生成することがあります。

アプリケーションは、リターン・コードをチェックするだけでなく、メッセージを処理する必要があります。

サーバ・メッセージとメッセージ結果

サーバ・メッセージとメッセージ結果を混同しないように注意してください。

サーバ・メッセージは、サーバ・エラーやその他の例外的な状態に応じてサーバによって生成されます。各サーバ・メッセージには、番号、テキスト、重大度レベルが含まれています。

メッセージ結果は、通常のコマンド実行に応じて送信される結果のタイプです。詳細については、「メッセージ結果の処理」を参照してください。

サーバ・メッセージとメッセージ結果は関係がありません。

2 つのメッセージ処理方法

アプリケーションは、次の 2 つの方法のどちらかを使用して、Client-Library メッセージとサーバ・メッセージを処理できます。

- コールバック − アプリケーションは、独自のルーチンをインストールして、Client-Library メッセージとサーバ・メッセージを処理します。メッセージが生成されると、Client-Library は該当するコールバックを呼び出し、コールバックの入力パラメータを使用してメッセージに関する詳細情報を渡します。
- インライン・メッセージ処理 − アプリケーションは、メインライン・コードの中で、定期的に ct_diag を呼び出して、メッセージを取得します。

コールバックには、次のような利点があります。

- コールバックは比較的自動的に行われます。コールバックは、一度インストールすると、メッセージが発生したときに自動的にトリガされます。
- コールバックでは、メッセージ処理コードが集中化されます。
- コールバックでは、予期しないエラーをうまく処理することができます。インライン方式でエラーを処理するアプリケーションは、予想外のエラーをトラップできないことがあります。

しかし、インライン・エラー処理は、アプリケーションの直接制御下で動作するという利点があります。直接制御のために、アプリケーションは特定の時点でメッセージの有無をチェックできます。たとえば、接続をカスタマイズするために、ct_con_props を何回も呼び出すような場合には、エラー・チェックは最後の呼び出しのあとで 1 回行うだけですみます。

ほとんどのアプリケーションは、コールバックを使用してメッセージを処理しますが、コールバックをサポートしないようなプラットフォームと言語の組み合わせで動作するアプリケーションでは、インライン方式を使用してください。

アプリケーションは、ct_callback を呼び出してメッセージ・コールバックをインストールするか、ct_diag を呼び出してインライン・メッセージ処理を初期化して、どちらの方式を使用するかを指示します。

両方式の組み合わせ

アプリケーションは、異なる接続では異なる方式を使用でき、2 つの方式の間で切り替えが可能ですが、このような手法は通常のアプリケーションでは便利ではありません。

インライン方式からコールバック方式に移る場合、接続にどちらのタイプのコールバックをインストールしても、インライン・エラー処理はオフになります。Client-Library は、保存されていたメッセージをすべて廃棄してしまいます。

逆に、コールバック方式からインライン方式に移る場合、ct_diag を呼び出してインライン・メッセージ処理を初期化すると、接続のメッセージ・コールバックは「インストール解除」されます。この状態が発生すると、接続の最初の ct_diag 呼び出しのときに、警告メッセージが表示されます。

コールバック・ルーチンによるメッセージ処理

ほとんどのアプリケーションは、コールバックを使用して、Client-Library メッセージとサーバ・メッセージを処理します。アプリケーションは、Client-Library メッセージとサーバ・メッセージを処理するコールバック・ルーチンを定義し、インストールします。メッセージが生成されると、Client-Library は該当するコールバックを呼び出し、コールバックの入力パラメータを使用してメッセージに関する詳細情報を渡します。

コールバック方式を使用するには、アプリケーションに、次のコールバックを定義し、インストールする必要があります。

- Client-Library メッセージを処理するクライアント・メッセージ・コールバック
- サーバ・メッセージを処理するサーバ・メッセージ・コールバック

アプリケーションは、ct_callback を呼び出して、メッセージ・コールバックをインストールします。コールバックは、一度インストールされると、Client-Library メッセージまたはサーバ・メッセージが発生したときに、自動的にトリガされます。

Client-Library は、CS_CONNECTION 構造体と CS_CONTEXT 構造体にコールバック・ロケーションを保管します。このため、CS_CONNECTION 構造体または CS_CONTEXT 構造体が使用できなくなるような Client-Library エラーが発生すると、Client-Library はクライアント・メッセージ・コールバックを呼び出すことができません。その代わりに、エラーを起こしたルーチンが CS_FAIL を返します。

クライアント・メッセージ・コールバックの定義

クライアント・メッセージ・コールバックは、次のように定義されている C 関数です。

```
CS_RETCODE clientmsg_cb(context, connection, message)

 CS_CONTEXT       *context;
 CS_CONNECTION    *connection;
 CS_CLIENTMSG     *message;
```

各オブジェクトの意味は、次のとおりです。

- *context* は、メッセージが発生した CS_CONTEXT 構造体を指すポインタです。
- *connection* は、メッセージが発生した CS_CONNECTION 構造体を指すポインタです。*connection* は、NULL の場合もあります。
- *message* は、Client-Library メッセージ情報を含んでいる CS_CLIENTMSG 構造体を指すポインタです。CS_CLIENTMSG 構造体については、『Open Client Client-Library/C リファレンス・マニュアル』の「CS_CLIENTMSG 構造体」の項を参照してください。

  message は、クライアント・メッセージ・コールバックが呼び出されるたびに、新しい値を持つことになります。

他のコールバックと同様に、クライアント・メッセージ・コールバックは、呼び出せる Client-Library ルーチンが制限されています。クライアント・メッセージ・コールバックは次のルーチンだけを呼び出すことができます。

- ct_config － 情報の取得のみを行う
- ct_con_props － 情報の取得または CS_USERDATA プロパティの設定のみを行う
- ct_cmd_props － 情報の取得または CS_USERDATA プロパティの設定のみを行う
- ct_cancel(CS_CANCEL_ATTN)

クライアント・メッセージ・コールバックは、次のリターン・コードの 1 つを返します。

- CS_SUCCEED － この接続で発生している処理を続けるように Client-Library に指示します。タイムアウト・エラーに備えて、CS_SUCCEED は、Client-Library にもう 1 回タイムアウト期間を待つように指示します。この期間が終了すると、Client-Library はクライアント・メッセージ・コールバックを再度呼び出します。
- CS_FAIL － この接続で現在発生している処理を終了するように Client-Library に指示します。CS_FAIL が返されると、接続は dead というマークが付けられます。この接続を使用して処理を続行するには、アプリケーションは、接続を一度クローズして、再オープンする必要があります。

サーバ・メッセージ・コールバックの定義

サーバ・メッセージ・コールバックは、次のように定義されている C 関数です。

```
CS_RETCODE servermsg_cb(context, connection, message)

 CS_CONTEXT       *context;
 CS_CONNECTION    *connection;
 CS_SERVERMSG     *message;
```

各オブジェクトの意味は、次のとおりです。

- *context* は、メッセージが発生した CS_CONTEXT 構造体を指すポインタです。
- *connection* は、メッセージが発生した CS_CONNECTION 構造体を指すポインタです。*connection* は、NULL の場合もあります。
- *message* は、サーバ・メッセージ情報を保管する CS_SERVERMSG 構造体を指すポインタです。CS_SERVERMSG フィールドについては、『Open Client Client-Library/C リファレンス・マニュアル』の「CS_SERVERMSG 構造体」の項を参照してください。

  message は、サーバ・メッセージ・コールバックが呼び出されるたびに、新しい値を持つことになります。

他のコールバックと同様に、サーバ・メッセージ・コールバックは、呼び出せる Client-Library ルーチンが制限されています。サーバ・メッセージ・コールバックは次のルーチンだけを呼び出すことができます。

- ct_config － 情報の取得のみを行う
- ct_con_props － 情報の取得または CS_USERDATA プロパティの設定のみを行う
- ct_cmd_props － 情報の取得または CS_USERDATA プロパティの設定のみを行う

- ct_cancel(CS_CANCEL_ATTN)
- ct_res_info、ct_bind、ct_describe、ct_fetch、ct_get_data － 拡張エラー・データだけを処理します。

サーバ・メッセージ・コールバックは CS_SUCCEED を返します。

コールバックのインストール

アプリケーションは、ct_callback を呼び出して、クライアントまたはサーバ・メッセージ・コールバックをインストールします。

アプリケーションがコンテキスト・レベルでコールバックをインストールした場合、コンテキスト内で割り付けられたすべての接続構造体がコールバックを継承します。

既存のコールバック・ルーチンのインストールを解除するには、*action* を CS_SET、*func* を NULL に設定して ct_callback を呼び出します。

既存のコールバック・ルーチンを新しいルーチンに置き換えるには、*action* を CS_SET に設定して ct_callback を呼び出し、新しいルーチンをインストールします。ct_callback は、既存のコールバックを新しいコールバックに置き換えます。

既存のコールバックを指すポインタを取得するには、*action* を CS_SET、*func* を CS_VOID * 変数のアドレスに設定して ct_callback を呼び出します。ct_callback はこの変数にコールバックのアドレスを入れます。

メッセージのインライン処理

Client-Library アプリケーションは、ct_diag を呼び出して、Client-Library メッセージとサーバ・メッセージをインラインで処理します。

アプリケーションがインライン・エラー処理を使用できるのは、接続レベルだけです。つまり、インライン・エラー処理は、コンテキストには使用できません。アプリケーションに複数の接続がある場合、接続ごとに別々の ct_diag 呼び出しを行う必要があります。

アプリケーションは ct_diag を呼び出して、次のことを行います。

- インライン・エラー処理を初期化する。
- メッセージをクリアする。
- メッセージを取得する。
- 保存するメッセージ数を制限する。

- 現在保存されているメッセージ数を調べる。
- 拡張エラー・データが用意されている CS_COMMAND 構造体を取得する。「拡張エラー・データ」(67 ページ) を参照してください。

Client-Library は、接続のインライン・エラー処理が初期化されるまで、その接続のメッセージの保存を開始しません。

アプリケーションは、クライアント・メッセージ情報を CS_CLIENTMSG 構造体に取り込むか、または SQLCA 構造体、SQLCODE 構造体、SQLSTATE 構造体のどれかに取り込むことができます。また、サーバ・メッセージ情報は CS_SERVERMSG 構造体に取り込むか、または SQLCA 構造体、SQLCODE 構造体、SQLSTATE 構造体のどれかに取り込むことができます。これらの構造体については、『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。

CS_CONNECTION 構造体を使用できなくなるような Client-Library エラーが発生すると、ct_diag は、元のエラーに関する情報を取得するために呼び出されたときに、CS_FAIL を返します。

CS_EXTRA_INF プロパティ

SQLCA、SQLCODE、または SQLSTATE へのメッセージを取得しているアプリケーションは、Client-Library プロパティ CS_EXTRA_INF を CS_TRUE に設定する必要があります。

CS_EXTRA_INF プロパティは、コマンドによって影響を受けるローの数など、ある種の情報メッセージを Client-Library が返すかどうかを決定します。通常、アプリケーションは、ct_res_info を呼び出して、この情報を取得します。CS_EXTRA_INF が CS_TRUE に設定されていると、情報は Client-Library メッセージとして返されます。

SQL 構造体を使用していないアプリケーションも、CS_EXTRA_INF を CS_TRUE に設定することができます。この場合、追加の情報は標準の Client-Library メッセージとして返されます。

CS_DIAG_TIMEOUT_FAIL プロパティ

インライン・エラー処理が有効な場合、CS_DIAG_TIMEOUT_FAIL プロパティは、Client-Library タイムアウト・エラーで Client-Library が実行不可能になるか、リトライできるかを決定します。

長いメッセージの連続化

メッセージ・コールバック・ルーチンと ct_diag は、Client-Library サーバ・メッセージを CS_CLIENTMSG 構造体と CS_SERVERMSG 構造体に返します。CS_CLIENTMSG 構造体では、メッセージ・テキストは *msgstring* フィールドに保管されます。CS_SERVERMSG 構造体では、メッセージ・テキストは *text* フィールドに保管されます。*msgstring* も *text* も長さは CS_MAX_MSG バイトです。

CS_MAX_MSG - 1 バイトよりも長いメッセージが生成された場合、Client-Library のデフォルト動作では、メッセージをトランケートします。ただし、アプリケーションは、CS_NO_TRUNCATE プロパティを使用して、長いメッセージをトランケートせずに「連続させる」ように Client-Library に指示できます。

Client-Library が長いメッセージを連続させている場合、Client-Library は、メッセージのすべてのテキストを返すために必要な分の CS_CLIENTMSG または CS_SERVERMSG 構造体を使用します。メッセージの最初の CS_MAX_MSG バイトは 1 番目の構造体に、次の CS_MAX_MSG バイトは 2 番目の構造体に (以降同様) 返されます。

Client-Library は、メッセージの最後のまとまりだけを NULL で終了させます。メッセージが CS_MAX_MSG バイトの長さと同じ場合、メッセージは 2 つのまとまりとして返されます。この場合、最初のまとまりは、CS_MAX_MSG バイトのメッセージで構成され、2 番目のまとまりは、null ターミネータだけで構成されます。

アプリケーションが、コールバック・ルーチンを使用してメッセージを処理している場合、Client-Library は各メッセージ部分に対して一度ずつ、コールバック・ルーチンを呼び出します。

アプリケーションが ct_diag を使用してメッセージを処理する場合、メッセージの各まとまりにつき 1 回 ct_diag を呼び出す必要があります。

注意 SQLCA 構造体、SQLCODE 構造体、SQLSTATE 構造体は、連続化メッセージをサポートしません。アプリケーションは、これらの構造体を使用して連続しているメッセージを取得することはできません。これらの構造体に対して長すぎるメッセージは、トランケートされます。

オペレーティング・システム・メッセージは、CS_CLIENTMSG 構造体の *osstring* フィールドにレポートされます。Client-Library はオペレーティング・システム・メッセージを連続化しません。

詳細については、『Open Client Client-Library/C リファレンス・マニュアル』の「エラーおよびメッセージの処理」の項を参照してください。

拡張エラー・データ

一部のサーバ・メッセージには、対応する拡張エラー・データがあります。拡張エラー・データは、エラーに関する追加情報です。Adaptive Server Enterprise メッセージの場合、通常、追加情報には、どのカラムがエラーを引き起こしたかが記述されています。

Client-Library は、拡張エラー・データをパラメータ結果セットの形式でアプリケーションに使用できるようにします。パラメータ結果セットの各結果項目は拡張エラー・データの一部です。拡張エラー・データの部分には、任意のデータ型の名前を付けることができます。

アプリケーションは拡張エラー・データを取得することはできますが、その必要はありません。

拡張エラー・データの用途

エンド・ユーザがデータを入力または編集できるアプリケーションの場合、一般に、カラム・レベルでユーザにエラーをレポートする必要があります。しかし、標準サーバ・メッセージ・メカニズムでは、カラム・レベルの情報は、サーバ・メッセージのテキスト内でしか使用できません。拡張エラー・データを使用すると、アプリケーションがカラム・レベルの情報に簡単にアクセスできるようになります。

たとえば、pubs2 データベースの titleauthor テーブルにエンド・ユーザがデータを入力および編集できるアプリケーションがあるとします。titleauthor は、au_id と title_id の 2 カラムからなるキーを使用します。この au_id と title_id 値が既存のローのキーと一致するローを入力しようとすると、“duplicate key” メッセージがアプリケーションに送信されます。

このメッセージを受け取ったアプリケーションは、ユーザが値をすぐに訂正できるように、問題のカラムをエンド・ユーザに指示する必要があります。この情報は、“duplicate key” メッセージのテキスト内にも使用可能ですが、アプリケーションはこのテキストを解析してカラム名を抽出しなければなりません。

拡張エラー・データを識別して処理する方法については、『Open Client Client-Library/C リファレンス・マニュアル』の「エラーおよびメッセージの処理」の項を参照してください。

サーバ・トランザクション・ステータス

サーバ・トランザクション・ステータス情報は、アプリケーションがトランザクションの状態を判別する必要があるときに便利です。表 4-1 に、トランザクション・ステータスを表す記号値を示します。

表 *4-1:* トランザクション・ステータス

記号値	意味
CS_TRAN_IN_PROGRESS	トランザクションは進行中である。
CS_TRAN_COMPLETED	直前のトランザクションは正常に終了した。
CS_TRAN_STMT_FAIL	現在のトランザクションで直前に実行した文は失敗した。
CS_TRAN_FAIL	直前のトランザクションは失敗した。
CS_TRAN_UNDEFINED	トランザクション･ステータスが定義されていない。

メインライン・コードおよびサーバ・コールバック・ルーチンからサーバ・トランザクション・ステータスを取得する方法については、『Open Client Client-Library/C リファレンス・マニュアル』の「エラーおよびメッセージの処理」の項を参照してください。

第 5 章

コマンド・タイプの選択

Client-Library には、いくつかのコマンド・タイプが用意されています。この章では、各コマンド・タイプを紹介し、その使い方と、各コマンド・タイプの利点と欠点について説明します。

トピック名	ページ
コマンドについて	69
コマンド・タイプ	70
コマンドの実行	70
言語コマンド	72
RPC コマンド	74
Client-Library カーソル・コマンド	79
動的 SQL コマンド	80
メッセージ・コマンド	81
パッケージ・コマンド	83
データ送信コマンド	83

コマンドについて

Client-Library アプリケーションでは、コマンドは、クライアントからサーバに送信される TDS プロトコル記号とデータのストリームです。コマンドには、サーバが実行すべきオペレーションが記述されているほか、オペレーションに必要なパラメータ・データも含まれています。Client-Library は、アプリケーションの API 呼び出しに応じて、コマンドを TDS プロトコルでコード化します。

コマンド・タイプ

表 5-1 は、Client-Library のコマンド・タイプの一覧表です。

表 *5-1:* コマンド・タイプの一覧表

コマンド・タイプ	起動コマンド	概要
言語	ct_command	サーバが解析、解釈、実行するクエリのテキストを定義する。
RPC、 パッケージ	ct_command	サーバが実行するサーバ・プロシージャ (Adaptive Server Enterprise ストアド・プロシージャまたは Open Server レジスタード・プロシージャ) の名前を指定する。プロシージャは前もってサーバ上になければならない。 パッケージ・コマンドは、CICS サーバ・アプリケーションのために Open Server に接続するクライアント・アプリケーションだけに使用可能である。それ以外の場合は、RPC コマンドと同じである。
カーソル	ct_cursor	Client-Library カーソルを管理するコマンドを起動する。
動的 SQL	ct_dynamic	リテラル SQL 文 (文の内容に制限がある文) を実行するコマンドまたは準備された動的 SQL 文を管理するコマンドを起動する。
メッセージ	ct_command	メッセージ・コマンドを起動し、メッセージ・コマンド ID 番号を指定する。
データ送信	ct_command	大きい text または image カラム値をサーバにアップロードするコマンドを起動する。

コマンドの実行

すべてのコマンドは、次のような手順で実行されます。

1 コマンドを起動する − この手順で、コマンド・タイプと実行する処理を指示します。

2 パラメータ値を定義する − コマンドの中には、パラメータ・データを入力として必要とするものもあります。

3 コマンドを送信する − ct_send コマンド記号とデータをネットワークに書き込みます。サーバは、コマンドを読み込み、解釈し、実行します。

4 コマンドの結果を処理する − ct_results ループで呼び出され、コマンドの結果を読み込みます。「基本ループの構造」(86 ページ) を参照してください。

コマンドの起動

アプリケーションは、いくつかのタイプのコマンドをサーバに送信できます。

- アプリケーションは、ct_command を呼び出して、言語コマンド、メッセージ・コマンド、パッケージ・コマンド、リモート・プロシージャ・コール (RPC) コマンド、データ送信コマンドを起動します。
- アプリケーションは、ct_cursor を呼び出して、カーソル・コマンドを起動します。
- アプリケーションは、ct_dynamic を呼び出して、動的 SQL コマンドを起動します。

コマンドのパラメータの定義

次のタイプのコマンドにはパラメータを指定できます。

- 言語コマンド (コマンド・テキストに変数が含まれている場合)
- RPC コマンド (ストアド・プロシージャがパラメータを持つ場合)
- カーソル宣言コマンド (カーソルの本体にホスト言語パラメータが含まれている場合)
- カーソル・オープン・コマンド (カーソルの本体にホスト言語パラメータが含まれている場合)
- メッセージ・コマンド
- 動的 SQL 実行コマンド

アプリケーションは、コマンドが必要とするパラメータごとに ct_param または ct_setparam を 1 回呼び出します。この 2 つのルーチンは同じ機能を実行します。ただし、ct_param はパラメータ値をコピーするのに対して、ct_setparam は値が入っている変数のアドレスをコピーします。ct_setparam を使用すると、Client-Library は、コマンドが送信されるときに、パラメータ値を読み込みます。ct_setparam 方式の場合、アプリケーションは、コマンドを再送信する前にパラメータ値を変更できます。

結果の処理

アプリケーションは、コマンドが送信されるたびに、結果を処理またはキャンセルする必要があります。標準的なアプリケーションは、CS_SUCCEED 以外の値が返されるまで、ct_results を呼び出します。「基本ループの構造」(86 ページ) を参照してください。

コマンドの再送信

Client-Library アプリケーションは、ほとんどのコマンド・タイプについて、前の実行の結果を処理した直後にコマンドを再送信できます。アプリケーションは、次のようにしてコマンドを再送信します。

- 必要な場合、アプリケーションはパラメータ送信元変数の値を変更します。

  アプリケーションは、ct_setparam コマンドを定義するときに、このコマンドにパラメータ送信元変数のアドレスを指定する必要があります。

- ct_send を呼び出して、コマンドを再送信します。

アプリケーションは、次のコマンドを除くすべてのタイプのコマンドを再送信できます。

- ct_command(CS_SEND_DATA_CMD) によって起動されたデータ送信コマンド
- ct_command(CS_SEND_BULK_CMD) によって起動されたバルク送信コマンド

言語コマンド

言語コマンドは、クエリのテキストをサーバに送信します。サーバは、コマンドを解析し実行して応答します。

Adaptive Server Enterprise の言語コマンドは、Transact-SQL を使用して記述してください。Replication Server® など、他のサーバは異なる言語を使用します。

言語コマンドの構築

アプリケーションは、*type* を CS_LANG_CMD に、**buffer* を言語テキストに設定して、ct_command を呼び出すことによって、言語コマンドを起動します。たとえば、次の呼び出しでは、pubs2 データベースの authors テーブルからローを選択する言語コマンドを起動します。

```
ret = ct_command(cmd, CS_LANG_CMD,
 "select au_lname, city from pubs2..authors ¥
 where state = 'CA'",
 CS_NULLTERM, CS_UNUSED);
```

言語コマンドにはパラメータを指定できます。Adaptive Server Enterprise クライアント・アプリケーションの場合、パラメータ位置は、コマンド・テキスト内の未宣言変数によって指示されます。たとえば、次の例のような言語コマンドは、“@state_name” に値が代入されるパラメータ値を持ちます。

```
select au_lname, city from pubs2..authors ¥
       where state = @state_name
```

同じ言語コマンドを複数回実行するようなコーディングの場合にパラメータを使用すると便利です。

言語コマンドの結果処理

標準結果ループで言語コマンドの結果を処理するように、アプリケーションをコーディングします。標準結果ループについては、「基本ループの構造」(86 ページ) で説明します。

言語コマンドは、状況に応じて、表 5-2 に示す結果タイプを返すことができます。

表 5-2: 言語コマンドの実行から返される結果タイプ

結果タイプ	意味および受信する時期
CS_ROW_RESULT	通常ロー。言語バッチまたは呼び出されたストアド・プロシージャによって実行された select 文に対する応答として送信される。
CS_COMPUTE_RESULT	計算ロー。compute 句が含まれている select 文に対する応答として送信される。select 文は、言語バッチまたは呼び出されたストアド・プロシージャによって実行できる。
CS_PARAM_RESULT	出力パラメータ値。パラメータ値を渡す exec 文に対する応答として送信される (パラメータは exec 文の output に条件を指定する)。出力パラメータ値は、プロシージャによって実行されたすべての文の結果のあとで受信する。
CS_STATUS_RESULT	ストアド・プロシージャのリターン・ステータス。exec 文に対する応答として送信される。リターン・ステータスは、プロシージャによって実行されたすべての文の結果のあとで受信する。
CS_COMPUTEFMT_ RESULT、CS_ROWFMT_RESULT	フォーマット結果。この結果タイプは、CS_EXPOSE_FMTS 接続プロパティが CS_TRUE (デフォルトは CS_FALSE) の場合にだけ、表示される。
CS_CURSOR_ RESULT	カーソル結果ローは、ct_fetch または ct_scroll_fetch を使用して取得できる。
CS_CMD_DONE	1 つの論理コマンドの結果が処理されたことを示すプレースホルダ。次のイベントのあとで表示される。 • 言語バッチ内で実行された各文の結果が処理された。 • 呼び出されたストアド・プロシージャによって実行された各 select 文の結果が完全に処理された。
CS_CMD_SUCCEED	言語バッチによって直接実行された insert、update、exec 文が正常に完了したことを示す。
CS_CMD_FAIL	言語バッチ内のコマンドまたは文が実行に失敗したことを示す。

言語コマンドの使用が適する場合

言語コマンドは、アドホック・クエリを実行するアプリケーションに適しています。たとえば、Sybase isql コマンド・インタプリタでは、エンド・ユーザがクエリを入力し、クエリを言語コマンドとしてサーバに送信し、結果を表示できます。

言語コマンドはまた、SQL クエリを Client-Library から Sybase サーバに渡すようなクライアント・サイドのミドルウェア・アプリケーションでも役立ちます。

言語コマンドの使用が適さない場合

常に同じクエリを実行するアプリケーションは、言語コマンドではなく、ストアド・プロシージャを呼び出すようにコーディングする方が、より良いパフォーマンスが得られます。このような場合、C アプリケーション・コードでクエリをコーディングしないで、クエリを実行するストアド・プロシージャを作成し、RPC コマンドを使用してそのストアド・プロシージャを呼び出すようにします。この方式は、サーバが実行するたびにクエリを解析および解釈する必要がないので、より高速になります。

ストアド・プロシージャは、1 回のプロシージャ呼び出しが複数のクライアント・コマンドと置き換わる場合、かなり高速になります。

ストアド・プロシージャは、execute 言語コマンドでも RPC コマンドでも実行できます。両者の違いについては、「RPC と execute 言語コマンドの比較」(79 ページ) を参照してください。

RPC コマンド

RPC コマンドは、ストアド・プロシージャまたはレジスタード・プロシージャの名前と、プロシージャにパラメータがある場合にはそのパラメータ値を、サーバに送信します。プロシージャが存在する場合、サーバはそれを実行して、結果を返します。

Adaptive Server Enterprise に対する RPC コマンドは、ストアド・プロシージャを呼び出します。Open Server アプリケーションに対する RPC コマンドは、レジスタード・プロシージャまたは Open Server の RPC イベント・ハンドラを呼び出します。

Adaptive Server Enterprise ストアド・プロシージャの作成方法については、『Transact-SQL ユーザーズ・ガイド』を参照してください。レジスタード・プロシージャについては、『Open Server Server-Library/C リファレンス・マニュアル』の「レジスタード・プロシージャ」の項を参照してください。

RPC コマンドの構築

アプリケーションは、*type* を CS_RPC_CMD に、**buffer* をプロシージャ名に、*option* を CS_NO_RECOMPILE、CS_RECOMPILE、CS_UNUSED のどれかにそれぞれ設定して ct_command を呼び出すことによって、RPC コマンドを起動します。次に例を示します。

```
ct_command(cmd, CS_RPC_CMD, rpc_name, CS_NULLTERM,
CS_NO_RECOMPILE)
```

option 値は、サーバがプロシージャを再コンパイルすべきかどうかを指示します。Adaptive Server Enterprise ストアド・プロシージャを呼び出す場合、CS_RECOMPILE は、同等の execute 文に with recompile 句を指定するのと同じです。どんな場合に再コンパイルが必要であるかについては、Adaptive Server Enterprise のマニュアルを参照してください。

RPC コマンドのパラメータ値は、ct_param または ct_setparam の呼び出しで渡されます。この 2 つのルーチンの機能はまったく同じですが、ct_param はデータ値をコピーするのに対して、ct_setparam はデータ値を指すポインタをコピーします。どちらのルーチンにも、CS_DATAFMT 構造体、インジケータ変数、データ値のアドレスが必要です。詳細については、『Open Client Client-Library/C リファレンス・マニュアル』の「ct_param」と「ct_setparam」のリファレンス・ページを参照してください。

RPC コマンドの場合、次の規則に従って ct_param または ct_setparam をコーディングします。

- ストアド・プロシージャのパラメータの宣言しているデータ型でパラメータ値を渡します。

  Client-Library は送信するパラメータ値を変換しません。必要があれば、cs_convert を使用して、パラメータ値を一致するデータ型に変換してください。

- すべてのパラメータを名前で、またはすべてのパラメータを位置で渡します。

  パラメータを名前で渡すには、ct_param または ct_setparam の *datafmt* パラメータの *name* フィールドに名前をコピーし、*datafmt.length* を宣言と一致するように設定します。ct_param または ct_setparam を呼び出さないパラメータは、NULL で渡すと効果的です。

  パラメータを位置で渡すには、*datafmt.length* を 0 に設定して、プロシージャの定義に指定されているパラメータ順序で ct_param または ct_setparam を呼び出します。パラメータを NULL で渡すには、対応する *indicator* 変数を -1 に設定します。

  すべてのパラメータを同じ方式で渡してください。パラメータを位置で渡す RPC コマンドは、一般に、パラメータを名前で渡す RPC コマンドよりもパフォーマンスが高くなります。

- *datafmt.status* を設定して、パラメータがリターン・パラメータであるかどうかを指示します。

  CS_RETURN はリターン・パラメータを指示します。リターン・パラメータ以外には、CS_INPUTVALUE を使用してください。

  リターン・パラメータは、一部のプログラミング言語が提供している「参照渡し」と似ています。パラメータの値とプロシージャ・コードが行った変更は、プロシージャが実行を完了したあと、クライアント・アプリケーションに対して使用可能になります。「リターン・パラメータ値」(77 ページ) を参照してください。

- コマンドがパラメータ値を変えて複数回送信される場合には、ct_param ではなく ct_setparam を使用します。

  ct_setparam の場合、起動されたコマンドにパラメータ送信元変数をバインドするので、アプリケーションは ct_send 呼び出し間でパラメータの値を変更できます。

パラメータを持つ RPC コマンドの定義例については、『Open Client Client-Library/C リファレンス・マニュアル』の「ct_param」のリファレンス・ページを参照してください。

RPC コマンド結果の処理

「基本ループの構造」(86 ページ) の説明に従って、標準結果ループで RPC コマンドの結果を処理するように、アプリケーションをコーディングします。

RPC コマンドは、状況に応じて、表 5-3 に示す結果タイプを返すことができます。

表 *5-3: RPC* コマンドの実行から返される結果タイプ

結果タイプ	意味および受信する時期
CS_ROW_RESULT	通常ロー。プロシージャによって実行された select 文に対する応答として送信される。
CS_COMPUTE_RESULT	計算ロー。compute by 句が含まれている select 文に対する応答として送信される。
CS_PARAM_RESULT	リターン (出力) パラメータ値。プロシージャ内のすべての文の結果が処理されたあとで受信する。
CS_STATUS_RESULT	プロシージャのリターン・ステータス。プロシージャ内のすべての文の結果が処理されたあとで受信する。
CS_COMPUTEFMT_ RESULT、CS_ROWFMT_RESULT	フォーマット結果。この結果タイプは、CS_EXPOSE_FMTS 接続プロパティが CS_TRUE (デフォルトは CS_FALSE) の場合にだけ、表示される。
CS_CMD_DONE	1 つの論理コマンドの結果が処理されたことを示すプレースホルダ。次のイベントのあとで表示される。 • 言語バッチ内で実行された各文の結果が処理された。 • 呼び出されたストアド・プロシージャによって実行された各 select 文の結果が完全に処理された。

結果タイプ	意味および受信する時期
CS_CMD_SUCCEED	プロシージャの呼び出しが正常に終了したことを示すが、ストアド・プロシージャ内のすべての文が正常に実行されたとは限らない。アプリケーションは、常に、ストアド・プロシージャのリターン・ステータス値をチェックして、エラーが発生しているかどうかを確認しなければならない (「リターン・ステータス値」(77 ページ) を参照)。
CS_CMD_FAIL	プロシージャ呼び出しが失敗したことを示す。すべてのエラーについて、CS_CMD_FAIL が返されるわけではない。ストアド・プロシージャ内の文が実行に失敗しても、サーバは CS_CMD_SUCCEED という結果タイプを返すことがある。 アプリケーションは、常に、ストアド・プロシージャのリターン・ステータス値をチェックして、エラーが発生しているかどうかを確認しなければならない (「リターン・ステータス値」(77 ページ) を参照)。

リターン・パラメータ値

サーバは、次の 2 つの条件を満たすパラメータについては、RPC コマンドの結果に対してパラメータ値を返します。

- パラメータが RPC コマンドのリターン・パラメータとして渡されている。
- パラメータがプロシージャの定義に出力パラメータとして定義されている。

パラメータ・データが返される場合、すべてのパラメータ値は、CS_PARAM_RESULT 結果セットに返されます。

リターン・ステータス値

リターン・ステータス値は、CS_STATUS_RESULT 結果セットとして返されます (「リターン・ステータス結果の処理」(93 ページ) を参照してください)。

注意　言語コマンドによって実行された場合に CS_CMD_FAIL の結果タイプを返す SQL 文が、ストアド・プロシージャによって実行された場合には CS_CMD_SUCCEED を返すことがあります。必ずストアド・プロシージャのリターン・ステータスをチェックして、プロシージャが正常に実行されたかどうかを調べてください。

プロシージャが正常に実行を完了した場合、リターン・ステータスは、プロシージャから明示的に返される値、または明示的な return 文がプロシージャにない場合には 0 です。ただし、ランタイム・エラーによっては、ストアド・プロシージャの実行が完了しないでアボートされることもあります。たとえば、プロシージャ内の select 文が、もう存在しないテーブルを参照することがあります。このようなエラーの場合、Adaptive Server Enterprise は、プロシージャの実行をアボートし、エラーを示すリターン・ステータス値を返します。リターン・ステータス・コードとその意味については、『ASE リファレンス・マニュアル』の「return」のリファレンス・ページを参照してください。

実行時エラーがストアド・プロシージャ内で発生した場合、Adaptive Server Enterprise は CS_CMD_FAIL の結果タイプを返しません。プロシージャ内でサーバ・サイドのエラーが発生しているかどうかを決定するために、アプリケーションは、常にストアド・プロシージャのリターン・ステータスをチェックする必要があります。Adaptive Server Enterprise も、ランタイム・エラーを記述したサーバ・メッセージを送信します。

RPC コマンドの使用が適する場合

RPC コマンドには、次のような利点があります。

- ストアド・プロシージャのパラメータ値は、サーバ・サイドでの変換が必要ありません。

  RPC コマンドでストアド・プロシージャを呼び出す場合、パラメータは宣言されているデータ型で渡されます。サーバは、パラメータを文字フォーマットから宣言されているデータ型に変換する必要がありません。

- Open Server レジスタード・プロシージャを実行する唯一の手段です。

  Open Server レジスタード・プロシージャを使用すると、Open Client および Open Server で比較的簡単に分散アプリケーションを開発できます。レジスタード・プロシージャは、Open Server アプリケーション・コード内の関数でも、クライアント・アプリケーションによって作成され、実行時にクライアント・ノーティフィケーション (通知) イベントをトリガするためだけに存在する特別なタイプのプロシージャでもかまいません。後者のタイプは、クライアント・アプリケーションが Open Server システム・レジスタード・プロシージャの sp_regcreate を呼び出したときに、作成されます。

  - レジスタード・プロシージャとして呼び出せる C 関数を定義する方法については、『Open Server Server-Library/C リファレンス・マニュアル』を参照してください。
  - Client-Library アプリケーションが Open Server でレジスタード・プロシージャを作成する方法については、『Open Server Server-Library/C リファレンス・マニュアル』の「sp_regcreate」のリファレンス・ページを参照してください。
  - Client-Library アプリケーションがレジスタード・プロシージャ・ノーティフィケーションを受信する方法については、『Open Client Client-Library/C リファレンス・マニュアル』の「レジスタード・プロシージャ」の項を参照してください。

RPC と *execute* 言語コマンドの比較

ストアド・プロシージャは、RPC コマンドからでも execute 言語文からでも実行できます。リモート・プロシージャ・コールには、execute 文と比べて次のような利点があります。

- RPC コマンドは、Adaptive Server Enterprise ストアド・プロシージャまたは Open Server レジスタード・プロシージャを実行するのに使用できます。

  Transact-SQL 言語コマンドは、Adaptive Server Enterprise ストアド・プロシージャの実行にしか使用できません (Open Server アプリケーションが Transact-SQL を理解できる場合を除く)。

- RPC コマンドは、ストアド・プロシージャのパラメータをネイティブ・データ型で渡します。これに対して、execute 文は、言語コマンドのテキスト内で文字フォーマットでパラメータを渡します。この違いのため、RPC 方式は execute 方式よりも高速であり、効率が良いといえます。RPC 方式の場合、アプリケーション・プログラムもサーバも、ネイティブ・データ型とこれに対応する文字フォーマットの間で変換を行う必要がないからです。

- ストアド・プロシージャを言語コマンドではなく RPC コマンドで呼び出すと、プロシージャのリターン・パラメータは、より簡単に、より高速になります。

  RPC コマンドの場合、リターン・パラメータ値はパラメータ結果セットとして自動的にアプリケーションに対して使用可能になります (リターン・パラメータは、最初に ct_param または ct_setparam で RPC コマンドに追加するときに、そのことを指定する必要があります)。

  一方、execute 文の場合、リターン・パラメータ値は、言語コマンドがローカル変数を宣言し、リターン・パラメータ用としてこれらの変数 (定数ではない) を渡した場合にだけ使用できます。言語コマンドには複数の SQL 文が含まれるので、言語コマンドが実行されるたびに余分な解析が必要になります。

Client-Library カーソル・コマンド

カーソルは、アプリケーションが select 文に付加する記号名です。カーソルは、select の結果セットを操作するオペレーションをサポートします。カーソル処理の一覧については、「カーソルの概要」(101 ページ) を参照してください。

Client-Library カーソルは、ct_cursor または ct_dynamic カーソル宣言コマンドによって作成されるカーソルです。

Client-Library カーソル・コマンドの構築

アプリケーションにおける Client-Library カーソル・コマンドの使い方については、「第 7 章 Client-Library カーソルの使い方」で説明します。標準的な呼び出しシーケンスについては、「Client-Library カーソルの使い方」(106 ページ) を参照してください。

Client-Library カーソルの使用が適する場合

1 つのサーバ接続だけを使用して同時に複数のコマンドを処理するような場合には、Client-Library カーソルを使用します。

Client-Library カーソル・オープン・コマンドは、アプリケーションがまだローを取り出している間に、同じ接続で新しいコマンドを送信できる唯一のコマンド・タイプです。ほかのタイプのコマンドを送信したあとは、アプリケーションは、そのコマンドの結果を完全に処理し終わるまでは、同じ接続で別のコマンドを送信することはできません。アプリケーションの設計上、この機能が必要な場合、Client-Library カーソル・コマンドを使用するしか方法はありません。「Client-Library カーソルの利点」(104 ページ) と「接続規則とコマンド規則」(31 ページ) を参照してください。

カーソルは、1 つの select 文だけを実行するように宣言してください。「手順 1：カーソルを宣言する」(108 ページ) を参照してください。

Client-Library カーソルの使用が適さない場合

カーソルの場合、言語コマンドまたは RPC コマンドを使用して select 文を実行するのに比べて、パフォーマンスは低下します。この違いは、カーソルが、通常ロー結果セットとは異なり、カーソル・ローを取得するのに、内部 Client-Library カーソル・フェッチ・コマンドを必要とするためです。つまり、カーソル・オープン・コマンドの結果の処理には、より多くのネットワークの往復が必要です。「手順 2：カーソル・ロー数を設定する」(113 ページ) を参照してください。カーソル処理では、余分な Adaptive Server Enterprise 内部のオーバヘッドが発生します。

動的 SQL コマンド

動的 SQL は、Client-Library の ct_dynamic ルーチンによって起動されるコマンドを使用して実行時に SQL 文を生成し、準備し、実行するプロセスです。

動的 SQL コマンドの構築

アプリケーションにおける Client-Library カーソル・コマンドの使い方については、「第 8 章 動的 SQL コマンドの使い方」で説明します。標準的な呼び出しシーケンスについては、「準備実行方式のプログラム構造」(126 ページ) を参照してください。

動的 SQL コマンドの使用が適する場合

動的 SQL 準備文コマンドは、アプリケーションがコマンドを実行するのに必要な、入力およびコマンドの結果のフォーマットについてサーバに問い合わせできる唯一のコマンド・タイプです。

- ct_dynamic 入力記述コマンドは、文を実行するのに必要なパラメータの数とフォーマットを送信するようにサーバに指示します。詳細については、「手順 2：コマンド入力の記述を取得する」(128 ページ) を参照してください。
- ct_dynamic 出力記述コマンドは、文が返す結果カラムの数とフォーマットを送信するようにサーバに指示します。詳細については、「手順 3：コマンド出力の記述を取得する」(129 ページ) を参照してください。

動的 SQL の使用が適さない場合

一般に、動的 SQL は、「動的 SQL の利点」(122 ページ) に挙げたような特別な利点を必要としないアプリケーションには、使用しないでください。動的 SQL コマンドは、言語コマンドより多くのオーバヘッドを要します。また、動的 SQL コマンドは内部的にはテンポラリ・ストアド・プロシージャとして実装されるため、Adaptive Server Enterprise の tempdb データベース内でリソースの競合問題を引き起こすことがあります。

「動的 SQL の制限事項」(122 ページ) と「動的 SQL に代わる方法」(124 ページ) を参照してください。

メッセージ・コマンド

メッセージ・コマンドは、カスタム Open Server アプリケーションで使用できます。Adaptive Server Enterprise はメッセージ・コマンドをサポートしません。クライアント・アプリケーション・プログラマの観点からは、メッセージ・コマンドは、名前ではなく番号で呼び出される RPC コマンドと同じです。

アプリケーションは、*type* を CS_MESSAGE_CMD に、**buffer* をメッセージ・コマンドの識別子を含む CS_INT 変数のアドレスに設定して、ct_command を呼び出すことによって、メッセージ・コマンドを起動します。次に例を示します。

```
CS_INT      msg_id;
if (ct_command(cmd, CS_MSG_CMD, (CS_VOID *)&msg_id,
CS_UNUSED, CS_UNUSED)
!= CS_SUCCEED)
{
fprintf(stderr, "ftclient:ct_command(MSG_CMD) failed.¥n");
return CS_FAIL;
}
```

メッセージ識別子は、クライアント・アプリケーションと Open Server アプリケーションの両方に認識されていなければなりません。一般に、サーバが応答するメッセージ・コマンドは、共有ヘッダ・ファイルに定義されています。Sybase は CS_USER_MSGID から CS_USER_MAX_MSGID までの範囲 (これらの値を含む) のメッセージ識別子をユーザ用として予約しています。

メッセージ・コマンドは、パラメータを持つことができます。パラメータは、ct_param または ct_setparam によって代入されます。パラメータが名前または位置のどちらで渡されるかは、Open Server がどのようにコーディングされているかによって決まります。

「基本ループの構造」(86 ページ) の説明に従って、メッセージ・コマンドの結果を標準結果ループで処理するように、アプリケーションをコーディングします。メッセージ・コマンドは、他の結果タイプとは別に、メッセージ結果 (CS_MSG_RESULT という結果タイプ) を返すことができます。「メッセージ結果の処理」(96 ページ) を参照してください。

メッセージ・コマンドの使用が適する場合

メッセージ・コマンドは、カスタム Open Server アプリケーションのクライアント・インタフェースの設計における RPC コマンドに代わるコマンドです。メッセージ・コマンドは、文字列の RPC 名ではなく整数識別子を使用します。メッセージ・コマンドには、Open Server レジスタード・プロシージャの固定パラメータ・リストもありません。

Open Server コードでは、メッセージ・コマンドは、メッセージ・イベント・ハンドラによって処理されます。詳細については、『Open Server Server-Library/C リファレンス・マニュアル』を参照してください。

メッセージ・コマンドの使用が適さない場合

Adaptive Server Enterprise はメッセージ・コマンドをサポートしません。

パッケージ・コマンド

パッケージ・コマンドは、CICS 上の Open Server 接続でだけサポートされます。それ以外の場合は、パッケージ・コマンドは RPC コマンドと変わりがありません。

データ送信コマンド

ct_command(CS_SEND_DATA) で起動されるデータ送信コマンドは、text または image カラム値をまとまりでアップロードするのに使用します。

アプリケーションでのデータ送信コマンドの使い方については、『Open Client Client-Library/C リファレンス・マニュアル』の「text および image データの処理」の項を参照してください。

データ送信コマンドの使用が適する場合

Adaptive Server Enterprise クライアント・アプリケーションの場合、データ送信コマンドは、大きな *text* または *image* カラム値を 1 つのまとまりとして一度にアップロードする唯一の方法です。アプリケーションが 1 つの連続メモリ・バッファに入りきらないほど大きい text または image 値をアップロードする場合、データ送信コマンドはアップロードを実行できる唯一の方法です。

連続メモリ・バッファに入りきる大きさの *text* または *image* カラム値の場合、アプリケーションは、insert 言語コマンドに値を埋め込むことによって、効率の良いパフォーマンスを得られます。この方法の詳細については、『Open Client Client-Library/C リファレンス・マニュアル』の「text および image データの処理」の項を参照してください。

データ送信コマンドの使用が適さない場合

一般に、データ送信コマンドは、カスタム Open Server アプリケーションのクライアント・インタフェースを設計する場合は、使用しないでください。データ送信コマンドの Open Server アプリケーション処理は、非常に複雑です。サーバが大きい値をまとまりでアップロードする場合には、メッセージ、RPC、言語コマンドを複数回呼び出して値をアップロードするようなインタフェースを設計する方法を使用してください。たとえば、メッセージ・コマンドの場合、1 つのメッセージ・コマンド識別子で、アップロード・オペレーションの始めを指示し、別のメッセージ・コマンド識別子で、データ値のまとまりが (パラメータとして) 入っているコマンドを指示するようにします。

第 6 章

結果処理コードの書き方

この章では、Client-Library の結果処理モデルについて説明します。この章の内容は、次のとおりです。

トピック名	ページ
結果のタイプ	85
基本ループの構造	86
通常ロー結果の処理	88
カーソル結果の処理	89
パラメータ結果の処理	92
リターン・ステータス結果の処理	93
計算結果の処理	94
メッセージ結果の処理	96
記述結果の処理	96
フォーマット結果の処理	97
コマンド・ステータスを表示する result_type の値	98
ct_results の最後のリターン・コード	99

結果のタイプ

アプリケーションは、サーバにコマンドを送信したあと、コマンドによって生成された結果を処理します。結果のタイプには、次のようなものがあります。

- 通常ロー結果 − サーバが select 文を処理したときに返されるロー
- カーソル・ロー結果 − サーバが Client-Library の ct_cursor カーソル・オープン・コマンドを処理したときに返されるロー
- パラメータ結果 − 表現できるフェッチ可能なデータ
  - Adaptive Server Enterprise のストアド・プロシージャのリターン・パラメータの出力値
  - Open Server のレジスタード・プロシージャのリターン・パラメータの出力値

    - 更新された *text*/*image* カラムの新しいタイムスタンプ値 (ct_command データ送信コマンドの結果を処理したときだけ表示される)
    - ブラウズ・モードの update 文が含まれている言語コマンドで更新されたローの新しいタイムスタンプ値
- ストアド・プロシージャのリターン・ステータス結果 － Adaptive Server Enterprise のストアド・プロシージャまたは Open Server のレジスタード・プロシージャからの戻り値
- 計算ロー結果 － サーバが compute by 句のある select 文を処理したときに返される中間ロー
- メッセージ結果 － メッセージ・コマンドの結果を処理しているときに、Open Server アプリケーションのメッセージ・コマンド・ハンドラから返されるメッセージ ID
- 記述結果 － 準備された動的 SQL 文の入力パラメータまたは結果カラムのフォーマットを記述している情報結果
- フォーマット結果 － Open Server ゲートウェイ・アプリケーションが実データが到着する前に通常ローと計算ローのフォーマットを取得するのに使用する情報結果

1 つのコマンドが複数のタイプの結果を生成することがあります。たとえば、ストアド・プロシージャを実行する言語コマンドは、複数の通常ローと計算ロー結果セット、1 つのパラメータ結果セット、1 つのリターン・ステータス結果セットを生成できます。このため、サーバによって生成される可能性があるすべてのタイプの結果を処理するように、アプリケーションをコーディングすることが重要です。

アプリケーションがすべての結果タイプを処理する最も簡単な方法は、次に説明するようなループで結果を処理する方法です。

基本ループの構造

ほとんどの同期 Client-Library プログラムは、ct_results で制御されるループを使用して結果を処理します。ループ内で、ct_results の *result_type* パラメータの値の指示に従って、現在処理に使用可能な結果のタイプが切り替わります。異なるタイプの結果には、異なるタイプの処理が必要です。

result_type は、たとえば、insert または delete コマンドのように、結果を返さないサーバ・コマンドの出力を表示するのにも使用します。

ほとんどの同期アプリケーションは、次に示すようなプログラム構造を使用して、結果を処理します。

```
while ct_results returns CS_SUCCEED
    (optional) ct_res_info to get current
        command number
    switch on result_type
        /*
        ** Values of result_type that indicate
        ** fetchable results:
        */
        case CS_COMPUTE_RESULT...
        case CS_CURSOR_RESULT...
        case CS_PARAM_RESULT...
        case CS_ROW_RESULT...
        case CS_STATUS_RESULT...
        /*
        ** Values of result_type that indicate
        ** non-fetchable results:
        */
        case CS_COMPUTEFMT_RESULT...
        case CS_MSG_RESULT...
        case CS_ROWFMT_RESULT...
        case CS_DESCRIBE_RESULT...
        /*
        ** Other values of result_type:
        */
        case CS_CMD_DONE...
            (optional) ct_res_info to get the
                number of rows affected by
                the current command
        case CS_CMD_FAIL...
        case CS_CMD_SUCCEED...
    end switch
end while
switch on ct_results' final return code
    case CS_END_RESULTS...
    case CS_CANCELED...
    case CS_FAIL...
end switch
```

通常ロー結果の処理

通常ロー結果セットは、サーバ上で Transact-SQL select 文を実行したときに、生成されます。

通常ロー結果セットには、0 または 1 以上の表形式データのローが含まれます。

アプリケーションは、一般に、次のルーチンを呼び出して通常ロー結果セットを処理します。

- ct_res_info − 現在の結果セットに関する情報を返します。アプリケーションが ct_res_info を使用するのは、現在の結果セット内のカラム数を取得するときです。ただし、ct_res_info は他のタイプの情報 (たとえば、現在のコマンドによって処理されたローの数など) も返します。
- ct_describe − 現在の結果セット内の特定の結果項目に関する情報を返します。一般に、アプリケーションは、各結果項目につき 1 回ずつ ct_describe を呼び出してから、各結果項目をプログラム変数にバインドする必要があります。
- ct_bind − 結果項目をプログラム変数にバインドします。バインドにより、結果項目とデータ領域が対応されます。
- ct_fetch − バインドされた変数に結果データをコピーします。

「バインド」は、結果項目をプログラム・データ領域と対応させるプロセスです。「フェッチ」は、結果項目のデータ・インスタンスを取得するプロセスです。結果項目にバインドが指定されていると、フェッチによって、その項目のデータ・インスタンスがプログラム・データ領域にコピーされます。

ほとんどの同期アプリケーションは、次に示すようなプログラム構造を使用して、通常ロー結果セットを処理します。

```
case CS_ROW_RESULT
    ct_res_info(CS_NUMDATA) to get the number of columns
    for each column:
        ct_describe to get a description of the column
        ct_bind to bind the column to a program variable
    end for
    while ct_fetch returns CS_SUCCEED or CS_ROW_FAIL
        if CS_SUCCEED
            process the row
        else if CS_ROW_FAIL
            handle the row failure
        end if
    end while
    switch on ct_fetch's final return code
        case CS_END_DATA...
        case CS_CANCELED...
        case CS_FAIL...
    end switch
end case
```

カーソル結果の処理

カーソル・ロー結果セットは、アプリケーションが Client-Library カーソル・オープン・コマンドを実行するときに生成されます。

注意 アプリケーションが Transact-SQL open 文を含んでいる言語コマンドを実行したときには、カーソル・ロー結果セットは生成されません。open 文が Adaptive Server Enterprise 言語カーソルをオープンし、オープンされた言語カーソルは、アプリケーションが Transact-SQL fetch 文を実行するたびに、通常ローを返します。「言語カーソルと Client-Library カーソルの比較」(102 ページ) を参照してください。

カーソル・ロー結果セットには、0 または 1 以上の表形式データのローが含まれます。

一般に、アプリケーションは、コマンドをサーバに送信する場合、最初のコマンドの結果の処理が完全に終了していることを ct_results が (CS_END_RESULTS、CS_CANCELED、または CS_FAIL を返すことによって) 示すまで、同じ接続で別のコマンドを送信することはできません。

ただし、ct_results がカーソル結果を示した場合、この規則は当てはまりません。この場合、アプリケーションはカーソル結果セットを処理している間も、ct_cursor と ct_send を呼び出して、カーソル更新コマンド、カーソル削除コマンド、カーソル・クローズ・コマンドを送信できます。異なる CS_COMMAND 構造体を使用すれば、同じ接続でサーバに新しいコマンドを送信することもできます。「Client-Library カーソルの利点」(104 ページ) を参照してください。

アプリケーションは、カーソル結果セットを処理しながら、ct_res_info、ct_describe、ct_bind、ct_fetch の他に、ct_keydata、ct_cursor、ct_param、ct_send、ct_results、ct_cancel を呼び出すこともできます。

ほとんどの同期アプリケーションは、次に示すようなプログラム構造を使用して、カーソル結果セットを処理します。

```
case CS_CURSOR_RESULT
ct_res_info(CS_NUMDATA) to get the number of columns
     for each column:
         ct_describe to get a description of the column
         ct_bind to bind the column to a program variable
     end for
    while ct_fetch returns CS_SUCCEED or CS_ROW_FAIL
           and cursor has not been closed
         if CS_SUCCEED
             process the row
         else if CS_ROW_FAIL
             handle the row failure
         end if

       /* For update or delete only:                    */
         if target row is not the row just fetched
```

```
                ct_keydata to specify the target row key
            end if
            /* End for update or delete only                */

          /* To send a nested cursor update, delete, or
close command: */
            ct_cursor to initiate the cursor command
            /* For updates/deletes whose "where" clause
contains variables */
            ct_param or ct_setparam for each parameter
            /* End for updates/deletes whose ... */
            ct_send to send the command
            while ct_results returns CS_SUCCEED
                (...process results...)
            end while
            /* End to send a nested cursor command */

      end while
       switch on ct_fetch's final return code
            case CS_END_DATA...
            case CS_CANCELED...
            case CS_FAIL...
        end switch
       if cursor was closed
            break out of outer ct_results loop
        end if

    end case
```

ct_results 呼び出しは、ct_fetch ループとより大きな ct_results ループ (上記の例には含まれていません) 内でネストされます。

ネストされたカーソル更新コマンドまたはカーソル削除コマンドの場合、ネストされたコマンドの結果が完全に処理し終わっていることを内側の ct_results が (CS_END_RESULTS、CS_FAIL、または CS_CANCELED を返すことによって) 示したあと、後続の ct_results 呼び出しが最初のカーソル・コマンドによって生成された結果を処理します。

ネストされたカーソル・クローズ・コマンドの場合、カーソルがクローズしたあとに、結果は残りません。この場合には、アプリケーションは、ネストされたカーソル・クローズ・コマンドの結果を処理し終わってから、外側の ct_results ループから出ます。

カーソル・オープン・コマンドから返されたカーソル・ローをキャンセルするのに、アプリケーションは、*type* を CS_CANCEL_CURRENT に設定して ct_cancel を呼び出すことができます。ただし、ネストされたカーソル・クローズ・コマンドでカーソルをクローズする方が効率的です。CS_CANCEL_CURRENT ct_cancel 呼び出しは、不要なローを取得して廃棄します (これは、すべてのバインドをクリアして、ct_fetch が CS_END_DATA を返すまで ct_fetch を呼び出すのと同じことです)。

注意　カーソル・アプリケーションでは、開いているカーソルのある接続には CS_CANCEL_CURRENT 以外のタイプのキャンセルを使用しないでください。CS_CANCEL_ALL または CS_CANCEL_ATTN を使用すると、接続のカーソルが未定義のステータスになる可能性があります。アプリケーションは、カーソルをキャンセルせずに、クローズするだけの場合もあります。

スクロール可能なカーソルの結果の処理

スクロール可能なカーソルの結果を処理するプログラム構造は、通常のカーソルのプログラム構造とほぼ同じです。主な違いは、CS_FALSE オプションを使用すると ct_scroll_fetch が CS_SCROLL_CURSOR ENDS を返す点です。この部分は次のように記述されます。

```
        end while
         switch on ct_scroll_fetch's final return code
              case CS_SCROLL_CURSOR_ENDS...
          end switch
         if cursor was closed
              break out of outer ct_results loop
          end if

    end case
```

注意　ct_scroll_fetch は、有効な戻り値として CS_END_DATA を返しません。

注意　一連のオペレーションにより、カーソルが結果セットの境界の外へ出てしまった場合、警告メッセージが生成されます。たとえば、現在のカーソル位置に対して大きいオフセットで CS_PREV、CS_NEXT、CS_ABSOLUTE、または CS_RELATIVE 呼び出しを連続して使用すると、カーソルが結果セットの境界の外へ出てしまいます。警告メッセージには、エラーが発生したことは示されません。『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。

パラメータ結果の処理

パラメータ結果セットは、パラメータの 1 つのローで構成されます。

次のようなタイプのデータをパラメータ結果セットの形式でアプリケーションに返すことができます。

- リターン・パラメータ値

  Adaptive Server Enterprise のストアド・プロシージャまたは Open Server のレジスタード・プロシージャは、出力パラメータ・データを返すことができます。CS_PARAM_RESULT 結果セットには、プロシージャ・コードの設定に従って、プロシージャのパラメータの新しい値が入っています。アプリケーションがストアド・プロシージャまたはレジスタード・プロシージャをどのようにして実行するかについては、「RPC コマンド」(74 ページ) を参照してください。

- ブラウズ・モード・タイムスタンプ値

  ブラウズ・モードは、対話型アプリケーションが取得されたローのアドホック・ローを更新するときに使用できるスキームです。ブラウズ・モードに関係するテーブルには、データへの同時アクセスを制御するために、timestamp カラムが必要です。クライアント・アプリケーションがブラウズ・モードの update 文を実行したあと、Adaptive Server Enterprise は、更新されたローの新しい timestamp 値が入っているパラメータ結果セットを返します。詳細については、『Open Client Client-Library/C リファレンス・マニュアル』の「ブラウズ・モード」の項を参照してください。

- text または image カラム・タイムスタンプ

  クライアント・アプリケーションがデータ送信コマンドで text または image カラムを更新したあと、Adaptive Server Enterprise は、そのカラムの新しいテキスト・タイムスタンプをパラメータ結果セットとして返します。詳細については、『Open Client Client-Library/C リファレンス・マニュアル』の「text および image データの処理」の項を参照してください。

- メッセージ結果パラメータ

  メッセージ結果セットはメッセージ識別子で構成されます (「メッセージ結果の処理」(96 ページ) を参照してください)。メッセージ結果セットのすぐ後ろに、メッセージ結果に伴うパラメータ値が入っているパラメータ結果セットが付くことがあります。

アプリケーションは、ct_res_info、ct_describe、ct_bind、ct_fetch を呼び出して、パラメータ結果セットを処理します。

ほとんどの同期アプリケーションは、次に示すようなプログラム構造を使用して、パラメータ結果セットを処理します。

```
case CS_PARAM_RESULT
    ct_res_info(CS_NUMDATA) to get the number of parameters
    for each parameter:
        ct_describe to get a description of the parameter
```

```
        ct_bind to bind the parameter to a variable
    end for

  while ct_fetch returns CS_SUCCEED or CS_ROW_FAIL
        if CS_SUCCEED
            process the row of parameters
        else if CS_ROW_FAIL
            handle the failure
        end if
    end while

  switch on ct_fetch's final return code
        case CS_END_DATA...
        case CS_CANCELED...
        case CS_FAIL...
    end switch
end case
```

リターン・ステータス結果の処理

リターン・ステータス結果セットは、ストアド・プロシージャの実行によって生成されます。すべてのストアド・プロシージャは、ステータス番号を返します。『ASE リファレンス・マニュアル』の return コマンドの説明を参照してください。

リターン・ステータス結果セットは、リターン・ステータスが入っている 1 つのローで構成されます。

アプリケーションは、ct_bind と ct_fetch を呼び出して、リターン・ステータスを処理します。

ほとんどの同期アプリケーションは、次に示すようなプログラム構造を使用して、リターン・ステータス結果セットを処理します。

```
case CS_STATUS_RESULT
    ct_bind to bind the status to a program variable
    while ct_fetch returns CS_SUCCEED or CS_ROW_FAIL
        if CS_SUCCEED
            process the return status
        else if CS_ROW_FAIL
            handle the failure
        end if
    end while
    switch on ct_fetch's final return code
        case CS_END_DATA...
        case CS_CANCELED...
        case CS_FAIL...
    end switch
end case
```

計算結果の処理

計算結果セットは、compute 句が含まれている Transact-SQL select 文の実行によって生成されます。compute 句は、bylist の値が変わるたびに、計算結果セットを生成します。計算結果セットは、compute 句内のロー集約数に等しいカラム数が入っている 1 つのローで構成されます。

たとえば、次のようなクエリがあるとします。

```
select type, price from titles
 where price > $12 and type like "%cook"
 order by type, price compute sum(price) by type
```

このクエリは通常ロー (type カラムと price カラムのあるロー) を返します。通常ローが混ざっているこのクエリは、通常ロー結果内で type の値が変わるたびに、計算結果セットを返します。各計算結果セットには、sum(price) 式の 1 つのカラムの 1 つのローが入っています。

compute 句が含まれているクエリの例については、『ASE リファレンス・マニュアル』を参照してください。

アプリケーションは、計算ロー結果を処理しながら、ct_res_info、ct_describe、ct_bind、ct_fetch の他に、ct_compute_info を呼び出すこともできます。ct_compute_info はさまざまな計算ロー情報を用意します。ct_compute_info から使用できる情報には、次のようなものがあります。

- 計算ローの計算 ID

  クエリには複数の compute 句を指定できます。ct_compute_info(CS_COMP_ID) は、計算結果セットを生成した compute 句の番号を取得します。計算ロー ID の 1 は、クエリの最初の compute 句に対応しています。

- 計算 bylist

  計算 bylist は、compute 句の by キーワードに続くカラムのリストです。アプリケーションでは、bylist は CS_SMALLINT 値の配列で表現されます。各値は select リスト内のカラムの位置を示します。次に例を示します。

  ```
  select dept, name, year, sales from employee
            order by dept, name, year
           compute count(name) by dept, name
  ```

  この例の場合、bylist 値は、1 と 2 であり、select リストの dept と name の位置に対応しています。

  ct_compute_info(CS_BYLIST_LEN) は bylist の長さを返し、ct_compute_info(CS_BYLIST) はアプリケーションが割り付けた配列に bylist カラム番号を挿入します。

- 計算ロー select リストのカラム ID

  select リストのカラム ID は、計算ローのカラムごとに用意されます。select リストのカラム ID は、計算ロー・カラムを抽出したカラムの selest リスト内の位置です。たとえば、次のクエリは sum(price) 式の 1 つのカラムが入っている計算ローを返します。

  ```
  select type, price from titles
   where price > $12 and type like "%cook"
   order by type, price compute sum(price) by type
  ```

  対応する select リストのカラム ID は 2 です。2 は select リスト内の price カラムの位置です。

  ct_compute_info は、*type* を CS_COMP_COLID に、*colnum* を計算カラム番号に設定して呼び出すと、計算カラム ID を取得します。

- 計算カラム演算子

  ct_compute_info は、*type* を CS_COMP_OP に、*colnum* を計算カラム番号に設定して呼び出すと、カラム値を計算したときの演算子を指示する記号定数を取得します。これらの演算子については、『Open Client Client-Library/C リファレンス・マニュアル』の「ct_compute_info」のリファレンス・ページを参照してください。

ほとんどの同期アプリケーションは、次に示すようなプログラム構造を使用して、計算結果セットを処理します。

```
case CS_COMPUTE_RESULT
    (optional)ct_compute_info to get bylist length,
    bylist, or compute row id
    ct_res_info(CS_NUMDATA) to get the number of columns
    for each column:
        ct_describe to get a description of the column
        ct_bind to bind the column to a program variable
        (optional:ct_compute_info to get the compute
            column id or the aggregate operator for the
            compute column)
    end for
    while ct_fetch returns CS_SUCCEED or CS_ROW_FAIL
        if CS_SUCCEED
            process the compute row
        else if CS_ROW_FAIL
            handle the failure
        end if
    end while
    switch on ct_fetch's final return code
        case CS_END_DATA...
        case CS_CANCELED...
        case CS_FAIL...
    end switch
end case
```

メッセージ結果の処理

すべてのタイプのサーバはメッセージ結果を返すことができます。

メッセージ結果セットには、フェッチできる結果を含みません。代わりに、メッセージは ID を持っています。アプリケーションは、ct_res_info(CS_MSGTYPE) を呼び出して、この ID を取得できます。

1 ～ 32,767 の範囲のメッセージ ID は、Adaptive Server Enterprise と Sybase 内部用として予約されています。

アプリケーション定義のメッセージ ID は、CS_USER_MSGID ～ CS_USER_MAX_MSGID の範囲にしてください。

パラメータ値は、メッセージと対応されると、メッセージ結果セットに続く別のパラメータ結果セットとして返されます。「パラメータ結果の処理」(92 ページ) を参照してください。

注意 メッセージ結果セットはサーバ・メッセージと同じではありません。サーバ・メッセージは、エラー状態や問題のあるサーバ状態に応じて生成され、通常は、アプリケーションのサーバ・メッセージ・コールバック内で処理されます。「第 4 章 エラーおよびメッセージの処理」を参照してください。

アプリケーションは、ct_res_info を呼び出して、メッセージ ID を取得します。

ほとんどの同期アプリケーションは、次に示すようなプログラム構造を使用して、メッセージ結果セットを処理します。

```
case CS_MSG_RESULT
    ct_res_info to get the message ID
    code to handle the message ID
end case
```

記述結果の処理

記述結果セットには、フェッチできるデータはありません。この結果セットは、動的 SQL 入力記述または出力記述コマンドの結果として返される記述情報の有無を示します。

「手順 2：コマンド入力の記述を取得する」(128 ページ) と「手順 3：コマンド出力の記述を取得する」(129 ページ) を参照してください。

アプリケーションは、ct_describe、ct_dyndesc、ct_dynsqlda を呼び出すことによって、この情報を取得できます。「パラメータ記述の処理」(128 ページ) と「カラム記述の処理」(130 ページ) を参照してください。

ほとんどのアプリケーションは、次に示すようなプログラム構造を使用して、記述結果セットを処理します。

```
case CS_DESCRIBE_RESULT
    ct_res_info to get the number of columns
    for each column:
        ct_describe or ct_dyndesc to get a description
    end for
end case
```

フォーマット結果の処理

通常、通常ローと計算ローの結果セットのフォーマット情報は、アプリケーションが結果セットを処理している間だけ使用できます。この場合、アプリケーションは、ct_res_info を呼び出して結果セット内の項目数を取得し、ct_describe を呼び出して各項目の内容を取得し、ct_compute_info を呼び出して計算情報を取得します。

このメカニズムは、ほとんどのアプリケーションで機能します。ただし、一部のアプリケーションでは、結果セットのフォーマット情報を得てから、結果セットを処理してください。このタイプのアプリケーションの例としては、Adaptive Server Enterprise 結果を再パッケージしてから Sybase 以外のクライアントに送信するゲートウェイ・アプリケーションがあります。

Client-Library は、「フォーマット結果」の形式でフォーマット情報を前もってアプリケーションに使用可能にします。フォーマット結果には次の 2 つのタイプがあります。

フォーマット結果セットには、フェッチできる結果はありません。代わりに、アプリケーションは、ct_results がフォーマット結果を表示してから、ct_res_info、ct_describe、ct_compute_info を呼び出して、フォーマット情報を取得できます。

フォーマット結果を受信するには、アプリケーションは CS_EXPOSE_FMTS プロパティを CS_TRUE に設定する必要があります。

アプリケーションは、ct_describe と ct_compute_info を呼び出して、フォーマット情報を取得できます。

ゲートウェイ・アプリケーションは、次に示すようなプログラム構造を使用して、フォーマット結果を処理します。

```
case CS_ROWFMT_RESULT
    ct_res_info(CS_NUMDATA) to get the number of columns
    for each column:
        ct_describe to get a column description
        send the information on to the gateway client
    end for
end case
```

```
case CS_COMPUTEFMT_RESULT
    ct_res_info to get the number of columns
    for each column:
        ct_describe to get a column description
        (if required:
            ct_compute_info for compute information
        end if required)
        send the information on to the gateway client
    end for
end case
```

コマンド・ステータスを表示する *result_type* の値

ct_results は、処理に使用可能な結果セットのタイプを表示するほか、*result_type* を次の値に設定して、コマンド処理のステータスを示します。

- CS_CMD_DONE － 「論理コマンド」の結果が完全に処理し終わったことを示します。「論理コマンド」については、「論理コマンド」(99 ページ) を参照してください。
- CS_CMD_SUCCEED － Transact-SQL の insert コマンドや delete コマンドなど、データを返さないコマンドが正常に実行されたことを示します。
- CS_CMD_FAIL － エラーのために、サーバがサーバ・コマンドの実行に失敗したことを示します。たとえば、言語コマンドのテキストに、構文エラーがあったり、存在しないオブジェクトを参照していたりするような場合です。ほとんどの場合、サーバはエラーを記述したサーバ・メッセージを返します。

  Client-Library コマンドは複数のサーバ・コマンドを実行できるので、アプリケーションは次のどちらかを行う必要があります。

  - ct_results を呼び出して、元の Client-Library コマンドに含まれている他のサーバ・コマンドが生成した結果を処理し続ける。
  - ct_cancel(CS_CANCEL_ALL) を呼び出して、Client-Library コマンドをキャンセルして、結果を廃棄する。

論理コマンド

ct_results は、*result_type* を CS_CMD_DONE に設定して、論理コマンドの結果の処理が完了したことを示します。「論理コマンド」は、ct_command、ct_dynamic、または ct_cursor を使用して定義される任意のコマンドです。ただし、次のような例外があります。

- ストアド・プロシージャ内でデータを返す各 Transact-SQL select 文は、論理コマンドです。ストアド・プロシージャ内のその他の Transact-SQL 文は、値をローカル変数に代入する select 文も含めて、論理コマンドとは見なされません。
- 動的 SQL コマンドによって実行された各 Transact-SQL 文は個別の論理コマンドです。
- 言語コマンドのテキスト内の各 Transact-SQL 文は、論理コマンドです。

論理コマンドと Client-Library コマンドは、同じではありません。Client-Library コマンドは、サーバ上で複数の論理コマンドを実行できます。たとえば、ストアド・プロシージャは、データを返す複数の select 文を実行でき、それらの各文が 1 つの論理コマンドに相当します。論理コマンドは 1 つまたは複数の結果セットを生成できます。たとえば、select 文は複数の通常ローと計算結果セットを返すことができます。

ct_results の最後のリターン・コード

コマンドのすべての結果を処理してから、ct_results からの最後のリターン・コードをチェックして、エラーが示されているかどうかを調べるように、アプリケーションをコーディングする必要があります。

最後のリターン・コード値は、次のいずれかです。

- CS_END_RESULT － 正常なループ終了を示します。
- CS_CANCELED － 結果がキャンセルされたこと、つまり、結果を処理しているときに、ct_cancel(CS_CANCEL_ALL) または ct_cancel(CS_CANCEL_ATTN) が呼び出されたことを示します。
- CS_FAIL － 通信障害またはメモリ不足など、重大なクライアント側のエラーまたはネットワーク・エラーが発生したことを示します。

第 7 章

Client-Library カーソルの使い方

この章では、Client-Library カーソルについて説明します。この章の内容は、次のとおりです。

トピック名	ページ
カーソルの概要	101
言語カーソルと Client-Library カーソルの比較	102
Client-Library カーソルの使用が適する場合	104
Client-Library カーソルの使い方	106
Client-Library のカーソル・プロパティ	120

カーソルの概要

カーソルは、アプリケーションが select 文に付加する記号名です。カーソルでオペレーションを行うことによって、select 文を実行し、その結果セットを操作できます。

カーソルは次のオペレーションをサポートします。

- 宣言 – 名前を与え、クエリを定義して、新しいカーソルを作成します。
- カーソル・ロー数の設定 – 各フェッチ・オペレーションで返される結果テーブルのロー数を指定します。
- オープン – カーソルのクエリを実行し、フェッチ・オペレーションのためにカーソルを準備します。
- フェッチ – カーソルからローを検索します。カーソルは前もってオープンしていなければなりません。フェッチが実行されるたびに、クエリの結果テーブルから単一のローが取得されます。この動作は、スクロール可能カーソルと非スクロール可能カーソルの両方に適用されます。特定の条件下では (“set cursor rows” 操作で定義されている)、1 回のフェッチ呼び出しで返されるローの数が通常より多くなることがあります。
- 更新 – フェッチされたローの値を変更します。更新は、ローを選択したテーブルに影響を与えます。
- 削除 – フェッチされたローを基本となるテーブルから削除します。
- クローズ – カーソルを再オープンまたは割り付け解除できる状態にします。
- 割り付け解除 – カーソルのリソースを解放します。

Adaptive Server Enterprise クライアント・アプリケーションでは、カーソルは、言語コマンドまたは ct_cursor コマンドで作成して、操作できます。Transact-SQL 言語コマンドを使用して作成されたカーソルを「言語カーソル」と呼びます。ct_cursor コマンドで作成されたカーソルを「Client-Library カーソル」と呼びます。表 7-1 (102 ページ) では、この 2 つのタイプのカーソルを比較します。

言語カーソルと Client-Library カーソルの比較

表 7-1 では、Transact-SQL (言語) カーソル・コマンドと Client-Library カーソル・コマンドを比較します。

表 *7-1: Transact-SQL* カーソル・コマンドと *Client-Library* カーソル・コマンドの比較

オペレーション	言語コマンド	Client-Library カーソル・コマンド
宣言	declare cursor	ct_cursor(CS_CURSOR_DECLARE) または ct_dynamic(CS_CURSOR_DECLARE)
カーソル・ロー数の設定	set cursor rows	ct_cursor(CS_CURSOR_ROWS)
オープン	open	ct_cursor(CS_CURSOR_OPEN)
フェッチ	fetch	ct_fetch または ct_scroll_fetch (ct_results が *result_type* として CS_CURSOR_RESULT を返した後)
更新	update ... where current of *cursor_name*	ct_cursor(CS_CURSOR_UPDATE) (デフォルトでは、最後にフェッチされたローが処理対象になるが、前にフェッチされた任意のローにリダイレクトすることもできる)
削除	delete ... where current of *cursor_name*	ct_cursor(CS_CURSOR_DELETE) (デフォルトでは、最後にフェッチされたローが処理対象になるが、前にフェッチされた任意のローにリダイレクトすることもできる)
クローズ	close	ct_cursor(CS_CURSOR_CLOSE)
割り付け解除	deallocate cursor	ct_cursor(CS_CURSOR_DEALLOC) または ct_cursor(CS_CURSOR_CLOSE) (ct_cursor の *option* パラメータに CS_DEALLOC ビットが設定されていると、1 つのコマンドでカーソルのクローズと割り付け解除の両方を行う)

言語カーソル

Adaptive Server Enterprise では、言語カーソルは、declare cursor 文で宣言され、open 文でオープンされて、fetch 文を使用してフェッチされます。これらのコマンドについては、『ASE リファレンス・マニュアル』を参照してください。Client-Library プログラムは、これらの文のすべてを通常の言語コマンドとして送信できます。

言語カーソルが一度宣言され、オープンされていると、各 fetch 言語コマンドは、一連の通常ローを返し (ct_results *result_type* は CS_ROW_RESULT)、select コマンドの結果とまったく同じように処理することができます (「通常ロー結果の処理」(88 ページ) を参照してください)。他の言語コマンドと同様に、各コマンドの結果の処理を ct_results (および必要があれば、ct_fetch) によって終了しなければ、別のコマンドを同じ接続で送信することはできません。

クライアント接続で送信される言語コマンド内で宣言されている言語カーソルは、範囲をその接続内に制限されます。言い換えれば、同じ接続上を送信される言語コマンドだけが、カーソルを参照できます。

言語カーソルは、Client-Library カーソルに比べて、次のような利点があります。

- Adaptive Server Enterprise では、カーソルは、Transact-SQL ストアド・プロシージャ内で宣言し、オープンできます。そのようなカーソルを「サーバ・カーソル」といいます。ストアド・プロシージャとサーバ・カーソルを使用して複雑なタスクを実装すると、Client-Library カーソルを使用する同等の実装よりも高いパフォーマンスが得られます。このパフォーマンスの違いの要因は、主に、Client-Library カーソルはカーソル・ローをフェッチする (そしてネストされた更新コマンドを実行する) のにネットワーク上を何回も往復する必要があるのに対して、サーバ・カーソルにはその必要がないことです。
- 言語カーソルは、アドホック言語コマンドを処理する既存のクライアント・アプリケーションで使用できます。たとえば、Sybase isql クライアント・アプリケーションのユーザは、カーソルをサポートする特別なコードが isql に含まれていなくても、言語カーソルを使用できます。

言語カーソルの詳細については、『ASE リファレンス・マニュアル』を参照してください。

Client-Library カーソル

Client-Library カーソルの場合、アプリケーション・プログラマは、カーソルを宣言し、オープンする ct_cursor 呼び出しをコーディングしてください。Client-Library カーソル・オープン・コマンドは、CS_CURSOR_RESULT タイプのフェッチ可能な結果セットを 1 つ返します。

Client-Library カーソルの範囲は、1 つのコマンド構造体に制限されています。実際に、あるコマンド構造体でカーソルが宣言されると、コマンド構造体は、カーソルのその後のオペレーションの専用「ハンドル」になります。

Client-Library カーソルは、言語カーソルに比べて、次のような利点があります。

- Client-Library カーソルからのフェッチの方が簡単です。

  Client-Library カーソルからの各フェッチには、1 つの ct_fetch または ct_scroll_fetch 呼び出しが実行されます。ローを返す各 ct_fetch または ct_scroll_fetch 呼び出しをしてから、アプリケーションは同じ接続で新しいコマンドを送信できます。

  言語カーソルからの各フェッチは、ct_command、ct_send、ct_results、ct_fetch などの呼び出しに関係する別の Client-Library コマンドです。fetch 言語コマンドの結果の処理を完全に終了しなければ、アプリケーションは同じ接続で新しいコマンドを送信することはできません。

- Client-Library カーソルは、前にフェッチされたローを変更するのに使用できます。言語カーソルは、直前にフェッチされたローの削除または更新にしか使用できません。

- Client-Library カーソルは、ストアド・プロシージャを実行するように宣言できます (ただし、1 つの select 文だけを実行するストアド・プロシージャに限ります。詳細については、「手順 1：カーソルを宣言する」(108 ページ) を参照してください)。言語カーソルは select 文で宣言してください。

Client-Library カーソルの使用が適する場合

Client-Library カーソルはいくつかの利点を持つ反面、他のコマンド・タイプに比べて、パフォーマンス面では不利な点もあります。

Client-Library カーソルの利点

Client-Library カーソルには、次のような利点があります。

- アプリケーションは同じ接続で同時に複数のコマンド処理を行えます。
- アプリケーションは 1 つの接続だけを使用して、テーブルからフェッチしながら、テーブルを更新できます。

ct_cursor カーソル・オープン・コマンドは、1 つの接続で同時コマンド処理が可能な唯一のコマンド・タイプです。他のタイプのコマンドを送信したあとは、アプリケーションは、コマンドの結果を完全に処理しなければ、別のコマンドを送信できません。クライアント・アプリケーションは、カーソル・オープン・コマンドの結果を処理しながら、次の 2 つのカテゴリの新しいコマンドを実行します。

- 同じコマンド構造体のネストされたカーソル・コマンド
- 異なるコマンド構造体を使用して実行された関連性のないコマンド

ネストされたカーソル・コマンド

「ネストされたカーソル・コマンド」は、カーソル・オープン・コマンドによって返されたローをフェッチしている間に送信されるカーソル・クローズ・コマンド、カーソル削除コマンド、またはカーソル更新コマンドです。これらのコマンドの処理は、カーソル・ローを返したカーソル・オープン・コマンドの処理内に「ネスト」されています。ネストされたカーソル・コマンドを送信する前に、アプリケーションは、ct_fetch を呼び出して、少なくとも 1 つのカーソル・ローを取得する必要があります。

「ネストされたカーソル更新またはカーソル削除コマンド」(117 ページ) と「ネストされたカーソル・クローズ・コマンド」(119 ページ) を参照してください。

Client-Library のブラウズ・モード機能でも、アプリケーションはテーブルからフェッチしながらテーブルを更新できます。ただし、ブラウズ・モードの場合、サーバに対して 2 つの接続が必要です。この機能については、『Open Client Client-Library/C リファレンス・マニュアル』の「ブラウズ・モード」の項を参照してください。

異なるコマンド構造体を使用して実行されるコマンド

カーソル・オープン・コマンドによって返されるローをフェッチしている間に、別のコマンド構造体を使用して、任意のコマンドを実行できます。たとえば、アプリケーションがカーソルのデータに基づいて select または update コマンドを発行する場合があります。このような場合、アプリケーションは、別のコマンド構造体での結果の処理を完全に終了しなければ、次のカーソル・ローをフェッチしたり、ネストされたカーソル・コマンドを送信したりできません。また、アプリケーションが新しいカーソルをオープンすることがあります。この場合には、新しいカーソルをオープンして、そのコマンド・ハンドルがカーソル・ローを返せる状態にならなければ、アプリケーションは最初のカーソルで別のオペレーションを実行することはできません。

たとえば、次のデータが入っているサンプル・テーブル employee からローを選択するアプリケーションを考えてみます。

emp_fname	emp_lname	emp_id	mgr_id
Bob	Burnett	3349	4572
Alice	Williams	4572	5237
Thomas	Cooper	7028	3198
Samuel	Jones	6193	4572
Jennifer	Uribe	0969	4572
Joachin	Palmer	3198	4572
Jerry	Howe	5939	5237
George	Latimer	5237	NULL
...	...	...	...

ここでは、emp_id は従業員 ID 番号、mgr_id は各従業員のマネージャの従業員 ID 番号です。アプリケーションが行う仕事の 1 つは、フェッチされた各 employee ローに対して、別のクエリを発行して、最後にフェッチされた従業員の部下は誰であるかを検索することです。

アプリケーションは、Client-Library カーソルを使用して employee テーブルからローを選択する場合、2 番目のクエリを別の CS_COMMAND 構造体を使用して送信できます。アプリケーションが、カーソルを使用していない場合には、サーバに対する別の接続を使用して 2 番目のクエリを発行するか、最初のクエリからのすべての結果の処理が終了するまで待って、同じ接続で新しいコマンドを送信する必要があります。

Client-Library カーソルを使用する場合のパフォーマンス問題

一般的に、Client-Library カーソルは、言語コマンドまたは RPC コマンドを使用して実行される同等の select 文よりもパフォーマンスが悪くなります。前述のような特別な利点を必要としないアプリケーションは、言語コマンドまたは RPC コマンドを使用すると、より高いパフォーマンスを得ることができます。

ただし、同じタスクを実行するのに、ほかの方法では複数の接続やある種のロー・バッファリング・メカニズムが必要になるような場合には、カーソルの方がより良いパフォーマンスを得られることもあります。

Client-Library カーソルの使い方

標準的なアプリケーションは、次の手順に従って Client-Library カーソルを宣言し、オープンします。

1 カーソル宣言コマンドを送信します。

select 文で宣言するカーソルの場合：

- ct_cursor(CS_CURSOR_DECLARE)
- ct_param または ct_setparam (ホスト変数フォーマットを定義する)
- ct_send (コマンドをバッチ処理にしない場合)
- ct_results (コマンドをバッチ処理にしない場合、ループで)

execute 文で宣言するカーソルの場合：

- ct_cursor(CS_CURSOR_DECLARE)
- ct_send (コマンドをバッチ処理にしない場合)
- ct_results (コマンドをバッチ処理にしない場合、ループで)

準備された動的 SQL 文で宣言されるカーソルの場合：

- ct_dynamic(CS_CURSOR_DECLARE)
- (オプション) ct_cursor(CS_CURSOR_OPTION)
- ct_send
- ct_results (ループで)
- カーソルを ct_cursor コマンドで宣言する場合、手順 1、2、3 のコマンドは、バッチで処理できます。バッチ処理にすると、1 回の ct_send 呼び出しで一括送信できます。

2　(オプション) カーソル・ロー・コマンドを送信します。

- ct_cursor(CS_CURSOR_ROWS)
- ct_send (コマンドをバッチ処理にしない場合)
- ct_results (コマンドをバッチ処理にしない場合、ループで)

3　カーソル・オープン・コマンドを送信します。

- ct_cursor(CS_CURSOR_OPEN)
- ct_param または ct_setparam (パラメータ値を渡す)
- ct_send
- ct_results (標準結果ループで呼び出す)

正常に実行されたオープン・コマンドは、CS_CURSOR_RESULT 結果セットを返します。コマンドをバッチ処理にしている場合、ct_results 呼び出しを数回行わなければ (バッチ処理されたコマンドからのステータス結果を取得するため)、カーソル・ローは使用可能になりません。

4　カーソル・ローを処理します。

- ct_bind (カーソル・ローをバインドする)
- ct_fetch または ct_scroll_fetch (各ローを取得するのにループで呼び出す)
- 少なくとも 1 つのローがフェッチされてから、ct_fetch または ct_scroll_fetch のループ内で新しいコマンドを送信できます。「手順 4：カーソル・ローを処理する」(116 ページ) を参照してください。

5　カーソルをクローズします。

- ct_cursor(CS_CURSOR_CLOSE)
- ct_send
- ct_results

アプリケーションは、カーソル・クローズ・コマンドを定義するときに ct_cursor の *option* パラメータの CS_DEALLOC ビットをオンに設定すると、1 つのコマンドでカーソルのクローズとその割り付け解除の両方を実行できます。その場合、次の手順 6 は必要ありません。

6 カーソルの割り付けを解除します。

- ct_cursor(CS_CURSOR_DEALLOC)
- ct_send
- ct_results

上記のプロセスの各手順は、サーバに 1 つの Client-Library カーソル・コマンドを送信します。各コマンドを送信してから、アプリケーションは ct_results で結果を処理する必要があります。「基本ループの構造」(86 ページ) の説明に従って、標準結果ループ内でカーソル・コマンドの結果を処理するようにアプリケーションをコーディングしてください。

手順 1：カーソルを宣言する

カーソル宣言コマンドには、3 つのタイプがあります。各タイプによって、カーソルの select 文の実行方法が異なります。

- 第 1 のタイプのカーソルは select 文を直接実行します。

  アプリケーションは、ct_cursor を呼び出して、ct_cursor の *text* 引数として select 文を渡します。

- 第 2 のタイプのカーソルは、ストアド・プロシージャを実行します。

  アプリケーションそれ自体またはアプリケーション管理者が、前もって作成されているストアド・プロシージャによって select 文を実行します。カーソルを宣言するには、ct_cursor を呼び出し、*text* 引数として、プロシージャを呼び出す execute 文を渡します。カーソルは、1 つの select 文を含むストアド・プロシージャにしか宣言できません。

- 第 3 のタイプのカーソルは、準備された動的 SQL 文を実行します。

  アプリケーションは、ct_dynamic(CS_PREPARE) を呼び出して、select 文を実行する準備文を作成します。次に、ct_dynamic(CS_CURSOR_DECLARE) を呼び出して、ct_dynamic の *id* 引数として文識別子を渡します。

select 文を直接実行するためのカーソルの宣言

select 文を直接実行するカーソルまたはスクロール可能カーソルを作成するには、*type* を CS_CURSOR_DECLARE に、*text* を select 文にそれぞれ設定して、ct_cursor を呼び出します。

簡単なカーソル宣言

次のコード例は、Client-Library カーソルを宣言しています。簡単にするために、この例では、リターン・コードのチェックは省略しています。

```
CS_CHAR body[1024];
strcpy(body, "select * from titles for read only");
 ret = ct_cursor(cmd, CS_CURSOR_DECLARE,
          "a cursor", CS_NULLTERM,
          body, CS_NULLTERM, CS_UNUSED);
```

次のコード例は、Client-Library スクロール可能カーソルを宣言しています。簡単にするために、この例では、リターン・コードのチェックは省略しています。

```
CS_CHAR body[1024];
strcpy(body, "select * from titles");
 ret = ct_cursor(cmd, CS_CURSOR_DECLARE,
          "s cursor", CS_NULLTERM,
          body, CS_NULLTERM, CS_SCROLL_CURSOR);
```

パラメータを持つカーソルの宣言

select 文には、カーソルをオープンしたときに、文のどこにパラメータを代入するかを指示するために、*@variable_name* 形式のホスト言語変数を含めることもできます。Adaptive Server Enterprise では、カーソルの where 句の値の代わりに変数を使用できます。たとえば、変数 int 値を持つカーソルを宣言するのに、次のような文を使用できます。

```
SELECT title_id, title, price FROM titles
 WHERE total_sales > @sales_val
```

この場合、カーソルを宣言してから、*data* ポインタを NULL にして ct_param または ct_setparam を呼び出して、パラメータ・フォーマットを指定してください。カーソルをオープンするたびに、アプリケーションは ct_param または ct_setparam を再度呼び出して、パラメータ値を与えます。この例を次に示します。

```
CS_CHAR     body[1024];
 CS_DATAFMT  intfmt;
 CS_INT      sales_val;
  strcpy(body, "select title_id, title, price from
       titles where total_sales > @sales_val
       for read only");
 ret = ct_cursor(cmd, CS_CURSOR_DECLARE,
          "a cursor", CS_NULLTERM,
          body, CS_NULLTERM, CS_UNUSED);
 ... error checking deleted ...

(CS_VOID)memset(&intfmt, 0,  sizeof(intfmt));
/*
 ** Define the format of @sales_val.
```

```
 */
 intfmt.datatype = CS_INT_TYPE;
 intfmt.maxlength = CS_SIZEOF(CS_INT);
 intfmt.status = CS_INPUTVALUE;
ret = ct_param(cmd, &intfmt, (CS_VOID *)NULL,
              CS_UNUSED, 0);
 ... error checking deleted ...
ret = ct_cursor(cmd, CS_CURSOR_OPEN, NULL,
               CS_UNUSED, NULL, CS_UNUSED,
               CS_UNUSED);
 ... error checking deleted ...
/*
 ** Supply a value for @sales_val. intfmt fields
 ** were set above.
 */
sales_val = 1;
 ret = ct_param(cmd, &intfmt,
              (CS_VOID *)&sales_val, CS_UNUSED, 0);
 ... error checking deleted ...
/*
 ** Send the batched cursor declare and open
 ** commands.
 */
 ret = ct_send(cmd);
 ... error checking deleted ...
```

更新可能なカラムの指定

Adaptive Server Enterprise に接続するアプリケーションの場合、select 文の for read only 句または for update of 句を使用して、更新するカラムを指定します。ct_cursor(CS_CURSOR_DECLARE) 呼び出しでは、ct_cursor の *option* パラメータに CS_UNUSED を指定して、サーバが更新するカラムを決定します。たとえば、次のような文で宣言されたカーソルは、price カラムの更新が可能です。

```
SELECT title_id, title, price FROM titles
 FOR UPDATE OF price
```

カスタム Open Server など、そのほかのサーバは、select 文の for read only 句または for update of 句を認識したり使用したりできません。これらのサーバの場合、どのカラムを更新するかは、クライアント・アプリケーションが ct_param または ct_setparam を別に呼び出して指示する必要があります。詳細については、『Open Client Client-Library/C リファレンス・マニュアル』の「ct_cursor」を参照してください。

ストアド・プロシージャを実行するカーソルの宣言

カーソルは、1 つの select 文を実行するストアド・プロシージャを実行するように宣言することができます。この形式のカーソルを作成するには、*type* を CS_CURSOR_DECLARE に、*text* をプロシージャを実行する execute 文に設定して、ct_cursor を呼び出します。

たとえば、前述の例の select 文は、次のようにストアド・プロシージャから呼び出すことができます。

```
CREATE PROCEDURE titlecursorproc
  @sales_val INT
 AS
  SELECT title_id, price, title FROM titles
  WHERE ( total_sales > @sales_val )
  FOR READ ONLY
```

Adaptive Server Enterprise のストアド・プロシージャを実行する Client-Library カーソルの場合、ホスト言語変数も使用せず、ct_param で変数フォーマットを定義することもしません。サーバは、ストアド・プロシージャの宣言からパラメータ・フォーマットを決定します。それ以外の場合については、カーソルを宣言してオープンするのに必要な手順は、「パラメータを持つカーソルの宣言」(109 ページ) で説明した手順と同じです。次に、*titlecursorproc* ストアド・プロシージャで Client-Library カーソルを宣言してオープンする方法の例を示します。

```
CS_CHAR     body[1024];
 CS_DATAFMT  intfmt;
 CS_INT      sales_val;
strcpy(body, "EXECUTE titlecursorproc");
 ret = ct_cursor(cmd, CS_CURSOR_DECLARE,
          "a cursor", CS_NULLTERM,
          body, CS_NULLTERM, CS_UNUSED);
 ... error checking deleted ...
ret = ct_cursor(cmd, CS_CURSOR_OPEN, NULL,
               CS_UNUSED, NULL, CS_UNUSED,
               CS_UNUSED);
 ... error checking deleted ...
/*
 ** Supply a value for the @sales_val parameter for
 ** titlecursorproc.
 */
 (CS_VOID)memset(&intfmt, 0,  sizeof(intfmt));
 intfmt.datatype = CS_INT_TYPE;
 intfmt.maxlength = CS_SIZEOF(CS_INT);
 intfmt.status = CS_INPUTVALUE;
sales_val = 1;
 ret = ct_param(cmd, &intfmt,
              (CS_VOID *)&sales_val, CS_UNUSED, 0);
 ... error checking deleted ...
/*
 ** Send the batched cursor declare and open
 ** commands.
 */
 ret = ct_send(cmd);
 ... error checking deleted ...
... results processing deleted ...
```

準備された動的 SQL 文を実行するカーソルの宣言

1 つの select 文を実行する準備された動的 SQL 文で、カーソルを宣言することができます。たとえば、次のような select 文を実行する文を準備できます。

```
SELECT title_id, title, price FROM titles
 WHERE total_sales > ? FOR READ ONLY
```

“?” 文字 (動的パラメータ・マーカ) は、カーソルをオープンするときに用意されるパラメータ値のプレースホルダです。動的 SQL 文は、ct_dynamic(CS_PREPARE) コマンドをサーバに送信し、その結果を処理することによって、作成されます。詳細については、「手順 1：文を準備する」(127 ページ) を参照してください。

アプリケーションは、文を準備してから、*type* を CS_CURSOR_DECLARE に、*id* を文識別子に設定して ct_dynamic を呼び出すことができます。

更新するカラムを指定するには、select 文で for read only 句または for update of 句を使用します。文にこれらの句がなければ、アプリケーションは、カーソル宣言コマンドを起動する ct_dynamic を呼び出した直後に、ct_cursor(CS_CURSOR_OPTION) を呼び出すことができます。

ct_dynamic カーソル宣言コマンドは、ct_cursor カーソル・ロー・コマンドまたは ct_cursor カーソル・オープン・コマンドと一緒にバッチ処理することはできません。

次のコード例は、準備文を使用してカーソルを宣言しオープンする方法を示しています。

```
  /*
  ** Prepare the statement.
  */
  strcpy(body, "SELECT title_id, title, price FROM titles
                WHERE price > ? FOR READ ONLY");
  strcpy(stmt_id, "dyn_a");
  retcode = ct_dynamic(cmd, CS_PREPARE, stmt_id, CS_NULLTERM,
             body, CS_NULLTERM);
  if (retcode != CS_SUCCEED)
  {
    ex_error("DoCursor:ct_dynamic(prepare) failed");
    return retcode;
  }

if ((retcode = ct_send(cmd)) != CS_SUCCEED)
  {
    ex_error("DoCursor:ct_send() failed");
    return retcode;
  }

  ... ct_results() loop goes here.No fetchable results are
      returned ...

 /*
  ** Declare the cursor
  */
```

```
  retcode = ct_dynamic(cmd, CS_CURSOR_DECLARE,
                       stmt_id, CS_NULLTERM,
                       "cursor_a", CS_NULLTERM);
  if (retcode != CS_SUCCEED)
  {
    ex_error("DoCursor:ct_dynamic(cursor declare) failed");
    return retcode;
  }

if ((retcode = ct_send(cmd)) != CS_SUCCEED)
  {
    ex_error("DoCursor:ct_send() failed");
    return retcode;
  }

  ... ct_results() loop goes here.No fetchable results are
      returned by the cursor-declare command ...
```

手順 2：カーソル・ロー数を設定する

Client-Library カーソルを宣言したあと、アプリケーションは ct_cursor を呼び出して、カーソルのカーソル・ロー設定値を指定できます。カーソル・ロー設定値には、ct_fetch 呼び出しごとにアプリケーションに返されるローの数ではなく、内部フェッチ要求ごとにサーバが Client-Library に返すローの数が定義されます。内部フェッチ要求は、ct_fetch 要求を満たす以上のローがサーバから要求されたときに、発行されます。

デフォルトでは、カーソル・ロー設定値は 1 です。アプリケーションがカーソル・オープン・コマンドより前にカーソル・ロー・コマンドを送信しない場合、カーソル・ロー設定値は 1 になります。ct_cursor コマンドでカーソルを宣言する場合、カーソル・ロー・コマンドをカーソル・オープン・コマンドとバッチ処理できます。

カーソル・ロー設定値によって、Client-Library が各内部 Client-Library フェッチ要求に対してサーバからいくつのローを受信するかが決まります。たとえば、カーソル・ロー数が 5 に設定されていて、アプリケーションが配列バインドを使用しない場合、Client-Library は、アプリケーションが最初に ct_fetch を呼び出したとき、次は 6 回目に呼び出したときというように、5 回ごとの呼び出しに内部フェッチ要求を行います。

ct_scroll_fetch の実行結果として複数のローを得るには、カーソル・ロー設定値を 1 より大きい値にする必要があります。効率を最大限に高めるため、配列バインドも使用する必要があります。配列バインド数は、CS_CURSOR_ROWS と同じ値にしてください。

注意 CS_CURSOR_ROWS 設定値が 1 より大きい場合、ct_scroll_fetch で配列バインドを行う必要があります。配列バインドは、ct_fetch と ct_scroll_fetch のどちらでも使用できます。CS_CURSOR_ROWS にデフォルト値 1 が設定されている場合は、どちらの API コールでも標準プログラム変数を使用できます。

---

1 より大きいカーソル・ロー設定値を指定すると、Client-Library バッファは、追加内部ロー・フェッチを透過的に処理します。アプリケーションが ct_fetch を呼び出して、カーソル・ローをフェッチすると、Client-Library はネットワークから直接そのローを読み込んで、サーバに内部フェッチ要求を送信してさらにローを得るか、または内部ロー・バッファからローを取得します。次のような場合には、Client-Library はカーソル・ローを内部でバッファする必要があります。

- アプリケーションがネストされたカーソル更新またはカーソル削除コマンドを送信した場合
- アプリケーションがカーソルとは異なるコマンド構造体でコマンドを送信した場合

このような状態の場合には、Client-Library は、書き込み用の接続をクリアするために、未読のローを読み込んでバッファに入れる必要があります。

一般に、読み込み専用カーソルを処理する場合には、カーソル・ロー数を高く設定すると、アプリケーションのパフォーマンスが高くなります。カーソル・ロー数の設定値を高くすると、ローをフェッチするのに必要なネットワーク往復回数は少なくなります。ただし、カーソル・ロー数が高すぎて、Client-Library がローをバッファしなければならなくなると、バッファのためのオーバヘッドが発生し、往復回数の減少による利点は相殺されてしまいます。

Client-Library がローをバッファする必要が生じる可能性を最小化するために、カーソル・ロー数設定値に合った配列サイズで配列バインドを使用してください。『Open Client Client-Library/C リファレンス・マニュアル』の「ct_bind」のリファレンス・ページを参照してください。

手順 3：カーソルをオープンする

ct_cursor(CS_CURSOR_OPEN) を呼び出して、カーソル・オープン・コマンドを起動します。カーソルに入力パラメータが必要な場合には、ct_param または ct_setparam を呼び出して、パラメータを定義します。各パラメータ値に、1 回の呼び出しが必要です。次の条件が 1 つでも当てはまる場合には、パラメータ値が必要になります。

- カーソルの本体が、ホスト変数が含まれている SQL テキスト文字列である。
- カーソルの本体が、入力パラメータ値を必要とするストアド・プロシージャである。
- カーソルの本体が、動的パラメータ・マーカを含む動的 SQL 文である。

カーソル・オープン・コマンドをリストアするアプリケーションの場合、ct_param ではなく、ct_setparam を呼び出して、カーソル・オープン・コマンドのパラメータ値を指定してください。ct_setparam を使用した場合、アプリケーションはリストアされたカーソル・オープン・コマンドのパラメータ値を変更できます。「カーソルの再オープン」(115 ページ) を参照してください。

カーソル・コマンドのバッチ

ct_cursor で宣言したカーソルは、バッチ処理することができます。カーソルを最初にオープンするときに、アプリケーションは、カーソル宣言コマンド、カーソル・ロー・コマンド、カーソル・オープン・コマンドを 1 回の ct_send 呼び出しで送信し、その結果を 1 つの結果ループで処理できます。

カーソルを再オープンする場合、アプリケーションは、カーソル・ロー・コマンドをカーソル・オープン・コマンドとバッチ処理できます。

コマンドをバッチにすると、カーソルをオープンするのに必要なネットワーク往復回数は少なくなります。

カーソルの再オープン

カーソル・オープン・コマンドの結果を処理し終えてから、前のカーソル・オープン・コマンドを 1 回の ct_cursor 呼び出しでリストアできます (構文は下記参照)。リストア・オペレーションによって、コマンド構造体は前のカーソル・オープン・コマンドをいつでも送信できる状態になります。次のようなコマンド情報がリストアされます。

- カーソル・オープン・コマンドとバッチにしたカーソル・ロー・コマンド
- ct_param で渡したカーソル・オープン・コマンドのパラメータ値
- ct_setparam で設定したパラメータ送信元変数のバインド情報 (ct_send は、リストアされたコマンドが送信されると、現在の値を読み込みます)。

カーソル・オープン・コマンドとバッチ処理されたカーソル宣言コマンドは、リストアされません。

カーソル・オープン・コマンドをリストアするには、*type* を CS_CURSOR_OPEN に、*option* を CS_RESTORE_OPEN に設定して、ct_cursor を呼び出します。ほとんどのアプリケーションは、次に示すようなプログラム構造を使用して、カーソル・オープン・コマンドをリストアし、送信します。

```
/*
** Assign new variables in the program variables
** bound with ct_setparam.
*/
... assignment statement for each parameter
     source variable ...
ct_cursor(CS_CURSOR_OPEN, ..., CS_RESTORE_OPEN)
ct_send
... handle cursor results ...
```

カーソルは、新しいカーソル・オープン・コマンドを (必要であれば、カーソル・ロー・コマンドを前に付けて) 起動することによって、再オープンすることもできます。ただし、前のコマンドをリストアする方式の方が、Client-Library 呼び出し回数が少なくなります。

手順 4：カーソル・ローを処理する

カーソル結果は、標準ループ構造で ct_results を呼び出して処理する方法をおすすめします (「基本ループの構造」(86 ページ) を参照してください)。カーソル・ローは、CS_CURSOR_RESULT に等しい *result_type* が ct_results から返され、使用可能になります。カーソル・ローは、他のフェッチ可能な結果セットと同じように処理されます (「カーソル結果の処理」(89 ページ) を参照してください)。

他の結果タイプとの違いは、アプリケーションがカーソル・ローをフェッチしながら新しいコマンドを発行できる点です。これらのコマンドは、次のどちらかのタイプです。

- ネストされたカーソル・コマンド ― カーソルを制御するコマンド構造体を使用して実行されるカーソル・クローズ・コマンド、カーソル削除コマンド、カーソル更新コマンド
- その他のすべてのコマンド ― 別のコマンド構造体を使用して実行されるコマンド

ネストされたカーソル更新またはカーソル削除コマンド

アプリケーションは、カーソル結果セットを処理しながら、カーソル結果セット内の、前にフェッチされたローを更新または削除できます。この変更は、カーソル結果セットを抽出したベース・テーブルまで戻って適用されます。

カーソル更新コマンドは、*type* を CS_CURSOR_UPDATE、*name* をベース・テーブルの名前、*text* を SQL 更新句に設定した ct_cursor を呼び出して起動します。たとえば、次の呼び出しでは、pubs2 データベースの *authors* テーブル内のローを更新するコマンドが構築されます。

```
ret_code = ct_cursor(cmd, CS_CURSOR_UPDATE,
     "authors", CS_NULLTERM, "update authors ¥
     set au_lname = 'Barr'", CS_NULLTERM,
     CS_UNUSED);
ct_send(cmd);
ct_results(cmd, &res_type);
```

カーソル更新では、1 つのテーブルからだけカラムを更新できます。複数のテーブルのカラムを更新する場合には、別々にコマンドを送信できます。

カーソル削除コマンドは、*type* は CS_CURSOR_DELETE に、*name* はローを削除するベース・テーブルの名前に、*text* は NULL に設定した ct_cursor を呼び出して起動します。

カーソル更新またはカーソル削除コマンドを送信してから、アプリケーションが更新または削除オペレーションの処理を完全に終了しなければ、再度 ct_fetch を呼び出すことはできません。

キー・カラム

カーソル結果セットのプライマリ・キーの一部になっているカラムはアプリケーションで更新しないようにしてください。ct_describe は、*datafmt.status* フィールドの CS_KEY ビットをオンに設定して、カラムが結果セットのプライマリ・キーであることを示します。

更新または削除のリダイレクト

デフォルトでは、カーソル更新またはカーソル削除は、最後にフェッチされたローが処理対象になります。ただし、更新または削除をリダイレクトして、前にフェッチされたローを処理することもできます。リダイレクトされた更新または削除は、配列バインドを行ってカーソル・ローを処理するアプリケーションでよく使用されます。

カーソル更新またはカーソル削除は、コマンドを送信する前に ct_keydata を呼び出して、リダイレクトします。

更新をリダイレクトするアプリケーションの場合、コマンド構造体の CS_HIDDEN_KEYS プロパティが CS_TRUE であることを確認してから、カーソルをオープンしてください (ct_cmd_props を使用してコマンド構造体のプロパティを設定してからカーソルをオープンするか、または ct_con_props を使用して接続レベルでコマンド構造体のプロパティを設定してからコマンド構造体を割り付けてください)。カーソルの隠しキー・カラムをアプリケーションに公開するかどうかは CS_HIDDEN_KEYS で決めます。

「隠しキー・カラム」は、カーソルの結果セットと一緒に返されますが、カーソルの select リストに指定されていないカラムです。ct_describe は、*datafmt.status* フィールド内の CS_HIDDEN ビットを設定して、カラムがカーソルの select リストにはないことを示します。

隠しキー・カラムには、サーバがカーソル更新およびカーソル削除の送信先ローを見つけるのに必要な追加情報が保管されています。通常、Client-Library はこれらの追加カラムを内部で処理し、アプリケーションには公開しません。ただし、リダイレクトされた更新または削除を行うアプリケーションは、隠しキー・カラムを明示的に処理する必要があります。

カーソル更新またはカーソル削除をリダイレクトするには、アプリケーションは ct_keydata を呼び出して、バージョン・キーまたはプライマリ・キーであるローの各カラムについて (隠しカラムも含めて) 値を指定してください。バージョン・キーとプライマリ・キーについて、以下に説明します。

- 「プライマリ・キー・カラム」は、カーソル結果セットのプライマリ・キーの一部です。ct_describe は、*datafmt.status* フィールドの CS_KEY ビットをオンに設定して、カラムが結果セットのプライマリ・キーであることを示します。
- 「バージョン・キー・カラム」は、カーソル結果セットのプライマリ・キーには含まれない実テーブルのカラム (select リスト内の式ではない) です。ct_describe は、*datafmt.status* フィールド内の CS_VERSION_KEY ビットを設定して、カラムが結果セットのバージョン・キーであることを示します。

隠しキー・カラムは、プライマリ・キー・カラム、バージョン・キー・カラムのどちらでもかまいません。

カーソル更新をリダイレクトするアプリケーションは、次の規則に従ってコーディングしてください。

- CS_HIDDEN_KEYS プロパティが CS_TRUE であることを確認してから、カーソルをオープンする。
- カーソル・ローを処理する場合、ct_describe を呼び出して、隠しカラムを含めて、すべてのカーソル・カラムの CS_DATAFMT 情報を得る。あとの更新で使用できるように、この情報を保存する。
- 対話型アプリケーションの場合、CS_DATAFMT *status* フィールド内の CS_HIDDEN ビットを使用して、カラムを表示するかどうかを決定する。
- ローを取得する場合、更新できるすべてのローのカラム値を保存する。これらの値は、ct_keydata の入力値として要求される。
- 前にフェッチされたローを更新するには、対応する CS_DATAFMT の *status* フィールドの CS_KEY または CS_VERSION_KEY ビットがオンに設定されているローのすべてのカラムに対して ct_keydata を呼び出す。
- キー・カラムを更新しない。CS_DATAFMT の status フィールドの CS_KEY ビットをチェックして、カラムがキー・カラムであるかどうかを判別する。

ネストされたカーソル・クローズ・コマンド

アプリケーションは、カーソル・クローズ・コマンドを送信し、その結果を処理することによって、すべてのローをフェッチする前に、カーソルをクローズできます。詳細については、「手順 5：カーソルをクローズする」(119 ページ)を参照してください。

次のような理由から、ct_cancel を呼び出して不要なカーソル・ローを廃棄するよりも、カーソルをクローズすることの方をおすすめします。

- ct_cancel(CS_CANCEL_ALL) または ct_cancel(CS_CANCEL_ATTN) を呼び出すと、接続のカーソルが未定義のステータスになることがあります。
- ct_cancel(CS_CANCEL_CURRENT) を呼び出すと、ネットワーク帯域が無駄に使用される場合があります。この呼び出しでは、Client-Library はネットワークを介して残りのローをフェッチして廃棄します。

異なるコマンド構造体でのコマンド送信

アプリケーションは、オリジナル・カーソルからローをフェッチしながら、オリジナル・カーソルに関係ないコマンドを別のコマンド構造体で送信できます。

たとえば、アプリケーションがカーソルのデータに基づいて select または update コマンドを発行する場合があります。この場合には、アプリケーションは、次のカーソル・ローをフェッチする前に、新しいコマンドの結果の処理を完全に終了する必要があります。また、アプリケーションが新しいカーソルをオープンすることがあります。この場合には、新しいカーソルをオープンして、そのコマンド・ハンドルがカーソル・ローを返せる状態にならなければ、アプリケーションは最初のカーソルで別のオペレーションを実行することはできません。

手順 5：カーソルをクローズする

アプリケーションは、*type* を CS_CURSOR_CLOSE に設定した ct_cursor を呼び出して、カーソル・クローズ・コマンドを起動します。アプリケーションは、再度カーソルを使用しない場合、ct_cursor の *option* パラメータを CS_DEALLOC で渡して、1 つのコマンドでカーソルのクローズとその割り付け解除を行うことができます。それ以外の場合は、*option* は CS_UNUSED にします。

手順 6：カーソルの割り付けを解除する

アプリケーションは、type を CS_CURSOR_DEALLOC に設定して ct_cursor を呼び出して、カーソル割り付け解除コマンドを起動します。アプリケーションが明示的にカーソル割り付けを解除しない場合には、アプリケーションが切り離されたときに、割り付けも解除されます。

Client-Library のカーソル・プロパティ

Client-Library のカーソルは、一度宣言されると、1 つのコマンド構造体と対応付けされます。アプリケーションは、ct_cmd_props を呼び出して次のプロパティを取得することによって、コマンド構造体に対応するカーソルに関する情報を取得できます。

- CS_CUR_ID － カーソルのサーバ識別番号が含まれます。カーソルの識別番号は、ct_cmd_props(CS_CUR_STATUS) を呼び出してから、カーソルが特定のコマンド領域に存在するかどうかを確認するために取得できます。
- CS_CUR_NAME － カーソルの名前が含まれます。アプリケーションは、ct_cursor(CS_CURSOR_DECLARE) 呼び出しで CS_SUCCEED が返されてからいつでも、CS_CUR_NAME プロパティを使用して、カーソルの名前を取得できます。
- CS_CUR_ROWCOUNT － カーソル・ロー数の設定値が含まれます。この値は、内部フェッチ要求ごとに Client-Library に返されるローの数です。カーソルのロー数は、ct_cmd_props(CS_CUR_STATUS) 呼び出しをしてから、カーソルが特定のコマンド領域に存在するかどうかを確認するために取得できます。
- CS_CUR_STATUS － カーソル・ステータスを表示します。アプリケーションは、CS_CUR_STATUS プロパティを使用して、次のことを判断します。
  - カーソルが特定のコマンド領域内に存在するか
  - カーソルがオープンしているか
  - カーソルを更新に使用できるか
  - カーソルが読み込み専用か、sensitivity の値があるか、スクロール可能であるか

  ct_cancel を呼び出すと、接続のカーソルが未定義のステータスになることがあります。アプリケーションは、カーソル・ステータス・プロパティを使用して、キャンセル・オペレーションがカーソルにどのように影響を与えたかを知ることができます。
- CS_HAVE_CUROPEN － リストアできるカーソル・オープン・コマンドがコマンド構造体にあるかどうかを表示します。「カーソルの再オープン」(115 ページ) を参照してください。

これらのプロパティはすべて、ct_cmd_props を呼び出して値を取得できる取得専用のコマンド構造体のプロパティです。詳細については、『Open Client Client-Library/C リファレンス・マニュアル』の「ct_cmd_props」を参照してください。

第 8 章

動的 SQL コマンドの使い方

この章では、動的 SQL について説明します。次の項目が含まれます。

トピック名	ページ
動的 SQL の概要	121
動的 SQL の利点	122
動的 SQL の制限事項	122
動的 SQL に代わる方法	124
即時実行方式の使い方	124
準備実行方式の使い方	125
動的 SQL とストアド・プロシージャの比較	131

動的 SQL の概要

動的 SQL は、Client-Library の ct_dynamic ルーチンによって起動されるコマンドを使用して実行時に SQL 文を生成し、準備し、実行するプロセスです。

動的 SQL は、主として、プリコンパイラ・サポートに役立ちますが、対話型アプリケーションでも使用できます。

Client-Library と Adaptive Server Enterprise は、次の 2 つの方式で動的 SQL コマンドを実行できます。

- 即時実行 – クライアント・アプリケーションはサーバに、リテラル文を実行する 1 つの ct_dynamic コマンドを送信します。これは、言語コマンドを送信するプロセスと基本的には同じですが、より多くの制限があります (「言語コマンド」(72 ページ) を参照してください)。
- 準備実行 – クライアント・アプリケーションは、文を作成して、それを 1 回または複数回実行する一連のサーバ・コマンドをサーバに送信します。アプリケーションは、文の入力パラメータおよび文から返される結果セットのフォーマットについてサーバに問い合わせを行う追加のコマンドも送信できます。

準備実行方式の場合、クライアント・アプリケーションは、ct_dynamic (CS_PREPARE) コマンドをサーバに送信して、「準備文」を作成します。準備文は Adaptive Server Enterprise ストアド・プロシージャに似ています。どちらの場合も、サーバは、作成された SQL 文の構文をチェックし、最適クエリ・プランを構築し、実行できる形式でそのクエリ・プランを保管します。準備文とストアド・プロシージャの重要な違いは、次のとおりです。

- 準備文は、クライアント・プログラムが切り離されると、自動的に削除されますが、ストアド・プロシージャは削除されません。
- 準備文は、文を作成した接続だけから見える識別子によって参照されますが、ストアド・プロシージャの名前はどのクライアント接続からも見えます (ただし、プロシージャのパーミッションによって、実行できるユーザは制限されます)。
- 準備文の場合、クライアント・プログラムは、実行しないで入力 (パラメータ) および出力 (結果) カラムのフォーマットを簡単に決定できます。

動的 SQL の利点

アプリケーションは、動的 SQL コマンドを使用して、「汎用」SQL 文を 1 回準備するだけで、何回でも実行できます。また、文の中に、実行時に入力するパラメータ値のマーカを含むことができるので、さまざまな入力値で文を実行できます。

動的 SQL の制限事項

動的 SQL には、重要な制限事項がいくつかあります。

動的 SQL コマンドのパフォーマンス

永久 Adaptive Server Enterprise ストアド・プロシージャを作成しておいてクライアント・プログラムから RPC コマンドで呼び出す方式と比べると、動的 SQL によるアプリケーションの実装は、パフォーマンスが劣ります。

アプリケーションのために Adaptive Server Enterprise ストアド・プロシージャを作成する場合、プロシージャが作成された時点に 1 回だけ、SQL 文のコンパイルと最適化が行われます。一方、動的 SQL アプリケーションの場合、クライアント・プログラムが実行されるたびに、コンパイルと最適化のオーバヘッドがかかります。動的 SQL の実装の場合、クライアント・プログラムを実行するたびにアプリケーションの準備文のコンパイル済みバージョンを別に作成しなければならないので、データベース領域のオーバヘッドもかかってきます。逆に、ストアド・プロシージャと RPC コマンドを使用するようにアプリケーションを設計した場合、クライアント・プログラムを実行するたびに常に同じストアド・プロシージャを共有できます。

Adaptive Server Enterprise の制限事項とデータベースの稼働条件

Adaptive Server Enterprise は、テンポラリ・ストアド・プロシージャを使用して、動的 SQL を実装します。テンポラリ・ストアド・プロシージャは、SQL 文が準備されるときに作成され、準備文の割り付けが解除されるときに破棄されます。準備文の割り付けは、ct_dynamic(CS_DEALLOC) 呼び出しで明示的に解除できますが、接続がクローズされると暗黙的に解除されます。

この実装の結果として、Adaptive Server Enterprise にアクセスして動的 SQL を使用するアプリケーションには、Adaptive Server Enterprise ストアド・プロシージャの制限事項が適用されます。このため、次のような影響を受けます。

- 準備文の割り付けが解除されると、テンポラリ・テーブルは破棄されます。
- text および image データ型のパラメータはサポートされません。
- サポートされる最大パラメータ数は、255 です。
- 動的 SQL 文自体が (Transact-SQL execute 文で) ストアド・プロシージャを実行する場合、出力パラメータ値とリターン・ステータスはクライアント・アプリケーションには使用できません。
- プレースホルダで示されるパラメータのデータ型は解析時には明らかになっている必要があります。次の文は有効ではありません。

```
? <op> ?, (? is null)

CONVERT(<type>, ?)
```

ストアド・プロシージャの詳細については、『Transact-SQL ユーザーズ・ガイド』を参照してください。

動的 SQL に代わる方法

別の DBMS システムを理解してから Sybase を使用するシステム開発者は、Sybase の動的 SQL の実装と他のベンダの方式との違いを十分に認識してください。Adaptive Server Enterprise では、ほとんどのコマンド・タイプが「動的」です。Adaptive Server Enterprise が提供する「静的 SQL コマンド」に最も近い手法がストアド・プロシージャです。ただし、ストアド・プロシージャのパーミッションがクライアント・プログラムのユーザに実行を許可しているかぎり、どのクライアント・アプリケーションもストアド・プロシージャを呼び出すことができます。他の DBMS システムでは、一般に、プリコンパイルされた静的 SQL コマンドのスコープはプリコンパイルされたアプリケーションに限定されます。

Adaptive Server Enterprise アプリケーションの場合、他の DBMS システムでは動的 SQL を使用する必要がある多くのタスクを、動的 SQL ではない Client-Library コマンド・タイプで実装できます。次に例を示します。

- 実行以前にはテキストがわからない SQL 文を実行しなければならないアプリケーションの場合、ct_command を呼び出して言語コマンドを定義するようにクライアント・プログラムをコーディングできます。この方式は、1 回または数回だけ実行するコマンドに適しています。
- 実行以前にテキストがわかっていてパフォーマンスが重要であるコマンドを実行しなければならないアプリケーションの場合、Adaptive Server Enterprise ストアド・プロシージャを作成して、RPC コマンド (ct_command で定義される) でプロシージャを呼び出すようにクライアント・プログラムをコーディングできます。
- 対話形式でカーソルを定義しオープンしなければならないアプリケーションの場合、ct_cursor でカーソル宣言コマンドを定義するようにクライアント・プログラムをコーディングできます。

即時実行方式の使い方

即時実行方式では、サーバに 1 つのコマンドを送信して 1 つの SQL 文を実行します。

即時実行方式の使用が適する場合

動的 SQL 文は、次の基準を満たす場合にかぎって、即時実行できます。

- フェッチ可能なデータを返さない (select 文ではない)。
- パラメータのプレースホルダ (文のテキストでは疑問符 (?) で示される) がない。

動的パラメータ・マーカは、実行時に SQL 文に代入される実データをユーザが指定できるようにするプレースホルダとして機能します。

一般に、即時実行方式は、アプリケーションが文を 1 回だけ実行するような場合に使用します。即時実行方式を使用して文を複数回実行することもできますが、この方式では、文を繰り返し準備することによるオーバヘッドがかかります。

即時実行コマンドのコーディング

即時実行方式を使用して動的 SQL 文を実行するには、次の手順に従ってアプリケーションをコーディングします。

1　動的 SQL 文のテキストを文字列ホスト変数に保管します。

2　*type* を CS_EXEC_IMMEDIATE に、*buffer* を SQL 文を含む文字列のアドレスに、*id* を NULL に設定して ct_dynamic を呼び出し、文を実行するコマンドを起動します。

3　ct_send を呼び出して、サーバにコマンドを送信します。

4　「基本ループの構造」(86 ページ) の説明に従って、ct_results を標準ループで呼び出します。**result_type* パラメータの値は、コマンドの実行が成功したか (CS_CMD_SUCCEED)、失敗したか (CS_CMD_FAIL) を示します。

準備実行方式の使い方

準備実行方式の場合、サーバはコンパイルを行い、個別のコマンドに応じて別々に、オペレーションを実行します。

準備実行方式の使用が適する場合

動的 SQL 文が次の基準のどちらかを満たしている場合、この方式を使用してください。

- データを返す。
- 実行時に入力される値のプレースホルダが含まれている (プレースホルダは文のテキスト内では疑問符 (?) で示される)。

アプリケーションが文を複数回実行するような場合には、この方式を使用してください。この方式では、文の準備に関係するオーバヘッドは、最初に文を準備するときだけにかかります。その後の文の実行では、文を再コンパイルするコストはかかりません。

準備実行方式は、即時実行方式に比べて、次のような利点があります。

- select 文を実行できます。
- 文が複数回実行される場合、パフォーマンスが高くなります。
- 文には、文を実行するたびに値を変更できるパラメータを指定できます。

準備実行方式のプログラム構造

ほとんどのアプリケーションは、次のような手順に従って、動的 SQL 文を準備し実行します。

1 動的 SQL 文を作成します。

- ct_dynamic(CS_PREPARE)
- ct_send
- ct_results (ループで)

準備コマンドはフェッチ可能な結果を返しません。

2 (オプション) 準備文を実行するのに必要なパラメータの記述を取得します。

- ct_dynamic(CS_DESCRIBE_INPUT)
- ct_send
- ct_results (ループで)

ct_results は、*result_type* を CS_DESCRIBE_RESULT に設定して戻り、パラメータ記述が使用可能であることを示します。

3 (オプション) 準備文から返される結果カラムの記述を取得します。

- ct_dynamic(CS_DESCRIBE_OUTPUT)
- ct_send
- ct_results (ループで)

ct_results は、*result_type* を CS_DESCRIBE_RESULT に設定して戻り、記述が使用できることを示します。

4 準備文を実行するか、準備文でカーソルを宣言し、オープンします。

次のコマンドで、準備文を実行します (カーソルを使用しない場合)。

- ct_dynamic(CS_EXECUTE)。
- 必要があれば、ct_param、ct_setparam、ct_dyndesc、または ct_dynsqlda でパラメータ値を定義します。

- ct_send
- ct_results (ループで) フェッチ可能な結果は処理してください。

カーソルを使用して準備文を実行する方法については、「Client-Library カーソルの使い方」(106 ページ) を参照してください。

5　準備文の割り付けを解除します。

文でカーソルが宣言されている場合、まずカーソルをクローズして、その割り付けを解除します。

- ct_cursor(CS_CURSOR_CLOSE, CS_DEALLOC)、またはカーソルがオープンしていない場合 ct_cursor(CS_CURSOR_DEALLOC)
- ct_send
- ct_results (ループで)
- 準備文の割り付けを解除するコマンドを起動し、送信します。
- ct_dynamic(CS_DEALLOC)
- ct_send
- ct_results (ループで)

割り付け解除コマンドはフェッチ可能な結果を返しません。

上記のプロセスの各手順が、1 つの動的 SQL コマンドをサーバに送信します。各コマンドを送信してから、アプリケーションは ct_results で結果を処理する必要があります。「基本ループの構造」(86 ページ) の説明に従って、動的 SQL コマンドの結果を標準結果ループで処理するようにアプリケーションをコーディングしてください。

手順 1：文を準備する

動的 SQL 文を準備するコマンドを起動するには、アプリケーションは、*type* を CS_PREPARE に、*id* を文字列の文識別子に、*buffer* を準備する文に設定して、ct_dynamic を呼び出します。次に例を示します。

```
char     *query = "select type, title, price ¥
         from titles ¥
         where title_id = ?"
ct_dynamic(cmd, CS_PREPARE, "myid", CS_NULLTERM,
     query, CS_NULLTERM);
```

文識別子は、同一接続で準備された他の動的 SQL 文との間でユニークにしてください。

ct_send でサーバに準備コマンドを送信し、標準 ct_results ループで結果を処理します。

手順 2：コマンド入力の記述を取得する

文を準備してから、アプリケーションは、文を実行するのに必要なパラメータの記述を得るために、入力記述コマンドをサーバに送信します。この記述には、入力値の数、データ型、長さなどがあります。アプリケーションはこの情報を使用して、エンド・ユーザに入力値を入力するよう指示します。入力値の入力を指示してから、アプリケーションは、文を実行する直前に、これらの入力値を準備文に渡します。

入力記述コマンドの起動

入力記述コマンドを起動するには、アプリケーションは、*type* を CS_DESCRIBE_INPUT に、*id* を文識別子に設定して ct_dynamic を呼び出します。ct_send でサーバにコマンドを送信し、標準 ct_results ループで結果を処理します。

パラメータ記述の処理

ct_results は *result_type* を CS_DESCRIBE_RESULT に設定して戻り、入力パラメータ・フォーマットが使用可能であることを示します。アプリケーションは、次の方法のどちらかでパラメータ・フォーマットを取得できます。

- ct_res_info と ct_describe を使用する方法

  アプリケーションは、ct_res_info を呼び出して、パラメータの数を決定します。次に、各パラメータについて ct_describe を呼び出し、パラメータの記述で CS_DATAFMT 構造体を初期化します。

  一般に、この方法を使用するアプリケーションは、CS_DATAFMT 構造体を、あとで ct_param または ct_setparam 呼び出しに使用できるように、配列またはリストの形で保管します。

- ct_dyndesc または ct_dynsqlda を使用する方法

  どちらのルーチンを使用しても、アプリケーションは、文を実行するコマンドにパラメータを渡すのにあとで使用する構造体に、フォーマットを取得して取り込むことができます。どちらのルーチンでも、次の処理が可能です。

  - 動的 SQL 準備文の実行に必要な入力パラメータの記述を取得する
  - 準備文を実行するための入力パラメータ値を定義する
  - 準備文の実行時に返される結果データの記述を取得する
  - 準備文の実行によって返された結果セットのデータ値を取得する

これらのルーチンの違いは次のとおりです。

- ct_dynsqlda － フォーマットを、SQLDA 構造体に取り込みます。アプリケーションは、構造体にメモリを割り付けてから、フォーマットを取得する必要があります。ct_dynsqlda は、オペレーションを行うごとに呼び出しが 1 回だけ必要です。
- ct_dyndesc － フォーマットを、アプリケーションに公開されていない内部 Client-Library データ構造体に取り込みます。ct_dyndesc は、1 つのオペレーションを行うために数回の呼び出しが必要です。

ct_dyndesc と ct_dynsqlda はどちらも、内部で ct_res_info と ct_describe を呼び出します。パラメータ値を渡すのに使用される場合、ct_dyndesc と ct_dynsqlda はどちらも、内部で ct_param を呼び出します。

手順 3：コマンド出力の記述を取得する

アプリケーションは、出力記述コマンドを送信して、準備文を実行すると返される結果カラムのフォーマットを取得できます。たとえば、対話型アプリケーションは、クエリ結果を表示するときに使用するデータ構造体を準備するために、出力記述コマンドを使用して、結果カラムの数とフォーマットを決定します。出力記述コマンドを使用すれば、アプリケーションは準備文を実行しないで結果フォーマットを決定できます。

注意　1 回の動的 SQL バッチに、複数の SQL 文を含めることができます。ただし、準備文による出力記述には、最初の結果セットの記述のみ含まれます。動的 SQL 文を実行したときのみ、それぞれの結果セットのすべての記述が受信されます。

出力記述コマンドの起動

出力記述コマンドを起動するには、アプリケーションは、*type* を CS_DESCRIBE_OUTPUT に、*id* を文識別子に設定して ct_dynamic を呼び出します。ct_send でサーバにコマンドを送信し、標準 ct_results ループで結果を処理します。

カラム記述の処理

ct_results は *result_type* を CS_DESCRIBE_RESULT に設定して戻り、結果カラム・フォーマットが使用可能であることを示します。アプリケーションは、次の 2 つの方法のどちらかでカラム・フォーマットを取得できます。

- ct_res_info と ct_describe を使用する方法

  アプリケーションは、ct_res_info を呼び出して、カラム数を取得し、次に各パラメータについて ct_describe を呼び出して、カラムの記述で CS_DATAFMT 構造体を初期化します。

  一般に、この方法を使用するアプリケーションは、あとの ct_bind 呼び出しで使用できるように、CS_DATAFMT 構造体を配列またはリストの形で保管します

- ct_dyndesc または ct_dynsqlda を使用する方法

  どちらのルーチンを使用しても、アプリケーションは、準備文を実行するときにロー・データを取得するのにあとで使用する構造体に、フォーマットを取得できます。

  - ct_dynsqlda はフォーマットを、SQLDA 構造体に取り込みます。アプリケーションは、構造体にメモリを割り付けてから、フォーマットを取得する必要があります。
  - ct_dyndesc はフォーマットを、アプリケーションに公開されていない内部 Client-Library データ構造体に取り込みます。

  ct_dyndesc と ct_dynsqlda はどちらも、内部で ct_res_info と ct_describe を呼び出します。ロー・データを取得するのに使用する場合、ct_dyndesc と ct_dynsqlda はどちらも、内部で ct_bind を呼び出します。

手順 4：準備文を実行する

準備文を実行するコマンドを起動するには、アプリケーションは、*type* を CS_EXECUTE に、*id* を文識別子に設定して ct_dynamic を呼び出します。アプリケーションは、準備文を実行するのに必要なすべてのパラメータを定義してください。パラメータ値は次のどれかの方法で定義できます。

- 各パラメータについて ct_param を 1 回呼び出す方法 − ct_param と ct_setparam はどちらも最高のパフォーマンスを提供しますが、ct_param では、アプリケーションはコマンドを再送信する前にパラメータ値を変更できません。
- 各パラメータについて ct_setparam を 1 回呼び出す方法 − ct_setparam はパラメータ送信元値へのポインタを持ちます。この方法は、コマンドを再送信する前にパラメータ値を変更できる唯一の方法です。

- ct_dyndesc を数回呼び出す方法 − 動的記述子領域を割り付けて、そこにデータ値を挿入して、コマンドに適用します。ct_dyndesc(CS_USE_DESC) は内部で ct_param を呼び出します。
- ct_dynsqlda を呼び出す方法 − この方法では、ユーザ割り付けの SQLDA 構造体の内容がコマンドに適用されます。ct_dynsqlda(CS_SQLDA_PARAM) は内部で ct_param を呼び出します。

アプリケーションは、準備文を実行する前に、入力記述コマンドを送信し、その結果を処理することによって、準備文のパラメータの数とフォーマットを決定できます。「手順 2：コマンド入力の記述を取得する」(128 ページ) を参照してください。

ct_send でサーバに準備コマンドを送信し、標準 ct_results ループで結果を処理します。「基本ループの構造」(86 ページ) の説明に従って、結果を標準結果ループで処理するようにアプリケーションをコーディングしてください。

手順 5：準備文の割り付けを解除する

準備文の割り付けを解除すると、準備文に対応していたすべてのリソースが解放されます。明示的な割り付け解除はオプションです。アプリケーションが明示的に準備文の割り付けを解除しない場合には、クライアント・プログラムが切り離されたときに、サーバが割り付けを解除します。

準備文でカーソルを宣言している場合、アプリケーションは、カーソルの割り付けを解除してから、文の割り付けを解除する必要があります。詳細については、「手順 6：カーソルの割り付けを解除する」(119 ページ) を参照してください。

準備文の割り付けを解除するコマンドを起動するには、アプリケーションは、*type* を CS_DEALLOC に、*id* を文識別子に設定して ct_dynamic を呼び出します。ct_send でサーバにコマンドを送信し、標準 ct_results ループで結果を処理します。

動的 SQL とストアド・プロシージャの比較

動的 SQL よりも高いパフォーマンスを達成するために、アプリケーション設計者は、アプリケーションの稼働条件が許す場合には、動的 SQL の代わりに、Adaptive Server Enterprise のストアド・プロシージャを使用できます。

動的 SQL とストアド・プロシージャには、次のような類似点があります。

- ストアド・プロシージャの作成は、動的 SQL 文の作成と類似しています。
- ストアド・プロシージャの入力パラメータは、動的パラメータ・マーカと同じ機能を持ちます。
- ストアド・プロシージャの実行は、準備文の実行と同じです。

ストアド・プロシージャと動的 SQL 準備文は、同じような機能を提供しますが、次の点だけは異なります。

- 動的 SQL では、準備文のパラメータ・フォーマットを取得できますが、ストアド・プロシージャではできません。「手順 2：コマンド入力の記述を取得する」(128 ページ) を参照してください。
- ストアド・プロシージャの結果のフォーマットは、プロシージャを実行しないでプログラム的に簡単に定義することはできません。動的 SQL の場合、準備文の結果カラム・フォーマットは文を実行しなくても取得できます。「手順 3：コマンド出力の記述を取得する」(129 ページ) を参照してください。
- ユーザ作成のストアド・プロシージャは永久データベース・オブジェクトですが、ユーザがサーバから切り離されると、準備文は、自動的に割り付けを解除されます。

動的 SQL 文は、同じ結果を返すストアド・プロシージャで置き換えることができます。たとえば、次の動的 SQL 文は、*pubs2..titles* テーブルに問い合わせて、価格範囲と種類を特定して本を検索します。

```
select * from pubs2..titles
     where type = ?
     and price between ?and ?
```

この例にある動的 SQL 文では、1 つの *type* 値と 2 つの *price* 値に動的パラメータ・マーカ (?) が付いています。

上の動的 SQL と同等のストアド･プロシージャは次のようにして作成できます。

```
create proc titles_type_pricerange
   @type char(12),
   @price1 money,
   @price2 money
 as
   select * from titles
     where
        type = @type
        and price between @price1 and @price2
```

上の準備文とストアド・プロシージャは、同じ入力パラメータ値で実行すると、同じローを返します。ストアド・プロシージャは、リターン・ステータス結果も返します。

第 9 章

ディレクトリ・サービスの使い方

この章では、Client-Library アプリケーションでディレクトリ・サービスを使用する方法を説明します。

トピック名	ページ
ディレクトリ・サービスとは	133
アプリケーションはディレクトリ・サービスをどのように使用するか	134
ディレクトリの検索	134
手順 1：検索を開始する	135
手順 2：ディレクトリ・コールバックで検索結果を収集する	140
手順 3：ディレクトリ・オブジェクトを検査する	143
手順 4：クリーンアップする	157

ディレクトリ・サービスとは

「ディレクトリ」は、情報を「ディレクトリ・エントリ」として保管し、各エントリと論理名を対応させます。各ディレクトリ・エントリの内容は、ユーザ、サーバ、プリンタなどのネットワーク・エンティティに関する情報です。

「ディレクトリ・サービス」(ネーミング・サービスと呼ばれることもあります) は、ディレクトリ・エントリの作成、修正、取得を管理します。

デフォルトでは、Client-Library は Sybase の interfaces ファイルをディレクトリ・ソースとして使用します。Sybase では、DCE の Cell Directory Service (CDS) や Windows レジストリ・サービスなど、いくつかのネットワーク・ベースのディレクトリ・サービス用にディレクトリ・ドライバも提供しています。プラットフォームごとの使用可能なディレクトリ・ドライバについては、『Open Client/Server 設定ガイド Windows 版』または『Open Client/Server 設定ガイド UNIX 版』を参照してください。

アプリケーションはディレクトリ・サービスをどのように使用するか

Sybase のサーバに関する情報は、ディレクトリに保管されます。アプリケーションは、ct_connect を呼び出してサーバ接続をオープンするとき、ct_connect の *server_name* パラメータとして該当サーバのディレクトリ・エントリの名前を渡します。ct_connect は、エントリを参照して、該当サーバのネットワーク・アドレスや、接続を確立するのに必要なその他の情報を検索します。

アプリケーションは、Client-Library ルーチンを使用して、使用できるサーバを検索できます。

ディレクトリの検索

アプリケーションは、Client-Library プログラミング環境を設定して、CS_CONNECTION 構造体を割り付けてから、ディレクトリを検索する必要があります。Client-Library を初期化して接続構造体を割り付ける方法については、「第 1 章 Client-Library を使用する前に」を参照してください。

サンプル・プログラム

usedir.c は、Client-Library アプリケーションがどのようにしてディレクトリの検索を行うかを示すサンプル・プログラムです。この章で示すプログラムのすべての例は、*usedir.c* からの引用です。

プログラム構造

ディレクトリ検索を行うには、次の手順に従ってアプリケーションをコーディングします。

1 検索を開始します。

- ct_con_props – ディレクトリ・サービス・プロパティを設定します。
- ct_callback – 接続構造体にアプリケーションのディレクトリ・コールバックへのポインタをインストールします。

アプリケーション・コードを実行して、ディレクトリ・オブジェクトを収集するリストまたは配列を初期化します。

- ct_ds_lookup – 検索を開始します。

アプリケーションは、ct_callback をここで呼び出す代わりに、接続を割り付ける前に、接続の親コンテキスト構造体にコールバックをあらかじめインストールしておくことができます。コールバックは、コンテキストから割り付けられるすべての接続のデフォルト・ディレクトリ・コールバックになります。

2　ディレクトリ・コールバックで検索結果を収集します。

- (オプション) ct_ds_objinfo － オブジェクトを検査します。
- (オプション) ct_ds_dropobj － 不要なオブジェクトを削除します。

アプリケーション・コードを実行して、アプリケーション定義のリストまたは配列でディレクトリ・オブジェクトを収集します。

ディレクトリ検索時に、ct_ds_lookup は検索中に見つかった各エントリにつき 1 回ずつ、ディレクトリ・コールバックを呼び出します。

3　ディレクトリ・オブジェクトを検査します。各ディレクトリ・オブジェクトについて、次のコマンドが必要です。

- ct_ds_objinfo － オブジェクトの完全修飾名を取得します。
- ct_ds_objinfo － 属性の数を取得します。
- ct_ds_objinfo － 各属性のメタデータと値を取得します。

4　クリーンアップします。

ct_ds_dropobj (各オブジェクトについて) － ディレクトリ・オブジェクトの割り付けを解除します。

手順 1：検索を開始する

アプリケーションは、結果を保管するアプリケーション・データ構造体を初期化し、ディレクトリ・コールバックをインストールし、ct_ds_lookup を呼び出して、ディレクトリ検索を開始します。

データ構造体の初期化

この章のサンプル・コードは、SERVER_INFO_LIST という名前のデータ構造体にディレクトリ・オブジェクトを収集します。このデータ構造体は CS_DS_OBJECT ポインタの配列またはリストとして実装することができます。

このコードは、次のサンプル・ルーチンを呼び出して、ディレクトリ・オブジェクト構造体を収集します。

- sil_init_list － 空の SERVER_INFO_LIST を割り付け、初期化します。

- sil_add_object − ディレクトリ・オブジェクトを SERVER_INFO_LIST の末尾に追加します。
- sil_extract_object − ベース 1 のインデックス番号として、SERVER_INFO_LIST からディレクトリ・オブジェクトを取得します。
- sil_list_len − SERVER_INFO_LIST に保管されているオブジェクトの数を取得します。
- sil_drop_list − SERVER_INFO_LIST とそのすべての構成要素の割り付けを解除します。リスト内の各ディレクトリ・オブジェクトの割り付けを解除するために、ct_ds_dropobj を呼び出します。

これらのルーチンは CS_DS_OBJECT ポインタのリストを管理するだけです。ここでは、これらの実装の例は示しません。Client-Library サンプル・プログラム内の *usedir.c* サンプル・ファイルに完全なコードが入っていますので、参照してください。

ディレクトリ・サービス・プロパティの設定

アプリケーションは、ct_con_props を呼び出して、接続のディレクトリ・サービス・プロパティを設定します。一般に、次のプロパティを設定して、ディレクトリ検索を制御します。

- CS_DS_DITBASE − 検索を開始するディレクトリのノードを指定します。DIT ベース値はディレクトリ・サービスの構文規則に従ってください。サンプル DIT ベース値については、『Open Client Client-Library/C リファレンス・マニュアル』の「ディレクトリ・サービス」の項を参照してください。
- CS_DS_SEARCH − 検索を DIT ベースの下のどこまで行うか、その深度を制限します。CS_DS_SEARCH の有効な値は次のとおりです。

値	意味
CS_SEARCH_ONE_LEVEL (デフォルト)	検索対象は、CS_DS_DITBASE に指定されているノードの直接下層であるリーフ・エントリだけとする。
CS_SEARCH_SUBTREE	CS_DS_DITBASE で指定されたルートの下のサブツリー全体を検索する。

注意 DCE ディレクトリ・ドライバの場合、CS_DS_SEARCH をデフォルトの CS_SEARCH_ONE_LEVEL 以外の値に設定することはできません。

すべてのディレクトリ・サービス・プロパティは、「CS_DS」で始まる記号名を持っています。Client-Library プロパティの完全なリストについては、『Open Client Client-Library/C リファレンス・マニュアル』の「プロパティ」の項を参照してください。

ディレクトリ・コールバックのインストール

アプリケーションは、*action* パラメータを CS_SET に、*type* パラメータを CS_DS_LOOKUP_CB に、*func* パラメータをアプリケーション・ディレクトリ・コールバック・ルーチンのアドレスに設定し、ct_callback を呼び出して、ディレクトリ・コールバックをインストールします。

ディレクトリ・コールバックはコンテキスト・レベルか接続レベルでインストールできます。コンテキストから割り付けられた接続は、そのコンテキストのディレクトリ・コールバックを継承します。この場合には、コールバックを接続レベルでインストールします。

コールバック・ルーチンのコーディングについては、「手順 2：ディレクトリ・コールバックで検索結果を収集する」(140 ページ) で説明します。

ct_ds_lookup の呼び出し

アプリケーションは、*action* を CS_SET に設定して ct_ds_lookup を呼び出すことによって、検索を開始します。

ct_ds_lookup は、検索要求を記述する *lookup_info* パラメータとして CS_DS_LOOKUP_INFO 構造体を取ります。*lookup_info–>objclass* は、ディレクトリ・オブジェクト・クラス CS_OID_OBJSERVER を示す CS_OID 構造体を指している必要があります。そのほかの CS_DS_LOOKUP_INFO フィールドは現時点では使用されていないので、すべて NULL として渡してください。

ct_ds_lookup は、*userdata* パラメータとしてユーザ割り付けのデータへのポインタも持ちます。ct_ds_lookup がアプリケーションのディレクトリ・コールバックを呼び出すと、コールバックは同じポインタ値を入力パラメータとして受信します。

ディレクトリ検索を開始するサンプル・プログラム

次のサンプル・プログラムは、サーバ・ディレクトリ・クラス・オブジェクトを検索するアプリケーション・ルーチン get_servers を宣言します。

```
/*
** get_servers() -- Query the directory for servers and
**    get a list of directory objects that contain details
**    for each.
**
** Parameters
**   conn -- Pointer to allocated connection structure.
**   pserver_list -- Address of a pointer to a SERVER_INFO_LIST.
**       Upon successful return, the list will be initialized
**       and contain an object for each server found in the
**       search.
**
**       NOTE:The caller must clean up the list with sil_drop_list()
```

```
**       when done with it.
**
** Returns
**   CS_SUCCEED or CS_FAIL.
*
CS_RETCODE get_servers (conn, pserver_list)
CS_CONNECTION *conn;
SERVER_INFO_LIST **pserver_list;
{
  CS_RETCODE          ret;
  CS_INT              reqid;
  CS_VOID             *oldcallback;
  CS_OID              oid;
  CS_DS_LOOKUP_INFO   lookup_info;

  /*
  ** Steps for synchronous-mode directory searches:
  **
  ** 1. If necessary, initialize application specific data structures
  **    (Our application collects directory objects in *pserver_list).
  ** 2. Save the old directory callback and install our own.
  ** 3. Set the base node in the directory to search beneath
  **    (CS_DS_DITBASE property).
  ** 4. Call ct_ds_lookup to begin the search, passing any application
  **    specific data structures as the userdata argument.
  ** 5. Client-Library invokes our callback once for each found object
  **    (or once to report that no objects were found).The callback
  **    (directory_cb) receives pointers to found servers and appends
  **    each to the list of servers.
  ** 6. Check the return status of ct_ds_lookup.
  ** 7. Restore callbacks and properties that we changed.
  */

  /*
  ** Step 1. Initialize the data structure (*pserver_list).
  */
  ret = sil_init_list(pserver_list);
  if (ret != CS_SUCCEED || (*pserver_list) == NULL)
  {
    ex_error("get_servers:Could not initialize list.");
    return CS_FAIL;
  }

  /*
  ** Step 2. Save the old directory callback and install our own callback,
  **   directory_cb(), to receive the found objects.
  */
  ret = ct_callback(NULL, conn, CS_GET,
                    CS_DS_LOOKUP_CB, &oldcallback);
  if (ret == CS_SUCCEED)
  {
    ret = ct_callback(NULL, conn, CS_SET,
```

```
                    CS_DS_LOOKUP_CB, (CS_VOID *)directory_cb);
}
if (ret != CS_SUCCEED)
{
  ex_error("get_servers:Could not install directory callback.");
  return CS_FAIL;
}

/*
** Step 3. Set the base node in the directory to search beneath
**    (the CS_DS_DITBASE connection property).
*/

ret = provider_setup(conn);
if (ret != CS_SUCCEED)
{
  ex_error("get_servers:Provider-specific setup failed.");
  return CS_FAIL;
}

/*
** Step 4. Call ct_ds_lookup to begin the search, passing the server list
**    pointer as userdata.
** Step 5. Client-Library invokes our callback once for each found object
**    (or once to report that no objects were found).Our callback,
**    directory_cb, will receive a pointer to each found server object
**    and appends it to the list.
** Step 6. Check the return status of ct_ds_lookup.
*/

/*

** Set the CS_DS_LOOKUP_INFO structure fields.
*/
lookup_info.path = NULL;
lookup_info.pathlen = 0;
lookup_info.attrfilter = NULL;
lookup_info.attrselect = NULL;

strcpy(oid.oid_buffer, CS_OID_OBJSERVER);
oid.oid_length = STRLEN(oid.oid_buffer);
lookup_info.objclass = &oid;

/*
** Begin the search.
*/
ret = ct_ds_lookup(conn, CS_SET, &reqid,
                   &lookup_info, (CS_VOID *)pserver_list);
if (ret != CS_SUCCEED)
{
  ex_error("get_servers:Could not run search.");
  return CS_FAIL;
}
```

```
   /*
   ** Step 7.  Restore callbacks and properties that we changed.
   */
   ret = ct_callback(NULL, conn, CS_SET,
                     CS_DS_LOOKUP_CB, oldcallback);
   if (ret != CS_SUCCEED)
   {
     ex_error("get_servers:Could not restore directory callback.");
     return CS_FAIL;
   }

 return CS_SUCCEED;

 } /* get_servers() *
```

手順 2：ディレクトリ・コールバックで検索結果を収集する

ディレクトリ検索時に、ct_ds_lookup は検索中に見つかった各エントリにつき 1 回ずつ、ディレクトリ・コールバックを呼び出します。

ディレクトリ・コールバックの定義

ディレクトリ・コールバックのプロトタイプは次のとおりです。

```
CS_RETCODE CS_PUBLIC
directory_cb (connection, reqid, status,
              numentries, ds_object, userdata)
CS_CONNECTION   *connection;
CS_INT          reqid;
CS_RETCODE      status;
CS_INT          numentries;
CS_DS_OBJECT    *ds_object;
CS_VOID         *userdata;
```

パラメータの意味は次のとおりです。

- *connection* は、ディレクトリ検索に使用される CS_CONNECTION 構造体を指すポインタです。
- *reqid* は、ディレクトリ検索を開始した ct_ds_lookup 呼び出しによって返される要求識別子です。

- *status* は、ディレクトリ検索要求のステータスです。*status* は次のいずれかの値を取ります。

ステータス値	意味
CS_SUCCEED	検索は成功した。
CS_FAIL	検索は失敗した。
CS_CANCELED	検索は ct_ds_lookup(CS_CLEAR) でキャンセルされた。

- *numentries* は、調べる対象として残っているディレクトリ・オブジェクトの数です。エントリが見つかった場合、*numentries* には、現在のオブジェクトが含まれます。エントリが見つからなかった場合、*numentries* は 0 です。
- *ds_object* は、1 つのディレクトリ・オブジェクトについての情報へのポインタです。*ds_object* は、次のいずれかの条件に合う場合は (CS_DS_OBJECT *)NULL です。
  - ディレクトリ検索が失敗した (*status* 値が CS_SUCCEED ではない)。
  - 一致するオブジェクトが見つからなかった (*numentries* 値が 0 以下であった)。
- *userdata* は、ユーザが供給するデータ領域を指すポインタです。アプリケーションが ct_ds_lookup の *userdata* パラメータとしてポインタを渡した場合、ディレクトリ・コールバックは呼び出されたときに同じポインタを受信します。

  userdata はコールバックがメインライン・コードと通信する手段のひとつです。

コールバックは CS_CONTINUE または CS_SUCCEED を返します。

- CS_SUCCEED が返されると、検索結果はトランケートされます。Client-Library は、残りのディレクトリ・オブジェクトを廃棄して、コールバックの呼び出しを停止します。
- CS_CONTINUE が返されると、Client-Library は検索結果の次のディレクトリ・オブジェクトでコールバックを呼び出します。

ディレクトリ・コールバックの例

次のサンプル・プログラムは、ディレクトリ・コールバックを定義します。このコールバックは次のことを行います。

- ディレクトリ・オブジェクト・ポインタが有効であるかどうかを確認します。
- sil_add_object サンプル・ルーチンを呼び出して、アプリケーションのサーバ・リストにディレクトリ・オブジェクトを追加します。メインライン・コードは、ct_ds_lookup を呼び出すとき、初期化された SERVER_INFO_LIST のアドレスを ct_ds_lookup の *userdata* パラメータとして渡します。コールバックは同じアドレスを自身の *userdata* パラメータとして受信します。
- サーバのリストが満杯である場合、コールバックは CS_SUCCEED を返して、検索結果をトランケートします。それ以外の場合、コールバックは CS_CONTINUE を返します。

```
/*
** directory_cb() -- Directory callback to install in Client-Library.
**   When we call ct_ds_lookup(), Client-Library calls this function
**   once for each object that is found in the search.
**
**   This particular callback collects the objects in
**   the SERVER_INFO_LIST that is received as userdata.
**
** Parameters
**   conn -- The connection handle passed to ct_ds_lookup() to
**      begin the search.
**   reqid -- The request id for the operation (assigned by Client-Library).
**   status -- CS_SUCCEED when search succeeded (ds_object is valid).
**      CS_FAIL if the search failed (ds_object is not valid).
**   numentries -- The count of objects to be returned for the
**      search.Includes the current object.Can be 0 if search
**      failed.
**   ds_object -- Pointer to a CS_DS_OBJECT structure.Will
**      be NULL if the search failed.
**   userdata -- The address of user-allocated data that was
**      passed to ct_ds_lookup().
**
**      This particular callback requires userdata to be the
**      address of a valid, initialized SERVER_INFO_LIST pointer.
**      (SERVER_INFO_LIST is an application data structure defined
**      by this sample).
**
** Returns
**   CS_CONTINUE unless the SERVER_INFO_LIST pointed at by userdata fills
**     up, then CS_SUCCEED to truncate the search results.
*/

S_RETCODE          CS_PUBLIC
directory_cb(conn, reqid, status, numentries, ds_object, userdata)
CS_CONNECTION      *conn;
```

```
CS_INT              reqid;
CS_RETCODE          status;
CS_INT              numentries;
CS_DS_OBJECT       *ds_object;
CS_VOID            *userdata;
{

    CS_RETCODE          ret;
    SERVER_INFO_LIST   *server_list;

    if (status != CS_SUCCEED || numentries <= 0)
    {
        return CS_SUCCEED;
    }
    /*
    ** Append the object to the list of servers.
    */
    server_list = *((SERVER_INFO_LIST **)userdata);
    ret = sil_add_object(server_list, ds_object);
    if (ret != CS_SUCCEED)
    {

        /*
        ** Return CS_SUCCEED to discard the rest of the objects that were
        ** found in the search.
        */
        ex_error(
           "directory_cb:Too many servers!Truncating search results.");
        return CS_SUCCEED;
    }
    /*
    ** Return CS_CONTINUE so Client-Library will call us again if more
    ** entries are found.
    */
    return CS_CONTINUE;

} /* directory_cb() */
```

手順 3：ディレクトリ・オブジェクトを検査する

アプリケーションは、ct_ds_objinfo を数回呼び出して、ディレクトリ・オブジェクトの内容を検査します。アプリケーションが参照できるディレクトリ・オブジェクト項目は次の 3 つです。

- オブジェクトが属するオブジェクト・クラス
- オブジェクトの完全修飾名
- 番号付き属性セット

オブジェクトのディレクトリ・オブジェクト・クラスは、オブジェクトの属性と各属性の値の予期構文 (つまり、データ型) を定義します。

オブジェクト属性は番号付きセットとして表示されますが、アプリケーションは属性が返される順序に関係なく機能するようにコーディングしてください。ディレクトリ・オブジェクト・クラスには属性の順序は定義されないので、ほとんどのディレクトリ・サービスの場合、オブジェクト・クラスが同じでもディレクトリ・オブジェクトが異なると属性の順序が同じになるとは限りません。

ほとんどのアプリケーションは、次に示すようなプログラム構造を使用して、ディレクトリ・オブジェクトを検査します。

```
ct_ds_objinfo to get the directory object class (optional)
 ct_ds_objinfo to get the fully qualified name
... application code to process fully qualified name ...
for each desired attribute type
  ct_ds_objinfo to get number of attributes
  i = 0
  while i is less than number of attributes
    i = i + 1
    ct_ds_objinfo to retrieve the metadata for attribute i
    compare returned attribute type to desired attribute type
    if attribute types match
       /* i is the number of the desired attribute */
       break while
    end if
  end while
  allocate sufficient space for attribute i’s values
  ct_ds_objinfo to retrieve attribute i’s values
  ... application code to process attribute values ...
end for
```

属性データ構造体

属性のメタデータは、次のような CS_ATTRIBUTE 構造体で表します。

```
typedef struct _cs_attribute
 {
     CS_OID          attr_type;
     CS_INT          attr_syntax;
     CS_INT          attr_numvals;
 } CS_ATTRIBUTE;
```

各オブジェクトの意味は、次のとおりです。

- *attr_type* は、属性のタイプをユニークに記述した CS_OID 構造体です。このフィールドによって、アプリケーションはオブジェクトのどの属性を受信したかがわかります。
- *attr_syntax* は、属性値をどのように表現するかを指示する構文指定子です。属性値は CS_ATTRVALUE 共用体内に渡され、構文指定子は使用する共用体のメンバを示します。
- *attr_numvals* は、属性に含まれている値の数を指示します。この情報は、属性値を保管する CS_ATTRVALUE 共用体の配列のサイズを決定するのに使用します。

属性の値は CS_ATTRVALUE 共用体によって表します。

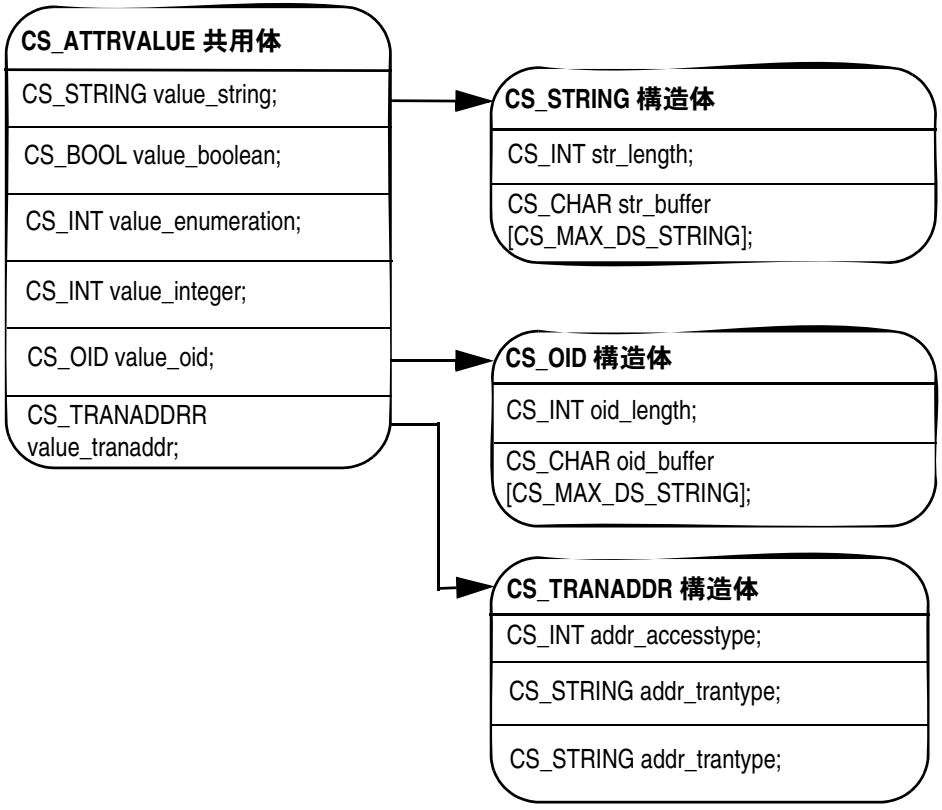
```
typedef struct _cs_ds_lookup_info
 {
     CS_OID          *objclass;
     CS_CHAR         *path;
     CS_INT          pathlen;
     CS_DS_OBJECT    *attrfilter;
     CS_DS_OBJECT    *attrselect;
 } CS_DS_LOOKUP_INFO;
```

アプリケーションは、CS_ATTRIBUTE 構造体の *syntax* フィールドをチェックして、CS_ATTRVALUE 共用体のどのメンバに実値が入っているかを定義します。次の表に、両者の対応を示します。

CS_ATTRIBUTE 構文指定子	CS_ATTRVALUE 共用体メンバ
CS_ATTR_SYNTAX_STRING	*value_string*
CS_ATTR_SYNTAX_BOOLEAN	*value_boolean*
CS_ATTR_SYNTAX_INTEGER	*value_integer*
CS_ATTR_SYNTAX_TRANADDR	*value_tranaddr*
CS_ATTR_SYNTAX_OID	*value_oid*

図 9-1 は CS_ATTRVALUE 共用体とそのメンバ構造体の展開図です。

図 *9-1: CS_ATTRVALUE* 共用体の展開図



ディレクトリ・オブジェクトを検査するサンプル・プログラム

次のサンプル・プログラムでは、ディレクトリ・オブジェクトの内容をテキストとして出力するサンプル・ルーチン show_server_info を宣言します。

このコーディングは、静的配列 *AttributesToDisplay* を使用して、値を取得する属性の属性タイプを (OID 文字列として) 出力順にリストします。

この例では、*AttributesToDisplay* 内の各ローについて、属性タイプの値を取得し、出力します。

```
/*
** AttributesToDisplay is a read-only static array used by
** the show_server_info() function.It contains the Object
** Identifier (OID) strings for the server attributes to
** display, in the order that they are to be displayed.
*/
typedef struct
{
   CS_CHAR             type_string[CS_MAX_DS_STRING];
   CS_CHAR             english_name[CS_MAX_DS_STRING];
} AttrForDisplay;
#define N_ATTRIBUTES 7
CS_STATIC AttrForDisplay AttributesToDisplay[N_ATTRIBUTES + 1] =
{
   {CS_OID_ATTRSERVNAME, "Server name"},
   {CS_OID_ATTRSERVICE, "Service type"},
   {CS_OID_ATTRVERSION, "Server entry version"},
   {CS_OID_ATTRSTATUS, "Server status"},
   {CS_OID_ATTRADDRESS, "Network addresses"},
   {CS_OID_ATTRRETRYCOUNT, "Connection retry count"},
   {CS_OID_ATTRLOOPDELAY, "Connection retry loop delay"},
   {"", ""}
};
/*
** show_server_info()
**   Selectively display the attributes of a server directory
**   object.
**
** Parameters
**   ds_object -- Pointer to the CS_DS_OBJECT that describes the
**     server's directory entry.
**   outfile -- Open FILE handle to write the output to.
**
** Dependencies
**   Reads the contents of the AttributesToDisplay global array.
**
** Returns
**   CS_SUCCEED or CS_FAIL.
*/
CS_RETCODE
show_server_info(ds_object, outfile)
CS_DS_OBJECT       *ds_object;
FILE               *outfile;
{
   CS_RETCODE          ret;
   CS_CHAR             scratch_str[512];
   CS_INT              outlen;
   CS_INT              cur_attr;
   CS_ATTRIBUTE        attr_metadata;
   CS_ATTRVALUE       *p_attrvals;
```

```
/*
** Distinguished name of the object.
*/
   ret = ct_ds_objinfo(ds_object, CS_GET, CS_DS_DIST_NAME, CS_UNUSED,
                       (CS_VOID *)scratch_str, CS_SIZEOF(scratch_str),
                        &outlen);
   if (ret != CS_SUCCEED)
   {
      ex_error("show_server_info:get distinguished name failed.");
      return CS_FAIL;
   }

   fprintf(outfile, "Name in directory:%s¥n", scratch_str);

   for (cur_attr = 0; cur_attr < N_ATTRIBUTES; cur_attr++)
   {
   /*
   ** Look for the attribute. attr_get_by_type() fails if the object
   ** instance does not contain a value for the attribute.If this
   ** happens, we just go on to the next attribute.
   */
      ret = attr_get_by_type(ds_object,
                             AttributesToDisplay[cur_attr].type_string,
                             &attr_metadata, &p_attrvals);
      if (ret == CS_SUCCEED)
      {
         fprintf(outfile, "%s:¥n",
         AttributesToDisplay[cur_attr].english_name);
        /*
        ** Display the attribute values.
         */
         ret = attr_display_values(&attr_metadata, p_attrvals, outfile);
         if (ret != CS_SUCCEED)
         {
            ex_error(
            "show_server_info:display attribute values failed.");
            free(p_attrvals);
            return CS_FAIL;
         } /* if */
         free(p_attrvals);
      } /* if */
   } /* for */

   return CS_SUCCEED;
} /* show_server_info() */
```

属性値の取得

次の例には、attr_get_by_type サンプル・ユーティリティ・ルーチンのコードが含まれています。attr_get_by_type は、希望の属性タイプを指定する OID 文字列を持ち、ディレクトリ・オブジェクトの属性セット内で希望の属性を検索して、その属性のメタデータと値が見つかれば、それらを返します。

```
/*
** get_attr_by_type()
**   Get metadata and attribute values for a given attribute type.
**
** Parameters
**   ds_object -- Pointer to a valid CS_DS_OBJECT hidden structure.
**   attr_type_str -- Null-terminated string containing the OID for the
**            desired attribute type.
**   attr_metadata -- Pointer to a CS_ATTRIBUTE structure to
**            fill in.
**   p_attrvals -- Address of a CS_ATTRVALUE union pointer.
**            If successful, this routine allocates an array
**            of size attr_metadata->numvalues, retrieves values into
**            it, and returns the array address in *p_attr_values.
**            NOTE:The caller must free this array when it is no longer
**            needed.
**
** Returns
**   CS_FAIL if no attribute of the specified type was found.
**   CS_SUCCEED for success.
*/

CS_RETCODE
attr_get_by_type(ds_object, attr_type_str, attr_metadata, p_attrvals)
CS_DS_OBJECT       *ds_object;
CS_CHAR            *attr_type_str;
CS_ATTRIBUTE       *attr_metadata;
CS_ATTRVALUE      **p_attrvals;
{
   CS_RETCODE          ret;
   CS_INT              num_attrs;
   CS_INT              cur_attr;
   CS_INT              outlen;
   CS_INT              buflen;
   CS_BOOL             found = CS_FALSE;

   /*
   ** Check input pointers.If not NULL, make them fail safe.
   */
   if (attr_metadata == NULL || p_attrvals == NULL)
   {
      return CS_FAIL;
   }
   attr_metadata->attr_numvals = 0;
   *p_attrvals = NULL;
```

```
/*
** Get number of attributes.
*/
ret = ct_ds_objinfo(ds_object, CS_GET, CS_DS_NUMATTR, CS_UNUSED,
                    (CS_VOID *)#_attrs, CS_SIZEOF(num_attrs),
                     NULL);
if (ret != CS_SUCCEED)
{
   ex_error("attr_get_by_type:get number of attributes failed.");
   return CS_FAIL;
}

/*
** Look for the matching attribute, get the values if found.
*/
for (cur_attr = 1;
     cur_attr <= num_attrs && found != CS_TRUE;
     cur_attr++)
{
   /*
   ** Get the attribute's metadata.
   */
   ret = ct_ds_objinfo(ds_object, CS_GET, CS_DS_ATTRIBUTE, cur_attr,
                       (CS_VOID *)attr_metadata,
                        CS_SIZEOF(CS_ATTRIBUTE), NULL);
   if (ret != CS_SUCCEED)
   {
      ex_error("attr_get_by_type:get attribute failed.");
      return CS_FAIL;
   }

   /*
   ** Check for a match.
   */
   if (match_OID(&(attr_metadata->attr_type), attr_type_str))
   {
      found = CS_TRUE;
      /*
      ** Get the values -- we first allocate an array of
      ** CS_ATTRVALUE unions.
      */
      *p_attrvals = (CS_ATTRVALUE *) malloc(sizeof(CS_ATTRVALUE)
                                        * (attr_metadata->attr_numvals));
      if (p_attrvals == NULL)
      {
         ex_error("attr_get_by_type:out of memory!");
         return CS_FAIL;
      }
      buflen = CS_SIZEOF(CS_ATTRVALUE) * (attr_metadata->attr_numvals);
      ret = ct_ds_objinfo(ds_object, CS_GET, CS_DS_ATTRVALS, cur_attr,
                          (CS_VOID *)(*p_attrvals), buflen, &outlen);
      if (ret != CS_SUCCEED)
```

```
         {
            ex_error("attr_get_by_type:get attribute values failed.");
            free(*p_attrvals);
            *p_attrvals = NULL;
            attr_metadata->attr_numvals = 0;
            return CS_FAIL;
         }
      }
   }

   /*
   ** Got the attribute.
   */
   if (found == CS_TRUE)
   {
      return CS_SUCCEED;
   }
/*
** Not found.
*/
   attr_metadata->attr_numvals = 0;
   return CS_FAIL;
} /* attr_get_by_type() */


/*
** match_OID()
**              Compare a pre-defined OID string to the contents of a
**              CS_OID structure.
**
** Parameters
**              oid -- Pointer to a CS_OID structure.OID->oid_length should be
**              the length of the string, not including any null-terminator.
**              oid_string -- Null-terminated OID string to compare.
**
** Returns
**              Non-zero if contents of oid->oid_buffer matches contents
**              of oid_string.
*/
int match_OID(oid, oid_string)

CS_OID *oid;
CS_CHAR *oid_string;
{
   return ((strncmp(oid_string, oid->oid_buffer, oid->oid_length) == 0)
   && ((oid->oid_length == strlen(oid_string))));
} /* match_OID() */
```

属性値の処理

次のコード例は、属性の値をテキストとして出力するサンプル・ルーチン attr_display_values を宣言します。attr_display_values は次の 2 つのユーティリティ・ルーチンを呼び出して、この作業を行います。

- attr_val_as_string － 属性値をテキストとしてフォーマットし、結果を文字配列に置きます。
- attr_enum_english_name － 整数または列挙型属性値を出力可能な文字列に変換します。

```
/*
** attr_display_values()
**   Writes an attribute's values to the specified text
**   file.
**
** Parameters
**   attr_metadata -- address of the CS_ATTRIBUTE structure that
**      contains metadata for the attribute.
**   attr_vals -- address of an array of CS_ATTRVALUE structures.
**       This function assumes length is attr_metadata->attr_numvals
**       and value syntax is attr_metadata->attr_syntax.
**   outfile -- Open FILE handle to write to.
**
** Returns
**   CS_SUCCEED or CS_FAIL.
*/
CS_RETCODE
attr_display_values(attr_metadata, attr_vals, outfile)
CS_ATTRIBUTE       *attr_metadata;
CS_ATTRVALUE       *attr_vals;
FILE               *outfile;
{
   CS_INT              i;
   CS_CHAR             outbuf[CS_MAX_DS_STRING * 3];
   CS_RETCODE          ret;

   /*
   ** Print each value.
   */
   for (i = 0; i < attr_metadata->attr_numvals; i++)
   {
      ret = attr_val_as_string(attr_metadata, attr_vals + i,
            outbuf, CS_MAX_DS_STRING * 3, NULL);
      if (ret != CS_SUCCEED)
      {
         ex_error("attr_display_values:attr_val_as_string() failed.");
         return CS_FAIL;
      }
      fprintf(outfile, "¥t%s¥n", outbuf);
   }
```

```
   return CS_SUCCEED;

} /* attr_display_values() */
/*
** attr_val_as_string() -- Convert the contents of a CS_ATTRVALUE union to
**   a printable string.
**
** Parameters
**   attr_metadata -- The CS_ATTRIBUTE structure containing metadata
**       for the attribute value.
**   val -- Pointer to the CS_ATTRVALUE union.
**   buffer -- Address of the buffer to receive the converted value.
**   buflen -- Length of *buffer in bytes.
**   outlen -- If supplied, will be set to the number of bytes written
**      to *buffer.
**
** Returns
**   CS_SUCCEED or CS_FAIL.
*/

CS_RETCODE
attr_val_as_string(attr_metadata, val, buffer, buflen, outlen)
CS_ATTRIBUTE       *attr_metadata;
CS_ATTRVALUE       *val;
CS_CHAR            *buffer;
CS_INT              buflen;
CS_INT             *outlen;
{
   CS_CHAR             outbuf[CS_MAX_DS_STRING * 4];
   CS_CHAR             scratch[CS_MAX_DS_STRING];
   CS_RETCODE          ret;

   if (buflen == 0 || buffer == NULL)
   {
      return CS_FAIL;
   }
   if (outlen != NULL)
   {
      *outlen = 0;
   }
   switch ((int)attr_metadata->attr_syntax)
   {
      case CS_ATTR_SYNTAX_STRING:
         sprintf(outbuf, "%.*s",
            (int)(val->value_string.str_length),
            val->value_string.str_buffer);
         break;
      case CS_ATTR_SYNTAX_BOOLEAN:
         sprintf(outbuf, "%s",
            val->value_boolean == CS_TRUE ?"True" :"False");
         break;
```

```
case CS_ATTR_SYNTAX_INTEGER:
case CS_ATTR_SYNTAX_ENUMERATION:
/*
** Some enumerated or integer attribute values should be converted
** into an english-language equivalent. attr_enum_english_name()
** contains all the logic to convert #define's into human
** language.
*/
   ret = attr_enum_english_name((CS_INT)(val->value_enumeration),
         &(attr_metadata->attr_type),
         scratch, CS_MAX_DS_STRING, NULL);
   if (ret != CS_SUCCEED)
   {
      ex_error("attr_val_as_string:attr_enum_english_name() failed.");
      return CS_FAIL;
   }
   sprintf(outbuf, "%s", scratch);
   break;

case CS_ATTR_SYNTAX_TRANADDR:
/*
** The access type is an enumerated value.Get an english language
** string for it.
*/
   switch ((int)(val->value_tranaddr.addr_accesstype))
   {
      case CS_ACCESS_CLIENT:
         sprintf(scratch, "client");
         break;
      case CS_ACCESS_ADMIN:
         sprintf(scratch, "administrative");
         break;
      case CS_ACCESS_MGMTAGENT:
         sprintf(scratch, "management agent");
         break;
      default:
         sprintf(scratch, "%ld",
                (long)(val->value_tranaddr.addr_accesstype));
         break;
   }

   sprintf(outbuf,
           "Access type '%s'; Transport type '%s'; Address '%s'",
            scratch,
            val->value_tranaddr.addr_trantype.str_buffer,
            val->value_tranaddr.addr_tranaddress.str_buffer);

   break;

case CS_ATTR_SYNTAX_OID:
   sprintf(outbuf, "%.*s",
   (int)(val->value_oid.oid_length),
```

```
            val->value_oid.oid_buffer);
            break;
         default:
            sprintf(outbuf, "Unknown attribute value syntax");
            break;

      } /* switch */
      if (strlen(outbuf) + 1 > buflen || buffer == NULL)
      {
         return CS_FAIL;
      }
      else
      {
         sprintf(buffer, "%s", outbuf);
         if (outlen != NULL)
         {
            *outlen = strlen(outbuf) + 1;
         }
      }
      return CS_SUCCEED;
} /* attr_val_as_string() */
/*
** attr_enum_english_name()
**   Based on the attribute type, associate an english phrase with
**   a CS_INT value.Use this function to get meaningful names for
**   CS_ATTR_SYNTAX_ENUMERATION or CS_ATTR_SYNTAX_INTEGER attribute
**   values.
**
**   If the attribute type represents a quantity and not a numeric code,
**   then the value is converted to the string representation of the
**   number.Unknown codes are handled the same way.
**
** Parameters
**   enum_val -- The integer value to convert to a string.
**   attr_type -- Pointer to an OID structure containing the OID string
**      that tells the attribute's type.
**   buffer -- Address of the buffer to receive the converted value.
**   buflen -- Length of *buffer in bytes.
**   outlen -- If supplied, will be set to the number of bytes written
**      to *buffer.
**
** Returns
**   CS_SUCCEED or CS_FAIL
*/

CS_RETCODE
attr_enum_english_name(enum_val, attr_type, buffer, buflen, outlen)
CS_INT              enum_val;
CS_OID             *attr_type;
CS_CHAR            *buffer;
CS_INT              buflen;
```

```
CS_INT              *outlen;
{
   CS_CHAR              outbuf[CS_MAX_DS_STRING];

   if (buffer == NULL || buflen <= 0)
   {
      return CS_FAIL;
   }
   if (outlen != NULL)
   {
      *outlen = 0;
   }

   /*
   ** Server version number.
   */
   if (match_OID(attr_type, CS_OID_ATTRVERSION))
   {
      sprintf(outbuf, "%ld", (long)enum_val);
   }
   /*
   ** Server's status.
   */
   else if (match_OID(attr_type, CS_OID_ATTRSTATUS))
   {
      switch ((int)enum_val)
      {
         case CS_STATUS_ACTIVE:
            sprintf(outbuf, "running");
            break;
         case CS_STATUS_STOPPED:
            sprintf(outbuf, "stopped");
            break;
         case CS_STATUS_FAILED:
            sprintf(outbuf, "failed");
            break;
         case CS_STATUS_UNKNOWN:
            sprintf(outbuf, "unknown");
            break;
         default:
            sprintf(outbuf, "%ld", (long)enum_val);
            break;
      }
   }

   /*
   ** Anything else is either an enumerated type that we don't know
   ** about, or it really is just a number.We print the numeric value.
   */
   else
   {
      sprintf(outbuf, "%ld", (long)enum_val);
```

```
   }

   /*
   ** Transfer output to the caller's buffer.
   */
   if (strlen(outbuf) + 1 > buflen || buffer == NULL)
   {
      return CS_FAIL;
   }
   else
   {
      sprintf(buffer, "%s", outbuf);
      if (outlen != NULL)
      {
         *outlen = strlen(outbuf) + 1;
      }
   }
   return CS_SUCCEED;
} /* attr_enum_english_name() */
```

手順 4：クリーンアップする

アプリケーションは、ct_ds_dropobj を呼び出して、ディレクトリ・コールバックから受信した各ディレクトリ・オブジェクトの割り付けを解除できます。

また、アプリケーションが ct_con_drop を呼び出して親接続を削除すると、ディレクトリ・オブジェクトは暗黙的に解除されます。

付 録　A

呼び出しの論理シーケンス

Client-Library は、「ステータス・マシン」を使用して、論理シーケンスを操作します。これは、アプリケーションが行った最後の呼び出しについての情報を保管して、有効な呼び出しだけがあとに続くように、呼び出しを制限します。たとえば、アプリケーションでは、ct_connect を呼び出してサーバに接続してから、ct_send を呼び出してコマンドを送信する必要があります。

Client-Library のステータス・マシン

Client-Library のアプリケーション・プログラミング・インタフェース (API) 層は、3 つのステータス・マシンから構成されます。各マシンは、3 つの基本制御構造体である CS_CONTEXT、CS_CONNECTION、CS_COMMAND に対応しています。基本制御構造体については、「隠し構造体」(29 ページ) を参照してください。

コンテキスト・レベルでは、アプリケーションは、1 つまたは複数のコンテキスト構造体の割り付け、コンテキストの CS-Library プロパティの設定、Client-Library の初期化、コンテキストの Client-Library プロパティの設定を行うことによって、環境を設定します。「手順 1：Client-Library プログラミング環境を設定する」(17 ページ) を参照してください。

接続レベルでは、アプリケーションは、1 つまたは複数の接続構造体の割り付け、接続のプロパティの設定、接続のオープン、接続のサーバ・オプションの設定を行うことによって、サーバに接続します。アプリケーションは、コンテキスト構造体が割り付けられて初めて、接続構造体を割り付けることができます。「手順 3：サーバに接続する」(21 ページ) を参照してください。

コマンド・レベルでは、アプリケーションは 1 つまたは複数のコマンド構造体の割り付け、コマンドの送信、結果の処理を行います。アプリケーションは、接続構造体が割り付けられて初めて、コマンド構造体を割り付けることができます。「手順 4：サーバにコマンドを送信する」(23 ページ) を参照してください。

呼び出しのコマンド・レベル・シーケンス

呼び出しの論理シーケンスは、コマンド・レベルで複雑になります。これは、コマンド・レベルで管理されるルーチンの数が増えるためです。

Client-Library のコマンド・ステータス・マシンは、特定のルーチンへの呼び出しが許可されるかどうかを確認するときに、2 つの別のステータス・テーブルから情報を取得します。これらのテーブルは、起動されたコマンド・ステータス・テーブルと、結果タイプ・ステータス・テーブルです。

コマンド・ステータス・テーブル

コマンド・テーブルは、アプリケーションの「ステータス」を定義します。たとえば、アプリケーションが最後に行った呼び出しが ct_send であることを示すように、コマンド送信ステータスを定義します。

コマンド・テーブルは、有効な Client-Library ルーチンへの各ステータスのマップも行います。アプリケーションは、そのステータスの間、それらのルーチンを呼び出すことができます。たとえば、Command Sent ステータスのアプリケーションは、コマンドまたは結果セットのキャンセル、コマンド構造体プロパティの取得と設定、動的 SQL 記述子領域に対する操作の実行、サーバからの TDS パケットの受信、処理結果の設定を行うことができます。

各コマンド・ステータスの詳細については、「コマンド・ステータス」(162 ページ) を参照してください。各コマンド・ステータスでの有効な呼び出しのリストは、「各コマンド・ステータスで呼び出し可能なルーチン」(164 ページ) を参照してください。

起動されたコマンド・ステータス・テーブル

起動されたコマンド・テーブルは、サーバに送信されるコマンドの開始と設定を行うルーチン (ct_command、ct_cursor、ct_dynamic、ct_param など) の使用を制御します。このテーブルは、コマンド・テーブルよりも高度なレベルの機能を提供します。

たとえば、コマンド・ステータス・マシンは、コマンドが起動した後にのみ ct_param が確実に呼び出されるようにします。しかし、起動されたコマンドがパラメータを使用しない場合 (ct_cursor(CS_CURSOR_CLOSE) の場合など) に、アプリケーションが ct_param を呼び出すのを防ぐことはできません。起動されたコマンド・テーブルは、このような場合に、論理シーケンスの呼び出しを実施します。

別の例としては、Client-Library カーソルが *cmd1* CS_COMMAND 構造体を使用して宣言された場合を想定します。カーソル宣言コマンドがサーバに送信されて、その結果が処理されると、ステータス・マシンは Idle ステータスになります。

この Idle ステータスから、コマンド・ステータス・マシンは、アプリケーションが新しいコマンドを起動できるようにします。これは、アプリケーションが、最初のカーソル (*cmd1*) に使用したものと同じ CS_COMMAND 構造体を使用して、2 番目のカーソルを宣言するのを防ぐことはできません。

しかし、起動されたコマンド・テーブルは、コマンド・ハンドル上のカーソルのステータスを追跡し続けます。このテーブルは、カーソルが特定の CS_COMMAND 構造体を使用して以前に宣言されている場合、同じ CS_COMMAND 構造体を使用したカーソルの宣言が再度行われると、それを無効と認識します。

各起動されたコマンド・ステータスの詳細については、「起動されたコマンド」(174 ページ) を参照してください。起動されたコマンド・ステータスと Client-Library ルーチンのマッピングについては、「起動されたコマンドで呼び出し可能なルーチン」(176 ページ) を参照してください。

結果タイプ・ステータス・テーブル

結果タイプ・テーブルは、結果セットに関する情報を返すルーチンを中心に扱います。コマンド・ステータス・マシンは、結果がいつ使用できるかを示すステータス (Fetchable Results や Fetchable Cursor Results など) を定義します。結果タイプ・テーブルは、使用できる結果のタイプを示します。

ルーチンによっては、特定の結果タイプにのみ有効なものがあるため、この情報は重要です。たとえば、ct_compute_info の呼び出しは、計算結果が使用できる場合にのみ成り立ちます。また、ct_br_column の呼び出しは、正規ロー結果が使用できる場合にのみ成り立ちます。このような場合、結果タイプ・テーブルは、論理シーケンスの呼び出しを実施します。

各結果タイプ・ステータスの詳細については、「結果タイプ」(178 ページ) を参照してください。結果タイプ・ステータスと Client-Library ルーチンのマッピングについては、「各結果タイプで呼び出し可能なルーチン」(180 ページ) を参照してください。

まとめ

この後の情報は、有効な Client-Library アプリケーションの動作に関するリファレンスです。特定シーケンスのルーチン呼び出しが有効かどうかを確認する場合、またはこの後必要な作業を知る必要がある場合に使用してください。

注意　Client-Library は、アプリケーションが論理シーケンスでルーチンを呼び出さなかった場合、その原因を説明するエラー・メッセージを実行時に返します。

コマンド・ステータス

Client-Library は、コマンドの現在のステータスを追跡します。コマンドは、次のいずれかのステータスにあります。

表 *A-1:* コマンド・ステータス

コマンド・ステータス	意味
Idle	アプリケーションのステータス： • コマンドを起動していない。 • 前回のコマンド結果を完全に処理した。 • すべてのカーソル・ローをフェッチしたが、Client-Library カーソルをクローズしていない。 • 未処理の結果に関連している Client-Library カーソルをクローズした。
Command initiated	アプリケーションが ct_command、ct_cursor、または ct_dynamic を呼び出してコマンドを起動したが、それをサーバに送信していない。
Command sent	アプリケーションが ct_send を呼び出してサーバにコマンドを送信したが、ct_results を呼び出して処理用の結果データを設定していない。
Non-fetchable results available	アプリケーションが ct_results を呼び出したが、結果セットに実際の結果データが入っていない。ct_results を呼び出す必要がある。 または アプリケーションが ct_fetch を呼び出し、それが CS_END_DATA を返した。
ANSI-style cursor end-data	アプリケーションが ct_fetch を呼び出し、それが CS_END_DATA を返した。また、CS_ANSI_BINDS プロパティが設定されている。
Fetchable results	アプリケーションが ct_results を呼び出し、結果セットにフェッチ可能な非カーソル結果 (計算結果、リターン・パラメータ結果、通常ロー結果、ストアド・プロシージャのリターン・ステータス結果) が入っている。ct_fetch が呼び出されていない。
Fetchable cursor results	アプリケーションが ct_results を呼び出し、結果セットにフェッチ可能なカーソル結果が入っている。ct_fetch が呼び出されていない。
Fetchable nested command	アプリケーションが、フェッチ可能なカーソル結果が入っている結果セットからフェッチする前に、カーソル・クローズ・コマンド (ct_cursor(CS_CURSOR_CLOSE)) を起動した。
Sent fetchable nested command	アプリケーションが、フェッチ可能なカーソル結果が入っている結果セットからフェッチする前に、ct_send を呼び出してカーソル・クローズ・コマンドをサーバに送信した。
Processing fetchable nested command	アプリケーションが、フェッチ可能なカーソル結果が入っている結果セットからフェッチする前に、ct_results を呼び出して、カーソル・クローズ・コマンドの結果を処理した。
Fetching results	アプリケーションが、ct_fetch を少なくとも 1 回呼び出して、フェッチ結果 (計算結果、リターン・パラメータ結果、通常ロー結果、ストアド・プロシージャのリターン・ステータス結果) を現在処理している。
Fetching cursor results	アプリケーションが、ct_fetch を少なくとも 1 回呼び出して、フェッチしたカーソル・ロー結果を現在処理している。

コマンド・ステータス	意味
Fetching nested command	アプリケーションが、カーソル結果の入っている結果セットからフェッチしている間、次のコマンドのどれかを起動した。 • カーソル・クローズ (ct_cursor(CS_CURSOR_CLOSE)) • カーソル更新 (ct_cursor(CS_CURSOR_UPDATE)) • カーソル削除 (ct_cursor(CS_CURSOR_DELETE))
Sent fetching nested command	アプリケーションが、カーソル結果の入っている結果セットからフェッチしている間、ct_send を呼び出して、カーソル・クローズ、カーソル更新、またはカーソル削除のいずれかのコマンドをサーバに送信した。
Processing fetching nested command	アプリケーションが、カーソル結果の入っている結果セットからフェッチしている間、ct_results を呼び出して、カーソル・クローズ、カーソル更新、またはカーソル削除のいずれかのコマンドの結果を処理した。
Result set canceled	アプリケーションが、現在のコマンドをキャンセルした (ct_cancel(CS_CANCEL_ALL))。アプリケーションは、もう一度 ct_results を呼び出して、コマンドを Idle ステータスに戻すことができる。
Undefined	コマンド構造体が定義されていないステータスにある。ct_cancel(CS_CANCEL_ALL) を呼び出す。
In receive passthrough	アプリケーションが ct_recvpassthru を呼び出して、CS_PASSTHRU_MORE が返された。
In send passthrough	アプリケーションが ct_sendpassthru を呼び出して、CS_PASSTHRU_MORE が返された。

コマンド・レベル・ルーチン

次の Client-Library ルーチンは、コマンド・レベルで管理されます。

ct_bind	ct_data_info	ct_param
ct_br_column	ct_describe	ct_recvpassthru
ct_br_table	ct_dynamic	ct_res_info
ct_cancel	ct_dyndesc	ct_results
ct_cmd_drop	ct_dynsqlda	ct_send
ct_cmd_props	ct_fetch	ct_send_data
ct_command	ct_get_data	ct_sendpassthru
ct_compute_info	ct_getformat	ct_setparam
ct_cursor	ct_keydata	

各コマンド・ステータスで呼び出し可能なルーチン

表 A-2 は、各コマンド・ステータスと、ステータス中にアプリケーションが呼び出すことができる Client-Library ルーチンのマッピングを示しています。この表は、ルーチン終了後のコマンドのステータスも示しています。

表 *A-2:* 各コマンド・ステータスで呼び出し可能なルーチン

起動時のステータス	呼び出し可能なルーチン	終了後のコマンド・ステータス
Idle	ct_cancel(CS_CANCEL_ALL) ct_cancel(CS_CANCEL_ATTN)	• CS_SUCCEED の場合、Idle • CS_FAIL の場合、Undefined
	ct_cmd_drop	Idle
	ct_cmd_props	Idle
Idle	ct_command	• CS_SUCCEED の場合、Command initiated • CS_FAIL の場合、Idle
	ct_cursor	• CS_SUCCEED の場合、Command initiated • CS_FAIL の場合、Idle
	ct_dynamic	• CS_SUCCEED の場合、Command initiated • CS_FAIL の場合、Idle
	ct_dyndesc	Idle
	ct_dynsqlda	Idle
	ct_sendpassthru	• CS_PASSTHRU_MORE の場合、In send passthrough • CS_PASSTHRU_EOM の場合、Command sent • CS_FAIL の場合、Undefined
Command initiated	ct_cancel(CS_CANCEL_ALL)	• CS_SUCCEED の場合、Idle • CS_FAIL の場合、Command initiated
	ct_cancel(CS_CANCEL_ATTN)	Command initiated
	ct_cmd_props	Command initiated
	ct_cursor	Command initiated
	ct_data_info(CS_SET)	Command initiated
	ct_dyndesc	Command initiated
	ct_dynsqlda	Command initiated
	ct_param	Command initiated
	ct_setparam	Command initiated
	ct_send	• CS_SUCCEED の場合、Command sent • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
	ct_send_data	• CS_SUCCEED の場合、Command initiated • CS_FAIL の場合、Undefined

起動時のステータス	呼び出し可能なルーチン	終了後のコマンド・ステータス
Command sent	ct_cancel(CS_CANCEL_ALL)	• CS_SUCCEED の場合、Result set canceled • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_ATTN)	• CS_SUCCEED の場合、Command sent • CS_FAIL の場合、Undefined
	ct_cmd_props	Command sent.
	ct_dynsqlda	Command sent.
	ct_dyndesc	Command sent.
	ct_recvpassthru	• CS_PASSTHRU_MORE の場合、In receive passthrough • CS_PASSTHRU_EOM、CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
	ct_results	• CS_SUCCEED と **result_type* が CS_MSG_RESULT、CS_CMD_SUCCEED、CS_CMD_FAIL、CS_CMD_DONE、CS_ROWFMT_RESULT、CS_COMPUTEFMT_RESULT、または CS_DESCRIBE_RESULT の場合、Non-fetchable results available • CS_SUCCEED と **result_type* が CS_ROW_RESULT、CS_COMPUTE_RESULT、CS_PARAM_RESULT、または CS_STATUS_RESULT の場合、Fetchable results • CS_SUCCEED と **result_type* が CS_CURSOR_RESULT の場合、Fetchable cursor results • CS_CANCELED または CS_END_RESULTS の場合、Idle • CS_SUCCEED と **result_type* が CS_CMD_FAIL の場合、Undefined
Non-fetchable results available	ct_br_column	Non-fetchable results available.
	ct_br_table	Non-fetchable results available.
	ct_cancel(CS_CANCEL_ALL)	• CS_SUCCEED の場合、Result set canceled • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_ATTN)	• CS_SUCCEED の場合、Non-fetchable results available • CS_FAIL の場合、Undefined

起動時のステータス	呼び出し可能なルーチン	終了後のコマンド・ステータス
Non-fetchable results available	ct_cancel(CS_CANCEL_CURRENT)	• CS_SUCCEED の場合、Non-fetchable results available • CS_FAIL の場合、Undefined
	ct_cmd_props	Non-fetchable results available.
	ct_compute_info	Non-fetchable results available.
	ct_describe	Non-fetchable results available.
	ct_dyndesc	Non-fetchable results available.
	ct_dynsqlda	Non-fetchable results available.
	ct_getformat	Non-fetchable results available.
	ct_res_info	Non-fetchable results available.
ANSI-style cursor end-data	ct_results	• CS_SUCCEED と **result_type* が CS_ROW_RESULT、CS_COMPUTE_RESULT、CS_PARAM_RESULT、または CS_STATUS_RESULT の場合、Fetchable results • CS_SUCCEED と **result_type* が CS_CURSOR_RESULT の場合、Fetchable cursor results • CS_CANCELED または CS_END_RESULTS の場合、Idle • CS_FAIL の場合、Undefined
	ct_bind	ANSI-style cursor end-data
	ct_br_column	ANSI-style cursor end-data
	ct_br_table	ANSI-style cursor end-data
	ct_cancel(CS_CANCEL_ALL)	• CS_SUCCEED の場合、Result set canceled • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_ATTN)	• CS_SUCCEED の場合、ANSI-style cursor end-data • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_CURRENT)	ANSI-style cursor end-data
	ct_cmd_props	ANSI-style cursor end-data
	ct_compute_info	ANSI-style cursor end-data
	ct_describe	ANSI-style cursor end-data
	ct_dyndesc	ANSI-style cursor end-data
	ct_dynsqlda	ANSI-style cursor end-data
	ct_fetch	• CS_END_DATA の場合、ANSI-style cursor end-data • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined

起動時のステータス	呼び出し可能なルーチン	終了後のコマンド・ステータス
ANSI-style cursor end-data	ct_getformat	ANSI-style cursor end-data
	ct_res_info	ANSI-style cursor end-data
	ct_results	• CS_SUCCEED と **result_type* が CS_MSG_RESULT または CS_CMD_DONE の場合、Non-fetchable results available • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
Fetchable results	ct_bind	Fetchable results
	ct_br_column	Fetchable results
	ct_br_table	Fetchable results
	ct_cancel(CS_CANCEL_ALL)	• CS_SUCCEED の場合、Result set canceled • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_ATTN)	• CS_SUCCEED の場合、Fetchable results • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_CURRENT)	• CS_SUCCEED の場合、Non-fetchable results available • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
	ct_cmd_props	Fetchable results
	ct_compute_info	Fetchable results
	ct_describe	Fetchable results
	ct_dyndesc	Fetchable results
	ct_dynsqlda	Fetchable results
	ct_fetch	• CS_SUCCEED または CS_ROW_FAIL の場合、Fetching results • CS_END_DATA の場合、Non-fetchable results available • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
	ct_getformat	Fetchable results
	ct_res_info	Fetchable results

起動時のステータス	呼び出し可能なルーチン	終了後のコマンド・ステータス
Fetchable cursor results	ct_bind	Fetchable cursor results
	ct_cancel(CS_CANCEL_ALL)	• CS_SUCCEED の場合、Result set canceled • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_ATTN)	• CS_SUCCEED の場合、Fetchable cursor results • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_CURRENT)	• CS_SUCCEED の場合、Non-fetchable results available • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
	ct_cmd_props	Fetchable cursor results
	ct_cursor	• CS_SUCCEED の場合、Fetchable nested command • CS_FAIL の場合、Fetchable cursor results
	ct_describe	Fetchable cursor results
	ct_dyndesc	Fetchable cursor results
	ct_dynsqlda	Fetchable cursor results
	ct_fetch	• CS_SUCCEED または CS_ROW_FAIL の場合、Fetching cursor results • CS_CANCELED の場合、Idle • CS_END_DATA の場合、Non-fetchable results available • CS_END_DATA と CS_ANSI_BINDS プロパティが設定されている場合、ANSI-style cursor end-data • CS_FAIL の場合、Undefined
	ct_getformat	Fetchable cursor results
	ct_res_info	Fetchable cursor results
Fetchable nested command	ct_cancel(CS_CANCEL_ALL)	• CS_SUCCEED の場合、Fetchable cursor results • CS_FAIL の場合、Fetchable nested command
	ct_cancel(CS_CANCEL_ATTN)	• CS_SUCCEED の場合、Fetchable nested command • CS_FAIL の場合、Undefined
	ct_cmd_props	Fetchable nested command
	ct_dyndesc	Fetchable nested command
	ct_dynsqlda	Fetchable nested command
	ct_param	Fetchable nested command
	ct_setparam	Fetchable nested command
	ct_send	• CS_SUCCEED の場合、Sent fetchable nested • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined

起動時のステータス	呼び出し可能なルーチン	終了後のコマンド・ステータス
Sent fetchable nested	ct_cancel(CS_CANCEL_ALL)	• CS_SUCCEED の場合、Result set canceled • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_ATTN)	• CS_SUCCEED の場合、Sent fetchable nested • CS_FAIL の場合、Undefined
	ct_cmd_props	Sent fetchable nested
	ct_results	• CS_CMD_SUCCEED または CS_CMD_FAIL の場合、Processing fetchable nested command • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
Processing fetchable nested command	ct_cancel(CS_CANCEL_ALL)	• CS_SUCCEED の場合、Result set canceled • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_ATTN)	• CS_SUCCEED の場合、Processing fetchable nested command • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_CURRENT)	• CS_SUCCEED の場合、Processing fetchable nested command • CS_FAIL の場合、Undefined
	ct_cmd_props	Processing fetchable nested command
	ct_dyndesc	Processing fetchable nested command
	ct_dynsqlda	Processing fetchable nested command
	ct_res_info	Processing fetchable nested command
	ct_results	• CS_END_RESULTS の場合、Fetchable cursor results • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined

起動時のステータス	呼び出し可能なルーチン	終了後のコマンド・ステータス
Fetching results	ct_bind	Fetching results
	ct_br_column	Fetching results
	ct_br_table	Fetching results
	ct_cancel(CS_CANCEL_ALL)	• CS_SUCCEED の場合、Result set canceled • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_ATTN)	• CS_SUCCEED の場合、Fetching results • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_CURRENT)	• CS_SUCCEED の場合、Non-fetchable results available • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
	ct_cmd_props	Fetching results
	ct_compute_info	Fetching results
	ct_data_info(CS_GET)	Fetching results
	ct_describe	Fetching results
	ct_dyndesc	• CS_SUCCEED の場合、Fetching results • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
Fetching results	ct_dynsqlda	• CS_SUCCEED の場合、Fetching results • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
	ct_fetch	• CS_SUCCEED の場合、Fetching results • CS_END_DATA の場合、Non-fetchable results available • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
	ct_get_data	• CS_SUCCEED の場合、Fetching results • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
	ct_getformat	Fetching results
	ct_res_info	Fetching results

起動時のステータス	呼び出し可能なルーチン	終了後のコマンド・ステータス
Fetching cursor results	ct_bind	Fetching cursor results
	ct_cancel(CS_CANCEL_ALL)	• CS_SUCCEED の場合、Result set canceled • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_ATTN)	• CS_SUCCEED の場合、Fetching cursor results • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_CURRENT)	• CS_SUCCEED の場合、Non-fetchable results available • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
	ct_cmd_props	Fetching cursor results
	ct_cursor	• CS_SUCCEED の場合、Fetching nested command • CS_FAIL の場合、Fetching cursor results
	ct_describe	Fetching cursor results
	ct_dyndesc	• CS_SUCCEED の場合、Fetching cursor results • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
	ct_dynsqlda	• CS_SUCCEED の場合、Fetching cursor results • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
	ct_fetch	• CS_SUCCEED の場合、Fetching cursor results • CS_END_DATA の場合、Non-fetchable results available • CS_END_DATA と CS_ANSI_BINDS プロパティが設定されている場合、ANSI-style cursor end-data • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
	ct_get_data	• CS_SUCCEED の場合、Fetching cursor results • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
	ct_getformat	Fetching cursor results
	ct_keydata	Fetching cursor results
	ct_res_info	Fetching cursor results

起動時のステータス	呼び出し可能なルーチン	終了後のコマンド・ステータス
Fetching nested command	ct_cancel(CS_CANCEL_ALL)	• CS_SUCCEED の場合、Fetching cursor results • CS_FAIL の場合、Fetching nested command
	ct_cancel(CS_CANCEL_ATTN)	• CS_SUCCEED の場合、Fetching nested command • CS_FAIL の場合、Undefined
	ct_cmd_props	Fetching nested command
	ct_dyndesc	Fetching nested command
	ct_dynsqlda	Fetching nested command
	ct_param	Fetching nested command
	ct_setparam	Fetching nested command
	ct_send	• CS_SUCCEED の場合、Sent fetching nested command • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
Sent fetching nested command	ct_cancel(CS_CANCEL_ALL)	• CS_SUCCEED の場合、Result set canceled • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_ATTN)	• CS_SUCCEED の場合、Sent fetching nested command • CS_FAIL の場合、Undefined
	ct_cmd_props	Sent fetching nested command
	ct_results	• CS_CMD_SUCCEED または CS_CMD_FAIL の場合、Processing fetching nested command • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined

起動時のステータス	呼び出し可能なルーチン	終了後のコマンド・ステータス
Processing fetching nested command	ct_cancel(CS_CANCEL_ALL)	• CS_SUCCEED の場合、 Result set canceled • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_ATTN)	• CS_SUCCEED の場合、Processing fetching nested command • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_CURRENT)	• CS_SUCCEED の場合、Processing fetching nested command • CS_FAIL の場合、Undefined
	ct_cmd_props	Processing fetching nested command
	ct_dyndesc	Processing fetching nested command
	ct_dynsqlda	Processing fetching nested command
	ct_keydata	Processing fetching nested command
	ct_res_info	Processing fetching nested command
	ct_results	• CS_SUCCEED の場合、Processing fetching nested command • CS_END_RESULTS の場合、Fetching cursor results • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
Result set canceled	ct_cancel(CS_CANCEL_ALL)	• CS_SUCCEED の場合、Idle • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_ATTN)	• CS_SUCCEED の場合、Idle • CS_FAIL の場合、Undefined
	ct_cmd_drop	Idle
	ct_cmd_props	Idle
	ct_command	• CS_SUCCEED の場合、Command initiated • CS_FAIL の場合、Idle
	ct_cursor	• CS_SUCCEED の場合、Command initiated • CS_FAIL の場合、Idle
	ct_dynamic	• CS_SUCCEED の場合、Command initiated • CS_FAIL の場合、Idle
	ct_dyndesc	• CS_SUCCEED、CS_ROW_FAIL、または CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
	ct_dynsqlda	• CS_SUCCEED、CS_ROW_FAIL、または CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
	ct_results	• CS_SUCCEED または CS_FAIL の場合、 Result set canceled • CS_CANCELED の場合、Idle
	ct_sendpassthru	Result set canceled

起動時のステータス	呼び出し可能なルーチン	終了後のコマンド・ステータス
Undefined	ct_cancel(CS_CANCEL_ALL)	• CS_SUCCEED の場合、Idle • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_ATTN)	Undefined
	ct_cmd_props	Undefined
	ct_dyndesc	Undefined
	ct_dynsqlda	Undefined
In receive passthrough	ct_cancel(CS_CANCEL_ALL)	• CS_SUCCEED の場合、Idle • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_ATTN)	• CS_SUCCEED の場合、In receive passthrough • CS_FAIL の場合、Undefined
	ct_cmd_props	In receive passthrough
	ct_recvpassthru	• CS_PASSTHRU_EOM または CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined
In send passthrough	ct_cancel(CS_CANCEL_ALL)	• CS_SUCCEED の場合、Idle • CS_FAIL の場合、Undefined
	ct_cancel(CS_CANCEL_ATTN)	• CS_SUCCEED の場合、In send passthrough • CS_FAIL の場合、Undefined
	ct_cmd_props	In send passthrough
	ct_sendpassthru	• CS_PASSTHRU_EOM の場合、Command sent • CS_CANCELED の場合、Idle • CS_FAIL の場合、Undefined

起動されたコマンド

Client-Library は、コマンド・ステータスだけでなく、起動されたコマンドも追跡します。起動されたコマンドは、次のいずれかのステータスにあります。

表 *A-3:* 起動されたコマンドのステータス

起動されたコマンドのステータス	意味
Idle	アプリケーションが、コマンドを起動していないか、または前回のコマンド結果を完全に処理した。
Idle, with declared cursor	アプリケーションがカーソル宣言コマンド (ct_cursor(CS_CURSOR_DECLARE)) を起動して、コマンドをサーバに送信し、結果を完全に処理した。
Idle, with opened cursor	アプリケーションがカーソル・オープン・コマンド (ct_cursor(CS_CURSOR_OPEN)) を起動して、そのコマンドを送信し、結果をすべてフェッチした (ct_results が CS_END_RESULTS を返した) が、カーソルをクローズしていない。
Opened cursor, no rows fetched	アプリケーションが ct_results を呼び出したが、結果を何も処理していない。

起動されたコマンドのステータス	意味
Opened cursor, fetching rows	アプリケーションが ct_fetch を少なくとも 1 回呼び出して、現在フェッチした結果を処理している。
ct_command command initiated	アプリケーションが、ct_command を使用して、言語、メッセージ、パッケージ、または RPC コマンドを起動した。
Initiated send-data	アプリケーションが、ct_command を使用して、データ送信、またはバルク・データ送信コマンドを起動した。
Initiated cursor-declare	アプリケーションが、カーソル宣言コマンド (ct_cursor(CS_CURSOR_DECLARE)) を起動したが、ct_send を使用して、それをサーバに送信していない。
Initiated cursor-rows	アプリケーションが、ct_cursor(CS_CURSOR_ROWS) を使用して、カーソル・ロー・コマンドを起動した。
Initiated cursor-open	アプリケーションが、カーソル・オープン・コマンド (ct_cursor(CS_CURSOR_OPEN)) を起動したが、それをサーバに送信していない。
Initiated cursor-close	アプリケーションが、カーソル・クローズ・コマンド (ct_cursor(CS_CURSOR_CLOSE)) を起動したが、それをサーバに送信していない。
Initiated cursor-deallocate	アプリケーションが、カーソル割り付け解除コマンド (ct_cursor(CS_CURSOR_DEALLOC)) を起動したが、それをサーバに送信していない。
Initiated cursor-update	カーソル更新コマンド (ct_cursor(CS_CURSOR_UPDATE)) を起動したが、それをサーバに送信していない。
Initiated cursor-delete row	アプリケーションが、カーソル削除コマンド (ct_cursor(CS_CURSOR_DELETE)) を起動したが、それをサーバに送信していない。
Initiated dynamic cursor-declare	アプリケーションが、準備された動的 SQL 文に対してカーソル宣言コマンド (ct_dynamic(CS_CURSOR_DECLARE)) を起動したが、それをサーバに送信していない。
Initiated dynamic deallocate	アプリケーションが、準備された SQL 文の割り付けを解除するコマンド ct_dynamic(CS_DEALLOC)) を起動したが、それをサーバに送信していない。
Initiated dynamic describe	アプリケーションが、入力パラメータ情報を取得するコマンド (ct_dynamic(CS_DESCRIBE_INPUT))、またはカラム・リスト情報を取得するコマンド (ct_dynamic(CS_DESCRIBE_OUTPUT)) を起動したが、それをサーバに送信していない。
Initiated dynamic execute	アプリケーションが、準備された SQL 文を実行するコマンド (ct_dynamic(CS_EXECUTE)) を起動したが、それをサーバに送信していない。
Initiated dynamic execute immediate	アプリケーションが、準備された SQL 文を即時実行するコマンド (ct_dynamic(CS_EXEC_IMMEDIATE)) を起動したが、それをサーバに送信していない。
Initiated dynamic prepare	アプリケーションが、SQL 文を準備するコマンド (ct_dynamic(CS_PREPARE)) を起動したが、それをサーバに送信していない。
ct_send_data succeeded	アプリケーションが ct_send_data を少なくとも 1 回正常に呼び出した。
Initiated send-bulk command	アプリケーションが、バルク・データ送信コマンド (ct_command(CS_SEND_BULK_CMD)) を起動したが、それをサーバに送信していない。

起動されたコマンド・ルーチン

次の Client-Library ルーチンは、起動されたコマンドの処理に役立ちます。

ct_cmd_drop	ct_dynamic	ct_send_data
ct_command	ct_dyndesc	ct_setparam
ct_cursor	ct_dynsqlda	ct_sendpassthru
ct_data_info	ct_param	

起動されたコマンドで呼び出し可能なルーチン

表 A-4 は、起動されたコマンドの各ステータスと、そのステータスのときにアプリケーションが呼び出すことができる Client-Library ルーチンのマッピングを示しています。

ここで、「なし」と記載されている場合、アプリケーションは、「起動されたコマンド・ルーチン」(176 ページ) のリストに示されたルーチンを呼び出すことはできません。「呼び出し可能なルーチン」欄が「なし」になっているステータスからアプリケーションが行えるのは、起動されたコマンドを送信することか (ct_send)、キャンセルすることです (ct_cancel)。

表 *A-4:* 起動されたコマンドの各ステータスに対して呼び出し可能なルーチン

起動されたコマンド	呼び出し可能なルーチン
Idle	ct_cmd_drop ct_command(CS_LANG_CMD) ct_command(CS_MSG_CMD) ct_command(CS_PACKAGE_CMD) ct_command(CS_RPC_CMD) ct_command(CS_SEND_BULK_CMD) ct_command(CS_SEND_DATA_CMD) ct_command(CS_SEND_DATA_NOCMD) ct_cursor(CS_CURSOR_DECLARE) ct_dynamic(CS_CURSOR_DECLARE) ct_dynamic(CS_DEALLOC) ct_dynamic(CS_DESCRIBE_INPUT) ct_dynamic(CS_DESCRIBE_OUTPUT) ct_dynamic(CS_EXECUTE) ct_dynamic(CS_EXEC_IMMEDIATE) ct_dynamic(CS_PREPARE) ct_sendpassthru
Idle, with declared cursor	ct_cursor(CS_CURSOR_ROWS) ct_cursor(CS_CURSOR_OPEN) ct_cursor(CS_CURSOR_CLOSE, CS_DEALLOC) ct_cursor(CS_CURSOR_DEALLOC) ct_dynamic(CS_DEALLOC)
Idle, with opened cursor	ct_cursor(CS_CURSOR_CLOSE) ct_cursor(CS_CURSOR_CLOSE, CS_DEALLOC) ct_dynamic(CS_DEALLOC)
Opened cursor, no rows fetched	ct_cursor(CS_CURSOR_CLOSE) ct_cursor(CS_CURSOR_CLOSE, CS_DEALLOC)
Opened cursor, fetching rows	ct_cursor(CS_CURSOR_CLOSE) ct_cursor(CS_CURSOR_CLOSE, CS_DEALLOC) ct_cursor(CS_CURSOR_UPDATE) ct_cursor(CS_CURSOR_DELETE)
ct_command command initiated	ct_dyndesc(CS_USE_DESC) ct_dynsqlda(CS_SQLDA_PARAM) ct_param ct_setparam
Initiated send-data	ct_data_info(CS_SET) ct_send_data
Initiated cursor-declare	ct_cursor(CS_CURSOR_ROWS) ct_cursor(CS_CURSOR_OPEN) ct_cursor(CS_CURSOR_OPTION) ct_dyndesc(CS_USE_DESC) ct_dynsqlda(CS_SQLDA_PARAM) ct_param ct_setparam
Initiated cursor-rows	ct_cursor(CS_CURSOR_OPEN)

起動されたコマンド	呼び出し可能なルーチン
Initiated cursor-open	ct_dyndesc(CS_USE_DESC) ct_dynsqlda(CS_SQLDA_PARAM) ct_param ct_setparam
Initiated cursor-close	なし
Initiated cursor-deallocate	なし
Initiated cursor-update	ct_dyndesc(CS_USE_DESC) ct_dynsqlda(CS_SQLDA_PARAM) ct_param ct_setparam
Initiated cursor-delete	なし
Initiated dynamic cursor-declare	なし
Initiated dynamic deallocate	なし
Initiated dynamic describe	なし
Initiated dynamic execute	ct_dyndesc(CS_USE_DESC) ct_dynsqlda(CS_SQLDA_PARAM) ct_param ct_setparam
Initiated dynamic execute immediate	なし
Initiated dynamic prepare	ct_dyndesc(CS_USE_DESC) ct_dynsqlda(CS_SQLDA_PARAM) ct_param ct_setparam
ct_send_data succeeded	ct_send_data
Initiated send-bulk command	ct_send_data

結果タイプ

Client-Library は、コマンドが次のいずれかのステータスにある場合、結果タイプに基いて、呼び出し可能なルーチンを制限します。

- Results available
- Fetchable results
- Fetchable cursor results
- Fetchable nested command
- Sent fetchable nested command

- Processing fetchable nested command
- Fetching results
- Fetching cursor results
- Fetching nested command
- Sent fetching nested command
- Processing fetching nested command

表 A-5 は、各結果タイプを簡単に説明します。

表 *A-5:* 結果タイプの定義

結果タイプ	意味
Regular row results	Transact-SQL select 文の実行によって生成された表形式データのゼロ個以上のロー。
Cursor row results	アプリケーションが Client-Library cursor-open コマンドを実行したときに生成された表形式データのゼロ個以上のロー。
Parameter results	メッセージ・パラメータまたはストアド・プロシージャのリターン・パラメータの単一ロー。
Stored procedure return status results	1 つの値 (リターン・ステータス) を含んでいる単一ロー。
Message results	使用できるデータがないが、アプリケーションは ct_res_info を呼び出して、メッセージ ID を取得できる。
Compute row results	計算ローを生成した compute 句にリストされたカラム数と同じカラム数を持つ表データの単一ロー。
CS_CMD_DONE	コマンドの結果が完全に処理された。
CS_CMD_SUCCEED	データを返さないコマンド (Transact-SQL insert 文を含んでいる言語コマンド) が正常に終了した。
CS_CMD_FAIL	コマンド実行中にサーバがエラーを検出した。
Regular row format results	対応した通常ロー結果セットのフォーマット情報。
Compute row format results	対応した計算ロー結果セットのフォーマット情報。
Describe results	動的 SQL の記述入力または出力コマンドの結果として記述情報が返された。
Extended error data results	拡張されたエラー・データの単一ロー。
Notification results	レジスタード・プロシージャが呼び出された引数の単一ロー。

各結果タイプの詳細については、「第 6 章 結果処理コードの書き方」を参照してください。

結果タイプ処理ルーチン

次の Client-Library ルーチンは、各結果タイプの処理に役立ちます。

ct_bind	ct_data_info	ct_getformat
ct_br_column	ct_describe	ct_keydata
ct_br_table	ct_dyndesc	ct_res_info
ct_compute_info	ct_dynsqlda	

各結果タイプで呼び出し可能なルーチン

アプリケーションが ct_results を呼び出して、使用できる結果タイプを検索する場合、Client-Library は、ct_results **result_type* パラメータの値に基いて、呼び出しできるルーチンを定義します。

表 A-6 は、各結果タイプと、その結果タイプを処理するためにアプリケーションが呼び出すことができる有効な Client-Library ルーチンのマッピングを示します。

表 *A-6:* 各結果タイプで呼び出し可能なルーチン

結果タイプ	呼び出し可能なルーチン
Regular row results	ct_bind ct_br_column ct_br_table ct_data_info(CS_GET) ct_describe ct_getformat ct_res_info(CS_BROWSE_INFO) ct_res_info(CS_CMD_NUMBER) ct_res_info(CS_NUMDATA) ct_res_info(CS_NUMORDERCOLS) ct_res_info(CS_ORDERBY_COLS) ct_res_info(CS_TRANS_STATE) ct_dyndesc(CS_USE_DESC) ct_dynsqlda(CS_SQLDA_BIND)
Cursor row results	ct_bind ct_describe ct_getformat ct_keydata ct_res_info(CS_CMD_NUMBER) ct_res_info(CS_CMD_NUMDATA) ct_res_info(CS_TRANS_STATE) ct_dyndesc(CS_USE_DESC) ct_dynsqlda(CS_SQLDA_BIND)
Parameter results	ct_bind ct_describe ct_res_info(CS_CMD_NUMBER) ct_res_info(CS_NUMDATA) ct_res_info(CS_TRANS_STATE) ct_dyndesc(CS_USE_DESC) ct_dynsqlda(CS_SQLDA_BIND)
Stored procedure return status results	ct_bind ct_describe ct_res_info(CS_CMD_NUMBER) ct_res_info(CS_CMD_NUMDATA) ct_res_info(CS_TRANS_STATE) ct_dyndesc(CS_USE_DESC) ct_dynsqlda(CS_SQLDA_BIND)
Message results	ct_res_info(CS_CMD_NUMBER) ct_res_info(CS_MSGTYPE) ct_res_info(CS_TRANS_STATE)

結果タイプ	呼び出し可能なルーチン
Compute row results	ct_bind ct_compute_info ct_describe ct_res_info(CS_CMD_NUMBER) ct_res_info(CS_NUM_COMPUTES) ct_res_info(CS_NUMDATA) ct_res_info(CS_TRANS_STATE) ct_dyndesc(CS_USE_DESC) ct_dynsqlda(CS_SQLDA_BIND)
CS_CMD_DONE	ct_res_info(CS_CMD_NUMBER) ct_res_info(CS_ROW_COUNT) ct_res_info(CS_TRANS_STATE)
CS_CMD_SUCCEED	ct_res_info(CS_CMD_NUMBER) ct_res_info(CS_ROW_COUNT) ct_res_info(CS_TRANS_STATE)
CS_CMD_FAIL	ct_res_info(CS_CMD_NUMBER) ct_res_info(CS_ROW_COUNT) ct_res_info(CS_TRANS_STATE)
Regular row format results	ct_describe ct_res_info(CS_CMD_NUMBER) ct_res_info(CS_CMD_NUMDATA) ct_res_info(CS_TRANS_STATE)
Compute row format results	ct_compute_info ct_describe ct_res_info(CS_CMD_NUMBER) ct_res_info(CS_NUM_COMPUTES) ct_res_info(CS_NUMDATA) ct_res_info(CS_TRANS_STATE)
Describe results	ct_describe ct_res_info(CS_CMD_NUMBER) ct_res_info(CS_NUMDATA) ct_res_info(CS_TRANS_STATE) ct_dyndesc(CS_GETATTR) ct_dyndesc(CS_GETCNT) ct_dynsqlda(CS_GET_IN) ct_dynsqlda(CS_GET_OUT)
Extended error data results	ct_bind ct_describe ct_res_info(CS_NUMDATA) ct_res_info(CS_TRANS_STATE)
Notification results	ct_bind ct_describe ct_res_info(CS_NUMDATA) ct_res_info(CS_TRANS_STATE)

保留中の結果

同じ接続を共有する複数のコマンド構造体は、結果が接続上で保留中の場合、相互にブロックし合うことがあります。「保留中の結果」とは、コマンド結果が完全に処理されていない状態を示す用語です。

たとえば、2 つのコマンド構造体 (A と B) が、同じ接続構造体を共有する場合を想定します。A が Results Available のステータスにある場合、B は、サーバへのコマンド送信をブロックされます。これは、接続に保留中の結果があるためです。B は、A が現在のコマンドの結果すべてを処理して、保留中の結果がないことを示すステータスに移行するまで、ブロックされたままの状態になります。

保留中の結果を示すステータスは次のとおりです。

- Command sent
- Results available
- ANSI-style cursor end-data
- Fetchable results
- Sent fetchable nested command
- Processing fetchable nested command
- Fetching results
- Sent fetching nested command
- Undefined
- In receive passthrough
- In send passthrough

保留中の結果を示さないステータスは次のとおりです。

- Idle
- Command initiated
- Fetchable cursor results
- Fetchable nested command
- Fetching cursor results
- Fetching nested command
- Processing fetching nested command
- Result set canceled

各コマンド・ステータスの定義については、表 A-1 (162 ページ) を参照してください。

索引

A

action パラメータ　40
Adaptive Server Enterprise
　動的 SQL の実装　123
　トランザクション・ステータス　67
　メッセージと拡張エラー・データ　67
　ユーザ定義データ型　58

B

binary データ型　51
bit データ型　52
blk_alloc　32
blk_drop　32
buffer パラメータ　41
buflen パラメータ　41

C

character データ型　52
Client-Library
　アプリケーションのコンパイルとリンク　5
　エラーとメッセージ　34
　拡張エラー・データ　67
　終了　26
　初期化　17, 18
　メッセージ　59
　メッセージの生成　59
　リターン・コード　59
Client-Library の終了　26
CS_BIGDATETIME データ型　53
CS_BIGTIME データ型　53
CS_BINARY データ型　51
CS_BIT データ型　52
CS_BLKDESC 構造体　29, 32
CS_BROWSEDESC 構造体　33
CS_CLIENTMSG 構造体　33, 34
　メッセージ・テキストの保管　66
CS_CMD_DONE 結果タイプ
　意味　98
CS_CMD_FAIL 結果タイプ
　意味　98
CS_CMD_SUCCEED 結果タイプ
　意味　98
CS_COMMAND 構造体「コマンド構造体」参照　23
cs_config　18
　呼び出す場合　5
CS_CONNECTION 構造体　29
　「接続構造体」参照　21
CS_CONTEXT 構造体　29
　「コンテキスト構造体」参照　17
cs_ctx_alloc
　呼び出す場合　5
cs_ctx_drop
　呼び出す場合　6
CS_CUR_ID プロパティ　120
CS_CUR_NAME プロパティ　120
CS_CUR_ROWCOUNT プロパティ　120
CS_CUR_STATUS プロパティ　120
CS_DATAFMT 構造体　33, 34, 35
　Client-Library ルーチン　34
　CS-Library ルーチン　34
CS_DATE データ型　53
CS_DATEREC 構造体　33
CS_DATETIME データ型　53
CS_DATETIME4 データ型　53
CS_DECIMAL データ型　55
CS_DIAG_TIMEOUT_FAIL プロパティ
　インライン・メッセージ処理　65
CS_DS_OBJECT 隠し構造体　29
CS_EXTRA_INF プロパティ
　インライン・メッセージ処理　65
CS_FAIL 記号　38
CS_FALSE 記号　38
CS_FLOAT データ型　55
CS_FMT_PADBLANK フォーマット定数　37
CS_FMT_PADNULL フォーマット定数　37
CS_FMT_UNUSED フォーマット定数　37
CS_IMAGE データ型　56
CS_INT データ型　55
CS_IODESC 構造体　33, 35

CS_LOC_PROP プロパティ　18
CS_LOCALE 構造体　29, 32
CS_LOGINFO 構造体　29, 32
CS_LONGBINARY データ型　51
CS_LONGCHAR データ型　52
CS_MAX_NAME 記号　38
CS_MESSAGE_CB プロパティ　18
CS_MONEY データ型　55
CS_MONEY4 データ型　55
CS_NO_TRUNCATE プロパティ　66
　連続したメッセージ　65
CS_NULLTERM 記号　38
CS_NUMERIC データ型　55
CS_PROP_SSL_LOCALID 構造体　33
CS_REAL データ型　55
CS_SERVERMSG 構造体　33, 35
　メッセージ・テキストの保管　66
CS_SMALLINT データ型　55
CS_TEXT データ型　56
CS_TIME データ型　53
CS_TINYINT データ型　55
CS_TRAN_COMPLETED トランザクション・ステータス　68
CS_TRAN_FAIL トランザクション・ステータス　68
CS_TRAN_IN_PROGRESS トランザクション・ステータス　68
CS_TRAN_STMT_FAIL トランザクション・ステータス　68
CS_TRAN_UNDEFINED トランザクション・ステータス　68
CS_UNITEXT データ型　56
CS_VARBINARY データ型　51
CS_VARCHAR データ型　52
CS-Library
　CS-Library メッセージ・コールバックのインストール　21
　コンテキスト・プロパティの設定　18
cstypes.h ヘッダ・ファイル　43
ct_bind　88
　CS_DATAFMT 構造体　34
　呼び出す場合　6
ct_br_column　34
ct_callback　64
　呼び出す場合　5
ct_cancel
　カーソル結果のキャンセル　91
ct_close
　呼び出す場合　6
ct_cmd_alloc　23
　呼び出す場合　6
ct_cmd_drop
　呼び出す場合　6
ct_cmd_props　24
ct_command　24, 71
　言語コマンドの起動　72
　呼び出す場合　6
ct_compute_info　94
　呼び出す場合　94, 95
ct_con_alloc
　呼び出す場合　5
ct_con_props　22
　呼び出す場合　5
ct_config　19
　呼び出す場合　5
ct_connect　23
　呼び出す場合　5
ct_cursor　71, 106
　select 文を直接実行するためのカーソルの宣言　109
　ストアド・プロシージャを実行するためのカーソルの宣言　110
　呼び出す場合　6
ct_describe　88
　CS_DATAFMT 構造体　34
　CS_DATEREC 構造体　35
　呼び出す場合　6
ct_diag
　使用　64
　メッセージのインライン処理　64
ct_dynamic　71, 126
　準備文を実行するためのカーソルの宣言　112
　呼び出す場合　6
ct_exit
　呼び出す場合　6
ct_fetch　88
　呼び出す場合　6
ct_getloginfo
　CS_LOGINFO 構造体　32
ct_init　19
　呼び出す場合　5, 19
ct_keydata
　呼び出す場合　117
ct_options
　呼び出す場合　5
ct_param　71
　CS_DATAFMT 構造体　34
ct_res_info　88
　呼び出す場合　6

ct_results 86
CS_CMD_DONE 98
CS_CMD_FAIL 98
CS_CMD_SUCCEED 98
result_type の他の値 97
カーソル結果 89
完全に処理された結果 89
呼び出す場合 6
ct_send 24
呼び出す場合 6
ct_setloginfo
CS_LOGINFO 構造体 32
ct_setparam 71
ctpublic.h ヘッダ・ファイル
データ型定義 43
内容 17

D

datetime データ型 53
decimal データ型 55

F

float データ型 55

M

money データ型 55

N

null 代入値 57
CS_BIGDATETIME デフォルト 57
CS_BIGTIME デフォルト 57
CS_BINARY_TYPE デフォルト 57
CS_BIT_TYPE デフォルト 57
CS_BOUNDARY_TYPE デフォルト 57
CS_CHAR_TYPE デフォルト 57
CS_DATETIME_TYPE デフォルト 57
CS_DATETIME4_TYPE デフォルト 57
CS_DECIMAL_TYPE デフォルト 57
CS_FLOAT_TYPE デフォルト 57
CS_IMAGE_TYPE デフォルト 57
CS_INT_TYPE デフォルト 57
CS_MONEY_TYPE デフォルト 57
CS_MONEY4_TYPE デフォルト 57
CS_NUMERIC_TYPE デフォルト 57
CS_REAL_TYPE デフォルト 57
CS_SENSITIVITY_TYPE デフォルト 57
cs_setnull 57
CS_SMALLINT_TYPE デフォルト 57
CS_TEXT_TYPE デフォルト 57
CS_TINYINT_TYPE デフォルト 57
CS_VARBINARY_TYPE デフォルト 57
CS_VARCHAR_TYPE デフォルト 57
ユーザ定義データ型の定義 58
NULL パラメータ 38
numeric データ型 55

O

Open Client
ユーザ定義データ型 58
outlen パラメータ 41

R

real データ型 55
RPC コマンド
起動 71

S

SQL
動的 SQL 121
SQLCA 構造体 33, 35, 36
CS_EXTRA_INF プロパティ 65
連続したメッセージをサポートしない 66
SQLCODE 構造体 33, 36
CS_EXTRA_INF プロパティ 65
連続したメッセージをサポートしない 66
SQLSTATE 構造体 33, 36
CS_EXTRA_INF プロパティ 65
連続したメッセージをサポートしない 66

T

text および image
　データの記述　35
　データを操作するルーチン　56
text および image データ型　56

あ

アプリケーション
　簡単なプログラムの手順　5
　コンパイルとリンク　5
　終了　26
　ランタイム動作条件　5

い

インライン・メッセージ処理　64
　CS_DIAG_TIMEOUT_FAIL プロパティ　65
　CS_EXTRA_INF プロパティ　65
　ct_diag　61
　SQLCA、SQLCODE、SQLSTATE 構造体　36
　コールバックとの組み合わせ　61
　コールバックと比較したときの利点　60

え

エラー処理とメッセージ処理
　2 つの方法　60
　インライン方式　64
　コールバック・メソッド　61
　定義　5
　必要性　5
　メッセージのトランケーションを防ぐ　65
　連続したメッセージ　65
エラー。「メッセージ」参照　59

お

オペレーティング・システム・メッセージ　66

か

カーソル
　ct_cursor での宣言　109, 110
　ct_dynamic での宣言　112
　select 文を実行するための宣言　109
　カーソルのサーバ ID 番号の取得　120
　カーソルの名前の取得　120
　カーソル・ローの現在の値の取得　120
　カーソル・ローの設定　113
　準備された動的 SQL 文　112
　ステータスの取得　120
　ストアド・プロシージャを実行するための宣言　110
　プロパティ　120
カーソル結果
　処理方法　88
カーソル・コマンド
　起動　71, 106
階層、制御構造体　31
隠し構造体
　CS_COMMAND　29
　CS_CONNECTION　29
　CS_CONTEXT　29
　CS_DS_OBJECT　29
　CS_LOCALE　29
　CS_LOGINFO　29
拡張エラー・データ　67
カスタム・データ変換ルーチン
　インストール　58
型
　定義　43
型定数　37
　定義　49
カラム・レベルのデータ・アクセス　67

き

記号
　CS_FALSE　38
　CS_SUCCEED　38
　CS_TRUE　38
記号定数　38
　変更の可能性のある値　38
記述結果
　処理のルーチン　96
　処理方法　96

規則
パラメータ 38, 42
起動
コマンド 24, 70

く

クライアント・メッセージ 59
クライアント・メッセージ・コールバック
Client-Library が呼び出せないとき 62
定義 62
有効な戻り値 62
呼び出し可能な Client-Library ルーチン 62

け

計算結果
処理のルーチン 93
処理方法 94
計算フォーマット結果
処理方法 97
結果
処理方法 6, 24
結果処理 6, 24
結果処理のループ 86
言語コマンド
起動 71

こ

公開された構造体 33
CS_BROWSEDESC 33
CS_CLIENTMSG 33
CS_DATAFMT 33
CS_DATAREC 33
CS_IODESC 33
CS_PROP_SSL_LOCALID 33
CS_SERVERMSG 33
SQLCA 33
SQLCODE 33
SQLSTATE 33
構造体 29, 36
CS_BLKDESC 32
CS_CLIENTMSG 34
CS_COMMAND 30
CS_COMMAND 構造体 29
CS_COMMAND 構造体の割り付け 23
CS_CONNECTION 30
CS_CONNECTION 構造体 29
CS_CONNECTION 構造体の割り付け 21
CS_CONTEXT 29
CS_CONTEXT 構造体 29
CS_CONTEXT 構造体の割り付け 17
CS_DATAFMT 34
CS_DATEREC 35
CS_DS_OBJECT 32
CS_IODESC 35
CS_LOCALE 32
CS_LOGINFO 32
CS_SERVERMSG 35
SQLCA 35, 36
SQLCODE 36
SQLDA 36
SQLSTATE 36
隠し構造体 29
基本制御構造体 31
公開された構造体 33
コマンド構造体 29
コンテキスト構造体 29
制御構造体の階層 31
接続構造体 29
項目番号パラメータ 40
コールバック 20
インストール 64
インストール解除 64
インライン・メッセージ処理との組み合わせ 61
インライン・メッセージ処理と比較したときの利点 60
置き換え 64
コールバック・ロケーションの保管 62
メッセージ処理のための使用 61
「クライアント・メッセージ・コールバック」参照 20
国際化のサポート 33
このガイドの章、要約 ix
コマンド
起動 24, 70
サーバへの送信 23
パラメータの定義 71
コマンド構造体 31
プロパティの設定と取得 23
割り付け 23
割り付け解除 26

コンテキスト構造体　29, 30
　CICS の制限　30
　Client-Library プロパティの設定　19
　CS-Library プロパティの設定　18
　情報をプロパティとして保管する　30
　割り付け　17
　割り付け解除　27
コンパイルとリンク　5

さ

サーバ
　コマンドの送信　5, 23
　サーバへの接続　5, 21
　サーバへのログイン　23
　トランザクション・ステータス　67
サーバの結果
　処理方法　24
サーバへの接続　5, 21
サーバへのログイン　22
サーバ・メッセージ　36, 59
　拡張エラー・データ　67
　サーバ・メッセージとメッセージ結果の相違点　96
　説明　60
サーバ・メッセージ結果　60
サーバ・メッセージ・コールバック
　定義　63
　有効な戻り値　64
　呼び出し可能な Client-Library ルーチン　63

し

準備実行オペレーション
　基準　125
　実行手順　126
　利点　126
準備文
　使用するとき　125
　定義　122, 127
初期化
　Client-Library　18
　例　17

す

ストアド・プロシージャ
　Client-Library カーソル　111
　実行するためのカーソルの宣言　111

せ

制御構造体
　基本制御構造体　31
制御構造体のスコープ　31
接続構造体　30, 32
　情報をプロパティとして保管する　30
　プロパティの設定と取得　21
　割り付け　21
　割り付け解除　26
設定
　Client-Library コンテキスト・プロパティ　19
　CS-Library コンテキスト・プロパティ　18
　コマンド構造体プロパティ　23
　接続構造体プロパティ　21

そ

送信、サーバへのコマンドの送信　5, 23
即時実行オペレーション
　基準　124

つ

通常ロー結果
　処理方法　87
通常ローのフォーマット結果
　処理方法　97

て

定数　36, 38
　型定数　37
　その他の定数　38
　フォーマット定数　37
ディレクトリ・オブジェクト構造体　32
データ
　データとプログラム変数の記述　34

データ型
Adaptive Server のユーザ定義型 58
binary 51
bit 52
CS_BIGDATETIME 53
CS_BIGTIME 53
CS_BINARY 51
CS_BIT 52
CS_DATE 53
CS_DATETIME 53
CS_DATETIME4 53
CS_DECIMAL 55
CS_FLOAT 55
CS_IMAGE 56
CS_INT 55
CS_LONGBINARY 51
CS_LONGCHAR 52
CS_MONEY 55
CS_MONEY4 55
CS_NUMERIC 55
CS_REAL 55
CS_TEXT 56
CS_TIME 53
CS_TINYINT 55
CS_UNITEXT 56
CS_VARBINARY 51
CS_VARCHAR 52
datetime 53
decimal 55
float 55
money 55
real 55
SMALLINT 55
text および image 56
型定数 37
数値 55
データ型のまとめ 50
文字 52
ユーザ定義データ型 58
データ型の定義 43
データ送信コマンド
起動 71
データの変換
カスタム変換ルーチンのインストール 58

と

動作のバージョン、Client-Library
設定 18
動的 SQL
Adaptive Server Enterprise の制限事項と稼働条件 123
Adaptive Server での実装方法 123
カーソル 112
ストアド・プロシージャの出力パラメータと戻り値の取得不可 123
制限事項 122
制約 122
代替 131
代替としてのストアド・プロシージャ 131
パフォーマンスの制限 122
目的 121
利点 122
動的 SQL コマンド
起動 71, 126
トランザクション・ステータス 67
CS_TRAN_FAIL 68
CS_TRAN_IN_PROGRESS 68
CS_TRAN_STMT_FAIL 68
CS_TRAN_UNDEFINED 68

な

長いメッセージ 65

は

バインド
定義 88
パッケージ・コマンド
起動 71
パラメータ
action パラメータ 40
action、buffer、buflen、outlen パラメータの相関関係 41
buffer パラメータ 41
buflen パラメータ 41
NULL パラメータ 38
outlen パラメータ 41
規則 38, 42
項目番号 40
コマンドのパラメータの定義 71

出力パラメータ文字列　39
入力パラメータ文字列　39
非ポインタ・パラメータ　39
ポインタ・パラメータ　38
未使用パラメータ　38
パラメータ結果
処理のルーチン　92
処理方法　92
バルク・コピー
CS_BLKDESC 構造体　32

ふ

ファイル
ヘッダ・ファイル　17
ファイル名、ライブラリ用　5
フェッチ
定義　88
フォーマット結果
CS_EXPOSE_FMTS プロパティ　97
処理のルーチン　97
処理方法　97
フォーマット定数　37
CS_FMT_NULLTERM　37
CS_FMT_PADBLANK　37
CS_FMT_PADNULL　37
CS_FMT_UNUSED　37
ブラウズ・モード・カラムの情報　34
プログラム構造　5, 27
簡単なプログラムの手順　5
結果処理　24
コールバックのインストール　20
コマンドの送信　23
サーバへの接続　21
終了　26
設定　17
プログラムの環境の設定　17
プログラム変数
記述　34
プロパティ
Client-Library コンテキスト・プロパティの設定　19
CS-Library コンテキスト・プロパティの設定　18
コマンド・プロパティの設定　23
接続プロパティの設定　21
ログイン・プロパティ　32

へ

ヘッダ・ファイル　17
ctpublic.h　43

み

未使用パラメータ　38

め

メッセージ
Client-Library メッセージ　59
Sybase およびユーザ定義のメッセージの範囲　96
オペレーティング・システム・メッセージ　66
クライアント・メッセージ　34, 59
サーバ・メッセージ　36, 59, 60
チャンク　65
トランケーションを防ぐ　65
連続化　65
メッセージ結果
サーバ・メッセージと異なるメッセージ結果　60
処理のルーチン　95
処理方法　95
メッセージ・コールバック
Client-Library　20
CS-Library　21
メッセージ・コマンド
起動　71
メッセージ処理とエラー処理。「エラー処理とメッセージ処理」参照　59

ゆ

ユーザ定義データ型　58
Adaptive Server Enterprise のユーザ定義型　58

り

リターン・コード　59
リターン・ステータスの結果
　処理のルーチン　93
　処理方法　93
リモート・プロシージャ・コール
　RPC と execute 文との比較　79
　利点　79

れ

連続したメッセージ　65
　CS_NO_TRUNCATE プロパティ　66

ろ

ローカライゼーション
　CS_LOCALE 構造体　32
　CS_LOCALE 構造体を操作するルーチン　33
ロー結果
　処理方法　87
ログイン・プロパティ　32

わ

割り付け
　CS_BLKDESC 構造体　32
　CS_COMMAND 構造体　23
　CS_CONNECTION 構造体　21
　CS_CONTEXT 構造体　17
割り付け解除
　CS_BLKDESC 構造体　32
　CS_COMMAND 構造体　26
　CS_CONNECTION 構造体　26
　CS_CONTEXT 構造体　27