Client-Library/C リファレンス・マニュアル

# Open Client™

15.7

ドキュメント ID：DC32433-01-1570-01

改訂：2012 年 6 月

Copyright © 2012 by Sybase, Inc. All rights reserved.

このマニュアルは Sybase ソフトウェアの付属マニュアルであり、新しいマニュアルまたはテクニカル・ノートで特に示されないかぎりは、後続のリリースにも付属します。このマニュアルの内容は予告なしに変更されることがあります。このマニュアルに記載されているソフトウェアはライセンス契約に基づいて提供されるものであり、無断で使用することはできません。

削除このマニュアルの内容を弊社の書面による事前許可を得ずに、電子的、機械的、手作業、光学的、またはその他のいかなる手段によっても、複製、転載、翻訳することを禁じます。

Sybase の商標は、the Sybase trademarks page (http://www.sybase.com/detail?id=1011207) で確認できます。Sybase およびこのリストに掲載されている商標は、米国法人 Sybase, Inc. の商標です。® は、米国における登録商標であることを示します。

このマニュアルに記載されている SAP、その他の SAP 製品、サービス、および関連するロゴは、ドイツおよびその他の国における SAP AG の商標または登録商標です。

Java および Java 関連の商標は、米国およびその他の国における Oracle およびその関連会社の商標または登録商標です。

Unicode と Unicode のロゴは、Unicode, Inc. の登録商標です。

このマニュアルに記載されている上記以外の社名および製品名は、当該各社の商標または登録商標の場合があります。

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

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

# 目次

はじめに ........ xi

第 1 章 Client-Library を使用する前に ........ 1
- Sybase クライアント／サーバ・アーキテクチャ ........ 1
  - クライアントのタイプ ........ 2
  - サーバのタイプ ........ 2
- Open Client と Open Server ........ 3
  - Open Client ........ 4
  - Open Server ........ 5
  - 共有共通ライブラリ ........ 5
  - 汎用インタフェースとしての Client-Library ........ 7
  - Embedded SQL とライブラリ・アプローチとの比較 ........ 8
- アプリケーションの開発に必要な知識 ........ 8
  - プログラミング・インタフェース ........ 9
  - はじめる前に ........ 10

第 2 章 Client-Library トピックス ........ 11
- 非同期プログラミング ........ 12
  - 非同期アプリケーション ........ 13
  - 非同期ルーチン ........ 14
  - CS_BUSY リターン・コード ........ 14
  - 完了 ........ 15
  - Client-Library の割り込みレベルでのメモリ要件 ........ 19
  - レイヤ構成のアプリケーション ........ 19
- ブラウズ・モード ........ 22
  - ブラウズ・モードの使用 ........ 23
  - ブラウズ・モードの where 句 ........ 24
  - ブラウズ・モード条件 ........ 24
- コールバック ........ 25
  - コールバックのタイプ ........ 26
  - コールバックは常にサポートされているわけではない ........ 30
  - コールバック・ルーチンのインストール ........ 30
  - コールバック・イベントが発生する場合 ........ 31

コールバック・ルーチンの取得および置換 ............................ 31
コールバックでの Client-Library 呼び出しの制限 ................... 32
CS_PUBLIC によるコールバック宣言 .................................. 33
クライアント・メッセージ・コールバック ........................... 33
完了コールバック ............................................................. 37
ディレクトリ・コールバック ................................................ 42
暗号コールバック ............................................................. 45
ネゴシエーション・コールバック ......................................... 50
ノーティフィケーション・コールバック ................................ 53
セキュリティ・セッション・コールバック ........................... 55
サーバ・メッセージ・コールバック ...................................... 59
シグナル・コールバック ..................................................... 63
SSL 検証コールバック ....................................................... 65
機能 ....................................................................................... 67
ワイド・テーブルと大きなページ・サイズ ........................... 67
unichar データ型 ................................................................ 73
unitext データ型 ................................................................. 76
xml データ型 ...................................................................... 78
機能および接続の TDS レベル .............................................. 79
機能の設定および取得 ......................................................... 80
Client-Library と SQL 構造体 ................................................. 81
公開された構造体と隠し構造体 ............................................ 82
CS_BROWSEDESC 構造体 ................................................... 84
CS_CLIENTMSG 構造体 ....................................................... 85
CS_DATAFMT 構造体 .......................................................... 93
CS_IODESC 構造体 ............................................................. 99
CS_OID 構造体 ................................................................. 101
CS_SERVERMSG 構造体 .................................................... 103
SQLCA 構造体 .................................................................. 106
SQLCODE 構造体 .............................................................. 108
SQLSTATE 構造体 ............................................................. 108
コマンド ............................................................................... 109
コマンドの送信 ................................................................ 110
使用するコマンド・タイプの決定 ....................................... 112
接続マイグレーション ............................................................ 112
デバッグ ............................................................................... 113
デバッグの有効化 ............................................................. 114
ディレクトリ・サービス ......................................................... 115
ディレクトリ・サービスのプロバイダとドライバ ................. 115
LDAP .............................................................................. 116
アプリケーションでのディレクトリの使い方 ....................... 118
ディレクトリの編成 ........................................................... 119

エラー処理 ........ 136
    初期化時のエラー・レポート ........ 136
    エラー処理とメッセージ処理 ........ 137
    CS_EXTRA_INF プロパティ ........ 141
    長いメッセージの連続化 ........ 141
    拡張エラー・データ ........ 143
    サーバ・トランザクション・ステータス ........ 146
サンプル・プログラム ........ 147
    サンプル・プログラム内の Client-Library ルーチン ........ 150
ヘッダ・ファイル ........ 153
高可用性フェールオーバ ........ 153
    interfaces ファイルへの hafailover 行の追加 ........ 154
    Client-Library アプリケーションの変更 ........ 155
    Sybase フェールオーバでの isql の使い方 ........ 156
interfaces ファイル ........ 157
    interfaces ファイル・エントリの概要 ........ 158
    interfaces ファイルのサーバ・オブジェクト ........ 159
国際化のサポート ........ 161
    アプリケーションが CS_LOCALE 構造体を使用する必要がある場合 ........ 161
    CS_LOCALE 構造体の使用 ........ 162
    ローカライゼーション情報の検索 ........ 164
    ロケール・ファイル ........ 165
ストアド・プロシージャ・パラメータとしてのラージ・オブジェクト ........ 167
    パラメータとしての少量の LOB データの送信 ........ 167
    パラメータとしての大量の LOB データの送信 ........ 170
マクロ ........ 175
    メッセージ番号の復号化 ........ 175
    CS_CAP_TYPE 構造体内のビット操作 ........ 175
    sizeof 演算子の使い方 ........ 175
    関数のプロトタイプ宣言 ........ 176
マルチスレッド・アプリケーション : シグナル処理 ........ 177
    基本概念 ........ 177
    非スレッド環境でのシグナル処理 ........ 177
    シグナルの種類 ........ 178
    シグナル・ハンドラ ........ 178
    シグナルのマスキング ........ 178
    シグナルの配信 ........ 179
    sigwait を使用した非同期シグナルの処理 ........ 180
    Sybase の専用シグナル・ハンドラ ........ 181
    SIGTRAP シグナル ........ 182
    Sun の ALARM ルーチンと SETITIMER ルーチンの使用 ........ 182

マルチスレッド・プログラミング ........................................... 182
スレッドについて ........................................... 183
複数スレッドの利点 ........................................... 184
スレッドの種類 ........................................... 184
スレッドセーフ・コードの書き込み ........................................... 185
共有データおよび共有リソースに対する
アクセスの逐次化 ........................................... 186
依存アクションの同期化 ........................................... 187
スレッドアンセーフなシステム・ルーチンの呼び出し ........ 188
デッドロックの回避 ........................................... 189
マルチスレッド・プログラムに対する Client-Library の
制限事項 ........................................... 189
コンテキスト・レベル・ルーチンの呼び出し ..................... 190
接続レベル・ルーチンの呼び出し ..................................... 193
CS_LOCALE 構造体の使用 ........................................... 194
スレッドセーフなコールバック・ルーチンのコー
ディング ........................................... 195
スレッドと完全非同期モード ........................................... 195
Client-Library のマルチスレッド・プログラミング・
モデル ........................................... 198
オプション ........................................... 200
外部からのオプションの設定 ........................................... 200
パラメータ ........................................... 207
バッチ・パラメータ ........................................... 207
プロパティ ........................................... 208
プロパティ、オプション、機能の比較 ............................. 208
ログイン・プロパティ ........................................... 209
プロパティの設定と取得 ........................................... 209
3 つのコンテキスト・プロパティ ..................................... 209
プロパティがサポートされているかどうかのチェック ........ 210
ログイン・プロパティのコピー ..................................... 211
外部からのプロパティの設定 ........................................... 212
プロパティのクイック・リファレンス ............................. 212
プロパティについて ........................................... 235
レジスタード・プロシージャ ........................................... 276
Client-Library のノーティフィケーションの受信 ................. 278
ノーティフィケーションの非同期受信 ............................. 279
結果 ........................................... 280
通常ローの結果 ........................................... 281
カーソル・ロー結果 ........................................... 281
パラメータ結果 ........................................... 281
ストアド・プロシージャ・リターン・ステータス結果 ........ 282
計算ロー結果 ........................................... 282
メッセージ結果 ........................................... 283

記述結果 ........ 283
フォーマット結果 ........ 283
結果処理のプログラム構造体 ........ 284
項目の値の取得 ........ 288
バッチ処理での結果バインドの保持 ........ 289
可変長データの複数のローを配列に読み込む ........ 289
セキュリティ機能 ........ 290
ネットワークベースのセキュリティ ........ 291
Open Client/Open Server の SSL (Secure Socket Layer)..... 303
インターネット通信の概要 ........ 304
SSL の概要 ........ 307
Adaptive Server Enterprise のセキュリティ機能 ........ 316
サーバ・ディレクトリ・オブジェクト ........ 319
サーバ・ディレクトリ・オブジェクトの使い方 ........ 319
サーバ・ディレクトリ・オブジェクトの内容 ........ 320
interfaces ファイルのサーバ・オブジェクト ........ 325
サーバの制限 ........ 326
Open Server の制限 ........ 326
Adaptive Server Enterprise の制限 ........ 327
サポートされるクライアント／サーバ機能 ........ 327
text および image データの処理 ........ 328
text カラムまたは image カラムの取得 ........ 328
text または image カラムの更新 ........ 330
text カラムや image カラムを含むテーブルへの
データ設定 ........ 335
text および image を更新するためのサーバのグロー
バル変数 ........ 336
データ型のサポート ........ 339
データ型の概要 ........ 339
データ型を操作するルーチン ........ 341
Open Client のデータ型 ........ 341
Open Client のユーザ定義データ型 ........ 351
ランタイム設定ファイルの使い方 ........ 352
外部設定の有効化 ........ 353
Open Client/Server ランタイム設定ファイルの構文 ........ 355
ランタイム設定ファイルのキーワード ........ 359

**第 3 章 ルーチン ........ 369**
ct_bind ........ 371
ct_br_column ........ 384
ct_br_table ........ 385
ct_callback ........ 387
ct_cancel ........ 392
ct_capability ........ 397

ct_close ..... 406
ct_cmd_alloc ..... 409
ct_cmd_drop ..... 411
ct_cmd_props ..... 412
ct_command ..... 418
ct_compute_info ..... 428
ct_con_alloc ..... 432
ct_con_drop ..... 434
ct_con_props ..... 436
ct_config ..... 452
ct_connect ..... 461
ct_cursor ..... 466
ct_data_info ..... 492
ct_debug ..... 496
ct_describe ..... 501
ct_diag ..... 507
ct_ds_dropobj ..... 515
ct_ds_lookup ..... 516
ct_ds_objinfo ..... 523
ct_dynamic ..... 530
ct_dyndesc ..... 538
ct_dynsqlda ..... 548
ct_exit ..... 556
ct_fetch ..... 559
ct_get_data ..... 566
ct_getformat ..... 572
ct_getloginfo ..... 573
ct_init ..... 575
ct_keydata ..... 580
ct_labels ..... 583
ct_options ..... 585
ct_param ..... 590
ct_poll ..... 601
ct_recvpassthru ..... 607
ct_remote_pwd ..... 609
ct_res_info ..... 612
ct_results ..... 619
ct_scroll_fetch ..... 629
ct_send ..... 639
ct_send_data ..... 644
ct_send_params ..... 654
ct_sendpassthru ..... 656
ct_setloginfo ..... 658
ct_setparam ..... 660
ct_wakeup ..... 673

付録 A **国際化ライブラリのメッセージ**


国際化ライブラリのメッセージ ........ 677
INTE_NOVAL ........ 677
INTE_NOENTRY ........ 677
INTE_OFLOW ........ 678
INTE_ENTRYOF ........ 678
INTE_ODDHEX ........ 679
INTE_BADFILE ........ 679
INTE_BADLOC ........ 680
INTE_NOCOM ........ 680
INTE_BADFFMT ........ 680
INTE_BADVER ........ 681
INTE_BADPH ........ 681
INTE_BADTYPE ........ 682
INTE_SPECOF ........ 682
INTE_NOCUST ........ 683
INTE_BADFMTSTR ........ 683
INTE_INVALBUF ........ 684
INTE_NEGBUFLEN ........ 684
INTE_INVALCS ........ 684
INTE_BADLFNM ........ 685
INTE_INVALTEXT ........ 685
INTE_INVALSRC ........ 686
INTE_INVALPTR ........ 686
INTE_BADNSTARS ........ 687
INTE_MONTHS ........ 687
INTE_SMONTHS ........ 688
INTE_DAYS ........ 688
INTE_PATHOF ........ 688
INTE_LTLONG ........ 689
INTE_DUPDF ........ 689
INTE_BADSECT ........ 690
INTE_FOPEN ........ 690
INTE_FCLOSE ........ 691
INTE_FREAD ........ 691
INTE_NOSYB ........ 691
INTE_FINFO ........ 692
INTE_NOMEM ........ 692


付録 B **SSL エラー・メッセージ**


SSL エラー・メッセージ ........ 693
1: ベンダの呼び出しの失敗 ........ 693
3: メモリ割り付けの失敗 ........ 693
6: 不正なポインタ ........ 694
60: SSL マスタ・コンテキスト初期化の失敗 ........ 694
61: 部分 I/O の設定の失敗 ........ 694

62: SSL プロトコル・バージョンの設定の失敗 .......................... 695
63: ランダム番号ジェネレータの作成の失敗 .............................. 695
64: ランダム番号ジェネレータの初期化の失敗........................... 696
65: ランダム番号ジェネレータのエントロピの生成の失敗......... 696
69: コンテキストの複製の失敗 .................................................. 696
70: 子 SSL/TLS コンテキストの作成の失敗 ................................ 697
71: プロトコル・バージョンの取得の失敗 ................................. 697
72: 不明なプロトコル・バージョン........................................... 698
73: 不明な暗号 .......................................................................... 698
74: 暗号スイートの設定の失敗 .................................................. 698
75: ローカル identity プロパティのロードの失敗 ........................ 699
76: 認証局のファイルのロードまたは読み込みの失敗................ 699
77: ピアの認証情報取得の失敗 .................................................. 700
78: ピアの認証取得の失敗 ......................................................... 700
81: 認証リファレンスの設定の失敗........................................... 700
84: SSL ハンドシェイクの失敗 .................................................. 701
85: SSL のサーバ側への設定の失敗 .......................................... 701
86: SSL のクライアント側への設定の失敗 ................................ 702
87: SSL のエンドポイント情報取得の失敗 ................................ 702
88: SSL コンテキスト情報取得の失敗....................................... 702
89: 読み込みエラー ................................................................... 703
90: 書き込みエラー ................................................................... 703
91: リモート認証の DN フィールドのカウント取得の失敗......... 704
92: 識別名情報の抽出の失敗...................................................... 704
93: リモート認証の拡張子のカウント取得の失敗 ....................... 704
94: 拡張子情報の抽出の失敗...................................................... 705
95: クライアント認証の取得の失敗........................................... 705

**用語解説 ........................................................................................................ 707**

**索引 ............................................................................................................... 727**

# はじめに

このマニュアルは、Open Client™ Client-Library の C バージョンのリファレンス情報について説明しています。

**対象読者**

このマニュアルは、Client-Library アプリケーションを作成するプログラマ向けのリファレンス・マニュアルです。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、Client-Library を簡単に紹介します。
- 「第 2 章 Client-Library トピックス」では、Client-Library ルーチンを使用してサーバから text および image 値を読み込む方法など、プログラムを作成するための情報について説明します。この章では、Client-Library 構造体、オプション、エラー・メッセージ、規則についても説明します。
- 「第 3 章 ルーチン」では、使用できるパラメータや戻り値など、各 Client-Library ルーチンに固有の情報について説明します。
- 「付録 A 国際化ライブラリのメッセージ」では、国際化エラー・メッセージの情報について説明します。
- 「付録 B SSL エラー・メッセージ」では、SSL 関連のエラー・メッセージの情報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 および SDK 新機能』( 各 Windows、Linux、UNIX 版 ) では、Open Server と Software Developer's Kit の新機能について説明しています。このマニュアルは、新機能の提供に伴って改訂されます。
- 使用しているプラットフォームの 『Open Server のリリース・ノート』には、Open Server に関する重要な最新情報が記載されています。

- 使用しているプラットフォームの『Software Developer’s Kit リリース・ノート』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』には、jConnect に関する重要な最新情報が記載されています。
- 使用しているプラットフォームの『Open Client/Server 設定ガイド』では、システムを設定して Open Client/Server 製品を実行する方法について説明しています。
- 『Open Client Client-Library/C プログラマーズ・ガイド』では、Client-Library アプリケーションの設計方法および実装方法について説明しています。
- 『Open Server Server-Library/C リファレンス・マニュアル』では、Open Server Server-Library のリファレンス情報について説明しています。
- 『Open Client および Open Server Common Libraries リファレンス・マニュアル』では、CS-Library のリファレンス情報について説明しています。CS-Library は、Client-Library と Server-Library の両方のアプリケーションで役に立つユーティリティ・ルーチンの集まりです。
- 『Open Server DB-Library/C リファレンス・マニュアル』では、C バージョンの Open Client DB-Library™ のリファレンス情報について説明しています。
- 使用しているプラットフォームの『Open Client/Server プログラマーズ・ガイド補足』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
- 『Sybase® SDK DB-Library Kerberos 認証オプションのインストールおよびリリース・ノート』では、DB-Library で使用する MIT Kerberos セキュリティ メカニズムをインストールして有効にする方法について説明しています。DB-Library でサポートされる Kerberos セキュリティ・メカニズムの機能は、ネットワーク認証サービスと相互認証サービスのみです。

- 『Open Client/Server 開発者用国際化ガイド』では、国際化されたアプリケーションとローカライズされたアプリケーションを作成する方法について説明しています。
- 『Open Client Embedded SQL™/C プログラマーズ・ガイド』では、C アプリケーションで Embedded SQL および Embedded SQL プリコンパイラを使用する方法について説明しています。
- 『Open Client Embedded SQL™/COBOL プログラマーズ・ガイド』では、COBOL アプリケーションで Embedded SQL および Embedded SQL プリコンパイラを使用する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』(Microsoft Windows および UNIX 版 ) では、Microsoft Windows および UNIX 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ています。
- 『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、Adaptive Server 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ています。
- 『Perl 用 Adaptive Server Enterprise データベース・ドライバ・プログラマーズ・ガイド』では、Perl 開発者が Perl スクリプトを使用して Adaptive Server のデータベースに接続し、情報をクエリまたは変更する方法について説明しています。
- 『PHP 用 Adaptive Server Enterprise 拡張モジュール・プログラマーズ・ガイド』では、PHP 開発者が Adaptive Server データベースに対してクエリを実行する方法について説明しています。
- 『Python 用 Adaptive Server Enterprise 拡張モジュール・プログラマーズ・ガイド』では、Adaptive Server データベースに対してクエリを実行するときに使用できる Sybase 固有の Python インタフェースについて説明しています。

**その他の情報**

Sybase Product Documentation Web サイトを使用して製品について詳しく知ることができます。

- Sybase Product Documentation Web サイトには、標準の Web ブラウザを使用してアクセスできます。また、製品ドキュメントのほか、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。

  Sybase Product Documentation Web サイトは、Product Documentation (http://www.sybase.com/support/manuals/) にあります。

**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 (http://www.sybase.com/support/techdocs/) を指定します。
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 (http://certification.sybase.com/) を指定します。
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 (http://www.sybase.com/support/techdocs/) を指定します。
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 the 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**

| 凡例 | 定義 |
|---|---|
| **コマンド** | 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。 |
| *変数* | 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。

**コード例**

このマニュアルにあるコード例の大半は、Client-Library にあるオンラインのプログラム例と同じものです。使用しているプラットフォームのコード例とその Sybase インストール・ディレクトリでのロケーションの説明については、『Open Client/Server プログラマーズ・ガイド補足』を参照してください。

マニュアルにあるコード例の多くは、次のようにサンプル・プログラムで定義されているルーチンや記号を使用します。

```
if (ct_close(connection, CS_UNUSED) != CS_SUCCEED)
 {
     ex_error("ct_close failed");
 }
```

このマニュアルのコード例で使用される *ex_* および *EX_* 記号は、すべてサンプル・プログラムで定義されています。これらは Client-Library のプログラミング・インタフェースの一部ではありません。

**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、クライアント／サーバ・アーキテクチャと Open Server アプリケーションの概要について説明します。この章は、次の項目で構成されています。



| トピック | ページ |
|---|---|
| Sybase クライアント／サーバ・アーキテクチャ | 1 |
| Open Client と Open Server | 3 |
| アプリケーションの開発に必要な知識 | 8 |



この章では、Client-Library アプリケーションを開発するための初歩的な情報については説明していません。初歩的な情報については、『Open Client Client-Library/C プログラマーズ・ガイド』の「第 1 章 Client-Library を使用する前に」を参照してください。

## Sybase クライアント／サーバ・アーキテクチャ

クライアント／サーバ・アーキテクチャでは、コンピューティング作業が「クライアント」と「サーバ」間で分担されます。

クライアントはサーバに対して要求を行い、サーバから返された要求の結果を処理します。たとえば、データベース・サーバのデータを要求するクライアント・アプリケーションもあれば、部屋の温度を下げるよう、環境制御サーバに要求するクライアント・アプリケーションもあります。

サーバは、データやその他の情報をクライアントへ返したり、何らかのアクションをとったりして、要求に応答します。たとえば、データベース・サーバは表形式のデータとそのデータについての情報をクライアントに返し、電子メール・サーバは受信メールを最終的な宛先へ送信します。

クライアント／サーバ・アーキテクチャは、従来のプログラム・アーキテクチャよりも次のような点で優れています。

- 共通のサービスが 1 つの場所で、つまり 1 つのサーバで処理されるため、アプリケーションのサイズと複雑さが大幅に軽減されます。これによって、クライアント・アプリケーションが単純化され、コードの重複が減り、アプリケーションの保守が容易になります。
- クライアント／サーバ・アーキテクチャは、さまざまなアプリケーションの間での通信を容易にします。類似性のない通信プロトコルを使用するクライアント・アプリケーションは直接通信することはできませんが、両方のプロトコルを「理解できる」サーバを介せば通信できます。
- クライアント／サーバ・アーキテクチャにより、アプリケーションを独立したコンポーネントの集合として開発することが可能になります。これにより、アプリケーションの他の部分に影響を及ぼさずに、個々のコンポーネントを変更または置換できます。

## クライアントのタイプ

クライアントとは、サーバへの要求を行うアプリケーションです。クライアントには次のものが含まれています。

- OmniConnect™ や OpenSwitch™ などの Sybase ミドルウェア製品
- isql および bcp など、Adaptive Server Enterprise が提供するスタンドアロン・ユーティリティ
- Open Client ライブラリを使用して作成されたアプリケーション
- jConnect™ for JDBC™ で記述された Java アプレットや Java アプリケーション
- Embedded SQL™ を使用して記述されたアプリケーション

## サーバのタイプ

Sybase の製品群には、次に示すように、サーバとサーバを構築するためのツールが含まれています。

- Adaptive Server Enterprise はデータベース・サーバです。Adaptive Server Enterprise は、1 つまたは複数のデータベースに格納された情報を管理します。

- Open Server は、カスタム・サーバ・アプリケーションを作成するために必要なツールとインタフェースを提供します。

  Open Server アプリケーションは、どのタイプのサーバにもなりえます。たとえば、Open Server アプリケーションは、特殊な計算を行ったり、リアルタイム・データにアクセスしたり、電子メールなどのサービスにインタフェースできます。Open Server アプリケーションは、Open Server Server-Library で提供されるビルディング・ブロックを使って、個々に開発されます。

Adaptive Server Enterprise と Open Server アプリケーションには、次のような類似点があります。

- 両者ともサーバであり、クライアントの要求に応答します。
- クライアントは、Adaptive Server Enterprise と Open Server アプリケーションのどちらとも、Open Client 製品を介して通信します。

次のような相違点もあります。

- アプリケーション・プログラマは、Server-Library のビルディング・ブロックを使用し、カスタム・コードを供給して Open Server アプリケーションを作成しなければなりません。Adaptive Server Enterprise は完成されており、カスタム・コードは必要ありません。
- Open Server アプリケーションは、どのタイプのサーバにもなることができるので、各種プログラム言語を理解できるよう作成できます。Adaptive Server Enterprise はデータベース・サーバであり、理解するのは Transact-SQL だけです。
- Open Server アプリケーションは、Sybase のアプリケーションとサーバはもちろん、Sybase プロトコル以外でも通信できます。Adaptive Server Enterprise は、仲介として Open Server ゲートウェイ・アプリケーションを使用して、Sybase 以外のアプリケーションやサーバと通信できますが、直接には Sybase のアプリケーションおよびサーバとしか通信できません。

# Open Client と Open Server

Sybase では、カスタマによるクライアントおよびサーバ・アプリケーション・プログラムの作成を可能にする次の 2 つの製品群を提供しています。

- Open Client
- Open Server

## Open Client

Open Client は、カスタマ・アプリケーション、サードパーティ製品、および他の Sybase 製品が Adaptive Server Enterprise および Open Server と通信するためのインタフェースです。

Open Client には 2 つのコンポーネントがあると考えられます。つまり、プログラミング・インタフェースとネットワーク・サービスです。

Open Client は、クライアント・アプリケーションを作成するための中心的なプログラミング・インタフェースとして Client-Library と DB-Library を提供しています。

- Open Client Client-Library/C については、このマニュアルで説明しています。Client-Library インタフェースは、サーバによって管理されるカーソルと、System 10 以降の製品群に追加された、その他の新しい機能をサポートしています。
- Open Client DB-Library は、以前の Open Client アプリケーションをサポートするための別個の API です。DB-Library については、『Open Client DB-Library/C リファレンス・マニュアル』を参照してください。

Client-Library プログラムも、Client-Library アプリケーションと Server-Library アプリケーションの両方で使用されるルーチンを提供する CS-Library を使用します。Client-Library アプリケーションは、Bulk-Library のルーチンを使用して高速なデータ転送を行うこともできます。

CS-Library と Bulk-Library はどちらも Open Client 製品に含まれています。これらのライブラリについては、「共有共通ライブラリ」を参照してください。

Open Client ネットワーク・サービスには Sybase Net-Library があり、TCP/IP や DECnet などの、特定のネットワーク・プロトコルをサポートします。アプリケーション・プログラマには Net-Library インタフェースは見えません。ただし、プラットフォームによってはアプリケーションが別のシステムのネットワーク設定用に異なる Net-Library ドライバを必要とすることがあります。Net-Library ドライバは、使用するホスト・プラットフォームに応じて、システムの Sybase の設定で指定するか、プログラムをコンパイルしてリンクするときに指定します。

ドライバの設定方法については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。Client-Library プログラムの構築方法については、使用しているプラットフォームの『Open Client/Server プログラマーズ・ガイド補足』を参照してください。

## Open Server

Open Server は、カスタム・サーバの作成に必要なツールとインタフェースを提供します。Open Client のように、Open Server は、インタフェースとネットワーク・サービスから構成されます。

Open Server アプリケーションを作成するための中心的なプログラミング・インタフェースは Server-Library です。Server-Library 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。Server-Library プログラムは Client/Server-Library ( 省略して CS-Library と呼びます ) を使用します。Gateway Server-Library アプリケーションも Client-Library と Bulk-Library のルーチンを使用できます。Client-Library、CS-Library、Bulk-Library はすべて Open Server 製品に含まれています。

Open Server ネットワーク・サービスは、透過的です。

## 共有共通ライブラリ

Open Client 製品と Open Server 製品はどちらも Bulk-Library と CS-Library を提供しています。これらのライブラリはクライアント・アプリケーションとサーバ・アプリケーションのどちらにも役立つルーチンを提供します。CS-Library と 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-Library

CS-Library は Client-Library プログラムと Open Server プログラム用のユーティリティ・ルーチンを提供します。CS-Library は Client-Library プログラムのための中心的なデータ構造体 (CS_CONTEXT) を割り付けます。CS-Library は、データの変換やクライアント文字セットと言語のローカライズのための機能も提供します。クライアントとサーバ間で送信されるデータ型の定義は、CS-Library、Client-Library、Server-Library で同じです。

DB-Library は CS-Library と統合されていません。DB-Library と CS-Library は共通のデータ構造体を共有せず、そのデータ型の定義は異なります。

## Bulk-Library

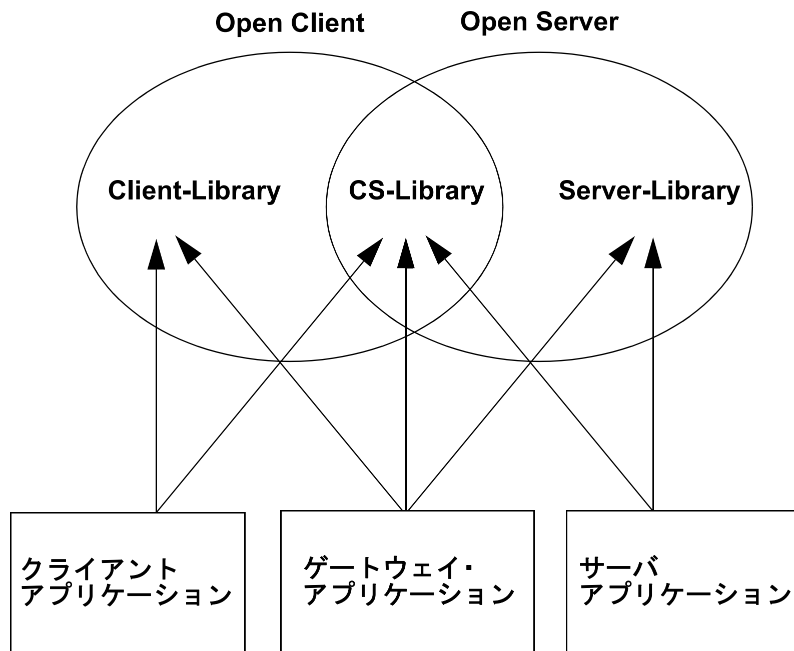
Bulk-Library/C は、Client-Library アプリケーションと Server-Library アプリケーションが、高速なデータ転送用に Adaptive Server Enterprise の「バルク・コピー」インタフェースを使用できるようにするルーチンを提供します。Client-Library プログラマは、そのアプリケーションがバルク・コピー・インタフェースを使用してデータを転送する必要がない場合は Bulk-Library について知る必要はありません。Bulk-Library、Client-Library、Server-Library は、クライアントとサーバ間で受け渡すデータの共通な型定義を共有します。

Adaptive Server Enterprise が暗号化カラムをサポートしている場合は、暗号化カラムのバルク・コピーもサポートされます。

DB-Library には専用のバルク・コピー・インタフェースがあり、Bulk-Library とともに使用することはできません。

次の図は、Open Client と Open Server に含まれているライブラリ間の関係を示します。

**図 1-1: Open Client/Opesn Server ライブラリの関係**




たとえば、クライアント・アプリケーションには、Client-Library と CS-Library への呼び出しがありますが、クライアントとサーバの両方として動作するアプリケーションには、Client-Library、CS-Library、および Server-Library への呼び出しがあります。

DB-Library は、Client-Library、CS-Library、Bulk-Library とはまったく別のインタフェースですが、Open Server ゲートウェイで使用できます。DB-Library には、共通のデータ構造体と型定義を Server-Library と共有するという Client-Library の利点がありません。

## 汎用インタフェースとしての Client-Library

Client-Library は汎用インタフェースです。Open Server やゲートウェイ・アプリケーションを介することで、Client-Library アプリケーションは、Adaptive Server Enterprise と同様に、外部のアプリケーションや Sybase 以外のサーバに接続できます。

Client-Library は汎用的なので、特定のサーバの制限が適用されることも影響を受けることもありません。たとえば、Client-Library では、ストアド・プロシージャのパラメータに text および image を使用できますが、Adaptive Server Enterprise ではこれらは使用できません。

Client-Library のアプリケーションを記述する場合、アプリケーションがどのサーバを使用するかを考慮しなければなりません。あるサーバについてどのような制約があるか不明なときは、そのサーバのマニュアルを参照してください。

アプリケーションは、ct_capability を呼び出し、特定のクライアント／サーバの接続がサポートする機能を検知できます。

## Embedded SQL とライブラリ・アプローチとの比較

Open Client ライブラリ・アプリケーションと Embedded SQL アプリケーションのどちらも、Adaptive Server Enterprise に SQL コマンドを送信できます。

Embedded SQL アプリケーションは、SQL コマンドがインラインにあります。Embedded SQL のプリコンパイラは、SQL コマンドを ClientLibrary の呼び出しに置き換えて処理します。11.0 以降の Sybase プリコンパイラはすべて、文書化された Client-Library と CS-Library 呼び出しだけで構成されるランタイム・ライブラリを使用します。基本的にプリコンパイラは Embedded SQL アプリケーションを Client-Library アプリケーションに変換します。この Client-Library アプリケーションはホスト言語コンパイラを使用してコンパイルされます。

Open Client ライブラリ・アプリケーションはライブラリ・ルーチンを使用して SQL コマンドを送り、プリコンパイラを必要としません。

一般に、Embedded SQL アプリケーションの方が作成とデバッグは簡単ですが、ライブラリ・アプリケーションの方が Open Client ルーチンの柔軟性と機能をより十分に活用できます。

# アプリケーションの開発に必要な知識

以下に、必要なプログラミング・インタフェースについて説明するとともに、Client-Library の機能の概要を示します。

## プログラミング・インタフェース

新しい Client-Library プログラマは、プログラミング・インタフェースについて、次のような知識が必要です。

- Client-Library：クライアント・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合です。Client-Library のルーチン名は、ct_init のように「ct_」で始まります。「第 3 章 ルーチン」を参照してください。
- CS-Library：クライアント・アプリケーションとサーバ・アプリケーションの両方に役立つユーティリティ・ルーチンの集まりです。Client-Library ルーチンは CS-Library 内に割り付けられている構造体を使用するので、すべての Client-Library アプリケーションには、少なくとも 1 つの CS-Library に対する呼び出しが含まれます。CS-Library のルーチン名は、cs_ctx_alloc のように「cs_」で始まります。これらの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』の CS-Library についての章を参照してください。
- Client-Library アプリケーションと Server-Library アプリケーションが高速なデータ転送用に Adaptive Server Enterprise の「**バルク・コピー**」インタフェースを使用できるようにするルーチンの集まりである Bulk-Library についての知識。Adaptive Server Enterprise が暗号化カラムをサポートしている場合は、暗号化カラムのバルク・コピーもサポートされます。Client-Library プログラマは、そのプログラムでバルク・コピー・インタフェースを使用してデータを転送する必要がない場合は Bulk-Library について知る必要はありません。Bulk-Library ルーチン名は、blk_alloc のように「blk_」で始まります。これらの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』の Bulk-Library についての章を参照してください。

Client-Library プログラマは、そのクライアント・プログラムが接続するサーバについての知識も必要です。

- Adaptive Server Enterprise に接続する場合は、クライアント・アプリケーション開発者は、Transact-SQL 言語、Adaptive Server Enterprise データベースにアクセスするための Sybase による SQL (Structured Query Language) の実装についての知識が必要です。クライアント・アプリケーション・プログラマは、アプリケーションで使用する Adaptive Server Enterprise データベース内のテーブルとストアド・プロシージャについての知識も必要です。

- Open Server ゲートウェイや他の Open Server アプリケーションに接続する場合、クライアント・アプリケーション開発者は、サーバによってサポートされる機能についての知識が必要です。たとえば、すべての Open Server が言語コマンドをサポートするように作成されているとはかぎりません。RPC コマンドで使用できるレジスタード・プロシージャのセットだけを提供しているものもあります。サーバが言語コマンドをサポートする場合は、クライアント・プログラマはサポートされるクエリ言語についての知識が必要です。

## はじめる前に

簡単なサンプル・プログラムを含む、Client-Library 機能の概要については、『Open Client Client-Library/C プログラマーズ・ガイド』の「第 1 章 Client-Library を使用する前に」を参照してください。

# 第 2 章　Client-Library トピックス

この章では、次の項目について説明します。

- 非同期プログラミング、ブラウズ・モード、text および image のサポートなどの、Client-Library プログラミングのトピックス
- カーソルの宣言やオープンなど、特定のプログラミング作業を行うためのルーチンの使用方法
- Client-Library のプロパティ、データ型、オプション、パラメータ規則、および構造体

この章では、次の項目について説明します。


| トピック | ページ |
| --- | --- |
| 非同期プログラミング | 12 |
| ブラウズ・モード | 22 |
| コールバック | 25 |
| 機能 | 67 |
| Client-Library と SQL 構造体 | 81 |
| コマンド | 109 |
| 接続マイグレーション | 112 |
| デバッグ | 113 |
| ディレクトリ・サービス | 115 |
| エラー処理 | 136 |
| サンプル・プログラム | 147 |
| ヘッダ・ファイル | 153 |
| 高可用性フェールオーバ | 153 |
| interfaces ファイル | 157 |
| 国際化のサポート | 161 |
| ストアド・プロシージャ・パラメータとしての ラージ・オブジェクト | 167 |
| マクロ | 175 |
| マルチスレッド・アプリケーション：シグナル処理 | 177 |
| マルチスレッド・プログラミング | 182 |
| オプション | 200 |
| パラメータ | 207 |

| トピック | ページ |
| --- | --- |
| プロパティ | 208 |
| レジスタード・プロシージャ | 276 |
| 結果 | 280 |
| セキュリティ機能 | 290 |
| サーバ・ディレクトリ・オブジェクト | 319 |
| サーバの制限 | 326 |
| text および image データの処理 | 328 |
| データ型のサポート | 339 |
| ランタイム設定ファイルの使い方 | 352 |


# 非同期プログラミング

非同期アプリケーションは、ある種のオペレーションが完了するまで待たされる時間を効率的に利用するために設計されています。通常、ネットワークまたは外部デバイスからの読み込みおよびそれらへの書き込みは、直接プログラムを実行するよりかなり遅くなります。サーバがコマンドを処理して結果をクライアント・アプリケーションに送り返すのにも時間がかかります。

アプリケーションによっては、アイドル時間を含んでいるいくつかのタスクを実行するものがあります。対話型アプリケーションでは、たとえば次のことを行います。

1 ユーザ入力を待つ。

2 サーバ X への接続に関するコマンドを実行する。

3 サーバ Y への接続に関するコマンドを実行する。

Client-Library の非同期モードは、そのようなアプリケーションがこれらのタスクを同時に実行するのに役立ちます。サーバ・コマンドを実行するときに、コマンドを送信したり、結果を読み込んだりするルーチンはすぐに制御を戻します。これは、アプリケーションが別のルーチンを呼び出して、別の接続に関するコマンド・オペレーションを開始でき、ユーザ入力に対する応答時間を短縮できることを意味します。

Client-Library の非同期モードは、タスクの同時実行を実現する方法の 1 つです。この他に、複数のスレッドを使用する方法があります。マルチスレッド環境で Client-Library を使用する方法については、「マルチスレッド・プログラミング」(182 ページ ) を参照してください。

## 非同期アプリケーション

デフォルトでは、Client-Library アプリケーションは同期アプリケーションです。ネットワークに対して読み込みや書き込みを行うルーチンは、必要な I/O 要求すべてを完了するまでは呼び出し元に制御を戻しません。

非同期アプリケーションを作成する場合は、アプリケーション・プログラマは Client-Library プロパティ CS_NETIO を設定することによって、コンテキストまたは接続レベルで Client-Library の非同期動作を有効にしてください。使用可能なネットワーク I/O モードは次のとおりです。

- 完全非同期 (CS_ASYNC_IO) − 非同期ルーチンはすぐに CS_PENDING を返します。要求されたオペレーションが完了すると、接続の完了コールバックが自動的に呼び出されます。
- 遅延非同期 (CS_DEFER_IO) − 非同期ルーチンはすぐに CS_PENDING を返します。アプリケーションは周期的に ct_poll を呼び出してオペレーションが完了したかどうか調べる必要があります。オペレーションが完了した場合は、ct_poll は接続の完了コールバックを呼び出します。ct_poll は ( 完了しているオペレーションがある場合は ) どのオペレーションが完了したかも示すので、遅延非同期アプリケーションでは、必要に応じて完了コールバックがなくても動作できます。
- 同期 (CS_SYNC_IO) − すべての Client-Library ルーチンは要求されたオペレーションが完了するまでは制御が戻りません。このモードはデフォルトです。

完全非同期モードまたは遅延非同期モードが有効な場合、ネットワークに対して読み込みや書き込みを行うすべての Client-Library ルーチンは次のいずれかの動作を実行します。

- 要求されたオペレーションを開始し、すぐに CS_PENDING を返します。
- この接続に対して非同期オペレーションが開始され結果がまだ処理されていないことを示す CS_BUSY を返します。接続に対して非同期オペレーションが開始され結果が未処理の場合、非同期ではないルーチンが呼び出されたときも、CS_BUSY を返します。

CS_PENDING を返すことによって、ルーチンは、要求されたオペレーションが開始され非同期に完了することを示します。ポーリングによって ( つまり、ct_poll を定期的に呼び出すことによって )、または Client-Library がアプリケーションの完了コールバックを呼び出すときに、アプリケーションはその呼び出しの完了ステータスを受け取ります。これらの方法については、「完了」(15 ページ ) を参照してください。

## 非同期ルーチン

次の Client-Library ルーチンが非同期で動作します。

- ct_cancel
- ct_close
- ct_connect
- ct_ds_lookup
- ct_fetch
- ct_get_data
- ct_options
- ct_recvpassthru
- ct_results
- ct_send
- ct_send_data
- ct_sendpassthru

## CS_BUSY リターン・コード

コマンドまたは接続構造体をパラメータとする Client-Library ルーチンは、CS_BUSY を返します。CS_BUSY 応答は、関連する接続が現在ビジーであり、非同期オペレーションが完了するのを待っている状態で、ルーチンが実行できないことを示します。

非同期オペレーションが未処理の状態になっているときに、アプリケーションは次のルーチンを呼び出すことができます。

- CS_CONTEXT 構造体をパラメータとするいずれかのルーチン。CS_CONTEXT 構造体がオプションのパラメータである場合、NULL 以外の値にしてください。
- ct_cancel(CS_CANCEL_ATTN)
- ct_cmd_props(CS_USERDATA)
- ct_con_props(CS_USERDATA)
- ct_poll

# 完了

CS_PENDING を返す非同期モードの Client-Library 呼び出しはすべて完了ステータス (completion status) を生成します。この値は、ルーチンに対する同期モードの呼び出しによって返されるリターン・コード (CS_SCCEED や CS_FAIL など ) に相当します。

アプリケーションは、非同期ルーチンの完了をポーリングまたは完了コールバックによって調べます。ポーリングの場合は、アプリケーションは ct_poll を定期的に呼び出して非同期呼び出しが完了したかどうかを調べます。完了コールバックの場合は、非同期呼び出しが完了するとアプリケーションに自動的に通知されます。

Client-Library を正常に終了するには、すべての非同期オペレーションが完了した後に ct-exit を呼び出します。

非同期オペレーション実行中に ct_exit が呼び出されると、ルーチンは CS_FAIL を返し、CS_FORCE_EXIT を使用しても Client-Library は正常に終了しません。

## 遅延非同期の完了

アプリケーションは、オペレーションが完了したことを ct_poll が示すまで、ct_poll を定期的に呼び出して完了ステータスを調べます。このモードのオペレーションは「遅延非同期 (deferred-asynchronous)」と呼ばれ、CS_NETIO プロパティを CS_DEFER_IO に設定した場合に相当します。

アプリケーションが完了コールバックをインストールすると、ct_poll が完了を検出したときに ct_poll によってコールバック・ルーチンが呼び出されます。アプリケーション自体が ct_poll を呼び出す必要があります。

アプリケーションは、ct_poll から非同期呼び出しの完了ステータスを受け取ります。完了コールバックがインストールされている場合は、完了ステータスを入力パラメータとしても受け取ります。

---

**注意**　サポートされている非同期モードの詳細については、使用しているプラットフォーム用の『Open Client/Server プログラマーズ・ガイド補足』の Client-Library の章を参照してください。

---

## 完全非同期の完了

Client-Library がシグナル駆動型 I/O またはスレッド駆動型 I/O を使用しているプラットフォームでは、非同期ルーチンが完了するとClient-Library が自動的にアプリケーションの完了コールバック・ルーチンを呼び出します。このモードのオペレーションは「完全非同期(fully asynchronous)」と呼ばれ、CS_NETIO プロパティをCS_ASYNC_IO に設定した場合に相当します。

完全非同期接続の場合、アプリケーションは完了ステータスを調べる必要がありません。Client-Library が自動的にアプリケーションの完了コールバックを呼び出して、その完了コールバックが完了ステータスを入力パラメータとして受け取ります。完了コールバックについては、「完了コールバックの定義」(38 ページ ) を参照してください。

---

**注意**　ネイティブ・スレッドのサポートにより構築された Open Server ライブラリを使用する場合、CS_NETIO は CS_ASYNC_IO に設定でき、完全非同期動作がサポートされます。ネイティブ・スレッドのサポートで構築されていない Open Server ライブラリを使用する場合、CS_NETIO を CS_ASYNC_IO に設定すると CS_DEFER_IO と同様の動作になります。ただし、この場合はアプリケーションが ct_poll を呼び出して I/O 操作を完了するのではなく、Open Server がアプリケーションのポーリング・ルーチンを呼び出します。

---

非同期接続では、Client-Library が非同期オペレーションを完了して、開始したルーチンに制御を戻す前にコールバック・ルーチンを呼び出すことができます。この場合でも、開始したルーチンは CS_PENDING を返し、アプリケーションの完了コールバックは完了ステータスを受け取ります。

Client-Library の完全非同期オペレーションは、スレッド駆動型かシグナル駆動型のどちらかです。マルチスレッドとシグナル駆動型 I/O のどちらもサポートしないプラットフォームでは、Client-Library は完全非同期では動作できません。

### シグナル駆動型の完了処理

UNIX などの一部のプラットフォームでは、Client-Library はオペレーティング・システムのシグナル ( 割り込みとも呼ばれます ) を使用して、ネットワークを介して結果の読み込みやコマンドの送信を行います。内部的には、Client-Library はブロックされないシステム・コールを使用してネットワークを介して通信し、専用の内部シグナル・ハンドラをインストールしてこれらのシステム・コールの完了ステータスを受け取ります。

シグナル駆動型 I/O を使用しているプラットフォームでは、CS_NETIO プロパティが CS_ASYNC_IO でない場合でも Client-Library がシグナル駆動型である場合があります。シグナル駆動型 I/O のプラットフォームでは、次のいずれかの条件に合う場合に Client-Library がシグナル駆動型 I/O を使用します。

- CS_NETIO 接続プロパティの値が CS_ASYNC_IO である場合。
- CS_ASYNC_NOTIFS 接続プロパティの値が CS_TRUE である場合。デフォルトは CS_FALSE。このプロパティの詳細については、「非同期ノーティフィケーション」(237 ページ ) を参照してください。
- CS_PROP_MIGRATABLE の値が CS_TRUE ( デフォルト ) であり、かつクライアントがクライアント自身をマイグレートする可能性があるサーバにすでに接続している場合。

---

**警告！** Client-Library がシグナル駆動型 I/O を使用する場合は、アプリケーションによって発行されたシステム・コールの処理に、シグナルが割り込むことができます。システム・コールが割り込まれたことがエラー・コードに示されている場合は、再度呼び出しを発行してください。

---

Client-Library の完全非同期モードの実装にシグナル駆動型 I/O が使用されるプラットフォームの場合、完全非同期アプリケーションに次の制限があります。

- アプリケーションによって必要になるシグナル・ハンドラは、ct_callback を使用してインストールする必要があります。「シグナル・コールバック」(63 ページ ) を参照してください。
- アプリケーションは、Client-Library が割り込みレベルでメモリを取得できる安全な方法を提供しなければなりません。「Client-Library の割り込みレベルでのメモリ要件」(19 ページ ) を参照してください。
- Client-Library が完全非同期モードでシグナル駆動型 I/O を使用するシステム (UNIX) では、各システム・コールの後に戻り値とエラー・コードを調べて、そのシステム・コールが正常に完了したかどうか確認してください。システム・コールの中には、シグナルによって割り込まれると失敗するものもあります。呼び出しが割り込みを受けたことをエラー・コードが示している場合は、その呼び出しを再発行してください。この制限は、Client-Library がシグナルを使用しない Windows などのプラットフォームでは問題ありません。

### スレッド駆動型の完了処理

Windows などのプラットフォームでは、Client-Library はスレッド駆動型 I/O を使用して完全非同期モードで動作します。

この I/O 方式が使用される場合、Client-Library は内部的なワーカー・スレッドを生成してネットワークと通信します。ネットワーク I/O を必要とするルーチンをアプリケーションが呼び出すと、その I/O 要求はワーカー・スレッドに渡されます。非同期ルーチンは CS_PENDING を返し、ワーカー・スレッドは I/O の完了を待ちます。I/O 要求が完了すると、ワーカー・スレッドはアプリケーションの完了コールバックを呼び出します。

スレッド駆動型 I/O が使用されるプラットフォームでは、完全非同期アプリケーションに次のような制限があります。

- それぞれの完全非同期接続用にインストールされるアプリケーションのコールバック関数は、すべてスレッドセーフでなければなりません。
- アプリケーションの完了コールバックは Client-Library のワーカー・スレッドによって呼び出されるので、完了コールバックがスレッドセーフな方法でメインライン・コードと通信できるようにアプリケーション・ロジックを設計する必要があります。

Windows などのスレッド駆動型 I/O のプラットフォームでは、完全非同期プログラムはメインライン・コードがシングルスレッドの場合であってもコールバックの実行にマルチスレッドを使用します。スレッド駆動型 I/O の場合、Client-Library のワーカー・スレッドはそれぞれの完全非同期接続についてネットワークと通信します。ワーカー・スレッドは、検出したすべてのコールバック・イベントの接続のコールバックを呼び出します。「完全非同期の完了」(16 ページ ) を参照してください。

---

**注意**　Client-Library がスレッド駆動型 I/O を使用しているプラットフォームで完全非同期 I/O が有効な場合、アプリケーションのコールバックが Client-Library のワーカー・スレッドによって呼び出されます。このようなプラットフォームでは、アプリケーション自体がシングルスレッドの設計を使用している場合でも、完全非同期アプリケーションのコールバックはマルチスレッドを使用します。

---

マルチスレッド・アプリケーションの設計に影響する問題については、「マルチスレッド・プログラミング」(182 ページ ) を参照してください。

## Client-Library の割り込みレベルでのメモリ要件

Client-Library がシグナル駆動型 I/O を使用する UNIX ベースのシステムなどのオペレーティング・システムの場合、完全非同期アプリケーションは Client-Library が割り込みレベルのメモリ要件を満たすための手段を提供する必要があります。

通常、Client-Library ルーチンは、malloc を呼び出すことによって、メモリ要件を満たします。しかし、malloc の実装のすべてが、割り込みレベルで安全に呼び出されるわけではありません。この理由から、完全非同期アプリケーションは、Client-Library のメモリ要件を満たすために、Client-Library に代替手段を提供する必要があります。

Client-Library は、次の 2 とおりのメカニズムを提供し、非同期アプリケーションが Client-Library のメモリ要件を満たせるようにします。

- アプリケーションは CS_MEM_POOL プロパティを使用して、メモリ要件に対して使用するメモリ・プールを Client-Library に提供できます。
- アプリケーションは、CS_USER_ALLOC プロパティおよび CS_USER_FREE プロパティを使用して、Client-Library が割り込みレベルで安全に呼び出すことができるメモリの割り付けルーチンをインストールできます。

シグナル駆動型 I/O を使用しているプラットフォームでは、Client-Library が安全にメモリ要件を満たせるようにする安全な手段を完全非同期アプリケーションが提供できない場合、Client-Library の動作は予測不可能です。

Client-Library は、次の順で次のソースから、必要なメモリを確保しようとします。

1 メモリ・プール

2 ユーザ提供の割り付けおよび解放ルーチン

3 システム・ルーチン

## レイヤ構成のアプリケーション

非同期アプリケーションは、多くの場合、レイヤ構成になります。この種のアプリケーションでは、低レイヤは、低レベルの非同期処理の影響を与えないように高レイヤを保護する責任があります。

高レベルのレイヤは、一般的に次のものから構成されています。

- メインライン・コード
- 大きなオペレーションを非同期で実行するルーチン

  ここでいう「大きなオペレーション」とは、数個の Client-Library 呼び出しの完了が必要な作業のことです。たとえば、データベース・テーブルの更新は大きなオペレーションになります。その更新を実行するために、アプリケーションは ct_command、ct_send、および ct_results を呼び出すからです。

低レベルのレイヤは、一般に次のものから構成されています。

- 大きなオペレーションを実行するために必要な Client-Library ルーチン
- 低レベル非同期オペレーションの完了を処理するコード

## ct_wakeup および CS_DISABLE_POLL の使用

ct_wakeup および CS_DISABLE_POLL プロパティは、次に示すレイヤ構成の非同期アプリケーションで使用されます。

- レイヤ構成のアプリケーションは、CS_DISABLE_POLL を使用して、ct_poll が非同期の Client-Library ルーチンの完了を報告するのを抑制します。
- レイヤ構成のアプリケーションは、ct_wakeup を使用して、それより高レベルのレイヤに「大きな非同期オペレーション」が完了したことを通知します。

「大きなオペレーション」を実行するためのルーチンを使用しているレイヤ構成のアプリケーションは、一般的に、次のように ct_wakeup および CS_DISABLE_POLL を使用します。

1. アプリケーションは、初期化 ( 必要な場合 )、コールバック・ルーチンのインストール、接続のオープンなどを行います。
2. アプリケーションは、大きなオペレーションを実行するルーチンを呼び出します。
3. アプリケーションが ct_poll を使用して非同期完了をチェックする場合は、ルーチンではポーリングを無効にしてください。これにより、ct_poll が、低レベルの非同期完了を高レベル・レイヤに報告するのを抑制します。ポーリングを無効にするため、ルーチンは CS_DISABLE_POLL を CS_TRUE に設定します。

   アプリケーションが ct_poll を呼び出さない場合、ルーチンはポーリングを無効にする必要はありません。

4 ルーチンは ct_callback を呼び出して、高レベル・レイヤの完了コールバックを独自の完了コールバックと置き換えます。
5 ルーチンが作業を実行します。
6 ルーチンが、高レベル・レイヤの完了コールバックを再インストールします。
7 ポーリングが無効になっている場合、ルーチンは CS_DISABLE_POLL プロパティを CS_FALSE に設定して、再びポーリングを有効にします。
8 ルーチンは ct_wakeup を呼び出して、高レベル・レイヤの完了コールバック・ルーチンをトリガします。

## 例

非同期でデータベース更新を実行するアプリケーションは、ルーチン do_update を含む場合があります。ここで、do_update は、データベースの更新を実行するために必要なすべての Client-Library ルーチンを呼び出します。

メイン・アプリケーションは非同期で do_update を呼び出し、他の作業を続けます。

do_update は、呼び出されると、メイン・アプリケーションの完了コールバック・ルーチンを do_update 自体のコールバックで置き換えます ( これにより、メイン・アプリケーションのコールバック・ルーチンが低レベルの非同期完了によってトリガされることがなくなります )。その後、更新作業が続行されます。更新作業を実行するために、do_update は、非同期で動作する ct_send および ct_results を含む数個の Client-Library ルーチンを呼び出します。各非同期ルーチンが完了すると、do_update の完了コールバックがトリガされます。

do_update は、更新オペレーションを終了すると、メイン・アプリケーションの完了コールバックを再インストールし、関数 ID を、function に設定して ct_wakeup を呼び出します。これで、メイン・アプリケーションの完了コールバックがトリガされ、do_update の完了をメイン・アプリケーションに通知します。

# ブラウズ・モード

---

**注意** ブラウズ・モードは、Open Server アプリケーションや旧バージョンの Open Client ライブラリとの互換性を保つために Client-Library でサポートされています。新しい Open Client Client-Library アプリケーションでのブラウズ・モードの使用はおすすめしません。同じ機能を持ち、ブラウズ・モードより移植性の高い柔軟性のあるカーソルの使用をおすすめします。また、ブラウズ・モードは Sybase 独自のものであり、異機種環境での使用には適していないこともその理由です。

---

ブラウズ・モードは、データベース・ローをブラウズし、一度に 1 つのローの値を更新する手段を提供します。アプリケーション・プログラムの点からみると、各ローがブラウズされて更新される前に、データベースからプログラム変数に転送されなければならないため、この処理には数段階必要です。

ブラウズするローはデータベースにある実際のローではなく、プログラム変数にあるコピーなので、プログラムは、変数に対する値の変更を行って、元のデータベース・ローを確実に更新するようなものにしてください。特にマルチユーザ環境では、プログラムがローを選択して、そのローを更新するコマンドを送信する間に、あるユーザが行ったデータベースへの更新が、別のユーザの更新によって上書きされてしまうことを確実に防ぐ必要があります。ブラウズ可能なテーブル内の timestamp カラムが、このようなマルチユーザの更新を調整するのに必要な情報を提供します。

アプリケーションの中には、ユーザがアドホック・ブラウズ・モード・クエリを入力できるものもあるため、Client-Library は、2 つのルーチン、つまり ct_br_table と ct_br_column を提供しています。これにより、アプリケーションは、ブラウズ・モード結果セットが取り出される元テーブルおよびカラムについての情報を取得できます。この情報は、アプリケーションがブラウズ・モードの更新を実行するためにコマンドを構成しているときに役立ちます。

ブラウズ・モード・アプリケーションには 2 つの接続が必要です。1 つはデータを選択するための接続、もう 1 つは更新を実行するための接続です。

『ASE リファレンス・マニュアル』を参照してください。

## ブラウズ・モードの使用

概念上、ブラウズ・モードは次の 2 段階の手順で構成されます。

1. 1 つまたは複数のデータベース・テーブルから取り出されたカラムを持つローを選択します。
2. 適切である場合には、( 実際のデータベース・ローではなく ) 結果ローのカラムの値を一度にローを 1 つずつ変更し、新しい値を使用して元のデータベース・テーブルを更新します。

プログラムでは、上記の手順は次のように実装されます。

1. 接続の CS_HIDDEN_KEYS プロパティを CS_TRUE に設定します。これにより、Client-Library は、確実に結果セットの一部としてテーブルの *timestamp* カラムを返します。ブラウズ・モード更新では、*timestamp* カラムはマルチユーザ更新を調整するために使用されます。
2. select...for browse 言語コマンドを実行します。このコマンドは、通常ローの結果セットを生成します。この結果セットには、明示的に選択されたカラムの他に、隠しキー・カラム (*timestamp* カラムの 1 つ ) が含まれています。
3. ct_results が通常ロー結果を示した後で、ct_describe を呼び出して、結果カラムの CS_DATAFMT 記述を取得します。
    - *timestamp* カラムを示すために、ct_describe は、CS_TIMESTAMP と CS_HIDDEN ビットを *datafmt–>status フィールドに設定します。
    - 一般的な隠しキー・カラムを示すために、ct_describe は、CS_HIDDEN ビットを *datafmt–>status フィールドに設定します。CS_HIDDEN ビットが設定されていない場合、カラムは明示的に選択されたカラムです。
4. ct_bind を呼び出し、必要な結果カラムをバインドします。アプリケーションは、更新時に where 句を構築するためにこれらのカラムの値を必要とするので、すべての隠しカラムをバインドしてください。
5. 必要であれば ct_br_table を呼び出し、結果セットが取り出される元となったデータベース・テーブルについての情報を引き出します。また、必要に応じて ct_br_column を呼び出し、特定の結果セット・カラムについての情報を取得します。この種の情報は両方とも、データベースを更新するための言語コマンドを構築するときに役立ちます。

6 ループで ct_fetch を呼び出し、ローをフェッチします。変更が必要な値を含むローがフェッチされたときに、新しい値でデータベース・テーブルを更新します。必要な作業は次のとおりです。

- ローの隠しカラム (*timestamp* カラムなど ) を使用する where 句とともに、Transact-SQL の update 文が含まれる言語コマンドを作成します。
- サーバに言語コマンドを送り、コマンド結果を処理します。

  ブラウズ・モード update 文を含む言語コマンドが、CS_PARAM_RESULT 結果タイプの結果セットを生成します。この結果セットには、1 つの結果項目、つまり、ローの新しいタイムスタンプが含まれています。

  アプリケーションがこの同じローを再度更新しようとする場合、新しいタイムスタンプは後で使用するために保存されます。

ブラウズ・モードの 1 つのローが更新されると、アプリケーションは次のローをフェッチし、処理します。

## ブラウズ・モードの *where* 句

ブラウズ・モードで更新を実行するには、アプリケーションは次のような形式の where 句を使用して update 言語コマンドを送信します。

```
where key1 = value_1
 and key2 = value_2 ...
 and tsequal(timestamp, ts_value)
```

各パラメータの意味は次のとおりです。

- *key1*、*value_1*、*key2*、*value_2* などは、ct_br_table と ct_br_column を呼び出すことによって得られるキー・カラムとその値です。
- *ts_value* は、文字列に変換されるバイナリの timestamp カラム値です。

## ブラウズ・モード条件

ブラウズ・モードを使用するには、次の条件を満たしている必要があります。

- 結果セットを生成した select コマンドは、for browse というキーワードで終わらなければなりません。

- 更新されるテーブルは、ブラウズ可能なもので、各テーブルがユニーク・インデックスおよび timestamp カラムを持っている必要があります。
- 更新される結果カラムは、colname + 1 のような SQL 式の結果であってはなりません。

# コールバック

コールバックとは、あるトリガ・イベントが発生すると自動的に Client-Library によって呼び出されるユーザ指定のルーチンです。「コールバック・イベント」と呼ばれています。

コールバック・イベントの中には、アプリケーションが受け取るサーバの応答結果であるものもあります。たとえば、ノーティフィケーション・コールバック・イベントは、レジスタード・プロシージャのノーティフィケーションを Open Server から受け取るときに発生します。

その他のコールバック・イベントは、内部の Client-Library レベルで発生します。たとえば、クライアント・メッセージ・コールバック・イベントは、Client-Library がエラー・メッセージを生成するときに発生します。

Client-Library は、コールバック・イベントを認識すると、適切なコールバック・ルーチンを呼び出します。

Client-Library は、いくつかのコールバック・イベントを認識するために、ネットワークの読み込み処理に集中する必要があります。このタイプのほとんどのコールバック・イベントは、Client-Library がネットワークから結果を読み込むときに自動的に発生します。

ただし、Client-Library の非同期モードまたは Open Server のレジスタード・プロシージャ・ノーティフィケーションを使用するアプリケーションは、次の 2 つのタイプのコールバック・イベントで特別な処理が必要な場合があります。

- 完了コールバック・イベント。これは非同期モード・アプリケーションで、Client-Library の非同期ルーチンが完了したときに発生します。オペレーティング・システムに応じて、アプリケーションは完了通知を自動的に受け取るか、ポーリングによって受け取ります。「完了」(15 ページ ) を参照してください。

- ノーティフィケーション・コールバック・イベント。これはアプリケーションが Open Server ノーティフィケーションを受け取ったときに発生します。アプリケーションはノーティフィケーション・イベントを確実に受け取れるように特別な手順を実行する必要があります。「ノーティフィケーションの非同期受信」(279 ページ )を参照してください。

---

**注意**　コールバック・ルーチンのタイプによっては、システムの割り込みハンドラ内から、または Client-Library のワーカー・スレッドから実行できるものもあるので、アプリケーションのメインライン・コードとコールバックの両方からアクセスされるデータが安全に共有され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。

---

## コールバックのタイプ

表 2-1 は、コールバックのタイプと呼び出されるタイミングを示します。

**表 2-1 : コールバックのタイプ**

| コールバックのタイプ | 呼び出されるタイミング | 呼び出され方 |
|---|---|---|
| クライアント・メッセージ | Client-Library エラーまたは情報メッセージに対して応答するとき | Client-Library は、エラーまたは情報メッセージを生成したときに、クライアント・メッセージ・コールバックを自動的にトリガする。<br>「クライアント・メッセージ・コールバック」(33 ページ ) を参照。 |
| 完了 | 非同期 Client-Library ルーチンが完了したとき | 非同期ルーチンの完了はいつでも発生する。<br>シグナル駆動型またはスレッド駆動型 I/O をサポートしているプラットフォームでは、完了コールバックは、完了発生時に自動的に呼び出される。シグナル駆動型またはスレッド駆動型 I/O をサポートしていないプラットフォームでは、アプリケーションは ct_poll を使用して、完了しているルーチンがあるかチェックできる。<br>「完了コールバック」(37 ページ ) を参照。 |
| ディレクトリ | アプリケーションが ct_ds_lookup を呼び出した場合に開始されるディレクトリ検索のとき | 検索で見つかったディレクトリ・オブジェクトをアプリケーションに渡すために、Client-Library によって自動的に呼び出される。非同期接続では、完了コールバックの前に呼び出される。同期接続では、ct_ds_lookup が制御を戻す前に呼び出される。Client-Library は次のどちらかの状態になるまでコールバックを繰り返して呼び出す。<br>• 検索オペレーションで見つかったすべてのディレクトリ・オブジェクトをコールバックが受け取った場合<br>• コールバックが CS_SUCCEED を返す場合<br>「ディレクトリ・コールバック」(42 ページ ) を参照。 |

| コールバックのタイプ | 呼び出されるタイミング | 呼び出され方 |
|---|---|---|
| 暗号化 | 接続処理中、暗号化されたパスワードに対するサーバ要求に対して応答するとき | パスワードの暗号化が有効で暗号コールバックがインストールされている場合は、接続要求中にサーバが暗号化されたパスワードを要求するとき、Client-Library が自動的に暗号コールバックをトリガする。<br>暗号化が有効であり暗号コールバックがインストールされていない場合は、Client-Library はデフォルトのパスワード暗号化を実行する。<br>詳細については、「暗号コールバック」(45 ページ ) を参照。 |
| ネゴシエーション | 接続処理中の次のタイミング<br>• ログイン・セキュリティ・ラベルに対するサーバ要求に応答するとき<br>• サーバ・チャレンジに応答するとき | 接続の CS_SEC_NEGOTIATE プロパティが CS_TRUE に設定されている場合、接続試行中にサーバがログイン・セキュリティ・ラベルを要求したとき、Client-Library はネゴシエーション・コールバックを自動的にトリガする。<br>接続の CS_SEC_CHALLENGE プロパティが CS_TRUE に設定されている場合、接続試行中にサーバがチャレンジを発行したとき、Client-Library がネゴシエーション・コールバックを自動的にトリガする。<br>詳細については、「ネゴシエーション・コールバック」(50 ページ ) を参照。 |
| ノーティフィケーション | Open Server のノーティフィケーションを受け取ったとき | Open Server のノーティフィケーションは、いつでも受け取る可能性がある。Client-Library はノーティフィケーション情報を読み込んでアプリケーションのノーティフィケーション・コールバックを呼び出す。<br>CS_ASYNC_NOTIFS プロパティは、ノーティフィケーション・コールバックをトリガする方法を決定する。このプロパティの詳細については、「非同期ノーティフィケーション」(237 ページ ) と「ノーティフィケーション・コールバック」(53 ページ ) を参照。 |

| コールバックのタイプ | 呼び出されるタイミング | 呼び出され方 |
|---|---|---|
| セキュリティ・セッション | 接続処理中に接続がネットワーク・ベースのセキュリティ・サービスを使用するとき | ターゲット・サーバからのセキュリティ・セッション・チャレンジに応答して ct_connect によって自動的に呼び出される。<br>詳細については、「セキュリティ・セッション・コールバック」(55 ページ ) を参照。<br>**注意**　セキュリティ・セッション・コールバックは、自分のクライアントとリモート・サーバ間でダイレクト・セキュリティ・セッションを設定するゲートウェイ・アプリケーションだけで必要とされる。 |
| サーバ・メッセージ | サーバ・エラーまたは情報メッセージに対して応答するとき | サーバ・メッセージは、特定のコマンドの結果として発生する。アプリケーションがコマンド結果を処理するときに、Client-Library は、コマンドに関連するすべてのエラーまたは情報メッセージを読み込む。これらのメッセージは、自動的にサーバ・メッセージ・コールバックをトリガする。<br>詳細については、「サーバ・メッセージ・コールバック」(59 ページ ) を参照。 |
| シグナル | オペレーティング・システム・シグナルに対して応答するとき | ct_callback を使用してシグナル・ハンドラがインストールされている場合にシグナルを受信すると、Client-Library 自体のシグナル・ハンドラが自動的にシグナル・コールバックを呼び出す。<br>シグナルをサポートするプラットフォームでは、アプリケーションは ct_callback を呼び出して必要なシグナル・ハンドラをインストールする必要がある。<br>シグナル・コールバックの詳細については、「シグナル・コールバック」(63 ページ ) を参照。 |

| コールバックのタイプ | 呼び出されるタイミング | 呼び出され方 |
|---|---|---|
| SSL の検証 | 接続処理中に接続が SSL セッション・ベースのセキュリティ・サービスを使用するとき | SSL ハンドシェイク中に、ct_connect により自動的に呼び出される。<br>SSL 検証コールバックは、CS_SSLVALIDATE_CB を使用して ct_callback によりインストールされる。<br>SSL 検証コールバックの詳細については、「SSL 検証コールバック」(65 ページ ) を参照。 |

## コールバックは常にサポートされているわけではない

ポインタ参照による関数呼び出しをサポートしないプログラミング言語とプラットフォームの組み合わせでは、コールバックが実装されていない場合もあります。このような場合には、アプリケーションは次のようにします。

- ct_diag を使用して、Client-Library およびサーバ・メッセージをインラインで処理する必要があります。
- ct_poll を使用して、完了またはノーティフィケーション・コールバック・イベントのチェックを行うことができますが、そのイベントを処理する何らかのルーチンを直接呼び出す必要があります。

Client-Library のプログラミング言語およびプラットフォームのバージョンでコールバックがサポートされていない場合は、その言語およびプラットフォームの『Open Client/Server プログラマーズ・ガイド補足』にサポートされないことが記載されています。

## コールバック・ルーチンのインストール

コールバックが必要な場合は、実行時にそれをインストールするようにアプリケーションを作成してください。アプリケーションは ct_callback を呼び出すことによってコールバック・ルーチンをインストールし、コールバック・ルーチンにポインタを渡して、type パラメータによってそのタイプを指定します。

特定のタイプのコールバックは、コンテキストまたは接続レベルでインストールできます。接続が割り付けられると、親コンテキストからデフォルト・コールバックが選択されます。アプリケーションは、ct_callback を呼び出してこれらのデフォルト・コールバックを上書きし、接続レベルで新しいコールバックをインストールします。

## コールバック・イベントが発生する場合

ほとんどのタイプのコールバックでは、コールバック・イベントが発生すると次のようになります。

- 適切なレベルに適切なタイプのコールバックがある場合、コールバックが呼び出されます。
- 適切なレベルに適切なタイプのコールバックがない場合、コールバック・イベント情報は廃棄されます。

クライアント・メッセージ・コールバックは、このルールの例外です。インストールされているクライアント・メッセージ・コールバックを持っていない接続に対して、エラーまたは情報メッセージが生成される場合、Client-Library は、そのメッセージを廃棄するのではなく、接続の親コンテキストのクライアント・メッセージ・コールバックを呼び出します。コンテキストがインストールされているクライアント・メッセージ・コールバックを持っていない場合は、そのメッセージを廃棄します。

## コールバック・ルーチンの取得および置換

現在インストールされているコールバックに対するポインタを取得するには、パラメータ action を CS_GET に設定して ct_callback を呼び出してください。ct_callback は、*func に現在のコールバックのアドレスを設定します。アプリケーションは、後で再使用するためにこのアドレスを保存します。

コールバックのインストールを解除するには、action パラメータを CS_SET に設定し、func パラメータを NULL に設定して、ct_callback を呼び出してください。

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、ct_callback を呼び出して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。

## コールバックでの Client-Library 呼び出しの制限

表 2-2 に示すように、すべてのコールバック・ルーチンで、呼び出し可能な Client-Library ルーチンが制限されています。

**表 2-2：コールバックが呼び出し可能な Client-Library ルーチン**

| コールバックのタイプ | 呼び出し可能なルーチン | 呼び出す状況 |
|---|---|---|
| すべてのコールバック・ルーチン | ct_config | 情報のみを取得する。 |
| | ct_con_props | 情報を取得する。または CS_USERDATA プロパティのみを設定する。 |
| | ct_cmd_props | 情報のみを取得する。CS_USERDATA プロパティは、ct_cmd_alloc で割り付けられるコマンド構造体に設定できる。<br>CS_USERDATA プロパティは、ct_con_props(CS_EED_CMD) または ct_con_props(CS_NOTIF_CMD) を呼び出すことによって得られるコマンド構造体に設定できない。 |
| | ct_cancel (CS_CANCEL_ATTN) | |
| サーバ・メッセージ | ct_describe | ルーチンは、コールバックの ct_con_props(CS_EED_CMD) 呼び出しによって返されるコマンド構造体で呼び出す必要がある。<br>「拡張エラー・データ」(143 ページ ) を参照。 |
| ノーティフィケーション | ct_bind、ct_describe、ct_fetch、ct_get_data、ct_res_info(CS_NUMDATA) | ルーチンは、コールバックの ct_con_props(CS_NOTIF_CMD) 呼び出しによって返されるコマンド構造体で呼び出す必要がある。<br>このコマンド構造体を使用することによって、アプリケーションはノーティフィケーション・イベントに対応するパラメータ値を取得できる。「レジスタード・プロシージャ」(276 ページ ) を参照。 |
| 完了 | cs_objects(CS_SET)、ct_init、ct_exit、ct_poll、ct_setloginfo、ct_getloginfo を除く、すべての Client-Library または CS-library ルーチン | **注意** cs_objects(CS_SET) は、非同期に対して安全ではない。ct_init、ct_exit、ct_getloginfo は、システムレベルでメモリの割り付けまたは解放を行うものなので、使用してはならない。 |
| ディレクトリ | ct_ds_dropobj、ct_ds_objinfo | ディレクトリ・オブジェクトを削除または検査する。 |

## CS_PUBLIC によるコールバック宣言

アプリケーションの Client-Library コールバックと CS-Library コールバックはすべて、CS_PUBLIC を使用して宣言してください。Windows など一部のプラットフォームでは、コンパイラは生成されるコード内の関数に対して、複数ある呼び出し規則の中の 1 つを使用します。関数の呼び出し規則は、関数が呼び出されるときにマシン・レジスタとマシン・スタックがどのように操作されるかを設定します。コンパイラは異なる呼び出し規則について、それぞれ異なるマシン命令を生成します。CS_PUBLIC は ( 必要なすべてのコンパイラ・オプションとともに )、Client-Library から呼び出されるときと同じ呼び出し規則を使用してアプリケーションのコールバックがコンパイルされるようにします。

---

**注意**　コンパイラ・オプションについては、使用しているプラットフォーム用の『Open Client/Server プログラマーズ・ガイド補足』を参照してください。

---

多くのプラットフォームでは、CS_PUBLIC は関数の宣言に何も追加しないように定義されています。これらのプラットフォームでは、CS_PUBLIC を使用してコールバックを宣言したアプリケーションと CS_PUBLIC を省略したアプリケーションとの動作は異なりません。ただし、移植性に考慮し、どのプラットフォームについても CS_PUBLIC を使用してコールバックを宣言してください。

## クライアント・メッセージ・コールバック

アプリケーションは、Client-Library エラーおよび情報メッセージをインラインで、またはクライアント・メッセージ・コールバック・ルーチンを使用して処理します。

接続が割り付けられると、親コンテキストからデフォルト・クライアント・メッセージが選択されます。親コンテキストがクライアント・メッセージ・コールバックをインストールしていない場合、デフォルト・クライアント・メッセージ・コールバックなしで、接続は作成されます。

接続を割り付けると、アプリケーションは、次のことを行います。

- その接続に対して、異なるクライアント・メッセージ・コールバックをインストールします。

- ct_diag を呼び出して、その接続のインラインのメッセージ処理を初期化します。ct_diag は、その接続のすべてのメッセージ・コールバックのインストールを自動的に解除するので、注意してください。

クライアント・メッセージ・コールバックが接続またはその親コンテキストに対してインストールされておらず、インラインのメッセージ処理が使用できない場合、Client-Library はメッセージ情報を廃棄します。

Client-Library のプログラミング言語またはプラットフォームの特定バージョンでコールバックが実装されていない場合、アプリケーションは、ct_diag を使用してインラインで Client-Library メッセージ処理を行う必要があります。

接続がクライアント・メッセージ・コールバックを介して Client-Library メッセージを処理している場合、Client-Library がエラーまたは情報メッセージを生成するたびに、そのコールバックが呼び出されます。

---

**注意**　例外として、ほとんどのタイプのコールバック・ルーチンの中からメッセージが生成されるときには、Client-Library はクライアント・メッセージ・コールバックを呼び出しません。メッセージが完了コールバックの中から生成されるときに、Client-library は、クライアント・メッセージ・コールバックを呼び出します。つまり、完了コールバック以外のコールバック中に Client-Library ルーチンの障害が生じた場合、ルーチンは、CS_FAIL を返しますが、クライアント・メッセージ・コールバックをトリガすることはありません。

---

## 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ます。

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

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

各パラメータ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。この構造体については、「Client-Library と SQL 構造体」(81 ページ ) 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になる場合があるので、注意してください。

クライアント・メッセージ・コールバックは、CS_SUCCEED または CS_FAIL のどちらかを返す必要があります。

- CS_SUCCEED は、この接続で発生している処理を続行するように Client-Library に指示します。

  タイムアウト・エラーが発生したためにコールバックが呼び出された場合、CS_SUCCEED が返されることによって、Client-Library はタイムアウト時間が完全に過ぎるまで待機してからクライアント・メッセージ・コールバックを再度呼び出すようになります。コマンドがタイムアウトにならずに成功するか、クライアント・メッセージ・コールバックからの ct_cancel(CS_CANCEL_ATTN) 呼び出しに応答してサーバが現在のコマンドをキャンセルするまでこの動作は続きます。

  **注意**　場合によっては、サーバがクライアントの ct_cancel コマンドに応答できないことがあります。このような状況は、たとえば、サーバが非常に複雑なクエリの処理中で、割り込み不可能なステータスになっている場合に発生することがあります。

- CS_FAIL は、この接続で現在実行中のすべての処理を終了するように Client-Library に要求します。CS_FAIL が返されると、その接続は「dead」とマーク付けされるか、使用できない状態になります。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から再オープンする必要があります。

表 2-3 は、クライアント・メッセージ・コールバックから呼び出すことのできる Client-Library ルーチンの一覧です。

表 2-3 : クライアント・メッセージ・コールバックが呼び出し可能なルーチン

| 呼び出し可能なルーチン | 呼び出す状況 |
|---|---|
| ct_config | 情報のみを取得する。 |
| ct_con_props | 情報を取得する。または CS_USERDATA プロパティのみを設定する。 |
| ct_cmd_props | 情報を取得する。または CS_USERDATA プロパティのみを設定する。 |
| ct_cancel (CS_CANCEL_ATTN) | すべての状況。 |

ほとんどのアプリケーションでは、エラーの詳細を表示したり、それらのログをファイルに取るだけのクライアント・メッセージ・コールバックを使用します。ただし、一部のアプリケーションでは特定のエラーを認識して特定のアクションを実行するコールバックが必要です。「特定の Client-Library メッセージの処理」(92 ページ ) を参照してください。

## クライアント・メッセージ・コールバックの例

次に、クライアント・メッセージ・コールバックの例を示します。

```
/*
 ** ex_clientmsg_cb()
 **
 ** Type of function:
 **      Example program client message handler
 **
 ** Purpose:
 **      Installed as a callback into Open Client.
 **
 ** Returns:
 **      CS_SUCCEED
 **
 ** Side Effects:
 **      None
 */

CS_RETCODE CS_PUBLIC
 ex_clientmsg_cb(context, connection, errmsg)
 CS_CONTEXT     *context
 CS_CONNECTION  *connection;
 CS_CLIENTMSG   *errmsg;
```

```
{
    fprintf(EX_ERROR_OUT, "¥nOpen Client Message:¥n");
    fprintf(EX_ERROR_OUT, "Message number:
        LAYER = (%ld) ORIGIN = (%ld) ",
        CS_LAYER(errmsg->msgnumber),
        CS_ORIGIN(errmsg->msgnumber));
    fprintf(EX_ERROR_OUT, "SEVERITY = (%ld)
        NUMBER = (%ld)¥n",
        CS_SEVERITY(errmsg->msgnumber),
        CS_NUMBER(errmsg->msgnumber));
    fprintf(EX_ERROR_OUT, "Message String:%s¥n",
        errmsg->msgstring);
    if (errmsg->osstringlen > 0)
    {
        fprintf(EX_ERROR_OUT, "Operating System ¥
            Error:%s¥n", errmsg->osstring);
    }
return CS_SUCCEED;
}
```

## 完了コールバック

完了コールバックは、非同期ルーチンが完了したことをアプリケーションに通知します。

コンテキストまたは接続が非同期になるように定義すると、ネットワークに読み込みや書き込みを行うルーチンは、必要な I/O オペレーションを完了するまでブロックするのではなく、すぐに制御を戻すようになります。接続構造体の CS_NETIO プロパティの値によって、Client-Library ルーチンが非同期に動作するかどうかが決まります。詳細については、「ネットワーク I/O」(256 ページ ) を参照してください。

非同期接続の場合は、ネットワーク I/O を行う Client-Library ルーチンは、要求されたオペレーションを完了してから制御を戻すのではなく、すぐに CS_PENDING を返します。CS_NETIO が CS_ASYNC_IO である完全非同期アプリケーションでは、非同期オペレーションの完了をメインライン・コードに通知するための完了コールバックが必要です。

「非同期プログラミング」(12 ページ ) を参照してください。

## 完了コールバックの定義

完了コールバックは、次のように定義されます。

```
CS_RETCODE CS_PUBLIC completion_cb(connection, cmd,
                                   function,status)

CS_CONNECTION        *connection;
CS_COMMAND           *cmd;
CS_INT               function;
CS_RETCODE           status;
```

各パラメータの意味は次のとおりです。

- *connection* は、CS_CONNECTION 構造体へのポインタで、この構造体は I/O を実行するルーチンの接続を表します。
- *cmd* は、ルーチンの CS_COMMAND 構造体を指すポインタです。*cmd* は、NULL の場合もあります。
- *function* は、完了したルーチンを示します。表 2-4 (39 ページ ) は、*function* で使用可能な記号値の一覧です。

**表 2-4：完了コールバック関数パラメータの値**

| 値 | 意味 |
|---|---|
| BLK_DONE | blk_done が完了した。 |
| BLK_INIT | blk_init が完了した。 |
| BLK_ROWXFER | blk_rowxfer が完了した。 |
| BLK_SENDROW | blk_sendrow が完了した。 |
| BLK_SENDTEXT | blk_sendtext が完了した。 |
| BLK_TEXTXFER | blk_textxfer が完了した。 |
| CT_CANCEL | ct_cancel が完了した。 |
| CT_CLOSE | ct_close が完了した。 |
| CT_CONNECT | ct_connect が完了した。 |
| CT_DS_LOOKUP | ct_ds_lookup が完了した。 |
| CT_FETCH | ct_fetch が完了した。 |
| CT_GET_DATA | ct_get_data が完了した。 |
| CT_OPTIONS | ct_options が完了した。 |
| CT_RECVPASSTHRU | ct_recvpassthru が完了した。 |
| CT_RESULTS | ct_results が完了した。 |
| CT_SEND | ct_send が完了した。 |
| CT_SEND_DATA | ct_send_data が完了した。 |
| CT_SENDPASSTHRU | ct_sendpassthru が完了した。 |
| ユーザ定義の値：この値は、CT_USER_FUNC 以上でなくてはならない。 | ユーザ定義の関数が完了した。 |

- *status* は、完了したルーチンの完了ステータスです。この値は、同じ条件下の同期呼び出しによって返される値に相当します。*status* として可能な値については、*function* パラメータの値に対応する、各ルーチンのリファレンス・ページの「戻り値」を参照してください。

  アプリケーションが完了コールバックを呼び出すために ct_wakeup を呼び出す場合は、ct_wakeup の呼び出しは完了コールバックが受け取るステータス値を指定します。

完了コールバック・ルーチンは、cs_objects (CS_SET)、ct_init、ct_exit、ct_setloginfo、ct_getloginfo 以外の Client-Library または CS-Library ルーチンを呼び出します。cs_objects(CS_SET) は非同期動作に関して安全ではなく、ct_init、ct_exit、ct_setloginfo、ct_getloginfo はシステムレベルでメモリの割り付けと解除を行います。

完了コールバックが非同期 Client-Library ルーチンを呼び出す場合は、そのルーチン自体によって返される値を返す必要があります。この他の場合は、完了コールバックが返すことができる値に対する制限はありません。ただし、完了コールバックが成功した場合は CS_SUCCEED を返し、エラーが発生した場合は CS_FAIL を返すようにすることをおすすめします。

### 完了コールバックの例

次に、完了コールバックの例を示します。このコードは、Client-Library のサンプル・プログラム (*ex_alib.c* ファイル ) からの抜粋です。

```
/*
** ex_acompletion_cb()
**
** Type of function:
**       example async lib
**
** Purpose:
**      Installed as a callback into Open Client. It
**    will dispatch to the appropriate completion
**       processing routine based on async state.
**
**       Another approach to callback processing is to
**       have each completion routine install the
**       completion callback for the next step in
**       processing. We use one dispatch point to aid
**       in debugging the async processing (only need
**       to set one breakpoint).
**
** Returns:
**       Return of completion processing routine.
**
** Side Effects:
**      None
*/
CS_STATIC CS_RETCODE CS_PUBLIC
ex_acompletion_cb(connection, cmd, function, status)
CS_CONNECTION   *connection;
CS_COMMAND      *cmd;
CS_INT          function;
CS_RETCODE      status;
{
     CS_RETCODE retstat;
     ExAsync    *ex_async;
```

```
    /*
     ** Extract the user area out of the command
     ** handle.
     */
     retstat = ct_cmd_props(cmd, CS_GET, CS_USERDATA,
          &ex_async, CS_SIZEOF(ex_async), NULL);
     if (retstat != CS_SUCCEED)
     {
          return retstat;
     }

    fprintf(stdout, "¥nex_acompletion_cb:function ¥
          %ld Completed", function);

    /* Based on async state, do the right thing */
     switch ((int)ex_async->state)
     {
          case EX_ASEND:
          case EX_ACANCEL_CURRENT:
          retstat = ex_asend_comp(ex_async, connection,
               cmd, function, status);
          break;

         case EX_ARESULTS:
          retstat = ex_aresults_comp(ex_async,
               connection, cmd, function, status);
          break;

         case EX_AFETCH:
          retstat = ex_afetch_comp(ex_async,
               connection, cmd, function, status);
          break;

         case EX_ACANCEL_ALL:
          retstat = ex_adone_comp(ex_async, connection,
               cmd, function, status);
          break;

         default:
          ex_apanic("ex_acompletion_cb:unexpected ¥
               async state");
          break;
     }

 return retstat;
}
```

# ディレクトリ・コールバック

ct_ds_lookup ルーチンとアプリケーションのディレクトリ・コールバックにより、アプリケーションがディレクトリ・エントリの内容を調べるためのメカニズムが提供されます。

アプリケーションが ct_ds_lookup を呼び出してディレクトリ検索を開始すると、Client-Library はディレクトリから適切なエントリを検索して、各エントリごとに一度ずつディレクトリ・コールバックを呼び出します。コールバックは呼び出されるごとに、ディレクトリ・オブジェクト構造体を指すポインタを 1 つ受け取ります。各ディレクトリ・オブジェクト構造体はディレクトリ・エントリから読み込まれた情報のコピーを保持します。

Client-Library は、コールバックが CS_CONTINUE を返す間は、検索された各エントリごとに一度ずつディレクトリ・コールバックを呼び出します。コールバックが CS_SUCCEED を返すと、Client-Library はコールバックが受け取っていない残りのオブジェクトをすべて廃棄します。

ディレクトリ・コールバックは、Client-Library ルーチン ct_con_props、ct_config、ct_ds_objinfo、ct_ds_dropobj だけを呼び出すことができます。非同期接続では、アプリケーションが完了コールバックを使用して別の Client-Library ルーチンを呼び出します ( 表 2-2 (32 ページ ) を参照してください )。

## ディレクトリ・コールバックの定義

ディレクトリ・コールバックは次のように定義されます。

```
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;
```

各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す場合、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け取ります。*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。

## ディレクトリ検索結果の処理

ディレクトリ・コールバックは、ディレクトリ検索の結果を収集してオプションとして処理するために一般的に次のように実行します。

1 検索が成功したかどうか、そしてエントリが返されたかどうかを調べるために、*status* と *numentries* の値を調べます。
   - CS_SUCCEED の *status* 値は検索が成功したことを示します。
   - 0 より大きい *numentries* 値はエントリが見つかったことを示します。

2 ディレクトリ・オブジェクトへのポインタを保存するか、( 情報を取り出すために ct_ds_objinfo を使用して ) 保持する必要がある情報をコピーしてから、ct_ds_dropobj を使用してディレクトリ・オブジェクトのメモリを解放します。

3 次のいずれかの方法で Client-Library に制御を戻します。

  - 調べられていない残りのすべてのエントリを削除するように CS_SUCCEED を返します。
  - ディレクトリ検索によって返される次のオブジェクトを処理するために Client-Library がコールバック・ルーチンを再度呼び出すように、CS_CONTINUE を返します。

## コールバックの呼び出し手順

検索が成功した場合、Client-Library は調べる対象のエントリの総数として *numentries* を使用してディレクトリ・コールバックを呼び出します。検索によってエントリが見つからない場合は、*numentries* は 0 になります。検索によって 1 つまたは複数のエントリが見つかった場合は、*numentries* は現在のエントリを含んだ未調査のエントリの数を表します。

アプリケーションは、Client-Library がコールバックを呼び出すたびにコールバックから CS_CONTINUE を返すことによって、すべてのエントリを調べることができます。ct_ds_lookup は、次のいずれかの条件が満たされるまでコールバックを繰り返し呼び出します。

- コールバックが CS_SUCCEED を返す場合。
- コールバックが検索結果にディレクトリ・オブジェクトを受け取った場合。*numentries* が 0 または 1 であるときに、コールバックが CS_CONTINUE を返す場合は、ct_ds_lookup が完了する前にコールバックが再度呼び出されることはありません。
- コールバックが CS_CONTINUE または CS_SUCCEED 以外の値を返す場合は、現在の Client-Library の応答は CS_SUCCEED の場合と同じです。ただし、この動作は今後のバージョンでは変更されることがあります。今後のバージョンとの互換性を保つためには、アプリケーションはディレクトリ・コールバックから CS_CONTINUE または CS_SUCCEED だけを返すようにしてください。

非同期ネットワーク I/O が接続に対して有効である場合、ディレクトリ・コールバックの呼び出しはすべて Client-Library がアプリケーションの完了コールバックを呼び出す前に発生します。

同期ネットワーク I/O が接続に対して有効である場合は、ディレクトリ・コールバックの呼び出しはすべて ct_ds_lookup が制御を戻す前に発生します。

### ディレクトリ・コールバックの例

ディレクトリ・コールバックは ct_ds_lookup とともに使用されます。ディレクトリ・コールバックの例については、ct_ds_lookup のリファレンス・ページを参照してください。

## 暗号コールバック

Adaptive Server Enterprise と Open Server は、クライアントから要求されたときに暗号化されたパスワードによるハンドシェイクを使用します。

クライアント・アプリケーションは、パスワードの暗号化を可能にするために、ct_con_props を呼び出して CS_SEC_EXTENDED_ENCRYPTION プロパティまたは CS_SEC_ENCRYPTION プロパティを設定する必要があります。CS_SEC_EXTENDED_ENCRYPTION と CS_SEC_ENCRYPTION の両方が CS_TRUE に設定されている状態で Open Client アプリケーションでサーバにログインすると、サーバでは拡張パスワード暗号化が最優先で使用されます。

Client-Library のデフォルトの暗号ハンドラは、Adaptive Server Enterprise が必要とするパスワードの暗号化を実行します。これらのサーバに接続する単純なクライアント・アプリケーションの場合は暗号コールバックを必要としません。ただし、Adaptive Server Enterprise に対するゲートウェイとして動作する Client-Library アプリケーションの場合は、パスワードの暗号化を明示的に処理する必要があります。このようなアプリケーションは、サーバの暗号キーをクライアントに渡して、暗号化されたパスワードをサーバに送り返す暗号コールバック・ルーチンをインストールする必要があります。「ゲートウェイ・アプリケーションでのパスワード暗号化」(48 ページ ) を参照してください。

カスタマイズされたパスワード暗号化技術を使用して Open Server に接続する Client-Library アプリケーションの場合も、必要なパスワード暗号化を行うために暗号コールバック・ルーチンをインストールする必要があります。

パスワード暗号化のハンドシェイク処理については、「セキュリティ・ハンドシェイク：暗号化したパスワード」(317 ページ ) を参照してください。

---

**注意**　パスワードの暗号化をデータの暗号化と混同しないでください。暗号コールバックが暗号化するのはパスワードだけです。データの暗号化とは、その接続で送信されるすべてのコマンドと結果を暗号化することであり、外部のセキュリティ・サービス・プロバイダによって行われます。「セキュリティ機能」(290 ページ ) を参照してください。

---

## 暗号コールバックの定義

拡張パスワード暗号化および通常のパスワード暗号化の暗号コールバックのプロトタイプは、次のように定義されます。

**通常の暗号化パスワード**

```
CS_RETCODE CS_PUBLIC encrypt_cb(connection, pwd,
              pwdlen, key, keylen, buf, buflen, outlen)


CS_CONNECTION        *connection;
CS_BYTE              *pwd;
CS_INT               pwdlen;
CS_BYTE              *key;
CS_INT               keylen;
CS_BYTE              *buffer;
CS_INT               buflen;
CS_INT               *outlen;
```

各パラメータの意味は次のとおりです。

- *connection* は CS_CONNECTION 構造体を指すポインタで、サーバにログインしている接続を表しています。
- *pwd* は、暗号化されるユーザ・パスワードまたはリモート・サーバ・パスワードです。ユーザ・パスワードは CS_PASSWORD 接続プロパティの値と一致します。リモート・サーバのパスワードは ct_remote_pwd に渡される文字列に一致します。*pwd* 文字列は null で終了するとはかぎりません。
- *pwdlen* は、バイト単位のパスワードの長さです。
- *key* は、暗号コールバックがパスワードを暗号化するために使用するキーです。暗号キーは、リモート・サーバによって与えられます。

- *keylen* は、バイト単位の暗号キーの長さです。
- *buffer* は、バッファを指すポインタです。暗号コールバックはこのバッファへ暗号化されたパスワードを入れます。このバッファは、Client-Library により割り付けおよび解放されます。バッファの長さは *buflen* で指示されます。
- *buflen* は、**buffer* データ領域のバイト単位の長さです。
- *outlen* は、CS_INT を指すポインタです。暗号コールバックは、**outlen* を **buffer* にある暗号化されたパスワードの長さに設定する必要があります。

**拡張パスワード暗号化**

```
CS_RETCODE extended_encrypt_cb(
      CS_CONNECTION *connection,
      CS_BYTE       *pwd,
      CS_INT        pwdlen,
      CS_INT        *ciphersuite,
      CS_BYTE       *pubkey,
      CS_INT        pubkeylen,
      CS_VOID       *buffer,
      CS_INT        buflen,
      CS_INT        *outlen)
```

各パラメータの意味は次のとおりです。

- *connection* は CS_CONNECTION 構造体を指すポインタで、サーバにログインしている接続を表しています。
- *pwd* は、暗号化されるユーザ・パスワードまたはリモート・サーバ・パスワードです。ユーザ・パスワードは CS_PASSWORD 接続プロパティの値と一致します。リモート・サーバのパスワードは ct_remote_pwd に渡される文字列に一致します。*pwd* 文字列は null で終了するとはかぎりません。
- *pwdlen* は、バイト単位のパスワードの長さです。
- *ciphersuite* は、パスワードの暗号化に使用される ciphersuite を指すポインタです。このパラメータは、デフォルト暗号化では使用されません。
- *pubkey* は、パスワードの暗号化に使用されるパブリック・キーを指すポインタです。
- *pubkeylen* は、バイト単位のパブリック・キーの長さです。

- *buffer* は、バッファを指すポインタです。暗号コールバックはこのバッファへ暗号化されたパスワードを入れます。このバッファは、Client-Library により割り付けおよび解放されます。バッファの長さは *buflen* で指示されます。
- *buflen* は、**buffer* データ領域のバイト単位の長さです。
- *outlen* は、新たに受け入れられたパスワードの長さを格納する CS_INT を指すポインタです。暗号コールバックは、**outlen* を **buffer* にある暗号化されたパスワードの長さに設定する必要があります。

暗号コールバックは、パスワードが正常に暗号化されたことを示すために、CS_SUCCEED を返します。暗号コールバックが、CS_SUCCEED 以外の値を返した場合、Client-Library は、接続をアボートし、これにより ct_connect は CS_FAIL を返します。

### ゲートウェイ・アプリケーションでのパスワード暗号化

暗号化されたパスワードを処理するには、ゲートウェイ・アプリケーションは次のようにする必要があります。

- 暗号コールバック・ルーチンを使用します。
- ct_callback を呼び出して、コンテキスト・レベルでまたは特定の接続用に暗号コールバックをインストールします。
- ct_con_props を呼び出して、CS_SEC_EXTENDED_ENCRYPTION プロパティまたは CS_SEC_ENCRYPTION プロパティを CS_TRUE に設定します。

リモート・サーバに接続するためにゲートウェイが ct_connect を呼び出すと、次のような処理が行われます。

1. リモート・サーバは暗号キーを使用して応答し、Client-Library に暗号コールバックをトリガさせます。
2. 暗号コールバックはゲートウェイのクライアントにキーを渡します。
3. ゲートウェイのクライアントはパスワードを暗号化して、暗号コールバックに返します。
4. 暗号コールバックは、暗号化されたパスワードを **buffer* に入れ、**outlen* を設定して、ステータス・コードを Client-Library に返します。
   - コールバックが CS_SUCCEED を返すと、Client-Library は暗号化されたパスワードをリモート・サーバに送信します。

- コールバックが CS_FAIL を返した場合、Client-Library は接続処理をアボートし、これにより ct_connect は CS_FAIL を返します。

Client-Library は最初に一度だけ暗号コールバックを呼び出して、CS_PASSWORD によって定義されているパスワードを暗号化し、ct_remote_pwd によって定義されているそれぞれのリモート・サーバ・パスワードごとにもう一度暗号コールバックを呼び出します。

Adaptive Server Enterprise に対するゲートウェイは、暗号化されたリモート・パスワードが正しく処理されるように特別な手順を踏む必要があります。接続要求に対して暗号コールバックが初めて呼び出されたときに、ゲートウェイは次のようなアクションを実行しなければなりません。

1. ct_remote_pwd(CS_CLEAR) を使用してデフォルトのリモート・パスワードをクリアします。

   ct_connect は、ct_connect を呼び出す前にゲートウェイがリモート・パスワードを定義していない場合は、デフォルトのリモート・パスワードを作成します。ゲートウェイはこのデフォルトをクリアしなければなりません。

2. srv_negotiate を使用して、ゲートウェイのクライアントに対して暗号化されたローカル・パスワードとリモート・パスワードのチャレンジをします。

3. 暗号化された各リモート・パスワードごとに ct_remote_pwd を一度ずつ呼び出します。

4. 暗号化されたローカル・パスワードを **buffer* に入れ、**outlen* にその長さを設定します。

5. エラーが発生しなかった場合、CS_SUCCEED を返します。

以後のコールバックの呼び出しはいずれも、チャレンジに応答してゲートウェイのクライアントから読み込まれ、暗号化されたリモート・パスワードの 1 つを返す必要があります。

ゲートウェイは暗号キーを転送して、Server-Library 呼び出しを使用してクライアントの応答を読み込みます。詳細については、『Open Server Server-Library/C リファレンス・マニュアル』の srv_negotiate の項を参照してください。

「ネットワーク・セキュリティ・メカニズムの選択」(293 ページ ) を参照してください。

## ネゴシエーション・コールバック

Client-Library は、ネゴシエーション・コールバックを使用して、trusted ユーザ・セキュリティ・ハンドシェイクおよびチャレンジ／応答セキュリティ・ハンドシェイクの両方を処理します。

「セキュリティ機能」(290 ページ ) を参照してください。

### チャレンジ／応答セキュリティ・ハンドシェイク

サーバへのログイン中に、サーバがチャレンジを発行するとチャレンジ／応答セキュリティ・ハンドシェイクが発生します。クライアントはこのチャレンジに応答する必要があります。

接続は、ネゴシエーション・コールバックを使用して、その応答をチャレンジに渡します。そうするために、接続はネゴシエーション・コールバック・ルーチンをインストールします。接続時に、Client-Library がサーバ・チャレンジを受け取ると、Client-Library はネゴシエーション・コールバックをトリガします。

チャレンジ／応答セキュリティ・ハンドシェイクを使用する接続は、その CS_SEC_CHALLENGE プロパティまたは CS_SEC_APPDEFINED プロパティを CS_TRUE に設定してください。

サーバに接続するためにアプリケーションが ct_connect を呼び出す場合は、次のようになります。

1. サーバがチャレンジで応答した場合、Client-Library は接続のネゴシエーション・コールバック・ルーチンを呼び出します。
2. ネゴシエーション・コールバック・ルーチンは応答を生成して、CS_CONTINUE、CS_SUCCEED、CS_FAIL のいずれかを返します。
   - コールバック・ルーチンが CS_CONTINUE を返した場合は、Client-Library は再度ネゴシエーション・コールバックを呼び出して追加応答を受け取ります。
   - コールバック・ルーチンが CS_SUCCEED を返した場合は、Client-Library はサーバにその応答を送信します。
   - コールバックが CS_FAIL を返した場合、Client-Library は接続処理をアボートし、これにより ct_connect は CS_FAIL を返します。

## ネゴシエーション・コールバックの定義

ネゴシエーション・コールバックは、次のように定義されます。

```
CS_RETCODE CS_PUBLIC
     negotiation_cb(connection, inmsgid, outmsgid,
                    inbuffmt, inbuf,outbuffmt,
                    outbuf, outbufoutlen)

CS_CONNECTION       *connection;
CS_INT              inmsgid;
CS_INT              *outmsgid;
CS_DATAFMT          *inbuffmt;
CS_BYTE             *inbuf;
CS_DATAFMT          *outbuffmt;
CS_BYTE             *outbuf;
CS_INT              *outbufoutlen;
```

各パラメータの意味は次のとおりです。

- *connection* は CS_CONNECTION 構造体を指すポインタで、サーバにログインしている接続を表しています。
- *inmsgid* は、サーバが要求している情報のタイプです。*inmsgid* には次の値が使用されます。

| inmsgid の値 | 意味 |
|---|---|
| CS_MSG_GETLABELS | サーバがセキュリティ・ラベルを要求している。 |
| CS_USER_MSGID より小さい値 | サーバは、Sybase が定義した値を要求している。 |
| CS_USER_MSGID 以上で、CS_USER_MAX_MSGID 以下のユーザ定義の値 | Open Server アプリケーションは、アプリケーションが定義した値を要求している。ネゴシエーション・コールバックは *inmsgid* を解釈する必要がある。 |

- *outmsgid* は、ネゴシエーション・コールバックが返す情報のタイプです。次の表は、*outmsgid* に対する有効な値の一覧です。

| outmsgid の値 | 意味 |
|---|---|
| CS_MSG_LABELS | ネゴシエーション・コールバックが、セキュリティ・ラベルを返している。 |
| CS_USER_MSGID より小さい値 | コールバックが、Sybase が定義している値を返している。 |
| CS_USER_MSGID 以上で、CS_USER_MAX_MSGID 以下のユーザ定義の値 | コールバックが、アプリケーションが定義している値を返している。 |

- *inbuffmt* は、CS_DATAFMT 構造体を指すポインタです。ネゴシエーション・コールバックが、trusted ユーザ・ハンドシェイクを処理している場合、*inbuffmt* は NULL です。ネゴシエーション・コールバックがチャレンジ／応答ハンドシェイクを処理している場合、**inbuffmt* は、*inbuf* チャレンジ・キーを示しています。
- *inbuf* は、データ領域を指すポインタです。ネゴシエーション・コールバックが trusted ユーザ・ハンドシェイクを処理している場合、*inbuf* は NULL です。ネゴシエーション・コールバックがチャレンジ／応答ハンドシェイクを処理している場合、*inbuf* はチャレンジ・キーを指します。
- *inbuffmt* は、CS_DATAFMT 構造体を指すポインタです。ネゴシエーション・コールバックは、この CS_DATAFMT 構造体にセキュリティ・ラベルまたは返されている応答を入れます。

  Client-Library は、CS_DATAFMT 構造体の中に設定する必要があるフィールドを定義しません。
- *outbuf* は、バッファを指すポインタです。ネゴシエーション・コールバックは、このバッファにセキュリティ・ラベルまたは応答を入れます。このバッファは、Client-Library により割り付けおよび解放されます。バッファの長さは、*outbuffmt−>maxlength* で記述されます。
- *outbufoutlen* は、**outbuf* に入れたデータのバイト単位での長さです。

ネゴシエーション・コールバックは、CS_SUCCEED、CS_FAIL、または CS_CONTINUE を返す必要があります。

- コールバックが CS_CONTINUE を返した場合、Client-Library は再度ネゴシエーション・コールバックを呼び出して、追加のセキュリティ・ラベルまたは応答を生成します。
- コールバックが CS_SUCCEED を返した場合、Client-Library はサーバにセキュリティ・ラベルまたは応答を送信します。
- コールバックが CS_FAIL を返した場合、Client-Library は接続処理をアボートし、これにより ct_connect は CS_FAIL を返します。

## ノーティフィケーション・コールバック

レジスタード・プロシージャは、稼働中の Open Server 内で定義されてインストールされるプロシージャです。Client-Library アプリケーションはリモート・プロシージャ・コール・コマンドを使用して、レジスタード・プロシージャを実行し、また、他のアプリケーションまたはこのアプリケーション自体によるレジスタード・プロシージャの実行を「監視」します。

レジスタード・プロシージャの実行を監視するためには、Client-Library アプリケーションはホストの Open Server に接続する必要があります。クライアント・アプリケーションは Open Server の sp_regwatch システム・レジスタード・プロシージャをリモートから呼び出します。

レジスタード・プロシージャが実行されると、これを監視しているアプリケーションは、プロシージャ名と呼び出されたときの引数を含むノーティフィケーションを受け取ります。Client-Library はノーティフィケーションを受け取り (Open Server への接続を介して )、アプリケーションのノーティフィケーション・コールバック・ルーチンを呼び出します。

CS_ASYNC_NOTIFS プロパティは、ノーティフィケーション・コールバックをトリガする方法を決定する。詳細については、「非同期ノーティフィケーション」(237 ページ ) のこのプロパティについての説明を参照してください。

レジスタード・プロシージャが呼び出されたときの引数は、パラメータ結果セットとしてノーティフィケーション・コールバック内で取得できます。これらの引数を取得するために、アプリケーションは次のようにします。

- ct_con_props(CS_NOTIF_CMD) を呼び出して、パラメータ結果セットを持つコマンド構造体へのポインタを取得します。
- ct_res_info(CS_NUMDATA)、ct_describe、ct_bind、ct_fetch、ct_get_data を呼び出して、そのパラメータの記述、バインド、およびフェッチを行います。

「レジスタード・プロシージャ」(276 ページ ) を参照してください。

## ノーティフィケーション・コールバックの定義

ノーティフィケーション・コールバックは、次のように定義されます。

```
CS_RETCODE CS_PUBLIC notification_cb(conn, proc_name,
                                              namelen)

CS_CONNECTION    *conn;
CS_CHAR          *proc_name;
CS_INT           namelen;
```

各パラメータの意味は次のとおりです。

- *connection* は、ノーティフィケーションを受け取る CS_CONNECTION 構造体を指すポインタです。この CS_CONNECTION は、ここでノーティフィケーションを受け取る要求を送った CS_COMMAND の親接続です。
- *proc_name* は、実行されたレジスタード・プロシージャ名を指すポインタです。
- *namelen* は、**proc_name* のバイト単位での長さです。

ノーティフィケーション・コールバックは、CS_SUCCEED を返す必要があります。

表 2-5 (55 ページ) は、ノーティフィケーション・コールバックが呼び出す Client-Library ルーチンのリストです。

表 2-5：ノーティフィケーション・コールバックが呼び出し可能なルーチン

| 呼び出し可能なルーチン | 呼び出す状況 |
|---|---|
| ct_config | 情報のみを取得する。 |
| ct_con_props | 情報を取得する。または CS_USERDATA プロパティのみを設定する。 |
| ct_cmd_props | 情報のみを取得する。CS_USERDATA プロパティは、ct_cmd_alloc で割り付けられるコマンド構造体に設定できる。<br>**注意**　CS_USERDATA プロパティは、コールバックの ct_con_props(CS_NOTIF_CMD) 呼び出しによって取得されたコマンド構造体には設定できない。 |
| ct_cancel (CS_CANCEL_ATTN) | すべての状況。 |
| ct_bind、ct_describe、ct_fetch、ct_get_data、ct_res_info(CS_NUMDATA) | ノーティフィケーション・パラメータ値を取得する。ルーチンは、コールバックの ct_con_props(CS_NOTIF_CMD) 呼び出しによって返されるコマンド構造体で呼び出す必要がある。 |

### ノーティフィケーション・パラメータの取得

レジスタード・プロシージャを呼び出したパラメータの値は、ノーティフィケーション・コールバックで使用することができます。この値を得るには、アプリケーションは CS_NOTIF_CMD 接続プロパティとして保管されているコマンド構造体を取得する必要があります。このコマンド構造体を使用して、アプリケーションは、ct_res_info(CS_NUMDATA)、ct_describe、ct_fetch に対する通常の呼び出しを使用してパラメータ値を取得します。

「レジスタード・プロシージャ」(276 ページ ) を参照してください。

## セキュリティ・セッション・コールバック

Open Server ゲートウェイは、次のすべての条件を満たす場合にかぎり、セキュリティ・セッション・コールバックを必要とします。

- Open Server がゲートウェイである場合
- クライアントがネットワーク・ベースのユーザ認証を使用して接続することをゲートウェイが認めている場合

- ゲートウェイのクライアントとリモート・サーバ間でダイレクト・セキュリティ・セッションを確立することをゲートウェイが必要とする場合

上のどの条件も満たさない場合、Client-Library は適切なデフォルト・コールバックを提供します。

「ログイン認証サービスの要求」(295 ページ ) を参照してください。

## ダイレクト・セキュリティ・セッションの確立

「セキュリティ・セッション」は、クライアントとサーバが (DCE などの ) 外部セキュリティ・メカニズムと ( データ暗号化などの ) セキュリティ・サービスを使用することを認めたクライアント／サーバ接続です。

ゲートウェイ・アプリケーションでは、「ダイレクト・セキュリティ・セッション」がゲートウェイのクライアントとリモート・サーバ間で確立されます。ゲートウェイはセッションの確立中は仲介者として動作しますが、その後はゲートウェイはセキュリティ・セッションには関与しません。ダイレクト・セキュリティ・セッションは次のような場合に役立ちます。

- パケットごとのセキュリティ・サービスをサポートする完全なパススルー・ゲートウェイの場合

  完全なパススルー・ゲートウェイは、ダイレクト・セキュリティ・セッションを確立して、データの整合性やデータの機密性などのパケットごとのセキュリティ・サービスをサポートするとともに、関連するオーバヘッドを減少させます。たとえば、ゲートウェイがダイレクト・セキュリティ・セッションを使用しないでデータの機密性をサポートする場合、ゲートウェイを通過する各 TDS パケットの内容を受信時に復号化して、送信時に再度暗号化する必要があります。ゲートウェイでパケットの内容を調べる必要がない場合、これは不要なオーバヘッドです。ダイレクト・セキュリティ・セッションを使用すると、ゲートウェイ内部ではパケットごとのサービスは実行されません。

- 委任されたクライアント・クレデンシャルを使用できないゲートウェイの場合

  ゲートウェイのクライアントはゲートウェイに対して (CS_SEC_DELEGATION 接続プロパティを使用して ) セキュリティ・クレデンシャルを委任できません。または、セキュリティ・メカニズムがクレデンシャルの委任をサポートできません。この場合、ゲートウェイではゲートウェイのクライアントと同じユーザ名を使用してリモート・サーバに接続するように、ダイレクト・セキュリティ・セッションを設定する必要があります。

セキュリティ・セッション・コールバックを使用することによって、ゲートウェイはダイレクト・セキュリティ・セッションを設定できます。リモート・サーバに接続するときに、コールバック・ルーチンはリモート・サーバとゲートウェイのクライアント間で必要とされるハンドシェイクのための仲介者として動作します。ハンドシェイク処理の概要は次のとおりです。

1. ゲートウェイが ct_connect を呼び出すと、リモート・サーバは 1 つまたは複数のセキュリティ・セッション・メッセージを発行します。
2. リモート・サーバから送信されたそれぞれのセキュリティ・セッション・メッセージに対して、Client-Library はコールバックを呼び出し、リモート・サーバから送信されたセキュリティ・セッション情報をコールバックの入力パラメータとして渡します。
3. コールバックは、Server-Library ルーチン srv_negotiate(CS_SET, SRV_NEG_SECSESSION) を呼び出して、この情報をゲートウェイのクライアントに転送します。
4. 次に、コールバックはクライアントの応答を読み込み、それをコールバックの出力パラメータを使用して Client-Library に返します。
5. Client-Library はリモート・サーバに応答を転送します。

リモート・サーバが別のセキュリティ・セッション・メッセージを送信した場合は、この処理が繰り返されます。

## セキュリティ・セッション・コールバックの定義

セキュリティ・セッション・コールバックは次のように定義されます。

```
CS_RETCODE CS_PUBLIC
     secsession_cb (conn, numinputs, infmt, inbuf,
                    numoutputs, outfmt, outbuf, outlen)

CS_CONNECTION       *conn;
CS_INT              numinputs;
CS_DATAFMT          *infmt;
CS_BYTE             **inbuf;
CS_INT              *numoutputs;
CS_DATAFMT          *outfmt;
CS_BYTE             **outbuf;
CS_INT              *outlen;
```

各パラメータの意味は次のとおりです。

- *connection* は、ゲートウェイのリモート・サーバへの接続を制御する接続構造体を指すポインタです。
- *numinputs* は、セキュリティ・セッション・メッセージでリモート・サーバによって送信される入力パラメータの数です。
- *infmt* は、リモート・サーバによって送信される各入力パラメータを記述する CS_DATAFMT 構造体の配列のアドレスです。
- *inbuf* は、各入力パラメータのデータを保有するバッファを指す CS_BYTE* ポインタの配列のアドレスです。各バッファ *inbuf*[*i*] の長さは、*infmt*[*i*] ->*.maxlength* として与えられます。
- *numoutputs* は CS_INT のアドレスです。コールバックはクライアントによって送信される項目の数を ****numoutputs*** に返す必要があります。入力の場合は、****numoutputs*** は、***outfmt***、***outbuf***、***outlen*** 配列の長さを示します。
- *outfmt* は CS_DATAFMT 構造体の配列のアドレスです。コールバックでは、クライアントの応答の各項目の記述を、対応する CS_DATAFMT 構造体に入れる必要があります。****numoutputs*** の入力値はこの配列の長さを示します。
- *outbuf* は CS_BYTE* バッファの配列のアドレスです。コールバックでは、クライアントの応答のデータ項目を対応するバッファにコピーする必要があります。****numoutputs*** の入力値により、この配列の長さが指定されます。また、バッファ *i* のそれぞれに対して、*outfmt*[*i*]–>*maxlength* の入力値によって *outbuf*[*i*] が指すバッファに割り付けられた長さが指定されます。
- *outlen* は CS_INT の配列のアドレスです。コールバックは各バッファに書き込まれたバイト数を *outlen*[*i*] に入れます。

コールバックはセキュリティ・セッション・メッセージ・データを転送し、Server-Library 呼び出しを使用してクライアントの応答を読み込みます。詳細については、『Open Server Server-Library/C リファレンス・マニュアル』の srv_negotiate のリファレンス・ページを参照してください。

セキュリティ・セッション・コールバックは CS_SUCCEED または CS_FAIL を返します。コールバックが CS_FAIL を返すと、Client-Library は接続要求をアボートします。この他の戻り値は正しくありません。Client-Library はエラーを発生し接続要求をアボートすることによって応答します。

## サーバ・メッセージ・コールバック

アプリケーションは、サーバ・エラーおよび情報メッセージをインラインで、またはサーバ・メッセージのコールバック・ルーチンを使用して処理します。

接続が割り付けられると、親コンテキストからデフォルト・サーバ・メッセージ・コールバックが選択されます。親コンテキストがサーバ・メッセージ・コールバックをインストールしていない場合、デフォルト・サーバ・メッセージ・コールバックなしで接続が割り付けられます。

接続を割り付けると、アプリケーションは、次のことを行います。

- その接続に対して、異なるサーバ・メッセージ・コールバックをインストールします。
- ct_diag を呼び出して、その接続のインラインのメッセージ処理を初期化します。ct_diag は、その接続のすべてのメッセージ・コールバックのインストールを自動的に解除するので、注意してください。

サーバ・メッセージ・コールバックがインストールされておらず、インラインのメッセージ処理が使用できない場合、Client-Library は、サーバ・メッセージ情報を廃棄します。

Client-Library の特定のプログラミング言語とプラットフォームのバージョンで、コールバックが実装されていない場合、アプリケーションは、ct_diag を使用してインラインでサーバ・メッセージを処理しなくてはなりません。

サーバ・メッセージ・コールバックを使用して接続がサーバ・メッセージを処理している場合、サーバ・メッセージを受け取るたびに、そのコールバックが呼び出されます。

## 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ます。

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

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

各パラメータ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。
- *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。この構造体の詳細については、「CS_SERVERMSG 構造体」(103 ページ ) 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になる場合があるので、注意してください。
- サーバ・メッセージ・コールバックは CS_SUCCEED を返す必要があります。

**表 2-6：サーバ・メッセージ・コールバックが呼び出し可能なルーチン**

| 呼び出し可能なルーチン | 呼び出す状況 |
|---|---|
| ct_config | 情報のみを取得する。 |
| ct_con_props | 情報を取得する。または CS_USERDATA プロパティのみを設定する。 |
| ct_cmd_props | 情報のみを取得する。CS_USERDATA プロパティは、ct_cmd_alloc で割り付けられるコマンド構造体に設定できる。<br>CS_USERDATA プロパティは、コールバックの ct_con_props(CS_EED_CMD) によって取得されたコマンド構造体には設定できない。 |
| ct_cancel (CS_CANCEL_ATTN) | すべての状況。 |
| ct_bind、ct_describe、ct_fetch、ct_get_data、ct_res_info | ルーチンは、コールバックの ct_con_props(CS_EED_CMD) LAN によって返されるコマンド構造体で呼び出す必要がある。<br>サーバ・メッセージ・コールバックは、エラー・データが有効な間だけ、つまり ct_fetch が、CS_END_DATA を返すまで、これらのルーチンを呼び出す。<br>「拡張エラー・データ」(143 ページ ) を参照。 |

**警告！** ct_poll を、Client-Library コールバック関数、またはそれ以外のシステム割り込みレベルで実行可能な関数の中から呼び出さないでください。ct_poll をシステム割り込みレベルで呼び出すと、Open Client/Server 内部リソースが破壊され、アプリケーション内で予定外の再帰が発生します。

## サーバ・メッセージ・コールバックの例

次に、サーバ・メッセージ・コールバックの例を示します。

```
/*
 ** ex_servermsg_cb()
 **
 ** Type of function:
 **      Example program server message handler
 **
 ** Purpose:
 **      Installed as a callback into Open Client.
 **
```

```
** Returns:
**      CS_SUCCEED
**
** Side Effects:
**      None
*/
CS_RETCODE CS_PUBLIC
ex_servermsg_cb(context, connection, srvmsg)
CS_CONTEXT        *connection;
CS_CONNECTION     *cmd;
CS_SERVERMSG      *srvmsg;
{
     fprintf(EX_ERROR_OUT, "¥nServer message:¥n");
     fprintf(EX_ERROR_OUT, "Message number:%ld, ¥
          Severity %ld, ", srvmsg->msgnumber,
          srvmsg->severity);
     fprintf(EX_ERROR_OUT, "State %ld, Line %ld",
          srvmsg->state, srvmsg->line);

    if (srvmsg->svrnlen > 0)
     {
          fprintf(EX_ERROR_OUT, "¥nServer '%s'",
               srvmsg->svrname);
     }

    if (srvmsg->proclen > 0)
     {
          fprintf(EX_ERROR_OUT, " Procedure '%s'",
               srvmsg->proc);
     }

    fprintf(EX_ERROR_OUT, "¥nMessage String:%s",
          srvmsg->text);

    return CS_SUCCEED;
}
```

## 特定のメッセージの処理

アプリケーションによっては、プログラマが特定のメッセージ番号について特別な処理を作成する必要がある場合があります。

たとえば、メッセージが情報メッセージであってエラー・メッセージではない場合は、アプリケーションがエンド・ユーザにそのメッセージを表示しないようにする必要がある場合があります。次の例は、メッセージ 5701、5703、5704 を表示しないようにするサーバ・メッセージ・コールバックからの抜粋です。Adaptive Server Enterprise は接続がオープンされると常に 5701 メッセージを送信し、他の 2 つのメッセージを送信することもあります。Adaptive Server Enterprise も use database コマンドが成功するごとに 5701 メッセージを送信します。エンド・ユーザによっては、このようなメッセージを表示しないようにする必要がある場合があります。次に示すコードがサーバ・メッセージ・コールバックの先頭にあると、これらのメッセージ番号は無視されます。

```
/*
 ** Ignore these Server messages:
 **  5701 (changed database),
 **  5703 (changed language),
 **  or 5704 (changed client character set)
 */
 if (srvmsg->msgnumber == 5701
     || srvmsg->msgnumber == 5703
     || srvmsg->msgnumber == 5704)
 {
   return CS_SUCCEED;
 }
```

このコードは Adaptive Server Enterprise 専用です。Open Server ゲートウェイや Open Server のカスタム・アプリケーションなどの別のタイプのサーバに接続される場合、これらのメッセージ番号はまったく別のことを意味することもあります。

## シグナル・コールバック

シグナル・コールバックは、プロセスが UNIX プラットフォーム上のシグナルを受け取るたびに呼び出されます。

UNIX プラットフォームでは、Client-Library はシグナル駆動型 I/O を使用してネットワークと通信します。これらのプラットフォームでは、アプリケーションがシグナルを処理する場合は、アプリケーションはそのシグナルが Client-Library 以外の処理に関係するものであっても、Client-Library を使用してシグナル・ハンドラをインストールしなければなりません。シグナル・ハンドラをインストールするには、システム・コールを使用するのではなく ct_callback を呼び出します。シグナル・ハンドラをインストールするシステム・コールは、Client-Library のシグナル・ハンドラを上書きします。このような場合、Client-Library の動作は予測不可能です。

Client-Library を Open Server ゲートウェイで使用する場合は、Server-Library ルーチンを使用してシグナル・ハンドラをインストールしてください。

Client-Library がシグナルを受け取ると、Client-Library のシグナル・ハンドラは次のタスクを実行します。

- 必要な Client-Library 内部処理をすべて実行します。
- 適切なユーザ定義シグナル・コールバックがあれば、それを呼び出します。

## シグナル・コールバックの定義

オペレーティング・システムの仕様に従ってシグナル・コールバックを定義してください。

シグナル・コールバックを定義してインストールするアプリケーションには、適切なオペレーティング・システム・ヘッダ・ファイル ( ほとんどの UNIX プラットフォームでは *sys/signal.h*) をインクルードしてください。

## シグナル・コールバックのインストール

シグナル・コールバックは、コンテキスト・レベルでのみインストールできます。シグナル・コールバックは、定義されている定数 CS_SIGNAL_CB にシグナル番号を加えることにより識別されます。

次のルーチンは、シグナル・コールバックをインストールする方法を示しています。

```
/*
 ** INSTALLSIGNALCB
 **
 ** This routine installs a signal callback for the
 ** specified signal
 **
 ** Parameters:
 **      cp   Context handle
 **      signo   Signal number
 **      signalhandler   Signal handler to install
 **
 ** Returns:
 **      CS_SUCCEED   Signal handler was installed
 **                   successfully
 **      CS_FAIL      An error was detected while
 **                   installing the signal handler
 */
 CS_RETCODE  installsignalcb(cp, signo, signalhandler)
 CS_CONTEXT  *cp;
 CS_INT      signo;
 CS_VOID     *signalhandler;
 {
      CS_INT        adjustedsigno;
      CS_RETCODE    ret;

     /*
      ** Add the signal number to the CS_SIGNAL_CB
      ** define to indicate the signal number that this
      ** handler is being installed for.
      */
      adjustedsigno = CS_SIGNAL_CB + signo;

     ret = ct_callback(cp, (CS_CONNECTION *)NULL,
           CS_SET, adjustedsigno, signalhandler);

     return (ret);
 }
```

## SSL 検証コールバック

SSL (Secure Socket Layer) 検証コールバックは、SSL ハンドシェイクを傍受して、SSL 検証チェックを無効にします。SSL 検証コールバックが必要になるのは、Client-Library アプリケーションで SSL 検証チェックを無効にする必要がある場合だけです。

たとえば、ct_con_props(CS_SET, CS_SERVERADDR) を使用し、サーバのアドレスを *hostname port ssl* に設定して、SSL 接続を行う場合などです。

ct_connect に渡された *server_name* パラメータがサーバの証明書の共通名と一致しない場合、SSL 検証は失敗します。このチェックを無効にするには、SSL 検証コールバックを使用します。

## SSL 検証コールバックの定義

SSL 検証コールバックは、次のように定義されます。

```
CS_RETCODE CS_PUBLIC
validate_srvname_cb(CS_VOID *userdata, CS_SSLCERT *certptr,
        CS_INT certcount, CS_INT valid)
```

各パラメータの意味は次のとおりです。

- *userdata* は、接続構造体の CS_USERDATA を指します。
- *certptr* は、CS_SSLCERT 構造体の配列を指すポインタです。
- *certcount* は、配列のエントリ数を示します。
- *valid* は SSL 検証チェックにより決定される値です。*valid* には次の値のいずれかを指定できます。

| valid の値 | 意味 |
|---|---|
| CS_SSL_VALID_CERT | 有効な証明書である。 |
| CS_SSL_INVALID_BADCHAIN | 証明書チェーンが無効である。 |
| CS_SSL_INVALID_EXPCERT | チェーンにある証明書の有効期限が切れている。 |
| CS_SSL_INVALID_INCOMPLETE | 証明書チェーンが自己署名ルート証明書で終わっていない。 |
| CS_SSL_INVALID_UNKNOWN | 認識できない理由により、SSL 検証チェックが失敗した。 |
| CS_SSL_INVALID_UNTRUSTED | 証明書チェーンに、信頼された証明書が含まれていない。 |
| CS_SSL_INVALID_MISSINGNAME | 証明書に共通名がない。 |
| CS_SSL_INVALID_MISMATCHNAME | 共通名がサーバ名と一致していない。 |

### SSL 検証コールバックの例

次に、SSL 検証コールバックの例を示します。

```
CS_RETCODE CS_PUBLIC
validate_srvname_cb(CS_VOID *userdata, CS_SSLCERT *certptr,
    CS_INT certcount, CS_INT valid)
{
    if (valid == CS_SSL_INVALID_MISMATCHNAME)
    {
          return CS_SSL_VALID_CERT;
    }
    else
    {
          return valid;
    }
}
```

## 機能

クライアント／サーバ接続がサポートする機能を示します。特に、アプリケーションが特定の接続について送信できる要求のタイプと、サーバが特定の接続について返すことができるサーバ応答のタイプについて説明します。

### ワイド・テーブルと大きなページ・サイズ

Open Client と Open Server では、クライアント・アプリケーションは、Adaptive Server Enterprise でサポートされているワイド・データとカラム数の多いデータ (255 バイトを超えるカラムと、255 カラムを超えるテーブル ) を送受信できます。

**注意**　12.5 より前のバージョンでコンパイルした Client-Library アプリケーションは、大きいバイト制限を有効にするために 12.5 以降のバージョンで再コンパイルする必要があります。

## ページ・サイズ

Open Client と Open Server は、2K、4K、8K、16K の論理ページ・サイズをサポートしています。Open Client と Open Server は、Bulk-Library (blklib) ルーチンを使用してこれらのページにデータを取り込みます。

表 2-7 に、バルク・ライブラリの定数とその値を示します。

**表 2-7 : ページ・サイズの値**

| blk_pagesize | blk_maxdatarow | blk_maxcolsize | blk_maxcolno | blk_boundary |
|---|---|---|---|---|
| 2K | 1,962 | 1,960 | 1,962 | 1,960 |
| 4K | 4,010 | 4,008 | 4,010 | 4,008 |
| 8K | 8,106 | 8,104 | 8,106 | 8,104 |
| 16K | 16,298 | 16,296 | 16,298 | 16,298 |

ページ・サイズの上限が大きくなったことにより、テーブルのタイプによっては使用できるカラム数が増えました。上限は次のとおりです。

- 全ページ・ロック (APL) テーブルとデータオンリー・ロック (DOL) テーブルの固定長カラムは 1,024 まで
- APL テーブルの可変長カラムは 254 まで
- DOL テーブルの可変長カラムは 1,024 まで

### 互換性

次の場合に、ワイド・データとカラム数の増加が自動的に有効になります。

- クライアントが CS_VERSION_125 以降に設定されている場合
- クライアントが Open Client Server 12.5 以降にリンクしている場合
- 接続先の Adaptive Server Enterprise が、ワイド・テーブルを処理する機能を備えている場合。Adaptive Server Enterprise のバージョンを調べるには以下のように入力します。

```
1> select @@version
2> go
```

Open Client と Open Server 12.5 以降の blklib がバージョン 12.5 以降の bcp アプリケーションにリンクしており、そのアプリケーションが 12.5 より前の Adaptive Server Enterprise と通信している場合、bcp ユーティリティでは Adaptive Server Enterprise のページ・サイズが 2K であると想定されます。

blklib が、バージョン 12.5 より前のユーティリティで構築した bcp アプリケーションにリンクしている場合、サイズの大きいページのコピーはサポートされません。

## ワイド・テーブル

Open Client と Open Server では、255 カラム以上のテーブルと、256 バイト以上のカラム・サイズまたは 256 バイト以上のバイナリ・データがサポートされています。

### 機能

ワイド・テーブルをサポートするために、クライアントはサーバに、機能パケットとログイン・パケットを送信します。使用できる ct_capability パラメータとしては、次のものがあります。

- CS_WIDETABLE － クライアントがより大きいデータ・テーブルを受信する機能を持っていることをサーバに示すために、クライアントからサーバに送信される要求機能
- CS_NOWIDETABLE － この特定の接続に対して、ワイド・テーブルのサポートを無効にするようサーバに要求するため、クライアントからサーバに送信される応答機能

アプリケーションのバージョンが CS_VERSION_125 以降に設定されている場合、Client-Library は常にサーバに CS_WIDETABLE 機能を送信します。アプリケーションは要求機能を制御することはできません。ただし、接続の確立前に CS_NOWIDETABLE 応答機能を設定することで、ワイド・テーブル機能を無効にすることをサーバに要求します。

ct_capability の構文は次のとおりです。

```
CS_RETCODE ct_capability (connection, action, type,
         capability, value)
CS_CONNECTION   *connection;
CS_INT          action;
CS_INT          type;
CS_INT          capability;
CS_VOID         *value;
```

ここで、*type* の値は CS_WIDETABLE または CS_NOWIDETABLE です。

ワイド・テーブルをサポートしないようにする場合は、CS_NOWIDETABLE 応答機能を設定してから、ct_connect ルーチンを呼び出します。これは、サーバに接続する前に実行してください。

```
...
CS_BOOL     boolv = CS_TRUE
...
retcode = ct_capability (*conn_ptr, CS_SET,
            CS_CAP_RESPONSE, CS_NOWIDETABLES, &boolv);
...
```

CS_CURSOR_DECLARE を指定した ct_dynamic() は、CS_PREPARE、CS_EXECUTE、CS_EXEC_IMMEDIATE フラグをサポートしており、Adaptive Server Enterprise 12.5 の 1,024 カラム制限を参照する動的 SQL 文の準備と実行ができます。

ct_param() を使用すると、動的 SQL 文に 1,024 個の引数を渡すことができます。

### アプリケーション・プログラムでの変更

取得するカラム・データが CS_MAX_CHAR (256 文字または 256 バイナリ・データ ) を超えている場合は、CS_DATAFMT 構造体フィールドの datafmt.maxlength 定義を編集し、取得するデータの最大長 ( バイト数 ) を指定する必要があります。これを行わないと、トランケート・エラーが発生します。

クライアント・プログラムでこれより長いカラムを使用する可能性がある場合は、アプリケーション・プログラムでカラム配列サイズを変更してください。

たとえば、アプリケーションで幅 300 バイトのカラムを使用する場合、そのカラムの所定の位置に CS_CHAR col1[300] と記述する必要があります。カラムの長さが 255 バイトを超える場合は、RPC アプリケーション用の CS_DATAFMT 構造体の maxlength パラメータに、適切な文字長のデータ型を割り当ててください。CS_DATAFMT パラメータは次のように指定することをおすすめします。

```
datafmt.datatype = CS_LONGCHAR_TYPE
datafmt.maxlength = sizeof(col1)
```

次の例は、pubs2 データベースを使用する小さな ctlib プログラムです。

1 *authors* テーブルに変更を加え、*varchar(500)* を宣言する「comment」カラムを追加します。

```
1>alter table authors add comment varchar(500) null
2>go
```

2 次のようにテーブル内の新しいカラムを更新します。

```
1>update authors set comment = replicate
(substring(state,1,1), 500)
2>go
/* This SQL command will update the comment column with
a replicate of
500 times the first letter of the state for each row.*/
```

3 *example.h* ファイルを修正し、「新しい制限」機能を設定します。

```
#define EX_CTLIB_VERSION CS_VERSION_155
```

4. *exutils.h* ファイルを更新し、MAX_CHAR_BUF を 16,384 (16K) にリセットします。
5. 15.7 ヘッダとライブラリを使用して、ctlib を再コンパイルおよびリンクします。
6. Adaptive Server Enterprise バージョン 12.5 以降の *X*k ページ・サイズのサーバで実行し、テストします。

   CS_VERSION_157 を設定した場合、次のように表示されます ( 表示されるのは最後の 2 行のみです )。

```
Heather McBadden 95688 CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
Anne Ringer 84152 UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
```

7. *example.h* ファイルを更新し、ctlib を CS_VERSION_120 に再設定します。*OCS-15_0* ヘッダとライブラリを使用し、再コンパイルしてリンクを設定します。

   **注意**　先に CS_VERSION_157 を設定しないで同じプログラムを実行した場合、Adaptive Server Enterprise バージョン 12.5 以降と OCS 15.7 ライブラリを使用していても、コメント・カラムの先頭から 255 バイトのみが取得され、ワイド・カラムは取得できません。

   Open Client メッセージは、次のとおりです。

```
Message number: LAYER = (1) ORIGIN = (4) SEVERITY = (1) NUMBER = (132)
Message String: ct_fetch(): user api layer: internal common library
error: The bind of result set item 4 resulted in truncation.
Error on row 21.
Heather McBadden 95688 CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

Open Client メッセージは、次のとおりです。

```
Message number: LAYER = (1) ORIGIN = (4) SEVERITY = (1) NUMBER = (132)
Message String: ct_fetch(): user api layer: internal common library
error: The bind of result set item 4 resulted in truncation.
Error on row 22.
Anne Ringer 84152 UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
```

### ワイド・テーブルの互換性

次の場合は、ワイド・テーブルが自動的にサポートされます。

- クライアントが CS_VERSION_125 以降に設定されている場合
- クライアントが Open Client Server 12.5 以降にリンクしている場合
- 接続先の Adaptive Server Enterprise が、ワイド・テーブルを処理する機能を備えている場合。

Client-Library アプリケーションのバージョン文字列が CS_VERSION_125 以降に設定されていない場合、Client-Library アプリケーションが Open Client と Open Server 12.5 以降にリンクしていても、アプリケーションは制限の拡張をサポートしないため、動作は変わりません。

Open Client と Open Server バージョン 15.0、15.5 または 15.7 が、12.5 より前の Adaptive Server Enterprise に接続している場合、ワイド・テーブルをサポートしていないことを示す 0 の機能ビットがサーバから返されます。この場合も接続は確立されますが、動作は変わりません。

バージョン 12.5 より前の Open Client と Open Server が Adaptive Server Enterprise 15.0 に接続した場合も、制限の拡張は適用されません。ただし、古いバージョンのクライアントにワイド・テーブルを送信する必要があると Adaptive Server Enterprise が判断した場合は、データはトランケートされて送信されます。

---

**注意**　Adaptive Server Enterprise バージョン 11.0 以降では、マスク長が 7 バイトを超える場合は常にマスク長として 0 を返します。接続要求が機能マスクとして 0 を受信すると、以下のエラー・メッセージが表示されます。

```
ct_connect(): protocol specific layer: external error:
"This server does not accept new larger cpability mask,
the original cap mask will be used."
```

この場合、制限の拡張は適用されません。

---

### CS_RES_NOXNLMETADATA 応答機能

CS_RES_NOXNLMETADATA 応答機能は、サーバ・アプリケーションがクライアント・アプリケーションに送信する情報のタイプと構造を最適化することによって、サーバ・アプリケーションとクライアント・アプリケーションのパフォーマンスを向上させます。この情報が不要である場合には、CS_RES_NOXNLMETADATA を使用してメタデータ ( カラム・ラベル、カタログ名、スキーマ名、テーブル名など ) を送信しないようサーバに伝えることができます。

デフォルトでは、バージョンが CS_VERSION_125、CS_VERSION_150、CS_VERSION_155 に設定されているアプリケーションでは CS_RES_NOXNLMETADATA は有効です。

## unichar データ型

Open Client と Open Server の unichar では 2 バイト文字をサポートしており、多言語クライアント・アプリケーションがサポートされ、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 15.0 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 の 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、新しい 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 2-8 に、CS_DATAFMT ビットマスク・フィールドを示します。

**表 2-8：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 2-9：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、ct_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、スペースが保持されます。さらに、XML では、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。

## 機能および接続の TDS レベル

Sybase のクライアントとサーバは、Tabular Data Stream™ (TDS) プロトコルを使用して通信します。TDS のバージョンが異なると、サポートされる機能も異なります。たとえば、リモート・プロシージャ・コール (RPC) をサポートするのは TDS のバージョン 4.0 以降です。

TDS のバージョン・レベルは、接続が確立されたときに決定されます。サーバに接続するためにアプリケーションが ct_connect を呼び出すと、Client-Library はサーバに適切な TDS レベルを示します。サーバがこの TDS レベルをサポートできない場合は、Client-Library とネゴシエートして使用可能な TDS レベルを探します。

---

**注意**　jConnect は TDS バージョンをネゴシエートしません。サーバが TDS 5.0 をサポートしていない場合、jConnect は接続を終了します。

---

機能は、接続を介して送信できるクライアント要求とサーバ応答を記述します。デフォルトでは、機能は TDS のバージョン・レベルに基づきますが、クライアント・アプリケーションは応答機能をさらに制限するように、サーバは要求機能をさらに制限するようにコーディングできます。

Client-Library が ct_capability を呼び出すときは、次のことが行われます。

1. 接続をオープンする前に、接続構造体を設定して、サーバに対して接続について特定のタイプの応答を送信しないように通知します。
2. 接続をオープンした後に、接続が特定のタイプの要求または応答をサポートするかどうか調べます。

Open Server アプリケーションが機能を設定および取得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。

機能には、次の 2 つのタイプがあります。

- CS_CAP_REQUEST 機能 ( 要求機能 ) : サーバ接続上で送ることができるクライアント要求のタイプを記述します。
- CS_CAP_RESPONSE 機能 ( 応答機能 ) : 接続が受け取りを望まないサーバ応答のタイプを記述します。

機能のリストについては、ct_capability のリファレンス・ページを参照してください。

## 機能の設定および取得

ct_connect を呼び出す前に、アプリケーションは次のことを行います。

- 要求機能または応答機能を取得して、接続の現在の TDS バージョン・レベルで、通常サポートされている要求機能および応答機能を調べます。接続の TDS レベルは、デフォルトで、アプリケーションが ct_init への呼び出しで要求したバージョン・レベルになります。アプリケーションでは、*property* を CS_TDS_VERSION に設定して ct_con_props を呼び出すことにより、接続の TDS レベルを変更できます (「TDS バージョン」(266 ページ ) を参照 )。
- 応答機能を設定して、接続が特定タイプの応答の受け取りを望まないことを示します。たとえば、アプリケーションは、接続の TDS_RES_NOEED 機能を CS_TRUE に設定して、接続で拡張エラー・データの受け取りが要求されないことを示します。

接続をオープンすると、アプリケーションは次のことを行います。

- 要求機能を取得して、接続がサポートするタイプの要求を知ることができます。
- 応答機能を取得して、以前に指定した応答タイプを接続ではサーバが使用しないことに同意したかどうかを知ることができます。

### 複数機能の設定および取得

ゲートウェイ・アプリケーションでは、1 つのタイプ・カテゴリーのすべての機能を、ct_capability の 1 回の呼び出しで設定または取得することが必要な場合がよくあります。このことを行うために、アプリケーションは、次のようにして ct_capability を呼び出します。

- 対象のタイプ・カテゴリーを *type* として設定
- CS_ALL_CAPS を *capability* に設定
- CS_CAP_TYPE 構造体を *value* に設定

Client-Library は、次のマクロを提供し、アプリケーションが CS_CAP_TYPE 構造体のビットを設定、解除、およびテストできるようにします。

- CS_CLR_CAPMASK(*mask, capability*) － *capability* で指定されているビットをクリアすることにより、CS_CAP_TYPE 構造体 *mask* を修正する。
- CS_SET_CAPMASK(*mask, capability*) － *capability* で指定されているビットを設定することにより、CS_CAP_TYPE 構造体 *mask* を修正する。
- CS_TST_CAPMASK(*mask, capability*) － *capability* で指定されているビットが CS_CAP_TYPE *mask* に含まれるかどうかを判別する。

## Client-Library と SQL 構造体

この項では、Client-Library 構造体と SQL 構造体の概要を説明します。

# 公開された構造体と隠し構造体

Client-Library 構造体は、2 つのカテゴリーに分けられます。内部が文書化されていない「隠し構造体 (hidden structure)」と、内部が文書化されている「公開された構造体 (exposed structure)」です。

## 公開された構造体

公開された構造体は、Client-Library がアプリケーションと情報を交換し合うための方法を提供します。一般に、アプリケーションは、構造体をパラメータとして Client-Library ルーチンへ渡す前に、公開された構造体のフィールドを設定します。Client-Library ルーチンを呼び出した後で、公開された構造体のフィールドの値を取得します。

公開された構造体には次のものがあります。

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

これらの公開された構造体については、後で説明します。

## 隠し構造体

Client-Library は、さまざまな内部タスクを管理するために、隠し構造体を使用します。

Client-Library アプリケーションは、隠し構造体の内部に直接アクセスすることはできません。アプリケーションは、隠し構造体の割り付け、操作、割り付けの解除を行うために、Client-Library ルーチンを呼び出す必要があります。

隠し構造体には、次のようなものがあります。

- CS_BLKDESC ー Client-Library と Server-Library のバルク・コピー・ルーチンで使用される制御構造体。
- CS_CAP_TYPE ー 機能情報を保管するために使用する。
- CS_COMMAND ー コマンドとプロセス結果を送信するために使用する。

- CS_CONNECTION － 個別クライアント／サーバ接続を定義する。
- CS_CONTEXT － Client-Library プログラミング・コンテキストを定義する。
- 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_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る。

表 2-10 に、隠し構造体の割り付け、操作、および割り付けの解除を行うルーチンとマクロの一覧を示します。

**表 2-10：隠し構造体を操作するルーチン**

| 構造体 | ルーチン | 参照先 |
| --- | --- | --- |
| CS_BLKDESC | blk_alloc、blk_drop | 『Open Client/Server Common Libraries リファレンス・マニュアル』。 |
| CS_CAP_TYPE | CS_CLR_CAPMASK、CS_SET_CAPMASK、CS_TST_CAPMASK | 「複数機能の設定および取得」(81 ページ )。 |
| CS_COMMAND | ct_cmd_alloc、ct_cmd_props、ct_cmd_drop | 個々のルーチンのリファレンス・ページ。 |
| CS_CONNECTION | ct_con_alloc、ct_con_props、ct_con_drop | 個々のルーチンのリファレンス・ページ。 |
| CS_CONTEXT | cs_ctx_alloc、ct_config、cs_config cs_ctx_drop | 個々のルーチンのリファレンス・ページ。<br>CS-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。 |
| CS_LOCALE | cs_loc_alloc、cs_locale、cs_loc_drop | 「国際化のサポート」(161 ページ )。<br>『Open Client/Server Common Libraries リファレンス・マニュアル』。 |
| CS_LOGINFO | ct_getloginfo、ct_setloginfo | 個々のルーチンのリファレンス・ページ。 |

## CS_BROWSEDESC 構造体

ct_br_column は、CS_BROWSEDESC 構造体を使用して、ブラウズ・モードの select の結果として返されたカラムについての情報を返します。この情報は、アプリケーションが言語コマンドを構成し、ブラウズ・モード・テーブルを更新する必要があるときに役立ちます。

CS_BROWSEDESC 構造体の定義は次のとおりです。

```
/*
 ** CS_BROWSEDESC
 ** The Client-Library browse column description
 ** structure.
 */
 typedef struct _cs_browsedesc
 {
     CS_INT      status;
     CS_BOOL     isbrowse;
     CS_CHAR     origname[CS_MAX_CHAR];
     CS_INT      orignlen;
     CS_INT      tablenum;
     CS_CHAR     tablename[CS_OBJ_NAME];
     CS_INT      tabnlen;
 } CS_BROWSEDESC;
```

各パラメータの意味は次のとおりです。

- *status* は次の記号のビットマスクで、ビットごとの OR オペレーションが実行されます。
    - CS_EXPRESSION は、カラムが式の結果、たとえば、「select sum*2 from areas」クエリの「sum*2」になっていることを示します。
    - CS_HIDDEN は、カラムが公開されている「隠しカラム」であることを示します。「隠しキー」(250 ページ ) を参照してください。
    - CS_KEY は、カラムがキー・カラムであることを示します。詳細については、ct_keydata のリファレンス・ページを参照してください。
    - CS_RENAMED は、カラムの見出しがカラムのオリジナル名ではないことを示します。カラムが次の形式のクエリの結果になっている場合は、データベースのカラム名とは異なる見出しを持ちます。

      ```
      select Author = au_lname from authors
      ```

- *isbrowse* は、カラムがブラウズ・モード更新ができるかどうかを示します。

  カラムは、式の結果となっていない場合でブラウズ可能テーブルに属している場合は、更新することができます。テーブルは、ユニーク・インデックスおよびタイムスタンプ・カラムを持っている場合は、ブラウズ可能です。

  *isbrowse* は、カラムを更新できる場合は CS_TRUE に、更新できない場合は CS_FALSE に設定されます。

- *origname* は、データベース内のカラムの元の名前です。*origname* は、null で終了する文字列です。

  カラムに対するいかなる更新も、select 文のカラムを与えられた見出しによってではなく、元の名前によって参照してください。

- *orignlen* は、*origname* の長さのバイト数です。
- *tablenum* は、カラムの所属しているテーブルの番号です。select 文の from リストにある最初のテーブルのテーブル番号は 1、2 番目のテーブル番号は 2 ( 以下同様 ) というようになります。
- *tablename* は、カラムが属しているテーブル名です。*tablename* は、null で終了する文字列です。
- *tabnlen* は、*tablename* の長さ ( バイト数 ) です。

## CS_CLIENTMSG 構造体

CS_CLIENTMSG 構造体には、Client-Library エラーまたは情報メッセージについての情報が含まれています。

Client-Library は、次の 2 とおりの方法で CS_CLIENTMSG 構造体を使用します。

- メッセージの処理にコールバック方式を使用している接続の場合、CS_CLIENTMSG は、Client-Library が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・ルーチンへ渡す第 3 パラメータになります。
- 接続がインラインでメッセージを処理する場合、ct_diag は CS_CLIENTMSG で情報を返すことができます。

Client-Library のエラー処理およびサーバ・メッセージ処理については、「エラー処理」(136 ページ ) を参照してください。

CS_CLIENTMSG 構造体は、次のように定義されます。

```
/*
 ** CS_CLIENTMSG
 ** The Client-Library client message structure.
 */

 typedef struct _cs_clientmsg
 {
     CS_INT      severity;
     CS_MSGNUM   msgnumber;
     CS_CHAR     msgstring[CS_MAX_MSG];
     CS_INT      msgstringlen;

     /*
     ** If the error involved the operating
     ** system, the following fields contain
     ** operating-system-specific information:
     */
     CS_INT      osnumber;
     CS_CHAR     osstring[CS_MAX_MSG];
     CS_INT      osstringlen;

     /*
     ** Other information:
     */
     CS_INT      status;
     CS_BYTE     sqlstate[CS_SQLSTATE_SIZE];
     CS_INT      sqlstatelen;

 } CS_CLIENTMSG;
```

各パラメータの意味は次のとおりです。

- *severity* は、メッセージの重大度を表す記号値です。表 2-11 に重大度の有効値を示します。

**表 2-11：CS_CLIENTMSG の severity フィールドの値**

| Severity | 説明 |
| --- | --- |
| CS_SV_INFORM | エラーは何も発生していない。このメッセージは情報メッセージである。 |
| CS_SV_CONFIG_FAIL | Sybase 設定エラーが検出された。設定エラーには、ローカライゼーション・ファイルが見つからない場合のエラー、*interfaces* ファイルが見つからない場合のエラー、認識できないサーバ名が *interfaces* ファイル内にある場合のエラーがある。 |
| CS_SV_RETRY_FAIL | オペレーションは失敗したが、そのオペレーションをリトライできる。<br>このタイプのオペレーションの例としては、タイムアウトするネットワーク読み込みがある。 |
| CS_SV_API_FAIL | Client-Library ルーチンがエラーを生成した。一般的にこのエラーは、不正なパラメータまたは呼び出し順が原因で発生する。サーバ接続はおそらく使用可能である。 |
| CS_SV_RESOURCE_FAIL | リソース・エラーが発生した。一般にこのエラーは、malloc の失敗、またはファイル記述子がないことが原因で発生する。サーバ接続はおそらく使用可能ではない。 |
| CS_SV_COMM_FAIL | サーバでリカバリ不可能なエラーが発生した。<br>サーバ接続は使用可能ではない。 |
| CS_SV_INTERNAL_FAIL | 内部 Client-Library エラーが発生した。 |
| CS_SV_FATAL | 重大なエラーが発生した。すべてのサーバ接続は使用不可能である。 |

- *msgnumber* は、Client-Library のメッセージ番号です。「Client-Library メッセージの番号」(89 ページ ) を参照してください。
- *msgstring* は、NULL で終了する Client-Library メッセージ文字列です。

  アプリケーションがメッセージを連続させていない場合、*msgstring* は、null で終了することが保証されています。メッセージがトランケートされても、null で終了することが保証されます。

  アプリケーションが、メッセージを連続させていない場合、*msgstring* は、連続したメッセージの最後の部分だけが null で終了しています。

  「長いメッセージの連続化」(141 ページ ) を参照してください。
- *msgstringlen* は、*msgstring* のバイト単位の長さです。これは、常に実際の長さです。記号値 CS_NULLTERM ではありません。

- *osnumber* は、オペレーティング・システム・エラーがある場合には、そのエラー番号です。オペレーティング・システム・エラーが発生していない場合、Client-Library は、*osnumber* を 0 に設定します。
- *osstring* は、オペレーティング・システム・エラーがある場合、NULL で終了するオペレーティング・システム・エラー文字列です。
- *osstringlen* は、*osstring* の長さです。これは、常に実際の長さです。記号値 CS_NULLTERM ではありません。
- *status* は、エラー・メッセージの最初の部分なのか、中間または最後の部分なのかなど、さまざまなタイプの情報を示すビットマスクです。*status* に設定できる値は次のとおりです。

**表 2-12：CS_CLIENTMSG の status フィールドの値**

| 記号値 | 意味 |
| --- | --- |
| CS_FIRST_CHUNK | *msgstring* に含まれているメッセージ・テキストは、メッセージの最初の部分である。<br>CS_FIRST_CHUNK と CS_LAST_CHUNK が両方ともオンの場合、*msgstring* はすべてのメッセージを含んでいる。<br>CS_FIRST_CHUNK および CS_LAST_CHUNK が両方ともオンでない場合、*msgstring* は、メッセージの中間の部分を含んでいる。<br>「長いメッセージの連続化」(141 ページ ) を参照。 |
| CS_LAST_CHUNK | *msgstring* に含まれているメッセージ・テキストは、メッセージの最後の部分である。<br>CS_FIRST_CHUNK と CS_LAST_CHUNK が両方ともオンの場合、*msgstring* はすべてのメッセージを含んでいる。<br>CS_FIRST_CHUNK および CS_LAST_CHUNK が両方ともオンでない場合、*msgstring* は、メッセージの中間の部分を含んでいる。<br>「長いメッセージの連続化」(141 ページ ) を参照。 |

- *sqlstate* は、エラーを説明するバイト文字列です。

  クライアント・メッセージすべてが、メッセージに関連する SQL ステータス値を持つわけではありあません。メッセージに SQL ステータス値がない場合、*sqlstate* の値は「ZZZZZ」になります。
- *sqlstatelen* は、*sqlstate* 文字列の長さ ( バイト数 ) です。

## Client-Library メッセージの番号

Client-Library メッセージの番号は、4 バイトのサイズのコンポーネントをコード化する CS_INT 値によって表現されます。

### メッセージ番号の復号化

コンポーネントを別々に表示するために、Client-Library はメッセージ番号を復号化するための次のようなマクロを備えています。

- CS_LAYER − メッセージを生成した Client-Library レイヤを識別するレイヤ番号をアンパックします。
- CS_ORIGIN − エラーが Client-Library の内部と外部のどちらから発生したかを示す、メッセージのオリジンをアンパックします。
- CS_SEVERITY − メッセージの重大度をアンパックします。重大度コードとその意味の一覧については、「Client-Library メッセージの重大度」(90 ページ ) を参照してください。
- CS_NUMBER − ( 重大度、レイヤ、オリジンとともに ) メッセージを識別する、レイヤ固有のメッセージ番号をアンパックします。

これらのマクロは、(*ctpublic.h* にインクルードされている ) ヘッダ・ファイル *cstypes.h* 内で定義されます。

一般的なアプリケーションは、これらのマクロを使用してメッセージ番号を 4 つの部分に分け、その後それぞれの部分を別々に表示します。これらのマクロの使用例については、「クライアント・メッセージ・コールバックの例」(36 ページ ) および「タイムアウト・エラーの処理」(268 ページ ) を参照してください。

Client-Library と CS-Library は、メッセージ番号のコンポーネント・レイヤ、オリジン、番号をキーとして使用して、ライブラリのロケール・ファイルから取得されたテキストからローカライズ・メッセージ文字列を作成します。その後ローカライズ・メッセージ文字列は、CS_CLIENTMSG 構造体の *msgstring* としてアプリケーションに渡されます。

---

**注意**　使用しているプラットフォームの Sybase ローカライゼーション・ファイルの構造体については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

---

エラー・メッセージのテキストは、次のようなコンポーネントから構成されます。

```
routine: layer: origin: description
```

各パラメータの意味は次のとおりです。

- *routine* は、エラーが発生したライブラリ・ルーチンの名前です。
- *layer* は、*cslib.loc* の [cslayer] セクション (CS-Library エラーの場合 ) と *ctlib.loc* の [ctlayer] セクション (Client-Library エラーの場合 ) のどちらかから取得されたレイヤ記述です。
- *origin* は、*cslib.loc* の [csorigin] セクション (CS-Library エラーの場合 ) と *ctlib.loc* の [ctorigin] セクション (Client-Library エラーの場合 ) のどちらかから取得されたフレーズです。
- *description* は、ファイルの適切なレイヤ固有のセクションから取得されたエラー記述です。

次は、一般的なクライアント・メッセージ・コールバック・ルーチンによって出力される可能性がある英語のエラー文字列です。

```
Client Library error(16843066):
 severity(1) number(58) origin(1) layer(1)
 ct_bind():user api layer:external error:The format
field of the CS_DATAFMT structure must be CS_FMT_UNUSED
if the datatype field is int.
```

### Client-Library メッセージの重大度

表 2-13 に、Client-Library メッセージの重大度を示します。

**表 2-13：Client-Library メッセージの重大度**

| 重大度 | 説明 | ユーザのアクション |
|---|---|---|
| CS_SV_INFORM | エラーは何も発生していない。このメッセージは情報メッセージである。 | 対処不要。 |
| CS_SV_CONFIG_FAIL | Sybase 設定エラーが検出された。設定エラーには、ローカライゼーション・ファイルが見つからない場合のエラー、*interfaces* ファイルが見つからない場合のエラー、認識できないサーバ名が *interfaces* ファイル内にある場合のエラーがある。 | アプリケーションのエンド・ユーザが問題点を解決できるように、エラーを表示する。 |

| 重大度 | 説明 | ユーザのアクション |
|---|---|---|
| CS_SV_RETRY_FAIL | オペレーションは失敗したが、そのオペレーションをリトライできる。<br>このタイプのオペレーションの例としては、タイムアウトするネットワーク読み込みがある。 | アプリケーションのクライアント・メッセージ・コールバックからの戻り値によって、Client-Library がそのオペレーションをリトライするかどうかを決定する。<br>クライアント・メッセージ・コールバックが CS_SUCCEED を返した場合、Client-Library は、そのオペレーションをリトライする。<br>クライアント・メッセージ・コールバックが CS_FAIL を返した場合、Client-Library はそのオペレーションをリトライしないで、接続を "dead" とマーク付けする。この場合、ct_close(CS_FORCE_CLOSE) を呼び出して接続をクローズし、その後、ct_connect ルーチンを呼び出してその接続を再オープンすること。 |
| CS_SV_API_FAIL | Client-Library ルーチンがエラーを生成した。一般的にこのエラーは、不正なパラメータまたは呼び出し順が原因で発生する。サーバ接続はおそらく使用可能である。 | ct_cancel(CS_CANCEL_ALL) を呼び出して、接続をクリーンアップすること。<br>ct_cancel(CS_CANCEL_ALL) が CS_SUCCEED を返した場合、サーバ接続は影響を受けない。クライアント・メッセージ・コールバック・ルーチン内からこのタイプのキャンセルを実行することはできない。 |
| CS_SV_RESOURCE_FAIL | リソース・エラーが発生した。一般にこのエラーは、malloc の失敗、またはファイル記述子がないことが原因で発生する。サーバ接続はおそらく使用可能ではない。 | ct_close(CS_FORCE_CLOSE) を呼び出してサーバ接続をクローズし、その後、必要に応じて ct_connect ルーチンを呼び出してその接続を再オープンすること。クライアント・メッセージ・コールバック・ルーチン内からこれらのルーチンを呼び出すことはできない。 |
| CS_SV_COMM_FAIL | サーバ通信チャネル内でリカバリ不可能なエラーが発生した。<br>サーバ接続は使用可能ではない。 | ct_close(CS_FORCE_CLOSE) を呼び出してサーバ接続をクローズし、その後、必要に応じて ct_connect ルーチンを呼び出して、その接続を再オープンすること。クライアント・メッセージ・コールバック・ルーチン内からこれらのルーチンを呼び出すことはできない。 |
| CS_SV_INTERNAL_FAIL | 内部 Client-Library エラーが発生した。 | ct_exit(CS_FORCE_EXIT) を呼び出して Client-Library を終了してから、アプリケーションを終了すること。クライアント・メッセージ・コールバック・ルーチン内から ct_exit を呼び出すことはできない。 |

| 重大度 | 説明 | ユーザのアクション |
| --- | --- | --- |
| CS_SV_FATAL | 重大なエラーが発生した。すべてのサーバ接続は使用不可能である。 | ct_exit(CS_FORCE_EXIT) を呼び出して Client-Library を終了してから、アプリケーションを終了すること。クライアント・メッセージ・コールバック・ルーチン内から ct_exit を呼び出すことはできない。 |

### 特定の Client-Library メッセージの処理

ほとんどの Client-Libary メッセージは、プログラム内のコーディング・エラーを表し、エラー記述はユーザにその問題点を知らせます。これらのエラーは、メッセージを表示するか、アプリケーションのエラー・ファイルにそのメッセージのログを書き込むかのどちらかによって最適に処理されます。

これ以外の場合は、プログラムがエラーを認識して、特定のアクションを行う必要があることもあります。次に例を示します。

- サーバからの読み込みがタイムアウトした場合、プログラムは、処理されるコマンドのキャンセルを決定することがあります。
- 設定エラーの場合、プログラムは特定の問題点を認識して、アプリケーションのエンド・ユーザに特定の指示を与えるアプリケーション定義メッセージを表示する必要があることもあります。

エラーは、そのエラーの 4 つのコンポーネントによってユニークに記述されます。次に例を示す ERROR_SNOL などのマクロは、メッセージ番号を認識するのに便利です。

```
/*
 ** ERROR_SNOL(error_numb, severity, number, origin, layer)
 **
 **   Error comparison for Client-Library or CS-Library errors.
 **   Breaks down a message number and compares it to the given
 **   constants for severity, number, origin, and layer. Returns
 **   non-zero if the error number matches the 4 components.
 */
 #define ERROR_SNOL (e, s, n, o, l) ¥
   ( (CS_SEVERITY(e) == s) && (CS_NUMBER(e) == n) ¥
    && (CS_ORIGIN(e) == o) && (CS_LAYER(e) == l ) )
```

表 2-14 に、Client-Library の一部のメッセージのエラー・コードを示します。これらのエラーはリカバリ可能であるか、またはクライアント・マシンとリモート・サーバ・マシンのどちらかでの設定上の問題を表します。

表 2-14：Client-Library エラー

| 重大度 | 番号 | オリジン | レイヤ | 原因 |
|---|---|---|---|---|
| CS_SV_RETRY_FAIL | 63 | 2 | 1 | サーバからの読み込みがタイムアウトした。「タイムアウト・エラーの処理」(268 ページ ) を参照。 |
| CS_SV_CONFIG_FAIL | 8 | 3 | 5 | *interfaces* ファイル ( または使用しているプラットフォームの対応するファイル ) が見つからなかった。<br>「interfaces ファイルのロケーション」(251 ページ ) を参照。 |
| CS_SV_CONFIG_FAIL | 3 | 3 | 5 | *interfaces* ファイルまたは接続のディレクトリ・ソース内でサーバ名が見つからない。 |
| CS_SV_COMM_FAIL | 4 | 3 | 4 | 接続しようとしたが、リモート・サーバとのログイン・ダイアログを確立できなかったために失敗した。<br>このエラーは、リモート・サーバがダウンしている場合に発生する。 |
| CS_SV_COMM_FAIL | 131 | 3 | 5 | Net-Library ドライバを初期化できなかったために、ct_init ルーチンが失敗した。このエラーに対してはクライアント・メッセージ・コールバックは呼び出されない。Client-Library は *stderr* デバイスにメッセージを出力する。このエラーは、*libtcl.cfg* ファイルの [DRIVER] セクションの設定の誤りによる場合がほとんどである。Client-Library での Net-Library ドライバのロード方法の詳細については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照。 |
| CS_SV_API_FAIL | 132 | 4 | 1 | データのフェッチ中に、結果項目のバインドでトランケーションが発生する。<br>受信されるデータに対して (ct_bind でバインドされた ) 変数が小さすぎる場合に ct_fetch ルーチンを呼び出すと、このエラーが発生する。カラム・インジケータが使用される場合、アプリケーションはそのインジケータ値をチェックして、トランケートされたカラムを確認する。 |

# CS_DATAFMT 構造体

CS_DATAFMT 構造体は、データ値とプログラム変数を記述するために使用します。次に例を示します。

- ct_bind は、対象変数を記述する CS_DATAFMT 構造体を要求します。

- ct_describe は、結果データ項目を記述する CS_DATAFMT 構造体を返します。
- ct_param と ct_setparam は、入力パラメータを記述する CS_DATAFMT 構造体を要求します。
- cs_convert は、変換前後のデータを記述する CS_DATAFMT 構造体を要求します。cs_convert 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。

ほとんどのルーチンは、CS_DATAFMT のフィールドのサブセットしか使用しません。たとえば、ct_bind は *name*、*status*、および *usertype* フィールドを使用しません。また、ct_describe は *format* フィールドを使用しません。ルーチンが CS_DATAFMT のどのフィールドを使用するかについては、該当するルーチンのリファレンス・ページを参照してください。

```
typedef struct _cs_datafmt
 {
     CS_CHAR name[CS_MAX_CHAR];  /* Name of data */
     CS_INT  namelen;            /* Length of name */
     CS_INT  datatype;           /* Datatype */
     CS_INT  format;             /* Format symbols */
     CS_INT  maxlength;          /* Data max length */
     CS_INT  scale;              /* Scale of data */
     CS_INT  precision;          /* Data precision */
     CS_INT  status;             /* Status symbols */

   /*
     ** The following field indicates the number of
     ** rows to copy, per ct_fetch call, to a bound
     ** program variable. ct_describe sets this field
     ** to a default value of 1. ct_bind is the only
     ** routine that reads this field.
     */
     CS_INT         count;

   /*
     ** These fields are used to support Adaptive Server Enterprise
     ** user-defined datatypes and international
     ** datatypes:
     */
    CS_INT      usertype;   /* Svr user-def'd type */
    CS_LOCALE   *locale;    /* Locale information */

} CS_DATAFMT;
```

各パラメータの意味は次のとおりです。

- *name* は、データ名です。*name* は、カラム名またはパラメータ名の場合もあります。
- *namelen* は *name* の長さ ( バイト数 ) です。*namelen* を CS_NULLTERM に設定し、null で終了する名前を示してください。*name* が NULL の場合は、*namelen* を 0 に設定します。
- *datatype* は、データのデータ型を表すタイプ定数です。 これは、Open Client のデータ型の 1 つ、または Open Client ユーザ定義データ型です。データ型の詳細については、「データ型のサポート」(339 ページ ) を参照してください。

  *datatype* フィールドを *usertype* フィールドと混同しないでください。*datatype* は、常に、データの Open Client データ型を記述するために使用されます。*usertype* は、データが Open Client データ型に加えて Adaptive Server Enterprise のユーザ定義のデータ型を持っている場合に使用されます。

  たとえば、次の Adaptive Server コマンドは Adaptive Server Enterprise のユーザ定義タイプ *birthday* を作成します。

  ```
  sp_addtype birthday, datetime
  ```

  さらに、次のコマンドによって、この型のカラムを含むテーブルが作成されます。

  ```
  create table birthdays
   (
       name        varchar(30),
       happyday    birthday
   )
  ```

  Client-Library アプリケーションがこのテーブルに対して select を実行し、ct_describe を呼び出して、結果セットの *birthday* カラムの記述を得ると、CS_DATAFMT 構造体の *datatype* フィールドおよび *usertype* フィールドは次のように設定されます。

  *datatype* は、CS_DATETIME_TYPE に設定されます。
  *usertype* は、タイプ *birthday* に対する Adaptive Server Enterprise ID に設定されます。

- *format* は、文字またはバイナリ・データ対象のフォーマットを記述します。*format* は、次の記号のビットマスクで、OR 演算子と組み合わせます。

**表 2-15：CS_DATAFMT の format フィールドの値**

| 記号 | 意味 | 注意 |
|---|---|---|
| CS_FMT_NULLTERM | データは null で終了するものにする。 | 文字または text データ |
| CS_FMT_PADBLANK | 送信先変数の長さいっぱいまで、データの後にブランクを埋め込む必要がある。 | 文字または text データ |
| CS_FMT_PADNULL | 送信先変数の長さいっぱいまで、データの後に NULL を埋め込む必要がある。 | 文字、text、バイナリ、または image データ |
| CS_FMT_UNUSED | フォーマット情報は提供されていない。 | すべてのデータ型 |

- *maxlength* は、CS_DATAFMT を使用している Open Client ルーチンによって長さが異なります。表 2-16 に *maxlength* の意味を示します。

**表 2-16：CS_DATAFMT の maxlength フィールドの値**

| Open Client ルーチン | maxlength |
|---|---|
| ct_bind | バインド変数の長さ。 |
| ct_describe | 記述されているカラムまたはパラメータの可能最大長。 |
| ct_dyndesc | 記述されているカラムまたはパラメータの可能最大長。 |
| ct_dynsqlda | 記述されているカラムまたはパラメータの可能最大長。 |
| ct_param | リターン・パラメータ・データの期待最大長。 |
| ct_setparam | リターン・パラメータ・データの期待最大長。ct_setparam ルーチンの *datalen* パラメータが NULL として渡される場合、*maxlength* はそのパラメータのすべての入力値の長さを指定する。 |
| cs_convert | 送信元データの長さおよび対象バッファ領域の長さ。 |

- *scale* は、データの小数点以下の最大桁数です。*scale* は、decimal または numeric データ型でだけ使用されます。

  *scale* の有効値は 0 から 77 です。デフォルトは 0 です。CS_MIN_SCALE、CS_MAX_SCALE、CS_DEF_PREC は、位取りの最小値、最大値、およびデフォルト値をそれぞれ定義します。

  送信先データが送信元データと同じ位取りを使用するには、*scale* を CS_SRC_VALUE に設定します。

  *scale* は、*precision* 以下でなければなりません。

- *precision* は、データ内に表すことができる decimal の最大桁数です。*precision* は、decimal または numeric データ型でだけ使用されます。

  *precision* の有効値は 1 から 77 です。デフォルトは 18 です。CS_MIN_PREC、CS_MAX_PREC、および CS_DEF_PREC は、精度の最小値、最大値、およびデフォルト値をそれぞれ定義します。

  送信先データが送信元データと同じ精度を使用しなければならないことを示すためには、*precision* を CS_SRC_VALUE に設定してください。

  *precision* は、*scale* 以上でなければなりません。

- *status* はさまざまなタイプの情報を示すビットマスクです。表 2-17 に *status* を構成できる値を示します。

表 2-17：CS_DATAFMT の status フィールドの値

| 記号値 | 意味 | 値が有効となるルーチン |
|---|---|---|
| CS_CANBENULL | カラムは値 NULL を含むことができる。 | ct_describe、ct_dyndesc、ct_dynsqlda |
| CS_HIDDEN | カラムは公開された隠しカラムである。<br>「隠しキー」(250 ページ) を参照。 | ct_describe、ct_dyndesc、ct_dynsqlda |
| CS_IDENTITY | カラムは識別カラムである。 | ct_describe、ct_dyndesc、ct_dynsqlda |
| CS_KEY | カラムは、キー・カラムである。<br>詳細については、ct_keydata のリファレンス・ページを参照。 | ct_describe、ct_dyndesc、ct_dynsqlda |
| CS_UPDATABLE | カラムは、更新可能なカーソル・カラムである。 | ct_describe、ct_dyndesc、ct_dynsqlda |
| CS_VERSION_KEY | カラムは、ローのバージョン・キーの一部である。<br>Adaptive Server Enterprise は、カーソルの位置付けにバージョン・キーを使用する。<br>詳細については、ct_keydata のリファレンス・ページを参照。 | ct_describe、ct_dyndesc、ct_dynsqlda |
| CS_TIMESTAMP | カラムは、*timestamp* カラム。アプリケーションは、ブラウズ・モード更新の実行時に、*timestamp* カラムを使用する。 | ct_describe |
| CS_UPDATECOL | パラメータは、カーソル宣言コマンドの update 句内のカラム名。 | ct_param、ct_setparam、ct_dyndesc、ct_dynsqlda |
| CS_INPUTVALUE | パラメータは、Client-Library コマンドのための入力パラメータ値である。 | ct_param、ct_setparam、ct_dyndesc、ct_dynsqlda |
| CS_RETURN | パラメータは、RPC コマンドへの戻りパラメータ。 | ct_param、ct_setparam、ct_dyndesc、ct_dynsqlda |

- *count* は、ct_fetch 呼び出しごとにプログラム変数にコピーするローの数です。ct_bind のみが、*count* を使用します。
- *usertype* は、サーバによって返されたデータがあれば、そのデータのサーバ・ユーザ定義データ型です。*usertype* は、Client-Library ユーザ定義データ型に対してではなく、サーバ・ユーザ定義データ型に対してのみ使用されます。Client-Library ユーザ定義データ型については、「データ型のサポート」(339 ページ) を参照してください。
- *locale* は、ローカライゼーション情報が格納されている CS_LOCALE 構造体へのポインタです。ローカライゼーション情報が必要でない場合には、*locale* を NULL に設定してください。

  CS_DATAFMT 構造体を使用する前に、*locale* を NULL または有効な CS_LOCALE 構造体のアドレスのいずれかに設定して、*locale* が有効な値となっていることを確認してください。

## CS_IODESC 構造体

CS_IODESC 構造体、つまり「*I/O 記述子構造体*」は *text* データまたは *image* データを記述します。

アプリケーションは、後で更新を予定している *text* または *image* 値を取得した後で、ct_data_info を呼び出して CS_IODESC 構造体を取得します。有効な CS_IODESC を取得後、一般のアプリケーションでは、CS_IODESC 構造体を使用して *text* または *image* の値を更新する前に、*locale*、*total_txtlen*、*log_on_update* フィールドの値だけが変更されます。

アプリケーションは、ct_command を呼び出して *text* 値または *image* 値を更新するためのデータ送信オペレーションを開始した後に、ct_data_info を呼び出して CS_IODESC 構造体を定義します。

CS_IODESC は下記のように定義されています。

```
typedef struct _cs_iodesc
 {
     CS_INT         iotype;                /* CS_IODATA.          */
     CS_INT         datatype;              /* Text or image.      */
     CS_LOCALE      *locale;               /* Locale information. */
     CS_INT         usertype;              /* User-defined type.  */
     CS_INT         total_txtlen;          /* Total data length.  */
     CS_INT         offset;                /* Reserved.           */
     CS_BOOL        log_on_update          /* Log the insert?     */
     CS_CHAR        name[CS_OBJ_NAME];     /* Name of data object.*/
     CS_INT         namelen                /* Length of name.     */
```

```
    CS_BYTE        timestamp[CS_TS_SIZE];   /* Adaptive Server id. */
    CS_INT         timestamplen;            /* Length of timestamp.*/
    CS_BYTE        textptr[CS_TP_SIZE];     /* Adaptive Server ptr.*/
    CS_INT         textptrlen;              /* Length of textptr.  */
    CS_INT         delete_length;           /* Number of bytes to  */
                                            /* delete/overwrite for*/
                                            /* partial updates.    */
} CS_IODESC;
```

各パラメータの意味は次のとおりです。

- *iotype* は実行する I/O のタイプを示します。text オペレーションおよび image オペレーションの場合、*iotype* の値は CS_IODATA または CS_IOPARTIAL となります。CS_IOPARTIAL 設定は、text カラムまたは image カラムで部分更新を実行するよう指定します。
- *datatype* は、データ・オブジェクトのデータ型です。*datatype* の値は、CS_TEXT_TYPE と CS_IMAGE_TYPE のみです。
- *locale* は、text または image 値のローカライゼーション情報を含む CS_LOCALE 構造体へのポインタです。ローカライゼーション情報が必要でない場合には、*locale* を NULL に設定してください。

  CS_IODESC 構造体を使用する前に、*locale* を NULL または有効な CS_LOCALE 構造体のアドレスのいずれかに設定して、*locale* が有効な値となっていることを確認してください。
- *usertype* は、データ・オブジェクトの Adaptive Server Enterprise ユーザ定義データ型がある場合にはそのデータ型です。データ送信オペレーションでは、*usertype* は無視されます。データ受信オペレーションでは、Client-Library は *datatype* を設定する代わりに *usertype* を設定するのではなく、*datatype* を設定したうえで、*usertype* も設定します。
- *total_txtlen* は、text または image 値の全体の長さ ( バイト数 ) です。

**Unicode と部分更新**

使用しているクライアント・アプリケーションが 2 バイトの Unicode データ型を使用して部分更新を実行する場合、文字の分断を回避するため、確実に偶数のバイト数を送信する必要があります。ct_send_data の *buflen* パラメータと CS_IODESC の *total_txtlen* フィールドを使用して、Unicode データの長さをバイト単位で指定できます。Unitext データの部分更新を実行するには、*offset* 値と *delete_length* 値を文字数として指定すると同時に、*total_txtlen* をバイト単位で指定してください。

- *offset* は、部分更新の影響を受けるカラムの最初のバイトを示します。
- *log_on_update* は、サーバがこの text または image 値に対する更新のログを取る必要があるかどうかを記述します。
- *name* は、text または image カラム名です。*name* は、*table.column* フォームの null で終了する文字列です。
- *namelen* は、*name* の NULL ターミネータを含まないバイト単位の長さです。CS_IODESC を埋めている場合、アプリケーションは、NULL で終了する名前を示すために *namelen* を CS_NULLTERM に設定します。
- *timestamp* は、カラムのテキスト・タイムスタンプです。テキスト・タイムスタンプは、text または image カラムが最後に修正された時間を示します。
- *timestamplen* は、*timestamp* の長さ ( バイト数 ) です。
- *textptr* は、カラムのテキスト・ポインタです。テキスト・ポインタは、text または image カラムのデータを指す内部サーバ・ポインタです。*textptr* は、データ送信オペレーションでターゲット・カラムを識別します。
- *textptrlen* は、*textptr* の長さ ( バイト数 ) です。
- *delete_length* は、部分更新がすでに指定されている text カラムまたは image カラムで上書きされるかカラムから削除されることになるバイト数を示します。

## CS_OID 構造体

CS_OID 構造体にはオブジェクト識別子が保管されます。

「**オブジェクト識別子 (OID: Object Identifier)**」はコード化された文字列であり、マシンやネットワークに依存せずに、分散環境内のオブジェクトをユニークに識別するための手段を提供します。OID は分散環境内のすべてのアプリケーションに対して同じ「**シンボリック・グローバル名**」として機能します。

Sybase では次のようなものを表すために OID を使用します。

- ディレクトリ・オブジェクト
- ディレクトリ・オブジェクト内の属性タイプ

- セキュア・クライアント／サーバ接続を保証するセキュリティ・メカニズム。セキュリティ・メカニズムでは、クライアント・マシン上とサーバ・マシン上で「**ローカル名**」が異なる場合があります。混乱を避けるために、クライアントとサーバの両方のセキュリティ・メカニズムを識別するグローバル名として OID が使用されます。セキュリティ・メカニズムと接続との関連については、「ネットワーク・セキュリティ・メカニズムの選択」(293 ページ ) を参照してください。

## オブジェクト識別子のコード化

OID はピリオドで区切られた 10 進整数のシーケンスとしてコード化されます。OID は ISO 標準に従って定義されており、異なるベンダ間での重複を避けるため階層構造になっています。階層構造内では、各ベンダにユニークなプレフィクスが割り当てられます。たとえば、Sybase のプレフィクスは「1.3.1.4.1.897」であり、すべての Sybase OID はこのプレフィックスを持ちます。

## CS_OID 構造体の定義

CS_OID 構造体は、Client-Library ルーチンとアプリケーション・コード間で OID を通信するために必要とされます。

CS_OID 構造体は ct_ds_lookup または ct_ds_objinfo を呼び出す場合に使用します。

CS_OID 構造体は次のように定義されます。

```
typedef struct _cs_oid
 {
         CS_INT  oid_length;
         CS_CHAR oid_buffer[CS_MAX_DS_STRING];
 } CS_OID;
```

各パラメータの意味は次のとおりです。

- *oid_length* は OID 文字列の長さです。OID 文字列が null で終了する場合は、この長さには null ターミネータは含まれません。
- *oid_buffer* は OID 文字列を保持するバイト配列です。この文字列は常に null で終了するとはかぎりません。

### 事前に定義された OID 文字列の使い方

Client-Library のヘッダ・ファイルは、アプリケーションが OID の初期設定または比較に使用できるように OID 文字列を定義します。事前に定義された OID 文字列は次の目的で使用されます。

- ディレクトリ・オブジェクト・クラスの識別。Sybase ディレクトリ・オブジェクトは Server、OID は CS_OID_OBJSERVER です。

  「サーバ・ディレクトリ・オブジェクト」(319 ページ ) を参照してください。

- 所定のディレクトリ・オブジェクトの属性の識別。各属性を識別する事前に定義された OID 文字列についてはディレクトリ・オブジェクトの定義を参照してください。

## CS_SERVERMSG 構造体

CS_SERVERMSG 構造体には、サーバ・エラーまたは情報メッセージについての情報が含まれています。

Client-Library は、次の 2 とおりの方法で CS_SERVERMSG 構造体を使用します。

- メッセージの処理にコールバックを使用している接続の場合、CS_SERVERMSG は、Client-Library が接続のサーバ・メッセージ・コールバックへ渡す第 3 のパラメータです。
- 接続がインラインでメッセージを処理する場合、ct_diag は CS_SERVERMSG で情報を返すことができます。

エラー処理とメッセージ処理については、「エラー処理」(136 ページ ) を参照してください。

CS_SERVERMSG 構造体の定義は次のとおりです。

```
/*
 ** CS_SERVERMSG
 ** The Client-Library server message structure.
 */

typedef struct _cs_servermsg
 {
     CS_MSGNUM   msgnumber;
     CS_INT      state;
     CS_INT      severity;
     CS_CHAR     text[CS_MAX_MSG];
     CS_INT      textlen;
     CS_CHAR     svrname[CS_MAX_CHAR];
     CS_INT      svrnlen;
```

```
        /*
        ** If the error involved a stored procedure,
        ** the following fields contain information
        ** about the procedure:
        */
        CS_CHAR      proc[CS_MAX_CHAR];
        CS_INT       proclen;
        CS_INT       line;

        /*
        ** Other information.
        */
        CS_INT    status;
        CS_BYTE   sqlstate[CS_SQLSTATE_SIZE];
        CS_INT    sqlstatelen;

    } CS_SERVERMSG;
```

各パラメータの意味は次のとおりです。

- *msgnumber* は、サーバ・メッセージ番号です。Adaptive Server Enterprise メッセージの一覧を見るには、次のように Transact-SQL コマンドを実行してください。

  ```
  select * from sysmessages
  ```

- *state* は、サーバ・エラーのステータスです。
- *severity* は、メッセージの重大度です。Adaptive Server Enterprise メッセージの重大度の一覧を見るには、次のように Transact-SQL コマンドを実行してください。

  ```
  select distinct severity from sysmessages
  ```

- text は、サーバ・メッセージのテキストです。

  アプリケーションが、メッセージを連続させている場合、text はトランケートされていても、null で終了していることが保証されます。

  アプリケーションが、メッセージを連続させている場合、text は、連続しているメッセージの最後のまとまりの場合にかぎり、null で終了します。

  「長いメッセージの連続化」(141 ページ ) を参照してください。

- *textlen* は、text のバイト単位での長さです。これは、常に実際の長さです。記号値 CS_NULLTERM ではありません。
- *svrname* は、メッセージを生成したサーバ名です。これは、*interfaces* ファイルに現れるサーバ名です。*svrname* は null で終了する文字列です。
- *svrnlen* は *svrname* の長さ ( バイト数 ) です。

- *proc* は、メッセージを生成したストアド・プロシージャがある場合には、そのプロシージャ名です。*proc* は null で終了する文字列です。
- *proclen* は、*proc* の長さ ( バイト数 ) です。
- *line* は、メッセージを生成した行がある場合には、その行番号です。*line* は、ストアド・プロシージャ内の行番号またはコマンド・バッチ内の行番号になることもあります。
- *status* は、拡張エラー・データがメッセージに含まれるかどうかなど、さまざまなタイプの情報を示すために使用されるビットマスクです。表 2-18 に *status* に設定できる値を示します。

**表 2-18：CS_SERVERMSG の status フィールドの値**

| 記号値 | 意味 |
|---|---|
| CS_HASEED | 拡張エラー・データがメッセージに含まれている。<br>「拡張エラー・データ」(143 ページ ) を参照。 |
| CS_FIRST_CHUNK | text に含まれるメッセージ・テキストは、メッセージの最初のまとまり。<br>CS_FIRST_CHUNK と CS_LAST_CHUNK が両方ともオンになっていると、text にはメッセージ全体が含まれる。<br>CS_FIRST_CHUNK も CS_LAST_CHUNK もオンになっていないと、text にはメッセージの中間のまとまりが含まれる。<br>「長いメッセージの連続化」(141 ページ ) を参照。 |
| CS_LAST_CHUNK | text に含まれるメッセージ・テキストは、メッセージの最後のまとまり。<br>CS_FIRST_CHUNK と CS_LAST_CHUNK が両方ともオンになっていると、text にはメッセージ全体が含まれる。<br>CS_FIRST_CHUNK も CS_LAST_CHUNK もオンになっていないと、text にはメッセージの中間のまとまりが含まれる。<br>「長いメッセージの連続化」(141 ページ ) を参照。 |

- *sqlstate* は、エラーを説明するバイト文字列です。

  すべてのサーバ・メッセージが SQL ステータス値を持っているわけではありません。メッセージに SQL ステータス値がない場合、*sqlstate* の値は「ZZZZZ」になります。

- *sqlstatelen* は、*sqlstate* 文字列の長さ ( バイト数 ) です。

## SQLCA 構造体

SQLCA 構造体は、ct_diag とともに使用して、Client-Library およびサーバのエラー・メッセージと情報メッセージを取得します。

SQLCA 構造体は、次のように定義されます。

```
/*
 ** SQLCA
 ** The SQL Communications Area structure.
 */

 typedef struct _sqlca
 {
     char    sqlcaid[8];
     long    sqlcabc;
     long    sqlcode;

     struct
     {
         long    sqlerrml;
         char    sqlerrmc[256];
     } sqlerrm;

     char    sqlerrp[8];
     long    sqlerrd[6];
     char    sqlwarn[8];
     char    sqlext[8];

 } SQLCA;
```

各パラメータの意味は次のとおりです。

- *sqlcaid* は「SQLCA」です。
- *sqlcabc* は無視されます。

- *sqlcode* は、サーバまたは Client-Library メッセージ番号です。Client-Library によるメッセージ番号の *sqlcode* へのマップ方法については、「SQLCODE 構造体」(108 ページ ) を参照してください。
- *sqlerrml* は、(*sqlerrmc* に入っているテキストの長さではなく ) 実際のメッセージ・テキストの長さです。
- *sqlerrmc* は、メッセージの null で終了するテキストです。メッセージがその配列に対して長すぎる場合、Client-Library は、NULL ターミネータを付け加える前にそのメッセージをトランケートします。
- *sqlerrp* は、エラーの発生時に実行中のストアド・プロシージャがある場合は、null で終了するプロシージャの名前です。名前がその配列に対して長すぎる場合、Client-Library は、NULL ターミネータを付け加える前にそのメッセージをトランケートします。
- *sqlerrd[2]* は、現在のコマンドによって影響を受けるロー数です。このフィールドは、現在のメッセージが「number of rows affected」の場合にのみ設定されます。この他の場合、*sqlerrd[2]* は、CS_NO_COUNT 値になります。
- *sqlwarn* は、警告の配列です。
  - *sqlwarn[0]* がブランクの場合、その他の *sqlwarn* 変数はすべてブランクになります。*sqlwarn[0]* がブランクではない場合、他の *sqlwarn* 変数のうちの少なくとも 1 つが「W」に設定されます。
  - *sqlwarn[1]* が「W」である場合、Client-Library がカラム値をホスト変数へコピーするときに少なくとも 1 つのカラム値がトランケートされています。
  - *sqlwarn[2]* が「W」である場合、少なくとも 1 つの null 値が関数の引数セットから削除されています。
  - *sqlwarn[3]* が「W」である場合、結果セットの項目のすべてではなく一部がバインドされています。このフィールドは、CS_ANSI_BINDS プロパティが CS_TRUE に設定されている場合にのみ設定されます (「ANSI 形式のバインド」(236 ページ ) を参照 )。
  - *sqlwarn[4]* が「W」である場合、動的 SQL の update 文または delete 文に where 句が含まれていませんでした。
  - *sqlwarn[5]* が「W」である場合、サーバの変換またはトランケート・エラーが発生しています。
- *sqlext* は無視されます。

# SQLCODE 構造体

SQLCODE 構造体は、Client-Library およびサーバのエラー・メッセージと情報メッセージ・コードを取得するために、ct_diag と組み合わせて使用できます。

アプリケーションは、long 型の整数として SQLCODE 構造体を宣言しなくてはなりません。

Client-Library は常に、SQLCODE と SQLCA 構造体の *sqlcode* フィールドを同一に設定します。

## SQLCODE へのサーバ・メッセージのマッピング

サーバ・メッセージ番号は、重大度が 0 になっている場合に、0 の SQLCODE にマップされます。また、その他のサーバ・メッセージも 0 の SQLCODE にマップされる場合があります。

サーバ・メッセージ番号は、SQLCODE に入れられる前に反転されます。これにより、エラーが発生した場合に SQLCODE が必ず負になります。

サーバ・メッセージの一覧を取得するには、次の Transact-SQL 文を実行してください。

```
select * from sysmessages
```

## SQLCODE への Client-Library メッセージのマッピング

Client-Library メッセージ「No rows affected」は、100 の SQLCODE にマップされます。重大度が CS_SV_INFORM である Client-Library メッセージは、0 の SQLCODE にマップされます。また、その他の Client-Library メッセージも 0 の SQLCODE にマップされる場合があります。

Client-Library メッセージ番号は、SQLCODE に入れられる前に反転されます。これにより、エラーが発生した場合に SQLCODE が必ず負になります。

「Client-Library メッセージの番号」(89 ページ ) を参照してください。

# SQLSTATE 構造体

SQLSTATE 構造体は、ct_diag とともに使用し、Client-Library またはサーバ・メッセージと関連する SQL ステータス情報があれば、それを取得します。

アプリケーションは、6 文字の配列として SQLSTATE 構造体を宣言する必要があります。

CS_CLIENTMSG および CS_SERVERMSG 構造体の *sqlstate* フィールドは SQLSTATE に対して、8 バイトで定義されることを除いて、同一に扱われます。最後の 2 バイトは無視されます。

# コマンド

クライアント／サーバ・モデルでは、サーバは複数のクライアントからコマンドを受け取って、データやその他の情報をクライアントに返すことによって応答します。Open Client アプリケーションは Client-Library ルーチンを使用してサーバにコマンドを送信します。

表 2-19 は、Client-Library のコマンド・タイプの一覧表です。

**表 2-19：コマンド・タイプ**

| 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、<br>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。<br>パッケージ・コマンドは、メインフレーム Open Server サーバだけでサポートされる。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 | 大きな text および image カラム値をサーバにアップロードするためのコマンドを起動する。 |

## コマンドの送信

すべてのコマンドは次の 3 つの手順で定義されて送信されます。

1. コマンドを起動します。これによって、コマンド・タイプとその実行内容を識別します。
2. 必要に応じて、パラメータ値を定義します。
3. コマンドを送信します。ct_send はコマンド記号とデータをネットワークに書き込みます。サーバはそのコマンドを読み込んで解釈して実行します。

### コマンドの起動

アプリケーションは次のような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 )
- カーソル自体がホスト変数を含んでいるか、カーソルのカラムの一部が更新用である場合は、カーソル宣言コマンド
- カーソル自体がホスト言語パラメータを含んでいる場合は、カーソル・オープン・コマンド
- 更新文のテキストに変数が含まれる場合は、カーソル更新コマンド
- 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各パラメータについて一度だけ ct_param または ct_setparam を呼び出します。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。これにより、アプリケーションはコマンドを再送する前に ct_setparam で指定されたパラメータ値を変更できます。

## コマンドの送信

コマンドが開始されてそのパラメータが定義されている場合、アプリケーションは ct_send を呼び出して、コマンドをサーバに送信します。サーバはこのコマンドを解釈して実行し、結果をクライアント・アプリケーションに返します。

## コマンドの再送信

ほとんどのコマンド・タイプについて、Client-Library は前のコマンドの実行結果が処理された後、アプリケーションがそのコマンドを再送できるようにします。バージョン 11.1 で ct_send、ct_cursor、ct_bind の機能が強化され、ct_setparam ルーチンが追加されたことによって、バッチ処理を行うアプリケーションは、同一のサーバ・コマンドを繰り返して実行するときにコマンドを再送してバインドを再使用できます。この新しい機能により、ct_bind、ct_command、ct_cursor、ct_param に対する冗長な呼び出しをなくすことができます。

アプリケーションは次のようにしてコマンドを再送します。

- 必要な場合、アプリケーションはパラメータ・ソース変数の値を変更します。

  コマンドがパラメータを必要とする場合は、アプリケーションは ct_param を使用して値を渡すのではなく ct_setparam を使用してパラメータ・ソース変数を定義します。ct_param を使用して渡す入力パラメータ値は、コマンドを再送するときに変更することはできません。

- アプリケーションは、前のコマンドの実行結果を処理した後、コマンド構造体で新しいコマンドが開始される前に、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) によって開始されたデータ送信コマンド
- ct_command(CS_SEND_BULK_CMD) によって起動されるバルク送信コマンド
- カーソル更新またはカーソル削除以外の ct_cursor カーソル・コマンド
- 即時実行コマンドまたは準備文を実行するコマンド以外の ct_dynamic コマンド

## 使用するコマンド・タイプの決定

アプリケーションに適したコマンド・タイプを決定するためのガイドラインについては、『Open Client Client-Library/C プログラマーズ・ガイド』の「第 5 章 コマンド・タイプの選択」を参照してください。

# 接続マイグレーション

Open Client では、接続マイグレーション・プロトコルを認識するサーバへの接続に対して、接続マイグレーションをサポートしています。ログインが完了した後にクライアント接続を別のサーバに移動することができます。接続マイグレーションは、CS_PROP_MIGRATABLE プロパティを使用すると有効になります。このプロパティのデフォルトは CS_TRUE であり、このプロパティは ct_config と ct_con_props の両方で有効です。

---

**注意**　DB-Library は接続マイグレーションをサポートしていません。

---

# デバッグ

次の環境変数を使用すると、修正と再リンクを行うことなく CT-Library アプリケーションをデバッグできます。

- SYBOCS_DEBUG_FLAGS － 特定の診断サブシステムを有効にする。複数のデバッグ・オプションを有効にするには、変数にカンマで区切ったフラグのリストを指定する。
- SYBOCS_DEBUG_LOGFILE － 診断を記録するログ・ファイルを指定する。この変数を設定しない場合、メッセージは stdout に書き込まれる。

---

**注意**　デバッグ・フラグが *devlib* ライブラリを必要とする場合、SYBOCS_DEBUG_LOGFILE を使用するときでも、*devlib* ライブラリが必要とされます。devlib ライブラリを必要とする ct_debug フラグ・パラメータについては、「ct_debug」(496 ページ ) を参照してください。

---

これらの変数を使用するには、CT-Library アプリケーションを呼び出す前に設定します。たとえば、UNIX では次のようにします。

```
% setenv SYBOCS_DEBUG_FLAGS CS_DBG_SSL, CS_DBG_PROTOCOL
% setenv SYBOCS_DEBUG_LOGFILE libsybfssl.log
% ./isql -U sa -P -S my_ssl_server
% more libsybfssl.log
```

Windows の場合：

```
C:¥> set SYBOCS_DEBUG_FLAGS=CS_DBG_SSL
C:¥> set SYBOCS_DEBUG_LOGFILE=.¥libsybfssl.log
C:¥> isql -Usa -P -S my_ssl_server
C:¥> type libsybfssl.log
```

## デバッグの有効化

表 2-20 接続のデバッグ・オプションを設定するキーワードを示します。

**表 2-20：デバッグ・オプションのための設定ファイルのキーワード**

| キーワード | 値 |
|---|---|
| CS_DBG_FILE | テキスト・フォーマットのデバッグ情報を保管するファイルの名前を指定する文字列。 |
| CS_DBG_PROTOCOL_FILE | この ct_debug パラメータは、*devlib* ライブラリを使用せずに設定できる。接続に対してパラメータが設定されていない場合、mktemp が呼び出され、プロトコル・パケットのダンプ先となるユニークなファイル名が生成される。mkstemp に渡されるプレフィックス文字列は、*capture* である。Ribo を使用すると、結果として生成されるプロトコル・ファイルを復号化できる。 |
| CS_PROTOCOL_FILE | バイナリ・フォーマットのデバッグ情報を保管するファイルの名前を指定する文字列。 |
| CS_DEBUG | カンマで区切られたデバッグ・フラグのリストを指定する文字列。 |

CS_DEBUG は、CS_DBG_FILE ファイルに書き込まれるデータを指定します。値として、ct_debug の *flag* パラメータのビットマスクに対応するフラグのリストを指定できます。これらのデバッグ・フラグの意味については、「ct_debug」(496 ページ ) を参照してください。

使用可能なフラグは次のとおりです。

- CS_DBG_ALL
- CS_DBG_API_LOGCALL
- CS_DBG_API_STATES
- CS_DBG_ASYNC
- CS_DBG_DIAG
- CS_DBG_ERROR
- CS_DBG_MEM
- CS_DBG_NETWORK
- CS_DBG_PROTOCOL
- CS_DBG_PROTOCOL_FILE
- CS_DBG_PROTOCOL_STATES
- CS_DBG_SSL

# ディレクトリ・サービス

「**ディレクトリ**」はシステム情報を「**ディレクトリ・エントリ**」として保管し、論理名を各エントリと関連付けます。各ディレクトリ・エントリは、ユーザ、サーバ、またはプリンタなどのネットワーク・エンティティについての情報を保持します。ディレクトリはこの情報を編成し、ネットワーク・エンティティのロケーションが変わったときに、アプリケーションを修正しなくてすむようにします。

「**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、ディレクトリ名エントリの作成、修正、取得を管理します。

## ディレクトリ・サービスのプロバイダとドライバ

ディレクトリ・ドライバの設定により、クライアント・アプリケーションのデフォルト・ディレクトリ・ソースが決まります。ディレクトリは次のいずれかによって提供されます。

- ローカル・ホスト・マシン上のオペレーティング・システム・ファイルである Sybase の「**interfaces ファイル**」。明示的に設定されていない場合は、*interfaces* がデフォルトになります。
- DCE/CDS (Distributed Computing Environment Cell Directory Services)、LDAP (Lightweight Directory Access Protocol)、Windows レジストリなどの、ネットワークベースのディレクトリ・サービス・ソフトウェア。

「ディレクトリ・サービス・プロバイダ」(132 ページ ) を参照してください。

ディレクトリ・ドライバの設定方法については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

### ネットワーク・ベースのディレクトリ・サービス

分散ディレクトリ・サービスによって、Client-Library と Server-Library は Sybase の interfaces ファイルではなく、ネットワーク上の「*ディレクトリ*」をサーバ・アドレス情報のソースとして使用できます。ネットワーク・ディレクトリ・サービスを使用することで、多数のクライアント・マシンを含む環境の管理を単純化することができます。

ネットワーク・ベースのディレクトリを使用する場合は、ネットワーク・ディレクトリ・サービスと通信するための Sybase ディレクトリ・ドライバが必要です。Client-Library アプリケーションの場合、ct_connect と ct_ds_lookup 呼び出しが使用するディレクトリ・ソースは、CS_DS_PROVIDER 接続プロパティで指定します。

Client-Library の ct_ds_lookup、ct_ds_objinfo、ct_ds_dropobj の各ルーチンは、ディレクトリをブラウズするルーチンです。これらのルーチンを使用すると、アプリケーションはディレクトリまたは *interfaces* ファイルから使用可能なサーバを探すことができます。

## LDAP

Lightweight Directory Access Protocol (LDAP) は、ディレクトリ・リストへのアクセスに使用します。ディレクトリ・リストやサービスは、ネットワーク上のユーザとリソースの名前、プロファイル情報、マシン・アドレスを提供します。ユーザ・アカウントとネットワーク・パーミッションを管理するのに、これを使用できます。

LDAP サーバは一般的には階層構造で、高速なリソースの検索ができます。従来の Sybase *interfaces* ファイル (Windows では *sql.ini*) の代わりに、LDAP を使用して Sybase サーバの情報を保管したり取得したりできます。

LDAP サービスは、どのようなタイプでも ( 実際のサーバであっても、その他の LDAP サービスへのゲートウエイであっても )、LDAP サーバと呼ばれます。LDAP ドライバは LDAP クライアント・ライブラリを呼び出して、LDAP サーバへの接続を確立します。LDAP ドライバとクライアント・ライブラリは、暗号化を有効にするかどうかなどの通信プロトコル、およびクライアントとサーバの間で交換されるメッセージのコンテンツを定義します。メッセージとは、データ・フォーマット情報も含めたクライアントの読み込み、書き込み、クエリ、サーバ応答などの要求です。

LDAP ドライバが LDAP サーバに接続すると、サーバは、匿名アクセスおよびユーザ名とパスワード認証の、2 つの認証方法をベースとした接続を確立します。

- 匿名アクセス － 認証情報を必要としないため、属性を設定する必要がありません。匿名アクセスは、一般には読み取り専用権限に使用します。

- ユーザ名とパスワード － LDAP URL の拡張機能として *libtcl.cfg* ファイル (64 ビットのプラットフォームでは *libtcl64.cfg* ファイル ) で指定するか、Client-Library への属性呼び出しで設定できます。Ct-Lib を介して LDAP サーバに渡されるユーザ名とパスワードは、Adaptive Server Enterprise へのログインに使用されるユーザ名とパスワードとは別のものです。Sybase では、ユーザ名とパスワード認証を使用されることを強くおすすめします。

## OpenLDAP

OpenLDAP は、LDAP のオープンソース・バージョンです。OpenLDAP ライブラリを使用する Open Client と Open Server の各プラットフォームのリストについては、『Open Server および SDK 新機能』( 各 Microsoft Windows、Linux、UNIX、Mac OS X 版 ) を参照してください。設定の詳細については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

## SSL/TLS

Open Client と Open Server 15.0 ESD #7 以降では、アプリケーションと LDAP サーバとの間に暗号化された (SSL) 接続を確立できます。この暗号化された接続は、次の 2 つのうちいずれかの方法で設定されます。

- LDAPS － LDAP ディレクトリ・サーバのセキュア・ポート ( 通常はポート 636) に接続する。この方法は、LDAP over SSL とも呼ばれ、非標準だが広くサポートされている。
- StartTLS － 既存の標準接続 ( 通常はポート 389 を使用 ) を Transport Layer Security を使用するセキュア接続にアップグレードする。この方法が可能なのは、接続に LDAPv3 が使用される場合のみ。

SSL/TLS ネゴシエーションの間、LDAP サーバは証明書を送信して身元を証明します。クライアントは、この証明書が信頼された認証局 (CA: Certificate Authority) によって署名されたことを確認します。信頼された CA のリストは、信頼されたルート・ファイル *trusted.txt* に保管されています。このファイルは、*$SYBASE/config* かまたは CS_PROP_SSL_CA プロパティに格納される別のファイルの場所にあります。

LDAP サーバの認証が正常に完了すると、クライアントと LDAP サーバは引き続き SSL ハンドシェイクを行って暗号化された接続を確立します。いったん開始されると、LDAPS で確立された接続と StartTLS で確立された接続との間に違いはありません。ただし、LDAPS では LDAP サーバ用の独立したリスナが必要とされます。

詳細については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

### LDAP ディレクトリ・サーバの lookup 時間制限

LDAP サーバがハングしていたり、それ以外の理由で LDAP サーバを使用できないために、LDAP ディレクトリ・サーバへの接続やディレクトリ検索を完了できない場合があります。

CS_DS_TIMELIMIT を設定することにより、LDAP ディレクトリ・サーバへの接続または検索の失敗に対して時間制限を指定できます。CS_DS_TIMELIMIT を設定しない場合、LDAP ディレクトリ・サーバ検索のデフォルトの時間制限としてログインのタイムアウト値が使用されます。CS_DS_TIMELIMIT の詳細については、表 2-30 を参照してください。

**再試行オプションと遅延オプション**

再試行オプションは、最初の接続試行の失敗またはタイムアウトの後、LDAP ディレクトリ・サーバへの検索接続を再試行する回数を指定します。遅延オプションは、失敗した試行と新しい再試行の間の待機時間 ( 秒 ) です。これらのオプションは両方とも、*libtcl.cfg* で設定し、指定された LDAP ディレクトリ・サーバにのみ適用されます。次に例を示します。

```
[DIRECTORY]
myldap=libsybdldap.so retry=3 delay=5
ldap://nlnognix/dc=sybase,dc=com????bind...
```

デフォルトでは、これらのオプションは両方とも 0 になっています。

### Microsoft Active Directory 用 LDAP

Sybase は、Microsoft Active Directory 用 LDAP をサポートしています。これは、ネットワークを一元化しネットワーク・リソースについての情報を保管するためのディレクトリ・サービスです。ディレクトリ・スキーマのインポートと Sybase サーバ・エントリ用コンテナの作成の詳細については、「Microsoft Active Directory のスキーマと名前構文」(123 ページ ) を参照してください。

## アプリケーションでのディレクトリの使い方

Client-Library アプリケーションはサーバに接続するためにディレクトリを必要とします。アプリケーションが ct_connect を呼び出すと、Client-Library はディレクトリ内からサーバ名を検索して、サーバに接続するために必要な情報を読み込みます。

アプリケーションは Sybase によって定義されているエントリについても ct_ds_lookup を呼び出すことによってディレクトリを検索します。たとえば、アプリケーションは ct_ds_lookup を呼び出して、使用可能なサーバを検索したり、特定のサーバのステータスを検査したりできます。

## ディレクトリの編成

各種ベンダがディレクトリ・サービスを提供しているので、各ディレクトリにエントリを編成して保管する方法もさまざまです。

ディレクトリは、フラット構造か階層構造のどちらかです。階層構造を使用すると、関連するエントリを親エントリの下位の個別の論理グループに結合できます。フラット構造では、ディレクトリ内のすべてのエントリは 1 つの論理グループになります。

階層構造は上下が逆になった枝構造のようなものです。「ルート」エントリは最上位にあり、他のすべてのエントリの「始点」になります。「親」エントリは、関連するエントリの論理グループを表します。下位に何もない親エントリを「リーフ」エントリと呼びます。

ディレクトリ構造では、各エントリはユニークに識別されるように、「**完全に修飾された名前**」を持ちます。エントリは、同じ親ノードを持つエントリ間でだけユニークな「**共通名**」を持ちます。

階層ディレクトリ構造では、名前は位置情報を含む必要があります。ルート・ノードの場合だけは、共通名と完全に修飾された名前は同じです。他のエントリの場合、完全に修飾された名前は、エントリの親ノードの完全に修飾された名前とエントリの共通名で構成されます。

フラット・ディレクトリ構造にはルート・ノードは存在しません。それぞれのエントリの完全に修飾された名前はその共通名と同じです。

Sybase の *interfaces* ファイルはフラット・ディレクトリの例です。通常のネットワーク・ベースのディレクトリ・サービスは階層ディレクトリを提供します。

## ディレクトリ・エントリ名のフォーマット

エントリ名は、ディレクトリ・プロバイダ・ソフトウェアによって認識されなければなりません。プロバイダごとに要求される名前構文が異なります。表 2-21 に、完全に修飾された名前の例を示します。

---

**注意**　これらの例は説明のためだけのものです。*interfaces* ファイル以外のディレクトリの名前構文の情報については、システムで使用するネットワーク・ディレクトリ・プロバイダ・ソフトウェアのマニュアルを参照してください。このマニュアル内のエントリの例はすべて架空のものです。

---

**表 2-21：完全に修飾された名前構文の例**

| ディレクトリ・サービス・プロバイダ | 完全に修飾された名前の例 |
|---|---|
| OSF DCE Cell Directory Services (DCE CDS) | */.:/dataservers/sybase/license_data*<br>( セル相対 )<br>*/.../sales.fictional.com/dataservers/sybase/license_data*<br>( グローバル ) |
| Windows レジストリ | *SOFTWARE ¥SYBASE ¥SERVER ¥the_server* |
| LDAP ディレクトリ・サービス | *ldap://host:port/ditbase??scope??bindname=username?password*<br>**注意**　LDAP URL は、1 行で記述してください。 |
| Sybase *interfaces* ファイル | *my_server* |

表 2-21 は、サポートされるディレクトリ・サービス・プロバイダをすべて網羅しているわけではなく、記載されたプロバイダがすべてのプラットフォームでサポートされているとはかぎりません。サポートされているディレクトリ・プロバイダについては、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

## DCE CDS の名前構文

Sybase アプリケーションは、ディレクトリ・プロバイダとして DCE CDS (Cell Directory Service) を使用して DCE ディレクトリにアクセスします。

DCE CDS では、ディレクトリ・ネーム・スペースはセルに分割されます。各セルは、ネットワーク・リソースとそのユーザを管理するための管理用ドメインとして使用されます。CDS では、完全に修飾された名前はセルに相対的なものでも、グローバルに修飾されるものでもかまいません。

- セルに相対的に修飾された名前は、特別なトークン「/.:」で始まります。下位ノードの共通名は、( 左から右に ) 順番にリストされ、共通名はそれぞれスラッシュ (/) で親と区切られます。次の例はセルに相対的に修飾された名前を示します。

  ```
  /.:/eng/license_data
  ```

- グローバルに修飾された名前は、特別なトークン「/...」で始まります。「/...」の後に、DCE セルの DNS (Domain Name Service) 名が続きます。名前の残りの部分は、左から右の順にスラッシュで区切られたセル・ルートの下位ノードで構成されます。次の例はグローバルに修飾された名前を示します。次の例では、「sales.fictional.com」によってエントリが含まれるセルが識別されます。

  ```
  /.../sales.fictional.com/dataservers/license_data
  ```

## Windows レジストリの名前構文

Windows レジストリは階層構造を構成し、そのノードは「キー」と呼ばれています。下位ノードの共通名は、( 左から右に ) 順番にリストされ、共通名はそれぞれ円記号 (Γ) で親と区切られます。レジストリ領域はマシンごとにローカルに存在しますが、エントリは、完全に修飾された名前にマシン名を含めることによって別のマシンのレジストリからも読み込むことができます。

次の例は、ローカル・レジストリ内のエントリに対して完全に修飾された名前を示します。

```
SOFTWARE¥SYBASE¥SERVER¥the_server
```

次の例は、マシン queenbee のレジストリ内のエントリを指定しています。

```
queenbee:SOFTWARE¥SYBASE¥SERVER¥the_server
```

Sybase ディレクトリ・エントリのエントリ名はすべて、キー「￥HKEY_LOCAL_MACHINE￥」に対して相対的な位置に配置されます。

レジストリ・エントリは大文字と小文字を区別しません。

## LDAP ディレクトリ・サービスの名前構文

*libtcl.cfg* ファイルと *libtcl64.cfg* ファイル ( 総称は *libtcl*.cfg* ファイル ) は、*interfaces* ファイルと LDAP ディレクトリ・サービスのどちらを使用するかを指定します。*libtcl*.cfg* ファイルに LDAP が指定してある場合は、サーバ接続時に -I パラメータを渡すことによってアプリケーションが明示的に *libtcl*.cfg* ファイルを上書きしないかぎり、*interfaces* ファイルは無視されます。

*libtcl*.cfg* ファイルを使用して、LDAP サーバへの接続を認証するための LDAP サーバ名、ポート番号、DIT ベース、ユーザ名、パスワードを指定します。LDAP ディレクトリ・サービスは、*libtcl*.cfg* ファイル内の [DIRECTORY] セクションで URL によって指定されています。

次に例を示します。

```
[DIRECTORY]
ldap=libsybdldap.so
ldap://huey:11389/dc=sybase,dc=com??
     one????bindname=cn=Manager,dc=sybase,dc=com secret
```

表 2-22 に、*ldapurl* 変数のキーワードの定義を示します。

**表 2-22：ldapurl 変数**

| キーワード | 説明 | デフォルト | CS_* プロパティ |
|---|---|---|---|
| *host* ( 必須 ) | LDAP サーバを実行しているマシンのホスト名または IP アドレス | なし | |
| *port* | LDAP サーバが受信に使用しているポート番号 | 389 | |
| *ditbase* ( 必須 ) | デフォルトの DIT ベース | なし | CS_DS_DITBASE |
| *username* | 認証するユーザの DN ( 識別名 ) | NULL ( 匿名認証 ) | CS_DS_PRINCIPAL |
| *password* | 認証されるユーザのパスワード | NULL ( 匿名認証 ) | CS_DS_PASSWORD |

次の場所に、Sybase のすべての LDAP ディレクトリ・スキーマのリストがあります。

- UNIX の場合 － *$SYBASE/$SYBASE_OCS/config*
- Windows の場合 － *%SYBASE%\%SYBASE_OCS%\ini*
  同じディレクトリに、*sybase-schema.conf* と呼ばれるファイルもあります。このファイルには、同じスキーマですが、Netscape 固有の構文のものがあります。

LDAPS を使用して LDAP サーバとの暗号化された接続を作成するには、次のようにします。

```
ldap=libsybdldap.so
ldaps://huey:636/dc=sybase,dc=com????
bindname=cn=Manager,dc=Sybase,dc=com?secret
```

ldaps:// を使用してポート番号を指定しない場合、ポート番号 636 がデフォルトで使用されます。

標準の LDAP リスナを使用、アップグレードして、暗号化された接続を作成するには、次のようにします。

```
ldap=libsybdldap.so starttls
ldap://huey:389/dc=sybase,dc=com????
bindname=cn=Manager,dc=Sybase,dc=com?secret
```

ldap:// を使用してポート番号を指定しない場合、ポート番号 389 がデフォルトで使用されます。

### Microsoft Active Directory のスキーマと名前構文

Microsoft Active Directory で使用するディレクトリ・スキーマは、*sybase.ldf* です。ADAM インストール環境で提供されている ldifde.exe コマンドを使用して、*sybase.ldf* を Active Directory (AD) インスタンスまたは Active Directory Application Mode (ADAM) インスタンスにインポートできます。ディレクトリ・スキーマをインポートするには、次の構文を使用して ADAM インストール環境から ldifde.exe コマンドを実行します。

```
ldifde -i -u -f sybase.ldf -s server:port -b username
domain password -j .-c "cn=Configuration,dc=X"
#configurationNamingContext
```

スキーマを正常に Active Directory にインポートしたら、Sybase サーバ・エントリ用のコンテナを作成し、そのコンテナと子オブジェクトに適切な読み込みと書き込みのパーミッションを設定できます。

たとえば、相対識別名 (RDN)「CN=SybaseServers」をドメイン「mycompany.com」の Active Directory ルートに作成して、Sybase サーバ・エントリ名の保管と検索を行います。このコンテナのルート識別名 (rootDN) は、次のように *libtcl.cfg* ファイルに反映されます。

```
ldap=libsybdldap.dll ldap://localhost:389/
cn=SybaseServers,dc=mycompany,dc=com??...
```

Sybase サーバ・エントリの追加と修正を行うために、Active Directory にアカウント名「Manager」、パスワード「secret」で専用のユーザ・アカウントを作成する場合、*libtcl.cfg* ファイル内の完全なエントリは、次のようになります。

- Windows の場合

```
ldap=libsybdldap.dll
ldap://localhost:389/cn=SybaseServers,dc=mycompany,
dc=com????bindname=cn=Manager,cn=Users,dc=mycompay,
dc=com?secret
```

- UNIX の場合

```
ldap=libsybdldap.so
ldap://myADhost:389/cn=SybaseServers,dc=mycompany,
dc=com????bindname=cn=Manager,cn=Users,dc=mycompay,
dc=com?secret
```

適切な読み込みと書き込みのパーミッションを設定すると、Sybase ユーティリティ・プログラム (dscp や dsedit など ) を使用して、Active Directory 内の Sybase サーバ・エントリの保管、表示、修正を行えるようになります。

## interfaces ファイルの名前構文

*interfaces* ファイルはフラット・ディレクトリです。*interfaces* ファイル・エントリの完全に修飾された名前は共通名と同じです。「interfaces ファイル」(157 ページ ) を参照してください。

## DIT ベースによるエントリの配置

「**DIT (Directory Information Tree)** ベース」は部分的なエントリ名を修飾するために使用するディレクトリ・ツリーの中間的なノードです。アプリケーションの DIT ベース設定は、概念的には階層ファイル・システム内のアプリケーションの現在の作業ディレクトリに類似しています。

*interfaces* ファイル以外のディレクトリ・ソースの場合、アプリケーションでは、CS_DS_DITBASE 接続プロパティを設定して DIT ベースを指定できます ( 「ディレクトリ検索のベース」(128 ページ ) を参照してください )。

ct_connect は DIT ベースを使用して部分的なサーバ名を解決します。アプリケーションは次の 2 つの方法のいずれかで ct_connect のサーバ名を指定できます。

- 完全に修飾された名前の指定
- CS_DS_DITBASE 接続プロパティの設定、および CS_DS_DITBASE ノードへの相対的な名前の指定

ディレクトリ・サービス・プロバイダには、エントリが完全に修飾されていることを示すために特別な名前構文を提供しているものもあります。これらのディレクトリ・プロバイダを使用すると、現在の DIT ベースはアプリケーションによって上書きされます。

次の項では、DIT ベースを部分名と結合する方法の例を示します。この規則はディレクトリ・サービス・プロバイダごとに異なります。使用するディレクトリ・サービス・プロバイダでの方法がここに記載されていない場合は、使用するプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

### DCE CDS の DIT ベース

DCE CDS をディレクトリ・プロバイダとして使用する場合は、DIT ベースとしてセルに相対的な名前またはグローバル名を使用できます。グローバル名を使用する場合は、セルを完全に識別できる十分な情報が必要です。

次の 2 つの例は、DCE CDS 用の DIT ベースの設定を示します。最初の例は現在の DCE セル内の DIT ベースを示します。

```
/.:/dataservers
```

2 つ目の例は、グローバル名を指定することによって、セル `sales.fictional.com` の DIT ベースを示します。

```
/.../sales.fictional.com/dataservers
```

次の例は、(*server_name* パラメータとして ) ct_connect に渡すことができる部分名を示します。

```
sybase/test_server
```

ct_connect は、DIT ベースと *server_name* の値を次のように結合します。

```
dit_base_value/server_name
```

次に例を示します。

```
/.:/dataservers/sybase/test_server
```

または

```
/.../sales.fictional.com/dataservers/sybase/test_server
```

Client-Library は DIT ベースにスラッシュ (/) と *server_name* 値を追加します。DIT ベースはスラッシュで終了することはできません。*server_name* 値はスラッシュで始めることはできません。

*server_name* が完全に修飾された名前であることを示す特別な構文を含んでいる場合は、Client-Library は DIT ベースを無視します。この構文は次のとおりです。

- セルに相対的に修飾された名前 (*server_name* は「/.:」で始まります )。
- グローバルに修飾された名前 (*server_name* は「/...」で始まります )。

どちらの場合も、*server_name* は完全に修飾された名前であるとみなされ、ct_connect は DIT ベースを無視します。

DCE CDS ディレクトリ・ドライバのデフォルト DIT ベースは次のとおりです。

```
/.:/subsys/sybase/dataservers
```

このデフォルトはディレクトリ・ドライバの設定で上書きできます。設定されているデフォルトを上書きするには、ct_con_props を呼び出して CS_DS_DITBASE プロパティを設定します。

### Windows レジストリの DIT ベース

接続のディレクトリ・サービス・プロバイダとしてレジストリを使用する場合は、ct_connect は DIT ベース値に円記号「Ɪ」と *server_name* 値を付加します。DIT ベースは円記号で終了することはできません。部分名を表す *server_name* 値は円記号で始めることはできません。

次に Windows レジストリの DIT ベースの例を示します。

```
SOFTWARE¥SYBASE¥SERVER
```

次の例は、ct_connect に付ける部分名を示します。

```
dataserver¥fin_data
```

これらを結合すると次のようになります。

```
SOFTWARE¥SYBASE¥SERVER¥dataserver¥fin_data
```

レジストリ・ディレクトリ・ドライバのデフォルト DIT ベースは次のとおりです。

```
SOFTWARE¥SYBASE¥SERVER
```

このデフォルトはディレクトリ・ドライバの設定で上書きできます。設定されているデフォルトを上書きするには、ct_con_props を呼び出して CS_DS_DITBASE プロパティを設定します。

DIT ベース値で始まる名前は完全に修飾された名前であるとみなされます。たとえば、DIT ベースが「SOFTWAREƖSYBASEƖSERVER」である場合には、次の名前は完全に修飾された名前になります。

```
SOFTWARE¥SYBASE¥SERVER¥debug¥fin_data
```

DIT ベース・ノードはすべて、「ƖHKEY_LOCAL_MACHINEƖ」キーに対して相対的な位置に配置されます。

別のマシンのレジストリから DIT ベース・ノードを指定するには、DIT ベース値にマシン名とコロン (:) を含めます。たとえば、次の DIT ベース値はマシン *queenbee* のレジストリを指します。

```
queenbee:SOFTWARE¥SYBASE¥SERVER
```

### interfaces ファイルの DIT ベース

CS_DS_DITBASE プロパティは、接続のディレクトリ・ソースが *interfaces* ファイルである場合にはサポートされません。

## ディレクトリ・エントリの表示

Client-Library を使用すると、ディレクトリ・コールバック・ルーチンをインストールするアプリケーションをコーディングして、ct_ds_lookup と ct_ds_objinfo を呼び出すことによってディレクトリ・エントリを表示できます。詳細については、『Open Client Client-Library/C プログラマーズ・ガイド』の「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。

## ディレクトリ・オブジェクト

ディレクトリ・オブジェクトの属性は、そのディレクトリ・オブジェクトの性質によって決まります。Sybase ディレクトリ・オブジェクトは Server、OID は CS_OID_OBJSERVER です。「サーバ・ディレクトリ・オブジェクト」(319 ページ ) を参照してください。

## ディレクトリ・サービスのプロパティ

次のプロパティは、アプリケーションによるディレクトリ・サービスの使用を制御します。

### ディレクトリ・サービスによるキャッシュの使用

CS_DS_COPY は、接続のディレクトリ・サービス・プロバイダがキャッシュ上の情報を使用してディレクトリ内の情報の要求に応ずるかどうかを決定します。このプロパティをサポートするディレクトリ・ドライバのデフォルトは CS_TRUE であり、キャッシュ上の情報を使用できます。

すべてのディレクトリ・サービス・プロバイダがキャッシュをサポートするとはかぎりません。アプリケーションは ct_con_props(CS_SUPPORTED) を呼び出すことによって、現在のディレクトリ・ドライバがキャッシュをサポートしているかどうか調べます。

---

**注意**　Client-Library がキャッシュをサポートするディレクトリ・サービス・プロバイダを使用している場合を除き、CS_DS_COPY を設定、クリア、または取得できません。

---

一部のディレクトリ・サービス・プロバイダでは、「DSA (Directory Server Agent)」と「DUAs (Directory User Agent)」を使用して分散モデルをサポートしています。DSA はディレクトリを管理して DUA からの要求に応答するプログラムです。DUA はそれぞれのマシン上で実行され、アプリケーションの要求を DSA に送信して、アプリケーションにその応答を転送します。

ディレクトリのキャッシュによって、DUA は DSA に要求を送らなくても最新の読み込み情報のキャッシュ・コピーを提供できます。このためディレクトリ要求処理を高速化できます。

ローカル・コピーを使用すると高速化できますが、実際のディレクトリに対してクエリを実行すれば、アプリケーションがディレクトリ・エントリに対する最新の変更を受け取れることが保証されます。

### ディレクトリ検索のベース

CS_DS_DITBASE はディレクトリの検索を開始するディレクトリ・ノードを指定します。このノードは DIT ベースと呼ばれます。

---

**注意**　Client-Library が *interfaces* ファイルではなくネットワーク・ベースのディレクトリ・サービスを使用する場合を除き、CS_DS_DITBASE を設定、クリア、または取得できません。

---

デフォルト DIT ベース値は次のように指定します。

- ディレクトリ・ドライバの設定で指定します。

- 設定にデフォルト DIT ベース値を指定しない場合は、ドライバ固有のデフォルトで指定されます。

ディレクトリ・ドライバの設定については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

DIT ベース値は、Client-Library が使用するディレクトリ・サービスの名前構文を持つ完全に修飾された名前にしてください。さらに、それぞれのドライバとプロバイダの組み合わせには、完全に修飾された名前と部分的に修飾された名前を結合するための異なった規則があります。詳細については、「ディレクトリ・エントリ名のフォーマット」(120 ページ ) を参照してください。

### ディレクトリ・サービスのエイリアスの拡張

CS_DS_EXPANDALIAS は、接続のディレクトリ・サービス・プロバイダがディレクトリを検索するときにエイリアス・エントリを拡張するかどうかを決定します。このプロパティをサポートするディレクトリ・ドライバに対するデフォルトは CS_TRUE であり、エイリアス・エントリは拡張されます。

すべてのディレクトリ・サービス・プロバイダがエイリアスをサポートしているとはかぎりません。アプリケーションは ct_con_props(CS_SUPPORTED) を呼び出すことによって、現在のディレクトリ・ドライバがこのプロパティをサポートしているかどうか調べます。

---

**注意**　Client-Library がエイリアス・エントリをサポートするディレクトリ・ドライバを使用している場合を除き、CS_DS_EXPANDALIAS を設定、クリア、または取得できません。

---

一部のディレクトリ・サービス・プロバイダではディレクトリ・エイリアス・エントリを作成できます。エイリアス・エントリはプライマリ・エントリへのリンクを含んでいます。エイリアスを使用すると、プライマリ・エントリを異なったロケーションの 1 つまたは複数のエントリとして使用できます。

CS_DS_EXPANDALIAS が CS_TRUE である場合は、ディレクトリ・サービス・プロバイダはディレクトリを検索するときにエイリアスのリンクをたどることができます。値が CS_FALSE である場合は、エイリアス・エントリのリンクをたどることはできません。

**警告！**エイリアス・エントリを含んだディレクトリには、エイリアス・リンクの結果として循環的な検索パスが含まれる場合があります。CS_DS_EXPANDALIAS が有効な場合、ディレクトリ・ツリーに循環的な検索パスが含まれていると、ct_ds_lookup によって開始された検索が無限に繰り返し実行されます。

### ディレクトリ・サービスのフェールオーバ

CS_DS_FAILOVER は、現在のディレクトリ・ドライバをロードできない場合または現在のディレクトリ・サーバを使用できない場合に、Client-Library が *libtcl.cfg* ファイル内で次のディレクトリ・ドライバ・エントリ、そして最終的には *interfaces* ファイルにフェールオーバするのかどうかを決定します。デフォルトの CS_DS_FAILOVER 値は CS_TRUE であり、現在のディレクトリ・ドライバをロードできない場合は、Client-Library は暗黙的にフェールオーバします。

Client-Library は特に論理サーバ名をネットワーク・アドレスにマップするためのディレクトリを必要とします。ディレクトリは、Sybase の *interfaces* ファイル、または DCE Cell Directory Services (CDS) などのネットワーク・ベースのディレクトリ・サービスのどれでもかまいません。

*interfaces* ファイル以外のディレクトリ・ソースを使用するためには、Client-Library はディレクトリ・ドライバを必要とします。

フェールオーバは、アプリケーションが *interfaces* ファイルではなくネットワーク・ベースのディレクトリ・サービスを使用することを要求する ( またはデフォルトで使用する ) 場合に発生します。デフォルトでは、Client-Library がディレクトリ・ドライバをロードできない場合には、*libtcl.cfg* ファイル内で次のディレクトリ・ドライバ・エントリへのフェールオーバが発生し、最終的には *interfaces* ファイルにフェールオーバします。アプリケーションは CS_DS_FAILOVER を CS_FALSE に設定してフェールオーバを防ぐことができます。

ディレクトリ・サービスのフェールオーバが許可されない場合に、Client-Library が指定されたディレクトリ・ドライバをロードすると、その接続のディレクトリ・ソースが不定になります。この場合は、ディレクトリ・アクセスを必要とする以降のアクションはすべて失敗します。失敗するのは次に示すアクションです。

- CS_DS_FAILOVER または CS_DS_PROVIDER 以外のすべての CS_DS_ プロパティを取得、設定、またはクリアするための ct_con_props に対する呼び出し。
- CS_DS_PROVIDER プロパティを取得またはクリアするための ct_con_props に対する呼び出し。
- ct_connect または ct_ds_lookup に対する呼び出し。

Client-Library がディレクトリ・ドライバをロードする場合の説明については、「ディレクトリ・サービス・プロバイダ」(132 ページ ) を参照してください。

Sybase フェールオーバ・オプションの詳細については、「高可用性フェールオーバ」(153 ページ ) を参照してください。

### ディレクトリ・サービスのパスワード

CS_DS_PASSWORD は、CS_DS_PRINCIPAL として指定されるプリンシパル ( ユーザ ) 名を使用して作業するためのディレクトリ・サービスのパスワードを指定します。ディレクトリ・プロバイダによっては、ディレクトリ・エントリに対するアプリケーションのアクセスを管理するために、認証されたプリンシパル ( ユーザ ) 名が必要です。

CS_DS_PRINCIPAL の詳細については、「ディレクトリ・サービスのプリンシパル名」(131 ページ ) を参照してください。

すべてのディレクトリ・サービスがパスワードをサポートするとはかぎりません。アプリケーションは ct_con_props(CS_SUPPORTED) を呼び出すことによって、現在のディレクトリ・ドライバがこのプロパティをサポートしているかどうか調べます。

---

**注意**　このプロパティをサポートするディレクトリ・サービス・プロバイダを Client-Library が使用している場合を除き、CS_DS_PASSWORD を設定、クリア、または取得できません。

---

### ディレクトリ・サービスのプリンシパル名

CS_DS_PRINCIPAL は、CS_DS_PASSWORD として指定されるパスワードを使用して作業するためのディレクトリ・サービスのプリンシパル ( ユーザ ) 名を指定します。ディレクトリ・プロバイダによっては、ディレクトリ・エントリに対するアプリケーションのアクセスを管理するために、認証されたプリンシパル ( ユーザ ) ID が必要です。このプロパティをサポートするドライバのデフォルトは NULL です。

CS_DS_PASSWORD の詳細については、「ディレクトリ・サービスのパスワード」(131 ページ ) を参照してください。

すべてのディレクトリ・サービスが CS_DS_PRINCIPAL をサポートするとはかぎりません。アプリケーションは ct_con_props(CS_SUPPORTED) を呼び出すことによって、現在のディレクトリ・ドライバがこのプロパティをサポートしているかどうか調べます。

---

**注意**　このプロパティをサポートするディレクトリ・サービス・プロバイダを Client-Library が使用している場合を除き、CS_DS_PRINCIPAL を設定、クリア、または取得できません。

---

### ディレクトリ・サービスのランダム・オフセット

デフォルトでは、現在のインストールが中断しないように、CS_DS_RAND_OFFSET は true に設定されます。true に設定されると、CS_DS_RAND_OFFSET がランダム・オフセットから開始され、接続に成功するまでネットワーク・アドレス・リストをスキャンします。ランダム・オフセットは、ネットワーク・アドレス・リストがディレクトリ・サービスから取得されるときに決定されます。

CS_DS_RAND_OFFSET が false に設定された場合、接続試行はネットワーク・アドレス・リストの最初のエントリから開始されます。

CS_DS_RAND_OFFSET は、ct_con_props、ct_config、または *ocs.cfg* を使用して設定できます。

### ディレクトリ・サービス・プロバイダ

CS_DS_PROVIDER は、現在のディレクトリ・サービス・プロバイダの名前を null で終了する文字列として保持します。

Client-Library はドライバ設定ファイルを使用してディレクトリ・サービス・プロバイダ名をディレクトリ・ドライバのファイル名にマップします。大部分のプラットフォームでは、このファイルは *libtcl.cfg* という名前です。このファイルの詳細については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

**デフォルト・ディレクトリ・ドライバのロード**

デフォルト・プロバイダ名は、*libtcl.cfg* ドライバ設定ファイルの [DIRECTORY] セクション内の最初のエントリに相当します。このセクションには、次のような形式のエントリがあります。

```
[DIRECTORY]
 provider_name = driver_file_name init_string
 provider_name = driver_file_name init_string
```

各パラメータの意味は次のとおりです。

- *provider_name* は CS_DS_PROVIDER プロパティの有効な値です。
- *driver_name* はドライバのファイル名です。
- *init_string* はドライバの起動用の設定値です。

システム上にドライバ設定ファイルが存在しない場合、またはファイルに [DIRECTORY] セクションが存在しない場合のデフォルト・プロバイダ名は「InterfacesDriver」であり、Client-Library が *interfaces* ファイルをディレクトリ・ソースとして使用することを示します。

使用しているプラットフォームでのドライバ設定の詳細については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

それぞれの接続の構造体に対して、Client-Library はデフォルト・ディレクトリ・ドライバを次のいずれかの状況でロードします。

- ドライバがまだロードされていない場合、CS_DS_FAILOVER または CS_DS_PROVIDER 以外の CS_DS_ プロパティを取得、設定、またはクリアするために ct_con_props を呼び出すと、デフォルト・ディレクトリ・ドライバがロードされます。
- ドライバがまだロードされていない場合、CS_DS_PROVIDER プロパティを取得するために ct_con_props を呼び出すと、デフォルト・ディレクトリ・ドライバがロードされます。CS_DS_PROVIDER をクリアするための呼び出しでは、常に既存のドライバがアンロードされ、デフォルト・ドライバが再ロードされます。
- ドライバがまだロードされていない場合、ct_connect または ct_ds_lookup を呼び出すと、デフォルト・ディレクトリ・ドライバがロードされます。

Client-Library がディレクトリ・ドライバをロードできない場合には、Client-Library はデフォルトで *interfaces* ファイルにフェールオーバします。アプリケーションは上記のアクションを実行する前に CS_DS_FAILOVER プロパティを設定してこの動作を変更できます。詳細については、「ディレクトリ・サービスのフェールオーバ」(130 ページ ) を参照してください。

**異なるディレクトリ・サービス・プロバイダへの変更**

アプリケーションは、ct_con_props(CS_SET、CS_DS_PROVIDER) を呼び出して、接続のディレクトリ・サービス・プロバイダを変更します。

CS_DS_PROVIDER を設定するときは、有効なディレクトリ・ドライバに新しいプロパティ値をマップする必要があります。マッピングがある場合には、Client-Library は新しいドライバをロードして初期設定を行います。

Client-Library が要求されたドライバをロードできない場合は、CS_DS_FAILOVER プロパティの値およびドライバが以前ロードされたかどうかによって接続のステータスが異なります。

- CS_DS_FAILOVER は、ドライバをロードできない場合または現在のディレクトリ・サーバを使用できない場合に、Client-Library が *libtcl.cfg* ファイル内で次のディレクトリ・ドライバ・エントリにフェールオーバするのかどうかを決定します。Client-Library は、*libtcl.cfg* の最後のエントリに達すると、*interfaces* ファイルにフェールオーバします。詳細については、「ディレクトリ・サービスのフェールオーバ」(130 ページ ) を参照してください。
- アプリケーションが以前に CS_DS_PROVIDER プロパティを設定している場合、またはドライバを必要とする呼び出しをアプリケーションが以前に発行している場合には、接続には事前にロードされているドライバが存在します。デフォルト・ディレクトリ・ドライバをロードする呼び出しのリストについては、「デフォルト・ディレクトリ・ドライバのロード」(132 ページ ) を参照してください。

次の表は、ct_con_props(CS_SET, CS_DS_PROVIDER) に対する呼び出しが失敗した後のディレクトリ・ソースを示します。

| ドライバが事前にロードされているか | CS_DS_FAILOVER | 新しいディレクトリ・プロバイダ |
|---|---|---|
| はい | CS_TRUE | 次の *libtcl.cfg* エントリまたは *interfaces* ファイル |
| はい | CS_FALSE | 以前のドライバに戻る |
| いいえ | CS_TRUE | 次の *libtcl.cfg* エントリまたは *interfaces* ファイル |
| いいえ | CS_FALSE | Undefined ( 未定義 ) |

## ディレクトリ・ドライバの初期設定

ディレクトリ・ドライバがロードされると、Client-Library は対応する設定ファイル・エントリに基づいて DIT ベースのプロパティにデフォルト値を割り当てます。

ドライバの設定方法については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

### ディレクトリ・サービスの検索の深さ

CS_DS_SEARCH は、ディレクトリ検索において、始点から見た検索対象となる階層の深さを制限します。

---

**注意**　プロパティをサポートするディレクトリ・ドライバを Client-Library が使用している場合を除き、CS_DS_SEARCH を設定、クリア、または取得できません。

---

次の表は CS_DS_SEARCH で有効な値を示します。

| 値 | 意味 |
|---|---|
| CS_SEARCH_ONE_LEVEL ( デフォルト ) | 検索は、CS_DS_DITBASE で指定されたノードのすぐ下にあるリーフ・エントリだけを含む。 |
| 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。 |

すべてのディレクトリ・サービスが検索の深さのプロパティをサポートするとはかぎりません。アプリケーションは ct_con_props(CS_SUPPORTED) を呼び出すことによって、現在のディレクトリ・ドライバがこのプロパティをサポートしているかどうか調べます。

---

**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 以外の値に設定できません。

---

検索は CS_DS_DITBASE プロパティの値で示されるディレクトリ・ノードから開始されます。「ディレクトリ検索のベース」(128 ページ ) を参照してください。

### ディレクトリ検索サイズの制限

CS_DS_SIZELIMIT は、ct_ds_lookup によって開始されるディレクトリ検索で返されるエントリの数を制限します。デフォルトは 0 で、サイズの制限がないことを示します。

すべてのディレクトリ・サービス・プロバイダが検索結果のサイズの制限をサポートするとはかぎりません。アプリケーションは ct_con_props(CS_SUPPORTED) を呼び出すことによって、現在のディレクトリ・ドライバがこのプロパティをサポートしているかどうか調べます。

---

**注意** プロパティをサポートするディレクトリ・ドライバを Client-Library が使用している場合を除き、CS_DS_SIZELIMIT を設定、クリア、または取得できません。

---

#### ディレクトリ検索の時間制限

CS_DS_TIMELIMIT は、ディレクトリ検索の完了までの時間制限を秒単位で指定します。デフォルトは 0 であり、時間制限がないことを示します。

すべてのディレクトリ・サービス・プロバイダが検索時間の制限をサポートするとはかぎりません。アプリケーションは ct_con_props(CS_SUPPORTED) を呼び出すことによって、現在のディレクトリ・ドライバがこのプロパティをサポートしているかどうか調べます。

---

**注意** プロパティをサポートするディレクトリ・ドライバを Client-Library が使用している場合を除いて、CS_DS_TIMELIMIT を設定、クリア、または取得できません。

---

# エラー処理

すべての Client-Library ルーチンは、正常終了か失敗かの表示を返します。アプリケーションで、これらのリターン・コードをチェックすることをおすすめします。

## 初期化時のエラー・レポート

この項では、Client-Library アプリケーションの初期化中にエラー情報がどのように返されるかについて説明します。

### *cs_ctx_alloc と cs_ctx_global*

cs_ctx_alloc または cs_ctx_global に対するアプリケーション呼び出しで CS_FAIL が返された場合、詳細なエラー情報が標準エラー (STDERR) とファイル *sybinit.err* に送られます。*sybinit.err* ファイルは、現在の作業ディレクトリに作成されます。

### *ct_init*

ct_init に対するアプリケーション呼び出しで、Net-Library エラーが原因で CS_FAIL が返された場合、詳細なエラー情報が標準エラー (STDERR) とファイル *sybinit.err* に送られます。*sybinit.err* ファイルは、現在の作業ディレクトリに作成されます。

## エラー処理とメッセージ処理

初期化の後、Client-Library アプリケーションでは次の 2 種類のエラーおよび情報メッセージを処理する必要があります。

- 「クライアント・メッセージ」として知られる Client-Library メッセージは、Client-Library によって生成されます。これらのメッセージは、その重大度によって、情報メッセージから致命的なエラーまであります。
- サーバ・メッセージは、サーバによって生成されます。これらのメッセージは、その重大度によって、情報メッセージから致命的なエラーまであります。

Adaptive Server Enterprise メッセージのテキストは、sysmessages システム・テーブル内に保管されます。このテーブ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。

Open Server メッセージのリストについては、『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。

---

**注意**　Client-Library メッセージおよびサーバ・メッセージと CS_MSG_RESULT タイプの結果セットとを混同しないでください。Client-Library およびサーバ・メッセージは、Client-Library とサーバがエラーおよび情報の状態をアプリケーションへ通信する手段となります。アプリケーションは、メッセージ・コールバック・ルーチンで、または ct_diag を使用してインラインで、Client-Library およびサーバのメッセージにアクセスします。一方、メッセージ結果セットは、サーバがアプリケーションへ返すことのできるさまざまなタイプの結果セットの 1 つです。アプリケーションは、メッセージの ID を取得する ct_res_info を呼び出すことにより、CS_MSG_RESULT タイプの結果セットを処理します。

---

## メッセージ処理の 2 つの方法

アプリケーションは、次の 2 とおりの方法のいずれかで、Client-Library およびサーバ・メッセージを処理します。

- メッセージを処理するコールバック・ルーチンをインストールする方法。
- インラインで、Client-Library ルーチン ct_diag を使用する方法。

コールバック方式には、次の利点があります。

- メッセージ処理コードの集約。
- 予期しないエラーを処理する方法の提供：Client-Library は、メッセージが生成されるたびに、適当なメッセージ・コールバックを自動的に呼び出します。このため、アプリケーションが予期しないエラーをトラップできます。メインライン・コードでのエラー処理ロジックだけを使用するアプリケーションは、予期しないエラーを正常にトラップできないことがあります。

インライン・メッセージ処理方式には、アプリケーションが特定の時点でメッセージの有無をチェックできるという利点があります。たとえば、接続を作成しているアプリケーションは、メッセージをチェックする前に、接続に関連するコマンドが与えられるまで待機しようとします。

ほとんどのアプリケーションは、コールバック方式を使用してメッセージを処理します。しかし、コールバックをサポートしていないプラットフォームと言語が組み合わさった環境で動作しているアプリケーションは、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、または、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ことで、どちらの方法を使用するかを示します。

アプリケーションは、異なった接続では異なった方式を使用できます。たとえば、アプリケーションは、コンテキスト・レベルでメッセージ・コールバックをインストールし、2 つの接続を割り付け、ct_diag を呼び出していずれか一方に対して接続インライン・メッセージ処理を初期化できます。もう一方の接続は、その親コンテキストから選択したデフォルト・メッセージ・コールバックを使用します。

アプリケーションは、次のように、インライン方式とコールバック方式との切り替えができます。

- クライアント・メッセージ・コールバックまたはサーバ・メッセージ・コールバックのいずれか一方をインストールすると、インライン・メッセージ処理が無効になります。保存されたメッセージはすべて破棄されます。
- 同様に、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・コールバックのインストールが解除されます。この状態が発生すると、その接続の ct_diag に対する最初の CS_GET 呼び出しで、警告メッセージが取得されます。

適切なタイプのコールバックがインストールされておらず、インライン・メッセージ処理が使用できない場合、Client-Library は、メッセージ情報を廃棄します。

## メッセージ処理のためのコールバックの使用

アプリケーションは、ct_callback を使用してメッセージ・コールバックをインストールします。

Client-Library は、CS_CONNECTION および CS_CONTEXT 構造体にコールバックを記録します。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。ただし、エラーが発生したルーチンは CS_FAIL を返します。

詳細については、「コールバック」(25 ページ ) と 「ct_callback 」(387 ページ ) のリファレンス・ページを参照してください。

## インライン・メッセージ処理

アプリケーションは、ct_diag を呼び出して、接続に対するインライン・メッセージ処理を初期化します。一般の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た直後に ct_diag を呼び出します。

アプリケーションは、コンテキスト・レベルでは ct_diag を使用できません。つまり、アプリケーションは、ct_diag を使用して (CS_CONNECTION ではなく ) CS_CONTEXT をパラメータとするルーチンが生成したメッセージを取得することはできません。これらのメッセージは、インライン・エラー処理を使用しているアプリケーションでは利用できません。

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」(141 ページ ) を参照してください。

CS_DIAG_TIMEOUT プロパティは、Client-Library ルーチンがタイムアウト・エラーを生成したときに、Client-Library が失敗するか、またはリトライするかを制御します。

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。

「ct_diag 」(507 ページ ) を参照してください。

## Client-Library のメッセージ構造体

Client-Library は、次の構造体を使用してメッセージ情報を返します。

- CS_CLIENTMSG － 「Client-Library と SQL 構造体」(81 ページ ) 参照。
- CS_SERVERMSG － 「Client-Library と SQL 構造体」(81 ページ ) 参照。
- SQLCA － 「Client-Library と SQL 構造体」(81 ページ ) 参照。
- SQLCODE － 「Client-Library と SQL 構造体」(81 ページ ) 参照。
- SQLSTATE － 「Client-Library と SQL 構造体」(81 ページ ) 参照。

## CS_EXTRA_INF プロパティ

CS_EXTRA_INF プロパティは、Client-Library がある種の情報メッセージを返すかどうかを決定します。

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。これは、Client-Library が通常では返すことのない情報を SQL 構造体が要求するためです。CS_EXTRA_INF が設定されていない場合、情報が消失することがあります。

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 に設定できます。この場合、標準 Client-Library メッセージとして追加情報が返されます。

返された補足情報には、最後に入力したコマンドによって影響を受けたローの数が含まれます。

## 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 から 1 を引いたバイト数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、アプリケーションで CS_NO_TRUNCATE プロパティを使用して、メッセージをトランケートしないで、連続した長いメッセージにするよう 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 バイトのメッセージは、2 番目の構造体に、と返されます。

Client-Library は、メッセージの最後の部分だけ NULL で終了します。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している場合、アプリケーションは、各メッセージ部分に対して一度ずつ ct_diag を呼び出さなければなりません。

---

**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* フィールドによってレポートされます。Client-Library では、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。

---

## 連続したメッセージ用のメッセージ構造体フィールド

CS_CLIENTMSG および CS_SERVERMSG 構造体の *status* フィールドは、その構造体にメッセージ全体が含まれているのか、またはメッセージの一部分が含まれているのかを示します。

- 連続しているメッセージに関連する *status* 値を次に示します。

| 記号値 | 意味 |
|---|---|
| CS_FIRST_CHUNK | メッセージ・テキストは、メッセージの最初の部分。 |
| CS_LAST_CHUNK | メッセージ・テキストは、メッセージの最後の部分。 |

  - CS_FIRST_CHUNK と CS_LAST_CHUNK が両方ともオンの場合、構造体内のメッセージ・テキストはメッセージ全体です。
  - CS_FIRST_CHUNK と CS_LAST_CHUNK が両方ともオンでない場合、構造体内のメッセージ・テキストはメッセージの中間の部分です。

- CS_CLIENTMSG 構造体の *msgstringlen* フィールドと CS_SERVERMSG 構造体の *textlen* フィールドは、現在対象になっているメッセージ部分の長さを表しています。
- CS_CLIENTMSG 構造体および CS_SERVERMSG 構造体内の他のすべてのフィールドは、各メッセージ部分で繰り返されます。

### 連続したメッセージと拡張エラー・データ

連続しているサーバ・メッセージが、そのメッセージについての拡張エラー・データを持っている場合、アプリケーションは、連続しているメッセージの一部分を処理している間、その拡張エラー・メッセージを取得することができます。しかし、ひとたびアプリケーションが拡張エラー・データを取得すると、取得ができなくなります。「拡張エラー・データ」(143 ページ ) を参照してください。

### 連続しているメッセージと ct_diag

アプリケーションで連続しているエラー・メッセージを使用している場合、ct_diag は、メッセージではなく、メッセージのまとまりに対して動作します。これには、次のような影響があります。

- ct_diag(CS_GET, *index*) 呼び出しは、番号 *index* を持ったメッセージのまとまりを返します。
- ct_diag(CS_MSGLIMIT) 呼び出しは、Client-Library が記録するメッセージの数ではなく、Client-Library が記録するまとまりの数を制限します。
- ct_diag(CS_STATUS) の呼び出しは、現在保管されているメッセージ数ではなく、現在保管されているまとまりの数を返します。

## 拡張エラー・データ

サーバ・メッセージの中には、そのメッセージに関連した「拡張エラー・データ」が入っているものがあります。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 のメッセージについては、補足情報は一般にエラーを引き起こした 1 つまたは複数のカラムです。

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。1 つの拡張エラー・データには、名前およびデータ型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・データの取得が可能ですが、必ずしもそうすることは必要ありません。

## 拡張エラー・データの使用

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・メッセージのメカニズムでは、カラム・レベルの情報はサーバ・メッセージのテキスト内に限って使用できます。拡張エラー・データは、カラム・レベルの情報に簡単にアクセスするための方法をアプリケーションに提供します。

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。既存のローと等しい *au_id* と *title_id* を持つローを入力しようとすると、「重複キー」のメッセージがアプリケーションへと送られます。

このメッセージを受け取ると、アプリケーションは、エンド・ユーザに対して問題のある 1 つまたは複数のカラムを明らかにする必要があります。この結果、ユーザはその問題点を修正することができます。この情報は、メッセージ・テキスト以外の、重複キー・メッセージでは使用できません。この情報は、拡張エラー・データとして使用できます。

## 拡張エラー・データ

すべてのサーバ・メッセージが拡張エラー・データを提供するわけではありません。メッセージで拡張エラー・データを利用できる場合、Client-Library が標準サーバ・メッセージ情報を CS_SERVERMSG 構造体に入れてアプリケーションへ返すときに、CS_SERVERMSG 構造体の *status* フィールドの CS_HASEED ビットを設定します。

拡張エラー・データは、Client-Library が提供する特別な CS_COMMAND 構造体のパラメータ結果セットのフォームで、アプリケーションへ返されます。

拡張エラー・データを取得するために、アプリケーションはパラメータ結果セットを処理します。

## サーバ・メッセージ・コールバックおよび拡張エラー・データ

サーバ・メッセージ・コールバック・ルーチンでは、アプリケーションは、次に示すように、*property* を CS_EED_CMD に設定し、ct_con_props を呼び出して拡張エラー・データを含む CS_COMMAND を取得します。

```
CS_RETCODE       ret;
CS_COMMAND       *eed_cmd;
CS_INT           outlen;

ret = ct_con_props(connection, CS_GET, CS_EED_CMD,
     &eed_cmd, CS_UNUSED, &outlen);
```

ct_con_props は、*eed_cmd* に拡張エラー・データを含む CS_COMMAND へのポインタを設定します。

CS_COMMAND を持つと、コールバック・ルーチンは、通常のパラメータ結果セットとして拡張エラー・データを処理します。また、ct_res_info、ct_describe、ct_bind、ct_fetch、ct_get_data を呼び出して、パラメータの記述、バインド、およびフェッチを行います。コールバック・ルーチンは、必ずしも ct_results を呼び出す必要はありません。

## インライン・エラー処理および拡張エラー・データ

サーバ・メッセージをインラインで処理しているアプリケーションは、次に示すように、*operation* を CS_EED_CMD に設定して ct_diag を呼び出すことにより、拡張エラー・データを持つ CS_COMMAND を取得します。

```
CS_RETCODE       ret;
CS_COMMAND       *eed_cmd;
CS_INT           index;

ret = ct_diag (connection, CS_EED_CMD,
     CS_SERVERMSG_TYPE, index, &eed_cmd);
```

この呼び出しでは、*type* は CS_SERVERMSG_TYPE に、*index* は拡張エラー・データを持つメッセージのインデックスになっていなければなりません。ct_diag は、*eed_cmd* に拡張エラー・データを含む CS_COMMAND へのポインタを設定します。

CS_COMMAND を持つと、アプリケーションは、通常のパラメータ結果セットとして拡張エラー・データを処理し、ct_res_info、ct_describe、ct_bind、ct_fetch、ct_get_data を呼び出して、パラメータの記述、バインド、およびフェッチを行います。アプリケーションは、必ずしも ct_results を呼び出す必要はありません。

## 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。

トランザクション・ステータスを表す記号値の一覧表を次に示します。

**表 2-23： トランザクション・ステータス**

| 記号値 | 意味 |
|---|---|
| CS_TRAN_IN_PROGRESS | トランザクションが進行中である。 |
| 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最後に実行された文が失敗した。 |
| CS_TRAN_FAIL | 最新のトランザクションが失敗した。 |
| CS_TRAN_UNDEFINED | トランザクション・ステータスは現在定義されていない。 |

### メインライン・コードでのトランザクション・ステータスの取得

メインライン・コードでは、アプリケーションは、次のように、*type* に CS_TRANS_STATE を設定して ct_res_info を呼び出すことにより、トランザクション・ステータスを取得します。

```
CS_RETCODE       ret;
CS_INT           outlen;
CS_INT           trans_state;

ret = ct_res_info (cmd, CS_TRANS_STATE,
      &trans_state, CS_UNUSED, &outlen)
```

ct_res_info は、*trans_state* に 表 2-23 (146 ページ ) に示されている記号値のいずれかを設定します。

トランザクション・ステータス情報は、保留中の結果またはオープン・カーソルのある CS_COMMAND 構造体に対してのみ使用できます。つまり、ct_results へのアプリケーションの最終呼び出しが CS_SUCCEED を返した場合は、トランザクション・ステータス情報を利用できます。

ct_results が **result_type* に CS_CMD_DONE、CS_CMD_SUCCEED、または CS_CMD_FAIL を設定した後にかぎり、トランザクション・ステータス情報は正しいと保証されます。

### サーバ・メッセージ・コールバックでのトランザクション・ステータスの取得

拡張エラー・データが利用できる場合にかぎり、アプリケーションは、サーバ・メッセージ・コールバック内でトランザクション・ステータスを取得します。

サーバ・メッセージ・コールバックでは、Client-Library は、メッセージを記述する CS_SERVERMSG 構造体の *status* フィールドの CS_HASEED ビットを設定することにより、拡張エラー・データが使用可能であることを示します。

拡張エラー・データが使用可能である場合、アプリケーションは、次のように現在のトランザクション・ステータスを取得できます。

1. *property* を CS_EED_CMD に設定して ct_con_props を呼び出すことにより、拡張エラー・データを含む CS_COMMAND を取得します。
2. *type* を CS_TRANS_STATE に設定して、ct_res_info を呼び出します。ct_res_info は、その **buffer* パラメータに、表 2-23 (146 ページ ) に示されている記号値のいずれかを設定します。

## サンプル・プログラム

次のサンプル・プログラムとヘッダ・ファイルは、Client-Library と一緒にインストールされます。各ファイルには、ファイルの内容と目的を説明しているヘッダ部分があります。各サンプル・プログラムの詳細については、readme ファイルを参照してください。

| サンプル・プログラム | 説明 |
|---|---|
| *arraybind.c* | ct_command により起動された CS_LANG_CMD とともに配列バインドを使用する方法を示す。 |
| *blktxt.c* | バルク・コピー・ルーチンを使用して、静的データをテーブルにコピーする。 |
| *compute.c* | Transact-SQL コマンドの送信方法と、計算結果および通常結果を処理する方法を示す。 |
| *csr_disp.c* | 読み込み専用カーソルの使い方を示す。 |
| *csr_disp_implicit.c* | スクロール可能なカーソルを使用して pubs2 データベース内の author テーブルからデータを取り出す。1 つのプリフェッチ・バッファと、通常のプログラム変数も使用する。 |

| サンプル・プログラム | 説明 |
|---|---|
| *csr_disp_ scrollcurs.c* | スクロール可能なカーソルを使用して pubs2 データベース内の author テーブルからデータを取り出す。1 つのプリフェッチ・バッファと、通常のプログラム変数も使用する。 |
| *csr_disp_ scrollcurs2.c* | プログラム変数として配列とともにスクロール可能なカーソルを使用し、配列バインドを使用する。1 回の ct_scroll_fetch 呼び出しにより、配列の結果を表示する。 |
| *ctexact.c* | 2 フェーズ・コミットのサンプル・プログラム。 |
| *ctpr.c* | 固定長データの印刷の最大長を指定する。 |
| *ex_alib.c* *ex_amain.c* | Client-Library の上位に非同期レイヤを作成する方法の例を示すルーチンの集まり。 |
| *example.h* | Client-Library サンプル・プログラム用のヘッダ・ファイル。 |
| *exasync.h* | 言語コマンドを送信し、結果を非同期に処理する。*ex_alib.c* と *ex_amain.c* 内の定数とデータ構造用のヘッダ・ファイル。 |
| *exconfig.c* | デフォルトの外部設定ファイル *$SYBASE/$SYBASE_OCS/config/ocs.cfg* を使用して CS_SERVERNAME プロパティ値を設定する方法を示す。 |
| *exutils.c* | 他のすべてのサンプル・プログラムによって使用されるユーティリティ・ルーチンを含み、アプリケーションが上位レベルのプログラムから Client-Library の実装の詳細の一部を隠す方法を示す。 |
| *exutils2.c* | スクロール可能カーソルのサンプル・プログラムで使用するユーティリティ・ルーチンが含まれている。csr_disp_scrollable および csr_disp_scrollable2 のサンプルとともに使用する。 |
| *exutils.h* | *exutils.c* および *exutils2.c* 内のユーティリティ関数用のヘッダ・ファイル。 |
| *firstapp.c* | サーバに接続して select クエリを送信し、ローを出力する。 |
| *getsend.c* | text データを取得して更新する方法を示す。 |
| *id_update.c* | identity_update オプションの使用方法を示す。 |
| *i18n.c* | Client-Library で使用できる国際化機能の一部を示す。 |
| *multithrd.c* | *thrdfunc.c* を使用して、Client-Library でマルチスレッド・クライアント・アプリケーションをコード化する方法を示す。<br>**注意**　このサンプルは、Solaris ネイティブ・スレッド・パッケージと、DCE pthread API で使用するために作成されている。 |
| *rpc.c* | RPC コマンドをサーバに送信して、リモート・プロシージャから返されるロー、パラメータ、ステータス結果を処理する方法を示す。 |

| サンプル・プログラム | 説明 |
|---|---|
| *secct.c* | Client-Library アプリケーションでネットワーク・ベースのセキュリティ機能を使用する方法を示す。プログラムを使用するには、DCE または CyberSafe Kerberos をインストールして実行し、ネットワーク・ベースのセキュリティをサポートするサーバに接続する必要がある。 |
| *secct_dec*<br>*secct_krb* | DCE または CyberSafe Kerberos で、ネットワークベースのセキュリティを使用する方法を示す。 |
| *thrdfunc.c* | *multithrd.c* を使用して、Client-Library でマルチスレッド・クライアント・アプリケーションをコード化する方法を示す。 |
| *thrdutil.c* | マルチスレッドのサンプル・プログラムで使用されるユーティリティ・ルーチンが含まれている。アプリケーションが、より高いレベルのプログラムから Client-Library の実装の詳細部分を隠す方法を示す。 |
| *twophase.c* | 2 つの異なるサーバに対して簡単な更新を実行する、2 フェーズによる commit のサンプル・プログラム。このサンプル・プログラムを実行した後で各サーバに対して isql を使用すると、更新が実際に行われたかどうかを調べることができる。 |
| *uni_blktxt.c* | バルク・コピー・ルーチンを使用して、静的データをサーバ・テーブルにコピーする。 |
| *uni_compute.c* | 計算結果の処理方法を示す。*compute.c* の修正版。 |
| *uni_csr_disp.c* | 読み込み専用カーソルの使い方を示す。*uni_csr_disp.c* サンプル・プログラムの修正版。実行するには unipubs データベースが必要。 |
| *uni_firstapp.c* | unichar データ型と univarchar データ型を使用するように *firstapp.c* を修正したもの。サーバに接続する初歩的なサンプル・プログラムであり、select クエリを送信し、ローを出力する。 |
| *uni_rpc.c* | サーバに RPC コマンドを送信し、結果を処理する。unichar データ型と univarchar データ型を使用するように *rpc.c* サンプル・プログラムを修正したもの。実行するには unipubs データベースが必要。 |
| *usedir.c* | 接続先として使用できるサーバを検索する例を示す。 |
| *wide_compute.c* | ワイド・テーブルと大きなカラム・サイズを使用して計算結果を処理する方法を示す。 |
| *wide_curupd.c*,<br>*wide_ dynamic.c* | カーソルを使用して pubs2 データベース内の publishers テーブルからデータを取り出す。 |
| *wide_rpc.c* | サーバに RPC コマンドを送信し、結果を処理する。*wide_rpc.c* プログラムと同じだが、ワイド・テーブルと大きなカラム・サイズを使用する点が異なる。 |

| サンプル・プログラム | 説明 |
|---|---|
| *wide_util.c* | *wide_** サンプル・プログラムで使用される一般的なルーチンが含まれている。init_db、cleanup_db、connect_db、handle_returns、fetch_n_print が含まれる。 |

サンプル・プログラムを構築して実行する前に、サーバとクライアント・アプリケーションの環境が正しく設定されていることを確認してください。さらに、サンプル・プログラムがサーバへの接続に使用するユーザ名を変更する必要がある場合もあります。手順については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

## サンプル・プログラム内の Client-Library ルーチン

次の表に、Client-Library ルーチンと CS-Library ルーチン、およびその使い方を示すサンプル・プログラムをリストします。

| ルーチン | サンプル・プログラム |
|---|---|
| blk_alloc | *blktxt.c*、*uni_blktxt.c* |
| blk_bind | *blktxt.c*、*uni_blktxt.c*、*wide_compute.c* |
| blk_done | *blktxt.c*、*uni_blktxt.c* |
| blk_drop | *blktxt.c*、*uni_blktxt.c* |
| blk_init | *blktxt.c*、*uni_blktxt.c*、*uni_compute.c* |
| blk_props | *blktxt.c* |
| blk_rowxfer | *blktxt.c*、*uni_blktxt.c* |
| blk_textxfer | *blktxt.c*、*uni_blktxt.c* |
| cs_config | *i18n.c*、*firstapp.c*、*thrdutil.c*、*uni_compute.c* |
| cs_convert | *exutils.c*、*i18n.c*、*rpc.c*、*thrdutil.c*、*uni_rpc.c*、*wide_rpc.c* |
| cs_ctx_alloc | *ex_amain.c*、*exutils.c*、*firstapp.c*、*thrdutil.c*、*csr_disp_scrollcurs.c*、*csr_disp_scrollcurs2.c*、*uni_compute.c*、*uni_csr_disp.c*、*wide_compute.c* |
| cs_ctx_drop | *ex_amain.c*、*exutils.c*、*firstapp.c*、*secct.c*、*thrdutil.c*、*csr_disp_scrollcurs.c*、*csr_disp_scrollcurs2.c* |
| cs_loc_alloc | *i18n.c* |
| cs_loc_drop | *i18n.c* |
| cs_locale | *i18n.c* |
| cs_set_convert | *i18n.c* |
| cs_setnull | *i18n.c*、*rpc.c* |

| ルーチン | サンプル・プログラム |
| --- | --- |
| cs_will_convert | *exutils.c*、*thrdutil.c* |
| ct_bind | *compute.c*、*ex_alib.c*、*exutils.c*、*firstapp.c*、*getsend.c*、*i18n.c*、*thrdutil.c*、*csr_disp_scrollcurs.c*、*csr_disp_scrollcurs2.c*、*uni_compute.c*、*uni_csr_disp.c*、*uni_rpc.c*、*wide_rpc.c* |
| ct_callback | *ex_alib.c*、*ex_amain.c*、*exutils.c*、*firstapp.c*、*thrdutil.c*、*usedir.c*、*csr_disp_scrollcurs.c*、*csr_disp_scrollcurs2.c*、*uni_compute.c*、*uni_csr_disp.c*、*wide_compute.c* |
| ct_cancel | *ex_alib.c*、*ex_amain.c*、*exutils.c*、*getsend.c*、*thrdutil.c* |
| ct_close | *ex_amain.c*、*exutils.c*、*firstapp.c*、*secct.c*、*thrdutil.c*、*csr_disp_scrollcurs.c*、*csr_disp_scrollcurs2.c*、*uni_compute.c*、*uni_csr_disp.c*、wide_compute.c |
| ct_cmd_alloc | *compute.c*、*csr_disp.c*、*ex_alib.c*、*exutils.c*、*firstapp.c*、*getsend.c*、*i18n.c*、*multthrd.c*、*rpc.c*、*csr_disp_scrollcurs.c*、*csr_disp_scrollcurs2.c*、*uni_compute.c*、*uni_csr_disp.c*、*wide_compute.c* |
| ct_cmd_drop | *compute.c*、*csr_disp.c*、*ex_alib.c*、*exutils.c*、*firstapp.c*、*i18n.c*、*multthrd.c*、*thrdutil.c*、*csr_disp_scrollcurs.c*、*csr_disp_scrollcurs2.c*、*uni_compute.c*、*uni_csr_disp.c*、*wide_compute.c* |
| ct_cmd_props | *ex_alib.c*、*rpc.c*、*thrdutil.c*、、*csr_disp_scrollcurs.c*、*csr_disp_scrollcurs2.c* |
| ct_command | *compute.c*、*ex_alib.c*、*exutils.c*、*firstapp.c*、*getsend.c*、*i18n.c*、*multthrd.c*、*rpc.c*、*thrdutil.c*、*arraybind.c*、*uni_rpc.c*、*wide_rpc.c* |
| ct_compute_info | *compute.c*、*uni_compute.c*、*wide_compute.c* |
| ct_con_alloc | *blktxt.c*、*ex_amain.c*、*exconfig.c*、*exutils.c*、*firstapp.c*、*secct.c*、*thrdutil.c*、*usedir.c*、*csr_disp_scrollcurs.c*、*csr_disp_scrollcurs2.c*、*uni_compute.c*、*uni_csr_disp.c*、*wide_compute.c* |
| ct_con_drop | *blktxt.c*、*ex_amain.c*、*exutils.c*、*firstapp.c*、*secct.c*、*thrdutil.c*、*usedir.c*、*csr_disp_scrollcurs.c*、*csr_disp_scrollcurs2.c*、*uni_compute.c*、*uni_csr_disp.c*、wide_compute.c |
| ct_con_props | *blktxt.c*、*ex_alib.c*、*ex_amain.c*、*exconfig.c*、*exutils.c*、*firstapp.c*、*rpc.c*、*secct.c*、*thrdutil.c*、*usedir.c*、*uni_compute.c*、*uni_csr_disp.c*、*wide_compute.c* |
| ct_config | *exutils.c*、*thrdutil.c*、*csr_disp_scrollcurs.c*、*csr_disp_scrollcurs2.c*、*uni_csr_disp.c*、*wide_compute.c* |

| ルーチン | サンプル・プログラム |
|---|---|
| ct_connect | *blktxt.c*、*ex_amain.c*、*exconfig.c*、*exutils.c*、*firstapp.c*、*secct.c*、*thrdutil.c*、*csr_disp_scrollcurs.c*、*csr_disp_scrollcurs2.c*、*uni_compute.c*、*uni_csr_disp.c*、*wide_compute.c* |
| ct_ctx_drop | *uni_compute.c*、*uni_csr_disp.c*、*wide_compute.c* |
| ct_cursor | *csr_disp.c*、*multithrd.c*、*csr_disp_scrollcurs.c*、*csr_disp_scrollcurs2.c*、*uni_csr_disp.c* |
| ct_data_info | *getsend.c* |
| ct_debug | *ex_alib.c*、*ex_amain.c*、*exutils.c*、*thrdutil.c* |
| ct_describe | *compute.c*、*ex_alib.c*、*exutils.c*、*getsend.c*、*i18n.c*、*thrdutil.c*、*csr_disp_scrollcurs.c*、*csr_disp_scrollcurs2.c*、*uni_compute.c*、*uni_csr_disp.c*、*wide_compute.c* |
| ct_ds_dropobj | *usedir.c* |
| ct_ds_lookup | *usedir.c* |
| ct_ds_objinfo | *usedir.c* |
| ct_exit | *ex_amain.c*、*exutils.c*、*firstapp.c*、*secct.c*、*thrdutil.c*、*csr_disp_scrollcurs.c*、*csr_disp_scrollcurs2.c*、*uni_compute.c*、*uni_csr_disp.c*、wide_compute.c |
| ct_fetch | *compute.c*、*ex_alib.c*、*exutils.c*、*firstapp.c*、*getsend.c*、*i18n.c*、*thrdutil.c*、*arraybind.c*、*uni_compute.c*、*uni_csr_disp.c*、*wide_compute.c* |
| ct_get_data | *getsend.c* |
| ct_init | *ex_amain.c*、*exutils.c*、*firstapp.c*、*thrdutil.c*、*csr_disp_scrollcurs.c*、*csr_disp_scrollcurs2.c*、*uni_csr_disp.c*、*wide_compute.c* |
| ct_param | *rpc.c*、*uni_rpc.c*、*wide_rpc.c* |
| ct_poll | *ex_amain.c* |
| ct_res_info | *compute.c*、*ex_alib.c*、*exutils.c*、*i18n.c*、*rpc.c*、*thrdutil.c*、*csr_disp_scrollcurs.c*、*csr_disp_scrollcurs2.c*、*uni_compute.c*、*uni_csr_disp.c*、*uni_rpc.c*、*wide_compute.c*、*wide_rpc.c* |
| ct_results | *compute.c*、*csr_disp.c*、*ex_alib.c*、*exutils.c*、*getsend.c*、*i18n.c*、*multthrd.c*、*rpc.c*、*thrdutil.c*、*csr_disp_scrollcurs.c*、*csr_disp_scrollcurs2.c*、*arraybind.c*、*uni_compute.c*、*uni_csr_disp.c*、*wide_compute.c* |
| ct_send | *compute.c*、*csr_disp.c*、*ex_alib.c, exutils.c*、*firstapp.c*、*getsend.c*、*i18n.c*、*multthrd.c*、*rpc.c*、*thrdutil.c*、*csr_disp_scrollcurs.c*、*csr_disp_scrollcurs2.c*、*uni_compute.c*、*uni_csr_disp.c*、*wide_compute.c* |
| ct_scroll_fetch | *csr_disp_scrollcurs.c*、*csr_disp_scrollcurs2.c*、*ex_utils2.c* |

| ルーチン | サンプル・プログラム |
|---|---|
| ct_send_data | *getsend.c* |
| ct_wakeup | *ex_alib.c* |

# ヘッダ・ファイル

*ctpublic.h* というヘッダ・ファイルは、Client-Library への呼び出しをする、すべてのアプリケーション・ソース・ファイルに必要です。

*ctpublic.h* には、次のものが含まれています。

- Client-Library ルーチンで使用される記号定数の定義
- Client-Library ルーチンの宣言
- CS-Library ヘッダ・ファイル *cspublic.h*

*cspublic.h* には、次のものが含まれています。

- クライアント／サーバ共通の記号定数の定義
- クライアント／サーバ共通の構造体のタイプ定義
- CS-Library ルーチンの宣言
- Client-Library データ型の型宣言を含む *cstypes.h*
- SQLCA 構造体のタイプ宣言を含む *sqlca.h*
- プラットフォーム依存のデータ型と定義が入っている *csconfig.h*

# 高可用性フェールオーバ

高可用性クラスタは 2 台のマシンで構成され、一方のマシン ( またはアプリケーション ) がダウンした場合、もう一方のマシンが 2 台分の負荷を受け持ちます。2 台のマシンはそれぞれ、高可用性クラスタの「ノード」と呼ばれます。高可用性クラスタは通常、常時稼働していなければならない環境で使用します。

Sybase のフェールオーバ機能については、Adaptive Server Enterprise の『高可用性システムにおける Sybase フェールオーバの使用』マニュアルを参照してください。この項では、フェールオーバ中に Open Client アプリケーションがセカンダリ・コンパニオンに接続できるようにするための設定に必要な事項を説明します。

## *interfaces* ファイルへの *hafailover* 行の追加

プライマリ・コンパニオンがクラッシュしたり、shutdown または shutdown with nowait を発行してフェールオーバが発生した場合は、フェールオーバ・プロパティのあるクライアントは、セカンダリ・コンパニオンに自動的に再接続します。クライアントにフェールオーバ・プロパティを付与するには、*interfaces* ファイルに「hafailover」という行を追加し、クライアントがセカンダリ・コンパニオンに接続するのに必要な情報を提供してください。この行を追加するには、ファイル・エディタか dsedit ユーティリティを使用します。

**UNIX プラットフォームの場合**

以下の UNIX *interfaces* ファイル・エントリは、プライマリ・コンパニオン PERSONNEL1 とセカンダリ・コンパニオン MONEY1 を非対称型に設定するためのものです。これには hafailover エントリが含まれていて、PERSONNEL1 に接続しているクライアントがフェールオーバ時に MONEY1 に再接続できるようになっています。

```
MONEY1
   master tcp ether FN1 9835
   query tcp ether FN1 9835
   hafailover PERSONNEL1
PERSONNEL1
   master tcp ether HUM1 7856
   query tcp ether HUM1 7856
   hafailover MONEY1
```

**Windows**

次は、MONEY1 コンパニオンと PERSONNEL1 コンパニオンとの間で対称設定を行う場合の Windows *sql.ini* エントリです。

```
[MONEY1]
 query=TCP,FN1,9835
 master=TCP,FN1,9835
 hafailover=PERSONNEL1
[PERSONNEL1]
 query=TCP,HUM1,7586
 master=TCP,HUM1,7586
 hafailover=MONEY1
```

詳細については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

---

**注意**　クライアント・アプリケーションは、フェールオーバによって送信できなかったクエリを再送する必要があります。また、カーソル宣言などの接続固有のその他の情報は、リストアする必要があります。

---

## Client-Library アプリケーションの変更

---

**注意**　クラスタにインストールされているアプリケーションは、プライマリ・コンパニオンとセカンダリ・コンパニオンの両方で実行可能でなければなりません。並列設定が必要なアプリケーションをインストールする場合は、フェールオーバの間にセカンダリ・コンパニオンがアプリケーションを実行できるように、セカンダリ・コンパニオンにも並列処理の設定を行う必要があります。

---

Client-Library 呼び出しで記述されたアプリケーションを、Sybase フェールオーバ・ソフトウェアで実行できるようにするには、変更が必要です。変更するには、次の手順に従います。

1. ct_config と ct_con_props の各 Client-Library API 呼び出しを使用して、CS_HAFAILOVER プロパティを設定します。プロパティの有効値は CS_TRUE と CS_FALSE です。デフォルト値は CS_FALSE です。次のコードを使用して、コンテキストまたは接続レベルのどちらかにこのプロパティを設定できます。

   ```
   CS_INT TRUE = CS_TRUE;
   CS_INT FALSE = CS_FALSE;
   retcode = ct_config(context, CS_SET, CS_HAFAILOVER,
   &true, CS_UNUSED, NULL);
   retcode = ct_con_props(connection, CS_SET,
   CS_HAFAILOVER, &false, CS_UNUSED, NULL);
   ```

2. フェールオーバ・メッセージを処理します。コンパニオンがシャットダウン処理を始めると、クライアントはフェールオーバが発生するという情報メッセージを受け取ります。これは、クライアント・エラー・ハンドラの情報メッセージとして扱って下さい。

3 フェールオーバ設定を確認します。フェールオーバ・プロパティを設定し、*interfaces* ファイルにセカンダリ・コンパニオン・サーバの有効なエントリが設定されていると、接続はフェールオーバ接続になり、クライアントは適切に再接続します。

   ただし、フェールオーバが設定されていても、*interfaces* ファイルに hafailover サーバのエントリが無い場合 ( またはその逆 ) は、フェールオーバ接続にはなりません。この場合は、フェールオーバ・プロパティがオフになった、高可用性ではない通常の接続になります。フェールオーバ・プロパティを確認して、接続がフェールオーバ接続かどうかを確認してください。これを行うには、CS_GET の *action* とともに ct_con_props を呼び出します。

4 リターン・コードを検証します。フェールオーバが成功したら、ct_results と ct_send を呼び出して、CS_RET_HAFAILOVER を返します。

   同期接続では、API 呼び出しは CS_RET_HAFAILOVER を直接返します。非同期接続では、API は CS_PENDING を返し、コールバック機能は CS_RET_HAFAILOVER を返します。リターン・コードによっては、next コマンドを送信して実行するなど、アプリケーションは必要なプロセスを行います。

5 オプション値をリストアします。クライアントがプライマリ・コンパニオンから切断されると、このクライアント接続に合わせて設定してある set オプション ( たとえば、set role など ) は失われます。フェールオーバした接続で、これらのオプションをリセットします。

6 アプリケーションを再構築し、フェールオーバ・ソフトウェアに含まれるライブラリにリンクさせます。

---

**注意** sp_companion resume を発行しないと、フェールオーバ・プロパティ (isql -Q など ) が設定されたクライアントを接続できません。sp_companion prepare_failback を発行してからクライアントを再接続しようとすると、クライアントは sp_companion resume を発行するまでハングします。

---

## Sybase フェールオーバでの *isql* の使い方

isql を使用してフェールオーバ機能のあるプライマリ・サーバに接続するには、次の手順に従います。

- *interfaces* エントリで指定されているセカンダリ・コンパニオン・サーバのあるプライマリ・サーバを選択します。
- -Q コマンド・ライン・オプションを使用します。

*interfaces_file_name* ファイルに、「interfaces ファイルへの hafailover 行の追加」で指定したエントリの例がある場合は、`isql -S PERSONNEL1 -Q` を入力して、フェールオーバで isql を使用できます。

# interfaces ファイル

「**interfaces ファイル**」には、Adaptive Server Enterprises と Open Server アプリケーションの接続情報が保管されています。クライアントが接続するすべてのサーバについて、*interfaces* ファイルはサーバ名とそのサーバに接続するために必要な情報を含んでいるエントリを保持します。

*interfaces* ファイルは Client-Library のデフォルトのディレクトリです。ただし、アプリケーションは Sybase ディレクトリ・ドライバを使用するようにして、Client-Library がネットワーク・ベースのディレクトリ・サービス・プロバイダを使用するように設定できます。Sybase ディレクトリ・ドライバの設定方法については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。ネットワークベースのディレクトリ・サービスについては、「ディレクトリ・サービス」(115 ページ ) を参照してください。

*interfaces* ファイルを使用する Open Server または Open Client のアプリケーションの場合、送信接続を作成すると、*interfaces* ファイル・エントリをすべて記載したリンクされたリストがメモリにロードされます。それ以降の送信接続では、リンクされたこのリストが参照されます。アプリケーションが複数の *interfaces* ファイルを使用する場合には、これらのファイルすべてに対応するリンクされたリストがメモリにロードされます。アプリケーションの実行中に *interfaces* ファイルが更新される場合、新しい接続が作成されると、更新された *interfaces* フィアルに対応するリンクされたリストがメモリにロードされます。古い、更新されない *interfaces* ファイルに対応するリンクされたリストは、古い *interfaces* ファイルに基づいて作成された接続がすべて閉じられるまでは、メモリから解放されません。その結果、1 つの所定の *interfaces* ファイルに対して複数のリンクされたリストがメモリ内に同時に存在する可能性があります。

ほとんどのプラットフォームでは、*interfaces* ファイルはテキスト形式のオペレーティング・システム・ファイルです。これらのシステムでは、*interfaces* ファイルのデフォルト名、デフォルト・ロケーション、内部フォーマットはプラットフォームごとに異なります。この他のプラットフォームでは、別の形式の記憶領域を使用します。表 2-24 は、一般的なプラットフォームの *interfaces* ファイルを示します。

**表 2-24：各プラットフォームの interfaces ファイルの名前**

| プラットフォームまたはプラットフォーム・ファミリ | ファイル名 |
|---|---|
| UNIX ( すべて ) | *interfaces*<br>パスは SYBASE 環境変数を設定して指定する。 |
| Windows | *sql.ini*<br>パスは SYBASE 環境変数を設定して指定する。 |

アプリケーションは CS_IFILE コンテキスト・プロパティを設定して、デフォルト以外のファイル名とロケーションを指定できます (「interfaces ファイルのロケーション」(251 ページ ) を参照してください )。*interfaces* ファイルの代替のデフォルトのファイル名とパスは、CS_DEFAULT_IFILE プロパティで指定できます。CS_DEFAULT_IFILE プロパティの詳細については、『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。

## interfaces ファイル・エントリの概要

*interfaces* ファイルのフォーマットはプラットフォームごとに異なります。*interfaces* ファイルを編集する場合は、使用しているプラットフォーム用の『Open Client/Server 設定ガイド』を参照してください。設定ガイドでは、*interfaces* ファイルについての詳細と、各プラットフォーム上で ct_connect と ct_ds_lookup によって *interfaces* ファイルがどのように使用されるかを詳しく説明しています。

次にプラットフォームに依存しない *interfaces* ファイル・エントリの概要と Client-Library での使用方法について説明します。

表 2-25 に、*interfaces* ファイル・エントリの共通コンポーネントを示します。

**表 2-25：interfaces ファイルのエントリのコンポーネント**

| コンポーネント | 説明 |
|---|---|
| Transport Address Values | プラットフォーム固有のフォーマットで、サーバ名に対応する 1 つまたは複数のアドレス。 |

| コンポーネント | 説明 |
|---|---|
| Retry Count Value | UNIX プラットフォームでは、CS_RETRY_COUNT 接続プロパティの設定の代わりにこのコンポーネントを提供している。<br><br>**注意**　代わりに CS_RETRY_COUNT を使用することをおすすめします。 |
| Loop Delay Value | UNIX プラットフォームでは、CS_LOOP_DELAY 接続プロパティの設定の代わりにこのコンポーネントを提供している。<br><br>**注意**　代わりに CS_LOOP_DELAY を使用することをおすすめします。 |
| Security Mechanisms | 「オブジェクト識別子」の文字列リスト。それぞれの文字列はサーバでサポートされるセキュリティ・メカニズムのグローバル名を表す。 |

## interfaces ファイルのサーバ・オブジェクト

次のいずれかが発生すると、ct_ds_lookup は *interfaces* ファイル内のサーバ・ディレクトリ・オブジェクトを探します。

- アプリケーションが *interfaces* ファイルを CS_CONNECTION 構造体のディレクトリ・ソースとして使用することを選択した場合、またはデフォルトとして使用する場合。接続のディレクトリ・ソースは CS_DS_PROVIDER 接続プロパティを指定して使用します (「ディレクトリ・サービス・プロバイダ」(132 ページ ) を参照してください )。
- Client-Library がドライバの設定で指定されたディレクトリ・ドライバをロードできないで、*interfaces* ファイルへのフェールオーバが発生した場合。ディレクトリ・サービスのフェールオーバは、CS_DS_FAILOVER 接続プロパティを使用して有効にされている場合にだけ発生します。「ディレクトリ・サービスのフェールオーバ」(130 ページ ) を参照してください。

これらの場合には、Client-Library はそれぞれの *interfaces* ファイル・エントリの内容を、ct_ds_objinfo を使用して表示できるサーバ・ディレクトリ・オブジェクトのインスタンスにマップします。表 2-26 は、このマッピングについての説明です。

**表 2-26：interfaces ファイル・エントリに対するサーバ・ディレクトリ・オブジェクト属性のマッピング**

| 属性 | 対応する interfaces ファイルのコンポーネント |
|---|---|
| Server Entry Version | なし。この値は、*interfaces* ファイルを検索する場合は常に 1。 |
| Server Name Attribute | *interfaces* ファイル内のサーバの名前。*interfaces* ファイルを検索する場合は、名前属性の値とディレクトリ・エントリ名は同一。 |
| Service Type | なし。この値は、*interfaces* ファイルを検索する場合は常に「Adaptive Server Enterprise」。 |
| Server Status | なし。ステータスは、*interfaces* ファイルを検索する場合は常に CS_STATUS_UNKNOWN。 |
| Transport Address | *interfaces* ファイル・エントリ内の各「query」行の内容。CS_TRANADDR 構造体でアプリケーションに返される。<br>*interfaces* ファイル内に複数の「query」行が存在する場合は、この属性の値を含む CS_ATTRVALUE 配列の順序は *interfaces* ファイルと同じになる。<br>「master」行は無視される。接続を確立する場合、クライアントは「query」行だけを使用するので、ct_ds_lookup が *interfaces* ファイルを読み込むときに「master」行は無視される。<br>トランスポート・アドレスのフォーマットについては、「トランスポート・アドレス値」(530 ページ ) を参照。 |
| Security Mechanisms | エントリの「secmech」行にリストされる OID。各 OID は、CS_OID 構造体内に存在する。 |
| Retry Count | 一部のプラットフォームでは、「retry_count」オプションを *interfaces* ファイル・エントリ内に指定できる。このオプションは、Client-Library が各サーバ・アドレスへの接続を試行する回数を制御する。アプリケーションでは、接続の CS_RETRY_COUNT プロパティを設定すると、同等の動作が実行される。「リトライ回数」(264 ページ ) を参照。<br>エントリ内に存在する場合は、この値は OID 文字列 CS_OID_ATTRRETRYCOUNT と整数の構文を持つ属性として返される。 |
| Loop Delay | 一部のプラットフォームでは、「loop_delay」オプションを *interfaces* ファイル内に指定できる。アプリケーションでは、CS_LOOP_DELAY 接続プロパティを設定すると、同等の動作が実行される。「ループ遅延」(253 ページ ) を参照。<br>エントリ内に存在する場合は、この値は OID 文字列 CS_OID_ATTRLOOPDELAY と整数の構文を持つ属性として返される。 |

# 国際化のサポート

Client-Library は「**ローカライゼーション**」により、国際化されたアプリケーションをサポートします。一般に、ローカライズされたアプリケーションは、次のようになります。

- Client-Library および Adaptive Server Enterprise メッセージにネイティブ言語を使用します。
- 各国別の日時フォーマットを使用します。
- 文字列を変換または比較をする場合、指定した文字セットおよび順番 ( **照合順** ) を使用します。

Client-Library は、ほとんどのプラットフォームで、環境変数を使用して、アプリケーションが使用するデフォルト・ローカライゼーション値を決定します。これらのデフォルト値が、アプリケーションのニーズを満たしている場合は、それ以上アプリケーションをローカライズする必要はありません。

デフォルト値が、アプリケーションのニーズを満たしていない場合、アプリケーションでは、CS_LOCALE 構造体を使用して、コンテキスト、接続、またはデータ要素レベルのカスタム・ローカライゼーション値を設定できます。CS_LOCALE 構造体の使用については、「CS_LOCALE 構造体の使用」(162 ページ ) を参照してください。

国際化対応の Open Client および Open Server アプリケーションを開発するためのガイドラインについては、『Open Client/Server 開発者用国際化ガイド』を参照してください。このトピック・ページでは、Client-Library アプリケーションの開発に特有の情報の概要を示します。

## アプリケーションが CS_LOCALE 構造体を使用する必要がある場合

**警告！**デフォルト・ロケールを決定するための各プラットフォームのメカニズムについては、使用しているプラットフォームの『Open Client/Server 設定ガイド』のローカライゼーションの章を参照してください。Client-Library のローカライゼーション・メカニズムはプラットフォームごとに異なるため、この章を読み、使用しているプラットフォームでのデフォルト・ロケールの決定方法を理解してください。

一般に、アプリケーションのデフォルト・ロケールは、ローカル環境の言語または文字セットを反映しています。デフォルト・ロケールは、CS-Library のコンテキスト・プロパティ CS_LOC_PROP の値によって指定されます。一般のアプリケーションの場合、コンテキスト構造体のロケールと異なる言語または文字セットで作業する場合にのみ CS_LOCALE 構造体を使用します。

次に例を示します。

- ドイツ語のアプリケーションは、Client-Library のエラー・メッセージをフランス語で受信するために、CS_LOCALE 構造体を接続構造体と関連付ける必要がある場合があります。
- 独自の文字セット変換を実行するアプリケーションは、cs_convert ルーチンを呼び出して、使用するための CS_LOCALE 構造体を初期化する必要があります。

## CS_LOCALE 構造体の使用

CS_LOCALE 構造体は、ローカライゼーション値を定義します。アプリケーションは、CS_LOCALE 構造体を使用して、コンテキスト・レベル、接続レベル、およびデータ要素レベルでカスタム・ローカライゼーション値を定義します。

アプリケーションは、次のようにしてカスタム・ローカライゼーション値を定義します。

1. cs_loc_alloc を呼び出して CS_LOCALE 構造体を割り付けます。
2. cs_locale を呼び出し、カスタム・ローカライゼーション値で CS_LOCALE をロードします。パラメータの種類により、cs_locale は LC_ALL、LC_CTYPE、LC_COLLATE、LC_MESSAGE、LC_TIME、または LANG 環境変数を検索します。
3. CS_LOCALE を使用します。アプリケーションは次のことを行います。
   - *property* を CS_LOC_PROP に設定して cs_config を呼び出し、カスタム・ローカライゼーション値をコンテキスト構造体へコピーします。
   - *property* を CS_LOC_PROP に設定して ct_con_props を呼び出し、カスタム・ローカライゼーション値を接続構造体へコピーできます。CS_LOC_PROP はログイン・プロパティであるため、接続がオープンした後では、アプリケーションがその値を変更できないことに注意してください。

- カスタム・ローカライゼーション値を受け取るルーチンへのパラメータとして CS_LOCALE を指定できます (cs_strcmp、cs_time)。
- 変換先のプログラム変数を記述する CS_DATAFMT 構造体に CS_LOCALE を含めることができます (cs_convert、ct_bind)。

4　cs_loc_drop を呼び出して、CS_LOCALE. の割り付けを解除します。

## コンテキスト・レベルのローカライゼーション

コンテキスト・レベルのローカライゼーション値は、Open Client コンテキストのローカライゼーションを定義します。

アプリケーションが CS_CONTEXT 構造体を割り付けると、CS-Library は新しいコンテキストにデフォルト・ローカライゼーション値を割り付けます。ほとんどのプラットフォームでは、環境変数がデフォルト値を決定します。使用しているプラットフォームのデフォルト・ローカライゼーション値の割り付け方法については、『Open Client/Server 設定ガイド』を参照してください。

デフォルト・ローカライゼーション値は、常に定義されているので、アプリケーションは、デフォルト値が受け入れ不可能な値である場合にだけ、コンテキスト・レベルの新しいローカライゼーション値を定義する必要があります。

## 接続レベルのローカライゼーション

接続レベルのローカライゼーション値は、特定のクライアント・サーバ接続のローカライゼーションを定義します。

新しい接続は、その親コンテキストからデフォルト・ローカライゼーション値を継承します。このため、アプリケーションは、親コンテキストの値が受け入れ不可能である場合にかぎり、接続のローカライゼーション値を新たに定義する必要があります。

アプリケーションが、ct_connect を呼び出して接続をオープンすると、サーバは、接続の言語および文字セットをサポートできるかどうかを調べます。サポートすることができない場合、その接続要求は失敗します。

---

**注意**　この機能は DB-Library の機能とは異なります。DB-Library では、アプリケーションが DBSETLNATLANG を呼び出してネイティブ言語名を設定しなければ、接続は Adaptive Server のデフォルトのネイティブ言語を使用します。

---

## データ要素レベルのローカライゼーション

データ要素レベルでは、CS_LOCALE は特定のデータ要素、たとえば、バインド変数のローカライゼーション値を定義します。

既存の接続のローカライゼーション値が受け入れ不可能の場合にのみ、アプリケーションはデータ要素レベルでローカライゼーション値を定義する必要があります。

たとえば、接続がアメリカ英語ロケール (U.S. English 言語、iso_1 文字セットと適切な日時フォーマット ) を使用しているとしても、接続はフランス語の月日名を使用して、*datetime* 結果カラムを表示しなければならない場合があります。

アプリケーションは次のことを行います。

- cs_loc_alloc を呼び出して CS_LOCALE 構造体を割り付けます。
- cs_locale を呼び出して、CS_LOCALE 構造体をフランス語の日時フォーマットを使用してロードします。
- cs_dt_info を呼び出して、CS_LOCALE 構造体の日時変換フォーマットをカスタマイズします。
- ct_bind を呼び出して、結果カラムを文字変数にバインドします。バインド変数を記述する CS_DATAFMT 構造体は、フランス語の CS_LOCALE を参照しなければなりません。

アプリケーションが ct_fetch を呼び出すと、結果カラムの日時の値は、フランス語の日付と月を含む文字列に自動的に変換され、バインドされた変数に自動的にコピーされます。

# ローカライゼーション情報の検索

使用するローカライゼーション値を決定するには、Client-Library はデータ要素レベルから検索を始めます。検索の順番は次のとおりです。

1. データ要素ローカライゼーション値
    - データ要素を記述している CS_DATAFMT 構造体に対応している CS_LOCALE
    - パラメータとしてルーチンに渡される CS_LOCALE
2. 接続構造体ローカライゼーション値
3. コンテキスト構造体のローカライゼーション値

アプリケーションがコンテキスト構造体を割り付けると、CS-Library は、デフォルトのローカライゼーション値を持つ新しいコンテキストを提供するため、コンテキスト・レベルのローカライゼーション値は、常に定義されています。

コンテキストを割り付けると、アプリケーションは、cs_loc_alloc、cs_locale、および cs_config を呼び出すことによって、そのローカライゼーション値を変更することができます。

## ロケール・ファイル

Sybase のロケール・ファイルは、言語、文字セット名、およびソート順をロケール名に関連付けます。Open Client/Server 製品は、ローカライゼーション情報をロードするときにこのロケール・ファイルを使用します。

ロケール・ファイルは、Open Client/Server 製品に、言語、文字セット名、およびソート順名を指示します。しかし、ロケール・ファイルには、実際のローカライズされたメッセージおよび文字セット情報は含まれていません。

『Open Client/Server 設定ガイド』を参照してください。

### ロケール・ファイル・エントリ

ロケール・ファイルは、プラットフォーム固有のセクションを持っています。各セクションには、次のような形式のエントリがあります。

```
locale = locale_name, language, charset, sortorder
```

*sortorder* は、オプション・フィールドです。*sortorder* が指定されていない場合、指定したロケールのソート順はデフォルトのバイナリになります。

各エントリは、言語、文字セット、およびソート順を関連付けることによりロケール名を定義します。

たとえば、ロケール・ファイルのセクションには、次のようなエントリがあります。

```
locale = default, us_english, iso_1, dictionary
locale = fr, french, iso_1, noaccents
locale = japanese.sjis, japanese, sjis
```

これらのエントリは、次のことを示します。

- 「default」とロケール名が指定された場合、「us_english」という言語、「iso_1」という文字セット、「dictionary」というソート順が使用されます。
- 「fr」とロケール名が指定された場合、「french」という言語、「iso_1」という文字セット、「noaccents」というソート順が使用されます。
- 「japanese.sjis」とロケール名が指定された場合、「japanese」という言語、「sjis」という文字セット、「binary」( デフォルトのソート順 ) というソート順が使用されます。

---

**注意**　Sybase は、ロケール・ファイルにロケール名のエントリを含んでいるので、いくつかのロケール名はあらかじめ定義されています。これらのエントリが必要としているものと一致しない場合、それらを修正または新しいロケール名を定義するエントリを追加することができます。

---

## *cs_locale* およびロケール・ファイル

CS_LOCALE 構造体を使用して、コンテキスト、接続、またはデータ要素に対するカスタム・ローカライゼーション値を設定する前に、Client-Library アプリケーションは、cs_locale を呼び出して、要求されているローカライゼーション値を持つ CS_LOCALE 構造体をロードしなければなりません。

CS_LOCALE 構造体のロード時に、cs_locale は次のことを実行します。

1. 使用するロケール名を決定します。
   - cs_locale の *buffer* パラメータが指定されている場合は、これがロケール名になります。
   - cs_locale の *buffer* パラメータが NULL の場合、cs_locale は、ロケール名のプラットフォーム固有のオペレーティング・システム検索を実行します。この検索の詳細については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。
2. ロケール・ファイルのロケール名を検索して、関連付けられている言語、文字セット、およびソート順を決定します。
3. *type* パラメータによって指定された情報のタイプを CS_LOCALE へロードします。たとえば、*type* が CS_LC_CTYPE の場合、cs_locale は文字セット情報をロードします。*type* が CS_LC_MESSAGE の場合、cs_locale は、メッセージ情報をロードします。

# ストアド・プロシージャ・パラメータとしてのラージ・オブジェクト

Client アプリケーションは、ストアド・プロシージャの入力パラメータおよび動的 SQL 文のパラメータとして text、unitext、image を使用することをサポートしています。

これらの接続機能により、ラージ・オブジェクト (LOB) データ型をパラメータとして使用するためのログイン・ネゴシエーションが容易になります。

- CS_RPCPARAM_LOB － クライアント・アプリケーションはこの要求機能をサーバに送信して、LOB データ型をパラメータとして使用できるかどうか判断します。サーバはこの機能をサポートできないときに、初期ログイン・ネゴシエーションのこの機能ビットをクリアします。ユーザがこのようなサーバに LOB データ型をパラメータとして送信しようとすると、エラーが発生します。
- CS_RPCPARAM_NOLOB － クライアント・アプリケーションはこの応答機能を送信し、パラメータとして LOB データ型を送信しないようにサーバに要求します。この機能はデフォルトにより有効になっています。

## パラメータとしての少量の LOB データの送信

少量の LOB データをストアド・プロシージャの入力パラメータまたは準備された SQL 文のパラメータとして送信するプロセスは、LOB 以外のパラメータを送信するプロセスと同じです。

少量の LOB データを送信するには、コマンドとデータ用のメモリを割り付け、ct_param() または ct_setparam() を使用してこれらをサーバに直接送信します。

text、unitext、image の各データ型をパラメータとして使用する際には、CS_DATAFMT 構造体の *maxlength* フィールドを設定する必要があります。*maxlength* 値は、すべての LOB データがサーバに一度に送信されるか、ストリーミングされるかを示します。*maxlength* がゼロより大きい場合、LOB データは 1 つのまとまりで送信されます。*maxlength* が CS_UNUSED に設定されている場合、LOB データはデータをまとまりで送信するための ct_send_data() 呼び出しのループを使用して、ストリームで送信されます。ゼロのまとまりの長さは、データ・ストリームの終わりを示します。

**例 1** 少量の LOB データをストアド・プロシージャの入力パラメータとして送信します。

```
CS_TEXT textvar[50];
CS_DATAFMT paramfmt;
CS_INT datalen;
CS_SMALLINT ind;

...
ct_command(cmd, CS_RPC_CMD, ...)

/*
** Clear and setup the CS_DATAFMT structure, then pass
** each of the parameters for the RPC.
*/
memset(&paramfmt, 0, sizeof(paramfmt));

/*
** First parameter, an integer.
*/
strcpy(paramfmt.name, "@intparam");
paramfmt.namelen = CS_NULLTERM;
paramfmt.datatype = CS_INT_TYPE;
paramfmt.maxlength = CS_UNUSED;
paramfmt.status = CS_INPUTVALUE;
paramfmt.locale = NULL;
ct_param(cmd, &paramfmt, (CS_VOID *)&intvar,
   sizeof(CS_INT), ind))

/*
** Second parameter, a (small) text parameter.
*/

strcpy((CS_CHAR *)textvar, “The Open Client and Open
   Server products both include Bulk-Library and
   CS-Library.”);
datalen = sizeof(textvar);
strcpy(paramfmt.name, "@textparam");
paramfmt.namelen = CS_NULLTERM;
paramfmt.datatype = CS_TEXT_TYPE;
paramfmt.maxlength = EX_MYMAXTEXTLEN;
paramfmt.status = CS_INPUTVALUE;
paramfmt.locale = NULL;
ct_setparam(cmd, &paramfmt, (CS_VOID *)&textvar,
   &datalen, &ind);

ct_send(cmd);
```

```
ct_results(cmd, &res_type);

...
```

**例 2**　少量の LOB データを準備文を使用して送信します。

```
/*
** Prepare the sql statement.
*/
sprintf(statement, "select title_id from mybooks where
   title like (?) ");

/*
** Send the prepared statement to the server
*/
ct_dynamic(cmd, CS_PREPARE, "my_dyn_stmt", CS_NULLTERM,
   statement, CS_NULLTERM);

ct_send(cmd);
handle_results(cmd);

/*
** Prompt user to provide a value for title
*/
printf("Enter title id value - enter an X if you wish
   to stop:¥n");

while (toupper(title[0]) != 'X')
{
   printf("Retrieve detail record for title:?");
   fgets(mytexttitle, 50,  stdin);

   /*
   ** Execute the dynamic statement.
   */

   ct_dynamic(cmd, CS_EXECUTE, "my_dyn_stmt",
   CS_NULLTERM, NULL, CS_UNUSED);

    /*
   ** Define the input parameter
   */

   memset(&data_format, 0, sizeof(data_format));
   data_format.status = CS_INPUTVALUE;
   data_format.namelen = CS_NULLTERM ;
   data_format.datatype = CS_TEXT_TYPE;
```

```
      data_format.format = CS_FMT_NULLTERM;
      data_format.maxlength = EX_MYMAXTEXTLEN;
      ct_setparam(cmd, &data_format,
         (CS_VOID *)mytexttitle, &datalen, &ind);

   ct_send(cmd);
   handle_results(cmd);
   ...
   }
```

## パラメータとしての大量の LOB データの送信

大量の LOB データは、リソースを効率的に管理するために、ストリームでサーバに送信されます。ct_send_data() をループ内で使用して、まとまったデータをサーバに送信します。

LOB データ・パラメータをまとまりで送信するには :

- CS_DATAFMT 構造体の *datatype* フィールドを CS_TEXT_TYPE、CS_UNITEXT_TYPE、または CS_IMAGE_TYPE に設定します。
- CS_DATAFMT 構造体の *maxlength* フィールドを CS_UNUSED に設定します。
- ct_param() 関数の **data* ポインタ引数を NULL に設定します。
- ct_param() 関数の *datalen* 引数を 0 に設定します。

**例 1** 大量の LOB データ・パラメータをまとまりで送信します。

```
#define BUFSIZE 2048

int fp;
char sendbuf[BUFSIZE]

/*
** Clear and setup the CS_DATAFMT structure, then pass
** each of the parameters for the RPC.
*/
memset(&paramfmt, 0, sizeof(paramfmt));
strcpy(paramfmt.name, "@intparam");
paramfmt.namelen = CS_NULLTERM;
paramfmt.datatype = CS_INT_TYPE;
paramfmt.maxlength = CS_UNUSED;
paramfmt.status = CS_INPUTVALUE;
paramfmt.locale = NULL;
```

```
ct_param(cmd, &paramfmt, (CS_VOID *)&intvar,
   sizeof(CS_INT), 0))

/*
** Text parameter, sent as a BLOB.
*/
strcpy(paramfmt.name, "@textparam");
paramfmt.namelen = CS_NULLTERM;
paramfmt.datatype = CS_TEXT_TYPE;
paramfmt.maxlength = CS_UNUSED;
paramfmt.status = CS_INPUTVALUE;
paramfmt.locale = NULL;

/*
** Although the actual data will not be sent here, we
** must invoke ct_setparam() for this parameter to send
** the parameter format (paramfmt) information to the
** server, prior to sending all parameter data.
** Set *data to NULL and datalen = 0, to indicate that
** the length of text data is unknown and we want to
** send it in chunks to the server with ct_send_data().
*/
ct_setparam(cmd, &paramfmt, NULL, 0, 0);

/*
** Another LOB parameter (image), sent in chunks with
** ct_send_data()
*/
strcpy(paramfmt.name, "@textparam");
paramfmt.namelen = CS_NULLTERM;
paramfmt.datatype = CS_IMAGE_TYPE;
paramfmt.maxlength = CS_UNUSED;
paramfmt.status = CS_INPUTVALUE;
paramfmt.locale = NULL;

/*
** Just like the previous parameter, invoke
** ct_setparam() for this parameter to send the
** parameter format.
*/
ct_setparam(cmd, &paramfmt, NULL, 0, 0);

/*
** Repeat this sequence of filling paramfmt and calling
** ct_param() for any subsequent parameter that needs
** to be sent before finally sending the data chunks for
```

```
** the LOB type parameters.
*/
strcpy(paramfmt.name, "@any_otherparam");
paramfmt.namelen = CS_NULLTERM;
paramfmt.datatype = CS_MONEY_TYPE;
...

/*
** Send the first LOB (text) parameter in chunks of
** 'BUFSIZE' to the server. We must end with a 0 bytes
** write to indicate the end of the current parameter.
*/
fp = open("huge_text_file", O_RDWR, 0666);

do
{
   num_read = read(fp, sendbuf, BUFSIZE);
   ct_send_data(cmd, (CS_VOID *)sendbuf, num_read);
}  while (num_read != 0);


/*
** Repeat the ct_send_data() loop for the next LOB
** parameter.
** Send the image parameter in chunks of 'BUFSIZE'
** to the server as well and end with a 0 bytes write
** to indicate the end of the current parameter.
*/
fp = open("large_image_file", O_RDWR, 0666);
do
{
   num_read = read(fp, sendbuf, BUFSIZE);
   ct_send_data(cmd, (CS_VOID *)sendbuf, num_read);
} while (num_read != 0);

/*
** Ensure that all the data is flushed to the server
*/
ct_send(cmd);
```

**例 2**　準備された SQL 文を使用して、ストリームとして LOB データを送信します。

```
/*
** Prepare the sql statement.
*/
sprintf(statement, "select title_id from mybooks
```

```
    where title like (?) ");

/*
** Send the prepared statement to the server
*/
ct_dynamic(cmd, CS_PREPARE, "mydyn_stmt", CS_NULLTERM,
    statement, CS_NULLTERM);

ct_send(cmd);
handle_results();

/*
** Promt user to provide a value for title
*/
printf("Enter title id value - enter an X if you wish
    to stop:¥n");

while (toupper(myblobtitle[0]) != 'X')
{
    printf("Retrieve detail record for title:?");
    fgets(myblobtitle, 50,  stdin);

    /*
    ** Execute the dynamic statement.
    */
    ct_dynamic(cmd, CS_PREPARE, "my_dyn_stmt",
    CS_NULLTERM, statement, CS_NULLTERM);

    /*
    ** Define the input parameter, a TEXT type that we
    want to send in chunks to the server.
    */
    memset(&data_format, 0, sizeof(data_format)) ;
    data_format.namelen = CS_NULLTERM ;
    data_format.datatype = CS_TEXT_TYPE;
    data_format.maxlength = CS_UNUSED;
    data_format.status = CS_INPUTVALUE;
    ct_setparam(cmd, &data_format, NULL, 0, 0);

    /*
    ** Send the 'myblobtitle' data in chunks of
    ** 'CHUNKSIZE' to the server with ct_send_data() and
    ** end with 0 bytes to indicate the end of data for
    ** this parameter. This is just an example to show
    ** how chunks can be sent.(myblobtitle[] is used as
    ** a simple example. This could also be replaced by
```

```
    ** large file which would be read in chunks from disk
    ** for example).
    */
    bytesleft = strlen(myblobtitle);
    bufp = myblobtitle;

    do
    {
       sendbytes = min(bytesleft, CHUNKSIZE);
       ct_send_data(cmd, (CS_VOID *)bufp, sendbytes);
       bufp += bufp + sendbytes;
       bytesleft -= sendbytes;
    }  while (bytesleft > 0)

    /*
    ** End with 0 bytes to indicate the end of current
    data.
    */
    ct_send_data(cmd, (CS_VOID *)bufp, 0);

    /*
    ** Insure that all the data is sent to the server.

    */
    ct_send(cmd);
    handle_results(cmd)
    ...
 }

 /*
 ** Deallocate the prepared statement and finish up.
 */

 ct_dynamic(cmd, CS_DEALLOC, "my_dyn_stmt", CS_NULLTERM,
    NULL, CS_UNUSED);

 ct_send(cmd);
 handle_results(cmd);
```

# マクロ

マクロは、一般的に 1 つまたは複数の引数を使用し、ソース・ファイルがプリプロセッサで処理されるときにインラインの C コードに展開される C 言語定義です。次の項では、Open Client マクロの機能内容を説明します。

## メッセージ番号の復号化

Client-Library と CS-Library のメッセージ番号は、レイヤ、オリジン、重大度、番号の 4 つのコンポーネントで構成される CS_INT のサイズの整数です。マクロ CS_LAYER、CS_ORIGIN、CS_SEVERITY、CS_NUMBER はメッセージ番号からコンポーネントを取り出します。「Client-Library メッセージの番号」(89 ページ ) を参照してください。

## CS_CAP_TYPE 構造体内のビット操作

クライアント／サーバ接続がサポートする機能を示します。各接続の機能についての情報は CS_CAP_TYPE 構造体に保管されます。

マクロ CS_CLR_CAPMASK、CS_SET_CAPMASK、CS_TST_CAPMASK は CS_CAP_TYPE 構造体のビットを操作します。これらのマクロについては、「複数機能の設定および取得」(81 ページ ) を参照してください。

## *sizeof* 演算子の使い方

C の sizeof 演算子は、指定された項目のサイズをバイト単位で返します。戻り値のデータ型はプラットフォームごとに異なるので、Client-Library ルーチンに CS_INT 引数の代わりに sizeof を指定すると、返される型が CS_INT と同じベース型ではない場合はコンパイル・エラーになるか警告メッセージが表示されます。

Client-Library は、Client-Library ルーチンを呼び出すときに、アプリケーションが sizeof 関数を使用できるように次のマクロを提供します。

CS_SIZEOF(*variable*) − sizeof の戻り値を CS_INT にキャストします。

このマクロはヘッダ・ファイル *cstypes.h* 内に定義されています。

## 関数のプロトタイプ宣言

C コンパイラによっては、その関数が受け取る各引数の個数とデータ型を示す ANSI 形式のプロトタイプを使用して、それぞれの関数を宣言する必要があります。それ以外のコンパイラでは ANSI 形式のプロトタイプを認識しません。

PROTOTYPE マクロを使用すれば、ANSI コンパイラと ANSI 以外のコンパイラの両方で使用できる事前宣言ができます。このマクロは、次のように C 関数の事前宣言に使用します。

```
PROTOTYPE (( argument_list ));
```

*argument_list* には ANSI 形式の引数リストを指定します。PROTOTYPE は条件付きで定義されます。コンパイラが ANSI 形式のプロトタイプをサポートする場合には、PROTOTYPE はコンパイルされるコードに引数リストを反映させます。サポートしない場合には、PROTOTYPE は何も行いません。

次の例は PROTOTYPE の使い方を示します。

```
extern CS_RETCODE CS_PUBLIC ex_clientmsg_cb PROTOTYPE((
         CS_CONTEXT *context,
         CS_CONNECTION *connection,
         CS_CLIENTMSG *errmsg
         ));

CS_RETCODE CS_PUBLIC
 ex_clientmsg_cb(context, connection, errmsg)
 CS_CONTEXT      *context;
 CS_CONNECTION   *connection;
 CS_CLIENTMSG    *errmsg;
 {
    ... function body goes here ...
 }
```

CS_PUBLIC は、マシン固有の宣言に必要な条件を満たすように、コールバック機能のプロトタイプに使用します。「CS_PUBLIC によるコールバック宣言」(33 ページ ) を参照してください。

# マルチスレッド・アプリケーション：シグナル処理

この項では、UNIX プラットフォームにおけるマルチスレッド・アプリケーションのシグナル処理について説明します。ここでは、マルチスレッド・アプリケーションを構築する Sybase のライブラリのリエントラント・バージョンの使い方を説明する Open Client/Server のマニュアルの内容を補足します。

## 基本概念

UNIX オペレーティング・システムでは、シグナルを使ってプロセスの例外的な状況をレポートします。シグナルには、無効なアドレスの参照などの同期イベントをレポートするものがあります。また、電話回線の切断などの非同期イベントをレポートするシグナルもあります。

ハンドラ関数をインストールすると、シグナル発生時に実行されるアクションを指定できます。シグナルが発生すると、オペレーティング・システムがシグナルのハンドラ機能を実行します。

Sybase 提供の呼び出しを使って、シグナル・ハンドラをインストールします。オペレーティング・システム・コールを使ってシグナル・ハンドラをインストールすると、Sybase のライブラリの内部動作に悪影響を及ぼします。

## 非スレッド環境でのシグナル処理

シグナル処理は、バージョン 12.0 以前またはバージョン 12.0 以降の非スレッドの Sybase ライブラリを使用する、従来の非スレッド UNIX 環境で簡単に行うことができます。各プロセスには、制御スレッドが 1 つあります。Open Client/Server ライブラリ・コールを使用して、所定のシグナル用のハンドラを登録します。Client-Library では ct_callback を使用し、Server-Library では srv_signal を使用します。

シグナルが発生すると、Sybase のライブラリがシグナルをトラップし、指定されたシグナル・ハンドラを呼び出します。シグナルをマスキングして、シグナルがプロセスに配信されないようにブロックするには、UNIX システム・コールの sigprocmask を使用します。

## シグナルの種類

シグナルは、それを生成したイベントによって 2 つのカテゴリーに分類されます。

| イベントの種類 | シグナルの種類 |
|---|---|
| 例外 | 同期シグナル |
| 外部イベント | 非同期シグナル |

**例外と同期シグナル**

同期シグナルは、プログラム内の不正な操作が原因で生じる例外やエラーによって生成されます。例外には、無効なメモリ・アドレスへのアクセスやゼロによる除算などがあります。

同期シグナルには SIGILL、SIGFPE、SIGBUS、SIGSEGV、SIGSYS、SIGPIPE などがあります。

**外部イベントと非同期シグナル**

非同期シグナルは、そのシグナルを受け取るプロセスの制御外のイベントによって生成され、その発生は予測できません。非同期シグナルは、実行中の命令とは関係なくプロセスに配信されます。

通常、非同期シグナルには、SIGHUP、SIGINT、SIGQUIT、SIGALRM、SIGTERM、SIGUSR1、SIGUSR2、SIGCHLD、SIGPWR、SIGVTALRM、SIGPROF、SIGIO、SIGWINCH、SIGTSTP、SIGCONT、SIGTTIN、SIGTTOU、SIGURG があります。

Sybase のライブラリでは、非同期シグナル SIGTRAP を同期シグナルとして扱います。「SIGTRAP シグナル」(182 ページ ) を参照してください。

## シグナル・ハンドラ

すべての UNIX プラットフォームでは、シグナル・ハンドラはプロセスごとにインストールされています。マルチスレッド環境では、各シグナルのシグナル・ハンドラは、プロセス内に 1 つしかありません。スレッドに対してインストールされた最新のシグナル・ハンドラは、プロセス内のすべてのスレッドに対して有効です。ハンドラは、シグナルが配信されたときに呼び出されます。

## シグナルのマスキング

シグナルをマスキングすると、ある条件を満たすまでシグナルを配信しないように指定できます。

非スレッド環境では、制御スレッドが 1 つしかありません。各シグナルは、プロセス全体にわたってマスキング／マスキング解除されます。

マルチスレッド環境では、シグナルのマスキング方法はプラットフォームごとに異なります。

- ネイティブ・スレッドをサポートしていないプラットフォームでは、シグナルはプロセスごとにマスキングされます。あるスレッドでシグナルのマスキングを変更すると、プロセス全体に影響します。
- ネイティブ・スレッドをサポートしているプラットフォームでは、シグナルはスレッドごとにマスキングされます。あるスレッドのシグナルをマスキングしても、そのスレッド以外には影響しません。プロセス全体にわたってシグナルをマスキングするには、各スレッドについてシグナルをマスキングする必要があります。

  親スレッドが生成したスレッドは、親スレッドのシグナルのマスキングを継承します。シグナルのマスキング継承を利用するアプリケーションを構築することができます。シグナルをプロセス全体にわたってマスキングする必要があるときは、メイン・スレッド、または最初のスレッドでシグナルをマスキングします。その後作成されたすべてのスレッドは、このスレッドのシグナルのマスキングを継承します。ネイティブ・スレッドのサポートの詳細については、使用しているプラットフォームの『Open Client/Server プログラマーズ・ガイド補足』を参照してください。

## シグナルの配信

非スレッド環境では、制御スレッドが 1 つしかありません。同期シグナルと非同期シグナルはプロセスに配信されます。

マルチスレッド環境では、複数のスレッドが複数の制御スレッドを表します。同期シグナルは、例外を発生させたスレッドに常に配信されます。非同期シグナルは、シグナルの配信が可能な、最初の実行スレッドに配信されます。

1 つまたは一連のスレッドに対して非同期シグナルを配信するように指定できます。一連のスレッドに対してシグナルをマスキング解除して、これらのスレッドにシグナルを配信できるようにします。その他のすべてのスレッドに対してシグナルをマスキングすると、シグナルが配信されないようにできます。シグナルを配信できるスレッドが実行されるまで、カーネルはシグナルを保持します。

## sigwait を使用した非同期シグナルの処理

Client-Library と Server-Library は、ct_callback ルーチンと srv_signal ルーチンを使用するシグナル・ハンドラのインストールをサポートしています。非同期シグナルセーフではない関数を使用できるようにするには、マルチスレッド・アプリケーションは通常のシグナル・ハンドラをインストールするのではなく、sigwait を呼び出して保留中のシグナルを取得するスレッドをインストールする必要があります。ct_callback ルーチンと srv_signal ルーチンがシグナル・ハンドラ関数をこの方法で正しくインストールできるように、cs_ctx_alloc または cs_ctx_global への最初の呼び出しが行われると、1 つを除いてすべてのスレッドでシグナルがブロックされます。スレッド・シグナルのブロックの開始と停止をこの 1 つのスレッドに指示できます。このスレッドを「キャッチャ・スレッド」と呼びます。シグナル・ハンドラが、ct_callback ルーチンまたは srv_signal ルーチンを使用してインストールされると、キャッチャ・スレッドは対応するシグナルをブロックします。すると、別のスレッドが生成され、このシグナルの sigwait を呼び出し、シグナルを受信したときに適切なユーザ指定のシグナル・ハンドラ関数を実行します。

---

**注意**　この機能が実現されるのは、スレッドを作成する前に cs_ctx_alloc ルーチンと cs_ctx_global ルーチンを呼び出すアプリケーションで、かつ ct_callback ルーチンまたは srv_signal ルーチンを使用してシグナル・ハンドラをインストールするアプリケーションだけです。

---

Open Client/Open Server のライブラリが介入してスレッド・シグナルを処理することは望ましくないこともあります。Open Client/Open Server によるスレッド処理を無効にして、アプリケーション自体にシグナル処理をさせるには、次の手順に従います。

❖ **アプリケーションによるスレッド・シグナル処理の有効化**

1. メイン・プロセス・スレッドで、cs_ctx_alloc または cs_ctx_global への最初の呼び出しの前に処理するシグナルをブロックします。
2. ダミーのシグナル・ハンドラをインストールして、シグナル・ハンドラが SIG_IGN に設定されないようにします。
3. キャッチャ・スレッドを含むプロセス・スレッドを作成するため、cs_ctx_global を呼び出します。キャッチャ・スレッドはシグナルをブロックします。

4 次のいずれかを実行します。

- 独自のスレッドをインストールし、sigwait を呼び出す。
- シグナルのブロックを解除して、通常のシグナル・ハンドラをインストールする (sigaction などを使用 )。シグナル・ハンドラが非同期シグナルセーフであることを確認する。

5 メイン・プロセス・スレッドから、手順 1 でブロックされたシグナルのブロックを解除します。これで、すべての Open Client/Open Server スレッドでシグナルがブロックされます。メイン・プロセス・スレッドでシグナルがブロックされることはなく、メイン・スレッドにスレッドが直接作成されることもありません。ct_callback または srv_signal を使用してシグナル・ハンドラをインストールしないでください。

## Sybase の専用シグナル・ハンドラ

非スレッド環境では、UNIX システム・コールを使用してシグナルをマスキング／マスキング解除できます。

マルチスレッド環境では、Open Client/Server 12.0 より前のバージョンを使用した場合、Sybase のライブラリによって内部で使用されているスレッドのマスキングを変更できません。しかし、Sybase ライブラリのバージョン 12.0 以降を使用すると、シグナルのマスキング／マスキング解除に次の 2 つの専用シグナル・ハンドラを使用できます。

- CS_SIGNAL_BLOCK － シグナルをマスキングするには、Sybase が提供するシグナル・インストール・ルーチンを使用して、このシグナル・ハンドラをインストールします。シグナルが発生すると、マスキング解除するまでシグナルが保持されます。
- CS_SIGNAL_UNBLOCK － シグナルをマスキング解除するには、Sybase が提供するシグナル・インストール・ルーチンを使用して、このシグナル・ハンドラをインストールします。

マルチスレッド環境のその他の特殊なシグナル・ハンドラには、次のものがあります。

- CS_SIGNAL_IGNORE － シグナルを無視します。CS_SIGNAL_IGNORE は、UNIX の専用シグナル・ハンドラ SIG_IGN と同様の働きをします。
- CS_SIGNAL_DEFAULT － シグナルが発生したときに、デフォルトのアクションを実行します。CS_SIGNAL_DEFAULT は、UNIX の専用シグナル・ハンドラ SIG_DFL と同様の働きをします。

## SIGTRAP シグナル

Sybase のライブラリでは、非同期シグナル SIGTRAP を同期シグナルとして扱います。

非同期シグナルとして扱われると、アプリケーションが srv_init または ct_init を呼び出したときに、このシグナルが呼び出しスレッド上でマスキングされます。これにより、デバッグができなくなります。多くのデバッガは、デバッグ中のアプリケーションと通信するために SIGTRAP を使用するためです。デバッグに悪影響を与えないために、SIGTRAP は同期シグナルとして扱われます。

---

**注意**　UNIX では、非同期シグナルを同期シグナルのようには処理できません。Sybase が提供する呼び出しを使用して、SIGTRAP のシグナル・ハンドラをインストールすることはできません。

---

## Sun の ALARM ルーチンと SETITIMER ルーチンの使用

Solaris 2.8 で、Sun のルーチンである ALARM または SETITIMER を使用する場合は、各ルーチンの man ページで説明されているバグを確認してください。

バグに対処するには、alarm を使用し、次のように *thread* の前に *pthread* をリンクしてください。

```
-lpthread -lthread
```

# マルチスレッド・プログラミング

このバージョンの Client-Library は、マルチスレッド・プログラミングをサポートしています。マルチスレッド・アプリケーションは、Client-Library に含まれているマルチスレッド・ライブラリにリンクさせる必要があります。

すべてのオペレーティング・システムにスレッドがあるわけではなく、Client-Library がシステム上で使用可能なスレッド・インタフェースをすべてサポートしているわけでもありません。Client-Library がサポートするスレッド *interfaces* については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

スレッドのサポートを使用できないプラットフォームの場合、アプリケーションでは Client-Library の非同期インタフェースを使用して同じ効果を得ることができます。「非同期プログラミング」を参照してください。

## スレッドについて

メモリ内の、プログラムを介した実行のパスです。従来のシステムでは、システム上の個々のプロセスには 1 つのみ実行スレッドがありました。マルチスレッド・システムでは、1 つのプロセス内で複数のスレッドを開始できます。プロセス内のスレッドは、メモリやオープン可能なファイル記述子などのプロセス・リソースに対する同一のアクセスを共有します。

一般に、マルチスレッド・システムには次の機能があります。

- スレッドを作成および破棄するスレッド管理ルーチン。
- 同一プロセス内の複数スレッドの同時実行を管理するスレッド・スケジューラ。
- 各スレッドからの共有リソースへのアクセスが相互排他的であることを保証するスレッド・シリアライゼーション・プリミティブ。つまり、あるスレッドが共有リソースへのアクセスを開始すると、それが終了するまで他のスレッドがそのリソースにアクセスできなくなります。

  たとえば、リンクリストが複数のスレッドで共有されている場合、検索、挿入、削除の操作は、他の検索、挿入、削除との逐次化が必要なクリティカル・セクションです。このようなクリティカル・セクションはすべて逐次化して、あるスレッドのクリティカル・セクションの実行中に他のスレッドの関連クリティカル・セクションが割り込んで実行されないようにする必要があります。

  シリアライゼーション・プリミティブは、ロックできるオブジェクト (mutex など ) とオブジェクトのロックとロック解除を行うルーチンで構成されます。
- 複数のスレッドで実行される依存アクションの同期化に使用されるスレッド同期プリミティブ。同期プリミティブは、同期オブジェクト ( 条件変数など )、条件を待つルーチン、条件が満たされたことを知らせるルーチンで構成されます。

## 複数スレッドの利点

アプリケーション設計者は、複数スレッドを使用すると、プログラムの複数の部分が同時に実行されるようにできます。

たとえば、対話型 Client-Library アプリケーションの場合、あるスレッドを使用してサーバに問い合わせを行う一方で、別のスレッドでユーザ・インタフェースを管理できます。このようなアプリケーションは、ユーザに対する応答能力がより優れているといえます。これは、クエリ・スレッドが結果を待っている間も、ユーザ・インタフェース・スレッドがユーザ・アクションに応答できるからです。

また、1 つ以上のサーバに対して複数の接続を使用するアプリケーションもあります。このようなアプリケーションは、接続のそれぞれを専用スレッド内で実行できます。1 つのスレッドがコマンドの結果を待っている間に、他のスレッドは受け取った結果を処理したり、新しいコマンドを送信したりできます。このような処理方式では、アプリケーションが結果を待つアイドル時間が短縮され、スループットが向上します。

複数スレッドを使用する理由として、複数プロセッサのシステムの場合に、システムのスレッド・ライブラリが、アプリケーションの各スレッドを異なるプロセッサで実行されるようにスケジュールできることも挙げられます。

スレッドは、Client-Library プログラムの同時実行性を実現する 1 つの方法を提供します。別の方法として、Client-Library の非同期プログラミング・インタフェースを使用することもできます。非同期プログラミングを使用すると、同時実行性を制限できます。「非同期プログラミング」(12 ページ ) を参照してください。

## スレッドの種類

アプリケーション設計者は、複数スレッドを使用すると、プログラムの複数の部分が同時に実行されるようにできます。

- native スレッドは、オペレーティング・システム・ルーチンに対する直接呼び出しを介してアプリケーションによって作成され、オペレーティング・システムによってスケジュールされるスレッドです。
- Open Server スレッドは、Server-Library によって作成およびスケジュール管理されるスレッドです。ゲートウェイ・アプリケーションは Open Server スレッドを使用します。

場合によっては、native スレッドを使用して、Open Server スレッドを実装できることもあります。ただし、Open Server アプリケーションでは、Open Server が native スレッド実装を使用している場合でも常に Server-Library ルーチンを呼び出してスレッド操作が管理されます。このマニュアルでは、native スレッドという用語を、システム・ルーチンに対するアプリケーション呼び出しによって直接作成されるスレッドを意味するものとして使用しています。

native スレッドは、すべてのプラットフォームで使用可能であるとは限らないため注意が必要です。また、一部のプラットフォームでは、オペレーティング・システムがシステムレベルのスレッドを提供していない場合でも、DCE pthread ライブラリを使用できます。これらのプラットフォームでは、DCE スレッドで使用するための特定バージョンの Client-Library ライブラリ・ファイルが提供されていることがあります。詳細については、使用しているプラットフォームの『Open Client/Server プログラマーズ・ガイド補足』を参照してください。

各プラットフォームの『Open Client/Server プログラマーズ・ガイド補足』には、Client-Library とシステムで使用可能なスレッド・インタフェースの使用に関するプラットフォーム固有の重要な情報が記載されています。

## スレッドセーフ・コードの書き込み

スレッドを使用するとアプリケーションは複数のタスクを同時に実行できますが、プログラム・ロジックが複雑になります。マルチスレッド・プログラムは、スレッドセーフになるようにコーディングする必要があります。スレッドセーフ・プログラムは次の条件を満たすことが必要です。

1. データの読み込みと書き込みの一貫性が保持され、アトミックになるように、共有データ ( グローバル変数など ) へのアクセスを逐次化します。「共有データおよび共有リソースに対する アクセスの逐次化」を参照してください。
2. リソースの一貫性が保持されるように、共有リソース ( ファイル記述子など ) へのアクセスを逐次化します。「共有データおよび共有リソースに対する アクセスの逐次化」を参照してください。
3. 複数のスレッドの依存アクションを、要求された順序で実行されるように同期化します。「依存アクションの同期化」を参照してください。

4　一度に 1 つの呼び出しのみが有効になるように、スレッドアンセーフなシステム・ルーチンに対する呼び出しを逐次化します。「スレッドアンセーフなシステム・ルーチンの呼び出し」を参照してください。

5　デッドロックを回避できるように、スレッド・シリアライゼーション・プリミティブを使用します。「デッドロックの回避」を参照してください。

6　CS-Library ルーチン、Client-Library ルーチン、Bulk-Library ルーチンに対する呼び出しが、「マルチスレッド・プログラムに対する Client-Library の 制限事項」で説明している制限を満たすことが必要です。

これらの制限を満たしていないプログラム・コードは「スレッドアンセーフ」です。一般に、「スレッドアンセーフ」のコードをマルチスレッド・プログラムで実行した場合の動作は、予測不能です。制限 1 〜 5 はアプリケーションをスレッドセーフにするための一般的な規則です。制限 6 は Client-Library アプリケーションに固有のものです。以降では、これらの制限の詳細について説明します。

---

**注意**　ここでの説明は、使用しているシステムのスレッド・インタフェースのマニュアルの代わりとなるものではありません。システムのマニュアルを熟読し、理解してから、マルチスレッド環境で Client-Library を使用するようにしてください。

---

**注意**　Client-Library に含まれているマルチスレッド・ライブラリを使用する場合、exec() を使用しないと、fork() はサポートされません。これは、I/O 操作には Net Library スレッドが必要で、fork() システム・コールにより子で複製されないからです。

---

## 共有データおよび共有リソースに対するアクセスの逐次化

すべてのスレッドで同じメモリその他のプロセス・リソースが共有されるため、別のスレッドでデータやリソースが変更されると、一貫性が損なわれる場合があります。この問題は、シリアライゼーション・プリミティブを正しく使用し、データ・アクセスがアトミックであるようにして回避します。

たとえば、複数のスレッドでグローバル・カウンタ変数の読み込みおよび増分が実行される場合は、カウンタへのアクセスが逐次化されるようにアプリケーションを設計する必要があります。ミューテックスとカウンタを関連付け、カウンタの読み込みと増分の前にミューテックスをロックするコードを追加すると、それぞれのデータ・アクセスをアトミックにできます。

一般に、絶対に必要な場合以外は、リソースの共有を避けてください。シリアライゼーション・プリミティブの使用は、プログラムを複雑にします。また、一部のシステムでは、過度のロック呼び出しによってパフォーマンスが低下する場合があります。

使用するスレッド・システムのマニュアルを熟読し、使用可能なシリアライゼーション・プリミティブとその使用方法を理解してください。

## 依存アクションの同期化

異なるスレッドで実行される依存アクションでは、複数のスレッドが同時に実行されるため、同期化してアクションが適切な意図したとおりの順序で実行されるようにする必要があります。アプリケーションは、条件変数などの同期プリミティブを使用して希望どおりの順序で実行されるように設計する必要があります。

たとえば、複数のスレッドがキューを共有している場合に、キューから読み込むスレッド ( コンシューマ・スレッド ) と、同じキューに対する書き込みを行うスレッド ( プロデューサ・スレッド ) が存在することもあります。このような場合、キューへのアクセスの逐次化と同期化の両方が必要になります。逐次化はキューのデータの一貫性を保ちます。同期化は、コンシューマが空のキューから読み込んだり、プロデューサが一杯になったキューに書き込んだりしないようにします。

ミューテックス queue_mutex と条件変数 *queue_notempty* と *queue_notfull* をキューに関連付けることで、両方の条件を満たすことができます。POSIX pthread インタフェースを使用する場合は、プロデューサ・スレッドのそれぞれが以降のステップを実行してキューにデータを挿入します。

```
pthread_mutex_lock(queue_mutex)
while queue is full
pthread_cond_wait(queue_notfull, queue_mutex)
end while
insert an item
pthread_cond_signal(queue_notempty, queue_mutex)
pthread_mutex_unlock(queue_mutex)
```

一方、コンシューマ・スレッドは、以降のステップを実行してキューからデータを読み込みます。

```
pthread_mutex_lock(queue_mutex)
while queue is empty
pthread_cond_wait(queue_notempty, queue_mutex)
end while
remove an item
pthread_cond_signal(queue_notfull, queue_mutex)
pthread_mutex_unlock(queue_mutex)
```

コンシューマ・スレッドは、キューが空であることを検出すると *queue_notempty* 条件の pthread_cond_wait ルーチンを呼び出します。この呼び出しは、プロデューサ・スレッドが *queue_notempty* が指定された pthread_cond_signal を呼び出すまで、制御を戻しません。プロデューサ・スレッドがアイテムを挿入すると、pthread_cond_signal がシグナルに呼び出され、*queue_notempty* 条件が満たされたことが通知されます。

使用するスレッド・システムのマニュアルを熟読し、使用可能な同期プリミティブとその使用方法を理解してください。

## スレッドアンセーフなシステム・ルーチンの呼び出し

マルチスレッド・コードからスレッドアンセーフなルーチンが呼び出される場合は、それぞれの呼び出しを逐次化して、アンセーフ・ルーチンに対する呼び出しが同時に有効にならないようにします。このような場合は、「グローバル・ロック」などのシリアライゼーション・プリミティブを使用できます。

システムによっては、C 標準ライブラリ・ルーチンの一部が「スレッドセーフ」ではありません。マルチスレッド・コードでルーチンが呼び出される場合、そのルーチンのマニュアルを参照して、「スレッドセーフ」かどうか、および「スレッドセーフ」の使用条件を調べてください。

Client-Library ルーチンについては、「スレッドセーフ」の使用についてまとめた項「マルチスレッド・プログラムに対する Client-Library の 制限事項」を参照してください。

## デッドロックの回避

マルチスレッド・コードでは、2 つのスレッドのそれぞれがもう一方によって保持されているロックを要求する場合にデッドロックが発生することがあります。たとえば、スレッドが 2 つ ( スレッド 1 とスレッド 2) とミューテックスが 2 つ (A と B) があるとします。その場合、次のような状況ではデッドロックが発生します。

```
スレッド 1 が B をロックする
スレッド 2 が A をロックする
スレッド 2 が B のロックを要求する
スレッド 1 が A のロックを要求する
```

この状況では、スレッド 1 とスレッド 2 は両方とも、要求したロックを永遠に待ち続けることになります。

一般にデッドロックは、アプリケーション用の「ロック・プロトコル」を設計して回避します。これらのプロトコルでは、同時に保持されるロックが要求される順序を指定します。上記の状況でも、このプロトコルを「ミューテックス A と B の両方が使用される場合、A が最初に取得されなければならない」のように記述する必要があります。

一部のシステムでは、スレッドがすでに保持しているロックを要求すると、そのスレッド自体によるデッドロックが発生することがあります。推奨されるデッドロックの回避方法については、使用しているスレッド・システムのマニュアルを参照してください。

## マルチスレッド・プログラムに対する Client-Library の制限事項

Client-Library アプリケーションをスレッドセーフにするには、前述の一般的な制限事項と次に説明する Client-Library 固有の使用制限を満たす必要があります。

スレッドセーフの使用に関する Client-Library の制限は、以下のとおりです。

- **コンテキスト・レベル** － CS_CONTEXT 構造体へのアクセスに関するスレッドセーフの制限。詳細については、「コンテキスト・レベル・ルーチンの呼び出し」を参照してください。
- **接続レベル** － CS__CONNECTION 構造体と下位の構造体 (CS_COMMAND、CS_BLKDESC) の使用に関するスレッドセーフの制限。詳細については、「接続レベル・ルーチンの呼び出し」を参照してください。

- **CS_LOCALE の使用** － CS_LOCALE 構造体の使用に関するスレッドセーフの制限。詳細については、「CS_LOCALE 構造体の使用」を参照してください。
- **コンテキスト・レベル** － CS_CONTEXT 構造体へのアクセスに関するスレッドセーフの制限。詳細については、「コンテキスト・レベル・ルーチンの呼び出し」を参照してください。

## コンテキスト・レベル・ルーチンの呼び出し

Client-Library および CS-Library のコンテキスト・レベルのルーチンのリストは、表 2-27 にあります。

コンテキスト・レベル・ルーチンに対するスレッドセーフな呼び出しは、次の制限に従います。

- cs_ctx_alloc および cs_ctx_drop に対する呼び出しは、cs_ctx_alloc または cs_ctx_drop に対する他の呼び出しと同時には発生してはなりません。
- ct_init および ct_exit に対する呼び出しは、ct_init または ct_exit に対する他の呼び出しと同時に発生してはなりません。
- CS_CONTEXT 構造体が複数のスレッドで共有され、その CS_CONTEXT 構造体に対してスレッドアンセーフな呼び出しが実行される場合、この CS_CONTEXT のコンテキスト・レベル・ルーチンに対するすべての呼び出しを逐次化する必要があります。スレッドアンセーフなコンテキスト・レベルの呼び出しを 表 2-27 に示します。

**表 2-27：CS-Library と Client-Library のコンテキスト・レベル・ルーチンのスレッドセーフな使用**

| ルーチン名 | スレッドセーフな呼び出し | スレッドアンセーフな呼び出し | 注意 |
|---|---|---|---|
| cs_calc | すべて。 | | |
| cs_cmp | すべて。 | | |
| cs_config | action が CS_GET の場合。 | action が CS_SET または CS_XCLEAR の場合。 | |

| ルーチン名 | スレッドセーフな呼び出し | スレッドアンセーフな呼び出し | 注意 |
| --- | --- | --- | --- |
| cs_convert | すべて。 | | srcfmt または destfmt 内で CS_LOCALE ポインタを使用する場合、CS_LOCALE 構造体へのアクセスを「スレッドセーフ」にする必要がある。 |
| cs_ctx_alloc | | すべて。 | すべての context に対してスレッドアンセーフ。「コンテキストの初期化とクリーンアップ」を参照。 |
| cs_ctx_drop | | すべて、あらゆる context に対して。 | すべての context に対してスレッドアンセーフ。「コンテキストの初期化とクリーンアップ」を参照。 |
| cs_ctx_global | 最初の呼び出しが完了した後のすべての呼び出し。 | 最初に実行された呼び出しのみ。 | |
| cs_dt_crack | すべて。 | | |
| cs_dt_info | すべて。 | | CS_LOCALE 構造体へのアクセスを「スレッドセーフ」にする必要がある。 |
| cs_diag | action が CS_INIT でない場合。 | action が CS_INIT の場合。 | 呼び出し元のスレッドによって生成されたメッセージのみが表示される。「CS-Library のエラー処理」を参照。 |
| cs_loc_alloc | すべて。 | | CS_LOCALE 構造体へのアクセスを「スレッドセーフ」にする必要がある。「CS_LOCALE 構造体の使用」を参照。 |
| cs_loc_drop | すべての呼び出し。 | | |
| cs_locale | すべての呼び出し。 | | |
| cs_objects | すべての呼び出し。 | | |
| cs_set_convert | action が CS_GET の場合。 | action が CS_SET または CS_XCLEAR の場合。 | |

| ルーチン名 | スレッドセーフな呼び出し | スレッドアンセーフな呼び出し | 注意 |
|---|---|---|---|
| cs_setnull | | 同じ context を共有するすべての呼び出し。 | |
| cs_strbuild | すべての呼び出し。 | | |
| cs_strcmp | すべての呼び出し。 | | |
| cs_time | すべての呼び出し。 | | |
| cs_will_convert | すべての呼び出し。 | | |
| ct_callback | context が NULL でない場合。 | | 接続レベルでスレッドアンセーフ (context が NULL のすべての呼び出し)。「接続レベル・ルーチンの呼び出し」を参照。 |
| ct_con_alloc | すべての呼び出し。 | | |
| ct_con_drop | すべての呼び出し。 | | |
| ct_config | action が CS_GET の場合。 | action が CS_SET または CS_XCLEAR の場合。 | |
| ct_debug | | context が NULL でない場合。 | context が NULL の場合は、「接続レベル・ルーチンの呼び出し」を参照。 |
| ct_exit | | すべての呼び出し、あらゆる context に対して。 | すべての context に対してスレッドアンセーフ。「コンテキストの初期化とクリーンアップ」を参照。 |
| ct_init | | すべての呼び出し、あらゆる context に対して。 | すべての context に対してスレッドアンセーフ。「コンテキストの初期化とクリーンアップ」を参照。 |
| ct_poll | | context が NULL でない場合のみ。 | |

## コンテキストの初期化とクリーンアップ

cs_ctx_alloc、cs_ctx_drop、ct_init、ct_exit の各ルーチンはスレッドアンセーフで、次のように逐次化する必要があります。

- cs_ctx_alloc または cs_ctx_drop に対する呼び出しはすべて、cs_ctx_alloc または cs_ctx_drop に対する他の呼び出しとともに逐次化する必要があります。

- また、ct_init または ct_exit に対する呼び出しはすべて、ct_init または ct_exit に対する他の呼び出しとともに逐次化する必要があります。

プログラムが、必要なすべての CS_CONTEXT 構造体の割り付けと初期化をシングルスレッドの初期化コードで実行し、コンテキスト・レベルのクリーンアップ・オペレーションをシングルスレッドのクリーンアップ・コードで実行する場合は、この点を考慮する必要はありません。また、特定のコンテキスト構造体の使用をシングルスレッドに限定して逐次化の必要をなくす方法もあります。

### CS-Library のエラー処理

複数のスレッドでコンテキストを共有する場合、すべてのスレッドで、同じ方法を使用してエラーを処理する必要があります。コンテキストを共有するすべてのスレッドが、インライン (cs_diag) 方式、またはコールバック方式のいずれかのみを使用する必要があります。

エラーが cs_diag によってインラインで処理される場合は、各スレッドが cs_diag を呼び出して、それぞれの CS-Library エラー・メッセージを取得する必要があります。cs_diag は、呼び出し元のスレッドが生成したメッセージのみを表示します。

## 接続レベル・ルーチンの呼び出し

接続レベル・ルーチンとは、次の構造体のいずれかに対するポインタを引数とする Client-Library または Bulk-Library のルーチンです。

- 接続構造体 (CS_CONNECTION *)
- コマンド構造体 (CS_COMMAND *)
- ディレクトリ・オブジェクト構造体 (CS_DS_OBJECT *)
- Bulk-Library バルク記述子構造体 (CS_BLKDESC *)

次のルーチンも接続レベル・ルーチンとみなされます。

- ct_callback ( 非 NULL 接続引数が指定された場合のみ )。
- ct_debug (NULL コンテキスト引数が指定された場合のみ )。ct_debug(CS_DBG_ALL) およびその他の非 NULL コンテキストを必要とする呼び出しは、スレッドアンセーフのコンテキスト・レベルの呼び出しとみなされます。
- ct_poll ( 非 NULL 接続引数が指定された場合のみ )。

コマンド構造体、ディレクトリ・オブジェクト構造体、またはバルク記述子構造体を使用してルーチンを呼び出した場合は、親接続の接続レベルの呼び出しとして処理されます。接続レベル・ルーチンに対するスレッドセーフな呼び出しは、次の制限に従います。

- 同じ接続を共有するスレッドは、接続レベルの呼び出しを同期化する必要があります。その目的は、以下のとおりです。
    - 呼び出しが同時に有効にならないようにする。
    - 意図した順序で呼び出しが実行されるようにする。
- 指定の接続構造体を参照する呼び出しは、接続の親コンテキストにアクセスするスレッドアンセーフのコンテキスト・レベルの呼び出しと同時に有効にすることはできません。スレッドアンセーフなコンテキスト・レベルの呼び出しのリストは、表 2-27 にあります。

## CS_LOCALE 構造体の使用

Client-Library、CS-Library、Bulk-Library の各ルーチンは、公開された構造体内で CS_LOCALE 構造体に対するポインタを直接的または間接的に受け取ることができます。公開された構造体の CS_DATAFMT および CS_IODESC のそれぞれに CS_LOCALE ポインタを保持できるロケール・フィールドがあります。

非 NULL の CS_LOCALE ポインタを受け取る Client-Library、CS-Library、または Bulk-Library の各ルーチンに対する呼び出しはすべて、次の条件を満たすかぎりスレッドセーフになります。

- ルーチンによって CS_LOCALE 構造体が変更されないこと。この後の表に、CS_LOCALE 構造体を変更するルーチンを示します。
- 以下の呼び出しがいずれも、同じ CS_LOCALE 構造体を参照する他の呼び出しと同時には有効になりません。
    - cs_dt_info(CS_SET)
    - cs_loc_drop
    - cs_locale(CS_SET)

## スレッドセーフなコールバック・ルーチンのコーディング

コールバック関数を複数のスレッドから呼び出せる場合は、コールバックを「スレッドセーフ」にする必要があります。一般に、コールバックは、コールバック・イベントを発生させるスレッドから呼び出されますが、一部のコールバックは、内部の Client-Library のワーカー・スレッドから呼び出され、ワーカー・スレッドのコンテキスト内で実行されます。表 2-28 (197 ページ ) は、どのスレッドがどのタイプのコールバックを呼び出すかをまとめたものです。

マルチスレッド環境で使用されるコールバックは、次の規則に従ってコーディングしてください。

- コールバックは、「コールバック」で説明している一般的な実装規則に従ってください。
- コールバック・コードは、スレッドセーフにして、「スレッドセーフ・コードの書き込み」に示した制限に従ってください。
- Client-Library のワーカー・スレッドによる呼び出しが可能なコールバックは、それに適した設計が必要です。コールバックは、メインライン・コードと同時に実行でき、コールバックとメインライン・コード間で共有されるすべてのデータ構造 (CS_USERDATA プロパティとして格納されるデータを含む ) に対するアクセスを「スレッドセーフ」にしてください。

## スレッドと完全非同期モード

Windows など、プラットフォームによっては、Client-Library がネットワーク I/O を処理する内部ワーカー・スレッドを生成して、完全非同期のネットワーク I/O を実装します。このようなプラットフォームで完全非同期 I/O が有効になると、内部 Client-Library スレッドはそれぞれの I/O 要求の完了を待ってから、アプリケーションの完了コールバックを呼び出します。オペレーションの完了前に、スレッドが他のアプリケーション・コールバックを呼び出すことがあります。たとえば、サーバがサーバ・メッセージを送信すると、内部 Client-Library スレッドはメッセージを読み込み、アプリケーションのサーバ・メッセージ・コールバックを呼び出します。

このような場合、コールバック・コードとアプリケーションのメインライン・コードが別のスレッドで実行されます。Client-Library がスレッド駆動型 I/O を使用するプラットフォーム上で完全非同期アプリケーションをコーディングする場合は、コールバックとメインライン・コードとの通信が正しく行われることを確認する必要があります。表 2-28 は、どのスレッドがどのタイプのコールバックを呼び出すかをまとめたものです。

**表 2-28：コールバック・タイプおよびそれらを呼び出すスレッド**

| コールバックのタイプ | 呼び出しスレッド |
|---|---|
| CS-Library エラー・ハンドラ | エラー・イベントを発生させるスレッド。 |
| クライアント・メッセージ | CS_NETIO 接続プロパティが CS_DEFER_IO または CS_SYNC_IO のときには、エラーを発生させたスレッドからコールバックが呼び出される。<br>スレッド駆動型 I/O プラットフォームでは、CS_NETIO 接続プロパティが CS_ASYNC_IO のとき、Client-Library のワーカー・スレッドまたはエラー・イベントを発生させたスレッドからコールバックを呼び出すことができる ( どちらで呼び出すかは、エラーがいつ検出されたかによって決まる )。 |
| 完了 | CS_NETIO 接続プロパティが CS_DEFER_IO のときには、ct_poll を呼び出したスレッドからコールバックを呼び出すことができる。<br>スレッド駆動型 I/O プラットフォームでは、CS_NETIO 接続プロパティが CS_ASYNC_IO のとき、Client-Library ワーカー・スレッドからコールバックが呼び出される。 |
| ディレクトリ | CS_NETIO 接続プロパティが CS_SYNC_IO または CS_DEFER_IO のときには、ct_ds_lookup を呼び出したスレッドからコールバックが呼び出される。<br>スレッド駆動型 I/O プラットフォームでは、CS_NETIO 接続プロパティが CS_ASYNC_IO のとき、コールバックは内部の Client-Library スレッドから呼び出される。 |
| 暗号化、ネゴシエーション、セキュリティ・セッション | CS_NETIO 接続プロパティが CS_SYNC_IO または CS_DEFER_IO のときには、ct_connect を呼び出したスレッドからコールバックが呼び出される。<br>スレッド駆動型 I/O プラットフォームでは、CS_NETIO 接続プロパティが CS_ASYNC_IO のとき、コールバックは内部の Client-Library スレッドから呼び出される。 |

| コールバックのタイプ | 呼び出しスレッド |
|---|---|
| ノーティフィケーション | CS_ASYNC_NOTIFS プロパティが CS_FALSE ( デフォルト ) のときには、ノーティフィケーション到着時にネットワークからの読み込みを実行していたスレッドから、コールバックが呼び出される。<br>スレッド駆動型 I/O プラットフォームでは、CS_ASYNC_NOTIFS プロパティが CS_TRUE のとき、内部の Client-Library スレッドからコールバックが呼び出される。 |
| サーバ・メッセージ | CS_NETIO 接続プロパティが CS_DEFER_IO または CS_SYNC_IO のときには、メッセージ到着時に、結果を処理するスレッドまたは接続にコマンドを送信するスレッドからコールバックが呼び出される。<br>スレッド駆動型 I/O プラットフォームでは、CS_NETIO 接続プロパティが CS_ASYNC_IO のとき、内部 Client-Library スレッドからコールバックが呼び出される。 |
| シグナル | シグナルをサポートし、Client-Library がスレッド駆動型 I/O を使用するプラットフォームでは、ct_callback でシグナル・コールバックをインストールする。シグナル・コールバックは、割り込みレベルではなく、内部 Client-Library スレッドによって呼び出される。 |

## Client-Library のマルチスレッド・プログラミング・モデル

この項では、マルチスレッド・アプリケーションの設計を単純化するためのプログラミング方針の概要を説明します。

### 1 スレッド、1 接続のモデル

このモデルでは、プログラムは次のようになります。

- シングルスレッドの初期化およびクリーンアップ・コードで、必要なライブラリの初期化とクリーンアップをすべて実行します。
- 接続ごとに専用のスレッドを作成し、個々の接続で専用スレッドのみが使用されるようにします。

1 スレッド、1 接続のモデルは、最も単純で、スレッド間の同期が最小限に抑えられます。Open Server ゲートウェイの最も自然なモデルでもあります。

基本的な手順は、次のとおりです。

- 初期化 – スレッドアンセーフなコンテキスト・レベルの呼び出し ( 表 2-27 (190 ページ ) にリストを記載 ) はすべて、シングルスレッドの初期化コードで呼び出されます。アプリケーションが Client-Library ルーチンを呼び出すマルチスレッド・ライブラリの場合、ライブラリのパブリック初期化ルーチンで POSIX pthread_once() ルーチン ( または使用しているシステムの、これに相当するルーチン ) を呼び出して、Client-Library を初期化する内部のシングルスレッド・ルーチンを安全に呼び出すことができます。一般に、スタートアップ・スレッドは、プログラム ( またはライブラリ ) の終了を示すイベントを待ちます。
- 処理 – 初期化がすべて完了すると、アプリケーションは作成する接続のそれぞれに 1 つずつスレッドを生成します。次に、このスレッドは、ct_con_alloc を使用してスレッド自体の接続を割り付け、サーバに接続し、その接続の処理を実行します。
- 停止 – プログラムまたはライブラリが終了を決定すると、接続にバインドされていたそれぞれのスレッドが接続をクローズし、必要に応じてスレッド自体を終了します。この後、アプリケーションはシングルスレッド・コード内でクリーンアップ (ct_exit および cs_ctx_drop の呼び出しなど ) を実行します。

## ワーカー・スレッド・モデル

このモデルでは、利用可能な Client-Library 接続構造体のプールを管理するアプリケーションを設計します。接続は、複数スレッドによって共有できますが、特定の 1 接続を使用するスレッドは、一度に 1 つに限られます。スレッドは、Client-Library オペレーションの実行が必要になると、利用可能な接続を取得し、「使用不可」のマークを付けてから、接続レベルのオペレーションを実行します。接続レベルのオペレーションが完了すると、スレッドは接続に「使用可能」のマークを付けます。アプリケーションの設計では、CS_USERDATA 接続プロパティを使用して、ステータス情報 ( 可用性など ) と接続構造体を関連付けます。

このモデルは、1 スレッド、1 接続モデルと似ていますが、接続とスレッドのバインドが静的でなく動的である点が異なります。「接続プール」の管理に使用されるアプリケーション・コードとデータ構造体はスレッドセーフである必要があります。

### その他のスレッド・モデル

他のプログラミング・モデルを使用した場合、共有接続が複数のスレッドで有効になったり、スレッドアンセーフなコンテキスト・レベルの呼び出しがマルチスレッド・コードで実行されることがあります。このような状況になると、同期がさらに複雑になります。プログラムは、この「マルチスレッド・プログラミング」の項で説明しているすべての制限に従うことが必要です。

## オプション

オプションは、Sybase Server がコマンドに応答する方法に影響を与えます。

アプリケーションはオプションを設定して、サーバのクエリ処理方法をカスタマイズします。たとえば、アプリケーションは CS_OPT_FIPSFLAG オプションを設定して、受信するすべての非標準 SQL コマンドを通知するようサーバに指示します。

Client-Library アプリケーションは、次のいずれかの方法で、Adaptive Server Enterprise のクエリ処理オプションの設定またはクリアを行います。

- Transact-SQL 言語コマンド (set) の使用
- ct_options の呼び出し

Client-Library とサーバ間の通信が混同する可能性があるため、アプリケーションは、これらの方法のいずれか一方だけを使用できます。

アプリケーションがオプションのステータスをチェックすることができるので、ct_options を呼び出す方法をおすすめします。これは、Transact-SQL の set コマンドによる方法では不可能です。

詳細については、『ASE リファレンス・マニュアル』の「set コマンド」を参照してください。

### 外部からのオプションの設定

Client-Library ルーチン ct_connect は、オプションで Open Client/Server ランタイム設定ファイルからあるセクションを読み込んで、新しくオープンした接続のサーバ・オプションを設定します。

この機能については、「ランタイム設定ファイルの使い方」(352 ページ ) を参照してください。

表 2-29 は、ct_options で使用される記号定数を示します。

**表 2-29：サーバ・オプションの記号定数**

| 記号定数 | 意味 | デフォルト値 |
|---|---|---|
| CS_OPT_ANSINULL | SQL の等号 (=) または不等号 (!=) の比較で NULL 値のオペランドの評価が ANSI 標準に準拠しているかどうかを判別する。<br>CS_TRUE の場合、Adaptive Server Enterprise は、「= NULL」と「is NULL」は同義ではないとする ANSI の動作を適用する。標準の SQL では、「= NULL」と「is NULL」は同義とみなされている。<br>このオプションは、「<> NULL」と「is not NULL」の動作にも同様に作用する。 | CS_FALSE |
| CS_OPT_ANSIPERM | Adaptive Server Enterprise による update 文と delete 文に対するパーミッションについてのチェックが ANSI 標準準拠であるかを判別する。<br>CS_TRUE の場合、Adaptive Server Enterprise は ANSI 標準準拠である。 | CS_FALSE |
| CS_OPT_ARITHABORT | 算術演算エラーが発生したときに Adaptive Server Enterprise がどのように動作するかを指定する。<br>このオプションが CS_TRUE に設定されている場合は、arith_overflow オプションと numeric_truncation オプションは on に設定される。明示的または暗黙的なデータ型変換中に 0 による除算エラーが発生したり精度が失われたりしたときには、エラーが発生したトランザクションまたはバッチ全体がロールバックされる。暗黙的なデータ型変換中に真数値タイプによって位取りが失われた場合、エラーを発生させた文はアボートされるが、トランザクションまたはバッチ内のその他の文の処理は続行される。<br>このオプションが CS_FALSE に設定されている場合は、arith_overflow オプションと numeric_truncation オプションは off に設定される。明示的または暗黙的なデータ型変換中に 0 による除算エラーを発生させたり精度を失わせたりした文はアボートされるが、トランザクションまたはバッチ内のその他の文の処理は続行される。暗黙的なデータ型変換中に真数値タイプによって位取りが失われた場合、クエリ結果はトランケートされ、トランザクションまたはバッチ内のその他の文の処理は続行される。 | CS_FALSE |

| 記号定数 | 意味 | デフォルト値 |
|---|---|---|
| CS_OPT_ARITHIGNORE | 0 による除算エラーまたは精度を失った後に、Adaptive Server Enterprise が、メッセージを返すかどうかを決定する。<br>このオプションが CS_TRUE に設定されている場合は、これらのエラーの後に警告メッセージは表示されない。このオプションが CS_FALSE に設定されている場合は、これらのエラーの後に警告メッセージが表示される。 | CS_FALSE |
| CS_OPT_AUTHOFF | 現在のサーバ・セッションについて、指定された権限レベルを無効にする。ユーザのログイン時に、そのユーザに与えられた権限がすべて自動的に無効になる。 | 適用されない |
| CS_OPT_AUTHON | 現在のサーバ・セッションについて、指定された権限レベルを有効にする。ユーザのログイン時に、そのユーザに与えられた権限がすべて自動的に有効になる。 | 適用されない |
| CS_OPT_CHAINXACTS | CS_TRUE の場合、Adaptive Server Enterprise は、連鎖トランザクション動作をする。連鎖トランザクション動作は、各サーバ・コマンドが別個のトランザクションとみなされていることを意味する。Adaptive Server Enterprise は delete、fetch、insert、open、select、update 文の前に、暗黙的に begin transaction を実行する。<br>CS_FALSE の場合、アプリケーションは明示的な commit transaction 文を指定して、トランザクションを終了してから新規に開始する。 | CS_FALSE |
| CS_OPT_<br>CURCLOSEONXACT | CS_TRUE の場合、トランザクションでオープンされるすべてのカーソルは、そのトランザクションの完了時にクローズされる。 | CS_FALSE |
| CS_OPT_DATEFIRST | 週の最初の日を設定する。 | us_english の場合、デフォルトは CS_OPT_SUNDAY |
| CS_OPT_<br>DATEFORMAT | *date*、*datetime*、または *smalldatetime* データの入力のため、日付部分の月／日／年の順序を設定する。 | us_english の場合、デフォルトは CS_OPT_FMTMDY |

| 記号定数 | 意味 | デフォルト値 |
|---|---|---|
| CS_OPT_FIPSFLAG | SQL 拡張機能が使用されたら Adaptive Server Enterprise が警告メッセージを表示するかどうかを決定する。<br>CS_TRUE の場合、Adaptive Server Enterprise は送信される標準でない SQL コマンドすべてにフラグを立てる。<br>CS_FALSE の場合、Adaptive Server Enterprise は ANSI 準拠でない SQL であることを示すフラグを立てない。 | CS_FALSE |
| CS_OPT_FORCEPLAN | CS_TRUE の場合、Adaptive Server Enterprise は、クエリの from 句でのテーブルのリスト順にテーブルをジョインする。 | CS_FALSE |
| CS_OPT_FORMATONLY | CS_TRUE の場合、Adaptive Server Enterprise は、select クエリに応答してデータそのものではなくデータの記述を送信する。<br>CS_FALSE の場合、Adaptive Server Enterprise は select クエリに応答してデータを送信する。 | CS_FALSE |
| CS_OPT_HIDE_VCC | CS_TRUE に設定される場合、CS_OPT_HIDE_VCC によりテーブルの仮想計算カラム (VCC) が隠される。このため、たとえば blk_bind に渡されるカラム番号には、VCC カラムが含まれなくなる。<br>CS_FALSE に設定される場合、VCC はテーブルに含められる。 | CS_FALSE |
| CS_OPT_IDENTITYOFF | テーブルの IDENTITY カラムへの挿入を無効にする。<br>Adaptive Server Enterprise のマニュアルで、set コマンド (identity_insert オプション ) の項を参照。 | 適用されない |
| CS_OPT_IDENTITYON | テーブルの IDENTITY カラムへの挿入を有効にする。<br>Adaptive Server Enterprise のマニュアルで、set コマンド (identity_insert オプション ) の項を参照。 | 適用されない |
| CS_OPT_IDENTITYUPD_OFF | ID 更新オプションを無効にする。 | Null |
| CS_OPT_IDENTITYUPD_ON | ID 更新オプションを有効にする。このオプションを使用すると、必要なローの範囲を指定して、それらのローを正しい値と置換する SQL 更新文を 1 つ作成して、「上限を超える」範囲外の identity カラム値を更新できる。 | Null |

| 記号定数 | 意味 | デフォルト値 |
|---|---|---|
| CS_OPT_ISOLATION | トランザクションの独立性レベルを指定する。指定できるレベルは CS_OPT_LEVEL0、CS_OPT_LEVEL1、CS_OPT_LEVEL3 である。<br>これらのレベルは、Adaptive Server Enterprise の set transaction isolation level コマンドの 3 つのレベルと対応している。CS_OPT_LEVEL0 を使用するには、Adaptive Server Enterprise が必要。 | CS_OPT_LEVEL1 |
| CS_OPT_NOCOUNT | 各 SQL 文の影響を受けるローの数を返すことを中止する。アプリケーションは、ct_res_info を呼び出すことによってこの情報を取得する。 | CS_FALSE |
| CS_OPT_NOEXEC | CS_TRUE の場合、Adaptive Server Enterprise は各クエリをコンパイルするが、実行はしない。<br>このオプションは、CS_OPT_SHOWPLAN とともに使用する。 | CS_FALSE |
| CS_OPT_PARSEONLY | CS_TRUE の場合、Adaptive Server Enterprise はクエリの構文をチェックし、必要に応じてエラー・メッセージを返すが、クエリの実行はしない。 | CS_FALSE |
| CS_OPT_QUOTED_ IDENT | CS_TRUE の場合、Adaptive Server Enterprise は、二重引用符 ( “) で囲まれたすべての文字列を識別子として扱う。 | CS_FALSE |
| CS_OPT_RESTREES | CS_TRUE の場合、Adaptive Server Enterprise はクエリの構文をチェックし、( 通常ロー結果セットにある image カラムのフォームで ) 構文解析ツリーと、必要に応じてエラー・メッセージを返すが、クエリの実行はしない。 | CS_FALSE |
| CS_OPT_ROWCOUNT | クエリの影響を受ける可能性があるローの数に制限を設定する。つまり、select 文を実行した結果返される通常ローの数と、update 文または delete 文の影響を受けるローの数を制限する。<br>このオプションが 0 に設定されている場合、コマンドを実行した結果返されたり影響を受けたりするローの数は制限されない。<br>このオプションが 0 よりも大きな値に設定されている場合、指定された数のローが影響を受けた時点で、Adaptive Server Enterprise はコマンドの処理を停止する。<br>このオプションは、返される計算ローの数を制限しない。 | 0 |

| 記号定数 | 意味 | デフォルト値 |
| --- | --- | --- |
| CS_OPT_SHOW_FI | CS_TRUE に設定された場合、CS_OPT_SHOW_FI は各機能インデックス (FI) のテーブルにカラムを追加する。<br>一方、CS_OPT_SHOW_FI が CS_FALSE に設定された場合、FI は隠される。 | CS_FALSE |
| CS_OPT_SHOWPLAN | それぞれのクエリの処理プランの記述が、そのクエリのコンパイルと実行の間に返されるかどうかを設定する。<br>このオプションが CS_TRUE に設定されている場合、Adaptive Server Enterprise はクエリをコンパイルして処理プランの記述を生成し、その後でクエリを実行する。<br>Client-Library は、その記述を一連の情報サーバ・メッセージとして受信する。アプリケーション・プログラムは、ユーザ指定サーバ・メッセージ・ハンドラによって、これらの統計にアクセスする。 | CS_FALSE |
| CS_OPT_SORTMERGE | セッション内でのソートマージ・ジョインの使用が有効か無効かを指定する。<br>詳細については、『パフォーマンス＆チューニング・ガイド』を参照。 | CS_FALSE |
| CS_OPT_STATS_IO | 各クエリで Adaptive Server Enterprise の内部 I/O 統計 ( スキャン数、論理読み込み数、物理読み込み数、書き込まれたページ数 ) を返すか決定する。<br>CS_TRUE の場合、統計が返る。<br>Client-Library は、これらの統計を情報サーバ・メッセージとして受け取る。アプリケーション・プログラムは、ユーザ指定サーバ・メッセージ・ハンドラによって、これらの統計にアクセスする。 | CS_FALSE |
| CS_OPT_STATS_TIME | 各クエリで、Adaptive Server Enterprise の解析、コンパイル、および実行の時間統計を返すか決定する。<br>CS_TRUE の場合、統計が返る。<br>Client-Library は、これらの統計を情報サーバ・メッセージとして受け取る。アプリケーション・プログラムは、ユーザ指定サーバ・メッセージ・ハンドラによって、これらの統計にアクセスする。 | CS_FALSE |

| 記号定数 | 意味 | デフォルト値 |
| --- | --- | --- |
| CS_OPT_STR_<br>RTRUNC | Adaptive Server Enterprise が文字データの右トランケートを ANSI 標準準拠で行うか決定する。<br>このオプションが CS_TRUE に設定されている場合、挿入オペレーションまたは更新オペレーションで char カラム値または varchar カラム値がトランケートされ、そのトランケートされた文字がすべてブランクではない場合に、Adaptive Server Enterprise はエラーを表示する。この動作は ANSI 標準に準拠している。<br>このオプションが CS_FALSE に設定されている場合、Adaptive Server Enterprise は、カラム定義に対して長すぎる char 値または varchar 値を、通知を発行しないで黙示的にトランケートする。 | CS_FALSE |
| CS_OPT_TEXTSIZE | Adaptive Server Enterprise のグローバル変数 @@*textsize* の値を指定する。この変数は、Adaptive Server Enterprise が返す text 値または image 値のサイズを制限する。<br>このオプションを設定するときは、Adaptive Server Enterprise が返す必要がある最も長い text 値または image 値の長さをバイト単位で指定するパラメータを使用すること。<br>Client-Library プロパティ CS_TEXTLIMIT も同様の効果がある。CS_TEXTLIMIT プロパティは、Client-Library がアプリケーションに返す text 値または image 値の最大値を設定する。CS_TEXTLIMIT プロパティは、サーバの処理には影響を与えない。したがって Client-Library は、text 値または image 値をネットワークから読み込むときにトランケートする。これに対して CS_OPT_TEXTSIZE オプションを設定すると、サーバは値をトランケートしてから送信するようになる。<br>アプリケーション・ユーザが特定のクエリを実行できるプログラムでは、ユーザは Transact-SQL の set textsize コマンドを使用して、CS_OPT_TEXTSIZE オプションを上書きできる。ユーザが上書きできないテキスト制限を設定するには、代わりに CS_TEXTLIMIT プロパティを使用すること。 | 32,768 バイト |

| 記号定数 | 意味 | デフォルト値 |
|---|---|---|
| CS_OPT_ TRUNCIGNORE | CS_TRUE の場合、Adaptive Server Enterprise は、トランケート・エラーを無視する。これは、ANSI 標準の動作である。<br>CS_FALSE の場合、変換でトランケートが起こると、Adaptive Server Enterprise はエラーを発生させる。 | CS_FALSE |

# パラメータ

## バッチ・パラメータ

ct_set_params() CT-Library ルーチンにより、コマンド自体を編集せずに複数のセットのコマンド・パラメータを送信できるようになりました。このルーチンを繰り返し使用してパラメータを転送します。この場合、前のコマンドの結果を処理したり、コマンド自体を再送する必要はありません。「ct_send_params」(654 ページ ) を参照してください。

## ct_setparam を使用した再バインド

複数のパラメータ・セットを送信する場合、アプリケーションが指す CT-Library はメモリ内の前のパラメータ・セットのロケーション以外でなければならない場合があります。パラメータを再バインドするには、ct_setparam() を使用して、別のデータ・ロケーションを指定します。既存の ct_setparam() 宣言を次に示します。

```
ct_setparam(cmd, datafmt, data, datalenp, indp)

  CS_COMMAND      *cmd;
  CS_DATAFMT      *datafmt;
  CS_VOID         *data;
  CS_INT          *datalenp;
  CS_SMALLINT     *indp;
```

ct_setparam() 呼び出しの *data*、*datalenp*、*indp* の各パラメータに新しい値を指定して、別のメモリ・ロケーションにバインドします。

ct_send_params() 呼び出しの後、該当するパラメータのフォーマットは変更できなくなります。したがって、ct_send_params() 呼び出しの後に実行される ct_setparam() 呼び出しで、*datafmt* に NULL 値を渡す必要があります。

再バインドすることができるパラメータは、最初に ct_setparam() でバインドされていたパラメータだけです。

# プロパティ

プロパティとは、隠し構造体 CS_CONTEXT、CS_CONNECTION、または CS_COMMAND に保管されている名前付きの値です。

プロパティは、Client-Library の動作を定義します。たとえば、接続構造体の CS_NETIO プロパティは、接続が同期か非同期かを設定し、コマンド構造体の CS_HIDDEN_KEYS プロパティは、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設定します。

## プロパティ、オプション、機能の比較

プロパティをサーバの「オプション (options)」や接続の「機能 (capabilities)」と混同しないようにしてください。サーバ・オプションは、クライアントに送信されたコマンドを実行している間のサーバの動作を制御します。接続の機能は、ある接続で送信できるクライアントの要求やサーバの応答のタイプを決定します。

一般に、プロパティは Client-Library の動作を制御しますが、サーバ・オプションはコマンドに対するサーバの応答を制御します。低いレベルでは、機能はクライアントとサーバが通信に使用するプロトコルに制約を加えます。

たとえば、アプリケーションが選択できる text または image データ型の値のサイズを制限するという問題があると仮定します。この問題を解決するには、プログラマは次のいずれかを行うアプリケーションを開発できます。

- CS_OPT_TEXTSIZE オプションを設定して、サーバがネットワーク上で送信する text 値または image 値を制限します ( 最適な解決方法です )。
- サーバから取得される CS_TEXT データ値または CS_IMAGE データ値を Client-Library がトランケートするように、CS_TEXTLIMIT 接続プロパティを設定します。トランケーションの前にネットワーク上で値全体を送信する必要があるので、効率的でない解決方法です。
- ct_capability を呼び出して、接続の CS_DATA_TEXT 応答機能と CS_DATA_IMAGE 応答機能を停止させてから、接続をオープンします。この場合サーバは text または image 値を送信できないので、おすすめできない解決方法です。

「オプション」(200 ページ ) および「機能」(67 ページ ) を参照してください。

## ログイン・プロパティ

「ログイン・プロパティ」は、サーバへのログイン時に使用される値を定義します。ログイン・プロパティには、CS_USERNAME、CS_PASSWORD、CS_PACKETSIZE が含まれます。

サーバは、ログイン・プロセス中に、いくつかのログイン・プロパティの値を変更します。たとえば、アプリケーションが CS_PACKETSIZE を 2,048 バイトに設定してからログインする場合、サーバは指定されたパケット・サイズを使用するか、それより小さいパケット・サイズまたは大きいパケット・サイズを選択できます。

## プロパティの設定と取得

アプリケーションは、ct_config、ct_con_props、ct_cmd_props を呼び出して、コンテキスト、接続、コマンドの構造体レベルで、それぞれ、Client-Library のプロパティの設定と取得を行います。アプリケーションは、cs_config を呼び出して、CS-Library のコンテキスト・プロパティの設定と取得を行います。

接続構造体は、割り付けられるときに、その親コンテキストのデフォルトのプロパティ値を取得します。たとえば、CS_TEXTLIMIT がコンテキスト・レベルで 16,000 に設定されている場合、このコンテキスト内で作成されたどんな接続も、16,000 のデフォルト・テキスト最大値を持ちます。同様に、コマンド構造体が割り付けられると、その親接続からデフォルト・プロパティ値を選択します。

アプリケーションは、cs_config、ct_config、ct_con_props、または ct_cmd_props を呼び出してそのプロパティ値を変更することにより、デフォルト・プロパティ値を上書きします。

ほとんどのプロパティ値は、アプリケーションによって設定および取得されますが、一部のプロパティは「取得のみ可能」です。

## 3 つのコンテキスト・プロパティ

コンテキスト・プロパティは 3 種類あります。

- CS-Library 固有のコンテキスト・プロパティ
- Client-Library 固有のコンテキスト・プロパティ
- Server-Library 固有のコンテキスト・プロパティ

cs_config は、CS-Library 固有のコンテキスト・プロパティの値を設定および取得します。CS_LOC_PROP を除く cs_config によって設定されるプロパティは、CS-Library にだけ反映されます。CS-Library 固有のコンテキスト・プロパティについては、『Open Client/Server Common Libraries リファレンス・マニュアル』の cs_config のリファレンス・ページに示されています。

ct_config は、Client-Library 固有のコンテキスト・プロパティの値を設定および取得します。ct_config によって設定されるプロパティは、Client-Library にだけ反映されます。Client-Library 固有のコンテキスト・プロパティについては、表 2-30 (213 ページ ) を参照してください。

srv_props は、Server-Library 固有のコンテキスト・プロパティの値を設定および取得します。srv_props によって設定されるプロパティは、Server-Library にだけ反映されます。

## プロパティがサポートされているかどうかのチェック

プロパティは常にサポートされているわけではありません。次のような場合には、プロパティがサポートされないことがあります。

- プロパティが外部のディレクトリ・プロバイダ・ソフトウェアと対応している場合。

  CS_DS プロパティの中には、( 接続の CS_DS_PROVIDER プロパティによって示される ) 外部のディレクトリ・プロバイダ・ソフトウェアの動作を制御するものもあります。Sybase のディレクトリ・ドライバは、Client-Library のプロパティを、対応するサービス・プロバイダの設定にマップします。ただし、相当する設定がプロバイダにない場合、このプロパティはサポートされません。アプリケーションは ct_con_props ルーチンを呼び出して、サポートされていないディレクトリ・プロパティの値の取得、設定、またはクリアを実行することはできません。

- プロパティが外部のセキュリティ・プロバイダ・ソフトウェアと対応している場合。

CS_SEC プロパティの中には、(CS_SEC_MECHANISM プロパティによって示される ) 外部のセキュリティ・ソフトウェアが実行する、データ暗号化などのセキュリティ・サービスを有効にするものもあります。Sybase のセキュリティ・ドライバは、Client-Library のプロパティを、対応するサービス・プロバイダの設定にマップします。ただし、セキュリティ・メカニズムがあらゆるサービスをサポートしていない場合もあります。アプリケーションでは、ct_con_props や ct_config を呼び出して、接続またはコンテキストに対して現在のセキュリティ・メカニズムがサポートしていないセキュリティ・サービスを有効にすることはできません。

アプリケーションは、*action* パラメータを CS_SUPPORTED に、buffer パラメータを CS_BOOL 変数のアドレスにして、ct_config ルーチンまたは ct_con_props ルーチンを呼び出して、プロパティがサポートされているかどうかをチェックします。

たとえば次に示すのは、CS_DS_SEARCH プロパティがサポートされているかどうかを調べるコードです。このサンプル・コードの CS_DS_SEARCH パラメータを変更して、別のプロパティのサポートについて調べることができます。

```
/* Is CS_DS_SEARCH supported?*/
 ret = ct_con_props(conn, CS_SUPPORTED,
                   CS_DS_SEARCH, &boolval,
                   CS_UNUSED, NULL);
if (ret != CS_SUCCEED)
   ... handle the error ...
 printf("CS_DS_SEARCH %s supported",
        boolval == CS_TRUE ?"is" :"is not");
```

---

**注意**　ディレクトリ・ドライバまたはセキュリティ・ドライバと対応するプロパティでのみ、*action* パラメータを CS_SUPPORTED に設定できます。

---

## ログイン・プロパティのコピー

ログイン・プロパティとは、サーバに接続するのに必要な値を指定する接続プロパティです。たとえば CS_USERNAME と CS_PASSWORD はログイン・プロパティです。

アプリケーションは、確立している接続から新しい接続構造体へログイン・プロパティをコピーします。このために、アプリケーションは次のことを行います。

1. 接続構造体を割り付けます (ct_con_alloc)。
2. 接続をカスタマイズします (ct_con_props)。
3. 接続をオープンします (ct_connect)。
4. ct_getloginfo を呼び出して、CS_LOGINFO 構造体を割り付け、接続のログイン・プロパティをコピーします。
5. 第 2 の接続構造体を割り付けます (ct_con_alloc)。
6. ct_setloginfo を呼び出して、CS_LOGINFO 構造体から第 2 の接続構造体へログイン・プロパティをコピーします。プロパティをコピーした後に、ct_setloginfo は CS_LOGINFO 構造体の割り付けを解除します。
7. 第 2 の接続でプロパティを変更する必要がある場合は、それをカスタマイズします (ct_con_props)。
8. 第 2 の接続をオープンします (ct_connect)。

## 外部からのプロパティの設定

Client-Library ルーチン ct_init と ct_connect はオプションで Open Client/Server ランタイム設定ファイルから、あるセクションを読み込み、コンテキストまたは接続に対するプロパティ値を設定します。この機能については、「ランタイム設定ファイルの使い方」(352 ページ )を参照してください。

## プロパティのクイック・リファレンス

表 2-30 に Client-Library プロパティの一覧を示します。この表に示されたコンテキスト・プロパティは、ct_config で設定されます。cs_config で設定されるコンテキスト・プロパティの一覧については、『Open Client/Server Common Libraries リファレンス・マニュアル』の cs_config のリファレンス・ページを参照してください。

**表 2-30：Client-Library プロパティ**

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_ANSI_BINDS | ANSI 形式のバインドを使用するかどうか。<br>「ANSI 形式のバインド」(236 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | |
| CS_APPNAME | コンテキスト・レベルでは、アプリケーション呼び出し自身の名前を指定する。接続レベルでは、サーバへのログイン時に使用されるアプリケーション名。<br>「アプリケーション名」(236 ページ ) を参照。 | 文字列。<br>デフォルトは NULL。 | 接続。<br>コンテキスト・レベルで設定するには、cs_config を呼び出す。 | ログイン・プロパティ。<br>接続レベルでは、接続確立後の設定はできない。 |
| CS_ASYNC_ NOTIFS | 接続がレジスタード・プロシージャ・ノーティフィケーションを非同期に受け取るかどうか。<br>「非同期ノーティフィケーション」(237 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | 接続。 | アイドル接続でノーティフィケーションを受け取るには CS_TRUE に設定する必要がある。 |
| CS_BULK_LOGIN | 接続がバルク・コピー・イン・オペレーションの実行が可能かどうか。<br>「バルク・コピー・オペレーション」(240 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | 接続。 | ログイン・プロパティ。<br>接続確立後の設定はできない。 |
| CS_CHARSETCNV | 文字セット変換が行われるかどうか。<br>「文字セット変換」(240 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは適用されない。 | 接続。 | 接続確立後は取得のみ可能である。 |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_COMMBLOCK | 通信セッション・ブロックへのポインタ。<br>このプロパティは、IBM-370 システムに固有のもので、他のすべてのプラットフォームでは無視される。<br>「通信セッション・ブロック」(240 ページ ) を参照。 | ポインタ値。<br>デフォルトは NULL。 | 接続。 | 接続確立後の設定はできない。 |
| CS_CONNECTED_ADDR | 現在接続が確立されているサーバのトランスポート・アドレス。 | 有効なトランスポート・アドレス。 | 接続。 | このプロパティは設定できません。サーバのアドレスが格納される CS_TRANADDR 構造体を指すポインタが必要。 |
| CS_CON_KEEPALIVE | KEEPALIVE オプションを使用するかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_TRUE。 | コンテキストまたは接続。 | 一部の Net-Library プロトコル・ドライバでは、このプロパティはサポートされていない。「プロパティがサポートされているかどうかのチェック」(210 ページ ) を参照。 |
| CS_CON_STATUS | 接続のステータス。<br>「接続ステータス」(240 ページ ) を参照。 | CS_INT 長のビットマスク。 | 接続。 | 取得のみ可能。 |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_CON_TCP_NODELAY | TCP_NODELAY プロパティを使用するかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_TRUE。 | コンテキストまたは接続。 | 一部の Net-Library プロトコル・ドライバでは、このプロパティはサポートされていない。「プロパティがサポートされているかどうかのチェック」(210 ページ) を参照。 |
| CS_CONFIG_BY_SERVERNAME | ct_connect が *server_name* パラメータまたは CS_APPNAME プロパティの値を外部設定データを読み込むセクション名として使用するかどうか。<br>「ランタイム設定ファイルの使い方」(352 ページ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | 接続。 | CS_EXTERNAL_CONFIG の設定により外部設定が可能な場合のみ有効である。<br>CS_VERSION_110 以降での初期化が必要。 |
| CS_CONFIG_FILE | Open Client/Server ランタイム設定ファイルの名前とロケーション。<br>「ランタイム設定ファイルの使い方」(352 ページ) を参照。 | 文字列。<br>デフォルトは NULL で、プラットフォーム固有のデフォルトを使用する。 | 接続。 | CS_EXTERNAL_CONFIG の設定により外部設定が可能な場合のみ有効である。<br>CS_VERSION_110 以降での初期化が必要。 |
| CS_CUR_ID | カーソルの識別番号。<br>「カーソル ID」(242 ページ) を参照。 | 整数値。<br>デフォルトは適用されない。 | コマンド | CS_CUR_STATUS が既存のカーソルを示した後は取得のみ可能。 |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_CUR_NAME | アプリケーションの ct_cursor(CS_CURSOR_DECLARE) 呼び出しで定義されたカーソルの名前。<br>「カーソル名」(242 ページ) を参照。 | 文字列。<br>デフォルトは適用されない。 | コマンド。 | ct_cursor (CS_CURSOR_DECLARE) が CS_SUCCEED を返した後は取得のみ可能。 |
| CS_CUR_ROWCOUNT | カーソル・ローの現在値。カーソル・ローは、内部フェッチ要求ごとに Client-Library へ返されるロー数。<br>「カーソル・ロー数」(242 ページ) を参照。 | 整数値。<br>デフォルトは適用されない。 | コマンド。 | CS_CUR_STATUS が既存のカーソルを示した後は取得のみ可能。 |
| CS_CUR_STATUS | カーソルのステータス。<br>「カーソル・ステータス」(243 ページ) を参照してください。 | CS_INT 長のビットマスク。 | コマンド。 | 取得のみ可能。 |
| CS_DIAG_TIMEOUT | インライン・エラー処理が有効であるときに、タイムアウト・エラーで Client-Library は失敗するべきか、それともリトライするべきか。<br>「診断タイムアウトの失敗」(245 ページ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。この値は、Client-Library がリトライしなければならないことを意味する。 | 接続。 | |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_DISABLE_POLL | ポーリングをしないようにするかどうか。ポーリングをしない場合、ct_poll は非同期オペレーション完了をレポートしない。<br>「ポーリングの抑制」(245 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。この値は、ポーリングを行うことを意味する。 | コンテキスト、接続。 | レイヤ構成の非同期アプリケーションで有効。 |
| CS_DS_COPY | アプリケーションの要求に応えるために、ディレクトリ・サービスがディレクトリ・エントリのキャッシュされたコピーを使用するかどうか。<br>「ディレクトリ・サービスによるキャッシュの使用」(128 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_TRUE で、この場合キャッシュが使用できる。 | 接続。 | すべてのディレクトリ・プロバイダがサポートするのではない。 |
| CS_DS_DITBASE | ディレクトリの検索が開始されるディレクトリ・ノードの完全に修飾された名前。<br>「ディレクトリ検索のベース」(128 ページ ) を参照。 | 文字列。<br>デフォルトはディクトリ・プロバイダ固有。 | 接続。 | すべてのディレクトリ・プロバイダがサポートするのではない。 |
| CS_DS_EXPAND ALIAS | ディレクトリ・サービスがディレクトリ・エイリアス・エントリを拡張するかどうか。<br>「ディレクトリ・サービスのエイリアスの拡張」(129 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_TRUE で、エイリアスの拡張が可能。 | 接続。 | すべてのディレクトリ・プロバイダがサポートするのではない。 |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_DS_FAILOVER | ディレクトリ・サービス・ドライバが初期化できない場合に、次の *libtcl.cfg* エントリまたは *interfaces* ファイルへのフェールオーバが可能かどうか。<br>「ディレクトリ・サービスのフェールオーバ」(130 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_TRUE。 | 接続。 | |
| CS_DS_PASSWORD | CS_DS_PRINCIPAL として指定されたディレクトリ・ユーザ ID と一緒に使用されるパスワード。<br>「ディレクトリ・サービスのパスワード」(131 ページ ) を参照。 | 文字列。<br>デフォルトは NULL。 | 接続。 | すべてのディレクトリ・プロバイダがサポートするのではない。<br>ユーザ認証のために LDAP サーバに渡されるユーザ名とパスワードは、Adaptive Server Enterprise へのアクセスに使用するユーザ名とパスワードとはまったく異なる。 |
| CS_DS_PRINCIPAL | CS_DS_PASSWORD として指定されたパスワードと対になる、ディレクトリ・サービスを使用するためのディレクトリ・ユーザ ID。<br>「ディレクトリ・サービスのプリンシパル名」(131 ページ ) を参照。 | 文字列。<br>デフォルトは NULL。 | 接続。 | すべてのディレクトリ・プロバイダがサポートするのではない。<br>ユーザ認証のために LDAP サーバに渡されるユーザ名とパスワードは、Adaptive Server Enterprise へのアクセスに使用するユーザ名とパスワードとはまったく異なる。 |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
| --- | --- | --- | --- | --- |
| CS_DS_PROVIDER | 接続のためのディレクトリ・プロバイダの名前。<br>「ディレクトリ・サービス・プロバイダ」(132 ページ ) を参照。 | 文字列。<br>デフォルトはディレクトリ・ドライバ設定によって異なる。 | 接続。 | |
| CS_DS_RAND_OFFSET | 接続リストのランダム・オフセットを有効または無効にする。<br>「ディレクトリ・サービスのランダム・オフセット」(132 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続 | ネットワーク・アドレス・リストがディレクトリ・サービスから取得されるときに決定される。 |
| CS_DS_SEARCH | ディレクトリ検索の深さを制限する。<br>「ディレクトリ・サービスの検索の深さ」(135 ページ ) を参照。 | CS_INT サイズの記号値。<br>有効値のリストについては、「ディレクトリ・サービスの検索の深さ」(135 ページ ) を参照。 | 接続。 | すべてのディレクトリ・プロバイダがサポートするのではない。 |
| CS_DS_SIZELIMIT | ct_ds_lookup を使用して開始された検索によって返されるディレクトリ・エントリの数を制限する。<br>「ディレクトリ検索サイズの制限」(135 ページ ) を参照。 | 0 以上の CS_INT 値。<br>デフォルトは 0 で、サイズの制限がないことを示します。 | 接続。 | すべてのディレクトリ・プロバイダがサポートするのではない。 |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_DS_TIMELIMIT | ct_ds_lookup を使用して開始されたディレクトリ検索の完了までの絶対時間の制限を秒単位で設定する。<br>「ディレクトリ検索の時間制限」(136 ページ ) を参照。 | 0 以上の CS_INT 値。<br>デフォルトは 0 であり、時間制限がないことを示す。 | 接続。 | すべてのディレクトリ・プロバイダがサポートするのではない。 |
| CS_EED_CMD | 拡張エラー・データを含むコマンド構造体へのポインタ。<br>「拡張エラー・データ・コマンド構造体」(246 ページ ) を参照。 | ポインタ値。<br>デフォルトは適用されない。 | 接続。 | 取得のみ可能。 |
| CS_ENDPOINT | 接続のファイル記述子。<br>「エンドポイントのポーリング」(246 ページ ) を参照。 | 整数値。<br>デフォルトは適用されない。<br>プラットフォームでは値は -1 で、エンドポイントの処理をサポートしていないことを意味する。 | 接続。 | 接続確立後は取得のみ可能である。 |
| CS_EXPOSE_FMTS | CS_ROWFMT_RESULT および CS_COMPUTEFMT_RESULT タイプの結果を公開するかどうか。<br>「公開フォーマット」(247 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | 接続確立後の設定はできない。 |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_EXTERNAL_CONFIG | ct_init または ct_connect が Open Client/Server ランタイム設定ファイルを読み込んで、オープンする接続のためにプロパティとオプションを設定するかどうか。<br>「ランタイム設定ファイルの使い方」(352 ページ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは同名の CS-Library コンテキスト・プロパティから継承する。 | コンテキスト、接続。 | CS_VERSION_110 以降での初期化が必要。 |
| CS_EXTRA_INF | SQLCA、SQLCODE、または SQLSTATE を使用して、Client-Library メッセージをインラインで処理するときに必要とされる追加情報を返すかどうか。<br>「追加情報」(248 ページ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | |
| CS_HAFAILOVER | 「高可用性フェールオーバ」(153 ページ) を参照。 | CS_TRUE または CS_FALSE。 | コンテキスト、接続。 | CS_VERSION_120 以降での初期化が必要。 |
| CS_HAVE_BINDS | 現在の結果セットに対して保存されている結果バインドがあるかどうか。<br>「バインド」(248 ページ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは適用されない。 | コマンド。 | 取得のみ可能。 |
| CS_HAVE_CMD | コマンド構造体に対して、再送可能なコマンドがあるかどうか。<br>「再送可能なコマンド」(248 ページ) を参照。 | CS_TRUE または CS_FALSE。 | コマンド。 | 取得のみ可能。 |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_HAVE_CUROPEN | コマンド構造体に対して、リストア可能なカーソル・オープン・コマンドがあるかどうか。<br>「リストア可能なカーソル・オープン・コマンド」(249 ページ ) を参照。 | CS_TRUE または CS_FALSE。 | コマンド。 | 取得のみ可能。 |
| CS_HIDDEN_KEYS | 隠しキーを公開するかどうか。<br>「隠しキー」(250 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続、コマンド。 | 結果が未処理またはカーソルがオープンしている場合は、コマンド・レベルでの設定はできない。 |
| CS_HOSTNAME | ホスト・マシン名。<br>「ホスト名」(251 ページ ) を参照。 | 文字列。<br>デフォルトは NULL。 | 接続。 | ログイン・プロパティ。<br>接続確立後の設定はできない。 |
| CS_IFILE | *interfaces* ファイルのパスと名前。<br>「interfaces ファイルのロケーション」(251 ページ ) を参照。 | 文字列。 | コンテキスト。 | |
| CS_LOC_PROP | ローカライゼーション情報を定義する CS_LOCALE 構造体。<br>「ロケール情報」(252 ページ ) を参照。 | CS_LOCALE 構造体。<br>接続は、親コンテキストからデフォルト・ローカライゼーション情報を選択する。 | 接続。<br>コンテキスト・レベルで CS_LOC_PROP を設定するには、cs_config を呼び出す。 | ログイン・プロパティ。<br>接続確立後の設定はできない。 |
| CS_LOGIN_STATUS | 接続がオープンしているかどうか。<br>「ログイン・ステータス」(252 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは適用されない。 | 接続。 | 取得のみ可能。 |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_LOGIN_TIMEOUT | ログイン・タイムアウト値。<br>「ログイン・タイムアウト」(253 ページ ) を参照。 | 整数値。<br>デフォルトは 60 秒。CS_NO_LIMIT 値は、無限タイムアウト時間を表す。 | コンテキスト、接続。 | |
| CS_LOOP_DELAY | サーバ名に対応するアドレスのシーケンスをリトライする前に ct_connect が待つ秒単位の遅延時間。<br>「ループ遅延」(253 ページ ) を参照。 | CS_INT >= 0。<br>デフォルトは 0。 | 接続。 | リトライ回数は CS_RETRY_COUNT により指定される。 |
| CS_MAX_CONNECT | このコンテキストでの最大接続数。<br>「最大接続数」(254 ページ ) を参照。 | 整数値。<br>デフォルトは 25。 | コンテキスト。 | |
| CS_MEM_POOL | 割り込みレベルで必要なメモリを確保するために Client-Library が使用するメモリ・プール。<br>「メモリ・プール」(254 ページ ) を参照。 | ポインタ値。<br>デフォルトは NULL ( ユーザ提供のメモリ・プールはなし )。 | コンテキスト。 | 非同期アプリケーションで役立つ。<br>コンテキストに接続があるときの設定またはクリアはできない。 |
| CS_NETIO | ネットワーク I/O が同期、完全非同期、または遅延非同期のいずれであるか。<br>「ネットワーク I/O」(256 ページ ) を参照。 | CS_SYNC_IO、CS_ASYNC_IO、または CS_DEFER_IO。<br>デフォルトは CS_SYNC_IO。 | コンテキスト、接続。 | オープン接続のコンテキストの設定はできない。<br>CS_DEFER_IO はコンテキスト・レベルでのみ有効。<br>CS_ASYNC_IO は Open Server ゲートウェイでは使用できない。 |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_NO_TRUNCATE | Client-Library が CS_MAX_MSG より長いメッセージをトランケートするべきか、連続させるべきか。<br>「トランケートの禁止」(257 ページ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。この値は、Client-Library が長いメッセージをトランケートすることを意味する。 | コンテキスト。 | |
| CS_NOAPI_CHK | アプリケーションが Client-Library ルーチンを呼び出すときに、Client-Library が引数とステータスのチェックを実行するかどうか。<br>「API チェックの禁止」(258 ページ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。これは Client-Library が API チェックを行うことを意味する。 | コンテキスト。 | |
| CS_NOCHARSETCN_REQD | サーバの文字セットがクライアントの文字セットと異なる場合、サーバで文字セット変換を実行するかどうか。<br>「文字変換は不要」(259 ページ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE で、必要に応じて変換が行われることを意味する。 | 接続。 | 接続確立後の設定はできない。 |
| CS_NOINTERRUPT | 特定のコールバック・イベントによるアプリケーションへの割り込みが可能かどうか。<br>「割り込みの禁止」(259 ページ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。この値は、アプリケーションへの割り込みが可能であることを意味する。 | コンテキスト。 | |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_NOTIF_CMD | レジスタード・プロシージャのノーティフィケーション・パラメータを含むコマンド構造体のポインタ。 | ポインタ値。<br>デフォルトは適用されない。 | 接続。 | 取得のみ可能。 |
| CS_PACKETSIZE | TDS パケット・サイズ ( バイト数 )。<br>「パケット・サイズ」(259 ページ ) を参照。 | デフォルトは 512 バイトです。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 ( たとえば、Adaptive Server Enterprise 15.0 など ) では、パケット・サイズを 512 バイトから 65,535 バイトの間で自由に設定できる。CS_NO_SRVPKTSIZE が設定されていない場合、パケット・サイズをここで指定した値より大きくすることはできない。 | 接続。 | ネゴシエートされたログイン・プロパティ。<br>接続確立後の設定はできない。 |
| CS_PARENT_HANDLE | コマンドまたは接続構造体の親構造体のアドレス。<br>「親構造体」(260 ページ ) を参照。 | ポインタ値。 | 接続、コマンド。 | 取得のみ可能。 |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_PARTIAL_TEXT | クライアント・アプリケーションで部分更新を実行するかどうかを示す。<br>「text データと image データの部分更新」(260 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続 | このプロパティは、サーバへの接続が確立される前に設定する必要がある。サーバが部分更新をサポートしていない場合は、このプロパティは CS_FALSE に再設定される。 |
| CS_PASSWORD | サーバへのログインに使用するパスワード。<br>「パスワード」(260 ページ ) を参照。 | 文字列。<br>デフォルトは NULL。 | 接続。 | ログイン・プロパティ。 |
| CS_PROP_APPLICATION_SPID | Adaptive Server Enterprise SPID は、ログイン時に保存され、プロパティとして使用できる。<br>「拡張フェールオーバ」(246 ページ ) を参照。 | サーバのサーバ・プロセス ID (spid) に対応する CS_INT 値。 | 接続。 | ログイン・プロパティ。 |
| CS_PROP_EXTENDEDFAILOVER | サーバが指定したフェールオーバ・ターゲットを有効または無効にする。<br>「拡張フェールオーバ」(246 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_TRUE。 | コンテキスト、接続。 | ログイン・プロパティ。 |
| CS_PROP_MIGRATABLE | 接続マイグレーションを有効または無効にする。<br>「接続マイグレーション」(241 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_TRUE。 | コンテキスト、接続。 | ログイン・プロパティ。 |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_PROP_REDIRECT | ログイン・リダイレクト・サポートを有効または無効にする。<br>「ログインのリダイレクト」(254 ページ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_TRUE。 | コンテキスト、接続。 | ログイン・プロパティ。 |
| CS_PROP_SSL_PROTOVERSION | サポートされている SSL/TLS プロトコルのバージョン。 | CS_INT | コンテキスト、接続。 | 次のいずれかの値を指定する。<br>CS_SSLVER_20<br>CS_SSLVER_30<br>CS_SSLVER_TLS1 |
| CS_PROP_SSL_CIPHER | CipherSuite 名をカンマで区切ったリスト。 | CS_CHAR | コンテキスト、接続。 | |
| CS_PROP_SSL_LOCALID | ローカル ID ( 証明書 ) ファイルのパスの指定に使用するプロパティ。 | 文字列 | コンテキスト、接続。 | 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る構造体。 |
| CS_PROP_SSL_CA | 信頼された CA 証明書を含むファイルへのパスを指定する。 | CS_CHAR | コンテキスト、接続。 | |
| CS_RETRY_COUNT | サーバのアドレスへの接続をリトライする回数。<br>「リトライ回数」(264 ページ) を参照。 | CS_INT >= 0。<br>デフォルトは 0 です。 | 接続。 | ログイン・ダイアログの確立だけに影響する。ログインの失敗はリトライされない。 |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_RPCPARAM_ LOB | ラージ・オブジェクト (LOB) データ型をストアド・プロシージャの入力パラメータとして使用できるかどうかを判断する。 「ストアド・プロシージャ・パラメータとしての ラージ・オブジェクト」(167 ページ ) を参照。 | CS_TRUE または CS_FALSE。 デフォルトは CS_FALSE。 | 接続。 | 接続確立後の設定はできない。 |
| CS_RPCPARAM_ NOLOB | パラメータとして LOB データを送信しないようにサーバに要求します。 「ストアド・プロシージャ・パラメータとしての ラージ・オブジェクト」(167 ページ ) を参照。 | CS_TRUE または CS_FALSE。 デフォルトは CS_TRUE。 | 接続。 | 接続確立後の設定はできない。 |
| CS_SEC_ APPDEFINED | アプリケーションが定義しているチャレンジ／応答セキュリティ・ハンドシェイクを接続が使用するかどうか。 「セキュリティ・ハンドシェイク：チャレンジ／応答」(316 ページ ) を参照。 | CS_TRUE または CS_FALSE。 デフォルトは CS_FALSE。 | 接続。 | 接続確立後の設定はできない。 |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
| --- | --- | --- | --- | --- |
| CS_SEC_CHALLENGE | Sybase が定義しているチャレンジ／応答セキュリティ・ハンドシェイクを接続が使用するかどうか。<br>「セキュリティ・ハンドシェイク：チャレンジ／応答」(316 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | 接続。 | 接続確立後の設定はできない。 |
| CS_SEC_CHANBIND | 接続のセキュリティ・メカニズムがチャンネルのバインドを行うかどうか。<br>「ログイン認証サービスの要求」(295 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_CONFIDENTIALITY | 接続でデータの暗号化サービスを実行するかどうか。<br>「パケットごとのセキュリティ・サービスの要求」(301 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_CREDENTIALS | 委任されたユーザのクレデンシャルを転送するためにゲートウェイ・アプリケーションで使用される。<br>「ログイン認証サービスの要求」(295 ページ ) を参照。 | A CS_VOID * ポインタ。 | コンテキスト、接続。 | 読み込み不可能。<br>接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_SEC_CREDTIMEOUT | ユーザのクレデンシャルが期限切れかどうか。<br>「ログイン認証サービスの要求」(295 ページ) を参照。 | CS_INT。有効な値とその意味については、表 2-33 (297 ページ) を参照。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_DATAORIGIN | 接続のセキュリティ・メカニズムがデータ・オリジンの検証をするかどうか。<br>「パケットごとのセキュリティ・サービスの要求」(301 ページ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_DELEGATION | ユーザの委任クレデンシャルを使用してサーバを他のサーバに接続させるかどうか。<br>「ログイン認証サービスの要求」(295 ページ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_DETECTREPLAY | 接続のセキュリティ・メカニズムがリプレイされた転送を検出するかどうか。<br>「パケットごとのセキュリティ・サービスの要求」(301 ページ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_SEC_DETECTSEQ | 接続のセキュリティ・メカニズムが不正なシーケンスの転送を検出`するかどうか。<br>「パケットごとのセキュリティ・サービスの要求」(301 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_ENCRYPTION | 暗号化したパスワード・セキュリティ・ハンドシェイクを接続が使用するかどうか。<br>「セキュリティ・ハンドシェイク：暗号化したパスワード」(317 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | 接続。 | 接続確立後の設定はできない。 |
| CS_SEC_INTEGRITY | 接続のセキュリティ・メカニズムがデータ整合性のチェックを実行するかどうか。<br>「パケットごとのセキュリティ・サービスの要求」(301 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_KEYTAB | 接続のセキュリティ・メカニズムが CS_USERNAME プロパティと一緒に使用されるセキュリティ・キーを読み込むファイルのパスと名前。<br>「ログイン認証サービスの要求」(295 ページ ) を参照。 | 文字列。<br>デフォルトは NULL。これは、アプリケーションが ct_connect を呼び出す前に、ユーザがクレデンシャルを確立しておく必要があることを意味する。 | 接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_SEC_MECHANISM | 接続のためのセキュリティ・サービスを実行する、ネットワーク・セキュリティ・メカニズムの名前。<br>「ネットワーク・セキュリティ・メカニズムの選択」(293 ページ ) を参照。 | 文字列値<br>デフォルトはセキュリティ・ドライバ設定によって異なる。 | コンテキスト、接続。 | 接続確立後の設定はできない。 |
| CS_SEC_MUTUALAUTH | サーバが、クライアントに対してサーバ自体を認証する必要があるかどうか。<br>「ログイン認証サービスの要求」(295 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_NEGOTIATE | trusted ユーザ・セキュリティ・ハンドシェイクを接続が使用するかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | 接続。 | 接続確立後の設定はできない。 |
| CS_SEC_NETWORKAUTH | 接続のセキュリティ・メカニズムがネットワークベースのユーザの認証を実行するかどうか。<br>「ログイン認証サービスの要求」(295 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムと、CS_USERNAME と一致する確立済みのクレデンシャルが必要。 |
| CS_SEC_SERVERPRINCIPAL | 接続がオープンされるサーバのネットワーク・セキュリティ・プリンシパルの名前。<br>「ログイン認証サービスの要求」(295 ページ ) を参照。 | 文字列値<br>デフォルトは NULL。このとき ct_connect は、サーバのプリンシパル名がその *server_name* パラメータと同じであると見なす。 | 接続。 | 接続確立後の設定はできない。<br>ネットワークベースのユーザの認証を使用する接続でのみ意味がある。 |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_SEC_SESSTIMEOUT | 接続のセキュリティ・セッションの期限が切れているかどうか。<br>「ログイン認証サービスの要求」(295 ページ ) を参照。 | CS_INT。有効な値とその意味については、表 2-33 (297 ページ ) を参照。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SENDDATA_NOCMD | ct_connect の呼び出し時に sp_mda プロシージャがサーバで実行されるかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | 接続。 | CS_SENDDATA_NOCMD は、ct_connect を呼び出す前に設定する必要がある。SQL コマンドを使用せずに text データまたは image データのみを送信する ct_send_data ルーチンが、サーバでサポートされない場合、このプロパティは再設定される。 |
| CS_SERVERADDR | 接続先のサーバのアドレス。 | 「hostname portnumber [filter]」の形式。filter はオプション。 | 接続。 | このプロパティを使用すると、ctlib はサーバのホスト名とインタフェースのポート番号をバイパスする。 |
| CS_SERVERNAME | 接続先のサーバの名前。<br>「サーバ名」(265 ページ ) を参照。 | 文字列値。<br>デフォルトは適用されない。 | 接続。 | 接続確立後は取得のみ可能である。 |
| CS_STICKY_BINDS | サーバ・コマンドが繰り返し実行されたときに、結果項目とプログラム変数間のバインドが継続するかどうか。<br>「継続結果バインド」(261 ページ ) を参照。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コマンド。 | |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_TDS_VERSION | 接続が使用している TDS プロトコルのバージョン。<br>「TDS バージョン」(266 ページ ) を参照。 | バージョン・レベル記号。<br>CS_VERSION を基にした値がデフォルト。 | 接続。 | ネゴシエートされたログイン・プロパティ。<br>接続確立後の設定はできない。 |
| CS_TEXTLIMIT | この接続で返される最大の text または image 値。<br>「text および image の最大値」(267 ページ ) を参照。 | 整数値。<br>デフォルトは CS_NO_LIMIT。 | コンテキスト、接続。 | |
| CS_TIMEOUT | サーバからの結果の読み込みに対するタイムアウト値。<br>「タイムアウト」(267 ページ ) を参照。 | 整数値。<br>デフォルトは CS_NO_LIMIT。 | コンテキスト、接続。 | |
| CS_TRANSACTION_NAME | Open Server for CICS への接続に使用されるトランザクション名。<br>「トランザクション名」(271 ページ ) を参照。 | 文字列値<br>デフォルトは NULL。 | 接続。 | |
| CS_USER_ALLOC | ユーザ定義のメモリ割り付けルーチン。<br>「ユーザ割り付け関数」(271 ページ ) を参照。 | ユーザ定義の関数を指すポインタ。<br>デフォルトは適用されない。 | コンテキスト。 | 非同期アプリケーションで役立つ。 |
| CS_USER_FREE | ユーザ定義のメモリ解放ルーチン。<br>「ユーザ解放関数」(272 ページ ) を参照。 | ユーザ定義の関数を指すポインタ。<br>デフォルトは適用されない。 | コンテキスト。 | 非同期アプリケーションで役立つ。 |

| プロパティ名 | 意味 | 指定できる値 | 適用可能なレベル | 注意 |
|---|---|---|---|---|
| CS_USERDATA | ユーザ割り付けデータ。<br>「ユーザ・データ」(273 ページ ) を参照。 | ユーザ割り付けデータ。 | 接続、コマンド。<br>コンテキスト・レベルで CS_USERDATA を設定するには、cs_config を呼び出す。 | |
| CS_USERNAME | サーバへのログインに使用する名前。<br>「ユーザ名」(275 ページ ) を参照。 | 文字列。<br>デフォルトは NULL。 | 接続。 | ログイン・プロパティ。<br>接続確立後の設定はできない。 |
| CS_VER_STRING | Client-Library の正確なバージョン文字列。<br>「Client-Library のバージョン文字列」(275 ページ ) を参照。 | 文字列。<br>デフォルトは適用されない。 | コンテキスト。 | 取得のみ可能。 |
| CS_VERSION | このコンテキストで使用している Client-Library のバージョン。<br>「Client-Library のバージョン」(275 ページ ) を参照。 | バージョン・レベル記号。<br>CS_VERSION は、コンテキストの ct_init 呼び出しからその値を取得する。<br>使用可能な値の説明を参照。 | コンテキスト。 | 取得のみ可能。 |

## プロパティについて

この項では、Client-Library の各プロパティについて詳しく説明します。

## ANSI 形式のバインド

CS_ANSI_BINDS は、Client-Library が ANSI 形式のバインドと ANSI 形式のカーソル・エンドデータ処理を使用するかどうかを設定します。

ANSI 形式のバインドが有効であると、ct_fetch は次のような状態でエラーになります。

- 結果セットにある項目のすべてではなく、一部だけをバインドすることは、エラーとみなされます。アプリケーションは、項目をまったくバインドしないか、すべてをバインドするかのいずれかでなければなりません。
- ct_fetch は、NULL またはトランケートされた文字列値を、インジケータが関連付けられていない変数へコピーすると、エラーを生成します。

上記のいずれの場合でも、ct_fetch は CS_ROW_FAIL を返します。

ANSI 形式のカーソル・エンドデータ処理が有効になっている場合、カーソル結果が処理されているときに ct_fetch ルーチンはエラーを表示しません。ct_fetch ルーチンは CS_END_DATA を返していて、次のルーチンを呼び出します。

- ct_bind
- ct_fetch

CS_ANSI_BINDS プロパティが CS_TRUE でない場合、この状態で上記のルーチンを呼び出すと、ct_fetch ルーチンはエラーを表示して失敗します。

## アプリケーション名

CS_APPNAME は、次のように使用され、アプリケーション名を指定します。

- コンテキスト・レベルでは、CS_APPNAME プロパティは、ct_init ルーチンがデフォルトの Client-Library コンテキスト・プロパティを読み込んでくる設定ファイル・セクションを指定します。この機能の詳細については、「ランタイム設定ファイルの使い方」(352 ページ ) を参照してください。CS-Library ルーチン cs_config を呼び出すことによって、コンテキスト・レベルで CS_APPNAME プロパティを設定できます。

- 接続レベルでは、CS_APPNAME プロパティは、サーバに接続するときに、その接続が使用するアプリケーション名を定義します。接続に対して外部設定が有効になっている場合、CS_APPNAME プロパティは、ct_connect ルーチンがその接続に対するデフォルトのプロパティ、サーバ・オプション、機能を読み込んでくる設定ファイルのセクションも識別することがあります。この機能の詳細については、「ランタイム設定ファイルの使い方」(352 ページ )を参照してください。

Adaptive Server Enterprise は、アプリケーション名を指定して、*master* データベースの *sysprocesses* テーブルでの接続プロセスを識別します。

接続構造体は、割り付けられるときに親コンテキスト構造体から CS_APPNAME 設定を継承します。継承した値が変更されない場合は、接続がオープンされるときにその値がアプリケーション名になります。アプリケーションが ct_con_props を呼び出して、接続のアプリケーション名を変更した後に、接続がオープンします。

## 非同期ノーティフィケーション

CS_ASYNC_NOTIFS 接続プロパティは、Client-Library アプリケーションが Open Server アプリケーションからレジスタード・プロシージャ・ノーティフィケーションを受け取る方法を制御します。CS_ASYNC_NOTIFS は、接続がレジスタード・プロシージャ・ノーティフィケーションを非同期で受け取るかどうかを決定します。

Open Server アプリケーションは、ノーティフィケーション ( 通知 ) を 1 つまたは複数の TDS パケットとしてクライアントに送信します。Client-Library が接続からノーティフィケーション・パケットを読み、アプリケーションのノーティフィケーション・コールバックを起動すると、クライアント・アプリケーションにノーティフィケーションが通知されます。

レジスタード・プロシージャ・ノーティフィケーションによって、クライアントは Open Server 上での 1 つまたは複数のレジスタード・プロシージャの実行を監視できます。監視されているプロシージャを任意のクライアントが実行する場合、Open Server は、その特定のレジスタード・プロシージャを監視している各クライアントに、ノーティフィケーションを送信します。

サーバは、1 つまたは複数の Tabular Data Stream パケットとしてノーティフィケーションを送信します。アプリケーションがノーティフィケーションを知ることができるようにするため、Client-Library ではこれらのパケットを読み込んで、アプリケーションのノーティフィケーション・コールバックをトリガします。CS_ASYNC_NOTIFS プロパティは、アプリケーションがノーティフィケーションをどのようにして知るかを決定します。

- 同期接続は、CS_ASYNC_NOTIFS プロパティを CS_TRUE に設定して非同期ノーティフィケーションを受け取ります。
- 非同期接続が CS_ASYNC_NOTIFS を CS_TRUE に設定しない場合、非同期接続はノーティフィケーションを非同期に受け取りません。
- ノーティフィケーションの受信だけに使用される接続では、CS_ASYNC_NOTIFS プロパティが CS_TRUE に設定されていない場合、ct_poll ルーチンはノーティフィケーションを検索しません。

### CS_ASYNC_NOTIFS が CS_TRUE の場合

CS_ASYNC_NOTIFS が CS_TRUE に設定されている場合、Client-Library は、アプリケーションを中断してレジスタード・プロシージャ・ノーティフィケーションの到着を報告します。

割り込み駆動型 I/O またはスレッド駆動型 I/O をサポートしているプラットフォームでは、Client-Library が自動的にノーティフィケーション情報を読み込んで、ノーティフィケーションが接続に到着したときに接続のノーティフィケーション・コールバックを呼び出します。

その他のプラットフォームでは、他の方法で接続をアクティブにできない場合、ct_poll で接続をポーリングして、ノーティフィケーション・コールバックをトリガします。アイドルな接続で ct_poll ルーチンがノーティフィケーション・コールバックにトリガするようにするには、CS_ASYNC_NOTIFS プロパティを CS_TRUE に設定してください。

### CS_ASYNC_NOTIFS が CS_FALSE の場合

CS_ASYNC_NOTIFS が CS_FALSE に設定されている場合 ( デフォルト )、Client-Library 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報告するために、アプリケーションはネットワークから読み込んでいる必要があります。サーバがノーティフィケーションを送信するとき、Client-Library はノーティフィケーション・データを読み込み、次にサーバと対話するときにアプリケーションのノーティフィケーション・コールバックをトリガします。

同様に、CS_ASYNC_NOTIFS が CS_FALSE の場合、ct_poll はネットワークからノーティフィケーション・データを読み込まず、アプリケーションのノーティフィケーション・コールバックをトリガしません。これは、アプリケーションが ct_poll の結果を読み込んで、レジスタード・プロシージャ・ノーティフィケーションを通知する必要があることを意味しています。ct_poll がノーティフィケーションをレポートすると、アプリケーションのノーティフィケーション・コールバックが自動的に呼び出されます。

---

**注意**　接続がレジスタード・プロシージャ・ノーティフィケーションの受信だけに使用される場合は、CS_ASYNC_NOTIFS プロパティを CS_TRUE に設定して、ノーティフィケーションを受信してください。ct_poll ルーチンを呼び出して接続がポーリングされている場合でも、非同期ノーティフィケーションを有効にしてください。

---

### CS_ASYNC_NOTIFS の設定

次のサンプル・プログラムは、非同期ノーティフィケーションを使用可能にします。

```
/* Turn on read-ahead notifications.*/
 boolval = CS_TRUE;
 if (ct_con_props(conn, CS_SET, CS_ASYNC_NOTIFS, &boolval,
                  CS_UNUSED, (CS_INT *)NULL)!CS_SUCCEED)
 {
   fprintf(stderr,
      "Error:ct_con_props(SET, CS_ASYNC_NOTIFS) failed¥n");
   (CS_VOID)ct_close(conn, CS_UNUSED);
   (CS_VOID)ct_con_drop(conn);
 }
```

CS_ASYNC_NOTIFS を CS_FALSE に設定しても、非同期ノーティフィケーションをすぐには停止しません。非同期ノーティフィケーションを停止するために、アプリケーションは、CS_ASYNC_NOTIFS を CS_FALSE に設定した後に、サーバへコマンドを送る必要があります。

CS_ASYNC_NOTIFS は、ノーティフィケーションが非同期に受け取られるかどうかを決定する唯一のプロパティです。

- 一方の同期接続は、非同期ノーティフィケーションを受け取ります。
- 非同期接続が CS_ASYNC_NOTIFS を CS_TRUE に設定しない場合、非同期接続はノーティフィケーションを非同期に受け取りません。

レジスタード・プロシージャ・ノーティフィケーションについては、「レジスタード・プロシージャ」(276 ページ ) を参照してください。

## バルク・コピー・オペレーション

CS_BULK_LOGIN は、接続がデータベースへのバルク・コピー・オペレーションを実行できるかどうかを記述します。デフォルトの CS_FALSE はバルク・コピー・オペレーションを禁止します。

接続上でバルク・コピー・オペレーションを実行するアプリケーションは、CS_BULK_LOGIN 接続プロパティを CS_TRUE に設定してから、ct_connect を呼び出して接続をオープンする必要があります。

ユーザによるアドホック・クエリの実行が許可されるアプリケーションでは、このプロパティを CS_TRUE に設定しないようにして、ユーザが SQL コマンドでバルク・コピー・シーケンスを開始できないようにする場合もあります。いったんバルク・コピー・シーケンスを起動すると、通常の SQL コマンドでは停止することはできません。アプリケーションは、Bulk-Library 呼び出しを使用してバルク・コピー・オペレーションを実行します。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。

## 文字セット変換

CS_CHARSETCNV は、サーバがクライアント文字セットとサーバ文字セットとの間の変換を行っているかどうかを記述します。このプロパティは、接続確立後は取得のみ可能です。

CS_TRUE の値は、サーバがクライアントの文字セットとサーバの文字セット間の変換をしていることを示します。一方、CS_FALSE は、変換がされていないことを示します。

## 通信セッション・ブロック

CS_COMMBLOCK プロパティは、通信ブロックへのポインタを定義します。このプロパティは、IBM370 システムに固有のもので、他のすべてのプラットフォームでは無視されます。

## 接続ステータス

CS_CON_STATUS は、接続の現在のステータスを示す CS_INT 長のビットマスクです。

CS_CON_STATUS を構成する記号値の一覧表を次に示します。

| 記号値 | 意味 |
|---|---|
| CS_CONSTAT_CONNECTED | 接続がオープンしている。 |
| CS_CONSTAT_DEAD | 接続が「dead」とマーク付けされている。 |

Client-Library は、エラーによって接続が使用不可能となった場合、または、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・ルーチンが CS_FAIL を返した場合、接続に dead とマーク付けします。アプリケーションは、ct_close および ct_con_drop を呼び出して、「dead」( 使用不可能 ) とマーク付けされた接続をクローズして切断する必要があります。例外は、あるタイプの結果処理エラーに対して発生します。結果の処理中に接続が dead とマーク付けされた場合、アプリケーションは、*type* に CS_CANCEL_ALL または CS_CANCEL_ATTN を指定して ct_cancel を呼び出すことによって、接続の再開を試みることができます。再開に失敗した場合、アプリケーションは、接続をクローズして切断しなければなりません。

## サーバ名での設定

CS_CONFIG_BY_SERVERNAME プロパティは、ct_connect ルーチンが外部設定データを読み込むセクション名として、*server_name* パラメータと CS_APPNAME プロパティの値のどちらを使用するかを決定します。この機能の詳細については、「ランタイム設定ファイルの使い方」(352 ページ ) を参照してください。

## 設定ファイル名

CS_CONFIG_FILE プロパティは、プロパティ、サーバ・オプション、機能のデフォルト値を設定するために Client-Library が読み込む Open Client/Server ランタイム設定ファイルの名前とロケーションを指定します。この機能の詳細については、「ランタイム設定ファイルの使い方」(352 ページ ) を参照してください。

## 接続マイグレーション

CS_PROP_MIGRATABLE が CS_TRUE ( デフォルト ) の場合、接続マイグレーション・プロトコルを認識できるサーバで接続をマイグレートできます。また、ログインが完了した後に、クライアント接続を別のサーバに移動できます。

CS_PROP_MIGRATABLE プロパティは、ct_config と ct_con_props を使用して設定できます。

## カーソル ID

CS_CUR_ID は、カーソルに対して割り付けられたサーバ識別番号です。

アプリケーションは、ct_cmd_props(CS_CUR_STATUS) を呼び出して、カーソルが目的のコマンド領域にあることを確認してから、カーソルの識別番号を取得します。

CS_CUR_ID はコマンド構造体プロパティであり、接続またはコンテキスト・レベルでは取得できません。

カーソル・プロパティは、カーソル情報をクライアントへ送るゲートウェイ・アプリケーションで有効です。

カーソル ID を取得する例については、「カーソル・ステータスの例」(320 ページ ) を参照してください。

## カーソル名

CS_CUR_NAME は、カーソルが宣言されたときの名前です。アプリケーションは、ct_cursor(CS_CURSOR_DECLARE) を呼び出すことによって、カーソルを宣言します。

アプリケーションは、ct_cursor(CS_CURSOR_DECLARE) の呼び出しが CS_SUCCEED を返した後であればいつでも、カーソル名を取得できます。

CS_CUR_NAME は、コマンド構造体プロパティであり、接続またはコンテキスト・レベルでは取得できません。

カーソル・プロパティは、カーソル情報をクライアントへ送るゲートウェイ・アプリケーションで有効です。

カーソル名を取得する例については、「カーソル・ステータスの例」(320 ページ ) を参照してください。

## カーソル・ロー数

CS_CUR_ROWCOUNT は、カーソルのカーソル・ローの現在値です。

カーソル・ローは、内部フェッチ要求ごとに Client-Library へ返されるロー数です。これは、一度の ct_fetch 呼び出しでアプリケーションに返されるローの数ではありません。後ろにある方の数字は、コマンド構造体の所定の位置でバインドによって指定されたものです。詳細については、「配列バインド」(383 ページ ) を参照してください。

カーソル・ローは、デフォルトで 1 に設定されます。このデフォルト値は、アプリケーションが ct_fetch ルーチンを呼び出すたびに、Client-Library が ct_fetch 呼び出しによって要求されるローごとに、内部カーソル・フェッチ・コマンドを 1 つずつ発行することを意味します。

それぞれの内部カーソル・フェッチ・コマンドは、クライアントとサーバの間の対話を要求します。したがってカーソル・ローの値を大きくすると、そのカーソルからフェッチするのに必要なネットワーク上のデータ送信の回数は減少します。ただし、アプリケーションがネストされたカーソル・コマンドを送信する場合や、カーソルからフェッチしている間に別のコマンド構造体にコマンドを送信する場合に、Client-Library が新たなコマンドを送信するには、ct_fetch によってフェッチされていないローをバッファする必要があります。したがってカーソル・ローの値を大きくすると、Client-Library によるメモリ使用率も大きくなることがあります。

アプリケーションは、カーソルがオープンされる前に ct_cursor ルーチンを呼び出してカーソル・ローの値を大きくすることができます。詳細については、「カーソル・ロー・コマンド」(484 ページ ) を参照してください。

アプリケーションは、ct_cmd_props(CS_CUR_STATUS) を呼び出して、目的のコマンド領域にカーソルがあることを確認した後で、CS_CUR_ROWCOUNT を取得します。

CS_CUR_ROWCOUNT はコマンド構造体プロパティであり、コンテキスト・レベルでは取得できません。

カーソル・プロパティは、カーソル情報をクライアントへ送るゲートウェイ・アプリケーションで有効です。

## カーソル・ステータス

CS_CUR_STATUS は、カーソルの現在のステータスを表す CS_INT 長の値です。

ステータスは、そのコマンド構造体にはカーソルがないことを示す CS_CURSTAT_NONE か、ステータスを示すために設定されるビットでのステータス値のどちらかになります。

CS_CURSTATUS プロパティが CS_CURSTAT_NONE ではない場合、アプリケーションは次の表に示されているビットマスク値を適用して、カーソル・ステータスを決定します。たとえばカーソルが更新可能である場合は、次のようなテストを行ってください。

```
if ((cur_status & CS_CURSTAT_UPDATABLE)
    == CS_CURSTAT_UPDATABLE)
```

表 2-31 に、CS_CUR_STATUS 値をテストするための記号のビットマスク値の一覧を示します。

**表 2-31：カーソル・ステータスのビットマスク値**

| ビットマスク値 | テスト対象 |
|---|---|
| CS_CURSTAT_CLOSED | コマンド領域にクローズしたカーソルがある。アプリケーションは、クローズしたカーソルをオープンしたり割り付けを解除できる。 |
| CS_CURSTAT_DECLARED | 現在、このコマンド領域でカーソルが宣言されている。アプリケーションは、宣言されたカーソルをオープンしたり割り付けを解除できる。 |
| CS_CURSTAT_ROWCOUNT | アプリケーションは、カーソル・ロー・コマンドをサーバへ送信したが、カーソルがオープンされない。 |
| CS_CURSTAT_OPEN | このコマンド領域にオープンしたカーソルがある。アプリケーションは、オープンしたカーソルをクローズできる。 |
| CS_CURSTAT_RDONLY | カーソルは読み込み専用で、更新の実行には使用できない。 |
| CS_CURSTAT_UPDATABLE | カーソルは更新の実行に使用できる。 |
| CS_SCROLL_INSENSITIVE | スクロール可能で非反映型のカーソルを宣言する。 |
| CS_SCROLL_SEMISENSITIVE | スクロール可能で半反映型のカーソルを宣言する。 |
| CS_SCROLL_CURSOR | 非反映型のスクロール可能カーソルを宣言する (デフォルト)。 |
| CS_NOSCROLL_INSENSITIVE | 非反映型で非スクロール可能なカーソルを宣言する。 |

サーバがカーソル・ステータスをレポートします。アプリケーションは CS_CUR_STATUS プロパティ値への変更を知る前に、ct_cursor コマンドを送信して結果の処理を開始する必要があります。以下の場合、カーソル・ステータスが正しいことが保証されます。

- ct_results が、*result_type* パラメータに CS_CMD_SUCCEED、CS_CMD_FAIL、または CS_CURSOR_RESULT を設定し、CS_SUCCEED を返した後
- ct_cancel(CS_CANCEL_ALL) が CS_SUCCEED を返した後
- Client-Library または CS-Library のいずれかのルーチンが CS_CANCELED を返した後

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う影響したのかを判断します。

CS_CUR_STATUS はコマンド構造体プロパティであり、接続またはコンテキスト・レベルでは取得できません。

カーソル・プロパティは、カーソル情報をクライアントへ送るゲートウェイ・アプリケーションで有効です。

このプロパティを取得してカーソル・ステータスをチェックするフラグメント例については、「カーソル・ステータスの例」(320 ページ ) を参照してください。

## 診断タイムアウトの失敗

インライン・エラー処理が有効な場合、CS_DIAG_TIMEOUT プロパティによって、Client-Library のタイムアウト・エラーが発生したときに Client-Library が実行不可能になるかリトライできるかを設定します。

CS_DIAG_TIMEOUT が CS_TRUE である場合、Client-Library ルーチンがタイムアウト・エラーを生成したときに、Client-Library は接続を「dead」とマーク付けします。

CS_DIAG_TIMEOUT が CS_FALSE である場合、Client-Library ルーチンがタイムアウト・エラーを生成したときに、Client-Library は何度でもリトライします。

## ポーリングの抑制

CS_DISABLE_POLL プロパティでは、ct_poll が非同期オペレーションの完了をレポートするかどうかを設定します。

レイヤ構成の非同期アプリケーションは、CS_DISABLE_POLL を使用して、ct_poll が低レベルの非同期オペレーションの完了をレポートしないようにします。

CS_DISABLE_POLL プロパティが CS_TRUE に設定されている場合、アプリケーションは ct_wakeup を呼び出すことができません。

「レイヤ構成のアプリケーション」(19 ページ ) を参照してください。

## ディレクトリ・サービスのプロパティ

ディレクトリ・サービスに関連するプロパティの詳細については、「ディレクトリ・サービスのプロパティ」(127 ページ) を参照してください。

## 拡張エラー・データ・コマンド構造体

CS_EED_CMD プロパティは、拡張エラー・データを含む CS_COMMAND 構造体へのポインタを定義します。

サーバ・メッセージ・コールバックでは、Client-Library は、メッセージを記述する CS_SERVERMSG 構造体の *status* フィールドの CS_HASEED ビットを設定することにより、拡張エラー・データが使用可能であることを示します。

拡張エラー・データが得られない場合に、CS_EED_CMD を取得するとエラーとなります。

「拡張エラー・データ」(143 ページ) を参照してください。

## 拡張フェールオーバ

CS_PROP_EXTENDEDFAILOVER は、デフォルトで TRUE に設定され、CS_HAFAILOVER も TRUE の場合にのみ使用されます。この場合、ディレクトリ・サービスから最初に取得された情報の代わりに使用するネットワーク・アドレスの一覧が HA Aware により送信されます。CS_PROP_EXTENDEDFAILOVER が FALSE に設定されている場合、ディレクトリ・サービス・レイヤからフェールオーバ情報が取得されます。

## エンドポイントのポーリング

CS_ENDPOINT プロパティを使用すると、アプリケーションはファイル記述子、つまりリモート・サーバへの接続と対応する番号を取得できます。Client-Library 呼び出しと Server-Library 呼び出しの両方を含むゲートウェイ・アプリケーションでは、この機能が便利な場合があります。Client-Library を使用するリモート・サーバへの接続を確立した後は、Server-Library ルーチン srv_poll でその接続と対応するファイル記述子を使用できます。srv_poll ルーチンを呼び出すと、接続で使用できる結果が存在するようになるまで、現在のスレッドは再スケジュールされます。

CS_ENDPOINT プロパティの使用は、現時点では UNIX プラットフォームに固有の機能であるため、おすすめできません。

## 公開フォーマット

CS_EXPOSE_FMTS は、Client-Library がフォーマット結果セットを公開するかどうかを指定します。

フォーマット結果セットには、関連する結果セットについてのフォーマット情報が含まれています。フォーマット情報には、結果セットの項目数および各項目の記述が含まれています。フォーマット結果セットには次の 2 つのタイプがあります。

- CS_ROWFMT_RESULT − 通常ロー結果セットのフォーマット情報が含まれる。
- CS_COMPUTEFMT_RESULT − 計算ロー結果セットのフォーマット情報が含まれる。

コマンドにより生成されたすべてのフォーマット結果セットは、コマンドにより生成された通常ローおよび計算ロー結果セットより優先されます。

フォーマット結果セットが公開されていない場合、アプリケーションは、結果セットを処理している間のみ、フォーマット情報を取得します。たとえば、ct_results が CS_ROW_RESULT という *result_type* を返した後に、アプリケーションが ct_res_info を呼び出して結果セットのカラム数を調べたり、ct_describe を呼び出して各カラムの記述を取得したりします。

フォーマット結果セットの公開は、結果セットを処理する前に、アプリケーションがフォーマット情報を取得できるようにします。

フォーマット結果セットの公開は、Adaptive Server Enterprise の結果を再パッケージしてから外部のクライアントに送るゲートウェイ・アプリケーションに役立ちます。

アプリケーションは、CS_EXPOSE_FMTS プロパティを CS_TRUE に設定して、フォーマット結果を公開します。

「フォーマット結果」(283 ページ ) を参照してください。

## 外部設定

CS_EXTERNAL_CONFIG プロパティは、Client-Library が設定ファイルを読み込んで、プロパティ、サーバ・オプション、機能のデフォルト値を設定するかどうかを指定します。この機能については、「ランタイム設定ファイルの使い方」(352 ページ ) を参照してください。

## 追加情報

CS_EXTRA_INF は、ct_diag を呼び出して SQLCA、SQLCODE、または SQLSTATE 構造体に書き込む追加情報を Client-Library が返すかどうかを指定します。

この追加情報には、直前のコマンドで影響を受けたローの数などがあります。アプリケーションは、ct_res_info(CS_ROW_COUNT) を呼び出すことによって、この情報を取得します。

アプリケーションが SQLCA、SQLCODE、または SQLSTATE にメッセージを取得していない場合、通常、Client-Library メッセージとして追加情報が返されます。

## バインド

CS_HAVE_BINDS プロパティは、現在の結果セットに対して、保存された結果バインドがあるかどうかを知らせます。このプロパティは、ct_cmd_props ルーチンを呼び出すことによって取得できます。

CS_HAVE_BINDS プロパティは、必ず CS_STICKY_BINDS プロパティとともに使用されます。CS_COMMAND 構造体上で同じコマンドを繰り返して実行するバッチ処理アプリケーションの中には、Client-Library が同じコマンドの実行の合間に結果バインドを保存するように、CS_STICKY_BINDS コマンド・プロパティを設定するものもあります。これらのアプリケーションは CS_HAVE_BINDS プロパティをチェックして、保存されたバインドが現在の結果セットに適しているかどうかを調べます。このプロパティの値が CS_TRUE に設定されている場合は、1 つまたは複数のプログラム変数が現在の結果セット内の 1 つまたは複数の項目にバインドされていることを示します。

CS_STICKY_BINDS プロパティの詳細については、「継続結果バインド」(261 ページ ) を参照してください。

フェッチ可能なデータがコマンド構造体上にあることが ct_results ルーチンによって示されると、CS_HAVE_BINDS プロパティは正しいことが保証されます。

## 再送可能なコマンド

CS_HAVE_CMD プロパティは、前に実行されたサーバ・コマンドをアプリケーションが再送できるかどうかを指定します。このプロパティは読み込み専用です。このプロパティの値が CS_TRUE に設定されている場合は、再送可能なコマンドがあることを示します。

Client-Library では、アプリケーションは、前の実行結果が処理されたらすぐにいくつかのタイプのコマンドを再送できます。

コマンドを再送するために、アプリケーションは次のことを行います。

1. コマンドのパラメータ・ソース変数がある場合は、その変数内の値を更新します。ct_command ルーチン、ct_cursors ルーチン、または ct_dynamic ルーチンを呼び出してコマンドを開始した後で、ct_setparam ルーチンを呼び出してパラメータ・ソース変数のアドレスが指定されている必要があります。

2. ct_send ルーチンを呼び出して、コマンドを再送します。ct_send ルーチンは、更新されたパラメータ値を読み込みます。

すべてのタイプのコマンドを再送できるわけではありません。「コマンドの再送信」(643 ページ ) を参照してください。

コマンドを再送するアプリケーションでは、CS_STICKY_BINDS プロパティを設定して、そのコマンドの最初の実行結果を処理している間に確立されたバインドを再使用すると便利な場合があります。このプロパティの詳細については、「継続結果バインド」(261 ページ ) を参照してください。

## リストア可能なカーソル・オープン・コマンド

CS_HAVE_CUROPEN プロパティは、前に実行された ct_cursor カーソル・オープン・コマンドのバッチをアプリケーションがリストアできるかどうかを指定します。このプロパティは読み込み専用です。このプロパティが CS_TRUE に設定されている場合は、リストア可能なカーソル・オープン・コマンドのバッチがあることを示します。

アプリケーションは ct_cursor ルーチンを呼び出して、カーソル・オープン・コマンドをリストアします。この機能の詳細については、「カーソル・オープン・コマンドのリストア」(488 ページ ) を参照してください。

オープン・カーソルは、カーソル・オープン・コマンドをリストアできるようになるまではクローズしておいてください。CS_HAVE_CUROPEN プロパティは、Client-Library が最初のカーソル・オープン・コマンドのコマンド情報を保存したことを示します。これは、カーソルが現在のステータスである間はアプリケーションが正規にそのカーソルを再オープンできることを示すわけではありません。

CS_CUR_STATUS プロパティは、コマンド構造体で宣言されたカーソルがあれば、その現在のステータスをアプリケーションに知らせます。このプロパティの詳細については、「カーソル・ステータス」(243 ページ ) を参照してください。

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では、CS_STICKY_BINDS プロパティを設定して、そのコマンドの最初の実行結果を処理している間に確立されたバインドを再使用すると便利な場合があります。このプロパティの詳細については、「継続結果バインド」(261 ページ ) を参照してください。

## 隠しキー

CS_HIDDEN_KEYS は、結果セットの一部である「隠しキー」を Client-Library が公開するかどうかを指定します。隠しキーは、クエリで明示的に選択されるのではなく、テーブル・キーの一部またはすべてを構成するためにクライアントへ返されるカラムです。

通常、これらのカラムの存在は見えないようになっています。クライアントからは、これらのカラムが結果セットの一部であることはわかりません。

クライアントは、CS_HIDDEN_KEYS プロパティを CS_TRUE に設定することにより、隠しキーを公開します。

いったん隠しキーが公開されると、それらのキーは、通常のカラムとして返されます。たとえば、アプリケーションが ct_res_info を呼び出して結果セットのカラム数を取得する場合、その数には公開されたカラムが含まれています。アプリケーションは、公開されたカラムのロー値のバインドおよびフェッチを行うことができます。

カラムが公開された隠しキーである場合、ct_describe には、そのカラムを記述する *status* フィールドのビットマスクに CS_HIDDEN が含まれています。

アプリケーションは、テーブル・キーとともに使用して ct_keydata を呼び出すことで、カーソル位置を変更します。この実行方法の詳細については、「ct_keydata 」(580 ページ ) を参照してください。

結果が未処理であるか、カーソルがオープンしている場合、アプリケーションではコマンド・レベルで CS_HIDDEN_KEYS プロパティを設定できません。

## ホスト名

CS_HOSTNAME は、ホスト・マシン名であり、サーバへのログイン時に使用されます。

Adaptive Server Enterprise は、*master* データベースの *sysprocesses* テーブルにプロセスのホスト名のリストを作成します。

## interfaces ファイルのロケーション

CS_IFILE は、*interfaces* ファイルの名前およびロケーションを定義します。

*interfaces* ファイルには、ネットワーク上で使用可能なすべてのサーバの名前とネットワーク・アドレスが記録されています。これにより、クライアントとサーバ間の通信が確立されます。クライアントが接続するサーバごとに、*interfaces* ファイルには、サーバ名、マシン名、およびサーバ・アドレスを持つエントリが存在します。Client-Library アプリケーションでは、ct_connect を呼び出すたびに *interfaces* ファイルが検索されます。

ほとんどのプラットフォームでは、ct_config で特定の *interfaces* ファイルが指定されなかった場合、ct_connect は、SYBASE 環境変数または論理名によって指定されたディレクトリの *interfaces* という名のファイルを使用しようとします (Windows プラットフォームでは *sql.ini* ファイルが使用されます )。SYBASE が設定されていない場合、ct_connect は、「sybase」という名のユーザのホーム・ディレクトリにある *interfaces* という名のファイルを使用しようとします。

「interfaces ファイル」(157 ページ ) を参照してください。

---

**注意**　すべてのプラットフォームで、*interfaces* ファイルを使用するわけではありません。使用しているプラットフォームで *interfaces* ファイルを使用するかどうかが不明な場合は、システム管理者に問い合わせるか、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

---

*interfaces* ファイルの代替のデフォルトのファイル名とパスは、CS_DEFAULT_IFILE プロパティで指定できます。CS_DEFAULT_IFILE プロパティの詳細については、『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。

## ロケール情報

CS_LOC_PROP は、ローカライゼーション情報を含む CS_LOCALE 構造体を定義します。ローカライゼーション情報には、言語、文字セット、日時フォーマット、および照合順が含まれています。

アプリケーションは、接続レベルで ct_con_props を呼び出し、CS_LOC_PROP を設定または取得できます。

- CS_LOC_PROP を設定するには、アプリケーションは、ct_con_props に CS_LOCALE 構造体を渡します。ct_con_props は情報を CS_LOCALE からコピーし、それを内部で記録します。ct_con_props を呼び出した後に、アプリケーションは、CS_LOCALE の割り付けを解除します。
- CS_LOC_PROP を取得するとき、アプリケーションは ct_con_props に CS_LOCALE 構造体を渡します。ct_con_props は、現在のローカライゼーション情報をこの CS_LOCALE にコピーします。

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

アプリケーションは、コンテキスト・レベルで cs_config を呼び出し、CS_LOC_PROP を設定または取得します。

アプリケーションが、cs_config を呼び出してコンテキストのローカライゼーション情報を定義しない場合、コンテキストは、割り付け時に割り付けられるデフォルト・ローカライゼーション値を使用します。ほとんどのプラットフォームでは、環境変数がデフォルト値を決定します。使用しているプラットフォームのデフォルト・ローカライゼーション値の割り付け方法については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

## ログイン・ステータス

CS_LOGIN_STATUS は、接続がオープンされている場合は CS_TRUE、接続がオープンされていない場合は CS_FALSE です。このプロパティは取得のみ可能です。

ct_connect は、接続をオープンするために使用されます。

ct_close は、接続をクローズするために使用されます。

## ログイン・タイムアウト

CS_LOGIN_TIMEOUT は、Client-Library が接続を試みるときに、ログイン応答の待ち時間の秒単位での長さを定義します。Client-Library アプリケーションは、ct_connect を呼び出すことによって接続を試みます。

このタイムアウトは、クライアントが要求を出してからサーバの応答を受信するまでの間の、可能な往復遅延を指定します。ct_connect ルーチンが返る前に、クライアントとサーバの間を複数回往復する場合があります。CS_LOGIN_TIMEOUT プロパティは、それぞれの往復に適用されます。

デフォルト・タイムアウト値は、60 秒です。タイムアウト値 CS_NO_LIMIT は、タイムアウト時間が無限であることを表します。

---

**注意**　CS_LOGIN_TIMEOUT が適用されるのは同期接続の場合に限られます。

---

ct_con_props では、接続ごとに CS_LOGIN_TIMEOUT 値を指定できます。

「タイムアウト・エラーの処理」(268 ページ ) を参照してください。

## ループ遅延

CS_LOOP_DELAY プロパティは、サーバ名と対応する一連のネットワーク・アドレスをリトライする前に ct_connect が待つ遅延を秒単位で指定します。デフォルトは 0 です。

CS_RETRY_COUNT プロパティは、Client-Library がそれぞれのアドレスに対して何回リトライするかを指定します。「リトライ回数」(264 ページ ) を参照してください。

CS_LOOP_DELAY プロパティと CS_RETRY_COUNT プロパティは、ログイン・ダイアログを確立する場合にだけ適用されます。サーバが応答するアドレスを Client-Library が検出すると、Client-Library とサーバの間のログイン・ダイアログが開始されます。ログインが失敗しても、Client-Library は他のアドレスをリトライすることはありません。

アドレスは、ネットワークベースのディレクトリ内と Sybase *interfaces* ファイル内のどちらかのサーバ名と対応しています。

UNIX プラットフォームでは、CS_RETRY_COUNT プロパティと CS_LOOP_DELAY プロパティに対してアプリケーションが指定した設定を上書きするように、サーバの *interfaces* ファイル・エントリを設定できます。

使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

## ログインのリダイレクト

CS_PROP_REDIRECT は、操作中のライブラリのバージョンに関係なく、デフォルトで TRUE に設定されます。CS_PROP_REDIRECT が TRUE に設定された場合、ct_connect はログインのリダイレクトを有効にしてサーバ・ログインの試行を開始します。CS_PROP_REDIRECT が FALSE に設定された場合、ct_connect はログインのリダイレクトを無効にしてサーバ・ログインの試行を開始します。

ログインのリダイレクトが発生すると、追加のデータがクライアントに送信されるため、ログインに必要な時間が長くなることがあります。さらに、リダイレクトされたクライアントで、ログイン・プロセスを再度開始する必要が生じる可能性もあります。

## 最大接続数

CS_MAX_CONNECT は、あるコンテキストで同時にオープン可能な接続の最大数を定義します。CS_MAX_CONNECT のデフォルト値は 25 です。負の値および 0 は、CS_MAX_CONNECT では使用できません。

ct_config を呼び出し、現在オープンしている接続数よりも少ない値を CS_MAX_CONNECT に設定した場合、ct_config は、CS_MAX_CONNECT 値を変更しないで Client-Library エラーを生成し、CS_FAIL を返します。

## メモリ・プール

CS_MEM_POOL は、Client-Library が、メモリ要件を満たすために使用するメモリ・プールを識別します。

通常、Client-Library ルーチンは、malloc を呼び出すことによって、メモリ要件を満たします。しかし、malloc の実装のすべてが再入可能とはかぎらないため、割り込みレベルで呼び出される Client-Library が malloc を使用することは安全ではありません。このため、Client-Library がシグナル駆動型ネットワーク I/O を使用するシステム (UNIX システムなど ) では、完全非同期アプリケーションは、Client-Library にそのメモリ要件を満たす代替手段を提供する必要があります。これは、スレッド駆動型ネットワーク I/O を使用しているプラットフォームや、完全非同期接続を使用しないアプリケーションの場合には必要ありません。使用しているプラットフォームのネットワーク I/O 方式の詳細については、『Open Client/Server プログラマーズ・ガイド補足』を参照してください。

Client-Library には、非同期アプリケーションが Client-Library のメモリ要件を満たせるように次の 2 種類のメカニズムが用意されています。

- アプリケーションは CS_MEM_POOL プロパティを使用して、メモリ要件に対して使用するメモリ・プールを Client-Library に提供できます。
- アプリケーションは、CS_USER_ALLOC および CS_USER_FREE プロパティを使用して、Client-Library がオペレーティング・システムの割り込みレベルで安全に呼び出すことができるメモリの割り付けルーチンをインストールできます。

完全非同期アプリケーションが Client-Library にメモリ要件を安全に満たす手段を提供できない場合、Client-Library の動作は予測不可能です。

アプリケーションが Client-Library の最小プール・サイズの要件に満たないメモリ・プールを設定しようとすると、ct_config は CS_FAIL を返します。

UNIX システムでは、接続ごとに約 6K のメモリ・プールが必要です。

Client-Library は、次の順で次のソースから必要なメモリを確保しようとします。

1. メモリ・プール
2. ユーザ提供の割り付けおよび解放ルーチン
3. システム・ルーチン

接続が必要なメモリを得ることができない場合、Client-Library は、接続に 「dead」とマーク付けします。

アプリケーションは、CS_MEM_POOL によって識別されたメモリの割り付けおよび解放を行います。

アプリケーションは、action に CS_SET を設定し、*buffer* に新しいプールのアドレスを設定して ct_config を呼び出すことにより、メモリ・プールを置き換えることができます。

アプリケーションは、次の 2 とおりの方法でメモリ・プールをクリアします。

- action に CS_SET を、*buffer* に NULL を設定して、ct_config を呼び出す方法
- action に CS_CLEAR を設定して ct_config を呼び出す方法

アプリケーションは、現在 CS_CONNECTION 構造体があるコンテキストに対して、メモリ・プールの設定およびクリアを行うことはできません。コンテキストは、すべての CS_CONNECTION 構造体を削除してから、メモリ・プールをクリアする必要があります。

## ネットワーク I/O

CS_NETIO は、接続が同期、完全非同期、または遅延非同期のいずれであるかを指定します。

- 「同期 (synchronous)」接続では、サーバの応答を必要とするルーチンが、応答を受け取るまで接続をブロックします。
- 「完全非同期 (fully asynchronous)」接続では、サーバの応答を必要とするルーチンが、すぐに CS_PENDING を返します。応答が到着し、ルーチンがその処理を終えると、Client-Library が接続の完了コールバックを自動的に呼び出します。

  ホスト・プラットフォームに応じて、システム割り込みレベルで ( シグナル駆動型ネットワーク I/O を使用しているプラットフォームの場合 )、または Client-Library のランタイム・スレッドから ( スレッド駆動型ネットワーク I/O を使用しているプラットフォームの場合 )、完了コールバックが呼び出されます。使用しているプラットフォームのネットワーク I/O 方式の詳細については、使用しているプラットフォームの『Open Client/Server プログラマーズ・ガイド補足』を参照してください。
- 「遅延非同期 (deferred-asynchronous)」接続では、サーバの応答を必要とするルーチンが、すぐに CS_PENDING を返します。ルーチンが完了しているかどうかを調べるため、接続は ct_poll を呼び出す必要があります。アプリケーションが完了コールバックをインストールして、ルーチンが完了している場合、ct_poll ルーチンは完了コールバックを呼び出してから返ります。

マルチスレッド機能またはシグナル駆動型ネットワーク I/O をサポートしていないプラットフォームでは、接続は同期か遅延非同期のどちらかに限定されます。CS_NETIO プロパティが CS_ASYNC_IO に設定されている場合でも、接続は遅延非同期になり、アプリケーションは ct_poll ルーチンを呼び出して完了のポーリングを行う必要があります。

---

**警告！** Open Server ゲートウェイ・アプリケーションでは、CS_NETIO プロパティを CS_ASYNC_IO に設定できません。Open Server スレッド・スケジューラは、Open Server アプリケーションでのマルチタスク機能を可能にします。

---

アプリケーションでは、**buffer* を CS_DEFER_IO に設定して ct_config を呼び出すことによって、コンテキスト・レベルでのみ遅延非同期接続を設定できます。CS_DEFER_IO は、接続レベルでは有効値ではありません。

非同期接続は、接続の親コンテキストと一致する非同期 I/O のタイプを使用します。たとえば、アプリケーションが、コンテキスト・レベルで遅延非同期接続を設定して、さらに、コンテキスト内で同期接続を作成したと仮定します。アプリケーションが、後から **buffer* を CS_ASYNC_IO に設定して、ct_con_props を呼び出してこの接続を非同期にした場合、接続は完全非同期ではなく、遅延非同期になります。

コンテキストは、同期および非同期の両方の接続を含むことができます。しかし、コンテキスト内のすべての非同期接続は、完全非同期または遅延非同期のどちらかでなければなりません。

アプリケーションによる CS_NETIO の使用には、次の制限があります。

- アプリケーションは、コンテキストがオープン接続を持っている場合、コンテキストの CS_NETIO を設定することができません。
- アプリケーションは、接続がアクティブ・コマンドまたは未処理の結果を持っている場合、接続の CS_NETIO を設定することができません。

「非同期プログラミング」(12 ページ ) を参照してください。

## トランケートの禁止

CS_NO_TRUNCATE は、Client-Library が、CS_MAX_MSG - 1 バイトより長い Client-Library およびサーバ・メッセージをトランケートするか連続させるかを指定します。

Client-Library のデフォルトの動作では、CS_MAX_MSG - 1 バイトより長いメッセージがトランケートされます。ただし、Client-Library がメッセージを連続させている場合、Client-Library は、メッセージの全テキストを返すために必要な数の 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。1 番目のまとまりにはメッセージの CS_MAX_MSG バイトが含まれ、2 番目のまとまりには null ターミネータが含まれます。

「長いメッセージの連続化」(141 ページ ) を参照してください。

## API チェックの禁止

CS_NOAPI_CHK プロパティは、アプリケーションが Client-Library ルーチンを呼び出すときに、Client-Library が引数とステータスのチェックを行うかどうかを指定します。

CS_NOAPI_CHK プロパティが CS_FALSE ( デフォルト値 ) に設定されている場合、Client-Library は引数とステータスのチェックを行います。この設定の場合、ユーザが Client-Library ルーチンを呼び出すたびに、Client-Library は次のエラー・チェックを行います。

- パラメータ値の検証
- 可視構造体内のフィールド値に無効な組み合わせがあるかどうかのチェック
- アプリケーションがその関数を実行するための正しいステータスになっていることの確認

問題が検出された場合、ルーチンは失敗して、エラー・メッセージが表示されます。

CS_NOAPI_CHK が CS_TRUE の場合、Client-Library の通常のチェックは無効になります。この設定の結果、次のようになります。

- アプリケーションが無効な引数を渡したり、不正なタイミングでルーチンを呼び出したりすると、アプリケーションではメモリの破壊、メモリのアクセス違反、または不正な結果が生じることがあります。
- API チェックが無効になっている場合、Client-Library は使用上のエラーがあるかどうかをチェックしません。API チェックが無効になっている場合、使用上のエラーの中にはトラップされないものもあります。これらのエラーが発生すると、API チェックが有効になっている場合はエラー・メッセージが表示されますが、API チェックが無効になっている場合はアプリケーションが不正な動作をするようになります。

**警告！**アプリケーションを完全にデバッグするまでは、API チェックを無効にしないでください。

## 文字変換は不要

CS_NOCHARSETCNV_REQD プロパティは、サーバがサーバ自体の文字セットとの間で双方向に文字データの変換を行うかどうかを指定します。

CS_NOCHARSETCNV_REQD が CS_FALSE ( デフォルト ) の場合、接続の文字セットがサーバの文字セットと一致していません。サーバはクライアントと通信するときに、サーバ自体の文字セットとの間の文字変換を双方向で実行します。

CS_NOCHARSETCNV_REQD が CS_TRUE に設定されている場合、サーバは接続の文字セットにかかわらず、文字セットの変換を実行しません。これは、サーバが Open Server ゲートウェイの場合など、サーバがデータを解釈しないで別のサーバに渡す場合に役立ちます。

接続がオープンされた後は、CS_NOCHARSETCNV_REQD プロパティを設定できません。

接続の文字セットは、接続の CS_LOCALE 構造体内で定義されます。「ロケール情報」(252 ページ ) を参照してください。

## 割り込みの禁止

CS_NOINTERRUPT では、Client-Library の完了イベントがアプリケーションに割り込めるようにするかどうかを設定します。

CS_NOINTERRUPT が CS_TRUE の場合、CS_NOINTERRUPT が CS_FALSE にリセットされるまで、完了イベントは遅延されます。

アプリケーションは、CS_NOINTERRUPT プロパティを使用して、コードのクリティカル・セクションを保護します。

**注意**　Client-Library の CS_NOINTERRUPT プロパティは、オペレーティング・システムの割り込み処理には何も影響を与えません。CS_NOINTERRUPT が影響を与えるのは完了イベントのみです。ノーティフィケーション・イベントへの影響はありません。

## パケット・サイズ

CS_PACKETSIZE は、TDS (Tabular Data Stream) パケットを送信するときに Client-Library が使用するパケット・サイズを指定します。

アプリケーションが、大量のデータの送受信を必要とする場合、パケット・サイズを大きくすると効率が高まります。

Open Client には、CS_REQ_SRVPKTSIZE と CS_NO_SRVPKTSIZE の 2 つの機能が含まれています。

- CS_REQ_SRVPKTSIZE は、必ずこのバージョンの CT-Library により設定され、ct_capability を使用して取得できます。
- CS_NO_SRVPKTSIZE は、要求されたサイズより大きいパケット・サイズをクライアントが処理できない場合に使用され、ct_capability を使用して設定および取得されます。

## 親構造体

CS_PARENT_HANDLE では、コマンドまたは接続構造体の親構造体へのポインタを定義します。

- コマンド構造体レベルで取得された場合、CS_PARENT_HANDLE は、コマンド構造体の親接続構造体へのポインタとなります。
- 接続構造体レベルで取得された場合、CS_PARENT_HANDLE は、接続構造体の親コンテキスト構造体へのポインタとなります。

## *text* データと *image* データの部分更新

Open Client は、text カラムと image カラムの部分更新をサポートしています。CS_PARTIAL_TEXT は、クライアントが部分更新を実行する必要があるかどうかを示します。このプロパティは、ct_con_props() を使用して接続レベル、ct_config() を使用してコンテキスト・レベルで設定できます。CS_PARTIAL_TEXT の有効な値は、CS_TRUE と CS_FALSE です。

CS_PARTIAL_TEXT プロパティは、サーバへの接続が確立される前に設定する必要があります。サーバが部分更新をサポートしていない場合は、CS_PARTIAL_TEXT はデフォルト値である CS_FALSE に再設定されます。

## パスワード

CS_PASSWORD は、サーバへのログイン時に接続が使用するパスワードを定義します。

接続でネットワークベースの認証が必要な場合、パスワードは無視されます。アプリケーションは CS_SEC_NETWORKAUTH プロパティを設定して、ネットワークベースの認証を要求します。「ログイン認証サービスの要求」(295 ページ ) を参照してください。

ネットワーク認証を使用しないアプリケーションは、Client-Library がパスワードを暗号化した形式でサーバに送信できるように、CS_SEC_ENCRYPT プロパティを設定できます。「Client-Library アプリケーションでのパスワードの暗号化の使用」(318 ページ ) を参照してください。

## 継続結果バインド

通常、アプリケーションがコマンドの結果を処理した後に、Client-Library はアプリケーションの送信先変数とコマンドとの間のバインドを削除します。

ただし、CS_STICKY_BINDS プロパティは、ct_bind によって確立されたバインドが、あるコマンドが繰り返し実行される間継続するかどうかを決定します。CS_STICKY_BINDS プロパティが有効 (CS-TRUE) になっている場合、アプリケーションがルーチン ct_command、ct_cursor、ct_dynamic、または ct_sendpassthru を呼び出して新しいコマンドを開始するまでは、Client-Library はバインドを削除しません。

CS_STICKY_BINDS プロパティを CS_TRUE に設定してから、ct_send ルーチンを呼び出して、結果バインドが保存されるコマンドを実行してください。このプロパティを設定すると、そのコマンド構造体でその後実行されるすべてのコマンド処理に影響を与えます。

CS_STICKY_BINDS プロパティは、同じコマンドを繰り返し実行するアプリケーションだけで設定し、そのコマンドによって返される結果フォーマットが異なる可能性がない場合にだけ設定してください。コマンドの結果フォーマット情報は、次に示す結果セットの一連の特性で構成されます。

- 結果タイプ (ct_results の *result_type* パラメータによってアプリケーションに示される )
- アプリケーションが ct_res_info を呼び出して取得できるカラム数 ( フェッチ可能な結果にのみ適用 )
- アプリケーションがカラムごとに ct_describe を呼び出して取得できる各カラムのフォーマット ( フェッチ可能な結果にのみ適用 )

サーバ・コマンドに条件論理がある場合、そのコマンドの 2 回目以降の実行で返される結果のフォーマットは、最初の実行で返される結果のフォーマットとは一致しない可能性があります。この場合、最初の実行で確立されたバインドは Client-Library によって自動的にクリアされます。Client-Library が結果のフォーマットで不一致を検出した場合、ct_results ルーチンは情報エラーを表示します ( そして CS_SUCCEED を返します )。

### 継続バインド用のプログラム構造体

アプリケーションは、コマンドがサーバに送信される前に CS_STICKY_BINDS コマンド・プロパティを CS_TRUE にしてバインドを再使用できます。アプリケーションは、CS_HAVE_BINDS コマンド・プロパティを検査して、結果セットに対してバインドが確立されているかどうか調べることができます。

たとえば、1 つの select 文を含むストアド・プロシージャを実行するために同一の RPC コマンドを繰り返して実行するアプリケーションがあるとします。このようなアプリケーションでは、次のようなプログラム論理を使用して、コマンドを再実行し、結果バインドを再使用できます。

```
/*
 ** Enable persistent result bindings.
 */
 ct_cmd_props to set CS_STICKY_BINDS to CS_TRUE

 /*
 ** Initiate the RPC command.
 */
 ct_command(CS_RPC_COMMAND, proc_name)
 ct_setparam for each parameter
 set values in parameter source variables
 ct_send
 loop while ct_results returns CS_SUCCEED
 switch(result_type)
 case CS_ROW_RESULT:
 ct_bind for each column
 loop on ct_fetch
           ... process row data ...
        end loop
     case CS_STATUS_RESULT:
        ct_bind for the procedure’s return status
        loop on ct_fetch
           ... process the return status value ...
        end loop
     ... other cases...
   end switch
 end loop

/*
 ** Change the input parameter values and resend the command.
 */
 set values in parameter source variables
 ct_send
```

```
loop while ct_results returns CS_SUCCEED
  switch(result_type)
    case CS_ROW_RESULT:
       (optional) ct_cmd_props to check CS_HAVE_BINDS
       loop on ct_fetch
          ... process row data ...
       end loop
    case CS_STATUS_RESULT:
       (optional) ct_cmd_props to check CS_HAVE_BINDS
       loop on ct_fetch
          ... process the return status value ...
       end loop
    ... other cases...
  end switch
end loop

/*
** Execute a new command. A call to ct_command, ct_cursor, or
** ct_dynamic clears the previous initiated command from the
** command structure.
*/
ct_command
... and so forth ...
```

---

**注意**　コマンドが複数の結果セットを返す場合 ( たとえば、上記の例のストアド・プロシージャに複数の select 文が含まれる場合 ) には、上記の結果ループの論理が ct_res_info(CS_CMD_NUMBER) に対する呼び出しを使用して、それぞれの結果セットを区別します。

---

CS_STICKY_BINDS プロパティが CS_TRUE に設定されている場合、Client-Library は結果セット・フォーマットを保存して比較する必要があり、若干の内部オーバヘッドが発生します。同じコマンドを繰り返し実行しないで結果バインドを再使用するアプリケーションでは、このプロパティをデフォルト設定 (FALSE) のままにしておいてください。

CS_STICKY_BINDS プロパティは、拡張エラー・データまたはノーティフィケーション・パラメータ値を制御するコマンド構造体で確立されたバインドには影響を与えません。アプリケーションは、それぞれ CS_EED_CMD 接続プロパティと CS_NOTIF_CMD 接続プロパティとしてこれらのコマンド構造体にアクセスします。アプリケーションは、これらのコマンド構造体からフェッチするときには必ず再バインドする必要があります。

上記ルーチンの使用方法の詳細については、「第 3 章 ルーチン」の各ルーチンのリファレンス・ページを参照してください。

アプリケーションは CS_HAVE_BINDS コマンド・プロパティをチェックして、保存されたバインドが現在の結果セットに対して確立されているかどうかを調べます。「バインド」(248 ページ)、「コマンドの再送信」(643 ページ)、「カーソル・オープン・コマンドのリストア」(488 ページ) を参照してください。

## リトライ回数

CS_RETRY_COUNT プロパティは、ct_connect がサーバ名と対応する一連のネットワーク・アドレスをリトライする回数を指定します。デフォルトは 0 です。

CS_LOOP_DELAY プロパティは、アドレスのリスト全体をリトライする前に ct_connect ルーチンが待つ遅延時間を秒単位で指定します。「ループ遅延」(253 ページ) を参照してください。

CS_LOOP_DELAY プロパティと CS_RETRY_COUNT プロパティは、ログイン・ダイアログを確立する場合にだけ適用されます。サーバが応答するアドレスを Client-Library が検出すると、Client-Library とサーバの間のログイン・ダイアログが開始されます。ログインが失敗しても、Client-Library は他のアドレスをリトライすることはありません。

アドレスは、ネットワークベースのディレクトリ内と Sybase *interfaces* ファイル内のどちらかのサーバ名と対応しています。詳細については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

UNIX プラットフォームでは、CS_RETRY_COUNT プロパティと CS_LOOP_DELAY プロパティに対してアプリケーションが指定した設定を上書きするように、サーバの *interfaces* ファイル・エントリを設定できます。

## セキュリティ・プロパティ

すべての CS_SEC プロパティの詳細については、「セキュリティ機能」(290 ページ) を参照してください。

## サーバ名

CS_SERVERNAME は、接続先サーバの名前を指定します。

CS_SERVERNAME プロパティは読み込み専用です。ct_connect ルーチンを呼び出して接続がオープンされた後は、アプリケーションはその値の取得のみできます。

---

**注意**　その接続で外部設定が有効になっている場合、設定ファイル内の CS_SERVERNAME 定義を修正してサーバ名を変更できます。「外部設定の有効化」(353 ページ ) を参照してください。

---

接続先のサーバの名前を指定するには、ct_connect にそのサーバ名を渡します。

## TCP ソケット・バッファ・サイズの設定

CS_TCP_RCVBUF および CS_TCP_SNDBUF の各 context/connection プロパティを使用して、クライアント側で TCP ソケットの入出力バッファ・サイズを設定します。Open Client アプリケーションは、これらのプロパティ設定を使用して、オペレーティング・システムの setsockopt コマンドでバッファ・サイズを設定します。setsockopt は TCP の connect コマンドおよび accept コマンドの前に呼び出す必要があるので、これらのプロパティを設定してから接続の確立を試みてください。

これらのプロパティをアプリケーションに応じて設定します。たとえば、クライアントが大量のデータをサーバに送信することが想定される場合は、CS_TCP_SNDBUF を大きな値に設定して対応するバッファ・サイズを増加します。

---

**注意**　SRV_S_TCP_RCVBUF および SRV_S_TCP_SNDBUF の各サーバ・プロパティを使用して、クライアント側で TCP ソケットの入出力バッファ・サイズを設定します。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。

---

**コンテキスト例**

```
ct_config(*context, CS_SET, CS_TCP_RCVBUF, &bufsize,
CS_UNUSED, NULL);
```

**接続例**

```
ct_con_props(*connection, CS_SET, CS_TCP_RCVBUF,
&bufsize, CS_UNUSED, NULL);
```

## TDS バージョン

CS_TDS_VERSION は、接続が使用中の Tabular Data Stream (TDS) のプロトコル・バージョンを定義します。

CS_TDS_VERSION は、ネゴシエートされたログイン・プロパティであるため、その値をログイン・プロセスで変更できます。アプリケーションは、ct_connect を呼び出す前に、CS_TDS_VERSION を設定して TDS レベルを要求できます。ct_connect が接続を作成するとき、サーバが要求された TDS バージョンを提供することができない場合は、新しい ( 低い ) TDS バージョンがネゴシエートされます。アプリケーションは接続確立後に CS_TDS_VERSION 値を取得し、実際に使用されている TDS バージョンを判別します。

表 2-32 は、CS_TDS_VERSION の記号値を示します。以前のバージョンでサポートされていた機能は、新しいバージョンでも継承されています。

**表 2-32：CS_TDS_VERSION の値**

| 記号値 | 意味 | 新しくサポートされた機能 |
|---|---|---|
| CS_TDS_40 | 4.0 TDS | ブラウズ・モード、text および image 処理、リモート・プロシージャ・コール、バルク・コピー |
| CS_TDS_42 | 4.2 TDS | 国際化のサポート |
| CS_TDS_46 | 4.6 TDS | レジスタード・プロシージャ、TDS パススルー、ネゴシエート可能な TDS パケット・サイズ、マルチバイトの文字セット |
| CS_TDS_50 | 5.0 TDS | カーソル |

他の値に設定されていなければ、CS_TDS_VERSION は、アプリケーションが ct_init で要求した CS_VERSION レベルに基づく値をデフォルトとします。

接続の CS_TDS_VERSION レベルが、その親コンテキストの CS_VERSION レベルに関連付けられたデフォルト TDS レベルより高くなることはありません。

たとえば、CS_VERSION_110 以降のバージョン・レベルには、TDS レベル 5.0 が関連付けられています。アプリケーションが *version* を CS_VERSION_110 に設定して ct_init を呼び出す場合、そのコンテキスト内で作成される接続はすべて 5.0 以下の CS_TDS_VERSION に制限されます。

アプリケーションが、CS_TDS_VERSION プロパティを設定する場合、Client-Library は、既存の機能値を新しい TDS バージョンに対応するデフォルト機能値で上書きします。このため、アプリケーションは CS_TDS_VERSION を設定してから、接続の機能を設定する必要があります。

## text および image の最大値

CS_TEXTLIMIT は、アプリケーションが受け取ることができる最長の text 値または image 値のバイト単位での最大値を示します。Client-Library は、この最大値を超える text または image 値のすべての部分を読み込みますが、無視してしまいます。

CS_TEXTLIMIT のデフォルト値は、CS_NO_LIMIT です。これは、Client-Library が、サーバが送ったすべてのデータを読み込み、アプリケーションへ返すことを意味します。

text 値が非常に大きい場合、text 値全体がネットワークで返されるまでに多少時間がかかります。Adaptive Server Enterprise が最初からこの余分なテキストを送らないようにするには、ct_options の CS_TEXTSIZE_OPT オプションを使用して、サーバのグローバル変数 @@*textsize* を設定します。

## タイムアウト

CS_TIMEOUT は、Client-Library がコマンドへのサーバ応答を待つ時間の秒単位での長さを指定します。

デフォルト・タイムアウト値は CS_NO_LIMIT で、タイムアウト時間が無限になります。負の値および 0 は、CS_TIMEOUT では使用できません。

### タイムアウト値の設定

アプリケーションが ct_connect を呼び出して接続をオープンする前後どちらでも、ct_config を呼び出してタイムアウト値を設定できます。ct_config の呼び出しは、オープンしているすべての接続にすぐに効果を及ぼします。

次のプログラム例は、60 秒のタイムアウト制限を設定します。

```
CS_INT timeval;
 timeval = 60;
 if (ct_config(ctx, CS_SET, CS_TIMEOUT,
            (CS_VOID *)&timeval,
```

```
                 CS_UNUSED, NULL)
         != CS_SUCCEED)
    {
      fprintf(stdout, "Can't config timeout. Exiting.");
      (void)ct_exit(ctx, CS_FORCE_EXIT);
      (void)cs_ctx_drop(ctx);
      exit(1);
    }
```

#### タイムアウト・エラーの処理

CS_TIMEOUT プロパティか CS_LOGIN_TIMEOUT プロパティまたはその両方を CS_NO_LIMIT 以外の値に設定した同期アプリケーションでは、タイムアウト・エラーが発生します。CS_LOGIN_TIMEOUT プロパティは、ログイン試行時にサーバの応答を読み込むときのタイムアウト時間を設定します。それに対して CS_TIMEOUT プロパティは、サーバ・コマンドの結果を読み込むときのタイムアウト時間を設定します。どちらの場合も、タイムアウト時にアプリケーションに返される Client-Library メッセージは同じです。CS_LOGIN_TIMEOUT プロパティの詳細については、「ログイン・タイムアウト」(253 ページ ) を参照してください。

---

**注意**　ct_con_props では、接続ごとに CS_TIMEOUT 値または CS_LOGIN_TIMEOUT 値を指定できます。

---

インライン・エラー処理を使用するアプリケーションは、CS_DIAG_TIMEOUT プロパティを設定して、タイムアウト時に Client-Library がアボートするかリトライするかを指定する必要があります。「診断タイムアウトの失敗」(245 ページ ) を参照してください。

コールバックを使用して Client-Library メッセージを処理するアプリケーションはタイムアウト・エラーを識別して、そのオペレーションをキャンセルするか、または別のタイムアウト時間にリトライすることができます。クライアント・メッセージ・コールバックには、タイムアウト・メッセージを処理するための次のオプションがあります。

- CS_FAIL を返してそのオペレーションをキャンセルし、接続を「dead」としてマーク付けします。これは、タイムアウトしたログインの試みをアボートする唯一の方法です。
- ( ログイン以外のタイムアウトの場合のみ ) ct_cancel (CS_CANCEL_ATTN) を呼び出して、処理中のコマンドをキャンセルしてから、CS_SUCCEED を返します。
- CS_SUCCEED を返して、別のタイムアウト時間にリトライします。

(CS_CLIENTMSG 構造体の *number* フィールドによって識別される ) エラー番号を 4 つのコンポーネントに分けて、そのエラー番号が次の特性と一致しているかどうかをチェックすることによって、タイムアウト・エラーを識別します。

- 重大度 － CS_SV_RETRY_FAIL
- 番号 － 63
- オリジン － 2
- レイヤ － 1

アプリケーションは CS_SEVERITY マクロ、CS_NUMBER マクロ、CS_ORIGIN マクロ、CS_LAYER マクロを使用して、エラー番号をコンポーネントに分けます。これらのマクロの詳細については、「Client-Library メッセージの番号」(89 ページ ) を参照してください。次に、タイムアウト・エラーのテスト例を示します。

コールバックは CS_LOGIN_STATUS 接続プロパティの値をチェックして、接続の確立中またはコマンドの処理中にタイムアウトが発生しているかどうかを調べます。このプロパティが CS_TRUE に設定されている場合、すでに接続が確立されて、コマンドの処理中にサーバはタイムアウトしています。

次のコード例は、タイムアウト・エラーを処理するクライアント・メッセージ・コールバックを定義します。

```
/*
 ** ERROR_SNOL(error_numb, severity, number, origin, layer)
 **
 **   Error comparison for Client-Library or CS-Library errors.
 **   Breaks down a message number and compares it to the given
 **   constants for severity, number, origin, and layer.
 **   Returns non-zero if the error number matches the 4
 **   constants.
 */
 #define ERROR_SNOL (e, s, n, o, l) ¥
   ( (CS_SEVERITY(e) == s) && (CS_NUMBER(e) == n) ¥
    && (CS_ORIGIN(e) == o) && (CS_LAYER(e) == l ) )

CS_RETCODE  client_msg_handler(cp, conn, emsgp)
 CS_CONTEXT      *cp;
 CS_CONNECTION   *conn;
 CS_CLIENTMSG    *emsgp;
 {
   CS_RETCODE    ret;
   CS_INT        status;
```

```
   ... code to print message details and handle any other
       errors besides timeout ...
 /*
   ** Is this a timeout error?
   */
   if (ERROR_SNOL(emsgp->msgnumber, CS_SV_RETRY_FAIL, 63, 2, 1))
   {
     /*
     ** Read from server timed out. Timeouts happen on synchronous
     ** connections only, and you must have set one or both of the
     ** following context properties to see them:
     **  CS_TIMEOUT for results timeouts
     **  CS_LOGIN_TIMEOUT for login-attempt timeouts
     **
     ** If we return CS_FAIL, the connection is marked as dead and
     ** unrecoverable. If we return CS_SUCCEED, the timeout
     ** continues for another quantum.
     **
     ** We kill the connection for login timeouts, and send a
     ** cancel for results timeouts. We determine which case we
     ** have through the CS_LOGIN_STATUS property.
     */
     status = 0;
     if (ct_con_props(conn, CS_GET, CS_LOGIN_STATUS,
                     (CS_VOID *)&status,
                      CS_UNUSED, NULL) != CS_SUCCEED)
     {
       fprintf(stdout, "ct_con_props() failed in error handler.");
       return CS_FAIL;
     }
   if (status)
     {
       /* Results timeout */
       fprintf(stdout, "Issuing a cancel on the query...¥n");
       (CS_VOID)ct_cancel(conn, (CS_COMMAND *)NULL,
                          CS_CANCEL_ATTN);
     }
   else
     {
       /* Login timeout */
       fprintf(stdout, "Aborting connection attempt...¥n");
       return CS_FAIL;
     }
   }
   return (CS_SUCCEED);
}
```

## トランザクション名

CS_TRANSACTION_NAME は、Open Server for CICS への接続に使用されるトランザクション名を定義します。

Open Server for CICS は、CICS のもとで実行中の実行可能イメージをトランザクション名を使用して識別します。詳細については、Open Server for CICS のマニュアルを参照してください。

Sybase Server アプリケーションのトランザクション名は、CS_TRANSACTION_NAME ではなく、トランザクションの開始にマークを付ける Transact-SQL の begin tran 文によって指定されます。『ASE リファレンス・マニュアル』を参照してください。

すべての Client-Library アプリケーションは、CS_TRANSACTION_NAME を設定できます。トランザクション名が必要でない場合、CS_TRANSACTION_NAME は無視されます。

## ユーザ割り付け関数

CS_USER_ALLOC は、システム割り込みレベルで実行している間、Client-Library がメモリ管理に使用する、ユーザ提供のメモリの割り付けルーチンを特定します。

CS_USER_ALLOC と CS_USER_FREE により、非同期アプリケーションは独自のメモリ管理を行うことができます。

ユーザ提供のメモリの割り付けルーチンは次のように定義してください。

```
void      *user_alloc(size)
size_t    size;
```

通常、Client-Library ルーチンは、malloc を呼び出すことによって、メモリ要件を満たします。しかし、malloc の実装のすべてが再入可能とはかぎらないため、割り込みレベルで呼び出される Client-Library が malloc を使用することは安全ではありません。このため、Client-Library がシグナル駆動型ネットワーク I/O を使用するシステム (UNIX システムなど ) では、完全非同期アプリケーションは、Client-Library にそのメモリ要件を満たす代替手段を提供する必要があります。

これは、スレッド駆動型ネットワーク I/O を使用しているプラットフォームや、完全非同期接続を使用しないアプリケーションの場合には必要ありません。使用しているプラットフォームのネットワーク I/O 方式の詳細については、『Open Client/Server プログラマーズ・ガイド補足』を参照してください。

Client-Library は、非同期アプリケーションが Client-Library のメモリ要件を満たせるようにするために、次の 2 とおりのメカニズムを提供します。

- アプリケーションは CS_MEM_POOL プロパティを使用して、メモリ要件に対して使用するメモリ・プールを Client-Library に提供できます。
- アプリケーションは、CS_USER_ALLOC および CS_USER_FREE プロパティを使用して、Client-Library が割り込みレベルで安全に呼び出すことができるメモリの割り付けと解放のルーチンをインストールできます。

完全非同期アプリケーションが Client-Library にメモリ要件を安全に満たす手段を提供できない場合、Client-Library の動作は予測不可能です。

Client-Library は、次の順で次のソースから必要なメモリを確保しようとします。

1. メモリ・プール
2. ユーザ提供の割り付けおよび解放ルーチン
3. システム・ルーチン

接続が必要なメモリを得ることができない場合、Client-Library は、接続に 「dead」とマーク付けします。

アプリケーションは、action に CS_SET を設定し、*buffer* に新しいルーチンのアドレスを設定して ct_config を呼び出すことで、ユーザ定義のメモリ・ルーチンを置き換えることができます。

アプリケーションでメモリ・ルーチンをクリアする方法には、次の 2 とおりがあります。

- action に CS_SET を、*buffer* に NULL を設定して、ct_config を呼び出す方法
- action に CS_CLEAR を設定して ct_config を呼び出す方法

## ユーザ解放関数

CS_USER_FREE は、Client-Library がシステムの割り込みレベルのメモリ管理に使用する、ユーザ提供のメモリ割り付け解除ルーチンを特定します。

CS_USER_ALLOC と CS_USER_FREE により、非同期アプリケーションは独自の割り込みレベル・メモリ管理を行うことができます。

ユーザ提供のメモリの割り付け解除ルーチンは次のように定義してください。

```
void      user_free(ptr)
void      *ptr;
```

「ユーザ割り付け関数」(271 ページ ) を参照してください。

## ユーザ・データ

CS_USERDATA プロパティは、ユーザ割り付けデータを定義します。このプロパティを使用すると、アプリケーションは、ユーザ・データを特定の接続またはコマンド構造体と関連付けることができます。

CS_USERDATA プロパティにはデフォルト値はありません。値が設定されないときにアプリケーションがこのプロパティを取得した場合、ct_con_props ルーチンまたは ct_cmd_props ルーチンは、*outlen* パラメータを 0 に設定して返ります。

コールバック・ルーチンとメインライン・アプリケーションが、グローバル変数を使用しないで情報を共有する必要がある場合に、CS_USERDATA が役立ちます。

アプリケーションが、CS_USERDATA を使用してデータを保管する場合、Client-Library は、データのポインタではなく、ct_con_props または ct_cmd_props; の *buffer* パラメータによって示される実際のデータを内部データ領域にコピーします。

CS_USERDATA プロパティの値は、アプリケーションが定義したデータのどれかになります。このプロパティを設定する場合、アプリケーションは (CS_VOID * にキャストされた ) データへのポインタを渡して、データの正確な長さをバイト単位で指定します。実際には大部分のアプリケーションは、アプリケーションが割り付けたデータ構造体のアドレスを CS_USERDATA としてインストールします。これにより、アプリケーションはデータへのポインタを CS_USERDATA として取得できます。アプリケーションはそのポインタを使用してデータを変更しますが、変更後にそのコンテキスト、接続、またはコマンド構造体の中のデータを再インストールする必要はありません。

アプリケーションは、cs_config を呼び出して、ユーザ・データをコンテキスト構造体と関連付けます。CS_USERDATA プロパティ値は、接続またはコマンドレベルで継承されません。

次に、CS_USERDATA プロパティを指定するコード例を示します。

```
CS_CHAR          set_charbuf[32];
CS_CHAR          get_charbuf[32];
```

```
    CS_CONNECTION   *con;
    CS_RETCODE      ret;
    CS_INT          outlen;
    CS_COMMAND      *set_cmd;
    CS_COMMAND      *get_cmd;

  /*
    ** Store a character string in the userdata field.
    ** Set the length field to one greater than the length
    ** of the string so that the null terminator will be
    ** stored as part of the user data. If the null
    ** terminator is not explicitly stored as part of the
    ** userdata, then the string will not be null-
    ** terminated when it is retrieved.
    */
    strcpy(set_charbuf, "some userdata");
    ret = ct_con_props(con, CS_SET, CS_USERDATA,
         set_charbuf, strlen(set_charbuf) + 1, NULL);
    if (ret != CS_SUCCEED)
    {
         error("ct_con_props() failed");
    }

  ret = ct_con_props(con, CS_GET, CS_USERDATA,
         get_charbuf, sizeof(get_charbuf), &outlen);
    if (ret != CS_SUCCEED)
    {
         error("ct_con_props() failed");
    }

  /*
    ** The next example stores a pointer to a CS_COMMAND
    ** structure in the connection's user data field.
    */
    ret = ct_con_props(con, CS_SET, CS_USERDATA,
         &set_cmd, sizeof(set_cmd), NULL);
    if (ret != CS_SUCCEED)
    {
         error("ct_con_props() failed");
    }
  ret = ct_con_props(con, CS_GET, CS_USERDATA,
         &get_cmd, sizeof(get_cmd), &outlen);
    if (ret != CS_SUCCEED)
    {
         error("ct_con_props() failed");
    }
```

## ユーザ名

CS_USERNAME は、サーバへのログイン時に接続が使用するユーザ・ログイン名を定義します。

アプリケーションでは、ネットワークベースのユーザ認証を要求しない場合に、CS_PASSWORD 接続プロパティの値をユーザのパスワードと一致するように設定する必要があります。「パスワード」(260 ページ ) を参照してください。

アプリケーションが CS_SEC_NETWORKAUTH プロパティを使用してネットワークベースの認証を要求した場合、ユーザは CS_USERNAME と同じ名前を使用して、接続のネットワーク・セキュリティ・メカニズムにすでにログインしている必要があります。この場合、CS_PASSWORD プロパティは無視されます。

アプリケーションは CS_SEC_NETWORKAUTH プロパティを設定して、ネットワークベースの認証を要求します。「ログイン認証サービスの要求」(295 ページ ) を参照してください。

## Client-Library のバージョン文字列

CS_VER_STRING は、アプリケーションが使用している Client-Library の実際のバージョンを表す文字列を定義します。このプロパティは、取得のみ可能です。

新しいバージョンの Client-Library は以前のバージョンの動作をエミュレートするため、CS_VER_STRING と CS_VERSION は異なるバージョン・レベルを示します。

CS_VER_STRING は、使用中の Client-Library の実際のバージョンを表します。CS_VERSION は、アプリケーションが ct_init を使用して要求した Client-Library 動作のバージョンを表します。

## Client-Library のバージョン

CS_VERSION プロパティは、アプリケーションが ct_init を呼び出して要求した Client-Library の動作のバージョンを表します。このプロパティ値は取得のみ可能です。

CS_VERSION の有効な値は、次のとおりです。

- CS_VERSION_100 ( バージョン 10.0 を示す )
- CS_VERSION_110 ( バージョン 11.0 を示す )
- CS_VERSION_120 ( バージョン 12.0 を示す )

- CS_VERSION_125 ( バージョン 12.5 を示す )
- CS_VERSION_150 ( バージョン 15.0 を示す )
- CS_VERSION_155 ( バージョン 15.5 を示す )
- CS_VERSION_157 ( バージョン 15.7 を示す )

コンテキスト内で割り付けられた接続では、その親コンテキストの CS_VERSION レベルに基づくデフォルトの CS_TDS_VERSION 値が使用されます。「TDS バージョン」(266 ページ ) を参照してください。

Client-Library と CS-Library にはともに、CS_VERSION プロパティがあります。ct_config は、Client-Library の CS_VERSION 値を返します。cs_config は、CS-Library の CS_VERSION 値を返します。

# レジスタード・プロシージャ

レジスタード・プロシージャは、動作している Open Server アプリケーションで定義されインストールされるプロシージャで、Adaptive Server Enterprise の機能を拡張します。

Client-Library アプリケーションに対して、レジスタード・プロシージャは、アプリケーション間での通信および同期の手段を提供します。これは、Open Server に接続されている Client-Library アプリケーションが、レジスタード・プロシージャの実行を監視するからです。レジスタード・プロシージャが実行されると、監視しているアプリケーションは、プロシージャ名と呼び出し時の引数が含まれるノーティフィケーション ( 通知 ) を受け取ります。

たとえば、次のように想定してみてください。

- stockprice は、株価をモニタするリアルタイムの Client-Library アプリケーションです。
- price_change は、stockprice によって Open Server 内に作成されたレジスタード・プロシージャであり、price_change は、株式名および株価変動をパラメータとして取ります。
- sellstock は、株式を売りに出すアプリケーションで、price_change を実行するとノーティフィケーションを受信することを要求します。

モニタリング・アプリケーションである stockprice は、Extravagant Auto Parts の株価が $1.10 上昇したことを検知すると、「Extravagant Auto Parts」と「+1.10」をパラメータとして渡して price_change を実行します。

price_change を実行するときに、Open Server は、プロシージャ名 (price_change) と、そのプロシージャに渡される引数 (「Extravagant Auto Parts」と「+1.10」) が含まれたノーティフィケーションを sellstock に送信します。sellstock は、そのノーティフィケーションの情報を使用して、Extravagant Auto Parts 社の株式を売却するかどうかを決定します。

price_change は、stockprice アプリケーションと sellstock アプリケーションが通信を行うための手段です。

通信手段としてのレジスタード・プロシージャには、次の利点があります。

- レジスタード・プロシージャを実行するための一度の呼び出しにより、プロシージャの実行を多くのアプリケーションに通知します。プロシージャを実行しているアプリケーションは、クライアント数や、情報を要求していたクライアントについて知る必要はありません。
- レジスタード・プロシージャによる通信メカニズムは、サーバベースです。Open Server は、接続アドレスの集中レポジトリとなります。これにより、クライアント・アプリケーションは互いに直接接続することなく通信します。各クライアントは Open Server に接続するだけです。

Client-Library アプリケーションは、次のような目的で、Open Server システム・レジスタード・プロシージャへのリモート・プロシージャ・コールを行います。

- Open Server 上にレジスタード・プロシージャを作成するため。

---

**注意**　Client-Library アプリケーションは、実行可能な文を持たないレジスタード・プロシージャだけを作成します。これらのプログラム本体を持たないプロシージャは、主に、通信および同期のために有効です。

---

- レジスタード・プロシージャを削除するため。
- Open Server で定義されたレジスタード・プロシージャをすべてリストするため。
- 特定のレジスタード・プロシージャの実行時に、通知を受けることを要求するため。
- クライアント接続が待っているレジスタード・プロシージャ・ノーティフィケーションをすべてリストするため。
- レジスタード・プロシージャを実行するため。

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。

アプリケーションは、次のような目的で Client-Library ルーチンを呼び出します。

- 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ときに呼び出されるユーザ提供のノーティフィケーション・コールバック・ルーチンをインストールするため。
- 必要に応じてネットワークをポーリングして、待ち状態のレジスタード・プロシージャ・ノーティフィケーションがあるかどうかを調べるため。

## Client-Library のノーティフィケーションの受信

Client-Library は、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ときに、アプリケーションのノーティフィケーション・コールバック・ルーチンを呼び出します。ホスト・クライアント・プラットフォームに応じて、Client-Library がノーティフィケーション・コールバックを呼び出すために、アプリケーションが (ct_poll ルーチンを呼び出して ) ネットワークをポーリングする必要がある場合もあります。「ノーティフィケーションの非同期受信」(279 ページ ) を参照してください。

レジスタード・プロシージャ名は、ノーティフィケーション・コールバック・ルーチンの第 2 パラメータとして取得できます。

レジスタード・プロシージャが呼び出されたときの引数は、パラメータ結果セットとしてノーティフィケーション・コールバック内で取得できます。これらの引数を取得するために、アプリケーションは次のようにします。

- ct_con_props(CS_NOTIF_CMD) を呼び出して、パラメータ結果セットを持つコマンド構造体へのポインタを取得します。
- ct_res_info(CS_NUMDATA)、ct_describe、ct_bind、ct_fetch、ct_get_data を呼び出して、そのパラメータの記述、バインド、およびフェッチを行います。

「ノーティフィケーション・コールバック」(53 ページ ) を参照してください。

## ノーティフィケーションの非同期受信

アプリケーションでのノーティフィケーション・イベントの受信は、CS_ASYNC_NOTIFS プロパティと、クライアント・プラットフォームでサポートされているネットワーク I/O 方式によって異なります。

CS_ASYNC_NOTIFS プロパティは、接続がノーティフィケーションを非同期で受信するかどうかを指定します。「非同期ノーティフィケーション」(237 ページ ) を参照してください。

Open Server への接続にノーティフィケーション以外のアクティビティがほとんどないかまたはまったくない場合は、CS_ASYNC_NOTIFS プロパティを CS_TRUE に設定して、非同期ノーティフィケーションを有効にしてください。このプロパティはデフォルトで CS_FALSE に設定されます。このデフォルト値は、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ために、その接続上でサーバと対話している必要があることを意味します ( これによって Client-Library はネットワークから読み込むようになります )。

---

**注意**　接続がレジスタード・プロシージャ・ノーティフィケーションの受信だけに使用される場合、接続がポーリングされるときでも、その接続に対して非同期ノーティフィケーションを有効にしてください。アイドルな接続では、CS_ASYNC_NOTIFS プロパティが CS_TRUE でなければ、ct_poll ルーチンはノーティフィケーション・コールバックをトリガしません。デフォルト設定は CS_FALSE です。

---

### ノーティフィケーションの検出

シグナル駆動型またはスレッド駆動型の I/O をサポートするプラットフォームで、非同期ノーティフィケーションが有効になっている場合、接続でノーティフィケーションが着信すると、Client-Library がその接続のノーティフィケーション・コールバックを自動的に呼び出します。

その他のプラットフォームでは、接続がアクティブではない場合に、アプリケーションは ct_poll ルーチンを呼び出してその接続をポーリングする必要があります。ct_poll でノーティフィケーションがレポートされるようにするには、CS_ASYNC_NOTIFS を CS_TRUE に設定します。

# 結果

Client-Library コマンドがサーバ上で実行されると、さまざまなタイプの結果が生成されて、コマンドを送信したアプリケーションに返されます。これらの結果には、次のタイプがあります。

- 通常ローの結果
- カーソル・ロー結果
- パラメータ結果
- ストアド・プロシージャ・リターン・ステータス結果
- 計算ロー結果
- メッセージ結果
- 記述結果
- フォーマット結果

結果は、「**結果セット**」の形式でアプリケーションへ返されます。結果セットには、ただ 1 つのタイプの結果データだけが含まれています。通常ローおよびカーソル・ロー結果セットは、複数のデータ・ローを含むことが可能ですが、その他のタイプの結果セットは 1 つのデータ・ローしか含みません。

アプリケーションは、ct_results を呼び出すことによって結果を処理します。また、ct_results は、**result_type* の設定でその結果タイプを示します。

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。論理コマンドは一般に、ct_command、ct_dynamic、または ct_cursor で定義される任意の Client-Library コマンドであると考えられます。このルールの例外は、「ct_results と論理コマンド」(625 ページ ) に記述されています。

コマンドの中には、Transact-SQL の update 文を含む言語コマンドのように、結果を生成しないものもあります。ct_results は、**result_type* を CS_CMD_SUCCEED または CS_CMD_FAIL に設定して、結果を返さないコマンドのステータスを示します。

## 通常ローの結果

通常ロー結果セットは、サーバ上で Transact-SQL の 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。

## カーソル・ロー結果

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 の fetch 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。fetch 言語文からのカーソル・ローは CS_ROW_RESULT 結果セットとして返されます。

---

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。

カーソル・ロー結果セットは、アプリケーションが ct_cursor を使用して、カーソル・ローをフェッチする間に基本テーブルを更新する点が通常ロー結果セットとは異なります。これは、通常ローでは不可能です。

## パラメータ結果

パラメータ結果セットには、パラメータの 1 つの「ロー」が含まれます。いくつかのタイプのデータは、次のものを含んでいるパラメータ結果セットとして返されます。

- メッセージ・パラメータ － メッセージ結果セット (CS_MSG_RESULT) には、関連するパラメータが含まれます。メッセージ・パラメータは、CS_PARAM_RESULT 結果セットとして CS_MSG_RESULT 結果タイプの直後に指定されます。
- RPC リターン・パラメータ － 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・データを返します。このデータは、プロシージャ・コードによって設定される、プロシージャのパラメータの新しい値が含まれる CS_PARAM_RESULT 結果セットです。

拡張エラー・データとレジスタード・プロシージャ・ノーティフィケーション・パラメータも、パラメータ結果セットとして返されますが、アプリケーションは、これらのタイプのデータを処理する ct_results を呼び出さないため、CS_PARAM_RESULT の結果タイプが認識されません。その代わりに、パラメータ・ローは、アプリケーションが、そのデータを含んでいる CS_COMMAND 構造体を取得した後で、簡単にフェッチされます。

拡張エラー・データについては、「拡張エラー・データ」(143 ページ ) を参照してください。 レジスタード・プロシージャ・ノーティフィケーション・パラメータについては、「レジスタード・プロシージャ」(276 ページ ) を参照してください。

## ストアド・プロシージャ・リターン・ステータス結果

ステータス結果セットは、1 つの値 ( リターン・ステータス ) を含む 1 つのローで構成されています。

Adaptive Server Enterprise で実行されるすべてのストアド・プロシージャは、ステータス番号を返します。ストアド・プロシージャは通常は 0 を返して、正常に終了したことを示します。Adaptive Server Enterprise のデフォルト・リターン・ステータス番号のリストについては、『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。

リターン・ステータス番号はストアド・プロシージャの機能の 1 つであるため、RPC コマンド、または execute 文を含む言語コマンドだけが、リターン・ステータスを生成します。

## 計算ロー結果

計算ロー結果セットは、計算ローを生成した compute 句にリストされているカラム数と同数のカラムを持つ、1 つの表形式データ・ローを含んでいます。

詳細については、『ASE リファレンス・マニュアル』の compute 句の説明を参照してください。

## メッセージ結果

メッセージ結果セットには、実際にはデータは何も含まれていません。ただし、メッセージには ID があります。アプリケーションは、メッセージの ID を取得するために、ct_results が CS_MSG_RESULT という *result_type* を返した後に、ct_res_info を呼び出します。

パラメータは、メッセージと関連付けられている場合、メッセージ結果セットのすぐ後に、別のパラメータ結果セットとして返されます。

## 記述結果

記述結果セットには、フェッチ可能なデータは含まれていませんが、動的 SQL の入力コマンドの記述または出力コマンドの記述結果として返される記述情報が存在することを示します。

アプリケーションは、次のいずれかの方法によって、この記述情報を取得します。

- ct_res_info を呼び出して項目数を取得し、ct_describe を呼び出して各項目の記述を取得します。
- ct_dyndesc を数回呼び出して、項目数と各項目の記述を取得します。
- ct_res_info を呼び出して項目数を取得し、ct_dynsqlda を一度呼び出して項目の記述を取得します。

『Open Client Client-Library/C プログラマーズ・ガイド』の「第 8 章 動的 SQL コマンドの使い方」を参照してください。

## フォーマット結果

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、フェッチ可能なデータは含まれていませんが、フォーマット結果セットはむしろ、関連した通常ローおよび計算ロー結果セットのフォーマット情報を利用できることを示します。

コマンドのフォーマット情報はすべて、どんなデータよりも先に返されます。つまり、あるコマンドのロー・フォーマットおよび計算フォーマット結果セットは、そのコマンドが生成する通常ローおよび計算ロー結果セットより先に返されます。

フォーマット情報は、主に、ゲートウェイ・アプリケーションで有効です。そして、ゲートウェイ・アプリケーションは、Adaptive Server Enterprise から得た結果を外部のクライアントへ送る前に再パッケージする必要があります。

一般に、ゲートウェイ・アプリケーションはフォーマット結果セットを一度に 1 カラムずつ処理し、ct_describe と ct_compute_info を呼び出してカラムのフォーマット情報を取得し、Server-Library ルーチンを使用してフォーマット情報を送信します。

接続は、その CS_EXPOSE_FMTS プロパティが CS_TRUE に設定されている場合にかぎり、フォーマット結果を受信します。

## 結果処理のプログラム構造体

次のコード例は、一般的なアプリケーションが、どのようにさまざまなタイプの結果データを処理するかを示しています。

```
while ct_results returns CS_SUCCEED
     case CS_ROW_RESULT
          ct_res_info 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
     case CS_CURSOR_RESULT
          ct_res_info 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
```

**(while ct_scroll_fetch returns CS_SUCCEED or CS_CURSOR_BEFORE_FIRST or CS_CURSOR_AFTER_LAST for scrollable cursors)**

```
                    process the row
                    /*
                    ** Nested cursor commands are legal
                    ** here.
                    */
               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
case CS_PARAM_RESULT
    ct_res_info 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
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
case CS_COMPUTE_RESULT
    (optional:ct_compute_info to get bylist
        length, bylist, or compute row id)
    ct_res_info 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 (or ct_scroll_fetch for scrollable cursors)
        final return code
        case CS_END_DATA (or CS_SCROLL_CURSOR_ENDS for scrollable
        cursors)...
        case CS_CANCELED...
        case CS_FAIL...
    end switch
```

```
        end case
        case CS_MSG_RESULT
            ct_res_info to get the message id
            code to handle the message
        end case
        case CS_DESCRIBE_RESULT
            ct_res_info to get the number of columns
            for each column:
                ct_describe to get a
                    description
            end for
        end case
        case CS_ROWFMT_RESULT
            ct_res_info 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
        case CS_CMD_DONE
            indicates a command's results are completely
                processed
        end case
        case CS_CMD_SUCCEED
            indicates the success of a command that
                returns no results
        end case
        case CS_CMD_FAIL
            indicates a command failed
        end case
    end while
    switch on ct_results' final return code
        case CS_END_RESULTS
```

```
            indicates no more results
        end case
        case CS_CANCELED
            indicates results were canceled
        end case
        case CS_FAIL
            indicates ct_results failed
        end case
    end switch
```

## 項目の値の取得

結果セットの処理時にアプリケーションが結果項目の値を取得する方法は、次のように 4 通りあります。

- ct_bind ルーチンを呼び出して、結果項目をプログラム変数と対応付けます。プログラムが ct_fetch を呼び出して結果ローをフェッチする場合、項目の値がフェッチ先の変数のフォーマットに自動変換され、その結果はバインドされた変換先変数に置かれます。大部分のアプリケーションは、大きな text 値または image 値を除くすべての結果項目にこの方法を使用します。「text および image データの処理」(328 ページ ) を参照してください。
- ct_get_data ルーチンを呼び出して、結果項目の値をまとまりで取得します。ct_fetch を呼び出してローをフェッチした後に、アプリケーションはループ内で ct_get_data を呼び出します。ct_get_data を呼び出すたびに、結果項目の値のまとまりが 1 つずつ取得されます。大部分のアプリケーションでは、ct_get_data ルーチンを呼び出すのは、大きな text 値または image 値の取得を行う場合のみです。
- ct_dyndesc(CS_USE_DESC) を呼び出して、動的記述子を結果セットと対応付けます。動的記述子が結果セットと対応付けられると、アプリケーションは繰り返し ct_fetch ルーチンを呼び出して、それぞれのローをフェッチします。また各ローに対しては、結果項目ごとに一度 ct_dyndesc ルーチンを呼び出します。一般的なアプリケーションは、プリコンパイラのサポートを目的とした ct_dyndesc ルーチンを使用しません。
- ct_dynsqlda (CS_USE_DESC) を呼び出して、アプリケーションが管理する SQLDA 構造体を結果カラムと関連付けることができます。アプリケーションは、ct_dynsqlda ルーチンを一度呼び出して、SQLDA 構造体によって示された値バッファにすべての結果カラムをバインドします。その後 ct_fetch を呼び出すと、値バッファにカラム値が保管されます。一般的なアプリケーションは、プリコンパイラのサポートを目的とした ct_dynsqlda ルーチンを使用しません。

## バッチ処理での結果バインドの保持

バッチ処理アプリケーションは、同じサーバ・コマンドを何度も再送信することがあります。アプリケーションは、その前のコマンド実行の結果処理後すぐに ct_send ルーチンを呼び出して、そのコマンドを再送信します。「コマンドの再送信」(643 ページ ) を参照してください。

コマンドを再送信するバッチ処理アプリケーションでは、CS_STICKY_BINDS コマンド・プロパティを設定すると便利な場合があります。このプロパティが CS_TRUE ( デフォルトは CS_FALSE) に設定されている場合、コマンドが再送信されるときに、Client-Library は結果バインドを再使用します。これによって、アプリケーションでの ct_bind 呼び出しの重複がなくなります。

以下を参照してください。

- CS_STICKY_BINDS プロパティの詳細については、「継続結果バインド」(261 ページ )
- 「ct_bind 」(371 ページ ) のリファレンス・ページ

## 可変長データの複数のローを配列に読み込む

可変長データ (VARCHAR または VARBINARY) の複数のローが選択されてバッファに読み込まれると、前のアイテムの長さが *datafmt->maxlength* のバイト数より短い場合でも、新しいアイテムはそれぞれ *datafmt->maxlength* の倍数のインデックスから開始されます。これを次のコード例で示します。

```
/* This example demonstrates selecting multiple rows of
variable-length data into a buffer. In this case, the
first row to be returned will have one column with the
value "first string" and a second row with a column with
the value "second string".*/
datafmt.count = 2;
datafmt.maxlength = 25;
retcode = ct_results(cmd, &restype);
 if (retcode != CS_SUCCEED)
{
       /* error handling code deleted */
       . . .
}
if (restype == CS_ROW_RESULT)
{
       retcode = ct_bind(cmd, 1, datafmt, buffer,
```

```
CS_NULL, CS_NULL);
        if (retcode != CS_SUCCEED)
       {
          /* error handling code deleted */
          . . .
       }
       retcode = ct_fetch(cmd, CS_UNUSED, CS_UNUSED,
          &nrows);
        if (retcode != CS_SUCCEED)
       {
          /* error handling code deleted */
          . . .
       }

/* At this point, the string "first string" begins at
buffer[0] and the string "second string" begins at
buffer[25], even though the first data item was less
than 25 characters long.*/
}
```

# セキュリティ機能

Client-Library は、3 つのカテゴリーのセキュリティ関連機能を提供します。

- ネットワークベースのセキュリティ ー Client-Library アプリケーションと Server-Library アプリケーションは、DCE や Microsoft LAN Manager などのネットワーク・システム・ソフトウェアが提供するセキュリティ・サービスと統合できます。この機能は、サービスの中でも特に ( ユーザがネットワーク・ユーザ名とパスワードを使用して Sybase サーバに接続する ) 統一化されたログインと、( クライアントとサーバの間のすべての通信の暗号化などの ) パケットごとのセキュリティ・サービスを提供しています。

  この機能を使用するには、Sybase がサポートしている個別のネットワーク・セキュリティ・ソフトウェアと、そのソフトウェア用の Sybase 提供セキュリティ・ドライバが必要です。

- Secure Socket Layer (SSL) ネットワークベースのセキュリティ ー バージョン 12.5 以降の Client-Library アプリケーションと Server-Library アプリケーションには、セッションベースのセキュリティである SSL を有効化するネットワークライブラリ・ドライバが組み込まれています。

SSL は、クライアントからサーバ、およびサーバからサーバへワイヤまたはソケット・レベルで暗号化されたデータを送信する業界標準です。クライアントは、サポートされている SSL オプションとともに接続要求をサーバに送信します。サーバからの応答時には、サーバの身元を証明するサーバ証明書とともに、サポートされている CipherSuite のリストも返されます。クライアントとサーバが CipherSuite について合意すると、SSL 対応のセッションが開始され、すべての送信データがセッションベースの暗号化によって保護されます。

- Sybase のセキュリティ機能 － パスワードの暗号化やチャレンジ／応答セキュリティ・ハンドシェイクなどがあります。

  Client-Library は、アプリケーションの要求に応じてユーザのパスワードを暗号化します。パスワードは、ハンドシェイク・プロトコルを使用して暗号化されます。このハンドシェイク・プロトコルでは、サーバが暗号化キーを送信し、クライアントがそのキーを使用してユーザのパスワードを暗号化します。

  チャレンジ／応答ハンドシェイクを使用すると、アプリケーションは、接続時にサーバがクライアントにチャレンジするセキュリティ方式を実装できます。この方式では、サーバは、チャレンジに対して予期された応答ができないクライアントからの接続を拒絶します。

  これらの機能は TDS プロトコルの一部であり、外部ソフトウェアを必要としません。Adaptive Server Enterprise および Open Server では、これらの機能をサポートしています。

## ネットワークベースのセキュリティ

分散クライアント／サーバ・コンピューティング環境では、ローカル・システムの場合よりもセキュリティを考慮しています。ユーザが見えないうえに、データがシステムからパブリック・データ・ネットワーク全体に渡って移動しているので、侵入者が機密データを見たり不正な変更を加えたりする可能性があります。セキュリティ・サービスによって、クライアントおよびサーバ・アプリケーションは、安全な接続を作成することができます。

ネットワークベースのセキュリティでは、サードパーティ提供のセキュリティ・ソフトウェアを活用して、ネットワーク・ユーザの認証を行ったり、ネットワーク上で転送されるデータを保護したりします。

## セキュリティ・メカニズムとセキュリティ・ドライバ

Sybase では「**セキュリティ・メカニズム**」を、接続時にセキュリティ・サービスを提供する外部ソフトウェアと定義しています。たとえば、次のようなセキュリティ・メカニズムを Client-Library 接続上で使用できます。

- DCE セキュリティ・サーバおよびセキュリティ・クライアント。DCE セル内のクライアントとサーバにセキュリティ・サービスを提供します。
- CyberSafe Kerberos。Windows 上のクライアント、UNIX 上のクライアント、UNIX 上のサーバにセキュリティ・サービスを提供します。
- Windows NT LAN Manager の SSPI (Security Services Provider Interface)。Windows 上のサーバとクライアントにセキュリティ・サービスを提供します。

Sybase ではセキュリティ・ドライバを提供します。これで、Client-Library アプリケーションと Server-Library アプリケーションは、インストールされたネットワーク・セキュリティ・システムを活用できます。セキュリティ・ドライバを使用することによって、Client-Library と Server-Library は、さまざまなネットワーク・セキュリティ・システムで動作する安全なアプリケーションを実装するための移植性の高いインタフェースを提供します。

接続上でセキュリティ・メカニズムを使用するには、以下の条件を満たす必要があります。

- クライアントとサーバは、互換性のあるセキュリティ・ドライバを使用する。たとえば、Windows NT 上で稼働するサーバが NT 用の Microsoft SSPI ドライバを使用する場合は、Windows 95 クライアント・アプリケーションは Windows 95 用の Microsoft SSPI ドライバを使用する。
- クライアントは、接続プロパティを設定してからサーバに接続することによって、サービスを要求する。
- 基本となるセキュリティ・メカニズムは、要求されたサービスをサポートする。

## ネットワーク・セキュリティ・メカニズムの選択

CS_SEC_MECHANISM 接続プロパティの値は、接続を確立するために使用するセキュリティ・メカニズムの名前を決定します。このプロパティのデフォルトは、使用しているシステムの Sybase セキュリティ・ドライバの設定内容によって決まります。

Client-Library はドライバ設定ファイルを使用して、セキュリティ・メカニズム名をセキュリティ・ドライバ・ファイル名にマップします。大部分のプラットフォームでは、このファイルは *libtcl.cfg* という名前です。ドライバ設定ファイルの詳細については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

### デフォルト・セキュリティ・メカニズムの決定

デフォルト・セキュリティ・メカニズムの名前は、ドライバ設定ファイル *libtcl.cfg* の [SECURITY] セクション内の最初のエントリに対応しています。このセクションには、次のような形式のエントリがあります。

```
[SECURITY]
 mechanism_name = driver_file_name init_string
 mechanism_name = driver_file_name init_string
```

この場合、*mechanism_name* は CS_SEC_MECHANISM プロパティに指定可能な値、*driver_file_name* はドライバのファイル名であり、*init_string* はドライバの起動時の設定を指定します。

ドライバ設定ファイルがシステム上にない場合や、ファイルに [SECURITY] セクションがない場合、CS_SEC_MECH プロパティはデフォルトで NULL に設定されます。

使用しているシステムでのドライバ設定ファイルの詳細については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

### デフォルト・セキュリティ・ドライバのロード

Client-Library は、アプリケーションからドライバ名を指定しないでドライバを要求されると、デフォルト・セキュリティ・ドライバがあれば、必要に応じてそのドライバをロードします。セキュリティ・ドライバがロードされていない場合は、*action* パラメータに CS_SET または CS_SUPPORTED を、*property* パラメータには下記のいずれかの値を指定して ct_con_props または ct_config を呼び出すと、デフォルト・ドライバがロードされます。

- CS_SEC_CHANBIND (CS_TRUE に設定する場合のみ )

- CS_SEC_CONFIDECTIALITY (CS_TRUE に設定する場合のみ )
- CS_SEC_CREDTIMEOUT
- CS_SEC_DATAORIGIN (CS_TRUE に設定する場合のみ )
- CS_SEC_DELEGATION (CS_TRUE に設定する場合のみ )
- CS_SEC_DETECTREPLAY (CS_TRUE に設定する場合のみ )
- CS_SEC_DETECTSEQ (CS_TRUE に設定する場合のみ )
- CS_SEC_INTEGRITY (CS_TRUE に設定する場合のみ )
- CS_SEC_KEYTAB
- CS_SEC_MECHANISM (CS_CLEAR に設定した場合は、デフォルト・ドライバがロードされます。CS_GET に設定した場合は、ドライバがまだロードされていなければデフォルト・ドライバがロードされます。CS_SET に設定した場合は、要求したドライバがロードされます。)
- CS_SEC_MUTUALAUTH (CS_TRUE に設定する場合のみ )
- CS_SEC_NETWORKAUTH (CS_TRUE に設定する場合のみ )
- CS_SEC_SESSTIMEOUT

### グローバル・メカニズム名

ドライバ設定ファイル内のセキュリティ・メカニズム名はローカルな名前であり、システムによって異なります。クライアントとサーバがともに接続のセキュリティ・メカニズムの ID を設定するには、セキュリティ・メカニズムに不変のグローバル名が必要です。

Client-Library は、CS_SEC_MECHANISM プロパティを設定したり、デフォルト・セキュリティ・ドライバをロードしたりする場合、設定ファイル ( グローバル・オブジェクト識別子ファイル ) を読み込み、ローカル・セキュリティ・メカニズム名をオブジェクト識別子 (OID) の文字列にマップします。大部分のプラットフォームでは、このファイルは *objectid.dat* という名前です。Client-Library は、セクション [SECMECH] にあるセキュリティ・メカニズムの OID を検索します。このセクション内のエントリは、次のような形式になっています。

```
[SECMECH]
 mechanism_oid = local_name1, local_name2, ...
```

ここで、*mechanism_oid* は、セキュリティ・メカニズムをグローバルに識別する OID 文字列、*local_name1*、*local_name2* などは、*libtcl.cfg* ファイル内で指定されているローカル・セキュリティ・プロバイダ名です。詳細については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

## ネットワーク・セキュリティ・サービスの要求

それぞれのセキュリティ・メカニズムは、1 組のセキュリティ・サービスを提供します。それぞれのセキュリティ・サービスには、セキュリティについて考慮する点がいくつかあります。Client-Library アプリケーションでは、要求されたサービスはコンテキスト・プロパティまたは接続プロパティに対応しています。

セキュリティ・メカニズムがすべてのセキュリティ・サービスをサポートしているわけではありません。あるサービスが現在のセキュリティ・メカニズムでサポートされているかどうかを判断するには、*action* パラメータに CS_SUPPORTED を、*buffer* パラメータに CS_BOOL 変数のアドレスを、*property* パラメータにはセキュリティ・サービスを表す記号プロパティ定数を設定して、アプリケーションから ct_config または ct_con_props を呼び出します。サービスがサポートされている場合は、**buffer* パラメータが CS_TRUE に設定されます。アプリケーションから現在のセキュリティ・メカニズムではサポートされていないサービスを要求すると、ct_config も ct_con_props も失敗します。

ネットワーク・セキュリティ・サービスは、次の 2 つに分類できます。

- ログイン認証サービス。このサービスを使用すると、アプリケーションは安全な接続を確立できます。
- パケットごとのセキュリティ・サービス。このサービスは、確立された接続上で転送されるデータを保護します。

### ログイン認証サービスの要求

基本的なセキュリティ・サービスは「**ログイン認証**」で、ユーザが本人であることを確認するものです。ログイン認証には、ユーザ名とパスワードが必要です。ユーザはユーザ名によってユーザ自身を識別し、そのユーザ本人であることの証拠としてパスワードを入力します。

Sybase アプリケーションでは、クライアントとサーバの間の接続ごとに対応するユーザ名が 1 つずつあります。アプリケーションがセキュリティ・メカニズムを使用する場合、Sybase はそのメカニズムを使用して、接続が確立されるときにこのユーザ名を認証します。このサービスの利点は、ユーザ名とパスワードを個々のサーバのシステム・カタログ内ではなく中央レポジトリ内で管理できることです。

アプリケーションがネットワークベースの認証を使用してサーバへの接続を要求する場合、Client-Library は接続のセキュリティ・メカニズムに問い合わせて、指定されたユーザ名がアプリケーションを実行している認証ユーザを示すことを確認します。つまり、ユーザはパスワードを指定しなくてもサーバに接続できます。その代わり、ユーザはネットワーク・セキュリティ・システムに対して本人であることを証明してから、接続しようとします。接続時に、Client-Library はセキュリティ・メカニズムから「**クレデンシャル・トークン**」を取得して、このトークンをパスワードの代わりにサーバに送信します。その後サーバはそのトークンを再度セキュリティ・メカニズムに渡して、そのユーザ名が認証されていることを確認します。

次に示す接続プロパティは、ログイン認証と関係があります。これらのプロパティを有効にするには、接続が確立される前に設定する必要があります。接続レベルでは、接続がオープンしている場合、次に示すすべてのプロパティが取得のみ可能です。

- CS_USERNAME プロパティは、接続するユーザの名前を指定します。アプリケーションがネットワークベースの認証を要求した場合、ユーザはネットワーク・セキュリティ・システムにログインしている必要があります。そうでない場合は、CS_PASSWORD プロパティをユーザのサーバ・パスワードに設定してください。
- CS_SEC_NETWORKAUTH プロパティを使用すると、ネットワークベースの認証が有効になります。このプロパティのデフォルトは CS_FALSE です。このデフォルト値は、ネットワークベースの認証が無効であることを意味します。
- CS_SEC_CREDTIMEOUT プロパティと CS_SEC_SESSTIMEOUT プロパティを使用すると、アプリケーションは、ユーザのネットワーク・クレデンシャルまたはセキュリティ・セッションの有効期限を指定したり、期限が切れているかどうかをチェックしたりできます。どちらのプロパティも、接続でネットワークベースの認証が有効になっている場合にのみ適用されます。

クレデンシャルのタイムアウト時間は、ユーザがクレデンシャルを取得するとき ( つまり、ユーザがネットワークにログインするとき ) から始まります。ネットワーク・セキュリティ・システムには、ユーザのクレデンシャルのタイムアウト値を管理者が指定できるものがあります。クレデンシャルの有効期限が切れると、クレデンシャルは無効になります。また、クレデンシャルのタイムアウト時間をアプリケーションが設定できるシステムもあります。

セッションのタイムアウト時間は、接続がオープンしたときから始まります。ネットワーク・セキュリティ・システムには、すべてのセキュリティ・セッションのタイムアウト値を管理者が指定できるものがあります。また、セッションのタイムアウト値をアプリケーションが設定できるシステムもあります。

表 2-33 に、クレデンシャル・タイムアウト・プロパティとセッション・タイムアウト・プロパティに設定できる値を示します。

**表 2-33：CS_SEC_SESSTIMEOUT と CS_SEC_CREDTIMEOUT の値**

| 値 | 意味 |
| --- | --- |
| 正の整数 | クレデンシャルの有効期限が切れるまでの残り時間 ( 秒数 )。 |
| 0 | クレデンシャルの有効期限が切れた。 |
| CS_UNEXPIRED | クレデンシャルは有効である。有効期限が切れるまでの残り時間は不定である。 |
| CS_NO_LIMIT | クレデンシャルに有効期限はない。 |

クレデンシャル・タイムアウトまたはセッション・タイムアウトをサポートしていないセキュリティ・メカニズムもあります。どちらかのタイプがサポートされていない場合、取得されるタイムアウト値は常に CS_NO_LIMIT です。セキュリティ・メカニズムには、タイムアウトをサポートしていてもタイムアウト値をアプリケーションにレポートしないものがあります。こうしたセキュリティ・メカニズムの場合は、取得されるタイムアウト値が CS_UNEXPIRED または 0 です。

アプリケーションは対応するプロパティを正の整数または CS_NO_LIMIT に設定して、別のクレデンシャル・タイムアウト値またはセッション・タイムアウト値を要求できます。ただし、セキュリティ・システムの管理設定によって、アプリケーションが要求する値が制限されます。たとえば、すべてのセッションが 10 分後にタイムアウトするようにシステムが設定されている場合は、アプリケーションが 20 分 (1,200 秒 ) のセッション・タイムアウト値を要求しても無効になります。

特定のクレデンシャル・タイムアウト値またはセッション・タイムアウト値に対するアプリケーションの要求が許可されない場合でも、エラーは表示されません。接続のセキュリティ・メカニズムがクレデンシャル・タイムアウトまたはセッション・タイムアウトをサポートしていない場合は、CS_SEC_CREDTIMEOUT プロパティまたは CS_SEC_SESSTIMEOUT プロパティの呼び出しが無効になります。

ユーザのクレデンシャルまたはセッションの有効期限が切れた場合、次のように Client-Library とサーバのどちらかが接続をクローズします。

- Client-Library は、ネットワークへの書き込みを行う前にクレデンシャルまたはセッションの有効期限が切れていないかをチェックして、セッションの有効期限が切れている場合に接続をクローズします。
- サーバは、クライアントにデータを送信する前にクレデンシャルまたはセッションの有効期限が切れていないかをチェックして、セッションの有効期限が切れている場合に接続をクローズします。セッションの有効期限が切れているためにサーバが接続をクローズする場合、クライアントには警告メッセージが送信されません。

- CS_SEC_MUTUALAUTH は、接続のセキュリティ・メカニズムが相互認証を実行するように要求します。相互認証を行うには、接続がオープンされる前に、サーバはクライアントにそのサーバ自体であることの証明を示す必要があります。このプロパティのデフォルトは CS_FALSE です。デフォルト値の場合、相互認証は行われません。

  相互認証が要求された場合、接続の確立時に、サーバはクライアントにそのサーバ自体であることの証明を示します。この証明は、サーバが Client-Library に送信したクレデンシャル・トークンから構成されます。このトークンは、サーバのプリンシパル名とその名前が認証されていることの証明をコード化する、隠されたデータのまとまりです。Client-Library はセキュリティ・メカニズムに問い合わせて、受信したトークンが本物であることを確認します。本物でない場合、Client-Library は接続試行をアボートします。

- CS_SEC_SERVERPRINCIPAL プロパティは、接続がオープンされるサーバのネットワーク・セキュリティ・プリンシパル名を指定します。このプロパティのデフォルトは NULL です。デフォルト値の場合、ct_connect は、サーバのプリンシパル名がサーバのディレクトリ・エントリ名と一致するものとみなします。CS_SEC_SERVERPRINCIPAL プロパティは、ネットワークベースの認証が要求される場合にのみ有効です。
- CS_SEC_DELEGATION プロパティは、サーバが委任されたクレデンシャルを使用してリモート・サーバに接続できるかどうかを決定します。このプロパティのデフォルトは CS_FALSE です。デフォルト値の場合、クレデンシャル委任は許可されません。

  委任が適用されるのは、Open Server ゲートウェイに接続する場合にネットワークベースのユーザ認証を使用するアプリケーションだけです。

  クライアントがゲートウェイ・サーバに接続する場合、ゲートウェイは、同じセキュリティ・メカニズムを使用して、ネットワークベースの認証をサポートする 2 番目のリモート・サーバへの接続を確立することがあります。クレデンシャル委任を使用すると、ゲートウェイはクライアントの委任クレデンシャルを使用してリモート・サーバに接続できます。
- CS_SEC_CREDENTIALS プロパティを使用すると、ゲートウェイ・アプリケーションはリモート・サーバにユーザ・クレデンシャルを転送できます。クライアント・アプリケーションは、CS_SEC_DELEGATION 接続プロパティを CS_TRUE に設定してクレデンシャル委任を許可しておく必要があります。

  ゲートウェイによって委任をサポートする場合は、Open Server スレッド・プロパティ、SRV_T_SEC_DELEGCRED に、Client-Library 接続プロパティ CS_SEC_CREDENTIALS の値を設定します。委任が機能するためには、ゲートウェイのクライアント、ゲートウェイ、ゲートウェイのリモート・サーバが同じセキュリティ・メカニズムを使用するようにしてください。

  CS_SEC_CREDENTIALS プロパティは、設定またはクリア専用です。
- CS_SEC_CHANBIND プロパティは、接続のセキュリティ・メカニズムがチャネル・バインドを行うかどうかを指定します。このプロパティのデフォルトは CS_FALSE です。デフォルト値の場合、チャネル・バインドは行われません。

チャネル・バインドが有効である場合は、Client-Library とサーバの両方から、接続のセキュリティ・メカニズムに対して ( クライアントとサーバのネットワーク・アドレスで構成される ) ネットワーク・チャネル識別子が提供されます。

- CS_SEC_KEYTAB プロパティは、オペレーティング・システム・ファイル (「**キータブ・ファイル**」と呼ばれます ) の名前とファイルへのパスを指定します。接続のセキュリティ・メカニズムは、そのファイルからセキュリティ・キーを読み込んで、CS_USERNAME プロパティによって指定されたユーザ名と合わせて使用します。

**注意**　キータブ・ファイルをサポートするのは、DCE セキュリティ・ドライバだけです。

CS_SEC_KEYTAB プロパティは、セキュリティ・メカニズムに DCE を使用してネットワークベースの認証を要求した接続に対してのみ有効です。アプリケーションは、キータブ・ファイルを指定して、アプリケーションを実行している DCE ユーザとは別のユーザ名でサーバに接続します。アプリケーションによって CS_USERNAME プロパティが新しいユーザ名に設定されて CS_SEC_KEYTAB プロパティが設定されると、新しいユーザのセキュリティ・キーを指定するキータブ・ファイルが表示されます。CS_SEC_KEYTAB プロパティのデフォルトは NULL であり、キータブ・ファイルは読み込まれません。この場合、CS_USERNAME プロパティはアプリケーション・ユーザの DCE 名を示す必要があり、また、ユーザは DCE にすでにログインしている必要があります。

キータブ・ファイルは、DCE の dcecp ユーティリティを使用して作成されます (DCE のマニュアルを参照してください )。Client-Library アプリケーションを実行しているユーザが読み込めるように、キータブ・ファイルを作成してください。

### ログインパスワード暗号化の FIPS-140-2 準拠

Open Client と Open Server のログイン・パスワードとリモート・パスワードの暗号化は、Sybase CSI (Common Security Infrastructure) によって実現されます。CSI 2.6 は、連邦情報処理標準 (FIPS: Federal Information Processing Standard) 140-2 に準拠しています。

FIPS 暗号化をサポートするため、まだ Certicom Security Builder を使用していないプラットフォームに *libsbgse2.so* (UNIX と Linux の各プラットフォーム ) または *libsbgse2.dll* (Microsoft Windows プラットフォーム ) という名前の Certicom Security Builder 共有ライブラリがインストールされます。また、*$SYBASE/$SYBASE_OCS/lib3p* または *$SYBASE/$SYBASE_OCS/lib3p64* にある *sybcsi* サブディレクトリは、削除されました。

**パケットごとのセキュリティ・サービスの要求**

ネットワークが物理的に安全でない環境では、分散アプリケーションの設計者がその環境に対処する必要があります。たとえば、認証されていないグループでも、アナライザを物理回線に接続したり、無線伝送を傍受したりして、ダイアログを受信できる可能性があります。

上記のような環境でのアプリケーションは、安全化されたダイアログを保証するために転送データを保護する必要があります。パケットごとのセキュリティ・サービスによって、転送データが保護されます。

パケットごとのサービスでは、接続上で送信されるそれぞれの TDS パケットに対して、次のオペレーションのどちらかまたは両方を実行する必要があります。

- パケットの内容の暗号化
- パケット内容およびその他の必要な情報をコード化する、デジタル署名の計算

---

**注意**　この項で説明するサービスを使用するアプリケーションでは、クライアントとサーバ間のすべての通信でパケットごとのオーバヘッドが生じます。アプリケーションのパフォーマンスよりセキュリティを重視する場合以外は、パケットごとのセキュリティ・サービスを使用しないでください。

---

アプリケーションが複数のパケットごとのサービスを選択した場合、それぞれのオペレーションはパケットごとに一度だけ実行されます。たとえばアプリケーションがデータ機密性サービス、順序検証サービス、データ整合性サービス、チャネル・バインド・サービスを選択した場合、それぞれのパケットは暗号化されて、パケットの内容、パケットの順序情報、ネットワーク・チャネル識別子をコード化するデジタル署名が付加されます。

パケットごとのサービスの場合、データ機密性を除くすべてのサービスで、接続上で送信される各パケットのデジタル署名を接続のセキュリティ・メカニズムによって計算することが要求されます。署名では、パケットの内容に関する情報をコード化するだけでなく、それ以外の情報もコード化することがあります。クライアントとサーバの両方でパケット署名を計算し、それぞれの TDS パケットに計算結果を付加して送信します。パケットと署名を受信すると、セキュリティ・メカニズムは受信した署名を検証します。パケット署名が拒否された場合、接続は次のようにしてクローズされます。

- Client-Library がネットワークから結果を読み込んでいるときにエラーが発生した場合、Client-Library はエラーを表示して、接続をクローズします。
- クライアントが送信したパケットをサーバが読み込んでいるときにエラーが発生した場合、サーバは接続をクローズします。この場合、クライアント・アプリケーションは、ネットワークから結果を読み込もうとするまでエラーを検出できません。

次に示す接続プロパティは、パケットごとのセキュリティ・サービスの使用を制御します。これらのプロパティを有効にするには、接続が確立される前に設定する必要があります。接続レベルでは、接続がオープンしている場合、次に示すすべてのプロパティが取得のみ可能です。以下のプロパティは CS_BOOL **buffer* 値を取り、プロパティのデフォルトは CS_FALSE です。

- CS_SEC_CONFIDENTIALITY プロパティは、転送データの暗号化を要求します。サーバに送信されるすべてのコマンドとサーバから返されるすべての結果が暗号化されます。

  通信媒体が物理的に安全とはいえないパブリック・ネットワーク上で送信されるデータを、データ機密性サービスによって保護します。たとえば、不正ユーザが物理回線にアナライザを接続したり、無線伝送を傍受したりする可能性があります。

- CS_SEC_INTEGRITY プロパティは、接続上で転送されるすべてのデータについて整合性のチェックを行うことを要求します。このサービスでは、サーバとの間で送受信されるすべての TDS パケットをチェックして、パケットの内容に変更がなかったことを確かめます。

  接続がネットワークベースのユーザ認証も使用している場合にのみ、データ整合性のチェックを使用します。

- CS_SEC_DATAORIGIN プロパティは、接続のセキュリティ・メカニズムがデータ・オリジン・スタンピングを行うかどうかを指定します。このサービスでは、接続上で転送される TDS パケットのそれぞれに、パケットの送信者と内容に関する情報をコード化するデジタル署名が付加されます。
- CS_SEC_DETECTREPLAY プロパティは、転送された TDS パケットの不正な繰り返しを、接続のセキュリティ・メカニズムが検出するかどうかを指定します。

  リプレイ検出によって、パケットの受信やリプレイを確実に検出します。たとえば、不正ユーザがパケットを受信し、サーバに送信されたコマンドを、権限もなしに再び実行しようとしてリプレイする可能性があります。
- CS_SEC_DETECTSEQ プロパティは、送信順とは異なる順序で到着した、転送済みの TDS パケットを、接続のセキュリティ・メカニズムが検出するかどうかを指定します。

  リプレイの検出サービスと順序検証サービスは似ていますが、別個のサービスです。たとえば、クライアントが送信したパケットに P1、P2、P3 という送信順で番号が付けられている場合を考えてみます。サーバが P1、P2、P2 という順序でパケットを受信した場合はリプレイ・エラーですが、順序不整合エラーではありません。サーバが P1、P3、P2 という順序でパケットを受信した場合は、順序不整合エラーですが、リプレイ・エラーではありません。

## Open Client/Open Server の SSL (Secure Socket Layer)

SSL は、セッションベースの通信プロトコルの一種であり、クレジット・カード番号、株式売買、銀行取引などの機密情報を、インターネット上で安全に転送するための標準です。

このマニュアルでは、パブリック・キー暗号法については詳しく説明しませんが、SSL によってインターネット通信チャネルの安全性が保証される仕組みを理解できるように、基本的なことについては説明します。したがって、このマニュアルをご覧になっても、すべての知識が得られるわけではありません。

Open Client/Open Server の SSL 機能の実装は、サイトのセキュリティ・ポリシーやニーズを十分に理解し、SSL やパブリック・キー暗号法について全般的な知識のあるシステム・セキュリティ担当者がいることを前提としています。

# インターネット通信の概要

TCP/IP は、クライアント／サーバ・コンピューティングで使用される主要な通信プロトコルであり、インターネット上でのデータ転送を制御します。TCP/IP は、複数のコンピュータを媒介して、送信元から受信先へデータを転送します。複数のコンピュータを経由することによって、通信システムの中に安全性の低いリンクが生じ、データの改ざん、盗難、盗聴、なりすましなどを受けやすくなります。

SSL 対応のクライアント・アプリケーションでは、パブリック・キー暗号法の標準技術を使用して、サーバの証明書を認証します。また、クレジット・カード番号などの個人情報を接続上で送信する前にサーバ証明書が信頼済み CA によって発行されているかどうかも検証します。

## パブリック・キー暗号法

インターネット通信の安全性を確保する目的で、パブリック・キー暗号法と総称されるいくつかのメカニズムの開発や実装が行われてきました。このパブリック・キー暗号法によって、インターネット上で転送される重要なデータが保護されます。パブリック・キーの暗号化には、データの暗号化、キー交換、デジタル署名、デジタル証明書が含まれます。

### 暗号化

暗号化のプロセスでは、暗号化アルゴリズムを使用して情報をコード化し、その情報を目的の受信者以外の者から保護します。暗号化に使用するキーには、次の 2 種類があります。

- 対称キー暗号化では、メッセージの暗号化と復号化に同じアルゴリズム ( キー ) を使用します。この暗号化方式では、簡単に解読できる単純なキーを使用しているため、最低限のセキュリティしか保証されません。ただし、対称キーによるデータの暗号化では、メッセージの暗号化と復号化に必要な計算が少なくて済むので、データ転送が高速になります。

- パブリック・キー／プライベート・キー ( 非対称キーとも呼ばれる ) は、パブリック・コンポーネントとプライベート・コンポーネントの組み合わせによってメッセージの暗号化と復号化を行うキー・ペアです。通常、送信者はプライベート・キーを使用してメッセージを暗号化し、受信者は送信者のパブリック・キーを使用してメッセージを復号化しますが、この組み合わせは異なる場合もあります。送信者が受信者のパブリック・キーを使用してメッセージを暗号化し、受信者が受信者自身のプライベート・キーを使用してメッセージを復号化する場合もあります。

  パブリック・キーとプライベート・キーを作成するときに使用するアルゴリズムは複雑なので、解読するのは容易ではありません。ただし、パブリック・キー／プライベート・キー暗号化方式では、必要な計算量が増えて接続上でのデータ送信量も多くなるので、データ転送速度がかなり遅くなります。

### キー交換

安全性を損なうことなく、計算によるオーバヘッドを減らしてトランザクションを高速化するには、対称キー暗号化とパブリック・キー／プライベート・キー暗号化の両方を組み合わせて使用します。この方法を、キー交換と呼びます。

データ量が多い場合は、対称キーを使用して元のメッセージを暗号化します。次に、送信者は、送信者自身のプライベート・キーまたは受信者のパブリック・キーを使用して、対称キーを暗号化します。暗号化されたメッセージと暗号化された対称キーの両方が受信者に送信されます。メッセージを暗号化するときにはパブリック・キーまたはプライベート・キーを使用しますが、そのときに使用しなかった方のキーを使用して、受信者は対称キーを復号化します。キーの交換が終了すると、受信者は対称キーを使用してメッセージを復号化します。

### デジタル署名

デジタル署名は、改ざん検出と否認防止に使用されます。デジタル署名は、テキスト・メッセージからユニークかつ固定長の数値文字列を作成する算術アルゴリズムを使用して作成されます。作成された文字列は、ハッシュまたはメッセージ・ダイジェストと呼ばれます。

メッセージの整合性を保証するために、メッセージ・ダイジェストは署名者のプライベート・キーで暗号化され、ハッシュ・アルゴリズムについての情報とともに受信者に送信されます。受信者は、署名者のパブリック・キーを使用してメッセージを復号化します。また、この処理では、元のメッセージ・ダイジェストも再生成されます。これらのダイジェストが一致すれば、メッセージは損なわれておらず、改ざんされてもいないことになります。一致しない場合は、転送中にデータが変更されているか、改ざん者によりデータが署名されていることになります。

また、デジタル署名により、否認防止が可能になります。つまり、送信者は、自身のプライベート・キーでメッセージを暗号化したために、メッセージを送ったことを否認できないことになります。ただし、盗難や解読によってプライベート・キーの機密性が損なわれている場合は、当然、デジタル署名は否認防止に対して意味をなしません。

### 証明書

証明書はパスポートのようなものです。証明書がユーザに割り当てられると、認証局は、システムにおけるユーザのあらゆる ID 情報を持つことになります。入国管理局は、ある国から別の国へ移動する旅行者の情報にアクセスできます。パスポートと同様に、証明書は、あるエンティティ ( サーバ、ルータ、Web サイトなど ) を別のエンティティに対して識別するために使用します。

証明書には、次の 2 つのタイプがあります。

- サーバ証明書 – サーバ証明書を保有しているサーバを認証します。米国国務省からパスポートが発行されるように、信頼されたサードパーティの認証局 (CA: Certificate Authority) によって証明書が発行されます。CA は、証明書の保有者の身元を検証し、所有者のパブリック・キーなどの ID 情報を、デジタル証明書に埋め込みます。証明書には、発行元 CA のデジタル署名が含まれています。これによって、証明書データの整合性が確認され、証明書を使用できるようになります。
- CA 証明書 – CA 証明書は、「信頼されたルート証明書」とも呼ばれ、リモート・プロシージャ・コール (RPC : Remote Procedure Call) の呼び出し時など、サーバがクライアントとして機能するときに、サーバが使用する証明書です。Adaptive Server Enterprise は、RPC を実行するためにリモート・サーバに接続するときに、リモート・サーバの証明書に署名した CA が、Adaptive Server Enterprise 自身の CA の信頼されたルート・ファイルにある「信頼された」CA かどうかを検証します。信頼された CA でない場合は、接続が許可されません。

これらのメカニズムの組み合わせにより、インターネットを介して送信されるデータを盗聴や改ざんから守ります。また、なりすまし攻撃からもユーザを保護します。なりすまし攻撃には、あるエンティティが別のエンティティの振りをするもの ( スプーフィング ) や、組織または個人が、機密情報の入手という本当の目的を隠して別の目的を偽るもの ( 虚偽の陳述 ) があります。

## SSL の概要

SSL は、クライアントからサーバ、およびサーバからサーバへワイヤまたはソケット・レベルで暗号化されたデータを送信する業界標準です。サーバとクライアントは何度か I/O を交換し、安全な暗号化セッションをネゴシエートして合意してから、SSL 接続が確立されます。これは、「SSL ハンドシェイク」と呼ばれています。

### SSL ハンドシェイク

クライアント・アプリケーションが接続を要求すると、SSL 対応サーバは証明書を提示し、ID を証明してから、データを送信します。基本的に、SSL ハンドシェイクは次の手順によって構成されています。

- クライアントはサーバに接続要求を送信します。要求には、クライアントがサポートしている SSL ( または TLS: Transport Layer Security) オプションが含まれています。
- サーバは、証明書とサポートされている CipherSuite のリストを返します。これには、SSL/TLS サポート・オプション、キー交換で使用されるアルゴリズム、デジタル署名が含まれます。
- クライアントとサーバがお互いに CipherSuite に合意すると、安全で暗号化されたセッションが確立されます。

「SSL ハンドシェイク」と SSL/TLS プロトコルについては、Internet Engineering Task Force Web site (http://www.ietf.org) を参照してください。

#### パフォーマンス

安全なセッションの確立に必要な、追加のオーバヘッドがあります。データを暗号化するとサイズが増え、情報の暗号化と復号化に追加の計算が必要になるからです。一般に、SSL ハンドシェイク中に生じる I/O の増加によって、ユーザ・ログインにかかる時間が 10 倍から 20 倍になることがあります。

### CipherSuite

SSL ハンドシェイク中に、クライアントとサーバは、CipherSuite を介して共通のセキュリティ・プロトコルをネゴシエートします。CipherSuite は、SSL プロトコルで使用するキー交換アルゴリズム、ハッシュ方式、暗号化方式の優先リストです。CipherSuite の詳細については、IETF の Web サイト (IETF organization Web site (http://www.ietf.org/rfc/rfc2246.txt)) を参照してください。

デフォルトでは、クライアントとサーバの両方がサポートしている最も強力な CipherSuite は、SSL ベースのセッションに使用される CipherSuite です。

サーバ接続の属性は、LDAP、DCE などのディレクトリ・サービス、または従来の Sybase の *interfaces* ファイルで指定されます。

---

**注意** 次に示す CipherSuite が TLS 仕様に準拠しています。TLS (Transport Layer Security) は、SSL 5.0 の拡張バージョンで、SSL バージョン 5.0 CipherSuite のエイリアスでもあります。

---

Open Client/Open Server と Adaptive Server Enterprise は、SSL Plus のライブラリ API と、暗号化エンジンの Security Builder ( どちらも Certicom 製 ) を備えた CipherSuite をサポートしています。

```
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
SSL_DHE_DSS_WITH_DES_CBC_SHA
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
SSL_DHE_RSA_WITH_DES_CBC_SHA
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
SSL_RSA_WITH_NULL_MD5
SSL_RSA_EXPORT_WITH_RC4_40_MD5
SSL_RSA_WITH_RC4_128_MD5
SSL_RSA_WITH_RC4_128_SHA
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
SSL_RSA_WITH_DES_CBC_SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA RSA
TLS_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA
TLS_DHE_DSS_EXPORT1024_WITH_RC4_56_SHA
TLS_DHE_DSS_WITH_RC4_128_SHA
TLS_RSA_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_AES_128_CBC_SHA
```

## Open Client/Open Server の SSL

SSL には、いくつかのセキュリティ・レベルがあります。

- SSL-対応サーバへの接続を確立すると、サーバは接続対象のサーバであることを自己認証し、暗号化された SSL セッションが開始されてからデータが送信されます。
- SSL セッションが確立されると、ユーザ名とパスワードが暗号化された安全な接続によって送信されます。
- サーバ証明書のデジタル署名を比較して、サーバから受信したデータが転送中に変更されたかどうかを判断します。

## SSL フィルタ

SSL-対応 Adaptive Server Enterprise への接続を確立するとき、SSL セキュリティ・メカニズムは、*interfaces* ファイル (Windows では *sql.ini*) の master 行と query 行のフィルタとして設定されます。TCP/IP 接続の上層に位置する Open Client/Open Server プロトコル層として SSL を使用します。

SSL フィルタは、*interfaces* ファイル (Windows では *sql.ini*) の SECMECH (security mechanism) 行で定義されている DCE や Kerberos などの他のセキュリティ・メカニズムとは異なります。master 行と query 行では、接続に使用されるセキュリティ・プロトコルを指定します。

たとえば、SSL を使用している UNIX マシンの一般的な *interfaces* ファイルは、次のようになります。

```
[SERVER]
query tcp /dev/tcp add1 ssl
master tcp /dev/tcp add1 ssl
```

SSL を使用している Windows 上の一般的な *sql.ini* ファイルは、次のようになります。

```
[SERVER]
query=TCP,hostname,address1, ssl
master=TCP,hostname,address1, ssl
```

*hostname* にはクライアントが接続しているサーバの名前が、*address1* にはホスト・マシンのポート番号が入ります。*interfaces* ファイル内で SSL フィルタが指定されている master エントリまたは query エントリに接続するには、その接続で SSL プロトコルがサポートされている必要があります。サーバを、SSL 接続を受け入れ、他の接続によってプレイン・テキスト ( 非暗号化データ ) を受け入れるように設定したり、他のセキュリティ・メカニズムを使用するように設定できます。

たとえば、SSL ベースの接続とプレーン・テキストの接続の両方をサポートする UNIX の Adaptive Server Enterprise の *interfaces* ファイルは、次のようになります。

```
SYBSRV1
    master tcp /dev/tcp ¥x00020abc123456780000000000000000 ssl
    query tcp /dev/tcp ¥x00020abc123456780000000000000000 ssl
    master tcp /dev/tcp ¥x00020abd123456780000000000000000
```

または、UNIX の新しいスタイルの Sybase *interfaces* ファイルの場合、同じエントリは次のようになります。

```
SYBSRV1
    master tcp hostname 2748 ssl
    query tcp hostname 2748 ssl
    master tcp hostname 2749
```

ソケットスタイルの *interfaces* ファイルでは、次のようになります。

```
SYBSRV1
    master tcp ether hostname 2748 ssl
    query tcp ether hostname 2748 ssl
    master tcp ether hostname 2749
```

これらの例では、SSL セキュリティ・サービスはポート番号 2748 (0x0abc) に設定されています。SYBSRV1 では、Adaptive Server Enterprise はポート番号 2749 (0x0abd) でクリア・テキストを受信します。これには、セキュリティ・メカニズムとセキュリティ・フィルタがありません。

## 証明書によるサーバの検証

Open Client/Open Server から SSL 対応サーバに接続するときは、サーバ側に証明書ファイルが必要です。このファイルの内容は、サーバの証明書と、暗号化されたプライベート・キーです。また、証明書は CA がデジタル署名したものでなければなりません。

既存のクライアント接続が確立されるのと同じように、Open Client アプリケーションは Adaptive Server Enterprise へのソケット接続を確立します。ネットワークのトランスポート層の接続コールがクライアント・サイドで完了し、受け入れコールがサーバ・サイドで完了すると、SSL ハンドシェイクが行われます。それから、ユーザのデータが送信されます。

SSL- 対応サーバに正しく接続するには、次の手順に従ってください。

- クライアント・アプリケーションが接続要求を行った場合は、SSL- 対応サーバは証明書を提出しなければなりません。

- クライアント・アプリケーションは、証明書に署名した CA を認識しなければなりません。「信頼された」CA すべてを含んだリストは、信頼されたルート・ファイルにあります。「信頼されたルート・ファイル」(313 ページ ) を参照してください。
- SSL-対応サーバへの接続では、サーバ証明書の共通名は *interfaces* ファイルのサーバ名とも一致していなければなりません。

---

**注意**　SSL ハンドシェイクを傍受して、SSL 検証チェックを無効にする SSL 検証コールバックのインストールを選択できます。SSL 検証コールバックは、CS_SSLVALIDATE_CB を使用して ct_callback によりインストールされます。

---

SSL 対応 Adaptive Server Enterprise への接続を確立すると、Adaptive Server Enterprise は起動時に次の場所からサーバ自体のコード化された証明書ファイルをロードします。

UNIX の場合 － *$SYBASE/$SYBASE_ASE/certificates/servername.crt*

Windows の場合 － *%SYBASE% ¥%SYBASE_ASE% ¥certificates ¥ servername.crt*

ここで、*servername* は、コマンド・ラインからサーバを起動したときに -S フラグで指定した Adaptive Server Enterprise の名前か、またはサーバの環境変数 $DSLISTEN で指定した Adaptive Server Enterprise の名前です。

ほかのタイプのサーバでは、別のロケーションに証明書を保管することがあります。サーバの証明書のロケーションの詳細については、ベンダ提供マニュアルを参照してください。

### SDC 環境での検証

Open Client と Open Server における SSL 検証のデフォルトの動作は、サーバ証明書での共通名を ct_connect で指定されたサーバ名と比較することです。ただし、共有ディスク・クラスタ (SDC: Shared Disk Cluster) 環境では、クライアントはサーバ名または SDC インスタンス名とは無関係の SSL 証明書の共通名を指定できます。クライアントは、複数のサーバ・インスタンスを表すクラスタ名で SDC に接続することも、特定の 1 つのサーバ・インスタンスに接続することもできます。

Open Client と Open Server は、SDC 環境における共通名の検証をサポートしています。これが可能なのは、トランスポート・アドレスを使用して、証明書の検証で使用される共通名を指定できるからです。したがって、Adaptive Server Enterprise の SSL 証明書の共通名は、サーバ名またはクラスタ名と同じである必要はありません。トランスポート・アドレスは、ディレクトリ・サービス (*interfaces* ファイル、LDAP、NT レジストリなど ) のいずれか、または接続プロパティ CS_SERVERADDR で指定できます。

次に示すのは、UNIX 用の SSL 対応 Adaptive Server Enterprise とクラスタの *interfaces* ファイルの例です。

```
CLUSTERSSL
query tcp ether hostname1 5000 ssl="CN=name1"
query tcp ether hostname2 5000 ssl="CN=name2"
query tcp ether hostname3 5000 ssl="CN=name3"

ASESSL1
master tcp ether hostname1 5000 ssl="CN=name1"
query tcp ether hostname1 5000 ssl="CN=name1"

ASESSL2
master tcp ether hostname2 5000 ssl="CN=name2"
query tcp ether hostname2 5000 ssl="CN=name2"

ASESSL3
master tcp ether hostname3 5000 ssl="CN=name3"
query tcp ether hostname3 5000 ssl="CN=name3"
```

次に示すのは、Windows 用の SSL 対応 Adaptive Server Enterprise とクラスタの *interfaces* ファイルの例です。

```
[CLUSTERSSL]
query=tcp,hostname1,5000, ssl="CN=name1"
query=tcp,hostname2,5000, ssl="CN=name2"
query=tcp,hostname3,5000, ssl="CN=name3"

[ASESSL1]
master=tcp,hostname1,5000, ssl="CN=name1"
query=tcp,hostname1,5000, ssl="CN=name1"

[ASESSL2]
master=tcp,hostname2,5000, ssl="CN=name2"
query=tcp,hostname2,5000, ssl="CN=name2"

[ASESSL3]
master=tcp,hostname3,5000, ssl="CN=name3"
query=tcp,hostname3,5000, ssl"CN=name3"
```

## 信頼されたルート・ファイル

信頼された既知の CA のリストは、信頼されたルート・ファイルに保管されています。エンティティ ( クライアント・アプリケーション、サーバ、ネットワーク・リソースなど ) に既知の CA の証明書がある以外は、信頼されたルート・ファイルは証明書ファイルのフォーマットと同じです。システム・セキュリティ担当者が、標準 ASCII テキスト・エディタを使って認証局を追加したり、削除したりします。

Open Client/Open Server の信頼されたルート・ファイルは、次のロケーションにあります。

- UNIX の場合 － *$SYBASE/config/trusted.txt*
- Windows の場合 － *%SYBASE% ¥ini ¥trusted.txt*

現時点で認識されている CA は、Thawte、Entrust、Baltimore、VeriSign、RSA です。

デフォルトでは、Adaptive Server Enterprise はサーバ自身の信頼されたルート・ファイルを次のロケーションに保管します。

- UNIX － *$SYBASE/$SYBASE_ASE/certificates/servername.txt*
- Windows － *%SYBASE% ¥%SYBASE_ASE% ¥certificates ¥ servername.txt*

Open Client と Open Server の両方を使用すると、次のように信頼されたルート・ファイルを別のロケーションに設定できます。

- Open Client

  ```
  ct_con_props (connection, CS_SET, CS_PROP_SSL_CA,
  "$SYBASE/config/trusted.txt", CS_NULLTERM, NULL);
  ```

  ここで、*$SYBASE* には、インストール・ディレクトリが入ります。ct_config を使用してコンテキスト・レベルに、または ct_con_props を使用して接続レベルに CS_PROP_SSL_CA を設定できます。

- Open Server

  ```
  srv_props (context, CS_SET, SRV_S_CERT_AUTH,
  "$SYBASE/config/trusted.txt", CS_NULLTERM, NULL);
  ```

  ここで、*$SYBASE* には、インストール・ディレクトリが入ります。

bcp ユーティリティと isql ユーティリティでも、別の場所にある信頼されたルート・ファイルを指定できます。新しいパラメータ -x が構文に追加されており、このパラメータを使用して *trusted.txt* ファイルの場所を指定します。

## 証明書の取得

システム・セキュリティ担当者が、署名付きサーバ証明書とプライベート・キーをサーバにインストールします。次の手順によって、サーバ証明書を取得できます。

- 顧客環境に配備されている既存のパブリック・キー・インフラストラクチャで提供されているサードパーティのツールを使用します。
- Sybase 証明書要求ツールをサードパーティの信頼済み CA に使用します。

証明書を取得するには、CA に証明書を要求してください。サードパーティに証明書を要求し、その証明書が PKCS #12 フォーマットの場合は、certpk12 ユーティリティを使用して、Open Client/Open Server が理解できるフォーマットに証明書を変換します。

証明書要求ツールをテストし、認証方法がサーバで機能していることを確認するために、Open Client/Open Server は、検証目的で certreq ツールと certauth ツールを提供しています。このツールを使用すると、ユーザは CA として機能できるので、CA 署名済み証明書をユーザ自身に発行できます。

サーバで使用する証明書を作成するには、次の手順に従います。

1. 証明書要求を生成します。
2. パブリック・キーとプライベート・キーのペアを生成します。
3. プライベート・キーを安全な場所に保管します。
4. 証明書要求を CA に送信します。
5. 署名付きの証明書が CA から返信されたら、その証明書にプライベート・キーを付加します。
6. サーバのインストール・ディレクトリに証明書を保管します。

### 証明書を要求するサードパーティ・ツール

多くのサードパーティ PKI ベンダといくつかのブラウザには、証明書とプライベート・キーを生成するユーティリティがあります。これらのユーティリティの多くはグラフィカルなウィザード形式で、一連の質問にユーザが答えると証明書の識別名と共通名が定義されます。

ウィザードの指示に従って、証明書要求を作成します。PKCS #12 フォーマットの署名付き証明書を受け取ったら、certpk12 を使用して、証明書ファイルとプライベート・キー・ファイルを生成します。2 つのファイルを連結して、1 つの *servername.crt* ファイルにします。*servername* はサーバの名前です。このファイルは、サーバのインストール・ディレクトリに配置されます。デフォルトでは、Adaptive Server Enterprise の証明書は *$SYBASE/$SYBASE_ASE/certificates* に保管されます。

### Sybase ツールによる証明書の要求と認証

Sybase では、証明書の要求と認証を行うツールを提供しています。certreq では、パブリック・キーとプライベート・キーのペア、および証明書要求を生成します。certauth では、サーバ証明書要求を CA 署名済み証明書に変換します。

- UNIX の場合 － *$SYBASE/$SYBASE_OCS/bin*
- Windows の場合 － *%SYBASE% ¥%SYBASE_OCS% ¥bin*

---

**警告！** certauth は、テスト専用で使用します。商用 CA のサービスを利用することをおすすめします。商用 CA はルート証明書の整合性を保護しており、広く承認された CA により署名された証明書を使用すれば、クライアント証明書を使用する形式の認証への移行が促進されるからです。

---

次の手順 1 ～ 5 に従って、サーバの信頼されたルート証明書を用意します。サーバ証明書を作成できることを確認するために、5 つの手順すべてを行い、検査用の信頼されたルート証明書を作成します。テスト版の CA 証明書 ( 信頼されたルート証明書 ) を作成したら、手順 3 ～ 5 を繰り返してサーバ証明書に署名します。

1. certreq を使用して、証明書を要求します。
2. certauth を使用して、証明書要求を CA の自己署名済み証明書 ( 信頼されたルート証明書 ) に変換します。
3. certreq を使用して、サーバ証明書とプライベート・キーを要求します。
4. certauth を使用して、証明書要求を CA 署名付きサーバ証明書に変換します。

5 プライベート・キーのテキストをサーバ証明書に付加して、サーバのインストール・ディレクトリに証明書を格納します。

---

**注意** certauth と certreq は、RSA と DSA のアルゴリズムに依存しています。これらのツールは、ベンダが提供する暗号モジュールがある場合にのみ実行されます。この暗号モジュールでは、RSA と DSA アルゴリズムを使用して証明書要求を構築します。

---

Adaptive Server Enterprise でサーバ証明書を追加、削除、表示する方法については、Adaptive Server Enterprise の『システム管理ガイド』を参照してください。

## Adaptive Server Enterprise のセキュリティ機能

Adaptive Server Enterprise、または Open Server バージョン 10.0 以降に接続するクライアント・アプリケーションでは、パスワードの暗号化、およびチャレンジ／応答セキュリティ・ハンドシェイクを利用できます。

### セキュリティ・ハンドシェイク：チャレンジ／応答

サーバは、チャレンジ／応答セキュリティ・ハンドシェイクを使用して、ログイン・セキュリティ・チェックのレベルを強化できます。

このハンドシェイク方法で要求される応答を提供するには、アプリケーションを次のようにコーディングしてください。

- アプリケーションから ct_con_props ルーチンを呼び出して次のプロパティのどちらかを設定してから、ct_connect ルーチンを呼び出します。
    - Sybase 定義のチャレンジ／応答セキュリティ・ハンドシェイクを要求する CS_SEC_CHALLENGE プロパティ
    - Open Server アプリケーション定義のチャレンジ／応答セキュリティ・ハンドシェイクを要求する CS_SEC_APPDEFINED プロパティ

  これらのプロパティのどちらかまたはその両方が CS_TRUE である場合、ct_connect はサーバのチャレンジに応答して、アプリケーションのネゴシエーション・コールバックを呼び出します。

- アプリケーションにネゴシエーション・コールバックをコーディングして、要求された応答を返せるようにします。

- アプリケーションから ct_callback ルーチンを呼び出して、コンテキスト・レベルでコールバックをインストールするか、特定の接続のコールバックをインストールします。

「ネゴシエーション・コールバックの定義」(51 ページ ) を参照してください。

## セキュリティ・ハンドシェイク：暗号化したパスワード

クライアントがパスワードの暗号化を要求する場合に、Sybase Server は暗号化したパスワードのハンドシェイクを使用します。暗号化したパスワードのセキュリティ・ハンドシェイクは、サーバへの接続が確立されている間に発生します。

---

**注意**　アプリケーションは、CS_SEC_EXTENDED_ENCRYPTION 接続プロパティまたは CS_SEC_ENCRYPTION 接続プロパティを CS_TRUE ( デフォルトは CS_FALSE) に設定して、パスワードの暗号化を要求する必要があります。そうでない場合、パスワードはプレーン・テキストとしてサーバに送信されます。

---

### パスワードの暗号化の処理

パスワードの暗号化が有効になっている場合、サーバは次の手順に従って、ユーザのパスワードとリモート・サーバのパスワードを受信します。

1. 最初に Client-Library は、長さ 0 の文字列からなるダミー・パスワードをサーバに送信します。
2. サーバは、暗号化したパスワードをクライアントに要求し、暗号化キーをクライアントに送信して応答します。
    - クライアント・プログラムが暗号化コールバックをインストールしている場合、Client-Library は、ローカル・パスワードに対して一度、それぞれのリモート・サーバ・パスワードに対して一度コールバックを呼び出します。Client-Library は、暗号化コールバックを呼び出すたびに、暗号化されるパスワードと暗号化キーを引数として提供します。
    - クライアント・プログラムが暗号化コールバックをインストールしていない場合、Client-Library はすべてのパスワードに対してデフォルトの暗号化を実行します。

### Client-Library アプリケーションでのパスワードの暗号化の使用

デフォルトではパスワードの暗号化が無効であるため、パスワードの暗号化を必要とするアプリケーションでは、CS_SEC_EXTENDED_ENCRYPTION プロパティまたは CS_SEC_ENCRYPTION プロパティを CS_TRUE に設定してから、ct_connect を呼び出す必要があります。以下に、パスワード暗号化を有効にするために使用できるコードのサンプルを示します。

#### 通常のパスワード暗号化の有効化

```
CS_BOOL boolval;
/* Enable password encryption for the connection attempt.*/
boolval = CS_TRUE;

if (ct_con_props(conn, CS_SET, CS_SEC_ENCRYPTION, (CS_VOID *)&boolval,
    CS_UNUSED,(CS_INT *)NULL) != CS_SUCCEED)
{
    fprintf(stdout,"ct_con_props(SEC_ENCRYPTION) failed.Exiting¥n");
    (CS_VOID)ct_con_drop(conn);
    (CS_VOID)ct_exit(ctx, CS_FORCE_EXIT);
    (CS_VOID)cs_ctx_drop(ctx);
    exit(1);
}
```

#### 拡張パスワード暗号化の有効化

```
...
CS_INT Ex_encryption = CS_TRUE;
CS_INT Ex_nonencryptionretry = CS_FALSE;
...
main()
{
    ...
    /*
    ** This needs to be called before calling ct_connect()
    */
     ret = ct_con_props(connection, CS_SET, CS_SEC_EXTENDED_ENCRYPTION,
                        &Ex_encryption, CS_UNUSED, NULL);
    EXIT_ON_FAIL(context, ret, "Could not set extended encryption");

    ret = ct_con_props(connection, CS_SET, CS_SEC_NON_ENCRYPTION_RETRY,
                        &Ex_nonencryptionretry, CS_UNUSED, NULL);
    EXIT_ON_FAIL(context, ret, "Could not set non encryption retry");

    ....
}
```

パスワードの暗号化は、Client-Library のデフォルトの暗号化ハンドラを使用するか、ct_callback によってインストールされるアプリケーション・ハンドラを使用して実行されます。

デフォルトの暗号化ハンドラは、Adaptive Server Enterprise が予測している暗号化を実行します。Adaptive Server Enterprise、または Adaptive Server Enterprise への Open Server ゲートウェイに接続するアプリケーションは、デフォルトの暗号化を使用する必要があります。大部分のアプリケーションは、このタイプに分類されます。

暗号化ハンドラを要求するアプリケーションには、次のものがあります。

- Adaptive Server Enterprise に接続する Open Server ゲートウェイは、暗号化コールバックによるパスワードの暗号化をサポートする必要があります。暗号化コールバックでは、(srv_negotiate を呼び出して ) ゲートウェイのクライアントから暗号化されたパスワードが取得され、( コールバックの出力パラメータを使用して ) リモート・サーバにそれぞれのパスワードが転送されます。
- ( カスタム Open Server に接続するアプリケーションなど ) カスタムのパスワード暗号化方法を要求するクライアント・アプリケーションは、サーバが仮定している暗号化を実行するカスタム暗号化コールバックをインストールする必要があります。

パスワード暗号化コールバックを定義する方法については、「暗号コールバックの定義」(46 ページ ) を参照してください。

# サーバ・ディレクトリ・オブジェクト

「サーバ・ディレクトリ・オブジェクト」は、Sybase のサーバについて記述するディレクトリ・エントリの論理的な内容を概略説明します。

「ディレクトリ・サービス」(115 ページ ) を参照してください。

## サーバ・ディレクトリ・オブジェクトの使い方

サーバ・ディレクトリ・オブジェクトは、ct_connect を使用してサーバに接続するとき、暗黙的にアクセスされます。また、ct_ds_lookup とディレクトリ・コールバックを使用して、アプリケーションからディレクトリ内のサーバ・エントリを検索することもできます。

Client-Library アプリケーションは、ct_ds_objinfo を使用して、ディレクトリ・オブジェクトの内容を調査します。

## サーバ・ディレクトリ・オブジェクトの内容

Client-Library は、ディレクトリ内のサーバ・エントリをここで説明するサーバ・ディレクトリ・オブジェクトにマップします。サーバ・ディレクトリ・オブジェクトによって、実際の記憶フォーマットに依存しなくてもディレクトリ・エントリを表示できます。オブジェクトは、サーバ・エントリが属性の値を保持できるように、属性のセットとして定義されます。

ディレクトリ・エントリの実際の記憶フォーマットは、使用するディレクトリ・サービスによって異なります。各「**ディレクトリ・ドライバ**」は、エントリのフォーマットを、ネイティブな記憶フォーマットからサーバ・ディレクトリ・オブジェクトのフォーマットに変換します。オブジェクト・フォーマットは、ディレクトリ・エントリの一般的な表示を Client-Library アプリケーションに提供します。

### オブジェクト属性のフォーマット

ディレクトリ・オブジェクトはそれぞれ、そのタイプのディレクトリ・エントリに保管する属性セットを指定します。属性は、メタデータと 1 つ以上の値を持ちます。属性のメタデータは CS_ATTRIBUTE 構造体によって表され、次の内容で構成されます。

- 属性を識別する名前

  属性の命名規則はディレクトリ・プロバイダ間で異なる可能性があるため、Client-Library ではオブジェクト識別子 (OID : Object IDentifier) を使用してそれぞれの属性を識別します。Client-Library は、それぞれの属性に対して事前に定義されている OID 文字列マクロを提供します。

- 値の構文識別子

  これは、属性値を保持する C のデータ型を示す整数コードです。

- 属性のインスタンス内での値の数

  値は、CS_ATTRVALUE 共用体を使用して取得されます。アプリケーションでは、構文識別子を使用して、共用体のどのメンバが値を保持しているかを認識します。

CS_ATTRIBUTE 構造体と CS_ATTRVALUE 構造体については、「オブジェクト属性と属性値の取得」(528 ページ ) を参照してください。

## 属性のリスト

表 2-34 は、サーバ・ディレクトリ・オブジェクトの属性、および各属性の構文と OID 文字列を示します。詳細については、表の後の説明を参照してください。

---

**注意**　ct_ds_objinfo を使用してサーバ・ディレクトリ・オブジェクトを調べるアプリケーションは、予期していない属性を適切に処理できるようにコーディングしてください。Sybase では、ここにリストしていないサーバ・ディレクトリ・オブジェクトに対して、属性を追加することがあります。

---

**表 2-34：サーバ・ディレクトリ・オブジェクトの属性**

| 属性と対応する OID 文字列 | 値の構文 | 説明 |
|---|---|---|
| サーバ・エントリのバージョン<br>CS_OID_ATTRVERSION | 整数 | サーバのバージョン・レベル。 |
| サーバ名属性<br>CS_OID_ATTRSERVNAME | 文字列 | サーバの名前。<br>名前属性の値は、サーバのディレクトリ・エントリの完全に修飾された名前と異なってもかまわない。 |
| サービス記述<br>CS_OID_ATTRSERVICE | 文字列 | サーバが提供するサービスの説明。<br>意味を持つ記述を任意に指定できる。 |
| サーバのステータス<br>CS_OID_ATTRSTATUS | 整数 | サーバの動作ステータス。<br>有効な値とその意味については、「サーバのステータス」(323 ページ ) を参照。<br>**注意**　Adaptive Server Enterprise の場合、常にステータスは不定である。 |
| トランスポート・アドレス<br>CS_OID_ATTRADDRESS | Transport Address | サーバに対する 1 つ以上のトランスポート・アドレス。<br>トランスポート・アドレス属性には、次の 3 つの要素がある。<br>• トランスポート・タイプ<br>• アクセス・タイプ<br>• トランスポート・アドレス |

| 属性と対応する OID 文字列 | 値の構文 | 説明 |
| --- | --- | --- |
| セキュリティ・メカニズム<br>CS_OID_ATTRSECHMECH | OID | 1 つ以上のサーバでサポートされるセキュリティ・メカニズム。この属性はオプション。 |

## サーバ・エントリのバージョン

サーバ・エントリ・バージョンは、サーバのソフトウェア・バージョンを示す記号整数コードを保持します。バージョン属性は、ディレクトリ・ユーザに便利なように提供されるものです。

バージョン属性は、管理専用で使用します。この属性の値は、サーバへの接続の機能には影響しません。

## サーバ名属性

サーバ名属性によって提供されるサーバ名は、ct_ds_lookup を使用してディレクトリを検索するアプリケーションが認識できるサーバ名です。

名前には、CS_MAX_DS_STRING 以下のバイト長の任意の文字列を指定できます。命名規則に従い、名前属性は、サーバ自体が使用する名前と一致させる必要があります (Adaptive Server Enterprise の場合は、ローカル・サーバ名は sp_addname で指定 )。

サーバ名属性と、ディレクトリ・エントリのロケーションを示すために使用する名前とを混同しないでください。後者は、ディレクトリ・プロバイダの名前構文で表される、ディレクトリ・エントリの完全に修飾された名前です。ct_connect は、完全に修飾された名前を使用してディレクトリ・エントリを検索します。名前属性は、ディレクトリ・ユーザに便利なように提供される任意の文字列値です。混同を避けるため、ディレクトリの管理者は、名前属性がサーバの完全に修飾された名前と少なくとも部分的に一致する ( たとえば、属性値をエントリの共通名にする ) ようにしてください。

---

**注意**　ディレクトリ・プロバイダが *interfaces* ファイルである場合は、名前属性の値はエントリの名前と同じです。

---

## サービス記述

サービス記述属性によって、サーバが提供するサービスを記述します。サービス・タイプの値には、CS_MAX_DS_STRING 以下のバイト長の任意の文字列を指定できます。

Sybase の *interfaces* ファイルがディレクトリ・ソースである場合は、この値は常に「Adaptive Server Enterprise」です。

## サーバのステータス

サーバ・ステータスは、サーバの動作ステータスを示す記号整数コードです。使用できる値を表 2-35 にリストします。

**表 2-35 : ステータス属性値 ( サーバ・ディレクトリ・オブジェクト )**

| ステータス値 | 意味 |
|---|---|
| CS_STATUS_ACTIVE | サーバは稼働中である。 |
| CS_STATUS_STOPPED | サーバはオフラインにされていて、再起動されていない。 |
| CS_STATUS_FAILED | サーバはエラーのためにオフラインになっている。 |
| CS_STATUS_UNKNOWN | サーバのステータスは不定である。詳細については、「不定ステータス値」(323 ページ ) を参照。 |

### 不定ステータス値

ステータス属性の値は、次のような場合に不定になることがあります。

- サーバが Adaptive Server Enterprise である場合 － Adaptive Server Enterprise は、そのステータスをディレクトリ・サービスに登録しません。Adaptive Server Enterprise ディレクトリ・エントリのステータス属性は、常に CS_STATUS_UNKNOWN です。
- ディレクトリ検索で *interfaces* ファイルを使用する場合 － 調べる対象のディレクトリ・オブジェクトが *interfaces* ファイルからのものである場合、ステータス属性は常に不定です。*interfaces* ファイルはステータス属性をサポートしないので、ct_ds_lookup でファイル・エントリを取得してそれらをディレクトリ・オブジェクトに変換する場合、ステータス属性はデフォルトで CS_STATUS_UNKNOWN になります。
- 未登録の Open Server アプリケーションの場合 － Open Server アプリケーションは、自己登録してディレクトリ・サービスを初期設定の一部として使用します。

  Open Server が自己登録した場合には、Server-Library はサーバの現在の動作状況を反映するように自動的にステータス属性値を設定します。アプリケーションが自己登録しない場合は、ステータス属性は常に CS_STATUS_UNKNOWN になります。

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。

## トランスポート・アドレス

トランスポート・アドレス属性を ct_connect で使用して、サーバへの接続を確立します。トランスポート・アドレス属性には、複数のトランスポート・アドレス値を指定できます。

---

**注意**　SDC 環境で、クライアントがサーバ名または SDC インスタンス名とは無関係の SSL 証明書の共通名を指定する場合、クライアントはトランポート・アドレスを使用して、証明書の検証で使用される共通名を指定します。

---

Client-Library アプリケーションでは、トランスポート・アドレス値を CS_TRANADDR 構造体として表示します。この構造体のフォーマットの詳細については、「トランスポート・アドレス値」(530 ページ )を参照してください。

### 複数のトランスポート・アドレス・タイプ

サーバは、複数のネットワーク・トランスポート・タイプに接続できるようにしている場合があります。使用しているネットワークのインストール環境と Sybase ネットワーク・ドライバの設定内容によって、システム上で Client-Library が使用するトランスポート・タイプが決まります。使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

### スタンバイ・サーバ・アドレッシング

サーバ・エントリには、ネットワーク設定に使用する複数のアドレス値を指定できます。この場合、ct_connect は次のいずれかの条件が満たされるまで、必要に応じて繰り返しそれぞれの有効なアドレスに順番に接続しようとします。

- 接続ダイアログがそのアドレスで受け入れられた場合。
- 各アドレスに対する接続の試行を *retry_count* 回行った場合 (*retry_count* は CS_RETRY_COUNT 接続プロパティの値 )。

CS_LOOP_DELAY 接続プロパティは、この手順を再開するまでに Client-Library が待機する時間を秒単位で設定します。Client-Library は、この手順のそれぞれのアドレスから次のアドレスへの間では待機しません。

CS_RETRY_COUNT プロパティについては「リトライ回数」(264 ページ ) を、CS_LOOP_DELAY プロパティについては「ループ遅延」(253 ページ ) を参照してください。

## セキュリティ・メカニズム

セキュリティ・メカニズム属性はオプションであり、複数の値を取る属性です。この属性には、サーバがサポートしている Sybase セキュリティ・メカニズム用に、1 つ以上の OID 文字列を指定できます。

クライアント・アプリケーションは、CS_SEC_MECHANISM 接続プロパティを設定 ( または、デフォルトを使用 ) して接続のセキュリティ・メカニズムを指定します。「ネットワーク・セキュリティ・メカニズムの選択」(293 ページ ) を参照してください。

セキュリティ・メカニズムの OID は、Sybase グローバル・オブジェクト・ファイルによってローカル・セキュリティ・メカニズム名にマップされます。「グローバル・メカニズム名」(294 ページ ) を参照してください。

サーバのディレクトリ・エントリにセキュリティ・メカニズム属性が存在する場合、クライアントは、リストされているサービスだけを使用して指定のサーバに接続します。セキュリティ・メカニズム属性が存在しない場合、クライアントは、サーバがサポートするように設定されている任意のメカニズムを使用して接続します。

# interfaces ファイルのサーバ・オブジェクト

ネットワーク上のディレクトリから読み込むように設定されていないアプリケーションは、Sybase の *interfaces* ファイルからサーバ・ディレクトリ・オブジェクトを読み込みます。

Client-Library が *interfaces* ファイル・エントリをサーバ・ディレクトリ・オブジェクトにマップする方法については、「interfaces ファイルのサーバ・オブジェクト」(159 ページ ) を参照してください。

Sybase の *interfaces* ファイルの詳細については、「interfaces ファイル」(157 ページ ) を参照してください。

# サーバの制限

Client-Library は、汎用的なプログラミング・インタフェースです。これは、機能的に接続先のサーバに依存しないことを意味します。これによって Open Client アプリケーションは、Adaptive Server Enterprise、Open Server といったアプリケーションと通信できるだけでなく、Open Server アプリケーションがゲートウェイであれば Sybase 以外のサーバとも通信できます。

機能的に依存性がないので、Open Client は、サーバが特定の機能を実装するために選択する方法について認識する必要はありません。複数のサーバによって実装される同一の機能が、さまざまに異なった動作を示す可能性があります。サーバ機能の動作は、そのときにアクセスしているサーバに固有なものです。

Open Client アプリケーションの開発者は、作成するアプリケーションが使用されるサーバの動作について十分に理解してください。サポートされる機能、および適用される制限についても知っておく必要があります。

## Open Server の制限

Open Client と Open Server は、Adaptive Server Enterprise の制限を継承しません。これは、Open Client アプリケーションと Open Server アプリケーション間での通信は、Sybase Server の動作に対する規則によって制限されないことを意味します。

ただし、通信は Open Server アプリケーションの実装による制限を受けます。たとえば、Open Server アプリケーションの開発者が、SRV_RPC イベント・ハンドラをインストールせず、リモート・プロシージャ・コール (RPC：Remote Procedure Call) をサポートしないことに決定する可能性があります。Open Client アプリケーションの開発者はこのことを考慮に入れてください。

Open Client と Open Server は、お互いを十分に反映する必要があります。Open Server は、Open Client から送信されたものをすべて受信できます。また、Open Client は、Open Server から送信されたものをすべて受信できます。制限は、Open Server アプリケーションに対して実装固有の制限がある場合だけでなく、Open Server で使用できる機能が有効でない場合にも発生します。

## Adaptive Server Enterprise の制限

アプリケーションが Adaptive Server Enterprise での制限に注意する必要があるのは、Open Client アプリケーションが Adaptive Server Enterprise にアクセスする場合だけです。たとえば、Adaptive Server Enterprise にはログイン名に関する条件があります。つまり、ログイン名を Adaptive Server Enterprise 識別子の規則に従ってユニークなものにする必要があります。Open Client アプリケーションが Adaptive Server Enterprise にアクセスする場合は、以下の条件に従ってください。

次に、Adaptive Server Enterprise の重要な制限をいくつか示します。

- 動的 SQL は、テンポラリ・ストアド・プロシージャを使用して実装されます。したがって、ストアド・プロシージャの制限を継承します。
- ロング可変長バイナリ・データ型とロング可変長文字データ型はサポートされません。
- 定義により、カーソルは 1 つだけの select 文に関連付けられます。つまり、Client-Library のカーソルを宣言するストアド・プロシージャに指定できるのは、1 つの select 文だけです。
- ストアド・プロシージャは、一部のデータ型のパラメータをサポートしないことがあります。ストアド・プロシージャのパラメータに関する制限の詳細については、Adaptive Server Enterprise のマニュアルを参照してください。
- イベント・ノーティフィケーションはサポートされません。
- メッセージ・コマンドはサポートされません。
- ローカライゼーションの POSIX ロケール・メソッドはサポートされません。

## サポートされるクライアント／サーバ機能

特定の接続でサポートされるクライアントとサーバの機能を確認するために、アプリケーションは ct_capability を呼び出します。ct_capability の *value* パラメータは、その機能が有効かどうかについての情報を返します。

特に次の点について知ることができます。

- サポートされるデータ型
- 有効な要求のタイプ

詳細については、ct_capability のリファレンス・ページを参照してください。

# text および image データの処理

text と image は、大きな text 値または image 値を保持するための Adaptive Server Enterprise のデータ型です。text データ型は、2,147,483,647 バイトまでの出力可能な文字を保持します。image データ型は、2,147,483,647 バイトまでのバイナリ・データを保持します。

text および image の値は、非常に大きくなることがあるため、実際にはデータベース・テーブルに保管されません。その代わりに、text 値または image 値への「**テキスト・ポインタ**」がテーブルに保管されます。

競合するアプリケーションどうしが、データベースに加えた変更を互いに消去しあうことがないように、個々の text カラムや image カラムにはタイムスタンプが関連付けられています。このタイムスタンプを、「**テキスト・タイムスタンプ**」と呼びます。

Client-Library は、text または image カラムに対するテキスト・ポインタおよびテキスト・タイムスタンプを I/O 記述子構造体 CS_IODESC に記録します。カラムに対する I/O 記述子にも、カラム名およびカラムのデータ型を含めて、そのカラムについてのその他の情報も含まれています。

CS_IODESC 構造体の詳細については、「CS_IODESC 構造体」(99 ページ) を参照してください。

## text カラムまたは image カラムの取得

アプリケーションが text カラムまたは image カラムを取得する場合、次の 2 つの方法があります。

- アプリケーションによって、カラムの選択、カラムのバインド、ローのフェッチを行います。つまり、アプリケーションが text または image カラムの取得や処理を行う場合は、他のタイプのカラムの取得や処理を行う場合と同じ方法で行います。
- アプリケーションは、カラムの選択、ct_fetch による結果ローのループ、ct_get_data による text カラムと image カラムのデータの取得を実行します。アプリケーションはこの方法を使用して、大きすぎて効率的にバインドできない text 値や image 値を処理します。

## *text* および *image* 値をフェッチする *ct_get_data* の使用

ct_bind を呼び出してバインドした最後のカラム以降にあるカラムだけが、ct_get_data を使用して利用可能です。

たとえば、アプリケーションが、すべてテキストであるカラムを 4 つ選択し、1 番目と 3 番目のカラムをプログラム変数にバインドする場合、アプリケーションは、ct_get_data を使用して 2 番目のカラムに含まれるテキストを取得することはできません。しかし、ct_get_data を使用して、4 番目のカラムにあるテキストを取得することは可能です。select 文を制御するアプリケーションは、select リストを再度整えることができるので、text と image カラムを最後にすることができます。

ct_get_data を使用して text 値または image 値を取得するために、アプリケーションでは、次のような手順で処理を進めます。

1. text または image カラムを含む結果セットを生成するコマンドを実行します。

   アプリケーションは、言語コマンド、RPC コマンドまたは動的 SQL コマンドを使用して、text カラムまたは image カラムを含む結果セットを生成します。

   たとえば、*pubs2* データベースの *au_pix* テーブルにある *pic* カラムに、著者のピクチャが含まれているとします。このピクチャを取得するには、アプリケーションは次の言語コマンドを実行します。

   ```
   ct_command(cmd, CS_LANG_CMD,
            "select pic from au_pix",
            CS_NULLTERM, CS_UNUSED);
    ct_send(cmd);
   ```

2. text カラムまたは image カラムを含む結果セットを処理します。

   アプリケーションは、ct_fetch を使用して、結果セットに含まれているローをループします。このループの内部では、バインドされていない text または image カラムそれぞれに対して、アプリケーションは次のようなことができます。

   - ループで ct_get_data を呼び出してそのカラムの text または image データを取得します。
   - アプリケーションは、ct_get_info を呼び出して、後でカラムを更新する I/O 記述子を取得します。

   ほとんどのアプリケーションで、次のようなプログラム構造が使用されます。

   ```
   while ct_fetch is returning rows
       process any bound columns
   ```

```
            for each unbound text or image column
                while ct_get_data is returning data
                    process the data
                end while
                ct_data_info to get the column's CS_IODESC
            end for
        end while
```

一方、バインドされていない text カラムまたは image カラムそれぞれに対して、アプリケーションは次のようなことが可能です。

- *buflen* パラメータを 0 に設定して、ct_get_data を呼び出すことができます。その結果、データは何も返りませんが、カラムの I/O 記述子をリフレッシュします。
- ct_get_data を呼び出して、カラムの I/O 記述子を取得できます。この構造体の *total_txtlen* フィールドは、text 値または image 値の全長を表しています。
- ct_get_data を必要なだけ呼び出して、値を取得します。

この方法には、アプリケーションが text 値または image 値を取得する前に、その全長を調べられるという利点があります。

## *text* または *image* カラムの更新

次の 3 とおりの方法で text カラムまたは image カラムを更新します。

- update 言語コマンドのテキスト内に新しい値を埋め込みます。この方法の利点は簡単であることです。また、この方法の欠点は、アプリケーションが一度に値全体を送信する必要があることです。この方法は、非常に大きな ( つまり、プログラムが領域を割り付けることができないくらい大きな ) カラムの場合には適していない場合があります。Adaptive Server Enterprise は値がコマンド・テキスト内に埋め込まれ、コマンド・パラメータとして渡されないことを要求します。Adaptive Server Enterprise は、text 型または image 型のパラメータを許可しません。
- ct_command ルーチンを使用してデータ送信コマンドを開始し、ct_send_data ルーチンを使用して値をまとまりとして送信します。この方法では、プログラムのバッファ領域よりも大きい値の処理が可能ですが、より複雑です。オペレーティング・システム・ファイルなどの外部ソースから値をまとまりとして読み込むアプリケーションの場合には、この方法の方が埋め込む方法よりも適していることがあります。

- *type* パラメータを CS_SEND_DATA_NOCMD に設定し、ct_command ルーチンを呼び出すことによって、データ送信コマンドを開始します。その後、クライアント・アプリケーションは、データ送信コマンドを使用してサーバのバルク・ハンドラに text データのみまたは image データのみを送信できます。サーバでバルク・イベントが発生すると、送信する合計バイト数を示す 4 バイトのフィールドに続き、text または image データが送信されます。バルク・ハンドラは srv_text_info を使用して予想される合計バイト数を読み込み、srv_get_data を使用してデータを読み取ります。

---

**注意** text カラムまたは image カラムをこの方法で更新するには、ct_connect ルーチンを呼び出す前に、CS_SENDDATA_NOCMD 接続プロパティを設定してください (ct_command *type* パラメータの CS_SEND_DATA_NOCMD 設定と混同しないでください )。

---

アプリケーションが (ct_data_info ルーチンを使用して ) 更新しようとしているカラムの、現在の I/O 記述子設定がすでに定義されている場合は、ct_send_data ルーチンを使用して text カラムまたは image カラムのみを更新します。I/O 記述子の設定は、CS_IODESC 構造体内に指定されています ( 「CS_IODESC 構造体」(99 ページ ) を参照 )。Adaptive Server Enterprise は、更新を実行するために正しく初期化された I/O 記述子を要求し、クライアント・アプリケーションは要求された I/O 記述子設定をサーバから取得する必要があります。

## I/O 記述子設定の取得

アプリケーションは ct_data_info ルーチンを呼び出して、I/O 記述子設定を取得します。リリース 11.0 以降の Adaptive Server Enterprise を使用している場合、アプリケーションは、サーバのグローバル変数を使用して、I/O 記述子を直接選択します。

ct_data_info ルーチンを呼び出して現在の I/O 記述子を取得するには、アプリケーションはまず取得するローの中の取得するカラムを選択する必要があります。select 文から返されたロー結果を処理している間に、アプリケーションは次の手順に従って I/O 記述子を取得します。

1. ct_fetch を呼び出して、対象となるローをフェッチします。
2. ct_get_data を呼び出してカラムの値を取得し、そのカラムの I/O 記述子を更新します。*buflen* を 0 に設定して ct_get_data を呼び出し、カラムのデータを取得せずに I/O 記述子をリフレッシュします。
3. ct_data_info を呼び出して、I/O 記述子を取得します。

リリース 11.0 からは、Adaptive Server Enterprise は I/O 記述子フィールドを設定するための代替手段を提供しています。一度に 1 つの text カラムまたは image カラムの更新だけを行うアプリケーションは、この方法を使用します。「グローバル変数を使用した text または image カラムの更新」(337 ページ ) を参照してください。

## 新しいカラム値の送信

一度カラム値の現在の I/O 記述子を持つと、アプリケーションは、次のようにして更新します。

1. ct_command を呼び出して、データ送信コマンドを開始します。
2. 必要であれば、I/O 記述子を修正します。ほとんどのアプリケーションでは、*locale*、*total_txtlen*、または *log_on_update* のフィールドの値だけを変更します。
3. ct_get_data を呼び出して、カラム値の I/O 記述子を設定します。I/O 記述子構造体の *textptr* フィールドは、データ送信操作のターゲット・カラムを識別します。
4. ループで ct_send_data を呼び出し、text または image 値全体を書き込みます。ct_send_data のそれぞれの呼び出しで、text または image 値の一部を書き込みます。
5. ct_send を呼び出して、そのコマンドを送信します。
6. ct_results を呼び出して、コマンドの結果を処理します。text または image カラムの更新により、その値の新しいテキスト・タイムスタンプである、1 つのパラメータを含むパラメータ結果セットを生成します。アプリケーションが、このカラム値を再度更新する予定がある場合、ct_data_info ( 前述のステップ 3) を呼び出して再更新のための I/O 記述子を設定する前に、新しいタイムスタンプを保存して、カラム値の CS_IODESC にコピーしてください。

ほとんどのアプリケーションは、次のようなプログラム構造体を使用して、text カラムまたは image カラムを更新します。

```
ct_con_alloc to allocate connection1 and connection2
ct_cmd_alloc to allocate cmd1 and cmd2

ct_command(cmd1) to select columns
      (including text) from table
ct_send to send the command
while ct_results returns CS_SUCCEED
      (optional) ct_res_info to get description of result set
      (optional) ct_describe to get descriptons of columns
      (optional) ct_bind if binding any columns
```

```
    while ct_fetch(cmd1) returns rows
        for each text column
            /* Retrieve the current CS_IODESC for the column */
            if you want the column’s data, loop on ct_get_data
                    while there’s data to retrieve
            if you don’t want the column’s data, call
                    ct_get_data once with buflen of 0 to
                    refresh the CS_IODESC
            ct_data_info(cmd1, CS_GET) to get the CS_IODESC

            /* Update the column */
            ct_command(cmd2) to initiate a send-data command
            if necessary, modify fields in the CS_IODESC
            ct_data_info(cmd2, CS_SET) to set the CS_IODESC for
                    the column
            while there is data to send
                ct_send_data(cmd2) to send a chunk of data
            end while
            ct_send(cmd2) to send the send-data command
            ct_results(cmd2) to process the send-data results
        end for
    end while
end while
```

## *text* データと *image* データの部分更新

Open Client は、text カラムと image カラムの部分更新をサポートしています。部分更新では、置換、削除、または挿入する text フィールドまたは image フィールドの部分を指定できます。また、フィールド全体を修正するのではなく、その部分だけを更新することもできます。

---

**注意**　現在、Adaptive Server Enterprise は、text データと image データの部分更新をサポートしていません。ただし、Open Server は、text データと image データの部分更新をサポートしています。詳細については、『Open Server Server-Library/C リファレンス・マニュアル』の「第 2 章 トピック」を参照してください。

---

部分更新を実行するには、ct_data_info を使用して *iotype*、*delete_length*、*offset* を設定します。*delete_length* の値、および ct_send_data を呼び出してサーバに渡されるデータの値により、部分更新の動作が決まります。

| delete_length | text データ | サーバの動作 |
|---|---|---|
| 0 | 提供される | text データを *offset* に挿入する。 |
| != 0 | 提供される | *offset* から開始して、text データで *delete_length* バイト分のデータを上書きする。 |
| != 0 | 提供されない | *offset* から開始して、データで *delete_length* バイト分のデータを削除する。 |
| NULL | 提供される / されない | *offset* から text カラムまたは image カラムの末尾までのデータを削除する。 |

### *ct_send_data* による部分更新の送信

ct_send_data ルーチンを使用して、部分的に更新されたデータを送信できます。部分的に更新されたデータの場合、ct_send_data は、Transact-SQL updatetext 文を作成します。そして、複数の ct_send_data 呼び出しを使用して、データがまとめて送信されます。updatetext の構文は次のとおりです。

```
updatetext table_name.column_name text_pointer
   {NULL | offset} {NULL | delete_length} [with_log]
```

**注意**　updatetext 文が作成されるのは、CS_IODESC 構造体の *iotype* 値が CS_IOPARTIAL に設定されている場合だけです。

### unitext データの処理

使用しているクライアント・アプリケーションが 2 バイトの Unicode データ型を使用して部分更新を実行する場合、文字の分断を回避するため、確実に偶数のバイト数を送信する必要があります。ct_send_data の *buflen* パラメータと CS_IODESC の *total_txtlen* フィールドを使用して、Unicode データの長さをバイト単位で指定できます。Unitext の場合、*offset* 値と *delete_length* 値を文字数として指定すると同時に、*total_txtlen* をバイト単位で指定してください。その他のデータ型の場合には、*offset*、*delete_length*、*total_txtlen* をバイト単位で指定します。

# *text* カラムや *image* カラムを含むテーブルへのデータ設定

挿入するデータ値のサイズによって、アプリケーションで text カラムまたは image カラムがあるテーブルにデータを設定する方法は異なります。

## *text* および *image* 値が小さい場合

ほとんどのアプリケーションは、100K より小さい text または image 値を insert 文に埋め込みます。

```
insert blurbs values ("486-29-1786", "If Chastity
     Locksley didn't exist, this troubled...")
insert au_pix values ("486-29-1786", 0x67f44c...,
     "ICT", "30220", "626", "635")
```

## *text* および *image* 値が大きい場合

Adaptive Server Enterprise テーブルに 100K より大きな text 値または image 値を格納するには、次の方法をおすすめします。

1. text 値または image 値以外のすべてのデータをそのローへ挿入 (insert) します。
2. text または image カラムの値を NULL に設定して、ローを更新 (update) します。null 値を持つ text カラムまたは image カラムのローが、有効なテキスト・ポインタを持つのは、その null 値が update 文で明示的に入力された場合だけであるので、この手順は重要です。
3. カラムの I/O 記述子設定を取得します。これは、次の 2 とおりの方法で実行できます。
    - 対象の text カラムまたは image カラムを選択し、そのカラムの値を取得してから ct_data_info を呼び出します。この方法の詳細については、「I/O 記述子設定の取得」(331 ページ ) を参照してください。この方法は、text または image データ型をサポートするすべての Sybase Server で機能します。
    - Adaptive Server Enterprise が提供した text グローバル変数および image グローバル変数を使用します。この方法の詳細については、「グローバル変数を使用した text または image カラムの更新」(337 ページ ) を参照してください。この方法を使用するには、Adaptive Server Enterprise 11.0 以降が必要です。

4 「新しいカラム値の送信」(332 ページ) の説明に従って、カラムを更新します。

## *text* および *image* を更新するためのサーバのグローバル変数

Adaptive Server Enterprise リリース 11.0 以降には、text と image をサポートするための専用のグローバル変数があります。これらの変数は次のとおりです。

| 変数 | 説明 | データ型 |
|---|---|---|
| *@@textptr* | あるプロセスで挿入されたり更新されたりする最後の text カラムまたは image カラムのテキスト・ポインタ。 | binary(16) |
| *@@textts* | *@@textptr* によって参照されるカラムのタイムスタンプ。 | varbinary(8) |
| *@@textcolid* | *@@textptr* が参照するカラムの ID。 | tinyint |
| *@@textdbid* | *@@textptr* が参照するカラムのオブジェクトが入っているデータベースの ID。 | smallint |
| *@@textobjid* | *@@textptr* が参照するカラムが入っているオブジェクトの ID。 | int |

Adaptive Server Enterprise への各接続には、これらの変数の接続自体のインスタンスがあります。これらの変数は、セッションの開始時には 0 に設定されます。Adaptive Server Enterprise は、次のようにしてこれらの変数を更新します。

- セッション中にアプリケーションが text データ・カラムまたは image データ・カラムを更新する場合。同じロー内の複数の text または image カラムが同時に更新される場合、変数は更新されたローの中の最後の text または image カラムを記述します。
- アプリケーションが NULL 以外の text 値または image 値を含むローを挿入する場合。複数の NULL 以外の text または image カラムを同じローに挿入する場合、変数はローの最後の NULL 以外の text または image カラムを記述します。

## グローバル変数を使用した *text* または *image* カラムの更新

一度に 1 つの text カラムまたは image カラムの挿入または更新だけを行うアプリケーションでは、text グローバル変数または image グローバル変数は、ct_send_data ルーチンを呼び出して text カラムまたは image カラムを更新するのに必要な I/O 記述子フィールドに必要事項を書き込む代替手段を提供します。

「text または image カラムの更新」(330 ページ ) で説明されているように、アプリケーションが対象の text カラムまたは image カラムを選択し、取得してからでないと、ct_data_info を呼び出して I/O 記述子のフィールドを設定できません。ct_data_info ルーチンを呼び出す代わりに、アプリケーションは、text グローバル変数と image グローバル変数を取得して、I/O 記述子内に書き込むためにこれらの値を使用します。このためには、アプリケーションは次のタスクを実行する必要があります。

- 言語コマンドを実行してカラムを更新するか、新しいローを挿入します。
- 同じ言語バッチでは、text グローバル変数と image グローバル変数の現在の値を選択します。
- 結果を処理して、I/O 記述子フィールドにその値を取得します。

大部分のアプリケーションは次の手順に従って、サーバの text グローバル変数と image グローバル変数を使用して text または image の更新を実行します。

1. ct_command ルーチンを呼び出して、update 文または insert 文を含む言語コマンドを開始します。これによって、Adaptive Server Enterprise は text グローバル変数と image グローバル変数に適切な I/O 記述子の値を指定します。カラムを更新する言語コマンドをダミー値に送信して、タスクを実行します。コマンドは、CS_IODESC 構造体の *textptr* フィールド、*timestamp* フィールド、*name* フィールドに適した Transact-SQL の式を選択する必要があります。たとえば my_table テーブルのキーが int_col カラムである場合、適切な言語コマンドのバッチは次のとおりです。

```
update my_table set text_col = NULL
         where int_col = 23
 if @@rowcount != 0
    select @@textptr,
           @@textts,
           colname = object_name(@@textobjid) +
           '.' + col_name(@@textobjid,
                          @@textcolid,
                           @@textdbid)
```

新しいローを挿入する場合、update コマンドは、同じバッチ内で insert コマンドの後に指定されるか、insert コマンドに置き換えられます。text カラムまたは image カラムに NULL を指定する insert コマンドの場合は、その後に、カラムを NULL に更新する update コマンドを指定してください。update コマンドを指定しない場合、サーバはそのカラムを記述する *@@text* 変数を更新しません。text カラムまたは image カラムに NULL 以外の値を指定する insert コマンドの場合は、その後に続けて update コマンドを指定する必要はありません。

前述の例の update コマンドが成功した場合、I/O 記述子に必要な情報が選択されて、3 つのカラムとして返されます。1 番目のカラムはテキスト・ポインタ値、2 番目のカラムは新しいタイムスタンプ、3 番目のカラムは *table_name.column* という形式の文字列です。

2 ct_results ループ内で結果を処理します。

選択された式が通常結果ロー ( 結果タイプ CS_ROW_RESULT) として返されます。アプリケーションは、ct_bind ルーチンを呼び出して CS_IODESC 構造体のフィールドに値をバインドし、ct_fetch ルーチンを呼び出してその値を取得します。アプリケーションは、次の表に従って構造体のフィールドをバインドします。

| **CS_IODESC フィールド** | **カラム値** |
|---|---|
| *timestamp*、*timestamplen* | ct_bind ルーチンを呼び出して *timestamp* を *@@textts* にバインドし、*timestamplen* のアドレスを ct_bind の *copied* パラメータとして渡す。 |
| *textptr*、*textptrlen* | ct_bind ルーチンを呼び出して *@@textptr* にバインドし、*textptrlen* のアドレスを ct_bind の *copied* パラメータとして渡す。 |
| *name*、*namelen* | ct_bind ルーチンを呼び出して *name* を返される値にバインドする。<br>`object_name(@@textobjid) + "." + col_name(@@textobjid, @@textcolid, @@textdbid)`<br>ct_bind ルーチンの呼び出しでは、*name* フィールドにバインドするときに、ct_bind ルーチンの *copied* パラメータとして *namelen* のアドレスを渡す。 |

3 次のように、残りのすべての I/O 記述子を適切な値に設定します。

```
iodesc->iotype = CS_IODATA;
 iodesc->usertype = 0;
 iodesc->offset = 0;
 iodesc->locale = (CS_LOCALE *) NULL;
```

```
iodesc->total_txtlen = length_of_new_value;
iodesc->log_on_update = CS_TRUE; /* or CS_FALSE */
```

これらの手順を実行すると、「新しいカラム値の送信」(332 ページ ) で説明されているように、アプリケーションは新しい text 値または image 値を送信する準備ができます。

# データ型のサポート

Client-Library は、広範囲のデータ型をサポートします。これらのデータ型は、Open Client CS-Library と Server-Library で使用されます。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。

表 2-36 に、Open Client/Server のデータ型定数、対応する C のデータ型、対応する Adaptive Server Enterprise のデータ型を示します。

また、表 2-36 に続いて、データ型を操作する上で役立つ Open Client ルーチンと各データ型について示してあります。

データ型の詳細については、『Open Client Client-Library/C プログラマーズ・ガイド』の「第 3 章 Open Client/Server データ型の使い方」を参照してください。

## データ型の概要

表 2-36 に、Open Client/Server のデータ型定数、対応する C のデータ型、対応する Adaptive Server Enterprise のデータ型を示します。

**表 2-36：データ型の概要**

| 型カテゴリ | 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 |

| 型カテゴリ | 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 |
|---|---|---|---|---|
| 文字型 | 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<br>univarchar |
| XML 型 | 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 |
| 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 |

| 型カテゴリ | 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 |
|---|---|---|---|---|
| LOB ( ラージ・オブジェクト ) ロケータ型 | CS_TEXTLOCATOR_TYPE | ロケータ型 | CS_LOCATOR | text_locator |
| | CS_IMAGELOCATOR_TYPE | ロケータ型 | CS_LOCATOR | image_locator |
| | CS_UNITEXTLOCATOR_TYPE | ロケータ型 | CS_LOCATOR | unitext_locator |
| text 型および image 型 | CS_TEXT_TYPE | テキスト型 | CS_TEXT | text |
| | CS_UNITEXT_TYPE | 可変長文字型 | CS_UNITEXT | unitext |
| | CS_IMAGE_TYPE | イメージ型 | CS_IMAGE | image |

## データ型を操作するルーチン

Open Client CS-Library は、データ型を操作するために役立ついくつかのルーチンを提供しています。その中には、次のものがあります。

- cs_calc：decimal、money、numeric の各データ型の算術演算を実行します。
- cs_cmp：datetime、decimal、money、numeric の各データ型を比較します。
- cs_convert：あるデータ型から別のデータ型へデータ値を変換します。
- cs_dt_crack：マシンが読み取れる日時値をユーザがアクセスできるフォーマットに変換します。
- cs_dt_info：各言語固有の日時情報を設定または取得します。
- cs_strcmp：2 つの文字列を比較します。

これらの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』に記載されています。

## Open Client のデータ型

この項では、Open Client のデータ型とその定義について説明します。

## binary 型

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

- 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_BINARY は、次のように定義されます。

  ```
  typedef unsigned char     CS_BINARY;
  ```

  **警告！** CS_LONGBINARY および CS_VARBINARY は、Adaptive Server Enterprise データ型には対応していません。

- 一部の Open Server アプリケーションは、CS_LONGBINARY をサポートしています。アプリケーションは、CS_DATA_LBIN 機能を使用して、Open Server 接続が CS_LONGBINARY をサポートしているかどうかを調べます。サポートしている場合は、結果データ項目が記述されるときに ct_describe が 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_LONGBINARY の定義は次のとおりです。

  ```
  typedef unsigned char    CS_LONGBINARY;
  ```

- CS_VARBINARY は、どの Adaptive Server Enterprise データ型にも対応しません。このため、Open Client ルーチンは、CS_VARBINARY_TYPE を返しません。CS_VARBINARY は、Open Client で C 以外のプログラミング言語が使用できるように提供されています。一般的なクライアント・アプリケーションでは、CS_VARBINARY を使用しません。

  CS_VARBINARY は、次のように定義されます。

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

  各パラメータの説明は、次のとおりです。

  - *len* はバイナリ配列の長さです。
  - *array* は配列そのものです。

CS_VARBINARY 変数は、可変長値の格納に使用されますが、固定長型とみなされます。これは一般的に、アプリケーションが Client-Library に CS_VARBINARY 変数の長さを提供する必要がないことを意味しています。たとえば、ct_bind は、CS_VARBINARY 変数にバインドすると、*datafmt−>maxlength* の値を無視します。

## 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。この型は、0 または 1 というサーバのビット値 ( またはブール値 ) を保持するものです。他の型をビットに変換すると、0 以外の値はすべて 1 に変換されます。

```
typedef unsigned char    CS_BIT;
```

## character 型

Open Client には、CS_CHAR、CS_LONGCHAR、CS_VARCHAR、CS_UNICHAR の 4 つの character 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_CHAR は次のように定義されます。

  ```
  typedef char    CS_CHAR;
  ```

  **警告！** CS_LONGCHAR と CS_VARCHAR は、Adaptive Server Enterprise データ型には対応していません。CS_VARCHAR は、Adaptive Server Enterprise の varchar データ型には対応していません。

- CS_LONGCHAR は、Adaptive Server Enterprise のデータ型には対応していませんが、Open Server アプリケーションの中には、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、CS_DATA_LCHAR 機能を使用して、Open Server 接続が CS_LONGCHAR をサポートしているかどうかを調べます。サポートしている場合は、結果データ項目を記述しているときに、ct_describe が CS_LONGCHAR を返します。

  CS_LONGCHAR 値の最大長は、2,147,483,647 バイトです。CS_LONGCHAR は、次のように定義されます。

```
typedef unsigned char      CS_LONGCHAR;
```

- 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、Open Client で C 以外のプログラミング言語が使用できるように提供されています。一般的なクライアント・アプリケーションは、CS_VARCHAR を使用しません。

  CS_VARCHAR は、次のように定義されます。

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

  各パラメータの説明は、次のとおりです。

  - *len* は文字列の長さです。
  - *str* は文字列です。*str* は null で終了しないので注意してください。

    CS_VARCHAR 変数は、可変長値の格納に使用されますが、固定長型とみなされます。これは一般的に、アプリケーションが Client-Library に CS_VARBINARY 変数の長さを提供する必要がないことを意味しています。たとえば、ct_bind は、CS_VARCHAR 変数にバインドすると、*datafmt−>maxlength* の値を無視します。

  - CS_UNICHAR は、Adaptive Server Enterprise の unichar 固定幅および univarchar 可変幅のデータ型に対応します。CS_UNICHAR は、CS_CHAR データ型が使用されるどの場所でも使用できる共有 C プログラミング・データ型です。CS_UNICHAR データ型は、2 バイトの Unicode UTF-16 フォーマットで文字データを保存します。

    CS_UNICHAR の定義は次のとおりです。

    ```
    typedef unsigned char       CS_UNICHAR;
    ```

## XML 型

CS_XML は、Adaptive Server Enterprise の xml 可変長データ型に直接対応します。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。

CS_XML の定義は次のとおりです。

```
typedef unsigned char       CS_XML
```

## 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、4 バイトまたは 8 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。これらのデータ型は、8 バイトの binary 値を保持します。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。

Open Client アプリケーションは、CS-Library ルーチン cs_dt_crack を使用して、日時構造体から日付部分 ( 年、月、日など ) を抽出します。

- CS_DATE は、Adaptive Server Enterprise の date データ型に対応します。有効な CS_DATE 値の範囲は、0001 年 1 月 1 日から、9999 年 12 月 31 日までです。CS_DATE は、次のように定義されます。

  ```
  typedef CS_INT CS_DATE; /* 4-byte date type*/
  ```

- CS_TIME は、Adaptive Server Enterprise の time データ型に対応します。有効な CS_TIME 値の範囲は、12:00:00.000 から 11:59:59:999 までで、精度は 300 分の 1 秒 (3.33 ms) です。CS_TIME は、次のように定義されます。

  ```
  typedef CS_INT CS_TIME; /* 4-byte time type*/
  ```

- CS_DATETIME は、Adaptive Server Enterprise の datetime データ型に対応しています。CS_DATETIME の有効値は 1753 年 1 月 1 日から 9999 年 12 月 31 日の範囲で、精度は 1 秒の 300 分の 1 (3.33 ミリ秒 ) です。CS_DATETIME は、次のように定義されます。

  ```
  typedef struct _cs_datetime
   {
      CS_INT              dtdays;
      CS_INT   dttime;
   } CS_DATETIME;
  ```

各パラメータの説明は、次のとおりです。

- *dtdays* は 1900 年 1 月 1 日から数えた日数です。
- *dttime* は、深夜 0 時からの 300 分の 1 秒の数です。

- CS_DATETIME4 は、Adaptive Server Enterprise の *smalldatetime* データ型に対応しています。CS_DATETIME4 の有効値は、1900 年 1 月 1 日から 2079 年 6 月 6 日の範囲で、精度は 1 分です。
  CS_DATETIME は、次のように定義されます。

```
typedef struct _cs_datetime4
 {
     CS_USHORT    days;
     CS_USHORT    minutes;
 } CS_DATETIME4;
```

  各パラメータの説明は、次のとおりです。

  - *days* は 1900 年 1 月 1 日から数えた日数です。
  - *minutes* は、深夜 0 時からの分数です。

- 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_BIGDATETIME の定義は、*cstypes.h* にあります。

```
typedef CS_UBIGINT CS_BIGDATETIME;
```

- 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_BIGTIME の定義は、*cstypes.h* にあります。

```
typedef CS_UBIGINT CS_BIGTIME;
```

- 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 ( エンディアン ) 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。

## 整数値型

Open Client でサポートされる整数型は、CS_TINYINT、CS_SMALLINT、CS_INT、CS_BIGINT、CS_USMALLINT、CS_UINT、CS_UBIGINT の 7 つです。

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。

```
typedef unsigned char      CS_TINYINT;
typedef short              CS_SMALLINT;
typedef int                CS_INT;
typedef long long          CS_BIGINT;
typedef unsigned char      CS_USMALLINT;
typedef unsigned int       CS_UINT;
typedef unsigned long long CS_UBIGINT;
```

## real、float、numeric、decimal 型

- CS_REAL は、Adaptive Server Enterprise のデータ型 *real* に対応しています。これは、C 言語の *float* 型として実装されています。

```
typedef float                  CS_REAL;
```

---

**注意**　6 桁精度の bigint データ型または ubigint データ型を *real* のデータ型に変換する場合、次の最大値および最小値に注意してください。

- -9223370000000000000.0 < bigint < 9223370000000000000.0
- 0 < ubigint < 18446700000000000000.0

これらの範囲外の値により、オーバフロー・エラーが発生します。

---

- CS_FLOAT は、Adaptive Server Enterprise の *float* データ型に対応しています。これは、C 言語の *double* 型として実装されています。

```
typedef double                 CS_FLOAT;
```

**注意**　15 桁精度の bigint データ型または ubigint データ型を *float* のデータ型に変換する場合、次の最大値および最小値に注意してください。

- -9223372036854770000.0 < bigint < 9223372036854770000.0
- 0 < ubigint < 18446744073709500000.0

これらの範囲外の値により、オーバフロー・エラーが発生します。

- 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、精度と位取りを持った数値に対して、プラットフォームに依存しないサポートを提供します。

**警告！** Client-Library プログラムおよび ESQL/C プログラムの出力パラメータ CS_DECIMAL と CS_NUMERIC については、精度と位取りを定義してから、ct_param を呼び出してください。これは、出力パラメータには定義時に関連付けられる値がなく、無効な精度と位取りが関連付けられているためです。値を初期化できない場合は、精度または位取りが無効であることを示すメッセージが出力されます。

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。

```
typedef struct_cs_numeric
 {
     CS_BYTE        precision;
     CS_BYTE        scale;
     CS_BYTE        array[CS_MAX_NUMLEN];
 } CS_NUMERIC;

 typedef CS_NUMERIC           CS_DECIMAL;
```

各パラメータの説明は、次のとおりです。

- *precision* は、256 を基数とする数体系で対応する桁数によって表現できる 10 進数の桁数の最大値です。たとえば、10 進数の精度が 4 桁 (0 〜 9999) である場合、256 を基数とする数の 2 桁で表現できます。現時点での *precision* の有効値は 1 から 77 までです。デフォルトの精度は 18 です。CS_MIN_PREC、CS_MAX_PREC、CS_DEF_PREC は、最小、最大、デフォルトの精度の値をそれぞれ定義します。
- *array* は、数値の 256 を基数とする表現です。インデックス 0 でのバイトは、符号を表します。ただし、0 ( バイト値 00000000) は正の数、1 ( バイト値 0000001) は負の数を表します。残りのバイト 1 〜 n は、リトル・エンディアン順序での 256 を基数とする数を表します。インデックス 1 でのバイトは、最上位バイトです。

  *array* で使用されるバイトの数は、選択された数値の精度を基本にしています。マッピングは、使用される配列の長さに対する数値の精度を基にして行われます。
- *scale* は小数点以下の最大桁数です。現時点での *scale* の有効値は 0 から 77 までです。デフォルトの位取りは 0 です。CS_MIN_SCALE、CS_MAX_SCALE、CS_DEF_SCALE は、最小、最大、デフォルトの位取りの値をそれぞれ定義します。
- *scale* は、*precision* 以下でなければなりません。

  CS_DECIMAL 型は、精度 (*precision*) と位取り (*scale*) について CS_NUMERIC 型と同じデフォルト値を使用します。

## money 型

Open Client は、CS_MONEY と CS_MONEY4 の 2 つの money 型をサポートします。これらのデータ型は、それぞれ 8 バイトと 4 バイトの money 型を保持できます。

- CS_MONEY は、Adaptive Server Enterprise の *money* データ型に対応しています。有効な CS_MONEY 値の範囲は、+$922,337,203,685,477.5807 から -$922,337,203,685,477.5807 までの間です。

```
typedef struct _cs_money
 {
     CS_INT      mnyhigh;
     CS_UINT     mnylow;
 } CS_MONEY;
```

- CS_MONEY4 は、Adaptive Server Enterprise の *smallmoney* データ型に対応しています。CS_MONEY4 の有効値は、-$214,748.3648 から +$214,748.3647 の範囲です。

```
typedef struct _cs_money4
 {
     CS_INT    mny4;
 } CS_MONEY4;
```

## text 型および image 型

Open Client は、text データ型 CS_TEXT と CS_UNITEXT、および image データ型 CS_IMAGE をサポートします。

- CS_TEXT は、Adaptive Server Enterprise データ型 text に対応しています。このデータ型は、2,147,483,647 バイトまでの出力可能な文字データが入った可変長カラムを表します。CS_TEXT は符号なしの文字型として定義されます。

```
typedef unsigned char          CS_TEXT;
```

- CS_UNITEXT は、Adaptive Server Enterprise の unitext 可変長データ型に直接対応します。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 2 バイトの Unicode UTF-16 形式でコード化されることです。CS_UNITEXT は、CS_TEXT が使用されるどの場所でも使用できます。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_UNITEXT の定義は次のとおりです。

```
typedef unsigned short     CS_UNITEXT;
```

- CS_IMAGE は、Adaptive Server Enterprise データ型 image に対応しています。このデータ型は、2,147,483,647 バイトまでのバイナリ・データが入った可変長カラムを表します。CS_IMAGE は符号なしの文字型として定義されます。

```
typedef unsigned char    CS_IMAGE;
```

## LOB ロケータ・データ型

CS_LOCATOR は opaque データ型であり、使用可能な内部データ構造の public typedef はありません。

CS-Library ルーチンの cs_locator_alloc を使用して、CS_LOCATOR データ型構造体を割り付けます。

CS-Library ルーチンの cs_locator を使用して、プリフェッチされたデータ、サーバ内の LOB の全長、ロケータのポインタの文字表現などの情報を CS_LOCATOR データ型構造体から取得します。

CS-Library ルーチンの cs_locator_drop を使用して、CS_LOCATOR データ型構造体の割り付けを解除します。

このデータ型の使用法については、*locator.c* のサンプルを参照してください。

## Open Client のユーザ定義データ型

標準的な Open Client 型セット以外のデータ型を使用する必要があるアプリケーションは、ユーザ定義データ型を作成できます。

Client-Library アプリケーションは、次のようにユーザ定義データ型を宣言して、ユーザ定義データ型を作成します。

```
typedef char CODE_NAME;
```

Open Client ルーチン ct_bind および cs_set_convert は、整数記号定数を使用してデータ型を識別するため、アプリケーションがユーザ定義データ型に型定数を宣言しておくと便利です。ユーザ定義データ型は、CS_USERTYPE 以上の値として定義します。

```
#define CODE_NAME_TYPE (CS_USERTYPE + 2)
```

一度ユーザ定義データ型が作成されると、アプリケーションでは次の処理が可能になります。

- cs_set_convert を呼び出して、標準的な Open Client 型とユーザ定義データ型との間で変換するカスタム変換ルーチンをインストールできます。
- 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。

変換ルーチンがインストールされると、アプリケーションは、次のようにサーバ結果をユーザ定義データ型へバインドします。

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

カスタム変換ルーチンは、必要に応じて、ct_fetch ( 変換を指定する ct_bind が事前に呼び出された場合 ) および cs_convert によって透過的に呼び出されます。

---

**注意**　Open Client ユーザ定義データ型と、Adaptive Server Enterprise ユーザ定義データ型を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。

---

## ランタイム設定ファイルの使い方

デフォルトでは、Client-Library は Open Client/Server ランタイム設定ファイルを読み込んで、次のようなランタイム値を設定します。

- プロパティ値 ( 通常は、ct_config または ct_con_props を呼び出して設定 )
- サーバ・オプション値 ( 通常は、接続がオープンされてから ct_option を呼び出して設定 )
- サーバ機能 ( 通常は、接続がオープンされる前に ct_capability を呼び出して設定 )
- ランタイム設定ファイル内でのみ設定でき、ct_config 呼び出しまたは ct_con_props 呼び出しでは設定できないプロパティ
- デバッグ・オプション ( 通常は、ct_debug を呼び出して設定 )

設定ファイルを読み込んで、これらの設定を適用させるアプリケーションは、ct_con_props ルーチンまたは前述のその他のルーチンへの呼び出しを数回省きます。コードを再コンパイルしなくてもアプリケーションのランタイム設定を変更できることも利点です。

環境変数 SYBOCS_DBVERSION を使用すると、DB-Library バージョン・レベルを実行時に外部から設定できます。これは、dbsetversion を呼び出した後、アプリケーション・コードを変更することにより実行できます。

---

**注意**　Sybase の外部設定ファイルがある場合は、次のセクションを追加して bcp と isql を有効にします。

[BCP]

[isql]

---

## 外部設定の有効化

次のプロパティは、Open Client/Server ランタイム設定ファイルの使用を制御します。

- *CS_EXTERNAL_CONFIG* － コンテキスト・レベルでこのプロパティが CS_TRUE に設定されている場合、ct_init は Open Client/Server ランタイム設定ファイルからデフォルトの Client-Library コンテキスト・プロパティ値を読み込みます。

  コンテキスト・レベルでは、デフォルトの Open Client/Server ランタイム設定ファイルがある場合に、CS_EXTERNAL_CONFIG プロパティがデフォルトで CS_TRUE に設定されます。それ以外の場合には、CS_FALSE に設定されます。外部設定ファイルの名前は、CS_CONFIG_FILE プロパティを使用して決定されます。アプリケーションは cs_config ルーチンを呼び出して、コンテキスト・レベルのデフォルトを上書きできます。

  接続レベルでは、割り付けられた接続構造体が親コンテキストから CS_EXTERNAL_CONFIG プロパティを継承します。接続レベルで CS_EXTERNAL_CONFIG プロパティが CS_TRUE に設定されている場合、ct_connect ルーチンは、Open Client/Server ランタイム設定ファイルからデフォルトの接続プロパティ、機能、サーバ・オプション、デバッグ・オプションを読み込みます。

- *CS_CONFIG_FILE* － Open Client/Server ランタイム設定ファイルの名前とロケーションを指定します。CS_CONFIG_FILE プロパティは、cs_config ルーチンを呼び出してコンテキスト・レベルで設定するか、ct_con_props ルーチンを呼び出して接続レベルで設定します。デフォルト値は NULL で、プラットフォーム固有のデフォルト・ファイルが使用されることを意味します。

- UNIX プラットフォームの場合、デフォルトの設定ファイルは *$SYBASE/$SYBASE_OCS/config/ocs.cfg* です。

  *$SYBASE* は Sybase インストール・ディレクトリへのパスです。このパスは SYBASE 環境変数の値として指定されます。*$SYBASE_OCS* は Open Client/Server のサブディレクトリです。このパスは SYBASE_OCS 環境変数の値として指定されます。

- Windows プラットフォームの場合、デフォルトの設定ファイルは *%SYBASE% ¥%SYBASE_OCS% ¥ini ¥ocs.cfg* です。

  *%SYBASE%* は Sybase インストール・ディレクトリへのパスです。このパスは SYBASE 環境変数の値として指定されます。*%SYBASE_OCS%* は Open Client/Server のサブディレクトリです。このパスは SYBASE_OCS 環境変数の値として指定されます。

その他のプラットフォームの、デフォルトの Open Client/Server ランタイム設定ファイルの名前については、『Open Client/Server 設定ガイド』を参照してください。

- *CS_CONFIG_BY_SERVERNAME* － ct_connect がファイル・セクション名として、接続の CS_APPNAME プロパティの値とサーバ名のどちらを使用するかを制御します。デフォルトでは、CS_APPNAME プロパティの値が使用されます。ct_con_props を呼び出して、接続レベルで CS_CONFIG_BY_SERVERNAME を設定できます。

  たとえば、接続で外部設定が有効であり、アプリケーション名が「Monthly Report」、*server_name* の値が「FinancialDB」である場合は、次のようになります。

  - CS_CONFIG_BY_SERVERNAME プロパティが CS_FALSE に設定されている場合、ct_connect ルーチンは [Monthly Report] ラベルが付いているセクションを検索します。
  - CS_CONFIG_BY_SERVERNAME プロパティが CS_TRUE に設定されている場合、ct_connect ルーチンは [FinancialDB] ラベルが付いているセクションを検索します。

  **注意**　CS_CONFIG_BY_SERVERNAME が CS_TRUE に設定されていない場合は、CS_SERVERNAME を外部設定ファイルで変更できません。

設定セクションを使用してサーバ名とアプリケーション名を変更します。これによって、管理者は、アプリケーション内にハードコードされているサーバ名またはアプリケーション名を上書きできます。たとえばアプリケーションがセクション名 FinancialDB を読み込むように設定されている場合、セクションは次のような内容になっている可能性があります。

```
[FinancialDB]
 CS_APPNAME = "Monthly Financial Report"
 CS_SERVERNAME = "Dev_FinancialDB" ; redirect to
                                   ; development
                                   ; server
```

- *SYBOCS_CFG* － 使用する設定ファイルを指定します。このファイルにより、次の場所にあるデフォルトの *ocs.cfg* ファイルが上書きされます。
    - UNIX の場合：*$SYBASE/$SYBASE_OCS/config/ocs.cfg*
    - Windows の場合：*%SYBASE% ¥%SYBASE_OCS% ¥ini ¥ocs.cfg*
- *CS_APPNAME* － コンテキスト・レベルで、ファイルのどのセクションから値が読み込まれるかを指定します。アプリケーションは cs_config ルーチンを呼び出して、コンテキスト・レベルで CS_APPNAME プロパティを設定します。アプリケーションがコンテキスト構造体の CS_APPNAME プロパティを設定しない場合、ct_init ルーチンは [DEFAULT] ラベルが付いているセクションを検索します。接続レベルでは、外部設定が有効になっていて CS_CONFIG_BY_SERVERNAME プロパティのデフォルト値が CS_FALSE である場合、ct_connect ルーチンは CS_APPNAME プロパティによって示されたファイル・セクションを読み込みます。

## Open Client/Server ランタイム設定ファイルの構文

Open Client/Server ランタイム設定ファイルはテキスト・ファイルです。このファイルはセクションに分割されています。それぞれのセクションは、角カッコ ([ ]) で囲まれたセクション名で始まり、次のセクション名または [the end of the file] のいずれかで終わります。

それぞれのセクションには、次に示すように 1 つまたは複数の設定が含まれています。

```
[section name]
 keyword = value ; comment
 keyword = value
```

```
; more comments
[next section name]
... and so forth ...
```

一般的に、ファイル内でサポートされているすべてのキーワードは、Client-Library/C プログラム内でプロパティ、オプション、または機能を識別する記号定数の名前と一致します。ただし、設定ファイル内ですべてのプロパティを設定できるわけではありません。あるキーワードがサポートされていない場合は、その設定は無視されます。

構文は、次のとおりです。

- ; − コメント行を示します。
- [section_name] − セクション名は角カッコ ([]) で囲まれています。Open Client/Server 設定ファイルには DEFAULT という名前のセクションがあります。-x オプションを使用してコンパイルされたアプリケーションでは、アプリケーション名がセクション名として使用されます。-e オプションを使用してコンパイルされたアプリケーションの場合、そのサーバ名がセクション名に使用されます。複数のセクションで使用される設定が入っているセクションの名前は、どのような名前でも使用できます。次の例は、「GENERIC」という任意の名前のセクションと、そのセクションがその他のセクションにどのように指定されているかを示します。

  ```
  [GENERIC]
   CS_OPT_ANSINULL=CS_TRUE
  [APP_PAYROLL]
   include=GENERIC
   CS_CAP_RESPONSE=CS_RES_NOSTRIPBLANKS
  [APP_HR]
   include=GENERIC
   CS_OPT_QUOTED_IDENT=CS_TRUE
  ```

- entry_name=entry_value
  - エントリ値には、整数や文字列など、どのような値でも指定できます。エントリ値の行が '¥'<newline> で終了する場合、そのエントリ値は次の行に続きます。
  - エントリ値の最初と最後にあるスペースはトリムされます。
  - エントリ値の最初または最後にスペースが必要な場合は、そのスペースを二重引用符で囲んでください。

- 二重引用符で始まるエントリは、二重引用符で終了する必要があります。引用符で囲まれた文字列の中に連続する 2 つの二重引用符がある場合、それは値文字列の中の 1 つの二重引用符を表します。二重引用符内に改行文字が検出された場合は、文字どおりにその値の一部とみなされます。
- エントリ名とセクション名は、アルファベット ( 大文字と小文字の両方 )、0 ～ 9 の数字、次に示す区切り文字 ( デリミタ ) から構成できます。

  ! " # $ % & ' ( ) * + , - . / : ; < > ? @ ¥ ^ _ ` { | } ~

  角カッコ、スペース、等号記号 (=) はサポートされません。ただし、最初の文字は必ずアルファベットにしてください。
- エントリ名とセクション名は大文字と小文字が区別されます。
- Include=*earlier_section*

  セクションに include というエントリがある場合、以前に定義されたセクションの内容全体がこのセクション内にコピーされるとみなされます。つまり、以前のセクションで定義されたプロパティはこのセクションによって継承されます。

  ---

  **注意**　インクルードされるセクションは、別のセクションにインクルードされる前に定義されている必要があります。これによって、設定ファイルの解析を単一のパスで行うことができ、再帰的にインクルードされたディレクティブ ( 命令語 ) を検出する必要がなくなります。

  ---

  インクルードされたセクションが順に別のセクションをインクルードする場合、エントリ値の順序は、インクルードされたセクションの「入れ子の深い方から浅い方へ」の検索によって定義されます。

  セクションには、そのセクション自体への参照をインクルードすることはできません。つまり、以前に定義されたセクションをインクルードする必要があるので再帰は不可能です。また、定義されていないセクションをインクルードすることはできません。

あるセクションで定義されたすべての直接エントリ値は、別のセクションからインクルードされた可能性のある値を置き換えます。次の例では、CS_OPT_ANSINULL は APP.PAYROLL アプリケーションでは false に設定されます。

---

**注意**　include 文の位置は、この規則には影響を与えません。

---

```
[GENERIC]
 CS_OPT_ANSINULL=CS_TRUE
[APP_PAYROLL]
 CS_OPT_ANSINULL=CS_FALSE
 include=GENERIC
```

- C プログラム内のエントリの値が記号定数を取る場合、これらの定数の名前が有効な値になります。次に例を示します。

```
CS_NETIO = CS_SYNC_IO
```

- C プログラム内のエントリの値が整数値を取る場合、有効値は整数値の有効範囲と一致します。次に例を示します。

```
CS_TIMEOUT = 60
```

- C プログラム内のエントリの値がブール値を取る場合、有効値は CS_TRUE と CS_FALSE です。次に例を示します。

```
CS_DIAG_TIMEOUT = CS_TRUE
```

- C プログラム内のエントリの値が文字列を取る場合、その文字列はファイルに直接入力されます。次に例を示します。

```
CS_USERNAME = winnie
```

文字列の値の中には、引用符で囲む必要があるものもあります。文字列に先行スペースまたはコメント文字のセミコロン (;) がある場合、値を引用符で囲む必要があります。null 文字列の値の場合も、間にスペースを入れない連続した引用符で示す必要があります。次に例を示します。

```
CS_APPNAME = "   Monthly report; Financials   "
 CS_PASSWORD = ""
```

長い文字列の値の場合は、行末に円記号 (¥) を指定すると、次の行に続けることができます。引用符で囲まれた文字列中の行末に円記号が指定されていない場合は、値の一部として読み込まれます。最後に、円記号をエスケープ文字ではなく本来の文字として、文字列値の中に指定する場合は、円記号を 2 つ続けて指定してください。

- C プログラム内のプロパティの値が CS_CHAR 以外のデータ型へのポインタを取る場合、外部設定ではプロパティを設定できません。唯一の例外は CS_LOCALE キーワードです。このキーワードには、CS_LOCALE 構造体を設定してコンテキストまたは接続の CS_LOC_PROP プロパティとしてインストールする場合と同じ効果があります。たとえば次の行は、コンテキストまたは接続にフランス語のロケールを割り当てます。

  ```
  CS_LOCALE = french
  ```

- 1 つのセクション内にキーワードが 2 回示される場合は、最初の定義だけが使用されます。
- 次の構文を使用して、あるセクションに別のセクション内のキーワードをインクルードできます。

  ```
  [section name]
   include = previous section name
  ... more settings ...
  ```

  インクルードされたセクション名のもとで定義された設定は、そのセクションをインクルードするセクションで定義されます。インクルードされた設定は、インクルードするセクション内で明示的に設定することで必ず置き換えられます。たとえば次の [Finance] セクションでは、CS_TIMEOUT が 30 に定義されます。DEFAULT セクションからインクルードされた設定は、次のように明示的に設定して置き換えられます。

  ```
  [DEFAULT]
   CS_TIMEOUT = 45

   [Finance]
   include = DEFAULT
   CS_TIMEOUT = 30
  ```

## ランタイム設定ファイルのキーワード

以降の表は、Client-Library のランタイム動作の設定に有効なキーワードと、それぞれのキーワードに対して認められている値を示します。

## ローカライゼーションのキーワード

次の表は、コンテキストまたは接続のロケールを設定するためのキーワードを示します。これらの設定は、コンテキストまたは接続の CS_LOC_PROP プロパティを設定するのに必要な呼び出しを置き換えます。

| キーワード | 有効値 |
|---|---|
| CS_LOCALE | ホスト・プラットフォームのロケール・ファイル内で定義されるロケール名 |

## コンテキスト・プロパティまたは接続プロパティのキーワード

アプリケーションが外部設定を要求した場合、アプリケーションが ct_init ルーチンと ct_connect ルーチンを呼び出すと、これらのルーチンはそれぞれ設定ファイルのセクションを読み込みます。

ct_init があるセクションを読み込むときにコンテキスト・プロパティが設定されている場合、ct_con_props を呼び出して同じプロパティを設定すると、すでに設定されていた内容が上書きされます。

ct_connect ルーチンがセクションを読み込むときにプロパティが設定されている場合、ct_con_props ルーチンを呼び出して同じプロパティを設定すると、次のどちらかになります。

- ct_connect ルーチンの前に ct_con_props を呼び出した場合は、ファイルの値に置き換えられます。
- ct_connect ルーチンの後に ct_con_props を呼び出した場合は、ファイルの値が置き換えられます。

たとえば、設定セクション内で設定される CS_USERNAME プロパティと CS_PASSWORD プロパティの値は、アプリケーション・コード内にハードコードされた値を必ず上書きします。これは、アプリケーションが、これらのプロパティを設定してから ct_connect ルーチンを呼び出すためです。

表 2-37 に、コンテキスト・プロパティまたは接続プロパティを設定するキーワードを示します。それぞれのプロパティが何を制御するかについては、「プロパティ」(208 ページ ) を参照してください。

**表 2-37：プロパティを設定する設定ファイルのキーワード**

| キーワード | 読み込まれるルーチン | 有効値 |
|---|---|---|
| CS_ANSI_BINDS | ct_init、ct_connect | CS_TRUE または CS_FALSE |
| CS_APPNAME | ct_connect | 文字列 |
| CS_ASYNC_NOTIFS | ct_connect | CS_TRUE または CS_FALSE |
| CS_BULK_LOGIN | ct_connect | CS_TRUE または CS_FALSE |
| CS_CON_KEEPALIVE | ct_connect | CS_TRUE または CS_FALSE |
| CS_CON_TCP_NODELAY | ct_connect | CS_TRUE または CS_FALSE |
| CS_DIAG_TIMEOUT | ct_connect | CS_TRUE または CS_FALSE |
| CS_DISABLE_POLL | ct_init、ct_connect | CS_TRUE または CS_FALSE |
| CS_DS_COPY | ct_connect | CS_TRUE または CS_FALSE |
| CS_DS_DITBASE | ct_connect | 文字列 |
| CS_DS_FAILOVER | ct_connect | CS_TRUE または CS_FALSE |
| CS_DS_PASSWORD | ct_connect | 文字列 |
| CS_DS_PRINCIPAL | ct_connect | 文字列 |
| CS_DS_PROVIDER | ct_connect | 文字列 |
| CS_DS_RAND_OFFSET | ct_config、ct_con_props | CS_TRUE または CS_FALSE |
| CS_EXPOSE_FMTS | ct_init、ct_connect | CS_TRUE または CS_FALSE |
| CS_EXTENDED_ENCRYPT_CB | ct_connect | CS_TRUE または CS_FALSE |
| CS_EXTRA_INF | ct_init、ct_connect | CS_TRUE または CS_FALSE |
| CS_HAFAILOVER | ct_config、ct_con_props | CS_TRUE または CS_FALSE |
| CS_HIDDEN_KEYS | ct_init、ct_connect | CS_TRUE または CS_FALSE |
| CS_HOSTNAME | ct_connect | 文字列 |
| CS_IFILE | ct_init | 文字列 |
| CS_LOGIN_TIMEOUT | ct_init | 整数値 |
| CS_LOOP_DELAY | ct_connect | 整数値 |
| CS_MAX_CONNECT | ct_init | 整数値 |
| CS_NETIO | ct_init、ct_connect | CS_SYNC_IO、CS_ASYNC_IO、または CS_DEFER_IO |
| CS_NOAPI_CHK | ct_init | CS_TRUE または CS_FALSE |

| キーワード | 読み込まれるルーチン | 有効値 |
|---|---|---|
| CS_NO_TRUNCATE | ct_init | CS_TRUE または CS_FALSE |
| CS_NOINTERRUPT | ct_init | CS_TRUE または CS_FALSE |
| CS_PACKETSIZE | ct_connect | 整数値 |
| CS_PASSWORD | ct_connect | 文字列 |
| CS_PROP_EXTENDEDFAILOVER | ct_config、ct_con_props | CS_TRUE または CS_FALSE |
| CS_PROP_REDIRECT | ct_config、ct_con_props | CS_TRUE または CS_FALSE |
| CS_RETRY_COUNT | ct_connect | 整数値 |
| CS_SEC_APPDEFINED | ct_connect | CS_TRUE または CS_FALSE |
| CS_SEC_CHALLENGE | ct_connect | CS_TRUE または CS_FALSE |
| CS_SEC_CHANBIND | ct_init、ct_connect | CS_TRUE または CS_FALSE |
| CS_SEC_CONFIDENTIALITY | ct_init、ct_connect | CS_TRUE または CS_FALSE |
| CS_SEC_CREDTIMEOUT | ct_init、ct_connect | 正の整数値または CS_NO_LIMIT |
| CS_SEC_DATAORIGIN | ct_init、ct_connect | CS_TRUE または CS_FALSE |
| CS_SEC_DELEGATION | ct_init、ct_connect | CS_TRUE または CS_FALSE |
| CS_SEC_DETECTREPLAY | ct_init、ct_connect | CS_TRUE または CS_FALSE |
| CS_SEC_DETECTSEQ | ct_init、ct_connect | CS_TRUE または CS_FALSE |
| CS_SEC_ENCRYPTION | ct_connect | CS_TRUE または CS_FALSE |
| CS_SEC_EXTENDED_ENCRYPTION | ct_connect | CS_TRUE または CS_FALSE |
| CS_SEC_INTEGRITY | ct_init、ct_connect | CS_TRUE または CS_FALSE |
| CS_SEC_KEYTAB | ct_connect | 文字列 |
| CS_SEC_MECHANISM | ct_init、ct_connect | 文字列 |
| CS_SEC_MUTUALAUTH | ct_init、ct_connect | CS_TRUE または CS_FALSE |
| CS_SEC_NETWORKAUTH | ct_init、ct_connect | CS_TRUE または CS_FALSE |

| キーワード | 読み込まれるルーチン | 有効値 |
|---|---|---|
| CS_SEC_NON_ENCRYPTION_RETRY | ct_connect | CS_TRUE または CS_FALSE |
| CS_SEC_SERVERPRINCIPAL | ct_connect | 文字列 |
| CS_SEC_SESSTIMEOUT | ct_init、ct_connect | 正の整数値または CS_NO_LIMIT |
| CS_TDS_VERSION | ct_connect | CS_TDS_40、CS_TDS_42、CS_TDS_46、CS_TDS_50 |
| CS_TEXTLIMIT | ct_init、ct_connect | 整数値または CS_NO_LIMIT |
| CS_TIMEOUT | ct_init | 整数値 |
| CS_USERNAME | ct_connect | 文字列値 |

## サーバ・オプションのキーワード

表 2-38 は、接続のサーバ・オプションを設定するためのキーワードを示します。

アプリケーションが ct_options ルーチンを呼び出すと、設定ファイル内の同じ設定が上書きされます。

表 2-38 は、サーバ・オプションを設定するためのキーワードを示します。それぞれのオプションが何を制御するかについては、「オプション」(200 ページ ) を参照してください。

**表 2-38：サーバ・オプションのための設定ファイルのキーワード**

| キーワード | 有効値 |
|---|---|
| CS_OPT_ANSINULL | CS_TRUE または CS_FALSE |
| CS_OPT_ANSIPERM | CS_TRUE または CS_FALSE |
| CS_OPT_ARITHABORT | CS_TRUE または CS_FALSE |
| CS_OPT_ARITHIGNORE | CS_TRUE または CS_FALSE |
| CS_OPT_AUTHOFF | 文字列値 |
| CS_OPT_AUTHON | 文字列値 |
| CS_OPT_CHAINXACTS | CS_TRUE または CS_FALSE |
| CS_OPT_CURCLOSEONXACT | CS_TRUE または CS_FALSE |
| CS_OPT_CURREAD | 文字列値 |
| CS_OPT_CURWRITE | 文字列値 |
| CS_OPT_DATEFIRST | CS_OPT_SUNDAY、CS_OPT_MONDAY、CS_OPT_TUESDAY、CS_OPT_WEDNESDAY、CS_OPT_THURSDAY、CS_OPT_FRIDAY、CS_OPT_SATURDAY |
| CS_OPT_DATEFORMAT | CS_OPT_FMTMDY、CS_OPT_FMTDMY、CS_OPT_FMTYMD、CS_OPT_FMTYDM、CS_OPT_FMTMYD、CS_OPT_FMTDYM |
| CS_OPT_FIPSFLAG | CS_TRUE または CS_FALSE |
| CS_OPT_FORCEPLAN | CS_TRUE または CS_FALSE |
| CS_OPT_FORMATONLY | CS_TRUE または CS_FALSE |
| CS_OPT_GETDATA | CS_TRUE または CS_FALSE |
| CS_OPT_IDENTITYOFF | 文字列値 |
| CS_OPT_IDENTITYON | 文字列値 |
| CS_OPT_ISOLATION | CS_OPT_LEVEL0、CS_OPT_LEVEL1、CS_OPT_LEVEL3 |
| CS_OPT_NOCOUNT | CS_TRUE または CS_FALSE |
| CS_OPT_NOEXEC | CS_TRUE または CS_FALSE |
| CS_OPT_PARSEONLY | CS_TRUE または CS_FALSE |
| CS_OPT_QUOTED_IDENT | CS_TRUE または CS_FALSE |
| CS_OPT_RESTREES | CS_TRUE または CS_FALSE |
| CS_OPT_ROWCOUNT | 整数値 |

| キーワード | 有効値 |
|---|---|
| CS_OPT_SHOWPLAN | CS_TRUE または CS_FALSE |
| CS_OPT_SORTMERGE | CS_TRUE または CS_FALSE |
| CS_OPT_STATS_IO | CS_TRUE または CS_FALSE |
| CS_OPT_STATS_TIME | CS_TRUE または CS_FALSE |
| CS_OPT_TEXTSIZE | 整数値 |
| CS_OPT_TRUNCIGNORE | CS_TRUE または CS_FALSE |

## サーバ機能のキーワード

外部から設定できるのは、応答機能だけです。応答機能がファイルから読み込まれる場合、アプリケーションがその接続に対して ct_capability ルーチンを呼び出して、設定したどのような応答機能も置き換えます。

次の表は、接続に対してサーバ機能を設定するためのキーワードを示します。それぞれの機能が何を制御するかの詳細については、ct_capability のリファレンス・ページを参照してください。

| キーワード | 有効値 |
|---|---|
| CS_CAP_RESPONSE | クライアントが受信を望まない機能をカンマで区切ったリスト。リスト値には、表 3-6 (402 ページ ) に示されている記号定数を指定できる。 |

## プロパティを排他的に設定するキーワード

これらのキーワードは、次のように、ランタイム設定ファイル内でのみ指定でき、cs_config または ct_con_props を使用して設定できないプロパティを設定します。

**表 2-39：プロパティを排他的に設定するキーワード**

| キーワード | 説明 | 使用できる場所 |
|---|---|---|
| CS_SANITIZE_DISC_APPNAME | 名前のないアプリケーション (CS_APPNAME がアプリケーションで明示的に設定されていない場合 ) に対して検出されたアプリケーション名 ( オペレーティング・システムから取得された実行プログラム名 ) をそのまま使用するか、大文字か小文字に変換してから使用するかを指定します。<br>有効な値<br>• CS_CNVRT_UPPERCASE － 検出された名前を大文字に変換してから使用する。<br>• CS_CNVRT_LOWERCASE － 検出された名前を小文字に変換してから使用する。<br>• CS_CNVRT_NOTHING ( デフォルト ) － 検出された名前をそのまま使用する。 | [DEFAULT] セクションのみ |
| CS_USE_DISCOVERED_APPNAME | ランタイム設定ファイルを名前のないアプリケーション (CS_APPNAME がアプリケーションで明示的に設定されていない場合 ) のアプリケーション固有設定に対して解析するかどうか、および検出された設定をアプリケーションに適用するかどうかを指定する。オペレーティングシステムから取得された実行プログラム名は、CS_APPNAME と設定され、ランタイム設定ファイルの解析に使用されます。<br>有効な値<br>• CS_TRUE － 設定ファイルからのアプリケーション固有の設定を解析し、適用します。<br>• CS_FALSE ( デフォルト ) － アプリケーション固有の設定について、設定ファイルを解析しません。 | [DEFAULT] セクションのみ |

## *ct_debug* オプションのキーワード

次の表に、接続に対してデバッグ・オプションを設定するためのキーワードを示します。

CS_DBG_FILE キーワードは、Client-Library がテキスト形式のデバッグ情報を書き込むファイルの名前を指定します。Client-Library は、要求されたデバッグ情報だけを記録します。

デバッグ情報は、その他のキーワードを使用して要求されます。これらのキーワードは、ct_debug ルーチンの *flag* パラメータのビットマスクに対応しています。これらのデバッグ・フラグの意味については、ct_debug ルーチンのリファレンス・ページを参照してください。

**表 2-40：デバッグ・オプションのための設定ファイルのキーワード**

| キーワード | 有効値 |
| --- | --- |
| CS_DBG_FILE | テキスト・フォーマットのデバッグ情報を保管するファイルの名前を指定する文字列。 |
| CS_DEBUG | カンマで区切られたデバッグ・フラグのリストを指定する文字列。 |
| CS_PROTOCOL_FILE | バイナリ・フォーマットのデバッグ情報を保管するファイルの名前を指定する文字列。 |

CS_DEBUG は、CS_DBG_FILE ファイルに書き込まれるデータを指定します。値として、ct_debug の *flag* パラメータのビットマスクに対応するフラグのリストを指定できます。各デバッグ・フラグの意味については、『Open Client Client-Library/C リファレンス・マニュアル』の ct_debug のリファレンス・ページを参照してください。

使用可能なフラグは次のとおりです。

- CS_DBG_ALL
- CS_DBG_API_LOGCALL
- CS_DBG_API_STATES
- CS_DBG_ASYNC
- CS_DBG_DIAG
- CS_DBG_ERROR
- CS_DBG_MEM
- CS_DBG_NETWORK
- CS_DBG_PROTOCOL
- CS_DBG_PROTOCOL_FILE
- CS_DBG_PROTOCOL_STATES
- CS_DBG_SSL

第 3 章

# ルーチン

この章では、Client-Library のルーチンについて説明します。


| ルーチン | 説明 | ページ |
|---|---|---|
| ct_bind | サーバ結果をプログラム変数にバインドする。 | 371 |
| ct_br_column | ブラウズ・モード select によって生成されたカラムについての情報を取得する。 | 384 |
| ct_br_table | ブラウズ・モード・テーブルについての情報を返す。 | 385 |
| ct_callback | Client-Library コールバック・ルーチンをインストールまたは取得する。 | 387 |
| ct_cancel | コマンドまたはコマンドの結果をキャンセルする。 | 392 |
| ct_capability | クライアント／サーバ機能を設定または取得する。 | 397 |
| ct_close | サーバ接続をクローズする。 | 406 |
| ct_cmd_alloc | CS_COMMAND 構造体を割り付ける。 | 409 |
| ct_cmd_drop | CS_COMMAND 構造体の割り付けを解除する。 | 411 |
| ct_cmd_props | コマンド構造体プロパティを設定または取得する。コマンドを再送するアプリケーションによって使用される。 | 412 |
| ct_command | 言語、パッケージ、RPC、メッセージ、またはデータ送信コマンドを開始する。 | 418 |
| ct_compute_info | 計算結果の情報を取得する。 | 428 |
| ct_con_alloc | CS_CONNECTION 構造体を割り付ける。 | 432 |
| ct_con_drop | CS_CONNECTION 構造体の割り付けを解除する。 | 434 |
| ct_con_props | 接続構造体のプロパティを設定または取得する。 | 436 |
| ct_config | コンテキスト・プロパティを設定または取得する。 | 452 |
| ct_connect | サーバに接続する。 | 461 |
| ct_cursor | Client-Library カーソル・コマンドを開始する。 | 466 |

| ルーチン | 説明 | ページ |
|---|---|---|
| ct_data_info | データ I/O 記述子構造体を定義または取得する。 | 492 |
| ct_debug | デバッグ用のライブラリ・オペレーションを管理する。 | 496 |
| ct_describe | 結果データの記述を返す。 | 501 |
| ct_diag | インライン・エラー処理を管理する。 | 507 |
| ct_ds_dropobj | ディレクトリ・オブジェクトによって使用されているメモリを解放する。 | 515 |
| ct_ds_lookup | ディレクトリ検索オペレーションを開始またはキャンセルする。 | 516 |
| ct_ds_objinfo | ディレクトリ・オブジェクトに関する情報を取得する。 | 523 |
| ct_dynamic | 動的 SQL コマンドを開始する。 | 530 |
| ct_dyndesc | 動的 SQL 記述子領域でオペレーションを実行する。 | 538 |
| ct_dynsqlda | SQLDA 構造体に関する処理を行う。 | 548 |
| ct_exit | Client-Library を終了する。 | 556 |
| ct_fetch | 結果データをフェッチする。 | 559 |
| ct_get_data | データのまとまりをサーバから読み込む。 | 566 |
| ct_getformat | 結果カラムに関連するサーバのユーザ定義フォーマット文字列を返す。 | 572 |
| ct_getloginfo | TDS ログイン応答情報を CS_CONNECTION 構造体から新しく割り付けた CS_LOGINFO 構造体に転送する。 | 573 |
| ct_init | アプリケーション・コンテキストの Client-Library を初期化する。 | 575 |
| ct_keydata | キー・カラムの内容を指定または抽出する。 | 580 |
| ct_labels | 接続のセキュリティ・ラベルを定義またはクリアする。 | 583 |
| ct_options | サーバのクエリ処理オプションの値を設定、取得、またはクリアする。 | 585 |
| ct_param | サーバ・コマンドの入力パラメータに値を提供する。 | 590 |
| ct_poll | 非同期オペレーションの完了とレジスタード・プロシージャ・ノーティフィケーションを確認するために接続のポーリングを実行する。 | 601 |
| ct_recvpassthru | サーバから TDS (Tabular Data Stream) パケットを受信する。 | 607 |

| ルーチン | 説明 | ページ |
|---|---|---|
| ct_remote_pwd | サーバ間接続に使用するパスワードを定義またはクリアする。 | 609 |
| ct_res_info | 現在の結果セット情報またはコマンド情報を取得する。 | 612 |
| ct_results | 結果データを処理対象として設定する。 | 619 |
| ct_scroll_fetch | スクロール可能なフェッチ関数。 | 629 |
| ct_send | コマンドをサーバに送信する。 | 639 |
| ct_send_data | text データまたは image データのまとまりをサーバに送信する。 | 644 |
| ct_sendpassthru | TDS (Tabular Data Stream) パケットをサーバに送信する。 | 656 |
| ct_setloginfo | TDS ログイン応答情報を CS_LOGINFO 構造体から CS_CONNECTION 構造体に転送する。 | 658 |
| ct_setparam | ct_send がサーバ・コマンドのための入力パラメータ値を読み込むソース変数を指定する。 | 660 |
| ct_wakeup | 接続の完了コールバックを呼び出す。 | 673 |



# ct_bind

**説明**

サーバ結果をプログラム変数にバインドします。

**構文**

```
CS_RETCODE ct_bind(cmd, item, datafmt, buffer, copied, indicator)

CS_COMMAND   cmd;
CS_INT       item;
CS_DATAFMT   *datafmt;
CS_VOID      *buffer;
CS_INT       *copied;
CS_SMALLINT  *indicator;
```

**パラメータ**

*cmd*

クライアント／サーバ・オペレーションを管理する CS_COMMAND 構造体を指すポインタです。

*item*

バインドするカラム、パラメータ、またはステータスの番号を表す整数です。

カラムをバインドする場合、*item* は、そのカラムのカラム番号です。select 文の select リストにある最初のカラムのカラム番号は 1、2 番目のカラムのカラム番号は 2、以下同様に続きます。

計算カラムをバインドする場合、*item* は、その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。最初に返されるカラムの番号は 1 です。

リターン・パラメータをバインドする場合、*item* は、パラメータ番号になります。ストアド・プロシージャによって返される最初のパラメータは、番号 1 のパラメータです。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 文に最初に指定されたときの順序で返されます。この順序は、そのストアド・プロシージャを呼び出す RPC 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際には、リターン・パラメータ以外はカウントしないでください。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。

ストアド・プロシージャ・リターン・ステータスをバインドする場合、リターン・ステータス結果セットにあるステータスは、1 つだけなので、*item* は 1 でなければなりません。

すべてのバインドをクリアするには、*item* を CS_UNUSED に、*datafmt*、*buffer*、*copied*、*indicator* を NULL にして渡してください。

*datafmt*

送信先変数または配列を記述する CS_DATAFMT 構造体のアドレスです。ct_bind は、**datafmt* の内容をコピーしてから返します。ct_bind が返した後、Client-Library は *datafmt* 内でアドレスを参照しません。

次の表に、ct_bind によって使用される **datafmt* のフィールドと、そのフィールドについての一般情報を示します。ct_bind は、使用しないフィールドを無視します。

**表 3-1：ct_bind に対する CS_DATAFMT のフィールド設定**

| フィールド名 | 使用する場合 | 設定内容 |
|---|---|---|
| *name* | 未使用。 | 適用されない。 |
| *namelen* | 未使用。 | 適用されない。 |
| *datatype* | すべてのタイプの結果をバインドする場合。 | 送信先変数のデータ型を表わす型定数 (CS_xxx_TYPE)。<br>有効な定数については、「データ型のサポート」(339 ページ ) を参照。Open Client のユーザ定義データ型については、cs_set_convert を使用してユーザ提供の変換ルーチンがインストールされている場合のみ有効。*datatype* が Open Client ユーザ定義データ型の場合、ct_bind は、*count* を除くすべての CS_DATAFMT フィールドを検証しない。<br>ct_bind は、広い範囲の型変換をサポートしているので、*datatype* は、サーバによって返される型と異なっていてもかまわない。たとえば、CS_FLOAT_TYPE の送信先型を指定することによって、CS_MONEY 結果は CS_FLOAT プログラム変数にバインドされる。適切なデータ変換が自動的に行われる。ct_bind は、cs_convert がサポートする変換をすべて実行できる。サポートしている変換については、Open Client/Open Server の cs_convert のリファレンス・ページを参照。<br>*datatype* が CS_BOUNDARY_TYPE または CS_SENSITIVITY_TYPE の場合、プログラム変数 **buffer* の型は CS_CHAR でなければならない。 |
| *format* | 結果項目を文字、バイナリ、text、または image の送信先変数にバインドする場合。それ以外は、CS_FMT_UNUSED。 | 次の記号のビットマスク。<br>文字および text の送信先だけの場合、データを null で終了させるには、CS_FMT_NULLTERM。<br>変数の全長までスペースを埋め込ませるには、CS_FMT_PADBLANK。<br>文字、バイナリ、text、および image の送信先の場合、変数の全長まで null を埋め込ませるには、CS_FMT_PADNULL。<br>すべての型の送信先について、フォーマット情報が提供されない場合には、CS_FTM_UNUSED。 |

| フィールド名 | 使用する場合 | 設定内容 |
|---|---|---|
| *maxlength* | すべてのタイプの結果を非固定長型にバインドする場合。<br>固定長型にバインドする場合、*maxlength* は無視される。 | **buffer* 送信先変数の長さ。*buffer* が配列を指す場合は、*maxlength* に配列の 1 つの要素の長さを設定する必要がある。<br>文字またはバイナリ送信先にバインドする場合、*maxlength* には、NULL ターミネータなどの特別な終了バイトに必要な領域を含めて、送信先変数の全長を指定しなければならない。<br>**buffer* が結果データ項目を保持できるほど大きくないことを *maxlength* が示している場合、フェッチの時点で、ct_fetch は大きすぎる結果項目を廃棄し、ローの残りの項目をフェッチして、CS_ROW_FAIL を返す。これが発生した場合、**buffer* の内容は定義されていない。 |
| *scale* | 数値または 10 進数の送信先にバインドする場合のみ。 | 送信先の変数での小数点の右側の最大桁数。<br>変換元データと変換先データの型が同じ場合、変換先データが変換元データから *scale* の値を取得する必要があることを示すために、*scale* を CS_SRC_VALUE に設定できる。<br>*scale* は、*precision* 以下でなければならない。 |
| *precision* | 数値または 10 進数の送信先にバインドする場合のみ。 | 送信先の変数で表すことができる 10 進数の最大桁数。<br>変換元データと変換先データの型が同じ場合には、変換先データが変換元データから *precision* の値を取得することを示すには、*precision* を CS_SRC_VALUE に設定する。<br>*precision* は、*scale* 以上でなければならない。 |
| *status* | 未使用。 | 適用されない。 |

| フィールド名 | 使用する場合 | 設定内容 |
|---|---|---|
| *count* | すべてのタイプの結果をバインドする場合。 | ct_fetch または ct_scroll_fetch の呼び出しごとにプログラム変数にコピーする結果ロー数。<br>*count* が使用可能なローの数よりも大きい場合は、使用可能なローだけがコピーされる ( 通常ローおよびカーソル・ロー結果セットだけが複数のローを持つことに注意する必要がある )。<br>*count* は、1 つの例外を除いて、結果セットのすべてのカラムで同じ値を持たなければならない。つまり、アプリケーションでは、0 と 1 の *count* を混在させることができる。<br>*count* が 0 の場合、1 ローがフェッチされる。<br>ct_scroll_fetch 呼び出しの場合、*count* 値は CS_CURSOR_ROWS 以上でなくてはならない。予期しない結果になる可能性があるため *count* 値が CS_CURSOR_ROWS を下回ることはできない。 |
| *usertype* | 未使用。 | 適用されない。 |
| *locale* | すべてのタイプの結果をバインドする場合。 | *buffer 送信先変数についてのロケール情報を持つ CS_LOCALE 構造体を指すポインタ。<br>その変数がカスタム・ロケール情報を必要としない場合は、*locale* を NULL にして渡す必要がある。 |

*buffer*

*datafmt–>count* 変数の配列のアドレス。各変数のサイズは *datafmt–>maxlength*。

**buffer* は、ct_bind がサーバ結果をバインドするプログラム変数です。アプリケーションが結果データをフェッチするために ct_fetch を呼び出すとき、この領域にコピーされます。

*buffer* が NULL の場合、ct_bind はこの結果項目のバインドをクリアします。*buffer* が NULL の場合は、*datafmt*、*copied*、*indicator* も NULL でなければならないことに注意してください。

**注意**　バインドがコマンド構造体上でアクティブである間、*buffer* アドレスは常に有効にしておいてください。

*copied*

*datafmt–>count* 整数の配列のアドレス。フェッチの時点で、ct_fetch はコピーされたデータの長さをこの配列に入れます。*copied* はオプションのパラメータなので、NULL として渡すことができます。

*indicator*

*datafmt–>count* CS_SMALLINT 変数の配列のアドレス。フェッチの時点で、各変数は、フェッチ対象のデータに関する一定の条件を示すために使用されます。*indicator* は、オプションのパラメータなので、NULL として渡すことができます。

次の表に、indicator 変数が持つことのできる値を示します。

| indicator の値 | 意味 |
| --- | --- |
| -1 | フェッチされたデータは NULL である。この場合、データは **buffer* にコピーされない。 |
| 0 | フェッチが成功した。 |
| 整数値 > 0 | フェッチでトランケートされた場合、サーバ・データの実際の長さ。 |

**戻り値**

ct_bind は、次の値を返します。

| 戻り値 | 意味 |
| --- | --- |
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

ct_bind の失敗の主な理由は、次のとおりです。

- *datafmt–>datatype* で指定されているデータ型が無効です。
- *datafmt–>locale* ポインタが不正です。使用されていない場合は、*datafmt–>locale* を NULL に初期化してください。
- 必要とされている変換が使用可能ではありません。

**例**

```
CS_RETCODE        retcode;
 CS_INT           num_cols;
 CS_INT           i;
 CS_INT           j;
 CS_INT           row_count = 0;
 CS_INT           rows_read;
 CS_INT           disp_len;
 CS_DATAFMT       *datafmt;
 EX_COLUMN_DATA   *coldata;
/* Determine the number of columns in this result set */
 .... ct_res_info code deleted ....
/*
```

```
 ** Our program variable, called 'coldata', is an array of
 ** EX_COLUMN_DATA structures. Each array element represents
 ** one column. Each array element will be re-used for each
 ** row.
 **
 ** First, allocate memory for the data element to process.
 */
 coldata = (EX_COLUMN_DATA *)malloc(num_cols *
            sizeof (EX_COLUMN_DATA));

if (coldata == NULL)
 {
    ex_error("ex_fetch_data:malloc() failed");
    return CS_MEM_ERROR;
 }
 datafmt = (CS_DATAFMT *)malloc(num_cols *
           sizeof (CS_DATAFMT));
 if (datafmt == NULL)
 {
    ex_error("ex_fetch_data:malloc() failed");
    free(coldata);
    return CS_MEM_ERROR;
 }

/*
 ** Loop through the columns, getting a description of each
 ** one and binding each one to a program variable.
 **
 ** We're going to bind each column to a character string;
 ** this will show how conversions from server native
 ** datatypes to strings can occur using bind.
 **
 ** We're going to use the same datafmt structure for both
 ** the describe and the subsequent bind.
 **
 ** If an error occurs within the for loop, a break is used
 ** to get out of the loop and the data that was allocated
 ** is freed before returning.
 */

for (i = 0; i < num_cols; i++)
 {
    /*
    ** Get the column description.  ct_describe() fills
    ** the datafmt parameter with a description of the
    ** column.
    */
    retcode = ct_describe(cmd, (i + 1), &datafmt[i]);
```

```
 if (retcode != CS_SUCCEED)
 {
     ex_error("ex_fetch_data:ct_describe() failed");
     break;
 }

/*
 ** Update the datafmt structure to indicate that we
 ** want the results in a null terminated character
 ** string.
 **
** First, update datafmt.maxlength to contain the
 ** maximum possible length of the column. To do this,
 ** call ex_display_len() to determine the number of
 ** bytes needed for the character string
 ** representation, given the datatype described
 ** above. Add one for the null termination character.
 */
datafmt[i].maxlength
       = ex_display_dlen(&datafmt[i]) + 1;

/*
 ** Set datatype and format to tell bind we want things
 ** converted to null terminated strings.
 */
 datafmt[i].datatype = CS_CHAR_TYPE;
 datafmt[i].format = CS_FMT_NULLTERM;


/*
 ** Allocate memory for the column string
 */
 coldata[i].value = (CS_CHAR *)malloc
     (datafmt[i].maxlength);
 if (coldata[i].value == NULL)
 {
     ex_error("ex_fetch_data:malloc() failed");
     retcode = CS_MEM_ERROR;
     break;
 }

/* Now bind.*/
 retcode = ct_bind(cmd, (i + 1), &datafmt[i],
     coldata[i].value, &coldata[i].valuelen,
     &coldata[i].indicator);
```

```
        if (retcode != CS_SUCCEED)
        {
            ex_error("ex_fetch_data:ct_bind() failed");
            break;
        }
    }
```

このコードは、*exutils.c* サンプル・プログラムにある関数 *ex_fetch_data()* ルーチンからの抜粋です。ct_bind を使用する詳細な例については、*compute.c*、*ex_alib.c*、*getsend.c*、*i18n.c* サンプル・プログラムを参照してください。

**使用法**

- ct_bind は、通常の結果カラムやカーソル結果カラム、計算カラム、リターン・パラメータ、またはストアド・プロシージャ・ステータス番号をバインドするために使用されます。通常カラムまたはカーソル・カラムをバインドする場合、そのカラムの複数のローを ct_bind への 1 つの呼び出しでバインドすることができます。

  ---

  **注意**　メッセージ、記述、ロー・フォーマット、および計算フォーマット結果は、バインドされません。これは、CS_MSG_RESULT、CS_DESCRIBE_RESULT、CS_ROWFMT_RESULT、CS_COMPUTEFMT_RESULT タイプの結果セットがフェッチ可能なデータを持たないからです。その代わり、これらの結果セットは一定のタイプの情報を利用できることを示します。アプリケーションは、ct_res_info のような他の Client-Library ルーチンを呼び出すことによって、その情報を取得することができます。「結果」(280 ページ ) を参照してください。

  ---

- バインドは、結果データ項目をプログラム変数に関連付けます。フェッチの時点で、各 ct_fetch 呼び出しは、データ項目のロー・インスタンスをその項目に関連付けられた変数にコピーします。

  結果データ項目が非常に大きい場合 ( サイズの大きな text カラムや image カラムなど )、アプリケーションでデータ項目の値をまとめて取り出すには、バインドされた変数に値全体をコピーするより、ct_get_data を使用したほうが便利です。詳細については、ct_get_data のリファレンス・ページと 「text および image データの処理」(328 ページ ) を参照してください。

- ct_bind は、現在の結果タイプだけをバインドします。ct_results は、その *result_type* パラメータを介して現在の結果タイプを指示します。たとえば、ct_results が **result_type* を CS_STATUS_RESULT に設定した場合、リターン・ステータスをバインドで利用することができます。

- アプリケーションは、現在の結果セットの項目番号を決定するために、ct_res_info を呼び出すことができ、また、各項目の記述を取得するために、ct_describe を呼び出すことができます。
- アプリケーションは、結果項目を 1 つのプログラム変数だけにバインドできます。アプリケーションが結果項目を複数の変数にバインドした場合、最後のバインドだけが有効です。
- アプリケーションは、ct_bind を使用して、変換ルーチンがインストールされている Open Client のユーザ定義データ型をバインドできます。ユーザ定義データ型に対する変換ルーチンをインストールするには、アプリケーションは cs_set_convert を呼び出します。「Open Client のユーザ定義データ型」(351 ページ ) を参照してください。

**既存のバインドの置き換え**

- アプリケーションは、実際にローをフェッチしている間、再バインドできます。つまり、結果項目のバインドを変更する必要がある場合、アプリケーションは、ct_fetch ループ内で ct_bind を呼び出すことができます。
- アプリケーションでは、同じコマンドによって生成されるそれぞれの通常ローおよび計算ロー結果を再バインドする必要はありません。変更がない場合、ct_results が CS_CMD_DONE を返し、論理コマンドの結果が完全に処理されたことを示すまで、特定のタイプの結果のバインドは有効です。

  たとえば、compute および order by 句を持つ select 文を含んでいる言語コマンドでは、複数の通常ロー結果セットと計算ロー結果セットを混在させて生成できます。それらの結果セットは、同じコマンドによって生成されるので、各通常ロー結果セットと各計算ロー結果セットは同一カラムを持ちます。アプリケーションは、( 各タイプの最初の結果セットをフェッチする前に ) それぞれを一度バインドする必要があるだけです。これらのバインドは、両方の結果セットが完全に処理されるまで ( つまり、ct_results が CS_CMD_DONE の result_type を返すまで ) 有効です。

  この動作は、CS_STICKY_BINDS プロパティの値の影響を受けません。

**バインドのクリア**

- 結果項目のバインドをクリアするには、*buffer*、*datafmt*、*copied*、*indicator* を NULL にして、ct_bind を呼び出してください。CS_STICKY_BINDS プロパティがコマンド構造体に対して有効である場合 (CS_TRUE)、結果項目のバインドは後続のコマンドの実行すべてに対してクリアされます。
- すべてのバインドをクリアするには、item を CS_UNUSED にし、*buffer*、*datafmt*、*copied*、*indicator* を NULL にして、ct_bind を呼び出してください。CS_STICKY_BINDS プロパティがコマンド構造体に対して有効である場合 (CS_TRUE)、ct_results が CS_CMD_DONE を返すまで ( つまり、現在のコマンドの実行に対してだけ )、結果項目のバインドがクリアされます。同じコマンドを再び実行すれば、コマンド構造体はそれまでのバインドを復元します。
- 存在しないバインドのクリアはエラーではありません。

**バインドの継続時間**

- デフォルトでは、結果項目とプログラム変数間のバインドは次の動作があるまでアクティブのままです。
  - ct_results が CS_CMD_DONE を返す。
  - アプリケーションが結果項目を再バインドする。
  - アプリケーションがバインドをクリアする。
- CS_STICKY_BINDS コマンド・プロパティを設定することで、アプリケーションのデフォルトのバインド継続時間を変更できます。このプロパティを CS_TRUE に設定すると、同じサーバ・コマンドの実行中、結果項目のバインドはアクティブのままです。特に、結果項目とプログラム変数間のバインドは、次のいずれかの動作があるまでアクティブのままです。
  - アプリケーションが、ct_command、ct_cursor、ct_dynamic、または ct_sendpassthru と同じコマンド構造体で新しいサーバ・コマンドを開始する ( ただし、ネストされた cursor-close、cursor-update、または cursor-delete コマンドはバインドをクリアしない )。
  - アプリケーションが結果項目を再バインドする。
  - アプリケーションがバインドをクリアする。
  - アプリケーションが ct_results を呼び出す。そして、バインドが確立されたときの結果セットと現在の結果セットの間でフォーマットの不一致が検出される。

同じコマンドを繰り返し実行するバッチ処理のアプリケーションでは、CS_STICKY_BINDS プロパティが便利です。

- コマンドは複数の結果セットを返すことができます。CS_STICKY_BINDS プロパティが CS_TRUE である場合、同じコマンドを実行するときに使用するため、最初のコマンドの実行が返した結果セットに対するバインドすべてを維持します。最初のコマンドの実行中、Client-Library は返された結果セットのフォーマットおよびシーケンスについての情報も保存します。ct_results に対する各呼び出しは、同じコマンドを次に実行した後、現在の結果フォーマットを保存されている結果フォーマットと比較します。ct_results が不一致を検出した場合、すべてのバインドをクリアし、情報エラーを発生させて CS_SUCCEED を返します。

  同じコマンドの繰り返し実行からの結果フォーマットが変化するのは、コマンドが条件付きのサーバ側の論理 ( たとえば、if または while 文を含む Adaptive Server Enterprise のストアド・プロシージャ ) を含む場合のみです。

- バインドが現在のコマンドの前回の実行から保存されているかどうか調べるために、アプリケーションは CS_HAVE_BINDS コマンド・プロパティの値を調べることができます。CS_TRUE の値は、現在の結果セットについて 1 つまたは複数のバインドがアクティブであることを示します。たとえば、バッチ処理アプリケーションは次のような論理を使用して結果ローを取得できます。

  ```
  retrieve CS_HAVE_BINDS property with ct_cmd_props
   if CS_HAVE_BINDS is CS_FALSE
       bind variables with ct_bind
   end if
   while ct_fetch returns CS_SUCCEED or CS_ROW_FAIL
     process row data
   end while
  ```

  ct_bind を呼び出しても CS_HAVE_BINDS の値は変化しません。このプロパティは、コマンドの前回の実行のときに確立されたバインドが有効のままかどうかを表します。

- 結果項目のバインドがアクティブのままであるかぎり、ct_bind の *buffer* パラメータとして与えられるメモリ・アドレスは有効にしておいてください。ct_fetch に対する呼び出しはそれぞれ *buffer* アドレスにデータを書き込みます。アドレスが正しくない場合は、アプリケーションはメモリ内容を壊すか、メモリ・アクセス違反を発生させます。たとえば、アプリケーションの C ルーチンが自動変数のアドレスをバインドし、その後アプリケーションが ct_fetch が呼び出す前にこの C ルーチンが制御を返す場合、バインドされたアドレスは無効です。

#### 配列バインド

- 配列バインドは結果カラムをプログラム変数の配列にバインドする処理です。フェッチの時点で、複数のローが 1 回の ct_fetch または ct_scroll_fetch 呼び出しで、変数の配列にコピーされます。アプリケーションは、*datafmt–>count* を 1 よりも大きい値に設定して配列バインドを示します。
- 配列バインドは、通常ローおよびカーソル結果にのみ有効です。これは、他の結果タイプが、1 つのローと同等であるとみなされるからです。
- カラムを配列にバインドする場合、カラムをバインドする一連の ct_bind 呼び出しは、すべて *datafmt–>count* に同じ値を使用しなければなりません。たとえば、3 つのカラムを配列にバインドする場合、最初の 2 つの ct_bind 呼び出しに 5 の *count* を使用し、最後の ct_bind 呼び出しに 3 の *count* を使用することはエラーです。

  ただし、アプリケーションによっては、0 と 1 の *count* を混在させることができます。0 と 1 の *count* は、両方とも ct_fetch が 1 つのローをフェッチするので、同等とみなされます。
- CS_CURSOR_ROWS 値が 1 より大きい場合で、スクロール可能クライアント・カーソルを使用している場合は、配列バインドを使用する必要があります。配列バインドの使用に失敗すると、効率が低下し、予測不可能な動作が発生することがあります。

  非スクロール可能カーソルの場合は、配列または通常のプログラム変数を使用できます。

#### LOB ロケータ・データ型のバインド

LOB ロケータのデータ型を使用するときは、次の内容が適用されます。

- ct_bind() は CS_DATAFMT の *maxlength* 値を無視します。これは、Client-Library がロケータのデータ型の長さを固定と見なすためです。LOB ロケータを使用して送信される、オプションのプリフェッチされたデータに必要なメモリは、その長さ全体に対して内部で割り付けられます。*maxlength* の値がプリフェッチされたデータの長さに影響することはありません。
- 受信 LOB ロケータは CS_CHAR_TYPE にバインドできます。ただし、受信 LOB ロケータ値を CS_TEXT_TYPE、CS_IMAGE_TYPE、または CS_UNITEXT_TYPE データ型にバインドすることはできません。まず、LOB ロケータを LOB ロケータ・データ型にバインドおよびフェッチしてから、明示的に cs_convert を使用して text、image、または unitext データ型に変換して、プリフェッチされたデータを取得する必要があります。

**参照** ct_describe、ct_fetch, ct_res_info、ct_results、データ型のサポート

# ct_br_column

**説明** ブラウズ・モード select によって生成されたカラムについての情報を取得します。

**構文** CS_RETCODE ct_br_column(cmd, colnum, browsedesc)

```
CS_COMMAND       *cmd;
CS_INT           colnum;
CS_BROWSEDESC    *browsedesc;
```

**パラメータ**

*cmd*

クライアント／サーバ・オペレーションを管理する CS_COMMAND 構造体を指すポインタです。

*colnum*

記述するカラムの番号です。select 文の select リストにある最初のカラムのカラム番号が 1、2 番目のカラムのカラム番号が 2、以下同様に続きます。

*browsedesc*

CS_BROWSEDESC 構造体を指すポインタです。ct_br_column は、この構造体を *colnum* によって指定されたカラムについての情報で満たします。

CS_BROWSEDESC 構造体については、「CS_BROWSEDESC 構造体」(84 ページ ) を参照してください。

**戻り値**

ct_br_column は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

現在の結果セットが、select...for browse 言語コマンドによって生成されていなかった場合、ct_br_column は、失敗します。

**使用法**

- ct_br_column は、*browsedesc を *colnum* によって指定されたカラムについての情報で満たします。
- カラムは、次のすべての条件に一致する場合のみブラウズ・モードによって更新できます。
  - ブラウズ可能テーブルに属する。
  - select...for browse の結果である。
  - max(colname) のような SQL 式の結果ではない。
- ブラウズ・モード情報が利用できない場合、ct_br_column を呼び出すことはエラーです。一般的に、現在の結果セットが select...for browse によって生成された CS_ROW_RESULT 結果セットの場合、ブラウズ・モード情報は使用可能です。

  ct_br_column の呼び出しの前に、アプリケーションは *type* を CS_BROWSE_INFO にして ct_res_info を呼び出し、ブラウズ・モード情報が使用可能かどうかチェックできます。

**参照**

「ブラウズ・モード」(22 ページ )、ct_br_table

# ct_br_table

**説明**

ブラウズ・モード・テーブルについての情報を返します。

**構文**

```
CS_RETCODE ct_br_table(cmd, tabnum, type, buffer, buflen, outlen)

CS_COMMAND      *cmd;
CS_INT          tabnum;
CS_INT          type;
CS_VOID         *buffer;
CS_INT          buflen;
CS_INT          *outlen;
```

**パラメータ**

*cmd*

クライアント／サーバ･オペレーションを管理する CS_COMMAND 構造体を指すポインタです。

*tabnum*

対象となるテーブルの番号です。select 文の select リストにある最初のテーブルのテーブル番号が 1、2 番目のテーブル番号が 2 ( 以下同様 ) というようになります。

*type*

返される情報のタイプです。次の表に、*type* の記号値を示します。

| type の値 | ct_br_table が返す内容 | *buffer に設定される内容 |
|---|---|---|
| CS_ISBROWSE | テーブルがブラウズ可能かどうか。テーブルは、ユニーク・インデックスおよびタイムスタンプ・カラムを持っている場合は、ブラウズ可能です。 | CS_TRUE または CS_FALSE。 |
| CS_TABNAME | 番号が *tabnum* のテーブルの名前。 | 文字列値。 |
| CS_TABNUM | ブラウズ・モード select で指定されたテーブルの番号。<br>*type* が CS_TABNUM の場合は、*tabnum* を CS_UNUSED にして渡す必要がある。 | 整数値。 |

*buffer*

ct_br_table が、要求された情報を入れる領域を指すポインタです。

*buflen*

**buffer* データ領域のバイト単位の長さです。

*type* が CS_ISBROWSE または CS_TABNUM の場合は、*buflen* を CS_UNUSED にして渡してください。

*outlen*

整数変数を指すポインタです。

指定された場合、ct_br_table は、要求された情報のバイト単位の長さを **outlen* に設定します。

要求された情報が *buflen* バイトよりも大きい場合、呼び出しは失敗します。アプリケーションは、情報の保持に必要なバイト数を判断するために、**outlen* の値を使用できます。

**戻り値**

ct_br_table は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

ct_br_table は、現在の結果セットが select...for browse 言語コマンドによって生成されていなかった場合、失敗します。

**使用法**

- ct_br_table は、select 文で指定されたテーブルの番号、または特定のテーブルについての情報のどちらかを返します。
- テーブルは、ユニーク・インデックスおよびタイムスタンプ・カラムを持っている場合は、ブラウズ可能です。
- ブラウズ・モード情報が使用可能でない場合、ct_br_table を呼び出すことはエラーです。一般的に、現在の結果セットが select...for browse によって生成された CS_ROW_RESULT 結果セットの場合、ブラウズ・モード情報は使用可能です。
- ct_br_table を呼び出す前に、アプリケーションは、*type* を CS_BROWSE_INFO にして ct_res_info を呼び出し、ブラウズ・モード情報が利用できるかどうかチェックできます。

**参照**

「ブラウズ・モード」(22 ページ )、ct_br_column

# ct_callback

**説明**

Client-Library コールバック・ルーチンをインストールまたは取得します。

**構文**

```
CS_RETCODE ct_callback(context, connection, action, type, func)

CS_CONTEXT        *context;
CS_CONNECTION     *connection;
CS_INT            action;
CS_INT            type;
CS_VOID           *func;
```

パラメータ

*context*

CS_CONTEXT 構造体を指すポインタです。CS_CONTEXT 構造体は Client-Library アプリケーション・コンテキストを定義します。

*context* または *connection* が NULL である必要があります。

- *context* が指定された場合、コールバックは、指定された context の「デフォルト・コールバック」としてインストールされる。一度インストールされると、コンテキスト内で今後割り付けられるすべての接続がデフォルト・コールバックを継承する。
- *context* が NULL の場合、コールバックは、*connection* によって指定された個々の接続でインストールされる。

*connection*

CS_CONNECTION 構造体を指すポインタです。CS_CONNECTION 構造体は、特定のクライアント／サーバ接続の情報を含んでいます。

*context* または *connection* が NULL である必要があります。

- *connection* が指定された場合、コールバックは、指定された接続でインストールされる。
- *connection* が NULL の場合、コールバックは *context* で指定されたアプリケーション・コンテキストにインストールされる。

*action*

次の記号値のいずれかです。

| **action の値** | **意味** |
| --- | --- |
| CS_SET | コールバックをインストールする。 |
| CS_GET | 現在インストールされている、このタイプのコールバックを取得する。 |

*type*

対象となるコールバック・ルーチンのタイプです。次の表に、*type* の記号値を示します。

**表 3-2：ct_callback type パラメータの値**

| **type の値** | **意味** |
|---|---|
| CS_CLIENTMSG_CB | クライアント・メッセージ・コールバック。「クライアント・メッセージ・コールバック」(33 ページ ) を参照。 |
| CS_COMPLETION_CB | 完了コールバック。「完了コールバック」(37 ページ ) を参照。 |
| CS_DS_LOOKUP_CB | ディレクトリ・コールバック。「ディレクトリ・コールバック」(42 ページ ) を参照。 |
| CS_ENCRYPT_CB | 暗号コールバック。「暗号コールバック」(45 ページ ) を参照。 |
| CS_EXTENDED_ENCRYPT_CB | 暗号コールバック。「暗号コールバック」(45 ページ ) を参照。 |
| CS_CHALLENGE_CB | ネゴシエーション・コールバック。「ネゴシエーション・コールバック」(50 ページ ) を参照。 |
| CS_NOTIF_CB | レジスタード・プロシージャ・ノーティフィケーション・コールバック。「ノーティフィケーション・コールバック」(53 ページ ) を参照。 |
| CS_SECSESSION_CB | セキュリティ・セッション・コールバック。「セキュリティ・セッション・コールバック」(55 ページ ) を参照。 |
| CS_SERVERMSG_CB | サーバ・メッセージ・コールバック。「サーバ・メッセージ・コールバック」(59 ページ ) を参照。 |
| CS_SIGNAL_CB + *signal_number* | シグナル・コールバック。「シグナル・コールバック」(63 ページ ) を参照。<br>シグナル・コールバックは、明示定数 CS_SIGNAL_CB に関連するシグナル番号を追加することで識別される。たとえば、SIGALRM シグナルに対するシグナル・コールバックをインストールする場合、*type* を CS_SIGNAL_CB + SIGALRM として渡す。 |
| CS_SSLVALIDATE_CB | SSL 検証コールバック。「SSL 検証コールバック」(65 ページ ) を参照。 |

*func*

ポインタ値。

コールバック・ルーチンをインストールする場合、*func* はインストールするコールバック・ルーチンのアドレスです。

コールバック・ルーチンを取得する場合、ct_callback は、現時点でインストールされているコールバック・ルーチンのアドレスを **func* に設定します。

## 戻り値

ct_callback は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

## 例

```
/*
 ** Install message and completion handlers.
 */
 retstat = ct_callback(Ex_context, NULL, CS_SET,
      CS_CLIENTMSG_CB,(CS_VOID *)ex_clientmsg_cb);
 if (retstat != CS_SUCCEED)
 {
      ex_panic("ct_callback failed");
 }
 retstat = ct_callback(Ex_context, NULL, CS_SET,
      CS_SERVERMSG_CB,(CS_VOID *)ex_servermsg_cb);
 if (retstat != CS_SUCCEED)
 {
      ex_panic("ct_callback failed");
 }

retstat = ct_callback(Ex_context, NULL, CS_SET,
      CS_COMPLETION_CB,(CS_VOID *)CompletionCB);
 if (retstat != CS_SUCCEED)
 {
      ex_panic("ct_callback failed");
 }
```

このコードは、*ex_amain.c* サンプル・プログラムからの抜粋です。ct_callback を使用する他の例については、*ex_alib.c* および *exutils.c* サンプル・プログラムを参照してください。

## 使用法

- 一般のアプリケーションは、コールバック・ルーチンをインストールするためだけに ct_callback を使用します。ただし、一部のアプリケーションでは、以前にインストールされたコールバックを取得する必要がある場合があります。
- コールバック・ルーチンをインストールするために、アプリケーションは、*action* を CS_SET、*func* をインストールするコールバックのアドレスにして、ct_callback を呼び出します。

- 以前にインストールされたコールバックのアドレスを取得するために、アプリケーションは、*action* を CS_GET、*func* をポインタを指すポインタにして、ct_callback を呼び出します。この場合、ct_callback は指定されたタイプの現在のコールバックのアドレスを **func* に設定します。アプリケーションは、このアドレスを後の再利用のために保存することができます。コールバックのアドレスの取得は、そのコールバックのインストールを解除しないことに注意してください。
- ct_callback は、コンテキストまたは特定の接続のどちらかに対しコールバック・ルーチンをインストールするために使用されます。コンテキストのコールバックをインストールするには、*connection* を NULL にして渡してください。接続のコールバックをインストールするには、*context* を NULL にして渡してください。
- コンテキストが割り付けられるとき、コールバック・ルーチンはインストールされません。アプリケーションは、必要なコールバックを明確にインストールしなければなりません。
- 接続を割り付けるとき、デフォルト・コールバック・ルーチンが、その親コンテキストから取得されます。アプリケーションは、ct_callback を呼び出し、これらのデフォルト・コールバックを上書きし、接続レベルで新しいコールバックをインストールできます。
- 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ために、アプリケーションは *func* を NULL にして ct_callback を呼び出すことができます。また、アプリケーションは、新しいコールバック・ルーチンをいつでもインストールすることができます。新しいコールバックは、既存のコールバックを自動的に置き換えます。
- ほとんどのタイプのコールバックで、特定のタイプのコールバックが接続に対してインストールされていない場合、Client-Library は、そのタイプのコールバック情報を廃棄します。

  クライアント・メッセージ・コールバックは、このルールの例外です。インストールされているクライアント・メッセージ・コールバックを持っていない接続に対して、エラーまたは情報メッセージが生成される場合、Client-Library は、そのメッセージを廃棄するのではなく、接続の親コンテキストのクライアント・メッセージ・コールバックを呼び出します。コンテキストがインストールされているクライアント・メッセージ・コールバックを持っていない場合は、そのメッセージを廃棄します。
- 接続は、割り付けられたときに一度だけ、その親コンテキストのコールバック・ルーチンを取得します。これには重要な点が 2 つあります。

- 既存の接続は、親コンテキストのコールバック・ルーチンが変更されてもその影響を受けません。
- ある接続に対して、特定のタイプのコールバック・ルーチンがインストールを解除されている場合、その接続では親コンテキストのコールバック・ルーチンは取得されません。その代わり、接続は、このタイプのコールバック・ルーチンがインストールされていないとみなされます。

- アプリケーションは、コールバック・ルーチンとそれをトリガしたプログラム・コードの間で情報を転送するために、CS_USERDATA プロパティを使用できます。CS_USERDATA プロパティにより、アプリケーションは、ユーザ・データを Client-Library 内部領域に保存して、後でそれを取得できます。

---

**注意**　Digital UNIX の場合、Client-Library は、同期モードを含むすべてのネットワーク I/O モードに対して、割り込み駆動型 I/O を使用します。これは一定のアプリケーションのコーディングに影響します。

Digital UNIX では、専用のシグナル・ハンドラを必要とする Client-Library アプリケーションは、必要なシグナル・ハンドラを、ct_callback を使用してインストールする必要があります。UNIX システム呼び出しを発行するプログラムは、システムの割り込みに起因するシステム呼び出しの失敗をチェックし、中断されたシステム呼び出しがあれば再発行します。

---

**参照**

「コールバック」(25 ページ )、ct_capability、ct_config、ct_con_props、ct_connect

# ct_cancel

**説明**

コマンドまたはコマンドの結果をキャンセルします。

**構文**

```
CS_RETCODE ct_cancel(connection, cmd, type)

CS_CONNECTION    *connection;
CS_COMMAND       *cmd;
CS_INT           type;
```

**パラメータ**

*connection*

CS_CONNECTION 構造体を指すポインタです。CS_CONNECTION 構造体は、特定のクライアント／サーバ接続の情報を含んでいます。

CS_CANCEL_CURRENT キャンセルでは、*connection* は NULL にしてください。

CS_CANCEL_ATTN および CS_CANCEL_ALL キャンセルでは、*connection* と *cmd* のうちの 1 つは、NULL にしてください。*connection* が指定され、*cmd* が NULL の場合、キャンセル・オペレーションは、この接続で未処理のすべてのコマンドに適用されます。

*cmd*

クライアント／サーバ・オペレーションを管理する CS_COMMAND 構造体を指すポインタです。

CS_CANCEL_CURRENT キャンセルでは、*cmd* を指定してください。キャンセル・オペレーションは、このコマンド構造体で未処理の結果だけに適用されます。

CS_CANCEL_ATTN および CS_CANCEL_ALL キャンセルでは、*cmd* が指定され *connection* が NULL である場合、キャンセル・オペレーションはこのコマンド構造体で未処理のコマンドだけに適用されます。*cmd* が NULL で、*connection* が指定された場合、キャンセル・オペレーションは、この接続で未処理のすべてのコマンドに適用されます。

*type*

キャンセルのタイプ。次の表に、*type* の有効な記号値を示します。

**表 3-3：ct_cancel type パラメータの値**

| type の値 | 結果 | 注意 |
|---|---|---|
| CS_CANCEL_ALL | ct_cancel は、現在のコマンドのキャンセルを指示するアテンションをサーバに送信する。<br>Client-Library は、そのコマンドによって生成されたすべての結果をただちに廃棄する。 | この接続のカーソルは定義されていない状態になる。<br>カーソルの状態を決定するために、アプリケーションは *property* を CS_CUR_STATUS にして ct_cmd_props を呼び出すことができる。 |
| CS_CANCEL_ATTN | ct_cancel は、現在のコマンドのキャンセルを指示するアテンションをサーバに送信する。<br>アプリケーションがサーバから次に読み込みを行ったとき、Client-Library は、キャンセルされたコマンドによって生成されたすべての結果を廃棄する。 | この接続のカーソルは定義されていない状態になる。<br>カーソルの状態を決定するために、アプリケーションは *property* を CS_CUR_STATUS にして ct_cmd_props を呼び出すことができる。 |
| CS_CANCEL_CURRENT | ct_cancel は、現在の結果セットを廃棄する。 | オープン・カーソルを持つ接続での使用が安全である。 |

## 戻り値

ct_cancel は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_PENDING | 非同期ネットワーク I/O が有効。「非同期プログラミング」(12 ページ) を参照。 |
| CS_CANCELED | キャンセル・オペレーションは取り消された。CS_CANCEL_CURRENT タイプのキャンセルだけを取り消すことができる。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ) を参照。 |
| CS_TRYING | この接続では、キャンセル・オペレーションは未処理である。 |

## 例

```
if (query_code == CS_FAIL)
 {
        /*
        ** Terminate results processing and break out of
        ** the results loop.
        */
        retcode = ct_cancel(NULL, cmd, CS_CANCEL_ALL);
        if (retcode != CS_SUCCEED)
        {
            ex_error("ex_execute_cmd:ct_cancel() failed");
        }
        break;
 }
```

このコードは、*exutils.c* サンプル・プログラムからの抜粋です。

## 使用法

- コマンドのキャンセルは、現在のコマンドの実行の停止を命令するアテンションをサーバに送信することと同等です。コマンドをキャンセルすると、そのコマンドによって生成された結果は、アプリケーションでは利用できません。
- 結果のキャンセルは、結果セットをフェッチしてから廃棄する処理と同等です。一度キャンセルした結果を、その後アプリケーションで利用することはできません。ただし、結果セットが完全に処理されていない場合、以降の結果は使用可能です。

**コマンドのキャンセル**

- アプリケーションで現在のコマンドとそれによって生成されたすべての結果をキャンセルするには、*type* に CS_CANCEL_ATTN または CS_CANCEL_ALL を指定して ct_cancel を呼び出します。これらの呼び出しは、両方とも Client-Library に次のことを指示します。

  - 現在のコマンドの実行の停止を指示するアテンションをサーバに送信します。

  - コマンドによってすでに生成された結果を廃棄します。

- 両方のタイプのキャンセルは、処理中のコマンドがない場合、アテンションをサーバに送信することなく、CS_SUCCEED をただちに返します。

- 開始されたコマンドまたはコマンド・バッチを送信する ct_send がアプリケーションでまだ呼び出されていない場合は、次のようになります。

  - CS_CANCEL_ALL キャンセルは、アテンションをサーバに送信しないで、その開始されたコマンドまたはコマンド・バッチを廃棄します。CS_CANCEL_ATTN キャンセルは無効です。

- 接続はエラーのために使用できなくなることがあります。この状態が発生した場合、Client-Library は、その接続を「dead」とマーク付けします。アプリケーションは、接続が「dead」とマーク付けされていないかどうかを判断するために、CS_CON_STATUS プロパティを使用できます。

  接続が結果処理エラーのために「dead」とマーク付けされている場合、アプリケーションは、ct_cancel (CS_CANCEL_ALL または CS_CANCEL_ATTN) を呼び出して接続のリカバリを試みることができます。リカバリに失敗した場合、アプリケーションは、接続のクローズおよびその CS_CONNECTION 構造体の解除をしなければなりません。

- CS_CANCEL_ALL と CS_CANCEL_ATTN の違いは、次のとおりです。

  - CS_CANCEL_ALL では、Client-Library は、キャンセルされたコマンドの結果をただちに廃棄します。

  - CS_CANCEL_ATTN の場合、Client-Library は、アプリケーションがサーバから読み込もうとするまで待ってから、結果を廃棄します。

- Client-Library が結果を廃棄するために結果ストリームから読み込まれなければならず、また、結果ストリームからの読み込みは常に安全とはかぎらないので、この違いは重要です。

  コールバックまたは割り込みハンドラの中から、または非同期ルーチンが未処理の場合、結果ストリームからの読み込みは安全ではありません。非同期オペレーションが未処理のときを除いて、アプリケーションがそのメインライン・コードで動作しているときはいつでも、結果ストリームからの読み込みが安全です。

  コールバックまたは割り込みハンドラの中から、または非同期オペレーションが未処理の場合に、CS_CANCEL_ATTN を使用してください。

  非同期オペレーションが未処理のときを除いて、メインライン・コードで CS_CANCEL_ALL を使用してください。

- CS_CANCEL_ALL では、コマンド構造体が「クリーン」状態のままなので、引き続き別のオペレーションで使用できます。ただし、コマンドが CS_CANCEL_ATTN でキャンセルされると、そのコマンド構造体は、Client-Library ルーチンが CS_CANCELED を返すまで再び利用することはできません。

  CS_CANCELED を返すことができる Client-Library ルーチンは、次のとおりです。

  - ct_cancel(CS_CANCEL_CURRENT)
  - ct_fetch
  - ct_get_data
  - ct_options
  - ct_recvpassthru
  - ct_results
  - ct_send
  - ct_sendpassthru

- CS_CANCEL_ATTN には、主な用途が 2 つあります。

  - アプリケーションの割り込みハンドラまたはコールバック・ルーチンの中からコマンドをキャンセルする。
  - 非同期アプリケーションで、結果処理ルーチン ct_results と ct_fetch に対する未処理の呼び出しをキャンセルする。

- コマンドが送信されて、ct_results が呼び出されなかった場合、ct_cancel(CS_CANCEL_ATTN) の呼び出しは無効です。
- オープン・カーソルを持つ接続上のコマンドのキャンセルは、そのカーソルの状態に予期しないような影響を与えることがあります。このため、CS_CANCEL_ALL と CS_CANCEL_ATTN タイプのキャンセルは、オープン・カーソルを持つ接続に使用しないことをおすすめします。カーソル・コマンドをキャンセルする代わりに、アプリケーションは、カーソルを単にクローズすることができます。

**現在の結果のキャンセル**

- 現在の結果をキャンセルするために、アプリケーションは、*type* を CS_CANCEL_CURRENT にして ct_cancel を呼び出します。これは、Client-Library に現在の結果の廃棄を指示します。CS_END_DATA を返すまでの ct_fetch の呼び出しと同等です。
- バッファ内に次の結果がある場合には、アプリケーションで利用でき、現在コマンドは影響を受けません。
- 結果セットのキャンセルは、結果項目とプログラム変数の間のバインドをクリアします。
- CS_CANCEL_CURRENT タイプのキャンセルは、たとえフェッチ可能な結果が含まれていなくても、すべてのタイプの結果セットに有効です。結果セットにフェッチ可能な結果が含まれていない場合、キャンセルは、無効です。

**参照**　ct_fetch、ct_results

# ct_capability

**説明**　クライアント／サーバ機能を設定または取得します。

**構文**

```
CS_RETCODE ct_capability(connection, action, type, capability, value)

CS_CONNECTION    *connection;
CS_INT           action;
CS_INT           type;
CS_INT           capability;
CS_VOID          *value;
```

パラメータ

*connection*

CS_CONNECTION 構造体を指すポインタです。CS_CONNECTION 構造体は、特定のクライアント／サーバ接続の情報を含んでいます。

*action*

次の記号値のいずれかです。

| **action の値** | **意味** |
|---|---|
| CS_SET | 機能を設定する。 |
| CS_GET | 機能を取得する。 |

*type*

機能のタイプ・カテゴリーです。次の表に、*type* の記号値を示します。

**表 3-4：ct_capability type パラメータの値**

| **type の値** | **意味** |
|---|---|
| CS_CAP_REQUEST | 要求機能。<br>これらの機能は、接続がサポートできる要求のタイプを記述する。<br>要求機能は、取得のみ可能。 |
| CS_CAP_RESPONSE | 応答機能。<br>これらの機能は、サーバが接続に送信できる応答のタイプを記述する。<br>アプリケーションは、接続がオープンされる前に応答機能を設定することができ、また、いつでも応答機能を取得することができる。 |

*capability*

対象となる機能です。次の 2 つの表に、*capability* の有効な記号値を示します。

**注意**　表に示した値に加えて、アプリケーションが、すべての応答または要求機能を同時に設定または取得することを示すために、*capability* は、特別な値 CS_ALL_CAPS を持つことができます。CS_ALL_CAPS は、主に、ゲートウェイ・アプリケーションで使用されます。一般の Client-Library アプリケーションでは、少数の機能だけを設定または取得する必要があります。

表 3-5 は、CS_CAP_REQUEST の機能の一覧です。

表 3-5：要求機能

| CS_CAP_REQUEST 機能 | 意味 | 機能対象 |
|---|---|---|
| CS_CON_INBAND | 範囲内 ( 非優先 ) アテンション。 | 接続 |
| CS_CON_LOGICAL | 論理マッピング。 | 接続 |
| CS_CON_OOB | 範囲外 ( 優先 ) アテンション。 | 接続 |
| CS_CSR_ABS | 指定された絶対カーソル・ローのフェッチ。 | カーソル |
| CS_CSR_FIRST | 最初のカーソル・ローのフェッチ。 | カーソル |
| CS_CSR_LAST | 最後のカーソル・ローのフェッチ。 | カーソル |
| CS_CSR_MULTI | マルチ・ロー・カーソルのフェッチ。 | カーソル |
| CS_CSR_PREV | 以前のカーソル・ローのフェッチ。 | カーソル |
| CS_CSR_REL | 指定された相対カーソル・ローのフェッチ。 | カーソル |
| CS_CUR_IMPLICIT | TDS の最適化された読み込み専用カーソル。 | カーソル |
| CS_DATA_BIGDATETIME | Bigdatetime データ型。 | データ型 |
| CS_DATA_BIGTIME | Bigtime データ型。 | データ型 |
| CS_DATA_BIN | binary データ型。 | データ型 |
| CS_DATA_VBIN | 可変長 binary 型。 | データ型 |
| CS_DATA_LBIN | long binary データ型。 | データ型 |
| CS_DATA_BIT | bit データ型。 | データ型 |
| CS_DATA_BITN | null が許可されるビット値。 | データ型 |
| CS_DATA_BOUNDARY | boundary データ型。 | データ型 |
| CS_DATA_CHAR | character データ型。 | データ型 |
| CS_DATA_VCHAR | 可変長 character データ型。 | データ型 |
| CS_DATA_LCHAR | long character データ型。 | データ型 |
| CS_DATA_DATE | date データ型。 | データ型 |
| CS_DATA_DATE4 | short datetime データ型。 | データ型 |
| CS_DATA_DATE8 | datetime データ型。 | データ型 |
| CS_DATA_DATETIMEN | NULL datetime 値。 | データ型 |
| CS_DATA_DEC | decimal データ型。 | データ型 |
| CS_DATA_FLT4 | 4 バイト float データ型。 | データ型 |
| CS_DATA_FLT8 | 8 バイト float データ型。 | データ型 |
| CS_DATA_FLTN | null が許可される float 値。 | データ型 |
| CS_DATA_IMAGE | image データ型。 | データ型 |

| CS_CAP_REQUEST 機能 | 意味 | 機能対象 |
|---|---|---|
| CS_DATA_INT1 | tiny integer データ型。 | データ型 |
| CS_DATA_INT2 | small integer データ型。 | データ型 |
| CS_DATA_INT4 | integer データ型。 | データ型 |
| CS_DATA_INTN | NULL integer。 | データ型 |
| CS_DATA_INT8 | 8 バイト integer データ型。 | データ型 |
| CS_DATA_LBIN | long binary データ型。 | データ型 |
| CS_DATA_LCHAR | long character データ型。 | データ型 |
| CS_DATA_UINT2 | 符号なしの 2 バイト integer データ型。 | データ型 |
| CS_DATA_UINT4 | 符号なしの 4 バイト integer データ型。 | データ型 |
| CS_DATA_UINT8 | 符号なしの 8 バイト integer データ型。 | データ型 |
| CS_DATA_UINTN | 符号なしのデータ型。 | データ型 |
| CS_DATA_UCHAR | 符号なしの文字。 | データ型 |
| CS_DATA_UNITEXT | 符号なしの文字。 | データ型 |
| CS_DATA_MNY4 | short money データ型。 | データ型 |
| CS_DATA_MNY8 | money データ型。 | データ型 |
| CS_DATA_MONEYN | NULL money 値。 | データ型 |
| CS_DATA_NUM | numeric データ型。 | データ型 |
| CS_DATA_SENSITIVITY | Secure Server sensitivity データ型。 | データ型 |
| CS_DATA_TEXT | text データ型。 | データ型 |
| CS_DATA_TIME | time データ型。 | データ型 |
| CS_DATA_XML | 可変幅 character データ型。 | データ型 |
| CS_DOL_BULK | DOL テーブルでのバルク・コピー用トークン。 | バルク・コピー |
| CS_OBJECT_CHAR | サーバでストリーミング文字データの送受信ができるかどうかを指定する。 | Java オブジェクト |
| CS_OBJECT_BINARY | サーバでストリーミング・バイナリ・データの送受信ができるかどうかを指定する。 | ストリーミング・データ |
| CS_OBJECT_JAVA1 | 直列化された Java オブジェクトをサーバで送受信できるかどうかを指定する。 | ストリーミング・データ |
| CS_OPTION_GET | クライアントが、サーバから現在のオプション値を得ることができるかどうか。 | オプション |

| CS_CAP_REQUEST 機能 | 意味 | 機能対象 |
|---|---|---|
| CS_PROTO_BULK | トークン化されたバルク・コピー。 | バルク・コピー |
| CS_PROTO_DYNAMIC | 準備文の記述を準備時点に戻す。 | 動的 SQL |
| CS_PROTO_DYNPROC | Client-Library が、「create proc」を動的 SQL 準備文に追加する。 | 動的 SQL |
| CS_REQ_BCP | バルク・コピー要求。 | コマンド |
| CS_REQ_CURSOR | カーソル要求。 | コマンド |
| CS_REQ_DBRPC2 | 長い RPC 名の要求。 | コマンド |
| CS_REQ_DYN | 動的 SQL 要求。 | コマンド |
| CS_REQ_LANG | 言語要求。 | コマンド |
| CS_REQ_MSG | メッセージ・コマンド。 | コマンド |
| CS_REQ_MSTMT | 1 つの Client-Library 言語コマンドごとに複数のサーバ・コマンド。 | コマンド |
| CS_REQ_NOTIF | レジスタード・プロシージャ・ノーティフィケーション。 | コマンド |
| CS_REQ_PARAM | PARAM/PARAMFMT TDS ストリームを要求に使用。 | コマンド |
| CS_REQ_RESERVED1 | 今後のために予約済み。 | コマンド |
| CS_REQ_RESERVED2 | 今後のために予約済み。 | コマンド |
| CS_REQ_URGNOTIF | TDS パケット・ヘッダに「緊急」ビットを設定したノーティフィケーションの送信。 | レジスタード・プロシージャ |
| CS_REQ_RPC | リモート・プロシージャ要求。 | コマンド |
| CS_WIDETABLES | ワイド・テーブルのサポート。 | 接続 |

表 3-6 は、CS_CAP_RESPONSE の機能の一覧です。

**表 3-6：応答機能**

| CS_CAP_RESPONSE 機能 | 意味 | 機能<br>対象 |
|---|---|---|
| CS_CON_NOINBAND | 範囲内 ( 非優先 ) アテンションなし。 | 接続 |
| CS_CON_NOOOB | 範囲外 ( 優先 ) アテンションなし。 | 接続 |
| CS_DATA_NOBIGDATETIME | bigdatetime データ型なし。 | データ型 |
| CS_DATA_NOBIGTIME | bigtime データ型なし。 | データ型 |
| CS_DATA_NOBIN | binary データ型なし。 | データ型 |
| CS_DATA_NOBOUNDARY | セキュリティ boundary データ型なし。 | データ型 |
| CS_DATA_NOVBIN | 可変長 binary 型なし。 | データ型 |
| CS_DATA_NOLBIN | long binary データ型なし。 | データ型 |
| CS_DATA_NOBIT | bit データ型なし。 | データ型 |
| CS_DATA_NOCHAR | character データ型なし。 | データ型 |
| CS_DATA_NOVCHAR | 可変長 character データ型なし。 | データ型 |
| CS_DATA_NOLCHAR | long character データ型なし。 | データ型 |
| CS_DATA_NODATE | date データ型なし。 | データ型 |
| CS_DATA_NODATE4 | short datetime データ型なし。 | データ型 |
| CS_DATA_NODATE8 | datetime データ型なし。 | データ型 |
| CS_DATA_NODATETIMEN | NULL datetime 値なし。 | データ型 |
| CS_DATA_NODEC | decimal データ型なし。 | データ型 |
| CS_DATA_NOFLT4 | 4 バイト float データ型なし。 | データ型 |
| CS_DATA_NOFLT8 | 8 バイト float データ型なし。 | データ型 |
| CS_DATA_NOIMAGE | image データ型なし。 | データ型 |
| CS_DATA_NOINT1 | tiny integer データ型なし。 | データ型 |
| CS_DATA_NOINT2 | small integer データ型なし。 | データ型 |
| CS_DATA_NOINT4 | integer データ型なし。 | データ型 |
| CS_DATA_NOINT8 | 8 バイトの integer データ型なし。 | データ型 |
| CS_DATA_NOINTN | NULL integer なし。 | データ型 |
| CS_DATA_NOLBIN | long binary データ型なし。 | データ型 |
| CS_DATA_NOLCHAR | long character データ型なし。 | データ型 |
| CS_DATA_NOMNY4 | short money データ型なし。 | データ型 |
| CS_DATA_NOMNY8 | money データ型なし。 | データ型 |
| CS_DATA_NOMONEYN | NULL money 値なし。 | データ型 |
| CS_DATA_NONUM | numeric データ型なし。 | データ型 |
| CS_DATA_NOSENSITIVITY | Secure Server sensitivity データ型なし。 | データ型 |

| **CS_CAP_RESPONSE 機能** | **意味** | **機能対象** |
| --- | --- | --- |
| CS_DATA_NOTEXT | text データ型なし。 | データ型 |
| CS_DATA_NOTIME | time データ型なし。 | データ型 |
| CS_DATA_NOUCHAR | 符号なしの character データ型なし。 | データ型 |
| CS_DATA_NOUINT2 | 符号なしの 2 バイトの integer データ型なし。 | データ型 |
| CS_DATA_NOUINT4 | 符号なしの 4 バイトの integer データ型なし。 | データ型 |
| CS_DATA_NOUINT8 | 符号なしの 8 バイトの integer データ型なし。 | データ型 |
| CS_DATA_NOUINTN | 符号なしの integer データ型なし。 | データ型 |
| CS_DATA_NOUNITEXT | 符号なしの short データ型なし。 | データ型 |
| CS_DATA_NOXML | 可変幅 character データ型なし。 | データ型 |
| CS_DATA_NOZEROLEN | 長さゼロのデータ型なし。 | データ型 |
| CS_NOWIDETABLES | ワイド・テーブルなし。 | 接続 |
| CS_RES_NOEED | 拡張エラー結果なし。 | 結果 |
| CS_RES_NOMSG | メッセージ結果なし。 | 結果 |
| CS_RES_NOPARAM | PARAM/PARAMFMT TDS ストリームを RPC 結果に使用しない。 | 結果 |
| CS_RES_NOSTRIPBLANKS | null 可能な固定長文字カラムからデータを返すとき、サーバはブランクを削除しない。 | 結果 |
| CS_RES_NOTDSDEBUG | 特定の dbcc コマンドへの応答、TDS デバッグ・トークンなし。 | 結果 |
| CS_RES_NOXNLMETADATA | テーブルのメタデータなし。 | 結果 |
| CS_RES_RESERVED | 今後のために予約済み。 | 今後使用 |
| CS_RES_SUPPRESS_FMT | サーバはロー・フォーマット・キャッシュをサポート可能。 | 結果 |

*value*

機能を設定する場合、*value* は、値 CS_TRUE または CS_FALSE を持つ CS_BOOL 変数を指します。

機能を取得する場合、value は、ct_capability が CS_TRUE または CS_FALSE に設定する CS_BOOL 長の変数を指します。

CS_TRUE は、機能が使用可能であることを示します。たとえば、CS_RES_NOEED 機能を CS_TRUE に設定した場合、拡張エラー・データはその接続で返されません。

---

**注意**　機能が CS_ALL_CAPS の場合、値は CS_CAP_TYPE 構造体を指さなければなりません。

---

**戻り値**

ct_capability は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ) を参照。 |

**使用法**

- 機能は、接続がサポートするクライアント／サーバ機能を記述します。
- 機能には、次の 2 つのタイプがあります。「**応答機能**」とも呼ばれる CS_CAP_RESPONSE 機能と、「**要求機能**」とも呼ばれる CS_CAP_REQUEST 機能です。
    - アプリケーションは、サーバ接続がサポートする要求の種類を判断するために、要求機能を使用します。たとえば、アプリケーションは、接続がカーソル要求をサポートするかどうかを調べるために、CS_REQ_CURSOR 機能を取得できます。
    - アプリケーションは、アプリケーションで処理できないタイプの応答をサーバが送信することを防ぐために、応答機能を使用します。たとえば、アプリケーションは、CS_DATA_NOMONEYN 応答機能を CS_TRUE に設定して、サーバによる NULL money 値の送信を防ぐことができます。
- 接続をオープンする前に、アプリケーションは次のことができます。

    - 要求機能または応答機能を取得して、アプリケーションの現在の TDS (Tabular Data Stream) バージョン・レベルで通常サポートされている要求機能および応答機能を調べます。アプリケーションの TDS レベルは、デフォルトで、そのアプリケーションが ct_init の呼び出しで要求した CS_VERSION レベルに基づく値に設定されます。
    - 応答機能を設定することによって、接続で特定のタイプのサーバ応答の受信を望まないことを指定できます。アプリケーションでは、取得のみ可能な要求機能は設定できません。
- 接続をオープンすると、アプリケーションでは次のことが可能になります。
    - 要求機能を取得して、接続がサポートするタイプの要求を知ることができます。
    - 応答機能を取得して、以前に指定した応答タイプを接続ではサーバが使用しないことに同意したかどうかを知ることができます。
- 機能は、接続の TDS バージョン・レベルによって決まります。すべての TDS バージョンが同じ機能をサポートするわけではありません。たとえば、4.0 TDS は、レジスタード・プロシージャ・ノーティフィケーションまたはカーソル要求をサポートしません。ただし、4.0 TDS は、バルク・コピー要求、リモート・プロシージャ・コール要求、ロー結果、および計算ロー結果をサポートします。接続の TDS バージョン・レベルは、その接続処理中にネゴシエートされます。
- アプリケーションが、CS_TDS_VERSION プロパティを設定する場合、Client-Library は、既存の機能値を新しい TDS バージョンに対応するデフォルト機能値で上書きします。このため、アプリケーションは CS_TDS_VERSION を設定してから、接続の機能を設定する必要があります。

  CS_TDS_VERSION は、ネゴシエートされたログイン・プロパティであるため、サーバはその値を接続の時点で変更できます。これが行われた場合、Client-Library は、既存の機能値を新しい TDS バージョンに対応するデフォルト機能値で上書きします。
- 機能値は接続の時点で変更できるので、アプリケーションは、その接続で有効な機能値を判断するために、接続がオープンされた後に ct_capability を呼び出す必要があります。
- 接続がクローズされると、Client-Library は機能値をそのアプリケーションのデフォルトの TDS バージョンに対応する値にリセットします。

**複数機能の設定および取得**

- ゲートウェイ・アプリケーションでは、1 つのタイプ・カテゴリーのすべての機能を、ct_capability の 1 回の呼び出しで設定または取得することが必要な場合がよくあります。このことを行うために、アプリケーションは、次のようにして ct_capability を呼び出します。
  - 対象のタイプ・カテゴリーを *type* として設定
  - CS_ALL_CAPS を *capability* に設定
  - CS_CAP_TYPE 構造体のポインタを *value* として設定
- Client-Library は、次のマクロを提供し、アプリケーションが CS_CAP_TYPE 構造体のビットを設定、解除、およびテストできるようにします。
  - CS_SET_CAPMASK(*mask*, *capability*)
  - CS_CLR_CAPMASK(*mask*, *capability*)
  - CS_TST_CAPMASK(*mask*, *capability*)

  ここで、*mask* は CS_CAP_TYPE 構造体へのポインタで、*capability* は対象となる機能です。

**機能の外部設定**

- ct_connect は、必要に応じて Open Client/Server ランタイム設定ファイルのセクションを読み込んで、接続のサーバ機能を設定します。

  この機能については、「ランタイム設定ファイルの使い方」(352 ページ ) を参照してください。

**参照**　「機能」(67 ページ )、ct_con_props、ct_connect、ct_options、「プロパティ」(208 ページ )

# ct_close

**説明**　サーバ接続をクローズします。

**構文**

```
CS_RETCODE ct_close(connection, option)

CS_CONNECTION    *connection;
CS_INT           option;
```

**パラメータ**

*connection*

CS_CONNECTION 構造体を指すポインタです。CS_CONNECTION 構造体は、特定のクライアント／サーバ接続の情報を含んでいます。

*option*

クローズに使用するオプションです。次の表に、*option* の記号値を示します。

| **option の値** | **意味** |
|---|---|
| CS_UNUSED<br>(10.0+ サーバ専用 ) | デフォルトの動作。<br>ct_close は、接続をクローズする前に、ログアウト・メッセージをサーバに送信して、このメッセージに対する応答を読み込む。<br>接続が未処理の結果を持つ場合、ct_close は CS_FAIL を返す。 |
| CS_FORCE_CLOSE | 接続は、結果が未処理かどうかにかかわらず、サーバに通知することなしにクローズされる。<br>このオプションは、アプリケーションがサーバ応答を待たされているとき、主に使用される。また、ct_results、ct_fetch、または ct_cancel が CS_FAIL を返した場合にも役立つ。 |

**戻り値**

ct_close は、次の値を返します。

| **戻り値** | **意味** |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_PENDING | 非同期ネットワーク I/O が有効。「非同期プログラミング」(12 ページ )を参照。<br>非同期ネットワーク I/O が有効であり、option が CS_FORCE_CLOSE で ct_close が呼び出された場合、ネットワーク応答を示す CS_SUCCEED または CS_FAIL をただちに返す。この場合、完了コールバック・イベントは発生しない。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ )を参照。<br>ct_close は、*option* が CS_FORCE_CLOSE で呼び出されたとき、CS_BUSY を返さないので注意が必要。 |

ct_close(CS_UNUSED) が失敗する最も一般的な原因は、接続上の未処理の結果です。

**例**

```
CS_RETCODE    retcode;
CS_INT        close_option;
close_option = (status != CS_SUCCEED)?CS_FORCE_CLOSE :
                CS_UNUSED;

retcode = ct_close(connection, close_option);
```

```
 if (retcode != CS_SUCCEED)
{
    ex_error("ex_con_cleanup: ct_close() failed");
    return retcode;
}
```

このコードは、*exutils.c* サンプル・プログラムからの抜粋です。

**使用法**

- CS_CONNECTION の割り付けを解除するために、アプリケーションは、その接続のクローズが正常に終了した後、ct_con_drop を呼び出すことができます。
- 接続はエラーのために使用できなくなることがあります。この状態が発生した場合、Client-Library は、その接続を「dead」とマーク付けします。アプリケーションは、接続が「dead」とマーク付けされていないかどうかを判断するために、CS_CON_STATUS プロパティを使用できます。

  接続が「dead」とマーク付けされている場合、アプリケーションは、その接続をクローズするために ct_close(CS_FORCE_CLOSE) を、また、その CS_CONNECTION 構造体を削除するためには、ct_con_drop を呼び出さなければなりません。

  例外は、あるタイプの結果処理エラーに対して発生します。結果処理中に接続が「dead」とマーク付けされた場合、アプリケーションから ct_cancel(CS_CANCEL_ALL または CS_CANCEL_ATTN) を呼び出して接続のリカバリを試みることができます。リカバリに失敗した場合、アプリケーションは、接続のクローズおよびその CS_CONNECTION 構造体の解除をしなければなりません。
- 接続をクローズすると、その接続上のすべてのオープン・カーソルは、自動的にクローズされます。
- 接続が、非同期ネットワーク I/O を使用している場合、ct_close は CS_PENDING を返します。サーバ応答が届いたとき、Client-Library は、その接続をクローズして、その接続にインストールされた完了コールバックを呼び出します。
- ct_close の動作は、クローズのタイプを決定する *option* の値に依存します。次の各セクションには、特定のクローズのタイプについての情報が含まれています。

**デフォルトのクローズ動作**

- 接続が未処理の結果を持つ場合、ct_close は CS_FAIL を返します。接続がオープン・カーソルを持つ場合、Client-Library がその接続をクローズするときにサーバはそのカーソルをクローズします。

- 10.0+ サーバと接続した場合、接続を終了する前に、ct_close はログアウト・メッセージをサーバに送信し、このメッセージに対する応答を読み込みます。このメッセージの内容は、ct_close の動作には影響しません。
- 非同期オペレーションが未処理のとき、アプリケーションは ct_close(CS_UNUSED) を呼び出せません。

CS_FORCE_CLOSE の動作

- オープン・カーソルまたは未処理の結果を持つかどうかにかかわらず、接続をクローズします。
- ct_close は、CS_FORCE_CLOSE オプションで呼び出されたとき、非同期に動作しません。ct_close(CS_FORCE_CLOSE) を呼び出して非同期接続をクローズすると、ネットワーク応答を示す、CS_SUCCEED または CS_FAIL がただちに返されます。この場合、完了コールバック・イベントは発生しない。
- CS_FORCE_CLOSE は、次のときに役立ちます。
  - 接続が「dead」とマーク付けされている場合。
  - アプリケーションがサーバ応答を待たされている場合。
  - 結果が未処理のため、アプリケーションが ct_close(CS_UNUSED) を呼び出せない場合。
- ログアウト・メッセージがサーバに送信されないので、サーバはそのクローズが意図的かどうか、または接続が失われた結果またはクライアントがクラッシュした結果によるものかどうかを区別することができません。
- 非同期オペレーションが未処理のとき、アプリケーションは、ct_close(CS_FORCE_CLOSE) を呼び出すことができます。

参照　ct_callback、ct_con_drop、ct_connect、ct_con_props

# ct_cmd_alloc

説明　CS_COMMAND 構造体を割り付ける。

構文　CS_RETCODE ct_cmd_alloc(connection, cmd_pointer)

```
CS_CONNECTION    *connection;
CS_COMMAND       **cmd_pointer;
```

**パラメータ**

*connection*

CS_CONNECTION 構造体を指すポインタです。CS_CONNECTION 構造体は、特定のクライアント／サーバ接続の情報を含んでいます。

*cmd_pointer*

ポインタ変数のアドレスです。cs_cmd_alloc は、**cmd_pointer* に、新しく割り付けた CS_COMMAND 構造体のアドレスを設定します。

**戻り値**

ct_cmd_alloc は、次の値を返します。

| 戻り値 | 意味 |
| --- | --- |
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

ct_cmd_alloc が失敗する最も一般的な原因は、メモリの不足です。

**例**

```
/* Allocate a command handle to send the text with */
 if ((retcode = ct_cmd_alloc(connection, &cmd)) !=
      CS_SUCCEED)
 {
      ex_error("UpdateTextData:ct_cmd_alloc() failed");
      return retcode;
 }
```

このコードは、*getsend.c* サンプル・プログラムからの抜粋です。

**使用法**

- CS_COMMAND 構造体はコマンド構造体とも呼ばれ、Client-Library アプリケーションがサーバにコマンドを送信するためと、それらのコマンドの結果を処理するために使用する制御構造体です。
- アプリケーションは、ct_con_alloc を呼び出して接続にコマンド構造体を割り付ける前に、ct_cmd_alloc を呼び出して接続構造体を割り付けてください。

  ただし、その接続構造体がオープン接続を表している必要はありません ( アプリケーションは、ct_connect を呼び出してサーバに接続することによって、接続をオープンします )。

**参照**

ct_command、ct_cmd_drop、ct_cmd_props、ct_con_alloc、ct_cursor、ct_dynamic

# ct_cmd_drop

**説明**

CS_COMMAND 構造体の割り付けを解除する。

**構文**

CS_RETCODE ct_cmd_drop(cmd)

CS_COMMAND　*cmd;

**パラメータ**

*cmd*

CS_COMMAND 構造体を指すポインタです。

**戻り値**

ct_cmd_drop は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

次のような場合、ct_cmd_drop が CS_FAIL を返します。

- **cmd* がアクティブ・コマンドを持っている場合。初期化されているがまだ送信されていないコマンドは、アクティブとみなされます。
- **cmd* がオープン・カーソルを持っている場合。
- **cmd* が未処理の結果を持っている場合。

**例**

```
if ((retcode = ct_cmd_drop(cmd)) != CS_SUCCEED)
 {
        ex_error("DoCompute: ct_cmd_drop() failed");
        return retcode;
 }
```

このコードは、*compute.c* サンプル・プログラムからの抜粋です。

**使用法**

- CS_COMMAND 構造体は、Client-Library アプリケーションがコマンドをサーバに送信するためと、このコマンドの結果を処理するために使用する制御構造体です。
- 一度、コマンド構造体の割り付けを解除すると、再利用することはできません。新しい CS_COMMAND 構造体を割り付けるには、アプリケーションは ct_cmd_alloc を呼び出します。

- コマンド構造体の割り付けを解除する前に、アプリケーションはすべてのアクティブ・コマンドのキャンセル、すべての未処理の結果の処理またはキャンセル、およびコマンド構造体のすべてのオープン・カーソルのクローズおよび割り付けの解除を行ってください。

**参照**

ct_command、ct_cmd_alloc

# ct_cmd_props

**説明**

コマンド構造体プロパティを設定または取得します。コマンドを再送するアプリケーションによって使用されます。

**構文**

```
CS_RETCODE ct_cmd_props(cmd, action, property, buffer,
            buflen, outlen)

CS_COMMAND   *cmd;
CS_INT       action;
CS_INT       property;
CS_VOID      *buffer;
CS_INT       buflen;
CS_INT       *outlen;
```

**パラメータ**

*cmd*

クライアント／サーバ・オペレーションを管理する CS_COMMAND 構造体を指すポインタです。

*action*

次の記号値のいずれかです。

| action の値 | 意味 |
|---|---|
| CS_SET | プロパティの値を設定する。 |
| CS_GET | プロパティの値を取得する。 |
| CS_CLEAR | Client-Library デフォルト値へのリセットによって、プロパティの値をクリアする。 |

*property*

値を設定または取得するプロパティの記号名です。「プロパティ」には、すべての Client-Library プロパティが示されています。ct_cmd_props で有効なプロパティの一覧については、表 3-7 (417 ページ ) を参照してください。

*buffer*

プロパティ値を設定する場合、*buffer* はプロパティの設定に使用する値を指します。

プロパティ値を取得する場合、*buffer* は ct_cmd_props が要求された情報を入れる領域を指します。

*buflen*

一般に、*buflen* は **buffer* のバイト単位の長さです。

プロパティ値を設定する場合に **buffer* の値が null で終了する文字列だと、*buflen* は CS_NULLTERM として渡されます。

**buffer* が固定長または記号値の場合、*buflen* を CS_UNUSED として渡します。

*outlen*

整数変数を指すポインタです。

プロパティ値を設定する場合、*outlen* は使用されません。この場合、NULL として渡してください。

プロパティ値を取得するのに *outlen* を指定した場合、ct_cmd_props は要求された情報の長さをバイト単位で **outlen* に設定します。

要求された情報が *buflen* バイトよりも大きい場合、呼び出しは失敗します。アプリケーションは、情報の保持に必要なバイト数を判断するために、**outlen* の値を使用できます。

**戻り値**

ct_cmd_props は、次の値を返します。

| 戻り値 | 意味 |
| --- | --- |
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

**例**

コマンド・レベルのユーザ・データの例

- 次のコード例では、CS_USERDATA プロパティ値を取得します。このコードは、*ex_alib.c* サンプル・プログラムからの抜粋です。ct_cmd_props を使用するその他の例については、*rpc.c* サンプル・プログラムを参照してください。

```
/*
** Extract the user area out of the command handle.
*/
retstat = ct_cmd_props(cmd, CS_GET, CS_USERDATA,
    &ex_async, CS_SIZEOF(ex_async), NULL);
```

```
 if (retstat != CS_SUCCEED)
 {
     return retstat;
}
```

*カーソル・ステータスの例*

- 次のコード例では、関数 *cursor_status* を示します。この関数は ct_cmd_props を呼び出して、Client-Library のカーソルに関するステータス情報を取得します。

```
#define RETURN_IF(a,b) if (a != CS_SUCCEED)¥
   { fprintf(stderr, "Error in:%s line %d¥n", ¥
             b, __LINE__); return a ;}

 /*
 **   cursor_status() -- Print status information about the
 **   Client-Library cursor (if any) declared on a CS_COMMAND
 **   structure.
 **
 ** PARAMETERS:
 **   cmd -- an allocated CS_COMMAND structure.
 **
 **
 ** RETURNS
 **   CS_FAIL if an error occurred.
 **   CS_SUCCEED if everything went ok.
 */

 CS_RETCODE
 cursor_status(cmd)
 CS_COMMAND *cmd;
 {
   CS_RETCODE    ret;
   CS_INT        cur_status;
   CS_INT        cur_id;
   CS_CHAR       cur_name[CS_MAX_NAME];
   CS_CHAR       updatability[CS_MAX_NAME];
   CS_CHAR       status_str[CS_MAX_NAME];
   CS_INT        outlen;

   /*
   ** Get the cursor status property.
   */
   ret = ct_cmd_props(cmd, CS_GET, CS_CUR_STATUS, &cur_status,
                      CS_UNUSED, (CS_INT *) NULL);
   RETURN_IF(ret, "cursor_status:ct_cmd_props(CUR_STATUS)");
```

```
/*
** Is there a cursor?
** Note that CS_CURSTAT_NONE is not a bitmask, but the
** other values are.
*/
if (cur_status == CS_CURSTAT_NONE)
  fprintf(stdout,
     "cursor_status:no cursor on this command structure¥n");
else
{
  /*
 ** A cursor exists, so check its state. Is it
 ** declared, opened, or closed?
 */
 if ((cur_status & CS_CURSTAT_DECLARED)
      == CS_CURSTAT_DECLARED)
    strcpy(status_str, "declared");
  if ((cur_status & CS_CURSTAT_OPEN) == CS_CURSTAT_OPEN)
    strcpy(status_str, "open");
  if ((cur_status & CS_CURSTAT_CLOSED) == CS_CURSTAT_CLOSED)
    strcpy(status_str, "closed");

  /*
  ** Is the cursor updatable or read only?
  */
  if ((cur_status & CS_CURSTAT_RDONLY) == CS_CURSTAT_RDONLY)
    strcpy(updatability, "read only");
  else if ((cur_status & CS_CURSTAT_UPDATABLE)
          == CS_CURSTAT_UPDATABLE)
    strcpy(updatability, "updatable");
  else
    updatability[0] = '¥0';

  /*
  ** Get the cursor id.
  */
  ret = ct_cmd_props(cmd, CS_GET, CS_CUR_ID, &cur_id,
                     CS_UNUSED, (CS_INT *) NULL);
  RETURN_IF(ret, "cursor_status:ct_cmd_props(CUR_ID)");

  /*
  ** Get the cursor name.
  */
  ret = ct_cmd_props(cmd, CS_GET, CS_CUR_NAME, cur_name,
                     CS_MAX_NAME, &outlen);
```

```
    RETURN_IF(ret, "cursor_status:ct_cmd_props(CUR_NAME)");

    /*
    ** Null terminate the name.
    */
    if (outlen < CS_MAX_NAME)
      cur_name[outlen] = '¥0';
    else
      RETURN_IF(CS_FAIL, "cursor_status:name too long");

    /* Print it all out                                     */
    fprintf(stdout, "Cursor '%s' (id %d) is %s and %s.¥n",
            cur_name, cur_id, updatability, status_str);
  }
  return CS_SUCCEED;
} /* cursor_status */
```

*スクロール可能カーソル・ステータスの例*

- スクロール可能カーソルのコード例は上の例と同じですが、次の部分が異なります。

```
    /*
    ** Is the cursor scrollable or read-only?
    */
    if (( cur_status & CS_CURSTAT_SCROLLABLE == CS_CURSTAT_SCROLLABLE )
        strcpy(updatability, "scrollable")
    else if ((cur_status & CS_CURSTAT_RDONLY) == CS_CURSTAT_RDONLY)
        strcpy(updatability, "read-only")
    else
        updatability[0] = '¥0';
```

**使用法**

*action*、*buffer*、*buflen*、*outlen* については、『Open Client Client-Library/C プログラマーズ・ガイド』の「第 2 章 構造体、定数、規則の説明」を参照してください。

- コマンド構造体プロパティは、コマンド構造体レベルでのアプリケーションの動作に影響を与えます。
- 接続に割り付けられたすべてのコマンド構造体は、デフォルト・プロパティ値をその親接続から取得します。アプリケーションは、ct_cmd_props の呼び出しによって、これらのデフォルト値を上書きできます。

  接続にコマンド構造体を割り付けた後、アプリケーションが接続プロパティ値を変更した場合、既存のコマンド構造体はその新しいプロパティ値を取得しません。接続に割り付けられた新しいコマンド構造体は、その新しいプロパティ値をデフォルトとして使用します。

- 「プロパティ」(208 ページ ) を参照してください。
- アプリケーションは、表 3-7 で示すプロパティを設定または取得するために ct_cmd_props を使用できます。

**表 3-7：コマンド構造体のプロパティ**

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
|---|---|---|---|---|
| CS_CMD_ SUPPRESS_FMT | ロー・フォーマットのキャッシュが有効か無効か。ロー・フォーマットのキャッシュが有効な場合、動的 SQL 文が起動されるたびにデータ・サーバはロー・フォーマット情報を送信しない。 | CS_TRUE または CS_FALSE。 デフォルトは適用されない。 | コマンド、接続。 | |
| CS_CUR_ID | カーソルの識別番号。 | 整数値。 | コマンド。 | CS_CUR_STATUS が既存のカーソルを示した後は読み込み専用。 |
| CS_CUR_NAME | アプリケーションの ct_cursor(CS_CUR SOR_DECLARE) 呼び出しで定義されたカーソルの名前。 | null で終了する文字列。 | コマンド。 | ct_cursor(CS_CURSOR_ DECLARE) が CS_SUCCEED を返した後は取得のみ可能。 |
| CS_CUR_ ROWCOUNT | カーソル・ローの現在値。カーソル・ローは、内部フェッチ要求ごとに Client-Library へ返されるロー数。 | 整数値。 | コマンド。 | CS_CUR_STATUS が既存のカーソルを示した後は読み込み専用。 |
| CS_CUR_STATUS | カーソルのステータス。 | CS_INT の値。 使用可能な値については、「カーソル・ステータス」(243 ページ ) を参照。 | コマンド。 | 取得のみ可能。 |
| CS_HAVE_BINDS | 現在の結果セットに対して保存されている結果バインドがあるかどうか。 | CS_TRUE または CS_FALSE。 デフォルトは適用されない。 | コマンド。 | 取得のみ可能。 |
| CS_HAVE_CMD | コマンド構造体に対して、再送可能なコマンドがあるかどうか。 | CS_TRUE または CS_FALSE。 | コマンド。 | 取得のみ可能。 |

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
|---|---|---|---|---|
| CS_HAVE_CUROPEN | リストア可能なカーソル・オープン・コマンドがコマンド構造体に対して存在するかどうか。 | CS_TRUE または CS_FALSE。 | コマンド。 | 取得のみ可能。 |
| CS_HIDDEN_KEYS | 隠されたキーを公開するかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続、コマンド。 | 結果が未処理またはカーソルがオープンしている場合は、コマンド・レベルでの設定はできない。 |
| CS_PARENT_HANDLE | コマンド構造体の親接続のアドレス。 | アドレス。 | 接続、コマンド。 | 取得のみ可能。 |
| CS_STICKY_BINDS | コマンドが繰り返し実行されるときに結果項目とプログラム変数間のバインドが継続するかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コマンド。 | |
| CS_USERDATA | ユーザ割り付けデータ。 | ユーザ割り付けデータ。 | 接続、コマンド。<br>コンテキスト・レベルで USERDATA を設定するには、cs_config を呼び出す。 | なし。 |

参照　ct_config、ct_cmd_alloc、ct_con_props、ct_res_info

# ct_command

説明　言語、パッケージ、RPC、メッセージ、またはデータ送信コマンドを開始します。

**構文**

```
CS_RETCODE ct_command(cmd, type, buffer, buflen,
          option)

CS_COMMAND  *cmd;
CS_INT      type;
CS_VOID     *buffer;
CS_INT      buflen;
CS_INT      option;
```

**パラメータ**

*cmd*

クライアント／サーバ・オペレーションを管理する CS_COMMAND 構造体を指すポインタです。

*type*

開始するコマンドのタイプです。表 3-9 に、*type* の記号値を示します。

*buffer*

データ領域を指すポインタです。表 3-9 に、**buffer* 値のデータ型と意味を示します。

*buflen*

**buffer* データのバイト単位の長さ、または、**buffer* が固定長または記号値を表す場合は、CS_UNUSED です。

*option*

このコマンドに関連するオプションです。

言語、RPC ( リモート・プロシージャ・コール )、送信データ、およびバルク・データ送信コマンドがオプションを持ちます。他のすべてのタイプのコマンドでは、*option* を CS_UNUSED として渡してください。

次の表に、*option* の記号値を示します。

**表 3-8：ct_command option パラメータの値**

| type の値 | option の値 | 意味 |
|---|---|---|
| CS_LANG_CMD | CS_MORE | バッファ内のテキストは、実行される言語コマンドの一部だけである。 |
| | CS_END | バッファ内のテキストは、実行される言語コマンドの最後の部分である。 |
| | CS_UNUSED | CS_END と同じ。 |
| CS_RPC_CMD | CS_RECOMPILE | 実行する前にストアド・プロシージャを再コンパイルする。 |
| | CS_NO_RECOMPILE | 実行する前にストアド・プロシージャを再コンパイルしない。 |
| | CS_UNUSED | CS_NO_RECOMPILE と同じ。 |
| CS_SEND_DATA_CMD | CS_COLUMN_DATA | データは、text または image カラムの更新に使用される。 |
| | CS_BULK_DATA | Sybase の内部使用のみ。データは、バルク・コピー・オペレーションに使用される。 |
| CS_SEND_BULK_CMD | CS_BULK_INIT | Sybase の内部使用のみ。バルク・コピー・オペレーションを初期化する。 |
| | CS_BULK_CONT | Sybase の内部使用のみ。バルク・コピー・オペレーションを続行する。 |

**戻り値**

ct_command は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

### 例

```
/*
 ** ex_execute_cmd()
 **
 ** Type of function:
 **      example program utility api
 **
 ** Purpose:
 ** Sends a language command to the server.
 */
 CS_RETCODE CS_PUBLIC
 ex_execute_cmd(connection, cmdbuf)
 CS_CONNECTION   *connection;
 CS_CHAR         *cmdbuf;
 {
 CS_RETCODE      retcode;
 CS_INT          restype;
 CS_COMMAND      *cmd;
 CS_RETCODE      query_code;

   /*
    ** Get a command structure,store the command string in it,
    ** and send it to the server.
    */
    if ((retcode = ct_cmd_alloc(connection, &cmd)) !=
         CS_SUCCEED)
    {
        ex_error("ex_execute_cmd:ct_cmd_alloc() failed");
        return retcode;
    }

   if ((retcode = ct_command(cmd, CS_LANG_CMD, cmdbuf,
         CS_NULLTERM, CS_UNUSED)) != CS_SUCCEED)
    {
        ex_error("ex_execute_cmd:ct_command() failed");
        (void)ct_cmd_drop(cmd);
        return retcode;
    }
   /* Now send the command and process the results */
    ... ct_send, ct_results, and so forth deleted ...
   return CS_SUCCEED;
 }
```

このコードは、*exutils.c* サンプル・プログラムからの抜粋です。

### 使用法

表 3-9 に ct_command の使用法を示します。

**表 3-9：ct_command パラメータの一覧**

| type の値 | Command initiated | buffer の値 | buflen |
|---|---|---|---|
| CS_LANG_CMD | 言語コマンド | 言語コマンド・テキストのすべてまたは一部を含む CS_CHAR 文字列を指すポインタ。<br>CS_MORE および CS_END オプションを使用して、コマンド・テキストを部分ごとに構築する。詳細については、表 3-8 を参照。 | **buffer* データまたは CS_NULLTERM の長さ。 |
| CS_MSG_CMD | メッセージ・コマンド | メッセージ ID を含む CS_INT 変数を指すポインタ。 | CS_UNUSED。 |
| CS_PACKAGE_CMD | パッケージ・コマンド | パッケージ名を含む CS_CHAR 配列を指すポインタ。 | **buffer* データまたは CS_NULLTERM の長さ。 |
| CS_RPC_CMD | リモート・プロシージャ・コール・コマンド | リモート・プロシージャ名を含む CS_CHAR 配列を指すポインタ。 | **buffer* データまたは CS_NULLTERM の長さ。 |
| CS_SEND_DATA_CMD | データ送信コマンド | NULL。 | CS_UNUSED。 |
| CS_SEND_DATA_NOCMD | データ送信コマンド | NULL。 | CS_UNUSED。 |
| CS_SEND_BULK_CMD | Sybase 内部用のバルク・データ送信コマンド | データベース・テーブル名を含む CS_CHAR 配列を指すポインタ。 | **buffer* データまたは CS_NULLTERM の長さ。 |

- ct_command は、いくつかのタイプのサーバ・コマンドを開始します。

  Client-Library のコマンド・タイプの概要については、『Open Client Client-Library/C プログラマーズ・ガイド』の「第 5 章 コマンド・タイプの選択」を参照してください。

- コマンドを開始するのは、コマンドをサーバに送信する最初の手順です。クライアント・アプリケーションがサーバ・コマンドを実行するために、Client-Library はコマンドを、サーバに送信できる記号コマンド・ストリームに変換する必要があります。コマンド・ストリームには、コマンドのタイプに関する情報と実行に必要なデータが含まれます。たとえば、RPC コマンドはプロシージャ名、パラメータ数、( 必要に応じて ) パラメータ値を必要とします。

ct_command を使用してサーバ・コマンドを実行する手順は、次のとおりです。

a　ct_command を呼び出すことによって、そのコマンドを開始します。この手順は、サーバに送信するコマンド・ストリームの作成に使用される内部構造体を設定します。

b　必要に応じて、コマンドが必要とする各パラメータに対して ct_param または ct_setparam を一度呼び出し、コマンドにパラメータを渡します。

　　すべてのコマンドがパラメータを必要とするわけではありません。たとえば、リモート・プロシージャ・コール・コマンドでは、呼び出されるストアド・プロシージャに応じて、パラメータが必要な場合と不必要な場合があります。

c　ct_send を呼び出して、コマンドをサーバに送信します。ct_send はコマンド構造体の親接続に記号コマンド・ストリームを書き込みます。

d　CS_SUCCEED を返さなくなるまで、繰り返し ct_results を呼び出して、コマンドの結果を処理します。結果処理の詳細については、『Open Client Client-Library/C プログラマーズ・ガイド』の「第 6 章 結果処理コードの書き方」を参照してください。

- アプリケーションが前の実行の結果を処理した後ですぐに ct_send を呼び出すことによって、送信データコマンドおよび送信バルク・コマンド以外の ct_command コマンド・タイプを再送信できます。アプリケーションが ct_command、ct_cursor、ct_dynamic、または ct_sendpassthru を使用して新しいコマンドを開始するまで、Client-Library は開始されたコマンドの情報をコマンド構造体に保管します。ct_setparam を使用してコマンドに対するパラメータ・ソース変数を指定した場合、アプリケーションは、再び ct_setparam を呼び出さずに、パラメータ値を変更できます。「コマンドの再送信」(643 ページ ) を参照してください。

- アプリケーションは、*type* に CS_CANCEL_ALL を指定した ct_cancel を呼び出すことによって、開始しているが送信されていないコマンドをクリアできます。

- ct_command の使用には、次の規則が適用されます。

  - コマンド構造体が開始されるとき、アプリケーションでは、すでに開始されているコマンドを送信するかクリアしなければ、ct_command、ct_cursor、ct_dynamic、または ct_sendpassthru を使用して新しいコマンドを開始することはできません。

- コマンドを送信した後、アプリケーションでは、そのコマンドの実行から返されたすべての結果を完全に処理するかキャンセルしなければ、同じコマンド構造体で新しいコマンドを開始することはできません。
- アプリケーションは、カーソルを管理しているコマンド構造体に対して ct_command を呼び出してコマンドを開始することはできません。アプリケーションは、まず、カーソルの割り付けを解除する必要があります ( または、別のコマンド構造体を使用します )。

- 次の各記述には、ct_command が開始できるコマンドのタイプについての情報が含まれています。

**言語コマンド**

- 言語コマンドには、サーバ固有の言語で記述された 1 つまたは複数のコマンドを表す文字列が含まれています。たとえば、次の言語コマンドには、Transact-SQL コマンドが含まれています。

```
ct_command(cmd, CS_LANG_CMD,
           "select * from pubs2..authors",
          CS_NULLTERM,
          CS_UNUSED);
```

- ct_command の *option* 値として CS_MORE および CS_END を指定すると、アプリケーションは言語バッファにテキストを追加できます。言語コマンド・テキストは、連続する呼び出しを使用して断片をまとめることができます。
- 言語バッファでは、複数のサーバ・コマンドを表すことができます。たとえば、次のシーケンスは、3 つの Transact-SQL 文を含む言語コマンドを構築します。

```
ct_command(cmd, CS_LANG_CMD,
           "select * from pubs2..titles ",
           CS_NULLTERM, CS_MORE);
 ct_command(cmd, CS_LANG_CMD,
           "select * from pubs2..authors ",
           CS_NULLTERM, CS_MORE);
 ct_command(cmd, CS_LANG_CMD,
           "select * from pubs2..publishers ",
           CS_NULLTERM, CS_END);
```

コマンド・バッファへの追加時に、ct_command は空白を追加しないので、ユーザが空白を指定する必要があります。

- CS_UNUSED オプションが指定された場合、Client-Library では、アプリケーションが ct_command に対する一度の呼び出しで言語テキストをすべて渡す必要があります。
- 言語コマンドは、指定されたサーバが理解できる言語であれば、どの言語でもかまいません。Adaptive Server Enterprise は Transact-SQL を理解しますが、Server-Library を使用して構築する Open Server アプリケーションは、任意の言語を理解するよう作成できます。
- 言語コマンド文字列がホスト変数を含む場合、アプリケーションは、言語文字列に含まれる各変数ごとに一度ずつ ct_param または ct_setparam を呼び出すことによって、これらの変数の値を渡すことができます。サーバにコマンドを 2 回以上送信する場合、ct_setparam を使用してください。「コマンドの再送信」(643 ページ ) を参照してください。
- Transact-SQL 変数は、@ 記号で始めてください。
- アプリケーションが ct_command を呼び出して、Adaptive Server Enterprise の言語カーソルに対して fetch 言語コマンドを実行すると、そのカーソルによって通常ロー結果セットが生成されます。Transact-SQL の fetch コマンドは、通常ロー結果を生成します。この通常ロー結果には、言語カーソルに対して設定されている現在の「カーソル・ロー」と等しいローの番号が含まれています。

**メッセージ・コマンド**

- メッセージ・コマンドおよび結果は、特殊化された情報をお互いに通信する方法をクライアントとサーバに提供します。メッセージ・コマンドは RPC コマンドと似ていますが、メッセージ・コマンドは RPC 名ではなく、「**メッセージ ID**」と呼ばれる番号によって識別します。
- メッセージ・コマンドは、アプリケーションが CS_INT 変数で提供する ID を持ちます。アプリケーションは CS_INT のアドレスを ct_command の *buffer* パラメータとして渡します。
- 独自の Open Server アプリケーションはユーザ定義メッセージに対応するイベント・ハンドラを使用してプログラミングできます。ユーザ定義メッセージに対する ID は、CS_USER_MSGID 以上、CS_USER_MAX_MSGID 以下でなければなりません。
- メッセージがパラメータを必要とする場合、アプリケーションは、メッセージに必要なパラメータごとに一度ずつ ct_param または ct_setparam を呼び出します。RPC コマンドをサーバに 2 回以上送信する場合、ct_setparam を使用します。「コマンドの再送信」(643 ページ ) を参照してください。

**パッケージ・コマンド**

- パッケージ・コマンドは、IBM DB/2 データベース・サーバにパッケージの実行を命令します。パッケージは、リモート・プロシージャに似ています。パッケージには、そのパッケージを呼び出すと 1 つの単位として実行される、プリコンパイルされた SQL 文が含まれます。
- パッケージがパラメータを必要とする場合、アプリケーションは、パッケージに必要なパラメータごとに一度ずつ ct_param または ct_setparam を呼び出すことができます。パッケージ・コマンドをサーバに 2 回以上送信する場合、ct_setparam を使用します。「コマンドの再送信」(643 ページ ) を参照してください。

**RPC コマンド**

- RPC ( リモート・プロシージャ・コール ) コマンドは、このサーバ上またはリモート・サーバ上のストアド・プロシージャの実行をサーバに命令します。
- アプリケーションは、**buffer* をストアド・プロシージャの名前に設定して ct_command を呼び出して実行することによって RPC コマンドを開始します。
- パラメータを必要とするストアド・プロシージャの実行時に、アプリケーションが RPC コマンドを使用する場合、そのアプリケーションは、ストアド・プロシージャに必要なパラメータごとに一度ずつ ct_param または ct_setparam を呼び出す必要があります。RPC コマンドをサーバに 2 回以上送信する場合、ct_setparam を使用します。「コマンドの再送信」(643 ページ ) を参照してください。
- アプリケーションは ct_send を使用して RPC コマンドを送信した後、ct_results と ct_fetch を使用してストアド・プロシージャの結果を処理します。ct_results と ct_fetch は、ストアド・プロシージャによって生成された結果ローの処理と、そのプロシージャからのリターン・パラメータやステータスがある場合はそれらの処理の両方に使用されます。
- ストアド・プロシージャを呼び出す代替方法に、Transact-SQL の execute 文を含む言語コマンドを実行する方法があります。言語コマンドを使用してストアド・プロシージャを実行するときは、( 必要に応じて ) パラメータ値を文字フォーマットに変換し、言語コマンド・テキストの一部として渡します。または、パラメータ値をホスト変数として言語コマンドに含めます。RPC コマンドで ct_param または ct_setparam を使用して、宣言されたデータ型にパラメータを渡すことができます。

### データ送信コマンド

- アプリケーションは、大量の text または image データをサーバに書き込むために、データ送信コマンドを使用します。
- アプリケーションは、通常、次を呼び出します。
    - ct_command：データ送信コマンドを開始する。
    - ct_data_info：オペレーションの I/O 記述子を設定する。
    - ct_send_data：値をデータ・ストリームにまとまりの単位で書き込む。
    - ct_send：コマンドをサーバに送信する。
- データ送信コマンドは、再送信できません。
- 「text および image データの処理」(328 ページ ) を参照してください。

### バルク・データ送信コマンド

- 内部的に、Sybase は、Bulk-Library ルーチンの実装の一部として、バルク・データ送信コマンドを使用します。
- バルク・データ送信コマンドは、再送信できません。

### コマンドの抑制

text または image カラムを更新するには、通常、クライアント・アプリケーションでは ct_command ルーチンを呼び出し、データ送信コマンドを開始します。その後、クライアントは ct_data_info コマンドを呼び出し、CS_IODESC を取得し、後続の ct_send_data ルーチンの呼び出しで生成する適切な SQL コマンド (update または writetext) を判断します。

この処理を単純化しパフォーマンスを向上させるために、クライアントは SQL コマンド (update または writetext) の生成を抑制し、サーバのバルク・ハンドラに直接データを送信することもできます。その場合、クライアントは *type* パラメータを CS_SEND_DATA_NOCMD に設定し、ct_command ルーチンを呼び出すことによってデータ送信コマンドを開始しなければなりません。その後、クライアント・アプリケーションは、データ送信コマンドを使用してサーバのバルク・ハンドラに text データのみまたは image データのみを送信できます。サーバでバルク・イベントが発生すると、送信する合計バイト数を示す 4 バイトのフィールドに続き、text または image データが送信されます。バルク・ハンドラは srv_text_info を使用して予想される合計バイト数を読み取り、srv_get_data を使用してデータを読み取ります。

サーバは sp_mda を定義して、SQL コマンドを使用せずに text データまたは image データのみを送信する ct_send_data ルーチンをそれがサポートするかどうかを指定しなければなりません。サーバの sp_mda プロシージャは、ct_connect ルーチンの呼び出し前に、クライアント・アプリケーションで ct_con_props(CS_SENDDATA_NOCMD) などの特定のプロパティが設定されている場合にのみ呼び出されます。これらのいずれかのプロパティ (CS_PARTIAL_TEXT や CS_SENDDATA_NOCMD 接続プロパティなど ) が設定されている場合、ct_connect の実行時にサーバの sp_mda プロシージャが呼び出されます。SQL コマンドを使用せずに text データのみまたは image データのみを送信する ct_send_data ルーチンがサーバでサポートされていないことが sp_mda で指定されている場合は、*type* パラメータを CS_SEND_DATA_NOCMD に設定した ct_command ルーチンの呼び出しは失敗します。

サーバが SQL コマンドを使用せずに text または image データを受信できる場合は、sp_mda は次の結果を返します。

| パラメータ | 値 |
|---|---|
| mdinfo | 「SENDDATA_NOCMD」 |
| querytype | 2 |
| query | senddata no cmd |

**注意**　Adaptive Server では、SQL コマンドを使用せずに image データまたは text データを受信することはできません。

参照

ct_cmd_alloc、ct_cursor、ct_dynamic、ct_param、ct_send、ct_setparam

# ct_compute_info

説明

計算結果の情報を取得します。

構文

```
CS_RETCODE ct_compute_info(cmd, type, colnum, buffer,
                buflen, outlen)
CS_COMMAND        *cmd;
CS_INT            type;
CS_INT            colnum;
CS_VOID           *buffer;
CS_INT            buflen;
CS_INT            *outlen;
```

**パラメータ**

*cmd*

クライアント／サーバ・コマンドを管理する CS_COMMAND 構造体を指すポインタです。

*type*

返される情報のタイプです。*type* の記号値の一覧表については、表 3-10 ( 430 ページ ) を参照してください。

*colnum*

計算ロー結果セットに現れる、対象となる計算カラムの番号です。計算カラムは、select 文の compute 句に指定された順序で現れます。最初のカラムが番号 1、次が番号 2、以下同様に続きます。

*buffer*

ct_compute_info が、要求された情報を入れる領域を指すポインタです。

要求された情報を保持できるほど **buffer* が大きくないことを *buflen* が示している場合、ct_compute_info は CS_FAIL を返します。

*buflen*

**buffer* データ領域のバイト単位の長さか、**buffer* が固定長または記号値を表す場合は CS_UNUSED です。

*outlen*

整数変数を指すポインタです。

**戻り値**

ct_compute_info は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

**例**

次のコマンドが実行されたと仮定します。

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

1　次のような呼び出しがあるとします。

```
CS_INT       mybuffer;
ct_compute_info(cmd, CS_BYLIST_LEN, CS_UNUSED,
          &mybuffer, CS_UNUSED, CS_UNUSED);
```

この呼び出しでは、bylist に 2 つの項目があるので、*mybuffer* は 2 に設定されます。

2 次のような呼び出しがあるとします。

```
CS_SMALLINT    mybuffer[2];
 CS_INT          outlength;
ct_compute_info(cmd, CS_COMP_BYLIST, CS_UNUSED,
         mybuffer, sizeof(mybuffer), &outlength)
```

この呼び出しでは、bylist が select リストのカラム 1 と 2 から構成されることを示すために、CS_SMALLINT 値 1 および 2 が *mybuffer[0]* と *mybuffer[1]* にコピーされます。

3 次のような呼び出しがあるとします。

```
CS_INT      mybuffer;
ct_compute_info(cmd, CS_COMP_COLID, 1, &mybuffer,
        CS_UNUSED, NULL);
```

この呼び出しでは、*name* が select リストの第 2 カラムなので、*mybuffer* は 2 に設定されます。

4 次のような呼び出しがあるとします。

```
CS_INT      mybuffer;
ct_compute_info(cmd, CS_COMP_ID, CS_UNUSED,
         &mybuffer, CS_UNUSED, NULL);
```

この呼び出しでは、select 文に 1 つの compute 句だけがあるので、*mybuffer* は 1 に設定されます。

5 次のような呼び出しがあるとします。

```
CS_INT      mybuffer;
ct_compute_info(cmd, CS_COMP_OP, 1, &mybuffer,
        CS_UNUSED, NULL);
```

この呼び出しでは、最初の計算カラムの集合演算子が *count* なので、*mybuffer* は、記号値 CS_OP_COUNT に設定されます。

**使用法**

表 3-10 に、ct_compute_info の使用法を示します。

**表 3-10：ct_compute_info パラメータの一覧**

| type の値 | colnum の値 | 取得する情報 | *buffer に設定される内容 | *outlen に設定される内容 |
|---|---|---|---|---|
| CS_BYLIST_LEN | CS_UNUSED | bylist 配列の要素番号 | 整数値 | (CS_INT) のサイズ |
| CS_COMP_BYLIST | CS_UNUSED | この計算ローを生成した bylist を含む配列 | CS_SMALLINT 値の配列 | 配列の長さ ( バイト単位 ) |

| type の値 | colnum の値 | 取得する情報 | *buffer に設定される内容 | *outlen に設定される内容 |
|---|---|---|---|---|
| CS_COMP_COLID | 計算カラムのカラム番号 | その計算カラムが取り出されたカラムの select リスト・カラム ID | 整数値 | (CS_INT) のサイズ |
| CS_COMP_ID | CS_UNUSED | 現在の計算ローの計算 ID | 整数値 | (CS_INT) のサイズ |
| CS_COMP_OP | 計算カラムのカラム番号 | 計算カラムの集合演算子タイプ | 次の記号値のいずれかです。<br>CS_OP_SUM<br>CS_OP_AVG<br>CS_OP_COUNT<br>CS_OP_MIN<br>CS_OP_MAX | (CS_INT) のサイズ |

- 計算ローは select 文の compute 句の結果として生成されます。compute 句は、その by カラム・リストの値が変わるたびに、計算ローを生成します。計算ローには、compute 句の集合演算子ごとに 1 つのカラムが含まれます。select 文が複数の compute 句を含む場合、個別の計算ローがそれぞれの句によって生成されます。

  サーバによって返された各計算ローは、個別の結果セットとみなされます。つまり、タイプ CS_COMPUTE_RESULT のそれぞれの結果セットには、1 つのローが含まれます。

- ct_compute_info の呼び出しが有効なのは、計算情報が利用できるとき、つまり、ct_results が CS_COMPUTE_RESULT または CS_COMPUTEFMT を返した後だけです。
- 次に、特定のタイプの計算結果情報について説明します。

#### 計算ローの bylist

- select 文の compute 句には、キーワード by に続けてカラムのリストを指定できます。このリストは、「**bylist**」として知られ、指定されたカラムの値の変更に基づいて、結果をサブグループに分類します。compute 句の集合演算子は各サブグループに適用され、各サブグループで計算ローが生成されます。

#### 計算カラムの select リストのカラム ID

- 計算カラムの select リストのカラム ID は、その計算カラムが取り出されたカラムの select リストの位置です。

**計算ローの計算 ID**

- SQL select 文は、個別の計算ローを返す compute 句を複数持つことができます。select 文の最初の compute 句に対応する計算 ID は、1 です。

**特定計算ロー・カラムの集合演算子**

- *type* が CS_COMP_OP で呼び出されたとき、ct_compute_info は、**buffer* に次の集合演算子タイプのうちの 1 つを設定します。

**表 3-11：集合演算子のタイプ**

| *buffer の設定 | 意味 |
|---|---|
| CS_OP_AVG | 平均集合演算子 |
| CS_OP_COUNT | カウント集合演算子 |
| CS_OP_MAX | 最大集合演算子 |
| CS_OP_MIN | 最小集合演算子 |
| CS_OP_SUM | 合計集合演算子 |

参照

ct_bind、ct_describe、ct_res_info、ct_results

# ct_con_alloc

説明

CS_CONNECTION 構造体を割り付けます。

構文

```
CS_RETCODE ct_con_alloc(context, con_pointer)

CS_CONTEXT        *context;
CS_CONNECTION     **con_pointer;
```

パラメータ

*context*

CS_CONTEXT 構造体を指すポインタです。

*con_pointer*

ポインタ変数のアドレスです。ct_con_alloc は、**con_pointer* に、新しく割り付けた CS_CONNECTION 構造体のアドレスを設定します。

戻り値

ct_con_alloc は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗した。 |

ct_con_alloc が失敗する最も一般的な原因は、メモリ不足です。

**例**

```
/*
 ** DoConnect()
 **
 ** Type of function:
 ** async example program api
 */
 CS_STATIC CS_CONNECTION CS_INTERNAL *
 DoConnect(argc, argv)
 int      argc;
 char     **argv;
 {
     CS_CONNECTION    *connection;
     CS_INT           netio_type = CS_ASYNC_IO;
     CS_RETCODE       retcode;
    /* Open a connection to the server */
     retcode = ct_con_alloc(Ex_context, &connection);
     if (retcode != CS_SUCCEED)
     {
         ex_panic("ct_con_alloc failed");
     }
    /* Set properties for the connection */
     ...ct_con_props calls deleted ...
    /* Open the connection */
     ...ct_connect call deleted.....
 }
```

**使用法**

- CS_CONNECTION 構造体は、接続構造体とも呼ばれ、特定のクライアント／サーバ接続についての情報を格納しています。
- ct_con_alloc を呼び出す前に、アプリケーションは、CS-Library ルーチン cs_ctx_alloc の呼び出しでコンテキスト構造体を割り付け、ct_init の呼び出しで Client-Library を初期化しなければなりません。
- サーバへの接続は 3 段階のプロセスです。サーバに接続するために、アプリケーションは、次のことを行います。

  a　CS_CONNECTION 構造体を割り付けるために ct_con_alloc を呼び出します。

  b　必要に応じて、接続固有プロパティの値を設定するために ct_con_props を呼び出します。

  c　接続を作成してサーバにログインするために、ct_connect を呼び出します。

- アプリケーションは、1 つ以上のサーバに対して複数のオープン接続を同時に持つことができます。

たとえば、アプリケーションは、同時に、サーバ MARS への接続を 2 つ、VENUS への接続を 1 つ、PLUTO への接続を 1 つ持つことができます。ct_config によって設定されるコンテキスト・プロパティ CS_MAX_CONNECT は、コンテキストが許容するオープン接続の最大数を決定します。

それぞれのサーバ接続には、個別の CS_CONNECTION 構造体が必要です。

- コマンドをサーバに送信するには、1 つ以上のコマンド構造体を 1 つの接続に割り付けなければなりません。ct_cmd_alloc は、コマンド構造体を割り付けます。

**参照**

cs_ctx_alloc、ct_cmd_alloc、ct_close、ct_connect、ct_con_props

# ct_con_drop

**説明**

CS_CONNECTION 構造体の割り付けを解除します。

**構文**

```
CS_RETCODE ct_con_drop(connection)
CS_CONNECTION    *connection;
```

**パラメータ**

*connection*

CS_CONNECTION 構造体を指すポインタです。CS_CONNECTION 構造体は、特定のクライアント／サーバ接続の情報を含んでいます。

**戻り値**

ct_con_drop は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

ct_con_drop が失敗する最も一般的な原因は、接続がまだオープンされていることです。

**例**

```
/* ex_con_cleanup() */
CS_RETCODE CS_PUBLIC
 ex_con_cleanup(connection, status)
 CS_CONNECTION      *connection;
 CS_RETCODE         status;
```

```
{
     CS_RETCODE    retcode;
     CS_INT        close_option;
    /* Close connection */
     ...CODE DELETED.....
    retcode = ct_con_drop(connection);
     if (retcode != CS_SUCCEED)
     {
         ex_error("ex_con_cleanup:ct_con_drop()
             failed");
         return retcode;
     }
    return retcode;
}
```

このコードは、*exutils.c* サンプル・プログラムからの抜粋です。

**使用法**

- CS_CONNECTION 構造体の割り付けを解除すると、CS_CONNECTION に関連するすべての CS_COMMAND 構造体の割り付けが解除されます。
- CS_CONNECTION 構造体は、特定のクライアント／サーバ接続の情報を含んでいます。
- 一度 CS_CONNECTION 構造体の割り付けを解除すると、再利用することはできません。新しい CS_CONNECTION を割り付けるために、アプリケーションは ct_con_alloc を呼び出すことができます。
- CS_CONNECTION 構造体が表す接続がクローズされるまで、アプリケーションは、その構造体の割り付けを解除することはできません。接続をクローズするために、アプリケーションは ct_close を呼び出します。
- 接続はエラーによって使用できなくなることがあります。この状態が発生した場合、Client-Library は、その接続を「dead」とマーク付けします。アプリケーションは、接続が「dead」とマーク付けされていないかどうかを判断するために、CS_CON_STATUS プロパティを使用できます。

  接続が「dead」とマーク付けされている場合、アプリケーションは、その接続をクローズするために ct_close(CS_FORCE_CLOSE) を、また、その CS_CONNECTION 構造体を削除するためには、ct_con_drop を呼び出さなければなりません。

例外は、あるタイプの結果処理エラーに対して発生します。結果処理中に接続が「dead」とマーク付けされた場合、アプリケーションから ct_cancel(CS_CANCEL_ALL または CS_CANCEL_ATTN) を呼び出して接続のリカバリを試みることができます。リカバリに失敗した場合、アプリケーションは、接続のクローズおよびその CS_CONNECTION 構造体の解除をしなければなりません。

**参照** ct_con_alloc、ct_close、ct_connect、ct_con_props

# ct_con_props

**説明** 接続構造体のプロパティを設定または取得します。

**構文**

```
CS_RETCODE ct_con_props(connection, action, property,
           buffer, buflen, outlen)

CS_CONNECTION    *connection;
CS_INT           action;
CS_INT           property;
CS_VOID          *buffer;
CS_INT           buflen;
CS_INT           *outlen;
```

**パラメータ**

*connection*

CS_CONNECTION 構造体を指すポインタです。CS_CONNECTION 構造体は、特定のクライアント／サーバ接続の情報を含んでいます。

*action*

次の記号値のいずれかです。

| action の値 | 結果 |
|---|---|
| CS_SET | プロパティの値を設定する。 |
| CS_GET | プロパティの値を取得する。 |
| CS_CLEAR | プロパティをデフォルト値にリセットする。 |
| CS_SUPPORTED | 分散サービス・ドライバがプロパティをサポートするかどうかをチェックする。ディレクトリ・ドライバまたはセキュリティ・ドライバの動作に影響を与えるプロパティだけに使用すること。「プロパティがサポートされているかどうかのチェック」(210 ページ) を参照。 |

*property*

値の設定または取得で使用するプロパティの記号名です。表 3-12 ( 440 ページ ) には、ct_con_props で設定できるプロパティを示します。「プロパティ」(208 ページ ) には、Client-Library の全プロパティを示します。

*buffer*

プロパティ値を設定する場合、*buffer* はプロパティの設定に使用する値を指します。

*buflen*

通常、*buflen* は **buffer* のバイト単位の長さです。

**buffer* が固定長または記号値の場合、*buflen* を CS_UNUSED として渡します。

*outlen*

整数変数を指すポインタです。

プロパティ値を設定する場合、*outlen* は使用されないので、NULL として渡してください。

プロパティ値を取得する場合に outlen を指定した場合、ct_con_props は要求された情報の長さをバイト単位で **outlen* に設定します。

情報が *buflen* バイトより大きい場合、アプリケーションは情報の保持に必要なバイト数を判断するために、**outlen* の値を使用することができます。

**戻り値**

ct_con_props は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

**例**

**例 1** このコードは *blktxt.c* サンプル・プログラムからの抜粋です。

```
/*

 ** EstablishConnection()
 **
 ** Purpose:
 ** This routine establishes a connection to the server
 ** identified in example.h and sets the CS_USER,
 ** CS_PASSWORD, and CS_APPNAME properties for the
 ** connection.
```

```
 **
 ** NOTE:The user name, password, and server are defined
 ** in the example header file.
 */
CS_STATIC CS_RETCODE
 EstablishConnection(context, connection)
 CS_CONTEXT       *context;
 CS_CONNECTION    *connection;
 {
     CS_INT       len;
     CS_RETCODE   retcode;
     CS_BOOL      bool;

    /* Allocate a connection structure */
     ...CODE DELETED.....

    /*
     ** If a user name is defined in example.h, set the
     ** CS_USERNAME property.
     */
     if (retcode == CS_SUCCEED && Ex_username != NULL)
     {
         if ((retcode = ct_con_props(*connection, CS_SET,
             CS_USERNAME, Ex_username, CS_NULLTERM, NULL))
             != CS_SUCCEED)
         {
             ex_error("ct_con_props(username) failed");
         }
     }

    /*
     ** If a password is defined in example.h, set the
     ** CS_PASSWORD property.
     */
     if (retcode == CS_SUCCEED && Ex_password != NULL)
     {
         if ((retcode = ct_con_props(*connection, CS_SET,
             CS_PASSWORD, Ex_password, CS_NULLTERM, NULL))
             != CS_SUCCEED)
         {
             ex_error("ct_con_props(passwd) failed");
         }
     }

    /* Set the CS_APPNAME property */
     ...CODE DELETED.....

    /* Enable the bulk login property */
     if (retcode == CS_SUCCEED)
```

```
                {
                    bool = CS_TRUE;
                    retcode = ct_con_props(*connection, CS_SET,
                        CS_BULK_LOGIN, &bool, CS_UNUSED, NULL);
                    if (retcode != CS_SUCCEED)
                    {
                         ex_error("ct_con_props(bulk_login) failed");
                    }
                }

               /* Open a server connection */
                ...CODE DELETED.....

           }
```

**例 2** 次の例では、CS_SEC_EXTENDED_ENCRYPTION が無効になります。

```
...
CS_INT Ex_encryption = CS_FALSE;
CS_INT Ex_nonencryptionretry = CS_FALSE;
...
main()
{
    ...
    /*

    ** This needs to be called before calling ct_connect()
    */
     ret = ct_con_props(connection, CS_SET, CS_SEC_EXTENDED_ENCRYPTION,
           &Ex_encryption, CS_UNUSED, NULL);
    EXIT_ON_FAIL(context, ret, "Could not set extended encryption");
    ret = ct_con_props(connection, CS_SET, CS_SEC_NON_ENCRYPTION_RETRY,
          &Ex_nonencryptionretry, CS_UNUSED, NULL);
    EXIT_ON_FAIL(context, ret, "Could not set non encryption retry");
    ...
}
```

**使用法**

*action*、*buffer*、*buflen*、*outlen* については、『Open Client Client-Library/C プログラマーズ・ガイド』の「第 2 章 構造体、定数、規則の説明」を参照してください。

- 接続プロパティは、接続レベルでの Client-Library 動作の概要を定義します。特定のプロパティがサポートされているかどうかを確認するには、確立した接続でアプリケーションから ct_con_props を呼び出します。この呼び出しは、CS_SUPPORTED アクション・パラメータと、CS_BOOL 変数のアドレスとしてバッファ・パラメータを使用する必要があります。

- コンテキスト内で作成されたすべての接続は、デフォルト・プロパティ値をその親コンテキストから取得します。アプリケーションは、ct_con_props の呼び出しによって、これらのデフォルト値を上書きできます。

  コンテキストに接続を割り付けた後、アプリケーションがコンテキスト・プロパティ値を変更した場合、既存の接続はその新しいプロパティ値を取得しません。コンテキスト内で割り付けられる新しい接続は、その新しいプロパティ値をデフォルトとして使用します。

- 接続に割り付けられたすべてのコマンド構造体は、デフォルト・プロパティ値をその親接続から取得します。アプリケーションは、コマンド構造体レベルでプロパティ値を設定するために ct_cmd_props の呼び出しによって、これらのデフォルト値を上書きできます。

  接続にコマンド構造体を割り付けた後、アプリケーションが接続プロパティ値を変更した場合、既存のコマンド構造体はその新しいプロパティ値を取得しません。接続に割り付けられた新しいコマンド構造体は、その新しいプロパティ値をデフォルトとして使用します。

- 一部の接続プロパティは、アプリケーションが接続を確立するために、ct_connect を呼び出す前に設定される場合にのみ有効です。これらのプロパティについては、表 3-12 の「注意」欄で示します。
- 「プロパティ」(208 ページ ) を参照してください。
- アプリケーションは、次のプロパティを設定または取得するために ct_con_props を使用できます。

**表 3-12：接続構造体プロパティ**

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
|---|---|---|---|---|
| CS_ANSI_BINDS | ANSI スタイルのバインドを使用するかどうか。 | CS_TRUE または CS_FALSE。 | コンテキスト、接続。 | |
| CS_APPNAME | サーバにログインする場合に使用されるアプリケーション名。 | 文字列。 | コンテキスト、接続。<br>コンテキスト・レベルで設定するには、cs_config を呼び出す。 | ログイン・プロパティ。<br>接続確立後の設定はできない。 |

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
| --- | --- | --- | --- | --- |
| CS_ASYNC_NOTIFS | 接続が、レジスタード・プロシージャ・ノーティフィケーションを非同期に受け取るかどうか。 | CS_TRUE または CS_FALSE。 | 接続。 | |
| CS_BULK_LOGIN | 接続でバルク・コピー・イン・オペレーションを実行可能かどうか。 | CS_TRUE または CS_FALSE。 | 接続。 | ログイン・プロパティ。<br>接続確立後の設定はできない。 |
| CS_CHARSETCNV | 文字セット変換を実行するかどうか。 | CS_TRUE または CS_FALSE。 | 接続。 | 接続確立後は取得のみ可能である。 |
| CS_COMMBLOCK | 通信セッション・ブロックへのポインタ。<br>このプロパティは、IBM370 システムに固有のものであり、他のすべてのプラットフォームでは無視される。 | ポインタ値。 | 接続。 | 接続確立後の設定はできない。 |
| CS_CONNECTED_ADDR | 現在接続が確立されているサーバのトランスポート・アドレス。 | 有効なトランスポート・アドレス。 | 接続。 | このプロパティは設定できない。サーバのアドレスが格納される CS_TRANADDR 構造体を指すポインタが必要。 |
| CS_CON_KEEPALIVE | KEEPALIVE オプションを使用するかどうか。 | CS_TRUE ( デフォルト ) または CS_FALSE。 | コンテキストまたは接続。 | 一部の Net-Library プロトコル・ドライバでは、このプロパティはサポートされていない。そのようなプロトコル・ドライバで接続が確立された後に CS_GET または CS_SET で ct_con_props を呼び出すと、CS_FAIL が返される。 |
| CS_CON_STATUS | 接続のステータス。 | CS_INT 長のビットマスク。 | 接続。 | 取得のみ可能。 |

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
|---|---|---|---|---|
| CS_CON_TCP_NODELAY | TCP_NODELAY オプションを使用するかどうか。 | CS_TRUE ( デフォルト ) または CS_FALSE。 | コンテキストまたは接続。 | 一部の Net-Library プロトコル・ドライバでは、このプロパティはサポートされていない。そのようなプロトコル・ドライバで接続が確立された後に CS_GET または CS_SET で ct_con_props を呼び出すと、CS_FAIL が返される。 |
| CS_CONFIG_BY_SERVERNAME | ct_connect が *server_name* パラメータまたは CS_APPNAME プロパティの値を外部設定データを読み込むセクション名として使用するかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。CS_APPNAME が使用されていることを意味する。 | 接続。 | CS_VERSION_110 以降での初期化が必要。 |
| CS_CONFIG_FILE | Open Client/Server ランタイム設定ファイルの名前とパス。<br>「ランタイム設定ファイルの使い方」(352 ページ ) を参照。 | 文字列。<br>デフォルトは NULL で、プラットフォーム固有のデフォルトを使用する。 | 接続。 | CS_VERSION_110 以降での初期化が必要。 |
| CS_DIAG_TIMEOUT | インライン・エラー処理が有効な場合に、タイムアウト・エラーの発生時に処理をエラー終了するか、またはリトライするか。 | CS_TRUE または CS_FALSE。 | 接続。 | |

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
|---|---|---|---|---|
| CS_DISABLE_POLL | ポーリングを無効にするかどうか。ポーリングをしない場合、ct_poll は非同期オペレーション完了をレポートしない。この場合もレジスタード・プロシージャ・ノーティフィケーションは報告される。 | CS_TRUE または CS_FALSE。 | コンテキスト、接続。 | レイヤ構成の非同期アプリケーションで有効。 |
| CS_DS_COPY | アプリケーションの要求に応えるために、ディレクトリ・サービスがディレクトリ・エントリのキャッシュされたコピーを使用するかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_TRUE で、この場合キャッシュが使用できる。 | 接続。 | すべてのディレクトリ・プロバイダがサポートするのではない。 |
| CS_DS_DITBASE | ディレクトリの検索が開始されるディレクトリ・ノードの完全に修飾された名前。 | 文字列。<br>デフォルトはディクトリ・プロバイダ固有。 | 接続。 | すべてのディレクトリ・プロバイダがサポートするのではない。 |
| CS_DS_EXPAND ALIAS | ディレクトリ・サービスがディレクトリ・エイリアス・エントリを拡張するかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_TRUE で、エイリアスの拡張が可能。 | 接続。 | すべてのディレクトリ・プロバイダがサポートするのではない。 |
| CS_DS_FAILOVER | ディレクトリ・サービス・ドライバが初期化できない場合に、次の *libtcl.cfg* エントリまたは *interfaces* ファイルへのフェールオーバが可能かどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_TRUE。 | 接続。 | |
| CS_DS_PASSWORD | CS_DS_PRINCIPAL として指定されたディレクトリ・ユーザ ID と一緒に使用されるパスワード。 | 文字列。<br>デフォルトは NULL。 | 接続。 | すべてのディレクトリ・プロバイダがサポートするのではない。 |

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
|---|---|---|---|---|
| CS_DS_PRINCIPAL | CS_DS_PASSWORD として指定されたパスワードと対になる、ディレクトリ・サービスを使用するためのディレクトリ・ユーザ ID。 | 文字列。<br>デフォルトは NULL。 | 接続。 | すべてのディレクトリ・プロバイダがサポートするのではない。 |
| CS_DS_PROVIDER | 接続のためのディレクトリ・プロバイダの名前。 | 文字列。<br>デフォルトはディレクトリ・ドライバ設定によって異なる。 | 接続。 | |
| CS_DS_RAND_ OFFSET | 接続リストのランダム・オフセットを有効または無効にする。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | ネットワーク・アドレス・リストがディレクトリ・サービスから取得されるときに決定される。 |
| CS_DS_SEARCH | ディレクトリ検索の深さを制限する。 | CS_INT サイズの記号値。<br>有効値のリストについては、「ディレクトリ・サービスの検索の深さ」(135 ページ) を参照。 | 接続。 | すべてのディレクトリ・プロバイダがサポートするのではない。 |
| CS_DS_SIZELIMIT | ct_ds_lookup を使用して開始された検索によって返されるディレクトリ・エントリの数を制限する。 | 0 以上の CS_INT 値。<br>値 0 は、サイズ制限がないことを示す。 | 接続。 | |
| CS_DS_TIMELIMIT | ディレクトリ検索の完了までの絶対時間制限を秒単位で設定する。 | 0 以上の CS_INT 値。<br>値 0 は、時間制限がないことを示す。 | 接続。 | すべてのディレクトリ・プロバイダがサポートするのではない。 |
| CS_EED_CMD | 拡張エラー・データを含むコマンド構造体へのポインタ。 | ポインタ値。 | 接続。 | 取得のみ可能。 |

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
|---|---|---|---|---|
| CS_ENDPOINT | 接続のファイル記述子。 | 整数値、またはプラットフォームが CS_END POINT をサポートしていない場合は、-1。 | 接続。 | 接続確立後は取得のみ可能である。 |
| CS_EXPOSE_FMTS | CS_ROWFMT_RESULT および CS_COMPUTEFMT_RESULT タイプの結果を公開するかどうか。 | CS_TRUE または CS_FALSE。 | コンテキスト、接続。 | 接続確立後の設定はできない。 |
| CS_EXTENDED_ENCRYPT_CB | 接続で、デフォルト以外のパブリック・キー暗号化ハンドラを使用してパスワードの非対称暗号化が設定されるかどうか。 | CS_TRUE または CS_FALSE。 | 接続。 | 接続確立後の設定はできない。 |
| CS_EXTERNAL_CONFIG | ct_connect が、オープンする接続のプロパティとオプションの設定時に外部設定ファイルを読み込むかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | CS_VERSION_110 以降での初期化が必要。 |
| CS_EXTRA_INF | SQLCA、SQLCODE、および SQLSTATE 構造体を使用して Client-Library メッセージをインライン処理するときに必要となる追加情報を返すかどうか。 | CS_TRUE または CS_FALSE。 | コンテキスト、接続。 | |
| CS_HIDDEN_KEYS | 隠しキーを公開するかどうか。 | CS_TRUE または CS_FALSE。 | コンテキスト、接続、コマンド。 | |
| CS_HOSTNAME | クライアント・マシンのホスト名。 | 文字列。 | 接続。 | ログイン・プロパティ。<br>接続確立後の設定はできない。 |

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
|---|---|---|---|---|
| CS_LOC_PROP | ローカライゼーション情報を定義する CS_LOCALE 構造体。 | アプリケーションによって以前に割り付けられた CS_LOCALE 構造体。 | 接続。 | ログイン・プロパティ。<br>接続確立後の設定はできない。 |
| CS_LOGIN_STATUS | 接続がオープンしているかどうか。 | CS_TRUE または CS_FALSE。 | 接続。 | 取得のみ可能。 |
| CS_LOOP_DELAY | サーバ名に対応するアドレスのシーケンスをリトライする前に ct_connect が待つ秒単位の遅延時間。 | CS_INT >= 0。<br>デフォルトは 0。 | 接続。 | リトライ回数は CS_RETRY_COUNT により指定される。 |
| CS_NETIO | ネットワーク I/O が同期、完全非同期、または遅延非同期のいずれであるか。 | CS_SYNC_IO、CS_ASYNC_IO、CS_DEFER_IO。 | コンテキスト、接続 | 非同期接続は、その親コンテキストと一致する完全または遅延非同期のどちらか。 |
| CS_NOCHARSETCNV_REQD | サーバの文字セットがクライアントの文字セットと異なる場合、サーバで文字セット変換を実行するかどうか。 | CS_TRUE または CS_FALSE。 | 接続。 | 接続確立後の設定はできない。 |
| CS_NOTIF_CMD | レジスタード・プロシージャのノーティフィケーション・パラメータを含むコマンド構造体のポインタ。 | ポインタ値。 | 接続。 | 取得のみ可能。 |
| CS_PACKETSIZE | TDS パケット・サイズ。 | 整数値。 | 接続。 | ネゴシエートされたログイン・プロパティ。<br>接続確立後の設定はできない。 |
| CS_PARENT_HANDLE | 接続構造体の親コンテキストのアドレス。 | アドレス。 | 接続、コマンド。 | 取得のみ可能。 |

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
|---|---|---|---|---|
| CS_PARTIAL_TEXT | クライアント・アプリケーションで部分更新を実行するかどうかを示す。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | このプロパティは、サーバへの接続が確立される前に設定する必要がある。サーバが部分更新をサポートしていない場合は、このプロパティは CS_FALSE に再設定される。 |
| CS_PASSWORD | サーバへのログインに使用するパスワード。 | 文字列。 | 接続。 | ログイン・プロパティ。 |
| CS_PROP_APPLICATION_SPID | Adaptive Server Enterprise SPID は、ログイン時に保存され、プロパティとして使用できる。<br>「拡張フェールオーバ」(246 ページ ) を参照。 | サーバのサーバ・プロセス ID (spid) に対応する CS_INT 値。 | 接続。 | ログイン・プロパティ。 |
| CS_PROP_EXTENDED FAILOVER | サーバが指定したフェールオーバ・ターゲットを有効または無効にする。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_TRUE。 | コンテキスト、接続。 | ログイン・プロパティ。 |
| CS_PROP_MIGRATABLE | 接続マイグレーションを有効または無効にする。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_TRUE。 | コンテキスト、接続。 | ログイン・プロパティ。 |
| CS_PROP_REDIRECT | ログイン・リダイレクト・サポートを有効または無効にする。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_TRUE。 | コンテキスト、接続。 | ログイン・プロパティ。 |
| CS_PROP_SSL_PROTOVERSION | サポートされている SSL/TLS プロトコルのバージョン。 | CS_INT | コンテキスト、接続。 | 次のいずれかの値を指定する。<br>• CS_SSLVER_20<br>• CS_SSLVER_30<br>• CS_SSLVER_TLS1 |
| CS_PROP_SSL_CIPHER | CipherSuite 名をカンマで区切ったリスト。 | CS_CHAR | コンテキスト、接続。 | |

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
|---|---|---|---|---|
| CS_PROP_SSL_LOCALID | ローカル ID ( 証明書 ) ファイルのパスの指定に使用するプロパティ。 | 文字列。 | コンテキスト、接続。 | 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る構造体。 |
| CS_PROP_SSL_CA | 信頼された CA 証明書を含むファイルへのパスを指定する。 | CS_CHAR | コンテキスト、接続。 | |
| CS_RETRY_COUNT | サーバのアドレスへの接続をリトライする回数。 | CS_INT >= 0。<br>デフォルトは 0。 | 接続。 | ログイン・ダイアログの確立だけに影響する。ログインの失敗はリトライされない。 |
| CS_SEC_APPDEFINED | アプリケーションが定義しているチャレンジ／応答セキュリティ・ハンドシェイクを接続が使用するかどうか。 | CS_TRUE または CS_FALSE。 | 接続。 | 接続確立後の設定はできない。 |
| CS_SEC_CHALLENGE | Sybase が定義しているチャレンジ／応答セキュリティ・ハンドシェイクを接続が使用するかどうか。 | CS_TRUE または CS_FALSE。 | 接続。 | 接続確立後の設定はできない。 |
| CS_SEC_CHANBIND | 接続のセキュリティ・メカニズムがチャンネルのバインドを行うかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_CONFIDENTIALITY | 接続でデータの暗号化サービスを実行するかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_CREDENTIALS | 委任されたユーザのクレデンシャルを転送するためにゲートウェイ・アプリケーションで使用される。 | A CS_VOID * ポインタ。 | コンテキスト、接続。 | 読み込み不可能。<br>接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
|---|---|---|---|---|
| CS_SEC_ CREDTIMEOUT | ユーザのクレデンシャルが期限切れかどうか。 | CS_INT。有効な値とその意味については、表 2-33 (297 ページ ) を参照。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_ DATAORIGIN | 接続のセキュリティ・メカニズムがデータ・オリジンの検証をするかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_ DELEGATION | ユーザの委任クレデンシャルを使用してサーバを他のサーバに接続させるかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_ DETECTREPLAY | 接続のセキュリティ・メカニズムがリプレイされた転送を検出するかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_ DETECTSEQ | 接続のセキュリティ・メカニズムが不正なシーケンスの転送を検出するかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_ ENCRYPTION | 接続で、パスワードの対称暗号化を使用するかどうか。 | CS_TRUE または CS_FALSE。 | 接続。 | 接続確立後の設定はできない。 |
| CS_SEC_ EXTENDED_ ENCRYPTION | 接続で、パスワードの非対称暗号化を使用するかどうか。 | CS_TRUE または CS_FALSE。 | 接続。 | 接続確立後の設定はできない。 |

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
|---|---|---|---|---|
| CS_SEC_NON_ ENCRYPTION_ RETRY | サーバがパスワードの対称／非対称暗号化を使用できない場合に、接続でプレーン・テキスト形式のパスワードを使用するかどうか。 | CS_TRUE または CS_FALSE。 | 接続。 | 接続確立後の設定はできない。 |
| CS_SEC_ INTEGRITY | 接続のセキュリティ・メカニズムがデータ整合性のチェックを実行するかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_ KEYTAB | 接続のセキュリティ・メカニズムが CS_USERNAME プロパティと一緒に使用されるセキュリティ・キーを読み込むファイルのパスと名前。 | 文字列。<br>デフォルトは NULL。これは、アプリケーションが ct_connect を呼び出す前に、ユーザがクレデンシャルを確立しておく必要があることを意味する。 | 接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_ MECHANISM | 接続のためのセキュリティ・サービスを実行する、ネットワーク・セキュリティ・メカニズムの名前。 | 文字列値<br>デフォルトはセキュリティ・ドライバ設定によって異なる。 | コンテキスト、接続。 | 接続確立後の設定はできない。 |
| CS_SEC_ MUTUALAUTH | サーバが、クライアントに対してサーバ自体を認証する必要があるかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_ NEGOTIATE | trusted ユーザ・セキュリティ・ハンドシェイクを接続が使用するかどうか。 | CS_TRUE または CS_FALSE。 | 接続。 | 接続確立後の設定はできない。 |

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
|---|---|---|---|---|
| CS_SEC_ NETWORKAUTH | 接続のセキュリティ・メカニズムがネットワークベースのユーザの認証を実行するかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムと、CS_USERNAME と一致する既存のクレデンシャルが必要。 |
| CS_SEC_ SERVERPRINCIPAL | 接続がオープンされるサーバのネットワーク・セキュリティ・プリンシパルの名前。 | 文字列値<br>デフォルトは NULL。このとき ct_connect は、サーバのプリンシパル名がその *server_name* パラメータと同じであると見なす。 | 接続。 | 接続確立後の設定はできない。<br>ネットワークベースのユーザの認証を使用する接続でのみ意味がある。 |
| CS_SEC_ SESSTIMEOUT | 接続のセキュリティ・セッションの期限が切れているかどうか。 | CS_INT。有効な値とその意味については、表 2-33 (297 ページ ) を参照。 | コンテキスト、接続。 | 接続確立後の設定はできない。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SENDDATA_ NOCMD | ct_connect の呼び出し時に sp_mda プロシージャがサーバで実行されるかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | 接続。 | CS_SENDDATA_ NOCMD は、ct_connect を呼び出す前に設定する必要がある。SQL コマンドを使用せずに text データまたは image データのみを送信する ct_send_data ルーチンが、サーバでサポートされない場合、このプロパティは再設定される。 |
| CS_SERVERADDR | 接続先のサーバのアドレス。 | 「hostname portnumber [filter]」の形式。filter はオプション。 | 接続。 | このプロパティを使用すると、ctlib はサーバのホスト名とインタフェースのポート番号をバイパスする。 |
| CS_SERVERNAME | 接続先のサーバの名前。 | 文字列値。 | 接続。 | 接続確立後は取得のみ可能である。 |

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
|---|---|---|---|---|
| CS_TDS_VERSION | 接続が使用している TDS プロトコルのバージョン。 | バージョン・レベル記号。 | 接続。 | ネゴシエートされたログイン・プロパティ。<br>接続確立後の設定はできない。 |
| CS_TEXTLIMIT | この接続で返される最大の text または image 値。 | 整数値。 | コンテキスト、接続。 | |
| CS_TRANSACTION_NAME | Open Server for CICS への接続に使用されるトランザクション名。 | 文字列値。 | 接続。 | |
| CS_USERDATA | ユーザ割り付けデータ。 | ユーザ割り付けデータ。 | 接続、コマンド。 | |
| CS_USERNAME | サーバへのログインに使用する名前。 | 文字列。 | 接続。 | ログイン・プロパティ。<br>接続確立後の設定はできない。 |
| CS_VALIDATE_CB | ct_callback によって登録された Client-Library ルーチン。 | 整数値。 | 接続、コマンド。 | |

**参照** ct_capability、ct_cmd_props、ct_connect、ct_config、ct_init、「プロパティ」(208 ページ )

# ct_config

**説明** コンテキスト・プロパティを設定または取得します。

**構文**

```
CS_RETCODE ct_config(context, action, property,
           buffer, buflen, outlen)

CS_CONTEXT    *context;
CS_INT        action;
CS_INT        property;
CS_VOID       *buffer;
CS_INT        buflen;
CS_INT        *outlen;
```

パラメータ

*context*

CS_CONTEXT 構造体を指すポインタです。

*action*

次の記号値のいずれかです。

| action の値 | 結果 |
|---|---|
| CS_SET | プロパティの値を設定する。 |
| CS_GET | プロパティの値を取得する。 |
| CS_CLEAR | Client-Library デフォルト値へのリセットによって、プロパティの値をクリアする。 |
| CS_SUPPORTED | 分散サービス・ドライバがプロパティをサポートするかどうかをチェックする。ディレクトリ・ドライバまたはセキュリティ・ドライバの動作に影響を与えるプロパティだけに使用すること。「プロパティがサポートされているかどうかのチェック」(210 ページ ) を参照。 |

*property*

値の設定または取得に使用するプロパティの記号名です。表 3-13 ( 455 ページ ) では、Client-Library コンテキスト・プロパティを示します。「プロパティ」(208 ページ ) では、Client-Library の全プロパティを示します。

*buffer*

プロパティ値を設定する場合、*buffer* はプロパティの設定に使用する値を指します。

プロパティ値を取得する場合、*buffer* は、ct_config が要求された情報を入れる領域を指します。

*buflen*

通常、*buflen* は **buffer* のバイト単位の長さです。

プロパティ値を設定する場合、**buffer* の値が null で終了している場合には、*buflen* を CS_NULLTERM として渡します。

**buffer* が固定長値、記号値、または関数の場合、*buflen* を CS_UNUSED として渡してください。

*outlen*

整数変数を指すポインタです。

プロパティ値を設定する場合、*outlen* は使用されません。この場合、NULL として渡してください。

プロパティ値の取得時に *outlen* を指定した場合、ct_config は要求された情報の長さをバイト単位で **outlen* に設定します。

情報が *buflen* バイトより大きい場合、アプリケーションは情報の保持に必要なバイト数を判断するために、**outlen* の値を使用することができます。

**戻り値**

ct_config は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗した。 |

**例**

```
/* Set the input/output type to asynchronous */
 CS_INT        propvalue;
 if (retcode == CS_SUCCEED)
 {
     propvalue = CS_ASYNC_IO;
     retcode = ct_config(*context, CS_SET, CS_NETIO,
         (CS_VOID *)&propvalue, CS_UNUSED, NULL);
     if (retcode != CS_SUCCEED)
     {
         ex_error("ex_init:ct_config(netio) failed");
     }
 }
```

このコードは *exutils.c* サンプル・プログラムからの抜粋です。

**使用法**

*action*、*buffer*、*buflen*、*outlen* については、『Open Client Client-Library/C プログラマーズ・ガイド』の「第 2 章 構造体、定数、規則の説明」を参照してください。

- コンテキスト・プロパティはコンテキスト・レベルでの Client-Library 動作の概要を定義します。
- ct_config は、CS_CONTEXT 内で確立されるすべての接続において、*libtcl*.cfg* ファイルよりも優先されます。
- ct_config は、接続プロパティと、コンテキストを設定する外部ファイルの使用を制御します。「ランタイム設定ファイルの使い方」(352 ページ ) を参照してください。

- コンテキスト内で作成されたすべての接続は、デフォルト・プロパティ値をその親コンテキストから取得します。アプリケーションは、接続レベルでプロパティ値を設定するために ct_con_props を呼び出すことによって、これらのデフォルト値を上書きできます。

  コンテキストに接続を割り付けた後、アプリケーションがコンテキスト・プロパティ値を変更した場合、既存の接続はその新しいプロパティ値を取得しません。コンテキスト内で割り付けられる新しい接続は、その新しいプロパティ値をデフォルトとして使用します。

- コンテキスト・プロパティは 3 種類あります。

  - CS-Library 特有のコンテキスト・プロパティ
  - Client-Library に固有なコンテキスト・プロパティ
  - Server-Library に固有なコンテキスト・プロパティ

  cs_config は、CS-Library 固有のコンテキスト・プロパティの値を設定および取得します。cs_config によって設定されるプロパティは、CS-Library にだけ反映されます。

  ct_config は、Client-Library 固有のコンテキスト・プロパティの値を設定および取得します。ct_config によって設定されるプロパティは、Client-Library にだけ反映されます。

  srv_props は、Server-Library 固有のコンテキスト・プロパティの値を設定および取得します。srv_props によって設定されるプロパティは、Server-Library にだけ反映されます。

- 「プロパティ」(208 ページ ) を参照してください。

- アプリケーションは、次のプロパティを設定または取得するために ct_config を使用できます。

**表 3-13：Client-Library コンテキスト構造体のプロパティ**

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
|---|---|---|---|---|
| CS_ANSI_BINDS | ANSI スタイルのバインドを使用するかどうか。 | CS_TRUE または CS_FALSE。 | コンテキスト、接続。 | |
| CS_DISABLE_POLL | ポーリングを無効にするかどうか。ポーリングをしない場合、ct_poll は非同期オペレーション完了をレポートしない。 | CS_TRUE または CS_FALSE。 | コンテキスト、接続。 | レイヤ構成の非同期アプリケーションで有効。 |

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
| --- | --- | --- | --- | --- |
| CS_DS_RAND_OFFSET | 接続リストのランダム・オフセットを有効または無効にする。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | ネットワーク・アドレス・リストがディレクトリ・サービスから取得されるときに決定される。 |
| CS_EXPOSE_FMTS | CS_ROWFMT_RESULT および CS_COMPUTEFMT_RESULT タイプの結果を公開するかどうか。 | CS_TRUE または CS_FALSE。 | コンテキスト、接続。 | 接続を確立する前に設定する場合にのみ有効。 |
| CS_EXTERNAL_CONFIG | ct_connect が、オープンする接続のプロパティとオプションの設定時に外部設定ファイルを読み込むかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | CS_VERSION_110 以降での初期化が必要。 |
| CS_EXTRA_INF | SQLCA、SQLCODE、および SQLSTATE 構造体を使用して Client-Library メッセージをインライン処理するときに必要となる追加情報を返すかどうか。 | CS_TRUE または CS_FALSE。 | コンテキスト、接続。 | |
| CS_HIDDEN_KEYS | 隠しキーを公開するかどうか。 | CS_TRUE または CS_FALSE。 | コンテキスト、接続、コマンド。 | |
| CS_IFILE | *interfaces* ファイルのパスと名前。 | 文字列。 | コンテキスト。 | |
| CS_LOGIN_TIMEOUT | ログイン・タイムアウト値。 | 整数値。 | コンテキスト。 | |
| CS_MAX_CONNECT | このコンテキストでの最大接続数。 | 整数値。 | コンテキスト。 | |
| CS_MEM_POOL | 割り込みレベルで必要なメモリを確保するために Client-Library が使用するメモリ・プール。 | *action* が CS_SET の場合、**buffer* はバイトのプール。<br>*action* が CS_GET の場合、**buffer* はバイトのプールのアドレスに設定される。 | コンテキスト。 | 非同期アプリケーションで役立つ。<br>コンテキストに接続があるときの設定またはクリアはできない。 |

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
|---|---|---|---|---|
| CS_NETIO | ネットワーク I/O が同期、完全非同期、または遅延非同期のいずれであるか。 | CS_SYNC_IO、CS_ASYNC_IO、または CS_DEFER_IO。 | コンテキスト、接続。 | オープン接続のコンテキストの設定はできない。 |
| CS_NO_TRUNCATE | Client-Library が CS_MAX_MSG より長いメッセージをトランケートするべきか、連続させるべきか。 | 連続させる場合は CS_TRUE。トランケートする場合は CS_FALSE。 | コンテキスト。 | |
| CS_NOAPI_CHK | アプリケーションが Client-Library ルーチンを呼び出すときに、Client-Library が引数とステータスのチェックを実行するかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。これは Client-Library が API チェックを行うことを意味する。 | コンテキスト。 | |
| CS_NOINTERRUPT | 特定のコールバック・イベントによるアプリケーションへの割り込みが可能かどうか。 | CS_TRUE または CS_FALSE。 | コンテキスト。 | 完了イベントにのみ作用する。ノーティフィケーション・イベントには作用しない。 |
| CS_PARTIAL_TEXT | クライアント・アプリケーションで部分更新を実行するかどうかを示す。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | このプロパティは、サーバへの接続が確立される前に設定する必要がある。サーバが部分更新をサポートしていない場合は、このプロパティは CS_FALSE に再設定される。 |
| CS_PROP_ EXTENDEDFAILOVER | サーバが指定したフェールオーバ・ターゲットを有効または無効にする。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_TRUE。 | コンテキスト、接続。 | ログイン・プロパティ。 |
| CS_PROP_ MIGRATABLE | 接続マイグレーションを有効または無効にする。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_TRUE。 | コンテキスト、接続。 | ログイン・プロパティ。 |

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
|---|---|---|---|---|
| CS_PROP_REDIRECT | ログイン・リダイレクト・サポートを有効または無効にする。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_TRUE。 | コンテキスト、接続。 | ログイン・プロパティ。 |
| CS_PROP_SSL_ PROTOVERSION | サポートされている SSL/TLS プロトコルのバージョン。 | CS_INT。 | コンテキスト、接続。 | 次のいずれかの値を指定する。<br>• CS_SSLVER_20<br>• CS_SSLVER_30<br>• CS_SSLVER_ TLS1 |
| CS_PROP_SSL_ CIPHER | CipherSuite 名をカンマで区切ったリスト。 | CS_CHAR。 | コンテキスト、接続。 | |
| CS_PROP_SSL_ LOCALID | ローカル ID ( 証明書 ) ファイルのパスの指定に使用するプロパティ。 | 文字列。 | コンテキスト、接続。 | 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る構造体。 |
| CS_PROP_SSL_CA | 信頼された CA 証明書を含むファイルへのパスを指定する。 | CS_CHAR。 | コンテキスト、接続。 | |
| CS_SEC_ CHANBIND | 接続のセキュリティ・メカニズムがチャンネルのバインドを行うかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_ CONFIDENTIALITY | 接続でデータの暗号化サービスを実行するかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_ CREDENTIALS | 委任されたユーザのクレデンシャルを転送するためにゲートウェイ・アプリケーションで使用される。 | A CS_VOID * ポインタ。 | コンテキスト、接続。 | 読み込み不可能。<br>サポートするネットワーク・セキュリティ・メカニズムが必要である。 |

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
|---|---|---|---|---|
| CS_SEC_ CREDTIMEOUT | ユーザのクレデンシャルが期限切れかどうか。 | CS_INT。有効な値とその意味については、表 2-33 (297 ページ ) を参照。 | コンテキスト、接続。 | サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_ DATAORIGIN | 接続のセキュリティ・メカニズムがデータ・オリジンの検証をするかどうか。 | CS_TRUE または CS_FALSE. デフォルトは CS_FALSE。 | コンテキスト、接続。 | サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_ DELEGATION | ユーザの委任クレデンシャルを使用してサーバを他のサーバに接続させるかどうか。 | CS_TRUE または CS_FALSE。デフォルトは CS_FALSE。 | コンテキスト、接続。 | サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_ DETECTREPLAY | 接続のセキュリティ・メカニズムがリプレイされた転送を検出するかどうか。 | CS_TRUE または CS_FALSE。デフォルトは CS_FALSE。 | コンテキスト、接続。 | サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_ DETECTSEQ | 接続のセキュリティ・メカニズムが不正なシーケンスの転送を検出するかどうか。 | CS_TRUE または CS_FALSE。デフォルトは CS_FALSE。 | コンテキスト、接続。 | サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_ INTEGRITY | 接続のセキュリティ・メカニズムがデータ整合性のチェックを実行するかどうか。 | CS_TRUE または CS_FALSE。デフォルトは CS_FALSE。 | コンテキスト、接続。 | サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_ MECHANISM | 接続のためのセキュリティ・サービスを実行する、ネットワーク・セキュリティ・メカニズムの名前。 | 文字列値 デフォルトはセキュリティ・ドライバ設定によって異なる。 | コンテキスト、接続。 | |
| CS_SEC_ MUTUALAUTH | サーバが、クライアントに対してサーバ自体を認証する必要があるかどうか。 | CS_TRUE または CS_FALSE。デフォルトは CS_FALSE。 | コンテキスト、接続。 | サポートするネットワーク・セキュリティ・メカニズムが必要である。 |

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
|---|---|---|---|---|
| CS_SEC_ NETWORKAUTH | 接続のセキュリティ・メカニズムがネットワークベースのユーザの認証を実行するかどうか。 | CS_TRUE または CS_FALSE。<br>デフォルトは CS_FALSE。 | コンテキスト、接続。 | サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_SEC_ SESSTIMEOUT | 接続のセキュリティ・セッションの期限が切れているかどうか。 | CS_INT。有効な値とその意味については、表 2-33 (297 ページ) を参照。 | コンテキスト、接続。 | サポートするネットワーク・セキュリティ・メカニズムが必要である。 |
| CS_TCP_RCVBUF | クライアント・アプリケーションの TCP ソケット入力バッファのサイズ。 | 正の整数値。 | コンテキスト、接続。 | |
| CS_TCP_SNDBUF | クライアント・アプリケーションの TCP ソケット出力バッファのサイズ。 | 正の整数値。 | コンテキスト、接続。 | |
| CS_TEXTLIMIT | この接続で返される最大の text または image 値。 | 整数値。 | コンテキスト、接続。 | |
| CS_TIMEOUT | タイムアウト値。 | 整数値。 | コンテキスト。 | |
| CS_USER_ALLOC | ユーザ定義のメモリ割り付けルーチン。 | *action* が CS_SET の場合、**buffer* はインストールするユーザ定義関数。<br>*action* が CS_GET の場合、**buffer* は、現在インストールされているユーザ定義関数のアドレスに設定される。 | コンテキスト。 | 非同期アプリケーションで役立つ。 |

| プロパティ | 意味 | *buffer の値 | レベル | 注意 |
|---|---|---|---|---|
| CS_USER_FREE | ユーザ定義のメモリ解放ルーチン。 | *action* が CS_SET の場合、*buffer* はインストールするユーザ定義関数。<br>*action* が CS_GET の場合、*buffer* は、現在インストールされているユーザ定義関数のアドレスに設定される。 | コンテキスト。 | 非同期アプリケーションで役立つ。 |
| CS_VER_STRING | Client-Library の正確なバージョン文字列。 | 文字列。 | コンテキスト。 | 取得のみ可能。 |
| CS_VERSION | このコンテキストで使用している Client-Library のバージョン。 | バージョン・レベル記号。 | コンテキスト。 | 取得のみ可能。 |

**参照**

cs_config、ct_cmd_props、ct_capability、ct_con_props、ct_connect、ct_init、「プロパティ」(208 ページ )

# ct_connect

**説明**

サーバに接続します。

**構文**

```
CS_RETCODE ct_connect(connection, server_name,
          snamelen)

CS_CONNECTION    *connection;
CS_CHAR          *server_name;
CS_INT           snamelen;
```

**パラメータ**

*connection*

CS_CONNECTION 構造体を指すポインタです。CS_CONNECTION 構造体は、特定のクライアント／サーバ接続の情報を含んでいます。

CS_CONNECTION 構造体を割り付けるには ct_con_alloc を、また、この構造体をログイン・パラメータで初期化するには ct_con_props を使用してください。

*server_name*

接続するサーバ名を指すポインタです。**server_name* には、その接続のディレクトリ・ソースに含まれる該当のサーバ・エントリの名前を指定します。ct_connect はその接続のディレクトリ・ソース内で、**server_name* に指定されたサーバ名を検索し、そのサーバへの接続方法を判別します。接続のディレクトリ・ソースは CS_DS_PROVIDER プロパティで指定します。「ディレクトリ・サービス・プロバイダ」(132 ページ) を参照してください。これは、Sybase *interfaces* ファイルまたはネットワークベースのディレクトリ・サービスです。

*server_name* は、接続のディレクトリ・プロバイダによって認識される命名構文を使用してください。ほとんどのネットワークベース・ディレクトリ・プロバイダの場合、ベース・ディレクトリ・パス (DIT ベース ) は CS_DS_DITBASE 接続プロパティで指定できます。*server_name* が部分的に修飾された名前である場合、ディレクトリ・プロバイダはこの名前を DIT ベースと組み合わせて、完全に修飾された名前を作ります。

*server_name* が NULL である場合、ct_connect はサーバ名にはプラットフォーム固有のデフォルトを使用します。環境変数または論理名をサポートするプラットフォームの場合、これは DSQUERY 環境変数または論理名の値です。このようなプラットフォームでは、DSQUERY が設定されていなかった場合、ct_connect は SYBASE という名前のサーバを探します。

*snamelen*

**server_name* のバイト単位の長さです。**server_name* が null で終了する場合、*snamelen* を CS_NULLTERM にして渡してください。*server_name* が NULL の場合、*snamelen* を 0 または CS_UNUSED にして渡してください。

**戻り値**

ct_connect は、次の値を返します。

| 戻り値 | 意味 |
| --- | --- |
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_PENDING | 非同期ネットワーク I/O が有効。「非同期プログラミング」(12 ページ ) を参照。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

ct_connect の失敗の主な原因は、次のとおりです。

- 十分なメモリの割り当てができません。

- 最大数の接続がすでに確立されています。コンテキストごとに許可されている接続の最大数を設定するには、ct_config を使用します。
- ソケットをオープンできません。
- *interfaces* ファイル内にサーバ名がありません。
- ホスト・マシン名がわかりません。
- Adaptive Server Enterprise が使用できないか、存在しません。
- ログインが正しくありません。
- *interfaces* ファイルまたはディレクトリ・サービス・セッションをオープンできません。
- 要求されたディレクトリ・ドライバにロードできません。

ct_connect が CS_FAIL を返すとき、エラーを示す Client-Library エラー番号が生成されます。

---

**注意**　DLL のエントリ関数で ct_connect を呼び出すと、デッドロックが発生することがあります。このシステムでは、オペレーティング・システム・スレッドを作成し、システム・ユーティリティを使用してそのスレッドを同期させようとします。この同期化は、オペレーティング・システムの直列化プロセスと競合します。

---

**例**

```
/* ex_connect() */
 CS_RETCODE CS_PUBLIC
 ex_connect(context, connection, appname,username, password,
       サーバ)
   CS_CONTEXT    *context;
   CS_CONNECTION *connection;
   CS_CHAR       *appname;
   CS_CHAR       *username;
   CS_CHAR       *password;
   CS_CHAR       *server;
   {
       CS_INT        len;
       CS_RETCODE    retcode;

      /* Allocate a connection structure */
       ...CODE DELETED.....

      /* Set properties for new connection */
       ...CODE DELETED.....

      /* Open the connection */
```

```
     if (retcode == CS_SUCCEED)
     {
         len = (server == NULL) ?0 :CS_NULLTERM;
         retcode = ct_connect(*connection, server, len);
         if (retcode != CS_SUCCEED)
         {
             ex_error("ct_connect failed");
         }
     }
    if (retcode != CS_SUCCEED)
     {
         ct_con_drop(*connection);
         *connection = NULL;
     }
    return retcode;
}
```

このコードは、*exutils.c* サンプル・プログラムからの抜粋です。

**使用法**

- 接続についての情報は、その接続をユニークに識別する CS_CONNECTION 構造体に記録されます。接続の確立処理では、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て、接続固有のプロパティ情報をサーバに送信します。
- 接続の作成時にはサーバへのログインが行われるため、アプリケーションで、ct_connect を呼び出す前にログイン・パラメータ ( サーバ・ユーザ名やパスワードなど ) を定義する必要があります。アプリケーションでログイン・パラメータを定義するには、ct_con_props を呼び出します。
- 接続には同期接続と非同期接続があります。接続が同期と非同期のどちらで行われるかは、Client-Library の CS_NETIO プロパティの設定で決まります。

  「非同期プログラミング」(12 ページ ) を参照してください。
- コンテキストごとのオープン接続の最大数は、CS_MAX_CONNECT プロパティ (ct_config で設定 ) によって決まります。明示的に設定されない場合、接続の最大数はプラットフォーム固有の値がデフォルトで設定されます。プラットフォーム固有のプロパティ値については、プラットフォームの『Open Client/Server プログラマーズ・ガイド補足』を参照してください。
- クライアントとサーバ間で接続が試行されたときに、そのプロセスで障害が発生する状況として次の 2 つがあります ( システムは正しく設定されていると仮定します )。

  - サーバが存在するマシンが正しく動作しており、ネットワークも正しく動作している場合。

    この場合、指定されたポートで受信するサーバがないとき、サーバがあると想定されるマシンが、接続を確立できないことをネットワーク・エラーを使用してクライアントに知らせます。ログイン・タイムアウト値にかかわらず、接続は障害を起こします。

  - サーバが存在するマシンがダウンしている場合。

    この場合は、サーバが存在するマシンが応答しません。「無応答」はエラーとはみなされないので、ネットワークはエラーが発生したという通知をクライアントに送信しません。ただし、ログイン・タイムアウト時間が設定されている場合、クライアントがその設定時間内に応答を受信できないときはタイムアウト・エラーが発生します。

    CS_LOGIN_TIMEOUT プロパティでログイン・タイムアウトの終了を指定します。「ログイン・タイムアウト」(253 ページ ) を参照してください。

- 接続をクローズするために、アプリケーションは ct_close を呼び出すことができます。

#### サーバのアドレス情報

- Client-Library は、要求されたサーバ名と関係のあるネットワークのアドレスを含むディレクトリ・ソースを必要とします。ディレクトリ・ソースは、Sybase *interfaces* ファイルまたはネットワークベースのディレクトリ・サービスです。
- ct_connect で使用するディレクトリ・ソースは、CS_DS_PROVIDER 接続プロパティの設定によって異なります。CS_DS_PROVIDER プロパティの説明については、「ディレクトリ・サービス・プロバイダ」(132 ページ ) を参照してください。
- ネットワークベースのディレクトリ・サービスについては、「ディレクトリ・サービス」(115 ページ ) および「サーバ・ディレクトリ・オブジェクト」(319 ページ ) を参照してください。
- サーバ名には複数のアドレスを関連付けることができます。この場合、ct_connect は、サーバからの応答があった最初のアドレスでログイン・ダイアログを開始します。
  - CS_RETRY_COUNT プロパティは、ct_connect が各サーバ・アドレスをリトライする回数を制御します。

- CS_LOOP_DELAY プロパティは、ct_connect がシーケンスをリトライする前に待つ時間を制御します。

これらのプロパティについては、「リトライ回数」(264 ページ ) および「ループ遅延」(253 ページ ) を参照してください。

**接続のデフォルトの外部設定**

- ct_connect は、必要に応じて Open Client/Server ランタイム設定ファイルを読み込んで、接続の接続プロパティ、サーバ・オプション、デバッグ・オプションを設定します。この機能によって、プログラマは ct_con_props、ct_options、ct_debug をハードコードで呼び出すのではなく、設定値を外部化することができます。
- デフォルトでは、ct_connect は設定ファイルを読み込みません。外部設定を使用可能にするには、アプリケーションに CS_EXTERNAL_CONFIG プロパティを設定する必要があります。「ランタイム設定ファイルの使い方」(352 ページ ) を参照してください。

**参照**

ct_close、ct_con_alloc、ct_con_drop、ct_con_props、ct_remote_pwd、「ディレクトリ・サービス」(115 ページ )、「interfaces ファイル」(157 ページ )、「プロパティ」(208 ページ )、「サーバ・ディレクトリ・オブジェクト」(319 ページ )

# ct_cursor

**説明**

Client-Library カーソル・コマンドを開始します。

**構文**

```
CS_RETCODE ct_cursor(cmd, type, name, namelen, text,
          textlen, option)

CS_COMMAND        *cmd;
CS_INT            type;
CS_CHAR           *name;
CS_INT            namelen;
CS_CHAR           *text;
CS_INT            textlen;
CS_INT            option;
```

**パラメータ**

*cmd*

クライアント／サーバ・オペレーションを管理する CS_COMMAND 構造体を指すポインタです。

*type*

開始するコマンドのタイプです。表 3-14 に、*type* の記号値を示します。

*name*

カーソル・コマンドに関連する名前がある場合に、その名前を指すポインタです。表 3-14 ( 474 ページ ) に、名前を必要とするコマンドのタイプを示します。

*namelen*

**name* のバイト単位の長さです。**name* が null で終了している場合、*namelen* を CS_NULLTERM にして渡してください。*name* が NULL の場合、*namelen* を CS_UNUSED にして渡してください。

text

カーソル・コマンドに関連するテキストを指すポインタです。表 3-14 に、テキストを必要とするコマンドとそのテキストの内容を示します。

*textlen*

*text のバイト単位の長さです。*text が null で終了する場合、*textlen* を CS_NULLTERM にして渡してください。text が NULL の場合、*textlen* を CS_UNUSED にして渡してください。

*option*

このコマンドに関連するオプションです。表 3-14 に、オプションをとるコマンドと使用可能なオプションについて示します。

**戻り値**

ct_cursor は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗した。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

**例**

**例 1** 次のコードは、*csr_disp.c* サンプル・プログラムからの抜粋で、通常のカーソルの機能について説明しています。

```
/* DoCursor(connection) */
 CS_STATIC CS_RETCODE
 DoCursor(connection)
 CS_CONNECTION    *connection;
{
     CS_RETCODE        retcode;
     CS_COMMAND        *cmd;
     CS_INT            res_type;
```

```
/* Use the pubs2 database */
 ...CODE DELETED.....

/*
 ** Allocate a command handle to declare the
 ** cursor on.
 */
 retcode = ct_cmd_alloc(connection, &cmd)
 if (retcode != CS_SUCCEED)
 {
     ex_error("DoCursor:ct_cmd_alloc() failed");
     return retcode;
 }

/*
 ** Declare the cursor. SELECT is a select
 ** statement defined in the header file.
 */
 retcode = ct_cursor(cmd, CS_CURSOR_DECLARE,
     "cursor_a", CS_NULLTERM, SELECT, CS_NULLTERM,
     CS_READ_ONLY);
 if (retcode != CS_SUCCEED)
 {
     ex_error("DoCursor:ct_cursor(declare)
         failed");
         return retcode;
 }

/* Set cursor rows to 10*/
 retcode = ct_cursor(cmd, CS_CURSOR_ROWS, NULL,
     CS_UNUSED, NULL, CS_UNUSED, (CS_INT)10);
 if (retcode != CS_SUCCEED)
 {
     ex_error("DoCursor:ct_cursor(currows)
         failed");
     return retcode;
 }

/* Open the cursor */
 retcode = ct_cursor(cmd, CS_CURSOR_OPEN, NULL,
     CS_UNUSED, NULL, CS_UNUSED, CS_UNUSED);
 if (retcode != CS_SUCCEED)
 {
     ex_error("DoCursor:ct_cursor() failed");
     return retcode;
 }

/*
 ** Send (batch) the last 3 cursor commands to
```

```
         ** the server
         */
         retcode = ct_send(cmd)
         if (retcode != CS_SUCCEED)
         {
             ex_error("DoCursor:ct_send() failed");
             return retcode;
         }

        /*
         ** Process the results. Loop while ct_results()
         ** returns CS_SUCCEED, and then check ct_result's
         ** final return code to see if everything went ok.
         */
         ...CODE DELETED.....

        /*
         ** Close and deallocate the cursor. Note that we
         ** don't have to do this, since it is done
         ** automatically when the connection is closed.
         */
         retcode = ct_cursor(cmd, CS_CURSOR_CLOSE, NULL,
             CS_UNUSED, NULL, CS_UNUSED, CS_DEALLOC);
         if (retcode != CS_SUCCEED)
         {
             ex_error("DoCursor:ct_cursor(dealloc)
                 failed");
             return retcode;
         }

        /* Send the cursor command to the server */
         retcode = ct_send(cmd)
         if (retcode != CS_SUCCEED)
         {
             ex_error("DoCursor:ct_send() failed");
             return retcode;
         }

        /*
         ** Check its results. The command won't generate
         ** fetchable results.
         */
         ...CODE DELETED.....

        /* Drop the cursor's command structure */
         ...CODE DELETED.....

        return retcode;
    }
```

**例 2** 次のコードは、*csr_disp_scrollcurs.c* サンプル・プログラムからの抜粋で、スクロール可能カーソルの機能について説明しています。

```
CS_STATIC CS_RETCODE
DoCursor(connection)
CS_CONNECTION*connection;
{
    CS_RETCODE   retcode;
    CS_COMMAND   *cmd;
    CS_INT       res_type;

         if ((retcode = ex_use_db(connection, Ex_dbname)) != CS_SUCCEED)
         {
                 ex_error("DoCursor:ex_use_db(pubs2) failed");
                 return retcode;
         }

         if ((retcode = ct_cmd_alloc(connection, &cmd)) != CS_SUCCEED)
         {
                 ex_error("DoCursor:ct_cmd_alloc() failed");
                 return retcode;
         }

    /*
    ** Declare an insensitive, scrollable cursor. The same result
    ** would be obtained by using CS_SCROLL_INSENSITIVE.
    */
    retcode = ct_cursor(cmd, CS_CURSOR_DECLARE, "cursor_a", CS_NULLTERM,
           SELECT, CS_NULLTERM, CS_SCROLL_CURSOR);

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

    /*
    ** This example relies on CS_CURSOR_ROWS set to 1, e.g. fetch a single
    ** row at any time for the server. No row buffering here.
    */
    retcode = ct_cursor(cmd, CS_CURSOR_ROWS, NULL, CS_UNUSED, NULL,
           CS_UNUSED, (CS_INT)1);
     if (retcode != CS_SUCCEED)
    {
                 ex_error("DoCursor:ct_cursor(currows) failed");
       return retcode;
    }
```

```
    retcode = ct_cursor(cmd, CS_CURSOR_OPEN, NULL, CS_UNUSED, NULL,
           CS_UNUSED, CS_UNUSED);
     if (retcode != CS_SUCCEED)
    {
                ex_error("DoCursor:ct_cursor() failed");
       return retcode;
    }

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

    while((retcode = ct_results(cmd, &res_type)) == CS_SUCCEED)
    {
       switch ((int)res_type)
       {
            case CS_CMD_SUCCEED:
           break;

            case CS_CMD_DONE:
           break;

       case CS_CMD_FAIL:
           ex_error("DoCursor: ct_results() returned CMD_FAIL");
           break;

       case CS_CURSOR_RESULT:
           retcode = ex_scroll_fetch_1(cmd);
            if (retcode != CS_SUCCEED)
           {
              if (retcode == CS_SCROLL_CURSOR_ENDS ||
                 retcode == CS_CURSOR_BEFORE_FIRST ||
                 retcode == CS_CURSOR_AFTER_LAST)
              {
                 retcode = CS_SUCCEED;
              }
           }
           else
           {
              ex_error("DoCursor: ex_scroll_fetch_1() failed on
CS_CURSOR_RESULT ");
              return retcode;
           }
```

```
            break;

        default:
            ex_error("DoCursor: ct_results() returned unexpected result
type");
            return CS_FAIL;
        }
    }

    switch ((int)retcode)
    {
        case CS_SUCCEED:
        case CS_END_RESULTS:
            break;

        case CS_FAIL:
            ex_error("DoCursor: ct_results() failed");
            return retcode;

        default:
            ex_error("DoCursor: ct_results() returned unexpected result
code");
            return retcode;
    }

    /*
    ** cursor close only
    */
    retcode = ct_cursor(cmd, CS_CURSOR_CLOSE, NULL, CS_UNUSED, NULL,
            CS_UNUSED, CS_UNUSED);

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

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

    while((retcode = ct_results(cmd, &res_type)) == CS_SUCCEED)
    {
        switch ((int)res_type)
```

```
        {
             case CS_CMD_SUCCEED:
             case CS_CMD_DONE:
            break;

        case CS_CMD_FAIL:
            ex_error("DoCursor: ct_results() close returned CMD_FAIL");
            break;

        default:
            ex_error("DoCursor: ct_results() close returned unexpected result
type");
            return CS_FAIL;
        }
    }

    if (retcode != CS_END_RESULTS)
    {
        ex_error("DoCursor: close ENDRESULTS ct_results() failed");
        return retcode;
    }

    /*
    ** cursor dealloc only, but this could be combined with the close.
    */
    retcode = ct_cursor(cmd, CS_CURSOR_DEALLOC, NULL, CS_UNUSED, NULL,
            CS_UNUSED, CS_UNUSED);

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

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

    while((retcode = ct_results(cmd, &res_type)) == CS_SUCCEED)
    {
        switch ((int)res_type)
        {
             case CS_CMD_SUCCEED:
             case CS_CMD_DONE:
            break;
```

```
        case CS_CMD_FAIL:
            ex_error("DoCursor: ct_results() returned CMD_FAIL");
            break;
        default:
            ex_error("DoCursor: ct_results() returned unexpected result
type");
            return CS_FAIL;
        }
    }

    if (retcode != CS_END_RESULTS)
    {
        ex_error("DoCursor: cursor_dealloc ENDRESULTS ct_results() failed");
        return retcode;
    }

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

    return retcode;
}
```

使用法

**表 3-14 : ct_cursor パラメータの一覧**

| type の値 | 初期化されるコマンド | name 値 | text 値 | option 値 |
|---|---|---|---|---|
| CS_CURSOR_ DECLARE | カーソル宣言コマンド。 | カーソル名を指すポインタ。 | カーソルの本体である SQL テキストを指すポインタ。 | CS_UNUSED、または 表 3-15 ( 481 ページ ) に示す値のビットごとの論理和。 |

| type の値 | 初期化されるコマンド | name 値 | text 値 | option 値 |
|---|---|---|---|---|
| CS_CURSOR_OPTION | カーソル・オプション設定コマンド。 | NULL | NULL | • カーソルが「更新用」であることを示す場合、CS_FOR_UPDATE。<br>• カーソルが「読み込み専用」であることを示す場合、CS_READ_ONLY。<br>• カーソルが更新可能かどうかをサーバが決定することを示す場合、CS_UNUSED。<br>• スクロール可能カーソルを非反映型と宣言する場合、CS_SCROLL_INSENSITIVE。<br>• 半反映型のスクロール可能カーソルを宣言する場合、CS_SCROLL_SEMISENSITIVE。<br>• 非反映型のスクロール可能カーソル ( デフォルト ) を宣言する場合、CS_SCROLL_CURSOR。<br>• カーソルを非反映型および非スクロール可能と宣言する場合、CS_NOSCROLL_INSENSITIVE。 |
| CS_CURSOR_ROWS | カーソル・ロー設定コマンド。 | NULL | NULL | 1 つのフェッチ要求で返されるロー数を表す整数。<br>後続の ct_cursor 呼び出しで指定されていない場合、デフォルトは 1。<br>ct_scroll_fetch の結果として複数ローを得るには、CS_CURSOR_ROWS 値を 1 より大きくする必要がある。<br>最高のパフォーマンスを得るには、CS_CURSOR_ROWS を ct_bind 呼び出しの count フィールドと同じ値に設定する。<br>「ct_bind」(371 ページ ) を参照。 |

| type の値 | 初期化されるコマンド | name 値 | text 値 | option 値 |
|---|---|---|---|---|
| CS_CURSOR_OPEN | カーソル・オープン・コマンド。 | NULL | NULL | • CS_RESTORE_OPEN は、前に送信したカーソル・オープン・コマンドのパラメータ・バインド情報をリストアする。詳細については、「カーソル・オープン・コマンドのリストア」(488 ページ ) を参照。<br>• カーソルを初めてオープンするときには、CS_UNUSED を使用すること。 |
| CS_CURSOR_UPDATE | カーソル更新コマンド。 | 更新を行うテーブル名を指すポインタ。 | SQL 更新文を指すポインタ。 | • *text が更新文全体である場合は、CS_UNUSED。<br>• *text が更新文の一部である場合は、CS_MORE。<br>• *text が更新文の最後の部分である場合は、CS_END。 |
| CS_CURSOR_DELETE | カーソル削除コマンド。 | 削除を行うテーブル名を指すポインタ。 | NULL | CS_UNUSED |
| CS_CURSOR_CLOSE | カーソル・クローズ・コマンド。 | NULL | NULL | • カーソルを割り付け解除してクローズする場合、CS_DEALLOC。<br>• カーソルを割り付け解除しないでクローズする場合、CS_UNUSED。 |
| CS_CURSOR_DEALLOC | カーソル割り付け解除コマンド。 | NULL | NULL | CS_UNUSED |

- コマンドを開始するのは、コマンドをサーバに送信する最初の手順です。Client-Library カーソル・コマンドとしては、基本テーブル内のローの更新と削除を行うコマンドの他に、カーソルの宣言、オープン、カーソル・ロー設定、クローズ、割り付け解除を行うコマンドがあります。Client-Library カーソルの詳細については、『Open Client Client-Library/C プログラマーズ・ガイド』の「第 7 章 Client-Library カーソルの使い方」を参照してください。
- カーソル・コマンドをサーバに送信するために、アプリケーションでは次のことを行う必要があります。

a　ct_cursor を呼び出して、コマンドを開始します。これで、サーバに送信するコマンド・ストリームを構築するときに使用する内部構造体が設定されます。

b　必要に応じて、コマンドが必要とする各パラメータに対して ct_param または ct_setparam を一度呼び出し、コマンドにパラメータを渡します。

c　カーソル宣言、カーソル・オープン、カーソル更新コマンドには、パラメータが必要な場合があります。その他のカーソル・コマンドには必要ありません。

d　ct_send を呼び出して、コマンドをサーバに送信します。

e　CS_END_RESULTS、CS_CANCELED、または CS_FAIL が返されるまで、ct_results を繰り返し呼び出してコマンドの結果を処理します。カーソル・オープン・コマンドは、CS_CURSOR_RESULT 結果タイプ ( およびステータス情報を示す他の結果タイプ ) を返します。その他のカーソル・コマンドの場合、フェッチ可能な結果を返しませんが、コマンド・ステータスを示し結果タイプだけは返します。結果処理の詳細については、「結果」(280 ページ ) を参照してください。

- Client-Library の場合、アプリケーションは、前の実行の結果を処理し終わった直後に ct_send を呼び出して、コマンドを再送信できます。アプリケーションは、ct_cursor で開始したどのコマンドでも再送信できます。ただし、サーバで正常に再実行されるのは、カーソル更新コマンドとカーソル削除コマンドだけです。その他のカーソル・コマンドは特別のシーケンスで実行する必要があり、再送信すると、サーバ処理エラーが発生することがあります。

### カーソル・コマンドの順序

- サーバでは、カーソル・コマンドを以下に説明する順序で実行する必要があります。以下の各手順は、別の結果を生成する別々のサーバ・コマンドです。

a　カーソルを宣言します。この手順では、カーソルのソース・クエリを指定し、また必要に応じて、そのカーソルの結果セット内の更新可能なカラムを指定します。カーソルを宣言するには ct_cursor または ct_dynamic を使用します。ct_cursor を使用したこの手順の詳細については、「カーソル宣言コマンド」(478 ページ ) を参照してください。ct_dynamic を使用したカーソルの宣言方法については、「準備文でのカーソル宣言」(534 ページ ) を参照してください。

b　カーソル・オプションを指定します (ct_dynamic で宣言したカーソルについてのみ)。詳細については、「動的 SQL のカーソル・オプション」(484 ページ ) を参照してください。

c　カーソル・ロー設定を指定します。詳細については、「カーソル・ロー・コマンド」(484 ページ ) を参照してください。

d　カーソルをオープンします。カーソルを初めてオープンするときに、手順 1 〜 4 のコマンドをバッチ処理にして、サーバとの間のネットワーク往復回数を減らすことができます。詳細については、「カーソル・オープン・コマンド」(486 ページ ) と「カーソル・オープン・コマンドのバッチ」(487 ページ ) を参照してください。

e　ct_results および ct_fetch を使用してカーソル・オープン結果を処理します。スクロール可能カーソルの場合は、ct_results および ct_scroll_fetch を使用します。ct_fetch が CS_SUCCEED または CS_ROW_FAIL を返すたびに、アプリケーションは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同じ CS_COMMAND 構造体で発行できます。また、アプリケーションは、異なる CS_COMMAND 構造体を使用して、カーソルから再度フェッチする前にコマンドの結果を処理する場合にかぎり、新しいコマンド ( カーソルに関係ないコマンド ) も送信できます。結果処理の詳細については、「結果」(280 ページ ) を参照してください。ネストされたカーソル・コマンドの詳細については、「カーソル更新コマンド」(489 ページ ) と「カーソル削除コマンド」(491 ページ ) を参照してください。

f　「カーソル・クローズ・コマンド」(491 ページ ) の説明に従って、カーソルをクローズします。クローズしたカーソルは再オープンできます。手順 3 〜 6 は何回でも繰り返せます。カーソルは、新しいカーソル・オープン・コマンドを開始する方法でも、前に開始したカーソル・オープン・コマンドをリストアする方法でも、再オープンできます。詳細については、「カーソル・オープン・コマンドのリストア」(488 ページ ) を参照してください。

g　カーソルの割り付けを解除します。詳細については、「カーソル割り付け解除コマンド」(492 ページ ) を参照してください。

### カーソル宣言コマンド

- Client-Library カーソルを宣言することは、カーソル名を select 文と対応させることと同じです。この SQL 文をカーソルの「本体」といいます。

- ct_cursor カーソル宣言コマンドには、次の規則が適用されます。
  - 各 CS_COMMAND 構造体には、1 つのカーソルしか宣言できません。ただし、同じ接続を共有する別の CS_COMMAND 構造体に別のカーソルを宣言することはできます。
  - Client-Library カーソルのすべてのオペレーションは、その宣言から割り付け解除まで、カーソルを作成したときに使用したコマンド構造体を参照していなければなりません。
  - ある CS_COMMAND 構造体にカーソルを宣言すると、その構造体は、カーソルの割り付けが解除されるまで、サーバ・コマンド ct_command、ct_dynamic、または ct_sendpassthru の実行には使用できません。
  - 動的 SQL 文に対応するカーソルは、ct_cursor ではなく ct_dynamic で宣言します。
- カーソル本体は、*text パラメータとして直接指定できる他、ストアド・プロシージャのテキストとして間接的にも指定できます。ストアド・プロシージャの場合、*text パラメータは、そのストアド・プロシージャを実行するコマンドでなければなりません。ストアド・プロシージャで宣言されたカーソルを、「**実行カーソル**」といいます。
  - 次の例では、*titles* テーブルのローに対して、*title_cursor* という名前のカーソルを宣言しています。

    ```
    ct_cursor(cmd, CS_CURSOR_DECLARE,
         "title_cursor", CS_NULLTERM,
         "select * from titles", CS_NULLTERM,
         CS_UNUSED);

    ct_send(cmd);
    ```

  - 次の例では、ストアド・プロシージャ title_cursor_proc に対して実行カーソルを宣言しています。

    ```
    ct_cursor(cmd, CS_CURSOR_DECLARE,
         "mycursor", CS_NULLTERM,
         "exec title_cursor_proc", CS_NULLTERM,
         CS_UNUSED);

    ct_send(cmd);
    ```

この例の場合、カーソルの本体は、ストアド・プロシージャを構成しているテキストです。ストアド・プロシージャ・テキストには、1 つの select 文しか含めることはできません。上記の例では、title_cursor_proc が次のように作成されます。

```
create proc title_cursor_proc as
   select * from titles for read only
```

**注意**　実行カーソルで使用されるストアド・プロシージャは、1 つの select 文だけで構成してください。このストアド・プロシージャのリターン・ステータスはクライアント・プログラムには使用できません。出力パラメータ値もクライアント・プログラムには使用できません。

- カーソルに対応する select 文には、ホスト変数を含めることができます。ホスト変数が含まれている場合、カーソルを宣言した後で各変数のフォーマットを記述する必要があります。各ホスト変数のフォーマットを記述するには、まず、変数のフォーマットを記述するための CS_DATAFMT 構造体を初期化し、次に CS_DATAFMT をパラメータとして指定して ct_param を呼び出します。

  カーソル宣言時には、ct_param はホスト言語変数のフォーマット情報を用意するだけです。実際の値は、カーソル・オープン時に、パラメータ値を持つ ct_param またはパラメータ・ソース変数へのポインタを持つ ct_setparam を呼び出すことによって用意されます。

- カーソルに対応する execute 文には、ホスト言語変数を含めることができないため、カーソル宣言時に ct_param で変数のフォーマットを指定する必要はありません。カーソルのオープン時に、ct_param または ct_setparam を使用してプロシージャのパラメータを入力します。実行カーソルの場合、ストアド・プロシージャのパラメータのフォーマットはプロシージャの宣言によって決定されます。

### ct_cursor(CS_CURSOR_DECLARE) のオプション値

- カーソル宣言コマンドを開始するとき、ct_cursor の *option* パラメータに次の値を渡すことができます。

**表 3-15：ct_cursor(CS_CURSOR_DECLARE) のオプション値**

| option の値 | 意味 |
|---|---|
| CS_MORE | *text はカーソル本体の一部分であり、後続の呼び出しで入力される残り部分があることを示す。<br>このビットが設定されている場合、他のすべてのオプションは無視される。このビットが設定されていない場合、*text はカーソル本体全体であると見なす。 |
| CS_END | *text はカーソル本体の最後の部分であることを示す。 |
| CS_FOR_UPDATE | カーソルが「更新用」であることを示す。CS_END と一緒でもこれだけ単独でも使用できる。このオプションが単独で表示される場合、カーソル本体全体を 1 つの呼び出しで指定する。<br>**注意**　Adaptive Server Enterprise 接続の場合、カーソル・ローが更新可能かどうかを指定するには、カーソル本体に for update of 句または for read only 句を使用してください。Adaptive Server Enterprise は *option* 値を認識しません。 |
| CS_READ_ONLY | カーソルが読み込み専用であることを示す。 |
| CS_UNUSED | CS_END ビット ( これのみ ) を設定するのと同じ。 |
| CS_IMPLICIT_CURSOR | これは、TDS ベースのクライアント・カーソルで、最適化によりネットワークの往復回数が減る可能性がある。前回のロー・フェッチ後に挿入された新しいローは、今後のフェッチでは認識されない。 |
| CS_SCROLL_INSENSITIVE | スクロール可能な非反映型のカーソルを宣言する。カーソルがオープンされている場合は、カーソル結果セットは静的で、既知のローの数を示す。ベース・テーブルに加えられた変更は参照できない。 |
| CS_SCROLL_SEMISENSITIVE | スクロール可能な半反映型のカーソルを宣言する。カーソルのオープン時には、カーソル結果セットのローの数は不明。ベース・テーブルのデータが変更されると、カーソル結果セットも変更される。 |
| CS_SCROLL_CURSOR | スクロール可能な非反映型のカーソルにマップする。 |
| CS_NOSCROLL_INSENSITIVE | 前方向のみの非反映型の読み取り専用カーソルを宣言する。このオプションは、ct_fetch でのみ使用でき、ct_scroll_fetch では使用できない。 |
| CS_CUR_RELLOCKS_ONCLOSE | カーソルがクローズしたら、サーバが共有ロックを解除しなければならないことを示す。 |

- カーソルのテキスト値を分割して構築するには、オプション値 CS_MORE および CS_END を使用します。CS_MORE を使用する 1 つ以上の ct_cursor 呼び出しのシーケンスは、CS_END を指定した呼び出しで終わらせてください。次に例を示します。

```
ct_cursor(cmd, CS_CURSOR_DECLARE,
       "select title_id, contract" , ..., CS_MORE);
 ct_cursor(cmd, CS_CURSOR_DECLARE,
       "from titles ", ... , CS_MORE);
 ct_cursor(cmd, CS_CURSOR_DECLARE,
       "where contract=FALSE ", ...., CS_MORE);
 ct_cursor(cmd, CS_CURSOR_DECLARE,
       "for update of contract", ...., CS_END);
```

  最後のカーソル宣言呼び出しに、CS_END を指定してください。CS_READ_ONLY と CS_FOR_UPDATE は CS_MORE と一緒には使用できません。これらのオプション・ビットのどちらかを設定する必要がある場合には、最後の呼び出しで設定してください ( たとえば、CS_END | CS_READONLY を使用してください )。

  Client-Library は *text 値を付加するときにスペースを追加しません。

- CS_FOR_UPDATE オプションと CS_READ_ONLY オプションはサーバに渡されます。どちらのオプションも設定されていない場合には、サーバは、*text に指定されたカーソル本体の内容に基づいてカーソルが更新可能かどうかを判断します。

**更新可能かどうかの指定**

- Client-Library のカーソルを宣言するときには、そのカーソルが「**更新可能**」かどうか、つまり、取得したカーソル・ローを ct_cursor の更新コマンドを使用して更新することがあるかどうか、アプリケーションで指定する必要があります。この指定は、送信先サーバに応じて、カーソルの本体文の内容で行う場合と ct_cursor のオプション・パラメータで行う場合があります。

  サーバが Adaptive Server Enterprise である場合、カーソルに対応する select 文に、テーブル・ローが更新可能であるかどうかを定義します。アプリケーションは、カーソルが更新可能であるかどうかを指定するのに、Transact-SQL の for update of 句または for read only 句を使用します。たとえば、次の呼び出しの文では、price カラムは更新されるが、その他のカラムは更新されないと指定しています。

```
#define TITLE_CUR ¥
 "select title_id, title, price from titles ¥
 for update of price"
```

```
ret = ct_cursor(cmd, CS_CURSOR_DECLARE,
          "titles_cursor", CS_NULLTERM,
          TITLE_CUR, CS_NULLTERM, CS_END);
```

サーバが Open Server アプリケーションである場合、クライアントでの CS_READ_ONLY オプションと CS_FOR_UPDATE オプションの使用を要求するか、select 文を解析します。どちらにするかは、Open Server アプリケーションの設計によります。サーバが *option* を使用してカーソルが更新可能かどうかを決定する場合、ct_cursor の使用法は次のようになります。

- カーソルを「読み込み専用」として宣言するには、アプリケーションは *option* の値として CS_READ_ONLY を指定します。
- カーソルを「更新用」として宣言するには、アプリケーションは *option* の値として CS_FOR_UPDATE を指定します。

  カーソルのカラムの一部のみが「更新用」である場合、アプリケーションは、各更新カラムにつき一度ずつ ct_param を呼び出して、どのカラムが「更新用」であるかを指定します。カーソルのすべてのカラムが「更新用」である場合には、アプリケーションは ct_param を呼び出して更新カラムを指定する必要はありません。

  たとえば、*au_id* カラムと *au_lname* カラムを「更新用」として指定するには、次のようにコーディングします。

```
     ct_cursor(cmd, CS_CURSOR_DECLARE,
 "au_cursor",
          CS_NULLTERM, "select * from authors"
          CS_NULLTERM, CS_FOR_UPDATE);
      format.status = CS_UPDATECOL;
      ct_param(cmd, &format, "au_id",
               CS_NULLTERM, 0);
      format.status = CS_UPDATECOL;
      ct_param(cmd, &format, "au_lname",
               CS_NULLTERM, 0);
      ct_send(cmd);
```

  カーソルによって返されるすべてのカラムを「更新用」として指定するには、次のようにコーディングします。

```
     ct_cursor(cmd, CS_CURSOR_DECLARE,
  "au_cursor",
             CS_NULLTERM, "select * from authors"
              CS_NULLTERM, CS_FOR_UPDATE);
      ct_send(cmd);
```

### 動的 SQL のカーソル・オプション

- 動的 SQL アプリケーションは準備文でカーソルを宣言できます。準備文でカーソルを宣言するには、ct_dynamic(CS_CURSOR_DECLARE) を呼び出します。この時点から、ct_cursor 呼び出しを使用してカーソルを操作します。
- 動的 SQL のカーソル宣言コマンドには、カーソル・オプションを指定する方法は用意されていません。オプションを設定するには、ct_dynamic を呼び出した後、ct_send を呼び出す前に、ct_cursor(CS_CURSOR_OPTION) を呼び出してください。
- サーバが Adaptive Server Enterprise である場合、CS_READ_ONLY と CS_FOR_UPDATE オプションは、基本サーバ・テーブルに影響を与えません。テーブル・ローが更新可能であるかどうかは、カーソルに対応する select 文に定義します。
- サーバが Open Server アプリケーションである場合、CS_READ_ONLY と CS_FOR_UPDATE オプションはサーバに使用できます。

  この場合、カーソルのカラムのすべてではなく一部が「更新用」である場合には、アプリケーションは、各更新カラムにつき一度ずつ ct_param を呼び出して、どのカラムが更新用であるか示します。カーソルのすべてのカラムが「更新用」である場合には、アプリケーションは、ct_param を呼び出して更新カラムを指定する必要はありません。
- カーソル・オプションは、カーソル宣言コマンドを送信する前に指定してください。

### カーソル・ロー・コマンド

- ct_cursor(CS_CURSOR_ROWS) コマンドでは、一度の内部フェッチ要求に対してサーバが Client-Library に返すローの数が指定されます。これは、一度の ct_fetch 呼び出しでアプリケーションに返されるローの数ではありません。一度の ct_fetch 呼び出しでアプリケーションに返されるローの数は、カーソル結果カラムをバインドするときに使用する CS_DATAFMT 構造体の *count* フィールドの値によって決定されます。
- アプリケーションは、カーソルをオープンした後は、カーソル・ローを設定できません。
- デフォルトでは、カーソル・ロー設定値は 1 つのローです。

### 暗黙カーソルの使用

Client-Library では暗黙カーソルを使用できます。ローのフェッチにおいて、暗黙カーソルは読み込み専用カーソルと同じように機能しますが、暗黙カーソルの方がシステム・リソースをより効率的に使用します。

次の例では、読み込み専用カーソルを使用します。

```
ct_cursor(cmd, CS_CURSOR_DECLARE, "cursor_a",
CS_NULLTERM, SELECT, CS_READ_ONLY)

ct_cursor(cmd, CS_CURSOR_ROWS, NULL, CS_UNUSED, NULL,
CS_UNUSED, CS_INT)5)

ct_cursor(cmd, CS_CURSOR_OPEN, NULL, CS_UNUSED, NULL,
CS_UNUSED, CS_UNUSED)
```

次の例では、暗黙カーソルを使用します。

```
ct_cursor(cmd, CS_CURSOR_DECLARE, "cursor_a",
CS_NULLTERM, SELECT, CS_IMPLICIT CURSOR)

ct_cursor(cmd, CS_CURSOR_ROWS, NULL, CS_UNUSED, NULL,
CS_UNUSED, CS_INT)5)

ct_cursor(cmd, CS_CURSOR_OPEN, NULL, CS_UNUSED, NULL,
CS_UNUSED, CS_UNUSED)
```

暗黙カーソルを使用するには、`cs_ctx_alloc(CS_VERSION_`*`xxx`*`, context)` または `ct_init(*context, CS_VERSION_`*`xxx`*`)` を設定する必要があります。その場合、*xxx* は 125 ( バージョン 12.5) 以降です。1 つのローをフェッチする場合は CS_CURSOR_ROWS を最小値の 2 に設定し、複数のローをフェッチする場合は 3 以上の値に設定してください。

---

**警告！** 暗黙カーソルは Client-Library バージョン 12.5 以上のバージョンでのみ使用可能です。Client-Library バージョン 12.5 より前のバージョンで暗黙カーソルを使用すると、自動的に読み込み専用カーソルに変換されます。

---

### カーソル・クローズ時のロックの解放

CS_CUR_RELLOCKS_ONCLOSE オプションを使用して、カーソルがクローズしたら、Adaptive Server Enterprise が共有の読み取り専用ロックを解除するように要求します。読み取り専用カーソルまたはスクロール可能カーソルに使用するには、OR ビット処理演算子、「|」( パイプ ) を使用します。

- CS_CUR_RELLOCKS_ONCLOSE
- CS_CUR_RELLOCKS_ONCLOSE | CS_READ_ONLY

- CS_CUR_RELLOCKS_ONCLOSE | CS_FOR_UPDATE
- CS_CUR_RELLOCKS_ONCLOSE | CS_SCROLL_CURSOR
- CS_CUR_RELLOCKS_ONCLOSE | CS_SCROLL_INSENSITIVE
- CS_CUR_RELLOCKS_ONCLOSE | CS_SCROLL_SEMISENSITIVE
- CS_CUR_RELLOCKS_ONCLOSE | CS_NOSCROLL_INSENSITIVE

*例 1.* クローズ時に共有ロックを解放するカーソルを宣言します。

```
ct_cursor(cmd, CS_CURSOR_DECLARE, cursor_name,
   CS_NULLTERM, select_statement, CS_NULLTERM,
   CS_CUR_RELOCKS_ONCLOSE);
```

*例 2.* クローズ時に共有ロックを解放する insensitive スクロール可能カーソルを宣言します。

```
ct_cursor(cmd, CS_CURSOR_DECLARE, cursor_name,
   CS_NULLTERM, select_statement, CS_NULLTERM,
   CS_CUR_RELOCKS_ONCLOSE | CS_SCROLL_INSENSITIVE);
```

この機能を示す Open Client サンプル・プログラムについては、*csr_disp_scrollcurs3.c* を参照してください。

**カーソル・オープン・コマンド**

- ct_cursor(CS_CURSOR_OPEN) コマンドは、Client-Library のカーソル本体を実行して、CS_CURSOR_RESULT 結果セットを生成します。
  - アプリケーションは、カーソル・ローにアクセスするために、ct_results、ct_bind、ct_fetch を呼び出して、カーソル結果セットを処理します。
  - アプリケーションは、カーソル結果セット内のローをフェッチしながら、同じ CS_COMMAND 構造体を使用して、ネストされたカーソル・コマンド ( カーソル更新、カーソル削除、カーソル・クローズ ) を送信できます。
  - アプリケーションは、カーソル結果セット内のローをフェッチしながら、別の CS_COMMAND 構造体を使用して、カーソル・コマンドではないコマンド ( または別のカーソルを宣言してオープンするコマンド ) をサーバに送信できます。
- カーソルは、ct_cursor(CS_DECLARE) または ct_dynamic(CS_CURSOR_DECLARE) であらかじめ宣言しておかなければオープンできません。クローズしたカーソルは再オープンできます。

カーソルが ct_cursor で宣言される場合、宣言コマンドとオープン・コマンドをバッチ処理することができます。「カーソル・オープン・コマンドのバッチ」(487 ページ ) を参照してください。

- カーソルのオープン時にはパラメータ値が必要となる場合があります。この場合、アプリケーションは ct_cursor を呼び出した後、ct_param または ct_setparam を呼び出してカーソル・オープン・コマンドにパラメータ値を渡すことができます。次のいずれかが当てはまる場合、カーソル・オープン・コマンドにはパラメータが必要です。
    - カーソルの本体が、ホスト変数を含む SQL 文である。
    - 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。
- 動的 SQL 準備文でカーソルをオープンするには、カーソルを動的に宣言するのに使用したのと同じコマンド構造体 (ct_dynamic(CS_CURSOR_DECLARE)) を指定します。
- カーソルを最初にオープンするときには、ct_send の一度の呼び出して、カーソルの宣言、カーソル・ローの設定、カーソルのオープンを実行するすべてのサーバ・コマンドを送信できます。後続のカーソル・オープン・コマンドの場合、アプリケーションは CS_RESTORE_OPEN オプションを使用して、重複する ct_cursor(CS_CURSOR_ROWS) 呼び出しと ct_param 呼び出しを回避できます。これらの機能については、次を参照してください。
    - 「カーソル・オープン・コマンドのバッチ」(487 ページ )、および
    - 「カーソル・オープン・コマンドのリストア」(488 ページ )。
    - カーソル・オープン・コマンドのテキストは、複数の ct_cursor 呼び出しに分割して組み立てることができます。オープン文を分割して指定するには、*option* パラメータに CS_MORE と CS_END を使用します。

**カーソル・オープン・コマンドのバッチ**

- カーソルをオープンするとき、アプリケーションは ct_cursor コマンドをバッチにして、ネットワーク・トラフィックを減らし、アプリケーション・パフォーマンスを高めることができます。カーソルを宣言してオープンするのに必要なすべてのコマンドを、一度の ct_send 呼び出しで送信できます。

アプリケーションは、Client-Library カーソルを宣言し、ローを設定して、カーソルをオープンするコマンドを次の手順に従ってバッチ処理します。

a ct_cursor を呼び出して、カーソルを宣言します。

b 必要であれば、ct_param または ct_setparam を呼び出して、ホスト変数のフォーマットを定義します。

c 必要に応じて、ct_cursor を呼び出して、カーソルのローを設定します。

d ct_cursor を呼び出して、カーソルをオープンします。

e 必要であれば、ct_param または ct_setparam を呼び出して、ホスト変数の値を入力します。CS_RESTORE_OPEN オプションを使用してカーソルを再オープンする場合には、アプリケーションは ct_setparam を使用する必要があります。ct_setparam は、プログラム変数を入力パラメータにバインドし、アプリケーションがコマンドを再送信するときにパラメータ値を変更できるようにします。アプリケーションが ct_param を使用した場合、カーソル・オープン・コマンドをリストアするとき、パラメータ値を変更することはできません。

f ct_send を呼び出して、コマンド・バッチをサーバに送信します。

呼び出しのシーケンスは次のとおりです。

```
ct_cursor(CS_CURSOR_DECLARE)
ct_param or ct_setparam for each parameter
ct_cursor(CS_CURSOR_ROWS)
ct_cursor(CS_CURSOR_OPEN)
ct_param or ct_setparam for each parameter
ct_send
ct_results
```

バッチにされた各コマンドは別々の結果を生成するので、ct_results に対して複数の呼び出しが必要です。

**カーソル・オープン・コマンドのリストア**

- カーソルを再オープンするとき、アプリケーションは CS_RESTORE_OPEN オプションを使用して、最後に送信したカーソル・オープン・コマンドをリストアできます。

- アプリケーションが、元のカーソル・オープン・コマンドの値の入力に ct_param を使用した場合は、リストアされたカーソル・オープン・コマンドでも同じパラメータ値を使用します。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。
- アプリケーションがカーソル・ロー・コマンドを、前のカーソル・オープン・コマンドとバッチにした場合、Client-Library はカーソル・オープン・コマンドと一緒にカーソル・ロー・コマンドも再送信します。カーソルは同じカーソル・ロー設定で再オープンされます。
- カーソル・オープン・コマンドをリストアする呼び出しのシーケンスは、次のとおりです。

```
/*
 ** Assign new variables in the program variables
 ** bound with ct_setparam.
 */
  ... assignment statement for each parameter
      source value ...
ct_cursor(CS_CURSOR_OPEN, CS_RESTORE_OPEN)
ct_send
... handle cursor results ...
```

- アプリケーションは、割り付けを解除されたカーソルをリストアすることはできません。
- アプリケーションは、CS_HAVE_CUROPEN プロパティをチェックして、コマンド構造体にリストア可能なカーソル・オープン・コマンドがあるかどうかを調べることができます。詳細については、「リストア可能なカーソル・オープン・コマンド」(249 ページ ) を参照してください。
- 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は、CS_STICKY_BINDS コマンド・プロパティを設定すると、パフォーマンスが高くなります。CS_TRUE の場合、このプロパティによって、アプリケーションは元のカーソル結果バインドを再使用して、重複する ct_bind 呼び出しを回避することができます。このプロパティの詳細については、「継続結果バインド」(261 ページ ) を参照してください。

#### カーソル更新コマンド

- ct_cursor(CS_CURSOR_UPDATE) コマンドは、現在のカーソル・ローの新しいカラム値を定義します。定義された新しい値は、基本テーブルの更新に使用されます。

- カーソル更新コマンドは、常に「ネスト」されます。つまり、このコマンドは、カーソルのローが ct_fetch によって処理されている間、ct_results ループ内から送信されます。

  ネストされたカーソル・コマンドは、ct_results が CS_CURSOR_RESULT という *result_type* 値を返した後、送信できます。カーソル更新コマンドが許可される前に、少なくとも 1 つのローがフェッチされていなければなりません。ct_fetch が CS_END_DATA を返した後は、カーソル更新コマンドは許可されません。

- デフォルトでは、最後にフェッチされたローが更新されます。アプリケーションは更新をカーソル結果セット内の別のローにリダイレクトできます。更新をリダイレクトするには、カーソル更新コマンドを送信する前に、ct_keydata で異なるキー値を指定します。

- Adaptive Server Enterprise テーブルを更新する場合、アプリケーションは更新するテーブルの名前を 2 回指定する必要があります。1 回目は、ct_cursor の **name* パラメータの値として指定し、2 回目は、更新文自体 (update *tablename*) に指定します。

- カーソル・オープン・コマンドとカーソル更新コマンドのテキストは、複数の ct_cursor 呼び出しに分割して組み立てることができます。更新文を分割して指定するには、*option* パラメータに CS_MORE と CS_END を使用します。CS_MORE は、アプリケーションが更新文にテキストを追加することを示します。分割して指定するには、1 つの更新文で更新するのはテーブル 1 つだけにする必要があります。

- 更新文のテキストには、ホスト言語変数を含めることができます。この変数が含まれる場合、アプリケーションは、ct_send を呼び出す前に、ct_param または ct_setparam で変数の値を指定する必要があります。カーソル更新コマンドにパラメータが必要で、このコマンドが複数回サーバに送信される場合は、ct_setparam を使用してください。

- カーソル更新コマンドは、他のコマンドと同じように結果を生成します。アプリケーションは、カーソルから再フェッチする前に、結果を処理しなければなりません。

- カーソル更新コマンドは、前の実行結果の処理を終了した直後に ct_send を呼び出すことによって、再送信できます。カーソル更新コマンドは、次の条件がすべて満たされる場合にかぎり、再送信できます。

- アプリケーションが新しくネストされたカーソル・コマンドを開始していない。
- カーソルがまだオープンしている。
- ct_fetch がまだ CS_END_DATA を返していない。

**カーソル削除コマンド**

- ct_cursor(CS_CURSOR_DELETE) コマンドは、カーソル結果セットからローを削除します。この削除は、基本サーバ・テーブルまで戻って伝達されます。
- カーソル削除コマンドは、常に「ネスト」されます。つまり、このコマンドは、カーソルのローが ct_fetch によって処理されている間、ct_results ループ内から送信されます。

  ネストされたカーソル・コマンドは、ct_results が CS_CURSOR_RESULT という *result_type* 値を返した後、送信できます。カーソル削除コマンドが許可される前に、少なくとも 1 つのローがフェッチされていなければなりません。ct_fetch が CS_END_DATA を返した後は、カーソル削除コマンドは許可されません。
- デフォルトでは、最後にフェッチされたローが削除されます。アプリケーションは削除をカーソル結果セット内の別のローにリダイレクトできます。削除をリダイレクトするには、カーソル削除コマンドを送信する前に、ct_keydata で異なるキー値を指定します。
- カーソル削除コマンドは他のコマンドと同じように結果を生成します。アプリケーションは、カーソルから再フェッチする前に、結果を処理しなければなりません。
- カーソル削除コマンドは、カーソル更新コマンドの場合と同じ条件で再送信できます。

**カーソル・クローズ・コマンド**

- ct_cursor(CS_CURSOR_CLOSE) コマンドは、カーソルをオープンしたときに生成されたカーソル結果セットを放棄します。カーソルのすべてのローをフェッチし終わったら、カーソル・クローズ・コマンドを発行します。このコマンドを発行してからでないと、カーソルを再オープンできません。
- アプリケーションはクローズしたカーソルを再オープンできます。
- カーソル・クローズ・コマンドは「ネスト」できます。つまり、カーソル・クローズ・コマンドは、カーソルのローが ct_fetch によって処理されている間、ct_results ループ内から送信できます。

- ネストされたカーソル・クローズ・コマンドは、ct_results が CS_CURSOR_RESULT という *result_type* 値を返した後、ct_fetch が CS_END_DATA を返す前に、送信できます。
- ct_fetch が CS_END_DATA を返した後は、カーソル・クローズ・コマンドはもうネストできません。また、ct_results が CS_END_RESULTS または CS_CANCELED を返すまで、送信できません。

ネストされたカーソル・クローズ・コマンドは、カーソル・オープン・コマンドから返されたローを放棄する手段として ct_cancel より優れています。これは、ct_cancel を使用すると、接続のカーソルが定義されていないステータスになることがあるためです。

- ネストされていないカーソル・クローズ・コマンドは、CS_COMMAND 構造体がアイドル状態のときに、つまり、ct_results が CS_END_RESULTS または CS_CANCELED を返した後で、送信する必要があります。

**カーソル割り付け解除コマンド**

- ct_cursor(CS_CURSOR_DEALLOC) コマンドは、Client-Library カーソルの割り付けを解除します。カーソルの割り付けが解除されると、カーソルは再オープンできません。
- アプリケーションはオープンしているカーソルの割り付けを解除することはできません。
- Client-Library カーソルのクローズと割り付け解除の両方を行うコマンドを開始するには、*type* に CS_CURSOR_CLOSE、*option* に CS_DEALLOC を指定して ct_cursor を呼び出します。

**参照**

「コマンド」(109 ページ )、ct_cmd_alloc、ct_keydata、ct_param、ct_results、ct_send、ct_setparam、ct_scroll_fetch。

# ct_data_info

**説明**

データ I/O 記述子構造体を定義または取得します。

**構文**

```
CS_RETCODE ct_data_info(cmd, action, colnum, iodesc)

CS_COMMAND   *cmd;
CS_INT       action;
CS_INT       colnum;
CS_IODESC    *iodesc;
```

**パラメータ**

*cmd*

クライアント／サーバ・オペレーションを管理する CS_COMMAND 構造体を指すポインタです。

*action*

次の記号値のいずれかです。

| 値 | 意味 |
|---|---|
| CS_SET | I/O 記述子を定義する。 |
| CS_GET | I/O 記述子を取得する。 |

*colnum*

I/O 記述子を取得する text または image カラムの番号です。

*action* が CS_SET の場合、*colnum* を CS_UNUSED にして渡してください。

*action* が CS_GET の場合、*colnum* は text または image カラムの select リスト ID を参照します。最初のカラムがカラム番号 1、次がカラム番号 2、以下同様に続きます。アプリケーションは、text または image カラムを更新する前に、そのカラムを選択してください。

*colnum* に指定するカラムは、text カラムまたは image カラムでなければなりません。

*iodesc*

CS_IODESC 構造体を指すポインタです。CS_IODESC 構造体には、text または image データを記述する情報が含まれます。「CS_IODESC 構造体」(99 ページ ) を参照してください。

**戻り値**

ct_data_info は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

**例**

```
/*
 ** FetchResults()
 **
 ** The result set contains four columns:integer, text,
 ** float, and integer.
 */

CS_STATIC CS_RETCODE
```

```
FetchResults(cmd, textdata)
CS_COMMAND           *cmd;
TEXT_DATA            *textdata;
{
CS_RETCOD    retcode;
    CS_DATAFMT    fmt;
    CS_INT        firstcol;
    CS_TEXT       *txtptr;
    CS_FLOAT      floatitem;
    CS_INT        count;
    CS_INT        len;

   /*
    ** Before we call ct_get_data(), we can only bind
    ** columns that come before the column on which we
    ** perform the ct_get_data().
    ** To demonstrate this, bind the first column
    ** returned.
    */
    ...CODE DELETED.....

   /* Retrieve and display the result */
    while(((retcode = ct_fetch(cmd, CS_UNUSED, CS_UNUSED,
        CS_UNUSED,&count)) == CS_SUCCEED) ||
        (retcode == CS_ROW_FAIL) )
    {
        /* Check for a recoverable error */
        ...CODE DELETED.....

       /* Get the text data item in the 2nd column */
        ...CODE DELETED.....

    /*
     ** Retrieve the descriptor of the text data. It
     ** is available while retrieving results of a select
     ** query. The information will be needed for later
     ** updates.
     */
        retcode = ct_data_info(cmd, CS_GET,  2,
            &textdata->iodesc);
        if (retcode != CS_SUCCEED)
        {
            ex_error("FetchResults:cs_data_info()
                failed");
            return retcode;
        }

       /* Get the float data item in the 3rd column */
        ...CODE DELETED.....
```

```
        /* Last column not retrieved */
    }

  /*
   ** We're done processing rows. Check the final return
   ** value of ct_fetch().
   */
   ...CODE DELETED.....

  return retcode;
}
```

このコードは、*getsend.c* サンプル・プログラムからの抜粋です。

**使用法**

- ct_data_info は、text または image カラムのための CS_IODESC (「**I/O 記述子構造体**」ともいう ) を定義または取得します。
- アプリケーションは、後で更新を予定する text または image カラム値を取得するために ct_get_data を呼び出した後、I/O 記述子を取得するために ct_data_info を呼び出します。この I/O 記述子には、サーバが text または image カラムの更新の管理に使用するテキスト・ポインタとテキスト・タイムスタンプが含まれます。

  I/O 記述子を取得した後、一般のアプリケーションはその I/O 記述子を更新オペレーションで使用する前に、*locale*、*total_txtlen*、および *log_on_update* フィールドの値だけを変更します。

  - CS_IODESC の *total_txtlen* フィールドは、新しい text または image 値の全長をバイト単位で表します。
  - CS_IODESC の *6log_on_update* フィールドは、サーバが更新のログを記録するかどうかを示します。
  - CS_IODESC の *locale* フィールドは、その値のローカライゼーション情報を持つ CS_LOCALE 構造体を指します。
- アプリケーションは、ct_data_info を呼び出してひとまとまりのデータやイメージ・データをサーバに送信する前に、ct_send_data を呼び出して I/O 記述子を定義します。これらの呼び出しは両方とも、text または image 更新オペレーション中に行われます。
- text または image 値の更新が正常に終了すると、その text または image 値に新しいテキスト・タイムスタンプを持つパラメータ結果セットが生成されます。アプリケーションが、この text または image 値の更新を再度予定する場合、ct_data_info を呼び出して更新オペレーションの CS_IODESC を定義する前に、この新しいテキスト・タイムスタンプを保存して、その値の CS_IODESC にコピーしてください。

- ほとんどの場合、アプリケーションは、ct_data_info を呼び出す前に、検索するカラムのために ct_get_data を呼び出す必要があります。ただし、ゲートウェイ Open Server アプリケーションで ct_get_data を Open Server srv_send_data ルーチンと共に使用して text、image、XML カラムをチャンクで転送する場合、アプリケーションでは ct_get_data を呼び出す前に ct_data_info を呼び出す必要があります。これにより Open Server では、カラムが読み込まれる前にオブジェクト名などの固定 I/O フィールドを取得し、ロー全体が読み込まれる前にローのデータを送信できます。text データや、text データの長さを指すポインタなど、I/O 記述子内で変更可能なフィールドは、カラムを読み込んだ後にのみ取得できます。

  この ct_get_data 呼び出しではデータを実際に取得する必要はありません。アプリケーションはバッファ長を 0 にして ct_get_data を呼び出し、その記述子を取得するために ct_data_info を呼び出すことができます。この手法は、アプリケーションが text 値や image 値を取得する前にその長さを確認する必要があるときに役立ちます。

  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。

- 「CS_IODESC 構造体」(99 ページ ) を参照してください。

**参照**

ct_get_data、ct_send_data、「text および image データの処理」(328 ページ )

# ct_debug

**説明**

デバッグ用のライブラリ・オペレーションを管理します。

**構文**

```
CS_RETCODE ct_debug(context, connection, operation,
             flag, filename, fnamelen)

CS_CONTEXT        *context;
CS_CONNECTION     *connection;
CS_INT            operation;
CS_INT            flag;
CS_CHAR           *tablename;
CS_INT            fnamelen;
```

**パラメータ**

*context*

CS_CONTEXT 構造体を指すポインタです。CS_CONTEXT 構造体は Client-Library アプリケーション・コンテキストを定義します。

*operation* が CS_SET_DBG_FILE のときには、*context* を指定して、*connection* は NULL にしてください。

フラグの設定またはクリア時に *context* を指定するかどうかについては、表 3-16 を参照してください。

*connection*

CS_CONNECTION 構造体を指すポインタです。*connection* は有効な CS_CONNECTION 構造体を指していなければなりませんが、デバッグ・オペレーションを使用可能にするために、サーバへの実際の接続は必要ありません。

*operation* が CS_SET_PROTOCOL_FILE のときには、*connection* を指定して、*context* は NULL にしてください。

フラグの設定またはクリア時に *connection* を指定するかどうかについては、表 3-16 ( 498 ページ ) を参照してください。

*operation*

実行するオペレーションです。表 3-17 ( 500 ページ ) に *operation* の記号値を示します。

*flag*

デバッグ・サブシステムを表すビットマスクです。次の表に、*flag* を構成できる記号値の一覧を示します。

**表 3-16：ct_debug flag パラメータの値**

| 値 | 必須 | Client-Library の動作の結果 |
|---|---|---|
| CS_DBG_ALL | *context* および *connection* | 可能なすべてのデバッグ・アクションを行う。 |
| CS_DBG_API_ LOGCALL | *context* | アプリケーションが Client-Library ルーチンを呼び出すごとに情報を出力する。ルーチン名とパラメータ値を含む。 |
| CS_DBG_API_STATES | *context* | Client-Library 関数レベルの状態遷移についての情報を出力する。 |
| CS_DBG_ASYNC | *context* | 非同期関数の開始または完了の各時点で関数トレース情報を出力する。 |
| CS_DBG_DIAG | *connection* | Client-Library メッセージまたはサーバ・メッセージが生成されたとき、そのメッセージ・テキストを出力する。 |
| CS_DBG_ERROR | *context* | Client-Library エラーが発生したとき、トレース情報を出力する。これにより、プログラマはエラーが発生している場所を正確に判断することができる。 |
| CS_DBG_MEM | *context* | メモリ管理についての情報を出力する。 |
| CS_DBG_NETWORK | *context* | Client-Library のネットワークの対話についての情報を出力する。 |
| CS_DBG_PROTOCOL | *connection* | この ct_debug パラメータは、*devlib* ライブラリを使用せずに設定できる。このパラメータは、サーバと交換した情報をプロトコル固有 ( たとえば、TDS) フォーマットで取得する。この情報はそのままでは判読可能ではない。 |
| CS_DBG_PROTOCOL_ FILE | *connection* | この ct_debug パラメータは、*devlib* ライブラリを使用せずに設定できる。接続に対してパラメータが設定されていない場合、mktemp が呼び出され、プロトコル・パケットのダンプ先となるユニークなファイル名が生成される。mktemp に渡されるプレフィックス文字列は、*capture* である。結果として生成されるプロトコル・ファイルは、Ribo で復号化できる。 |

| 値 | 必須 | Client-Library の動作の結果 |
|---|---|---|
| CS_DBG_PROTOCOL_STATES | *connection* | Client-Library プロトコル・レベルの状態遷移についての情報を出力する。 |
| CS_DBG_SSL | *connection* | CT-Library アプリケーションで SSL 関連の診断とエラー・コードを標準出力 (stdout) またはログ・ファイルに書き込むことを可能にする。このフラグは isql で使用できる。CS_DBG_SSL は通常の ( デバッグ以外の ) ライブラリでも使用できる。 |

*filename*

生成されたデバッグ情報を ct_debug が書き込むファイルのフル・パスおよび名前です。

*fnamelen*

*filename* のバイト単位での長さです。*filename* が null で終了する文字列である場合は CS_NULLTERM です。

**戻り値**

ct_debug は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

**例**

```
...CODE DELETED.....
 #ifdef EX_API_DEBUG
     /*
     ** Enable this function right before any call to
     ** Client-Library that is returning failure.
     */
     retcode = ct_debug(*context, NULL, CS_SET_FLAG,
         CS_DBG_API_STATES, NULL, CS_UNUSED);
     if (retcode != CS_SUCCEED)
     {
         ex_error("ex_init:ct_debug() failed");
     }
 #endif
 ...CODE DELETED.....
```

このコードは、*exutils.c* サンプル・プログラムからの抜粋です。

使用法

表 3-17：ct_debug パラメータの一覧

| operation の値 | フラグ | ファイル名 | 結果 |
|---|---|---|---|
| CS_SET_FLAG | 指定する | NULL | フラグで指定されたサブシステムを使用可能にする。 |
| CS_CLEAR_FLAG | 指定する | NULL | フラグで指定されたサブシステムを使用不可能にする。 |
| CS_SET_DBG_FILE | CS_UNUSED | 指定する | 文字フォーマットのデバッグ情報を書き込むファイル名を記録する。 |
| CS_SET_PROTOCOL_FILE | CS_UNUSED | 指定する | プロトコル・フォーマットのデバッグ情報を書き込むファイル名を記録する。 |

- ct_debug は、アプリケーションが特定の診断サブシステムを有効化／無効化して、その結果トレース情報をファイルに出力できるように、デバッグ・ライブラリ・オペレーションを管理します。
- ct_debug 機能は、Client-Library のデバッグ・バージョンの中からだけ利用できます。標準 Client-Library の中から呼び出されると、CS_FAIL を返します。
- デバッグ・フラグには、接続レベルだけで有効化できるものと、コンテキスト・レベルだけで有効化できるものがあります。表 3-16 ( 498 ページ ) に、各フラグを使用可能にできるレベルを示します。
- アプリケーションが ct_debug を呼び出してデバッグ・ファイルを指定しないと、ct_debug は文字フォーマットのデバッグ情報を *stdout* に書き込み ( 標準出力が利用可能な場合 )、プロトコル・フォーマットのデバッグ情報を次のファイルに書き込みます。
  - Windows の場合：
    *capXXXX.tmp*
    ここで、*XXXX* はユニークなコードです。
  - UNIX の場合：
    *captureXXXXXX*
    ここで、*XXXXXX* はユニークなコードです。

  これらのファイルは、アプリケーションの作業ディレクトリにあります。
- Client-Library のデバッグ・バージョンがアプリケーションにリンクされたとき、次の動作が自動的に行われます。
  - メモリ参照チェック：Client-Library は、内部メモリ参照とアプリケーション固有メモリ参照がどちらもすべて有効であることを検証します。

    - データ構造体検証：Client-Library 関数がデータ構造体にアクセスするたびに、Client-Library は最初にその構造体を検証します。
    - 特別な妥当性チェック：Client-Library は、文字列を含むすべての配列参照が範囲内にあることをチェックします。

- Client-Library のデバッグ・バージョンが広範囲の内部チェックを実行するので、デバッグ・ライブラリの使用中は、アプリケーションのパフォーマンスが低下します。パフォーマンスの低下のレベルは、使用可能になっているトレース・サブシステムのタイプと数によって異なります。パフォーマンスの低下を最小にするために、アプリケーション・プログラマは、負荷のかかるトレースを問題のあるコード領域に限定して、トレース・サブシステムを選択的に使用可能にできます。
- デバッグ・ライブラリを使用すると、タイミングの問題が発生している非同期アプリケーションの動作が変わってしまいます。このような場合には、外部トレース・ツール ( たとえば、ネットワーク・プロトコル・アナライザ ) の使用をおすすめします。

**参照**　「エラー処理」(136 ページ )、「デバッグの有効化」(114 ページ )

# ct_describe

**説明**　結果データの記述を返します。

**構文**

```
CS_RETCODE ct_describe(cmd, item, datafmt)

CS_COMMAND   *cmd;
CS_INT       item;
CS_DATAFMT   *datafmt;
```

**パラメータ**

*cmd*

クライアント／サーバ・オペレーションを管理する CS_COMMAND 構造体を指すポインタです。

*item*

対象となる結果項目を表す整数です。

カラムの記述を取得する場合は、対象のカラムのカラム番号を *item* に指定します。select リストの最初のカラムがカラム番号 1、次が番号 2、以下同様に続きます。

計算カラムの記述を取得する場合には、その計算カラムのカラム番号を *item* に指定します。計算カラムは compute 句に指定した順序で返されます。最初に返されるカラムの番号は 1 です。

*リターン・パラメータの記述を取得する場合には*、*そのパラメータのパラメータ番号を item* に指定します。ストアド・プロシージャによって返される最初のパラメータは、番号 1 のパラメータです。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 文に最初に指定されたときの順序で返されます。この順序は、そのストアド・プロシージャを呼び出す RPC 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際には、リターン・パラメータ以外はカウントしないでください。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。

ストアド・プロシージャのリターン・ステータスの記述を取得する場合は、*item* に 1 と指定します。これは、リターン・ステータスの結果セットにはステータスが 1 つしかないためです。

フォーマット情報を取得する場合には、カラムまたは計算カラムの番号を *item* に指定します。

---

**注意**　ct_results がタイプ CS_MSG_RESULT の結果セットをした後、アプリケーションは ct_describe を呼び出すことはできません。これは、CS_MSG_RESULT の結果タイプが関連するデータ項目を持たないからです。メッセージに関連するパラメータは、CS_PARAM_RESULT 結果セットとして返されます。同様に、ct_results がコマンド・ステータス情報を示すために、その **result_type* パラメータを CS_CMD_DONE、CS_CMD_SUCCEED、または CS_CMD_FAIL に設定した後、アプリケーションは ct_describe を呼び出すことはできません。

---

*datafmt*

CS_DATAFMT 構造体を指すポインタです。ct_describe は、*item* によって参照される結果データ項目の記述で **datafmt* を満たします。

ct_describe は CS_DATAFMT の次のフィールドにデータを取り込みます。

表 3-18：CS_DATAFMT 構造体のフィールド (ct_describe)

| フィールド名 | 対象となる結果項目のタイプ | ct_describe が設定するフィールドの内容 |
|---|---|---|
| *name* | 通常カラム、カラム・フォーマット、およびリターン・パラメータ。 | 指定されている場合、NULL で終了するデータ項目の名前。*namelen* が 0 の場合は NULL。 |
| *namelen* | 通常カラム、カラム・フォーマット、およびリターン・パラメータ。 | NULL ターミネータを含まない名前の実際の長さ。<br>*name* が NULL の場合は 0。 |
| *datatype* | 通常カラム、カラム・フォーマット、リターン・パラメータ、リターン・ステータス、計算カラム、および計算カラム・フォーマット。 | 型定数 (CS_xxx_TYPE) は、項目のデータ型を表す。<br>「データ型のサポート」(339 ページ ) に示すすべての型定数 (CS_VARCHAR_TYPE と CS_VARBINARY_TYPE を除く ) が有効。<br>リターン・ステータスは CS_INT_TYPE のデータ型を持つ。<br>計算カラムのデータ型は、基本カラムのデータ型とそのカラムを作成した集合演算子に依存する。 |
| *format* | 未使用。 | |
| *maxlength* | 通常カラム、カラム・フォーマット、およびリターン・パラメータ。 | カラムまたはパラメータのデータの可能最大長 ( バイト単位 )。 |
| *scale* | タイプが数値または 10 進数の通常カラム、カラム・フォーマット、リターン・パラメータ、計算カラム、または計算カラム・フォーマット。 | 結果データ項目の小数点以下の最大桁数。 |
| *precision* | タイプが数値または 10 進数の通常カラム、カラム・フォーマット、リターン・パラメータ、計算カラム、または計算カラム・フォーマット。 | 結果データ項目に表示できる最大 10 進桁数。 |

| フィールド名 | 対象となる結果項目のタイプ | ct_describe が設定するフィールドの内容 |
|---|---|---|
| *status* | 通常カラムとカラム・フォーマット。 | 次の記号のビットマスク。<br>• そのカラムが NULL 値を持てることを示す CS_CANBENULL。<br>• そのカラムが公開されている「隠しカラム」であることを示す CS_HIDDEN。隠しカラムについては、「隠しキー」(250 ページ ) を参照。<br>• カラムが identity カラムであることを示す CS_IDENTITY。<br>• カラムがテーブルのキーの一部であることを示す CS_KEY。<br>• そのカラムがローのバージョン・キーの一部であることを示す CS_VERSION_KEY。<br>• そのカラムがタイムスタンプ・カラムであることを示す CS_TIMESTAMP。<br>• そのカラムが更新可能カーソル・カラムであることを示す CS_UPDATABLE。<br>• そのカラムがカーソル宣言コマンドの更新句にあることを示す CS_UPDATECOL。<br>• そのカラムが RPC コマンドのリターン・パラメータであることを示す CS_RETURN。 |
| *count* | 通常カラム、カラム・フォーマット、リターン・パラメータ、リターン・ステータス、計算カラム、および計算カラム・フォーマット。 | *count* は、ct_fetch の呼び出しごとにプログラム変数にコピーするロー数を表す。アプリケーションが ct_describe のリターン CS_DATAFMT を ct_bind の入力 CS_DATAFMT として使用する場合、ct_describe はデフォルト値を提供するために *count* を 1 に設定する。 |
| *usertype* | 通常カラム、カラム・フォーマット、およびリターン・パラメータ。 | 指定されている場合には、カラムまたはパラメータの Adaptive Server Enterprise ユーザ定義データ型。*usertype* は、*datatype* に加えて ( 代わりではなく ) 設定される。 |

| フィールド名 | 対象となる結果項目のタイプ | ct_describe が設定するフィールドの内容 |
| --- | --- | --- |
| *locale* | 通常カラム、カラム・フォーマット、リターン・パラメータ、リターン・ステータス、計算カラム、および計算カラム・フォーマット。 | データのロケール情報を持つ CS_LOCALE 構造体を指すポインタ。<br>このポインタは NULL でもかまわない。 |

- ct_describe は呼び出されると、記述されているカラムまたはパラメータに関する情報を **datafmt* に取り込みます。**datafmt* の *status* フィールドは、次の値のビットマスクです。
  - そのカラムが NULL 値を持てることを示す CS_CANBENULL。
  - そのカラムが公開されている「隠しカラム」であることを示す CS_HIDDEN。
  - カラムが identity カラムであることを示す CS_IDENTITY。
  - カラムがテーブルのキーの一部であることを示す CS_KEY。
  - カラムがローのバージョン・キーの一部であることを示す CS_VERSION_KEY。
  - カラムがタイムスタンプ・カラムであることを示す CS_TIMESTAMP。
  - そのカラムが更新可能カーソル・カラムであることを示す CS_UPDATABLE。
  - カラムがカーソル宣言コマンドの更新句にあることを示す CS_UPDATECOL。
  - そのカラムが RPC コマンドのリターン・パラメータであることを示す CS_RETURN。

## 戻り値

ct_describe は、次の値を返します。

| 戻り値 | 意味 |
| --- | --- |
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

*item* が有効な結果データ項目を示さない場合、ct_describe は CS_FAIL を返します。

### 例

```
/* ex_fetch_data()*/

CS_RETCODE CS_PUBLIC
 ex_fetch_data(cmd)
 CS_COMMAND    *cmd;
 {
     CS_RETCODE       retcode;
     CS_INT           num_cols;
     CS_INT           i;
     CS_INT           j;
     CS_INT           row_count = 0;
     CS_DATAFMT       *datafmt;
     EX_COLUMN_DATA   *coldata;

    /*
     ** Determine the number of columns in this result
     ** set.
     */
     ...CODE DELETED...

    for (i = 0; i < num_cols; i++)
     {
         /*
         ** Get the column description.  ct_describe()
         ** fills the datafmt parameter with a
         ** description of the column.
         */
         retcode = ct_describe(cmd, (i + 1),
             &datafmt[i]);
         if (retcode != CS_SUCCEED)
         {
             ex_error("ex_fetch_data:ct_describe()
                 failed");
             break;
         }

        /* Now bind columns */
         ...CODE DELETED.....
     }

    /* Now fetch rows */
     ...CODE DELETED.....

    return retcode;
 }
```

このコードは、*exutils.c* サンプル・プログラムからの抜粋です。

**使用法**

- アプリケーションは、ct_describe を使用して、通常の結果カラム、リターン・パラメータ、ストアド・プロシージャのリターン・ステータス番号、または計算カラムの記述を取得できます。

  また、アプリケーションは、フォーマット情報を取得するために ct_describe を使用することができます。Client-Library は、ct_results の **result_type* を CS_ROWFMT_RESULT または CS_COMPUTEFMT_RESULT に設定して、フォーマット情報が利用できることを示します。

- ct_results が、**result_type* パラメータを CS_MSG_RESULT、CS_CMD_SUCCEED、CS_CMD_DONE、または CS_CMD_FAIL に設定した後、アプリケーションは ct_describe を呼び出すことはできません。これは、このような場合には記述する結果項目がないためです。
- アプリケーションは、現在の結果セットに存在する結果項目の数を知るために、ct_res_info を呼び出すことができます。
- アプリケーションで ct_bind を使用してプログラム変数に結果項目をバインドするときには、通常、事前に ct_describe を呼び出して結果データを記述する必要があります。
- 詳細については、「CS_DATAFMT 構造体」(93 ページ ) を参照してください。
- 結果タイプの記述については、「結果」(280 ページ ) を参照してください。

**参照**

ct_bind、ct_fetch、ct_res_info、ct_results、「結果」(280 ページ )

# ct_diag

**説明**

インライン・エラー処理を管理します。

**構文**

```
CS_RETCODE ct_diag(connection, operation, type, index,
          buffer)

CS_CONNECTION    *connection;
CS_INT           operation;
CS_INT           type;
CS_INT           index;
CS_VOID          *buffer;
```

**パラメータ**

*connection*

CS_CONNECTION 構造体を指すポインタです。CS_CONNECTION 構造体は、特定のクライアント／サーバ接続の情報を含んでいます。

*operation*

実行するオペレーションです。表 3-20 に *operation* の記号値を示します。

*type*

*type* には、*operation* に指定した値に応じて、メッセージ情報を受信する構造体のタイプかオペレーションの対象とするメッセージのタイプ、またはその両方を指定します。表 3-19 に *type* の記号値を示します。

**表 3-19：ct_diag type パラメータの値**

| type の値 | 意味 |
|---|---|
| SQLCA_TYPE | SQLCA 構造体。 |
| SQLCODE_TYPE | 長整数型の SQLCODE 構造体。 |
| SQLSTATE_TYPE | 6 バイト文字配列の SQLSTATE 構造体。 |
| CS_CLIENTMSG_TYPE | CS_CLIENTMSG 構造体。Client-Library メッセージを示すためにも使用される。 |
| CS_SERVERMSG_TYPE | CS_SERVERMSG 構造体。サーバ・メッセージを示すためにも使用される。 |
| CS_ALLMSG_TYPE | Client-Library メッセージとサーバ・メッセージ。 |

*index*

対象となるメッセージのインデックスです。最初のメッセージのインデックスが 1、2 番目のメッセージは 2、以下同様に続きます。

type が CS_CLIENTMSG_TYPE の場合、index は Client-Library メッセージだけを参照します。type が CS_SERVERMSG_TYPE の場合、index はサーバ・メッセージだけを参照します。type が CS_ALLMSG_TYPE の場合、index は Client-Library メッセージとサーバ・メッセージを合わせて参照します。

*buffer*

データ領域を指すポインタ。

*operation* の値に応じて、*buffer* は構造体または CS_INT を指します。

**戻り値**

ct_diag は、次の値を返します。

| 戻り値 | 意味 |
| --- | --- |
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。<br>原因となったエラーが接続を使用不能にしている場合、ct_diag は CS_FAIL を返す。 |
| CS_NOMSG | アプリケーションが、有効なインデックスの中で最大のものより大きいインデックスを持つメッセージを取得しようとした。たとえば、2 つのメッセージだけがキューイングされているとき、アプリケーションがメッセージ番号 3 を取得しようとした場合。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。<br>「非同期プログラミング」(12 ページ ) を参照。 |

ct_diag の失敗の主な原因は、次のとおりです。

- *connection* が無効
- メモリの割り付けができない
- パラメータの組み合わせが無効

**使用法**

**表 3-20：ct_diag パラメータの一覧**

| operation の値 | 結果アクション | タイプ | インデックス | バッファ |
| --- | --- | --- | --- | --- |
| CS_INIT | インライン・エラー処理を初期化する。 | CS_UNUSED | CS_UNUSED | NULL |
| CS_MSGLIMIT | 保管するメッセージの最大数を設定する。 | Client-Library メッセージだけを制限する場合、CS_CLIENTMSG_TYPE。<br>サーバ・メッセージだけを制限する場合、CS_SERVERMSG_TYPE。<br>Client-Library メッセージとサーバ・メッセージを合わせた総数を制限する場合、CS_ALLMSG_TYPE。 | CS_UNUSED | 整数値を指すポインタ。 |

| operation の値 | 結果アクション | タイプ | インデックス | バッファ |
|---|---|---|---|---|
| CS_CLEAR | この接続のメッセージ情報をクリアする。<br>*buffer* が NULL でなく、type が CS_ALLMSG_TYPE でない場合、ct_diag は **buffer* 構造体をブランクと NULL の両方またはいずれかで適切に初期化することによってクリアする。 | すべての有効な *type* 値。<br>*type* が CS_CLIENTMSG_TYPE の場合、ct_diag は Client-Library メッセージだけをクリアする。<br>*type* が CS_SERVERMSG_TYPE の場合、ct_diag はサーバ・メッセージだけをクリアする。<br>*type* がその他の有効な値の場合、ct_diag は Client-Library メッセージとサーバ・メッセージの両方をクリアする。 | CS_UNUSED | タイプが *type* によって定義されている構造体を指すポインタ、または NULL。 |
| CS_GET | 特定のメッセージを取得する。 | CS_ALLMSG_TYPE を除く、すべての有効な *type* 値。<br>*type* が CS_CLIENTMSG_TYPE の場合、Client-Library メッセージが CS_CLIENTMSG 構造体に取得される。<br>*type* が CS_SERVERMSG_TYPE の場合、サーバ・メッセージが CS_SERVERMSG 構造体に取得される。<br>*type* がその他の有効値である場合、Client-Library メッセージまたはサーバ・メッセージのどちらかが取得される。 | 1 から始まる取得するメッセージのインデックス。 | タイプが *type* によって定義されている構造体を指すポインタ。 |
| CS_STATUS | 現在保管されているメッセージ数を返す。 | Client-Library メッセージの数を取得する場合、CS_CLIENTMSG_TYPE。<br>サーバ・メッセージの数を取得する場合、CS_SERVERMSG_TYPE。<br>Client-Library メッセージとサーバ・メッセージを合わせた総数を取得する場合、CS_ALLMSG_TYPE。 | CS_UNUSED | 整数変数を指すポインタ。 |

| operation の値 | 結果アクション | タイプ | インデックス | バッファ |
|---|---|---|---|---|
| CS_EED_CMD | **buffer* を拡張エラー・データを含む CS_COMMAND 構造体のアドレスに設定する。 | CS_SERVERMSG_TYPE | 拡張エラー・データが利用できるメッセージの 1 から開始されるインデックス。 | ポインタ変数を指すポインタ。 |

- Client-Library アプリケーションは、Client-Library メッセージとサーバ・メッセージを次の 2 つの方法で処理できます。
  - 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クライアント・メッセージ・コールバックおよびサーバ・メッセージ・コールバックをインストールするために、ct_callback を呼び出すことができます。
  - アプリケーションは、ct_diag を使用して、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できます。

  アプリケーションは、この 2 つの方法を交互に切り替えることができます。この実行方法の詳細については、「エラー処理」(136 ページ ) を参照してください。

- ct_diag は、特定の接続のインライン・メッセージ処理を管理します。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、コンテキスト・レベルでは ct_diag を使用できません。つまり、アプリケーションは、ct_diag を使用して (CS_CONNECTION ではなく ) CS_CONTEXT をパラメータとするルーチンが生成したメッセージを取得することはできません。これらのメッセージは、インライン・エラー処理を使用しているアプリケーションでは利用できません。
- アプリケーションは、Client-Library メッセージまたはサーバ・メッセージのいずれかまたは両方のオペレーションを実行できます。

  たとえば、アプリケーションは、次のように使用することによって、サーバ・メッセージに影響を及ぼすことなく、Client-Library メッセージをクリアできます。

  ```
  ct_diag(connection, CS_CLEAR, CS_CLIENTMSG,
              CS_UNUSED, NULL);
  ```

- ct_diag により、アプリケーションはメッセージ情報を取得して、標準 Client-Library 構造体 (CS_CLIENTMSG と CS_SERVERMSG)、SQLCA、SQLCODE、または SQLSTATE にその結果を入れることができます。メッセージを取得する場合、ct_diag は、*type* で示された型の構造体を *buffer* が指すものとみなします。

  取得したメッセージを SQLCA、SQLCODE、または SQLSTATE に保管するアプリケーションでは、Client-Library の CS_EXTRA_INF プロパティを CS_TRUE に設定する必要があります。これは、その SQL 構造体が、Client-Library のエラー処理メカニズムによって通常は返されない情報を必要とするためです。

  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 に設定できます。この場合、標準 Client-Library メッセージとして追加情報が返されます。

- ct_diag が、新しいメッセージの保存に十分な内部記憶領域を確保していない場合、読み込まれていないメッセージはすべて破棄され、メッセージの保存が停止されます。次回、operation を CS_GET に設定して呼び出されたとき、空き領域の問題を示すために特別なメッセージを返します。このメッセージを返した後、ct_diag はメッセージの保存を再度開始します。

**インライン・エラー処理の初期化**

- アプリケーションでインライン・エラー処理を初期化するには、*operation* に CS_INIT を指定して ct_diag を呼び出します。
- 通常、接続がインライン・エラー処理を使用する場合、アプリケーションは、割り付け直後に ct_diag を呼び出して、接続に対するインライン・エラー処理を初期化する必要があります。

**メッセージのクリア**

- アプリケーションで接続のメッセージ情報をクリアするには、*operation* に CS_CLEAR を指定して ct_diag を呼び出します。
  - Client-Library メッセージだけをクリアするために、アプリケーションは *type* を CS_CLIENTMSG_TYPE にして渡します。
  - サーバ・メッセージだけをクリアするために、アプリケーションは *type* を CS_SERVERMSG にして渡します。
  - Client-Library メッセージとサーバ・メッセージの両方をクリアするには、*type* として SQLCA_TYPE、SQLCODE_TYPE、SQLSTATE_TYPE、または CS_ALLMSG_TYPE を渡します。

- CS_ALLMSG_TYPE 以外の値を *type* に指定すると、次のようになります。
  - ct_diag は、*buffer* が *type* の値と対応する型の構造体を指すものとみなします。
  - ct_diag は、**buffer* 構造体にブランクと NULL の両方またはいずれかを適宜設定して、この構造体をクリアします。
- アプリケーションが operation に CS_CLEAR を設定して ct_diag を明示的に呼び出すまで、メッセージ情報はクリアされません。メッセージを取得してもメッセージ・キューからは削除されません。

**メッセージの取得**

- メッセージを取得するために、アプリケーションは、CS_GET を *operation* に、メッセージを取得する構造体のタイプを *type* に、1 から始まる対象となるメッセージのインデックスを *index* に、さらに、適切なタイプの構造体を **buffer* に設定して、ct_diag を呼び出します。
- *type* が CS_CLIENTMSG_TYPE の場合、index は Client-Library メッセージだけを参照します。*type* が CS_SERVERMSG_TYPE の場合、*index* はサーバ・メッセージだけを参照します。*type* がその他の値の場合、*index* は両方のタイプのメッセージを合わせた「集合キュー」を参照します。
- ct_diag は **buffer* 構造体にメッセージ情報を記入します。
- 指定されている最大有効インデックスよりも大きいインデックスを持つメッセージを取得しようとすると、ct_diag は、使用可能なメッセージがないことを示す CS_NOMSG を返します。
- これらの構造体については、次のページを参照してください。
  - 「SQLCA 構造体」(106 ページ )
  - 「SQLCODE 構造体」(108 ページ )
  - 「SQLSTATE 構造体」(108 ページ )
  - 「CS_CLIENTMSG 構造体」(85 ページ )
  - 「CS_SERVERMSG 構造体」(103 ページ )

**メッセージの制限**

- 制限されたメモリを持つプラットフォームで動作するアプリケーションでは、Client-Library が保存するメッセージ数を制限する場合があります。

- アプリケーションは、保存される Client-Library メッセージやサーバ・メッセージの数、およびメッセージの総数を制限できます。
- 保存するメッセージの数をアプリケーションで制限するには、*operation* に CS_MSGLIMIT を指定し、*type* に CS_CLIENTMSG_TYPE、CS_SERVERMSG_TYPE、または CS_ALLMSG_TYPE を指定して ct_diag を呼び出します。
  - *type* が CS_CLIENTMSG_TYPE の場合、Client-Library メッセージの数が制限されます。
  - *type* が CS_SERVERMSG_TYPE の場合、サーバ・メッセージの数が制限されます。
  - *type* が CS_ALLMSG_TYPE の場合、Client-Library メッセージとサーバ・メッセージを合わせた総数が制限されます。
  - 特定のメッセージ制限値に達すると、Client-Library はそのタイプの新しいメッセージを廃棄します。メッセージ総数が制限値に達すると、Client-Library は新しいメッセージをすべて廃棄します。Client-Library がメッセージを廃棄する場合、同じ意味のメッセージを保存します。
  - アプリケーションは、現在保存されているメッセージの数よりも少ないメッセージ数を設定できません。
  - Client-Library のデフォルト動作では、保存できるメッセージ数を制限していません。アプリケーションでこのデフォルトの動作に戻すには、メッセージの制限数を CS_NO_LIMIT に設定します。

**メッセージ数の取得**

- 現在のメッセージ数を取得するために、アプリケーションは、*operation* を CS_STATUS、*type* を対象となるメッセージのタイプにして、ct_diag を呼び出します。

**拡張エラー・データのための CS_COMMAND の取得**

- 拡張エラー・データがある場合に、拡張エラー・データが格納された CS_COMMAND 構造体のポインタを取得するには、*operation* に CS_EED_CMD、*type* に CS_SERVERMSG_TYPE を指定して ct_diag を呼び出します。ct_diag は **buffer* を拡張エラー・データが格納された CS_COMMAND 構造体のアドレスに設定します。

- アプリケーションがサーバ・メッセージを CS_SERVERMSG 構造体から取得するとき、Client-Library は、CS_SERVERMSG 構造体の *status* フィールドに CS_HASEED ビットを設定することによって、拡張エラー・データがそのメッセージで利用できることを示します。
- 拡張エラー・データが利用できないとき、*operation* を CS_EED_CMD にして ct_diag を呼び出すとエラーになります。
- 「拡張エラー・データ」(143 ページ ) を参照してください。

**連続しているメッセージと *ct_diag***

- アプリケーションで連続しているエラー・メッセージを使用している場合、ct_diag は、メッセージではなく、メッセージのまとまりに対して動作します。これには、次のような影響があります。
  - ct_diag(CS_GET、*index*) 呼び出しは、*index* 番号のメッセージのまとまりを返します。
  - A ct_diag(CS_MSGLIMIT) 呼び出しは、Client-Library が保管するまとまりの数を制限します。メッセージの数を制限するのではありません。
  - ct_diag(CS_STATUS) の呼び出しは、現在保管されているメッセージ数ではなく、現在保管されているまとまりの数を返します。
- 「長いメッセージの連続化」(141 ページ ) を参照してください。

**参照**

「エラー処理」(136 ページ ) 「CS_CLIENTMSG 構造体」(85 ページ )、「CS_SERVERMSG 構造体」(103 ページ )、「SQLCA 構造体」(106 ページ )、「SQLCODE 構造体」(108 ページ )、「SQLSTATE 構造体」(108 ページ )、ct_callback、ct_options

# ct_ds_dropobj

**説明**

ディレクトリ・オブジェクトによって使用されているメモリを解放します。

**構文**

```
CS_RETCODE ct_ds_dropobj(connection, ds_object)

CS_CONNECTION     *connection;
CS_DS_OBJECT      *ds_object;
```

**パラメータ**

*connection*

CS_CONNECTION 構造体を指すポインタです。ct_ds_lookup は、CS_CONNECTION 構造体にインストールされているアプリケーションのディレクトリ・コールバックに対して検索結果を返します。

*ds_object*

廃棄対象のディレクトリ・オブジェクトを指すポインタです。

**戻り値**

ct_ds_dropobj は次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |

**使用法**

- ct_ds_dropobj は、CS_DS_OBJECT 隠し構造体を廃棄して、その構造体が使用しているメモリを解放します。ct_ds_dropobj は、オブジェクトに関連付けられているディレクトリ・エントリにはなにも影響を与えません。
- ディレクトリ・オブジェクトに関する情報を保存しておくには、オブジェクトを削除する前にその情報をアプリケーションのメモリにコピーしてください。
- アプリケーションが ct_ds_dropobj を使用してディレクトリ・オブジェクトを暗黙的に削除しない場合は、親接続を削除するために ct_con_drop を呼び出したときに、Client-Library がそれらのディレクトリ・オブジェクトを削除します。

**参照**

ct_ds_lookup、ct_ds_objinfo

# ct_ds_lookup

**説明**

ディレクトリ検索オペレーションを開始またはキャンセルします。

**構文**

```
CS_RETCODE ct_ds_lookup(connection,
          action, reqid, lookup_info,
          userdata)

CS_CONNECTION        connection;
CS_INT               action;
CS_INT               *reqid;
CS_DS_LOOKUP_INFO    *lookup_info;
CS_VOID              *userdata;
```

**パラメータ**

*connection*

CS_CONNECTION 構造体を指すポインタです。CS_CONNECTION 構造体は特定の接続についての情報を保持します。

*action*

次の記号値のいずれかです。

| アクション | 実行される機能 |
|---|---|
| CS_CLEAR | *reqid* によって指定される、ディレクトリ検索オペレーションをキャンセルする。非同期接続でのみサポートされる。 |
| CS_SET | ディレクトリ検索オペレーションを開始する。 |

*reqid*

CS_INT 変数を指すポインタです。

*action* に CS_SET を指定すると、Client-Library は **reqid* に要求識別子を返します。

*action* に CS_CLEAR を指定すると、**reqid* はキャンセルするオペレーションの要求識別子を指定します。

*lookup_info*

CS_DS_LOOKUP_INFO 構造体のアドレスを指定します。

CS_DS_LOOKUP_INFO 構造体は次のように定義されます。

```
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;
```

*action* に CS_SET を指定すると、**lookup_info* のフィールドは次のように設定されます。

**表 3-21：ct_ds_lookup(CS_SET) 呼び出し用 *lookup_info の内容**

| フィールド | 設定内容 |
|---|---|
| *objclass* | 検索対象のディレクトリ・オブジェクトのクラスを指定する CS_OID 構造体のアドレス。*objclass->oid_buffer* にはオブジェクト・クラスの OID 文字列、*objclass->oid_length* には OID 文字列の長さを指定する (null ターミネータはカウントしない)。<br>ct_ds_lookup は、クラスが *lookup_info->objclass* の内容と一致するディレクトリ・エントリだけを探す。 |
| *path* | 予約済み。Client-Library の将来のバージョンとの互換性を保つために NULL に設定する。 |
| *pathlen* | 予約済み。Client-Library の将来のバージョンとの互換性を保つために 0 に設定する。 |
| *attrfilter* | 予約済み。Client-Library の将来のバージョンとの互換性を保つために NULL に設定する。 |
| *attrselect* | 予約済み。Client-Library の将来のバージョンとの互換性を保つために NULL に設定する。 |

**注意**　非同期モードでは、接続の完了コールバックまたは ct_poll によって要求の完了またはキャンセルが示されるまで、**lookup_info* の内容とその中に含まれるポインタは有効でなければなりません。

*action* に CS_CLEAR を指定すると、*lookup_info* を NULL にして渡す必要があります。

*userdata*

ディレクトリ・コールバックに渡すユーザ割り付けデータのアドレスです。

action に CS_SET を指定した場合、*userdata* はオプションになり、NULL として渡すことができます。ct_ds_lookup は一致するディレクトリ・エントリを探し、Client-Library は接続のディレクトリ・コールバックを呼び出します。ディレクトリ・コールバックは *userdata* として指定されたアドレスを受け取ります。*userdata* は、コールバックが ct_ds_lookup を呼び出したメインライン・コードに取得結果を渡す手段を提供します。

action に CS_CLEAR を指定すると、*userdata* は (CS_VOID *) NULL として渡す必要があります。

**戻り値**

ct_ds_lookup は次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗した。 |
| CS_PENDING | 非同期ネットワーク I/O が有効。「非同期プログラミング」(12 ページ ) を参照。<br><br>**注意**　Client-Library がスレッド駆動型 I/O を使用しないプラットフォームでは、接続の CS_NETIO 設定が CS_ASYNC_IO である場合でも常に、アプリケーションで ct_ds_lookup の完了を調べる必要がある。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |
| CS_CANCELED | 検索要求はアプリケーションによってキャンセルされた。検索要求は、非同期接続に関してだけキャンセルできる。 |

**例**

この例の手順の説明については、『Open Client Client-Library/C プログラマーズ・ガイド』の「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。

**使用法**

- ct_ds_lookup は、ディレクトリ検索要求を開始またはキャンセルします。
- ct_ds_lookup、ct_ds_objinfo、および接続のディレクトリ・コールバック・ルーチンは、Client-Library アプリケーションがディレクトリ・エントリを調べるためのメカニズムを提供します。一般的な処理手順を次に示します。

  a　アプリケーションは検索結果を処理するためにディレクトリ・コールバックをインストールします。

  b　( ネットワークベースのディレクトリのみ ) アプリケーションは検索対象のサブツリーを指定するために、CS_DS_DITBASE 接続プロパティを設定します。

  c　( ネットワークベースのディレクトリのみ ) アプリケーションは、検索を行うために必要なすべてのディレクトリ・サービス・プロパティを設定します。

  d　アプリケーションは ct_ds_lookup を呼び出して、ディレクトリの検索を開始します。

e　Client-Library は、検出されたディレクトリ・エントリごとに一度ずつ、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します。コールバックは呼び出されるたびに CS_DS_OBJECT ポインタを受け取り、そのポインタがディレクトリ・エントリの内容の要約を提供します。

f　アプリケーションは、必要に応じて何度でも ct_ds_objinfo を呼び出して各オブジェクトを調べることができます。この処理はコールバックまたはメインライン・コード内で実行できます。

g　アプリケーションは、検索によって返されたディレクトリ・オブジェクトの処理が終了すると、ct_ds_dropobj を呼び出して使用していたメモリを解放します。

**ディレクトリ・コールバック**

- ディレクトリ検索の結果は、接続のディレクトリ・コールバックに返されます。Client-Library は検索されたエントリごとに一度ずつディレクトリ・コールバックを呼び出し、呼び出すごとにエントリについて記述する CS_DS_OBJECT へのポインタを受け取ります。
- 検索を開始する前に、アプリケーションは、検索結果を受け取れるように、ct_callback(CS_DS_LOOKUP_CB) を使用してディレクトリ・コールバックをインストールする必要があります。そうしないと、結果は廃棄されてしまいます。
- ディレクトリ・コールバックの説明については「ディレクトリ・コールバックの定義」(42 ページ ) を参照してください。

**ディレクトリ検索の開始**

- ct_ds_lookup(CS_SET) は、CS_DS_PROVIDER 接続プロパティの現在の設定で指定されているディレクトリ・ドライバに要求を渡します。「ディレクトリ・サービス・プロバイダ」(132 ページ ) を参照してください。
- 検索に *interfaces* ファイルではなくディレクトリ・サービスを使用した場合は、次の条件に合うすべてのディレクトリ・エントリが検索されます。
  - オブジェクト・クラスが ct_ds_lookup の *lookup_info* パラメータの *objclass* フィールドに指定された OID に一致するエントリ。
  - CS_DS_DITBASE 接続プロパティとして指定されたディレクトリ・ノードの下位にあるエントリ。「ディレクトリ検索のベース」(128 ページ ) を参照。
  - CS_DS_SEARCH 接続プロパティで定義された深さの制限内にあるエントリ。「ディレクトリ・サービスの検索の深さ」(135 ページ ) を参照。デフォルトでは、ct_ds_lookup は DIT ベース・ノードの直下にあるエントリだけを返す。

- *interfaces* ファイルを検索する場合は、サーバ・オブジェクト (CS_OID_SERVERCLASS) に対して検索を実行してください。すると、*interfaces* ファイル内に定義されているすべてのサーバの記述が返されます。
- ディレクトリ・サービス・プロバイダの中には、ディレクトリ・エントリに対するアクセスを制限しているものがあります。この場合、アプリケーションは CS_DS_PRINCIPAL 接続プロパティに対する値を提供する必要があります。「ディレクトリ・サービスのプリンシパル名」(131 ページ ) を参照してください。

### 同期接続と非同期接続のディレクトリ検索

- ct_ds_lookup(CS_SET) は検索要求を基本となるディレクトリ・サービスに渡します。要求は、一致するオブジェクトを Client-Library に返します。処理手順は、非同期接続と同期接続の場合で異なります。
- 同期接続の場合、検索要求が完了し、アプリケーションがディレクトリ・コールバックに返されたオブジェクトを調べ終わるまで、ct_ds_lookup の処理が中断します。同期処理は次のように発生します。

  a　アプリケーションのメインライン・コードが ct_ds_lookup(CS_SET) を呼び出して、検索オペレーションを開始します。

  b　検索が完了すると、Client-Library はディレクトリ・コールバックの呼び出しを開始して、返されたオブジェクトをアプリケーションに渡します。

  なんらかの原因で検索が失敗した場合は、Client-Library が *status* コールバック引数の値として CS_FAIL を渡します。

  コールバックは、検索された各オブジェクトごとに一度ずつ、またはディレクトリ・コールバックから CS_SUCCEED が返されるまで繰り返し呼び出されます。

  c　ct_ds_lookup はメインライン・コードに制御を戻します。

  **注意**　完全な非同期サポートを提供するためには、スレッド駆動型の I/O を使用する Client-Library のバージョンが ct_ds_lookup に必要です。他のバージョンでは、CS_NETIO が CS_ASYNC_IO または CS_DEFER_IO に設定された場合に、ct_ds_lookup が遅延非同期動作を行います。

- 完全非同期接続または遅延非同期接続の場合、ct_ds_lookup はすぐに制御を戻します。処理の詳細は次のとおりです。

a　アプリケーションのメインライン・コードが ct_ds_lookup(CS_SET) を呼び出して検索オペレーションを開始します。

b　Client-Library はディレクトリ・サービス・ドライバに要求を渡します。

ディレクトリ・ドライバが要求を受け入れると、ct_ds_lookup が CS_PENDING を開始します。Client-Library がスレッドを使用しているプラットフォームでは、Client-Library は内部ワーカー・スレッドを生成し、この時点で要求を処理します。

要求をキューに入れることができない場合は、ct_ds_lookup が CS_FAIL を返します。

c　接続のディレクトリ・コールバックが呼び出されます。完全非同期接続では、Client-Library がコールバックを自動的に呼び出します。遅延非同期接続では、アプリケーションが ct_poll を呼び出すときに Client-Library がコールバックを呼び出します。

検索によってオブジェクトが返された場合は、アプリケーションがすべてのオブジェクトを検出するか、コールバックが CS_SUCCEED を返すまで、コールバックが繰り返し呼び出されてオブジェクトをアプリケーションに渡します。

検索してもオブジェクトが見つからなかった場合には、値が 0 の *numentries* コールバック引数を使用してコールバックが一度だけ呼び出されます。

検索が失敗するかキャンセルされた場合は、*status* コールバック引数として CS_FAIL または CS_CANCELED を使用して、コールバックが一度だけ呼び出されます。

d　接続の完了コールバックが呼び出されます。完全非同期接続では、Client-Library が完了コールバックを自動的に呼び出します。遅延非同期接続では、アプリケーションは ct_poll を使用して要求の完了を調べる必要があるので、ct_poll がコールバックを呼び出します。完了コールバックは、検索オペレーションの最終的なリターン・ステータス (CS_SUCCEED、CS_FAIL、または CS_CANCELLED) を受け取ります。

---

**注意**　完全非同期接続では、ディレクトリ・コールバックと完了コールバックが Client-Library の内部スレッドによって呼び出されます。メインライン・コード、他のアプリケーション・スレッド、Client-Library スレッドで実行されるコールバック・コードによる同時アクセスから共有データが保護されていることを確認してください。**userdata* の内容も同時アクセスから保護されている必要があります。

---

**ディレクトリ検索のキャンセル ( 非同期接続の場合のみ )**

- 接続のネットワーク I/O モード (CS_NETIO プロパティ ) が完全非同期式か遅延非同期式である場合は、検索が完了する前に ct_ds_lookup(CS_CLEAR, &*reqid*) を呼び出すことによって検索オペレーションをキャンセルできます。
    - ct_ds_lookup(CS_CLEAR) は、CS_SUCCEED ステータスまたは CS_FAIL ステータスですぐに制御を戻します。ただし、ディレクトリ・プロバイダが要求を確認するまで、その接続はビジーのままです。このとき、Client-Library は CS_CANCELED ステータスでディレクトリ・コールバックと完了コールバックを順に呼び出します。
    - 接続の完了コールバックや ct_poll によって接続の完了が示された後は、ct_ds_lookup(CS_CLEAR) を呼び出すことはできません。この時点では、要求の処理が終わっているので ct_ds_lookup(CS_CLEAR) は失敗します。
- アプリケーションは、ディレクトリ・コールバックから CS_CONTINUE ではなく CS_SUCCEED を返すことによって検索結果をトランケートすることもできます。
- 同期接続に対して行われた検索要求はキャンセルできません。ただし、基本となるディレクトリ・サービス・プロバイダがサポートしている場合は、要求完了までの時間制限を設定できます。「ディレクトリ検索の時間制限」(136 ページ ) を参照してください。

**参照**　ct_ds_objinfo、ct_ds_dropobj、「ディレクトリ・サービス」(115 ページ )、「サーバ・ディレクトリ・オブジェクト」(319 ページ )

# ct_ds_objinfo

**説明**　ディレクトリ・オブジェクトに関する情報を取得します。

**構文**

```
CS_RETCODE ct_ds_objinfo(ds_object, action, infotype, number, buffer,
                                            buflen, outlen)

CS_DS_OBJECT     *ds_object;
CS_INT           action;
CS_INT           infotype;
CS_INT           number;
CS_VOID          *buffer;
CS_INT           buflen;
CS_INT           *outlen;
```

**パラメータ**

*ds_object*

CS_DS_OBJECT 構造体を指すポインタです。アプリケーションは、ディレクトリ・コールバックへの入力パラメータとしてディレクトリ・オブジェクト・ポインタを受け取ります。

*action*

CS_GET である必要があります。

*infotype*

**buffer* に取り出す情報のタイプです。有効なタイプの説明については、表 3-22 ( 525 ページ ) を参照してください。

*number*

*infotype* が CS_DS_ATTRIBUTE または CS_DS_ATTRVALS の場合は、*number* に、取得する属性の番号を指定します。属性番号は 1 から始まります。

他の *infotype* 値については、*number* に CS_UNUSED を指定して渡します。

*buffer*

要求した情報を格納するバッファのアドレスを指定します。表 3-22 ( 525 ページ ) に、*infotype* の値に対応する **buffer* のデータ型を示します。

*buflen*

**buffer* のバイト単位の長さです。

*outlen*

この引数を指定すると、**buffer* に返された値の長さが **outlen* に設定されます。この引数はオプションであり、NULL として渡すことができます。

**戻り値**

ct_ds_objinfo は次の値を返します。

| 戻り値 | 意味 |
| --- | --- |
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗した。 |

**例**

この例の手順の説明については、『Open Client Client-Library/C プログラマーズ・ガイド』の「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。

**使用法**

次の表は、action に CS_GET を指定する場合の ct_ds_objinfo 呼び出しの構文を示します。

**表 3-22：ct_ds_objinfo パラメータの一覧**

| infotype の値 | number の値 | *buffer のデータ型 | *buffer に書き込まれる値 |
|---|---|---|---|
| CS_DS_CLASSOID | CS_UNUSED | CS_OID 構造体。 | ディレクトリ・オブジェクト・クラスの OID。 |
| CS_DS_DIST_NAME | CS_UNUSED | CS_CHAR 配列。 | オブジェクトの完全に修飾された ( 区別できる ) ディレクトリ名。最大 512 バイト。<br>出力名は null で終了する。<br>*outlen* が NULL ではない場合、Client-Library は **buffer* に書き込まれるバイトの数 (null ターミネータを含まない ) を **outlen* に入れる。 |
| CS_DS_NUMATTR | CS_UNUSED | CS_INT 変数。 | オブジェクトに対応する属性の数。 |
| CS_DS_ATTRIBUTE | 正の整数 | CS_ATTRIBUTE 構造体。 | *number* の値で指定される属性のためのメタデータを保持する CS_ATTRIBUTE 構造体。<br>CS_ATTRVALUE 構造体および CS_ATTRIBUTE データ構造体の詳細については、「オブジェクト属性と属性値の取得」(528 ページ ) を参照。 |
| CS_DS_ATTRVALS | 正の整数 | CS_ATTRVALUE 共用体の配列。配列は CS_ATTRIBUTE 構造体によって示される値の数よりも十分に長くする。 | *number* の値で指定される属性の値。<br>CS_ATTRVALUE 構造体および CS_ATTRIBUTE データ構造体の詳細については、「オブジェクト属性と属性値の取得」(528 ページ ) を参照。 |

- ct_ds_lookup、ct_ds_objinfo、および接続のディレクトリ・コールバック・ルーチンは、Client-Library アプリケーションがディレクトリ・エントリを調べるためのメカニズムを提供します。一般的な処理手順を次に示します。

  a　アプリケーションは検索結果を処理するためにディレクトリ・コールバックをインストールします。

  b　( ネットワークベースのディレクトリのみ ) アプリケーションは検索対象のサブツリーを指定するために、CS_DS_DITBASE 接続プロパティを設定します。

  c　( ネットワークベースのディレクトリのみ ) アプリケーションは、検索を行うために必要なすべてのディレクトリ・サービス・プロパティを設定します。

d アプリケーションは ct_ds_lookup を呼び出して、ディレクトリの検索を開始します。

e Client-Library は、検出されたディレクトリ・エントリごとに一度ずつ、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します。コールバックは呼び出されるたびに CS_DS_OBJECT ポインタを受け取り、そのポインタがディレクトリ・エントリの内容の要約を提供します。

f アプリケーションは、必要に応じて何度でも ct_ds_objinfo を呼び出して各オブジェクトを調べることができます。この処理はコールバック、メインライン・コード、またはその両方で実行できます。

g アプリケーションは、検索によって返されたディレクトリ・オブジェクトの処理が終了すると、ct_ds_dropobj を呼び出して使用していたメモリを解放します。

### ディレクトリ・オブジェクトの構造体

- ディレクトリの物理構造は、ディレクトリ・サービス・プロバイダごとに異なります。Client-Library は物理ディレクトリ・エントリを CS_DS_OBJECT 隠し構造体の内容にマップします。これによって、特定の物理ディレクトリ構造に対するアプリケーションの依存性を最小にすることができます。
- アプリケーションは ct_ds_objinfo を使用して CS_DS_OBJECT 隠し構造体の内容を調べます。
- 一般的なアプリケーションでは、ct_ds_objinfo を何度も呼び出してオブジェクトの内容を調べます。一般的な呼び出し手順を次に示します。

  a アプリケーションは、CS_DS_CLASSOID として *infotype*、CS_OID 構造体のアドレスとして *buffer* を使用して ct_ds_objinfo を呼び出し、ディレクトリ・エントリのオブジェクト・クラスを与える OID を取得します。この手順はオプションであり、アプリケーションがオブジェクト・クラスをすでに認識している場合は省略できます。

  b アプリケーションは、CS_DS_DISTNAME として *infotype*、文字列のアドレスとして *buffer* を使用して ct_ds_objinfo を呼び出し、エントリの完全に修飾された名前を取得します。

  c アプリケーションは、CS_DS_NUMATTR として *infotype*、CS_INT 変数のアドレスとして *buffer* を使用して ct_ds_objinfo を呼び出し、オブジェクトに存在する属性の数を取得します。

d　アプリケーションは、CS_DS_NUMATTR として *infotype*、CS_ATTRIBUTE 構造体のアドレスとして *buffer* を使用して ct_ds_objinfo を呼び出し、オブジェクトの各属性のメタデータを取得します。

e　アプリケーションは、CS_ATTRIBUTE 構造体の *attribute.attr_type* フィールドに指定されている OID を調べることによって、属性の値が必要かどうか調べます。アプリケーションに値が必要な場合は、*attribute.attr_numvals* で示されるサイズの CS_ATTRVALUE 共用体の配列を割り付けます。アプリケーションは次に CS_DS_ATTRVALS として *infotype*、配列のアドレスとして *buffer* を使用して ct_ds_objinfo を呼び出し、値を取得します。

f　アプリケーションは、各属性について手順 d と e を繰り返します。

**オブジェクト・クラスの取得**

- 返されるディレクトリ・オブジェクトのクラスを識別するには、*infotype* に CS_DS_CLASSOID を指定し、*buffer* に CS_OID 構造体のアドレスを指定して ct_ds_objinfo を呼び出します。
  - ct_ds_objinfo は、ディレクトリ・エントリ・オブジェクト・クラスの OID を指定するために CS_OID のフィールドを設定します。

    返される CS_OID 構造体の *oid–>oid_buffer* フィールドには、オブジェクト・クラスの OID 文字列が含まれています。*oid–>oid_length* には、文字列の長さ (null ターミネータはカウントしない ) が含まれています。
  - *oid_buffer* フィールドは、必要なオブジェクト・クラスの OID 文字列定数と比較できます。

**完全に修飾されたエントリ名の取得**

- オブジェクトの完全に修飾されたディレクトリ名を取得するには、CS_DS_DIST_NAME として *infotype*、CS_CHAR 文字列のアドレスとして *buffer* を使用して ct_ds_objinfo を呼び出します。
- 名前文字列は null で終わります。
- サーバ (CS_OID_OBJSERVER) クラスのオブジェクトについては、完全に修飾された名前をアプリケーションから ct_connect に渡して、そのオブジェクトが表すサーバへの接続をオープンできます。

#### オブジェクト属性と属性値の取得

- ディレクトリ・オブジェクトの属性は、番号付きのセットとして使用できます。ただし、セット内での各属性の位置はディレクトリ・サービス・プロバイダごとに異なる場合があります。また、ディレクトリ・プロバイダによっては、属性の順序が一定に保たれることが保証されない場合があります。Sybase でも、新しいバージョンでディレクトリ・オブジェクト・クラスに新しい属性を追加することがあります。

  このため、オブジェクト属性の数と順序に依存しないで動作するように、アプリケーションを作成する必要があります。

- ct_ds_objinfo は CS_ATTRIBUTE 構造体を使用して属性値のメタデータを定義し、CS_ATTRVALUE 共用体の配列に値を返します。

**CS_ATTRIBUTE 構造体**

CS_ATTRIBUTE 構造体は、ディレクトリ・オブジェクトの属性を記述するために ct_ds_objinfo で使用されます。

```
typedef struct
 {
     CS_OID          attr_type;
     CS_INT          attr_syntax;
     CS_INT          attr_numvals;
 } CS_ATTRIBUTE;
```

各パラメータ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。

ディレクトリ・オブジェクト・クラスの定義は、オブジェクトが保持できる属性タイプを決定します。

*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。

**CS_ATTRVALUE 共用体**

属性値は、CS_ATTRVALUE 共用体を使用してアプリケーションに返されます。この共用体は、属性値を表すために必要な各データ型のメンバで構成されています。宣言は次のようなものです。

```
typedef union _cs_attrvalue
 {
         CS_STRING        value_string;
         CS_BOOL          value_boolean;
```

```
            CS_INT        value_enumeration;
            CS_INT        value_integer;
            CS_TRANADDR   value_tranaddr;
            CS_OID        value_oid;
    } CS_ATTRVALUE;
```

属性値は ct_ds_objinfo によって CS_ATTRVALUE 共用体の配列に取り出されます。配列のサイズは、CS_ATTRIBUTE 構造体の *attr_numvals* フィールドと一致する必要があります。この値は、CS_ATTRIBUTE 構造体の *attr_syntax* フィールドによって指定される共用体メンバとして扱う必要があります。表 3-23 は属性構文指定子と CS_ATTRVALUE のメンバの関連を示します。

**表 3-23：CS_ATTRVALUE 共用体の構文指定子**

| 属性構文指定子 | 共用体メンバ |
| --- | --- |
| CS_ATTR_SYNTAX_STRING | *value_string*<br>文字列値は CS_STRING 構造体によって表される。後述の「文字列値」を参照。 |
| CS_ATTR_SYNTAX_BOOLEAN | *value_boolean*<br>ブール値は CS_BOOL として表される。 |
| CS_ATTR_SYNTAX_ENUMERATION | *value_enumeration*<br>列挙されている値は CS_INT として表される。 |
| CS_ATTR_SYNTAX_INTEGER | *value_integer*<br>整数値は CS_INT として表される。 |
| CS_ATTR_SYNTAX_TRANADDR | *value_tranaddr*<br>トランスポート・アドレスは CS_TRANADDR 構造体として表される。後述の「トランスポート・アドレス値」を参照。 |
| CS_ATTR_SYNTAX_OID | *value_oid*<br>OID 値は CS_OID 構造体として表される。101 ページの「説明」を参照。 |

**文字列値**

CS_STRING 構造体は次のように定義されます。

```
typedef struct _cs_string
 {
         CS_INT  str_length;
         CS_CHAR str_buffer[CS_MAX_DS_STRING];
 } CS_STRING;
```

*str_buffer* の内容は null で終了します。*str_length* の長さには null ターミネータはカウントされません。

**トランスポート・アドレス値**

トランスポート・アドレスは、次に示す CS_TRANADDR 構造体の中に Sybase 固有のフォーマットでコード化されます。

```
typedef struct _cs_tranaddr
 {
        CS_INT      addr_accesstype;
        CS_STRING   addr_trantype;
        CS_STRING   addr_tranaddress;
 } CS_TRANADDR;
```

**参照**

ct_ds_lookup, ct_ds_dropobj、「ディレクトリ・サービス」(115 ページ )、「サーバ・ディレクトリ・オブジェクト」(319 ページ )

# ct_dynamic

**説明**

動的 SQL コマンドを開始します。

**構文**

```
CS_RETCODE ct_dynamic(cmd, type, id, idlen, buffer, buflen)

CS_COMMAND         *cmd;
CS_INT             type;
CS_CHAR            *id;
CS_INT             idlen;
CS_CHAR            *buffer;
CS_INT             buflen;
```

**パラメータ**

*cmd*

クライアント／サーバ・オペレーションを管理する CS_COMMAND 構造体を指すポインタです。

*type*

開始する動的 SQL コマンドのタイプです。表 3-24 に、*type* の記号値を示します。

*id*

文識別子を指すポインタです。この識別子はアプリケーションで指定され、サーバの標準に準拠する必要があります。

*idlen*

**id* のバイト単位の長さです。**id* が null で終了する場合は、*idlen* に CS_NULLTERM を指定して渡します。*id* が NULL の場合は、*idlen* に CS_UNUSED を指定して渡してください。

*buffer*

データ領域を指すポインタ。

*buflen*

**buffer* のバイト単位の長さです。**buffer* が null で終了する場合は、*buflen* には CS_NULLTERM を指定して渡してください。*buffer* が NULL の場合は、*buflen* を CS_UNUSED として渡してください。

**戻り値**

ct_dynamic は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

**使用法**

表 3-24 に ct_dynamic の使用法を示します。

**表 3-24：ct_dynamic パラメータの一覧**

| type の値 | 結果 | *id の値 | *buffer の値 |
|---|---|---|---|
| CS_CURSOR_DECLARE | 前もって準備された SQL 文にカーソルを宣言する。 | 準備文の識別子。 | カーソル名。 |
| CS_DEALLOC | 準備された SQL 文の割り付けを解除する。 | 準備文の識別子。 | NULL |
| CS_DESCRIBE_INPUT | 準備文の実行に必要な入力パラメータの記述をサーバから取得する。サーバが記述を送信すると、ct_results は *result_type* の値として CS_DESCRIBE_RESULT を返す。<br>アプリケーションは、ct_res_info と ct_describe、ct_dynsqlda、または ct_dyndesc を呼び出すことによって、この情報にアクセスできる。 | 準備文の識別子。 | NULL |
| CS_DESCRIBE_OUTPUT | 準備文が実行された場合に返される結果セットのロー・フォーマットの記述をサーバから取得する。サーバが記述を送信すると、ct_results は *result_type* の値として CS_DESCRIBE_RESULT を返す。<br>アプリケーションは、ct_res_info と ct_describe、ct_dynsqlda、または ct_dyndesc を呼び出すことによって、この情報にアクセスできる。 | 準備文の識別子。 | NULL |
| CS_EXECUTE | パラメータを必要としないか 1 個以上必要とする、準備された SQL 文を実行する。 | 準備文の識別子。 | NULL |
| CS_EXEC_IMMEDIATE | リテラル SQL 文を実行する。 | NULL | 実行する SQL 文。 |
| CS_PREPARE | SQL 文を準備する。 | 準備文の識別子。 | 準備する SQL 文。 |

- ct_dynamic は動的 Adaptive Server Enterprise コマンドを開始します。
- 動的 SQL コマンドの概要については、『Open Client Client-Library/C プログラマーズ・ガイド』の「第 8 章 動的 SQL コマンドの使い方」を参照してください。
- コマンドを開始するのは、コマンドをサーバに送信する最初の手順です。クライアント・アプリケーションがサーバ・コマンドを実行するために、Client-Library はコマンドを、サーバに送信できる記号コマンド・ストリームに変換する必要があります。コマンド・ストリームには、コマンドのタイプに関する情報と実行に必要なデータが含まれます。たとえば、動的 SQL 準備コマンドには、準備する文の文識別子とテキストが必要です。動的 SQL コマンドを実行する手順は、次のとおりです。

  a ct_dynamic を呼び出して、コマンドを開始します。このルーチンは、サーバに送信するコマンド・ストリームの構築に使用される内部構造体を設定します。

  b 必要であれば、コマンドにパラメータを渡します。ほとんどのアプリケーションは、コマンドに必要なパラメータごとに一度ずつ ct_param または ct_setparam を呼び出してパラメータを渡しますが、ct_dyndesc または ct_dynsqlda を使用して 1 つのコマンドのパラメータをまとめて渡すこともできます。

  c ct_send を呼び出して、コマンドをサーバに送信します。

  d ct_results を呼び出して、コマンド結果を処理します。

  準備文を実行する動的 SQL コマンドは、フェッチ可能な結果を返します。その他の動的 SQL コマンド・タイプは、フェッチ可能な結果を返しません。代わりに、コマンド・ステータス結果を返します。結果の処理については、「結果」(280 ページ ) を参照してください。
- ct_dynamic の使用には、次の規則が適用されます。
  - コマンド構造体が開始されるとき、アプリケーションでは、すでに開始されているコマンドを送信するかクリアしなければ、ct_command、ct_cursor、ct_dynamic、または ct_sendpassthru を使用して新しいコマンドを開始することはできません。
  - コマンドを送信した後、アプリケーションでは、そのコマンドの実行から返されたすべての結果を完全に処理するかキャンセルしなければ、同じコマンド構造体で新しいコマンドを開始することはできません。

    - カーソルを管理しているコマンド構造体に対して、アプリケーションで ct_dynamic を呼び出してコマンドを開始することはできません。先にカーソルの割り付けを解除するか、別のコマンド構造体を使用する必要があります。

- Client-Library では、アプリケーションは、前の実行からの結果を処理し終わった直後に ct_send を呼び出すことによって、コマンドを再送信できます。コマンドを再送信するには、アプリケーションは、ct_setparam で指定されたパラメータ・ソース変数の内容を更新してから、ct_send を呼び出します。次の動的 SQL コマンドは正常に再送信できます。
    - 即時実行コマンド
    - 準備文の中の実行コマンド
    - 出力記述または入力記述コマンド

  アプリケーションが他の動的 SQL コマンドを再送信すると、サーバ処理エラーが発生します。Client-Library では、アプリケーションは、新しいコマンドが ct_command、ct_cursor、ct_dynamic、または ct_sendpassthru で開始されていないかぎり、コマンドを再送信できます。

**文の準備**

- アプリケーションで文を準備するコマンドを開始するには、*type* に CS_PREPARE を指定し、*id* にユニークな文識別子を指定し、*buffer* に文のテキストを指定して ct_dynamic を呼び出します。
- 準備された SQL 文とは、サーバによってコンパイルされ、保管されている SQL 文です。それぞれの準備文はユニークな識別子に関連付けられます。
- アプリケーションはその数に制限なく文を準備できますが、準備文の識別子は接続内でユニークにしてください。
- 文の準備に使用するコマンド構造体は、その文の実行に使用するコマンド構造体と異なっていてもかまいませんが、両方のコマンド構造体は同じ接続に属していなければなりません。
- 準備文には、値の「**プレースホルダ**」が含まれている Transact-SQL も使用できます。プレースホルダは準備文の変数と同じように機能します。プレースホルダは、文中では疑問符 (?) で示されます。プレースホルダは次のようなロケーションに入れることができます。
    - insert 文の 1 つ以上の値の代わり
    - update 文の set 句
    - select 文または update 文の where 句

準備文を実行するコマンドを構築するとき、アプリケーションは、ct_param、ct_setparam、ct_dyndesc、または ct_dynsqlda を呼び出して、各動的パラメータ・マーカの値を代入します。

文が準備されていると、アプリケーションは、動的 SQL 入力記述コマンドをサーバに送信して、文を実行するのに必要な入力パラメータの記述を取得できます。

- ストアド・プロシージャを実行する文を準備するコマンドを開始するには、SQL テキストとして「exec sp_name」(「sp_name」は実行するストアド・プロシージャ名 ) を指定します。

```
ct_dynamic(cmd, CS_PREPARE, “myid”, CS_NULLTERM,
     “exec sp_2”, CS_NULLTERM);
```

- 文の準備が完了したら、アプリケーションは、その割り付けが解除されるまで繰り返し実行できます。

**準備文でのカーソル宣言**

- 準備文でカーソルを宣言するコマンドを開始するために、アプリケーションは *type* を CS_CURSOR_DECLARE にして ct_dynamic を呼び出します。
- 準備文でカーソルを宣言した後、アプリケーションは、ct_cursor(CS_CURSOR_OPTION) を呼び出して、カーソル宣言コマンドのオプション (「readonly」または「‘for update」) を設定できます。この手順は、カラムが更新可能であるかどうかを指定する for read only または for update of が select 文に含まれていない場合にだけ必要です。呼び出しのシーケンスは次のとおりです。
  - ct_dynamic(CS_CURSOR_DECLARE)
  - ct_cursor(CS_CURSOR_OPTION)
  - ct_send
  - ct_results ( 必要な回数だけ )
- ct_dynamic カーソル宣言コマンドは、後続の ct_cursor カーソル・ローまたはカーソル・オープン・コマンドとバッチにすることはできません。
- 準備文でカーソルを宣言した後、ct_cursor を使用してカーソルで追加のコマンドを開始します。
- アプリケーションは、準備文を実行する前に、準備文でカーソルを宣言する必要があります。

**準備文入力の記述の取得**

- アプリケーションは通常、最初に準備文を実行する前に準備文入力パラメータの記述を取得します。
- 準備文の入力の記述を取得するには、次の手順に従います。

  a　*type* を CS_DESCRIBE_INPUT にして ct_dynamic を呼び出して、記述を取得するコマンドを開始します。

  b　ct_send を呼び出して、サーバにコマンドを送信します。

  c　必要な回数だけ ct_results を呼び出して、コマンドの結果を処理します。CS_DESCRIBE_INPUT コマンドはタイプ CS_DESCRIBE_RESULT の結果セットを生成します。この結果セットには、フェッチ可能な結果は含まれず、各入力値の記述情報だけが含まれています。

  d　ct_res_info を呼び出して、入力値の数を取得します。ここでは、CS_DESCRIBE_RESULT が返されたものと想定します。次の手順も同じ想定です。

  e　各入力値について、ct_describe を呼び出します。

  この他に、アプリケーションは、ct_dyndesc または ct_dynsqlda を使用して記述を取得することもできます。ct_dyndesc は、入力値の数と各入力値のフォーマットを取得するまで数回呼び出す必要があります。ct_dynsqlda は一度の呼び出しで記述を取得できますが、アプリケーション管理の SQLDA 構造体が必要です。これらの方法については、次のセクションで説明します。

  - ct_dynsqlda を使用する方法については、「Sybase SQLDA：入力フォーマットの取得」(551 ページ ) を参照してください。
  - ct_dyndesc を使用する方法については、「ct_dyndesc によるコマンド入力または出力の記述の取得」(546 ページ ) を参照してください。

**準備文出力の記述の取得**

- アプリケーションは、一般に、初めて準備文を実行する前に準備文結果カラムの記述を取得します。

---

**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取得されます。

---

- 準備文の出力カラムの記述を取得するには、次の手順に従います。

  a *type* を CS_DESCRIBE_OUTPUT にして ct_dynamic を呼び出して、記述を取得するコマンドを開始します。

  b ct_send を呼び出して、サーバにコマンドを送信します。

  c 必要な回数だけ ct_results を呼び出して、コマンドの結果を処理します。ct_dynamic(CS_DESCRIBE_OUTPUT) コマンドは、タイプ CS_DESCRIBE_RESULT の結果セットを生成します。この結果セットには、フェッチ可能なデータは含まれず、各出力カラムの記述情報だけが含まれています。

  d ct_res_info を呼び出して、出力カラムの数を取得します。ここでは、CS_DESCRIBE_RESULT が返されたものと想定します。次の手順も同じ想定です。

  e 各出力カラムについて、ct_describe を呼び出します。

  この他に、アプリケーションは、ct_dyndesc または ct_dynsqlda を使用して記述を取得することもできます。ct_dyndesc は、カラムの数と各カラムのフォーマットを取得するまで数回呼び出す必要があります。ct_dynsqlda は一度の呼び出しで記述を取得できますが、アプリケーション管理の SQLDA 構造体が必要です。これらの方法については、次のセクションで説明します。

  - ct_dynsqlda を使用する方法については、「Sybase SQLDA : 出力フォーマットの取得」(552 ページ ) を参照してください。

  - ct_dyndesc を使用する方法については、「ct_dyndesc によるコマンド入力または出力の記述の取得」(546 ページ ) を参照してください。

### 準備文の実行

- 準備文を実行するには、次の手順に従います。

  a　*type* を CS_EXECUTE にして ct_dynamic を呼び出して、文を実行するコマンドを開始します。

  b　SQL 文の入力値を定義します。次の手順で行います。

    - 各パラメータについて ct_param を一度呼び出す方法 － 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 を一度呼び出す方法 － 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 によるパラメータ値の受け渡し」(547 ページ ) を参照してください。
      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。「Sybase SQLDA：コマンド入力パラメータの受け渡し」(553 ページ ) を参照してください。
      ct_dynsqlda(CS_SQLDA_PARAM) は内部で ct_param を呼び出します。

  c　ct_send を呼び出して、サーバにコマンドを送信します。

  d　必要な回数だけ ct_results を呼び出して、コマンドの結果を処理します。

### リテラル文の実行

- 次の基準を満たす場合、動的 SQL 文をただちに実行できます。
  - 動的 SQL 文がデータを返さない (select 文ではない )。
  - 動的 SQL 文に、パラメータのためのプレースホルダが含まれていない。プレースホルダは文のテキスト内では疑問符 (?) で示されます。

- 動的パラメータ・マーカは、ランタイムに SQL 文に代入される実データをユーザが指定できるプレースホルダとして機能します。

- リテラル文を実行するには、次の手順に従います。

  a *type* を CS_EXEC_IMMEDIATE に、*id* を NULL に、*buffer* を実行する文にして ct_dynamic を呼び出します。

  b ct_send を呼び出して、サーバにコマンドを送信します。

  c 必要な回数だけ ct_results を呼び出して、コマンドの結果を処理します。

**準備文の割り付け解除**

- 準備文を割り付け解除するコマンドを開始するために、アプリケーションは *type* を CS_DEALLOC に、*id* を文識別子にして ct_dynamic を呼び出します。

**参照** ct_dyndesc、ct_dynsqlda、ct_param、ct_setparam、ct_send、ct_cursor

# ct_dyndesc

**説明** 動的 SQL 記述子領域でオペレーションを実行します。

**構文**

```
CS_RETCODE ct_dyndesc(cmd, descriptor, desclen, operation, index,
                      datafmt, buffer, buflen, copied, indicator)

CS_COMMAND       *cmd;
CS_CHAR          *descriptor;
CS_INT           desclen;
CS_INT           operation;
CS_INT           index;
CS_DATAFMT       *datafmt;
CS_VOID          *buffer;
CS_INT           buflen;
CS_INT           *copied;
CS_SMALLINT      *indicator;
```

**パラメータ**

*cmd*

CS_COMMAND 構造体を指すポインタです。記述子が割り付けられる同じコンテキスト内のすべての CS_COMMAND は、記述子の操作に使用できます。

*descriptor*

記述子名を指すポインタです。記述子名はコンテキスト内でユニークにしてください。

*desclen*

**descriptor* のバイト単位の長さです。*descriptor が null で終了する場合、*desclen* に CS_NULLTERM を指定して渡してください。

*operation*

開始する記述子オペレーションです。次の表に、*operation* の値の一覧を示します。

**表 3-25：ct_dyndesc operation パラメータの値**

| operation の値 | 結果 |
|---|---|
| CS_ALLOC | 記述子を割り付ける。 |
| CS_DEALLOC | 記述子の割り付けを解除する。 |
| CS_GETATTR | パラメータまたは結果項目の属性を取得する。 |
| CS_GETCNT | パラメータ数またはカラム数を取得する。 |
| CS_SETATTR | パラメータの属性を設定する。 |
| CS_SETCNT | パラメータ数またはカラム数を設定する。 |
| CS_USE_DESC | 記述子を文またはコマンド構造体に関連付ける。 |

*index*

使用される場合は、整数変数です。

*index* は、*operation* の値に応じて、記述子項目の 1 から開始されるインデックス、または記述子に関連する項目の数のどちらかになります。

*datafmt*

使用される場合は、CS_DATAFMT 構造体を指すポインタです。

*buffer*

使用される場合は、データ領域を指すポインタです。

*buflen*

使用される場合は、**buffer* データのバイト単位の長さです。

*copied*

使用される場合は、整数変数を指すポインタです。ct_dyndesc は、**copied* を **buffer* に入れるデータのバイト単位の長さに設定します。

*indicator*

使用される場合は、インジケータ変数を指すポインタです。

**表 3-26：ct_dyndesc indicator パラメータの値**

| operation の値 | *indicator の値 | 意味 |
|---|---|---|
| CS_GETATTR | -1 | Client-Library によるサーバ値のトランケート。 |
| | 0 | トランケートなし。 |
| | 整数値 | サーバによるアプリケーション値のトランケート。 |
| CS_SETATTR | -1 | パラメータは、null 値を持つ。 |

**戻り値**

ct_dyndesc は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_ROW_FAIL | リカバリ可能なエラーが発生している。リカバリ可能なエラーとは、メモリ割り付けエラーや、値をプログラム変数にコピーするときに発生する変換エラーなど。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

**使用法**

- 動的 SQL 記述子領域には、動的 SQL 文の入力パラメータまたは動的 SQL 文の実行によって生成された結果データ項目についての情報が格納されます。
- ct_dyndesc はパラメータとして CS_COMMAND 構造体を取りますが、動的 SQL の記述子領域の範囲は Client-Library のコンテキストです。言い換えると、次のようになります。
  - 記述子名はコンテキスト内でユニークにしてください。
  - アプリケーションは、コンテキスト内の任意のコマンド構造体を使用して、そのコンテキストの記述子領域を参照できます。たとえば、あるコマンド構造体を通じて割り付けられた記述子領域は、同じコンテキスト内の別のコマンド構造体によって割り付けを解除できます。
- 『Open Client Client-Library/C プログラマーズ・ガイド』の「第 8 章 動的 SQL コマンドの使い方」を参照してください。

**記述子の割り付け**

- 記述子を割り付けるために、アプリケーションは *operation* を CS_ALLOC にして ct_dyndesc を呼び出します。
- 表 3-27 に、CS_ALLOC オペレーションのパラメータの値を示します。

**表 3-27：ct_dyndesc(CS_ALLOC) オペレーションのパラメータの値**

| descriptor、desclen | index | datafmt | buffer、buflen | copied | indicator |
|---|---|---|---|---|---|
| 割り付ける記述子の名前、その名前の長さまたは CS_NULLTERM の長さ。 | 記述子が取り込む項目の最大数。 | NULL | NULL、CS_UNUSED | NULL | NULL |

**記述子の割り付け解除**

- 記述子の割り付けを解除するために、アプリケーションは *operation* を CS_DEALLOC にして ct_dyndesc を呼び出します。
- 表 3-28 に、CS_DEALLOC オペレーションのパラメータの値を示します。

**表 3-28：ct_dyndesc(CS_DEALLOC) オペレーションのパラメータの値**

| descriptor、desclen | index | datafmt | buffer、buflen | copied | indicator |
|---|---|---|---|---|---|
| 割り付けを解除する記述子の名前、その名前の長さまたは CS_NULLTERM の長さ。 | CS_UNUSED | NULL | NULL、CS_UNUSED | NULL | NULL |

**パラメータまたは結果項目の属性の取得**

- パラメータ属性または結果データ項目の属性を取得するために、アプリケーションは *operation* を CS_GETATTR にして ct_dyndesc を呼び出します。
- 表 3-29 に、CS_GETATTR オペレーションのパラメータの値を示します。

**表 3-29：ct_dyndesc(CS_GETATTR) オペレーションのパラメータの値**

| descriptor、desclen | index | datafmt | buffer、buflen | copied | indicator |
|---|---|---|---|---|---|
| 対象となる記述子の名前、その名前の長さまたは CS_NULLTERM の長さ。 | 記述子を必要とする項目数。<br>インデックス番号は 1 から開始される。 | 入力パラメータとして、**datafmt* が **buffer* を記述する。<br>ct_dyndesc は **datafmt* を項目の記述子で上書きする。 | 指定されている場合、**buffer* には項目の値が設定される。<br>*buffer* が NULL の場合、項目の記述子だけが返される。<br>*buflen* は CS_UNUSED<br>*datafmt->maxlength* は、**buffer* の長さを記述する。 | 指定されている場合、**copied* には **buffer* に入れられたバイト数が設定される。<br>NULL の場合もある。 | 指定されている場合、**indicator* には項目の indicator の値が設定される。<br>NULL の場合もある。 |

- ct_dyndesc は、必要に応じて、**datafmt* に指定されているフォーマットにカラムのソース・データを変換し、結果を **buffer* に格納します。**indicator* と **copied* にポインタが入力されている場合、ポインタはそれに応じて設定されます。
- アプリケーションは、CS_GETATTR オペレーションの **datafmt* フィールドを、ct_bind の呼び出しのときの設定とまったく同じにする必要があります。ただし、*datafmt–>count* の値だけは 0 または 1 に設定します ( 一度に取得できるカラム値は 1 つだけです)。表 3-30 に、使用される CS_DATAFMT の各フィールドを示します。

**表 3-30：ct_dyndesc(CS_GETATTR) オペレーションで設定される CS_DATAFMT**

| フィールド名 | 設定値 |
|---|---|
| *datatype* | *buffer* 変数のデータ型。 |
| *format* | フォーマット記号のビットマスク。 |
| *maxlength* | *buffer* データ領域の長さ。 |
| *scale* | *buffer* が数値または 10 進変数である場合、小数点の右側に表示できる最大桁数。その他のデータ型の場合、*scale* は無視される。 |
| *precision* | *buffer* が数値または 10 進変数である場合、表示できる最大 10 進桁数。その他のデータ型の場合、*precision* は無視される。 |
| *count* | 0 または 1。 |
| *locale* | 有効な CS_LOCALE 構造体を指すポインタまたは NULL。 |

他のすべてのフィールドは無視される。

- ct_dyndesc(CS_GETATTR) では、ct_describe の場合とまったく同じように **datafmt* フィールドが設定されます。表 3-31 に、ct_dyndesc で設定される **datafmt* のフィールドを示します。

**表 3-31：ct_dyndesc(CS_GETATTR) オペレーション中の CS_DATAFMT フィールドの設定**

| フィールド名 | ct_dyndesc が設定するフィールド |
|---|---|
| *name* | 指定されている場合、NULL で終了するデータ項目の名前。*namelen* が 0 の場合は NULL。 |
| *namelen* | NULL ターミネータを含まない名前の実際の長さ。<br>*name* が NULL の場合は 0。 |
| *datatype* | 項目のデータ型。「データ型のサポート」(339 ページ ) で示しているすべてのデータ型 ( ただし、CS_VARCHAR と CS_VARBINARY を除く ) が有効。 |
| *maxlength* | カラムまたはパラメータのデータの可能最大長。 |
| *scale* | 結果データ項目の小数点以下の最大桁数。 |
| *precision* | 結果データ項目に表示できる最大 10 進桁数。 |
| *status* | 次の記号のビットマスク。<br>• そのカラムが NULL 値を持てることを示す CS_CANBENULL。<br>• そのカラムが公開されている「隠しカラム」であることを示す CS_HIDDEN。隠しカラムについては、「隠しキー」(250 ページ ) を参照。<br>• カラムが identity カラムであることを示す CS_IDENTITY。<br>• **カラム**がテーブルのキーの一部であることを示す CS_KEY。<br>• そのカラムがローのバージョン・キーの一部であることを示す CS_VERSION_KEY。<br>• そのカラムがタイムスタンプ・カラムであることを示す CS_TIMESTAMP。<br>• そのカラムが更新可能カーソル・カラムであることを示す CS_UPDATABLE。<br>• カラムがカーソル宣言コマンドの更新句にあることを示す CS_UPDATECOL。<br>• そのカラムが RPC コマンドのリターン・パラメータであることを示す CS_RETURN。 |
| *count* | ct_dyndesc は、*count* を 1 に設定する。 |
| *usertype* | 指定されている場合には、カラムまたはパラメータの Adaptive Server Enterprise ユーザ定義データ型。usertype は、datatype に加えて ( 代わりではなく ) 設定される。 |
| *locale* | データのロケール情報を持つ CS_LOCALE 構造体を指すポインタ。<br>このポインタは NULL でもかまわない。 |

**パラメータ数またはカラム数の取得**

- 記述子が記述できるパラメータまたは結果項目の数を取得するために、アプリケーションは *operation* を CS_GETCNT にして ct_dyndesc を呼び出します。
- ct_dyndesc は、入力パラメータと出力カラムのどちらが記述されているかに応じて、**buffer* に動的パラメータの指定数、または動的 SQL 文の select リストのカラム数を設定します。
- 次の表に、CS_GETCNT オペレーションのパラメータの値を示します。

**表 3-32：ct_dyndesc(CS_GETCNT) オペレーションのパラメータの値**

| descriptor、desclen | index | datafmt | buffer、buflen | copied | indicator |
|---|---|---|---|---|---|
| 対象となる記述子の名前、その名前の長さまたは CS_NULLTERM の長さ。 | CS_UNUSED | NULL | CS_INT を指すポインタ、CS_UNUSED を指すポインタ。 | 指定されている場合、**copied* には **buffer* に入れられたバイト数が設定される。<br>NULL の場合もある。 | NULL |

**パラメータの属性の設定**

- パラメータ属性を設定するために、アプリケーションは *operation* を CS_SETATTR にして ct_dyndesc を呼び出します。
- 表 3-33 に、CS_SETATTR オペレーションのパラメータの値を示します。

**表 3-33：ct_dyndesc(CS_SETATTR) オペレーションのパラメータの値**

| descriptor、desclen | index | datafmt | buffer、buflen | copied | indicator |
|---|---|---|---|---|---|
| 対象となる記述子の名前、その名前の長さまたは CS_NULLTERM の長さ。 | 記述が設定されている項目数。<br>インデックス番号は 1 から開始される。 | **datafmt* は項目の記述を含んでいる。 | 項目の値を指すポインタ、その値の長さ。<br>*buffer* が固定長型を指している場合、*buflen* を CS_UNUSED として渡すこと。 | NULL | 指定された場合、**indicator* は項目の indicator の値。<br>**indicator* が -1 の場合、*buffer* が無視され、項目の値には NULL が設定される。<br>*indicator* は NULL でもかまわない。 |

- アプリケーションでは、CS_SETATTR オペレーションの **datafmt* フィールドの設定を、ct_param の呼び出しのときの設定とまったく同じにする必要があります。表 3-34 に、使用される各フィールドを示します。

**表 3-34：ct_dyndesc(CS_SETATTR) オペレーションの CS_DATAFMT フィールド**

| フィールド名 | 設定値 |
|---|---|
| *name* | パラメータの名前。 |
| *namelen* | その名前の長さまたは CS_NULLTERM の長さ。 |
| *datatype* | 設定されている項目のデータ型。 |
| *maxlength* | 可変長リターン・パラメータに対して、*maxlength* はこのパラメータへ返される最大のバイト数。<br>*status* が CS_INPUTVALUE の場合または *datatype* が固定長型を表している場合、*maxlength* は無視される。 |
| *status* | CS_INPUTVALUE、CS_UPDATECOL、または CS_RETURN。<br>CS_UPDATECOL はカーソル宣言コマンドの更新カラムを示す。<br>CS_RETURN はリターン・パラメータを示す。 |
| *locale* | 有効な CS_LOCALE 構造体を指すポインタまたは NULL。 |
| 他のすべてのフィールドは無視される。 | |

#### パラメータ数またはカラム数の設定

- 記述子が記述できるパラメータまたはカラムの数を設定するために、アプリケーションは *operation* を CS_SETCNT にして ct_dyndesc を呼び出します。
- 表 3-35 に、CS_SETCNT オペレーションのパラメータの値を示します。

**表 3-35：ct_dyndesc(CS_SETCNT) オペレーションのパラメータの値**

| descriptor、desclen | index | datafmt | buffer、buflen | copied | indicator |
|---|---|---|---|---|---|
| 対象となる記述子の名前、その名前の長さまたは CS_NULLTERM の長さ。 | 新しい記述子のカウント。 | NULL | NULL、CS_UNUSED | NULL | NULL |

#### 文またはコマンド構造体への記述子の関連付け

- 記述子を準備文またはコマンド構造体に関連付けるために、アプリケーションは *operation* を CS_USE_DESC に設定して ct_dyndesc を呼び出します。
- 表 3-36 に、CS_USE_DESC オペレーションのパラメータの値を示します。

**表 3-36：ct_dyndesc(CS_USE_DESC) オペレーションのパラメータの値**

| descriptor、desclen | index | datafmt | buffer、buflen | copied | indicator |
|---|---|---|---|---|---|
| 対象となる記述子の名前、その名前の長さまたは CS_NULLTERM の長さ。 | CS_UNUSED | NULL | NULL、CS_UNUSED | NULL | NULL |

- 記述子領域は、通常、コンテキスト構造体に関連付けられます。ただし、記述子領域をカーソルに対する入出力の記述に使用する場合は、まず、オープンしたコマンド構造体にカーソルを関連付けてください。
- カーソル入力を記述するために記述子を使用する場合、アプリケーションの一般的な呼び出しシーケンスは次のとおりです。

```
ct_dyndesc(CS_ALLOC)
 ct_dyndesc(CS_SETCNT)
 for each input value:
     ct_dyndesc(CS_SETATTR)
 end for
 ct_cursor to open the cursor
 ct_dyndesc(CS_USE_DESC)
 ct_send
```

***ct_dyndesc* によるコマンド入力または出力の記述の取得**

- ct_dyndesc で準備文の入力パラメータまたは結果カラムの記述を取得する呼び出しシーケンスは、次のとおりです。

  a　*operation* を CS_ALLOC にして ct_dyndesc を呼び出し、記述子エリアを割り付けます。

  b　ct_dynamic を呼び出して、記述を取得するコマンドを開始します。ct_dynamic の *type* 引数を、入力記述の場合は CS_DESCRIBE_INPUT、出力記述の場合は CS_DESCRIBE_OUTPUT で渡します。

  c　ct_send を呼び出して、サーバにコマンドを送信します。

  d　必要な回数だけ ct_results を呼び出して、コマンドの結果を処理します。記述コマンドはタイプ CS_DESCRIBE_RESULT の結果セットを生成します。この結果セットには、フェッチ可能な結果は含まれず、各入力値の記述情報だけが含まれています。

e　*operation* に CS_USE_DESC を指定して ct_dyndesc を呼び出し、準備文を手順 1 で割り付けた記述子領域と関連付けます。ここでは、ct_results の現在の *result_type* 値として CS_DESCRIBE_RESULT が返されたものと想定します。次の 2 つの手順も同じ想定です。

f　*operation* を CS_GETCNT にして ct_dyndesc を呼び出し、パラメータまたはカラムの数を取得します。

g　各パラメータまたはカラムについて、*operation* に CS_GETATTR を指定して ct_dyndesc を呼び出し、値の記述を取得します。

### *ct_dyndesc* によるパラメータ値の受け渡し

- 準備された動的 SQL 文を実行するとき、アプリケーションは ct_dyndesc を使用して入力パラメータ値を入力できます。呼び出しのシーケンスは次のとおりです。

  a　ct_dynamic(CS_EXECUTE) を呼び出して、コマンドを開始します。

  b　必要な各入力パラメータについて、*operation* を CS_SETATTR にして ct_dyndesc を呼び出し、記述子エリアにパラメータ値を入れます。必要であれば、まず cs_convert で値を変換します。CS_SETATTR の使用法については、「パラメータの属性の設定」(544 ページ ) で説明しています。

  c　*operation* を CS_USE_DESC にして ct_dyndesc を呼び出し、パラメータ値をコマンドに適用します。

  d　ct_send を呼び出して、サーバにコマンドを送信します。

  e　コマンドの結果を処理します。Client-Library の結果モデルについては、「結果」(280 ページ ) を参照してください。

- Client-Library では、アプリケーションは前の実行の結果を処理し終わった直後に ct_send を呼び出すことによって、動的 SQL 実行コマンドを再送信できます。ただし、ct_dyndesc を使用してパラメータ値を入力する場合、コマンドのすべての実行について、パラメータ値は一定です。別のパラメータ値を使用してコマンドを再送信するアプリケーションでは、ct_setparam を使用してください。ct_setparam および「コマンドの再送信」(643 ページ ) を参照してください。

### *ct_dyndesc* による結果カラム値の取得

- フェッチ可能な結果を処理するとき、アプリケーションは ct_dyndesc と ct_fetch を使用して結果カラム値を取得できます。( フェッチ可能な結果は ct_results *result_type* パラメータの値で示されます )。

- 呼び出しのシーケンスは次のとおりです。このシーケンスでは、ct_results が、フェッチ可能なデータを示す *result_type* 値を返しているものと想定します。

  a　*operation* を CS_USE_DESC にして ct_dyndesc を呼び出し、記述子を結果ローと対応させます。

  b　ct_fetch を呼び出して、結果ローをフェッチします。ct_fetch が CS_END_DATA を返した場合、すべてのローが取得されたことになります。

  c　結果セットの各カラムについて、*operation* に CS_GETATTR を指定して ct_dyndesc を呼び出し、カラムの値を取得します。CS_GETATTR の使用法については、「パラメータまたは結果項目の属性の取得」(541 ページ ) で説明しています。

  d　ct_fetch が CS_END_DATA を返すまで、前述の手順 2 ～ 4 を繰り返します。

**参照**

ct_bind、ct_cursor、ct_describe、ct_dynamic、ct_dynsqlda、ct_fetch、ct_param

# ct_dynsqlda

**説明**

SQLDA 構造体に関する処理を行います。

**構文**

```
CS_RETCODE ct_dynsqlda(cmd, sqlda_type, dap, operation)

CS_COMMAND   *cmd;
CS_INT       sqlda_type;
SQLDA        *dap;
CS_INT       operation;
```

**パラメータ**

*cmd*

CS_COMMAND 構造体を指すポインタです。

*sqlda_type*

*dap* が指す SQLDA 構造体のタイプを示す記号定数です。このリリースでは、*sqlda_type* は Sybase 形式の SQLDA 構造体を示す CS_SQLDA_SYBASE である必要があります。

*dap*

SQLDA 構造体のアドレスです。SQLDA 構造体は Sybase の *sqlda.h* ヘッダ・ファイル内で定義されます。この構造体の定義については、「Sybase 形式の SQLDA 構造体」(549 ページ ) を参照してください。

*operation*

実行するオペレーションです。表 3-37 に、ct_dynsqlda の使用法についてまとめます。

**表 3-37：ct_dynsqlda operation パラメータの値**

| operation の値 | 機能 |
|---|---|
| CS_GET_IN | 準備された動的 SQL 文の入力パラメータの記述を **dap* に入れる。 |
| CS_GET_OUT | 準備された動的 SQL 文によって返されるカラムの記述を **dap* に入れる。 |
| CS_SQLDA_PARAM | SQLDA 構造体は、準備文の実行用の入力パラメータを与えるために使用する。<br>準備された動的 SQL 文を実行する場合は、このオペレーションは入力パラメータとしての **dap* の内容に適用される。 |
| CS_SQLDA_BIND | SQLDA 構造体は、準備文の実行による結果を処理するために使用する。<br>準備された動的 SQL 文の実行によって返される結果を処理する場合、このオペレーションは **dap* の内容を結果カラムにバインドする。 |

**戻り値**

ct_dynsqlda は次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

**使用法**

- SQLDA 構造体は、準備された動的 SQL 文で使用されます。この構造体は、フォーマット記述子、および ( オプションとして ) コマンド入力パラメータ値または結果カラム値を保持します。
- 『Open Client Client-Library/C プログラマーズ・ガイド』の「第 8 章 動的 SQL コマンドの使い方」を参照してください。
- ct_dynsqlda は SQLDA 構造体を管理します。SQLDA 構造体には、記述のデータ領域、およびコマンドの入力パラメータの値または結果値が格納されます。

Sybase 形式の SQLDA 構造体

- Sybase 形式の SQLDA は、次のように宣言される自己記述型の可変長構造体です。

```
typedef struct _sqlda
 {
     CS_SMALLINT sd_sqln; /* Actual length of column array    */
     CS_SMALLINT sd_sqld; /* Current number of columns        */
     /*
     ** The following array is treated as if it were the length
     ** indicated by sd_sqln.
     */
     struct _sd_column
     {
         CS_DATAFMT sd_datafmt; /* Format of column i.       */
         CS_VOID *sd_sqldata;   /* Value buffer for column i.*/
         CS_INT sd_sqllen;      /* Length of current value.  */
         CS_SMALLINT sd_sqlind; /* Indicator for column i.   */
         CS_VOID *sd_sqlmore;   /* Reserved for future use.  */
     } sd_column[1];
 } sqlda;

 #define SYB_SQLDA_SIZE(n) (sizeof(sqlda) ¥
         - sizeof(struct _sd_column) ¥
                 + (n) * sizeof(struct _sd_column))
```

SQLDA 構造体の割り付け

- アプリケーションは、*dap* が指す構造体の割り付けと初期設定を正しく行う必要があります。構造体の実際のサイズは、構造体が記述するカラムの数によって異なります。アプリケーションは、SYB_SQLDA_SIZE マクロを使用して適切なサイズの SQLDA バッファを割り付けることができます。メモリを割り付けるために malloc を使用するシステムでは、次のように行います。

```
#define MAX_COLUMNS 16
 SQLDA *dap;

 dap = (SQLDA *) malloc(
                  SYB_SQLDA_SIZE(MAX_COLUMNS) );
 if (dap == (SQLDA *) NULL)
   ... out of memory ...

 memset((void *)dap, 0,
        SYB_SQLDA_SIZE(MAX_COLUMNS) );
 dap->sd_sqln = MAX_COLUMNS;
```

アプリケーションは、SQLDA_DECL マクロを呼び出して、静的な SQLDA 構造体を宣言できます。次のように呼び出します。

```
SQLDA_DECL(name, size);
```

この呼び出しは次の宣言と同じです。

```
struct {
        CS_SMALLINT sd_sqln;
        CS_SMALLINT sd_sqld;
        struct {
                CS_DATAFMT sd_datafmt;
                CS_VOID *sd_sqldata;
                CS_SMALLINT sd_sqlind;
                CS_INT sd_sqllen;
                CS_VOID *sd_sqlmore;
        } sd_column[(size)];
 } name;
```

- ct_dynsqlda の CS_SQLDA_PARAM オペレーションまたは CS_SQLDA_BIND オペレーションを使用して ) 入力パラメータを渡すか結果を取り出すために構造体を使用する場合、アプリケーションは、項目の値を入れるためのバッファを割り付けて、構造体にバッファのサイズを設定する必要もあります。
- Sybase 形式の SQLDA の使用法については、次の項で説明します。

**Sybase SQLDA：入力フォーマットの取得**

- ct_dynsqlda(*cmd*, CS_SQLDA_SYBASE, &*sqlda*, CS_GET_IN) は、準備文を実行するために必要な入力パラメータの記述を *sqlda* のフィールドに入れます。
- 準備された動的 SQL 文は、実行時に供給される値のためのパラメータ・マーカを持つことができます。
- 動的 SQL 文には、実行時に提供されるパラメータ用のパラメータ・マーカを含めることができます。動的 SQL 文が準備された後は、アプリケーションは文に使用するパラメータのフォーマットの記述を要求できます。この手順を次に示します。

  a　ct_dynamic(CS_DESCRIBE_INPUT) コマンドを構築してサーバに送ります。

  b　ct_results を使用してコマンドの結果を処理します。ct_results が、CS_DESCRIBE_RESULT になっている *result_type* 値を返す場合は、パラメータのフォーマットを使用できます。

  c　必要に応じて ct_res_info(CS_NUMDATA) を呼び出し、文が必要とするパラメータの数を調べます。SQLDA 構造体はカラム記述子の配列を保持します。配列には、必要な各パラメータのエントリが少なくとも 1 つ含まれている必要があります。

d ct_dynsqlda を呼び出してパラメータのフォーマットを取得します。

- アプリケーションは、SQLDA 構造体とその構成ポインタが指すメモリを割り付ける必要があります。CS_GET_IN オペレーションのためのフィールド設定は次のとおりです。

**表 3-38：ct_dynsqlda(CS_GET_IN) 呼び出しのための SQLDA フィールド**

| フィールド | 説明 |
|---|---|
| *sqlda->sd_sqln* | 入力に関して、*sqlda->sd_column* から始まる配列内の要素の数。SQLDA には、このための十分な大きさが必要。「SQLDA 構造体の割り付け」(550 ページ ) を参照。 |
| *sqlda->sd_sqld* | 出力に関して、実際の項目数。 |
| *sqlda->sd_column*[*i*].*sd_sqldata* | 未使用 ( 無視される )。 |
| *sqlda->sd_column*[*i*].*sd_sqllen* | 未使用 ( 無視される )。 |
| *sqlda->sd_column*[*i*].*sd_datafmt* | 出力に関して、各パラメータの CS_DATAFMT フィールドは ct_describe による設定とまったく同じに設定される ( 表 3-18 ( 503 ページ ) を参照 )。 |
| *sqlda->sd_column*[*i*].*sd_sqlind* | 未使用 ( 無視される )。 |

Sybase SQLDA：出力フォーマットの取得

- ct_dynsqlda(*cmd*, CS_SQLDA_SYBASE, &*sqlda*, CS_GET_OUT) は、準備文の実行によって返される結果の記述を *sqlda* のフィールドに入れます。
- 動的 SQL 文には、サーバの select コマンドを含めることができます。動的 SQL 文が準備されると、アプリケーションはその文によって返されるロー・データのフォーマットの記述を要求できます。この手順を次に示します。

a ct_dynamic(CS_DESCRIBE_OUTPUT) コマンドを構築してサーバに送ります。

b ct_results を使用してコマンドの結果を処理します。ct_results が CS_DESCRIBE_RESULT になっている *result_type* 値を返す場合は、アプリケーションは出力フォーマットを使用できます。

c　必要に応じて ct_res_info(CS_NUMDATA) を呼び出し、文が返すカラムの数を調べます。SQLDA 構造体はカラム記述子の配列のアドレスを保持します。この配列は、各カラムについて少なくとも 1 つのエントリを持つ必要があります。

d　ct_dynsqlda を呼び出してカラムのフォーマットを取得します。

- アプリケーションは、SQLDA 構造体とその構成ポインタが指すメモリを割り付ける必要があります。CS_GET_OUT オペレーションのためのフィールド設定は次のとおりです。

**表 3-39：ct_dynsqlda(CS_GET_OUT) 呼び出しのための SQLDA フィールド**

| フィールド | 説明 |
| --- | --- |
| *sqlda->sd_sqln* | 入力に関して、*sqlda->sd_column* から始まる配列内の要素の数。SQLDA には、このための十分な大きさが必要。「SQLDA 構造体の割り付け」(550 ページ) を参照。 |
| *sqlda->sd_sqld* | 出力に関して、実際の項目数。 |
| *sqlda->sd_column*[*i*].*sd_sqldata* | 未使用 ( 無視される )。 |
| *sqlda->sd_column*[*i*].*sd_sqllen* | 未使用 ( 無視される )。 |
| *sqlda->sd_column*[*i*].*sd_datafmt* | 出力に関して、各カラムの CS_DATAFMT フィールドは ct_describe による設定とまったく同じに設定される ( 表 3-18 ( 503 ページ ) を参照 )。 |
| *sqlda->sd_column*[*i*].*sd_sqlind* | 未使用 ( 無視される )。 |

Sybase SQLDA：**コマンド入力パラメータの受け渡し**

- ct_dynsqlda(*cmd*, CS_SQLDA_SYBASE, &*sqlda*, CS_SQLDA_PARAM) は、準備文の実行用の入力パラメータ値として SQLDA 構造体の内容に適用されます。
- ct_dynsqlda を使用して、準備文の実行用のパラメータを渡す手順は次のとおりです。

a　必要に応じて、「Sybase SQLDA：入力フォーマットの取得」(551 ページ ) の説明に従ってコマンド入力の記述を取得します。

b　ct_dynamic(CS_EXECUTE) を呼び出して、コマンドを開始します。

c　後述の表の説明に従って SQLDA のフィールドに入力します。

d　ct_dynsqlda(CS_SQLDA_PARAM) を呼び出して、SQLDA の内容を入力パラメータ値として適用します。

e　ct_send を使用してコマンドを送信します。

f　コマンドの結果を処理します。

- アプリケーションは、SQLDA 構造体とその構成ポインタが指すメモリを割り付ける必要があります。CS_SQLDA_PARAM オペレーションのためのフィールド設定は次のとおりです。

**表 3-40：ct_dynsqlda(CS_SQLDA_PARAM) 呼び出しのための SQLDA フィールド**

| フィールド | 説明 |
|---|---|
| *sqlda->* *sd_sqln* | 入力に関して、*sqlda->sd_column* から始まる配列内の要素の数。配列の項目数は、*sd_sqld* フィールドによって要求される項目数と同数である必要がある。「SQLDA 構造体の割り付け」(550 ページ ) を参照。 |
| *sqlda->* *sd_sqld* | 入力に関して、パラメータ値として適用される *sqlda->* *sd_column* 配列内の項目の数。 |
| *sqlda->* *sd_column*[*i*]. *sd_sqldata* | コマンドを実行する場合、( 文の最初のパラメータ・マーカを 0 とする ) パラメータ *i* の値を含んでいるバッファのアドレスを含む。 |
| *sqlda->* *sd_column*[*i*]. *sd_sqllen* | *sd_column*[*i*].*sd_sqldata* が指すバッファのバイト単位の長さ。 |
| *sqlda->* *sd_column*[*i*]. *sd_datafmt* | 各カラムの CS_DATAFMT フィールドは、ct_param で必要となる設定とまったく同じにする必要がある ( 表 3-49 (600 ページ ) を参照 )。 |
| *sqlda->* *sd_column*[*i*]-> *sd_sqlind* | コマンドを実行する場合は、-1 という値はパラメータ *i* の値が NULL であることを示す。 |

Sybase SQLDA：**結果の取得**

- ct_dynsqlda(*cmd*, CS_SQLDA_SYBASE, &*sqlda*, CS_SQLDA_BIND) は、準備文の実行によって返される結果内のカラムに SQLDA 構造体の内容をバインドします。
- ct_dynsqlda を使用して結果を処理する手順は次のとおりです。

a　必要に応じて、「Sybase SQLDA：出力フォーマットの取得」(552 ページ ) の説明に従ってコマンド出力の記述を取得します。

b　ct_dynamic(CS_EXECUTE) を呼び出して、コマンドを開始します。

c　実行に必要なパラメータ値をすべて指定します。

d　ct_send を使用してコマンドを送信します。

e　コマンドの結果を処理します。ct_results が、CS_ROW_RESULT になっている *result_type* 値を返す場合、SQLDA 構造体は結果のローにバインドできます。

f　次の表の説明に従って SQLDA のフィールドに入力して、ct_dynsqlda(CS_SQLDA_BIND) を呼び出して結果のロー内のカラム値にバインドします。

g　ct_fetch を使用してローを処理します。ct_fetch に対する呼び出しはそれぞれ、必要に応じて変換された値を SQLDA のバインドされたフィールドに入れます。

- アプリケーションは、SQLDA 構造体とその構成ポインタが指すメモリを割り付ける必要があります。CS_SQLDA_BIND オペレーションのためのフィールド設定は次のとおりです。

表 3-41 : ct_dynsqlda(CS_SQLDA_BIND) 呼び出しのための SQLDA フィールド

| フィールド | 説明 |
|---|---|
| *sqlda->sd_sqln* | 入力に関して、*sqlda->sd_column* から始まる配列内の要素の数。配列には、*sd_sqld* フィールドによって要求される項目の数と同等以上の長さが必要である。「SQLDA 構造体の割り付け」(550 ページ) を参照。 |
| *sqlda->sd_sqld* | 入力に関して、結果カラムにバインドされる必要のある *sqlda->sd_column* 配列内の項目の数。 |
| *sqlda->sd_column*[*i*].*sd_sqldata* | ct_fetch が ( 最初のカラムを 0 とする ) カラム *i* のための値を入れるバッファのアドレスを含む。 |
| *sqlda->sd_column*[*i*].*sd_sqllen* | *sd_column*[*i*]->*sd_sqldata* が指すバッファのバイト単位の長さ。 |
| *sqlda->sd_column*[*i*].*sd_datafmt* | 各カラム用の CS_DATAFMT フィールドは、ct_bind で必要となる設定とまったく同じにする必要がある ( 表 3-1 (373 ページ ) を参照 )。 |
| *sqlda->sd_column*[*i*].*sd_sqlind* | ct_fetch に対する以後の呼び出しは、各カラムに対してインジケータ値を書き込む。インジケータ値は次のとおり。<br>• -1 はカラム値が NULL であることを示す。<br>• 0 はフェッチが成功したことを示す。<br>• 正の整数の場合は、トランケーションを示す。この値はトランケーション前のカラム値の実際の長さを示す。 |

**参照**

ct_bind、ct_cursor、ct_describe、ct_dynamic、ct_dyndesc、ct_fetch、ct_param、ct_res_info

# ct_exit

**説明**

Client-Library を終了します。

**構文**

```
CS_RETCODE ct_exit(context, option)

CS_CONTEXT    *context;
CS_INT        option;
```

**パラメータ**

*context*

CS_CONTEXT 構造体を指すポインタです。

*context* は終了する Client-Library コンテキストを識別します。

*option*

ct_exit は、*option* に指定された値に応じて動作が異なります。次の表に、*option* の有効な記号値の一覧を示します。

| option の値 | 結果 |
|---|---|
| CS_UNUSED | ct_exit は、未処理の結果がないすべてのオープン接続をクローズして、このコンテキストの Client-Library を終了する。1 つ以上の接続で結果が未処理の場合、ct_exit は CS_FAIL を返し、Client-Library を終了しない。 |
| CS_FORCE_EXIT | ct_exit は、未処理の結果があるかどうかにかかわらず、このコンテキストのすべてのオープン接続をクローズして、このコンテキストの Client-Library を終了する。 |

Client-Library を正常に終了するには、すべての非同期オペレーションが完了した後に ct-exit を呼び出します。

非同期オペレーション実行中に ct_exit が呼び出されると、ルーチンは CS_FAIL を返し、CS_FORCE_EXIT を使用しても Client-Library は正常に終了しません。

**戻り値**

ct_exit は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |

**例**

```
/*
 ** ex_ctx_cleanup()
 **
 ** Parameters:
 **  context  Pointer to context structure.
 **  status   Status of last interaction with Client-
 **           Library.
 **           If not ok, this routine will perform a
 **           force exit.
 **
 ** Returns:
 ** Result of function calls from Client-Library.
 */

CS_RETCODE CS_PUBLIC
 ex_ctx_cleanup(context, status)
 CS_CONTEXT    *context;
 CS_RETCODE    status;
 {
     CS_RETCODE       retcode;
     CS_INT           exit_option;

   exit_option = (status != CS_SUCCEED) ?CS_FORCE_EXIT :
```

```
        CS_UNUSED;
    retcode = ct_exit(context, exit_option);
    if (retcode != CS_SUCCEED)
    {
        ex_error("ex_ctx_cleanup:ct_exit() failed");
        return retcode;
    }
    retcode = cs_ctx_drop(context);
    if (retcode != CS_SUCCEED)
    {
        ex_error("ex_ctx_cleanup:cs_ctx_drop() failed");
        return retcode;
    }
    return retcode;
}
```

このコードは、*exutils.c* サンプル・プログラムからの抜粋です。

**使用法**

- ct_exit は特定のコンテキストの Client-Library を終了します。オープンされているすべての接続をクローズし、内部データ領域の割り付けを解除して、プラットフォーム固有の初期化をクリーンアップします。
- ct_exit は、Client-Library コンテキスト内で呼び出される最後の Client-Library ルーチンにしてください。
- アプリケーションで ct_exit を呼び出した後に Client-Library ルーチンを呼び出す必要が生じた場合は、ct_init を再度呼び出して、Client-Library を再初期化できます。
- そのコンテキストの接続の中に結果が未処理のものがあり、*option* として CS_FORCE_EXIT が渡されていない場合、ct_exit は CS_FAIL を返します。つまり、Client-Library が正しく終了されないので、アプリケーションは、その接続の未処理の結果を処理した後に、ct_exit を再び呼び出さなければなりません。
- ct_exit は、そのコンテキストの接続のいずれかに非同期ネットワーク I/O が指定されている場合でも、常に同期的に完了します。
- アプリケーションは、1 つの接続をクローズするために、ct_close を呼び出すことができます。
- あるコンテキストに対して ct_init を呼び出した場合、ct_exit を呼び出す前にそのコンテキストの割り付けを解除するとエラーになります。

**参照**

ct_close、ct_init

# ct_fetch

**説明**

結果データをフェッチします。

**構文**

```
CS_RETCODE ct_fetch(cmd, type, offset, option,
rows_read)

CS_COMMAND   *cmd;
CS_INT       type;
CS_INT       offset;
CS_INT       option;
CS_INT       *rows_read;
```

**パラメータ**

*cmd*

クライアント／サーバ・オペレーションを管理する CS_COMMAND 構造体を指すポインタです。

*type*

このパラメータは現在未使用ですが、将来のバージョンの Client-Library との互換性を保証するために、CS_UNUSED を渡す必要があります。

*offset*

このパラメータは現在未使用ですが、将来のバージョンの Client-Library との互換性を保証するために、CS_UNUSED を渡す必要があります。

*option*

このパラメータは現在未使用ですが、将来のバージョンの Client-Library との互換性を保証するために、CS_UNUSED を渡す必要があります。

*rows_read*

整数変数を指すポインタです。ct_fetch は、ct_fetch 呼び出しによって読み込まれたロー数を *rows_read* に設定します。

*rows_read* は、配列バインドを使用するアプリケーションによって使用されるオプション・パラメータです。

非同期モードでは、ct_fetch が完了するまで **rows_read* は設定されません。

## 戻り値

ct_fetch は、次の値を返します。

**表 3-42：ct_fetch の戻り値**

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。<br>ct_fetch は読み込んだロー数を **rows_read* に入れる。<br>結果データのフェッチが完了するまで、アプリケーションは ct_fetch を呼び出し続けなくてはならない。 |
| CS_END_DATA | 現在の結果セットのすべてのローがフェッチされた。<br>アプリケーションは ct_results を呼び出して、次の結果セットを取得すべきである。<br>この戻り値は、「ct_scroll_fetch」(629 ページ ) には適用されない。<br>ct_scroll_fetch が CS_END_DATA を返した場合は、致命的内部エラー。 |
| CS_ROW_FAIL | ローのフェッチ中に復元可能なエラーが発生した。アプリケーションは、ローの取得を続けるために ct_fetch の呼び出しを続行する必要があるが、ct_cancel を呼び出して残りの結果をキャンセルすることもできる。<br>配列バインドを使用している場合、CS_ROW_FAIL は、バインドされた配列で部分結果が使用可能であることを示す。ct_fetch は **row_count* を設定して、転送されたローの数 ( エラーのあるローも含む ) を示すが、このローの後はローを転送しない。ct_fetch の次の呼び出しでは、エラーが発生したローの次のローから読み込む。<br>リカバリ可能なエラーとは、メモリ割り付けエラーや、ロー値をプログラム変数にコピーするときに発生する変換エラー ( 送信先バッファのオーバフローなど ) などである。バッファ・オーバフロー・エラーの場合、ct_fetch は該当する **indicator* 変数を 0 より大きい値に設定する。インジケータ変数は、アプリケーションからの ct_bind の呼び出しにあらかじめ指定されていなければならない。 |
| CS_FAIL | ルーチンが失敗。<br>ct_fetch はフェッチしたロー数を **rows_read* に入れる。この数には失敗したローも含まれる。<br>ルーチンがアプリケーション・エラー ( たとえば、不正なパラメータ ) による失敗でない場合、追加結果ローは使用できない。<br>ct_fetch が CS_FAIL を返した場合、アプリケーションは、影響を受けたコマンド構造体を別のコマンドの送信に使用する前に、*type* を CS_CANCEL_ALL にして ct_cancel を呼び出す必要がある。<br>ct_cancel が CS_FAIL を返した場合、アプリケーションは、ct_close(CS_FORCE_CLOSE) を呼び出して接続を強制的にクローズしなければならない。 |

| 戻り値 | 意味 |
| --- | --- |
| CS_CANCELED | 現在の結果セットとすべての追加結果セットがキャンセルされた。データは利用できない。<br>ct_fetch は、キャンセルが発生する前にフェッチしたロー数を **rows_read* に入れる。 |
| CS_PENDING | 非同期ネットワーク I/O が有効。「非同期プログラミング」(12 ページ ) を参照。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

ct_fetch で発生する障害の一般的な原因は、ct_bind で指定されたプログラム変数が、フェッチされるデータ項目に対して十分な大きさではないことです。

**例**

```
/* ex_fetch_data()*/

CS_RETCODE CS_PUBLIC
 ex_fetch_data(cmd)
 CS_COMMAND       *cmd;
 {
     CS_RETCODE    retcode;
     CS_INT        num_cols;
     CS_INT        i;
     CS_INT        j;
     CS_INT        row_count = 0;
     CS_INT        rows_read;

    /*
     ** Determine the number of columns in this
     ** result set.
     */
     ...CODE DELETED.....

    /* Get column descriptions and bind columns */
     ...CODE DELETED.....

    /*
     ** Fetch the rows. Loop while ct_fetch() returns
     ** CS_SUCCEED or CS_ROW_FAIL
     */
     while (((retcode = ct_fetch(cmd, CS_UNUSED,
         CS_UNUSED, CS_UNUSED,&rows_read)) ==
         CS_SUCCEED) || (retcode == CS_ROW_FAIL))
     {
         /*
         ** Increment our row count by the number of
```

```
        ** rows just fetched.
        */
        row_count = row_count + rows_read;

       /* Check if we hit a recoverable error */
        if (retcode == CS_ROW_FAIL)
        {
            fprintf(stdout, "Error on row %d.¥n",
                row_count);
        }

       /*
        ** We have a row. Loop through the columns
        ** displaying the column values.
        */
        for (i = 0; i < num_cols; i++)
        {
            ...CODE DELETED.....
        }
        fprintf(stdout, "¥n");
    }

   /* Free allocated space */
    ...CODE DELETED.....

   /*
    ** We're done processing rows. Let's check the
    ** final return value of ct_fetch().
    */
    switch ((int)retcode)
    {
        case CS_END_DATA:
            /* Everything went fine */
            fprintf(stdout, "All done processing
                rows.¥n");
            retcode = CS_SUCCEED;
            break;

       case CS_FAIL:
            /* Something terrible happened */
            ex_error("ex_fetch_data:ct_fetch()
                failed");
            return retcode;
            break;

       default:
            /* We got an unexpected return value */
            ex_error("ex_fetch_data:ct_fetch() ¥
                returned an unexpected retcode");
```

```
                return retcode;
                break;
        }
        return retcode;
}
```

このコードは、*exutils.c* サンプル・プログラムからの抜粋です。

**使用法**

- 「**結果データ**」は、サーバがアプリケーションに返すことができるすべてのデータのタイプに対する総称です。データのタイプには次のものがあります。
  - 通常ロー
  - カーソル・ロー
  - リターン・パラメータ：メッセージ・パラメータ、ストアド・プロシージャ・リターン・パラメータ、拡張エラー・データ、レジスタード・プロシージャ・ノーティフィケーション・パラメータなど
  - ストアド・プロシージャ・ステータス値
  - 計算ロー

  ct_fetch は、これらすべてのデータ・タイプのフェッチに使用されます。

- 概念的に、結果データは「**結果セット**」を構成する 1 つ以上のローの形式でアプリケーションに返されます。

  通常ローおよびカーソル・ロー結果セットは、複数のローを持つことができます。たとえば、通常ロー結果セットはローを 100 持つことがあります。

  通常ローまたはカーソル・ローの結果セット内のデータ項目に配列バインドが指定されている場合、ct_fetch の一度の呼び出しで複数のローをフェッチできます。

  ---
  **注意**　非同期アプリケーションは常に、配列バインドを指定して、一度に複数のローをフェッチする必要があります。これにより、Client-Library がアプリケーションの完了コールバック・ルーチンを呼び出す前に、アプリケーションで何らかの処理を行う十分な時間が確保されます。

  ---

  リターン・パラメータ、ステータス番号、計算ロー結果セットには、「ロー」は 1 つしか格納されません。そのため、配列バインドが指定されていても、フェッチされるデータ・ローは 1 つだけです。

- ct_results では **result_type* の設定によって、利用できる結果のタイプが示されます。アプリケーションが ct_fetch を呼び出す前に、ct_results は結果タイプ CS_ROW_RESULT、CS_CURSOR_RESULT、CS_PARAM_RESULT、CS_STATUS_RESULT、または CS_COMPUTE_RESULT のいずれかを示していなければなりません。
- ct_results が、フェッチ可能な結果を示す *result_type* を返した後、アプリケーションは次のことができます。
    - 結果項目のバインドとデータのフェッチによって、結果ローを取得します。一般のアプリケーションは、データ項目の数を得るために ct_res_info、データ記述を得るために ct_describe、結果項目をバインドするために ct_bind、結果ローをフェッチするために ct_fetch、および結果セットが大きな text または image 値を持つ場合には ct_get_data をそれぞれ呼び出します。
    - ct_dyndesc または ct_dynsqlda を ct_fetch と一緒に使用して、結果ローを取得します。一般に、動的 SQL コマンドを実行するアプリケーションだけがこれらのルーチンを使用しますが、ct_dyndesc または ct_dynsqlda は、どのコマンド・タイプから返されたフェッチ可能なデータを処理するのにも使用できます。
    - カーソル結果以外のものについては ct_cancel を使用し、カーソル結果については ct_cursor(CS_CURSOR_CLOSE) を使用して、結果ローを廃棄します。
- アプリケーションが結果セットをキャンセルしない場合は、ローが利用できることを ct_fetch が示すかぎり、ct_fetch を呼び出してその結果セットを完全に処理しなければなりません。

  最も簡単な方法は、ct_fetch が CS_SUCCEED または CS_ROW_FAIL のどちらも返さないときに終了するループを使用することです。ループの終了後、アプリケーションで switch 文を使用して ct_fetch の最終リターン・コードを調べ、終了の原因を確認できます。

  結果セットに含まれているロー数が 0 の場合、アプリケーションからの最初の ct_fetch 呼び出しで CS_END_DATA が返されます。

---

**注意**　アプリケーションでは、結果セットが単一ローだけを持つ場合でも、ループで ct_fetch を呼び出す必要があります。また、CS_SUCCEED または CS_ROW_FAIL のどちらも返らなくなるまで、ct_fetch を呼び出す必要があります。

---

- 結果項目の取得中に変換エラーが発生した場合、ロー内の残りの項目は取得されます。トランケーションが発生した場合、この項目に対するアプリケーションの ct_bind 呼び出しでインジケータ変数が提供されていれば、その変数は結果データの実際の長さに設定されます。

  変換またはトランケート・エラーが発生した場合、ct_fetch は CS_ROW_FAIL を返します。

#### 通常ローとカーソル・ローのフェッチ

- 通常ローとカーソル・ローは、一度に 1 つのロー、または一度に複数のローをフェッチできます。
- アプリケーションは、結果カラムをプログラム変数にバインドする ct_bind 呼び出しの *datafmt–>count* フィールドを使用して、ct_fetch 呼び出しごとにフェッチするロー数を指定します。*datafmt–>count* が 0 または 1 の場合、ct_fetch の各呼び出しは 1 つのローをフェッチします。*datafmt–>count* が 1 より大きい場合は、配列バインドが有効であるとみなされ、ct_fetch の各呼び出しは、*datafmt–>count* 値と同じ数のローをフェッチします。*datafmt–>count* は、結果セットのすべての ct_bind 呼び出しで同じ値を持たなければならないことに注意してください。
- 複数のローのフェッチ時にいずれかのローで変換エラーが発生した場合、この ct_fetch の呼び出しでは、それ以上のローは取得されません。

#### リターン・パラメータのフェッチ

- パラメータの結果セットとして、さまざまなタイプのデータをアプリケーションに返すことができます。たとえば次のものを返します。
  - ストアド・プロシージャ・リターン・パラメータ
  - メッセージ・パラメータ
- 拡張エラー・データとレジスタード・プロシージャ・ノーティフィケーション・パラメータも、パラメータ結果セットとして返されますが、アプリケーションは、これらのタイプのデータを処理する ct_results を呼び出さないため、CS_PARAM_RESULT の結果タイプが認識されません。その代わりに、パラメータ・ローは、アプリケーションが、そのデータを含んでいる CS_COMMAND 構造体を取得した後で、簡単にフェッチされます。
- リターン・パラメータ結果セットは、リターン・パラメータ数と等しい数のカラムを持つ 1 つのローから構成されます。

**リターン・ステータスのフェッチ**

- ストアド・プロシージャ・リターン・ステータス結果セットは、1 つのカラムを持つ 1 つのローから構成されるステータス番号です。

**計算ローのフェッチ**

- 計算ローは select 文の compute 句の結果として生成されます。
- 計算ローの結果セットは、そのローを生成した compute 句の集合演算子の数と等しい数のカラムを持つ 1 つのローから構成されます。
- それぞれの計算ローは個別の結果セットとみなされます。

**参照**

ct_bind、ct_describe、ct_get_data、ct_results、「結果」(280 ページ )、ct_scroll_fetch

# ct_get_data

**説明**

データのまとまりをサーバから読み込みます。

**構文**

```
CS_RETCODE ct_get_data(cmd, item, buffer, buflen, outlen)

CS_COMMAND        *cmd;
CS_INT            item;
CS_VOID           *buffer;
CS_INT            buflen;
CS_INT            *outlen;
```

**パラメータ**

*cmd*

クライアント／サーバ・オペレーションを管理する CS_COMMAND 構造体を指すポインタです。

*item*

対象となるデータ項目を表す整数です。ct_get_data を使用して結果セットの複数の項目に対応するデータを取得する場合、*item* の値は増やすことしかできません。つまり、アプリケーションで項目番号 4 のデータを取得した後に、項目番号 3 のデータを取得することはできません。

カラムを取得する場合は、*item* にそのカラムのカラム番号を指定します。select リストの最初のカラムがカラム番号 1、次が番号 2、以下同様に続きます。

計算カラムを取得する場合は、*item* にその計算カラムのカラム番号を指定します。計算カラムは compute 句に指定した順序で返されます。最初に返されるカラムの番号は 1 です。

リターン・パラメータを取得する場合は、*item* にパラメータ番号を指定します。ストアド・プロシージャによって返される最初のパラメータは、番号 1 のパラメータです。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 文に最初に指定されたときの順序で返されます。この順序は、そのストアド・プロシージャを呼び出す RPC 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際には、リターン・パラメータ以外はカウントしないでください。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。

ストアド・プロシージャのリターン・ステータスを取得する場合、リターン・ステータス結果セットには 1 つのステータスだけが許されるので、*item* は 1 でなければなりません。

*buffer*

データ領域を指すポインタです。ct_get_data は *buflen* に指定されたサイズのまとまったカラム値を **buffer* に取り込みます。

*buffer* を NULL にすることはできません。

*buflen*

**buffer* のバイト単位の長さです。

*buflen* が 0 の場合、ct_get_data は、データの取得をしないで項目の I/O 記述子を更新します。

*buflen* は固定長のバッファにも必要です。buflen を CS_UNUSED にすることはできません。

*outlen*

整数変数を指すポインタです。

*outlen* が指定された場合、ct_get_data は **outlen* を **buffer* にあるバイト数に設定します。

## 戻り値

ct_get_data は、次の値を返します。

**表 3-43：ct_get_data の戻り値**

| 戻り値 | 意味 |
| --- | --- |
| CS_SUCCEED | ct_get_data が、このカラムの最後のまとまりではないデータのまとまりを正常に取得した。 |
| CS_FAIL | ルーチンが失敗。<br>ルーチンがアプリケーション・エラー ( たとえば、不正なパラメータ ) により失敗しなかった場合、追加結果ローは利用できない。 |
| CS_END_ITEM | ct_get_data が、このカラムの最後のデータのまとまりを正常に取得した。これはロー内の最後のカラムではない。 |
| CS_END_DATA | ct_get_data が、このカラムの最後のデータのまとまりを正常に取得した。これはロー内の最後のカラムである。 |
| CS_CANCELED | オペレーションがキャンセルされた。この結果のデータは利用できない。 |
| CS_PENDING | 非同期ネットワーク I/O が有効。「非同期プログラミング」(12 ページ ) を参照。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

## 例

```
/*
 ** FetchResults()
 **
 ** The result set contains four columns: integer, text,
 ** float, and integer.
 */

CS_STATIC CS_RETCODE
 FetchResults(cmd, textdata)
 CS_COMMAND      *cmd;
 TEXT_DATA       *textdata;
 {
     CS_RETCODE      retcode;
     CS_DATAFMT      fmt;
     CS_INT          firstcol;
     CS_TEXT         *txtptr;
     CS_FLOAT        floatitem;
     CS_INT          count;
     CS_INT          len;

    /*
     ** All binds must be of columns prior to the columns
     ** to be retrieved by ct_get_data().
```

```
 ** To demonstrate this, bind the first column returned.
 */
 ...CODE DELETED.....

/* Retrieve and display the results */
 while(((retcode = ct_fetch(cmd, CS_UNUSED, CS_UNUSED,
     CS_UNUSED,&count)) == CS_SUCCEED) ||
     (retcode == CS_ROW_FAIL) )
 {
     /* Check for a recoverable error */
     ...CODE DELETED.....

    /*
     ** Get the text data item in the second column.
     ** Loop until we have all the data for this item.
     ** The text used for this example could be
     ** retrieved in one ct_get_data call, but data
     ** could be too large for this to be the case.
     ** Instead, the data would have to be retrieved
     ** in chunks. This example will retrieve the text
     ** in 5 byte increments to demonstrate retrieving
     ** data items in chunks.
     */
     txtptr = textdata->textbuf;
     textdata->textlen = 0;
     do
     {
         retcode = ct_get_data(cmd, 2, txtptr, 5,
             &len);
         textdata->textlen += len;
         /*
         ** Protect against overflowing the string
         ** buffer.
         */
         if ((textdata->textlen + 5) > (EX_MAX_TEXT -
             1))
         {
             break;
         }
         txtptr += len;
     } while (retcode == CS_SUCCEED);

    if (retcode != CS_END_ITEM)
     {
         ex_error("FetchResults:ct_get_data()
             failed");
         return retcode;
     }
```

```
        /*
         ** Retrieve the descriptor of the text data. It is
         ** available while retrieving results of a select
         ** query. The information will be needed for
         ** later updates.
         */
          ...CODE DELETED...

        /* Get the float data item in the 3rd column */
         retcode = ct_get_data(cmd, 3, &floatitem,
              sizeof (floatitem), &len);
         if (retcode != CS_END_ITEM)
         {
              ex_error("FetchResults:ct_get_data()
                  failed");
              return(retcode);
         }

        /*
         ** When using ct_get_data to process results,
         ** it is not required to get all the columns
         ** in the row. To illustrate this, the last
         ** column of the result set is not retrieved.
         */
     }

    /*
     ** We're done processing rows. Check the
     ** final return value of ct_fetch().
     */
        ...CODE DELETED.....

    return retcode;
}
```

このコードは、*getsend.c* サンプル・プログラムからの抜粋です。

**使用法**

- アプリケーションは、通常、大きな text または image 値を検索するループ内で **ct_get_data** を呼び出しますが、どんなデータ型のカラムでも使用できます。ct_get_data の各呼び出しは *buflen* で指定された長さのまとまりのデータを取得します。
- ct_get_data を使用して text 値や image 値を取得する手順については、「text および image 値をフェッチする ct_get_data の使用」(329 ページ ) を参照してください。

- ct_get_data は、サーバによって送信された状態でデータを取得します。変換は実行されません。このため、**buffer* に含まれるデータを解釈する場合は注意してください。特に、CS_CHAR データは null で終了しないことがあり、マルチバイト文字列は 1 つの文字を定義するバイト列の途中で分割されることがあります。
- アプリケーションは、対象となるローをフェッチするために、ct_fetch を呼び出した後 ct_get_data を呼び出します。配列バインドが ct_bind の以前の呼び出しで指定された場合、アプリケーションは ct_get_data を使用できません。
- 最後にバインドされたカラムの後続のカラムだけが ct_get_data に利用できます。バインドされたカラムに先行するバインドされていないカラム・データは廃棄されます。たとえば、アプリケーションがカラム番号 1 ～ 4 を選択し､そのうちカラム番号 1 と 3 のカラムをバインドした場合、そのアプリケーションで ct_get_data を使用してカラム番号 2 のカラムを取得することはできません。この場合､ct_get_data でカラム番号 4 のカラムのデータは取得できます。
- 一度、カラムのデータが取得されると、そのカラムはもはや利用できません。
- 後から更新が必要な text カラムや image カラムをアプリケーションで読み込む場合、そのカラムの I/O 記述子を取得する必要があります。それには、アプリケーションは、対象カラムで ct_get_data を呼び出した後に、ct_data_info を呼び出します。
- カラムの値が null の場合、ct_get_data は **outlen* を 0 に設定して CS_END_ITEM または CS_END_ITEM を返します。
- アプリケーションは、カラムに ct_get_data を呼び出す前に、そのカラムの I/O 記述子を取得することはできません。ただし、この ct_get_data 呼び出しではデータを実際に取得する必要はありません。つまり､アプリケーションは buflen を 0 にして ct_get_data を呼び出し、その記述子を取得するために ct_data_info を呼び出すことができます。この手法は、アプリケーションが text 値や image 値を取得する前にその長さを確認する必要があるときに役立ちます。

**参照**

ct_bind、ct_data_info、ct_fetch、ct_send_data、text および image データの処理

# ct_getformat

**説明**

結果カラムに関連するサーバのユーザ定義フォーマット文字列を返します。

**構文**

```
CS_RETCODE ct_getformat (cmd, colnum, buffer, buflen,
          outlen)

CS_COMMAND        *cmd;
CS_INT            colnum;
CS_VOID           *buffer;
CS_INT            buflen;
CS_INT            *outlen;
```

**パラメータ**

*cmd*

クライアント／サーバ・オペレーションを管理する CS_COMMAND 構造体を指すポインタです。

*colnum*

ユーザ定義フォーマットが必要なカラムの番号です。select リストの最初のカラムが番号 1、次が番号 2、以下同様に続きます。

*buffer*

ct_getformat が NULL で終了するフォーマット文字列を入れる領域を指すポインタです。

*buflen*

**buffer* データ領域のバイト単位の長さです。

*outlen*

整数変数を指すポインタです。

*outlen* が指定された場合、ct_getformat はフォーマット文字列のバイト単位の長さを **outlen* に設定します。この長さは NULL ターミネータを含んでいます。

フォーマット文字列が *buflen* バイトより大きい場合、アプリケーションでは、その文字列の保持に必要なバイト数を判断するために、**outlen* の値を使用できます。

*colnum* で識別されたカラムに関連付けられているフォーマット文字列がない場合、ct_getformat は **outlen* を 1 (NULL ターミネータ ) に設定します。

**戻り値**

ct_getformat は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

**使用法**

- アプリケーションは ct_results が CS_ROW_RESULT タイプの結果を示した後に、ct_getformat を呼び出すことができます。
- *colnum* で識別されたカラムに関連付けられているフォーマット文字列がない場合、ct_getformat は **outlen* を 1 に設定します。
- 一般のアプリケーションでは、ct_getformat は使用しません。このルーチンは主にゲートウェイ・アプリケーションをサポートするためのものです。

**参照**

ct_bind、ct_describe

# ct_getloginfo

**説明**

TDS ログイン応答情報を CS_CONNECTION 構造体から新しく割り付けた CS_LOGINFO 構造体に転送します。

**構文**

```
CS_RETCODE ct_getloginfo (connection, logptr)

CS_CONNECTION    *connection;
CS_LOGINFO       **logptr;
```

**パラメータ**

*connection*

CS_CONNECTION 構造体を指すポインタです。CS_CONNECTION 構造体は、特定のクライアント／サーバ接続の情報を含んでいます。

*logptr*

ct_getloginfo が新しく割り付けた CS_LOGINFO 構造体のアドレスを設定するプログラム変数を指すポインタです。

**戻り値**

ct_getloginfo は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

**使用法**

- TDS (Tabular Data Stream) は、クライアントとサーバの間における要求と要求結果の転送に使用される通信プロトコルです。
- アプリケーションが ct_getloginfo を呼び出す理由は 2 つあります。
  - TDS パススルーを使用した Open Server ゲートウェイ・アプリケーションの場合

- オープン接続から新しく割り付けた接続構造体に、ログイン・プロパティをコピーするため

> **注意** ct_getloginfo は、完了コールバック・ルーチン内からは呼び出さないでください。ct_getloginfo は再入不可能なシステム・レベルのメモリ関数を呼び出します。

### TDS パススルー

- クライアントがサーバと直接接続している場合、2 つのプログラムは、データの送受信に使用する TDS のフォーマットをネゴシエートします。ゲートウェイ・アプリケーションが TDS パススルーを使用する場合、そのゲートウェイは、クライアントとリモート・サーバの間で TDS パケットを検査および処理することなく転送します。このため、リモート・サーバとクライアントは、使用する TDS フォーマットについて合意しておく必要があります。
- ct_getloginfo は、クライアントとリモート・サーバ間で TDS フォーマットをネゴシエートするときに行われる 4 つの呼び出しのうち 3 番目の呼び出しです。4 つのうち 2 つは Server Library の呼び出しです。Open Server SRV_CONNECT イベント・ハンドラだけで行うことができる呼び出しは、次のとおりです。

    a. CS_LOGINFO 構造体を割り付けて、それをクライアント・ログイン要求からの TDS 情報で満たす srv_getloginfo。

    b. 手順 1 で取得された TDS 情報を CS_LOGINFO 構造体から Client-Library CS_CONNECTION 構造体に転送する ct_setloginfo。ゲートウェイは、リモート・サーバとの接続を確立する ct_connect 呼び出しに、この CS_CONNECTION 構造体を使用します。

    c. クライアントの TDS 情報に対するリモート・サーバの応答を、CS_CONNECTION 構造体から新しく割り付けた CS_LOGINFO 構造体に転送する ct_getloginfo。

    d. 手順 3 で取得されたリモート・サーバの応答をクライアントに送信する srv_setloginfo。

### ログイン・プロパティのコピー

オープン接続から新しく割り付けた接続構造体にログイン・プロパティをコピーするための ct_getloginfo の使用については、「プロパティ」(208 ページ) を参照してください。

**参照**　ct_recvpassthru、ct_sendpassthru、ct_setloginfo

# ct_init

**説明**

アプリケーション・コンテキストの Client-Library を初期化します。

**構文**

```
CS_RETCODE ct_init(context, version)

CS_CONTEXT    *context;
CS_INT        version;
```

**パラメータ**

*context*

CS_CONTEXT 構造体を指すポインタです。アプリケーションは、CS-Library ルーチン cs_ctx_alloc を呼び出して、このコンテキスト構造体を事前に割り付ける必要があります。

*context* は初期化する Client-Library コンテキストを特定します。

*version*

アプリケーションが必要とする Client-Library の動作のバージョンです。表 3-44 に、*version* の記号値を示します。

**表 3-44：ct_init version パラメータの値**

| **version の値** | **意味** | **サポートされる機能** |
|---|---|---|
| CS_VERSION_100 | 10.0 の動作作 | カーソル、レジスタード・プロシージャ、リモート・プロシージャ・コール。<br>これは Client-Library の初期バージョン。 |
| CS_VERSION_110 | 11.0 の動作 | 10.0 の全機能にバージョン 11.1 の以下の機能が加わる。<br>• ネットワークベースのディレクトリおよびセキュリティ・サービス<br>• プロパティ、オプション、および機能の外部設定 |
| CS_VERSION_120 | 12.0 の動作 | これまでの機能に以下の機能が加わる。<br>• 高可用性フェールオーバ<br>• Digital UNIX プラットフォームでのネイティブ・スレッドのサポート<br>• バルク・ローの挿入<br>• ソートマージ・ジョインを有効化／無効化する新しいプロパティ |
| CS_VERSION_125 | 12.5 の動作 | バージョン 12.5 で追加された機能には次が含まれる。<br>• LDAP セキュリティ機能<br>• SSL セキュリティ機能<br>• 2 バイト文字用 Unichar-16 のサポート<br>• ワイド・カラムとワイド・テーブルのサポート |
| CS_VERSION_150 | 15.0 の動作 | BCP パーティション、BCP 計算カラム、長い識別子、Unilib、Adaptive Server Enterprise のデフォルト・パケット・サイズ、スクロール可能カーソル、およびクラスタのサポート。同時に unitext、xml、bigint、usmallint、uint、および ubigint の各データ型もサポートする。Sybase ライブラリ名の変更に注意。 |
| CS_VERSION_155 | 15.5 の動作 | CS_BIGDATETIME データ型と CS_BIGTIME データ型、マイクロ秒の精度の time データ、ct_send_data の強化、Open Server 動的なリスナ、Open Client CS_RES_NOXNLMETADATA 応答機能、FIPS-140-2 準拠のパスワード暗号化。 |

| version の値 | 意味 | サポートされる機能 |
|---|---|---|
| CS_VERSION_157 | 15.7 の動作 | ラージ・オブジェクト (LOB) ロケータ・サポート、ストアド・プロシージャ・パラメータとしての LOB、ロー内とロー外の LOB のサポート、Bulk-Library と *bcp* による非実体化カラムの処理後続ゼロ維持のサポート、名前のないアプリケーションの設定に関する新しい処理、TCP ソケット・バッファ・サイズの設定、可変長のローの拡張、カーソル・クローズ時のロックの解放ロー・フォーマットのキャッシュ。 |

## 戻り値

ct_init は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_MEM_ERROR | メモリの割り付けエラーのため、ルーチンが失敗した。 |
| CS_FAIL | ルーチンが失敗した。 |

Client-Library が *version* レベルの動作を提供できない場合、ct_init は CS_FAIL を返します。

**注意**　Net-Library エラーによって ct_init が CS_FAIL を返した場合は、標準エラー (STDERR) と、現在の作業ディレクトリに作成される sybinit.err ファイルに、詳細なエラー情報が送られます。

ct_init 障害によって、通常、**context* が使用不可能になることはありません。コンテキスト構造体を解除する代わりに、アプリケーションは、ct_init の呼び出しを同じ *context* ポインタを使用して再試行できます。

## 例

```
/*
 ** ex_init() -- Allocate and initialize a CS_CONTEXT
 ** structure.
 **
 ** EX_CTLIB_VERSION is defined in the examples header file
 ** as CS_VERSION_110.
 */

 CS_RETCODE CS_PUBLIC
  ex_init(context)
  CS_CONTEXT  **context;
  {
```

```
    CS_RETCODE       retcode;

    /* Get a context handle to use */
    retcode = cs_ctx_alloc(EX_CTLIB_VERSION, context);
         ... error checking code deleted ...

    /* Initialize Open Client */
    retcode = ct_init(*context, EX_CTLIB_VERSION);
    if (retcode != CS_SUCCEED)
    {
        ex_error("ex_init:ct_init() failed");
        cs_ctx_drop(*context);
        *context = NULL;
        return retcode;
    }

    /* Install client and server message handlers */
    ... ct_callback calls deleted .....

    /* Call ct_config to set context properties */
    ... ct_config calls deleted ...

    /* Exit from Client-Library */
    retcode = ct_exit(context, CS_UNUSED);
    if (retcode != CS_SUCCEED)
    {
        ct_exit(*context, CS_FORCE_EXIT);
        cs_ctx_drop(*context);
        *context = NULL;
    }

    return retcode;
}
```

このコードは、*exutils.c* サンプル・プログラムからの抜粋です。

**使用法**

- ct_init は、内部制御構造体を設定し、アプリケーションが必要とする Client-Library 動作のバージョンを定義します。
- ct_init は、Client-Library アプリケーション・コンテキストで呼び出される最初の Client-Library ルーチンである必要があります。他の Client-Library ルーチンが ct_init の前に呼び出されると、失敗します。

**注意**　Client-Library アプリケーションは、ct_init を呼び出す前に CS-Library ルーチンを呼び出すことができます ( 実際には、ct_init を呼び出す前に CS-Library ルーチン cs_ctx_alloc を呼び出す必要があります )。

- ct_init が CS_SUCCEED を返した場合、Client-Library は使用中の Client-Library の実際のバージョンにかかわらず、要求された動作を提供します。Client-Library が要求された動作を提供できない場合、ct_init は CS_FAIL を返します。一般的に、Client-Library の上位レベルのバージョンは下位レベルの動作を提供できますが、下位バージョンは上位レベルの動作を提供できません。
- アプリケーションはエラー処理を設定する前に ct_init を呼び出すので、ct_init のリターン・コードを調べて障害を検出する必要があります。
- アプリケーションが、同じコンテキストに対して ct_init を複数回呼び出してもエラーにはなりません。この状態になった場合、最初の呼び出しだけが影響します。一部のアプリケーションでは、いくつかのモジュールのうちどれを最初に実行するか決定できないものがあるため、Client-Library では、この機能を提供しています。このような場合、各モジュールでは ct_init の呼び出しを含む必要があります。
- *version* には、アプリケーションが必要とする Client-Library のバージョンを指定します。*version* の設定によって、そのコンテキストの CS_VERSION プロパティの値が決まります。コンテキスト内で割り付けられた接続は、その親コンテキストの CS_VERSION レベルに基づいた、CS_TDS_VERSION のデフォルト値を使用します。

**コンテキスト・プロパティの外部設定**

- アプリケーションが ct_init を呼び出す前に CS_CONFIG_FILE コンテキスト・プロパティを設定する cs_config を呼び出して外部設定を要求した場合、Client-Library は、Open Client/Server 設定ファイルを読み込んで、デフォルト・コンテキスト・プロパティ値を取得します。
- 外部設定によって、アプリケーション内の ct_config 呼び出し回数は減少します。また、アプリケーションが外部設定を要求するようにコーディングされている場合、再コンパイルなしでアプリケーションのランタイム・プロパティ設定値を変更することもできます。「ランタイム設定ファイルの使い方」(352 ページ ) を参照してください。

**参照**　　cs_ctx_alloc、ct_exit、ct_config

# ct_keydata

**説明**

キー・カラムの内容を指定または抽出します。

**構文**

```
CS_RETCODE ct_keydata (cmd, action, colnum, buffer,
        buflen, outlen)

CS_COMMAND   *cmd;
CS_INT       action;
CS_INT       colnum;
CS_VOID      *buffer;
CS_INT       buflen;
CS_INT       *outlen;
```

**パラメータ**

*cmd*

クライアント／サーバ・カーソル・オペレーションを管理する CS_COMMAND 構造体を指すポインタです。

*action*

次の記号値のいずれかです。

| action の値 | 結果 |
|---|---|
| CS_SET | キー・カラムの内容を設定する。 |
| CS_GET | キー・カラムの内容を取得する。 |

*colnum*

対象となるカラムの番号です。結果セットの最初のカラムが番号 1、次が番号 2、以下同様に続きます。

*colnum* に指定するカラムは CS_KEY カラムか CS_VERSION_KEY カラムでなければなりません。ct_describe は、カラムが CS_KEY または CS_VERSION_KEY カラムのどちらであるかを示すために、*datafmt–>status* フィールドを設定します。

*buffer*

キー・カラムを設定する場合、*buffer* はそのキー・カラムの設定に使用する値を指します。

キー・カラム値を取得する場合、*buffer* は ct_keydata が要求された情報を入れる領域を指します。

*buflen*

**buffer* のバイト単位の長さです。

キー・カラム値を設定していて **buffer* の値が null で終了する場合は、*buflen* に CS_NULLTERM を指定して渡してください。

キー・カラム値を取得していて、**buffer* が要求された情報を保持できるほど大きくないことを *buflen* が示している場合、ct_keydata は要求された情報の長さを **outlen* に設定して CS_FAIL を返します。

*buflen* は固定長のバッファにも必要です。また、CS_UNUSED として渡すことはできません。

*outlen*

整数変数を指すポインタです。

キー・カラム値を設定する場合、*outlen* は使用されないので、NULL として渡されなければなりません。

キー・カラム値を取得する場合、ct_keydata は要求された情報の長さをバイト単位で **outlen* に設定します。

情報が *buflen* バイトより大きい場合、アプリケーションは情報の保持に必要なバイト数を判断するために、**outlen* の値を使用することができます。

アプリケーションがキー・カラム値を設定する場合、またはリターンの長さの情報について関知しない場合、*outlen* を NULL として渡すことができます。

**戻り値**

ct_keydata は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

*colnum* がキー・カラムを表していない場合、ct_keydata は CS_FAIL を返します。

**使用法**

- アプリケーションは、ct_keydata を使用して現在のカーソル位置を再定義してから、カーソルの更新または削除を実行できます。
- ct_keydata の主な用途は次の 2 つです。

- クライアントとサーバ間でカーソル・ローをバッファリングするゲートウェイ・アプリケーション用です。この場合、カーソル位置についてのクライアントの概念がゲートウェイの概念が異なることがあります。クライアントが位置付け更新または位置付け削除の要求を送信した場合、ゲートウェイは、ct_keydata を使用して、サーバに対するターゲット・ローを正しく識別できます。
- ユーザがデータ・ローをブラウズし、それらを任意の順序で変更または削除できるアプリケーション用です。この場合、ユーザは、現在のカーソル・ローではないローの変更または削除をアプリケーションに依頼することがあります。アプリケーションは、ct_keydata を使用してターゲット・ローを現在のローとして再定義できます。

- キーは複数のカラムにまたがることがあるので、ローの全体キーを指定するためにアプリケーションから ct_keydata を複数回呼び出さなければならないことがあります。
- ct_fetch を呼び出すと、アプリケーションが指定していたキー・カラム値はすべて消去されます。
- アプリケーションは次の状況においてのみ、ct_keydata を呼び出すことができます。
  - 現在の結果タイプが CS_CURSOR_RESULT の場合
  - カーソルをサポートしているコマンド構造体が CS_TRUE に設定された CS_HIDDEN_KEYS プロパティを持っている場合
  - カーソルが最低でも 1 つフェッチされた場合
- キーを更新する場合、すべてのキー・カラムを更新してください。ローの全体キーが再定義されていないときに位置付け更新または位置付け削除が行われた場合、ct_cursor は CS_FAIL を返します。
- アプリケーションは、*buffer* を NULL、*buflen* を 0 または CS_UNUSED にして ct_keydata を呼び出すことによって、キー・カラムの値を NULL に設定できます。カラムが null 値を許可しない場合、ct_keydata は CS_FAIL を返します。

**参照** ct_cursor、ct_describe、ct_res_info、ct_results

# ct_labels

**説明**

接続のセキュリティ・ラベルを定義またはクリアします。

**構文**

```
CS_RETCODE ct_labels(connection, action,
          labelname, namelen, labelvalue,
          valuelen, outlen)

CS_CONNECTION    *connection;
CS_INT           action;
CS_CHAR          *labelname;
CS_INT           namelen;
CS_CHAR          *labelvalue;
CS_INT           valuelen;
CS_INT           *outlen;
```

**パラメータ**

*connection*

CS_CONNECTION 構造体を指すポインタです。CS_CONNECTION 構造体は、特定のクライアント／サーバ接続の情報を含んでいます。

*connection* にはクローズした接続を指定する必要があります。

*action*

次の記号値のいずれかです。

| action の値 | 結果 |
|---|---|
| CS_SET | セキュリティ・ラベルを設定する。 |
| CS_CLEAR | この接続のために前に指定したセキュリティ・ラベルをクリアする。 |

*labelname*

*action* が CS_SET の場合、*labelname* は設定されているセキュリティ・ラベルを指します。

*action* が CS_CLEAR の場合、*labelname* を NULL にしてください。

*namelen*

**labelname* のバイト単位の長さです。**labelname* が null で終了している場合、*namelen* を CS_NULLTERM として渡します。

セキュリティ・ラベル名は、1 バイト以上で CS_MAX_NAME バイト以下でなければなりません。

*action* が CS_CLEAR の場合、*namelen* を CS_UNUSED として渡します。

*labelvalue*

*action* が CS_SET の場合、*labelvalue* は設定されているセキュリティ・ラベルの値を指します。

*action* が CS_CLEAR の場合、*labelvalue* を NULL にしてください。

*valuelen*

**labelvalue* のバイト単位の長さです。**labelvalue* が null で終了している場合、*valuelen* を CS_NULLTERM として渡します。

セキュリティ・ラベル値は、1 バイト以上にしてください。

*action* が CS_CLEAR の場合、*valuelen* を CS_UNUSED として渡します。

*outlen*

このパラメータは現在使用していません。NULL として渡します。

**戻り値**

ct_labels は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

**使用法**

- trusted ユーザ・セキュリティ・ハンドシェイクを使用してサーバと接続する場合、アプリケーションではセキュリティ・ラベルを定義する必要があります。
- アプリケーションがセキュリティ・ラベルを定義する方法には、次の 2 つがあります。アプリケーションは、次の 2 通りの方法のどちらかまたはその両方を使用できます。
  - アプリケーションは、定義するラベルごとに一度 ct_labels ルーチンを呼び出すことができます。
  - アプリケーションは ct_callback ルーチンを呼び出して、セキュリティ・ラベルを生成するユーザ提供のネゴシエーション・コールバックをインストールできます。接続時に、Client-Library はセキュリティ・ラベルの要求に応答して、自動的にコールバックをトリガします。

  アプリケーションが前述の両方の方法を使用する場合、ct_labels によって定義されたラベルと、ネゴシエーション・コールバックによって生成されたラベルが同時にサーバに送信されます。

- trusted ユーザ・セキュリティ・ハンドシェイクに参加する接続では、CS_SEC_NEGOTIATE プロパティを CS_TRUE に設定します。
- 接続に対して定義できるセキュリティ・ラベルの数に制限はありません。
- ct_labels は、セキュリティ・ラベルに対していかなるチェックも行わずに、サーバにラベル名とラベル値を渡します。

  たとえば、アプリケーションが同じラベル名に 2 つのラベル値を付けても ct_labels はエラーになりません。

**参照**

ct_callback、ct_con_props、ct_connect

# ct_options

**説明**

サーバのクエリ処理オプションの値を設定、取得、またはクリアします。

**構文**

```
CS_RETCODE ct_options(connection, action, option,
             param, paramlen, outlen)

CS_CONNECTION    *connection;
CS_INT           action;
CS_INT           option;
CS_VOID          *param;
CS_INT           paramlen;
CS_INT           *outlen;
```

**パラメータ**

*connection*

CS_CONNECTION 構造体を指すポインタです。CS_CONNECTION 構造体は、特定のクライアント／サーバ接続の情報を含んでいます。

*connection* はオプションを設定、取得、またはクリアするサーバ接続です。

*action*

次の記号値のいずれかです。

| action の値 | 結果 |
|---|---|
| CS_SET | オプションを設定する。 |
| CS_GET | オプションを取得する。 |
| CS_CLEAR | オプションをデフォルト値にリセットすることによりクリアする。デフォルト値はアプリケーションが接続されたサーバによって決定される。 |

*option*

対象とするサーバ・オプションです。表 3-45 ( 587 ページ ) に *option* の記号値を示します。「オプション」(200 ページ ) を参照してください。

*param*

すべてのオプションにパラメータが必要です。

オプションを設定する場合、*param* は記号値、ブール値、整数値、文字列のいずれかを指すことができます。

次に例を示します。

- CS_OPT_DATEFIRST オプションはパラメータとして記号値を取ります。

```
CS_INT       parmvalue;
 parmamvalue = CS_OPT_TUESDAY;
 ct_options(conn, CS_SET, CS_OPT_DATEFIRST,
      &paramvalue, CS_UNUSED, NULL);
```

- CS_OPT_CHAINXACTS オプションは、パラメータとしてブール値を取ります。

```
CS_BOOL    parmvalue;
 parmamvalue = CS_TRUE;
 ct_options(conn, CS_SET, CS_OPT_CHAINXACTS,
      &paramvalue, CS_UNUSED, NULL);
```

- CS_OPT_ROWCOUNT オプションは、パラメータとして整数を取ります。

```
CS_INT       parmvalue;
 paramvalue = 50;
 oc_options(conn, CS_SET, CS_OPT_ROWCOUNT,
      &paramvalue, CS_UNUSED, NULL);
```

- CS_OPT_IDENTITYOFF オプションは、パラメータとして文字列を取ります。

```
ct_options(conn, CS_SET, CS_OPT_IDENTITYOFF,
      "authors", CS_NULLTERM, NULL);
```

オプションを取得する場合、*param* は ct_options がオプションの値を入れる領域を指します。

**param* がオプションの値を保持できるほど大きくないことを *paramlen* が示している場合、ct_option は **outlen* にその値の長さを設定して CS_FAIL を返します。

オプションをクリアする場合、*param* を NULL にしてください。

*paramlen*

**param* のバイト単位の長さです。

固定長パラメータを持つオプションを設定または取得する場合、*paramlen* に CS_UNUSED を指定して渡してください。

文字列パラメータを持つオプションを設定する場合、**param* の値が null で終了していれば、*paramlen* に CS_NULLTERM を指定して渡してください。

オプションを取得するとき、**param* が要求された情報を保持できるほど大きくないことを *paramlen* が示している場合、ct_options は要求された情報の長さを **outlen* に設定して CS_FAIL を返します。

オプションをクリアする場合、*paramlen* は CS_UNUSED にしてください。

*outlen*

整数変数を指すポインタです。

オプションを設定またはクリアする場合、*outlen* は使用されないので、NULL として渡してください。

オプションを取得する場合、ct_options はそのオプションの値のバイト単位の長さを **outlen* に設定します。この長さには、適用可能であれば null ターミネータが含まれます。

オプションの値が *paramlen* のバイト数より大きい場合、アプリケーションは、**outlen* の値に基づいて、その情報の保持に必要なバイト数を判断できます。

## 戻り値

ct_options は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。ct_options が CS_FAIL を返した場合、**param* は元のまま。 |
| CS_CANCELED | オペレーションがキャンセルされた。 |
| CS_PENDING | 非同期ネットワーク I/O が有効。「非同期プログラミング」(12 ページ ) を参照。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

## 使用法

**表 3-45：ct_options パラメータの一覧**

| option の値 | *param の値 | *param の有効値 | デフォルト |
|---|---|---|---|
| CS_OPT_ANSINULL | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_ANSIPERM | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |

| option の値 | *param の値 | *param の有効値 | デフォルト |
|---|---|---|---|
| CS_OPT_ARITHABORT | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_ARITHIGNORE | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_AUTHOFF | 権限レベルを表す文字列。 | 文字列値<br>可能な値には「sa」、「sso」、および「oper」がある。 | 適用しない |
| CS_OPT_AUTHON | 権限レベルを表す文字列。 | 文字列値<br>可能な値には「sa」、「sso」、および「oper」がある。 | 適用しない |
| CS_OPT_CHAINXACTS | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_CHARSET | *locale.dat* ファイル上のサポートされている言語の名前。オープンしている接続上での言語と文字セットの設定に使用する。 | *locales.dat* ファイル上で、プラットフォームの言語を表す値。 | NULL |
| CS_OPT_CURCLOSEONXACT | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_DATEFIRST | 週の最初の曜日として使用する日を表す記号値。 | CS_OPT_SUNDAY、CS_OPT_MONDAY、CS_OPT_TUESDAY、CS_OPT_WEDNESDAY、CS_OPT_THURSDAY、CS_OPT_FRIDAY、CS_OPT_SATURDAY | us_english では、デフォルトは CS_OPT_SUNDAY。 |
| CS_OPT_DATEFORMAT | 日付値に使用する年、月、および日の順序を表す記号値。 | CS_OPT_FMTMDY、CS_OPT_FMTDMY、CS_OPT_FMTYMD、CS_OPT_FMTYDM、CS_OPT_FMTMYD、CS_OPT_FMTDYM | us_english では、デフォルトは CS_OPT_FMTMDY。 |
| CS_OPT_FIPSFLAG | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_FORCEPLAN | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_FORMATONLY | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_GETDATA | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_HIDE_VCC | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_IDENTITYOFF | テーブル名を表す文字列値。 | 文字列値。 | NULL |
| CS_OPT_IDENTITYON | テーブル名を表す文字列値。 | 文字列値。 | NULL |
| CS_OPT_IDENTITYUPD_OFF | ID 更新オプションを無効にする。 | 文字列値。 | NULL |

| option の値 | *param の値 | *param の有効値 | デフォルト |
|---|---|---|---|
| CS_OPT_IDENTITYUPD_ON | ID 更新オプションを有効にする。 | 文字列値。 | NULL |
| CS_OPT_ISOLATION | トランザクション独立性レベルを表す記号値。 | CS_OPT_LEVEL1、CS_OPT_LEVEL0、CS_OPT_LEVEL3<br>CS_OPT_LEVEL0 を指定できるのは、Adaptive Server Enterprise リリース 11.0 以降。 | CS_OPT_LEVEL1 |
| CS_OPT_LOBLOCATOR | サーバからの LOB ロケータの送信を有効にする。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_NATLANG | *locale.dat* ファイル上のサポートされている言語の名前。オープンしている接続上での言語と文字セットの設定に使用する。 | *locales.dat* ファイル上で、プラットフォームの言語を表す値。 | NULL |
| CS_OPT_NOCOUNT | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_NOEXEC | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_PARSEONLY | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_PREFETCHSIZE | クライアントにロケータ値と共に送信する実際の LOB のデータ量を決定する ( プリフェッチ・データ )。 | 整数値 >= 0 または -1。<br>-1 では、LOB データ全体がプリフェッチされる。 | 0 の場合、プリフェッチ・データが送信される。 |
| CS_OPT_QUOTED_ IDENT | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_RESTREES | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_ROWCOUNT | 1 つのクエリによって影響を受ける最大ロー数。select で返される通常ローの数か、update または delete で変更されるローの数を制限する。 | 整数値。<br>0 は制限がないことを意味する。 | 0 ( 無制限 )。 |
| CS_OPT_SHOW_FI | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_SHOWPLAN | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_STATS_IO | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_STATS_TIME | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_STR_RTRUNC | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |
| CS_OPT_TEXTSIZE | サーバが返す最大の text または image 値のバイト単位での長さ。 | 整数値。 | 32,768 バイト |
| CS_OPT_TRUNCIGNORE | ブール値。 | CS_TRUE、CS_FALSE | CS_FALSE |

- クエリ処理オプションは Transact-SQL set コマンドを通じて設定およびクリアできますが、Client-Library アプリケーションでは代わりに ct_options を使用することをおすすめします。これは、アプリケーションは ct_options を使用してオプションのステータスをチェックできますが、set コマンドではチェックできないためです。
- アプリケーションは、ct_options を使用して、一度に 1 接続に対してのみサーバ・オプションを変更できます。接続はオープンされている必要があり、アクティブ・コマンドや未処理の結果はあってはいけませんが、オープン・カーソルはあっても構いません。
- ルーチン ct_connect は、必要に応じて Open Client/Server ランタイム設定ファイルから 1 つのセクションを読み込んで、新しくオープンした接続のサーバ・オプションを設定します。この機能については、「ランタイム設定ファイルの使い方」(352 ページ ) を参照してください。

参照　ct_capability、ct_con_props、「オプション」(200 ページ )

# ct_param

説明　サーバ・コマンドの入力パラメータの値を提供します。

構文

```
CS_RETCODE ct_param(cmd, datafmt, data, datalen, indicator);

CS_COMMAND        *cmd;
CS_DATAFMT        *datafmt;
CS_VOID           *data;
CS_INT            datalen;
CS_SMALLINT       indicator;
```

パラメータ

*cmd*

クライアント/サーバ・オペレーションを管理する CS_COMMAND 構造体を指すポインタです。

*datafmt*

パラメータを記述する CS_DATAFMT へのポインタです。

ct_param の実際の使用時にこれらのフィールドを設定する方法については、「使用法」を参照してください。

*data*

パラメータ・データのアドレスです。

null 値を持つパラメータを示すには、2 つの方法があります。

- *indicator* を -1 として渡してください。この場合、*data* と *datalen* は無視されます。
- *data* を NULL、*datalen* を 0 または CS_UNUSED として渡します。

*datalen*

パラメータ・データのバイト単位の長さです。

*datafmt–>datatype* がパラメータは固定長タイプであることを示している場合、*datalen* は無視されます。CS_VARBINARY および CS_VARCHAR は固定長タイプです。

*indicator*

null 値のパラメータを示すために使用する整数変数です。null 値のパラメータを示すには、*indicator* を -1 にして渡してください。*indicator* が -1 の場合、*data* と *datalen* は無視されます。

**戻り値**

ct_param は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

**例**

このコードは、*rpc.c* サンプル・プログラムからの抜粋です。

```
/*
 ** BuildRpcCommand()
 **
 ** Purpose:
 **     Builds an RPC command but does not send it.
 **
 */

CS_STATIC CS_RETCODE
 BuildRpcCommand(cmd)
 CS_COMMAND       *cmd;
 {
       CS_CONNECTION     *connection;
       CS_CONTEXT        *context;
       CS_RETCODE        retcode;
       CS_DATAFMT        datafmt;
       CS_DATAFMT        srcfmt;
```

```
 CS_DATAFMT       destfmt;
 CS_INT           intvar;
 CS_SMALLINT      smallintvar;
 CS_FLOAT         floatvar;
 CS_MONEY         moneyvar;
 CS_BINARY        binaryvar;
 char             moneystring[10];
 char             rpc_name[15];
 CS_INT           destlen;

/*
 ** Assign values to the variables used for
 ** parameter passing.
 */
 intvar = 2;
 smallintvar = 234;
 floatvar = 0.12;
 binaryvar = (CS_BINARY)0xff;
 strcpy(rpc_name, "sample_rpc");
 strcpy(moneystring, "300.90");
 /*
 ** Clear and setup the CS_DATAFMT structures used
 ** to convert datatypes.
 */
 memset(&srcfmt, 0, sizeof (CS_DATAFMT));
 srcfmt.datatype = CS_CHAR_TYPE;
 srcfmt.maxlength = strlen(moneystring);
 srcfmt.precision = 5;
 srcfmt.scale = 2;
 srcfmt.locale = NULL;

memset(&destfmt, 0, sizeof (CS_DATAFMT));
 destfmt.datatype = CS_MONEY_TYPE;
 destfmt.maxlength = sizeof(CS_MONEY);
 destfmt.precision = 5;
 destfmt.scale = 2;
 destfmt.locale = NULL;

/*
 ** Convert the string representing the money value
 ** to a CS_MONEY variable. Since this routine
 ** does not have the context handle, we use the
 ** property functions to get it.
 */
 if ((retcode = ct_cmd_props(cmd, CS_GET,
      CS_PARENT_HANDLE, &connection, CS_UNUSED,
      NULL)) != CS_SUCCEED)
         ...error checking deleted ...
```

```
 if ((retcode = ct_con_props(connection, CS_GET,
      CS_PARENT_HANDLE, &context, CS_UNUSED,
      NULL)) != CS_SUCCEED)
        ...error checking deleted ...

retcode = cs_convert(context, &srcfmt,
      (CS_VOID *)moneystring, &destfmt, &moneyvar,
      &destlen);
 if (retcode != CS_SUCCEED)
        ...error checking deleted ...
 /*
 ** Initiate the RPC command for our stored
 ** procedure.
 */
 if ((retcode = (cmd, CS_RPC_CMD,
      rpc_name, CS_NULLTERM, CS_NO_RECOMPILE)) !=
      CS_SUCCEED)
          ...error checking deleted ...

/*
 ** Clear and set up the CS_DATAFMT structure, then
 ** pass each of the parameters for the RPC.
 */
 memset(&datafmt, 0, sizeof (datafmt));
 strcpy(datafmt.name, "@intparam");
 datafmt.namelen = CS_NULLTERM;
 datafmt.datatype = CS_INT_TYPE;
 datafmt.maxlength = CS_UNUSED;
 datafmt.status = CS_INPUTVALUE;
 datafmt.locale = NULL;

if ((retcode = ct_param(cmd, &datafmt,
      (CS_VOID *)&intvar, sizeof(CS_INT),0))
     != CS_SUCCEED)
       ...error checking deleted ...

strcpy(datafmt.name, "@sintparam");
 datafmt.namelen = CS_NULLTERM;
 datafmt.datatype = CS_SMALLINT_TYPE;
 datafmt.status = CS_RETURN;
 datafmt.locale = NULL;
 if ((retcode = ct_param(cmd, &datafmt,
      (CS_VOID *)&smallintvar,
      sizeof(CS_SMALLINT), 0))
     != CS_SUCCEED)
 ...error checking deleted ...

strcpy(datafmt.name, "@floatparam");
 datafmt.namelen = CS_NULLTERM;
```

```
 datafmt.datatype = CS_FLOAT_TYPE;
 datafmt.status = CS_RETURN;
 datafmt.locale = NULL;
 if ((retcode = ct_param(cmd, &datafmt,
      (CS_VOID *)&floatvar,sizeof(CS_FLOAT),0))
     != CS_SUCCEED)
        ...error checking deleted ...

strcpy(datafmt.name, "@moneyparam");
 datafmt.namelen = CS_NULLTERM;
 datafmt.datatype = CS_MONEY_TYPE;
 datafmt.status = CS_RETURN;
 datafmt.locale = NULL;
 if ((retcode = ct_param(cmd, &datafmt,
      (CS_VOID *)&moneyvar, sizeof(CS_MONEY),0))
     != CS_SUCCEED)
     ...error checking deleted ...

 strcpy(datafmt.name, "@dateparam");
 datafmt.namelen = CS_NULLTERM;
 datafmt.datatype = CS_DATETIME4_TYPE;
 datafmt.status = CS_RETURN;
 datafmt.locale = NULL;
 /*
 ** The datetime variable is filled in by the RPC
 ** so pass NULL for the data, 0 for data length,
 ** and -l for the indicator arguments.
 */
 if((retcode = ct_param(cmd, &datafmt, NULL, 0,
      -1)) != CS_SUCCEED)
            ...error checking deleted ...

strcpy(datafmt.name, "@charparam");
 datafmt.namelen = CS_NULLTERM;
 datafmt.datatype = CS_CHAR_TYPE;
 datafmt.maxlength = EX_MAXSTRINGLEN;
 datafmt.status = CS_RETURN;
 datafmt.locale = NULL;
 /*
 ** The character string variable is filled in by
 ** the RPC so pass NULL for the data 0 for data
 ** length, and -l for the indicator arguments.
 */
 if((retcode = ct_param(cmd, &datafmt, NULL, 0,
      -1)) != CS_SUCCEED)
         ...error checking deleted ...

strcpy(datafmt.name, "@binaryparam");
```

```
        datafmt.namelen = CS_NULLTERM;
        datafmt.datatype = CS_BINARY_TYPE;
        datafmt.maxlength = EX_MAXSTRINGLEN;
        datafmt.status = CS_RETURN;
        datafmt.locale = NULL;
        if ((retcode = ct_param(cmd, &datafmt,
             (CS_VOID *)&binaryvar,
             sizeof(CS_BINARY), 0))
            != CS_SUCCEED)
              ...error checking deleted ...

    return retcode;
}
```

**使用法**

表 3-46 に ct_param の使用法を示します。

**表 3-46：ct_param パラメータの一覧**

| コマンドのタイプ | ct_param の目的 | datafmt->status | *data および datalen |
|---|---|---|---|
| カーソル宣言 | 更新カラムの識別 | CS_UPDATECOL | 更新カラムの名前とその名前の長さ |
| カーソル宣言 | ホスト変数フォーマットの定義 | CS_INPUTVALUE | NULL および CS_UNUSED |
| カーソル・オープン | パラメータ値の受け渡し | CS_INPUTVALUE | パラメータ値と長さ |
| カーソル更新 | パラメータ値の受け渡し | CS_INPUTVALUE | パラメータ値と長さ |
| 動的 SQL 実行 | パラメータ値の受け渡し | CS_INPUTVALUE | パラメータ値と長さ |
| 言語 | パラメータ値の受け渡し | CS_INPUTVALUE | パラメータ値と長さ |
| メッセージ | パラメータ値の受け渡し | CS_INPUTVALUE | パラメータ値と長さ |
| RPC | パラメータ値の受け渡し | リターン・パラメータを渡す場合は CS_RETURN、リターン・パラメータ以外のパラメータを渡す場合は CS_INPUTVALUE | パラメータ値と長さ |

- ct_param は開始されたコマンドのパラメータ値を入力します。

- 実行の最初の手順としてコマンドを開始します。一部のコマンドでは、アプリケーションで ct_param または ct_setparam を使用して入力パラメータを定義してから、ct_send を呼び出してコマンドをサーバに送信する必要があります。この機能については、「コマンドの再送信」(643 ページ ) を参照してください。
- ct_setparam と ct_param の機能は、次の点を除いて同じです。
  - ct_param はプログラム変数の内容をコピーします。
  - ct_setparam はプログラム変数のアドレスをコピーして、ct_send に対する以後の呼び出しは変数の内容を読み込みます。ct_setparam を使用することによって、アプリケーションはコマンドの再送時にパラメータ値を変更できます。

  ct_param と ct_setparam の呼び出しは混在させることができます。
- アプリケーションは次の目的で ct_param を呼び出さなければならないことがあります。
  - カーソル宣言コマンドの更新カラムを識別するため
  - カーソル宣言コマンドのホスト変数フォーマットを定義するため
  - カーソル・オープン、カーソル更新、動的 SQL 実行、言語、メッセージ、または RPC の各コマンドの入力パラメータ値を渡すため

  アプリケーションは、ct_command を呼び出して、言語、RPC、またはメッセージのコマンドを開始し、ct_cursor を呼び出して、カーソル宣言コマンドまたはカーソル・オープン・コマンドを開始し、ct_dynamic を呼び出して、動的 SQL 実行コマンドを開始します。

  これらのそれぞれの使用法については、次の各項を参照してください。

  
  - 「入力パラメータ値を渡す」(598 ページ )
  - 「ホスト変数フォーマットの定義」(597 ページ )
  - 「カーソル宣言コマンドの更新カラムの識別」(597 ページ )
  
- Client-Library は、パラメータをサーバに渡す前に、パラメータを変換しません。したがって、アプリケーションはサーバが要求するデータ型でパラメータを入力する必要があります。アプリケーションは、必要に応じて cs_convert を呼び出してパラメータ値を必要なデータ型に変換できます。

### カーソル宣言コマンドの更新カラムの識別

- カーソルは更新可能だが、すべてのカラムが更新用なっていない場合、サーバによっては、カーソル宣言コマンドに対する更新カラムの識別をクライアント・アプリケーションに要求することもあります。更新カラムは、基本となるデータベース・テーブルの値の変更に使用できます。
- Adaptive Server Enterprise は、この項で説明する別の ct_param 呼び出しや ct_setparam 呼び出しを使用した更新カラムの指定を、アプリケーションに対して要求しません。実際、Adaptive Server Enterprise はここで説明する更新カラムの識別要求を無視します。アプリケーションは、select 文に Transact-SQL の for read only 構文または for update of 構文を使用して、更新可能なカラムを指定する必要があります ( この構文については、Adaptive Server Enterprise を参照してください )。設計によっては、この項で説明するように、Open Server アプリケーションがクライアントに対してカーソルの更新カラムを指定するよう要求する場合もあります。
- カーソルのカラムのすべてが「更新用」の場合、アプリケーションはそれらを個々に指定するために ct_param を呼び出す必要はありません。
- カーソル宣言コマンドで更新カラムを識別するために、アプリケーションは *datafmt–>status* を CS_UPDATECOL、**data* をカラム名にして ct_param を呼び出します。
- カーソル宣言コマンドの更新カラムを識別するときに使用する **datafmt* のフィールドを次の表に示します。

**表 3-47：更新カラムを識別するための CS_DATAFMT フィールド**

| フィールド名 | 設定内容 |
|---|---|
| *status* | CS_UPDATECOL |

他のすべてのフィールドは無視される。

### ホスト変数フォーマットの定義

- 宣言するカーソルのテキストがホスト変数を含む SQL 文字列の場合、アプリケーションはカーソル宣言コマンドのホスト変数フォーマットを定義してください。
- ホスト変数フォーマットの定義を行うには、アプリケーションは *datafmt–>status* を CS_INPUTVALUE、*datafmt–>datatype* をホスト変数のデータ型、*data* を NULL、*datalen* を CS_UNUSED にして ct_param を呼び出します。

- アプリケーションは、カーソル宣言コマンドの間にホスト変数フォーマットを定義しますが、カーソルがオープンするまでデータ値を変数に渡しません。
- ホスト変数フォーマットを定義する場合、変数は名前付きでも名前なしでもかまいません。変数を 1 つでも名前で指定する場合は、すべての変数を名前で指定してください。変数が名前付きでない場合は、位置によって解釈されます。
- 次の表は、ホスト変数のフォーマットを定義する場合に使用する **datafmt* 内のフィールドを示します。

**表 3-48：ホスト変数フォーマットを定義するための CS_DATAFMT フィールド**

| 名前 | 設定内容 |
| --- | --- |
| *name* | ホスト変数の名前。 |
| *namelen* | *name* のバイト単位の長さ、または名前のないパラメータであることを示す 0。 |
| *datatype* | ホスト変数のデータ型。<br>CS_TEXT_TYPE、CS_UNITEXT_TYPE、CS_IMAGE_TYPE、CS_XML_TYPE、および Client-Library のユーザ定義データ型を除く、Client-Library の標準データ型はすべて有効。<br>*datatype* が CS_VARCHAR_TYPE または CS_VARBINARY_TYPE の場合、*data* は CS_VARCHAR または CS_VARBINARY 構造体を指す必要がある。 |
| *status* | CS_INPUTVALUE |

他のすべてのフィールドは無視される。

**入力パラメータ値を渡す**

- アプリケーションは、次のコマンドに対して入力パラメータ値を渡さなければならないことがあります。
  - Client-Library のカーソル・オープン・コマンド
  - Client-Library のカーソル更新コマンド
  - 動的 SQL 実行コマンド
  - 言語コマンド
  - メッセージ・コマンド
  - パッケージ・コマンド
  - RPC コマンド

- 入力パラメータ値を渡す場合、パラメータは名前付きでも名前なしでもかまいません。あるパラメータが名前付きの場合は、すべてのパラメータを名前付きにしてください。パラメータが名前付きでない場合、位置によって解釈されます。
- 場合によっては、アプリケーションは null 値を持つパラメータを渡す必要があります。たとえば、アプリケーションは、null 入力パラメータにデフォルト値を代入するストアド・プロシージャに、null 値のパラメータを渡すことがあります。

  null 値を持つパラメータを示すには、2 つの方法があります。

  - *indicator* を -1 として渡します。ct_param は *data* と *datalen* を無視します。
  - *data* を NULL、*datalen* を 0 または CS_UNUSED として渡します。
- Client-Library のカーソル・オープン・コマンドは、次の場合に入力パラメータ値が必要です。
  - カーソルの本体がホスト変数を含む SQL テキスト文字列である場合。
  - カーソルの本体が、パラメータを要するストアド・プロシージャである場合。このような場合、*datafmt–>status* は CS_INPUTVALUE になります。
  - 疑問符で示されるプレースホルダを含む準備された動的 SQL 文に関して、カーソルが宣言される場合。
- 更新コマンドを表す SQL テキストにホスト変数が含まれている場合、Client-Library のカーソル更新コマンドには入力パラメータ値が必要です。
- 実行対象の準備文に動的パラメータ・マーカが含まれている場合、動的 SQL 実行コマンドには入力パラメータ値が必要です。
- 言語コマンドのテキストにホスト変数が含まれている場合、言語コマンドには入力パラメータ値が必要です。
- メッセージでパラメータが使用されている場合、メッセージ・コマンドには入力パラメータ値が必要です。
- 実行するストアド・プロシージャまたはパッケージでパラメータが使用されている場合、RPC コマンドとパッケージ・コマンドには入力パラメータ値が必要です。

- メッセージ、パッケージ、RPC の各コマンドは、リターン・パラメータを持つことがあります。リターン・パラメータは *datafmt–>status* を CS_RETURN として渡すことで示されます。
- リターン・パラメータを使用するコマンドは、リターン・パラメータ値を含むパラメータ結果セットを生成する場合があります。アプリケーションがパラメータ結果セットから値を取得する方法については、「ct_results」を参照してください。
- 入力パラメータ値を渡す場合に使用する **datafmt* のフィールドを次の表に示します。

**表 3-49：入力パラメータ値の受け渡しにおける CS_DATAFMT フィールド**

| 名前 | 設定内容 |
|---|---|
| *name* | パラメータの名前。<br>*name* は動的 SQL 実行コマンドの場合は無視される。 |
| *namelen* | *name* のバイト単位の長さ、または名前のないパラメータであることを示す 0。<br>*namelen* は動的 SQL 実行コマンドの場合は無視される。 |
| *datatype* | 入力パラメータ値のデータ型。<br>CS_TEXT_TYPE、CS_UNITEXT_TYPE、CS_IMAGE_TYPE、CS_XML_TYPE、および Client-Library のユーザ定義データ型を除く、Client-Library の標準データ型はすべて有効。<br>*datatype* が CS_VARCHAR_TYPE または CS_VARBINARY_TYPE の場合、*data* は CS_VARCHAR または CS_VARBINARY 構造体を指す必要がある。 |
| *maxlength* | RPC コマンドにリターン・パラメータを渡す場合、*maxlength* はこのパラメータに対して返されるデータの最大バイト長を表す。<br>*maxlength* は、他のタイプのコマンドの入力パラメータ値を渡す場合には使用されない。 |
| *status* | RPC コマンドのリターン・パラメータを渡す場合は CS_RETURN、その他の場合は CS_INPUTVALUE。 |

他のすべてのフィールドは無視される。

**参照**

ct_command、ct_cursor、ct_dynamic、ct_send、ct_setparam

# ct_poll

**説明**

非同期オペレーションの完了とレジスタード・プロシージャ・ノーティフィケーションを確認するために接続のポーリングを実行します。

**構文**

```
CS_RETCODE ct_poll (context, connection, milliseconds, compconn,
                    compcmd, compid, compstatus)

CS_CONTEXT         *context;
CS_CONNECTION      *connection;
CS_INT             milliseconds;
CS_CONNECTION      **compconn;
CS_COMMAND         **compcmd;
CS_INT             *compid;
CS_RETCODE         *compstatus;
```

**パラメータ**

*context*

CS_CONTEXT 構造体を指すポインタです。

*context* か *connection* のどちらかは NULL にしてください。*context* が NULL の場合、ct_poll は 1 つの接続だけを調べます。

*connection*

CS_CONNECTION 構造体を指すポインタです。CS_CONNECTION 構造体は、特定のクライアント／サーバ接続の情報を含んでいます。

*context* か *connection* のどちらかは NULL にしてください。*connection* が NULL の場合、ct_poll は、そのコンテキストのオープンされているすべての接続を調べます。

*milliseconds*

未処理のオペレーションが完了するのを待つ、ミリ秒単位での時間です。

*milliseconds* が 0 の場合、ct_poll はただちに戻ります。ブロッキングせずにオペレーション完了をチェックするには、*milliseconds* を 0 にして渡してください。

*milliseconds* が CS_NO_LIMIT である場合、ct_poll は次の条件のいずれかを満たすまで戻りません。

- サーバ応答が到着した。これは、レジスタード・プロシージャ・ノーティフィケーションでも、非同期ルーチンの呼び出しを完了するのに必要なデータでもかまいません。
- 保留中の非同期ルーチン完了がない。ct_poll は、呼び出されたとき保留中の完了がなければ、CS_QUIET を返します ( 詳細については、「戻り値」の項を参照してください )。

- システム割り込みが発生している。

---

**注意** ct_poll はノーティフィケーション・イベントの到着を待ちません。ただし、ct_poll は、呼び出されたときに存在したノーティフィケーション・イベント、または非同期ルーチン完了を待っている間に到着したノーティフィケーション・イベントに対して、ノーティフィケーション・コールバックをトリガします。

---

*compconn*

ポインタ変数のアドレスです。*connection* が NULL の場合、すべての接続が調べられ、ct_poll は見つけた最初の完了オペレーションを管理する接続構造体を指すように **compconn* を設定します。

ct_poll が戻るまでに完了したオペレーションがない場合、ct_poll は **compconn* に NULL を設定します。

*connection* が指定された場合は、*compconn* を NULL にする必要があります。

*compcmd*

ポインタ変数のアドレスです。ct_poll は、検出した最初の完了オペレーションを管理するコマンド構造体を指すように **compcmd* を設定します。ct_poll が戻るまでに完了したオペレーションがない場合、ct_poll は **compcmd* を NULL に設定します。

*compid*

整数変数のアドレスです。ct_poll は完了したオペレーションを示すために **compid* を次の記号値のうちの 1 つに設定します。

**表 3-50：ct_poll *compid パラメータの値**

| compid の値 | 意味 |
|---|---|
| BLK_DONE | blk_done が完了した。 |
| BLK_INIT | blk_init が完了した。 |
| BLK_ROWXFER | blk_rowxfer が完了した。 |
| BLK_SENDROW | blk_sendrow が完了した。 |
| BLK_SENDTEXT | blk_sendtext が完了した。 |
| BLK_TEXTXFER | blk_textxfer が完了した。 |
| CT_CANCEL | ct_cancel が完了した。 |
| CT_CLOSE | ct_close が完了した。 |
| CT_CONNECT | ct_connect が完了した。 |
| CT_DS_LOOKUP | ct_ds_lookup が完了した。 |
| CT_FETCH | ct_fetch が完了した。 |
| CT_GET_DATA | ct_get_data が完了した。 |
| CT_NOTIFICATION | ノーティフィケーションを受信した。 |
| CT_OPTIONS | ct_options が完了した。 |
| CT_RECVPASSTHRU | ct_recvpassthru が完了した。 |
| CT_RESULTS | ct_results が完了した。 |
| CT_SEND | ct_send が完了した。 |
| CT_SEND_DATA | ct_send_data が完了した。 |
| CT_SENDPASSTHRU | ct_sendpassthru が完了した。 |
| ユーザ定義の値：この値は、CT_USER_FUNC 以上でなくてはならない。 | ユーザ定義の関数が完了した。 |

*compstatus*

CS_RETCODE 型の変数を指すポインタです。ct_poll は完了オペレーションの最終リターン・コードを示すために **compstatus* を設定します。この値は、同じ状態のルーチンに対する同期呼び出しが返す値に対応します。これは、CS_PENDING を除いた、各ルーチンが返すリターン・コードのいずれかです。

戻り値

ct_poll は、次の値を返します。

**表 3-51：ct_poll の戻り値**

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | オペレーションが正常に終了した。 |
| CS_FAIL | エラーが発生した。 |
| CS_TIMED_OUT | *milliseconds* で指定されたタイムアウト値が、オペレーションが完了する前に経過した。<br>非同期オペレーションが動作中のことがある。 |
| CS_QUIET | ct_poll がただちに戻ることを示すために、*milliseconds* に 0 が渡された。<br>実行中の非同期オペレーションはない。完了オペレーションまたはレジスタード・プロシージャ・ノーティフィケーションは見つからなかった。 |
| CS_INTERRUPT | システム割り込みが発生した。 |

接続が 「dead」状態の場合、ct_poll は CS_FAIL を返します。

**例**

```
/*
 ** BusyWait()
 **
 ** Type of function:
 ** async example program api
 **
 ** Purpose:
 **   Silly routine that prints out dots while waiting
 **   for an async operation to complete. It demonstrates
 **   the ability to do other work while an async
 **   operation is pending.
 **
 ** Returns:
 **   completion status.
 **
 ** Side Effects:
 **      None
 */

 CS_STATIC CS_RETCODE CS_INTERNAL
 BusyWait(connection, where)
 CS_CONNECTION   *connection;
 char            *where;
 {
    CS_COMMAND      *compcmd;
    CS_INT          compid;
    CS_RETCODE      compstat;
    CS_RETCODE      retstat;
```

```
    fprintf(stdout, "¥nWaiting [%s]", where);
    fflush(stdout);
    do
    {
      fprintf(stdout, ".");
      fflush(stdout);
      retstat = ct_poll(NULL, connection, 100, NULL, &compcmd,
                        &compid, &compstat);
      if (retstat != CS_SUCCEED
          && retstat != CS_TIMED_OUT
          && retstat != CS_INTERRUPT)
      {
        fprintf(stdout,
                "¥nct_poll returned unexpected status of %d¥n",
                retstat);
        fflush(stdout);
        break;
      }
    } while (retstat != CS_SUCCEED);

    if (retstat == CS_SUCCEED)
    {
      fprintf(stdout,
        "¥nct_poll completed:compid = %ld, compstat = %ld¥n",
              compid, compstat);
      fflush(stdout);
    }

    return compstat;
}
```

このコードは、*ex_amain.c* サンプル・プログラムからの抜粋です。

**使用法**

表 3-52 に ct_poll の使用法を示します。

**表 3-52：ct_poll パラメータの一覧**

| コンテキスト | 接続 | compconn | 結果 |
|---|---|---|---|
| NULL | 値を持たなければならない。 | NULL でなければならない。 | *connection* によって指定された 1 つの接続を調べる。 |
| 値を持つ | NULL でなければならない。 | 値を持たなければならない。 | このコンテキスト内のすべての接続を調べる。検出した最初の完了オペレーションを管理する接続を指すように **compconn* を設定する。 |

- ct_poll は、特定の接続または特定のコンテキストのすべての接続のポーリングを実行します。

---

**注意**　Client-Library がシグナルを使用して非同期ネットワーク I/O を処理するプラットフォームの場合、アプリケーションのコールバック・ルーチンはシステム割り込みレベルで実行できます。

ct_poll は、Client-Library のコールバック関数、またはシステム割り込みレベルで実行できるその他の関数からは呼び出さないでください。

ct_poll をシステム割り込みレベルで呼び出すと、Open Client/Server 内部リソースが破壊され、アプリケーション内で予定外の再帰が発生します。

---

- プラットフォームが割り込み駆動型またはスレッド駆動型 I/O を提供しない場合、アプリケーションはネットワークから定期的に読み込んで、非同期オペレーションの完了とレジスタード・プロシージャ・ノーティフィケーションを調べる必要があります。

  CS_PENDING を返すことができるすべてのルーチンは、ネットワークから読み込みを行います。アプリケーションがこれらのルーチンを実際に使用していない場合、非同期オペレーションの完了とレジスタード・プロシージャ・ノーティフィケーションを調べるには ct_poll を呼び出す必要があります。

  - 非同期オペレーションの完了を認識するために、ct_poll を定期的に呼び出す必要があります。ct_poll は、完了したルーチンと非同期オペレーションの完了ステータスをレポートします。完了が発生した接続に完了コールバックがインストールされている場合には、ct_poll によって完了コールバックが呼び出されます。
  - レジスタード・プロシージャ・ノーティフィケーションの場合、アプリケーションは、( 結果を処理するコマンドを送信する通常のプロセスの一環として ) 接続からの読み込みを続けるか、または ct_poll を呼び出して Client-Library にノーティフィケーション・イベントを認識させることができます。ノーティフィケーション・コールバックは、ct_poll または接続から読み込んでいる任意のルーチンから呼び出すことができます。接続上でコマンドを積極的に送信したり結果を処理したりしないアプリケーションの場合には、ノーティフィケーション・イベントを受信するために、ct_poll を使用して接続をポーリングする必要があります。

- CS_ASYNC_NOTIFS が CS_FALSE の場合、ct_poll はネットワークからの読み込みを行いません。これは、アプリケーションが ct_poll の結果を読み込んで、レジスタード・プロシージャ・ノーティフィケーションを通知する必要があることを意味しています。
- コールバック関数を使用できるプラットフォームでは、ct_poll は完了オペレーションまたはノーティフィケーションを検出すると、適切なコールバック・ルーチンがインストールされていれば、それを自動的に呼び出します。
- CS_DISABLE_POLL プロパティが CS_TRUE に設定されている場合、ct_poll は非同期オペレーションの完了をチェックしません。
- 保留中の非同期オペレーションがない場合は、*milliseconds* の値に関係なく ct_poll がすぐに戻ります。

**参照**

「非同期プログラミング」(12 ページ)、「コールバック」(25 ページ)、ct_callback、ct_wakeup

# ct_recvpassthru

**説明**

サーバから TDS (Tabular Data Stream) パケットを受信します。

**構文**

```
CS_RETCODE ct_recvpassthru (cmd, recvptr)

CS_COMMAND   *cmd;
CS_VOID      **recvptr;
```

**パラメータ**

*cmd*

CS_COMMAND 構造体を指すポインタです。

*recvptr*

ポインタ変数のアドレスです。ct_recvpassthru は最後に受信した TDS パケットを含むバッファのアドレスを、この変数に設定します。アプリケーションがこのバッファを割り付ける必要はありません。

**戻り値**

ct_recvpassthru は、次の値を返します。

**表 3-53 : ct_recvpassthru の戻り値**

| 戻り値 | 意味 |
|---|---|
| CS_PASSTHRU_MORE | パケットを正常に受信した。パケットがさらに存在する。 |
| CS_PASSTHRU_EOM | パケットを正常に受信した。パケットはこれ以上存在しない。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_CANCELED | パススルー・オペレーションがキャンセルされた。 |
| CS_PENDING | 非同期ネットワーク I/O が有効。「非同期プログラミング」(12 ページ ) を参照。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

**使用法**

- TDS は、クライアントとサーバの間で要求および要求結果の転送に使用する通信プロトコルです。一般的な状況下では、Client-Library がデータ・ストリームを管理するので、ゲートウェイ・アプリケーション以外は通常、TDS を取り扱う必要はありません。
- ct_recvpassthru と ct_sendpassthru はゲートウェイ・アプリケーションで役立ちます。アプリケーションが両者 ( クライアントとリモート・サーバ、または 2 つのサーバなど ) の仲介の役目をするとき、TDS ストリームをあるサーバから他のサーバに渡すためにこれらのルーチンを使用して、情報の解釈と情報の再コード化のプロセスを省略できます。
- ct_recvpassthru はバイト・パケットをサーバ接続から読み込み、そのバイトを格納するバッファを指すように **recvptr* を設定します。
- デフォルト・パケット・サイズはプラットフォームによって異なります。ほとんどのプラットフォームでは、パケットのデフォルト・サイズは 512 バイトです。接続では、ct_con_props を通じてそのパケット・サイズを変更できます。
- サーバによって EOM (End Of Message) というマークが TDS パケットに付けられていた場合、ct_recvpassthru は CS_PASSTHRU_EOM を返します。TDS パケットにこのマークが付いていない場合、ct_recvpassthru は CS_PASSTHRU_MORE を返します。
- パススルー・オペレーションに使用されている接続は、CS_PASSTHRU_EOM が受信されるまで、他の Client-Library 関数では使用できません。

**参照**

ct_getloginfo、ct_sendpassthru、ct_setloginfo

# ct_remote_pwd

**説明**

サーバ間接続に使用するパスワードを定義またはクリアします。

**構文**

```
CS_RETCODE ct_remote_pwd(connection, action,
          server_name, snamelen, password,
          pwdlen)

CS_CONNECTION    *connection;
CS_INT           action;
CS_CHAR          *server_name;
CS_INT           snamelen;
CS_CHAR          *password;
CS_INT           pwdlen;
```

**パラメータ**

*connection*

CS_CONNECTION 構造体を指すポインタです。CS_CONNECTION 構造体は、特定のクライアント／サーバ接続の情報を含んでいます。

オープンされている接続のリモート・パスワードを定義することは無効です。

*action*

次の記号値のいずれかです。

| action の値 | 結果 |
|---|---|
| CS_SET | リモート・パスワードを設定する。 |
| CS_CLEAR | この接続に指定されたすべてのリモート・パスワードを NULL に設定してクリアする。 |

*server_name*

パスワードを定義するサーバ名を指すポインタです。**server_name* は、*interfaces* ファイルでそのサーバに対して指定されている名前です。

*server_name* が NULL の場合、指定したパスワードはユニバーサル・パスワードとみなされ、明示的に指定されたパスワードを持たないサーバに使用されます。

*action* が CS_CLEAR の場合、*server_name* は NULL にしてください。

*snamelen*

**server_name* のバイト単位の長さです。**server_name* が null で終了する場合、*snamelen* を CS_NULLTERM にして渡してください。

*action* が CS_SET で、*server_name* が NULL の場合、*snamelen* を 0 または CS_UNUSED にして渡してください。

*action* が CS_CLEAR の場合、*snamelen* は CS_UNUSED にしてください。

*password*

**server_name* サーバへのリモート・ログイン用にインストールされたパスワードを指すポインタです。

*action* が CS_CLEAR の場合、*password* を NULL にしてください。

*pwdlen*

**password* のバイト単位の長さです。**password* が null で終了する場合、*pwdlen* を CS_NULLTERM にして渡してください。

*action* が CS_SET で、*password* が NULL の場合、*pwdlen* を 0 または CS_UNUSED にして渡してください。

*action* が CS_CLEAR の場合、*pwdlen* は CS_UNUSED にしてください。

**戻り値**

ct_remote_pwd は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ) を参照。 |

**使用法**

- ct_remote_pwd はサーバが別のサーバにログインするとき使用するパスワードを定義します。
- サーバ上で実行される Transact-SQL 言語コマンドやストアド・プロシージャから、別のサーバ上にあるストアド・プロシージャを実行できます。このサーバ間通信では、アプリケーションから ct_connect によって接続した最初のサーバが、第 2 のリモート・サーバに実際にログインしてサーバ間のリモート・プロシージャ・コールを実行します。

  ct_remote_pwd により、アプリケーションは、最初のサーバがリモート・サーバにログインするとき使用するパスワードを指定できます。
- ログインする必要のあるサーバごとに 1 つずつ、複数のパスワードが指定されることがあります。各パスワードは、それぞれ ct_remote_pwd を個別に呼び出して定義します。
- アプリケーションは、NULL の *server_name* と *password* 値を指定した ct_remote_pwd を呼び出して、サーバ間通信にユニバーサル・パスワードを指定できます。接続がオープンされると、その接続のサーバは、ct_remote_pwd によってサーバ名／パスワードのペアが指定されていないリモート・サーバに、このパスワードを使用してログインします。

アプリケーションがリモート・サーバのパスワードを指定しなかった場合、Client-Library は、サーバ間通信用のデフォルトのユニバーサル・パスワードとして接続パスワードを送信します。接続パスワードは ct_con_props(CS_PASSWORD) によって設定されます。デフォルトは NULL です。したがって、アプリケーション・ユーザが、異なるサーバで同じパスワードを持っている場合、アプリケーションは ct_remote_pwd を呼び出す必要はありません。

ただし、アプリケーションが特定のサーバについてパスワードを指定した場合は、ユニバーサル・パスワードを明示的に定義する必要があります。たとえば、次のコードでは、サーバ「honey_tree」用のパスワードとして「tigger2」を指定し、その他のリモート・サーバすべてに対して使用するユニバーサル・パスワードとして「christopher」を指定しています。

```
/*
 ** User's password is "tigger2" on the "honey_tree" server.
 */
 retcode = ct_remote_pwd(conn, CS_SET, "honey_tree", CS_NULLTERM,
                         "tigger2", CS_NULLTERM);
 if (retcode != CS_SUCCEED)
        ... handle the error ...

 /*
 ** User's password is "christopher" everywhere else.
 */
 retcode = ct_remote_pwd(conn, CS_SET, (CS_CHAR *) NULL, 0
                         "christopher", CS_NULLTERM);
 if (retcode != CS_SUCCEED)
        ... handle the error ...
```

- リモート・パスワードを保管する内部バッファは 255 バイトのサイズしかありません。このバッファ内の各パスワード・エントリは、パスワード自体と、関連するサーバ名、および追加の 2 バイトで構成されます。このバッファに対するパスワードの追加でオーバフローが発生した場合、ct_remote_pwd は CS_FAIL を返し、問題を示す Client-Library エラー・メッセージを生成します。
- すでにオープンされている接続のリモート・パスワードを定義するために ct_remote_pwd を呼び出すとエラーになります。アクティブな接続を作成するには、リモート・パスワードを定義してから、ct_connect を呼び出してください。
- アプリケーションは、接続のリモート・パスワードをクリアするためにいつでも ct_remote_pwd を呼び出すことができます。

**参照**　ct_con_props、ct_connect

# ct_res_info

**説明**

現在の結果セット情報またはコマンド情報を取得します。

**構文**

```
CS_RETCODE ct_res_info(cmd, type, buffer, buflen,
           outlen)

CS_COMMAND   *cmd;
CS_INT       type;
CS_VOID      *buffer;
CS_INT       buflen;
CS_INT       *outlen;
```

**パラメータ**

*cmd*

クライアント／サーバ・コマンドを管理する CS_COMMAND 構造体を指すポインタです。

*type*

返される情報のタイプです。表 3-54 に、*type* の記号値を示します。

*buffer*

ct_res_info が、要求された情報を入れる領域を指すポインタです。

*buflen* が、要求された情報を保持するだけの **buffer* がないことを示すと、ct_res_info は **outlen* を要求された情報の長さに設定し、CS_FAIL を返します。

*buflen*

**buffer* データ領域のバイト単位の長さか、**buffer* が固定長または記号値を表す場合は CS_UNUSED です。

*outlen*

整数変数を指すポインタです。

cs_dt_info は、**outlen* を、要求された情報の長さ ( バイト単位 ) に設定します。

要求された情報が *buflen* バイトよりも大きい場合、アプリケーションは **outlen* の値を使用して、情報の保持に必要なバイト数を判別します。

**戻り値**

ct_res_info は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

要求された情報が *buflen* バイトより大きい場合、または現在の結果セットがない場合、ct_res_info は CS_FAIL を返します。

**例**

このコードは、*rpc.c* サンプル・プログラムからの抜粋です。フェッチ可能な結果セット内のカラム数を取得します。

```
CS_INT    num_cols;

/*
 ** Determine the number of columns in this result
 ** set.
 */
 retcode = ct_res_info(cmd, CS_NUMDATA, #_cols,
      CS_UNUSED, NULL);
 if (retcode != CS_SUCCEED)
 {
      ...CODE DELETED...
 }
```

このコードは、*rpc.c* サンプル・プログラムからの抜粋です。メッセージ結果からメッセージ識別子を取得します。

```
CS_SMALLINT msg_id;

... ct_results has returned with a CS_MSG_RESULT
     result type ...

case CS_MSG_RESULT:
      retcode = ct_res_info(cmd, CS_MSGTYPE,
           (CS_VOID *)&msg_id, CS_UNUSED, NULL);
      if (retcode != CS_SUCCEED)
      {
           ...CODE DELETED...
      }
      fprintf(stdout, "ct_result returned ¥
           CS_MSG_RESULT where msg id = %d.¥n", msg_id);
      break;
```

**使用法**

表 3-54 に ct_res_info の使用法を示します。

**表 3-54：ct_res_info パラメータの一覧**

| type の値 | ct_res_info によって *buffer に返される値 | ct_results が *result_type パラメータを次の値に設定した後に情報が使用可能 | *buffer のデータ型 |
|---|---|---|---|
| CS_BROWSE_INFO | ブラウズ・モード情報が利用できる場合は CS_TRUE、利用できない場合は CS_FALSE。 | CS_ROW_RESULT | CS_BOOL |

| type の値 | ct_res_info によって *buffer に返される値 | ct_results が *result_type パラメータを次の値に設定した後に情報が使用可能 | *buffer のデータ型 |
|---|---|---|---|
| CS_CMD_NUMBER | 現在の結果セットを生成したコマンドの番号。 | 任意の値 | CS_INT |
| CS_MSGTYPE | 現在の結果セットを構成するメッセージの ID を表す整数。 | CS_MSG_RESULT | CS_USHORT |
| CS_NUM_COMPUTES | 現在のコマンド内の compute 句の数。 | CS_COMPUTE_RESULT | CS_INT |
| CS_NUMDATA | 現在の結果セット内の項目数。 | CS_COMPUTE_RESULT、CS_COMPUTEFMT_RESULT、CS_CURSOR_RESULT、CS_DESCRIBE_RESULT、CS_PARAM_RESULT、CS_ROW_RESULT、CS_ROWFMT_RESULT、CS_STATUS_RESULT | CS_INT |
| CS_NUMORDERCOLS | 現在のコマンドの order-by 句に指定されたカラム数。 | CS_ROW_RESULT | CS_INT |
| CS_ORDERBY_COLS | 現在のコマンドの order by 句に指定されたカラムの select リスト ID 番号。 | CS_ROW_RESULT | CS_INT の配列 |
| CS_ROW_COUNT | 現在のコマンドによって影響を受けたロー数。 | CS_CMD_DONE、CS_CMD_FAIL、CS_CMD_SUCCEED | CS_INT |
| CS_TRANS_STATE | 現在のサーバのトランザクション・ステータス。 | 任意の値。CT_RESULTS は CS_SUCCEED を返す必要がある。 | CS_INT |

- ct_res_info は、現在の結果セットまたは現在のコマンドについての情報を返します。現在のコマンドは、現在の結果セットを生成したコマンドとして定義されます。
- 結果セットは 1 つのタイプの結果データの集合です。結果セットはコマンドによって生成されます。詳細については、ct_results のリファレンス・ページと 「結果」(280 ページ ) を参照してください。
- 一般に、アプリケーションは、結果セット内の項目数を確認する目的で、*type* に CS_NUMDATA を指定して ct_res_info を呼び出します。

#### ブラウズ・モード情報が利用できるかどうかの判断

- ブラウズ・モード情報が利用できるかどうかを判断するには、*type* を CS_BROWSE_INFO にして ct_res_info を呼び出してください。
- ブラウズ・モード情報が利用できる場合、アプリケーションはその情報を取得するために ct_br_column と ct_br_table を呼び出すことができます。ブラウズ・モード情報が利用できない場合、ct_br_column または ct_br_table を呼び出すと Client-Library エラーになります。
- 「ブラウズ・モード」(22 ページ ) を参照してください。

#### 現在の結果のコマンド番号の取得

- 現在の結果を生成したコマンドの番号を確認するには、*type* に CS_CMD_NUMBER を指定して ct_res_info を呼び出します。
- Client-Library は、ct_results が CS_CMD_DONE を返す回数をカウントすることによって、コマンド番号を記録します。

  ct_send の呼び出しに続く、アプリケーションからの ct_results の最初の呼び出しでは、コマンド番号が 1 に設定されます。これ以降、コマンド番号は、ct_results が CS_CMD_DONE を返した後呼び出されるたびに増加します。
- CS_CMD_NUMBER は次の場合に役立ちます。
  - 言語コマンドにおいて、現在の結果セットを生成した Transact-SQL コマンドを確認する場合
  - カーソル・コマンドのバッチにおいて、現在の結果セットを生成したカーソル・コマンドを確認する場合
  - ストアド・プロシージャにおいて、現在の結果セットを生成した select コマンドを確認する場合
- 言語コマンドは、Transact-SQL テキストを含みます。このテキストは 1 つ以上の Transact-SQL コマンドを表します。言語コマンドで使用した場合、「コマンド番号」はその言語コマンド内の Transact-SQL コマンドの番号を指します。

  次に例を示します。

```
select * from authors
 select * from titles
 insert newauthors
      select *
      from authors
      where city = "San Francisco"
```

この文字列は、3 つの Transact-SQL コマンドを表しており、そのうち 2 つは結果セットを生成できます。この場合、ct_res_info が返すコマンド番号は、ct_res_info がいつ呼び出されるかに応じて、1 ～ 3 までになります。

- ストアド・プロシージャの内部で、select 文だけがコマンド番号を増加させます。ストアド・プロシージャが Transact-SQL コマンドを 7 つ持ち、そのうち 3 つが select である場合、ct_res_info が返すコマンド番号は、現在の結果セットを生成した select に応じて、1 ～ 3 までの整数のいずれかになります。
- ct_cursor はカーソル・コマンドを開始するために使用されます。いくつかのカーソル・コマンドは、サーバに送信する前にバッチとして定義できます。カーソル・コマンド・バッチで使用した場合、「コマンド番号」はそのコマンド・バッチ内のカーソル・コマンドの番号を指します。

  たとえば、アプリケーションは次の呼び出しを行うことができます。

  ```
  ct_cursor(...CS_CURSOR_DECLARE...);
   ct_cursor(...CS_CURSOR_ROWS...);
   ct_cursor(...CS_CURSOR_OPEN...);
   ct_send();
  ```

  ct_res_info が返すコマンド番号は、現在の結果タイプを生成したカーソル・コマンドに応じて、1 ～ 3 になります。

#### メッセージ ID の取得

- メッセージ ID を取得するには、*type* に CS_MSGTYPE を指定して ct_res_info を呼び出します。
- サーバは、クライアント・アプリケーションにメッセージを送信できます。メッセージは「メッセージ結果セット」の形式で受信されます。メッセージ結果セットはフェッチ可能なデータを含みませんが、代わりに ID 番号を持ちます。
- メッセージはパラメータを持つ場合もあります。メッセージ・パラメータは、メッセージ結果セットの直後にパラメータ結果セットとしてアプリケーションに返されます。

#### *compute* 句の数の取得

- 現在の結果セットを生成したコマンド内の compute 句の数を確認するには、*type* に CS_NUM_COMPUTES を指定して ct_res_info を呼び出します。

- Transact-SQL select 文には、計算結果セットを生成する compute 句を含めることができます。

### 結果データ項目数の取得

- 現在の結果セット内の結果データ項目数を確認するには、*type* を CS_NUMDATA にして ct_res_info を呼び出します。
- 結果セットには結果データ項目が含まれます。ロー、カーソル、および計算結果セットにはカラムが、パラメータ結果セットにはパラメータが、ステータス結果セットにはステータスがそれぞれ含まれます。カラム、パラメータ、およびステータスは「**結果データ項目**」として知られています。
- メッセージ結果セットにはどんなデータ項目も含まれません。

### *order by* 句内のカラム数の取得

- Transact-SQL select 文の order by 句に含まれているカラム数を確認するには、*type* に CS_NUMORDERCOLS を指定して ct_res_info を呼び出します。
- Transact-SQL select 文には、select 文によって取得されたローの表示順序を決定する order by 句を含めることができます。

### order-by カラムのカラム ID の取得

- order-by カラムの select リスト・カラム ID を得るには、*type* を CS_ORDERBY_COLS にして ct_res_info を呼び出します。
- select 文の order by 句で指定するカラムは、select リストでも指定する必要があります。select リストのカラムには、リストに表示される番号である select リスト ID が付いています。たとえば、次のクエリでは、au_lname と au_fname は 1 と 2 の select リスト ID をそれぞれ持ちます。

```
select au_lname, au_fname from authors
     order by au_fname, au_lname
```

- 上記のクエリの後に、次のクエリを実行したとします。

```
ct_res_info(cmd, CS_ORDERBY_COLS, myspace, 8,
     outlength)
```

  この呼び出しの結果、**myspace* に、整数 2 と 1 を持つ 2 つの CS_INT 値の配列が設定されます。

### 現在のコマンドにおけるロー数の取得

- 現在のコマンドによって影響を受けたか返されたロー数を確認するには、*type* を CS_ROW_COUNT にして ct_res_info を呼び出します。

- ct_results が **result_type* パラメータに CS_CMD_SUCCEED、CS_CMD_DONE、または CS_CMD_FAIL を設定した後、アプリケーションはロー・カウントを取得できます。ロー・カウントは、ct_results が **result_type* に CS_CMD_DONE を設定した場合だけ、正しいことが保証されます。
- アドホック・クエリ入力が可能なアプリケーションは、結果を処理するとき、影響を受けたローを通知するメッセージを (isql クライアント・アプリケーションの場合と同じように ) 出力することが必要な場合があります。そのためには、アプリケーションは、ct_results が CS_CMD_DONE *result_type* 値を示したときに、次の処理を実行する必要があります。

  a ct_res_info(CS_ROW_COUNT) でロー数を取得します。

  b ロー数が CS_NO_COUNT ではない場合、その値を出力します。

  データを修正するコマンド (insert 文や update 文など ) のロー数だけが必要な場合、アプリケーションは、ct_results が CS_CMD_SUCCEED *result_type* 値を示したときに、上記の手順を実行します。
- Transact-SQL exec 言語コマンドやリモート・プロシージャ・コール・コマンドのように、コマンドがストアド・プロシージャを実行するものである場合、ct_res_info は、ローに影響を与えるストアド・プロシージャの最後の文から影響を受けたロー数を **buffer* に設定します。
- 次のいずれかの場合、ct_res_info は **buffer* に CS_NO_COUNT を設定します。
  - Transact-SQL コマンドが、構文エラーなど、何らかの理由で失敗する場合
  - コマンドが、Transact-SQL print コマンドのようにローに影響を及ぼさないものである場合
  - コマンドがどのローにも影響を及ぼさないストアド・プロシージャを実行する場合
  - CS_OPT_NOCOUNT オプションがオンの場合

**現在のサーバのトランザクション・ステータスの取得**

- 現在のサーバのトランザクション・ステータスを確認するには、*type* を CS_TRANS_STATE にして ct_res_info を呼び出します。

**参照**

ct_cmd_props、ct_con_props、ct_results、「オプション」(200 ページ )、「サーバ・トランザクション・ステータス」(146 ページ )

# ct_results

**説明**

結果データを処理対象として設定します。

**構文**

```
CS_RETCODE ct_results(cmd, result_type)

CS_COMMAND      *cmd;
CS_INT          *result_type;
```

**パラメータ**

*cmd*

クライアント／サーバ・オペレーションを管理する CS_COMMAND 構造体を指すポインタです。

*result_type*

ct_results が結果の現在タイプを示すために設定する整数変数を指すポインタです。

次の表に **result_type* に使用できる値を示します。

**表 3-55：ct_results *result_type パラメータの値**

| 結果カテゴリー | *result_type の値 | 意味 | 結果セットの内容 |
|---|---|---|---|
| コマンド・ステータスを示す値 | CS_CMD_DONE | 論理コマンドの結果の処理が完了した。 | 適用されない。 |
| | CS_CMD_FAIL | コマンド実行中にサーバでエラーが発生した。 | 結果なし。 |
| | CS_CMD_SUCCEED | Transact-SQL insert 文を含む言語コマンドなどの、データを返さないコマンドが正常に終了した。 | 結果なし。 |
| フェッチ可能な結果を示す値 | CS_COMPUTE_RESULT | 計算ロー結果。 | 計算結果の 1 つのロー。 |
| | CS_CURSOR_RESULT | ct_cursor カーソル・オープン・コマンドからのカーソル・ロー結果。 | 表形式データの 0 個以上のロー。 |
| | CS_PARAM_RESULT | リターン・パラメータ結果。 | リターン・パラメータの 1 つのロー。 |
| | CS_ROW_RESULT | 通常ロー結果。 | 表形式データの 0 個以上のロー。 |
| | CS_STATUS_RESULT | ストアド・プロシージャ・リターン・ステータス結果。 | 1 つのステータスを含んでいる 1 つのロー。 |

| 結果カテゴリー | *result_type の値 | 意味 | 結果セットの内容 |
|---|---|---|---|
| 情報が利用できることを示す値 | CS_COMPUTEFMT_RESULT | 計算フォーマット情報。 | フェッチ可能な結果はない。アプリケーションは、現在のコマンドの今後の計算結果フォーマットを取得できる。アプリケーションは、ct_res_info、ct_describe、および ct_compute_info を呼び出して、計算フォーマット情報を取得できる。 |
| | CS_ROWFMT_RESULT | ロー・フォーマット情報。 | フェッチ可能な結果はない。アプリケーションは、ct_describe と ct_res_info を呼び出して、ロー・フォーマット情報を取得できる。 |
| | CS_MSG_RESULT | メッセージの着信。 | フェッチ可能な結果はない。アプリケーションは、ct_res_info を呼び出してメッセージ ID を取得できます。メッセージに関連するパラメータがある場合は、別のパラメータ結果セットとして返されます。 |
| | CS_DESCRIBE_RESULT | 入力記述または出力記述コマンドからの動的 SQL 記述情報。 | フェッチ可能な結果ではなく、コマンド入力または出力の記述。アプリケーションは次の方法のいずれかで結果を取得できる。<br>• ct_res_info を呼び出して項目数を取得し、ct_describe を呼び出して項目の記述を取得する。<br>• ct_dyndesc を数回呼び出して、項目数と各項目の記述を取得する。<br>• ct_res_info を呼び出して項目数を取得し、ct_dynsqlda を一度呼び出して項目の記述を取得する。 |

**戻り値**

ct_results は、次の値を返します。

**表 3-56：ct_results の戻り値**

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | 結果セットを処理に利用できる。 |
| CS_END_RESULTS | すべての結果セットが完全に処理された。 |
| CS_FAIL | ルーチンが失敗した。残っている結果は使用できない。<br>ct_results が CS_FAIL を返した場合、アプリケーションは、影響を受けたコマンド構造体を別のコマンドの送信に使用する前に、*type* を CS_CANCEL_ALL にして ct_cancel を呼び出す必要がある。<br>ct_cancel が CS_FAIL を返した場合、アプリケーションは、ct_close(CS_FORCE_CLOSE) を呼び出して接続を強制的にクローズしなければならない。 |
| CS_CANCELED | 結果がキャンセルされた。 |
| CS_PENDING | 非同期ネットワーク I/O が有効。「非同期プログラミング」(12 ページ ) を参照。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

**例**

このコードは、*compute.c* サンプル・プログラムからの抜粋です。

```
/*
** DoCompute(connection)
*/

CS_STATIC CS_RETCODE
DoCompute(connection)
CS_CONNECTION    *connection;
{

    CS_RETCODE   retcode;
    CS_COMMAND   *cmd;
    /* Result type from ct_results */
    CS_INT       res_type;

    /* Use the pubs2 database */
        ...CODE DELETED.....

    /*
    ** Allocate a command handle to send the compute
    ** query with.
    */
        ...CODE DELETED.....
```

```
/*
** Define a language command that contains a
** compute clause. SELECT is a select statment
** defined in the header file.
*/
    ...CODE DELETED.....

/* Send the command to the server */
    ...CODE DELETED.....

/*
** Process the results.
** Loop while ct_results() returns CS_SUCCEED.
*/
while ((retcode = ct_results(cmd, &res_type)) == CS_SUCCEED)
{
    switch ((int)res_type)
    {
                case CS_CMD_SUCCEED:
            /*
            ** Command returning no rows completed successfully.
            */
            break;

            case CS_CMD_DONE:
            /*
            ** This means we're done with one result set.
            */
            break;

        case CS_CMD_FAIL:
            /*
            ** This means that the server encountered
            ** an error while processing our command.
            */
            ex_error("DoCompute: ct_results() ¥ returned CMD_FAIL");
            break;

        case CS_ROW_RESULT:
            retcode = ex_fetch_data(cmd);
             if (retcode != CS_SUCCEED)
            {
                ex_error("DoCompute: ex_fetch_data()¥ failed");
                return retcode;
            }
            break;

                case CS_COMPUTE_RESULT:
            retcode = FetchComputeResults(cmd);
```

```
                if (retcode != CS_SUCCEED)
                {
                    ex_error("DoCompute: ¥ FetchComputeResults() failed");
                    return retcode;
                }
                break;

            default:
                /* We got an unexpected result type */
                ex_error("DoCompute: ct_results() ¥ returned unexpected
                            returned unexpected result type");
                return CS_FAIL;
        }
    }

    /*
    ** We've finished processing results. Let's check the return value
    ** of ct_results() to see if everything went ok.
    */
    switch ((int)retcode)
    {
        case CS_END_RESULTS:
        /* Everything went fine */
        break;

        case CS_FAIL:
            /* Something went wrong */
            ex_error("DoCompute: ct_results() ¥ failed”);
            return retcode;

        default:
            /* We got an unexpected return value */
            ex_error("DoCompute: ct_results() ¥
                        returned unexpected result code");
            return retcode;
    }

    /* Drop our command structure */
        ...CODE DELETED.....

    return retcode;
}
```

**使用法**

- ct_send を使用してコマンドをサーバに送信したあと、アプリケーションは ct_results を必要ならば何回でも呼び出します。
- コマンドがフェッチ可能な結果データを返す場合、ct_results は、コマンドから返された結果データを、アプリケーションが ct_fetch または ct_res_info を使用して読み込めるようにサーバ接続を準備します。

- 「**結果データ**」は、サーバがアプリケーションに返すことができるすべてのデータのタイプに対する総称です。データのタイプには次のものがあります。
  - 通常ロー
  - カーソル・ロー
  - リターン・パラメータ
  - ストアド・プロシージャ・リターン・ステータス番号
  - 計算ロー
  - 動的 SQL 記述情報
  - 通常ローおよび計算ローのフォーマット情報
  - メッセージ

  ct_results は、処理する結果のこれらのタイプすべてを設定するために使用します。

  ---

  **注意**　メッセージ結果をサーバ・エラーや情報メッセージと混同しないでください。エラーおよび情報メッセージの詳細については、「エラー処理」(136 ページ ) を参照してください。

  ---

- 結果データは「**結果セット**」の形式でアプリケーションに返されます。結果セットには 1 つのタイプの結果データだけが含まれます。たとえば、通常ロー結果セットには通常ローだけが含まれ、リターン・パラメータ結果セットにはリターン・パラメータだけが含まれます。

*ct_results ループ*

- コマンドは複数の結果セットを生成することがあるので、アプリケーションは、ct_results から結果の存在を示す CS_SUCCEED が返されるかぎり、ct_results を呼び出さなければなりません。
- 結果を読み込む最も単純な方法は、ct_results が CS_SUCCEED を返さない場合に終了するループを使用することです。ループから抜けた後、アプリケーションは case 文を使用して ct_results の最終リターン・コードをテストし、ループが終了した理由を確認できます。結果処理ループの論理には次の規則が適用されます。
  - 結果がアプリケーションに対して有効であるかぎり、ct_results は CS_SUCCEED を返します。

    - ct_results が、フェッチ可能な結果データを示す値に *result_type* を設定した場合、アプリケーションは、データをフェッチまたはキャンセルしてから処理を続行する必要があります。
    - 論理コマンドの結果を完全に処理し終わると、ct_results は *result_type* の値を CS_CMD_DONE に設定します。論理コマンドについては、この後の「ct_results と論理コマンド」で説明します。
    - すべての結果が正常に処理されたときには、ct_results は CS_END_RESULTS を返します。
    - アプリケーションが ct_cancel(CS_CANCEL_ALL) または ct_cancel(CS_CANCEL_ATTN) を使用して結果をキャンセルすると、ct_results は CS_CANCELED を返します。

- 結果は生成された順序でアプリケーションに返されます。ただし、この順序は常に予測できるわけではありません。たとえば、アプリケーションがあるストアド・プロシージャを呼び出し、そのプロシージャが今度は別のストアド・プロシージャを呼び出すような場合、アプリケーションは複数の通常ローおよび計算ローの結果セットをリターン・パラメータおよびリターン・ステータス結果セットとともに受信することがあります。これらの結果が返される順序はストアド・プロシージャの記述方法に依存します。

  このような理由から、アプリケーションでは、受信可能なすべての結果タイプを処理する case 文に制御が移るように ct_results ループをコーディングすることをおすすめします。リターン・パラメータ *result_type* は、結果セットに含まれる結果データのタイプを記号で示します。

- Client-Library コマンドによって生成された結果の一部が処理されなかった場合、接続は「**保留中の結果**」を持つことになります。一般的に、アプリケーションは、保留中の結果が存在する接続で新しいコマンドを送信できません。ただし、CS_CURSOR_RESULT 結果は例外です。詳細については、『Open Client Client-Library/C プログラマーズ・ガイド』の「第 7 章 Client-Library カーソルの使い方」を参照してください。

### *ct_results* と論理コマンド

- 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**」とは、ct_command、ct_dynamic、または ct_cursor によって定義されたコマンドのことです。ただし、次の場合は除きます。

- ストアド・プロシージャ内部のカラムを返す各 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。
- 言語コマンド内の各 Transact-SQL 文は論理コマンドです。

- クライアント・アプリケーションによって送信されたコマンドは、サーバで複数の論理コマンドを実行できます。
- 論理コマンドは 1 つ以上の結果セットを生成できます。
- たとえば、Client-Library 言語コマンドに次の Transact-SQL 文が含まれていると仮定します。

```
select type, price
      from titles
      order by type, price
      compute sum(price) by type

select type, price, advance
      from titles
      order by type, advance
      compute sum(price), max(advance) by type
```

この言語コマンドの結果を処理するために呼び出すと、アプリケーションは次を受け取ります。

ct_results の **result_types*

```
CS_ROW_RESULT       Row and compute results from
 CS_COMPUTE_RESULT   the first select,
 ...                 repeated as many times as the
                     value of the type column
                     changes.
 CS_CMD_DONE         Indicates that the results
                     of the first query have been
                     processed.


CS_ROW_RESULT       Row and compute results from
 CS_COMPUTE_RESULT   the second select,
 ...                 repeated as many times as the
                     value of the type column
                     changes.
 CS_CMD_DONE         Indicates that the results of
                     the second query have been
                     processed.
```

- **result_type* CS_CMD_SUCCEED または CS_CMD_FAIL の直後には、**result_type* CS_CMD_DONE が続きます。

結果のキャンセル

- コマンドの残りのすべての結果をキャンセルする ( また、CS_SUCCEED が返らなくなるまで ct_results を繰り返し呼び出す手間を省く ) には、*type* に CS_CANCEL_ALL を指定して ct_cancel を呼び出します。
- 現在の結果だけをキャンセルするには、*type* に CS_CANCEL_CURRENT を指定して ct_cancel を呼び出します。
- ct_cursor カーソル・オープン・コマンドからの不要なカーソル結果をキャンセルしないでください。このような場合は、キャンセルする代わりに ct_cursor カーソル・クローズ・コマンドでカーソルをクローズしてください。

特別な種類の結果セット

- 「**メッセージ結果セット**」には、実際の結果データは含まれません。その代わり、個々のメッセージは ID を持ちます。アプリケーションは、ct_res_info を呼び出してメッセージ ID を取得できます。メッセージは、ID の他にパラメータを持つことができます。メッセージ・パラメータは、メッセージ結果セットの直後にパラメータ結果セットとしてアプリケーションに返されます。
- 「**ロー・フォーマット**」と「**計算フォーマット**」の結果セットには、実際の結果データは含まれません。その代わり、フォーマット結果セットには関連する通常ローまたは計算ロー結果セットのフォーマット情報が含まれます。

  このタイプのフォーマット情報は、Adaptive Server Enterprise フォーマット情報を外部クライアントに送信する前に組み立て直す必要があるゲートウェイ・アプリケーションで、主に役立ちます。ct_results がフォーマット結果を示した後、ゲートウェイ・アプリケーションは、次のものを呼び出してフォーマット情報を取得できます。

  - カラム数については ct_res_info
  - 各カラムの記述については ct_describe
  - 計算ローを生成した compute 句の情報については ct_compute_info

  コマンドのフォーマット情報はすべて、どんなデータよりも先に返されます。つまり、コマンドのロー・フォーマットおよび計算フォーマット結果セットは、そのコマンドで生成された通常ローおよび計算ロー結果セットに先行します。

Client-Library の CS_EXPOSE_FMTS プロパティが CS_TRUE に設定されないかぎり、アプリケーションはフォーマット結果を受信しません。

- 「**記述結果セット**」には実際の結果データは含まれません。その代わり、記述結果セットには、動的 SQL 入力記述または出力記述コマンドによって生成された記述情報が含まれます。ct_results が記述結果を示した後、アプリケーションは次の方法のいずれかで記述を取得できます。
  - ct_res_info を呼び出して項目数を取得し、ct_describe を呼び出して各項目の記述を取得します。
  - ct_dyndesc を数回呼び出して、項目数と各項目の記述を取得します。
  - ct_res_info を呼び出して項目数を取得し、ct_dynsqlda を一度呼び出して項目の記述を取得します。

*ct_results* **とストアド・プロシージャ**

- execute 文を含んでいる言語コマンドのランタイム・エラーは、**result_type* として CS_CMD_FAIL を生成します。たとえば、execute 文の中で名前を付けられたプロシージャが見つからないときに生成されます。

  ただし、ストアド・プロシージャ内部の文の実行時エラーは CS_CMD_FAIL を生成しません。たとえば、ストアド・プロシージャに insert 文が含まれているけれども、データベース・テーブルへの挿入のパーミッションがユーザに付与されていない場合は、insert 文は失敗しますが、ct_results はこの場合も SUCCEED を返します。ストアド・プロシージャ内部のランタイム・エラーをチェックするには、ローおよびパラメータの結果の直後にストアド・プロシージャからリターン・ステータス結果セットとして返される、プロシージャのリータン・ステータス番号を調べてください。エラーがサーバ・メッセージを生成した場合、アプリケーションでも利用することができます。

*ct_results* **と CS_STICKY_BINDS プロパティ**

- 同じコマンドを繰り返し実行するアプリケーションは、CS_STICKY_BINDS プロパティを設定して、Client-Library にコマンドの初回実行時に確立した結果バインドを保存することができます。このプロパティの詳細については、「継続結果バインド」(261 ページ ) を参照してください。

- CS_STICKY_BINDS が有効になっている場合、ct_results は現在の結果セットのフォーマットを、バインドが確立したときに適用されたフォーマットと比較します。コマンドの結果フォーマット情報は、次に示す結果セットの一連の特性で構成されます。
  - ct_results ルーチンの *result_type* パラメータによってアプリケーションに示された結果タイプ。
  - ( フェッチ可能な結果のみ ) カラム数。アプリケーションは ct_res_info を使用して取得できます。
  - ( フェッチ可能な結果のみ ) 各カラムのフォーマット。アプリケーションは各カラムに対する ct_describe を使用して取得できます。
- ct_results は、フォーマットの不一致を検出すると、最初の結果シーケンス内のすべての結果セットについて保存されていたすべてのバインドをクリアします。この状態になると、ct_results は情報エラーを提示して、CS_SUCCEED を返します。フォーマットの不一致は、条件論理が含まれているコマンド ( たとえば、if 句や while 句が含まれているストアド・プロシージャなど ) を実行しているときしか発生しません。

**参照**

ct_bind、ct_command、ct_cursor、ct_describe、ct_dynamic、ct_fetch、ct_send、「結果」(280 ページ )

# ct_scroll_fetch

**説明**

サポートされるスクロール可能カーソルが宣言され、正常にオープンされた後、スクロール可能なフェッチに使用されます。

ct_scroll_fetch の機能プロパティは、対応する Adaptive Server Enterprise サーバがスクロール可能カーソルをサポートしているかどうかを検出します。スクロール可能カーソルがサポートされていない場合、致命的エラーが生成され、ct_scroll_fetch を使用できません。この場合は、代わりに ct_fetch を使用します。

**構文**

```
CS_RETCODE ct_scroll_fetch(md, type, offset, option, rows_read)

CS_COMMAND   *cmd;
CS_INT       type;
CS_INT       offset;
CS_INT       option;
CS_INT       *rows_read;
```

**パラメータ**

*cmd*

スクロール可能カーソルの定義を保持するコマンド・ハンドル。

*type*

フェッチの方向。表 3-57 に示す有効なエントリを使用します。

**表 3-57：ct_scroll_fetch type の値**

| データ型 | オフセット値 | 意味 |
|---|---|---|
| CS_NEXT | 無視 | 呼び出すたびに、CS_CURSOR_ROWS を返す。<br>CS_CURSOR_ROWS のローの数が、カーソル結果セットのローの数より多い場合、配列の値は未定義となる場合がある。この状況は、最後のフェッチで CS_CURSOR_ROWS より少ないローが生成された場合も発生する。<br>返されたローの数を調べるには、rows_read を参照。これにより、アプリケーションの配列エントリを検証できる。<br>CS_NEXT のシーケンスを繰り返しても、カーソルが最後のテーブル・ローの外に移動する。これが発生すると、0 個のローが返され、ct_scroll_fetch は CS_CURSOR_AFTER_LAST を返す。カーソルが結果セットの境界の外へ移動したことを知らせる警告メッセージも生成される。これは警告であり、エラーが発生したことを示すものではない。<br>CS_CURSOR_ROWS のローの数がカーソル結果セットのローの数より多い場合、それに続いて CS_NEXT を呼び出すとカーソルが最後のローを越えて配置される。 |
| CS_FIRST | 無視 | CS_FIRST を設定すると、最初のローから開始して CS_CURSOR_ROWS が返される。後に CS_PREV が続くと、0 個のローが返され、ct_scroll_fetch は CS_CURSOR_BEFORE_FIRST を返す。 |

| データ型 | オフセット値 | 意味 |
|---|---|---|
| CS_PREV | 無視 | CS_PREV は、カーソルを現在の位置より前のローに配置する。<br>繰り返した場合、CS_PREV 呼び出しによりカーソルはロー 1 に戻され、次の CS_PREV 呼び出しにより 0 個のローが返されて、CS_CURSOR_BEFORE_FIRST が返される。カーソルが結果セットの境界の外へ移動したことを知らせる警告メッセージも生成される。これは警告であり、エラーが発生したことを示すものではない。 |
| CS_LAST | 無視 | CS_CURSOR_ROWS の最後のローを返す。CS_LAST の次に CS_NEXT が続く場合、0 個のローが返され、ct_scroll_fetch は CS_CURSOR_AFTER_LAST を返す。 |
| CS_RELATIVE | 正または負 (0 の場合はクライアントにより警告が生成される )。 | オフセット値が、符号付き整数 (CS_INT) として扱われ、正または負になる。これは、現在のカーソル位置からの相対ジャンプを示している。 |
| CS_ABSOLUTE | 正または負 (0 の場合はクライアントにより警告が生成される )。 | 絶対ロー番号を指定する必要がある。これは符号付き整数 (CS_INT) である。 |

*offset*

符号付き整数として渡され、タイプが CS_RELATIVE または CS_ABSOLUTE の場合のみ有効です。それ以外の場合、オフセットは CS_UNUSED となります。

*option*

スクロールの継続または停止に使用します。*option* が CS_TRUE の場合、*type* および *offset* に指定された ( 新しい ) 値に基づいて ct_scroll_fetch が継続されます。*option* が CS_FALSE の場合、カーソルのスクロールが停止し、CS_SCROLL_CURSOR_ENDS が返されます。

*rows_read*

ct_scroll_fetch 呼び出しごとのローの数を返します。

戻り値

ct_scroll_fetch は、「ct_fetch の戻り値」(560 ページ ) に示されている値 (CS_END_DATA を除く ) に加えて、次の値を返します。

**表 3-58：ct_scroll_fetch の戻り値**

| 戻り値 | 意味 |
| --- | --- |
| CS_SCROLL_ CURSOR_ENDS | CS_SCROLL_CURSOR_ENDS は、ct_scroll_fetch が CS_FALSE 値を受け取った場合に返される。<br>戻り値は、Adaptive Server Enterprise からこれ以上データをフェッチしないことを示すためにも使用されることがある。<br>通常、ct_scroll_fetch は ct_results の制御の下で実行される。ct_scroll_fetch を呼び出すたびに、CS_CURSOR_ROW に示されたローの最大数が返される。アプリケーションは、新しい ct_scroll_fetch 呼び出しを発行するか、フェッチを停止する。<br>アプリケーションがフェッチを停止した場合、ct_scroll_fetch により CS_SCROLL_CURSOR_ENDS が返され、ct_results が処理されて内部クリーンアップが実行される。 |
| CS_CURSOR_ BEFORE_FIRST | CS_CURSOR_BEFORE_FIRST は、ct_scroll_fetch への呼び出しにより、カーソルが Adaptive Server Enterprise 結果セットの最初のローより前に移動した場合に返される。ローは返されず、rows_read は 0 になる。エラー・ハンドラがインストールされた場合は、CS_CURSOR_BEFORE_FIRST により警告が生成されることがある。 |
| CS_CURSOR_ AFTER_LAST | CS_CURSOR_AFTER_LAST は、ct_scroll_fetch への呼び出しにより、カーソルが Adaptive Server Enterprise 結果セットの最後のローより後に移動した場合に返される。ローは返されず、rows_read は 0 になる。エラー・ハンドラがインストールされた場合は、CS_CURSOR_BEFORE_FIRST により警告が生成されることがある。 |

**例**

次のコードは、フェッチをスクロールする呼び出しシーケンスの例です。

```
CS_RETCODE CS_PUBLIC
ex_scroll_fetch_1(CS_COMMAND *cmd)
{
    CS_RETCODE       retcode;
    CS_INT           num_cols;
    CS_INT           i;
    CS_INT           j;
    CS_INT           k;
    CS_INT           row_count = 0;
    CS_INT           rows_read;
    CS_INT           disp_len;
    CS_INT           sc_type;
    CS_INT           sc_offset;
```

```
        CS_INT            sc_option;
        CS_DATAFMT        *datafmt;
        EX_COLUMN_DATA    *coldata;

    /*
    ** Find out how many columns there are in this result set.
    */
    retcode = ct_res_info(cmd, CS_NUMDATA, &num_cols, CS_UNUSED, NULL);
    if (retcode != CS_SUCCEED)
    {
        ex_error("ex_scroll_fetch_data: ct_res_info() failed");
        return retcode;
    }

    /*
    ** Make sure we have at least one column
    */
    if (num_cols <= 0)
    {
        ex_error("ex_scroll_fetch_data: ct_res_info() returned zero columns");
        return CS_FAIL;
    }

    /*
    ** Our program variable, called 'coldata', is an array of
    ** EX_COLUMN_DATA structures. Each array element represents
    ** one column. Each array element will re-used for each row.
    **
    ** First, allocate memory for the data element to process.
    */
    coldata = (EX_COLUMN_DATA *)malloc(num_cols * sizeof (EX_COLUMN_DATA));
    if (coldata == NULL)
    {
        ex_error("ex_scroll_fetch_data: malloc() failed");
        return CS_MEM_ERROR;
    }

    datafmt = (CS_DATAFMT *)malloc(num_cols * sizeof (CS_DATAFMT));
    if (datafmt == NULL)
    {
        ex_error("ex_scroll_fetch_data: malloc() failed");
        free(coldata);
        return CS_MEM_ERROR;
    }

    /*
```

```
** Loop through the columns getting a description of each one
** and binding each one to a program variable.
**
** We're going to bind each column to a character string;
** this will show how conversions from server native datatypes
** to strings can occur via bind.
**
** We're going to use the same datafmt structure for both the describe
** and the subsequent bind.
**
** If an error occurs within the for loop, a break is used to get out
** of the loop and the data that was allocated is free'd before
** returning.
*/
for (i = 0; i < num_cols; i++)
{
    /*
    ** Get the column description.  ct_describe() fills the
    ** datafmt parameter with a description of the column.
    */
    retcode = ct_describe(cmd, (i + 1), &datafmt[i]);
    if (retcode != CS_SUCCEED)
    {
        ex_error("ex_scroll_fetch_data: ct_describe() failed");
        break;
    }

    /*
    ** update the datafmt structure to indicate that we want the
    ** results in a null terminated character string.
    **
    ** First, update datafmt.maxlength to contain the maximum
    ** possible length of the column. To do this, call
    ** ex_display_len() to determine the number of bytes needed
    ** for the character string representation, given the
    ** datatype described above. Add one for the null
    ** termination character.
    */
    datafmt[i].maxlength = ex_display_dlen(&datafmt[i]) + 1;

    /*
    ** Set datatype and format to tell bind we want things
    ** converted to null terminated strings
    */
    datafmt[i].datatype = CS_CHAR_TYPE;
    datafmt[i].format   = CS_FMT_NULLTERM;
```

```
        /*
        ** Allocate memory for the column string
        */
        coldata[i].value = (CS_CHAR *)malloc(datafmt[i].maxlength);
        if (coldata[i].value == NULL)
        {
            ex_error("ex_scroll_fetch_data: malloc() failed");
            retcode = CS_MEM_ERROR;
            break;
        }

        /*
        ** Now bind.
        */
        retcode = ct_bind(cmd, (i + 1), &datafmt[i],
                    coldata[i].value, &coldata[i].valuelen,
                   (CS_SMALLINT *)&coldata[i].indicator);
         if (retcode != CS_SUCCEED)
        {
            ex_error("ex_scroll_fetch_data: ct_bind() failed");
            break;
        }
    }
    if (retcode != CS_SUCCEED)
    {
            for (j = 0; j < i; j++)
            {
                free(coldata[j].value);
            }
        free(coldata);
        free(datafmt);
        return retcode;
    }

    /*
    ** Display column header
    */
    ex_display_header(num_cols, datafmt);

    /*
    ** Fetch the rows. Call ct_scroll_fetch() as long as it returns
    ** CS_SUCCEED, CS_ROW_FAIL, CS_CURSOR_BEFORE_FIRST or
    ** CS_CURSOR_AFTER_LAST.
    ** These are recoverable or "row" producing conditions, e.g. non-fatal.
    ** All other terminate the loop, either by error or choice.
    */
```

```
for (i = 0; i < EX_MAX_ARR; i++)
{
    sc_type = scroll_index(type_list0[i], scroll_arrmap);
    sc_offset = offset_list0[i];

    if (type_list0[i] != EX_BADVAL)
    {
        sc_option = CS_TRUE;
    }
    else
    {
        /*
        ** Since EX_BADVAL is not valid to pass into
        ** either sc_type or sc_offset we set these
        ** to CS_UNUSED respectively.
        */
        sc_type = CS_UNUSED;
        sc_offset = CS_UNUSED;
        sc_option = CS_FALSE;
    }

    retcode = ct_scroll_fetch(cmd, sc_type, sc_offset, sc_option,
                &rows_read);
    switch ((int)retcode)
    {
        case CS_ROW_FAIL:

            fprintf(stdout, "Error on row %d.¥n", row_count);
            fflush(stdout);
            break;

        case CS_CURSOR_BEFORE_FIRST:

            fprintf(stdout, " Cursor before first row¥n");
            fflush(stdout);
            break;

        case CS_CURSOR_AFTER_LAST:

            fprintf(stdout, " Cursor after last row¥n");
            fflush(stdout);
            break;

        case CS_SUCCEED:

            /*
```

```
        ** Increment our row count by the number of
        ** rows just fetched.
        */
        row_count = row_count + rows_read;

        /*
        ** Assume we have a row. Loop through the
        ** columns displaying the column values.
        */
        for (k = 0; k < num_cols; k++)
        {
            /*
            ** Display the column value
            */
            fprintf(stdout, "%s", coldata[k].value);
            fflush(stdout);

            /*
            ** If not last column, Print out spaces between
            ** this column and next one.
            */
            if (k != num_cols - 1)
            {
                disp_len = ex_display_dlen(&datafmt[k]);
                disp_len -= coldata[k].valuelen - 1;
                for (j = 0; j < disp_len; j++)
                {
                    fputc(' ', stdout);
                }
            }
        }
        fprintf(stdout, "¥n");
        fflush(stdout);
        break;

    case CS_FAIL:

        /*
        ** Free allocated space.
        */
        for (k = 0; k < num_cols; k++)
        {
            free(coldata[k].value);
        }
        free(coldata);
        free(datafmt);
        return retcode;
```

```
        case CS_SCROLL_CURSOR_ENDS:

            /*
            ** User signalled ct_scroll_fetch() to stop
            ** scrolling, we are done with this result set.
            ** Free allocated space.
            */
            for (k = 0; k < num_cols; k++)
            {
                free(coldata[k].value);
            }
            free(coldata);
            free(datafmt);
            return retcode;

        default:

            fprintf(stdout, "Hit default, this should not happen.
                    Exiting program.¥n");
            fflush(stdout);
            exit(0);
        } /* end switch */
    } /* end for */
    return CS_SUCCEED;
}
```

**使用法**

- 結果セット内の最初のローの番号は 1 です。
- rows_read は、ct_scroll_fetch を呼び出すたびにフェッチされた実際のローの数を返します。アプリケーション配列において、有益でない可能性がある情報ではなく、ローの実際の数を調べるには、rows_read を使用します。

**参照**

ct_fetch

# ct_send

**説明**

コマンドをサーバに送信します。

**構文**

CS_RETCODE ct_send(cmd)

CS_COMMAND   *cmd;

**パラメータ**

*cmd*

クライアント／サーバ・オペレーションを管理する CS_COMMAND 構造体を指すポインタです。

**戻り値**

ct_send は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。<br>あまり重大でない障害の場合、アプリケーションは ct_cancel(CS_CANCEL_ALL) を呼び出し、コマンド構造体をクリーンアップしなければならない。<br>より重大な障害の場合、アプリケーションは ct_close(CS_FORCE_CLOSE) を呼び出して接続をクローズしなければならない。 |
| CS_CANCELED | ルーチンがキャンセルされた。 |
| CS_PENDING | 非同期ネットワーク I/O が有効。「非同期プログラミング」(12 ページ ) を参照。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

ct_send が失敗する一般的な原因として、結果保留中エラーや、開始されていないコマンドの送信などがあります。

- 結果保留中エラー

  ct_send は、呼び出されたときに Client-Library が結果を読み込んでいる途中であった場合、結果保留中エラーを提示します。このエラーは、アプリケーションが同じ親接続に属する別のコマンド構造体に対応しているカーソル結果でない結果を取得している場合に、発生します。この問題は、Client-Library カーソルを使用するようにアプリケーションを書き直すことで解決できる場合もあります ( 詳細については、「ct_cursor」を参照してください )。アプリケーションがカーソルを使用できない場合、別の接続が必要です。

- 開始されていないコマンドを送信する試み

  ct_send は、コマンドに ct_command、ct_cursor、または ct_dynamic が定義されていないと失敗します。

**例**

次のコードでは、言語コマンドを送信して結果を処理する関数を宣言しています。ここでは、コマンドはフェッチ可能な結果を返さないものと想定します。

```
/*
 ** ex_execute_cmd()
 */
CS_RETCODE CS_PUBLIC
 ex_execute_cmd(connection, cmdbuf)
 CS_CONNECTION   *connection;
 CS_CHAR         *cmdbuf;
 {
      CS_RETCODE    retcode;
      CS_INT        restype;
      CS_COMMAND    *cmd;
      CS_RETCODE    query_code;

     /*
      ** Get a command handle, store the command string
      ** in it, and send it to the server.
      */
      if ((retcode = ct_cmd_alloc(connection, &cmd)) !=
            CS_SUCCEED)
      {
           ex_error("ex_execute_cmd:ct_cmd_alloc() ¥
                failed");
           return retcode;
      }

     if ((retcode = ct_command(cmd, CS_LANG_CMD,
           cmdbuf, CS_NULLTERM, CS_UNUSED)) !=
           CS_SUCCEED)
      {
           ex_error("ex_execute_cmd:ct_command() ¥
                failed");
           (void)ct_cmd_drop(cmd);
           return retcode;
      }

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

     /*
      ** Examine the results coming back. If any errors
      ** are seen, the query result code (which we will
      ** return from this function) will be set to FAIL.
      */
      ...CODE DELETED.....
```

```
    /* Clean up the command handle used */
     if (retcode == CS_END_RESULTS)
     {
          retcode = ct_cmd_drop(cmd);
          if (retcode != CS_SUCCEED)
          {
               query_code = CS_FAIL;
          }
     }
    else
     {
          (void)ct_cmd_drop(cmd);
          query_code = CS_FAIL;
     }
    return query_code;
}
```

このコードは、*exutils.c* サンプル・プログラムからの抜粋です。

**使用法**

- ct_send はサーバが実行するコマンドをネットワークを介して送信します。
- ct_send を呼び出す前に、アプリケーションはコマンドのタイプと実行に必要なデータを指定する必要があります。この手順が行われると、コマンド構造体は「**開始された状態**」になります。コマンドはルーチン ct_command、ct_cursor、または ct_dynamic で開始されます。

  Client-Library アプリケーションで使用できるサーバ・コマンドについては、『Open Client Client-Library/C プログラマーズ・ガイド』の「第 5 章 コマンド・タイプの選択」を参照してください。
- ほとんどのコマンド・タイプの場合、前の実行からの結果をすべて処理した後で、ct_send を呼び出して、前に実行したコマンドを再送信することができます。ただし、「コマンドの再送信」で説明するような例外があります。

**最初の送信**

- 最初の実行に対するサーバへのコマンドの送信は、次のような複数のプロセスで行われます。

  a ct_command、ct_cursor、または ct_dynamic を呼び出してコマンドを開始します。これらのルーチンは、サーバに送信する記号コマンド・ストリームの構築に使用する内部構造体を設定します。

b　必要に応じて、コマンドが必要とする各パラメータに対して ct_param または ct_setparam を一度呼び出し、コマンドにパラメータを渡します。

すべてのコマンドがパラメータを必要とするわけではありません。たとえば、リモート・プロシージャ・コール・コマンドでは、呼び出されるストアド・プロシージャに応じて、パラメータが必要な場合と不必要な場合があります。

c　ct_send を呼び出して、コマンドをサーバに送信します。ct_send はコマンド構造体の親接続に記号コマンド・ストリームを書き込みます。

d　CS_SUCCEED を返さなくなるまで、繰り返し ct_results を呼び出して、コマンドの実行結果を処理します。結果処理の詳細については、「結果」(280 ページ ) を参照してください。

- アプリケーションは、ct_cancel(CS_CANCEL_ALL) を呼び出して、開始されているが送信されていないコマンドをキャンセルできます。しかし、アプリケーションは、コマンドを送信した後、ct_cancel を呼び出してコマンド実行の結果をキャンセルする前に、ct_results を呼び出す必要があります。
- ct_send は非同期書き込みを使用するため、サーバからの応答を待ちません。アプリケーションは、コマンドの成功を確認するため、およびコマンド結果処理の準備をするために、ct_results を呼び出す必要があります。

**コマンドの再送信**

- ほとんどのタイプのコマンドは、前のコマンドの結果をすべて処理した直後に再送信できます。ただし、次のコマンドは例外です。
  - ct_command(CS_SEND_DATA_CMD) によって開始されたデータ送信コマンド
  - ct_command(CS_SEND_BULK_CMD) によって起動されるバルク送信コマンド

  その他のタイプのコマンドの場合、アプリケーションは、前の実行結果をすべて処理した直後に ct_send でコマンドを再送信できます。アプリケーションが ct_command、ct_cursor、ct_dynamic、または ct_sendpassthru を使用して新しいコマンドを開始するまで、Client-Library は開始されたコマンドの情報を廃棄しません。
- 一般に、コマンドを再送信するアプリケーションでは、ct_param ではなく ct_param でコマンド・パラメータを入力するようにします。

- ct_setparam 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。ct_setparam は、パラメータ値が入っているプログラム変数へのポインタを受け取ります。変数の内容は、後続の ct_send 呼び出しで読み込まれます。コマンド・パラメータとプログラム変数のバインドは、アプリケーションが ct_command、ct_cursor、ct_dynamic、または ct_sendpassthru で新しいコマンドを開始するまで持続します。
- ct_param は、戻る前に、プログラム変数からデータをコピーします。ct_param を呼び出してコマンド・パラメータを入力した場合、コマンドを再送信するときにパラメータ値を変更できません。

パラメータがリテラル値である ( つまり、変更されない ) 場合には、コマンドを再送信するような場合でも、ct_param でパラメータを定義する方が適切です。

- アプリケーションは、CS_HAVE_CMD プロパティをチェックして、コマンド構造体に再送信可能なコマンドが存在するかどうかを調べることができます。このプロパティの詳細については、「再送可能なコマンド」(248 ページ ) を参照してください。
- コマンドを再送信するアプリケーションは、冗長な ct_bind 呼び出しを避けるために、CS_STICKY_BINDS プロパティを使用できる場合があります。このプロパティの詳細については、「継続結果バインド」(261 ページ ) を参照してください。

**参照** ct_command、ct_cursor、ct_dynamic、ct_fetch、ct_param、ct_results、ct_setparam

# ct_send_data

**説明** text データまたは image データのまとまりをサーバに送信します。

**構文**

```
CS_RETCODE ct_send_data(cmd, buffer, buflen)

CS_COMMAND   *cmd;
CS_VOID      *buffer;
CS_INT       buflen;
```

**パラメータ**

*cmd*

クライアント／サーバ・オペレーションを管理する CS_COMMAND 構造体を指すポインタです。

*buffer*
サーバに書き込む値を指すポインタです。

*buflen*
**buffer* のバイト単位の長さです。

CS_NULLTERM は、*buflen* に対する有効な値ではありません。

**戻り値**

ct_send_data は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_CANCELED | データ送信オペレーションがキャンセルされた。 |
| CS_PENDING | 非同期ネットワーク I/O が有効。「非同期プログラミング」(12 ページ ) を参照。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

**例**

**例 1** 次のコードは、データ送信コマンドを構築して送信する呼び出しシーケンスの例です。

```
/*
 ** UpdateTextData()
 */

CS_STATIC CS_RETCODE
 UpdateTextData(connection, textdata, newdata)
 CS_CONNECTION connection;
 TEXT_DATA      textdata;
 char           *newdata;
 {
       CS_RETCODE    retcode;
       CS_INT        res_type;
       CS_COMMAND    *cmd;
       CS_INT        i;
       CS_TEXT       *txtptr;
       CS_INT        txtlen;

      /*
       ** Allocate a command handle to send the text with
       */
       ...CODE DELETED.....

      /*
       ** Inform Client-Library the next data sent will
       ** be used for a text or image update.
       */
```

```
if ((retcode = ct_command(cmd, CS_SEND_DATA_CMD,
     NULL, CS_UNUSED, CS_COLUMN_DATA)) !=
     CS_SUCCEED)
{
     ex_error("UpdateTextData:ct_command() ¥
          failed");
     return retcode;
}
/*
** Fill in the description information for the
** update and send it to Client-Library.
*/
txtptr = (CS_TEXT *)newdata;
txtlen = strlen(newdata);
textdata->iodesc.total_txtlen = txtlen;
textdata->iodesc.log_on_update = CS_TRUE;
retcode = ct_data_info(cmd, CS_SET, CS_UNUSED,
     &textdata->iodesc);
if (retcode != CS_SUCCEED)
{
     ex_error("UpdateTextData:ct_data_info() ¥
          failed");
     return retcode;
}
/*
** Send the text one byte at a time. This is not
** the best thing to do for performance reasons,
** but does demonstrate that ct_send_data()
** can handle arbitrary amounts of data.
*/
for (i = 0; i < txtlen; i++, txtptr++)
{
     retcode = ct_send_data(cmd, txtptr,
          (CS_INT)1);
     if (retcode != CS_SUCCEED)
     {
        ex_error("UpdateTextData:ct_send_data() ¥
               failed");
          return retcode;
     }
}
/*
** ct_send_data() writes to internal network
** buffers. To insure that all the data is
** flushed to the server, a ct_send() is done.
*/
```

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

/* Process the results of the command */
 while ((retcode = ct_results(cmd, &res_type)) ==
      CS_SUCCEED)
 {
      switch ((int)res_type)
      {
         case CS_PARAM_RESULT:
         /*
         ** Retrieve a description of the
         ** parameter data. Only timestamp data is
         ** expected in this example.
         */
         retcode = ProcessTimestamp(cmd, textdata);
           if (retcode != CS_SUCCEED)
           {
                ex_error("UpdateTextData:¥
                     ProcessTimestamp() failed");
                /*
                ** Something failed, so cancel all
                ** results.
                */
               ct_cancel(NULL, cmd, CS_CANCEL_ALL);
                return retcode;
           }
           break;

          case CS_CMD_SUCCEED:
           case CS_CMD_DONE:
           /*
           ** This means that the command succeeded
           ** or is finished.
           */
           break;

          case CS_CMD_FAIL:
           /*
           ** The server encountered an error while
           ** processing our command.
           */
           ex_error("UpdateTextData:ct_results() ¥
           returned CS_CMD_FAIL");
           break;
```

```
            default:
             /*
             ** We got something unexpected.
             */
             ex_error("UpdateTextData:ct_results() ¥
                  returned unexpected result type”);
             /* Cancel all results */
             ct_cancel(NULL, cmd, CS_CANCEL_ALL);
             break;
        }
    }

   /*
    ** We're done processing results. Let's check the
    ** return value of ct_results() to see if
    ** everything went ok.
    */
    ...CODE DELETED.....

   return retcode;
}
```

このコードは、*getsend.c* サンプル・プログラムからの抜粋です。

**例 2** 次のコードは、部分更新データを送信する呼び出しシーケンスの例です。

```
/*
** UpdateTextData()
*/
CS_STATIC CS_RETCODE
UpdateTextData(connection, textdata, newdata)
CS_CONNECTION    connection;
TEXT_DATA        textdata;
char             *newdata;
{
   CS_RETCODE    retcode;
   CS_INT        res_type;
   CS_COMMAND    *cmd;
   CS_INT        i;
   CS_TEXT       *txtptr;
   CS_INT        txtlen;
   /*
   ** Allocate a command handle to send the text with
   */
   ...CODE DELETED.....
   /*
   ** Inform Client-Library the next data sent will
```

```
** be used for a text or image update.
*/
if ((retcode = ct_command(cmd, CS_SEND_DATA_CMD,
NULL, CS_UNUSED, CS_COLUMN_DATA)) != CS_SUCCEED)
{
   ex_error("UpdateTextData:ct_command() ¥
      failed");
   return retcode;
}
/*
** Fill in the description information for the
** update and send it to Client-Library.
*/
txtptr = (CS_TEXT *)newdata;
txtlen = strlen(newdata);
textdata->iodesc.total_txtlen = txtlen;
textdata->iodesc.log_on_update = CS_TRUE;
/*
**  Insert newdata at offset 20.
*/
textdata->iodesc.iotype = CS_IOPARTIAL;
textdata->iodesc.offset = 20;
textdata->iodesc.delete_length = 0;
retcode = ct_data_info(cmd, CS_SET, CS_UNUSED,
&textdata->iodesc);
 if (retcode != CS_SUCCEED)
{
   ex_error("UpdateTextData:ct_data_info() ¥
      failed");
   return retcode;
}
/*
** Send the text one byte at a time. This is not
** the best thing to do for performance reasons,
** but does demonstrate that ct_send_data()
** can handle arbitrary amounts of data.
*/
for (i = 0; i < txtlen; i++, txtptr++)
{
   retcode = ct_send_data(cmd, txtptr,(CS_INT)1);
    if (retcode != CS_SUCCEED)
   {
      ex_error("UpdateTextData:ct_send_data() ¥
         failed");
      return retcode;
   }
}
```

```
/*
** ct_send_data() writes to internal network
** buffers. To insure that all the data is
** flushed to the server, a ct_send() is done.
*/
if ((retcode = ct_send(cmd)) != CS_SUCCEED)
{
   ex_error("UpdateTextData:ct_send() failed");
   return retcode;
}
/* Process the results of the command */
while ((retcode = ct_results(cmd, &res_type)) ==
   CS_SUCCEED)
{
   switch ((int)res_type)
   {
      case CS_PARAM_RESULT:


         /*
         ** Retrieve a description of the
         ** parameter data. Only timestamp data is
         ** expected in this example.
         */
         retcode = ProcessTimestamp(cmd, textdata);
          if (retcode != CS_SUCCEED)
         {
            ex_error("UpdateTextData:¥
              ProcessTimestamp() failed");
            /*
            ** Something failed, so cancel all
            ** results.
            */
            ct_cancel(NULL, cmd, CS_CANCEL_ALL);
            return retcode;
         }
         break;
      case CS_CMD_SUCCEED:
      case CS_CMD_DONE:
         /*
         ** This means that the command succeeded
         ** or is finished.
         */
         break;
      case CS_CMD_FAIL:
         /*
         ** The server encountered an error while
```

```
                ** processing our command.
                */
                ex_error("UpdateTextData:ct_results() ¥
                   returned CS_CMD_FAIL");
                break;
             default:
                /*
                ** We got something unexpected.
                */
                ex_error("UpdateTextData:ct_results() ¥
                   returned unexpected result type” );
                /* Cancel all results */
                ct_cancel(NULL, cmd, CS_CANCEL_ALL);
                break;
          }
       }
       /*
       ** We're done processing results. Let's check the
       ** return value of ct_results() to see if
       ** everything went ok.
       */
       ...CODE DELETED.....
       return retcode;
    }
```

このコードは *uctext.c* サンプル・プログラムからの抜粋です。

**使用法**

- 別のデータベースに含まれていて、ビューには表示されないカラムであっても、アプリケーションは ct_send_data を使用してそのデータベース・カラムに text 値または image 値を書き込むことができます。ただし、この場合、ユーザは基本となるテーブルの更新特権を持っていなければなりません。この書き込みオペレーションは実際には update になります。つまり、新しい値を書き込むために ct_send_data を呼び出す場合、そのカラムは値を持っていなければなりません。

  これは、ct_send_data がカラムに書き込むときにテキスト・タイムスタンプ情報を使用するため、および、カラムが値を持つまで有効なテキスト・タイムスタンプを持たないためです。text または image カラムに含まれる値は NULL にすることができますが、NULL は SQL update 文で明示的に入力してください。

- ct_send_data を使用して text または image カラムを更新する手順については、「text または image カラムの更新」(330 ページ ) を参照してください。ct_send_data を使用した部分更新の送信の詳細については、「ct_send_data による部分更新の送信」(334 ページ ) を参照してください。

- データ送信オペレーションを実行するには、アプリケーションは現在の I/O 記述子 ( 更新するカラム値について記述している CS_IODESC 構造体 ) を持たなければなりません。
  - CS_IODESC の *textptr* フィールドはターゲット・カラムを識別します。
  - CS_IODESC の *timestamp* フィールドはそのカラム値のテキスト・タイムスタンプです。*timestamp* がその値の現在のデータベース・テキスト・タイムスタンプと一致しない場合、更新オペレーションは失敗します。
  - CS_IODESC の *total_txtlen* フィールドは、カラムの新しい値のバイト単位の全長を示します。アプリケーションは、ct_send を呼び出して text または image 更新オペレーションの終了を示す前に、このバイト数を正確に書き込むためにループで ct_send_data を呼び出す必要があります。
  - CS_IODESC の *log_on_update* は更新オペレーションのログを取るかどうかをサーバに指示します。
  - CS_IODESC の *locale* フィールドは、新しい値のローカライゼーション情報がある場合、それを含む CS_LOCALE 構造体を指します。

  一般のアプリケーションは、更新オペレーションで I/O 記述子を使用する前に、*locale*、*total_txtlen*、*log_on_update* フィールドの値だけを変更しますが、同じカラム値を何度も更新するアプリケーションは *timestamp* フィールドの値も変更する必要があります。

- text または image 値の更新が正常に終了すると、その text または image 値に新しいテキスト・タイムスタンプを持つパラメータ結果セットが生成されます。アプリケーションが、この text または image 値の更新を再度予定する場合、ct_data_info を呼び出して更新オペレーションの CS_IODESC を定義する前に、この新しいテキスト・タイムスタンプを保存して、その値の CS_IODESC にコピーしてください。
- text または image 更新オペレーションは、Transact-SQL update 文を含む言語コマンドに相当します。
- *cmd* によって識別されたコマンド領域は、text または image 更新オペレーションが開始されるまでアイドル状態でなければなりません。コマンド領域にアクティブ・コマンド、未処理の結果、またはオープン・カーソルがない場合、その領域はアイドル状態です。

### コマンドの抑制

text または image カラムを更新するには、通常、クライアント・アプリケーションでは ct_command ルーチンを呼び出し、データ送信コマンドを開始します。その後、クライアントは ct_data_info コマンドを呼び出し、CS_IODESC を取得し、後続の ct_send_data ルーチンの呼び出しで生成する適切な SQL コマンド (update または writetext) を判断します。

この処理を単純化しパフォーマンスを向上させるために、クライアントは SQL コマンド (update または writetext) の生成を抑制し、サーバのバルク・ハンドラに直接データを送信することもできます。そのためには、クライアントは *type* パラメータを CS_SEND_DATA_NOCMD に設定し、ct_command ルーチンを呼び出すことによってデータ送信コマンドを開始しなければなりません。その後、クライアント・アプリケーションは、データ送信コマンドを使用してサーバのバルク・ハンドラに text データのみまたは image データのみを送信できます。サーバでバルク・イベントが発生すると、送信する合計バイト数を示す 4 バイトのフィールドに続き、text または image データが送信されます。バルク・ハンドラは srv_text_info を使用して予想される合計バイト数を読み取り、srv_get_data を使用してデータを読み取ります。

サーバは sp_mda を定義して、SQL コマンドを使用せずに text データまたは image データのみを送信する ct_send_data ルーチンをそれがサポートするかどうかを指定しなければなりません。サーバの sp_mda プロシージャは、ct_connect ルーチンの呼び出し前に、クライアント・アプリケーションで ct_con_props(CS_SENDDATA_NOCMD) などの特定のプロパティが設定されている場合にのみ呼び出されます。これらのいずれかのプロパティ (CS_PARTIAL_TEXT や CS_SENDDATA_NOCMD 接続プロパティなど ) が設定されている場合、ct_connect の実行時にサーバの sp_mda プロシージャが呼び出されます。SQL コマンドを使用せずに text データのみまたは image データのみを送信する ct_send_data ルーチンがサーバでサポートされていないことが sp_mda で指定されている場合は、*type* パラメータを CS_SEND_DATA_NOCMD に設定した ct_command ルーチンの呼び出しは失敗します。

サーバが SQL コマンドを使用せずに text または image データを受信できる場合は、sp_mda は次の結果を返します。

| パラメータ | 値 |
| --- | --- |
| mdinfo | 「SENDDATA_NOCMD」 |
| querytype | 2 |
| query | senddata no cmd |

**注意** Adaptive Server では、SQL コマンドを使用せずに image データまたは text データを受信することはできません。

参照

ct_data_info、ct_get_data、「text および image データの処理」(328 ページ)

# ct_send_params

説明

コマンド・パラメータをバッチ送信します。

構文

```
CS_RETCODE ct_send_params(
CS_COMMAND *cmd,
CS_INT reserved)
```

パラメータ

*cmd*

CS_COMMAND 構造体へのポインタ。

*reserved*

CS_UNUSED に設定されます。これは後で使用できるように予約されているプレースホルダです。

戻り値

ct_send_params で次の値が戻されます。

| 戻り値 | 意味 |
| --- | --- |
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |

例

**例 1** バインドされたパラメータ・バッファの再利用 : 再バインドなし。

```
CS_CHAR linedata[MAX_LINE];
CS_UINT linenum;
retcode = ct_command(cmd, CS_LANG_CMD, sqlcommand, CS_NULLTERM, CS_END);
...
retcode = ct_setparam(cmd, &datafmt2, &linedata, &linelen, &gooddata);
...
retcode = ct_setparam(cmd, &datafmt1, &linenum, &unused, &gooddata);
```

```
...
while (fgets(linedata, sizeof(linedata), file) != NULL)
{
   linenum++;
   /*
   ** Send the parameters. This also starts sending the command if
   ** it's the first set of parameters.
   */
   retcode = ct_send_params(cmd, CS_UNUSED);
   ...
}
retcode = ct_send(cmd);
...
retcode = ex_handle_results(cmd);
...
```

**例 2** ct_setparam(cmd, NULL, ..) によるパラメータの再バインド

```
typedef struct _my_data
{
   CS_INT number;
   CS_CHAR *string;
} MY_DATA;
MY_DATA da[];
...
retcode = ct_dynamic(cmd, CS_EXECUTE, dyn_id, CS_NULLTERM, NULL,
   CS_UNUSED);
...
retcode = ct_setparam(cmd, &datafmt1, NULL, &unused, NULL);
retcode = ct_setparam(cmd, &datafmt2, NULL, NULL, NULL);
...
for (i = 0; i < count; i++)
{
   printf("Sending:%i, %s¥n", da[i].number, da[i].string);
   retcode = ct_setparam(cmd, NULL, &da[i].number, &unused, &gooddata);
   ...
   retcode = ct_setparam(cmd, NULL, da[i].string, &nullterm, &gooddata);
   ...
   retcode = ct_send_params(cmd, CS_UNUSED);
   ...
}
retcode = ct_send(cmd);
...
retcode = ex_handle_results(cmd);
...
```

**使用法**

この関数を呼び出すと、ct_param() または ct_setparam() を使用して示されたパラメータが送信されます。パラメータの送信を停止するには、最後の ct_send_params() 呼び出しの後で ct_send() 呼び出しを使用します。これはパラメータの終了を示すシグナルとなり、現在のコマンドが完了します。

- 最初の ct_send_params() 呼び出しによって、実際のコマンド、すべてのパラメータのパラメータ・フォーマット、および最初のパラメータ・セットがサーバに送信されます。後続の呼び出しでは、フォーマットなしで各パラメータのみが送信されます。
- サーバでコマンドの処理を開始できるように、パラメータが格納されているネットワーク・バッファが ct_send_params() に対する呼び出しのたびにフラッシュされます。
- ct_send() とは異なり、ct_send_params() で現在のコマンドは終了しません。ct_send_params() を繰り返し呼び出すことによって、複数のパラメータ・セットを送信することができます。
- 結果の処理は、コマンドを完了するための ct_send() 呼び出しの後でのみ可能になります。ct_results() が ct_send() の前に呼び出されると、エラーが発生します。

# ct_sendpassthru

**説明**

TDS (Tabular Data Stream) パケットをサーバに送信します。

**構文**

```
CS_RETCODE ct_sendpassthru (cmd, sendptr)

CS_COMMAND   *cmd;
CS_VOID      *sendptr;
```

**パラメータ**

*cmd*

CS_COMMAND 構造体を指すポインタです。

*sendptr*

サーバに送信される TDS パケットを含むバッファを指すポインタです。

**戻り値**

ct_sendpassthru は、次の値を返します。

**表 3-59：ct_sendpassthru の戻り値**

| 戻り値 | 意味 |
| --- | --- |
| CS_PASSTHRU_MORE | パケットを正常に送信した。パケットがさらに存在する。 |
| CS_PASSTHRU_EOM | パケットを正常に送信した。パケットはこれ以上存在しない。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_CANCELLED | ルーチンがキャンセルされた。 |
| CS_PENDING | 非同期ネットワーク I/O が有効。「非同期プログラミング」(12 ページ ) を参照。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

**使用法**

- TDS は、クライアントとサーバの間で要求および要求結果の転送に使用する通信プロトコルです。通常は、Client-Library がデータ・ストリームを管理するので、ゲートウェイ・アプリケーション以外は TDS を取り扱う必要はありません。
- ct_recvpassthru と ct_sendpassthru はゲートウェイ・アプリケーションで役立ちます。アプリケーションが両者 ( クライアントとリモート・サーバ、または 2 つのサーバなど ) の仲介の役目をするとき、TDS ストリームをあるサーバから他のサーバに渡すためにこれらのルーチンを使用して、情報の解釈と情報の再コード化のプロセスを省略できます。
- ct_sendpassthru は、バイトのパケットを **sendptr* バッファから送信します。通常、*sendptr* は srv_recvpassthru によって返される **recvptr* になります。*sendptr* は、送信するパケットを格納するユーザ割り付けバッファのアドレスにもなります。
- デフォルト・パケット・サイズはプラットフォームによって異なります。ほとんどのプラットフォームでは、パケットのデフォルト・サイズは 512 バイトです。接続では、ct_con_props を通じてそのパケット・サイズを変更できます。
- バッファ内の TDS パケットに EOM (End Of Message) というマークが付いている場合、ct_sendpassthru は CS_PASSTHRU_EOM を返します。EOM マークが付いていない場合、ct_sendpassthru は CS_PASSTHRU_MORE を返します。
- パススルー・オペレーションに使用されている接続は、CS_PASSTHRU_EOM が受信されるまで、他の Client-Library 関数では使用できません。

**参照**

ct_getloginfo、ct_recvpassthru、ct_setloginfo

# ct_setloginfo

**説明**

TDS ログイン応答情報を CS_LOGINFO 構造体から CS_CONNECTION 構造体に転送します。

**構文**

```
CS_RETCODE ct_setloginfo (connection, loginfo)

CS_CONNECTION    *connection;
CS_LOGINFO       *loginfo;
```

**パラメータ**

*connection*

CS_CONNECTION 構造体を指すポインタです。CS_CONNECTION 構造体は、特定のクライアント／サーバ接続の情報を含んでいます。

*loginfo*

CS_LOGINFO 構造体を指すポインタです。

**戻り値**

ct_setloginfo は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ) を参照。 |

**使用法**

- TDS (Tabular Data Stream) は、Sybase のクライアントとサーバの間における、要求と要求結果の転送に使用される通信プロトコルです。
- ct_setloginfo は、TDS 情報を転送した後に CS_LOGINFO 構造体を解放するので、アプリケーションでその CS_LOGINFO を再利用することはできません。アプリケーションは、ct_getloginfo を呼び出すことによって、新しい CS_LOGINFO を取得できます。
- アプリケーションが ct_setloginfo を呼び出す理由として、次の 2 つがあります。
  - TDS パススルーを使用した Open Server ゲートウェイ・アプリケーションの場合
  - オープン接続から新しく割り付けた接続構造体に、ログイン・プロパティをコピーするため

**注意**　ct_setloginfo は、完了コールバック・ルーチン内からは呼び出さないでください。ct_setloginfo は再入不可能なシステム・レベルのメモリ関数を呼び出します。

TDS パススルー

- クライアントがサーバと直接接続している場合、2 つのプログラムは、データの送受信に使用する TDS のフォーマットをネゴシエートします。ゲートウェイ・アプリケーションが TDS パススルーを使用する場合、そのゲートウェイは、クライアントとリモート・サーバの間で TDS パケットを検査および処理することなく転送します。このため、リモート・サーバとクライアントは、使用する TDS フォーマットについて合意しておく必要があります。
- ct_setloginfo は、クライアントとリモート・サーバが TDS フォーマットをネゴシエートするときに行われる 4 つの呼び出しのうち 2 番目の呼び出しです。4 つのうち 2 つは、Server Library の呼び出しです。Open Server SRV_CONNECT イベント・ハンドラだけで行うことができる呼び出しは、次のとおりです。

  a　CS_LOGINFO 構造体を割り付けて、それをクライアント・ログイン要求からの TDS 情報で満たす srv_getloginfo。

  b　手順 1 で取得された TDS 情報を CS_LOGINFO 構造体から Client-Library CS_CONNECTION 構造体に転送する ct_setloginfo。ゲートウェイは、リモート・サーバとの接続を確立する ct_connect 呼び出しに、この CS_CONNECTION 構造体を使用します。

  c　クライアントの TDS 情報に対するリモート・サーバの応答を、CS_CONNECTION 構造体から新しく割り付けた CS_LOGINFO 構造体に転送する ct_getloginfo。

  d　手順 3 で取得されたリモート・サーバの応答をクライアントに送信する srv_setloginfo。

ログイン・プロパティのコピー

ログイン・プロパティをオープン接続から新しく割り付けた接続構造体にコピーするための ct_setloginfo の使用については、「ログイン・プロパティのコピー」(211 ページ ) を参照してください。

参照　　ct_getloginfo、ct_recvpassthru、ct_sendpassthru

# ct_setparam

**説明**

ct_send がサーバ・コマンドのための入力パラメータ値を読み込むソース変数を指定します。

**構文**

```
CS_RETCODE ct_setparam(cmd, datafmt, data,
         datalenp, indp)

CS_COMMAND   *cmd;
CS_DATAFMT   *datafmt;
CS_VOID      *data;
CS_INT       *datalenp;
CS_SMALLINT  *indp;
```

**パラメータ**

*cmd*

クライアント／サーバ・オペレーションを管理する CS_COMMAND 構造体を指すポインタです。

*datafmt*

パラメータを記述する CS_DATAFMT 構造体を指すポインタです。ct_setparam は制御を戻す前に *datafmt の内容をコピーします。Client-Library は、以後は *datafmt* を参照しません。

*data*

値バッファのアドレスです。Client-Library は、後続の ct_send の呼び出しで、このパラメータの現在の値を *data から読み込みます。

null 値を持つパラメータを示すには、次の 3 つの方法があります。

- * *indp* を -1 にして ct_send を呼び出します。この場合は、ct_send は *data を無視します。
- *datalenp を 0 にして ct_send を呼び出します。
- *data*、*datalenp*、および *indp* を NULL にして ct_setparam を呼び出します。

*datalenp*

*data 内のパラメータ値のバイト単位の長さを指定する整数変数のアドレス、またはこのパラメータの値が変らない場合は NULL を指定します。

*datalenp* が NULL でない場合は、後続の ct_send の呼び出しで、現在の値の長さが *datalenp から読み込まれます。長さが 0 の場合は null 値であることを示します。

*datalenp* が NULL で、*data* が NULL でない場合、*datafmt–>maxlength* は、このパラメータの null でない値のすべての長さを示します。*datalenp* が NULL の場合は、ct_send に対する以後の呼び出しに対するパラメータ値が null であることを示すためにインジケータ変数を使用する必要があります。

*indp*

パラメータの現在の値が NULL であるかどうかを示す値である CS_SMALLINT 変数のアドレスです。パラメータが null 値を持つことを示すには、**indp* に -1 を設定します。**indp* が -1 の場合、ct_send は **data* と **datalenp* を無視します。

**戻り値**

ct_setparam は次の値を返します。

| 戻り値 | 意味 |
| --- | --- |
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

**例**

次の例は、パラメータを使用するカーソルの宣言、オープン、再オープンのためのコードで使用される ct_setparam を示します。

例 : カーソルを再オープンするための ct_setparam

```
/*
 ** Data structures to describe a parameter and a cursor.
 */
 typedef struct _langparam
 {
         CS_CHAR        *name;
         CS_INT         type;
         CS_INT         len;
         CS_INT         maxlen;
         CS_VOID        *data;
         CS_SMALLINT    indicator;
 } LANGPARAM;

typedef struct _cur_control
 {
         CS_CHAR        *name;
         CS_CHAR        *query;
         LANGPARAM      *params;
         CS_INT         numparams;
 } CUR_CONTROL;
```

```
/*
 ** Static data for a parameterized cursor body.
 */
 CS_STATIC CS_MONEY  PriceVal;
 CS_STATIC CS_INT    SalesVal;
 CS_STATIC LANGPARAM Params [] =
 {
         { "@price_val", CS_MONEY_TYPE,
            CS_SIZEOF(CS_MONEY), CS_SIZEOF(CS_MONEY),
            (CS_VOID *)&PriceVal, 0
         },
         { "@sales_val", CS_INT_TYPE,
            CS_SIZEOF(CS_INT), CS_SIZEOF(CS_INT),
            (CS_VOID *)&SalesVal, 0
         },
 };
 #define NUMPARAMS  (CS_SIZEOF(Params) / CS_SIZEOF(LANGPARAM))

 #define QUERY ¥
 "select title_id, title, price, total_sales from titles ¥
  where price > @price_val and total_sales > @sales_val ¥
  for read only"

 CS_STATIC CUR_CONTROL Cursor_Control =
 { "curly", QUERY, Params, NUMPARAMS };

/*
 ** OpenCursor() -- Declare and open a new cursor or reopen
 **         an existing cursor (which must have been originally
 **         declared and opened using this function).
 **
 **         If the open is successful, this function processes the cursor
 **         results up to the CS_CURSOR_RESULT result type value. In
 **         other words, the command handle is ready for
 **         ct_bind/ct_fetch/etc.
 **
 ** Parameters
 **   cmd -- CS_COMMAND handle for the new cursor.
 **   cur_control -- address of a CUR_CONTROL structure that contains
 **     the cursor body statement plus parameter formats and value
 **     areas.
 **
 **     If a first-time open is successful, OpenCursor() can be used to
 **     reopen the cursor with new parameter values.
 **
 **     For later opens, the cursor must be closed.
 **
```

```
** Returns
**   CS_SUCCEED or CS_FAIL
*/

CS_RETCODE
 OpenCursor(cmd, cur_control)
 CS_COMMAND      *cmd;
 CUR_CONTROL     *cur_control;
 {
   CS_RETCODE    ret;
   CS_INT        i;
   CS_DATAFMT    dfmt;
   LANGPARAM     *params;
   CS_BOOL       have_restorable_cursor;

  /*
   ** Check whether a cursor-open command can be restored with this
   ** command handle.
   */
   ret = ct_cmd_props(cmd, CS_GET, CS_HAVE_CUROPEN,
                      &have_restorable_cursor, CS_UNUSED,
                      (CS_INT *)NULL);
   if (ret != CS_SUCCEED)
   {
     ex_error("OpenCursor: ct_cmd_props() failed!");
     return CS_FAIL;
   }

  /*
   ** If CS_HAVE_CUROPEN is CS_FALSE, then this is a first-time open. So,
   ** we initiate a new declare command and bind to the parameter source
   ** variables in the CUR_CONTROL structure.
   */
   if (have_restorable_cursor != CS_TRUE)
   {

    /*
     ** Initiate the declare command.
     */
     ret = ct_cursor(cmd, CS_CURSOR_DECLARE,
                     cur_control->name, CS_NULLTERM,
                     cur_control->query, CS_NULLTERM,
                     CS_UNUSED);
     if (ret != CS_SUCCEED)
     {
       ex_error("OpenCursor: Initiate-declare failed");
       return CS_FAIL;
     }
```

```
    /*
     ** Specify formats for the host language parameters in the cursor
     ** declare command.
     */
     params = cur_control->params;
     (CS_VOID *)memset(&dfmt, 0, sizeof(dfmt));
     dfmt.status = CS_INPUTVALUE;

    for (i = 0; i < cur_control->numparams; i++)
     {
       dfmt.datatype = params[i].type;
       dfmt.maxlength = params[i].maxlen;
       strcpy(dfmt.name, params[i].name);
       dfmt.namelen = strlen(dfmt.name);

      ret = ct_setparam(cmd, &dfmt,
                        (CS_VOID *)NULL, (CS_INT *)NULL,
                        (CS_SMALLINT *)NULL);
       if (ret != CS_SUCCEED)
       {
         ex_error("OpenCursor: ct_setparam() failed");
         return CS_FAIL;
       }
     }
   }

  /*
   ** Initiate or restore the cursor-open command.
   **
   ** The first time we open the cursor, this call initiates an
   ** open-cursor command which gets batched with the declare command.
   ** Since there is no cursor to restore, ct_cursor ignores the
   ** CS_RESTORE_OPEN option.
   **
   ** The second (and later) times we open the cursor, this call
   ** restores the cursor-open command so that we can send it again.
   ** The declare-cursor command (originally batched with the open
   ** command) is not restored.
   */
   ret = ct_cursor(cmd, CS_CURSOR_OPEN,
                   (CS_CHAR *)NULL, CS_UNUSED,
                   (CS_CHAR *)NULL, CS_UNUSED,
                   CS_RESTORE_OPEN);
   if (ret != CS_SUCCEED)
   {
     ex_error("OpenCursor: Initiate-open failed.");
     return CS_FAIL;
   }
```

```
/*
 ** For the first-time open, supply the address of variables that have
 ** values for the cursor parameters. These variables will be read by
 ** ct_send.
 **
 ** The second (and later) times we open the cursor, we don't have to
 ** call ct_setparam here -- the parameter bindings were restored by
 ** ct_cursor(OPEN, RESTORE_OPEN).
 **
 ** In either case, we assume that our caller has already set the
 ** desired values, lengths, and indicators.
 */
 for (i = 0;
      ((have_restorable_cursor != CS_TRUE) &&
       (i < cur_control->numparams));
      i++)
 {
   dfmt.datatype = params[i].type;
   dfmt.maxlength = params[i].maxlen;
   strcpy(dfmt.name, params[i].name);
   dfmt.namelen = strlen(dfmt.name);

  ret = ct_setparam(cmd, &dfmt,
                    params[i].data, &params[i].len,
                    &params[i].indicator);
   if (ret != CS_SUCCEED)
   {
     ex_error("OpenCursor: ct_setparam() failed");
     return CS_FAIL;
   }
 }

/*
 ** Send the command batch.
 */
 ret = ct_send(cmd);
 if (ret != CS_SUCCEED)
 {
   ex_error("OpenCursor: ct_send() failed.");
   return CS_FAIL;
 }

/*
 ** GetToCursorRows() calls ct_results() until cursor rows are
 ** fetchable on the command structure. GetToCursorRows() fails if
 ** the declare or open command fails on the server.
 */
 ret = GetToCursorRows(cmd);
```

```
    if (ret != CS_SUCCEED)
    {
      ex_error("OpenCursor: Cursor could not be opened.");
      return CS_FAIL;
    }

   return CS_SUCCEED;
  } /* OpenCursor() */

/*
 ** GetToCursorRows() -- Flush results from a cursor-open command
 **   batch until ct_results returns a CS_CURSOR_RESULT result type.
 **
 ** Parameters
 **   cmd -- The command handle to read results from.
 **
 ** Returns
 **   CS_SUCCEED -- Cursor rows are ready to be fetched.
 **   CS_FAIL -- Failure. Could be due to any of the following:
 **     - No cursor results in the results stream.
 **     - Other kinds of fetchable results in the results stream.
 **     - ct_results failure.
 */

CS_STATIC CS_RETCODE
 GetToCursorRows(cmd)
 CS_COMMAND       *cmd;
 {
   CS_RETCODE       results_ret;
   CS_RETCODE       ret;
   CS_INT           result_type = CS_END_RESULTS;
   CS_BOOL          failing = CS_FALSE;
   CS_INT           intval;
   CS_CHAR          scratch[512];

   while (((results_ret = ct_results(cmd, &result_type)) == CS_SUCCEED)
           && (result_type != CS_CURSOR_RESULT))
    {
      switch ((int)result_type)
      {
      case CS_CMD_SUCCEED:
      case CS_CMD_DONE:
        break;

     case CS_CMD_FAIL:
        /*
        ** Declare or open failed on the server.
        */
```

```
        ret = ct_res_info(cmd, CS_CMD_NUMBER, (CS_VOID *)&intval,
                          CS_UNUSED, (CS_INT *) NULL);
        if (ret == CS_SUCCEED)
        {
          sprintf(scratch, “Command %ld failed”, (long)intval);
          ex_error(scratch);
        }
        failing = CS_TRUE;
        break;

     default:
        /*
        ** Nothing else is expected. Just return fail and let the caller
        ** decide how to clean up.
        */
        ex_error(
           “Unexpected result types received for cursor declare/open.”);
        return CS_FAIL;

    }
    }
   /*
    ** We are leaving the cursor results pending on the connection.
    */
    if (results_ret == CS_CANCELED)
    {

     /*
      ** Could happen if the connection has a timeout and the error
      ** handler did ct_cancel(CS_CANCEL_ATTN);
      */
      ex_error(“Cursor declare/open was canceled.”);
      failing = CS_TRUE;
    }
    else if (results_ret != CS_SUCCEED)
    {
      ex_error(“Cursor declare/open:ct_results failed.”);
      failing = CS_TRUE;
    }

   return (failing == CS_TRUE) ?CS_FAIL :CS_SUCCEED;
  } /* GetToCursorRows() */
```

**使用法**

- ct_setparam は、サーバ・コマンドの入力パラメータ値に使用するプログラム・ソース変数を指定します。

- 実行の最初の手順としてコマンドを開始します。一部のコマンドでは、アプリケーションで ct_param または ct_setparam を使用して入力パラメータを定義してから、ct_send を呼び出してコマンドをサーバに送信する必要があります。
- ct_setparam と ct_param は、次の点を除いて同じ機能を実行します。
  - ct_param はプログラム変数の内容をコピーします。
  - ct_setparam はプログラム変数のアドレスをコピーして、ct_send に対する以後の呼び出しは変数の内容を読み込みます。ct_setparam を使用することによって、アプリケーションはコマンドの再送時にパラメータ値を変更できます。この機能については、「コマンドの再送信」(643 ページ ) を参照してください。

  ct_param と ct_setparam の呼び出しは混在させることができます。
- ct_setparam は次のような場合に必要です。
  - ct_cursor で開始したカーソル・オープン・コマンドまたはカーソル更新コマンド、ct_command で開始した言語、メッセージ、または RPC コマンド、ct_dynamic で開始した動的 SQL 実行コマンドの入力パラメータ値を指定する場合。ct_setparam のこの使用法については、「入力パラメータ・ソースを定義するための ct_setparam の使用法」(669 ページ ) を参照してください。
  - ct_cursor または ct_dynamic で開始したカーソル宣言コマンドに対する、ホスト言語変数フォーマットのフォーマットを定義する場合。ct_setparam のこの使用法については、「カーソル・パラメータのフォーマットを定義するための ct_setparam の使用法」(671 ページ ) を参照してください。カーソル宣言コマンドは再送できないので、パラメータ・フォーマットを定義するために ct_param ではなく ct_setparam を使用する利点はありません。
  - カーソル宣言コマンド (ct_cursor または ct_dynamic で開始 ) に対して更新カラムを定義する場合。ct_setparam のこの使用法については、「更新可能なカーソル・カラムを識別するための ct_setparam の使用法」(672 ページ ) を参照してください。カーソル宣言コマンドは再送できないので、更新カラムを定義するために ct_param ではなく ct_setparam を使用する利点はありません。

- Client-Library は、パラメータをサーバに渡す前に、パラメータを変換しません。したがって、アプリケーションはサーバが要求するデータ型でパラメータを入力する必要があります。アプリケーションは、必要に応じて cs_convert を呼び出してパラメータ値を必要なデータ型に変換できます。

**入力パラメータ・ソースを定義するための *ct_setparam* の使用法**

- アプリケーションは、以下のコマンドの入力パラメータ値を指定することが必要な場合があります。
  - Client-Library のカーソル・オープン・コマンド
  - Client-Library のカーソル更新コマンド
  - 動的 SQL 実行コマンド
  - 言語コマンド
  - メッセージ・コマンド
  - パッケージ・コマンド
  - RPC コマンド
- ct_setparam は、**data*、**datalenp*、および **indp* として渡される変数と 1 つのコマンド・パラメータ間のバインドを作成します。ct_send に対する以後の呼び出しは、パラメータ値が null かどうか判断し、null でない場合は現在の値と長さを調べるために、これらの変数の内容を読み込みます。次の場合、値は null であるとみなされます。
  - **datalen* が 0 の場合
  - **indp* が -1 の場合
  - ct_setparam の呼び出しで *data*、*datalenp*、および *indp* をすべて NULL として渡した場合。
- ct_setparam の各呼び出しに関連するコマンド・パラメータは、名前または位置によって指定します。
  - 名前で指定する場合は、パラメータの名前に *datafmt–>name*、名前の長さに *datafmt–>namelen* を設定します。
  - 位置で指定する場合は、各呼び出しに対して *datafmt–>namelen* を 0 にして、SQL 文またはストアド・プロシージャ定義内でパラメータが出現する順序で ct_setparam を呼び出します。

  すべてのパラメータを名前で指定するか、またはすべてのパラメータを位置で指定してください。

- Client-Library のカーソル・オープン・コマンドは、次の場合に入力パラメータ値が必要です。
  - ホスト言語変数を含んでいる Transact-SQL の select 文でカーソルが宣言される場合。
  - Transact-SQL の execute 文でカーソルが宣言されて、呼び出されるストアド・プロシージャがパラメータを必要とする場合。この場合、入力パラメータを指定するために **datafmt–>status* を CS_INPUTVALUE にしてください。
  - 疑問符で示されるプレースホルダを含む準備された動的 SQL 文に関して、カーソルが宣言される場合。
- 更新コマンドを表す SQL テキストにホスト変数が含まれている場合、Client-Library のカーソル更新コマンドには入力パラメータ値が必要です。
- 実行対象の準備文に疑問符 (?) で示される動的パラメータ・マーカが含まれている場合、動的 SQL 実行コマンドには入力パラメータ値が必要です。
- 言語コマンドのテキストにホスト変数が含まれている場合、言語コマンドには入力パラメータ値が必要です。
- メッセージでパラメータが使用されている場合、メッセージ・コマンドには入力パラメータ値が必要です。
- 実行するストアド・プロシージャまたはパッケージでパラメータが使用されている場合、RPC コマンドとパッケージ・コマンドには入力パラメータ値が必要です。
- メッセージ、RPC、パッケージの各コマンドは、リターン・パラメータを持つことがあります。リターン・パラメータは *datafmt–>status* を CS_RETURN として渡すことで示されます。
- リターン・パラメータを使用するコマンドは、リターン・パラメータ値を含むパラメータ結果セットを生成する場合があります。アプリケーションがパラメータ結果セットから値を取得する方法については、「ct_results」を参照してください。
- 表 3-60 に、入力パラメータ値を渡すときに使用される **datafmt* のフィールドの一覧を示します。ct_setparam が戻った後にパラメータのフォーマットを変更することはできません。

**表 3-60：入力パラメータ値の受け渡しにおける CS_DATAFMT フィールド**

| フィールド | 説明 |
|---|---|
| *name* | パラメータの名前。<br>*name* は動的 SQL 実行コマンドの場合は無視される。 |
| *namelen* | *name* のバイト単位の長さ、または名前のないパラメータであることを示す 0。<br>*namelen* は動的 SQL 実行コマンドの場合は無視される。 |
| *datatype* | 入力パラメータ値のデータ型。<br>CS_TEXT_TYPE、CS_IMAGE_TYPE、および Client-Library のユーザ定義データ型を除く、Client-Library の標準データ型はすべて有効。<br>*datatype* が CS_VARCHAR_TYPE または CS_VARBINARY_TYPE の場合、*data* は CS_VARCHAR または CS_VARBINARY 構造体を指す必要がある。 |
| *maxlength* | RPC コマンドにリターン・パラメータを渡す場合、*maxlength* はこのパラメータに対して返されるデータの最大バイト長を表す。<br>ct_setparam *datalenp* パラメータが NULL として渡される場合、*maxlength* はパラメータに対する全入力値の長さも指定する。このとき、対応するリターン・パラメータ・データの最大長は入力値の長さと一致する必要がある。 |
| *status* | RPC コマンドにリターン・パラメータを渡す場合は CS_RETURN、その他の場合は CS_INPUTVALUE に設定する。 |

他のすべてのフィールドは無視される。

### カーソル・パラメータのフォーマットを定義するための *ct_setparam* の使用法

- ホスト言語変数を含んでいる select 文を使用してカーソルを宣言する場合は、アプリケーションはカーソル宣言コマンドに対してホスト変数フォーマットを定義する必要があります。
- ホスト変数フォーマットは、ct_cursor(CS_CURSOR_DECLARE) を呼び出してカーソル宣言コマンドを開始した後に、ct_param または ct_setparam を使用して定義します。カーソル宣言コマンドは再送できないので、この場合には ct_setparam が ct_param よりも有利な点はありません。
- ct_setparam を使用してホスト変数のフォーマットを定義するには、アプリケーションは *datafmt–>status* を CS_INPUTVALUE、*datafmt–>datatype* をホスト変数のデータ型、*data*、*datalenp*、および *indp* を NULL にして渡します。

- アプリケーションはカーソル宣言コマンドの一部としてホスト変数のフォーマットを定義しますが、カーソルに対するカーソル・オープン・コマンドを開始するまでは変数に対してデータ値を指定しません。
- ホスト変数のフォーマットを定義する場合、ct_setparam の各呼び出しに関連するホスト言語変数は、名前 (*datafmt –>name* と *datafmt –> namelen* をそれぞれ設定 ) または ct_setparam と ct_param の呼び出しの順序 (*datafmt-> namelen* を 0 に設定 ) によって指定できます。変数を 1 つでも名前で指定する場合は、すべての変数を名前で指定してください。
- 次の表は、ホスト変数のフォーマットを定義する場合に使用する **datafmt* 内のフィールドを示します。

**表 3-61：ホスト変数フォーマットを定義するための CS_DATAFMT フィールド**

| フィールド | 説明 |
|---|---|
| *name* | ホスト変数の名前。 |
| *namelen* | *name* のバイト単位の長さ、または名前のないパラメータであることを示す 0。 |
| *datatype* | ホスト変数のデータ型。<br>CS_TEXT_TYPE、CS_IMAGE_TYPE、および Client-Library のユーザ定義データ型を除く、Client-Library の標準データ型はすべて有効。 |
| *status* | CS_INPUTVALUE。 |

他のすべてのフィールドは無視される。

#### 更新可能なカーソル・カラムを識別するための *ct_setparam* の使用法

- 一部のカラムが更新可能な場合、サーバによっては、カーソル宣言コマンドに対する更新カラムの識別をクライアント・アプリケーションに要求することもあります。更新カラムは、基本となるデータベース・テーブルの値の変更に使用できます。
- Adaptive Server Enterprise は、この項で説明する別の ct_param 呼び出しや ct_setparam 呼び出しを使用した更新カラムの指定を、アプリケーションに対して要求しません。実際、Adaptive Server Enterprise はここで説明する更新カラムの識別要求を無視します。アプリケーションは、select 文に Transact-SQL の for read only 構文または for update of 構文を使用して、更新可能なカラム ( 存在する場合 ) を指定する必要があります ( この構文については、Adaptive Server Enterprise を参照してください )。設計によっては、この項で説明するように、Open Server アプリケーションがクライアントに対してカーソルの更新カラムを指定するよう要求する場合もあります。

- カーソルのすべてのカラムが更新可能である場合、アプリケーションは ct_param や ct_setparam を呼び出してそれらを個別に指定する必要はありません。
- カーソル宣言コマンド用に更新カラムを識別するために、アプリケーションは *datafmt–>status* に CS_UPDATECOL、**data* にカラムの名前を使用して ct_param または ct_setparam を呼び出します。
- 次の表は、カーソル宣言コマンド用に更新カラムを識別するために ct_setparam を呼び出すときに使用する、**datafmt* 内のフィールドを示します。

**表 3-62：更新カラムを識別するための CS_DATAFMT フィールド**

| フィールド名 | 設定内容 |
|---|---|
| *status* | CS_UPDATECOL。 |

他のすべてのフィールドは無視される。

**参照**

ct_command、ct_cursor、ct_dynamic、ct_param、ct_send

# ct_wakeup

**説明**

接続の完了コールバックを呼び出します。

**構文**

```
CS_RETCODE ct_wakeup(connection, cmd, function,
               status)

CS_CONNECTION    *connection;
CS_COMMAND       *cmd;
CS_INT           function;
CS_RETCODE       status;
```

**パラメータ**

*connection*

完了コールバックを呼び出す CS_CONNECTION 構造体を指すポインタです。CS_CONNECTION 構造体は、特定のクライアント／サーバ接続の情報を含んでいます。

*connection* か *cmd* のどちらかを NULL 以外にしてください。

*connection* が指定された場合、その完了コールバックが呼び出されます。*connection* が NULL の場合、*cmd* の親接続の完了コールバックが呼び出されます。

*connection* が指定された場合、それは完了コールバックに *connection* パラメータとして渡されます。*connection* が NULL の場合、*cmd* の親接続が完了コールバックの *connection* パラメータとして渡されます。

*cmd*

クライアント／サーバ・オペレーションを管理する CS_COMMAND 構造体を指すポインタです。

*connection* か *cmd* のどちらかを NULL 以外にしてください。

*connection* が NULL の場合、*cmd* の親接続の完了コールバックが呼び出されます。

*cmd* は、完了コールバックに *command* パラメータとして渡されます。*cmd* が NULL の場合、*command* パラメータには NULL が渡されます。

*function*

どのルーチンが完了したかを示す記号値です。*function* はユーザ定義の値も指定できます。*function* は、完了コールバックの *function* パラメータとして渡されます。表 3-63 に、*function* の有効な記号値を示します。

**表 3-63：ct_wakeup function パラメータの値**

| function の値 | 意味 |
|---|---|
| BLK_ROWXFER | blk_rowxfer が完了した。 |
| BLK_SENDROW | blk_sendrow が完了した。 |
| BLK_SENDTEXT | blk_sendtext が完了した。 |
| BLK_TEXTXFER | blk_textxfer が完了した。 |
| CT_CANCEL | ct_cancel が完了した。 |
| CT_CLOSE | ct_close が完了した。 |
| CT_CONNECT | ct_connect が完了した。 |
| CT_DS_LOOKUP | ct_ds_lookup が完了した。 |
| CT_FETCH | ct_fetch が完了した。 |
| CT_GET_DATA | ct_get_data が完了した。 |
| CT_OPTIONS | ct_options が完了した。 |
| CT_RECVPASSTHRU | ct_recvpassthru が完了した。 |
| CT_RESULTS | ct_results が完了した。 |
| CT_SEND | ct_send が完了した。 |
| CT_SEND_DATA | ct_send_data が完了した。 |
| CT_SENDPASSTHRU | ct_sendpassthru が完了した。 |
| ユーザ定義の値：この値は、CT_USER_FUNC 以上でなくてはならない。 | ユーザ定義の関数が完了した。 |

*status*

完了したルーチンのリターン・ステータスです。この値は、完了コールバックに *status* パラメータとして渡されます。

**戻り値**

ct_wakeup は、次の値を返します。

| 戻り値 | 意味 |
|---|---|
| CS_SUCCEED | ルーチンが正常に終了した。 |
| CS_FAIL | ルーチンが失敗。 |
| CS_BUSY | この接続では、非同期オペレーションが保留中である。「非同期プログラミング」(12 ページ ) を参照。 |

**例**

```
...CODE DELETED.....
/* Force a wakeup on the connection handle */
 retstat = ct_wakeup(connection, NULL,
     EX_ASYNC_QUERY, status);
 if (retstat != CS_SUCCEED)
     {
          return retstat;
     }

   ...CODE DELETED.....
```

このコードは、*ex_alib.c* サンプル・プログラムからの抜粋です。

**使用法**

- ct_wakeup は、Client-Library の最上部に非同期レイヤを作成するアプリケーションで使用します。
- CS_DISABLE_POLL プロパティが CS_TRUE に設定されている場合、アプリケーションは ct_wakeup を呼び出すことはできません。

**参照**

「非同期プログラミング」(12 ページ )、「コールバック」(25 ページ )、ct_callback、ct_poll

付 録 A

# 国際化ライブラリのメッセージ

この付録では、国際化ライブラリのエラー・メッセージについて説明します。

## INTE_NOVAL

**影響を受けた Open Server/SDK コンポーネント**

両方

**メッセージ**

```
Syntax error:no value found.
```

**考えられる原因**

- *$SYBASE/locales* 内の *common.loc* ファイルにある [datetime] セクションで、「firstday」、「dateformat」または 「timeformat」の値を検出できません。
- *$SYBASE/locales* 内の *locales.loc* ファイルにある [file format] セクションで、「version」、list_seperator 文字、またはエスケープ文字の値を検出できません。

**アクション / 解決法**

上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。

**追加情報**

**このエラーが発生したバージョン**

すべて

## INTE_NOENTRY

**影響を受けた Open Server/SDK コンポーネント**

両方

**メッセージ**

```
Syntax error:no entry found.
```

**考えられる原因** ロケール・ファイルのセクションで有効なエントリを取得できません。

**アクション / 解決法** 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。

**追加情報**

**このエラーが発生したバージョン** すべて

# INTE_OFLOW

**影響を受けた Open Server/SDK コンポーネント** 両方

**メッセージ**

```
Copying string would result in overflow of buffer.
```

**考えられる原因** バッファのサイズが小さくて文字列を保管できません。たとえば、*$SYBASE* のパス名などの文字列です。

**アクション / 解決法** 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。

**追加情報**

**このエラーが発生したバージョン** すべて

# INTE_ENTRYOF

**影響を受けた Open Server/SDK コンポーネント** 両方

**メッセージ**

```
エントリが長すぎます。
```

**考えられる原因** ロケール・ファイルのエントリが長すぎます。現在のエントリの最大長は 64 です。

**アクション / 解決法** 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。

**追加情報**

**このエラーが発生したバージョン** すべて

# INTE_ODDHEX

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Odd number of hex digits in localization file.` |
| 考えられる原因 | 2 つの 16 進数が予期されますが、1 つしか取得されていません。たとえば、ロケール・ファイルの通貨フォーマットは 2 つの 16 進数です。ロケール・ファイルから読み込まれた 16 進数が 1 つのみの場合に、このエラーがレポートされます。 |
| アクション / 解決法 | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| 追加情報 | |
| このエラーが発生したバージョン | すべて |

# INTE_BADFILE

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Bad file pointer.` |
| 考えられる原因 | ロケール・ファイルへのファイル・ポインタが無効です。ファイルを開くために使用する UNIX のシステム・ルーチン open() または Windows のシステム・ルーチン _open() により、このポインタが返されます。 |
| アクション / 解決法 | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| 追加情報 | |
| このエラーが発生したバージョン | すべて |

# INTE_BADLOC

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Bad INTL_LOCFILE pointer.` |
| 考えられる原因 | INTL_LOCFILE 構造体へのポインタが無効です。このエラーはアプリケーションを実行するためのメモリが十分でない場合、または無効なメモリ・アクセスが存在する場合に発生します。 |
| アクション / 解決法 | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| 追加情報 | |
| このエラーが発生したバージョン | すべて |

# INTE_NOCOM

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Bad INTL_LOCFILE pointer.` |
| 考えられる原因 | 構文エラー：コメント文字を取得できません。 |
| アクション / 解決法 | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| 追加情報 | |
| このエラーが発生したバージョン | すべて |

# INTE_BADFFMT

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Syntax error in file format section of locfile.` |

| | |
|---|---|
| **考えられる原因** | ローカライゼーション・ファイルにフォーマット・エラーが存在します。 |
| **アクション / 解決法** | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| **追加情報** | |
| **このエラーが発生したバージョン** | すべて |

# INTE_BADVER

| | |
|---|---|
| **影響を受けた Open Server/SDK コンポーネント** | 両方 |
| **メッセージ** | `Bad version number.` |
| **考えられる原因** | ローカライゼーション・ファイルのバージョン番号が正しくありません。 |
| **アクション / 解決法** | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| **追加情報** | |
| **このエラーが発生したバージョン** | すべて |

# INTE_BADPH

| | |
|---|---|
| **影響を受けた Open Server/SDK コンポーネント** | 両方 |
| **メッセージ** | `Unable to build string:illegal place holder found in text string.` |
| **考えられる原因** | このエラーは、ユーザ API intl_strblist() によりレポートされます。変数のプレースホルダを含むテキストから出力可能な文字列を構築するときに、不正なプレースホルダが検出されます。 |

| | |
|---|---|
| **アクション / 解決法** | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| **追加情報** | |
| **このエラーが発生したバージョン** | すべて |

# INTE_BADTYPE

| | |
|---|---|
| **影響を受けた Open Server/SDK コンポーネント** | 両方 |
| **メッセージ** | `Unknown datatype token in intl_strbuild() or intl_strblist().` |
| **考えられる原因** | 変数のプレースホルダを含むテキストから出力可能な文字列を構築するときに、不明なデータ型が検出されます。 |
| **アクション / 解決法** | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| **追加情報** | |
| **このエラーが発生したバージョン** | すべて |

# INTE_SPECOF

| | |
|---|---|
| **影響を受けた Open Server/SDK コンポーネント** | 両方 |
| **メッセージ** | `Custom format specifier too long.` |
| **考えられる原因** | 変数のプレースホルダを含むテキストから出力可能な文字列を構築するときに、カスタムのフォーマット指定子の長さが INTL_MAXSPECLEN (20) より長くなります。 |
| **アクション / 解決法** | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| **追加情報** | |
| **このエラーが発生したバージョン** | すべて |

# INTE_NOCUST

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `No custom format specifier found to match specifier in formats string.` |
| 考えられる原因 | 変数のプレースホルダを含むテキストから出力可能な文字列を構築するときに、カスタムのフォーマット指定子がフォーマット・リストにあるどの指定子とも一致しません。 |
| アクション / 解決法 | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| 追加情報 | |
| このエラーが発生したバージョン | すべて |

# INTE_BADFMTSTR

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Null format string parameter to intl_fmtinstall().` |
| 考えられる原因 | 変数のプレースホルダを含むテキストから出力可能な文字列を構築するとき、カスタムのフォーマット指定子が空です。 |
| アクション / 解決法 | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| 追加情報 | |
| このエラーが発生したバージョン | すべて |

# INTE_INVALBUF

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Null buffer.` |
| 考えられる原因 | バッファを指すポインタが Null です。 |
| アクション / 解決法 | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| 追加情報 | |
| このエラーが発生したバージョン | すべて |

# INTE_NEGBUFLEN

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Negative buffer length.` |
| 考えられる原因 | バッファの長さが負の値です。 |
| アクション / 解決法 | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| 追加情報 | |
| このエラーが発生したバージョン | すべて |

# INTE_INVALCS

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Null charset directory.` |

| | |
|---|---|
| **考えられる原因** | 文字セットの値のポインタが NULL です。考えられる原因は、bcp などのアプリケーションを実行するときに文字セットが適切に設定されていないことです。 |
| **アクション / 解決法** | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| **追加情報** | |
| **このエラーが発生したバージョン** | すべて |

# INTE_BADLFNM

| | |
|---|---|
| **影響を受けた Open Server/SDK コンポーネント** | 両方 |
| **メッセージ** | `Null localization file name.` |
| **考えられる原因** | ローカライゼーション・ファイル、エラー・メッセージ、または設定ファイルの名前が NULL になっています。 |
| **アクション / 解決法** | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| **追加情報** | |
| **このエラーが発生したバージョン** | すべて |

# INTE_INVALTEXT

| | |
|---|---|
| **影響を受けた Open Server/SDK コンポーネント** | 両方 |
| **メッセージ** | `Null text string.` |
| **考えられる原因** | intllib がテキスト文字列を構築して、それをバッファへ格納しようとするとき、テキスト・バッファを指すポインタが NULL であることを検出しました。 |

| | |
|---|---|
| **アクション / 解決法** | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| **追加情報** | |
| **このエラーが発生したバージョン** | すべて |

# INTE_INVALSRC

| | |
|---|---|
| **影響を受けた Open Server/SDK コンポーネント** | 両方 |
| **メッセージ** | `intl_xlate():Null source string.` |
| **考えられる原因** | 文字列を 1 つの文字セットから別の文字セットに変換しようとするとき、ソース文字列が NULL であることが検出されます。 |
| **アクション / 解決法** | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| **追加情報** | |
| **このエラーが発生したバージョン** | すべて |

# INTE_INVALPTR

| | |
|---|---|
| **影響を受けた Open Server/SDK コンポーネント** | 両方 |
| **メッセージ** | `Null pointer.` |
| **考えられる原因** | 文字列を 1 つの文字セットから別の文字セットに変換しようとするとき、変換のステータスを保存する際に変数に使用されたポインタなどの NULL ポインタが検出されます。 |
| **アクション / 解決法** | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| **追加情報** | |
| **このエラーが発生したバージョン** | すべて |

# INTE_BADNSTARS

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `intl_strblist():Only 0, 1, or 2 stars allowed in format string.` |
| 考えられる原因 | 変数のプレースホルダやそれらに入力する値の配列を含むフォーマット文字列から、出力可能な文字列が構築されます。このエラーは、フォーマット文字列に 2 つを超えるスターが存在する場合に発生します。 |
| アクション / 解決法 | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| 追加情報 | |
| このエラーが発生したバージョン | すべて |

# INTE_MONTHS

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Too few months in datetime section.` |
| 考えられる原因 | *$SYBASE/locales* 内の *common.loc* ファイルにある [datetime] セクションの「months」アイテムが 12 カ月より小さい値で検出されます。 |
| アクション / 解決法 | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| 追加情報 | |
| このエラーが発生したバージョン | すべて |

# INTE_SMONTHS

**影響を受けた Open Server/SDK コンポーネント** 両方

**メッセージ**

```
Too few short months in datetime section.
```

**考えられる原因** *$SYBASE/locales* 内の *common.loc* ファイルにある [datetime] セクションの「shortmonths」アイテムが 12 カ月より小さい値で検出されます。

**アクション / 解決法** 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。

**追加情報**

**このエラーが発生したバージョン** すべて

# INTE_DAYS

**影響を受けた Open Server/SDK コンポーネント** 両方

**メッセージ**

```
Too few short days in datetime section.
```

**考えられる原因** *$SYBASE/locales* 内の *common.loc* ファイルにある [datetime] セクションの「day」アイテムが 7 日より小さい値で検出されます。

**アクション / 解決法** 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。

**追加情報**

**このエラーが発生したバージョン** すべて

# INTE_PATHOF

**影響を受けた Open Server/SDK コンポーネント** 両方

**メッセージ**

```
Pathname too long.
```

| | |
|---|---|
| 考えられる原因 | Sybase ホーム・ディレクトリのパス名などのパス名が INTL_MAXPATHLEN ( 現在値は 512) より長くなっています。 |
| アクション / 解決法 | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| 追加情報 | |
| このエラーが発生したバージョン | すべて |

## INTE_LTLONG

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Line in localization file too long.` |
| 考えられる原因 | ローカライゼーション・ファイルの長さが INTL_MAXLINE ( 現在値は 1,024) より長くなっています。 |
| アクション / 解決法 | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| 追加情報 | |
| このエラーが発生したバージョン | すべて |

## INTE_DUPDF

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Duplicate dateformat value.` |
| 考えられる原因 | *$SYBASE/locales* 内の *common.loc* ファイルにある [datetime] セクションの「dateformat」アイテムを読み込む際に、重複した値が検出されます。有効なフォーマット文字列は、「m」、「d」および 「y」を 1 つのみ使用する 3 文字の文字列にする必要があります。たとえば、「mdy」や「dmy」などです。フォーマット文字列で「m」、「d」または 「y」が 2 つ以上検出されると、このエラーがレポートされます。 |

**アクション / 解決法** 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。

**追加情報**

**このエラーが発生したバージョン** すべて

## INTE_BADSECT

**影響を受けた Open Server/SDK コンポーネント** 両方

**メッセージ**

```
Syntax error in section heading.
```

**考えられる原因** *$SYBASE/locales* 内の *common.loc* ファイルにある [datetime] などのセクション名に構文エラーが存在します。

**アクション / 解決法** 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。

**追加情報**

**このエラーが発生したバージョン** すべて

## INTE_FOPEN

**影響を受けた Open Server/SDK コンポーネント** 両方

**メッセージ**

```
Unable to open file.
```

**考えられる原因** ファイルを開く際にシステム・エラーが発生します。

**アクション / 解決法** 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。

**追加情報**

**このエラーが発生したバージョン** すべて

## INTE_FCLOSE

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Unable to close file.` |
| 考えられる原因 | ファイルを閉じる際にシステム・エラーが発生します。 |
| アクション / 解決法 | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| 追加情報 | |
| このエラーが発生したバージョン | すべて |

## INTE_FREAD

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Unable to read file.` |
| 考えられる原因 | ファイルから読み込む際にシステム・エラーが発生します。 |
| アクション / 解決法 | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| 追加情報 | |
| このエラーが発生したバージョン | すべて |

## INTE_NOSYB

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Couldn't find the user 'sybase'.` |
| 考えられる原因 | *$SYBASE* ディレクトリを検出できませんでした。 |

| | |
|---|---|
| **アクション / 解決法** | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| **追加情報** | |
| **このエラーが発生したバージョン** | すべて |

# INTE_FINFO

| | |
|---|---|
| **影響を受けた Open Server/SDK コンポーネント** | 両方 |
| **メッセージ** | `Unable to access file information.` |
| **考えられる原因** | システム・ルーチン stat() を呼び出して、ファイルの情報の取得を試みると、システム・エラーが発生します。 |
| **アクション / 解決法** | 上記のエラー・メッセージが表示され、ユーザ API により致命的なエラーが返されます。 |
| **追加情報** | |
| **このエラーが発生したバージョン** | すべて |

# INTE_NOMEM

| | |
|---|---|
| **影響を受けた Open Server/SDK コンポーネント** | 両方 |
| **メッセージ** | |
| **考えられる原因** | メモリの割り付けができません。 |
| **アクション / 解決法** | アプリケーション内の予期しないエラーです。エラー・メッセージは発行されません。致命的エラーのみが返されます。 |
| **追加情報** | |
| **このエラーが発生したバージョン** | すべて |

付 録　B

# SSL エラー・メッセージ

この付録では、SSL のエラー・メッセージについて説明します。

## 1: ベンダの呼び出しの失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Vendor call failed` |
| 考えられる原因 | CSI ファクトリを正常に作成できなかったか、SYBCSI API 関数の呼び出しに失敗しました。 |
| アクション / 解決法 | 適切な OpenSSL サードパーティのライブラリが *$SYBASE/$SYBASE_OCS/lib3p* (32 ビット ) または *$SYBASE/$SYBASE_OCS/lib3p64* (64 ビット ) に格納されていることを確認してください。 |
| 追加情報 | IBM Power Linux、Windows 64 ビットなどの OpenSSL を使用するプラットフォームにのみ適用します。 |
| このエラーが発生したバージョン | 15.5 |

## 3: メモリ割り付けの失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Memory allocation error` |
| 考えられる原因 | メモリ割り付けルーチンがメモリの必要サイズの割り付けに失敗しました。 |
| アクション / 解決法 | 使用可能なメモリを増設します。 |
| 追加情報 | |
| このエラーが発生したバージョン | 15.5 |

# 6: 不正なポインタ

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Bad pointer` |
| 考えられる原因 | SSL リモート認証プロパティを取得する際に、SSL コンテキスト・ポインタ ( 認証を保持するためのポインタ ) が空です。 |
| アクション / 解決法 | コンテキスト・ポインタ ( 認証を保持するためのポインタ ) をチェックしてください。 |
| 追加情報 | |
| このエラーが発生したバージョン | 15.5 |

# 60: SSL マスタ・コンテキスト初期化の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Could not initialize SSL/TLS Master Context` |
| 考えられる原因 | SSL マスタ・コンテキストの作成に失敗し、SSLInitialize() が失敗を返します。 |
| アクション / 解決法 | SSLInitialize() の引数を検証してください。 |
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

# 61: 部分 I/O の設定の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Could not initialize SSL/TLS Master Context` |

| | |
|---|---|
| 考えられる原因 | SSL マスタ・コンテキストの作成に失敗し、SSLInitialize() が失敗を返します。 |
| アクション / 解決法 | SSLInitialize() の引数を検証してください。 |
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

# 62: SSL プロトコル・バージョンの設定の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Could not set SSL/TLS protocol version` |
| 考えられる原因 | SSL プロトコル・バージョンを設定できず、SSLSetProtocolVersion() が失敗を返します。 |
| アクション / 解決法 | バージョン番号の正当性を検証してください。 |
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

# 63: ランダム番号ジェネレータの作成の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Could not create PRNG object` |
| 考えられる原因 | ランダム番号ジェネレータを作成できず、EZCreateObject() が失敗を返します。 |
| アクション / 解決法 | バージョン番号の正当性を検証してください。 |
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

# 64: ランダム番号ジェネレータの初期化の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Could not initialize PRNG object` |
| 考えられる原因 | ランダム番号ジェネレータを初期化できず、EZInitObject() が失敗を返します。 |
| アクション / 解決法 | EZCreateObject() の引数を検証してください。 |
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

# 65: ランダム番号ジェネレータのエントロピの生成の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Could not seed PRNG` |
| 考えられる原因 | ランダム番号ジェネレータのエントロピの生成に失敗しました。 |
| アクション / 解決法 | |
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

# 69: コンテキストの複製の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Could not duplicate SSL/TLS context information` |
| 考えられる原因 | SSL コンテキストを複製できず、SSLDuplicateContext() が失敗を返します。 |
| アクション / 解決法 | SSLDuplicateContext() の引数を検証してください。 |

| | |
|---|---|
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

# 70: 子 SSL/TLS コンテキストの作成の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Could not create a child SSL/TLS context` |
| 考えられる原因 | 新しいフィルタ・セッションを複製できず、SSLDuplicateContext() が失敗を返します。 |
| アクション / 解決法 | SSLDuplicateContext() の引数を検証してください。 |
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

# 71: プロトコル・バージョンの取得の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Could not get SSL/TLS protocol version` |
| 考えられる原因 | ネゴシエートされた SSL/TLS プロトコル・バージョンを取得できず、SSLGetProtocolVersion() が失敗を返します。 |
| アクション / 解決法 | SSLGetProtocolVersion() の引数を検証してください。 |
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

# 72: 不明なプロトコル・バージョン

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Unknown SSL/TLS protocol version` |
| 考えられる原因 | SSLGetProtocolVersion() から返されたバージョン番号が無効です。 |
| アクション / 解決法 | 適切な SSL ドライバが使用されているかチェックしてください。 |
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

# 73: 不明な暗号

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Set cipher saw no recognized cipher suites` |
| 考えられる原因 | インストールされた暗号が、SSL でサポートされている暗号スイートのものと異なります。 |
| アクション / 解決法 | インストールされた暗号の有効性をチェックしてください。 |
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

# 74: 暗号スイートの設定の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Error setting cipher suites` |
| 考えられる原因 | 暗号リストの設定に失敗し、SSLSetCipherSuites() が失敗を返します。 |
| アクション / 解決法 | SSLSetCipherSuites() の引数を検証してください。 |

| | |
|---|---|
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

# 75: ローカル identity プロパティのロードの失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Attempt to load local identity failed` |
| 考えられる原因 | ローカル identity のロードに失敗し、SSLLoadLocalIdentity() が失敗を返します。 |
| アクション / 解決法 | SSLLoadLocalIdentity() の引数を検証してください。 |
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

# 76: 認証局のファイルのロードまたは読み込みの失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Unable to open certificate authority file or read the certificates in it` |
| 考えられる原因 | 認証局のファイルのロードまたは読み込みに失敗し、SSLLoadTrustedCertificateFile() が失敗を返します。 |
| アクション / 解決法 | SSLLoadLocalIdentity() の引数および認証局のファイルの正当性を検証してください。 |
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

# 77: ピアの認証情報取得の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Could not get length of peer's certificate` |
| 考えられる原因 | ピアの認証の長さのフェッチに失敗し、SSLGetPeerCertificateChainLength() が失敗を返します。 |
| アクション / 解決法 | リモート接続で確実に SSL 接続を確立できるようにしてください。 |
| 追加情報 | |
| このエラーが発生したバージョン | 15.5 |

# 78: ピアの認証取得の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Could not get peer's certificate` |
| 考えられる原因 | ピアの認証のフェッチに失敗しました。 |
| アクション / 解決法 | リモート接続で確実に SSL 接続を確立できるようにしてください。 |
| 追加情報 | IBM Power Linux、Windows 64 ビットなどの OpenSSL を使用するプラットフォームにのみ適用します。 |
| このエラーが発生したバージョン | 15.5 |

# 81: 認証リファレンスの設定の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Could not set TLS/SSL certificate reference` |

| | |
|---|---|
| 考えられる原因 | 認証リファレンス・ポインタの設定に失敗し、SSLSetCheckCertificateRef() が失敗を返します。 |
| アクション / 解決法 | SSLSetCheckCertificateRef() の引数を検証し、セッション・ポインタが適切であることを確認してください。 |
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

## 84: SSL ハンドシェイクの失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `TLS/SSL handshake failed. Check certificate for possible corruption.` |
| 考えられる原因 | SSL ハンドシェイクに失敗し、SSLHandshake() が失敗を返します。 |
| アクション / 解決法 | 認証の破損の可能性をチェックし、SSLHandshake() の引数を検証してください。 |
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

## 85: SSL のサーバ側への設定の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Could not set TLS/SSL to server side` |
| 考えられる原因 | SSL のサーバ側への設定に失敗し、SSLSetProtocolSide() が失敗を返します。 |
| アクション / 解決法 | SSLSetProtocolSide() の引数を検証してください。 |
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

# 86: SSL のクライアント側への設定の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Could not set TLS/SSL to client side` |
| 考えられる原因 | SSL のクライアント側への設定に失敗し、SSLSetProtocolSide() が失敗を返します。 |
| アクション / 解決法 | SSLSetProtocolSide() の引数を検証してください。 |
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

# 87: SSL のエンドポイント情報取得の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `I/O attempted before TLS/SSL was set up` |
| 考えられる原因 | SSL のエンドポイント情報が Null です。 |
| アクション / 解決法 | ネットワーク・フィルタが適切に設定されているかチェックしてください。 |
| 追加情報 | |
| このエラーが発生したバージョン | 15.5 |

# 88: SSL コンテキスト情報取得の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `I/O attempted before SSL Handshake` |
| 考えられる原因 | SSL のコンテンツ情報は Null です。 |

| | |
|---|---|
| アクション / 解決法 | ネットワーク・フィルタが適切に設定されているかチェックしてください。 |
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

## 89: 読み込みエラー

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Error during read` |
| 考えられる原因 | SSL の読み込みプロセスのエラーのため、SSLRead() が失敗を返します。 |
| アクション / 解決法 | SSLRead() の引数を検証してください。 |
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

## 90: 書き込みエラー

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Error during write` |
| 考えられる原因 | SSL の書き込みプロセスのエラーのため、SSLWrite() が失敗を返します。 |
| アクション / 解決法 | SSLWrite() の引数を検証してください。 |
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

## 91: リモート認証の DN フィールドのカウント取得の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Could not get count of remote certificate’s DN fields` |
| 考えられる原因 | リモート認証フィールドカウントの取得に失敗し、SSLCountSubjectDNFields() または sybcsi_x509_get_subjectname_count() が失敗を返します。 |
| アクション / 解決法 | リモート認証の正当性をチェックしてください。 |
| 追加情報 | |
| このエラーが発生したバージョン | 15.5 |

## 92: 識別名情報の抽出の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Could not extract Distinguished Name information from remote certificate` |
| 考えられる原因 | リモート認証の識別名情報の取得に失敗し、SSLExtractSubjectDNFieldIndex() または sybcsi_x509_get_subjectname_by_index() が失敗を返します。 |
| アクション / 解決法 | リモート認証の正当性をチェックしてください。 |
| 追加情報 | |
| このエラーが発生したバージョン | 15.5 |

## 93: リモート認証の拡張子のカウント取得の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Could not count remote certificate’s extensions` |

| | |
|---|---|
| 考えられる原因 | リモート認証の拡張子カウントの取得に失敗し、SSLCountExtensions() または sybcsi_x509_get_extension_count() が失敗を返します。 |
| アクション / 解決法 | リモート認証の正当性をチェックしてください。 |
| 追加情報 | |
| このエラーが発生したバージョン | 15.5 |

## 94: 拡張子情報の抽出の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Could not extract extension information from remote certificate` |
| 考えられる原因 | リモート認証の拡張子情報取得に失敗し、SSLExtractExtensionIndex()、SSLGetPeerCertificateRef()、sybcsi_x509_get_extension_by_index()、または sybcsi_x509_list_get_element() が失敗を返します。 |
| アクション / 解決法 | リモート認証の正当性をチェックしてください。 |
| 追加情報 | |
| このエラーが発生したバージョン | 15.5 |

## 95: クライアント認証の取得の失敗

| | |
|---|---|
| 影響を受けた Open Server/SDK コンポーネント | 両方 |
| メッセージ | `Require client certificate call failed` |
| 考えられる原因 | クライアント認証の取得に失敗し、SSLSetRequestClientCert() が失敗を返します。 |
| アクション / 解決法 | クライアントの認証をチェックしてください。 |
| 追加情報 | OpenSSL を使用しないプラットフォームに適用されます。 |
| このエラーが発生したバージョン | 15.5 |

# 用語解説

**Adaptive Server Enterprise**
: Sybase のクライアント／サーバ・アーキテクチャにおけるサーバ。Adaptive Server Enterprise は、複数のデータベースと複数のユーザを管理します。ディスク上にあるデータの実際のロケーションを監視し、論理データ記述から物理データ記憶領域へのマッピングを管理します。メモリ内のデータ・キャッシュとプロシージャ・キャッシュの保守も行います。

**array**
: 複数の同じタイプの変数からなる構造体。各変数は、個々にアドレッシングされます。

**配列バインド (array binding)**
: 結果カラムを配列変数にバインドする処理。フェッチのときは、複数のロー分のカラムが変数にコピーされます。

**非同期 (asynchronous)**
: プログラムのメイン制御フローとは関係なく、いつでも発生すること。「**同期 (synchronous)**」と比較してください。Client-Library には、「**遅延非同期 (deferred-asynchronous)**」と「**完全非同期 (fully asynchronous)**」という 2 つの非同期オペレーション・モードがあります。

**非同期ルーチン (asynchronous routine)**
: Client-Library において、ネットワークと対話するルーチン。非同期ルーチンは CS_PENDING を返すことができます。

**バッチ (batch)**
: コマンドまたは文の集まり。

Client-Library のコマンド・バッチは、ct_send へのアプリケーションの呼び出しで終了する、1 つ以上の Client-Library のコマンドです。たとえば、アプリケーションは、カーソルに対する宣言、ローの選択、オープンを実行する複数のコマンドをまとめてバッチ処理することができます。

Transact-SQL 文バッチは、1 つの Client-Library コマンドまたは Embedded SQL 文によって Adaptive Server Enterprise に送信される 1 つ以上の Transact-SQL 文です。

**ブラウズ・モード**
: DB-Library と Client-Library アプリケーションが、一度に 1 つのローの値を更新しながらデータベース・ローをブラウズする方法。同様の機能を果たすカーソルの方が、一般に扱いやすくなっています。

**バルク・コピー (bulk copy)**
データベース・テーブルへのデータ転送の高速化のために Adaptive Server Enterprise が提供するネットワーク・インタフェース。bcp ユーティリティを使用すると、管理者はバルク・コピー・インタフェースを使用して、データをデータベースにまたはデータベースからコピーできます。Client-Library と Server-Library のプログラマは、「**Bulk-Library**」を使用してこのインタフェースにアクセスできます (DB-Library プログラムには、DB-Library に構築される Bulk-Copy 特殊ライブラリを使用してください。

**バルク記述子構造体**
Bulk-Library がバルク・コピー・オペレーションを管理するために使用する隠し制御構造体 (CS_BLKDESC)。「**Bulk-Library**」、「**バルク・コピー (bulk copy)**」も参照してください。

**Bulk-Library**
Client-Library と Server-Library アプリケーションが、Adaptive Server Enterprise バルク・コピー・インタフェースにアクセスできるようにするためのルーチンの集まり。「**バルク・コピー (bulk copy)**」も参照してください。

**bylist**
サブグループにソートされる結果セット。bylist は、compute 句を使用して、キーワード by の後にカラムのリストを指定することによって生成されます。

**コールバック (callback)**
コールバック・イベントと呼ばれるトリガ・イベントに応答して、Open Client または Open Server が呼び出すルーチン。

**コールバック・エラー処理 (callback error handling)**
Client-Library アプリケーションにおけるエラーの処理方法の 1 つ。Client-Library または CS-Library でエラーが検出されたときや、サーバからサーバ・メッセージが送信されたときに呼び出される「**コールバック**」関数をプログラムがインストールします。「**クライアント・メッセージ・コールバック (client message callback)**」、「**サーバ・メッセージ・コールバック (server message callback)**」、「**CS-Library エラー・ハンドラ (CS-Library error handler)**」、「**インライン・エラー処理 (inline error handling)**」も参照してください。

**コールバック・イベント (callback event)**
Open Client と Open Server におけるコールバック・ルーチンをトリガするイベント。

**機能**

クライアント／サーバ接続に対して使用される「**TDS (Tabular Data Stream)**」通信プロトコルのバージョンがサポートする機能のセット。Client-Library アプリケーションは、ct_capability を呼び出して、接続がクライアント要求またはサーバ応答の特定のタイプをサポートするかどうかをチェックします。クライアント・アプリケーションは、接続がオープンされる前に ct_capability を呼び出して、特定のタイプのサーバ応答をサポートしないこともできます。接続がオープンされたときに機能は決定され、その後は変更できません。「**オプション (options)**」、「**プロパティ (properties)**」も参照してください。

**文字セット (character set)**

各文字をユニークに定義するコード化スキームを持つ特定の ( 通常、標準化された ) 文字の集まり。ASCII と ISO 8859-1 (Latin 1) は、よく使用される文字セットです。「**文字セット変換 (character set conversion)**」、「**クライアント文字セット (client character set)**」も参照してください。

**文字セット変換 (character set conversion)**

サーバへ入出力するときの文字セットのコード化スキームの変換。サーバとクライアントが異なる「**文字セット (character set)**」を使用して通信するとき、変換が行われます。たとえば、サーバが ISO 8859-1 を使用し、クライアントが Code Page 850 を使用する場合、文字セット変換をオンにして、サーバとクライアントが、受け渡しされるデータを同じように解釈するようにします。

**クライアント (client)**

クライアント／サーバ・システムにおいて、サーバへ要求を送り、この要求に対する結果に対して処理を行う部分。

**クライアント文字セット (client character set)**

クライアント／サーバ・システムにおいて、クライアント・アプリケーションが使用する「**文字セット (character set)**」。クライアント文字セットはサーバが使用する文字セットとは異なってもかまいません。「**文字セット変換 (character set conversion)**」も参照してください。

**Client-Library**

Open Client の一部で、クライアント・アプリケーションを記述するためのルーチンの集まり。Client-Library は、カーソル、分散ネットワーク・サービス、およびその他の高度な機能を取り込むように設計されているライブラリです。「**CS-Library**」、「**Bulk-Library**」も参照してください。

**クライアント・メッセージ (client message)**

Client-Library が生成するエラーまたは情報メッセージの 1 つ。アプリケーションのエラーまたは例外の状態を知らせます。「**クライアント・メッセージ・コールバック (client message callback)**」、「**インライン・エラー処理 (inline error handling)**」、「**コールバック・エラー処理 (callback error handling)**」も参照してください。

**クライアント・メッセージ・コールバック**
アプリケーション・ルーチンの 1 つ。Client-Library がエラーまたは異常の状態を説明するクライアント・メッセージを生成するたびに呼び出されます。「**コールバック・エラー処理 (callback error handling)**」も参照してください。

**コード・セット (code set)**
「**文字セット (character set)**」を参照してください。

**照合順 (collating sequence)**
「**ソート順 (sort order)**」を参照してください。

**コマンド (command)**
Client-Library において、ct_command、ct_dynamic、または ct_cursor に対するアプリケーションの呼び出しで始まり、ct_send に対するアプリケーションの呼び出しで終了するサーバ要求。

**コマンド構造体**
Client-Library アプリケーションがコマンドの送信と結果の処理に使用する Client-Library の隠し制御構造体。

**共通名 (common name)**
同じ親ノードを持つエントリ間でだけユニークな名前。「**完全に修飾された名前 (fully qualified name)**」を参照してください。

**完了コールバック**
Client-Library アプリケーションにおける、アプリケーションの「**コールバック (callback)**」ルーチンのタイプの 1 つ。「**完全非同期 (fully asynchronous)**」接続の場合、Client-Library は自動的にアプリケーションの完了コールバックを起動して、「**非同期ルーチン (asynchronous routine)**」に対する各呼び出しの「**完了ステータス (completion status)**」を伝達します。

**完了ステータス (completion status)**
Client-Library アプリケーションにおいて、「**非同期ルーチン (asynchronous routine)**」に対する呼び出しの最後のリターン・ステータス。同期接続では、すべての必要なネットワーク対話が完了するまで非同期ルーチンはブロックします。そして、完了ステータスを直接返します。非同期接続では、非同期ルーチンは CS_PENDING をすぐに返すので、アプリケーションはポーリングまたは「**完了コールバック (completion callback)**」によって完了ステータスを確認する必要があります。「**同期 (synchronous)**」、「**遅延非同期 (deferred-asynchronous)**」、「**完全非同期 (fully asynchronous)**」も参照してください。

**接続構造体**
コンテキスト内にクライアント／サーバ接続を定義する Client-Library の隠し制御構造体。「**コマンド構造体 (command structure)**」も参照してください。

**コンテキスト構造体**

Client-Library または Open Server アプリケーション内でアプリケーション「コンテキスト」または操作環境を定義する CS-Library の隠し構造体。「**CS-Library**」ルーチンの cs_ctx_alloc と cs_ctx_drop は、それぞれコンテキスト構造体の割り付けと解除を行います。

**変換 (conversion)**

データ値をある表現から別の表現に変換する動作。変換によって、異なるデータ型の新しい値を得られます。また文字から文字への変換では、異なるフォーマットまたは異なる「**文字セット (character set)**」の新しい値を得られます。「**文字セット変換 (character set conversion)**」も参照してください。

**クレデンシャル・トークン (credential token)**

ユーザがネットワーク・セキュリティ・システムに対して自身の ID を証明してから、ユーザが接続を試行するネットワークベース認証。次に、Client-Library はセキュリティ・メカニズムからクレデンシャル・トークンを取得して、パスワードの代わりにそのトークンをサーバに送信します。

**クリティカル・セクション (critical section)**

マルチスレッド・アプリケーションにおいて、複数のスレッド内で同時に実行できないコードのセクション。一般的に、クリティカル・セクションは複数のスレッドが共有するリソースにアクセスするコードです。同じ共有リソースにアクセスするクリティカル・セクションは、「関連する」と言われます。「**スレッド・シリアライゼーション (thread serialization)**」、「**スレッドセーフ (thread-safe)**」も参照してください。

**CS-Library**

Client-Library と Server-Library のアプリケーションの両方で役立つユーティリティ・ルーチンの集まり。Open Client および Open Server の両方に含まれています。「**コンテキスト構造体 (context structure)**」も参照してください。

**CS-Library エラー・ハンドラ**

CS-Library がエラーまたは例外の状態を説明するエラー・メッセージを生成するたびに呼び出されるアプリケーション・ルーチン。CS-Library エラー・ハンドラは **Server-Library** アプリケーションで必要ですが、**Client-Library** アプリケーションでも使用することをおすすめします。「**コールバック・エラー処理 (callback error handling)**」も参照してください。

**現在のロー (current row)**

カーソルが置かれているロー。フェッチは、カーソルに対して現在のローを取得します。「**カーソル (cursor)**」も参照してください。

**カーソル (cursor)**

select 文に関連付けられた記号名。カーソルは「ロー・ポインタ」と選択条件が一致するローのセットとを対応させます。

Transact-SQL において、言語カーソルは declare cursor 言語コマンドで宣言され、fetch 言語コマンドでスクロールされるカーソルです。

Client-Library において、Client-Library カーソルは ct_cursor(CS_CURSOR_DECLARE) で作成されたサーバ・オブジェクトです。アプリケーションは ct_fetch を使用して Client-Library カーソルをスクロールします。

Embedded SQL において、カーソルは複数ローのデータをホスト・プログラムに渡すデータ・セレクタです。このローの受け渡しは、一度に 1 つずつ行われます。

「**スクロール可能カーソル (scrollable cursor)**」も参照してください。

**データベース (database)**

特別な目的のために組織化された、関連するデータ・テーブルとその他のデータベース・オブジェクトの集まり。

**データ型 (datatype)**

変数に有効な値と演算を表す定義属性。

**DB-Library**

Open Client の一部で、クライアント・アプリケーションの記述に使用する独立したルーチンの集合。DB-Library は、DB-Library で作成された以前の Open Client アプリケーションに対してソースコードの互換性を提供します。

**デッドロック (deadlock)**

1. Adaptive Server Enterprise 内でそれぞれのデータにロックを保持している 2 人のユーザが、互いに他方のデータのロックを獲得しようとしたときに起こる状態。Adaptive Server Enterprise がデッドロックを検出すると、片方のユーザのプロセスを強制終了することによってこの状態を解決します。

2. マルチスレッド・アプリケーション内でシリアライゼーション・プリミティブを、それぞれが制御している 2 つのスレッドがある場合、一方のスレッドが所有するシリアライゼーション・プリミティブを、もう一方のスレッドがロックしようとする状況。デッドロックによって、マルチスレッド・アプリケーションがフリーズする場合があります。

**デフォルト (default)**

1. Open Client または Open Server アプリケーションで、指定されないときに Open Client/Open Server 製品が使用する値、オプション、または動作。

2. Transact-SQL で、insert 文が値を指定しない場合にカラムに挿入される値。

**デフォルト・データベース (default database)**

ユーザが、データベース・サーバにログインしたとき、デフォルトで指定されるデータベース。

**デフォルト言語 (default language)**

1. アプリケーションに対してローカライゼーションの指定を行わないとき、Open Client/Server 製品が使用する言語。デフォルト言語は、ロケール・ファイルの「default」エントリにより決定されます。

2. ユーザが言語を選択しなかったとき、Adaptive Server Enterprise がメッセージとプロンプトに使用する言語。

**遅延非同期 (deferred-asynchronous)**
Client-Library 接続用のオペレーションの非同期モードの 1 つ。アプリケーションが非同期ルーチンに対する各呼び出しの完了ステータスをポーリングする必要があります。「**完全非同期 (fully asynchronous)**」と比較してください。

**記述子領域**
動的 SQL 文内の動的パラメータについての情報を格納するために、データベース管理システム (DBMS) が使用する領域。

**ディレクトリ**
ユニークな名前とネットワーク・エンティティ ( サーバ、プリンタ、ユーザなど ) について格納された情報とを関連付ける辞書。ディレクトリへのアクセスには、「**ディレクトリ・サービス・プロバイダ (directory service provider)**」が必要です。「**interfaces ファイル (interfaces file)**」も参照してください。

**ディレクトリ・ドライバ (directory driver)**
ディレクトリ・エントリをそのネイティブの記憶領域フォーマットからサーバ・ディレクトリ・オブジェクトのフォーマットに変換します。

**ディレクトリ・エントリ (directory entry)**
指定された「**完全に修飾された名前 (fully qualified name)**」と関連する情報が格納されます。

**ディレクトリ・オブジェクト・クラス (directory object class)**
「**ディレクトリ・エントリ (directory entry)**」内に格納されている属性 ( データ ) を設定するための指定。

**ディレクトリ・オブジェクト構造体**
Client-Library アプリケーションにおいて、ct_ds_lookup ルーチンに対する呼び出しによって読み込まれた「**ディレクトリ・エントリ (directory entry)**」のコピーを含む隠し構造体 ( データ型 CS_DS_OBJECT)。

**ディレクトリ・サービス**
ディレクトリ・エントリの作成、修正、取得を管理します ( ネーミング・サービスと呼ばれる場合もあります )。

**ディレクトリ・サービス・プロバイダ (directory service provider)**
アプリケーションに「**ディレクトリ (directory)**」へのアクセスを提供するシステム・ソフトウェア。Windows などのプラットフォームでは、ディレクトリ・サービス・プロバイダはオペレーティング・システム内に構築されます。その他のプラットフォームでは、DCE などのサード・パーティのプロバイダを使用できるように設定されているシステムもあります。

**DIT ベース (DIT base)**
逆ツリー構造を持つディレクトリ内では、内部ノードの名前。DIT ベース名は部分的に修飾された名前と結合され、完全に修飾された名前を作成します。「**ディレクトリ (directory)**」、「**完全に修飾された名前 (fully qualified name)**」も参照してください。

**動的 SQL**
Embedded SQL または Client-Library アプリケーションで、名前と準備された SQL 文を対応させせることができます。一度準備すれば、SQL 文を名前で繰り返し実行できます。また、実行時に値が決定される変数を含むことができます。Adaptive Server Enterprise において、ユーザが接続を終了するときに、準備された動的 SQL 文は自動的に削除されます。「**ストアド・プロシージャ (stored procedure)**」と比較してください。

**エラー・メッセージ (error message)**
Open Client/Server 製品がエラー状態を検出したときに発行するメッセージ。

**イベント (event)**
Server-Library アプリケーションに何らかの動作を行うよう要求するオカレンス。クライアント・コマンドおよび Open Server アプリケーション・コードの特定のコマンドは、イベントをトリガできます。イベントが発生すると、Server-Library は、アプリケーション・コード内の適切なイベント処理ルーチン、または適切なデフォルト・イベント・ハンドラのどちらかを呼び出します。「**イベント・ハンドラ (event handler)**」、「**イベント駆動型プログラミング (event-driven programming)**」も参照してください。

**イベント駆動型プログラミング (event-driven programming)**
Open Server アプリケーション用のプログラミング・スタイル。アプリケーションがイベントの各クラスに対してイベント・ハンドラを提供し、Open Server スレッド・スケジューラがアプリケーションのイベント・ハンドラを呼び出して、イベントを「ディスパッチ」します。「**イベント (event)**」、「**イベント・ハンドラ (event handler)**」、「**Open Server スレッド (Open Server thread)**」も参照してください。

**イベント・ハンドラ (event handler)**
Open Server におけるイベントを処理するルーチン。Open Server アプリケーションは、Open Server が提供するデフォルト・ハンドラを使用することができます。また、カスタマイズしたイベント・ハンドラをインストールすることもできます。「**イベント (event)**」、「**イベント駆動型プログラミング (event-driven programming)**」も参照してください。

**実行カーソル (execute cursor)**
ストアド・プロシージャとともに宣言されるカーソル。

**公開された構造体 (exposed structure)**
内部が Open Client/Server プログラマに公開されている構造体。Open Client/Open Server プログラマは、公開された構造体の宣言、操作、割り付け解除を直接行うことができます。公開された構造体には、CS_DATAFMT 構造体などがあります。

**拡張トランザクション**
Embedded SQL における、複数の Embedded SQL 文からなるトランザクション。

**FIPS**

Federal Information Processing Standards ( 連邦情報処理標準 ) の略。FIPS フラグが有効なとき、Adaptive Server Enterprise および Embedded SQL プリコンパイラは、標準でない拡張された SQL 文を検出すると警告を発行します。

**完全非同期 (fully asynchronous)**

Client-Library 接続に対するオペレーションの非同期モードの 1 つ。「**非同期ルーチン (asynchronous routine)**」に対する各呼び出しが完了すると、アプリケーションに自動でノーティフィケーションが送信されます。「**遅延非同期 (deferred-asynchronous)**」、「**シグナル駆動型 I/O (signal-driven I/O)**」、「**スレッド駆動型 I/O (thread-driven I/O)**」も参照してください。

**完全に修飾された名前 (fully qualified name)**

「**ディレクトリ・エントリ (directory entry)**」を識別するユニークで明確な名前。エントリの完全に修飾された名前は、「**ディレクトリ・サービス・プロバイダ (directory service provider)**」がエントリの検索に必要とする情報すべてを提供します。

**ゲートウェイ (gateway)**

直接通信できないクライアントとサーバとの仲介として動作するアプリケーション。ゲートウェイ・アプリケーションは、クライアントとサーバの両方として動作します。クライアントからの要求をサーバに渡し、サーバからの結果をクライアントに返します。

**グローバル名 (global name)**

OID は分散環境内のすべてのアプリケーションに対して同じ「シンボリック・グローバル名」として機能します。「**オブジェクト識別子 (object identifier)**」を参照してください。

**隠し構造体 (hidden structure)**

内部が Open Client/Server プログラマに対して隠されている構造体。Open Client/Server プログラマは、隠し構造体の割り付け、操作、割り付け解除を行うために、Open Client/Server ルーチンを使用しなければなりません。隠し構造体には、CS_CONTEXT 構造体などがあります。

**ホスト言語 (host language)**

アプリケーションを記述するときに使われるプログラミング言語。

**ホスト・プログラム (host program)**

Embedded SQL における、Embedded SQL コードを含むアプリケーション・プログラム。

**ホスト変数 (host variable)**

Embedded SQL における、Adaptive Server Enterprise とアプリケーション・プログラム間のデータ転送を可能にする変数。「**インジケータ変数 (indicator variable)**」、「**入力変数 (input variable)**」、「**出力変数 (output variable)**」、「**結果変数 (result variable)**」、「**ステータス変数 (status variable)**」も参照してください。

**インジケータ変数 (indicator variable)**

他の変数の値やフェッチしたデータについての特別な条件を示す変数。

Embedded SQL ホスト変数とともに使用すると、インジケータ変数は、データベースの値が null である箇所を示します。

**開始済みコマンド (initiated command)**
Client-Library アプリケーションにおいて ct_command、ct_cursor、または ct_dynamic によってコマンドのタイプが定義されている場合、コマンドは開始されますが、まだ ct_send を使用してサーバに送信されていません。「**コマンド (command)**」も参照してください。

**インライン・エラー処理**
Client-Library アプリケーションにおける、エラーの処理方法の 1 つ。CS-Library または Client-Library ルーチンの各呼び出し後に、プログラムが「**クライアント・メッセージ (client message)**」、CS-Library エラー、または「**サーバ・メッセージ (server message)**」のオカレンスをテストします。「**コールバック・エラー処理 (callback error handling)**」と比較してください。

**入力変数 (input variable)**
情報をルーチン、ストアド・プロシージャ、または Adaptive Server Enterprise に渡すときに使う変数。

**interfaces ファイル (interfaces file)**
サーバ名をトランスポート・アドレスにマップするファイル。クライアント・アプリケーションが、サーバに接続するために ct_connect または dbopen を呼び出すと、Client-Library または DB-Library が *interfaces* ファイルからサーバのアドレスを検索します。この用途では、Client-Library は *interfaces* ファイルの代わりに「**ディレクトリ・サービス (directory service)**」も使用できます。*interfaces* ファイルを使用しないプラットフォームもあります。interfaces ファイルを使用しないプラットフォームでは、別のメカニズムでクライアントにサーバ・アドレスを知らせます。

**割り込み駆動型 I/O (interrupt-driven I/O)**
「**シグナル駆動型 I/O (signal-driven I/O)**」を参照してください。

**isql スクリプト・ファイル (isql script file)**
Embedded SQL において、プリコンパイラが生成できる 3 つのファイルのうちの 1 つ。isql スクリプト・ファイルには、Transact-SQL で記述されるプリコンパイラが生成したストアド・プロシージャが含まれます。

**キー (key)**
ローをユニークに識別するロー・データのサブセット。キー・データは、オープンされたカーソル内の「**現在のロー**」をユニークに記述します。

**keytab ファイル**
オペレーティング・システム・ファイルへの名前とパス。

**キーワード (keyword)**
Transact-SQL または Embedded SQL で排他的に利用するように予約されているワードまたはフレーズ。「予約語 (reserved word)」とも呼ばれます。

**リスナ**
「**Open Server**」アプリケーションにおいて、クライアントからの接続試行を待ち、クライアント接続を処理するための新しいスレッドを作成する、内部 Server-Library システム・スレッド。srv_init に対する呼び出しでリスナは起動します。

**リスティング・ファイル (listing file)**
Embedded SQL において、プリコンパイラが生成できる 3 つのファイルのうちの 1 つ。リスティング・ファイルには、入力ファイルのソース文と、情報、警告、エラーなどのメッセージが含まれます。

**ロケール名**
言語と文字セットのペアを表す文字列。ロケール名は「**ロケール・ファイル (locales file)**」にリストされています。Sybase があらかじめ定義しているロケール名の他に、「**システム管理者 (system administrator)**」が別のロケール名を定義し、ロケール・ファイルに追加することもできます。

**ロケール構造体 (locale structure)**
Client-Library または Open Server アプリケーションのためのカスタム・ローカライゼーション値を定義する CS-Library の隠し構造体。アプリケーションは、CS_LOCALE 構造体を使用して、使用される言語、文字セット、日付順、ソート順を定義できます。ロケール構造体の割り付けと割り付け解除には、CS-Library ルーチン cs_loc_alloc および cs_loc_drop を使用します。

**ロケール・ファイル**
ロケール名を言語と文字セットのペアにマッピングするファイル。Open Client/Server 製品は、ローカライゼーション情報をロードするときにこのロケール・ファイルを調べます。

**ローカライゼーション (localization)**
アプリケーションを特定のネイティブ言語環境で使用するために設定する処理のこと。ローカライズされたアプリケーションは、通常、各国の言語と文字セットでメッセージを作成し、その国の日時表記フォーマットを使用します。

**論理コマンド (logical command)**
Client-Library アプリケーションにおいて、ct_command、ct_dynamic、または ct_cursor によって定義されたコマンドのことです。その他に、次のような例外もあります。

- ストアド・プロシージャ内の各 Transact-SQL の 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 文は論理コマンドには含めません。
- 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。
- 言語コマンド内の各 Transact-SQL 文は論理コマンドです。

**ログイン認証 (login authentication)**
ユーザ名とパスワードを使用することによって、ユーザが本人であることを確認するセキュリティ・サービス。

**ログイン名**

ユーザが、Adaptive Server Enterprise にログインするときに使用する名前。Adaptive Server Enterprise のログイン名が有効になるのは、Adaptive Server Enterprise がシステム・テーブル *syslogins* にそのユーザのエントリを持つ場合です。

**メッセージ番号 (message number)**

エラー・メッセージをユニークに識別する番号。

**メッセージ・キュー (message queue)**

Open Server において、スレッドが通信するとき使用するメッセージ・ポインタのリンク・リスト。スレッドは、キューにメッセージを書き込んだり、キューからメッセージを読み込んだりすることができます。

**マルチバイト文字セット (multibyte character set)**

複数のバイトを使用してコード化された文字を含む文字セット。マルチバイト文字セットには、EUC JIS、シフト JIS などがあります。

**マルチスレッド**

プログラム・コードのプロパティ。マルチスレッドのコードは複数のスレッドで同時に実行できます。したがって、「**スレッドセーフ (thread-safe)**」でなければなりません。「**スレッド (thread)**」も参照してください。

**mutex**

相互排他セマフォ。つまり、mutex は、Server-Library またはオペレーティング・システム・スレッド・インタフェースが提供する「**シリアライゼーション・プリミティブ (serialization primitive)**」です。mutex は、マルチスレッド・アプリケーションが複数のスレッドに共有されているリソースへのアクセスを逐次化するための方法を提供します。「**ネイティブ・スレッド (native thread)**」、「**Open Server スレッド (Open Server thread)**」も参照してください。

**ネーミング・サービス (naming service)**

「**ディレクトリ・サービス・プロバイダ (directory service provider)**」を参照してください。

**ネイティブ・スレッド (native thread)**

ホスト・オペレーティング・システムによってその存在とスケジュールが管理される「**スレッド (thread)**」。「**スレッド・スケジューリング (thread scheduling)**」、「**Open Server スレッド (Open Server thread)**」も参照してください。

**ネゴシエートされたプロパティ**

TDS バージョンのサポートに関するプロパティなど、サーバがログイン・プロセス中に変更できる特定のログイン・プロパティ。

**NULL**

1. データ値に関しては、明示的に割り付けられた値を持たないこと。NULL は、0 でもブランクでもありません。NULL の値は、他の値と比べて大きいとも、小さいとも、同じであるともみなされません。他の NULL と比べるときも同じです。

2. C 言語ポインタに関しては、メモリのアドレスと関連していない特別な NULL アドレス値。

**オブジェクト識別子 (object identifier)**
マルチベンダ、マルチプラットフォーム環境にあるオブジェクトをユニークに命名する 10 進数の文字列。OID は異なる環境にある異なる名前を持つ項目を識別するための手段です。たとえば、同じ文字セットを異なるオペレーティングシステムで別々に命名することができます。「**グローバル名 (global name)**」を参照してください。

オブジェクト識別子は、ISO 8825 が定義した BER (Basic Encoding Rules) に従ってコード化されます。Sybase が定義した OID はすべて次のプレフィクスで始めます。

```
1.3.6.1.4.1.897
```

**OID**
「**オブジェクト識別子 (object identifier)**」を参照してください。

**OID 文字列 (OID string)**
「**オブジェクト識別子 (object identifier)**」を含む文字列。Client-Library と Server-Library アプリケーションは、OID 文字列を使用してオブジェクト識別子を表します。*cspublic.h* ヘッダ・ファイルが Sybase 固有の OID 文字列を定義します。

**Open Server**
カスタム・サーバを作成するためのツールとインタフェースを提供する Sybase 製品。「**Server-Library**」も参照してください。

**Open Server アプリケーション**
「**Open Server**」で構築されたカスタム・サーバ。「**Open Server スレッド (Open Server thread)**」、「**イベント駆動型プログラミング (event-driven programming)**」も参照してください。

**Open Server スレッド (Open Server thread)**
「**Open Server**」アプリケーションとライブラリ・コード、およびパスに関連するスタック・スペース、ステータス情報、イベント・ハンドラを介した実行のパス。Open Server スレッドは、**Server-Library** によってその存在とスケジュールが管理されるスレッドです。「**スレッド (thread)**」、「**スレッド・スケジューリング (thread scheduling)**」、「**ネイティブ・スレッド (native thread)**」も参照してください。

**オプション**
Adaptive Server Enterprise のコマンド処理を制御するソフトウェア。アプリケーションはサーバへの接続がオープンされてから ct_options Client-Library ルーチンを呼び出して、オプションを設定、取得、またはクリアします。「**機能 (capabilities)**」、「**プロパティ (properties)**」も参照してください。

**出力変数 (output variable)**
Embedded SQL において、ストアド・プロシージャからアプリケーション・プログラムにデータを渡す変数。

**パラメータ (parameter)**
1. データをルーチンに渡すとき、およびルーチンからデータを取得するときに使用する変数。

2. ストアド・プロシージャへの引数。

**パススルー・モード (passthrough mode)**

ゲートウェイが、クライアントとリモート・データ・ソース間の TDS (Tabular Data Stream) パケットを、そのパケットの内容をアンパックすることなく中継するモード。

**プレースホルダ (placeholder)**

文中で疑問符 (?) で識別され、準備文の中で変数のような役割を果たすインジケータ。

**プロパティ**

隠し構造体に格納される名前付きの値。コンテキスト構造体、接続構造体、スレッド構造体、コマンド構造体は、プロパティを持ちます。構造体のプロパティは、ポインタをパラメータとして構造体に渡す呼び出しに対する CS-Library、Client-Library、または Server-Library の応答を決定します。「**機能 (capabilities)**」、「**オプション (options)**」も参照してください。

**クエリ (query)**

1. データの検索要求。通常は select 文です。

2. データを操作する任意の SQL 文。

**レジスタード・プロシージャ (registered procedure)**

Open Server アプリケーションにおいて、クライアントがリモートで呼び出すことができる、サーバ上の実行可能なエンティティ。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 C 関数、システム・レジスタード・プロシージャとして使用できる内部 Server-Library ルーチン、またはクライアントによる sp_regcreate システム・レジスタード・プロシージャの呼び出しによって作成された、「プログラム本体を持たない」レジスタード・プロシージャです。「**レジスタード・プロシージャ・ノーティフィケーション (registered procedure notifications)**」、「**システム・レジスタード・プロシージャ (system registered procedure)**」、「**リモート・プロシージャ・コール (remote procedure call)**」も参照してください。

**レジスタード・プロシージャ・ノーティフィケーション (registered procedure notifications)**

クライアントが特定の「**レジスタード・プロシージャ (registered procedure)**」の実行を監視できるようにする Open Server 機能。クライアントが Open Server のレジスタード・プロシージャを「監視」するように要求すると、プロシージャの実行時に Open Server からクライアントにノーティフィケーションが送信されます。レジスタード・プロシージャ・ノーティフィケーションは、分散しているアプリケーション内のクライアントの同期を可能にします。「**システム・レジスタード・プロシージャ (system registered procedure)**」も参照してください。

**リモート・プロシージャ・コール**

1. クライアント・アプリケーションが Adaptive Server Enterprise ストアド・プロシージャを実行する 2 つの方法のうちの 1 つ ( もう 1 つの方法では、Transact-SQL の execute 文を使用します )。Client-Library のアプリケーションは、ct_command を呼び出すことによって、リモート・プロシージャ・コール・コマンドを開始します。DB-Library アプリケーションは、dbrpcinit を呼び出すことによって、リモート・プロシージャ・コール・コマンドを開始します。

2. クライアントが、Open Server アプリケーションを使用して利用できる要求のタイプの 1 つ。これに応答して Open Server は、対応するレジスタード・プロシージャを実行するか、または Open Server アプリケーションの RPC イベント・ハンドラを呼び出します。

3. Transact-SQL で、ユーザが接続しているサーバとは異なるサーバ上で実行される「ストアド・プロシージャ」。

**要求機能**

サーバ接続がサポートする要求の種類を判断するために、アプリケーションにより使用されます。

**予約語 (reserved word)**

「**キーワード (keyword)**」を参照してください。

**応答機能**

アプリケーションが処理できないタイプの応答をサーバが送信することを予防するために、アプリケーションにより使用されます。

**結果データ**

サーバがアプリケーションに返すことができるすべてのデータのタイプに対する総称。

**結果セット (result set)**

結果がアプリケーションに返される形式。結果セットには、ただ 1 つのタイプの結果データだけが含まれています。通常ローおよびカーソル・ロー結果セットは、複数のデータ・ローを含むことが可能ですが、その他のタイプの結果セットは 1 つのデータ・ローしか含みません。

**結果変数 (result variable)**

Embedded SQL において、select または fetch 文の結果を受け取る変数。

**セキュリティ・メカニズム (security mechanism)**

接続に対してセキュリティ・メカニズムを提供する外部ソフトウェア。

**select リスト (select list)**

Transact-SQL の select 文が選択するカラムのリスト。

**select リスト id (select-list id)**

Transact-SQL の select 文の結果のカラムに対する数値識別子。select リストの最初のカラムは ID 1、2 番目のカラムは ID 2 となり、以降も同様です。たとえば、次のクエリでは、*title* カラムの select リスト ID は 1 で、*Units Sold* カラムの select リスト ID は 3 です。

```
select title, price, “Units Sold” = total_sales from titles
```

「**select リスト (select list)**」も参照してください。

**シリアライゼーション・プリミティブ (serialization primitive)**
共有リソースへのアクセスの逐次化を可能にする論理オブジェクトとルーチン。「**mutex**」はシリアライゼーション・プリミティブの例です。「**ネイティブ・スレッド (native thread)**」、「**Open Server スレッド (Open Server thread)**」も参照してください。

**サーバ (server)**
クライアント／サーバ・システムにおけるクライアント要求を処理し、結果をクライアントに返す部分。サーバには、「**Adaptive Server Enterprise**」や「**Open Server アプリケーション (Open Server application)**」があります。

**サーバ・ディレクトリ・オブジェクト (server directory object)**
Sybase サーバについて記述するディレクトリ・エントリの論理的な内容を一般化して記述したもの。

**Server-Library**
「**Open Server アプリケーション (Open Server application)**」の記述に使用するルーチンの集まり。

**サーバ・メッセージ (server message)**
サーバがクライアントに送信するエラー・メッセージまたは情報メッセージ。サーバはサーバ・メッセージをクライアントに送信して、クライアントから送信されたコマンドの処理中に発生したエラーまたは異常な状態を説明します。「**サーバ・メッセージ・コールバック (server message callback)**」、「**コールバック・エラー処理 (callback error handling)**」、「**インライン・エラー処理 (inline error handling)**」も参照してください。

**サーバ・メッセージ・コールバック**
Client-Library アプリケーションにおいて、サーバが送信した各「**サーバ・メッセージ (server message)**」を受信するためにインストールされる「**コールバック (callback)**」関数。「**コールバック・エラー処理 (callback error handling)**」も参照してください。

**シグナル駆動型 I/O (signal-driven I/O)**
Client-Library が使用するプラットフォーム固有の方法の 1 つ。これによって非ブロッキング・ネットワークが読み込み、および書き込みを行えます。Client-Library は、内部に自身の内部システム割り込みハンドラをインストールし、非ブロッキング・システム呼び出しを使用してネットワークと対話します。「**スレッド駆動型 (thread-driven I/O)**」と比較してください。

**ソート順 (sort order)**
文字データをソートするときの順序の決定に使用されます。「**照合順 (collating sequence)**」とも呼ばれます。

**SQLCA**
1. Embedded SQL アプリケーションにおいて、Adaptive Server Enterprise とアプリケーション・プログラムの間の通信パスを提供する SQLCA 構造体。各 SQL 文の実行後、プリコンパイラによって生成されたソース・コードが、リターン・コードを SQLCA に格納します。

2. Client-Library アプリケーションにおいて、アプリケーションによって Client-Library およびサーバのエラー・メッセージと情報メッセージの取得に使用される SQLCA 構造体。

**SQLCODE**

1. Embedded SQL アプリケーションにおいて、Adaptive Server Enterprise とアプリケーション・プログラムの間の通信パスを提供する SQLCODE 構造体。各 SQL 文の実行後、プリコンパイラによって生成されたソース・コードが、リターン・コードを SQLCODE に格納します。SQLCODE は、独立して存在することも、SQLCA 構造体の変数になることもできます。

2. Client-Library アプリケーションにおいて、アプリケーションによって Client-Library およびサーバのエラー・メッセージ・コードと情報メッセージ・コードの取得に使用される SQLCODE 構造体。

**スクロール可能カーソル (scrollable cursor)**

現在のカーソル位置を、カーソル結果セットの任意の場所に設定できるようにします。「**カーソル (cursor)**」も参照してください。

**文 (statement)**

Transact-SQL または Embedded SQL におけるキーワードで始まる命令。キーワード名は、基本オペレーションまたは実行するコマンドを表します。

**ステータス変数 (status variable)**

Embedded SQL において、ストアド・プロシージャのリターン・ステータス値を受け取ることによって、プロシージャの成功または失敗を示す変数。

**ストアド・プロシージャ (stored procedure)**

Adaptive Server Enterprise における、名前を付けて保管された SQL 文とオプションのフロー制御文の集まり。Adaptive Server Enterprise が提供するストアド・プロシージャは、「**システム・ストアド・プロシージャ (system stored procedure)**」と呼ばれます。

**同期プリミティブ (synchronization primitive)**

複数のスレッドが実行する従属動作を同期させるための論理オブジェクトおよび関連するルーチン。ネイティブ・スレッド用の同期プリミティブは、ホスト・オペレーティング・システムから提供されます ( たとえば、条件変数およびバリア )。Open Server スレッド用の同期プリミティブは、Server-Library から提供されます ( たとえば、「**メッセージ・キュー (message queue)**」)。「**ネイティブ・スレッド (native thread)**」、「**Open Server スレッド (Open Server thread)**」も参照してください。

**同期 (synchronous)**

メインライン・プログラム・コードの論理によって、明確に指定されたタイミングで予期した通りに発生すること。「**非同期 (asynchronous)**」と比較してください。

**システム管理者 (System Administrator)**
ユーザ・アカウントの作成、パーミッションの割り当て、および新しいデータベースの作成を含むサーバ・システム管理を担当するユーザ。Adaptive Server Enterprise では、システム管理者のログイン名は「sa」です。

**システム記述子 (system descriptor)**
Embedded SQL において、動的 SQL 文で使われる変数の記述を保持するメモリの領域。

**システム・レジスタード・プロシージャ (system registered procedure)**
Open Server が「**レジスタード・プロシージャ (registered procedure)**」のノーティフィケーションの管理とサーバ・ステータスの監視用に提供する内部レジスタード・プロシージャ。「**Open Server**」も参照してください。

**システム・ストアド・プロシージャ (system stored procedure)**
Adaptive Server Enterprise が、システム管理のために提供するストアド・プロシージャ。これらのプロシージャは、システム・テーブルからの情報の取得を簡単にし、データベース管理とシステム・テーブルの更新などを可能にします。

**ターゲット・ファイル (target file)**
Embedded SQL において、プリコンパイラが生成できる 3 つのファイルのうちの 1 つ。ターゲット・ファイルは元の入力ファイルと似ていますが、すべての SQL 文が Client-Library の関数呼び出しに変換されています。

**TDS**
Sybase のクライアントとサーバが通信に使用するアプリケーション・レベルのプロトコル。TDS によって、クライアントとサーバ間で要求と結果を転送できます。「**機能 (capabilities)**」も参照してください。

**テキスト・ポインタ**
大きな text または image 値の代わりとなり、データベース・テーブルに格納されるポインタ。

**テキスト・タイムスタンプ**
各 *text* カラムまたは *image* カラムに関連付けられているタイムスタンプ。アプリケーションの競合によって、一方のアプリケーションで行ったデータベースの変更を、もう一方のアプリケーションが上書きするのを防ぎます。

**スレッド (thread)**
プログラムを介した実行のパス。「**スレッド・スケジューリング (thread scheduling)**」、「**マルチスレッド (multithreaded)**」、「**ネイティブ・スレッド (native thread)**」、「**Open Server スレッド (Open Server thread)**」も参照してください。

**スレッド駆動型 I/O (thread-driven I/O)**

Client-Library が使用するプラットフォーム固有の方法の 1 つ。これによって非ブロッキング・ネットワークが読み込み、および書き込みを行えます。Client-Library は、内部にワーカー・スレッドを作成してネットワークと対話します。内部ワーカー・スレッドは読み込み、または書き込み用にブロックする場合がありますが、ユーザ・アプリケーション・スレッドはブロックしません。「**シグナル駆動型 I/O (signal-driven I/O)**」と比較してください。

**スレッドセーフ (thread-safe)**

マルチスレッド・アプリケーション内で、複数のスレッドによって安全に実行できるルーチンまたはルーチンの集合についてのプロパティ。「**スレッド・シリアライゼーション (thread serialization)**」、「**スレッド同期 (thread synchronization)**」、「**スレッドアンセーフ (thread-unsafe)**」も参照してください。

**スレッド・スケジューリング (thread scheduling)**

複数のスレッドの同時実行を管理する動作。スレッド・スケジューラは明確に定義されたアルゴリズムを使用して、定期的にスレッドを中断してそのステータスを保存し、別のスレッドの実行を再開 ( または開始 ) します。「**ネイティブ・スレッド (native thread)**」、「**Open Server スレッド (Open Server thread)**」も参照してください。

**スレッド・シリアライゼーション (thread serialization)**

異なるスレッドによって関連付けられているクリティカル・セクションが相互に排他的に実行されることを確実にするため、マルチスレッド・コード内でシリアライゼーション・プリミティブを使用すること。逐次化 ( シリアライゼーション ) では、異なるスレッド内の関連付けられているクリティカル・セクションの同時実行からクリティカル・セクションを「保護」するために、シリアライゼーション・プリミティブが使用されます。言い換えると、逐次化は、一度クリティカル・セクションの実行を開始したら、異なるスレッド内の関連付けられているクリティカル・セクションの実行を割り込ませません。逐次化は一般にコードが共有されているリソースを「**スレッドセーフ (thread-safe)**」にするために使用されます。「**クリティカル・セッション (critical section)**」、「**シリアライゼーション・プリミティブ (serialization primitive)**」も参照してください。

**スレッド同期 (thread synchronization)**

複数のスレッドが実行するコードの特有の実行順序を確実にするために、同期プリミティブを使用すること。同期化によって、別々のスレッドが実行する従属の動作を正しい順序で確実に実行できます。「**同期プリミティブ (synchronization primitive)**」、「**ネイティブ・スレッド (native thread)**」、「**Open Server スレッド (Open Server thread)**」も参照してください。

**スレッドアンセーフ (thread-unsafe)**
「**スレッドセーフ (thread-safe)**」でないこと。スレッドアンセーフは、複数のスレッドによる実行を禁止するプログラム・コードのプロパティを記述します。スレッドアンセーフのコードを、複数のスレッドから同時に実行する場合、予知できない動作を行います。

**Transact-SQL**
データベース言語 SQL の拡張バージョン。アプリケーションは、Transact-SQL を使用して、Adaptive Server Enterprise と通信できます。

**トランザクション (transaction)**
1 つの単位として扱われる 1 つ以上のサーバ・コマンド。トランザクション内のコマンドは、1 つのグループとしてコミットされます。したがって、すべてのコマンドがコミットされるか、すべてロールバックされるかのどちらかとなります。

**トランザクション・モード (transaction mode)**
Adaptive Server Enterprise がトランザクションを管理する方法。Adaptive Server Enterprise は、2 つのトランザクション・モードをサポートしています。Transact-SQL モード (「*非連鎖トランザクション*」とも呼ばれる ) と ANSI モード (「*連鎖トランザクション*」とも呼ばれる ) です。

**スループット (throughput)**
一定の時間当たりに行う処理の単位。たとえば、「**バルク・コピー (bulk copy)**」オペレーションのスループットは、1 秒当たりに転送されるローの数で測定します。

**更新可能**
アプリケーションが、ct_cursor 更新コマンドを使用して更新できるカーソルの説明。

**ユーザ名**
「**ログイン名 (login name)**」を参照してください。

# 索引

## 記号


@@textcolid グローバル変数　336
@@textdbid グローバル変数　336
@@textobjid グローバル変数　336
@@textptr グローバル変数　336
@@texttts グローバル変数　336


## A


Adaptive Server
　Open Server との相違点　3
　Open Server との類似点　3
　オプションのリスト　201
　オプションを設定する 2 つの方法　200
　拡張エラー・データ　143
　サーバ・メッセージの処理　136
　制限　326
　接続するサーバの指定　251
　トランザクション・ステータス　145
　プロセスのホスト名のリスト作成　251
　メッセージのリスト　137
Adaptive Server の保護
　trusted ユーザのセキュリティ・ハンドシェイク　584
　セキュリティ・ラベルの処理　49
　チャレンジの処理　49
　チャレンジ／応答セキュリティ・ハンドシェイク　316
ANSI 形式のカーソル・エンドデータ処理　236
ANSI 形式のバインド　236


## B


bcp
　メッセージ　677, 693
binary データ型　341
bit データ型　343
BLK_DONE 完了 ID　603
BLK_INIT 完了 ID　603
BLK_ROWXFER 完了 ID　603, 674
BLK_SENDROW 完了 ID　603, 674
BLK_SENDTEXT 完了 ID　603, 674
BLK_TEXTXFER 完了 ID　603, 674
blktxt.c サンプル・プログラム　147
Bulk-Library
　定義　9


## C


character データ型　343
Client-Library
　Client-Library エラー処理　136
　Embedded SQL との比較　8
　typedefs　341
　グローバル・プロパティ　452
　今後のリリースでの下位互換性　579
　再初期化　558
　サンプル・プログラム　147
　終了　556
　初期化　575
　定義　9
　データ型　339
　バージョン　578
　バージョン・プロパティ　275
　バージョン文字列プロパティ　275
　汎用インタフェース　7
　プロパティ　208
　ユーザ定義データ型　351
Client-Library カーソル・コマンド
　開始　466
　サーバへの送信　476
Client-Library の初期化　575

Client-Library メッセージ 88, 93
 SQLCODE 構造体へのマッピング 108
 重大度の説明 90
 メッセージ番号を復号化するためのマクロ 89
compute 句
 bylist 431
 compute 句の数の取得 616
compute.c サンプル・プログラム 147
CS_ALL_CAPS 定数 81, 398
CS_ALLMSG_TYPE メッセージ・タイプ 508
CS_ALLOC 記述子領域オペレーション 539
CS_ANSI_BINDS プロパティ 213, 361, 440, 455
 詳細な説明 235
CS_APPNAME プロパティ 213, 361, 440
 詳細な説明 236
CS_ASYNC_IO 定数 13
CS_ASYNC_NOTIFS プロパティ 213, 361, 441
 ct_poll 607
 詳細な説明 237
CS_BIGDATETIME データ型 345, 346
CS_BIGINT データ型 347
CS_BIGTIME データ型 345, 346
CS_BINARY データ型 342
CS_BIT データ型 343
CS_BLKDESC 構造体 82
CS_BROWSEDESC 構造体 82, 84
CS_BULK_CONT コマンド・オプション 420
CS_BULK_DATA コマンド・オプション 420
CS_BULK_INIT コマンド・オプション 420
CS_BULK_LOGIN プロパティ 213, 361, 441
 詳細な説明 240
 例 240
CS_BUSY 定数 13
 意味 14
CS_BYLIST_LEN 計算結果情報タイプ 430
CS_CANBENULL ビット 98, 504
CS_CANCEL_ALL キャンセル・タイプ 393
 CS_CANCEL_ATTN との違い 395
 使用しないとき 397
 使用する時期 396
CS_CANCEL_ATTN キャンセル・タイプ 393
 CS_CANCEL_ALL との違い 395
 コマンド構造体の再利用 396
 使用しないとき 397
 使用する時期 396
CS_CANCEL_CURRENT キャンセル・タイプ
 使用する時期 397
CS_CANCEL_CURRENT キャンセル・タイプ 393
CS_CANCELED 戻り値 561, 568
CS_CAP_REQUEST 機能 398
 CS_CON_INBAND 399
 CS_CON_OOB 399
 CS_CSR_ABS 399
 CS_CSR_FIRST 399
 CS_CSR_LAST 399
 CS_CSR_MULTI 399
 CS_CSR_PREV 399
 CS_CSR_REL 399
 CS_DATA_BIGDATETIME 399
 CS_DATA_BIGTIME 399
 CS_DATA_BIN 399
 CS_DATA_BIT 399
 CS_DATA_BITN 399
 CS_DATA_CHAR 399
 CS_DATA_DATE 399
 CS_DATA_DATE4 399
 CS_DATA_DATE8 399
 CS_DATA_DATETIMEN 399
 CS_DATA_DEC 399
 CS_DATA_FLT4 399
 CS_DATA_FLT8 399
 CS_DATA_FLTN 399
 CS_DATA_IMAGE 399
 CS_DATA_INT1 400
 CS_DATA_INT2 400
 CS_DATA_INT4 400
 CS_DATA_INTN 400
 CS_DATA_LBIN 399
 CS_DATA_LCHAR 399
 CS_DATA_MNY4 400
 CS_DATA_MNY8 400
 CS_DATA_MONEYN 400
 CS_DATA_NUM 400
 CS_DATA_SENSITIVITY 400
 CS_DATA_TEXT 400
 CS_DATA_TIME 400
 CS_DATA_VBIN 399
 CS_DATA_VCHAR 399
 CS_OPTION_GET 400
 CS_PROTO_BULK 401
 CS_PROTO_DYNAMIC 401

CS_PROTO_DYNPROC 401
CS_REQ_BCP 401
CS_REQ_CURSOR 401
CS_REQ_DYN 401
CS_REQ_LANG 401
CS_REQ_MSG 401
CS_REQ_MSTMT 401
CS_REQ_NOTIF 401
CS_REQ_PARAM 401
CS_REQ_RPC 401
CS_REQ_URGNOTIF 401
意味 80
CS_CAP_RESPONSE 機能 398
CS_CON_NOINBAND 402
CS_CON_NOOOB 402
CS_DATA_NOBIGDATETIME 402
CS_DATA_NOBIGTIME 402
CS_DATA_NOBIN 402
CS_DATA_NOBIT 402
CS_DATA_NOBOUNDARY 402
CS_DATA_NOCHAR 402
CS_DATA_NODATE 402
CS_DATA_NODATE4 402
CS_DATA_NODATE8 402
CS_DATA_NODATETIMEN 402
CS_DATA_NODEC 402
CS_DATA_NOFLT4 402
CS_DATA_NOFLT8 402
CS_DATA_NOIMAGE 402
CS_DATA_NOINT1 402
CS_DATA_NOINT2 402
CS_DATA_NOINT4 402
CS_DATA_NOINT8 402
CS_DATA_NOINTN 402
CS_DATA_NOLBIN 402
CS_DATA_NOLCHAR 402
CS_DATA_NOMNY4 402
CS_DATA_NOMNY8 402
CS_DATA_NOMONEYN 402
CS_DATA_NONUM 402
CS_DATA_NOTEXT 403
CS_DATA_NOTIME 403
CS_DATA_NOVBIN 402
CS_DATA_NOVCHAR 402
CS_RES_NOEED 403
CS_RES_NOMSG 403
CS_RES_NOPARAM 403
CS_RES_NOSTRIPBLANKS 403
CS_RES_NOTDSDEBUG 403
CS_RES_NOXNLMETADATA 73, 403
意味 80
CS_CAP_TYPE 構造体 82
ビット操作 175
CS_CHALLENGE_CB コールバック・タイプ 389
CS_CHAR データ型 343
CS_CHARSETCNV プロパティ 213, 441
詳細な説明 240
CS_CLEAR action の値 412, 453
CS_CLEAR オペレーション 510
CS_CLEAR_FLAG デバッグ・
オペレーション 500
CS_CLIENTMSG 構造体 82, 85
CS_CLIENTMSG_CB コールバック・タイプ 389
CS_CLIENTMSG_TYPE 構造体タイプ 508
CS_CLR_CAPMASK マクロ 81, 406
CS_CMD_DONE 結果タイプ 619
CS_CMD_FAIL 結果タイプ 619
CS_CMD_NUMBER 情報タイプ
役立つ場合 615
CS_CMD_SUCCEED 結果タイプ 619
CS_CMD_SUPPRESS_FMT 417
CS_COLUMN_DATA コマンド・オプション 420
CS_COMMAND 構造体 82
削除 411
定義 410
割当 409
割り付け解除 411
CS_COMMBLOCK プロパティ 214, 441
詳細な説明 240
CS_COMP_BYLIST 計算結果情報タイプ 430
CS_COMP_COLID 計算結果情報タイプ 431
CS_COMP_ID 計算結果情報タイプ 431
CS_COMP_OP 計算結果情報タイプ 431
CS_COMPLETION_CB コールバック・
タイプ 389
CS_COMPUTE_RESULT
結果タイプ 564
CS_COMPUTE_RESULT
結果タイプ 431, 619
CS_COMPUTEFMT_RESULT 結果タイプ 620

CS_COMPUTEFMT_RESULT フォーマット
結果セット 247
CS_CON_INBAND 機能 399
CS_CON_NOINBAND 機能 402
CS_CON_NOOOB 機能 402
CS_CON_OOB 機能 399
CS_CON_STATUS プロパティ 214, 441
詳細な説明 240
CS_CONFIG_BY_SERVERNAME
プロパティ 442
CS_CONFIG_BY_SERVERNAME
プロパティ 215
CS_CONFIG_FILE プロパティ 215, 442
CS_CONNECTED_ADDR プロパティ 214, 441
CS_CONNECTION 構造体 83
削除 434
割当 432
割り付け解除 434
CS_CONSTAT_CONNECTED 記号値 241
CS_CONSTAT_DEAD 記号値 241
CS_CONTEXT 構造体 83
プロパティ 453
CS_CSR_ABS 機能 399
CS_CSR_FIRST 機能 399
CS_CSR_LAST 機能 399
CS_CSR_MULTI 機能 399
CS_CSR_PREV 機能 399
CS_CSR_REL 機能 399
CS_CUR_ID プロパティ 215, 417
詳細な説明 241
CS_CUR_NAME プロパティ 216, 417
詳細な説明 242
CS_CUR_ROWCOUNT プロパティ 216, 417
詳細な説明 242
CS_CUR_STATUS プロパティ 216, 417
詳細な説明 243
CS_CURSOR_CLOSE カーソル・コマンド・
タイプ 476
CS_CURSOR_DEALLOC カーソル・コマンド・
タイプ 476
CS_CURSOR_DECLARE カーソル・コマンド・タイ
プ 474
CS_CURSOR_DECLARE
動的 SQL オペレーション 531
CS_CURSOR_DELETE カーソル・コマンド・
タイプ 476
CS_CURSOR_OPEN カーソル・コマンド・
タイプ 476
CS_CURSOR_OPTION カーソル・コマンド・
タイプ 475
CS_CURSOR_RESULT 結果タイプ 564, 619
CS_CURSOR_ROWS カーソル・コマンド・
タイプ 475
CS_CURSOR_UPDATE カーソル・コマンド・
タイプ 476
CS_CURSTAT_CLOSED 記号値 244
CS_CURSTAT_DECLARED 記号値 244
CS_CURSTAT_NONE 記号値 243
CS_CURSTAT_OPEN 記号値 244
CS_CURSTAT_RDONLY 記号値 244
CS_CURSTAT_ROWCOUNT 記号値 244
CS_CURSTAT_UPDATABLE 記号値 244
CS_DATA_BIGDATETIME 機能 399
CS_DATA_BIGTIME 機能 399
CS_DATA_BIN 機能 399
CS_DATA_BIT 機能 399
CS_DATA_BITN 機能 399
CS_DATA_CHAR 機能 399
CS_DATA_DATE 機能 399
CS_DATA_DATE4 機能 399
CS_DATA_DATE8 機能 399
CS_DATA_DATETIMEN 機能 399
CS_DATA_DEC 機能 399
CS_DATA_FLT4 機能 399
CS_DATA_FLT8 機能 399
CS_DATA_FLTN 機能 399
CS_DATA_IMAGE 機能 399
CS_DATA_INT1 機能 400
CS_DATA_INT2 機能 400
CS_DATA_INT4 機能 400
CS_DATA_INTN 機能 400
CS_DATA_LBIN 機能 399
CS_DATA_LCHAR 機能 399
CS_DATA_MNY4 機能 400
CS_DATA_MNY8 機能 400
CS_DATA_MONEYN 機能 400
CS_DATA_NOBIGDATETIME 機能 402
CS_DATA_NOBIGTIME 機能 402
CS_DATA_NOBIN 機能 402
CS_DATA_NOBIT 機能 402

CS_DATA_NOBOUNDARY 機能　402
CS_DATA_NOCHAR 機能　402
CS_DATA_NODATE 機能　402
CS_DATA_NODATE4 機能　402
CS_DATA_NODATE8 機能　402
CS_DATA_NODATETIMEN 機能　402
CS_DATA_NODEC 機能　402
CS_DATA_NOFLT4 機能　402
CS_DATA_NOFLT8 機能　402
CS_DATA_NOIMAGE 機能　402
CS_DATA_NOINT1 機能　402
CS_DATA_NOINT2 機能　402
CS_DATA_NOINT4 機能　402
CS_DATA_NOINT8 機能　402
CS_DATA_NOINTN 機能　402
CS_DATA_NOLBIN 機能　402
CS_DATA_NOLCHAR 機能　402
CS_DATA_NOMNY4 機能　402
CS_DATA_NOMNY8 機能　402
CS_DATA_NOMONEYN 機能　402
CS_DATA_NONUM 機能　402
CS_DATA_NOTEXT 機能　403
CS_DATA_NOTIME 機能　403
CS_DATA_NOVBIN 機能　402
CS_DATA_NOVCHAR 機能　402
CS_DATA_NUM 機能　400
CS_DATA_SENSITIVITY 機能　400
CS_DATA_TEXT 機能　400
CS_DATA_TIME 機能　400
CS_DATA_VBIN 機能　399
CS_DATA_VCHAR 機能　399
CS_DATAFMT 構造体　82, 93
　ct_bind　372
　ct_describe　502
CS_DATE データ型　345
CS_DATETIME データ型　345
CS_DATETIME4 データ型　345
CS_DBG_ALL デバッグ・フラグ　498
CS_DBG_API_LOGCALL デバッグ・フラグ　498
CS_DBG_API_STATES デバッグ・フラグ　498
CS_DBG_ASYNC デバッグ・フラグ　498
CS_DBG_DIAG デバッグ・フラグ　498
CS_DBG_ERROR デバッグ・フラグ　498
CS_DBG_MEM デバッグ・フラグ　498
CS_DBG_NETWORK デバッグ・フラグ　498
CS_DBG_PROTOCOL デバッグ・フラグ　498
CS_DBG_PROTOCOL_STATES デバッグ・フラグ　499
CS_DBG_SSL デバッグ・フラグ　499
CS_DEALLOC 記述子領域オペレーション　539
CS_DEALLOC 動的 SQL オペレーション　531
CS_DECIMAL データ型　348
CS_DEF_PREC 定数　96, 97
CS_DESCRIBE_INPUT
　動的 SQL オペレーション　531
CS_DESCRIBE_OUTPUT
　動的 SQL オペレーション　531
CS_DESCRIBE_RESULT 結果タイプ　620
CS_DIAG_TIMEOUT プロパティ　216, 361, 442
　インライン・メッセージ処理　140
　詳細な説明　245
CS_DISABLE_POLL プロパティ
　ct_poll　607
　ct_wakeup　675
　詳細な説明　245
　レイヤ構成の非同期アプリケーション　20
CS_DISABLE_POLL
　プロパティ　217, 361, 443, 455
CS_DS_COPY プロパティ　217, 361
CS_DS_DITBASE プロパティ　217, 361
CS_DS_EXPANDALIAS プロパティ　217
CS_DS_FAILOVER プロパティ　218, 361
CS_DS_LOOKUP_CB コールバック・タイプ　389
CS_DS_PASSWORD プロパティ　218, 361
CS_DS_PRINCIPAL プロパティ　218, 361
CS_DS_PROVIDER プロパティ　219, 361
CS_DS_SEARCH プロパティ　219
CS_DS_SIZELIMIT プロパティ　219
CS_DS_TIMELIMIT プロパティ　220
CS_EED_CMD オペレーション　511
CS_EED_CMD プロパティ　220, 444
　詳細な説明　246
CS_ENCRYPT_CB コールバック・タイプ　389
CS_END
　コマンド・オプション　420
CS_END_DATA 戻り値　560, 568, 633
CS_END_ITEM 戻り値　568
CS_ENDPOINT プロパティ　220, 445
CS_EXEC_IMMEDIATE 動的 SQL
　オペレーション　531

CS_EXECUTE 動的 SQL オペレーション　531
CS_EXPOSE_FMTS プロパティ　220, 361, 445, 456
　詳細な説明　246
　フォーマット結果を受信するため
　　に必ず有効にする　628
CS_EXTENDED_ENCRYPT_CB プロパティ　445
CS_EXTERNAL_CONFIG プロパティ　221, 445, 456
CS_EXTRA_INF プロパティ　361
　インライン・メッセージ処理　140
　詳細な説明　247
CS_EXTRA_INF
　　プロパティ　221, 361, 362, 363, 445, 456
CS_FAIL 定数　34
CS_FIRST_CHUNK 記号　88, 105
　連続したメッセージ　142
CS_FLOAT データ型　347
CS_FMT_NULLTERM 記号　96
CS_FMT_PADBLANK 記号　96
CS_FMT_PADNULL 記号　96
CS_FMT_UNUSED 記号　96
CS_FORCE_CLOSE オプション　407
　使用する時期　409
CS_FORCE_EXIT オプション　557
CS_GET action の値　412, 453
CS_GET オペレーション　510
CS_GETATTR 記述子領域オペレーション　539
CS_GETCNT 記述子領域オペレーション　539
CS_HASEED 記号　105
CS_HAVE_BINDS プロパティ　221, 248, 417
CS_HAVE_CMD プロパティ　221, 248, 417
CS_HAVE_CUROPEN プロパティ　222, 418
CS_HIDDEN ビット　23, 98, 504
CS_HIDDEN_KEYS プロパティ
　ct_keydata　582
　詳細な説明　250
　設定できない場合　250
　ブラウズ・モード　23
CS_HIDDEN_KEYS
　　プロパティ　222, 361, 418, 445, 456
CS_HOSTNAME プロパティ　222, 361, 445
　詳細な説明　250
CS_IDENTITY ビット　98, 504
CS_IFILE プロパティ　222, 361, 456
　詳細な説明　251
CS_IMAGE データ型　350
CS_INIT オペレーション　509
CS_INPUTVALUE ビット　98
CS_INT データ型　347
CS_INTERRUPT 戻り値　604
CS_IODESC 構造体　82, 99, 495
　ct_send_data　652
CS_ISBROWSE 情報タイプ　386
CS_KEY ビット　98, 504, 505
CS_LANG_CMD コマンド・タイプ　420, 422
CS_LAST_CHUNK 記号　88, 105
　連続したメッセージ　142
CS_LAYER マクロ　89
CS_LOC_PROP プロパティ　222, 446
　詳細な説明　252
CS_LOCALE 構造体　83
　使用する時期　161
CS_LOGIN_STATUS プロパティ　222, 446
　詳細な説明　252
CS_LOGIN_TIMEOUT プロパティ　223, 361, 456
　詳細な説明　252
CS_LOGINFO 構造体　83
　再利用できない　658
CS_LONGBINARY データ型　342
CS_LONGCHAR データ型　343
CS_LOOP_DELAY プロパティ　361
CS_MAX_CONNECT プロパティ
　詳細な説明　253
　デフォルト値　254
CS_MAX_CONNECT
　　プロパティ　223, 361, 456, 464
CS_MAX_MSG 定数　141
CS_MAX_PREC 定数　97
CS_MAX_SCALE 定数　96
CS_MEM_ERROR の戻り値　577
CS_MEM_POOL プロパティ　223, 456
　詳細な説明　254
CS_MIN_PREC 定数　97
CS_MIN_SCALE 定数　96
CS_MONEY データ型　349
CS_MONEY4 データ型　349
CS_MORE コマンド・オプション　420
CS_MSG_CMD コマンド・タイプ　422
CS_MSG_GETLABELS 定数　51
CS_MSG_LABELS 定数　51
CS_MSG_RESULT 結果タイプ　620

CS_MSGLIMIT オペレーション 509
CS_NETIO プロパティ 223, 361, 446, 457, 464
詳細な説明 255
制限 257
CS_NO_LIMIT タイムアウト値 253
CS_NO_LIMIT メッセージ制限値 514
CS_NO_RECOMPILE コマンド・オプション 420
CS_NO_TRUNCATE プロパティ 224, 362, 457
詳細な説明 257
連続したメッセージ 141
CS_NOAPICHK プロパティ 224, 257, 361, 457
CS_NOCHARSETCNV_REQD プロパティ 224, 259, 446
CS_NOINTERRUPT プロパティ 224, 362, 457
詳細な説明 259
CS_NOSCROLL_INSENSITIVE 記号値 244
CS_NOTIF_CB コールバック・タイプ 389
CS_NOTIF_CMD プロパティ 225, 446
CS_NULLTERM 定数 95
CS_NUMBER マクロ 89
CS_NUMERIC データ型 348
CS_OID 構造体 101
CS_OP_AVG 集合演算子タイプ 432
CS_OP_COUNT 集合演算子タイプ 432
CS_OP_MAX 集合演算子タイプ 432
CS_OP_MIN 集合演算子タイプ 432
CS_OP_SUM 集合演算子タイプ 432
CS_OPT_ANSINULL オプション 201, 364, 587
CS_OPT_ARITHABORT オプション 201, 364, 588
CS_OPT_ARITHIGNORE オプション 202, 364, 588
CS_OPT_AUTHOFF オプション 202, 364, 588
CS_OPT_AUTHON オプション 202, 364, 588
CS_OPT_CHAINXACTS オプション 202, 364, 588
CS_OPT_CURCLOSEONXACT オプション 202, 364, 588
CS_OPT_CURREAD オプション 364
CS_OPT_CURWRITE オプション 364
CS_OPT_DATEFIRST オプション 202, 364, 588
CS_OPT_DATEFORMAT オプション 202, 364, 588
CS_OPT_FIPSFLAG オプション 203, 364, 588
CS_OPT_FORCEPLAN オプション 203, 364, 588
CS_OPT_FORMATONLY オプション 203, 364, 588
CS_OPT_GETDATA オプション 364, 588
CS_OPT_HIDE_VCC オプション 203, 588
CS_OPT_IDENTITYOFF オプション 203, 364, 588
CS_OPT_IDENTITYON オプション 203, 588
CS_OPT_IDENTITYON オプション 203, 364, 588, 589
CS_OPT_ISOLATION オプション 204, 364, 589
CS_OPT_LOBLOCATOR 589
CS_OPT_NOCOUNT オプション 204, 364, 589
CS_OPT_NOEXEC オプション 204, 364, 589
CS_OPT_PARSEONLY オプション 204, 364, 589
CS_OPT_PREFETCHSIZE 589
CS_OPT_QUOTED_IDENT オプション 204, 364, 589
CS_OPT_RESTREES オプション 204, 364, 589
CS_OPT_ROWCOUNT オプション 204, 364, 589
CS_OPT_SHOW_FI オプション 589
CS_OPT_SHOW_VI オプション 205
CS_OPT_SHOWPLAN オプション 205, 365, 589
CS_OPT_STATS_IO オプション 205, 365, 589
CS_OPT_STATS_TIME オプション 205, 365, 589
CS_OPT_STR_RTRUNC おぷしょん 206, 589
CS_OPT_TEXTSIZE オプション 206, 365, 589
CS_OPT_TRUNCIGNORE オプション 207, 365, 589
CS_OPTION_GET 機能 400
CS_ORIGIN マクロ 89
CS_PACKAGE_CMD コマンド・タイプ 422
CS_PACKETSIZE プロパティ 225, 362, 446
CS_PARAM_RESULT 結果タイプ 24, 564, 619
CS_PARENT_HANDLE プロパティ 225, 418, 446
詳細な説明 259
CS_PARTIAL_TEXT プロパティ 226, 447, 457
CS_PASSTHRU_EOM 戻り値 608, 657
CS_PASSTHRU_MORE 戻り値 608, 657
CS_PASSWORD プロパティ 226, 227, 362, 447
CS_PENDING 戻り値 13, 37, 561, 568
CS_PREPARE 動的 SQL オペレーション 531
CS_PROP_APPLICATION_SPID プロパティ 226, 447
CS_PROP_EXTENDEDFAILOVER プロパティ 226
CS_PROP_MIGRATABLE プロパティ 226, 447, 457

CS_PROTO_BULK 機能 401
CS_PROTO_DYNAMIC 機能 401
CS_PROTO_DYNPROC 機能 401
CS_PUBLIC マクロ
  コールバック 33
  説明 176
CS_QUIET 戻り値 604
CS_REAL データ型 347
CS_RECOMPILE コマンド・オプション 420
CS_REQ_BCP 機能 401
CS_REQ_CURSOR 機能 401
CS_REQ_DYN 機能 401
CS_REQ_LANG 機能 401
CS_REQ_MSG 機能 401
CS_REQ_MSTMT 機能 401
CS_REQ_NOTIF 機能 401
CS_REQ_PARAM 機能 401
CS_REQ_RPC 機能 401
CS_REQ_URGNOTIF 機能 401
CS_RES_NOEED 機能 403
CS_RES_NOMSG 機能 403
CS_RES_NOPARAM 機能 403
CS_RES_NOSTRIPBLANKS 機能 403
CS_RES_NOTDSDEBUG 機能 403
CS_RES_SUPPRESS_FMT 403
CS_RETRY_COUNT プロパティ 227, 228, 362
CS_RETURN ビット 98, 504
CS_ROW_FAIL 戻り値 560
CS_ROW_RESULT 結果タイプ 564, 619
CS_ROWFMT_RESULT 結果タイプ 247, 620
CS_RPC_CMD コマンド・タイプ 420, 422
CS_SCROLL_CURSOR 記号値 244
CS_SCROLL_INSENSITIVE 記号値 244
CS_SCROLL_SEMISENSITIVE 記号値 244
CS_SEC_APPDEFINED プロパティ 228, 362, 448
CS_SEC_CHALLENGE プロパティ 229, 362, 448
CS_SEC_CHANBIND プロパティ 362
CS_SEC_CONFIDENTIALITY プロパティ 362
CS_SEC_CREDTIMEOUT プロパティ 362
CS_SEC_DATAORIGIN プロパティ 362
CS_SEC_DELEGATION プロパティ 362
CS_SEC_DETECTREPLAY プロパティ 362
CS_SEC_DETECTSEQ プロパティ 362
CS_SEC_ENCRYPTION プロパティ 231, 362, 449
CS_SEC_EXTENDED_ENCRYPTION
    プロパティ 449
CS_SEC_INTEGRITY プロパティ 362
CS_SEC_KEYTAB プロパティ 362
CS_SEC_MECHANISM プロパティ 362
CS_SEC_MUTUALAUTH プロパティ 362
CS_SEC_NEGOTIATE プロパティ 232, 450
  trusted ユーザのセキュリティ・
    ハンドシェイク 585
CS_SEC_NETWORKAUTH プロパティ 362
CS_SEC_NON_ENCRYPTION_RETRY
    プロパティ 450
CS_SEC_SERVERPRINCIPAL プロパティ 363
CS_SEC_SESSTIMEOUT プロパティ 363
CS_SECSESSION_CB コールバック・タイプ 389
CS_SEND_BULK_CMD コマンド・
    タイプ 420, 422
CS_SEND_DATA_CMD コマンド・
    タイプ 420, 422
CS_SEND_DATA_NOCMD コマンド・
    タイプ 422
CS_SENDDATA_NOCMD プロパティ 233, 451
CS_SERVERADDR プロパティ 233, 451
CS_SERVERMSG 構造体 82, 103
CS_SERVERMSG_CB コールバック・タイプ 389
CS_SERVERMSG_TYPE 構造体タイプ 508
CS_SERVERNAME プロパティ 232, 233, 451
  詳細な説明 264
CS_SET action の値 412, 453
CS_SET_CAPMASK マクロ 81, 406
CS_SET_DBG_FILE デバッグ・
    オペレーション 500
CS_SET_FLAG デバッグ・オペレーション 500
CS_SET_PROTOCOL_FILE デバッグ・オペレー
    ション 500
CS_SETATTR 記述子領域オペレーション 539
CS_SETCNT 記述子領域オペレーション 539
CS_SEVERITY マクロ 89
CS_SIGNAL_CB コールバック・タイプ 64, 389
CS_SIZEOF マクロ 175
CS_SMALLINT データ型 347
CS_SRC_VALUE 定数 96, 97
CS_SSLVALIDATE_CB コールバック・
    タイプ 389
CS_STATUS オペレーション 510
CS_STATUS_RESULT 結果タイプ 564, 619
CS_STICKY_BINDS プロパティ 233, 418
  ct_results 628
  詳細な説明 261

CS_SUPPORTED action の値　436, 453
CS_SV_API_FAIL メッセージの重大度　87, 91
CS_SV_COMM_FAIL メッセージの
　　重大度　87, 91
CS_SV_CONFIG_FAIL メッセージの
　　重大度　87, 90
CS_SV_FATAL メッセージの重大度　87, 92
CS_SV_INFORM メッセージの重大度　87, 90
CS_SV_INTERNAL_FAIL メッセージの
　　重大度　87, 91
CS_SV_RESOURCE_FAIL メッセージの
　　重大度　87, 91
CS_SV_RETRY_FAIL メッセージの
　　重大度　87, 91
CS_TABNAME 情報タイプ　386
CS_TABNUM 情報タイプ　386
CS_TCP_RCVBUF プロパティ　265, 460
CS_TCP_SND プロパティ　460
CS_TCP_SNDBUF プロパティ　265
CS_TDS_VERSION プロパティ　234, 363, 452
　機能　405
　詳細な説明　264
CS_TEXT データ型　350
CS_TEXTLIMIT プロパティ　234, 363, 452, 460
　詳細な説明　267
　デフォルト値　267
CS_TIME データ型　345
CS_TIMED_OUT 戻り値　604
CS_TIMEOUT プロパティ　234, 363, 460
　詳細な説明　267
CS_TIMESTAMP ビット　23, 98, 504
CS_TINYINT データ型　347
CS_TRAN_COMPLETED トランザクション・ス
　　テータス　146
CS_TRAN_FAIL トランザクション・
　　ステータス　146
CS_TRAN_IN_PROGRESS トランザクション・ス
　　テータス　146
CS_TRAN_STMT_FAIL トランザクション・ス
　　テータス　146
CS_TRAN_UNDEFINED トランザクション・ス
　　テータス　146
CS_TRANSACTION_NAME プロパティ
　詳細な説明　270
CS_TRANSACTION_NAME
　　プロパティ　234, 452
CS_TST_CAPMASK マクロ　81, 406
CS_UBIGINT データ型　347
CS_UINT データ型　347
CS_UNICHAR データ型　343, 344
CS_UNITEXT データ型　350
CS_UNUSED
　コマンド・オプション　420
CS_UNUSED オプション　557
CS_UNUSED コマンド・オプション　420
CS_UPDATABLE ビット　98, 504
CS_UPDATECOL ビット　98
CS_USE_DESC 記述子領域オペレーション　539
CS_USER_ALLOC プロパティ　234, 460
　詳細な説明　271
CS_USER_FREE プロパティ　234, 461
　詳細な説明　272
CS_USER_MAX_MSGID 定数　51
CS_USER_MSGID 定数　51
CS_USERDATA プロパティ　235, 418, 452
　コールバックと組み合わせて使用　392
　詳細な説明　273
CS_USERNAME プロパティ　235, 363, 452
　詳細な説明　274
CS_VARBINARY データ型　342
CS_VARCHAR データ型　343
CS_VER_STRING プロパティ　235, 275, 461
　詳細な説明　275
CS_VERSION プロパティ　235, 275, 461
　値の決定　579
　詳細な説明　275
　有効な値　275
CS_VERSION_100 バージョン　576, 631
CS_VERSION_110 バージョン　576
CS_VERSION_KEY ビット　98, 504
csconfig.h ヘッダ・ファイル　153
CS-Library
　定義　9
cspublic.h ヘッダ・ファイル　153
csr_disp.c サンプル・プログラム　147
csr_disp_scrollcurs.c サンプル・プログラム　147,
　　148
csr_disp_scrollcurs2.c サンプル・プログラム　148

cstypes.h ヘッダ・ファイル 89, 153, 175
ct_bind 371, 384
CS_DATAFMT 構造体 372
CS_HAVE_BINDS コマンド・プロパティ 248
CS_STICKY_BINDS プロパティの影響 261
コード例 383
障害の一般的な原因 376
バッチ処理 289
ct_br_column 384, 385
呼び出す場合 23
ct_br_table 385, 387
呼び出す場合 23
ct_callback 387, 392
レイヤ構成のアプリケーション 21
ct_cancel 392, 397
コード例 397
非同期オペレーションが未処理状態のときに呼び出し可能 14
非同期動作 14
CT_CANCEL 完了 ID 603, 674
ct_capability 327, 397, 406
ct_close 406, 409
コード例 409
障害の一般的な原因 407
非同期動作 14
CT_CLOSE 完了 ID 603, 674
ct_cmd_alloc 409, 410
コード例 410
失敗の原因 410
ct_cmd_drop 411, 412, 418
コード例 412
失敗の原因 411
ct_cmd_props 412, 418
コード例 417
使用する時期 416
非同期オペレーションが未処理状態のときに呼び出し可能 14
ct_command 110, 428
コード例 427
ct_compute_info 428, 432
コード例 432
呼び出す場合 431
ct_con_alloc 432, 434
コード例 434
障害の一般的な原因 432
使用する時期 410
呼び出す前の処理 433
ct_con_drop 434, 436
コード例 436
障害の一般的な原因 434
呼び出す前の処理 435
「dead」接続 435
ct_con_props 436, 452
コード例 440
非同期オペレーションが未処理状態のときに呼び出し可能 14
ct_config 452, 461
コード例 455
ct_connect 461, 466
CS_MAX_CONNECT プロパティ 464
CS_NETIO プロパティ 464
コード例 466
失敗の原因 462
ディレクトリ・サービス 118
非同期動作 14
呼び出す前の処理 464
CT_CONNECT 完了 ID 603, 674
ct_cursor 110, 466
CS_HAVE_CUROPEN プロパティ 249
コード例 492
ct_data_info 492, 496
ct_debug 496, 501
コード例 501
デフォルトの動作 500
ct_describe 501, 507
CS_DATAFMT 構造体 502
コード例 507
使用する時期 507
呼び出せない場合 502, 507
ct_diag 507, 515
拡張エラー・データ 514
コンテキスト・レベルでは使用不可 140, 511
失敗の原因 509
接続固有のインライン・メッセージ処理 511
メッセージ・コールバックのインストール解除 34

連続したメッセージ　143, 515
ct_ds_dropobj　515, 516
ct_ds_lookup　516, 523
CT_DS_LOOKUP 完了 ID　603
ct_ds_objinfo　523, 530
ct_dynamic　110, 530, 538
ct_dyndesc　538, 548
ct_dynsqlda　548, 556
ct_exit　556, 558
コード例　558
失敗の原因　558
使用する時期　558
ct_fetch　559, 566
コード例　566
失敗の原因　561
非同期動作　14
非同期プログラミング　563
CT_FETCH 完了 ID　603, 674
ct_get_data　566, 571
ct_bind の代替　379
text 値または image 値のフェッチ　328
使用する時期　571
データ廃棄　571
非同期動作　14
変換が実行されない　570
CT_GET_DATA 完了 ID　603, 674
ct_getformat　572, 573
使用する時期　573
ct_getloginfo　573, 574
使用しないとき　574
使用する時期　573
ct_init　575, 579
コード例　579
複数回呼び出す　579
呼び出す場合　578
呼び出す前の処理　578
ct_keydata　580, 582
主な用途　581
サーバに対する現在のローの識別　582
呼び出す状況　582
ct_labels　583, 585
CT_NOTIFICATION 完了 ID　603
ct_options　585, 590
非同期動作　14
CT_OPTIONS 完了 ID　603, 674
ct_param　111, 590, 600
ct_setparam との違い　596
コード例　600
使用する時期　596
ct_poll　601, 607
CS_ASYNC_NOTIFS プロパティ　607
CS_DISABLE_POLL プロパティ　607
コールバック　607
使用する時期　606
非同期オペレーションが未処理状態のときに呼び出し可能　14
非同期の完了をチェックするための使用　15
ルーチン完了の報告を抑制　20
レイヤ構成のアプリケーション　20
ct_recvpassthru　607, 608
非同期動作　14
CT_RECVPASSTHRU 完了 ID　603, 674
ct_remote_pwd　609, 611
使用しないとき　611
複数のパスワードの定義　610
ct_res_info　612, 618
使用する時期　614
ct_res_info での CS_BROWSE_INFO 情報タイプ　613
ct_res_info での CS_CMD_NUMBER 情報タイプ　614
ct_res_info での CS_MSGTYPE 情報タイプ　614
ct_res_info での CS_NUM_COMPUTES 情報タイプ　614
ct_res_info での CS_NUMDATA 情報タイプ　614
ct_res_info での CS_NUMORDERCOLS 情報タイプ　614
ct_res_info での CS_ORDERBY_COLS 情報タイプ　614
ct_res_info での CS_ROW_COUNT 情報タイプ　614
ct_res_info での CS_TRANS_STATE 情報タイプ　614
ct_results　618, 629
CS_STICKY_BINDS プロパティ　628
結果をループで処理　624
コード例　629
ストアド・プロシージャ　628

非同期動作　14
CT_RESULTS 完了 ID　603, 674
ct_scroll_fetch　629
ct_send　111, 639, 644
CS_HAVE_CMD プロパティ　248
コード例　644
サーバからの応答を待たない　643
非同期動作　14
CT_SEND 完了 ID　603, 674
ct_send_data　644, 654
使用する時期　651
非同期動作　14
部分更新の送信　334
CT_SEND_DATA 完了 ID　603, 674
ct_sendpassthru　656, 657
非同期動作　14
CT_SENDPASSTHRU 完了 ID　603, 674
ct_setloginfo　658, 659
CS_LOGINFO 構造体の解放　658
使用しないとき　658
使用する時期　658
ct_setparam　111, 660, 673
ct_param との違い　668
CT_USER_FUNC 完了 ID　603, 674
ct_wakeup　673, 675
CS_DISABLE_POLL プロパティ　675
レイヤ構成の非同期アプリケーション　20
ctpublic.h ヘッダ・ファイル　153


## D


datetime 型
CS_DATE　345
CS_TIME　345
datetime データ型　344
DB-Library
定義　4
「dead」状態の接続　240
「dead」状態の接続
定義　241
defncopy
メッセージ　677, 693


## E


Embedded SQL
Client-Library との比較　8
EX_ 記号とデータ型
サンプル・プログラムの検索　xvi
ex_ ルーチン
サンプル・プログラムの検索　xvi
ex_alib.c サンプル・プログラム　148
ex_amain.c サンプル・プログラム　148
example.h ヘッダ・ファイル　148
exasync.h ヘッダ・ファイル　148
exconfig
サンプル・プログラム　148
exutils.c サンプル・プログラム　148
exutils.h ヘッダ・ファイル　148


## F


firstapp
サンプル・プログラム　148


## G


getsend.c サンプル・プログラム　148


## I


I/O 記述子構造体　99
ct_data_info　495
ct_send_data　495
使用方法　495
定義と取得　492
i18n.c サンプル・プログラム　148
interfaces ファイル
ct_connect　251, 462
interfaces ファイル・プロパティ　251
定義　157
ディレクトリ・サービス　115
デフォルト・ファイル名　157

優先度　122
isql
メッセージ　677, 693


## L


LDAP
ldapurl の定義　122
libtcl*.cfg ファイル　122
接続タイプ　116
定義　116
ディレクトリ・スキーマ　122
ldapurl
キーワード　122
例　122
libtcl*.cfg
上書き　122
libtcl*.cfg ファイル　122
優先度　122
libtcl.cfg ファイル
セキュリティ・ドライバ　293
ディレクトリ・ドライバ　132


## M


malloc
割り込みレベルでは安全ではない　19
money でーたがた　349
multithrd
サンプル・プログラム　148


## N


Net-Library　4


## O


objectid.dat ファイル
セキュリティ・ドライバ　294
Open Client
アクセスするサーバ　326
アプリケーション開発者の必要事項　326
製品の説明　3
デバッグ　113
汎用的なプログラミング・インタフェース　326
ライブラリ呼び出しの図　6
Open Server
Adaptive Server との相違点　3
Adaptive Server との類似点　3
制限　326
説明　4
ネットワーク・サービス　5
プログラミング・インタフェース　5
ライブラリ呼び出しの図　6


## P


PROTOTYPE マクロ
使用　176
説明　176


## R


RPC コマンド
起動　110
rpc.c サンプル・プログラム　148


## S


CS_OPT_ANSIPERM オプション　201, 364, 587
S_UPDATECOL ビット　504
SDC
「共有ディスク・クラスタ」参照　311
secct_dec
サンプル・プログラム　149
secct_krb
サンプル・プログラム　149
select リスト・カラム ID
計算カラムの取得　431
select...for browse コマンド　23
sizeof 演算子　175

SQLCA 構造体　82, 327
　CS_EXTRA_INF プロパティ　141
　連続したメッセージをサポートしない　142
sqlca.h ヘッダ・ファイル　153
SQLCA_TYPE 構造体タイプ　508
SQLCODE 構造体　82, 107
　Client-Library メッセージのマッピング　108
　CS_EXTRA_INF プロパティ　141
　サーバ・メッセージのマッピング　108
　連続したメッセージをサポートしない　142
SQLCODE_TYPE 構造体タイプ　508
SQLDA 構造体　549
　定義　549
　割り付け　550
SQLDA_DECL マーカ　550
SQLSTATE 構造体　82, 108
　CS_EXTRA_INF プロパティ　141
　連続したメッセージをサポートしない　142
SQLSTATE_TYPE 構造体タイプ　508
SSL 検証コールバック　65
　インストール　389
　定義　66
　トリガされる方法　30
　呼び出されるタイミング　30
　例　67
SYB_SQLDA_SIZE マクロ、定義　550

# T

TDS (Tabular Data Stream)
　EOM (End of Message) とマーク付けされたパケット　608
　TDS バージョン・プロパティ　264
　TDS パケットの受信　607
　TDS フォーマットのネゴシエート　574, 659
　機能の決定　405
　サーバへの TDS パケットの送信　656
　接続の TDS バージョン・レベルの変更　80
　接続のデフォルト・バージョン・レベル　80
　パケット・サイズ・プロパティ　259
　パススルー・オペレーション　574, 607, 659
　プラットフォームによって異なるデフォルト・パケット・サイズ　657
　ログイン応答情報の転送　573, 658
TDS バージョン・プロパティ
　CS_TDS_40 の値　266, 363
　CS_TDS_42 の値　266, 363
　CS_TDS_46 の値　266, 363
　CS_TDS_50 の値　266, 363
text データと image データの部分更新　333
text と image　82, 339
　CS_IODESC 構造体　99, 495
　CS_TEXTSIZE_OPT オプション　267
　ct_get_data の使用による text 値または image 値のフェッチ　328
　ct_get_data を使用した大きい値の取得　570
　text および image データの保管　328
　text および image の最大値　267
　text および image の最大値プロパティ　267
　text および image のデータの記述　99
　text 値および image 値の挿入　332
　text または image カラムの更新　330, 652
　text または image カラムの取得　328
　後で更新するデータの読み込み　571
　取得前に値の長さを確認する　571
　データ送信コマンド　427
　データのまとまりをサーバに送信　644
　テキスト・タイムスタンプ　328
trusted ユーザのセキュリティ・ハンドシェイク
　CS_SEC_NEGOTIATE プロパティ　585
　セキュリティ・ラベル　584
typedefs
　Open Client　341

# U

unichar データ型　73
　isql および bcp ユーティリティ　75
　機能　73
　制限事項　75
unitext データ
　部分更新での処理　334
unitext データ型　76
　isql および bcp ユーティリティ　77
　機能　76
　制限事項　77
usedir.c サンプル・プログラム　149, 150

## X

xml データ型 78
isql および bcp ユーティリティ 78
機能 78
制限事項 79
XML のデータ型 344

## あ

アプリケーション
アプリケーション開発者の必要事項 326
アプリケーション名プロパティ 236
レイヤ構成 19
ローカライズ 161
暗号化コールバック 45
インストール 389
定義 46
トリガされる方法 28
有効な戻り値 47, 48
呼び出されるタイミング 28
暗号化されたパスワードのセキュリティ・ハンドシェイク 48, 317
暗号化パスワード 45

## い

イベント、コールバック
「コールバック」参照 25
インライン・メッセージ処理
Client-Library タイムアウト・エラー 245
CS_COMMAND 構造体へのポインタの取得 514
CS_EXTRA_INF プロパティ 512
CS_NO_LIMIT による、メッセージ数の制限 514
ct_diag 507
ct_diag による未読メッセージの破棄 512
拡張エラー・データ 145, 514
管理 507
コールバック・ルーチンと比較したときの利点 138
初期化 512
接続のメッセージのクリア 512
メッセージ数の取得 514
メッセージの取得 513
メッセージの制限 513
連続したメッセージ 515
インライン・メッセージ処理のオペレーション
CS_CLEAR 510
CS_EED_CMD 511
CS_GET 510
CS_INIT 509
CS_MSGLIMIT 509
CS_STATUS 510

## え

エラー
タイムアウト 268
エラー処理とメッセージ処理 136, 147
Client-Library によるメッセージ情報の破棄 139
CS_CLIENTMSG 構造体 85
CS_NO_TRUNCATE プロパティによるメッセージのトランケートを禁止 257
CS_SERVERMSG 構造体 103
ct_diag を使用したインライン・メッセージ処理 140
オペレーティング・システム・メッセージ 142
拡張エラー・データ 145
クライアント・メッセージ・コールバックによる Client-Library エラーの処理 33
コールバック方式およびインライン方式についての説明 137
コールバック方式とインライン方式の切り替え 139
異なる接続での処理 139
サーバ・メッセージ情報は廃棄可能 59
サーバ・メッセージ・コールバックによるサーバ・エラーの処理 59
メッセージ構造体 140
メッセージ処理のためのコールバックの使用 139
メッセージのトランケーションを防ぐ 141
連続したメッセージ 141
「インライン・メッセージ処理」参照 507

エラー処理
　タイムアウト　268
演算子
　sizeof　175

## お

応答
　サーバ応答の抑制　80
応答機能　398
オプション
　Adaptive Server　200
　CS_OPT_ANSINULL　201
　CS_OPT_ANSIPERM　201
　CS_OPT_ARITHABORT　201
　CS_OPT_ARITHIGNORE　202
　CS_OPT_AUTHOFF　202
　CS_OPT_AUTHON　202
　CS_OPT_CHAINXACTS　202
　CS_OPT_CURCLOSEONXACT　202
　CS_OPT_DATEFIRST　202
　CS_OPT_DATEFORMAT　202
　CS_OPT_FIPSFLAG　203
　CS_OPT_FORCEPLAN　203
　CS_OPT_FORMATONLY　203
　CS_OPT_HIDE_VCC　203
　CS_OPT_IDENTITYOFF　203
　CS_OPT_IDENTITYON　203
　CS_OPT_ISOLATION　204
　CS_OPT_NOCOUNT　204
　CS_OPT_NOEXEC　204
　CS_OPT_PARSEONLY　204
　CS_OPT_QUOTED_IDENT　204
　CS_OPT_RESTREES　204
　CS_OPT_ROWCOUNT　204
　CS_OPT_SHOW_VI　205
　CS_OPT_SHOWPLAN　205
　CS_OPT_STATS_IO　205
　CS_OPT_STATS_TIME　205
　CS_OPT_STR_RTRUNC　206
　CS_OPT_TEXTSIZE　206
　CS_OPT_TRUNCIGNORE　207
　外部設定　352
　サーバ・オプションのステータスの
　　チェック　590
　サーバ・オプションの設定と取得　585
　接続ごとのサーバ・オプション設定　590
オペレーティング・システム・シグナル
　シグナル・コールバックによる処理　63
オペレーティング・システム・メッセージ
　連続にしない　142

## か

カーソル
　Client-Library カーソル・オープン・
　　コマンド　484
　Client-Library カーソル・クローズ・
　　コマンド　491
　Client-Library カーソル更新コマンド　489
　Client-Library カーソル・コマンドの開始　466
　Client-Library カーソル・コマンドのサーバへの
　　送信　476
　Client-Library カーソル・コマンドの
　　バッチ　487
　Client-Library カーソル削除コマンド　491
　Client-Library カーソル宣言コマンド　478
　Client-Library カーソルによるコマンド構造体の
　　使用　479
　Client-Library カーソル・ロー・コマンド　484
　Client-Library カーソル割り付け解除
　　コマンド　492
　オープン　249
　オプション　484
　カーソル・オープン・コマンドの
　　リストア　249
　カーソル・ローの再位置付け　582
　カーソル・ローの設定　484
　更新　489
　更新カラム　483
　更新カラムの識別　596, 597, 672
　準備された動的 SQL 文での宣言　483
　入力パラメータ値を渡す　487, 596
　ホスト変数フォーマットの定義　596
　読み込み専用 Client-Library カーソル　483
カーソル ID プロパティ　241
カーソル・コマンド
　起動　110

カーソル・コマンド・タイプ
- CS_CURSOR_CLOSE 476
- CS_CURSOR_DEALLOC 476
- CS_CURSOR_DECLARE 474
- CS_CURSOR_DELETE 476
- CS_CURSOR_OPEN 476
- CS_CURSOR_OPTION 475
- CS_CURSOR_ROWS 475
- CS_CURSOR_UPDATE 476

カーソル・ステータス
- 正しいと保証されるカーソル・ステータス 244

カーソル・ステータス・プロパティ 243
カーソル名プロパティ 242
カーソル・ロー結果 281
- 処理 624
- フェッチ 565

カーソル・ロー数プロパティ 242
外部設定ファイル
- 外部設定ファイルでの機能の設定 365
- 関連するプロパティ 353
- 構文 355
- サーバ・オプションの設定 363
- セクション名 354, 355
- デフォルト・ファイル名 353
- プロパティの設定 360
- ロケールの指定 360

隠しキー
- ct_describe 250
- ct_res_info 250
- 定義 250

隠しキーの公開 250
隠しキー・プロパティ 250
隠し構造体
- リスト 82
- CS_BLKDESC 構造体 82
- CS_CAP_TYPE 構造体 82
- CS_COMMAND 構造体 82
- CS_CONNECTION 構造体 82
- CS_CONTEXT 構造体 83
- CS_LOCALE 構造体 83
- CS_LOGINFO 構造体 83
- 関連するルーチン 83

拡張エラー・データ 143
- インライン・エラー処理 145
- サーバ・メッセージ・コールバック 144
- 使用可能かどうかの通知 144
- 利点 144
- 連続したメッセージ 143

拡張エラー・データ・プロパティ 246
カラム
- order-by カラムのカラム ID の取得 617
- order-by 句内のカラム数の取得 617
- カラムの取得 566
- 記述の取得 502
- ブラウズ・モード・カラムに関する情報の取得 384
- プログラム変数へのバインド 372

完了
- 非同期ルーチン 15

完了 ID
- BLK_DONE 603
- BLK_INIT 603
- BLK_ROWXFER 603, 674
- BLK_SENDROW 603, 674
- BLK_SENDTEXT 603, 674
- BLK_TEXTXFER 603, 674
- CT_CANCEL 603, 674
- CT_CLOSE 603, 674
- CT_CONNECT 603, 674
- CT_FETCH 603, 674
- CT_GET_DATA 603, 674
- CT_NOTIFICATION 603
- CT_OPTIONS 603, 674
- CT_RECVPASSTHRU 603, 674
- CT_RESULTS 603, 674
- CT_SEND 603, 674
- CT_SEND_DATA 603, 674
- CT_SENDPASSTHRU 603, 674
- CT_USER_FUNC 603, 674

完了コールバック 16, 37
- インストール 389
- 定義 37
- トリガされる方法 27
- 目的 37
- 有効な戻り値 39
- 呼び出されるタイミング 27
- 呼び出し 673
- 呼び出しできる Client-Library ルーチン 39
- 例 40

完了コールバック・イベント
　発生するとき　25
完了コールバック・サーバ・メッセージ・コール
　　バック　27

## き

キー・カラム
　以前指定した値を ct_fetch を使用して
　　削除する　582
　隠しキーの公開　250
　カラム値の NULL への設定　582
　更新する場合、すべてのキー・カラムの
　　更新　582
　指定　580
　内容の抽出　580
キー・カラムの内容の抽出　580
記号
　CS_CONSTAT_CONNECTED　241
　CS_CONSTAT_DEAD　241
　CS_CURSTAT_CLOSED　244
　CS_CURSTAT_DECLARED　244
　CS_CURSTAT_NONE　243
　CS_CURSTAT_OPEN　244
　CS_CURSTAT_RDONLY　244
　CS_CURSTAT_ROWCOUNT　244
　CS_CURSTAT_UPDATABLE　244
　CS_FIRST_CHUNK　88, 105
　CS_FMT_NULLTERM　96
　CS_FMT_PADBLANK　96
　CS_FMT_PADNULL　96
　CS_FMT_UNUSED　96
　CS_HASEED　105
　CS_LAST_CHUNK　88, 105
　CS_SCROLL_INSENSITIVE　244
　NOSCROLL_INSENSITIVE　244
　SCROLL_CURSOR　244
　SCROLL_SEMISENSITIVE　244
記述結果　283, 628
記述子構造体
　定義と取得　492
記述子領域
　オペレーションの実行　538
　記述子領域の範囲は Client-Library
　　コンテキスト　540
　コンテキスト内でのコマンド構造体の使用
　　540
　定義　540
　名前はコンテキスト内で常にユニーク　540
　パラメータ数またはカラム数の取得　544
　パラメータ数またはカラム数の設定　545
　パラメータの属性の設定　544
　パラメータまたは結果項目の属性の取得　541
　文またはコマンド構造体との関連付け　545
　割当　540
　割り付け解除　541
記述子領域オペレーション
　CS_ALLOC　539
　CS_DEALLOC　539
　CS_GETATTR　539
　CS_GETCNT　539
　CS_SETATTR　539
　CS_SETCNT　539
　CS_USE_DESC　539
起動
　コマンド　110
機能　65, 81
　CS_CAP_REQUEST 機能　398
　CS_CAP_RESPONSE 機能　398
　CS_CAP_TYPE 構造体への保管　175
　CS_CLR_CAPMASK マクロ　406
　CS_SET_CAPMASK マクロ　406
　CS_TDS_VERSION プロパティ　405
　CS_TST_CAPMASK マクロ　406
　ct_capability　405
　TDS バージョン・レベル　405
　外部設定　352
　機能の決定方法　405
　種類　80
　使用　67, 79, 404
　接続　404
　設定と取得　80, 397
　タイプ　398
　複数機能の設定および取得　405

キャンセル
- 結果　392
- 結果破棄の危険性　396
- 現在の結果　627
- コマンド　392
- 残りの結果　627
- バインドへの影響　397

キャンセル・タイプ
- CS_CANCEL_ALL　393
- CS_CANCEL_ATTN　393
- CS_CANCEL_CURRENT　393

共有ディスク・クラスタ
- 証明書の検証　311

## く

クライアント
- クライアントのタイプ　2
- 役割　1

クライアント・メッセージ　137

クライアント・メッセージ・コールバック　33
- Client-Library が呼び出せないとき　139
- インストール　389
- 定義　34
- トリガされる方法　27
- 有効な戻り値　35
- 呼び出されるタイミング　27
- 呼び出し可能な Client-Library ルーチン　36
- 例　36
- 例外動作　31, 391

変換
- クライアント文字セットとサーバ文字セット間　240

クライアント／サーバ
- アーキテクチャ　1
- クライアントとサーバの関連を示す図　1
- 優れた点　2

クリティカル・コード
- CS_NOINTERRUPT プロパティによる保護　259

グローバル・プロパティ
- 取得　452
- 設定　452

## け

計算 ID
- 計算ローの取得　431

計算カラム
- select リスト ID　431
- 記述の取得　502
- 計算カラムの取得　566
- 集合演算子　432
- プログラム変数へのバインド　372, 502

計算結果　282
- bylist 項目数の取得　430
- bylist の取得　430
- select リスト・カラム ID の取得　431
- 計算ローの ID の取得　431
- 集合演算子の取得　431
- 情報　428
- フェッチ　566

計算結果情報タイプ
- CS_BYLIST_LEN　430
- CS_COMP_BYLIST　430
- CS_COMP_COLID　431
- CS_COMP_ID　431
- CS_COMP_OP　431

計算フォーマット結果　627

計算ロー
- ID　431
- 処理　624
- 定義　431

ゲートウェイ・アプリケーション
- Adaptive Server の結果の再パッケージ　247
- TDS パススルー　574, 608, 657
- 暗号化されたパスワードの処理　45, 317
- 位置付け更新と ct_keydata　581
- カーソル情報　245
- カラムのユーザ定義フォーマット文字列
  - を返す　573
- フォーマット情報の取得　627

結果 279, 289
CS_COMPUTE_RESULT 564
CS_CURSOR_RESULT 564
CS_PARAM_RESULT 564
CS_ROW_RESULT 564
CS_STATUS_RESULT 564
ct_fetch を使用した処理 564
ct_results ループ 624
カーソル・ロー結果 281
カラムのユーザ定義フォーマット文字列を返す 572
完全に処理された結果 625
記述結果 283, 628
計算フォーマット結果 627
計算ロー結果 282
結果タイプのリスト 280
結果のキャンセル 392, 627
結果廃棄の危険性 396
現在の結果セット情報 612
現在の結果セットのコマンド番号の取得 615
取得時の変換エラー 565
種類 280
処理 280, 619
処理を示すコード例 284
ステータス結果 282
すべてのコマンドで生成されるわけではない 280
タイプ 564
通常ロー結果 281
定義 280, 563
廃棄 394
パラメータ結果 281
フェッチ 559
フォーマット結果 283
プログラム変数への結果のバインド 371
保留中の結果 625
メッセージ結果 282, 627
ロー結果 281
ロー・フォーマット結果 627
結果項目
値を取得するさまざまな方法 285
結果タイプ
CS_CMD_DONE 619
CS_CMD_FAIL 619
CS_CMD_SUCCEED 619
CS_COMPUTE_RESULT 431, 619
CS_COMPUTEFMT_RESULT 620
CS_CURSOR_RESULT 619
CS_DESCRIBE_RESULT 620
CS_MSG_RESULT 620
CS_PARAM_RESULT 24, 619
CS_ROW_RESULT 619
CS_ROWFMT_RESULT 620
CS_STATUS_RESULT 619
結果データ
記述の取得 501
結果データ項目数の取得 617
定義 563, 623, 639
結果の処理 619
「結果」参照 619
結果の廃棄 394
結果破棄の危険性 396
言語
ネイティブ言語の設定 161
言語カーソル
通常ロー結果セットの生成時 425
言語コマンド
開始 424
起動 110
ホスト変数 425
検索
サーバ・メッセージ・コールバックでのトランザクション・ステータス 147
メインライン・コードでのトランザクション・ステータス 146

## こ

公開された構造体 82
CS_BROWSEDESC 構造体 82
CS_CLIENTMSG 構造体 82
CS_DATAFMT 構造体 82
CS_IODESC 構造体 82
CS_SERVERMSG 構造体 82
SQLCA 構造体 82
SQLCODE 構造体 82
SQLSTATE 構造体 82

公開フォーマット・プロパティ　246
更新
　text または image カラム　330
　キー・カラム　581
更新カラム
　識別　597, 672
構造体　82, 109
　CS_BROWSEDESC 構造体　84
　CS_CAP_TYPE 構造体、ビット操作　175
　CS_CLIENTMSG 構造体　85
　CS_DATAFMT 構造体　93
　CS_IODESC 構造体　99
　CS_SERVERMSG 構造体　103
　SQLCA 構造体　106
　SQLCODE 構造体　108
　SQLDA　549
　SQLSTATE 構造体　108
　親構造体プロパティ　259
　公開された構造体と隠し構造体　82
コールバック　25, 65
　CS_USERDATA を使用した情報の転送　392
　インストール　30, 387, 391
　インストール解除　31, 391
　インライン・メッセージ処理と比較したときの利点　138
　親コンテキストからのコールバック・ルーチンの継承　391
　クライアント・メッセージ・コールバックを参照　27
　継承の影響　391
　コールバック・ルーチンの置き換え　31
　コールバック・ルーチンの定義　31
　取得　387
　種類　26
　セキュリティ・セッション　55, 57
　タイプ　388
　常に実装されているわけではない　30
　トリガされる方法　26
　廃棄される情報　391
　非同期プログラミング　16
　非同期ルーチンの完了でのトリガ　16
　ポインタの取得　31
　メインライン・コードとの情報共有　273
　呼び出されるタイミング　25, 26
　呼び出し可能な Client-Library ルーチン　32
コールバック・イベント　25
　認識　25
　ネットワークから読み込まない場合　25
　廃棄される情報　31
コールバック・タイプ
　CS_CHALLENGE_CB　389
　CS_CLIENTMSG_CB　389
　CS_COMPLETION_CB　389
　CS_DS_LOOKUP_CB　389
　CS_ENCRYPT_CB　389
　CS_NOTIF_CB　389
　CS_SEC_SESSION_CB　389
　CS_SERVERMSG_CB　389
　CS_SIGNAL_CB　389
　CS_SSLVALIDATE_CB　389
コールバックのトリガ　26
国際化のサポート　161, 166
　デフォルトの動作　164
コマンド
　CS_HAVE_CMD プロパティ　248
　ct_command の使用規則　423
　RPC コマンド　426
　開始　418
　開始されているコマンドのクリア　423
　起動　110
　キャンセル　392, 643
　言語コマンド　424
　現在の結果セットのコマンド番号の取得　615
　現在のコマンド情報　612
　コマンドを送信する手順　642
　サーバへの送信　111, 423, 639
　再送信　248
　準備された動的 SQL 文コマンドの開始　530
　データ送信コマンド　426
　パッケージ・コマンド　425
　パラメータの定義　110
　バルク・データ送信コマンド　427
　メッセージ・コマンド　425

コマンド・オプション
CS_BULK_CONT 420
CS_BULK_DATA 420
CS_BULK_INIT 420
CS_COLUMN_DATA 420
CS_END 420
CS_MORE 420
CS_NO_RECOMPILE 420
CS_RECOMPILE 420
CS_UNUSED 420
コマンド構造体
コマンド構造体の割り付けを解除する前に行うこと 411
削除 411
プロパティ 412
割当 409
コマンド・タイプ
CS_LANG_CMD 422
CS_MSG_CMD 422
CS_PACKAGE_CMD 422
CS_RPC_CMD 422
CS_SEND_BULK_CMD 422
CS_SEND_DATA_CMD 422
CS_SEND_DATA_NOCMD 422
コマンド・パラメータの定義 590
コンテキスト・プロパティ 452
cs_config 209, 455
ct_config 210, 455
srv_props 210, 455
外部設定 352
コンテキスト・プロパティの種類 455

## さ

サーバ
interfaces ファイル 251
オプションの設定と取得 585
オプションのリスト 201
サーバからのデータの読み込み 566
サーバ接続のクローズ 406
サーバのタイプ 2
サーバへの接続 433
制限 325
接続先 461
動作 326
トランザクション・ステータス 145
パスワードの定義とクリア 609
役割 1
サーバ・オプション
外部設定 352
サーバからのデータの読み込み 566
サーバ間接続
デフォルト・パスワード 611
パスワードの定義とクリア 609
リモート・パスワードの保管 611
サーバ接続のクローズ 406
Open Client
サーバの動作に依存しない 326
サーバへの接続 433, 461
サーバへのログイン 461
サーバ・メッセージ 137
SQLCODE 構造体へのマッピング 108
拡張エラー・データ 143
サーバ・メッセージ・コールバック 59
インストール 389
拡張エラー・データ 145, 246
トランザクション・ステータスの取得 147
定義 59
トリガされる方法 29
有効な戻り値 60
呼び出されるタイミング 29
呼び出し可能な Client-Library ルーチン 60
例 61
削除
キー・カラム 581
サンプル・プログラム
exconfig 148
firstapp 148
multithrd 148
secct_dec 149
secct_krb 149

## し

シグナル・コールバック 63
インストール 64, 389
定義 64
トリガされる方法 29
呼び出されるタイミング 29

システム・コール、割り込み駆動型 I/O による失敗 17
集合演算子
計算カラムの取得 431
集合演算子のタイプ
CS_OP_AVG 432
CS_OP_COUNT 432
CS_OP_MAX 432
CS_OP_MIN 432
CS_OP_SUM 432
終了
Client-Library 556
取得
compute 句の数 616
order-by カラムのカラム ID 617
order-by 句内のカラム数 617
カラム 566
機能 80
計算カラム 566
計算結果情報 428
結果カラムのユーザ定義フォーマット 572
結果データ項目数 617
結果データの記述 501
現在の結果セット情報またはコマンド情報 612
現在の結果セットのコマンド番号 615
現在のコマンドのロー数 617
現在のサーバのトランザクション・ステータス 618
コマンド構造体情報 412
サーバ・オプション 585
データ、ct_get_data を使用してフェッチ 566
メッセージ ID 616
リターン・パラメータ 566
準備された動的 SQL 文コマンドの開始 530
準備文
Transact-SQL コマンドでホスト変数を指定する方法 533
カーソルの宣言 534
コマンド文は同じ接続に属している必要がある 533
実行 536
出力の記述の取得 535
定義 533
動的 SQL 文コマンドの開始 530
入力パラメータの記述の取得 534
文の準備 533
ユニークな識別子に関連付けられた準備文 533
割り付け解除 538
照合順
指定 161
情報タイプ
CS_BROWSE_INFO 613
CS_CMD_NUMBER 614
CS_ISBROWSE 386
CS_MSGTYPE 614
CS_NUM_COMPUTES 614
CS_NUMDATA 614
CS_NUMORDERCOLS 614
CS_ORDERBY_COLS 614
CS_ROW_COUNT 614
CS_TABNAME 386
CS_TABNUM 386
CS_TRANS_STATE 614
証明書の検証
共有ディスク・クラスタ環境 311
診断サブシステム
有効化／無効化 500
診断情報のトレース 500

## す

スクロール可能カーソル、フェッチ 629
スクロール、ロー
ブラウズ・モード方式 21
ステータス結果 282
ストアド・プロシージャ
ct_results 628
ランタイム・エラー 628
リターン・ステータス記述の取得 502
リターン・ステータス処理 624
リターン・ステータスの取得 567
リターン・パラメータのフェッチ 565
ストアド・プロシージャの結果
リターン・ステータス 282
リターン・パラメータ 281

## せ

制限
Adaptive Server 326
Open Server 326
サーバ 325
整数値データ型 346
セキュリティ
CyberSafe 292
DCE 292
概要 290
データ型 350
ドライバ 292
ネットワークベース 291
メカニズム 292
セキュリティ・セッション
説明 56
ダイレクト 56
セキュリティ・セッション・コールバック
インストール 389
説明 55
定義 57
トリガされる方法 29
呼び出されるタイミング 29
セキュリティ・ラベル
接続単位での数の制限はない 585
定義とクリア 583
接続
CS_CONNECTION 構造体 464
CS_FORCE_CLOSE オプション 407
CS_FORCE_CLOSE の動作 409
CS_MAX_CONNECT プロパティ 434
ct_cancel を使用した「dead」接続のリカバリ 395
ct_con_props を使用したログイン・パラメータの定義 464
TDS バージョン・レベルの変更 80
オープン 461
親コンテキストからのコールバックの継承 31
親コンテキストのプロパティ値の継承 439
外部設定 352
完了コールバックの呼び出し 673
機能 404
強制的なクローズ 408
クローズ 406, 408, 558
最大数の設定 253
ステータスの決定 240
ステータスの検査 240
接続できない例 464
接続の最大数 464
接続の割り付けの解除 408
デフォルトの TDS バージョン・レベル 80
デフォルトのクローズ動作 408
同期または非同期 464
動作の定義 439
非同期オペレーションの完了とレジスタード・プロシージャ・ノーティフィケーションのポーリング 601
非同期ネットワーク I/O の使用 408
保留中の結果 625
「dead」状態であるかの判断 240
「dead」状態であるかの判断 395, 408
「dead」接続のリカバリ 408
「dead」の意味 241
接続構造体
削除 434
接続構造体プロパティ
外部設定 352
設定と取得 436
接続ステータス・プロパティ 240
接続タイプ
LDAP 116
接続プロパティの最大数 253
Open Client
接続マイグレーション 112
接続マイグレーション 112
CS_PROP_MIGRATABLE プロパティを使用して有効にする 112
設定
ct_cmd_props の呼び出し 412
ct_con_props の呼び出し 436
ct_config の呼び出し 452
外部設定ファイル 352
機能 80
サーバ・オプション 585

## そ


送信、サーバへのコマンドの送信　111
即時実行オペレーション
　基準　537


## た


代替サーバ
　接続先　324
タイプ　339, 352
　「データ型」参照　339
タイムアウト
　タイムアウトと非同期接続　253
　タイムアウト・プロパティ　267
　デフォルト値　267
　ログイン・タイムアウト・プロパティ　252
タイムアウト・エラー
　処理　268
タイムスタンプ・カラム
　ブラウズ・モードで使用　22
妥当性チェック　501


## ち


チャレンジ／応答セキュリティ・ハンドシェイク　316
　ネゴシエーション・コールバック　50
チャレンジ／応答セキュリティ・ハンドシエイク　316


## つ


追加情報プロパティ　247
通常ロー結果　281
　処理　624
　フェッチ　565
通信セッション・ブロック・プロパティ　240


## て


定数
　CS_FAIL　34
　CS_ALL_CAPS　81
　CS_ASYNC_IO　13
　CS_BUSY　13
　CS_DEF_PREC　96, 97
　CS_DEFER_IO　13
　CS_MAX_MSG　141
　CS_MAX_PREC　97
　CS_MAX_SCALE　96
　CS_MIN_PREC　97
　CS_MIN_SCALE　96
　CS_MSG_GETLABELS　51
　CS_MSG_LABELS　51
　CS_NULLTERM　95
　CS_SRC_VALUE　96, 97
　CS_SYNC_IO　13
　CS_USER_MAX_MSGID　51
　CS_USER_MSGID　51
ディレクトリ・コールバック
　インストール　389
　説明　41
　定義　42
　トリガされる方法　27
　呼び出されるタイミング　27
　呼び出し手順　44
　例　45
ディレクトリ・サービス
　ct_connect　118, 462
　DCE　120, 125
　interfaces ファイル　115
　Windows レジストリ　121, 126
　エントリの配置　124
　概要　115
　関連するプロパティ　127
　選択　132
　ソフトウェア　115
　名前構文　120
ディレクトリ・スキーマ・ファイル
　ロケーション　122

データ
ct_get_data を使用してサーバからデータを読み込む 566
接続ストリームからの直接読み込み 566
フェッチ 563
フェッチ可能な結果項目の取得 563
プログラム変数へのテーブル・カラムのバインド 371
ユーザ定義データ型 351
ユーザ割り付けデータのコマンド構造体への関連付け 273
ユーザ割り付けデータの接続構造体への関連付け 273
ユーザ割り付けデータの定義 273
データ型
binary 341
datetime 344
money 349
リスト 339
bit 343
CS_BIGDATETIME 345, 346
CS_BIGINT 347
CS_BIGTIME 345, 346
CS_BINARY 342
CS_BIT 343
cs_calc 341
CS_CHAR 343
cs_cmp 341
cs_convert 341
CS_DATE 345
CS_DATETIME 345
CS_DATETIME4 345
CS_DECIMAL 348
cs_dt_crack 341
cs_dt_info 341
CS_FLOAT 347
CS_IMAGE 350
CS_INT 347
CS_LONGBINARY 342
CS_LONGCHAR 343
CS_MONEY 349
CS_MONEY4 349
CS_NUMERIC 348
CS_REAL 347
CS_SMALLINT 347
cs_strcmp 341
CS_TEXT 350
CS_TIME 345
CS_TINYINT 347
CS_UBIGINT 347
CS_UINT 347
CS_UNICHAR 343
CS_UNITEXT 350
CS_VARBINARY 342
CS_VARCHAR 343
XML 344
記述のための構造体 93
整数 346
セキュリティ 350
データ型を操作するルーチン 341
文字 343
ユーザ定義データ型 351
データ型のサポート
Sybase Client/Server のデータ型 339
データ構造体検証 500
データ送信コマンド
CS_IODESC 構造体が必要 652
開始 426, 644
起動 110
データ・フォーマット構造体 93
テキスト・タイムスタンプ 328
テスト 240
デバッグ 113
環境変数による有効化 113
妥当性チェック 501
データ構造体検証 501
デバッグ・ファイルの指定 500
デバッグ・ライブラリ・オペレーションの管理 496
パフォーマンスへの影響 501
非同期プログラムへの影響 501
メモリ参照チェック 500
デバッグ・オペレーション
CS_CLEAR_FLAG 500
CS_SET_DBG_FILE 500
CS_SET_FLAG 500
CS_SET_PROTOCOL_FILE 500

デバッグ・フラグ
- CS_DBG_ALL 498
- CS_DBG_API_STATES 498
- CS_DBG_APISTATES 498
- CS_DBG_ASYNC 498
- CS_DBG_DIAG 498
- CS_DBG_ERROR 498
- CS_DBG_MEM 498
- CS_DBG_NETWORK 498
- CS_DBG_PROTOCOL 498
- CS_DBG_PROTOCOL_STATES 499
- 外部設定 352

## と

動的 SQL 401
- 記述情報の処理 624
- 記述子領域でのオペレーションの実行 538
- コマンドのサーバへの送信 532
- 準備された動的 SQL 文コマンドの開始 530

動的 SQL オペレーション
- CS_CURSOR_DECLARE 531
- CS_DEALLOC 531
- CS_DESCRIBE_INPUT 531
- CS_DESCRIBE_OUTPUT 531
- CS_EXEC_IMMEDIATE 531
- CS_EXECUTE 531
- CS_PREPARE 531

動的 SQL コマンド
- 起動 110

トランザクション・ステータス 145
- CS_TRAN_COMPLETED 146
- CS_TRAN_FAIL 146
- CS_TRAN_IN_PROGRESS 146
- CS_TRAN_STMT_FAIL 146
- CS_TRAN_UNDEFINED 146
- 現在のサーバのトランザクション・ステータスの取得 618
- サーバ・メッセージ・コールバックの取得 146
- 情報が利用可能な場合 146
- メインライン・コードでの取得 146

トランザクション名プロパティ 270

## な

長いメッセージ 141

## に

入力パラメータ値
- 渡す 598

## ね

ネイティブ言語サポート 161

ネゴシエーション・コールバック 49
- trusted ユーザのセキュリティ・ハンドシェイク 50
- インストール 389
- チャレンジ／応答セキュリティ・ハンドシェイク 50
- 定義 50
- トリガされる方法 28
- 有効な戻り値 52
- 呼び出されるタイミング 28

ネゴシエートされたプロパティ 209

ネットワーク I/O プロパティ 255
- 制限 257

Open Client
- ネットワーク・サービス 4

## の

ノーティフィケーション・コールバック 52
- インストール 389
- 定義 53
- トリガされる方法 28
- 有効な戻り値 54
- 呼び出されるタイミング 28
- 呼び出し可能な Client-Library ルーチン 54

ノーティフィケーション・コールバック・イベント
- 発生するとき 26

## は


バージョン
Client-Library 578
Client-Library バージョン・プロパティ 275
Client-Library バージョン文字列プロパティ 275
CS_VERSION プロパティの値の決定 579
バージョン番号
設定 575
配列バインド 383
バインド
ct_describe 380
ct_res_info 380
大きい値のバインド 379
カラムを配列にバインド 383
代わりに ct_get_data を使用 379
継続 261, 381
効果の継続時間 261, 289, 381
再バインド 380
配列バインド 383
バインドのクリア 372, 380
バインドのスタイル 235
バッチ処理 289
複数の変数へのバインドはできない 380
プログラム変数への結果のバインド 371
目的 379
パケット
TDS パケットの受信 607
プラットフォームによって異なるデフォルト・パケット・サイズ 608
パスワード
リモート・サーバのパスワードの定義とクリア 609
リモート・サーバ用のデフォルト・パスワード 611
リモート・パスワードの保管 611
パスワード暗号化ハンドラ
カスタムの暗号化技術用パスワード暗号化ハンドラ 45
ゲートウェイ・アプリケーションで使用 45
デフォルト 45
パッケージ・コマンド
開始 425
起動 110
目的 426
パラメータ
NULL 値を渡す 599
コマンドのパラメータの定義 110
定義 590
データ型の変換 596, 669
入力パラメータ値を渡す 598
パラメータ結果 281
プログラム変数へのバインド 372
バルク・コピー
CS_IODESC 構造体 99
バルク・コピー・オペレーションのプロパティ 240
バルク・コピー・データの記述 99
バルク・データ送信コマンド
開始 427
ハンドシェイク
trusted ユーザのセキュリティ 50
暗号化されたパスワードのセキュリティ 45, 48, 317
チャレンジ／応答セキュリティ 316


## ひ


ビット
CS_CANBENULL 98, 504
CS_HIDDEN 23, 98, 504
CS_IDENTITY 98, 504, 505
CS_INPUTVALUE 98
CS_KEY 98, 504
CS_RETURN 98, 504
CS_TIMESTAMP 23, 98, 504
CS_UPDATABLE 98, 504
CS_UPDATECOL 98, 504
CS_VERSION_KEY 98, 504
非同期動作
Client-Library ルーチン 13
CS_BUSY を返す 14
CS_NETIO プロパティを使用した有効化 13

非同期プログラミング　12, 21
　ct_poll　601
　ct_wakeup　675
　オペレーションが未処理状態のときに呼び出し可能なルーチン　14
　完了コールバックの定義　37
　タイミング問題に関わる動作にデバッグが与える影響　501
　遅延非同期接続の設定　256
　非同期ルーチンの一覧　14
　非同期ルーチンの完了について　14
　ポーリングの抑制　245
　メモリ・プールのプロパティ　19
　メモリ要件　18, 254, 271, 272
　ユーザ割り付け関数のプロパティ　19
　レイヤ構成のアプリケーション　19
　ローのフェッチ　563
非同期プログラミング用のレイヤ構成のアプリケーション
　ct_callback　21
　ct_poll　20
　ルーチン完了の報告を抑制　20, 245

## ふ

フェッチ
　カーソル・ロー　565
　計算ロー　566
　結果データ　559
　通常ロー　565
　データ、ct_get_data を使用してフェッチ　566
　リターン・ステータス　565
　リターン・パラメータ　565
フェッチ、スクロール可能カーソル　629
フォーマット
　公開フォーマット・プロパティ　246
　データ・フォーマットの記述　93
　日時、通貨、数値にネイティブ・フォーマットを使用　161
　ホスト変数フォーマットの定義　597
フォーマット結果　283
　CS_EXPOSE_FMTS を必ず有効にする　628
　カラムのユーザ定義フォーマット文字列を返す　572
フォーマット結果セット
　説明　247
フォーマット情報
　実データより先行　627
　取得　502
　処理　624
複数ユーザによる更新
　ブラウズ・モードでの調整　23
部分更新
　ct_send_data　334
　unitext データの処理　334
ブラウズ・モード　21, 25
　CS_BROWSEDESC 構造体　84
　CS_HIDDEN_KEYS プロパティ　23
　select...for browse コマンド　23
　アドホック・クエリと ct_br_column　22
　アドホック・クエリと ct_br_table　22
　カラムの更新条件　385
　実装の手順　22
　使用条件　24
　接続の必要条件　22
　ブラウズ可能なテーブルの属性　387
　ブラウズ・モード・カラムに関する情報の取得　384
　ブラウズ・モード情報が使用可能な場合　385, 614
　ブラウズ・モード・テーブルに関する情報の取得　385
　目的　22
プログラミング
　非同期　12, 21
　「非同期プログラミング」参照　12
Open Client
　プログラミング・インタフェース　4
プログラム
　例　147

プロパティ 208, 276
リスト 212
Client-Library 固有のコンテキスト・プロパティ 455
CS_ANSI_BINDS 213, 361, 440, 455
CS_APPNAME 213, 361, 440
CS_ASYNC_NOTIFS 213, 361, 441
CS_BULK_LOGIN 213, 361, 441
CS_CHARSETCNV 213, 441
CS_COMMBLOCK 214, 441
CS_CON_KEEPALIVE 441
CS_CON_STATUS 214, 441
CS_CON_TCP_NODELAY 442
cs_config 209
CS_CONFIG_BY_SERVERNAME 215, 442
CS_CONFIG_FILE 215, 442
CS_CONNECTED_ADDR 214, 441
CS_CUR_ID 215, 417
CS_CUR_NAME 216, 417
CS_CUR_ROWCOUNT 216, 417
CS_CUR_STATUS 216, 417
CS_DIAG_TIMEOUT 216, 361, 442
CS_DISABLE_POLL 217, 361, 443, 455
CS_DS_COPY 217, 361
CS_DS_DITBASE 217, 361
CS_DS_EXPANDALIAS 217
CS_DS_FAILOVER 218, 361
CS_DS_PASSWORD 218, 361
CS_DS_PRINCIPAL 218, 361
CS_DS_PROVIDER 219, 361
CS_DS_SEARCH 219
CS_DS_SIZELIMIT 219
CS_DS_TIMELIMIT 220
CS_EED_CMD 220, 444
CS_ENDPOINT 220, 445
CS_EXPOSE_FMTS 220, 361, 445, 456
CS_EXTENDED_ENCRYPT_CB 445
CS_EXTERNAL_CONFIG 221, 445, 456
CS_EXTRA_INF 221, 361, 362, 363, 445, 456
CS_HAVE_BINDS 221, 417
CS_HAVE_BINDS ( 詳細な説明 ) 248
CS_HAVE_CMD 221, 248, 417
CS_HAVE_CUROPEN 222, 418
CS_HIDDEN_KEYS 222, 361, 418, 445, 456
CS_HOSTNAME 222, 361, 445
CS_IFILE 222, 361, 456
CS_LOC_PROP 222, 446
CS_LOGIN_STATUS 222, 446
CS_LOGIN_TIMEOUT 223, 361, 456
CS_LOOP_DELAY 361
CS_MAX_CONNECT 223, 361, 456
CS_MEM_POOL 223, 456
CS_NETIO 223, 361, 446, 457
CS_NO_TRUNCATE 224, 362, 457
CS_NOAPICHK 224, 361, 457
CS_NOCHARSETCNV_REQD 224, 446
CS_NOINTERRUPT 224, 362, 457
CS_NOTIF_CMD 225, 446
CS_PACKETSIZE 225, 362, 446
CS_PARENT_HANDLE 225, 418, 446
CS_PARTIAL_TEXT 226, 447, 457
CS_PASSWORD 226, 227, 362, 447
CS_PROP_APPLICATION_SPID 226, 447
CS_PROP_EXTENDEDFAILOVER 226
CS_PROP_MIGRATABLE 226, 447, 457
CS_RETRY_COUNT 227, 228, 362
CS_SEC_APPDEFINED 228, 362, 448
CS_SEC_CHALLENGE 229, 362, 448
CS_SEC_CHANBIND 362
CS_SEC_CONFIDENTIALITY 362
CS_SEC_CREDTIMEOUT 362
CS_SEC_DATAORIGIN 362
CS_SEC_DETECTREPLAY 362
CS_SEC_DETECTSEQ 362
CS_SEC_ENCRYPTION 231, 362, 449
CS_SEC_EXTENDED_ENCRYPTION 449
CS_SEC_INTEGRITY 362
CS_SEC_KEYTAB 362
CS_SEC_MECHANISM 362
CS_SEC_MUTUALAUTH 362
CS_SEC_NEGOTIATE 232, 450
CS_SEC_NETWORKAUTH 362
CS_SEC_NON_ENCRYPTION_RETRY 450
CS_SEC_SERVERPRINCIPAL 363
CS_SEC_SESSTIMEOUT 363
CS_SENDDATA_NOCMD 233, 451
CS_SERVERADDR 233, 451
CS_SERVERNAME 232, 233, 451
CS_STICKY_BINDS 233, 261, 418
CS_TCP_RCVBUF 460
CS_TCP_SND 460
CS_TDS_VERSION 234, 363, 452

CS_TEXTLIMIT 234, 363, 452, 460
CS_TIMEOUT 234, 363, 460
CS_TRANSACTION_NAME 234, 452
CS_USER_ALLOC 234, 460
CS_USER_FREE 234, 461
CS_USERDATA 235, 418, 452
CS_USERNAME 235, 363, 452
CS_VER_STRING 235, 275, 461
CS_VERSION 235, 275, 461
CS-Library 固有のコンテキスト・プロパティ 455
ct_cmd_props 209
ct_con_props 209
ct_config 209
CCS_SEC_DELEGATION 362
Server-Library 固有のコンテキスト・プロパティ 455
外部設定 352
コマンド構造体プロパティ 412
コンテキスト構造体プロパティ 452
コンテキスト・プロパティの種類 209, 455
サーバ・オプションとの比較 208
接続構造体プロパティ 436
デフォルト値 209
ネゴシエートされたプロパティ 209
プロパティの設定と取得 209
まとめ 212
ログイン・プロパティ 209
ログイン・プロパティのコピー 211


## へ


ヘッダ・ファイル 153
csconfig.h 153
cspublic.h 153
cstypes.h 89, 153, 175
ctpublic.h 153
example.h 148
exasync.h 148
exutils.h 148
sqlca.h 153
変数
プログラム変数への結果のバインド 371
ホスト変数フォーマットの定義 597


## ほ


ポーリング
接続 601
無効化 245
ホスト変数
フォーマットの定義 597
ホスト名プロパティ 250
保留中の結果 625


## ま


Open Client
マクロ 175
マクロ
CS_CLR_CAPMASK 81
CS_LAYER 89
CS_NUMBER 89
CS_ORIGIN 89
CS_SET_CAPMASK 81
CS_SEVERITY 89
CS_TST_CAPMASK 81
Open Client マクロ 166
SQLDA_DECL 550
SYB_SQLDA_SIZE 550
定義 175


## め


メインライン・コード
コールバック・ルーチンとの情報共有 273
トランザクション・ステータスの取得 146
メッセージ
bcp 677, 693
defncopy 677, 693
isql 677, 693
長い 141
連続 141
「エラー処理とメッセージ処理」参照 136
メッセージ ID
メッセージ ID の取得 616
メッセージ結果 282, 627
処理 624

メッセージ・コマンド
開始 425
起動 110
目的 425
ユーザ定義メッセージの有効な範囲 425
メッセージ・コマンド識別子 425
メッセージの重大度
CS_SV_API_FAIL 87, 91
CS_SV_COMM_FAIL 87, 91
CS_SV_CONFIG_FAIL 87, 90
CS_SV_FATAL 87, 92
CS_SV_INFORM 87, 90
CS_SV_INTERNAL_FAIL 87, 91
CS_SV_RESOURCE_FAIL 87, 91
CS_SV_RETRY_FAIL 87, 91
メッセージ・パラメータ 281
フェッチ 565
メッセージ番号
復号化 175
メッセージ番号の復号化 175
メモリ解放プロパティ 272
メモリ参照チェック 500
メモリの割り付け
カスタム・メモリ割り付けルーチンのインストール 19
メモリ・プール
ct_config による置き換え 255
ct_config によるクリア 255
メモリ・プール・プロパティ 254
メモリ要件
Client-Library がメモリを確保する方法 19
UNIX システム 255
非同期プログラミング 19
メモリ割り付けプロパティ 271

## も

文字セット
サーバの文字セット変換の無効化 259
指定 161
文字セット変換プロパティ 240
文字セット変換 ( サーバ )
無効化 259

## ゆ

ユーザ解放関数プロパティ 272
ユーザ定義データ型 351
ユーザ定義のメモリ・ルーチン
ct_config による置き換え 272
クリア 272
ユーザ定義フォーマット
取得 572
ユーザ提供のメモリ解放ルーチン
識別 272
ユーザ・データ・プロパティ 273
ユーザ名プロパティ 274
ユーザ割り付け関数プロパティ 271
ユーザ割り付けデータ
cs_config 273
定義 273

## よ

要求
サポートされる要求のタイプの決定 80
要求機能 398

## り

リターン・ステータス
ストアド・プロシージャのリターン・ステータスの取得 567
フェッチ 566
プログラム変数へのバインド 372
リターン・パラメータ
記述の取得 502
処理 624
フェッチ 565
リターン・パラメータの取得 567
リテラル文
動的 SQL リテラル文の実行 537
リモート・プロシージャ・コール
開始 426
結果の処理 426
サーバ間通信 610
目的 426

## れ


レイヤ構成のアプリケーション
  ct_wakeup 20
  非同期プログラミング 19
  例 21
レイヤ構成のアプリケーションによる非同期プログラミング
  ct_callback 21
  ct_poll 20
  ct_wakeup 20
  非同期ルーチン完了の報告を抑制 20
  例 21
レジスタード・プロシージャ 276, 279
  CS_ASYNC_NOTIFS プロパティ 279
  説明 276
  ノーティフィケーション・コールバック 52
  ノーティフィケーション・コールバックのインストール 389
  ノーティフィケーションの処理 52
  ノーティフィケーションのポーリング 601
  ノーティフィケーションを受信した場合 278
  引数の取得 53
  非同期ノーティフィケーション・プロパティ 237
  メリット 277
連続したメッセージ 141
  ct_diag 143
  拡張エラー・データ 143
  メッセージ構造体フィールド 142


## ろ


ロー
  直前のコマンドで影響を受けたローの数 248, 617
ローカライゼーション
  Client-Library が値を検索する場所 164
  cs_config 252
  cs_locale 166
  CS_LOCALE 構造体 161
  ct_con_props 252
  親コンテキストからの値の継承 163
  カスタム値の設定 161
  コンテキスト・レベル 163
  接続レベル 163
  データ要素レベル 164
  デフォルト値 161
ロー結果 281
ロー・フォーマット結果 627
ログイン応答情報
  転送 573, 658
ログイン・ステータス・プロパティ 252
ログイン・タイムアウト・プロパティ 252
ログイン・プロパティ 209
  新しい接続にコピーする方法 211, 574, 659
ログイン名
  定義 275
ロケール情報 161
ロケール情報プロパティ 252
ロケール・ファイル
  エントリ 165
  定義済みロケール名 166
  役割 165
ロケール名
  定義済み 166


## わ


割当
  CS_COMMAND 構造体 409
  CS_CONNECTION 構造体 432
割り込み
  CS_NOINTERRUPT プロパティによる割り込みの禁止 259
  割り込みが起こる状況の例 259
割り込み禁止のプロパティ 259
割り込み駆動型 I/O
  システム・コールの失敗 17
割り込みレベル
  メモリ要件 19