SYBASE®

Server-Library/C リファレンス・マニュアル

Open Server™

15.5

ドキュメント ID：DC32626-01-1550-01
改訂：2009 年 10 月

マニュアルの注文

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

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

目次

はじめに

このマニュアルでは、Open Server™ Server-Library (C バージョン用) のリファレンス情報について説明します。

対象読者

『Open Server Server-Library/C リファレンス・マニュアル』は、Open Server アプリケーションを作成するプログラマを対象としたリファレンス・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 Open Server の概要」では、Open Server の概要について説明します。
- 「第 2 章 トピック」では、「サーバ」から text 値や image 値を読み込むために「Server-Library」ルーチンを使用するなど、特定のプログラミング・タスクを実行する方法について説明します。また、Open Server の構造体、プログラミング技法、エラー処理に関する情報も提供します。
- 「第 3 章 ルーチン」では、個々のルーチンが使用するパラメータや戻り値など、各 Server -Library ルーチンに関する固有の情報について説明します。
- 「第 4 章 システム・レジスタード・プロシージャ」では、Server-Library が自動的に提供するレジスタード・プロシージャについて説明します。また、パラメータ、結果、メッセージに関する情報も提供します。

関連マニュアル

詳細については、次のマニュアルを参照できます。

- 『Open Server リリース・ノート Microsoft Windows 版』には、Open Server に関する重要な最新情報が記載されています。
- 『Software Developer's Kit リリース・ノート Microsoft Windows 版』には、Open Client™ および SDK に関する重要な最新情報が記載されています。
- 『jConnect for JDBC リリース・ノート バージョン 6.05 および 7.0』には、jConnect™ に関する重要な最新情報が記載されています。
- 『Open Client/Server 設定ガイド Microsoft Windows 版』では、システムを設定して Open Client/Server 製品を実行する方法について説明しています。
- 『Open Client Client-Library/C リファレンス・マニュアル』では、Open Client Client-Library のリファレンス情報について説明しています。

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

その他の情報

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

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

- SyBooks CD には製品マニュアルが収録されています。この CD は製品のソフトウェアに同梱されています。Eclipse ベースの SyBooks ブラウザを使用すれば、使いやすい HTML 形式のマニュアルにアクセスできます。

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

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

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

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

Web 上の Sybase 製品の動作確認情報

Sybase Web サイトの技術的な資料は頻繁に更新されます。

❖ 製品認定の最新情報にアクセスする

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

❖ コンポーネント認定の最新情報にアクセスする

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

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

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

1 Web ブラウザで Technical Documents を指定します。
(http://www.sybase.com/support/techdocs/)

2 [MySybase] をクリックし、MySybase プロファイルを作成します。

Sybase EBF とソフトウェア・メンテナンス

❖ **EBF とソフトウェア・メンテナンスの最新情報にアクセスする**

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

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

3 製品を選択します。

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

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

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

表記規則

表 *1:* 構文の表記規則

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

アクセシビリティ機能

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

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

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

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

不明な点があるときは

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

第 1 章

Open Server の概要

この章の内容は、次のとおりです。

クライアント／サーバ・アーキテクチャの概要

クライアント／サーバ・アーキテクチャは、コンピューティング作業をクライアントとサーバ間で分担します。

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、あるクライアント・アプリケーションはデータベース・サーバの温度データを要求し、部屋の温度を下げるよう環境制御サーバに要求する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、受け取ったメールを最終的な宛先に転送します。

クライアント／サーバ・アーキテクチャには、従来のプログラム・アーキテクチャよりも優れた点がいくつかあります。

- 共通するサービスが 1 か所で、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大きさや複雑さがかなり軽減されます。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の通信を容易にします。異なった通信プロトコルを使用するクライアント・アプリケーション同士は、そのままの状態では直接通信できませんが、両方のプロトコルを「認識する」サーバを介せば通信できるようになります。このようなサーバを「ゲートウェイ」と呼びます。
- クライアント／サーバ・アーキテクチャを使用すると、アプリケーションを、独立した個々のコンポーネントの集合体として開発できます。これらのコンポーネントは、アプリケーションの他の部分に影響を与えずに、変更したり置き換えたりできます。

クライアントのタイプ

クライアントとは、サーバへ要求を行うアプリケーションのことです。Sybase のクライアントには、次のものが含まれています。

- Sybase SQL Toolset™ 製品
- isql や bcp といった Adaptive Server Enterprise が提供する独立したユーティリティ
- 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
- Embedded SQL™ を使用して記述されたアプリケーション
- PowerBuilder® 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・サーバの作成に必要なツールやインタフェースを提供します。Open Server で構築したカスタム・サーバを「Open Server アプリケーション」と呼びます。

Open Server アプリケーションは、あらゆる種類のサーバとして機能します。たとえば、Open Server アプリケーションは、特殊な計算を実行したり、リアルタイムでデータにアクセスしたり、電子メールのようなサービスに対するインタフェースとして機能することができます。Open Server アプリケーションは、Open Server Server-Library から提供されるルーチンを組み合わせて個々に開発します。

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。

- Adaptive Server Enterprise と Open Server はどちらもサーバであり、クライアントの要求に応答する。
- クライアントは Open Client ライブラリを介して、Adaptive Server Enterprise と Open Server の両方のアプリケーションと通信する。

しかし、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、Open Server から提供されるルーチンを使用してカスタム・コードを開発し、Open Server アプリケーションを作成する必要がある。Adaptive Server Enterprise は完全な製品なので、カスタム・コードを必要としない。
- Open Server アプリケーションは、あらゆる種類のサーバとして使用できるため、さまざまなプログラム言語を使用できるように作成できる。Adaptive Server Enterprise はデータベース・サーバであり、Transact-SQL のみを認識する。
- Open Server アプリケーションは、Sybase の Tabular Data Stream™ (**TDS**) プロトコルに基づいていない「外部」のアプリケーションやサーバと通信できる。また、Sybase のアプリケーションやサーバとも通信できる。Adaptive Server Enterprise が直接通信できるのは、Sybase のアプリケーションとサーバだけである。外部アプリケーションや外部サーバと通信する場合、Adaptive Server Enterprise は Open Server のゲートウェイ・アプリケーションを仲介として使用しなくてはならない。

Open Server の構成

クライアント／サーバ・アーキテクチャにおける Open Server アプリケーションの位置付けは、Open Server がどのように機能しているかによって異なります。Open Server アプリケーションの機能は、次の 3 つに分類されます。

- スタンドアロン
- 補助
- ゲートウェイ

スタンドアロン Open Server アプリケーション

クライアントは、スタンドアロン Open Server アプリケーションに直接接続します。

クライアントは、次の方法を使用して要求をサーバに送信します。

- リモート・プロシージャ・コール (RPC) － RPC の呼び出しによって、Open Server アプリケーションで「レジスタード・プロシージャ」を実行できます。レジスタード・プロシージャは、Open Server アプリケーションに格納される Open Server のコードの一部として定義されます。これらは、ユーザ定義またはシステム定義のプロシージャです。
- 「カーソル」コマンド
- さまざまなクライアント「コマンド」

Open Server アプリケーションのプログラマは、クライアント・コマンドを処理するコードを記述します。

スタンドアロン Open Server アプリケーションは、クライアント要求に応答するために外部要求を行いません。

補助 Open Server アプリケーション

補助 Open Server アプリケーションは、リモート・プロシージャ・コール (RPC) を処理することによって Adaptive Server Enterprise をサポートします。

クライアントは、Transact-SQL を使用し、Adaptive Server Enterprise に直接接続します。Open Server アプリケーション上でレジスタード・プロシージャを実行する場合、クライアントは、Transact-SQL 文のプロシージャ名のプレフィクスとして、使用する Open Server アプリケーション名を指定します。Adaptive Server Enterprise は、この名前を使用して RPC を開始します。たとえば、次のクライアント文では、“OpnSrv211” という名前の Open Server アプリケーション上で “print_calls” プロシージャを実行します。

```
exec OpnSrv211...print_calls
```

RPC は、Adaptive Server Enterprise から Open Server アプリケーションに直接送信できる唯一のクライアント・コマンドです。Adaptive Server Enterprise 内でストアド・プロシージャ、トリガ、またはスレッショルド管理を使用して RPC 呼び出しを開始できます。RPC を使用すると、次の機能にアクセスできます。

- 電子メールの送信と印刷などのオペレーティング・システム機能
- Open Server アプリケーションのコードに定義したすべての機能

Open Server アプリケーションは、Adaptive Server Enterprise に情報を返したり、Adaptive Server Enterprise を経由してクライアントに情報を戻したりすることができます。

サーバ間の RPC を使用することによって、Open Server アプリケーションは、特殊な計算を実行したり、リアルタイムでデータにアクセスしたり、電子メールなどのサービスへのアクセスを Adaptive Server Enterprise に許可したりすることができます。

ゲートウェイ Open Server アプリケーション

ゲートウェイ・サーバを使用すると、クライアントは、クライアント接続を直接受け入れるかどうかわからないサーバにもアクセスできます。ゲートウェイは、Adaptive Server Enterprise などの DBMS に接続する必要はありません。ファイル・システムや、サーバとして動作可能なアプリケーション・プログラムであれば、ゲートウェイを接続できます。

Adaptive Server Enterprise や他の Open Server アプリケーションにアクセスする Open Server アプリケーションは、Client-Library ルーチンと Server-Library ルーチンの両方を含んでいます。両方のルーチンを含むことで、クライアントとサーバのどちらの役割も果たすことができます。サーバの役割としては、クライアントとのインタフェースとして Open Server を使用します。クライアントの役割としては、Client-Library ルーチンを使用して、Adaptive Server Enterprise や他の Open Server に要求を送信したり、結果を受け取ったりします。詳細については、「ゲートウェイ・アプリケーション」(91 ページ) を参照してください。

上記のゲートウェイは、クライアントを Adaptive Server Enterprise に接続します。図内の点線は、この特定のゲートウェイが Sybase クライアントと Sybase サーバ間で要求と結果を受け渡すオーバヘッドの小さい方法である “TDS **パススルー** モード ” を使用することを示しています。詳細については、「パススルー・モード」(121 ページ) を参照してください。

Open Server

Open Server では、カスタム・サーバ・アプリケーション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 Server は、大きく分けて、関数のライブラリとしてのプログラミング・インタフェースとネットワーク・サービスで構成されます。

Open Server ライブラリ

Open Server プログラミング・インタフェースを構成しているライブラリは、次のとおりです。

- Server-Library：サーバ・アプリケーションの作成に使用するルーチンの集まりです。Server-Library には、次のルーチンが含まれます。
  - クライアントから送信されたコマンドを受信するルーチン
  - 結果をクライアントに返すルーチン
  - アプリケーション属性を設定するルーチン
  - エラー条件を処理するルーチン
  - クライアントとの対話をスケジュールするルーチン
  - クライアント接続についてのさまざまな情報を提供するルーチン
- 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。Server-Library ルーチンは CS-Library に割り当てられている構造体を使用しているので、すべての Server-Library プログラムは CS-Library への呼び出しを最低 1 つは含んでいなければなりません。

  (Open Client と Open Server はともに CS-Library を使用しますが、この CS-Library には、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両方のユーティリティ・ルーチンが入っています。)

スタンドアロン Open Server アプリケーション、および補助 Open Server アプリケーションには、Server-Library と CS-Library への呼び出しが含まれています。ゲートウェイ Open Server アプリケーションには、Server-Library、CS-Library、Client-Library への呼び出しが含まれています。

Open Server には、Server-Library ルーチンが使用する構造体、型、値を定義する次のヘッダ・ファイルが含まれています。

- *ospublic.h*
- *oserror.h*
- *oscompat.h*

ネットワーク・サービス

通常、Open Server のネットワーク・サービスは、Open Server 開発者や Open Server アプリケーションのエンド・ユーザにとって透過的で意識されません。ただし、PC プラットフォームではネットワーク・サービスは外部に存在し、意識されます。

ネットワーク・サービスには Net-Library があり、TCP/IP 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。

Open Server の使用

Server-Library および CS-Library への呼び出しを使用して、構造体の設定、クライアントや他のサーバからの接続要求の受信、クライアントからの要求の処理、メモリのクリーンアップを行い、Open Server アプリケーションを作成します。ゲートウェイ・アプリケーションにも、Client-Library ルーチンへの呼び出しが含まれています。

Open Server アプリケーション・プログラムは、C 言語プログラムと同じようにコンパイルされます。ほとんどの UNIX プラットフォームでは、プログラムのコンパイルやリンクを実行するときには次のライブラリが必要です (ファイル名や拡張子はプラットフォームによって異なります)。

- *libsybsrv.a*
- *libsybcs.a*
- *libsybcomn.a*
- *libsybtcl.a*
- *libsybintl.a*
- *libsybblk.a* － バルク・コピー・ルーチンを使用している場合
- *libsybct.a* － ゲートウェイを使用している場合

ライブラリ・ファイルは、*$SYBASE/$SYBASE_OCS/lib* ディレクトリにあります。

CS_CONTEXT 構造体

Open Server アプリケーションには、特定のアプリケーション「コンテキスト」やオペレーション環境を定義する CS_CONTEXT 構造体が必要です。
CS_CONTEXT には、サーバワイドな制御情報だけでなく、ローカライゼーション情報も含まれています。どのような Open Server アプリケーション・プログラムでも、最初に cs_ctx_alloc を呼び出して CS_CONTEXT 構造体を割り付けます。

アプリケーション・プログラマは、CS_CONTEXT 構造体の内容を操作することによってアプリケーションの動作や属性を決定します。「プロパティ」(130 ページ) を参照してください。

簡単なプログラムの手順

ほとんどのプラットフォームでは、次の手順に従って、簡単な Open Server アプリケーション・プログラムを作成できます。

手順	機能	ルーチン
1	構造体の割り付け、および「プロパティ」と呼ばれるグローバル属性の設定を行い、Open Server オペレーティング環境を設定する。	cs_ctx_alloc srv_version srv_props
2	エラー処理を定義する。アプリケーションにエラー処理ルーチンをインストールし、エラーが検出されたときに Open Server がこのルーチンを呼び出すようにする。アプリケーションは、アドホック・ベースで srv_sendinfo ルーチンを呼び出してクライアントにエラー・メッセージを送信することもできる。また、srv_log を呼び出してログ・ファイルに書き込むこともできる。詳細については、「エラー」(81 ページ) を参照。	srv_props(SRV_S_ERRHANDLE)
3	サーバを初期化する。	srv_init
4	クライアント・コマンドが Open Server イベントをトリガするときに Open Server が呼び出すイベント処理ルーチンをインストールする。Open Server アプリケーションの作業のほとんどが、イベント処理ルーチンの中で行われる。「Open Server イベント」(13 ページ) を参照。	srv_handle
5	サーバを起動させる。この状態では、サーバはクライアント要求を受信するだけである。	srv_run
6	クリーンアップして終了する。	cs_ctx_drop

次の項のサンプル・プログラムには、手順 4 を除くすべての手順を示します。この例では、ユーザ定義のイベント・ハンドラをインストールするのではなく、デフォルトのハンドラを実行します。

基本的な Open Server プログラム

以下に、基本的な Open Server アプリケーション・プログラムのコードを示します。

```
/*
 **    This program demonstrates the minimum steps necessary
 **    to initialize and start up an Open Server application.
 **    No user-defined event handlers are installed, therefore
 **    the default handlers will execute instead.
 */

/*
 ** Include the required Open Server header files.
 **
 **    ospublic.h: Public Open Server structures, typedefs,
 **    defines, and function prototypes.
 **
 **    oserror.h:   Open Server error number #defines.This header
 **    file is only required if the Open Server application wants
 **    to detect specific errors inside the Open Server error
 **    handler.
 */

#include        <ospublic.h>
 #include        <oserror.h>

/*
 **    Include the operating system specific header files required
 **    by this Open Server application.
 */
 #include        <stdio.h>

 /*
 ** Local defines.
 **
 **    OS_ARGCOUNT    Expected number of command line arguments
 */
 #define        OS_ARGCOUNT    2


 /*
 **    This Open Server application expects the following
 **    command line arguments:
 **
 **    servername: The name of the Open Server application.
 **
 **    This name must exist in the interfaces file defined by
 **    the SYBASE environment variable.
 **
 ** Returns:
 **    0        Open Server exited successfully.
 **    1        An error was detected during initialization.
```

```
*/


int    main(argc, argv)
int    argc;
char    *argv[];
{
    CS_CONTEXT    *cp;              /* Context structure */
    CS_CHAR       *servername;      /* Open Server name */
    CS_CHAR       logfile[512];     /* Log file name */
    CS_BOOL       ok;               /* Error control flag */
    SRV_SERVER    *ssp;             /* Server control structure*/

    /* Initialization.      */
    ok = CS_TRUE;

    /*
    ** Read the command line options.There must be one
    ** argument specifying the server name.
    */

    if(argc != OS_ARGCOUNT)
    {
        (CS_VOID)fprintf(stderr, "Invalid number of
        arguments(%d)¥n",argc);


        (CS_VOID)fprintf(stderr, "Usage:<program>
        <server name>¥n");
        exit(1);
    }

    /*
    ** Initialize 'servername' to the command line argument
    ** provided.
    */


    servername = (CS_CHAR *)argv[1];

    /*
    ** Allocate a CS-Library context structure to define the
    ** default localization information.  Open Server
    ** also stores global state information in this structure
    ** during initialization.
    */
    if(cs_ctx_alloc(CS_VERSION_155, &cp) != CS_SUCCEED)
    {
        (CS_VOID)fprintf(stderr, "%s: cs_ctx_alloc failed",
        servername);
        exit(1);
    }
```

```
/*
** Default Open Server localization information can be
** changed here before calling srv_version, using cs_config
** and cs_locale.
*/

/*
** Set the Open Server version and context information
*/
if(srv_version(cp, CS_VERSION_155) != CS_SUCCEED)
{
    /*
    ** Release the context structure already allocated.
    */
    (CS_VOID)cs_ctx_drop(cp);

    (CS_VOID)fprintf(stderr, "%s: srv_version failed",
    servername);
    exit(1);
}

/*
** There is no error handler installed in this sample
** Open Server application.Any errors detected by Open
** Server are written to the Open Server log file
** configured below.A real Open Server application would
** install an error handler after calling srv_version, using
** srv_props(SRV_S_ERRHANDLE).Then, any subsequent errors
** will be detected by the Open Server application code.
*/

/*
** Default Open Server global properties can be changed here
** before calling srv_init.We choose just to change the
** default log file name to use the name of this Open
** Server application.
*/

/*
** Build a new Open Server log file name using 'servername'
*/
(CS_VOID)sprintf(logfile, "%s.log", servername);

/*
** Set the new log file name using the global SRV_S_LOGFILE
** property.
*/
if(srv_props(cp, CS_SET, SRV_S_LOGFILE, logfile,
CS_NULLTERM,(CS_INT *)NULL) != CS_SUCCEED)
{
    /*
    ** Release the context structure already allocated.
```

```
        */
        (CS_VOID)cs_ctx_drop(cp);

        (CS_VOID)fprintf(stderr,"%s: srv_props(SRV_S_LOGFILE)
        failed¥n",servername);
        exit(1);
    }

    /*
    ** Initialize Open Server.This causes Open Server to
    ** allocate internal control structures based on the global
    ** properties set above.Open Server also looks up
    ** the application name in the interfaces file.
    */
    if((ssp = srv_init((SRV_CONFIG *)NULL, servername,
    CS_NULLTERM))== (SRV_SERVER *)NULL)
    {
        /*
        ** Release the context structure already allocated
        */
        (CS_VOID)cs_ctx_drop(cp);

        (CS_VOID)fprintf(stderr, "%s: srv_init failed¥n",
        servername);
        exit(1);
    }

    /*
    ** Start the Open Server application running.We don't
    ** install any event handlers in this simple example.  This
    ** causes Open Server to use the default event handlers.
    **
    ** The call to srv_run does not return until a fatal error is
    ** detected by this Open Server application, or a SRV_STOP
    ** event is queued.Since we haven't installed any event
    ** handlers, the only way to stop this Open Server
    ** application is to kill the operating system process in
    ** which it is running.
    */
    if(srv_run((SRV_SERVER *)NULL) == CS_FAIL)
    {
        (CS_VOID)fprintf(stderr, "%s: srv_run failed¥n",
        servername);
        ok = CS_FALSE;
    }

    /*
    ** Release all allocated control structures and exit.
    */
    (CS_VOID)srv_free(ssp);
    (CS_VOID)cs_ctx_drop(cp);
    exit(ok ? 0 : 1);
}
```

Open Server イベント

クライアントが Open Server アプリケーションに要求を送信すると、サーバ内の「イベント」がトリガされます。これによって、クライアントのサーバ・プロセス、つまり「スレッド」が、イベントを処理するルーチンを実行します。このルーチンを「イベント・ハンドラ」と呼びます。

Server-Library を使用して、さまざまな標準イベントが内部的に定義されます。次の表には、代表的な標準イベントを示します。

クライアント要求	イベント・タイプ	Open Server イベント
ct_command(CS_LANG_CMD) ct_send	言語	SRV_LANGUAGE
ct_command(CS_RPC_CMD) ct_send	RPC	SRV_RPC
ct_cancel	アテンション	SRV_ATTENTION
ct_connect	接続	SRV_CONNECT
ct_close ct_exit	切断	SRV_DISCONNECT
クライアントからは開始されない	起動	SRV_START
クライアントからは開始されない	停止	SRV_STOP

「イベント」(84 ページ) を参照してください。

デフォルトのイベント・ハンドラ

標準イベントにはデフォルトのイベント・ハンドラがありますが、通常は、ユーザ自身がコーディングしたイベント・ハンドラに置き換えます。ほとんどのデフォルトのイベント・ハンドラは、単に要求をそのまま返すだけです。たとえば、デフォルトの言語イベント・ハンドラは、次のメッセージを返します。

```
No language handler installed.
```

あるイベント・ハンドラをインストールすると、デフォルトのイベント・ハンドラは自動的に上書きされます。

クライアントでは開始されないイベント

次に示すイベントのように、クライアント・プログラムではトリガできないイベントもあります。

- ユーザ定義イベント
- Open Server のコードに srv_event を記述して呼び出すとトリガされる SRV_STOP
- 起動プロセスの一部として発生する SRV_START

レジスタード・プロシージャ

レジスタード・プロシージャとは、名前で識別できる Open Server/C コードの 1 つです。アプリケーションはプロシージャを登録するときに、プロシージャ名をルーチンにマップします。これにより、Open Server は、受信 RPC データ・ストリーム内でそのプロシージャ名を検出すると、SRV_RPC イベントを発生させずに特定のルーチンをただちに呼び出すことができます。

Open Server アプリケーションは RPC を受信すると、レジスタード・プロシージャのリスト内でそのプロシージャ名を検索します。プロシージャ名が登録されている場合、ランタイム・システムは、そのレジスタード・プロシージャと対応するルーチンを実行します。プロシージャ名がレジスタード・プロシージャのリストに見つからない場合には、Open Server は SRV_RPC イベント・ハンドラを呼び出します。

システム・レジスタード・プロシージャは、すべての Open Server アプリケーションに組み込まれているプロシージャです。個々のシステム・レジスタード・プロシージャの詳細については、「第 4 章 システム・レジスタード・プロシージャ」を参照してください。

レジスタード・プロシージャの詳細については、「レジスタード・プロシージャ」(151 ページ) を参照してください。

クライアントへの結果の送信

この項では、クライアントに対して送受信できる結果データの種類と順序について説明します。

結果データの種類

Open Server アプリケーションからクライアントに送信される結果は、次のとおりです。

- メッセージ
- データ・ロー
- 結果パラメータ
- ステータス値

1 つのクライアント要求には、2 つ以上の結果セットを指定できます。最初の結果セットを送信した後で、その要求に対してさらに結果セットがある場合には、SRV_DONE_MORE ステータスを使用して srv_senddone を呼び出します。それ以上結果がない場合は、SRV_DONE_FINAL ステータスを使用して srv_senddone を呼び出してください。SRV_DONE_FINAL ステータスを使用して srv_senddone を呼び出した場合は、クライアント要求への応答が最小限になります。クライアントは、srv_senddone (SRV_DONE_FINAL) を受信した後で処理を実行します。

メッセージ

アプリケーションは、srv_sendinfo でクライアントにエラー・メッセージを送信できます。Client-Library プログラムは、メッセージ・ハンドラ・ルーチンでメッセージを処理します。このルーチンは、通常、メッセージ情報をユーザの端末に表示します。メッセージの種類が「エラー・メッセージ」の場合、クライアント・プログラムは、エラーからリカバリを試みることも終了することもできます。

データ・ロー

Adaptive Server Enterprise が SQL クエリの結果を返すのと同様に、Open Server はデータ・ローをクライアントに返すことができます。ローは、1 つ以上のデータ・カラムで構成されています。詳細については、「パラメータとロー・データの処理」(126 ページ) を参照してください。

パラメータ

パラメータは、クライアントと Open Server アプリケーションとの間で、クライアント・コマンドを使用して通信されるデータです。

ステータス値

アプリケーションは srv_sendstatus を呼び出して、オプションのステータス値をクライアント・アプリケーションに返すことができます。ステータスは、アプリケーション固有の意味を持つ CS_INT 値です。CS_INT は、Open Server のデータ型です。「データ型」(187 ページ) を参照してください。1 つの結果セットには、ステータスは 1 つしかありません。

結果の順序

クライアントに結果を返す順序は重要です。

- データ・ローのセットには、他の種類の結果を割り込ませないでください。データ・ローは、データ・ロー全体がクライアントに送信されるまで、続けて送信する必要があります。たとえば、いくつかのデータ・ローを送信できない場合には、メッセージを送信してから、さらにローを送信します。
- 存在するすべてのデータ・ローについて送信を完了すると、メッセージやステータス情報を順序に関係なくクライアントに送信できます。
- 結果セットの最後に、結果が終わったことを伝える srv_senddone ルーチンを呼び出します。

エラー処理

Open Server アプリケーションで最初に実行する必要があるアクションの 1 つが、srv_props でエラー・ハンドラをインストールすることです。エラー・ハンドラがインストールされていない場合には、Open Server はエラー・メッセージをログ・ファイルに書き込みます。詳細については、「エラー」(81 ページ) を参照してください。

マルチスレッド・プログラミング

Open Server は、マルチスレッド・アーキテクチャを備えています。このアーキテクチャによって、アプリケーション開発者はマルチスレッド・サーバを作成することができます。マルチスレッド・サーバとは、それぞれが独自のタスクを達成するためにルーチンを実行するスレッドの集合です。たとえば、各クライアントは、その接続を管理するスレッドを使用し、要求を満たすイベント・ハンドラとプロシージャを実行します。Open Server ランタイム・システムは、メッセージの転送、ネットワーク通信処理、サーバ内のタスクのスケジューリングなどのサーバ・アクティビティを管理するためのスレッドをいくつか持っています。他の非クライアント・アクティビティのためにスレッドを「発生」させることができます。

詳細については、「マルチスレッド・プログラミング」(102 ページ) を参照してください。

第 2 章

トピック

この章では、次の項目について説明します。

- パラメータとロー・データの処理、text および image のサポートなどの Open Server のプログラミングについてのトピック
- カーソル要求への応答やエラー処理など、具体的なプログラミング・タスクの実行に Open Server のルーチンを使用する方法
- Open Server のプロパティ、データ型、構造体

この章の内容は、次のとおりです。

アテンション・イベント

クライアント・アプリケーションが dbcancel または ct_cancel コマンドを使用して要求をキャンセルすると、Open Server SRV_ATTENTION イベントがトリガされます。次に、Open Server によって、Open Server アプリケーションの SRV_ATTENTION イベント・ハンドラが呼び出されます。呼び出された SRV_ATTENTION イベント・ハンドラが返されると、Open Server は、アテンション・イベントが検出された時点で中断された位置から処理を再開します。

割り込みレベルのアクティビティ

SRV_ATTENTION イベント・ハンドラは、割り込みレベルで動作する唯一のイベント・ハンドラです。Open Server アプリケーションは、SRV_ATTENTION の中から次の Server-Library 呼び出しのみを発行できます。

- wakeflags 引数として SRV_M_WAKE_INTR を設定した srv_wakeup
- wakeflags 引数として SRV_M_WAKE_INTR を設定した srv_ucwakeup
- cmd 引数として CS_GET を設定した srv_thread_props
- cmd 引数として CS_GET を設定した srv_props
- srv_event_deferred

上記以外の Server-Library ルーチンの場合は、SRV_ATTENTION イベント・ハンドラや他の割り込みレベルのコードを使用して呼び出すことはできません。

アテンション・イベントに関するコーディングの考慮事項

アテンション・イベントが問題となるのは、非割り込みレベルのハンドラのコードを実行しているときにアテンション・イベントが到着する場合です。クライアントが要求をキャンセルしたために、アプリケーションが、必要のなくなった作業を行う可能性があります。

非割り込みレベルで、時間のかかる I/O タスクや計算を中心とした作業を実行している場合は、アプリケーションがアテンション・イベントを定期的にチェックする必要があります。アプリケーション・コードは、srv_thread_props を使用してアテンション・イベントを定期的にチェックする必要があります。このとき、cmd に CS_GET を、property に SRV_T_GOTATTENTION を設定します。

アテンション・イベントを検出すると、Open Server アプリケーション・コードは引き続き結果を送信できますが、クライアントはこれを無視します。アプリケーションがアテンション・イベントに応答する最も簡単な方法は、クライアントに SRV_DONE_FINAL を送信して戻ることです。

ゲートウェイ・アプリケーション・コードの中の Client-Library 部分が実行されているときに、アテンション・イベントが到着する可能性があります。アプリケーションは、SRV_ATTENTION イベント・ハンドラで type 引数に CS_CANCEL_ATTN を設定して ct_command を呼び出し、Client-Library ルーチンを非割り込みレベルのコードに戻します。アテンション・イベントが到着していない場合はこのコマンドは無効なので、ゲートウェイ・アプリケーションは、このコマンドを定期的に呼び出す必要があります。

クライアント I/O を行うすべてのゲートウェイ呼び出しは、srv_thread_props を使用してアテンション・イベントをチェックしてから、ct_send を呼び出します。このプロシージャによって、クライアントがキャンセルした「クエリ」がリモート・サーバに送信されることはありません。

切断の処理

Open Server アプリケーションがクライアントに結果を送信している途中で、クライアント接続が突然切断された場合、アプリケーションは、接続がクローズされたことを検出するまで引き続き結果を送信します。続いて、Open Server は SRV_DISCONNECT イベント・ハンドラを呼び出します。この場合は、すでに受信できない状態にあるクライアントに対して、アプリケーションが結果を送信し続けていることになります。次のような場合、クライアント接続が突然切断されることがあります。

- サーバから送信されるすべての結果をクライアントが処理する前に ct_close を呼び出した場合
- クライアントが処理を突然中断した場合
- マシンがダウンした場合

この事態を避けるために、アプリケーションは、クライアント接続の切断に対して Open Server が最初にアプリケーションの SRV_ATTENTION イベント・ハンドラを呼び出した後、SRV_DISCONNECT イベント・ハンドラを呼び出すように要求できます。Open Server がこのような形で切断を処理するには、アプリケーションは srv_props を使用して SRV_S_DISCONNECT プロパティを CS_TRUE に設定する必要があります。SRV_DISCONNECT イベント・ハンドラは通常どおりに呼び出されますが、SRV_ATTENTION ハンドラの後に呼び出されます。SRV_S_DISCONNECT プロパティのデフォルトは、CS_FALSE です。

SRV_ATTENTION ハンドラは、I/O アクティビティを終了したり、切断時に実行中だったルーチンからの結果の応答を停止したりするための、適切な手順を開始します。このように、アプリケーションは、アテンションに応答する場合と同様に切断にも応答することができます。

アプリケーションは、SRV_ATTENTION イベント・ハンドラを使用して、アテンションと接続の切断のどちらがハンドラをトリガしたのかを判断することもできます。そのためには、cmd を CS_GET に、property を SRV_S_ATTNREASON に設定して srv_props を呼び出します。

例

サンプル・プログラム *ctos.c* には、アテンション処理のコードが記述されています。

ブラウズ・モード

注意 ブラウズ・モードは、バージョン 11.1 以降の Open Client ライブラリとの互換性を実現するためのモードです。Sybase では、新しい Open Server Server-Library アプリケーションでブラウズ・モードを使用することはおすすめしません。ブラウズ・モードと同じ機能性を備えており、ブラウズ・モードよりポータブルで柔軟性のあるカーソルを使うことをおすすめします。さらに、ブラウズ・モードは Sybase 固有のものであり、異機種接続環境での使用には適していません。

ブラウズ・モードを使用すると、データベース・ローを検索し、その検索結果の値をロー単位で更新できます。ただし、その処理を実施するには、クライアント・アプリケーション内の処理としていくつかの手順が必要になります。その理由は、各ローをデータベースからクライアント・アプリケーション・プログラム用の変数に変換してからでないと、ローを検索したり更新したりできないからです。

ブラウズされているローはデータベースに常駐している実際のローではなく、プログラム変数に常駐しているコピーなので、プログラムは、変数の値になされた変更に基づいて元のデータベース・ローを更新しなければなりません。マルチユーザ環境の場合は、あるユーザがデータベースを変更しても、それ以前に別のユーザによって変更された内容が書き換えられることがないように、プログラムによる保証が必要です。このような書き換えが起こるのは、クライアント・アプリケーションはデータベース内の複数のローを一度に選択して更新するのに対して、アプリケーションのユーザはデータベース内のローを一度に 1 つずつ検索して更新するためです。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。

アドホックのブラウズ・モード・クエリの入力をユーザに許可するクライアント・アプリケーションでは、ユーザ・コマンドによってテーブルの内容が変更された場合は、基本データベース・テーブルを更新しなければなりません。そのため、これらのアプリケーションには、ブラウズ・モード・コマンドの基本構造体についての情報が必要な場合があります。

Open Server には、そのような情報を提供する 2 つのルーチン、srv_tabname と srv_tabcolname が用意されています。

- srv_tabname は、ブラウズ・モード・コマンドに含まれる各テーブルの名前と番号を返します。
- srv_tabcolname は、CS_BROWSEDESC 構造体を介して、結果カラムに関するさまざまな情報を返します。「CS_BROWSEDESC 構造体」(47 ページ)を参照してください。

Open Server アプリケーションがブラウズ・モード要求を受け取ると、標準のデータ・バインド・ルーチンに従ってブラウズ・モード情報を返すために、上記の 2 つのルーチンを呼び出すことができます。具体的な手順は、次のとおりです。

1　結果ローのソースとなるテーブルごとに、srv_tabname を 1 回ずつ呼び出します。

2　結果ロー内のカラムごとに、srv_descfmt と srv_tabcolname をこの順序で 1 回ずつ呼び出します。

Open Server アプリケーションで CS_BROWSEDESC 構造体の status フィールドが CS_RENAMED に設定されている場合は、クライアント・アプリケーションのブラウズ・モードの select 文によってカラム名が変更されています。Open Server アプリケーションでは、CS_BROWSEDESC 構造体の origname フィールドと orignlen フィールドにデータベース内のカラムのオリジナル名およびカラム名の長さを格納してから、srv_tabcolname を呼び出す必要があります。

3 カラム・データのバインドには srv_bind ルーチンを、転送には srv_xferdata ルーチンを使用します。

注意 srv_tabcolname には srv_tabname から返される情報 (ユニークなテーブル番号) が必要なため、srv_tabcolname を呼び出す前に srv_tabname を呼び出してください。

『Open Client Client-Library/C リファレンス・マニュアル』を参照してください。

例

サンプル・プログラム *ctos.c* には、ブラウズ・モード情報を処理するコードが記述されています。

機能

クライアントから発行可能な要求、および Open Server アプリケーションから返信可能な応答については、Open Server アプリケーションとクライアントがお互いに認識できる必要があります。たとえば、クライアントが言語要求を発行したくても、Open Server アプリケーションにはそのような要求を処理するパーサが提供されていない可能性があります。同様に、text データや image データをクライアントが処理できない場合は、クライアントは、Open Server アプリケーションからの text データや image データの送信を希望しない可能性があります。クライアント／サーバ接続の機能が、その接続で許可されるクライアントの要求とサーバ応答のタイプを決定します。

接続についてどの機能が有効であるかは、Open Server アプリケーションが最終的に決定します。有効と決定された機能がクライアントに受け入れられない場合は、接続をクローズするしかありません。

機能のネゴシエーションには、透過と明示の 2 つのタイプがあります。透過的ネゴシエーションの場合、Open Server アプリケーションは、可能なクライアント要求と Open Server 応答の「デフォルト・セット」を割り当てます。明示的ネゴシエーションの場合、Open Server アプリケーションには、srv_capability_info ルーチンを使って機能をネゴシエーションするためのコードが記述されます。

透過的ネゴシエーションは、Open Server と Open Client の双方のデフォルト動作の一部です。したがって、Open Server アプリケーションが機能のデフォルト・セット以外のものをサポートする場合は、srv_capability_info を呼び出す必要があります。

要求機能

表 2-1 は、要求機能の一覧を示します。

表 *2-1:* 要求機能

CS_REQUEST 機能	意味	関連する機能
CS_CAP_ EXTENDEDFAILOVER	拡張 HA フェールオーバ	接続
CS_CON_INBAND	バンド内 (非優先) アテンション	接続
CS_CON_OOB	バンド外 (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	先頭カーソル・ローのフェッチ	カーソル
CS_CSR_LAST	最終カーソル・ローのフェッチ	カーソル
CS_CSR_MULTI	複数ロー・カーソルのフェッチ	カーソル
CS_CSR_PREV	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_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 データ型	データ型
CS_DATA_LBIN	long 可変長 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 bit データ型	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 可変長 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 が許可される 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 データ型	データ型
CS_DATA_IMAGE	image データ型	データ型
CS_DATA_INT1	tinyint 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_INT8	big integer データ型	データ型
CS_DATA_INTN	NULL integer	データ型

CS_REQUEST 機能	意味	関連する機能
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	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_UCHAR	2 バイト character データ型	データ型
CS_DATA_UNITEXT	Unitext データ型	データ型
CS_DATA_XML	XML データ型	データ型
CS_OPTION_GET	現在のオプション値	データ型
CS_PROTO_DYNAMIC	TDS DESCIN/OUT プロトコルの使用	コマンド
CS_PROTO_DYNPROC	動的準備の前に “create proc” を追加	コマンド
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_LARGEIDENT	長い識別子の要求	コマンド
CS_REQ_MIGRATE	マイグレーション要求	接続
CS_REQ_MSG	メッセージ・データ	コマンド
CS_REQ_MSTMT	1 つの Client-Library 要求に対して複数のサーバ・コマンド	接続
CS_REQ_NOTIF	イベント通知	接続
CS_REQ_SRVPKTSIZE	サーバで指定された packetsize	接続
CS_REQ_PARAM	パラメータ・データ	コマンド
CS_REQ_RPC	リモート・プロシージャ要求	コマンド
CS_REQ_URGNOTIF	5.0 イベント通知プロトコルの使用	コマンド
CS_WIDETABLES	表ごとのカラムの拡大、数の増加	コマンド

応答機能

表 2-2 は、応答機能の一覧を示します。

注意　応答機能は、クライアントが受信したくない応答の種類を示します。

表 *2-2:* 応答機能

CS_RESPONSE 機能	意味	関連する機能
CS_CON_NOINBAND	バンド内 (非優先) アテンションなし	接続
CS_CON_NOOOB	バンド外 (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_NOVBIN	可変長 binary データ型なし	データ型
CS_DATA_NOLBIN	long 可変長 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_NOBOUNDARY	boundary 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 可変長 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	tinyint 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	big integer データ型なし	データ型
CS_DATA_NOINTN	NULL integer 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	sensitivity データ型なし	データ型
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	2 バイト character データ型なし	データ型

CS_RESPONSE 機能	意味	関連する機能
CS_DATA_NOUNITEXT	unitext データ型なし	データ型
CS_DATA_NOXML	XML データ型なし	データ型
CS_NO_SRVPKTSIZE	サーバで指定された packetsize 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	結果パラメータなし	結果
CS_RES_NOTDSDEBUG	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_NO_LARGEIDENT	長い識別子なし	コマンド
CS_NOWIDETABLES	表ごとのカラムサイズの拡大、カラム数の増加なし	コマンド

注意 Open Server アプリケーションが、srv_descfmt ルーチンを使用してクライアントのデータ・フォーマットを定義するときに、Open Server は、関連するデータ型の応答機能が設定されていないことを確認します。その応答機能が設定されている場合は、クライアントはサーバがそのデータ型に関係する結果を送らないことを要求しているか、またはクライアント接続の TDS バージョンがそのデータ型をサポートしていないかのどちらかです。その場合、Open Server はエラーを起こし、srv_descfmt は CS_FAIL を返します。

透過的ネゴシエーション

Open Server には、機能のデフォルト値が設定されています。デフォルト値のリストについては、「サーバワイドなデフォルト」(27 ページ) を参照してください。これらのデフォルトはサーバ全体にわたるものであり、すべてのクライアント接続に適用されます。デフォルトが使用されると、Open Server がサポートする機能がすべてオンになります。

初期化中にサーバワイドなデフォルト値を変更するには、Open Server アプリケーションで srv_props ルーチンを呼び出します。srv_props (313 ページ) を参照してください。

DB-Library クライアントまたは Client-Library クライアントが Open Server アプリケーションにログインするときには、そのクライアントは希望する機能のリストをログイン・レコードの中に含めて送信します。透過的ネゴシエーションにおいては、Open Server はデフォルト値とクライアント値の共通部分を見つけます。その結果生じる値が、その接続でサポートされる値となります。

透過的ネゴシエーションが発生する場合

透過的ネゴシエーションは、次の場合に発生します。

- Open Server アプリケーションに、デフォルト・ハンドラ以外の SRV_CONNECT ハンドラがない場合。
- Open Server アプリケーションのカスタムの SRV_CONNECT イベント・ハンドラ内に、デフォルト機能を無効にするようなコードが明示的に記述されていない場合。

注意　パススルー・モードでは、srv_getloginfo と srv_setloginfo によって機能ネゴシエーションが透過的に処理されます。

サーバワイドなデフォルト

表 2-3 は、TDS のバージョン別に各要求機能のデフォルト設定をまとめたものです。*1* は、TDS バージョンで機能がサポートされていることを示します。*0* は、機能がサポートされていないことを示します。

表 *2-3: TDS* バージョン別の要求機能

CS_REQUEST 機能	4.0	4.0.2	4.2	4.6	5.0
CS_CAP_EXTENDEDFAILOVER	0	0	0	0	1
CS_CON_INBAND	0	0	0	0	1
CS_CON_OOB	1	1	1	1	0
CS_CSR_ABS	0	0	0	0	0
CS_CSR_FIRST	0	0	0	0	0
CS_CSR_LAST	0	0	0	0	0
CS_CSR_MULTI	0	0	0	0	0
CS_CSR_PREV	0	0	0	0	0
CS_CSR_REL	0	0	0	0	0
CS_DATA_BIGDATETIME	0	0	0	0	1
CS_DATA_BIGTIME	0	0	0	0	1
CS_DATA_BIN	1	1	1	1	1
CS_DATA_BIT	1	1	1	1	1
CS_DATA_BITN	0	0	0	0	0
CS_DATA_BOUNDARY	0	0	0	0	0
CS_DATA_CHAR	1	1	1	1	1
CS_DATA_DATE	0	0	0	0	1
CS_DATA_DATE4	0	0	1	1	1
CS_DATA_DATE8	1	1	1	1	1
CS_DATA_DATETIME	1	1	1	1	1
CS_DATA_DEC	0	0	0	0	0

CS_REQUEST 機能	4.0	4.0.2	4.2	4.6	5.0
CS_DATA_FLT4	0	0	1	1	1
CS_DATA_FLT8	1	1	1	1	1
CS_DATA_FLTN	1	1	1	1	1
CS_DATA_IMAGE	1	1	1	1	1
CS_DATA_INT1	1	1	1	1	1
CS_DATA_INT2	1	1	1	1	1
CS_DATA_INT4	1	1	1	1	1
CS_DATA_INT8	0	0	0	0	1
CS_DATA_INTN	1	1	1	1	1
CS_DATA_LBIN	0	0	0	0	0
CS_DATA_LCHAR	0	0	0	0	0
CS_DATA_MNY4	0	0	1	1	1
CS_DATA_MNY8	1	1	1	1	1
CS_DATA_MONEYN	1	1	1	1	1
CS_DATA_NUM	0	0	0	0	0
CS_DATA_SENSITIVITY	0	0	0	0	0
CS_DATA_TEXT	1	1	1	1	1
CS_DATA_TIME	0	0	0	0	1
CS_DATA_UCHAR	0	0	0	0	1
CS_DATA_UNITEXT	0	0	0	0	1
CS_DATA_VBIN	1	1	1	1	1
CS_DATA_VCHAR	1	1	1	1	1
CS_DATA_XML	0	0	0	0	1
CS_OPTION_GET	0	0	0	0	0
CS_PROTO_DYNAMIC	0	0	0	0	0
CS_PROTO_DYNPROC	0	0	0	0	0
CS_REQ_BCP	1	1	1	1	1
CS_REQ_CURSOR	0	0	0	0	0
CS_REQ_DBRPC2	0	0	0	0	1
CS_REQ_DYN	0	0	0	0	0
CS_REQ_LANG	1	1	1	1	1
CS_REQ_LARGEIDENT	0	0	0	0	1
CS_REQ_MIGRATE	0	0	0	0	1
CS_REQ_MSG	0	0	0	0	0
CS_REQ_MSTMT	0	0	0	0	0
CS_REQ_NOTIF	0	0	0	1	1
CS_REQ_PARAM	0	0	0	0	0
CS_REQ_RPC	1	1	1	1	1
CS_REQ_SRVPKTSIZE	0	0	0	0	1
CS_REQ_URGNOTIF	0	0	0	0	0
CS_WIDETABLES	0	0	0	0	1

表 2-4 は、TDS のバージョン別の各応答機能のデフォルト設定をまとめたものです。

- *1* は、TDS バージョンで機能がサポートされていないことを示します。
- *0* は、機能がサポートされていることを示します。

表 *2-4: TDS* バージョン別の応答機能

CS_RESPONSE 機能	4.0	4.0.2	4.2	4.6	5.0
CS_CON_NOINBAND	1	1	1	1	1
CS_CON_NOOOB	0	0	0	0	0
CS_DATA_NOBIGDATETIME	1	1	1	1	0
CS_DATA_NOBIGTIME	1	1	1	1	0
CS_DATA_NOBIN	0	0	0	0	0
CS_DATA_NOBIT	0	0	0	0	0
CS_DATA_NOBOUNDARY	1	1	1	1	1
CS_DATA_NOCHAR	0	0	0	0	0
CS_DATA_NODATE4	1	1	0	0	0
CS_DATA_NODATE8	0	0	0	0	0
CS_DATA_NODATETIME	0	0	0	0	0
CS_DATA_NODEC	1	1	1	1	1
CS_DATA_NOFLT4	1	1	0	0	0
CS_DATA_NOFLT8	0	0	0	0	0
CS_DATA_NOIMAGE	0	0	0	0	0
CS_DATA_NOINT1	0	0	0	0	0
CS_DATA_NOINT2	0	0	0	0	0
CS_DATA_NOINT4	0	0	0	0	0
CS_DATA_NOINT8	1	1	1	1	0
CS_DATA_NOINTN	0	0	0	0	0
CS_DATA_NOLBIN	1	1	1	1	1
CS_DATA_NOLCHAR	1	1	1	1	1
CS_DATA_NOMNY4	1	1	0	0	0
CS_DATA_NOMNY8	0	0	0	0	0
CS_DATA_NOMONEY	0	0	0	0	0
CS_DATA_NONUM	1	1	1	1	1
CS_DATA_NOSENSITIVITY	1	1	1	1	1
CS_DATA_NOSINT1	1	1	1	1	0
CS_DATA_NOTEXT	0	0	0	0	0
CS_DATA_NOUCHAR	1	1	1	1	0
CS_DATA_NOUNITEXT	1	1	1	1	0
CS_DATA_NOVBIN	0	0	0	0	0
CS_DATA_NOVCHAR	0	0	0	0	0
CS_DATA_NOXML	1	1	1	1	0

CS_RESPONSE 機能	4.0	4.0.2	4.2	4.6	5.0
CS_RES_NOEED	1	1	1	1	1
CS_RES_NOMSG	1	1	1	1	1
CS_RES_NOPARAM	1	1	1	1	1
CS_RES_NOTDSDEBUG	1	1	1	1	1
CS_RES_NOXNLMETADATA	1	1	1	1	0
CS_NO_LARGEIDENT	1	1	1	1	0
CS_NO_SRVPKTSIZE	1	1	1	1	0
CS_NOWIDETABLES	1	1	1	1	0

明示的ネゴシエーション

明示的ネゴシエーションは、接続時に SRV_CONNECT イベント・ハンドラ内部から開始されます。Open Server アプリケーションは、クライアントが送信する要求機能のリストを取得して、受け取る要求機能のリストを返します。今度は、クライアントが受け取りたくない応答機能、または、Open Server アプリケーションが返すことのできない応答機能を返すプロセスが繰り返されます。

アプリケーションは、一度に 1 つずつ機能を取得して送信したり、機能を取得して機能のビットマスク全体をただちに送信したりすることができます。Open Server 提供のマクロを使用して、機能マスク内のビットをテスト、クリア、設定できます。「機能マクロ」(31 ページ) を参照してください。

一度に 1 つずつの機能のネゴシエート

要求機能を一度に 1 つずつネゴシエートするには、ネゴシエートする機能ごとに次の呼び出しを行う必要があります。

1 cmd 引数を CS_GET に、type 引数を CS_CAP_REQUEST に、capability 引数を目的の機能に設定して srv_capability_info を呼び出します。*valp 引数に CS_TRUE が含まれている場合は、クライアントは、このようなタイプの機能を要求します。*valp 引数に CS_FALSE が含まれている場合は、クライアントはこのような機能を要求しません。

2 cmd 引数を CS_SET に、type 引数を CS_CAP_REQUEST に、capability 引数を目的の機能に、*valp をブール値に設定して srv_capability_info を呼び出します。アプリケーションがこのタイプの機能をサポートする場合は *valp を CS_TRUE に設定し、サポートしない場合は CS_FALSE に設定します。

アプリケーションは、type 引数を CS_CAP_RESPONSE に設定しなければならないこと以外は、同様の方法で応答機能をネゴシエートします。

Open Server アプリケーションは、明示的にネゴシエートする要求および応答機能についてだけ srv_capability_info を呼び出す必要があります。その他の機能には、すべてデフォルト値が使用されます。

機能ビットマスクを使用したネゴシエート

機能ビットマスクを使用して要求機能をネゴシエートするには、アプリケーションは次の処理が必要です。

1　cmd 引数を CS_GET に、type 引数を CS_CAP_REQUEST に、capability 引数を CS_ALL_CAPS に設定し、valp がビットマスクを含む CS_CAP_TYPE 構造体を指すようにして srv_capability_info を呼び出し、ビットマスク全体を読み込みます。

2　ビットマスク内の特定のビットをテスト、設定、またはクリアするには、CS_TST_CAPMASK、CS_SET_CAPMASK、CS_CLR_CAPMASK の各マクロを使用します。

アプリケーションは、type 引数を CS_CAP_RESPONSE に設定しなければならないこと以外は、同様の方法で応答機能をネゴシエートします。

ゲートウェイ・アプリケーションは、機能のネゴシエーションを行うためには、マスク方式を使用します。次の図に示されているように、ゲートウェイは、srv_capability_info を呼び出してリモート・クライアントの機能マスクを取得し、ct_capability を呼び出してそれらの機能をリモート・サーバに送信してから、ct_connect を呼び出します。リモート接続が確立すると、ゲートウェイは、リモート・サーバが ct_capability を使用して送信した機能マスクを取得し、srv_capability_info を使用して、リモート・クライアントの接続時にそれらの機能マスクを定義できます。

機能マクロ

表 2-5 は、アプリケーションが機能ビットマスクを操作するために使用するマクロについて説明しています。

表 *2-5:* 機能マクロ

マクロ名	関数
CS_TST_CAPMASK	特定の機能が CS_TRUE と CS_FALSE のどちらに設定されているかをテストする。
CS_SET_CAPMASK	特定の機能を CS_TRUE に設定する。
CS_CLR_CAPMASK	特定の機能を CS_FALSE に設定する。

機能を明示的に扱う場合、デフォルト設定を使用する代わりに、次の 2 つの規則を適用します。

- CS_CAP_REQUEST
  アプリケーションでは、CS_CAP_REQUEST 機能を “on” ステータスから “off” ステータスに切り換える操作だけが可能です。

  アプリケーションが CS_CAP_REQUEST 機能を “off” にしようとしたときに、すでに “off” ステータスになっている場合、Open Server はデフォルトのステータスをリストアして、エラーが発生しないようにします。

- CS_CAP_RESPONSE
  アプリケーションでは、CS_CAP_RESPONSE 機能を “off” ステータスから “on” ステータスに切り換える操作だけが可能です。

  アプリケーションが CS_CAP_RESPONSE 機能を “on” にしようとしたときに、すでに “on” ステータスになっている場合、Open Server はデフォルトのステータスをリストアして、エラーが発生しないようにします。

機能情報の特定の検索

Open Server アプリケーションは、特定のクライアント接続に対して有効な機能リストを取得するために、いつでも、どのハンドラの内部からでも srv_capability_info を呼び出すことができます。しかし、SRV_CONNECT イベント・ハンドラにおいては、取得された機能マスクはその接続の最終的なマスクとはならないことに注意してください。これらのマスクは、Open Server アプリケーションのデフォルトと結合したクライアントが要求した機能のものです。SRV_CONNECT ハンドラが返されるまで、接続機能が最終のものとはなりません。

バージョン 10.0 以前のクライアントに関する注意

Open Server アプリケーションは、どのような TDS バージョンを実行しているクライアントとでも機能をネゴシエートできます。10.0 以前のクライアントを接続する場合、Open Server は、機能ネゴシエーションのシミュレートを行います。この場合に、Open Server アプリケーションは、クライアントが実行している TDS バージョンを認識する必要はありません。

例

オンラインのサンプル・プログラム *ctos.c* には、機能ネゴシエーションのコードが記述されています。

クライアント・コマンド・エラー

クライアントは、不完全または意味のない要求を Open Server アプリケーションに送ることがあります。クライアントのコードに誤りがあったりネットワークに問題があったりすると、要求は不完全になったり無意味になったりします。Open Server アプリケーションは、クライアント要求に対するイベント・ハンドラで、適切なエラー・メッセージをクライアントに送信して、これらのエラーを処理します。

srv_sendinfo によるメッセージの送信

Open Server アプリケーションは srv_sendinfo を呼び出し、エラー・メッセージをクライアントに送ります。Open Server アプリケーションは、CS_SERVERMSG 構造体にメッセージを記述し、srv_sendinfo を呼び出して、この記述をクライアントに送ります。

「CS_SERVERMSG 構造体」(55 ページ) を参照してください。

長いメッセージの連続化

Open Server アプリケーションは、メッセージ・テキストを CS_SERVERMSG 構造体の text フィールドに格納します。text の最大長は CS_MAX_MSG バイトです。

Open Server アプリケーションは、メッセージ・テキスト全体を送るために必要な数だけ CS_SERVERMSG 構造体を使用します。アプリケーションは最初の構造体内の CS_MAX_MSG バイトだけ返し、2 番目の構造体内の 2 番目の CS_MAX_MSG バイトを返します。以下同様にこの処理を繰り返します。この処理はメッセージの「連続化」と呼ばれます。

アプリケーションは、「連続化した」数と同じ数だけ srv_sendinfo を呼び出します。メッセージ全体が 1 つの構造体に収まる場合、アプリケーションは srv_sendinfo を一度だけ呼び出します。

連続化されたメッセージに対する CS_SERVERMSG 構造体のフィールド

CS_SERVERMSG 構造体の status フィールドは、構造体がメッセージ全体を含むのかメッセージのまとまり (部分) を含むのかを示します。

表 2-6 は、連続化されたメッセージに関連する status 値のリストです。

表 *2-6:* 連続化されたメッセージの *status* 値

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。 構造体の中のメッセージ・テキストがメッセージ全体の場合、アプリケーションは CS_FIRST_CHUNK と CS_LAST_CHUNK の両方を設定する。 構造体の中のメッセージ・テキストが中間のテキストの場合、アプリケーションは CS_FIRST_CHUNK と CS_LAST_CHUNK のどちらも設定しない。

CS_SERVERMSG 構造体の中の textlen フィールドは、常に現在のメッセージのまとまりの長さを反映します。

CS_SERVERMSG 構造体の他のフィールドは、各メッセージのまとまりごとに繰り返されます。

拡張エラー・データ

サーバ・メッセージは、関連する「拡張エラー・データ」を含んでいる場合があります。拡張エラー・データは、エラーに関する追加情報です。

Adaptive Server Enterprise メッセージの場合、通常、追加情報にはエラーを引き起こしたカラムが記述されています。

拡張エラー・データの利点

ユーザがデータを入力または編集できるクライアント・アプリケーションは、カラム・レベルでユーザにエラーをレポートする必要があります。しかし、標準サーバ・メッセージ・メカニズムでは、カラム・レベルの情報はサーバ・メッセージのテキスト内でしか使用できません。拡張エラー・データは、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、ユーザが pubs2 データベースの titleauthor テーブルにデータを入力および編集できるクライアント・アプリケーションがあるとします。titleauthor は、2 つのカラム au_id と title_id からなる「キー」を使用します。既存のローに合致する au_id と title_id を含むローを入力しようとすると、「重複キー」メッセージがクライアント・アプリケーションに送信されます。

このメッセージを受信したクライアント・アプリケーションは、ユーザが修正できるよう、問題のカラムをユーザに指示する必要があります。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・データのクライアントへの送信

メッセージで拡張エラー・データが使用可能な場合、Open Server アプリケーションは CS_HASEED 構造体の status フィールドを CS_SERVERMSG に設定します。

Open Server アプリケーションは、srv_sendinfo ルーチンのパラメータとして拡張エラー・データを送ります。アプリケーションは、srv_descfmt、srv_bind、srv_xferdata を使用してエラー・パラメータの記述、バインド、送信を行います。

アプリケーションは、srv_sendinfo を呼び出した直後に、エラー・パラメータを記述、バインド、送信してから、他の結果を返したり srv_senddone を呼び出したりします。アプリケーションは、type 引数を SRV_ERRORDATA に設定して、srv_descfmt、srv_bind、srv_xferdata を呼び出さなければなりません。

アプリケーションが CS_SERVERMSG 構造体の status フィールドを CS_HASEED に設定して srv_sendinfo を呼び出したがエラー・パラメータの送信に失敗した場合、Open Server は、アプリケーションが srv_senddone を呼び出したときに致命的なプロセス・エラーを起こします。

接続マイグレーション

接続マイグレーションにより、Open Server アプリケーションはその負荷を動的に分散し、透過的フェールオーバをサポートできます。また、異なる関数を実行する複数の Open Server アプリケーションがある場合、クライアントの要求を満たすことができる Open Server にクライアントをリダイレクトできます。

以下で説明するアプリケーション・プログラミング・インタフェース (API) を使用すると、Open Server はマイグレーション要求を開始、完了、およびキャンセルし、クライアントからのマイグレーション・メッセージに反応することができます。さらに、新しい接続がマイグレーション接続であるかどうかを検出して、接続からユニークな識別子を取得することもできます。

バッチ内マイグレーションとアイドル・マイグレーション

バッチ内マイグレーションでは、クライアントはオリジナル・サーバから結果を待機中にマイグレートします。反対に、アイドル・マイグレーションでは、クライアントはオリジナル・サーバから結果を待機しません。

バッチ内マイグレーションにより、Open Server は接続のマイグレーション後まで、結果の送信または完了を遅延できるようになります。これは、Open Server が特定の要求を処理できない場合や、要求を完了する時間がない場合に役立ちます。バッチ内マイグレーションでは、Open Server はオリジナル・サーバーからの結果の一部を送信でき、マイグレーション後にクライアントのマイグレーション先のサーバが SRV_MIGRATE_RESUME イベント・ハンドラからの結果の残りを送信できます。

注意　新しいサーバが結果をまったく送信しない場合、オリジナル・サーバが結果全体をクライアントに送信できます。同様に、新しいサーバが結果全体をクライアントに送信する必要がある場合、オリジナル・サーバが結果をクライアントにまったく送信しないことがあります。

バッチ内マイグレーションでは、未送信のコマンドおよびメッセージがクライアント・コンテキストの一部であることをアプリケーションが確認する必要があります。新しいサーバは、コマンドの影響を受けるローの数と、接続のトランザクション・ステータスにアクセスする必要もあります。新しいサーバは、srv_senddone() を使用してこの情報をクライアントに送信します。

コンテキスト・マイグレーション

Open Server は、クライアントの接続のシームレスなマイグレーションをサポートします。ただし、クライアントのコンテキストの共有およびマイグレーションはアプリケーションが行います。共有ファイルの使用や、ネットワーク通信など、コンテキスト・マイグレーションはさまざまな方法で実装できます。

バッチ内マイグレーションの場合、クライアントのマイグレーション先のサーバはオリジナル・サーバ内で発生したイベントのタイプを認識しません。アプリケーションがこの情報を必要とする場合、クライアントのコンテキストの一部として情報をマイグレートする必要があります。

アイドル・マイグレーションでは、クライアントは Open Server からの実際の結果を待機しません。アクティブなクエリがないため、アイドル・マイグレーションはバッチ内マイグレーションより簡単に実装できます。ただし、アイドル・マイグレーションでも、クライアントがマイグレーションを開始する前に発生した保留中の要求をすべてアプリケーションが満たす必要があります。

接続マイグレーションで使用される API

この項では、接続マイグレーションをサポートする API について説明します。「クライアントに対する異なるサーバへのマイグレーションの指示」(42 ページ)を参照してください。

CS_REQ_MIGRATE

CS_REQ_MIGRATE 要求機能は、クライアントがマイグレーション・プロトコルをサポートするかどうかと、クライアントが要求時に別のサーバへのマイグレーションを実行できるかどうかを示します。srv_capability_info() を使用すると、CS_REQ_MIGRATE 機能情報を取得できます。次に例を示します。

```
CS_RETCODE ret;
CS_BOOL migratable;
ret = srv_capability_info(sp, CS_GET, CS_CAP_REQUEST,
   CS_REQ_MIGRATE, &migratable);
```

SRV_CTL_MIGRATE

SRV_CTL_MIGRATE は、srv_send_ctlinfo() 制御タイプです。クライアントがマイグレーションをサポートし、セッションへの最初の接続時にセッション ID を受信した場合、SRV_CTL_MIGRATE を使用すると、マイグレーション要求をクライアントに送信したり、以前のマイグレーション要求をキャンセルすることができます。

クライアント・マイグレーションの要求

次のサンプル・コードは、サーバ “target” にマイグレートする要求をクライアントに送信します。

```
CS_RETCODE ret;
SRV_CTLITEM *srvitems;
CS_CHAR *target;
/*
** request a migration to server 'target'
*/
srvitems = (SRV_CTLITEM *) srv_alloc(sizeof
   (SRV_CTLITEM));
srvitems[0].srv_ctlitemtype = SRV_CT_SERVERNAME;
srvitems[0].srv_ctllength = strlen(target);
srvitems[0].srv_ctlptr = target;
ret = srv_send_ctlinfo(sp, SRV_CTL_MIGRATE, 1,
   srvitems);
srv_free(srvitems);
```

アプリケーションは、マイグレーションがすでに要求されている場合でも、SRV_CTL_MIGRATE 制御タイプを送信できます。Open Server は、以前のマイグレーション要求をキャンセルし、新しい要求をクライアントに送信します。新しいマイグレーション要求の戻り値は、次のとおりです。

戻り値	説明
CS_SUCCEED	マイグレーション要求が正常に送信されました。
CS_FAIL	次のいずれかの理由のため、マイグレーション要求に失敗しました。 • Open Server スレッドが接続マイグレーションをサポートしていない。 • 以前のマイグレーション要求が送信され、クライアントが新しいサーバへのマイグレーションを開始した。

マイグレーションのキャンセル

SRV_CTL_MIGRATE 制御タイプを使用して、以前のマイグレーション要求をキャンセルすることもできます。この場合、*paramcnt* を 0 にし、*param* を NULL ポインタにする必要があります。次に例を示します。

```
ret = srv_send_ctlinfo(sp, SRV_CTL_MIGRATE, 0, NULL);
if (ret != CS_SUCCEED)
{
...
}
```

SRV_CTL_MIGRATE は、Open Server アプリケーション内のどのスレッドでも使用できます。ただし、クライアント・スレッドの接続のマイグレーションをキャンセルするスレッドの要件は、自身の接続マイグレーションをキャンセルするクライアント・スレッドの要件とは異なります。

- どの Open Server スレッドでもマイグレーションをキャンセルできますが、SRV_MIGRATE_STATE イベント・ハンドラが、クライアントでマイグレーションの準備ができたことをクライアント・スレッドに知らせる前に、キャンセルが要求される必要があります。
- クライアント・スレッドは、SRV_MIGRATE_STATE イベント・ハンドラ内でもマイグレーションをキャンセルできます。ただし、クライアント・スレッドは、SRV_MIG_READY ステータスの SRV_MIGRATE_STATE イベントを終了した後は、マイグレーションをキャンセルできません。

マイグレーションのキャンセルの戻り値は次のとおりです。

戻り値	説明
CS_SUCCEED	マイグレーション要求が正常にキャンセルされました。
CS_FAIL	次のいずれかの理由のため、マイグレーションのキャンセルに失敗しました。 • 進行中のマイグレーションがない。 • クライアントが新しいサーバへのマイグレーションを開始した。

注意 マイグレーション要求が正常にキャンセルされると、Open Server は新しいマイグレーション・ステータス・イベントをトリガしません。

SRV_MIGRATE_RESUME

クライアントが結果の待機中に新しいサーバにマイグレートすると、クライアント接続が正常にマイグレートされた後、新しいサーバは SRV_MIGRATE_RESUME イベントを呼び出します。マイグレーション要求が失敗した場合やキャンセルされた場合、イベントはオリジナル・サーバから呼び出されます。

SRV_MIGRATE_RESUME イベント・ハンドラでは、アプリケーションは実際の結果をクライアントに送信する必要はありません。ただし、必ず送信する必要がある SRV_DONE_FINAL 結果タイプは除きます。デフォルトの SRV_MIGRATE_RESUME がクライアントに送信する結果は、SRV_DONE_FINAL だけです。

SRV_MIGRATE_RESUME イベント・ハンドラの例を次に示します。

```
/*
** Simple migrate_resume event handler.
*/
CS_RETCODE CS_PUBLIC
migrate_resume_handler(SRV_PROC *sp)
{
   CS_RETCODE ret;
   ret = srv_senddone(sp, SRV_DONE_FINAL,
      CS_TRAN_COMPLETED, 0);
   if (ret == CS_FAIL)
   {

      ...

   }
   return CS_SUCCEED;
}

...

/*
** Install the migrate-resume event handler
*/
srv_handle(server, SRV_MIGRATE_RESUME,
      migrate_resume_handler);
...
```

SRV_MIGRATE_STATE

SRV_MIGRATE_STATE は、マイグレーション・ステータスが SRV_MIG_READY または SRV_MIG_FAILED に移行すると必ずトリガされるイベントです。移行は、クライアントからマイグレーション・メッセージが送信された結果です。SRV_MIGRATE_STATE イベント・ハンドラは、次のような状況で呼び出されます。

SRV_T_MIGRATE_STATE	状況	発生する可能性があるアプリケーションの動作
SRV_MIG_READY	クライアントが、要求を検出し、マイグレーションの準備ができたことを示すメッセージをサーバに送信した。サーバは、マイグレーションを続行するかどうかを判断する。	次のうちのいずれか • コンテキストを他のサーバが使用できるようにする。 • マイグレーションが必要なくなったとアプリケーションが判断した場合は、マイグレーションをキャンセルする。 • 新しいマイグレーション・ターゲットが選択されている場合は、別のマイグレーションを要求する。

SRV_T_MIGRATE_STATE	状況	発生する可能性があるアプリケーションの動作
SRV_MIG_FAILED	クライアントが、マイグレーションが失敗したことを示すメッセージをサーバに送信した。	次のうちのいずれか • クライアント・コンテキストにアクセスし、接続の提供を続行する。 • 別のマイグレーションを要求する。

SRV_MIGRATE_STATE イベント・ハンドラの例を次に示します。

```
/*
** Simple migrate-state event handler
*/
CS_RETCODE CS_PUBLIC
migrate_state_handler(SRV_PROC *sp)
{
   SRV_MIG_STATE migration_state;
   ret = srv_thread_props(sp, CS_GET,
      SRV_T_MIGRATE_STATE, &migration_state,
      sizeof (migration_state), NULL);

   ...

   switch(migration_state)
   {
      case SRV_MIG_READY:
           ...
      case SRV_MIG_FAILED:
           ...
   }
}

...

/*
** Install the migrate-state change event handler
*/
srv_handle(server, SRV_MIGRATE_STATE, migrate_state_handler);
...
```

SRV_MIGRATE_STATE イベント・ハンドラを使用するときは、次のことが当てはまります。

- クライアント・スレッドが SRV_MIGRATE_STATE イベント・ハンドラ内からマイグレーションをキャンセルした場合、アプリケーションはコンテキストの一貫性を保つ必要があります。たとえば、アプリケーションが作成したコンテキストを別のサーバが使用することはありません。
- 新しいマイグレーション要求が SRV_MIGRATE_STATE イベント・ハンドラ内から送信された場合、このハンドラはクライアントが新しく要求されたマイグレーションを開始する準備ができたときに再度呼び出されます。

SRV_T_MIGRATE_STATE プロパティと SRV_MIG_STATE 列挙型

SRV_T_MIGRATE_STATE は、クライアントのマイグレーション・ステータスを示します。SRV_T_MIGRATE_STATE は、どのスレッドでもアクセスできる読み取り専用プロパティです。示されるマイグレーション・ステータスは次のとおりです。

ステータス	値	説明
SRV_MIG_NONE	0	進行中のマイグレーションがない。
SRV_MIG_REQUESTED	1	マイグレーションがサーバにより要求された。
SRV_MIG_READY	2	クライアントが要求を受信し、マイグレーションの準備ができた。
SRV_MIG_MIGRATING	3	クライアントが指定されたサーバにマイグレート中である。
SRV_MIG_CANCELLED	4	マイグレーション要求がキャンセルされた。
SRV_MIG_FAILED	5	クライアントがマイグレーションに失敗した。

SRV_MIG_STATE は、SRV_T_MIGRATE_STATE プロパティをモデル化する列挙データ型です。次のように SRV_MIG_STATE を宣言します。

```
typedef enum
{
   SRV_MIG_NONE,
   SRV_MIG_REQUESTED,
   SRV_MIG_READY,
   SRV_MIG_MIGRATING,
   SRV_MIG_CANCELLED,
   SRV_MIG_FAILED
} SRV_MIG_STATE;
```

次のサンプル・コードは、SRV_T_MIGRATE_STATE 値を取得する方法を示しています。マイグレーションに成功した場合、クライアントが終了し、SRV_DISCONNECT イベント・ハンドラが SRV_MIG_MIGRATING ステータスで呼び出されます。

```
CS_RETCODE ret;
SRV_MIG_STATE migration_state;
ret = srv_thread_props(sp, CS_GET, SRV_T_MIGRATE_STATE,
   &migration_state, sizeof (migration_state), NULL);
if (ret != CS_SUCCEED)
{
...
}
```

SRV_T_MIGRATED

SRV_T_MIGRATED は、接続が新しい接続であるか、マイグレートされた接続であるかを示す Boolean プロパティです。この読み込み専用プロパティは、クライアントがマイグレート中であるか、サーバにマイグレートした場合は true に設定されます。次のサンプル・コードは、SRV_T_MIGRATED の値を取得します。

```
CS_RETCODE ret;
CS_BOOL migrated;
status = srv_thread_props(sp, CS_GET, SRV_T_MIGRATED,
   &migrated, sizeof (migrated), NULL);
```

SRV_T_SESSIONID

SRV_T_SESSIONID は、クライアントから Open Server に送信されたセッション ID を取得するスレッド・プロパティです。次の場合は、srv_thread_props() 関数を使用して SRV_T_SESSIONID プロパティを設定できます。

- srv_thread_props (CS_SET, SRV_T_SESSIONID) 呼び出しが SRV_CONNECT イベント・ハンドラ内で行われた。
- クライアントで接続マイグレーションまたは高可用性がサポートされている。

次のサンプル・コードは、SRV_T_SESSIONID プロパティを設定します。

```
CS_RETCODE ret;
CS_SESSIONID hasessionid;
ret = srv_thread_props(sp, CS_SET, SRV_T_SESSIONID,
   hasessionid, sizeof(hasessionid), NULL);
```

注意 HA フェールオーバの場合、セッション ID がクライアントに送信されるように srv_negotiate() シーケンスをプログラミングする必要があります。

クライアントに対する異なるサーバへのマイグレーションの指示

この項では、他のサーバにクライアントをマイグレートするための Open Server の要件について説明します。クライアントを異なるサーバにマイグレートするとき、アプリケーションは次の処理を実行する必要があります。

1 ユニーク・セッション ID を作成し、それを接続ハンドラでクライアントに送信します。

2 接続マイグレーションを開始します。

3 マイグレーション・イベントを処理します。

4　接続のセッション ID を使用して、他のサーバへの接続のコンテキストを共有します。

5　(オプション) 既存のハンドラで進行中のマイグレーションを操作します。

これ以降は、これらのアクティビティについて詳細に説明します。

クライアントへのマイグレーションの要求

Open Server は、srv_send_ctlinfo() を使用してマイグレーション要求をクライアントに送信できます。クライアント・マイグレーションは、どの Open Server スレッドからでも要求できます。

接続 (SRV_CONNECT) イベントのマイグレーション

SRV_CONNECT イベント・ハンドラでは、アプリケーションは次の処理を実行する必要があります。

- SRV_T_MIGRATED プロパティを確認し、接続がマイグレートされた接続であるかどうかを判断します。マイグレートされた接続である場合、アプリケーションはクライアントにより提供されるセッション ID に基づいてコンテキストにアクセスする必要があります。セッション ID は、SRV_T_SESSIONID スレッド・プロパティを使用して取得できます。
- CS_REQ_MIGRATE を確認して、クライアントが接続マイグレーションをサポートするかどうかを判断します。クライアントが接続マイグレーションをサポートする場合、アプリケーションは SRV_T_SESSIONID プロパティを使用してセッション ID をクライアントに送信する必要があります (クライアントがまだセッション ID を取得していない場合)。クライアントにセッション ID を割り当てることで、アプリケーションは必要が生じたときにマイグレートするようにクライアントに指示できます。

マイグレーション・ステータス (SRV_MIGRATE_STATE) イベントの管理

SRV_MIGRATE_STATE イベント・ハンドラは、マイグレーション・ステータスの変更を管理し、各変更に適したアクションを実行できます。

- SRV_MIG_READY に変更された SRV_MIGRATE_STATE

  “ready” マイグレーション・ステータスは、クライアントでマイグレーションの準備ができており、現在のところは要求を送信していないことを示しています。SRV_MIGRATE_STATE イベント・ハンドラでは、Open Server はマイグレート先のサーバとクライアント・コンテキストを共有します。その後、アプリケーションはイベント・ハンドラから戻ることができ、Open Server はマイグレーションの開始を自動的にクライアントに指示できます。

- SRV_MIG_FAILED に変更された SRV_MIGRATE_STATE

  マイグレーション・ステータスが “failed” に変更されたために SRV_MIGRATE_STATE イベント・ハンドラがトリガされた場合、アプリケーションは再度コンテキストにアクセスする必要があります。アプリケーションは、srv_send_ctlinfo() 関数を使用して SRV_MIG_STATE イベント・ハンドラから別のマイグレーション試行を要求できます。ただし、クライアントは再度マイグレートする準備ができたことを示す前に、別のクエリを送信した可能性があります。アプリケーションは、そのような要求を処理またはマイグレートできる必要があります。

クライアント・コンテキストの共有

サーバがクライアントの処理を開始して続行する場合、サーバはクライアントのセッション ID により識別されるクライアントのコンテキストにアクセスできる必要があります。通常、クライアントのコンテキストには、クライアントのイベント・ハンドラがアクセス可能なデータ (グローバル・データなど) が含まれています。接続に必要なコンテキストの量は、Open Server アプリケーションが提供するサービスによって決まります。サービスのコンテキストが少なければ少ないほど、共有する必要があるコンテストは少なくなります。

マイグレーション再開 (SRV_MIGRATE_RESUME) イベントの管理

アプリケーションは、残りの結果とメッセージを SRV_MIGRATE_RESUME イベント・ハンドラ内でクライアントに送信します。Open Server がクライアントに送信する結果とメッセージは、アプリケーションとマイグレーションのタイプによって異なります。ただし、アプリケーションは SRV_DONE_FINAL 結果タイプをクライアントに送信して、SRV_MIGRATE_RESUME イベント・ハンドラを終了する必要があります。

切断 (SRV_DISCONNECT) イベントのマイグレーション

SRV_DISCONNECT イベント・ハンドラでは、アプリケーションは SRV_T_MIGRATE_STATE を確認してクライアントのマイグレーション・ステータスを判断する必要があります。

- マイグレーション・ステータス SRV_MIG_REQUESTED は、クライアントがマイグレーション要求に応答できるようになる前に Open Server アプリケーションが接続を終了したため、SRV_DISCONNECT イベントがトリガされたことを示しています。
- マイグレーション・ステータス SRV_MIG_MIGRATING は、新しいサーバへのマイグレーションが成功した後でクライアント・アプリケーションが接続を閉じたため、SRV_DISCONNECT イベントがトリガされたことを示しています。

- 他のすべてのマイグレーション・ステータスの場合、他のサーバがこのコンテキストを取得しないため、クライアントは接続固有のコンテキストがクリーンアップされるようにする必要があります。

バッチ内マイグレーションの管理

長時間実行されているイベント・ハンドラは、マイグレーション・ステータスを定期的に検査する必要があります。他の Open Server スレッドは、イベント・ハンドラ・プロセスがまだ実行中の間でも、マイグレーション要求を送信できます。この場合、イベント・ハンドラは可能であればプロセスを中断し、接続が新しいサーバにマイグレートされるまで結果の生成と送信を延期する必要があります。

アテンション処理

クライアントがアテンション・メッセージを送信して未処理の要求をキャンセルすると、SRV_T_GOTATTENTION スレッド・プロパティが CS_TRUE に設定されて SRV_ATTENTION イベント・ハンドラが呼び出されます。接続マイグレーションで必要になる具体的なアテンション処理は次のとおりです。

- SRV_MIGRATE_STATE イベント・ハンドラと SRV_MIG_READY ステータスの場合

  マイグレーションの準備ができたことをクライアントが示す前にアテンション・メッセージが SRV_MIGRATE_STATE イベント・ハンドラに到着した場合、SRV_MIGRATE_STATE イベント・ハンドラが終了すると Open Server はアテンションを確認します。これにより、クライアントからの要求が完了します。マイグレーションが成功すると、クライアントのマイグレート先のサーバはこのアテンション・メッセージを受信しません。クライアントが Open Server からの結果を待機していないため、SRV_MIGRATE_RESUME イベント・ハンドラは呼び出されません。

  したがって、アプリケーションは他のサーバがコンテキストを使用できるようにする前に、SRV_T_GOTATTENTION プロパティが CS_TRUE に設定されているかどうかを確認する必要があります。SRV_T_GOTATTENTION が CS_TRUE に設定されている場合、コンテキストを更新して、クライアントが操作をキャンセルしたことを示す必要があります。

- SRV_MIGRATE_RESUME イベント・ハンドラの場合

  マイグレーションの準備ができたことが示され、マイグレーションが成功した後、クライアントがアテンション・メッセージを送信した場合、アテンションはクライアントのマイグレート先のサーバに送信されます。このため、マイグレーションが成功すると、オリジナル・サーバがコンテキストを更新してキャンセルを反映した場合でも、SRV_MIGRATE_RESUME イベント・ハンドラがアテンションを受信可能な可能性があります。したがって、アプリケーションは SRV_MIGRATE_RESUME イベント・ハンドラを実行する前に、クライアントがアテンションをサーバに送信したかどうかを確認する必要があります。

Open Server の切断

アプリケーションは、マイグレーションが要求されたときでもクライアント接続を終了できますが、Open Server が終了コマンドを発行する直前に送信された新しいクライアント・コマンドは失われる可能性があります。これを回避するには、アプリケーションが次の処理を実行する必要があります。

- 可能であれば、クライアントがマイグレーションを指示された場合は接続を終了しないようにします。
- クライアントを切断する必要がある場合、Open Server はマイグレーションを要求する前に適切な待機時間を設定する必要があります。これにより、別のコマンドを発行するまでにマイグレーション要求を検出する時間がクライアントに与えられます。
- Open Server が接続を終了すると、SRV_DISCONNECT イベント・ハンドラが呼び出されます。このハンドラ内で、マイグレーション・ステータスが SRV_MIG_REQUESTED に設定されたままの場合は、他のサーバがコンテキストを使用できるようにします。

マイグレートされたクライアントからの接続の受け入れ

Open Server は、SRV_CONNECT イベント・ハンドラ内の SRV_T_MIGRATED プロパティを検査することで、新しい接続がマイグレート中か、マイグレートを完了したかを判断できます。SRV_T_MIGRATED が TRUE の場合、SRV_T_SESSIONID プロパティを使用してクライアントからセッション ID を取得できます。セッション ID を変更することもできますが、セッション ID は後でクライアントをマイグレートするために必要ではありません。

クライアントのマイグレート時にクライアントがコマンドを実行していた場合、SRV_MIGRATE_RESUME イベントがトリガされ、Open Server は結果をクライアントに送信してコマンドを完了できます。アプリケーションは、セッション情報の取得を行います。さらに、SRV_MIGRATE_RESUME イベント・ハンドラ内から結果をクライアントに引き続き送信する必要があるかどうかも判断する必要があります。

エラー・メッセージ

接続マイグレーション機能を使用しているときに発生する可能性があるエラー・メッセージを次に示します。

エラー	説明
srv_thread_props(): プロパティ (SRV_T_SESSIONID) は利用できません。	クライアントがまだ取得していないセッション ID を取得しようとしました。
srv_send_ctlinfo(SRV_CTL_MIGRATE): 接続はマイグレートできません。	クライアントがマイグレーションをサポートしません。
srv_send_ctlinfo(SRV_CTL_MIGRATE)：この時点でマイグレーションをキャンセルすることはできません。	すでに開始されたマイグレーションのキャンセルを要求しました。
マイグレーションが失敗しました。SRV_MIGRATE_STATE ハンドラはインストールされていません。	デフォルトの SRV_MIGRATE_STATE ハンドラがマイグレーション失敗を検出しました。

CS_BROWSEDESC 構造体

srv_tabname と srv_tabcolname は、CS_BROWSEDESC 構造体を使ってブラウズ・モード・クエリの基本構造体に関する情報を返します。

CS_BROWSEDESC 構造体の定義は次のとおりです。

```
/*
 ** CS_BROWSEDESC
 ** The Open Server browse column description
 ** structure.
 */
 typedef struct _cs_browsedesc
 {
     CS_INT       status;
     CS_BOOL      isbrowse;
     CS_CHAR      orignname[CS_MAX_NAME];
     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、次に示す記号の OR (論理和) 演算の結果のビットマスクです。

  CS_EXPRESSION は、カラムが式 (次のクエリの “sum*2” など) の結果であることを示します。

  ```
  select sum*2 from areas
  ```

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である場合には、データベース内のカラム名と異なった見出しになります。

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

- isbrowse は、ブラウズ・モードでカラムを更新できるかどうかを示します。

  カラムが式の結果でもなく timestamp カラムでもない場合や、カラムがブラウズ可能なテーブルに所属している場合は、更新が可能です。ユニークなインデックスと timestamp カラムがあるテーブルは、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オリジナル名です。

  カラムを更新する場合は、select 文で指定された見出しではなく、オリジナル名を使用する必要があります。

- 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” リストの最初のテーブルがテーブル 1、次がテーブル 2 というようになります。

- tablename は、カラムが所属しているテーブルの名前です。

- tabnlen は、tablename の長さ (バイト数) です。

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。たとえば、次のように使用します。

- srv_bind は、CS_DATAFMT 構造体を使って送信元または送信先のプログラム変数を記述します。
- srv_descfmt は、CS_DATAFMT 構造体を使ってクライアント・データを記述します。
- cs_convert を実行するには、変換前と変換後のデータを記述する CS_DATAFMT 構造体が必要です。

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、srv_bind は name および usertype フィールドを使用しません。また、srv_descfmt は format フィールドを使用しません。ルーチンが使用する CS_DATAFMT のフィールドについては、そのルーチンのリファレンス・ページを参照してください。

CS_DATAFMT 構造体の定義は次のとおりです。

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

    /*
       ** The following field is not used in Open Server.
       ** It must be set to 1 or 0.
       */
       CS_INT     count;

     /*
       ** These fields are used to support user-defined
       ** datatypes and international datatypes:
       */
       CS_INT    usertype;            /* User-defined 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、つまりカラムまたはパラメータの名前です。
- 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です。「データ型」(187 ページ) にリストされている Open Server データ型のいずれかです。

注意　datatype フィールドには、データの Open Server データ型を記述します。usertype は、Open Server データ型のほかにアプリケーション定義データ型がデータに設定されている場合にのみ使用します。

たとえば、次の Adaptive Server Enterprise 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、この新しい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
 )
```

ユーザ定義のデータ型をサポートする Open Server アプリケーションは、CS_DATAFMT の datatype フィールドを CS_DATETIME_TYPE に、usertype フィールドを型 birthday のユーザ定義の ID に設定することによって、クライアントにこの情報を返します。

- format は、文字またはバイナリ・データの送信先のフォーマットを記述します。format は、次の記号の OR (論理和) 演算の結果のビットマスクです。表 2-7 に、format の有効値を示します。

表 2-7: *format* の値 (CS_DATAFMT)

記号	意味	注意
CS_FMT_NULLTERM	データは NULL で終了しなければならない。	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。	バイナリ、image、文字、または text データ
CS_FMT_UNUSED	このデータ型には埋め込みも NULL 終了も適用されない。	すべてのデータ型

- maxlength は、CS_DATAFMT を使用している Open Server ルーチンに応じて、さまざまな長さを表すことができます。表 2-8 に、maxlength が表す長さを示します。

表 2-8: *maxlength* の意味 (CS_DATAFMT)

Open Server ルーチン	maxlength の意味
srv_bind	バインド変数の長さ
srv_descfmt	記述されているカラムまたはパラメータの可能な最大長
cs_convert	変換元データの長さと変換先バッファ容量の長さ

- scale はデータの位取りです。データ型が decimal または numeric の場合だけ、scale を使用します。

  scale の有効値は、CS_MIN_SCALE から CS_MAX_SCALE の範囲です。デフォルトの位取りは CS_DEF_SCALE です。

  送信先データが送信元データと同じ位取りを使用することを示すには、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 はデータの精度です。データ型が decimal または numeric の場合だけ、precision を使用します。

  precision の有効値は、CS_MIN_PREC から CS_MAX_PREC の範囲です。デフォルトの精度は CS_DEF_PREC です。

  送信先データが送信元データと同じ精度を使用することを示すには、precision を CS_SRC_VALUE に設定します。

  - precision は、scale 以上に設定します。
  - status は、情報の種類を示すために使用されるビットマスクです。表 2-9 に、status に指定できる情報の種類を示します。

表 *2-9: status の値 (CS_DATAFMT)*

記号値	意味
CS_CANBENULL	カラムは、NULL 値を含むことができる。
CS_DESCIN	CS_DATAFMT 構造体は、動的 SQL 入力パラメータを記述している。
CS_DESCOUT	CS_DATAFMT 構造体は、動的 SQL 出力パラメータを記述している。
CS_HIDDEN	カラムは公開された「隠しカラム」。
CS_INPUTVALUE	パラメータは、カーソル・オープン・コマンドまたは戻り値のない RPC パラメータのための入力値。
CS_KEY	カラムは、キー・カラム。
CS_RETURN	パラメータは、RPC コマンドへの戻りパラメータ。
CS_TIMESTAMP	カラムは、*timestamp* カラム。アプリケーションは、ブラウズ・モードで更新を行うときに timestamp カラムを使用する。
CS_UPDATABLE	カラムは、更新可能なカーソル・カラム。
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。
CS_VERSION_KEY	カラムは、ローのバージョン・キーの一部である。 Adaptive Server Enterprise は、バージョン・キーを位置付けのために使用する。
CS_NODEFAULT	パラメータに対して指定されたデフォルトはない。

- count は、Server Library ルーチンでは使用しません。常に 0 または 1 に設定してください。
- usertype は、返されるデータが存在する場合のユーザ定義のデータ型です。
- 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_IODESC 構造体

CS_IODESC は「I/O 記述子構造体」とも呼ばれ、text または image データを記述します。

クライアントの text または image データを処理するときには、Open Server アプリケーションは cmd 引数を CS_GET に設定して srv_text_info を呼び出します。この呼び出しでデータが挿入されるのは、CS_IODESC 引数の total_textlen フィールドだけです。

クライアントにデータ・カラムを送信する場合には、アプリケーションは、cmd 引数を CS_SET に設定して srv_text_info を呼び出します。この場合、CS_IODESC 構造体は、送信されている text または image カラムを記述しています。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 Enterprise id.*/
      CS_INT    timestamplen;           /* Length of timestamp.*/
      CS_BYTE   textptr[CS_TP_SIZE];    /* Adaptive Server Enterprise pt  */
      CS_INT    textptrlen;             /* Length of textptr.*/
 } 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 の種類です。text および image に関するオペレーションでは、iotype の値は CS_IODATA です。
- datatype は、データ・オブジェクトの「データ型」です。datatype の有効値は、CS_TEXT_TYPE と CS_IMAGE_TYPE だけです。
- locale は、現在、Open Server では使用されません。NULL に設定します。
- usertype は、Open Server では使用されません。
- total_txtlen は、text または image 値の全体の長さ (バイト数) です。
- offset は、今後の使用のために予約されています。
- log_on_update は、この text または image 値に対する更新をログに記録するかどうかを表します。
- name は、text または image カラムの名前です。

- namelen は、name の長さ (バイト数) です。CS_NULLTERM の場合は、名前が null で終了することを示します。
- 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、カラムの挿入または取得のための text または image バイトの配列です。
- textptrlen は、textptr の長さ (バイト数) です。

CS-Library

CS-Library は、Open Server アプリケーションと Open Client アプリケーションの両方で有用あるいは必要であるユーティリティ・ルーチンおよび構造体の集まりです。以前のバージョンでは、Server-Library と Client-Library の両方でこのようなユーティリティ・ルーチンと構造体を提供していたため、無駄な重複がありました。

共通ルーチン

CS-Library のルーチンは、次の機能をサポートします。

- データ型変換
- 算術演算
- 文字セット変換
- 日時オペレーション
- ソート順オペレーション
- ローカライゼーション・ルーチン

CS-Library には、CS-Library の構造体を割り付けるルーチンも含まれています。

スタンドアロンの CS-Library アプリケーションを作成することもできますが、このライブラリの主な機能は、Open Client アプリケーションと Open Server アプリケーションに共通のユーティリティを提供することです。

これらのルーチンが提供する機能には、現行の Server-Library ルーチンが提供しているものもあります。Server-Library ルーチンを対応する CS-Library ルーチンに置き換える必要は現時点ではありませんが、将来的には必要になる可能性があります。

共通データ構造体

共通ルーチンに加えて、CS-Library は Open Client と Open Server の両方のアプリケーションに役立つデータ構造体を提供しています。これらのデータ構造体の中には、アプリケーション・プログラミング環境、つまり「コンテキスト」に関する情報が格納された CS_CONTEXT 構造体が含まれています。

Open Server アプリケーションのプログラマは、この構造体に格納されているグローバル・アプリケーション属性を設定することで、アプリケーションの動作を調整できます。この機能の詳細については、「プロパティ」(130 ページ) を参照してください。

他の CS-Library 構造体は、Open Client とアプリケーションと Open Server アプリケーションの間で通信されるデータについての情報を含んでいます。

注意 Client-Library プログラムと Server-Library プログラムは、コンテキスト構造体を必要とし、CS-Library だけがこの構造体を割り付けることができます。そのため、Client-Library プログラムと Server-Library プログラムはすべて、CS-Library への呼び出しを少なくとも 2 つ、つまり、CS_CONTEXT を割り付ける呼び出しとそれを解除する呼び出しを含んでいます。

エラー処理

Open Server アプリケーションは、CS-Library エラーを報告するために、cs_config ルーチンで「メッセージ・コールバック・ルーチン」をインストールしなければなりません。srv_props でインストールした標準の Open Server エラー・ハンドラは、cs_convert への呼び出しで発生したデータ変換エラーなどの CS-Library エラーを捕捉しません。

Open Server アプリケーションが CS-Library ハンドラをインストールしていない場合、Open Server は、アプリケーションが srv_version を呼び出したときにデフォルト・ハンドラをインストールします。このデフォルト・ハンドラは、Open Server のログに CS-Library エラーを書き込みます。

CS-Library エラーの処理の詳細と CS-Library の全般的な情報については、『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_SERVERMSG 構造体

CS_SERVERMSG 構造体は、サーバ・エラー・メッセージに関する情報を含んでいます。

Open Server では、CS_SERVERMSG 構造体を使用し、srv_sendinfo ルーチンを介してクライアントにエラー・メッセージを送信します。

CS_SERVERMSG 構造体の定義は次のとおりです。

```
/*
 ** CS_SERVERMSG
 ** The server message structure.
 */

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

    /*
     ** If the error involved a stored procedure,
     ** the following fields contain information
     ** about the procedure:
     */
     CS_CHAR  proc[CS_MAX_NAME];
     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、クライアントに報告される Open Server またはアプリケーションの「メッセージ番号」です。
- state は、メッセージ生成時の状態です。アプリケーションによって定義します。
- severity は、メッセージの重大度です。
- text は、メッセージ・テキストです。
- textlen は、text の長さ (バイト数) です。

- svrname は、メッセージを生成したサーバ名です。この値を、現在実行中の Open Server アプリケーションの名前などの別の名前にすることもできます。
- 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」の名前です (存在する場合)。
- proclen は、proc の長さ (バイト数) です。
- line は、メッセージを生成したストアド・プロシージャ内の行番号です (存在する場合)。
- status は、メッセージのまとまりがメッセージの最初、最後、中間のどの部分であるか、メッセージに拡張エラー・データが含まれているかどうかについての情報を含んでいます。status は、バイト順フラグなので、2 つ以上の値を設定できます。次に例を示します。

  ```
  mrec.status = CS_FIRST_CHUNK | CS_LAST_CHUNK;
  ```

  mrec は、CS_SERVERMSG 構造体として宣言されます。

  表 2-10 に、status の有効値を示します。

表 *2-10: CS_SERVERMSG* 構造体の *status* フィールドの値

値	意味
CS_HASEED	メッセージに関連する拡張エラー・データがある。
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。 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。 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。
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。 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。 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。

- 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。

「クライアント・コマンド・エラー」(32 ページ) を参照してください。

カーソル

Adaptive Server Enterprise が実装しているカーソルは Server-Library と Client-Library によってサポートされています。

Adaptive Server Enterprise にカーソルがどのように実装されているかについては、『ASE リファレンス・マニュアル』を参照してください。

Client-Library でカーソルがどのようにサポートされているかについては、『Open Client Client-Library/C リファレンス・マニュアル』を参照してください。

カーソルの概要

カーソルとは、SQL 文にリンクされた記号名です。カーソルを宣言すると、リンクが確立されます。SQL 文は、次のいずれかです。

- SQL select 文
- Transact-SQL execute 文
- 動的 SQL 準備文

カーソルに関連付けられた SQL 文は、カーソルの本体と呼ばれます。クライアントがカーソルをオープンすると、カーソルの本体が実行され、結果セットが生成されます。Open Server アプリケーションは、カーソル要求を検出してカーソル結果をクライアントに返します。

カーソルの利点

カーソルを使うことによって、クライアント・アプリケーションは、単にデータ・ローの完全なセットを取得するだけでなく、結果セット内の個々のローにアクセスできます。

1 つの接続で同時に複数のカーソルをオープンすることもできます。すべてのカーソル結果セットを同時にアプリケーションで使用できるため、アプリケーションは、随時、カーソル結果セットからデータ・ローをフェッチすることができます。これは、1 ローずつ順番に処理しなければならない他のタイプの結果セットとは対照的な方法です。

また、クライアント・アプリケーションは、カーソル結果セットのローをフェッチしながら基本データベース・テーブルを更新することができます。

Open Server アプリケーションとカーソル

この項では、Open Server のカーソル・サポートの基本的な情報について説明します。SRV_CURSOR イベント・ハンドラの構築方法の詳細については、「特定の要求への応答方法」(65 ページ) を参照してください。

カーソル要求の生成方法

クライアント・アプリケーションは、Open Server アプリケーションにカーソル・コマンドを発行してカーソルを要求します。

クライアント・アプリケーションは、Client-Library コマンドの ct_cursor を呼び出してカーソル・コマンドを開始します。『Open Client Client-Library/C リファレンス・マニュアル』を参照してください。

カーソルを要求すると、Open Server は SRV_CURSOR イベントを生成します。カーソル要求に応答する場合、Open Server アプリケーションには SRV_CURSOR イベント・ハンドラが必要です。

カーソル・コマンドのタイプ

表 2-11 に、クライアントが発行できるカーソル・コマンドのタイプを示します。

表 *2-11:* カーソル・コマンドの概要

コマンドのタイプ	実行する内容
宣言	カーソル名をカーソルの本体と関連付ける。
オープン	カーソルの本体を実行し、カーソル結果セットを生成する。
情報	カーソルのステータスを報告するか、カーソル・ロー・フェッチ・カウントを設定する。
フェッチ	カーソル結果セットからローをフェッチする。
更新または削除	現在のカーソル・ローの内容を更新または削除する。
クローズ	カーソル結果セットを使用できないようにする。カーソルを再オープンすると、カーソル結果セットが再生成される。
割り付け解除	カーソルを存在しないようにする。割り付け解除されたカーソルを再オープンすることはできない。

通常、クライアント・アプリケーションは 表 2-11 にリストされた順序でカーソル・コマンドを発行しますが、必要に応じてこの順序が変わることもあります。たとえば、カーソルをフェッチしてからクローズし、再びオープンしてもう一度ローをフェッチする、というような場合です。

クライアントとのカーソル情報の交換

SRV_CURSOR イベント・ハンドラは、srv_cursor_props ルーチンと SRV_CURDESC 構造体を使って、カーソル情報をクライアントと交換します。srv_cursor_props では、現在の情報をクライアントに送信してクライアントからカーソル情報を取得する場合、SRV_CURDESC 構造体にアクセスします。

srv_cursor_props (239 ページ) を参照してください。

クライアントとサーバは、1 つの接続で複数のカーソル情報を交換できるので、それぞれのカーソルを識別する必要があります。Open Server アプリケーションは、カーソル宣言への応答として、ユニークなカーソル ID を送り返します。このカーソルが存在する間は、クライアントとサーバはこの ID を使用してこのカーソルを参照します。

SRV_CURDESC 構造体

SRV_CURDESC 構造体には、次のようなカーソル情報が含まれます。

- ユニークなカーソル ID
- クライアントが最後に発行したカーソル・コマンドのタイプ
- カーソルのステータス

SRV_CURDESC 構造体の定義は次のとおりです。

```
/*
 ** SRV_CURDESC
 ** The Open Server cursor description
 ** structure.
 */

 typedef struct srv_curdesc
 {
     CS_INT       curid;
     CS_INT       numupcols;
     CS_INT       fetchcnt;
     CS_INT       curstatus;
     CS_INT       curcmd;
     CS_INT       cmdoptions;
     CS_INT       fetchtype;
     CS_INT       rowoffset;
     CS_INT       curnamelen;
     CS_CHAR      curname[CS_MAX_CHAR];
     CS_INT       tabnamelen;
     CS_CHAR      tabname[CS_MAX_CHAR];
     CS_VOID      *userdata;

 } SRV_CURDESC;
```

表 2-12 に、SRV_CURDESC 構造体のフィールドのリストを示します。

表 *2-12: SRV_CURDESC* 構造体のフィールド

フィールド名	説明	注意
curid	現在のカーソル識別子	Open Server アプリケーションは、クライアントからの CS_CURSOR_DECLARE コマンドに応答するときに curid を設定しなければならない。その後にクライアントから発行される、宣言されたカーソルに関連するコマンドはすべて、識別子として curid を使用しなければならない。現在のカーソル識別子がないか、クライアントがすべての使用可能なカーソルのステータスを要求している場合には、curid は 0 に設定される。
numupcols	カーソル更新句内のカラム数	numupcols は、更新カラムがない場合には 0 に設定される。この情報は、カーソルが宣言されるときに使用できる。
fetchcnt	このカーソルに関する現在のロー・フェッチ・カウント、つまり CS_CURSOR_FETCH コマンドに応答してクライアントに送信されるローの数	fetchcnt は、CS_CURSOR_INFO コマンドをクライアントから受け取るか、このコマンドに応答するクライアントに CS_CURSOR_INFO コマンドを送信するときに、設定される。クライアントでロー・フェッチ・カウントが明示的に設定されていない場合は、fetchcnt は 1 に設定される。Open Server アプリケーションが、要求されたフェッチ・カウントをサポートできない場合は、応答する前にこのフィールドを別の値に設定することができる。
curstatus	現在のカーソルのステータス	Open Server は、クライアントから受け取ったカーソル・コマンドに応答してカーソル・ステータスを設定する。有効な値のリストについては、「curstatus の値」(62 ページ) を参照。
curcmd	現在のカーソル・コマンド・タイプ	有効な値のリストについては、表 2-14 を参照。
cmdoptions	カーソル・コマンドに関するオプション	すべてのコマンドに関連オプションがあるわけではない。cmdoptions の値は、カーソル・コマンドのタイプによって異なる。コマンド別の cmdoptions の値のリストについては、表 2-14 を参照。

フィールド名	説明	注意
fetchtype	クライアントが要求したフェッチのタイプ	fetchtype は、クライアントから CS_CURSOR_FETCH コマンドを受け取ったときに書き込まれる。有効なフェッチ・タイプとその意味は次のとおり。 • CS_NEXT − 次のロー • CS_PREV − 前のロー • CS_FIRST − 最初のロー • CS_LAST − 最後のロー • CS_ABSOLUTE − rowoffset フィールドで識別されるロー • CS_RELATIVE − 現在のローに対して rowoffset フィールドの値を加減算したもの Adaptive Server Enterprise への要求は必ず CS_NEXT の fetchtype である。
rowoffset	CS_ABSOLUTE または CS_RELATIVE フェッチに関するロー位置	rowoffset は、他のすべてのフェッチ・タイプに対して定義されていない。rowoffset は、クライアントから CS_CURSOR_FETCH コマンドを受け取ったときに設定される。
curnamelen	curname 内のカーソル名の長さ	curname が有効ではない場合は、curnamelen が 0 に設定される。curnamelen によって、カーソル名の長さが返される。
curname	現在のカーソルの名前	
tabnamelen	tabname のテーブル名の長さ	tabname が有効ではない場合は、tabnamelen が 0 に設定される。tabnamelen によって、テーブル名の長さが返される。tabnamelen は、クライアントから CS_CURSOR_UPDATE コマンドまたは CS_CURSOR_DELETE コマンドを受け取ったときに書き込まれる。
tabname	カーソルの update または delete コマンドに関連するテーブル名	tabname は、カーソルの update コマンドまたは delete コマンドに関連付けられているテーブル名。tabname は、クライアントから CS_CURSOR_UPDATE コマンドまたは CS_CURSOR_DELETE コマンドを受け取ったときに書き込まれる。
userdata	ユーザ用のデータ領域に対するポインタ	このフィールドを使用すると、グローバル変数や静的な変数を使用しなくてもデータを特定のカーソルに関連付けることができる。Open Server は、userdata を操作しない。このコマンドは、Open Server アプリケーション・プログラマのために提供されている。

curstatus の値

SRV_CURDESC 構造体の curstatus フィールドは、次の表に示すどのような値の組み合わせでも可能なビットマスクです。

表 *2-13: curstatus* の値 *(SRV_CURDESC)*

値	意味
CS_CURSTAT_DECLARED	カーソルが宣言された。このステータスは、次のカーソル・コマンドが処理された後にリセットされる。
CS_CURSTAT_OPEN	カーソルがオープンされた。
CS_CURSTAT_ROWCNT	カーソルが、CS_CURSOR_FETCH コマンドで返されるローの数を指定した。
CS_CURSTAT_RDONLY	カーソルは、読み込み専用なので更新できない。CS_CURSOR_UPDATE または CS_CURSOR_DELETE を受信した場合は、Open Server アプリケーションからクライアントにこのカーソルについてのエラーが返される。
CS_CURSTAT_UPDATABLE	カーソルを更新できる。
CS_CURSTAT_CLOSED	カーソルはクローズされたが、割り付け解除されていない。後で再オープンできる。カーソル宣言においても、このステータスが設定される。Open Server は、CS_CURSOR_OPEN を受信した場合はこの値をクリアし、CS_CURSOR_CLOSE を受信した場合はこの値を再設定する。
CS_CURSTAT_DEALLOC	カーソルがクローズされ、割り付け解除された。このとき、他のステータス・フラグは設定されない。

curcmd の値

SRV_CURDESC 構造体の curcmd フィールドには、表 2-14 に示す値を設定できます。この表には、関連する cmdoptions の値も示します。

表 *2-14: curcmd* の値 *(SRV_CURDESC)*

値	意味	cmdoptions の有効値
CS_CURSOR_CLOSE	カーソルの close コマンド。	SRV_CUR_DEALLOC または SRV_CUR_UNUSED。SRV_CUR_DEALLOC は、カーソルが再オープンされないことを示している。Open Server アプリケーションは、関連するカーソル・リソースをすべて削除しなければならない。カーソル ID 番号は再使用できる。
CS_CURSOR_DECLARE	カーソルの declare コマンド。アプリケーションは、srv_langlen と srv_langcpy を介してカーソル文の実際のテキストを取得できる。	SRV_CUR_UPDATABLE、SRV_CUR_RDONLY、または SRV_CUR_DYNAMIC。SRV_CUR_DYNAMIC は、動的に準備された SQL 文に対してクライアントがカーソルを宣言することを示す。この場合、カーソル文のテキストは、実際には、作成された文の名前になる。
CS_CURSOR_DELETE	カーソルの delete コマンド。カーソルを使用して定位置ローの delete を実行する。	このコマンドには有効なオプションはない。cmdoptions の値は常に SRV_CUR_UNUSED。
CS_CURSOR_FETCH	カーソルの fetch コマンド。カーソルを使用してローの fetch を実行する。	このコマンドには有効なオプションはない。cmdoptions の値は常に SRV_CUR_UNUSED。

値	意味	cmdoptions の有効値
CS_CURSOR_INFO	カーソル情報コマンド。クライアントは、このコマンドを Open Server アプリケーションに送信し、カーソル・ロー・フェッチ・カウントを設定するか、カーソル・ステータス情報を要求する。Open Server アプリケーションは、現在のカーソルを記述するために、任意のカーソル・コマンド (CS_CURSOR_INFO 自体も含む) に応答して、クライアントにこのコマンドを送信する。	クライアントが現在のロー・フェッチ・カウントを書き込む場合は、SRV_CUR_SETROWS。fetchcnt フィールドには、要求されたフェッチ・カウントが含まれている。 クライアントが現在のカーソルのステータス情報を要求する場合は、SRV_CUR_ASKSTATUS。クライアントがアテンション送信後に使用可能なカーソルを調べたいときに、この値が発生する。curid フィールドには 0 が含まれる。Open Server アプリケーションは、現在使用可能な各カーソルに対して CS_CURSOR_INFO 応答を戻す必要がある。 Open Server アプリケーションが CS_CURSOR_INFO コマンドに応答する場合は、SRV_CUR_INFORMSTATUS。curstatus フィールドには、カーソル・ステータスが含まれる。
CS_CURSOR_OPEN	カーソルの open コマンド。	SRV_CUR_HASARGS または SRV_CUR_UNUSED。
CS_CURSOR_UPDATE	カーソルの update コマンド。カーソルを使用して定位置ローの update を実行する。Open Server アプリケーションは、srv_langlen と srv_langcpy を呼び出して、カーソルの update 文の実際のテキストを取得できる。	SRV_CUR_HASARGS または SRV_CUR_UNUSED。

カーソル要求の処理

Open Server アプリケーションは、SRV_CURSOR イベント・ハンドラを使用してカーソル要求を処理します。このハンドラには、発行されたカーソル・コマンドを検出したり、適切な情報とともに応答したりするコードが含まれています。

最初に、イベント・ハンドラは cmd 引数を CS_GET に設定して srv_cursor_props を呼び出し、現在のカーソルと、SRV_CURSOR イベントをトリガしたカーソル・コマンドを特定します。次に、Open Server は、Open Server アプリケーションの SRV_CURDESC 構造体の curcmd フィールドにコマンド・タイプを格納します。

この時点で他に取得する必要がある情報やクライアントに返信するデータがあれば、アプリケーションはそれを決定できます。場合によっては、パラメータ・フォーマットとパラメータを取得しなければならないことがあります。また、必要ならば、現在のカーソルのステータスとフェッチするロー数を取得しなければなりません。CS_CURSOR_INFO コマンドを送り返すだけで済む場合もありますが、結果データまたは戻りパラメータを送信しなければならないこともあります。

特定の要求への応答方法

この項では、SRV_CURSOR イベント・ハンドラが、特定のタイプのカーソル要求に応答する方法について説明します。

cmd を CS_SET に設定して srv_cursor_props を呼び出す前に、Open Server アプリケーションは SRV_CURDESC 構造体の curid フィールドおよびその他の関係するフィールドを設定しておく必要がある点に注意してください。

表 2-15 に、クライアントと Open Server アプリケーション間のカーソル要求と応答の有効な交換についてまとめてあります。右矢印 (→) は、cmd が CS_GET に設定されている (Open Server アプリケーションはクライアントから情報を取得する) ことを示しています。左矢印 (←) は、cmd が CS_SET に設定されている (Open Server アプリケーションは情報をクライアントに送信する) ことを示しています。

表 *2-15:* 有効なカーソル要求と応答

クライアントのアクション	Open Server アプリケーションの応答
カーソルの宣言 (SRV_CURDESC の curcmd フィールドは CS_CURSOR_DECLARE を含んでいる)	→ SRV_CURDESC から curcmd 値を取得する。 (srv_cursor_props) → カーソル・パラメータの数を取得する (存在する場合)。 (srv_numparams) → カーソル・パラメータのフォーマットを取得する (存在する場合)。 (type 引数を SRV_CURDATA に設定した srv_descfmt) → 更新カラム情報を取得する (存在する場合)。 (type 引数を SRV_UPCOLDATA に設定した srv_descfmt) → カーソル・コマンドの実際のテキストを取得する。 (srv_langlen と srv_langcpy) ← カーソル ID を設定する。curcmd フィールドを CS_CURSOR_INFO に、curid フィールドをユニークなカーソル ID に設定する。 (srv_cursor_props) ← DONE パケットを送信する。 (status 引数を SRV_DONE_FINAL に設定した srv_senddone)
現在のカーソルのステータスの要求またはフェッチ・カウントの送信 (SRV_CURDESC の curcmd フィールドは CS_CURSOR_INFO を含んでいる)	→ SRV_CURDESC 構造体から curcmd、curid cmdoptions の値を取得する。 (srv_cursor_props) ← クライアント側で cmdoptions フィールドが SRV_CUR_SETROWS に設定された場合は、1 回のフェッチで返されるローの数を送信する。 (curcmd を CS_CURSOR_INFO に設定した srv_cursor_props) ← クライアント側で cmdoptions フィールドが SRV_CUR_ASKSTATUS に設定された場合は、使用可能なすべてのカーソルのステータスを送信する。curcmd フィールドを CS_CURSOR_INFO に、curid フィールドをカーソル ID に設定する。 (アクティブなカーソル、つまり、宣言、オープン、またはクローズされたカーソルごとに srv_cursor_props を 1 回ずつ) ← DONE パケットを送信する。 (status 引数を SRV_DONE_FINAL に設定した srv_senddone)

クライアントのアクション	Open Server アプリケーションの応答
カーソルのオープン (SRV_CURDESC の curcmd フィールドは CS_CURSOR_OPEN を含んでいる)	→ SRV_CURDESC 構造体から curcmd 値と curid 値を取得する。 (srv_cursor_props) → カーソル・パラメータの数を取得する (存在する場合)。 (srv_numparams) → カーソル・パラメータと実際のパラメータのフォーマットを取得する (存在する場合)。 (type 引数を SRV_CURDATA に設定した srv_descfmt、srv_bind、srv_xferdata) ← カーソル・ステータスを送信する。curid を現在のカーソル ID に、curcmd を CS_CURSOR_INFO に設定する。 (srv_cursor_props) ← 結果ロー・フォーマットを記述する。 (type 引数を SRV_ROWDATA に設定した srv_descfmt) ← DONE パケットを送信する。 (status 引数を SRV_DONE_FINAL に設定した srv_senddone)
ローのフェッチ (SRV_CURDESC の curcmd フィールドは CS_CURSOR_FETCH を含んでいる)	→ SRV_CURDESC 構造体から curcmd 値と curid 値を取得する。 (srv_cursor_props) ← 結果ローを fetchcnt の数だけ送信する。 (type 引数を SRV_ROWDATA に設定した srv_bind、srv_xferdata) ← DONE パケットを送信する。 (status 引数を SRV_DONE_FINAL に設定した srv_senddone)

クライアントのアクション	Open Server アプリケーションの応答
カーソル更新コマンドの発行 (SRV_CURDESC の curcmd フィールドは CS_CURSOR_UPDATE を含んでいる)	→ SRV_CURDESC 構造体から curcmd 値と curid 値を取得する。 (srv_cursor_props)
または カーソル削除コマンドの発行 (SRV_CURDESC の curcmd フィールドは CS_CURSOR_DELETE を含んでいる)	→ 現在のローのキー・カラムを取得する。 (type 引数を SRV_KEYDATA に設定した srv_descfmt、srv_bind、srv_xferdata)
	→ curcmd が CS_CURSOR_UPDATE の場合は、更新値の数を取得する。 (srv_numparams)
	curcmd が CS_CURSOR_UPDATE の場合は、更新文の実際のテキストを取得する。 (srv_langlen と srv_langcpy)
	→ curcmd が CS_CURSOR_UPDATE の場合は、更新値を取得する。 (type 引数を SRV_CURDATA に設定した srv_descfmt、srv_bind、srv_xferdata)
	← DONE パケットを送信する。 (status 引数を SRV_DONE_FINAL に設定した srv_senddone)
カーソル・クローズ・コマンドの送信 (SRV_CURDESC の curcmd フィールドは CS_CURSOR_CLOSE を含んでいる)	→ SRV_CURDESC 構造体から curcmd 値と curid 値を取得する。 (srv_cursor_props)
	→ カーソル・ステータスを送信する。 (srv_cursor_props)
	← DONE パケットを送信する。 (status 引数を SRV_DONE_FINAL に設定した srv_senddone)

次のことに注意してください。

- Open Server アプリケーションがカーソル・コマンドに応答する場合、応答を終了するときには、必ず status 引数を “SRV_DONE_FINAL” に設定した srv_senddone を呼び出します。
- Open Server アプリケーションが、cmd を “SET” に設定した最初の srv_cursor_props コマンドを発行した後、status 引数を SRV_DONE_FINAL に設定して srv_senddone を発行するまでの間は、このアプリケーションから送信されるすべての情報がこのカーソルに適用されます。
- 内部的には、Open Server はクライアントがカーソルを宣言するときに受けるパラメータ・フォーマットを、クライアントがカーソルをオープンするときに受けるものと入れ替えます。この手順は、受けたパラメータのフォーマットが、パラメータ宣言のフォーマットと正確には同じでない場合に必要となります。たとえば、CS_INT としてパラメータを宣言しても、カーソルがオープンされるときに渡されるパラメータの型が CS_SMALLINT であることもあります。
- srv_xferdata は 1 つのデータ・ローを送信するため、CS_CURSOR_FETCH コマンドに応答するときは、現在のカーソルのロー・フェッチ・カウントの回数だけこのコマンドを呼び出します。

キー・データ

キーは、ローをユニークに識別するロー・データのサブセットです。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。これは、CS_CURSOR_DELETE または CS_CURSOR_UPDATE コマンドを処理するために使われます。カラムがキー・カラムである場合は、そのカラムを記述している CS_DATAFMT 構造体の status フィールドには CS_KEY ビットマスク・セットが設定されています。

更新カラム

クライアントがカーソルを「更新用」と宣言した場合は、SRV_CURDESC 構造体の cmdoptions フィールドは CS_FOR_UPDATE に設定され、numupcols フィールドはそのカーソルと関連する更新カラムの数に設定されます。

例

サンプル *ctos.c* には、カーソル・コマンド処理のコードが記述されています。

スクロール可能カーソル

スクロール可能カーソルの機能を利用すると、FETCH 文で NEXT 句、PREVIOUS 句、FIRST 句、LAST 句、ABSOLUTE 句、または RELATIVE 句を指定することで、結果セットの任意の場所に現在位置を設定することができます。実装されるスクロール可能カーソルは読み込み専用で、INSENSITIVE プロパティまたは SEMI_SENSITIVE プロパティが設定されます。

非スクロール可能で非反映型のカーソルも Open Server でサポートされており、CS_NOSCROLL_INSENSITIVE オプションを使用して設定されます。

Open Server に接続するリモート・クライアントは、ログイン時に、CS_REQ_CURINFO3 を使用してスクロール可能カーソルのサポートを要求できます。

SRV_CURDESC2 構造体

Open Server の SRV_CURDESC2 スクロール可能カーソル構造体は、「SRV_CURDESC 構造体」(59 ページ) で説明している SRV_CURDESC カーソル構造体のスーパーセットです。

表 2-12 に示すフィールドの他に、表 2-16 に示すフィールドが SRV_CURDESC2 構造体に追加されています。

表 *2-16: SRV_CURDESC2* 構造体の追加フィールド

フィールド名	説明
currow_pos	カーソルの現在のロー位置。
curtotalrowcount	結果セット内のローの総数 (非反映型のスクロール可能カーソルの場合のみ)。

curstatus の値

表 2-13 に示したオプションに加えて、次に示すカーソル宣言関連のオプションが SRV_CURDESC2 の *curstatus* フィールドで使用可能です。

表 *2-17: curstatus* の値 *(SRV_CURDESC2)*

値	意味
CS_CURSTAT _SCROLLABLE	読み込み専用の、非反映型スクロール可能カーソル。
CS_CURSTAT _INSENSITIVE	読み込み専用の、非スクロール可能、非反映型カーソル。 このカーソルを指定する場合は、CS_CURSTAT_INSENSITIVE を有効にし、CS_CURSTAT_SCROLLABLE を無効にする必要があります。 非反映型のスクロール可能カーソルを指定する場合は、CS_CURSTAT_INSENSITIVE と CS_CURSTAT_SCROLLABLE の両方を有効にする必要があります。
CS_CURSTAT _SEMISENSITIVE	読み込み専用の、半反映型スクロール可能カーソル。 このカーソルを指定する場合は、CS_CURSTAT_SCROLLABLE も有効にする必要があります。

curcmd の値

表 2-14 に示した値に加えて、表 2-18 に示す値が SRV_CURDESC2 構造体の curcmd フィールドで使用可能です。この表には、関連する cmdoptions の値も示します。

表 *2-18: curcmd* の値 *(SRV_CURDESC2)*

値	意味	cmdoptions の有効値
CS_NOSCROLL_ INSENSITIVE	非スクロール可能、非反映型のカーソル。	このコマンドには有効なオプションはない。cmdoptions の値は常に SRV_CUR_UNUSED。 **注意** CTOS アプリケーションを使用する場合、非スクロール可能カーソルでは ct_scroll_fetch ルーチンを使用しないでください。代わりに、ct_fetch ルーチンを使用してください。
CS_CURSOR_ DECLARE	スクロール可能カーソルのコマンド・オプション。	SRV_CUR_SCROLL, SRV_CUR_SCROLL_ INSENS、SRV_CUR_ SCROLL_SEMISENS, SRV_CUR_NOSCROLL_ INSENS. これらの cmdoptions はカーソルの declare サイクルにおいてのみ有効であり、SRV_CURDESC2 構造体の curcmd フィールドには、ct_cursor を発行しているリモート・クライアントに基づいて、これらのオプションのいずれかが格納されます。

srv_cursor_props2 ルーチン

SRV_CURDESC2 構造体をサポートするために、srv_cursor_props2 ルーチンが Open Server に追加されています。

15.0 より前のアプリケーションで、CS_VERSION_125 を設定する場合は、SRV_CURDESC 構造体と srv_cursor_props ルーチンを使用する必要があります。

Open Server でスクロール可能カーソルをサポートするバージョン 15.0 以降のアプリケーションでは、SRV_CURDESC2 構造体を使用します。また、アプリケーションを CS_VERSION_*xxx* に設定します。ここで、*xxx* は Open Server のバージョンに対応します。

srv_cursor_props2 の引数は次のとおりです。

```
ret = srv_cursor_props2(SRV_PROC *spp, CS_INT cmd,
SRV_CURDESC2 *cdp);
```

データ・ストリーム・メッセージ

データ・ストリーム・メッセージとは

データ・ストリーム・メッセージは、クライアントと Open Server アプリケーションとが情報を交換するための方法を提供します。

RPC も、機能は同様ですが、クライアントからサーバへの一方向のみです。メッセージはさまざまな通信の用途に適合させることによって、両方向で機能します。たとえば、Sybase では、ログイン時にセキュリティ・ハンドシェイクを実行するためにメッセージを使用します。

メッセージは、メッセージ ID と 0 個以上のパラメータで構成されます。クライアントと Open Server アプリケーションは、各メッセージ ID の意味を同一に解釈するようにプログラムされている必要があります。

ユーザ定義のメッセージ ID は、CS_USER_MSGID 以上 CS_USER_MAX_MSGID 以下でなければなりません。SRV_MINRESMSG から SRV_MAXRESMSG の間のメッセージ ID は、Sybase の内部使用のために予約されています。

クライアント・アプリケーションは、type を CS_MSG_CMD に設定して ct_command を呼び出し、メッセージを送ります。これにより、Open Server アプリケーションで SRV_MSG イベントがトリガされます。

クライアント・データ・ストリーム・メッセージの取得

メッセージ・データ・ストリームは、Open Server アプリケーションの SRV_MSG イベント・ハンドラをトリガします。このハンドラはクライアント・メッセージを取得できます。必要な作業は次のとおりです。

1 cmd を CS_GET に設定し、msgidp には Open Server がメッセージ ID を格納するバッファへのポインタを指定して、srv_msg を呼び出します。

メッセージがパラメータを持つ場合、srv_msg は *statusp* パラメータに SRV_HASPARAMS を設定します。

srv_msg (292 ページ) を参照してください。

2 必要な場合、パラメータの数を取得するために srv_numparams を呼び出します。

3　各パラメータを記述および取得するために、srv_descfmt、srv_bind、srv_xferdata を呼び出します。「パラメータとロー・データの処理」(126 ページ) を参照してください。

Open Server アプリケーションで、SRV_MSG イベント・ハンドラを使用せずにメッセージを取得することはできません。

クライアントへのデータ・ストリーム・メッセージの送信

Open Server アプリケーションは、クライアントにメッセージを送信できます。これを実行するために、アプリケーションは次のことを行います。

1　cmd を CS_SET に設定し、msgidp にはメッセージ ID が格納されているバッファを指すポインタを指定して、srv_msg を呼び出します。

SRV_HASPARAMS の *statusp 値は、メッセージがパラメータを持つことを意味します。SRV_NOPARAMS は、メッセージがパラメータを持たないことを意味します。

srv_msg (292 ページ) を参照してください。

2　srv_descfmt、srv_bind、srv_xferdata を呼び出して、各パラメータを記述し、送信します。

Open Server アプリケーションは、SRV_ATTENTION、SRV_CONNECT、SRV_DISCONNECT、SRV_URGDISCONNECT、SRV_START ハンドラを除く任意のイベント・ハンドラの中からメッセージを送信できます。

ディレクトリ・サービス

この項では、ディレクトリ・サービスを使用するために Open Server アプリケーションが実行する必要のあるタスクについて説明します。説明する内容は、次のとおりです。

- ディレクトリ・ドライバの指定
- ディレクトリ・サービスへの Open Server アプリケーションの登録

ディレクトリは、情報をディレクトリ・エントリとして保管し、それぞれのエントリに論理名を対応付けます。それぞれのディレクトリ・エントリには、ユーザ、サーバ、またはプリンタなど特定のネットワーク・エンティティについての情報が入っています。ディレクトリ・サービス (ネーミング・サービスと呼ばれる場合もあります) は、ディレクトリ・エントリの作成、修正、取得を管理します。

『Open Client Client-Library/C リファレンス・マニュアル』を参照してください。

ディレクトリ・ドライバの指定

libtcl.cfg ファイル内で正しいディレクトリ・サービス・プロバイダが指定されるように編集してあることを確認してから、ディレクトリ・サービスを使用するアプリケーションを実行してください。*libtcl.cfg* ファイルは、*$SYBASE/$SYBASE_OCS/config* ディレクトリ、またはコンテキスト・プロパティ CS_LIBTCL_CFG により指定されるパスにあります。サーバ・プロパティ SRV_DS_PROVIDER は、*libtcl.cfg* ファイルに指定されているドライバ名を返します。詳細については、各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。SRV_DS_PROVIDER プロパティの詳細については、srv_props (313 ページ) を参照してください。

使用しているプラットフォームの Open Client/Server でサポートされているディレクトリ・サービスについては、その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・サービスへの Open Server アプリケーションの登録

Open Server アプリケーションは、使用するディレクトリ・サービス・プロバイダを起動時に指定して、アプリケーション自体をディレクトリ・サービスに登録できます。

デフォルト以外のディレクトリ・サービス・プロバイダを指定するには、srv_props を使用して SRV_S_DS_PROVIDER サーバ・プロパティを設定してください。SRV_S_DS_PROVIDER のデフォルト値は、プラットフォームによって異なります。詳細については、使用しているプラットフォームの『Open Client/Server 設定ガイド』を参照してください。

ディレクトリ・サービスに Open Server アプリケーションを登録するには、srv_props を使用して、SRV_S_DS_REGISTER サーバ・プロパティを CS_TRUE (デフォルト) に設定してください。SRV_S_DS_REGISTER を CS_FALSE に設定すると、登録できません。

(cs_ctx_alloc と srv_version を使用して) CS_CONTEXT 構造体の割り付けと初期化を行ってから、上記のプロパティを設定し、その後で srv_init を呼び出してください。

srv_init が呼び出されると、Open Server アプリケーションは次の処理を行います。

- ディレクトリ・サービスから受信するアドレスを取得します。
- SRV_S_DS_REGISTER が CS_TRUE に設定されている場合は、Open Server アプリケーションのディレクトリ・サービス・エントリを更新するようにディレクトリ・サービスに指示します。
- その結果、ディレクトリ・サービスは、その “currentStatus” 属性を “active” に設定します。

ディレクトリ・サービス・ドライバの初期化に失敗した場合、Open Server は自動的に interfaces ファイルをバックアップ・ディレクトリとして使用します。次のいずれかの場合には、srv_init 呼び出しが失敗して、指定されたディレクトリ・サービスにアクセスできないことがあります。

- あると仮定していたロケーションに *libtcl.cfg* ファイルがない場合、つまり、そのファイルが読み込み不可能である場合

  情報エラーが返されます。

- あると仮定していたロケーションにディレクトリ・サービス・ドライバがない場合、つまり、そのドライバが読み込み不可能である場合

  情報エラーが返されます。

- ディレクトリ・サービスが要求に応答していない場合

  情報エラーが返されます。

- ディレクトリ・サービス内でサーバ・エントリが見つからない場合

  リスナが存在しないことを示すエラーが返されます。この場合、Open Server アプリケーションは interfaces ファイルをバックアップ・ディレクトリとして使用しません。

動的 SQL

動的 SQL を使用すると、クライアント・アプリケーションでは、実行時に値が決定される変数を含む SQL 文を実行することができます。

クライアント・アプリケーションは、プレースホルダを含む SQL 文と識別子を関連付けて、この文を部分的なコンパイルと保存のために Open Server アプリケーションに送ることによって、動的 SQL 文を準備します。この状態の文を、「準備文」と呼びます。

クライアント・アプリケーションは、準備文を実行する用意が整うと、SQL 文のプレースホルダに置き換える値を定義し、その文を実行するコマンドを送信します。この値は、コマンドの入力パラメータになります。

定められた回数だけ文が実行されると、クライアント・アプリケーションは文の割り付けを解除します。

動的 SQL の利点

動的 SQL によって、クライアント・アプリケーションは、Open Server にユーザからのさまざまな情報をさまざまなタイミングでやり取りして対話的に動作できます。これによって Open Server アプリケーションは、ユーザが提供したデータを使用して SQL クエリの欠けている部分を補うことができます。

『Embedded SQL/C Programmers Guide』を参照してください。

動的 SQL 要求の処理

クライアントが動的コマンドを発行すると、Open Server は SRV_DYNAMIC イベントを発生させます。Open Server アプリケーションが動的 SQL の結果を返す場合は、動的 SQL の要求に応答できるように、このアプリケーションに SRV_DYNAMIC イベント・ハンドラが含まれている必要があります。

srv_dynamic ルーチン

SRV_DYNAMIC イベント・ハンドラ内から、Open Server アプリケーションは srv_dynamic ルーチンを他の Server-Library ルーチンとともに使用して、クライアントの動的 SQL コマンドを取得し、応答します。srv_dynamic (253 ページ) を参照してください。各クライアント・コマンド・タイプ (準備、実行、割り付け解除) は、Open Server アプリケーションからの特定の応答を必要とします。

コマンド・タイプの検出

SRV_DYNAMIC イベント・ハンドラ内の最初のタスクは、クライアントが発行した動的コマンドのタイプを取得することです。場合によっては、動的文の ID とテキストも取得します。イベント・ハンドラは、取得した情報を保存して、後でクライアント要求に応答するときにはその情報を参照する必要があります。

クライアント動的 SQL コマンドへの応答

表 2-19 は、クライアントと Open Server アプリケーション間の動的 SQL 要求と応答の有効なやり取りを示します。右矢印 (→) は、cmd が CS_GET に設定されている (Open Server アプリケーションはクライアントから情報を取得する) ことを示しています。左矢印 (←) は、cmd が CS_SET に設定されている (Open Server アプリケーションは情報をクライアントに送信する) ことを示しています。

表 *2-19:* 有効な動的 *SQL* 要求と応答

クライアントのアクション	Open Server アプリケーションの応答
準備要求の発行 (オペレーション・タイプは CS_PREPARE)	→ オペレーション・タイプを取得する。 (srv_dynamic) → 文 ID の長さを取得する。 (srv_dynamic) → 文 ID を取得する。 (srv_dynamic) → 文の長さを取得する。 (srv_dynamic) → 文を取得する。 (srv_dynamic) ← クライアント・コマンドに肯定応答する。 (srv_dynamic) ← 文 ID の長さを送信する。 (srv_dynamic) ← 文 ID を送信する。 (srv_dynamic) ← DONE パケットを送信する。 (status 引数を SRV_DONE_FINAL に設定した srv_senddone)
文の入力パラメータの記述の要求 (オペレーション・タイプは CS_DESCRIBE_INPUT)	→ オペレーション・タイプを取得する。 (srv_dynamic) → 文 ID の長さを取得する。 (srv_dynamic) → 文 ID を取得する。 (srv_dynamic) ← クライアント・コマンドに肯定応答する。 (srv_dynamic) ← 文 ID の長さを送信する。 (srv_dynamic) ← 文 ID を送信する。 (srv_dynamic) ← 入力パラメータのフォーマットを送信する。 (type 引数を SRV_DYNDATA に設定した srv_descfmt と srv_xferdata。アプリケーションはフォーマットを送信するが、実際のデータは送信しないので、srv_bind の呼び出しは不要。CS_DATAFMT 構造体の status フィールドについて CS_DESCIN との論理和 (OR) 演算を実行してから、srv_descfmt を呼び出す必要がある。) ← DONE パケットを送信する。 (status 引数を SRV_DONE_FINAL に設定した srv_senddone)

クライアントのアクション	Open Server アプリケーションの応答
文の出力パラメータの記述の要求 (オペレーション・タイプは CS_DESCRIBE_OUTPUT)	→ オペレーション・タイプを取得する。 (srv_dynamic) → 文 ID の長さを取得する。 (srv_dynamic) → 文 ID を取得する。 (srv_dynamic) ← クライアント・コマンドに肯定応答する。 (srv_dynamic) ← 文 ID の長さを送信する。 (srv_dynamic) ← 文 ID を送信する。 (srv_dynamic) ← 結果ローのフォーマットを送信する。 (type 引数を SRV_DYNDATA に設定した srv_descfmt と srv_xferdata。アプリケーションはフォーマットを送信するが、実際のデータは送信しないので、srv_bind の呼び出しは不要。CS_DATAFMT 構造体の status フィールドについて CS_DESCOUT との論理和 (OR) 演算を実行してから、srv_descfmt を呼び出す必要がある)。 ← DONE パケットを送信する。 (status 引数を SRV_DONE_FINAL に設定した srv_senddone)
実行要求の発行 (オペレーション・タイプは CS_EXECUTE)	→ オペレーション・タイプを取得する。 (srv_dynamic) → 文 ID の長さを取得する。 (srv_dynamic) → 文 ID を取得する。 (srv_dynamic) → 動的パラメータの数を取得する。 (srv_numparams) → 入力パラメータ値を取得する。 (type 引数を SRV_DYNDATA に設定した srv_descfmt、srv_bind、srv_xferdata) ← クライアント・コマンドに肯定応答する。 (srv_dynamic) ← 文 ID の長さを送信する。 (srv_dynamic) ← 文 ID を送信する。 (srv_dynamic) ← 結果ローを送信する。 (type 引数を SRV_ROWDATA に設定した srv_descfmt、srv_bind、srv_xferdata) ← DONE パケットを送信する。 (status 引数を SRV_DONE_FINAL に設定した srv_senddone)

クライアントのアクション	Open Server アプリケーションの応答
即時実行要求の発行 (オペレーション・タイプは CS_EXEC_IMMEDIATE)	→ オペレーション・タイプを取得する。 (srv_dynamic) → 文 ID の長さを取得する。ID の長さは 0。(srv_dynamic) → 文の長さを取得する。 (srv_dynamic) → 文を取得する。 (srv_dynamic) ← クライアント・コマンドに肯定応答する。 (srv_dynamic) ← DONE パケットを送信する。 (status 引数を SRV_DONE_FINAL に設定した srv_senddone)
割り付け解除要求の発行 (オペレーション・タイプは CS_DEALLOC)	→ オペレーション・タイプを取得する。 (srv_dynamic) → 文 ID の長さを取得する。 (srv_dynamic) → 文 ID を取得する。 (srv_dynamic) ← クライアント・コマンドに肯定応答する。 (srv_dynamic) ← 文 ID の長さを送信する。 (srv_dynamic) ← 文 ID を送信する。 (srv_dynamic) ← DONE パケットを送信する。 (status 引数を SRV_DONE_FINAL に設定した srv_senddone)

例

サンプル *ctos.c* には、動的 SQL コマンドを処理するコードが記述されています。

動的なリスナ

動的なリスナにより、Open Server アプリケーションは srv_run ルーチンに対する呼び出しが行われた後に新しいリスナを作成できます。この結果、interface ファイル内のエントリが最小限の数で Open Server アプリケーションを起動でき、Open Server アプリケーションはリスナを再起動できます。

設定

SRV_LISTEN_PREBIND Open Server イベントは、指定された SRV_PROC 制御構造体により識別されるリスナを詳細に設定するために使用されます。たとえば、代替の SSL 証明書を指定できます。この設定は、スレッド・プロパティを使用して実行されます。

SRV_LISTEN_POSTBIND Open Server イベントにより、指定された SRV_PROC 制御構造体を使用して識別する動的なリスナの最終的な設定を行うことができます。たとえば、スレッド・プロパティを使用してリスナがバインドされるアドレスを決定できます。

プロパティ

次の 2 つの Open Server プロパティは、動的なリスナをサポートして使用されます。

- SRV_S_NUMLISTENERS プロパティは、クライアント接続の受信に関与する SRV_PROC 制御構造体の数を返します。
- SRV_S_MAXLISTENERS プロパティは、リスナ・スレッドの最大数を制限するために使用されます。

スレッド・タイプ

SRV_TLISTENER クライアント・スレッド・タイプは動的リスナに使用されます。

スレッド・プロパティ

次の 3 つのスレッド・プロパティは、動的なリスナのサポートに明示的に使用されます。

- SRV_T_LISTENADDR プロパティは、指定された SRV_PROC 制御構造体により識別されたリスナのアドレスを返します。SRV_PROC 制御構造体がリスナの場合、このプロパティは、リスナが接続を受け入れるアドレスを返します。
- SRV_T_LOCALID プロパティは、リスナに使用する SSL 証明書を指定します。
- SRV_T_REMOTEADDR プロパティは、SRV_PROC ピアのアドレスを返します。

動的なリスナの起動

動的なリスナは、次のパラメータ値を使用して srv_spawn ルーチンを呼び出すことで起動できます。

- sppp － スレッド構造体ポインタへのポインタは NULL であることが必要です。
- *stacksize* － スタック・サイズを CS_UNUSED として指定する必要があります。
- *funcp* － エントリ・ポイント関数ポインタは SRV_C_START_LISTENER であることが必要です。
- *argp* － このパラメータが CS_TRANADDR 構造体を指し示す必要があります。
- *priority* – 優先度を CS_UNUSED として指定する必要があります。

エラー

デフォルトでは、Open Server がエラーに対応する場合は、エラー・メッセージをログ・ファイルに書き込みます。エラー処理ルーチンをインストールすると、開発者はアプリケーションの応答を調整できます。

通常、エラー・ハンドラはエラーのタイプと重大度を検出し、検出した値に基づいて特定の処理を実行します。たとえば、アプリケーションは、srv_sendinfo ルーチンを介してクライアントに特定のエラーを送信し、それ以外のエラーをログ・ファイルに書き込みます。

エラー・ハンドラをインストールするには、property 引数を SRV_S_ERRHANDLE に設定した srv_props ルーチンを使用します。すべてのタイプのエラーを取得するには、アプリケーションで srv_version を呼び出した直後にエラー・ハンドラをインストールする必要があります。エラーが発生すると、Open Server は最も最近インストールされたエラー処理ルーチンを呼び出します。

srv_props (313 ページ) を参照してください。

エラーのタイプ

Open Server アプリケーション、クライアント・アプリケーション、Open Server 自体のどれもが、Open Server エラーを引き起こす可能性があります。以下に、発生するエラーの各タイプを説明します。

- Open Server アプリケーション・エラー ― このタイプのエラーは、アプリケーション・コードの誤りが原因で発生します。たとえば、アプリケーションが、最初にデータのフォーマットを記述しないで、クライアントにローのデータを送信しようとすると、Open Server はエラーを発生させます。
- クライアント・コマンド・エラー ― このタイプのエラーは、クライアントが送信した要求が不完全な場合や無意味な場合に発生します。クライアントのコードに誤りがあったりネットワークに問題があったりすると、要求は不完全になったり無意味になったりします。Open Server アプリケーションは、このようなエラーをクライアント要求のためのイベント・ハンドラで処理します。通常は srv_sendinfo を使って該当するエラー・メッセージをクライアントに送ることで、この処理を行います。詳細については、「クライアント・コマンド・エラー」(32 ページ) を参照してください。また、アプリケーションは、クライアント要求がエラーを引き起こしたことを示すために、srv_senddone の status パラメータを SRV_DONE_ERROR に設定することもできます。
- Open Server リソース・エラー ― このタイプのエラーは、Open Server 自体で発生します。通常はメモリやユーザ接続など、何らかのリソースの不足が原因で発生します。

エラーの重大度

各 Open Server エラーには、番号、重大度レベル、メッセージが関連付けられています。

エラーが発生すると、現在インストールされているエラー・ハンドラ関数が、エラー番号、エラー重大度レベル、メッセージ・テキストを使用して呼び出されます。エラー・ハンドラがインストールされていない場合は、Open Server のログ・ファイルにこの情報が記録されます。また、アプリケーションは、srv_log を呼び出してログ・ファイルに明示的に書き込むこともできます。

Open Server アプリケーションでログ・ファイルの最大サイズを設定するには、property 引数を SRV_S_LOGSIZE に設定して srv_props を実行します。

エラー番号と重大度レベルは、ヘッダ・ファイル *oserror.h* で定義されています。アプリケーションで、定義されているエラー値を使用するには、*oserror.h* をインクルードする必要があります。

表 2-20 に、Open Server エラーの重大度レベルを示します。

表 *2-20:* エラーの重大度

重大度	意味	適用されるエラーのタイプ
SRV_INFO	情報エラー。ほとんどのエラーがこの重大度である。この重大度レベルは、エラーは起こったが致命的ではないことを示す。通常は、Server-Library 関数の誤った呼び出しによって発生する。たとえば、srv_descfmt ですべてのカラムを記述しないうちに srv_xferdata を呼び出してローを送ると、SRV_INFO エラーが発生する。	Open Server アプリケーション・エラー クライアント・コマンド・エラー
SRV_FATAL_PROCESS	致命的なスレッド・エラー。エラーを受信したスレッドには、リカバリ不可能な内部エラーがある。たとえば、アプリケーションが srv_senddone を呼び出さずにイベントから戻ることがある。この重大度のエラーが発生すると、スレッドがクライアント・スレッド、SUB-PROC、またはサイト・ハンドラの場合、Open Server はスレッドに対して SRV_DISCONNECT イベントをキューイングする。次に、Open Server はスレッドを強制終了する。	Open Server リソース・エラー
SRV_FATAL_SERVER	致命的なサーバ・エラー。Open Server は、リカバリ不可能な内部エラーを検出した。このエラーが発生すると、Open Server は Open Server アプリケーションの SRV_STOP イベントをキューイングし、その結果、srv_run は CS_FAIL を返す。	Open Server リソース・エラー

オペレーティング・システム・エラー

オペレーティング・システム・エラーが発生した場合、オペレーティング・システム・エラー番号は SRV_ENO_OS_ERR 以外の値になります。また、オペレーティング・システム・エラー・テキストにはオペレーティング・システム・エラーの説明が記述されています。たとえば、srv_init が interfaces ファイルをオープンできない場合、オペレーティング・システム・パーミッション・エラーが原因である可能性があります。

エラー番号と対応するメッセージ・テキスト

エラー・トークンの完全なリストについては、ヘッダ・ファイルの *oserror.h* を参照してください。対応するエラー・テキストについては、ファイル *oslib.loc* を参照してください。

例

すべてのサンプル・プログラムには、Open Server エラー・ハンドラが含まれています。

イベント

この項では、次の内容について説明します。

- イベントとは
- イベント・ハンドラとは
- 標準イベント
- プログラマ定義のイベント
- 例

イベントとは

Open Server アプリケーションは、クライアントからの要求に応答します。これらの要求には、Server-Library イベントをトリガするものがあります。

すべてのイベントがクライアントのアクティビティによって引き起こされるわけではありません。アプリケーション自体が、ユーザ定義のイベント、および SRV_DISCONNECT、SRV_URGDISCONNECT、SRV_STOP の各イベントをキューイングするため、srv_event または srv_event_deferred ルーチンを呼び出します。srv_event ルーチンを使用してイベントを発生させる方法については、関連するページを参照してください。Open Server は、致命的なサーバ・エラーに応答すると SRV_STOP イベントをトリガすることもできます。Open Server では、サーバの開始処理の一部として SRV_START イベントが自動的に起動します。

イベントとは特定のコンテキストで発生するもので、特定のカテゴリのアクティビティに対応しています。たとえば、クライアントやリモート・サーバが接続しようとすると SRV_CONNECT イベントがトリガされるのに対して、クライアントからバルク・コピーが要求されると、Open Server は SRV_BULK イベントを発生させます。

Open Server には、標準イベントとプログラマ定義イベントの 2 種類のイベントがあります。標準イベントは、Open Server 内で定義されます。プログラマ定義イベントは、その名のとおり、アプリケーション内で定義されます。それぞれのイベントの詳細については、「標準イベント」(86 ページ) と「プログラマ定義のイベント」(90 ページ) を参照してください。

イベント・ハンドラとは

イベント・ハンドラとは、イベントを開始する場合に実行する一連のコードのことです。イベントをトリガすると、Open Server は、そのイベントとアクティブ・スレッドを実行キューに入れます。次に、そのスレッドは、イベントを処理するルーチンを実行します。このルーチンを「イベント・ハンドラ」と呼びます。

デフォルト・ハンドラとカスタム・ハンドラ

Open Server のデフォルトのイベント・ハンドラ・ルーチンは、個々の標準イベントに対して 1 つずつあり、さらに全プログラマ定義イベントに対して 1 つあります。デフォルト・ハンドラは、アプリケーション・プログラマが srv_handle ルーチンを使用してインストールするカスタム・イベント・ハンドラのプレースホルダです。デフォルト・ハンドラを使用しないアプリケーションでは、各カスタム・イベント処理ルーチンを定義してインストールしなければなりません。srv_handle (278 ページ) を参照してください。

イベント・ハンドラは、動的にインストールすることができます。次回のイベント発生時に、新しいイベント・ハンドラが呼び出されます。イベント・ハンドラは、成功した場合に CS_SUCCEED、失敗した場合には CS_FAIL を返します。現在、Open Server がリターン・コードをチェックする唯一のイベント・ハンドラは、SRV_START ハンドラです。SRV_START ハンドラから CS_FAIL が返されると、srv_run は、Open Server を開始しないでアプリケーションに CS_FAIL を返します。

カスタム・ハンドラのコーディング

アプリケーション・プログラマは、イベントへの応答方法を決定し、それに従ってイベント・ハンドラをコーディングします。イベント・ハンドラには、通常は、イベント・データを処理する呼び出しの標準セットが含まれます。それ以外のコードは、アプリケーション固有のものです。たとえば、パラメータだけでなくメッセージ・テキストも取得できるように、SRV_MSG イベント・ハンドラをコーディングします。ただし、特別なメッセージが取得された場合には、SRV_MSG イベント・ハンドラにコードを追加してユーザにメールを送信することもできます。

標準イベント

表 2-21 に、Open Server の標準イベントと、対応するカスタム・イベント・ハンドラに必要な引数を示します。対応するデフォルト・イベント・ハンドラが実行する関数についても説明しています。

表 *2-21:* イベントの説明

イベント	説明	ハンドラに対する引数	デフォルト・イベント・ハンドラ
SRV_ATTENTION	アテンションが受信された。このイベントは通常、クライアントが結果の処理を中断するために ct_cancel を呼び出したときに発生する。SRV_ATTENTION は即時イベントなので、このイベントが発生すると、Open Server はクライアントのイベント・キューに追加しないでただちに処理する。SRV_ATTENTION イベントは割り込みレベルで発生する。	SRV_PROC*	デフォルト・ハンドラは、追加動作を行わない。
SRV_BULK	クライアントがバルク・コピー要求を発行した。	SRV_PROC*	デフォルト・ハンドラは、“No bulk handler installed” というメッセージをクライアントに送信する。Open Server はバルク・データを廃棄し、DONE ERROR をクライアントに返す。
SRV_CONNECT	Client-Library クライアントが ct_connect を呼び出した。	SRV_PROC*	デフォルト・ハンドラは接続を受け付ける。
SRV_CURSOR	クライアントがカーソル要求を送信した。	SRV_PROC*	デフォルト・ハンドラは、“No SRV_CURSOR handler installed” というメッセージをクライアントに送信する。Open Server は DONE ERROR をクライアントに返す。

イベント	説明	ハンドラに対する引数	デフォルト・イベント・ハンドラ
SRV_DISCONNECT	クライアント接続を切断する要求が出された。これは、クライアントのサーバからの切断、Open Server の致命的なスレッド・エラー、SRV_STOP イベント、またはクライアントを切断するために明示的にアプリケーション内から出された srv_event の呼び出しによって、トリガされる。 Client-Library プログラムは、Open Server からログアウトするため、ct_close または ct_exit を呼び出す。リモート・プロシージャ・コールが完了すると、Adaptive Server Enterprise のリモート接続は終了する。	SRV_PROC*	デフォルト・ハンドラはアクションを起こさない。
SRV_DYNAMIC	クライアントが動的 SQL 要求を送信した。	SRV_PROC*	デフォルト・ハンドラは、“No SRV_DYNAMIC handler installed” というメッセージをクライアントに送信する。Open Server は DONE ERROR をクライアントに返す。
SRV_FULLPASSTHRU	接続に対するネットワーク読み込みが完了した。 (このイベントを発生させるため、スレッドの SRV_T_FULLPASSTHRU プロパティを CS_TRUE に設定しておくこと)	SRV_PROC*	このイベントに対するデフォルト・イベント・ハンドラはない。
SRV_LANGUAGE	クライアントから SQL 文などの言語要求が送信された。Client-Library クライアントが言語要求を送信するときは、ct_command や ct_send が使用される。isql などの対話型クエリ・ツールからも、Open Server アプリケーションに言語要求を送ることができる。	SRV_PROC*	デフォルト・ハンドラは、“No language handler installed” というメッセージと言語要求の最初の数文字をクライアントに送信する。Open Server は DONE ERROR をクライアントに返す。
SRV_LISTEN_PREBIND	指定された SRV_PROC 制御構造体で識別されたリスナの詳細な設定を許可する。たとえば、代替の SSL 証明書を指定できる。この設定は、スレッド・プロパティを使用して実行される。	SRV_PROC*	デフォルト・ハンドラはアクションを起こさない。

イベント	説明	ハンドラに対する引数	デフォルト・イベント・ハンドラ
SRV_LISTEN_POSTBIND	指定された SRV_PROC 制御構造体で識別されたリスナの最終的な設定を判断できる。たとえば、リスナがバインドされているアドレスを判断できる。設定は、スレッド・プロパティを使用して判断される。	SRV_PROC*	デフォルト・ハンドラはアクションを起こさない。
SRV_MIGRATE_STATE	このイベントは、マイグレーション・ステータスが SRV_MIG_READY または SRV_MIG_FAILED に移行すると必ずトリガされる。移行は、クライアントからマイグレーション・メッセージが送信された結果。 詳細については、「接続マイグレーション」(35 ページ) を参照。	SRV_PROC*	ステータスが SRV_MIG_READY の場合、デフォルト・ハンドラはアクションを起こさないため、クライアントがマイグレーションを続行できる。ステータスが SRV_MIG_FAILED に変わった場合は、エラーが記録される。
SRV_MIGRATE_RESUME	クライアントが結果の待機中に新しいサーバにマイグレートすると、クライアント接続が正常にマイグレートされた後、新しいサーバは SRV_MIGRATE_RESUME イベントを呼び出す。マイグレーション要求が失敗した場合やキャンセルされた場合、イベントはオリジナル・サーバから呼び出される。 詳細については、「接続マイグレーション」(35 ページ) を参照。	SRV_PROC*	デフォルト・ハンドラは、最終の完了 (SRV_DONE_FINAL) のみクライアントに送信して、結果を終了する。
SRV_MSG	クライアントがメッセージを送信した。	SRV_PROC*	デフォルト・ハンドラは、“No SRV_MSG handler installed” というメッセージをクライアントに送信する。Open Server は DONE ERROR をクライアントに返す。
SRV_OPTION	クライアントがオプション・コマンドを送信した。	SRV_PROC*	デフォルト・ハンドラは、“No SRV_OPTION handler installed” というメッセージをクライアントに送信する。Open Server は DONE ERROR をクライアントに返す。

イベント	説明	ハンドラに対する引数	デフォルト・イベント・ハンドラ
SRV_RPC	クライアントまたはリモート Adaptive Server Enterprise がリモート・プロシージャ・コール (RPC) を発行した。	SRV_PROC*	デフォルト・ハンドラは、“RPC < *rpcname* > received. No remote procedure call handler installed” というメッセージをクライアントに送信する。Open Server は DONE ERROR をクライアントに返す。
SRV_START	srv_run を呼び出すと、SRV_START イベントがトリガされる。Open Server アプリケーションは稼働状態。SRV_START イベント・ハンドラは、サーバ・リソースを初期化したり、サービス・スレッドを発生させるのに適している。	SRV_SERVER*	デフォルト・ハンドラはアクションを起こさない。
SRV_STOP	srv_event への呼び出し、または Open Server の致命的サーバ・エラーによって、Open Server アプリケーションを停止する要求がトリガされた。Open Server アプリケーションが停止する。srv_run から返されるものは、アプリケーションが SRV_STOP イベントを要求した場合は CS_SUCCEED、致命的サーバ・エラーによって SRV_STOP イベントが発生した場合は CS_FAIL。 このイベントのためのカスタム・ハンドラは、Open Server アプリケーションが終了する前に必要なクリーンアップをすべて実行できる。	SRV_SERVER*	デフォルト・ハンドラはアクションを起こさない。

イベント	説明	ハンドラに対する引数	デフォルト・イベント・ハンドラ
SRV_URGDISCONNECT	このイベントは、Open Server アプリケーションが srv_event を呼び出す場合にのみトリガされる。このイベントに応答して、Open Server は、スレッドの SRV_DISCONNECT ハンドラを呼び出す。Open Server は、スレッド・イベント・キューの最初にこのイベントを登録するので、このイベントは次のイベントとして処理される。 キューに入っている他のイベントをバイパスして、即時にスレッドを終了する場合には、アプリケーションがこのイベントを発生させる必要がある。SRV_URGDISCONNECT イベントが発生すると、そのスレッドに関連する I/O チャネルは、dead とマーク付けされる。	SRV_PROC*	デフォルト・ハンドラはアクションを起こさない。

プログラマ定義のイベント

アプリケーションは、srv_define_event を使用してプログラマ定義のイベントを定義し、srv_handle を使用してインストールします。新しいイベントをクライアントのイベント・キューに登録するには、srv_event または srv_event_deferred を呼び出す必要があります。

デフォルトのプログラマ定義イベント・ハンドラは、ハンドラがインストールされていないという内容のメッセージをクライアントに送ります。このメッセージには、イベントの番号と名前が含まれています。

プログラマ定義イベントを使用すると、Open Server アプリケーション内の他のスレッドにサービスを提供できます。たとえば、スレッドがトランザクションのログをディスク・ファイルに格納できるようになります。このような機能を設定するには、srv_define_event でイベントを定義し、ディスク・ファイルに書き込むハンドラ・ルーチンをインストールして、イベントがキューイングされるサービス・スレッドを作成します。このサービス・スレッドがトランザクション・ロギング・コードを提供します。

例

サンプル・プログラム *lang.c* には、簡単な SRV_LANGUAGE イベント・ハンドラの例が記述されています。

ゲートウェイ・アプリケーション

クライアントとしてもサーバとしても動作する Open Server アプリケーションは、「ゲートウェイ・アプリケーション」と呼ばれます。通常、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が直接的に通信できない場合の仲介役として機能します。

たとえば、Open Client アプリケーションは、Oracle データベースのエンジン部分とは直接通信することはできませんが、Oracle データベースへのゲートウェイ・サービスを行う Open Server アプリケーションとは通信できます。この場合、ゲートウェイは、Open Client アプリケーションに対するサーバとして、また Oracle データベースのエンジン部分に対するクライアントとして動作します。

もう 1 つの例としては、クライアントとリモート Adaptive Server がそれぞれまったく異なるネットワーク上で実行されているために、クライアントがリモート Adaptive Server Enterprise に直接アクセスできない場合が挙げられます。ゲートウェイ・サーバは、クライアントのデータを取得し、パケット化し直してリモート Adaptive Server Enterprise に送信することによって、この問題を解決します。Sybase のミラー・イメージ・クライアントおよびサーバ・ルーチンは、このプロセスを単純化します。サーバ・コンポーネントとクライアント・コンポーネントが、同一のデータ記述構造体を共有することもできます。つまり、ゲートウェイは Server-Library 呼び出しを使用して構造体にリモート・クライアントからの情報を格納し、次に、リモート・サーバと送信するために Client-Library 呼び出しまたは DB-Library 呼び出しを使用して同じ情報を構造体から抽出します。

Adaptive Server Enterprise または Open Server アプリケーションのクライアントとして動作するゲートウェイは、Client-Library または DB-Library ルーチンを使って、クライアントとして機能します。

Open Client アプリケーションのサーバとして動作するゲートウェイは、Server-Library ルーチンを使用して、サーバとして機能します。

警告！ Client-Library は、Open Server アプリケーションの完全な非同期モードでは実行できません。

サンプル・プログラム *ctos.c* は、「仮想 Adaptive Server Enterprise」ゲートウェイの一例です。このゲートウェイは、リモート Adaptive Server Enterprise から Sybase クライアントへのデータの渡し方を示します。

警告！ ゲートウェイ・アプリケーションでは、クライアントのルーチンは Open Server のプロセス、つまりスレッドのコンテキスト内で実行されます。この処理 (または Open Server アプリケーション全体) が終了する場合、まだ実行途中のクライアント・ルーチンは、予想しない結果を引き起こします。

パススルー・モード

Adaptive Server Enterprise を介して Open Server アプリケーションが Sybase クライアント・アプリケーションに接続するような特殊な場合には、Client-Library と DB-Library は、Open Server が内容に割り込まずにクライアントとサーバ間で TDS パケットを渡せるようにする一連のアプリケーション・プロトコル・パススルー・ルーチンを提供します。この処理は、到着した TDS 情報をアンパックし、送信する前に再びパケット化するよりも効率的です。サンプル *fullpass.c* には、このタイプのゲートウェイの例が記述されています。「パススルー・モード」(121 ページ) を参照してください。

注意 バージョン 10.0 より前の DB-Library を、Open Server バージョン 10.0 以降を使用するアプリケーションにリンクしないでください。ただし、10.0 以降の Open Server のクライアントとして動作するアプリケーション・プログラムの中で使用することはできます。

国際化のサポート

次に示すように、Open Server は国際化アプリケーションをサポートします。

- Open Server アプリケーションをローカライズできます。

  ローカライズされた Open Server アプリケーションでは、通常、次の処理が行われます。

  - 各国の言語と文字セットでのエラー・メッセージの生成
  - 各国の日時形式の使用
  - 文字列の変換および比較での、特定の文字セットと「照合順」(「ソート順」ともいう) の使用

- ローカライズされたクライアントをサポートします。

  ローカライズされたクライアントでは、ロケールに応じた言語、日時形式、文字セットが使用されます。これらは、Open Server アプリケーションの言語、日時形式、文字セットとは異なる場合があります。ローカライズされたクライアントをサポートするには、Open Server アプリケーションは入力データをアプリケーション自体の言語と文字セットに変換するだけでなく、出力メッセージやデータをクライアントの言語と文字セットに変換しなければなりません。

ここでは、次の内容について説明します。

- Open Server アプリケーションのローカライゼーション
- ローカライズされたクライアントのサポート
- ローカライゼーションに関連するクライアント要求
- ローカライゼーション・プロパティ
- ローカライゼーションのプログラム例

Open Client/Server のローカライゼーションについては、『Open Client/Server 開発者用国際化ガイド』に記載されています。このマニュアルを参照して、Server-Library のローカライゼーション・メカニズムとローカライゼーションに環境変数がどのように影響するかを理解してください。

使用しているプラットフォームのローカライゼーションに関する情報については、『Open Client/Server 設定ガイド』を参照してください。

Open Server アプリケーションのローカライゼーション

Open Server アプリケーションのローカライゼーションによって、次の内容が決まります。

- エラー・メッセージを生成する言語と文字セット

  注意　SRV_S_USESRVLANG プロパティと SRV_T_USESRVLANG プロパティを使用して指定された言語は、エラー・メッセージ生成時に、サーバの言語よりも優先されます。

- すべてのデータ操作に使用される文字セットと照合順

Open Server アプリケーションは初期ローカライゼーション値やカスタム・ローカライゼーション値、またはその両方を使用することもできます。

通常、国際化された Open Server アプリケーションは、LC_ALL および LANG 環境変数、またはロケール・ファイル内の「デフォルト」のエントリによって特定された初期ローカライゼーション値を使ってローカライズが行われます。

初期ローカライゼーション値は実行時に確定されます。このとき、Open Server アプリケーションが CS-Library ルーチン cs_ctx_alloc を呼び出して CS_CONTEXT 構造体を割り付けます。アプリケーションがこの呼び出しをする場合、CS-Library は最初のローカライゼーション情報を新しいコンテキスト構造体にロードします。

初期ローカライゼーション値がアプリケーションの要求を満たさない場合、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構造体にカスタム・ローカライゼーション値を設定できます。「CS_LOCALE 構造体を使用したカスタム・ローカライゼーション値の設定」(94 ページ) を参照してください。

ローカライズされたクライアントのサポート

Open Server アプリケーションによっては、ローカライズされたクライアントに対する最初のローカライゼーション値で十分な場合もあります。このような Open Server アプリケーションでは、ローカライズされたクライアントをサポートするために、それ以上の手順は必要ありません。

しかし、ローカライズされたクライアントに対して追加のサポートが必要な Open Server アプリケーションもあります。特に、次のような場合は、Open Server アプリケーションには、ローカライズされたクライアントをサポートするための追加の手順が必要です。

- CS-Library エラー・メッセージをクライアントに送り返す場合

  この場合、Open Server アプリケーションは、CS-Library がクライアントの言語と Open Server アプリケーションの文字セットでメッセージを生成していることを確認する必要があります。

  この実行方法の詳細については、「クライアントに対する CS-Library メッセージのローカライズ」(95 ページ) を参照してください。

- ゲートウェイとして機能する場合

  この場合、Open Server アプリケーションは、リモート・サーバへの接続がクライアントの言語と Open Server アプリケーションの文字セットを使用していることを確認する必要があります。

  この実行方法の詳細については、「ゲートウェイ・アプリケーションに対するローカライズされた接続の作成」(96 ページ) を参照してください。

- クライアント・アプリケーションが言語または文字セットの変更を要求した場合

  この場合、Open Server アプリケーションは、クライアント・スレッドに対する言語または文字セットを変更する必要があります。

  この実行方法の詳細については、「言語と文字セットの変更」(97 ページ) を参照してください。

CS_LOCALE 構造体を使用したカスタム・ローカライゼーション値の設定

クライアントが Open Server アプリケーションに接続するときに、Open Server は、クライアントの言語と文字セットを反映する CS_LOCALE 構造体を作成します。たとえば、french/cp850 のクライアントが us_english/iso_1/binary の Open Server アプリケーションにログインする場合、Open Server アプリケーションはこの接続に対する french/cp850 の CS_LOCALE 構造体を作成します。

Open Server プログラマは、cs_locale を呼び出して、新しく割り当てた CS_LOCALE 構造体にこの情報をコピーし、構造体内の情報を使用できます。

srv_version を呼び出す前に、アプリケーションワイドなコンテキスト構造体にカスタム・ローカライゼーション情報をインストールできます。このために、アプリケーションは次のことを行います。

1　cs_loc_alloc を呼び出して CS_LOCALE 構造体を割り付けます。

2　type を CS_LC_ALL に設定して cs_locale を呼び出し、CS_LOCALE にカスタム・ローカライゼーション値をロードします。type が CS_LC_ALL に設定されると、内部的に一貫性のあるローカライゼーション値が CS_LOCALE にロードされます。

3　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アプリケーションのコンテキスト構造体にコピーします。

4　cs_loc_drop を呼び出して、CS_LOCALE の割り付けを解除します。

クライアントに対する CS-Library メッセージのローカライズ

Open Server アプリケーションが、アプリケーション自体のコンテキスト構造体をパラメータとして使用して CS-Libary ルーチンを呼び出すと、呼び出しの結果として CS-Library が生成するエラー・メッセージでは、Open Server アプリケーションの言語と文字セットが使用されます。

たとえば、cs_convert 呼び出しのコンテキスト・パラメータが us_english/iso_1 を示している場合、cs_convert 呼び出しが失敗すると、CS-Library は us_english/iso_1 のメッセージを生成します。

注意　CS-Library ルーチンのパラメータとして CS_LOCALE 構造体が指定されると、この構造体の中のローカライゼーション値によってコンテキスト・パラメータの中のローカライゼーション値が上書きされます。

Open Server アプリケーションの言語と文字セットで CS-Library メッセージを取得できるのは、Open Server アプリケーションが CS-Library メッセージのログを取るか、または CS-Library メッセージを保持する場合のみです。

しかし、Open Server アプリケーションが CS-Library のエラー・メッセージをクライアントに送り返す場合、CS-Library はクライアントの言語と Open Server アプリケーションの文字セットでメッセージを生成します。

メッセージは、クライアントが理解できるクライアントの言語で生成される必要があります。

次の 2 つの理由から、メッセージは Open Server アプリケーションの文字セットでなければなりません。

- Open Server アプリケーションは、通常は、すべてのメッセージをログ・ファイルに記録します。したがって、ログに記録されたすべてのメッセージが同一の文字セットを使用することが重要です。

- Open Server は、メッセージなどの出力データに対して自動的に文字セット変換を行います。Open Server の文字セットで生成されたメッセージは、クライアントの文字セットに正しく変換されます。

アプリケーションは、クライアント・スレッドごとに正しくローカライズされた CS_CONTEXT 構造体を設定することによって、メッセージを正しい言語および文字セットで確実に生成します。また、クライアントの代わりに CS-Library ルーチンを呼び出すときに、これらの CS_CONTEXT 構造体を使用できます。

CS_CONTEXT 構造体のローカライズ方法については、「CS_CONTEXT 構造体のローカライズ」(97 ページ) を参照してください。

ゲートウェイ・アプリケーションに対するローカライズされた接続の作成

Open Server アプリケーションがゲートウェイとして機能している場合は、クライアントの言語と Open Server の文字セットを使用してリモート・サーバに接続されるようにする必要があります。

注意 Open Server の文字セットは、リモート・サーバの文字セットと同じである必要はありませんが、リモート・サーバがそのサーバ自体の文字セットに変換可能なものである必要があります。

Adaptive Server Enterprise は、2 つの西欧文字セット間での変換や 2 つの日本語文字セット間での変換はできますが、西欧文字セットから日本語文字セットへの変換はできません (また、その逆の日本語文字セットから西欧文字セットへの変換もできません)。

たとえば Adaptive Server Enterprise は、ともに西欧言語グループの文字セットである ISO 8859-1 と CP850 との間では変換できますが、西欧言語グループの文字セットである ISO 8859-1 と、東欧言語グループの文字セットである CP 1250 との間での変換は実行できません。

Open Server では、同じ言語グループの文字セットであるかどうかに関係なく、サポートされている 2 つの文字セット間での変換ができます。ただし、異なる言語グループの文字セット間での変換を行う場合は、アルファベット以外の文字は失われる可能性があります。

このためのアプリケーションにとって一番簡単な方法は、クライアント接続ごとに正しくローカライズされた CS_CONTEXT 構造体を設定し、ローカライズされたコンテキストのもとでクライアントに対するリモート接続を割り当てることです。

CS_CONTEXT 構造体のローカライズ方法については、次の CS_CONTEXT 構造体のローカライズを参照してください。

CS_CONTEXT 構造体のローカライズ

Open Server アプリケーションで、クライアント・スレッドに対する CS_CONTEXT 構造体を正しくローカライズするには、次の手順に従います。

1　cs_ctx_alloc を呼び出し、クライアント・スレッドに対する CS_CONTEXT を割り付けます。

2　cs_loc_alloc を呼び出し、CS_LOCALE 構造体を割り付けます。

3　srv_thread_props を呼び出し、クライアント・スレッドの既存の CS_LOCALE 構造体をコピーします。これによって、新しい CS_LOCALE はクライアントの言語と文字セットで設定されます。

4　type を CS_SYB_CHARSET に設定して cs_locale を呼び出し、クライアントの文字セットを Open Server の文字セットに置き換えます。

5　*property* を CS_LOC_PROP に設定して cs_config を呼び出し、ローカライゼーション情報を CS_LOCALE から CS_CONTEXT にコピーします。

6　必要に応じて、cs_loc_drop を呼び出し、CS_LOCALE の割り付けを解除します。アプリケーションは、必要に応じて cs_locale を呼び出し、CS_LOCALE 構造体を再利用してローカライゼーション値を変更できます。

クライアント要求に対する応答

クライアントは、次のことを要求できます。

- 言語と文字セットの変更
- ローカライゼーション情報

言語と文字セットの変更

クライアントは、Open Server に接続するときに、ログイン・レコード内で言語と文字セットを指定します。Open Server は、この情報を使用して、クライアント・スレッドに対する CS_LOCALE と「文字セット変換ルーチン」を設定します。

Open Server は、自動的にこの処理を行います。Open Server アプリケーションは、ログイン時にローカライズされたクライアントを処理する必要はありません。

ただし、ログインした後でも、クライアントは言語と文字セットを変更できます。クライアントから言語または文字セットの変更要求が送信されると、Open Server アプリケーションはクライアント・スレッドの CS_LOCALE 構造体で要求された変更を実行します。

クライアントは、次の 2 つの方法で言語または文字セットの変更を要求できます。

- ct_command を使用して送信される、言語に基づいたオプション・コマンドを使用する。このタイプのコマンドは SRV_LANGUAGE イベントをトリガします。その結果、Open Server アプリケーションは SRV_LANGUAGE イベント・ハンドラの内部で要求を処理します。
- ct_options を使用して送信される、オプション・コマンドを使用する。このタイプのコマンドは SRV_OPTION イベントをトリガします。その結果、Open Server アプリケーションは SRV_OPTION イベント・ハンドラの内部で要求を処理します。

どちらの場合も、Open Server アプリケーションは次の方法で応答します。

1 新しい言語または文字セットを使用して、CS_LOCALE 構造体を設定します。

2 property を SRV_T_LOCALE に設定して srv_thread_props ルーチンを呼び出し、スレッド接続の言語または文字セットを変更します。

表 2-22 は、クライアント・スレッドの言語または文字セットを変更する方法を示します。

表 2-22: 言語または文字セットの変更

手順	アプリケーションの手順	目的	詳細
1	cs_loc_alloc を呼び出す。	CS_LOCALE 構造体を割り付ける。	この呼び出しは、Open Server アプリケーション・コンテキストの現在のローカライゼーション情報を新しい CS_LOCALE 構造体にコピーする。
2	*property* を SRV_T_LOCALE に設定して srv_thread_props(GET) を呼び出す。	クライアント・スレッドの既存のローカライゼーション値を新しい CS_LOCALE 構造体にコピーする。	
3	cs_locale を呼び出す。	要求された言語または文字セットを使用して CS_LOCALE 構造体を上書きする。	「CS_CONTEXT 構造体のローカライズ」(97 ページ) を参照。
4	property を SRV_T_LOCALE に設定して srv_thread_props(SET) を呼び出す。	新しい言語または文字セットを使用して、クライアント・スレッドを設定する。	
5	オプションで cs_loc_drop を呼び出す。	CS_LOCALE 構造体の割り付けを解除する。	アプリケーションは、CS_LOCALE 構造体の割り付けを解除する前に構造体を再使用できる。 構造体のローカライゼーション値を再利用する前に、必要に応じてアプリケーションは cs_locale を呼び出して、この値を変更できる。

ローカライゼーション情報の要求

ログインした後、クライアントは次の情報を要求できます。

- サーバの文字セットの名前
- サーバのソート順の名前
- クライアント文字セットのための文字セットの定義
- クライアントのソート順のためのソート順の定義

クライアントは、RPC コマンドを使用し、sp_serverinfo システム・レジスタード・プロシージャを介してこれらの情報を要求します。

Open Server からの応答として、sp_serverinfo システム・レジスタード・プロシージャを介して要求された情報が自動的に送り返されます。このとき、Open Server アプリケーションは何も行う必要はありません。実際には、Open Server アプリケーションは、要求が送られたことを認識していません。

「レジスタード・プロシージャ」(151 ページ) を参照してください。

ローカライゼーション・プロパティ

ローカライゼーションには 2 つのプロパティが関係します。

- SRV_S_USESRVLANG
- SRV_T_USESRVLANG

これらのプロパティは、Open Server がエラー・メッセージを Open Server アプリケーションの言語で生成するかクライアントの言語で生成するかを指定します。

SRV_S_USESRVLANG はサーバワイドなプロパティであり、srv_props によって設定されます。この値は、SRV_T_USESRVLANG のデフォルト値として使用されます。

SRV_T_USESRVLANG はスレッドのプロパティであり、srv_thread_props によって設定されます。新しいスレッド構造体が割り当てられたとき、SRV_T_USESRVLANG は SRV_S_USESRVLANG からデフォルト値を抽出します。

SRV_T_USESRVLANG が CS_TRUE の場合、Open Server はエラー・メッセージをサーバの言語で生成します。

SRV_T_USESRVLANG が CS_FALSE の場合、Open Server はエラー・メッセージをクライアントの言語で生成します。

「プロパティ」(130 ページ) を参照してください。

ローカライゼーションの例

サンプル *ctos.c* には、CS_LOCALE 構造体のカスタマイズ方法の一例が記述されています。サンプル *intlchar.c* は、文字セットと各国言語の設定、およびクエリを処理します。

言語呼び出し

Open Server は、言語イベントを柔軟に処理する機能を備えています。クライアント・アプリケーションが、type 引数を CS_LANG_CMD に設定した ct_command を使って情報を送ると、SRV_LANGUAGE イベントがトリガされます。RPC ストリームが名前とパラメータという個別の要素から構成されているのに対して、言語情報は区別のない文字ストリームとして着信します。SRV_LANGUAGE イベント・ハンドラには、その文字ストリームを有効なコンポーネントに解析できるコードが必要がります。SQL クエリは、言語ストリームの一例です。

これは、自然言語で入力を受け入れるアプリケーションにとって有効な機能です。たとえば、SQL データベースに対してユーザが英語でクエリを実行できる衣料店アプリケーションがあり、販売員が “How many shirts in blue?” と入力したとします。この場合、フロントエンド・クライアント・アプリケーションは、ct_command を呼び出して、この自然言語によるクエリを Open Server ゲートウェイ・アプリケーションに送信します。SRV_LANGUAGE ハンドラは、このテキストを解析して下記の Transact-SQL クエリを作成し、クエリをリモート・データベースに送信します。

```
select quantity from inventory_tab where color = “blue” and type = “shirt”
```

SRV_LANGUAGE イベント・ハンドラは、次の手順で言語データを処理する必要があります。

1 srv_langlen を呼び出して、言語要求バッファの長さを取得します。

2 srv_langlen から返された長さに NULL 終了バイト分の 1 バイトを加算した大きさを持つ、ローカル・アプリケーション・バッファを割り付けます。

3 srv_langcpy を呼び出し、要求データの全部または一部をローカル・バッファにコピーします。

4 ローカル・バッファの内容を処理します。

ログイン・リダイレクトと拡張 HA フェールオーバのサポート

ログイン・リダイレクトと拡張 HA フェールオーバのサポートにより、サーバのクラスタで、すべての着信クライアント接続を対象に負荷のバランスをとることができます。

srv_send_ctlinfo、srv_getserverbyname、srv_freeserveraddrs という 3 つの API ルーチンがこの機能をサポートします。

srv_send_ctlinfo ルーチンは、ログイン・リダイレクトと拡張 HA フェールオーバの両方をサポートしますが、srv_getserverbyname と srv_freeserveraddrs は、Open Server アプリケーションでサーバ名を接続情報に変換できるようにします。これらのルーチンの詳細については、「srv_send_ctlinfo」(362 ページ)、「srv_getserverbyname」(277 ページ)、「srv_freeserveraddrs」(265 ページ) を参照してください。

次のプロパティがこれらのルーチンをサポートします。

- SRV_S_HASERVER は読み込み専用のサーバ・プロパティで、interfaces ファイルから HAFAILOVER 値を返します。返される値は、srv_init によって設定されたサーバ名に対応します。
- SRV_T_REDIRECT は読み込み専用のスレッド・プロパティで、ログイン・レコード内の TDS_HA_LOG_REDIRECT ビットの設定を返します。
- SRV_T_HA はスレッド・プロパティで、ログイン・レコードから HA 関連情報の設定を CS_INT ビットマスクとして返します。提供される情報には、セッション (SRV_HA_LOGIN)、フェールオーバ (SRV_HA_LOGIN_FAILOVER)、再開 (SRV_HA_LOGIN_RESUME) の各ビットが含まれます。
- CS_SESSIONID は、セッション ID を保持する型定義です。
- SRV_T_SESSIONID は、クライアントが Open Server に送信するセッション ID をログイン・レコードで返します。

  SRV_T_SESSIONID を使用しても、SRV_CONNECT ハンドラでセッション ID をクライアントに送信できます。「クライアントに対する異なるサーバへのマイグレーションの指示」(42 ページ) を参照してください。
- SRV_NEG_SESSIONID は、srv_negotiate の語法でネゴシエートされたログイン情報の一種で、クライアントのセッション ID 情報の送信をサポートします。

メッセージ

Open Server では、次の 3 つのタイプのメッセージがあります。

- データ・ストリーム・メッセージ – クライアントおよびサーバは、データ・ストリーム・メッセージを使用して情報を交換できます。「データ・ストリーム・メッセージ」(72 ページ) を参照してください。
- スレッド・メッセージ – スレッドは、スレッド・メッセージを使用して情報を交換できます。「マルチスレッド・プログラミング」(102 ページ) を参照してください。
- エラー・メッセージ – Open Server は、エラー・メッセージを使用してエラー状態を通知します。「エラー」(81 ページ) を参照してください。

マルチスレッド・プログラミング

Open Server は、マルチスレッド・アーキテクチャを使用します。マルチスレッド・サーバ・アプリケーションは、「スレッド」の集まりとして動作し、それぞれのスレッドが独自のタスクを達成するためにルーチンを実行します。

スレッドについて

スレッドは、Open Server アプリケーションを通した、特定の実行の道筋と考えることができます。各クライアントは、スレッドを使用して接続を管理し、イベント・ハンドラとプロシージャを呼び出して要求に応答します。Open Server ランタイム・システムには、メッセージの配信、サーバ対サーバ通信の処理、タスクのスケジュールなどのサーバ・アクティビティを管理するスレッドが複数あります。アプリケーションは、他のアプリケーション固有のアクティビティのためにサーバ・スレッドを生成できます。

マルチスレッド・システムとしての Open Server アプリケーションでは、スレッドが実行するさまざまなアクティビティのスケジューリング、共有リソースへのスレッドのアクセスのネゴシエーション、スレッドが相互通信するための手段の提供を行う必要があります。「スケジューリング」(106 ページ) および「ツールと手法」(108 ページ) を参照してください。

スレッドの種類

Open Server では、プリエンプティブ、イベント駆動型、サービス、サイト・ハンドラの 4 種類のスレッドが使用されます。

プリエンプティブ・スレッド

Open Server バージョン 12.5 以上では、すべてのプラットフォームに対してプリエンプティブ・スレッドが導入されました。スレッド・ライブラリを使ってアプリケーションを作成する場合の事前の注意点について以下に説明します。

スレッドセーフな関数

アプリケーションがリエントラントであることを確認するため、次の点を確認してください。

- 提供される C ライブラリ関数のリエントラント・バージョンを使用している。
- 非リエントラント C (またはその他の) ライブラリ関数を安全な方法で使用している。
- ミューテックス・ロック、またはその他のロックを使用して、グローバル変数と共有構造体を保護している。
- アプリケーションで、静的バッファを指すポインタを返す関数を使用していない。
- 正しいプロセッサ・フラグとリンカ・ディレクティブを使用してコンパイルを実行する。

注意　ある UNIX システムでリエントラントである C ライブラリ関数が、他の UNIX システムでもリエントラントであるとは限りません。ご使用のプラットフォームの移植に関するガイドを参照して、C 関数がリエントラントであるかどうかを確認してください。

スレッドセーフ・コードとプリエンプティブ・モード

一度に複数の Open Server スレッドを実行できます。また、1 つのスレッドを他のスレッドのためにプリエンプティブにできます。この操作によって同時実行性が向上します。特に、SMP システムの場合に効果があります。ただし、コードがスレッドセーフである必要があります。これは Open Server コード、ユーザのイベント・ハンドラ、コールバック関数に適用されます。

SRV_S_PREEMPT の動作

SRV_S_PREEMPT を CS_TRUE に設定した場合、複数の Open Server スレッドを同時に実行できます。また、オペレーティング・システムにより、これらのスレッドは相互にプリエンプティブになります。これらのスレッドは、バインドされません。

SRV_S_PREEMPT が CS_FALSE に設定されている場合は、ある Open Server スレッドの実行に割り込みをかけて別の Open Server スレッドを実行することはできません。また、2 つの Open Server スレッドを同時に実行することもできません。

また、スレッド・ライブラリとともに使用する場合、SRV_S_PREEMPT を CS_TRUE または CS_FALSE のいずれに設定しても、SRV_S_CURTHREAD の一部の関数は無効になります。これは、SRV_S_PREEMPT の設定にかかわらず、スレッド・ライブラリがシグナル処理スレッドにより処理されるシグナルを使用するためです。

Open Server スレッドの実行が再開されると、1 つのミューテックスが有効化されます。SRV_C_SUSPEND コールバックの実行後、Open Server が特定のタスクを実行できる状態になったときに、ミューテックスが解放されます。この場合、サーバワイドのミューテックスは 1 つだけ存在します。

オペレーティング・システムがこのようなスレッドを再開しても、SRV_C_RESUME と SRV_C_SUSPEND コールバック関数は呼び出されません。これらの関数は、特定の Open Server スレッドが実行を停止または再開したときにのみ呼び出されます。たとえば、ユーザ Open Server スレッドの言語要求が着信し、言語イベント・ハンドラの実行後、このスレッドがスリープ状態になる前の時点などです。

実装固有の事項

ほとんどの UNIX プラットフォームでは、スレッドは POSIX スレッドに基づいており、バインド解除されています。HP および Linux では、スレッドはバインドされています。Windows では、スレッドは Win32 スレッドです。

プラットフォームでのスレッドの使用方法の詳細については、ご使用のプラットフォームに付属の各ベンダーのマニュアルを参照してください。

イベント駆動型スレッド

クライアント接続を制御するスレッドは、イベント駆動型です。アクションを必要とする要求は、サーバ・イベントをトリガします。イベントの詳細については、「イベント」(84 ページ) を参照してください。

クライアント・イベントが発生すると、そのイベントは Open Server によってスレッドのイベント・キューに登録されます。スレッドの次回実行時には、イベント・キューから次のイベント要求を読み込みます。Open Server は、そのイベントに関連しているイベント・ハンドラを呼び出します。ハンドラが復帰すると、スレッドはキューにある次のイベントを読み込もうとします。イベントがない場合は、スレッドは「スリープ」します。

たとえば、クライアント・アプリケーションがサーバにログインしようとするときには、ランタイム・システムは接続を処理するスレッドを作成し、そのスレッドのキューに SRV_CONNECT イベントを登録します。このスレッドの実行時には、SRV_CONNECT イベントを処理するためにインストールされているルーチンが実行されます。デフォルト・ハンドラは、単に接続を受諾するだけです。SRV_CONNECT のカスタム・ハンドラをインストールすることもできます。この場合は、ログイン名とパスワードをチェックして両方とも有効なときにユーザのログインを許可します。

イベント駆動型のスレッドは主にクライアント要求を処理するために存在しますが、サーバ内でサービス・ルーチンを実行するためにはユーザ定義イベントと併せて使用することもできます。

サービス・スレッド

どのクライアント接続からも独立して動作する Open Server スレッドを作成できます。このようなスレッドがサービス・スレッドと呼ばれるのは、実行するルーチンが、通常は、イベント駆動型のクライアント・スレッドに対してサービスを提供するからです。クライアント・スレッドとは違って、サービス・スレッドはイベントによって起動しません。その代わり、スレッド作成時にスレッド用の実行ルーチンを設定します。このルーチンは、サーバによってすぐに実行キューに登録されます。あるルーチンを実行するために作成されたサービス・スレッドは、そのルーチンが戻ったところで、消滅します。

アプリケーションは、Open Server でさまざまなタスクを実行するためにサービス・スレッドを使用できます。実際、Open Server ランタイム・システムは、サーバ管理ルーチンを実行するサービス・スレッドから構成されています。サービス・スレッドは、クライアント I/O、つまりクライアント・コマンドを読み込んで結果を返すことはできません。

Open Server は、イベントがトリガされたときに動作させるイベント・コードをスケジューリングします。それとは対照的に、アプリケーションは srv_wakeup、srv_sleep、srv_yield ルーチンを使って明示的にサービス・スレッド・コードをスケジューリングする必要があります。さらに、メッセージ・キューのスケジュールについては、プリエンプティブ・モードで動作していないときに行う必要があります。

サイト・ハンドラ・スレッド

Adaptive Server Enterprise が Open Server アプリケーションに接続すると、Open Server はサイト・ハンドラ・スレッドを作成します。

Open Server アプリケーションがサーバ対サーバ RPC を受け取ると、Open Server は SUB-PROC を作成します。SUB-PROC は、サーバ対サーバ RPC が完了すると消えます。サイト・ハンドラ・スレッドは、Adaptive Server Enterprise が Open Server アプリケーションへの接続をクローズすると消えます。

Open Server アプリケーションがサイト・ハンドラ・スレッドにアクセスするのは、SRV_CONNECT または SRV_DISCONNECT イベント・ハンドラ内だけです。このイベント・ハンドラ以外では、サイト・ハンドラ・スレッドは完全に内部的に使用されます。

スケジューリング

Open Server は、実行中のスレッドを定期的に中断して他のスレッドを再開する方法で、同時実行性を実現します。このコンテキスト切り替えは頻繁にしかも迅速に行われるため、Open Server クライアントからはスレッドが連続的に動作しているように見えます。

スケジューラは、コンテキスト切り替えを実行するランタイム・システムのスレッドです。スレッドに、スタックとマシン・レジスタ環境を含む実行コンテキストがあります。スケジューラは、実行中のスレッドの実行コンテキストを保存し、再開するスレッドを選択し、保存したコンテキストをリカバリしてから、コンテキストを実行します。スケジューラは透過的に動作しますが、Open Server コードを書くには、次のことを理解しておく必要があります。

- スケジューラがどのように呼び出されるか (スケジューリング方法)
- スケジューラがどのようにして再開するスレッドを選択するか

スケジューリング方法

スケジューリング方法によって、1 つの実行中スレッドから別のスレッドに制御が移行される時期を決定します。Open Server アプリケーションでは、「非プリエンプティブ」と「プリエンプティブ」の 2 つのスケジューリング方法のいずれかを使用します。デフォルトでは、非プリエンプティブです。ほとんどのプラットフォームでは非プリエンプティブの方法しか使用できません。

非プリエンプティブ・スケジューリング

非プリエンプティブ・スケジューリングでは、コンテキストの切り替えを予測できます。次に示す状況でのみ切り替えが行われます。

- スレッドが、ネットワーク I/O を行っている Server-Library または Client-Library ルーチンを呼び出す。

  スレッドがネットワーク接続に対して読み出しまたは書き込みを行った場合は、ランタイム・システムは読み込みまたは書き込みの完了を待っているスレッドの実行を中断します。ネットワーク I/O は比較的遅いため、サーバは I/O が完了する間に他のスレッドを実行することによって、時間をより有効に使用することができます。

- スレッドが、実行が再開されるのを待機してスリープする。

  たとえば、他のスレッドが共有メモリ内のデータ・オブジェクトを更新している場合、そのオブジェクトにアクセスするには、スレッドは更新が終了するまで待機している必要があります。アプリケーションが次のルーチンを呼び出すと、スレッドはスリープします。

  - srv_sleep
  - srv_getmsgq(SRV_M_WAIT) や srv_lockmutex のように、要求されたリソースを待機している間はスレッドがスリープする Server-Library ルーチン

- スレッドがそれ自体を故意に中断させ、他のスレッドを実行可能にするために srv_yield を呼び出す。そのスレッドは実行可能のままであり、srv_yield 呼び出し後の文でオペレーションを再開する。スリープしない、またはネットワーク I/O を実行しない、処理時間の長いルーチンを作成する場合は、ルーチンがサーバを独占しないように定期的に srv_yield を呼び出してください。

プリエンプティブ・スケジューリング

プリエンプティブ・スケジューリングでは、上記のイベントのいずれかが起きた場合、また、システムが実行中のスレッドに割り込むときに、コンテキスト切り替えが起こる可能性があります。プリエンプティブ・スケジューリングはオペレーティング・システムのスレッド管理機能に依存しているので、システムが開始する切り替えは、予測できません。通常、オペレーティング・システムには、確実にスレッド間での効率的な時間配分を行う複雑なアルゴリズムがあります。

property を SRV_S_PREEMPT に設定した srv_props ルーチンを使用することによって、プリエンプティブ・スケジューリングを選択できます。プリエンプティブ・スケジューリングは、すべてのプラットフォームで使用できるわけではありません。使用しているアプリケーションのプラットフォームでプリエンプティブ・スケジューリングが使用可能かどうかを確認するには、srv_capability を呼び出します。

再開するスレッドの選択

Open Server は、複数の実行キューを保持します。実行キューとは、中断されているがスリープはしていないスレッドのリストです。各キューには、同じ実行優先順位を持つスレッドがあります。スケジューラは、最高優先順位のキューの中の、最も長い期間そのキューにあったスレッドを再開させます。通常は、スレッドは同じ優先順位で実行されるので、この選択方法では先入れ先出しを基本に実行時間を割り当てます。

スレッドの優先順位を調節して、他のスレッドより先にスケジューラがまずそのスレッドを実行するようにしたり、他に実行するスレッドがない場合にのみそのスレッドが実行されるようにすることができます。たとえば、リアルタイム・データを読み込むスレッドについては、処理するデータが存在する場合はいつでも実行できるように、高い優先順位を設定できます。優先順位の調節には、注意が必要です。他のスレッドよりも高い優先順位を持つスレッドが実行可能な状態にある場合は、スケジューラはこのスレッドを引き続き実行します。この高い優先順位が変わることなくスレッドもスリープすることがない場合は、それよりも低い優先順位を持つ他のスレッドはまったく実行されません。スレッドの優先順位の調節については、srv_setpri (388 ページ) を参照してください。

Open Server が新しいスレッドを確立するときに、そのスレッドが他のスレッドと CPU 時間を完全に共有できるようにするには、その前にスケジューラがある作業を行う必要があります。この新しいスレッドの起動中に、スケジューラは、既存のスレッドが実行できるようにする一連の内部 srv_yield 呼び出しを効率的に実行します。その結果、確立済みの実行可能なスレッドが「CPU を占有」して、新しいスレッドの起動を遅らせるように見える場合があります。スレッドが確立されて実行可能になると、そのスレッドの優先順位に応じて CPU 時間を共有します。

実行優先順位が問題となるのは、非プリエンプティブ・モードで動作している Open Server アプリケーションの場合のみです。

ツールと手法

マルチ・スレッド環境でプログラムを作成するには、常にスレッド間の対話に気を配る必要があります。この環境で特に役立つプログラミング・ツールや方法があります。Open Server は、共有リソースへのアクセスを制御するための相互排除セマフォ (ミューテックス)、スレッドが互いに調整し通信することを可能にするメッセージ・キューを提供しています。

ミューテックス

相互排除セマフォ、つまり「ミューテックス」は、Open Server の論理オブジェクトの 1 つで、最大でも 1 個のスレッドしかこのオブジェクトをロックすることができません。ミューテックスは、共有リソースを保護したり、より複雑なツールを開発したりするのに便利です。

ミューテックスの使用方法を理解するには、次の点に注意してください。

UNIX プラットフォーム上で動作している Open Server アプリケーションにおいては、標準入力と標準出力はどのスレッドでも同じです。スレッドが定期的に標準出力に書き込む場合は、アプリケーションのコード上では、標準出力に対して複数のスレッドの出力を指定しないようにしてください。

スレッドの出力が混合することを避ける 1 つの方法は、*stdout* デバイスにミューテックスを関連付けて、スレッドが *stdout* に書き込む前に必ずミューテックスをロックさせることです。一度に 1 つのスレッドしかミューテックスをロックできないので、*stdout* に書き込むことができるスレッドは一度に 1 つだけになります。他のスレッドは、そのミューテックスをロックすることが可能になるまで待機します。

プログラミングの詳細については、srv_createmutex、srv_lockmutex、srv_unlockmutex、srv_deletemutex のページを参照してください。

メッセージ・キュー

メッセージ・キューによって、スレッド間の通信が可能になります。メッセージ・キューは、通常、他のスレッドにサービスを行う生成スレッドにデータを送るために使用されます。たとえば、すべてのスレッドは、宛先がログ・ファイルになっているデータを入れるためのメッセージ・キューを作成できます。生成されたスレッドは、キューからメッセージを読み込み、受け取った順にログ・ファイルに書き込むことができます。

メッセージ・キューのメッセージは 4 バイトの値で、通常は、送受信スレッドが共有するメモリ内にあるデータのアドレスを指すポインタです。キューにメッセージを入れるスレッドと、そのメッセージを読むスレッドのメッセージ・フォーマットは、同じである必要があります。

メッセージが他にデータを参照する場合は、メッセージを送ったスレッドがデータ領域を更新する前に、メッセージを読むスレッドがデータを終了していることを確認する必要があります。メッセージが受け取られる前に、送信ルーチンがメッセージを上書きまたは解放することがないように、メッセージを書き込むルーチンである srv_putmsgq には、メッセージがキューから読み込まれるまで送信スレッドをスリープさせるオプションがあります。

プログラミングの詳細については、srv_createmsgq、srv_putmsgq、srv_getmsgq、srv_deletemsgq のページを参照してください。

クリティカル・セクションの保護

Open Server がスレッドを中断しないようにするために、srv_setpri を呼び出して、一時的にスレッドの優先順位を上げることができます。サーバ・スレッドは、すべて、同一の優先順位で開始されます。この優先順位は、*ospublic.h* に定義されている SRV_C_DEFAULTPRI 定数によって表されます。スレッド優先順位の範囲は SRV_C_LOWPRIORITY から SRV_C_MAXPRIORITY までで、この範囲の中間点は SRV_C_DEFAULTPRI です。

Open Server は、必ず、最高優先順位の実行可能なスレッドから再開します。実行可能なスレッドが複数存在し同じ優先順位である場合には、Open Server は最初に実行可能になったスレッドを再開します。1 つのスレッドの優先順位を他のスレッドの優先順位よりも高くした場合には、そのスレッドが実行可能でなくなるか、優先順位が下げられるまで、Open Server はそのスレッドを実行し続け、他のスレッドを実行しません。

スレッドの優先順位を上げることは、クリティカル・セクション中に他のスレッドによる介入を防ぐための効果的な手段ですが、その一方で同時実行性に悪影響を及ぼすこともあります。優先順位を上げるということは、1 つのスレッドにサーバを独占させることになります。スレッドの優先順位を SRV_C_DEFAULTPRI 以上に上げると、Open Server ランタイム・システムを構成しているスレッドさえも実行できなくなります。このような影響を最小限に抑えるためには、どうしても必要になるまで優先順位の引き上げを遅らせ、できるだけ早くまた下げてください。クリティカル・セクションに不要なコードを入れないでください。

コールバック・ルーチン

srv_callback ルーチンを使って、スレッドのコールバック・ハンドラをインストールできます。Open Server は、スレッドのステータスがユーザの指定したものに変わったときに、ユーザのルーチンを呼び出します。たとえば、あるスレッドが中断したときに実行する SRV_C_SUSPEND コールバック・ハンドラをインストールできます。

注意 コールバック・ハンドラをインストールし実行できるかは、プラットフォームに依存します。現在使用しているプラットフォーム上で特定のステータス移行についてコールバック・ハンドラがインストール可能であるかどうかは、srv_capability を使って調べてください。

表 2-23 は、srv_callback によるコールバック・ハンドラのインストールが可能なステータス移行をまとめたものです。

表 *2-23:* ステータスの移行

ステータスの移行	意味
SRV_C_EXIT	スレッドは、その実行のために生成されたルーチンの実行を終了したか、接続を切断したクライアントに関連付けられている。ハンドラは、終了するスレッドのコンテキスト内で実行する。
SRV_C_PROCEXEC	Open Server は、レジスタード・プロシージャを実行しようとするときにこのコールバック・ハンドラを呼び出す。ハンドラは、レジスタード・プロシージャを要求したスレッドのコンテキストで実行する。結果として、SRV_C_PROCEXEC コールバック・ハンドラは、クライアントがなんらかのレジスタード・プロシージャ・オペレーションを試行するときに実行する。クライアントがレジスタード・プロシージャを作成、削除、または実行する能力を制限するコールバック・ハンドラをインストールできる。
SRV_C_RESUME	スレッドは再開中である。ハンドラはスケジューラ・スレッドのコンテキストで実行され、スケジューラのスタックを使用する。

ステータスの移行	意味
SRV_C_SUSPEND	スレッドは中断している。ハンドラは、中断しているスレッドのコンテキストで実行され、そのスタックを使用する。
SRV_C_TIMESLICE	スレッドは、SRV_TIMESLICE、SRV_VIRTCLKRATE、SRV_VIRTTIMER 設定パラメータで指定した時間 (タイム・スライス) の間、実行した。このハンドラを使うと、長時間実行しているスレッドに対して、他のスレッドを実行可能にするための srv_yield を呼び出すように信号を送ることができる。

プログラミングに関する注意事項

Open Server スレッドは、各自のスタックとレジスタ環境を持つ実行のスレッドですが、Open Server ランタイム・システムのオペレーティング・システム・プロセスのリソースも共有しています。

次に、マルチスレッド・プログラミングに関する重要事項を示します。

- グローバル・データ、ファイル・ハンドル、デバイスなどの共有リソースを保護しなければなりません。

  共有グローバル・データ項目を更新しているときには、他のスレッドがそのデータをアクセスできないようにする手段をとっていないのであれば、そのスレッドを中断することができるルーチンを呼び出さないでください。さもなければ、他のスレッドは矛盾するデータで作業することにもなります。

  特定のリソースに対して自身だけがアクセス権を持っているかのように動作するプログラム・ロジックには注意してください。たとえば、グローバル変数の値を使って計算の一部を行い、中断するようなルーチンです。その間に他のスレッドがグローバル変数を変更することが可能であり、矛盾の問題を引き起こします。計算が、完了する前に不正確なものになることがあります。

- 複数のスレッドが実行できるルーチンでは、静的変数は避けてください。

  静的変数を変更するルーチンを、複数のスレッドによって呼び出すことが可能な場合は、そのルーチンの複数のインスタンスが競合しないことを保証しなければなりません。そのルーチンが、静的変数のポインタを返すように作られている場合は、スレッドの中断中に変数の内容が変更されることもあるので、データの矛盾がさらに起こりやすくなります。スレッドはそれぞれ専用のスタックを持っているので、自動変数を使用するほうが安全です。アプリケーションはメモリを提供し、そこに結果をコピーしなければなりません。静的変数を使用しなければならない場合は、上記の手段で保護するようにしてください。

- SRV_ATTENTION イベントは、割り込みレベルで実行できます。SRV_ATTENTION ハンドラが操作するアプリケーション構造が、他のイベント・ハンドラやサービス・スレッドなどの非割り込みレベル・コードでも変更またはテストされている場合は、その変更やテストの結果は信頼できません。割り込みレベルの SRV_ATTENTION ハンドラと非割り込みレベル・コードの調整を行うためには、アテンション・レベルの起動とスリープを使用してください。

例

サンプル・プログラム *multthrd.c* には、さまざまなマルチスレッド・プログラミングの例が記述されています。

ネゴシエートされた動作

Open Server アプリケーションは、クライアントとネゴシエートして、さまざまな分野におけるアプリケーションの動作を決定します。ネゴシエーションにはクライアントがログインするときに行われるものもあれば、Open Server ランタイム・システムの存続期間中にアドホックに発生するものもあります。

ログイン・ネゴシエーション

ログイン時にネゴシエートされる要素はいくつかあります。いくつかは、Open Server が透過的にネゴシエートするものであり、Open Server アプリケーションがアクションを起こす必要はありません。他のものは、アプリケーション呼び出しで明示的に処理されます。ログイン・ネゴシエーションは、常に SRV_CONNECT イベント・ハンドラ内で起こります。

透過的ネゴシエーション

次の要素は、アプリケーションに対して透過的に決定されます。

- 文字データが表示される文字セット。クライアントがログインするときには、さまざまな情報を提供しますが、その中にはロケールに適当な文字セットの名前があります。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、Open Server はデータをクライアントの文字セットに変換します。
- 表示する Open Server エラー・メッセージの言語。
- バイト順序：プラットフォームに依存。

- TDS プロトコル・レベル。
- 浮動小数点表現：プラットフォームに依存。

サーバのデフォルトの各国言語と文字セットは、サーバの初期化中に確立されます。

クライアントは、文字セットと各国言語を後で再ネゴシエーションすることができます。「アドホック・ネゴシエーション」(114 ページ) を参照してください。

明示的ネゴシエーション

次の要素は、アプリケーション自体がクライアントとネゴシエートして決定します。

- アプリケーションがデフォルトを使用しない場合には、クライアントのできる要求の種類と、Open Server アプリケーションが返せる応答の種類
- クライアントとサーバが通信するセキュリティ・レベル

クライアントは、ログイン・レコードを送った後に、機能情報を送ります。クライアントと Open Server アプリケーションは、その特定の接続において送信が可能である要求と応答のセットに同意しなければなりません。これらの機能を確立しなければ、さらに要求や応答を送ることはできません。機能の詳細については、「機能」(22 ページ) を参照してください。

セキュア接続のネゴシエーション

Open Server アプリケーションは、クライアントとの間にセキュア接続を確立することができます。セキュア接続とは、クライアントの ID を詳細に認証し、パスワードを確認した後で確立される接続のことです。

注意　アプリケーションは、アプリケーション自体のセキュリティ・コードを組み込まずに、セキュリティ・サービス・プロバイダが提供する外部セキュリティ・システムを使用できます。サード・パーティのセキュリティ・サービス・プロバイダを使用するように Open Server アプリケーションを設定する方法については、「セキュリティ・サービス」(158 ページ) を参照してください。

アプリケーションは、次の方法のいずれかまたはすべてを使ってこのセキュリティ・チェックを行うことができます。

- クライアントにチャレンジを送り、これに対応する応答をするようにクライアントに要求します。
- クライアントに「暗号化キー」を送ります。クライアントは応答として暗号化パスワードを返し、アプリケーションがこれを複号化して検証します。
- クライアントに「セキュリティ・ラベル」を要求します。クライアントがこれを送り、接続のセキュリティのレベルが確立されます。

- アプリケーション定義のログイン・ハンドシェイクを開始します。
- 透過的なセキュリティ・ハンドシェイクを開始します。このためには、*libtcl.cfg* ファイル内に security エントリが指定されていることと、要求されたセキュリティ・サービスのドライバがインストールされている必要があります。「interfaces ファイルへの変更」(171 ページ) および「セキュリティ・サービス」(158 ページ) を参照してください。
- セキュリティ・セッション・コールバックを使って、リモート・サーバとゲートウェイ・クライアントとの間でセキュリティ・セッションのネゴシエーション・データを交換します。「ダイレクト・セキュリティ・セッションでのフル・パススルー・ゲートウェイ」(178 ページ) および『Open Client Client-Library/C リファレンス・マニュアル』を参照してください。

アプリケーションがセキュア・ログインをネゴシエートするには、srv_negotiate ルーチンを SRV_CONNECT イベント・ハンドラの中で実行します。

アドホック・ネゴシエーション

アプリケーションは、サーバが稼働中であればいつでも、いくつかの要素をクライアントとネゴシエートまたは再ネゴシエートできます。アドホック・ネゴシエーションは、SRV_LANGUAGE イベント・ハンドラか SRV_OPTION ハンドラ内で起こります。クライアントは、次のことができます。

- Transact-SQL 言語コマンドまたはオプション・コマンドを使用して、文字セットと各国言語の再ネゴシエーションを行います。
- Transact-SQL 言語コマンドまたはオプション・コマンドを使用して、クエリ処理動作の特性を決定します。クライアントは、特定のオプションの現在のステータスを要求できるだけではなく、オプションの設定やクリアも要求することもできます。

SRV_OPTION イベントの説明とオプションのリストについては、「オプション」(115 ページ) を参照してください。

各国言語と文字セットのネゴシエーションの詳細については、「国際化のサポート」(92 ページ) で説明しています。

『ASE リファレンス・マニュアル』および「セキュリティ・サービス」(158 ページ) を参照してください。

例

サンプル・プログラム *ctos.c* には、ネゴシエーション・ログインのコードが記述されています。

オプション

Adaptive Server Enterprise では、クエリ処理をどのように扱うかをクライアントが決めることができます。クエリ処理の動作に関するさまざまな設定可能オプションがあります。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。

Open Server アプリケーションは、クエリ処理オプションに関するクライアント要求に応答することができます。

クライアント・アプリケーションは、次の 2 つの方法のどちらかで、Adaptive Server Enterprise のクエリ処理オプションの現在の値を設定、クリア、要求できます。

- Transact-SQL 言語コマンドを使用する
- オプション・コマンドを発行する

アプリケーションは、クライアントが要求するオプションのために必要になる言語コマンドを発行すると予期した場合には、SRV_LANGUAGE イベント・ハンドラでそのような要求を解析するためのコードを含んでいなければなりません。

クライアント・オプション・コマンドは、SRV_OPTION イベントをトリガします。アプリケーションは、このような要求に対して srv_options コマンドを使って、SRV_OPTION イベント・ハンドラから応答します。

SRV_OPTION イベント・ハンドラの内部構造

クライアントは、オプションの設定やクリア、またはその現在の値を返すように要求できます。これらのコマンドはいずれも、SRV_OPTION イベントをトリガします。SRV_OPTION イベント・ハンドラを使用して、アプリケーションは次のことを行います。

1　cmd 引数を CS_GET に設定して srv_options を呼び出します。クライアントが発行したコマンドの種類 (SRV_SETOPTION、SRV_CLEAROPTION、または SRV_GETOPTION) が、optcmdp に返されます。オプション自体は、optionp に返されます。*bufp は、そのオプションに関連するすべての有効値を含みます。

   たとえば、クエリの影響を受けるローの数を Adaptive Server Enterprise がレポートしないようにクライアントが要求した場合は、optcmdp に SRV_SETOPTION が格納され、*optionp に CS_OPT_NOCOUNT が格納され、*bufp に CS_TRUE が格納されます。

2　optcmdp が SRV_SETOPTION または SRV_CLEAROPTION の場合、スタンドアロンの Open Server アプリケーションでは、アプリケーション自身がオプションを設定またはクリアします。アプリケーションがゲートウェイである場合は、リモート・サーバのオプションを操作するための適切なクライアント呼び出しを送信しなければなりません。

3 optcmdp が SRV_GETOPTION の場合は、cmd を CS_SET に設定し、optcmd を SRV_SENDOPTION に設定し、optionp をクライアントが値を求めているオプションに設定し、bufp を現在の値に設定して、srv_options を呼び出します。

オプションの説明とデフォルト値

表 2-24 に、クライアントが設定、取得、クリアできるオプションと、各オプションのデフォルト値を示します。

表 *2-24:* サーバ・オプションの記号定数

記号定数	オプションの役割	デフォルト値
CS_OPT_ANSINULL	このオプションが CS_TRUE に設定されている場合、Adaptive Server Enterprise は “=NULL” と “is NULL” が同一ではないという ANSI 動作を強制する。標準 Transact SQL では、“=NULL” と “is NULL” は同一のものとして扱われる。 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	このオプションが CS_TRUE に設定されている場合、Adaptive Server Enterprise は update 文および delete 文に対するパーミッションの確認を ANSI に準拠させる。	CS_FALSE
CS_OPT_ARITHABORT	このオプションが CS_TRUE に設定されている場合、Adaptive Server Enterprise は実行中に演算例外が起こったときにクエリをアボートする。	CS_FALSE
CS_OPT_ARITHIGNORE	このオプションが設定されている場合、クエリ実行中に演算例外が起こったときには、Adaptive Server Enterprise は選択値または更新値を NULL に置き換える。Adaptive Server Enterprise は警告メッセージを返しません。CS_OPT_ARITHABORT または CS_OPT_ARITHIGNORE のどちらも設定されていない場合は、Adaptive Server Enterprise は NULL 値による置き換えを行い、クエリの実行完了後に警告メッセージを出力します。	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有効になる。	適用しない
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。	適用しない

記号定数	オプションの役割	デフォルト値
CS_OPT_CHAINXACTS	このオプションが CS_TRUE に設定されている場合、Adaptive Server Enterprise は連鎖トランザクション動作を使用する。 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。 非連鎖トランザクション動作は、トランザクションを定義する明示的な commit transaction 文を必要とする。	CS_FALSE (非連鎖トランザクション動作)
CS_OPT_CURCLOSEONXACT	このオプションが CS_TRUE に設定されている場合、トランザクション領域内でオープンされたカーソルはすべて、トランザクションが完了するとクローズされる。	CS_FALSE
CS_OPT_CURREAD	現在の読み込みレベルを指定する、セキュリティ・ラベルを設定する。	NULL
CS_OPT_CURWRITE	現在の書き込みレベルを指定する、セキュリティ・ラベルを設定する。	NULL
CS_OPT_DATEFIRST	このオプションは、週の「最初」の日と見なす曜日を設定する。	us_english の場合、デフォルトは CS_OPT_SUNDAY
CS_OPT_DATEFORMAT	このオプションは、datetime または smalldatetime データ入力のための日付の要素 (月／日／年) の順番を設定する。	us_english の場合、デフォルトは CS_OPT_FMTMDY
CS_OPT_FIPSFLAG	このオプションが CS_TRUE に設定されている場合、Adaptive Server Enterprise は送られる非標準 SQL コマンドすべてにフラグを立てる。	CS_FALSE
CS_OPT_FORCEPLAN	このオプションが CS_TRUE に設定されている場合、Adaptive Server Enterprise はクエリの “from” 句にリストされている順序でテーブルをジョインする。	CS_FALSE
CS_OPT_FORMATONLY	このオプションが CS_TRUE に設定されている場合、Adaptive Server Enterprise は select クエリに対して、データ自体ではなくデータの説明を返す。	CS_FALSE
CS_OPT_GETDATA	このオプションが CS_TRUE に設定されている場合は、Adaptive Server Enterprise から insert、delete、update の各コマンドに関する情報が返される。この情報は、メッセージ結果セットとパラメータの形で返される。アプリケーションは、これを利用すると、挿入または削除されるローが格納されるテンポラリ・テーブルの名前を構築できる。更新は、挿入と削除から成ることに注意。	CS_FALSE
CS_OPT_IDENTITYOFF	テーブルの identity カラムへの挿入を実行できないようにする。Adaptive Server Enterprise のマニュアルで set コマンドを参照してください。	適用しない
CS_OPT_IDENTITYON	テーブルの identity カラムへの挿入を有効にする。Adaptive Server Enterprise のマニュアルで set コマンドを参照してください。	適用しない

記号定数	オプションの役割	デフォルト値
CS_OPT_ISOLATION	このオプションは、トランザクションの分離レベルを指定するために使う。有効値は、CS_OPT_LEVEL1 と CS_OPT_LEVEL3。CS_OPT_ISOLATION を CS_OPT_LEVEL3 に設定すると、トランザクション内の select クエリで指定されたテーブルの全ページを、トランザクションの全期間にわたってロックさせる。	CS_OPT_LEVEL1
CS_OPT_NOCOUNT	このオプションを指定すると、各 SQL 文の影響を受けるローの数に関する情報が Adaptive Server Enterprise から返されなくなる。	CS_FALSE
CS_OPT_NOEXEC	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はするが、実行はしない。このオプションは、CS_OPT_SHOWPLAN とともに使われる。	CS_FALSE
CS_OPT_PARSEONLY	このオプションが設定されている場合、サーバ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、クエリを実行することはしない。	CS_FALSE
CS_OPT_QUOTED_IDENT	このオプションが CS_TRUE に設定されている場合、Adaptive Server Enterprise は二重引用符内のすべての文字列を識別子として扱う。	CS_FALSE
CS_OPT_RESTREES	このオプションが設定されていると、Adaptive Server Enterprise はクエリの構文はチェックしても実行はせず、必要に応じてクライアントに対して解析解決ツリー (通常のロー結果セットの image カラムの形式) やエラー・メッセージを返す。	CS_FALSE
CS_OPT_ROWCOUNT	このオプションが設定されている場合、select 文実行時に Adaptive Server Enterprise から返される通常ローの数は、指定された最大数までとなる。このオプションは、返される計算ローの数を制限しない。 CS_OPT_ROWCOUNT の動作は、他のオプションとは多少異なる。このオプションは常にオンに設定され、オフになることはない。CS_OPT_ROWCOUNT を 0 に設定するとデフォルトの動作となり、select 文によって生成されたすべてのローを返す。したがって、CS_OPT_ROWCOUNT をオフにするには、カウント 0 で設定をオンにする。	0 (すべてのローが返される)
CS_OPT_SHOWPLAN	このオプションが CS_TRUE に設定されている場合には、Adaptive Server Enterprise はコンパイル終了後に処理プランの記述を作成し、クエリの実行を継続する。	CS_FALSE
CS_OPT_STATS_IO	このオプションは、Adaptive Server Enterprise の内部 I/O 統計を各クエリ後にクライアントに返すかどうかを決定する。	CS_FALSE
CS_OPT_STATS_TIME	このオプションは、Adaptive Server Enterprise の解析、コンパイル、および実行時間統計を各クエリ後にクライアントに返すかどうかを決定する。	CS_FALSE

記号定数	オプションの役割	デフォルト値
CS_OPT_STR_RTRUNC	このオプションが CS_TRUE に設定されている場合、Adaptive Server Enterprise は、文字データの右側のトランケーションについて ANSI 標準に準拠する。	CS_FALSE
CS_OPT_TEXTSIZE	このオプションは、Adaptive Server Enterprise のグローバル変数 @@*textsize* の値を変更する。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。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示すパラメータを使用すること。	32,768 バイト
CS_OPT_TRUNCIGNORE	このオプションが CS_TRUE に設定されている場合には、Adaptive Server Enterprise はトランケート・エラーを無視する。これが標準 ANSI 動作。 このオプションが CS_FALSE に設定されている場合、変換の結果トランケートが生じたときには Adaptive Server Enterprise はエラーを出す。	CS_FALSE

srv_options (304 ページ) に、各オプションの有効値とデータ型を示します。

例

サンプル・プログラム *ctos.c* には、クライアント・オプション・コマンドを処理するコードが記述されています。

部分更新

Open Client および Open Server 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。『Open Client Client-Library/C リファレンス・マニュアル』を参照してください。

注意　現在、Adaptive Server Enterprise は、text データと image カラムの部分更新をサポートしていません。

Open Server の設定

この項では、部分更新をサポートするために Open Server を設定する方法について説明します。

sp_mda

sp_mda は、サーバからメタデータを取得するストアド・プロシージャです。部分更新をサポートするには、Open Server アプリケーションが sp_mda ストアド・プロシージャを定義し、Open Client アプリケーションが使用する必要がある updatetext 構文を指定する必要があります。

Open Client アプリケーションは、次のパラメータと値を使用して sp_mda を呼び出す必要があります。

パラメータ	値	説明
clienttype	5	5 は、クライアントが Client-Library であることを示しています。
mdaversion	1	
clientversion	0	clientversion は、クライアントのバージョンを示すオプションのパラメータです。デフォルトは 0 です。

サーバが部分更新をサポートする場合、sp_mda は次の値を返します。

パラメータ	値
mdinfo	“UPDATETEXT”
querytype	2
query	*updatetext_syntax* 例： `updatetext ?? ? {NULL	?} {NULL	?}` ここで、“?” は updatetext パラメータを示しています。

『Mainframe Connect DB2 UDB Options for IBM CICS and IMS Installation and Administration Guide』(英語) を参照してください。sp_mda のサンプル実装については、*$SYBASE/$SYBASE_OCS/sample/srvlibrary/updtext.c* を参照してください。

SRV_T_BULKTYPE

クライアントにより送信された部分更新済みデータを正しく取得するには、Open Server アプリケーションは SRV_T_BULKTYPE を SRV_TEXTLOAD、SRV_UNITEXTLOAD、SRV_IMAGELOAD に設定する必要があります。「SRV_T_BULKTYPE」(145 ページ) を参照してください。

ハンドラ

SRV_LANGUAGE および SRV_BULK ハンドラは、Open Server にインストールする必要があります。Open Server は、SRV_LANGUAGE を使用して、Client-Library から updatetext 文を取得します。一方、SRV_BULK は、ct_send_data() を通して送信されたデータを取得します。

『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。

パススルー・モード

Open Client アプリケーションと Adaptive Server Enterprise 間でゲートウェイとしての役割を果たす Open Server アプリケーションは、内容を確かめずにクライアントとサーバの間で TDS パケットのやり取りを行うことができます。TDS パケットをこのように処理する Open Server は、パススルー・モードで動作していると言います。

Open Server ゲートウェイ・アプリケーションがクライアントから受け取った TDS 情報をアンパックしたり、Adaptive Server Enterprise に送信する前に再パックしたりする必要がないので、パススルー・モードは非常に効率的です。

Open Client Server 12.5.1 以前のパススルー・モードでは、クライアントから要求されるパケット・サイズを Open Server によりサポートされる最大サイズに制限することにより、ネゴシエート済みパケット・サイズの妥当性を保証しています。

サーバ指定のパケットサイズをサポートするリモート・サーバが Open Server 内での設定より長いパケットサイズを指定する場合、長い方のパケットサイズが SRV_S_NETBUFSIZE の設定にかかわらず使用されます。

パススルー・モードには、次の 2 種類があります。

- 通常パススルー・モード
- イベント・ハンドラ・パススルー・モード

どちらの種類のパススルー・モードも、パススルー・ルーチン srv_recvpassthru、ct_sendpassthru、ct_recvpassthru、srv_sendpassthru を使用します。相違点は次のとおりです。

- 通常パススルー・モードでは、Open Server アプリケーションはイベントを認識し、イベント・ハンドラをトリガします。これらのイベント・ハンドラは、パススルー・ルーチンを呼び出すようにコーディングされます。

  「通常パススルー・モード」(122 ページ) を参照してください。

- イベント・ハンドラ・パススルー・モードでは、Open Server アプリケーションは接続されているイベントのほとんどの種類を認識しません。代わりに、接続のネットワーク読み込みが完了するたびに、フル・パススルー・イベント・ハンドラがトリガされます。フル・パススルー・イベント・ハンドラは、パススルー・ルーチンを呼び出すようにコーディングされます。

  「イベント・ハンドラ・パススルー・モード」(124 ページ) を参照してください。

DB-Library にも、パススルー・モードをサポートするルーチンがあります。詳細については、『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。

通常パススルー・モード

Sybase では、最初はこの種類のパススルー・モードだけをサポートしていました。

通常パススルー・モードでは、Open Server はイベント (SRV_LANGUAGE、SRV_RPC など) を認識し、適切なイベント・ハンドラをトリガします。各イベント・ハンドラは、パススルー・ルーチンを呼び出すようにコーディングする必要があります。

パススルー・モードでの TDS プロトコル・レベルのネゴシエーション

Sybase クライアントとサーバが接続するときには、互いに使用する TDS プロトコルをまず合わせますが、これは通常、両方のプログラムが認識できる最新バージョンのプロトコルです。「ネゴシエートされた動作」(112 ページ) を参照してください。

Open Server ゲートウェイ・アプリケーションがパススルー・モードで動作している場合、TDS パケットの作成と解釈は、ゲートウェイではなく、リモート Sybase クライアントと Adaptive Server Enterprise によって行われます。したがって、TDS ネゴシエーションは 2 つのリモート・プログラム間で行われることになります。ゲートウェイは、両者間の応答を伝達することによって、このネゴシエーションを簡単にしなければなりません。TDS ネゴシエーション・プロセスは SRV_CONNECT イベント・ハンドラ内部で起こらなければならず、次の手順が必要です。

1 次のいずれかのプロパティを設定します。

- スレッドが通常パススルー・モードを使用することを示す SRV_T_PASSTHRU
- スレッドがイベント・ハンドラ・パススルー・モードを使用することを示す SRV_T_FULLPASSTHRU

srv_getloginfo および ct_setloginfo がクライアント／サーバ機能をパススルー・モードに正しくネゴシエートするために、これらのプロパティのどちらかを設定してください。

2　srv_getloginfo － CS_LOGINFO 構造体を割り付け、クライアント・スレッドからのログイン情報を格納します。

3　ct_setloginfo － 手順 2 で取得したログイン情報を使用して CS_LOGINFO 構造体を準備します。

4　クライアント・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いる場合は、次の手順に従ってクライアントのセキュリティ・プリンシパル名を転送します。セキュリティ・プリンシパル名は CS_LOGINFO 構造体の一部ではないので、これらの手順が必要です。

- srv_thread_props(..CS_GET, SRV_T_USER) を呼び出して、クライアントのセキュリティ・プリンシパル名を取得します。
- ct_con_props(..CS_SET, CS_USERNAME) を呼び出して、ターゲット・サーバへの接続のプリンシパル名を設定します。

5　ct_connect を呼び出して、リモート・サーバにログインします。

6　ct_getloginfo － CS_CONNECTION 構造体から、新しく割り付けられた CS_LOGINFO 構造体に、ログイン応答情報を転送します。

7　srv_setloginfo － 手順 6 で取得したリモート・サーバの応答をクライアントに送信し、CS_LOGINFO 構造体を解放します。

通常パススルー・モードの使用

通常の TDS パススルーは、SRV_ATTENTION、SRV_CONNECT、SRV_DISCONNECT、SRV_START、SRV_STOP 以外のどのイベント・ハンドラでも起こります。

クライアント要求は、1 つまたは複数の TDS パケットのストリームの形で届きます。info 引数が SRV_I_PASSTHRU_MORE に設定されている間は、ハンドラは繰り返し srv_recvpassthru を呼び出します。各パケットを受け取るたびに、ハンドラはそのパケットをリモート Adaptive Server Enterprise または Open Server に渡すために ct_sendpassthru を呼び出します。リモート・サーバは、接続クライアントから直接受け取るのとまったく同じ TDS ストリームを受け取ります。

警告！ TDS の最新バージョンでは、1 つのバッチで複数のコマンドを送る機能が導入されています。最初のコマンドだけがイベント・ハンドラをトリガします。残りのコマンドについては、Open Server はイベント・ハンドラを呼び出しません。

Client-Library ルーチンである ct_recvpassthru は、接続時に到着する TDS パケットを受け取ります。srv_sendpassthru Server-Library ルーチンは、パケットをクライアントにそのまま送ります。CS_PASSTHRU_MORE が返される間は、ct_recvpassthru ルーチンは別の TDS パケットを受け取ります。

例

サンプル・プログラム *fullpass.c* には、パススルー・モード・ゲートウェイの例が記述されています。

イベント・ハンドラ・パススルー・モード

この種類のパススルー・モードでは、Open Server はほとんどのタイプのイベントを認識しません。その代わり Open Server は、接続からのネットワーク読み込みが完了するたびに、フル・パススルー・イベント・ハンドラを起動します。

イベント・ハンドラ・パススルー・モードでは、パケットごとのセキュリティ・サービス (暗号化など) を使用するクライアント／サーバ接続がパススルー・モードを使用できるように設計されています。

通常パススルー・モードでは、Open Server がパケットを解釈して特定のイベントを識別する必要があります。パケットが暗号化されている場合は、これは不可能です。

スレッドに対してイベント・ハンドラ・パススルー・モードを使用するには、次の手順に従います。

- フル・パススルー・イベント・ハンドラをコーディングしてインストールします。「フル・パススルー・イベント・ハンドラのコーディングとインストール」(124 ページ) を参照してください。
- 特定のスレッドに対してイベント・ハンドラ・パススルー・モードを有効にするために、Open Server 接続ハンドラ内で SRV_T_FULLPASSTHRU を CS_TRUE に設定します。「スレッドに対するイベント・ハンドラ・パススルー・モードの有効化」(125 ページ) を参照してください。
- クライアントとターゲット・サーバとの間で TDS プロトコル・レベルをネゴシエートするルーチンを呼び出します。「TDS プロトコル・レベルのネゴシエーション」(125 ページ) を参照してください。

フル・パススルー・イベント・ハンドラのコーディングとインストール

フル・パススルー・イベント・ハンドラのプロトタイプは次のとおりです。

```
CS_RETCODE CS_PUBLIC func (SRV_PROC *sproc);
```

フル・パススルー・イベント・ハンドラは、次のルーチンを呼び出してパケットの送受信を行います。

- srv_recvpassthru

- ct_sendpassthru
- ct_recvpassthru
- srv_sendpassthru

srv_recvpassthru/ct_sendpassthru ループの実行中はアテンション・イベントを転送できません。イベント・ハンドラのコードとアテンション・ハンドラのコードにロジックを追加して、コマンド全体がリモート・サーバに転送されるまでアテンション・イベントが転送されないようにしてください。

フル・パススルー・イベント・ハンドラは CS_SUCCEED を返して、正常終了をレポートする必要があります。戻り値が CS_SUCCEED 以外の場合は、現在の Open Server スレッドが強制終了されます。

フル・パススルー・イベント・ハンドラをインストールするには、srv_handle の *event* パラメータを SRV_FULLPASSTHRU に、*handler* パラメータをハンドラ・ルーチンのアドレスに設定して srv_handle を呼び出してください。

スレッドに対するイベント・ハンドラ・パススルー・モードの有効化

特定のスレッドに対してイベント・ハンドラ・パススルー・モードを有効にするには、Open Server 接続ハンドラ内で SRV_T_FULLPASSTHRU スレッド・プロパティを CS_TRUE に設定します。

イベント・ハンドラ・パススルー・モードが有効になると、接続からのネットワーク読み込みが完了するたびに、Open Server はフル・パススルー・ハンドラを起動します。

このスレッドでは SRV_LANGUAGE、SRV_RPC、SRV_BULK、SRV_CURSOR、SRV_MSG、SRV_OPTION、または SRV_DYNAMIC タイプのイベントは発生しません。

ただし、SRV_ATTENTION イベントは発生します。Open Server アプリケーションは SRV_ATTENTION ハンドラをインストールして、キャンセル要求を正しく処理する必要があります。

TDS プロトコル・レベルのネゴシエーション

イベント・ハンドラ・パススルー・モードを使用するゲートウェイ・アプリケーションでは、通常パススルー・モードを使用するアプリケーションとまったく同じ方法で、クライアント・アプリケーションとターゲット・サーバとの間の TDS プロトコル・レベルのネゴシエーションが行われます。

アプリケーションの接続ハンドラ内で、SRV_FULLPASSTHRU を CS_TRUE に設定してから、srv_getloginfo、ct_setloginfo、ct_getloginfo、srv_setloginfo の各ルーチンを呼び出してください。

「パススルー・モードでの TDS プロトコル・レベルのネゴシエーション」(122 ページ) を参照してください。

パラメータとロー・データの処理

用語についての注意

「パラメータ・データ」という用語は、クライアントから取得した、またはクライアントに返されたパラメータを指します。その中には入力パラメータもあれば、出力パラメータ、つまり「リターン・パラメータ」もあります。リターン・パラメータは 2 段階に分けて処理されます。つまり、Open Server アプリケーションによってプログラム変数に読み込まれた時点で部分的に処理され、クライアントに返されるときに処理が完了します。

Open Server のデータ処理モデル

Open Server では、3 つのルーチンが共同作業としてクライアントからパラメータ・データやフォーマットを取得し、ローのデータを送り、またクライアントにパラメータとそのフォーマットを返します。これらのルーチンとは、srv_descfmt、srv_bind、srv_xferdata です。

アプリケーションは、パラメータを提供したり結果を要求するようなクライアント・コマンドはすべて、これらのルーチンを使って処理します。RPC コマンド、言語コマンド、カーソル・コマンド、動的 SQL コマンド、メッセージ・コマンド、ネゴシエートされたログイン・コマンドはすべて、これに当てはまります。

これらの 3 つのルーチンはそれぞれ type 引数を持ち、ここに記述、バインド、または転送されるデータの型を示します。たとえば、カーソル・コマンドの入力パラメータのフォーマットを記述するときには、type は SRV_CURDATA に設定されますが、結果ローを処理するときには、type は SRV_ROWDATA に設定されます。type の有効値のリストについては、第 3 章の各ルーチンのページを参照してください。

また、このルーチンは 3 つとも、cmd 引数を持ちます。これは、データ・フローの方向を示します。CS_GET の値は Open Server アプリケーションにクライアントから情報を取得することを指示しますが、CS_SET の値は、アプリケーションがクライアントに結果を返すことを指示します。

アプリケーションは、これらのルーチンを使用して、次の処理を実行できます。

- SRV_RPC、SRV_CURSOR、SRV_DYNAMIC、SRV_MSG、または SRV_CONNECT イベント・ハンドラ内で、入力およびリターン・パラメータ情報を取得します。
- SRV_RPC、SRV_CURSOR、SRV_DYNAMIC、SRV_LANGUAGE、または SRV_MSG イベント・ハンドラ内で結果ロー情報を返信します。
- SRV_LANGUAGE または SRV_RPC ハンドラ内でリターン・パラメータ情報を返信します。

パラメータの取得

パラメータを処理するには、アプリケーションは次の手順に従います。

1　コマンドに含まれているパラメータ数 (パラメータがある場合) を判断するために、srv_numparams を呼び出します。

2　srv_descfmt を呼び出して、各パラメータの記述を取得します。この記述の中に、パラメータがリターン・パラメータであるかどうかの情報も含まれています。リターン・パラメータがある場合には、取得処理はその時点で終わります。パラメータが入力パラメータである場合には、アプリケーションは手順 3 と 4 を続けて行います。

3　ネットワーク経由でクライアントから入ってくるパラメータ・データを保存するプログラム変数を提供するために、srv_bind を呼び出します。

4　手順 3 で指定されたアプリケーション・プログラム変数にクライアント・データを転送するために、srv_xferdata を呼び出します。

リターン・パラメータは、クライアントから取得したときには有効なデータを含んでいません。アプリケーションがリターン・パラメータをクライアントに返すときに、有効データを入力します。Open Server は、リターン・パラメータ・フォーマットを、プログラム変数フォーマットからクライアント・フォーマットに透過的に変換します。

SRV_LANGUAGE ハンドラ内では、最初に実際のパラメータを取得しなくても、アプリケーションは無区別の言語ストリームからリターン・パラメータを「構成」できます。詳細については、「言語データ・ストリームでのパラメータの返送」(130 ページ) を参照してください。

srv_xferdata は全パラメータ・ストリームのために一度だけ呼び出されるのに対し、srv_descfmt と srv_bind は各パラメータごとに呼び出されます。アプリケーションは、すべてのパラメータが記述されバインドされるまでは srv_xferdata を呼び出してはいけません。

アプリケーションがクライアントから情報を取得するので、アプリケーションはこの 3 つのルーチンを、それぞれの cmd 引数を CS_GET に設定して呼び出さなければなりません。

ローの返送

ロー・データの処理には、次の 3 つの基本手順が必要です。

1　srv_descfmt を呼び出し、ロー内の各カラムを記述します。

2　srv_bind を呼び出し、アプリケーションがロー・データを保存した場所を示し、そのフォーマットを識別します。

3　srv_xferdata を呼び出して、手順 2 で指定したアプリケーション・プログラム変数からクライアントにデータを転送します。

srv_descfmt ルーチンはローのカラムごとに一度ずつ呼び出さなければなりませんが、srv_xferdata ルーチンと srv_bind ルーチンは結果ローの数だけ呼び出します。すべてのカラムが記述されバインドされるまでは、アプリケーションは srv_xferdata を呼び出せません。

アプリケーションがクライアントに情報を返すので、アプリケーションはこの 3 つのルーチンを、それぞれの cmd 引数を CS_SET に設定して呼び出します。

リターン・パラメータの返送

リターン・パラメータの処理には、次の 2 つの基本手順が必要です。

1 srv_bind を呼び出し、アプリケーションがリターン・パラメータ・データを保存した場所を示し、そのフォーマットを識別します。

2 srv_xferdata を呼び出して、手順 2 で指定したアプリケーション・プログラム変数からクライアントにリターン・パラメータ・データを転送します。

アプリケーションがクライアントに情報を返すので、アプリケーションはこの 2 つのルーチンをそれぞれの cmd 引数を CS_SET に設定して呼び出します。

リターン・パラメータがテキスト・ストリームから「構成」されている場合、バインドと転送だけでなく、記述する必要もあります。詳細については、「言語データ・ストリームでのパラメータの返送」(130 ページ) を参照してください。

記述、バインド、転送

この項では、記述、バインド、および転送処理の詳細について説明します。

記述

srv_descfmt ルーチンは、アプリケーションがクライアントの予期しているフォーマットでデータを返すために必要な情報を Open Server アプリケーションに提供します。概念的に、このルーチンは、クライアントがどのようにデータを見たか (CS_GET) または見るのか (CS_SET) に関する情報を伝達します。srv_descfmt ルーチンは、さまざまなパラメータとローのプロパティを取得したり設定したりします。

これらのプロパティには、次のような情報が含まれます。

- パラメータまたはカラムの名前
- パラメータまたはカラムの名前の長さ
- パラメータまたはカラムの番号 (ストリームの先頭のパラメータまたはカラムの番号は 1 から開始される)

- パラメータまたはカラムのデータ型
- パラメータまたはカラムで null を設定できるかどうか
- パラメータがリターン・パラメータであるかどうか

srv_descfmt への clfmtp 引数は、この情報を含んでいる CS_DATAFMT 構造体を指します。詳細については、「CS_DATAFMT 構造体」(48 ページ) を参照してください。

バインド

Open Server アプリケーションは、クライアントから受け取るデータを調べるために、そのデータをローカル・プログラム変数の形で保存しなければなりません。アプリケーションは、srv_bind を呼び出すときに、パラメータまたはカラムのデータをローカル・プログラム変数と関連付け、その変数のフォーマットを記述します。

cmd を CS_GET に設定して srv_bind を呼び出すことによって、クライアントから送られるデータの保管位置を Open Server に指示します。cmd を CS_SET に設定して srv_bind を呼び出すことによって、クライアントに返すデータがどこにあるかを Open Server に指示します。

srv_bind への osfmtp 引数は、ローカル・プログラム変数に関する構成情報を含んでいる CS_DATAFMT 構造体を指します。

転送

srv_xferdata ルーチンは、srv_bind 呼び出しで指定されたローカル・プログラム変数にデータを出し入れします。cmd が CS_GET に設定されているときには、srv_xferdata はクライアントからの入力パラメータ・データを変数に入れます。cmd が CS_SET に設定されているときには、このルーチンはカラムとリターン・パラメータ・データをローカル・プログラム変数から取り出しクライアントに送ります。

注意　現時点では、srv_senddone を実行するとフォーマットとカラム情報がネットワークにフラッシュされますが、今後のバージョンでは、フラッシュされなくなります。アプリケーションは、必ず srv_xferdata を使ってネットワークに情報をフラッシュしてください。

---

srv_bind、srv_descfmt、srv_xferdata の詳細については、それぞれのページを参照してください。

自動変換

アプリケーションがデータを取得するときに、アプリケーションのローカル・プログラム変数のフォーマットと異なるフォーマットでクライアントがデータを送信した場合、Open Server はデータをローカル・フォーマットに変換します。アプリケーションがクライアントにデータを返送するときに同じ状況が起きた場合には、Open Server はデータをクライアント・フォーマットに変換します。

言語データ・ストリームでのパラメータの返送

言語データ・ストリームには、パラメータの概念はありません。ただし、テキスト・ストリームを解析できるように装備されている Open Server アプリケーションは、受信ストリームからリターン・パラメータを「構成」できます。その後、パラメータにデータをロードして、記述／バインド／転送のプロシージャを使用して返送できます。

たとえば、クライアントがリターン・パラメータを含む Transact-SQL ストアド・プロシージャを送るとします。このクエリを予期している Open Server アプリケーションは、“output = @var” という文字列 (*var* はリターン・パラメータのプレースホルダ) を解析し、*var* のフォーマット情報とデータを返すことができます。

アプリケーションが cmd を CS_SET に、type を SRV_RPCDATA に設定して srv_descfmt を呼び出すことができるのは、言語イベント・ハンドラからだけです。

例

サンプル・プログラム *ctos.c* は、一連の記述／バインド／転送の呼び出しを使ってパラメータとカラムのデータを処理します。

プロパティ

プロパティは、Open Server アプリケーションの動作のさまざまな内容を定義します。Open Server プロパティは次の 3 つに分類されます。

- コンテキスト・プロパティ
- サーバ・プロパティ
- スレッド・プロパティ

コンテキスト・プロパティとサーバ・プロパティは、全体として Open Server アプリケーションに付属しています。これらのプロパティはサーバワイドな動作を制御して、すべてのクライアント／サーバ接続に有効です。

スレッド・プロパティは、クライアント・スレッドとサービス・スレッドに付属しています。これらのプロパティのほとんどは取得のみが可能で、設定することはできません。アプリケーションはいくつかのスレッド・プロパティを設定して、接続ごとにいくつかのサーバワイドな属性を上書きできます。

プログラマはプロパティを設定することで、Open Server アプリケーションの機能を調節できます。さらにアプリケーションは、情報が必要なときにいくつかのプロパティを取得することもできます。

cs_config、srv_props、srv_thread_props の各プロパティを使用して、コンテキスト・プロパティ、サーバ・プロパティ、スレッド・プロパティをそれぞれ設定します。

「コンテキスト・プロパティ」(131 ページ)、「サーバ・プロパティ」(132 ページ)、「スレッド・プロパティ」(139 ページ) を参照してください。

『Open Client and Open Server Common Libraries リファレンス・マニュアル』の「cs_config」の項と、このマニュアルの srv_props および srv_thread_props のリファレンス・ページを参照してください。

コンテキスト・プロパティ

コンテキスト・プロパティは、CS-Library の CS_CONTEXT 構造体内に保管されます。アプリケーションでコンテキスト・プロパティを設定または取得するには、CS-Library ルーチン cs_config を使用します。このルーチンの詳細については、『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。

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

  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 特有のコンテキスト・プロパティのリストについては、『Open Client/Server Common Libraries リファレンス・マニュアル』の「cs_config」を参照してください。

- Client-Library に固有なコンテキスト・プロパティ

  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。『Open Client Client-Library/C リファレンス・マニュアル』を参照してください。

- Server-Library 固有のコンテキスト・プロパティ

  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。

Open Server アプリケーションが設定できるコンテキスト・プロパティは、次のとおりです。

- Open Server が CS-Library エラーを検出した場合に呼び出すルーチン
- Open Server の各国言語、文字セット、ソート順などのローカライゼーション情報
- アプリケーションのデータ領域へのポインタのロケーション。このプロパティを使用すると、アプリケーションは制御情報を Open Server のコンテキストと対応付けることができます。Open Server はこのポインタを使用しません。つまりこのポインタは、Open Server アプリケーション・プログラマの利便性を考慮して提供されています。

これらのコンテキスト・プロパティの設定と取得はどちらも、cs_config ルーチンを使用して行うことができます。『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。

サーバ・プロパティ

サーバ・プロパティは CS_CONTEXT 構造体内に保管されます。アプリケーションは Server-Library ルーチン srv_props を使用して、サーバ・プロパティの設定または取得を行います。

サーバ・プロパティによって、メモリ割り付けルーチンや確立できる物理ネットワーク接続の最大数など、Open Server アプリケーションの動作のさまざまな内容が定義されます。

サーバ・プロパティの内容を有効にするには、アプリケーションが初期化の前にこれらのプロパティを設定する必要があります。初期化の後でサーバ・プロパティが設定された場合は、Open Server がエラーを表示します。

アプリケーションの初期化コードには、次の手順が必要です。

1 cs_ctx_alloc を呼び出して、CS_CONTEXT 構造体を割り付けます。

2 srv_version ルーチンを呼び出して、Open Server のバージョン番号を設定します。srv_version ルーチンは CS_CONTEXT 構造体へのポインタを取ります。

3 srv_props ルーチンを呼び出して、プロパティのデフォルトを設定します。

4 srv_init ルーチンを呼び出して、サーバを初期化します。

5 srv_run を呼び出して、サーバの稼働を開始します。

設定と取得の両方が可能なプロパティもありますが、いずれか一方しか実行できないプロパティもあります。詳細については、srv_props (313 ページ) を参照してください。

表 *2-25:* サーバ・プロパティ

プロパティ名	定義	注意
SRV_S_ALLOCFUNC	Open Server がメモリを割り付けるのに使用するルーチンのアドレス。	
SRV_S_APICHK	Server-Library 引数の検証とステータスのチェックを有効(CS_TRUE)にするか無効(CS_FALSE) にするかを示すブール値。	多くの Server-Library ルーチンは内部的に CS-Library ルーチンを呼び出す。このため、引数とステータスを完全にチェックするアプリケーション・プログラマは、cs_config プロパティ CS_NOAPICHK を CS_FALSE に設定する必要がある。
SRV_S_ATTNREASON	Open Server アプリケーションのアテンション・ハンドラが呼び出された理由。	クライアントのアテンションが SRV_ATTENTION イベントをトリガした場合は SRV_ATTENTION を返す。クライアント接続の切断がそのイベントをトリガした場合は、SRV_DISCONNECT を返す。
SRV_S_CERT_AUTH	CS_CHAR 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。	このプロパティの最大長は SRV_MAXCHAR バイト。
SRV_S_CURTHREAD	アクティブなスレッドの内部制御構造体のアドレス。	スレッド・ライブラリとともに SRV_S_PREEMPT を使用した場合、一部の SRV_S_CURTHREAD 機能が無効になる。
SRV_S_DEFQUEUESIZE	遅延イベント・キューのサイズ。	
SRV_S_DISCONNECT	このプロパティを CS_TRUE に設定すると、クライアントの接続が切断されたときにアプリケーションの SRV_ATTENTION イベント・ハンドラが呼び出される。	割り込み時にクライアント接続の切断が検出された場合は、割り込みレベルで SRV_ATTENTION イベント・ハンドラを呼び出すことができる。
SRV_S_DSPROVIDER	ディレクトリ・サービス・プロバイダの名前。デフォルト値はプラットフォームごとに異なる。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。	このプロパティの最大長は SRV_MAXCHAR バイト。
SRV_S_DSREGISTER	Server-Library が起動時に Server-Library 自体をディレクトリ・サービスに登録する必要があることを示すには、CS_TRUE に設定する。登録を行わないようにするには、CS_FALSE に設定する。	
SRV_S_ERRHANDLE	Open Server エラー・ハンドラのアドレス。	
SRV_S_FREEFUNC	Open Server がメモリを解放するのに使用するルーチンのアドレス。	
SRV_S_IFILE	Open Server で使用可能な interfaces ファイルの名前。	このプロパティの最大長は SRV_MAXCHAR バイト。

プロパティ名	定義	注意
SRV_S_LOGFILE	Open Server が書き込みを行うログ・ファイルの名前。	SRV_S_LOGFILE プロパティは、srv_init の呼出し後に設定できる。 srv_init を呼び出した後、*bufp* を空の文字列 ("") に設定し、*buflen* を 0 に設定した状態で SRV_S_LOGFILE プロパティを設定すると、ログ・ファイルが閉じる。 このプロパティの最大長は SRV_MAXCHAR バイト。
SRV_S_LOGSIZE	ログ・ファイルの最大サイズ。ログがこのサイズよりも大きくなった場合、Open Server はログ・ファイルの現在の内容を *currentfilename*_old という名前の別のファイルに移動して、現在のログを 0 バイトにトランケートする。	
SRV_S_MAXLISTENERS	リスナ・スレッドの最大値を制限する。	デフォルト値は CS_MAX_NOMAX。この値は、新しいリスナ・スレッドを作成するときにのみ使用される。このプロパティを SRV_S_NUMLISTENERS の現在の値よりも小さい値に設定すると、リスナは失敗しない。
SRV_S_MSGPOOL	実行時に Open Server アプリケーションに対して使用できるメッセージの数。	Open Server アプリケーションでは、srv_putmsgq でメッセージを使用する。メッセージは、srv_getmsgq によって受信されるまで継続して使用される。SRV_S_MSGPOOL 設定パラメータの値は、アプリケーションでこの 2 つのルーチンをどのように使用するかに基づいて決定する。
SRV_S_NETBUFSIZE	クライアント接続が使用するネットワーク I/O バッファの最大サイズ。明示的に設定しない場合、SRV_S_NETBUFSIZE はデフォルトの最大値である 8192 バイトになる。	Open Client Server 12.5.1 以前では、ログイン時にネットワーク・バッファのサイズが決定される。より小さなバッファ・サイズが要求される場合でも、Open Server はメモリ・バッファのサイズを変更しないで、その部分を未使用のままにしておく。したがって、この値を必要以上に大きくしないこと。必要以上に大きくすると、使用されないメモリが割り付けられる。
SRV_S_NETTRACEFILE	このファイルに書き込まれる Net-Library のトレース。	このプロパティの最大長は SRV_MAXCHAR バイト。

プロパティ名	定義	注意
SRV_S_NUMCONNECTIONS	Open Server アプリケーションが受け入れる物理ネットワーク接続の最大数。	サブチャネルがいくつ使用されるかに関係なく、サーバからサーバへの接続は物理接続 1 つだけである。たとえば、パススルー Open Server アプリケーション内の送信クライアント・ライブラリ接続は、CS_MAX_CONNECT プロパティにより制限される。CS_MAX_CONNECT は、ct_config() を使用して設定できる。
SRV_S_NUMLISTENERS	クライアント接続の受信に関連する SRV_PROC 制御構造体の数を返す。	これは、取得だけが可能なプロパティである。
SRV_S_NUMMSGQUEUES	Open Server アプリケーションに対して使用できるメッセージ・キューの数。	
SRV_S_NUMMUTEXES	Open Server アプリケーションに対して使用できる相互排他セマフォの数。	
SRV_S_NUMREMBUF	サーバからサーバへの接続で使用されるウィンドウ・サイズ。これは、受信確認が要求される前に論理サブチャネル上に未処理のまま残っているパケットの最大数を示す。	
SRV_S_NUMREMSITES	一定の時間にアクティブになっているリモート・サーバ・サイト・ハンドラの最大数。	
SRV_S_NUMTHREADS	Open Server アプリケーションに対して使用できるスレッドの最大数。	
SRV_S_NUMUSEREVENTS	アプリケーションが定義できるユーザ・イベントの数。	
SRV_S_PREEMPT	ブール値。このプロパティが CS_TRUE に設定されている場合、Open Server はプリエンプティブ・スケジューリングを使用する。このプロパティが CS_FALSE に設定されている場合、Open Server は非プリエンプティブ・スケジューリングを使用する。	すべてのプラットフォームでプリエンプティブ・スケジューリングを使用できるわけではない。srv_capability ルーチンを使用して、プリエンプティブ・スケジューリングを使用できるかどうかを調べること。 スレッド・ライブラリとともに SRV_S_PREEMPT を使用した場合、一部の SRV_S_CURTHREAD 機能が無効になる。
SRV_S_REALLOCFUNC	Open Server がメモリを再割り付けするのに使用するルーチンのアドレス。	
SRV_S_REQUEST_CAP	Open Server アプリケーションが受け入れるデフォルトのクライアント要求。	「機能」(22 ページ) を参照。
SRV_S_RESPONSE_CAP	Open Server アプリケーションがサポートしているクライアントへのデフォルト応答。	「機能」(22 ページ) を参照。
SRV_S_RETPARAMS	実行時にエラーが発生した場合にリターン・パラメータが送信される。	このサーバ・プロパティをデフォルト (false) に設定することで、特定のスレッドに対して動作を制限できる。

プロパティ名	定義	注意
SRV_S_SEC_KEYTAB	DCE セキュリティ・ドライバとともに使用するキータブ・ファイル名 (パス名を含む)。	アプリケーションを実行している現在のログイン・ユーザ以外のプリンシパルを指定できる。プロパティ SRV_S_SEC_PRINCIPAL はプリンシパル名を設定する。DCE ユーティリティ dcecp を使用して、キータブ・ファイルを作成できる。キータブ・ファイルは通常の UNIX ファイルであるため、このファイルにパーミッションを設定してアクセスを制限する必要がある。このファイルは、Open Server アプリケーションを起動するユーザが読み込み可能なファイルである必要がある。「セキュリティ・サービス」(158 ページ) を参照。 このプロパティの最大長は SRV_MAXCHAR バイト。
SRV_S_SEC_PRINCIPAL	Open Server アプリケーションのクレデンシャルを得るときに使用するプリンシパル名。 このプロパティのデフォルト値は Open Server アプリケーションのネットワーク名であるが、この名前は srv_init の実行時に指定できる。	このプロパティの最大長は SRV_MAXCHAR バイト。 「セキュリティ・サービス」(158 ページ) を参照。
SRV_S_SERVERNAME	Open Server アプリケーションの名前。	この名前は、Open Server アプリケーションが稼働しているときにそのアプリケーションを認識するのに使用される。また、interfaces ファイル内でそのアプリケーションの受信アドレスを検索する場合にも使用される。 このプロパティの最大長は SRV_MAXCHAR バイト。
SRV_S_SSL_CIPHER	CipherSuite 名をカンマで区切ったリスト。	このプロパティの最大長は SRV_MAXCHAR バイト。
SRV_S_SSL_LOCAL_ID	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。	このプロパティの最大長は SRV_MAXCHAR バイト。
SRV_S_SSL_REQUEST_CLIENT_CERT	クライアントが、Open Server アプリケーションにログインするためには証明書を提示することが必要なことを要求する。	
SRV_S_SSL_VERSION	定義されたいずれかの値でなければならない。	定義済みの値 • CS_SSLVER_20 • CS_SSLVER_30 • CS_SSLVER_TLS1 Adaptive Server Enterprise は、デフォルト (CS_SSLVER_TLS1) を使用した接続のみ受け付ける。

プロパティ名	定義	注意
SRV_S_STACKSIZE	各スレッドに割り付けられるスタックのサイズ。	
SRV_S_TDSVERSION	Open Server がすべてのクライアント接続のネゴシエーションに使用する TDS のプロトコル・バージョン。	値のリストについては、「SRV_S_TDSVERSION」(137 ページ) を参照。
SRV_S_TIMESLICE	アクティブなスレッド 1 つが消費するクロック・チックの数。この数に達すると、タイム・スライス・コールバック・ルーチンが呼び出される。	タイム・スライス・コールバックの詳細については、srv_callback のページを参照。
SRV_S_TRACEFLAG	適切なトレース・タイプ。	フラグのリストについては、「SRV_S_TRACEFLAG」(138 ページ) を参照。
SRV_S_TRUNCATELOG	ブール値。このプロパティが CS_TRUE に設定されている場合、Open Server は起動中にログ・ファイルをトランケートする。	SRV_S_TRUNCATELOG プロパティは、srv_init の呼出し後に設定できる。
SRV_S_USERVLANG	ブール値。このプロパティが CS_TRUE に設定されている場合、エラー・メッセージに Open Server アプリケーションのネイティブ言語が使用される。CS_FALSE に設定されている場合は、エラー・メッセージにクライアントの各国言語が使用される。	
SRV_S_VERSION	使用中の Open Server Server-Library の名前、バージョンのリリース年月日、版権情報が格納されている文字列。	
SRV_S_VIRTCLKRATE	1 チックあたりのクロック・レート (ミリ秒単位)。	
SRV_S_VIRTIMER	ブール値。このプロパティが CS_TRUE に設定されている場合、仮想タイマが有効になる。このプロパティが CS_FALSE に設定されている場合、仮想タイマは無効になる。	

SRV_S_TDSVERSION

クライアントのログイン処理中に、Open Server はクライアント・アプリケーションとネゴシエートして、TDS バージョンについて合意します。SRV_S_TDSVERSION プロパティ値は、Open Server の開始ポイントを決定します。クライアントは、この開始ポイント以下で通信することに合意します。その後で、ログイン処理中に Open Server アプリケーションは SRV_T_TDSVERSION スレッド・プロパティを使用して、特定の接続に対する TDS バージョンを再度ネゴシエートできます。詳細については、「スレッド・プロパティ」(139 ページ) を参照してください。

表 2-26 は、このプロパティに有効な値を示したものです。

表 2-26: SRV_S_TDSVERSION の値

SRV_S_TDSVERSION の値	意味
SRV_TDSNONE	認識できない TDS バージョン。
SRV_TDS_4.0	TDS 4.0 からネゴシエーションが開始される。
SRV_TDS_4_0_2	TDS 4.0.2 からネゴシエーションが開始される。
SRV_TDS_4_2	TDS 4.2 からネゴシエーションが開始される。
SRV_TDS_4_6	TDS 4.6 からネゴシエーションが開始される。
SRV_TDS_4_9_5	TDS 4.9.5 からネゴシエーションが開始される。
SRV_TDS_5_0	TDS 5.0 からネゴシエーションが開始される。

SRV_S_TRACEFLAG

SRV_S_TRACEFLAG プロパティはビットマップです。このプロパティのフラグは論理和をとることができます。表 2-27 は、このフラグの意味を示したものです。

表 2-27: SRV_S_TRACEFLAG の値

フラグ	意味
SRV_TR_ATTN	Open Server は、Open Server アプリケーションがアテンションを受信したかまたは受信確認したかを示す情報を表示する。
SRV_TR_DEFQUEUE	Open Server は、イベント・キューのアクティビティをトレースする。
SRV_TR_EVENT	Open Server は、トリガをかけたイベントについての情報を表示する。
SRV_TR_MSGQ	Open Server は、「メッセージ・キュー」のアクティビティをトレースする。
SRV_TR_NETDRIVER	Open Server は、TCL Net-Library ドライバ要求をトレースする。
SRV_TR_NETREQ	Open Server は、TCL 要求をトレースする。
SRV_TR_NETWAKE	Open Server は、TCL ウェイクアップ要求をトレースする。
SRV_TR_TDSDATA	Open Server は、TDS パケットの内容を 16 進数または ASCII 形式で表示する。これは、クライアントと Open Server アプリケーションとの間の実際の TDS トラフィックである。
SRV_TR_TDSHDR	Open Server は、パケットのタイプや長さなどの TDS プロトコル・パケットのヘッダ情報を表示する。

スレッド・プロパティ

スレッドとは、特定のタスクまたは一連のタスクを遂行するために実行する 1 つのコードです。Open Server スレッドにはいくつかのタイプがあります。スレッド・プロパティはスレッドの動作のさまざまな内容を定義し、そのリソースに制限を設定します。

Open Server スレッドの詳細については、「マルチスレッド・プログラミング」(102 ページ) を参照してください。

設定できるのはごくわずかのスレッド・プロパティだけですが、すべてのスレッド・プロパティを取得できます。アプリケーションは、srv_thread_props を呼び出して、スレッド・プロパティ値の取得と設定を行います。設定できるプロパティについては、srv_thread_props のページを参照してください。アプリケーションは、初期化の後ではいつでもスレッド・プロパティの取得と設定ができます。

Open Server は、初期化時にスレッドを作成するときに、設定できるそれぞれのスレッド・プロパティに対してデフォルトを割り当てます。デフォルトのリストについては、srv_thread_props (409 ページ) を参照してください。

表 *2-28:* スレッド・プロパティ

プロパティ名	定義	注意
SRV_T_APPLNAME	クライアント・アプリケーションの名前。	
SRV_T_BYTEORDER	クライアントが要求したバイト順スキーム。SRV_LITTLE_ENDIAN は、最小有効バイトが上位バイトであることを示す。SRV_BIG_ENDIAN は、最小有効バイトが下位バイトであることを示す。	
SRV_T_BULKTYPE	クライアントが送信するバルク転送のタイプ。	有効な値のリストについては、「SRV_T_BULKTYPE」(145 ページ) を参照。
SRV_T_CHARTYPE	文字データ表現のタイプ。	有効な値のリストについては、「SRV_T_CHARTYPE」(146 ページ) を参照。
SRV_T_CIPHER_SUITE	CS_CHAR* CipherSuite は、SSL ベースのセッション中に交換されるデータの暗号化や暗号の解読に使用する。CipherSuite は、接続ハンドシェイク中にネゴシエートされる。	このプロパティは、SRV_LISTEN_PREBIND Open Server イベント内から設定できます。
SRV_T_CLIB	Open Server アプリケーションに接続するのにクライアントが使用するライブラリ製品の名前。	
SRV_T_CLIBVERS	Open Server アプリケーションへの接続にクライアントが使用するライブラリ製品のバージョン。	

プロパティ名	定義	注意
SRV_T_CLIENTLOGOUT	ブール値。クライアントが正常ログアウトとアボート・ログアウトのどちらを完了したかを示す。CS_TRUE は正常ログアウトを示す。	このプロパティは、SRV_DISCONNECT イベント・ハンドラ内部からのみ取得できる。
SRV_T_CONVERTSHORT	ブール値。4 バイトの datetime データ型、4 バイトの float データ型、4 バイトの money データ型を、8 バイトのデータ型に自動的に変換するかどうかを示す。	
SRV_T_DUMPLOAD	ブール値。このクライアント接続に対してダンプ／ロードとバルク挿入の使用を禁止するかどうかを示す。	
SRV_T_ENDPOINT	接続されたクライアントのファイル記述子またはファイル処理。サブチャネルの場合、サイト・ハンドラの終了ポイント値が返される。 SRV_T_ENDPOINT は Client-Library での CS_ENDPOINT 値に相当する。	このプロパティは、クライアント・スレッド、サイト・ハンドラ、サブチャネルには有効であるが、サービス・スレッドには有効ではない。 SRV_T_ENDPOINT の使用例については、「SRV_T_ENDPOINT」(146 ページ) を参照。
SRV_T_EVENT	スレッドが現在存在している Open Server イベント。	有効な値のリストについては、「SRV_T_EVENT」(146 ページ) を参照。
SRV_T_EVENTDATA	Open Server アプリケーションによって発生する特定のイベントと対応する汎用データ・アドレス。	srv_event ルーチンを使用して設定されるデータ・アドレス。
SRV_T_FULLPASSTHRU	ブール値。このプロパティが CS_TRUE に設定されている場合、SRV_FULLPASSTHRU イベント・ハンドラはそのスレッドに対してアクティブになる。	Open Server アプリケーションの接続ハンドラ内部からのみ設定できる。 SRV_T_EVENT プロパティの値は、フル・パススルー・イベント・ハンドラ内部から取得される場合は SRV_FULLPASSTHRU である。
SRV_T_FLTTYPE	クライアントが使用する浮動小数点表現のタイプ。	有効な値のリストについては、「SRV_T_FLTTYPE」(147 ページ) を参照。
SRV_T_GOTATTENTION	ブール値。クライアント・スレッドがアテンションを受信したかどうかを示す。	
SRV_T_HOSTNAME	クライアント接続が発生したホスト・マシンの名前。	
SRV_T_HOSTPROCID	クライアント・プログラムのプロセス ID。	これは、クライアントのログイン・レコード内で受信されるオペレーティング・システム・プロセス ID である。
SRV_T_IODEAD	ブール値。スレッドの I/O チャネルが有効かどうかを示す。	CS_TRUE は、スレッドで I/O を正常に実行できないことを示し、CS_FALSE は正常に実行できることを示す。Open Server は、サービス・スレッドに対しては必ず CS_FALSE を返す。

プロパティ名	定義	注意
SRV_T_LISTENADDR	指定された SRV_PROC 制御構造体で識別されたリスナのアドレスを返す。SRV_PROC 制御構造体がリスナの場合、このプロパティは、リスナが接続を受け入れるアドレスを返す。	このプロパティには、引数として CS_TRANADDR 構造体へのポインタが必要。 これは、取得だけが可能なプロパティである。
SRV_T_LOCALE	Open Server アプリケーションが割り付ける CS_LOCALE 構造体へのポインタ。	このプロパティを使用して、ローカライゼーション情報の取得または設定を行うこと。
SRV_T_LOCALID	リスナで使用する SSL 証明書を指定する。	このプロパティを使用すると、リスナはグローバル・サーバ・レベルの SSL 証明書とは異なる SSL 証明書を使用できる。 これは、設定専用プロパティである。
SRV_T_LOGINTYPE	受信されるログイン・レコードのタイプ。	有効な値のリストについては、「SRV_T_LOGINTYPE」(147 ページ) を参照。
SRV_T_MIGRATED	ブール値。接続が新しい接続であるか、マイグレートされた接続であるかを示す。この読み込み専用プロパティは、クライアントがマイグレート中であるか、サーバにマイグレートした場合は true に設定される。	詳細については、「SRV_T_MIGRATED」(148 ページ) を参照。
SRV_T_MIGRATE_STATE	クライアントのマイグレーション・ステータスを示す。どのスレッドでもアクセスできる読み取り専用プロパティ。	詳細については、「SRV_T_MIGRATE_STATE」(148 ページ) を参照。
SRV_T_MACHINE	クライアント・スレッドが実行されているマシンのホスト名。	
SRV_T_NEGLOGIN	クライアントからの要求があった場合に、ネゴシエートされたログインのタイプ。	このプロパティはビットマスクであり、次に示す 5 つの値のいずれかを取る。 • SRV_CHALLENGE は、クライアントがチャレンジ／応答の交換によってネゴシエートしようとしていることを通知する。 • SRV_ENCRYPT は、クライアントが対称暗号化パスワードを渡そうとしていることを通知する。 • SRV_SECLABEL は、クライアントがセキュリティ・ラベルを送信することを示す。 • SRV_APPDEFINED は、アプリケーションで定義されたログイン・ハンドシェイクが使用されていることを示す。 • SRV_EXTENDED_ENCRYPT は、クライアントが非対称暗号化パスワードを渡そうとしていることを通知する。

プロパティ名	定義	注意
SRV_T_NOTIFYCHARSET	ブール値。使用中の文字セットが変更されたときにクライアントに通知するかどうかを示す。	
SRV_T_NOTIFYDB	ブール値。Transact-SQL コマンド use db の結果をクライアントに通知するかどうかを示す。	
SRV_T_NOTIFYLANG	ブール値。使用中の各国言語が変更されたときにクライアントに通知するかどうかを示す。	
SRV_T_NOTIFYPND	クライアントに配信する必要がある保留中の通知の数。	このプロパティは取得だけが可能である。
SRV_T_NUMRMTPWDS	リモート・パスワードの数。	
SRV_T_PACKETSIZE	クライアントとの通信に使用されるネゴシエートされたパケット・サイズ。	パケット・サイズは、ログイン時に透過的にネゴシエートされる。
SRV_T_PASSTHRU	ブール値。クライアント・スレッドがパススルー・モードで動作しているかどうかを示す。	このプロパティは、アプリケーションの接続ハンドラ内部から設定できる。 このプロパティが CS_TRUE に設定されている場合、srv_getloginfo ルーチンと ct_setloginfo ルーチンは、Open Server の機能に関係なくクライアント接続の機能をネゴシエートする。フル・パススルー・ゲートウェイではさまざまなコマンドと結果のタイプは認識されないので、これは適切な動作である。
SRV_T_PRIORITY	Open Server がスレッドに対して設定する予定の優先順位レベル。	このプロパティは取得だけが可能である。スレッドの優先順位を設定するには、srv_setpri を呼び出す。
SRV_T_PWD	クライアントがログイン・レコード内に送信したパスワード文字列。	リモート・サーバ接続の場合、このプロパティはリモート・サーバのパスワードを返す。
SRV_T_REMOTEADDR	SRV_PROC ピアのアドレスを返す。このプロパティは、クライアントの SRV_PROC に対してのみ有効である。	このプロパティには、引数として CS_TRANADDR 構造体へのポインタが必要。 これは、取得だけが可能なプロパティである。
SRV_T_RETPARAMS	実行時にエラーが発生した場合にリターン・パラメータが送信される。	SRV_S_RETPARAMS が設定されている場合は、RPC のリターン動作がすべてのスレッドに適用される。
SRV_T_RMTCERTIFICATE	CS_SSLCERT * クライアントの証明書を表すポインタ。	
SRV_T_RMTPWDS	SRV_RMTPWD の配列。	この構造体の定義については、「SRV_T_RMTPWDS」(149 ページ) を参照。
SRV_T_RMTSERVER	クライアント接続の場合は、ローカル・サーバ名。サーバからサーバへの接続の場合は、リモート・サーバ名。	

プロパティ名	定義	注意
SRV_T_ROWSENT	このイベントでクライアントに返されるローの数。	
SRV_T_SEC_CHANBIND	このスレッドと対応するクライアント／サーバ接続で、チャネル・バインドが使用されているかどうかを示すブール値。	
SRV_T_SEC_ CONFIDENTIALITY	このスレッドと対応するクライアント／サーバ接続で、データ機密保持サービスが使用されているかどうかを示すブール値。	このプロパティは通常、データの暗号化技術を使用して実装される。
SRV_T_SEC_ CREDTIMEOUT	このスレッドと対応するクライアント／サーバ接続で、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)。	次のいずれかの値を取る。 • CS_NO_LIMIT − 制限時間はなく、有効期限が切れることはない。 • CS_UNEXPIRED − まだ有効期限が切れていない。 • 0 − 有効期限が切れた。 • 正の数 − 有効期限が切れるまでの残り時間 (秒数)。
SRV_T_SEC_DATAORIGIN	このスレッドと対応するクライアント／サーバ接続で、データ・オリジン・サービスが使用されているかどうかを示すブール値。	
SRV_T_SEC_DELEGATION	クライアントによって委任が有効になっているかどうかを示すブール値。	このスレッドで行われるすべての作業は、クライアントの権限レベルを使用する必要がある。プリンシパル名にアクセスするには、SRV_T_USER プロパティを使用すること。別のセキュリティ・ピアを使用してセキュリティ・セッションを開始するときに使用する委任クレデンシャルを得るには、SRV_T_SEC_DELEGCRED プロパティを使用すること。
SRV_T_SEC_DELEGCRED	現在のセキュリティ・セッションでのクライアントの委任クレデンシャル (存在する場合)。	SRV_T_SEC_DELEGATION プロパティは、クライアントによって委任が有効になっているかどうかを示す。委任が有効になっている場合、Open Server アプリケーションは SRV_T_SEC_DELEGCRED プロパティを使用して、委任クレデンシャルを得ることがある。
SRV_T_SEC_ DETECTREPLAY	このスレッドと対応するクライアント／サーバ接続で、メッセージ・リプレイの検出サービスが使用されているかどうかを示すブール値。	
SRV_T_SEC_DETECTSEQ	このスレッドと対応するクライアント／サーバ接続で順序不整合の検出のメッセージ・サービスが使用されているかどうかを示すブール値。	

プロパティ名	定義	注意
SRV_T_SEC_INTEGRITY	このスレッドと対応するクライアント／サーバ接続で、整合性サービスが使用されているかどうかを示すブール値。	通常このプロパティは、暗号化シグニチャを使用して実装される。
SRV_T_SEC_MECHANISM	このスレッドと対応するクライアント／サーバ接続で使用されているセキュリティ・メカニズムのローカル名。	
SRV_T_SEC_MUTUALAUTH	このスレッドと対応するクライアント／サーバ接続で、相互認証が実行されたかどうかを示すブール値。	
SRV_T_SEC_NETWORKAUTH	このスレッドと対応するクライアント／サーバ接続で、ネットワーク認証が実行されたかどうかを示すブール値。	
SRV_T_SEC_SESSTIMEOUT	このスレッドと対応するクライアント／サーバ接続で、セキュリティ・セッションの有効期限が切れるまでの残り時間 (秒数)。	次のいずれかの値を取る。 • CS_NO_LIMIT − 制限時間はなく、有効期限が切れることはない。 • CS_UNEXPIRED − まだ有効期限が切れていない。 • 0 − 有効期限が切れた。 • 正の数 − 有効期限が切れるまでの残り時間 (秒数)。
SRV_T_SESSIONID	クライアントが Open Server に送信するセッション ID を取得する。さらに、セッション ID がクライアントに送信されるように SRV_CONNECT ハンドラで設定する。	詳細については、「SRV_T_SESSIONID」(149 ページ) を参照。
SRV_T_SSL_VERSION	接続ハンドシェイク中にネゴシエートされた SSL/TLS プロトコル・バージョン。	このプロパティは、SRV_LISTEN_PREBIND Open Server イベント内から設定できます。
SRV_T_SPID	スレッドのプロセス ID。	このプロパティは、このスレッドに割り当てられたユニークな ID である。スレッドが終了すると、スレッド ID は再使用される。
SRV_T_STACKLEFT	スレッドに使用できる未使用スタックのサイズ。	
SRV_T_TDSVERSION	クライアント・スレッドが使用している TDS のバージョン。	SRV_CONNECT イベント・ハンドラ内でこのスレッドを設定すると、Open Server アプリケーションは、そのスレッドに対する Open Server のデフォルト以外の値に TDS バージョンをネゴシエートできる。有効な値のリストについては、「SRV_T_TDSVERSION」(150 ページ) を参照。
SRV_T_TYPE	スレッド・タイプ。	有効な値のリストについては、「SRV_T_TYPE」(150 ページ) を参照。

プロパティ名	定義	注意
SRV_T_USER	クライアント・スレッドがログインしたときに使用したユーザ名。	
SRV_T_USERDATA	アプリケーションごとに異なる目的で使用される汎用データ・アドレス。	このプロパティは設定可能である。
SRV_T_USESRVLANG	ブール値。サーバの各国言語でエラー・メッセージを表示させる場合には、このプロパティを CS_TRUE に設定する。クライアントの各国言語でエラー・メッセージを表示する場合には、このプロパティを CS_FALSE に設定する。	スレッドのサーバワイドな SRV_S_USESRVLANG プロパティを上書きするには、このプロパティを設定する。
SRV_T_USTATE	スレッドの現在のステータスを記述する文字列。	このプロパティは設定可能である。

SRV_T_BULKTYPE

クライアント・アプリケーションから Open Server アプリケーションに転送可能なバルク・データには、バルク・コピー・データ、text データ、image データの 3 種類があります。クライアントによって開始されるバルク・データ転送のタイプの設定または取得を行うには、SRV_T_BULKTYPE プロパティを使用します。

表 2-29 は、SRV_T_BULKTYPE スレッド・プロパティの有効値をまとめたものです。

表 *2-29: SRV_T_BULKTYPE* の値

値	意味
SRV_BULKLOAD	クライアントはバルク・コピー・データを転送する準備をしている。
SRV_TEXTLOAD	クライアントは text データを転送する準備をしている。
SRV_IMAGELOAD	クライアントは image データを転送する準備をしている。
SRV_UNITEXTLOAD	クライアントは unitext データを転送する準備をしている。

Open Server は、クライアントが送信するバルク・データ・ストリームのタイプを自動的に決定できません。Open Server アプリケーションは srv_thread_props ルーチンを使用してこの情報を取得し、実際の SRV_BULK イベントより前にその情報を Open Server に提供する必要があります。その後アプリケーションは、一度実際のバルク要求が行われていれば、SRV_BULK イベント・ハンドラ内部からデータを取得します。

『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。「text と image」(184 ページ) を参照してください。

SRV_T_CHARTYPE

クライアント・アプリケーションは、文字データが特定の方法で表現されると仮定します。クライアントが想定している文字データ表現方法を Open Server アプリケーション側で取得するには、property を SRV_T_CHARTYPE に設定し、cmd を CS_GET に設定して srv_thread_props を呼び出します。クライアントは *bufp で次の値を返します。

表 *2-30:* 文字データ表現

値	意味
SRV_CHAR_ASCII	ASCII 文字フォーマット
SRV_CHAR_EBCDIC	EBCDIC 文字フォーマット
SRV_CHAR_UNKNOWN	認識できない文字フォーマット

SRV_T_ENDPOINT

次の例は、SRV_T_ENDPOINT の使用方法を示します。

```
CS_INT ep;
/*
 ** Get the end point
 */

if(srv_thread_props(spp, CS_GET, SRV_T_ENDPOINT, (CS_VOID *)&ep,
     CS_SIZEOF(ep), (CS_INT *)NULL) == CS_FAIL)
{
    return(CS_FAIL);
}
```

SRV_T_EVENT

スレッドは一度に 1 つの特定のイベント・ハンドラを実行します。イベントに対応するイベント・ハンドラを実行するときに、スレッドはそのイベントの内部にあると言えます。Open Server アプリケーションは、property を SRV_T_EVENT に、cmd を CS_GET に設定して srv_thread_props を呼び出し、スレッドが内部にあるイベントを取得できます。アプリケーションが複数のイベントに対して同じイベント・ハンドラ・コードを使用する場合は、この手順が便利です。

このようなイベントには、次のものがあります。

- SRV_ATTENTION
- SRV_BULK
- SRV_CONNECT
- SRV_CURSOR
- SRV_DISCONNECT
- SRV_DYNAMIC

- SRV_FULLPASSTHRU
- SRV_LANGUAGE
- SRV_LISTEN_PREBIND
- SRV_LISTEN_POSTBIND
- SRV_MSG
- SRV_OPTION
- SRV_RPC
- SRV_START
- SRV_STOP
- ユーザ定義イベント

「イベント」(84 ページ) を参照してください。

SRV_T_FLTTYPE

クライアント・アプリケーションは、浮動小数点データが特別な方法で表現されると仮定します。クライアントが想定している浮動小数点データ表現方法を Open Server アプリケーション側で取得するには、property を SRV_T_FLTTYPE に設定し、cmd を CS_GET に設定して srv_thread_props を呼び出します。クライアントは、bufp が示すアドレス領域内に次の値のいずれかを返します。

- SRV_FLT_IEEE － IEEE の浮動小数点フォーマット
- SRV_FLT_ND5000 － ND5000 の浮動小数点フォーマット
- SRV_FLT_ VAX － VAX ‘D’ の浮動小数点フォーマット
- SRV_FLT_UNKNOWN － 認識できない浮動小数点フォーマット

SRV_T_LOGINTYPE

Open Server アプリケーションは、ログイン処理中にさまざまなタイプのスレッド・ログイン・レコードのすべてを受信できます。SRV_T_LOGINTYPE プロパティはログイン・タイプを示します。アプリケーションは、property を SRV_T_LOGINTYPE に、cmd を CS_GET に設定して srv_thread_props を呼び出し、ログイン・タイプを取得できます。このログイン・タイプは、bufp が示すバッファ内に返されます。表 2-31 に、それぞれのログイン・タイプを示します。

表 2-31: スレッド・ログイン・タイプ

値	ログイン・タイプ
SRV_SITEHANDLER	リモート・サーバからのサイト・ハンドラ・ログイン要求
SRV_SUBCHANNEL	リモート・サーバからのサイト・ハンドラ・サブチャネル・ログイン
SRV_CLIENT	クライアント・アプリケーションからのログイン要求

SRV_T_MIGRATED

接続が新しい接続であるか、マイグレートされた接続であるかを示す Boolean プロパティです。この読み込み専用プロパティは、クライアントがマイグレート中であるか、サーバにマイグレートした場合は true に設定されます。次のサンプル・コードは、SRV_T_MIGRATED の値を取得します。

```
CS_RETCODE ret;
CS_BOOL migrated;
status = srv_thread_props(sp, CS_GET, SRV_T_MIGRATED,
   &migrated, sizeof (migrated), NULL);
```

詳細については、「接続マイグレーション」(35 ページ) を参照してください。

SRV_T_MIGRATE_STATE

SRV_T_MIGRATE_STATE は、クライアントのマイグレーション・ステータスを示します。どのスレッドでもアクセスできる読み取り専用プロパティです。示されるマイグレーション・ステータスは次のとおりです。

ステータス	値	説明
SRV_MIG_NONE	0	進行中のマイグレーションがない。
SRV_MIG_REQUESTED	1	マイグレーションがサーバにより要求された。
SRV_MIG_READY	2	クライアントが要求を受信し、マイグレーションの準備ができた。
SRV_MIG_MIGRATING	3	クライアントが指定されたサーバにマイグレート中である。
SRV_MIG_CANCELLED	4	マイグレーション要求がキャンセルされた。
SRV_MIG_FAILED	5	クライアントがマイグレーションに失敗した。

SRV_MIG_STATE は、SRV_T_MIGRATE_STATE プロパティをモデル化する列挙データ型です。SRV_MIG_STATE は、次のように宣言されます。

```
typedef enum
{
   SRV_MIG_NONE,
   SRV_MIG_REQUESTED,
   SRV_MIG_READY,
   SRV_MIG_MIGRATING,
   SRV_MIG_CANCELLED,
   SRV_MIG_FAILED
} SRV_MIG_STATE;
```

次のサンプル・コードは、SRV_T_MIGRATE_STATE 値を取得する方法を示しています。マイグレーションに成功した場合、クライアントが終了し、SRV_DISCONNECT イベント・ハンドラが SRV_MIG_MIGRATING ステータスで呼び出されます。

```
CS_RETCODE ret;
SRV_MIG_STATE migration_state;
ret = srv_thread_props(sp, CS_GET, SRV_T_MIGRATE_STATE,
   &migration_state, sizeof (migration_state), NULL);
if (ret != CS_SUCCEED)
{
...
}
```

詳細については、「接続マイグレーション」(35 ページ) を参照してください。

SRV_T_RMTPWDS

アプリケーションは、SRV_T_RMTPWDS プロパティを使用してリモート・サーバの名前とパスワードを取得します。この名前とパスワードが格納される SRV_T_RMTPWD 構造体の定義は次のとおりです。

```
typedef struct srv_rmtpwd
 {
     CS_INT servnamelen;
     CS_BYTEservname[CS_MAX_NAME];
     CS_INTpwdlen;
     CS_BYTEpwd[CS_MAX_NAME];
 } SRV_RMTPWD;
```

SRV_T_SESSIONID

SRV_T_SESSIONID は、クライアントから Open Server に送信されたセッション ID を取得するスレッド・プロパティです。Open Server アプリケーションは、次の場合に srv_thread_props() 関数を使用して SRV_T_SESSIONID プロパティを設定することもできます。

- srv_thread_props(CS_SET, SRV_T_SESSIONID) 呼び出しが SRV_CONNECT イベント・ハンドラ内で行われた。
- クライアントで接続マイグレーションまたは高可用性がサポートされている。

次のサンプル・コードは、SRV_T_SESSIONID プロパティを設定します。

```
CS_RETCODE ret;
CS_SESSIONID hasessionid;
ret = srv_thread_props(sp, CS_SET, SRV_T_SESSIONID,
   hasessionid, sizeof(hasessionid), NULL);
```

SRV_T_TDSVERSION

クライアントのログイン処理中に、Open Server はクライアント・アプリケーションとネゴシエートして、すべてのスレッドに対してある TDS バージョンで合意します。SRV_S_TDSVERSION プロパティ値は、Open Server の開始ポイントを決定します。クライアントは、この開始ポイント以下で通信することに合意します。SRV_S_TDSVERSION プロパティの詳細については、「スレッド・プロパティ」(139 ページ) を参照してください。その後で、ログイン処理中に Open Server アプリケーションは SRV_T_TDSVERSION プロパティを使用して、特定のスレッドに対して TDS バージョンを再度ネゴシエートできます。

表 2-32 は、このプロパティに有効な値を示したものです。

表 *2-32: SRV_T_TDSVERSION* の値

SRV_T_TDSVERSION の値	意味
SRV_TDSNONE	認識できない TDS バージョン。
SRV_TDS_4.0	TDS 4.0 からネゴシエーションが開始される。
SRV_TDS_4_0_2	TDS 4.0.2 からネゴシエーションが開始される。
SRV_TDS_4_2	TDS 4.2 からネゴシエーションが開始される。
SRV_TDS_4_6	TDS 4.6 からネゴシエーションが開始される。
SRV_TDS_4_9_5	TDS 4.9.5 からネゴシエーションが開始される。
SRV_TDS_5_0	TDS 5.0 からネゴシエーションが開始される。

SRV_T_TYPE

Open Server スレッドにはいくつかのタイプがあります。SRV_T_TYPE スレッド・プロパティは、スレッドのタイプを示します。アプリケーションでスレッドのタイプを取得するには、property を SRV_T_TYPE に設定し、cmd を CS_GET に設定して srv_thread_props を呼び出します。

表 2-33 は、有効なスレッド・タイプを示したものです。

表 *2-33:* スレッドの種類

値	スレッド・タイプ
SRV_TCLIENT	クライアント・スレッド
SRV_TLISTENER	クライアント接続を受け入れるスレッド
SRV_TSITE	サイト・ハンドラ・スレッド
SRV_TSUBPROC	サイト・ハンドラ・スレッド上のリモート・サーバ接続
SRV_TSERVICE	サービス・スレッド

「マルチスレッド・プログラミング」(102 ページ) を参照してください。

レジスタード・プロシージャ

レジスタード・プロシージャとは、名前によって識別されたコードです。アプリケーションはプロシージャを登録するときに、プロシージャ名をルーチンにマップします。これにより、Open Server は、受信 RPC データ・ストリーム内でそのプロシージャ名を検出すると、SRV_RPC イベントを発生させずに特定のルーチンをただちに呼び出すことができます。

Open Server が RPC を 受け取ると、Open Server はレジスタード・プロシージャのリストでプロシージャ名を探します。名前が登録されている場合、ランタイム・システムは、存在するレジスタード・プロシージャと関連しているルーチンすべてを実行します。プロシージャ名がレジスタード・プロシージャのリストに見つからない場合には、Open Server は SRV_RPC イベント・ハンドラを呼び出します。

標準リモート・プロシージャ・コール

Open Server アプリケーションは、通常の RPC を、アプリケーションの SRV_RPC イベント・ハンドラ内から処理します。ハンドラ・コードは、RPC データ・ストリームを解析して、このプロセスの RPC 名、パラメータの数、パラメータのフォーマット、パラメータ値を取得しなければなりません。その後、ハンドラはこれらの値に基づいてアクションを起こすことができます。SRV_RPC イベント・ハンドラは、ネットワーク経由で送られてくると考えられるすべての RPC を処理できるようにコーディングされている必要があります。

レジスタード・プロシージャの利点

次の理由により、レジスタード・プロシージャを利用すると、Open Server アプリケーションでの RPC 処理が単純化されます。

- レジスタード・プロシージャは、コードを 1 か所に統合します。レジスタード・プロシージャは、Open Server アプリケーションが SRV_RPC イベント・ハンドラに加えて、他のイベント・ハンドラから呼び出すことができる実行可能オブジェクトです。
- レジスタード・プロシージャは、Server-Library 呼び出しまたは外部の Client-Library 呼び出しや DB-Library 呼び出しによって、サーバが稼動しているときであればいつでも作成できます。それに対して、SRV_RPC イベント・ハンドラは、サーバの起動の前にあらかじめコーディングされていなければなりません。
- レジスタード・プロシージャには自動データ型チェック機能があるので、Open Server アプリケーション・コードの解析は必要ありません。

- クライアントは、レジスタード・プロシージャの実行時に通知 (ノーティフィケーション) を要求できます。この「通知」は、次の要素で構成されます。
  - レジスタード・プロシージャ名
  - レジスタード・プロシージャの実行に関連するパラメータ値
- 通知要求は、Server-Library 呼び出しにより内部的に、または Client-Library または DB-Library 呼び出しにより外部的に発行できます。
- クライアントは、レジスタード・プロシージャのリストや自分が通知を要請したプロシージャのリストを要求することができます。

ノーティフィケーション・プロシージャ

プログラマが提供するコードがない状態で、Open Server アプリケーションは、Client-Library または DB-Library クライアントがレジスタード・プロシージャを作成し、それを実行し、その実行の通知 (ノーティフィケーション) を受け取ることができるようにします。

レジスタード・プロシージャは、Open Server アプリケーションに実行可能ルーチンを持っている必要はありません。実際、DB-Library や Client-Library 呼び出しで作成されたレジスタード・プロシージャは、Open Server のルーチンを呼び出すことができません。実行可能なルーチンと対応付けられていないレジスタード・プロシージャは、実行を待機しているクライアントへの通知が唯一の目的なので、「ノーティフィケーション (通知) プロシージャ」と呼ばれます。

ノーティフィケーション・プロシージャを使用すれば、クライアント・アプリケーションはどの Open Server アプリケーションによってでも相互に通信できます。

この機能を使用可能にするためにコードを書く必要はありませんが、レジスタード・プロシージャを使用不可にしたり制限したりするために、コールバック・ハンドラをインストールすることができます。詳細については、「レジスタード・プロシージャでのコールバック・ハンドラの使用」(155 ページ) を参照してください。

レジスタード・プロシージャの作成

Open Server アプリケーションは、標準レジスタード・プロシージャとノーティフィケーション・プロシージャの両方を作成することができます。Client-Library と DB-Library アプリケーションは、ノーティフィケーション・プロシージャを作成することができます。Client-Library ルーチンを使ってレジスタード・プロシージャを作成する方法については、『Open Client Client-Library/C リファレンス・マニュアル』を参照してください。

レジスタード・プロシージャのメカニズム

この項では、Open Server アプリケーション内でレジスタード・プロシージャを作成して実行する方法について説明します。

レジスタード・プロシージャ

Open Server 呼び出しによってプロシージャを登録するには、次の手順に従います。

1　プロシージャ名を定義し、プロシージャが実行されるときに呼び出される関数に名前をマップするために、srv_regdefine を呼び出します。

2　定義されるプロシージャのパラメータを記述するために、srv_regparam を呼び出します。

3　プロシージャの登録を完了するために、srv_regcreate を呼び出します。

4　プロシージャの登録を解除するために、srv_regdrop を呼び出します。

レジスタード・プロシージャの実行

RPC が登録されている場合には、Open Server はクライアントまたはリモート Adaptive Server Enterprise による RPC に応答してレジスタード・プロシージャを実行します。しかし、Open Server アプリケーションは、RPC への応答の中でそれを実行する代わりに、レジスタード・プロシージャを明示的に実行することができます。たとえば、アプリケーションが、特定のノーティフィケーション・プロシージャをアプリケーションの特定の時点で実行することによって、複数のクライアントのアクティビティを同期できます。

レジスタード・プロシージャを明示的に実行するにも、いくつかの手順が必要です。それらは、次のとおりです。

1　レジスタード・プロシージャの実行を開始するために、srv_reginit を呼び出します。このルーチンは、実行されるレジスタード・プロシージャの名前を指定します。Open Server アプリケーションは、通知リストのクライアント・スレッドの 1 つが通知されるのか、またはすべてが通知されるのかを調べるためにも、このルーチンを使用します。

2　実行のためのパラメータ・データを提供するために、srv_regparam を呼び出します。

3　レジスタード・プロシージャを実際に実行するために、srv_regexec を呼び出します。

リストの管理

Open Server アプリケーションは、すべてのレジスタード・プロシージャのリストと、特定のレジスタード・プロシージャが実行されたときに通知するクライアントのリストを管理しています。この通知は、自動的に起こります。リストの管理に関連しているルーチンは次のとおりです。

- srv_reglist － Open Server アプリケーション内で登録されているすべてのプロシージャのリストを返します。
- srv_regwatchlist － 指名されたクライアント・スレッドが通知要求待ちを示す、対象となるレジスタード・プロシージャのすべてのリストを返します。
- srv_regwatch － レジスタード・プロシージャの通知リストにスレッドを追加します。
- srv_regnowatch － 特定のレジスタード・プロシージャの通知リストからクライアントを削除します。
- srv_reglistfree － srv_reglist または srv_regwatchlist によって割り付けられた SRV_PROCLIST 構造体を解放します。

システム・レジスタード・プロシージャ

各 Open Server アプリケーションは、「システム・レジスタード・プロシージャ」と呼ばれる組み込みレジスタード・プロシージャを持っています。これらは、サーバが起動する時点でランタイム・システムが作成します。システム・レジスタード・プロシージャについては、「第 4 章 システム・レジスタード・プロシージャ」で説明されています。これらのプロシージャの中には、Open Server アプリケーションを対話型で管理するために役立つものもあります。たとえば、sp_who と sp_ps を使用すると、アクティブなサーバ・プロセスをリストできます。また、sp_terminate を使用すると、プロセスを消滅させることができます。

クライアント・アプリケーションでは、システム・レジスタード・プロシージャを実行することで、次の処理を行うことができます。

- レジスタード・プロシージャのリストの取得
- レジスタード・プロシージャの実行
- レジスタード・プロシージャの実行の通知の要求
- 通知要求のリストの取得

ほとんどのシステム・レジスタード・プロシージャは、同等の Open Server ルーチンに対応します。Open Server アプリケーションとクライアントは、異なるルーチンによって同じ種類の情報を要求できます。

表 2-34 は、システム・レジスタード・プロシージャと対応する Server-Library ルーチンをまとめたものです。

表 *2-34:* システム・レジスタード・プロシージャと対応する *Server-Library* ルーチン

システム・レジスタード・プロシージャ	Server-Library ルーチン
sp_ps	該当なし
sp_regcreate	srv_regcreate/srv_regdefine
sp_regdrop	srv_regdrop
sp_reglist	srv_reglist
sp_regnowatch	srv_regnowatch
sp_regwatch	srv_regwatch
sp_regwatchlist	srv_regwatchlist
sp_serverinfo	該当なし
sp_terminate	srv_termproc
sp_who	該当なし

レジスタード・プロシージャでのコールバック・ハンドラの使用

表 2-34 に示されているように、組み込みレジスタード・プロシージャのいくつかは、レジスタード・プロシージャを作成、削除、実行する Server-Library ルーチンおよび DB-Library ルーチンに相当します。これらのプロシージャは、レジスタード・プロシージャが実行しようとするときにいつでも実行するようなコールバック・ハンドラをインストールすることによって、レジスタード・プロシージャのためのセキュリティ・システムを実装できるようにします。クライアント・アプリケーションがシステム・レジスタード・プロシージャまたはそれに相当する Client-Library や DB-Library ルーチンのひとつを実行すると、コールバック・ハンドラが実行されます。SRV_S_INHIBIT を返した場合は、レジスタード・プロシージャは実行されません。

たとえば、“sa” 以外のクライアントが “reinitialize” という名前のプロシージャを実行できないようにするには、レジスタード・プロシージャ・コールバック・ハンドラに次のコードを含めます。

```
/*
 **   Stop users other than "sa" from executing the "reinitialize"
 **   registered procedure.
 **
 ** Parameters:
 **    spp - Handle to the current client connection.
 **
 ** Returns:
 **    CS_TRUE  Allow the user to execute
 **    CS_FALSE Disallow execution.
 */
 CS_BOOL    rpc_permission(spp)
 SRVPROC    *spp;
 {
```

```
        CS_INT ulen;        /* User name length  */
        CS_INT rlen;        /* RPC name length   */
        CS_CHAR *rname;     /* Pointer to the RPC name */
        CS_CHAR user[256]; /* Buffer for the user name */

        /*
        ** Get the name of the rpc command
        */
        if ((rname = srv_rpcname(spp, &rlen)) == (CS_CHAR *)NULL)
        {
        return (CS_FALSE);
        }

        /*
        ** Get the user name.
        */
        if (srv_thread_props(spp, CS_GET, SRV_T_USER,
        (CS_VOID *)user,CS_SIZEOF(user), &ulen) == CS_FAIL)
        {
        return (CS_FALSE);
        }

        /*
        ** If either the user name or the rpc name is NULL,
        ** indicate an error.
        */
        if (rlen <= 0 || ulen <= 0)
        {
              error ("API error");
              return (CS_FALSE);
        }

        /* Null terminate the user name buffer */
        user[ulen] == '¥0';

        /*
        ** Compare the RPC name and User name for permission.
        */
        if ((strcmp(rname, "reinitialize") == 0) &&
         (strcmp(user, "sa") == 0))
        {
        return (CS_TRUE);
        }

        return (CS_FALSE);
}
```

例

サンプル・プログラム *regproc.c* には、Open Server アプリケーションによるレジスタード・プロシージャの使用例が記述されています。

リモート・プロシージャ・コール

リモート・プロシージャ・コール (「*RPC*」) は、クライアント・アプリケーションが Open Server アプリケーションと通信するためのメカニズムです。通常は、クライアントは、Open Server アプリケーションから情報を得るために RPC を発行します。RPC は、名前と、多くの場合 (必ずとは限らない) パラメータから構成されています。たとえば、デパートのアプリケーションは、get_cust という RPC への応答として顧客の氏名と住所を返します。この RPC は、1 つのパラメータ、つまり顧客 ID 番号を受け取ります。

クライアントが RPC を送ると、Open Server はその RPC が登録済みかどうかをチェックします。「レジスタード・プロシージャ」は、Open Server がアプリケーションの SRV_RPC イベント・ハンドラを呼び出すことなく直接認識して実行する、特殊な RPC です。「レジスタード・プロシージャ」(151 ページ) を参照してください。

RPC が登録されていない場合、Open Server は SRV_RPC イベントをトリガします。SRV_RPC イベント・ハンドラ内から、アプリケーションは RPC の名前とパラメータ (ある場合) を取得し、適切に応答することができます。イベント・ハンドラは、クライアントから送られる可能性のあるすべての RPC の名前と、それぞれが持つパラメータの数を検証するようにコーディングされています。ハンドラには、個々の RPC に応答するためのコードが含まれており、RPC を認識できない場合は、クライアントにエラー情報を返します。

SRV_RPC イベント・ハンドラの中では、アプリケーションは次の手順を実行します。

1　RPC 名を取得するために、srv_rpcname を呼び出します (アプリケーションは、srv_rpcnumber、srv_rpcowner、srv_rpcdb を使用して、RPC 番号、所有者、関連データベースをそれぞれ取得することもできます)。該当する RPC が存在しない場合や、番号、所有者、またはデータベースの情報が無効である場合には、アプリケーションは srv_sendinfo によってエラー情報を返します。

2　srv_numparams を呼び出して、適切な数のパラメータが送信されたことを検証します。パラメータ情報に無効なものがあれば、srv_sendinfo によってエラー情報を返します。

3　srv_descfmt、srv_bind、srv_xferdata を呼び出して、パラメータを処理します。詳細については、「パラメータとロー・データの処理」(126 ページ) を参照してください。

4　srv_descfmt、srv_bind、srv_xferdata を呼び出して、データをクライアントに返します。詳細については、「パラメータとロー・データの処理」(126 ページ) を参照してください。

RPC パラメータは、名前または位置で受け渡されます。一部のパラメータが名前で、他のパラメータが位置で受け渡されるような状況で RPC が呼び出された場合には、エラーが発生します。

アプリケーションは、すべてのプロシージャを登録し、SRV_RPC イベント・ハンドラを使用してエラーをトラップすることもできます。この場合には、クライアントが無登録の、つまり無効な RPC を送った場合にのみ Open Server が SRV_RPC イベント・ハンドラを呼び出すことになります。これを受けて SRV_RPC イベント・ハンドラが srv_sendinfo を使って、クライアントに対して無効な RPC を発行したことを通知します。

例

サンプル・プログラム *regproc.c* には、リモート・プロシージャ・コールの例が記述されています。

セキュリティ・サービス

セキュリティ・サービスを利用すると、Open Server アプリケーションで、サード・パーティの分散セキュリティを使用してユーザを認証し、クライアントとサーバとの間で転送されるデータを保護することができます。

使用しているプラットフォームで使用可能な分散セキュリティ・サービス・プロバイダについては、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。

特定のプロバイダから入手できるセキュリティ・サービスを、「セキュリティ・メカニズム」と呼びます。Open Server アプリケーションは、入手可能かどうかによって複数の「セキュリティ・メカニズム」をサポートできます。Open Server アプリケーションは、クライアントまたはサーバ・ダイアログごとに (クライアント接続要求に基づいて) セキュリティ・メカニズムを選択します。

Open Server のセキュリティ・サービスを使用して、次のタスクを実行できます。

- システム上に確立されている「クレデンシャル」にアクセスできます。

  クレデンシャルとは、ピアの ID を確立するためにピア (クライアントとサーバ) 間で転送されるデータです。

- ダイアログの確立中に要求されたセキュリティ・メカニズムと通信できます。
- リモート・クライアントまたはリモート・サーバとのセキュリティ・セッションを確立できます。

  セキュリティ・サービスは、セキュリティ・セッションの確立中にネゴシエートされます。セキュリティ・セッションは、直接クライアント・ダイアログにマップされます。

- 内部が隠されたトークンをダイアログ上で通信することで、ピア・コンポーネントとの通信でセキュリティ・メカニズムを使用できます。これらのトークンはセッションの確立中に送信され、必要に応じて、パケットごとのセキュリティ・サービスに使用できます。

  トークンとは、ピア間でのセキュリティ情報交換のためにセキュリティ・メカニズムが生成するビット文字列です。トークンは、暗号化されて保護される場合もあります。
- セキュリティ・セッションにチャネル識別情報をバインドできます。
- トークンの発生元を保証するために、トークンにデジタル化処理できます。

セキュリティ・サービス・プロパティ

ネットワーク・セキュリティ・サービスは、大きく分けて次の 3 つのタイプに分類できます。

- ログイン認証サービス
- パケットごとのセキュリティ・サービス
- SSL (Secure Sockets Layer) 暗号化

ログイン認証サービス

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、ユーザ名とその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 はその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認証ユーザを示していることを確認します。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。その代わりに、ユーザはネットワーク・セキュリティ・システムに対してユーザ自身を認証した後で、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。

次の表は、ログイン認証に関係のあるプロパティを示します。

表 *2-35:* ログイン認証を制御するプロパティ

プロパティ	説明
CS_USERNAME	接続時に使用するユーザ名を指定する。
CS_SEC_NETWORKAUTH	ネットワークベースのユーザ認証を有効にする。
CS_SEC_CREDTIMEOUT	ユーザのクレデンシャルの有効期限が切れているかどうかを知らせる。
CS_SEC_SESSTIMEOUT	クライアントとサーバとの間のセッションの有効期限が切れているかどうかを知らせる。
CS_SEC_MUTUALAUTH	クライアント・アプリケーションはこのプロパティを設定して、サーバがそのサーバ自体であることをクライアントに対して認証することを要求する。
CS_SEC_DELEGATION	クライアント・アプリケーションはこのプロパティを設定して、ゲートウェイ・サーバがクライアントの委任クレデンシャル・トークンを使用してリモート・サーバに接続できるようにする。
CS_SEC_CREDENTIALS	ゲートウェイ・アプリケーションはこのプロパティを使用して、ゲートウェイのクライアントからリモート・サーバに委任クレデンシャル・トークンを転送する。

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。

ネットワーク認証は、すべてのセキュリティ・メカニズムでサポートされています。クレデンシャルとセッションのタイムアウトは、いくつかのセキュリティ・メカニズムではサポートされていますが、すべてのセキュリティ・メカニズムでサポートされているわけではありません。どのサービスがどのセキュリティ・メカニズムでサポートされているかについては、『Open Client/Server 設定ガイド』を参照してください。

『Open Client Client-Library/C リファレンス・マニュアル』も参照してください。

パケットごとのセキュリティ・サービス

環境によっては、分散アプリケーションが物理的に安全でないネットワークに対処しなければならない場合があります。たとえば権限のないグループが、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。

これらの環境では、アプリケーションの保護と転送データの認証を使用して、安全なダイアログを保証してください。

次の表は、さまざまなパケットごとのサービスの使用方法を制御するプロパティを示します。

表 *2-36:* データ認証プロパティ

プロパティ	説明
CS_SEC_ CONFIDENTIALITY	データ機密保持サービスを有効にする。 データ機密保持サービスはすべての転送データを暗号化して、不明なユーザが転送データを理解できないことを保証する。
CS_SEC_ INTEGRITY	データ整合性サービスを有効にする。 データ整合性サービスは、転送データを不正に変更しようとする動作が検出されることを保証する。
CS_SEC_ DATAORIGIN	データ・オリジン・スタンピング・サービスを有効にする。 データ・オリジン・スタンピング・サービスは、受信データが、そのクライアントまたはサーバから実際に送信されたことを保証する。
CS_SEC_ DETECTREPLAY	リプレイ検出サービスを有効にする。 リプレイ検出サービスは、受信した転送データを不明なユーザがリプレイ (再生) しようとする動作が検出されることを保証する。
CS_SEC_ DETECTSEQ	順序検証サービスを有効にする。 順序検証サービスは、送信された順序とは異なる順序で到達した転送を検出する。
CS_SEC_ CHANBIND	チャネル・バインド・サービスを有効にする。 チャネル・バインド・サービスは、クライアントのアドレスとサーバのアドレスの暗号化記述を使用して、それぞれの転送にマークを付ける。

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もアプリケーションのセキュリティを重視する場合以外は、データ認証サービスを使用しないでください。

パケットごとのサービスすべてで、1 つの接続で送信される TDS パケットごとに、次に示す処理のどちらかまたは両方が行われます。

- パケットの内容の暗号化
- その他の必要な情報だけでなくパケット内容もコード化する、デジタル署名の計算

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。

『Open Client Client-Library/C リファレンス・マニュアル』を参照。

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の両者が 1 つの CipherSuite について合意すると、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 プロトコルの詳細については、Internet Engineering Task Force Web サイト (http://www.ietf.org) を参照してください。

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 SECHMECH (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 ether hostname, port ssl
master tcp ether hostname, port 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, port, ssl
master=TCP,hostname, port, ssl
```

hostname はクライアントが接続しているサーバの名前、*port* はホスト・マシンのポート番号です。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。SSL 接続を受け付け、別の接続では暗号化されないプレーン・テキストを受け付けるようにサーバを設定することも、他のセキュリティ・メカニズムを使用するように設定することもできます。

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 ether hostname 2748 ssl
query tcp ether hostname 2748 ssl
master tcp ether hostname 2749
```

この例では、SSL セキュリティ・サービスがポート番号 2748 で定義されています。SYBSRV1 では、Adaptive Server Enterprise はクリア・テキストをポート番号 2749 で受信します。このポートには、セキュリティ・メカニズムやセキュリティ・フィルタは定義されていません。

証明書によるサーバの検証

Open Client/Open Server が SSL 対応サーバに接続する場合は、サーバは証明書ファイルが必要です。これには、サーバの証明書と暗号化プライベート・キーが含まれています。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」(166 ページ) を参照してください。
- SSL-対応サーバへの接続では、デフォルトの動作としてサーバ証明書の共通名と interfaces ファイルのサーバ名が比較されます。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。SDC 環境での共通名の検証については、「SDC 環境での共通名の検証」(165 ページ) を参照してください。

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 － *$SYBASE/$SYBASE_ASE/certificates/servername.crt*

Windows － *%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 環境での共通名の検証

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。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。

クライアントはトランスポート・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ため、Adaptive Server Enterprise の SSL 証明書の共通名がサーバ名やクラスタ名と異なっていてもかまいません。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。

UNIX での構文

UNIX での SSL 対応 Adaptive Server Enterprise およびクラスタのサーバ・エントリの構文を次に示します。

```
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"
query tcp ether hostname4 5000 ssl="CN=name4"

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"

ASESSL4
master tcp ether hostname1 5000 ssl="CN=name4"
query tcp ether hostname1 5000 ssl="CN=name4"
```

Windows での構文

Windows での SSL 対応 Adaptive Server Enterprise およびクラスタのサーバ・エントリの構文を次に示します。

```
[CLUSTERSSL]
query=tcp,hostname1,5000, ssl="CN=name1"
query=tcp,hostname2,5000, ssl="CN=name2"
query=tcp,hostname3,5000, ssl="CN=name3"
query=tcp,hostname4,5000, ssl="CN=name4"

[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"

[ASESSL4]
master=tcp,hostname4,5000, ssl="CN=name4"
query=tcp,hostname4,5000, ssl="CN=name4"
```

信頼されたルート・ファイル

既知で信頼された認証局のリストは、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) に既知の認証局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 – *$SYBASE/$SYBASE_OCS/config/trusted.txt*

Windows – *%SYBASE%¥%SYBASE_OCS%¥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);
  ```

  $SYBASE はインストール・ディレクトリです。ct_config() を使ってコンテキスト・レベルに、または ct_con_props() を使って接続レベルに 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);
  ```

  $SYBASE はインストール・ディレクトリです。

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。

SSL およびパブリック・キー暗号法については、『Open Client Client-Library/C リファレンス・マニュアル』を参照してください。

Open Server でのセキュリティ・サービスの機能

セキュリティ・サービスを開始するには、クライアントはダイアログを確立するときに、セキュリティ・メカニズムにマップされる「オブジェクト識別子」をサーバに送信します。サーバは、そのオブジェクト識別子をセキュリティ・メカニズムのローカル名にマップします。要求されたセキュリティ・メカニズムをサーバがサポートしていない場合や、セキュリティ・セッションをまったくサポートしていない場合、ダイアログの要求は失敗して、Open Server はエラーを返します。

オブジェクト識別子を使用すると、セキュリティ・メカニズムのローカル名はクライアントとサーバで異なっていてもかまいません。この場合、システム管理者とアプリケーション・プログラマは、セキュリティ・メカニズムに対して独自の別々のローカル命名規則を開発できます。「オブジェクト識別子」(170 ページ) を参照してください。

Server-Library を使用すると、クレデンシャルを取得するときに使用されるプリンシパル名を指定できます。この「プリンシパル名」は、セキュリティ・サービス・プロバイダが Open Server アプリケーションを認識している名前です。アプリケーションのプリンシパル名を設定するには、SRV_S_SEC_PRINCIPAL サーバ・プロパティを指定して srv_props 関数を実行します。

プリンシパル名が設定されていない場合のデフォルト値は Open Server アプリケーションのネットワーク名ですが、このネットワーク名は一般に、srv_init で指定されます。

Open Server はクライアントとのセキュリティ・セッションを確立するときに、クレデンシャルを使用します。

クライアントのログイン名はセキュリティ・セッションから得られます。したがって、ログイン・レコード内にどのような名前が指定されていても無視されます。

セキュリティ・サービスを使用する場合のクライアントの役割については、『Open Client Client-Library/C リファレンス・マニュアル』を参照してください。

セキュリティ・サービスを使用する Client/Server ダイアログで必要な手順

クライアントがセキュリティ・サービスを使用するダイアログを開始する場合、Open Server は次の手順に従います。

1 クライアントとのトランスポート接続を確立します。

2 クライアントのログイン・レコードと内部が隠されたセキュリティ・トークンを受信して、内部が隠された必要なトークンでクライアントに応答します。

3 セキュリティ・メッセージ・ハンドシェイクが成功したときに、セキュリティ・セッションを確立します。

Open Server アプリケーションがクライアントから情報を受信した場合は、次の手順に従います。

1 クライアントから受信した応答に関連付けられているセキュリティ・メッセージ (暗号化シグニチャなど) がある場合は、処理します (暗号化シグニチャによって、メッセージの整合性が確保されます)。

2 セキュリティ・セッションでサポートされているセキュリティ・サービスに基づいて、適切なルーチン (シグニチャを検証するルーチンなど) を呼び出します。

3 TDS (Tabular Data Stream) に対し、通常の処理を実行します。

Open Server は、次の手順に従ってクライアントに応答を送信します。

1 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が切れていないかどうかをチェックします。有効期限切れが検出された場合、Open Server はエラーの処理を実行します。

2 このダイアログでサポートされているセキュリティ・サービスに基づいて、適切なルーチン (応答に対して暗号化シグニチャを生成するルーチンなど) を呼び出します。

3 要求された TDS を生成して、パケットごとのセキュリティ・サービスを識別します。

セキュリティ・セッションは、対応するクライアント・ダイアログが終了したときに終了します。セキュリティ・セッションは通常のクライアント・ログアウトで終了する場合もあれば、エラー条件が発生したために終了する場合もあります。

Open Server アプリケーションでのセキュリティ・メカニズムの使用

この項では、Open Server アプリケーションでサード・パーティのセキュリティ・メカニズムを使用するために行う必要がある変更について説明します。これらの変更には、次のようなエントリを追加する作業も含まれます。

- *libtcl.cfg*　ファイル内で、セキュリティ・メカニズムからドライバへの各マッピングのエントリを追加する作業。
- グローバル・オブジェクト識別ファイル *objectid.dat* 内で、グローバルにユニークなオブジェクト識別子に各セキュリティ・メカニズムのローカル名をマップするエントリを追加する作業。
- サード・パーティのセキュリティ・メカニズムを使用する各サーバの interfaces ファイル内で、サーバがサポートしているすべてのセキュリティ・メカニズムを指定するエントリを追加する作業。

セキュリティ・ドライバ

Sybase は「セキュリティ・ドライバ」を提供しています。このセキュリティ・ドライバを使用すると、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。Client-Library と Server-Library には、安全化されているアプリケーションを実装するための汎用インタフェースが備わっています。この汎用インタフェースは、それぞれの Sybase セキュリティ・ドライバによってセキュリティ・プロバイダのインタフェースにマップされます。

セキュリティ・ドライバは動的にロードでき、1 つまたは複数のセキュリティ・メカニズムをサポートしています。

現在サポートされている各セキュリティ・プロバイダのドライバは、次のとおりです。

- *libsybsdce*

  DCE セキュリティ・サービスの場合
- *libsybsmssp*

  Microsoft NT SSPI の場合

libtcl.cfg 設定ファイル

libtcl.cfg 設定ファイルは、セキュリティ・メカニズムのローカル名を、そのメカニズムをサポートするのに必要なセキュリティ・ドライバにマップします。*libtcl.cfg* ファイルは、*$SYBASE/$SYBASE_OCS/config* ディレクトリ、またはコンテキスト・プロパティ CS_LIBTCL_CFG により指定されるパスにあります。このファイルの正確なロケーションについては、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。

libtcl.cfg ファイル内には、それぞれのセキュリティ・ドライバのエントリが指定されている必要があります。それぞれのドライバは 1 つのセキュリティ・メカニズムをサポートしている場合もあれば、複数のセキュリティ・メカニズムをサポートしている場合もあります。ドライバが複数のセキュリティ・メカニズムをサポートしている場合は、*libtcl.cfg* ファイル内にそれぞれのセキュリティ・メカニズムのエントリが指定されている必要があります。

このファイルのフォーマットは次のとおりです。

```
[SECURITY]
 local-name-of-security-mechanism = path-to-the-driver init-string
```

各パラメータの意味は、次のとおりです。

- *path-to-the-driver* － オブジェクト・ファイルへの完全に修飾されたパス名です。
- *init-string* － ドライバに応じて異なる引数リストです。一般的な形式は、*token* = *value*, *token* = *value*, ... です。

以下は UNIX プラットフォームで指定する場合の例です。

```
[SECURITY]
 csfkrb5=libsybskrb.so secbase=@MYREALM libgss=/krb5/lib/libgss.so
```

libtcl.cfg ファイル内の最初のエントリは、デフォルトのセキュリティ・メカニズムです。アプリケーションがセキュリティ・サービスを要求する場合に、Open Server はデフォルトのセキュリティ・メカニズムを使用しますが、セキュリティ・メカニズムの設定は行いません。

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。

オブジェクト識別子

それぞれのセキュリティ・メカニズムには、対応するオブジェクト識別子があります。グローバルにユニークなオブジェクト識別子は、グローバル・オブジェクト識別ファイル *Objectid.dat* 内で、1 つのセキュリティ・メカニズムのローカル名にマップされます。これによって、柔軟で一貫性のある方法で、クライアントとサーバとの間でセキュリティ・メカニズム名を通信できます。*Objectid.dat* ファイルは、*$SYBASE/config* ディレクトリにあります。

グローバル識別ファイルのフォーマットは次のとおりです。

```
[Object Class]
      Object_Identifier Object_Name_List
```

セキュリティ・メカニズムのエントリは次のようになります。

Object Class － “secmech” です。

Object_Identifier － ドットで区切られた負でない一連の整数値です。オブジェクト識別子は、国際的な標準化組織である CCITT と ISO で定義された用語規定に基づいています。DCE セキュリティ・ドライバの *sybase* ルートからのオブジェクト識別子の例は、897.4.6.1 です。

Object_Name_List － ローカル・セキュリティ・メカニズム名をカンマで区切ったリストです。

次に例を示します。

```
[secmech]
        1.3.6.1.4.1.897.4.6.3 = NTLM
```

interfaces ファイルへの変更

interfaces ファイルのフォーマットは、サーバがサポートしているセキュリティ・メカニズムを指定できるように拡張されています。フォーマットは次のとおりです。

```
SERVERNAME
      query tcp sun-ether joyce 2901
      master tcp sun-ether joyce 2901
      secmech mechanism1, mechanism2,..., mechanismN
```

secmech 識別子は、サーバがサポートしているすべてのセキュリティ・メカニズムのリストです。また、次の条件が適用されます。

- この行はオプションであり、サーバが Sybase 独自のセキュリティ・メカニズムを使用していない場合にだけ使用されます。
- interfaces ファイル内にサーバの secmech エントリがない場合、そのサーバは *libtcl.cfg* ファイルの secmech エントリで指定されているすべてのセキュリティ・メカニズムをサポートしています。
- interfaces ファイル内にサーバの secmech エントリがあるが、セキュリティ・メカニズムが指定されていない場合、そのサーバはどのようなセキュリティ・メカニズムもサポートしていません。

mechanism1, mechanism2,...mechanismN は、サーバがサポートしているセキュリティ・メカニズムのオブジェクト識別子です。カンマをセパレータとして使用して複数のセキュリティ・メカニズムを指定できます。「オブジェクト識別子」(170 ページ) を参照してください。

interfaces ファイルへの変更：SSL フィルタ

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 ether hostname port ssl
master tcp ether hostname port 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, port, ssl
master=TCP,hostname,port, ssl
```

hostname はクライアントが接続しているサーバの名前、*port* はホスト・マシンのポート番号です。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。SSL 接続を受け付け、別の接続では暗号化されないプレーン・テキストを受け付けるようにサーバを設定することも、他のセキュリティ・メカニズムを使用するように設定することもできます。

アクティブなセキュリティ・サービスの決定

クライアント・サーバ間ダイアログでどのセキュリティ・サービスがアクティブであるかを調べるには、srv_thread_props を使用して、次のスレッド・プロパティの値を取得します。

- SRV_T_SEC_CHANBIND
- SRV_T_SEC_CONFIDENTIALITY
- SRV_T_SEC_DATAORIGIN
- SRV_T_SEC_DELEGATION
- SRV_T_SEC_DETECTREPLAY
- SRV_T_SEC_DETECTSEQ
- SRV_T_SEC_INTEGRITY
- SRV_T_SEC_MUTUALAUTH
- SRV_T_SEC_NETWORKAUTH

各スレッド・プロパティの説明については、表 2-28 (139 ページ) を参照してください。

Open Server アプリケーションでのセキュリティ・サービスの使用例

この項では、さまざまな Open Server アプリケーション設定でセキュリティ・サービスを使用する方法について説明します。具体的には、次の場合について説明します。

- セキュリティ・セッションを使用する単純な Open Server アプリケーション
- 個別のセキュリティ・セッションを使用するゲートウェイ Open Server アプリケーション
- 委任を使用して個別のセキュリティ・セッションを使用するゲートウェイ Open Server アプリケーション
- ダイレクト・セキュリティ・セッションを使用するフル・パススルー・ゲートウェイ Open Server アプリケーション

セキュリティ・セッションを使用する単純なアプリケーション

最も単純な設定では、クライアントはセキュリティ・メカニズムによって提供される認証サービスを使用して、ダイアログを確立します。Open Server はログイン・ネゴシエーションを実行してから、接続ハンドラを呼び出します。接続ハンドラによって srv_senddone(SRV_DONE_FINAL) が発行されると、Open Server はステータスが “success” のログイン確認をクライアントに送信します。

この設定では、接続ハンドラをインストールする必要はありません。デフォルトの接続ハンドラで十分です。それでも接続ハンドラをインストールする場合は、次の例で示すように少なくとも srv_senddone(SRV_DONE_FINAL) だけは送信してください。

```
CS_RETCODE CS_PUBLIC connect_handler(spp)
 SRV_PROC *spp;
 {

   .......

   /*
    ** You do not need to test this srv_senddone's return value
    ** since Open Server will kill this thread if this call fails.
    */
    (CS_VOID)srv_senddone(spp, SRV_DONE_FINAL, CS_TRAN_UNDEFINED,
                    (CS_INT)0);
    return(CS_SUCCEED);
 }
```

個別のセキュリティ・セッションを使用するゲートウェイ・アプリケーション

以下に示す例では、Open Server アプリケーションはクライアントと別のサーバの間のゲートウェイとして動作します。クライアントとゲートウェイ・アプリケーションの間のセキュリティ・セッションを確立するのに使用されるネットワーク ID は、ゲートウェイとリモート・サーバの間のセキュリティ・セッションを確立するのに使用されるネットワーク ID とは異なる場合があります。

ゲートウェイ・アプリケーションは最後のログイン確認を保留して、そのクライアントとのログイン・セキュリティ・ネゴシエーションを完了してから、接続ハンドラを呼び出します。接続ハンドラは Client-Library 呼び出しを使用して、リモート・サーバへのセキュリティ・セッション・ベースのログインを開始してから、srv_senddone(SRV_DONE_FINAL) をクライアントに送信してそのログインを完了する必要があります。次は、接続ハンドラの例です。

```
CS_RETCODE CS_PUBLIC connect_handler(spp)
 SRV_PROC *spp;
 {
    CS_CONNECTION   *conn;    /* the connection handle */
    CS_BOOL         trueval = CS_TRUE;
    CS_INT          outlen;

  ......

  allocate and set user data in spp...

  ......

  /* Allocate a connection handle */
    if (ct_con_alloc(Context, &(userdata->conn)) == CS_FAIL)
    {
       clean up and report error...
      return(CS_FAIL);

   }

   ......

   conn = userdata->conn;
    /*
    ** Initiate security session based login with the remote
    ** server.The user name used here may be the same as the
    ** client user name or different
    */
    if (ct_con_props(conn, CS_SET, CS_USERNAME,
        (CS_VOID*)Username, STRLEN(Username), (CS_INT*)NULL)
        == CS_FAIL)
    {
       handle failure...
   }

   /*
    ** Set the desired security mechanism(s) or use the default
    ** security mechanism.
    */
```

```
    if (ct_con_props(conn, CS_SET, CS_SEC_MECHANISM,
           (CS_VOID*)Mechanismname, STRLEN(Mechanismname),
           (CS_INT*)NULL) == CS_FAIL)
    {
       handle failure...
    }

    /* Set the security service-network authentication */
    if (ct_con_props(conn, CS_SET, CS_SEC_NETWORKAUTH,
           (CS_VOID*)&trueval, CS_SIZEOF(CS_BOOL), (CS_INT*)NULL)
           == CS_FAIL)
    {
       handle failure...
    }

    set other security services if required
    get and set the user's application name, response capabilities
    set the locale and other login properties
    /* Attempt a connection to the remote server */
    if (ct_connect(conn, Servername, CS_NULLTERM) == CS_FAIL)
    {
       cleanup...
       return(CS_FAIL);
    }

    get and set the REQUEST capabilities
    get and set the RESPONSE capabilities
    ......
    /*
    ** You do not need to test this srv_senddone's return value
    ** since Open Server will kill this thread if this call fails.
    */
    (CS_VOID)srv_senddone(spp, SRV_DONE_FINAL, CS_TRAN_UNDEFINED,
                  (CS_INT)0);
    return(CS_SUCCEED);
}
```

委任を使用して個別のセキュリティ・セッションを使用するゲートウェイ

Open Server アプリケーションはクライアントと別のサーバの間のゲートウェイとしても動作しますが、ゲートウェイ・アプリケーションはリモート・サーバとのセキュリティ・セッションを確立するときに、委任クライアント・クレデンシャルを使用します。クライアントは、そのクライアント自体のクレデンシャルだけを委任できます。

一度セキュリティ・セッションが確立されると Open Server アプリケーションが委任クレデンシャルを得ることができるように、クライアントは CS_SEC_DELEGATION サービスを要求する必要があります。

クライアントとゲートウェイ Open Server アプリケーションとの間のセキュリティ・セッションは、最後のログイン確認を除いて、「セキュリティ・セッションを使用する単純なアプリケーション」(173 ページ) と同様に確立されます。

接続ハンドラ内では、ゲートウェイ・アプリケーションは次の手順に従います。

1 srv_thread_props(CS_GET, SRV_T_SEC_DELEGCRED) を使用して、委任クレデンシャルを取得します。

2 ct_con_props(CS_SET, CS_SEC_CREDENTIALS) を使用して、リモート・サーバへの接続に使用するための委任クレデンシャルを Client-Library 接続構造体内に設定します。

3 ct_connect ルーチンを使用して、リモート・サーバに接続しようとします。

4 srv_senddone(SRV_DONE_FINAL) を送信して、クライアントのログインを確認します。

次は、接続ハンドラの例です。

```
CS_RETCODE CS_PUBLIC connect_handler(spp)
 SRV_PROC *spp;
 {
    CS_CONNECTION *conn;    /* Connection handle */
    CS_VOID       *creds;     /* security credentials */
    CS_BOOL       trueval = CS_TRUE;
    CS_BOOL       boolval;
    CS_CHAR       mechanismname[MAX_NAMESIZE];
    CS_CHAR       username[MAX_NAMESIZE];
    CS_INT        outlen;
   ......
   allocate and set user data in spp
   ......
   /* Allocate a connection handle for the connection attempt.*/
    if (ct_con_alloc(Context, &(userdata->conn)) == CS_FAIL)
    {
       return(CS_FAIL);
    }
   ......
   conn = userdata->conn;
   /*
    ** Initiate security session based login to the target server
    */
   /* Retrieve the client user name */
    if (srv_thread_props(spp, CS_GET, SRV_T_USER,
         (CS_VOID *)username, MAX_NAMESIZE, &outlen) == CS_FAIL)
    {
       handle failure...
   }
   /*
    ** Set the client's security principal name to connect to the
    ** target server
    */
    if (ct_con_props(conn, CS_SET, CS_USERNAME,
       (CS_VOID *)username, outlen, (CS_INT *)NULL) == CS_FAIL)
    {
       handle failure...
```

```
}
/* Retrieve and set the security mechanism */
 if (srv_thread_props(spp, CS_GET, SRV_T_SEC_MECHANISM,
      (CS_VOID *)mechanismname, MAX_NAMESIZE, &outlen)
      == CS_FAIL)
 {
    handle failure...
}
if (ct_con_props(conn, CS_SET, CS_SEC_MECHANISM,
     (CS_VOID *)mechanismname, outlen, (CS_INT *)NULL)
      == CS_FAIL)
 {
    handle failure...
}
/*
 ** Set security service-network authentication.Alternatively
 ** retrieve services from the current thread and set it.
 */
 if (ct_con_props(conn, CS_SET, CS_SEC_NETWORKAUTH,
    (CS_VOID *)&trueval, CS_SIZEOF(CS_BOOL), (CS_INT *)NULL)
    == CS_FAIL)
 {
    handle failure...
}
set other security services if needed...
/* Ensure that the client enabled security delegation */
 if (srv_thread_props(spp, CS_GET, SRV_T_SEC_DELEGATION,
    (CS_VOID *)&boolval, CS_SIZEOF(CS_BOOL), (CS_INT *)NULL)
    == CS_FAIL)
 {
     handle failure...
 }
if (boolval != CS_TRUE)
 {
    /* delegation not handled on this dialog */
   handle failure...
}
/* Retrieve the delegated credentials */
 if (srv_thread_props(spp, CS_GET, SRV_T_SEC_DELEGCRED,
    (CS_VOID *)&creds, CS_SIZEOF(CS_VOID*), (CS_INT *)NULL)
    == CS_FAIL)
 {
     handle failure...
 }
/*
 ** Set the delegated credentials to authenticate to the target
 ** server.
 */
 if (ct_con_props(conn, CS_SET, CS_SEC_CREDENTIALS,
    (CS_VOID *)&creds, CS_SIZEOF(CS_VOID *), (CS_INT *)NULL)
```

```
      == CS_FAIL)
    {
       handle failure...
   }
   get and set the user's application name and response
    capabilities...
   set the locale and other properties...
   /* Attempt a connection to the remote server */
    if (ct_connect(conn, Servername, CS_NULLTERM) == CS_FAIL)
    {
       handle failure...
   }
   Get and set the REQUEST capabilities...
   Get and set the RESPONSE capabilities...
   ......
   /*
    ** You do not need to test this srv_senddone's return value
    ** since Open Server will kill this thread if this call fails.
    */
    (CS_VOID)srv_senddone(spp, SRV_DONE_FINAL, CS_TRAN_UNDEFINED,
                  (CS_INT)0);
    return(CS_SUCCEED);
}
```

ダイレクト・セキュリティ・セッションでのフル・パススルー・ゲートウェイ

クライアントがセキュリティ・セッションを確立するには、リモート・サーバが必要です。クライアントとリモート・サーバの間のどのような中間サーバでも、パケットごとのセキュリティ・サービスは実行されません。クライアントが機密保持を要求する場合、ゲートウェイはメッセージ・パケットから TDS トークンを取得できません。この設定では、受信パケットを復号化するのに使用されるサービスや、転送前に受信パケットを再暗号化するのに使用されるサービスなどのパケットごとのサービスはゲートウェイ内では実行されないので、オーバヘッドが減ります。

「転送サーバ」の連鎖を形成する複数の中間ゲートウェイがある場合もあります。この場合、各転送サーバで同じセキュリティ・メカニズムがサポートされている必要があります。

ダイレクト・セキュリティ・セッションを設定するには、Open Server ゲートウェイ・アプリケーションの接続ハンドラで次の手順に従います。

1 srv_getloginfo ルーチンを使用して、クライアント・スレッドからログイン情報を得ます。

2 ct_setloginfo ルーチンを使用して、リモート・サーバへの接続に使用される接続構造体内に、この情報を設定します。

3 次のコマンドを使用して、セキュリティ・セッション・コールバックをインストールします。

```
ct_callback(conn, CS_SET, CS_SECSESSION_CB, secsession_cb)
```

リモート・サーバへの接続が確立されると、コールバックは、リモート・サーバとゲートウェイのクライアントとの間で必要なハンドシェイクの仲介者として動作します。

コールバックの内容については、「セキュリティ・セッション・コールバック」(181 ページ) を参照してください。

コールバックの詳細については、『Open Client Client-Library/C リファレンス・マニュアル』を参照してください。

4 ct_connect ルーチンを呼び出して、リモート・サーバに接続します。この呼び出しはクライアントとリモート・サーバの間のネゴシエーションを開始して、セキュリティ・セッションを確立します。ct_connect ルーチンが CS_SUCCEED を返した場合は、セキュリティ・セッションの確立が成功したことを示します。

5 srv_senddone(SRV_DONE_FINAL) を使用して、ログインが完了したことをクライアントに通知します。

接続ハンドラの例

```
CS_RETCODE CS_PUBLIC connect_handler(spp)
 SRV_PROC *spp;
 {
    CS_CONNECTION *conn;    /* connection handle */
    CS_VOID       *creds;   /* security credentials */
    CS_LOGINFO    *loginfo; /* login information */
    CS_BOOL       boolval;
   ......
   allocate and set user data in spp
   /* Allocate a connection handle for the connection attempt. */
    if (ct_con_alloc(Context, &(userdata->conn)) == CS_FAIL)
    {
       handle failure...
   }
   ......
   conn = userdata->conn;
   /*
    ** Save the pointer to thread control structure in the
    ** connection handle
    */
    if (ct_con_props(conn, CS_SET, CS_USERDATA, &spp,
       CS_SIZEOF(spp), (CS_INT *)NULL) == CS_FAIL)
    {
       handle failure...
   }
   /* Verify that security based login is requested */
    if (srv_thread_props(spp, CS_GET, SRV_T_SEC_NETWORKAUTH,
       (CS_VOID *)&boolval, CS_SIZEOF(CS_BOOL), (CS_INT *)NULL)
       == CS_FAIL)
    {
```

```
      handle failure...
   }
   if (boolval != CS_TRUE)
    {
      handle the client request that does not use security
      session based login
     ......
     return(CS_SUCCEED);
    }
 /* Get and set the login information */
   if (srv_getloginfo(spp, &loginfo) == CS_FAIL)
    {
      handle failure...
   }
   if (ct_setloginfo(conn, loginfo) == CS_FAIL)
    {
      handle failure...
   }
   /* Install a security session callback for this connection */
    if (ct_callback((CS_CONTEXT *)NULL, conn, CS_SET,
      CS_SECSESSION_CB, (CS_VOID *)secsession_cb) == CS_FAIL)
    {
      handle failure...
   }
   /* Attempt a connection to the remote server */
    if (ct_connect(conn, Servername, CS_NULLTERM) == CS_FAIL)
    {
      handle failure...
   }
   /* Get and set the login information */
    if (ct_getloginfo(conn, &loginfo) == CS_FAIL)
    {
      handle failure...
   }
   if (srv_setloginfo(spp, loginfo) == CS_FAIL)
    {
      handle failure...
   }
   ......
   /*
    ** You do not need to test this srv_senddone's return value
    ** since Open Server will kill this thread if this call fails.
    */
    (CS_VOID)srv_senddone(spp, SRV_DONE_FINAL, CS_TRAN_UNDEFINED,
                  (CS_INT)0);
    return(CS_SUCCEED);
}
```

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・ルーチンは、ターゲット・サーバ (またはそのゲートウェイの次の中間ゲートウェイ) とゲートウェイのクライアント・アプリケーションとの間でセキュリティ・トークンを交換して、クライアントとリモート・サーバとの間のダイレクト・セキュリティ・セッションを確立します。このコールバック手順は、異なるパラメータを使用することを除けば、チャレンジ／応答コールバックと似ています。

ゲートウェイが ct_connect ルーチンを呼び出すと、リモート・サーバはセキュリティ・セッション情報が入っている 1 つまたは複数のメッセージを発行します。それぞれのセキュリティ・メッセージに対して、Client-Library はリモート・サーバから送信されたメッセージ・パラメータを使用して、コールバックを開始します。

コールバック・ルーチンは次の手順に従う必要があります。

1　リモート・サーバのメッセージからパラメータを取得します。

2　次のルーチンを使用して、クライアントにパラメータを送信します。

- srv_negotiate(..., CS_SET, SRV_NEG_SECSESSION)
- srv_descfmt(..., CS_SET, SRV_NEGDATA, ...)
- srv_bind(..., CS_SET, ...)
- srv_xferdata(..., CS_SET, ...)

3　クライアントに srv_senddone(SRV_DONE_FINAL) を送信します。

4　srv_negotiate(CS_GET, SRV_NEG_SECSESSION) を使用して、クライアントからの応答を待ちます。

5　クライアントが応答すると、コールバック・ルーチンは、次の関数を使用して、対応するセッション・データをクライアントから出力バッファにコピーし、リモート・サーバに送信します。

- srv_descfmt(CS_GET)
- srv_bind(CS_GET)
- srv_xferdata(CS_GET)

6　リモート・サーバがまた別のセキュリティ・メッセージを送信する場合は、この処理を繰り返します。

セキュリティ・セッション・コールバックの定義については、『Open Client Client-Library/C リファレンス・マニュアル』を参照してください。

Client-Library セキュリティ・セッション・コールバック・ルーチンの例

```
CS_RETCODE CS_PUBLIC secsession_cb(conn, innumparams, infmt,
    inbuf, outnumparams, outfmt, outbuf, outlen)
 CS_CONNECTION *conn;
 CS_INT        innumparams;
 CS_DATAFMT    *infmt;
 CS_BYTE       **inbuf;
 CS_INT        *outnumparams;
 CS_DATAFMT    *outfmt;
 CS_BYTE       **outbuf;
 CS_INT        *outlen;
 {
   SRV_PROC *spp; /* The SRVPROC structure associated with the
                   ** client connection */
    CS_INT i;

   /* Get the previously saved spp for the client */
    if (ct_con_props(conn, CS_GET, CS_USERDATA, &spp,
       CS_SIZEOF(spp), (CS_INT *)NULL) != CS_SUCCEED)
    {
       return(CS_FAIL);
    }
   /*
    ** Use srv_negotiate to tell the client to expect a security
    ** token
    */
    if (srv_negotiate(spp, CS_SET, SRV_NEG_SECSESSION)
       != CS_SUCCEED)
    {
       return(CS_FAIL);
    }

  /* Describe and send the security token */
    for (i = 0; i < innumparams; i++)
    {
       if (srv_descfmt(spp, CS_SET, SRV_NEGDATA, i + 1, &infmt[i]
          != CS_SUCCEED)
       {
          return(CS_FAIL);
       }

      if (srv_bind(spp, CS_SET, SRV_NEGDATA, i + 1, &infmt[i],
          inbuf[i], &(infmt[i]->maxlength), (CS_SMALLINT *)NULL)
          != CS_SUCCEED)
       {
          return(CS_FAIL);
       }

   }

   if (srv_xferdata(spp, CS_SET, SRV_NEGDATA) != CS_SUCCEED)
    {
       return(CS_FAIL);
```

```
    }

   /* Complete this portion of the exchange */
    if (srv_senddone(spp, SRV_DONE_FINAL, CS_TRAN_UNDEFINED, 0)
       != CS_SUCCEED)
    {
       return(CS_FAIL);
    }

   /* Wait until the client responds */
    if (srv_negotiate(spp, CS_GET, SRV_NEG_SECSESSION)
       != CS_SUCCEED)
    {
       return(CS_FAIL);
    }

   /* Get the number of parameters in the client's response */
    if (srv_numparams(spp, outnumparams) != CS_SUCCEED)
    {
       return(CS_FAIL);
    }

   /* Read in the client's response */
    for (i = 0; i < (*outnumparams); i++)
    {
       srv_bzero(&outfmt[i], sizeof(CS_DATAFMT));

      if (srv_descfmt(spp, CS_GET, SRV_NEGDATA, i + 1, &outfmt[i]
          != CS_SUCCEED)
       {
          return(CS_FAIL);
       }

      if (srv_bind(spp, CS_GET, SRV_NEGDATA, i + 1, &outfmt[i],
          outbuf[i], &outlen[i], (CS_SMALLINT *)NULL)
          != CS_SUCCEED)
       {
          return(CS_FAIL);
       }

   }

   if (srv_xferdata(spp, CS_GET, SRV_NEGDATA) != CS_SUCCEED)
    {
       return(CS_FAIL);
    }

   /* Return success */
    return(CS_SUCCEED);
}
```

text と image

text と image Adaptive Server Enterprise データ型には、大きな text または image の値が入ります。text データ型は、印刷可能文字が最大 2,147,483,647 バイト入ります。image データ型は、バイナリ・データが最大 2,147,483,647 バイト入ります。

text および image 値は、非常に大きくなる可能性があるため、実際にはデータベース・テーブルには保存されません。代わりに text 値または image 値へのポインタがテーブルに格納されます。このポインタを「テキスト・ポインタ」と呼びます。

競合するクライアント・アプリケーションが、他方が行ったデータベースへの変更を上書きしないように、各 text または image カラムに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。

text および image データの処理

クライアントは、*writetext* ストリームと呼ばれる無区別のデータ・ストリームの形で text および image データを送ります。パラメータが分かれてはいないので、Open Server アプリケーションでは、srv_descfmt、srv_bind、srv_xferdata などの受信パラメータ・データの処理に通常使用するルーチンを使用できません。そこで、text および image ルーチンの特殊なセットを使用する必要があります。

Open Server アプリケーションは、リターン・ローにどれだけのカラムが含まれているかによって、2 つの方法のどちらかを使って text または image データをクライアントに返すことができます。リターン・ローに含まれるカラムが 1 つだけで、そのカラムの内容が text または image データの場合は、無区別データ・ストリームとして扱われ、その処理は通常のものとは異なります。しかし、そのローに text または image カラム以外のカラムが含まれている場合には、text または image データは記述／バインド／転送の方式を使って処理されます。両方に、共通の手順があることに注意してください。

記述／バインド／転送の 3 つの呼び出しの詳細については、「パラメータとロー・データの処理」(126 ページ) を参照してください。

クライアントからのデータの取得

writetext ストリームは、SRV_BULK イベントをトリガします。クライアントから取得された text および image データはバルク・データとして扱われるので、Open Server アプリケーションは、入ってくる text および image データを、バルク・ハンドラの中から処理します。『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。

アプリケーションは、受信 text および image データを次の 2 段階で処理します。

1　srv_text_info ルーチンは、text データまたは image データの記述を取得し、その情報を CS_IODESC 構造体に格納します。この呼び出しは、さまざまな情報を返しますが、その中で最も重要なのが、データの総合の長さです。この長さに基づいて、アプリケーションはデータを一度に取得するかセクションに分けて取得するかを決定でき、またデータを保存するバッファの大きさも決めることができます。srv_text_info は、cmd 引数を CS_GET に設定して呼び出します。

2　srv_get_text ルーチンは、指定されたセクション・サイズでクライアントから実際にデータを取得し、指定されたバッファに保存します。

srv_get_text の呼び出しの前に、必ず srv_text_info を呼び出すことに注意してください。srv_get_text ルーチンは、すべての text がクライアントから読み込まれるまで、呼び出し続けなければなりません。

コマンドの抑制

text または image カラムの更新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 CS_SENDDATA_NOCMD 接続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。
CS_SENDDATA_NOCMD が設定されている場合、サーバの sp_mda プロシージャは ct_connect の実行時に呼び出されます。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。

クライアントへのデータの返送

アプリケーションは、ロー結果を返すことができるすべてのイベント・ハンドラから text または image データを返すことができます。アプリケーションは、ローのデータに含まれているカラムの数によって、いくつかの異なった手順で、出力される text または image データを処理します。カラムが 1 つしかなく、それが text または image カラムである場合には、アプリケーションは次の手順をとります。

1 srv_descfmt を使って、クライアントが text または image カラムを受け取るフォーマットを記述します。

2 cmd を CS_SET に設定して srv_text_info を呼び出して、text 全体の長さを設定します。

3 データをまとまりでクライアントに送信するために、srv_send_text を呼び出します。

text カラムと image カラムの他にカラムがある場合、アプリケーションは次の手順に従わなければなりません。

1 srv_descfmt を使って、クライアントが text または image カラムを受け取るフォーマットを記述します。これは各カラムにつき一度ずつ呼び出されます。

2 srv_bind を使って、Open Server アプリケーションが情報を保存するローカル・プログラム変数のフォーマットと位置を記述します。これは各カラムにつき一度ずつ呼び出さなければなりません。

3 srv_text_info を呼び出すことで、テキスト・ポインタとタイムスタンプ情報を提供します。cmd を CS_SET に設定して、各 text または image カラムにつき一度ずつ呼び出さなければなりません。

4 srv_xferdata を使用してデータを転送します。このルーチンは、ローの数と同じ回数だけ呼び出さなければなりません。

text および image カラムの部分更新の詳細については、「パラメータとロー・データの処理」(126 ページ) を参照してください。

例

サンプル・プログラム *ctos.c* には、text データと image データを処理するコードが記述されています。

データ型

Open Server は、広い範囲にわたるデータ型をサポートします。これらのデータ型は、CS-Library、Client-Library と共有されます。ほとんどの場合には、これらは Adaptive Server Enterprise データ型と直接対応します。

表 2-37 に、Open Server の型定義と、対応する型、対応する Adaptive Server Enterprise のデータ型を示します。各データ型に関する詳細情報は、表の後にあります。

このバージョンにおいては、下位互換性を保つために、2.0 Open Server データ型が含まれています。このバージョンでは、2.0 Server-Library ルーチンは 2.0 データ型を使用しなければなりません。表 2-37 は、今後の Open Server バージョンですべてのルーチンが使用しなければならない Open Server データ型をまとめたものです。

表 *2-37:* データ型の概要

型	Open Client と Open Server の型定数	説明	対応する Open Client/Server の型定義	対応する Adaptive Server Enterprise のデータ型
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	NONE
	CS_VARBINARY_TYPE	可変長バイナリ型	CS_VARBINARY	NONE
bit 型	CS_BIT_TYPE	ビット型	CS_BIT	boolean
character 型	CS_CHAR_TYPE	文字型	CS_CHAR	char、varchar
	CS_LONGCHAR_TYPE	長い文字型	CS_LONGCHAR	NONE
	CS_VARCHAR_TYPE	可変長文字型	CS_VARCHAR	NONE
	CS_UNICHAR_TYPE	可変長または固定長文字型	CS_UNICHAR	unichar、univarchar
XML 型	CS_XML_TYPE	可変長文字型	CS_XML	xml

型	Open Client と Open Server の型定数	説明	対応する Open Client/Server の型定義	対応する Adaptive Server Enterprise のデータ型
datetime 型	CS_DATE_TYPE	4 バイトの日付データ型	CS_DATE	date
	CS_TIME_TYPE	4 バイトの時刻データ型	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 バイトの整数型	CS_TINYINT	tinyint
	CS_SMALLINT_TYPE	2 バイトの整数型	CS_SMALLINT	smallint
	CS_INT_TYPE	4 バイトの整数型	CS_INT	int
	CS_BIGINT_TYPE	8 バイトの整数型	CS_BIGINT	bigint
	CS_USMALLINT_TYPE	2 バイトの符号なし整数型	CS_USMALLINT	usmallint
	CS_UINT_TYPE	4 バイトの符号なし整数型	CS_UINT	uint
	CS_UBIGINT_TYPE	8 バイトの符号なし整数型	CS_UBIGINT	ubigint
	CS_DECIMAL_TYPE	10 進数型	CS_DECIMAL	decimal
	CS_NUMERIC_TYPE	数値型	CS_NUMERIC	numeric
	CS_FLOAT_TYPE	8 バイトの浮動小数点型	CS_FLOAT	float
	CS_REAL_TYPE	4 バイトの浮動小数点型	CS_REAL	real
money 型	CS_MONEY_TYPE	8 バイトの通貨型	CS_MONEY	money
	CS_MONEY4_TYPE	4 バイトの通貨型	CS_MONEY4	smallmoney
text 型および image 型	CS_TEXT_TYPE	テキスト型	CS_TEXT	text
	CS_UNITEXT_TYPE	符号なし可変長文字型	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

CS-Library には、データ型の操作に便利なルーチンがあります。その中には、次のものがあります。

- cs_calc ：decimal、float、money、numeric、real のデータ型に対して算術演算を行います。
- cs_cmp ：datetime、decimal、float、money、numeric、real のデータ型の比較を行います。
- cs_convert ：ある型のデータ値を別のデータ型に変換します。
- cs_dt_crack：マシンが読み取れる日時値をユーザがアクセス可能なフォーマットに変換します。
- cs_dt_info ：各国言語の日時情報を取得します。

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 Server のデータ型

binary 型

Open Server には、CS_BINARY、CS_LONGBINARY、CS_VARBINARY という 3 つの binary 型があります。

- CS_BINARY は、Adaptive Server Enterprise の binary データ型と varbinary データ型に対応します。つまり、Server-Library はサーバの binary 型と varbinary 型を CS_BINARY として解釈します。たとえば、srv_descfmt はクライアントからバイナリ・パラメータの記述を取得するときに、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;
  ```

- CS_LONGBINARY はどの Adaptive Server Enterprise データ型にも対応していませんが、一部の Open Server アプリケーションは CS_LONGBINARY をサポートします。アプリケーションでは、CS_DATA_LBIN 機能を使用して、Client-Library 接続で CS_LONGBINARY がサポートされているかどうかを確認できます。

  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。そのため、Open Server ルーチンは、CS_VARBINARY_TYPE を返しません。データ型が CS_VARBINARY_TYPE と記述された場合には、Open Server はそれを自動的に NULL が許される CS_BINARY_TYPE に変換してからクライアントに送ります。CS_VARBINARY_TYPE は、プログラム変数をバインドするときにのみ使用可能です。CS_VARBINARY によって、プログラマは C 以外のプログラミング言語で Open Server のプログラムを書くことができます。一般的なサーバ・アプリケーションは、CS_VARBINARY を使用しません。

  CS_VARBINARY の定義は次のとおりです。

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

  各パラメータの意味は、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。

ビット型

Open Server でサポートされているビット型は、CS_BIT のみです。このデータ型には、0 または 1 のサーバ・ビット値 (またはブール値) が格納されます。他の型を bit 型に変換すると、ゼロ以外の値はすべて 1 に変換されます。

```
typedef unsigned char      CS_BIT;
```

character 型

Open Server 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 データ型と varchar データ型に対応します。つまり、Server-Library はサーバの char データ型と varchar データ型を CS_CHAR として解釈します。たとえば、srv_descfmt はクライアントから文字パラメータの記述を取得するときに、CS_CHAR_TYPE を返します。

  CS_CHAR の定義は次のとおりです。

```
typedef char               CS_CHAR;
```

- CS_LONGCHAR はどの Adaptive Server Enterprise データ型にも対応していませんが、Client-Library アプリケーションによっては CS_LONGCHAR をサポートするものもあります。アプリケーションでは、CS_DATA_LCHAR 機能を使用して、Client-Library 接続が CS_LONGCHAR をサポートしているかどうかを調べることができます。

  CS_LONGCHAR 値は、最大 2,147,483,647 バイトの長さをサポートします。CS_LONGCHAR の定義は次のとおりです。

  ```
  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 Server ルーチンは、CS_VARCHAR_TYPE を返しません。データ型が CS_VARCHAR_TYPE と記述された場合には、Open Server はそれを自動的に null 入力可能な CS_CHAR_TYPE に変換してからクライアントに送ります。CS_VARCHAR_TYPE は、プログラム変数をバインドするときにのみ使用できます。CS_VARCHAR によって、プログラマは C 以外のプログラミング言語で Open Server のプログラムを書くことができます。一般的なサーバ・アプリケーションは、CS_VARCHAR を使用しません。

  CS_VARCHAR の定義は次のとおりです。

  ```
  typedef struct _cs_varchar
   {
       CS_SMALLINT          len;
       CS_BYTE              str[CS_MAX_CHAR];
    } CS_VARCHAR;
  ```

  各パラメータの意味は、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 で終了する文字列ではないことに注意してください。

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

Open Server アプリケーションは、CS-Library ルーチンの cs_dt_crack を使って日時構造体から日付要素 (年、月、日など) を抽出できます。

- 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。

  ```
  typedef struct _cs_datetime
   {
       CS_INT                dtdays;
       CS_INT                dttime;
   } CS_DATETIME;
  ```

  各パラメータの意味は、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。

```
typedef struct _cs_datetime4
 {
     unsigned short        days;
     unsigned short        minutes;
 } CS_DATETIME4;
```

  各パラメータの意味は、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_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、1753 年 1 月 1 日から 9999 年 12 月 31 日までです。

```
typedef struct _cs_date
 {
     CS_INT          days;
 } CS_DATE;
```

  days は、1900 年 1 月 1 日からの日数です。

- 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 ミリ秒) です。

```
typedef struct _cs_time
 {
     CS_INT          time;
 } CS_TIME;
```

  time は、深夜 0 時からの 300 分の 1 秒の数です。

- 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。

datetime の最小値と最大値

次の表に、datetime 型の最小値と最大値を示します。

表 *2-38: datetime* の最小値と最大値

データ型	最小値	最大値
CS_BIGDATETIME	January 1, 0001 00:00:00.000000	December 31, 9999 23:59:59.999999
CS_BIGTIME	00:00:00.000000	23:59:59.999999
CS_DATE	January 1, 0001	December 31, 9999
CS_DATETIME	January 1, 1753 00:00:00.000	December 31, 9999 23:59:59.999
CS_DATETIME4	January 1, 1900 00:00:00.000	June 6, 2079 23:59:59.999
CS_TIME	00:00:00.000	23:59:59.999

整数値型

Open Server 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。

大半のプラットフォームで、CS_TINYINT は 1 バイトの整数、CS_SMALLINT は 2 バイトの整数、CS_INT は 4 バイトの整数、CS_BIGINT は 8 バイトの整数、CS_USMALLINT は符号なし 2 バイトの整数、CS_UINT は符号なし 4 バイトの整数、CS_UBIGINT は符号なし 8 バイトの整数です。

```
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、プラットフォームに依存する 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、プラットフォームに依存する 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。

  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;
```

各パラメータの意味は、次のとおりです。

- *precision* は、数値の精度です。*precision* の有効値は、CS_MIN_PREC から CS_MAX_PREC の範囲です。デフォルトの精度は CS_DEF_PREC です。CS_MIN_PREC、CS_MAX_PREC、CS_DEF_PREC はそれぞれ、精度の最小値、最大値、デフォルト値を示します。
- *scale* は数値の位取りです。*scale* の有効値は、CS_MIN_SCALE から CS_MAX_SCALE の範囲です。デフォルトの位取りは CS_DEF_SCALE です。CS_MIN_SCALE、CS_MAX_SCALE、CS_DEF_SCALE はそれぞれ、位取りの最小値、最大値、デフォルト値を示します。
- *scale* は、*precision* 以下でなければなりません。

CS_DECIMAL 型の精度 (*precision*) と位取り (*scale*) のデフォルト値は、CS_NUMERIC 型と同じです。

money 型

Open Server では、CS_MONEY と CS_MONEY4 の 2 つの money 型がサポートされます。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。CS_MONEY の有効値は、+/- $922,337,203,685,477.5807 の範囲です。

```
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;
```

security 型

Open Server では、型定数 CS_BOUNDARY_TYPE と CS_SENSITIVITY_TYPE を定義することで、Secure Adaptive Server Enterprise の boundary と sensitivity データ型をサポートします。

これらの型定数は、他の Open Server の型定数とは異なり、似たような名前の型定義に対応しません。これらは CS_CHAR に対応します。

つまり、Open Server ルーチンがカラムや変数のデータ型を記述するために CS_BOUNDARY_TYPE と CS_SENSITIVITY_TYPE を受け入れて返す場合も、対応するプログラム変数の型は CS_CHAR でなければなりません。

たとえば、アプリケーションが、CS_DATAFMT 構造体の *datatype* フィールドを CS_SENSITIVITY_TYPE に設定し srv_bind を呼び出す場合、データがバインドされるプログラム変数の型は CS_CHAR でなければなりません。

text 型および image 型

Open Server は、text データ型 CS_TEXT と CS_UNITEXT、image データ型 CS_IMAGE をサポートします。

- CS_TEXT は、最大 2,147,483,647 バイトの印刷可能文字データを格納する可変長カラムを定義する、サーバのデータ型 text に対応しています。CS_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、共通の構文と語義を使用します。ただし、CS_UNITEXT では、文字データが 2 バイト UTF-16 形式でコード化されます。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、最大 2,147,483,647 バイトのバイナリ・データを格納する可変長カラムを定義する、サーバのデータ型 image に対応しています。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;
  ```

第 3 章 ルーチン

この章は、各 Server-Library ルーチンについて説明します。

ルーチン	説明	ページ
srv_alloc	メモリを割り付けます。	203
srv_alt_bind	計算ローのカラムにソース・データを記述し、バインドします。	205
srv_alt_descfmt	計算ロー・カラムの集合演算子と、クライアントに返されるカラム・データのフォーマットを記述します。	209
srv_alt_header	計算ローのロー識別子および bylist を記述します。	212
srv_alt_xferdata	クライアントに計算ローを送ります。	215
srv_bind	カラムまたはパラメータにプログラム変数を記述し、バインドします。	217
srv_bmove	バイトを、1 つのメモリ・ロケーションから別のメモリ・ロケーションにコピーします。	222
srv_bzero	メモリ・ロケーションの内容を 0 に設定します。	223
srv_callback	スレッドに対して状態遷移ハンドラをインストールします。	225
srv_capability	Open Server がプラットフォーム依存のサービスをサポートするかどうかを決定します。	228
srv_capability_info	クライアント接続に関する機能情報を定義または取得します。	229
srv_createmsgq	メッセージ・キューを作成します。	233
srv_createmutex	相互排他セマフォを作成します。	235
srv_createproc	非クライアントのイベント駆動型スレッドを作成します。	237
srv_cursor_props	現在のカーソルに関する情報を取得または設定します。	239
srv_dbg_stack	スレッドのコール・スタックを表示します。	241
srv_dbg_switch	デバッグのために、もう 1 つのスレッド・コンテキストを一時的にリストアします。	243
srv_define_event	ユーザ・イベントを定義します。	244
srv_deletemsgq	メッセージ・キューを削除します。	246
srv_deletemutex	srv_createmutex を使用して作成したミューテックスを削除します。	248
srv_descfmt	クライアントとやり取りするカラムまたはパラメータの記述を、記述または取得します。	250

ルーチン	説明	ページ
srv_dynamic	クライアントの動的 SQL コマンドを読み込む、またはコマンドに応答します。	253
srv_envchange	環境の変化をクライアントに通知します。	257
srv_event	スレッドの要求処理キューに、イベント要求を追加します。	259
srv_event_deferred	非同期イベントの結果として、スレッドのイベント・キューにイベント要求を追加します。	262
srv_free	割り付けられているメモリを解放します。	264
srv_freeserveraddrs	srv_getserverbyname によって割り付けられたメモリを解放します。	265
srv_get_text	連続したデータとして、text または image のデータ・ストリームをクライアントから読み込みます。	266
srv_getloginfo	リモート・サーバとのパススルー接続の準備のために、クライアント・スレッドからログイン情報を取得します。	268
srv_getmsgq	メッセージ・キューから次のメッセージを取得します。	270
srv_getobjid	指定の名前を持つ、メッセージ・キューまたはミューテックスのオブジェクト ID を検索します。	273
srv_getobjname	指定した名前を持つ、メッセージ・キューまたはミューテックスの名前を取得します。	275
srv_getserverbyname	*server_name* の接続情報を返し、必要に応じてメモリを割り付けます。	277
srv_handle	Open Server アプリケーションにイベント・ハンドラをインストールします。	278
srv_init	Open Server アプリケーションを初期化します。	281
srv_langcpy	クライアントの言語要求を、アプリケーション・バッファにコピーします。	282
srv_langlen	言語要求バッファの長さを返します。	285
srv_lockmutex	ミューテックスをロックします。	286
srv_log	Open Server アプリケーション・ログ・ファイルへメッセージを書き込みます。	288
srv_mask	SRV_MASK_ARRAY 構造体のビットを初期化、チェック、設定、またはクリアします。	290
srv_msg	メッセージ・データ・ストリームを送信または受信します。	292
srv_negotiate	ネゴシエーション・ログイン情報をクライアントに送信およびクライアントから受信します。	296

ルーチン	説明	ページ
srv_numparams	現在のクライアント・コマンドに含まれているパラメータの数を返します。	302
srv_options	オプション情報をクライアントに送信、またはクライアントから受信します。	304
srv_orderby	クライアントに order-by リストを返します。	309
srv_poll (UNIX のみ)	一連のオープン・ストリームのファイル記述子の I/O イベントをチェックします。	311
srv_props	Open Server プロパティを定義または取得します。	313
srv_putmsgq	メッセージ・キューにメッセージを入れます。	320
srv_realloc	メモリの再割り付けを行います。	322
srv_recvpassthru	クライアントからプロトコル・パケットを受信します。	323
srv_regcreate	レジスタード・プロシージャの登録を完了します。	326
srv_regdefine	プロシージャを登録するプロセスを開始します。	328
srv_regdrop	プロシージャの登録を解除します。	331
srv_regexec	レジスタード・プロシージャを実行します。	333
srv_reginit	レジスタード・プロシージャの実行を開始します。	335
srv_reglist	Open Server に登録されているすべてのプロシージャのリストを取得します。	337
srv_reglistfree	以前に割り付けられた SRV_PROCLIST 構造体を解放します。	338
srv_regnowatch	レジスタード・プロシージャの通知リストから、クライアント・スレッドを削除します。	340
srv_regparam	定義されているレジスタード・プロシージャに対してパラメータを記述する、またはレジスタード・プロシージャの実行に対してデータを提供します。	342
srv_regwatch	指定されたプロシージャの通知リストに、クライアント・スレッドを追加します。	345
srv_regwatchlist	クライアント・スレッドが通知要求待ちになっている、すべてのレジスタード・プロシージャのリストを返します。	347
srv_rpcdb	現在のリモート・プロシージャ・コールで指定されているデータベース要素を返します。	349
srv_rpcname	現在のリモート・プロシージャ・コールの名前から、RPC 名に当たる部分を返します。	351
srv_rpcnumber	現在のリモート・プロシージャで指定されている番号要素を返します。	353
srv_rpcoptions	現在のリモート・プロシージャ・コールのランタイム・オプションを返します。	355

ルーチン	説明	ページ
srv_rpcowner	現在のリモート・プロシージャ・コールで指定されている所有者要素を返します。	356
srv_run	Open Server アプリケーションを開始します。	358
srv_s_ssl_local_id	ローカル ID (認証) ファイルへのパスを指定するために使用します。	359
srv_select (UNIX のみ)	指定の I/O オペレーションに対してファイル記述子の準備ができているかどうかをチェックします。	359
srv_send_ctlinfo	Client-Library に制御メッセージを送信します。	362
srv_send_data	複数のカラムがあるローをクライアントに転送します。	364
srv_send_text	連続したデータとして、text または image のデータ・ストリームをクライアントに送ります。	368
srv_senddone	クライアントに結果完了メッセージを送るか、結果の一部をフラッシュします。	370
srv_sendinfo	クライアントにエラーメッセージ、または情報メッセージを送ります。	375
srv_sendpassthru	クライアントにプロトコル・パケットを送ります。	378
srv_sendstatus	クライアントにステータス値を送信します。	381
srv_setcolutype	カラムと関連付けられるユーザ・データ型を指定します。	382
srv_setcontrol	カラムのユーザ制御またはフォーマット情報を記述します。	384
srv_setloginfo	リモート・サーバからクライアントにプロトコルのフォーマット情報を返します。	386
srv_setpri	スレッドのスケジューリング優先順位を変更します。	388
srv_signal (UNIX のみ)	SIGIO または SIGURG シグナル用に signal と同じインタフェースを使用した UNIX シグナル・ハンドラをインストールします。	390
srv_sleep	現在実行中のスレッドを休止します。	392
srv_spawn	サービス・スレッドを割り付けます。	395
srv_symbol	Open Server のトークン値を読み込み可能な文字列に変換します。	398
srv_tabcolname	ブラウズ・モードの結果カラムに結果テーブルを関連付けます。	402
srv_tabname	一連のブラウズ・モードの結果と関連付けられたテーブルの名前を提供します。	404
srv_termproc	スレッドの実行を中止します。	406
srv_text_info	text または image データの記述を設定または取得します。	407

ルーチン	説明	ページ
srv_thread_props	スレッド・プロパティを定義し、取得します。	409
srv_timedsleep	イベントが通知されるまでスリープします。	414
srv_unlockmutex	ミューテックスのロックを解除します。	417
srv_version	アプリケーションが使用している Server-Library のバージョンを定義し、アプリケーションのデフォルトの使用言語と文字セットを定義します。	419
srv_wakeup	スリープ中のスレッドの実行を可能にします。	420
srv_xferdata	クライアントにパラメータやデータを送信するか、またはクライアントからパラメータやデータを受信します。	422
srv_yield	他のスレッドの実行を可能にします。	424

srv_alloc

説明

メモリを割り付けます。

構文

```
CS_VOID *srv_alloc(size)
CS_INT       size;
```

パラメータ

size

割り付けるバイト数です。

戻り値

表 *3-1:* 戻り値 *(srv_alloc)*

戻り値	意味
新しく割り付けられた領域へのポインタ	新しい領域のロケーション。
null の CS_VOID ポインタ	Open Server が *size* バイトを割り付けできなかった。

例

```
#include    <ospublic.h>
/*
 ** Local Prototype
 */
 CS_RETCODE      ex_srv_alloc PROTOTYPE((
 CS_BYTE         **bpp,
 CS_INT          size
 ));
/*
 ** EX_SRV_ALLOC
 **
 **    Example routine to allocate the specified amount of memory
 **    using srv_alloc.
```

```
**
** Arguments:
**    bpp     Return pointer to allocated memory here.
**    size    Amount of memory to allocate.
**
** Returns:
**
**    CS_SUCCEED            Memory was allocated successfully.
**    CS_FAIL               An error was detected.
*/
CS_RETCODE            ex_srv_alloc(bpp, size)
CS_BYTE              **bpp;
CS_INT               size;
{
    /* Initialization. */
    *bpp = (CS_BYTE *)NULL;

  /*
    ** Allocate size number of bytes.
    */
    if((*bpp = (CS_BYTE *)srv_alloc(size)) == (CS_BYTE *)NULL)
    {
        return(CS_FAIL);
    }
    return(CS_SUCCEED);
}

   /*
** Allocate size number of bytes.
*/
if((*bpp = (CS_BYTE *)srv_alloc(size)) == (CS_BYTE *)NULL)
    {
        return(CS_FAIL);
    }
    return(CS_SUCCEED);
}
```

使用法

- srv_alloc は、メモリを動的に割り付けます。このルーチンは、*size* バイト (それだけのバイト数が使用可能な場合) に、ポインタを返します。
- srv_alloc を使用して割り付けられたメモリは、srv_free を呼び出して解放します。
- 標準の C メモリ割り付けルーチンを使用する場合は、srv_alloc を使用します。

- 現行バージョンでは、srv_alloc は、C ルーチンの malloc を呼び出します。しかし、Open Server アプリケーションでは、srv_props ルーチンを使用して独自のメモリ管理ルーチンをインストールすることができます。ユーザによりインストールされたルーチンのパラメータの受け渡しに関する規則は、malloc の規則と同一でなければなりません。ユーザによりインストールされたルーチンを使用できるようにアプリケーションが設定されていない場合は、Open Server は malloc を呼び出します。

参照　srv_free、srv_props、srv_realloc

srv_alt_bind

説明　計算ローのカラムにソース・データを記述し、バインドします。

構文

```
CS_RETCODE srv_alt_bind(spp, altid, item, osfmtp,
      varaddr, varlenp, indp)

SRV_PROC        *spp;
CS_INT          altid;
CS_INT          item;
CS_DATAFMT      *osfmtp;
CS_BYTE         *varaddrp;
CS_INT          *varlenp;
CS_SMALLINT     *indp;
```

パラメータ

spp
内部スレッド制御構造体へのポインタです。

altid
この計算カラムが含まれている計算ロー用のユニークな識別子です。*altid* は、srv_alt_header を使って定義されます。

item
計算ロー内のカラムの番号です。カラム番号は、1 から始まります。

osfmtp
CS_DATAFMT 構造体を指すポインタです。この構造体は、アプリケーション・プログラム変数に含まれている計算ロー・カラム・データのフォーマットを記述します。

varaddrp
送信データがバインドされているプログラム変数へのポインタです。

varlenp
**varaddrp* の長さが格納されているプログラム変数へのポインタです。

indp
null 値のインジケータが格納されているバッファへのポインタです。次の表に、**indp* に指定できる値を示します。

表 3-2: indp の値 (srv_alt_bind)

値	意味
CS_NULLDATA	カラム・データは null。
CS_GOODDATA	カラム・データは null ではない。

indp が NULL の場合、カラム・データは有効、つまり、NULL ではないとみなされます。

戻り値

表 3-3: 戻り値 (srv_alt_bind)

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

```
#include     <ospublic.h>
/*
 ** Local prototype
 */
 CS_RETCODE             ex_srv_alt_bind PROTOTYPE((
 SRV_PROC               *spp,
 CS_INT                 altid,
 CS_VOID                *sump
 ));
/*
 ** EX_SRV_ALT_BIND
 **
 **    Example routine to describe and bind the source data for
 **    a compute row column.This example binds a value which
 **    is the sum of the first column of row data.
 **
 ** Arguments:
 **    spp    - A pointer to an internal thread control structure.
 **             The thread must be an active client thread that
 **             can handle row data.
 **
 **    altid    -     The id for this compute row.
 **
 **    sump    -    A pointer to the variable which will contain
 **                 the sum of the first column of row data.
 **
 ** Returns:
 **     CS_SUCCEED - Compute row column was successfully bound.
 **     CS_FAIL    - An error was detected.
 **
 */
 CS_RETCODE          ex_srv_alt_bind(spp, altid, sump)
 SRV_PROC            *spp;
 CS_INT              altid;
 CS_VOID             *sump;
```

```
{
   CS_DATAFMT compute_colfmt;
     /*
     **Format for this compute column.
     */
     CS_INT         namelen;
     /*
     **Length of compute column name
     */

   CS_INT        compute_colnum;
    /*
    **    The column number for this compute column.
    */

  CS_SMALLINT     indicator;
    /*
    **    Null indicator.
    */
    CS_INT            sumlen;
    /*
    **    Length of the compute value
    */
    CS_RETCODE        result;
    /*
    **Return value from srv_alt_bind.
    */

  /*
    ** Initialize the compute column's data format.This compute
    ** column represents a sum of the first column of data.
    */
    namelen = 3;
    srv_bmove("sum", compute_colfmt.name, namelen);

  compute_colfmt.namelen = namelen;
    compute_colfmt.datatype = CS_INT_TYPE;
    compute_colfmt.format = CS_FMT_UNUSED;
    compute_colfmt.maxlength = sizeof(CS_INT);
    compute_colfmt.scale = 0;

  compute_colfmt.precision = CS_DEF_PREC;
    compute_colfmt.status = 0;
    compute_colfmt.count = 0;
    compute_colfmt.usertype = 0;
    compute_colfmt.locale = (CS_LOCALE *)NULL;

  /*
    ** Perform the bind
    */
    compute_colnum = 1;
    indicator = CS_GOODDATA;
    sumlen = sizeof(CS_INT);
```

```
    result = srv_alt_bind(spp, altid, compute_colnum,
             &compute_colfmt, sump, &sumlen, &indicator);
     return (result);
}
```

使用法

- 計算ローの情報を返す Adaptive Server Enterprise の機能を模擬的に行うアプリケーションのみが、srv_alt_bind を呼び出す必要があります。srv_alt_bind は、Adaptive Server Enterprise へのゲートウェイとしての役割を果たしているアプリケーションにとって便利です。
- srv_alt_bind は、計算ロー・カラムのデータが保存されているアプリケーション・プログラム変数のフォーマットを記述します。アプリケーションは、計算ローの各カラムで一度ずつ srv_alt_bind を呼び出さなければなりません。
- srv_alt_bind ルーチンは、次の表に示す CS_DATAFMT フィールドに対して、読み取り (CS_GET) や設定 (CS_SET) を行います。他のすべてのフィールドは、srv_alt_bind には未定義です (“osfmtp” は構造体を指すポインタであることに注意してください)。

表 *3-4:* 使用される *CS_DATAFMT* フィールド *(srv_alt_bind)*

フィールド	CS_SET	CS_GET
osfmtp→datatype	アプリケーション・プログラム変数のデータ型	アプリケーション・プログラム変数のデータ型
osfmtp→maxlength	未使用	プログラム変数の最大長
osfmtp→count	0 または 1	0 または 1

- *osfmtp* が記述するフォーマットと srv_alt_descfmt (*clfmtp*) で設定されたクライアント・フォーマットが異なる場合には、Open Server は自動的にデータをクライアント・フォーマットに変換します。
- 計算結果セットは、1 つのローしか含みません。ただし、アプリケーションは、それぞれが独自の *altid* を持った複数の結果セットを返すことができます。
- 計算ローのデータを処理するために、Open Server アプリケーションが行うことを次に示します。
  a 計算ロー識別子を定義するために、srv_alt_header を呼び出します。
  b クライアントが受け取るときのカラム・データのフォーマットを記述するために、各カラムで srv_alt_descfmt を呼び出します。
  c データをローカル・プログラム変数にバインドするために、各カラムで srv_alt_bind を呼び出します。
  d 計算ローの各カラムが記述され、そのデータがプログラム変数にバインドされると、ローをクライアントに送信するために、srv_alt_xferdata を呼び出します。

- *varaddrp*、*lenp*、および *indp* が指すバッファの内容は、srv_xferdata が呼び出されるまでは、有効ではありません。

参照

srv_alt_descfmt、srv_alt_header、srv_alt_xferdata、「CS_DATAFMT 構造体」(48 ページ)

srv_alt_descfmt

説明

計算ロー・カラムの集合演算子と、クライアントに返されるカラム・データのフォーマットを記述します。

構文

```
CS_RETCODE srv_alt_descfmt(spp, altid, optype,
        operand, item, clfmtp)

SRV_PROC        *spp;
CS_INT          altid;
CS_INT          optype;
CS_TINYINT      operand;
CS_INT          item;
CS_DATAFMT      *clfmtp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

altid

この計算カラムが含まれている計算ロー用のユニークな識別子です。*altid* は、srv_alt_header を使って定義されます。

item

計算ロー内のカラムの番号です。カラム番号は、1 から始まります。

optype

計算ロー・カラムの集合演算子のタイプです。次の表に、有効な演算子のタイプを示します。

表 *3-5: optype* の値 *(srv_alt_descfmt)*

演算子の種類	機能
CS_OP_COUNT	カウント集合演算子
CS_OP_SUM	合計集合演算子
CS_OP_AVG	平均集合演算子
CS_OP_MIN	最小集合演算子
CS_OP_MAX	最大集合演算子

operand

集合演算の対象となる select リストのカラムです。

clfmtp

CS_DATAFMT 構造体へのポインタです。この構造体では、クライアントがカラム・データを受け取るときに使用する、カラム・データを含んだフォーマットを記述します。

戻り値

表 3-6: 戻り値 (srv_alt_descfmt)

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

```
#include       <ospublic.h>
 /*
 ** Local Prototype
 */
 CS_RETCODE      ex_srv_alt_descfmt PROTOTYPE((
 SRV_PROC        *sproc,
 CS_INT          altid,
 CS_DATAFMT      clfmtp[]
 ));
/*
 ** EX_SRV_ALT_DESCFMT
 **   An example routine to describe the aggregate operator of 2
 **   compute row columns and the format of each of the two column
 **   data returned to the client.We will do the sum on the first
 **   column and average on the second column.
 **
 ** Arguments:
 **   sproc     A pointer to an internal thread control structure.
 **   altid     The id for the compute row in which this compute
 **             column is contained.The altid is obtained by
 **             calling srv_alt_header.
 **   clfmtp    A pointer to the array of structures describing
 **             the format of the compute row column
 **             data when the client receives it.
 **
 ** Returns:
 **   CS_SUCCEED     If the aggregate operator and the datatype of
 **                  the compute row columns were successfully
 **                  described.
 **   CS_FAIL        An error was detected.
 */
 CS_RETCODE     ex_srv_alt_descfmt(sproc, altid, clfmtp)
 SRV_PROC       *sproc;
 CS_INT         altid;
 CS_DATAFMT     clfmtp[];
 {
      /*
      ** Describe the aggregate operator of the first compute row
```

```
        ** column and the format of the column data.
        */
        if ( srv_alt_descfmt(sproc, altid, (CS_INT)1, CS_OP_SUM,
             (CS_TINYINT)1, &clfmtp[0]) == CS_FAIL )
        {
             return(CS_FAIL);
        }

       /*
        ** Now do the same for the second column if (srv_alt_descfmt
        ** (sproc, altid, (CS_INT)2, CS_OP_AVG, (CS_TINYINT)2,
        ** &clfmtp[1]) == CS_FAIL )
        {
             return(CS_FAIL);
        }
        */

        return(CS_SUCCEED);
    }
```

使用法

- 計算ローの情報を返す Adaptive Server Enterprise の機能を模擬的に行うアプリケーションのみが、srv_alt_descfmt を呼び出す必要があります。srv_alt_descfmt は、Adaptive Server Enterprise へのゲートウェイとしての役割を果たしているアプリケーションにとって便利です。
- srv_alt_descfmt は、アプリケーションがクライアントに送る計算ロー・カラムを記述します。アプリケーションは、計算ローの各カラムで、一度ずつ srv_alt_descfmt を呼び出します。
- srv_alt_descfmt ルーチンは、次の表に示す CS_DATAFMT フィールドに対して、読み取り (CS_GET) や設定 (CS_SET) を行います。他のすべてのフィールドは、srv_alt_descfmt には未定義です (“clfmtp” は構造体を指すポインタであることに注意してください)。

表 *3-7:* 使用される *CS_DATAFMT* 構造体フィールド *(srv_alt_descfmt)*

フィールド	CS_SET	CS_GET
clfmtp→namelen	名前の長さ	名前の長さ
clfmtp→status	パラメータ／カラム・ステータス	パラメータ・ステータス
clfmtp→name	パラメータ／カラム名	パラメータ名
clfmtp→datatype	リモート・データ型をここに設定	リモート・データ型をここから取得
clfmtp→maxlength	リモート・データ型の最大長をここに設定	リモート・データ型の最大長をここから取得
clfmtp→format	リモート・データ型のフォーマット	リモート・データ型のフォーマット

- *clfmtp* により記述されるフォーマットが、その後で srv_alt_bind (*osfmtp*) で設定されたアプリケーション・プログラム変数のフォーマットと異なっている場合には、Open Server は、自動的にデータを *clfmtp* フォーマット記述に変換します。
- 計算ローのデータを処理するために、Open Server アプリケーションが行うことを次に示します。
  a 計算ロー識別子を定義するために、srv_alt_header を呼び出します。
  b クライアントが受け取るときのカラム・データのフォーマットを記述するために、各カラムで srv_alt_descfmt を呼び出します。
  c データをローカル・プログラム変数にバインドするために、各カラムで srv_alt_bind を呼び出します。
  d 計算ローの各カラムが記述され、そのデータがプログラム変数にバインドされると、ローをクライアントに送信するために、srv_alt_xferdata を呼び出します。

参照 srv_alt_bind、srv_alt_header、srv_alt_xferdata、「CS_DATAFMT 構造体」(48 ページ)

srv_alt_header

説明 計算ローのロー識別子および bylist を記述します。

構文

```
CS_RETCODE srv_alt_header(spp, altid, numbylist,
        bylistarrayp)

SRV_PROC        *spp;
CS_INT          altid;
CS_INT          numbylist;
CS_SMALLINT     *bylistarrayp;
```

パラメータ

spp
内部スレッド制御構造体へのポインタです。

altid
この計算ロー用のユニークな識別子です。

numbylist
計算ローの bylist 内のカラム数です。

bylistarrayp
計算ロー用の bylist を構成しているカラム番号の配列を指すポインタです。要素は、*numbylist* に指定されている数だけあります。*numbylist* が 0 の場合は、*bylistarrayp* は無視されます。

戻り値

表 3-8: 戻り値 *(srv_alt_header)*

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

```
#include            <ospublic.h>
/*
 ** Local Prototype
 */
 CS_RETCODE           ex_srv_alt_header PROTOTYPE((
 SRV_PROC             *spp
 ));

/*
 ** EX_SRV_ALT_HEADER
 **
 **     Example routine to illustrate the use of srv_alt_header
 **     to describe a compute row's row identifier and bylist.
 **
 ** Arguments:
 **     spp - A pointer to an internal thread control structure.
 **
 ** Returns:
 **
 **   CS_SUCCEED     A compute row was successfully described.
 **   CS_FAIL        An error was detected.
 */
 CS_RETCODE     ex_srv_alt_header(spp)
 SRV_PROC     *spp;

{
      CS_INT          altid;
      CS_SMALLINT     bylist[2];

    /*
      ** Let us describe a fictitious compute row with altid =1,
      ** and bylist = [2,4].
      */
      altid = (CS_INT)1;
      bylist[0] = (CS_SMALLINT)2;
      bylist[1] = (CS_SMALLINT)4;

     if (srv_alt_header(spp, altid,
           sizeof(bylist)/sizeof(CS_SMALLINT),
                bylist) == CS_FAIL)
           return (CS_FAIL);

     return (CS_SUCCEED);
 }
```

使用法

- 計算ローの情報を返す Adaptive Server Enterprise の機能を模擬的に行うアプリケーションのみが、srv_alt_header を呼び出す必要があります。srv_alt_header は、Adaptive Server Enterprise へのゲートウェイとしての役割を果たしているアプリケーションにとって便利です。
- srv_alt_header は、各計算ローにユニークな識別子を割り当て、各計算ローと関連する bylist を記述します。各計算ローで一度ずつ srv_alt_header を呼び出さなければなりません。
- Adaptive Server Enterprise では、計算ローは、Transact-SQL select 文の compute 句によって生成されます。Transact-SQL の select 文に複数の compute 句がある場合には、各句によって別々の計算ローが生成されます。Open Server は、Adaptive Server Enterprise の Transact-SQL の compute 句に対する応答を真似て、計算データのローを返すことができます。
- Transact-SQL の select 文の compute 句には、by という「キーワード」を含めて、その後にカラムのリストを続けることができます。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。
- **bylistarrayp* の配列では、bylist の各カラムに関連する番号が保管されます。この番号は、select 文内のカラムの位置によって決まります。たとえば、カラムが、select 文での 3 つ目のアイテムである場合には、配列でナンバー 3 としてリストされます。
- 計算ローのデータを処理するために、Open Server アプリケーションが行うことを次に示します。
  a 計算ロー識別子を定義するために、srv_alt_header を呼び出します。
  b クライアントが受け取るときのカラム・データのフォーマットを記述するために、各カラムで srv_alt_descfmt を呼び出します。
  c データをローカル・プログラム変数にバインドするために、各カラムで srv_alt_bind を呼び出します。
  d 計算ローの各カラムが記述され、そのデータがプログラム変数にバインドされると、ローをクライアントに送信するために、srv_alt_xferdata を呼び出します。

参照

srv_alt_bind、srv_alt_descfmt、srv_alt_xferdata

srv_alt_xferdata

説明

クライアントに計算ローを送ります。

構文

```
CS_RETCODE srv_alt_xferdata(spp, altid)
SRV_PROC    *spp;
CS_INT      altid;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

altid

クライアントに送られる計算ロー用のユニークな識別子です。*altid* は、srv_alt_header を使って定義されます。

戻り値

表 *3-9:* 戻り値 *(srv_alt_xferdata)*

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

```
#include          <ospublic.h>
/*
 ** Local Prototype.
 */

CS_RETCODE      ex_srv_alt_xferdata PROTOTYPE((
 SRV_PROC        *spp,
 CS_INT          altid
 ));

/*
 ** EX_SRV_ALTXFERDATA
 **
**     Example routine to send a compute row the the client using
 **     srv_altxferdata.

**
** Arguments:
**      spp     A pointer to an internal thread control structure.
**  altid       The compute row identifier (defined using
 **              srv_alt_header).
 **
 ** Returns:
 **
 **    CS_SUCCEED    The row was sent to the client.
**    CS_FAIL       An error was detected.
```

```
 */
 CS_RETCODE          ex_srv_alt_xferdata(spp, altid)
SRV_PROC        *spp;
CS_INT          altid;
{
     /*
     ** Send the compute row to the client.
     */
     if (srv_alt_xferdata(spp, altid) != CS_SUCCEED)
     {
          return (CS_FAIL);
     }
     return (CS_SUCCEED);
}
```

使用法

- 計算ローの情報を返す Adaptive Server Enterprise の機能を模擬的に行うアプリケーションのみが、srv_alt_xferdata を呼び出す必要があります。これは、Adaptive Server Enterprise へのゲートウェイとしての役割を果たしているアプリケーションにとって便利です。
- srv_alt_xferdata は、クライアントに計算ローを送ります。このルーチンは、各 *altid* で一度呼び出されます。
- 計算ローのデータを処理するために、Open Server アプリケーションが行うことを次に示します。
  a 計算ロー識別子を定義するために、srv_alt_header を呼び出します。
  b クライアントが受け取るときのカラム・データのフォーマットを記述するために、各カラムで srv_alt_descfmt を呼び出します。
  c データをローカル・プログラム変数にバインドするために、各カラムで srv_alt_bind を呼び出します。
  d 計算ローの各カラムが記述され、そのデータがプログラム変数にバインドされると、ローをクライアントに送信するために、srv_alt_xferdata を呼び出します。
- srv_senddone で送信完了のステータスを送る前に、すべての計算ローをクライアントに送ります。

参照

srv_alt_bind、srv_alt_header、srv_alt_descfmt

srv_bind

説明

カラムまたはパラメータにプログラム変数を記述し、バインドします。

構文

```
CS_RETCODE srv_bind(spp, cmd, type, item, osfmtp,
        varaddrp, varlenp, indp)
SRV_PROC        *spp;
CS_INT          cmd;
CS_INT          type;
CS_INT          item;
CS_DATAFMT      *osfmtp;
CS_BYTE         *varaddrp;
CS_INT          *varlenp;
CS_SMALLINT     *indp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

cmd

cmd は、プログラム変数が、クライアントに送られるデータまたはクライアントから入ってくるデータを保存するかどうかを示します。次の表に、*cmd* の有効値を示します。

表 *3-10: cmd* の値 *(srv_bind)*

値	説明
CS_SET	**varaddrp* のデータは、srv_xferdata が呼び出されたときにクライアントに送信される。
CS_GET	**varaddrp* は、srv_xferdata の呼び出しの後に、クライアントからのデータで初期化される。

type

プログラム変数に保管されたり、プログラム変数から読み取られるデータの型です。表 3-11 に、*type* の有効値を示します。

表 *3-11: type* の値 *(srv_bind)*

型	有効な cmd	データの内容
SRV_RPCDATA	CS_SET または CS_GET	RPC またはストアド・プロシージャ・パラメータ
SRV_ROWDATA	CS_SET のみ	結果ロー・カラム
SRV_CURDATA	CS_GET のみ	カーソル・パラメータ
SRV_KEYDATA	CS_GET のみ	カーソル・キー・カラム
SRV_ERRORDATA	CS_SET のみ	エラー・メッセージ・パラメータ
SRV_DYNAMICDATA	CS_SET または CS_GET	動的 SQL パラメータ
SRV_NEGDATA	CS_SET または CS_GET	ネゴシエーション・ログイン・パラメータ
SRV_MSGDATA	CS_SET または CS_GET	メッセージ・パラメータ
SRV_LANGDATA	CS_GET のみ	言語パラメータ

item
カラムまたはパラメータの番号です。カラムやパラメータ番号は、1 から始まります。

osfmtp
CS_DATAFMT 構造体を指すポインタです。この構造体は、**varaddrp* に保存されているデータのフォーマットを記述します。

varaddrp
カラムまたはパラメータのデータがバインドされているプログラム変数へのポインタです。

varlenp
varaddrp の長さを指すポインタです。その正確な意味およびプロパティは、*cmd* の値によって変わります。表 3-12 に、*varlenp* の有効値を示します。

表 *3-12: varlenp* の値 *(srv_bind)*

cmd	varlenp
CS_SET (クライアントに送られるデータ)	• NULL にできない。 • **varaddrp* のデータの実際の長さを指す。 • srv_xferdata が呼び出されるまでは有効である必要はない。
CS_GET (クライアントから入ってくるデータ)	• NULL にできる (Open Server アプリケーションがすでにデータ長を認識していることを示す)。 • Open Server が実データ長を格納するプログラム変数へのポインタ。 • srv_xferdata への呼び出しの後に値が取り込まれる。

データを取得するときには、アプリケーションが srv_xferdata を呼び出すまで、**varlenp* は空です。srv_xferdata を呼び出すと、Open Server は新しく受信した値の長さをバッファに格納します。データを送信するときには、アプリケーションは、データを送信するために srv_xferdata を呼び出す前に **varlenp* が指すバッファにデータを設定します。

indp
null 値のインジケータが格納されているバッファへのポインタです。表 3-13 に、**indp* の有効値を示します。

表 *3-13: indp* の値 *(srv_bind)*

値	意味
CS_NULLDATA	カラムまたはパラメータのデータは null である。
CS_GOODDATA	カラムまたはパラメータのデータは null ではない。

indp が NULL の場合、カラム・データは有効、つまり、NULL ではないとみなされます。

戻り値

表 *3-14:* 戻り値 *(srv_bind)*

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

```
#include  <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE      ex_srv_bind PROTOTYPE((
SRV_PROC                *spp,
CS_INT                  *nump,
CS_BYTE                 *namep,
CS_INT                  *lenp
));
/*
 ** EX_SRV_BIND
**
**     Example routine using srv_bind to describe and bind two
 **     program.
**     variables to receive client RPC parameters.For this
 **     example, the
**     RPC is passed an employee number, and last name.A third
 **     program.
**     variable will be bound to receive the length of the
 **     employee's name.
**     This routine is called prior to srv_xferdata, which will
 **     actually transfer the data into the program variables.
**
** Arguments:
**     spp     A pointer to an internal thread control structure.
**     nump    A Pointer to the integer to receive the employee
 **             number.
**     namep   A Pointer to the memory area to receive the
 **             employee name.
**    lenp     A Pointer to the integer to receive the length of
 **             the employee's name.(On input, points to the
 **             maximum length of the memory area available.)
**
 ** Returns:
**    CS_SUCCEED    Program variables were successfully bound.
**    CS_FAIL       An error was detected.
 */
CS_RETCODE     ex_srv_bind(spp, nump, namep, lenp)
SRV_PROC        *spp;
CS_INT          *nump;
CS_BYTE         *namep;
CS_INT          *lenp;
```

```
{
     CS_INT               param_no;
     CS_DATAFMT           varfmt;
     srv_bzero((CS_VOID *)&varfmt, (CS_INT)sizeof(varfmt));
     /*
     ** First, bind the integer to receive the employee number,
     ** param 1. Here, we know the length of the data, so no
     ** length pointer is required.
     */
     param_no = 1;
     varfmt.datatype = CS_INT_TYPE;
     varfmt.maxlength = (CS_INT)sizeof(CS_INT);
     if (srv_bind(spp, (CS_INT)CS_GET, (CS_INT)SRV_RPCDATA,
          param_no, &varfmt, (CS_BYTE *)nump, (CS_INT *)NULL,
          (CS_SMALLINT *)NULL) != CS_SUCCEED)
     {
          return(CS_FAIL);
     }
     /*
      ** Then, bind the character memory to receive the
      ** employee name, param 2.
      */
     param_no = 2;
     varfmt.datatype = CS_CHAR_TYPE;
     varfmt.maxlength = *lenp;
     if (srv_bind(spp, (CS_INT)CS_GET, (CS_INT)SRV_RPCDATA,
           param_no,
          &varfmt, namep, lenp, (CS_SMALLINT *)NULL) !=
           CS_SUCCEED)
     {
          return(CS_FAIL);
     }
     return(CS_SUCCEED);
}
```

使用法

- srv_bind は、ロー・カラムまたはパラメータのフォーマットを記述し、それをアプリケーション・プログラム変数と関連付けます。
- srv_bind は、結果ローの各カラムでまたはパラメータ・ストリームの各パラメータで、一度ずつ呼び出されなければなりません。
- ローを送信している間にローカル・プログラム変数アドレス (*varaddrp*、*varlenp*、または *indp*) を変えたいアプリケーションでは、変更を行うたびに srv_bind、続いて srv_xferdata を呼び出されなければなりません。
- Server-Library アプリケーションは、クライアントにデータを 2 段階に分けて送信します。

最初に、srv_bind を CS_SET と等しい *cmd* で呼び出します。*varaddrp*、*varlenp*、*indp* の各パラメータはそれぞれ、検索するデータへのポインタ、長さへのポインタ、インジケータ変数へのポインタを保持しています。このとき、Server-Library はこれらのポインタ・パラメータに渡されたアドレスを記録します。

これらの値は、アプリケーションが srv_xferdata を呼び出すまで、つまり Server-Library がそれらのメモリ・ロケーションから値を読み取るまで有効でなければなりません。たとえば、複数のデータ・アイテムが srv_bind への異なる呼び出しに対して渡されるときには、異なるバッファを使用する必要があります。

- エラー・データ・パラメータは、srv_sendinfo の呼び出しの直後で、srv_senddone を呼び出す前に記述され (srv_descfmt)、バインドされ (srv_bind)、クライアントに送られ (srv_xferdata) なければなりません。srv_descfmt、srv_bind、srv_xferdata ルーチンの type 引数は、SRV_ERRORDATA に設定されています。
- メッセージ・データ・パラメータは、srv_msg ルーチンの呼び出しの後に記述され (srv_descfmt)、バインドされ (srv_bind)、転送 (srv_xferdata) されなければなりません。srv_descfmt、srv_bind、srv_xferdata ルーチンの type 引数は、SRV_MSGDATA に設定されています。
- srv_bind ルーチンは、次の表に示す CS_DATAFMT フィールドに対して、読み取り (CS_GET) や設定 (CS_SET) を行います。他のすべてのフィールドは、srv_bind には未定義です (“osfmtp” は構造体を指すポインタであることに注意してください)。

表 *3-15:* 使用される *CS_DATAFMT* フィールド *(srv_bind)*

フィールド	CS_SET オペレーションにおける定義	CS_GET オペレーションにおける定義
osfmtp->datatype	アプリケーション・プログラム変数のデータ型	アプリケーション・プログラム変数のデータ型
osfmtp->maxlength	プログラム変数の実際の長さ	プログラム変数の最大長
osfmtp->count	0 または 1	0 または 1
osfmtp->status	null 値を送信している場合、CS_CANBENULL を設定	未使用

カラムの null 値を送信するには、そのカラムの CS_DATAFMT 構造体の *status* 値に CS_CANBENULL ビット・セットが設定されている必要があります。CS_DATAFMT 構造体の *status* の有効値については、表 2-9 (51 ページ) を参照してください。

- *osfmtp* によって記述されたフォーマットが、クライアントから受信したデータ・フォーマット (*cmd* を CS_GET に設定) と異なる場合には、Open Server は、動的にフォーマットを *osfmtp* に変換します。フォーマットが、クライアントに送られるフォーマット (*cmd* を CS_SET に設定) と異なる場合には、Open Server は、自動的にフォーマットをクライアント・フォーマットに変換します (*clfmtp*)。

参照 srv_cursor_props、srv_descfmt, srv_msg、srv_sendinfo、srv_xferdata、「CS_DATAFMT 構造体」(48 ページ)、「パラメータとロー・データの処理」(126 ページ)

srv_bmove

説明 バイトを、1 つのメモリ・ロケーションから別のメモリ・ロケーションにコピーします。

構文

```
CS_VOID srv_bmove(sourcep, destp, count)
CS_VOID     *sourcep;
CS_VOID     *destp;
CS_INT      count;
```

パラメータ

sourcep
コピーされるデータの元を指す非 null ポインタです。

destp
コピーされるデータの行き先を指す非 null ポインタです。

count
sourcep から *destp* にコピーされるバイト数です。

戻り値 なし。

例

```
#include  <ospublic.h>
/*
 ** Local Prototype
 */
CS_VOID          ex_srv_bmove PROTOTYPE((
CS_VOID          *src,
CS_VOID          *dest,
CS_INT           count
));

/*
** EX_SRV_BMOVE
**
**     Example routine to copy data from one area of memory to
        another.
 **
** Arguments:
**     src     - The address of the source data.
**     dest    - The address of the destination buffer.
**     count   - The number of bytes to copy.
**
 ** Returns:
**          Nothing.
```

```
 */
CS_VOID          ex_srv_bmove(src, dest, count)
CS_VOID          *src;
CS_VOID          *dest;
CS_INT           count;
{
     /*
     ** Call the Open Server routine that will do the
      ** actual copy.
     */
     srv_bmove(src, dest, count);

     /*
     ** All done.
     */
     return;
}
```

使用法

- srv_bmove は、*count* バイトをメモリ・ロケーション **sourcep* からメモリ・ロケーション **destp* にコピーします。
- *sourcep* も *destp* も、有効な非 null ポインタでなければ、メモリ・フォールトが生成されます。
- 移動するのは、*count* バイトだけで、null ターミネータは追加されません。

参照

srv_bzero

srv_bzero

説明

メモリ・ロケーションの内容を 0 に設定します。

構文

```
CS_VOID srv_bzero(locationp, count)
CS_VOID    *locationp;
CS_INT     count;
```

パラメータ

locationp

0 に設定されるバッファのアドレスを指す非 null ポインタです。

count

locationp で、0x00 の値に設定するバイト数です。

戻り値

なし。

例

```
#include          <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE           ex_srv_bzero PROTOTYPE((
```

```
 CS_VOID              *locationp,
 CS_INT               cnt
 ))
/*
 ** EX_SRV_BZERO
**     Example routine to set the contents of a section of memory
 **     to zero using srv_bzero
**
** Arguments:
**
**     memp          Pointer to section of memory.
**     count         Number of bytes to set to zero.
**
** Returns
**     CS_SUCCEED     Arguments were valid and srv_bzero called.
**     CS_FAIL        An error was detected.
**
 */
CS_RETCODE      ex_srv_bzero(memp, count)
CS_VOID         *memp;
CS_INT          count;
{
     /* Check arguments. */
     if(memp == (CS_VOID *)NULL)
     {
          return(CS_FAIL);
     }
     if(count < 0)
     {
          return(CS_FAIL);
     }

     /*
     ** Set the section of memory to the value 0x00.
     */
     (CS_VOID)srv_bzero(memp,count);
     return(CS_SUCCEED);
}
```

使用法

- srv_bzero は、*locationp* のメモリ・ロケーションで *count* 数のバイトを 0x00 の値に設定します。
- *locationp* が有効な非 null ポインタでない場合は、メモリ・フォールトが発生します。

参照

srv_bmove

srv_callback

説明

スレッドに対して状態遷移ハンドラをインストールします。

構文

```
CS_RETCODE srv_callback(spp, callback_type, funcp)
SRV_PROC        *spp;
CS_INT          callback_type;
CS_RETCODE      (*funcp)();
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

callback_type

コールバックがインストールされている対象の状態遷移を示す整数です。表 3-16 に、*callback_type* の有効値を示します。

表 3-16: *callback_type* の値 *(srv_callback)*

値	説明
SRV_C_EXIT	スレッドは、srv_spawn で指定されたエントリ・ポイントから戻ったか、あるいは切断されたクライアントと関連している。ハンドラは、終了しているスレッドのコンテキストで実行。
SRV_C_PROCEXEC	レジスタード・プロシージャが呼び出され、実行しようとしている。ハンドラは、レジスタード・プロシージャを要求したスレッドのコンテキストで実行する。
SRV_C_RESUME	スレッドは再開中である。ハンドラはスケジューラ・スレッドのコンテキストで実行され、そのスタックを使用。
SRV_C_SUSPEND	スレッドは中断している。ハンドラは、中断しているスレッドのコンテキストで実行され、そのスタックを使用する。
SRV_C_TIMESLICE	この状態遷移のためにインストールするコールバック・ルーチンは、SRV_S_TIMESLICE、SRV_S_VIRTCLKRATE、SRV_S_VIRTTIMER のサーバ・プロパティによって決められた期間 (タイム・スライス) を、スレッドが実行したときに呼び出される。詳細については、srv_props (313 ページ) と「プロパティ」(130 ページ) を参照してください。

funcp

指定の状態遷移が起こったときに呼び出す関数へのポインタです。

コールバック関数は、スレッド・ポインタ引数を持ちます。

戻り値

表 3-17: 戻り値 *(srv_callback)*

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

```
#include          <stdio.h>
#include          <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE        suspend_handler PROTOTYPE((
SRV_PROC          *srvproc
));
CS_RETCODE ex_srv_callback PROTOTYPE((
SRV_PROC          *srvproc
));

CS_RETCODE     suspend_handler(srvproc)
SRV_PROC       *srvproc;
{
     printf("Wake me when it's over...¥n");
     return(CS_SUCCEED);
}

/*
** EX_SRV_CALLBACK
**
**     Example routine to install a state transition handler.
**
** Arguments:
**   srvpro  - A pointer to an internal thread control structure.
**
 ** Returns:
**
**     CS_SUCCEED
**     CS_FAIL
*/
CS_RETCODE          ex_srv_callback(srvproc)
SRV_PROC            *srvproc;
{
     return(srv_callback(srvproc, SRV_C_SUSPEND,
                 suspend_handler));
}
```

使用法

- スレッドのステータスが別のステータスに遷移するときに実行するルーチンを、srv_callback を使って指定します。
- アプリケーションは、ステータス遷移中のスレッドへのポインタを指定して、コールバック・ルーチンを呼び出します。

- 表 3-18 に、各コールバック・ルーチンが返す値を示します。

表 *3-18:* コールバック・ルーチンの有効な戻り値 *(srv_callback)*

コールバック・ルーチンのタイプ	戻り値	戻り値の説明
SRV_C_EXIT	Open Server では無視されるが、将来の互換性を考えて SRV_CONTINUE に設定することを推奨。	
SRV_C_PROCEXEC	SRV_S_INHIBIT	レジスタード・プロシージャの実行を中止。
	SRV_S_CONTINUE	レジスタード・プロシージャの実行を継続。
SRV_C_RESUME	Open Server では無視されるが、将来の互換性を考えて SRV_CONTINUE に設定することを推奨。	
SRV_C_SUSPEND	Open Server では無視されるが、将来の互換性を考えて SRV_CONTINUE に設定することを推奨。	
SRV_C_TIMESLICE	SRV_CONTINUE	実行を中断せず継続。
	SRV_TERMINATE	スレッドを終了。
	SRV_DEBUG	後でデバッガで検査するために、スレッドをデバッグ・キューに追加。

- コールバックの中には、プラットフォームによっては使用できないものもあります。現在のプラットフォームでハンドラがインストール可能かどうかは、srv_capability を呼び出して調べることができます。
- 以前の srv_callback への呼び出しでインストールしたコールバック・ルーチンを削除するには、そのコールバック・ルーチンの代わりに null 関数をインストールします。たとえば、先にインストールした SRV_C_TIMESLICE ハンドラのインストールを取り消したい場合は、次のコマンドを発行します。

```
srv_callback(spp, SRV_C_TIMESLICE, NULL);
```

- アプリケーションが、コールバック・ハンドラを通知にだけしか使用しない場合は、*funcp* 引数を NULL に設定してください。詳細については、「レジスタード・プロシージャ」(151 ページ) を参照してください。

参照　srv_capability、srv_props、srv_termproc

srv_capability

説明

Open Server がプラットフォーム依存のサービスをサポートするかどうかを決定します。

構文

```
CS_BOOL srv_capability(capability)
CS_INT          capability;
```

パラメータ

capability

テストする Open Server サービスを表す定数です。表 3-19 に *capability* の有効値を示します。

表 3-19: capability の値 (srv_capability)

値	説明
SRV_C_DEBUG	srv_dbg_stack および srv_dbg_switch をサポートする。
SRV_C_EXIT	スレッドが終了すると、コールバック・ルーチンを呼び出すことができる。
SRV_C_RESUME	スレッドが実行を再開すると、コールバック・ルーチンを呼び出すことができる。
SRV_C_PREEMPT	プリエンプティブ・スケジューリングをサポートする。
SRV_C_SELECT	srv_select をサポートする。
SRV_C_SUSPEND	スレッドが中断すると、コールバック・ルーチンを呼び出すことができる。
SRV_C_TIMESLICE	スレッドがクロック・チックの最大数を超えると、コールバック・ルーチンを呼び出すことができる。
SRV_POLL	srv_poll をサポートする。

戻り値

表 3-20: 戻り値 (srv_capability)

戻り値	意味
CS_TRUE	Open Server はサービスをサポートする。
CS_FALSE	Open Server はサービスをサポートしない。

例

```
#include          <ospublic.h>
/*
 ** Local Prototype
 */
extern CS_RETCODE ex_srv_capability PROTOTYPE((void));
/*
 ** EX_SRV_CAPABILITY
**
**     Example routine to determine whether srv_poll is supported
 **     on this platform.
 **
** Arguments:
**     None.
**
 ** Returns:
```

```
**
**      CS_SUCCEED    srv_poll is supported on this platform.
**      CS_FAIL       srv_poll is not supported on this platform.
 */
CS_RETCODE          ex_srv_capability()
{
     CS_BOOL supported;
     /*
      ** Check to see whether srv_poll is supported on this
      ** platform.
      */
      supported = srv_capability(SRV_C_POLL);
     /*
      ** If "supported" is CS_TRUE, we return CS_SUCCEED, if it is
      ** CS_FALSE we return CS_FAIL.
      */
      return(supported ?CS_SUCCEED : CS_FAIL);
}
```

使用法

- srv_capability を使用すると、移植可能な Open Server アプリケーションを作成しながら、プラットフォームによっては使用できないサービスも利用することができます。
- Open Server には、プラットフォーム機能とプロトコル機能の 2 つの機能があります。srv_capability ルーチンは、プラットフォーム機能に関するものです。srv_capability_info ルーチンは、プロトコル機能に関するものです。詳細については、srv_capability_info のページを参照してください。

参照

srv_callback、srv_capability、srv_dbg_stack、srv_dbg_switch、srv_poll (UNIX のみ)、srv_select (UNIX のみ)、srv_capability_info

srv_capability_info

説明

クライアント接続に関する機能情報を定義または取得します。

構文

```
CS_RETCODE srv_capability_info(spp, cmd, type,
                capability, valp)
SRV_PROC    *spp;
CS_INT      cmd;
CS_INT      type;
CS_INT      capability;
CS_VOID     *valp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

cmd

Open Server アプリケーションが機能情報を定義しているのか取得しているのかを示します。表 3-21 に、*cmd* の有効値を示します。

表 3-21: cmd の値 (srv_capability_info)

値	意味
CS_SET	Open Server アプリケーションは機能情報を定義している。
CS_GET	Open Server アプリケーションは、クライアントからの機能情報を取得している。

type

機能グループのタイプです。表 3-22 に、有効な 2 つのタイプを示します。

表 3-22: type の値 (srv_capability_info)

値	意味
CS_CAP_REQUEST	クライアントが送る可能性のあるコマンド。
CS_CAP_RESPONSE	クライアントが Open Server アプリケーションを保留させる可能性のある応答。

capability

対象となる機能項目を指定します。*type* カテゴリのすべての機能項目のビットマップを取得するには、*capability* を CS_ALL_CAPS に設定します。すべての要求および応答機能のリストについては、「機能」(22 ページ) を参照してください。

valp

プログラム変数へのポインタです。クライアントに情報を送る場合 (CS_SET)、この変数には機能の値が設定されます。クライアントからの情報を取得する場合 (CS_GET)、Open Server によって、この変数に機能の値が設定されます。*valp* は、アプリケーションが個々の機能アイテムを定義または取得している場合には、CS_BOOL ポインタ、およびすべての機能項目のビットマップを定義または取得している場合には、CS_CAP_TYPE ポインタでなくてはなりません (つまり、*capability* は CS_ALL_CAPS)。

戻り値

表 3-23: 戻り値 (srv_capability_info)

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

```
#include <ospublic.h>
CS_RETCODE     ex_srv_capability_info PROTOTYPE((
 SRV_PROC       *spp
));
/*
** EX_SRV_CAPABILITY_INFO
**
**     Example routine to retrieve and define capability
 **     information on a client connection.
**
**     This routine must called in the context of the connect
 **     handler, so that it is legal to negotiate capabilities.
**
** Arguments:
**     spp  A pointer to an internal thread control structure.
**
 ** Returns:
**      CS_SUCCEED - Successfully retrieved and bound capability
 **                   information.
**      CS_FAIL - An error was detected.
**
*/
CS_RETCODE               ex_srv_capability_info(spp)
SRV_PROC                 *spp;

{
     CS_RETCODE       retval;  /* Return value from Open */
                               /* Server API calls. */

     CS_CAP_TYPE      capabilities;  /* Our bit mask. */

     CS_BOOL     value; /* Set to CS_TRUE or CS_FALSE */
                        /* for individual capabilities. */

     /*
     ** In this example, we don't want to support text or image,
      ** so we'll see first if the client has requested this.
      ** We'll do this by getting the entire bit mask.
     */
     retval = srv_capability_info(spp, CS_GET, CS_CAP_REQUEST,
                          CS_ALL_CAPS, (CS_VOID *)&capabilities);

     if (retval == CS_FAIL)

     {
          return (CS_FAIL);
     }

     /*
```

```
        ** Turn off text and image.
        **
        ** The other way to do this is to just clear the
         ** CS_DATA_TEXT and CS_DATA_IMAGE bits in the capabilities
         ** bit mask, and then call srv_capability_info() with
         ** CS_ALL_CAPS for the "type" parameter and the altered
         ** bit mask as the value.
        */
        if (CS_TST_CAPMASK(&capabilities, CS_DATA_TEXT) == CS_TRUE)
        {
             value = CS_FALSE;
             retval = srv_capability_info(spp, CS_SET,
                   CS_CAP_REQUEST, CS_DATA_TEXT, (CS_VOID *)&value);
             if (retval == CS_FAIL)
             {
                  return (CS_FAIL);
             }
        }

        if (CS_TST_CAPMASK(&capabilities, CS_DATA_IMAGE) == CS_TRUE)
        {
             value = CS_FALSE;
             retval = srv_capability_info(spp, CS_SET,
                        CS_CAP_REQUEST, CS_DATA_IMAGE, (CS_VOID*)
                        &value);
             if (retval == CS_FAIL)
             {
                  return (CS_FAIL);
             }
        }

        return (CS_SUCCEED);
    }
```

使用法

- クライアントから発行可能な要求、および Open Server アプリケーションから返信可能な応答については、Open Server アプリケーションとクライアントがお互いに認識できる必要があります。クライアント／サーバ接続の機能が、その接続について許可されるクライアント要求とサーバ応答の種類を決定します。
- Open Server は、すべての接続について、機能のデフォルト・セットを指定します。特定の接続に対して機能のデフォルト・セットを適用しない場合、Open Server アプリケーションは、srv_capability_info を呼び出して異なる機能のセットを明示的にネゴシエートできます。

- 要求および応答機能のデフォルト・セットのリストについては、「機能」(22 ページ) を参照してください。

注意　応答機能は、クライアントが受信したくない応答の種類を示します。

- Open Server には、プラットフォーム機能とプロトコル機能の 2 つの機能があります。srv_capability ルーチンは、プラットフォーム機能に関するものです。srv_capability_info ルーチンは、プロトコル機能に関するものです。srv_capability の詳細については、srv_capability を参照してください。

参照　srv_capability、srv_props、「機能」(22 ページ)、「プロパティ」(130 ページ)

srv_createmsgq

説明　メッセージ・キューを作成します。

構文

```
CS_RETCODE srv_createmsgq(msgqnamep, msgq_namelen,
        msgqidp)
CS_CHAR      *msgqnamep;
CS_INT       msgqname_len;
SRV_OBJID    *msgqidp;
```

パラメータ

msgqnamep

作成するキューの名前を指すポインタです。既存のキューを作成しようとするとエラーが発生します。

msgqname_len

**msgqnamep* 内の名前の長さです。名前が null で終了している場合、*msgqname_len* は CS_NULLTERM とすることもできます。メッセージ・キューの最大長は、SRV_MAXNAME の文字の長さです。

msgqidp

Open Server は、新しく作成されたメッセージ・キューの ID を **msgqidp* に返します。

戻り値

表 3-24: 戻り値 (srv_createmsgq)

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

```
#include <ospublic.h>

/*
** Local Prototype
*/
CS_RETCODE      ex_srv_createmsgq PROTOTYPE((
     SRV_OBJID     *msgqp,
     CS_CHAR       *msgqnm
));

/*
** EX_SRV_CREATEMSGQ
**
**     Example routine to create an Open Server message queue
 **     using srv_createmsgq.
**
** Arguments:
**     msgqp   Return pointer to the created message queue
 **             identifier.
**     msgqn   Null terminated name for the created queue.
**
 ** Returns:
**  CS_SUCCEED  Message queue with given name successfully
                created.
**  CS_FAIL     An error was detected.
*/
CS_RETCODE            ex_srv_createmsgq(msgqp, msgqnm)
SRV_OBJID             *msgqp;
CS_CHAR               *msgqnm;
{
     /* Check parameters. */
     if ((CS_INT)strlen(msgqnm) > SRV_MAXNAME)
     {
          return(CS_FAIL);
     }

     /* Create the message queue. */
     if (srv_createmsgq(msgqnm, (CS_INT)CS_NULLTERM, msgqp) !=
           CS_SUCCEED)
     {
          return(CS_FAIL);
     }
     return(CS_SUCCEED);
}
```

使用法

- メッセージ・キューを作成する場合は、メッセージ・キューに名前を付けます。メッセージ・キューが作成されると、名前と ID のどちらでもメッセージ・キューを参照できます。
- メッセージ・キュー ID がわかっている場合は、srv_getobjname を使って名前を検索してください。
- SRV_OBJID は CS_INT として定義されています。
- SRV_S_NUMMSGQUEUES サーバ・プロパティは、Open Server アプリケーションが使用できるメッセージ・キューの数を決定します。「サーバ・プロパティ」(132 ページ) を参照してください。
- SRV_S_MSGPOOL サーバ・プロパティは、ランタイムに Open Server アプリケーションが使用できるメッセージの数を決定します。「サーバ・プロパティ」(132 ページ) を参照してください。

参照

srv_deletemsgq、srv_getmsgq、srv_getobjname、srv_putmsgq

srv_createmutex

説明

相互排他セマフォを作成します。

構文

```
CS_RETCODE srv_createmutex(mutex_namep, mutex_namelen,
        mutex_idp)
CS_CHAR         *mutex_namep;
CS_INT          mutex_namelen;
SRV_OBJID       *mutex_idp;
```

パラメータ

mutex_namep

作成するミューテックスの名前を指すポインタです。

mutex_namelen

**mutex_namep* 内の名前の長さです。文字列が NULL で終了する場合、*mutex_namelen* は CS_NULLTERM とすることもできます。

mutex_idp

Open Server は、新しいミューテックスの ID を **mutex_idp* に返します。

戻り値

表 *3-25:* 戻り値 *(srv_createmutex)*

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

```
#include     <ospublic.h>
/*
 ** Local Prototype.
 */
CS_RETCODE               ex_srv_createmutex PROTOTYPE((
CS_CHAR           *name,
CS_INT            namelen,
SRV_OBJID         *idp
));

/*
** EX_SRV_CREATEMUTEX
**
**     Example routine to create an Open Server mutex.
**
** Arguments:
**
**     name      The name of the mutex to create.
**     namelen   The length of name.
**     idp       The address of a SRV_OBJID, which will be set
 **               to the unique identifier for the created mutex.
**
 ** Returns:
**          CS_SUCCEED      The mutex was created successfuly.
**          CS_FAIL         An error was detected.
*/
CS_RETCODE           ex_srv_createmutex(name, namelen, idp)
CS_CHAR              *name;
CS_INT               namelen;
SRV_OBJID            *idp;
{
     /*
     ** Call the Open Server routine that will create
      ** the mutex.
     */
     if( srv_createmutex(name, namelen, idp) == CS_FAIL )
     {
          /*
          ** An error was al&ready raised.
          */
          return CS_FAIL;
     }

     /*
     ** All done.
     */
     return CS_SUCCEED;
}
```

使用法

- ミューテックスを作成する場合は、ミューテックスに名前を付けます。ミューテックスが作成されると、名前と ID のどちらでもミューテックスを参照できます。
- ミューテックスの ID がわかっている場合は、srv_getobjname を使って名前を検索します。
- ミューテックスを作成しても、作成者にロック権が付与されるわけではありません。作成したミューテックスをロックするには、srv_lockmutex を使用してください。
- SRV_OBJID は CS_INT として定義されています。

参照

srv_deletemutex、srv_getobjname、srv_lockmutex、srv_unlockmutex

srv_createproc

説明

非クライアントのイベント駆動型スレッドを作成します。

構文

```
SRV_PROC      *srv_createproc(ssp)
SRV_SERVER    *ssp;
```

パラメータ

ssp

Open Server ステータス情報の制御構造体へのポインタです。

戻り値

成功した場合は、srv_createproc によって新しいスレッド制御構造体にポインタが返されます。失敗した場合は、srv_createproc によって NULL スレッド・ポインタが返され、Open Server はエラーとなります。

表 3-26: 戻り値 (srv_createproc)

戻り値	意味
新しいスレッド制御構造体へのポインタ	Open Server はスレッドを作成した。
null スレッド・ポインタ	Open Server はスレッドを作成できなかった。 Open Server はエラーになる。

例

```
#include          <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE              ex_srv_creatp     PROTOTYPE((
SRV_SERVER              *ssp,
SRV_PROC                *newsp
));
/*
** EX_SRV_CREATP
```

```
**     Example routine to create a non-client, event driven
 **     thread.
**
** Arguments:
**
**     ssp     A pointer to the Open Server state information
 **             control structure.
**    newsp    A pointer that will be returned by srv_createproc
 **             and point to the new thread control structure.
**
** Returns
**
**     CS_SUCCEED            Thread was created.
**     CS_FAIL               An error was detected.
**
 */
CS_RETCODE             ex_srv_creatp(ssp, newsp)
SRV_SERVER             *ssp;
SRV_PROC               *newsp;
{
     /* Check arguments. */
     if(ssp == (SRV_SERVER *)0)
          return(CS_FAIL);

     /*
     ** Create the new thread
     */
     newsp = srv_createproc(ssp);
     if(newsp == (SRV_PROC *)NULL)
          return(CS_FAIL);
     return(CS_SUCCEED);
}
```

使用法

- srv_createproc は、srv_event または srv_event_deferred から発生するユーザ定義イベントによって駆動される、スレッドを作成します。
- 非クライアント・スレッドは、ユーザ定義イベントのみを受け取り、クライアント生成イベントを受け取りません。
- srv_createproc で作成されたスレッドを終了するには、srv_termproc を使用してください。
- 非クライアント・スレッドにはクライアント I/O がありません。非クライアント・スレッドの場合は、property 引数を (SRV_T_IODEAD) に設定して srv_thread_props を呼び出すと、常に CS_FALSE が返ります。

参照

srv_event、srv_event_deferred、srv_spawn、srv_termproc、srv_thread_props

srv_cursor_props

説明

現在のカーソルに関する情報を取得または設定します。

構文

```
CS_RETCODE srv_cursor_props(spp, cmd, cdp)
SRV_PROC        *spp;
CS_INT          cmd;
SRV_CURDESC     *cdp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

cmd

srv_cursor_props がカーソル情報をクライアントに送信するのか、クライアントからカーソル情報を取得するのかを示します。次の表に、*cmd* の有効値を示します。

表 3-27: *cmd* の値 (*srv_cursor_props*)

値	説明
CS_SET	srv_cursor_props が現在のカーソルに関する情報をクライアントに送信する。
CS_GET	srv_cursor_props が現在のカーソル・コマンドに関する情報をクライアントから取得する。

cdp

SRV_CURDESC 構造体へのポインタです。アプリケーションがカーソル情報を設定する場合は、SRV_CURDESC 構造体が現在のカーソルを記述します。アプリケーションが情報を取得する場合は、Open Server は、現在のカーソルに関する情報を使って SRV_CURDESC 構造体を更新します。現在のカーソル・コマンドによって、さまざまなフィールドがそれぞれ異なった時点で設定されたり、データが取り込まれたりします。*cdp* の各フィールドの説明と、それぞれのフィールドにいつどのようにデータが取り込まれるかについては、「SRV_CURDESC 構造体」(59 ページ) を参照してください。

戻り値

表 3-28: 戻り値 (*srv_cursor_props*)

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

```
#include          <ospublic.h>
/*
 ** Local Prototype.
 */
extern CS_RETCODE     ex_srv_cursor_props PROTOTYPE((
CS_VOID               *spp
));
/*
 ** EX_SRV_CURSOR_PROPS
```

```
**
**     Example routine to retrieve information on the current
 **     cursor.
** Arguments:
**  spp  Apointer to an internal control structure.
**
 ** Returns:
**
**  CS_SUCCEED     Cursor information was retrieved successfully.
**  CS_FAIL        An error was detected.
*/
CS_RETCODE          ex_srv_cursor_props(spp)
SRV_PROC            *spp;
{
     SRV_CURDESC    curdesc;

     if(srv_cursor_props(spp, CS_GET, &curdesc) == CS_FAIL)
     {
          return(CS_FAIL);
     }
     return(CS_SUCCEED);
}
```

使用法

- Open Server アプリケーションは、クライアントとのアクティブ・カーソル情報の交換に srv_cursor_props を使用します。
- 常に、クライアントはカーソル・コマンドを発行することによって、この交換を開始します。つまり、現在のカーソルを指定するのはクライアントです。
- アプリケーションが srv_cursor_props を呼び出せるのは、SRV_CURSOR イベント・ハンドラ内からのみです。
- Open Server はクライアントから受け取った各カーソル・コマンドでそれぞれ SRV_CURSOR イベントを生成します。これを受けて、アプリケーションの SRV_CURSOR イベント・ハンドラは、*cmd* を CS_GET に設定して srv_cursor_props を呼び出すことによって、現在のカーソルと受けたカーソル・コマンドの種類を特定することができます。そうした場合、どのように応答するかを決めることができます。有効なカーソル・コマンドの種類と有効な応答の説明については、「カーソル」(57 ページ) を参照してください。
- 各カーソル・コマンドは、Open Server アプリケーションから独自の応答を引き出します。アプリケーションは SRV_CURDESC 構造体から情報 (要求されたフェッチ・カウントなど) を抜き出し、そのデータに基づいて決定を下し、構造体で情報を設定し、srv_cursor_props を使用して情報をクライアントに返します。状況によっては、アプリケーションはパラメータを読み込んだり、結果ローやパラメータを返したりすることもできます。

- SRV_CURSOR イベント・ハンドラは、カーソル情報コマンドを返すことによって、フェッチ、更新、削除を除く、すべてのカーソル・コマンドに応答しなければなりません。ハンドラは SRV_CURDESC 構造体の *curcmd* フィールドを CS_CURSOR_INFO に設定し、cmd を CS_SET に設定して srv_cursor_props を呼び出します。これが、ハンドラが返す最初の情報です。
- CURSOR_DECLARE コマンドに応答して、Open Server アプリケーションは、現在のカーソルをユニークに識別できるカーソル ID を選択します。アプリケーションは、*cmd* を CS_SET に設定した状態で srv_cursor_props を呼び出すことによって、カーソル ID をクライアントに返します。それ以降、クライアントおよび Open Server アプリケーションは、現在のカーソルを名前ではなく ID を使って参照します。

参照　srv_bind、srv_descfmt、srv_numparams、srv_xferdata、「カーソル」(57 ページ)

srv_dbg_stack

説明　スレッドのコール・スタックを表示します。

構文

```
CS_RETCODE srv_dbg_stack(spp, depth, funcp)
SRV_PROC        *spp;
CS_INT          depth;
CS_RETCODE      (*funcp)();
```

パラメータ

spp
内部スレッド制御構造体へのポインタです。

depth
表示するコール・スタック・レベルの最大数です。*depth* が 1 の場合は、すべてのレベルが表示されます。

funcp
コール・スタック表示の各ラインを処理するためにユーザが提供する関数へのポインタです。この関数を呼び出す場合は、null で終了する文字列、および文字列の長さである整数を指定します。文字列には、プログラム・カウンタとルーチンのパラメータが 16 進数形式で格納されています。この関数から CS_FAIL が返された場合は、スタック・トレースが終了します。CS_FAIL 以外が返された場合は、コール・スタックのすべてのルーチンが処理されるか、*depth* スタック・フレームが処理されるまで、スタック・トレースが続きます。*funcp* が NULL の場合は、Open Server はコール・スタックの内容を *stderr* に書き込みます。

次に、一般的な例を示します。

```
CS_RETCODE callstack_display(linebuf, length)
 CS_CHAR  *linebuf;
 CS_INT   length;
 {
      /*
      ** Output each line of the stack trace to stderr.
      */
      fprintf(stderr,"%s¥n", linebuf);
      return(CS_SUCCEED);
 }
```

戻り値

表 3-29: 戻り値 (srv_dbg_stack)

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

```
    #include <ospublic.h>
    /*
    ** Local prototype.
    */
    CS_RETCODE      ex_srv_dbg_stack PROTOTYPE((
    SRV_PROC        *spp
    ));

    /*
    ** EX_SRV_DBG_STACK
    **
    **     Example routine to display the call stack of a thread.
    **
    ** Arguments:
    **     spp  -  A pointer to an internal thread control structure.
    **
     ** Returns:
    **     CS_SUCCEED    Call stack successfully displayed.
    **     CS_FAIL       An error was detected.
    **
    */
    CS_RETCODE            ex_srv_dbg_stack(spp)
    SRV_PROC       *spp;
    {
         CS_RETCODE retval;

         retval = srv_dbg_stack(spp, -1, (CS_RETCODE(*)())NULL);

         return (retval);
    }
```

使用法

- srv_dbg_stack は、すべてのプラットフォームで使用できるわけではありません。srv_capability を使って、現在のプラットフォームで使用可能かどうかを調べてください。
- srv_dbg_stack を使用すると、デバッグ中、または実行エラーの処理中に、スレッドのコール・スタックを検査することができます。srv_dbg_stack は、デバッガ、または実行中のアプリケーションから呼び出すことができます。
- 通常、srv_dbg_stack は、重大なエラーが生じたときにスタック・フレームをエラー・ログに記録するために使用します。
- 呼び出しスタックの各ルーチンは、16 進数のプログラム・カウンタ、続いて同じく 16 進数の各パラメータで構成される文字列にフォーマットされます。プログラム・カウンタを関数名に変換するには、実行プログラムのロード・マップが必要です。
- 現在実行しているスレッドのスタックを表示するように呼び出すと、srv_dbg_stack およびそのコール・ルーチンはスタック上に表示されます。

参照

srv_capability、srv_dbg_switch

srv_dbg_switch

説明

デバッグのために、もう 1 つのスレッド・コンテキストを一時的にリストアします。

構文

```
CS_RETCODE srv_dbg_switch(spid)
CS_INT          spid;
```

パラメータ

SPID

コンテキストが一時的にリストアするスレッドのサーバ・プロセス ID (spid) です。

戻り値

表 3-30: 戻り値 (srv_dbg_switch)

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。

使用法

- srv_dbg_switch は、すべてのプラットフォームで使用できるわけではありません。srv_capability を使って、プラットフォームで srv_dbg_switch をサポートしているかどうかを調べてください。
- スレッド・コンテキストが切り替えられると、アプリケーションを実行し続けた場合、元のスレッド・コンテキストがリストアされ、アプリケーションは正常に実行し続けます。

- コンテキストがリストアされているスレッドは、実行できません。この場合、スレッドは調べる対象にしかなりません。
- UNIX システムの場合、システム・サービス・ルーチン内から srv_dbg_switch を呼び出さないでください。呼び出した場合は、SIGTRAP シグナルが発行され、プログラムは終了します。
- *spid* を取得する場合は、property 引数を SRV_T_SPID に設定して srv_thread_props を呼び出します。現在実行しているスレッドのコンテキストをリストアしようとすると、エラーになります。

参照 srv_capability、srv_dbg_stack

srv_define_event

説明 ユーザ・イベントを定義します。

構文

```
int srv_define_event(ssp, type, namep, namelen)
SRV_SERVER     *ssp;
CS_INT          type;
CS_CHAR        *namep;
CS_INT          namelen;
```

パラメータ

ssp

Open Server の制御構造体へのポインタです。

type

イベントのタイプです。現在、ユーザ定義イベントは SRV_EQUEUED のタイプでなければなりません。

namep

イベント名へのポインタです。

namelen

**namep* の文字列の長さをバイト数で示したものです。文字列が null で終了する場合、*namelen* は CS_NULLTERM とすることもできます。

戻り値

表 *3-31:* 戻り値 *(srv_define_event)*

戻り値	意味
0 でない整数	イベント用のユニークな id。
0	Open Server はイベントを定義できない。 Open Server はエラーになる。

例

```
#include          <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE      ex_srv_define_event PROTOTYPE((
CS_CHAR        *namep,
CS_INT         namelen,
CS_INT         *event_no
));
/*
 ** EX_SRV_DEFINE_EVENT
**
**   Example routine to illustrate the use of srv_define_event to
 **   define an user event.
**
** Arguments:
**     namep   A pointer to the name of event.
**     namelen The length, in bytes, of string in *namep.
**     event_no  A CS_INT pointer that is initialized with
 **               the unique number for the event.
 ** Returns:
**
**  CS_SUCCEED   If the event was defined successfully.
**  CS_FAIL      An error was detected.
 */
CS_RETCODE     ex_srv_define_event(namep, namelen, event_no)
CS_CONTEXT *cp;
 CS_VOID        *bufp;
 CS_CHAR        *namep;
 CS_INT         namelen;
 CS_INT         *event_no;
 CS_INT         result;
{
     SRV_PROC *srvproc_ptr; /* A pointer to an internal thread
                             ** control structure */

     result = srv_props(cp, CS_GET, SRV_S_CURTHREAD,
            bufp, sizeof(CS_INT));
     if (result == CS_FAIL)
      {
          return (CS_FAIL);
      }
     /* Now define the event. */
     if ((*event_no = srv_define_event(srvproc_ptr, SRV_EQUEUED,
           namep, namelen)) == (CS_INT)0)
          return (CS_FAIL);
     return (CS_SUCCEED);
}
```

使用法

- ユーザ定義イベントは、クライアント・アクションによってトリガされるのではなく、srv_event を呼び出すことによってトリガされます。Open Server プログラマは、イベントがトリガされたときに実行するハンドラ・ルーチンを提供します。
- ユーザ定義イベントのイベント・ハンドラは、通常、srv_handle を使ってインストールします。
- ユーザ定義イベントのハンドラは、イベントを受けたスレッドのスレッド制御構造体へのポインタを受け取ります。
- ユーザ定義イベントが許可されるように Open Server アプリケーションを設定していない場合は、イベントを定義できません。詳細については、srv_props のページを参照してください。

参照

srv_event、srv_event_deferred、srv_handle、srv_props、「イベント」(84 ページ)

srv_deletemsgq

説明

メッセージ・キューを削除します。

構文

```
CS_RETCODE srv_deletemsgq(msgqnamep, msgqname_len,
        msgqid)
CS_CHAR     *msgqnamep;
CS_INT      msgqname_len;
SRV_OBJID   msgqid;
```

パラメータ

msgqnamep

削除するメッセージ・キュー名へのポインタです。存在しないメッセージ・キューを削除しようとすると、エラーになります。

msgqname_len

msgqname によって示される、名前の長さです。名前が NULL で終了する場合、*msgqname_len* は CS_NULLTERM とすることもできます。

msgqid

削除するメッセージ・キューの識別子を指定する SRV_OBJID です。

戻り値

表 3-32: 戻り値 *(srv_deletemsgq)*

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

```
#include          <ospublic.h>
/*
 ** Local Prototype.
 */
CS_RETCODE      ex_srv_deletemsgq PROTOTYPE((
CS_CHAR         *msgqname,
CS_INT          msgqname_len,
SRV_OBJID msgqid
));
/*
 ** EX_SRV_DELETEMSGQ
**
**     Example routine using srv_deletemsgq to delete an Open
 **     Server message queue previously create by srv_createmsgq.
** This routine can be passed a value message queue name, or
 ** NULL,in which case the message queue identifier will be used.
**
** Arguments:
**  msgqname        The name of the message queue to delete. If
 **                  NULL, the msgqid is used.
 **  msgqname_len    The length of the name to which msgqname
 **                  points.
 **  msgqid          A SRV_OBJID that specifies the identifier of
 **                  the message queue to delete.
 **
 ** Returns:
**
**    CS_SUCCEED    The message queue was successfully deleted.
**    CS_FAIL       An error was detected.
 */
CS_RETCODE      ex_srv_deletemsgq(msgqname, msgqname_len, msgqid)
CS_CHAR         *msgqname;
CS_INT          msgqname_len;
SRV_OBJID       msgqid;
{
     /*
      ** Delete a message queue.
      */
     if (srv_deletemsgq(msgqname, msgqname_len, msgqid) !=
           CS_SUCCEED)
     {
          return(CS_FAIL);
     }
     return(CS_SUCCEED);
}
```

使用法

- メッセージ・キューを削除する場合は、名前または ID を使用します。*msgqname* が NULL でない場合は、メッセージ・キュー名を使用します。そうでない場合は、メッセージ・キュー ID を使用します。
- キュー内の未読メッセージは、キューが削除される前にフラッシュされます。srv_putmsgq で待機しているスレッドがウェイクアップし、srv_putmsgq は CS_FAIL を返します。
- メッセージ・キューが削除されると、そのキューからのメッセージを待っているスレッドは srv_getmsgq からの戻り値 CS_FAIL でウェイクアップし、srv_getmsgq の *infop* 引数は SRV_I_DELETED に設定されます。
- SRV_S_NUMMSGQUEUES サーバ・プロパティは、Open Server アプリケーションが使用できるメッセージ・キューの数を決定します。「サーバ・プロパティ」(132 ページ) を参照してください。
- SRV_S_MSGPOOL サーバ・プロパティは、ランタイムに Open Server アプリケーションが使用できるメッセージの数を決定します。「サーバ・プロパティ」(132 ページ) を参照してください。

参照

srv_createmsgq、srv_getmsgq、srv_getobjname、srv_putmsgq

srv_deletemutex

説明

srv_createmutex を使用して作成したミューテックスを削除します。

構文

```
CS_RETCODE srv_deletemutex(mutex_namep, mutex_namelen,
        mutex_id)
CS_CHAR     *mutex_namep;
CS_INT      mutex_namelen;
SRV_OBJID   mutex_id;
```

パラメータ

mutex_namep

ミューテックス作成時にミューテックスに関連付けられた、名前へのポインタです。

mutex_namelen

mutex_namep の長さをバイト数で示したものです。文字列が NULL で終了する場合、*mutex_namelen* は CS_NULLTERM とすることもできます。

mutex_id

srv_createmutex が返すユニークな識別子です。

戻り値

表 *3-33:* 戻り値 *(srv_deletemutex)*

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

```
#include <ospublic.h>
/*
 ** Local Prototype.
 */
CS_RETCODE    ex_srv_deletemutex PROTOTYPE((
CS_CHAR         *mtxnm,
SRV_OBJID       mtxid
));
/*
 ** EX_SRV_DELETEMUTEX
**     Example routine using srv_deletemutex to delete an
 **     Open Server mutex previously created by srv_createmutex.
 **     This routine can be passed a mutex name, or NULL,
 **     in which case the mutex identifier will be used.
** Arguments:
**     mtxnm     Null terminated mutex name, or NULL to use mutex
 **               id.
 **    mtxid      Mutex identifier (valid only if mtxnm is NULL).
 ** Returns:
**   CS_SUCCEED     mutex was successfully queued for deletion.
 **   CS_FAIL        An error was detected.
 */
CS_RETCODE              ex_srv_deletemutex(mtxnm, mtxid)
CS_CHAR                 *mtxnm;
SRV_OBJID               mtxid;
{
     /* Delete the mutex. */
      if (srv_deletemutex(mtxnm, (CS_INT)CS_NULLTERM, mtxid) !=
           CS_SUCCEED)
     {
           return(CS_FAIL);
      }
      return(CS_SUCCEED);
}
```

使用法

- ミューテックスを削除する場合は、名前または ID を使用します。*mutex_namep* が NULL でない場合は名前を、そうでない場合は ID を使用します。
- 他のスレッドがミューテックスをロックしようとして待機している場合は、そのスレッドの要求が満たされてロックが解放されるまでは、ミューテックスは削除されません。
- ミューテックスの使用例が、srv_createmutex のページに示されています。

参照

srv_createmutex、srv_getobjid、srv_getobjname、srv_lockmutex

srv_descfmt

説明

クライアントとやり取りするカラムまたはパラメータの記述を、記述または取得します。

構文

```
CS_RETCODE srv_descfmt(spp, cmd, type, item,
        clfmtp)
SRV_PROC      *spp;
CS_INT        cmd;
CS_INT        type;
CS_INT        item;
CS_DATAFMT    *clfmtp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

cmd

srv_descfmt はクライアントに送られるデータを記述するのか、クライアントから受けるデータの記述を取得するのかを示します。表 3-34 に、*cmd* の有効値を示します。

表 *3-34: cmd* の値 *(srv_descfmt)*

値	説明
CS_SET	srv_descfmt は、クライアントがデータを受け取るときのフォーマットを記述する。
CS_GET	srv_descfmt は、クライアントがデータを送ったときのフォーマットを取得する。

type

cmd が CS_SET の場合は、記述されているデータの型です。*cmd* が CS_GET の場合は、取得されているデータの型です。表 3-35 に、有効な型とその適切なコンテキストを示します。

表 *3-35: type* の値 *(srv_descfmt)*

型	cmd の許容される設定	説明
SRV_RPCDATA	CS_SET または CS_GET	RPC またはストアド・プロシージャ・パラメータ
SRV_ROWDATA	CS_SET のみ	ローのデータ
SRV_CURDATA	CS_GET のみ	カーソル・パラメータ
SRV_UPCOLDATA	CS_GET のみ	カーソル更新カラム
SRV_KEYDATA	CS_GET のみ	カーソル・キー・データ
SRV_ERRORDATA	CS_SET のみ	拡張エラー・データ
SRV_DYNDATA	CS_SET または CS_GET	動的 SQL データ
SRV_NEGDATA	CS_SET または CS_GET	ネゴシエーション・ログイン・データ
SRV_MSGDATA	CS_SET または CS_GET	MSG パラメータ
SRV_LANGDATA	CS_GET のみ	言語パラメータ

item
パラメータまたはカラムの番号です。パラメータやカラムの番号は 1 から始まります。

clfmtp
データの記述が格納されている CS_DATAFMT 構造体へのポインタです。

戻り値

表 *3-36:* 戻り値 *(srv_descfmt)*

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

```
#include    <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE      ex_srv_descfmt PROTOTYPE((
SRV_PROC        *spp,
CS_INT          item,
CS_DATAFMT      *dp
));

/*
** EX_SRV_DESCFMT
**
**     Example routine used to get an RPC parameter description.
**
** Arguments:
**
**     spp          A pointer to an internal thread control
                     structure.
**     item         The parameter number we're looking for.
**     dp           The address of a CS_DATAFMT to be filled with
 **                  the parameter's description.
**
 ** Returns:
**          CS_SUCCEED if the description was obtained, or
 **          CS_FAIL if an error was detected.
*/
CS_RETCODE      ex_srv_descfmt(sp, item, dp)
SRV_PROC        *sp;
CS_INT          item;
CS_DATAFMT      *dp;
{
     /*
     ** Call srv_descfmt to get the RPC parameter description.
     */
     if( srv_descfmt(sp, CS_GET, SRV_RPCDATA, item, dp) ==
           CS_FAIL )
```

```
        {
             /*
             ** An error was al&ready raised.
             */
             return CS_FAIL;
        }

        /*
        ** All done.
        */
        return CS_SUCCEED;
    }
```

使用法

- srv_descfmt は、さまざまなカラムやパラメータのフォーマットを記述します。詳細については、「CS_DATAFMT 構造体」(48 ページ) を参照してください。
- クライアントにローやパラメータを送信する場合 (CS_SET)、クライアントへのデータの表示方法を記述するために srv_descfmt を呼び出します。クライアントからパラメータを受け取る場合 (CS_GET)、クライアントがデータを送信したときのフォーマットの記述を取得するために srv_descfmt を呼び出します。このクライアントのフォーマット情報については、リモート・サーバに渡せるように、ゲートウェイ・アプリケーションが保存する場合があります。
- srv_descfmt ルーチンは、次の表に示す CS_DATAFMT フィールドに対して、読み取り (CS_GET) や設定 (CS_SET) を行います。他のすべてのフィールドは、srv_descfmt には未定義です (“clfmtp” は構造体を指すポインタであることに注意してください)。

表 3-37: 使用される *CS_DATAFMT* フィールド *(srv_descfmt)*

フィールド	CS_SET	CS_GET
clfmtp→ namelen	名前の長さ	名前の長さ
clfmtp→ status	パラメータ／カラム・ステータス	パラメータ・ステータス
clfmtp→ name	パラメータ／カラム名	パラメータ名
clfmtp→ datatype	リモート・データ型をここに設定	リモート・データ型をここから取得
clfmtp→ maxlength	リモート・データ型の最大長をここに設定	リモート・データ型の最大長をここから取得
clfmtp→ format	リモート・データ型のフォーマット	リモート・データ型のフォーマット

- CS_DATAFMT 構造体で記述されたフォーマット (*clfmtp*) が、その後に続く srv_bind の呼び出しで記述されるフォーマット (*osfmtp*) と異なる場合、Open Server は、*cmd* が CS_SET のときにはクライアント・フォーマット (*clfmtp*) に、*cmd* が CS_GET のときにはアプリケーション・フォーマット (*osfmtp*) に、自動的に変換されます。

- データ・ストリームの各カラムやパラメータが一度記述されバインドされたら、プログラム変数のデータをクライアントに送ったり、クライアントからのデータでプログラム変数を更新したりするために、srv_xferdata を呼び出してください。
- srv_negotiate が正常に返った後であれば、ネゴシエーション・ログイン・オペレーションの一部として、SRV_NEGDATA パラメータを送信したり受信したりできます。
- キー・カラム番号は、ローの番号に対応します。

参照

srv_bind、srv_cursor_props、srv_dynamic、srv_msg, srv_negotiate、srv_numparams、srv_sendinfo、srv_xferdata、「CS_DATAFMT 構造体」(48 ページ)

srv_dynamic

説明

クライアントの動的 SQL コマンドを読み込む、またはコマンドに応答します。

構文

```
CS_RETCODE srv_dynamic(spp, cmd, item, bufp,
                       buflen, outlenp)
SRV_PROC      *spp;
CS_INT         cmd;
CS_INT         item;
CS_VOID        *bufp
CS_INT         buflen;
CS_INT        *outlenp
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

cmd

動的コマンドがクライアントから読み出されているのか、クライアントに送られているのかを示します。表 3-38 に、*cmd* の有効値を示します。

表 *3-38: cmd* の値 *(srv_dynamic)*

値	説明
CS_SET	srv_dynamic は、動的コマンドの応答をクライアントに返す。
CS_GET	srv_dynamic は、動的コマンドをクライアントから読み出す。

item

どのような情報が送信または取得されているのかを示します。表 3-39 に、*item* の有効値を示します。

表 3-39: item の値 (srv_dynamic)

値	意味
SRV_DYN_TYPE	実行されている動的オペレーションの種類
SRV_DYN_IDLEN	動的文の ID の長さ
SRV_DYN_ID	動的文の ID
SRV_DYN_STMTLEN	動的文の長さ
SRV_DYN_STMT	準備または実行されている動的文

bufp

item 値が返される (CS_GET) か、設定される (CS_SET) バッファへのポインタです。

buflen

**bufp* バッファのバイト単位の長さです。表 3-40 に、必要なバッファ・サイズを示します。

表 3-40: 必要なバッファ・サイズ (srv_dynamic)

値	要求されるフォーマット (サイズ)
SRV_DYN_TYPE	(CS_INT) のサイズ
SRV_DYN_IDLEN	(CS_INT) のサイズ
SRV_DYN_ID	可変。まず item を CS_DYN_IDLEN に設定して srv_dynamic を呼び出して長さを確定し、それに合ったバッファ・サイズを割り付ける。
SRV_DYN_STMTLEN	(CS_INT) のサイズ
SRV_DYN_STMT	可変。まず item を CS_DYN_STMTLEN に設定して srv_dynamic を呼び出して長さを確定し、それに合ったバッファ・サイズを割り付ける。

outlenp

クライアントからデータを取得しているときに (*cmd* は CS_GET) **bufp* にコピーされたデータの実際の長さに設定された整数変数へのポインタです。*cmd* が CS_SET の場合には、この引数は必要ではありません。

戻り値

表 3-41: 戻り値 (srv_dynamic)

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

```
#include        <ospublic.h>
/*
 ** Local Prototype
 */
extern CS_RETCODE               ex_srv_dynamic PROTOTYPE((
CS_VOID     *spp,
CS_INT      *optypep
));
/*
 ** EX_SRV_DYNAMIC
**
**    Example routine to retrieve dynamic operation type from a
 **    client.
**
** Arguments:
**    spp      Thread control structure.
**    optypep  Dynamic operation type.
**
 ** Returns:
**
**    CS_SUCCEED   Dynamic information was retrieved
 **                  successfully.
**    CS_FAIL      An error was detected.
 */
CS_RETCODE      ex_srv_dynamic(spp, optypep)
SRV_PROC   *spp;
CS_INT     *optypep;
{
CS_INT     outlen;

    if(srv_dynamic(spp, CS_GET, SRV_DYN_TYPE, optypep,
         sizeof(*optypep), &outlen) == CS_FAIL)
    {
        return(CS_FAIL);
    }
    return(CS_SUCCEED);
}
```

使用法

- srv_dynamic ルーチンでは、Open Server アプリケーションが動的 SQL コマンドを読み込んだり、応答を送ることができます。
- 有効なオペレーションの種類 (SRV_DYN_TYPE) は、次に示すとおりです。

  CS_PREPARE － 文を準備します (CS_GET のみ)。

  CS_DESCRIBE_INPUT － 現在の準備文に使用する入力パラメータ・フォーマットを要求します (CS_GET のみ)。

  CS_DESCRIBE_OUTPUT － 現在の準備文に使用するカラム・フォーマットを要求します (CS_GET のみ)。

CS_EXECUTE － 準備文を実行します (CS_GET のみ)。

CS_EXEC_IMMEDIATE － パラメータがなく結果を返さない、準備されていない文を実行します (CS_GET のみ)。

CS_DEALLOC － 準備文の割り付けを解除します (CS_GET のみ)。

CS_ACK － クライアントからの動的 SQL コマンドに応答します (CS_SET のみ)。

- クライアントから受け取る動的コマンドはすべて、SRV_DYNAMIC イベントをトリガします。その時点で、Open Server アプリケーションは、各クライアント動的コマンドに応答してオペレーション・タイプ、文の ID および文を取得し保存するために srv_dynamic を呼び出すことが可能になり、*type* を CS_ACK に設定して srv_dynamic 呼び出しを発行することによって、クライアント通信に応答することができます。
- SRV_DYNAMIC ハンドラ以外の、いかなるハンドラにおいても srv_dynamic を呼び出すとエラーが発生します。
- CS_ACK が、唯一設定可能な動的オペレーション・タイプです (*cmd* を CS_SET に設定)。
- CS_PREPARE、CS_DESCRIBE_INPUT、CS_DESCRIBE_OUTPUT、CS_EXECUTE、CS_EXEC_IMMEDIATE、そして CS_DEALLOC のみが、取得可能な動的オペレーション・タイプです (*cmd* を CS_GET に設定)。
- 完全な動的 SQL 応答をクライアントに送ることは、ID 長、ID、オペレーション・タイプを渡すことによって行われます。このためには、srv_dynamic に対する 3 つの個別の呼び出しが必要です。たとえば、文の ID のみを設定して srv_senddone を呼び出すと、エラーとなります。唯一の例外が、オペレーション・タイプが CS_EXEC_IMMEDIATE の場合で、これには関連する文の ID がありません。
- パラメータ・データ・フォーマットおよび出力フォーマットは、CS_PREPARE 動的コマンドに応答して、type 引数を SRV_DYNDATA にして srv_descfmt と srv_xferdata を使用して、クライアントに送信することができます。これはアプリケーションが単にフォーマットを送信しているだけなので、srv_bind は必要ではないことに注意してください。
- Open Server アプリケーションは、クライアントが CS_EXECUTE 動的コマンドに続いて送信したパラメータ・データを、type 引数を SRV_DYNDATA にして srv_descfmt、srv_bind、srv_xferdata を使用して、取得し保存します。パラメータの数は、アプリケーションが srv_numparams を使用して決めます。
- アプリケーションは、type 引数を SRV_ROWDATA にして srv_descfmt、srv_bind、srv_xferdata を使用して、CS_EXECUTE 動的 SQL コマンドに応答して、クライアントに動的 SQL 結果ローを送ります。

- CS_EXEC_IMMEDIATE の動的 SQL コマンドは、クライアントがパラメータなしで文を実行し、結果として DONE のみを受けることを希望していることを示しています。この文は CS_EXEC_IMMEDIATE コマンド・ストリームに含まれ、SRV_DYN_STMT を介してアクセスすることが可能です。文は前もって準備されておらず (文の ID 長 (SRV_DYN_IDLEN) は 0 になる)、SRV_DYNAMIC イベント・ハンドラが終了すると、存在しなくなります。

参照　srv_bind、srv_descfmt、srv_numparams、srv_xferdata、「動的 SQL」(75 ページ)

srv_envchange

説明　環境の変化をクライアントに通知します。

構文

```
CS_RETCODE srv_envchange(spp, type, oldvalp
        oldvallen, newvalp, newvallen)

SRV_PROC    *spp;
CS_INT       type;
CS_CHAR     *oldvalp;
CS_INT       oldvallen
CS_CHAR     *newvalp;
CS_INT       newvallen
```

パラメータ

spp
内部スレッド制御構造体へのポインタです。

type
変更されている環境です。現在、有効値は SRV_ENVDATABASE と SRV_ENVLANG のみで、それぞれ、現在のデータベース名と各国言語を表します。

oldvalp
古い値が格納されている文字列へのポインタです。NULL の場合もあります。ポインタの長さは、バイト数で *oldvallen* に保存されます。

oldvallen
oldvalp* の文字列の長さをバイト数で示したものです。長さが CS_NULLTERM の場合は、oldvalp* の文字列は NULL で終了します。また、CS_UNUSED の場合は、**oldvalp* の文字列が NULL であることを示しています。

newvalp
環境変数の新しい値が格納されている文字列へのポインタです。NULL の場合もあります。ポインタの長さは、バイト数で *newvallen* に保存されます。

newvallen
newvalp* の文字列の長さをバイト数で示したものです。長さが CS_NULLTERM の場合は、*newvalp* の文字列は NULL で終了します。また、CS_UNUSED の場合は、newvalp* の文字列が NULL であることを示しています。

戻り値

表 *3-42:* 戻り値 *(srv_envchange)*

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

```
#include   <ospublic.h>
/*
 ** Local Prototype.
 */
CS_RETCODE  ex_srv_envchange PROTOTYPE((
SRV_PROC    *spp
));
/*
 ** EX_SRV_ENVCHANGE
**
**    Example routine to notify the client of an environment
 **    change.
**
** Arguments:
**    spp    A pointer to an internal thread control structure.
**
** Returns:
**     CS_SUCCEED    Succesfully notified client of environment
 **                    change.
**     CS_FAIL       An error was detected.
**
*/
CS_RETCODE   ex_srv_envchange(spp)
SRV_PROC     *spp;
{
    CS_RETCODE   retval;
    /*
     ** Notify the client that we've changed the database
     ** from "master" to "pubs2".
     */
    retval = srv_envchange(spp, SRV_ENVDATABASE, "master",
             CS_NULLTERM, "pubs2", CS_NULLTERM);
    return (retval);
}
```

使用法

- さまざまな環境変数を設定できます。Open Server では一部の環境変数が自動的に処理されますが、自動処理されない環境変数については Open Server アプリケーションが処理する必要があります。現行バージョンのアプリケーションにできることは、現在のデータベースや各国言語に関する変更をクライアントに通知することだけです。

- これらの値が変わるたびに、Open Server は Open Server アプリケーションのエラー・ハンドラを呼び出します。Open Server アプリケーションが srv_envchange によってこれを変更することも、Open Server が内部コードを使用して変更することもできます。また、その両方も可能です。エラー・ハンドラに渡されるエラー番号は、これらの値のどれかが変わるときにクライアントに返送される Adaptive Server Enterprise メッセージ番号です。このようにして、クライアントが Open Server に接続していても Adaptive Server Enterprise に接続していても、同じメッセージ番号を変化する値に一致させることができます。表 3-43 にメッセージ番号と、それぞれの変化値に対応する *oserror.h* の #define を示します。

表 3-43: 環境変数 *(srv_envchange)*

変更される値	メッセージ番号	oserror.h の #define
現在のデータベース	5701	SQLSRV_ENVDB
各国言語	5703	SQLSRV_ENVLANG

srv_event

説明

スレッドの要求処理キューに、イベント要求を追加します。

構文

CS_INT srv_event(spp, event, datap)

```
SRV_PROC    *spp;
CS_INT      event;
CS_VOID     *datap;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

イベント

クライアントのイベント・キューに追加するイベント用のトークンです。定義されたイベントのリストについては、「イベント」(84 ページ) を参照してください。

datap

Open Server プログラマによって提供されたデータへのポインタ (CS_VOID) です。アプリケーションは、イベント・ハンドラ内から、property を SRV_T_EVENTDATA に設定して srv_thread_props を呼び出すことによって、データを取得することができます。

戻り値

表 3-44: 戻り値 *(srv_event)*

戻り値	意味
要求されたイベントのためのトークン	Open Server が新しいイベントを追加した。
CS_FAIL	ルーチンが失敗した。

例

```
#include  <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE   ex_srv_event PROTOTYPE((
SRV_PROC     *spp,
CS_INT       event,
CS_VOID     *datap
));
/*
 ** EX_SRV_EVENT
**
**    Example routine to queue an event request to an Open Server
 **    thread's request-handling queue.
**
**    Note that if the event is an user-defined one, it
 **    must have been defined earlier using srv_define_event.
**
** Arguments:
**    spp        A pointer to a control structure for an Open
 **               Server thread.
**    event      The token for the event to be added to the queue.
**    datap      Data pointer.
**
 ** Returns:
**
**    CS_SUCCEED    The event was queued successfully
**    CS_FAIL       An error was detected.
*/
CS_RETCODE  ex_srv_event(spp, event, datap)
SRV_PROC    *spp;
CS_INT      event;
CS_VOID     *datap;
{
    if (srv_event(spp, event, datap) == CS_FAIL)
        return (CS_FAIL);
    else
        return (CS_SUCCEED);
}
```

使用法

- 特定のクライアント・スレッドのイベント・キューに、イベント要求を追加します。通常、イベント要求は、クライアント・アプリケーションからの Client-Library 呼び出しなどで自動的にイベント要求キューに追加されます。ただし、Open Server プログラマは、srv_event を使用して具体的に要求を追加できます。

srv_event を使用して、次のイベントをイベント・キューに追加できます。

- SRV_DISCONNECT
- SRV_URGDISCONNECT
- SRV_STOP
- ユーザ定義のイベント

- srv_handle は、イベント発生時にどのイベント・ハンドラを呼び出すかを Open Server に通知します。特定のイベントについてハンドラが定義されていない場合は、デフォルトの Open Server イベント・ハンドラが呼び出されます。
- SRV_URGDISCONNECT イベントは、Open Server アプリケーションの SRV_DISCONNECT イベント・ハンドラを呼び出します。
- SRV_URGDISCONNECT イベントは、緊急イベントとしてキューイングされます。そのため、切断イベントは、すでにキューイングされているイベントよりも優先され、スレッドのイベント・キューの先頭に配置されます。これは、Open Server スレッドを即刻中止する場合に役立ちます。
- ユーザ定義のイベントの場合は、トリガ可能となる前に、まず srv_define_event で定義する必要があります。
- srv_event は、SRV_STOP または SRV_START 以外のイベントをスレッドのイベント・キューに追加します。SRV_STOP または SRV_START イベントの場合、*spp* は、イベントを要求しているスレッドの内部スレッド制御構造体を指します。
- Open Server アプリケーションは、I/O として機能するルーチンをユーザ定義イベント内からは呼び出せません。

警告！ 割り込みレベルのコードでは、srv_event ではなく srv_event_deferred を使用します。

参照

srv_define_event、srv_handle、srv_event_deferred、srv_thread_props、「イベント」(84 ページ)

srv_event_deferred

説明

非同期イベントの結果として、スレッドのイベント・キューにイベント要求を追加します。

構文

CS_INT srv_event_deferred(spp, event, datap)

```
SRV_PROC    *spp;
CS_INT       event;
CS_VOID     *datap;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

イベント

スレッドのイベント・キューに追加するイベントです。

datap

Open Server プログラマによって提供されたデータへのポインタ (CS_VOID) です。アプリケーションは、イベント・ハンドラ内から、property を SRV_T_EVENTDATA に設定して srv_thread_props を呼び出すことによって、データを取得することができます。

戻り値

要求されたイベントを返します。エラーの場合は、-1 が返されます。

表 *3-45:* 戻り値 *(srv_event_deferred)*

戻り値	意味
要求されたイベントのためのトークン	Open Server が新しいイベントを追加した。
-1	ルーチンが失敗した。

例

```
#include   <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE   ex_srv_event_deferred PROTOTYPE((
SRV_PROC     *spp,
CS_INT       event,
CS_VOID      *datap
));
/*
 ** EX_SRV_EVENT_DEFERRED
**    Example routine to queue up a deferred event using
 **    srv_event_deferred.A deferred event request will
 **    typically be made from within interrupt-level code.
** Arguments:
**    spp       A pointer to the internal thread control
 **              structure.
**    event     The event to add to the thread's queue.
**    datap     A pointer to data to attach to the event.
 ** Returns:
**    CS_SUCCEED    The event was sucecssfully queued.
```

```
**    CS_FAIL       An error was detected.
 */
CS_RETCODE          ex_srv_event_deferred(spp, event, datap)
SRV_PROC            *spp;
CS_INT              event;
CS_VOID             *datap;
{
      /*
      ** Add a deferred event to the event queue.
      */
     if (srv_event_deferred(spp, event, datap) == -1)
     {
            return(CS_FAIL);
      }
     return(CS_SUCCEED);
}
```

使用法

- srv_event_deferred は、たとえば UNIX におけるシグナルの配信などのように、割り込みレベル・コードからスレッドのイベント・キューにイベント要求を追加します。イベント要求は、srv_event_deferred が呼び出されたときに、Open Server の内部的な重要な機能が実行中の場合は、それらがすべて終了するまで遅延されます。
- Open Server アプリケーションによっては、割り込みレベル・コードからイベントを発生させることができなければなりません。たとえば、アテンション・ハンドラ内でイベントを発生させたい場合や、Open Server アプリケーション・コードでアラーム・シグナルを使用している場合は、srv_event の代わりに srv_event_deferred を使用する必要があります。srv_event_deferred は、リンクされたリストの更新や内部的なハウスキーピング実行などの重要な機能が、イベント要求が実行される前に完了することを保証します。

  警告！ 割り込みレベルのコードでは、srv_event ではなく srv_event_deferred を使用します。

- Open Server は通常、スレッドのイベント要求キューに自動的にイベント要求を追加します。ただし、srv_event_deferred で具体的な要求を追加することもできます。
- 次のイベントは、srv_event_deferred でイベント・キューに追加できます。
  - SRV_DISCONNECT
  - SRV_URGDISCONNECT
  - SRV_STOP
  - ユーザ定義のイベント

- srv_handle は、イベント発生時にどのイベント・ハンドラを呼び出すかを Open Server に通知します。特定のイベントについてハンドラが定義されていない場合は、デフォルトの Open Server イベント・ハンドラが呼び出されます。
- ユーザ定義のイベントの場合は、トリガ可能となる前に、まず srv_define_event で 定義する必要があります。
- srv_event は、SRV_STOP または SRV_START 以外のイベントをスレッドのイベント・キューに追加します。SRV_STOP または SRV_START イベントの場合、*spp* は、イベントを要求しているスレッドの内部スレッド制御構造体を指します。
- Open Server アプリケーションは、I/O として機能するルーチンをユーザ定義イベント内からは呼び出せません。

参照 srv_define_event、srv_event、srv_handle、srv_thread_props、「イベント」(84 ページ)

srv_free

説明 割り付けられているメモリを解放します。

構文

```
CS_RETCODE srv_free(mp)
CS_VOID        *mp;
```

パラメータ

mp

解放するメモリを指すポインタです。

戻り値

表 *3-46:* 戻り値 *(srv_free)*

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

```
#include  <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE ex_srv_free PROTOTYPE((
CS_BYTE   *p
));
/*
 ** EX_SRV_FREE
**
**   Example routine to free memory allocated through srv_alloc.
**
** Arguments:
**        p - The address of the memory block to be freed.
```

```
**
 ** Returns:
**
**   CS_SUCCEED      Memory was freed successfully.
**   CS_FAIL         An error was detected.
*/
CS_RETCODE     ex_srv_free(p)
CS_BYTE        *p;
{
    /*
    ** Free the memory block.
    */
    if( srv_free(p) == CS_FAIL )
    {
        return CS_FAIL;
    }
    return CS_SUCCEED;
}
```

使用法

- srv_free は、srv_alloc、srv_init、または srv_realloc で割り付けられたメモリを解放する場合にのみ使用します。
- 現行バージョンでは、srv_free は C ルーチンである free を呼び出します。しかし、Open Server アプリケーションでは、srv_props ルーチンを使用して独自のメモリ管理ルーチンをインストールすることができます。ユーザ・インストールのルーチンのパラメータ転送規則は、free のものと同一でなければなりません。ユーザ・インストールのルーチンを使用できるようにアプリケーションが設定されていない場合は、free を使用してください。

参照　srv_alloc、srv_props、srv_realloc、srv_init

srv_freeserveraddrs

説明　srv_getserverbyname によって割り付けられたメモリを解放します。

構文　CS_RETCODE srv_freeserveraddrs(void **resultptr*)

パラメータ

resultptr

srv_getserverbyname によって返されるメモリへのポインタです。

戻り値

表 3-47: 戻り値 *(srv_freeserveraddrs)*

戻り値	意味
CS_SUCCEED	srv_freeserveraddrs の呼び出しを正常に実行した。
CS_FAIL	*resultptr* が NULL であるか、割り付け解除に失敗した。

参照　srv_getserverbyname、srv_send_ctlinfo

srv_get_text

説明

連続したデータとして、text または image のデータ・ストリームをクライアントから読み込みます。

構文

```
CS_RETCODE srv_get_text(spp, bp, buflen, outlenp)
SRV_PROC    *spp;
CS_BYTE     *bp;
CS_INT       buflen;
CS_INT      *outlenp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

bp

クライアントからのデータが格納されているバッファへのポインタです。

buflen

**bp* ポインタのサイズです。このサイズは、連続して転送される 1 つのデータのまとまりをバイト数で示したものです。

outlenp

**bp* バッファに読み込まれたバイト数が、このパラメータに返されます。

戻り値

表 *3-48:* 戻り値 *(srv_get_text)*

戻り値	意味
CS_SUCCEED	srv_get_text の呼び出しを正常に実行した。
CS_FAIL	ルーチンが失敗した。
CS_END_DATA	Open Server が text または image データ・ストリーム全体を読み込んだ。

例

```
#include        <ospublic.h>
#include        <stdio.h>
/*
 ** Local Prototype
 */
CS_RETCODE    ex_srv_get_text        PROTOTYPE((
SRV_PROC      *spp,
CS_INT        *outlenp,
CS_BYTE       *bbuf
));
/*
 ** EX_SRV_GET_TEXT
 **
**    Example routine to read chunks of text or image datastream
 **    from a client into a buffer and then write it to a disk
 **    file.
**
** Arguments:
**
```

```
**    spp         Pointer to thread control structure.
**    outlenp     Number of bytes read and written.
**    bbuf        Pointer to very large buffer for text.
**
** Returns
**
**    CS_SUCCEED    The data was successfully read.
**    CS_FAIL       An error was detected.
 **
 */
#define    BUFSIZE     256
#define    FPUTS(a,b)      fputs(a,b)
CS_RETCODE  ex_srv_get_text(spp,outlenp,bbuf)
SRV_PROC    *spp;
CS_INT      *outlenp;
CS_BYTE     *bbuf;
{
   CS_INT       llen;    /* Local length. */
    CS_INT       lout;    /* Local read count. */
    CS_RETCODE   lret;    /* Local return code. */
    CS_BYTE      *lbufp;    /* Local pointer into bbuf. */
   /* Check arguments. */
   if(bbuf == (CS_VOID *)0)
      return(CS_FAIL);
   if(spp == (SRV_PROC *)0)
      return(CS_FAIL);
   llen = BUFSIZE;
   lbufp = bbuf;
   /*
    ** Loop around getting data and copy it to bbuf.
    */
   while(lret != CS_END_DATA)
   {
        (CS_VOID)srv_bzero(lbufp,BUFSIZE);
        lout = 0;
        lret = srv_get_text(spp, lbufp, llen, &lout);
        if(lret == CS_FAIL)
            break;
        *outlenp += lout;
        lbufp += lout;
    }
    if(lret == CS_END_DATA)
        return(CS_SUCCEED);
    else
        return(lret);
}
```

使用法

- srv_get_text は、クライアントからバルク・データを読むために使用します。バルク・データは、text や image でも可能です。
- クライアントからのバルク・データがすべて読み込まれるまで、srv_get_text を呼び出し続けなければなりません。データ・ストリーム全体が読み込まれると、CS_END_DATA を返します。
- srv_get_text は、SRV_BULK イベント・ハンドラからのみ呼び出すことができます。
- srv_get_text で読み込まれたカラムは、text または image でなければなりません。
- Open Server アプリケーションは、データ・ストリームに対して初めて srv_get_text を呼び出す前に、srv_text_info を呼び出す必要があります。次に、アプリケーションは、データ・ストリームを取得するために srv_get_text を呼び出します。カラム全体を読み込むために必要な回数だけ、srv_get_text が呼び出されます。
- Open Server は、text と image のデータ・ストリームを同等に扱いますが、例外として、text データだけを Open Server アプリケーションに送る前に変換します。Open Server が行う唯一の変換は、文字セット変換です。

参照

srv_bind、srv_descfmt、srv_send_text、srv_text_info、srv_thread_props、srv_xferdata、「国際化のサポート」(92 ページ)、「text と image」(184 ページ)

srv_getloginfo

説明

リモート・サーバとのパススルー接続の準備のために、クライアント・スレッドからログイン情報を取得します。

構文

```
CS_RETCODE srv_getloginfo(spp, loginfo)
SRV_PROC        *spp;
CS_LOGINFO      **loginfo;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

loginfo

新しく割り当てられた CS_LOGINFO 構造体のアドレスに設定される CS_LOGINFO ポインタへのポインタです。

戻り値

表 *3-49:* 戻り値 *(srv_getloginfo)*

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

```
#include  <ospublic.h>
/*
 ** Local Prototype
 */
extern CS_RETCODE   ex_srv_getloginfo PROTOTYPE((
CS_VOID             *spp,
CS_VOID             **loginfopp
));
/*
 ** EX_SRV_GETLOGINFO
**
**    Example routine to retrieve the client's login structure.
**
** Arguments:
**   spp        Thread control structure.
**   loginfopp  A pointer to client's login record returned here.
**
** Returns:
**
**    CS_SUCCEED    Login structure was retrieved successfully.
**    CS_FAIL       An error was detected.
*/
CS_RETCODE ex_srv_getloginfo(spp, loginfopp)
SRV_PROC   *spp;
CS_LOGINFO **loginfopp;
{
    /* Initialization. */
    *loginfopp = (CS_LOGINFO *)NULL;
    if(srv_getloginfo(spp, loginfopp) == CS_FAIL)
    {
        return(CS_FAIL);
    }
    return(CS_SUCCEED);
}
```

使用法

- srv_getloginfo は、パススルー・モードを使うゲートウェイ・アプリケーションで使用します。パススルー・モードにおいては、ゲートウェイ・アプリケーションはプロトコルを解釈することなく、クライアントとリモート Sybase サーバ間でパケットを相互に渡します。
- クライアントが直接サーバに接続する場合は、2 つのプログラムは、データの送受信に使用するプロトコル・フォーマットをネゴシエートします。ゲートウェイ・アプリケーションでプロトコル・パススルーを使用すると、Open Server は、クライアントとリモート・サーバ間でプロトコル・パケットを相互に渡します。つまり、クライアントとリモート・サーバは、プロトコル・バージョンについては一致していなければなりません。

- srv_getloginfo は、クライアントとリモート・サーバ間のプロトコル・フォーマットのネゴシエーションを可能にする 4 つの呼び出しの最初のものです (そのうち 2 つは CS-Library 呼び出しです)。これらの呼び出しは、SRV_CONNECT イベント・ハンドラにおいてのみ行えます。次にその呼び出しを示します。

  a srv_getloginfo − CS_LOGINFO 構造体を割り付け、クライアント・スレッドからのプロトコル情報を取り込みます。

  b ct_setloginfo − 手順 1 で取得したプロトコル情報を使用して CS_LOGINFO 構造体を準備し、ct_connect を使用してリモート・サーバにログインします。

  c ct_getloginfo − CS_CONNECTION 構造体から、新しく割り付けられた CS_LOGINFO 構造体にプロトコル・ログイン応答情報を転送します。

  d srv_setloginfo − 手順 3 で取得したリモート・サーバの応答をクライアントに送信し、CS_LOGINFO 構造体を解放します。

参照 srv_recvpassthru、srv_sendpassthru、srv_setloginfo

srv_getmsgq

説明 メッセージ・キューから次のメッセージを取得します。

構文

```
CS_RETCODE srv_getmsgq(msgqid, msgp, getflags, infop)
SRV_OBJID    msgqid;
CS_VOID      **msgp;
CS_INT       getflags;
CS_INT       *infop;
```

パラメータ

msgqid

メッセージを取得するためのメッセージ・キューの識別子です。メッセージ・キューを名前で参照するには、メッセージ・キュー ID を取得するための名前を使って srv_getobjid を呼び出します。

msgp

srv_getmsgq がメッセージのアドレスに設定するポインタ変数へのポインタです。

getflags

getflags の値は、論理和をとることができます。表 3-50 に、*getflags* の有効値とその意味を示します。

表 3-50: getflags の値 (srv_getmsgq)

値	説明
SRV_M_WAIT	メッセージがなければ、メッセージが届けられるまで srv_getmsgq はスリープする。
SRV_M_NOWAIT	メッセージがあるないに関係なく、srv_getmsgq は即座に戻る。
SRV_M_READ_ONLY	srv_getmsgq のデフォルト動作は、メッセージ・リストからメッセージを削除し、そのメッセージが読み込まれることを待っているスレッドがあればそれをウェイクアップする。SRV_M_READ_ONLY が設定された場合は、メッセージ・ポインタは返されるがメッセージはリストからは削除されず、メッセージを待っているスレッドはウェイクアップしない。このオプションは、メッセージ・キューの先頭を確認して、メッセージがそのスレッドのためのものかを判断することに使用する。

infop

CS_INT を指すポインタです。表 3-51 に、srv_getmsgq が CS_FAIL を返す場合に **infop* に返される可能性がある値を示します。

表 3-51: infop の値 (srv_getmsgq)

値	意味
SRV_I_WOULDWAIT	SRV_M_NOWAIT フラグが getflags フィールドに設定されて、読み込むべきメッセージがない。
SRV_I_DELETED	メッセージを待っている間に、メッセージ・キューが削除された。
SRV_I_INTERRUPTED	SRV_M_WAIT フラグが getflags フィールドに設定され、この呼び出しはメッセージが到着する前に中断された。
SRV_I_UNKNOWN	その他のエラーが発生した。メッセージはログ・ファイルを確認する。

戻り値

表 3-52: 戻り値 (srv_getmsgq)

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

```
#include <ospublic.h>
/*
 ** Local prototype
 */
CS_VOID    ex_srv_getmsgq PROTOTYPE((
SRV_OBJID  msgqid,
CS_INT     *infop
));
/*
 ** EX_SRV_GETMSGQ
**
**    Example routine to get messages from a message queue.
**
** Arguments:
**    msgqid-    The id of the message queue from which to get
 **               the message.
**
**    infop-     Will hold information about why this routine
 **               failed.Comes directly from srv_getmsg.
 ** Returns:
**    Nothing.If this routine returns, it is because srv_getmsgq
 **    failed.Check infop to see why it failed.
*/
CS_VOID    ex_srv_getmsgq(msgqid, infop)
SRV_OBJID  msgqid;
CS_INT     *infop;
{
     CS_CHAR   *message;   /* This message is a string. */
    /*
     ** Loop processing messages.Go to sleep if no messages are
     ** available.
     */
    while (srv_getmsgq(msgqid, (CS_VOID *)&message, SRV_M_WAIT,
              infop)== CS_SUCCEED)
    {
        /* Process message.*/
    }
    /* infop will contain the reason why it failed. */
    return ;
}
```

使用法

- srv_getmsgq は、メッセージ・キュー *msgqid* からの次のメッセージのアドレスを **msgp* に置きます。
- メッセージを送信したスレッドが、メッセージが読み込まれるまでスリープするように指定している場合は、スレッドはウェイクアップします。

参照

srv_createmsgq、srv_deletemsgq、srv_getobjid、srv_putmsgq

srv_getobjid

説明

指定の名前を持つメッセージ・キューまたはミューテックスのオブジェクト ID を調べます。

構文

```
CS_RETCODE srv_getobjid(obj_type, obj_namep,
                obj_namelen, obj_idp, infop)
CS_INT          obj_type;
CS_CHAR         *obj_namep;
CS_INT          obj_namelen;
SRV_OBJID       *obj_idp;
CS_INT          *infop;
```

パラメータ

obj_type

オブジェクトがミューテックス (SRV_C_MUTEX) またはメッセージ・キュー (SRV_C_MQUEUE) のどちらであるかを示します。

obj_namep

オブジェクトの名前が格納されている CS_CHAR バッファへのポインタです。

obj_namelen

**obj_namep* の文字列の長さです。文字列が null で終了する場合、*obj_namelen* は CS_NULLTERM とすることもできます。

obj_idp

オブジェクトの識別子を受け取る SRV_OBJID 構造体へのポインタです。

infop

CS_INT を指すポインタです。表 3-53 に、srv_getobjid が CS_FAIL を返す場合に **infop* に返される可能性がある値を示します。

表 *3-53: infop* の値 *(srv_getobjid)*

値	意味
SRV_I_NOEXIST	オブジェクトは存在しない。
SRV_I_UNKNOWN	null オブジェクト名など、他のエラーが生じた。

戻り値

表 *3-54:* 戻り値 *(srv_getobjid)*

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

```
#include  <ospublic.h>
/*
 ** Local Prototype
 */
CS_INT    ex_srv_getobjid PROTOTYPE((
CS_INT    obj_type,
CS_CHAR   *obj_name,
SRV_OBJID *obj_idp
));
```

```
/*
 ** EX_SRV_GETOBJID
**   An example routine to retrieve the object id for a specified
 **   message queue or mutex name.
 ** Arguments:
**   obj_type  SRV_C_MUTEX if requesting a mutex object id, and
 **             SRV_C_MQUEUE if requesting a message queue object
                id.
**   obj_name  A null terminated string which specifies the name
 **             of the message queue or the mutex.
**   obj_idp   A pointer to a SRV_OBJID structure that will store
 **             the identifier for the object.
 ** Returns:
**   CS_SUCCEED    If the object id was retrieved
                    successfully.
**   SRV_I_NOEXIST   If the object does not exist.
** CS_FAIL    If the object was not retrieved due to an error
 */
CS_INT        ex_srv_getobjid(obj_type, obj_name, obj_idp)
CS_INT        obj_type;
CS_CHAR       *obj_name;
SRV_OBJID     *obj_idp;
{
     CS_INT   info;  /* The reason for failure. */
     CS_INT   status;  /* The return status. */
    /* Validate the obj_type. */
     if ( (obj_type != SRV_C_MUTEX) && (obj_type !=
             SRV_C_MQUEUE) )
     {
         return(CS_FAIL);
     }
    /* Make sure that the object name is not null. */
     if ( obj_name == (CS_CHAR *)NULL )
    {
         return(CS_FAIL);
     }
    /* Ensure that the pointer to the SRV_OBJID is not null */
     if ( obj_idp == (SRV_OBJID *)NULL )
    {
         return(CS_FAIL);
     }
    /* Get the object id. */
     status = (CS_INT)srv_getobjid( obj_type, obj_name,
              CS_NULLTERM, obj_idp, &info);
    /* Check the status. */
     if ( (status == CS_FAIL) && (info == SRV_I_NOEXIST) )
    {
         status = SRV_I_NOEXIST;
     }
    return(status);
 }
```

使用法

Open Server は、メッセージ・キューやミューテックスのユニークなオブジェクト識別子をそれぞれの名前にマップするテーブルを保持しています。名前が与えられると、srv_getobjid はその識別子を探します。

参照

srv_createmsgq、srv_createmutex、srv_deletemsgq、srv_deletemutex、srv_getmsgq、srv_getobjname、srv_lockmutex、srv_putmsgq、srv_unlockmutex

srv_getobjname

説明

指定した識別子を持つ、メッセージ・キューまたはミューテックスの名前を取得します。

構文

```
CS_RETCODE srv_getobjname(obj_type, obj_id, obj_namep,
        obj_namelenp, infop)

CS_INT          obj_type;
SRV_OBJID       obj_id;
CS_CHAR         *obj_namep;
CS_INT          *obj_namelenp;
CS_INT          *infop;
```

パラメータ

obj_type
　オブジェクトがミューテックス (SRV_C_MUTEX) またはメッセージ・キュー (SRV_C_MQUEUE) のどちらであるかを示します。

obj_id
　オブジェクトのユニークな識別子です。

obj_namep
　オブジェクトの名前がコピーされている CS_CHAR バッファへのポインタです。バッファのサイズは、オブジェクト名、および null 文字 (*obj_namelenp* が NULL の場合) を格納できる大きさでなければなりません。オブジェクト名の最大長は、null 終了バイトを含めない SRV_MAXNAME の文字の長さです。

obj_namelenp
　オブジェクトの長さを受け取る CS_INT へのポインタです。*obj_namelenp* が NULL の場合、検索された名前は **obj_namep* にコピーされ、NULL 文字で終了します。NULL 以外の場合は、**obj_namep* の名前の長さは **obj_namelenp* に格納されます。

infop
　ID *obj_id* のオブジェクトが存在しない場合に、SRV_I_NOEXIST に設定される CS_INT へのポインタです。

戻り値

表 *3-55:* 戻り値 *(srv_getobjname)*

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

```
#include  <ospublic.h>
#include  <stdio.h>
/*
 ** Local Prototype
 */
CS_RETCODE  ex_srv_getobjname PROTOTYPE((
CS_INT      obj_type,
SRV_OBJID   obj_id
));
/*
 ** EX_SRV_GETOBJNAME
**    Example routine to illustrate the use of srv_getobjname to
 **    get the name of mutex or message queue with id = obj_id
 **    where obj_id was earlier returned by srv_createmutex or
 **    srv_createmsgq.
 ** Arguments:
**    obj_type - Type of object; SRV_C_MUTEX or SRV_C_MQUEUE.
**    obj_id   - The unique identifier of the object.
 ** Returns:
**    CS_SUCCEED   Memory was allocated successfully.
**    CS_FAIL      Memory allocation failure occured.
 */
CS_RETCODE   ex_srv_getobjname(obj_type, obj_id)
CS_INT       obj_type;
SRV_OBJID    obj_id;
{
    CS_CHAR      obj_name[SRV_MAXNAME+1];
    CS_INT       obj_namelen;
    CS_INT       info;
    CS_RETCODE   ret;
    /* Get object name. */
     ret = srv_getobjname(obj_type, obj_id, obj_name,
     &obj_namelen, &info);
    /* Print information depending on retcode */
     switch(ret)
    {
       case CS_FAIL:
         if (info == SRV_I_NOEXIST)
         {
             fprintf(stderr, "%s object with id: %d does not
                     exist¥n", (obj_type == SRV_C_MUTEX) ?
                     "Mutex" :"Message Queue", (CS_INT)obj_id);
         }
         else
             fprintf (stderr, "srv_getobjname failed¥n");
        break;
     case CS_SUCCEED:
         fprintf (stderr, "%s name: %s for id: %d¥n",
            (obj_type == SRV_C_MUTEX) ?"Mutex" : "Message Queue",
```

```
                obj_name, (CS_INT)obj_id);
            break;
        default:
            fprintf (stderr, "Unknown return code from
                     srv_getobjname¥n");
            ret = CS_FAIL;
            break;
        }
        return (ret);
    }
```

使用法

- Open Server では、メッセージ・キューやミューテックスのユニークな識別子をメッセージ・キューやミューテックスの名前にマップするテーブルを保持しています。識別子を指定すると、srv_getobjname によって名前が検索されます。
- アプリケーションによっては、リファレンス・メッセージ・キューやミューテックスを、名前で調べた方が便利な場合もあります。ミューテックスやメッセージ・キュー・サービスを使用している識別子を調べるために、srv_getobjid を使用することもできます。

参照

srv_createmsgq、srv_createmutex、srv_deletemsgq、srv_deletemutex、srv_getmsgq、srv_getobjid、srv_lockmutex、srv_putmsgq、srv_unlockmutex

srv_getserverbyname

説明

server_name の接続情報を返し、必要に応じてメモリを割り付けます。srv_getserverbyname によって割り付けたメモリは、srv_freeserveraddrs を呼び出すことで解放できます。

構文

CS_RETCODE srv_getserverbyname(CS_CHAR **server_name*, CS_INT *namelen*, CS_INT *querytype*, CS_INT *result_type*, void **resultptr*, CS_INT **result_cnt*)

パラメータ

server_name

検索するサーバの名前です。

namelen

server_name の長さ。CS_NULLTERM として指定できます。

querytype

server_name に対してマスタ (CS_ACCESS_CLIENT_MASTER) またはクエリ (CS_ACCESS_CLIENT_QUERY) のエントリを選択します。

result_type

接続情報のデータ・フォーマットを示します。*result_type* は、SRV_C_GETADDRS または SRV_C_GETSTRS として指定できます。

resultptr
クエリの結果を保持するために srv_getserverbyname によって割り付けられるポインタです。resultptr は、クエリ結果のアドレスを受け取るポインタのアドレスです。

result_cnt
server_name に対して返されるアドレスの数を含む CS_INT へのポインタです。

使用法

result_type は SRV_C_GETADDRS として指定できます。情報は CS_TRANADDR 構造体の配列として返されます。または、*result_type* を SRV_C_GETSTRS として指定すると、文字列へのポインタの配列を *network-protocol protocol-address filter-information* というフォーマットで返すことができます。たとえば、*network-protocol* が “tcp”、*protocol-address* が “myhost 4000”、*filter-information* が “ssl” である場合は、“tcp myhost 4000 ssl” の結果を受け取ります。

参照

srv_freeserveraddrs、srv_send_ctlinfo

srv_handle

説明

Open Server アプリケーションにイベント・ハンドラをインストールします。

構文

```
SRV_EVENTHANDLE_FUNC (*srv_handle(ssp, event,
                handler))()

SRV_SERVER                  *ssp;
CS_INT                      event;
SRV_EVENTHANDLE_FUNC   handler;
```

パラメータ

ssp
Open Server の制御構造体へのポインタです。このパラメータはオプションであり、下位互換性を提供します。

event
handler が処理するイベントです。通常の Open Server イベントのリストを次に示します。

- SRV_ATTENTION
- SRV_BULK
- SRV_CONNECT
- SRV_CURSOR
- SRV_DISCONNECT/SRV_URGDISCONNECT
- SRV_DYNAMIC
- SRV_FULLPASSTHRU
- SRV_LANGUAGE
- SRV_MSG

- SRV_OPTION
- SRV_RPC
- SRV_START
- SRV_STOP

ユーザ定義のイベント ― ユーザ定義のイベントは、srv_define_event を使用して定義されます。

各イベントの詳細については、「イベント」(84 ページ) を参照してください。

handler

event 要求が生じたときに呼び出す関数へのポインタです。ハンドラとして NULL を渡すことによって、デフォルトのイベント・ハンドラがインストールされます。

戻り値

表 *3-56:* 戻り値 *(srv_handle)*

戻り値	意味
イベント処理関数へのポインタ	関数のロケーション。
null ポインタ	ルーチンが失敗した。

例

```
#include   <ospublic.h>
/*
 ** Local Prototype
 */
extern CS_RETCODE  ex_srv_handle PROTOTYPE((
SRV_EVENTHANDLE_FUNC       funcp
));
/*
 ** EX_SRV_HANDLE
**   Install a SRV_START handler.
 ** Arguments:
**    funcp  Handler to install.
 ** Returns:
**    CS_SUCCEED   Start handler was installed successfully.
 **    CS_FAIL      An error was detected.
 */
CS_RETCODE               ex_srv_handle(funcp)
SRV_EVENTHANDLE_FUNC     funcp;
{
    if(srv_handle((SRV_SERVER *)NULL, SRV_START, funcp) ==
         CS_FAIL )
    {
        return(CS_FAIL);
    }
    return(CS_SUCCEED);
}
```

使用法

- srv_handle は、Open Server が特定のイベントの処理要求を受けた場合、Open Server に特定の関数を呼び出すように通知します。
- Open Server は、1 つの引数を使って *handler* を呼び出します。

  下記のイベントの場合、イベント・ハンドラの引数として、Open Server 制御構造体へのポインタが必要です。

  - SRV_START
  - SRV_STOP

  下記のイベントの場合、イベント・ハンドラの引数として、スレッド制御構造体へのポインタが必要です。

  - SRV_ATTENTION
  - SRV_BULK
  - SRV_CONNECT
  - SRV_CURSOR
  - SRV_DISCONNECT/SRV_URGDISCONNECT
  - SRV_DYNAMIC
  - SRV_FULLPASSTHRU
  - SRV_LANGUAGE
  - SRV_MSG
  - SRV_OPTION
  - SRV_RPC

  あらゆるユーザ定義のイベント
- 各 Open Server イベントには、既知の名前を持つデフォルト・ハンドラがあります。srv_handle でイベント・ハンドラをインストールすると、デフォルト・ハンドラが置換されます。
- イベント・ハンドラは、動的にインストールすることができます。次回のイベント発生時に、新しいイベント・ハンドラが呼び出されます。
- イベント・ハンドラは、CS_SUCCEED を返す必要があります。

参照

srv_define_event、srv_event、srv_event_deferred、「イベント」(84 ページ)

srv_init

説明

Open Server アプリケーションを初期化します。

構文

```
SRV_SERVER *srv_init(scp, servernamep, namelen)
SRV_CONFIG      *scp;
CS_CHAR         *servernamep;
CS_INT          namelen;
```

パラメータ

scp

すべての Open Server 設定オプションの値を保持する設定構造体です。この引数はオプションであり、下位互換性を提供します。

servernamep

Open Server アプリケーション名へのポインタです。指定した名前は、必要なネットワーク情報を取得するために interfaces ファイル内で検索されます。Open Server の名前として (CS_CHAR *) NULL を使用する場合は、DSLISTEN の値がサーバ名になります。DSLISTEN が明示的に設定されていない場合には、デフォルトの文字列 “SYBASE” がサーバ名になります。

namelen

**servernamep* の文字列の長さをバイト数で示したものです。文字列が (CS_CHAR *) NULL の場合は、*namelen* は無視されます。文字列が null で終了する場合、*namelen* は CS_NULLTERM とすることもできます。

戻り値

表 3-57: 戻り値 (srv_init)

戻り値	意味
SRV_SERVER ポインタ	ルーチンが正常に実行された。
(SRV_SERVER *) NULL	ルーチンが失敗した。

例

```
#include <ospublic.h>
/*
 ** Local prototype.
 */
SRV_SERVER    *ex_srv_init PROTOTYPE((
SRV_CONFIG    *scp
));
/*
 ** EX_SRV_INIT
**
**    Example routine to initialize an Open Server application.
**
 ** Arguments:
**    scp   -   A pointer to the configuration structure.
 **
 ** Returns:
**    On success, a pointer to a newly allocated SRV_SERVER
       structure.
**    On failure, NULL.
 **
```

```
 */
SRV_SERVER   *ex_srv_init(scp)
SRV_CONFIG   *scp;
{
    SRV_SERVER   *server;
    CS_CHAR      *servername = "EX_SERVER";
    server = srv_init(scp, servername, CS_NULLTERM);
    return (server);
}
```

使用法

- サーバは初期化してから srv_run で起動します。
- srv_init は Open Server アプリケーションを初期化します。初期化の過程を構成する主なものは、サーバに必要なデータ構造体の割り付け、サーバ・ステータスの初期化、およびネットワーク・リスナの起動です。
- デフォルト以外の値を指定する場合は、srv_init を呼び出す前に設定オプションを設定する必要があります。設定可能なオプションのリストについては、srv_props のページを参照してください。
- ライブラリのバージョン情報とデフォルト国際化値を設定するには、srv_version を呼び出してから srv_init を呼び出します。
- SRV_STOP イベントが発生すると、Open Server は SRV_SERVER 構造体を解放します。SRV_SERVER 構造体を Open Server アプリケーションが解放しないようにしてください。
- interfaces ファイルの指定については、srv_props のページを参照してください。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。

参照

srv_props、srv_run、srv_version

srv_langcpy

説明

クライアントの言語要求を、アプリケーション・バッファにコピーします。

構文

CS_INT srv_langcpy(spp, start, nbytes, bp)

```
SRV_PROC   *spp;
CS_INT      start;
CS_INT      nbytes;
CS_BYTE    *bp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

start

要求バッファの文字のコピーを開始する位置です。要求バッファの最初の文字の位置は、0 番目となります。

nbytes
コピーする文字数です。*nbytes* が -1 の場合は、srv_langcpy は可能なかぎりの文字をコピーします。0 バイトをコピーすることもできます。コピーする文字が *nbytes* に満たない場合は、srv_langcpy は要求バッファにあるだけの文字をコピーします。

bp
バイトのコピー先となるユーザ提供バッファへの CS_CHAR ポインタです。

戻り値

表 *3-58:* 戻り値 *(srv_langcpy)*

戻り値	意味
整数	コピーしたバイト数。
-1	このクライアントから現在の言語要求はない。

例

```
#include  <ospublic.h>

/*
** Local Prototype
*/
CS_RETCODE     ex_srv_langcpy PROTOTYPE((
SRV_PROC       *spp,
CS_CHAR        *buf,
CS_INT         size,
CS_INT         *outlen
));

/*
** EX_SRV_LANGCPY
**
**    Example routine to illustrate the use of srv_langcpy to
 **    copy language commands sent by a client.
**
 ** Arguments:
**    spp      A pointer to internal thread control structure.
**    buf      A CS_CHAR pointer to buffer for language commands.
**    size     The size of the buffer; A CS_INT.
**    outlen   A pointer to CS_INT; the actual length of
                language query copied to buf is returned here. -1
 **             is returned in case of failure.
**
 ** Returns:
**
**    CS_SUCCEED   Language request was copied successfully.
**    CS_FAIL      An error was detected.
*/
CS_RETCODE   ex_srv_langcpy(spp, buf, size, outlen)
SRV_PROC     *spp;
CS_CHAR      *buf;
CS_INT       size;
```

```
CS_INT       *outlen;
{
   CS_INT        act_len; /* actual length of language request */

    /* Initialization.*/
    *outlen = (CS_INT)-1;

    /* Get the length of language request.*/
    if ((act_len = srv_langlen(spp)) == -1)
        return (CS_FAIL);

    /* Check to see whether we got a buffer of adequate size. */
    if (size < (act_len +1))
        return (CS_FAIL);

    /* Copy language commands.*/
    if (srv_langcpy(spp, (CS_INT)0, act_len, buf) <= 0)
        return (CS_FAIL);

    /* Set the actual length copied. */
    *outlen = act_len;

    return (CS_SUCCEED);
}
```

使用法

- クライアントから言語要求を受信すると、srv_langcpy は要求バッファの一部を Open Server プログラム変数にコピーできます。送信先バッファに格納されたコピーは、null で終了します。
- srv_langcpy は、カーソル宣言または更新文において言語文字列の処理にも使用されます。

  警告！ srv_langcpy は、送信先バッファが最低でも nbytes + 1 のバイト数を処理できるサイズがあることを想定します。

- 言語要求バッファの全体の長さを変更するには、srv_langlen を呼び出します。
- 要求バッファには、Transact-SQL 文だけでなく、どのような文字列でも格納できます。文字列の処理は、Open Server アプリケーションの責任です。

参照

srv_langlen

srv_langlen

説明　言語要求バッファの長さを返します。

構文

```
CS_INT srv_langlen(spp)
SRV_PROC   *spp;
```

パラメータ

spp
　内部スレッド制御構造体へのポインタです。

戻り値

表 *3-59:* 戻り値 *(srv_langlen)*

戻り値	意味
整数	言語要求バッファの長さをバイト数で示したもの。
-1	このクライアントから現在の言語要求はない。

例

```
#include  <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE ex_srv_langlen PROTOTYPE((
SRV_PROC  *spp,
CS_INT    *len
));

/*
 ** EX_SRV_LANGLEN
**  Example routine to return the length of the language request
 **  buffer using srv_langlen.
**
 ** Arguments:
**  spp    A pointer to the internal thread control structure.
**  len    Return pointer for the length of the language string.
 **         If there is no language command -1 is returned.
**
 ** Returns:
**
**   CS_SUCCEED   Language length was retrieved successfully.
**   CS_FAIL      An error was detected.
 */
CS_RETCODE   ex_srv_langlen(spp, len)
SRV_PROC     *spp;
CS_INT       *len;
{
    /* Retrieve the language length.    */
    if ((*len = srv_langlen(spp)) < 0)
    {
        return(CS_FAIL);
    }
    return(CS_SUCCEED);
}
```

使用法

- クライアントから言語要求を受信すると、srv_langlen は要求バッファの長さを返します。
- srv_langlen は、カーソル宣言または更新文において言語文字列の処理にも使用されます。
- 要求バッファの全部または一部は、srv_langcpy でアクセスできます。
- 要求バッファは、Transact-SQL 文を含む、どのような文字列でも含むことができます。文字列の処理は、Open Server アプリケーションの責任です。

参照

srv_langcpy

srv_lockmutex

説明

ミューテックスをロックします。

構文

```
CS_RETCODE srv_lockmutex(mutex_id, waitflag, infop)
SRV_OBJID       mutex_id;
CS_INT          waitflag;
CS_INT          *infop;
```

パラメータ

mutex_id

srv_createmutex への呼び出しで返されるユニークなミューテックス識別子です。ミューテックスの名前を指定すると、srv_getobjid の呼び出しで *mutex_id* を取得できます。

waitflag

ミューテックスがただちに付与されない場合は、ミューテックス・ロックを要求しているスレッドは待っているか、単に戻るかを指定します。**indp* の値は、ロックが付与されたかどうかを示します。*waitflag* の有効値は 2 つあります。ロックがただちに付与されない場合にスレッドが待つことを指定する SRV_M_WAIT と、ロックが付与されない場合にスレッドが戻ることを指定する SRV_M_NOWAIT です。

infop

次の値のいずれかに設定した CS_INT へのポインタです。

SRV_I_SYNC － ロックは同期的に付与されました。つまり、ロックを要求していたスレッドにロックを待機するための中断はありませんでした。srv_lockmutex は CS_SUCCEED を返しました。

SRV_I_GRANTED － ロックを要求していたスレッドが中断されて、他のスレッドがミューテックスのロックを解放するまで待機した後に、ロックは付与されました。srv_lockmutex は CS_SUCCEED を返しました。

SRV_I_INTERRUPTED − スレッドはロックを待機している間にアテンションを受け取りました。ロックは付与されず、srv_lockmutex は CS_FAIL を返しました。

SRV_I_WOULDWAIT − *waitflag* パラメータは SRV_M_NOWAIT に設定され、スレッドはロックを待機しなければならない状態でした。ロックは付与されず、srv_lockmutex は CS_FAIL を返しました。

SRV_I_UNKNOWN − ミューテックスが存在しないなど、その他のエラーが発生しました。srv_lockmutex は CS_FAIL を返しました。

戻り値

表 3-60: 戻り値 (srv_lockmutex)

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

```
#include  <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE    ex_srv_lockmutex PROTOTYPE((
SRV_OBJID     mid
));
/*
 ** EX_SRV_LOCKMUTEX
 **
 **  Example routine to illustrate the use of srv_lockmutex.
 **
  ** Arguments:
 **       mid - The id of the mutex to lock.
 **
  ** Returns:
 **
 **  CS_SUCCEED   Mutex successfully locked.
 **  CS_FAIL      An error was detected.
 */
CS_RETCODE   ex_srv_lockmutex(mid)
SRV_OBJID    mid;      /* The mutex id. */
{
    CS_INT   info;     /* Information output variable. */

    /*
     ** Request the mutex lock - sleep until we get it.
     */
    if( srv_lockmutex(mid, SRV_M_WAIT, &info) == CS_FAIL )
    {
        /*
         ** An error was al&ready raised.
         */
```

```
            return CS_FAIL;
        }

        /*
         ** All done.
         */
        return CS_SUCCEED;
    }
```

使用法

- ミューテックスは、複数スレッドによる同時アクセスから保護されなければならないデータ・オブジェクトやプログラム・リソースと関連付けられています。
- ミューテックス・ロックは、先着優先で付与されます。
- ロックが付与されるのは、他のスレッドがすでにミューテックスのロックを得ていない場合のみです。
- srv_lockmutex は、SRV_START または SRV_ATTENTION ハンドラでは使用できません。
- スレッドは同じミューテックスを何度もロックすることができますが、他のスレッドがミューテックスをロックする前に srv_lockmutex を呼び出した回数だけ、srv_unlockmutex も一度ずつ呼び出す必要があります。
- 待っていたミューテックスが削除された場合、srv_lockmutex は CS_FAIL を返します。

参照 srv_createmutex、srv_deletemutex、srv_getobjid、srv_unlockmutex

srv_log

説明 Open Server ログ・ファイルにメッセージを書き込みます。

構文

```
CS_RETCODE srv_log(ssp, datestamp, msgp, msglen)
SRV_SERVER    *ssp;
CS_BOOL        datestamp;
CS_CHAR        *msgp;
CS_INT         msglen;
```

パラメータ

ssp

Open Server へのハンドラです。この引数はオプションであり、下位互換性を提供します。

datestamp

datestamp が CS_TRUE の場合、現在の日時がログ・メッセージの先頭に追加されます。*datestamp* が CS_FALSE の場合は、ログ・メッセージは追加されません。

msgp
メッセージの実際のテキストへのポインタです。

msglen
msg の長さをバイト数で示したものです。**msgp* の文字列が null で終了する場合、*msglen* は CS_NULLTERM とすることもできます。

戻り値

表 3-61: 戻り値 (srv_log)

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

```
#include  <ospublic.h>
#include  <string.h>
/*
 ** Local Prototype.
 */
CS_RETCODE   ex_srv_log PROTOTYPE((
SRV_SERVER   *ssp,
CS_CHAR      *msg_txt
));
/*
 ** EX_SRV_LOG
**
**    Example routine to log a message.
**
** Arguments:
**
**  ssp         A pointer to the Open Server state information
 **              control structure.
**    msg_txt   Text of message to log.
** Returns
**
**  CS_SUCCEED      Thread was created.
**  CS_FAIL         An error was detected.
**
 */
CS_RETCODE  ex_srv_log(ssp, msg_txt)
SRV_SERVER  *ssp;
CS_CHAR     *msg_txt;
{
    CS_RETCODE        lret;
    CS_INT            msg_len;
    /* Check arguments.                    */
    if(ssp == (SRV_SERVER *)0)
        return(CS_FAIL);
    if(msg_txt == (CS_CHAR *)NULL)
        return(CS_FAIL);
    msg_len=strlen(msg_txt);
```

```
        /*
        ** Log the message -  We use CS_TRUE as the second argument
         **                    to force the date and time to be
         **                    added to the beginning of the logged
         **                    message.If you do not want a
         **                    datestamp then use CS_FALSE.
        */
        lret = srv_log(ssp,CS_TRUE,msg_txt,msg_len);
        return(lret);
    }
```

使用法

- srv_log は、Open Server ログ・ファイルにメッセージを書き込みます。ログ・ファイルのデフォルト名は、*srv.log* です。名前は、srv_props で設定できます。
- メッセージはログ・ファイルに付け加えられます。
- ログ・ファイルの名前は、srv_props ルーチンでアクセスできます。
- 改行文字は、**msgp* のテキストには追加されません。
- ログ・ファイルは、srv_props によって設定された SRV_TRUNCATELOG プロパティに基づいてトランケートされます。
- メッセージの長さが SRV_MAXMSG を超える場合、Open Server はメッセージをトランケートします。これは、メッセージが null で終了しているかどうかに関係ありません。
- srv_init が終了していない場合、メッセージはブート・ウィンドウに表示されます。

参照 srv_props

srv_mask

説明 SRV_MASK_ARRAY 構造体のビットを初期化、設定、クリア、またはチェックします。

構文

```
CS_RETCODE srv_mask(cmd, maskp, bit, infop)
CS_INT              cmd;
SRV_MASK_ARRAY      *maskp;
CS_INT              bit;
CS_BOOL             *infop;
```

パラメータ

cmd

実行中のアクションです。表 3-62 に、*cmd* の有効値を示します。

表 3-62: cmd の有効値 (srv_mask)

値	動作
CS_SET	**maskp* の SRV_MASK_ARRAY の bit を設定。
CS_GET	**maskp* の SRV_MASK_ARRAY で bit が現在設定されているかどうかを調べる。bit が設定されている場合は、**infop* は CS_TRUE に設定される。そうでない場合は、CS_FALSE に設定される。
CS_CLEAR	**maskp* の SRV_MASK_ARRAY の bit をクリアする。
CS_ZERO	すべての bit がオフになるように、**maskp* の SRV_MASK_ARRAY を初期化する。*cmd* が CS_ZERO に設定されている場合は、*bit* と *infop* が無視される。

maskp

SRV_MASK_ARRAY 構造体へのポインタです。

bit

SRV_MASK_ARRAY で初期化、設定、クリア、またはチェックされているビットです。これは、0 から SRV_MAXMASK_LENGTH の間の整数でなければなりません。SRV_MAXMASK_LENGTH は、*ospublic.h* で定義されます。

infop

bit が設定されているかどうかを示す変数へのポインタです。*cmd* が CS_SET、CS_CLEAR、または CS_ZERO の場合は、このパラメータは無視されます。

戻り値

表 3-63: 戻り値 (srv_mask)

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

```
#include  <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_mask PROTOTYPE((
SRV_MASK_ARRAY  *maskptr,
CS_INT          bit
));

/*
** EX_SRV_MASK
**
**    Example routine to manipulate bits in a SRV_MASK_ARRAY
 **    structure.
**
```

```
 ** Arguments:
**    maskptr    A pointer to a mask array.
**    bit        The bit to examine.
**
 ** Returns:
**
**    CS_SUCCEED
**    CS_FAIL
*/
CS_RETCODE       ex_srv_mask(maskptr, bit)
SRV_MASK_ARRAY  *maskptr;
CS_INT  bit;
{
    CS_BOOL    info = CS_TRUE;

    if (srv_mask(CS_GET, maskptr, bit, &info) == CS_FAIL)
    {
         return(CS_FAIL);
    }
    else
    {
        /* Has the bit been set? */
        if (info == CS_FALSE)
            return(CS_FAIL);
        else
            return(CS_SUCCEED);
    }
}
```

使用法 srv_mask は、SRV_MASK_ARRAY にアクセスし、変更するために使用されます。

srv_msg

説明 メッセージ・データ・ストリームを送信または受信します。

構文

```
CS_RETCODE srv_msg(spp, cmd, msgidp, status)
SRV_PROC    *spp;
CS_INT       cmd;
CS_INT       *msgidp;
CS_INT      *statusp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

cmd

アプリケーションがメッセージの送信または取得のどちらを行うために srv_msg を呼び出しているかを示します。表 3-64 に、cmd の有効値を示します。

表 3-64: cmd の値 (srv_msg)

値	説明
CS_SET	srv_msg は、メッセージをクライアントに送信する前に *status* および *msgid* の値を設定している。
CS_GET	srv_msg は、受信しているメッセージの *status* および *msgid* の値を取得している。

msgidp

現在のメッセージのメッセージ ID へのポインタです。Open Server アプリケーションでメッセージを送信する場合 (CS_SET)、ここで ID を指定してください。アプリケーションがメッセージを読み込んでいる場合 (CS_GET)、受信したメッセージのメッセージ ID がここに返されます。SRV_MINRESMSG から SRV_MAXRESMSG の値は、Sybase の内部使用のために予約されています。続いてメッセージ ID が TDS から smallint (2 バイト) として送信されるので、メッセージ ID を符号なしの CS_SMALLINT として定義した場合、ユーザ自身のメッセージに使用できる有効範囲は SRV_MAXRESMSG から 65535 までです。

statusp

現在のメッセージのステータスへのポインタです。Open Server アプリケーションがメッセージを受信している場合 (CS_GET)、Open Server はメッセージ・ステータスで **statusp* を更新します。アプリケーションがメッセージを送信している場合 (CS_SET)、**statusp* には、送られるメッセージのステータスが含まれていることが必要です。表 3-65 に、**statusp* の有効値を示します。

表 3-65: statusp の値 (srv_msg)

値	説明
SRV_HASPARAMS	メッセージにパラメータがある。
SRV_NOPARAMS	メッセージにパラメータはない。

戻り値

表 3-66: 戻り値 (srv_msg)

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

```
#include  <ospublic.h>
/*
 ** Local prototype.
 */
CS_RETCODE    ex_srv_msg PROTOTYPE((
SRV_PROC      *spp
));

/*
** EX_SRV_MSG
 **
 **    Example routine to receive and send a message datastream.
 **
 ** Arguments:
 **  spp  A pointer to an internal thread control structure.
 **
 ** Returns:
 **  CS_SUCCEED if we were successful in both receiving and
 **  sending a message stream.
 **
 **  CS_FAIL if an error was detected.
 **
 */
CS_RETCODE    ex_srv_msg(spp)
SRV_CONFIG    *scp;
{
    CS_RETCODE      result;
    CS_INT          msgid;
    CS_INT          status;

    /*
     ** We will first get a message and process any parameters.
     */

    result = srv_msg(spp, CS_GET, &msgid, &status);

    if (result == CS_FAIL)
    {
        return (CS_FAIL);
    }

    if (status == SRV_HASPARAMS)
    {
        /*
         ** Process parameters here using srv_bind and
         ** srv_xferdata.
         */
    }
```

```
        /*
        ** Now, an example of sending a message.
        */
        msgid = 32768;
        status = SRV_NOPARAMS;

        result = srv_msg(spp, CS_SET, &msgid, &status);

        if (result == CS_FAIL)
        {
            return (CS_FAIL);
        }
        /*
         ** If the message has parameters, send it across using     ** srv_xferdata
         */
        if (status == SRV_HASPARAMS)
         {
            result = srv_xferdata(spp, CS_SET, SRV_MSGDATA);
        }
        return(result);
    }
```

使用法

- srv_msg は、TDS メッセージ・データ・ストリームの送信または受信に使用されます。
- クライアントから受信するメッセージ・データ・ストリームはすべて、SRV_MSG イベントを発生させます。受信された各メッセージに対して個別のイベントが発生します。
- メッセージにパラメータがある場合は、**statusp* は CS_HASPARAMS の値を含んでいます。アプリケーションは、*type* を SRV_MSGDATA に設定して、srv_descfmt、srv_bind、srv_xferdata を使用してパラメータの取得と保存ができます。
- アプリケーションは、メッセージのパラメータの数を srv_numparams を呼び出して調べることができます。
- srv_msg ルーチンは、ステータスと ID を送信するために使用されます。メッセージの実際のパラメータは、パラメータが存在する場合、*type* 引数を SRV_MSGDATA に設定して、srv_descfmt、srv_bind、srv_xferdata を使用して送信されます。
- アプリケーションは、複数のメッセージ・データ・ストリームを送受信することができます。
- srv_xferdata は、メッセージ・パラメータの取得または送信の場合にかぎり必要です。これらの目的で使用する場合、送信または受信するメッセージごとに、srv_xferdata を 1 回呼び出してください。パラメータが存在しないときに srv_xferdata を使用すると、Open Server はエラーを返します。

- srv_msg を SRV_MSG イベント・ハンドラで呼び出せるのは、*cmd* が CS_GET に設定されているときに限ります。*cmd* が CS_SET の場合は、どのハンドラでも呼び出すことができます。

参照

srv_bind、srv_descfmt、srv_numparams、srv_xferdata、「データ・ストリーム・メッセージ」(72 ページ)

srv_negotiate

説明

ネゴシエーション・ログイン情報をクライアントに送信およびクライアントから受信します。

構文

```
CS_RETCODE srv_negotiate(spp, cmd, type)
SRV_PROC    *spp;
CS_INT       cmd;
CS_INT       type;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

cmd

アプリケーションがネゴシエーション・ログイン情報の送信または取得のどちらを行うために srv_negotiate を呼び出しているかを示します。表 3-67 に、*cmd* の有効値を示します。

表 3-67: *cmd* の値 (*srv_negotiate*)

値	説明
CS_SET	*type* で定義されるネゴシエーション・ログイン情報を、クライアントに送信する。
CS_GET	*type* で定義されるネゴシエーション・ログイン情報を、クライアントから受信する。

type

クライアントから送信または読み取られるネゴシエーション・ログイン情報のタイプを示します。表 3-68 に、type の有効値を示します。

表 *3-68: type* の値 *(srv_negotiate)*

値	説明
SRV_NEG_CHALLENGE	ネゴシエーション・ログイン情報は、クライアントに送信されるチャレンジ・バイト・ストリーム (CS_SET)、またはクライアントから読み込まれるチャレンジ応答バイト・ストリーム (CS_GET) である。
SRV_NEG_ENCRYPT	ネゴシエーション・ログイン情報は、クライアントに送信される暗号化キーから構成される。次に、クライアントはこれを使用して、ローカル・パスワードおよびリモート・パスワードを暗号化する。このタイプは、*cmd* が CS_SET のときにのみ有効。
SRV_NEG_EXTENDED_ENCRYPT	パスワードの暗号化に使用されるネゴシエーション・ログイン情報とパブリック・キー。これらの情報はクライアントが使用する。このタイプは、*cmd* が CS_SET のときにのみ有効。
SRV_NEG_EXTENDED_LOCPWD	SRV_NEG_EXTENDED_ENCRYPT チャレンジに応答してクライアントから送信された暗号化されたパスワードのパブリック・キー。このタイプは、*cmd* が CS_GET のときにのみ有効。
SRV_NEG_EXTENDED_REMPWD	ネゴシエーション・ログイン情報は、SRV_NEG_EXTENDED_ENCRYPT チャレンジに応答してクライアントから送信されたリモート・サーバ名と対応する暗号化されたパスワードのパブリック・キーが対になっている可変数。このタイプは、*cmd* が CS_GET のときにのみ有効。
SRV_NEG_LOCPWD	SRV_NEG_ENCRYPT チャレンジに応答してクライアントから送信された暗号化されたローカル・パスワード。このタイプは、*cmd* が CS_GET のときにのみ有効。
SRV_NEG_REMPWD	ネゴシエーション・ログイン情報は、SRV_NEG_ENCRYPT チャレンジに応答してクライアントから送信された、リモート・サーバ名と暗号化されたリモート・パスワードが対になっている可変数。このタイプは、*cmd* が CS_GET のときにのみ有効。
SRV_NEG_SECLABEL	ネゴシエーション・ログイン情報は、クライアントに送信されるセキュリティ・ラベルまたはクライアントからサーバに送信された、セキュリティ・ラベルのセットの要求である。
SRV_NEG_SECSESSION	フル・パススルー・ゲートウェイ・アプリケーションがネゴシエーション・ログイン情報を使用して、ゲートウェイ・クライアントとリモート・サーバ間のダイレクト・セキュリティ・セッションを確立する。これは、チャレンジに応答するセキュリティ・ネゴシエーションと同様である。「セキュリティ・サービス」(158 ページ) を参照。
CS_USER_MSGID から CS_USER_MAX_MSGID の間の整数値	ネゴシエーション・ログイン情報は、*type* 引数自体で識別される、アプリケーション定義ハンドシェイクの一部。

戻り値

表 *3-69:* 戻り値 *(srv_negotiate)*

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

```
#include  <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE    ex_srv_negotiate PROTOTYPE((
SRV_PROC      *sproc
));

/*
** EX_SRV_NEGOTIATE
**  An example routine to retrieve negotiated login information
 **  by using srv_negotiate.
**
** Arguments:
**  sproc  A pointer to an internal thread control structure.
**
** Returns:
**  CS_SUCCEED   The login information was retrieved.
**  CS_FAIL      An error was detected.
*/
CS_RETCODE   ex_srv_negotiate(sproc)
SRV_PROC     *sproc;
{
    /*
    ** Check to make sure that the thread control structure is
     ** not NULL.
    */
    if ( sproc == (SRV_PROC *)NULL )
    {
        return(CS_FAIL);
    }

    /* Now get the login information. */
    if ( srv_negotiate(sproc, CS_GET, SRV_NEG_CHALLENGE) ==          CS_FAIL )
    {
        return(CS_FAIL);
    }

    return(CS_SUCCEED);
}
```

使用法

- srv_negotiate は、ネゴシエーション・ログイン情報をクライアントに送信し、クライアントからネゴシエーション・ログイン応答を受信するために使用されます。
- srv_negotiate によって、Open Server アプリケーションはその SRV_CONNECT イベント・ハンドラ内でセキュア・ログイン・プロセスを実行できます。安全なコンピューティング環境においては、接続時にクライアントの身元を確認するために、アプリケーションはネゴシエーション・ログインのチャレンジと暗号化パスワードを発行することによって、より厳格に認証することもできます。
- Open Server は、ログインを認証するために、SRV_CONNECT イベント・ハンドラ内からクライアントにチャレンジまたは暗号化パスワードを送信することを選択できます。
- 一度アプリケーションがネゴシエーション・ログイン・チャレンジまたは暗号化パスワードを送信すると、接続プロセスを継続する前にクライアントの応答を読み込まなければなりません。
- Open Server アプリケーションは、ログイン試行を認証するために必要なだけ、チャレンジまたは応答を繰り返すことができます。ただし、アプリケーションは、次のチャレンジを送信する前に、各チャレンジの応答を読み込まなければなりません。
- 一度ネゴシエーション・ログイン・チャレンジがクライアントに送信されると、アプリケーションは、応答を読み取らなければ接続プロセスを継続することはできません。
- Open Server アプリケーションは、いかなるタイプのチャレンジであっても srv_senddone の呼び出しで区切らなければなりません。アプリケーションが応答を読む前にいくつかのチャレンジの「バッチ」を発行する場合は、最後のバッチを除いて、各チャレンジの後に SRV_DONE_MORE の *status* 引数で srv_senddone を呼び出さなければなりません。バッチの最後のチャレンジの後に、SRV_DONE_FINAL の *status* 引数で srv_senddone を呼び出さなければなりません。
- アプリケーション定義のハンドシェイクにおいては、ハンドシェイクの種類を設定する (CS_SET) ため、またはクライアントが応答する返答のタイプを指定する (CS_GET) ために、Open Server アプリケーションは、*type* 引数を CS_USER_MSGID および CS_USER_MAX_MSGID の間の値に設定できます。Open Server アプリケーションが予期しない値を受信した場合は、Open Server はエラーを発行します。
- クライアントがチャレンジまたは暗号化パスワードに応答する場合は、srv_negotiate はクライアントの応答を受信するまでスレッドの実行を中断します。アプリケーションについては、セキュア SRV_CONNECT イベント・ハンドラをコーディングする場合は、このことを念頭に置いてください。

- ネゴシエーション・ログイン・チャレンジと応答は、srv_bind、srv_descfmt、srv_xferdata によって送受信されるパラメータを介してデータ値を運びます。この 3 つのルーチンは、ネゴシエーション・ログイン・データに対する定義またはアクセスを行うために SRV_NEGDATA の *type* 引数を受け取ります。
- 表 3-70 に、クライアントに送られる各チャレンジに伴うパラメータを示します。

表 *3-70: 必要なチャレンジ・パラメータ (srv_negotiate)*

ネゴシエーション・ログイン・タイプ	必要なパラメータ
SRV_NEG_CHALLENGE	パラメータは 1 つで、チャレンジ・データ値。データ型は、CS_DATAFMT *status* フィールドを CS_CANBENULL に設定した CS_BINARY_TYPE。
SRV_NEG_ENCRYPT	パラメータは 1 つで、暗号化キー・データ値。データ型は、CS_DATAFMT *status* フィールドを CS_CANBENULL に設定した CS_BINARY_TYPE。
SRV_NEG_SECLABEL	パラメータなし。
SRV_NEG_SECSESSION	セキュリティ・セッション・コールバックはパラメータの数とそのデータ・フォーマットを指定。詳細については、「セキュリティ・セッション・コールバック」(181 ページ) および『Open Client Client-Library/C リファレンス・マニュアル』を参照。
CS_USER_MSGID から CS_USER_MAX_MSGID の間の整数値	パラメータは 1 つで、アプリケーション定義のログイン・ハンドシェイク・データ値。

- 表 3-71 に、各ネゴシエーション・ログイン・チャレンジに対してクライアントから読み込むパラメータ示します。

表 *3-71: 予期されるチャレンジ・パラメータ (srv_negotiate)*

ネゴシエーション・ログイン・タイプ	存在するパラメータ
SRV_NEG_CHALLENGE	パラメータは 1 つで、チャレンジ応答データ。
SRV_NEG_LOCPWD	パラメータは 1 つで、暗号化ローカル・パスワード。
SRV_NEG_REMPWD	サーバ名／パスワードが対になっている可変数。
SRV_NEG_SECLABEL	次の 4 つのパラメータ。 Param 1：最大読み込みレベル・ラベル。 Param 2：最大書き込みレベル・ラベル。 Param 3：最小書き込みレベル・ラベル。 Param 4：現在の書き込みレベル・ラベル。

ネゴシエーション・ログイン・タイプ	存在するパラメータ
SRV_NEG_SECSESSION	セキュリティ・セッション・コールバックはパラメータの数とそのデータ・フォーマットを指定。詳細については、「セキュリティ・セッション・コールバック」(181 ページ) および『Open Client Client-Library/C リファレンス・マニュアル』を参照。
CS_USER_MSGID から CS_USER_MAX_MSGID の間の整数値	パラメータは 1 つで、アプリケーション定義のログイン・ハンドシェイク・データ値。

- パスワード暗号化チャレンジに対する応答である SRV_NEG_ENCRYPT は、パラメータの 2 つのセットから構成されることもあることに注意してください。SRV_NEG_LOCPWD 応答は、クライアントの暗号化パスワードを示すパラメータを渡します。また、クライアントは、クライアントの暗号化リモート・サーバ・パスワードとリモート・サーバ名をそれぞれ示すパラメータを渡す SRV_NEG_REMPWD 応答を送信することもできます。SRV_NEG_ENCRYPT チャレンジに対する SRV_NEG_LOCPWD 応答は、常に存在します。クライアントによってリモート・サーバ・パスワードがまったく送られていない場合には、SRV_NEG_REMPWD 応答を受ける要求は失敗します。
- Open Client と Open Server を使用してゲートウェイの機能を実装するアプリケーションは、クライアントとリモート・サーバ間においてネゴシエーション・ログイン・チャレンジと応答を転送するために、Open Client のネゴシエーション・ログイン・コールバック・メカニズムを使用しなければなりません。このようなアプリケーションでは、Open Client ネゴシエーション・ログイン・コールバックは、クライアントにチャレンジを転送してその応答を受信する (これを Open Client はリモート・サーバに返します) ために必要な Server-Library ルーチン呼び出しを含んでいなければなりません。

  ゲートウェイ・アプリケーションがクライアントとリモート・サーバ間のダイレクト・セキュリティ・セッションを確立する場合、Open Client セキュリティ・セッション・コールバックが必要です。このコールバックには、内部が隠されたセキュリティ・トークンをクライアントに転送してその応答を受信するために必要な Server-Library 呼び出しが含まれている必要があります。その後、この応答は Open Client によってリモート・サーバに返されます。詳細については、「セキュリティ・セッション・コールバック」(181 ページ) および『Open Client Client-Library/C リファレンス・マニュアル』を参照してください。

参照　srv_senddone、srv_thread_props

srv_numparams

説明

現在のクライアント・コマンドに含まれているパラメータの数を返します。

構文

```
CS_RETCODE srv_numparams(spp, numparamsp)
SRV_PROC    *spp;
CS_INT      *numparamsp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

numparamsp

現在のクライアント・コマンドまたはカーソル・データ・ストリームの引数の数へのポインタは、**numparamsp* で返されます。

戻り値

表 3-72: 戻り値 (srv_numparams)

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

```
#include  <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE    ex_srv_numparams PROTOTYPE((
SRV_PROC      *spp,
CS_INT        *countp
));

/*
** EX_SRV_NUMPARAMS
**
**    Example routine to illustrate the use of srv_numparams to
 **    get the number parameters contained in the current client
 **    command.
**
** Arguments:
**    spp    A pointer to an internal thread control structure.
**    countp A pointer to the buffer in which the number of
 **           parameters in the client command is returned.
**
** Returns:
**
**  CS_SUCCEED    The number of parameters was successfully
 **                returned.
**  CS_FAIL       An error was detected.
*/
CS_RETCODE        ex_srv_numparams(spp, countp)
SRV_PROC          *spp;
```

```
CS_INT            *countp;
{
     if (srv_numparams(spp, countp) == CS_FAIL)
        return (CS_FAIL);

    return(CS_SUCCEED);
}
```

使用法

- srv_numparams は、現在の MSG、RPC、DYNAMIC、カーソル・データ・ストリームのパラメータの数、または srv_negotiate (CS_GET) 呼び出しに対するクライアントの応答に含まれるパラメータの数を返します。この数は、ランタイム時に Open Server が満たすデフォルト・パラメータを含みます。
- srv_numparams は、特定のイベントのためのハンドラからしか呼び出すことができません。表 3-73 に、それらのイベントとそのパラメータを示します。

表 3-73: イベントとパラメータ *(srv_numparams)*

イベント	パラメータ
SRV_CURSOR	カーソル・パラメータ
SRV_RPC	RPC パラメータ
SRV_DYNAMIC	動的 SQL パラメータ
SRV_MSG	MSG パラメータ
SRV_LANGUAGE	言語パラメータ。言語ハンドラ内でパラメータ・データのチェックと取得を行うため、srv_numparams は 5.0 またはそれ以上の TDS レベルが要求される。接続の TDS レベルをチェックするコードをアプリケーションに追加し、その TDS バージョンが SRV_TDS_5_0 未満であれば、srv_numparams を省略する必要がある。srv_props ルーチンの SRV_S_TDSVERSION プロパティを使用すると、接続の TDS プロトコル・バージョンを取得できる (表 2-25 (133 ページ) を参照)。
srv_negotiate (CS_GET) 呼び出しの後	クライアントの応答内のパラメータ。たとえば、サンプル・プログラム *ctos.c* 内など。

参照

srv_bind、srv_cursor_props、srv_descfmt、srv_dynamic、srv_msg、srv_xferdata、「パラメータとロー・データの処理」(126 ページ)

srv_options

説明

オプション情報をクライアントに送信、またはクライアントから受信します。

構文

```
CS_RETCODE srv_options(spp, cmd, optcmdp, optionp,
   bufp, bufsize, outlenp)
SRV_PROC    *spp;
CS_INT      cmd;
CS_INT      *optcmdp;
CS_INT      *optionp;
CS_CHAR     *bufp;
CS_INT      bufsize;
CS_INT      *outlenp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

cmd

アプリケーションがオプション情報の送信または受信のどちらを行うために srv_options を呼び出しているかを示します。表 3-74 に、*cmd* の有効値を示します。

表 *3-74: cmd* の値 *(srv_options)*

値	説明
CS_SET	Open Server アプリケーションはクライアントにオプション・コマンドを送信している。
CS_GET	Open Server アプリケーションはクライアントからオプション・コマンドを受信している。

optcmdp

クライアントのオプション・コマンド (CS_GET) を含むプログラム変数、または Open Server アプリケーションのオプション・コマンド (CS_SET) を含むプログラム変数のいずれかへのポインタです。表 3-75 に、**optcmdp* の有効値を示します。

表 *3-75: optcmdp* の値 *(srv_options)*

値	説明	Cmd
SRV_SETOPTION	クライアントはオプションの設定を要求している。*optionp* と関連される値は、**bufp* で返される。Open Server は、返されるデータ・サイズのバイト数を *bufsize* に設定する。**bufp* がすべてのデータを保持できない場合は、この関数は CS_FAIL を返し、オプション値の実際のサイズは **outlenp* でバイト数を返し、*optionp* と *bufp* の値は未定義になる。	CS_GET
SRV_CLEAROPTION	クライアントは、*optionp* がデフォルト値に設定されることを要求している。*bufp* と *optionp* の値は未定義のままとなる。	CS_GET

値	説明	Cmd
SRV_GETOPTION	クライアントは、*optionp の現在の値に関する情報を要求している。*bufp* と *optionp* の値は未定義のままとなる。	CS_GET
SRV_SENDOPTION	アプリケーションは、SRV_GETOPTION コマンドに応答して現在のオプション値をクライアントに送っている。*bufp* はこのオプションに関連する引数を指し、*bufsize* は **bufp* にあるデータ・サイズのバイト数を持っている。	CS_SET

optionp

クライアントが要求したオプション (CS_GET) または Open Server アプリケーションが応答に使用しているオプション (CS_SET) のいずれかを指すポインタです。

bufp

オプション (CS_GET) に関連する値または要求者に送信されるオプション (CS_SET) の値のいずれかを持つバッファを指すポインタです。**optionp* は問題のオプションを含み、**bufp* はその値を含んでいます (CS_SET において)。オプションとその有効値の完全なリストについては、次の表を参照してください。

bufsize

**bufp* バッファの長さです。文字列オプション値を持つオプションを送信するときに、*bufp* が null で終了している場合は、*bufsize* を CS_NULLTERM として転送してください。

outlenp

**bufp* で返されているオプション値のサイズのバイト数に設定されたプログラム変数へのポインタです。このパラメータはオプションで、*cmd* が CS_GET に設定された場合にのみ使用されます。

戻り値

表 *3-76:* 戻り値 *(srv_options)*

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

```
#include  <ospublic.h>

/*
** Local Prototype
*/
CS_RETCODE    ex_srv_options PROTOTYPE((
SRV_PROC      *spp,
CS_INT        *rowcount
));

/*
** EX_SRV_OPTIONS
**
**    Example routine to recieve option information for the
 **    maximum number of regular rows to return (CS_OPT_ROWCOUNT)
 **    from a client.
**
** Arguments:
**   spp       A pointer to an internal thread control structure.
**   rowcount  Return pointer for the number of rows to return.
**
** Returns:
**
**    CS_SUCCEED   Successfully retrieved option.
**    CS_FAIL      An error was detected.
*/
CS_RETCODE       ex_srv_options(spp, rowcount)
SRV_PROC         *spp;
CS_INT           *rowcount;
{
    CS_INT       optcmdp;    /* The client's option command. */
    CS_INT       optionp;    /* The client's option request. */

    /* Initialization. */
    optcmdp = SRV_GETOPTION;
    optionp = CS_OPT_ROWCOUNT;

    /*
    ** Get the maximum number of rows to return.
    */
    if (srv_options(spp, CS_GET, &optcmdp, &optionp, (CS_VOID
         *)rowcount, CS_SIZEOF(CS_INT), (CS_INT *)NULL) !=
         CS_SUCCEED)
    {
        return(CS_FAIL);
    }
    return(CS_SUCCEED);
}
```

使用法

- srv_options は Open Server アプリケーションがクライアントからオプション情報を読み取ったり、クライアントにオプション情報を送信したりすることを可能にします。
- 表 3-77 に、有効なオプション、その有効値、および *optionp* パラメータのデータ型を示します。

表 *3-77:* オプションの説明 *(srv_options)*

オプション	有効値	bufp が指す対象
CS_OPT_ANSINULL	CS_TRUE、CS_FALSE	CS_BOOL
CS_OPT_ANSIPERM	CS_TRUE、CS_FALSE	CS_BOOL
CS_OPT_ARITHABORT	CS_TRUE、CS_FALSE	CS_BOOL
CS_OPT_ARITHIGNORE	CS_TRUE、CS_FALSE	CS_BOOL
CS_OPT_AUTHOFF	CS_OPT_SA、 CS_OPT_SSO、 CS_OPT_OPER	文字列
CS_OPT_AUTHON	CS_OPT_SA、 CS_OPT_SSO、 CS_OPT_OPER	文字列
CS_OPT_CHAINXACTS	CS_TRUE、CS_FALSE	CS_BOOL
CS_OPT_CURCLOSEONXACT	CS_TRUE、CS_FALSE	CS_BOOL
CS_OPT_CURREAD	読み込みラベル (文字列)	文字列
CS_OPT_CURWRITE	書き込みラベル (文字列)	文字列
CS_OPT_DATEFIRST	CS_OPT_SUNDAY CS_OPT_MONDAY CS_OPT_TUESDAY CS_OPT_WEDNESDAY CS_OPT_THURSDAY CS_OPT_FRIDAY CS_OPT_SATURDAY	週の最初の日に使用する曜日を示す記号値
CS_OPT_DATEFORMAT	CS_OPT_FMTMDY CS_OPT_FMTDMY CS_OPT_FMTYMD CS_OPT_FMTYDM CS_OPT_FMTMYD CS_OPT_FMTDYM	日時値で使用する年、月、日の順序を表す記号値
CS_OPT_FIPSFLAG	CS_TRUE、CS_FALSE	CS_BOOL
CS_OPT_FORCEPLAN	CS_TRUE、CS_FALSE	CS_BOOL
CS_OPT_FORMATONLY	CS_TRUE、CS_FALSE	CS_BOOL
CS_OPT_GETDATA	CS_TRUE、CS_FALSE	CS_BOOL
CS_OPT_IDENTITYOFF	テーブル名を表す文字列値	文字列
CS_OPT_IDENTITYON	テーブル名を表す文字列値	文字列
CS_OPT_ISOLATION	CS_OPT_LEVEL1 CS_OPT_LEVEL3	分離レベルを表す記号値
CS_OPT_NOCOUNT	CS_TRUE、CS_FALSE	CS_BOOL
CS_OPT_NOEXEC	CS_TRUE、CS_FALSE	CS_BOOL

オプション	有効値	bufp が指す対象
CS_OPT_PARSEONLY	CS_TRUE、CS_FALSE	CS_BOOL
CS_OPT_QUOTED_IDENT	CS_TRUE、CS_FALSE	CS_BOOL
CS_OPT_RESTREES	CS_TRUE、CS_FALSE	CS_BOOL
CS_OPT_ROWCOUNT	返す通常ローの最大数	CS_INT 0 が指定されるとすべてのローが返される。
CS_OPT_SHOWPLAN	CS_TRUE、CS_FALSE	CS_BOOL
CS_OPT_STATS_IO	CS_TRUE、CS_FALSE	CS_BOOL
CS_OPT_STATS_TIME	CS_TRUE、CS_FALSE	CS_BOOL
CS_OPT_STR_RTRUNC	CS_TRUE、CS_FALSE	CS_BOOL
CS_OPT_TEXTSIZE	サーバが返す最長の text または image の値	CS_INT
CS_OPT_TRUNCIGNORE	CS_TRUE、CS_FALSE	CS_BOOL

「オプション」(115 ページ) には、各オプションの説明とそれぞれのデフォルト値を示しています。

- Open Server はクライアントから受信した各オプション・コマンドに対して、SRV_OPTION イベントを発生させます。SRV_OPTION イベント・ハンドラ内で、アプリケーションはオプション情報を取得するために *cmd* を CS_GET に設定して srv_options を呼び出すことができます。srv_options が戻る場合、*optcmdp*、*optionp*、**bufp* にはクライアントから受け取ったオプション情報がすべて格納されています。SRV_OPTION イベント・ハンドラ以外のイベント・ハンドラから srv_options を呼び出すと、エラーになります。
- SRV_SETOPTION と SRV_CLEAROPTION への応答では、アプリケーションは引数を SRV_DONE_FINAL に指定した srv_senddone を呼び出さなければなりません。オプション処理が失敗した場合には、アプリケーションは、引数を SRV_DONE_FINAL | SRV_DONE_ERROR に指定した srv_senddone を呼び出さなければなりません。
- アプリケーションは SRV_GETOPTION コマンドを受け取るたびに、*optcmdp* を SRV_SENDOPTION に設定し、*bufp* がオプションの現在の値を指すようにして srv_options を呼び出さなければなりません。
- アプリケーションは、クライアントが SRV_SETOPTION コマンドを使用して送信した引数を受け取るために十分なサイズの **bufp* バッファを確保する必要があります。バッファが十分な大きさではない場合、srv_options は CS_FAIL を返し、*outlenp* は必要なサイズに設定されます。
- Open Server には、特定のオプションの意味に関する概念がまったくありません。Open Server アプリケーションは、クライアントのオプション・コマンドを保存し、コマンドが要求するアクションを実行します。SRV_OPTION イベント・ハンドラがインストールされていない場合は、クライアントから受けたオプション・コマンドはエラーとなり、拒否されます。

参照

srv_senddone、「オプション」(115 ページ)

srv_orderby

説明

クライアントに order-by リストを返します。

構文

CS_RETCODE srv_orderby(spp, numcols, collistp)

```
SRV_PROC    *spp;
CS_INT       numcols;
CS_INT      *collistp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

numcols

order-by リストのカラムの数です。カラムは CS_INT の配列として渡されるので、*numcols* は、実際には *collistp* 配列の要素の数です。

collistp

カラム番号の配列へのポインタです。この配列のサイズは、*numcols* です。

戻り値

表 *3-78:* 戻り値 *(srv_orderby)*

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

```
#include  <ospublic.h>
/*
** Local Prototype
*/
CS_RETCODE   ex_srv_orderby PROTOTYPE((
SRV_PROC     *spp
));

/*
** EX_SRV_ORDERBY
**
**    Example routine using srv_orderby to define and return to a
 **    client application the order-by list for a simple SQL
 **    command.
**    This example uses the SQL command:
**
**        "select a,b,c,d from my_tab
 **         order by c,a"
**
** Arguments:
**    spp    A pointer to the internal thread control structure.
**
** Returns:
**    CS_SUCCEED      Order-by list was successfully defined.
**    CS_FAIL         An error was detected..
*/
```

```
CS_RETCODE            ex_srv_orderby(spp)
SRV_PROC              *spp;
{
    /* There are two columns specified in the order-by clause. */
    CS_INT        collist[2];
    CS_INT        numcols;

    /* Initialization. */
    numcols = 2;

    /*
    ** Initialize the collist array in the order the
     ** columns occur in the order-by clause.
    **
    ** "c" is the 1st column specified in the order-by,
     ** and is the 3rd column specified in the select-list.
    */
    collist[0] = (CS_INT)3;
    /*
    ** "a" is the 2nd column specified in the order-by,
     ** and is the 1st column specified in the select-list.
    */
    collist[1] = (CS_INT)1;
    /*
    ** Define the order-by list.
    */
    if (srv_orderby(spp, numcols, collist) != CS_SUCCEED)
    {
        return(CS_FAIL);
    }
    return(CS_SUCCEED);
}
```

使用法

- order-by の情報を返すという Adaptive Server Enterprise の機能を模擬的に行うときにだけ、srv_orderby が必要になります。
- srv_orderby では、Open Server アプリケーションがソート順に関する情報をクライアントに返すことができます。SQL コマンドは、次のようになります。

  ```
  select a, b, c, d
   order by c, a
  ```

  ソート順は、先にカラム c、次にカラム a です。アプリケーションは、カラム番号配列において、まずカラム 3、次にカラム 1 をリストすることによって、この情報をクライアントに返します。
- select リストの先頭のカラムは、カラム 1 です。
- srv_orderby は、srv_descfmt の呼び出しの後、srv_bind の呼び出しの前に呼び出さなければなりません。

srv_poll (UNIX のみ)

説明

一連のオープン・ストリームのファイル記述子の I/O イベントをチェックします。

構文

```
CS_INT srv_poll(fdsp, nfds, waitflag)
SRV_POLLFD      *fdsp;
CS_INT          nfds;
CS_INT          waitflag;
```

パラメータ

fdsp

対象の各オープン・ファイル記述子に対して要素を 1 つずつ持っている SRV_POLLFD 構造体の配列を指すポインタです。SRV_POLLFD 構造体には、次のメンバがあります。

```
CS_INT   srv_fd;          /* File descriptor. */
 CS_INT   srv_events;      /* Relevant events. */
 CS_INT   srv_revents;     /* Returned events. */
```

nfds

**fdsp* 配列の要素の数です。

waitflag

目的のオペレーションのためのファイル記述子が使用可能になるまで、スレッドが休止されるのかどうかを示す CS_INT 値です。SRV_M_WAIT に設定すると、スレッドは休止し、指定されたオペレーションに対して **fdsp* 配列のファイル記述子 (いずれでも) が使用可能になると、ウェイクアップします。フラグが SRV_M_NOWAIT に設定されていると、srv_poll はチェックを行い、呼び出し元に戻ります。リターン・ステータスの値がゼロより大きい場合は、目的のオペレーションのファイル記述子が使用可能であったことを示します。

戻り値

表 *3-79:* 戻り値 *(srv_poll)*

戻り値	意味
整数	使用可能なファイル記述子の数。
-1	ルーチンが失敗した。
0	使用可能なファイル記述子がない。

例

```
#include   <ospublic.h>
/*
** Local Prototype
*/
CS_RETCODE      ex_srv_pollPROTOTYPE((
struct pollfd   *fdp,
CS_INT          nfds
));

/*
** EX_SRV_POLL
**
**   This routine demonstrates how to use srv_poll to poll
**   application-specific file descriptors.
**
** Arguments:
**         fdp - The address of the file descriptor array.
**         nfds - The number of file descriptors to poll.
**
** Returns
**
**    CS_SUCCEED    If the data address is returned.
**    CS_FAIL       If the call to srv_poll failed.
**
*/
CS_RETCODE      ex_srv_poll(fdp, nfds)
struct pollfd   *fdp;
CS_INT          nfds;
{
      /*
      ** Initialization.
      */
      lp = (CS_VOID *)NULL;

      /*
      ** Calls srv_poll to check if any of these file
      ** descriptors are active; ask to sleep until at
      ** least one of them is.
      */
      if( srv_poll(fdp, nfds, SRV_M_WAIT) == (CS_INT)-1 )
      {
          return CS_FAIL;
      }

      /*
      ** All done.
      */
      return CS_SUCCEED;
}
```

使用法

- アプリケーションはファイル記述子を調べるため、または実行する I/O があるまでスレッドを休止するために、srv_poll を使用することができます。
- 表 3-80 に、*srv_events* と *srv_revents* の有効値を示します。

表 3-80: srv_events および revents の値 (srv_poll)

値	説明
SRV_POLLIN	通常の読み込みイベント。
SRV_POLLPRI	優先イベントを受け取った。
SRV_POLLOUT	ファイル記述子は書き込みが可能。
SRV_POLLERR	ファイル記述子でエラーが起こった。
SRV_POLLHUP	ファイル記述子でハングした。この値は、リターン・イベントでのみ有効。
SRV_POLLNVAL	SRV_POLLFD で無効なファイル記述子が指定された。

- srv_poll は、すべての UNIX プラットフォームで使用できます。

注意　ネイティブの poll(2) システム・コールをサポートする UNIX プラットフォームでアプリケーションが srv_poll を使用する場合には、アプリケーションは *ospublic.h* の前に <*sys/poll.h*> を含む必要があります。

参照

srv_capability、srv_select (UNIX のみ)

srv_props

説明

Open Server プロパティを定義または取得します。

構文

```
CS_RETCODE srv_props(cp, cmd, property, bufp, buflen,
        outlenp)
CS_CONTEXT      *cp;
CS_INT          cmd;
CS_INT          property;
CS_VOID          *bufp;
CS_INT          buflen;
CS_INT          *outlenp;
```

パラメータ

scp

cs_ctx_alloc を使用して先に割り付けた CS_CONTEXT 構造体へのポインタです。

cmd

実行するアクションです。表 3-81 に、*cmd* の有効値を示します。

表 *3-81: cmd* の値 *(srv_props)*

値	意味
CS_SET	Open Server アプリケーションがプロパティを設定している。この場合、*bufp* は、プロパティが設定される値を持ち、*buflen* は、その値のサイズのバイト数でなければならない。
CS_GET	Open Server アプリケーションがプロパティを取得している。この場合、*bufp* は、プロパティ値が置かれているバッファを指し、*buflen* は、そのバッファのサイズのバイト数でなければならない。
CS_CLEAR	Open Server アプリケーションは、プロパティをデフォルト値にリセットしている。この場合は、*bufp*、*buflen*、*outlenp* は無視される。

property

設定、取得、またはクリアされるプロパティです。この引数の有効値のリストについては、「使用法」にある表を参照してください。

bufp

プロパティ値の情報が設定される (CS_SET)、またはプロパティ値の情報が取得される (CS_GET) Open Server アプリケーションのデータ・バッファへのポインタです。

buflen

バッファ長をバイト数で示したものです。

outlenp

取得されたプロパティ値の長さのバイト数を設定するために Open Server が使用する CS_INT 変数へのポインタです。この引数はオプションであり、*cmd* が CS_GET のときにのみ使用します。

戻り値

表 *3-82:* 戻り値 *(srv_props)*

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

```
#include<ospublic.h>
/*
** Local prototype
*/
CS_RETCODE ex_srv_set_propPROTOTYPE((
CS_CONTEXT *cp,
CS_INT     property,
CS_VOID    *bufp,
CS_INT     buflen
));
/*
** EX_SRV_SET_PROP
**
**    Example routine to set a property using srv_props.
**
** Arguments:
**
**    *cp       Pointer to a CS_CONTEXT structure previously
**              allocated by cs_ctx_alloc.
**    property  The property being set.
**    *bufp     Pointer to the value the property is to be
**              set to.
**    buflen    The length of the value.
**
** Returns
**
**   CS_SUCCEED   Arguments were valid and srv_props was called.
**   CS_FAIL      An error was detected.
**
*/
CS_RETCODE      ex_srv_set_prop(cp, property, bufp, buflen)
CS_CONTEXT      *cp;
CS_INT          property;
CS_VOID         *bufp;
CS_INT          buflen;
{
      /* Check arguments. */
      if(cp == (CS_CONTEXT *)NULL)
      {
            return(CS_FAIL);
      }
      if(buflen < 1)
            return(CS_FAIL);
return(srv_props(cp,(CS_INT)CS_SET,property,bufp,buflen,
  (CS_INT *)0));
}
```

使用法

- サーバワイドな設定パラメータとプロパティを定義および取得するために、srv_props が呼び出されます。
- srv_version を呼び出してから、srv_props を呼び出します。
- srv_init を呼び出す前に、SRV_S_TRACEFLAG、SRV_S_LOGFILE、SRV_S_TRUNCATELOG を除くすべてのプロパティが srv_props によって設定されていなければなりません。
- srv_init を呼び出した後、*bufp* を空の文字列 ("") に設定し、*buflen* をゼロに設定した状態で SRV_S_LOGFILE プロパティを設定すると、ログ・ファイルが閉じます。
- 表 3-83 に、サーバ・プロパティ値、設定や取得の可／不可、それぞれのプロパティ値のデータ型を示します。

表 *3-83:* サーバのプロパティとそのデータ型 *(srv_props)*

プロパティ	設定／クリア	取得	cmd が CS_SET のときの bufp	cmd が CS_GET のときの bufp
SRV_S_ALLOCFUNC	可	可	関数ポインタ	関数ポインタのアドレス
SRV_S_APICHK	可	可	CS_BOOL	CS_BOOL
SRV_S_ATTNREASON	不可	可	適用しない	CS_INT
SRV_S_CERT_AUTH	可	可	char*	char*
SRV_S_CURTHREAD	不可	可	適用しない	スレッド・ポインタのアドレス
SRV_S_DISCONNECT	可	可	CS_BOOL	CS_BOOL
SRV_S_DEFQUEUESIZE	可	可	CS_INT	CS_INT
SRV_S_DS_PROVIDER	可	可	文字列へのポインタ	文字列へのポインタ
SRV_S_DS_REGISTER	可	可	CS_BOOL	CS_BOOL
SRV_S_ERRHANDLE	可	可	関数ポインタ	関数ポインタのアドレス
SRV_S_FREEFUNC	可	可	関数ポインタ	関数ポインタのアドレス
SRV_S_IFILE	可	可	文字列	文字列
SRV_S_LOGFILE	可	可	文字列	文字列
SRV_S_LOGSIZE	可	可	CS_INT	CS_INT
SRV_S_MAXLISTENERS	可	可	CS_INT	CS_INT
SRV_S_MSGPOOL	可	可	CS_INT	CS_INT
SRV_S_NETBUFSIZE	可	可	CS_INT	CS_INT
SRV_S_NETTRACEFILE	可	可	文字列	文字列
SRV_S_NUMCONNECTIONS	可	可	CS_INT	CS_INT
SRV_S_NUMLISTENERS	不可	可	CS_INT	CS_INT
SRV_S_NUMMSGQUEUES	可	可	CS_INT	CS_INT
SRV_S_NUMMUTEXES	可	可	CS_INT	CS_INT
SRV_S_NUMREMBUF	可	可	CS_INT	CS_INT

プロパティ	設定／クリア	取得	cmd が CS_SET のときの bufp	cmd が CS_GET のときの bufp
SRV_S_NUMREMSITES	可	可	CS_INT	CS_INT
SRV_S_NUMTHREADS	可	可	CS_INT	CS_INT
SRV_S_NUMUSEREVENTS	可	可	CS_INT	CS_INT
SRV_S_PREEMPT	可	可	CS_BOOL	CS_BOOL
SRV_S_REALLOCFUNC	可	可	関数ポインタ	関数ポインタのアドレス
SRV_S_RETPARMS	可	可	CS_BOOL	CS BOOL
SRV_S_REQUESTCAP	可	可	CS_CAP_TYPE 構造体	CS_CAP_TYPE 構造体
SRV_S_RESPONSECAP	可	可	CS_CAP_TYPE 構造体	CS_CAP_TYPE 構造体
SRV_S_SEC_KEYTAB	可	可	文字列へのポインタ	文字列へのポインタ
SRV_S_SEC_PRINCIPAL	可	可	文字列へのポインタ	文字列へのポインタ
SRV_S_SERVERNAME	不可	可	文字列	文字列
SRV_S_SSL_CIPHER	可	不可	char*	
SRV_S_SSL_LOCAL_ID	可	可	struct	char*
SRV_S_SSL_VERSION	可	不可	CS_INT	
SRV_S_STACKSIZE	可	可	CS_INT	CS_INT
SRV_S_TDSVERSION	可	可	CS_INT	CS_INT
SRV_S_TIMESLICE	可	可	CS_INT	CS_INT
SRV_S_TRACEFLAG	可	可	CS_INT (ビットマスク)	CS_INT (ビットマスク)
SRV_S_TRUNCATELOG	可	可	CS_BOOL	CS_BOOL
SRV_S_USESRVLANG	可	可	CS_BOOL	CS_BOOL
SRV_S_VERSION	不可	可	適用しない	文字列
SRV_S_VIRTCLKRATE	可	可	CS_INT	CS_INT
SRV_S_VIRTTIMER	可	可	CS_BOOL	CS_BOOL

- 表 3-84 に、各サーバ・プロパティのデフォルト値を示します。

表 *3-84:* プロパティの有効値とそのデフォルト値 *(srv_props)*

プロパティ	デフォルト
SRV_S_ALLOCFUNC	malloc()
SRV_S_APICHK	CS_TRUE
SRV_S_ATTNREASON	デフォルトなし
SRV_S_CURTHREAD	適用せず
SRV_S_DEFQUEUESIZE	SRV_DEF_DEFQUEUESIZE
SRV_S_DISCONNECT	CS_FALSE
SRV_S_DS_PROVIDER	プラットフォームに依存する。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。
SRV_S_DS_REGISTER	CS_TRUE、Server-Library が起動時に自身でディレクトリに登録する
SRV_S_ERRHANDLE	エラー・ハンドラなし
SRV_S_FREEFUNC	free()
SRV_S_IFILE	*$SYBASE/interfaces*
SRV_S_LOGFILE	*srv.log*
SRV_S_LOGSIZE	最大整数値
SRV_S_MAXLISTENERS	CS_MAX_NOMAX
SRV_S_MSGPOOL	SRV_DEF_MSGPOOL
SRV_S_NETBUFSIZE	SRV_DEF_NETBUFSIZE
SRV_S_NETTRACEFILE	*sybnet.dbg*
SRV_S_NUMCONNECTIONS	SRV_DEF_NUMCONNECTIONS
SRV_S_NUMLISTENERS	適用せず
SRV_S_NUMMSGQUEUES	SRV_DEF_NUMMSGQUEUES
SRV_S_NUMMUTEXES	SRV_DEF_NUMMUTEXES
SRV_S_NUMREMBUF	SRV_DEF_NUMREMBUF
SRV_S_NUMREMSITES	SRV_DEF_NUMREMSITES
SRV_S_NUMTHREADS	SRV_DEF_NUMTHREADS
SRV_S_NUMUSEREVENTS	SRV_DEF_NUMUSEREVENTS
SRV_S_PREEMPT	CS_FALSE
SRV_S_REALLOCFUNC	realloc()
SRV_S_REQUESTCAP	詳細については、「機能」(22 ページ) を参照。
SRV_S_RESPONSECAP	詳細については、「機能」(22 ページ) を参照。
SRV_S_RETPARMS	デフォルトなし
SRV_S_SEC_KEYTAB	デフォルトなし
SRV_S_SEC_PRINCIPAL	セキュリティ・メカニズムに依存する。
SRV_S_SERVERNAME	DSLISTEN 環境変数
SRV_S_STACKSIZE	SRV_DEF_STACKSIZE
0	SRV_TDS_5_0

プロパティ	デフォルト
SRV_S_TIMESLICE	SRV_DEF_TIMESLICE
SRV_S_TRACEFLAG	0
SRV_S_TRUNCATELOG	CS_FALSE
SRV_S_USESRVLANG	CS_TRUE
SRV_S_VERSION	コンパイル時のバージョン文字列
SRV_S_VIRTCLKRATE	SRV_DEF_VIRTCLKRATE
SRV_S_VIRTTIMER	CS_FALSE

- デフォルトを持っていて設定可能なすべてのサーバ・プロパティは、*cmd* を CS_CLEAR に設定して srv_props を呼び出すことによって、デフォルト値にリセットできます。
- すべてのサーバ・プロパティは、*cmd* を CS_GET に設定して srv_props を呼び出すことによって、いつでも取得できます。Open Server アプリケーションがプロパティの値を定義していない場合は、デフォルト値が返されます。
- プロパティについては、「プロパティ」のページを参照してください。
- プロパティが取得されているときに、ユーザ・バッファのサイズがプロパティ値を格納するには十分のサイズではないことを *buflen* が示した場合には、Open Server は必要なバイト数を **outlenp* に設定し、ユーザ・バッファは変更されません。
- デフォルトの stacksize (SRV_S_STACKSIZE のデフォルト値) は、使用するプラットフォームに応じて異なります。

  Open Server のネイティブスレッド・バージョンでは、基本となるスレッドのデフォルトの stacksize が使用されます。この値は、SRV_S_STACKSIZE プロパティで stacksize を設定することにより変更できます。

  stacksize を設定する場合、その stacksize が小さすぎるときは、スタック・オーバフロー・エラーが発生する場合があることに注意してください。

参照　srv_init、srv_thread_props, srv_spawn、「プロパティ」(130 ページ)

srv_putmsgq

説明

メッセージ・キューにメッセージを入れます。

構文

```
CS_RETCODE srv_putmsgq(msgqid, msgp, putflags)
SRV_OBJID       msgqid;
CS_VOID         *msgp;
CS_INT          putflags;
```

パラメータ

msgqid

メッセージ・キューの識別子です。メッセージ・キューを名前で参照する場合は、srv_getobjid を呼び出して、名前を検索しメッセージ・キュー ID を返します。

msgp

メッセージへのポインタです。メッセージ・データは、受信されて処理されるまでは有効でなければなりません。

putflags

putflags の値は、論理和をとることができます。表 3-85 に、値とその意味を示します。

表 *3-85: putflags* の値 *(srv_putmsgq)*

値	説明
SRV_M_NOWAIT	このフラグが設定されると、メッセージがメッセージ・キューに置かれた直後に、srv_putmsgq の呼び出しが戻る。
SRV_M_WAIT	このフラグが設定されると、メッセージが読み込まれるかキューが削除されるまでは srv_putmsgq は戻らない。
SRV_M_URGENT	このフラグが設定されると、メッセージは、メッセージ・キューのメッセージ・リストの最後ではなく先頭に置かれる。1 つのキューに複数の緊急メッセージが追加された場合には、緊急メッセージは、キューに入った順でキューの先頭に表示される。

戻り値

表 *3-86:* 戻り値 *(srv_putmsgq)*

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

```
#include  <ospublic.h>

/*
 ** Local Prototype.
 */
CS_RETCODE ex_srv_putmsgq PROTOTYPE((
SRV_OBJID  mqid,
CS_INT     flags
));
/*
```

```
 ** EX_SRV_PUTMSGQ
**
**    Example routine to put a message into a message queue.
**
** Arguments:
**   msgqid    Message queue identifier.
**   putflags  Special instructions for srv_putmsgq.
**
** Returns:
**
**   CS_SUCCEED
**   CS_FAIL
 */
CS_RETCODE      ex_srv_putmsgq(mqid, flags)
SRV_OBJID       mqid;
CS_INT          flags;
{
     CS_CHAR    *msgp;
    msgqp = srv_alloc(20);
     strcpy(msgp, "Hi there");
     return(srv_putmsgq(mqid, msgp, flags));
}
```

使用法

- srv_putmsgq は、**msgp* のメッセージをメッセージ・キュー *msgqid* に入れます。
- メッセージは、ポインタとして渡されます。仮にメッセージを送信しているスレッドのコンテキストが変更されても、メッセージが指すデータは有効のままでなければなりません。

  特に、メッセージを送信するスレッドのコンテキストにあるスタック・アドレスを指すメッセージを転送するときには、注意してください。これを行う場合には、メッセージがキューから削除されるまでは、メッセージを送ったスレッドがメッセージを送ったフレームから戻らないことを保証しなければなりません。この保証がないと、メッセージは他の目的に使用されているスタックを指すこともあります。
- SRV_S_NUMMSGQUEUES サーバ・プロパティは、Open Server アプリケーションが使用できるメッセージ・キューの数を決定します。「サーバ・プロパティ」(132 ページ) を参照してください。
- SRV_S_MSGPOOL サーバ・プロパティは、ランタイムに Open Server アプリケーションが使用できるメッセージの数を決定します。「サーバ・プロパティ」(132 ページ) を参照してください。

参照

srv_createmsgq、srv_deletemsgq、srv_getmsgq、srv_getobjid

srv_realloc

説明

メモリの再割り付けを行います。

構文

```
CS_VOID* srv_realloc(mp,newsize)
CS_VOID        *mp;
CS_INT         newsize;
```

パラメータ

mp

古いメモリ・ブロックへのポインタです。

newsize

再割り付けするバイト数です。

戻り値

表 *3-87: 戻り値 (srv_realloc)*

戻り値	意味
新しく割り付けられた領域へのポインタ	新しい領域のロケーション。
null ポインタ	Server-Library は *newsize* バイトの割り付けができなかった。

例

```
#include   <ospublic.h>

/*
** Local Prototype.
*/
extern CS_RETCODE ex_srv_realloc PROTOTYPE((
CS_VOID       *mp,
CS_INT       newsize
));

/*
** EX_SRV_REALLOC
**
**   Reallocate a memory chunk.
**
** Arguments:
**    mp        A pointer to existing memory block.
**    newsize   The new size of the memory block.
**
** Returns:
**    CS_SUCCEED   Memory was allocated successfully.
**    CS_FAIL      An error was detected.
*/
CS_RETCODE      ex_srv_realloc(mp, newsize)
CS_VOID         *mp;
CS_INT          newsize;
{
      mp = srv_realloc(mp, newsize);
```

```
        if(mp == (CS_VOID *)NULL)
        {
             return(CS_FAIL);
        }
        return(CS_SUCCEED);
    }
```

使用法

- srv_realloc は、動的にメモリの再割り付けを行います。
- *mp* で参照されているブロックのサイズを *newsize* に変更し、(移動された可能性もある) ブロックへのポインタを返します。
- srv_realloc を使用して割り付けたメモリは、すべて srv_free を呼び出すことで解放しなければなりません。
- 通常の C 言語のメモリ割り付けルーチンが使用されるようなところで、srv_realloc を使用します。
- 現行バージョンでは、srv_realloc は、C 言語のルーチンの *realloc* を呼び出します。しかし、Open Server アプリケーションでは、srv_props ルーチンを使用して独自のメモリ管理ルーチンをインストールすることができます。ユーザ・インストール・ルーチンのパラメータ転送規則は、*realloc* のものと同一でなければなりません。アプリケーションがユーザ・インストール・ルーチンを使用できるように設定されていない場合には、Open Server は *realloc* を呼び出します。

参照　srv_alloc、srv_free、srv_props

srv_recvpassthru

説明　クライアントからプロトコル・パケットを受信します。

構文

```
CS_RETCODE srv_recvpassthru(spp, recv_bufp, infop)
SRV_PROC        *spp;
CS_BYTE         **recv_bufp;
CS_INT          *infop;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

recv_bufp

受信したプロトコル・パケットを含んでいるバッファの開始アドレスを受ける CS_BYTE ポインタへのポインタです。

infop

srv_recvpassthru が CS_FAIL を返した場合に SRV_I_UNKNOWN に設定される CS_INT へのポインタです。表 3-88 は、srv_recvpassthru が CS_SUCCEED を返す場合に **infop* に返される可能性のある値を示します。

表 3-88: CS_SUCCEED の値 (srv_recvpassthru)

値	説明
SRV_I_PASSTHRU_MORE	プロトコル・パケットを読むことに成功。これはメッセージ・パケットの終わりではない。
SRV_I_PASSTHRU_EOM	このパケットは、メッセージ・パケットの終わりである。

戻り値

表 3-89: 戻り値 (srv_recvpassthru)

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

```
#include      <ospublic.h>
/*
 ** Local prototype.
 */
CS_RETCODE      ex_srv_recvpassthru PROTOTYPE((
CS_VOID         *spp
));
/*
** EX_SRV_RECVPASSTHRU
**
**   Example routine to receive protocol packets from a client.
**
** Arguments:
**   spp    A pointer to an internal thread control structure.
**
** Returns:
**   CS_SUCCEED If we were able to receive the packets.
**   CS_FAIL If were unsuccessful at receiving the packets.
**
*/
CS_RETCODE    ex_srv_recvpassthru(spp)
SRV_PROC    *spp;
{
     CS_RETCODE    result;
     CS_BYTE       *recvbuf;
     CS_INT        info;
```

```
        /*
        ** Read packets until we get the EOM flag.
        */
        do
        {
                result = srv_recvpassthru(spp, &recvbuf, &info);
        }
        while (result == CS_SUCCEED && info == SRV_I_PASSTHRU_MORE);

        return (result);
    }
```

使用法

- srv_recvpassthru は、プロトコル・パケットを内容を解釈せずに受け取ります。
- 一度 srv_recvpassthru が呼び出されると、これを呼び出したイベント・ハンドラは「パススルー・モード」に入ったことになります。パススルー・モードは、**infop* に SRV_I_PASSTHRU_EOM が返されるときに終了します。
- イベント・ハンドラがパススルー・モード状態の場合は、他の Server-Library ルーチンを呼び出すことはできません。
- パススルー・モードでは、クライアントのための SRV_CONNECT ハンドラは、srv_getloginfo、ct_setloginfo、ct_getloginfo、srv_setloginfo を呼び出すことによって、クライアントとリモート・サーバがプロトコル・パケットをネゴシエートできるようにしなければなりません。これによって、異なるプラットフォーム上で動作しているクライアントおよびリモート・サーバは、必要なデータ変換を行うことができます。
- srv_recvpassthru は、SRV_START、SRV_CONNECT、SRV_STOP、SRV_DISCONNECT、SRV_URGDISCONNECT、SRV_ATTENTION 以外のすべてのイベント・ハンドラで呼び出すことができます。
- 一度 srv_recvpassthru を呼び出すと、実行中のスレッドは srv_senddone を発行するまで、I/O を実行するルーチンを呼び出すことができなくなります。ネットワーク I/O は、srv_recvpassthru を実行するイベント・ハンドラで発生できません。

参照　srv_getloginfo、srv_sendpassthru、srv_setloginfo

srv_regcreate

説明

レジスタード・プロシージャの登録を完了します。

構文

```
CS_RETCODE srv_regcreate(spp, infop)
SRV_PROC        *spp;
CS_INT          *infop;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

infop

CS_INT を指すポインタです。表 3-90 に、srv_regcreate が CS_FAIL を返す場合に **infop* に返される可能性がある値を示します。

表 *3-90: infop の値 (srv_regcreate)*

値	説明
SRV_I_PEXISTS	プロシージャはすでに登録されている。
SRV_I_UNKNOWN	その他のエラーが発生した。

戻り値

表 *3-91: 戻り値 (srv_regcreate)*

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

```
#include  <ospublic.h>

/*
** Local Prototype.
*/
CS_INT    ex_srv_regcreate PROTOTYPE((
SRV_PROC  *sproc
));

/*
** EX_SRV_REGCREATE
**    An example routine that completes the registration of a
**    registered procedure using srv_regcreate.
**
** Arguments:
**    sproc  A pointer to an internal thread control structure.
**
** Returns:
**    CS_SUCCEED   If the procedure was registered successfully.
**    CS_FAIL      If the supplied internal control structure is
 **                NULL.
**    SRV_I_EXIST       If the procedure is al&ready registered.
**    SRV_I_UNKNOWN     If some other error occurred.
*/
```

```
CS_INT    ex_srv_regcreate(sproc)
SRV_PROC  *sproc;
{
      CS_INT    info;     /* The reason for failure */

      /*
      ** Check whether the internal control structure is NULL.
      */
      if ( sproc == (SRV_PROC *)NULL )
      {
           return((CS_INT)CS_FAIL);
      }

      /*
      ** Now register the procedure al&ready defined by
      ** srv_regdefine and(or) srv_regparam.If an error
      ** occurred, return the cause of error.
      */
      if ( srv_regcreate(sproc, &info) == CS_FAIL )
      {
            return(info);
      }

      /* The procedure is registered. */
      return((CS_INT)CS_SUCCEED);
}
```

使用法

- プロシージャを登録するために必要な情報がすべて提供された後に、srv_regcreate は登録を完了します。
- プロシージャの名前とパラメータは、あらかじめそれぞれ srv_regdefine と srv_regparam を使用して定義されていなければなりません。
- 一度登録されると、クライアント・アプリケーションからでも Open Server アプリケーション・プログラムの中からでも、プロシージャを呼び出すことができます。
- プロシージャを登録する例は、srv_regdefine を参照してください。

参照

srv_regdefine、srv_regdrop、srv_reglist、srv_regparam

srv_regdefine

説明

プロシージャを登録するプロセスを開始します。

構文

```
CS_RETCODE srv_regdefine(spp, procnamep,
                namelen, funcp)
SRV_PROC          *spp;
CS_CHAR           *procnamep;
CS_INT            namelen;
SRV_EVENTHANDLE_FUNC(*funcp)();
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

procnamep

プロシージャ名へのポインタです。

namelen

プロシージャ名の長さです。**proc_namep* の文字列が null で終了する場合、*namelen* は CS_NULLTERM とすることもできます。

funcp

プロシージャが実行されるたびに呼び出される関数へのポインタです。このパラメータを null に設定すると、「ノーティフィケーション (通知) プロシージャ」が登録されます。ノーティフィケーション・プロシージャは、クライアント間での通信には便利です。「レジスタード・プロシージャ」(151 ページ) を参照してください。

戻り値

表 3-92: 戻り値 *(srv_regdefine)*

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

```
#include    <ospublic.h>
#include    <stdio.h>

/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_regdefine PROTOTYPE((
SRV_SERVER      *server
));
CS_RETCODE      stop_serv PROTOTYPE((
SRV_PROC        *spp
));
```

```
/*
** Local defines.
*/
#define   STOP_SERV   "stop_serv"

/*
** STOP_SERV
**    This function is called when the client sends the stop_serv
**    registered procedure.
**
** Arguments:
**    spp A pointer to internal thread control structure.
**
** Returns:
**    SRV_CONTINUE
*/
CS_INT     stop_serv(spp)
SRV_PROC   *spp;
{
 /* Queue a SRV_STOP event. */
 (CS_VOID)srv_log((SRV_SERVER *)NULL, CS_TRUE,
           "Stopping Server¥n", CS_NULLTERM);

     /* Send a final DONE to client to acknowledge the command. */
     if (srv_senddone(spp, SRV_DONE_FINAL, CS_TRAN_UNDEFINED,
                (CS_INT)0)
                == CS_FAIL)
     {
            fprintf(stderr, "srv_senddone failed¥n");
     }

     /* Queue a SRV_STOP event to shut down the server. */
 if (srv_event(spp, SRV_STOP, (CS_VOID *)NULL)
           == CS_FAIL)
     {
           fprintf(stderr, "Error queuing SRV_STOP event¥n");
     }
 return(SRV_CONTINUE);
}

/*
** EX_SRV_REGDEFINE
**
**    Example routine to illustrate the use of srv_regdefine to
**    register a procedure.
**
** Arguments:
**   server       A pointer to the Open Server control structure.
**
** Returns:
**
```

```
**   CS_SUCCEED  If procedure was registered successfully.
**   CS_FAIL     If an error occurred in registering the
**                  procedure.
*/
CS_RETCODE      ex_srv_regdefine (server)
SRV_SERVER      *server;
{
      SRV_PROC  *spp;
      CS_INT    info;

      /* Create a thread. */
      spp = srv_createproc(server);

      if (spp == (SRV_PROC *)NULL)
           return (CS_FAIL);

      /* Define the procedure. */
      if (srv_regdefine(spp, STOP_SERV, CS_NULLTERM, stop_serv)
           == CS_FAIL)
           return (CS_FAIL);

      /* Complete the registration. */
      if (srv_regcreate(spp, &info) == CS_FAIL)
           return (CS_FAIL);

      /*
      ** Terminate the thread created here.We do not care about
      ** the return code from srv_termproc here.
      */
      (CS_VOID)srv_termproc(spp);

      return (CS_SUCCEED);
}
```

使用法

- srv_regdefine は、プロシージャを登録するプロセスの最初の手順です。プロシージャは一度登録されると、クライアントから、または Open Server アプリケーション・プログラムの中からでも呼び出せます。
- srv_regdefine を呼び出した後、srv_regparam を使用してプロシージャのパラメータを定義します。
- プロシージャの登録プロセスは、srv_regcreate を呼び出して終了します。
- *procnamep* と同一の名前を持つレジスタード・プロシージャがある場合はエラーが検出され、srv_regcreate が呼び出されたときにレポートされます。
- すべての要求されたプロシージャは、SRV_CONTINUE を返します。

参照

srv_regcreate、srv_regdrop、srv_reglist、srv_regparam

srv_regdrop

説明

プロシージャの登録を解除します。

構文

```
CS_RETCODE srv_regdrop(spp, procnamep,
                namelen, info)
SRV_PROC        *spp;
CS_CHAR         *procnamep;
CS_INT           namelen;
CS_INT          *infop;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

procnamep

プロシージャ名へのポインタです。

namelen

レジスタード・プロシージャ名の長さです。名前が null で終了する場合、*namelen* は CS_NULLTERM とすることもできます。

infop

CS_INT へのポインタです。srv_regdrop が CS_FAIL を返すと、次の値のいずれかにフラグが設定されます。

- SRV_I_PNOTKNOWN − プロシージャは登録されていない。
- SRV_I_UNKNOWN − その他のエラーが発生した。

戻り値

表 *3-93:* 戻り値 *(srv_regdrop)*

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

```
#include    <ospublic.h>
/*
 ** Local Prototype.
 */
CS_RETCODE      ex_srv_regdrop PROTOTYPE((
SRV_PROC        *spp,
CS_CHAR         *name,
CS_INT          namelen,
CS_INT          *infop
));
```

```
/*
** EX_SRV_REGDROP
**
**    Example routine to unregister a registered procedure using
**    srv_regdrop.
**
** Arguments:
**   spp       A pointer to an internal thread control structure.
**   name      The name of the registered procedure to drop.
**   namelen   The length of the registered procedure name.
**   infop     A return pointer to an integer containing more
**             descriptive error information if this routine
**             returns CS_FAIL.
**
** Returns:
**   CS_SUCCEED    Registered procedure was successfully deleted.
**   CS_FAIL       Registered procedure was not deleted or does
**                  not exist.
*/
CS_RETCODE     ex_srv_regdrop(spp, name, namelen, infop)
SRV_PROC       *spp;
CS_CHAR        *name;
CS_INT         namelen;
CS_INT         *infop;
{
      /* Initialization. */
      *infop = (CS_INT)0;
      /* Execute the procedure. */
      if (srv_regdrop(spp, name, namelen, infop) != CS_SUCCEED)
      {
           /* Open Server has set infop to a specific error. */
           return(CS_FAIL);
      }
      return(CS_SUCCEED);
}
```

使用法

- srv_regdrop は、以前に srv_regcreate で登録したプロシージャを削除します。
- このプロシージャの通知を待っているクライアント・スレッドには、プロシージャが削除されたことが通知されます。

参照

srv_regcreate、srv_regdefine、srv_reglist、srv_regparam

srv_regexec

説明

レジスタード・プロシージャを実行します。

構文

```
CS_RETCODE srv_regexec(spp, infop)
SRV_PROC        *spp;
CS_INT          *infop;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

infop

CS_INT を指すポインタです。表 3-94 に、srv_regexec が CS_FAIL を返す場合に **infop* に返される可能性がある値を示します。

表 3-94: infop の値 (srv_regexec)

値	説明
SRV_I_PNOTKNOWN	プロシージャは登録されていない。
SRV_I_PPARAMERR	パラメータ・エラーがある。
SRV_I_PNOTIFYERR	通知を送信中にエラーが発生した。

戻り値

表 3-95: 戻り値 (srv_regexec)

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

```
#include <ospublic.h>

/*
** Local Prototype
*/
CS_RETCODE      ex_srv_regexec PROTOTYPE((
SRV_PROC        *spp,
CS_INT          &infop
));

/*
** EX_SRV_REGEXEC
**
**   Example routine to complete the execution of a registered
**   procedure using srv_regexec.This routine should be called
**   after srv_reginit and srv_regparam.
**
** Arguments:
**   spp       A pointer to an internal thread control structure.
**   infop     A return pointer to an integer containing more
**             descriptive error information if this routine
**             returns CS_FAIL.
```

```
**
** Returns:
**    CS_SUCCEED    Registered procedure executed successfully.
**    CS_FAIL       Registered procedure not executed, or
**                  notifications not completed successfully.
*/
CS_RETCODE ex_srv_regexec(spp, infop)
SRV_PROC   *spp;
CS_INT     &infop;
{
      /* Initialization. */
      &infop = (CS_INT)0;

      /* Execute the procedure. */
      if (srv_regexec(spp, infop) != CS_SUCCEED)
      {
           /*
           ** Open Server has set the argument to a specific
           ** error.
           */
           return(CS_FAIL);
      }
      return(CS_SUCCEED);
}
```

使用法

- srv_regexec は、レジスタード・プロシージャを実行します。
- srv_regexec を呼び出す前に、srv_reginit および srv_regparam でプロシージャ名とパラメータを指定します。

警告！ Open Server のシステム・レジスタード・プロシージャは、最終の DONE を送信します。アプリケーションがイベント・ハンドラから srv_regexec を使用してシステム・レジスタード・プロシージャを実行した場合には、アプリケーションはイベント・ハンドラ・コードから最終の DONE を送信しません。この場合、Open Server はステータス・エラーとなります。

参照

srv_reginit、srv_regparam

srv_reginit

説明

レジスタード・プロシージャの実行を開始します。

構文

```
CS_RETCODE srv_reginit(spp, procnamep,
                namelen, options)

SRV_PROC        *spp;
CS_CHAR         *procnamep;
CS_INT          namelen;
unsigned short  options;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

procnamep

レジスタード・プロシージャ名へのポインタです。

namelen

プロシージャ名の長さです。名前が null で終了する場合、*namelen* は CS_NULLTERM とすることもできます。

options

どのスレッドに通知するかを決めるフラグです。表 3-96 に、*options* の有効値を示します。

表 *3-96: options* の値 *(srv_reginit)*

値	説明
SRV_M_PNOTIFYALL	通知リストにある待機中のすべてのスレッドに通知する。
SRV_M_PNOTIFYNEXT	待機時間が最長のスレッドにのみ通知する。

戻り値

表 *3-97:* 戻り値 *(srv_reginit)*

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

```
#include    <ospublic.h>
/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_reginit  PROTOTYPE((
SRV_PROC        *sp,
CS_CHAR         *pname,
CS_INT          nlen
));

/*
** EX_SRV_REGINIT
**
**    This routine demonstrates how to use srv_reginit to
**    initiate the execution of a registered procedure.
```

```
**
** Arguments:
**         sp       A pointer to an internal thread control
                    structure.
**         pname    The name of the procedure to execute.
**         nlen     The length of the procedure name.
** Returns
**
**    CS_SUCCEED    If the registered procedure began execution.
**    CS_FAIL       If an error was detected.
**
*/
CS_RETCODE      ex_srv_reginit(sp, pname, nlen)
SRV_PROC        *sp;
CS_CHAR         *pname;
CS_INT          nlen;
{
      /*
      ** Call srv_reginit to initiate the execution of this
      ** registered procedure; ask that all threads waiting for
      ** notification of this event be notified.
      */
      if( srv_reginit(sp, pname, nlen, SRV_M_PNOTIFYALL) ==
            CS_FAIL )
      {
           /*
           ** An error was al&ready raised.
           */
           return CS_FAIL;
      }

      /*
      ** All done.
      */
      return CS_SUCCEED;
}
```

使用法

- srv_reginit はレジスタード・プロシージャの実行プロセスの最初の手順です。
- srv_reginit を呼び出した後に、srv_regparam を使ってプロシージャのパラメータを定義します。
- レジスタード・プロシージャの実行は、srv_regexec を呼び出して行います。
- プロシージャが存在しない場合には、srv_regexec は、エラーの検出とレポートを行います。

- レジスタード・プロシージャが実行されると、Open Server は、プロシージャの通知リスト内のスレッドに通知します。*options* パラメータを指定して、リスト中のすべてのスレッドに通知を送信するか、待機時間が最長のスレッドにだけ通知を送信するかを決定します。
- Open Server アプリケーションでは、レジスタード・プロシージャを最大 16 レベルまでネストすることができます。

参照　srv_regexec、srv_regparam

srv_reglist

説明　Open Server に登録されているすべてのプロシージャのリストを取得します。

構文

```
CS_RETCODE srv_reglist(spp, proclistp)
SRV_PROC            *spp;
SRV_PROCLIST        **proclistp;
```

パラメータ

spp
内部スレッド制御構造体へのポインタです。

proclistp
結果を含む SRV_PROCLIST のアドレスに設定される SRV_PROCLIST ポインタへのポインタです。srv_reglist が呼び出されるときに、Open Server はこの構造体用の領域を割り付けます。

戻り値

表 *3-98:* 戻り値 *(srv_reglist)*

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

```
#include   <ospublic.h>
/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_reglist PROTOTYPE((
SRV_PROC        *spp,
SRV_PROCLIST    **proclp
));

/*
** EX_SRV_REGLIST
**
** Arguments:
**
**   spp       Pointer to an internal thread control structure.
**   proclp    Pointer to a SRV_PROCLIST pointer that will be set
```

```
**             to point to the result.
**
** Returns
**
**   CS_SUCCEED    srv_reglist was successful.
**   CS_FAIL       An argument was invalid or srv_reglist failed.
**
*/
CS_RETCODE      ex_srv_reglist (spp, proclp)
SRV_PROC        *spp;
SRV_PROCLIST    **proclp;
{
      /* Check arguments. */
      if (spp == (SRV_PROC *)NULL)
      {
            return(CS_FAIL);
      }
      return(srv_reglist(spp,proclp));
}
```

使用法

- srv_reglist は、スレッドに対する、現在のレジスタード・プロシージャのリストをすべて返します。
- *proclistp* パラメータは、Open Server によって割り付けられ、初期化される構造体を指すように設定されます。SRV_PROCLIST 構造体は次のように定義されます。

```
typedef struct srv__proclist
 {
      CS_INT        num_procs;      /* The number of procedures */
      CS_CHAR       **proc_list;    /* Array of procedure names */
 } SRV_PROCLIST;
```

- SRV_PROCLIST 構造体は、不要になった場合、srv_reglistfree を使って割り付けを解除します。

参照 srv_reglistfree

srv_reglistfree

説明 以前に割り付けられた SRV_PROCLIST 構造体を解放します。

構文

```
CS_RETCODE srv_reglistfree(spp, proclistp)
SRV_PROC          *spp;
SRV_PROCLIST      *proclistp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

proc_list
以前に srv_reglist または srv_regwatchlist を使用して割り付けられた SRV_PROCLIST 構造体へのポインタです。

戻り値

表 3-99: 戻り値 (srv_reglistfree)

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

```
#include    <ospublic.h>

/*
** Local Prototype
*/
CS_RETCODE      ex_srv_reglistfree PROTOTYPE((
SRV_PROC        *srvproc,
SRV_PROCLIST    *reglistp
));

/*
** EX_SRV_REGLISTFREE
**
**   Example routine to free a previously allocated reglist.
**
** Arguments:
**  srvproc    A pointer to an internal thread control structure.
**  reglistp   A pointer to the list to free.
**
** Returns:
**
**    CS_SUCCEED
**    CS_FAIL
*/
CS_RETCODE      ex_srv_reglistfree(srvproc, reglistp)
 SRV_PROC        *srvproc;
SRV_PROCLIST    *reglistp;
{
       return(srv_reglistfree(srvproc, reglistp));
}
```

使用法

srv_reglistfree は、srv_reglist または srv_regwatchlist を使用して割り付けられた SRV_PROCLIST 構造体の割り付けを解除します。

参照

srv_reglist、srv_regwatchlist

srv_regnowatch

説明

レジスタード・プロシージャの通知リストから、クライアント・スレッドを削除します。

構文

```
CS_RETCODE srv_regnowatch(spp, procnamep,
                namelen, infop)

SRV_PROC        *spp;
CS_CHAR         *procnamep;
CS_INT           namelen;
CS_INT          *infop;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

procnamep

プロシージャ名へのポインタです。

namelen

プロシージャ名の長さです。名前が null で終了する場合、*namelen* は CS_NULLTERM とすることもできます。

infop

CS_INT を指すポインタです。表 3-100 に、srv_regnowatch が CS_FAIL を返す場合に **infop* に返される可能性がある値を示します。

表 *3-100: infop* の値 *(srv_regnowatch)*

値	説明
SRV_I_PNOTCLIENT	非クライアント・スレッドが指定された。
SRV_I_PNOTKNOWN	プロシージャは、Open Server アプリケーションには認識されていない。
SRV_I_PNOPENDING	そのスレッドは、このプロシージャの通知リストには含まれていない。
SRV_I_PPARAMERR	パラメータ・エラーが生じた。
SRV_I_UNKNOWN	その他のエラーが発生した。

戻り値

表 *3-101:* 戻り値 *(srv_regnowatch)*

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

```
#include   <ospublic.h>

/*
** Local Prototype.
*/
extern CS_RETCODE  ex_srv_regnowatch PROTOTYPE((
CS_VOID          *spp,
CS_CHAR          *procnamep,
CS_INT           namelen
));

/*
** EX_SRV_REGNOWATCH
**
**    Remove a client thread from the notification list for the
**    specified registered procedure.
**
** Arguments:
**   spp            A pointer to an internal thread control
                    structure.
**   procnamep      A pointer to the name of the registered
                    procedure.
**   namelen       The length of the registered procedure name.
**
** Returns:
**   CS_SUCCEED     The thread was removed from notification list.
**   CS_FAIL        An error was detected.
*/
CS_RETCODE       ex_srv_regnowatch(spp, procnamep, namelen)
SRV_PROC         *spp;
CS_CHAR          *procnamep;
CS_INT           namelen;
{
      if(srv_regnowatch(spp, procnamep, namelen, (CS_INT *)NULL)
           == CS_FAIL)
      {
           return(CS_FAIL);
      }
      return(CS_SUCCEED);
}
```

使用法

- srv_regnowatch は、指定したプロシージャを実行したときに、通知するスレッドのリストからクライアント・スレッドを削除します。
- プロシージャ名の最大長は SRV_MAXNAME です。

参照

srv_regwatch、srv_regwatchlist

srv_regparam

説明

定義されているレジスタード・プロシージャに対してパラメータを記述する、またはレジスタード・プロシージャの実行に対してデータを提供します。

構文

```
CS_RETCODE srv_regparam(spp, param_namep, namelen,
                type, datalen, datap)

SRV_PROC        *spp;
CS_CHAR         *param_namep;
CS_INT           namelen;
CS_INT           type;
CS_INT           datalen;
CS_BYTE         *datap;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

param_namep

パラメータ名へのポインタです。プロシージャを登録するときには、このパラメータは必須条件です。プロシージャを呼び出すときに、プロシージャが登録されたときのパラメータの定義の順序と同じ順序でパラメータを指定する場合は、このパラメータを null にすることもできます。

namelen

パラメータ名の長さです。*param_namep* が null で終了する場合、*namelen* は CS_NULLTERM とすることもできます。

type

パラメータのデータ型です。Open Server データ型のリストについては、「データ型」(187 ページ) を参照してください。

datalen

パラメータのデータ長です。固定長データ型の場合、このパラメータは無視されます。null データ値を示すには、*datalen* を 0 に設定します。クライアントがパラメータ値を指定しない場合には、Open Server アプリケーションはデフォルト値の長さを設定します。デフォルト値を定義したくない場合には、*datalen* を SRV_NODEFAULT に設定してください。

datap

データへのポインタです。プロシージャを登録する場合は、**datap* の値はプロシージャの今後の呼び出しのデフォルト値を示します。プロシージャを呼び出している場合、デフォルト値を受け取るには *datap* を NULL に設定します。

戻り値

表 3-102: 戻り値 *(srv_regparam)*

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

```
#include <ospublic.h>

/*
** Local prototype.
*/
CS_RETCODE  ex_srv_regparam PROTOTYPE((
SRV_PROC    *spp
));

/*
** Local defines.
*/
#define PARAMNAME (CS_CHAR *)”myparam” /* Parameter name. */

#define PARAMDEFAULT (CS_INT)100

/*
**The default value for the parameter.
*/

#define PARAMVAL (CS_INT)20 /* The value for this invocation. */

/*
** EX_SRV_REGPARAM
**
**    Example routine to describe a parameter for a registered
**    procedure.
**
** Arguments:
**   spp   A pointer to an internal thread control structure.
**
** Returns:
**    CS_SUCCEED If we were able to describe the parameter.
**    CS_FAIL If an error was detected.
*/
CS_RETCODE      ex_srv_regparam(spp)
SRV_PROC        *spp;
{
      CS_RETCODE    result;
      CS_INT        param;

      /* Define the parameter with a default. */
      param = PARAMDEFAULT;
      result = srv_regparam(spp, PARAMNAME, CS_NULLTERM,
             CS_INT_TYPE, sizeof(CS_INT), (CS_BYTE *)&param);
```

```
        if (result == CS_FAIL)
        {
              return (CS_FAIL);
        }

        /* Define the parameter with no default. */
        result = srv_regparam(spp, PARAMNAME, CS_NULLTERM,
                 CS_INT_TYPE, SRV_NODEFAULT, (CS_BYTE *)NULL);

        if (result == CS_FAIL)
        {
             return (CS_FAIL);
        }

        /* Give a non-default value for the parameter. */
        param = PARAMVAL;
        result = srv_regparam(spp, PARAMNAME, CS_NULLTERM,
                  CS_INT_TYPE, sizeof(CS_INT), (CS_BYTE *)&param);

        return (result);
    }
```

使用法

- srv_regparam は、プロシージャの呼び出しまたは登録のためのプロシージャ・パラメータを指定します。srv_regparam は、srv_reginit または srv_regdefine を呼び出してから実行します。
- レジスタード・プロシージャは、最大 1024 のパラメータを持つことができます。
- プロシージャを登録するときには、srv_regparam を使って、プロシージャのパラメータとデフォルト値のプロパティを定義します。
- プロシージャを呼び出すときには、デフォルト値が指定されているパラメータを除き、各パラメータについて srv_regparam を呼び出します。
- null データ値を示すには、*datalen* を 0 に設定します。
- プロシージャを実行するときにパラメータのデフォルト値を受け取るには、*datap* を NULL に設定します。
- パラメータに提供されたデフォルト値をプロシージャの実行に使用する場合は、srv_regparam を呼び出す必要はありません。

参照

srv_regcreate、srv_regdefine、srv_reginit、srv_regexec、「データ型」(187 ページ)

srv_regwatch

説明

指定されたプロシージャの通知リストに、クライアント・スレッドを追加します。

構文

```
CS_RETCODE srv_regwatch(spp, proc_namep, namelen,
                options, infop)

SRV_PROC        *spp;
CS_CHAR         *proc_namep;
CS_INT           namelen;
CS_INT           options;
CS_INT          *infop;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

proc_namep

プロシージャの名前です。

namelen

プロシージャ名の長さです。プロシージャ名が null で終了する場合、*namelen* は CS_NULLTERM とすることもできます。

options

一度かぎりの通知要求なのか、永続的な通知要求なのかを指定するフラグです。表 3-103 に、*options* の有効値を示します。

表 *3-103: options* の値 *(srv_regwatch)*

値	説明
SRV_NOTIFY_ONCE	一度目の通知の後に、クライアント・スレッドはプロシージャの通知リストから削除される。
SRV_NOTIFY_ALWAYS	srv_regnowatch を使ってプロシージャの通知リストからスレッドが削除されるまで、プロシージャが実行されるたびにクライアント・スレッドは通知を受ける。

infop

表 3-104 に、srv_regwatch が CS_FAIL を返す場合に **infop* に返される可能性がある値を示します。

表 *3-104: infop* の値 *(srv_regwatch)*

値	説明
SRV_I_PNOTKNOWN	プロシージャは、Open Server アプリケーションには認識されていない。スレッドは通知リストに追加されなかった。
SRV_I_PINVOPT	有効でない *options* 値が指定された。スレッドは通知リストに追加されなかった。
SRV_I_PNOTCLIENT	非クライアント・スレッドが指定された。スレッドは通知リストに追加されなかった。
SRV_I_PNOTIFYEXISTS	スレッドは、指定されたプロシージャの通知リストにすでに存在する。

戻り値

表 *3-105: 戻り値 (srv_regwatch)*

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

```
#include     <ospublic.h>

/*
** Local Prototype.
*/
CS_INT      ex_srv_regwatch PROTOTYPE((
SRV_PROC    *sproc,
CS_CHAR     *procedure_name
));

/*
** EX_SRV_REGWATCH
**   An example routine to add a client thread to the
**   notification list for a specified procedure.
**
** Arguments:

**   sproc              A pointer to an internal thread control
**                      structure.
**   procedure_name     The null terminated procedure name.
**
** Returns:
**   CS_SUCCEED            If the thread was added to the
**                         notification list.
**   SRV_I_PNOTKNOWN       The procedure is not known to the Open
**                         Server application.
**   SRV_I_PNOTCLIENT      A non-client thread was specified.
**   SRV_I_PNOTIFYEXISTS   The thread is al&ready on the
**                         notification list for the specified
**                         procedure.
**   CS_FAIL               The attempt to add the thread to the
**                         notification failed due to other
**                         errors.
*/
CS_INT      ex_srv_regwatch(sproc, procedure_name)
SRV_PROC    *sproc;
CS_CHAR     *procedure_name;
{
      CS_INT    info;

      if ( srv_regwatch(sproc, procedure_name, CS_NULLTERM,
            SRV_NOTIFY_ALWAYS, &info) == CS_FAIL )
      {
```

```
                if ( (info == SRV_I_PNOTKNOWN)
                || (info == SRV_I_PNOTCLIENT)
                || (info == SRV_I_PNOTIFYEXISTS) )
                {
                    return(info);
                }
                else
                {
                    return((CS_INT)CS_FAIL);
                }
            }

            return((CS_INT)CS_SUCCEED);
    }
```

使用法

- srv_regwatch は、指定されたプロシージャが実行したときに通知するスレッドのリストに、スレッドを追加します。
- *options* フラグでは、プロシージャ実行時に必ずスレッドに通知するか、次回のプロシージャ実行時に 1 回だけ通知するかを指定します。
- 通知要求を取り消すには、srv_regnowatch を使用します。

参照

srv_regnowatch、srv_regwatchlist

srv_regwatchlist

説明

クライアント・スレッドが通知要求待ちになっている、すべてのレジスタード・プロシージャのリストを返します。

構文

```
CS_RETCODE srv_regwatchlist(spp, proclistp)
SRV_PROC        *spp;
SRV_PROCLIST    **proclistp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

proclistp

レジスタード・プロシージャの数と各レジスタード・プロシージャの名前を持つ構造体へのポインタを指すポインタです。Open Server は、この構造体用に領域を割り付けます。

戻り値

表 *3-106:* 戻り値 *(srv_regwatchlist)*

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

```
#include   <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE     ex_srv_regwatchlist PROTOTYPE((
SRV_PROC       *spp
));

/*
** EX_SRV_REGWATCHLIST
**
**    Example routine to get a list of all registered procedures
**    for which a client thread has notifications pending.
**
** Arguments:
**    spp      A pointer to an internal thread control structure.
**
** Returns:
**
**    CS_SUCCEED    The list returned successfully.
**    CS_FAIL       An error was detected.
*/
CS_RETCODE      ex_srv_regwatchlist(spp)
SRV_PROC        *spp;
{
     SRV_PROCLIST  *listp;

     if (srv_regwatchlist(spp, &listp) == CS_FAIL)
           return (CS_FAIL);

      /*
      **   Process the information in the list and free the
      **   memory allocated for the list.
      */
      (CS_VOID)srv_reglistfree(spp, listp);

      return (CS_SUCCEED);
}
```

使用法

- srv_regwatchlist は、クライアント・スレッドが通知を要求したレジスタード・プロシージャのリストを返します。
- *proclistp* パラメータは、Open Server によって割り付けられ、初期化される SRV_PROCLIST 構造体を指します。SRV_PROCLIST 構造体を次に示します。

```
typedef struct srv__proclist
{
    CS_INT       num_procs;     /* The number of procedure names */
     CS_CHAR      **proc_list;   /* The list of procedure names */
} SRV_PROCLIST;
```

- アプリケーションは、srv_reglistfree を呼び出すことによって、SRV_PROCLIST 構造体の割り付けを解除します。

参照

srv_reglistfree

srv_rpcdb

説明

現在のリモート・プロシージャの対象のデータベース要素を返します。

構文

```
CS_CHAR *srv_rpcdb(spp, lenp)
SRV_PROC       *spp;
CS_INT         *lenp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

lenp

データベース名の長さが格納されている *int* 変数へのポインタです。*lenp* には NULL も指定できますが、その場合にはデータベース名の長さは返されません。

戻り値

表 *3-107: 戻り値 (srv_rpcdb)*

戻り値	意味
現在の RPC の名前のデータベース要素を含む、null で終了する文字列を指すポインタ	現在の RPC の名前のデータベース要素のロケーション。
(CS_CHAR *) NULL	現在の RPC が存在しない。 Open Server は *lenp* を -1 に設定し、情報エラーが発生する。

例

```
#include        <ospublic.h>
/*
 ** Local Prototype.
 */
CS_RETCODE      ex_srv_rpcdb PROTOTYPE((
SRV_PROC        *spp,
CS_CHAR         **dbp,
CS_INT          *lenp
));
/*
 ** EX_SRV_RPCDB
 **
**   Example routine to return the database component name of the
 **   current remote procedure call designation, using srv_rpcdb.
**
** Arguments:
**   spp   A pointer to an internal thread control structure.
 **   dbp   A return pointer to the null terminated database name.
 **   lenp  A return pointer to an integer containing the length
 **         of the database name.
**
** Returns:
**   CS_SUCCEED   Database component name returned successfully.
**   CS_FAIL      An error was detected.
 */
CS_RETCODE     ex_srv_rpcdb(spp, dbp, lenp)
SRV_PROC       *spp;
CS_CHAR        **dbp;
CS_INT         *lenp;
{
      /* Initialization.*/
      *lenp = (CS_INT)0;
      /* Retrieve the database component name. */
      if ((*dbp = (CS_CHAR *)srv_rpcdb(spp, lenp)) == (CS_CHAR
            *)NULL)
      {
           return(CS_FAIL);
      }
      return(CS_SUCCEED);
}
```

使用法

- srv_rpcdb は CS_CHAR ポインタを返します。これは、最後が null で終了する文字列へのポインタです。文字列には、現在のリモート・プロシージャ・コールの名前のデータベース名に当たる部分が含まれています。

- srv_rpcdb は、RPC の名前からデータベース名に当たる部分だけを返します。所有者や RPC 番号を示すオプションの指定子など、データベース名以外の部分は返しません。完全に修飾されたストアド・プロシージャの名前の形式は、*database.owner.rpcname;number* です。RPC の名前から他の部分 (存在する場合) を取得するには、srv_rpcname、srv_rpcowner、srv_rpcnumber を使用してください。

参照　srv_numparams、srv_rpcname、srv_rpcnumber、srv_rpcoptions、srv_rpcowner

srv_rpcname

説明　現在のリモート・プロシージャ・コールの名前から、RPC 名に当たる部分を返します。

構文

```
CS_CHAR *srv_rpcname(spp, lenp)
SRV_PROC        *spp;
CS_INT          *lenp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

lenp

RPC 名の長さが格納されているバッファへのポインタです。*lenp* には NULL も指定できますが、その場合には RPC 名の長さは返されません。

戻り値

表 *3-108:* 戻り値 *(srv_rpcname)*

戻り値	意味
現在の RPC の名前のデータベース要素を含む、null で終了する名前要素を指すポインタ	現在の RPC の名前のデータベース要素のロケーション。
null ポインタ	現在の RPC が存在しない。 Open Server は *lenp* を -1 に設定し、情報エラーが発生する。

例

```
#include        <ospublic.h>
/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_rpcname PROTOTYPE((
SRV_PROC        *sp,
CS_CHAR         *buf,
CS_INT          buflen,
CS_INT          *lenp
));
```

```
/*
** EX_SRV_RPCNAME
**
**    This routine demonstrates how to use srv_rrpcname to obtain
 **    the name of the remote procedure call received by this
 **    thread.
**
** Arguments:
**   sp       A pointer to an internal thread control
 **            structure.
**   buf      The address of the buffer in which the RPC
 **            name will be returned.
**   buflen   The size of the name buffer.
**   lenp     The address of an integer variable, which
 **            will be set to the length of the name
 **            returned.
**
** Returns
**   CS_SUCCEED       If the RPC name is returned.
**   CS_FAIL          If an error occurred.
*/
CS_RETCODE      ex_srv_rpcname(sp, buf, buflen, lenp)
SRV_PROC        *sp;
CS_CHAR         *buf;
CS_INT          buflen;
CS_INT          *lenp;
{
CS_CHAR         *np;    /* The procedure name pointer. */

     /*
     ** Initialization.
     */
     np = (CS_CHAR *)NULL;
     *lenp = (CS_INT)0;

     /*
     ** Get the procedure name.
     */
     np = srv_rpcname(sp, lenp);

     if( np == (CS_CHAR *)NULL )
     {
           /*
           ** An error was al&ready raised.
           */
           return CS_FAIL;
     }

     /*
     ** Copy the RPC name to the output buffer.
     */
```

```
        (void)strncpy(buf, np, buflen);

        /*
        ** All done.
        */
        return CS_SUCCEED;
    }
```

使用法

- srv_rpcname は CS_CHAR ポインタを返します。これは、現在のリモート・プロシージャ・コール (“RPC”) の名前の RPC 名に当たる部分を含む、null で終了する文字列へのポインタです。
- srv_rpcname は RPC 名だけを返します。データベース、所有者、RPC 番号を示すオプションの指定子など、RPC 名以外の部分は返しません。たとえば、Adaptive Server Enterprise では、RPC 用の完全に修飾されたオブジェクト名は、*database.owner.rpcname;number* です。RPC の名前から他の部分 (存在する場合) を取得するには、srv_rpcdb、srv_rpcowner、srv_rpcnumber を使用してください。
- ユーザは、srv_rpcname を呼び出すことによって、RPC が存在するかどうかを調べることができます。RPC が存在しない場合には、Open Server は SRV_ENORPC エラーを返します。ユーザは、このエラーが検出された場合に、自分のエラー・ハンドラがこのエラーを無視するようにコーディングすることができます。

参照

srv_numparams、srv_rpcdb、srv_rpcnumber、srv_rpcoptions、srv_rpcowner

srv_rpcnumber

説明

現在のリモート・プロシージャで指定されている番号要素を返します。

構文

```
CS_INT srv_rpcnumber(spp)
SRV_PROC *spp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

戻り値

表 3-109: 戻り値 (srv_rpcnumber)

戻り値	意味
0 でない整数	現在の RPC の名前の番号要素。
-1	現在の RPC が存在しない。 Open Server は *lenp* を -1 に設定し、情報エラーが発生する。
0	クライアントが RPC を呼び出したときに番号要素を使用しなかった。

例

```
#include    <ospublic.h>
/*
** Local Prototype.
*/
CS_INT      ex_srv_rpcnumber PROTOTYPE((
SRV_PROC    *spp
));

/*
** EX_SRV_RCPNUMBER
**
**    Example routine to show hiw to get the number of the
 **    current RPC designation.
**
**
** Arguments:
**
**   spp   A pointer to an internal thread control structure.
**
** Returns:
**
**    The number component of the current RPC's designation. If
 **    the client used no number component when it invoked the
 **    RPC, 0 is returned.If there is not a current RPC, -1 is
 **    returned and Open Server raises an informational error.
 */
CS_INT      ex_srv_rpcnumber(spp)
SRV_PROC    *spp;
{
      /* Check arguments. */
      if (spp == (SRV_PROC *)NULL)
      {
            return (-1);
      }
      return((CS_INT)srv_rpcnumber(spp));
}
```

使用法

- srv_rpcnumber は、現在のリモート・プロシージャ・コール (“RPC”) の名前の番号要素を返します。
- srv_rpcnumber は、RPC の名前から番号に当たる部分だけを返します。所有者や RPC 名を示すオプションの指定子など、番号以外の部分は返しません。RPC の完全に修飾された名前は、*database.owner.rpcname;number* となります。RPC の名前から他の部分 (存在する場合) を取得するには、srv_rpcname、srv_rpcowner、srv_rpcdb を使用してください。

参照

srv_numparams, srv_rpcdb, srv_rpcname, srv_rpcoptions, srv_rpcowner

srv_rpcoptions

説明

現在のリモート・プロシージャ・コールのランタイム・オプションを返します。

構文

```
CS_INT        srv_rpcoptions(spp)
SRV_PROC      *spp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

戻り値

表 *3-110: 戻り値 (srv_rpcoptions)*

戻り値	意味
現在の RPC のためのランタイム・フラグを含む 0 でない整数	現在の RPC のランタイム・フラグ。
0	現在の RPC が存在しない。 Open Server はエラーになる。

例

```
#include    <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_rpcoptions PROTOTYPE((
SRV_PROC        *spp
));

/*
** EX_SRV_RPCOPTIONS
**
**    Example routine to retrieve RPC runtime options
**
** Arguments:
**   spp   A pointer to an internal thread control structure.
**
** Returns:
**
**   CS_SUCCEED
**   CS_FAIL
*/
CS_RETCODE      ex_srv_rpcoptions(spp)
SRV_PROC        *spp;
{
   CS_INT        options;

   if ( (options = srv_rpcoptions(spp)) == 0 )
         return(CS_FAIL);

   return(CS_SUCCEED);
}
```

使用法

- srv_rpcoptions は、現在のレジスタード・プロシージャ・コールのランタイム・フラグを含む CS_INT 値を返します。
- 現在は、SRV_PARAMRETURN が唯一のフラグです。SRV_PARAMRETURN が CS_TRUE の場合には、RPC は実行される前に再コンパイルされなければなりません。これは、RPC が Adaptive Server Enterprise で実行しているストアド・プロシージャである場合にのみ重要です。

参照

srv_numparams、srv_rpcdb、srv_rpcname、srv_rpcnumber、srv_rpcowner

srv_rpcowner

説明

現在のリモート・プロシージャ・コールで指定されている所有者要素を返します。

構文

```
CS_CHAR *srv_rpcowner(spp, lenp)
SRV_PROC     *spp;
CS_INT       *lenp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

lenp

所有者名の長さが格納されているバッファへのポインタです。*lenp* には NULL も指定できますが、その場合にはデータベース所有者の長さは返されません。

戻り値

表 *3-111: 戻り値 (srv_rpcowner)*

戻り値	意味
現在の RPC の名前の null で終了する所有者要素を指すポインタ	現在の RPC の名前のデータベース要素のロケーション。
null ポインタ	現在の RPC が存在しない。 Open Server は *lenp* を -1 に設定し、情報エラーが発生する。

例

```
#include    <ospublic.h>

/*
** Local Prototype.
*/
extern CS_RETCODE    ex_srv_rpcowner PROTOTYPE((
CS_VOID              *spp,
CS_CHAR        *ownerp
));
```

```
/*
** EX_SRV_RPCOWNER
**
**   Determine the owner component of an RPC destination.
**
** Arguments:
**   spp     A pointer to an internal thread control structure.
**   ownerp  A pointer to the buffer to which Open Server
 **           returns the owner component.
**
** Returns:
**   CS_SUCCEED      Owner component returned successfully.
**   CS_FAIL         An error was detected.
*/
CS_RETCODE     ex_srv_rpcowner(spp, ownerp)
SRV_PROC       *spp;
CS_CHAR        *ownerp;
{
      CS_INT    len;

      ownerp = srv_rpcowner(spp, &len);

      if(len == (CS_INT
      {
            return(CS_FAIL);
      }
      return(CS_SUCCEED);
}
```

使用法

- srv_rpcowner は CS_CHAR ポインタを返します。これは、現在のリモート・プロシージャ・コール (“RPC”) の名前の所有者に当たる部分を含む、null で終了する文字列へのポインタです。
- srv_rpcowner は、RPC の名前から所有者に当たる部分だけを返します。データベース名や RPC 番号を示すオプションの指定子など、所有者以外の部分は返しません。RPC の完全に修飾された名前は、*database.owner.rpcname;number* となります。RPC の名前から所有者以外の部分 (存在する場合) を取得するには、srv_rpcname、srv_rpcdb、srv_rpcnumber を使用してください。

参照

srv_numparams、srv_rpcdb、srv_rpcname、srv_rpcnumber、srv_rpcoptions

srv_run

説明

Open Server アプリケーションを開始します。

構文

```
CS_RETCODE srv_run(ssp)
SRV_SERVER    *ssp;
```

パラメータ

ssp

Open Server の制御構造体へのポインタです。この引数はオプションです。

戻り値

表 3-112: 戻り値 *(srv_run)*

戻り値	意味
CS_SUCCEED	サーバが停止される。
CS_FAIL	Open Server がサーバを起動できなかった。 srv_run が CS_FAIL を返した場合、アプリケーションは、srv_init を呼び出してから、srv_run を再度呼び出します。

例

```
#include     <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE       ex_srv_run PROTOTYPE((CS_VOID));

/*
** EX_SRV_RUN
**   An example routine to start up an Open Server using srv_run.
**
** Arguments:
**   None.
**
** Returns:
**    SRV_STOP    If the server is stopped.
**    CS_FAIL     If the server can't be brought up.
*/
CS_RETCODE ex_srv_run()
{
      return(srv_run((SRV_SERVER *)NULL));
}
```

使用法

- srv_run は、Open Server アプリケーションを起動、あるいは再起動します。
- srv_run は、サーバが SRV_STOP イベントで停止したときに戻ります。
- 一度起動すると、サーバはクライアント要求を受信し、その要求を処理するために定義された関数を呼び出し、さらに要求を受信します。

- サーバが停止した場合には、srv_init を使って再初期化してから、再起動しなければなりません。

注意　srv_run が DLL のエントリ関数で呼び出された場合、デッドロックが起きることがあります。srv_run 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同期化しようとします。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。

参照　srv_init, srv_props、「イベント」(84 ページ)

srv_s_ssl_local_id

説明　ローカル ID (認証) ファイルへのパスを指定するために使用されるプロパティです。

構文

```
typedef struct _cs_sslid
{
      CS_CHAR     *identity_file;
      CS_CHAR     *identity_password;
} CS_SSLIDENTITY
```

パラメータ

identity_file

デジタル証明書およびそれに関連付けられたプライベート・キーを含むファイルへのパスを提供します。

CS_GET は、CS_CONNECTION に設定されている場合にのみ、使用されている *indentity_file* を返します。

identity_password

プライベート・キーを暗号化するのに使用されます。

srv_select (UNIX のみ)

説明　指定の I/O オペレーションに対してファイル記述子の準備ができているかどうかをチェックします。

構文

```
CS_INT srv_select(nfds, &readmaskp, writemaskp,
            exceptmaskp, waitflag)

CS_INT                  nfds;
SRV_MASK_ARRAY          *&readmaskp;
SRV_MASK_ARRAY          *writemaskp;
SRV_MASK_ARRAY          *exceptmaskp;
CS_INT                  waitflag;
```

パラメータ

nfds
チェックするファイル記述子の最大番号です。

&readmaskp
読み込みが使用可能であることをチェックするためにファイル記述子のマスクで初期化された SRV_MASK_ARRAY 構造体へのポインタです。

writemaskp
書き込みが使用可能であることをチェックするためにファイル記述子のマスクで初期化された SRV_MASK_ARRAY 構造体へのポインタです。

exceptmaskp
例外をチェックするためにファイル記述子のマスクで初期化された SRV_MASK_ARRAY 構造体へのポインタです。

waitflag
任意のファイル記述子が希望の動作に対して使用可能となるまで、スレッドを保留しておかなければならないかどうかを示す CS_INT です。*waitflag* の有効値については、次の「使用法」の項を参照してください。

戻り値

示された任意のオペレーションに対して使用可能であるファイル記述子の総数を返します。エラーが生じた場合には、-1 が返されます。

表 *3-113:* 戻り値 *(srv_select)*

戻り値	意味
整数	示されたどのオペレーションに対しても使用可能であるファイル記述子の総数。
-1	ルーチンが失敗した。

例

```
#include    <ospublic.h>

 /*
 ** Local Prototype.
 */
 CS_RETCODE     ex_srv_select PROTOTYPE((
 CS_INT         readfd
 ));
 /*
 ** EX_SRV_SELECT
 **
 **   Example routine to illustrate the use of srv_select.
 **
 ** Arguments:
 **   readfd  -  fd to be checked if it is &ready for a read **
                 operation.
 **
 ** Returns:
 **  CS_SUCCEED      If readfd is &ready for a read operation.
 **  CS_FAIL         If readfd is not &ready for a read operation.
```

```
*/
CS_RETCODE      ex_srv_select(readfd)
CS_INT          readfd;
{
      SRV_MASK_ARRAY       &readmask;
      CS_BOOL              &ready;

      /* Initialization. */
     (CS_VOID)srv_mask(CS_ZERO, &&readmask, (CS_INT)0, (CS_BOOL
            *)NULL);
      &ready = CS_FALSE;

      /* Set readfd in the mask. */
      (CS_VOID)srv_mask(CS_SET, &&readmask, readfd, (CS_BOOL
            *)NULL);

      /*
      ** Check whether the descriptor is &ready for a read
      ** operation.If it is not, return.
      */
      if (srv_select(readfd+1, &&readmask, (SRV_MASK_ARRAY *)NULL,
           (SRV_MASK_ARRAY *)NULL, SRV_M_NOWAIT) <= 0 )
          return (CS_FAIL);

      /*
      ** A file descriptor is &ready for a read operation.
      */
      (CS_VOID)srv_mask(CS_GET, &&readmask, readfd, &&ready);
      return ((&ready) ? CS_SUCCEED : CS_FAIL);

}
```

使用法

- I/O を要求せずに、ネットワーク I/O オペレーションをファイル記述子で行えるかどうかを調べたい場合には、srv_select を使用します。
- Open Server は、ファイル記述子の使用可能状況を調べるときに使うグローバル・マスクに、指定されたファイル記述子を含めます。
- SRV_MASK_ARRAY の定義は、次のとおりです。

```
#define SRV_MASK_SIZE             (CS_INT)40
#define SRV_MAXMASK_LENGTH
(CS_INT)(SRV_MASK_SIZE*CS_BITS_PER_LONG)
     typedef struct        srv_mask_array
{
  long      mask_bits[SRV_MASK_SIZE];
} SRV_MASK_ARRAY;
```

SRV_MASK_SIZE は SRV_MASK_ARRAY の要素の数を示し、SRV_MAXMASK_LENGTH は SRV_MASK_ARRAY を使って表現することが可能なファイル記述子の最大数を示します。

- 外部ファイル記述子を使用する Open Server アプリケーションは、規則に従ってそれらをクローズしなければなりません。アプリケーション・スレッドは、外部ファイル記述子をクローズする前に、未処理の srv_select 呼び出しが完了するのを待たなければなりません。待たなかった場合、Open Server は終了します。
- 表 3-114 に、*waitflag* の有効値を示します。

表 *3-114: waitflag* の値 *(srv_select)*

値	意味
SRV_M_WAIT	スレッドは休止しており、マスクに表現されているファイル記述子のどれかが指定されたオペレーションのために使用可能になった場合に、ウェイクアップする。リターン・ステータスは、対象となるオペレーションに対して使用可能なファイル記述子があるかどうかを示す。
SRV_M_NOWAIT	ルーチンは、次のネットワーク・チェックの直後に戻る。リターン・ステータスは、対象となるオペレーションに対して使用可能なファイル記述子があるかどうかを示す。

- アプリケーションは、srv_select を使用して、ファイル記述子を調べてただちに戻るか、またはファイル記述子のどれかが準備ができるまでは戻らないようにするかを指定できます。
- srv_select は、SRV_START または SRV_ATTENTION ハンドラでは使用できません。

参照 srv_mask

srv_send_ctlinfo

説明 Client-Library に制御メッセージを送信します。

構文 CS_RETCODE srv_send_ctlinfo(SRV_PROC **srvproc*, CS_INT *ctl_type*, SRV_CTL_MIGRATE *ctl_type*, CS_INT *paramcnt*, SRV_CTLITEM **param*)

パラメータ

srvproc
内部スレッド制御構造体へのポインタです。

ctl_type
送信する制御メッセージのタイプです。

paramcnt
param 配列の要素の数です。

param
ライブラリ制御メッセージのパラメータです。

使用法

- *ctl_type* には、次の値を指定します。
  - SRV_CTL_MIGRATE

    マイグレーション要求をクライアントに送信するか、前のマイグレーション要求を取り消します。SRV_CTL_MIGRATE を使用できるのは、クライアントがマイグレーションをサポートし、初めてセッションに接続するときにセッション ID を受け取った場合だけです。

    「SRV_CTL_MIGRATE」(36 ページ) を参照してください。
  - SRV_CTL_LOGINREDIRECT

    接続ハンドラ使用中のみ有効です。この値を使用すると、SRV_T_REDIRECT が true である SRV_PROC は、渡されたサーバ・アドレスを使用してログインを再開するようクライアントに指示します。
  - SRV_CTL_HAUPDATE

    srv_sendinfo が有効な場合は常に有効です。このメッセージがサーバからクライアントに送信されると、クライアントは現在の HA フェールオーバ・ターゲット情報を、*param* で表されたサーバ接続情報に置き換えます。
- *param* には次のフィールドがあります。
  - SRV_CTLTYPES *srv_ctlitemtype* − srv_ctlitemtype はパラメータの型を示します。次の型を使用できます。
    - *SRV_CT_SERVERNAME* − *srv_ctlptr* が、アドレスを検索する対象のサーバの名前が含まれている CS_CHAR 文字列を指すことを示します。
    - *SRV_CT_TRANADDR* − *srv_ctlptr* が、サーバ・アドレス情報が含まれている CS_TRANADDR 構造体を指すことを示します。
    - *SRV_CT_ADDRSTR* − *srv_ctlptr* が、srv_getserverbyname によってフォーマットされた文字列を指すことを示します。
    - *SRV_CT_OPTION* − *srv_ctlptr* が、このメッセージに使用するオプション・セットが含まれている *CS_INT* ビットマスクを指すことを示します。
  - CS_INT *srv_ctllength* − 変数サイズ・パラメータの長さです。*srv_ctlitemtype* が SRV_CT_SERVERNAME または SRV_CT_ADDRSTR である場合、*srv_ctlptr* によって指された文字列の長さまたは CS_NULLTERM になります。*srv_ctlitemtype* が SRV_CT_TRANADDR である場合、*srv_ctlptr* によって指された CS_TRANADDR 配列内の要素の数になります。
  - void **srv_ctlptr* − srv_ctlptr は実際のパラメータ・データを指します。

参照

srv_freeserveraddrs、srv_getserverbyname

srv_send_data

説明

srv_send_data によって、Open Server アプリケーションでは、複数のカラムがあるローをクライアントに転送できます。そのため、Open Server アプリケーションでは、テキスト、イメージ、および XML のデータをまとまりとして送信でき、メモリが必要以上に消費されるのを防ぐことができます。

構文

```
CS_RETCODE srv_send_data(spp, column, buf, buflen)
SRV_PROC     *spp;
CS_INT       *column;
CS_BYTE      *buf;
CS_INT       buflen;
```

パラメータ

spp
内部スレッド制御構造体へのポインタです。

column
ロー・セットのカラムの数です。

buf
クライアントに送るデータを含むバッファへのポインタです。これは、セクションのサイズを決めます。

buflen
**buf* バッファの長さです。

戻り値

表 *3-115:* 戻り値 *(srv_send_data)*

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

```
#include <ctpublic.h>
#include <ospublic.h>
/*
** Local Prototype.
*/
CS_RETCODE ex_srv_send_data PROTOTYPE((
SRV_PROC *spp,
CS_COMMAND *cmd,
CS_INT cols
));
#define MAX_BULK 51200

/*
** EX_SRV_SEND_DATA

** Example routine to demonstrate how to write columns
** of data in a row set to a client using srv_send_data.
** This routine will send all the columns of data read
** from a server back to the client.
** Arguments:
** spp        - A pointer to an internal thread control structure.
```

```
** cmd        - The command handle for the command that is returning
**               text data.
** cols       - The number of columns in a row set.
** Returns:
** CS_SUCCEED - Result set sent successfully to client.
** CS_FAIL    - An error was detected.
*/
CS_RETCODE ex_srv_send_data(spp, cmd, cols)
SRV_PROC *spp;
CS_COMMAND *cmd;
CS_INT cols;
{
    CS_INT  *len;          /* Length of column data. */
    CS_INT  *outlen;       /* Number of bytes received. */
    CS_BYTE **data;        /* Column data. */
    CS_BYTE buf[MAX_BULK]; /* Buffer for text data. */
    CS_BOOL ok;           /* Error control flag. */
    CS_INT i;
    CS_INT ret;

    /* Initialization. */
    ok = CS_TRUE;

    /*
    ** Transfer a row.
    */
    for (i = 0; i < cols; i++)
    {
        if ((fmt[i].datatype != CS_TEXT_TYPE) &&
            (fmt[i].datatype != CS_IMAGE_TYPE))
        {
             /*
             ** Transfer a non TEXT/IMAGE column.
             */

             /*
             ** Read the data of a non-text/image column
             ** from the server.
             */
             ret = ct_get_data(cmd, i+1, data[i],
                   len[i], &outlen[i]);
             if ((ret != CS_SUCCEED) && (ret != CS_END_DATA)
              && (ret != CS_END_ITEM))
             {
                  ok = CS_FALSE;
                  break;
             }

             /*
             ** Write the data of a non-text/image column
             ** to client.
```

```
            */
            if (ret = srv_send_data(srvproc, i+1, NULL, 0)
                 != CS_SUCCEED)
            {
                ok = CS_FALSE;
                break;
            }
        }
        else
        {
           /*
           ** Transfer a TEXT/IMAGE column in small trunks.
           */



           /*
           ** Read a chunk of data of a text/image column
           ** from the server.
           */
           while ((ret = ct_get_data(cmd, i+1, buf, MAX_BULK, &len[i]))
                   == CS_SUCCEED)
           {
                /*
                ** Write the chunk of data to client.
                */
                if (ret = srv_send_data(srvproc, i+1, buf, len[i])
                    != CS_SUCCEED)
                {
                     ok = CS_FALSE;
                     break;
                 }
            }
        }
    }

    switch(ret)
    {
        case CS_SUCCEED:
        /* The routine completed successfully. */
        case CS_END_ITEM:
        /* Reached the end of this item's value. */
        case CS_END_DATA:
        /* Reached the end of this row's data. */
        break;
        case CS_FAIL:
        /* The routine failed. */
        case CS_CANCELED:
        /* The get data operation was cancelled. */
        case CS_PENDING:
        /* Asynchronous network I/O is in effect. */
```

```
            case CS_BUSY:
            /* An asynchronous operation is pending. */
            default:
            ok = CS_FALSE;
        }
        return (ok ?CS_SUCCEED : CS_FAIL);
    }
```

使用法

- srv_send_data は、ロー・セットのカラムのデータをカラムごとにクライアントに送信します。
- テキスト・データ、イメージ・データ、または XML データを含んでいるカラムを送信するとき、Open Server アプリケーションが srv_text_info を呼び出してから srv_send_data を呼び出す必要があります。そうすることで、データ・ストリームが、送信されるデータの全体の長さに正しく設定されます。その後、アプリケーションがデータをまとめて送信するための srv_send_data を呼び出し、送信されるデータがなくなるまでこのルーチンの呼び出しが続行されます。
- Open Server アプリケーションでは、srv_bind と srv_xferdata を使用して、テキスト・データ、イメージ・データ、および XML データをクライアントに送信できます。ただし、これらのルーチンでは、すべてのデータ・カラムを一度に送信する必要があります。srv_send_data を使用すると、アプリケーションではテキスト・データとイメージ・データをまとめて送信できます。
- srv_send_data は一度に 1 つのカラムのデータを読み込んで送信するため、ロー・セットの最初のカラムと一緒にロー全体のデータ・フォーマットをクライアントに送信する必要があります。カラムを読み込む前のオブジェクト名などの固定 I/O フィールドを取得するには、ct_data_info() を呼び出します。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でのみ取得できることに注意してください。
- Open Server アプリケーションでは、文字セット変換を除き、テキスト・データ・ストリーム、イメージ・データ・ストリーム、および XML データ・ストリームを同一に扱います。これらの変換は、テキスト・データに対してのみ行われます。

参照

『Open Server 15.0 Server Library/C リファレンス・マニュアル』の関連する srv_bind、srv_get_text、srv_text_info、srv_xferdata、srv_get_data、srv_send_text ルーチン

srv_send_text

説明

連続したデータとして、text または image のデータ・ストリームをクライアントに送ります。

構文

```
CS_RETCODE srv_send_text(spp, bp, buflen)
SRV_PROC       *spp;
CS_BYTE        *bp;
CS_INT         buflen;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

bp

クライアントに送るデータを含むバッファへのポインタです。これは、セクションのサイズを決めます。

buflen

**bp* バッファのサイズです。

戻り値

表 *3-116: 戻り値 (srv_send_text)*

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

```
#include    <ctpublic.h>
#include    <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE ex_srv_send_text PROTOTYPE((
SRV_PROC    *spp,
CS_COMMAND  *cmd
));

/*
** EX_SRV_SEND_TEXT
**
**   Example routine to demonstrate how to write text to a client
 **   using srv_send_text.This routine will send all the text
 **   read from a server back to the client.
**
** Arguments:
**   spp   A pointer to an internal thread control structure.
**   cmd   The command handle for the command that is returning
 **         text data.
```

```
**
** Returns:
**   CS_SUCCEED   Result set sent successfully to client.
**   CS_FAIL      An error was detected.
*/
CS_RETCODE      ex_srv_send_text(spp, cmd)
SRV_PROC       *spp;
CS_COMMAND     *cmd;
{
   CS_BOOL     ok;            /* Error control flag.    */
   CS_INT      ret;           /* ct_fetch return value.  */
   CS_INT      len_read;      /* Amount of data read.    */
   CS_BYTE     data[1024];    /* Buffer for text data.    */

   /* Initialization. */
   ok = CS_TRUE;

   /* Read the text from the server. */
   while ((ret = ct_get_data(cmd, 1, data, CS_SIZEOF(data),
          &len_read))
         == CS_SUCCEED)
   {
         /* Write text to client a chunck at a time */
         if (srv_send_text(spp, data, len_read) != CS_SUCCEED)
         {
                ok = CS_FALSE;
                break;
         }
   }

   switch(ret)
   {
   case CS_SUCCEED:       /* The routine completed successfully. */
   case CS_END_ITEM:      /* Reached the end of this item's value. */
   case CS_END_DATA:   /* Reached the end of this item's value. */
        break;
   case CS_FAIL:              /* The routine failed.  */
   case CS_CANCELED:    /* The get data operation was cancelled. */
   case CS_PENDING:         /* Asynchronous network I/O is in effect. */
   case CS_BUSY:               /* An asynchronous operation is pending. */
   default:
          ok = CS_FALSE;
   }

   return (ok ? CS_SUCCEED : CS_FAIL);
}
```

使用法

- srv_send_text は、クライアントに text や image データのカラムをひとつだけ送るのに使用します。
- Open Server アプリケーションは、データの全長を送るために、データ・ストリームに対して srv_send_text を最初に呼び出す前に、srv_text_info を呼び出す必要があります。それから、アプリケーションはまとまりを送るために srv_send_text を呼び出します。srv_send_text は、まとまりの数だけ呼び出します。
- クライアントから送られるアイテムは、あらかじめ srv_descfmt を使って記述されたものでなければなりません。
- また、Open Server アプリケーションは srv_bind と srv_xferdata を使って、text データと image データをクライアントに書き込むことができます。srv_send_text を使用するとデータを連続して送ることができますが、標準の srv_bind/srv_xferdata 方式ではカラムのすべてのデータを一度に送る必要があります。
- srv_send_text で送られたカラムは、text または image 型でなければなりません。
- text データだけに対して行われる文字セット変換を除き、Open Server は text と image のデータ・ストリームを同一に扱います。

警告！ ローに 1 つのカラムしかなく、そのカラムが text または image データを含む場合にのみ、Open Server アプリケーションは srv_send_text を使用してローを送ることができます。

参照

srv_bind、srv_descfmt、srv_get_text、srv_text_info、srv_xferdata、「text と image」(184 ページ)

srv_senddone

説明

クライアントに結果完了メッセージを送るか、結果の一部をフラッシュします。

構文

```
CS_RETCODE srv_senddone(spp, status, transtate, count)
SRV_PROC        *spp;
CS_INT          status;
CS_INT          transtate;
CS_INT          count;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

status

1 つ以上のフラグの論理和から構成される 2 バイトのビットマスクです。表 3-117 に各フラグを示します。

表 3-117: status の値 (srv_senddone)

ステータス	説明
SRV_DONE_FINAL	結果の現在のセットは、結果の最終セット。
SRV_DONE_MORE	結果の現在のセットは、結果の最終セットではない。
SRV_DONE_COUNT	カウント・パラメータは有効カウントを含んでいる。
SRV_DONE_ERROR	現在のクライアント・コマンドでエラーが発生した。
SRV_DONE_FLUSH	現在の結果セットは、完全なパケットを待たずにクライアントに送られる。

transtate

トランザクションの現在のステータスです。表 3-118 に、*transtate* の有効値を示します。

表 3-118: transtate の値 (srv_senddone)

トランザクションのステータス	説明
CS_TRAN_UNDEFINED	現在トランザクション中ではない。
CS_TRAN_COMPLETED	現在のトランザクションは正常に終了した。
CS_TRAN_FAIL	現在のトランザクションは失敗した。
CS_TRAN_IN_PROGRESS	現在トランザクションにある。
CS_TRAN_STMT_FAIL	現在のトランザクション文は失敗した。

count

結果の現在のセットのカウントを含む 4 バイトのフィールドです。*status* フィールドに SRV_DONE_COUNT フラグが設定してある場合、カウントは有効です。

戻り値

表 3-119: 戻り値 (srv_senddone)

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

```
#include     <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_senddone PROTOTYPE((
SRV_PROC        *spp
));

/*
** Constants and data definitions.
*/
#define    NUMROWS         3
#define    MAXROWDATA      6

CS_STATIC CS_CHAR  *row_data[NUMROWS] = {
                    "Larry",
                    "Curly",
                    "Moe"
                    };

/*
** EX_SRV_SENDDONE
**
**    Example routine illustrating the use of srv_senddone. This
 **    routine will send a set of results to the client
 **    application, and then send the results completion message.
**
** Arguments:
**    spp     A pointer to an internal thread control structure.
**
** Returns:
**    CS_SUCCEED    Results set sent successfully to client.
**    CS_FAIL       An error was detected.
*/
CS_RETCODE      ex_srv_senddone(spp)
SRV_PROC        *spp;
{
      CS_DATAFMT      fmt;
      CS_INT          row_len;
      CS_INT          idx;

      /*
      ** Describe the format of the row data, with the single
       ** dummy column.
```

```
*/
srv_bzero((CS_VOID *)&fmt, (CS_INT)sizeof(fmt));
fmt.datatype = CS_CHAR_TYPE;
fmt.maxlength = MAXROWDATA;

if (srv_descfmt(spp, (CS_INT)CS_SET, (CS_INT)SRV_ROWDATA,
          (CS_INT)1, &fmt) != CS_SUCCEED)
{
       (CS_VOID)srv_senddone(spp,
          (CS_INT)(SRV_DONE_FINAL | SRV_DONE_ERROR),
          (CS_INT)CS_TRAN_FAIL, (CS_INT)0);
       return(CS_FAIL);
 }

for (idx = 0; idx < NUMROWS; ++idx)
{
     /*
     ** Bind the row_data array element.
     */
     row_len = (CS_INT)strlen(row_data[idx]);
     if (srv_bind(spp, (CS_INT)CS_SET, (CS_INT)SRV_ROWDATA,
         (CS_INT)1, &fmt, (CS_BYTE *)(row_data[idx]),
          &row_len, (CS_SMALLINT *)NULL) != CS_SUCCEED)

     {
          /* Communicate failure, and number of rows sent. */
          (CS_VOID)srv_senddone(spp,
               (CS_INT)(SRV_DONE_FINAL |
                        SRV_DONE_ERROR | SRV_DONE_COUNT),
               (CS_INT)CS_TRAN_FAIL, (CS_INT)idx);
          return(CS_FAIL);
     }

     /*
     ** Transfer the row data.
     */
     if (srv_xferdata(spp, (CS_INT)CS_SET, SRV_ROWDATA)
                           != CS_SUCCEED)
     {
          /* Communicate failure, and number of rows sent. */
          (CS_VOID)srv_senddone(spp,
              (CS_INT)(SRV_DONE_FINAL |
                  SRV_DONE_ERROR | SRV_DONE_COUNT),
              (CS_INT)CS_TRAN_FAIL, (CS_INT)idx);
          return(CS_FAIL);
      }
}

/* Send a status value. */
if (srv_sendstatus(spp, (CS_INT)0) != CS_SUCCEED)
{
```

```
                /* Communicate failure, and number of rows sent. */
                (CS_VOID)srv_senddone(spp,
                    (CS_INT)(SRV_DONE_FINAL |
                        SRV_DONE_ERROR | SRV_DONE_COUNT),
                    (CS_INT)CS_TRAN_FAIL, (CS_INT)NUMROWS);
                return(CS_FAIL);
        }

        /* Send the final DONE message, with the row count. */
        if (srv_senddone(spp, (CS_INT)(SRV_DONE_FINAL |
               SRV_DONE_COUNT),
                    (CS_INT)CS_TRAN_COMPLETED,
                    (CS_INT)NUMROWS) != CS_SUCCEED)
        {
                /* Communicate failure, and number of rows sent. */
                (CS_VOID)srv_senddone(spp,
                    (CS_INT)(SRV_DONE_FINAL |
                            SRV_DONE_ERROR | SRV_DONE_COUNT),
                    (CS_INT)CS_TRAN_FAIL, (CS_INT)NUMROWS);
                return(CS_FAIL);
        }
    return(CS_SUCCEED);
    }
```

使用法

- srv_senddone は、結果の現在のセットが完了したというメッセージをクライアントに送信します。クライアント要求は、サーバに複数のコマンドを実行させ、複数の結果セットを返させることもできます。各結果セットについて、srv_senddone で完了メッセージが返されなければなりません。
- 現在の結果がクライアント・コマンド・バッチの最後の結果セットでない場合には、Open Server は *status* マスクの SRV_DONE_MORE フィールドを設定しなければなりません。設定しない場合には、Open Server アプリケーションは現在のコマンド・バッチに対する結果が、これ以上ないことを示すために、SRV_DONE_FINAL を *status* フィールドに設定しなければなりません。
- *count* フィールドは、特定のコマンドにより影響を受けたローがいくつあるのかを示します。*count* に実際のカウントがある場合には、SRV_DONE_COUNT ビットが *status* フィールドに設定されていなくてはなりません。こうすることにより、クライアントは 0 の実際のカウントと未使用の *count* フィールドの区別が可能になります。
- SRV_CONNECT ハンドラがクライアント・ログインを拒否した場合には、Open Server アプリケーションは *status* パラメータを SRV_DONE_ERROR フラグに設定して、srv_senddone を呼び出す必要があります。そこで、SRV_CONNECT ハンドラは、srv_senddone を使って、クライアントに DONE パケットを送信しなければなりません。いずれにしても、SRV_CONNECT ハンドラが戻り、SRV_DONE_FINAL *status* フラグが設定される前に、srv_senddone を一度だけ呼び出さなければなりません。

- 書き込み中にネットワーク・バッファが満杯になった場合、Open Server はその内容をフラッシュします。ネットワーク・バッファがどれだけの内容を含んでいるかには関係なく、status を、SRV_DONE_FINAL または SRV_DONE_FLUSH に設定した状態で、srv_senddone を発行することによって、ネットワーク・バッファのフラッシュを引き起こします。SRV_DONE_FLUSH は SRV_DONE_MORE の指定に関わらず、設定できます。
- *status* を SRV_DONE_FLUSH に設定すると、アプリケーションは長期間にわたって蓄積したクライアント側の結果をフラッシュできます。
- アプリケーションは、SRV_CONNECTION エラー・ハンドラ内では *status* 引数を SRV_DONE_FLUSH に設定することはできません。
- Open Server は、トランザクション管理を行いません。要求されたとおりに *transtate* 引数を使用して、クライアントに対して現在のトランザクション・ステータスを通知するのは、Open Server アプリケーションが行います。

注意　*transtate* 引数は、Open Server 2.0 バージョンの srv_senddone の *info* 引数に代わるものです。現行のアプリケーションの *info* の値が 0 でない場合、現行のアプリケーションではランタイム・エラーが発生します。

参照　srv_bind、srv_descfmt、srv_sendstatus、srv_xferdata

srv_sendinfo

説明　クライアントにエラー・メッセージを送ります。

構文

```
CS_RETCODE srv_sendinfo(spp, errmsgp, transtate)
SRV_PROC        *spp;
CS_SERVERMSG    *errmsgp;
CS_INT          transtate;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

errmsgp

クライアントに送られるエラー・メッセージ情報を含む CS_SERVERMSG 構造体へのポインタです。「CS_SERVERMSG 構造体」(55 ページ) を参照してください。

transtate

トランザクションの現在のステータスです。表 3-120 に、*transtate* の有効値を示します。

表 3-120: transtate の値 (srv_sendinfo)

トランザクションのステータス	説明
CS_TRAN_UNDEFINED	現在トランザクション中ではない。
CS_TRAN_COMPLETED	現在のトランザクションは正常に終了した。
CS_TRAN_FAIL	現在のトランザクションは失敗した。
CS_TRAN_IN_PROGRESS	現在トランザクションにある。
CS_TRAN_STMT_FAIL	現在のトランザクション文は失敗した。

戻り値

表 3-121: 戻り値 (srv_sendinfo)

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

```
#include    <ospublic.h>
/*
** Local Prototype.
*/

CS_RETCODE      ex_srv_sendinfo  PROTOTYPE((
SRV_PROC        *sp,
CS_CHAR         *msg,
CS_INT          msglen,
CS_INT          msgnum
));

/*
** EX_SRV_SENDINFO
**
**    This routine demonstrates how to use srv_sendinfo to send
 **    an error message to a client.
**
** Arguments:
**         sp       A pointer to an internal thread control
 **                   structure.
**         msg      The message text to send.
**         msglen   The length of the message text to send.
**         msgnum   The message number to send.
**
** Returns
**         CS_SUCCEED     If the message is sent.
**         CS_FAIL        If an error occurred.
*/
CS_RETCODE      ex_srv_sendinfo(sp, msg, msglen, msgnum)
SRV_PROC        *sp;
CS_CHAR         *msg;
CS_INT          msglen;
CS_INT          msgnum;
```

```
{
      CS_SERVERMSG    &mrec;

      /*
      ** Initialization.
      */
      srv_bzero(&&mrec, sizeof(CS_SERVERMSG));

      /*
      ** First, determine if the message string will fit
       ** in the message structure.If not, truncate it.
      */
      if( msglen > CS_MAX_MSG )
      {
          msglen = CS_MAX_MSG;
      }

      /*
      ** Now copy the message string over.
      */
      srv_bmove(msg, &mrec.text, msglen);
      &mrec.textlen = msglen;

      /*
      ** Set the message number we want to send.
      */
      &mrec.msgnumber = msgnum;

      /* Set the message status so that &mrec.text contains
      ** the entire message
      */
      &mrec.status = CS_FIRST_CHUNK | CS_LAST_CHUNK;

      /*
      ** Now we're &ready to send the message.
      */
      if( srv_sendinfo(sp, &&mrec, CS_TRAN_UNDEFINED) == CS_FAIL )
      {
            /*
            ** An error was al&ready raised.
            */
            return CS_FAIL;
      }

      /*
      ** All done.
      */
      return CS_SUCCEED;
}
```

使用法

- srv_sendinfo は、クライアントにエラー・メッセージを送信します。送信された各メッセージで一度ずつ呼び出さなければなりません。
- アプリケーションは、結果ローの送信の前後にかかわらず、srv_sendinfo を呼び出すことができます。ただし、アプリケーションは、srv_descfmt に連続して呼び出している間や、srv_descfmt と srv_xferdata の呼び出しの間に、srv_sendinfo を呼び出すことはできません。
- Open Server アプリケーションが、エラー・メッセージに関するパラメータ・データを送りたい場合には、CS_SERVERMSG 構造体の *status* フィールドを、CS_HASEED に設定します。アプリケーションは srv_sendinfo を呼び出した直後に、他の結果を送ったり、srv_senddone を呼び出す前に、エラー・パラメータを記述、バインド、送信しなければなりません。アプリケーションは、*type* 引数を SRV_ERRORDATA に設定して、srv_descfmt、srv_bind、srv_xferdata を呼び出さなければなりません。
- アプリケーションが、CS_SERVERMSG 構造体の *status* フィールドを CS_HASEED に設定して srv_sendinfo を呼び出しても、エラー・パラメータを送らなかった場合には、アプリケーションが srv_senddone を呼び出した時点で、致命的なエラーとなります。
- アプリケーションが、CS_SERVERMSG 構造体の status フィールドを CS_HASEED に設定して、srv_sendinfo を呼び出すときには、Open Server は、CS_RES_NOEED 応答機能が設定されていないことを検証します。設定されている場合には、Open Server は、エラーを出します。それ以降、エラー・パラメータを記述するために srv_descfmt を呼び出したものも、エラーを引き起こします。
- 「クライアント・コマンド・エラー」(32 ページ) を参照してください。
- 「CS_SERVERMSG 構造体」(55 ページ) も参照してください。

参照

srv_bind、srv_descfmt、srv_senddone、srv_xferdata、「クライアント・コマンド・エラー」(32 ページ)

srv_sendpassthru

説明

クライアントにプロトコル・パケットを送ります。

構文

```
CS_RETCODE srv_sendpassthru(spp, send_bufp, infop)
SRV_PROC     *spp;
CS_BYTE      *send_bufp;
CS_INT       *infop;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

send_bufp

プロトコル・パケットが格納されているバッファへのポインタです。

infop

srv_sendpassthru が CS_FAIL を返した場合に SRV_I_UNKNOWN に設定される CS_INT へのポインタです。表 3-122 は、srv_sendpassthru が CS_SUCCEED を返す場合に **infop* に返される可能性のある値を示します。

表 3-122: CS_SUCCEED の値 (srv_sendpassthru)

値	説明
SRV_I_PASSTHRU_MORE	プロトコル・パケットの送信は成功し、メッセージ・パケットの終了ではない。
SRV_I_PASSTHRU_EOM	メッセージ・プロトコル・パケットの最後の送信は成功した。

戻り値

表 3-123: 戻り値 (srv_sendpassthru)

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

```
#include    <stdio.h>
#include    <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_sendpassthru PROTOTYPE((
SRV_PROC        *spp
));

/*
** EX_SRV_SENDPASSTHRU
**
**  Example routine to send a protocol packet to a client.
**
** Arguments:
**  spp  A pointer to an internal thread control structure.
**
** Returns:
**
**  CS_SUCCEED
**  CS_FAIL
*/
CS_RETCODE      ex_srv_sendpassthru(spp)
SRV_PROC        *spp;
{
      CS_BYTE    sendbuf[20];
      CS_INT     info;
```

```
        strcpy(sendbuf, "Here's what to send");

        if (srv_sendpassthru(spp, sendbuf, &info) == CS_FAIL)
        {
              return(CS_FAIL);
        }
        else
        {
              if (info == SRV_I_PASSTHRU_MORE)
              {
                  printf("more to come...¥n");
                  return(CS_SUCCEED);
              }
              else if (info == SRV_I_PASSTHRU_EOM)
              {
                  printf("That's all.¥n");
                  return(CS_SUCCEED);
              }
              else
              {
                  printf("Unknown flag returned.¥n");
                  return(CS_FAIL);
              }
        }
    }
```

使用法

- srv_sendpassthru はクライアント・プログラムまたは Adaptive Server Enterprise から受けたプロトコル・パケットを、内容を判断せずにそのまま転送します。
- srv_sendpassthru は、プロトコル・ヘッダ・フィールドのバイト順を設定します。
- 一度 srv_sendpassthru が呼び出されると、それを呼び出したスレッドはパススルー・モードになります。パススルー・モードは、SRV_PASSTHRU_EOM が返されると終了します。
- イベント・ハンドラがパススルー・モード状態の場合は、他の Server-Library ルーチンを呼び出すことはできません。
- パススルー・モードを使用するには、クライアントの SRV_CONNECT ハンドラは、srv_getloginfo、ct_setloginfo、ct_getloginfo、srv_setloginfo を呼び出すことによって、クライアントとリモート・サーバがプロトコルをネゴシエーションできるようにしなければなりません。これによって、異なるプラットフォーム上で動作しているクライアントおよびリモート・サーバは、必要なデータ変換を行うことができます。
- srv_sendpassthru は、SRV_CONNECT、SRV_DISCONNECT、SRV_START、SRV_STOP、SRV_URGDISCONNECT、SRV_ATTENTION を除くすべてのイベント・ハンドラで使用できます。

参照

srv_getloginfo、srv_recvpassthru、srv_setloginfo

srv_sendstatus

説明

クライアントにステータス値を送信します。

構文

```
CS_RETCODE srv_sendstatus(spp, value)
SRV_PROC       *spp;
CS_INT         value;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

value

要求のステータスです。規則により、0 は要求が正常に終了したことを示します。

戻り値

表 *3-124:* 戻り値 *(srv_sendstatus)*

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

```
#include    <ospublic.h>

/*
** Local prototype.
*/
CS_RETCODE        ex_srv_sendstatus PROTOTYPE((
SRV_PROC          *spp
));

/*
** EX_SRV_SENDSTATUS
**
**  Example routine to send a status value to a client.
**
** Arguments:
**    spp      A pointer to an internal thread control structure.
**
** Returns:
**  CS_SUCCEED if we were able to send the status.
**  CS_FAIL if an error was detected.
**
*/
CS_RETCODE    ex_srv_sendstatus(spp)
SRV_PROC      *spp;
{
      CS_RETCODE      result;
```

```
        /*
        ** Send an OK status.
        */
        result = srv_sendstatus(spp, (CS_INT)0);

        return (result);
    }
```

使用法

- srv_sendstatus は、クライアント要求に応答して、クライアントにリターン・ステータス値を送信します。要求を受信すると、ユーザ定義のイベント・ハンドラ・ルーチンが呼び出されます。要求に対する応答の一部は、ステータス値を返すためのものである場合があります。
- srv_sendstatus によって送られたステータス値は任意であり、アプリケーション指定のものです。これは srv_senddone *status* パラメータとは関連していません。
- すべてのロー (存在する場合) は、srv_xferdata でクライアントに送られた後、そして srv_senddone で完了ステータスが送られる前に、ステータス値を送ることができます。ステータス値は、srv_descfmt および srv_bind の呼び出しと、srv_xferdata の呼び出しの間に送ることはできません。
- 各結果セットに対し、ステータス値は 1 つしか送ることができません。

参照

srv_senddone

srv_setcolutype

説明

カラムと関連付けられるユーザ・データ型を指定します。

構文

```
CS_RETCODE srv_setcolutype(spp, column,utype)

SRV_PROC       *spp;
CS_INT          column;
CS_INT          utype;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

column

ユーザ・データ型の関連対象となるカラムの番号です。先頭のカラムは 1 です。

utype

カラムに関連するユーザ定義のデータ型です。

戻り値

表 *3-125: 戻り値 (srv_setcolutype)*

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

```
#include    <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE       ex_srv_setcolutype PROTOTYPE((
SRV_PROC         *spp,
CS_INT           column,
CS_INT            utype
));

/*
** EX_SRV_SETCOLUTYPE
**
**   Example routine to define the user datatype to be associated
 **   with a column using srv_setcolutype.
**
** Arguments:
**  spp        A pointer to an internal thread control structure.
**  column     The column number associated with the type.
**  utype      The type to be associated with the column.
**
** Returns:
**
**  CS_SUCCEED      The datatype was successfully associated with
 **                  the column.
**  CS_FAIL         An error was detected.
*/
CS_RETCODE     ex_srv_setcolutype(spp, column, utype)
SRV_PROC       *spp;
CS_INT         column;
CS_INT         utype;
{
      /*
      ** Associate the type with the column.
      */
      if (srv_setcolutype(spp, column, utype) != CS_SUCCEED)
      {
            return(CS_FAIL);
      }
      return(CS_SUCCEED);
}
```

使用法

srv_setcolutype によって設定されたデータ型は、クライアント・アプリケーションが DB-Library 呼び出しの dbcolutype または Client-Library 呼び出しの ct_describe によって受け取るデータ型です。

srv_setcontrol

説明

カラムのユーザ制御またはフォーマット情報を記述します。

構文

```
CS_RETCODE srv_setcontrol(spp, colnum, ctrlinfop,
        ctrllen)
SRV_PROC        *spp;
CS_INT          colnum;
CS_BYTE         *ctrlinfop;
CS_INT          ctrllen;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

colnum

制御情報が適用するカラムの番号です。ローの先頭カラムの番号は 1 です。

ctrlinfop

制御データへのポインタです。制御データの長さは、ctrllen パラメータで取得します。

ctrllen

制御データ長のバイト数です。各カラムは、最大で SRV_MAXCHAR バイトの制御情報があります。

戻り値

表 *3-126: 戻り値 (srv_setcontrol)*

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

```
#include    <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE     ex_srv_setcontrol PROTOTYPE((
SRV_PROC      *spp
));

/*
** Constants.
*
#define MAXROWDATA        20
#define COLCONTROL        "Emp name: %s"

/*
** EX_SRV_SETCONTROL
**
**    Example routine to describe format information for a column
 **    using srv_setcontrol.In this example, a simple character
 **    column contains an employee name.
```

```
**
** Arguments:
**    spp    A pointer to an internal thread control structure.
**
** Returns:
**    CS_SUCCEED      Control information successfully defined.
**    CS_FAIL    An error was detected.
*/
CS_RETCODE       ex_srv_setcontrol(spp)
SRV_PROC         *spp;
{
      CS_DATAFMT fmt;

      /* Describe the format of the row data for the column. */

      srv_bzero((CS_VOID *)&fmt, (CS_INT)sizeof(fmt));
      fmt.datatype = CS_CHAR_TYPE;
      fmt.maxlength = MAXROWDATA;

      if (srv_descfmt(spp, (CS_INT)CS_SET, (CS_INT)SRV_ROWDATA,
            (CS_INT)1, &fmt) != CS_SUCCEED)
      {
            return(CS_FAIL);
      }

      /* Define the control information for the column. */
      if (srv_setcontrol(spp, (CS_INT)1, (CS_BYTE *)COLCONTROL,
            (CS_INT)strlen(COLCONTROL)) != CS_SUCCEED)
      {
            return(CS_FAIL);
      }
      return(CS_SUCCEED);
}
```

使用法

- Open Server アプリケーションは、特定のカラムに関連するユーザ定義フォーマット情報を、srv_setcontrol を使用してクライアントに伝えます。たとえば、クライアントが、特定のカラムに合わせて、特定の文字列を送信することを希望しているような場合です。
- srv_setcontrol の呼び出しは、srv_xferdata の呼び出しの前か、srv_descfmt の呼び出しの後に行わなければなりません。他のコンテキストから呼び出すと、CS_FAIL を返します。
- 制御情報は、順序に関係なくカラムと関連付けることができます。唯一の条件は、カラムがあらかじめ srv_descfmt で定義されていなければならないということです。
- ローのすべてのカラムについて、srv_setcontrol を呼び出す必要はありません。Open Server アプリケーションが、カラムの制御情報を設定しなければ、null 制御文字列が、そのカラムについて返されます。

- クライアントが、クライアント・オプションの切り替えを行う CS_OPT_CONTROL によって、具体的に制御情報を要求していない場合、アプリケーションは、このような情報を返さないようにする必要があります。

参照 srv_bind、srv_descfmt、srv_xferdata

srv_setloginfo

説明 リモート・サーバからクライアントにプロトコルのフォーマット情報を返します。

構文

```
CS_RETCODE srv_setloginfo(spp, loginfop)
SRV_PROC        *spp;
CS_LOGINFO      *loginfop;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

loginfop

ct_getloginfo によって更新された CS_LOGINFO 構造体へのポインタです。

戻り値

表 3-127: 戻り値 (srv_setloginfo)

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

```
#include    <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_setloginfo     PROTOTYPE((
SRV_PROC        *spp,
CS_LOGINFO      *loginfop
));

/*
** EX_SRV_SETLOGINFO
**
**    Return protocol format information from a remote server to
 **    a client.
**
**
** Arguments:
**
**    spp      A pointer to an internal thread control structure.
```

```
**    loginfop A pointer to a CS_LOGINFO structure that has been
 **             updated by ct_getloginfo.
**
** Returns
**
**    CS_SUCCEED
**    CS_FAIL
**
*/
CS_RETCODE      ex_srv_setloginfo(spp, loginfop)
SRV_PROC        *spp;
CS_LOGINFO      *loginfop;
{
      /* Check arguments. */
      if (spp == (SRV_PROC *)NULL)
      {
            return(CS_FAIL);
      }
      return(srv_setloginfo(spp,loginfop));
}
```

使用法

- クライアントおよびリモート Sybase サーバ間で、パケットの内容を判断せずに、プロトコル (TDS) パケットを渡すゲートウェイ・サーバ・アプリケーションでは、srv_setloginfo を使用します。
- クライアントが直接サーバに接続する場合は、2 つのプログラムは、データの送受信に使用するプロトコル・フォーマットをネゴシエートします。ゲートウェイ・アプリケーションにおいて、プロトコル・パススルーを使用すると、Open Server は、クライアントとリモート・サーバ間において、プロトコル・パケットを転送します。
- srv_setloginfo は、クライアントとリモート・サーバ間の TDS フォーマットのネゴシエーションを可能にする 4 つの呼び出しの 4 番目です (そのうち 2 つは CT-Library 呼び出しです)。これらの呼び出しは、SRV_CONNECT イベント・ハンドラにおいてのみ行えます。次にその呼び出しを示します。
  a　srv_getloginfo − CS_LOGINFO 構造体を割り付け、クライアント・スレッドからの TDS 情報を格納します。
  b　ct_setloginfo − 手順 1 で取得したプロトコル情報を使用して CS_LOGINFO 構造体を準備し、ct_connect を使用してリモート・サーバにログインします。
  c　ct_getloginfo − CS_CONNECTION 構造体から、新しく割り付けられた CS_LOGINFO 構造体にプロトコル・ログイン応答情報を転送します。
  d　srv_setloginfo − 手順 3 で取得したリモート・サーバの応答をクライアントに送信し、CS_LOGINFO 構造体を解放します。

参照　srv_getloginfo、srv_recvpassthru、srv_sendpassthru

srv_setpri

説明

スレッドのスケジューリング優先順位を変更します。

構文

CS_RETCODE srv_setpri(spp, mode, priority_value)

```
SRV_PROC       *spp;
CS_INT          mode;
CS_INT         priority_value;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

mode

priority_value が現在の優先順位を調節する場合には SRV_C_DELTAPRI、*priority_value* が新しい優先順位である場合には SRV_C_NEWPRI となります。

priority_value

mode が、SRV_C_NEWPRI の場合には、*priority_value* がスレッドの新しい優先順位です。*mode* が SRV_C_DELTAPRI の場合には、負の *priority_value* は現在の優先順位からその絶対値を減らし、正の *priority_value* は現在の優先順位を上げます。

戻り値

表 *3-128: 戻り値 (srv_setpri)*

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

```
#include    <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE       ex_srv_setpri PROTOTYPE((
SRV_PROC         *spp,
CS_INT           mode,
CS_INT           priority
));

/*
** EX_SRV_SETPRI
**
**    Example routine to change a thread's scheduling priority.
**
```

```
** Arguments:
**  spp         A pointer to an internal thread control structure.
**  mode        Indicates whether a priority is relative or
 **               absolute.
**  priority    The change in priority value or the nrew
                  priority value.
**
** Returns:
**
**    CS_SUCCEED
**    CS_FAIL
*/
CS_RETCODE    ex_srv_setpri(spp, mode, priority)
SRV_PROC      *spp;
CS_INT        mode;
CS_INT        priority;
{
    return(srv_setpri(spp, mode, priority));
}
```

使用法

- クライアントが Open Server にログインした結果、あるいは srv_createproc や srv_spawn の呼び出しの結果としてスレッドが始動した場合には、優先順位は SRV_C_DEFAULTPRI になります。
- srv_setpri は、新しい値の設定、または現在の値を指定した値まで上昇／減少させることによって、優先順位を変更することができます。
- あるスレッドが、自分の優先順位よりも、他のスレッドの優先順位を高く設定した場合には、後者のスレッドはただちに優先順位が高くなるようスケジュールされます。それ以外の場合には、影響を受けたスレッドの新しい優先順位は、次にスケジューラが実行されるときから有効になります。
- スリープすることのないスレッドに他のスレッドよりも高い優先順位を持たせると、他のスレッドを実行する機会はまったくなくなります。
- 内部 Open Server スレッドは、SRV_C_DEFAULTPRI の優先順位で実行します。スレッドの優先順位を SRV_C_DEFAULTPRI 以上に設定した場合には、このような内部スレッドの実行を可能にするために、ときおりスリープさせる必要があります。
- 優先順位を SRV_C_LOWPRIORITY より低く設定したり、SRV_C_MAXPRIORITY より高く設定したりすると、エラーになります。
- srv_setpri は、SRV_START ハンドラでは使用できません。

参照　srv_createproc、srv_spawn

srv_signal (UNIX のみ)

説明

シグナル・ハンドラをインストールします。

構文

```
SRV_SIGNAL_FUNC srv_signal(sig, handler)
CS_INT                  sig;
SRV_SIGNAL_FUNC handler;
```

パラメータ

sig

ハンドラがインストールされる対象の UNIX シグナルの番号です。これは、*sgs/signal.h* で定義されます。

handler

sig が Open Server に対して配信されるときに、呼び出される関数へのポインタです。*handler* を SIG_DFL に設定すると、デフォルト・ハンドラがリストアされます。*handler* を SIG_IGN に設定すると、*sig* が無視されます。

戻り値

表 3-129: 戻り値 (srv_signal)

戻り値	意味
以前にインストールされたハンドラ関数へのポインタ	関数のロケーション。
null ポインタ	ルーチンが失敗した。

例

```
#include    <errno.h>
#include    <stdio.h>
#include    <signal.h>
#include    <ospublic.h>
/*
** Local Prototype.
*/
CS_STATIC CS_VOID         ex_sigio_handler PROTOTYPE((
CS_INT                    sig
));

CS_RETCODE       ex_srv_signal PROTOTYPE((
CS_INT           *uerrno
));

/*
** Static storage.
*/
CS_STATIC CS_INT io_events = 0;

/*
** EX_SRV_SIGNAL
**
**    Example routine to install a UNIX signal handler for SIGIO,
 **    using srv_signal.
**
** Arguments:
**    uerrno    A pointer to a user's error number indicator.
```

```
**
** Returns:
**
**    CS_SUCCEED    Handler successfully installed.
**    CS_FAIL       Handler not installed, UNIX global errno set.
*/
CS_RETCODE      ex_srv_signal(uerrno)
CS_INT          *uerrno;
{
      /*
      ** Install the handler.
      */
      (CS_VOID)srv_signal((int)SIGIO,
                          (SRV_SIGNAL_FUNC)ex_sigio_handler);

     /* Was there an error condition? */
     if ((*uerrno = errno) != 0)
         return(CS_FAIL);

      return(CS_SUCCEED);
}

/*
** EX_SIGIO_HANDLER
**
**    Example signal handler to count I/O events.It prints a
 **    message when the Open Server application has been up long
 **    enough to get 100,000 I/O events.
**
** Arguments:
**    sig  The signal number, always SIGIO.
**
** Returns:
**    Nothing.
*/
CS_STATIC CS_VOID     ex_sigio_handler(sig)
CS_INT                sig;
{
     if (io_events == 100000)
      {
           fprintf(stderr, "The server has been up a long
                    time!!¥n");
           io_events = 0;
      }
      else
      {
           io_events++;
      }
}
```

使用法

- Open Server は、SIGIO と SIGURG の UNIX シグナル・ハンドラをインストールします。これらのハンドラは、一度 Open Server が起動すると、常にアクティブの状態でなければなりません。ハンドラがアクティブの状態でない場合には、Server-Library の I/O やアテンション処理ルーチンは機能に障害を生じるか、あるいは信頼性が損なわれます。

  警告！ sigvec(2) または signal(2) を使って、UNIX シグナル・ハンドラをインストールすると、予期できない結果を引き起こす場合があります。アプリケーションでは、srv_signal を使用するようにしてください。

- Open Server は、アプリケーションがシグナル・ハンドラにある間、他のすべてのシグナルがブロックされることを保証します。
- この他の情報は、signal に関する UNIX のマニュアルを参照してください。

srv_sleep

説明

現在実行中のスレッドを休止します。

構文

```
CS_RETCODE srv_sleep(sleepeventp, sleeplabelp,
                sleepflags, infop, reserved1,
               reserved2)

CS_VOID       *sleepeventp;
CS_CHAR        *sleeplabelp;
CS_INT         sleepflags;
CS_INT        *infop;
CS_VOID       *reserved1;
CS_VOID       *reserved2;
```

パラメータ

sleepeventp

srv_wakeup は、スレッドをウェイクアップするために使う汎用の void ポインタです。このポインタは、スレッドがスリープ中のオペレーティング・システム・イベントに対してユニークなものでなければなりません。たとえば、メッセージが別のスレッドに渡される場合、送信スレッドは、そのメッセージが処理されるまでスリープします。メッセージへのポインタは、受信スレッドが送信側をウェイクアップさせるために srv_wakeup に渡せるような役立つ *sleepevent* となります。

sleeplabelp

スレッドがスリープ中のイベントを識別する、null で終了する文字列へのポインタです。これは、スレッドがスリープしている理由を判別するのに役立ちます。アプリケーションは、Open Server システム・レジスタード・プロシージャの sp_ps を使って、この情報を表示することができます。

sleepflags

このフラグの値は、スレッドがウェイクアップする状況を決定します。表 3-130 に、*sleepflags* の有効値を示します。

表 3-130: sleepflags の値 (srv_sleep)

値	説明
SRV_M_ATTNWAKE	スレッドがアテンションを受け取るとウェイクアップする。
SRV_M_NOATTNWAKE	アテンションは、スレッドをウェイクアップさせられない。

infop

CS_INT を指すポインタです。表 3-131 は、srv_sleep が CS_FAIL を返す場合に **infop* に返される可能性がある値を示します。

表 3-131: infop の値 (srv_sleep)

値	説明
SRV_I_INTERRUPTED	スレッドは、srv_ucwakeup によって無条件にウェイクアップさせられた。
SRV_I_UNKNOWN	その他のエラーが発生した。たとえば、スレッドはすでにスリープしている、または無効となっている。

reserved1

ミューテックスに対するプラットフォーム依存のハンドルです。この引数は、非プリエンプティブ・プラットフォームにおいては無視されます。非プリエンプティブ・プラットフォームにおいては (CS_VOID*) 0 に設定してください。

reserved2

このパラメータは、現時点では使用されていません。0 に設定してください。

戻り値

表 3-132: 戻り値 (srv_sleep)

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

```
#include     <ospublic.h>
/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_sleep  PROTOTYPE((
CS_VOID         *sleepevnt,
CS_CHAR         *sleeplbl,
CS_INT          *infop
));

/*
```

```
** EX_SRV_SLEEP
**
**   This routine will suspend the currently executing thread.
**
**
** Arguments:
**
**
**   sleepevnt A void pointer that srv_wakeup uses to wake up
 **             the thread.
**   sleeplbl  A pointer to a null terminated string that
 **             identifies the event being the thread is sleeping
 **             on.This is primarily used for debugging.
 **    infop    A pointer to a CS_INT that is set to one of the
 **             following values:
 **             SRV_I_INTERRUPTED - srv_ucwakeup
 **             unconditionally woke the thread.
 **             SRV_I_UNKNOWN - Some other error occurred.
**
**
** Returns
**
**    CS_SUCCEED
**    CS_FAIL
**
*/
CS_RETCODE      ex_srv_sleep(sleepevnt,sleeplbl,infop)
CS_VOID         *sleepevnt;
CS_CHAR         *sleeplbl;
CS_INT          *infop;
{
     /* Check arguments.  */
     if(sleepevnt == (CS_VOID *)NULL)
     {
           return(CS_FAIL);
     }
     /*
     ** Using SRV_M_ATTNWAKE means the thread should wake up
      ** unconditionally if it receives an attention.
     */

return(srv_sleep(sleepevnt,sleeplbl,SRV_M_ATTNWAKE,infop,(CS_VOID*)0,(CS_VOID*)0));
}
```

使用法

- srv_sleep は、現時点で実行しているスレッドを中断し、再スケジューリングを開始します。srv_wakeup が同じイベントで呼び出されるまで、スレッドはスリープします。
- *sleepflags* の値によっては、スリープ中のスレッドは、アテンションを受け取ることによって、ウェイクアップすることもできます。

- srv_sleep の呼び出しに続く文で、スレッドは実行を再開します。
- srv_sleep は、SRV_START ハンドラにおいては使用できません。
- srv_sleep を、割り込みレベル・コードから呼び出すことはできません。このルールに違反すると、問題が発生する可能性があります。
- 使用しているプラットフォームがプリエンプティブ・スケジューリングをサポートしているかどうかを調べるには、srv_capability を呼び出してください。
- スレッドがスリープしないうちに起動イベントが発生したら、*reserved1* パラメータはプリ エンプティブ・スケジューリングで起こり得る競合状況を阻止します。プリエンプティブ・スケジューリングの例については、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。

参照　srv_wakeup

srv_spawn

説明　サービス・スレッドを割り付けます。

構文

```
CS_RETCODE srv_spawn(sppp, stacksize, funcp,
        argp, priority)
SRV_PROC        **sppp;
CS_INT           stacksize;
CS_RETCODE       (*funcp)();
CS_VOID          *argp;
CS_INT           priority;
```

パラメータ

sppp

スレッド構造体ポインタへのポインタです。呼び出しが成功した場合は、内部のスレッド構造体のアドレスが *sppp* に返されます。

stacksize

スタックのサイズです。これは、最低 SRV_C_MINSTACKSIZE でなくてはなりません。srv_props によって設定されたスタック・サイズ (SRV_S_STACKSIZE) を使用するには、SRV_DEFAULT_STACKSIZE を指定してください。

funcp

新しく作成されたスレッドのためのエントリ・ポイントである関数へのポインタです。スレッドは、*funcp* に位置するルーチンを実行することで開始します。このルーチンから戻ったり、srv_termproc が呼び出されるとき、スレッドは解放されます。エントリ・ポイント関数ポインタは SRV_C_START_LISTENER であることが必要です。

argp
スレッドが実行を始めるとき、**funcp* のルーチンに渡されるポインタです。エントリ関数ポインタ (*funcp*) が SRV_C_START_LISTENER の場合は、*argp* が CS_TRANADDR 構造を指す必要があります。

priority
生成されたスレッドの基本優先順位を示す SRV_C_LOWPRIORITY から SRV_C_MAXPRIORITY の間の整数です。デフォルトの優先順位は、SRV_C_DEFAULTPRI です。

戻り値

スレッドが正常に生成された場合、srv_spawn は CS_SUCCEED を返します。これは十分な Open Server の内部リソースが使用可能であることだけを保証します。エントリ・ポイント・ルーチンまたはその引数の有効性を検証しません。スレッドが生成できない場合には、srv_spawn は CS_FAIL を返します。

表 3-133: 戻り値 (srv_spawn)

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

```
#include    <stdio.h>
#include    <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE       entryfunc PROTOTYPE((
CS_CHAR          *message
));

CS_RETCODE       ex_srv_spawn PROTOTYPE((
SRV_PROC         *spp,
CS_INT           stacksize,
CS_INT           priority
));

CS_RETCODE    entryfunc(message)
CS_CHAR       *message;
{
      printf("Welcome to a new thread - %s!¥n", message);
      return(CS_SUCCEED);
}

/*
** EX_SRV_SPAWN
**
**    Example routine to allocate a service thread
**
** Arguments:
```

```
**    spp      A pointer to an internal thread control
 **             structure.
**    stacks   The desired thread stack size.
**    priority The desired thread scheduling priority.
**
** Returns:
**
**    CS_SUCCEED
**    CS_FAIL
*/
CS_RETCODE    ex_srv_spawn(spp, stacksize, priority)
SRV_PROC      *spp;
CS_INT        stacksize;
CS_INT        priority;
{
    CS_CHAR      msgarg[20];

    strcpy(msgarg, "come in");

    return(srv_spawn(&spp, stacksize, entryfunc, msgarg,
              priority));
}
```

使用法

- srv_spawn は、イベント駆動型ではなく、どのクライアントとも関連しない「サービス・スレッド」を割り付けます。そのスレッドは、スケジューラの制御の下で実行されます。
- srv_spawn によって作成されたスレッドは、共有デバイスやデータ・オブジェクトにアクセスするなどの、イベント駆動型スレッドに要求されるサービスを提供することが多いため、サービス・スレッドと呼ばれます。
- srv_spawn は、Open Server に新しいスレッドについて知らせ、そのスレッドを実行可能にします。スレッドは、すぐには実行を開始しません。スレッドが実行を始める時点は、生成スレッドの優先順位や他の実行可能なスレッドの優先順位のような多くの要素によって決定されます。
- SRV_S_STACKSIZE で stacksize を設定するために srv_props を呼び出さない場合は、新しいスレッドがデフォルトの stacksize で作成されます。このデフォルトの stacksize は、使用するプラットフォームに応じて異なります。Open Server のネイティブスレッド・バージョンでは、基本となるスレッドのデフォルトの stacksize が使用されます。
- 複数のスレッドにより実行されたコードは、リエントラントでなくてはなりません。
- SRV_TLISTENER スレッド・タイプは動的リスナに使用されます。

- 指定されたホスト名は複数の IPv4 および IPv6 アドレスに変換できます。つまり、動的リスナを起動することで複数のスレッドが作成されることになります。これらのスレッドの SRV_PROC ポインタを取得する唯一の方法は、SRV_LISTEN_PREBIND イベントと SRV_LISTEN_POSTBIND イベントを使用することです。
- エントリ・ポイント関数ポインタ (*funcp*) が SRV_C_START_LISTENER の場合は、*stacksize* と *priority* の両方に CS_UNUSED を指定し、*sppp* を null に設定する必要があります。

参照

srv_callback、srv_createproc、srv_props、srv_termproc

srv_symbol

説明

Open Server のトークン値を読み込み可能な文字列に変換します。

構文

```
CS_CHAR *srv_symbol(type, symbol, lenp)
CS_INT          type;
CS_INT          symbol;
CS_INT         *lenp;
```

パラメータ

type

トークンのタイプです。表 3-134 に、有効なトークンのタイプを示します。

表 3-134: type に対応するトークンのタイプ (srv_symbol)

トークンのタイプ	説明
SRV_DATATYPE	データ型
SRV_EVENT	イベント・タイプ
SRV_DONE	DONE ステータス・タイプ
SRV_ERROR	エラー重大度トークン

symbol

実際のトークン値です。

lenp

返される文字列長を含む CS_INT 変数へのポインタです。

戻り値

表 3-135: 戻り値 (srv_symbol)

戻り値	意味
Open Server のトークン値の、読み込み可能な変換である null ターミネータ文字列を指すポインタ	トークン値。
null ポインタ	Open Server が *type* または *symbol* を認識しない。 Open Server が *lenp* を -1 に設定する。

例

```
#include    <ospublic.h>
/*
 ** Local Prototype
 */
extern CS_RETCODE       ex_srv_symbol PROTOTYPE((
CS_INT type,
CS_INT symbol,
CS_CHAR *namep
));
/*
 ** EX_SRV_SYMBOL
**
**   Retrieve a printable string representation of an Open Server
 **   symbol
**
** Arguments:
**     type         Symbol type
**     symbol       Symbol for which to retrieve string
**     namep        Return symbol string here
** Returns:
 **     CS_SUCCEED       Symbol string was retrieved successfully
 **     CS_FAIL          An error was detected
 */
CS_RETCODE           ex_srv_symbol(type, symbol, namep)
 CS_INT               type;
CS_INT               symbol;
CS_CHAR           *namep;
{
     CS_INT          len;
     namep = srv_symbol(type, symbol, &len);
     if(namep == (CS_CHAR *)NULL)
     {
          return(CS_FAIL);
     }
     return(CS_SUCCEED);
 }
```

使用法

- srv_symbol は、Open Server トークン値を記述する、読み込み可能な null で終了する文字列へのポインタを返します。
- srv_symbol のポインタは、決して上書きされない領域を指すポインタを返すので、同じ文で複数回 srv_symbol を呼び出しても安全です。
- 表 3-136 は、srv_symbol が変換できるトークンをまとめたものです。

表 3-136: 変換可能なトークン (srv_symbol)

トークンのタイプ	トークン	説明
SRV_ERROR	SRV_INFO	エラー重大度タイプ
SRV_ERROR	SRV_FATAL_PROCESS	エラー重大度タイプ
SRV_ERROR	SRV_FATAL_SERVER	エラー重大度タイプ
SRV_DONE	SRV_DONE_MORE	DONE パケット・ステータス・フィールド
SRV_DONE	SRV_DONE_ERROR	DONE パケット・ステータス・フィールド
SRV_DONE	SRV_DONE_FINAL	DONE パケット・ステータス・フィールド
SRV_DONE	SRV_DONE_FLUSH	DONE パケット・ステータス・フィールド
SRV_DONE	SRV_DONE_COUNT	DONE パケット・ステータス・フィールド
SRV_DATATYPE	CS_CHAR_TYPE	Char データ型
SRV_DATATYPE	CS_BINARY_TYPE	binary データ型
SRV_DATATYPE	CS_TINYINT_TYPE	1 バイト integer データ型
SRV_DATATYPE	CS_SMALLINT_TYPE	2 バイト integer データ型
SRV_DATATYPE	CS_INT_TYPE	4 バイト integer データ型
SRV_DATATYPE	CS_REAL_TYPE	real データ型
SRV_DATATYPE	CS_FLOAT_TYPE	float データ型
SRV_DATATYPE	CS_BIT_TYPE	bit データ型
SRV_DATATYPE	CS_DATETIME_TYPE	datetime データ型
SRV_DATATYPE	CS_DATETIME4_TYPE	4 バイト datetime データ型
SRV_DATATYPE	CS_MONEY_TYPE	money データ型
SRV_DATATYPE	CS_MONEY4_TYPE	4 バイト money データ型
SRV_DATATYPE	SRVCHAR	Char データ型
SRV_DATATYPE	SRVVARCHAR	可変長 char データ型
SRV_DATATYPE	SRVBINARY	binary データ型
SRV_DATATYPE	SRVVARBINARY	可変長 binary データ型
SRV_DATATYPE	SRVINT1	1 バイト integer データ型
SRV_DATATYPE	SRVINT2	2 バイト integer データ型
SRV_DATATYPE	SRVINT4	4 バイト integer データ型
SRV_DATATYPE	SRVINTN	integer データ型、null 可能
SRV_DATATYPE	SRVBIT	bit データ型

トークンのタイプ	トークン	説明
SRV_DATATYPE	SRVDATETIME	datetime データ型
SRV_DATATYPE	SRVDATETIME4	4 バイト datetime データ型
SRV_DATATYPE	SRVDATETIMN	datetime データ型、null 可能
SRV_DATATYPE	SRVMONEY	money データ型
SRV_DATATYPE	SRVMONEY4	4 バイト money データ型
SRV_DATATYPE	SRVMONEYN	money データ型、null 可能
SRV_DATATYPE	SRVREAL	4 バイト float データ型
SRV_DATATYPE	SRVFLT8	8 バイト float データ型
SRV_DATATYPE	SRVFLTN	8 バイト float データ型、null 可能
SRV_DATATYPE	SRV_LONGCHAR_TYPE	long char データ型
SRV_DATATYPE	SRV_LONGBINARY_TYPE	long binary データ型
SRV_DATATYPE	SRV_TEXT_TYPE	text データ型
SRV_DATATYPE	SRV_IMAGE_TYPE	image データ型
SRV_DATATYPE	SRV_NUMERIC_TYPE	numeric データ型
SRV_DATATYPE	SRV_DECIMAL_TYPE	decimal データ型
SRV_DATATYPE	SRVVOID	void データ型
SRV_EVENT	SRV_ATTENTION	Open Server イベント・タイプ
SRV_EVENT	SRV_BULK	Open Server イベント・タイプ
SRV_EVENT	SRV_CONNECT	Open Server イベント・タイプ
SRV_EVENT	SRV_CURSOR	Open Server イベント・タイプ
SRV_EVENT	SRV_DISCONNECT	Open Server イベント・タイプ
SRV_EVENT	SRV_DYNAMIC	Open Server イベント・タイプ
SRV_EVENT	SRV_LANGUAGE	Open Server イベント・タイプ
SRV_EVENT	SRV_MSG	Open Server イベント・タイプ
SRV_EVENT	SRV_OPTION	Open Server イベント・タイプ
SRV_EVENT	SRV_RPC	Open Server イベント・タイプ
SRV_EVENT	SRV_START	Open Server イベント・タイプ
SRV_EVENT	SRV_STOP	Open Server イベント・タイプ
SRV_EVENT	SRV_URGDISCONNECT	Open Server イベント・タイプ

参照　srv_descfmt

srv_tabcolname

説明

ブラウズ・モードの結果カラムに結果テーブルを関連付けます。

構文

```
CS_RETCODE srv_tabcolname(spp, colnum, brwsdescp)
SRV_PROC            *spp;
CS_INT               colnum;
CS_BROWSEDESC    *brwsdescp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

colnum

以前に srv_descfmt を使って記述したカラムを識別するために使用された番号です。

brwsdescp

該当するカラムのブラウズ情報を持つ構造体へのポインタです。具体的に、カラムや元のカラム名および名前の長さを含むテーブル (先に srv_tabname で記述された) の番号を含んでいなければなりません。カラムが select 文 (CS_BROWSEDESC 構造体において CS_RENAMED のステータスによって指示された) の中で名前変更されていた場合にのみ、元のカラム名と名前の長さが必要になることに注意してください。「CS_BROWSEDESC 構造体」(47 ページ) を参照してください。

戻り値

表 3-137: 戻り値 *(srv_tabcolname)*

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

```
#include          <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE          ex_srv_tabcolname PROTOTYPE((
SRV_PROC            *spp,
CS_INT              colnum,
CS_BROWSEDESC       *bdp
));

/*
** EX_SRV_TABCOLNAME
**
**    Example routine to associate a browse mode result column
 **    with result tables.
**
** Arguments:
**    spp       A pointer to an internal thread control structure.
**
```

```
**    colnum   The column number.
**
**    bdp      A pointer to the browse descriptor for the
 **             column.
**
** Returns:
**     CS_SUCCEED   If we successfully associated this result
 **                  column with its table.
**
**     CS_FAIL      If an error was detected.
**
*/
CS_RETCODE          ex_srv_tabcolname(spp, colnum, bdp)
SRV_PROC           *spp;
CS_INT             colnum;
CS_BROWSEDESC      *bdp;
{
     CS_RETCODE         result;

     result = srv_tabcolname(spp, colnum, bdp);

     return (result);
}
```

使用法

- srv_tabcolname は、クライアントにブラウズ・モードの結果情報を送るために使用されます。次にアプリケーションが送ることができる情報を示します。
  - 結果カラムがマップするテーブルの名前。
  - クライアント・クエリの select 文内で名前が変更されたカラムの本来の名前。
- カラムはあらかじめ srv_descfmt を使って定義されていなければなりません。
- テーブルはあらかじめ srv_tabname 使って定義されていなければなりません。
- srv_tabcolname は、結果ローのカラムである各結果カラムに対して、一度ずつ呼び出されます。

参照

srv_descfmt、srv_tabname、「ブラウズ・モード」(20 ページ)

srv_tabname

説明

一連のブラウズ・モードの結果と関連付けられたテーブルの名前を提供します。

構文

```
CS_RETCODE srv_tabname(spp, tablenum, tablenamep,
        namelen)
SRV_PROC      *spp;
CS_INT        tablenum;
CS_CHAR       *tablenamep;
CS_INT        namelen;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

tablenum

以降の srv_tabcolname の呼び出しにおいて、テーブルを識別するために使われる番号です。

tablenamep

テーブル名へのポインタです。テーブルには必ず名前があるので、名前が null ということはありません。

namelen

テーブル名の長さをバイト数で示したものです。*namelen* が CS_NULLTERM の場合は、Server Library はテーブル名が null で終了するものとみなします。

戻り値

表 *3-138: 戻り値 (srv_tabname)*

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

```
#include          <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_tabname PROTOTYPE((
SRV_PROC        *sproc,
CS_INT          tablenum,
CS_CHAR         *tablename
));

/*
```

```
** EX_SRV_TABNAME
**    An example routine to provide the name of the table
 **    associated with a set of browse mode results.
**
** Arguments:
**    sproc         A pointer to an internal thread control
 **                   structure.
**    tablenum      The number that will be used to identify
 **                   the table in subsequent calls to

 **                   srv_tabcolname.
**    tablename     A null terminated string specifying the
 **                   table name.
**
** Returns:
**    CS_SUCCEED   If the table is successfully described.
**    CS_FAIL      If an error was detected.
*/
CS_RETCODE         ex_srv_tabname(sproc, tablenum, tablename)
SRV_PROC           *sproc;
CS_INT             tablenum;
CS_CHAR            *tablename;
{
     return( srv_tabname(sproc, tablenum, tablename,
                CS_NULLTERM) );
}
```

使用法

- srv_tabname は、ブラウズ・モードの結果に対応したテーブルの名前をクライアントに送るのに使用されます。
- Open Server アプリケーションは、ブラウズ・モードの結果に関連する各テーブルに対して、一度ずつ srv_tabname を呼び出さなければなりません。
- *tablenum* は、記述されたすべてのテーブルに対してユニークなものでなくてはなりません。テーブルは、いかなる順序でも記述可能です。
- アプリケーションは、ブラウズ・モードの結果カラムを srv_tabcolname ルーチンを使って特定の結果のテーブルにリンクします。srv_tabname は、必ず srv_tabcolname より前に呼び出される必要があります。

参照

srv_descfmt、srv_tabcolname、「ブラウズ・モード」(20 ページ)

srv_termproc

説明

スレッドの実行を中止します。

構文

```
CS_RETCODE srv_termproc(spp)
SRV_PROC     *spp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

戻り値

表 3-139: 戻り値 (srv_termproc)

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

```
#include          <ospublic.h>
/*
 ** Local Prototype.
 */
CS_RETCODE         ex_srv_termproc PROTOTYPE((
SRV_PROC           *spp
));

/*
 ** EX_SRV_TERMPROC
**
**   Example routine to terminate the execution of a thread using
 **   srv_termproc.
**
** Arguments:
**   spp    A pointer to an internal thread control structure.
** Returns:
**
**    CS_SUCCEED    Thread successfully terminated
**    CS_FAIL       An error was detected.
 */
CS_RETCODE          ex_srv_termproc(spp)
SRV_PROC            *spp;
{
      /*
       ** Terminate the thread.
       */
      if (srv_termproc(spp) != CS_SUCCEED)
      {
              return(CS_FAIL);
      }
      return(CS_SUCCEED);
}
```

使用法

- srv_createproc を使用して、Open Server アプリケーションはクライアント接続に関連していないイベント・ドライバ・スレッドを作成できます。
- srv_termproc は、SRV_START ハンドラでは使用できません。
- srv_termproc を割り込みレベル・コードから呼び出さないでください。結果が予期できません。
- ミューテックス、ミューテックス・ロック、レジスタード・プロシージャ、キューされたイベント、およびスレッドに関連したメッセージは、スレッドが終了すると削除されます。
- 次のコードは、srv_termproc の例です。

参照

srv_createproc、srv_event、srv_event_deferred、srv_spawn

srv_text_info

説明

text または image データの記述を設定または取得します。

構文

```
CS_RETCODE srv_text_info(spp, cmd, item, iodescp)
SRV_PROC     *spp;
CS_INT        cmd;
CS_INT        item;
CS_IODESC    *iodescp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

cmd

データ・フローの方向です。表 3-140 に、*cmd* の有効値を示します。

表 *3-140: cmd* の値 *(srv_text_info)*

値	意味
CS_SET	Open Server アプリケーションは、text や image データを記述するための内部 Server-Library 構造体を設定している。srv_text_info の呼び出しは、*iodescp* の情報を使って (Open Server 内部の) text や image データを更新する (アプリケーションは、srv_descfmt を使ってカラムをあらかじめ記述していなければならない)。一般的には、srv_send_text、または srv_bind および srv_xferdata の呼び出しが後に続く。
CS_GET	Open Server はクライアントから読み込む text または image データの全長で *iodescp* 構造体を更新している。一般的には、srv_get_text の呼び出しが次に続く。CS_GET の方向についての制限については、「使用法」の項を参照。

item

記述されるカラムのカラム番号です。ローにある最初のカラムはカラム番号 1 です。*cmd* が CS_GET の時、このパラメータは無視されます。

iodescp

テキスト・カラムのオブジェクト名、テキスト・ポインタ、およびタイム・スタンプを記述する構造体へのポインタです。詳細については、「CS_IODESC 構造体」(52 ページ) を参照してください。

戻り値

表 *3-141: 戻り値 (srv_text_info)*

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

```
#include          <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_text_info PROTOTYPE((
SRV_PROC        *spp,
CS_INT          item
CS_IODESC       *iodp
));

/*
** EX_SRV_TEXT_INFO
**
**     Example routine to set a column's text or image data
 **     description before transferring a data row, using
 **     srv_text_info.This example routine would be used in a
 **     gateway application, where the Open Client application has
 **     initiated an update of text or image data.
**
** Arguments:
**    spp   A pointer to an internal thread control structure.
**    item  The column number of the column being described.
**    iodp  A pointer to a CS_IODESC structure that describes the
 **          text or image data (This stucture is passed from the
 **          Open Client application).
**
** Returns:
**    CS_SUCCEED    Text or image data successfully described.
**    CS_FAIL      An error occurred was detected.
*/
CS_RETCODE      ex_srv_text_info(spp, item, iodp)
SRV_PROC        *spp;
CS_INT          item;
CS_IODESC       *iodp;
```

```
{
        /*
        ** Describe the text or image data for the column.
        */
        if (srv_text_info(spp, (CS_INT)CS_SET, item, iodp) !=
                CS_SUCCEED)
        {
              return(CS_FAIL);
        }
        return(CS_SUCCEED);
}
```

使用法

- srv_text_info は、結果ローを送信したりパラメータを取得したりするために、text や image カラムを記述するために使用されています。
- *cmd* が CS_GET の場合、srv_text_info は SRV_BULK イベント・ハンドラから呼び出さなければなりません。
- *cmd* が CS_GET の場合、srv_text_info は srv_get_text の呼び出しよりも前に、呼び出さなければなりません。
- *cmd* が CS_SET の場合、srv_text_info は、srv_xferdata や srv_send_text が呼び出される前に、ローの各 text や image データ型のカラムに対して呼び出さなければなりません。
- text や image データは、srv_bind、srv_xferdata、または srv_send_text を使って、クライアントに転送されます。

参照

srv_bind、srv_descfmt、srv_get_text、srv_send_text、srv_xferdata、「text と image」(184 ページ)

srv_thread_props

説明

スレッド・プロパティを定義し、取得します。

構文

CS_RETCODE srv_thread_props(spp, cmd, property, bufp, buflen, outlenp)

```
SRV_PROC *spp;
CS_INT      cmd;
CS_INT      property;
CS_VOID     *bufp;
CS_INT      buflen;
CS_INT      *outlenp;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

cmd

実行するアクションです。表 3-142 に、*cmd* の有効値を示します。

表 3-142: cmd の値 (srv_thread_props)

値	意味
CS_SET	Open Server アプリケーションがプロパティを設定している。この場合、*bufp* は、プロパティが設定される値を持ち、*buflen* は、その値のサイズのバイト数でなければならない。
CS_GET	Open Server アプリケーションがプロパティを取得している。この場合、*bufp* は、プロパティ値が置かれているバッファを指し、*buflen* は、そのバッファのサイズのバイト数でなければならない。
CS_CLEAR	Open Server アプリケーションは、プロパティをデフォルト値にリセットしている。この場合は、*bufp*、*buflen*、*outlenp* は無視される。

property
設定、取得、またはクリアされるプロパティです。

bufp
クライアントからのプロパティ値の情報が設定されたり、プロパティ値の情報が取得されたりする Open Server アプリケーション・データ・バッファへのポインタです。

buflen
バッファ長をバイト数で示したものです。

outlenp
取得されたプロパティ値の長さのバイト数を設定するために Open Server が使用する CS_INT 変数へのポインタです。この引数はオプションであり、*cmd* が CS_GET のときにのみ使用します。

戻り値

表 3-143: 戻り値 (srv_thread_props)

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

```
#include <ospublic.h>
/*
** Local Prototype.
*/
CS_RETCODE ex_srv_thread_props PROTOTYPE((
SRV_PROC   *sp,
CS_CHAR    *user,
CS_INT     ulen,
CS_INT     *lenp
));
/*
** EX_SRV_THREAD_PROPS
**
** Example routine to obtain a client thread's user name through
** srv_thread_props.
**
```

```
** Arguments:
** sp     A pointer to an internal thread control structure.
** user   A pointer to the address of the user name buffer.
** ulen   The size of the user name buffer.
** lenp  A pointer to an integer variable, that will be set to the length
**         of the user name string.
**
** Returns:
** CS_TRUE   If the user name was returned succesfully.
** CS_FALSE  If an error was detected.
*/
CS_RETCODE ex_srv_thread_props(sp, user, ulen, lenp)
SRV_PROC   *sp;
CS_CHAR    *user;
CS_INT     ulen;
CS_INT     *lenp;
{
   /*
    ** Call srv_thread_props to get the user name.
   */
   if( srv_thread_props(sp, CS_GET, SRV_T_USER, user, ulen, lenp)
      == CS_FAIL )
   {
      /*
      ** An error was al&ready raised.
      */
      return CS_FAIL;
   }
/*
** All done.
*/
return CS_SUCCEED;
}
```

使用法

- srv_thread_props は、スレッド・プロパティを定義、取得、リセットするために呼び出されます。
- 表 3-144 に、有効なプロパティ値、設定や取得の可／不可、それぞれの値のデータ型を示します。

  各スレッド・プロパティの説明については、表 2-28 (139 ページ) を参照してください。

表 *3-144:* スレッド・プロパティとそのデータ型 *(srv_thread_props)*

プロパティ	設定／クリア	取得	cmd が CS_SET のときの bufp	cmd が CS_GET のときの bufp
SRV_T_APPLNAME	不可	可	適用しない	文字列へのポインタ
SRV_T_BYTEORDER	不可	可	適用しない	CS_INT へのポインタ
SRV_T_BULKTYPE	不可	可	適用しない	CS_INT へのポインタ
SRV_T_BYTEORDER	不可	可	適用しない	CS_INT へのポインタ
SRV_T_CHARTYPE	不可	可	適用しない	CS_INT へのポインタ
SRV_T_CLIB	不可	可	適用しない	文字列へのポインタ
SRV_T_CLIBVERS	不可	可	適用しない	文字列へのポインタ
SRV_T_CLIENTLOGOUT	不可	可	適用しない	CS_BOOL へのポインタ
SRV_T_CONVERTSHORT	不可	可	適用しない	CS_BOOL へのポインタ
SRV_T_DUMPLOAD	不可	可	適用しない	CS_BOOL へのポインタ
SRV_T_ENDPOINT	不可	可	適用しない	終了ポイント (ファイル記述子またはファイル処理) を保持できるサイズのバッファへの CS_VOID ポインタ
SRV_T_EVENT	不可	可	適用しない	CS_INT へのポインタ
SRV_T_EVENTDATA	不可	可	適用しない	CS_VOID ポインタのアドレス
SRV_T_FLTTYPE	不可	可	適用しない	CS_INT へのポインタ
SRV_T_GOTATTENTION	不可	可	適用しない	CS_BOOL へのポインタ
SRV_T_HOSTNAME	不可	可	適用しない	文字列へのポインタ
SRV_T_HOSTPROCID	不可	可	適用しない	文字列へのポインタ
SRV_T_IODEAD	不可	可	適用しない	CS_BOOL へのポインタ
SRV_T_LISTENADDR	不可	可	適用しない	CS_TRANADDR 構造体へのポインタ
SRV_T_LOCALE	可	可	CS_LOCALE ポインタへのポインタ	CS_LOCALE ポインタへのポインタ
SRV_T_LOCALID	可	不可	適用しない	リスナ SSL 証明書
SRV_T_LOGINTYPE	不可	可	適用しない	CS_INT へのポインタ
SRV_T_MACHINE	不可	可	適用しない	文字列へのポインタ
SRV_T_MIGRATED	不可	可	適用しない	CS_BOOL へのポインタ
SRV_T_MIGRATE_STATE	不可	可	適用しない	SRV_MIG_STATE へのポインタ
SRV_T_NEGLOGIN	不可	可	適用しない	CS_INT へのポインタ
SRV_T_NOTIFYCHARSET	不可	可	適用しない	CS_BOOL へのポインタ
SRV_T_NOTIFYDB	不可	可	適用しない	CS_BOOL へのポインタ
SRV_T_NOTIFYLANG	不可	可	適用しない	CS_BOOL へのポインタ
SRV_T_NOTIFYPND	不可	可	適用しない	CS_INT へのポインタ
SRV_T_NUMRMTPWDS	不可	可	適用しない	CS_INT へのポインタ
SRV_T_PACKETSIZE	不可	可	適用しない	CS_INT へのポインタ
SRV_T_PASSTHRU	不可	可	適用しない	CS_BOOL へのポインタ
SRV_T_PRIORITY	不可	可	適用しない	CS_INT へのポインタ

プロパティ	設定／クリア	取得	cmd が CS_SET のときの bufp	cmd が CS_GET のときの bufp
SRV_T_PWD	不可	可	適用しない	文字列へのポインタ
SRV_T_REMOTEADDR	不可	可	適用しない	CS_TRANADDR 構造体へのポインタ
SRV_T_RETPARMS	不可	可	適用しない	実行時にエラーが発生した場合に送信されるリターン・パラメータ
SRV_T_RMTPWDS	不可	可	適用しない	SRV_RMTPWD 構造体の配列へのポインタ
SRV_T_RMTSERVER	不可	可	適用しない	文字列へのポインタ
SRV_T_ROWSENT	不可	可	適用しない	CS_INT へのポインタ
SRV_T_SEC_CHANBIND	不可	可	適用しない	CS_BOOL へのポインタ
SRV_T_SEC_CONFIDENTIALITY	不可	可	適用しない	CS_BOOL へのポインタ
SRV_T_SEC_CREDTIMEOUT	不可	可	適用しない	CS_INT へのポインタ
SRV_T_SEC_DATAORIGIN	不可	可	適用しない	CS_BOOL へのポインタ
SRV_T_SEC_DELEGATION	不可	可	適用しない	CS_BOOL へのポインタ
SRV_T_SEC_DELEGCRED	不可	可	適用しない	CS_VOID へのポインタ
SRV_T_SEC_DETECTREPLAY	不可	可	適用しない	CS_BOOL へのポインタ
SRV_T_SEC_DETECTSEQ	不可	可	適用しない	CS_BOOL へのポインタ
SRV_T_SEC_INTEGRITY	不可	可	適用しない	CS_BOOL へのポインタ
SRV_T_SEC_MECHANISM	不可	可	適用しない	CS_CHAR へのポインタ
SRV_T_SEC_MUTUALAUTH	不可	可	適用しない	CS_BOOL へのポインタ
SRV_T_SEC_NETWORKAUTH	不可	可	適用しない	CS_BOOL へのポインタ
SRV_T_SEC_SESSTIMEOUT	不可	可	適用しない	CS_INT へのポインタ
SRV_T_SESSIONID	可	可	CS_SESSIONID へのポインタ	CS_SESSIONID へのポインタ
SRV_T_SPID	不可	可	適用しない	CS_INT へのポインタ
SRV_T_STACKLEFT	不可	可	適用しない	CS_INT へのポインタ
SRV_T_TDSVERSION	可	可	CS_INT へのポインタ	CS_INT へのポインタ
SRV_T_TYPE	不可	可	適用しない	CS_INT へのポインタ
SRV_T_USER	不可	可	適用しない	文字列へのポインタ
SRV_T_USERDATA	可	可	CS_VOID ポインタ	CS_VOID ポインタのアドレス
SRV_T_USERVLANG	可	可	CS_BOOL へのポインタ	CS_BOOL へのポインタ
SRV_T_USTATE	可	可	文字列へのポインタ	文字列へのポインタ

- 表 3-145 に、定義可能なスレッド・プロパティのデフォルト値を示します (CS_SET)。

表 3-145: 定義できるスレッド・プロパティとそのデフォルト値 (srv_thread_props)

プロパティ	デフォルト
SRV_T_USERDATA	(CS_VOID *)NULL
SRV_T_USTATE	NULL 文字列
SRV_T_TDSVERSION	最小値 (クライアントおよびサーバのデフォルト)
SRV_T_USESRVLANG	SRV_S_USESRVLANG の値
SRV_T_LOCALE	(CS_LOCALE *)NULL

- スレッド・プロパティを取得しているとき (CS_GET)、プロパティ値がユーザ・バッファに入り切らないことを *buflen* が示した場合には、Open Server は、*outlenp に要求されるバイト数を設定し、アプリケーション・バッファは変更されません。
- 各スレッド・プロパティの説明については、表 2-28 (139 ページ) を参照してください。

参照

srv_props、「プロパティ」(130 ページ)

srv_timedsleep

説明

イベントが通知されるまで、または指定された時間が経過するまでスリープします。srv_timedsleep が利用できるのはリエントラント・ライブラリのみです。

構文

```
CS_RETCODE srv_timedsleep(sleepevent, sleeplabel,
                sleepflags, infop, srvmutex, timeout)

CS_VOID         *sleepevent;
CS_CHAR         *sleeplabel;
CS_INT          sleepflags;
CS_VOID         *infop;
SRV_OBJID       srvmutex;
CS_INT          timeout;
```

パラメータ

sleepevent

スリープが発生するイベントへの汎用のポインタです。

sleeplabel

デバッグを目的とした文字列へのポインタです。

sleepflags
このパラメータは、現在実行中のスレッドを中断する srv_sleep と同じように使用されます。

infop
失敗の理由を示す整数へのポインタです。*infop* の整数値は次のとおりです。

- SRV_I_UNKNOWN － 不明なエラーまたはエラーなし。
- SRV_I_TIMEOUT － ルーチンのタイムアウト。
- SRV_I_INTERRUPTED － この関数を実行している srvlib プロセスが、srv_ucwakeup() への呼び出しによって中断された。

注意　この関数が SRV_I_INTERRUPTED を返す場合、イベントを待機している間、またはミューテックスをロックしようとしている間に srvlib プロセスが中断されたことを示します。

---

srvmutex
スリープ時に解放される srvlib ミューテックスです。これは、ウェイクアップ後にロックされます。srv_timedsleep() がミューテックスの解放とロックを実行しないようにするには、*0* を入力してください。

timeout
ミリ秒単位のタイムアウト時間です。無期限にブロックする場合は *0* を渡します。

戻り値

表 3-146: 戻り値 (srv_timedsleep)

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。詳細については、*infop* パラメータを参照。

使用法

この関数にミューテックスを渡して、ウェイクアップと同期化させることができます。ミューテックス・ロックを取得した後に srv_wakeup() を呼び出す別のスレッドが、このイベントについて、このスリープ関数を実行している srvlib プロセスを正常にウェイクアップした時点で、ミューテックスが解放されます。

ルーチンが CS_SUCCEED を返すと、srvlib ミューテックスはロックされます。CS_FAIL を返した場合は、このスレッドによってロックされません。

参照

srv_wakeup

srv_ucwakeup

説明

スリープしているスレッドを無条件にウェイクアップします。

構文

```
CS_RETCODE srv_ucwakeup(spp, wakeflags)
SRV_PROC    *spp;
CS_INT       wakeflags;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

wakeflags

srv_ucwakeup の動作を変えるビットマスクです。1 つのフラグのみ定義されます。使用しない場合は *wakeflags* に 0 を設定します。

SRV_M_WAKE_INTR

割り込みレベル・コードからの srv_ucwakeup の呼び出しであることを示します。割り込みレベル・コードからの srv_ucwakeup 呼び出しを行うときにこのフラグを設定しないと、Open Server アプリケーションは不安定な動作をすることがあります。

戻り値

表 *3-147:* 戻り値 *(srv_ucwakeup)*

戻り値	意味
CS_SUCCEED	ルーチンが正常に終了した。
CS_FAIL	スレッドが存在しないか、スリープ中でなかったため、ルーチンが失敗します。

例

```
&num;include <ospublic.h>
/*
** Local Protoype.
*/

CS_RETCODE      ex_srv_ucwakeup PROTOTYPE((
SRV_PROC        *sproc
));

/*
** EX_SRV_PROC
** An example routine to wake up a sleeping thread from
** a non-interrupt level by using srv_ucwakeup.
**
** Arguments:
** sproc   A pointer to an internal thread control
** structure.
**
** Returns:
** CS_SUCCEED  The specified thread was woken up.
** CS_FAIL      An error was detected.
*/

CS_RETCODE      ex_srv_ucwakeup(sproc)
SRV_PROC        *sproc;
```

```
{
/* Wake up the specified thread. */
return( srv_ucwakeup(sproc, 0));
}
```

使用法

- srv_ucwakeup でスレッドをウェイクアップすると、srv_sleep は SRV_I_INTERRUPTED を返します。
- スレッドを無条件にウェイクアップするには、srv_ucwakeup を使用します。デッドロック状態の解除やクリーンアップでは、これが必要な場合があります。
- srv_ucwakeup は、SRV_START ハンドラにおいては使用できません。
- srv_ucwakeup が割り込みレベル・コードから呼び出された場合は、*wakeflags* に SRV_M_WAKE_INTR を設定する必要があります。割り込みレベル・ルーチンの外部では、*wakeflags* に SRV_M_WAKE_INTR を設定しないでください。

参照

srv_sleep、srv_wakeup、srv_yield

srv_unlockmutex

説明

ミューテックスのロックを解除します。

構文

```
CS_RETCODE srv_unlockmutex(mutexid)
SRV_OBJID          mutexid;
```

パラメータ

mutexid

srv_createmutex によって返されるユニークなミューテックス識別子です。mutexid は、srv_getobjid を使ってミューテックス名から得られます。

戻り値

表 *3-148: 戻り値 (srv_unlockmutex)*

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

```
#include    <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_unlockmutex PROTOTYPE((
CS_CHAR         *mutex_name
));

/*
** EX_SRV_UNLOCKMUTEX
**
**    Example routine to illustrate the use of srv_unlockmutex.
**
** Arguments:
**  mutex_name   The name of the mutex to be unlocked.
**
** Returns:
**
**    CS_SUCCEED Mutex successfully unlocked.
**    CS_FAIL      An error was detected.
*/
CS_RETCODE     ex_srv_unlockmutex(mutex_name)
CS_CHAR        *mutex_name;
{
     SRV_OBJID      id;
     CS_INT        info;

      /* Get the object id for the mutex. */
      if (srv_getobjid(SRV_C_MUTEX, mutex_name, CS_NULLTERM,
            &id,  &info) == CS_FAIL)
           return (CS_FAIL);

     /* Call srv_unlockmutex to unlock it. */
    if (srv_unlockmutex(id) == CS_FAIL)
      return (CS_FAIL);

      return (CS_SUCCEED);
}
```

使用法

- ミューテックス (相互排除セマフォ) のロックの解除は、他のスレッドをミューテックスからアクセスできるように、セマフォのロックを解放します。
- srv_unlockmutex は、SRV_START ハンドラでは使えません。

参照

srv_createmutex、srv_deletemutex、srv_getobjid

srv_version

説明

アプリケーションが使用している Open Server のバージョンを定義します。

構文

```
CS_RETCODE srv_version(contextp, version)
CS_CONTEXT      *contextp;
CS_INT          version;
```

パラメータ

contextp

アプリケーションが cs_ctx_alloc の呼び出しによって得た CS_CONTEXT 構造体へのポインタです。CS_CONTEXT 構造体は、クライアント・ライブラリと共有した、サーバワイドな設定構造体の役割を果たしています。「CS-Library」(53 ページ) を参照してください。

version

アプリケーションが有効であると仮定する Open Server のバージョンです。

戻り値

表 *3-149: 戻り値 (srv_version)*

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

```
#include <stdio.h>
#include <ospublic.h>
......
/*
 ** This code fragment sets the Open Server version.
 */
main()
 {
CS_CONTEXT *cp;
if(cs_ctx_alloc(CS_VERSION_155, &cp) != CS_SUCCEED)
 {
     fprintf(stderr, "cs_ctx_alloc failed ¥n");
      exit(1);
}
if(srv_version(cp, CS_VERSION_155) != CS_SUCCEED)
{
     /*
      ** Release the context structure al&ready allocated.
      */
     (CS_VOID)cs_ctx_drop(cp);
     (CS_VOID)fprintf(stderr, "srv_version failed ¥n");
       exit(1);
}
......
}
```

使用法

- Open Server アプリケーションは、他の Server-Library ルーチンを呼び出す前に、srv_version を呼び出さなければなりません。また、srv_version の呼び出しは CS-Library ルーチンの cs_ctx_alloc を呼び出してから行う必要があります。
- アプリケーションは、最初に、cs_config を使って CS_CONTEXT 構造体で、ローカライゼーション設定のパラメータを設定できます。

参照

cs_ctx_alloc、cs_ctx_props

srv_wakeup

説明

スリープ中のスレッドの実行を可能にします。

構文

```
CS_RETCODE srv_wakeup(sleepeventp, wakeflags,
        reserved1, reserved2)

CS_VOID     *sleepeventp;
CS_INT      wakeflags;
CS_VOID     *reserved1;
CS_VOID     *reserved2;
```

パラメータ

sleepeventp

スレッドがスリープ中のオペレーティング・システム・イベントの汎用の void ポインタです。

wakeflags

srv_wakeup の動作を変えるビットマスクです。ビットが設定されていない場合、デフォルトの動作は、イベント上でスリープしているすべてのスレッドをウェイクアップさせることです。ビットは論理和をとることができます。表 3-150 に、*wakeflags* の有効値を示します。

表 *3-150: wakeflags* の値 *(srv_wakeup)*

値	説明
SRV_M_WAKE_INTR	srv_wakeup の呼び出しは、割り込みレベル・コードから行う。割り込みレベル・コードからの srv_wakeup 呼び出しを行うときにこのフラグを使用しないと、Open Server アプリケーションは不安定な動作をすることがある。 非割り込みレベルでのこのフラグを使用すると、Open Server アプリケーションは不安定な動作をする。
SRV_M_WAKE_FIRST	イベント上でスリープしている最初のスレッドだけが実行可能にされる。
SRV_M_WAKE_ALL	イベント上でスリープしているすべてのスレッドをウェイクアップさせる。

reserved1

このパラメータは、使用されません。(CS_VOID*) 0 に設定してください。

reserved2

このパラメータは、使用されません。(CS_VOID*) 0 に設定してください。

戻り値

srv_wakeup は、イベントのスリープ中のスレッドが見つからなかった場合やパラメータがエラーの場合は、CS_FAIL を返します。1 つまたは複数のスリープ中のスレッドが見つかった場合は、srv_wakeup は CS_SUCCEED を返します。

表 3-151: 戻り値 (srv_wakeup)

戻り値	意味
CS_SUCCEED	1 つまたは複数のスリープ中のスレッドが見つかり、実行可能だった。
CS_FAIL	ルーチンが失敗。またはスリープ中のスレッドが見つからなかった。

例

```
#include    <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE ex_srv_wakeup PROTOTYPE((
CS_VOID    *sep
));

/*
** EX_SRV_WAKEUP
**
**    Example routine using srv_wakeup to make all Open Server
 **    threads, which were previously sleeping on the specified
 **    sleep event, runnable again.
**
** Arguments:
**    sep    A generic void pointer, which was used previously in
 **           calls to srv_sleep to suspend threads.
**
** Returns:
**    CS_SUCCEED    Threads sleeping on the specified sleep event
 **                   are runnable again.
**    CS_FAIL       An error was detected.
*/
CS_RETCODE      ex_srv_wakeup(sep)
CS_VOID         *sep;
{
      /*
      ** Wake up threads for the specified sleep event, passing
       ** zero for reserved fields.
      */
      if (srv_wakeup(sep, (CS_INT)SRV_M_WAKE_ALL,
           (CS_VOID*)0, (CS_VOID*)0) != CS_SUCCEED)
     {
           return(CS_FAIL);
     }
     return(CS_SUCCEED);
}
```

使用法

- srv_wakeup は、*sleepevent* でスリープ中のスレッドをウェイクアップさせます。
- srv_wakeup が割り込みレベル・コードから呼び出される場合、実際のウェイクアップは、スケジューラが次に実行するまで遅延されます。
- srv_wakeup は、SRV_START ハンドラにおいては使用できません。
- Open Server でプリエンプティブ・プログラムを書く場合、srv_wakeup や srv_sleep は、プラットフォームに依存するミューテックスを使わなければなりません。プリエンプティブ・スケジューリングの例については、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。

参照

srv_sleep

srv_xferdata

説明

クライアントにパラメータやデータを送信するか、またはクライアントからパラメータやデータを受信します。

構文

```
CS_RETCODE srv_xferdata(spp, cmd, type)
SRV_PROC        *spp;
CS_INT           cmd;
CS_INT           type;
```

パラメータ

spp

内部スレッド制御構造体へのポインタです。

cmd

データがクライアントへ送信されているのか、それともクライアントから入ってきているのか示します。表 3-152 に、*cmd* の有効値を示します。

表 3-152: *cmd* の値 (srv_xferdata)

値	説明
CS_SET	アプリケーションは、クライアントにデータを送信するために srv_xferdata を呼び出している。
CS_GET	アプリケーションは、クライアントからのデータを取得するために srv_xferdata を呼び出している。

type

プログラム変数に保管されたり、プログラム変数から読み取られるデータの型です。表 3-153 に、有効な型とその適切なコンテキストを示します。

表 *3-153:* 型の値 *(srv_xferdata)*

型	有効な cmd	データの内容
SRV_RPCDATA	CS_SET または CS_GET	RPC パラメータ
SRV_ROWDATA	CS_SET のみ	結果ロー・カラム
SRV_CURDATA	CS_GET のみ	カーソル・パラメータ
SRV_KEYDATA	CS_GET のみ	カーソル・キー・カラム
SRV_ERRORDATA	CS_SET のみ	エラー・メッセージ・パラメータ
SRV_DYNDATA	CS_SET または CS_GET	動的 SQL パラメータ
SRV_NEGDATA	CS_SET または CS_GET	ネゴシエーション・ログイン・パラメータ
SRV_MSGDATA	CS_SET または CS_GET	メッセージ・パラメータ
SRV_LANGDATA	CS_GET のみ	言語パラメータ

戻り値

表 *3-154:* 戻り値 *(srv_xferdata)*

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

```
#include   <ospublic.h>
/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_xferdata PROTOTYPE((
SRV_PROC       *spp
));

/*
** EX_SRV_XFERDATA
**
**    This routine will send error message parameters to the
 **    specified client.
**
**
** Arguments:
**
**    spp    A pointer to an internal thread control structure.
**
** Returns
**
**    CS_SUCCEED
**    CS_FAIL
**
*/
CS_RETCODE      ex_srv_xferdata(spp)
SRV_PROC        *spp;
```

```
{
    /* Check arguments.   */
    if (spp == (SRV_PROC *)NULL)
    {
        return(CS_FAIL);
    }
    return(srv_xferdata(spp,CS_SET,SRV_ERRORDATA));
}
```

使用法

- srv_xferdata は、クライアントにパラメータまたはローのデータを送信したり (CS_SET)、クライアントからパラメータまたはキー・データを取得したりするために使用します。具体的には、データをローカル・プログラム変数から取り出してネットワークを通してクライアントに送信したり (CS_SET)、逆にクライアントからネットワークを通してデータをローカル・プログラム変数に設定したりします (CS_GET)。
- クライアントに表示しなければならない形のデータ (CS_SET) やクライアントに表示した形のデータ (CS_GET) は、事前に srv_descfmt を使って記述しておく必要があります。また、アプリケーションは、ローカル・プログラム変数を定義するために srv_bind を事前に呼び出しておく必要もあります。
- srv_xferdata は、各パラメータ・ストリーム (CS_GET、CS_SET) に対して一度ずつ、または各データ・ロー (CS_SET) に対して一度ずつ呼び出されなければなりません。

参照 srv_bind、srv_descfmt

srv_yield

説明 他のスレッドの実行を可能にします。

構文 CS_RETCODE srv_yield()

戻り値 なし。

例

```
#include   <stdio.h>
#include   <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE ex_srv_yield PROTOTYPE((
    ));
/*
** EX_SRV_YIELD
**
**   Example routine to suspend the current thread.
```

```
** Arguments:
**    None.
**
** Returns:
**
**    CS_SUCCEED
**    CS_FAIL
*/
CS_RETCODE    ex_srv_yield()
{
    printf("I'll wait this one out...¥n");
    if (srv_yield() == CS_FAIL)
    {
        printf("srv_yield() failed.¥n");
        return(CS_FAIL);
    }
    else
    {
        printf("I'm back!¥n");
        return(CS_SUCCEED);
    }
}
```

使用法

- srv_yield は、現在のスレッドを中断し、同じかそれ以上の優先順位を持った他の実行可能なスレッドを起動させます。そのスレッドは、後で再スケジュールされます。
- srv_yield は、基本的には非プリエンプティブ・スケジューリング時に役立ちます。
- スレッドが srv_yield を呼び出して、確立されている新しいスレッドを起動する場合、次のように動作します。

  a　Open Server は新しいスレッドの確立を完了します。

  b　新しいスレッドが実行可能にならない場合、その新しいスレッドは制御されないで、即座に現在のスレッドが再び制御されるようになると考えられます。

  「マルチスレッド・プログラミング」(102 ページ) を参照してください。
- srv_yield を呼び出すスレッドが、srv_yield の後に続く文を受けて実行を再開します。
- srv_yield は、SRV_START ハンドラでは使用できません。
- 割り込みレベル・コードから srv_yield は呼び出さないでください。

参照

srv_sleep、srv_wakeup

第 4 章

システム・レジスタード・プロシージャ

この章では、Server-Library の各システム・レジスタード・プロシージャについて説明します。システム・レジスタード・プロシージャは、Open Server に組み込まれているレジスタード・プロシージャです。サーバは、これらのプロシージャをすべての Open Server ランタイム・システムで使えるように、初期化のときに登録します。各プロシージャのページでは、パラメータ、それぞれが返す結果、メッセージについて説明します。

システム・レジスタード・プロシージャの詳細については、「レジスタード・プロシージャ」(151 ページ) を参照してください。

sp_ps

説明

指定された Open Server のスレッドの詳細なステータス情報を返します。

構文

sp_ps [*loginame* | '*spid*']

パラメータ

loginame

ユーザのログイン名です。

SPID

レポートの対象となるスレッドの内部 ID 番号です。*spid* は、以前の sp_who または sp_ps 呼び出しから得られます。デフォルトでは、すべてのスレッドがリストされます。

例

```
1>execute utility...sp_ps
 2>go


 spid  Login Name Host Name     Program Name  Task Type          ...
 ----  ---------- ---------     ------------  --------------     ...
    1                                        SERVER TASK        ...
    2                                        SERVER TASK        ...
    3                                        SERVER TASK        ...
    4                                        SERVICE TASK       ...
   11             hiram                      SITE HANDLER TASK  ...
   14  bud        sonoma        isql         CHILD TASK         ...
...   Status     Sleep Event   Sleep Label       Current Command    ...

 ...   -------    -----------   -----------       ---------------    ...
 ...   runnable   369448                          NETWORK HANDLER    ...
 ...   sleeping   369544        MSG AVAILABLE     CONNECT HANDLER    ...
 ...   sleeping   369640        MSG AVAILABLE     DEFERRED HANDLER   ...
 ...   runnable        0                          SCHEDULER          ...
 ...   sleeping   369736        MSG AVAILABLE                        ...
 ...   running    416480                                             ...

  ...  Blocked    Run     Current     Stack      Net        Net
  ...       By  Ticks    Priority    Origin    Writes     Reads
  ...  -------  ------   --------   --------   ------   -----
  ...        0       0          8     2794336        0        0
  ...        0       0          8     2810792        0        0
  ...        0       0          8     2827184        0        0
  ...        0       0         15     2843576        0        0
  ...        0       0          8     2859968        2        7
  ...        0       0          8     2909208        3        0
```

この例は、sp_ps プロシージャからの isql 出力を示しています。なおこのレポートは出力の都合上、途中を省略しています。

使用法

- sp_ps は、指定したサーバ・スレッドまたはすべての現在の Open Server スレッドのステータスを詳細にレポートします。この情報は、アプリケーション開発中のデバッグ作業に役立ちます。
- *loginame* および *spid* は文字列パラメータです。Adaptive Server Enterprise からのリモート・プロシージャ・コールとして isql を使用し、sp_ps を実行するときは、構文エラーを避けるために、*spid* を一重引用符 (') で囲みます。
- *loginame* または *spid* を指定しない場合、sp_ps はすべての現在のスレッドをリストします。

- 表 4-1 は、sp_ps が返す情報を示します。

表 *4-1:* 返される情報 *(sp_ps)*

情報の種類	意味
spid	スレッド内部のスレッド番号。
Login Name	ログインしたユーザの名前。クライアント・スレッドにのみ適用される。
Host Name	クライアント・タスクでは、これがクライアント・マシン名となる。サイト・ハンドラとサーバ間の RPC 接続では、これがリモート Adaptive Server Enterprise 名となる。
Program Name	クライアント・アプリケーション・プログラムの名前。
Task Type	スレッドのタイプ。カラムの有効値は、NETWORK, CLIENT、SERVER、SITE HANDLER, CHILD、SERVICE、UNKNOWN。
Status	スレッドの現在のステータス。カラムの有効値は、running、runnable、sleeping、sick、free、stopped、spawned、terminal、unknown。1 つしかない “running” タスクは、sp_ps を実行しているスレッド。
Sleep Event	スリープ・スレッドを実行可能にするイベント。
Sleep Label	スリープ・イベントを記述する文字列ラベル。
Current Command	スレッドのステータスを記述する文字列。カラムの内容は srv_thread_props ルーチンで設定される。
Blocked By	(現在は使用されていない)
Run Ticks	(現在は使用されていない)
Current Priority	スレッド実行の優先順位。
Stack Origin	スレッドのスタックが始まるメモリのアドレス。
Net Writes	スレッド開始以後のネットワークの書き込み数。この数字は、サイト・ハンドラとクライアント・スレッドにだけ適用する。
Net Reads	スレッド開始以後のネットワークの読み込み数。この数字は、サイト・ハンドラとクライアント・スレッドにだけ適用する。

表 4-2 は、各カラムにローの形式で返される結果を示しています。

表 4-2: 返される情報のフォーマット *(sp_ps)*

カラム名	データ型	データ長
spid	CS_INT_TYPE	4
Login Name	CS_CHAR_TYPE	SRV_MAXNAME
Host Name	CS_CHAR_TYPE	SRV_MAXNAME
Program Name	CS_CHAR_TYPE	SRV_MAXNAME
Task Type	CS_CHAR_TYPE	SRV_MAXNAME
Status	CS_CHAR_TYPE	SRV_MAXNAME
Sleep Event	CS_INT_TYPE	4
Sleep Label	CS_CHAR_TYPE	SRV_MAXNAME
Current Command	CS_CHAR_TYPE	SRV_MAXNAME
Blocked By	CS_INT_TYPE	4
Run Ticks	CS_INT_TYPE	4
Current Priority	CS_INT_TYPE	4
Stack Origin	CS_INT_TYPE	4
Net Writes	CS_INT_TYPE	4
Net Reads	CS_INT_TYPE	4

参照

sp_terminate、sp_who

sp_regcreate

説明

Open Server 上にレジスタード・プロシージャを作成します。

構文

sp_regcreate proc_name, parm1, parm2, ...

パラメータ

proc_name

作成するレジスタード・プロシージャ名を指定します。

parm1, parm2, ...

(オプション) クライアント・アプリケーションが追加パラメータを渡す場合、ここには新しいプロシージャのパラメータの名前、データ型、デフォルト値を指定します。

例

```
Client-Library クライアントから sp_regcreate を呼び出す
```

この例では、次のパラメータを必要とするレジスタード・プロシージャ np_test を作成します。

- 最初のパラメータは *@p1* であり、CS_INT データ型です。デフォルト値はありません (つまり、この値はデフォルトで NULL に設定されます)。
- 2 番目のパラメータは *@p2* であり、CS_CHAR データ型です。デフォルト値は “No value given” です。

- 3 番目のパラメータは *@p3* であり、CS_INT データ型です。デフォルト値は 0 (ゼロ) です。

この例では、プロシージャを作成する関数 np_create と RPC コマンドの結果を処理する関数 rpc_results のコードが指定されています。この例では (rpc_results を使用して呼び出される) 関数 ex_fetch_data は示されていません。この関数は、Client-Library のサンプル・プログラム内のファイル *exutils.c* で定義されています。

```
/*
 ** np_create() -- Example function to create a notification
 **   procedure on an Open Server.
 **
 ** Parameters:
 **   cmd - Command handle for sending commands.
 **
 ** Returns:
 **   CS_SUCCEED - The notification procedure was successfully
 **      created.
 **   CS_FAIL - Couldn't do it.This routine fails if the
 **      registered procedure already exists.
 */
 CS_RETCODE np_create(cmd)
 CS_COMMAND *cmd;
 {
   CS_DATAFMT datafmt;
   CS_INT     intval;
   CS_CHAR    charbuf[512];
   CS_BOOL    ok = CS_TRUE;
  /*
   ** Build up an RPC command to create the notification
   ** procedure np_test, defined as follows:
   **   np_test @p1 = <integer value>,
   **           @p2 = <character value>,
   **           @p3 = <integer value>
   */
  if (ok
       && (ct_command(cmd, CS_RPC_CMD,
           "sp_regcreate", CS_NULLTERM,
           CS_UNUSED) != CS_SUCCEED))
     ok = CS_FALSE;
  /*
   ** Name of the created procedure will be 'np_test'.
   */
   strcpy(datafmt.name, "proc_name");
   datafmt.namelen = strlen(datafmt.name);
   datafmt.datatype = CS_CHAR_TYPE;
   datafmt.status = CS_INPUTVALUE;
   datafmt.maxlength = 255;
   strcpy(charbuf, "np_test");
   if (ok &&
```

```
        ct_param(cmd, &datafmt,
                (CS_VOID *)charbuf, strlen(charbuf), 0)
        != CS_SUCCEED)
    {
      fprintf(stdout, "np_create: ct_param() @proc_name failed¥n");
      ok = CS_FALSE;
    }
   /*
    ** First parameter is named '@p1', is integer type, and has
    ** no default (i.e., defaults to NULL).We pass -1 as the
    ** indicator to ct_param() to specify a NULL value.
    */
    strcpy(datafmt.name, "@p1");
    datafmt.namelen = strlen(datafmt.name);
    datafmt.datatype = CS_INT_TYPE;
    datafmt.status = CS_INPUTVALUE;
    datafmt.maxlength = CS_UNUSED;
    if (ok &&
        ct_param(cmd, &datafmt, (CS_VOID *)NULL, CS_UNUSED, -1)
        != CS_SUCCEED)
    {
      fprintf(stdout, "np_create: ct_param() @p1 failed¥n");
      ok = CS_FALSE;
    }
   /*
    ** Second parameter is named '@p2', is character type, and has
    ** default “No value given”.
    */
    strcpy(datafmt.name, "@p2");
    datafmt.namelen = strlen(datafmt.name);
    datafmt.datatype = CS_CHAR_TYPE;
    datafmt.status = CS_INPUTVALUE;
    datafmt.maxlength = 255;
    strcpy(charbuf, "No value given");
    if (ok &&
        ct_param(cmd, &datafmt,
                (CS_VOID *)&charbuf, strlen(charbuf), 0)
        != CS_SUCCEED)
    {
      fprintf(stdout, "np_create: ct_param() @p2 failed¥n");
      ok = CS_FALSE;
    }
  /*
    ** Third parameter is named '@p3', is integer type, and
    ** has default 0 (zero).
    */
    strcpy(datafmt.name, "@p3");
    datafmt.namelen = strlen(datafmt.name);
    datafmt.datatype = CS_INT_TYPE;
    datafmt.status = CS_INPUTVALUE;
```

```
    datafmt.maxlength = CS_UNUSED;
    intval = 0;
    if (ok &&
        ct_param(cmd, &datafmt, (CS_VOID *)&intval, CS_UNUSED, 0)
        != CS_SUCCEED)
    {
      fprintf(stdout, "np_create: ct_param() @p3 failed¥n");
      ok = CS_FALSE;
    }

   /*
    ** Send the RPC command.
    */
    if (ok && ct_send(cmd) != CS_SUCCEED)
      ok = CS_FALSE;

   /*
    ** Process the results from the RPC execution.
    */
    if (ok && rpc_results(cmd, CS_FALSE) != CS_SUCCEED)
      ok = CS_FALSE;

  return (ok ? CS_SUCCEED : CS_FAIL);

  } /* np_create */

 /*
  ** rpc_results() -- Process results from an rpc.
  **
  ** Parameters
  **    cmd -- The command handle with results pending.
  **    expect_fetchable -- CS_TRUE means fetchable results
  **           are expected.They will be printed w/ the
  **           ex_fetch_data() routine (defined in file exutils.c).
  **           CS_FALSE means fetchable results cause this routine
  **           to fail.
  **
  ** Returns
  **   CS_SUCCEED -- no errors.

  **   CS_FAIL -- ct_results failed, returned a result_type value
  **      of CS_CMD_FAIL, or returned unexpected fetchable results.
  */
 CS_RETCODE rpc_results(cmd, expect_fetchable)
  CS_COMMAND *cmd;
  CS_BOOL    expect_fetchable;
  {
    CS_RETCODE results_ret;
    CS_INT     result_type;
    CS_BOOL    ok = CS_TRUE;
    CS_BOOL    cmd_failed = CS_FALSE;
```

```
while (ok &&
        (results_ret
         = ct_results(cmd, &result_type))
         == CS_SUCCEED)
 {
   switch((int)result_type)
   {
     case CS_STATUS_RESULT:
     case CS_ROW_RESULT:
     case CS_COMPUTE_RESULT:
     case CS_PARAM_RESULT:
        /*
         ** These cases indicate fetchable results.
         */
         if (expect_fetchable)
         {
           /* ex_fetch_data() is defined in exutils.c */
           ok = (ex_fetch_data(cmd) == CS_SUCCEED);
         }
         else
         {
           (CS_VOID)fprintf(stdout,
                            "RPC returned unexpected result¥n");
           (CS_VOID)ct_cancel(NULL, cmd, CS_CANCEL_ALL);
           ok = CS_FALSE;
         }
         break;
    case CS_CMD_SUCCEED:
     case CS_CMD_DONE:
        /* No action required */
        break;

    case CS_CMD_FAIL:
        (CS_VOID)fprintf(stdout,
          "RPC command failed on server.¥n");
        cmd_failed = CS_TRUE;
        break;
    default:
        /*
        ** Unexpected result type.
        */
        (CS_VOID)fprintf(stdout,
                         "RPC returned unexpected result¥n");
        (CS_VOID)ct_cancel(NULL, cmd, CS_CANCEL_ALL);
        ok = CS_FALSE;
        break;
   } /* switch */
 } /* while */
```

```
 switch((int) results_ret)
  {
    case CS_END_RESULTS:
    case CS_CANCELED:
      break;
    case CS_FAIL:
    default:
      ok = 0;
  }
 return ((ok && !cmd_failed) ? CS_SUCCEED : CS_FAIL);

} /* rpc_results() */
```

DB-Library クライアントから sp_regcreate を呼び出す

この例では、パラメータを 2 つ必要とする pricechange という名前のレジスタード・プロシージャを作成します。最初のパラメータは *@current_price* で、SYBMONEY データ型で表現されています。2 つ目のパラメータは *@sequence_num* で、SYBINT4 データ型です。どちらのパラメータにもデフォルト値はありません。

```
dbnpdefine(dbproc, "pricechange", DBNULLTERM);
 dbregparam(dbproc, "@current_price", DBNULLTERM,
            SYBMONEY, DBNODEFAULT, NULL);
 dbregparam(dbproc, "@sequence_num", DBNULLTERM,
            SYBINT4, DBNODEFAULT, NULL);
 status = dbnpcreate(dbproc);

 if (status == FAIL)
 {
 fprintf(stderr,
     "Could not create pricechange procedure.¥n");
 }
```

表 4-3 は、SRV_C_PROCEXEC コールバック・ハンドラが、pricechange プロシージャが登録中であることを調べるのに使用する呼び出しについてまとめたものです。

表 *4-3:* 戻り値 *(sp_regcreate)*

関数呼び出し	返されるデータ
srv_procname(srvproc, (int *) NULL)	“sp_regcreate”
srv_rpcparams(srvproc)	3
srv_paramdata(srvproc, 1)	“pricechange”
srv_paramdata(srvproc, 2)	“@current_price”
srv_paramdata(srvproc, 3)	“@sequence_num”

使用法

- クライアント・アプリケーションはリモートで sp_regcreate を呼び出して、レジスタード・プロシージャを作成します。
- クライアント・アプリケーションが作成するレジスタード・プロシージャは、「ノーティフィケーション (通知) プロシージャ」と呼ばれます。これらのプロシージャにはアプリケーションが定義するコードを指定できません。また、これらのプロシージャは、レジスタード・プロシージャ・ノーティフィケーションに依存するクライアント・アプリケーションに有効です。
- sp_regcreate の最初のパラメータ (*proc_name*) は、作成するプロシージャの名前です。新しいレジスタード・プロシージャがパラメータを必要とする場合、そのパラメータは追加パラメータを渡すことによって定義されます。新しいプロシージャの最初のパラメータは sp_regcreate の 2 番目のパラメータとして、2 番目のパラメータは sp_regcreate の 3 番目のパラメータとして、というように渡されます。
- Client-Library を使用して構築されたクライアント・アプリケーションは、sp_regcreate を呼び出す RPC コマンドを送信して、レジスタード・プロシージャを作成できます。

  この例は、「Client-Library クライアントから sp_regcreate を呼び出す」に示してあります。
- DB-Library プログラムは dbnpdefine、dbregparam、および dbnpcreate を使用して、レジスタード・プロシージャを作成します。dbnpdefine は内部的に RPC コマンドを生成して、リモートで sp_regcreate を呼び出します。dbnpcreate は RPC を送信して、その結果を処理します。

  この例は、「DB-Library クライアントから sp_regcreate を呼び出す」に示してあります。
- Server-Library プログラムは srv_regdefine、srv_regparam、および srv_regcreate を使用して、レジスタード・プロシージャを作成できます。

メッセージ

sp_regcreate は、次のメッセージを返すことができます。

番号	重大度	意味
16505	0	プロシージャが正しく登録された。
16506	11	プロシージャはすでに登録されている。
16507	11	プロシージャを登録できない。

参照

sp_regdrop、sp_regnowatch、sp_regwatch、srv_regdefine、srv_regexec、srv_reginit、srv_regparam

sp_regdrop

説明

レジスタード・プロシージャ・リストからプロシージャを削除します。

構文

sp_regdrop *proc_name*

パラメータ

proc_name
削除するレジスタード・プロシージャの名前です。

例

```
1>execute stock...sp_regdrop pricechange
 2>go
```

この例では、isql で Adaptive Server Enterprise にログインしたクライアントは、サーバ間のリモート・プロシージャ・コールを使って Open Server stock で sp_regdrop を実行します。このプロシージャは、stock から pricechange レジスタード・プロシージャを削除します。

```
dbrpcinit(dbproc, "sp_regdrop", NULL);
 dbrpcparam(dbproc, "proc_name", NULL, SYBCHAR, -1,
 11, "pricechange");
 dbrpcsend(dbproc);
```

この例では、DB-Library RPC のルーチンを使用し、単一のパラメータ “pricechange” に対して sp_regdrop を実行しています。これにより、sp_regdrop システム・プロシージャは、Open Server から pricechange レジスタード・プロシージャを削除します。

使用法

- プロシージャの登録が解除されると、未処理の通知 (ノーティフィケーション) 要求があるクライアントは、そのプロシージャはすでに登録されていないというメッセージを受け取ります。
- sp_regdrop は、クライアントが dbnpdrop を実行した時に実行されます。SRV_C_PROCEXEC コールバック・ハンドラは、sp_regdrop が実行中であるかどうかを調べるのに srv_rpcname を使用できます。次に srv_bind および srv_xferdata を使用し、最初のパラメータである *proc_name* のポインタを入手できます。

メッセージ

proc_name has been unregistered.

proc_name パラメータで指定されたプロシージャは登録を解除されました。

proc_name is not a registered procedure.

proc_name パラメータで指定されたプロシージャは、Open Server で登録されたものではありません。

Unable to unregister *proc_name*.

Open Server は、何らかの理由でプロシージャを登録解除することができませんでした。

参照

sp_regdrop、srv_regexec、srv_reginit、srv_regparam

sp_reglist

説明

Open Server 上のすべてのレジスタード・プロシージャをリストします。

構文

sp_reglist

例

```
1>execute utility...sp_reglist
 2>go

 Procedure Name
  ------------------------------
  sp_who



  sp_regwatch
  sp_ps
  sp_regdrop
  sp_reglist
  sp_regwatchlist
  sp_regcreate
  sp_regnowatch

 (0 rows affected)
```

この isql の例では、現時点でのレジスタード・プロシージャをすべてリストしています。

使用法

- sp_reglist は、現在 Open Server に登録されているプロシージャの名前をすべてロー・データとして返します。
- C 言語プログラムでも、sp_reglist を使ってレジスタード・プロシージャをリストできます。

結果は、SRV_MAXNAME 文字のデータ長を持つ、1 つの char カラムを含んだローとして返されます。

参照

sp_regcreate、sp_regdrop、sp_regwatch、sp_regwatchlist

sp_regnowatch

説明

レジスタード・プロシージャの通知 (ノーティフィケーション) リストからクライアントを削除します。

構文

sp_regnowatch proc_name

パラメータ

proc_name

レジスタード・プロシージャの名前です。

例

```
dbrpcinit(dbproc, "sp_regnowatch", (DBUSMALLINT)
            0);
 dbrpcparam(dbproc, "@proc_name", 0, SYBCHAR, 15,
            15, "pricechange");
 dbrpcsend(dbproc);
```

この例は、pricechange レジスタード・プロシージャの通知リストからクライアントを削除します。

使用法

- クライアントが dbregnowatch を呼び出すと、このレジスタード・プロシージャは実行します。
- SRV_C_PROCEXEC コールバック・ハンドラは、srv_rpcname を使って sp_regnowatch の実行を確認し、sp_paramdata を使って通知要求を削除する対象のプロシージャ名を入手します。

メッセージ

```
Notification request removed.
```

通知要求は正常に削除されました。

```
proc_name is not a registered procedure.
```

proc_name に指定されたプロシージャは、Open Server には登録されていません。

```
No requests pending.
```

クライアントは、プロシージャに対する未処理の通知要求がありませんでした。

```
Unable to remove notification request.
```

Open Server は、通知要求の削除に失敗しました。

参照

sp_regcreate、sp_regdrop、sp_regwatch、sp_regnowatch、sp_regwatch

sp_regwatch

説明

レジスタード・プロシージャの通知リストにクライアントを追加します。

構文

sp_regwatch *proc_name* [options]

パラメータ

proc_name

クライアントが通知 (ノーティフィケーション) を希望するレジスタード・プロシージャの名前です。

options

クライアントへの通知を、一度しか行わないか、またはプロシージャを実行するごとに行うか、および通知が同期かまたは非同期かを指定する CS_SMALLINT です。次の表 4-4 では、*options* に設定できる値を示します。これらの値はビット・フラグなので、一度に複数を設定できます。

表 *4-4: sp_regwatch の options パラメータの値*

option の値	機能
CS_NOTIFY_NOWAIT	非同期の通知を示す。
CS_NOTIFY_WAIT	同期の通知を示す。
SRV_NOTIFY_ALWAYS	Open Server はクライアントが切断するか、srv_regnowatch か dbregnowatch を呼び出すまでプロシージャを実行するごとにクライアントに通知する。これはデフォルト値です。
SRV_NOTIFY_ONCE	Open Server は通知を送信した後で、通知リストからクライアントを削除する。

例

```
dbrpcinit(dbproc, "sp_regwatch", (DBUSMALLINT) 0);
 dbrpcparam(dbproc, "@proc_name", 0, SYBCHAR,
                15, 15, "pricechange");
 dbrpcsend(dbproc);
```

この例では、pricechange というプロシージャの通知リストにクライアントを追加します。プロシージャが実行されるたびに、クライアントは通知を受けます。

```
optionval = SRV_NOTIFY_ONCE;
 dbrpcinit(dbproc, sp_regwatch, (DBUSMALLINT)
           DBWAIT);
 dbrpcparam(dbproc, "@proc_name", 0, SYBCHAR,
            15, 15, pricechange");
 dbrpcparam(dbproc, "@options", 0, SYBINT4, -1,
            -1, &optionval);
 dbrpcsend(dbproc);
```

この例では、pricechange というプロシージャの通知リストにクライアントを追加します。クライアントは、一度だけプロシージャが実行したという通知を受けています。

使用法

- Open Server は、クライアントが dbnpwatch を呼び出すと、sp_regwatch を内部的に実行します。
- クライアントが通知を待っている間にプロシージャが削除された場合、クライアントは、プロシージャはすでに登録されていないことを示すエラー・メッセージを受けます。

メッセージ

```
Notification request added.
```

通知要求が正常に追加されました。

```
proc_name is not a registered procedure.
```

proc_name パラメータで指定されたプロシージャは、Open Server では登録されていません。

```
Unable to add notification request.
```

Open Server は何らかの理由で要求を追加できませんでした。

参照

sp_regcreate、sp_regnowatch、sp_regdrop

sp_regwatchlist

説明

クライアントが通知を要求したレジスタード・プロシージャをリストします。

構文

sp_regwatchlist

例

```
1>execute utility...sp_regwatchlist
 2>go

 Procedure Name
  ------------------------------
  pricechange
```

このサーバ間 RPC の isql の例は、クライアントが pricechange レジスタード・プロシージャの通知を要求したことを示しています。

使用法

- Open Server は、クライアントが dbregwatchlist を呼び出すと、sp_regwatchlist を内部的に実行します。
- SRV_C_PROCEXEC コールバック・ハンドラは、srv_rpcname を呼び出し、sp_regwatchlist が実行しているのを確認することができます。

結果は、SRV_MAXNAME 文字の 1 つの char カラムを含むローとして返されます。

参照

sp_reglist、sp_regwatchlist

sp_serverinfo

説明

クライアントに文字セットまたはソート順の情報を送ります。

構文

sp_serverinfo　function [name]

パラメータ

function

表 4-5 は function の有効値を示します。

表 *4-5: function* の値 *(sp_serverinfo)*

値	意味
server_csname	Open Server アプリケーションの文字セットの名前は、単一カラムの 1 つの文字ローとしてクライアントに送信される。
server_soname	Open Server アプリケーションのソート順の名前は、単一カラムの 1 つの文字ローとしてクライアントに送信される。
csdefinition	文字セット定義を含むローがクライアントへ送信される。ローは、タイプが CS_SMALLINT_TYPE、ID が CS_TINYINT_TYPE、文字セット定義が CS_IMAGE_TYPE の 3 つのカラムで構成されている。
sodefinition	ソート順定義を含むローがクライアントへ送信される。ローは、タイプが CS_SMALLINT_TYPE、ID が CS_TINYINT_TYPE、ソート順定義が CS_IMAGE_TYPE の 3 つのカラムで構成されている。

name

文字セットまたはソート順の名前です。*function* が *csdefinition* または *sodefinition* に設定された場合にのみ、*name* が要求されます。

使用法

- リモート・プロシージャである sp_serverinfo は、標準システム・プロシージャ、たとえば sp_who として自動的に登録され、処理されます。sp_serverinfo が RPC として受信されると、Open Server が自動的に処理します。アプリケーション・コードは必要ありません。
- クライアントが言語要求で sp_serverinfo 要求を送信した場合、正しい応答を送信するには、レジスタード・プロシージャ・ルーチンを使用してこのストアド・プロシージャを実行しなければなりません。
- この情報はローとしてクライアントへ送られます。

sp_terminate

説明

Open Server のスレッドを終了します。

構文

sp_terminate *spid* [, *options*]

パラメータ

spid

スレッド ID。これは sp_who プロシージャまたは srv_thread_props の呼び出しで得られます。

options

スレッドが即時に終了するか、キューイングされた切断イベントによって終了するかを指定します。以前のイベント処理後に発生した切断イベントをキューイングするには、“deferred” を指定してください。これがデフォルトです。スレッドの現在のイベントやキューイングされたイベントを無視してスレッドを即座に終了させるには、“immediate” を指定してください。

例

```
1> execute utility...sp_who
 2> go
```

```
spid  status     loginame      hostname   blk    cmd
----- ---------- ------------ ---------- ----  ----------------
1     runnable                             0    NETWORK HANDLER
2     sleeping                             0    CONNECT HANDLER
3     sleeping                             0    DEFERRED HANDLER
4     runnable                             0    SCHEDULER
12    runnable    ned           sonoma     0    PRINT TASK
24    running     bud           sonoma     0

(0 rows affected)
```

この例は、isql を使って、異常を起こしたサーバ・スレッドを検索し、終了する方法を示しています。スレッドは即座に終了します。

```
1> execute utility...sp_terminate 12, "immediate"
 2> go
```

```
spid = 12;
dbrpcinit(dbproc, "sp_terminate", (DBUSMALLINT) 0);
dbrpcparam(dbproc, "@spid", 0, SYBINT4, -1,
            -1, &spid);
dbrpcparam(dbproc, "@options", 0, SYBCHAR, 9,
            9, "deferred");
dbrpcsend(dbproc);
```

この DB-Library の例では、スレッドを使用してスレッドの SRV_DISCONNECT イベントをキューイングしています。次にスレッドが実行可能になった時点で、このスレッドが切断イベントを受信し、終了します。

使用法

- sp_who または sp_ps を使用して、終了の対象となるスレッドの *spid* を検索します。
- Server-Library プログラムでは、srv_termproc を使用しスレッドを終了します。

メッセージ

spid terminated. (spid は終了されました)

spid scheduled for termination. (spid は終了するようにスケジューリングされました)

spid not currently in use. (spid は現在使用されていません)

参照

sp_who、srv_termproc

sp_who

説明

指定された Open Server スレッドに関するステータス情報を返します。

構文

sp_who [*loginame* | '*spid*']

パラメータ

loginame

ユーザのログイン名です。

SPID

レポートの対象となるスレッドの内部 ID 番号です。*spid* は、以前の sp_ps または sp_who 呼び出しから得られます。*spid* が指定されていないと、すべてのスレッドがリストされます。

例

```
1>execute utility...sp_who
 2>go
```

```
 spid    status     loginame     hostname   blk cmd
-------- ---------- ------------ ---------- --- ----------------
       1 runnable                           0   NETWORK HANDLER
       2 sleeping                           0   CONNECT HANDLER
       3 sleeping                           0   DEFERRED HANDLER
       4 runnable                           0   SCHEDULER
      11 sleeping                hiram      0
      14 running    bud          sonoma     0
```

この例は、sp_who プロシージャからの出力を示しています。

使用法

- sp_who は、指定されたサーバ・スレッド、またはすべての現在の Open Server スレッドについてのステータス情報をレポートします。
- sp_who システム・レジスタード・プロシージャからの出力は、Adaptive Server Enterprise の sp_who システム・プロシージャの出力と一致します。
- sp_who は sp_ps が返す情報のサブセットを返します。

- *loginame* および *spid* は文字列パラメータです。Adaptive Server Enterprise からのリモート・プロシージャ・コールとして isql を使用し、sp_who を実行するときは、構文エラーを避けるために、*spid* を一重引用符 (') で囲みます。
- *loginame* または *spid* を指定しなければ、sp_who はすべての現在のスレッドをリストします。
- sp_who は次の情報を返します。

  spid − スレッドの内部スレッド番号。

  status − スレッドの現在のステータス。このカラムの値は、次のとおりです。

  - running
  - runnable
  - sleeping
  - sick
  - free
  - stopped
  - spawned
  - terminal
  - unknown

  1 つしかない “running” タスクは、sp_who を実行しているスレッドです。

  loginame − ログインしたユーザの名前です。クライアント・スレッドにのみ適用されます。

  hostname − クライアント・タスクの場合には、クライアント・マシンの名前です。サイト・ハンドラ・スレッドの場合には、リモート Adaptive Server Enterprise の名前です。

  blk − このフィールドは使用されず、常に 0 に設定されています。

  cmd − スレッドのステータスを記述する文字列です。srv_thread_props ルーチンでこのカラムの内容を設定します。

表 4-6 は、各カラムにローの形式で返される結果を示しています。

表 *4-6:* 返される情報のフォーマット *(sp_who)*

カラム名	データ型	データ長
spid	CS_INT_TYPE	4
status	CS_CHAR_TYPE	10
loginame	CS_CHAR_TYPE	12
hostname	CS_CHAR_TYPE	10
blk	CS_INT_TYPE	3
cmd	CS_CHAR_TYPE	16

参照

sp_ps, sp_terminate

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。

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。

Client-Library

Open Client の一部で、クライアント・アプリケーションを記述するためのルーチンの集まり。Client-Library は、Sybase 製品ラインのカーソルや他の高度な機能を取り込むように設計されたライブラリです。

DB-Library

Open Client の一部で、クライアント・アプリケーションを記述するためのルーチンの集まり。

FIPS

Federal Information Processing Standards (連邦情報処理標準) の略。FIPS フラグが有効なとき、Adaptive Server Enterprise および Embedded SQL プリコンパイラは、標準でない拡張された SQL 文を検出すると警告を発行します。

interfaces ファイル (interfaces file)

サーバ名をトランスポート・アドレスにマップするファイル。クライアント・アプリケーションが、サーバに接続するために ct_connect または dbopen を呼び出すと、Client-Library または DB-Library が interfaces ファイルからサーバのアドレスを検索します。ただし、すべてのプラットフォームが interfaces ファイルを使うわけではありません。interfaces ファイルを使わないプラットフォームでは、別のメカニズムでクライアントにサーバ・アドレスを知らせます。

isql スクリプト・ファイル (isql script file)

Embedded SQL において、プリコンパイラが生成できる 3 つのファイルのうちの 1 つ。isql スクリプト・ファイルには、プリコンパイラが生成したストアド・プロシージャが含まれます。ストアド・プロシージャは、Transact-SQL で記述されます。

mutex

相互排他セマフォ。Open Server アプリケーションが、共有オブジェクトへ排他アクセスをするために使用する論理オブジェクトのことです。

NULL

NULL は、明示的に割り当てられた値を持ちません。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。

Open Server

カスタム・サーバを作成するためのツールとインタフェースを提供する Sybase 製品。

Open Server アプリケーション (Open Server application)

Open Server で構築されたカスタム・サーバ。

Server-Library

Open Server アプリケーションの記述に使用するルーチンの集まり。

SQLCA

1. Embedded SQL アプリケーションにおいて、Adaptive Server Enterprise とアプリケーション・プログラム間の通信パスを提供する構造体。Adaptive Server Enterprise は、各 SQL 文を実行したあと、SQLCA 内のリターン・コードを格納します。

2. Client-Library アプリケーションにおいて、アプリケーションが、Client-Library およびサーバのエラー・メッセージと情報メッセージを取得するのに使用する構造体。

sqlcode

1. Embedded SQL アプリケーションにおいて、Adaptive Server Enterprise とアプリケーション・プログラム間の通信パスを提供する構造体。Adaptive Server Enterprise は、各 SQL 文を実行したあと、SQLCODE 内のリターン・コードを格納します。SQLCODE は、独立して存在することも、SQLCA 構造体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、アプリケーションが Client-Library およびサーバのエラー・メッセージと情報メッセージ・コードを取得するのに使用する構造体。

TDS

Sybase のクライアントとサーバが通信に使用するアプリケーション・レベルのプロトコル。TDS は、コマンドと結果を記述します。

Transact-SQL

データベース言語 SQL の機能拡張バージョン。アプリケーションは、Transact-SQL を使用して、Adaptive Server Enterprise と通信できます。

イベント・ハンドラ (event handler)

Open Server におけるイベントを処理するルーチン。Open Server アプリケーションは、Open Server が提供するデフォルト・ハンドラを使用することができます。また、カスタマイズしたイベント・ハンドラをインストールすることもできます。

イベント (event)

Open Server アプリケーションに何らかの動作を行うよう要求するオカレンス。クライアント・コマンドおよび Open Server アプリケーション・コードの特定のコマンドは、イベントをトリガできます。イベントが発生すると、Open Server は、サーバ・アプリケーション・コード内の適切なイベント処理ルーチン、または適切なデフォルト・イベント・ハンドラのどちらかを呼び出します。

インジケータ変数 (indicator variable)

他の変数の値やフェッチしたデータについての特別な条件を示す変数。

Embedded SQL ホスト変数とともに使用すると、データベースの値が NULL である箇所を示します。

エラー・メッセージ (error message)

Open Client/Server 製品がエラー状態を検出したときに発行するメッセージ。

カーソル (cursor)

SQL 文に関連付けられた記号名。

Embedded SQL において、カーソルは複数ローのデータをホスト・プログラムに渡すデータ・セレクタです。このローの受け渡しは、一度に 1 つずつ行われます。

隠し構造体 (hidden structure)

内部が Open Client/Server プログラマに対して隠されている構造体。Open Client/Server プログラマは、隠し構造体の割り付け、操作、割り付け解除を行うために、Open Client/Server ルーチンを使用しなければなりません。隠し構造体には、CS_CONTEXT 構造体などがあります。

拡張トランザクション (extended transaction)

Embedded SQL における、複数の Embedded SQL 文からなるトランザクション。

キーワード (keyword)

Transact-SQL または Embedded SQL で排他的に利用するように予約されているワードまたはフレーズ。予約語とも呼ばれます。

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。

機能 (capabilities)

クライアント／サーバ接続の機能が、その接続について許可されるクライアント要求とサーバ応答の種類を決定します。

クエリ (query)

1. データの検索要求。通常は select 文です。

2. データを操作する任意の SQL 文。

クライアント (client)

クライアント／サーバ・システムにおいて、サーバへ要求を送り、この要求に対する結果に対して処理を行う部分。

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。

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。

現在のロー (current row)

カーソルに関連して、カーソルが置かれているロー。フェッチは、カーソルに対して現在のローを取得します。

コード・セット (code set)

「**文字セット** (character set)」を参照してください。

コールバック・イベント (callback event)

Open Client と Open Server におけるコールバック・ルーチンをトリガするイベント。

コールバック・ルーチン (callback routine)

コールバック・イベントと呼ばれるトリガ・イベントに応答して、Open Client または Open Server が呼び出すルーチン。

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。

コマンド構造体 (command structure)
(CS_COMMAND) Client-Library アプリケーションがコマンドの送信と結果の処理に使用する Client-Library の隠し構造体。

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。

コンテキスト構造体 (context structure)
(CS_CONTEXT) 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 routines 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。

サーバ (server)
クライアント／サーバ・システムにおけるクライアント要求を処理し、結果をクライアントに返す部分。

システム・プロシージャ (system procedure)
Adaptive Server Enterprise が、システム管理のために提供するストアド・プロシージャ。これらのプロシージャは、システム・テーブルからの情報の取得を簡単にし、データベース管理とシステム・テーブルの更新などを可能にします。

システム・レジスタード・プロシージャ (system registered procedures)
Open Server が、レジスタード・プロシージャのノーティフィケーション (通知) の管理とステータスの監視のために提供する内部レジスタード・プロシージャ。

システム管理者 (system administrator)
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。

出力変数 (output variable)
Embedded SQL において、ストアド・プロシージャからアプリケーション・プログラムにデータを渡す変数。

照合順 (collating sequence)
「**ソート順** (sort order)」を参照してください。

ス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可能にします。「**カーソル** (cursor)」も参照してください。

ステータス変数 (status variable)
Embedded SQL において、ストアド・プロシージャのリターン・ステータス値を受け取ることによって、プロシージャの成功または失敗を示す変数。

ストアド・プロシージャ
Adaptive Server Enterprise における、名前を付けて保管された SQL 文とオプションのフロー制御文の集まり。Adaptive Server Enterprise が提供するストアド・プロシージャは、「**システム・プロシージャ**」と呼ばれます。

スレッド (thread)
Open Server アプリケーションからライブラリ・コードまでの実行のパス。また、スタック領域、ステータス情報およびイベント・ハンドラに対応するパス。

接続構造体 (connection structure)
(CS_CONNECTION) コンテキスト内にクライアント／サーバ接続を定義する Client-Library の隠し構造体。

ソート順 (sort order)
文字データをソートするときの順序の決定に使用されます。「照合順」とも呼ばれます。

ターゲット・ファイル (target file)
Embedded SQL において、プリコンパイラが生成できる 3 つのファイル。ターゲット・ファイルは元の入力ファイルと似ていますが、すべての SQL 文が Client-Library の関数呼び出しに変換されています。

データ型 (datatype)
変数に有効な値と演算を表す定義属性。

データベース (database)
特別な目的のために組織化された、関連するデータ・テーブルとその他のデータベース・オブジェクトの集まり。

「**スクロール可能カーソル**」も参照してください。

デッドロック (deadlock)
データにロックを保持している 2 人のユーザが、互いに他方のデータのロックを獲得しようとしたときに起こる状態。Adaptive Server Enterprise がデッドロックを検出すると、片方のユーザのプロセスを強制終了することによってこの状態を解決します。

デフォルト・データベース (default database)
ユーザがデータベース・サーバにログインしたとき、デフォルトで指定されるデータベース。

デフォルト言語 (default language)
1. アプリケーションに対して明示的にローカライゼーションの指定を行わないとき、Open Client/Server 製品が使用する言語。デフォルト言語は、ロケール・ファイルの “default” エントリにより決定されます。

2. ユーザが明示的に言語を選択しなかったとき、Adaptive Server Enterprise がメッセージとプロンプトに使用する言語。

デフォルト (default)
明示的に何も指定されなかったときに、Open Client/Server 製品が使用する値、オプション、または動作。

トランザクション・モード (transaction mode)
Adaptive Server Enterprise がトランザクションを管理する方法。Adaptive Server Enterprise は、2 つのトランザクション・モードをサポートしています。Transact-SQL モード (「非連鎖トランザクション」とも呼ばれる) と ANSI モード (「連鎖トランザクション」とも呼ばれる) です。

トランザクション (transaction)
バックアップまたはリカバリのために 1 つの単位として扱われる、1 つ以上のサーバ・コマンド。トランザクション内のコマンドは、1 つのグループとしてコミットされます。したがって、すべてのコマンドがコミットされるか、すべてロールバックされるかのどちらかとなります。

動的 SQL (dynamic SQL)
動的 SQL によって、Embedded SQL または Client-Library アプリケーションは、実行時に値の決まる変数を含む SQL 文を実行できます。

入力変数 (input variable) 情報をルーチン、ストアド・プロシージャ、または Adaptive Server Enterprise に渡すときに使う変数。

配列 (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。

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。

バルク・コピー (bulk copy) データベースからデータをコピーしたり、データベースへデータをコピーしたりするコピー・ユーティリティ。bcp とも呼ばれます。

パススルー・モード (passthrough mode) ゲートウェイ・アプリケーションに関する 1 つの状態。

パススルー・モードのとき、ゲートウェイは、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します。

パラメータ (parameter) 1. データをルーチンに渡すとき、およびルーチンからデータを取得するときに使用する変数。

2. ストアド・プロシージャへの引数。

ブラウズ・モード (browse mode) DB-Library と Client-Library アプリケーションが、一度に 1 つのローの値を更新しながらデータベース・ローをブラウズする方法。同様の機能を果たすカーソルの方が、一般に扱いやすくなっています。

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。

プロパティ (property) 構造体に格納される名前付きの値。コンテキスト構造体、接続構造体、スレッド構造体、コマンド構造体は、プロパティを持ちます。構造体のプロパティは、構造体の動作を決めます。

変換 (conversion) 「文字セットの変換」を参照してください。

ホスト・プログラム (host program) Embedded SQL における、Embedded SQL コードを含むアプリケーション・プログラム。

ホスト言語 (host language) アプリケーションを記述するときに使われるプログラミング言語。

ホスト変数 (host variable)

Embedded SQL における、Adaptive Server Enterprise とアプリケーション・プログラム間のデータ転送を可能にする変数。「インジケータ変数」、「入力変数」、「出力変数」、「結果変数」、「ステータス変数」 を参照してください。

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。

メッセージ・キュー (message queue)

Open Server において、スレッドが通信するとき使用するメッセージ・ポインタのリンク・リスト。スレッドは、キューにメッセージを書き込んだり、キューからメッセージを読み込んだりすることができます。

メッセージ番号 (message number)

エラー・メッセージをユニークに識別する番号。

文字セット (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)

サーバへ入出力するときの文字セットのコード化スキームの変換。サーバとクライアントが異なる文字セットを使って通信するとき、変換が行われます。たとえば、Adaptive Server Enterprise が ISO 8859-1 を使用し、クライアントが CodePage 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。

ユーザ名

「**ログイン名** (login name)」 を参照してください。

リスティング・ファイル (listing file)

Embedded SQL において、プリコンパイラが生成できる 3 つのファイルのうちの 1 つ。リスティング・ファイルには、入力ファイルのソース文と、情報、警告、エラーなどのメッセージが含まれます。

リモート・プロシージャ・コール (remote procedure call)

1. クライアント・アプリケーションが Adaptive Server Enterprise ストアド・プロシージャを実行する 2 つの方法のうちの 1 つ (もう 1 つの方法では、Transact-SQL の execute 文を使用します)。Client-Library のアプリケーションは、ct_command を呼び出すことによって、リモート・プロシージャ・コール・コマンドを開始します。DB-Library アプリケーションは、dbrpcinit を呼び出すことによって、リモート・プロシージャ・コール・コマンドを開始します。

2. クライアントが Open Server アプリケーションを使って利用できる要求のタイプの 1 つ。これに応答して Open Server は、対応するレジスタード・プロシージャを実行するか、または Open Server アプリケーションの RPC イベント・ハンドラを呼び出します。

3. ユーザが接続しているサーバとは異なるサーバ上で実行される「**ストアド・プロシージャ**」。

レジスタード・プロシージャ (registered procedure)

Open Server において、名前を付けて保管される C で記述された文の集まり。Open Server が提供するレジスタード・プロシージャは、「**システム・レジスタード・プロシージャ**」 と呼ばれます。

ローカライゼーション (localization)
アプリケーションを特定の言語環境で使用するために設定する処理のこと。ローカライズされたアプリケーションは、通常、各国の言語と文字セットでメッセージを作成し、その国の日時表記フォーマットを使用します。

ログイン名
ユーザが、サーバにログインするとき使用する名前。Adaptive Server Enterprise のログイン名が有効となるのは、Adaptive Server Enterprise がシステム・テーブル syslogins にそのユーザのエントリを持つ場合です。

ロケール・ファイル (locales file)
ロケール名を言語と文字セットのペアにマッピングするファイル。Open Client/Server 製品は、ローカライゼーション情報をロードするときにこのロケール・ファイルを調べます。

ロケール構造体 (locale structure)
Client-Library または Open Server アプリケーションのためのカスタム・ローカライゼーション値を定義する CS-Library の隠し構造体 (CS_LOCALE)。アプリケーションは、CS_LOCALE を使用して、使用される言語、文字セット、日付順、ソート順を定義できます。ロケール構造体の割り付けと割り付け解除には、CS-Library ルーチン cs_loc_alloc および cs_loc_drop を使用します。

ロケール名 (locale name)
言語と文字セットのペアを表す文字列。ロケール名は、ロケール・ファイルにリストされています。Sybase があらかじめ定義しているロケール名の他に、システム管理者が別のロケール名を定義し、ロケール・ファイルに追加することもできます。

索引

記号

A

B

C

D

E

F

I

L

M

N

O

P

R

S

T

U

W

X

あ

い

う

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

て

と

な

に

ね

の

は

ひ

ふ

ほ

ま

み

め

も

ゆ

よ

り

れ

ろ

わ