SYBASE iAnywhere

Ultra Light
C/C++ プログラミング

2009 年 2 月

バージョン 11.0.1

版権と商標

目次

はじめに

このマニュアルの内容

このマニュアルでは、Ultra Light C および Ultra Light C++ のプログラミング・インタフェースについて説明します。Ultra Light を使用すると、ハンドヘルド、モバイル、埋め込みデバイスに対してデータベース・アプリケーションを開発し、配備できます。

対象読者

このマニュアルは、Ultra Light リレーショナル・データベースのパフォーマンス、リソース効率、堅牢性、セキュリティを利用してデータを格納、同期することを目的とする C および C++ のアプリケーション開発者を対象にしています。

SQL Anywhere のマニュアルについて

SQL Anywhere の完全なマニュアルは 4 つの形式で提供されており、いずれも同じ情報が含まれています。

- **HTML ヘルプ**　オンライン・ヘルプには、SQL Anywhere の完全なマニュアルがあり、SQL Anywhere ツールに関する印刷マニュアルとコンテキスト別のヘルプの両方が含まれています。

  Microsoft Windows オペレーティング・システムを使用している場合は、オンライン・ヘルプは HTML ヘルプ (CHM) 形式で提供されます。マニュアルにアクセスするには、**[スタート] - [プログラム] - [SQL Anywhere 11] - [マニュアル] - [オンライン・マニュアル]** を選択します。

  管理ツールのヘルプ機能でも、同じオンライン・マニュアルが使用されます。

- **Eclipse**　UNIX プラットフォームでは、完全なオンライン・ヘルプは Eclipse 形式で提供されます。マニュアルにアクセスするには、SQL Anywhere 11 インストール環境の *bin32* または *bin64* ディレクトリから *sadoc* を実行します。

- **DocCommentXchange**　DocCommentXchange は、SQL Anywhere マニュアルにアクセスし、マニュアルについて議論するためのコミュニティです。

  DocCommentXchange は次の目的に使用できます (現在のところ、日本語はサポートされておりません)。

  ○ マニュアルを表示する

  ○ マニュアルの項目について明確化するために、ユーザによって追加された内容を確認する

  ○ すべてのユーザのために、今後のリリースでマニュアルを改善するための提案や修正を行う

  http://dcx.sybase.com を参照してください。

- **PDF**　SQL Anywhere の完全なマニュアル・セットは、Portable Document Format (PDF) 形式のファイルとして提供されます。内容を表示するには、PDF リーダが必要です。Adobe Reader をダウンロードするには、http://get.adobe.com/reader/ にアクセスしてください。

  Microsoft Windows オペレーティング・システムで PDF マニュアルにアクセスするには、**[スタート] - [プログラム] - [SQL Anywhere 11] - [マニュアル] - [オンライン・マニュアル - PDF]** を選択します。

  UNIX オペレーティング・システムで PDF マニュアルにアクセスするには、Web ブラウザを使用して *install-dir/documentation/ja/pdf/index.html* を開きます。

マニュアル・セットに含まれる各マニュアルについて

SQL Anywhere のマニュアルは次の構成になっています。

- **『SQL Anywhere 11 - 紹介』**　このマニュアルでは、データの管理および交換機能を提供する包括的なパッケージである SQL Anywhere 11 について説明します。SQL Anywhere を使用す

ると、サーバ環境、デスクトップ環境、モバイル環境、リモート・オフィス環境に適したデータベース・ベースのアプリケーションを迅速に開発できるようになります。

- **『SQL Anywhere 11 - 変更点とアップグレード』** このマニュアルでは、SQL Anywhere 11 とそれ以前のバージョンに含まれる新機能について説明します。
- **『SQL Anywhere サーバ - データベース管理』** このマニュアルでは、SQL Anywhere データベースを実行、管理、構成する方法について説明します。データベース接続、データベース・サーバ、データベース・ファイル、バックアップ・プロシージャ、セキュリティ、高可用性、Replication Server を使用したレプリケーション、管理ユーティリティとオプションについて説明します。
- **『SQL Anywhere サーバ - プログラミング』** このマニュアルでは、C、C++、Java、PHP、Perl、Python、および Visual Basic や Visual C# などの .NET プログラミング言語を使用してデータベース・アプリケーションを構築、配備する方法について説明します。ADO.NET や ODBC などのさまざまなプログラミング・インタフェースについても説明します。
- **『SQL Anywhere サーバ - SQL リファレンス』** このマニュアルでは、システム・プロシージャとカタログ (システム・テーブルとビュー) に関する情報について説明します。また、SQL Anywhere での SQL 言語の実装 (探索条件、構文、データ型、関数) についても説明します。
- **『SQL Anywhere サーバ - SQL の使用法』** このマニュアルでは、データベースの設計と作成の方法、データのインポート・エクスポート・変更の方法、データの検索方法、ストアド・プロシージャとトリガの構築方法について説明します。
- **『Mobile Link - クイック・スタート』** このマニュアルでは、セッションベースのリレーショナル・データベース同期システムである Mobile Link について説明します。Mobile Link テクノロジは、双方向レプリケーションを可能にし、モバイル・コンピューティング環境に非常に適しています。
- **『Mobile Link - クライアント管理』** このマニュアルでは、Mobile Link クライアントを設定、構成、同期する方法について説明します。Mobile Link クライアントには、SQL Anywhere または Ultra Light のいずれかのデータベースを使用できます。また、dbmlsync API についても説明します。dbmlsync API を使用すると、同期を C++ または .NET のクライアント・アプリケーションにシームレスに統合できます。
- **『Mobile Link - サーバ管理』** このマニュアルでは、Mobile Link アプリケーションを設定して管理する方法について説明します。
- **『Mobile Link - サーバ起動同期』** このマニュアルでは、Mobile Link サーバ起動同期について説明します。この機能により、Mobile Link サーバは同期を開始したり、リモート・デバイス上でアクションを実行することができます。
- **『QAnywhere』** このマニュアルでは、モバイル・クライアント、ワイヤレス・クライアント、デスクトップ・クライアント、およびラップトップ・クライアント用のメッセージング・プラットフォームである、QAnywhere について説明します。
- **『SQL Remote』** このマニュアルでは、モバイル・コンピューティング用の SQL Remote データ・レプリケーション・システムについて説明します。このシステムによって、SQL Anywhere の統合データベースと複数の SQL Anywhere リモート・データベースの間で、電子メールやファイル転送などの間接的リンクを使用したデータ共有が可能になります。

- **『Ultra Light - データベース管理とリファレンス』** このマニュアルでは、小型デバイス用 Ultra Light データベース・システムの概要を説明します。
- **『Ultra Light - C/C++ プログラミング』** このマニュアルでは、Ultra Light C および Ultra Light C++ のプログラミング・インタフェースについて説明します。Ultra Light を使用すると、ハンドヘルド・デバイス、モバイル・デバイス、埋め込みデバイスのデータベース・アプリケーションを開発し、これらのデバイスに配備できます。
- **『Ultra Light - M-Business Anywhere プログラミング』** このマニュアルは、Ultra Light for M-Business Anywhere について説明します。Ultra Light for M-Business Anywhere を使用すると、Palm OS、Windows Mobile、または Windows を搭載しているハンドヘルド・デバイス、モバイル・デバイス、または埋め込みデバイスの Web ベースのデータベース・アプリケーションを開発し、これらのデバイスに配備できます。
- **『Ultra Light - .NET プログラミング』** このマニュアルでは、Ultra Light.NET について説明します。Ultra Light.NET を使用すると、PC、ハンドヘルド・デバイス、モバイル・デバイス、または埋め込みデバイスのデータベース・アプリケーションを開発し、これらのデバイスに配備できます。
- **『Ultra Light J』** このマニュアルでは、Ultra Light J について説明します。Ultra Light J を使用すると、Java をサポートしている環境用のデータベース・アプリケーションを開発し、配備することができます。Ultra Light J は、BlackBerry スマートフォンと Java SE 環境をサポートしており、iAnywhere Ultra Light データベース製品がベースになっています。
- **『エラー・メッセージ』** このマニュアルでは、SQL Anywhere エラー・メッセージの完全なリストを示し、その診断情報を説明します。

表記の規則

この項では、このマニュアルで使用されている表記規則について説明します。

オペレーティング・システム

SQL Anywhere はさまざまなプラットフォームで稼働します。ほとんどの場合、すべてのプラットフォームで同じように動作しますが、いくつかの相違点や制限事項があります。このような相違点や制限事項は、一般に、基盤となっているオペレーティング・システム (Windows、UNIX など) に由来しており、使用しているプラットフォームの種類 (AIX、Windows Mobile など) またはバージョンに依存していることはほとんどありません。

オペレーティング・システムへの言及を簡素化するために、このマニュアルではサポートされているオペレーティング・システムを次のようにグループ分けして表記します。

- **Windows** Microsoft Windows ファミリを指しています。これには、主にサーバ、デスクトップ・コンピュータ、ラップトップ・コンピュータで使用される Windows Vista や Windows XP、およびモバイル・デバイスで使用される Windows Mobile が含まれます。

  特に記述がないかぎり、マニュアル中に Windows という記述がある場合は、Windows Mobile を含むすべての Windows ベース・プラットフォームを指しています。

- **UNIX**　特に記述がないかぎり、マニュアル中に UNIX という記述がある場合は、Linux および Mac OS X を含むすべての UNIX ベース・プラットフォームを指しています。

ディレクトリとファイル名

ほとんどの場合、ディレクトリ名およびファイル名の参照形式はサポートされているすべてのプラットフォームで似通っており、それぞれの違いはごくわずかです。このような場合は、Windows の表記規則が使用されています。詳細がより複雑な場合は、マニュアルにすべての関連形式が記載されています。

ディレクトリ名とファイル名の表記を簡素化するために使用されている表記規則は次のとおりです。

- **大文字と小文字のディレクトリ名**　Windows と UNIX では、ディレクトリ名およびファイル名には大文字と小文字が含まれている場合があります。ディレクトリやファイルが作成されると、ファイル・システムでは大文字と小文字の区別が維持されます。

  Windows では、ディレクトリおよびファイルを参照するとき、大文字と小文字は**区別されません**。大文字と小文字を混ぜたディレクトリ名およびファイル名は一般的に使用されますが、参照するときはすべて小文字を使用するのが通常です。SQL Anywhere では、*Bin32* や *Documentation* などのディレクトリがインストールされます。

  UNIX では、ディレクトリおよびファイルを参照するとき、大文字と小文字は**区別されます**。大文字と小文字を混ぜたディレクトリ名およびファイル名は一般的に使用されません。ほとんどの場合は、すべて小文字の名前が使用されます。SQL Anywhere では、*bin32* や *documentation* などのディレクトリがインストールされます。

  このマニュアルでは、ディレクトリ名に Windows の形式を使用しています。ほとんどの場合、大文字と小文字が混ざったディレクトリ名をすべて小文字に変換すると、対応する UNIX 用のディレクトリ名になります。

- **各ディレクトリおよびファイル名を区切るスラッシュ**　マニュアルでは、ディレクトリの区切り文字に円記号を使用しています。たとえば、PDF 形式のマニュアルは *install-dir¥Documentation¥ja¥pdf* にあります。これは Windows の形式です。

  UNIX では、円記号をスラッシュに置き換えます。PDF マニュアルは *install-dir/documentation/ja/pdf* にあります。

- **実行ファイル**　マニュアルでは、実行ファイルの名前は、Windows の表記規則が使用され、*.exe* や *.bat* などの拡張子が付きます。UNIX では、実行ファイルの名前に拡張子は付きません。

  たとえば、Windows でのネットワーク・データベース・サーバは *dbsrv11.exe* です。UNIX では *dbsrv11* です。

- ***install-dir***　インストール・プロセス中に、SQL Anywhere をインストールするロケーションを選択します。このロケーションを参照する環境変数 SQLANY11 が作成されます。このマニュアルでは、そのロケーションを *install-dir* と表します。

  たとえば、マニュアルではファイルを install-dir¥readme.txt のように参照します。これは、Windows では、%SQLANY11%¥readme.txt に対応します。UNIX では、$SQLANY11/readme.txt または ${SQLANY11}/readme.txt に対応します。

install-dir のデフォルト・ロケーションの詳細については、「SQLANY11 環境変数」『SQL Anywhere サーバ - データベース管理』を参照してください。

- ***samples-dir*** インストール・プロセス中に、SQL Anywhere に含まれるサンプルをインストールするロケーションを選択します。このロケーションを参照する環境変数 SQLANYSAMP11 が作成されます。このマニュアルではそのロケーションを *samples-dir* と表します。

  Windows エクスプローラ・ウィンドウで *samples-dir* を開くには、**[スタート] - [プログラム] - [SQL Anywhere 11] - [サンプル・アプリケーションとプロジェクト]** を選択します。

  samples-dir のデフォルト・ロケーションの詳細については、「SQLANYSAMP11 環境変数」『SQL Anywhere サーバ - データベース管理』を参照してください。

コマンド・プロンプトとコマンド・シェル構文

ほとんどのオペレーティング・システムには、コマンド・シェルまたはコマンド・プロンプトを使用してコマンドおよびパラメータを入力する方法が、1 つ以上あります。Windows のコマンド・プロンプトには、コマンド・プロンプト (DOS プロンプト) および 4NT があります。UNIX のコマンド・シェルには、Korn シェルおよび bash があります。各シェルには、単純コマンドからの拡張機能が含まれています。拡張機能は、特殊文字を指定することで起動されます。特殊文字および機能は、シェルによって異なります。これらの特殊文字を誤って使用すると、多くの場合、構文エラーや予期しない動作が発生します。

このマニュアルでは、一般的な形式のコマンド・ラインの例を示します。これらの例に、シェルにとって特別な意味を持つ文字が含まれている場合、その特定のシェル用にコマンドを変更することが必要な場合があります。このマニュアルではコマンドの変更について説明しませんが、通常、その文字を含むパラメータを引用符で囲むか、特殊文字の前にエスケープ文字を記述します。

次に、プラットフォームによって異なるコマンド・ライン構文の例を示します。

- **カッコと中カッコ** 一部のコマンド・ライン・オプションは、詳細な値を含むリストを指定できるパラメータを要求します。リストは通常、カッコまたは中カッコで囲まれています。このマニュアルでは、カッコを使用します。次に例を示します。

  ```
  -x tcpip(host=127.0.0.1)
  ```

  カッコによって構文エラーになる場合は、代わりに中カッコを使用します。

  ```
  -x tcpip{host=127.0.0.1}
  ```

  どちらの形式でも構文エラーになる場合は、シェルの要求に従ってパラメータ全体を引用符で囲む必要があります。

  ```
  -x "tcpip(host=127.0.0.1)"
  ```

- **引用符** パラメータの値として引用符を指定する必要がある場合、その引用符はパラメータを囲むために使用される通常の引用符と競合する可能性があります。たとえば、値に二重引用符を含む暗号化キーを指定するには、キーを引用符で囲み、パラメータ内の引用符をエスケープします。

  ```
  -ek "my ¥"secret¥" key"
  ```


多くのシェルでは、キーの値は my "secret" key のようになります。

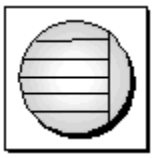
- **環境変数**　マニュアルでは、環境変数設定が引用されます。Windows のシェルでは、環境変数は構文 *%ENVVAR%* を使用して指定されます。UNIX のシェルでは、環境変数は構文 *$ENVVAR* または *${ENVVAR}* を使用して指定されます。

グラフィック・アイコン

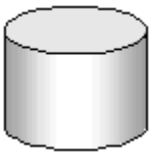
このマニュアルでは、次のアイコンを使用します。

- クライアント・アプリケーション。

- SQL Anywhere などのデータベース・サーバ。



- データベース。ハイレベルの図では、データベースとデータベースを管理するデータ・サーバの両方をこのアイコンで表します。



- レプリケーションまたは同期のミドルウェア。ソフトウェアのこれらの部分は、データベース間のデータ共有を支援します。たとえば、Mobile Link サーバ、SQL Remote Message Agent などが挙げられます。

- プログラミング・インタフェース。

ドキュメンテーション・チームへのお問い合わせ

このヘルプに関するご意見、ご提案、フィードバックをお寄せください。

SQL Anywhere ドキュメンテーション・チームへのご意見やご提案は、弊社までご連絡ください。頂戴したご意見はマニュアルの向上に役立たせていただきます。ぜひとも、ご意見をお寄せください。

DocCommentXchange

DocCommentXchange を使用して、ヘルプ・トピックに関するご意見を直接お寄せいただくこともできます。DocCommentXchange (DCX) は、SQL Anywhere マニュアルにアクセスしたり、マニュアルについて議論するためのコミュニティです。DocCommentXchange は次の目的に使用できます (現在のところ、日本語はサポートされておりません)。

- マニュアルを表示する
- マニュアルの項目について明確化するために、ユーザによって追加された内容を確認する
- すべてのユーザのために、今後のリリースでマニュアルを改善するための提案や修正を行う

http://dcx.sybase.com を参照してください。

詳細情報の検索／テクニカル・サポートの依頼

詳しい情報やリソースについては、iAnywhere デベロッパー・コミュニティ (http://www.iAnywhere.jp/developers/index.html) を参照してください。

ご質問がある場合や支援が必要な場合は、次に示す Sybase iAnywhere ニュースグループのいずれかにメッセージをお寄せください。

ニュースグループにメッセージをお送りいただく際には、ご使用の SQL Anywhere バージョンのビルド番号を明記し、現在発生している問題について詳しくお知らせくださいますようお願いいたします。バージョンおよびビルド番号を調べるには、コマンド **dbeng11 -v** を実行します。

ニュースグループは、ニュース・サーバ *forums.sybase.com* にあります。

以下のニュースグループがあります。

- ianywhere.public.japanese.general

Web 開発に関する問題については、http://groups.google.com/group/sql-anywhere-web-development を参照してください。

ニュースグループに関するお断り
iAnywhere Solutions は、ニュースグループ上に解決策、情報、または意見を提供する義務を負うものではありません。また、システム・オペレータ以外のスタッフにこのサービスを監視させて、操作状況や可用性を保証する義務もありません。

iAnywhere のテクニカル・アドバイザとその他のスタッフは、時間のある場合にかぎりニュースグループでの支援を行います。こうした支援は基本的にボランティアで行われるため、解決策や情報を定期的に提供できるとはかぎりません。支援できるかどうかは、スタッフの仕事量に左右されます。

Ultra Light for C/C++ の開発者

目次

C と C++ インタフェースは、小型デバイスを対象とした Ultra Light 開発者に次のような利益を提供します。

- 占有容量が小さくパフォーマンスが高いデータベース・ストア
- C または C++ 言語の優れた機能、効率、柔軟性
- Windows Mobile、Palm OS、Windows デスクトップ・プラットフォームでのアプリケーション配備機能

Ultra Light データベースの機能の詳細については、「Ultra Light データベースの作成と設定」『Ultra Light データベース管理とリファレンス』を参照してください。

C++ を使用する Ultra Light 開発者には、2 つのオプションが用意されています。

- Ultra Light C++ API
- ODBC プログラミング・インタフェース (コンポーネント・インタフェース)

C を使用する Ultra Light 開発者は、Embedded SQL または ODBC プログラミング・インタフェースを使用する必要があります。

Embedded SQL アプリケーションの開発

Embedded SQL アプリケーションを開発するときは、SQL 文に標準の C または C++ ソース・コードを混在させます。Embedded SQL アプリケーションを開発するには、C または C++ のプログラミング言語に精通していることが必要です。

Embedded SQL アプリケーションの開発プロセスは、次のとおりです。

1. Ultra Light データベースを作成します。
2. 通常 *.sqc* という拡張子の付いた Embedded SQL ソース・ファイルにソース・コードを記述します。

   ソース・コードにデータ・アクセスが必要な場合は、"EXEC SQL" キーワードに続いて SQL 文を指定して実行します。次に例を示します。

   ```
   EXEC SQL SELECT price, prod_name
     INTO :cost, :pname
     FROM ULProduct
     WHERE prod_id= :pid;
   if((SQLCODE==SQLE_NOTFOUND)||(SQLCODE<0)) {
    return(-1);
   }
   ```
3. *.sqc* ファイルの前処理を実行します。

   SQL Anywhere には SQL プリプロセッサ (sqlpp) が用意されており、*.sqc* ファイルを読み込んで *.cpp* ファイルを生成します。これらのファイルには Ultra Light ランタイム・ライブラリへの関数呼び出しが格納されています。
4. *.cpp* ファイルをコンパイルします。
5. *.cpp* ファイルをリンクします。

   コンパイルしたファイルは、Ultra Light ランタイム・ライブラリにリンクする必要があります。

Embedded SQL での開発の詳細については、「Embedded SQL アプリケーションの構築」 64 ページを参照してください。

システムの稼働条件とサポートされるプラットフォーム

開発プラットフォーム

Ultra Light C++ を使用してアプリケーションを開発するには、以下が必要です。

- Microsoft Windows デスクトップ (開発プラットフォーム)
- サポートされる C/C++ コンパイラ

ターゲット・プラットフォーム

Ultra Light C/C++ は、次のターゲット・プラットフォームをサポートしています。

- Windows Mobile 3.0 以降
- Palm OS 4.0 以降

サポートされるターゲット・プラットフォームの詳細については、http://www.ianywhere.jp/developers/technotes/os_components_1101.html を参照してください。

Ultra Light C++ コンポーネント・アーキテクチャ

Ultra Light C++ コンポーネント・インタフェースは、*uliface.h* ヘッダ・ファイルに定義されています。次のリストは、よく使用されるオブジェクトの一部を示します。

- **DatabaseManager**　アプリケーションごとに 1 つの DatabaseManager オブジェクトを作成します。
- **Connection**　Ultra Light データベースへの接続を示します。Connection オブジェクトは 1 つまたは複数作成できます。
- **Table**　データベース内のデータへのアクセスを提供します。
- **PreparedStatement、ResultSet、および ResultSetSchema**　動的 SQL 文の作成、クエリの記述、INSERT、UPDATE、DELETE 文の実行、プログラムによるデータベースの結果セットの制御を行います。
- **SyncParms**　Ultra Light データベースを Mobile Link 同期サーバと同期させます。

API リファレンスへのアクセスの詳細については、「Ultra Light C++ API リファレンス」 131 ページを参照してください。

SQLCA (SQL Communications Area) の概要

すべての Ultra Light C/C++ インタフェースは、同じ Ultra Light ランタイム・エンジンを使用します。したがって、各 API は同じ基本機能へのアクセスを提供します。

どの Ultra Light C/C++ インタフェースでも、Ultra Light ランタイムとアプリケーションとの間でデータをマーシャリングさせるのに、同じ基本データ構造体を共有しています。このデータ構造体が、SQLCA (SQL Communications Area) です。各 SQLCA には現在の接続があり、別々のスレッドが共通の SQLCA を共有することはできません。

アプリケーション・コードでは、データベースに接続する前に次の処理を実行してください。

- SQLCA の初期化。アプリケーションと Ultra Light ランタイムとの通信に備えます。
- エラー・コールバック関数の登録。
- データベースの起動。この処理は、接続を開く処理の一部として実行できます。

次の関数は、これらのタスクを同じように実行します。

タスク	インタフェース	関数
SQLCA の初期化	Embedded SQL	db_init
	C++	ULSqlca::Initialize
SQLCA の初期化とデータベースの起動	Embedded SQL	db_init db_start_database
	C++	UltraLite_DatabaseManager の接続関数の一部としてデータベースを起動する

データベースの作成

Ultra Light データベースは、次のどの方法によっても作成できます。

- Sybase Central の**データベース作成ウィザード**
- ulcreate、ulinit などのコマンド・ライン・ユーティリティ
- ULCreateDataBase 関数の呼び出し

Sybase Central を使用した場合、データベースは目的のテーブルやその他のスキーマ関連項目に適した定義とともに、対話型で作成されます。

ulcreate ユーティリティを使用した場合、テーブルが何も定義されていない空のデータベースが作成されます。ULCreateDatabase を呼び出してデータベースを作成するアプリケーションでは、SQL CREATE 文を実行してテーブルとインデックスの定義を作成する必要もあります。

データベース名の明示指定

インタフェースが異なると、データベースに対して使用するデフォルト・ファイル名も異なる場合があります。このため、インタフェースを混在させる場合は、データベースの作成時や接続時に、データベースの名前を必ず明示的に指定するのが効果的です。この操作は、DBN= 接続パラメータを使用して行います。「Ultra Light DBN 接続パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。

アプリケーション開発

この項では、Ultra Light C/C++ プログラミングにおける開発上の注意について説明します。

Ultra Light C++ API を使用したアプリケーションの開発

目次

Ultra Light ネームスペースの使用

Ultra Light C++ インタフェースには、UltraLite_Connection クラスや UltraLite_DatabaseManager クラスのように、名前に UltraLite_ というプレフィクスが付いているクラスのセットがあります。これらの各クラスのほとんどの関数は、文字列 _iface が追加された基本となるインタフェースからの関数を実装しています。たとえば、UltraLite_Connection クラスは UltraLite_Connection_iface からの関数を実装しています。

明示的に Ultra Light ネームスペースを使用するときは、省略名を使用して各クラスを参照できます。Ultra Light ネームスペースを使用している場合は、UltraLite_Connection オブジェクトとして接続を宣言するのではなく、次のように Connection オブジェクトとして接続を宣言できます。

```
using namespace UltraLite;
ULSqlca sqlca;
sqlca.Initialize();
DatabaseManager * dbMgr = ULInitDatabaseManager(sqlca);
Connection *  conn = UL_NULL;
```

このアーキテクチャの結果として、この章のコード・サンプルでは DatabaseManager、Connection、TableSchema などと記述されていますが、それぞれ UltraLite_DatabaseManager_iface、UltraLite_Connection_iface、UltraLite_TableSchema_iface へのリンクによって詳細を参照できます。

データベースへの接続

Ultra Light アプリケーションをデータベースに接続しないと、データを操作できません。この項では、Ultra Light データベースに接続するための方法について説明します。

サンプル・コードは、*samples-dir¥UltraLite¥CustDB¥* ディレクトリにあります。

Connection オブジェクトのプロパティ

- **コミット動作**　Ultra Light C++ API には、オートコミット・モードはありません。各トランザクションの後には Conn->Commit() 文を指定します。「トランザクションの管理」 23 ページを参照してください。
- **ユーザ認証**　接続許可の付与と取り消しを行うメソッドを使用すると、アプリケーションのユーザ ID とパスワードを (それぞれデフォルト値である **DBA** と **sql** から) 別の値に変更できます。各データベースは、最大 4 つのユーザ ID を保持できます。「ユーザの認証」 26 ページを参照してください。
- **同期**　Connection オブジェクトのメソッドを使用すると、Ultra Light を統合データベースと同期できます。「データの同期」 28 ページを参照してください。
- **テーブル**　Ultra Light データベース・テーブルには、Connection オブジェクトのメソッドを使用してアクセスします。「テーブル API を使用したデータへのアクセス」 17 ページを参照してください。
- **準備文**　SQL 文の実行を処理するメソッドが提供されます。「SQL を使用したデータへのアクセス」 13 ページと「UltraLite_PreparedStatement クラス」 207 ページを参照してください。

Ultra Light データベースへの接続

♦ **Ultra Light データベースに接続するには、次の手順に従います。**

1. Ultra Light ネームスペースを使用します。

   Ultra Light ネームスペースを使用すると、C++ インタフェースでクラスの省略名を使用できます。

   ```
   using namespace UltraLite;
   ```

2. DatabaseManager オブジェクトと ULSqlca (Ultra Light SQL Communications Area) を作成し、初期化します。ULSqlca は、アプリケーションとデータベースの間の通信を処理する構造体です。

   DatabaseManager オブジェクトは、オブジェクト階層のルートにあります。DatabaseManager オブジェクトは、1 つのアプリケーションに 1 つだけ作成します。多くの場合、DatabaseManager オブジェクトは、アプリケーションに対してグローバルに宣言するのが最も効果的です。

   ```
   ULSqlca sqlca;
   sqlca.Initialize();
   DatabaseManager * dbMgr = ULInitDatabaseManager(sqlca);
   ```

アプリケーションで SQL サポートが必要でなく、かつ Ultra Light ランタイムに直接リンクする場合、アプリケーションでは ULInitDatabaseManagerNoSQL を呼び出して ULSqlca を初期化できます。この方法を使用すると、アプリケーションのサイズを小さくできます。

「UltraLite_DatabaseManager_iface クラス」 192 ページを参照してください。

3. 既存のデータベースへの接続を開きます。または、指定のデータベース・ファイルが存在しない場合は、新しいデータベースを作成します。「OpenConnection 関数」 193 ページを参照してください。

   Ultra Light アプリケーションは空の初期データベースで配備することができます。また、Ultra Light データベースがまだ存在しない場合は、アプリケーションでデータベースを作成することもできます。初期データベースを配備するのが最も簡単なソリューションです。それ以外の場合は、アプリケーションで ULCreateDatabase 関数を呼び出してデータベースを作成したり、アプリケーションで必要なすべてのテーブルを作成したりする必要があります。「ULCreateDatabase 関数」 108 ページを参照してください。

```
Connection * conn = dbMgr->OpenConnection( sqlca, UL_TEXT("dbf=mydb.udb") );
if( sqlca.GetSQLCode() ==
  SQLE_ULTRALITE_DATABASE_NOT_FOUND ) {
    printf( "Open failed with sql code: %d.¥n" , sqlca.GetSQLCode() );
  }
}
```

マルチスレッド・アプリケーション

各接続と、それを基に作成されるすべてのオブジェクトは、単一のスレッドで使用してください。アプリケーションが Ultra Light データベースにアクセスするのに複数のスレッドを必要とする場合は、スレッドごとに個別の接続が必要です。

SQL を使用したデータへのアクセス

Ultra Light アプリケーションは、SQL 文を実行するかテーブル API を使用してテーブル・データにアクセスできます。この項では、SQL 文を使用したデータ・アクセスについて説明します。

テーブル API の使用方法については、「テーブル API を使用したデータへのアクセス」 17 ページを参照してください。

この項では、SQL を使用して次の操作を行う方法を説明します。

- ローの挿入、削除、更新
- 結果セットのローの取得
- 結果セットのローのスクロール

この項では、SQL 言語については説明しません。SQL 言語の詳細については、「Ultra Light SQL 文」 『Ultra Light データベース管理とリファレンス』を参照してください。

データ操作：挿入、削除、更新

Ultra Light では、ExecuteStatement メソッド (PreparedStatement クラスのメンバ) を使用して、SQL データ操作を実行できます。

「UltraLite_PreparedStatement クラス」 207 ページを参照してください。

準備文のパラメータの参照
Ultra Light では、? 文字を使用してクエリのパラメータを示します。INSERT 文、UPDATE 文、DELETE 文では必ず、準備文での順序位置に従ってそれぞれの ? が参照されます。たとえば、最初の ? はパラメータ 1、2 番目の ? はパラメータ 2、のようになります。

♦ ローを挿入するには、次の手順に従います。

1. PreparedStatement を宣言します。

   ```
   PreparedStatement * prepStmt;
   ```

   「PrepareStatement 関数」 172 ページを参照してください。
2. SQL 文を PreparedStatement オブジェクトに割り当てます。

   ```
   prepStmt = conn->PrepareStatement( UL_TEXT("INSERT INTO MyTable(MyColumn) values (?)") );
   ```
3. 文の入力パラメータ値を割り当てます。

   次のコードは、文字列パラメータを示します。

   ```
   prepStmt->SetParameter( 1, UL_TEXT("newValue") );
   ```
4. 準備文を実行します。

   戻り値は、文に影響されたローの数を示します。

```
ul_s_long rowsInserted;
rowsInserted = prepStmt->ExecuteStatement();
```

5. 変更をコミットします。

```
conn->Commit();
```

♦ **ローを削除するには、次の手順に従います。**

1. PreparedStatement を宣言します。

```
PreparedStatement * prepStmt;
```

2. SQL 文を PreparedStatement オブジェクトに割り当てます。

```
ULValue sqltext( );
prepStmt = conn->PrepareStatement( UL_TEXT("DELETE FROM MyTable WHERE MyColumn
= ?") );
```

3. 文の入力パラメータ値を割り当てます。

```
prepStmt->SetParameter( 1, UL_TEXT("deleteValue") );
```

4. 文を実行します。

```
ul_s_long rowsDeleted;
rowsDeleted = prepStmt->ExecuteStatement();
```

5. 変更をコミットします。

```
conn->Commit();
```

♦ **ローを更新するには、次の手順に従います。**

1. PreparedStatement を宣言します。

```
PreparedStatement * prepStmt;
```

2. PreparedStatement オブジェクトに文を割り当てます。

```
prepStmt = conn->PrepareStatement(
   UL_TEXT("UPDATE MyTable SET MyColumn1 = ? WHERE MyColumn1 = ?") );
```

3. 文の入力パラメータ値を割り当てます。

```
prepStmt->SetParameter( 1, UL_TEXT("newValue") );
prepStmt->SetParameter( 2, UL_TEXT("oldValue") );
```

4. 文を実行します。

```
ul_s_long rowsUpdated;
rowsUpdated = prepStmt->ExecuteStatement();
```

5. 変更をコミットします。

```
conn->Commit();
```


データ検索：SELECT

SELECT 文を使用すると、データベースから情報を取り出すことができます。SELECT 文を実行すると、PreparedStatement.ExecuteQuery メソッドは ResultSet オブジェクトを返します。

「UltraLite_PreparedStatement_iface クラス」 208 ページを参照してください。

♦ **SELECT 文を実行するには、次の手順に従います。**

1. 準備文オブジェクトを作成します。

```
PreparedStatement * prepStmt =
   conn->PrepareStatement( UL_TEXT("SELECT MyColumn FROM MyTable") );
```

2. 文を実行します。

   次のコードでは、SELECT クエリの結果に文字列が含まれています。これはコマンド・プロンプトへ出力されます。

```
#define MAX_NAME_LEN    100
ULValue val;
ResultSet * rs = prepStmt->ExecuteQuery();
while( rs->Next() ){
  char mycol[ MAX_NAME_LEN ];
  val = rs->Get( 1 );
  val.GetString( mycol, MAX_NAME_LEN );
  printf( "mycol= %s¥n", mycol );
}
```

SQL 結果セットのナビゲーション

ResultSet オブジェクトに関連したメソッドを使用して、結果セット内をナビゲーションすることができます。

結果セット・オブジェクトは、結果セットをナビゲーションする次のメソッドを提供します。

- **AfterLast**　カーソルを最後のローの直後に配置します。
- **BeforeFirst**　最初のローの直前に配置します。
- **First**　最初のローに移動します。
- **Last**　最後のローに移動します。
- **Next**　次のローに移動します。
- **Previous**　前のローに移動します。
- **Relative(offset)**　現在のローを基準にして、符号付きオフセット値で指定された数だけローを移動します。オフセット値を正の値で指定すると、現在の結果セットのカーソル位置から前方に移動します。負の値で指定すると後方に移動します。オフセット値が 0 の場合、カーソルは現在のロケーションから移動しませんが、ロー・バッファが再配置されます。

「UltraLite_ResultSet_iface クラス」 213 ページを参照してください。

結果セット・スキーマの説明

ResultSet->GetSchema メソッドを使用すると、カラム名、カラムの総数、カラム・スケール、カラム・サイズ、カラム SQL 型など、結果セットに関するスキーマ情報を取得できます。

例

次のサンプル・コードは、ResultSet.GetSchema メソッドを使用して、スキーマ情報をコマンド・プロンプトに表示する方法を示しています。

```
ResultSetSchema * rss = rs->GetSchema();
ULValue val;
char name[ MAX_NAME_LEN ];

for( int i = 1;
     i <= rss->GetColumnCount();
     i++ ){
   val = rss->GetColumnName( i );
   val.GetString( name, MAX_NAME_LEN );
   printf( "id= %d, name= %s ¥n", i, name );
}
```

「GetSchema 関数」 168 ページを参照してください。

テーブル API を使用したデータへのアクセス

Ultra Light アプリケーションは、SQL 文を実行するかテーブル API を使用してテーブル・データにアクセスできます。この項では、テーブル API を使用したデータ・アクセスについて説明します。

SQL 文を実行することによるデータへのアクセスの詳細については、「SQL を使用したデータへのアクセス」 13 ページを参照してください。

この項では、テーブル API を使用して次の操作を行う方法について説明します。

- テーブルのローのスクロール
- 現在のローの値へのアクセス
- find メソッドと lookup メソッドを使用したテーブルのローの検索
- ローの挿入、削除、更新

テーブルのローのナビゲーション

Ultra Light C++ API は、幅広いナビゲーション作業を行うために、テーブルをナビゲーションする複数のメソッドを提供します。

テーブル・オブジェクトは、テーブルをナビゲーションする次のメソッドを提供します。

- **AfterLast**　カーソルを最後のローの直後に配置します。
- **BeforeFirst**　最初のローの直前に配置します。
- **First**　最初のローに移動します。
- **Last**　最後のローに移動します。
- **Next**　次のローに移動します。
- **Previous**　前のローに移動します。
- **Relative(offset)**　現在のローを基準にして、符号付きオフセット値で指定された数だけローを移動します。オフセット値を正の値で指定すると、現在の結果セットのカーソル位置から前方に移動します。負の値で指定すると後方に移動します。オフセット値が 0 の場合、カーソルは現在のロケーションから移動しませんが、ロー・バッファが再配置されます。

「UltraLite_Table_iface クラス」 229 ページを参照してください。

例

次のサンプル・コードは、MyTable テーブルを開き、各ローの MyColumn カラムの値を表示します。

```
Table * tbl = conn->openTable( "MyTable" );
ul_column_num colID =
  tbl->GetSchema()->GetColumnID( "MyColumn" );
```

```
while ( tbl->Next() ){
  char buffer[ MAX_NAME_LEN ];
  ULValue colValue = tbl->Get(colID);
  colValue.GetString(buffer, MAX_NAME_LEN);
  printf( "%s¥n", buffer );
}
```

テーブル・オブジェクトを開くと、テーブルのローがアプリケーションに公開されます。デフォルトでは、ローはプライマリ・キー値の順に並んでいますが、テーブルを開くときにインデックスを指定すると特定の順序でローにアクセスできます。

例

次のコード・フラグメントは、ix_col インデックスで順序付けられた MyTable テーブルの最初のローに移動します。

```
ULValue table_name( UL_TEXT("MyTable") )
ULValue index_name( UL_TEXT("ix_col") )
Table * tbl =
  conn->OpenTableWithIndex( table_name, index_name );
```

「UltraLite_Table_iface クラス」 229 ページを参照してください。

Ultra Light のモード

Ultra Light モードは、バッファ内の値の使用方法を指定します。Ultra Light のモードは次のいずれかに設定できます。

- **挿入モード**　Insert メソッドを呼び出すと、バッファ内のデータが新しいローとしてテーブルに追加されます。
- **更新モード**　Update メソッドを呼び出すと、現在のローがバッファ内のデータに置き換えられます。
- **検索モード**　find メソッドの 1 つが呼び出されたときに、値がバッファ内のデータに正確に一致するローの検索が検索されます。
- **ルックアップ・モード**　いずれかの lookup メソッドが呼び出されたときに、バッファ内のデータと一致するか、それより大きい値のローが検索されます。

モードを設定するには、モードを設定するための対応メソッドを呼び出します。たとえば InsertBegin、BeginUpdate、FindBegin などです。

現在の行へのアクセス

Table オブジェクトは、次のいずれかの位置に常に置かれています。

- テーブルの最初のローの前
- テーブルのいずれかのローの上
- テーブルの最後のローの後ろ

Table オブジェクトがローの上に置かれている場合は、そのデータ型に適したメソッド・セットを使用して、そのローのカラムの値を取得したり、変更したりできます。

カラム値の取得

Table オブジェクトは、カラム値を取得するメソッド・セットを提供します。これらのメソッドは、カラム ID を引数として取ります。

次のコード・フラグメントは、lname カラムの値を取得します。このカラムの値は文字列です。

```
ULValue val;
char lname[ MAX_NAME_LEN ];
val = tbl->Get( UL_TEXT("lname") );
val.GetString( lname, MAX_NAME_LEN );
```

次のコードは、cust_id カラムの値を取得します。このカラムの値は整数です。

```
int id = tbl->Get( UL_TEXT("cust_id") );
```

カラム値の変更

値を取り出すメソッド以外に、値を設定するメソッドもあります。値を設定するメソッドは、カラム ID と値を引数として取ります。

たとえば、次のコードは、lname カラムの値を Kaminski に設定します。

```
ULValue lname_col( UL_TEXT("fname") );
ULValue v_lname( UL_TEXT("Kaminski") );
tbl->Set( lname_col, v_lname );
```

カラムの値を設定することにより、データベースのデータが直接変更されることはありません。位置がテーブルの最初のローの前または最後のローの後ろにある場合でも、プロパティに値を割り当てることができます。現在のローが定義されていないときに、データにアクセスしようとしないでください。たとえば、次の例でカラムの値をフェッチしようとすることは不正です。

```
// This code is incorrect
tbl.BeforeFirst();
id = tbl.Get( cust_id );
```

値のキャスト

選択するメソッドは、割り当てるデータ型に一致させてください。データ型に互換性がある場合は、Ultra Light が自動的にデータベースのデータ型をキャストするため、GetString メソッドを使用して整数値を文字列変数にフェッチしたりできます。「データ型の明示的な変換」『Ultra Light データベース管理とリファレンス』を参照してください。

ローの検索

Ultra Light には、データを操作するための操作モードがいくつかあります。検索では、これらのモードのうち、検索とルックアップの 2 つを使用できます。Table オブジェクトには、テーブル内の特定のローを検索するために、これらのモードに対応するメソッドがあります。

注意
Find メソッドと Lookup メソッドを使用して検索されるカラムは、テーブルを開くのに使用されたインデックスにあることが必要です。

- **find メソッド**　Table オブジェクトを開いたときに指定したソート順に基づいて、指定された検索値と正確に一致する最初のローに移動します。検索値が見つからない場合は、最初のローの前、または最後のローの後ろに位置設定されます。
- **lookup メソッド**　Table オブジェクトを開いたときに指定したソート順に基づいて、指定された検索値と一致するか、それより大きい値の最初のローに移動します。

♦ **ローを検索するには、次の手順に従います。**

1. 検索モードまたはルックアップ・モードを開始します。

   テーブル・オブジェクトでメソッドを呼び出してモードを設定します。たとえば、次のコードは検索モードを開始します。

   ```
   tbl.FindBegin();
   ```

2. 検索値を設定します。

   現在のローで検索値を設定します。これらの値の設定は、データベースではなく、現在のローを保持しているバッファにのみ影響します。たとえば、次のコードは、バッファ内の値を Kaminski に設定します。

   ```
   ULValue lname_col = t->GetSchema()->GetColumnID( UL_TEXT("lname") );
   ULValue v_lname( UL_TEXT("Kaminski") );
   tbl.Set( lname_col, v_lname );
   ```

3. ローを検索します。

   適切なメソッドを呼び出して検索を実行します。たとえば、次のコードは、現在のインデックスで指定された値と正確に一致する最初のローを検索します。

   マルチカラム・インデックスの場合、最初のカラムの値が常に使用され、他のカラムは省略できます。

   ```
   tCustomer.FindFirst();
   ```

4. ローの次のインスタンスを検索します。

   適切なメソッドを呼び出して検索を実行します。検索操作の場合は、FindNext でインデックス内のパラメータの次のインスタンスを検索します。ルックアップ操作では、MoveNext で次のインスタンスを検索します。

「UltraLite_Table_iface クラス」 229 ページを参照してください。

ローの更新

次の手順では、ローを更新します。

♦ **ローを更新するには、次の手順に従います。**

1. 更新するローに移動します。

   テーブルをスクロールするか、find メソッドと lookup メソッドを使用してテーブルを検索し、ローに移動できます。

2. 更新モードを開始します。

   たとえば、次の指示は、テーブル tbl 上で更新モードを開始します。

   ```
   tbl.BeginUpdate();
   ```

3. 更新するローの新しい値を設定します。たとえば、次の指示は、バッファ内の id カラムを 3 に設定します。

   ```
   tbl.Set( UL_TEXT("id"), 3 );
   ```

4. Update を実行します。

   ```
   tbl.Update();
   ```

更新操作が終了すると、更新したローが現在のローになります。

Ultra Light C++ API は、conn->Commit() を使用してコミットしないかぎり、データベースに変更内容をコミットしません。「トランザクションの管理」 23 ページを参照してください。

警告
ローのプライマリ・キーを更新しないでください。代わりに、ローを削除して新しいローを追加してください。

ローの挿入

ローの挿入手順は、ローの更新手順とほぼ同じです。ただし、挿入操作の場合は、テーブル内のローをあらかじめ指定する必要はありません。ローをテーブルに挿入する順序は重要ではありません。データは、常にインデックスに従ってデータベースに挿入されるからです。

例

次のコード・フラグメントでは、新しいローが挿入されます。

```
tbl.InsertBegin();
tbl.Set( UL_TEXT("id"), 3 );
tbl.Set( UL_TEXT("lname"), "Carlo" );
tbl.Insert();
tbl.Commit();
```

カラムの値を設定しない場合、そのカラムにデフォルト値があるときはデフォルト値が使用されます。カラムにデフォルトがない場合は、次のエントリが使用されます。

- NULL 入力可のカラムの場合は NULL
- NULL 入力不可の数値カラムの場合は 0
- NULL 入力不可の文字カラムの場合は空の文字列
- 明示的に値を NULL に設定するには、SetNull メソッドを使用します。

ローの削除

ローの削除手順は、ローの挿入や更新よりも簡単です。

次の手順は、ローを削除します。

♦ **ローを削除するには、次の手順に従います。**

1. 削除するローに移動します。
2. Table.Delete メソッドを実行します。

```
tbl.Delete();
```

トランザクションの管理

Ultra Light C++ API は、オートコミット・モードをサポートしません。トランザクションは、明示的にコミットまたはロール・バックされる必要があります。

♦ **トランザクションをコミットするには、次の手順に従います。**

● Conn->Commit 文を実行します。

「Commit 関数」 161 ページを参照してください。

♦ **トランザクションをロールバックするには、次の手順に従います。**

● Conn->Rollback 文を実行します。

「Rollback 関数」 174 ページを参照してください。

Ultra Light におけるトランザクション管理の詳細については、「Ultra Light でのトランザクション処理」『Ultra Light データベース管理とリファレンス』を参照してください。

スキーマ情報へのアクセス

API のオブジェクトは、テーブル、カラム、インデックス、同期パブリケーションを表します。各オブジェクトには、そのオブジェクトの構造情報へアクセスするための GetSchema メソッドがあります。

API によるスキーマの変更はできません。スキーマに関する情報の取得のみが可能です。

次のスキーマ・オブジェクトと情報にアクセスできます。

- **DatabaseSchema**　データベース内のテーブルの数と名前、日付と時刻のフォーマットなどのグローバル・プロパティを公開します。

  DatabaseSchema オブジェクトを取得するには、Conn->GetSchema を使用します。

  「GetSchema 関数」 168 ページを参照してください。

- **TableSchema**　このテーブル内のカラムとインデックスの数と名前を公開します。

  TableSchema オブジェクトを取得するには、tbl->GetSchema を使用します。

  「GetSchema 関数」 168 ページを参照してください。

- **IndexSchema**　インデックス内のカラムに関する情報を返します。インデックスには直接に対応するデータがないため、個別の Index クラスはなく、IndexSchema クラスのみが存在します。

  IndexSchema オブジェクトを取得するには、table_schema->GetIndexSchema メソッドまたは table_schema->GetPrimaryKey メソッドを呼び出します。

  「UltraLite_Table_iface クラス」 229 ページを参照してください。

エラー処理

データベース操作が終わるたびに、ULSqlca オブジェクトのメソッドを使用してエラーをチェックしてください。たとえば、LastCodeOK を使って操作が成功したかどうかをチェックします。また、GetSQLCode は SQLCode の数値を返します。これらの値の意味の詳細については、「SQL Anywhere のエラー・メッセージ (Sybase エラー・コード順)」『エラー・メッセージ』を参照してください。

Ultra Light では、明示的なエラー処理に加えて、エラー・コールバック関数をサポートしています。コールバック関数を登録すると、Ultra Light エラーが発生するたびに関数が呼び出されます。コールバック関数がアプリケーション・フローを制御することはありませんが、すべてのエラーを通知することができます。コールバック関数を使用すると、アプリケーションの開発中やデバッグ中は特に効果的です。コールバック関数の使用方法については、「チュートリアル : C++ API を使用したアプリケーションの構築」 329 ページを参照してください。

コールバック関数のサンプルについては、「ULRegisterErrorCallback のコールバック関数」 100 ページと「ULRegisterErrorCallback 関数」 122 ページを参照してください。

Ultra Light C++ API によってスローされるエラー・コードのリストについては、「SQL Anywhere のエラー・メッセージ (Sybase エラー・コード順)」『エラー・メッセージ』を参照してください。

ユーザの認証

Ultra Light データベースには、最大 4 つのユーザ ID を定義できます。Ultra Light データベースは、デフォルトのユーザ ID **DBA** とパスワード **sql** を使用して作成されます。Ultra Light データベースへのすべての接続では、ログイン・ユーザ ID とパスワードを指定する必要があります。パスワードの変更とユーザ ID の追加や削除は、接続が確立されると実行できます。

ユーザ ID を直接変更することはできません。ユーザ ID を追加して、既存のユーザ ID を削除することはできます。

♦ **ユーザを追加する、または既存のユーザのパスワードを変更するには、次の手順に従います。**

1. 既存のユーザとしてデータベースに接続します。
2. conn->GrantConnectTo メソッドを使用し、希望するパスワードでユーザに接続権限を付与します。

   新規ユーザを追加する場合も、既存のユーザのパスワードを変更する場合も、この手順は同じです。

   「GrantConnectTo 関数」 169 ページを参照してください。

♦ **既存のユーザを削除するには、次の手順に従います。**

1. 既存のユーザとしてデータベースに接続します。
2. conn->RevokeConnectFrom メソッドを使用して、ユーザの接続権限を取り消します。

   「RevokeConnectFrom 関数」 174 ページを参照してください。

データの暗号化

Ultra Light C++ API を使用して、Ultra Light データベースを暗号化するか、難読化するかを選択できます。暗号化ではデータベースのデータを非常に安全に表現できますが、難読化ではデータベースの内容を不用意に閲覧されないことを目的とした簡易的なセキュリティを実現します。

補足情報については、「Ultra Light で使用するデータベース作成パラメータの選択」『Ultra Light データベース管理とリファレンス』を参照してください。

暗号化

暗号化されたデータベースを作成するには、接続文字列に **key=** 接続パラメータを指定することによって、暗号化キーを指定します。CreateDatabase メソッドを呼び出すと、データベースが作成され、指定されたキーで暗号化されます。

データベースが暗号化された後は、データベースへのすべての接続で正しい暗号化キーを指定する必要があります。そうしないと、接続は失敗します。

「Ultra Light DBKEY 接続パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。

難読化

データベースを難読化するには、データベース作成パラメータとして obfuscate=1 を指定します。

「Ultra Light データベースの保護」『Ultra Light データベース管理とリファレンス』を参照してください。

データの同期

Ultra Light アプリケーションでは、データを中央のデータベースに同期できます。同期には、SQL Anywhere に付属の Mobile Link 同期ソフトウェアが必要です。

Ultra Light C++ API は、TCP/IP、TLS、HTTP、HTTPS 通信による同期をサポートします。同期は、Ultra Light アプリケーションによって開始されます。いずれの場合でも、接続オブジェクトのメソッドとプロパティを使用して同期を制御します。

同期の詳細については、「Ultra Light クライアント」『Ultra Light データベース管理とリファレンス』を参照してください。

同期に使用する ul_synch_info 構造体の詳細については、使用文字が ASCII 文字かワイド文字かに応じて「ul_synch_info_a 構造体」 135 ページまたは「ul_synch_info_w2 構造体」 138 ページを参照してください。

同期パラメータの詳細については、「Ultra Light の同期パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。

アプリケーションのコンパイルとリンク

ランタイム・ライブラリのセットは、いくつかのプラットフォームで Ultra Light C++ API を使用する場合に使用できます。これらのライブラリには、Windows Mobile と Windows 用に、複数のプロセスに同一のデータベースへのアクセスを許可するデータベース・エンジンが用意されています。

ランタイム・ライブラリは、*install-dir¥UltraLite¥Palm*、*install-dir¥UltraLite¥ce*、*install-dir¥UltraLite¥win32*、*install-dirx64* の各ディレクトリにあります。

Palm OS 用のランタイム・ライブラリ

Palm OS 上のアプリケーション用に用意されているライブラリは次のとおりです。これらのライブラリは、*install-dir¥UltraLite¥Palm¥68k¥lib¥cw* にあります。

- **ulrt.lib**　静的ライブラリ。
- **ulbase.lib**　個別の DLL (ダイナミック・リンク・ライブラリ) で提供できない追加の関数が含まれるライブラリ。Ultra Light 機能にアクセスする C/C++ アプリケーションには、このライブラリをリンクする必要があります。

Windows Mobile のランタイム・ライブラリ

Windows Mobile ライブラリは、*install-dir¥UltraLite¥ce¥arm.50¥Lib* ディレクトリにあります。

Windows Mobile 用には次の動的ライブラリが用意されています。

- **ulbase.lib**　個別の DLL (ダイナミック・リンク・ライブラリ) で提供できない追加の関数が含まれるライブラリ。Ultra Light 機能にアクセスする C/C++ アプリケーションには、このライブラリをリンクする必要があります。
- **ulrt.lib**　このライブラリをリンクする場合は、次のコンパイル・オプションを指定してください。

  ```
  /DUNICODE
  ```

- **ulrtc.lib**　Unicode 文字セットの静的ライブラリは、複数のプロセスに単一の Ultra Light データベースへのアクセスを可能にする Ultra Light エンジンとともに使用します。

  このライブラリをリンクする場合は、次のコンパイル・オプションを指定してください。

  ```
  /DUNICODE
  ```

Windows デスクトップのランタイム・ライブラリ

install-dir¥UltraLite¥win32¥386¥Lib¥vs8 および *install-dir¥UltraLite¥x64¥Lib¥vs8* ディレクトリには、サポートされている Windows デスクトップ・オペレーティング・システム用のライブラリが含まれています。用意されているライブラリは次のとおりです。

- **ulbase.lib**　個別の DLL (ダイナミック・リンク・ライブラリ) で提供できない関数が含まれるライブラリ。Ultra Light 機能にアクセスする C/C++ アプリケーションには、このライブラリをリンクする必要があります。

- **ulrt11.dll**　ANSI 文字セットのダイナミック・リンク・ライブラリ。このライブラリを使用するには、アプリケーションをインポート・ライブラリ *ulimp.lib* にリンクします。

  このライブラリをリンクする場合は、次のコンパイル・オプションを指定してください。

  ```
  /DUL_USE_DLL
  ```

Embedded SQL を使用したアプリケーションの開発

目次

この項では、Embedded SQL Ultra Light アプリケーション用のデータベース・アクセス・コードの記述方法について説明します。

Ultra Light C/C++ 開発の概要については、「Ultra Light for C/C++ の開発者」 1 ページを参照してください。

Embedded SQL のリファレンス情報については、「Embedded SQL API リファレンス」 265 ページを参照してください。

SQL プリプロセッサの詳細については、「Ultra Light SQL プリプロセッサ・ユーティリティ (sqlpp)」 『Ultra Light データベース管理とリファレンス』を参照してください。

Embedded SQL の例

Embedded SQL は、C/C++ プログラム・コードと擬似コードの組み合わせの環境です。従来の C/C++ コードの間に存在する擬似コードは、SQL 文のサブセットです。プリプロセッサは、embedded SQL 文を、アプリケーションを作成するためにコンパイルされる実際のコードの一部である関数呼び出しに変換します。

Embedded SQL プログラムの非常に簡単な例を次に示します。従業員 195 の姓を変更して Ultra Light データベース・レコードを更新します。

```
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;
main( )
{
  db_init( &sqlca );
  EXEC SQL WHENEVER SQLERROR GOTO error;
  EXEC SQL CONNECT "DBA" IDENTIFIED BY "sql";
  EXEC SQL UPDATE employee
    SET emp_lname = 'Johnson'
    WHERE emp_id = 195;
  EXEC SQL COMMIT;
  EXEC SQL DISCONNECT;
  db_fini( &sqlca );
  return( 0 );
  error:
    printf( "update unsuccessful: sqlcode = %ld¥n",
      sqlca.sqlcode );
    return( -1 );
}
```

この例は実際に使用するプログラムと比べてかなり簡単な内容ですが、Embedded SQL アプリケーションにおける次のような共通事項が示されています。

- 各 SQL 文には、キーワード EXEC SQL のプレフィクスが付いている。
- 各 SQL 文は、セミコロンで終わる。
- 標準の SQL では実装されていない Embedded SQL 文がある。INCLUDE SQLCA 文はその一例です。
- Embedded SQL では、SQL 文のほかに、特定のタスクを実行するためにライブラリ関数が提供される。関数 db_init と db_fini は、ライブラリ関数呼び出しの 2 例です。

初期化

前述のサンプル・コードでは、Ultra Light データベースのデータを操作する前に含める必要のある初期化文が示されています。

1. 次のコマンドを使用して、SQLCA (SQL Communication Area) を定義します。

   ```
   EXEC SQL INCLUDE SQLCA;
   ```

   この定義は、最初の Embedded SQL 文でなければならないため、通常はインクルード・リストの最後に記述します。

   アプリケーションに複数の *.sqc* ファイルがある場合は、各ファイルにこの行を含めます。

2. 最初のデータベース処理として、db_init という Embedded SQL ライブラリ関数を呼び出す必要があります。この関数は、Ultra Light ランタイム・ライブラリを初期化します。この呼び出しの前に実行できるのは、Embedded SQL の定義文だけです。

   「db_init 関数」 268 ページを参照してください。

3. Ultra Light データベースに接続するには、SQL CONNECT 文を使用する必要があります。

終了の準備

前述のサンプル・コードでは、終了の準備に必要となる呼び出しの順序も示しています。

1. 未処理の変更をコミットまたはロールバックします。
2. データベースを切断します。
3. db_fini というライブラリ関数を呼び出して、SQL の作業を終了します。

終了時に、コミットされていないデータベースの変更は、すべて自動的にロールバックされます。

エラー処理

この例では、SQL と C コード間の対話はまったくありません。C コードはプログラムのフロー制御だけを行います。WHENEVER 文はエラー・チェックに使用されています。エラー・アクション (この例では GOTO) は、いずれかの SQL 文がエラーになると実行されます。

Embedded SQL プログラムの構造

Embedded SQL 文は、必ず、EXEC SQL で始まり、セミコロンで終わります。ESQL 文の途中に、通常の C 言語のコメントを記述できます。

Embedded SQL を使用する C プログラムでは、ソース・ファイル内のどの Embedded SQL 文よりも前に、必ず次の文を置きます。

```
EXEC SQL INCLUDE SQLCA;
```

プログラムで実行される最初の Embedded SQL 文は、CONNECT 文である必要があります。CONNECT 文は、Ultra Light データベースへの接続を確立するために使用される接続パラメータを指定します。

Embedded SQL コマンドには C プログラム・コードを生成しないものや、データベースとのやりとりをしないものもあります。このようなコマンドは、CONNECT 文の前に記述できます。よく使われるのは、INCLUDE 文と、エラー処理を指定する WHENEVER 文です。

SQLCA (SQL Communications Area) の初期化

SQLCA (SQL Communications Area) とは、アプリケーションとデータベースの間で、統計情報とエラーをやりとりするのに使用されるメモリ領域です。SQLCA は、アプリケーションとデータベース間の通信リンクのハンドルとして使用されます。データベースとやりとりするデータベース・ライブラリ関数には、SQLCA が明示的に渡されます。また、Embedded SQL 文でも必ず暗黙のうちに渡されます。

生成コードには、SQLCA グローバル変数が 1 つ定義されています。プリプロセッサは、このグローバル SQLCA 変数の外部参照を生成します。外部参照の名前は sqlca、型は SQLCA です。実際のグローバル変数は、インポート・ライブラリ内で宣言されています。

SQLCA 型は、ヘッダ・ファイル *install-dir¥SDK¥Include¥sqlca.h* に定義されています。

アプリケーションでデータベースを操作するには、SQLCA (EXEC SQL INCLUDE SQLCA;) を宣言した後、db_init を呼び出して SQLCA を渡すことによって SQL Communication Area を初期化する必要があります。

```
db_init( &sqlca );
```

SQLCA にはエラー・コードが入る

SQLCA を参照すると、特定のエラー・コードの検査ができます。データベースへの要求がエラーになると、フィールド sqlcode にエラー・コードが入ります。sqlcode などの SQLCA のフィールドを参照するために、マクロが定義されています。

SQLCA のフィールド

SQLCA には、次のフィールドがあります。

- **sqlcaid**　SQLCA 構造体の ID として文字列 SQLCA が格納される 8 バイトの文字フィールド。このフィールドはデバッグ時にメモリの中身を見るときに役立ちます。
- **sqlcabc**　long integer。SQLCA 構造体の長さ (バイト単位) が入ります。
- **sqlcode**　long integer。データベースが検出した要求エラーのエラー・コードが入ります。エラー・コードの定義はヘッダ・ファイル *install-dir¥SDK¥Include¥sqlerr.h* にあります。エラー・コードは、0 (ゼロ) は成功、正の値は警告、負の値はエラーを示します。

  SQLCODE マクロを使用してこのフィールドに直接アクセスできます。

  エラー・コードのリストについては、「SQL Anywhere のエラー・メッセージ」 『エラー・メッセージ』を参照してください。
- **sqlerrml**　sqlerrmc フィールドの情報の長さ。

  Ultra Light アプリケーションでは、このフィールドは使用されません。
- **sqlerrmc**　エラー・メッセージに挿入する 1 つ以上の文字列。エラー・メッセージにプレースホルダ文字列 (*%1*) があると、このフィールドの文字列と置換されます。

  Ultra Light アプリケーションでは、このフィールドは使用されません。

- **sqlerrp**　予約。
- **sqlerrd**　long integer の汎用配列。
- **sqlwarn**　予約。

  Ultra Light アプリケーションでは、このフィールドは使用されません。
- **sqlstate**　SQLSTATE ステータス値。

  Ultra Light アプリケーションでは、このフィールドは使用されません。

データベースへの接続

Embedded SQL アプリケーションから Ultra Light データベースに接続するには、SQLCA を初期化した後、コードに EXEC SQL CONNECT 文を指定します。

CONNECT 文の形式は次のとおりです。

EXEC SQL CONNECT USING
'uid=*user-name***;pwd=***password***;dbf=***database-filename***';**

接続文字列 （一重引用符で囲まれている） には、追加のデータベース接続パラメータが含まれることがあります。

データベース接続パラメータの詳細については、「Ultra Light 接続パラメータ」 『Ultra Light データベース管理とリファレンス』を参照してください。

CONNECT 文の詳細については、「CONNECT 文 [ESQL] [Interactive SQL]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

複数の接続の管理

アプリケーションの中で複数のデータベース接続を使用する場合、複数の SQLCA を使用することもできれば、1 つの SQLCA で複数の接続を管理することもできます。

複数の SQLCA を使用するには、次の手順に従います。

- **複数の SQLCA の管理**

1. プログラムで使用する各 SQLCA は **db_init** を呼び出して初期化し、最後に **db_fini** を呼び出してクリーンアップします。

   「db_init 関数」 268 ページを参照してください。

2. Embedded SQL 文の SET SQLCA を使用して、SQL プリプロセッサにデータベース要求で特定の SQLCA を使用することを伝えます。通常は、次のような文をプログラムの先頭かヘッダ・ファイルに置いて、SQLCA 参照がタスク独自のデータを指すようにします。

   ```
   EXEC SQL SET SQLCA 'task_data->sqlca';
   ```

   この文はコードをまったく生成しないので、パフォーマンスに影響を与えません。この文はプリプロセッサ内部の状態を変更して、指定の文字列で SQLCA を参照するようにします。

   SQLCA の作成については、「SET SQLCA 文 [ESQL]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

1 つの SQLCA を使用するには、次の手順に従います。

複数の SQLCA を使用する代わりに、1 つの SQLCA で、データベースへの複数の接続を管理できます。

各 SQLCA はアクティブな接続、つまり現在の接続を持ちますが、その接続は変更が可能です。コマンドを実行する前に、SET CONNECTION 文でコマンドの実行対象となる接続を指定します。

「SET CONNECTION statement [Interactive SQL] [ESQL]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

ホスト変数の使用

Embedded SQL アプリケーションでは、ホスト変数を使用してデータベースと値をやり取りします。ホスト変数とは、宣言セクションにおいて SQL プリプロセッサが認識する C 変数です。

ホスト変数の宣言

宣言セクション内にホスト変数を配置して、ホスト変数を定義します。通常の C 変数宣言を BEGIN DECLARE SECTION 文と END DECLARE SECTION 文で囲むことで、ホスト変数を宣言します。

ホスト変数を SQL 文で使用するときは、変数名にコロン (:) をプレフィクスとして付けなければなりません。これは、SQL プリプロセッサが、(宣言済みの) ホスト変数が参照されていることを認識しており、SQL 文の中で他の識別子とホスト変数を区別するためです。

ホスト変数は、どの SQL 文でも値定数の代わりに使用できます。データベース・サーバがこのコマンドを実行すると、ホスト変数の値がホスト変数から読み込まれたり、逆にホスト変数に書き込まれたりします。ホスト変数をテーブル名やカラム名の代わりに使用することはできません。

SQL プリプロセッサは、宣言セクションの外では C 言語コードをスキャンしません。変数の初期化は宣言セクション内で行うことも可能ですが、**typedef** 型と構造体は使用できません。

INSERT コマンドでホスト変数を使用するサンプル・コードです。プログラム側で変数に値を設定してから、データベースに挿入しています。

```
/* Declare fields for personal data. */
EXEC SQL BEGIN DECLARE SECTION;
   long employee_number = 0;
   char employee_name[50];
   char employee_initials[8];
   char employee_phone[15];
EXEC SQL END DECLARE SECTION;
/* Fill variables with appropriate values. */
/* Insert a row in the database. */
EXEC SQL INSERT INTO Employee
   VALUES (:employee_number, :employee_name,
      :employee_initials, :employee_phone );
```

Embedded SQL のデータ型

プログラムとデータベース・サーバ間で情報を転送するには、それぞれのデータ項目についてデータ型を設定します。ホスト変数は、サポートされる任意のデータ型について作成できます。

ホスト変数として使用できる C のデータ型は非常に限られています。また、ホスト変数の型には、対応する C の型がないものもあります。

sqlca.h ヘッダ・ファイルで定義されたマクロは、VARCHAR、FIXCHAR、BINARY、DECIMAL、または SQLDATETIME 型のホスト変数を宣言するのに使用できます。これらのマクロは次のように使用します。


```
EXEC SQL BEGIN DECLARE SECTION;
  DECL_VARCHAR( 10 ) v_varchar;
  DECL_FIXCHAR( 10 ) v_fixchar;
  DECL_BINARY( 4000 ) v_binary;
  DECL_DECIMAL( 10, 2 ) v_packed_decimal;
  DECL_DATETIME v_datetime;
EXEC SQL END DECLARE SECTION;
```

プリプロセッサは宣言セクション内のこれらのマクロを認識し、変数を適切な型として処理します。

次のデータ型が、Embedded SQL プログラミング・インタフェースでサポートされます。

- **16 ビット符号付き整数**

  ```
  short int I;
  unsigned short int I;
  ```

- **32 ビット符号付き整数**

  ```
  long int I;
  unsigned long int I;
  ```

- **4 バイト浮動小数点数**

  ```
  float f;
  ```

- **8 バイト浮動小数点数**

  ```
  double d;
  ```

- **パック 10 進数**

  ```
  DECL_DECIMAL(p,s)
  typedef struct TYPE_DECIMAL {
    char array[1];
  } TYPE_DECIMAL;
  ```

- **ブランクが埋め込まれ、NULL で終了された文字列**

  ```
  char a[n]; /* n > 1 */
  char *a; /* n = 2049 */
  ```

  C 言語の配列には NULL ターミネータが必要であるため、char a[n] データ型は、CHAR(n - 1) SQL データ型にマッピングされます。SQL データ型は、n−1 文字まで保持できます。

char へのポインタ、WCHAR、TCHAR

SQL プリプロセッサでは、「char のポインタ」は 2049 バイトの文字配列を指しており、この文字配列が 2048 文字と NULL ターミネータを十分に保持できるとみなされます。つまり、char* データ型は、CHAR(2048) SQL 型にマッピングされます。この制限を超えると、アプリケーションによるメモリ破損が発生する場合があります。

16 ビット・コンパイラの場合、単純に 2049 バイトを確保するとプログラムのスタック・オーバフローを引き起こすこともあります。この問題を避けるため、宣言した配列を必要に応じて関数のパラメータとして使用し、SQL プリプロセッサに配列のサイズを通知するようにしてください。WCHAR と TCHAR も char と同じように機能します。

- **NULL で終了された Unicode 、またはワイド文字列**　文字ごとに 2 バイトの領域を占有するため、Unicode 文字を含めることができます。

  ```
  WCHAR a[n]; /* n > 1 */
  ```

- **システムに依存し、NULL で終了された文字列**　Unicode を使用するシステム (Windows Mobile など) では、TCHAR の文字セットは WCHAR と同じです。それ以外の場合、TCHAR は char と同じです。TCHAR データ型は、どちらかのシステムで文字列を自動的にサポートするように設計されています。

  ```
  TCHAR a[n]; /* n > 1 */
  ```

- **ブランクが埋め込まれた固定長文字列**

  ```
  char a; /* n  = 1 */
  DECL_FIXCHAR(n) a; /* n >= 1 */
  ```

- **2 バイトの長さフィールドを持つ可変長文字列**　データベース・サーバに情報を渡す場合は、長さフィールドを設定します。データベース・サーバから情報をフェッチする場合は、サーバが長さフィールドを設定します (埋め込みは行われません)。

  ```
  DECL_VARCHAR(n) a; /* n >= 1 */
  typedef struct VARCHAR {
    unsigned short int len;
    TCHAR array[1];
  } VARCHAR;
  ```

- **2 バイトの長さフィールドを持つ可変長バイナリ・データ**　データベース・サーバに情報を渡す場合は、長さフィールドを設定します。データベース・サーバから情報をフェッチする場合は、サーバが長さフィールドを設定します。

  ```
  DECL_BINARY(n) a; /* n >= 1 */
  typedef struct BINARY {
    unsigned short int len;
    unsigned char array[1];
  } BINARY;
  ```

- **タイムスタンプの各部分に対応するフィールドを持つ SQLDATETIME 構造体**

  ```
  DECL_DATETIME a;
  typedef struct SQLDATETIME {
    unsigned short year; /* for example: 1999 */
    unsigned char month; /* 0-11 */
    unsigned char day_of_week; /* 0-6, 0 = Sunday */
    unsigned short day_of_year; /* 0-365 */
    unsigned char day; /* 1-31 */
    unsigned char hour; /* 0-23 */
    unsigned char minute; /* 0-59 */
    unsigned char second; /* 0-59 */
    unsigned long microsecond; /* 0-999999 */
  } SQLDATETIME;
  ```

  SQLDATETIME 構造体は、型が DATE、TIME、TIMESTAMP (または、いずれかの型に変換できるもの) のフィールドを取り出すのに使用されます。アプリケーションは、日付に関して独自のフォーマットで処理をすることがありますが、この構造体を使ってデータをフェッチすると、以後の操作が簡単になります。この構造体の中のデータをフェッチすると、プログラマはこのデータを簡単に操作できます。また、型が DATE、TIME、TIMESTAMP のフィールドは、文字型であれば、どの型でもフェッチと更新が可能です。

SQLDATETIME 構造体を介してデータベースに日付、時刻、またはタイムスタンプを入力しようとすると、day_of_year と day_of_week メンバは無視されます。

詳細については、「データベース・オプション」『SQL Anywhere サーバ - データベース管理』の date_format、time_format、timestamp_format、date_order の各データベース・オプションを参照してください。

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

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

32 KB を超えるデータには、DECL_LONGVARCHAR 構造体を使用できます。データは、全体を一度にフェッチする方法と、GET DATA 文を使用して分割してフェッチする方法があります。また、サーバに対しても、全体を一度に送信する方法と、SET 文を使用してデータベース変数に追加することで分割して送信する方法があります。データは、NULL で終了しません。

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

- **DT_LONGBINARY** 長いバイナリ・データ。マクロによって、構造体が次のように定義されます。

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

32 KB を超えるデータには、DECL_LONGBINARY 構造体を使用できます。データは、全体を一度にフェッチする方法と、GET DATA 文を使用して分割してフェッチする方法があります。また、サーバに対しても、全体を一度に送信する方法と、SET 文を使用してデータベース変数に追加することで分割して送信する方法があります。

これらの構造体は *install-dir¥SDK¥Include¥sqlca.h* ファイルに定義されています。VARCHAR 型、BINARY 型、TYPE_DECIMAL 型は、データ格納領域が長さ 1 の文字配列のため、ホスト変数の宣言には向いていません。しかし、動的な変数の割り付けや他の変数の型変換を行うのには有効です。

データベースの DATE 型と TIME 型

データベースのさまざまな DATE 型と TIME 型に対応する、Embedded SQL インタフェースのデータ型はありません。これらの型は、SQLDATETIME 構造体または文字列を使用してフェッチと更新を行います。

データベースの LONG VARCHAR 型と LONG BINARY 型に対応する、Embedded SQL インタフェースのデータ型はありません。

ホスト変数の使用法

ホスト変数は次の場合に使用できます。

- SELECT、INSERT、UPDATE、DELETE 文で数値定数または文字列定数を記述できる場所。
- SELECT または FETCH 文の INTO 句。
- CONNECT、DISCONNECT、SET CONNECT 文では、ユーザ ID、パスワード、接続名、データベース名の代わりにホスト変数を使用できる。

ホスト変数は、テーブル名、カラム名の代わりには**使用できません**。

ホスト変数のスコープ

ホスト変数の宣言セクションは、C 変数を宣言できる通常の場所であれば、C の関数のパラメータの宣言セクションも含め、どこにでも記述できます。C 変数は通常のスコープを持っています (定義されたブロック内で使用可能)。ただし、SQL プリプロセッサは C コードをスキャンしないため、C ブロックを重視しません。

プリプロセッサはすべてのホスト変数をグローバルとみなす

SQL プリプロセッサから見ると、ホスト変数はその宣言に従って、ソース・モジュールに対してグローバルに認識されています。2 つのホスト変数が同じ名前を持つことはできません。この規則の例外として、2 つのホスト変数が同じ型 (必要な長さを含む) の場合、同じ名前を持つことができます。

ホスト変数ごとにユニークな名前を付けることが最善の方法であるといえます。

例

SQL プリプロセッサは C コードを解析できません。そのため、ホスト変数がどこで宣言されたかにかかわらず、宣言に従ってグローバルに認識されているとみなします。

```
// Example demonstrating poor coding
EXEC SQL BEGIN DECLARE SECTION;
  long emp_id;
EXEC SQL END DECLARE SECTION;
long getManagerID( void )
{
  EXEC SQL BEGIN DECLARE SECTION;
    long manager_id = 0;
  EXEC SQL END DECLARE SECTION;
  EXEC SQL SELECT manager_id
        INTO :manager_id
        FROM employee
        WHERE emp_number = :emp_id;
  return( manager_number );
}
void setManagerID( long manager_id )
{
  EXEC SQL UPDATE employee
        SET manager_number = :manager_id
        WHERE emp_number = :emp_id;
}
```


このコードは機能はするものの、*setManagerID* 内の文を処理するとき、SQL プリプロセッサが *getManagerID* 内の宣言に依存しているので、わかりにくくなっています。このコードを次のように書き換えます。

```
// Rewritten example
#if 0
  // Declarations for the SQL preprocessor
  EXEC SQL BEGIN DECLARE SECTION;
    long emp_id;
    long manager_id;
  EXEC SQL END DECLARE SECTION;
#endif
long getManagerID( long emp_id )
{
  long manager_id = 0;
  EXEC SQL SELECT manager_id
        INTO :manager_id
        FROM employee
        WHERE emp_number = :emp_id;
  return( manager_number );
}
void setManagerID( long emp_id, long manager_id )
{
  EXEC SQL UPDATE employee
        SET manager_number = :manager_id
        WHERE emp_number = :emp_id;
}
```

SQL プリプロセッサは、これらのディレクティブを無視するため、#if ディレクティブにあるホスト変数の宣言を調べます。それに対し、プロシージャ内の宣言は、DECLARE SECTION 内にないため、無視されます。これとは逆に、C コンパイラは #if ディレクティブ内の宣言を無視し、プロシージャ内の宣言を使用します。

これらの宣言が機能するのは、同じ名前を持つ変数が同じ型を持つように宣言されているときだけです。

ホスト変数で式を使用する

SQL プリプロセッサはポインタや参照式を認識しないため、ホスト変数は単純な名前でなければなりません。たとえば、次の文では、SQL プリプロセッサがドット演算子を理解しないため、**正しく機能しません**。同じ構文が、SQL では違う意味を持ちます。

```
// Incorrect statement:
EXEC SQL SELECT LAST sales_id INTO :mystruct.mymember;
```

上記の構文は使用できませんが、次の方法で式を使用できます。

- SQL 宣言セクションを #if 0 プリプロセッサ・ディレクティブで囲む。SQL プリプロセッサはプリプロセッサ・ディレクティブを無視するため、この宣言を読み込み、残りのモジュールで使用します。
- マクロをホスト変数と同じ名前で定義する。#if ディレクティブがあるため C コンパイラからは SQL 宣言セクションが見えず、競合が起きません。マクロがホスト変数と同じ型であると評価されることを確認してください。

次のコードは、SQL プリプロセッサから *host_value* 式を隠す方法を示しています。

```
#include <sqlerr.h>
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;
typedef struct my_struct {
   long     host_field;
} my_struct;
#if 0
   // Because it ignores #if preprocessing directives,
   // SQLPP reads the following declaration.
   EXEC SQL BEGIN DECLARE SECTION;
      long    host_value;
   EXEC SQL END DECLARE SECTION;
#endif
// Make C/C++ recognize the 'host_value' identifier
// as a macro that expands to a struct field.
#define host_value my_s.host_field
```

SQLPP プロセッサは条件付きコンパイルのディレクティブを無視するため、*host_value* は *long* ホスト変数として扱われ、その後ホスト変数として使用されたときにその名前を生成します。C/C++ コンパイラはこの生成されたファイルを処理し、このように名前が使用されている場合には *my_s.host_field* に置き換えます。

前述の宣言を使用した場合、次のようにして *host_field* にアクセスできます。

```
void main( void )
{
   my_struct      my_s;
   db_init( &sqlca );
   EXEC SQL CONNECT "DBA" IDENTIFIED BY "SQL";
   EXEC SQL DECLARE my_table_cursor CURSOR FOR
      SELECT int_col FROM my_table order by int_col;
   EXEC SQL OPEN my_table_cursor;
   for( ; ; ) {
      // :host_value references my_s.host_field
      EXEC SQL FETCH NEXT AllRows INTO :host_value;
      if( SQLCODE == SQLE_NOTFOUND ) {
         break;
      }
      printf( "%ld¥n", my_s.host_field );
   }
   EXEC SQL CLOSE my_table_cursor;
   EXEC SQL DISCONNECT;
   db_fini( &sqlca );
}
```

同じ方法で、他の lvalue をホスト変数として使用できます。

- ポインタの間接参照

  ```
  *ptr
  p_struct->ptr
  (*pp_struct)->ptr
  ```

- 配列参照

  ```
  my_array[ I ]
  ```

- 任意の複雑な lvalue

C++ でのホスト変数の使用

ホスト変数を C++ クラスで使用する場合も、同じような状況が発生します。一般に、クラスを別のヘッダ・ファイルで宣言すると便利です。たとえば、このヘッダ・ファイルには、次のような *my_class* の宣言が含まれています。

```
typedef short a_bool;
#define  TRUE  ((a_bool)(1==1))
#define  FALSE ((a_bool)(0==1))
public class {
  long  host_member;
  my_class();    // Constructor
  ~my_class();      // Destructor
  a_bool FetchNextRow( void );
    // Fetch the next row into host_member
} my_class;
```

この例では、各メソッドは、Embedded SQL ソース・ファイルに実装されます。簡単な変数だけがホスト変数として使用されます。あるクラスのデータ・メンバへのアクセスに、前の項で説明した方法を使用できます。

```
EXEC SQL INCLUDE SQLCA;
#include "my_class.hpp"
#if 0
  // Because it ignores #if preprocessing directives,
  // SQLPP reads the following declaration.
  EXEC SQL BEGIN DECLARE SECTION;
    long  this_host_member;
  EXEC SQL END DECLARE SECTION;
#endif
// Macro used by the C++ compiler only.
#define this_host_member this->host_member
my_class::my_class()
{
  EXEC SQL DECLARE my_table_cursor CURSOR FOR
    SELECT int_col FROM my_table order by int_col;
  EXEC SQL OPEN my_table_cursor;
}
my_class::~my_class()
{
  EXEC SQL CLOSE my_table_cursor;
}
a_bool my_class::FetchNextRow( void )
{
  // :this_host_member references this->host_member
  EXEC SQL FETCH NEXT AllRows INTO :this_host_member;
  return( SQLCODE != SQLE_NOTFOUND );
}
void main( void )
{
  db_init( &sqlca );
  EXEC SQL CONNECT "DBA" IDENTIFIED BY "SQL";
  {
    my_class mc; // Created after connecting.
    while( mc.FetchNextRow() ) {
      printf( "%ld¥n", mc.host_member );
    }
  }
  EXEC SQL DISCONNECT;
  db_fini( &sqlca );
}
```

この例では、SQL プリプロセッサに this_host_member を宣言しますが、マクロで C++ がこの宣言を this->host_member に変換します。この変換が行われないと、プリプロセッサはこの変数の型を知ることができません。C/C++ コンパイラは通常重複した宣言を黙認しません。#if ディレクティブは、2 番目の宣言をコンパイラから隠しますが、SQL プリプロセッサからは見える状態を保ちます。

複数の宣言は便利ですが、各宣言が同じ型に同じ変数名を割り当てるようにしてください。プリプロセッサは、C 言語を完全に解析することができないため、宣言に従ってホスト変数がグローバルに認識されているとみなします。

インジケータ変数の使用

インジケータ変数とは、特定のホスト変数に関する補足的な情報を保持する C 変数のことです。ホスト変数は、データのやりとりをするときに使用できます。NULL 値を扱うには、インジケータ変数を使用します。

インジケータ変数は、short int 型のホスト変数です。NULL 値を検出したり指定したりするため、SQL 文では通常のホスト変数の直後にインジケータ変数を記述します。

例

たとえば、次の INSERT 文では、**:ind_phone** がインジケータ変数です。

```
EXEC SQL INSERT INTO Employee
  VALUES (:employee_number, :employee_name,
  :employee_initials, :employee_phone:ind_phone );
```

インジケータ変数の値

次の表は、インジケータ変数の使用法をまとめたものです。

インジケータの値	データベースに渡す値	データベースから受け取る値
0	ホスト変数値	NULL でない値をフェッチした値
-1	NULL 値	NULL 値をフェッチした値

NULL を扱うためのインジケータ変数

SQL での NULL を同じ名前の C 言語の定数と混同しないでください。SQL 言語では、NULL は属性が不明であるか情報が適切でないかのいずれかを表します。C 言語の定数は、ポイント先がメモリのロケーションではないポインタ値を表します。

SQL Anywhere のマニュアルで使用されている NULL の場合は、上記のような SQL データベースを指します。C 言語の定数を指す場合は、null ポインタ (小文字) のように表記されます。

NULL は、カラムに定義されるどのデータ型の値とも同じではありません。NULL 値をデータベースに渡したり、結果に NULL を受取ったりするためには、通常のホスト変数の他に何か特別なものが必要です。このために使用されるのが、インジケータ変数です。

NULL を挿入する場合のインジケータ変数

INSERT 文は、次のようにインジケータ変数を含むことができます。

```
EXEC SQL BEGIN DECLARE SECTION;
short int employee_number;
char employee_name[50];
char employee_initials[6];
char employee_phone[15];
short int ind_phone;
EXEC SQL END DECLARE SECTION;
/* set values of empnum, empname,
   initials, and homephone */
if( /* phone number is known */ ) {
   ind_phone =  0;
} else {
   ind_phone = -1; /* NULL */
}
EXEC SQL INSERT INTO Employee
   VALUES (:employee_number, :employee_name,
   :employee_initials, :employee_phone:ind_phone );
```

インジケータ変数の値が -1 の場合は、NULL が書き込まれます。値が 0 の場合は、employee_phone の実際の値が書き込まれます。

NULL をフェッチする場合のインジケータ変数

インジケータ変数は、データをデータベースから受け取るときにも使用されます。この場合は、NULL 値がフェッチされた (インジケータが負) ことを示すために使用されます。NULL 値がデータベースからフェッチされたときにインジケータ変数が渡されない場合は、SQLE_NO_INDICATOR エラーが発生します。

SQLCA 構造体で返されるエラーと警告の詳細については、「SQLCA (SQL Communications Area) の初期化」 34 ページを参照してください。

データのフェッチ

ESQL でデータをフェッチするには SELECT 文を使用します。これには 2 つの場合があります。

1. SELECT 文がローをまったく返さないか、1 つだけローを返す場合。
2. SELECT 文が複数のローを返す場合。

1 つのローのフェッチ

「シングル・ロー・クエリ」がデータベースから取り出すローの数は多くても 1 つだけです。シングル・ロー・クエリの SELECT 文では、INTO 句が select リストの後、FROM 句の前にきます。INTO 句には、select リストの各項目の値を受け取るホスト変数のリストを指定します。select リスト項目と同数のホスト変数を指定してください。ホスト変数と一緒に、結果が NULL であることを示すインジケータ変数も指定できます。

SELECT 文が実行されると、データベース・サーバは結果を取り出して、ホスト変数に格納します。

- クエリが複数のローを返すと、データベース・サーバは SQLE_TOO_MANY_RECORDS エラーを返す。
- クエリがローを返さなかった場合、警告 SQLE_NOTFOUND が返される。

SQLCA 構造体で返されるエラーと警告の詳細については、「SQLCA (SQL Communications Area) の初期化」 34 ページを参照してください。

例

たとえば、次のコードは employee テーブルから正しくローをフェッチできた場合は 1 を、ローが存在しない場合は 0 を、エラーが発生した場合は -1 を返します。

```
EXEC SQL BEGIN DECLARE SECTION;
  long int    emp_id;
  char        name[41];
  char        sex;
  char        birthdate[15];
  short int   ind_birthdate;
EXEC SQL END DECLARE SECTION;
int find_employee( long employee )
{
  emp_id = employee;
  EXEC SQL SELECT emp_fname || ' ' || emp_lname,
       sex, birth_date
    INTO :name, :sex, birthdate:ind_birthdate
    FROM "DBA".employee
    WHERE emp_id = :emp_id;
  if( SQLCODE == SQLE_NOTFOUND ) {
    return( 0 ); /* employee not found */
  } else if( SQLCODE < 0 ) {
    return( -1 ); /* error */
  } else {
    return( 1 ); /* found */
```



```
    }
}
```

複数ローのフェッチ

カーソルは、結果セットに複数のローがあるクエリからローを取り出すために使用されます。カーソルは、SQL クエリ結果セットのためのハンドルつまり識別子であり、結果セット内の位置を示します。

カーソルの概要については、「カーソルを使用した操作」『SQL Anywhere サーバ - プログラミング』を参照してください。

♦ Embedded SQL でカーソルを管理するには、次の手順に従います。

1. DECLARE 文を使って、特定の SELECT 文のためのカーソルを宣言します。
2. OPEN 文を使って、カーソルを開きます。
3. FETCH 文を使用して、一度に 1 つのローをカーソルから取り出します。
   - 警告 SQLE_NOTFOUND が返されるまで、ローをフェッチします。エラー・コードと警告のコードは、SQL Communications Area 構造体で定義される変数 SQLCODE で返されます。
4. CLOSE 文を使ってカーソルを閉じます。

Ultra Light アプリケーションのカーソルは、常に WITH HOLD オプションを使用して開かれます。自動的に閉じられることはありません。CLOSE 文を使用して、各カーソルを明示的に閉じます。

次は、簡単なカーソル使用の例です。

```
void print_employees( void )
{
  int status;
  EXEC SQL BEGIN DECLARE SECTION;
  char name[50];
  char sex;
  char birthdate[15];
  short int ind_birthdate;
  EXEC SQL END DECLARE SECTION;
  /* 1. Declare the cursor. */
  EXEC SQL DECLARE C1 CURSOR FOR
    SELECT emp_fname || ' ' || emp_lname,
           sex, birth_date
    FROM "DBA".employee
    ORDER BY emp_fname, emp_lname;
  /* 2. Open the cursor. */
  EXEC SQL OPEN C1;
  /* 3. Fetch each row from the cursor. */
  for( ;; ) {
    EXEC SQL FETCH C1 INTO :name, :sex,
         :birthdate:ind_birthdate;
    if( SQLCODE == SQLE_NOTFOUND ) {
      break; /* no more rows */
    } else if( SQLCODE < 0 ) {
      break; /* the FETCH caused an error */
    }
```

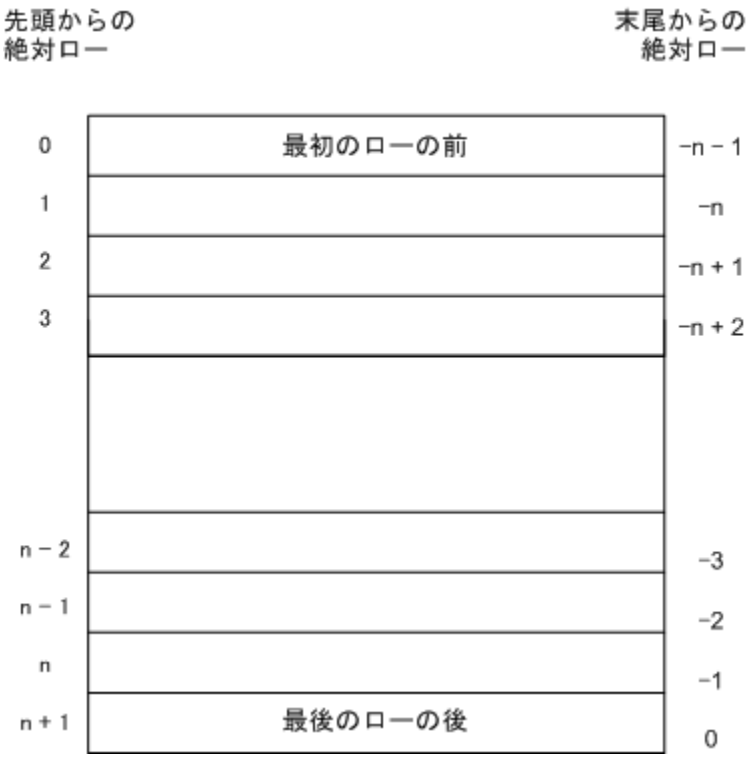
```
        if( ind_birthdate < 0 ) {
           strcpy( birthdate, "UNKNOWN" );
        }
        printf( "Name: %s Sex: %c Birthdate:
                %s¥n",name, sex, birthdate );
     }
     /* 4. Close the cursor. */
     EXEC SQL CLOSE C1;
  }
```

FETCH 文の詳細については、「FETCH 文 [ESQL] [SP]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

カーソル位置

カーソルは、次のいずれかの位置にあります。

- ローの上
- 最初のローの前
- 最後のローの後



カーソル内のローの順序

カーソル内のローの順序を制御するには、そのカーソルを定義する SELECT 文に ORDER BY 句を含めます。この句を省略すると、ローの順序が制御できなくなります。

明示的に順序を定義しない場合は、SQLE_NOTFOUND が返される前に一度だけ、フェッチごとに各ローが結果セットに返されます。

カーソルの再配置

カーソルを開くと、最初のローの前に置かれます。FETCH 文が自動的にカーソル位置を進めます。最後のローより後で FETCH 文を実行しようとすると、SQLE_NOTFOUND エラーが発生します。これは、ローの連続処理を完了するための信号として利用できます。

カーソルはクエリ結果の先頭または末尾を基準にした絶対位置に再配置できます。また、カーソルの現在位置を基準にした相対位置にも移動できます。カーソルの現在位置のローを更新または削除するために、特別な**位置付け**型の UPDATE 文と DELETE 文があります。先頭のローの前か、末尾のローの後にカーソルがある場合、SQLE_NOTFOUND エラーが返されます。

明示的な位置付けを行って予期しない結果が出るのを避けるには、そのカーソルを定義する SELECT 文に ORDER BY 句を含めます。

カーソルにローを挿入するには、PUT 文を使用します。

更新後のカーソル位置

開かれたカーソルがアクセスしている情報を変更した場合は、そのローをもう一度フェッチして表示するのが最適な方法です。単一のローの表示にカーソルが使用されている場合は、FETCH RELATIVE 0 が現在のローを再度フェッチします。現在のローが削除されていた場合は、次のローがカーソルからフェッチされます。ローがこれ以上ない場合は、SQLE_NOTFOUND が返されます。

テンポラリ・テーブルがカーソルに使用されている場合、基本となるテーブルに挿入されたローは、カーソルが閉じられて再び開かれるまでまったく表示されません。プログラマにとって、通常、SQL プリプロセッサが生成したコードを検査したり、テンポラリ・テーブルが使用される条件に精通していたりしないかぎり、SELECT 文にテンポラリ・テーブルが含まれているかどうかを検出するのは困難です。ORDER BY 句で使用されるカラムにインデックスを設定することによって、通常はテンポラリ・テーブルを回避できます。

テンポラリ・テーブルの詳細については、「クエリ処理におけるワーク・テーブルの使用 (All-rows 最適化ゴールの使用)」 『SQL Anywhere サーバ - SQL の使用法』を参照してください。

非テンポラリ・テーブルに対する挿入、更新、削除は、カーソル位置に影響を及ぼすことがあります。Ultra Light では、新しくローにデータが挿入されたり、ローが新しく削除されてデータがなくなったりしている場合には、その後の FETCH 操作に影響を及ぼします。これは、テンポラリ・テーブルが使用されていない場合、カーソル・ローを一度に 1 つだけ表示するためです。(一部の) ローが単一のテーブルから選択されている簡単な例では、挿入または更新されたローが SELECT 文の選択基準を満たす場合、そのローはカーソルの結果セットに表示されます。同様に、結果セットに表示されたローが新しく削除されると、そのローは結果セットに表示されなくなります。

ユーザの認証

Ultra Light データベースは、デフォルトのユーザ ID DBA とパスワード sql を使用して作成されるため、最初はこの初期ユーザとして接続します。新しいユーザは既存の接続から追加する必要があります。

ユーザ ID を変更することはできません。新しいユーザ ID を追加して、既存のユーザ ID を削除します。Ultra Light ではデータベースごとにユーザ ID が 4 つまで許可されます。

Palm OS において、ユーザが別のアプリケーションから元のアプリケーションに戻るたびにユーザ認証を行う必要がある場合は、PilotMain ルーチンを使用してユーザとパスワード情報のためのプロンプトを組み込んでください。

ユーザ認証の例

完全なサンプルは *samples-dir¥UltraLite¥esqlauth* ディレクトリにあります。次のコードは *samples-dir¥UltraLite¥esqlauth¥sample.sqc* の一部です。

```
//embedded SQL
app() {
  ...
/* Declare fields */
  EXEC SQL BEGIN DECLARE SECTION;
    char uid[31];
    char pwd[31];
  EXEC SQL END DECLARE SECTION;
  db_init( &sqlca );
  ...
  EXEC SQL CONNECT "DBA" IDENTIFIED BY "sql";
  if( SQLCODE == SQLE_NOERROR ) {
    printf("Enter new user ID and password¥n" );
    scanf( "%s %s", uid, pwd );
    ULGrantConnectTo( &sqlca,
      UL_TEXT( uid ), UL_TEXT( pwd ) );
    if( SQLCODE == SQLE_NOERROR ) {
      // new user added: remove DBA
      ULRevokeConnectFrom( &sqlca, UL_TEXT("DBA") );
    }
    EXEC SQL DISCONNECT;
  }
  // Prompt for password
   printf("Enter user ID and password¥n" );
   scanf( "%s %s", uid, pwd );
   EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
```

このコードでは、次のタスクを実行します。

1. db_init を呼び出してデータベースの機能を開始する。
2. デフォルトのユーザ ID とパスワードを使用して接続する。
3. 接続に成功したら、新しいユーザを追加する。
4. 新しいユーザが追加されたら、Ultra Light データベースから DBA を削除する。
5. 切断する。更新されたユーザ ID とパスワードがデータベースに追加される。
6. 更新されたユーザ ID とパスワードを使用して接続する。

次の項を参照してください。

- 「ULGrantConnectTo 関数」 288 ページ
- 「ULRevokeConnectFrom 関数」 293 ページ

データの暗号化

Ultra Light Embedded SQL を使用して、Ultra Light データベースを暗号化したり、難読化したりできます。

「データの暗号化」 54 ページを参照してください。

暗号化

Ultra Light データベースを (Sybase Central などを使用して) 作成する場合は、オプションで暗号化キーを指定できます。暗号化キーは、データベースの暗号化に使用されます。データベースが暗号化されると、その後のすべての接続で暗号化キーの指定が必要になります。指定されたキーが元の暗号化キーと照合され、キーが一致しないと接続は失敗します。

暗号化キーには簡単に推測できる値を選択しないでください。キーの長さは任意ですが、短いと推測されやすいため、一般的には長い方が適しています。数字、文字、特殊文字を組み合わせると、キーは推測されにくくなります。

キーにはセミコロンを含めないでください。キー自体を引用符で囲まないでください。

♦ 暗号化された Ultra Light データベースに接続するには、次の手順に従います。

1. EXEC SQL CONNECT 文に指定されている接続文字列で暗号化キーを指定します。

   暗号化キーは、key= 接続文字列パラメータの形で指定します。

   このキーは、データベースに接続するたびに指定する必要があります。キーを忘れた場合はデータベースにまったくアクセスできなくなります。

2. 間違ったキーを使用して暗号化されたデータベースを開こうとしてみてください。

   暗号化されたデータベースを開こうとして、間違ったキーが渡されると、db_init が ul_false を返し、SQLCODE -840 が設定されます。

暗号化キーの変更

データベースの暗号化キーは変更できます。既存のキーを使用してアプリケーションをデータベースに接続してから、変更を行ってください。

♦ Ultra Light データベースの暗号化キーを変更するには、次の手順に従います。

- 引数として新しいキーを指定して、ULChangeEncryptionKey 関数を呼び出します。

  この関数は、古いキーを使用してアプリケーションをデータベースに接続してから呼び出します。

  「ULChangeEncryptionKey 関数」 271 ページを参照してください。

難読化

♦ **Ultra Light データベースを難読化するには、次の手順に従います。**

● データベースの暗号化を使用する代わりに、データベースの難読化を指定するという方法があります。難読化とは、データベースのデータを簡単にマスキングすることで、低レベルのファイル検証ユーティリティを使用してデータベース内のデータが見られても内容がわからないようにすることです。難読化はデータベース作成オプションの 1 つで、データベースの作成時に指定する必要があります。

「Ultra Light で使用するデータベース作成パラメータの選択」『Ultra Light データベース管理とリファレンス』を参照してください。

アプリケーションへの同期の追加

多くの Ultra Light アプリケーションにとって、同期は重要な機能です。この項では、アプリケーションに同期の機能を追加する方法を説明します。

Ultra Light アプリケーションを統合データベースの最新状態と同期する論理は、アプリケーション自体にはありません。統合データベースに格納されている同期スクリプトは、Mobile Link サーバと Ultra Light ランタイム・ライブラリとともに、変更のアップロード時に変更をどのように処理するかを制御し、ダウンロードする変更はどれかを決定します。

概要

同期ごとの詳細は、同期パラメータのセットによって制御されます。これらのパラメータは、構造体に収集された後、関数呼び出しの引数として渡され、同期が行われます。このメソッドの概要は、どの開発モデルでも同じです。

♦ **アプリケーションに同期を追加するには、次の手順に従います。**

1. 同期パラメータが格納された構造体を初期化します。

   「同期パラメータの初期化」 56 ページを参照してください。

2. アプリケーションのパラメータ値を割り当てます。

   「Ultra Light 同期ストリームのネットワーク・プロトコルのオプション」 『Ultra Light データベース管理とリファレンス』を参照してください。

3. 同期関数を呼び出し、構造体またはオブジェクトを引数として指定します。

   「同期を呼び出す」 57 ページを参照してください。

同期するときに、コミットされていない変更がないことを確認してください。

同期パラメータ

ul_synch_info 構造体については、C/C++ コンポーネントの章で説明されています。ただし、この構造体のメンバは Embedded SQL 開発でも共通です。使用文字が ASCII 文字かワイド文字かに応じて、「ul_synch_info_a 構造体」 135 ページまたは「ul_synch_info_w2 構造体」 138 ページを参照してください。

同期パラメータの全般的な説明については、「Ultra Light の同期パラメータ」 『Ultra Light データベース管理とリファレンス』を参照してください。

同期パラメータの初期化

同期パラメータは、構造体に格納されます。

構造体のメンバは、初期化時に未定義です。構造体のメンバの初期値にパラメータを設定し、特別な関数を呼び出します。同期パラメータは、Ultra Light ヘッダ・ファイル *install-dir¥SDK¥Include¥ulglobal.h* で宣言された構造体で定義されます。

♦ **同期パラメータを初期化するには、次の手順に従います (Embedded SQL の場合)。**

● ULInitSynchInfo 関数を呼び出します。次に例を示します。

```
ul_synch_info synch_info;
ULInitSynchInfo( &synch_info );
```

同期パラメータの設定

次のコードは、TCP/IP の同期を開始します。Mobile Link ユーザ名は Betty Best、パスワードは TwentyFour、スクリプト・バージョンは default です。Mobile Link サーバはホスト・コンピュータ test.internal で実行されており、ポート 2439 を使用しています。

```
ul_synch_info synch_info;
ULInitSynchInfo( &synch_info );
synch_info.user_name = UL_TEXT("Betty Best");
synch_info.password = UL_TEXT("TwentyFour");
synch_info.version = UL_TEXT("default");
synch_info.stream = ULSocketStream();
synch_info.stream_parms =
  UL_TEXT("host=test.internal;port=2439");
ULSynchronize( &sqlca, &synch_info );
```

次のコードは、Palm Computing Platform 上のアプリケーション用です。これは、ユーザがアプリケーションを終了したときに呼び出されます。これによって HotSync 同期が実行されます。この場合の Mobile Link ユーザ名は 50、パスワードは空、スクリプト・バージョンは custdb です。HotSync コンジットは、TCP/IP を利用して Mobile Link サーバと通信します。Mobile Link サーバはコンジット (localhost) と同じコンピュータで実行され、デフォルトのポート (2439) を使用しています。

```
ul_synch_info synch_info;
ULInitSynchInfo( &synch_info );
synch_info.name = UL_TEXT("Betty Best");
synch_info.version = UL_TEXT("default");
synch_info.stream = ULConduitStream();
synch_info.stream_parms =
  UL_TEXT("stream=tcpip;host=localhost");
ULSetSynchInfo( &sqlca, &synch_info );
```

同期を呼び出す

同期を呼び出す方法は、ターゲット・プラットフォームと同期ストリームによって細かく異なります。

同期処理が機能するのは、Ultra Light アプリケーションを実行するデバイスが Mobile Link サーバと通信できる場合だけです。プラットフォームによっては、デバイスを、クレードルに置くかまたは適切なケーブルを使用してサーバ・コンピュータに接続して、物理的に接続する必要があります。同期が実行できない場合には、アプリケーションにエラー処理コードを追加する必要があります。

♦ **同期を呼び出すには、次の手順に従います (TCP/IP、TLS、HTTP、または HTTPS ストリームの場合)。**

- ULInitSynchInfo を呼び出して同期パラメータを初期化し、ULSynchronize を呼び出して同期を行います。

♦ **同期を呼び出すには、次の手順に従います (HotSync の場合)。**

- ULInitSynchInfo を呼び出して同期パラメータを初期化し、ULSetSynchInfo を呼び出して同期を管理してからアプリケーションを終了します。

  「ULSetSynchInfo 関数」 299 ページを参照してください。

同期呼び出しでは、その同期固有の情報が記述されたパラメータのセットを保持している構造体が必要です。使用される特定のパラメータは、ストリームによって異なります。

同期の前に変更をコミットする

Ultra Light データベースは、同期のときに変更をコミットしないでおくことはできません。Ultra Light データベースの同期をとろうとした時点で、コミットされていないトランザクションが接続にあると、同期は失敗し、例外がスローされ、SQLE_UNCOMMITTED_TRANSACTIONS エラーが設定されます。このエラー・コードは、Mobile Link サーバ・ログにも表示されます。

ダウンロード専用同期の詳細については、「Download Only 同期パラメータ」 『Ultra Light データベース管理とリファレンス』を参照してください。

アプリケーションへの初期データの追加

Ultra Light アプリケーションは、一般的に、使用前にデータが必要です。同期でアプリケーションにデータをダウンロードできます。アプリケーションが最初に実行されたとき、他のアクションが行われる前に必要なデータがすべてダウンロードされるように、アプリケーションに論理を追加できます。

パフォーマンスに関するヒント
アプリケーションを段階別に開発すると、エラーが発見しやすくなります。プロトタイプの開発中に、テストとデモンストレーションを目的としたデータを得るため、一時的に INSERT 文をアプリケーションで使用します。プロトタイプが正常に動作することを確認したら、一時的な INSERT 文を同期を実行するコードに置き換えます。

同期の開発に関するヒントについては、「Mobile Link 開発のヒント」 『Mobile Link - サーバ管理』を参照してください。

同期通信エラーの処理

次のサンプル・コードは、Embedded SQL アプリケーションから通信エラーを処理する方法を示しています。

```
if( psqlca->sqlcode == SQLE_COMMUNICATIONS_ERROR ) {
    printf( "  Stream error information:¥n"
            "    stream_error_code = %ld¥t(ss_error_code)¥n"
            "    error_string      = ¥"%s¥"¥n"
            "    system_error_code = %ld¥n",
    (long)info.stream_error.stream_error_code,
    info.stream_error.error_string,
    (long)info.stream_error.system_error_code );
```

SQLE_COMMUNICATIONS_ERROR は、通信エラーを表す一般的なエラー・コードです。特定のエラーに関する詳細な情報は、stream_error 同期パラメータのメンバを使用して、アプリケーションに渡されます。

Ultra Light のサイズを小さくするために、ランタイムによるレポートは、メッセージではなく数値で行われます。

同期のモニタとキャンセル

この項では、Ultra Light のアプリケーションからの同期をモニタしたりキャンセルしたりする方法について説明します。

同期のモニタ

- 同期構造体 (ul_synch_info) の observer メンバのコールバック関数の名前を指定します。
- 同期関数かメソッドを呼び出して同期を開始します。
- Ultra Light が、同期のステータスが変更するたびにコールバック関数を呼び出します。次の項では同期のステータスについて説明します。

次のコードは、Embedded SQL アプリケーションでこのタスクのシーケンスをどのように実装できるかを示しています。

```
ULInitSynchInfo( &info );
info.user_name = m_EmpIDStr;
...
//The info parameter of ULSynchronize() contains
// a pointer to the observer function
info.observer = ObserverFunc;
ULSynchronize( &sqlca, &info );
```

同期ステータス情報の処理

同期をモニタするコールバック関数は、ul_synch_status 構造体をパラメータとして取ります。詳細については、「ul_synch_status 構造体」 143 ページを参照してください。

ul_synch_status 構造体には次のようなメンバが含まれます。

```
struct ul_synch_status {
    struct {
        ul_u_long   bytes;
        ul_u_long   inserts;
        ul_u_long   updates;
        ul_u_long   deletes;
    }           sent;
    struct {
        ul_u_long   bytes;
        ul_u_long   inserts;
        ul_u_long   updates;
        ul_u_long   deletes;
    }           received;
    p_ul_synch_info  info;
    ul_synch_state   state;
    ul_u_short       db_tableCount;
    ul_u_short       table_id;
    char             table_name[];
    ul_wchar         table_name_w2[];
    ul_u_short       sync_table_count;
    ul_u_short       sync_table_index;
    ul_sync_state    state;
    ul_bool          stop;
    ul_u_short       flags;
    ul_void *        user_data;
    SQLCA *          sqlca;
}
```

- **sent.inserts**　これまでにアップロードされた挿入済みローの数。
- **sent.updates**　これまでにアップロードされた更新済みローの数。
- **sent.deletes**　これまでにアップロードされた削除済みローの数。
- **sent.bytes**　これまでにアップロードされたバイト数。
- **received.inserts**　これまでにダウンロードされた挿入済みローの数。
- **received.updates**　これまでにダウンロードされた更新済みローの数。
- **received.deletes**　これまでにダウンロードされた削除済みローの数。
- **received.bytes**　これまでにダウンロードされたバイト数。
- **info**　ul_synch_info 構造体へのポインタ。「ul_synch_info_a 構造体」 135 ページを参照してください。
- **db_tableCount**　データベース内のテーブルの数を返します。
- **table_id**　現在アップロードまたはダウンロードされているテーブルの番号 (1 から始まります)。同期されないテーブルがある場合には、この番号で値がスキップされることがあります。また、番号が必ず増加するとはかぎりません。
- **table_name[]**　現在のテーブルの名前。
- **table_name_w2[]**　現在のテーブルの名前 (ワイド文字バージョン)。このフィールドには、Windows (デスクトップまたは Mobile) 環境の場合のみ値が入ります。
- **sync_table_count**　同期中のテーブルの数を返します。

- **sync_table_index**　アップロードまたはダウンロードされているテーブルの番号。1 から始まり **sync_table_count** の値で終わります。同期されていないテーブルがある場合には、この番号で値がスキップされることがあります。
- **state**　以下のステータスのいずれかを表します。
  - **UL_SYNCH_STATE_STARTING**　同期処理はまだ行われていません。
  - **UL_SYNCH_STATE_CONNECTING**　同期ストリームは構築されていますが、まだ開かれていません。
  - **UL_SYNCH_STATE_SENDING_HEADER**　同期ストリームが開かれ、ヘッダが送信されようとしています。
  - **UL_SYNCH_STATE_SENDING_TABLE**　テーブルが送信されています。
  - **UL_SYNCH_STATE_SENDING_DATA**　スキーマ情報またはデータが送信されています。
  - **UL_SYNCH_STATE_FINISHING_UPLOAD**　アップロード処理が完了し、コミットが実行されています。
  - **UL_SYNCH_STATE_RECEIVING_UPLOAD_ACK**　アップロード完了の確認を受信しています。
  - **UL_SYNCH_STATE_RECEIVING_TABLE**　テーブルを受信しています。
  - **UL_SYNCH_STATE_RECEIVING_DATA**　スキーマ情報またはデータを受信しています。
  - **UL_SYNCH_STATE_COMMITTING_DOWNLOAD**　ダウンロード処理が完了し、コミットが実行されています。
  - **UL_SYNCH_STATE_SENDING_DOWNLOAD_ACK**　ダウンロード完了の確認が送信されています。
  - **UL_SYNCH_STATE_DISCONNECTING**　同期ストリームが閉じられようとしています。
  - **UL_SYNCH_STATE_DONE**　同期は正常に完了しました。
  - **UL_SYNCH_STATE_ERROR**　同期は完了しましたが、エラーが発生しました。
  - **UL_SYNCH_STATE_ROLLING_BACK_DOWNLOAD**　ダウンロード中にエラーが発生し、ダウンロードがロールバックされています。

  同期処理の詳細については、「同期処理」 『Mobile Link - クイック・スタート』を参照してください。
- **stop**　同期を中断するには、このメンバを true に設定します。SQL 例外の SQLE_INTERRUPTED が設定され、通信エラーが発生したかのように同期が停止します。observer は、適切なクリーンアップを実行するように、**常に** DONE または ERROR のステータスで呼び出されます。
- **flags**　現在の状態に関連する追加情報を示す、現在の同期フラグを返します。
- **user_data**　引数として ULRegisterSynchronizationCallback 関数に渡されるユーザ・データ・オブジェクトを返します。
- **sqlca**　接続のアクティブな SQLCA へのポインタ。

例

次の例は、ごく簡単な observer 関数を示しています。

```
extern void __stdcall ObserverFunc(
    p_ul_synch_status status )
{
    switch( status->state ) {
        case UL_SYNCH_STATE_STARTING:
        printf( "Starting¥n");
        break;
        case UL_SYNCH_STATE_CONNECTING:
            printf( "Connecting¥n"  );
            break;
        case UL_SYNCH_STATE_SENDING_HEADER:
            printf( "Sending Header¥n" );
            break;
        case UL_SYNCH_STATE_SENDING_TABLE:
            printf( "Sending Table %d of %d¥n",
                    status->tableIndex + 1,
                    status->tableCount );
            break;
        case UL_SYNCH_RECEIVING_UPLOAD_ACK:
            printf( "Receiving Upload Ack¥n" );
            break;
        case UL_SYNCH_STATE_RECEIVING_TABLE:
            printf( "Receiving Table %d of %d¥n",
                    status->tableIndex + 1,
                    status->tableCount );
            break;
        case UL_SYNCH_STATE_SENDING_DOWNLOAD_ACK:
            printf( "Sending Download Ack¥n" );
            break;
        case UL_SYNCH_STATE_DISCONNECTING:
            printf( "Disconnecting¥n" );
            break;
        case UL_SYNCH_STATE_DONE:
            printf( "Done¥n" );
            break;
        break;
...
```

この observer 関数では、2 つのテーブルが同期されたときに次のような出力が生成されます。

```
Starting
Connecting
Sending Header
Sending Table 1 of 2
Sending Table 2 of 2
Receiving Upload Ack
Receiving Table 1 of 2
Receiving Table 2 of 2
Sending Download Ack
Disconnecting
Done
```

CustDB の例

observer 関数の例は CustDB サンプル・アプリケーションに含まれています。CustDB を使った実装では、同期の進捗状況を示すウィンドウが表示されます。ユーザはそのウィンドウで同期をキャンセルすることができます。ユーザ・インタフェース・コンポーネントは observer 関数のプラットフォームを指定します。

CustDB サンプル・コードは *samples-dir¥UltraLite¥CustDB* ディレクトリにあります。observer 関数は *CustDB* ディレクトリのプラットフォーム固有のサブディレクトリに含まれています。

Embedded SQL アプリケーションの構築

この項では、Ultra Light Embedded SQL アプリケーションの一般的な構築プロシージャについて説明します。

この項は、Embedded SQL の開発モデルを全般的に理解している人を対象にしています。

一般的な構築手順

サンプル・コード

このプロセスを使用する makefile は、*samples-dir¥UltraLite¥ESQLSecurity* ディレクトリにあります。

別途ライセンスが必要な必須コンポーネント
ECC 暗号化と FIPS 認定の暗号化には、別途ライセンスが必要です。強力な暗号化テクノロジはすべて、輸出規制対象品目です。

「別途ライセンスが必要なコンポーネント」 『SQL Anywhere 11 - 紹介』を参照してください。

プロシージャ

♦ **Ultra Light Embedded SQL アプリケーションを構築するには、次の手順に従います。**

1. **各** Embedded SQL ソース・ファイルに対して SQL プリプロセッサを実行します。

   SQL プリプロセッサは sqlpp コマンドライン・ユーティリティです。Embedded SQL ソース・ファイルの前処理を実行し、アプリケーションにコンパイルする C++ ソース・ファイルを生成します。

   SQL プリプロセッサの詳細については、「Ultra Light SQL プリプロセッサ・ユーティリティ (sqlpp)」 『Ultra Light データベース管理とリファレンス』を参照してください。

   警告
   sqlpp は出力ファイルをその内容に関係なく上書きします。出力ファイルの名前が、どのソース・ファイルの名前とも一致していないことを確認してください。*sqlpp* は、デフォルトでは、ソース・ファイルの拡張子を *.cpp* に変更して出力ファイル名を作成します。一致するファイル名があるかどうかはっきり分からない場合は、ソース・ファイルの名前に従って出力ファイルの名前を明示的に指定してください。

2. **各** C++ ソース・ファイルを、選択したターゲット・プラットフォームに合わせてコンパイルします。次のファイルを含めます。
   - SQL プリプロセッサが生成した各 C++ ファイル
   - アプリケーションで必要な追加の C または C++ ソース・ファイル
3. これら**すべて**のオブジェクト・ファイルを Ultra Light ランタイム・ライブラリとともにリンクします。

開発ツールを Embedded SQL 開発用に設定する

多くの開発ツールは依存モデルを使用しています。これは makefile として表現されることもあり、ソース・ファイルのタイムスタンプが、ターゲット・ファイル (ほとんどの場合、オブジェクト・ファイル) のタイムスタンプと比較され、ターゲット・ファイルを再生成すべきかどうかが決定されます。

Ultra Light の開発では、開発プロジェクトで SQL 文が変更されると、生成コードを再生成する必要があります。SQL 文はリファレンス・データベースに保存されるため、個々のソース・ファイルのタイムスタンプには変更が反映されません。

この項では、Ultra Light アプリケーション開発、特に SQL プリプロセッサを依存ベースの構築環境に追加する方法について説明します。Visual C++ に関する特殊な指示もあります。また、これらを開発ツールとして使用するには修正する必要もあります。

ここで説明するテクニックは、CodeWarrior 用の Ultra Light プラグインによって自動的に提供され、Palm Computing Platform の開発者が利用できます。このプラグインの詳細については、「Palm OS 用 Ultra Light アプリケーションの開発」 67 ページを参照してください。

SQL 前処理

まず、SQL プリプロセッサを実行する命令を開発ツールに追加する手順について説明します。

♦ **Embedded SQL 前処理を依存ベースの開発ツールに追加するには、次の手順に従います。**

1. *.sqc* ファイルを開発プロジェクトに追加します。

   開発プロジェクトは、開発ツールで定義されています。

2. 各 *.sqc* ファイルのカスタム構築規則を追加します。
   - カスタム構築規則により SQL プリプロセッサを実行してください。Visual C++ の場合、構築規則には次のコマンドを含めてください (すべて 1 行に入力)。

     ```
     "%SQLANY11%¥Bin32¥sqlpp.exe" -q -u $(InputPath) $(InputName).cpp
     ```

     SQLANY11 は SQL Anywhere インストール・ディレクトリを指す環境変数です。

     SQL プリプロセッサのコマンド・ラインの詳細については、「Ultra Light SQL プリプロセッサ・ユーティリティ (sqlpp)」 『Ultra Light データベース管理とリファレンス』を参照してください。
   - コマンドの出力を **$(InputName).cpp** に設定します。
3. *.sqc* ファイルをコンパイルし、生成された *.cpp* ファイルを開発プロジェクトに追加します。

   生成されたファイルはソース・ファイルではありませんが、プロジェクトに追加する必要があります。これは、依存性と構築のオプションを設定できるようにするためです。

4. 生成された *.cpp* ファイルごとに、プリプロセッサの定義を設定します。
   - [全般] または [プリプロセッサ] で、[プリプロセッサ] の定義に UL_USE_DLL を追加します。

- [プリプロセッサ] に、*$(SQLANY11)¥SDK¥Include* と、インクルード・パスに必要なインクルード・フォルダを、カンマ区切りのリストとして追加します。

Palm OS 用 Ultra Light アプリケーションの開発

目次

CodeWarrior プラグインを使用すると、Embedded SQL または Ultra Light C++ による Ultra Light アプリケーションの作成が簡単になります。

プラグインは *install-dir¥UltraLite¥Palm¥68k¥cwplugin* ディレクトリにあります。このディレクトリにあるファイル *readme.txt* を確認してください。

CodeWarrior には Palm SDK が含まれています。ターゲットとする特定のデバイスによって異なりますが、この開発ツールに含まれている Palm SDK をより新しいバージョンにアップグレードすることをおすすめします。

サポートされるプラットフォームのリストについては、「サポートされるプラットフォーム」『SQL Anywhere 11 - 紹介』を参照してください。

サポートされているターゲット・オペレーティング・システムのリストについては、「サポートされるプラットフォーム」『SQL Anywhere 11 - 紹介』を参照してください。

Ultra Light plug-in for CodeWarrior のインストール

Ultra Light plug-in for CodeWarrior のファイルは、Ultra Light のインストール中にディスクにコピーされますが、プラグインは、追加のインストール手順を実行しなければ使用できません。

♦ Ultra Light plug-in for CodeWarrior のインストール

1. コマンド・プロンプトで、*install-dir¥UltraLite¥Palm¥68k¥cwplugin* ディレクトリに移動します。
2. *install.bat* を実行して、適切なファイルを CodeWarrior インストール・ディレクトリにコピーします。*install.bat* ファイルは、次の 2 つの引数を取ります。
   - CodeWarrior ディレクトリ
   - CodeWarrior バージョン

   たとえば、次のコマンドを 1 行に入力して実行すると、CodeWarrior 9 のプラグインがデフォルトの CodeWarrior インストール・ディレクトリにインストールされます。

   ```
   install "c:¥Program Files¥Metrowerks¥CodeWarrior for Palm OS Platform 9.0" r9
   ```

   パスにスペースが埋め込まれている場合は、ディレクトリ全体を二重引用符で囲んでください。

CodeWarrior プラグインのアンインストール

uninstall.bat を使用すると、Ultra Light plug-in for CodeWarrior をアンインストールできます。*uninstall.bat* ファイルには、前述した *install.bat* と同じ引数が必要です。

CodeWarrior での Ultra Light プロジェクトの作成

♦ **CodeWarrior で Ultra Light プロジェクトを作成するには、次の手順に従います。**

1. CodeWarrior を起動します。
2. 新規プロジェクトを作成します。
   a. **CodeWarrior** メニューから **[ファイル] - [新規]** を選択します。
   b. **[プロジェクト]** タブをクリックします。
   c. **[Palm OS アプリケーション・ステーショナリ]** を選択します。
   d. プロジェクトの名前と場所を選択し、**[OK]** をクリックします。
3. Ultra Light ステーショナリを選択します。

   Ultra Light プラグインによって、ステーショナリ・リストに次の選択肢が追加されます。

   - Palm OS Ultra Light C++アプリケーション
   - Palm OS Ultra Light ESQL アプリケーション

   使用する開発モデルを選択して **[OK]** をクリックすると、プロジェクトが作成されます。

   ステーショナリは、Embedded SQL では標準の C ステーショナリ、C++ では標準の C++ ステーショナリです。
4. Embedded SQL を使用している場合は、プロジェクトの **[Ultra Light プリプロセッサ]** パネルで設定してください。C++ を使用している場合は、これらの設定は無視されます。
   a. プロジェクト・ウィンドウ (*.mcp*) で、ツールバーの **[設定]** アイコンをクリックします。
   b. 左ウィンドウ枠にあるツリーで **[ターゲット] - [Ultra Light プリプロセッサ]** を選択します。プロジェクトの設定を入力します。

前処理

Embedded SQL プロジェクトを構築する場合、Ultra Light プラグインによって *sqlpp* が呼び出されて、*.sqc* ファイルが *.c*/*.cpp* ファイルに変換され、ESQL 文が Ultra Light 関数呼び出しに変換されます。

CodeWarrior 環境で Ultra Light Embedded SQL または C++ アプリケーションを構築する場合、プラグインによって、*install-dir¥SDK¥Include* および *install-dir¥UltraLite¥Palm¥68k¥lib¥cw¥* へのアクセス・パスは追加されません。また、Ultra Light ライブラリの *ulrt.lib* と *ulbase.lib* へのパスも追加されません。プラグインは、SQL プリプロセッサの実行で生成されたファイルだけを Ultra Light Embedded SQL アプリケーションの CodeWarrior プロジェクトに追加します。

既存の CodeWarrior プロジェクトから Ultra Light アプリケーションへの変換

Ultra Light plug-in for CodeWarrior をインストールすると、以前のプロジェクトを初めて開くときに、そのプロジェクトを変換するかどうかを確認するプロンプトが表示されます。この変換では、CodeWarrior が SQL プリプロセッサのデフォルト設定を完了し、設定内容をプロジェクト・ファイルに保存します。これによって SQL プリプロセッサを使用しないプロジェクトが悪影響を受けることはありません。さらにプロジェクトが SQL プリプロセッサの呼び出しを自動的に行うように変換するには、次の手順を実行する必要があります。

1. *.sqc* ファイルのファイル・マッピング・エントリを [ターゲット設定] の [ファイル・マッピング] パネルに追加します。

   このファイルのタイプは **TEXT** であり、コンパイラは **UltraLite Preprocessor** です。これらのファイルでは、フラグのチェックをすべてはずしてください。

2. Embedded SQL アプリケーションでは、生成されたすべてのファイルを [ファイル] ビューから削除します。これらのファイルは、*.sqc* ファイルが構築されるときに自動的に生成され、再び追加されます。

3. *.ulg* ファイルのファイル・マッピングをすべて削除します。

4. 必要なライブラリ・ファイルへの参照を確認します。

Ultra Light plug-in for CodeWarrior の使用

Embedded SQL では、Ultra Light plug-in for CodeWarrior は Ultra Light の前処理手順を CodeWarrior コンパイル・モデルに統合します。これにより、必要なときに SQL プリプロセッサを確実に実行できます。

プレフィクス・ファイルの使用

プレフィクス・ファイルは、CodeWarrior プロジェクトのすべてのソース・ファイルにインクルードする必要があるヘッダ・ファイルです。*install-dir¥SDK¥Include¥ulpalmos.h* をプレフィクス・ファイルとして使用してください。

独自のプレフィクス・ファイルがある場合は、*ulpalmos.h* をインクルードします。*ulpalmos.h* ファイルには、Ultra Light Palm アプリケーションに必要なマクロが定義されているほか、Ultra Light に必要な CodeWarrior コンパイラ・オプションが設定されています。

暗号化された同期
TLS プロトコルまたは HTTPS プロトコルを使用して暗号化された同期を実装する場合は、*ulrsa.lib*、*ulecc.lib*、または *ulfips.lib* と *gselst.lib* を CodeWarrior Ultra Light プロジェクトに追加する必要があります。

CodeWarrior を使用した CustDB サンプル・アプリケーションの構築

CustDB は簡単な販売管理アプリケーションです。

サンプルのデータベース・スキーマの図については、「CustDB サンプル・データベースについて」 『SQL Anywhere 11 - 紹介』を参照してください。

アプリケーション用のファイルは、*samples-dir¥UltraLite¥CustDB* ディレクトリにあります。汎用ファイルは *CustDB* ディレクトリにあります。CodeWarrior for Palm OS 固有のファイルは、次のディレクトリにあります。

- *samples-dir¥UltraLite¥CustDB¥cwcommon*
- *samples-dir¥UltraLite¥CustDB¥cw*

この項では、CodeWarrior 9 を使用した CustDB アプリケーションの構築方法について説明します。

♦ CodeWarrior を使用して CustDB サンプル・アプリケーションを構築するには、次の手順に従います。

1. CodeWarrior IDE を起動します。
2. CustDB プロジェクト・ファイルを開きます。
   - **[ファイル] - [開く]** を選択します。
   - プロジェクト・ファイル *samples-dir¥UltraLite¥CustDB¥cw¥custdb.mcp* を開きます。
3. ターゲット・アプリケーション (*custdb.prc*) を構築するには、**[プロジェクト] - [メイク]** を選択します。

拡張モード・アプリケーションの構築

CodeWarrior は、「拡張モード」というコード生成モードをサポートしています。このモードを使用すると、グローバル・データのメモリ利用効率が向上します。CodeWarrior バージョン 9 を使用している場合は、A5 ベースのジャンプ・テーブルで拡張モードを使用できます。使用するには、拡張モード・バージョンの Ultra Light ランタイム・ライブラリと Ultra Light ベース・ライブラリを使用する必要があります。これらのライブラリの拡張モード・バージョンは、次の位置にあります。

- *install-dir¥UltraLite¥Palm¥68k¥lib¥cw9_a4a5jt¥ulrt.lib*
- *install-dir¥UltraLite¥Palm¥68k¥lib¥cw9_a4a5jt¥ulbase.lib*

拡張モードは、グローバル・データの 64 KB の制限を超える大規模なアプリケーションで役立つ場合があります。拡張モードの制限は、HotSync を使用しないと暗号化された同期を使用できないことです。これは、Ultra Light の同期セキュリティ・ライブラリでは拡張モードが使用されないためです。

Ultra Light Palm アプリケーションのステータスの管理 (旧式)

接続を閉じるのではなくサスペンドしてアプリケーションを閉じる場合は、テーブルとカーソルの状態を保存できます。

現在の状態は、接続オブジェクトが開いているときに開いているテーブルに対してのみ保存されます。

Ultra Light アプリケーションを終了したり、別のアプリケーションに切り替えたりするときは、開いているカーソルとテーブルの状態が保存されます。

1. ユーザがアプリケーションに戻ったら、適切な open メソッドを呼び出します。
   - Embedded SQL の場合は、次の関数を呼び出します。
     - db_init
     - EXEC SQL CONNECT
   - C++ の場合は、次の関数を呼び出します。
     - ULSqlca.Initialize
     - ULInitDatabaseManager
     - OpenConnection
2. SQLCODE が SQLE_CONNECTION_RESTORED であることをチェックすることによって、接続が正しくリストアされたことを確認します。
3. 生成された結果セット・クラスのインスタンスを含むカーソル・オブジェクトについては、次のいずれかを実行できます。
   - ユーザが別のアプリケーションに切り替えたときにオブジェクトが必ず閉じるようにし、そのオブジェクトが次に必要となったときに Open を呼び出します。このオプションを選択した場合、現在の位置はリストアされません。
   - ユーザが別のアプリケーションに切り替えたときにオブジェクトを閉じないで、そのオブジェクトへのアクセスが次に必要となったときに Reopen を呼び出します。現在の位置は保持されますが、ユーザが他のアプリケーションを使用している間、このアプリケーションのメモリ使用量が増加します。
4. 生成されたテーブル・クラスのインスタンスを含め、テーブル・オブジェクトでは位置を保存することはできません。ユーザが別のアプリケーションに切り替える前にテーブル・オブジェクトを閉じ、それらのオブジェクトが必要となったときに Open を呼び出す必要があります。テーブル・オブジェクトに Reopen は使用しないでください。

接続を閉じると、コミットされていないトランザクションはロールバックされます。接続オブジェクトを閉じないことで未処理のトランザクションは保存されるため (コミットされません)、アプリケーションが再起動したときにそれらのトランザクションが表示され、コミットしたりロールバックしたりできます。コミットされていない変更は同期されません。

Ultra Light Palm アプリケーションの状態のリストア (旧式)

Palm OS 上でアプリケーションが再起動すると、アプリケーションの前回のシャットダウン時に明示的に閉じられなかったカーソルやテーブルの状態がリストアされます。

Palm OS の暗号化キーの保存、取り出し、クリア

Palm OS では、ユーザが別のアプリケーションに切り替えると、アプリケーションがシステムによって自動的に終了します。そのため、Palm OS 上で Ultra Light データベースを暗号化する場合は、アプリケーションに切り替える**たびに**キーの再入力を要求するプロンプトが表示されます。

♦ **暗号化キーの再入力を回避するには、次の手順に従います。**

1. Palm の機能として、暗号化キーを動的メモリに保存します。

   機能には、作成者と機能番号のインデックスが付けられます。アプリケーションでは、各自の作成者 ID または NULL を、機能番号または NULL と一緒に渡して、暗号化キーを保存したり取り出したりできます。

2. アプリケーションで、再起動時にキーを取り出すようにプログラミングします。

   注意
   デバイスのリセット時は暗号化キーがクリアされるため、そのときはキーの取り出しに失敗します。この場合、キーの再入力を要求するプロンプトが表示されます。

   次のサンプル・コード (Embedded SQL) は、暗号化キーの保存と取り出し方法を示しています。

```
startupRoutine() {
  ul_char buffer[MAX_PWD];

  if( !ULRetrieveEncryptionKey(
        buffer, MAX_PWD, NULL, NULL  ) ){
    // prompt user for key
    userPrompt( buffer, MAX_PWD );

    if( !ULSaveEncryptionKey( buffer, NULL, NULL ) ) {
      // inform user save failed
    }
  }
}
```

3. メニュー項目を使用して、暗号化キーをクリアしてデバイスがセキュリティで保護されるようにします。

   次のコード・サンプルは、この目的を達成する方法を示しています。

```
case MenuItemClear
   ULClearEncryptionKey( NULL, NULL );
   break;
```

参照

- Ultra Light.NET の場合：「ULConnectionParms クラス」 『Ultra Light - .NET プログラミング』
- Ultra Light for C++ の場合：「UltraLite_Connection_iface クラス」 158 ページ
- 「Ultra Light DBKEY 接続パラメータ」 『Ultra Light データベース管理とリファレンス』
- Ultra Light for M-Business Anywhere の場合：「データベースの暗号化と難読化」 『Ultra Light - M-Business Anywhere プログラミング』
- 「Ultra Light obfuscate 作成パラメータ」 『Ultra Light データベース管理とリファレンス』

Palm 作成者 ID の登録

すべての Palm OS アプリケーションと同様に、Palm OS の Ultra Light アプリケーションも作成者 ID を必要とします。この作成者 ID は作成時にアプリケーションに割り当てます。HotSync 同期を使用している場合は、Mobile Link 同期によって使用される作成者 ID を HotSync マネージャに登録します。

作成者 ID をアプリケーションに割り当てる方法については、開発ツールのマニュアルを参照してください。HotSync マネージャでの作成者 ID の登録については、「Palm OS の HotSync」『Ultra Light データベース管理とリファレンス』を参照してください。

作成者 ID は、1 ～ 4 文字の文字列です。Palm OS では、頭文字に小文字が使用される文字列は Palm OS システムが使用するために予約されているため、先頭の文字を大文字にしてください。

Palm アプリケーションへの HotSync 同期の追加

HotSync 同期は、Ultra Light アプリケーションが閉じるときに行われます。同期は HotSync によって開始されます。

HotSync を使用する場合は、アプリケーションを閉じる前に ULSetSynchInfo を呼び出して同期します。HotSync 同期には、ULSynchronize や ULConnection.Synchronize は使用しないでください。

アプリケーションから HotSync 同期を有効にするには、次の手順でコードを追加します。

1. ul_synch_info 構造体を準備します。
2. ULSetSynchInfo 関数を呼び出し、ul_synch_info 構造体を引数として指定します。

   この関数は、ユーザが Ultra Light アプリケーションから別のアプリケーションに切り替えたときに呼び出されます。すべての処理がコミットされていることを確認してから、db_fini を呼び出してください。ul_synch_info.stream パラメータは無視されるので、設定する必要はありません。

   次に例を示します。

   ```
   //C++ API
   ul_synch_info info;
   ULInitSynchInfo( &info );
   info.stream_parms =
     UL_TEXT( "stream=tcpip;host=localhost" );
   info.user_name = UL_TEXT( "50" );
   info.version = UL_TEXT( "custdb" );

   ULSetSynchInfo( &sqlca, &info );

   if( !db.Close( ) ) {
     return( false );
   }
   ```
3. db_fini を呼び出します。

「Ultra Light Palm アプリケーションのステータスの管理 (旧式)」 74 ページと「Ultra Light の同期パラメータ」 『Ultra Light データベース管理とリファレンス』を参照してください。

Ultra Light アプリケーションの HotSync 同期には、Ultra Light HotSync コンジットが必要です。Palm アプリケーションの終了時にコミットされていないトランザクションがあるときに同期を行った場合、コンジットは、データベース内にコミットされていない変更があるため同期が失敗したとレポートします。

ストリーム・パラメータの指定

ul_synch_info 構造体の同期ストリーム・パラメータは、Mobile Link サーバとの通信を制御します。HotSync 同期では、Ultra Light アプリケーションは Mobile Link サーバと直接通信しません。HotSync コンジットと通信します。

同期ストリーム・パラメータを次のいずれかの方法で指定することによって、Mobile Link コンジットの動作を制御できます。

- ULSetSynchInfo に渡される ul_synch_info の stream_parms メンバに必要な情報を提供します。

使用可能な値のリストについては、「Ultra Light 同期ストリームのネットワーク・プロトコルのオプション」『Ultra Light データベース管理とリファレンス』を参照してください。

- stream_parms メンバに NULL 値を指定します。Mobile Link コンジットは自らが動作しているコンピュータ上の *ClientParms* レジストリ・エントリを調べ、Mobile Link サーバへの接続方法に関する情報を見つけます。

  レジストリ・エントリ内のストリームとストリーム・パラメータは、ul_synch_info 構造体の stream_parms フィールドと同じフォーマットで指定します。

  「Ultra Light 同期パラメータとネットワーク・プロトコル・オプション」『Ultra Light データベース管理とリファレンス』を参照してください。

参照

- 「Palm OS の HotSync」『Ultra Light データベース管理とリファレンス』

Palm アプリケーションへの TCP/IP、HTTP、HTTPS 同期の追加

この項では、Palm アプリケーションに TCP/IP、HTTP、または HTTPS 同期を追加する方法について説明します。

Ultra Light アプリケーションに同期を追加する一般的な方法については、「Ultra Light アプリケーションへの同期の追加」『Ultra Light データベース管理とリファレンス』を参照してください。

Palm OS でのトランスポート・レイヤ・セキュリティ

CodeWarrior で構築した Palm アプリケーションでは、トランスポート・レイヤ・セキュリティを使用できます。

トランスポート・レイヤ・セキュリティの詳細については、「Mobile Link クライアント／サーバ通信の暗号化」『SQL Anywhere サーバ - データベース管理』を参照してください。

Palm デバイスとの同期に TCP/IP、HTTP、または HTTPS 通信を使用する場合は、ul_synch_info 構造体の stream メンバを適切なストリームに設定し、ULSynchronize または ULConnection.Synchronize を呼び出します。

TCP/IP、HTTP、または HTTPS 通信による同期では、db_init または db_fini が、アプリケーション終了時にアプリケーションのステータスを保存し、起動時にリストアしますが、同期には関わりません。

アプリケーションを閉じる前に、ul_synch_info 構造体を引数として指定した ULSetSynchInfo を使用して、同期情報を設定します。

Palm デバイスからの TCP/IP、HTTP、または HTTPS による同期を使用する場合は、ul_synch_info 構造体の stream_parms メンバに明示的なホスト名または IP アドレスを指定してください。NULL を指定すると、デフォルトの localhost が使用されます。これは、ホストではなくデバイスを表します。

ul_synch_info 構造体の詳細については、「Ultra Light 同期ストリームのネットワーク・プロトコルのオプション」『Ultra Light データベース管理とリファレンス』を参照してください。

Palm アプリケーションの配備

この項では、Palm アプリケーションの配備に関する次の事柄について説明します。

- アプリケーションの配備。
  「アプリケーションの配備」 81 ページを参照してください。
- Ultra Light HotSync 同期コンジットの配備。
  「Palm OS の HotSync」 『Ultra Light データベース管理とリファレンス』を参照してください。
- Ultra Light データベースの初期コピーの配備。
  「Ultra Light データベースの配備」 81 ページを参照してください。

Ultra Light アプリケーションを他の Palm OS アプリケーションと同じように Palm デバイスにインストールします。

アプリケーションの配備

♦ **アプリケーションを Palm デバイスにインストールするには、次の手順に従います。**

1. Palm Desktop Organizer Software に含まれている [インストール ツール] を開きます。
2. **[追加]** を選択し、コンパイル済みのアプリケーション (*.prc* ファイル) のロケーションを指定します。
3. インストール ツールを閉じます。
4. HotSync ユーティリティを使用してアプリケーションを Palm デバイスにコピーします。

Mobile Link 同期コンジットの配備

HotSync 同期を使用するアプリケーションについては、各エンド・ユーザがデスクトップ・マシンに Mobile Link 同期コンジットをインストールしてください。

Mobile Link 同期コンジットのインストールの詳細については、「Palm OS の HotSync」 『Ultra Light データベース管理とリファレンス』を参照してください。

Ultra Light データベースの配備

アプリケーションをデータベースなしで配備した場合、アプリケーションには、データベースを作成するために比較的複雑なコードを含める必要があります。Windows デスクトップで初期データベースを作成し、そのデータベースファイルを Palm デバイスにコピーすることをおすすめします。Sybase Central (または ulcreate ユーティリティ) を使用して初期データベースを作成できます。その後、ユーザは最初の同期でデータの初期コピーを取得する必要があります。uldbutil ユーティリティを使用して、Ultra Light データベースを PC にバックアップできます。データが格納されている初期データベースとともに多数の Ultra Light データベースを配備するには、初期同期を実行した後で Ultra Light データベースをバックアップします。このデータベースを他のデバイスに配備すると、各デバイスで初期同期を実行する必要がなくなります。

「Ultra Light の Palm OS 用データ管理ユーティリティ (ULDBUtil)」『Ultra Light データベース管理とリファレンス』を参照してください。

また、HotSync 同期を使用している場合は、エンド・ユーザが各自のデスクトップ・コンピュータに同期コンジットをインストールする必要があります。

同期コンジットのインストールの詳細については、「Ultra Light HotSync コンジットの配備」『Ultra Light データベース管理とリファレンス』を参照してください。

HotSync を使用してデータベースを配備する場合は、HotSync がデータベースにバックアップ・ビットを設定します。このバックアップ・ビットが設定されると、同期のたびにデータベース全体がデスクトップ・コンピュータにバックアップされます。この動作は、一般に Ultra Light データベースには適しません。Ultra Light アプリケーションが起動すると、Palm データ・ストアでバックアップ・ビットが true に設定されているかどうかが確認されます。設定されている場合はクリアされます。設定されていない場合は変更されません。

バックアップ・ビットを true のままにするには、データベース接続文字列にストア・パラメータ palm_allow_backup を設定します。

Windows Mobile 用 Ultra Light アプリケーションの開発

目次

Microsoft eMbedded Visual C++ は、Windows Mobile 環境用のアプリケーション開発に使用できます。この開発環境は、Microsoft の eMbedded Visual Tools の一部です。

Microsoft eMbedded Visual C++ は、Microsoft Developer Network (MSDN) の http://msdn.microsoft.com/ からダウンロードできます。

Windows Mobile を対象にしたアプリケーションでは、wchar_t のデフォルト設定を使用し、*install-dir¥ultralite¥ce¥arm.50¥lib¥* 内にある Ultra Light ランタイム・ライブラリにリンクする必要があります。

Windows Mobile 向けの開発でサポートされているホスト・プラットフォームと開発ツール、およびサポートされているターゲット Windows Mobile プラットフォームのリストについては、「サポートされるプラットフォーム」『SQL Anywhere 11 - 紹介』を参照してください。

ほとんどの Windows Mobile ターゲット・プラットフォームのエミュレータで、アプリケーションをテストできます。

ランタイム・ライブラリをリンクする方法の選択

Windows Mobile はダイナミック・リンク・ライブラリをサポートしています。リンク時には、インポート・ライブラリを使って Ultra Light アプリケーションをランタイム DLL にリンクするか、それとも Ultra Light ランタイム・ライブラリを使ってアプリケーションを静的にリンクするか、いずれかの方法を選択できます。

パフォーマンスに関するヒント
ターゲット・デバイスに 1 つの Ultra Light アプリケーションがある場合は、ライブラリを静的にリンクした方がメモリの使用量は少なくなります。ターゲット・デバイスに複数の Ultra Light アプリケーションがある場合は、DLL を使用した方がメモリ使用量の点で経済的です。

Ultra Light アプリケーションを低速リンクでデバイスに繰り返しダウンロードする場合は、初期ダウンロード後にダウンロードされる実行プログラムのサイズを最小化するために、DLL を使うとよいでしょう。

♦ **Ultra Light ランタイム DLL を使用してアプリケーションを構築し配備するには、次の手順に従います。**

1. コードを前処理してから、UL_USE_DLL で出力をコンパイルします。
2. Ultra Light インポート・ライブラリを使用してアプリケーションをリンクします。
3. アプリケーションの実行プログラムと Ultra Light ランタイム DLL の両方をターゲット・デバイスにコピーします。

CustDB サンプル・アプリケーションの構築

CustDB は簡単な販売管理アプリケーションです。CustDB は SQL Anywhere インストール環境の *samples-dir¥UltraLite¥* ディレクトリにあります。汎用ファイルは *CustDB* サブディレクトリにあります。Windows Mobile 用のファイルは *CustDB* の *EVC* サブディレクトリにあります。

CustDB アプリケーションは、eMbedded Visual C++ 3.0 プロジェクトとして提供されます。

サンプルのデータベース・スキーマの図については、「CustDB サンプル・データベースについて」 『SQL Anywhere 11 - 紹介』を参照してください。

♦ **CustDB サンプル・アプリケーションを構築するには、次の手順に従います。**

1. eMbedded Visual C++ を起動します。
2. 使用する eMbedded Visual C++ のバージョンに対応した、次のいずれかのプロジェクト・ファイルを開きます。
   - eVC 3.0 の場合は *samples-dir¥UltraLite¥CustDB¥EVC¥EVCCustDB.vcp*
   - eVC 4.0 の場合は *samples-dir¥UltraLite¥CustDB¥EVC40¥EVCCustDB.vcp*
3. **[ビルド] - [アクティブ プラットフォームの設定]** を選択し、ターゲット・プラットフォームを設定します。
   - 選択するプラットフォームを設定します。
4. **[ビルド] - [アクティブ構成の設定]** を選択し、構成を選択します。
   - 選択するアクティブ構成を設定します。
5. Pocket PC x86em エミュレータ・プラットフォーム用の CustDB を構築する場合にかぎり、次の手順を実行します。
   - **[プロジェクト] - [設定]** を選択します。
   - **[リンク]** タブの **[オブジェクト/ライブラリ モジュール]** フィールドで、Ultra Light ランタイム・ライブラリ・エントリを、*emulator* ディレクトリではなく *emulator30* ディレクトリに変更します。
6. アプリケーションを構築します。
   - **[F7]** キーを押すか、**[ビルド] - [EVCCustDB.exe をビルド]** を選択して、CustDB を構築します。

     アプリケーションの構築が完了すると、eMbedded Visual C++ は自動的に、そのアプリケーションをリモート・デバイスにアップロードしようとします。
7. Mobile Link サーバを起動します。
   - Mobile Link サーバを起動するには、**[スタート] - [プログラム] - [SQL Anywhere 11] - [Mobile Link] - [同期サーバのサンプル]** を選択します。
8. CustDB アプリケーションを実行します。

CustDB アプリケーションを実行する前に、custdb データベースをデバイスのルート・フォルダにコピーする必要があります。*samples-dir¥UltraLite¥CustDB¥custdb.udb* というデータベース・ファイルをデバイスのルートにコピーします。

[Ctrl + F5] キーを押すか、**[ビルド] - [実行 CustDB.exe]** を選択します。

フォルダのロケーションと環境変数
サンプル・プロジェクトでは、可能なかぎり環境変数を使用しています。アプリケーションを正しく構築するために、プロジェクトの調整が必要になることもあります。問題が発生した場合は、Microsoft Visual C++ のフォルダに足りないファイルがないかどうか調べ、適切なディレクトリ設定を追加してみてください。

Embedded SQL の場合、構築プロセスでは、*CustDB.sqc* ファイルを *CustDB.cpp* ファイルに変換するときに、SQL プリプロセッサ *sqlpp* を使用します。すべての Embedded SQL を 1 つのソース・モジュールに収めることができる小規模の Ultra Light アプリケーションでは、1 つのステップで処理できるこの方法が便利です。大規模な Ultra Light アプリケーションでは、複数の *sqlpp* 呼び出しを使用する必要があります。

「Embedded SQL アプリケーションの構築」 64 ページを参照してください。

永続的データの格納

Ultra Light データベースは Windows Mobile ファイル・システムに格納されます。デフォルト・ファイルは *¥UltraLiteDB¥ul_<project>.udb* です。*project* は 8 文字にトランケートされます。この設定は、ファイルベースの永続ストアのフル・パス名を指定する **file_name** 接続パラメータを使用して上書きできます。

Ultra Light ランタイムは **file_name** パラメータへの代入を行いません。ファイル名を有効化するためにディレクトリの作成が必要な場合は、**db_init** を呼び出す前にディレクトリが作成されることをアプリケーション側で確認してください。

たとえば、フラッシュ・メモリ・ストレージ・カードを使用する場合は、ストレージ・カードを検索し、そのストレージ・カードの名前の前に、次のようにディレクトリ名を追加します。次に例を示します。

```
file_name = "¥¥Storage Card¥¥My Documents¥¥flash.udb"
```


Windows Mobile アプリケーションの配備

Windows Mobile 用の Ultra Light アプリケーションをコンパイルでは、Ultra Light ランタイム・ライブラリを静的または動的にリンクできます。動的にリンクする場合は、目的のプラットフォーム用の Ultra Light ランタイム・ライブラリをターゲット・デバイスにコピーします。

♦ Ultra Light ランタイム DLL を使用してアプリケーションを構築し配備するには、次の手順に従います。

1. コードを前処理してから、UL_USE_DLL で出力をコンパイルします。
2. Ultra Light インポート・ライブラリを使用してアプリケーションをリンクします。
3. アプリケーションの実行プログラムと Ultra Light ランタイム DLL の両方をターゲット・デバイスにコピーします。

   Ultra Light ランタイム DLL は、SQL Anywhere インストール・ディレクトリの *¥ultralite¥ce* サブディレクトリの下にある、チップの種類ごとのディレクトリに保管されています。

Windows Mobile エミュレータ用の Ultra Light ランタイム DLL を配備するには、Windows Mobile tools ディレクトリの適切なサブディレクトリに DLL を配置します。次のディレクトリは、Pocket PC エミュレータ用のデフォルト設定です。

C:¥Program Files¥Windows CE Tools¥wce300¥MS Pocket PC¥emulation¥palm300¥windows

ActiveSync を使用するアプリケーションの配備

ActiveSync 同期を使用するアプリケーションを ActiveSync で登録し、デバイスにコピーします。「ActiveSync Manager を使用したアプリケーションの登録」『Ultra Light データベース管理とリファレンス』を参照してください。

ActiveSync 用 Mobile Link プロバイダもインストールします。「Ultra Light の ActiveSync プロバイダの配備」『Ultra Light データベース管理とリファレンス』を参照してください。

アプリケーションに対するクラス名の割り当て

ActiveSync で使用するアプリケーションを登録するには、ウィンドウ・クラス名を指定します。クラス名の割り当ては開発時に行うので、その方法については、アプリケーション開発ツールのマニュアルが主な情報源になります。

Microsoft Foundation Classes (MFC) ダイアログ・ボックスには、汎用クラス名である **Dialog** が指定され、システム内のすべてのダイアログで共有されます。この項では、MFC と eMbedded Visual C++ を使用しているときに、アプリケーションに固有のクラス名を割り当てる方法について説明します。

♦ eMbedded Visual C++ を使用して、MFC アプリケーションにウィンドウ・クラス名を割り当てるには、次の手順に従います。

1. デフォルトのクラスに基づいて、ダイアログ・ボックスのカスタム・ウィンドウ・クラスを作成して登録します。

   アプリケーションの起動コードに、次のコードを追加します。このコードを実行してから、ダイアログを作成します。

   ```
   WNDCLASS wc;
   if( ! GetClassInfo( NULL, L"Dialog", &wc ) ) {
    AfxMessageBox( L"Error getting class info" );
   }
   wc.lpszClassName = L"MY_APP_CLASS";
   if( ! AfxRegisterClass( &wc ) ) {
    AfxMessageBox( L"Error registering class" );
   }
   ```

   MY_APP_CLASS は、アプリケーションのユニークなクラス名です。

2. アプリケーションのメイン・ダイアログにするダイアログを決めます。

   MFC アプリケーション・ウィザードを使ってプロジェクトを作成した場合、ダイアログ名は通常 **CMyAppDlg** です。

3. メイン・ダイアログのリソース ID を調べて記録しておきます。

   リソース ID は、IDD_MYAPP_DIALOG のような、一般形式の定数です。

4. アプリケーションの実行中は常にメイン・ダイアログが開かれていることを確認します。

   アプリケーションの **InitInstance** 関数に、次の行を追加します。これにより、メイン・ダイアログ **dlg** を閉じたときにアプリケーションも確実に閉じるようになります。

   ```
   m_pMainWnd = &dlg;
   ```

   詳細については、**CWinThread::m_pMainWnd** に関する Microsoft のマニュアルを参照してください。

   アプリケーションの実行中にダイアログが閉じられてしまう場合には、他のダイアログのウィンドウ・クラスも変更してください。

5. 変更内容を保存します。

eMbedded Visual C++ が開いたままになっていれば、変更内容を保存してプロジェクトとワークスペースを閉じます。

6. プロジェクトのリソース・ファイルを変更します。

   ● メモ帳などのテキスト・エディタで、リソース・ファイル (拡張子は *.rc*) を開きます。

   メイン・ダイアログのリソース ID を見つけます。

   ● メイン・ダイアログの定義を変更して、次の例のように、新しいウィンドウ・クラスが使用されるようにします。変更するのは、**CLASS** 行の追加**だけ**です。

   ```
   IDD_MYAPP_DIALOG DIALOG DISCARDABLE  0, 0, 139, 103
   STYLE WS_POPUP | WS_VISIBLE | WS_CAPTION
   EXSTYLE WS_EX_APPWINDOW | WS_EX_CAPTIONOKBTN
   CAPTION "MyApp"
   FONT 8, "System"
   CLASS "MY_APP_CLASS"
   BEGIN
       LTEXT   "TODO: Place dialog controls here.",IDC_STATIC,13,33,112,17
   END
   ```

   MY_APP_CLASS は、前の手順で使用したウィンドウ・クラスの名前です。

   ● *.rc* ファイルを保存します。

7. eMbedded Visual C++ を再起動してプロジェクトをロードします。

8. 同期メッセージを取得するコードを追加します。

   「ActiveSync 同期の追加 (MFC の場合)」 93 ページを参照してください。

Windows Mobile での同期

Windows Mobile 上の Ultra Light アプリケーションは、以下のストリームを介して同期できます。

- **ActiveSync** 「アプリケーションへの ActiveSync 同期の追加」 92 ページを参照してください。
- **TCP/IP** 「Windows Mobile からの TCP/IP、HTTP、HTTPS 同期」 95 ページを参照してください。
- **HTTP** 「Windows Mobile からの TCP/IP、HTTP、HTTPS 同期」 95 ページを参照してください。

Windows Mobile 用に *user_name* パラメータと *stream_parms* パラメータを初期化する場合、これらのパラメータを **UL_TEXT()** マクロで囲んでください。コンパイル環境が Unicode ワイド文字であるため、このようにする必要があります。

同期パラメータの詳細については、「Ultra Light の同期パラメータ」 『Ultra Light データベース管理とリファレンス』を参照してください。

アプリケーションへの ActiveSync 同期の追加

ActiveSync は、Microsoft 社が提供するソフトウェアで、Windows を実行しているデスクトップ・コンピュータと、そのコンピュータに接続された Windows Mobile ハンドヘルド・デバイスとの間のデータの同期を処理します。Ultra Light は ActiveSync バージョン 3.5 以降をサポートしています。

この項では、ActiveSync プロバイダをアプリケーションに追加する方法について説明します。また、エンド・ユーザのコンピュータ上の ActiveSync で使用するアプリケーションの登録方法についても説明します。

ActiveSync を使用する場合、同期を開始できるのは、ActiveSync 自体だけです。デバイスがクレードルにある場合や、[ActiveSync] ウィンドウで同期コマンドが選択された場合に、ActiveSync は同期を自動的に開始します。Mobile Link プロバイダは、アプリケーションを起動し (まだ動作していなかった場合)、アプリケーションにメッセージを送ります。

ActiveSync の同期の設定については、「ActiveSync を使用するアプリケーションの配備」 89 ページを参照してください。

ActiveSync プロバイダは **wParam** パラメータを使用します。**wParam** の値が 1 の場合、アプリケーションが ActiveSync 用 Mobile Link プロバイダによって起動されたことを示します。同期の完了後、アプリケーションは自動的に停止しなければなりません。アプリケーションが ActiveSync 用 Mobile Link プロバイダに呼び出されたときに、アプリケーションがすでに動作していた場合、**wParam** の値は 0 になります。アプリケーションが動作を継続する必要があれば、**wParam** パラメータを無視できます。

プロバイダがサポートされているプラットフォームを確認するには、「プラットフォーム別 SQL Anywhere コンポーネント」を参照してください。

同期の追加方法は、Windows API を直接使用しているか、Microsoft Foundation Classes を使用しているかによって異なります。ここでは、両方の開発モデルについて説明します。

ActiveSync 同期の追加 (Windows API の場合)

Windows API に対して直接プログラミングしている場合は、Mobile Link プロバイダからのメッセージをアプリケーションの **WindowProc** 関数で処理します。メッセージを受信したかどうかを判断するには、**ULIsSynchronizeMessage** 関数を使用します。

次の例は、メッセージの処理方法を示しています。

```
LRESULT CALLBACK WindowProc( HWND hwnd,
        UINT uMsg,
        WPARAM wParam,
        LPARAM lParam )
{
  if( ULIsSynchronizeMessage( uMsg ) ) {
    DoSync();
    if( wParam == 1 ) DestroyWindow( hWnd );
    return 0;
  }
  switch( uMsg ) {
  // code to handle other windows messages
  default:
    return DefWindowProc( hwnd, uMsg, wParam, lParam );
  }
  return 0;
}
```

DoSync は実際に ULSynchronize を呼び出す関数です。

「ULIsSynchronizeMessage 関数」 290 ページを参照してください。

ActiveSync 同期の追加 (MFC の場合)

Microsoft Foundation Classes を使用してアプリケーションを開発している場合は、メイン・ダイアログ・クラスかアプリケーション・クラスで同期メッセージを取得できます。ここでは、両方の方法について説明します。

メッセージの通知用に、アプリケーションのカスタム・ウィンドウ・クラス名を作成し、登録しておいてください。「アプリケーションに対するクラス名の割り当て」 90 ページを参照してください。

♦ メイン・ダイアログ・クラスで ActiveSync 同期を追加するには、次の手順に従います。

1. 登録したメッセージを追加し、メッセージ・ハンドラを宣言します。

   メイン・ダイアログのソース・ファイルでメッセージ・マップを検索します (名前は *CMyAppDlg.cpp* と同じ形式です)。次の例のように、登録したメッセージを **static** を使用して追加し、メッセージ・ハンドラを ON_REGISTERED_MESSAGE を使用して宣言します。

   ```
   static UINT WM_ULTRALITE_SYNC_MESSAGE =
    ::RegisterWindowMessage( UL_AS_SYNCHRONIZE );
   ```

```
BEGIN_MESSAGE_MAP(CMyAppDlg, CDialog)
  //{{AFX_MSG_MAP(CMyAppDlg)
  //}}AFX_MSG_MAP
 ON_REGISTERED_MESSAGE( WM_ULTRALITE_SYNC_MESSAGE,
  OnDoUltraLiteSync )
END_MESSAGE_MAP()
```

2. メッセージ・ハンドラを実装します。

   次のシグネチャで、メイン・ダイアログ・クラスにメソッドを追加します。ActiveSync 用 Mobile Link プロバイダがアプリケーションの同期を要求するときは、常にこのメソッドが自動的に実行されます。このメソッドで、**ULSynchronize** を呼び出してください。

   ```
   LRESULT CMyAppDlg::OnDoUltraLiteSync(
     WPARAM wParam,
     LPARAM lParam
   );
   ```

   この関数の戻り値は 0 にしてください。

   同期メッセージの処理の詳細については、「ULIsSynchronizeMessage 関数」 290 ページを参照してください。

♦ アプリケーション・クラスで ActiveSync 同期を追加するには、次の手順に従います。

1. アプリケーション・クラスの**クラス・ウィザード**を開きます。
2. **[メッセージ]** リストで、**PreTranslateMessage** を強調表示して **[関数の追加]** をクリックします。
3. **[コードの編集]** をクリックします。PreTranslateMessage 関数が表示されます。それを次のように変更します。

   ```
   BOOL CMyApp::PreTranslateMessage(MSG* pMsg)
   {
    if( ULIsSynchronizeMessage(pMsg->message) ) {
     DoSync();
     // close application if launched by provider
     if( pMsg->wParam == 1 ) {
      ASSERT( AfxGetMainWnd() != NULL );
      AfxGetMainWnd()->SendMessage( WM_CLOSE );
     }
     return TRUE; // message has been processed
    }
    return CWinApp::PreTranslateMessage(pMsg);
   }
   ```

   DoSync は実際に ULSynchronize を呼び出す関数です。

   同期メッセージの処理の詳細については、「ULIsSynchronizeMessage 関数」 290 ページを参照してください。

Windows Mobile からの TCP/IP、HTTP、HTTPS 同期

TCP/IP、HTTP、または HTTPS では、同期のタイミングをアプリケーションが制御します。ユーザが同期を要求できるように、アプリケーションにはメニュー項目かユーザ・インタフェースを用意してください。

サンプルの eMbedded Visual C++ プロジェクト

サンプルの eMbedded Visual C++ プロジェクトが、*samples-dir¥UltraLite¥CEStarter* ディレクトリにあります。ワークスペース・ファイルは *samples-dir¥UltraLite¥CEStarter¥ul_wceapplication.vcw* です。

Ultra Light アプリケーション用に eMbedded Visual C++ を使用するには、プロジェクト設定を次のように変更してください。CEStarter アプリケーションの設定は、すでにそのように変更されています。

- コンパイラの設定
  - インクルード・パスに *$(SQLANY11)¥SDK¥Include* を追加します。
  - 適切なコンパイラ・ディレクティブを定義します。たとえば、eMbedded Visual C++ プロジェクトに対して UNDER_CE マクロを定義してください。
- リンカの設定
  - "*$(SQLANY11)¥ultralite¥ce¥processor¥lib¥ulrt.lib*" を追加します。

    processor は、アプリケーションのターゲット・プロセッサです。
  - *winsock.lib* を追加します。
- *.sqc* ファイルを次のように処理します (Embedded SQL の場合のみ)。
  - *ul_database.sqc* と *ul_database.cpp* をプロジェクトに追加します。
  - *.sqc* ファイルに、次のカスタム・ビルドのステップを追加します。

    ```
    "$(SQLANY11)¥Bin32¥sqlpp" -q $(InputPath) ul_database.cpp
    ```
  - 出力ファイルを *ul_database.cpp* に設定します。
  - *ul_database.cpp* のプリコンパイルされているヘッダの使用を無効にします。

API リファレンス

この項では、Ultra Light C/C++ プログラミングに必要な API リファレンスを提供します。

Ultra Light C/C++ 共通 API リファレンス

目次

この項では、Embedded SQL または C++ インタフェースで使用できる関数とマクロについて説明します。この項で説明するほとんどの関数には、「SQLCA (SQL Communications Area) の概要」 5 ページに記載されている SQLCA (SQL Communications Area) が必要です。

ULRegisterErrorCallback のコールバック関数

Ultra Light ランタイムがアプリケーションに通知するエラーを処理します。

この方法によるエラー処理の詳細については、「ULRegisterErrorCallback 関数」 122 ページを参照してください。

構文

```
ul_error_action UL_GENNED_FN_MOD error-callback-function (
SQLCA * sqlca,
ul_void * user_data,
ul_char * buffer
);
```

パラメータ

- **error-callback-function**　関数の名前。ULRegisterErrorCallback に名前を指定します。
- **sqlca**　SQLCA (SQL communications area) へのポインタ。

  SQLCA には、SQL コードが sqlca->sqlcode の形式で含まれています。エラー・パラメータは、すでに SQLCA から取り出され、*buffer* に格納されています。

  この sqlca ポインタは、アプリケーションの SQLCA を必ずしも指しません。また、Ultra Light へのコールバックに使用することはできません。これは、SQL コードをコールバックに伝達するためにのみ使用します。

  C++ コンポーネントの場合は、Sqlca.GetCA() メソッドを使用します。
- **user_data**　ULRegisterErrorCallback に提供されるユーザ・データ。このデータが Ultra Light によって変更されることはありません。コールバック関数はアプリケーション内の任意の位置で通知されるため、user_data 引数がグローバル変数を作成する代替の方法です。
- **buffer**　コールバック関数を登録したときに指定されたバッファ。バッファには、エラー・メッセージの代入パラメータを含む文字列が設定されます。Ultra Light をできるだけ小さくするために、Ultra Light では、エラー・メッセージ自体を提供することはありません。代入パラメータは、個々のエラーによって異なります。SQL エラーのエラー・パラメータの詳細については、「SQL Anywhere のエラー・メッセージ」 『エラー・メッセージ』を参照してください。

戻り値

次のいずれかのアクションが返されます。

- **UL_ERROR_ACTION_CANCEL**　エラーを引き起こした操作をキャンセルします。
- **UL_ERROR_ACTION_CONTINUE**　エラーを引き起こした操作を無視して、実行を続けます。
- **UL_ERROR_ACTION_DEFAULT**　エラー・コールバックがない場合と同じように動作します。
- **UL_ERROR_ACTION_TRY_AGAIN**　エラーを引き起こした操作をリトライします。

参照

- 「ULRegisterErrorCallback 関数」 122 ページ
- 「SQL Anywhere のエラー・メッセージ (Sybase エラー・コード順)」 『エラー・メッセージ』

ULRegisterSQLPassthroughCallback のコールバック関数

このコールバックは、SQL パススルー・スクリプトの実行時に、スクリプト実行の進行状況 (ステータス) を提供します。

構文

void ul_sql_passthrough_observer_fn(*ul_sql_passthrough_status * status*);

パラメータ

- **ul_sql_passthrough_status**　スクリプト実行の現在のステータスを提供します。これには、状態、実行するスクリプトの数、現在実行されているスクリプト、登録呼び出しで提供されたユーザ・データが含まれます。

例

進行状況 observer コールバックは、次のように定義されます。

```
typedef void(UL_CALLBACK_FN * ul_sql_passthrough_observer_fn)
   (ul_sql_passthrough_status * status);
```

ul_sql_passthrough_status 構造体は、次のように定義されます。

```
typedef struct {
   ul_sql_passthrough_state       state; // current state
   ul_u_long   script_count;         // total number of scripts to execute
   ul_u_long   cur_script;           // current script being executed (1-based)
   ul_bool     stop;                 // set to true to stop script execution
                                     // can only be set in the starting state
   ul_void *   user_data;            // user data provided in register call
   SQLCA *     sqlca;
} ul_sql_passthrough_status;
```

進行状況 observer コールバックは、次のメソッドによって登録されます。

```
UL_FN_SPEC ul_ret_void UL_FN_MOD ULRegisterSQLPassthroughCallback(
SQLCA *                        sqlca,
ul_sql_passthrough_observer_fn  callback,
ul_void *                      user_data );
```

ULRegisterSQLPassthroughCallback を呼び出すコールバックとコードのサンプルを次に示します。

```
static void UL_GENNED_FN_MOD passthroughCallback(ul_sql_passthrough_status * status) {
   switch( status->state ) {
      case UL_SQL_PASSTHROUGH_STATE_STARTING:
         printf( "SQL Passthrough script execution starting¥n" );
         break;
      case UL_SQL_PASSTHROUGH_STATE_RUNNING_SCRIPT:
         printf( "Executing script %d of %d¥n", status->cur_script, status->script_count );
         break;
      case UL_SQL_PASSTHROUGH_STATE_DONE:
         printf( "Finished executing SQL Passthrough scripts¥n" );
         break;
      default:
         printf( "SQL Passthrough script execution has failed¥n" );
         break;
```



```
    }
}

int main() {
    ULSqlca sqlca;

    sqlca.Initialize();
    ULRegisterSQLPassthroughCallback( sqlca.GetCA(), passthroughCallback, NULL );
    DatabaseManager * dm = ULInitDatabaseManager( sqlca );
    ...
}
```

戻り値

次のいずれかのアクションが返されます。

- **UL_ERROR_ACTION_CANCEL**　エラーを引き起こした操作をキャンセルします。
- **UL_ERROR_ACTION_CONTINUE**　エラーを引き起こした操作を無視して、実行を続けます。
- **UL_ERROR_ACTION_DEFAULT**　エラー・コールバックがない場合と同じように動作します。
- **UL_ERROR_ACTION_TRY_AGAIN**　エラーを引き起こした操作をリトライします。

参照

- 「ULRegisterErrorCallback 関数」 122 ページ
- 「SQL Anywhere のエラー・メッセージ (Sybase エラー・コード順)」 『エラー・メッセージ』

MLFileTransfer 関数

Mobile Link インタフェースを使用して、Mobile Link サーバからファイルをダウンロードします。

構文

ul_bool **MLFileTransfer (** ml_file_transfer_info * *info* **);**

パラメータ

info　ファイル転送情報が格納された構造体。

ML File Transfer パラメータ

ML File Transfer パラメータは、MLFileTransfer 関数にパラメータとして渡される構造体のメンバです。ml_file_transfer_info 構造体は、ヘッダ・ファイル *mlfiletransfer.h* に定義されています。構造体の各フィールドの定義は次のとおりです。

filename　必須。Mobile Link を実行しているサーバから転送されるファイルの名前。MobiLink は *username* サブディレクトリを検索してから、デフォルトのルート・ディレクトリを検索します。「-ftr オプション」 『Mobile Link - サーバ管理』を参照してください。

ファイルが見つからなかった場合は、error フィールドにエラーが設定されます。ファイル名にはドライブまたはパスの情報を含めないでください。そのような情報を含めると、ファイルが見えなくなります。

dest_path　ダウンロード・ファイルの格納先となるローカル・パス。このパラメータが空の場合 (デフォルト)、ダウンロード・ファイルは現在のディレクトリに格納されます。

- Windows Mobile では、dest_path が空の場合、ファイルはデバイスのルート (¥) ディレクトリに格納されます。
- デスクトップ・コンピュータでは、dest_path が空の場合、ファイルはユーザの現在のディレクトリに格納されます。
- Palm OS では、デバイスの外部記憶領域にダウンロードする場合、dest_path に **vfs:** というプレフィクスを付けてください。このプレフィクスの後に、プラットフォームのファイル命名規則に従ってパスを指定します。「Palm OS」 『Ultra Light データベース管理とリファレンス』を参照してください。

  dest_path フィールドが空の場合、MLFileTransfer は、ダウンロード対象が Palm レコード・データベース (*.pdb*) であると想定します。

dest_filename　ダウンロード・ファイルのローカル名。このパラメータが空の場合、ファイル名の値が使用されます。

stream　必須。protocol には、TCPIP、TLS、HTTP、HTTPS のいずれか 1 つを指定します。「Stream Type 同期パラメータ」 『Ultra Light データベース管理とリファレンス』を参照してください。

stream_parms　指定されたストリームのプロトコルのオプション。「Ultra Light 同期ストリームのネットワーク・プロトコルのオプション」 『Ultra Light データベース管理とリファレンス』を参照してください。

username　必須。Mobile Link ユーザ名。

password　Mobile Link ユーザ名のパスワード。

version　必須。Mobile Link スクリプトのバージョン。

observer　'observer' フィールドを使用することで、ファイルのダウンロードの進捗状況を確認するコールバックを実現できます。詳細については、後述のコールバック関数の説明を参照してください。

user_data　同期 observer で使用できるようにした、アプリケーション固有の情報。「User Data 同期パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。

force_download　true に設定すると、タイムスタンプからそのファイルが既に存在すると判断される場合でもダウンロードされます。false に設定すると、ファイルはサーバのバージョンとローカルのバージョンが異なる場合にダウンロードされます。この場合、サーバのバージョンによってローカル・クライアントのバージョンが上書きされます。クライアント上に同じ名前のファイルがあると、そのファイルが破棄されてからダウンロードされます。MLFileTransfer は、ファイルのサーバとクライアントのバージョンを比較するために、各ファイルの暗号化ハッシュ値を計算します。ハッシュ値は、ファイルの内容がまったく同じ場合にだけ同じ値になります。

enable_resume　true に設定すると、MLFileTransfer は、通信エラーまたはユーザのキャンセルによって中断した以前のダウンロードを再開します。サーバ上のファイルがローカルの部分ファイルより新しい場合、部分ファイルが破棄され、新しいバージョンがあらためてダウンロードされます。このパラメータより force_download パラメータが優先されます。

num_auth_parms　Mobile Link イベントの認証パラメータに渡される認証パラメータの数。「Number of Authentication Parameters パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。

auth_parms　Mobile Link イベントの認証パラメータにパラメータを渡します。「Authentication Parameters 同期パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。

downloaded_file　次のいずれかに設定されます。

- ファイルが正常にダウンロードされた場合は 1。
- エラーが発生した場合は 0。MLFileTransfer の呼び出し時にファイルがすでに最新の状態になっていると、エラーが発生します。この関数は、false ではなく true を返します。Palm OS でレコード・データベース (*.pdb*) ファイルをダウンロードする場合、ファイルは最新かどうかに関係なく必ずダウンロードされます。

auth_status　Mobile Link のユーザ認証のステータスをレポートします。Mobile Link サーバが、この情報をクライアントに提供します。「Authentication Status 同期パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。

auth_value　カスタム Mobile Link のユーザ認証スクリプトの結果をレポートします。Mobile Link サーバが、この情報をクライアントに提供します。「Authentication Value 同期パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。

file_auth_code　サーバ上の authenticate_file_transfer スクリプト (オプション) のリターン・コードが格納されます。

error　発生したエラーに関する情報が格納されます。

戻り値

- **ul_true**　ファイルが正常にダウンロードされました。
- **ul_false**　ファイルが正常にダウンロードされませんでした。ml_file_transfer_info 構造体のエラー・フィールドにエラー情報を格納できます。不完全なファイル転送は再開可能です。

備考

転送対象ファイルのソース・ロケーションを設定する必要があります。このロケーションは、Mobile Link サーバ上の Mobile Link ユーザ・ディレクトリ (または Mobile Link サーバ上のデフォルト・ディレクトリ) を指定する必要があります。また、ファイルのターゲット・ロケーションとファイル名を設定することもできます。

たとえば、新しいデータベースまたは置き換えるデータベースを Mobile Link サーバからダウンロードするようにアプリケーションをプログラミングできます。検索される最初のロケーションは各ユーザのサブディレクトリなので、ユーザごとにファイルをカスタマイズできます。サーバのルート・フォルダは、指定ファイルがユーザのフォルダになかった場合に検索されるロケーションなので、デフォルトの転送ファイルは、サーバのルート・フォルダに配置することもできます。

コールバック関数

ファイル転送の進捗状況を observer パラメータを使用して確認するコールバックのプロトタイプは次のとおりです。

```
typedef void(*ml_file_transfer_observer_fn)( ml_file_transfer_status * status );
```

コールバックに渡される ml_file_transfer_status オブジェクトは次のように定義されます。

```
typedef struct ml_file_transfer_status {
    asa_uint64              file_size;
    asa_uint64              bytes_received;
    asa_uint64              resumed_at_size;
    ml_file_transfer_info_a *   info;
    asa_uint16              flags;
    asa_uint8               stop;
} ml_file_transfer_status;
```

file_size　ダウンロード中のファイルの合計サイズ (バイト単位)。

bytes_received　現時点でダウンロード済みのファイルのサイズを示します。再開されたダウンロードの場合は、以前の同期も含みます。

resumed_at_size　ダウンロードの再開で使用され、現在のダウンロードの開始点を示します。

info　MLFileTransfer に渡された info オブジェクトへのポインタ。このポインタを使用して user_data パラメータにアクセスできます。

flags　追加情報が格納されます。MLFileTransfer がネットワーク呼び出しをブロックしており、observer 関数が前回呼び出されたときからダウンロード・ステータスが変わっていない場合、値 MLFT_STATUS_FLAG_IS_BLOCKING が設定されます。

stop　true に設定すると、現在のダウンロードをキャンセルできます。enable_resume パラメータが設定された場合にかぎり、MLFileTransfer を後で呼び出したときにそのダウンロードを再開できます。

ULCreateDatabase 関数

Ultra Light データベースを作成します。

構文

```
ul_bool ULCreateDatabase ( SQLCA * sqlca,
 ul_char * connection-parms,
 void const  * collation,
 ul_char * creation-parms,
 void * reserved
 );
```

パラメータ

sqlca　初期化済み SQLCA へのポインタ。

connection-parms　接続パラメータをセミコロンで区切った文字列。キーワードと値のペアとして設定されます。接続文字列には、データベースの名前を含める必要があります。ここに含まれるパラメータは、データベースの接続時に指定されるパラメータ・セットと同じです。完全なリストについては、「Ultra Light 接続パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。

collation
データベースの希望の照合順。照合順は適切な関数を呼び出して取得できます。次に例を示します。

```
void const * collation = ULGetCollation_1250LATIN2();
```

関数名は、使用する照合の名前に **ULGetCollation_** というプレフィクスを付けたものになっています。使用可能なすべての照合関数のリストについては、 *install-dir¥SDK¥Include¥ulgetcoll.h* を参照してください。**ULGetCollation_** 関数を呼び出すプログラムでは、このファイルをインクルードする必要があります。

creation-parms
データベース作成パラメータをセミコロンで区切った文字列。キーワードと値のペアとして設定されます。次に例を示します。

```
page_size=2048;obfuscate=yes
```

完全なリストについては、「Ultra Light で使用するデータベース作成パラメータの選択」『Ultra Light データベース管理とリファレンス』を参照してください。

reserved　このパラメータは、今後の使用のために予約されています。

戻り値

- **ul_true**　データベースが正常に作成されたことを示します。
- **ul_false**　詳細なエラー・メッセージが、SQLCA の SQLCODE フィールドに設定されています。通常、無効なファイル名やアクセスの拒否によって発生します。

備考

2 セットのパラメータで指定される情報を使用してデータベースが作成されます。

- connection-parms は標準の接続パラメータで、データベースがアクセスされるときは常に適用できます (ファイル名、ユーザ ID、パスワード、省略可能な暗号化キーなど)。
- creation-parms は、データベースの作成時のみに関係するパラメータです (難読化、ページサイズ、日時の形式など)。

アプリケーションでこの関数を呼び出すことができるのは、SQLCA の初期化後です。

例

次のコードは、ULCreateDatabase を使用して、ファイル *C:¥myfile.udb* に Ultra Light データベースを作成します。

```
if( ULCreateDatabase(&sqlca
    ,UL_TEXT("DBF=C:¥myfile.udb;uid=DBA;pwd=sql")
    ,ULGetCollation_1250LATIN2()
    ,UL_TEXT("obfuscate=1;page_size=8192")
    ,NULL)
{
    // success
};
```

ULEnableEccSyncEncryption 関数

SSL ストリームまたは TLS ストリームの ECC 暗号化を有効にします。これは、ストリーム・パラメータを TLS または HTTPS に設定するときに必要です。また、この場合は同期パラメータ tls_type を ECC として設定する必要があります。

構文

void **ULEnableEccSyncEncryption(** SQLCA * *sqlca* **);**

パラメータ

sqlca　初期化済み SQLCA へのポインタ。

C++ API では、Sqlca.GetCA メソッドを使用します。

備考

この関数は、C++ API アプリケーションと Embedded SQL アプリケーションで使用できます。この関数を呼び出してから Synchronize 関数を呼び出すようにしてください。同期タイプを有効にする呼び出しを実行する前に同期しようとすると、エラー SQLE_METHOD_CANNOT_BE_CALLED が発生します。

別途ライセンスが必要な必須コンポーネント
ECC 暗号化と FIPS 認定の暗号化には、別途ライセンスが必要です。強力な暗号化テクノロジはすべて、輸出規制対象品目です。

「別途ライセンスが必要なコンポーネント」『SQL Anywhere 11 - 紹介』を参照してください。

参照

- 「ULEnableZlibSyncCompression 関数」 119 ページ
- 「ULEnableRsaFipsSyncEncryption 関数」 114 ページ

ULEnableFIPSStrongEncryption 関数

データベースに対して FIPS ベースの強力な暗号化を有効にします。この関数を呼び出すと、適切な暗号化ルーチンがアプリケーションにインクルードされ、アプリケーションのコード・サイズがその分だけ増加します。

構文

void **ULEnableFIPSStrongEncryption(** SQLCA * *sqlca* **);**

パラメータ

sqlca　初期化済み SQLCA へのポインタ。

C++ API では、Sqlca.GetCA メソッドを使用します。

備考

この関数は、C++ API アプリケーションと Embedded SQL アプリケーションで使用できます。この関数を呼び出してから Synchronize 関数を呼び出すようにしてください。同期タイプを有効にする呼び出しを実行する前に同期しようとすると、エラー SQLE_METHOD_CANNOT_BE_CALLED が発生します。

別途ライセンスが必要な必須コンポーネント
ECC 暗号化と FIPS 認定の暗号化には、別途ライセンスが必要です。強力な暗号化テクノロジはすべて、輸出規制対象品目です。

「別途ライセンスが必要なコンポーネント」『SQL Anywhere 11 - 紹介』を参照してください。

参照

● 「ULEnableStrongEncryption 関数」 116 ページ

ULEnableHttpSynchronization 関数

HTTP 同期を有効にします。

構文

void **ULEnableHttpSynchronization(** SQLCA * *sqlca* **);**

パラメータ

sqlca　初期化済み SQLCA へのポインタ。

C++ API では、Sqlca.GetCA メソッドを使用します。

備考

この関数は、C++ API アプリケーションと Embedded SQL アプリケーションで使用できます。この関数を呼び出してから Synchronize 関数を呼び出すようにしてください。同期タイプを有効にする呼び出しを実行する前に同期しようとすると、エラー SQLE_METHOD_CANNOT_BE_CALLED が発生します。

ULEnableHttpsSynchronization 関数

HTTP の SSL 同期ストリームを有効にします。

構文

void **ULEnableHttpsSynchronization(** SQLCA * *sqlca* **);**

パラメータ

sqlca　初期化済み SQLCA へのポインタ。

C++ API では、Sqlca.GetCA メソッドを使用します。

備考

この関数は、C++ API アプリケーションと Embedded SQL アプリケーションで使用できます。この関数を呼び出してから Synchronize 関数を呼び出すようにしてください。同期タイプを有効にする呼び出しを実行する前に同期しようとすると、エラー SQLE_METHOD_CANNOT_BE_CALLED が発生します。

例

```
ULEnableHttpsSynchronization( sqlca );
ULEnableRsaSyncEncryption( sqlca );
synch_info.stream = "https";
synch_info.stream_parms = "tls_type=rsa";    // rsa is default, so setting this parameter is optional
conn->Synchronize( sqlca );
```


ULEnableRsaFipsSyncEncryption 関数

SSL ストリームまたは TLS ストリームの RSA FIPS 暗号化を有効にします。これは、ストリーム・パラメータを TLS または HTTPS に設定するときに必要です。この場合、同期パラメータ tls_type を RSA として設定する必要もあります。

構文

void **ULEnableRsaFipsSyncEncryption(** SQLCA * *sqlca* **);**

パラメータ

sqlca　初期化済み SQLCA へのポインタ。

C++ API では、Sqlca.GetCA メソッドを使用します。

備考

この関数は、C++ API アプリケーションと Embedded SQL アプリケーションで使用できます。この関数を呼び出してから Synchronize 関数を呼び出すようにしてください。同期タイプを有効にする呼び出しを実行する前に同期しようとすると、エラー SQLE_METHOD_CANNOT_BE_CALLED が発生します。

参照

- 「ULEnableRsaSyncEncryption 関数」 115 ページ
- 「ULEnableEccSyncEncryption 関数」 110 ページ

ULEnableRsaSyncEncryption 関数

SSL ストリームまたは TLS ストリームの RSA 暗号化を有効にします。これは、ストリーム・パラメータを TLS または HTTPS に設定するときに必要です。この場合、同期パラメータ tls_type を RSA として設定する必要もあります。

構文

void **ULEnableRsaSyncEncryption(** SQLCA * *sqlca* **);**

パラメータ

sqlca　初期化済み SQLCA へのポインタ。

C++ API では、Sqlca.GetCA メソッドを使用します。

備考

この関数は、C++ API アプリケーションと Embedded SQL アプリケーションで使用できます。この関数を呼び出してから Synchronize 関数を呼び出すようにしてください。同期タイプを有効にする呼び出しを実行する前に同期しようとすると、エラー SQLE_METHOD_CANNOT_BE_CALLED が発生します。

参照

- 「ULEnableEccSyncEncryption 関数」 110 ページ
- 「ULEnableRsaFipsSyncEncryption 関数」 114 ページ

ULEnableStrongEncryption 関数

強力な暗号化を有効にします。

構文

void **ULEnableStrongEncryption(** SQLCA * *sqlca* **);**

パラメータ

sqlca　初期化済み SQLCA へのポインタ。

C++ API では、Sqlca.GetCA メソッドを使用します。

備考

この関数は、C++ API アプリケーションと Embedded SQL アプリケーションで使用できます。この関数を呼び出してから db_init または ULInitDatabaseManager を呼び出すようにしてください。

注意
この関数を呼び出すと、暗号化ルーチンがアプリケーションにインクルードされ、アプリケーションのコード・サイズがその分だけ増加します。

ULEnableTcpipSynchronization 関数

TCP/IP 同期を有効にします。

構文

void **ULEnableTcpipSynchronization(** SQLCA * *sqlca* **);**

パラメータ

sqlca　初期化済み SQLCA へのポインタ。

C++ API では、Sqlca.GetCA メソッドを使用します。

備考

この関数は、C++ API アプリケーションと Embedded SQL アプリケーションで使用できます。この関数を呼び出してから Synchronize 関数を呼び出すようにしてください。同期タイプを有効にする呼び出しを実行する前に同期しようとすると、エラー SQLE_METHOD_CANNOT_BE_CALLED が発生します。

ULEnableTlsSynchronization 関数

TLS 同期を有効にします。

構文

void **ULEnableTlsSynchronization(** SQLCA * *sqlca* **);**

パラメータ

sqlca　初期化済み SQLCA へのポインタ。

C++ API では、Sqlca.GetCA メソッドを使用します。

備考

この関数は、C++ API アプリケーションと Embedded SQL アプリケーションで使用できます。この関数を呼び出してから Synchronize 関数を呼び出すようにしてください。同期タイプを有効にする呼び出しを実行する前に同期しようとすると、エラー SQLE_METHOD_CANNOT_BE_CALLED が発生します。

ULEnableZlibSyncCompression 関数

同期ストリームの ZLIB 圧縮を有効にします。

構文

void **ULEnableZlibSyncCompression(** SQLCA * *sqlca* **);**

パラメータ

sqlca　初期化済み SQLCA へのポインタ。

C++ API では、Sqlca.GetCA メソッドを使用します。

備考

この関数は、C++ API アプリケーションと Embedded SQL アプリケーションで使用できます。この関数を呼び出してから Synchronize 関数を呼び出すようにしてください。同期タイプを有効にする呼び出しを実行する前に同期しようとすると、エラー SQLE_METHOD_CANNOT_BE_CALLED が発生します。

ULInitDatabaseManager 関数

Ultra Light データベース・マネージャを初期化します。

構文

pointer **ULInitDatabaseManager(** SQLCA * *sqlca* **);**

パラメータ

sqlca　初期化済み SQLCA へのポインタ。

C++ API では、Sqlca.GetCA メソッドを使用します。

戻り値

- データベース・マネージャへのポインタ。
- 関数が失敗した場合は NULL。

備考

データベースが以前に初期化されておらず、シャットダウンを発行しなかった場合、この関数は失敗します。

ULInitDatabaseManagerNoSQL 関数

Ultra Light データベース・マネージャを初期化し、SQL 文処理のサポートを除外します (これによりアプリケーションのランタイム・サイズを大幅に減少できます)。

構文

pointer **ULInitDatabaseManagerNoSQL(** SQLCA * *sqlca* **);**

パラメータ

sqlca　初期化済み SQLCA へのポインタ。

C++ API では、Sqlca.GetCA メソッドを使用します。

戻り値

- データベース・マネージャへのポインタ。
- 関数が失敗した場合は NULL。

備考

データベースが以前に初期化されておらず、シャットダウンを発行しなかった場合、この関数は失敗します。

アプリケーションは、データへのアクセスに Table API を使用する必要があります。SQL 文は使用できません。データベース・スキーマにパブリケーションの述部が含まれている場合、この呼び出しは使用できません。代わりに ULInitDatabaseManager を使用してください。

ULRegisterErrorCallback 関数

エラーを処理するコールバック関数を登録します。

構文

```
void ULRegisterErrorCallback (
   SQLCA * sqlca,
   ul_error_callback_fn callback,
   ul_void *   user_data,
   ul_char *   buffer,
   size_t   len
);
```

パラメータ

- **sqlca**　SQLCA へのポインタ。

  C++ API では、Sqlca.GetCA メソッドを使用します。

- **callback**　コールバック関数の名前。この関数のプロトタイプの詳細については、「ULRegisterErrorCallback のコールバック関数」 100 ページを参照してください。

  コールバック値に UL_NULL を指定すると、以前に登録したコールバック関数が無効になります。

- **user_data**　グローバル変数の代わりに、コンテキスト情報をグローバルにアクセスできるようにします。コールバック関数はアプリケーションの任意のロケーションから呼び出すことができるため、このパラメータは必須です。提供するデータが Ultra Light によって変更されることはありません。コールバック関数の起動時に、データだけが渡されます。

  データ型を宣言し、コールバック関数内の適切なデータ型にキャストできます。たとえば、コールバック関数に次の形式の行を指定できます。

  ```
  MyContextType * context = (MyContextType *)user_data;
  ```

- **buffer**　NULL ターミネータを含む、エラー・メッセージの代入パラメータが格納されている文字配列。Ultra Light をできるだけ小さくするために、Ultra Light では、エラー・メッセージ自体を提供することはありません。代入パラメータは、個々のエラーによって異なります。完全なリストについては、「SQL Anywhere のエラー・メッセージ」 『エラー・メッセージ』を参照してください。

  バッファは、Ultra Light がアクティブな間、存在します。パラメータ情報を受け取る必要がない場合は、UL_NULL を指定します。

- **len**　ul_char 文字単位 の buffer (前述のパラメータ) の長さ。値が 100 であれば、ほとんどの場合、エラー・パラメータを保持するのに十分な長さです。バッファが小さすぎる場合、パラメータはトランケートされます。

備考

この関数を呼び出すと、Ultra Light がエラーを通知するたびに、ユーザ指定のコールバック関数が呼び出されます。このため、SQLCA を初期化した直後に ULRegisterErrorCallback を呼び出してください。

このコールバック関数を使用するエラー処理では、発生するすべてのエラーがアプリケーションに通知されるため、開発中は特に効果的です。ただし、コールバック関数は実行フローを制御するわけではないので、アプリケーションでは Ultra Light 関数の呼び出し後に必ず SQLCA の SQLCODE フィールドを確認してください。

例

次のコードは、Ultra Light C++ コンポーネント・アプリケーションのコールバック関数を登録します。

```
int main() {
    ul_char buffer[100];
    DatabaseManager * dm;
    Connection * conn;
    Sqlca.Initialize();
    ULRegisterErrorCallback(
        Sqlca.GetCA(),
        MyErrorCallBack,
        UL_NULL,
        buffer,
        sizeof (buffer) );
    dm = ULInitDatabaseManager( Sqlca );
    ...
}
```

コールバック関数のサンプルを次に示します。

```
ul_error_action UL_GENNED_FN_MOD MyErrorCallBack(
    SQLCA *     Sqlca,
    ul_void *   user_data,
    ul_char *   message_param )
{
    ul_error_action rc = 0;
    (void) user_data;

    switch( Sqlca->sqlcode ) {
        // The following error is used for flow control - don't report it here
        case SQLE_NOTFOUND:
            break;
        case SQLE_ULTRALITE_DATABASE_NOT_FOUND:
            _tprintf( _TEXT( "Error %ld: Database file %s not found¥n" ), Sqlca->sqlcode,
message_param );
            break;
        default:
            _tprintf( _TEXT( "Error %ld: %s¥n" ), Sqlca->sqlcode, message_param );
        break;
     }
    return rc;
}
```

参照

- 「SQL Anywhere のエラー・メッセージ (Sybase エラー・コード順)」 『エラー・メッセージ』
- 「ULRegisterErrorCallback のコールバック関数」 100 ページ

ULRegisterSQLPassthroughCallback

現在のステータスを提供するコールバック関数を登録します。

構文

```
void ULRegisterSQLPassthroughCallback (
   SQLCA * sqlca,
   ul_sql_passthrough_observer_fncallback,
   ul_void *   user_data
);
```

パラメータ

- **sqlca**　SQLCA へのポインタ。

  C++ API では、Sqlca.GetCA メソッドを使用します。

- **callback**　コールバック関数の名前。

  コールバック値に UL_NULL を指定すると、以前に登録したコールバック関数が無効になります。

- **user_data**　グローバル変数の代わりに、コンテキスト情報をグローバルにアクセスできるようにします。コールバック関数はアプリケーションの任意のロケーションから呼び出すことができるため、このパラメータは必須です。提供するデータが Ultra Light によって変更されることはありません。コールバック関数の起動時に、データだけが渡されます。

  データ型を宣言し、コールバック関数内の適切なデータ型にキャストできます。たとえば、コールバック関数に次の形式の行を指定できます。

  ```
  MyContextType * context = (MyContextType *) user_data;
  ```

例

ULRegisterSQLPassthroughCallback を呼び出すコールバックとコードのサンプルを次に示します。

```
static void UL_GENNED_FN_MOD passthroughCallback( ul_sql_passthrough_status * status ) {
   switch( status->state ) {
      case UL_SQL_PASSTHROUGH_STATE_STARTING:
         printf( "SQL Passthrough script execution starting¥n" );
         break;
      case UL_SQL_PASSTHROUGH_STATE_RUNNING_SCRIPT:
         printf( "Executing script %d of %d¥n", status->cur_script, status->script_count );
         break;
      case UL_SQL_PASSTHROUGH_STATE_DONE:
         printf( "Finished executing SQL Passthrough scripts¥n" );
         break;
      default:
         printf( "SQL Passthrough script execution has failed¥n" );
         break;
   }
}

int main() {
   ULSqlca sqlca;

   sqlca.Initialize();
   ULRegisterSQLPassthroughCallback( sqlca.GetCA(), passthroughCallback, NULL );
```

```
    DatabaseManager * dm = ULInitDatabaseManager( sqlca );
    ...
}
```

ULRegisterSynchronizationCallback

SQL SYNCHRONIZE 文によって同期が実行されたときに呼び出される関数を登録します。同期コールバック関数が定義され、Ultra Light に登録されている場合は、SYNCHRONIZE 文を実行するたびに、その同期の進行状況情報がコールバック関数に渡されます。コールバック関数が登録されていない場合、進行状況情報は渡されません。

構文

```
void ULRegisterSynchronizationCallback (
   SQLCA * sqlca,
    ul_synch_observer_fncallback,
   ul_void *   user_data
);
```

パラメータ

- **sqlca**　SQLCA へのポインタ。

  C++ API では、Sqlca.GetCA メソッドを使用します。

- **callback**　コールバック関数の名前。

  コールバック値に UL_NULL を指定すると、以前に登録したコールバック関数が無効になります。

- **user_data**　グローバル変数の代わりに、コンテキスト情報をグローバルにアクセスできるようにします。コールバック関数はアプリケーションの任意のロケーションから呼び出すことができるため、このパラメータは必須です。提供するデータが Ultra Light によって変更されることはありません。コールバック関数の起動時に、データだけが渡されます。

  データ型を宣言し、コールバック関数内の適切なデータ型にキャストできます。たとえば、コールバック関数に次の形式の行を指定できます。

  ```
  MyContextType * context = (MyContextType *) user_data;
  ```

参照

- 「Ultra Light SYNCHRONIZE 文」 『Ultra Light データベース管理とリファレンス』

Ultra Light C/C++ アプリケーションのマクロとコンパイラ・ディレクティブ

特に指定のないかぎり、ディレクティブは Embedded SQL と C++ API の両方のアプリケーションに適用されます。

コンパイラ・ディレクティブは、次の場所で指定できます。

- コンパイラのコマンド・ライン。一般にディレクティブは /D オプションを使用して設定します。たとえば、ユーザ認証を使用する Ultra Light アプリケーションをコンパイルする場合、Microsoft Visual C++ の makefile は、次のようになります。

  ```
  CompileOptions=/c /DPRWIN32 /Od /Zi /DWIN32
  /DUL_USE_DLL

  IncludeFolders= ¥
  /I"$(VCDIR)¥include" ¥
  /I"$(SQLANY11)¥SDK¥Include"

  sample.obj: sample.cpp
   cl $(CompileOptions) $(IncludeFolders) sample.cpp
  ```

  VCDIR は Visual C++ ディレクトリ、*SQLANY11* は SQL Anywhere インストール・ディレクトリです。

- ユーザ・インタフェースのコンパイラ設定ウィンドウ。
- ソース・コード。ディレクティブは #define 文を使用して指定します。

UL_AS_SYNCHRONIZE マクロ

ActiveSync 同期を指定するときに使用するコールバック・メッセージの名前を提供します。

備考

ActiveSync を使用する Windows Mobile アプリケーションのみに適用されます。

参照

- 「アプリケーションへの ActiveSync 同期の追加」 92 ページ

UL_SYNC_ALL マクロ

データベース内のすべてのテーブルを参照するパブリケーション・リスト文字列を提供します。これには、パブリケーションによって明示的に参照されていないものも含まれます。「非同期」とマーク付けされているテーブルは、このリストには含まれません。

参照

- 「ul_synch_info_a 構造体」 135 ページ
- 「ul_synch_info_w2 構造体」 138 ページ
- 「ULGetLastDownloadTime 関数」 283 ページ
- 「ULCountUploadRows 関数」 274 ページ
- 「UL_SYNC_ALL_PUBS マクロ」 128 ページ

UL_SYNC_ALL_PUBS マクロ

パブリケーション内で参照されている、データベース内のすべてのテーブルを参照するパブリケーション・リストを提供します。

参照

- 「ul_synch_info_a 構造体」 135 ページ
- 「ul_synch_info_w2 構造体」 138 ページ
- 「ULGetLastDownloadTime 関数」 283 ページ
- 「ULCountUploadRows 関数」 274 ページ
- 「UL_SYNC_ALL マクロ」 127 ページ

UL_TEXT マクロ

シングルバイト文字列またはワイド文字列としてコンパイルされる定数文字列を準備します。アプリケーションをコンパイルして文字列の Unicode 表現と非 Unicode 表現を使用する場合は、このマクロを使用してすべての定数文字列を囲みます。このマクロは、あらゆる環境やプラットフォームで文字列を適切に定義します。

UL_USE_DLL マクロ

静的ランタイム・ライブラリではなく、ランタイム・ライブラリ DLL を使用するようにアプリケーションを設定します。

備考

Windows Mobile アプリケーションと Windows アプリケーションに適用されます。

UNDER_CE マクロ

デフォルトでは、このマクロは、Microsoft eMbedded Visual C++ コンパイラによるすべての新規 eMbedded Visual C++ プロジェクトの中で定義されます。

備考

Windows Mobile アプリケーションに適用されます。

例

```
/D UNDER_CE=$(CEVersion)
```

参照

- 「Windows Mobile 用 Ultra Light アプリケーションの開発」 83 ページ

UNDER_PALM_OS マクロ

このマクロは、Ultra Light Palm OS アプリケーションにインクルードされる *ulpalmos.h* ヘッダ・ファイルで、Ultra Light プラグインによって定義されます。「Ultra Light plug-in for CodeWarrior の使用」 71 ページを参照してください。

備考

Palm OS のコンパイラ・ディレクティブにのみ適用されます。

参照

- 「Palm OS 用 Ultra Light アプリケーションの開発」 67 ページ

Ultra Light C++ API リファレンス

目次

ul_sql_passthrough_status 構造体

SQL パススルーのステータス情報です。

構文

public **ul_sql_passthrough_status**

プロパティ

名前	型	説明
cur_script	ul_u_long	現在実行されているスクリプト。(1 から始まります)
script_count	ul_u_long	実行されるスクリプトの総数です。
sqlca	SQLCA *	SQLCA へのポインタ。「GetCA 関数」 149 ページを参照してください。
state	ul_sql_passthrough_state	現在のステータスです。「同期ステータス情報の処理」 59 ページを参照してください。
stop	ul_bool	true に設定すると、スクリプトの実行を停止できます。
user_data	ul_void *	登録呼び出しで提供されたユーザ・データです。「ULRegisterSQLPassthroughCallback」 124 ページを参照してください。

ul_stream_error 構造体

同期の通信ストリーム・エラー情報です。

構文

public **ul_stream_error**

プロパティ

名前	型	説明
error_string	char	stream_error_code フィールドの配列です。
stream_error_code	ss_error_code	必須ではありません。値は常に 0 です。
system_error_code	asa_int32	システム固有のエラー・コード。エラー・コードの詳細については、プラットフォームのマニュアルを参照してください。Windows プラットフォームの場合は、Microsoft Developer Network Web サイトを参照してください。

備考

Windows における一般的なシステム・エラーを次に示します。

- 10048 (WSAADDRINUSE) アドレスがすでに使用されています。
- 10053 (WSAECONNABORTED) ソフトウェアが接続のアボートを引き起こしました。
- 10054 (WSAECONNRESET) 通信の他方の側がソケットを閉じました。
- 10060 (WSAETIMEDOUT) 接続がタイムアウトしました。
- 10061 (WSAECONNREFUSED) 接続が拒否されました。通常、Mobile Link サーバが稼働していないか、指定されたポートで受信していません。Microsoft Developer Network Web サイトを参照してください。

ul_synch_info_a 構造体

同期データを記述するために使用される構造体です。

構文

public **ul_synch_info_a**

プロパティ

名前	型	説明
additional_parms	const char *	「キーワード=値」のペアがセミコロンで区切られたリストとして指定された、追加の同期パラメータの文字列です。通常このフィールドには、使用頻度の低い同期パラメータが入ります。「Additional Parameters 同期パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。
auth_parms	char **	Mobile Link イベントの認証パラメータの配列です。
auth_status	ul_auth_status	Mobile Link のユーザ認証のステータスです。Mobile Link サーバが、この情報をクライアントに提供します。
auth_value	ul_s_long	カスタム Mobile Link のユーザ認証スクリプトの結果です。Mobile Link サーバが、この情報をクライアントに提供し、認証ステータスを判断できるようにします。
download_only	ul_bool	現在の同期中は、Ultra Light データベースから変更をアップロードしません。
ignored_rows	ul_bool	無視されたローのステータスです。同期中にスクリプトがないために Mobile Link サーバによってローが 1 つでも無視されると、この読み込み専用フィールドが true をレポートします。
init_verify	ul_synch_info_a *	検証を初期化します。
keep_partial_download	ul_bool	同期時の通信エラーが原因でダウンロードが失敗すると、このパラメータは、変更をロールバックしないで部分的なダウンロードを保持するかどうかを制御します。
new_password	char *	ユーザ名に対する新しい Mobile Link パスワードを指定する文字列です。このパラメータはオプションです。
num_auth_parms	ul_byte	Mobile Link イベントの認証パラメータに渡される認証パラメータの数です。
observer	ul_synch_observer_fn	同期をモニタするコールバック関数またはイベント・ハンドラへのポインタです。このパラメータはオプションです。

名前	型	説明
partial_download_retained	ul_bool	同期時の通信エラーが原因でダウンロードが失敗すると、このパラメータは、変更をロールバックしないで、ダウンロードされたこの変更が適用されたかどうかを示します。
password	char *	ユーザ名に対する既存の Mobile Link パスワードを指定する文字列です。このパラメータはオプションです。
ping	ul_bool	Ultra Light クライアントと Mobile Link サーバ間の通信を確認します。このパラメータが true に設定されている場合は、同期は行われません。
publications	const char *	同期に含めるデータを示す、パブリケーションをカンマで区切ったリストです。
resume_partial_download	ul_bool	失敗したダウンロードを再開します。同期によって変更はアップロードされず、失敗したダウンロードで変更のみがダウンロードされます。
send_column_names	ul_bool	アップロード時にカラム名が Mobile Link サーバに送信されるようアプリケーションに指示します。
send_download_ack	ul_bool	クライアントからダウンロード確認を提供するかどうかを Mobile Link サーバに指示します。
stream	const char *	同期に使用する Mobile Link ネットワーク・プロトコルです。
stream_error	ul_stream_error	通信エラー・レポート情報を保持する構造体です。
stream_parms	char *	選択されたネットワーク・プロトコルを設定するオプションです。
upload_ok	ul_bool	Mobile Link サーバにアップロードされたデータのステータスです。アップロードに成功すると、true をレポートします。
upload_only	ul_bool	現在の同期中は、統合データベースから変更をダウンロードしません。これにより、特に低速の通信リンクでは、通信時間を節約できます。
user_data	ul_void *	アプリケーション固有の情報を同期 observer で使用できるようにします。このパラメータはオプションです。
user_name	char *	Mobile Link サーバがユニークな Mobile Link ユーザを識別するために使用する文字列です。

名前	型	説明
version	char *	Ultra Light アプリケーションは、バージョン文字列により、同期スクリプトのセットから選択できます。

備考

同期パラメータは、Ultra Light データベースと Mobile Link サーバ間の同期の動作を制御します。Stream Type 同期パラメータ、User Name 同期パラメータ、Version 同期パラメータが必要です。これらが設定されていない場合、同期関数はエラー (SQLE_SYNC_INFO_INVALID またはこれと同じもの) を返します。Download Only、Ping、または Upload Only は一度に 1 つのみ指定できます。これらのパラメータが 1 つ以上 true に設定されると、同期関数はエラー (SQLE_SYNC_INFO_INVALID またはこれと同じもの) を返します。

ul_synch_info_w2 構造体

同期を記述するために使用されるワイド文字構造体です。

構文

public **ul_synch_info_w2**

プロパティ

名前	型	説明
additional_parms	const ul_wchar *	「キーワード=値」のペアがセミコロンで区切られたリストとして指定された、追加の同期パラメータの文字列です。通常このフィールドには、使用頻度の低い同期パラメータが入ります。「Additional Parameters 同期パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。
auth_parms	ul_wchar **	Mobile Link イベントの認証パラメータの配列です。
auth_status	ul_auth_status	Mobile Link のユーザ認証のステータスです。Mobile Link サーバが、この情報をクライアントに提供します。
auth_value	ul_s_long	Mobile Link サーバが、この情報をクライアントに提供し、認証ステータスを判断できるようにします。
download_only	ul_bool	現在の同期中は、Ultra Light データベースから変更をアップロードしません。
ignored_rows	ul_bool	無視されたローのステータスです。同期中にスクリプトがないために Mobile Link サーバによってローが 1 つでも無視されると、この読み込み専用フィールドが true をレポートします。
init_verify	ul_synch_info_w2 *	検証を初期化します。
keep_partial_download	ul_bool	同期時の通信エラーが原因でダウンロードが失敗すると、このパラメータは、変更をロールバックしないで部分的なダウンロードを保持するかどうかを制御します。
new_password	ul_wchar *	ユーザ名に対する新しい Mobile Link パスワードを指定する文字列です。このパラメータはオプションです。
num_auth_parms	ul_byte	Mobile Link イベントの認証パラメータに渡される認証パラメータの数です。
observer	ul_synch_observer_fn	同期をモニタするコールバック関数またはイベント・ハンドラへのポインタです。このパラメータはオプションです。

名前	型	説明
partial_download_retained	ul_bool	同期時の通信エラーが原因でダウンロードが失敗すると、このパラメータは、変更をロールバックしないで、ダウンロードされたこの変更が適用されたかどうかを示します。
password	ul_wchar *	ユーザ名に対する既存の Mobile Link パスワードを指定する文字列です。このパラメータはオプションです。
ping	ul_bool	Ultra Light クライアントと Mobile Link サーバ間の通信を確認します。このパラメータが true に設定されている場合は、同期は行われません。
publications	const ul_wchar *	同期に含めるデータを示す、パブリケーションをカンマで区切ったリストです。
resume_partial_download	ul_bool	失敗したダウンロードを再開します。同期によって変更はアップロードされず、失敗したダウンロードで変更のみがダウンロードされます。
send_column_names	ul_bool	アップロード時にカラム名が Mobile Link サーバに送信されるようアプリケーションに指示します。
send_download_ack	ul_bool	クライアントからダウンロード確認を提供するかどうかを Mobile Link サーバに指示します。
stream	const char *	同期に使用する Mobile Link ネットワーク・プロトコルです。
stream_error	ul_stream_error	通信エラー・レポート情報を保持する構造体です。
stream_parms	ul_wchar *	選択されたネットワーク・プロトコルを設定するオプションです。
upload_ok	ul_bool	Mobile Link サーバにアップロードされたデータのステータスです。アップロードに成功すると、true をレポートします。
upload_only	ul_bool	現在の同期中は、統合データベースから変更をダウンロードしません。これにより、特に低速の通信リンクでは、通信時間を節約できます。
user_data	ul_void *	アプリケーション固有の情報を同期 observer で使用できるようにします。このパラメータはオプションです。
user_name	ul_wchar *	Mobile Link サーバがユニークな Mobile Link ユーザを識別するために使用する文字列です。

名前	型	説明
version	ul_wchar *	Ultra Light アプリケーションは、バージョン文字列により、同期スクリプトのセットから選択できます。

備考

同期パラメータは、Ultra Light データベースと Mobile Link サーバ間の同期の動作を制御します。Stream Type 同期パラメータ、User Name 同期パラメータ、Version 同期パラメータが必要です。これらが設定されていない場合、同期関数はエラー (SQLE_SYNC_INFO_INVALID またはこれと同じもの) を返します。Download Only、Ping、または Upload Only は一度に 1 つのみ指定できます。これらのパラメータが 1 つ以上 true に設定されると、同期関数はエラー (SQLE_SYNC_INFO_INVALID またはこれと同じもの) を返します。「ul_synch_info_a 構造体」 135 ページを参照してください。

ul_synch_result 構造体

アプリケーションで適切なアクションを実行できるようにするために、同期の結果を格納する構造体です。

構文

public **ul_synch_result**

プロパティ

名前	型	説明
auth_status	ul_auth_status	同期認証ステータスです。
auth_value	ul_s_long	Mobile Link サーバが auth_status の結果を判断するために使用する値です。
ignored_rows	ul_bool	アップロードされたローが無視された場合は true に設定され、無視されなかった場合は false に設定されます。
partial_download_retained	ul_bool	部分的なダウンロードが保持されたことを通知する値です。「keep_partial_download」を参照してください。
sql_code	an_sql_code	最後の同期の SQL コードです。
sql_error_string	char	sql_code 内のエラー・コードに関連付けられているエラー・テキストです。
status	ul_synch_status	observer 関数によって使用されるステータス情報です。「observer」を参照してください。
stream_error	ul_stream_error	通信ストリーム・エラー情報です。
timestamp	SQLDATETIME	最後の同期の時刻と日付です。
upload_ok	ul_bool	アップロードが正常に終了した場合は true に設定され、正常に終了しなかった場合は false に設定されます。

ul_synch_stats 構造体

同期ストリームの統計情報をレポートします。

構文

public **ul_synch_stats**

プロパティ

名前	型	説明
bytes	ul_u_long	現在までに送信されたバイト数です。
deletes	ul_u_long	現在までに送信された削除済みのローの数です。
inserts	ul_u_long	現在までに挿入されたローの数です。
updates	ul_u_long	現在までに送信された更新済みのローの数です。

ul_synch_status 構造体

同期の進行状況のモニタリング・データを返します。

構文

public **ul_synch_status**

プロパティ

名前	型	説明
db_table_count	ul_u_short	データベース内のテーブルの数を返します。
flags	ul_u_short	現在の状態に関連する追加情報を示す、現在の同期フラグを返します。
info	ul_synch_info_a *	ul_synch_info_a 構造体へのポインタ。「ul_synch_info_a 構造体」 135 ページを参照してください。
received	ul_synch_stats	ダウンロードの統計情報を返します。
sent	ul_synch_stats	アップロードの統計情報を返します。
sqlca	SQLCA *	接続のアクティブな SQLCA です。
state	ul_synch_state	サポートされている多くのステータスの 1 つです。「同期ステータス情報の処理」 59 ページを参照してください。
stop	ul_bool	同期をキャンセルする bool 値です。値 true は同期がキャンセルされたことを意味します。
sync_table_count	ul_u_short	同期中のテーブルの数を返します。
sync_table_index	ul_u_short	1 から sync_table_count で定義される数までの値の範囲です。
table_id	ul_u_short	現在アップロードまたはダウンロードされているテーブルの ID です (1 から始まります)。同期されないテーブルがある場合には、この番号で値がスキップされることがあります。また、番号が必ず増加するとはかぎりません。
table_name	char	現在のテーブルの名前。
table_name_w2	char	現在のテーブルの名前 (ワイド文字フォーマット)。このフィールドには、Windows デスクトップまたは Mobile プラットフォームの場合のみ値が入ります。

名前	型	説明
user_data	ul_void *	ULRegisterSynchronizationCallback に渡されたユーザ・データ、または ul_synch_info 構造体で設定されたユーザ・データです。

ul_validate_data 構造体

検証のステータス情報です。

構文

public **ul_validate_data**

プロパティ

名前	型	説明
I	ul_u_long	整数としてのパラメータです。
parm_count	ul_u_short	構造体内のパラメータ数です。
parm_type	enum	パラメータ配列内の各パラメータのタイプを示します (整数または文字列)。
parms	struct ul_validate_data::@12	パラメータの配列です。
s	char	文字列としてのパラメータです (ワイド文字ではありません)。
status_id	ul_validate_status_id	検証プロセスでレポートされる対象を示します。
stop	ul_bool	検証をキャンセルする bool 値です。値 true は検証がキャンセルされたことを意味します。
type	parm_type	格納されるパラメータのタイプです。
user_data	ul_void *	検証ルーチンに渡されるユーザ定義のデータ・ポインタです。

ULSqlca クラス

「ULSqlcaBase クラス」 148 ページに SQLCA 構造体が含まれるため、外部構造体は必要ありません。

構文

public **ULSqlca**

基本クラス

- 「ULSqlcaBase クラス」 148 ページ

メンバ

ULSqlca のすべてのメンバ (継承されたメンバを含みます) を以下に示します。

- 「Finalize 関数」 148 ページ
- 「GetCA 関数」 149 ページ
- 「GetParameter 関数」 149 ページ
- 「GetParameter 関数」 149 ページ
- 「GetParameterCount 関数」 150 ページ
- 「GetSQLCode 関数」 150 ページ
- 「GetSQLCount 関数」 151 ページ
- 「GetSQLErrorOffset 関数」 151 ページ
- 「Initialize 関数」 151 ページ
- 「LastCodeOK 関数」 152 ページ
- 「LastFetchOK 関数」 152 ページ
- 「ULSqlca 関数」 146 ページ
- 「~ULSqlca 関数」 146 ページ

備考

このクラスは、ほとんどの C++ コンポーネント・アプリケーションで使用されます。SQLCA は、他の関数を呼び出す前に初期化する必要があります。それぞれのスレッドには、固有の SQLCA が必要です。

ULSqlca 関数

この関数は SQLCA コンストラクタです。

構文

ULSqlca::ULSqlca()

~ULSqlca 関数

この関数は SQLCA デストラクタです。

構文

ULSqlca::~ULSqlca()

ULSqlcaBase クラス

インタフェース・ライブラリとアプリケーション間の SQLCA を定義します。

構文

public **ULSqlcaBase**

派生クラス

- 「ULSqlca クラス」 146 ページ
- 「ULSqlcaWrap クラス」 153 ページ

メンバ

ULSqlcaBase のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「Finalize 関数」 148 ページ
- 「GetCA 関数」 149 ページ
- 「GetParameter 関数」 149 ページ
- 「GetParameter 関数」 149 ページ
- 「GetParameterCount 関数」 150 ページ
- 「GetSQLCode 関数」 150 ページ
- 「GetSQLCount 関数」 151 ページ
- 「GetSQLErrorOffset 関数」 151 ページ
- 「Initialize 関数」 151 ページ
- 「LastCodeOK 関数」 152 ページ
- 「LastFetchOK 関数」 152 ページ

備考

SQLCA を作成するには、このクラスのサブクラス (通常は「ULSqlca クラス」 146 ページ) を使用します。基本となる SQLCA オブジェクトが必ず存在しなければなりません。SQLCA は、他の関数を呼び出す前に初期化する必要があります。それぞれのスレッドには、固有の SQLCA が必要です。

Finalize 関数

この SQLCA をファイナライズします。

構文

void **ULSqlcaBase::Finalize()**

備考

SQLCA は、再度初期化しないと使用できません。

GetCA 関数

追加のフィールドに直接アクセスするための SQLCA 構造体を取得します。

構文

SQLCA * **ULSqlcaBase::GetCA()**

戻り値

未加工のの SQLCA 構造体。

GetParameter 関数

エラーのパラメータ文字列を取得します。

構文

```
size_t ULSqlcaBase::GetParameter(
  ul_u_long parm_num,
  char * buffer,
  size_t size
)
```

パラメータ

- **parm_num**　1 から始まるパラメータ番号。
- **buffer**　パラメータ文字列を受け取るバッファ。
- **size**　バッファのサイズ (文字数)。

戻り値

- この関数が正常に動作した場合の戻り値は、パラメータ文字列全体を保持するのに必要なバッファ・サイズです。
- 正常に動作しなかった場合は、戻り値は 0 です。無効な (範囲外の) パラメータ番号が指定されると、この関数は正常に動作しません。

備考

バッファが小さすぎてパラメータがトランケートされる場合であっても、出力パラメータ文字列は常に NULL で終了します。パラメータ番号は 1 から始まります。

GetParameter 関数

エラーのパラメータ文字列を取得します。

構文

```
size_t ULSqlcaBase::GetParameter(
  ul_u_long parm_num,
  ul_wchar * buffer,
  size_t size
)
```

パラメータ

- **parm_num**　1 から始まるパラメータ番号。
- **buffer**　パラメータ文字列を受け取るバッファ。
- **size**　バッファのサイズ (ul_wchar 数)。

戻り値

- この関数が正常に動作した場合の戻り値は、パラメータ文字列全体を保持するのに必要なバッファ・サイズです。
- 正常に動作しなかった場合は、戻り値は 0 です。無効な (範囲外の) パラメータ番号が指定されると、この関数は正常に動作しません。

備考

バッファが小さすぎてパラメータがトランケートされる場合であっても、出力パラメータ文字列は常に NULL で終了します。パラメータ番号は 1 から始まります。

GetParameterCount 関数

最後の操作のエラー・パラメータの数を取得します。

構文

```
ul_u_long ULSqlcaBase::GetParameterCount()
```

戻り値

現在のエラーのパラメータの数。

GetSQLCode 関数

最後の操作のエラー・コード (SQLCODE) を取得します。

構文

```
an_sql_code ULSqlcaBase::GetSQLCode()
```

戻り値

sqlcode 値。

GetSQLCount 関数

最後の操作の SQL カウント変数 (SQLCOUNT) を取得します。

構文

an_sql_code **ULSqlcaBase::GetSQLCount()**

戻り値

挿入、更新、または削除操作の影響を受けるローの数。影響を受けるローがない場合は 0 です。

GetSQLErrorOffset 関数

動的 SQL 文でのエラーのオフセットを取得します。

構文

ul_s_long **ULSqlcaBase::GetSQLErrorOffset()**

戻り値

- 取得可能な場合、戻り値は、現在のエラーに該当する (PrepareStatement 関数に渡される) 動的 SQL 文のオフセットです。
- 取得できない場合、戻り値は -1 です。

Initialize 関数

この SQLCA を初期化します。

構文

bool **ULSqlcaBase::Initialize()**

戻り値

- SQLCA が初期化された場合は、true。
- 処理が失敗した場合は、false。基本のインタフェース・ライブラリの初期化に失敗すると、このメソッドは失敗することがあります。システム・リソースが不足していると、ライブラリの失敗が発生します。

備考

SQLCA は、他の操作が発生する前に初期化する必要があります。

LastCodeOK 関数

最後の操作のエラー・コードを調べます。

構文

bool **ULSqlcaBase::LastCodeOK()**

戻り値

- sqlcode が SQLE_NOERROR であるか、警告の場合は TRUE。
- sqlcode がエラーを表す場合は、FALSE。

LastFetchOK 関数

最後のフェッチ操作のエラー・コードを調べます。

構文

bool **ULSqlcaBase::LastFetchOK()**

戻り値

- 最後の操作でローが正常にフェッチされたことを sqlcode が示している場合は、TRUE。
- ローがフェッチされなかったことを sqlcode が示している場合は、FALSE。

備考

この関数は、フェッチ操作の実行直後にのみ使用します。

ULSqlcaWrap クラス

既存の SQLCA オブジェクトにアタッチする「ULSqlcaBase クラス」 148 ページです。

構文

public **ULSqlcaWrap**

基本クラス

- 「ULSqlcaBase クラス」 148 ページ

メンバ

ULSqlcaWrap のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「Finalize 関数」 148 ページ
- 「GetCA 関数」 149 ページ
- 「GetParameter 関数」 149 ページ
- 「GetParameter 関数」 149 ページ
- 「GetParameterCount 関数」 150 ページ
- 「GetSQLCode 関数」 150 ページ
- 「GetSQLCount 関数」 151 ページ
- 「GetSQLErrorOffset 関数」 151 ページ
- 「Initialize 関数」 151 ページ
- 「LastCodeOK 関数」 152 ページ
- 「LastFetchOK 関数」 152 ページ
- 「ULSqlcaWrap 関数」 153 ページ
- 「~ULSqlcaWrap 関数」 154 ページ

備考

これは、前に初期化した SQLCA オブジェクトとともに使用できます (この場合、Initialize 関数を再度呼び出しません)。SQLCA は、他の関数を呼び出す前に初期化する必要があります。それぞれのスレッドには、固有の SQLCA が必要です。

ULSqlcaWrap 関数

コンストラクタです。

構文

```
ULSqlcaWrap::ULSqlcaWrap(
  SQLCA * sqlca
)
```

パラメータ

- **sqlca**　使用する SQLCA オブジェクト。

備考

このオブジェクトを作成する前に SQLCA オブジェクトを初期化できます。この場合は、「Initialize 関数」 151 ページを再度呼び出さないでください。

~ULSqlcaWrap 関数

デストラクタです。

構文

ULSqlcaWrap::~ULSqlcaWrap()

UltraLite_Connection クラス

Ultra Light データベースへの接続を表します。

構文

public **UltraLite_Connection**

基本クラス

- 「UltraLite_SQLObject_iface クラス」 220 ページ
- 「UltraLite_Connection_iface クラス」 158 ページ

メンバ

UltraLite_Connection のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

UltraLite_Connection_iface クラス

接続インタフェースです。

構文

public **UltraLite_Connection_iface**

派生クラス

- 「UltraLite_Connection クラス」 155 ページ

メンバ

UltraLite_Connection_iface のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「SetSynchInfo 関数」 177 ページ
- 「SetSynchInfo 関数」 177 ページ
- 「Shutdown 関数」 178 ページ
- 「StartSynchronizationDelete 関数」 178 ページ
- 「StopSynchronizationDelete 関数」 178 ページ
- 「StrToUUID 関数」 178 ページ
- 「StrToUUID 関数」 179 ページ
- 「Synchronize 関数」 179 ページ
- 「Synchronize 関数」 179 ページ
- 「SynchronizeFromProfile 関数」 180 ページ
- 「TriggerEvent 関数」 180 ページ
- 「UUIDToStr 関数」 181 ページ
- 「UUIDToStr 関数」 181 ページ
- 「ValidateDatabase 関数」 182 ページ

CancelGetNotification 関数

指定された名前に一致する、すべてのキューに登録されている保留中の取得通知コールをキャンセルします。

構文

```
ul_u_long UltraLite_Connection_iface::CancelGetNotification(
  const ULValue & queue_name
)
```

パラメータ

- **queue_name**　キャンセルするキューの名前。

戻り値

影響を受けるキューの数 (ブロックされた読み込み数とは異なります)。

参照

- 「DeclareEvent 関数」 162 ページ
- 「DestroyNotificationQueue 関数」 163 ページ
- 「GetNotification 関数」 166 ページ
- 「RegisterForEvent 関数」 172 ページ
- 「SendNotification 関数」 174 ページ
- 「TriggerEvent 関数」 180 ページ

ChangeEncryptionKey 関数

暗号化キーを変更します。

構文

```
bool UltraLite_Connection_iface::ChangeEncryptionKey(
  const ULValue & new_key
)
```

パラメータ

- **new_key**　データベースの新しい暗号化キーの値。

Checkpoint 関数

データベースにチェックポイントを設定します。

構文

```
bool UltraLite_Connection_iface::Checkpoint()
```

Commit 関数

現在のトランザクションをコミットします。

構文

```
bool UltraLite_Connection_iface::Commit()
```

CountUploadRows 関数

アップロードする必要があるローの数を特定します。

構文

```
ul_u_long UltraLite_Connection_iface::CountUploadRows(
  const ULValue & pub_list,
  ul_u_long threshold
)
```

パラメータ

- **pub_list**　対象のパブリケーションをカンマで区切ったリスト。
- **threshold**　カウントするローの数の制限。

CreateNotificationQueue 関数

この接続のイベント通知キューを作成します。

構文

```
bool UltraLite_Connection_iface::CreateNotificationQueue(
  const ULValue & name,
  const ULValue & parameters
)
```

パラメータ

- **name**　新しいキューの名前。
- **parameters**　作成パラメータ。現在は使われず、NULL に設定されています。

備考

キュー名は、接続ごとにスコープされるため、別々の接続で同じ名前を持つキューを作成できます。イベント通知が送信されると、データベース内で一致する名前を持つすべてのキューが、個別のインスタンスの通知を受け取ります。名前では大文字と小文字が区別されません。
「RegisterForEvent 関数」 172 ページを呼び出したときに、キューが指定されていない場合は、接続ごとに要求に応じてデフォルトのキューが作成されます。その名前がすでに存在する場合や有効でない場合は、エラーが発生して呼び出しが失敗します。

参照

- 「CancelGetNotification 関数」 160 ページ
- 「DeclareEvent 関数」 162 ページ
- 「DestroyNotificationQueue 関数」 163 ページ
- 「GetNotification 関数」 166 ページ

DeclareEvent 関数

登録およびトリガされるイベントを宣言します。

構文

```
bool UltraLite_Connection_iface::DeclareEvent(
  const ULValue & event_name
)
```

パラメータ

- **event_name**　新しいユーザ定義イベントの名前。

戻り値

イベントが正常に宣言された場合は ul_true、名前がすでに使用されているか無効な場合は ul_false。

備考

Ultra Light では、データベースまたは環境での操作によってトリガされるシステム・イベントの一部が事前に定義されています。この関数は、ユーザ定義イベントを宣言します。ユーザ定義イベントは、「TriggerEvent 関数」 180 ページによってトリガされます。イベント名は、ユニークにする必要があります。名前では大文字と小文字が区別されません。

参照

- 「CancelGetNotification 関数」 160 ページ
- 「DestroyNotificationQueue 関数」 163 ページ
- 「GetNotification 関数」 166 ページ
- 「RegisterForEvent 関数」 172 ページ
- 「SendNotification 関数」 174 ページ
- 「TriggerEvent 関数」 180 ページ

DestroyNotificationQueue 関数

指定されたイベント通知キューを破棄します。

構文

```
bool UltraLite_Connection_iface::DestroyNotificationQueue(
  const ULValue & name
)
```

パラメータ

- **name**　破棄するキューの名前。

備考

キュー内に未読の通知が残っている場合は、警告が通知されます。未読の通知は破棄されます。接続のデフォルトのイベント・キューが作成されている場合、接続が閉じると破棄されます。

参照

- 「CancelGetNotification 関数」 160 ページ
- 「DeclareEvent 関数」 162 ページ
- 「GetNotification 関数」 166 ページ
- 「RegisterForEvent 関数」 172 ページ
- 「SendNotification 関数」 174 ページ
- 「TriggerEvent 関数」 180 ページ

ExecuteNextSQLPassthroughScript 関数

次の SQL パススルー・スクリプトを実行します。

構文

```
bool UltraLite_Connection_iface::ExecuteNextSQLPassthroughScript()
```

戻り値

スクリプトの実行中にエラーが発生した場合は false。

参照

- 「ExecuteSQLPassthroughScripts 関数」 164 ページ
- 「GetSQLPassthroughScriptCount 関数」 168 ページ

ExecuteSQLPassthroughScripts 関数

使用可能なすべての SQL パススルー・スクリプトを実行します。

構文

bool **UltraLite_Connection_iface::ExecuteSQLPassthroughScripts()**

戻り値

スクリプトの実行中にエラーが発生した場合は false。

参照

- 「ExecuteNextSQLPassthroughScript 関数」 163 ページ
- 「GetSQLPassthroughScriptCount 関数」 168 ページ

GetConnectionNum 関数

接続の数を取得します。

構文

ul_connection_num **UltraLite_Connection_iface::GetConnectionNum()**

GetDatabaseID 関数

グローバル・オートインクリメント・カラムに使用されるデータベース ID を取得します。

構文

ul_u_long **UltraLite_Connection_iface::GetDatabaseID()**

GetDatabaseProperty 関数

データベースのプロパティを取得します。

構文

ULValue **UltraLite_Connection_iface::GetDatabaseProperty(**
  ul_database_property_id *id*
)

パラメータ

- **id**　要求されているプロパティの ID。

戻り値

要求されているプロパティの値。

GetDatabaseProperty 関数

データベースのプロパティを取得します。

構文

```
ULValue UltraLite_Connection_iface::GetDatabaseProperty(
  const ULValue & prop_name
)
```

パラメータ

- **prop_name**　要求されているプロパティの文字列名。

戻り値

要求されているプロパティの値。

GetLastDownloadTime 関数

前回のダウンロードの時刻を取得します。

構文

```
bool UltraLite_Connection_iface::GetLastDownloadTime(
  const ULValue & pub_list,
  DECL_DATETIME * value
)
```

パラメータ

- **pub_list**　対象のパブリケーションをカンマで区切ったリスト。
- **value**　前回のダウンロードの時刻。

GetLastIdentity 関数

@@identity の値を取得します。

構文

```
ul_u_big UltraLite_Connection_iface::GetLastIdentity()
```

GetNewUUID 関数

新しい UUID を作成します。

構文

```
bool UltraLite_Connection_iface::GetNewUUID(
  p_ul_binary uuid
)
```

パラメータ

- **uuid**　新しい UUID 値。

GetNewUUID 関数

新しい UUID を作成します。

構文

```
bool UltraLite_Connection_iface::GetNewUUID(
  GUID * uuid
)
```

パラメータ

- **uuid**　新しい UUID 値。

GetNotification 関数

イベント通知を読み込みます。

構文

```
ULValue UltraLite_Connection_iface::GetNotification(
  const ULValue & queue_name,
  ul_u_long wait_ms
)
```

パラメータ

- **queue_name**　読み取るキュー。デフォルト接続キューの場合は NULL。
- **wait_ms**　返す前に、待機 (ブロック) する時間。

戻り値

読み込まれたイベントの名前。エラーが発生した場合は空の文字列。

備考

この呼び出しは、通知が受信されるまで、または指定された待機時間が経過するまでブロックします。無限に待機する場合は、wait_ms に UL_READ_WAIT_INFINITE を渡します。待機をキャンセルするには、指定したキューに別の通知を送信するか、「CancelGetNotification 関数」 160 ページを使用します。通知を読み込んだら、ReadNotificationParameter() を使用して、追加のパラメータを名前で取得します。

参照

- 「CancelGetNotification 関数」 160 ページ
- 「DeclareEvent 関数」 162 ページ
- 「DestroyNotificationQueue 関数」 163 ページ
- 「RegisterForEvent 関数」 172 ページ
- 「SendNotification 関数」 174 ページ
- 「TriggerEvent 関数」 180 ページ

GetNotificationParameter 関数

「GetNotification 関数」 166 ページによって読み込まれたイベント通知のパラメータを取得します。

構文

```
ULValue UltraLite_Connection_iface::GetNotificationParameter(
  const ULValue & queue_name,
  const ULValue & parameter_name
)
```

パラメータ

- **queue_name**　「GetNotification 関数」 166 ページ呼び出しと一致するキュー名。
- **parameter_name**　読み込むパラメータの名前 (または "*")。

戻り値

パラメータ値。エラーが発生した場合は空の文字列。

備考

指定されたキューでの最後に読み込まれた通知のパラメータのみ取得できます。パラメータは名前によって取得されます。パラメータ名を "*" と指定すると、パラメータ文字列全体が取得されます。

参照

- 「CancelGetNotification 関数」 160 ページ
- 「DeclareEvent 関数」 162 ページ
- 「DestroyNotificationQueue 関数」 163 ページ
- 「GetNotification 関数」 166 ページ
- 「RegisterForEvent 関数」 172 ページ
- 「SendNotification 関数」 174 ページ
- 「TriggerEvent 関数」 180 ページ

GetSQLPassthroughScriptCount 関数

実行できる SQL パススルー・スクリプトの数を取得します。

構文

ul_u_long **UltraLite_Connection_iface::GetSQLPassthroughScriptCount()**

参照

- 「ExecuteNextSQLPassthroughScript 関数」 163 ページ
- 「ExecuteSQLPassthroughScripts 関数」 164 ページ

GetSchema 関数

データベース・スキーマを取得します。

構文

UltraLite_DatabaseSchema * **UltraLite_Connection_iface::GetSchema()**

GetSqlca 関数

この接続に関連付けられている SQLCA を取得します。

構文

ULSqlcaBase const & **UltraLite_Connection_iface::GetSqlca()**

GetSuspend 関数 (旧式)

サスペンドのプロパティを取得します。

構文

bool **UltraLite_Connection_iface::GetSuspend()**

戻り値

- 接続がサスペンドされる場合は、true。
- 接続がサスペンドされない場合は、false。

GetSynchResult 関数

前回の同期結果を取得します。

構文

```
bool UltraLite_Connection_iface::GetSynchResult(
  ul_synch_result * synch_result
)
```

パラメータ

- **synch_result**　同期結果を保持する「ul_synch_result 構造体」 141 ページへのポインタ。

GetUtilityULValue 関数

新しい「ULValue クラス」 247 ページを取得します。

構文

```
ULValue UltraLite_Connection_iface::GetUtilityULValue()
```

備考

「ULValue クラス」 247 ページオブジェクトが接続にバインドされていないと、そのメソッドの多くは成功しません。

GlobalAutoincUsage 関数

グローバル・オートインクリメントの値について、カウンタによる使用済み比率 (%) を取得します。

構文

```
ul_u_short UltraLite_Connection_iface::GlobalAutoincUsage()
```

GrantConnectTo 関数

指定されたユーザへの接続アクセスを許可します。

構文

```
bool UltraLite_Connection_iface::GrantConnectTo(
  const ULValue & uid,
  const ULValue & pwd
)
```

パラメータ

- **uid**　接続へのアクセス権が付与されているユーザ ID。
- **pwd**　認証されたユーザ ID のパスワード。

備考

新しいユーザを作成するには、新しいユーザの ID とパスワードの両方を指定します。

パスワードを変更するには、既存のユーザ ID を指定し、そのユーザの新しいパスワードを設定します。

InitSynchInfo 関数

同期情報の構造体を初期化します。

構文

```
void UltraLite_Connection_iface::InitSynchInfo(
  ul_synch_info_a * info
)
```

パラメータ

- **info**　同期パラメータを保持する ul_synch_info 構造体へのポインタ。

InitSynchInfo 関数

同期情報の構造体を初期化します。

構文

```
void UltraLite_Connection_iface::InitSynchInfo(
  ul_synch_info_w2 * info
)
```

パラメータ

- **info**　同期パラメータを保持する ul_synch_info 構造体へのポインタ。

OpenTable 関数

テーブルを開きます。

構文

```
UltraLite_Table * UltraLite_Connection_iface::OpenTable(
  const ULValue & table_id,
  const ULValue & persistent_name
)
```

パラメータ

- **table_id**　テーブルの名前または順序。
- **persistent_name**　サスペンド処理に使用されるインスタンスの名前。

備考

アプリケーションがテーブルを初めて開いたときは、カーソルの位置は BeforeFirst() に設定されます。

OpenTableEx 関数

ローを取得するためにテーブルを開きます。

構文

```
UltraLite_Table * UltraLite_Connection_iface::OpenTableEx(
  const ULValue & table_id,
  ul_table_open_type open_type,
  const ULValue & parms,
  const ULValue & persistent_name
)
```

パラメータ

- **table_id**　テーブルの名前または順序。
- **open_type**　ローが返される方法を制御します。
- **parms**　open_type に応じたオプションのパラメータ (インデックス名と同様)。
- **persistent_name**　サスペンド処理に使用されるインスタンスの名前。

備考

ローの順序は、テーブルを開くために使ったインデックスによって変わります。インデックスを使用しなかった場合は、ローは任意の順序になります。アプリケーションがテーブルを初めて開いたときは、カーソルの位置は BeforeFirst() に設定されます。

インデックスを使用しないでテーブルを開くと、パフォーマンスが向上することがあります。ただし、インデックスを使用しなかった場合には、返されたテーブルを使用してデータを変更することはできず、ルックアップを実行できません。

OpenTableWithIndex 関数

ローを順序付けるための指定されたインデックスを使用して、テーブルを開きます。

構文

```
UltraLite_Table * UltraLite_Connection_iface::OpenTableWithIndex(
  const ULValue & table_id,
  const ULValue & index_id,
  const ULValue & persistent_name
)
```

パラメータ

- **table_id**　テーブルの名前または順序。
- **index_id**　インデックスの名前または順序。
- **persistent_name**　サスペンド処理に使用されるインスタンスの名前。

備考

アプリケーションがテーブルを初めて開いたときは、カーソルの位置は BeforeFirst() に設定されます。

PrepareStatement 関数

SQL 文の準備を行います。

構文

```
UltraLite_PreparedStatement * UltraLite_Connection_iface::PrepareStatement(
  const ULValue & sql,
  const ULValue & persistent_name
)
```

パラメータ

- **sql**　文字列としての SQL 文。
- **persistent_name**　サスペンド処理に使用されるインスタンスの名前。

RegisterForEvent 関数

イベントの通知を受け取るためのキューを登録または登録解除します。

構文

```
bool UltraLite_Connection_iface::RegisterForEvent(
  const ULValue & event_name,
  const ULValue & object_name,
  const ULValue & queue_name,
```

```
  bool register_not_unreg
)
```

パラメータ

- **event_name**　登録するシステム定義またはユーザ定義のイベント。
- **object_name**　イベントを適用するオブジェクト (テーブル名と同様)。
- **queue_name**　NULL は、デフォルトの接続キューを表します。
- **register_not_unreg**　登録する場合は true、登録解除する場合は false。

戻り値

正常に登録できた場合は true、キューまたはイベントが存在しない場合は false。

備考

キュー名が指定されていない場合は、デフォルトの接続キューが暗黙で指定され、必要に応じて作成されます。特定のシステム・イベントでは、そのイベントが適用されるオブジェクト名を指定できます。たとえば、TableModified イベントではテーブル名を指定できます。「SendNotification 関数」 174 ページとは異なり、登録された特定のキューのみイベントの通知を受信します。別の接続の同じ名前を持つ他のキューは、明示的に登録されている場合を除き、イベントの通知を受信しません。

事前に定義されたシステム・イベントは次のとおりです。

- TableModified - テーブル内のローが挿入、更新、または削除されたときにトリガされます。要求の影響を受けるローの数にかかわらず、要求ごとに 1 つの通知が送信されます。object_name パラメータは、モニタするテーブルを指定します。値 "*" は、データベース内のすべてのテーブルを意味します。このイベントには 'table_name' というパラメータがあり、このパラメータの値は変更されたテーブルの名前です。
- Commit - コミットが完了した後にトリガされます。このイベントにはパラメータはありません。
- SyncComplete - 同期が完了した後にトリガされます。このイベントにはパラメータはありません。

参照

- 「CancelGetNotification 関数」 160 ページ
- 「DeclareEvent 関数」 162 ページ
- 「DestroyNotificationQueue 関数」 163 ページ
- 「GetNotification 関数」 166 ページ
- 「SendNotification 関数」 174 ページ
- 「TriggerEvent 関数」 180 ページ

ResetLastDownloadTime 関数

指定されたパブリケーションの最後のダウンロードの時刻をリセットします。

構文

bool **UltraLite_Connection_iface::ResetLastDownloadTime(**
  const ULValue & *pub_list*
)

パラメータ

- **pub_list**　リセットするパブリケーション。

RevokeConnectFrom 関数

既存のユーザを削除します。

構文

bool **UltraLite_Connection_iface::RevokeConnectFrom(**
  const ULValue & *uid*
)

パラメータ

- **uid**　接続する権限を取り消されるユーザ ID。

Rollback 関数

現在のトランザクションをロールバックします。

構文

bool **UltraLite_Connection_iface::Rollback()**

RollbackPartialDownload 関数

部分的なダウンロードをロールバックします。

構文

bool **UltraLite_Connection_iface::RollbackPartialDownload()**

SendNotification 関数

指定された名前と一致するすべてのキューに通知を送信します。

構文

ul_u_long **UltraLite_Connection_iface::SendNotification(**
  const ULValue & *queue_name*,
  const ULValue & *event_name*,


```
  const ULValue & parameters
)
```

パラメータ

- **queue_name**　対象となるキューの名前 (または "*")。
- **event_name**　通知の ID。
- **parameters**　パラメータのオプション・リストまたは NULL。

戻り値

送信済みの通知の数 (一致するキューの数)。

備考

これに含まれるのは、現在の接続におけるキューです。この呼び出しはブロックしません。特別なキュー名の "*" を使用すると、すべてのキューに送信します。指定されたイベント名は、システム定義またはユーザ定義のイベントと対応する必要はありません。読み込まれたイベント名は、通知を識別するためにそのまま渡され、送信者と受信者に対してしか意味を持たないからです。パラメータの引数には、「名前=値」のペアをセミコロンで区切ったオプション・リストを指定します。通知が読み込まれた後、パラメータの値が「GetNotificationParameter 関数」 167 ページによって読み込まれます。

参照

- 「CancelGetNotification 関数」 160 ページ
- 「DeclareEvent 関数」 162 ページ
- 「DestroyNotificationQueue 関数」 163 ページ
- 「GetNotification 関数」 166 ページ
- 「RegisterForEvent 関数」 172 ページ
- 「TriggerEvent 関数」 180 ページ

SetDatabaseID 関数

グローバル・オートインクリメント・カラムに使用されるデータベース ID を設定します。

構文

```
bool UltraLite_Connection_iface::SetDatabaseID(
  ul_u_long value
)
```

パラメータ

- **value**　グローバル・オートインクリメント・カラムの開始値を決定するデータベース ID。

SetDatabaseOption 関数

指定されたデータベース・オプションを設定します。

構文

```
bool UltraLite_Connection_iface::SetDatabaseOption(
  ul_database_option_id id,
  const ULValue & value
)
```

パラメータ

- **id**　設定されるオプションの ID。
- **value**　オプションの新しい値。

SetDatabaseOption 関数

指定されたデータベース・オプションを設定します。

構文

```
bool UltraLite_Connection_iface::SetDatabaseOption(
  const ULValue & option_name,
  const ULValue & value
)
```

パラメータ

- **option_name**　設定されるオプションの文字列名。
- **value**　オプションの新しい値。

SetSuspend 関数 (旧式)

サスペンドのプロパティを設定します。

構文

```
void UltraLite_Connection_iface::SetSuspend(
  bool suspend
)
```

パラメータ

- **suspend**　true に設定すると、接続がサスペンドし、データベースを再度開いたときにステータスをリストアできます。

戻り値

接続がサスペンドされる場合は、true。

備考

接続の名前 (または名前なし) によって、サスペンドされている接続が識別されます。

SetSynchInfo 関数

指定された ul_synch_info 構造体に基づいて、指定された名前を使用して同期プロファイルを作成します。

構文

```
bool UltraLite_Connection_iface::SetSynchInfo(
  char const * profile_name,
  ul_synch_info_a * info
)
```

パラメータ

- **profile_name**　同期のオプションを含む同期プロファイルの名前。profile_name が NULL の場合、プロファイルは使用されないため、同期に使用するすべてのオプションが info 構造体に指定されている必要があります。
- **info**　同期パラメータを保持する ul_synch_info 構造体へのポインタ。

備考

この名前の同期プロファイルがすでにある場合は、置き換えられます。ul_synch_info に NULL ポインタを指定すると、指定されたプロファイルが削除されます。

参照

- 「SynchronizeFromProfile 関数」 180 ページ

SetSynchInfo 関数

指定された ul_synch_info 構造体に基づいて、指定された名前を使用して同期プロファイルを作成します。

構文

```
bool UltraLite_Connection_iface::SetSynchInfo(
  ul_wchar const * profile_name,
  ul_synch_info_w2 * info
)
```

パラメータ

- **profile_name**　同期のオプションを含む同期プロファイルの名前。profile_name が NULL の場合、プロファイルは使用されないため、同期に使用するすべてのオプションが info 構造体に指定されている必要があります。
- **info**　同期パラメータを保持する ul_synch_info 構造体へのポインタ。

参照

- 「SynchronizeFromProfile 関数」 180 ページ

Shutdown 関数

この接続と、残りの関連オブジェクトを破棄します。

構文

void **UltraLite_Connection_iface::Shutdown()**

備考

接続をサスペンドするように設定されていないと、この接続はロールバックされます。

StartSynchronizationDelete 関数

この接続の START SYNCHRONIZATION DELETE を設定します。

構文

bool **UltraLite_Connection_iface::StartSynchronizationDelete()**

参照

- 「StopSynchronizationDelete 関数」 178 ページ

StopSynchronizationDelete 関数

この接続の STOP SYNCHRONIZATION DELETE を設定します。

構文

bool **UltraLite_Connection_iface::StopSynchronizationDelete()**

参照

- 「StartSynchronizationDelete 関数」 178 ページ

StrToUUID 関数

文字列をバイナリ UUID に変換します。

構文

```
bool UltraLite_Connection_iface::StrToUUID(
  p_ul_binary dst,
  size_t len,
  const ULValue & src
)
```


パラメータ

- **dst** 返される UUID 値。
- **len** ul_binary 配列の長さ。
- **src** UUID 値を保持する変換対象の文字列。

StrToUUID 関数

文字列を GUID 構造体に変換します。

構文

```
bool UltraLite_Connection_iface::StrToUUID(
  GUID * dst,
  const ULValue & src
)
```

パラメータ

- **dst** 返される GUID 値。
- **src** UUID 値を保持する変換対象の文字列。

Synchronize 関数

データベースの同期をとります。

構文

```
bool UltraLite_Connection_iface::Synchronize(
  ul_synch_info_a * info
)
```

パラメータ

- **info** 同期パラメータを保持する ul_synch_info 構造体へのポインタ。

例

```
ul_synch_info info;
conn->InitSynchInfo( &info );
info.user_name = UL_TEXT( user_name"user_name" );
info.version = UL_TEXT( version"test" );
conn->Synchronize( &info );
```

Synchronize 関数

データベースの同期をとります。

構文

```
bool UltraLite_Connection_iface::Synchronize(
  ul_synch_info_w2 * info
)
```

パラメータ

- **info**　同期パラメータを保持する ul_synch_info 構造体へのポインタ。

参照

- 「Synchronize 関数」 179 ページ

SynchronizeFromProfile 関数

指定されたプロファイルとマージ・パラメータを使用して、データベースの同期をとります。

構文

```
bool UltraLite_Connection_iface::SynchronizeFromProfile(
  ULValue const & profile_name,
  ULValue const & merge_parms
)
```

パラメータ

- **profile_name**　同期するプロファイルの名前。
- **merge_parms**　同期で使用するマージ・パラメータ。

備考

これは、SYNCHRONIZE 文を実行するのと同じです。

TriggerEvent 関数

ユーザ定義のイベントをトリガして、登録されたすべてのキューに通知を送信します。

構文

```
ul_u_long UltraLite_Connection_iface::TriggerEvent(
  const ULValue & event_name,
  const ULValue & parameters
)
```

パラメータ

- **event_name**　トリガするシステム定義またはユーザ定義のイベントの名前。
- **parameters**　パラメータのオプション・リストまたは NULL。

戻り値

送信済みのイベント通知の数。

備考

パラメータの値には、「名前=値」のペアをセミコロンで区切ったオプション・リストを指定します。通知が読み込まれた後、パラメータの値が GetNotificationParameter によって読み込まれます。

参照

- 「CancelGetNotification 関数」 160 ページ
- 「DeclareEvent 関数」 162 ページ
- 「DestroyNotificationQueue 関数」 163 ページ
- 「GetNotification 関数」 166 ページ
- 「RegisterForEvent 関数」 172 ページ
- 「SendNotification 関数」 174 ページ

UUIDToStr 関数

UUID を ANSI 文字列に変換します。

構文

```
bool UltraLite_Connection_iface::UUIDToStr(
  char * dst,
  size_t len,
  const ULValue & src
)
```

パラメータ

- **dst** 返される文字列。
- **len** ul_binary 配列の長さ。
- **src** 文字列に変換される UUID 値。

UUIDToStr 関数

UUID を Unicode 文字列に変換します。

構文

```
bool UltraLite_Connection_iface::UUIDToStr(
  ul_wchar * dst,
  size_t len,
  const ULValue & src
)
```

パラメータ

- **dst**　返される Unicode 文字列。
- **len**　ul_binary 配列の長さ。
- **src**　文字列に変換される UUID 値。

ValidateDatabase 関数

この接続のデータベースを検証します。

構文

```
bool UltraLite_Connection_iface::ValidateDatabase(
  ul_u_short flags,
  ul_validate_callback_fn fn,
  ul_void * user_data,
  const ULValue & table_id
)
```

パラメータ

- **flags**　検証のタイプを制御するフラグ。
- **fn**　検証の進行状況の情報を受け取る関数。
- **user_data**　コールバックにより呼び出し元に送り返すユーザ・データ。
- **table_id**　検証する特定のテーブル (オプション)。

戻り値

検証中にエラーが発生した場合は false。

備考

このルーチンに渡されるフラグに応じて、低レベルのストアかインデックス (または両方) を検証できます。検証中に情報を受け取るには、コールバック関数を実装し、アドレスをこのルーチンに渡します。検証対象を特定のテーブルに限定するには、テーブルの名前または ID を最後のパラメータとして渡します。

UltraLite_Cursor_iface クラス

Ultra Light データベースの双方向カーソルを表します。

構文

public **UltraLite_Cursor_iface**

派生クラス

- 「UltraLite_ResultSet クラス」 212 ページ
- 「UltraLite_Table クラス」 227 ページ

メンバ

UltraLite_Cursor_iface のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「AfterLast 関数」 183 ページ
- 「BeforeFirst 関数」 184 ページ
- 「Delete 関数」 184 ページ
- 「First 関数」 184 ページ
- 「Get 関数」 184 ページ
- 「GetRowCount 関数」 185 ページ
- 「GetState 関数」 185 ページ
- 「GetStreamReader 関数」 185 ページ
- 「GetStreamWriter 関数」 186 ページ
- 「GetSuspend 関数 (旧式)」 186 ページ
- 「IsNull 関数」 186 ページ
- 「Last 関数」 187 ページ
- 「Next 関数」 187 ページ
- 「Previous 関数」 187 ページ
- 「Relative 関数」 187 ページ
- 「Set 関数」 188 ページ
- 「SetDefault 関数」 188 ページ
- 「SetNull 関数」 188 ページ
- 「SetSuspend 関数 (旧式)」 189 ページ
- 「Update 関数」 189 ページ
- 「UpdateBegin 関数」 189 ページ

備考

カーソルとは、テーブルまたはクエリからの結果セットの一連のローです。

AfterLast 関数

カーソルを最後のローの後に移動します。

構文

bool **UltraLite_Cursor_iface::AfterLast()**

BeforeFirst 関数

カーソルを最初のローの前に移動します。

構文

bool **UltraLite_Cursor_iface::BeforeFirst()**

Delete 関数

現在のローを削除し、次の有効なローに移動します。

構文

bool **UltraLite_Cursor_iface::Delete()**

備考

このカーソルが、インデックスを使用しないで開かれたテーブルである場合、データは読み込み専用とみなされ、ローの削除はできません。

First 関数

カーソルを最初のローに移動します。

構文

bool **UltraLite_Cursor_iface::First()**

Get 関数

カラムから値をフェッチします。

構文

ULValue **UltraLite_Cursor_iface::Get(**
  const ULValue & *column_id*
)

パラメータ

- **column_id**　カラムの名前または順序。

GetRowCount 関数

テーブルのローの数を取得します。

構文

ul_u_long **UltraLite_Cursor_iface::GetRowCount(**
[ul_u_long *threshold*]
)

パラメータ

- **threshold** カウントするローの数のスレッショルドを設定するオプションのパラメータ。スレッショルドが指定されており、その値が 0 より大きい場合、返されるロー・カウントの最大値はスレッショルド値になります。実際のロー数は、スレッショルド値と同じか、それより大きい数です。

備考

スレッショルド値を設定してカウントするローの数を制限すると、特定数を超えるローが存在するかどうかを確認することが重要である場合に役立ちます。たとえば、25 個の項目があるリストにデータを入れるコードで、それ以上のローを表示するオプションをユーザに許可する必要があるかどうかを判定できます。

このメソッドを呼び出すのは、"SELECT COUNT(*) FROM table" を実行するのと同じです。

GetState 関数

カーソルの内部ステータスを取得します。

構文

UL_RS_STATE **UltraLite_Cursor_iface::GetState()**

備考

ulglobal.h の UL_RS_STATE 列挙体を参照してください。

GetStreamReader 関数

チャンク単位で文字列またはバイナリ・カラムのデータを読み込むストリーム・リーダ・オブジェクトです。

構文

UltraLite_StreamReader * **UltraLite_Cursor_iface::GetStreamReader(**
const ULValue & *id*
)

パラメータ

- **id** カラム識別子。1 から始まる順序数またはカラム名です。

GetStreamWriter 関数

文字列データまたはバイナリ・データをカラムにストリーミングするストリーム・ライタを取得します。

構文

```
UltraLite_StreamWriter * UltraLite_Cursor_iface::GetStreamWriter(
  const ULValue & column_id
)
```

パラメータ

- **column_id** カラム識別子。1 から始まる順序数またはカラム名です。

GetSuspend 関数 (旧式)

サスペンドのプロパティの値を取得します。

構文

```
bool UltraLite_Cursor_iface::GetSuspend()
```

戻り値

- カーソルがサスペンドされる場合は、true。
- そうでない場合は、false。

IsNull 関数

カラムが NULL であるかどうかをチェックします。

構文

```
bool UltraLite_Cursor_iface::IsNull(
  const ULValue & column_id
)
```

パラメータ

- **column_id** カラムの名前または順序。

Last 関数

カーソルを最後のローに移動します。

構文

bool **UltraLite_Cursor_iface::Last()**

Next 関数

カーソルをロー 1 つ分進めます。

構文

bool **UltraLite_Cursor_iface::Next()**

戻り値

- カーソルが正常に進められる場合は、true。true が返されても、カーソルが次のローに正常に移動したときに、エラーが送信されることがあります。たとえば SELECT 式の評価中に変換エラーが発生する可能性があります。この場合、カラム値を取得するときにもエラーが返されます。
- カーソルを進められなかった場合は、false。たとえば、次のローが存在しない可能性があります。この場合、カーソルの位置は「AfterLast 関数」 183 ページになります。

Previous 関数

カーソルをロー 1 つ分戻します。

構文

bool **UltraLite_Cursor_iface::Previous()**

備考

処理が失敗すると、カーソル位置は「BeforeFirst 関数」 184 ページとなります。

Relative 関数

カーソルを、現在のカーソルの位置から、offset で指定したロー数分移動します。

構文

bool **UltraLite_Cursor_iface::Relative(**
  ul_fetch_offset *offset*
)

パラメータ

- **offset**　移動するローの数。

Set 関数

カラムの値を設定します。

構文

```
bool UltraLite_Cursor_iface::Set(
  const ULValue & column_id,
  const ULValue & value
)
```

パラメータ

- **column_id**　カラムを識別する 1 から始まる順序数。
- **value**　カラムに設定される値。

SetDefault 関数

カラムを、そのデフォルト値に設定します。

構文

```
bool UltraLite_Cursor_iface::SetDefault(
  const ULValue & column_id
)
```

パラメータ

- **column_id**　カラムを識別する 1 から始まる順序数。

SetNull 関数

カラムを NULL に設定します。

構文

```
bool UltraLite_Cursor_iface::SetNull(
  const ULValue & column_id
)
```

パラメータ

- **column_id**　カラムを識別する 1 から始まる順序数。

SetSuspend 関数 (旧式)

サスペンドのプロパティの値を設定します。

構文

void **UltraLite_Cursor_iface::SetSuspend(**
bool *suspend*
)

パラメータ

- **suspend** true の場合は接続をサスペンドします。この場合、データベースが再度開かれたときに、データベースのステータスをリストアできます。

戻り値

- このカーソルがサスペンドされ、データベースが再度開かれたときにリストアされた場合は、true。
- カーソルがサスペンドされない場合は、false。

備考

関連するオブジェクトを開くときは、永続的な名前のパラメータを使用して、サスペンドされたカーソルを識別します。カーソルに永続的な名前のパラメータが指定されていない場合、そのカーソルはサスペンドできません。

Update 関数

現在の行を更新します。

構文

bool **UltraLite_Cursor_iface::Update()**

備考

この操作を成功させるには、テーブルが更新モードになっている必要があります。更新モードに切り換えるには、「UpdateBegin 関数」 189 ページを使用します。

UpdateBegin 関数

カラムの設定に使用される更新モードを選択します。

構文

bool **UltraLite_Cursor_iface::UpdateBegin()**

備考

更新モードの場合、プライマリ・キー内のカラムの修正はできません。このカーソルが、インデックスを使用しないで開かれたテーブルである場合、データは読み込み専用とみなされ、変更できません。

UltraLite_DatabaseManager クラス

同期リスナを管理し、Ultra Light データベースを削除できます。

構文

public **UltraLite_DatabaseManager**

基本クラス

- 「UltraLite_DatabaseManager_iface クラス」 192 ページ

メンバ

UltraLite_DatabaseManager のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「CreateDatabase 関数」 192 ページ
- 「DropDatabase 関数」 193 ページ
- 「OpenConnection 関数」 193 ページ
- 「Shutdown 関数」 194 ページ
- 「ValidateDatabase 関数」 194 ページ

UltraLite_DatabaseManager_iface クラス

接続とデータベースを管理します。

構文

public **UltraLite_DatabaseManager_iface**

派生クラス

- 「UltraLite_DatabaseManager クラス」 191 ページ

メンバ

UltraLite_DatabaseManager_iface のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「CreateDatabase 関数」 192 ページ
- 「DropDatabase 関数」 193 ページ
- 「OpenConnection 関数」 193 ページ
- 「Shutdown 関数」 194 ページ
- 「ValidateDatabase 関数」 194 ページ

備考

データベースを作成し、そのデータベースへの接続を確立することは、Ultra Light の使用に必要な最初の手順です。正しく接続してからデータ操作言語でデータベースを操作するようにしてください。

CreateDatabase 関数

新しいデータベースを作成します。

構文

```
bool UltraLite_DatabaseManager_iface::CreateDatabase(
  ULSqlcaBase & sqlca,
  ULValue const & access_parms,
  void const * coll,
  ULValue const & create_parms,
  void * reserved
)
```

パラメータ

- **sqlca**　初期化された sqlca。
- **access_parms**　データベースへのアクセスに使用される接続パラメータ
- **coll**　照合順
- **create_parms**　データベースの作成に使用されるパラメータ

- **reserved**　予約 (現在は未使用)

DropDatabase 関数

停止済みの既存のデータベースを消去します。

構文

```
bool UltraLite_DatabaseManager_iface::DropDatabase(
  ULSqlcaBase & sqlca,
  const ULValue & parms_string
)
```

パラメータ

- **sqlca**　初期化された sqlca。
- **parms_string**　データベース識別パラメータ。

備考

実行中のデータベースは消去できません。

OpenConnection 関数

既存のデータベースへの新しい接続を開きます。

構文

```
UltraLite_Connection * UltraLite_DatabaseManager_iface::OpenConnection(
  ULSqlcaBase & sqlca,
  ULValue const & parms_string
)
```

パラメータ

- **sqlca**　新しい接続に関連付ける初期化済みの sqlca。
- **parms_string**　接続文字列。

備考

この sqlca は新しい接続に関連付けられます。

- SQLE_CONNECTION_ALREADY_EXISTS - 指定した SQLCA と名前の (または名前のない) 接続は、すでに存在します。接続する前に、既存の接続を切断するか、CON パラメータを使用して別の接続名を指定してください。
- SQLE_INVALID_LOGON - 無効なユーザ ID または間違ったパスワードを入力しました。
- SQLE_INVALID_SQL_IDENTIFIER - C 言語インタフェースを通して、無効な識別子を指定しました。たとえば、カーソル名に NULL 文字列を指定した可能性があります。

- SQLE_TOO_MANY_CONNECTIONS - 同時データベース接続数の制限を超えました。

エラー情報を取得するには、対応する「ULSqlca クラス」 146 ページオブジェクトを使用します。

戻り値

- この関数が成功した場合、新しい接続オブジェクトが返されます。
- 失敗した場合は、NULL が返されます。

Shutdown 関数

すべてのデータベースを閉じ、データベース・マネージャを解放します。

構文

```
void UltraLite_DatabaseManager_iface::Shutdown(
  ULSqlcaBase & sqlca
)
```

パラメータ

- **sqlca**　初期化された sqlca。

備考

残りの関連オブジェクトは破棄されます。この関数を呼び出すと、データベース・マネージャは使用できなくなります (また、前に取得したオブジェクトも使用できなくなります)。

ValidateDatabase 関数

データベースで低レベルのインデックス検証を実行します。

構文

```
bool UltraLite_DatabaseManager_iface::ValidateDatabase(
  ULSqlcaBase & sqlca,
  ULValue const & start_parms,
  ul_u_short flags,
  ul_validate_callback_fn fn,
  ul_void * user_data
)
```

パラメータ

- **sqlca**　初期化された sqlca。
- **start_parms**　データベースを起動するためのパラメータ。
- **flags**　検証のタイプを制御するフラグ。
- **fn**　検証の進行状況の情報を受け取る関数。

- **user_data**　コールバックにより呼び出し元に送り返すユーザ・データ。

UltraLite_DatabaseSchema クラス

Ultra Light データベースのスキーマを表します。

構文

public **UltraLite_DatabaseSchema**

基本クラス

- 「UltraLite_SQLObject_iface クラス」 220 ページ
- 「UltraLite_DatabaseSchema_iface クラス」 197 ページ

メンバ

UltraLite_DatabaseSchema のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「AddRef 関数」 220 ページ
- 「GetCollationName 関数」 197 ページ
- 「GetConnection 関数」 220 ページ
- 「GetIFace 関数」 221 ページ
- 「GetPublicationCount 関数」 197 ページ
- 「GetPublicationID 関数」 198 ページ
- 「GetPublicationName 関数」 198 ページ
- 「GetTableCount 関数」 198 ページ
- 「GetTableName 関数」 198 ページ
- 「GetTableSchema 関数」 199 ページ
- 「IsCaseSensitive 関数」 199 ページ
- 「Release 関数」 221 ページ

UltraLite_DatabaseSchema_iface クラス

DatabaseSchema インタフェースです。

構文

public **UltraLite_DatabaseSchema_iface**

派生クラス

- 「UltraLite_DatabaseSchema クラス」 196 ページ

メンバ

UltraLite_DatabaseSchema_iface のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「GetCollationName 関数」 197 ページ
- 「GetPublicationCount 関数」 197 ページ
- 「GetPublicationID 関数」 198 ページ
- 「GetPublicationName 関数」 198 ページ
- 「GetTableCount 関数」 198 ページ
- 「GetTableName 関数」 198 ページ
- 「GetTableSchema 関数」 199 ページ
- 「IsCaseSensitive 関数」 199 ページ

GetCollationName 関数

現在の照合順の名前を取得します。

構文

ULValue **UltraLite_DatabaseSchema_iface::GetCollationName()**

戻り値

文字列を含む「ULValue クラス」 247 ページ。

GetPublicationCount 関数

データベース内のパブリケーション数を取得します。

構文

ul_publication_count **UltraLite_DatabaseSchema_iface::GetPublicationCount()**

備考

パブリケーション ID の範囲は、1 ～ 「GetPublicationCount 関数」 197 ページです。

GetPublicationID 関数

名前を指定して、1 から始まるパブリケーション ID を取得します。

構文

```
ul_u_short UltraLite_DatabaseSchema_iface::GetPublicationID(
  const ULValue & pub_id
)
```

パラメータ

- **pub_id**　1 から始まる順序数。

GetPublicationName 関数

1 から始まるインデックス ID を指定してパブリケーションの名前を取得します。

構文

```
ULValue UltraLite_DatabaseSchema_iface::GetPublicationName(
  const ULValue & pub_id
)
```

パラメータ

- **pub_id**　1 から始まる順序数。

GetTableCount 関数

データベース内のテーブルの数を返します。

構文

```
ul_table_num UltraLite_DatabaseSchema_iface::GetTableCount()
```

戻り値

- テーブルの数を表す整数。
- 接続が開いていない場合は、0。

GetTableName 関数

1 から始まるテーブル ID を指定してテーブルの名前を取得します。

構文

ULValue **UltraLite_DatabaseSchema_iface::GetTableName(**
  ul_table_num *tableID*
)

パラメータ

- **tableID**　1 から始まる順序数。

戻り値

指定されたテーブル ID で識別されたテーブルの名前。

備考

テーブル ID は、スキーマのアップグレード中に変更されることがあります。テーブルを正しく識別するには、名前でアクセスするか、キャッシュされている ID をスキーマのアップグレード後にリフレッシュします。テーブルが存在しない場合、返される「ULValue クラス」 247 ページオブジェクトは空です。

GetTableSchema 関数

1 から始まるテーブルの ID か名前を指定して TableSchema オブジェクトを取得します。

構文

UltraLite_TableSchema * **UltraLite_DatabaseSchema_iface::GetTableSchema(**
  const ULValue & *table_id*
)

パラメータ

- **table_id**　1 から始まる順序数。

戻り値

テーブルが存在しない場合は、UL_NULL。

IsCaseSensitive 関数

データベースで大文字と小文字が区別されるかどうかを調べます。

構文

bool **UltraLite_DatabaseSchema_iface::IsCaseSensitive()**

戻り値

- データベースで大文字と小文字が区別される場合は、true。
- それ以外の場合は、false。

備考

データベースで大文字と小文字が区別されるかどうかは、テーブルのインデックスと結果セットのソート方法に影響します。

UltraLite_IndexSchema クラス

Ultra Light テーブルのインデックスのスキーマを表します。

構文

public **UltraLite_IndexSchema**

基本クラス

- 「UltraLite_SQLObject_iface クラス」 220 ページ
- 「UltraLite_IndexSchema_iface クラス」 202 ページ

メンバ

UltraLite_IndexSchema のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「AddRef 関数」 220 ページ
- 「GetColumnCount 関数」 202 ページ
- 「GetColumnName 関数」 202 ページ
- 「GetConnection 関数」 220 ページ
- 「GetID 関数」 203 ページ
- 「GetIFace 関数」 221 ページ
- 「GetName 関数」 203 ページ
- 「GetReferencedIndexName 関数」 203 ページ
- 「GetReferencedTableName 関数」 204 ページ
- 「GetTableName 関数」 204 ページ
- 「IsColumnDescending 関数」 204 ページ
- 「IsForeignKey 関数」 204 ページ
- 「IsForeignKeyCheckOnCommit 関数」 205 ページ
- 「IsForeignKeyNullable 関数」 205 ページ
- 「IsPrimaryKey 関数」 205 ページ
- 「IsUniqueIndex 関数」 206 ページ
- 「IsUniqueKey 関数」 206 ページ
- 「Release 関数」 221 ページ

UltraLite_IndexSchema_iface クラス

IndexSchema インタフェースを表します。

構文

public **UltraLite_IndexSchema_iface**

派生クラス

- 「UltraLite_IndexSchema クラス」 201 ページ

メンバ

UltraLite_IndexSchema_iface のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「GetColumnCount 関数」 202 ページ
- 「GetColumnName 関数」 202 ページ
- 「GetID 関数」 203 ページ
- 「GetName 関数」 203 ページ
- 「GetReferencedIndexName 関数」 203 ページ
- 「GetReferencedTableName 関数」 204 ページ
- 「GetTableName 関数」 204 ページ
- 「IsColumnDescending 関数」 204 ページ
- 「IsForeignKey 関数」 204 ページ
- 「IsForeignKeyCheckOnCommit 関数」 205 ページ
- 「IsForeignKeyNullable 関数」 205 ページ
- 「IsPrimaryKey 関数」 205 ページ
- 「IsUniqueIndex 関数」 206 ページ
- 「IsUniqueKey 関数」 206 ページ

GetColumnCount 関数

インデックス内のカラム数を取得します。

構文

ul_column_num **UltraLite_IndexSchema_iface::GetColumnCount()**

GetColumnName 関数

インデックス内のカラムの位置を指定して、カラムの名前を取得します。

構文

ULValue **UltraLite_IndexSchema_iface::GetColumnName(**
  ul_column_num *col_id_in_index*
)

パラメータ

- **col_id_in_index**　インデックス内のカラムの位置を示す 1 から始まる順序数。

戻り値

カラムが存在しない場合は、空の「ULValue クラス」 247 ページオブジェクト。

備考

カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND。

GetID 関数

インデックスの ID を取得します。

構文

ul_index_num **UltraLite_IndexSchema_iface::GetID()**

戻り値

インデックスの ID。

GetName 関数

インデックスの名前を取得します。

構文

ULValue **UltraLite_IndexSchema_iface::GetName()**

GetReferencedIndexName 関数

関連付けられているプライマリ・インデックスの名前を取得します。

構文

ULValue **UltraLite_IndexSchema_iface::GetReferencedIndexName()**

戻り値

インデックスが外部キーではない場合は、空の「ULValue クラス」 247 ページオブジェクト。

備考

この関数は、外部キー専用です。

GetReferencedTableName 関数

関連付けられているプライマリ・テーブルの名前を取得します。

構文

ULValue **UltraLite_IndexSchema_iface::GetReferencedTableName()**

戻り値

インデックスが外部キーではない場合は、空の「ULValue クラス」 247 ページオブジェクト。

備考

このメソッドは、外部キー専用です。

GetTableName 関数

インデックスが含まれるテーブルの名前を取得します。

構文

ULValue **UltraLite_IndexSchema_iface::GetTableName()**

IsColumnDescending 関数

カラムが降順かどうかを調べます。

構文

bool **UltraLite_IndexSchema_iface::IsColumnDescending(**
  const ULValue & *column_name*
)

パラメータ

- **column_name**　カラムの名前。

戻り値

カラムが降順の場合は、true。

備考

カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND を設定します。

IsForeignKey 関数

インデックスが外部キーであるかどうかをチェックします。

構文

bool **UltraLite_IndexSchema_iface::IsForeignKey()**

戻り値

- インデックスが外部キーである場合は、true。
- インデックスが外部キーでない場合は、false。

備考

外部キー内のカラムは、別のテーブルの NULL 以外のユニーク・インデックスを参照することができます。

IsForeignKeyCheckOnCommit 関数

外部キーの参照整合性のチェックが、コミット時に行われるか、挿入時と更新時に行われるかを確認します。

構文

bool **UltraLite_IndexSchema_iface::IsForeignKeyCheckOnCommit()**

戻り値

- この外部キーが、コミット時に参照整合性をチェックする場合は、true。
- この外部キーが、挿入時に参照整合性をチェックする場合は、false。

IsForeignKeyNullable 関数

外部キーが NULL 入力可であるかどうかをチェックします。

構文

bool **UltraLite_IndexSchema_iface::IsForeignKeyNullable()**

戻り値

- インデックスが一意の外部キー制約である場合は、true。
- 外部キーが NULL 入力可でない場合は、false。

IsPrimaryKey 関数

インデックスがプライマリ・キーであるかどうかをチェックします。

構文

bool **UltraLite_IndexSchema_iface::IsPrimaryKey()**

戻り値

- インデックスがプライマリ・キーである場合は、true。
- インデックスがプライマリ・キーでない場合は、false。

備考

プライマリ・キー内のカラムでは NULL は許可されません。

IsUniqueIndex 関数

インデックスがユニークであるかどうかをチェックします。

構文

bool **UltraLite_IndexSchema_iface::IsUniqueIndex()**

戻り値

- インデックスがユニークである場合は、true。
- インデックスがユニークでない場合は、false。

IsUniqueKey 関数

インデックスがユニーク・キーであるかどうかをチェックします。

構文

bool **UltraLite_IndexSchema_iface::IsUniqueKey()**

戻り値

- インデックスがプライマリ・キーまたは一意性制約である場合は、true。
- インデックスがプライマリ・キーでも一意性制約でもない場合は、false。

備考

ユニーク・キー内のカラムでは NULL は許可されません。

UltraLite_PreparedStatement クラス

プレースホルダを使用して文を準備し、文の実行後にプレースホルダへ値を割り当てることができます。

構文

public **UltraLite_PreparedStatement**

基本クラス

- 「UltraLite_SQLObject_iface クラス」 220 ページ
- 「UltraLite_PreparedStatement_iface クラス」 208 ページ

メンバ

UltraLite_PreparedStatement のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「AddRef 関数」 220 ページ
- 「ExecuteQuery 関数」 208 ページ
- 「ExecuteStatement 関数」 208 ページ
- 「GetConnection 関数」 220 ページ
- 「GetIFace 関数」 221 ページ
- 「GetPlan 関数」 209 ページ
- 「GetPlan 関数」 209 ページ
- 「GetSchema 関数」 209 ページ
- 「GetStreamWriter 関数」 210 ページ
- 「HasResultSet 関数」 210 ページ
- 「Release 関数」 221 ページ
- 「SetParameter 関数」 210 ページ
- 「SetParameterNull 関数」 211 ページ

UltraLite_PreparedStatement_iface クラス

PreparedStatement インタフェースです。

構文

public **UltraLite_PreparedStatement_iface**

派生クラス

- 「UltraLite_PreparedStatement クラス」 207 ページ

メンバ

UltraLite_PreparedStatement_iface のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「ExecuteQuery 関数」 208 ページ
- 「ExecuteStatement 関数」 208 ページ
- 「GetPlan 関数」 209 ページ
- 「GetPlan 関数」 209 ページ
- 「GetSchema 関数」 209 ページ
- 「GetStreamWriter 関数」 210 ページ
- 「HasResultSet 関数」 210 ページ
- 「SetParameter 関数」 210 ページ
- 「SetParameterNull 関数」 211 ページ

ExecuteQuery 関数

SQL SELECT 文をクエリとして実行します。

構文

UltraLite_ResultSet * **UltraLite_PreparedStatement_iface::ExecuteQuery()**

戻り値

クエリの結果セット (ローのセット)。

ExecuteStatement 関数

SQL INSERT 文、DELETE 文、UPDATE 文のように、結果セットを返さない文を実行します。

構文

ul_s_long **UltraLite_PreparedStatement_iface::ExecuteStatement()**

GetPlan 関数

クエリ実行プランのテキストベースの記述を取得します。

構文

```
size_t UltraLite_PreparedStatement_iface::GetPlan(
  char * buffer,
  size_t size
)
```

パラメータ

- **buffer**　プランの記述を受信するバッファ。
- **size**　バッファのサイズ (ASCII 文字数)。

戻り値

クエリを実行するのに Ultra Light が使用するアクセス・プランを記述する文字列。

備考

この関数は、主に開発中の使用を目的とします。

GetPlan 関数

クエリ実行プランのテキストベースの記述をワイド文字で取得します。

構文

```
size_t UltraLite_PreparedStatement_iface::GetPlan(
  ul_wchar * buffer,
  size_t size
)
```

パラメータ

- **buffer**　プランの記述を受信するバッファ。
- **size**　バッファのサイズ (ul_wchar 数)。

戻り値

クエリを実行するのに Ultra Light が使用するアクセス・プランを記述する文字列。

備考

この関数は、主に開発中の使用を目的とします。

GetSchema 関数

結果セットのスキーマを取得します。

構文

UltraLite_ResultSetSchema * **UltraLite_PreparedStatement_iface::GetSchema()**

GetStreamWriter 関数

文字列データまたはバイナリ・データをパラメータにストリーミングするストリーム・ライタを取得します。

構文

```
UltraLite_StreamWriter * UltraLite_PreparedStatement_iface::GetStreamWriter(
  ul_column_num parameter_id
)
```

パラメータ

- **parameter_id**　カラム識別子。1 から始まる順序数またはカラム名です。

HasResultSet 関数

SQL 文に結果セットがあるかどうかを調べます。

構文

bool **UltraLite_PreparedStatement_iface::HasResultSet()**

戻り値

- この文が実行されたときに結果セットが生成される場合は、true。
- 結果セットが生成されない場合は、false。

SetParameter 関数

SQL 文のパラメータを設定します。

構文

```
void UltraLite_PreparedStatement_iface::SetParameter(
  ul_column_num parameter_id,
  ULValue const & value
)
```

パラメータ

- **parameter_id**　1 から始まるパラメータの順序。
- **value**　パラメータを設定する値。

SetParameterNull 関数

パラメータを NULL に設定します。

構文

```
void UltraLite_PreparedStatement_iface::SetParameterNull(
  ul_column_num parameter_id
)
```

パラメータ

- **parameter_id**　1 から始まるパラメータの順序。

UltraLite_ResultSet クラス

Ultra Light データベースの編集可能な結果セットを表します。

構文

public **UltraLite_ResultSet**

基本クラス

- 「UltraLite_SQLObject_iface クラス」 220 ページ
- 「UltraLite_ResultSet_iface クラス」 213 ページ
- 「UltraLite_Cursor_iface クラス」 183 ページ

メンバ

UltraLite_ResultSet のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「AddRef 関数」 220 ページ
- 「AfterLast 関数」 183 ページ
- 「BeforeFirst 関数」 184 ページ
- 「Delete 関数」 184 ページ
- 「DeleteNamed 関数」 213 ページ
- 「First 関数」 184 ページ
- 「Get 関数」 184 ページ
- 「GetConnection 関数」 220 ページ
- 「GetIFace 関数」 221 ページ
- 「GetRowCount 関数」 185 ページ
- 「GetSchema 関数」 213 ページ
- 「GetState 関数」 185 ページ
- 「GetStreamReader 関数」 185 ページ
- 「GetStreamWriter 関数」 186 ページ
- 「GetSuspend 関数 (旧式)」 186 ページ
- 「IsNull 関数」 186 ページ
- 「Last 関数」 187 ページ
- 「Next 関数」 187 ページ
- 「Previous 関数」 187 ページ
- 「Relative 関数」 187 ページ
- 「Release 関数」 221 ページ
- 「Set 関数」 188 ページ
- 「SetDefault 関数」 188 ページ
- 「SetNull 関数」 188 ページ
- 「SetSuspend 関数 (旧式)」 189 ページ
- 「Update 関数」 189 ページ
- 「UpdateBegin 関数」 189 ページ

備考

位置付け更新や削除を実行できる編集可能な結果セットです。

UltraLite_ResultSet_iface クラス

ResultSet インタフェースです。

構文

public **UltraLite_ResultSet_iface**

派生クラス

- 「UltraLite_ResultSet クラス」 212 ページ

メンバ

UltraLite_ResultSet_iface のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「DeleteNamed 関数」 213 ページ
- 「GetSchema 関数」 213 ページ

DeleteNamed 関数

現在のローを削除し、次の有効なローに移動します。

構文

```
bool UltraLite_ResultSet_iface:DeleteNamed(
  const ULValue & table_name
)
```

パラメータ

- **table_name**　テーブル名またはその相関 (同じテーブル名を共有する複数のカラムがデータベースに存在する場合に必要)。

GetSchema 関数

この結果セットのスキーマを取得します。

構文

UltraLite_ResultSetSchema * **UltraLite_ResultSet_iface::GetSchema()**

UltraLite_ResultSetSchema クラス

結果セットに関するスキーマ情報を取得します。たとえばカラム名、カラムの総数、カラム・スケール、カラム・サイズ、カラム SQL 型などです。

構文

public **UltraLite_ResultSetSchema**

基本クラス

- 「UltraLite_SQLObject_iface クラス」 220 ページ
- 「UltraLite_RowSchema_iface クラス」 215 ページ

メンバ

UltraLite_ResultSetSchema のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「AddRef 関数」 220 ページ
- 「GetBaseColumnName 関数」 215 ページ
- 「GetColumnCount 関数」 216 ページ
- 「GetColumnID 関数」 216 ページ
- 「GetColumnName 関数」 216 ページ
- 「GetColumnPrecision 関数」 217 ページ
- 「GetColumnScale 関数」 218 ページ
- 「GetColumnSize 関数」 218 ページ
- 「GetColumnSQLName 関数」 217 ページ
- 「GetColumnSQLType 関数」 218 ページ
- 「GetColumnType 関数」 219 ページ
- 「GetConnection 関数」 220 ページ
- 「GetIFace 関数」 221 ページ
- 「Release 関数」 221 ページ

UltraLite_RowSchema_iface クラス

RowSchema インタフェースです。

構文

public **UltraLite_RowSchema_iface**

派生クラス

- 「UltraLite_ResultSetSchema クラス」 214 ページ
- 「UltraLite_TableSchema クラス」 235 ページ

メンバ

UltraLite_RowSchema_iface のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「GetBaseColumnName 関数」 215 ページ
- 「GetColumnCount 関数」 216 ページ
- 「GetColumnID 関数」 216 ページ
- 「GetColumnName 関数」 216 ページ
- 「GetColumnPrecision 関数」 217 ページ
- 「GetColumnScale 関数」 218 ページ
- 「GetColumnSize 関数」 218 ページ
- 「GetColumnSQLName 関数」 217 ページ
- 「GetColumnSQLType 関数」 218 ページ
- 「GetColumnType 関数」 219 ページ

GetBaseColumnName 関数

結果セットのカラムのベース・カラム名に相関名またはエイリアスが存在する場合でも、そのベースとカラムの結合された名前を取得します。

構文

ULValue **UltraLite_RowSchema_iface::GetBaseColumnName(**
  ul_column_num *column_id*
)

パラメータ

- **column_id**　1 から始まる順序数。

戻り値

- 結合された「ULValue クラス」 247 ページオブジェクト。
- このカラムがテーブルの一部でない場合は、空の名前を返します。

備考

カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND が設定されます。

GetColumnCount 関数

テーブル内のカラム数を取得します。

構文

ul_column_num **UltraLite_RowSchema_iface::GetColumnCount()**

GetColumnID 関数

1 から始まるカラム ID を取得します。

構文

ul_column_num **UltraLite_RowSchema_iface::GetColumnID(**
  const ULValue & *column_name*
)

パラメータ

- **column_name**　カラムの名前。

戻り値

カラムが存在しない場合は、0 を返します。

備考

カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND を設定します。

GetColumnName 関数

1 から始まる ID を指定してカラムの名前を取得します。

構文

ULValue **UltraLite_RowSchema_iface::GetColumnName(**
  ul_column_num *column_id*
)

パラメータ

- **column_id**　1 から始まる順序数。

戻り値

カラムが存在しない場合は、空の「ULValue クラス」 247 ページオブジェクト。

備考

これは SELECT 文のエイリアスまたは相関名になります。

カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND を設定します。

GetColumnPrecision 関数

数値カラムの精度を取得します。

構文

```
size_t UltraLite_RowSchema_iface::GetColumnPrecision(
  const ULValue & column_id
)
```

パラメータ

- **column_id**　1 から始まる順序数。

戻り値

カラムが数値型ではないか、カラムが存在しない場合は、0 を返します。

備考

カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND を設定します。

カラム型が数値カラムではない場合は、SQLE_DATATYPE_NOT_ALLOWED を設定します。

GetColumnSQLName 関数

結果セット内のカラムの SQL 名を取得します。

構文

```
ULValue UltraLite_RowSchema_iface::GetColumnSQLName(
  ul_column_num column_id
)
```

パラメータ

- **column_id**　1 から始まる順序数。

戻り値

結合された「ULValue クラス」 247 ページオブジェクト。

備考

カラムにエイリアスがある場合は、その名前が使用されます。エイリアスがない場合、結果セット内のカラムがテーブルのカラムに対応する場合は、カラム名が使用されます。それ以外の場合、結合された名前は空です。

カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND が設定されます。

GetColumnSQLType 関数

カラムの SQL の型を取得します。

構文

```
ul_column_sql_type UltraLite_RowSchema_iface::GetColumnSQLType(
  const ULValue & column_id
)
```

パラメータ

- **column_id**　1 から始まる順序数。

戻り値

カラムが存在しない場合は、UL_SQLTYPE_BAD_INDEX。

備考

ulprotos.h の ul_column_sql_type を参照してください。

GetColumnScale 関数

数値カラムの位取りを取得します。

構文

```
size_t UltraLite_RowSchema_iface::GetColumnScale(
  const ULValue & column_id
)
```

パラメータ

- **column_id**　1 から始まる順序数。

戻り値

カラムが数値型ではないか、カラムが存在しない場合は、0 を返します。

備考

カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND を設定します。

カラム型が数値ではない場合は、SQLE_DATATYPE_NOT_ALLOWED を設定します。

GetColumnSize 関数

カラムのサイズを取得します。

構文

size_t **UltraLite_RowSchema_iface::GetColumnSize(**
  const ULValue & *column_id*
)

パラメータ

- **column_id**　1 から始まる順序数。

戻り値

カラムが存在しないか、カラムの型が可変長ではない場合は、0 を返します。

備考

カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND を設定します。

カラム型が UL_SQLTYPE_CHAR でも UL_SQLTYPE_BINARY でもない場合は、SQLE_DATATYPE_NOT_ALLOWED を設定します。

GetColumnType 関数

カラムの型を取得します。

構文

ul_column_storage_type **UltraLite_RowSchema_iface::GetColumnType(**
  const ULValue & *column_id*
)

パラメータ

- **column_id**　1 から始まる順序数。

戻り値

カラムが存在しない場合は、UL_TYPE_BAD_INDEX。

備考

ulprotos.h の ul_column_storage_type enum を参照してください。

UltraLite_SQLObject_iface クラス

SQLObject インタフェース

構文

public **UltraLite_SQLObject_iface**

派生クラス

- 「UltraLite_Connection クラス」 155 ページ
- 「UltraLite_DatabaseSchema クラス」 196 ページ
- 「UltraLite_IndexSchema クラス」 201 ページ
- 「UltraLite_PreparedStatement クラス」 207 ページ
- 「UltraLite_ResultSet クラス」 212 ページ
- 「UltraLite_ResultSetSchema クラス」 214 ページ
- 「UltraLite_StreamReader クラス」 222 ページ
- 「UltraLite_StreamWriter クラス」 226 ページ
- 「UltraLite_Table クラス」 227 ページ
- 「UltraLite_TableSchema クラス」 235 ページ

メンバ

UltraLite_SQLObject_iface のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「AddRef 関数」 220 ページ
- 「GetConnection 関数」 220 ページ
- 「GetIFace 関数」 221 ページ
- 「Release 関数」 221 ページ

AddRef 関数

オブジェクトの内部リファレンス・カウントを増やします。

構文

ul_ret_void **UltraLite_SQLObject_iface::AddRef()**

備考

オブジェクトを解放するには、この関数の各呼び出しと「Release 関数」 221 ページの呼び出しを対にする必要があります。

GetConnection 関数

Connection オブジェクトを取得します。

構文

UltraLite_Connection * **UltraLite_SQLObject_iface::GetConnection()**

戻り値

このオブジェクトに関連付けられている接続。

GetIFace 関数

今後の使用のために予約されています。

構文

ul_void * **UltraLite_SQLObject_iface::GetIFace(**
  ul_iface_id *iface*
)

パラメータ

- **iface**　今後の使用のために予約されています。

Release 関数

オブジェクトへの参照を解放します。

構文

ul_u_long **UltraLite_SQLObject_iface::Release()**

備考

すべての参照が削除されると、オブジェクトが解放されます。この関数は 1 回以上呼び出してください。「AddRef 関数」 220 ページを使用する場合は、それぞれの「AddRef 関数」 220 ページと対になるようにして呼び出す必要があります。

UltraLite_StreamReader クラス

Ultra Light StreamReader を表します。

構文

public **UltraLite_StreamReader**

基本クラス

- 「UltraLite_SQLObject_iface クラス」 220 ページ
- 「UltraLite_StreamReader_iface クラス」 223 ページ

メンバ

UltraLite_StreamReader のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「AddRef 関数」 220 ページ
- 「GetByteChunk 関数」 223 ページ
- 「GetConnection 関数」 220 ページ
- 「GetIFace 関数」 221 ページ
- 「GetLength 関数」 224 ページ
- 「GetStringChunk 関数」 224 ページ
- 「GetStringChunk 関数」 225 ページ
- 「Release 関数」 221 ページ
- 「SetReadPosition 関数」 225 ページ

UltraLite_StreamReader_iface クラス

StreamReader インタフェースです。

構文

public **UltraLite_StreamReader_iface**

派生クラス

- 「UltraLite_StreamReader クラス」 222 ページ

メンバ

UltraLite_StreamReader_iface のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「GetByteChunk 関数」 223 ページ
- 「GetLength 関数」 224 ページ
- 「GetStringChunk 関数」 224 ページ
- 「GetStringChunk 関数」 225 ページ
- 「SetReadPosition 関数」 225 ページ

備考

このインタフェースは、VARCHAR カラムと BINARY カラムの読み込みと検索をサポートしています。

GetByteChunk 関数

バッファ data に buffer_len バイトをコピーして、現在の StreamReader オフセットからバイトのチャンクを取得します。

構文

```
bool UltraLite_StreamReader_iface::GetByteChunk(
  ul_byte * data,
  size_t buffer_len,
  size_t * len_retn,
  bool * morebytes
)
```

パラメータ

- **data**　バイトの配列へのポインタ。
- **buffer_len**　バッファ (配列) の長さ。buffer_len は 0 以上であることが必要です。
- **len_retn**　出力パラメータ。返される長さです。
- **morebytes**　出力パラメータ。さらに読み込むバイトがある場合は、true。

備考

「SetReadPosition 関数」 225 ページが使用されないかぎり、前回の読み込みが終了したところからバイトが読み込まれます。

GetLength 関数

文字列またはバイナリの値の長さを取得します。

構文

```
size_t UltraLite_StreamReader_iface::GetLength(
  bool fetch_as_chars
)
```

パラメータ

- **fetch_as_chars**　バイト長の場合は false、文字長の場合は true。

戻り値

- バイナリ値の場合は、バイト数 (バイナリ値の場合 fetch_as_chars は無視)。
- 文字列値の場合は、文字数またはバイト数。

GetStringChunk 関数

バッファ str に buffer_len ワイド文字をコピーして、現在の StreamReader オフセットから文字列のチャンクを取得します。

構文

```
bool UltraLite_StreamReader_iface::GetStringChunk(
  ul_wchar * str,
  size_t buffer_len,
  size_t * len_retn,
  bool * morebytes
)
```

パラメータ

- **str**　ワイド文字の配列へのポインタ。
- **buffer_len**　バッファの長さ。
- **len_retn**　出力パラメータ。返される長さです。
- **morebytes**　出力パラメータ。さらに読み込む文字がある場合は、true。

備考

「SetReadPosition 関数」 225 ページが使用されないかぎり、前回の読み込みが終了したところから文字が読み込まれます。

GetStringChunk 関数

バッファ str に buffer_len バイトをコピーして、現在の StreamReader オフセットから文字列のチャンクを取得します。

構文

```
bool UltraLite_StreamReader_iface::GetStringChunk(
  char * str,
  size_t buffer_len,
  size_t * len_retn,
  bool * morebytes
)
```

パラメータ

- **str**　文字の配列へのポインタ。
- **buffer_len**　バッファ (配列) の長さ。buffer_len は 0 以上であることが必要です。
- **len_retn**　出力パラメータ。返される長さです。
- **morebytes**　出力パラメータ。さらに読み込む文字がある場合は、true。

備考

「SetReadPosition 関数」 225 ページが使用されないかぎり、前回の読み込みが終了したところから文字が読み込まれます。

SetReadPosition 関数

次の読み込みに使用されるデータ内のオフセットを設定します。

構文

```
bool UltraLite_StreamReader_iface::SetReadPosition(
  size_t offset,
  bool offset_in_chars
)
```

パラメータ

- **offset**　オフセット。
- **offset_in_chars**　offset が文字数の場合は、true。offset がバイト数の場合は、false。

UltraLite_StreamWriter クラス

Ultra Light StreamWriter を表します。

構文

public **UltraLite_StreamWriter**

基本クラス

- 「UltraLite_SQLObject_iface クラス」 220 ページ

メンバ

UltraLite_StreamWriter のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「AddRef 関数」 220 ページ
- 「GetConnection 関数」 220 ページ
- 「GetIFace 関数」 221 ページ
- 「Release 関数」 221 ページ

UltraLite_Table クラス

Ultra Light データベース内のテーブルを表します。

構文

public **UltraLite_Table**

基本クラス

- 「UltraLite_SQLObject_iface クラス」 220 ページ
- 「UltraLite_Table_iface クラス」 229 ページ
- 「UltraLite_Cursor_iface クラス」 183 ページ

メンバ

UltraLite_Table のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

UltraLite_Table_iface クラス

テーブル・インタフェースを表します。

構文

public **UltraLite_Table_iface**

派生クラス

- 「UltraLite_Table クラス」 227 ページ

メンバ

UltraLite_Table_iface のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「DeleteAllRows 関数」 229 ページ
- 「Find 関数」 230 ページ
- 「FindBegin 関数」 230 ページ
- 「FindFirst 関数」 230 ページ
- 「FindLast 関数」 231 ページ
- 「FindNext 関数」 231 ページ
- 「FindPrevious 関数」 232 ページ
- 「GetSchema 関数」 232 ページ
- 「Insert 関数」 232 ページ
- 「InsertBegin 関数」 232 ページ
- 「Lookup 関数」 233 ページ
- 「LookupBackward 関数」 233 ページ
- 「LookupBegin 関数」 234 ページ
- 「LookupForward 関数」 234 ページ
- 「TruncateTable 関数」 234 ページ

DeleteAllRows 関数

すべてのローをテーブルから削除します。

構文

bool **UltraLite_Table_iface::DeleteAllRows()**

戻り値

- 成功した場合は、true。
- 失敗した場合は、false。たとえばテーブルが開いていない場合や、SQL エラーが発生した場合などです。

備考

アプリケーションによっては、テーブル内のローをすべて削除してから、新しいデータ・セットをテーブルにダウンロードする方が便利なことがあります。接続で stop sync プロパティが設定されている場合は、削除されたローは同期されません。

別の接続からのコミットされていない挿入は削除されません。また、別の接続が「DeleteAllRows 関数」 229 ページを呼び出した後にロールバックを行った場合は、その接続からのコミットされていない削除は削除されません。

インデックスを使用しないでこのテーブルを開いた場合、テーブルは読み込み専用とみなされ、データの削除はできません。

Find 関数

この関数は、「FindFirst 関数」 230 ページと同じです。現在のインデックスに基づいて、テーブルを順方向にスキャンして完全一致のルックアップを行います。

構文

```
bool UltraLite_Table_iface::Find(
  ul_column_num ncols
)
```

パラメータ

- **ncols** 複合インデックスのための、ルックアップで使用するカラムの数。

FindBegin 関数

検索モードを開始することで、テーブルで新規に検索を実行する準備を行います。

構文

```
bool UltraLite_Table_iface::FindBegin()
```

備考

テーブルを開くのに使用されたインデックス内のカラムのみを設定できます。インデックスを使用しないでテーブルを開いた場合は、このメソッドを呼び出せません。

FindFirst 関数

現在のインデックスに基づいて、テーブルを順方向にスキャンして完全一致のルックアップを行います。

構文

bool **UltraLite_Table_iface::FindFirst(**
ul_column_num *ncols*
)

パラメータ

- **ncols** 複合インデックスのための、ルックアップで使用するカラムの数。

備考

検索する値を指定するには、インデックスの各カラムに値を設定します。カーソルは、インデックスの値と完全に一致した最初のローで停止します。インデックスの値に一致するローがない場合は、カーソルの位置は AfterLast() となり、false が返されます。

FindLast 関数

現在のインデックスに基づいて、テーブルを逆方向にスキャンして完全一致のルックアップを行います。

構文

bool **UltraLite_Table_iface::FindLast(**
ul_column_num *ncols*
)

パラメータ

- **ncols** 複合インデックスのための、ルックアップで使用するカラムの数。

備考

検索する値を指定するには、インデックスの各カラムに値を設定します。カーソルは、インデックスの値と完全に一致した最初のローに配置されます。インデックスの値に一致するローがない場合は、カーソルの位置は BeforeFirst() となり、false が返されます。

FindNext 関数

インデックスに完全に一致する次のローを取得します。

構文

bool **UltraLite_Table_iface::FindNext(**
ul_column_num *ncols*
)

パラメータ

- **ncols** 複合インデックスのための、ルックアップで使用するカラムの数。

戻り値

それ以上インデックスに一致するローがない場合は、false。

FindPrevious 関数

インデックスに完全に一致する前のローを取得します。

構文

bool **UltraLite_Table_iface::FindPrevious(**
ul_column_num *ncols*
)

パラメータ

- **ncols** 複合インデックスのための、ルックアップで使用するカラムの数。

戻り値

それ以上インデックスに一致するローがない場合は、false。この場合、カーソルは最初のローの前に配置されます。

GetSchema 関数

このテーブルのスキーマ・オブジェクトを取得します。

構文

UltraLite_TableSchema * **UltraLite_Table_iface::GetSchema()**

Insert 関数

新しいローをテーブルに挿入します。

構文

bool **UltraLite_Table_iface::Insert()**

備考

この操作を成功させるには、テーブルが挿入モードになっている必要があります。挿入モードに切り換えるには、「InsertBegin 関数」 232 ページを使用します。

InsertBegin 関数

設定されたカラムに対する挿入モードを選択します。

構文

bool **UltraLite_Table_iface::InsertBegin()**

備考

このモードでは、すべてのカラムを修正できます。インデックスを使用しないでこのテーブルを開いた場合、データは読み込み専用とみなされ、ローの挿入はできません。

Lookup 関数

この関数は、LookupForward と同じです。現在のインデックスに基づいて、テーブルを順方向にスキャンしてルックアップを行います。

構文

bool **UltraLite_Table_iface::Lookup(**
ul_column_num *ncols*
)

パラメータ

- **ncols** 複合インデックスのための、ルックアップで使用するカラムの数。

備考

結果としてカーソルの位置が AfterLast() となった場合は、戻り値は false です。

LookupBackward 関数

現在のインデックスに基づいて、テーブルを逆方向にスキャンしてルックアップを行います。

構文

bool **UltraLite_Table_iface::LookupBackward(**
ul_column_num *ncols*
)

パラメータ

- **ncols** 複合インデックスのための、ルックアップで使用するカラムの数。

戻り値

結果としてカーソルの位置が BeforeFirst() となった場合は、戻り値は false です。

備考

検索する値を指定するには、インデックスの各カラムに値を設定します。カーソルは、インデックスの値に一致するか、それより少ない値の最後のローで停止します。複合インデックスの場合、ncols はルックアップで使用するカラムの数を指定します。

LookupBegin 関数

ルックアップ・モードを開始することで、テーブルで新規に検索を実行する準備を行います。

構文

bool **UltraLite_Table_iface::LookupBegin()**

備考

テーブルを開くのに使用されたインデックス内のカラムのみを設定できます。インデックスを使用しないでテーブルを開いた場合は、このメソッドを呼び出せません。

LookupForward 関数

現在のインデックスに基づいて、テーブルを順方向にスキャンしてルックアップを行います。

構文

bool **UltraLite_Table_iface::LookupForward(**
ul_column_num *ncols*
)

パラメータ

- **ncols** 複合インデックスのための、ルックアップで使用するカラムの数。

戻り値

結果としてカーソルの位置が AfterLast() となった場合は、戻り値は false です。

備考

検索する値を指定するには、インデックスの各カラムに値を設定します。カーソルは、インデックスの値に一致するか、それより少ない値の最後のローで停止します。複合インデックスの場合、ncols はルックアップで使用するカラムの数を指定します。

TruncateTable 関数

テーブルをトランケートし、STOP SYNCHRONIZATION DELETE を一時的にアクティブにします。

構文

bool **UltraLite_Table_iface::TruncateTable()**

備考

インデックスを使用しないでこのテーブルを開いた場合、テーブルは読み込み専用とみなされ、データの削除はできません。

UltraLite_TableSchema クラス

テーブル・スキーマを表します。

構文

public **UltraLite_TableSchema**

基本クラス

- 「UltraLite_SQLObject_iface クラス」 220 ページ
- 「UltraLite_TableSchema_iface クラス」 237 ページ
- 「UltraLite_RowSchema_iface クラス」 215 ページ

メンバ

UltraLite_TableSchema のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「AddRef 関数」 220 ページ
- 「GetBaseColumnName 関数」 215 ページ
- 「GetColumnCount 関数」 216 ページ
- 「GetColumnDefault 関数」 237 ページ
- 「GetColumnID 関数」 216 ページ
- 「GetColumnName 関数」 216 ページ
- 「GetColumnPrecision 関数」 217 ページ
- 「GetColumnScale 関数」 218 ページ
- 「GetColumnSize 関数」 218 ページ
- 「GetColumnSQLName 関数」 217 ページ
- 「GetColumnSQLType 関数」 218 ページ
- 「GetColumnType 関数」 219 ページ
- 「GetConnection 関数」 220 ページ
- 「GetGlobalAutoincPartitionSize 関数」 238 ページ
- 「GetID 関数」 238 ページ
- 「GetIFace 関数」 221 ページ
- 「GetIndexCount 関数」 238 ページ
- 「GetIndexName 関数」 239 ページ
- 「GetIndexSchema 関数」 239 ページ
- 「GetName 関数」 240 ページ
- 「GetOptimalIndex 関数」 240 ページ
- 「GetPrimaryKey 関数」 240 ページ
- 「GetPublicationPredicate 関数」 240 ページ
- 「GetUploadUnchangedRows 関数」 241 ページ
- 「InPublication 関数」 241 ページ
- 「IsColumnAutoinc 関数」 242 ページ
- 「IsColumnCurrentDate 関数」 242 ページ
- 「IsColumnCurrentTime 関数」 243 ページ
- 「IsColumnCurrentTimestamp 関数」 243 ページ
- 「IsColumnGlobalAutoinc 関数」 244 ページ
- 「IsColumnInIndex 関数」 244 ページ
- 「IsColumnNewUUID 関数」 245 ページ
- 「IsColumnNullable 関数」 245 ページ
- 「IsNeverSynchronized 関数」 246 ページ
- 「Release 関数」 221 ページ

UltraLite_TableSchema_iface クラス

TableSchema インタフェースです。

構文

public **UltraLite_TableSchema_iface**

派生クラス

- 「UltraLite_TableSchema クラス」 235 ページ

メンバ

UltraLite_TableSchema_iface のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「GetColumnDefault 関数」 237 ページ
- 「GetGlobalAutoincPartitionSize 関数」 238 ページ
- 「GetID 関数」 238 ページ
- 「GetIndexCount 関数」 238 ページ
- 「GetIndexName 関数」 239 ページ
- 「GetIndexSchema 関数」 239 ページ
- 「GetName 関数」 240 ページ
- 「GetOptimalIndex 関数」 240 ページ
- 「GetPrimaryKey 関数」 240 ページ
- 「GetPublicationPredicate 関数」 240 ページ
- 「GetUploadUnchangedRows 関数」 241 ページ
- 「InPublication 関数」 241 ページ
- 「IsColumnAutoinc 関数」 242 ページ
- 「IsColumnCurrentDate 関数」 242 ページ
- 「IsColumnCurrentTime 関数」 243 ページ
- 「IsColumnCurrentTimestamp 関数」 243 ページ
- 「IsColumnGlobalAutoinc 関数」 244 ページ
- 「IsColumnInIndex 関数」 244 ページ
- 「IsColumnNewUUID 関数」 245 ページ
- 「IsColumnNullable 関数」 245 ページ
- 「IsNeverSynchronized 関数」 246 ページ

GetColumnDefault 関数

カラムのデフォルト値が存在する場合は取得します。

構文

```
ULValue UltraLite_TableSchema_iface::GetColumnDefault(
  const ULValue & column_id
)
```

パラメータ

- **column_id**　カラムの ID 番号。1 から始まる順序数にしてください。テーブルの最初のカラムの ID 値は 1 です。

戻り値

- 文字列として含まれるデフォルトを返します。
- カラムにデフォルト値が含まれていない場合は空になります。

備考

カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND を設定します。

GetGlobalAutoincPartitionSize 関数

分割サイズを取得します。

構文

```
bool UltraLite_TableSchema_iface::GetGlobalAutoincPartitionSize(
  const ULValue & column_id,
  ul_u_big * size
)
```

パラメータ

- **column_id**　カラムの ID 番号。1 から始まる順序数にしてください。テーブルの最初のカラムの ID 値は 1 です。
- **size**　出力パラメータ。カラムの分割サイズ。テーブルのすべてのグローバル・オートインクリメント・カラムは、同じグローバル・オートインクリメントの分割サイズを共有します。

戻り値

グローバル・オートインクリメント・カラムの分割サイズ。

GetID 関数

テーブル ID を取得します。

構文

```
ul_table_num UltraLite_TableSchema_iface::GetID()
```

GetIndexCount 関数

テーブル内のインデックス数を取得します。

構文

ul_index_num **UltraLite_TableSchema_iface::GetIndexCount()**

戻り値

テーブル内のインデックス数。

備考

インデックスの ID とカウントは、スキーマのアップグレード中に変更されることがあります。インデックスを正しく識別するには、名前でアクセスするか、キャッシュされている ID とカウントをスキーマのアップグレード後にリフレッシュします。

GetIndexName 関数

1 から始まる ID を指定してインデックスの名前を取得します。

構文

```
ULValue UltraLite_TableSchema_iface::GetIndexName(
  ul_index_num index_id
)
```

パラメータ

- **index_id**　1 から始まる順序数。

戻り値

インデックスが存在しない場合、「ULValue クラス」 247 ページオブジェクトは空になります。

備考

インデックスの ID とカウントは、スキーマのアップグレード中に変更されることがあります。インデックスを正しく識別するには、名前でアクセスするか、キャッシュされている ID とカウントをスキーマのアップグレード後にリフレッシュします。

GetIndexSchema 関数

指定された名前または ID で IndexSchema オブジェクトを取得します。

構文

```
UltraLite_IndexSchema * UltraLite_TableSchema_iface::GetIndexSchema(
  const ULValue & index_id
)
```

パラメータ

- **index_id**　インデックスを識別する名前または ID 番号。

戻り値

インデックスが存在しない場合は、UL_NULL。

GetName 関数

テーブルの名前を取得します。

構文

ULValue **UltraLite_TableSchema_iface::GetName()**

戻り値

文字列としてのテーブル名。

GetOptimalIndex 関数

カラム値を検索するのに最適なインデックスを特定します。

構文

ULValue **UltraLite_TableSchema_iface::GetOptimalIndex(**
  const ULValue & *column_id*
)

パラメータ

- **column_id**　カラムの ID 番号。1 から始まる順序数にしてください。テーブルの最初のカラムの ID 値は 1 です。

戻り値

インデックスの名前。

GetPrimaryKey 関数

テーブルのプライマリ・キーを取得します。

構文

UltraLite_IndexSchema * **UltraLite_TableSchema_iface::GetPrimaryKey()**

GetPublicationPredicate 関数

文字列としてのパブリケーション述部を取得します。

構文

ULValue **UltraLite_TableSchema_iface::GetPublicationPredicate(**
  const ULValue & *publication_name*
)

パラメータ

- **publication_name**　パブリケーションの名前。

戻り値

指定されたパブリケーションのパブリケーション述部文字列。

備考

パブリケーションが存在しない場合は、SQLE_PUBLICATION_NOT_FOUND が設定されます。

GetUploadUnchangedRows 関数

データベースが、変更されていないローをアップロードするように設定されたかどうかをチェックします。

構文

bool **UltraLite_TableSchema_iface::GetUploadUnchangedRows()**

戻り値

- 同期時に常にすべてのローをアップロードするようにマーク付けされている場合は、true。
- 変更されたローのみをアップロードするようにマーク付けされている場合は、false。

備考

未変更のローと変更済みのローをアップロードするように設定されたテーブルは、allsync テーブルと呼ばれることもあります。

InPublication 関数

テーブルが、指定されたパブリケーションに含まれているかどうかをチェックします。

構文

bool **UltraLite_TableSchema_iface::InPublication(**
  const ULValue & *publication_name*
)

パラメータ

- **publication_name**　パブリケーションの名前。

戻り値

- テーブルがパブリケーションに含まれている場合は、true。
- テーブルがパブリケーションに含まれていない場合は、false。

備考

パブリケーションが存在しない場合は、SQLE_PUBLICATION_NOT_FOUND が設定されます。

IsColumnAutoinc 関数

指定されたカラムのデフォルトがオートインクリメントに設定されているかどうかをチェックします。

構文

```
bool UltraLite_TableSchema_iface::IsColumnAutoinc(
  const ULValue & column_id
)
```

パラメータ

- **column_id**　カラムの ID 番号。1 から始まる順序数にしてください。テーブルの最初のカラムの ID 値は 1 です。

戻り値

- カラムのデフォルトがオートインクリメントに設定されている場合は、true。
- カラムがオートインクリメントでない場合は、false。

備考

カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND を設定します。

IsColumnCurrentDate 関数

指定されたカラムのデフォルトが、現在の日付に設定されているかどうかをチェックします。

構文

```
bool UltraLite_TableSchema_iface::IsColumnCurrentDate(
  const ULValue & column_id
)
```

パラメータ

- **column_id**　カラムの ID 番号。1 から始まる順序数にしてください。テーブルの最初のカラムの ID 値は 1 です。

戻り値

- カラムに、デフォルト値として現在の日付が存在する場合は、true。
- カラムのデフォルトが現在の日付でない場合は、false。

IsColumnCurrentTime 関数

指定されたカラムのデフォルトが、現在の時刻に設定されているかどうかをチェックします。

構文

```
bool UltraLite_TableSchema_iface::IsColumnCurrentTime(
  const ULValue & column_id
)
```

パラメータ

- **column_id**　カラムの ID 番号。1 から始まる順序数にしてください。テーブルの最初のカラムの ID 値は 1 です。

戻り値

- カラムに、デフォルト値として現在の時刻が存在する場合は、true。
- そうでない場合は、false。

IsColumnCurrentTimestamp 関数

指定されたカラムのデフォルトが、現在のタイムスタンプに設定されているかどうかをチェックします。

構文

```
bool UltraLite_TableSchema_iface::IsColumnCurrentTimestamp(
  const ULValue & column_id
)
```

パラメータ

- **column_id**　カラムの ID 番号。1 から始まる順序数にしてください。テーブルの最初のカラムの ID 値は 1 です。

戻り値

- カラムに、デフォルト値として現在のタイムスタンプが存在する場合は、true。
- そうでない場合は、false。

IsColumnGlobalAutoinc 関数

指定されたカラムのデフォルトがオートインクリメントに設定されているかどうかをチェックします。

構文

```
bool UltraLite_TableSchema_iface::IsColumnGlobalAutoinc(
  const ULValue & column_id
)
```

パラメータ

- **column_id**　カラムの ID 番号。1 から始まる順序数にしてください。テーブルの最初のカラムの ID 値は 1 です。

戻り値

- カラムがオートインクリメントの場合は、true。
- オートインクリメントでない場合は、false。

備考

カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND が設定されます。

IsColumnInIndex 関数

テーブルが、指定されたインデックスに含まれているかどうかをチェックします。

構文

```
bool UltraLite_TableSchema_iface::IsColumnInIndex(
  const ULValue & column_id,
  const ULValue & index_id
)
```

パラメータ

- **column_id**　カラムを識別する 1 から始まる順序数。column_id を取得するには、「GetColumnCount 関数」 216 ページを呼び出します。
- **index_id**　インデックスを識別する 1 から始まる順序数。テーブル内のインデックス数を取得するには、「GetIndexCount 関数」 238 ページを呼び出します。

戻り値

- カラムがインデックスに含まれている場合は、true。
- カラムがインデックスに含まれていない場合は、false。

備考

カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND を設定します。

インデックスが存在しない場合は、SQLE_INDEX_NOT_FOUND に設定されます。

IsColumnNewUUID 関数

指定されたカラムのデフォルトが新しい UUID に設定されているかどうかをチェックします。

構文

```
bool UltraLite_TableSchema_iface::IsColumnNewUUID(
  const ULValue & column_id
)
```

パラメータ

- **column_id**　カラムの ID 番号。1 から始まる順序数にしてください。テーブルの最初のカラムの ID 値は 1 です。

戻り値

- カラムに、デフォルト値として新しい UUID が存在する場合は、true。
- カラムのデフォルトが新しい UUID でない場合は、false。

IsColumnNullable 関数

指定されたカラムが NULL 入力可であるかどうかをチェックします。

構文

```
bool UltraLite_TableSchema_iface::IsColumnNullable(
  const ULValue & column_id
)
```

パラメータ

- **column_id**　カラムの ID 番号。1 から始まる順序数にしてください。テーブルの最初のカラムの ID 値は 1 です。指定されたカラムは、インデックス内の最初のカラムですが、インデックスには複数のカラムがある場合があります。

戻り値

- カラムが NULL 入力可の場合は、true。
- NULL 入力不可の場合は、false。

備考

カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND を設定します。

IsNeverSynchronized 関数

テーブルが、まったく同期されないようにマーク付けされているかどうかをチェックします。

構文

bool **UltraLite_TableSchema_iface::IsNeverSynchronized()**

戻り値

- テーブルが同期から省かれている場合は、true。まったく同期されないようにマーク付けされているテーブルは、パブリケーションに含まれているものであっても、まったく同期されていません。このようなテーブルは、nosync テーブルと呼ばれることもあります。
- テーブルが同期可能なテーブルに含まれる場合は、false。

ULValue クラス

構文

public **ULValue**

備考

「ULValue クラス」 247 ページです。

「ULValue クラス」 247 ページは、Ultra Light カーソルに格納されるデータ型に対するラッパーです。このため、データ型を気にすることなくデータを格納でき、Ultra Light C++ コンポーネントとの間で値をやり取りするために使用されます。

「ULValue クラス」 247 ページには、多数のコンストラクタとキャスト演算子が含まれるため、多くの場合、「ULValue クラス」 247 ページを明示的にインスタンス化しなくても、「ULValue クラス」 247 ページをシームレスに使用できます。

任意の基本 C++ データ型からオブジェクトを構成したり割り当てたりすることができます。任意の基本 C++ データ型にキャストすることもできます。

```
 x( 5 );       ULValue// Example of ULValue's constructor
 y = 5;        ULValue// Example of ULValue's assignment operator
int z = y;           // Example of ULValue's cast operator
```

この例は、文字列でも使用できます。

```
 x( UL_TEXT( ULValue"hello" ) );
 y = UL_TEXT( ULValue"hello" );
y.( buffer, BUFFER_LEN );    GetString// NOTE, there is no cast operator
```

多くの場合、「ULValue クラス」 247 ページオブジェクトの構成はコンパイラによって自動的に行われるため、明示的に構成する必要はありません。たとえば、カラムから値をフェッチするには、次の例を使用できます。

```
int x = table->Get( UL_TEXT( "my_column" ) );
```

table->Get() 呼び出しは「ULValue クラス」 247 ページオブジェクトを返します。C++ は、整数に変換するために、キャスト演算子を自動的に呼び出します。同様に、table->Get() 呼び出しは「ULValue クラス」 247 ページパラメータをカラム識別子として使用します。これにより、フェッチされるカラムが決まります。"my_column" リテラル文字列は「ULValue クラス」 247 ページオブジェクトに自動的に変換されます。

メンバ

ULValue のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「ULValue 関数」 259 ページ
- 「ULValue 関数」 259 ページ
- 「ULValue 関数」 259 ページ
- 「ULValue 関数」 259 ページ
- 「ULValue 関数」 260 ページ
- 「ULValue 関数」 260 ページ
- 「~ULValue 関数」 263 ページ

GetBinary 関数

現在の値を取り出してバイナリ・バッファに格納しますが、必要に応じてキャストが行われます。

構文

```
void ULValue::GetBinary(
  p_ul_binary bin,
  size_t len
)
```

パラメータ

- **bin**　バイトを受け取るバイナリ構造体。
- **len**　バッファの長さ。

備考

バッファが小さすぎる場合、値はトランケートされます。最大で len の文字が、指定されたバッファにコピーされます。

GetBinary 関数

現在の値を取り出してバイナリ・バッファに格納しますが、必要に応じてキャストが行われます。バッファが小さすぎる場合は、値はトランケートされます。

構文

```
void ULValue::GetBinary(
  ul_byte * dst,
  size_t len,
  size_t * retr_len
)
```

パラメータ

- **dst**　バイトを受け取るバッファ。
- **len**　バッファの長さ。
- **retr_len**　出力パラメータ。実際に返されたバイト数です。

備考

最大で len のバイトが、指定されたバッファにコピーされます。実際にコピーされたバイト数は、retr_len で返されます。

GetBinaryLength 関数

バイナリ値の長さを取得します。

構文

size_t **ULValue::GetBinaryLength()**

戻り値

「GetBinary 関数」 249 ページによって返されるバイナリ値の保持に必要なバイト数。

GetCombinedStringItem 関数

結合された名前の一部を取り出して文字列バッファに格納しますが、必要に応じてキャストが行われます。

構文

```
void ULValue::GetCombinedStringItem(
  ul_u_short selector,
  char * dst,
  size_t len
)
```

パラメータ

- **selector**　選択した内部値。
- **dst**　文字列値を受け取るバッファ。
- **len**　dst のバイト単位の長さ。

備考

値が結合されていない場合は、空の文字列がコピーされます。出力文字列は、常に NULL で終了します。バッファが小さすぎる場合、値はトランケートされます。NULL ターミネータを含め、最大で len の文字が指定されたバッファにコピーされます。

GetCombinedStringItem 関数

結合された文字列値の選択した部分を取得します。

構文

```
void ULValue::GetCombinedStringItem(
  ul_u_short selector,
  ul_wchar * dst,
  size_t len
)
```

パラメータ

- **selector**　選択した内部値。
- **dst**　文字列値を受け取るバッファ。
- **len**　dst のワイド文字単位の長さ。

GetString 関数

現在の値を取り出して文字列バッファに格納しますが、必要に応じてキャストが行われます。

構文

```
void ULValue::GetString(
  char * dst,
  size_t len
)
```

パラメータ

- **dst**　文字列値を受け取るバッファ。
- **len**　dst のバイト単位の長さ。

備考

出力文字列は、常に NULL で終了します。バッファが小さすぎる場合、値はトランケートされます。NULL ターミネータを含め、最大で len の文字が指定されたバッファにコピーされます。

GetString 関数

現在の値を取り出して文字列バッファに格納しますが、必要に応じてキャストが行われます。

構文

```
void ULValue::GetString(
  ul_wchar * dst,
  size_t len
)
```

パラメータ

- **dst**　文字列値を受け取るバッファ。
- **len**　dst のワイド文字単位の長さ。

GetStringLength 関数

文字列の長さを取得します。

構文

size_t **ULValue::GetStringLength(**
  bool *fetch_as_chars*
)

パラメータ

- **fetch_as_chars**　バイト長の場合は false、文字長の場合は true。

戻り値

いずれかの「GetString 関数」 251 ページメソッドによって返される文字列を保持するために必要なバイト数または文字数 (NULL ターミネータを含まない)。

例

使用方法を次に示します。

```
len = v.GetStringLength();
dst = new char[ len + 1 ];
( dst, len + 1 ); GetString
```

ワイド文字アプリケーションの場合の使用方法を次に示します。

```
len = v.GetStringLength( true );
dst = new ul_wchar[ len + 1 ];
( dst, len + 1 ); GetString
```

InDatabase 関数

値がデータベース内にあるかどうかをチェックします。

構文

bool **ULValue::InDatabase()**

戻り値

- このオブジェクトがカーソルのフィールドを参照している場合は、true。
- そうでない場合は、false。

IsNull 関数

「ULValue クラス」 247 ページオブジェクトが空であるかどうかをチェックします。

構文

bool **ULValue::IsNull()**

戻り値

- オブジェクトが、空の「ULValue クラス」 247 ページオブジェクトであるか、NULL に設定されたカーソルのフィールドを参照している場合は、true。
- それ以外の場合は、false。

SetBinary 関数

指定したバイナリ・バッファを参照する値を設定します。

構文

```
void ULValue::SetBinary(
  ul_byte * src,
  size_t len
)
```

パラメータ

- **src**　バイトのバッファ。
- **len**　バッファの長さ。

備考

値が使用されないかぎり、指定したバッファからバイトはコピーされません。

SetString 関数

「ULValue クラス」 247 ページを文字列にキャストします。

構文

```
void ULValue::SetString(
  const char * val,
  size_t len
)
```

パラメータ

- **val**　この「ULValue クラス」 247 ページの NULL で終了された文字列表現へのポインタ。
- **len**　文字列の長さ。

SetString 関数

「ULValue クラス」 247 ページを Unicode 文字列にキャストします。

構文

```
void ULValue::SetString(
  const ul_wchar * val,
  size_t len
)
```

パラメータ

- **val**　この「ULValue クラス」 247 ページの NULL で終了された Unicode 文字列表現へのポインタ。
- **len**　文字列の長さ。

StringCompare 関数

文字列、または「ULValue クラス」 247 ページオブジェクトの文字列表現を比較します。

構文

```
ul_compare ULValue::StringCompare(
  const ULValue & value
)
```

パラメータ

- **value**　比較文字列。

戻り値

- 文字列が同じ場合は、0。
- 現在の値が value より小さい場合は、-1。
- 現在の値が value より大きい場合は、1。
- いずれかの「ULValue クラス」 247 ページオブジェクトで sqlca が設定されていない場合は、-3。
- いずれかの「ULValue クラス」 247 ページオブジェクトの文字列表現が UL_NULL の場合は、-2。

ULValue 関数

「ULValue クラス」 247 ページを構成します。

構文

```
ULValue::ULValue()
```

ULValue 関数

既存の「ULValue クラス」 247 ページからコピーして新しいクラスを構成します。

構文

```
ULValue::ULValue(
 const ULValue & vSrc
)
```

パラメータ

- **vSrc** 「ULValue クラス」 247 ページとして扱われる値。

ULValue 関数

bool から「ULValue クラス」 247 ページを構成します。

構文

```
ULValue::ULValue(
 bool val
)
```

パラメータ

- **val** 「ULValue クラス」 247 ページとして扱われる bool 値。

ULValue 関数

short から「ULValue クラス」 247 ページを構成します。

構文

```
ULValue::ULValue(
 short val
)
```

パラメータ

- **val** 「ULValue クラス」 247 ページとして扱われる short 値。

ULValue 関数

long から「ULValue クラス」 247 ページを構成します。

構文

```
ULValue::ULValue(
  long val
)
```

パラメータ

- **val** 「ULValue クラス」 247 ページとして扱われる long 値。

ULValue 関数

int から「ULValue クラス」 247 ページを構成します。

構文

```
ULValue::ULValue(
  int val
)
```

パラメータ

- **val** 「ULValue クラス」 247 ページとして扱われる INTEGER 値。

ULValue 関数

unsigned INTEGER から「ULValue クラス」 247 ページを構成します。

構文

```
ULValue::ULValue(
  unsigned int val
)
```

パラメータ

- **val** 「ULValue クラス」 247 ページとして扱われる unsigned INTEGER 値。

ULValue 関数

FLOAT から「ULValue クラス」 247 ページを構成します。

構文

```
ULValue::ULValue(
  float val
)
```

パラメータ

- **val** 「ULValue クラス」 247 ページとして扱われる FLOAT 値。

ULValue 関数

DOUBLE から「ULValue クラス」 247 ページを構成します。

構文

```
ULValue::ULValue(
 double val
)
```

パラメータ

- **val** 「ULValue クラス」 247 ページとして扱われる DOUBLE 値。

ULValue 関数

unsigned CHAR から「ULValue クラス」 247 ページを構成します。

構文

```
ULValue::ULValue(
 unsigned char val
)
```

パラメータ

- **val** 「ULValue クラス」 247 ページとして扱われる unsigned CHAR 値。

ULValue 関数

unsigned SHORT から「ULValue クラス」 247 ページを構成します。

構文

```
ULValue::ULValue(
 unsigned short val
)
```

パラメータ

- **val** 「ULValue クラス」 247 ページとして扱われる unsigned SHORT 値。

ULValue 関数

unsigned LONG から「ULValue クラス」 247 ページを構成します。

構文

```
ULValue::ULValue(
 unsigned long val
)
```

パラメータ

- **val** 「ULValue クラス」 247 ページとして扱われる unsigned LONG 値。

ULValue 関数

ul_u_big から「ULValue クラス」 247 ページを構成します。

構文

```
ULValue::ULValue(
 const ul_u_big & val
)
```

パラメータ

- **val** 「ULValue クラス」 247 ページとして扱われる ul_u_big 値。

ULValue 関数

ul_s_big から「ULValue クラス」 247 ページを構成します。

構文

```
ULValue::ULValue(
 const ul_s_big & val
)
```

パラメータ

- **val** 「ULValue クラス」 247 ページとして扱われる ul_s_big 値。

ULValue 関数

ul_binary から「ULValue クラス」 247 ページを構成します。

構文

```
ULValue::ULValue(
 const p_ul_binary val
)
```

パラメータ

- **val** 「ULValue クラス」 247 ページとして扱われる ul_binary 値。

ULValue 関数

datetime から「ULValue クラス」 247 ページを構成します。

構文

```
ULValue::ULValue(
 DECL_DATETIME & val
)
```

パラメータ

- **val** 「ULValue クラス」 247 ページとして扱われる DATETIME 値。

ULValue 関数

STRING から「ULValue クラス」 247 ページを構成します。

構文

```
ULValue::ULValue(
 const char * val
)
```

パラメータ

- **val** 「ULValue クラス」 247 ページとして扱われる文字列へのポインタ。

ULValue 関数

Unicode 文字列から「ULValue クラス」 247 ページを構成します。

構文

```
ULValue::ULValue(
 const ul_wchar * val
)
```

パラメータ

- **val** 「ULValue クラス」 247 ページとして扱われる Unicode 文字列へのポインタ。

ULValue 関数

文字のバッファから「ULValue クラス」 247 ページを構成します。

構文

```
ULValue::ULValue(
 const char * val,
```

```
  size_t len
)
```

パラメータ

- **val** 「ULValue クラス」 247 ページとして扱われる文字列を保持するバッファ。
- **len** バッファの長さ。

ULValue 関数

Unicode 文字のバッファから「ULValue クラス」 247 ページを構成します。

構文

```
ULValue::ULValue(
  const ul_wchar * val,
  size_t len
)
```

パラメータ

- **val** 「ULValue クラス」 247 ページとして扱われる文字列を保持するバッファ。
- **len** バッファの長さ。

ULValue 関数

GUID 構造体から「ULValue クラス」 247 ページを構成します。

構文

```
ULValue::ULValue(
  GUID & val
)
```

パラメータ

- **val** 「ULValue クラス」 247 ページとして扱われる GUID 値。

DECL_DATETIME 演算子

「ULValue クラス」 247 ページを datetime にキャストします。

構文

```
ULValue::operator DECL_DATETIME()
```


GUID 演算子

「ULValue クラス」 247 ページを GUID 構造体にキャストします。

構文

ULValue::operator GUID()

bool 演算子

「ULValue クラス」 247 ページを bool にキャストします。

構文

ULValue::operator bool()

double 演算子

「ULValue クラス」 247 ページを double にキャストします。

構文

ULValue::operator double()

float 演算子

「ULValue クラス」 247 ページを float にキャストします。

構文

ULValue::operator float()

int 演算子

「ULValue クラス」 247 ページを int にキャストします。

構文

ULValue::operator int()

long 演算子

「ULValue クラス」 247 ページを long にキャストします。

構文

ULValue::operator long()

short 演算子

「ULValue クラス」 247 ページを short にキャストします。

構文

ULValue::operator short()

ul_s_big 演算子

「ULValue クラス」 247 ページを signed bigint にキャストします。

構文

ULValue::operator ul_s_big()

ul_u_big 演算子

「ULValue クラス」 247 ページを unsigned bigint にキャストします。

構文

ULValue::operator ul_u_big()

unsigned char 演算子

「ULValue クラス」 247 ページを char にキャストします。

構文

ULValue::operator unsigned char()

unsigned int 演算子

「ULValue クラス」 247 ページを unsigned int にキャストします。

構文

ULValue::operator unsigned int()

unsigned long 演算子

「ULValue クラス」 247 ページを unsigned long にキャストします。

構文

ULValue::operator unsigned long()

unsigned short 演算子

「ULValue クラス」 247 ページを unsigned short にキャストします。

構文

ULValue::operator unsigned short()

operator= 関数

ULValue の = 演算子を無効にします。

構文

ULValue & **ULValue::operator=(**
  const ULValue & *other*
)

パラメータ

- **other**　「ULValue クラス」 247 ページに割り当てられる値。

~ULValue 関数

「ULValue クラス」 247 ページのデストラクタです。

構文

ULValue::~ULValue()

Embedded SQL API リファレンス

目次

この項では、Embedded SQL アプリケーションで Ultra Light 機能をサポートする関数について説明します。

使用できる SQL 文の概要については、「Embedded SQL を使用したアプリケーションの開発」 31 ページを参照してください。

この章で説明する関数のプロトタイプは、EXEC SQL INCLUDE SQLCA コマンドを使用してインクルードします。

db_fini 関数

Ultra Light ランタイム・ライブラリで使用したリソースを解放します。

構文

```
unsigned short db_fini(
SQLCA * sqlca
);
```

戻り値

- 処理中にエラーが発生した場合は 0。エラー情報は SQLCA に設定されます。
- エラーが発生しなかった場合は 0 以外。

備考

db_fini が呼び出された後に、他の Ultra Light ライブラリ呼び出しをしたり、Embedded SQL コマンドを実行したりしないでください。

使用する SQLCA ごとに 1 回ずつ db_fini を呼び出します。

参照

- 「db_init 関数」 268 ページ

db_init 関数

Ultra Light ランタイム・ライブラリを初期化します。

構文

```
unsigned short db_init(
SQLCA * sqlca
);
```

戻り値

- 処理中にエラーが発生した場合は 0 (たとえば永続ストアの初期化時)。エラー情報は SQLCA に設定されます。
- エラーが発生しなかった場合は 0 以外。Embedded SQL コマンドと関数の使用を開始できます。

備考

この関数は、他の Ultra Light ライブラリを呼び出す前、および Embedded SQL コマンドを実行する前に呼び出す必要があります。

通常は、この関数を一度だけ呼び出して、ヘッダ・ファイル *sqlca.h* に定義されているグローバル変数 sqlca のアドレスを渡してください。アプリケーションに複数の実行パスがある場合、複数の db_init を呼び出すことができます。ただし、それぞれに別個の sqlca ポインタが必要です。この別個の SQLCA ポインタには、ユーザが定義したものを使用するか、db_fini で解放されたグローバル SQLCA を使用します。

マルチスレッド・アプリケーションでは、各スレッドは、別個の SQLCA を獲得するために db_init を呼び出します。同じ SQLCA を使用する接続とトランザクションは、1 つのスレッドで続けて実行してください。

SQLCA を初期化すると、その前の ULEnable 関数呼び出しによる設定がすべてリセットされます。SQLCA を再初期化する場合は、アプリケーションで必要な ULEnable 関数をすべて発行する必要があります。

参照

- 「db_fini 関数」 267 ページ

db_start_database 関数

データベースがまだ起動していない場合に、データベースを起動します。

構文

```
unsigned int db_start_database(
SQLCA * sqlca,
char * parms
);
```

パラメータ

sqlca　SQLCA 構造体へのポインタ。

parms　**KEYWORD**=*value* の形式のパラメータ設定をセミコロンで区切ったリストからなる、NULL で終了された文字列。通常、ファイル名だけが必要です。次に例を示します。

```
"DBF=c:¥¥db¥¥mydatabase.db"
```

戻り値

- データベースがすでに実行している場合、または正しく起動した場合は、true。この場合、SQLCODE は 0 に設定されます。
- エラー情報は SQLCA に返されます。

備考

Embedded SQL と C++ コンポーネントを結合したアプリケーションを開発する場合に必要です。

参照

- 「Ultra Light 接続パラメータ」『Ultra Light データベース管理とリファレンス』
- 「SQLCA (SQL Communications Area) の初期化」 34 ページ

db_stop_database 関数

データベースを停止します。

構文

```
unsigned int db_stop_database(
SQLCA * sqlca,
char * parms
);
```

パラメータ

sqlca　SQLCA 構造体へのポインタ。

parms　**KEYWORD**=*value* の形式のパラメータ設定をセミコロンで区切ったリストからなる、NULL で終了された文字列。通常、データベース・ファイル名だけが必要です。次に例を示します。

```
"DBF=c:¥¥db¥¥mydatabase.db"
```

戻り値

- エラーが発生しなかった場合は TRUE。

備考

すべての接続が閉じるとデータベースも自動的に停止されるので、通常、この関数は必要ありません。この関数は、Embedded SQL と C++ コンポーネントを結合したアプリケーションを開発する場合に役立つことがあります。

この関数によって、既存の接続を持つデータベースは停止されません。

参照

- 「Ultra Light 接続パラメータ」『Ultra Light データベース管理とリファレンス』
- 「SQLCA (SQL Communications Area) の初期化」 34 ページ

ULChangeEncryptionKey 関数

Ultra Light データベースの暗号化キーを変更します。

構文

```
ul_bool ULChangeEncryptionKey(
SQLCA *sqlca,
ul_char *new_key
);
```

備考

この関数を呼び出すアプリケーションでは、データベースが同期されていること、または信頼できるバックアップ・コピーが作成されていることを、先に確認しておく必要があります。ULChangeEncryptionKey は実行を継続する必要のある操作であるため、信頼できるバックアップがあることは重要です。データベース暗号化キーを変更すると、まずデータベースのすべてのローは古いキーを使用して復号され、次に新しいキーを使用して再度暗号化されて、書き込まれます。**この操作は元に戻せません。**暗号化変更処理が完了しなかった場合、データベースは無効な状態のままになり、もう一度アクセスできなくなります。

参照

- 「データの暗号化」 54 ページ

ULCheckpoint 関数

チェックポイント操作を実行し、保留中になっているコミット済みトランザクションをデータベースにフラッシュします。ULCheckpoint を呼び出しても、現在のトランザクションすべてがコミットされるわけではありません。ULCheckpoint 関数は、パフォーマンスを向上させるために後回しされた自動トランザクション・チェックポイントとともに使用されます。詳細については、「単一のトランザクションまたはグループ化されたトランザクションのフラッシュ」『Ultra Light データベース管理とリファレンス』を参照してください。

構文

```
void ULCheckpoint(
SQLCA * sqlca
);
```

備考

ULCheckpoint 関数を使用すると、保留中のコミット済みトランザクションはすべてデータベースの記憶領域に書き込まれることが保証されます。

参照

- 「Ultra Light でのトランザクション処理」『Ultra Light データベース管理とリファレンス』

ULClearEncryptionKey 関数

暗号化キーをクリアします。

構文

ul_bool **ULClearEncryptionKey(**
ul_u_long * *creator*,
ul_u_long * *feature-num* **);**

パラメータ

creator　暗号化キーを保持している機能の作成者 ID へのポインタ。デフォルト値は NULL です。

feature-num　暗号化キーを保持する機能番号へのポインタ。feature-num が NULL の場合、アプリケーションでは Ultra Light のデフォルト値である 100 を使用します。

備考

Palm OS では、暗号化キーは Palm の「機能」として動的メモリに保存されます。機能には、作成者と機能番号のインデックスが付けられます。

参照

- 「ULRetrieveEncryptionKey 関数」 292 ページ
- 「ULSaveEncryptionKey 関数」 295 ページ

ULCountUploadRows 関数

同期するためにアップロードする必要があるローの数を数えます。

構文

```
ul_u_long ULCountUploadRows (
SQLCA * sqlca,
ul_char pub-list,
ul_u_long threshold
);
```

パラメータ

sqlca　SQLCA へのポインタ。

pub-list　チェック対象となるパブリケーションのカンマ区切りのリストを含む文字列。空の文字列 (UL_SYNC_ALL マクロ) は、「非同期」とマーク付けされたものを除くすべてのテーブルを表します。アスタリスクのみの文字列 (UL_SYNC_ALL_PUBS マクロ) は、いずれかのパブリケーションで参照されているすべてのテーブルを表します。pub-list 文字列に "*" を指定すると、どのパブリケーションからも参照されないテーブルがある場合、そのテーブルは対象に含まれません。

threshold　カウントするローの最大数を判断します。呼び出しの所要時間を制限します。

- threshold が 0 の場合、制限はありません (つまり、同期する必要のあるすべてのローをカウントします)。
- threshold が 1 の場合、同期の必要なローがあるかどうかを簡単に判別するために使用できます。

戻り値

- 指定されたパブリケーションのセットまたはデータベース全体のいずれかで、同期を必要とするローの数。

備考

この関数を使用して、ユーザに同期を要求します。

例

次の呼び出しでは、データベース全体をチェックして、同期させるローの数を確認します。

```
count = ULCountUploadRows( sqlca, UL_SYNC_ALL_PUBS, 0 );
```

次の呼び出しでは、最大 1000 のローに対してパブリケーション PUB1 と PUB2 がチェックされます。

```
count = ULCountUploadRows( sqlca, UL_TEXT("PUB1,PUB2"), 1000 );
```

次の呼び出しでは、パブリケーション PUB1 と PUB2 で同期させる必要のあるローがあるかどうかがチェックされます。

```
count = ULCountUploadRows( sqlca, UL_TEXT("PUB1,PUB2"), 1 );
```

参照

- 「UL_SYNC_ALL マクロ」 127 ページ
- 「UL_SYNC_ALL_PUBS マクロ」 128 ページ

ULDropDatabase 関数

Ultra Light データベース・ファイルと、関連するテンポラリ・ファイルまたはワーク・ファイルを削除します。

構文

```
ul_bool ULDropDatabase (
SQLCA * sqlca,
ul_char * store-parms
);
```

パラメータ

sqlca　SQLCA へのポインタ。

store-parms　**KEYWORD**=*value* の形式のパラメータ設定をセミコロンで区切ったリストからなる、NULL で終了された接続文字列。

戻り値

- **ul_true**　データベース・ファイルが正常に削除されました。
- **ul_false**　データベース・ファイルを削除できませんでした。詳細なエラー・メッセージが、SQLCA の sqlcode フィールドに設定されています。よくある失敗の理由は、ファイル名が正しくない、別のアプリケーションでファイルが開かれているためファイルへのアクセスが拒否されたなどです。

備考

この関数は、次の場合にだけ呼び出すようにしてください。

- 開かれているデータベース接続がない。
- db_init を呼び出す前か、db_fini を呼び出した後。

Palm OS では、次の場合にだけ呼び出すようにしてください。

- データベースに接続されていない。
- ULEnable を呼び出した後。

警告
この関数は、データベース・ファイルとその中のすべてのデータを削除します。この操作は元に戻せません。そのため、この関数を使用する場合は注意してください。

例

次の呼び出しは、Ultra Light データベース・ファイル *myfile.udb* を削除します。

```
if( ULDropDatabase(&sqlca, UL_TEXT("file_name=myfile.udb") ) ){
    // success
};
```

ULExecuteNextSQLPassthroughScript

次の SQL パススルー・スクリプトを実行します。

構文

```
bool ULExecuteNextSQLPassthroughScript(
SQLCA * sqlca
)
```

パラメータ

sqlca　SQLCA へのポインタ。

戻り値

- スクリプトの実行中にエラーが発生した場合は false。

参照

- 「Ultra Light global_database_id オプション」 『Ultra Light データベース管理とリファレンス』

ULExecuteSQLPassthroughScripts

使用可能なすべての SQL パススルー・スクリプトを実行します。

構文

bool**ULExecuteSQLPassthroughScripts(**
SQLCA * *sqlca*
)

パラメータ

sqlca　SQLCA へのポインタ。

戻り値

- スクリプトの実行中にエラーが発生した場合は false。

参照

- 「Ultra Light global_database_id オプション」 『Ultra Light データベース管理とリファレンス』

ULGetDatabaseID 関数

現在のデータベース ID を取得します。

構文

ul_u_long **ULGetDatabaseID(**
SQLCA * *sqlca*
)

パラメータ

sqlca　SQLCA へのポインタ。

戻り値

- SetDatabaseID の最後の呼び出しで設定された値。
- ID がこれまで設定されていない場合は UL_INVALID_DATABASE_ID。

備考

グローバル・オートインクリメント・カラムに使用される現在のデータベース ID です。

参照

- 「Ultra Light global_database_id オプション」 『Ultra Light データベース管理とリファレンス』

ULGetDatabaseProperty 関数

データベース・プロパティの値を取得します。

構文

```
void ULGetDatabaseProperty (
SQLCA * sqlca,
ul_database_property_id id,
char * dst,
size_t buffer-size,
ul_bool * null-indicator
);
```

パラメータ

sqlca　SQLCA へのポインタ。

id　データベース・プロパティの識別子。

dst　プロパティの値を格納する文字配列。

buffer-size　文字配列 *dst* のサイズ。

null-indicator　データベース・パラメータが NULL であることを示すインジケータ。

参照

- 「Ultra Light データベース・プロパティ」『Ultra Light データベース管理とリファレンス』

ULGetErrorParameter 関数

序数によりエラー・パラメータを取得します。

構文

```
size_t ULGetErrorParameter (
SQLCA * sqlca,
ul_u_long parm_num,
ul_char * buffer,
size_t size
);
```

パラメータ

sqlca　SQLCA へのポインタ。

parm_num　序数のパラメータ番号。

buffer　エラー・パラメータを格納するバッファへのポインタ。

size　バッファのサイズ (バイト数)。

戻り値

この関数は、指定されたバッファにコピーされる文字数を返します。

参照

- 「ULGetErrorParameterCount 関数」 282 ページ

ULGetErrorParameterCount 関数

エラー・パラメータの数を取得します。

構文

```
ul_u_long ULGetErrorParameterCount (
SQLCA * sqlca
);
```

パラメータ

sqlca　SQLCA へのポインタ。

戻り値

この関数は、エラー・パラメータの数を返します。結果が 0 の場合を除き、1 からこの戻り値までの値を使用して、ULGetErrorParameter を呼び出し、対応するエラー・パラメータ値を取得できます。

参照

- 「ULGetErrorParameter 関数」 281 ページ

ULGetLastDownloadTime 関数

指定したパブリケーションが最後にダウンロードされた時刻を取得します。

構文

```
ul_bool ULGetLastDownloadTime (
SQLCA * sqlca,
ul_string pub-name,
DECL_DATETIME * value
);
```

パラメータ

sqlca　SQLCA へのポインタ。

pub-name　最終ダウンロード時間を取得するパブリケーション名を含む文字列。

value　投入する **DECL_DATETIME** 構造体へのポインタ。たとえば、値 January 1, 1990 は、パブリケーションがまだ同期されていないことを示します。

戻り値

- **true**　*pub-name* によって指定されたパブリケーションの最終ダウンロード時間までに *value* が正常に投入されました。
- **false**　*pub-name* によって複数のパブリケーションが指定されたか、指定されたパブリケーションが未定義です。*value* の内容が有効ではありません。

例

次の呼び出しでは、パブリケーション UL_PUB_PUB1 がダウンロードされた日付と時刻とともに dt 構造体が投入されます。

```
DECL_DATETIME dt;
ret = ULGetLastDownloadTime( &sqlca, UL_TEXT("UL_PUB_PUB1"), &dt );
```

参照

- 「UL_SYNC_ALL マクロ」 127 ページ
- 「UL_SYNC_ALL_PUBS マクロ」 128 ページ

GetSQLPassthroughScriptCount

実行できる SQL パススルー・スクリプトの数を取得します。

構文

ul_u_long**ULExecuteNextSQLPassthroughScript(**
SQLCA * *sqlca*
)

パラメータ

sqlca　SQLCA へのポインタ。

戻り値

- 実行できる SQL パススルー・スクリプトの数を返します。

参照

- 「Ultra Light global_database_id オプション」『Ultra Light データベース管理とリファレンス』

ULGetSynchResult 関数

アプリケーション・プログラムで適切なアクションを実行できるようにするために、最新の同期の結果を格納する構造体です。

構文

```
ul_bool ULGetSynchResult(
ul_synch_result  * synch-result
);
```

パラメータ

synch-result　同期結果を保持する構造体。この構造体は、*ulglobal.h* で次のように定義されます。

```
typedef struct ul_synch_result {
    an_sql_code         sql_code;
    char                sql_error_string[];
    ul_stream_error     stream_error;
    ul_bool             upload_ok;
    ul_bool             ignored_rows;
    ul_auth_status      auth_status;
    ul_s_long           auth_value;
    SQLDATETIME         timestamp;
    ul_bool             partial_download_retained;
    ul_synch_status     status;
} ul_synch_result, * p_ul_synch_result;
```

個々のパラメータは次のとおりです。

- **sql_code**　最後の同期の SQL コード。SQL コードのリストについては、「SQL Anywhere のエラー・メッセージ (SQLSTATE 順)」『エラー・メッセージ』を参照してください。
- **sql_error_string**　**sql_code** フィールドでレポートされたエラーに関連するエラー・メッセージ・テキスト。
- **stream_error**　ul_stream_error 型の構造体。「Stream Error 同期パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。
- **upload_ok**　アップロードが成功した場合は true、それ以外の場合は false。
- **ignored_rows**　アップロードされたローが無視された場合は true、それ以外の場合は false。
- **auth_status**　同期認証ステータス。「Authentication Status 同期パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。
- **auth_value**　Mobile Link サーバが **auth_status** の結果を判断するために使用する値。「Authentication Value 同期パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。
- **timestamp**　最後の同期の時刻と日付。
- **partial_download_retained**　部分的なダウンロードが保持されているかどうかを示すフラグ。

- **status**　observer 関数によって使用されるステータス情報。「Observer 同期パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。

戻り値

- 処理が成功した場合は true。
- 処理が失敗した場合は false。

備考

アプリケーションでは、ul_synch_result オブジェクトを割り当ててから ULGetSynchResult に渡してください。この関数は、ul_synch_result に最後の同期の結果を入れます。これらの結果は、データベースに永続的に格納されます。

Palm OS で HotSync を使用してアプリケーションを同期する場合、同期はアプリケーションの外部で行われるため、この関数を使用します。接続で設定された **SQLCODE** 値は、接続操作の結果そのものを表します。同期ステータスと結果は、HotSync ログにだけ書き込まれます。詳細な同期結果情報を取得するには、データベースに接続しているときに ULGetSynchResult を呼び出します。

例

次のコードは、前回の同期が正常に終了したかどうかをチェックします。

```
ul_synch_result synch_result;
memset( &synch_result, 0, sizeof( ul_synch_result ) );
db_init( &sqlca );
EXEC SQL CONNECT "dba" IDENTIFIED BY "sql";
if( !ULGetSynchResult( &sqlca, &synch_result ) ) {
    prMsg( "ULGetSynchResult failed" );
}
```

ULGlobalAutoincUsage 関数

グローバル・オートインクリメントのデフォルト値を持つすべてのカラムで、デフォルト値が使用されている比率 (%) を取得します。

構文

```
ul_u_short ULGlobalAutoincUsage(
SQLCA * sqlca
);
```

戻り値

- 0 ～ 100 の範囲の short 値。

備考

このデフォルト値を使用するカラムがデータベース内に複数含まれている場合は、すべてのカラムに対してこの値が計算され、最大値が返されます。たとえば、戻り値 99 は、少なくとも 1 つのカラムではデフォルト値が残されているが、きわめて少ないことを示します。

参照

- 「ULSetDatabaseID 関数」 296 ページ
- 「Ultra Light global_database_id オプション」 『Ultra Light データベース管理とリファレンス』

ULGrantConnectTo 関数

指定されたパスワードを持つ新しいまたは既存のユーザ ID に、Ultra Light データベースへのアクセスを許可します。

構文

```
void ULGrantConnectTo(
SQLCA * sqlca,
ul_char * userid,
ul_char * password
);
```

パラメータ

sqlca　SQLCA へのポインタ。

userid　ユーザ ID を保持する文字配列。最大長は 16 文字です。

password　そのユーザ ID のパスワードを保持する文字配列。最大長は 16 文字です。

備考

既存のユーザ ID を指定した場合は、この関数を使用してそのユーザのパスワードを更新します。

参照

- 「ユーザの認証」 52 ページ
- 「ULRevokeConnectFrom 関数」 293 ページ

ULInitSynchInfo 関数

同期情報の構造体を初期化します。

構文

```
void ULInitSynchInfo(
ul_synch_info *  synch_info
);
```

パラメータ

synch_info　同期構造体。この構造体のメンバの詳細については、「Ultra Light の同期パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。

ULIsSynchronizeMessage 関数

メッセージが ActiveSync に対する Mobile Link プロバイダからの同期メッセージであるかどうかを確認し、そのメッセージを処理するコードを呼び出すことができます。同期メッセージの処理が完了したときに、ULSignalSyncIsComplete 関数を呼び出す必要があります。

構文

ul_bool **ULIsSynchronizeMessage(**
ul_u_long *uMsg*
);

備考

この関数の呼び出しを、使用しているアプリケーションの WindowProc 関数にインクルードしてください。

ActiveSync を使用する Windows Mobile に適用されます。

例

以下のコードは、ULIsSynchronizeMessage を使用した同期メッセージの処理方法の箇所を抜粋したものです。

```
LRESULT CALLBACK WindowProc( HWND hwnd,
        UINT uMsg,
        WPARAM wParam,
        LPARAM lParam )
{
  if( ULIsSynchronizeMessage( uMsg ) ) {
    // execute synchronization code
    if( wParam == 1 ) DestroyWindow( hWnd );
    return 0;
  }

  switch( uMsg ) {

  // code to handle other windows messages

  default:
    return DefWindowProc( hwnd, uMsg, wParam, lParam );
  }
  return 0;
}
```

参照

- 「アプリケーションへの ActiveSync 同期の追加」 92 ページ
- 「Windows Mobile の ActiveSync」 『Ultra Light データベース管理とリファレンス』
- 「ULSignalSyncIsComplete 関数」 300 ページ

ULResetLastDownloadTime 関数

アプリケーションが以前にダウンロードされたデータを再同期するように、パブリケーションの最終ダウンロード時間をリセットします。

構文

```
void ULResetLastDownloadTime(
SQLCA * sqlca,
ul_string pub-list
);
```

パラメータ

sqlca　SQLCA へのポインタ。

pub-list　リセットするパブリケーションのカンマ区切りのリストを含む文字列。空の文字列は、「非同期」とマーク付けされたものを除くすべてのテーブルを表します。アスタリスクのみの文字列 ("*") は、すべてのパブリケーションを表します。pub-list 文字列に "*" を指定すると、どのパブリケーションからも参照されないテーブルがある場合、そのテーブルは対象に含まれません。

例

次の関数呼び出しは、すべてのテーブルの最終ダウンロード時間をリセットします。

```
ULResetLastDownloadTime( &sqlca, UL_TEXT("") );
```

参照

- 「ULGetLastDownloadTime 関数」 283 ページ
- 「タイムスタンプベースのダウンロード」『Mobile Link - サーバ管理』
- 「UL_SYNC_ALL マクロ」 127 ページ
- 「UL_SYNC_ALL_PUBS マクロ」 128 ページ

ULRetrieveEncryptionKey 関数

メモリから暗号化キーを取り出します。

構文

```
ul_bool ULRetrieveEncryptionKey(
ul_char * key,
ul_u_short len,
ul_u_long * creator,
ul_u_long * feature-num
);
```

パラメータ

key　取り出された暗号化キーを保管するバッファへのポインタ。

len　末尾の NULL 文字とともに暗号化キーを保管するバッファの長さ。

creator　暗号化キーを保持している機能の作成者 ID へのポインタ。デフォルト値は NULL です。

feature-num　暗号化キーを保持する機能番号へのポインタ。feature-num が NULL の場合、アプリケーションでは Ultra Light のデフォルト値である 100 を使用します。

戻り値

- 処理が成功した場合は true。
- 処理が失敗した場合は false。機能が見つからなかった場合や、指定したバッファの長さが、キーと末尾の NULL 文字の合計の長さよりも短かかった場合に、この値が返されます。

備考

Palm OS では、暗号化キーは Palm の「機能」として動的メモリに保存されます。機能には、作成者と機能番号のインデックスが付けられます。

Palm OS に適用されます。

参照

- 「ULClearEncryptionKey 関数」 273 ページ
- 「ULSaveEncryptionKey 関数」 295 ページ

ULRevokeConnectFrom 関数

Ultra Light データベースからユーザ ID のアクセス権を取り消します。

構文

```
void ULRevokeConnectFrom(
SQLCA * sqlca,
ul_char * userid
);
```

パラメータ

sqlca　SQLCA へのポインタ。

userid　データベース・アクセスから除外するユーザ ID を保持する文字配列。最大長は 16 文字です。

参照

- 「ユーザの認証」 52 ページ
- 「ULGrantConnectTo 関数」 288 ページ

ULRollbackPartialDownload 関数

失敗した同期からの変更をロールバックします。

構文

```
void ULRollbackPartialDownload(
SQLCA * sqlca
);
```

パラメータ

- **sqlca**　SQLCA へのポインタ。C++ API では、Sqlca.GetCA メソッドを使用します。

備考

同期のダウンロード時に通信エラーが発生した場合、Ultra Light ではダウンロードした変更を適用できるため、アプリケーションでは同期が中断した時点から同期を再開することができます。ダウンロードした変更が不要な場合 (ダウンロードが中断した時点での再開を望まない場合)、ULRollbackPartialDownload を使用することで、失敗したダウンロード・トランザクションをロールバックします。

参照

- 「GetCA 関数」 149 ページ
- 「失敗したダウンロードの再開」 『Mobile Link - サーバ管理』
- 「Keep Partial Download 同期パラメータ」 『Ultra Light データベース管理とリファレンス』
- 「Partial Download Retained 同期パラメータ」 『Ultra Light データベース管理とリファレンス』
- 「Resume Partial Download 同期パラメータ」 『Ultra Light データベース管理とリファレンス』

ULSaveEncryptionKey 関数

Palm の動的メモリに暗号化キーを保存します。

構文

```
ul_bool ULSaveEncryptionKey(
ul_char * key,
ul_u_long * creator,
ul_u_long * feature-num
);
```

パラメータ

key　暗号化キーへのポインタ。

creator　暗号化キーを保持している機能の作成者 ID へのポインタ。デフォルト値は NULL です。

feature-num　暗号化キーを保持する機能番号へのポインタ。feature-num が NULL の場合、アプリケーションでは Ultra Light のデフォルト値である 100 を使用します。

戻り値

- 処理が成功した場合は true。
- 処理が失敗した場合は false。機能が見つからなかった場合や、指定したバッファの長さが、キーと末尾の NULL 文字の合計の長さよりも短かかった場合に、この値が返されます。

備考

Palm OS では、暗号化キーは Palm の「機能」として動的メモリに保存されます。機能には、作成者と機能番号のインデックスが付けられます。これらの機能はバックアップされません。デバイスがリセットされるとクリアされます。

Palm OS アプリケーションに適用されます。

参照

- 「ULClearEncryptionKey 関数」 273 ページ
- 「ULRetrieveEncryptionKey 関数」 292 ページ

ULSetDatabaseID 関数

データベースの ID 番号を設定します。

構文

```
void ULSetDatabaseID(
SQLCA * sqlca,
ul_u_long id
);
```

パラメータ

sqlca　SQLCA へのポインタ。

id　レプリケーションまたは同期を設定する時に、特定のデータベースをユニークに識別する正の整数。

参照

- 「Ultra Light global_database_id オプション」 『Ultra Light データベース管理とリファレンス』
- 「ULGlobalAutoincUsage 関数」 287 ページ

ULSetDatabaseOptionString 関数

文字列値からデータベース・オプションを設定します。

構文

```
void ULSetDatabaseOptionString (
SQLCA * sqlca,
ul_database_option_id id,
ul_char * value
);
```

パラメータ

sqlca　SQLCA へのポインタ。

id　設定するデータベース・オプションの識別子。

value　データベース・オプションの値。

ULSetDatabaseOptionULong 関数

数値のデータベース・オプションを設定します。

構文

```
void ULSetDatabaseOptionULong(
SQLCA * sqlca,
ul_database_option_id id,
ul_u_long * value
);
```

パラメータ

sqlca　SQLCA へのポインタ。

id　設定するデータベース・オプションの識別子。

value　データベース・オプションの値。

ULSetSynchInfo 関数

HotSync で使用する同期パラメータを格納します。

構文

```
ul_bool ULSetSynchInfo(
SQLCA * sqlca,
PROFILE sync-profile-name
ul_synch_info *  synch_info
);
```

パラメータ

sqlca　SQLCA へのポインタ。

sync-profile-name　同期情報を格納する同期プロファイルの名前。HotSync の一部として、この名前で後から参照されます。「Palm OS 用 Ultra Light HotSync コンジットのインストール・ユーティリティ (ulcond11)」『Ultra Light データベース管理とリファレンス』を参照してください。

synch_info　同期構造体。この構造体のメンバの詳細については、「ul_synch_info_a 構造体」 135 ページを参照してください。

備考

通常、ULSetSynchInfo は、db_fini でアプリケーションを閉じる直前に呼び出します。

HotSync を使用する Palm OS アプリケーションに適用されます。

参照

- 「db_fini 関数」 267 ページ

ULSignalSyncIsComplete 関数

この関数は、同期メッセージの処理が完了したことを示すために呼び出されます。

構文

void **ULSignalSyncIsComplete();**

備考

ActiveSync プロバイダで登録したアプリケーションは、同期メッセージの処理が完了したときに、WNDPROC でこのメソッドを呼び出す必要があります。

参照

- 「ULIsSynchronizeMessage 関数」 290 ページ

ULSynchronize 関数

Ultra Light アプリケーションで同期を開始します。

構文

```
void ULSynchronize(
SQLCA * sqlca,
ul_synch_info * synch_info
);
```

パラメータ

sqlca　SQLCA へのポインタ。

synch_info　同期構造体。同期の定義は、同期パラメータのセットを介して制御されます。これらのパラメータの詳細については、「ul_synch_info_a 構造体」 135 ページを参照してください。

備考

TCP/IP または HTTP 同期では、ULSynchronize 関数が同期を開始します。同期中のエラーで handle_error スクリプトによって処理されないものは、SQL エラーとしてレポートされます。アプリケーション・プログラムでは、この関数の SQLCODE 戻り値をテストする必要があります。

Ultra Light ODBC API リファレンス

目次

この項では、Ultra Light でサポートされる ODBC インタフェースのコンポーネントについて説明します。

これは包括的な ODBC リファレンスではありません。ODBC のメイン・リファレンスである Microsoft の『ODBC Programmer's Reference』を補足するためのクイック・リファレンスです。

SQLAllocHandle 関数

Ultra Light ODBC の場合、ハンドルを割り当てます。

構文

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLAllocHandle(**
SQLSMALLINT *HandleType*,
SQLHANDLE *InputHandle*,
SQLHANDLE * *OutputHandle* **)**;

パラメータ

- **HandleType**　割り当てるハンドルのタイプ。Ultra Light では次のハンドル・タイプがサポートされます。
  - SQL_HANDLE_ENV (環境ハンドル)
  - SQL_HANDLE_DBC (接続ハンドル)
  - SQL_HANDLE_STMT (ステートメント・ハンドル)
- **InputHandle**　コンテキストに新しいハンドルが割り当てられるハンドル。接続ハンドルの場合、これは環境ハンドルです。ステートメント・ハンドルの場合、これは接続ハンドルです。
- **OutputHandle**　新しいハンドルが返されるバッファへのポインタ。

備考

ODBC では、ハンドルを使用してデータベース操作のコンテキストを提供しています。他のインタフェースにおける SQLCA (SQL Communications Area) と同様、環境ハンドルは、データ・ソースとの通信のコンテキストを提供します。接続ハンドルは、すべてのデータベース操作のコンテキストを提供します。ステートメント・ハンドルは、結果セットとデータ修正を管理します。記述子ハンドルは、結果セットのデータ型の処理を管理します。

SQLBindCol 関数

Ultra Light ODBC の場合、結果セットのカラムを、アプリケーション・データ・バッファにバインドします。

構文

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLBindCol (
SQLHSTMT StatementHandle,
SQLUSMALLINT ColumnNumber,
SQLSMALLINT TargetType,
SQLPOINTER TargetValue,
SQLLEN BufferLength,
SQLLEN * StrLen_or_Ind );
```

パラメータ

- **StatementHandle**　結果セットを返す文のハンドル。
- **ColumnNumber**　アプリケーション・データ・バッファにバインドする結果セット内のカラムの番号。
- **TargetType**　*TargetValue* ポインタのデータ型の識別子。
- **TargetValue**　カラムにバインドするデータ・バッファへのポインタ。
- **BufferLength**　*TargetValue* バッファのバイト単位の長さ。
- **StrLen_or_Ind**　カラムにバインドする長さバッファまたはインジケータ・バッファへのポインタ。文字列の場合、長さバッファには、返された実際の文字列の長さが格納されます。この値は、カラムに設定できる長さよりも短い場合があります。

備考

アプリケーションとデータベースの間で情報を交換するために、ODBC では、アプリケーション内のバッファを、カラムなど、データベース・オブジェクトにバインドします。指定したカラムの値を設定する場所として、アプリケーション内のバッファを識別するクエリを実行する場合は、SQLBindCol を使用します。

SQLBindParameter 関数

Ultra Light ODBC の場合、バッファ・パラメータを SQL 文の中のパラメータ・マーカにバインドします。

構文

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLBindParameter (**
SQLHSTMT *StatementHandle*,
SQLUSMALLINT *ParameterNumber*,
SQLSMALLINT *ParamType*,
SQLSMALLINT *CType*,
SQLSMALLINT *SqlType*,
SQLULEN *ColDef*,
SQLSMALLINT *Scale*,
SQLPOINTER *rgbValue*,
SQLLEN *cbValueMax*,
SQLLEN * *StrLen_or_Ind* **)**;

パラメータ

- **StatementHandle**　ステートメント・ハンドル。
- **ParameterNumber**　1 から順番にカウントされる、文の中のパラメータ・マーカの番号。
- **ParamType**　パラメータ・タイプ。以下のいずれかを表します。
  - SQL_PARAM_INPUT
  - SQL_PARAM_INPUT_OUTPUT
  - SQL_PARAM_OUTPUT
- **CType**　C データ型のパラメータ。
- **SQLType**　SQL データ型のパラメータ。
- **ColDef**　カラムのサイズまたはパラメータ・マーカの式。
- **Scale**　カラムの桁数またはパラメータ・マーカの式。
- **rgbValue**　パラメータのデータのバッファへのポインタ。
- **cbValueMax**　*rgbValue* バッファの長さ。
- **StrLen_or_Ind**　パラメータの長さのバッファへのポインタ。

備考

アプリケーションとデータベースの間で情報を交換するために、ODBC では、アプリケーション内のバッファを、カラムなど、データベース・オブジェクトにバインドします。クエリで指定したパラメータの値を取得または設定する場所として、アプリケーション内のバッファを識別する文を実行する場合は、SQLBindParameter を使用します。

SQLConnect 関数

Ultra Light ODBC の場合、データベースに接続します。

構文

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLConnect (
SQLHDBC ConnectionHandle,
SQLTCHAR * ServerName,
SQLSMALLINT NameLength1,
SQLTCHAR * UserName,
SQLSMALLINT NameLength2,
SQLTCHAR * Authentication,
SQLSMALLINT NameLength3 );
```

パラメータ

- **ConnectionHandle**　接続ハンドル。
- **ServerName**　アプリケーションが接続するデータベースを定義する接続文字列。Ultra Light ODBC では、ODBC データ・ソースは使用しません。代わりに、データベース接続パラメータ、さらにその他のオプションのパラメータを含む接続文字列を指定します。

  次は、ServerName パラメータの使用例です。

  ```
  (SQLTCHAR*)UL_TEXT(
     "dbf=customer.udb"
  )
  ```

  接続パラメータの完全なリストについては、「Ultra Light 接続パラメータ」 『Ultra Light データベース管理とリファレンス』を参照してください。
- **NameLength1**　**ServerName* の長さ。
- **UserName**　接続時に使用するユーザ ID。ユーザ ID は、ServerName パラメータに提供する接続文字列に指定することもできます。
- **NameLength2**　**UserName* の長さ。
- **Authentication**　接続時に使用するパスワード。パスワードは、ServerName パラメータに提供する接続文字列に指定することもできます。
- **NameLength3**　**Authentication* の長さ。

備考

データベースに接続します。Ultra Light 接続パラメータの詳細については、「Ultra Light 接続パラメータ」 『Ultra Light データベース管理とリファレンス』を参照してください。

SQLDescribeCol 関数

Ultra Light ODBC の場合、結果セット内のカラムの結果記述子を返します。

結果記述子には、カラム名、カラムのサイズ、データ型、桁数、NULL 入力が可能かどうかが含まれています。

構文

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLDescribeCol (
   SQLHSTMT StatementHandle,
   SQLUSMALLINT ColumnNumber,
   SQLTCHAR * ColumnName,
   SQLSMALLINT BufferLength,
   SQLSMALLINT * NameLength,
   SQLSMALLINT * DataType,
   SQLULEN * ColumnSize,
   SQLSMALLINT * DecimalDigits,
   SQLSMALLINT * Nullable );
```

パラメータ

- **StatementHandle**　ステートメント・ハンドル。
- **ColumnNumber**　結果セットの、1 から始まるカラム番号。
- **ColumnName**　カラム名が返されるバッファへのポインタ。
- **BufferLength**　*ColumnName の文字単位の長さ。
- **NameLength**　*ColumnName に返される有効なバイト (NULL 終了バイトを除く) の総数が返される、バッファへのポインタ。
- **DataType**　カラムの SQL データ型が返されるバッファへのポインタ。
- **ColumnSize**　データ・ソースのカラムのサイズが返されるバッファへのポインタ。
- **DecimalDigits**　データ・ソースのカラムの桁数が返されるバッファへのポインタ。
- **Nullable**　カラムに NULL 値を入力できるかどうかを示す値が返されるバッファへのポインタ。

SQLDisconnect 関数

Ultra Light ODBC の場合、データベースからアプリケーションを切断します。

構文

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLDisconnect (**
SQLHDBC *ConnectionHandle* **)**;

パラメータ

- **ConnectionHandle**　閉じる接続のハンドル。

備考

SQLDisconnect がいったん呼び出されると、新しい接続を開かないかぎり、データベースに対して操作は実行されません。

参照

- 「SQLConnect 関数」 308 ページ

SQLEndTran 関数

Ultra Light ODBC の場合、トランザクションをコミットまたはロールバックします。

構文

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLEndTran (**
SQLSMALLINT *HandleType*,
SQLHANDLE *Handle*,
SQLSMALLINT *CompletionType* **)**;

パラメータ

- **HandleType**　割り当てるハンドルのタイプ。Ultra Light では次のハンドル・タイプがサポートされます。
  - SQL_HANDLE_ENV
  - SQL_HANDLE_DBC
  - SQL_HANDLE_STMT
- **Handle**　トランザクションのスコープを示す接続ハンドル。
- **CompletionType**　以下の 2 つの値のいずれかを表します。
  - SQL_COMMIT
  - SQL_ROLLBACK

SQLExecDirect 関数

Ultra Light ODBC の場合、SQL 文を実行します。

構文

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLExecDirect (**
SQLHSTMT *StatementHandle*,
SQLTCHAR * *StatementText*,
SQLINTEGER *TextLength* **)**;

パラメータ

- **StatementHandle** ステートメント・ハンドル。
- **StatementText** SQL 文のテキスト。
- **TextLength** **StatementText* の長さ。

備考

SQLExecute とは違って、SQLExecDirect の場合は、実行前に文を準備する必要はありません。

SQLExecDirect では文が繰り返し実行されるため、SQLExecute に比べてパフォーマンスは低下します。

参照

- 「SQLExecute 関数」 313 ページ

SQLExecute 関数

Ultra Light ODBC の場合、準備された SQL 文を実行します。

構文

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLExecute (**
SQLHSTMT *StatementHandle* **)**;

パラメータ

- **StatementHandle**　実行する文のハンドル。

備考

SQLPrepare を使用して文を準備してから、実行してください。文にパラメータ・マーカがある場合は、SQLBindParameter を使用して変数にバインドしてから実行します。

SQLExecDirect を使用すると、文を最初に準備せずに実行できます。SQLExecDirect では文が繰り返し実行されるため、SQLExecute に比べてパフォーマンスは低下します。

参照

- 「SQLBindParameter 関数」 307 ページ
- 「SQLPrepare 関数」 322 ページ
- 「SQLExecDirect 関数」 312 ページ

SQLFetch 関数

Ultra Light の場合、結果セットの次のローをフェッチし、バインドされているすべてのカラムのデータを返します。

構文

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLFetch (**
SQLHSTMT *StatementHandle* **)**;

パラメータ

- **StatementHandle**　ステートメント・ハンドル。

備考

SQLBindCol を使用して結果セット内のカラムをバッファにバインドしてから、ローをフェッチしてください。結果セット内の次のロー以外のローをフェッチするには、SQLFetchScroll を使用します。

参照

- 「SQLFetchScroll 関数」 315 ページ
- 「SQLBindCol 関数」 306 ページ

SQLFetchScroll 関数

Ultra Light ODBC の場合、結果セットの指定されたローをフェッチし、バインドされているすべてのカラムのデータを返します。

構文

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLFetchScroll (**
SQLHSTMT *StatementHandle*,
SQLSMALLINT *FetchOrientation*,
SQLLEN *FetchOffset* **)**;

パラメータ

- **StatementHandle**　ステートメント・ハンドル。
- **FetchOrientation**　フェッチのタイプ。
- **FetchOffset**　フェッチするローの番号。解釈の仕方は、*FetchOrientation* の値に依存します。

備考

SQLBindCol を使用して結果セット内のカラムをバッファにバインドしてから、ローをフェッチしてください。SQLFetchScroll は、より簡単な SQLFetch が適切でない場合に使用します。

参照

- 「SQLFetch 関数」 314 ページ
- 「SQLBindCol 関数」 306 ページ

SQLFreeHandle 関数

Ultra Light ODBC の場合、割り当てられているハンドルを解放します。

構文

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLFreeHandle (**
SQLSMALLINT *HandleType*,
SQLHANDLE *Handle* **)**;

パラメータ

- **HandleType**　割り当てるハンドルのタイプ。Ultra Light では次のハンドル・タイプがサポートされます。
  - SQL_HANDLE_ENV
  - SQL_HANDLE_DBC
  - SQL_HANDLE_STMT
- **Handle**　解放するハンドル。

備考

SQLFreeHandle は、SQLAllocHandle を使用して割り当てたすべてのハンドルに対して、そのハンドルが不要になった場合に呼び出してください。

参照

- 「SQLAllocHandle 関数」 305 ページ

SQLGetCursorName 関数

Ultra Light ODBC の場合、指定の文のカーソルに関連付けられている名前を返します。

構文

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLGetCursorName (
   SQLHSTMT StatementHandle,
   SQLTCHAR * CursorName,
   SQLSMALLINT BufferLength,
   SQLSMALLINT * NameLength );
```

パラメータ

- **StatementHandle**　ステートメント・ハンドル。
- **CursorName**　*StatementHandle* に関連付けられているカーソルの名前が返されるバッファへのポインタ。
- **BufferLength**　**CursorName* の長さ。
- **NameLength**　**CursorName* に返される有効なバイト (NULL 終了文字を除く) の総数が返される、メモリへのポインタ。

参照

- 「SQLSetCursorName 関数」 325 ページ

SQLGetData 関数

Ultra Light ODBC の場合、結果セット内の 1 つのカラムのデータを取得します。

構文

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLGetData (
SQLHSTMT StatementHandle,
SQLUSMALLINT ColumnNumber,
SQLSMALLINT TargetType,
SQLPOINTER TargetValue,
SQLLEN BufferLength,
SQLLEN * StrLen_or_Ind );
```

パラメータ

- **StatementHandle**　ステートメント・ハンドル。
- **ColumnNumber**　バインドする結果セット内のカラムの番号。
- **TargetType**　出力ハンドル。
- **TargetValue**　カラムにバインドするデータ・バッファへのポインタ。
- **BufferLength**　*TargetValue* バッファのバイト単位の長さ。
- **StrLen_or_Ind**　カラムにバインドする長さバッファまたはインジケータ・バッファへのポインタ。

備考

SQLGetData は、通常、一部が可変長なデータを取得するときに使用します。

SQLGetDiagRec 関数

Ultra Light ODBC の場合、診断ステータス・レコードの複数のフィールドの現在値を返します。

構文

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLGetDiagRec (
SQLSMALLINT HandleType,
SQLHANDLE Handle,
SQLSMALLINT RecNumber,
SQLTCHAR * Sqlstate,
SQLINTEGER * NativeError,
SQLTCHAR * MessageText,
SQLSMALLINT BufferLength,
SQLSMALLINT * TextLength );
```

パラメータ

- **HandleType**　割り当てるハンドルのタイプ。Ultra Light では次のハンドル・タイプがサポートされます。
  - SQL_HANDLE_ENV
  - SQL_HANDLE_DBC
  - SQL_HANDLE_STMT
- **Handle**　入力ハンドル。
- **RecNumber**　出力ハンドル。
- **Sqlstate**　エラーの ANSI/ISO SQLSTATE 値。詳細については、「SQL Anywhere のエラー・メッセージ (SQLSTATE 順)」『エラー・メッセージ』を参照してください。
- **NativeError**　エラーの SQLCODE 値。詳細については、「SQL Anywhere のエラー・メッセージ (SQLCODE 順)」『エラー・メッセージ』を参照してください。
- **MessageText**　エラー・メッセージまたはステータス・メッセージのテキスト。
- **BufferLength**　*MessageText* バッファのバイト単位の長さ。
- **TextLength**　*MessageText* に返される有効なバイト (NULL 終了バイトを除く) の総数が返される、バッファへのポインタ。

SQLGetInfo 関数

Ultra Light ODBC の場合、現在の ODBC ドライバとデータ・ソースに関する一般的な情報を返します。

構文

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLGetInfo (
SQLHDBC ConnectionHandle,
SQLUSMALLINT InfoType,
SQLPOINTER * InfoValue,
SQLSMALLINT BufferLength,
SQLSMALLINT ODBCFAR * StringLength );
```

パラメータ

- **ConnectionHandle**　接続ハンドル。
- **InfoType**　返される情報のタイプ。SQL_DBMS_VER だけがサポートされます。返される情報は、ソフトウェアの現在のリリースを識別する文字列です。
- **InfoValue**　情報が返されるバッファへのポインタ。
- **BufferLength**　**InfoValue* バッファのバイト単位の長さ。
- **StringLength**　**InfoValue* に返される有効なバイト (文字データの NULL 終了文字を除く) の総数が返される、バッファへのポインタ。

SQLNumResultCols 関数

Ultra Light ODBC の場合、結果セットのカラム数を返します。

構文

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLNumResultCols (**
SQLHSTMT *StatementHandle*,
SQLSMALLINT * *ColumnCount* **)**;

パラメータ

- **StatementHandle**　ステートメント・ハンドル。
- **ColumnCount**　結果セットのカラムの総数が返されるバッファへのポインタ。

SQLPrepare 関数

Ultra Light ODBC の場合、実行する SQL 文を準備します。

構文

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLPrepare (**
SQLHSTMT *StatementHandle*,
SQLTCHAR * *StatementText*,
SQLINTEGER *TextLength* **)**;

パラメータ

- **StatementHandle**　ステートメント・ハンドル。
- **StatementText**　SQL 文のテキストを保持しているバッファへのポインタ。
- **TextLength**　**StatementText* の長さ。

参照

- 「SQLExecute 関数」 313 ページ

SQLRowCount 関数

Ultra Light ODBC の場合、挿入、更新、または削除操作の影響を受けるローの数を返します。

構文

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLRowCount (**
SQLHSTMT *StatementHandle*,
SQLLEN * *RowCount* **)**;

パラメータ

- **StatementHandle**　ステートメント・ハンドル。
- **RowCount**　ローの数が返されるバッファへのポインタ。

SQLSetConnectionName 関数

Ultra Light ODBC の場合、サスペンドと復帰の操作の接続名を設定します。この関数は Ultra Light 固有の関数であり、ODBC 規格には含まれません。

構文

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLSetConnectionName (
SQLHSTMT StatementHandle,
SQLTCHAR * ConnectionName,
SQLSMALLINT NameLength );
```

パラメータ

- **StatementHandle** ステートメント・ハンドル。
- **ConnectionName** 接続名を保持しているバッファへのポインタ。
- **NameLength** **ConnectionName* の長さ。

備考

SQLSetConnectionName は、SQLSetSuspend と一緒に、サスペンドと復帰の操作に使用する接続名を提供するために使用します。接続名を設定してから、接続を開き、アプリケーションの状態を復帰します。

参照

- 「Ultra Light Palm アプリケーションのステータスの管理 (旧式)」 74 ページ
- 「SQLSetSuspend 関数 (旧式)」 326 ページ

SQLSetCursorName 関数

Ultra Light ODBC の場合、SQL 文に関連付けられているカーソルの名前を設定します。

構文

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLSetCursorName (**
SQLHSTMT *StatementHandle*,
SQLTCHAR * *CursorName*,
SQLSMALLINT *NameLength* **)**;

パラメータ

- **StatementHandle**　ステートメント・ハンドル。
- **CursorName**　カーソル名を保持しているバッファへのポインタ。
- **NameLength**　**CursorName* の長さ。

参照

- 「SQLGetCursorName 関数」 317 ページ

SQLSetSuspend 関数 (旧式)

Ultra Light ODBC の場合、アプリケーションを閉じるときに、開いているカーソルの状態を保存するかどうかを示します。この関数は Ultra Light 固有の関数であり、ODBC 規格には含まれません。

構文

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLSetSuspend (**
SQLSMALLINT *HandleType*,
SQLHSTMT *StatementHandle*,
SQLSMALLINT *TrueFalse* **)**;

パラメータ

- **HandleType**　割り当てるハンドルのタイプ。Ultra Light では次のハンドル・タイプがサポートされます。
  - SQL_HANDLE_ENV
  - SQL_HANDLE_DBC
  - SQL_HANDLE_STMT
- **StatementHandle**　ステートメント・ハンドル。
- **TrueFalse**　出力ハンドル。

参照

- 「Ultra Light Palm アプリケーションのステータスの管理 (旧式)」 74 ページ

SQLSynchronize 関数

Ultra Light ODBC の場合、Mobile Link 同期を使用してデータベース内のデータを同期します。この関数は Ultra Light 固有の関数であり、ODBC 規格には含まれません。

構文

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLSynchronize (
SQLHDBC ConnectionHandle,
ul_synch_info * SynchInfo );
```

パラメータ

- **ConnectionHandle**　ハンドル。
- **SynchInfo**　同期情報を保持している構造体。「Ultra Light 同期パラメータとネットワーク・プロトコル・オプション」『Ultra Light データベース管理とリファレンス』を参照してください。

備考

SQLSynchronize は ODBC の拡張機能です。Mobile Link 同期操作を開始します。

参照

- 「Ultra Light 同期パラメータとネットワーク・プロトコル・オプション」『Ultra Light データベース管理とリファレンス』
- Mobile Link - サーバ管理

チュートリアル：C++ API を使用したアプリケーションの構築

目次

このチュートリアルでは、Ultra Light C++ アプリケーションを開発するプロセスを説明します。アプリケーションは Windows デスクトップ・オペレーティング・システム用に開発され、コマンド・プロンプトで実行されます。

このチュートリアルは、Microsoft Visual C++ を使用した開発を基にしていますが、任意の C++ 開発環境を使用できます。

このチュートリアルは、コードをコピーして貼り付ける場合、約 30 分で終了します。この章の最後には、このチュートリアルで説明しているプログラムの完全なソース・コードを掲載しています。

前提知識と経験

このチュートリアルは、次のことを前提にしています。

- C++ プログラミング言語に精通している。
- C++ コンパイラがマシンにインストールされている。
- **データベース作成ウィザード**を使用して Ultra Light データベースを作成する方法を知っている。

  詳細については、「データベース作成ウィザードを使用したデータベースの作成」『Ultra Light データベース管理とリファレンス』を参照してください。

このチュートリアルの目的は、Ultra Light C++ アプリケーションの開発プロセスについて、知識を得ることです。

レッスン 1 ：データベースの作成とデータベースへの接続

最初に、ローカル Ultra Light データベースを作成します。次に、作成したデータベースにアクセスする C++ アプリケーションを記述、コンパイル、実行します。

♦ Ultra Light データベースを作成するには、次の手順に従います。

1. INCLUDE 環境変数に、*c:¥vendor¥visualstudio8¥VC¥atlmfc¥src¥atl* を追加します。
2. このチュートリアルで作成されるファイルを保存するディレクトリを作成します。

   以降、このチュートリアルではこのディレクトリが *C:¥tutorial¥cpp¥* であることを前提に説明します。別の名前のディレクトリを作成した場合は、*C:¥tutorial¥cpp¥* の代わりにそのディレクトリを使用してください。
3. Sybase Central で Ultra Light を使用して、次の特性を持つデータベース *ULCustomer.udb* を新しいディレクトリに作成します。

   Sybase Central での Ultra Light の使用については、「データベース作成ウィザードを使用したデータベースの作成」『Ultra Light データベース管理とリファレンス』を参照してください。
4. **ULCustomer** という名前のテーブルをデータベースに追加します。次の ULCustomer テーブル仕様を使用します。

カラム名	データ型 (サイズ)	カラムの NULL 値の許可	デフォルト値	プライマリ・キー
cust_id	integer	いいえ	オートインクリメント	昇順
cust_name	varchar(30)	いいえ	なし	

5. Sybase Central でデータベースから切断します。そうしないと、実行ファイルが接続できません。

♦ Ultra Light データベースへの接続

1. Microsoft Visual C++ で、**[ファイル] - [新規作成]** を選択します。
2. **[ファイル]** タブで、**[C++ ソース ファイル]** を選択します。
3. チュートリアルのディレクトリに、そのファイルを *customer.cpp* として保存します。
4. Ultra Light ライブラリをインクルードし、Ultra Light ネームスペースを使用します。

   以下のコードを *customer.cpp* にコピーします。

```
#include <tchar.h>
#include <stdio.h>
#include "uliface.h"
using namespace UltraLite;
#define MAX_NAME_LEN 100
ULSqlca Tutca;
```


このコードでは、Tutca という名前の ULSqlca (Ultra Light SQL Communications Area) を定義しています。

Ultra Light ネームスペースを使用してクラス宣言を簡素化する方法の詳細については、「Ultra Light ネームスペースの使用」 10 ページを参照してください。

5. データベースに接続するための接続パラメータを定義します。

   次のコードでは、接続パラメータはハード・コードされています。実際のアプリケーションでは、ロケーションは実行時に指定されることもあります。

   以下のコードを *customer.cpp* にコピーします。

```
static ul_char const * ConnectionParms =
  UL_TEXT( "UID=DBA;PWD=sql" )
  UL_TEXT( ";DBF=C:¥¥tutorial¥¥cpp¥¥ULCustomer.udb" );
```

   接続パラメータの詳細については、「UltraLite_Connection_iface クラス」 158 ページを参照してください。

   特殊文字の処理
   ファイル名ロケーションの文字列にバックスラッシュ文字が含まれる場合は、バックスラッシュ文字をもう 1 つ追加してエスケープする必要があります。

6. アプリケーションでデータベース・エラーを処理するメソッドを定義します。

   Ultra Light は、アプリケーションにエラーを通知するためのコールバック・メカニズムを備えています。開発環境において、この関数は予期しないエラーを処理するメカニズムとして便利です。運用アプリケーションには、あらゆる一般的なエラー状況を処理するコードが含まれているのが普通です。アプリケーションでは、Ultra Light 関数を呼び出すごとにエラー確認するか、エラー・コールバック関数を使用するかを選択します。

   コールバック関数のサンプルを次に示します。

```
ul_error_action UL_GENNED_FN_MOD MyErrorCallBack(
 SQLCA *  Tutca,
 ul_void * user_data,
 ul_char * message_param )
{
   ul_error_action rc;

   (void) user_data;

   switch( Tutca->sqlcode ){
      // The following error is used for flow control - don't report it here
      case SQLE_NOTFOUND:
         rc = UL_ERROR_ACTION_CONTINUE;
      break;

      default:
         if (Tutca->sqlcode >= 0) { // warning or success
            rc = UL_ERROR_ACTION_DEFAULT;
         } else { // negative is real error
            _tprintf( _TEXT( "Error %ld: %s¥n" ), Tutca->sqlcode, message_param );
            rc = UL_ERROR_ACTION_CANCEL;
         }
      break;
   }
```

```
    return rc;
  }
```

Ultra Light では、ほとんどの場合はエラー SQLE_NOTFOUND を使用してアプリケーションのフローを制御しています。このエラーが通知されると、結果セットのループの終了がマークされます。上記の汎用エラー・ハンドラは、このエラー条件に対してはメッセージを出力しません。

エラー処理の詳細については、「エラー処理」 25 ページを参照してください。

7. データベースの接続を開くメソッドを定義します。

   データベース・ファイルがなかった場合は、エラー・メッセージが表示されます。そうでない場合は、接続が確立されます。

```
Connection * open_conn( DatabaseManager * dm ) {
  Connection * conn = dm->OpenConnection( Tutca, ConnectionParms );

  if( conn == UL_NULL ) {
      _tprintf( _TEXT("Unable to open existing database.¥n") );
   }
  return conn;
}
```

8. 次のタスクを実行する main メソッドを実装します。

   - DatabaseManager オブジェクトをインスタンス化します。Ultra Light オブジェクトはすべて、DatabaseManager オブジェクトから作成されます。
   - エラー処理関数を登録します。
   - データベースへの接続を開きます。
   - データベースとの接続を閉じ、データベース・マネージャを終了します。

```
int main() {
  ul_char buffer[ MAX_NAME_LEN ];

  Connection * conn;

  Tutca.Initialize();

  ULRegisterErrorCallback(
    Tutca.GetCA(), MyErrorCallBack,
    UL_NULL, buffer, sizeof (buffer));

  DatabaseManager * dm = ULInitDatabaseManager( Tutca );

  conn = open_conn( dm );

  if( conn == UL_NULL ){
   dm->Shutdown( Tutca );
      Tutca.Finalize();
      return 1;
  }
  // main processing code to be inserted here
  dm->Shutdown( Tutca );
  Tutca.Finalize();
  return 0;
}
```

9. ソース・ファイルのコンパイルとリンクを行います。

ソース・ファイルのコンパイル方法は、コンパイラによって異なります。以下の手順は、makefile で Microsoft Visual C++ コマンド・ライン・コンパイラを使用する場合です。

- コマンド・プロンプトを開き、チュートリアル・ディレクトリに変更します。
- *makefile* という名前の makefile を作成します。
- makefile で、ディレクトリをインクルード・パスに追加します。

  ```
  IncludeFolders=/I"$(SQLANY11)¥SDK¥Include"
  ```

- makefile で、ディレクトリをライブラリ・パスに追加します。

  ```
  LibraryFolders=/LIBPATH:"$(SQLANY11)¥UltraLite¥win32¥386¥Lib¥vs8"
  ```

- makefile で、ライブラリをリンカ・オプションに追加します。

  ```
  Libraries=ulimp.lib
  ```

  Ultra Light ランタイム・ライブラリの名前は *ulimp.lib* です。

- makefile で、コンパイラ・オプションを設定します。次のように、1 行でこれらのオプションを入力してください。

  ```
  CompileOptions=/c /nologo /W3 /Od /Zi /DWIN32 /DUL_USE_DLL
  ```

- makefile で、次のようにアプリケージョンのリンク命令を追加します。

  ```
  customer.exe: customer.obj
    link /NOLOGO /DEBUG customer.obj $(LibraryFolders) $(Libraries)
  ```

- makefile で、次のようにアプリケーションのコンパイル命令を追加します。

  ```
  customer.obj: customer.cpp
    cl $(CompileOptions) $(IncludeFolders) customer.cpp
  ```

- makefile を実行します。

  ```
  nmake
  ```

  customer.exe という実行ファイルが作成されます。

10. アプリケーションを実行します。

    コマンド・プロンプトで、**customer** と入力します。

レッスン 2 ：データベースへのデータの挿入

次の手順は、データベースにデータを追加する方法を示しています。

♦ データベースへローの追加

1. 以下のプロシージャを、*customer.cpp* の main メソッドの直前に追加します。

```
bool do_insert( Connection * conn ) {
  Table * table = conn->OpenTable( UL_TEXT("ULCustomer") );
  if( table == UL_NULL ) {
      _tprintf( _TEXT("Table not found: ULCustomer¥n") );
      return  false;
  }
  if( table->GetRowCount() == 0 ) {
      _tprintf( _TEXT("Inserting one row.¥n") );
      table->InsertBegin();
      table->Set( UL_TEXT("cust_name"), UL_TEXT("New Customer") );
      table->Insert();
      conn->Commit();
  } else {
      _tprintf( _TEXT("The table has %lu rows¥n"),
      table->GetRowCount() );
  }
  table->Release();
  return true;
}
```

   このプロシージャでは、次のタスクを実行します。

   - connection->OpenTable() メソッドを使用してテーブルを開きます。テーブルの操作を実行するには、Table オブジェクトを開いてください。
   - テーブルが空の場合は、ローを 1 つ追加します。このコードでは、ローを挿入するために、InsertBegin メソッドを使用して挿入モードに変更し、必要な各カラムに値を設定し、挿入を実行してデータベースにローを追加します。

     commit メソッドは、オートコミットがオフの場合のみ必要です。デフォルトでは、オートコミットは有効ですが、パフォーマンスを向上したり、複数操作トランザクションを行うために、オートコミットを無効にできます。
   - テーブルが空でない場合は、テーブルのロー数をレポートします。
   - Table オブジェクトを閉じ、関連付けられているリソースを解放します。
   - 操作が正常に完了したことを示す bool 値が返されます。
2. 作成した do_insert メソッドを呼び出します。

   次の行を、main() メソッドの open_conn の呼び出しの直後に追加します。

```
do_insert(conn);
```
3. nmake を実行してアプリケーションをコンパイルします。
4. コマンド・プロンプトで **customer** と入力してアプリケーションを実行します。

レッスン 3 ：テーブルのローの選択と出力

次の手順では、テーブルからローを取り出し、コマンド・ラインに出力します。

♦ テーブルのローのリスト

1. 以下のメソッドを *customer.cpp* に追加します。このメソッドでは、次のタスクを実行します。
   - Table オブジェクトを開きます。
   - カラム識別子を取得します。
   - 現在の位置を、テーブル内にある最初のローの前に設定します。

     テーブルに対するすべての操作は、現在の位置で実行されます。現在の位置は、最初のローの前、テーブルのいずれかのローの上、または最後のローの後ろです。この例のように、デフォルトでは、ローはプライマリ・キー値 (cust_id) に基づいて順序付けられています。別の順序でローを並べ替えるには、Ultra Light データベースにインデックスを追加し、そのインデックスを使用してテーブルを開きます。
   - 各ローに対して、cust_id と cust_name の値が書き出されます。ループは、最後のローの後に Next メソッドが false を返すまで繰り返されます。
   - Table オブジェクトを閉じます。

```
bool do_select( Connection * conn ) {
  Table * table = conn->OpenTable( _TEXT("ULCustomer") );
  if( table == UL_NULL ) {
    return false;
  }
  TableSchema * schema = table->GetSchema();
  if( schema == UL_NULL ) {
    table->Release();
    return false;
  }
  ul_column_num id_cid =
     schema->GetColumnID( UL_TEXT("cust_id") );
  ul_column_num cname_cid =
     schema->GetColumnID( UL_TEXT("cust_name") );

  schema->Release();

  _tprintf( _TEXT("¥n¥nTable 'ULCustomer' row contents:¥n") );

  while( table->Next() ) {
    ul_char cname[ MAX_NAME_LEN ];

    table->Get( cname_cid ).GetString(
              cname, MAX_NAME_LEN );

    _tprintf( _TEXT("id=%d, name=%s ¥n"), (int)table->Get(id_cid), cname);
  }
  table->Release();
  return true;
}
```

2. 次の行を、main メソッドの insert メソッドの呼び出しの直後に追加します。

```
do_select(conn);
```

3. *nmake* を実行してアプリケーションをコンパイルします。
4. コマンド・プロンプトで *customer* と入力してアプリケーションを実行します。

レッスン 4 ：アプリケーションへの同期の追加

このレッスンでは、アプリケーションを、コンピュータ上で動作している統合データベースに同期する方法について説明します。

次の手順では、アプリケーションに同期コードを追加し、Mobile Link サーバを起動し、アプリケーションを実行して同期します。

前のレッスンで作成した Ultra Light データベースは、Ultra Light 11 サンプル・データベースと同期します。Ultra Light 11 の Sample データベースの ULCustomer テーブルのカラムには、作成したローカル Ultra Light データベースの customer テーブルのカラムが含まれます。

このレッスンは、Mobile Link 同期についての知識を持っていることを前提としています。

♦ アプリケーションへの同期の追加

1. 以下のメソッドを *customer.cpp* に追加します。このメソッドでは、次のタスクを実行します。
   - ULEnableTcpipSynchronization を呼び出して、同期ストリームを TCP/IP に設定します。同期は、HTTP、HotSync、または HTTPS を使用しても実行できます。「Ultra Light クライアント」『Ultra Light データベース管理とリファレンス』を参照してください。
   - スクリプト・バージョンを設定します。Mobile Link 同期は、統合データベースに保存されているスクリプトによって制御されます。スクリプト・バージョンは、使用するスクリプト・セットを識別します。
   - Mobile Link ユーザ名を設定します。この値は、Mobile Link サーバでの認証に使用されます。アプリケーションによっては、Mobile Link のユーザ ID と Ultra Light データベースのユーザ ID を同じに設定しますが、これらの ID はあくまでも別のものです。
   - download_only パラメータを true に設定します。デフォルトでは、Mobile Link 同期は双方向です。このアプリケーションでは、テーブルのローがサンプル・データベースにアップロードされないように、ダウンロード専用同期を使用します。

```
bool do_sync( Connection * conn ) {
  ul_synch_info info;
  ul_stream_error * se = &info.stream_error;

  ULEnableTcpipSynchronization( Tutca.GetCA() );
  conn->InitSynchInfo( &info );
  info.stream = ULSocketStream();
  info.version = UL_TEXT( "custdb 11.0" );
  info.user_name = UL_TEXT( "50" );
  info.download_only = true;
  if( !conn->Synchronize( &info ) ) {
      _tprintf( _TEXT("Synchronization error ¥n" ));
    _tprintf( _TEXT("  stream_error_code is '%lu'¥n"), se->stream_error_code );
    _tprintf( _TEXT("  system_error_code is '%ld'¥n"), se->system_error_code );
    _tprintf( _TEXT("  error_string is '") );
    _tprintf( _TEXT("%s"), se->error_string );
    _tprintf( _TEXT("'¥n") );
   return false;
  }
  return true;
}
```

2. 次の行を、main メソッドの insert メソッドの呼び出しの直後、select メソッドの呼び出しの前に追加します。

```
do_sync(conn);
```

3. *nmake* を実行してアプリケーションをコンパイルします。

♦ データの同期

1. Mobile Link サーバを起動します。

   コマンド・プロンプトから次のコマンドを実行します。

```
mlsrv11 -c "dsn=SQL Anywhere 11 CustDB" -v+ -zu+
```

   -zu+ オプションを指定すると、ユーザの自動追加が行われます。-v+ オプションを指定すると、すべてのメッセージについて冗長ロギングがオンになります。

   このオプションの詳細については、「Mobile Link サーバ・オプション」 『Mobile Link - サーバ管理』を参照してください。

2. コマンド・プロンプトで *customer* と入力してアプリケーションを実行します。

   Mobile Link サーバのウィンドウでは、同期の進行状況を示すステータス・メッセージが表示されます。同期が正しく行われると、最後に「**同期が完了しました。**」というメッセージが表示されます。

チュートリアルのコード・リスト

この章で説明したチュートリアル・プログラムの完全なコードを次に示します。

```
#include <tchar.h>
#include <stdio.h>
#include "uliface.h"
using namespace UltraLite;
#define MAX_NAME_LEN 100
ULSqlca Tutca;

static ul_char const * ConnectionParms =
  UL_TEXT( "UID=DBA;PWD=sql;" )
  UL_TEXT( "DBF=C:¥¥temp¥¥ULCustomer.udb" );

ul_error_action UL_GENNED_FN_MOD MyErrorCallBack(
 SQLCA *  Tutca,
 ul_void * user_data,
 ul_char * message_param )
{
    ul_error_action rc;

    (void) user_data;

    switch( Tutca->sqlcode ){
        // The following error is used for flow control - don't report it here
        case SQLE_NOTFOUND:
            rc = UL_ERROR_ACTION_CONTINUE;
        break;

        default:
            if (Tutca->sqlcode >= 0) { // warning or success
                rc = UL_ERROR_ACTION_DEFAULT;
            } else { // negative is real error
                _tprintf( _TEXT( "Error %ld: %s¥n" ), Tutca->sqlcode, message_param );
                rc = UL_ERROR_ACTION_CANCEL;
            }
        break;
     }
    return rc;
 }
Connection * open_conn( DatabaseManager * dm ) {
  Connection * conn = dm->OpenConnection( Tutca, ConnectionParms );
  if( conn == UL_NULL ) {
      _tprintf( _TEXT("Unable to open existing database.¥n") );
   }
  return conn;
}
// Open table, insert 1 row if table is currently empty

bool do_insert( Connection * conn ) {
  Table * table = conn->OpenTable( UL_TEXT("ULCustomer") );
  if( table == UL_NULL ) {
      _tprintf( _TEXT("Table not found: ULCustomer¥n") );
      return  false;
  }
  if( table->GetRowCount() == 0 ) {
      _tprintf( _TEXT("Inserting one row.¥n") );
      table->InsertBegin();
      table->Set( UL_TEXT("cust_name"), UL_TEXT("New Customer") );
      table->Insert();

      conn->Commit();
```

```
  } else {
      _tprintf( _TEXT("The table has %lu rows¥n"),
      table->GetRowCount() );
  }
  table->Release();
  return true;
}// Open table, display data from all rows

bool do_select( Connection * conn ) {
  Table * table = conn->OpenTable( _TEXT("ULCustomer") );
  if( table == UL_NULL ) {
  return false;
  }
  TableSchema * schema = table->GetSchema();
  if( schema == UL_NULL ) {
    table->Release();
    return false;
  }
  ul_column_num id_cid = schema->GetColumnID( UL_TEXT("cust_id") );
  ul_column_num cname_cid = schema->GetColumnID( UL_TEXT("cust_name") );

  schema->Release();

  _tprintf( _TEXT("¥n¥nTable 'ULCustomer' row contents:¥n") );

  while( table->Next() ) {
    ul_char cname[ MAX_NAME_LEN ];

    table->Get( cname_cid ).GetString( cname, MAX_NAME_LEN );

    _tprintf( _TEXT("id=%d, name=%s ¥n"), (int)table->Get( id_cid ), cname );
  }
  table->Release();
  return true;
}
// sync database with MobiLink connection to reference database

bool do_sync( Connection * conn ) {
  ul_synch_info info;
  ul_stream_error * se = &info.stream_error;

  ULEnableTcpipSynchronization( Tutca.GetCA() );
  conn->InitSynchInfo( &info );
  info.stream = ULSocketStream();
  info.version = UL_TEXT( "custdb 11.0" );
  info.user_name = UL_TEXT( "50" );
  info.download_only = true;
  if( !conn->Synchronize( &info ) ) {
      _tprintf( _TEXT("Synchronization error ¥n" ));
    _tprintf( _TEXT("  stream_error_code is '%lu'¥n"), se->stream_error_code );
    _tprintf( _TEXT("  system_error_code is '%ld'¥n"), se->system_error_code );
    _tprintf( _TEXT("  error_string is '") );
    _tprintf( _TEXT("%s"), se->error_string );
    _tprintf( _TEXT("'¥n") );
   return false;
  }
  return true;
}

int main() {
  ul_char buffer[ MAX_NAME_LEN ];

  Connection * conn;
```



```
  Tutca.Initialize();

  ULRegisterErrorCallback(
    Tutca.GetCA(), MyErrorCallBack,
    UL_NULL, buffer, sizeof (buffer));

  DatabaseManager * dm = ULInitDatabaseManager( Tutca );

  if( dm == UL_NULL ){
    // You may have mismatched UNICODE vs. ANSI runtimes.
  Tutca.Finalize();
    return 1;
  }

  conn = open_conn( dm );

  if( conn == UL_NULL ){
   dm->Shutdown( Tutca );
      Tutca.Finalize();
      return 1;
  }

  do_insert (conn);
  do_sync (conn);
  do_select (conn);


  dm->Shutdown( Tutca );
  Tutca.Finalize();
  return 0;
}
```

用語解説

用語解説

Adaptive Server Anywhere (ASA)

SQL Anywhere Studio のリレーショナル・データベース・サーバ・コンポーネントであり、主に、モバイル環境と埋め込み環境、または小規模および中規模のビジネス用のサーバとして使用されます。バージョン 10.0.0 で、Adaptive Server Anywhere は SQL Anywhere サーバに、SQL Anywhere Studio は SQL Anywhere にそれぞれ名前が変更されました。

参照：「SQL Anywhere」 350 ページ。

Carrier

Mobile Link システム・テーブルまたは Notifier プロパティ・ファイルに保存される Mobile Link オブジェクトで、システム起動同期で使用される通信業者に関する情報が含まれます。

参照：「サーバ起動同期」 355 ページ。

DB 領域

データ用の領域をさらに作成する追加のデータベース・ファイルです。1 つのデータベースは 13 個までのファイルに保管されます (初期ファイル 1 つと 12 の DB 領域)。各テーブルは、そのインデックスとともに、単一のデータベース・ファイルに含まれている必要があります。CREATE DBSPACE という SQL コマンドで、新しいファイルをデータベースに追加できます。

参照：「データベース・ファイル」 359 ページ。

DBA 権限

ユーザに、データベース内の管理作業を許可するレベルのパーミッションです。DBA ユーザにはデフォルトで DBA 権限が与えられています。

参照：「データベース管理者 (DBA)」 359 ページ。

EBF

Express Bug Fix の略です。Express Bug Fix は、1 つ以上のバグ・フィックスが含まれる、ソフトウェアのサブセットです。これらのバグ・フィックスは、更新のリリース・ノートにリストされます。バグ・フィックス更新を適用できるのは、同じバージョン番号を持つインストール済みのソフトウェアに対してだけです。このソフトウェアについては、ある程度のテストが行われているとはいえ、完全なテストが行われたわけではありません。自分自身でソフトウェアの妥当性を確かめるまでは、アプリケーションとともにこれらのファイルを配布しないでください。

Embedded SQL

C プログラム用のプログラミング・インタフェースです。SQL Anywhere の Embedded SQL は ANSI と IBM 規格に準拠して実装されています。

FILE

SQL Remote のレプリケーションでは、レプリケーション・メッセージのやりとりのために共有ファイルを使うメッセージ・システムのことです。これは特定のメッセージ送信システムに頼らずにテストやインストールを行うのに便利です。

参照：「レプリケーション」 367 ページ。

grant オプション

他のユーザにパーミッションを許可できるレベルのパーミッションです。

iAnywhere JDBC ドライバ

iAnywhere JDBC ドライバでは、pure Java である jConnect JDBC ドライバに比べて何らかの有利なパフォーマンスや機能を備えた JDBC ドライバが提供されます。ただし、このドライバは pure Java ソリューションではありません。iAnywhere JDBC ドライバは一般に推奨されるドライバです。

参照：

- 「JDBC」 347 ページ
- 「jConnect」 347 ページ

InfoMaker

レポート作成とデータ管理用のツールです。洗練されたフォーム、レポート、グラフ、クロスタブ、テーブルを作成できます。また、これらを基本的な構成要素とするアプリケーションも作成できます。

Interactive SQL

データベース内のデータの変更や問い合わせ、データベース構造の修正ができる、SQL Anywhere のアプリケーションです。Interactive SQL では、SQL 文を入力するためのウィンドウ枠が表示されます。また、クエリの進捗情報や結果セットを返すウィンドウ枠も表示されます。

JAR ファイル

Java アーカイブ・ファイルです。Java のアプリケーションで使用される 1 つ以上のパッケージの集合からなる圧縮ファイルのフォーマットです。Java プログラムをインストールしたり実行したりするのに必要なリソースが 1 つの圧縮ファイルにすべて収められています。

Java クラス

Java のコードの主要な構造単位です。これはプロシージャや変数の集まりで、すべてがある一定のカテゴリに関連しているためグループ化されたものです。

jConnect

JavaSoft JDBC 標準を Java で実装したものです。これにより、Java 開発者は多層／異機種環境でもネイティブなデータベース・アクセスができます。iAnywhere JDBC ドライバは一般に推奨されるドライバです。

参照：

- 「JDBC」 347 ページ
- 「iAnywhere JDBC ドライバ」 346 ページ

JDBC

Java Database Connectivity の略です。Java アプリケーションからリレーショナル・データにアクセスすることを可能にする SQL 言語プログラミング・インタフェースです。推奨 JDBC ドライバは、iAnywhere JDBC ドライバです。

参照：

- 「jConnect」 347 ページ
- 「iAnywhere JDBC ドライバ」 346 ページ

Listener

Mobile Link サーバ起動同期に使用される、dblsn という名前のプログラムです。Listener はリモート・デバイスにインストールされ、Push 通知を受け取ったときにデバイス上でアクションが開始されるように設定されます。

参照：「サーバ起動同期」 355 ページ。

LTM

LTM (Log Transfer Manager) は、Replication Agent とも呼ばれます。Replication Server と併用することで、LTM はデータベース・トランザクション・ログを読み込み、コミットされた変更を Sybase Replication Server に送信します。

参照：「Replication Server」 350 ページ。

Mobile Link

Ultra Light と SQL Anywhere のリモート・データベースを統合データベースと同期させるために設計された、セッションベース同期テクノロジです。

参照：

- 「統合データベース」 374 ページ
- 「同期」 374 ページ
- 「Ultra Light」 351 ページ

Mobile Link クライアント

2 種類の Mobile Link クライアントがあります。SQL Anywhere リモート・データベース用の Mobile Link クライアントは、dbmlsync コマンド・ライン・ユーティリティです。Ultra Light リモート・データベース用の Mobile Link クライアントは、Ultra Light ランタイム・ライブラリに組み込まれています。

Mobile Link サーバ

Mobile Link 同期を実行する、mlsrv11 という名前のコンピュータ・プログラムです。

Mobile Link システム・テーブル

Mobile Link の同期に必要なシステム・テーブルです。Mobile Link 設定スクリプトによって、Mobile Link 統合データベースにインストールされます。

Mobile Link モニタ

Mobile Link の同期をモニタするためのグラフィカル・ツールです。

Mobile Link ユーザ

Mobile Link ユーザは、Mobile Link サーバに接続するのに使用されます。Mobile Link ユーザをリモート・データベースに作成し、統合データベースに登録します。Mobile Link ユーザ名はデータベース・ユーザ名から完全に独立しています。

Notifier

Mobile Link サーバ起動同期に使用されるプログラムです。Notifier は Mobile Link サーバに統合されており、統合データベースに Push 要求がないか確認し、Push 通知を送信します。

参照：

- 「サーバ起動同期」 355 ページ
- 「Listener」 347 ページ

ODBC

Open Database Connectivity の略です。データベース管理システムに対する Windows の標準的なインタフェースです。ODBC は、SQL Anywhere がサポートするインタフェースの 1 つです。

ODBC アドミニストレータ

Windows オペレーティング・システムに付属している Microsoft のプログラムです。ODBC データ・ソースの設定に使用します。

ODBC データ・ソース

ユーザが ODBC からアクセスするデータと、そのデータにアクセスするために必要な情報の仕様です。

PDB

Palm のデータベース・ファイルです。

PowerDesigner

データベース・モデリング・アプリケーションです。これを使用すると、データベースやデータ・ウェアハウスの設計に対する構造的なアプローチが可能となります。SQL Anywhere には、PowerDesigner の Physical Data Model コンポーネントが付属します。

PowerJ

Java アプリケーション開発に使用する Sybase 製品です。

Push 通知

QAnywhere では、メッセージ転送を開始するよう QAnywhere クライアントに対して指示するために、サーバから QAnywhere クライアントに配信される特殊なメッセージです。Mobile Link サーバ起動同期では、Push 要求データや内部情報を含むデバイスに Notifer から配信される特殊なメッセージです。

参照：

- 「QAnywhere」 349 ページ
- 「サーバ起動同期」 355 ページ

Push 要求

Mobile Link サーバ起動同期において、Push 通知をデバイスに送信する必要があるかどうかを判断するために Notifier が確認する、結果セット内の値のローです。

参照：「サーバ起動同期」 355 ページ。

QAnywhere

アプリケーション間メッセージング (モバイル・デバイス間メッセージングやモバイル・デバイスとエンタープライズの間のメッセージングなど) を使用すると、モバイル・デバイスや無線デバイスで動作しているカスタム・プログラムと、集中管理されているサーバ・アプリケーションとの間で通信できます。

QAnywhere Agent

QAnywhere では、クライアント・デバイス上で動作する独立のプロセスのことです。クライアント・メッセージ・ストアをモニタリングし、メッセージを転送するタイミングを決定します。

REMOTE DBA 権限

SQL Remote では、Message Agent (dbremote) で必要なパーミッションのレベルを指します。Mobile Link では、SQL Anywhere 同期クライアント (dbmlsync) で必要なパーミッションのレベルを指します。Message Agent (dbremote) または同期クライアントがこの権限のあるユーザとして接続した場合、DBA のフル・アクセス権が与えられます。Message Agent (dbremote) または同期クライアント (dbmlsync) から接続しない場合、このユーザ ID にはパーミッションは追加されません。

参照：「DBA 権限」 345 ページ。

Replication Agent

参照：「LTM」 347 ページ。

Replication Server

SQL Anywhere と Adaptive Server Enterprise で動作する、Sybase による接続ベースのレプリケーション・テクノロジです。Replication Server は、少数のデータベース間でほぼリアルタイムのレプリケーションを行うことを目的に設計されています。

参照：「LTM」 347 ページ。

SQL

リレーショナル・データベースとの通信に使用される言語です。SQL は ANSI により標準が定義されており、その最新版は SQL-2003 です。SQL は、公認されてはいませんが、Structured Query Language の略です。

SQL Anywhere

SQLAnywhere のリレーショナル・データベース・サーバ・コンポーネントであり、主に、モバイル環境と埋め込み環境、または小規模および中規模のビジネス用のサーバとして使用されます。SQL Anywhere は、SQL Anywhere RDBMS、Ultra Light RDBMS、Mobile Link 同期ソフトウェア、その他のコンポーネントを含むパッケージの名前でもあります。

SQL Remote

統合データベースとリモート・データベース間で双方向レプリケーションを行うための、メッセージベースのデータ・レプリケーション・テクノロジです。統合データベースとリモート・データベースは、SQL Anywhere である必要があります。

SQL ベースの同期

Mobile Link では、Mobile Link イベントを使用して、テーブル・データを Mobile Link でサポートされている統合データベースに同期する方法のことです。SQL ベースの同期では、SQL を直接使用したり、Java と .NET 用の Mobile Link サーバ API を使用して SQL を返すことができます。

SQL 文

DBMS に命令を渡すために設計された、SQL キーワードを含む文字列です。

参照：

- 「スキーマ」 357 ページ
- 「SQL」 350 ページ
- 「データベース管理システム (DBMS)」 359 ページ

Sybase Central

SQL Anywhere データベースのさまざまな設定、プロパティ、ユーティリティを使用できる、グラフィカル・ユーザ・インタフェースを持つデータベース管理ツールです。Mobile Link などの他の iAnywhere 製品を管理する場合にも使用できます。

SYS

システム・オブジェクトの大半を所有する特別なユーザです。一般のユーザは SYS でログインできません。

Ultra Light

小型デバイス、モバイル・デバイス、埋め込みデバイス用に最適化されたデータベースです。対象となるプラットフォームとして、携帯電話、ポケットベル、パーソナル・オーガナイザなどが挙げられます。

Ultra Light ランタイム

組み込みの Mobile Link 同期クライアントを含む、インプロセス・リレーショナル・データベース管理システムです。Ultra Light ランタイムは、Ultra Light の各プログラミング・インタフェースで使用されるライブラリと、Ultra Light エンジンの両方に含まれます。

Windows

Windows Vista、Windows XP、Windows 200x などの、Microsoft Windows オペレーティング・システムのファミリのことです。

Windows CE

「Windows Mobile」 351 ページを参照してください。

Windows Mobile

Microsoft がモバイル・デバイス用に開発したオペレーティング・システムのファミリです。

アーティクル

Mobile Link または SQL Remote では、テーブル全体もしくはテーブル内のカラムとローのサブセットを表すデータベース・オブジェクトを指します。アーティクルの集合がパブリケーションです。

参照：

- 「レプリケーション」 367 ページ
- 「パブリケーション」 362 ページ

アップロード

同期中に、リモート・データベースから統合データベースにデータが転送される段階です。

アトミックなトランザクション

完全に処理されるかまったく処理されないことが保証される 1 つのトランザクションです。エラーによってアトミックなトランザクションの一部が処理されなかった場合は、データベースが一貫性のない状態になるのを防ぐために、トランザクションがロールバックされます。

アンロード

データベースをアンロードすると、データベースの構造かデータ、またはその両方がテキスト・ファイルにエクスポートされます (構造は SQL コマンド・ファイルに、データはカンマ区切りの ASCII ファイルにエクスポートされます)。データベースのアンロードには、アンロード・ユーティリティを使用します。

また、UNLOAD 文を使って、データから抜粋した部分だけをアンロードできます。

イベント・モデル

Mobile Link では、同期を構成する、begin_synchronization や download_cursor などの一連のイベントのことです。イベントは、スクリプトがイベント用に作成されると呼び出されます。

インクリメンタル・バックアップ

トランザクション・ログ専用のバックアップです。通常、フル・バックアップとフル・バックアップの間に使用します。

参照：「トランザクション・ログ」 361 ページ。

インデックス

ベース・テーブルにある 1 つ以上のカラムに関連付けられた、キーとポインタのソートされたセットです。テーブルの 1 つ以上のカラムにインデックスが設定されていると、パフォーマンスが向上します。

ウィンドウ

分析関数の実行対象となるローのグループです。ウィンドウには、ウィンドウ定義内のグループ化指定に従って分割されたデータの、1 つ、複数、またはすべてのローが含まれます。ウィンドウは、入力の現在のローについて計算を実行する必要があるローの数や範囲を含むように移動します。ウィンドウ構成の主な利点は、追加のクエリを実行しなくても、結果をグループ化して分析する機会が増えることです。

エージェント ID

参照：「クライアント・メッセージ・ストア ID」 354 ページ。

エンコード

文字コードとも呼ばれます。エンコードは、文字セットの各文字が情報の 1 つまたは複数のバイトにマッピングされる方法のことで、一般的に 16 進数で表現されます。UTF-8 はエンコードの例です。

参照：

- 「文字セット」 375 ページ
- 「コード・ページ」 355 ページ
- 「照合」 372 ページ

オブジェクト・ツリー

Sybase Central では、データベース・オブジェクトの階層を指します。オブジェクト・ツリーの最上位には、現在使用しているバージョンの Sybase Central がサポートするすべての製品が表示されます。それぞれの製品を拡張表示すると、オブジェクトの下位ツリーが表示されます。

参照：「Sybase Central」 351 ページ。

カーソル

結果セットへの関連付けに名前を付けたもので、プログラミング・インタフェースからローにアクセスしたり更新したりするときに使用します。SQL Anywhere では、カーソルはクエリ結果内で前方や後方への移動をサポートします。カーソルは、カーソル結果セット (通常 SELECT 文で定義される) とカーソル位置の 2 つの部分から構成されます。

参照：

- 「カーソル結果セット」 354 ページ
- 「カーソル位置」 353 ページ

カーソル位置

カーソル結果セット内の 1 つのローを指すポインタ。

参照：

- 「カーソル」 353 ページ
- 「カーソル結果セット」 354 ページ

カーソル結果セット

カーソルに関連付けられたクエリから生成されるローのセットです。

参照：

- 「カーソル」 353 ページ
- 「カーソル位置」 353 ページ

クエリ

データベースのデータにアクセスしたり、そのデータを操作したりする SQL 文や SQL 文のグループです。

参照：「SQL」 350 ページ。

クライアント／サーバ

あるアプリケーション (クライアント) が別のアプリケーション (サーバ) に対して情報を送受信するソフトウェア・アーキテクチャのことです。通常この 2 種類のアプリケーションは、ネットワークに接続された異なるコンピュータ上で実行されます。

クライアント・メッセージ・ストア

QAnywhere では、メッセージを保管するリモート・デバイスにある SQL Anywhere データベースのことです。

クライアント・メッセージ・ストア ID

QAnywhere では、Mobile Link リモート ID のことです。これによって、クライアント・メッセージ・ストアがユニークに識別されます。

グローバル・テンポラリ・テーブル

明示的に削除されるまでデータ定義がすべてのユーザに表示されるテンポラリ・テーブルです。グローバル・テンポラリ・テーブルを使用すると、各ユーザが、1 つのテーブルのまったく同じインスタンスを開くことができます。デフォルトでは、コミット時にローが削除され、接続終了時にもローが削除されます。

参照：

- 「テンポラリ・テーブル」 360 ページ
- 「ローカル・テンポラリ・テーブル」 368 ページ

ゲートウェイ

Mobile Link システム・テーブルまたは Notifier プロパティ・ファイルに保存される Mobile Link オブジェクトで、システム起動同期用のメッセージの送信方法に関する情報が含まれます。

参照：「サーバ起動同期」 355 ページ。

コード・ページ

コード・ページは、文字セットの文字を数値表示 (通常 0 ～ 255 の整数) にマッピングするエンコードです。Windows Code Page 1252 などのコード・ページがあります。このマニュアルの目的上、コード・ページとエンコードは同じ意味で使用されます。

参照：

- 「文字セット」 375 ページ
- 「エンコード」 353 ページ
- 「照合」 372 ページ

コマンド・ファイル

SQL 文で構成されたテキスト・ファイルです。コマンド・ファイルは手動で作成できますが、データベース・ユーティリティによって自動的に作成することもできます。たとえば、dbunload ユーティリティを使うと、指定されたデータベースの再構築に必要な SQL 文で構成されたコマンド・ファイルを作成できます。

サーバ・メッセージ・ストア

QAnywhere では、サーバ上のリレーショナル・データベースです。このデータベースは、メッセージを、クライアント・メッセージ・ストアまたは JMS システムに転送されるまで一時的に格納します。メッセージは、サーバ・メッセージ・ストアを介して、クライアント間で交換されます。

サーバ管理要求

XML 形式の QAnywhere メッセージです。サーバ・メッセージ・ストアを管理したり、QAnywhere アプリケーションをモニタリングするために QAnywhere システム・キューに送信されます。

サーバ起動同期

Mobile Link サーバから Mobile Link 同期を開始する方法です。

サービス

Windows オペレーティング・システムで、アプリケーションを実行するユーザ ID がログオンしていないときにアプリケーションを実行する方法です。

サブクエリ

別の SELECT 文、INSERT 文、UPDATE 文、DELETE 文、または別のサブクエリの中にネストされた SELECT 文です。

相関とネストの 2 種類のサブクエリがあります。

サブスクリプション

Mobile Link 同期では、パブリケーションと Mobile Link ユーザ間のクライアント・データベース内のリンクであり、そのパブリケーションが記述したデータの同期を可能にします。

SQL Remote レプリケーションでは、パブリケーションとリモート・ユーザ間のリンクのことで、これによりリモート・ユーザはそのパブリケーションの更新内容を統合データベースとの間で交換できます。

参照：

- 「パブリケーション」 362 ページ
- 「Mobile Link ユーザ」 348 ページ

システム・オブジェクト

SYS または dbo が所有するデータベース・オブジェクトです。

システム・テーブル

SYS または dbo が所有するテーブルです。メタデータが格納されています。システム・テーブル (データ辞書テーブルとしても知られています) はデータベース・サーバが作成し管理します。

システム・ビュー

すべてのデータベースに含まれているビューです。システム・テーブル内に格納されている情報をわかりやすいフォーマットで示します。

ジョイン

指定されたカラムの値を比較することによって 2 つ以上のテーブルにあるローをリンクする、リレーショナル・システムでの基本的な操作です。

ジョイン・タイプ

SQL Anywhere では、クロス・ジョイン、キー・ジョイン、ナチュラル・ジョイン、ON 句を使ったジョインの 4 種類のジョインが使用されます。

参照：「ジョイン」 356 ページ。

ジョイン条件

ジョインの結果に影響を及ぼす制限です。ジョイン条件は、JOIN の直後に ON 句か WHERE 句を挿入して指定します。ナチュラル・ジョインとキー・ジョインについては、SQL Anywhere がジョイン条件を生成します。

参照：

- 「ジョイン」 356 ページ
- 「生成されたジョイン条件」 373 ページ

スキーマ

テーブル、カラム、インデックス、それらの関係などを含んだデータベース構造です。

スクリプト

Mobile Link では、Mobile Link のイベントを処理するために記述されたコードです。スクリプトは、業務上の要求に適合するように、データ交換をプログラム的に制御します。

参照：「イベント・モデル」 352 ページ。

スクリプト・バージョン

Mobile Link では、同期を作成するために同時に適用される、一連の同期スクリプトです。

スクリプトベースのアップロード

Mobile Link では、ログ・ファイルを使用した方法の代わりとなる、アップロード処理のカスタマイズ方法です。

ストアド・プロシージャ

ストアド・プロシージャは、データベースに保存され、データベース・サーバに対する一連の操作やクエリを実行するために使用される SQL 命令のグループです。

スナップショット・アイソレーション

読み込み要求を発行するトランザクション用のデータのコミットされたバージョンを返す、独立性レベルの種類です。SQL Anywhere では、スナップショット、文のスナップショット、読み込み専用文のスナップショットの 3 つのスナップショットの独立性レベルがあります。スナップショット・アイソレーションが使用されている場合、読み込み処理は書き込み処理をブロックしません。

参照：「独立性レベル」 375 ページ。

セキュア機能

データベース・サーバが起動されたときに、そのデータベース・サーバで実行されているデータベースでは使用できないように -sf オプションによって指定される機能です。

セッション・ベースの同期

統合データベースとリモート・データベースの両方でデータ表現の一貫性が保たれる同期です。Mobile Link はセッション・ベースです。

ダイレクト・ロー・ハンドリング

Mobile Link では、テーブル・データを Mobile Link でサポートされている統合データベース以外のソースに同期する方法のことです。アップロードとダウンロードの両方をダイレクト・ロー・ハンドリングで実装できます。

参照：

- 「統合データベース」 374 ページ
- 「SQL ベースの同期」 351 ページ

ダウンロード

同期中に、統合データベースからリモート・データベースにデータが転送される段階です。

チェックサム

データベース・ページを使用して記録されたデータベース・ページのビット数の合計です。チェックサムを使用すると、データベース管理システムは、ページがディスクに書き込まれるときに数が一致しているかを確認することで、ページの整合性を検証できます。数が一致した場合は、ページが正常に書き込まれたとみなされます。

チェックポイント

データベースに加えたすべての変更内容がデータベース・ファイルに保存されるポイントです。通常、コミットされた変更内容はトランザクション・ログだけに保存されます。

データ・キューブ

同じ結果を違う方法でグループ化およびソートされた内容を各次元に反映した、多次元の結果セットです。データ・キューブは、セルフジョイン・クエリと相関サブクエリを必要とするデータの複雑な情報を提供します。データ・キューブは OLAP 機能の一部です。

データベース

プライマリ・キーと外部キーによって関連付けられているテーブルの集合です。これらのテーブルでデータベース内の情報が保管されます。また、テーブルとキーによってデータベースの構造が定義されます。データベース管理システムでこの情報にアクセスします。

参照：

- 「外部キー」 369 ページ
- 「プライマリ・キー」 364 ページ
- 「データベース管理システム (DBMS)」 359 ページ
- 「リレーショナル・データベース管理システム (RDBMS)」 367 ページ

データベース・オブジェクト

情報を保管したり受け取ったりするデータベース・コンポーネントです。テーブル、インデックス、ビュー、プロシージャ、トリガはデータベース・オブジェクトです。

データベース・サーバ

データベース内にある情報へのすべてのアクセスを規制するコンピュータ・プログラムです。SQL Anywhere には、ネットワーク・サーバとパーソナル・サーバの 2 種類のサーバがあります。

データベース・ファイル

データベースは 1 つまたは複数のデータベース・ファイルに保持されます。まず、初期ファイルがあり、それに続くファイルは DB 領域と呼ばれます。各テーブルは、それに関連付けられているインデックスとともに、単一のデータベース・ファイルに含まれている必要があります。

参照：「DB 領域」 345 ページ。

データベース管理システム (DBMS)

データベースを作成したり使用したりするためのプログラムの集合です。

参照：「リレーショナル・データベース管理システム (RDBMS)」 367 ページ。

データベース管理者 (DBA)

データベースの管理に必要なパーミッションを持つユーザです。DBA は、データベース・スキーマのあらゆる変更や、ユーザやグループの管理に対して、全般的な責任を負います。データベース管理者のロールはデータベース内に自動的に作成されます。その場合、ユーザ ID は DBA であり、パスワードは sql です。

データベース所有者 (dbo)

SYS が所有しないシステム・オブジェクトを所有する特別なユーザです。

参照：

- 「データベース管理者 (DBA)」 359 ページ
- 「SYS」 351 ページ

データベース接続

クライアント・アプリケーションとデータベース間の通信チャネルです。接続を確立するためには有効なユーザ ID とパスワードが必要です。接続中に実行できるアクションは、そのユーザ ID に付与された権限によって決まります。

データベース名

サーバがデータベースをロードするとき、そのデータベースに指定する名前です。デフォルトのデータベース名は、初期データベース・ファイルのルート名です。

参照：「データベース・ファイル」 359 ページ。

データ型

CHAR や NUMERIC などのデータのフォーマットです。ANSI SQL 規格では、サイズ、文字セット、照合に関する制限もデータ型に組み込めます。

参照：「ドメイン」 360 ページ。

データ操作言語 (DML)

データベース内のデータの操作に使う SQL 文のサブセットです。DML 文は、データベース内のデータを検索、挿入、更新、削除します。

データ定義言語 (DDL)

データベース内のデータの構造を定義するときに使う SQL 文のサブセットです。DDL 文は、テーブルやユーザなどのデータベース・オブジェクトを作成、変更、削除できます。

デッドロック

先へ進めない場所に一連のトランザクションが到達する状態です。

デバイス・トラッキング

Mobile Link サーバ起動同期において、デバイスを特定する Mobile Link のユーザ名を使用して、メッセージのアドレスを指定できる機能です。

参照：「サーバ起動同期」 355 ページ。

テンポラリ・テーブル

データを一時的に保管するために作成されるテーブルです。グローバルとローカルの 2 種類があります。

参照：

- 「ローカル・テンポラリ・テーブル」 368 ページ
- 「グローバル・テンポラリ・テーブル」 354 ページ

ドメイン

適切な位置に精度や小数点以下の桁数を含み、さらにオプションとしてデフォルト値や CHECK 条件などを含んでいる、組み込みデータ型のエイリアスです。ドメインには、通貨データ型のように SQL Anywhere が事前に定義したものもあります。ユーザ定義データ型とも呼ばれます。

参照：「データ型」 360 ページ。

トランザクション

作業の論理単位を構成する一連の SQL 文です。1 つのトランザクションは完全に処理されるかまったく処理されないかのどちらかです。SQL Anywhere は、ロック機能のあるトランザクション処理をサポートしているので、複数のトランザクションが同時にデータベースにアクセスしてもデータを壊すことがありません。トランザクションは、データに加えた変更を永久なものにする COMMIT 文か、トランザクション中に加えられたすべての変更を元に戻す ROLLBACK 文のいずれかで終了します。

トランザクション・ログ

データベースに対するすべての変更内容が、変更された順に格納されるファイルです。パフォーマンスを向上させ、データベース・ファイルが破損した場合でもデータをリカバリできます。

トランザクション・ログ・ミラー

オプションで設定できる、トランザクション・ログ・ファイルの完全なコピーのことで、トランザクション・ログと同時に管理されます。データベースの変更がトランザクション・ログへ書き込まれると、トランザクション・ログ・ミラーにも同じ内容が書き込まれます。

ミラー・ファイルは、トランザクション・ログとは別のデバイスに置いてください。一方のデバイスに障害が発生しても、もう一方のログにリカバリのためのデータが確保されます。

参照：「トランザクション・ログ」 361 ページ。

トランザクション単位の整合性

Mobile Link で、同期システム全体でのトランザクションの管理を保証します。トランザクション全体が同期されるか、トランザクション全体がまったく同期されないかのどちらかになります。

トリガ

データを修正するクエリをユーザが実行すると、自動的に実行されるストアド・プロシージャの特別な形式です。

参照：

- 「ロー・レベルのトリガ」 368 ページ
- 「文レベルのトリガ」 375 ページ
- 「整合性」 372 ページ

ネットワーク・サーバ

共通ネットワークを共有するコンピュータからの接続を受け入れるデータベース・サーバです。

参照：「パーソナル・サーバ」 362 ページ。

ネットワーク・プロトコル

TCP/IP や HTTP などの通信の種類です。

パーソナル・サーバ

クライアント・アプリケーションが実行されているコンピュータと同じマシンで実行されているデータベース・サーバです。パーソナル・データベース・サーバは、単一のコンピュータ上で単一のユーザが使用しますが、そのユーザからの複数の同時接続をサポートできます。

パッケージ

Java では、それぞれが互いに関連のあるクラスの集合を指します。

ハッシュ

ハッシュは、インデックスのエントリをキーに変換する、インデックスの最適化のことです。インデックスのハッシュの目的は、必要なだけの実際のロー・データをロー ID に含めることで、インデックスされた値を特定するためのローの検索、ロード、アンパックという負荷の高い処理を避けることです。

パフォーマンス統計値

データベース・システムのパフォーマンスを反映する値です。たとえば、CURRREAD 統計値は、データベース・サーバが要求したファイル読み込みのうち、現在まだ完了していないものの数を表します。

パブリケーション

Mobile Link または SQL Remote では、同期されるデータを識別するデータベース・オブジェクトのことです。Mobile Link では、クライアント上にのみ存在します。1 つのパブリケーションは複数のアーティクルから構成されています。SQL Remote ユーザは、パブリケーションに対してサブスクリプションを作成することによって、パブリケーションを受信できます。Mobile Link ユーザは、パブリケーションに対して同期サブスクリプションを作成することによって、パブリケーションを同期できます。

参照：

- 「レプリケーション」 367 ページ
- 「アーティクル」 352 ページ
- 「パブリケーションの更新」 362 ページ

パブリケーションの更新

SQL Remote レプリケーションでは、単一のデータベース内の 1 つまたは複数のパブリケーションに対して加えられた変更のリストを指します。パブリケーションの更新は、レプリケーション・メッセージの一部として定期的にリモート・データベースへ送られます。

参照：

- 「レプリケーション」 367 ページ
- 「パブリケーション」 362 ページ

パブリッシャ

SQL Remote レプリケーションでは、レプリケートできる他のデータベースとレプリケーション・メッセージを交換できるデータベースの単一ユーザを指します。

参照：「レプリケーション」 367 ページ。

ビジネス・ルール

実世界の要求に基づくガイドラインです。通常ビジネス・ルールは、検査制約、ユーザ定義データ型、適切なトランザクションの使用により実装されます。

参照：

- 「制約」 372 ページ
- 「ユーザ定義データ型」 366 ページ

ヒストグラム

ヒストグラムは、カラム統計のもっとも重要なコンポーネントであり、データ分散を表します。SQL Anywhere は、ヒストグラムを維持して、カラムの値の分散に関する統計情報をオプティマイザに提供します。

ビット配列

ビット配列は、一連のビットを効率的に保管するのに使用される配列データ構造の種類です。ビット配列は文字列に似てますが、使用される要素は文字ではなく 0 (ゼロ) と 1 になります。ビット配列は、一般的にブール値の文字列を保持するのに使用されます。

ビュー

データベースにオブジェクトとして格納される SELECT 文です。ビューを使用すると、ユーザは 1 つまたは複数のテーブルのローやカラムのサブセットを参照できます。ユーザが特定のテーブルやテーブルの組み合わせのビューを使うたびに、テーブルに保持されているデータから再計算されます。ビューは、セキュリティの目的に有用です。またデータベース情報の表示を調整して、データへのアクセスが簡単になるようにする場合も役立ちます。

ファイルベースのダウンロード

Mobile Link では、ダウンロードがファイルとして配布されるデータの同期方法であり、同期変更のオフライン配布を可能にします。

ファイル定義データベース

Mobile Link では、ダウンロード・ファイルの作成に使用される SQL Anywhere データベースのことです。

参照：「ファイルベースのダウンロード」 363 ページ。

フェールオーバ

アクティブなサーバ、システム、またはネットワークで障害や予定外の停止が発生したときに、冗長な (スタンバイ) サーバ、システム、またはネットワークに切り替えることです。フェールオーバは自動的に発生します。

プライマリ・キー

テーブル内のすべてのローをユニークに識別する値を持つカラムまたはカラムのリストです。

参照：「外部キー」 369 ページ。

プライマリ・キー制約

プライマリ・キーのカラムに対する一意性制約です。テーブルにはプライマリ・キー制約を 1 つしか設定できません。

参照：

- 「制約」 372 ページ
- 「検査制約」 371 ページ
- 「外部キー制約」 370 ページ
- 「一意性制約」 369 ページ
- 「整合性」 372 ページ

プライマリ・テーブル

外部キー関係でプライマリ・キーを含むテーブルです。

プラグイン・モジュール

Sybase Central で、製品にアクセスしたり管理したりする方法です。プラグインは、通常、インストールすると Sybase Central にもインストールされ、自動的に登録されます。プラグインは、多くの場合、Sybase Central のメイン・ウィンドウに最上位のコンテナとして、その製品名 (たとえば SQL Anywhere) で表示されます。

参照：「Sybase Central」 351 ページ。

フル・バックアップ

データベース全体をバックアップすることです。オプションでトランザクション・ログのバックアップも可能です。フル・バックアップには、データベース内のすべての情報が含まれており、システム障害やメディア障害が発生した場合の保護として機能します。

参照：「インクリメンタル・バックアップ」 352 ページ。

プロキシ・テーブル

メタデータを含むローカル・テーブルです。リモート・データベース・サーバのテーブルに、ローカル・テーブルであるかのようにアクセスするときに使用します。

参照：「メタデータ」 365 ページ。

ベース・テーブル

データを格納する永久テーブルです。テーブルは、テンポラリ・テーブルやビューと区別するために、「ベース・テーブル」と呼ばれることがあります。

参照：

- 「テンポラリ・テーブル」 360 ページ
- 「ビュー」 363 ページ

ポーリング

Mobile Link サーバ起動同期において、Mobile Link Listerner などのライト・ウェイト・ポーラが Notifier から Push 通知を要求する方法です。

参照：「サーバ起動同期」 355 ページ。

ポリシー

QAnywhere では、メッセージ転送の発生時期を指定する方法のことです。

マテリアライズド・ビュー

計算され、ディスクに保存されたビューのことです。マテリアライズド・ビューは、ビュー (クエリ指定を使用して定義される) とテーブル (ほとんどのテーブルの操作をそのテーブル上で実行できる) の両方の特性を持ちます。

参照：

- 「ベース・テーブル」 365 ページ
- 「ビュー」 363 ページ

ミラー・ログ

参照：「トランザクション・ログ・ミラー」 361 ページ。

メタデータ

データについて説明したデータです。メタデータは、他のデータの特質と内容について記述しています。

参照：「スキーマ」 357 ページ。

メッセージ・システム

SQL Remote のレプリケーションでは、統合データベースとリモート・データベースの間でのメッセージのやりとりに使用するプロトコルのことです。SQL Anywhere では、FILE、FTP、SMTP のメッセージ・システムがサポートされています。

参照：

- 「レプリケーション」 367 ページ
- 「FILE」 346 ページ

メッセージ・ストア

QAnywhere では、メッセージを格納するクライアントおよびサーバ・デバイスのデータベースのことです。

参照：

- 「クライアント・メッセージ・ストア」 354 ページ
- 「サーバ・メッセージ・ストア」 355 ページ

メッセージ・タイプ

SQL Remote のレプリケーションでは、リモート・ユーザと統合データベースのパブリッシャとの通信方法を指定するデータベース・オブジェクトのことを指します。統合データベースには、複数のメッセージ・タイプが定義されていることがあります。これによって、リモート・ユーザはさまざまなメッセージ・システムを使って統合データベースと通信できることになります。

参照：

- 「レプリケーション」 367 ページ
- 「統合データベース」 374 ページ

メッセージ・ログ

データベース・サーバや Mobile Link サーバなどのアプリケーションからのメッセージを格納できるログです。この情報は、メッセージ・ウィンドウに表示されたり、ファイルに記録されたりすることもあります。メッセージ・ログには、情報メッセージ、エラー、警告、MESSAGE 文からのメッセージが含まれます。

メンテナンス・リリース

メンテナンス・リリースは、同じメジャー・バージョン番号を持つ旧バージョンのインストール済みソフトウェアをアップグレードするための完全なソフトウェア・セットです (バージョン番号のフォーマットは、*メジャー.マイナー.パッチ.ビルド*です)。バグ・フィックスとその他の変更については、アップグレードのリリース・ノートにリストされます。

ユーザ定義データ型

参照：「ドメイン」 360 ページ。

ライト・ウェイト・ポーラ

Mobile Link サーバ起動同期において、Mobile Link サーバからの Push 通知をポーリングするデバイス・アプリケーションです。

参照：「サーバ起動同期」 355 ページ。

リダイレクタ

クライアントと Mobile Link サーバ間で要求と応答をルート指定する Web サーバ・プラグインです。このプラグインによって、負荷分散メカニズムとフェールオーバ・メカニズムも実装されます。

リファレンス・データベース

Mobile Link では、Ultra Light クライアントの開発に使用される SQL Anywhere データベースです。開発中は、1 つの SQL Anywhere データベースをリファレンス・データベースとしても統合データベースとしても使用できます。他の製品によって作成されたデータベースは、リファレンス・データベースとして使用できません。

リモート ID

SQL Anywhere と Ultra Light データベース内のユニークな識別子で、Mobile Link によって使用されます。リモート ID は NULL に初期設定されていますが、データベースの最初の同期時に GUID に設定されます。

リモート・データベース

Mobile Link または SQL Remote では、統合データベースとデータを交換するデータベースを指します。リモート・データベースは、統合データベース内のすべてまたは一部のデータを共有できます。

参照：

- 「同期」 374 ページ
- 「統合データベース」 374 ページ

リレーショナル・データベース管理システム (RDBMS)

関連するテーブルの形式でデータを格納するデータベース管理システムです。

参照：「データベース管理システム (DBMS)」 359 ページ。

レプリケーション

物理的に異なるデータベース間でデータを共有することです。Sybase では、Mobile Link、SQL Remote、Replication Server の 3 種類のレプリケーション・テクノロジを提供しています。

レプリケーション・メッセージ

SQL Remote または Replication Server では、パブリッシュするデータベースとサブスクリプションを作成するデータベース間で送信される通信内容を指します。メッセージにはデータを含み、レプリケーション・システムで必要なパススルー文、情報があります。

参照：

- 「レプリケーション」 367 ページ
- 「パブリケーションの更新」 362 ページ

レプリケーションの頻度

SQL Remote レプリケーションでは、リモート・ユーザに対する設定の 1 つで、パブリッシャの Message Agent がレプリケーション・メッセージを他のリモート・ユーザに送信する頻度を定義します。

参照：「レプリケーション」 367 ページ。

ロー・レベルのトリガ

変更されているローごとに一回実行するトリガです。

参照：

- 「トリガ」 361 ページ
- 「文レベルのトリガ」 375 ページ

ローカル・テンポラリ・テーブル

複合文を実行する間だけ存在したり、接続が終了するまで存在したりするテンポラリ・テーブルです。データのセットを 1 回だけロードする必要がある場合にローカル・テンポラリ・テーブルが便利です。デフォルトでは、COMMIT を実行するとローが削除されます。

参照：

- 「テンポラリ・テーブル」 360 ページ
- 「グローバル・テンポラリ・テーブル」 354 ページ

ロール

概念データベース・モデルで、ある視点からの関係を説明する動詞またはフレーズを指します。各関係は 2 つのロールを使用して表すことができます。"contains (A は B を含む)" や "is a member of (B は A のメンバ)" などのロールがあります。

ロールバック・ログ

コミットされていない各トランザクションの最中に行われた変更のレコードです。ROLLBACK 要求やシステム障害が発生した場合、コミットされていないトランザクションはデータベースから破棄され、データベースは前の状態に戻ります。各トランザクションにはそれぞれロールバック・ログが作成されます。このログは、トランザクションが完了すると削除されます。

参照：「トランザクション」 361 ページ。

ロール名

外部キーの名前です。この外部キーがロール名と呼ばれるのは、外部テーブルとプライマリ・テーブル間の関係に名前を指定するためです。デフォルトでは、テーブル名がロール名になります。ただし、別の外部キーがそのテーブル名を使用している場合、デフォルトのロール名はテーブル名に 3 桁のユニークな数字を付けたものになります。ロール名は独自に作成することもできます。

参照：「外部キー」 369 ページ。

ログ・ファイル

SQL Anywhere によって管理されているトランザクションのログです。ログ・ファイルを使用すると、システム障害やメディア障害が発生してもデータベースを回復させることができます。また、データベースのパフォーマンスを向上させたり、SQL Remote を使用してデータをレプリケートしたりする場合にも使用できます。

参照：

- 「トランザクション・ログ」 361 ページ
- 「トランザクション・ログ・ミラー」 361 ページ
- 「フル・バックアップ」 364 ページ

ロック

複数のトランザクションを同時に実行しているときにデータの整合性を保護する同時制御メカニズムです。SQL Anywhere では、2 つの接続によって同じデータが同時に変更されないようにするために、また変更処理の最中に他の接続によってデータが読み込まれないようにするために、自動的にロックが適用されます。

ロックの制御は、独立性レベルを設定して行います。

参照：

- 「独立性レベル」 375 ページ
- 「同時性 (同時実行性)」 375 ページ
- 「整合性」 372 ページ

ワーク・テーブル

クエリの最適化の最中に中間結果を保管する内部保管領域です。

一意性制約

NULL 以外のすべての値が重複しないことを要求するカラムまたはカラムのセットに対する制限です。テーブルには複数の一意性制約を指定できます。

参照：

- 「外部キー制約」 370 ページ
- 「プライマリ・キー制約」 364 ページ
- 「制約」 372 ページ

解析ツリー

クエリを代数で表現したものです。

外部キー

別のテーブルにあるプライマリ・キーの値を複製する、テーブルの 1 つ以上のカラムです。テーブル間の関係は、外部キーによって確立されます。

参照：

- 「プライマリ・キー」 364 ページ
- 「外部テーブル」 370 ページ

外部キー制約

カラムまたはカラムのセットに対する制約で、テーブルのデータが別のテーブルのデータとどのように関係しているかを指定するものです。カラムのセットに外部キー制約を加えると、それらのカラムが外部キーになります。

参照：

- 「制約」 372 ページ
- 「検査制約」 371 ページ
- 「プライマリ・キー制約」 364 ページ
- 「一意性制約」 369 ページ

外部ジョイン

テーブル内のすべてのローを保護するジョインです。SQL Anywhere では、左外部ジョイン、右外部ジョイン、全外部ジョインがサポートされています。左外部ジョインは JOIN 演算子の左側にあるテーブルのローを保護し、右側にあるテーブルのローがジョイン条件を満たさない場合には NULL を返します。全外部ジョインは両方のテーブルに含まれるすべてのローを保護します。

参照：

- 「ジョイン」 356 ページ
- 「内部ジョイン」 375 ページ

外部テーブル

外部キーを持つテーブルです。

参照：「外部キー」 369 ページ。

外部ログイン

リモート・サーバとの通信に使用される代替のログイン名とパスワードです。デフォルトでは、SQL Anywhere は、クライアントに代わってリモート・サーバに接続するときは、常にそのクライアントの名前とパスワードを使用します。外部ログインを作成することによって、このデフォルトを上書きできます。外部ログインは、リモート・サーバと通信するときに使用する代替のログイン名とパスワードです。

競合

リソースについて対立する動作のことです。たとえば、データベース用語では、複数のユーザがデータベースの同じローを編集しようとした場合、そのローの編集権についての競合が発生します。

競合解決

Mobile Link では、競合解決は 2 人のユーザが別々のリモート・データベースの同じローを変更した場合にどう処理するかを指定するロジックのことです。

検査制約

指定された条件をカラムやカラムのセットに課す制約です。

参照：

- 「制約」 372 ページ
- 「外部キー制約」 370 ページ
- 「プライマリ・キー制約」 364 ページ
- 「一意性制約」 369 ページ

検証

データベース、テーブル、またはインデックスについて、特定のタイプのファイル破損をテストすることです。

作成者 ID

Ultra Light の Palm OS アプリケーションでは、アプリケーションが作成されたときに割り当てられる ID のことです。

参照元オブジェクト

テーブルなどのデータベースの別のオブジェクトをオブジェクト定義が直接参照する、ビューなどのオブジェクトです。

参照：「外部キー」 369 ページ。

参照整合性

データの整合性、特に異なるテーブルのプライマリ・キー値と外部キー値との関係を管理する規則を厳守することです。参照整合性を備えるには、それぞれの外部キーの値が、参照テーブルにあるローのプライマリ・キー値に対応するようにします。

参照：

- 「プライマリ・キー」 364 ページ
- 「外部キー」 369 ページ

参照先オブジェクト

ビューなどの別のオブジェクトの定義で直接参照される、テーブルなどのオブジェクトです。

参照：「プライマリ・キー」 364 ページ。

識別子

テーブルやカラムなどのデータベース・オブジェクトを参照するときに使う文字列です。A ～ Z、a ～ z、0 ～ 9、アンダースコア (_)、アットマーク (@)、シャープ記号 (#)、ドル記号 ($) のうち、任意の文字を識別子として使用できます。

述部

条件式です。オプションで論理演算子 AND や OR と組み合わせて、WHERE 句または HAVING 句に条件のセットを作成します。SQL では、unknown と評価される述部が false と解釈されます。

照合

データベース内のテキストのプロパティを定義する文字セットとソート順の組み合わせのことです。SQL Anywhere データベースでは、サーバを実行しているオペレーティング・システムと言語によって、デフォルトの照合が決まります。たとえば、英語版 Windows システムのデフォルトの照合は 1252LATIN1 です。照合は、照合順とも呼ばれ、文字列の比較とソートに使用します。

参照：

- 「文字セット」 375 ページ
- 「コード・ページ」 355 ページ
- 「エンコード」 353 ページ

世代番号

Mobile Link では、リモート・データベースがデータをアップロードしてからダウンロード・ファイルを適用するようにするためのメカニズムのことです。

参照：「ファイルベースのダウンロード」 363 ページ。

制約

テーブルやカラムなど、特定のデータベース・オブジェクトに含まれた値に関する制約です。たとえば、一意性制約があるカラム内の値は、すべて異なっている必要があります。テーブルに、そのテーブルの情報と他のテーブルのデータがどのように関係しているのかを指定する外部キー制約が設定されていることもあります。

参照：

- 「検査制約」 371 ページ
- 「外部キー制約」 370 ページ
- 「プライマリ・キー制約」 364 ページ
- 「一意性制約」 369 ページ

整合性

データが適切かつ正確であり、データベースの関係構造が保たれていることを保証する規則を厳守することです。

参照：「参照整合性」 371 ページ。

正規化

データベース・スキーマを改善することです。リレーショナル・データベース理論に基づく規則に従って、冗長性を排除したり、編成を改良します。

正規表現

正規表現は、文字列内で検索するパターンを定義する、一連の文字、ワイルドカード、演算子です。

生成されたジョイン条件

自動的に生成される、ジョインの結果に対する制限です。キーとナチュラルの 2 種類があります。キー・ジョインは、KEY JOIN を指定したとき、またはキーワード JOIN を指定したが、CROSS、NATURAL、または ON を使用しなかった場合に生成されます。キー・ジョインの場合、生成されたジョイン条件はテーブル間の外部キー関係に基づいています。ナチュラル・ジョインは NATURAL JOIN を指定したときに生成され、生成されたジョイン条件は、2 つのテーブルの共通のカラム名に基づきます。

参照：

- 「ジョイン」 356 ページ
- 「ジョイン条件」 357 ページ

接続 ID

クライアント・アプリケーションとデータベース間の特定の接続に付けられるユニークな識別番号です。現在の接続 ID を確認するには、次の SQL 文を使用します。

```
SELECT CONNECTION_PROPERTY( 'Number' );
```

接続プロファイル

ユーザ名、パスワード、サーバ名などの、データベースに接続するために必要なパラメータのセットです。便宜的に保管され使用されます。

接続起動同期

Mobile Link のサーバ起動同期の 1 つの形式で、接続が変更されたときに同期が開始されます。

参照：「サーバ起動同期」 355 ページ。

相関名

クエリの FROM 句内で使用されるテーブルやビューの名前です。テーブルやビューの元の名前か、FROM 句で定義した代替名のいずれかになります。

抽出

SQL Remote レプリケーションでは、統合データベースから適切な構造とデータをアンロードする動作を指します。この情報は、リモート・データベースを初期化するときに使用されます。

参照：「レプリケーション」 367 ページ。

通信ストリーム

Mobile Link では、Mobile Link クライアントと Mobile Link サーバ間での通信にネットワーク・プロトコルが使用されます。

転送ルール

QAnywhere では、メッセージの転送を発生させる時期、転送するメッセージ、メッセージを削除する時期を決定する論理のことです。

統合データベース

分散データベース環境で、データのマスタ・コピーを格納するデータベースです。競合や不一致が発生した場合、データのプライマリ・コピーは統合データベースにあるとみなされます。

参照：

- 「同期」 374 ページ
- 「レプリケーション」 367 ページ

統合化ログイン

オペレーティング・システムへのログイン、ネットワークへのログイン、データベースへの接続に、同一のユーザ ID とパスワードを使用するログイン機能の 1 つです。

動的 SQL

実行される前に作成したプログラムによって生成される SQL です。Ultra Light の動的 SQL は、占有容量の小さいデバイス用に設計された変形型です。

同期

Mobile Link テクノロジを使用してデータベース間でデータをレプリケートする処理です。

SQL Remote では、同期はデータの初期セットを使ってリモート・データベースを初期化する処理を表すために特に使用されます。

参照：

- 「Mobile Link」 347 ページ
- 「SQL Remote」 350 ページ

同時性 (同時実行性)

互いに独立し、場合によっては競合する可能性のある 2 つ以上の処理を同時に実行することです。SQL Anywhere では、自動的にロックを使用して各トランザクションを独立させ、同時に稼働するそれぞれのアプリケーションが一貫したデータのセットを参照できるようにします。

参照：

- 「トランザクション」 361 ページ
- 「独立性レベル」 375 ページ

独立性レベル

あるトランザクションの操作が、同時に処理されている別のトランザクションの操作からどの程度参照できるかを示します。独立性レベルには 0 から 3 までの 4 つのレベルがあります。最も高い独立性レベルには 3 が設定されます。デフォルトでは、レベルは 0 に設定されています。SQL Anywhere では、スナップショット、文のスナップショット、読み込み専用文のスナップショットの 3 つのスナップショットの独立性レベルがあります。

参照：「スナップショット・アイソレーション」 357 ページ。

内部ジョイン

2 つのテーブルがジョイン条件を満たす場合だけ、結果セットにローが表示されるジョインです。内部ジョインがデフォルトです。

参照：

- 「ジョイン」 356 ページ
- 「外部ジョイン」 370 ページ

物理インデックス

インデックスがディスクに保存されるときの実際のインデックス構造です。

文レベルのトリガ

トリガ付きの文の処理が完了した後に実行されるトリガです。

参照：

- 「トリガ」 361 ページ
- 「ロー・レベルのトリガ」 368 ページ

文字セット

文字セットは記号、文字、数字、スペースなどから成ります。"ISO-8859-1" は文字セットの例です。Latin1 とも呼ばれます。

参照：

- 「コード・ページ」 355 ページ
- 「エンコード」 353 ページ
- 「照合」 372 ページ

文字列リテラル

文字列リテラルとは、一重引用符 (') で囲まれ、シーケンスで並べられた文字のことです。

論理インデックス

物理インデックスへの参照 (ポインタ) です。ディスクに保存される論理インデックス用のインデックス構造はありません。

索引

記号

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

あ

い

う

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

つ

て

と

な

に

ね

は

ひ

ふ

へ

ほ

ま

み

め

も

ゆ

よ

ら

り

る

れ

ろ

わ