SYBASE iAnywhere

Ultra Light
C/C++ プログラミング

改訂 2007 年 3 月

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 と 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 と 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

このマニュアルは、SQL Anywhere のマニュアル・セットの一部です。この項では、マニュアル・セットに含まれる各マニュアルと使用法について説明します。

SQL Anywhere のマニュアル

SQL Anywhere の完全なマニュアルは、各マニュアルをまとめたオンライン形式とマニュアル別の PDF ファイルで提供されます。いずれの形式のマニュアルも、同じ情報が含まれ、次のマニュアルから構成されます。

- **『SQL Anywhere 10 - 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 10 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 10 - 変更点とアップグレード』** このマニュアルでは、SQL Anywhere 10 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 データベースの実行、管理、設定について説明します。管理ユーティリティとオプションのほか、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について説明します。
- **『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。
- **『SQL Anywhere サーバ - SQL リファレンス』** このマニュアルは、SQL Anywhere で使用する SQL 言語の完全なリファレンスです。また、SQL Anywhere のシステム・ビューとシステム・プロシージャについても説明しています。
- **『SQL Anywhere サーバ - プログラミング』** このマニュアルでは、C、C++、Java プログラミング言語、Visual Studio .NET 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。Visual Basic や PowerBuilder などのツールのユーザは、それらのツールのプログラミング・インタフェースを使用できます。
- **『SQL Anywhere 10 - 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、その診断情報とともに説明します。
- **『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 - 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 - 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。
- **『Mobile Link - サーバ起動同期』** このマニュアルでは、Mobile Link のサーバによって開始される同期について説明します。サーバによって開始される同期とは、統合データベースから同期またはその他のリモート・アクションの開始を可能にする Mobile Link の機能です。
- **『QAnywhere』** このマニュアルでは QAnywhere について説明します。QAnywhere は、従来のデスクトップ・クライアントやラップトップ・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ほか、モバイル・クライアントや無線クライアント用のメッセージング・プラットフォームでもあります。
- **『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。
- **『SQL Anywhere 10 - コンテキスト別ヘルプ』** このマニュアルには、[接続] ダイアログ、クエリ・エディタ、Mobile Link モニタ、SQL Anywhere コンソール・ユーティリティ、インデックス・コンサルタント、Interactive SQL のコンテキスト別のヘルプが収録されています。
- **『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 - AppForge プログラミング』** このマニュアルでは、Ultra Light for AppForge について説明します。Ultra Light for AppForge を使用すると、Palm OS、Symbian OS、または Windows CE を搭載しているハンドヘルド、モバイル、または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、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、モバイル、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 CE、または Windows XP を搭載しているハンドヘルド、モバイル、または埋め込みデバイスに対して Web ベースのデータベース・アプリケーションを開発、配備できます。
- **『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、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、配備できます。

マニュアルの形式

SQL Anywhere のマニュアルは、次の形式で提供されています。

- **オンライン・マニュアル** オンライン・マニュアルには、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。オンライン・マニュアルは、製品のメンテナンス・リリースごとに更新されます。これは、最新の情報を含む最も完全なマニュアルです。

Windows オペレーティング・システムでオンライン・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 10] − [オンライン・マニュアル] を選択します。オンライン・マニュアルをナビゲートするには、左ウィンドウ枠で HTML ヘルプの目次、索引、検索機能を使用し、右ウィンドウ枠でリンク情報とメニューを使用します。

UNIX オペレーティング・システムでオンライン・マニュアルにアクセスするには、SQL Anywhere のインストール・ディレクトリまたはインストール CD に保存されている HTML マニュアルを参照してください。

- **PDF ファイル** SQL Anywhere の完全なマニュアル・セットは、Adobe Reader で表示できる Adobe Portable Document Format (pdf) 形式のファイルとして提供されています。

  Windows では、PDF 形式のマニュアルはオンライン・マニュアルの各ページ上部にある PDF のリンクから、または Windows の [スタート] メニュー ([スタート] − [プログラム] − [SQL Anywhere 10] − [オンライン・マニュアル - PDF フォーマット]) からアクセスできます。

  UNIX では、PDF 形式のマニュアルはインストール CD にあります。

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書体およびグラフィック表現の規則について説明します。

SQL 構文の表記規則

SQL 構文の表記には、次の規則が適用されます。

- **キーワード**　SQL キーワードはすべて次の例に示す ALTER TABLE のように大文字で表記します。

  ALTER TABLE [*owner.*]*table-name*

- **プレースホルダ**　適切な識別子または式で置き換えられる項目は、次の例に示す *owner* や *table-name* のように表記します。

  ALTER TABLE [*owner.*]*table-name*

- **繰り返し項目**　繰り返し項目のリストは、次の例に示す *column-constraint* のように、リストの要素の後ろに省略記号 (ピリオド 3 つ …) を付けて表します。

  ADD *column-definition* [*column-constraint*, …]

  複数の要素を指定できます。複数の要素を指定する場合は、各要素間をカンマで区切る必要があります。

- **オプション部分**　文のオプション部分は角カッコで囲みます。

  RELEASE SAVEPOINT [*savepoint-name*]

  この例では、角カッコで囲まれた *savepoint-name* がオプション部分です。角カッコは入力しないでください。

- **オプション**　項目リストから 1 つだけ選択する場合や、何も選択しなくてもよい場合は、項目間を縦線で区切り、リスト全体を角カッコで囲みます。

  [**ASC** | **DESC**]

  この例では、ASC と DESC のどちらか 1 つを選択しても、選択しなくてもかまいません。角カッコは入力しないでください。

- **選択肢**　オプションの中の 1 つを必ず選択しなければならない場合は、選択肢を中カッコで囲み、縦棒で区切ります。

  [**QUOTES** { **ON** | **OFF** }]

  QUOTES オプションを使用する場合は、ON または OFF のどちらかを選択する必要があります。角カッコと中カッコは入力しないでください。

オペレーティング・システムの表記規則

- **Windows**　デスクトップおよびラップトップ・コンピュータ用の 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 ファミリには Windows Vista や Windows XP も含まれます。
- **Windows CE**　Microsoft Windows CE モジュラ・オペレーティング・システムに基づいて構築されたプラットフォームです。Windows Mobile や Windows Embedded CE などのプラットフォームが含まれます。

  Windows Mobile は Windows CE 上に構築されています。これにより、Windows のユーザ・インタフェースや、Word や Excel といったアプリケーションの小規模バージョンなどの追加機能が実現されています。Windows Mobile は、モバイル・デバイスで最も広く使用されています。

  SQL Anywhere の制限事項や相違点は、基盤となっているオペレーティング・システム (Windows CE) に由来しており、使用しているプラットフォーム (Windows Mobile など) に依存していることはほとんどありません。
- **UNIX**　特に記述がないかぎり、UNIX は Linux プラットフォームと UNIX プラットフォームの両方のことです。

ファイルの命名規則

マニュアルでは、パス名やファイル名などのオペレーティング・システムに依存するタスクと機能を表すときは、通常 Windows の表記規則が使用されます。ほとんどの場合、他のオペレーティング・システムで使用される構文に簡単に変換できます。

- **ディレクトリ名とパス名**　マニュアルでは、ドライブを示すコロンや、ディレクトリの区切り文字として使用する円記号など、Windows の表記規則を使用して、ディレクトリ・パスのリストを示します。次に例を示します。

  ```
  MobiLink¥redirector
  ```

  UNIX、Linux、Mac OS X では、代わりにスラッシュを使用してください。次に例を示します。

  ```
  MobiLink/redirector
  ```

  SQL Anywhere がマルチプラットフォーム環境で使用されている場合、プラットフォーム間でのパス名の違いに注意する必要があります。
- **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、拡張子 *.exe* が付きます。UNIX、Linux、Mac OS X では、実行ファイルの名前には拡張子は付きません。NetWare では、実行ファイルの名前には、拡張子 *.nlm* が付きます。

  たとえば、Windows では、ネットワーク・データベース・サーバは *dbsrv10.exe* です。UNIX、Linux、Mac OS X では、*dbsrv10* になります。NetWare では、*dbsrv10.nlm* になります。
- **install-dir**　インストール・プロセスでは、SQL Anywhere をインストールするロケーションを選択できます。マニュアルでは、このロケーションは *install-dir* という表記で示されます。

インストールが完了すると、環境変数 SQLANY10 によって SQL Anywhere コンポーネントがあるインストール・ディレクトリのロケーション (*install-dir*) が指定されます。SQLANYSH10 は、SQL Anywhere が他の Sybase アプリケーションと共有しているコンポーネントがあるディレクトリのロケーションを指定します。

オペレーティング・システム別の *install-dir* のデフォルト・ロケーションの詳細については、「SQLANY10 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・プロセスでは、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できます。マニュアルでは、このロケーションは *samples-dir* という表記で示されます。

  インストールが完了すると、環境変数 SQLANYSAMP10 によってサンプルがあるディレクトリのロケーション (*samples-dir*) が指定されます。Windows の [スタート] メニューから、[プログラム] – [SQL Anywhere 10] – [サンプル・アプリケーションおよびプロジェクト] を選択すると、このディレクトリで [Windows エクスプローラ] ウィンドウが表示されます。

  オペレーティング・システム別の *samples-dir* のデフォルト・ロケーションの詳細については、「サンプル・ディレクトリ」『SQL Anywhere サーバ - データベース管理』を参照してください。

- **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 では、環境変数を参照するのに、構文 *%envvar%* が使用されます。UNIX、Linux、Mac OS X では、環境変数を参照するのに、構文 *$envvar* または *${envvar}* が使用されます。

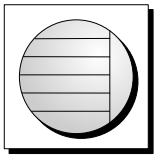
  UNIX、Linux、Mac OS X 環境変数は、*.cshrc* や *.tcshrc* などのシェルとログイン・スタートアップ・ファイルに格納されます。

グラフィック・アイコン

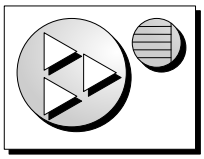
このマニュアルでは、次のアイコンを使用します。

- クライアント・アプリケーション

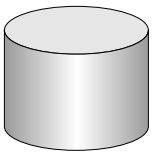
- SQL Anywhere などのデータベース・サーバ



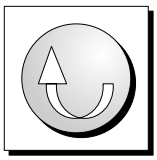
- Ultra Light アプリケーション



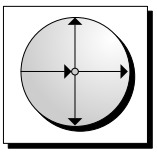
- データベース。高度な図では、データベースとデータベースを管理するデータ・サーバの両方をこのアイコンで表します。



- 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。



- Sybase Replication Server



- プログラミング・インタフェース

詳細情報の検索／フィードバックの提供

詳細情報の検索

詳しい情報やリソース (コード交換など) については、iAnywhere Developer Network (http://www.ianywhere.com/developer/) 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。バージョン情報は、コマンド・プロンプトで **dbeng10 -v** と入力して確認できます。

ニュースグループは、ニュース・サーバ *forums.sybase.com* にあります (ニュースグループにおけるサービスは英語でのみの提供となります)。以下のニュースグループがあります。

- sybase.public.sqlanywhere.general
- sybase.public.sqlanywhere.linux
- sybase.public.sqlanywhere.mobilink
- sybase.public.sqlanywhere.product_futures_discussion
- sybase.public.sqlanywhere.replication
- sybase.public.sqlanywhere.ultralite
- ianywhere.public.sqlanywhere.qanywhere

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。

フィードバック

このマニュアルに関するご意見、ご提案、フィードバックをお寄せください。

マニュアルに関するご意見、ご提案は、SQL Anywhere ドキュメンテーション・チームの iasdoc@ianywhere.com 宛てに電子メールでお寄せください。このアドレスに送信された電子メールに返信はいたしませんが、お寄せいただいたご意見、ご提案は必ず読ませていただきます。

マニュアルまたはソフトウェアについてのフィードバックは、上記のニュースグループを通してお寄せいただいてもかまいません。

パート I. 概要

パート I では、C/C++ プログラマを対象に Ultra Light の概要について説明します。C/C++ 言語インタフェースは、Palm OS、Windows CE、Symbian OS、Windows デスクトップの各プラットフォーム用 Ultra Light アプリケーションを作成するために使用できます。

第 1 章

C/C++ 開発者用 Ultra Light の概要

目次

Ultra Light と C/C++ プログラミング言語

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。

- 専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 CE、Symbian OS、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」 71 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 CE 3.0 以降
- Palm OS 4.0 以降
- Symbian OS 7.0、8.0、またはそれ以降

サポートされているターゲット・プラットフォームの詳細については、「SQL Anywhere 用 Ultra Light 展開オプション」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、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」 183 ページを参照してください。

パート II. アプリケーション開発

パート II 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。

第 2 章

Ultra Light C/C++ インタフェースの共通機能

目次

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。

第 3 章

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」 29 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) 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」 32 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」 34 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」 23 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」 19 ページと「UltraLite_PreparedStatement クラス」 252 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」 238 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」 239 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」 161 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」 23 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 文のリファレンス」 『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」 252 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、準備文での並び順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」 223 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」 253 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。

   以下のコードでは、SELECT クエリの結果に文字列が含まれています。これはコンソールへ出力されます。

   ```
   #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」 258 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、スキーマ情報をコンソール・ウィンドウ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」 220 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 文を実行してデータにアクセスする詳細については、「SQL を使用したデータへのアクセス」 19 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」 273 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」 273 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」 273 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」 29 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 入力可能なカラムの場合は 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」 217 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」 224 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と独立性レベル」『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」 220 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」 220 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」 273 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、「エラー・メッセージ (Sybase エラー・コード順)」『SQL Anywhere 10 - 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」 113 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」 155 ページと 「ULRegisterErrorCallback 関数」 178 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、「エラー・メッセージ (Sybase エラー・コード順)」『SQL Anywhere 10 - 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」 221 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」 224 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したり、難読化したりすることを選択できます。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 で使用する作成時のデータベース・プロパティの選択」『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

暗号化を使用して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 でのセキュリティの考慮事項」『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」『Mobile Link - クライアント管理』を参照してください。

同期に使用する ul_sync_info 構造体の詳細については、使用文字が ASCII かワイドかに応じて「ul_synch_info_a 構造体」 185 ページまたは「ul_synch_info_w2 構造体」 192 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」『Mobile Link - クライアント管理』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。このライブラリには、Windows CE 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* 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。

- **ulrt.lib**　静的ライブラリ。このライブラリは、*install-dir¥ultralite¥palm¥68k¥lib¥cw* にあります。
- **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。UltraLite の機能にアクセスできるように、C/C++ アプリケーションでこのライブラリに対してリンクしてください。

Windows CE のランタイム・ライブラリ

Windows CE ライブラリは *install-dir¥ultralite¥ce¥platform* ディレクトリにあります。*platform* は、*arm*、*386*、*arm.50*、*armt* のいずれかです。

Windows CE 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。
- **ulrt10.dll**　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 *install-dir¥ultralite¥ce¥platform¥lib¥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・オプションを必ず指定してください。

  ```
  /DUNICODE /DUL_USE_DLL
  ```

- **ulrt.lib**　このライブラリは、*install-dir¥ultralite¥ce¥platform¥lib¥* にあります。

  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
  ```

- **ulrtc.lib**　ユニコード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。このライブラリは、*install-dir¥ultralite¥ce¥platform¥lib¥* にあります。

  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 32 ビット・デスクトップのランタイム・ライブラリ

install-dir¥ultralite¥win32¥386 ディレクトリには、Windows CE 以外のサポートされている Windows 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。含ま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。
- **ulrt10.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 *install-dir¥ultralite¥win32¥386¥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
  ```

第 4 章

Embedded SQL を使用したアプリケーションの開発

目次

Embedded SQL 開発の概要

この章では、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、「C/C++ 開発者用 Ultra Light の概要」 3 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」 307 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 という名前の Embedded SQL「**ライブラリ関数**」を呼び出さなければなりません。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」 311 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。

SQLDA 型はヘッダ・ファイル *install-dir¥h¥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¥h¥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、「データベース・エラー・メッセージ」 『SQL Anywhere 10 - 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 の接続文字列パラメータのリファレンス」 『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」 311 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・ファイルで定義したマクロは、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 l;
  unsigned long int l;
  ```

- 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 データ型にマップされます。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 型にマップされます。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 CE 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¥h¥sqlca.h* ファイルに定義されています。VARCHAR 型、BINARY 型、TYPE_DECIMAL 型は、データ保管領域が長さ 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が、動的な変数の割り付けや他の変数を何度も割り当てるのには有用です。

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、*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、機能はするものの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。

次のコードでは、SQL プリプロセッサから *host_value* 式を隠す方法が示されています。

```
#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状態で、次のように *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。したがって、NULL 値をデータベースに渡したり、結果に NULL を受け取ったりするためには、通常のホスト変数の他に何かが必要です。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」 41 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」 41 ページを参照してください。

例

たとえば、次のコードは従業員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 ) {
```



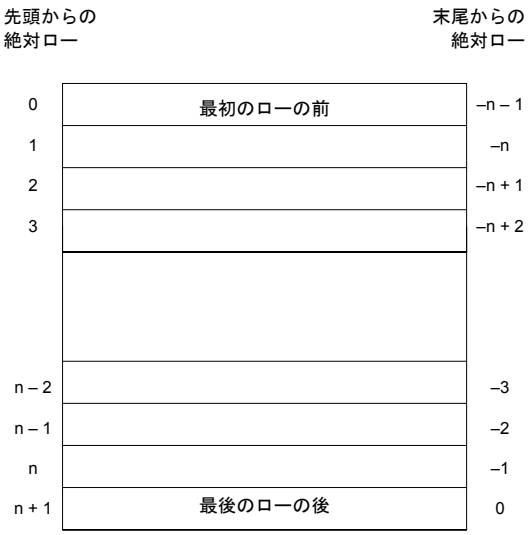
```
            break; /* the FETCH caused an error */
        }
        if( ind_birthdate < 0 ) {
            strcpy( birthdate, "UNKNOWN" );
        }
        printf( "Name: %s Sex: %c Birthdate:
                %s¥n",name, sex, birthdate );
    }
    /* 4. Close the cursor. */
    EXEC SQL CLOSE C1;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

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」 325 ページ
- 「ULRevokeConnectFrom 関数」 332 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」 62 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」 314 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 で使用する作成時のデータベース・プロパティの選択」 『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」 64 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」 『Mobile Link - クライアント管理』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」 65 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_sync_infor 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 かワイドかに応じて、「ul_synch_info_a 構造体」 185 ページまたは「ul_synch_info_w2 構造体」 192 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」 『Mobile Link - クライアント管理』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¥h¥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。

```
auto 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・マシン test.internal で実行されており、ポート 2439 を使用しています。

```
auto 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) と同じマシンで実行され、デフォルトのポート (2439) を使用しています。

```
auto 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」 338 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 データベースを同期しようとした時点で、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」 『Mobile Link - クライアント管理』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。

パフォーマンスのヒント
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_id         = %d¥t(ss_stream_id)¥n"
            "    stream_context    = %d¥t(ss_stream_context)¥n"
            "    stream_error_code = %ld¥t(ss_error_code)¥n"
            "    error_string      = ¥"%s¥"¥n"
            "    system_error_code = %ld¥n",
    (int)info.stream_error.stream_id,
    (int)info.stream_error.stream_context,
    (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する。
- 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が、**ul_synch_status** 構造体をパラメータとしてとります。

ul_synch_status 構造体には次のようなメンバが含まれます。

```
ul_synch_status   state;
ul_u_short        tableCount;
ul_u_short        tableIndex;
  struct {
    ul_u_long   bytes;
    ul_u_short  inserts;
    ul_u_short  updates;
    ul_u_short  deletes;
  }       sent;
  struct {
    ul_u_long   bytes;
    ul_u_short  inserts;
    ul_u_short  updates;
    ul_u_short  deletes;
  }       received;
p_ul_synch_info   info;
ul_bool           stop;
```

- **state**　以下のステータスのいずれかを表します。
  - **UL_SYNCH_STATE_STARTING**　同期アクションはまだ開始されていません。
  - **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**　同期ストリームはすでに開かれており、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**　アップロードが完了したという確認が受信されています。
  - **UL_SYNCH_STATE_RECEIVING_TABLE**　テーブルが受信されています。
  - **UL_SYNCH_STATE_SENDING_DATA**　スキーマ情報またはデータが受信されています。
  - **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**　ダウンロードが完了したという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。

    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。

- **tableCount**　同期中のテーブルの数を返します。テーブルごとに送信と受信のフェーズがあります。したがって、この数は同期されるテーブルの数より多い場合があります。
- **tableIndex**　アップロード処理またはダウンロード処理される現在のテーブルは 0 から開始されます。この数値は、すべてのテーブルが同期されるのではない場合には、値を省略することがあります。
- **info**　ul_synch_info 構造体へのポインタ。
- **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。
- **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。
- **getUserData**　ユーザ・データ・オブジェクトを返します。
- **getStatement**　同期を呼び出した文を返します。文は Ultra Light 内部の文であり、このメソッドを実際に使用することはほとんどありませんが、完了のために含まれています。
- **getErrorCode**　同期ステータスが ERROR に設定されているときに、診断エラー・コードを返すメソッドです。
- **isOKToContinue**　cancelSynchronization が呼び出されると、false に設定されます。それ以外の場合は true です。

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 )
{
    printf( "UL_SYNCH_STATE is %d: ",
          status->state );
    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;
...
```

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。

```
UL_SYNCH_STATE is 0: Starting
UL_SYNCH_STATE is 1: Connecting
UL_SYNCH_STATE is 2: Sending Header
UL_SYNCH_STATE is 3: Sending Table 1 of 2
UL_SYNCH_STATE is 3: Sending Table 2 of 2
UL_SYNCH_STATE is 4: Receiving Upload Ack
UL_SYNCH_STATE is 5: Receiving Table 1 of 2
UL_SYNCH_STATE is 5: Receiving Table 2 of 2
UL_SYNCH_STATE is 6: Sending Download Ack
UL_SYNCH_STATE is 7: Disconnecting
UL_SYNCH_STATE is 8: 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ダイアログが表示されます。ユーザはそのダイアログ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保管されています。

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* ディレクトリに保存されています。

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 承認の暗号化には、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 10 - 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 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・ファイルのサフィックスを *.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。

Metrowerks CodeWarrior 用の Ultra Light プラグインは、Palm Computing Platform の開発者に、ここで説明するテクニックを自動的に提供します。このプラグインの詳細については、「Metrowerks CodeWarrior で Ultra Light アプリケーションを開発する」 77 ページを参照してください。

非常に単純なプロジェクトを説明したチュートリアルについては、「チュートリアル : Embedded SQL を使用したアプリケーションの構築」 127 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。

   開発ツールには、「**開発プロジェクト**」が定義されています。

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)。

   ```
   "%SQLANY10%¥win32¥sqlpp.exe" -q -u $(InputPath) $(InputName).cpp
   ```

   SQLANY10 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、*$(SQLANY10)¥h* と、インクルード・パスに必要なインクルード・フォルダとを、カンマで区切ったリストとして追加します。

第 5 章

Palm OS の Ultra Light アプリケーションの開発

目次

Palm OS 開発の概要

開発環境

Ultra Light Palm アプリケーションは、次のいずれかの開発環境を使用して構築できます。

- Metrowerks CodeWarrior (Embedded SQL または Ultra Light C++ を使用)

  「Metrowerks CodeWarrior で Ultra Light アプリケーションを開発する」 77 ページを参照してください。

  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 10 - 紹介』を参照してください。

- AppForge MobileVB (Ultra Light MobileVB を使用)

  Ultra Light - AppForge プログラミング 『Ultra Light - AppForge プログラミング』を参照してください。

ターゲット・プラットフォーム

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 10 - 紹介』を参照してください。

Metrowerks CodeWarrior で Ultra Light アプリケーションを開発する

Metrowerks CodeWarrior プラグインを使用すると、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。

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 Ultra Light プロジェクトを作成する

♦ 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。タブのあるダイアログが表示されます。

b. [プロジェクト] タブの [Palm OS アプリケーション・ステーショナリ] を選択します。

c. [プロジェクト] タブで、プロジェクト名と場所も選択します。[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。[プロジェクト設定] ウィンドウが開きます。

   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ときは、プラグインはアクセス・パスを *install-dir¥h* と *install-dir¥ultralite¥palm¥68k¥lib¥cw¥* に追加しません。また、Ultra Light ライブラリの *ulrt.lib* と *ulbase.lib* にも追加しません。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 プロジェクトを Ultra Light アプリケーションに変換する

Ultra Light プラグインを CodeWarrior にインストールした場合は、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を 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 **Ultra Light 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、Metrowerks 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¥h¥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 コンパイラの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 10 - 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] – [作成] 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・データのメモリの使い方が向上します。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

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

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 for AppForge の場合：「暗号化と難読化」『Ultra Light - AppForge プログラミング』
- 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」 216 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 プロパティ」『Ultra Light - データベース管理とリファレンス』

Palm 作成者 ID を登録する

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」『Mobile Link - クライアント管理』を参照してください。

作成者 ID は、1 ー 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ファイルは 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 アプリケーションに HotSync 同期を追加する

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提供します。

   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」 81 ページと「Ultra Light の同期パラメータ」『Mobile Link - クライアント管理』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」『Mobile Link - クライアント管理』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マシン上の *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。

  「エンド・ユーザのデスクトップへの Ultra Light HotSync コンジットの配備」『Mobile Link - クライアント管理』を参照してください。

参照

- 「Palm OS の HotSync」『Mobile Link - クライアント管理』

Palm アプリケーションに TCP/IP、HTTP、または HTTPS 通信による同期を追加する

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」『Mobile Link - クライアント管理』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

Metrowerks 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、ULSetSynchInfo を使用して **ul_synch_info** 構造体を引数として提供して、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 アドレスを指定します。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」『Mobile Link - クライアント管理』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」 88 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」 『Mobile Link - クライアント管理』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」 88 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詳細については、「Palm OS の HotSync」 『Mobile Link - クライアント管理』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 同期を使用する場合は、エンド・ユーザが各自のデスクトップ・マシン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詳細については、「エンド・ユーザのデスクトップへの Ultra Light HotSync コンジットの配備」『Mobile Link - クライアント管理』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・マシン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。

第 6 章

Symbian OS 用 Ultra Light アプリケーションの開発

目次

Symbian OS 開発の概要

開発環境

Symbian OS 用 Ultra Light アプリケーションは、次のいずれかの開発環境を使用して構築できます。

- CodeWarrior for Symbian。現在のバージョンは 3.1 ですが、以前のバージョンもサポートされています。

  CodeWarrior には Symbian SDK が付属しています (インストール・パッケージの別 CD に収録されている場合があります)。ターゲットとするデバイスによっては、開発ツールに含まれている 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 10 - 紹介』を参照してください。

- Nokia の Carbide C++ Express Development 環境。これは、32 ビット Windows デスクトップ用の開発ツールと統合開発環境 (IDE) です。http://www.forum.nokia.com/carbide/ を参照してください。

ターゲット・プラットフォーム

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 10 - 紹介』を参照してください。

ランタイム・ライブラリをリンクする方法の選択

Symbian OS 用 Ultra Light サポートでは、2 セットのライブラリ・ファイルが提供されています。これらのライブラリは、*install-dir¥ultralite¥Symbian¥winscw¥* と *install-dir¥ultralite¥Symbian¥thumb¥* にあります。

- *ulrt10.dll* と *¥lib¥ulimp.lib* はアプリケーションとの動的リンク用です。
- *¥lib¥ulrt.lib* と *¥lib¥ulbase.lib* はアプリケーションとの静的リンク用です。

動的および静的 Ultra Light ライブラリは、どちらも Symbian OS の「スレッド・ローカル・ストレージ」技術を使用して、静的データを DLL に格納しています。つまり、どちらのライブラリも DLL 環境でのみ使用できます。データベース・アプリケーションが Symbian OS で実行する GUI アプリケーションである場合、APP ターゲットは特殊なタイプの DLL ということになります。つまり、アプリケーションを、静的な Ultra Light ランタイム・ライブラリ (*ulrt.lib* と *ulbase.lib*) か動的な Ultra Light ランタイム・ライブラリ (*ulimp.lib*) のどちらかとリンクできます。

コンソール・アプリケーションを作成している場合、ターゲットの種類は Symbian OS では EXE になり、アプリケーションは Ultra Light ランタイム・ライブラリの動的なバージョン (*ulimp.lib*) とだけリンクできます。この場合は、*ulrt10.dll* ファイルをエミュレータ・ディレクトリと実際のデバイスに必ずコピーしてください。

Symbian OS アプリケーションでのメモリの使用

Ultra Light ランタイムでは、スタックとヒープの領域としてメモリを大量に使用する可能性があります。Symbian OS の Ultra Light データベース・アプリケーションでは、スタック・サイズとヒープ・サイズをデフォルト設定より大きい値に指定する必要があります。どちらの指定も Symbian プロジェクト (mmp) ファイルで変更できます。Symbian プロジェクト・ファイル仕様の構文の詳細については、Symbian SDK のマニュアルの Tools And Utilities ≫ Build tools reference ≫ mmp file syntax を参照してください。

実行プログラムのスタック・サイズを指定するには epocstacksize 文を使用します。デフォルトのサイズは 8 KB です。

プロセスの初期ヒープの最大および最小サイズを指定するには、epocheapsize 文を使用します。デフォルトのサイズは最小サイズが 4 KB、最大サイズが 1 MB です。

CodeWarrior for Symbian を使用したアプリケーションの開発

この項では、CodeWarrior for Symbian バージョン 3.1 を使用して、Nokia Series 60 デバイス用にサンプル Ultra Light アプリケーションを正常にコンパイルして配備する手順について説明します。このサンプル・プログラムのソース・コードとプロジェクト・ファイルは、*samples-dir¥UltraLite¥SymbianTutorial¥* にあります。

♦ **チュートリアルのアプリケーションをコンパイルするには、次の手順に従います。**

1. チュートリアルのファイルを準備します。

   CodeWarrior for Symbian バージョン 3.1 を使用している場合は、チュートリアル用ディレクトリを (サブフォルダを含めて) SQL Anywhere インストール環境から C: ドライブにコピーします。CodeWarrior 3.1 では、C: ドライブからのプロジェクト・ファイル (拡張子 .mpp) のインポートだけが可能です。CodeWarrior 3.0 以前のバージョンではこの手順は不要ですが、以降の手順はこの手順が実行されていることを前提としています。

   ```
   xcopy /s %SQLANYSAMP10%¥UltraLite¥SymbianTutorial¥ C:¥Symbian¥ULTutorial
   ```

   UTF8 でエンコードされた Ultra Light サンプル・データベース・ファイルを、チュートリアルの「group」ディレクトリにコピーします。これにより、CodeWarrior パッケージング・スクリプトがデータベース・ファイルをデバイス配備用の sis インストーラ・ファイルに追加できるようになります。また、同じサンプル・データベース・ファイルを Nokia Series 60 SDK エミュレータの仮想ファイル・ディレクトリにもコピーします。これは、Ultra Light チュートリアルをエミュレータで実行するためです。環境変数 "%EPOC_ROOT%" は、通常、SDK の EPOC32 ディレクトリのロケーションに設定されています。

   ```
   copy %SQLANYSAMP10%¥UltraLite¥CustDB¥custdb.utf8.udb C:¥Symbian¥ULTutorial¥group
   ¥custdb.udb
   copy %SQLANYSAMP10%¥UltraLite¥CustDB¥custdb.utf8.udb %EPOC_ROOT%¥winscw¥c
   ```

   Ultra Light のランタイム dll とライブラリ・ファイルを Nokia Series 60 SDK のビルド・ターゲット・ディレクトリにコピーします。

   ```
   copy %SQLANY10%¥ultralite¥Symbian¥winscw¥ulrt10.dll %EPOC_ROOT%¥release¥winscw
   ¥udeb
   copy %SQLANY10%¥ultralite¥Symbian¥winscw¥lib¥ulimp.lib %EPOC_ROOT%¥release¥winscw
   ¥udeb

   copy %SQLANY10%¥ultralite¥Symbian¥thumb¥ulrt10.dll %EPOC_ROOT%¥release¥thumb¥urel
   copy %SQLANY10%¥ultralite¥Symbian¥thumb¥lib¥ulimp.lib %EPOC_ROOT%¥release¥thumb
   ¥urel
   ```

2. mmp プロジェクト・ファイルを CodeWarrior for Symbian IDE にインポートします。

   a. CodeWarrior IDE を起動します。

   b. [File] − [Import Project from mmp files] を選択します。

c. [Vendor] から Nokia、[SDK] から Series 60 2.0+ を選択します。

d. プラットフォームとして [WINSCW] または [THUMB] を選択します。

3. Ultra Light のインクルード・パスをプロジェクト設定に追加します。

a. [Edit] - [WINSCW UDEB Settings] - [Target Settings Panels] - [Target] - [Access Paths] - [User Paths] を選択します。

b. [Add] をクリックして %SQLANY10%¥h を [User Paths] に追加します (環境変数を使用せず、絶対パスを使用します)。

c. ビルド・ターゲットを THUMB UREL に変更し、同じ操作を行います。

d. [Add] をクリックして %SQLANY10%¥h を [User Paths] に追加します (環境変数なしで絶対パスを使用します)。

4. アプリケーションを構築します。

a. ターゲットを WINSCW に変更し、[Project] − [Make] を選択します。

これで [Project] - [Run] を選択すると、エミュレータ環境でアプリケーションを実行できます。

b. ターゲットを THUMB UREL に変更し、[Project] − [Make] を選択します。

これにより、プロジェクトが指定のデバイス環境用にコンパイルされます。

5. デバイスのパッケージを作成します。

a. *group¥ULTutorial.pkg* ファイルを編集します。Epoc32 フォルダへの参照を %EPOC_ROOT% 設定に変更し (*pkg* ファイルでは環境変数は認識されないので、絶対パスで指定します)、*custdb.udb* へのパスを使用コンピュータ上のサンプル・データベースのロケーションに置き換えます。

b. THUMB UREL ターゲットの下でソースを右クリックし、ファイル *group¥ULTutorial.pkg* を追加します。

c. ターゲット THUMB UREL のプロジェクトを構築し直し、ファイル *group¥Application.sis* が作成されたことを確認します。これがデバイスに配備するファイルです。

6. デバイスに配備します。

チュートリアル・アプリケーションを Nokia Series 60 携帯電話に配備するには、デバイスに接続し (USB クレードルか、Bluetooth または IRDA によるワイヤレス接続を使用)、Nokia PC Suite ソフトウェアを使用して Application.sis ファイルを携帯電話に転送します。

チュートリアル・プログラムについての注意

チュートリアル・プログラム・ディレクトリには数多くのファイルがあります。ほとんどのファイルは携帯電話のユーザ・インタフェース (GUI) に関係しています。Ultra Light アプリケーションのメイン・コードは、*¥src* サブディレクトリの *ULTutorialDB.cpp* というファイルに含まれています。

携帯電話上のデータベース・ファイルのロケーションと、Ultra Light 接続コードでそれを参照する方法に関して、いくつかの注意事項があります。データベースに接続するための C++ コードでは、設定 DBF=C:¥¥custdb.udb を使用してデータベース・ファイルの名前 (とパス) を指定しています。これは、データベース・ファイルが携帯電話の (ストレージ・カードにではなく) メイン・メモリに配置されていることを想定したものです。

アプリケーションを携帯電話で起動できなかった場合、データベース・ファイルのロケーションが正しくないことが原因である可能性があります。

第 7 章

Windows CE 用 Ultra Light アプリケーションの開発

目次

WindowsCE 開発の概要

Windows CE での開発でサポートされるホスト・プラットフォームと開発ツールのリストと、サポートされるターゲット Windows CE 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 10 - 紹介』を参照してください。

ほとんどの Windows CE 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。

Windows CE 開発の準備

Microsoft eMbedded Visual C++ は、Windows CE 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 CE 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・ライブラリでリンクする必要があります。

最初のアプリケーション

サンプルの eMbedded Visual C++ プロジェクトが、*samples-dir¥UltraLite¥CEStarter* ディレクトリに格納されています。ワークスペース・ファイルは *samples-dir¥UltraLite¥CEStarter¥ul_wceapplication.vcw* と呼ばれます。

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 *$(SQLANY10)¥h* を追加します。
  - 適切なコンパイラ・ディレクティブを定義します。たとえば、eMbedded Visual C++ プロジェクトに対して UNDER_CE マクロを定義してください。
- リンカの設定
  - "*$(SQLANY10)¥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。

    ```
    "$(SQLANY10)¥win32¥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。

ランタイム・ライブラリをリンクする方法の選択

Windows CE はダイナミック・リンク・ライブラリをサポートします。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。

パフォーマンスのヒント
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・ディレクトリの *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 CE 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 10 - 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] – [アクティブな構成の設定] 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。

     [プロジェクト設定] ダイアログが表示されます。
   - [Link] 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] – [Build EVCCustDB.exe] 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、[プログラム] – [SQL Anywhere 10] – [Mobile Link] – [Mobile Link 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」 71 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 CE 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、永続的保存ファイルのフル・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 CE アプリケーションの配備

Windows CE 用の Ultra Light アプリケーションのコンパイルでは、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 CE エミュレータ用の Ultra Light ランタイム DLL を配備するには、DLL を Windows CE tools ディレクトリの適切なサブディレクトリに置きます。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 用 Mobile Link プロバイダもインストールします。

「ActiveSync Manager を使用したアプリケーションの登録」『Mobile Link - クライアント管理』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、一般的なクラス名である **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。

   - ノートパッドなどのテキスト・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)」 107 ページを参照してください。

Windows CE での同期

Windows CE 上の Ultra Light アプリケーションは、以下のストリームのタイプ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」 106 ページを参照してください。
- **TCP/IP** 「Windows CE からの TCP/IP、HTTP、または HTTPS 同期」 109 ページを参照してください。
- **HTTP** 「Windows CE からの TCP/IP、HTTP、または HTTPS 同期」 109 ページを参照してください。

Windows CE では、初期化のときに *user_name* パラメータと *stream_parms* パラメータを **UL_TEXT()** マクロで囲んでください。これは、コンパイル環境がユニコード・ワイド文字であるためです。

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」 『Mobile Link - クライアント管理』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 CE 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」 103 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 1 は、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判断するには、「プラットフォーム別 SQL Anywhere 10.0.1 コンポーネント」にある Ultra Light の表の「ActiveSync」の項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)

直接 Windows API でプログラミングしている場合は、アプリケーションの **WindowProc** 関数を使用して、Mobile Link プロバイダからのメッセージを処理します。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」 329 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」 103 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。

   次のシグニチャで、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」 329 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] ボタンをクリックします。
3. [コードの編集] ボタンをクリックします。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」 329 ページを参照してください。

Windows CE からの TCP/IP、HTTP、または 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。

パート III. チュートリアル

パート III はチュートリアル形式で、簡単な Ultra Light アプリケーションを開発する手順を理解します。

第 8 章

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

Ultra Light C++ 開発の概要

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 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++ .NET 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、「Sybase Central からの Ultra Light 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. 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。

2. 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、「Sybase Central からの Ultra Light 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。

3. **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	

♦ Ultra Light データベースへの接続

1. Microsoft Visual C++ .NET 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 <stdafx.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」 16 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」 216 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」 31 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"$(SQLANY10)¥h"
  ```

- makefile で、ディレクトリをライブラリ・パスに追加します。

  ```
  LibraryFolders=/LIBPATH:"$(SQLANY10)¥ultralite¥win32¥386¥lib"
  ```

- 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シ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ブール値が返されます。

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( "¥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( "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 10.0 サンプル・データベースと同期します。Ultra Light 10.0 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」『Mobile Link - クライアント管理』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 10.0" );
  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。

```
mlsrv10 -c "dsn=SQL Anywhere 10 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 サーバのオプション」『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 <stdafx.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( "¥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( "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 10.0" );
  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;
}
```

第 9 章

チュートリアル：Embedded SQL を使用したアプリケーションの構築

目次

Embedded SQL 開発チュートリアルの概要

このチュートリアルでは、Microsoft eMbedded Visual C++ を使用して Embedded SQL Ultra Light アプリケーションを開発します。作成した Ultra Light アプリケーションは、Windows CE エミュレータまたはデバイスで実行できます。

開発プロセスの概要については、「Embedded SQL アプリケーションの開発」 4 ページを参照してください。

Embedded SQL Ultra Light アプリケーションの開発の詳細については、「Embedded SQL を使用したアプリケーションの開発」 37 ページを参照してください。

このチュートリアルは、Microsoft eMbedded Visual Tools がインストールされているマシンに、Ultra Light がすでにインストールされていることを前提としています。別の C/C++ 開発ツールを使用する場合は、eMbedded Visual C++ の命令を使用する開発ツールの対応する命令に変換する必要があります。

♦ **チュートリアルの準備をするには、次の手順に従います。**

· 作成するファイルを格納するディレクトリをデスクトップ・コンピュータに作成します。

以降、このチュートリアルではプロジェクト・ファイルのロケーションがディレクトリ *C:¥tutorial¥* であることを前提に説明します。

レッスン 1：Ultra Light データベースの作成

このチュートリアル・アプリケーションで使用する Ultra Light データベースを作成する必要があります。これを行うには、Sybase Central を使用して、Ultra Light データベースをデスクトップ・コンピュータに作成するのも 1 つの方法です。Ultra Light データベースを含むファイルは、後で Windows CE デバイスに転送します。このチュートリアルでは、データベースを *C:¥tutorial ¥esqldb.udb* というファイル名でデスクトップ・コンピュータに作成します。

Sybase Central では、ユーザ ID DBA とパスワード sql を使用してデータベースを作成します (Ultra Light データベースでは、パスワードの大文字と小文字は区別されますが、ユーザ名については区別されません)。

データベースには ULProduct というテーブルが 1 つ作成されます。このテーブルのフィールドと特性は次のとおりです。

プライマリ・キー	名前	データ型	NULL	ユニーク
yes	prod_id	integer	no	yes
	price	integer	no	no
	prod_name	varchar(30)	no	no

Windows エクスプローラまたは eMbedded Visual C++ に付属の「リモート ファイル ビューア」ツールを使用して、データベース・ファイルをターゲット・デバイスにコピーします。チュートリアル・プログラムのデータベース接続情報には **dbf=¥esqldb.udb** と指定されています。これは、デバイス上のデータベース・ファイルが CE デバイスのルート・フォルダにあることを示します。データベース・ファイルは、デバイスのファイル・システム上ならどこにでも配置できます。ただし、接続情報をそれに合わせて変更する必要があります。Windows CE のファイル・エクスプローラにはファイル拡張子が表示されないので、実行ファイル *esql.exe* と明確に区別されるように、データベース・ファイル名は *esqldb.udb* になっています。

レッスン 2：eMbedded Visual C++ の設定

次の手順は、Ultra Light 開発用に eMbedded Visual C++ を設定します。この手順のテストには eMbedded Visual C++ のバージョン 4.0 が使用されていますが、他のバージョンでもプロセスは同じです。

♦ **Ultra Light 開発用に eMbedded Visual C++ を設定するには、次の手順に従います。**

1. Microsoft eMbedded Visual C++ を起動します。

   [スタート] メニューから [すべてのプログラム] – [Microsoft eMbedded Visual C++ 4.0] – [eMbedded Visual C++ 4.0] を選択します。

2. Embedded SQL ヘッダ・ファイルと Ultra Light ライブラリ・ファイルを適切なディレクトリで検索するように eMbedded Visual C++ を設定します。

   a. [ツール] – [オプション] を選択します。

      [オプション] ダイアログが表示されます。

   b. [ディレクトリ] タブをクリックします。

   c. ターゲットのプラットフォームと CPU の各組み合わせについて、次の手順を実行します。

      ♦ [表示するディレクトリ] ドロップダウン・リストの [インクルード ファイル] を選択します。[参照] ボタンをクリックし、SQL Anywhere インストール・ディレクトリの **¥h** ディレクトリを選択に加えて、Embedded SQL ヘッダ・ファイルにアクセスできるようにします。

        install-dir¥h

      ♦ [表示するディレクトリ] ドロップダウン・リストの [ライブラリ ファイル] を選択します。プラットフォーム固有のディレクトリにある Ultra Light *¥lib* ディレクトリをインクルードします。たとえば、Pocket PC 2002 エミュレータの場合は次のように指定します (環境変数をそのまま使用することはできません。フル・パス名を使用してください)。

        install-dir¥UltraLite¥ce¥386¥lib

   d. [OK] をクリックします。

3. Ultra Light チュートリアル・プロジェクトで使用するプロジェクト・レベルの設定を行います。

   a. esql ファイルを右クリックし、[設定] を選択します。

      [プロジェクトの設定] ダイアログが表示されます。

   b. [設定の対象] ドロップダウン・リストから適切なプラットフォームを選択します。

   c. [リンク] タブをクリックし、適切なランタイム・ライブラリを [オブジェクト/ライブラリ モジュール] ボックスに追加します。

ulbase.lib を指定します。

静的な Ultra Light ライブラリの場合は、ulrt.lib を指定します。

Ultra Light のダイナミック・リンク・ライブラリの場合は、ulimp.lib を指定します。

Ultra Light ライブラリまたはダイナミック・リンク・ライブラリの使用
プラットフォームに応じて、アプリケーションは静的な Ultra Light ライブラリ (*ulrt.lib*) または Ultra Light ダイナミック・リンク・ライブラリ (DLL) を使用します。DLL を使用することを選択した場合は、適切な .dll ファイルをデバイスにコピーし、C/C++ プリプロセッサ定義として UL_USE_DLL を設定してください。
たとえば、WCE ARM プラットフォームの場合は、*install-dir¥ultralite¥ce¥arm¥ulrt10.dll* を ARM ベースの CE デバイスの *Windows* フォルダにコピーします。WCE Ax86 (エミュレータ) プラットフォームの場合は、*install-dir¥ultralite¥ce¥386¥ulrt10.dll* をエミュレータ・デバイスの *Windows* フォルダにコピーします。
静的ライブラリの使用を選択すると、アプリケーションは他のファイルの有無に依存しなくなり、アプリケーションの配布や配備が楽になります。

レッスン 3：Embedded SQL ソース・ファイルの記述

次の手順では、Ultra Light CustDB サンプル・データベースとの接続を確立するチュートリアル・プログラムを作成し、クエリを実行します。

♦ Embedded SQL Ultra Light チュートリアル・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. Microsoft eMbedded Visual C++ を起動します。

   [スタート] メニューから [すべてのプログラム] – [Microsoft eMbedded Visual C++ 4.0] – [eMbedded Visual C++ 4.0] を選択します。

2. eMbedded Visual C++ で、Ultra Light という名前の新しいワークスペースを作成します。

   a. [ファイル] – [新規作成] を選択します。

   b. [ワークスペース] タブをクリックします。

   c. [ブランク ワークスペース] を選択します。ワークスペース名 Ultra Light を指定し、このワークスペースを保存するロケーションとして *C:¥tutorial* を指定します。[OK] をクリックします。

      Ultra Light ワークスペースが [ワークスペース] ウィンドウに追加されます。

3. 新規プロジェクト esql を作成し、Ultra Light ワークスペースに追加します。

   a. [ファイル] – [新規作成] を選択します。

   b. [プロジェクト] タブをクリックします。

   c. 使用可能なアプリケーションの種類は、インストールした SDK によって異なります。使用デバイスに適したアプリケーションの種類を選択してください。たとえば、Pocket PC 2002 デバイスの場合は [WCE Pocket PC 2002 Application] を選択します。プロジェクト名 esql を指定し、[現在のワークスペースへ追加] を選択します。適用可能な CPU を選択します。[OK] をクリックします。

   d. [空のプロジェクト] の作成を選択し、[終了] をクリックします。

      プロジェクトが *c:¥tutorial¥esql* フォルダに保存されます。

4. ソース・ファイル *sample.sqc* を作成します。

   a. [ファイル] – [新規] を選択します。

   b. [ファイル] タブをクリックします。

   c. [C++ ソース ファイル] を選択します。

   d. [プロジェクトに追加] を選択し、ドロップダウン・リストから esql を選択します。

   e. ファイルに *sample.sqc* という名前を付けます。[OK] をクリックします。

f. 次のソース・コードをファイルにコピーします。次の項では、このチュートリアル・コードについて詳しく説明します。

```
#include <tchar.h>
#include <windows.h>

EXEC SQL INCLUDE SQLCA;

int WINAPI WinMain( HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPTSTR lpCmdLine,
    int nShowCmd)
{
  /* Declare fields */
  EXEC SQL BEGIN DECLARE SECTION;
     long pid=1;
     long cost;
     TCHAR pname[31];
  EXEC SQL END DECLARE SECTION;

  TCHAR       output[200];
  TCHAR       result[10];

  /* Before working with data*/
  db_init(&sqlca);

  /* Connect to database */

  EXEC SQL WHENEVER SQLERROR GOTO error;

  EXEC SQL CONNECT USING 'dbf=¥Windows¥Start Menu¥esqldb.udb';

  /* Fill table with data first */
  EXEC SQL INSERT INTO ULProduct(
       prod_id, price, prod_name)
    VALUES (1, 400, '4x8 Drywall x100');

  EXEC SQL INSERT INTO ULProduct (
       prod_id, price, prod_name)
    VALUES (2, 3000, '8"2x4 Studs x1000');

  /* Commit changes (INSERTs) to database */
  EXEC SQL COMMIT;


  /* Fetch row 1 from database */
  EXEC SQL SELECT price, prod_name
       INTO :cost, :pname
       FROM ULProduct
       WHERE prod_id= :pid;
       /* pid was initialized to 1, so get first row inserted above */

  /* Print query results */

  wsprintf( output
          , TEXT("Product id: %d¥n     price: %d¥n      name: %s")
          , pid , cost , pname );
  MessageBox(NULL, output, result, MB_OK);

  /* Preparing to exit: disconnect */
  EXEC SQL DISCONNECT;
  db_fini(&sqlca);
```



```
    return(0);

    /* Error handling.  If the row does not exist,
       or if an error occurs, -1 is returned */
    error:
     if((SQLCODE==SQLE_NOTFOUND)||(SQLCODE<0)) {
            wsprintf (output, TEXT("error: %d"), SQLCODE );
          MessageBox(NULL, output, result, MB_OK);
            return(-1);
     }
  }
```

g. ファイルを保存します。

5. *sample.sqc* ソース・ファイルの設定を、SQL プリプロセッサを起動して、ソース・ファイルの前処理を実行するように設定します。

   a. [ワークスペース] ウィンドウで *sample.sqc* ファイルを右クリックし、[設定] を選択します。

      [プロジェクト設定] ダイアログが表示されます。

   b. [設定の対象] ドロップダウン・リストから [すべての構成] を選択します。

   c. [カスタム ビルド] タブで、次の文を [コマンド] フィールドに入力します。文は、すべて 1 行に入力するようにしてください。

      ```
      "%SQLANY10%¥win32¥sqlpp.exe" -q -u $(InputPath) sample.cpp
      ```

      この文は、*sample.sqc* ファイルで SQL プリプロセッサ *sqlpp* を実行し、処理した出力を *sample.cpp* ファイルに書き込みます。SQL プリプロセッサは、ソース・ファイルの SQL 文を C/C++ 関数呼び出しに変換します。

      SQL プリプロセッサの詳細については、「SQL プリプロセッサ」 『SQL Anywhere サーバ - プログラミング』を参照してください。

   d. [出力] フィールドに **sample.cpp** と入力します。

   e. [OK] をクリックして変更を確定します。

6. *sample.sqc* ファイルの前処理を行います。

   a. [ワークスペース] ウィンドウで *sample.sqc* を選択します。[ビルド] – [コンパイル sample.sqc] を選択します。*sample.cpp* ファイルが作成され、*tutorial¥esql* ディレクトリに保存されます。

7. *sample.cpp* (SQL プリプロセッサの出力) をプロジェクトに追加します。

   a. [ワークスペース] ウィンドウの [ソース ファイル] フォルダを右クリックして、[ファイルをフォルダへ追加] を選択します。

   b. *c:¥tutorial¥esql¥sample.cpp* を参照し、[OK] をクリックします。

      sample.cpp ファイルが、[ソース ファイル] フォルダに表示されます。

チュートリアル・プログラムの説明

このチュートリアル・プログラムは簡単な内容ですが、データベースへのアクセスに使用する Embedded SQL ソース・ファイルに含まれる共通の要素が含まれています。

ここでは、このチュートリアル・プログラムの主な要素を解説します。Embedded SQL Ultra Light アプリケーションを作成するときには、この手順を参考にしてください。

1. 適切なヘッダ・ファイルを含めます。

   このチュートリアル・プログラムにインクルードされているヘッダ・ファイル *tchar.h* は TCHAR 型指定の使用をサポートしており、*windows.h* は MessageBox 関数を使用した CE デバイスへのメッセージ表示をサポートしています。

2. SQLCA (SQL Communications Area) を定義します。

   ```
   EXEC SQL INCLUDE SQLCA;
   ```

   この定義は、各ソース・ファイルの最初の Embedded SQL 文である必要があるため、インクルード・リストの最後に入れてください。

   プレフィクス SQL 文
   SQL 文は、キーワード EXEC SQL のプレフィクスで始め、セミコロンで終える必要があります。

3. 宣言セクションを作成して、ホスト変数を定義します。

   ホスト変数はデータベース・サーバに値を送信したり、データベース・サーバから値を受信したりするのに使用します。このチュートリアルのクエリは、製品 ID、価格、製品名をデータベースから取得します。ホスト変数を次のように定義します。

   ```
   EXEC SQL BEGIN DECLARE SECTION;
      long pid=1;
      long cost;
      TCHAR pname[31];
   EXEC SQL END DECLARE SECTION;
   ```

   「ホスト変数の使用」 45 ページを参照してください。

4. ローカル・プログラム変数を宣言します (このチュートリアル・アプリケーションではローカル変数を画面メッセージの作成と表示に使用します)。

   ```
   TCHAR       output[200];
   TCHAR       result[10];
   ```

5. Embedded SQL ライブラリ関数 db_init を呼び出して、Ultra Light ランタイム・ライブラリを初期化します。

   次のように、この関数を呼び出します。

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

6. SQL 文の処理中にランタイム・エラーが発生した場合に制御の転送先となる場所にラベルを定義します。


```
EXEC SQL WHENEVER SQLERROR GOTO error;
```

7. CONNECT 文を使用してデータベースに接続します。

   Ultra Light サンプル・データベースに接続するには、アプリケーションはデータベース・ファイルの名前とロケーションを指定する必要があります。データベース・ファイル名は *esqldb.udb* で、ロケーションはデバイスのルート・ディレクトリです。

   ```
   EXEC SQL CONNECT USING 'dbf=¥esqldb.udb'
   ```

8. データベース・テーブルにデータを挿入します。

   ```
   /* Fill table with data first */
   EXEC SQL INSERT INTO ULProduct(
       prod_id, price, prod_name)
     VALUES (1, 400, '4x8 Drywall x100');

   EXEC SQL INSERT INTO ULProduct (
       prod_id, price, prod_name)
     VALUES (2, 3000, '8"2x4 Studs x1000');
         /* Commit changes (INSERTs) to database */
   EXEC SQL COMMIT;
   ```

   リモート・データベースを統合データベースと同期すると、データがテーブルに挿入されます。それ以降、Select、Update、Delete コマンドを実行できます。

   このコードは、同期機能を使用せずに、データをテーブルに直接挿入します。Ultra Light 開発の初期段階では、データの直接挿入は効率的な方法であるといえます。

   同期機能を使用している状態でアプリケーションがクエリの実行に失敗した場合、その原因は同期プロセスの問題またはプログラム・エラーになりますが、どちらであるかを特定することは困難です。一方、同期機能に頼らず、ソース・コードのデータをテーブルに直接挿入する場合、アプリケーションが失敗すると、失敗の原因はプログラム・エラーであることがすぐにわかります。

   プログラムをテストして誤りがないことを確認したら、INSERT 文を削除し、ULSynchronize 関数の呼び出しに置き換えて、リモート・データベースを統合データベースと同期します。

   「レッスン 5 ：アプリケーションへの同期の追加」 139 ページを参照してください。

9. SQL クエリを実行します。

   ```
   /* Fetch row 1 from database */
   EXEC SQL SELECT price, prod_name
         INTO :cost, :pname
         FROM ULProduct
         WHERE prod_id= :pid;
         /* pid was initialized to 1, get first row inserted */
   ```

   チュートリアル・プログラムは、結果のローを 1 つ返すクエリを実行します。結果は、定義済みのホスト変数 cost と pname に格納されます。Embedded SQL 文では、変数名の先頭にはコロンが付いており、SQL プリプロセッサに対してその名前がプログラム変数であることを示します。

10. クエリの結果を表示します。

```
/* Print query results */

wsprintf( output, TEXT("Product id: %d¥n     price: %d¥n      name: %s"), pid, cost, pname );
MessageBox(NULL, output, result, MB_OK);
```

11. エラー処理ルーチンを指定します。

    チュートリアル・プログラムの前半に次の Embedded SQL 文があります。

```
EXEC SQL WHENEVER SQLERROR GOTO error;
```

    これは、エラーが発生した場合に制御をプログラム・ラベル error に転送して処理します。このプログラムに定義されているハンドラ・コードでは、フィールド SQLCODE から取得した SQL エラー・コードを含むメッセージを出力し、main 関数を結果 -1 で終了します。

12. データベースを切断します。

    アプリケーションは、データベースから切断する前に、変更があった場合は EXEC SQL COMMIT 文を発行してそれをコミットする必要があります。そうしないと保留中の変更が切断時に破棄されます。

    切断するには、次のように DISCONNECT 文を使用します。

```
EXEC SQL DISCONNECT;
```

13. 次のようにして関数 db_fini を呼び出して、データベース・インタフェースを適切に閉じ、使用したリソースを解放します。

```
db_fini(&sqlca);
```


レッスン 4：Embedded SQL Ultra Light チュートリアル・アプリケーションの構築

次の手順では、前のレッスンで作成したソース・ファイル *sample.cpp* を使用して、Embedded SQL Ultra Light チュートリアル・アプリケーションを作成します。

♦ Embedded SQL Ultra Light チュートリアル・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. 実行プログラムを構築します。

   - [ビルド] – [ビルド esql.exe] を選択します。

     esql 実行プログラムが作成されます。設定内容に応じて、実行プログラムがチュートリアルのディレクトリ内の Debug ディレクトリに作成されます。

2. アプリケーションを実行します。

   - [ビルド] – [実行 esql.exe] を選択します。

     実行プログラムがターゲット・デバイスにコピーされ、実行が開始されます。実行中にエラーが発生しなければ、メッセージ・ボックスが開き、製品テーブルの最初のローの内容が表示されます。エラーが発生した場合は、エラー・メッセージ・ボックスが開き、エラーに関連付けられている SQL エラー・コードの数値が表示されます。「エラー・メッセージ (SQL Anywhere の SQLCODE 順)」『SQL Anywhere 10 - エラー・メッセージ』を参照してください。

レッスン 5 ：アプリケーションへの同期の追加

プログラムが正しく動作することをテストしたら、データを ULProduct テーブルに手動で挿入するコードを、リモート・データベースと統合データベースとを同期するコードに置き換えます。同期することにより、テーブルにデータが挿入され、アプリケーションはそのデータを操作できます。

TCP/IP を介する同期

TCP/IP のソケット接続などを使用して、リモート・データベースを統合データベースと同期できます。まず ULEnableTcpipSynchronization (または同様の ULEnableXXX 関数) を呼び出し、その後に ULSynchronize の呼び出しを発行します。同期の種類の詳細については、「Ultra Light の同期パラメータ」 『Mobile Link - クライアント管理』を参照してください。

CustDB 統合データベースと同期するには、アプリケーションは従業員 ID フィールドの値を指定する必要があります。この ID は、Mobile Link サーバに対する、アプリケーションのインスタンスを識別します。50、51、52、53 の中から値を 1 つ選択できます。選択した値は、Mobile Link サーバによって、ダウンロード内容の決定、同期ステータスの記録、同期中の任意の割り込みからのリカバリに使用されます。

「ULSynchronize 関数」 340 ページを参照してください。

同期機能を使用してチュートリアル・アプリケーションを実行する

sample.sqc を変更したら、*esql.exe* を再構築します。

♦ **アプリケーションを同期するには、次の手順に従います。**

1. Embedded SQL コマンド INSERT を削除し、次のコードを追加します。

```
ULEnableTcpipSynchronization( &sqlca );
auto ul_synch_info synch_info;
ULInitSynchInfo( &synch_info );
synch_info.user_name = UL_TEXT("50");
synch_info.version = UL_TEXT("custdb 10.0");
ULSynchronize( &sqlca, &synch_info );
```

2. *sample.sqc* の前処理を実行します。

   [ビルド] – [コンパイル sample.sqc] を選択して、変更したファイルを再コンパイルします。メッセージが表示されたら、*sample.cpp* の再ロードを選択します。

3. 実行プログラムをビルドします。

   [ビルド] – [ビルド esql.exe] を選択して、実行プログラムをビルドします。

4. SQL Anywhere データベース・サーバが実行中であることを確認します。

5. Mobile Link サーバを起動します。

コマンド・プロンプトで、以下のコマンドを単一行で入力しても Mobile Link 同期サーバを起動できます。

```
mlsrv10 -c "DSN=SQL Anywhere 10 CustDB" -o ulsync.mls -v+ -x tcpip
```

6. アプリケーションを実行します。

   - [ビルド] – [実行 esql.exe] を選択して、アプリケーションを実行します。

     リモート・データベースは統合データベースと同期され、リモート・データベースのテーブルにデータが挿入されます。アプリケーションのクエリが処理され、クエリの結果のローが画面に表示されます。

第 10 章

チュートリアル：ODBC を使用したアプリケーションの構築

目次

Ultra Light ODBC の概要

「**ODBC**」は、標準のデータベース・プログラミング・インタフェースです。Ultra Light では、ODBC インタフェースのサブセットと、同期を行うための拡張機能をサポートしています。Ultra Light でサポートされている関数のリストについては、「Ultra Light ODBC API リファレンス」 341 ページを参照してください。

この項では、簡単な Ultra Light ODBC アプリケーションの作成について説明します。ODBC プログラミングについて詳しくは説明しません。ODBC のメイン・リファレンスは Microsoft の ODBC SDK マニュアルです。

Ultra Light ODBC では、他の C/C++ インタフェースと機能を共有しません。Ultra Light ODBC から使用できない関数のリストについては、「Ultra Light C/C++ 共通 API リファレンス」 153 ページを参照してください。

レッスン 1 ：はじめに

このチュートリアルでは、ソース・ファイルや実行可能ファイルを含む一連のファイルを作成します。これらのファイルを格納するディレクトリを作成してください。チュートリアル後半では、このディレクトリが *c:¥tutorial¥ulodbc* であることを前提に説明します。別の名前を選択する場合は、全体を通じてその名前を使用してください。

ODBC インタフェースは、特定の C/C++ コンパイラや開発環境には依存しません。チュートリアルでは、Microsoft の nmake 構文の makefile を使用します。別の開発環境を使用している場合は、適切な置き換えが必要です。

♦ 構築環境の作成とテスト

1. チュートリアル・ディレクトリ内の *makefile* ファイルに次のコードを追加します。

```
IncludeFolders= /I"$(SQLANY10)¥h"

CompileOptions=/c /nologo /W3 /Od /Zi /DWIN32 /DUL_USE_DLL

LibraryFolders= ¥
/LIBPATH:"$(SQLANY10)¥ultralite¥win32¥386¥lib"

Libraries= ulimp.lib

LinkOptions=/NOLOGO /DEBUG

sample.exe: sample.obj
  link $(LinkOptions) sample.obj $(LibraryFolders) $(Libraries)

sample.obj: sample.cpp
  cl $(CompileOptions) $(IncludeFolders) sample.cpp
```

   このオプションによって、ソース・ファイル *sample.cpp* が、Windows の Ultra Light インポート・ライブラリ (*ulimp.lib*) を使用して、実行プログラム *sample.exe* にコンパイルされます。これは、SQL Anywhere インストール・ディレクトリとして定義されている環境変数 SQLANY10 に依存します。「SQL Anywhere の環境変数」 『SQL Anywhere サーバ - データベース管理』を参照してください。

2. チュートリアル・ディレクトリ内の *sample.cpp* ファイルに次のコードを追加します。

```
#include "ulodbc.h"
#include <stdio.h>
#include <tchar.h>
int main() {
    return 0;
}
```

   このアプリケーションは、呼び出しを行った環境に単に 0 を返します。

3. *sample.cpp* のコンパイルとリンクを行います。

Microsoft コンパイラを使用している場合、コマンド・プロンプトで **nmake** と入力して、アプリケーションのコンパイルとリンクを行います。それ以外の場合は、開発環境に適したコマンドを使用します。

アプリケーションのコンパイルとリンクを行うことで、構築環境が正しくセットアップされていることが確認されます。これで、次に進む準備ができました。

レッスン 2：Ultra Light データベースの作成

このチュートリアルでは、簡単な 1 テーブルのデータベースを使用します。

この項では、Sybase Central の Ultra Light プラグインを使用してデータベースを作成できることを想定しています。

データベースの作成については、「Sybase Central からの Ultra Light 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。

♦ Ultra Light データベースの作成

· Sybase Central で、Ultra Light の [データベース作成] ウィザードを使用してデータベースを作成します。

以下のようにデータベースを作成します。

- **データベース・ファイル名** *c:¥tutorial¥ulodbc¥customer.udb*
- **テーブル名** customer
- **customer テーブルのカラム**

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
id	integer	いいえ	オートインクリメント
fname	varchar(15)	いいえ	なし
lname	varchar(20)	いいえ	なし
city	varchar(20)	はい	なし
phone	varchar(12)	はい	555-1234

- **プライマリ・キー** 昇順 id

レッスン 3 ：データベースへの接続

Ultra Light では、標準の ODBC プログラミング・メソッドを使用してデータベースに接続します。各アプリケーションには、Ultra Light との通信を管理する環境ハンドルと、特定の接続の接続ハンドルが必要です。

♦ 環境ハンドルを割り当てるコードの記述

1. opendb 関数と closedb 関数を *sample.cpp* に追加します。

```
static SQLHANDLE opendb( void ){
    SQLRETURN retn;
    SQLHANDLE henv;
    retn = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv );
    if( retn == SQL_SUCCESS ){
        _tprintf( "success in opendb: %d.¥n", retn );
        return henv;
    } else {
        _tprintf( "error in opendb: %d.¥n", retn );
        return henv;
    }
}

static void closedb( SQLHANDLE henv ){
    SQLRETURN retn;
    retn = SQLFreeHandle( SQL_HANDLE_ENV, henv );
}
```

   これらの関数はデータベースに接続するのではなく、Ultra Light 機能を管理する環境ハンドル henv を単に割り当てます。

2. main 関数から opendb と closedb を呼び出します。

   sample.cpp 内の main 関数を、次のように変更します。

```
int main() {
    SQLHANDLE henv;
    henv = opendb();
    closedb( henv );
    return 0;
}
```

3. アプリケーションのコンパイル、リンク、実行を行い、アプリケーションが正しく構築されたことを確認します。

この手順で呼び出されている関数の詳細については、「SQLAllocHandle 関数」 343 ページと「SQLFreeHandle 関数」 354 ページを参照してください。

次に、Ultra Light データベースに接続します。

♦ データベースとの接続のためのコードの作成

1. connect 関数と disconnect 関数を *sample.cpp* に追加します。


```
static SQLHANDLE connect ( SQLHANDLE henv ){
    SQLRETURN retn;
    SQLHANDLE hcon;
    retn = SQLAllocHandle( SQL_HANDLE_DBC, henv, &hcon );
    retn = SQLConnect( hcon
     , (SQLTCHAR*)UL_TEXT(
        "dbf=customer.udb" )
     , SQL_NTS
     , (SQLTCHAR*)UL_TEXT( "DBA" )
     , SQL_NTS
     , (SQLTCHAR*)UL_TEXT( "sql" )
     , SQL_NTS );
    if( retn == SQL_SUCCESS ){
        _tprintf( "success in connect: %d.¥n", retn );
        return hcon;
    } else {
        _tprintf( "error in connect: %d.¥n", retn );
        return hcon;
    }
}

static void disconnect( SQLHANDLE hcon, SQLHANDLE henv ){
    SQLRETURN retn;
    retn = SQLDisconnect( hcon );
    retn = SQLFreeHandle( SQL_HANDLE_DBC, hcon );
}
```

2. main 関数から connect と disconnect を呼び出します。

   sample.cpp 内の main 関数を、次のように変更します。

```
int main() {
    SQLHANDLE henv;
    SQLHANDLE hcon;
    henv = opendb();
    hcon = connect( henv );
    disconnect( hcon, henv );
    closedb( henv );
    return 0;
}
```

3. アプリケーションのコンパイル、リンク、実行を行い、アプリケーションが正しく構築されたことを確認します。

この手順で呼び出されている関数の詳細については、「SQLConnect 関数」 346 ページと「SQLDisconnect 関数」 348 ページを参照してください。

データベースに接続し、切断するアプリケーションができました。次に、データベースにデータを追加します。

レッスン 4 ：データベースへのデータの挿入

ODBC では、データベースの操作を行う関数のセットを提供します。このレッスンでは、SQLExecDirect という最も簡単な文を使用します。

♦ データベースにデータを挿入するコードの記述

1. insert 関数を *sample.cpp* に追加します。

```
static ul_bool insert( SQLHANDLE hcon )
{
    SQLRETURN retn;
    SQLHANDLE hstmt;
    retn = SQLAllocHandle( SQL_HANDLE_STMT, hcon, &hstmt );
    static const ul_char * sql = UL_TEXT(
     "INSERT customer( id, fname, lname ) VALUES ( 42, 'jane', 'doe' )" );
    retn = SQLExecDirect( hstmt, (SQLTCHAR*)sql, SQL_NTS );
    if( retn == SQL_SUCCESS ) {
        _tprintf( "success in insert.¥n" );
    } else {
        _tprintf( "error in insert: %d.¥n", retn );
    retn = SQLFreeHandle( SQL_HANDLE_STMT, hstmt );
    hstmt = 0;
    }
    return retn == SQL_SUCCESS;
}
```

2. main 関数から insert を呼び出します。

   sample.cpp 内の main 関数を、次のように変更します。

```
int main() {
    SQLHANDLE henv;
    SQLHANDLE hcon;
    henv = opendb();
    hcon = connect( henv );
    insert( hcon );
    disconnect( hcon, henv );
    closedb( henv );
    return 0;
}
```

3. アプリケーションのコンパイル、リンク、実行を行い、アプリケーションが正しく構築されたことを確認します。

   データの挿入が正しく行われたというメッセージが表示されていることを確認してください。

この手順で呼び出されている関数の詳細については、「SQLExecDirect 関数」 350 ページを参照してください。

レッスン 5 ：データベースのクエリ

クエリ結果セットを処理するために、ODBC では、文を準備してから実行します。このレッスンでは、文を準備して実行し、結果を出力します。

♦ データベースに対してクエリを実行するコードの記述

1. prepare 関数、execute 関数、fetch 関数を *sample.cpp* に追加します。

```
static SQLHANDLE prepare( SQLHANDLE hcon ){
    SQLRETURN retn;
    SQLHANDLE hstmt;
    static const ul_char * sql =
        UL_TEXT( "SELECT id, fname, lname FROM customer" );
    retn = SQLAllocHandle( SQL_HANDLE_STMT, hcon, &hstmt );
    retn = SQLPrepare( hstmt, (SQLTCHAR*)sql, SQL_NTS );
    if( retn == SQL_SUCCESS ) {
        _tprintf( "success in prepare.¥n" );
    } else {
        _tprintf( "error in prepare: %d.¥n", retn );
        retn = SQLFreeHandle( SQL_HANDLE_STMT, hstmt );
        hstmt = 0;
    }
    return hstmt;
}
```

prepare 関数は SQL 文を実行しません。

```
static ul_bool execute( SQLHANDLE hstmt )
{
    SQLRETURN retn;
    retn = SQLExecute( hstmt );
    if( retn == SQL_SUCCESS ) {
        _tprintf( "success in execute.¥n" );
    } else {
        _tprintf( "error in execute: %d.¥n", retn );
    retn = SQLFreeHandle( SQL_HANDLE_STMT, hstmt );
    hstmt = 0;
    }
    return retn == SQL_SUCCESS;
}
```

execute 関数はクエリを実行しますが、クライアント・アプリケーションが直接使用できる結果セットは作成しません。アプリケーションは、結果セットから必要なローを明示的にフェッチします。

```
static ul_bool fetch( SQLHANDLE hstmt )
{
#define NAME_LEN 20
    SQLCHAR     fName[NAME_LEN], lName[NAME_LEN];
    SQLINTEGER   id;
    SQLINTEGER     cbID = 0, cbFName = SQL_NTS, cbLName = SQL_NTS;
    SQLRETURN    retn;

    SQLBindCol( hstmt, 1, SQL_C_ULONG, &id, 0, &cbID );
```

```
    SQLBindCol( hstmt, 2, SQL_C_CHAR,
            fName, sizeof(fName), &cbFName );
    SQLBindCol( hstmt, 3, SQL_C_CHAR,
            lName, sizeof(lName), &cbLName );
    while ( ( retn = SQLFetch( hstmt ) ) != SQL_NO_DATA ){
        if (retn == SQL_SUCCESS || retn == SQL_SUCCESS_WITH_INFO){
            fName[ cbFName ] = '¥0';
            lName[ cbLName ] = '¥0';
            _tprintf( "%20s %d %20s¥n", fName, id, lName );
        } else {
            _tprintf ( "error while fetching: %d.¥n", retn );
            break;
        }
    }
    return retn == SQL_SUCCESS;
}
```

値は、カラムにバインドされている変数にフェッチされます。文字列変数は、NULL ターミネータと一緒には返されません。このため、出力時には NULL ターミネータが追加されます。返された実際の文字列の長さは、SQLBindCol の最終パラメータで得られます。

2. main 関数から prepare、execute、fetch を呼び出します。

   sample.cpp 内の main 関数を、次のように変更します。

```
int main() {
    SQLHANDLE henv;
    SQLHANDLE hcon;
    SQLHANDLE hstmt;

    henv = opendb();
    hcon = connect( henv );
    insert( hcon );
    hstmt = prepare( hcon );
    execute( hstmt );
    fetch( hstmt );
    SQLFreeHandle( SQL_HANDLE_STMT, hstmt );
    disconnect( hcon );
    closedb( henv );
    return 0;
}
```

3. アプリケーションのコンパイル、リンク、実行を行い、アプリケーションが正しく構築されたことを確認します。

   挿入したローのデータが表示されていることを確認してください。

この手順で呼び出されている関数の詳細については、下記を参照してください。

- 「SQLPrepare 関数」 360 ページ
- 「SQLExecute 関数」 351 ページ
- 「SQLBindCol 関数」 344 ページ
- 「SQLFetch 関数」 352 ページ

これでチュートリアルが完了しました。

パート IV. API リファレンス

パート IV 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。

第 11 章

Ultra Light C/C++ 共通 API リファレンス

目次

Ultra Light C/C++共通 API の概要

この章では、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。この章で説明するほとんどの関数には、SQLCA (SQL Communications Area) が必要です。

「Ultra Light C/C++ インタフェースの共通機能」 11 ページを参照してください。

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」 178 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、「データベース・エラー・メッセージ」 『SQL Anywhere 10 - 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」 178 ページ
- 「エラー・メッセージ (Sybase エラー・コード順)」 『SQL Anywhere 10 - 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 CE 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・コンピュータと Symbian OS 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 にはプレフィクスとして **vfs:** 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」『Mobile Link - クライアント管理』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」『Mobile Link - クライアント管理』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」『Mobile Link - クライアント管理』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」『Mobile Link - クライアント管理』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」『Mobile Link - クライアント管理』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」『Mobile Link - クライアント管理』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」『Mobile Link - クライアント管理』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分も含みます。

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 の接続文字列パラメータのリファレンス」『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¥h¥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、このファイルを含める必要があります。

creation-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。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 で使用する作成時のデータベース・プロパティの選択」『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 承認の暗号化には、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 10 - 紹介』を参照してください。

参照

-
- 「ULEnableRsaFipsSyncEncryption 関数」 169 ページ

ULEnableFileDB 関数 (旧式)

Palm OS のバージョン 4.0 以降を実行するデバイスでは、ファイルベースのデータ・ストアを使用します。

旧式な関数
この関数は、バージョン 10 以降の Ultra Light では不要です。Palm OS が動作しているデバイス上のデータ・ストアのロケーションは、ファイル名で特定されます。

構文

void **ULEnableFileDB(** SQLCA * *sqlca* **);**

パラメータ

sqlca SQLCA へのポインタ。この引数は、C++ コンポーネント・アプリケーションで提供されます。

C++ コンポーネントの場合は、Sqlca.GetCA メソッドを使用します。

備考

Palm 拡張カード上のファイルベースのデータ・ストアを使用するには、Ultra Light アプリケーションから ULEnableFileDB を呼び出し、永続ストレージ・ファイル I/O モジュールをロードしてからデータベースに接続します。

例

次の Embedded SQL コードは、ULEnableFileDB の使い方を示しています。

```
db_init( & sqlca );
ULEnableFileDB( &sqlca );
// connection code here
if( SQLCODE == SQLE_CONNECTION_RESTORED ){
    // connection was restored
    // cursor is already open
} else {
    // open cursor
}
```

参照

- 「ULEnablePalmRecordDB 関数 (旧式)」 168 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 承認の暗号化には、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 10 - 紹介』を参照してください。

参照

◆

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 );
```

ULEnablePalmRecordDB 関数 (旧式)

Palm Computing Platform で稼働するデバイス上で、標準的なレコードベースのデータ・ストアを使用します。

旧式な関数
この関数は、バージョン 10 以降の Ultra Light では不要です。Palm OS が動作しているデバイス上のデータ・ストアのロケーションは、ファイル名で特定されます。

構文

void **ULEnablePalmRecordDB(** SQLCA * *sqlca* **);**

パラメータ

sqlca SQLCA へのポインタ。この引数は、C++ コンポーネントと C++ API アプリケーションの場合でも提供されます。

C++ コンポーネントの場合は、Sqlca.GetCA メソッドを使用します。

例

次の Embedded SQL コードは、ULEnablePalmRecordDB の使い方を示しています。

```
db_init( & sqlca );
ULEnablePalmRecordDB( &sqlca );
// connection code here
if( SQLCODE == SQLE_CONNECTION_RESTORED ){
    // connection was restored
    // cursor is already open
} else {
    // open cursor
}
```

参照

- 「ULEnableFileDB 関数 (旧式)」 164 ページ

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。

説明

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」 170 ページ
- 「ULEnableEccSyncEncryption 関数」 163 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」 163 ページ
- 「ULEnableRsaFipsSyncEncryption 関数」 169 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。

ULEnableUserAuthentication 関数 (旧式)

Ultra Light アプリケーションでのユーザ認証を有効にします。

旧式な関数
バージョン 10 より前の Ultra Light では、アプリケーションはこの関数を呼び出すことでユーザ認証を有効にできました。バージョン 10 以降の Ultra Light では、ユーザ認証はデフォルトで有効であり、無効にできません。

構文

void **ULEnableUserAuthentication(** 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

この関数が呼び出されていない場合、ユーザが Ultra Light データベースにアクセスするときにユーザ ID もパスワードも指定する必要はありません。この関数が呼び出された後に、アプリケーションで有効なユーザ ID とパスワードを直接指定する必要があります。Ultra Light データベースは、認証された単一のユーザ ID **DBA** とともに作成されます。このユーザ ID の初期パスワードは **sql** (小文字) です。

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。この関数は、接続を開く前に呼び出す必要があります。

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」 155 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、データが単に渡されます。

  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、「データベース・エラー・メッセージ」『SQL Anywhere 10 - 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

- 「エラー・メッセージ (Sybase エラー・コード順)」 『SQL Anywhere 10 - エラー・メッセージ』
- 「ULRegisterErrorCallback のコールバック関数」 155 ページ

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"$(VCDIR)¥include" ¥
/I"$(SQLANY10)¥h"

sample.obj: sample.cpp
   cl $(CompileOptions) $(IncludeFolders) sample.cpp
```

  VCDIR は Visual C++ ディレクトリ、*SQLANY10* は SQL Anywhere 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ダイアログ・ボックス。
- 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 のみを使用する Windows CE アプリケーションに適用されます。

参照

- 「アプリケーションへの ActiveSync 同期の追加」 106 ページ

UL_SYNC_ALL マクロ

パブリケーションにないものも含めて、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・マスクを提供します。

参照

- 「PublicationMask 同期パラメータ」 『Mobile Link - クライアント管理』
- 「ULGetLastDownloadTime 関数」 321 ページ
- 「ULCountUploadRows 関数」 317 ページ

- 「UL_SYNC_ALL_PUBS マクロ」 181 ページ

UL_SYNC_ALL_PUBS マクロ

パブリケーションに含まれる、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・マスクを提供します。

参照

- 「PublicationMask 同期パラメータ」『Mobile Link - クライアント管理』
- 「ULGetLastDownloadTime 関数」 321 ページ
- 「ULCountUploadRows 関数」 317 ページ
- 「UL_SYNC_ALL マクロ」 180 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ユニコード表現と非ユニコード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 CE 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 CE アプリケーションに適用されます。

例

```
/D UNDER_CE=$(CEVersion)
```

参照

- 「Windows CE 用 Ultra Light アプリケーションの開発」 97 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する *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」 79 ページを参照してください。

備考

Palm OS 専用のコンパイラ・ディレクティブに適用されます。

参照

- 「Palm OS の Ultra Light アプリケーションの開発」 75 ページ

第 12 章

Ultra Light C++ API リファレンス

目次

ul_synch_info_a 構造体

構文

public **ul_synch_info_a**

説明

同期データを記述するために使用される構造体です。

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。

メンバ

ul_synch_info_a 構造体のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「auth_parms 変数」 186 ページ
- 「auth_status 変数」 186 ページ
- 「auth_value 変数」 186 ページ
- 「checkpoint_store 変数」 186 ページ
- 「disable_concurrency 変数」 186 ページ
- 「download_only 変数」 187 ページ
- 「ignored_rows 変数」 187 ページ
- 「init_verify 変数」 187 ページ
- 「keep_partial_download 変数」 187 ページ
- 「new_password 変数」 187 ページ
- 「num_auth_parms 変数」 188 ページ
- 「observer 変数」 188 ページ
- 「partial_download_retained 変数」 188 ページ
- 「password 変数」 188 ページ
- 「ping 変数」 188 ページ
- 「publication 変数」 189 ページ
- 「resume_partial_download 変数」 189 ページ
- 「send_column_names 変数」 189 ページ
- 「send_download_ack 変数」 189 ページ
- 「stream 変数」 189 ページ
- 「stream_error 変数」 190 ページ
- 「stream_parms 変数」 190 ページ
- 「table_order 変数」 190 ページ
- 「upload_ok 変数」 190 ページ
- 「upload_only 変数」 190 ページ
- 「user_data 変数」 191 ページ
- 「user_name 変数」 191 ページ
- 「version 変数」 191 ページ

auth_parms 変数

構文

char ** **ul_synch_info_a::auth_parms**

説明

Mobile Link イベントの認証パラメータの配列です。

auth_status 変数

構文

ul_auth_status **ul_synch_info_a::auth_status**

説明

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 変数

構文

ul_s_long **ul_synch_info_a::auth_value**

説明

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。

checkpoint_store 変数

構文

ul_bool **ul_synch_info_a::checkpoint_store**

説明

同期中にデータベースのチェックポイントを追加して、同期プロセス中のデータベースの拡張を制限します。

disable_concurrency 変数

構文

ul_bool **ul_synch_info_a::disable_concurrency**

説明

この同期中は他のスレッドからデータベースへのアクセスを禁止します。

download_only 変数

構文

ul_bool **ul_synch_info_a::download_only**

説明

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 変数

構文

ul_bool **ul_synch_info_a::ignored_rows**

説明

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 変数

構文

ul_synch_info_a * **ul_synch_info_a::init_verify**

説明

検証を初期化します。

keep_partial_download 変数

構文

ul_bool **ul_synch_info_a::keep_partial_download**

説明

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 変数

構文

char * **ul_synch_info_a::new_password**

説明

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 変数

構文

ul_byte **ul_synch_info_a::num_auth_parms**

説明

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 変数

構文

ul_synch_observer_fn **ul_synch_info_a::observer**

説明

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。

partial_download_retained 変数

構文

ul_bool **ul_synch_info_a::partial_download_retained**

説明

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 変数

構文

char * **ul_synch_info_a::password**

説明

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 変数

構文

ul_bool **ul_synch_info_a::ping**

説明

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。

publication 変数

構文

ul_publication_mask **ul_synch_info_a::publication**

説明

最終ダウンロード時間を取得するパブリケーションを OR で結合したリストです。このセットはマスクとして提供されます。このパラメータはオプションです。

resume_partial_download 変数

構文

ul_bool **ul_synch_info_a::resume_partial_download**

説明

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ダウンロードされるはずだった変更のみがダウンロードされます。

send_column_names 変数

構文

ul_bool **ul_synch_info_a::send_column_names**

説明

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に指定します。

send_download_ack 変数

構文

ul_bool **ul_synch_info_a::send_download_ack**

説明

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 変数

構文

const char * **ul_synch_info_a::stream**

説明

同期に使用する Mobile Link ネットワーク・プロトコルです。

stream_error 変数

構文

ul_stream_error **ul_synch_info_a::stream_error**

説明

通信エラー・レポート情報を保持する構造体です。

stream_parms 変数

構文

char * **ul_synch_info_a::stream_parms**

説明

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。

table_order 変数

構文

char * **ul_synch_info_a::table_order**

説明

Ultra Light のデフォルトのテーブルの順序が適切でない場合に、優先同期に必要なテーブルの順序です。このパラメータはオプションです。

upload_ok 変数

構文

ul_bool **ul_synch_info_a::upload_ok**

説明

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 変数

構文

ul_bool **ul_synch_info_a::upload_only**

説明

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 変数

構文

ul_void * **ul_synch_info_a::user_data**

説明

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 変数

構文

char * **ul_synch_info_a::user_name**

説明

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。

version 変数

構文

char * **ul_synch_info_a::version**

説明

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。

ul_synch_info_w2 構造体

構文

public **ul_synch_info_w2**

説明

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。

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 を参照してください。

メンバ

ul_synch_info_w2 構造体のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「auth_parms 変数」 193 ページ
- 「auth_status 変数」 193 ページ
- 「auth_value 変数」 193 ページ
- 「checkpoint_store 変数」 193 ページ
- 「disable_concurrency 変数」 193 ページ
- 「download_only 変数」 194 ページ
- 「ignored_rows 変数」 194 ページ
- 「init_verify 変数」 194 ページ
- 「keep_partial_download 変数」 194 ページ
- 「new_password 変数」 194 ページ
- 「num_auth_parms 変数」 195 ページ
- 「observer 変数」 195 ページ
- 「partial_download_retained 変数」 195 ページ
- 「password 変数」 195 ページ
- 「ping 変数」 195 ページ
- 「publication 変数」 196 ページ
- 「resume_partial_download 変数」 196 ページ
- 「send_column_names 変数」 196 ページ
- 「send_download_ack 変数」 196 ページ
- 「stream 変数」 196 ページ
- 「stream_error 変数」 197 ページ
- 「stream_parms 変数」 197 ページ
- 「table_order 変数」 197 ページ
- 「upload_ok 変数」 197 ページ
- 「upload_only 変数」 197 ページ
- 「user_data 変数」 198 ページ
- 「user_name 変数」 198 ページ

- 「version 変数」 198 ページ

auth_parms 変数

構文

ul_wchar ** **ul_synch_info_w2::auth_parms**

説明

Mobile Link イベントの認証パラメータの配列です。

auth_status 変数

構文

ul_auth_status **ul_synch_info_w2::auth_status**

説明

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 変数

構文

ul_s_long **ul_synch_info_w2::auth_value**

説明

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。

checkpoint_store 変数

構文

ul_bool **ul_synch_info_w2::checkpoint_store**

説明

同期中にデータベースのチェックポイントを追加して、同期プロセス中のデータベースの拡張を制限します。

disable_concurrency 変数

構文

ul_bool **ul_synch_info_w2::disable_concurrency**

説明

この同期中は他のスレッドからデータベースへのアクセスを禁止します。

download_only 変数

構文

ul_bool **ul_synch_info_w2::download_only**

説明

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 変数

構文

ul_bool **ul_synch_info_w2::ignored_rows**

説明

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 変数

構文

ul_synch_info_w2 * **ul_synch_info_w2::init_verify**

説明

検証を初期化します。

keep_partial_download 変数

構文

ul_bool **ul_synch_info_w2::keep_partial_download**

説明

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 変数

構文

ul_wchar * **ul_synch_info_w2::new_password**

説明

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 変数

構文

ul_byte **ul_synch_info_w2::num_auth_parms**

説明

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 変数

構文

ul_synch_observer_fn **ul_synch_info_w2::observer**

説明

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。

partial_download_retained 変数

構文

ul_bool **ul_synch_info_w2::partial_download_retained**

説明

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 変数

構文

ul_wchar * **ul_synch_info_w2::password**

説明

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 変数

構文

ul_bool **ul_synch_info_w2::ping**

説明

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。

publication 変数

構文

ul_publication_mask **ul_synch_info_w2::publication**

説明

最終ダウンロード時間を取得するパブリケーションを OR で結合したリストです。このセットはマスクとして提供されます。このパラメータはオプションです。

resume_partial_download 変数

構文

ul_bool **ul_synch_info_w2::resume_partial_download**

説明

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ダウンロードされるはずだった変更のみがダウンロードされます。

send_column_names 変数

構文

ul_bool **ul_synch_info_w2::send_column_names**

説明

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に指定します。

send_download_ack 変数

構文

ul_bool **ul_synch_info_w2::send_download_ack**

説明

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 変数

構文

const char * **ul_synch_info_w2::stream**

説明

同期に使用する Mobile Link ネットワーク・プロトコルです。

stream_error 変数

構文

ul_stream_error **ul_synch_info_w2::stream_error**

説明

通信エラー・レポート情報を保持する構造体です。

stream_parms 変数

構文

ul_wchar * **ul_synch_info_w2::stream_parms**

説明

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。

table_order 変数

構文

ul_wchar * **ul_synch_info_w2::table_order**

説明

Ultra Light のデフォルトのテーブルの順序が適切でない場合に、優先同期に必要なテーブルの順序です。このパラメータはオプションです。

upload_ok 変数

構文

ul_bool **ul_synch_info_w2::upload_ok**

説明

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 変数

構文

ul_bool **ul_synch_info_w2::upload_only**

説明

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 変数

構文

ul_void * **ul_synch_info_w2::user_data**

説明

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 変数

構文

ul_wchar * **ul_synch_info_w2::user_name**

説明

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。

version 変数

構文

ul_wchar * **ul_synch_info_w2::version**

説明

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。

ul_synch_result 構造体

構文

public **ul_synch_result**

説明

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。

メンバ

ul_synch_result 構造体のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「auth_status 変数」 199 ページ
- 「auth_value 変数」 199 ページ
- 「ignored_rows 変数」 199 ページ
- 「partial_download_retained 変数」 200 ページ
- 「sql_code 変数」 200 ページ
- 「status 変数」 200 ページ
- 「stream_error 変数」 200 ページ
- 「timestamp 変数」 200 ページ
- 「upload_ok 変数」 201 ページ

auth_status 変数

構文

ul_auth_status **ul_synch_result::auth_status**

説明

同期認証ステータスです。

auth_value 変数

構文

ul_s_long **ul_synch_result::auth_value**

説明

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 変数

構文

ul_bool **ul_synch_result::ignored_rows**

説明

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 変数

構文

ul_bool **ul_synch_result::partial_download_retained**

説明

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 変数

構文

an_sql_code **ul_synch_result::sql_code**

説明

最後の同期の SQL コードです。

status 変数

構文

ul_synch_status **ul_synch_result::status**

説明

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 変数

構文

ul_stream_error **ul_synch_result::stream_error**

説明

通信ストリーム・エラー情報です。

timestamp 変数

構文

SQLDATETIME **ul_synch_result::timestamp**

説明

最後の同期の時刻と日付です。

upload_ok 変数

構文

ul_bool **ul_synch_result::upload_ok**

説明

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

構文

public **ul_synch_stats**

説明

同期ストリームの統計情報をレポートします。

メンバ

ul_synch_stats 構造体のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「bytes 変数」 202 ページ
- 「deletes 変数」 202 ページ
- 「inserts 変数」 202 ページ
- 「padding 変数」 203 ページ
- 「updates 変数」 203 ページ

bytes 変数

構文

ul_u_long **ul_synch_stats::bytes**

説明

現在までに送信されたバイト数です。

deletes 変数

構文

ul_u_short **ul_synch_stats::deletes**

説明

現在までに送信された削除済みのローの数です。

inserts 変数

構文

ul_u_short **ul_synch_stats::inserts**

説明

現在までに挿入されたローの数です。

padding 変数

構文

ul_u_short **ul_synch_stats::padding**

説明

構造を手動で調整します。

updates 変数

構文

ul_u_short **ul_synch_stats::updates**

説明

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

構文

public **ul_synch_status**

説明

同期の進行状況のモニタリング・データを返します。

メンバ

ul_synch_status 構造体のすべてのメンバ (継承されたメンバも含みます) を以下に示します。

- 「flags 変数」 204 ページ
- 「info 変数」 204 ページ
- 「received 変数」 204 ページ
- 「sent 変数」 205 ページ
- 「state 変数」 205 ページ
- 「stop 変数」 205 ページ
- 「tableCount 変数」 205 ページ
- 「tableIndex 変数」 205 ページ

flags 変数

構文

ul_u_short **ul_synch_status::flags**

説明

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 変数

構文

ul_synch_info_a * **ul_synch_status::info**

説明

ul_synch_info_a 構造体へのポインタです。ul_synch_info_a 構造体を参照してください。

received 変数

構文

ul_synch_stats **ul_synch_status::received**

説明

ダウンロードの統計情報を返します。

sent 変数

構文

ul_synch_stats **ul_synch_status::sent**

説明

アップロードの統計情報を返します。

state 変数

構文

ul_synch_state **ul_synch_status::state**

説明

サポートされている多くのステータスのひとつです。「ul_synch_state」を参照してください。

stop 変数

構文

ul_bool **ul_synch_status::stop**

説明

同期をキャンセルするブール値です。値 true は同期がキャンセルされたことを意味します。

tableCount 変数

構文

ul_u_short **ul_synch_status::tableCount**

説明

同期中のテーブルの数を返します。テーブルごとに送信と受信のフェーズがあります。したがって、この数は同期されるテーブルの数より多い場合があります。

tableIndex 変数

構文

ul_u_short **ul_synch_status::tableIndex**

説明

アップロード処理またはダウンロード処理される現在のテーブルは 0 から開始されます。この数値は、すべてのテーブルが同期されるのではない場合には、値を省略することがあります。

ULSqlca クラス

構文

public **ULSqlca**

基本クラス

- 「ULSqlcaBase クラス」 208 ページ

説明

ULSqlcaBase クラス 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。

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。

メンバ

継承されるすべてのメンバを含め、ULSqlca のすべてのメンバ

- 「Finalize 関数」 208 ページ
- 「GetCA 関数」 209 ページ
- 「GetParameter 関数」 209 ページ
- 「GetParameter 関数」 209 ページ
- 「GetParameterCount 関数」 210 ページ
- 「GetSQLCode 関数」 210 ページ
- 「GetSQLCount 関数」 211 ページ
- 「GetSQLErrorOffset 関数」 211 ページ
- 「Initialize 関数」 211 ページ
- 「LastCodeOK 関数」 212 ページ
- 「LastFetchOK 関数」 212 ページ
- 「ULSqlca 関数」 206 ページ
- 「~ULSqlca 関数」 206 ページ

ULSqlca 関数

構文

ULSqlca::ULSqlca()

説明

この関数は SQLCA コンストラクタです。

~ULSqlca 関数

構文

ULSqlca::~ULSqlca()

説明

この関数は SQLCA デストラクタです。

ULSqlcaBase クラス

構文

public **ULSqlcaBase**

派生クラス

- 「ULSqlca クラス」 206 ページ
- 「ULSqlcaWrap クラス」 213 ページ

説明

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。

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) 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。

メンバ

継承されるすべてのメンバを含め、ULSqlcaBase のすべてのメンバ

- 「Finalize 関数」 208 ページ
- 「GetCA 関数」 209 ページ
- 「GetParameter 関数」 209 ページ
- 「GetParameter 関数」 209 ページ
- 「GetParameterCount 関数」 210 ページ
- 「GetSQLCode 関数」 210 ページ
- 「GetSQLCount 関数」 211 ページ
- 「GetSQLErrorOffset 関数」 211 ページ
- 「Initialize 関数」 211 ページ
- 「LastCodeOK 関数」 212 ページ
- 「LastFetchOK 関数」 212 ページ

Finalize 関数

構文

void **ULSqlcaBase::Finalize()**

説明

この SQLCA をファイナライズします。

SQLCA は、再度初期化しないと使用できません。

GetCA 関数

構文

SQLCA * **ULSqlcaBase::GetCA()**

説明

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。

戻り値

- 未加工のの SQLCA 構造体。

GetParameter 関数

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)。

説明

エラーのパラメータ文字列を取得します。

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。

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。

GetParameter 関数

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)。

説明

エラーのパラメータ文字列を取得します。

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。

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。

GetParameterCount 関数

構文

ul_u_long **ULSqlcaBase::GetParameterCount()**

説明

最後の操作のエラー・パラメータの数を取得します。

戻り値

- 現在のエラーのパラメータの数。

GetSQLCode 関数

構文

an_sql_code **ULSqlcaBase::GetSQLCode()**

説明

最後の操作のエラー・コード (SQLCODE) を取得します。

戻り値

- sqlcode 値。

GetSQLCount 関数

構文

an_sql_code **ULSqlcaBase::GetSQLCount()**

説明

最後の操作の SQL カウント変数 (SQLCOUNT) を取得します。

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

構文

ul_s_long **ULSqlcaBase::GetSQLErrorOffset()**

説明

動的 SQL 文でのエラーのオフセットを取得します。

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

構文

bool **ULSqlcaBase::Initialize()**

説明

この SQLCA を初期化します。

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。

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。

LastCodeOK 関数

構文

bool **ULSqlcaBase::LastCodeOK()**

説明

最後の操作のエラー・コードを調べます。

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

構文

bool **ULSqlcaBase::LastFetchOK()**

説明

最後のフェッチ操作のエラー・コードを調べます。

この関数は、フェッチ操作の実行直後にのみ使用します。

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。

ULSqlcaWrap クラス

構文

public **ULSqlcaWrap**

基本クラス

- 「ULSqlcaBase クラス」 208 ページ

説明

既存の SQLCA オブジェクトにアタッチする ULSqlcaBase クラス です。

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。

メンバ

継承されるすべてのメンバを含め、ULSqlcaWrap のすべてのメンバ

- 「Finalize 関数」 208 ページ
- 「GetCA 関数」 209 ページ
- 「GetParameter 関数」 209 ページ
- 「GetParameter 関数」 209 ページ
- 「GetParameterCount 関数」 210 ページ
- 「GetSQLCode 関数」 210 ページ
- 「GetSQLCount 関数」 211 ページ
- 「GetSQLErrorOffset 関数」 211 ページ
- 「Initialize 関数」 211 ページ
- 「LastCodeOK 関数」 212 ページ
- 「LastFetchOK 関数」 212 ページ
- 「ULSqlcaWrap 関数」 213 ページ
- 「~ULSqlcaWrap 関数」 214 ページ

ULSqlcaWrap 関数

構文

```
ULSqlcaWrap::ULSqlcaWrap(
  SQLCA * sqlca
)
```

パラメータ

- **sqlca**　使用する SQLCA オブジェクト。

説明

コンストラクタです。

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 を再度呼び出さないでください。

~ULSqlcaWrap 関数

構文

ULSqlcaWrap::~ULSqlcaWrap()

説明

デストラクタです。

UltraLite_Connection クラス

構文

public **UltraLite_Connection**

基本クラス

- 「UltraLite_SQLObject_iface クラス」 265 ページ
- 「UltraLite_Connection_iface クラス」 216 ページ

説明

Ultra Light データベースへの接続を示します。

UltraLite_Connection_iface クラス

構文

public **UltraLite_Connection_iface**

派生クラス

- 「UltraLite_Connection クラス」 215 ページ

説明

接続インタフェースです。

メンバ

継承されるすべてのメンバを含め、UltraLite_Connection_iface のすべてのメンバ

- 「ChangeEncryptionKey 関数」 217 ページ
- 「Checkpoint 関数」 217 ページ
- 「Commit 関数」 217 ページ
- 「CountUploadRows 関数」 217 ページ
- 「GetConnectionNum 関数」 218 ページ
- 「GetDatabaseID 関数」 218 ページ
- 「GetDatabaseProperty 関数」 218 ページ
- 「GetLastDownloadTime 関数」 218 ページ
- 「GetLastIdentity 関数」 219 ページ
- 「GetNewUUID 関数」 219 ページ
- 「GetPublicationMask 関数」 219 ページ
- 「GetSchema 関数」 220 ページ
- 「GetSqlca 関数」 220 ページ
- 「GetSuspend 関数」 220 ページ
- 「GetSynchResult 関数」 220 ページ
- 「GetUtilityULValue 関数」 221 ページ
- 「GlobalAutoincUsage 関数」 221 ページ
- 「GrantConnectTo 関数」 221 ページ
- 「InitSynchInfo 関数」 221 ページ
- 「InitSynchInfo 関数」 222 ページ
- 「OpenTable 関数」 222 ページ
- 「OpenTableWithIndex 関数」 222 ページ
- 「PrepareStatement 関数」 223 ページ
- 「ResetLastDownloadTime 関数」 223 ページ
- 「RevokeConnectFrom 関数」 224 ページ
- 「Rollback 関数」 224 ページ
- 「RollbackPartialDownload 関数」 224 ページ
- 「SetDatabaseID 関数」 224 ページ
- 「SetDatabaseOption 関数」 225 ページ
- 「SetSuspend 関数」 225 ページ
- 「SetSynchInfo 関数」 225 ページ
- 「SetSynchInfo 関数」 226 ページ

ChangeEncryptionKey 関数

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。

説明

暗号化キーを変更します。

Checkpoint 関数

構文

bool **UltraLite_Connection_iface::Checkpoint()**

説明

データベースにチェックポイントを設定します。

Commit 関数

構文

bool **UltraLite_Connection_iface::Commit()**

説明

現在のトランザクションをコミットします。

CountUploadRows 関数

構文

ul_u_long **UltraLite_Connection_iface::CountUploadRows(**
  ul_publication_mask *mask*,
  ul_u_long *threshold*
)

パラメータ

- **mask**　対象のパブリケーションのセット。
- **threshold**　カウントするローの数の制限。

説明

アップロードする必要があるローの数を特定します。

GetConnectionNum 関数

構文

ul_connection_num **UltraLite_Connection_iface::GetConnectionNum()**

説明

接続の数を取得します。

GetDatabaseID 関数

構文

ul_u_long **UltraLite_Connection_iface::GetDatabaseID()**

説明

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。

GetDatabaseProperty 関数

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。

説明

データベースのプロパティを取得します。

戻り値

- 要求されているプロパティの値。

GetLastDownloadTime 関数

構文

bool **UltraLite_Connection_iface::GetLastDownloadTime(**
  ul_publication_mask *mask*,

```
  DECL_DATETIME * value
)
```

パラメータ

- **mask**　パブリケーション・マスク。
- **value**　前回のダウンロードの時刻。

説明

前回のダウンロードの時刻を取得します。

GetLastIdentity 関数

構文

```
ul_u_big UltraLite_Connection_iface::GetLastIdentity()
```

説明

@@identity の値を取得します。

GetNewUUID 関数

構文

```
bool UltraLite_Connection_iface::GetNewUUID(
  p_ul_binary uuid
)
```

パラメータ

- **uuid**　新しい UUID 値。

説明

新しい UUID を作成します。

GetPublicationMask 関数

構文

```
ul_publication_mask UltraLite_Connection_iface::GetPublicationMask(
  const ULValue & pub_id
)
```

パラメータ

- **pub_id**　パブリケーションの名前または序数。

説明

特定のパブリケーション ID のパブリケーション・マスクを取得します。

パブリケーション・マスクは、OR で結合されたパブリケーションの配列です。パブリケーションが見つからない場合は、0 を返します。

GetSchema 関数

構文

UltraLite_DatabaseSchema * **UltraLite_Connection_iface::GetSchema()**

説明

データベース・スキーマを取得します。

GetSqlca 関数

構文

ULSqlcaBase const & **UltraLite_Connection_iface::GetSqlca()**

説明

この接続に関連付けられている SQLCA を取得します。

GetSuspend 関数

構文

bool **UltraLite_Connection_iface::GetSuspend()**

説明

サスペンドのプロパティを取得します。

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

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 へのポインタ。

説明

前回の同期結果を取得します。

GetUtilityULValue 関数

構文

ULValue **UltraLite_Connection_iface::GetUtilityULValue()**

説明

新しい ULValue クラス を取得します。

ULValue クラス オブジェクトは、そのメソッドの多くが成功するようにするために、接続にバインドされている必要があります。

GlobalAutoincUsage 関数

構文

ul_u_short **UltraLite_Connection_iface::GlobalAutoincUsage()**

説明

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。

GrantConnectTo 関数

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。

説明

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

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。

説明

同期情報の構造体を初期化します。

InitSynchInfo 関数

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。

説明

同期情報の構造体を初期化します。

OpenTable 関数

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序数。
- **persistent_name**　サスペンド処理に使用されるインスタンスの名前。

説明

テーブルを開きます。

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。

OpenTableWithIndex 関数

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序数。
- **index_id**　インデックスの名前または序数。
- **persistent_name**　サスペンド処理に使用されるインスタンスの名前。

説明

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。

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

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。

説明

SQL 文の準備を行います。

ResetLastDownloadTime 関数

構文

```
bool UltraLite_Connection_iface::ResetLastDownloadTime(
  ul_publication_mask mask
)
```

パラメータ

- **mask**　リセットするパブリケーションのセット。

説明

前回のダウンロードの時刻をリセットします。

RevokeConnectFrom 関数

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。

説明

既存のユーザを削除します。

Rollback 関数

構文

bool **UltraLite_Connection_iface::Rollback()**

説明

現在のトランザクションをロールバックします。

RollbackPartialDownload 関数

構文

bool **UltraLite_Connection_iface::RollbackPartialDownload()**

説明

部分的なダウンロードをロールバックします。

SetDatabaseID 関数

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。

説明

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。

SetDatabaseOption 関数

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。

説明

指定されたデータベース・オプションを設定します。

SetSuspend 関数

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。

説明

サスペンドのプロパティを設定します。

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。

戻り値

- 接続がサスペンドされる場合は、true。

SetSynchInfo 関数

構文

```
bool UltraLite_Connection_iface::SetSynchInfo(
  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。

説明

以後の同期で使用できるように、ul_synch_info 構造体を現在のデータベースにアタッチします。

同期情報はデータベースに保存されます。オートコミットの実行により以前保存されたすべての同期情報が置き換えられます。null ポインタが指定された場合は、現在保存されているすべての同期情報がクリアされます。

SetSynchInfo 関数

構文

bool **UltraLite_Connection_iface::SetSynchInfo(**
  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。

説明

以後の同期で使用できるように、ul_synch_info 構造体を現在のデータベースにアタッチします。

Shutdown 関数

構文

void **UltraLite_Connection_iface::Shutdown()**

説明

この接続と、残りの関連オブジェクトを破棄します。

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

構文

bool **UltraLite_Connection_iface::StartSynchronizationDelete()**

説明

この接続の START SYNCHRONIZATION DELETE を設定します。

StopSynchronizationDelete 関数

構文

bool **UltraLite_Connection_iface::StopSynchronizationDelete()**

説明

この接続の STOP SYNCHRONIZATION DELETE を設定します。

StrToUUID 関数

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。

説明

文字列を UUID に変換します。

Synchronize 関数

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。

説明

データベースを同期します。

次に例を示します。

```
 info;ul_synch_info
conn->InitSynchInfo( &info );
info. = UL_TEXT( user_name"user_name" );
info. = UL_TEXT( version"test" );
conn->Synchronize( &info );
```

Synchronize 関数

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。

説明

データベースを同期します。

Synchronize 関数 を参照してください。

Synchronize 関数

構文

bool **UltraLite_Connection_iface::Synchronize()**

説明

SetSynchInfo 関数 によって事前にデータベースに格納された同期情報を使用して、データベースを同期します。

次に例を示します。

```
 info;ul_synch_info
conn->InitSynchInfo( &info );
info. = UL_TEXT( user_name"user_name" );
info. = UL_TEXT( version"test" );
conn.SetSynchInfo( &info );
// ...
conn->Synchronize();
```

UUIDToStr 関数

構文

bool **UltraLite_Connection_iface::UUIDToStr(**
  char * *dst*,
  size_t *len*,
  p_ul_binary *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。

説明

UUID を ANSI 文字列に変換します。

UUIDToStr 関数

構文

```
bool UltraLite_Connection_iface::UUIDToStr(
  ul_wchar * dst,
  size_t len,
  p_ul_binary src
)
```

パラメータ

- **dst**　返されるユニコード文字列。
- **len**　ul_binary 配列の長さ。
- **src**　文字列に変換される UUID 値。

説明

UUID をユニコード文字列に変換します。

UltraLite_Cursor_iface クラス

構文

public **UltraLite_Cursor_iface**

派生クラス

- 「UltraLite_ResultSet クラス」 257 ページ
- 「UltraLite_Table クラス」 272 ページ

説明

Ultra Light データベースの双方向カーソルを示します。

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。

メンバ

継承されるすべてのメンバを含め、UltraLite_Cursor_iface のすべてのメンバ

- 「AfterLast 関数」 230 ページ
- 「BeforeFirst 関数」 231 ページ
- 「Delete 関数」 231 ページ
- 「First 関数」 231 ページ
- 「Get 関数」 231 ページ
- 「GetRowCount 関数」 231 ページ
- 「GetState 関数」 232 ページ
- 「GetStreamReader 関数」 232 ページ
- 「GetStreamWriter 関数」 232 ページ
- 「GetSuspend 関数」 233 ページ
- 「IsNull 関数」 233 ページ
- 「Last 関数」 233 ページ
- 「Next 関数」 233 ページ
- 「Previous 関数」 234 ページ
- 「Relative 関数」 234 ページ
- 「Set 関数」 234 ページ
- 「SetDefault 関数」 235 ページ
- 「SetNull 関数」 235 ページ
- 「SetSuspend 関数」 235 ページ
- 「Update 関数」 236 ページ
- 「UpdateBegin 関数」 236 ページ

AfterLast 関数

構文

bool **UltraLite_Cursor_iface::AfterLast()**

説明

カーソルを最後のローの後に移動します。

BeforeFirst 関数

構文

bool **UltraLite_Cursor_iface::BeforeFirst()**

説明

カーソルを最初のローの前に移動します。

Delete 関数

構文

bool **UltraLite_Cursor_iface::Delete()**

説明

現在のローを削除し、次の有効なローに移動します。

First 関数

構文

bool **UltraLite_Cursor_iface::First()**

説明

カーソルを最初のローに移動します。

Get 関数

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序数。

説明

カラムから値をフェッチします。

GetRowCount 関数

構文

ul_u_long **UltraLite_Cursor_iface::GetRowCount()**

説明

テーブルのローの数を取得します。

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

構文

UL_RS_STATE **UltraLite_Cursor_iface::GetState()**

説明

カーソルの内部ステータスを取得します。

ulglobal.h の UL_RS_STATE 列挙を参照してください。

GetStreamReader 関数

構文

```
UltraLite_StreamReader * UltraLite_Cursor_iface::GetStreamReader(
  const ULValue & id
)
```

パラメータ

- **id**　カラム識別子。1 から始まる序数またはカラム名です。

説明

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。

GetStreamWriter 関数

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 から始まる序数またはカラム名です。

説明

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。

GetSuspend 関数

構文

bool **UltraLite_Cursor_iface::GetSuspend()**

説明

サスペンドのプロパティの値を取得します。

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

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序数。

説明

カラムが NULL であるかどうかをチェックします。

Last 関数

構文

bool **UltraLite_Cursor_iface::Last()**

説明

カーソルを最後のローに移動します。

Next 関数

構文

bool **UltraLite_Cursor_iface::Next()**

説明

カーソルをロー 1 つ分進めます。

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 になります。

Previous 関数

構文

bool **UltraLite_Cursor_iface::Previous()**

説明

カーソルをロー 1 つ分戻します。

処理が失敗すると、カーソル位置は BeforeFirst 関数 となります。

Relative 関数

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。

説明

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。

Set 関数

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 から始まる序数。
- **value**　カラムに設定される値。

説明

カラムの値を設定します。

SetDefault 関数

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 から始まる序数。

説明

カラムを、そのデフォルト値に設定します。

SetNull 関数

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 から始まる序数。

説明

カラムを null に設定します。

SetSuspend 関数

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。

説明

サスペンドのプロパティの値を設定します。

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。

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。

Update 関数

構文

bool **UltraLite_Cursor_iface::Update()**

説明

現在の行を更新します。

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 を使用します。

UpdateBegin 関数

構文

bool **UltraLite_Cursor_iface::UpdateBegin()**

説明

カラムの設定に使用される更新モードを選択します。

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。

UltraLite_DatabaseManager クラス

構文

public **UltraLite_DatabaseManager**

基本クラス

- 「UltraLite_DatabaseManager_iface クラス」 238 ページ

説明

同期リスナを管理し、Ultra Light データベースを削除できます。

UltraLite_DatabaseManager_iface クラス

構文

public **UltraLite_DatabaseManager_iface**

派生クラス

- 「UltraLite_DatabaseManager クラス」 237 ページ

説明

接続とデータベースを管理します。

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。

メンバ

継承されるすべてのメンバを含め、UltraLite_DatabaseManager_iface のすべてのメンバ

- 「CreateDatabase 関数」 238 ページ
- 「DropDatabase 関数」 239 ページ
- 「OpenConnection 関数」 239 ページ
- 「Shutdown 関数」 240 ページ

CreateDatabase 関数

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)

説明

新しいデータベースを作成します。

DropDatabase 関数

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。

説明

停止済みの既存のデータベースを消去します。

実行中のデータベースは消去できません。

OpenConnection 関数

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。

説明

既存のデータベースへの新しい接続を開きます。

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 オブジェクトを使用します。<!-- 可能性のあるエラーには、次のものがあります。※削除。9.0.2 レビュー時に Sybase に確認済み。※ -->

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

構文

```
void UltraLite_DatabaseManager_iface::Shutdown(
  ULSqlcaBase & sqlca
)
```

パラメータ

- **sqlca**　初期化された sqlca。

説明

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。

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)。

UltraLite_DatabaseSchema クラス

構文

public **UltraLite_DatabaseSchema**

基本クラス

- 「UltraLite_SQLObject_iface クラス」 265 ページ
- 「UltraLite_DatabaseSchema_iface クラス」 242 ページ

説明

Ultra Light データベースのスキーマを表します。

UltraLite_DatabaseSchema_iface クラス

構文

public **UltraLite_DatabaseSchema_iface**

派生クラス

- 「UltraLite_DatabaseSchema クラス」 241 ページ

説明

DatabaseSchema インタフェースです。

メンバ

継承されるすべてのメンバを含め、UltraLite_DatabaseSchema_iface のすべてのメンバ

- 「GetCollationName 関数」 242 ページ
- 「GetPublicationCount 関数」 242 ページ
- 「GetPublicationID 関数」 243 ページ
- 「GetPublicationMask 関数」 243 ページ
- 「GetPublicationName 関数」 243 ページ
- 「GetTableCount 関数」 244 ページ
- 「GetTableName 関数」 244 ページ
- 「GetTableSchema 関数」 245 ページ
- 「IsCaseSensitive 関数」 245 ページ

GetCollationName 関数

構文

ULValue **UltraLite_DatabaseSchema_iface::GetCollationName()**

説明

現在の照合順の名前を取得します。

戻り値

- 文字列を含む ULValue クラス。

GetPublicationCount 関数

構文

ul_publication_count **UltraLite_DatabaseSchema_iface::GetPublicationCount()**

説明

データベース内のパブリケーション数を取得します。

パブリケーション ID の範囲は、1 ー GetPublicationCount 関数 です。

GetPublicationID 関数

構文

```
ul_u_short UltraLite_DatabaseSchema_iface::GetPublicationID(
  const ULValue & pub_id
)
```

パラメータ

- **pub_id**　1 から始まる序数。

説明

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。

GetPublicationMask 関数

構文

```
ul_publication_mask UltraLite_DatabaseSchema_iface::GetPublicationMask(
  const ULValue & pub_id
)
```

パラメータ

- **pub_id**　1 から始まる序数。

説明

特定のパブリケーション名のパブリケーション・マスクを取得します。

マスクは、パブリケーションのセットを定義します。セットは、個々のパブリケーションの論理和でグループとして作成されます。パブリケーション・マスクは、パブリケーション ID ではありません。

戻り値

- パブリケーションが見つからない場合は、0。

GetPublicationName 関数

構文

```
ULValue UltraLite_DatabaseSchema_iface::GetPublicationName(
  const ULValue & pub_id
)
```

パラメータ

- **pub_id**　1 から始まる序数。

説明

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。

パブリケーション・マスクは、パブリケーション ID ではありません。マスクは、パブリケーションのセットを定義します。セットは、個々のパブリケーションの論理和でグループとして作成されます。

GetTableCount 関数

構文

ul_table_num **UltraLite_DatabaseSchema_iface::GetTableCount()**

説明

データベース内のテーブルの数を返します。

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

構文

ULValue **UltraLite_DatabaseSchema_iface::GetTableName(**
  ul_table_num *tableID*
)

パラメータ

- **tableID**　1 から始まる序数。

説明

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。

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 オブジェクトは空です。

戻り値

- 指定したテーブル ID で識別されたテーブルの名前。

GetTableSchema 関数

構文

UltraLite_TableSchema * **UltraLite_DatabaseSchema_iface::GetTableSchema(**
  const ULValue & *table_id*
)

パラメータ

- **table_id**　1 から始まる序数。

説明

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。

戻り値

- テーブルが存在しない場合は、UL_NULL。

IsCaseSensitive 関数

構文

bool **UltraLite_DatabaseSchema_iface::IsCaseSensitive()**

説明

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。

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。

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。

UltraLite_IndexSchema クラス

構文

public **UltraLite_IndexSchema**

基本クラス

- 「UltraLite_SQLObject_iface クラス」 265 ページ
- 「UltraLite_IndexSchema_iface クラス」 247 ページ

説明

Ultra Light テーブルのインデックスのスキーマを表します。

UltraLite_IndexSchema_iface クラス

構文

public **UltraLite_IndexSchema_iface**

派生クラス

- 「UltraLite_IndexSchema クラス」 246 ページ

説明

IndexSchema インタフェースを表します。

メンバ

継承されるすべてのメンバを含め、UltraLite_IndexSchema_iface のすべてのメンバ

- 「GetColumnCount 関数」 247 ページ
- 「GetColumnName 関数」 247 ページ
- 「GetID 関数」 248 ページ
- 「GetName 関数」 248 ページ
- 「GetReferencedIndexName 関数」 248 ページ
- 「GetReferencedTableName 関数」 249 ページ
- 「GetTableName 関数」 249 ページ
- 「IsColumnDescending 関数」 249 ページ
- 「IsForeignKey 関数」 249 ページ
- 「IsForeignKeyCheckOnCommit 関数」 250 ページ
- 「IsForeignKeyNullable 関数」 250 ページ
- 「IsPrimaryKey 関数」 250 ページ
- 「IsUniqueIndex 関数」 251 ページ
- 「IsUniqueKey 関数」 251 ページ

GetColumnCount 関数

構文

ul_column_num **UltraLite_IndexSchema_iface::GetColumnCount()**

説明

インデックス内のカラム数を取得します。

GetColumnName 関数

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 から始まる序数。

説明

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。

戻り値

- カラムが存在しない場合は、空の ULValue クラス オブジェクト。
- カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND。

GetID 関数

構文

ul_index_num **UltraLite_IndexSchema_iface::GetID()**

説明

インデックスの ID を取得します。

戻り値

- インデックスの ID。

GetName 関数

構文

ULValue **UltraLite_IndexSchema_iface::GetName()**

説明

インデックスの名前を取得します。

GetReferencedIndexName 関数

構文

ULValue **UltraLite_IndexSchema_iface::GetReferencedIndexName()**

説明

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。

この関数は、外部キー専用です。

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 オブジェクト。

GetReferencedTableName 関数

構文

ULValue **UltraLite_IndexSchema_iface::GetReferencedTableName()**

説明

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。

このメソッドは、外部キー専用です。

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 オブジェクト。

GetTableName 関数

構文

ULValue **UltraLite_IndexSchema_iface::GetTableName()**

説明

インデックスが含まれるテーブルの名前を取得します。

IsColumnDescending 関数

構文

bool **UltraLite_IndexSchema_iface::IsColumnDescending(**
  const ULValue & *column_name*
)

パラメータ

- **column_name**　カラムの名前。

説明

カラムが降順かどうかを調べます。

戻り値

- カラムが降順の場合は、true を返します。
- カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND を設定します。

IsForeignKey 関数

構文

bool **UltraLite_IndexSchema_iface::IsForeignKey()**

説明

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。

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。

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。

IsForeignKeyCheckOnCommit 関数

構文

bool **UltraLite_IndexSchema_iface::IsForeignKeyCheckOnCommit()**

説明

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。

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

構文

bool **UltraLite_IndexSchema_iface::IsForeignKeyNullable()**

説明

外部キーが null 入力可能であるかどうかをチェックします。

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 入力可能でない場合は、false。

IsPrimaryKey 関数

構文

bool **UltraLite_IndexSchema_iface::IsPrimaryKey()**

説明

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。

プライマリ・キー内のカラムでは null は許可されません。

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。

IsUniqueIndex 関数

構文

bool **UltraLite_IndexSchema_iface::IsUniqueIndex()**

説明

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。

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

構文

bool **UltraLite_IndexSchema_iface::IsUniqueKey()**

説明

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。

ユニーク・キー内のカラムでは null は許可されません。

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。

UltraLite_PreparedStatement クラス

構文

public **UltraLite_PreparedStatement**

基本クラス

- 「UltraLite_SQLObject_iface クラス」 265 ページ
- 「UltraLite_PreparedStatement_iface クラス」 253 ページ

説明

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。

UltraLite_PreparedStatement_iface クラス

構文

public **UltraLite_PreparedStatement_iface**

派生クラス

- 「UltraLite_PreparedStatement クラス」 252 ページ

説明

PreparedStatement インタフェースです。

メンバ

継承されるすべてのメンバを含め、UltraLite_PreparedStatement_iface のすべてのメンバ

- 「ExecuteQuery 関数」 253 ページ
- 「ExecuteStatement 関数」 253 ページ
- 「GetPlan 関数」 254 ページ
- 「GetPlan 関数」 254 ページ
- 「GetSchema 関数」 254 ページ
- 「GetStreamWriter 関数」 255 ページ
- 「HasResultSet 関数」 255 ページ
- 「SetParameter 関数」 255 ページ
- 「SetParameterNull 関数」 256 ページ

ExecuteQuery 関数

構文

UltraLite_ResultSet * **UltraLite_PreparedStatement_iface::ExecuteQuery()**

説明

SQL SELECT 文をクエリとして実行します。

戻り値

- クエリの結果セット (ローのセット)。

ExecuteStatement 関数

構文

ul_s_long **UltraLite_PreparedStatement_iface::ExecuteStatement()**

説明

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。

GetPlan 関数

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)。

説明

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。

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。この関数は、主に開発中の使用を目的とします。

GetPlan 関数

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)。

説明

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。

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。この関数は、主に開発中の使用を目的とします。

GetSchema 関数

構文

UltraLite_ResultSetSchema * **UltraLite_PreparedStatement_iface::GetSchema()**

説明

結果セットのスキーマを取得します。

GetStreamWriter 関数

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 から始まる序数またはカラム名です。

説明

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。

HasResultSet 関数

構文

```
bool UltraLite_PreparedStatement_iface::HasResultSet()
```

説明

SQL 文に結果セットがあるかどうかを調べます。

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

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序数。
- **value**　パラメータを設定する値。

説明

SQL 文のパラメータを設定します。

SetParameterNull 関数

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序数。

説明

パラメータを null に設定します。

UltraLite_ResultSet クラス

構文

public **UltraLite_ResultSet**

基本クラス

- 「UltraLite_SQLObject_iface クラス」 265 ページ
- 「UltraLite_ResultSet_iface クラス」 258 ページ
- 「UltraLite_Cursor_iface クラス」 230 ページ

説明

Ultra Light データベースの編集可能な結果セットを示します。

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

構文

public **UltraLite_ResultSet_iface**

派生クラス

- 「UltraLite_ResultSet クラス」 257 ページ

説明

ResultSet インタフェースです。

メンバ

継承されるすべてのメンバを含め、UltraLite_ResultSet_iface のすべてのメンバ

- 「DeleteNamed 関数」 258 ページ
- 「GetSchema 関数」 258 ページ

DeleteNamed 関数

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)。

説明

現在のローを削除し、次の有効なローに移動します。

GetSchema 関数

構文

UltraLite_ResultSetSchema * **UltraLite_ResultSet_iface::GetSchema()**

説明

この結果セットのスキーマを取得します。

UltraLite_ResultSetSchema クラス

構文

public **UltraLite_ResultSetSchema**

基本クラス

- 「UltraLite_SQLObject_iface クラス」 265 ページ
- 「UltraLite_RowSchema_iface クラス」 260 ページ

説明

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。

UltraLite_RowSchema_iface クラス

構文

public **UltraLite_RowSchema_iface**

派生クラス

- 「UltraLite_ResultSetSchema クラス」 259 ページ
- 「UltraLite_TableSchema クラス」 279 ページ

説明

RowSchema インタフェースです。

メンバ

継承されるすべてのメンバを含め、UltraLite_RowSchema_iface のすべてのメンバ

- 「GetBaseColumnName 関数」 260 ページ
- 「GetColumnCount 関数」 261 ページ
- 「GetColumnID 関数」 261 ページ
- 「GetColumnName 関数」 261 ページ
- 「GetColumnPrecision 関数」 262 ページ
- 「GetColumnScale 関数」 263 ページ
- 「GetColumnSize 関数」 263 ページ
- 「GetColumnSQLName 関数」 262 ページ
- 「GetColumnSQLType 関数」 263 ページ
- 「GetColumnType 関数」 264 ページ

GetBaseColumnName 関数

構文

ULValue **UltraLite_RowSchema_iface::GetBaseColumnName(**
  ul_column_num *column_id*
)

パラメータ

- **column_id**　1 から始まる序数。

説明

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。

戻り値

- 結合された ULValue クラス オブジェクトを返します。
- 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。

- 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

構文

ul_column_num **UltraLite_RowSchema_iface::GetColumnCount()**

説明

テーブル内のカラム数を取得します。

GetColumnID 関数

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。

説明

1 から始まるカラム ID を取得します。

戻り値

- カラムが存在しない場合は、0 を返します。
- カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND を設定します。

GetColumnName 関数

構文

```
ULValue UltraLite_RowSchema_iface::GetColumnName(
  ul_column_num column_id
)
```

パラメータ

- **column_id**　1 から始まる序数。

説明

1 から始まる ID を指定してカラムの名前を取得します。

これは SELECT 文のエイリアスまたは相関名になります。

戻り値

- カラムが存在しない場合は、空の ULValue クラス オブジェクトを返します。

- 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

構文

```
size_t UltraLite_RowSchema_iface::GetColumnPrecision(
  const ULValue & column_id
)
```

パラメータ

- **column_id**　1 から始まる序数。

説明

数値カラムの精度を取得します。

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。
- 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

構文

```
ULValue UltraLite_RowSchema_iface::GetColumnSQLName(
  ul_column_num column_id
)
```

パラメータ

- **column_id**　1 から始まる序数。

説明

結果セット内のカラムの SQL 名を取得します。

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。

戻り値

- 結合された ULValue クラス オブジェクトを返します。
- 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

構文

ul_column_sql_type **UltraLite_RowSchema_iface::GetColumnSQLType(**
  const ULValue & *column_id*
)

パラメータ

- **column_id**　1 から始まる序数。

説明

カラムの SQL の型を取得します。

ulprotos.h の ul_column_sql_type を参照してください。

戻り値

- カラムが存在しない場合は、UL_SQLTYPE_BAD_INDEX。

GetColumnScale 関数

構文

size_t **UltraLite_RowSchema_iface::GetColumnScale(**
  const ULValue & *column_id*
)

パラメータ

- **column_id**　1 から始まる序数。

説明

数値カラムの位取りを取得します。

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。
- 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

構文

size_t **UltraLite_RowSchema_iface::GetColumnSize(**
  const ULValue & *column_id*
)

パラメータ

- **column_id**　1 から始まる序数。

説明

カラムのサイズを取得します。

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。
- 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

構文

```
ul_column_storage_type UltraLite_RowSchema_iface::GetColumnType(
  const ULValue & column_id
)
```

パラメータ

- **column_id**　1 から始まる序数。

説明

カラムの型を取得します。

ulprotos.h の ul_column_storage_type enum を参照してください。

戻り値

- カラムが存在しない場合は、UL_TYPE_BAD_INDEX。

UltraLite_SQLObject_iface クラス

構文

public **UltraLite_SQLObject_iface**

派生クラス

- 「UltraLite_Connection クラス」 215 ページ
- 「UltraLite_DatabaseSchema クラス」 241 ページ
- 「UltraLite_IndexSchema クラス」 246 ページ
- 「UltraLite_PreparedStatement クラス」 252 ページ
- 「UltraLite_ResultSet クラス」 257 ページ
- 「UltraLite_ResultSetSchema クラス」 259 ページ
- 「UltraLite_StreamReader クラス」 267 ページ
- 「UltraLite_StreamWriter クラス」 271 ページ
- 「UltraLite_Table クラス」 272 ページ
- 「UltraLite_TableSchema クラス」 279 ページ

説明

SQLObject インタフェース

メンバ

継承されるすべてのメンバを含め、UltraLite_SQLObject_iface のすべてのメンバ

- 「AddRef 関数」 265 ページ
- 「GetConnection 関数」 265 ページ
- 「GetIFace 関数」 266 ページ
- 「Release 関数」 266 ページ

AddRef 関数

構文

ul_ret_void **UltraLite_SQLObject_iface::AddRef()**

説明

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。

オブジェクトを解放するには、この関数の呼び出しと Release 関数 の呼び出しを対にする必要があります。

GetConnection 関数

構文

UltraLite_Connection * **UltraLite_SQLObject_iface::GetConnection()**

説明

Connection オブジェクトを取得します。

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

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。

説明

今後の使用のために予約されています。

Release 関数

構文

```
ul_u_long UltraLite_SQLObject_iface::Release()
```

説明

オブジェクトへの参照を解放します。

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 を使用する場合は、それぞれの AddRef 関数 と対になるようにして呼び出す必要もあります。

UltraLite_StreamReader クラス

構文

public **UltraLite_StreamReader**

基本クラス

- 「UltraLite_SQLObject_iface クラス」 265 ページ
- 「UltraLite_StreamReader_iface クラス」 268 ページ

説明

Ultra Light StreamReader を表します。

UltraLite_StreamReader_iface クラス

構文

public **UltraLite_StreamReader_iface**

派生クラス

- 「UltraLite_StreamReader クラス」 267 ページ

説明

StreamReader インタフェースです。

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。

メンバ

継承されるすべてのメンバを含め、UltraLite_StreamReader_iface のすべてのメンバ

- 「GetByteChunk 関数」 268 ページ
- 「GetLength 関数」 269 ページ
- 「GetStringChunk 関数」 269 ページ
- 「GetStringChunk 関数」 270 ページ
- 「SetReadPosition 関数」 270 ページ

GetByteChunk 関数

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。

説明

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。

SetReadPosition 関数 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

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。

説明

文字列またはバイナリの値の長さを取得します。

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

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。

説明

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。

SetReadPosition 関数 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

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。

説明

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。

SetReadPosition 関数 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

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。

説明

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。

UltraLite_StreamWriter クラス

構文

public **UltraLite_StreamWriter**

基本クラス

- 「UltraLite_SQLObject_iface クラス」 265 ページ

説明

Ultra Light StreamWriter を表します。

UltraLite_Table クラス

構文

public **UltraLite_Table**

基本クラス

- 「UltraLite_SQLObject_iface クラス」 265 ページ
- 「UltraLite_Table_iface クラス」 273 ページ
- 「UltraLite_Cursor_iface クラス」 230 ページ

説明

Ultra Light データベース内のテーブルを示します。

UltraLite_Table_iface クラス

構文

public **UltraLite_Table_iface**

派生クラス

- 「UltraLite_Table クラス」 272 ページ

説明

テーブル・インタフェースを表します。

メンバ

継承されるすべてのメンバを含め、UltraLite_Table_iface のすべてのメンバ

- 「DeleteAllRows 関数」 273 ページ
- 「Find 関数」 274 ページ
- 「FindBegin 関数」 274 ページ
- 「FindFirst 関数」 274 ページ
- 「FindLast 関数」 275 ページ
- 「FindNext 関数」 275 ページ
- 「FindPrevious 関数」 276 ページ
- 「GetSchema 関数」 276 ページ
- 「Insert 関数」 276 ページ
- 「InsertBegin 関数」 276 ページ
- 「Lookup 関数」 277 ページ
- 「LookupBackward 関数」 277 ページ
- 「LookupBegin 関数」 277 ページ
- 「LookupForward 関数」 278 ページ
- 「TruncateTable 関数」 278 ページ

DeleteAllRows 関数

構文

bool **UltraLite_Table_iface::DeleteAllRows()**

説明

すべてのローをテーブルから削除します。

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。

注意：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、DeleteAllRows 関数が呼び出された後に、別の接続が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。

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。

Find 関数

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

説明

この関数は、FindFirst 関数 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。

FindBegin 関数

構文

```
bool UltraLite_Table_iface::FindBegin()
```

説明

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。

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。

FindFirst 関数

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

説明

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。

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

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

説明

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。

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

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

説明

インデックスに完全に一致する次のローを取得します。

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、カーソルは最後のローの後ろに配置されます。

FindPrevious 関数

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

説明

インデックスに完全に一致する前のローを取得します。

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

構文

UltraLite_TableSchema * **UltraLite_Table_iface::GetSchema()**

説明

このテーブルのスキーマ・オブジェクトを取得します。

Insert 関数

構文

bool **UltraLite_Table_iface::Insert()**

説明

新しいローをテーブルに挿入します。

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 を使用します。

InsertBegin 関数

構文

bool **UltraLite_Table_iface::InsertBegin()**

説明

設定されたカラムに対する挿入モードを選択します。

このモードでは、すべてのカラムを修正できます。

Lookup 関数

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

説明

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。

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

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

説明

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。

検索する値を指定するには、インデックスの各カラムごとに値を設定します。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。

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。

LookupBegin 関数

構文

bool **UltraLite_Table_iface::LookupBegin()**

説明

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。

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。

LookupForward 関数

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

説明

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。

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。

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。

TruncateTable 関数

構文

bool **UltraLite_Table_iface::TruncateTable()**

説明

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。

UltraLite_TableSchema クラス

構文

public **UltraLite_TableSchema**

基本クラス

- 「UltraLite_SQLObject_iface クラス」 265 ページ
- 「UltraLite_TableSchema_iface クラス」 280 ページ
- 「UltraLite_RowSchema_iface クラス」 260 ページ

説明

テーブル・スキーマを表します。

UltraLite_TableSchema_iface クラス

構文

public **UltraLite_TableSchema_iface**

派生クラス

- 「UltraLite_TableSchema クラス」 279 ページ

説明

TableSchema インタフェースです。

メンバ

継承されるすべてのメンバを含め、UltraLite_TableSchema_iface のすべてのメンバ

- 「GetColumnDefault 関数」 280 ページ
- 「GetGlobalAutoincPartitionSize 関数」 281 ページ
- 「GetID 関数」 281 ページ
- 「GetIndexCount 関数」 282 ページ
- 「GetIndexName 関数」 282 ページ
- 「GetIndexSchema 関数」 282 ページ
- 「GetName 関数」 283 ページ
- 「GetOptimalIndex 関数」 283 ページ
- 「GetPrimaryKey 関数」 283 ページ
- 「GetPublicationPredicate 関数」 284 ページ
- 「GetUploadUnchangedRows 関数」 284 ページ
- 「InPublication 関数」 284 ページ
- 「IsColumnAutoinc 関数」 285 ページ
- 「IsColumnCurrentDate 関数」 285 ページ
- 「IsColumnCurrentTime 関数」 286 ページ
- 「IsColumnCurrentTimestamp 関数」 286 ページ
- 「IsColumnGlobalAutoinc 関数」 287 ページ
- 「IsColumnInIndex 関数」 287 ページ
- 「IsColumnNewUUID 関数」 288 ページ
- 「IsColumnNullable 関数」 288 ページ
- 「IsNeverSynchronized 関数」 289 ページ

GetColumnDefault 関数

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。

説明

カラムのデフォルト値が存在する場合は取得します。

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。カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND を設定します。

返された ULValue クラス オブジェクトには、次の項目があります。

GetGlobalAutoincPartitionSize 関数

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。

説明

分割サイズを取得します。

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

構文

```
ul_table_num UltraLite_TableSchema_iface::GetID()
```

説明

テーブル ID を取得します。

GetIndexCount 関数

構文

ul_index_num **UltraLite_TableSchema_iface::GetIndexCount()**

説明

テーブル内のインデックス数を取得します。

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。

戻り値

- テーブル内のインデックス数

GetIndexName 関数

構文

ULValue **UltraLite_TableSchema_iface::GetIndexName(**
  ul_index_num *index_id*
)

パラメータ

- **index_id**　1 から始まる序数

説明

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。

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。

戻り値

- インデックスが存在しない場合、ULValue クラス オブジェクトは空になります。

GetIndexSchema 関数

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

説明

指定した名前または ID で IndexSchema オブジェクトを取得します。

戻り値

- インデックスが存在しない場合は、UL_NULL

GetName 関数

構文

ULValue **UltraLite_TableSchema_iface::GetName()**

説明

テーブルの名前を取得します。

戻り値

- 文字列としてのテーブル名

GetOptimalIndex 関数

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。

説明

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。

戻り値

インデックス名

GetPrimaryKey 関数

構文

UltraLite_IndexSchema * **UltraLite_TableSchema_iface::GetPrimaryKey()**

説明

テーブルのプライマリ・キーを取得します。

GetPublicationPredicate 関数

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

説明

文字列としてのパブリケーション述部を取得します。

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
- 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

構文

bool **UltraLite_TableSchema_iface::GetUploadUnchangedRows()**

説明

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。

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。

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。

InPublication 関数

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

説明

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。

戻り値

- テーブルがパブリケーションに含まれる場合は、true を返します。
- テーブルがパブリケーションに含まれない場合は、false を返します。
- 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

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。

説明

指定し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。

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 を返します。
- カラムがオートインクリメントでない場合は、false を返します。
- カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND を設定します。

IsColumnCurrentDate 関数

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。

説明

指定し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。

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

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。

説明

指定し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。

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

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。

説明

指定し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。

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

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。

説明

指定し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。

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。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

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 から始まる序数。column_id を取得するには、GetColumnCount 関数 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 を呼び出します。

説明

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。

戻り値

- カラムがインデックスに含まれる場合は、true を返します。
- カラムがインデックスに含まれない場合は、false を返します。
- カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND を設定します。
- 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

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。

説明

指定し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。

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

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。指定したカラムは、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。

説明

指定したカラムが null 入力可能であるかどうかをチェックします。

戻り値

- カラムが null 入力可能の場合は、true を返します。
- null 入力可能でない場合は、false を返します。
- カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND を設定します。

IsNeverSynchronized 関数

構文

bool **UltraLite_TableSchema_iface::IsNeverSynchronized()**

説明

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。

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**

説明

ULValue クラス です。

ULValue クラス 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 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 を明示的にインスタンス化しなくても、ULValue クラス 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築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 オブジェクトの構築はコンパイラによって自動的に行われるため、明示的に構築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 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 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 オブジェクトに自動的に変換されます。

メンバ

継承されるすべてのメンバを含め、ULValue のすべてのメンバ

- 「GetBinary 関数」 292 ページ
- 「GetBinary 関数」 292 ページ
- 「GetBinaryLength 関数」 293 ページ
- 「GetCombinedStringItem 関数」 293 ページ
- 「GetCombinedStringItem 関数」 293 ページ
- 「GetString 関数」 294 ページ
- 「GetString 関数」 294 ページ

GetBinary 関数

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。

説明

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、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

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。

説明

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。

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

構文

size_t **ULValue::GetBinaryLength()**

説明

バイナリ値の長さを取得します。

戻り値

- GetBinary 関数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

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。

説明

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。

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

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。

説明

結合された文字列値の選択した部分を取得します。

GetString 関数

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。

説明

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。

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

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。

説明

文字列値を取得します。

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。

GetStringLength 関数

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。

説明

文字列の長さを取得します。

使用方法を次に示します。

```
len = v.GetStringLength();
dst = new char[ len ];
( dst, len ); 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 ];
( dst, len ); GetString
```

戻り値

- GetString 関数 メソッドのいずれかによって返される文字列を保持するために必要な、null ターミネータを含むバイト数または文字数。

InDatabase 関数

構文

bool **ULValue::InDatabase()**

説明

値がデータベース内にあるかどうかをチェックします。

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

構文

bool **ULValue::IsNull()**

説明

ULValue クラス オブジェクトが空であるかどうかをチェックします。

戻り値

- オブジェクトが、空の ULValue クラス 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

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。

説明

指定したバイナリ・バッファを参照する値を設定します。

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

構文

```
void ULValue::SetString(
  const char * val,
  size_t len
)
```

パラメータ

- **val** この ULValue クラス 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。

説明

ULValue クラス を文字列にキャストします。

SetString 関数

構文

```
void ULValue::SetString(
  const ul_wchar * val,
  size_t len
)
```

パラメータ

- **val**　この ULValue クラス の null で終了されたユニコード文字列表現へのポインタ。
- **len**　文字列の長さ。

説明

ULValue クラス をユニコード文字列にキャストします。

StringCompare 関数

構文

```
ul_compare ULValue::StringCompare(
  const ULValue & value
)
```

パラメータ

- **value**　比較文字列。

説明

文字列、または ULValue クラス オブジェクトの文字列表現を比較します。

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 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 オブジェクトで文字列表現が UL_NULL の場合は、-2。

ULValue 関数

構文

```
ULValue::ULValue()
```

説明

ULValue クラス を構成します。

ULValue 関数

構文

```
ULValue::ULValue(
 const ULValue & vSrc
)
```

パラメータ

- **vSrc**　ULValue クラス として扱われる値。

説明

既存の ULValue クラス からコピーして構築します。

ULValue 関数

構文

```
ULValue::ULValue(
 bool val
)
```

パラメータ

- **val**　ULValue クラス として扱われるブール値。

説明

bool から ULValue クラス を構成します。

ULValue 関数

構文

```
ULValue::ULValue(
 short val
)
```

パラメータ

- **val**　ULValue クラス として扱われる short 値。

説明

short から ULValue クラス を構成します。

ULValue 関数

構文

```
ULValue::ULValue(
 long val
)
```

パラメータ

- **val**　ULValue クラス として扱われる long 値。

説明

long から ULValue クラス を構成します。

ULValue 関数

構文

```
ULValue::ULValue(
 int val
)
```

パラメータ

- **val**　ULValue クラス として扱われる INTEGER 値。

説明

int から ULValue クラス を構成します。

ULValue 関数

構文

```
ULValue::ULValue(
 unsigned int val
)
```

パラメータ

- **val**　ULValue クラス として扱われる unsigned INTEGER 値。

説明

unsigned INTEGER から ULValue クラス を構成します。

ULValue 関数

構文

```
ULValue::ULValue(
 float val
)
```

パラメータ

- **val**　ULValue クラス として扱われる FLOAT 値。

説明

FLOAT から ULValue クラス を構成します。

ULValue 関数

構文

```
ULValue::ULValue(
  double val
)
```

パラメータ

- **val**　ULValue クラス として扱われる DOUBLE 値。

説明

DOUBLE から ULValue クラス を構成します。

ULValue 関数

構文

```
ULValue::ULValue(
  unsigned char val
)
```

パラメータ

- **val**　ULValue クラス として扱われる unsigned CHAR 値。

説明

unsigned CHAR から ULValue クラス を構成します。

ULValue 関数

構文

```
ULValue::ULValue(
  unsigned short val
)
```

パラメータ

- **val**　ULValue クラス として扱われる unsigned SHORT 値。

説明

unsigned SHORT から ULValue クラス を構成します。

ULValue 関数

構文

```
ULValue::ULValue(
  unsigned long val
)
```

パラメータ

- **val**　ULValue クラス として扱われる unsigned LONG 値。

説明

unsigned LONG から ULValue クラス を構成します。

ULValue 関数

構文

```
ULValue::ULValue(
  const ul_u_big & val
)
```

パラメータ

- **val**　ULValue クラス として扱われる ul_u_big 値。

説明

ul_u_big から ULValue クラス を構成します。

ULValue 関数

構文

```
ULValue::ULValue(
  const ul_s_big & val
)
```

パラメータ

- **val**　ULValue クラス として扱われる ul_s_big 値。

説明

ul_s_big から ULValue クラス を構成します。

ULValue 関数

構文

```
ULValue::ULValue(
  const p_ul_binary val
)
```

パラメータ

- **val**　ULValue クラス として扱われる ul_binary 値。

説明

ul_binary から ULValue クラス を構成します。

ULValue 関数

構文

```
ULValue::ULValue(
 DECL_DATETIME & val
)
```

パラメータ

- **val** ULValue クラス として扱われる DATETIME 値。

説明

datetime から ULValue クラス を構成します。

ULValue 関数

構文

```
ULValue::ULValue(
 const char * val
)
```

パラメータ

- **val** ULValue クラス として扱われる文字列へのポインタ。

説明

STRING から ULValue クラス を構成します。

ULValue 関数

構文

```
ULValue::ULValue(
 const ul_wchar * val
)
```

パラメータ

- **val** ULValue クラス として扱われるユニコード文字列へのポインタ。

説明

ユニコード文字列から ULValue クラス を構成します。

ULValue 関数

構文

```
ULValue::ULValue(
 const char * val,
 size_t len
)
```

パラメータ

- **val**　ULValue クラス 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。

説明

文字のバッファから ULValue クラス を構成します。

ULValue 関数

構文

```
ULValue::ULValue(
 const ul_wchar * val,
 size_t len
)
```

パラメータ

- **val**　ULValue クラス 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。

説明

ユニコード文字のバッファから ULValue クラス を構成します。

operator DECL_DATETIME 関数

構文

```
ULValue::operator DECL_DATETIME()
```

説明

ULValue クラス を datetime にキャストします。

operator bool 関数

構文

```
ULValue::operator bool()
```

説明

ULValue クラス を bool にキャストします。

operator double 関数

構文

```
ULValue::operator double()
```

説明

ULValue クラス を double にキャストします。

operator float 関数

構文

ULValue::operator float()

説明

ULValue クラス を float にキャストします。

operator int 関数

構文

ULValue::operator int()

説明

ULValue クラス を int にキャストします。

operator long 関数

構文

ULValue::operator long()

説明

ULValue クラス を long にキャストします。

operator short 関数

構文

ULValue::operator short()

説明

ULValue クラス を short にキャストします。

operator ul_s_big 関数

構文

ULValue::operator ul_s_big()

説明

ULValue クラス を signed big int にキャストします。

operator ul_u_big 関数

構文

ULValue::operator ul_u_big()

説明

ULValue クラス を unsigned big int にキャストします。

operator unsigned char 関数

構文

ULValue::operator unsigned char()

説明

ULValue クラス を char にキャストします。

operator unsigned int 関数

構文

ULValue::operator unsigned int()

説明

ULValue クラス を unsigned int にキャストします。

operator unsigned long 関数

構文

ULValue::operator unsigned long()

説明

ULValue クラス を unsigned long にキャストします。

operator unsigned short 関数

構文

ULValue::operator unsigned short()

説明

ULValue クラス を unsigned short にキャストします。

operator= 関数

構文

ULValue & **ULValue::operator=(**
  const ULValue & *other*
)

パラメータ

- **other**　ULValue クラス に割り当てられる値。

説明

ULValue の = 演算子を無効にします。

~ULValue 関数

構文

ULValue::~ULValue()

説明

ULValue クラス のデストラクタです。

第 13 章

Embedded SQL API リファレンス

目次

Embedded SQL API の概要

この章では、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」 37 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」 311 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 (たとえば継続保管の初期化時)。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」 310 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 の接続文字列パラメータのリファレンス」『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」 41 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 の接続文字列パラメータのリファレンス」『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」 41 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」 62 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と独立性レベル」 『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」 331 ページ
- 「ULSaveEncryptionKey 関数」 334 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_publication_mask *publication-mask*,
ul_u_long *threshold* **);**

パラメータ

sqlca　SQLCA へのポインタ。

publication-mask　チェック対象のパブリケーションのセット。値が 0 の場合は、データベース全体をチェックする必要があることを示します。それ以外の場合は、OR で結合したパブリケーションのパブリケーション・マスクを指定する必要があります。次に例を示します。

```
UL_PUB_PUB1 | UL_PUB_PUB2
```

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

- 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, 0, 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_PUB_PUB1 | UL_PUB_PUB2, 1000 );
```

次の呼び出しでは、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_SYNC_ALL, 1 );
```

参照

- 「PublicationMask 同期パラメータ」 『Mobile Link - クライアント管理』

ULDropDatabase 関数

Ultra Light データベース・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**　データベースが正常に削除されました。
- **ul_false**　データベース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、指定されたファイル名が不正だった、別のアプリケーションで使用中だったために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 を呼び出した後のどちらか。

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 関数を呼び出した後。

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
};
```

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 データベース設定のリファレンス」『Ultra Light - データベース管理とリファレンス』

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_publication_mask publication-mask,
DECL_DATETIME * value );
```

パラメータ

sqlca　SQLCA へのポインタ。

publication-mask　最終ダウンロード時間を取得するパブリケーションのセット。値 0 は、データベースのすべてのパブリケーションで、ダウンロードのタイムスタンプが必要であることを示します。このセットはマスクとして提供されます。たとえば、次のマスクはパブリケーション PUB1 と PUB2 に対応します。

```
UL_PUB_PUB1 | UL_PUB_PUB2
```

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**　*publication-mask* 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**　*publication-mask* によって複数のパブリケーションが指定されたか、パブリケーションが未定義です。*value* の内容に、意味はありません。

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_PUB_PUB1, &dt );
```

次の呼び出しでは、データベース全体が最後に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。この呼び出しでは、特殊な UL_SYNC_ALL パブリケーション・マスクが使用されます。

```
ret = ULGetLastDownloadTime( &sqlca, UL_SYNC_ALL, &dt );
```

参照

- 「PublicationMask 同期パラメータ」『Mobile Link - クライアント管理』
- 「UL_SYNC_ALL マクロ」 180 ページ
- 「UL_SYNC_ALL_PUBS マクロ」 181 ページ

ULGetSynchResult 関数

最新の同期の結果を格納し、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します。

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 {
   an_sql_code    sql_code;
   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_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、「エラー・メッセージ (SQLSTATE 順)」『SQL Anywhere 10 - エラー・メッセージ』を参照してください。
- **stream_error** ul_stream エラー型の構造体。「Stream Error 同期パラメータ」『Mobile Link - クライアント管理』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」『Mobile Link - クライアント管理』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」『Mobile Link - クライアント管理』を参照してください。
- **timestamp** 最後の同期の時刻と日付。
- **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」『Mobile Link - クライアント管理』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 ー 100 の範囲の short。

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」 335 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」 60 ページ
- 「ULRevokeConnectFrom 関数」 332 ページ

ULHTTPSStream 関数 (旧式)

HTTP を介する同期に適した、Ultra Light HTTPS ストリームを定義します。

構文

ul_stream_defn **ULHTTPSStream(** void **);**

備考

HTTPS ストリームは基本となるトランスポートとして TCP/IP を使用します。Ultra Light アプリケーションは Web ブラウザとして機能し、Mobile Link は Web サーバとして機能します。

旧式な機能
この関数は、SQL Anywhere 10 から使用されなくなりました。フィールド ul_synch_info のストリームには、必要な値を指定した文字列を設定できます。現状では、この関数は適切な文字列値を返します。

参照

- 「ULSynchronize 関数」 340 ページ
- 「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」 『Mobile Link - クライアント管理』

ULHTTPStream 関数 (旧式)

HTTP を介する同期に適した、Ultra Light HTTP ストリームを定義します。

構文

ul_stream_defn **ULHTTPStream(** void **);**

備考

HTTP ストリームは基本となるトランスポートとして TCP/IP を使用します。Ultra Light アプリケーションは Web ブラウザとして機能し、Mobile Link は Web サーバとして機能します。Ultra Light アプリケーションは、サーバへのデータ送信のために POST 要求を送り、サーバからのデータの読み込みのために GET 要求を送ります。

使用されなくなった関数
この関数は、SQL Anywhere 10 から使用されなくなりました。フィールド ul_synch_info のストリームには、必要な値を指定した文字列を設定できます。現状では、この関数は適切な文字列値を返します。

参照

- 「ULSynchronize 関数」 340 ページ
- 「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」 『Mobile Link - クライアント管理』

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」『Mobile Link - クライアント管理』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。

構文

ul_bool **ULIsSynchronizeMessage(** ul_u_long *uMsg* **);**

備考

この関数は、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 CE 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」 106 ページ
- 「Windows CE の ActiveSync」 『Mobile Link - クライアント管理』

ULResetLastDownloadTime 関数

最終ダウンロード時間をリセットするため、アプリケーションは以前にダウンロードされたデータを再同期します。

構文

void **ULResetLastDownloadTime(**
SQLCA * *sqlca*,
ul_publication_mask *publication-mask* **);**

パラメータ

sqlca　SQLCA へのポインタ。

publication-mask　マスクとして提供される、OR で結合したパブリケーションのセット。0 を使用すると、データベース全体のすべてのパブリケーションがリセットされます。たとえば、次のマスクはパブリケーション PUB1 と PUB2 に対応します。

```
UL_PUB_PUB1 | UL_PUB_PUB2
```

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_SYNC_ALL );
```

参照

- 「PublicationMask 同期パラメータ」『Mobile Link - クライアント管理』
- 「ULGetLastDownloadTime 関数」 321 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』

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」 316 ページ
- 「ULSaveEncryptionKey 関数」 334 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」 60 ページ
- 「ULGrantConnectTo 関数」 325 ページ

ULRollbackPartialDownload 関数

失敗した同期から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」 209 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」 『Mobile Link - クライアント管理』
- 「Partial Download Retained 同期パラメータ」 『Mobile Link - クライアント管理』
- 「Resume Partial Download 同期パラメータ」 『Mobile Link - クライアント管理』

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」 316 ページ
- 「ULRetrieveEncryptionKey 関数」 331 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」 324 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,
ul_char * *value* **);**

パラメータ

sqlca　SQLCA へのポインタ。

ul_database_option_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

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 ,
ul_u_long * *value* **);**

パラメータ

sqlca　SQLCA へのポインタ。

ul_database_option_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,
   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。この構造体のメンバの詳細については、「ul_synch_info_a 構造体」 185 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」 310 ページ

ULSocketStream 関数 (旧式)

TCP/IP を介する同期に適した Ultra Light ソケット・ストリームを定義します。

構文

ul_stream_defn **ULSocketStream(** void **);**

備考

この関数は、SQL Anywhere 10 から使用されなくなりました。フィールド ul_synch_info のストリームには、必要な値を指定した文字列を設定できます。現状では、この関数は適切な文字列値を返します。

参照

- 「ULSynchronize 関数」 340 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」 185 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。アプリケーションでは、この関数の SQLCODE 戻り値をテストしてください。

第 14 章

Ultra Light ODBC API リファレンス

目次

Ultra Light ODBC API の概要

この章では、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 SDK マニュアル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。

参照

- Microsoft の『*ODBC Programmer's Reference*』の「SQLAllocHandle」

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**　結果セットを返すステートメントのハンドル。
- **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。

参照

- Microsoft の『*ODBC Programmer's Reference*』の「SQLBindCol」

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。

参照

- Microsoft の『*ODBC Programmer's Reference*』の「SQLBindParameter」

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 の接続文字列パラメータのリファレンス」『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 の接続文字列パラメータのリファレンス」『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

- Microsoft の『*ODBC Programmer's Reference*』の「SQLConnect」

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。

参照

- Microsoft の『*ODBC Programmer's Reference*』の「SQLDescribeCol」

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」 346 ページ
- Microsoft の『*ODBC Programmer's Reference*』の「SQLDisconnect」

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

参照

- Microsoft の『*ODBC Programmer's Reference*』の「SQLEndTran」

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、実行前にステートメントを準備する必要はありません。

SQLExecDirect ではステートメント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

- Microsoft の『*ODBC Programmer's Reference*』の「SQLExecDirect」
- 「SQLExecute 関数」 351 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**　実行するステートメントのハンドル。

備考

SQLPrepare を使用してステートメントを準備してから、実行してください。ステートメントにパラメータ・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、ステートメント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」 345 ページ
- 「SQLPrepare 関数」 360 ページ
- Microsoft の『*ODBC Programmer's Reference*』の「SQLExecute」
- 「SQLExecDirect 関数」 350 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」 353 ページ
- 「SQLBindCol 関数」 344 ページ
- Microsoft の『*ODBC Programmer's Reference*』の「SQLFetch」

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」 352 ページ
- 「SQLBindCol 関数」 344 ページ
- Microsoft の『*ODBC Programmer's Reference*』の「SQLFetchScroll」

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」 343 ページ
- Microsoft の『*ODBC Programmer's Reference*』の「SQLFreeHandle」

SQLGetCursorName 関数

Ultra Light ODBC の場合、指定のステートメント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」 363 ページ
- Microsoft の『*ODBC Programmer's Reference*』の「SQLGetCursorName」

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。

参照

- Microsoft の『*ODBC Programmer's Reference*』の「SQLGetData」

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、「エラー・メッセージ (SQLSTATE 順)」『SQL Anywhere 10 - 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、「エラー・メッセージ (SQL Anywhere の SQLCODE 順)」『SQL Anywhere 10 - 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。

参照

- Microsoft の『*ODBC Programmer's Reference*』の「SQLGetDiagRec」

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。

参照

- Microsoft の『*ODBC Programmer's Reference*』の「SQLGetInfo」

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。

参照

- Microsoft の『*ODBC Programmer's Reference*』の「SQLNumResultCols」

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」 351 ページ
- Microsoft の『*ODBC Programmer's Reference*』の「SQLPrepare」

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。

参照

- Microsoft の『*ODBC Programmer's Reference*』の「SQLRowCount」

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」 81 ページ
- 「SQLSetSuspend 関数」 364 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」 355 ページ
- Microsoft の『*ODBC Programmer's Reference*』の「SQLSetCursorName」

SQLSetSuspend 関数

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」 81 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」『Mobile Link - クライアント管理』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」『Mobile Link - クライアント管理』.
- Mobile Link - サーバ管理 『Mobile Link - サーバ管理』

索引

記号

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

あ

い

え

お

さ

し

す

せ

そ

た

な

に

ね

は

ひ

ふ

へ

ほ

ま

も

ゆ

ら

り

る

れ

ろ

