iAnywhere SOLUTIONS™
A SYBASE COMPANY

Ultra Light C/C++
ユーザーズ・ガイド

パート番号：DC50026-01-0902-01
改訂：2005 年 3 月

版権

Copyright © 2005 iAnywhere Solutions, Inc., Sybase, Inc. All rights reserved.

ここに記載されている内容を iAnywhere Solutions, Inc.、Sybase, Inc. またはその関連会社の書面による事前許可を得ずに電子的、機械的、手作業、光学的、またはその他のいかなる手段によっても複製、転載、翻訳することを禁じます。

Sybase、SYBASE のロゴ、Adaptive Server、AnswerBase、Anywhere、EIP、Embedded SQL、Enterprise Connect、Enterprise Portal、GainMomentum、iAnywhere、jConnect MASS DEPLOYMENT、Netimpact、ObjectConnect、ObjectCycle、OmniConnect、Open ClientConnect、Open ServerConnect、PowerBuilder、PowerDynamo、Powersoft、Quickstart Datamart、Replication Agent、Replication Driver、SQL Anywhere、SQL Central、SQL Remote、Support Plus、SWAT、Sybase IQ、Sybase System 11、Sybase WAREHOUSE、SyBooks、XA-Library は米国法人 Sybase, Inc. の登録商標です。Backup Server、Client-Library、jConnect for JDBC、MainframeConnect、Net-Gateway、Net-Library、Open Client、Open Client/Server、S-Designor、SQL Advantage、SQL Debug、SQL Server、SQL Server Manager、Sybase Central、Watcom、Web.SQL、XP Server は米国法人 Sybase, Inc. の商標です。

Certicom, MobileTrust, および SSL Plus は Certicom Corp. の商標です。Security Builder は Certicom Corp. の登録商標です。

ここに記載されている上記以外の社名および製品名は、各社の商標または登録商標の場合があります。

目次

はじめに xi
SQL Anywhere Studio のマニュアル xii
表記の規則 xvi
CustDB サンプル・データベース xix
詳細情報の検索／フィードバックの提供 xx

1 C/C++ 開発者用 Ultra Light の概要 3
Ultra Light と C/C++ プログラミング言語 4
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 11
Ultra Light C++ のアーキテクチャ 12

2 Ultra Light C++ コンポーネントを使用したアプリケーションの開発 15
Ultra Light ネームスペースの使用 16
Ultra Light データベース・スキーマ 17
データベースへの接続 19
動的 SQL を使用したデータへのアクセス 24
テーブル API を使用したデータへのアクセス 30
トランザクションの管理 38
スキーマ情報へのアクセス 39
エラー処理 41
ユーザの認証 42
データの暗号化 43
データの同期 44
アプリケーションのコンパイルとリンク 45

3 静的型 C++ API を使用したアプリケーションの開発 49
概要 50
アプリケーションの機能の定義 51
データベースへの接続 54
データへのアクセス 55

ユーザの認証 57
データの暗号化 59
データの同期 61
静的型 C++ API アプリケーションの構築 70

4 Embedded SQL を使用したアプリケーションの開発 73
概要 74
SQLCA (SQL Communication Area) の初期化 77
データベースへの接続 79
ホスト変数の使用 81
データのフェッチ 96
ユーザの認証 102
データの暗号化 105
アプリケーションへの同期の追加 107
Embedded SQL アプリケーションの構築 118

5 Ultra Light C/C++ インタフェースの共通機能 129
SQLCA (SQL Communication Area) の概要 130
Ultra Light C/C++ インタフェースの結合 132
Ultra Light データベースのデフラグメンテーション 135

6 Palm Computing Platform 用 Ultra Light アプリケーションの開発 137
概要 138
Metrowerks CodeWarrior で Ultra Light アプリケーションを開発する 139
Ultra Light Palm アプリケーションの状態の保存 147
マルチセグメント・アプリケーションの構築 149
Palm アプリケーションに HotSync 同期を追加する 153
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 155
Palm アプリケーションの配備 156

7 Windows CE 用 Ultra Light アプリケーションの開発 159
概要 160
CustDB サンプル・アプリケーションの構築 163
永続的データの格納 166
Windows CE アプリケーションの配備 167
Windows CE での同期 171

8 チュートリアル：C++ コンポーネントを使用したアプリケーションの構築179
概要 180
レッスン 1 ：データベースへの接続....181
レッスン 2 ：データベースへのデータの挿入 189
レッスン 3 ：テーブルのローの選択....191
レッスン 4 ：アプリケーションへの同期の追加....193
レッスン 5 ：Windows CE デバイスへの配備....196

9 チュートリアル：静的型 C++ API を使用したアプリケーションの構築.... 197
Ultra Light 静的型 C++ API の概要....198
レッスン 1 ：はじめに 201
レッスン 2 ：Ultra Light データベース・テンプレートの作成 202
レッスン 3 ：Ultra Light ジェネレータの実行 204
レッスン 4 ：アプリケーション・ソース・コードの書き込み 205
レッスン 5 ：アプリケーションの構築と実行 208
レッスン 6：アプリケーションへの同期の追加....211
サンプル・データベースのリストア....214

10 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..... 215
概要 216
レッスン 1: eMbedded Visual C++ の設定....217
レッスン 2: Embedded SQL ソース・ファイルに記述....219
レッスン 3: Embedded SQL Ultra Light サンプル・アプリケーションの構築.....227
レッスン 4 ：アプリケーションへの同期の追加....229

11 チュートリアル：ODBC を使用したアプリケーションの構築 231
Ultra Light ODBC の概要....232
レッスン 1 ：はじめに 233
レッスン 2：Ultra Light データベース・スキーマ・ファイルの作成 235
レッスン 3：データベースへの接続....237
レッスン 4：データベースへのデータの挿入 240
レッスン 5：データベースのクエリ....242

12 Ultra Light C/C++ 共通 API リファレンス 247
ULRegisterErrorCallback のコールバック関数....248
ULRegisterSchemaUpgradeObserver のコールバック関数 251
ULEnableFileDB 関数....253

ULEnableGenericSchema 関数 (旧式) 254
ULEnablePalmRecordDB 関数 256
ULEnableStrongEncryption 関数....257
ULEnableUserAuthentication 関数 258
ULRegisterErrorCallback 関数....259
ULRegisterSchemaUpgradeObserver 関数 263
ULStoreDefragFini 関数....265
ULStoreDefragInit 関数....266
ULStoreDefragStep 関数 267
Ultra Light C/C++ アプリケーションのマクロとコンパイラ指令 268

13 Ultra Light C++ コンポーネント API リファレンス 275
クラス ULSqlca....276
クラス ULSqlcaBase....278
クラス ULSqlcaWrap....282
クラス UltraLite_Connection....284
クラス UltraLite_Connection_iface 286
クラス UltraLite_Cursor_iface....299
クラス UltraLite_DatabaseManager 304
クラス UltraLite_DatabaseManager_iface....305
クラス UltraLite_DatabaseSchema....308
クラス UltraLite_DatabaseSchema_iface 309
クラス UltraLite_IndexSchema....313
クラス UltraLite_IndexSchema_iface 314
クラス UltraLite_PreparedStatement 318
クラス UltraLite_PreparedStatement_iface 319
クラス UltraLite_ResultSet....322
クラス UltraLite_ResultSet_iface....323
クラス UltraLite_ResultSetSchema 324
クラス UltraLite_RowSchema_iface 325
クラス UltraLite_SQLObject_iface 329
クラス UltraLite_StreamReader....331
クラス UltraLite_StreamReader_iface 332
クラス UltraLite_StreamWriter....335
クラス UltraLite_Table 336
クラス UltraLite_Table_iface 338
クラス UltraLite_TableSchema....347
クラス UltraLite_TableSchema_iface 349

クラス ULValue 356

14 Ultra Light 静的型 C++ API リファレンス 373
C++ API クラス階層 374
C++ API 言語要素 376
ULConnection クラス 378
ULCursor クラス 395
ULData クラス 411
ULResultSet クラス 422
ULTable クラス 424
生成された結果セットのクラス 432
生成文のクラス 435
生成されたテーブル・クラス 438

15 Embedded SQL API リファレンス 445
db_fini 関数 446
db_init 関数 447
db_start_database 関数 448
db_stop_database 関数 449
ULActiveSyncStream 関数 450
ULChangeEncryptionKey 関数 451
ULClearEncryptionKey 関数 452
ULCountUploadRows 関数 453
ULDropDatabase 関数 455
ULGetDatabaseID 関数 457
ULGetLastDownloadTime 関数 458
ULGetSynchResult 関数 460
ULGlobalAutoincUsage 関数 462
ULGrantConnectTo 関数 463
ULHTTPSStream 関数 464
ULHTTPStream 関数 465
ULIsSynchronizeMessage 関数 466
ULPalmDBStream 関数 (旧式) 467
ULPalmExit 関数 (旧式) 468
ULPalmLaunch 関数 (旧式) 469
ULResetLastDownloadTime 関数 470
ULRetrieveEncryptionKey 関数 471
ULRevokeConnectFrom 関数 472

ULRollbackPartialDownload 関数 473
ULSaveEncryptionKey 関数 474
ULSetDatabaseID 関数 475
ULSetSynchInfo 関数 476
ULSocketStream 関数 477
ULSynchronize 関数 478

16 Ultra Light ODBC API リファレンス 479
SQLAllocHandle 関数 480
SQLBindCol 関数 481
SQLBindParameter 関数 482
SQLConnect 関数 484
SQLDescribeCol 関数 486
SQLDisconnect 関数 488
SQLEndTran 関数 489
SQLExecDirect 関数 490
SQLExecute 関数 491
SQLFetch 関数 492
SQLFetchScroll 関数 493
SQLFreeHandle 関数 494
SQLGetCursorName 関数 495
SQLGetData 関数 496
SQLGetDiagRec 関数 497
SQLGetInfo 関数 499
SQLNumResultCols 関数 500
SQLPrepare 関数 501
SQLRowCount 関数 502
SQLSetCursorName 関数 503
SQLSetConnectionName 関数 504
SQLSetSuspend 関数 505
SQLSynchronize 関数 506

17 同期パラメータ・リファレンス 507
同期パラメータ 508
auth_parms パラメータ 510
auth_status パラメータ 511
auth_value 同期パラメータ 512
checkpoint_store 同期パラメータ 513

disable_concurrency 同期パラメータ 514
download_only 同期パラメータ 515
keep_partial_download 同期パラメータ 516
ignored_rows 同期パラメータ 517
new_password 同期パラメータ 518
num_auth_parms パラメータ 519
observer 同期パラメータ 520
partial_download_retained 同期パラメータ 521
password 同期パラメータ 522
ping 同期パラメータ 523
publication 同期パラメータ 524
resume_partial_download 同期パラメータ 525
security 同期パラメータ 526
security_parms 同期パラメータ 527
send_column_names 同期パラメータ 529
send_download_ack 同期パラメータ 530
stream 同期パラメータ 531
stream_error 同期パラメータ 534
stream_parms 同期パラメータ 538
upload_ok 同期パラメータ 539
upload_only 同期パラメータ 540
user_data 同期パラメータ 541
user_name 同期パラメータ 542
version 同期パラメータ 543

索引 545

はじめに

このマニュアルの内容

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

対象読者

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

SQL Anywhere Studio 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 Studio のマニュアル

SQL Anywhere Studio のマニュアルは、各マニュアルを 1 つの大きなヘルプ・ファイルにまとめたオンライン形式、マニュアル別の PDF ファイル、および有料の製本版マニュアルで提供されます。SQL Anywhere Studio のマニュアルは、次の分冊マニュアルで構成されています。

- **『SQL Anywhere Studio の紹介』** このマニュアルでは、SQL Anywhere Studio のデータベース管理と同期テクノロジの概要について説明します。また、SQL Anywhere Studio を構成する各部分について説明するチュートリアルも含まれています。
- **『SQL Anywhere Studio 新機能ガイド』** このマニュアルは、SQL Anywhere Studio のこれまでのリリースのユーザを対象としています。ここでは、製品の今回のリリースと以前のリリースで導入された新機能をリストし、アップグレード手順を説明しています。
- **『Adaptive Server Anywhere データベース管理ガイド』** このマニュアルでは、データベースおよびデータベース・サーバの実行、管理、設定について説明しています。
- **『Adaptive Server Anywhere SQL ユーザーズ・ガイド』** 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。
- **『Adaptive Server Anywhere SQL リファレンス・マニュアル』** このマニュアルは、Adaptive Server Anywhere で使用する SQL 言語の完全なリファレンスです。また、Adaptive Server Anywhere のシステム・テーブルとシステム・プロシージャについても説明しています。
- **『Adaptive Server Anywhere プログラミング・ガイド』** このマニュアルでは、C、C++、Java プログラミング言語を使用してデータベース・アプリケーションを構築、配備する方法につい

て説明します。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。また、Adaptive Server Anywhere ADO.NET データ・プロバイダについても説明します。

- **『Adaptive Server Anywhere SNMP Extension Agent ユーザーズ・ガイド』** このマニュアルでは、Adaptive Server Anywhere SNMP Extension Agent を SNMP 管理アプリケーションとともに使用できるように設定して、Adaptive Server Anywhere データベースを管理できるようにする方法を説明します。

- **『Adaptive Server Anywhere エラー・メッセージ』** このマニュアルでは、Adaptive Server 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。

- **『SQL Anywhere Studio セキュリティ・ガイド』** このマニュアルでは、Adaptive Server Anywhere データベースのセキュリティ機能について説明します。Adaptive Server Anywhere 7.0 は、米国政府から TCSEC (Trusted Computer System Evaluation Criteria) の C2 セキュリティ評価を授与されています。このマニュアルには、Adaptive Server Anywhere の現在のバージョンを、C2 基準を満たした環境と同等の方法で実行することを望んでいるユーザにとって役に立つ情報が含まれています。

- **『Mobile Link 管理ガイド』** このマニュアルでは、モバイル・コンピューティング用の Mobile Link データ同期システムについてあらゆる角度から説明します。このシステムによって、Oracle、Sybase、Microsoft、IBM の単一データベースと、Adaptive Server Anywhere や Ultra Light の複数データベースの間でのデータ共有が可能になります。

- **『Mobile Link クライアント』** このマニュアルでは、Adaptive Server Anywhere リモート・データベースと Ultra Light リモート・データベースの設定を行い、これらを同期させる方法について説明します。

- **『Mobile Link サーバ起動同期ユーザーズ・ガイド』** 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の開始を可能にする Mobile Link の機能です。

- **『Mobile Link チュートリアル』** このマニュアルには、Mobile Link アプリケーションの設定と実行を行う方法を説明するチュートリアルがいくつか用意されています。

- **『QAnywhere ユーザーズ・ガイド』** このマニュアルでは、Mobile Link QAnywhere について説明します。Mobile Link QAnywhere は、従来のデスクトップ・クライアントやラップトップ・クライアントだけでなく、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・アプリケーションの開発と展開を可能にするメッセージング・プラットフォームです。

- **『Mobile Link およびリモート・データ・アクセスの ODBC ドライバ』** このマニュアルでは、Mobile Link 同期サーバから、または Adaptive Server Anywhere リモート・データ・アクセスによって、Adaptive Server Anywhere 以外の統合データベースにアクセスするための ODBC ドライバの設定方法について説明します。

- **『SQL Remote ユーザーズ・ガイド』** 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、あらゆる角度から説明します。このシステムによって、Adaptive Server Anywhere または Adaptive Server Enterprise の単一データベースと Adaptive Server Anywhere の複数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 Studio ヘルプ』** このマニュアルには、Sybase Central や Interactive SQL、その他のグラフィカル・ツールに関するコンテキスト別のヘルプが含まれています。これは、製本版マニュアル・セットには含まれていません。

- **『Ultra Light データベース・ユーザーズ・ガイド』** このマニュアルは、Ultra Light 開発者を対象としています。ここでは、Ultra Light データベース・システムの概要について説明します。また、すべての Ultra Light プログラミング・インタフェースに共通する情報を提供します。

- **Ultra Light のインタフェースに関するマニュアル** 各 Ultra Light プログラミング・インタフェースには、それぞれに対応するマニュアルを用意しています。これらのインタフェースは、RAD

(ラピッド・アプリケーション開発) 用の Ultra Light コンポーネントとして提供されているものと、C、C++、Java 開発用の静的インタフェースとして提供されているものがあります。

このマニュアル・セットの他に、PowerDesigner と InfoMaker には、独自のオンライン・マニュアル (英語版) がそれぞれ用意されています。

マニュアルの形式

SQL Anywhere Studio 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 Studio 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 9] － [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・ディレクトリに保存されている HTML マニュアルを参照してください。

- **PDF 版マニュアル** SQL Anywhere の各マニュアルは、Adobe Acrobat Reader で表示できる PDF ファイルで提供されています。

  PDF 版マニュアルは、オンライン・マニュアルまたは Windows の [スタート] メニューから利用できます。

- **製本版マニュアル** 製本版マニュアルをご希望の方は、ご購入いただいた販売代理店または弊社営業担当までご連絡ください。

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 つだけ選択するか、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、どちら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** }]

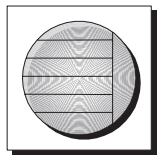
  QUOTES オプションを使用する場合は、ON または OFF のどちらかを選択する必要があります。角カッコと中カッコは入力しないでください。

グラフィック・アイコン

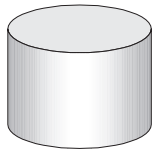
このマニュアルでは、次のアイコンを使用します。

- クライアント・アプリケーション

- Sybase Adaptive Server Anywhere などのデータベース・サーバ



- 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 同期サーバ、SQL Remote Message Agent などがあげられます。

- プログラミング・インタフェース

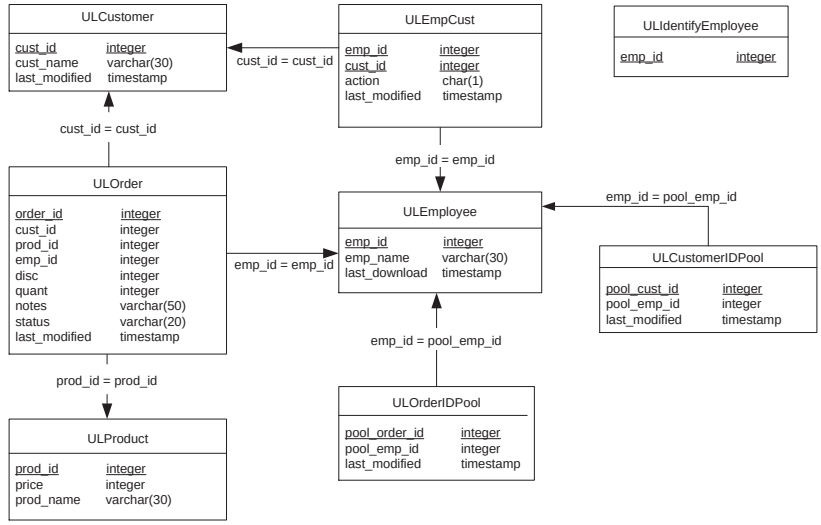
CustDB サンプル・データベース

Mobile Link と Ultra Light のマニュアルでは、多くの例で Ultra Light のサンプル・データベースが使用されています。

Ultra Light サンプル・データベースのリファレンス・データベースは、*custdb.db* という名前のファイルに保存され、SQL Anywhere ディレクトリのサブディレクトリ *Samples¥UltraLite¥CustDB* に置かれています。全面的にこのデータベースを使用して構築したアプリケーションも提供されています。

サンプル・データベースは、あるハードウェア販売会社の販売管理データベースです。データベースには、この販売会社の顧客、製品、営業戦力に関する情報が入っています。

次の図は、CustDB データベース内のテーブルと、各テーブル間の関係を示しています。



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 iAnywhere Solutions 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 Studio 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 **dbeng9 -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 Solutions 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。

第 1 部　概要

第 1 部では、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。

第 1 章

C/C++ 開発者用 Ultra Light の概要

この章の内容

この章では、Ultra Light for C/C++ について説明します。ここでは、『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light へようこそ」で説明した Ultra Light の機能について理解していることを前提としています。

Ultra Light for C/C++ を使用したアプリケーションの作成の詳細については、「Ultra Light C/C++ インタフェースの共通機能」129 ページを参照してください。

Ultra Light for C/C++ の概要のチュートリアルについては、「チュートリアル：C++ コンポーネントを使用したアプリケーションの構築」179 ページを参照してください。

Ultra Light と C/C++ プログラミング言語

C と C++ 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。

- 堅牢なリレーショナル・データベース・ストア
- C または C++ の優れた機能、効率、柔軟性
- Windows CE、Palm OS、Windows XP プラットフォームへの配備

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・ユーザーズ・ガイド』> 「Ultra Light データベース」を参照してください。

Ultra Light プログラミング・インタフェースは動的または静的です。静的インタフェースと動的インタフェースには異なる利点と制限事項があります。また、これらは開発モデルも異なります。

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

- Ultra Light C++ コンポーネント
- Embedded SQL (静的インタフェース)
- 静的型 C++ API (静的インタフェース)
- 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 を使用する必要があります。

詳細については、『Ultra Light データベース・ユーザーズ・ガイド』> 「コンポーネントと静的インタフェースの選択」を参照してください。

C++ コンポーネントの利点と制限事項

Ultra Light for C++ コンポーネントの利点と制約は次のとおりです。

- **動的 SQL**　Ultra Light C++ コンポーネントは、動的 SQL へのアクセスを提供し、開発可能なアプリケーションの範囲を広げます。動的 SQL を使用すると、SQL 文を実行時に定義できます。一方、静的インタフェース (Embedded SQL や静的型 C++ API など) を使用する場合、すべての SQL 文をコンパイル時に指定する必要があります。

- **開発モデル**　Ultra Light C++ コンポーネントを使用する場合、Ultra Light スキーマ・ペインタを使用して Ultra Light データベースを作成できます。また、前処理は必要ありません。一方、Embedded SQL および静的型 C++ API インタフェースでは、アプリケーションのコンパイル前に前処理の作業が必要です。さらに、Adaptive Server Anywhere リファレンス・データベースを構築することも必要になります。

- **効率性**　テーブル数の少ないデータベースでは、静的型インタフェースで開発されたアプリケーションは、コンポーネント型のインタフェースで開発されたアプリケーションよりも占有容量が少なくて済みます。しかし、より複雑なスキーマのデータベースでは、静的型インタフェースのこの利点はなくなります。

- **マルチプロセッサのサポート**　Ultra Light C++ コンポーネント API によって、複数のアプリケーションの接続をサポートする Ultra Light ランタイム・エンジンの使用が可能です。

熟練の開発者であれば、Ultra Light C++ コンポーネント API の機能と静的型 C++ API や Embedded SQL の機能とを組み合わせることができます。

詳細については、「Ultra Light C/C++ インタフェースの結合」132 ページを参照してください。

静的型 C++ API の利点と制限事項

Ultra Light には、静的開発モデル (静的型 C++ インタフェースはこのうちの 1 つ) と Ultra Light コンポーネントの 2 つ以外にも、いくつかのプログラミング・インタフェースが用意されています。静的型 C++ API の利点と制約の多くは、Embedded SQL のものと同じです。

静的型 C++ API には、次のような利点があります。

- **専有容量の小さいデータベース** 静的型 C++ API は各アプリケーションに合わせてコンパイルされた Ultra Light データベース・エンジンを使用するため、データベースの占有容量は、特にテーブルの数が少ない場合、通常の Ultra Light コンポーネントを使用する場合に比べて小さくなります。テーブルの数が多くなると、この利点もなくなります。

- **ハイパフォーマンス** C および C++ アプリケーションの優れたパフォーマンスと生成コードの最適化 (データ・アクセス計画など) とを統合することで、静的型 C++ API によるアプリケーション開発を効率化します。

- **さまざまな SQL をサポート** 静的型 C++ API では、さまざまな種類の SQL を開発アプリケーションに使用できます。

静的型 C++ API には、次のような制約があります。

- **C または C++ の知識が必要** C や C++ のプログラミング知識がない場合は、Ultra Light が提供する別のインタフェースを使用できます。Ultra Light コンポーネントには、複数の一般的なプログラミング言語とツールのインタフェースが用意されています。

- **複雑な開発モデル** Ultra Light データベース・スキーマを保持するリファレンス・データベースを使用し、さらに開発アプリケーションに合わせた API を生成する必要がある場合、静的型 C++ API による開発プロセスは複雑なものになります。Ultra Light コンポーネントを使用すると、開発プロセスは極めて単純なものなります。

- **設計時に SQL を指定する必要がある** コンパイル時に定義された SQL 文しかアプリケーションに組み込むことができません。Ultra Light コンポーネントでは、SQL 文をいつでも動的に組み込むことができます。

開発モデルは、個々のプロジェクトの要件やプログラマのスキルと経験を考慮して選定してください。

静的型 C++ アプリケーションの開発

静的型 C++ Ultra Light アプリケーションを開発するときは、リファレンス・データベースから生成されたプログラミング・インタフェースを使用します。これらの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。

静的型 C++ Ultra Light アプリケーションの開発プロセスは、次のとおりです。

1. データベースを設計します。

   Ultra Light データベースにインクルードするテーブルとインデックスを含む Adaptive Server Anywhere リファレンス・データベースを準備します。

2. SQL 文をデータベースに追加します。

   アプリケーションに使用する SQL 文をリファレンス・データベースに追加する必要があります。

3. アプリケーションの API を生成します。

   Ultra Light ジェネレータを使って、開発するアプリケーションの API を生成します。

4. アプリケーションを作成します。

   アプリケーション・コードのデータ・アクセス機能は、生成した API からの関数呼び出しを使用します。

5. .cpp ファイルをコンパイルします。

   生成した .cpp ファイルは、他の .cpp ファイルと同じようにコンパイルできます。

6. .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。

開発プロセスの詳細な説明については、『Ultra Light C/C++ ユーザーズ・ガイド』> 「静的型 C++ API アプリケーションの構築」を参照してください。

Embedded SQL の利点と制約

Ultra Light には、静的型開発モデル (Embedded SQL はこのうちの 1 つ) と Ultra Light コンポーネントの 2 つ以外にも、いくつかのプログラミング・インタフェースが用意されています。Embedded SQL の利点と制約の多くは、Ultra Light 静的型 C++ API のものと同じです。

Embedded SQL には、次のような利点があります。

- **専有容量の小さいデータベース** Embedded SQL は各アプリケーションに合わせてコンパイルされた Ultra Light データベース・エンジンを使用するため、データベースの占有容量は、特にテーブルの数が少ない場合、通常 Ultra Light コンポーネントを使用する場合に比べて小さくなります。テーブルの数が多くなると、この利点もなくなります。

- **ハイパフォーマンス** C および C++ アプリケーションの優れたパフォーマンスと生成コードの最適化 (データ・アクセス計画など) とを統合することで、Embedded SQL によるアプリケーション開発を効率化します。

- **さまざまな SQL をサポート** Embedded SQL では、さまざまな種類の SQL を開発アプリケーションに使用できます。

Embedded SQL には、次のような制約があります。

- **C または C++ の知識が必要** C や C++ のプログラミング知識がない場合は、Ultra Light が提供する別のインタフェースを使用できます。Ultra Light コンポーネントには、複数の一般的なプログラミング言語とツールのインタフェースが用意されています。

- **複雑な開発モデル** Ultra Light データベース・スキーマを保持するリファレンス・データベースを使用し、さらにソース・コード・ファイルを前処理する必要がある場合、

Embedded SQL による開発プロセスは複雑なものになります。Ultra Light コンポーネントを使用すると、開発プロセスは極めて単純なものなります。

- **設計時に SQL を指定する必要がある** コンパイル時に定義された SQL 文しかアプリケーションに組み込むことができません。Ultra Light コンポーネントでは、SQL 文をいつでも動的に組み込むことができます。

開発モデルは、個々のプロジェクトの要件やプログラマのスキルと経験を考慮して選定してください。

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. データベースを設計します。

   Ultra Light データベースにインクルードするテーブルとインデックスを含む Adaptive Server Anywhere リファレンス・データベースを準備します。

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」のプレフィクスが付いた、実行する SQL 文を使用します。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行います。

   SQL Anywhere Studio に装備された SQL プリプロセッサ (sqlpp) を使用して、.sqc ファイルを読み込み、Adaptive Server Anywhere リファレンス・データベースにアクセスし、.c または .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. 生成した .c または .cpp ファイルをコンパイルします。

   生成した .c または .cpp ファイルは、他の .c または .cpp ファイルと同じようにコンパイルできます。

5. コンパイルした .c または .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 の開発プロセスの詳細については、「Embedded SQL アプリケーションの構築」118 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 NT/2000/XP
- 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 以降または Palm OS 3.5 以降。

詳細については、『SQL Anywhere Studio の紹介』> 「Ultra Light 開発プラットフォーム」と『SQL Anywhere Studio の紹介』> 「Ultra Light ターゲット・プラットフォーム」を参照してください。

Ultra Light C++ のアーキテクチャ

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**　各 Connection オブジェクトは、Ultra Light データベースとの接続を表します。Connection オブジェクトは 1 つまたは複数作成できます。
- **Table**　Table オブジェクトは、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**　これらの動的 SQL オブジェクトを使って、動的 SQL 文の作成、クエリの記述、INSERT、UPDATE、および DELETE 文の実行、およびプログラムによるデータベースの結果セットの制御ができます。
- **SyncParms**　SyncParms オブジェクトを使用して、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』はオンライン・マニュアルで提供されています。API リファレンスへのアクセスの詳細については、「Ultra Light C++ コンポーネント API リファレンス」275 ページを参照してください。

第 2 部　アプリケーション開発

第 2 部では、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++ コンポーネントを使用したアプリケーションの開発

この章の内容

この章では、Ultra Light C++ コンポーネントを使用してアプリケーションを開発する方法について説明します。

チュートリアルについては、「チュートリアル：C++ コンポーネントを使用したアプリケーションの構築」179 ページを参照してください。

Ultra Light ネームスペースの使用

Ultra Light C++ コンポーネント・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。

Ultra Light データベース・スキーマ

データベース・スキーマとは、データベースを記述するものです。データベース・スキーマは、データベース内のテーブル、インデックス、キー、パブリケーションの集合であり、それらの間のすべての関係を示します。

Ultra Light データベースのスキーマを直接変更しないでください。代わりに、スキーマ (*.usm*) ファイルを作成し、アプリケーションに組み込まれた Ultra Light 関数を呼び出すことによって、そのファイルからデータベース・スキーマをアップグレードします。

スキーマ・ファイルは、データベース作成時の初期にデータベースの構造を指定するためにも使用されます。

Ultra Light データベース・スキーマ・ファイルの作成

Ultra Light スキーマ・ペインタまたは *ulinit* ユーティリティを使用して、Ultra Light スキーマ・ファイルを作成できます。

- **Ultra Light スキーマ・ペインタ**　Ultra Light スキーマ・ペインタは、Ultra Light スキーマ・ファイルの作成と編集に使用するグラフィカル・ユーティリティです。

  スキーマ・ペインタを起動するには、[スタート] － [プログラム] － [SQL Anywhere 9] － [Ultra Light] － [Ultra Light Schema Painter] を選択するか、Windows エクスプローラでスキーマ・ファイル (*.usm*) をダブルクリックします。

  Ultra Light スキーマ・ペインタの使用方法の詳細については、『Ultra Light データベース・ユーザーズ・ガイド』> 「レッスン 1：Ultra Light データベース・スキーマの作成」を参照してください。

- **ulinit ユーティリティ**　Adaptive Server Anywhere データベース管理システムがある場合は、*ulinit* コマンド・ライン・ユーティリティを使用して Ultra Light スキーマ・ファイルを生成できます。

ulinit ユーティリティの使用の詳細については、『Ultra Light データベース・ユーザーズ・ガイド』> 「ulinit ユーティリティ」を参照してください。

データベース・スキーマのアップグレード

既存のデータベース構造を変更するには、UpgradeSchemaFromFile メソッドを使用します。ほとんどの場合、データの消失は起こりません。ただし、カラムを削除したり、カラムのデータ型が互換性のない型に変更された場合は、データの消失が起こる場合があります。

例

次のコードで、新規スキーマ・ファイルが適用されます。

```
// issue a commit before applying a new schema
 // if there are any uncommitted transactions
 conn->Commit();
 conn->UpgradeSchemaFromFile(
   UL_TEXT("schema_file=genup01b.usm") ) );
```

詳細については、「UpgradeSchemaFromFile 関数」298 ページを参照してください。

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。サンプルのコードは、*Samples¥UltraLite¥CustDB* に格納されています。

Connection オブジェクトの使用

次に示す Connection オブジェクトのプロパティは、アプリケーションのグローバルな動作を管理します。

- **コミット動作**　Ultra Light C++ コンポーネントには、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 Connection.Commit 文を指定します。

  詳細については、「トランザクションの管理」38 ページを参照してください。

- **ユーザ認証**　Grant と Revoke の接続許可メソッドを使用すると、アプリケーションのユーザ ID とパスワードをデフォルト値の DBA と SQL から別の値に変更できます。各アプリケーションは、最大 4 つのユーザ ID を保持できます。

  詳細については、『Ultra Light データベース・ユーザーズ・ガイド』>「Ultra Light のユーザ認証」および「ユーザの認証」42 ページを参照してください。

- **同期**　Connection オブジェクトから、同期を管理するオブジェクトのセットにアクセスできます。

  詳細については、「データの同期」44 ページを参照してください。

- **テーブル**　Ultra Light 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。

  詳細については、「テーブル API を使用したデータへのアクセス」30 ページを参照してください。

- **準備文**　動的 SQL 文の実行を処理し、結果セットをナビゲーションするために、オブジェクトのセットが提供されています。

詳細については、「動的 SQL を使用したデータへのアクセス」24 ページを参照してください。

Connection オブジェクトの詳細については、「クラス UltraLite_Connection」284 ページ を参照してください。

❖ 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++ コンポーネント・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);
```

詳細については、「クラス UltraLite_DatabaseManager_iface」305 ページ を参照してください。

Palm OS の場合は、ULEnableFileDB や ULEnablePalmRecordDB を呼び出す前に、ULInitDatabaseManager を呼び出してください。Palm データ・ストアや仮想ファイル・システムを使用するためのデータベース・ストアが設定されます。詳細については、『Ultra

Light C/C++ ユーザーズ・ガイド』> 「ULEnablePalmRecordDB 関数」および『Ultra Light C/C++ ユーザーズ・ガイド』> 「ULEnableFileDB 関数」を参照してください。

3 Connection オブジェクトを宣言します。

ほとんどのアプリケーションは、Ultra Light データベースへの単一接続を使用し、接続を開いたままにしておきます。複数の接続が必要なのは、マルチスレッド・データ・アクセスの場合だけです。そのため、Connection 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。

```
Connection *  conn = UL_NULL;
```

詳細については、「クラス UltraLite_Connection_iface」286 ページ を参照してください。

4 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。

ほとんどの Ultra Light アプリケーションは、データベース・ファイルではなくスキーマ・ファイルを配備し、最初の接続時に Ultra Light にデータベース・ファイルを作成させます。次のコードは、既存のデータベースへの接続を試みます。データベース・ファイルが存在しない場合は、アプリケーションがデータベース・ファイルを作成します。

```
// specify the location of the database file
 static ul_char * parms = "file_name=mydata.udb";
 ULValue lp( parms );
 conn = dbMgr->OpenConnection( sqlca, lp );
 if( sqlca.GetSQLCode() ==
   SQLE_ULTRALITE_DATABASE_NOT_FOUND ) {
   static ul_char * parms =
UL_TEXT("schema_file=mydata.usm")
   UL_TEXT(";file_name=mydata.udb");
   ULValue lp( parms );
   conn = dm->CreateAndOpenDatabase( sqlca, lp );
```

```
        if( sqlca.GetSQLCode() < SQLE_NOERROR ){
          printf( "Open failed with sql code: %d.¥n" ,
          sqlca.GetSQLCode() );
        }
      }
```

詳細については、「OpenConnection 関数」306 ページおよび「CreateAndOpenDatabase 関数」305 ページを参照してください。

例

次のコードは、Ultra Light データベース *mydata.udb* への接続を開きます。

```
#include "uliface.h"
 using namespace UltraLite;
 static ul_char *  parms =
 UL_TEXT( ";file_name=tutcustomer.udb"  )
 UL_TEXT( ";schema_file=tutcustomer.usm" );
 ULSqlca sqlca;
 DatabaseManager * dm   = UL_NULL;
 Connection *   conn   = UL_NULL;
 sqlca.Initialize();
 dm = ULInitDatabaseManager( sqlca );
 if( dm == UL_NULL ){
    // You may have mismatched UNICODE vs. ANSI
runtimes.
    return 1;
 }
 ULValue  lp( parms );
 conn = dm->OpenConnection( sqlca, lp );
 if( sqlca.GetSQLCode() ==
     SQLE_ULTRALITE_DATABASE_NOT_FOUND ) {
    conn = dm->CreateAndOpenDatabase( sqlca, lp );
    if( sqlca.GetSQLCode() < SQLE_NOERROR ) {
       printf( "Open failed with sql code: %d.¥n" ,
       sqlca.GetSQLCode() );
       return NULL;
    }  else {
       printf( "Connected to a new database.¥n" );
    }
 } else {
    printf( "Connected to an existing database.¥n" );
 }
 return( conn );
```

マルチスレッド・アプリケーション

Connection オブジェクトと、このオブジェクトから作成されるすべてのオブジェクトは、それぞれ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。

動的 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、動的 SQL またはテーブル 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、動的 SQL 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」30 ページを参照してください。

この項では、動的 SQL を使用して次の操作を行う方法について説明します。

- 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 言語そのものについては説明しません。動的 SQL 機能の詳細については、『Ultra Light データベース・ユーザーズ・ガイド』> 「動的 SQL」を参照してください。

SQL 操作に必要な一連の操作の概要については、『Ultra Light データベース・ユーザーズ・ガイド』> 「動的 SQL の使用」を参照してください。

データ操作：INSERT、UPDATE、DELETE

Ultra Light では、SQL データ操作言語の操作を実行できます。これらの操作は、PreparedStatement クラスのメンバである ExecuteStatement メソッドを使用して実行します。

詳細については、「クラス UltraLite_PreparedStatement」318 ページ を参照してください。

準備文でのパラメータの使用
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、UPDATE、DELETE では必ず、各 ? は準備文での並び順を参照します。たとえば、最初の ? は 1、2 番目の ? は 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;
```

詳細については、「PrepareStatement 関数」293 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。

```
ULValue sqltext(
     "INSERT INTO MyTable(MyColumn) values (?)" );
 prepStmt = conn->PrepareStatement( sqltext );
```

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。

```
ULValue newValue( "string-value" );
 prepStmt->SetParameter( 1, newValue );
```

4 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 変更内容をコミットします。

```
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 文を PreparedStatement オブジェクトに割り当てます。

```
ULValue sqltext(
    "UPDATE MyTable SET MyColumn1 = ? WHERE
MyColumn2 = ?" );
 prepStmt = conn->PrepareStatement( sqltext );
```

3 文の入力パラメータ値を割り当てます。

```
ULValue newValue( new-value );
 ULValue oldValue( old-value );
 stmt->SetParameter( 1, newValue );
 stmt->SetParameter( 2, oldValue );
```

4 文を実行します。

```
long rowsUpdated = 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 PreparedStatement オブジェクトに文を割り当てます。

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

3 文の入力パラメータ値を割り当てます。

```
ULValue deleteValue( old-value );
 prepStmt->SetParameter( 1, deleteValue );
```

4　文を実行します。

```
long rowsDeleted = 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。

詳細については、「クラス UltraLite_PreparedStatement_iface」319 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　新しい準備文と結果セットを作成します。

```
PreparedStatement * prepStmt;
```

2　準備文を新しく作成した PreparedStatement オブジェクトに割り当てます。

```
ULValue sqltext( "SELECT MyColumn FROM MyTable" );
 prepStmt = conn->PrepareStatement( sqltext );
```

3　文を実行します。

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

```
#define MAX_NAME_LEN     100
 ULValue mycol;
 ResultSet * rs = stmt->ExecuteQuery();
 rs->BeforeFirst();
 while( rs->Next() ){
```

```
    char mycol[ MAX_NAME_LEN ];
    val = rs->Get( 1 );
    val.GetString( mycol, MAX_NAME_LEN );
    printf( "mycol= %s¥n", mycol );
}
```

動的 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()** 最後のローの後に移動します。
- **BeforeFirst()** 最初のローの前に移動します。
- **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、オフセットが指定し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、カーソルは移動しませんが、ロー・バッファが再配置されます。

詳細については、「クラス UltraLite_ResultSet_iface」323 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 );
 }
```

詳細については、「GetSchema 関数」323 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、動的 SQL またはテーブル 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。

動的 SQL については、「動的 SQL を使用したデータへのアクセス」24 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++ コンポーネントは、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()** 最後のローの後に移動します。
- **BeforeFirst()** 最初のローの前に移動します。
- **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、オフセットが指定し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、現在のテーブル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、カーソルは移動しませんが、ロー・バッファが再配置されます。

詳細については、「クラス UltraLite_Table_iface」338 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 * t = conn->openTable( "MyTable" );
 ul_column_num colValue =
    t->GetSchema()->GetColumnID( "MyColumn" );
 while ( t->Next() ){
    char lname[ MAX_NAME_LEN ];
    printf( "%s¥n", colValue );
 }
```

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( "MyTable" )
 ULValue index_name( "ix_col" )
 Table * t =
    conn->OpenTableWithIndex( table_name, index_name );
 t.moveFirst();
```

詳細については、「クラス UltraLite_Table_iface」338 ページ を参照してください。

Ultra Light のモードの使用

Ultra Light モードは、バッファ内の値の使用目的を指定します。Ultra Light には、デフォルト・モードに加えて、次の 4 つの操作モードがあります。

- **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** いずれかの find メソッドが呼び出されたときに、バッファ内のデータに正確に一致する値のローを検索します。
- **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を検索します。

現在のローの値へのアクセス

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、各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。

例

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 = t->Get( lname_col_id );
 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 = (int)(t->Get( id_col_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。

例

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

これらのプロパティへの値の割り当てによって、データベース内のデータの値が変更されることはありません。現在の位置がテーブルの最初のローの前でも、テーブルの最後のローの後ろであっても、プロパティに値を割り当てることができます。ただし、これらの位置に現在のローが置かれている場合に、たとえば、変数へのプロパティの割り当てなどによって、データにアクセスしようとするとエラーになります。

```
// This code is incorrect
 t.BeforeFirst();
 id = t.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。

find と lookup を使用したローの検索

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。これらのモードのうちの 2 つ (検索モードとルックアップ・モード) は、検索に使用されます。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。

モードを開始するには、テーブル・オブジェクトのメソッドを呼び出します。たとえば、次のコードは検索モードを開始します。

```
t.FindBegin();
```

2 検索値を設定します。

検索値は、現在のローの値を設定することで設定します。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(
"lname" );
 ULValie v_lname( "Kaminski" );
 t.Set( lname_col, v_lname );
```

3　ローを検索します。

適切なメソッドを使用して検索を実行します。たとえば、次の指示は、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、ほか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。

適切なメソッドを使用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。

詳細については、「クラス UltraLite_Table_iface」338 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、t 上で更新モードを開始します。

```
t.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。

```
t.SetInt( id, 3 );
```

4 Update を実行します。

```
t.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。Table オブジェクトを開いたときに指定したインデックスのカラム値を変更した場合は、現在のローは不確定です。

Ultra Light C++ コンポーネントは、`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。詳細については、「トランザクションの管理」38 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。テーブルへのローの挿入順序に意味はありません。

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。

```
t.InsertBegin();
 t.SetInt( id, 3 );
 t.SetString( lname, "Carlo" );
 t.Insert();
 t.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、次のいずれかのエントリが追加されます。

- 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。

更新操作の場合は、コミットを実行したときに、永続的な記憶領域のデータベースに挿入が適用されます。AutoCommit モードでは、insert メソッドの一部としてコミットが実行されます。

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。挿入モードや更新モードに対応する削除モードはありません。

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。

```
t.Delete();
```

トランザクションの管理

Ultra Light でのトランザクション管理の補足情報については、『Ultra Light データベース・ユーザーズ・ガイド』> 「トランザクション処理、リカバリ、バックアップ」を参照してください。

Ultra Light C++ コンポーネントは、オートコミット・モデルをサポートしません。トランザクションが明示的に完了する必要があります。

❖ 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。

- Connection.Commit() 文を実行します。

❖ 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。

- Connection.Rollback() 文を実行します。

詳細については、「クラス UltraLite_Connection_iface」286 ページ 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。

スキーマの修正については、「データベース・スキーマのアップグレード」18 ページを参照してください。

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、Connection.GetSchema を使用します。

  詳細については、「GetSchema 関数」290 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。

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

  詳細については、「GetSchema 関数」342 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。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、TableSchema.GetIndex、TableSchema.GetOptimalIndex、または TableSchema.GetPrimaryKey メソッドを呼び出します。

詳細については、「クラス UltraLite_Table_iface」338 ページ を参照してください。

- **PublicationSchema** パブリケーションに含まれるテーブルとカラムのリスト。パブリケーションもスキーマのみで構成されているため、Publication オブジェクトは存在しません。

  PublicationSchema オブジェクトを取得するには、DatabaseSchema.TableSchema.GetPublicationSchema メソッドを呼び出します。

  詳細については、「GetSchema 関数」342 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、エラーをチェックしてください。ULSqlca オブジェクトのメソッドを使用してチェックできます。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。値の意味については、『ASA エラー・メッセージ』> 「Adaptive Server Anywhere SQLCODE 順にリストしたエラー・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、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るため、アプリケーションの開発中は特に効果的です。コールバックの使用については、「チュートリアル：C++ コンポーネントを使用したアプリケーションの構築」179 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」248 ページと 「ULRegisterErrorCallback 関数」259 ページを参照してください。

Ultra Light C++ コンポーネント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、『ASA エラー・メッセージ』> 「Adaptive Server Anywhere SQLCODE 順にリストしたエラー・メッセージ」を参照してください。

詳細については、「クラス ULSqlca」276 ページ を参照してください。

ユーザの認証

新しいユーザは既存の接続から追加します。Ultra Light のすべてのデータベースは、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、まずは初期ユーザとして接続します。

ユーザ ID の変更はできません。ユーザを 1 人追加して既存のユーザを削除します。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。

詳細については、『Ultra Light データベース・ユーザーズ・ガイド』>「Ultra Light のユーザ認証」を参照してください。

❖ 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. DBA 権限のあるユーザとしてデータベースに接続します。
2. Connection.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。

   詳細については、「クラス UltraLite_Connection_iface」286 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. DBA 権限のあるユーザとしてデータベースに接続します。
2. Connection.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。

データの暗号化

Ultra Light C++ コンポーネントを使用して、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。

補足情報については、『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light データベースの暗号化」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。CreateAndOpenDatabase 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。

詳細については、『Ultra Light データベース・ユーザーズ・ガイド』> 「EncryptionKey 接続パラメータ」を参照してください。

UltraLite_Connection.ChangeEncryptionKey メソッドを使用して新しい暗号化キーを指定することで、暗号化キーを変更できます。

詳細については、「クラス UltraLite_Connection_iface」286 ページ を参照してください。

データベースが暗号化された後は、データベースへ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。

難読化

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

データベースの暗号化の詳細については、『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light データベースの暗号化」を参照してください。

データの同期

SQL Anywhere Studio 9.0 のユーザは、Ultra Light アプリケーションを統合データベースと同期できます。同期には、SQL Anywhere Studio に付属の Mobile Link 同期ソフトウェアが必要です。

この項では、同期の概要について簡単に説明し、Ultra Light C++ コンポーネントのユーザにとって特に関心があるいくつかの機能について説明します。

同期の詳細については、『Mobile Link クライアント』> 「Ultra Light クライアント」および 『Mobile Link クライアント』> 「Ultra Light 同期パラメータ」を参照してください。

Ultra Light C++ コンポーネントは、TCP/IP、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。

注意
HTTPS または TCP/IP 上の暗号化を使用して同期するには、別途ライセンスを取得できるセキュリティ・オプションが必要です。このオプションのご注文については、ご購入いただいた販売代理店または弊社営業担当までご連絡ください。

詳細については、『SQL Anywhere Studio の紹介』> 「SQL Anywhere Studio へようこそ」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++ コンポーネント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 XP 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、SQL Anywhere インストール環境の *UltraLite¥palm*、*UltraLite¥ce*、*UltraLite¥win32* の各サブディレクトリにあります。

Palm OS のランタイム・ライブラリは次のとおりです。

- **ulrt.lib** 静的ライブラリ。これは *UltraLite¥palm¥68k¥lib¥cw* にあります。

Windows CE ライブラリは *UltraLite¥ce¥<platform>* ディレクトリにあります。なお、プラットフォームは *arm*、*386*、*mips*、*emulator* のいずれかです。Windows CE の場合は、静的ライブラリと動的ライブラリの両方が提供されます。

- **ulrt9.dll** ユニコード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 *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** ユニコード文字セットの静的ライブラリ。これは *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。これは *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 XP のランタイム・ライブラリ

UltraLite¥win32¥386 ディレクトリには、Windows CE 以外の 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。次のライブラリがあります。

- **ulrt9.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 *UltraLite¥win32¥386¥ulimp.lib* にリンクします。このライブラリを使用して作成したデータベースは、Windows CE がユニコードを採用しているため、Windows CE デバイスには配備できません。

  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
  ```

- **ulrtw9.dll** ユニコード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 *UltraLite¥win32¥386¥ulimpw.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。

  ```
  /DUNICODE /DUL_USE_DLL
  ```

- **ulrtcw9.dll** ユニコード文字セットのダイナミック・リンク・ライブラリは、複数のプロセスに単一の Ultra Light データベースへのアクセスを可能にする Ultra Light エンジンとともに使用します。このライブラリを使用するには、アプリケーションをインポート・ライブラリ

UltraLite¥win32¥386¥ulimpcw.lib にリンクします。このライブラリを使用して作成したデータベースには、Ultra Light エンジンと DLL の配布が必要です。

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
```

Ultra Light エンジンについては、『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light エンジンの使用」を参照してください。

第 3 章

静的型 C++ API を使用したアプリケーションの開発

この章の内容

この章では、Ultra Light 静的型 C++ API を使用してアプリケーションを開発する方法について説明します。このインタフェースは、Ultra Light データベース内の定義済みのクエリやテーブルをオブジェクトとして表し、SQL を使用せずにそれらのオブジェクトをアプリケーションから操作できるメソッドを提供します。

概要

この章では、静的型 C++ API を使用して Ultra Light アプリケーションを記述、構築する開発者が注意しなければならない点について説明します。

この章の内容

この章には以下の情報があります。

- アプリケーションで使用するデータ・アクセス機能に関する情報

  「アプリケーションの機能の定義」51 ページを参照してください。

- リファレンス・データベースからの C++ API クラスの生成に関する情報

  「Ultra Light C++ クラスの生成」70 ページを参照してください。

- Ultra Light C++ API アプリケーションのコンパイルとリンクに関する注意

  「アプリケーションのコンパイルとリンク」71 ページを参照してください。

始める前に

C++ API を使った開発プロセスは、他の Ultra Light 開発モデルと似ています。この章は、Ultra Light 開発プロセスについての知識があることを前提としています。

詳細については、『Ultra Light データベース・ユーザーズ・ガイド』>「Ultra Light 静的インタフェースの使用」を参照してください。

アプリケーションの機能の定義

Ultra Light アプリケーションに含める SQL 文と Ultra Light データベース自体の構造を定義するには、使用するアプリケーション用にリファレンス・データベースに SQL 文を追加します。

プロジェクトの定義

Ultra Light ジェネレータを実行すると、特定の「**プロジェクト**」にあるすべての SQL 文のクラス定義が書き出されます。プロジェクトは、リファレンス・データベースに定義された名前です。これは、アプリケーションの SQL 文をグループ化します。複数のプロジェクトを定義することにより、1 つのリファレンス・データベースに複数のアプリケーションの SQL 文を保管できます。

プロジェクトの作成については、『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light プロジェクトの作成」を参照してください。

プロジェクト定義を削除するには、**ul_delete_project** ストアド・プロシージャを使用します。

プロジェクトに文を追加する

Ultra Light プロジェクトへの SQL 文の追加については、『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light プロジェクトに SQL 文を追加する」を参照してください。

プレースホルダの使用と、Ultra Light で使用する SQL 文の作成に関する他の機能については、『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light SQL 文の記述」を参照してください。

Ultra Light テーブルの定義

ジョインを実行しないのであれば、アプリケーション実行プログラムのサイズに厳しい制約がある場合は、Ultra Light アプリケーションに対してクエリの代わりにテーブルを定義できます。

リファレンス・データベースにパブリケーションを作成することによって、静的型 C++ API アプリケーションで使用するデータベースのサブセットを定義します。パブリケーションは、Ultra Light アプリケーションに含めるテーブルとそのカラムのセットを定義します。パブリケーションの使用は Ultra Light にとってあくまでも便宜的なものであり、SQL Remote または Mobile Link 同期との接続を意味するものではありません。

パブリケーションによって、ユーザがサブクエリとパラメータを使って取得するローの条件を指定できます。これらのデバイスは、Ultra Light で使用するパブリケーションを作成するときには使用できません。Ultra Light クラスの定義では、テーブルとそのカラムのセットのみ使用します。

テーブルかクエリか

テーブル定義またはクエリ定義を使用して、Ultra Light データベースに含めるデータとそのデータに対して実行する操作の範囲を定義することもできます。

アプリケーションを定義するには、SQL 文とプロジェクトを使用するアプローチが一般的です。このアプローチは、大規模なエンタープライズ・アプリケーションで最も多く使用されます。テーブル定義は、次のような場合に便利です。

- アプリケーションが一度に 1 つのテーブルのデータにしかアクセスしない場合。テーブル定義を使ってジョインを定義することはできません。
- メモリの容量に厳しい制限がある場合。テーブル定義は構造がより簡明であるため、生成されるコードはクエリの場合より短くなります。

静的型 C++ API アプリケーションのデータベース機能の定義

静的型 C++ API アプリケーションは、クラス階層にない関数を使用することがあります。これらの関数は、データベース・ストレージやアクセス設定などを制御します。次の関数があります。

- 「ULEnableFileDB 関数」253 ページ
- 「ULRegisterSchemaUpgradeObserver 関数」263 ページ
- 「ULEnablePalmRecordDB 関数」256 ページ
- 「ULEnableStrongEncryption 関数」257 ページ
- 「ULEnableUserAuthentication 関数」258 ページ

データベース・ストレージの他の特性は、UL_STORE_PARMS マクロを使用して設定します。詳細については、「UL_STORE_PARMS マクロ」270 ページを参照してください。

データベースへの接続

ULData オブジェクトは、データベース・オブジェクト内のデータをアプリケーションで使用可能にします。Ultra Light データベースに接続したり、データを操作したりするためには、事前に **ULData::Open()** を呼び出す必要があります。

ULData::Open() メソッドは、記憶領域とデータベースのアクセス・パラメータ (ファイル名、キャッシュ・サイズ、予約サイズ) を定義するパラメータを指定して呼び出すことができます。

ULData オブジェクトを開くと、データベース上で接続を開くことができます。接続を開くには、**ULConnection::Open()** メソッドを使用します。このメソッドには、**ULData** オブジェクトの参照と接続を確立する接続パラメータのセットを指定します。1 つのデータベースで複数の接続を使用できます。接続を確立すると、アプリケーションで使用されているテーブルまたは文を定義する **ULStatement**、**ULResultSet**、または **ULTable** オブジェクトが生成されるので、これらのオブジェクトを開き、データの操作に使用できます。

ULConnection オブジェクトは、データとの対話方法の一般的な特性を定義します。

同期は **ULConnection** オブジェクトを使って実行されます。Mobile Link サーバとのデータ同期は **Synchronize** メソッドを使って実行します。

データへのアクセス

各テーブルまたはクエリはクラスによって表されます。テーブルやクエリのローにアクセスや修正を行う API は、SQL「カーソル」を基にしています。これは、テーブルやクエリの位置を示すポインタです。

カーソルは次の位置を持つことができます。

- **最初のローの前**　この位置の値は 0 です。これは、テーブルまたはクエリを開いたときのカーソル位置です。
- **ローの上**　テーブルまたはクエリにローが *n* 行ある場合、カーソル位置 1 ～ *n* がそれぞれのローに対応します。
- **最後のローの後**　この位置の値は (*n* + 1) です。

オブジェクトのローからローへ移動するには、そのオブジェクトの **Next()** や **Previous()** などのメソッドを使用します。

Palm Computing Platform の開発者

Palm Computing Platform 用のアプリケーションを開発する場合は、これらのオブジェクトを使用する方法について他にもいくつか留意しなければならない点があります。

詳細については、『Mobile Link クライアント』> 「ネットワーク・プロトコルの選択」を参照してください。

ローの順序付け

オブジェクトのローの順序はオブジェクトを開いたときに決まります。デフォルトでは、テーブルはプライマリ・キーによって順序付けられます。Ultra Light ジェネレータは、オブジェクト定義の列挙を追加します。この列挙では、各メンバがリファレンス・データベースのテーブルの各インデックス (プライマリ・キーは **Primary**) に対応します。この列挙のメンバを指定することにより、オブジェクト内のローの順序を制御できます。

ローが現在の位置ではなくなるようにローを更新すると、カーソルの現在のローがそのローに移動します。

たとえば、値 A、B、C、E が入った単一カラムのオブジェクトがあるとします。

- カーソルがロー B (位置 2) にあるときに、値を D に変更すると、そのローは C と E の間に移動し (位置 3 となる)、カーソルの現在のローが位置 3 に変わります。

ローを挿入した場合、現在位置はそのローに移動しません。

ユーザの認証

新しいユーザは既存の接続から追加します。Ultra Light のすべてのデータベースは、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、まずは初期ユーザとして接続します。

ユーザ ID の変更はできません。ユーザを 1 人追加して既存のユーザを削除します。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。

詳細については、『Ultra Light データベース・ユーザーズ・ガイド』>「Ultra Light のユーザ認証」を参照してください。

❖ ユーザ認証を有効にするには、次の手順に従います (静的型 C++ API の場合)。

1. *ulapi.cpp* のコンパイル時に、コンパイラ指令 UL_ENABLE_USER_AUTH を定義します。
2. **ULEnableUserAuthentication** を呼び出してからデータベースを開きます。次に例を示します。

```
ULData db;
 ...
 ULEnableUserAuthentication( &sqlca );
 db.open();
 ...
```

次のサンプル・コードは、静的型 C++ API Ultra Light アプリケーションのユーザ管理と認証を実行します。

完全なサンプルは、SQL Anywhere ディレクトリの *Samples¥UltraLite¥apiauth* サブディレクトリにあります。次のコードは *Samples¥UltraLite¥apiauth¥sample.cpp* の一部です。

```
ULEnableUserAuthentication( &sqlca );
 db.Open() ;
    if( conn.Open( &db,
                   UL_TEXT( "dba" ),
                   UL_TEXT( "sql" ) ) ){
       // prompt for new user ID and password
       printf("Enter new user ID and password¥n" );
```

```
        scanf( "%s %s", uid, pwd );
        if( conn.GrantConnectTo( uid, pwd ) ){
           // new user added, remove dba
           conn.RevokeConnectFrom( UL_TEXT("dba") );
        }
        conn.Close();
     }
     // regular connection
     printf("Enter user ID and password¥n" );
     scanf( "%s %s", uid, pwd );
     if( conn.Open( &db, uid, pwd )){
     ...
```

このコードでは、次のタスクを実行します。

1. データベースのオブジェクトを開いて、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。

詳細については、「GrantConnectTo メソッド」385 ページおよび「RevokeConnectFrom メソッド」390 ページを参照してください。

データの暗号化

Ultra Light 静的型 C++ API を使用して、Ultra Light データベースを暗号化したり、難読化したりできます。

補足情報については、『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light データベースの暗号化」を参照してください。

暗号化

Ultra Light データベースは、最初の接続時に作成されます。Ultra Light データベースを暗号化するには、暗号化キーを指定してから最初の接続を行います。最初の接続時に、指定したキーを使用することでデータベースが暗号化されます。以降の接続では、指定したキーと暗号化キーが照合され、一致しない場合は接続が失敗します。

❖ Ultra Light データベースを高度に暗号化するには、次の手順に従います。

1. 暗号化モジュールをロードします。

   ULEnableStrongEncryption を呼び出してからデータベースを開きます。

   ULData::Open を呼び出してデータベースを開きます。

2. 暗号化キーを指定します。

   パラメータ名 **key** を指定して UL_STORE_PARMS マクロを定義します。

   ```
   #define UL_STORE_PARMS "key=a secret key"
   ```

   ほとんどのパスワードと同様、最善の方法は、簡単には推測できないキー値を選択することです。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。また、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。

引用符もキーの一部とみなされてしまいます。データベースを起動するたびに、キーを指定してください。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。

詳細については、「UL_STORE_PARMS マクロ」270 ページを参照してください。

3 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デ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。

  詳細については、「ULChangeEncryptionKey 関数」451 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。**

- 生成されたデータベースをコンパイルするときに、UL_ENABLE_OBFUSCATION コンパイラ指令を定義します。

  詳細については、「UL_ENABLE_OBFUSCATION マクロ」269 ページを参照してください。

データの同期

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 同期サーバと 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 (C/C++) 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. 同期構造体を初期化します。

   詳細については、「同期パラメータの初期化」108 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。

   詳細については、『Mobile Link クライアント』> 「Ultra Light 同期クライアントのネットワーク・プロトコルのオプション」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。

   詳細については、「同期を呼び出す」63 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。詳細については、「同期の前に変更をコミットする」64 ページを参照してください。

同期パラメータ

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、『Mobile Link クライアント』> 「Ultra Light 同期クライアントのネットワーク・プロトコルのオプション」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、C/C++ 構造体に保管されます。

C/C++ の場合、構造体のメンバは、初期化時に完全に定義されていないことがあります。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 *ulglobal.h* で宣言された構造体で定義されます。

同期パラメータの完全なリストについては、『Mobile Link クライアント』> 「Ultra Light 同期パラメータ」を参照してください。

❖ **同期パラメータを初期化するには、次の手順に従います (静的型 C++ API の場合)。**

- **Connection** オブジェクトで **InitSynchInfo()** メソッドを呼び出します。次に例を示します。

```
auto ul_synch_info synch_info;
 conn.InitSynchInfo( &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 `50`、パスワードは空、スクリプト・バージョンは `custdb` です。Mobile Link 同期サーバはアプリケーション (`localhost`) と同じマシンで実行されており、デフォルトのポート (2439) を使用しています。

```
auto ul_synch_info synch_info;
 conn.InitSynchInfo( &synch_info );
 synch_info.user_name = UL_TEXT( "50" );
 synch_info.version = UL_TEXT( "custdb" );
 synch_info.stream = ULSocketStream();
 synch_info.stream_parms =
    UL_TEXT("host=localhost");
 conn.Synchronize( &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同期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、HTTP、または HTTPS ストリームの場合)。

- **Connection.InitSynchInfo()** を呼び出して同期パラメータを初期化し、**Connection.Synchronize()** を呼び出して同期を行います。「Synchronize メソッド」393 ページを参照してください。

❖ 同期を呼び出すには、次の手順に従います (HotSync の場合)。

- **Connection.InitSynchInfo** 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。

  詳細については、「ULSetSynchInfo 関数」476 ページを参照してください。

  HotSync 同期は、アプリケーションの範囲外で管理されます。

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 同期サーバ・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、『Mobile Link クライアント』> 「Download Only 同期パラメータ」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。

開発のヒント
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に記述します。プロトタイプが正常に動作するようになったら、同期を有効にして、一時的に使用した INSERT 文を削除します。

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

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。

- 同期の進行状況のモニタや、同期のキャンセルに使用する API。
- 使用するアプリケーションに追加可能なインタフェースを実装する進行状況インジケータ・コンポーネント。

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呼び出される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。

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_state    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_RECEIVING_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。

  同期処理の概要については、『Mobile Link 管理ガイド』>「同期処理」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。

- **stops** 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、SQL Anywhere ディレクトリのサブディレクトリ *Samples¥UltraLite¥CustDB* 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。

静的型 C++ API アプリケーションの構築

この項の項目は、次のとおりです。

- 「Ultra Light C++ クラスの生成」70 ページ
- 「アプリケーションのコンパイルとリンク」71 ページ

コンパイル用の makefile を含む、小さなサンプル・アプリケーションをいくつか用意しています。これらのアプリケーションは、*Samples¥UltraLite* ディレクトリのサブディレクトリに保存されています。

Ultra Light C++ クラスの生成

ジェネレータは、データベース内のパブリケーションからテーブル・クラスを生成し、ul_add_statement ストアド・プロシージャで追加された SQL 文からクエリ・クラスを生成して、出力を次のファイルに書き込みます。

- *filename.hpp* このファイルには生成されたインタフェースのプロトタイプが収められています。このファイルを調べて、アプリケーションで使用する API を決定します。
- *filename.cpp* このファイルにはインタフェースのソースが収められています。このファイルは見る必要はありません。
- *filename.h* このファイルには Ultra Light が必要とする内部定義が収められています。このファイルは見る必要はありません。
- ここで、*filename* は **ulgen** コマンド・ラインに指定された名前です。

アプリケーションのクラスを定義するために、プロジェクト内のクエリ、パブリケーション、またはその両方のいずれを使用する場合でも、Ultra Light ジェネレータの 1 回の実行ですべてのコードを生成してください。

❖ **パブリケーション用の Ultra Light コードを生成するには、次の手順に従います。**

- `-u` コマンド・ライン・スイッチでパブリケーション名を指定して Ultra Light ジェネレータを実行します。次に例を示します。

```
ulgen -c "uid=dba;pwd=sql" -t c++ -u pubName -f
filename
```

❖ **Ultra Light プロジェクトの Ultra Light コードを生成するには、次の手順に従います。**

- `-j` コマンド・ライン・スイッチでプロジェクト名を指定して Ultra Light ジェネレータを実行します。次に例を示します。

```
ulgen -c "uid=dba;pwd=sql" -t c++ -j projectname -f
filename
```

❖ **プロジェクトとパブリケーションの両方の Ultra Light コードを生成するには、次の手順に従います。**

- プロジェクト名とパブリケーション名を指定して Ultra Light ジェネレータを実行します。次に例を示します。

```
ulgen -c "uid=dba;pwd=sql" -t c++ -j projectname -u
pubname -f filename
```

Ultra Light ジェネレータの詳細については、『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light ジェネレータ」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

Ultra Light アプリケーションをコンパイルするときには、コンパイラが必要なすべてのファイルを見つけられるようにしてください。

- **生成されたソース・ファイル**　プロジェクト内の API が記述された生成ファイルを含めます。これには、生成された *.cpp* ファイル、*h* ファイル、*.hpp* ファイルが含まれます。

- **Ultra Light ヘッダ・ファイル**　Ultra Light ヘッダ・ファイルを見つけられるようにコンパイラを設定します。

  これらのヘッダ・ファイルは、Adaptive Server Anywhere インストール・ディレクトリの *h* ディレクトリ内にインストールされています。

- **Ultra Light c ファイル**　Ultra Light API ファイル *ulapi.cpp* を見つけられるようにリンカを設定します。

  このファイルは、Adaptive Server Anywhere インストール・ディレクトリの *src* サブディレクトリにインストールされています。

- **ライブラリまたはインポート・ライブラリ**　ターゲット・プラットフォーム用の Ultra Light ランタイム・ライブラリ、または Ultra Light ランタイム DLL を使用する場合は、Ultra Light インポート・ライブラリを見つけられるようにコンパイラを設定します。

  これらのファイルは、Adaptive Server Anywhere インストール・ディレクトリの *UltraLite* サブディレクトリにインストールされています。各ターゲット・プラットフォームごとに対応するディレクトリがあり、1 つのプラットフォームに複数のプロセッサがある場合は、それぞれに固有のサブディレクトリがあります。

コンパイル・オプションが含まれているサンプル・アプリケーションについては、*Samples¥UltraLite¥apitutorial* 内のファイルを参照してください。

第 4 章

Embedded SQL を使用したアプリケーションの開発

この章の内容

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

始める前に

この章は、Ultra Light 開発プロセスについての基礎知識があることを前提としています。概要については、『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light 静的インタフェースの使用」を参照してください。

リファレンス情報については、「Embedded SQL API リファレンス」445 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、『ASA プログラミング・ガイド』> 「SQL プリプロセッサ」を参照してください。

概要

次に、ごく簡単な Embedded SQL プログラムを示します。このプログラムは、従業員 195 の姓を更新し、変更をコミットします。

```
#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 = 'Plankton'
       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 );
 }
```

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、この例では、次のような、Embedded SQL アプリケーションにおける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、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、その例として挙げられます。

データの処理をする前に

前述の例では、初期化に必要な文が定義されています。初期化に必要な文は、データベースのデータを処理する前に挿入しなければなりません。

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。

   詳細については、「db_init 関数」447 ページを参照してください。

3. CONNECT 文を使用して、データベースに接続します。

終了の準備

この例では、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。

データベースの変更をコミットせずに終了すると、コミットされていない操作は、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 文がエラーを発生させた後に実行されます。

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。Embedded SQL 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;
```

C プログラムで実行する最初の Embedded SQL 文は CONNECT 文にします。使用しているアプリケーションに Ultra Light ユーザ認証が組み込まれていない場合は、この CONNECT 文は無視されます。

Embedded SQL アプリケーションでの Ultra Light ユーザ認証については、「ユーザの認証」102 ページと 『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light のユーザ認証」を参照してください。

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 Communication Area) の初期化

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

SQLCA は、インストール・ディレクトリの *h* サブディレクトリにある、*sqlca.h* ヘッダ・ファイルで定義されています。

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を起こすと、フィールド **sqlcode** にエラー・コードが入ります (次の項を参照)。**sqlcode** などの SQLCA のフィールドを参照するために、C マクロが定義されています。

SQLCA のフィールド

SQLCA のフィールドの意味を次に示します。

- **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 (136 バイト) 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 *sqlerr.h* にあります。エラー・コードは、0 (ゼロ) は成功、正は警告、負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、『ASA エラー・メッセージ』> 「ASA 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 文は、次の 2 つの形式のいずれかを指定します。

EXEC SQL CONNECT "*user-name*" IDENTIFIED BY "*password*";

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

1 番目の構文では、起動パラメータが UL_STORE_PARMS で提供されることが前提です。詳細については、「UL_STORE_PARMS マクロ」270 ページを参照してください。

2 番目の構文では、UL_STORE_PARMS の設定が無視され、データベース起動パラメータ (特にファイル名) が明示的に指定されます。

CONNECT 文の詳細については、『ASA SQL リファレンス・マニュアル』> 「CONNECT 文 [ESQL] [Interactive SQL]」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。

❖ アプリケーションで複数の SQLCA を管理するには、次の手順に従います。

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。

   詳細については、「db_init 関数」447 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別の SQLCA を使用することを伝えます。通常は、次のような文を使用します。

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

こ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。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、『ASA SQL リファレンス・マニュアル』> 「SET SQLCA 文 [ESQL]」を参照してください。

1 つの SQLCA の使用

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。

詳細については、『ASA SQL リファレンス・マニュアル』> 「SET CONNECTION 文 [Interactive SQL] [ESQL]」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に値を送ったり、データベースから値を受け取ります。ホスト変数とは、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

ホスト変数は、「**宣言セクション**」に定義します。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、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。

例

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 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; /* e.g., 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できます。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、『ASA データベース管理ガイド』> 「データベース・オプション」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。これらのオプションは、Ultra Light プログラムの実行中には設定できませんが、プログラムの生成に使用したリファレンス・データベースでの設定と同じ値が使用されます。

- **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 K を超えるデータには、DECL_LONGVARCHAR 構造体を使用できます。このように大きいデータの場合は、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 K を超えるデータには、DECL_LONGBINARY 構造体を使用できます。このように大きいデータの場合は、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 *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、データベース環境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、これらの指令を無視するため、#if 指令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 指令内の宣言を無視し、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

ホスト変数は単純な名前でなければならないため、SQL プリプロセッサはポインタや参照式を認識しません。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 プリプロセッサ指令で囲む。SQL プリプロセッサはプリプロセッサ指令を無視するため、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 指令があるため 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。

```
EXEC SQL INCLUDE SQLCA;
 #include <sqlerr.h>
 #include <stdio.h>
 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指令を無視するため、*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` 指令は、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、メモリのロケーションをポイントしていないポインタ値を表します。

Adaptive Server 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ほかに何かが必要です。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。

エラーと警告は SQLCA 構造体で返されます。詳細については、「SQLCA (SQL Communication Area) の初期化」77 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可能性がある場合。

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選択すると、データベース・サーバは SQLE_TOO_MANY_RECORDS エラーを返す。
- クエリがローを選択しなかった場合、警告 SQLE_NOTFOUND が返される。

エラーと警告は SQLCA 構造体で返されます。詳細については、「SQLCA (SQL Communication Area) の初期化」77 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、『ASA プログラミング・ガイド』>「カーソルを使用した操作」を参照してください。

❖ ESQL 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 Communication 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:
```

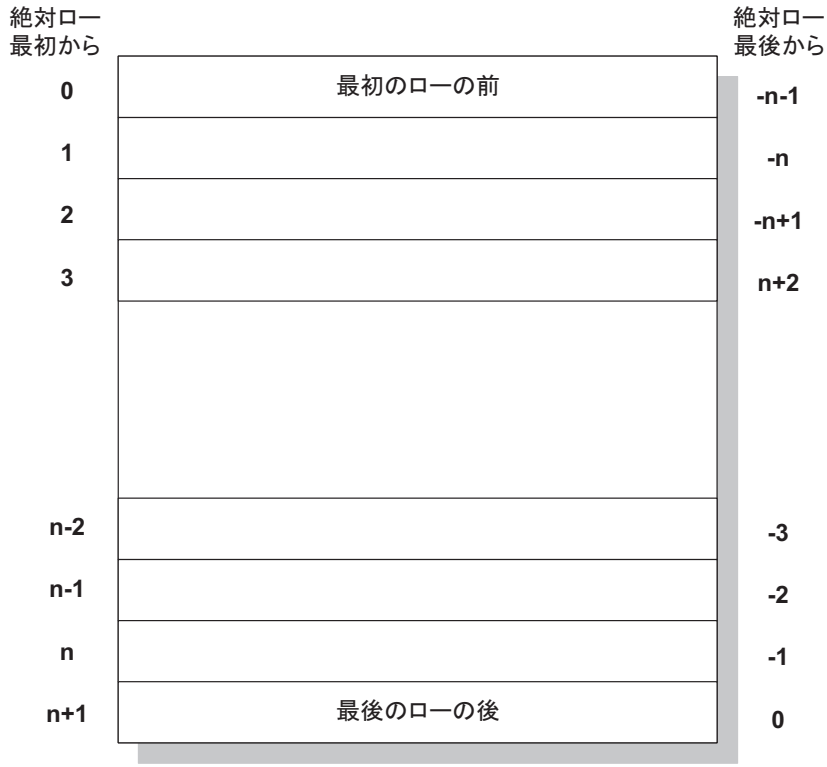
```
                    %s¥n",name, sex, birthdate );
        }
        /* 4. Close the cursor. */
        EXEC SQL CLOSE C1;
    }
```

FETCH 文の詳細については、『ASA SQL リファレンス・マニュアル』> 「FETCH 文 [ESQL] [SP]」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。

カーソル内のローの順序
カーソルに特定の順序が必要な場合は、カーソルの定義で SELECT 文に ORDER BY 句を含めます。この句がない場合は、順序は制御できず、順序が毎回異なる可能性があります。

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、UPDATE (*位置付け*) 文と DELETE (*位置付け*) 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、『ASA SQL ユーザーズ・ガイド』> 「クエリ処理中のワーク・テーブルの使用」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

新しいユーザは既存の接続から追加します。Ultra Light のすべてのデータベースは、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、まずは初期ユーザとして接続します。

ユーザ ID の変更はできません。ユーザを 1 人追加して既存のユーザを削除します。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。

詳細については、『Ultra Light データベース・ユーザーズ・ガイド』>「Ultra Light のユーザ認証」を参照してください。

静的インタフェースの場合は、ユーザ認証機能を有効にしてから、ユーザの追加、ユーザの削除、パスワードの変更を行うようにしてください。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。

❖ ユーザ認証を有効にするには、次の手順に従います (Embedded SQL の場合)。

- **ULEnableUserAuthentication** を呼び出してから **db_init** を呼び出します。次に例を示します。

```
app(){
    ...
    ULEnableUserAuthentication( &sqlca );
    db_init( &sqlca );
    ...
```

  db_init の呼び出しは、アプリケーションでのすべてのデータベース・アクティビティに先行する動作です。

次のサンプル・コードは、Embedded SQL Ultra Light アプリケーションのユーザ管理と認証を実行します。

ユーザ認証の例

完全なサンプルは、SQL Anywhere ディレクトリの *Samples¥UltraLite¥esqlauth* サブディレクトリにあります。次のコードは *Samples¥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;
    ULEnableUserAuthentication( &sqlca );
    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. **ULEnableUserAuthentication** を呼び出してユーザ認証機能を有効にする。
2. **db_init** を呼び出してデータベースの機能を開始する。
3. 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。
4. 接続に成功したら、新しいユーザを追加する。
5. 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。
6. 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。

7. 更新されたユーザ ID とパスワードを使用して接続する。

詳細については、「ULGrantConnectTo 関数」463 ページおよび「ULRevokeConnectFrom 関数」472 ページを参照してください。

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。

補足情報については、『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light データベースの暗号化」を参照してください。

暗号化

Ultra Light データベースは、最初の接続時に作成されます。Ultra Light データベースを暗号化するには、暗号化キーを指定してから最初の接続を行います。最初の接続時に、指定したキーを使用することでデータベースが暗号化されます。以降の接続では、指定したキーと暗号化キーが照合され、一致しない場合は接続が失敗します。

❖ Ultra Light データベースを高度に暗号化するには、次の手順に従います。

1. 暗号化モジュールをロードします。

   ULEnableStrongEncryption を呼び出してからデータベースを開きます。

   db_init を呼び出してデータベースを開きます。

2. 暗号化キーを指定します。

   パラメータ名 **key** を指定して UL_STORE_PARMS マクロを定義します。

   ```
   #define UL_STORE_PARMS "key=a secret key"
   ```

   ほとんどのパスワードと同様、最善の方法は、簡単には推測できないキー値を選択することです。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。また、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。

引用符もキーの一部とみなされてしまいます。データベースを起動するたびに、キーを指定してください。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。

詳細については、を参照してください。

3 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デ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。

  詳細については、「ULChangeEncryptionKey 関数」451 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。**

- 生成されたデータベースをコンパイルするときに、UL_ENABLE_OBFUSCATION コンパイラ指令を定義します。

  詳細については、「UL_ENABLE_OBFUSCATION マクロ」269 ページ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 同期サーバと 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。

   詳細については、「同期パラメータの初期化」108 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。

   詳細については、『Mobile Link クライアント』> 「Ultra Light 同期クライアントのネットワーク・プロトコルのオプション」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。

   詳細については、「同期を呼び出す」109 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。詳細については、「同期の前に変更をコミットする」64 ページを参照してください。

同期パラメータ

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、『Mobile Link クライアント』> 「Ultra Light 同期クライアントのネットワーク・プロトコルのオプション」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、初期化時に完全に定義されないことがあります。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 *ulglobal.h* で宣言された構造体で定義されます。

同期パラメータの完全なリストについては、『Mobile Link クライアント』> 「同期パラメータ」を参照してください。

❖ **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 同期サーバはホスト・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 同期サーバと通信します。Mobile Link 同期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同期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、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。

  詳細については、「ULSetSynchInfo 関数」476 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 同期サーバ・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、『Mobile Link クライアント』> 「Download Only 同期パラメータ」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。

開発のヒント
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に記述します。プロトタイプが正常に動作するようになったら、同期を有効にして、一時的に使用した INSERT 文を削除します。

同期の開発に関するヒントについては、『Mobile Link 管理ガイド』>「開発のヒント」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。数値の意味については、「stream_error 同期パラメータ」534 ページを参照してください。

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。

- 同期の進行状況のモニタや、同期のキャンセルに使用する API。
- 使用するアプリケーションに追加可能なインタフェースを実装する進行状況インジケータ・コンポーネント。

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呼び出される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_state    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。

  同期処理の概要については、『Mobile Link 管理ガイド』>「同期処理」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、SQL Anywhere ディレクトリのサブディレクトリ *Samples¥UltraLite¥CustDB* 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。アプリケーションに *.sqc* ファイルが 1 つしかない場合は、簡単な修正を使用することができます。詳細については、「単一のファイルを使用する構築プロシージャ」122 ページを参照してください。

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。詳細については、『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light 静的インタフェースの使用」を参照してください。

単一の Embedded SQL ファイルを使用するか、複数の Embedded SQL ファイルを使用するかによって、2 通りの構築方法があります。

一般的な構築手順

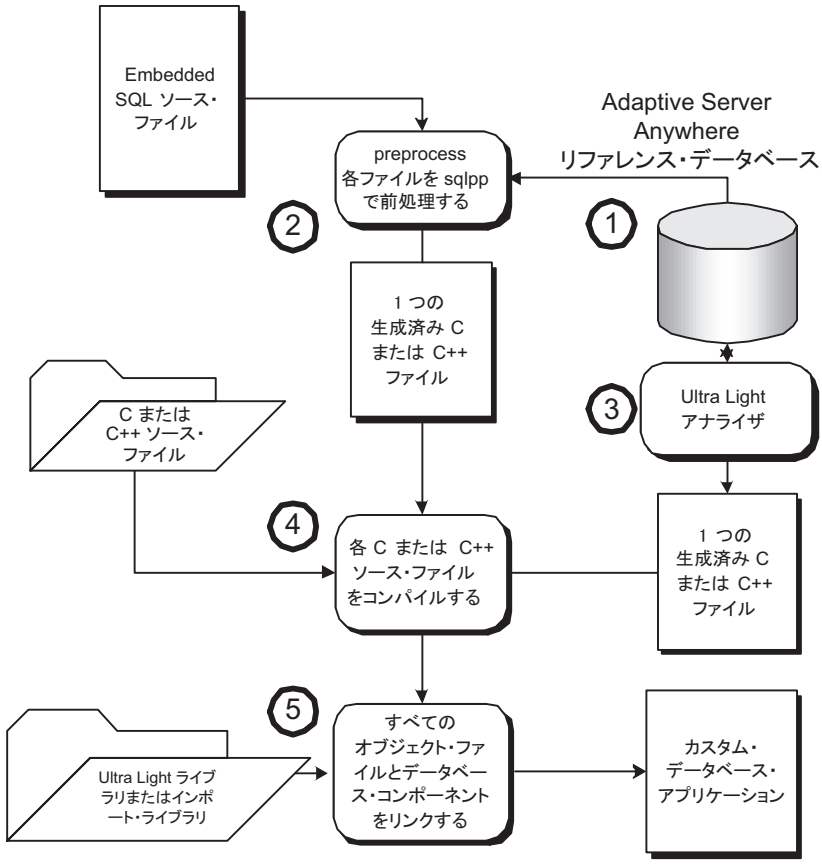
サンプル・コード

このプロセスを使用する makefile は、*Samples¥UltraLite¥ESQLSecurity* ディレクトリに保存されています。そのサンプルを構築するには、個別にライセンスを取得できるトランスポート・レイヤ・セキュリティ・オプションが必要です。

トランスポート・レイヤ・セキュリティ・オプションの取得については、弊社営業担当までご連絡ください。

プロシージャ

次の図は、Ultra Light Embedded SQL アプリケーションを構築する手順を表しています。ソース・ファイルに加えて、アプリケーションで使用するテーブルとインデックスを含むリファレンス・データベースが必要です。



❖ **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. リファレンス・データベースを指定して、Adaptive Server Anywhere パーソナル・データベース・サーバを起動します。

2. 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。SQL プリプロセッサは Ultra Light 開発プロジェクトで次の 2 つの機能を実行します。

- Embedded SQL 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 ファイルを生成します。
- Ultra Light ジェネレータが使用できるように、SQL 文をリファレンス・データベースに追加します。

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 *.c* 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。

---

sqlpp の `-c` コマンド・ライン・オプションを使用してリファレンス・データベースに接続し、`-p` コマンド・ライン・オプションを使用してプロジェクト名を指定します。プロジェクト内の各 Embedded SQL ファイルには、同じプロジェクト名を使用してください。

SQL プリプロセッサの詳細については、『Ultra Light データベース・ユーザーズ・ガイド』> 「SQL プリプロセッサ」を参照してください。

プロジェクトについては、『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light プロジェクトの作成」を参照してください。

3 Ultra Light ジェネレータを実行します。

このジェネレータは Embedded SQL ファイルの前処理中に収集された情報を分析します。ジェネレータは追加コードを準備し、新しい C ソース・ファイルを書き出します。この手順は、使用するリファレンス・データベースによって異なります。

コマンド・プロンプトで次のコマンドを入力します。

```
ulgen -"c "connection-string" options
```

options は、プロジェクトの詳細により異なります。

Ultra Light ジェネレータのコマンド・ラインによって動作がカスタマイズされます。次のコマンド・ライン・スイッチは特に重要です。

- **-c**　リファレンス・データベースに接続するための接続文字列を指定します。

  Adaptive Server Anywhere の接続文字列については、『ASA データベース管理ガイド』> 「接続パラメータ」を参照してください。

- **-f**　出力ファイル名を指定します。

- **-j**　Ultra Light プロジェクト名を指定します。

Ultra Light ジェネレータのオプションの詳細については、『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light ジェネレータ」を参照してください。

4　C または C++ の各ソース・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
- Ultra Light ジェネレータが作成した C ファイル
- アプリケーションを構成する追加の C または C++ ソース・ファイル

5　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。

例

- プロジェクトに *store.sqc* と *display.sqc* という名前の *2* つの Embedded SQL ソース・ファイルがあるとします。次のコマンドを使用して、プロジェクトに *salesdb* という名前を与え、この 2 つを処理できます (コマンドはそれぞれ 1 行に入力)。

```
sqlpp -c "uid=dba;pwd=sql" -p salesdb store.sqc
 sqlpp -c "uid=dba;pwd=sql" -p salesdb display.sqc
```

この 2 つのコマンドは、それぞれ *store.c* ファイルと *display.c* ファイルを生成します。また、Ultra Light アナライザが使用する情報をリファレンス・データベースに保管します。

単一のファイルを使用する構築プロシージャ

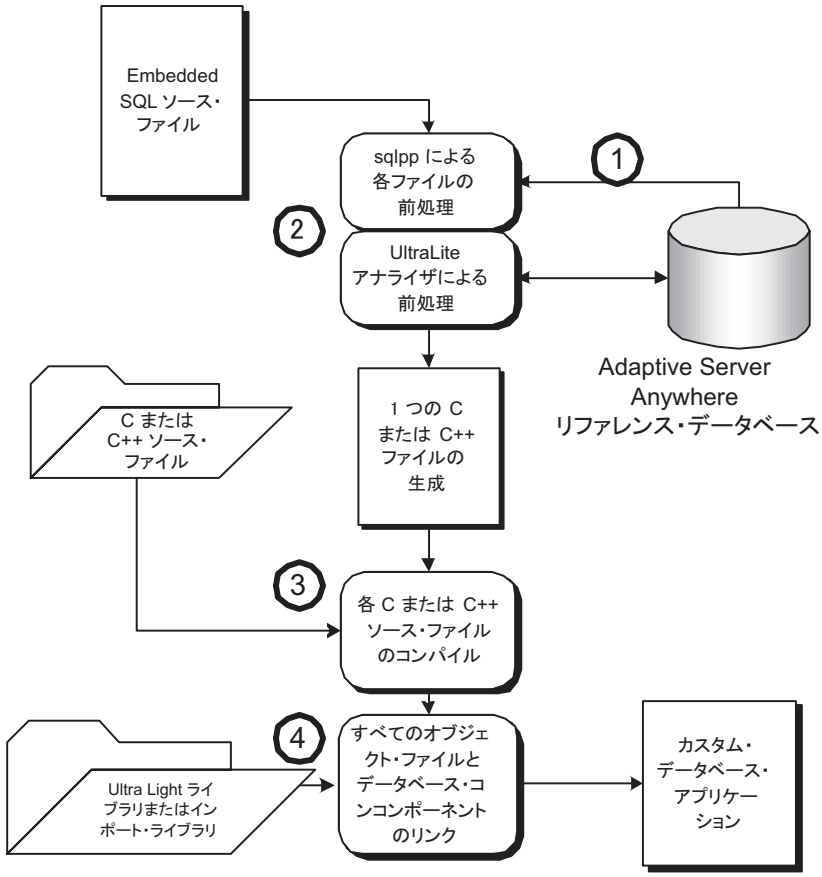
この項は、Embedded SQL の開発モデルを全般的に理解している人を対象にしています。詳細については、『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light 静的インタフェースの使用」を参照してください。

次の条件を満たしていれば、単一のファイルの構築プロシージャを使用できます。

- トランスポート・レイヤ・セキュリティを使用してない。
- 同期にパブリケーションを使用するつもりはない。
- Ultra Light プロジェクトの名前を指定する必要がない。
- 複数の Embedded SQL ソース・ファイルがある。

これらの条件を満たしていない場合は、一般的な構築プロセスを使用する必要があります。手順については、「Embedded SQL アプリケーションの構築」118 ページを参照してください。

次の図は、Ultra Light データベース・アプリケーションを単一のファイルで構築する手順を表しています。ソース・ファイルに加えて、アプリケーションで使用するテーブルとインデックスを含むリファレンス・データベースが必要です。



❖ **Ultra Light アプリケーションを構築するには、次の手順に従います (Embedded SQL ファイルが 1 つの場合)。**

1　リファレンス・データベースを指定して、Adaptive Server Anywhere パーソナル・データベース・サーバを起動します。

2　SQL プリプロセッサを使用した Embedded SQL ソース・ファイルの前処理

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。SQL プリプロセッサは Ultra Light ジェネレータを自動的に実行し、生成された C/C++ ソース・ファイルにコードを追加します。この手順は、使用するリファレンス・データベースとデータベース・サーバによって異なります。

sqlpp の `-c` コマンド・ライン・オプションを使用して、リファレンス・データベースに接続します。単一ファイルを使用する構築プロシージャでは、SQL プリプロセッサのコマンド・ラインでプロジェクトを指定しないでください。

sqlpp のパラメータのリストについては、『ASA プログラミング・ガイド』> 「SQL プリプロセッサ」を参照してください。

3 C または 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 C ファイル
- アプリケーションを構成する追加の C/C++ ソース・ファイル

4 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。

例

- アプリケーションには、*store.sqc* という名前の Embedded SQL ソース・ファイルが *1* つだけ含まれています。次のコマンドを使用してこのファイルを処理できます。プロジェクト名は指定しません。このコマンドにより、SQL プリプロセッサが *store.c* ファイルを記述します。

```
sqlpp -c "uid=dba;pwd=sql" store.sqc
```

また、プリプロセッサは自動的に Ultra Light ジェネレータを実行します。アナライザはアプリケーション・データベースを実装する追加の C/C++ コードを生成します。このコードは自動的に *store.c* ファイルに追加されます。

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 プリプロセッサと Ultra Light ジェネレータ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。このプラグインについては、「Metrowerks CodeWarrior で Ultra Light アプリケーションを開発する」139 ページを参照してください。

また、非常に単純なプロジェクトの詳細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」215 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。これは、Ultra Light ジェネレータが使用する Ultra Light プロジェクト名とは異なります。

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)。

```
"%asany9%¥win32¥sqlpp.exe" -q -o WINNT
        -c connection-string -p project-name
        $(InputPath) $(InputName).c
```

  ここで、*asany9* は、SQL Anywhere インストール・ディレクトリを指す環境変数です。*connection-string* はリファレンス・データベースとの接続を確立します。*project-name* は Ultra Light プロジェクトの名前です。

  Windows 以外のプラットフォームの実行プログラムを生成する場合は、WINNT ではなく、適切な設定を選択してください。

  SQL プリプロセッサのコマンド・ラインの詳細については、『ASA プログラミング・ガイド』> 「SQL プリプロセッサ」を参照してください。

- コマンドの出力を **$(InputName).c** に設定します。

3 *sqc* ファイルをコンパイルし、生成された *.c* 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 *.c* 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 *$(asany9)¥h* と、インクルード・パスに必要なインクルード・フォルダを、カンマで区切ったリストとして追加します。

Ultra Light コードの生成

次に、Ultra Light コード生成を開発ツールに追加する手順について説明します。

❖ Ultra Light コード生成を依存ベースの開発環境に追加するには、次の手順に従います。

1 ダミー・ファイルを開発プロジェクトに追加します。

uldatabase.ulg などの名前のファイルを、生成したファイルと同じディレクトリに追加します。

2 このファイルの構築規則が、Ultra Light ジェネレータ・コマンド・ラインになるように設定します。

Visual C++ では、次のフォームのコマンドを使用します (すべて 1 行に入力)。

```
"%asany9%¥win32¥ulgen.exe" -q -c "connection-string"
$(InputName) $(InputName).c
```

ここで、*asany9* は、SQL Anywhere インストール・ディレクトリを指す環境変数です。*connection-string* はリファレンス・データベースとの接続を確立します。*InputName* は Ultra Light プロジェクトの名前です。この名前は、テキスト・ファイル名のルートと一致するようにしてください。出力は *$(InputName).c* です。

3 ダミー・ファイルがプリプロセッサの出力ファイルに依存するよう設定します。

Visual C++ では、カスタム構築ページの [依存関係] をクリックし、SQL プリプロセッサが生成する *.c* ファイルの名前を入力します。

これにより、Visual C++ は必要な Embedded SQL ファイルをすべて前処理した後に、Ultra Light ジェネレータを実行します。

4 ダミー・ファイルをコンパイルして、Ultra Light データベースを実装する *.c* ファイルを生成します。

5 生成された Ultra Light データベース・ファイルをプロジェクトに追加し、その C/C++ 設定を変更します。

6 Ultra Light インポート・ライブラリをオブジェクト / ライブラリ モジュールのリストに追加します。

Visual C++ で、プロジェクト設定を表示し、[リンク] タブを選択して、次の行を Windows 開発環境の [オブジェクト / ライブラリ モジュール] リストに追加します。

```
$(asany9)¥ultralite¥win32¥386¥lib¥ulimp.lib
```

その他のターゲット・デバイスでは、適切なインポート・ライブラリを選択してください。

7 リファレンス・データベースで SQL 文を変更した場合は、ダミー・ファイルを使用してタイムスタンプを更新し、Ultra Light ジェネレータが強制的に実行されるようにします。

第 5 章

Ultra Light C/C++ インタフェースの共通機能

この章の内容

この章では、Ultra Light C/C++ インタフェースに共通する開発の問題について説明します。また、同じアプリケーションでインタフェースを混在させる方法についても説明します。

SQLCA (SQL Communication 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、各インタフェースは同じ基本機能にアクセスします。

また、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 Communication 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。

次の処理を実行してから、データベースに接続してください。

- 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。
- データ・ストア特性の設定とエラー・コールバック関数の登録。関数のリストについては、「Ultra Light C/C++ 共通 API リファレンス」247 ページを参照してください。
- 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されることがあります。

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
	静的型 C++ API	ULData::Initialize
	C++ コンポーネント	ULSqlca::Initialize
SQLCA の初期化とデータベースの起動	Embedded SQL	db_init db_start_database
	静的型 C++ API	ULData::Open

タスク	インタフェース	関数
	C++ コンポーネント	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 でデータベースを作成するには、スキーマ定義が必要です。生成コードを使用する静的インタフェースの場合、スキーマ定義は生成コードの中に含まれています。Ultra Light C++ コンポーネントの場合、スキーマ定義は外部のスキーマ・ファイルに含まれています。

次の接続プリミティブは生成コードのデータベース・スキーマを理解し、スキーマに一致するデータベースを作成します。

- db_start_database (Embedded SQL)
- ULData::Open (静的型 C++ API)
- EXEC SQL CONNECT 文 (Embedded SQL)

C++ コンポーネントにとって生成コードは必要ないので、生成コードのデータベース・スキーマを参照することはありません。静的アプリケーションに C++ コンポーネント・コードを追加する場合は、生成コードのデータベース・スキーマを理解できるいずれかのプリミティブを使用して、データベースを起動します。外部スキーマ・ファイルがなければ、C++ コンポーネントの DatabaseManager::CreateAndOpenDatabase() メソッドを使用してデータベースを起動することはできません。

データベース名の明示指定

各インタフェースは、データベースに対してそれぞれ異なるデフォルト・ファイル名を使用します。このため、インタフェースを混在させる場合は、データベースを作成したり接続したりするときに、常にデータベースの名前を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 C/C++ インタフェースの結合

1 つのアプリケーションの中で、個々の C/C++ インタフェースを組み合わせて使用できます。さまざまな理由で、このような操作が望まれます。次に例を示します。

- 新しい Ultra Light C++ コンポーネントの機能を、既存の静的インタフェース・アプリケーションに追加する。
- 静的インタフェースでのみサポートされる SQL 文と一緒に動的 SQL の機能を含める。

生成されたデータベース・スキーマの使用

インタフェースを組み合わせて使用する場合、静的コードは外部ファイルに格納されているデータベース・スキーマにアクセスできないため、データベース・スキーマは、静的インタフェースの生成コードで定義する必要があります。C++ コンポーネント機能と静的インタフェース機能を混在させる場合は、静的インタフェースを使用してデータベースを起動してください。つまり、ULData::StartDatabase (静的型 C++ API の場合) または db_start_database (Embedded SQL の場合) を使用してデータベースを起動するということです。

単純な混在

静的インタフェース・アプリケーションに動的 SQL または他の C++ コンポーネント機能を追加する最も簡単な方法は、2 つの機能セットで個別の SQLCA と個別の接続を使用する方法です。一方の SQLCA と接続が C++ コンポーネント・コードを使用し、動的 SQL クエリを実行する一方で、他方の SQLCA と接続が静的インタフェース機能を使用します。この方法の場合、静的インタフェース・コードと C++ コンポーネントの間の調整は必要ありません。個別の接続に関わるオーバヘッドは最小限で済みます。

インタフェースを混在させるこの方法での唯一の制約は、同じトランザクションで動的 SQL と静的文の両方を使用できないことです。

SQLCA の共有

同じトランザクションで静的 SQL と動的 SQL の文を使用するには、インタフェースの間で 1 つの接続を共有する必要があります。接続を共有するには、SQLCA を共有しなければなりません。SQLCA を共有するが、接続は異なるというアプリケーションを記述することもできます。

❖ 1 つの SQLCA を共有するには、次の手順に従います。

1 Ultra Light C++ コンポーネントの ULSqlca オブジェクトを使用して、SQLCA を宣言します。

```
//Declare a SQLCA using C++ Component
 ULSqlca    MySqlca;
```

2 このオブジェクトを使用するのに、静的インタフェースを設定します。

Embedded SQL の場合

```
//Set the embedded SQL SQLCA
 EXEC SQL SET SQLCA "MySqlca.GetCA()";
```

静的型 C++ API の場合

```
ULData MyData
 MyData.Initialize( MySqlca.GetCA() );
```

3 SQLCA を 1 回だけ初期化します。

Ultra Light C++ コンポーネントの ULSqlca::Initialize() メソッドまたは db_init を使用します。次の例では、C++ コンポーネントの初期化を使用しています。

```
//Initialize the SQLCA
 MySqlca.Initialize();
```

4 データベースを起動します。

最初の接続時に、内部生成されたスキーマを使用してデータベースを作成できるように、静的コンポーネントからデータを起動してください。

Embedded SQL の場合

```
db_start_database( );
```

静的型 C++ API の場合

```
MyData.StartDatabase( );
```

C++ コンポーネントの接続と静的インタフェースの接続を別々に使用できます。これらの接続は、同じ SQLCA を共有します。1 つの接続を共有するには、さらに別の手順を実行します。

❖ 1 つの接続を共有するには、次の手順に従います。

1. 前の手順で説明されているように、アプリケーションが 1 つの SQLCA を共有していることを確認します。

2. Ultra Light C++ コンポーネントを使用して、接続を管理します。

   DatabaseManager::OpenConnection メソッドと Connection::Release メソッドを使用して、接続を開いたり閉じたりします。静的インタフェースの接続メカニズムは使用しないでください。

   OpenConnection が戻ると、接続は SQLCA の現在の接続となり、その後、Embedded SQL または静的型 C++ API 文で使用できます。

Ultra Light データベースのデフラグメンテーション

Ultra Light の記憶領域は、空き領域を効率的に再使用するように設計されているので、通常の状況では明示的なデフラグメンテーションは必要ありません。この項では、Ultra Light データベースのデフラグメンテーションを明示的に実行するテクニックについて説明します。このテクニックは、領域要件が極端に厳しいアプリケーションで使用します。

Ultra Light には、データベースの一部分をデフラグメントするデフラグメンテーション・ステップ関数が用意されています。データベース全体のデフラグメンテーションを 1 回で行うには、**ul_true** が返されるまでデフラグメンテーション・ステップ関数をループで呼び出します。この処理は負荷が高くなる可能性があります。また、エラーを検出するために SQLCODE をチェックしてください (ここでのエラーは、通常はファイル I/O エラーになります)。

明示的なデフラグメンテーションは、アプリケーションの制御下で、そのアイドル時間に徐々に行われます。それぞれのステップは、小さな処理です。

詳細については、「ULStoreDefragFini 関数」265 ページ、「ULStoreDefragInit 関数」266 ページ、「ULStoreDefragStep 関数」267 ページを参照してください。

❖ Ultra Light データベースをデフラグメントするには、次の手順に従います。

1. p_ul_store_defrag_info 情報ブロックを取得します。次に例を示します。

```
p_ul_store_defrag_info     DefragInfo;
 //...
 db_init( &sqlca );
 DefragInfo = ULStoreDefragInit( &sqlca );
```

2. アイドル時間中に、UlStoreDefragStep を呼び出して、データベースの一部をデフラグメントします。次に例を示します。

```
ULStoreDefragStep( &sqlca, DefragInfo );
```

3 完了したら、デフラグメンテーション・ブロックを破棄します。次に例を示します。

```
ULStoreDefragFini( &sqlca, DefragInfo );
```

例

この Embedded SQL サンプルでは、デフラグメンテーションは、アプリケーション制御下でアイドル時間中に徐々に行われます。デフラグメンテーションのそれぞれのステップは、小さな処理です。

```
p_ul_store_defrag_info     DefragInfo;

 idle()
 {
    for( i = 0; i < DEFRAG_IDLE_STEPS; i++ ){
       ULStoreDefragStep( &sqlca, DefragInfo );
       if( SQLCODE != SQLE_NOERROR ) break;
    }
 }

 main()
 {
    db_init( &sqlca );
    DefragInfo = ULStoreDefragInit( &sqlca );
    //
    // main application code,
    // calls idle() when appropriate...
    //
    ULStoreDefragFini( &sqlca, DefragInfo );
    db_fini( &sqlca );
 }
```

データベース全体のデフラグメンテーションを 1 回で行うには、**ul_true** が返されるまで **ULStoreDefragStep** をループで呼び出します。この処理は負荷が高くなる可能性があります。また、ファイル I/O エラーなどのエラーを検出するために、SQLCODE をチェックしてください。

第 6 章

Palm Computing Platform 用 Ultra Light アプリケーションの開発

この章の内容

この章では、Palm Computing Platform 用アプリケーションの開発に固有の、開発、配備、同期の詳細について説明します。

概要

この章では、Palm Computing Platform に固有の Ultra Light 開発機能について説明します。

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 バージョン 8 または 9 (Embedded SQL、静的型 C++ API、または Ultra Light C++ コンポーネントを使用)。

  「Metrowerks CodeWarrior で Ultra Light アプリケーションを開発する」139 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。

  サポートされているプラットフォームのリストについては、『SQL Anywhere Studio の紹介』> 「Ultra Light 開発プラットフォーム」および『SQL Anywhere Studio の紹介』> 「Ultra Light ターゲット・プラットフォーム」を参照してください。

- AppForge MobileVB (Ultra Light MobileVB コンポーネントを使用)

  『『Ultra Light for MobileVB ユーザーズ・ガイド』> 「Ultra Light for MobileVB ユーザーズ・ガイド」』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、『SQL Anywhere Studio の紹介』> 「Ultra Light ターゲット・プラットフォーム」を参照してください。

参照

サポートされる各ホスト・プラットフォームに関する情報など、Palm の開発環境の一般的な情報については、Palm Computing Platform Development Zone Web site を参照してください。

サポートされる開発環境の詳細については、『SQL Anywhere Studio の紹介』> 「Ultra Light 開発プラットフォーム」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 バージョン 8 と 9 は、Ultra Light C++ コンポーネント、静的型 C++ API、Embedded SQL を使用する Palm Computing Platform アプリケーションの開発用にサポートされているプラットフォームです。

Ultra Light アプリケーションの構築を容易にするために CodeWarrior プラグインが提供されています。このプラグインは、*UltraLite¥Palm¥68k¥cwplugin* ディレクトリに保存されています。

この項では、CodeWarrior を使用した Ultra Light アプリケーションの開発方法について説明します。Palm Computing Platform に向けた CodeWarrior プログラミングの知識があることを前提とします。

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 をインストールするには、次の手順に従います。

1. CodeWarrior バージョン 8 または 9 を実行中であることを確認します。Metrowerks の Web サイトから CodeWarrior 用のパッチを取得できます。
2. コマンド・プロンプトで、SQL Anywhere ディレクトリの *UltraLite¥palm¥68k¥cwplugin* サブディレクトリに移動します。
3. *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 (すべて 1 行に入力) を実行すると、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
```

パスにスペースが入っている場合は、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 というファイルもあります。このファイルは、*install.bat* と同じように使用して、CodeWarrior から Ultra Light プラグインをアンインストールできます。

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] ー [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++ API アプリケーション

- Palm OS Ultra Light C++ コンポーネント・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、静的型 C++ API と C++ コンポーネントでは標準の C++ ステーショナリです。ほぼ空のソース・ファイルが入っています。

4 Embedded SQL または静的型 C++ API を使用している場合は、プロジェクトのターゲット設定を行います。

C++ コンポーネント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 [ターゲット] タブを選択し、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。使用するリファレンス・データベースなどのプロジェクト設定を入力できます。

前処理

Embedded SQL プロジェクトを構築する場合、Ultra Light プロジェクトによって *sqlpp* と *ulgen* ユーティリティが呼び出されて、すべての *.sqc* ファイルが *.c* または *.cpp* ファイルに変換され、データベース・コードが生成されます。

C++ API プロジェクトを構築する場合、Ultra Light プラグインによって *ulgen* が呼び出されて、Ultra Light API ファイルとデータベース・コードが生成されます。

C++ コンポーネントの場合は前処理が不要なため、ターゲット設定は無視されます。

また、Embedded SQL と静的型 C++ API の場合は、プラグインによって、ヘッダやランタイム・ライブラリなど Ultra Light の必須ファイルへのパスが検索パスに追加されます。

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、既存のプロジェクトを開くときに、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* ファイルと *.ulg* 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。

   これら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、SQL プリプロセッサによって生成されたすべての *.cpp* 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. 静的型 C++ API アプリケーションでは、*.ulg* ダミー・ファイルをダーティとしてマークし、[Ultra Light ファイル] フォルダを削除します。

Ultra Light plug-in for CodeWarrior の使用

Embedded SQL と静的型 C++ API の場合、Ultra Light plug-in for CodeWarrior は、Ultra Light の前処理手順 (Ultra Light ジェネレータを実行し、Embedded SQL アプリケーションの場合は SQL プリプロセッサを実行する) 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 プリプロセッサと Ultra Light ジェネレータを確実に実行できます。

Ultra Light プロジェクト名または生成されたデータベースの名前を変更した場合は、[Ultra Light ファイル] フォルダを削除してください。これにより、生成済みのファイルが再度生成されます。ファイル名の衝突を回避するために、生成されたデータベース名に *.sqc* ファイル名と同じ名前を使用しないでください。

静的型 C++ API Ultra Light プロジェクト内の SQL 文を変更したり、静的型 C++ API プロジェクトで使用されているパブリケーションを変更したりする場合は、ダミーの *.ulg* ファイルを手動で操作して、Ultra Light ジェネレータの実行を要求します。

プラグインが実行するタスクの概要については、『Ultra Light データベース・ユーザーズ・ガイド』> 「開発ツールを静的 Ultra Light 開発用に設定する」を参照してください。

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。プレフィクス・ファイルは、SQL Anywhere Studio のインストール環境にある *h* サブディレクトリの *ulpalmos.h* を使用してください。この設定は CodeWarrior プラグインによって自動的に行われます。

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。

注意

Ultra Light プラグインによって開発環境が拡張モード用に設定されることはありませんが、拡張モード・アプリケーションを構築することは可能です。詳細については、「拡張モード・アプリケーションの構築」145 ページを参照してください。

ULSecureCerticomTLSStream 関数または ULSecureRSATLSStream 関数を使用して、暗号化された同期を実装する場合は、*ulrsa.lib* または *ulecc.lib* を CodeWarrior 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」xix ページを参照してください。

このアプリケーション用のファイルは、SQL Anywhere ディレクトリの *Samples¥UltraLite¥CustDB* サブ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 Computing Platform 固有のファイルは、次のロケーションにあります。

- **cwcommon** CodeWarrior の全バージョンに共通するファイル
- **cw8** CodeWarrior 8 または CodeWarrior 9 用のファイル

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 8 を使用した場合も、プロセスはほとんど同じです。

❖ 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] ー [開く] を選択します。
   - SQL Anywhere ディレクトリのプロジェクト・ファイル *Samples¥UltraLite¥custdb¥cw8¥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] ー [作成] を選択します。

Ultra Light プラグインを使用して、アプリケーションの設定をカスタマイズできます。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」139 ページを参照してください。

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。このモードを使用するには、Ultra Light ランタイム・ライブラリの特定のバージョンを指定する必要があります。拡張モードには、次のランタイム・ライブラリを指定します。

```
ultralite¥palm¥68k¥lib¥cw9_a4a5jt¥ulrt.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 プラグインの使用

UltraLite plug-in for CodeWarrior は、拡張モード・ライブラリをアプリケーションに追加しません。

拡張モード・アプリケーションを Ultra Light プラグインから構築するには、拡張モード・ランタイム・ライブラリを通常のランタイム・ライブラリにコピーするのが最も簡単です。

❖ Ultra Light プラグインから拡張モード・ランタイム・ライブラリを使用するには

1. 標準のランタイム・ライブラリをバックアップします。

   ファイル *ultralite¥palm¥68k¥lib¥cw¥ulrt.lib* を安全な場所にコピーしてください。

2. 拡張モード・ランタイム・ライブラリを通常のランタイム・ライブラリにコピーします。

   ultralite¥palm¥68k¥lib¥cw9_a4a5jt¥ulrt.lib を *ultralite¥palm¥68k¥lib¥cw¥ulrt.lib* にコピーしてください。

   Ultra Light プラグインが、必要なパス、ヘッダ、ランタイム・ライブラリを CW プロジェクトに追加します。

3 更新された Ultra Light ソフトウェアをインストールした場合は、上記の手順を繰り返します。

更新ファイルを Ultra Light ソフトウェアにインストールすると、新しい標準モードのライブラリが、使用している拡張モード・ランタイム・ライブラリにコピーされます。拡張モード・ランタイム・ライブラリを使用するために、上記の手順を繰り返してください。

Ultra Light Palm アプリケーションの状態の保存

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、テーブルが閉じられていなくて、接続オブジェクトが閉じられていない場合にのみ保存されます。つまり、状態を維持するためには、接続オブジェクトを閉じずにアプリケーションを終了する、接続オブジェクトを定義しているルーチンを終了せずにアプリケーションを終了する、または接続オブジェクトの変数に何も代入しなかったり NULL を代入したりせずにアプリケーションを終了する必要があります。

Ultra Light アプリケーションを終了し、別のアプリケーションに切り替えるときは、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。
   - 静的型 C++ API の場合は、次の関数を呼び出します。
     - ULData::Open
     - ULEnablePalmRecordDB または ULEnableFileDB
     - ULConnection::Open
   - Embedded SQL の場合は、次の関数を呼び出します。
     - db_init
     - ULEnablePalmRecordDB または ULEnableFileDB
     - EXEC SQL CONNECT
   - C++ コンポーネントの場合は、次の関数を呼び出します。
     - ULSqlca.Initialize
     - ULInitDatabaseManager
     - ULEnablePalmRecordDB または ULEnableFileDB
     - OpenDatabase

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。また、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。

マルチセグメント・アプリケーションの構築

Palm Computing Platform 用のアプリケーション・コードは、「**セグメント**」に分割します。CodeWarrior では、これらのセグメントのサイズは最大 64 KB です。この項では、セグメントへのコード割り当てを管理する方法について説明します。

Ultra Light アプリケーションには、次の種類のコードが含まれています。

- **ユーザ定義コード**　アプリケーション・コード。SQL プリプロセッサによって生成された *.cpp* ファイルも含まれます。
- **SQL 文用の生成コード**　SQL 文を実行するために Ultra Light アナライザによって生成されたコード。
- **データベース・スキーム用の生成コード**　データベース・テーブルを表すために Ultra Light アナライザによって生成されたコード。
- **ランタイム・ライブラリ**　Ultra Light ランタイム・ライブラリは、マルチセグメント・コードとしてコンパイルされます。Ultra Light ランタイム・ライブラリ用に、ULRT*n* と ULRT*nn* のフォームのセグメント名が予約されています。

マルチセグメント・アプリケーションの構築は、Palm Computing Platform 用のアプリケーション開発における一般的な特徴です。これは、Ultra Light の使用に関係ありません。ここでは、開発ツールを使用したマルチセグメント・アプリケーションの開発について、多少の知識があることを前提とします。ユーザ定義コードは、他の標準的な Palm アプリケーションと同じです。ユーザ定義コードをセグメントに割り当てるときの注意事項については、「ユーザ定義コードのセグメントへの割り当て」152 ページを参照してください。

生成コードは、次の方法でセグメントに分割できます。

- マルチセグメント・コードの生成を有効にし、セグメントの割り当てを Ultra Light アナライザにデフォルトの方法で行わせる。

  詳細については、「マルチセグメント・コード生成の有効化」150 ページを参照してください。

- マルチセグメント・コードの生成を有効にし、セグメントを自分で明示的に割り当てる。

  詳細については、「セグメントの明示的な割り当て」151 ページを参照してください。

マルチセグメント・コード生成の有効化

この項では、Ultra Light アナライザで、デフォルト・スキームを使用してマルチセグメント・コードを生成する方法について説明します。セグメントに対するコードの割り当てを、セグメントに関数を明示的に割り当てることによってカスタマイズすることもできます。詳細については、「セグメントの明示的な割り当て」151 ページを参照してください。

生成コードのセグメントを有効にするには、マクロを定義します。

❖ マルチセグメント・コードの生成を有効にするには、次の手順に従います。

- 次の内容の CodeWarrior プロジェクト用プレフィクス・ファイルを定義します。

```
#define UL_ENABLE_SEGMENTS
 #include "ulpalmos.h"
```

  詳細については、「UL_ENABLE_SEGMENTS マクロ」269 ページを参照してください。

注意

マルチセグメント・コードの生成を有効にしたとき、Ultra Light アナライザのデフォルトの動作は次のようになります。

- 生成されるスキーマ・コードは単一のセグメントに収められ、ULSEGDB という名前のセグメントに割り当てられます。
- C++ API では、生成される文コードは、ULSEGDEF という名前のセグメントに割り当てられます。

- Embedded SQL では、生成される文コードは、*.sqc* ファイルに基づいて生成された名前を持つセグメントに割り当てられます。単一の *.sqc* ファイルのすべてのコードが、1 つのセグメントに収められます。

セグメントの明示的な割り当て

この項では、SQL 文用の生成コードをセグメントに明示的に割り当てる方法について説明します。最初に、「マルチセグメント・コード生成の有効化」150 ページの説明に従って、マルチセグメント・コードの生成を有効にします。

セグメントの明示的な割り当てを行うには、バージョン 8 以降の標準にアップグレードされたデータベースが必要です。

❖ 生成された文コードをセグメントに明示的に割り当てるには、次の手順に従います (Embedded SQL の場合)。

- *.sqc* ファイルを別個のファイルに分割します。各 *.sqc* ファイルに含まれる文用の生成コードは、それぞれ別々のセグメントに配置されます。

❖ 生成された文コードをセグメントに明示的に割り当てるには、次の手順に従います (静的型 C++ API の場合)。

- 次のいずれかを行います。
  - 文を割り当てるセグメントの名前を指定し、各 SQL 文に対して **ul_set_codesegment** プロシージャを呼び出します。

    たとえば、次の文を実行すると、プロジェクト **myproject** 内の **mystmt** 文がセグメント **MYSEG1** に割り当てられます。

```
call ul_set_codesegment(
      'myproject', 'mystmt', 'MYSEG1' )
```

詳細については、『Ultra Light データベース・ユーザーズ・ガイド』> 「ul_set_codesegment システム・プロシージャ」を参照してください。

- Sybase Central から、[Ultra Light プロジェクト] フォルダを開きます。文を右クリックし、ポップアップ・メニューから [プロパティ] を選択します。ボックスにコードのセグメント名を入力します。

ユーザ定義コードのセグメントへの割り当て

ユーザ定義コードのセグメントへの割り当ては、Palm Computing Platform 用のアプリケーションをプログラミングするための標準的な手順です。この項の目的は、Palm プログラマについて再確認することです。

❖ ユーザ定義コードをセグメントに割り当てるには、次の手順に従います (CodeWarrior の場合)。

- 次の行を *.sqc* ファイルまたは *.cpp* ファイルの適切な位置に追加します。

```
#pragma segment segment-name
```

ここで、*segment-name* はセグメントのユニークな名前です。これによって、各 `#pragma` 行より後のコードが強制的に別のセグメントに割り当てられます。

最初のセグメント

PilotMain と **PilotMain** 内で呼び出されるすべての関数は最初のセグメントに入れてください。

必要であれば、起動コードの前に次のような形式の行を追加できます。

```
#pragma segment segment-name
```

ここで、*segment-name* は最初のセグメントの名前です。

プレフィクス・ファイルとセグメントの詳細については、Palm の開発者用マニュアルを参照してください。

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 同期を呼び出すには、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** または **ULData.Close** 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。

```
//Static 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 または ULData.Close を呼び出します。

詳細については、「Ultra Light Palm アプリケーションの状態の保存」147 ページおよび『Mobile Link クライアント』> 「同期パラメータ」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、Mobile Link 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 同期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 同期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、『Mobile Link クライアント』> 「Ultra Light 同期クライアントのネットワーク・プロトコルのオプション」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 同期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。

  詳細については、『Mobile Link クライアント』> 「HotSync 設定の概要」を参照してください。

参照

Mobile Link HotSync コンジットの設定方法など、HotSync の設定については、『Mobile Link クライアント』> 「Mobile Link HotSync コンジットの設定」を参照してください。

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、「データの同期」61 ページを参照してください。

Palm Computing Platform 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・セキュリティについては、『Mobile Link 管理ガイド』> 「Mobile Link トランスポート・レイヤ・セキュリティ」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 ULData.Initialize、db_fini または ULData.Close 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 番号を指定します。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 構造体については、『Mobile Link クライアント』> 「Ultra Light 同期クライアントのネットワーク・プロトコルのオプション」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」156 ページを参照してください。

- Mobile Link HotSync 同期コンジットの配備。

  「Mobile Link 同期コンジットの配備」156 ページ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」157 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 Computing Platform 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) を見つけます。
3. インストール ツールを閉じます。
4. HotSync を実行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、『Mobile Link クライアント』> 「Mobile Link HotSync コンジットの配備」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、データベースは、アプリケーションから初めてアクセスされた時点で作成されます。その後、ユーザは最初の同期でデータの初期コピーをダウンロードする必要があります。**ULUtil** 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。

詳細については、『Ultra Light データベース・ユーザーズ・ガイド』> 「ULUtil ユーティリティ」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については、『Mobile Link クライアント』> 「Mobile Link HotSync コンジットの設定」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、UL_STORE_PARMS にストア・パラメータ **palm_allow_backup** を設定します。

詳細については、「UL_STORE_PARMS マクロ」270 ページを参照してください。

第 7 章

Windows CE 用 Ultra Light アプリケーションの開発

この章の内容

この章では、Windows CE 固有の開発、配備、同期の詳細について説明します。これらの指示は、一般的な開発プロセスについての知識があることを前提としています。これらの指示に基づいて、Ultra Light ソフトウェアに含まれる CustDB サンプル・アプリケーションを各プラットフォーム上で構築します。

概要

この項では、Microsoft Windows CE で使用する Ultra Light アプリケーションの構築に関して説明します。

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、『SQL Anywhere Studio の紹介』> 「Ultra Light 開発プラットフォーム」と『SQL Anywhere Studio の紹介』> 「Ultra Light ターゲット・プラットフォーム」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

Windows CE 用の開発環境として、コード記述に適しているのは、Microsoft eMbedded Visual C++ 3.0 です。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://www.microsoft.com/downloads/search.aspx?displaylang=ja からダウンロードできます。

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、SQL Anywhere ディレクトリの下の *Samples¥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¥UltraLite¥CEStarter¥ul_wceapplication.vcw* です。

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

- コンパイラの設定
  - インクルード・パスに *$(ASANY9)¥h* を追加します。
  - 適切なコンパイラ指令を定義します。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

- "*$(ASANY9)¥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。

    ```
    "$(ASANY9)¥win32¥sqlpp" -q -c
    "dsn=UltraLite 9.0 Sample" $(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。

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 Adaptive Server Anywhere インストール・ディレクトリの Ultra Light の *samples* 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。Windows CE 用のファイルは *CustDB* の *ce* 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」xix ページ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。
   - *Samples¥UltraLite¥CustDB¥EVC¥EVCCustDB.vcp* (eMbedded Visual C++ 3.0)
   - *Samples¥UltraLite¥CustDB¥EVC40¥EVCCustDB.vcp* (eMbedded Visual C++ 4.0)
3. [ビルド] － [Set Active Platform] 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] － [Set Active Configuration] 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] － [Settings] を選択します。[Project Settings] ダイアログが表示されます。
- [Link] タブの [Object/library modules] ボックスで、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 同期サーバを起動します。

- Mobile Link 同期サーバを起動するには、[プログラム] － [SQL Anywhere 9] － [Mobile Link] － [同期サーバのサンプル] を選択します。

8 CustDB アプリケーションを実行します。

  [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、MS VC++ 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.c* 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。規模の大きい Ultra Light

アプリケーションでは、*sqlpp* を数回にわたって呼び出し、その後に *ulgen* コマンドを 1 回実行して、カスタマイズされたリモート・データベースを作成する必要があります。

詳細については、「Embedded SQL アプリケーションの構築」118 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** 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。

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

例

次のサンプル Embedded SQL コードは **file_name** パラメータを設定します。

```
#undef UL_STORE_PARMS
 #define UL_STORE_PARMS UL_TEXT(
     "file_name=¥¥uldb¥¥my own
name.udb;cache_size=128k" )
 ...
 db_init( &sqlca );
```

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 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。

詳細については、『Mobile Link クライアント』> 「ActiveSync を使用するアプリケーションの配備」、『Mobile Link クライアント』> 「ActiveSync 用 Mobile Link プロバイダのインストール」、『Mobile Link クライアント』> 「ActiveSync で使用するアプリケーションの登録」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。

詳細については、「ActiveSync 同期の追加 (MFC の場合)」173 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」171 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」175 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」175 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。

アプリケーションへの同期の追加については、「データの同期」61 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、『Mobile Link クライアント』> 「同期パラメータ」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 Windows CE ハンドヘルド・デバイス用の同期ソフトウェアです。Ultra Light は ActiveSync バージョン 3.1 と 3.5 をサポートしています。

この項では、ActiveSync 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・ユーザのマシン上の 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 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」167 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。

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。

詳細については、「ULIsSynchronizeMessage 関数」466 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」168 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については、「ULIsSynchronizeMessage 関数」466 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については、「ULIsSynchronizeMessage 関数」466 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、通常、アプリケーションにはメニュー項目かユーザ・インタフェースが用意されます。

詳細については、「データの同期」61 ページを参照してください。

第 3 部　チュートリアル

第 3 部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++ コンポーネントを使用したアプリケーションの構築

この章の内容

この章は、簡単な Ultra Light C++ コンポーネント・アプリケーションを開発するプロセスについて理解していただくためのチュートリアルです。

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では、テキスト・エディタを使用して C++ ファイルを編集します。Microsoft Visual Studio などの C++ 開発環境を使用することもできます。

所要時間

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。自分でコードを入力する場合、これより長い時間が必要です。

能力と経験

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
- Ultra Light スキーマ・ペインタを使用して、Ultra Light スキーマを作成する方法を知っている

  詳細については、『Ultra Light データベース・ユーザーズ・ガイド』> 「レッスン 1：Ultra Light データベース・スキーマの作成」を参照してください。

注意
このチュートリアルの同期に関する操作では、SQL Anywhere Studio が必要です。

目的

このチュートリアルの目的は、Ultra Light C++ コンポーネント・アプリケーションの開発プロセスについて、知識と経験を得ることです。

レッスン 1 ：データベースへの接続

最初に、データベース・スキーマを作成します。次に、作成済みのスキーマを使用して、データベースを作成する C++ アプリケーションを記述、コンパイル、実行します。

❖ データベース・スキーマを作成するには、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。

このチュートリアルの後半では、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、チュートリアルを通じて、*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 Ultra Light スキーマ・ペインタを使用して、以下の特性を持つデータベース・スキーマを新しいディレクトリに作成します。

Ultra Light スキーマ・ペインタの使用方法の詳細については、『Ultra Light データベース・ユーザーズ・ガイド』> 「レッスン 1：Ultra Light データベース・スキーマの作成」を参照してください。

スキーマ・ファイルの名前 : **tutcustomer.usm**

テーブル名 : **customer**

カスタマのカラム :

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	char (15)	いいえ	なし
lname	char (20)	いいえ	なし

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
city	char (20)	はい	なし
phone	char (12)	はい	555-1234

プライマリ・キー : 降順 **id**

❖ 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 Microsoft Visual C++ で、[ファイル] ー [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・ディレクトリに *customer.cpp* という名前でファイルを保存します。

4 Ultra Light ライブラリをインポートし、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 "uliface.h"
 #include <stdio.h>
 #include <tchar.h>
 #include <assert.h>
 using namespace UltraLite;
 #define MAX_NAME_LEN 100
 ULSqlca Sqlca;
```

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。この例では、パラメータはデータベース・ファイルとスキーマ・ファイルのロケーションです。

次のコードでは、これらのロケーションはハード・コードされています。実際のアプリケーションでは、実行時にロケーションを指定します。また、これらの接続パラメータによって満たされるのは開発環境の接続だけです。Windows CE デバイスでアプリケーションを実行するには、追加のパラメータが必要です。

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=tutcustomer.udb" )
   UL_TEXT( ";schema_file=tutcustomer.usm" );
```

接続パラメータの詳細については、「クラス UltraLite_Connection_iface」286 ページ を参照してください。

6　エラー処理の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。

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_error_kind    kind,
     ul_void *    user_data,
     ul_char *    message_parameter )
 {
     ul_error_action rc;

     (void) user_data;
     switch( kind ) {
     case UL_ERROR_KIND_MEDIA_REMOVED:
         // Not handled in this sample: Prompt user
to re-insert media
         // (message_parameter contains the
filename).
         rc = UL_ERROR_ACTION_ABORT;
       // use UL_ERROR_ACTION_RETRY to retry
         break;

     case UL_ERROR_KIND_SQLCODE:
         switch( sqlca->sqlcode ){
```

```
        // The following errors are used for flow
control, and we don't want to
        // report them here:
        case SQLE_NOTFOUND:
        case SQLE_ULTRALITE_DATABASE_NOT_FOUND:
        break;
        case SQLE_CANNOT_ACCESS_SCHEMA_FILE:
        _tprintf( _TEXT(

    "Error %ld: UltraLite schema file %s not
found¥n" ), sqlca->sqlcode, message_parameter );
        break;
        case SQLE_COMMUNICATIONS_ERROR:
        _tprintf( _TEXT(

    "Error %ld: Communications error¥n" ), sqlca-
>sqlcode );
        break;
        default:
        _tprintf( _TEXT(

    "Error %ld: %s¥n" ), sqlca->sqlcode,
message_parameter );
        break;
        }
        rc = UL_ERROR_ACTION_ABORT; // N/A for
SQLCODE kind, return ABORT
        break;

    default:
        // future error kinds
        assert( 0 );
        rc = UL_ERROR_ACTION_ABORT; // default
action
        break;
    }
    return rc;
 }
```

Ultra Light では、2 つのエラー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。1 つは SQLE_ULTRALITE_DATABASE_NOT_FOUND エラーです。このエラーは最初の接続時に通知され (スキーマ・ファイルが存在する場合のみ)、スキーマ・ファイルからデータベー

スを作成することを、アプリケーションに対して要求するときに使用されます。もう 1 つの SQLE_NOTFOUND エラーは、結果セットのループの終了をマークします。

エラー処理の詳細については、「エラー処理」41 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 ファイルが存在しない場合は、SQLException がスローされます。スキーマ・ファイルを使用して新しいデータベースが作成され、そのデータベースへの接続が確立されます。

データベース・ファイルが存在する場合は、接続が確立されます。

```
Connection * open_conn( DatabaseManager * dm ) {
   Connection * conn;
   conn = dm->OpenConnection( Sqlca, ConnectionParms
);
   if( conn == NULL ) {
    if( Sqlca.GetSQLCode() ==
SQLE_ULTRALITE_DATABASE_NOT_FOUND ) {
     // The database doesn't exist yet -- create it
using the schema file.
      conn = dm->CreateAndOpenDatabase( Sqlca,
ConnectionParms );
      if( conn == NULL ) {
       handle_error( _TEXT("create database") );
      } else {
          _tprintf( _TEXT("Connected to a new
database.¥n") );
      }
    } else {
      handle_error( _TEXT("open database") );
    }
   } else {
    _tprintf( _TEXT("Connected to an existing
database.¥n") );
   }
   return conn;
 }
```

8　main() メソッドを実装します。

この main メソッドでは、次のタスクを実行します。

- 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[ 100 ];
   DatabaseManager * dm;
   Connection * conn;
   Sqlca.Initialize();
   ULRegisterErrorCallback(
     Sqlca.GetCA(), MyErrorCallBack,
     UL_NULL, buffer, 100 );
   dm = ULInitDatabaseManager( Sqlca );
   if( dm == UL_NULL ){
     //  You  may  have  mismatched  UNICODE  vs.
ANSI  runtimes.
     Sqlca.Finalize();
     return 1;
   }
   conn = open_conn( dm );
   if( conn == UL_NULL ){
     dm->Shutdown( Sqlca );
     Sqlca.Finalize();
     return 1;
   }
   dm->Shutdown( Sqlca );
   Sqlca.Finalize();
   return 0;
 }
```

9 Customer クラスをコンパイルして、リンクします。

クラスのコンパイルに使用するメソッドは、コンパイラによって異なります。以下の手順は、makefile を使用して、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・プロンプトから、チュートリアル・ディレクトリを参照します。
- *makefile* という名前の makefile を作成します。
- makefile で、次のようにディレクトリをインクルード・パスに追加します。

```
IncludeFolders= ¥
 /I"$(ASANY9)¥h"
```

- makefile で、次のようにディレクトリをライブラリ・パスに追加します。

```
LibraryFolders= ¥
 /LIBPATH:"$(ASANY9)¥ultralite¥win32¥386¥lib"
```

- makefile で、次のようにライブラリをリンカ・コマンド・ラインの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 (*ulimp.lib*) は、ASCII コード版のライブラリです。ユニコード版 (*ulimpw.lib*) を選択した場合は、コンパイラ・オプションに /DUNICODE を追加してください。

- この makefile で、以下のコンパイラ・オプションを 1 行にすべて設定します。

```
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
```

- ファイルの前処理バージョンを作成する命令を追加します。この手順はデバッグ用です。

```
customer.i: customer.cpp
    cl $(CompileOptions) $(IncludeFolders)
customer.cpp -P
```

- makefile を次のように実行します。

```
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。

❖ データベースにローを追加するには、次の手順に従います。

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。

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。
- テーブルの必要なカラムの識別子を取得します。テーブルのその他のカラムは NULL 値を受け入れるか、デフォルト値を持つことができます。
- テーブルが空の場合は、ローを 2 つ追加します。各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ます。
- 操作が完了したことを示すブール値が返されます。

```
bool do_insert( Connection * conn ) {
   Table * table = conn->OpenTable(
UL_TEXT("customer") );
   if( table == NULL ) {
   return  false;
```

```
    }
    if( table->GetRowCount() == 0 ) {
    _tprintf( UL_TEXT("Inserting two rows.¥n") );
       table->InsertBegin();
      table->Set( UL_TEXT("fname"), UL_TEXT("Penny")
);
      table->Set( UL_TEXT("lname"), UL_TEXT("Stamp")
);
       table->Insert();
       CHECK_ERROR();
    table->InsertBegin();
       table->Set( UL_TEXT("fname"), UL_TEXT("Gene")
);
      table->Set( UL_TEXT("lname"), UL_TEXT("Poole")
);
       table->Insert();
       CHECK_ERROR();
    conn->Commit();
       CHECK_ERROR();
    } else {
      _tprintf( UL_TEXT("The table has %lu rows¥n"),
    table->GetRowCount() );
    }
    table->Release();
    return true;
  }
```

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

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。

❖ テーブルのローをリストするには、次の手順に従います。

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 (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、id と名前が書き出されます。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(
UL_TEXT("customer") );
   if( table == NULL ) {
   return false;
   }
   TableSchema * schema = table->GetSchema();
   if( schema == NULL ) {
     table->Release();
     return false;
```

```
    }
    ul_column_num id_cid = schema->GetColumnID(
UL_TEXT("id") );
    ul_column_num fname_cid = schema->GetColumnID(
UL_TEXT("fname") );
    ul_column_num lname_cid = schema->GetColumnID(
UL_TEXT("lname") );
    schema->Release();
    while( table->Next() ) {
      ul_char fname[ MAX_NAME_LEN ];
      ul_char lname[ MAX_NAME_LEN ];
      table->Get( fname_cid ).GetString( fname,
MAX_NAME_LEN );
      table->Get( lname_cid ).GetString( lname,
MAX_NAME_LEN );
      _tprintf( "id=%d, name=%s %s¥n", (int)table-
>Get( id_cid ), fname, lname );
    }
    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を追加し、Mobile Link 同期サーバを起動し、アプリケーションを実行して同期します。

注意
このレッスンでは、SQL Anywhere Studio の一部である Mobile Link 同期を使用します。このレッスンを実行するには、SQL Anywhere Studio をコンピュータにインストールしておきます。

前のレッスンで作成した Ultra Light データベースは、Ultra Light 9.0 サンプル・データベースと同期します。Ultra Light 9.0 サンプル・データベースの ULCustomer テーブルのカラムは、作成した Ultra Light データベースの customer テーブルのカラムに対応します。

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。

❖ アプリケーションに同期を追加するには、次の手順に従います。

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。

   - 同期ストリームを TCP/IP に設定します。同期は、HTTP、ActiveSync、または HTTPS を使用しても実行できます。詳細については、『Mobile Link クライアント』> 「Ultra Light クライアント」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 同期サーバが同期スクリプトを自動的に生成できるように、sendColumnNames を true に設定します。

  詳細については、『Mobile Link 管理ガイド』> 「スクリプトの自動生成」を参照してください。

- Mobile Link ユーザ名を設定します。この値は、Mobile Link 同期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;
   conn->InitSynchInfo( &info );
   info.stream = ULSocketStream();
   info.version = UL_TEXT( "ul_default" );
   info.user_name = UL_TEXT( "sample" );
   info.send_column_names = true;
   info.download_only = true;
   if( !conn->Synchronize( &info ) ) {
   handle_error( _TEXT("synchronize") );
   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。

❖ データを同期するには、次の手順に従います。

1 Mobile Link 同期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。

```
dbmlsrv9 -c "dsn=UltraLite 9.0 Sample" -v+ -zu+ -za
```

`-zu+` と `-za` コマンド・ライン・オプションによって、ユーザの追加と同期スクリプトの生成が自動的に行われます。

これらのオプションの詳細については、『Mobile Link 管理ガイド』> 「Mobile Link 同期サーバのオプション」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 同期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。

レッスン 5 ：Windows CE デバイスへの配備

次の手順は、Ultra Light C++ コンポーネント・アプリケーションを Windows CE デバイスに配備する方法を示しています。

❖ Windows CE デバイスに配備するには、次の手順に従います。

1 デバイスとコンピュータが接続されていることを確認します。

2 デバイス上でファイル・エクスプローラを起動します。

[スタート] ー [プログラム] ー [ファイル エクスプローラ] を選択します。

3 Ultra Light ランタイムとアプリケーションを保存するディレクトリを作成します。

- デバイスのルートに移動します。ルートは [マイ デバイス] または [マイ ポケット PC] です。
- *UltraLite* という名前のディレクトリを作成します。
- UltraLite ディレクトリを開き、*lib* と *CustDB* という名前のサブディレクトリを作成します。

  ¥UltraLite¥lib は Ultra Light ランタイム・ファイルのロケーションで、*¥UltraLite¥CustDB* はアプリケーションのロケーションです。

4 Ultra Light ランタイム・ファイルを Windows CE デバイスにコピーします。

以上で、アプリケーションが Windows CE デバイスで実行できるようになりました。これでチュートリアルは完了です。

第 9 章

チュートリアル：静的型 C++ API を使用したアプリケーションの構築

この章の内容

この章は、静的型 C++ API を使用して Ultra Light アプリケーション開発のプロセスについて理解していただくためのチュートリアルです。ここでは簡単なアプリケーションの構築方法、およびアプリケーションに同期機能を追加する方法について説明します。

Ultra Light 静的型 C++ API の概要

Ultra Light 静的型 C++ API を使用すると、Embedded SQL の代わりに API を使用して Ultra Light C/C++ プログラムを開発できます。この API は Embedded SQL と同じ機能を提供しますが、使用形式は C++ インタフェースに類似しています。

基本クラス

Ultra Light C++ API は、Ultra Light アプリケーションの基本コンポーネントを示す一連の基本クラスから処理を開始します。基本テーブルは次のとおりです。

- **ULData**　Ultra Light データベースを表します。
- **ULConnection**　Ultra Light データベースへの接続を表し、同期を扱います。
- **ULCursor**　生成されたテーブルや結果セットのオブジェクトによって使用されるメソッドを提供します。これにより、データへのアクセスやデータの変更ができるようになります。
- **ULTable**　生成した結果セットのオブジェクトではなく、生成したテーブル・オブジェクトを使用するメソッドを表します。このクラスは **ULCursor** を継承します。
- **ULResultSet**　生成したテーブル・オブジェクトではなく、生成した結果セットのオブジェクトを使用するメソッドを表します。このクラスは **ULCursor** を継承します。**ULCursor** にある唯一のメソッドを含むため、このクラスを独立して記述することはありません。
- **ULStatement**　INSERT 文や UPDATE 文のように、結果セットを返さない文を表します。このクラスのすべてのメソッドが生成されます。

生成されるクラス

各アプリケーションでは、Ultra Light ジェネレータにより、使用中の Ultra Light データベースについて記述する一連のクラスが書き出されます。

- **生成される結果セット・クラス**　結果セットを返す SQL 文はそれぞれ、結果セットを参照し基本となるデータを修正するメソッドとともにクラスによって表されます。

- **生成されるテーブル・クラス**　アプリケーションの各テーブルは、クラスによって表されます。そのテーブルのメソッドを使用して、テーブルのローを修正できます。

  たとえば、Ultra Light ジェネレータは、Employee という名前のテーブルに対して **Employee** という名前のクラスも生成します。

- **生成される文のクラス**　結果セットを返さない SQL 文はそれぞれ、**Execute** メソッドを持つ簡単なクラスによって表されます。

データへのアクセス、データ修正、および統合データベースとの同期を行うには、アプリケーションでこれらのクラスを使用します。

概要

このチュートリアルでは、Ultra Light 静的型 C++ API を使用した、簡単なアプリケーションの構築方法について説明します。このアプリケーションは Microsoft Visual C++ を使って開発した Windows コンソール・アプリケーションであり、Ultra Light 9.0 サンプル・データベースの ULProduct テーブルに格納されているデータに対してクエリを実行します。

このチュートリアルでは、他の開発プラットフォームに慣れているユーザにもわかりやすいように、Visual C++ の設定を行うときに必要な手順について説明します。これらの手順を応用して、アプリケーション開発を始めることができます。

チュートリアルでは、次のタスクを実行するアプリケーションを記述し、構築します。

1. 単一テーブルからなる Ultra Light データベースに接続します。このテーブルは、Ultra Light サンプル・データベースの ULProduct テーブルのサブセットです。

2. テーブルにローを挿入します。初期データは通常、統合データベースと同期することで Ultra Light アプリケーションに追加されます。同期については、この章の後の方で追加します。

3. テーブルの最初のローを標準出力に書き出します。

アプリケーションを構築するには、次の手順を実行します。

1. Adaptive Server Anywhere リファレンス・データベース内の Ultra Light データベースを設計します。

   ここで、Ultra Light サンプル・データベース (CustDB) から単一のテーブルを使用します。

2. Ultra Light ジェネレータを実行し、Ultra Light データベース用の API を生成します。

   ジェネレータは C++ ファイルおよびヘッダ・ファイルを書き出します。

3. アプリケーションの論理を実装するソース・コードを記述します。

   ここでは、ソース・コードを *main.cpp* とします。

4. アプリケーションをコンパイルしてリンクし、実行します。

それから、アプリケーションに同期を追加します。

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。また、Ultra Light サンプル・データベースのコピーを作成して、実際の作業はそのコピーで行い、オリジナルのサンプル・データベースは他のプロジェクトでも使用できるように保管しておいてください。

このチュートリアルで使用するファイルのコピーは、SQL Anywhere ディレクトリのサブディレクトリ *Samples¥UltraLite¥APITutorial* に保管されています。

❖ チュートリアル・ディレクトリを準備するには、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作成します。チュートリアル後半では、このディレクトリが *c:¥APITutorial* であることを前提に説明します。

❖ サンプル・データベースのコピーを作成するには、次の手順に従います。

- Ultra Light 9.0 サンプル・データベースのバックアップ・コピーを作成し、チュートリアル・ディレクトリに保存します。Ultra Light 9.0 サンプル・データベースは *custdb.db* ファイルです。このファイルは、SQL Anywhere インストール・ディレクトリのサブディレクトリ *Samples¥UltraLite¥CustDB* にあります。このチュートリアルでは、オリジナルの Ultra Light 9.0 サンプル・データベースを使用します。チュートリアル終了後には、元のバージョンのサンプル・データベースを *APITutorial* ディレクトリから元の位置へコピーしなおします。

レッスン 2 ：Ultra Light データベース・テンプレートの作成

このチュートリアルでは、Ultra Light 9.0 サンプル・データベース (CustDB) のオリジナル・コピーを、リファレンス・データベースとして使用します。*APITutorial* ディレクトリに置くコピーはバックアップ・コピーになります。

「**Ultra Light データベース・テンプレート**」はテーブル・セットであり、テーブル内のカラムでもあります。これは Ultra Light データベースに含まれています。リファレンス・データベースにパブリケーションを作成して、Ultra Light データベース・テンプレートを作成します。パブリケーションは、テーブルおよびカラムベースのテーブルのサブセットをアセンブルする便利なデバイスです。SQL Remote または Mobile Link 同期に直接接続されることはありません。

また、SQL 文をリファレンス・データベースに追加して、Ultra Light データベースを定義できます。SQL 文によって、使用中の Ultra Light アプリケーションにおけるジョインおよび、より高度な機能を含めることができます。手順を簡易化するため、テーブルを定義して Ultra Light データベース・テンプレートを作成しましょう。

チュートリアルでは、SQL 文を使用して Ultra Light データベースを定義します。

データベースへの SQL 文の追加方法の例については、「レッスン 1 ：はじめに」201 ページを参照してください。

❖ Ultra Light データベース・テンプレートを作成するには、次の手順に従います。

1. Sybase Central を起動します。
2. Ultra Light 9.0 Sample データベースに接続します。
   - [ツール] － [接続] を選択します。
   - 表示されたプラグインのリストから [Adaptive Server Anywhere 9] を選択します。

- [接続] ダイアログで、ODBC データ・ソースとして [Ultra Light 9.0 Sample] を選択します。
- [OK] をクリックして接続します。

3 Ultra Light データベース内に格納するデータを説明したパブリケーションを作成します。

- Sybase Central で、custdb データベースを開きます。
- [パブリケーション] フォルダを開きます。[ファイル] － [新規] － [パブリケーション] を選択します。

  [パブリケーション作成] ウィザードが表示されます。
- パブリケーションに「ProductPub」という名前を付けます。
- ULProduct テーブルをパブリケーションに追加します。パブリケーションにはすべてのカラムを含めます。
- パブリケーションを作成するには [完了] をクリックしてください。

これで、Ultra Light データベース・テンプレートの設計が終了しました。Sybase Central とデータベース・サーバーを実行させたまま、次のレッスンに進んでください。

レッスン 3 ：Ultra Light ジェネレータの実行

Ultra Light ジェネレータを使って、Ultra Light データベース・テンプレートに指定されているとおりに、Ultra Light データベースへのインタフェースを定義する C++ ファイルおよびヘッダ・ファイルを書き出します。

❖ Ultra Light インタフェース・コードを生成するには、次の手順に従います。

1 コマンド・プロンプトから、*APITutorial* ディレクトリに移動します。

2 Ultra Light ジェネレータを、次の引数で実行します (すべて 1 行に入力)。

```
ulgen -c "dsn=UltraLite 9.0 Sample" -t c++ -
u ProductPub -f ProductPubAPI
```

ジェネレータにより次のファイルが書き出されます。

- **ProductPubAPI.hpp** 生成された API のプロトタイプ情報を保有するファイルです。*このファイルを調べて、アプリケーションで使用する API を決定します。*
- **ProductPubAPI.cpp** インタフェースのソース情報を保有するファイルです。このファイルは見る必要はありません。
- **ProductPubAPI.h** Ultra Light が必要とする内部定義を保有するファイルです。このファイルは見る必要はありません。

レッスン 4 ：アプリケーション・ソース・コードの書き込み

以下の手順にしたがって、アプリケーションのソース・コードを含む C++ ソース・ファイルを作成します。このコードには、エラー・チェックをはじめ、完全なアプリケーションに必要な機能は含まれていません。これは簡単なアプリケーションとして例示することを目的に提供されるアプリケーションです。

ソース・コードは *Samples¥UltraLite¥APITutorial¥sample.cpp* で検索できます。ただし、Insert 文のコメントを解除するためにファイルを編集する必要が生じる場合もあります。

❖ アプリケーション・ソース・コードを記述するには、次の手順に従います。

1. Microsoft Visual C++ で、[ファイル] ー [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。[OK] をクリックします。
3. 次のソース・コードを *APItutorial* ディレクトリにあるファイル *sample.cpp* にコピーして貼り付けます。

```
// (1) include headers
 #include <stdio.h>
 #include "ProductPubAPI.hpp"

 void main() {
     // (2) declare variables
     long price;
     ULData db;
     ULConnection conn;
     ULProduct productTable;

     // (3) connect to the UltraLite database
     db.Open() ;
         conn.Open( &db, "dba", "sql" );
     productTable.Open( &conn );
```

```
        // (4) insert sample data
        productTable.SetProd_id( 1 );
        productTable.SetPrice( 400 );
        productTable.SetProd_name( "4x8 Drywall x100"
);
        productTable.Insert();

        productTable.SetProd_id( 2 );
        productTable.SetPrice( 3000 );
       productTable.SetProd_name( "8' 2x4 Studs x1000"
);
        productTable.Insert();

        // (5) Write out the price of the items
        productTable.BeforeFirst();
        while( productTable.Next() ) {
            productTable.GetPrice( price );
            printf("Price: %d¥n", price );
        }

        // (6) close the objects to finish
        productTable.Close();
        conn.Close();
        db.Close();
    }
```

説明

コード内で番号付けされたコメントは、このルーチンが実行する主なタスクを示します。

1. ヘッダを含めます。

   stdio.h のほかに、生成されたヘッダ・ファイル *ProductPubAPI.hpp* も組み込んで、Product テーブルを記述する生成クラスを追加する必要があります。Ultra Light ヘッダ・ファイル *ulapi.h* がこのファイルに組み込まれます。

2. 変数を宣言します。

   Ultra Light データベースは **ULData** クラスのインスタンスとして宣言されます。データベースへの接続は **ULConnection** クラスのインスタンスです。これらのクラスは *ulapi.h* から組み込まれます。

   このテーブルは、リファレンス・データベースのテーブル名から生成された名前であるクラス **ULProduct** のインスタンスとして宣言されます。

3. データベースに接続します。

   宣言された各オブジェクトを開き、データへのアクセスを確立します。データベースを開くには引数を取りません。接続を確立するにはユーザ ID、パスワード、およびデータベース名が必要です。テーブルを開くには接続名が必要です。

4. サンプル・データを挿入します。

   運用アプリケーションでは、同期によりデータをデータベースに挿入します。実務上は、開発の初期段階において、いくつかのサンプル・データを挿入し、最終段階で同期を行う手順が最適な方法であるといえます。

   ULProduct クラス内の各メソッドは、リファレンス・データベース内のテーブル・カラムを表すユニークな名前を持っています。

   このルーチンに同期を追加する方法については、「レッスン 6：アプリケーションへの同期の追加」211 ページで説明します。

5. 各項目の価格を書き出します。

   価格を取り出し、テーブルの各ローに書き出します。

6. オブジェクトを閉じます。

   プログラムで使用するオブジェクトを閉じると、関連するメモリが開放されます。

レッスン 5 ：アプリケーションの構築と実行

選択した開発ツールを使って、アプリケーションのコンパイルおよびリンクを行います。この項では、Visual C++ を使用してコンパイルおよびリンクを行う方法を説明します。サポートされる別の開発ツールを使用する場合は、その開発ツールに応じて手順を読み替えてください。

1. デスクトップから通常の方法で Microsoft Visual C++ を起動します。

2. Visual C++ を設定し、Ultra Light ヘッダ・ファイルおよびライブラリ・ファイルが格納されているディレクトリを検索します。

   [ツール] － [オプション] を選択し、[ディレクトリ] タブをクリックします。[表示するディレクトリ] ドロップダウン・メニューの [インクルード ファイル] を選択します。次のディレクトリを含めて、ヘッダ・ファイルにアクセスできるようにします。

   ```
   C:¥Program Files¥Sybase¥SQL Anywhere 9¥h
   ```

   同じタブで、[表示するディレクトリ] ドロップダウン・メニューの [ライブラリ ファイル] を選択します。次のディレクトリを含めて、Ultra Light ライブラリ・ファイルにアクセスできるようにします。

   ```
   C:¥Program Files¥Sybase¥SQL Anywhere
   9¥ultralite¥win32¥386¥lib
   ```

   [OK] をクリックして変更を送信します。

3. プロジェクト **APITutorial** を作成します (ファイルを格納したディレクトリと同じ名前にしてください)。

   - [ファイル] － [新規作成] を選択します。[新規] ダイアログが表示されます。
   - [プロジェクト] タブで、[Win32 Console Application] を選択します。
   - プロジェクト名 **APITutorial** を指定します。

- *APITutorial* ディレクトリをロケーションとして指定します。
- [新規にワークスペースを作成] を選択し、[OK] をクリックします。
- [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。
- [ワークスペース] ウィンドウの [FileView] タブをクリックします。tutorial ワークスペースは、APITutorial プロジェクトだけで構成されます。[APITutorial ファイル] をダブルクリックすると、[Source Files]、[Header Files]、[Resource Files] という 3 つのディレクトリが表示されます。

4. プロジェクトを設定します。

- [APITutorial 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。
- [設定の対象] ドロップダウン・メニューから [すべての構成] を選択します。
- [リンク] タブをクリックします。次のランタイム・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。

  ```
  ulimp.lib
  ```
- [C/C++] タブをクリックします。[カテゴリ] ドロップダウン・メニューから [一般] を選択します。次のコードを [プリプロセッサの定義] リストに追加します。

  ```
  __NT__,UL_USE_DLL
  ```

  ここで、NT の両側にアンダースコアを付けてください (__NT__)。
- [OK] をクリックして完了します。

5. プロジェクトに *sample.cpp* と *ProductPubAPI.cpp* を追加します。

- [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。*sample.cpp* ファイルを指定して [OK] をクリックします。[ソース ファイル] フォルダを開いて *sample.cpp* が格納されていることを確認します。
- 生成された *ProductPubAPI.cpp* ファイルをプロジェクトに追加する操作を繰り返します。

6. Ultra Light API の基本クラスを含むファイルをプロジェクトに追加します。

   - [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。SQL Anywhere ディレクトリの *src* サブディレクトリにある *ulapi.cpp* を参照します。[OK] をクリックします。

7. アプリケーションのコンパイルおよびリンクを行います。

   - [構築] － [APITutorial.exe を構築] を選択し、実行プログラムをコンパイルしてリンクします。設定内容に応じて、実行プログラム *APITutorial.exe* が *APITutorial* ディレクトリの Debug ディレクトリに作成されます。

8. アプリケーションを実行します。

   - [構築] － [APITutorial.exe を実行] を選択します。

   コマンド・プロンプト・ウィンドウが表示され、製品テーブルの製品価格がリストされます。

これで、簡単な Ultra Light アプリケーションを構築し、実行することができました。次に、アプリケーションに同期を追加してみましょう。

レッスン 6：アプリケーションへの同期の追加

Ultra Light アプリケーションは統合データベースとデータをやりとりします。このレッスンでは、前項で作成した簡単なアプリケーションに同期を追加します。さらに、出力を変更して、同期が行われたことを確認します。

同期を追加すると、実際にコードが単純化されます。*main.cpp* の初期バージョンには次のコードが含まれています。これらのコードにより、Ultra Light データベースにデータが挿入されます。

```
productTable.SetProd_id( 1 );
 productTable.SetPrice( 400 );
 productTable.SetProd_name( "4x8 Drywall x100" );
 productTable.Insert();

 productTable.SetProd_id( 2 );
 productTable.SetPrice( 3000 );
 productTable.SetProd_name( "8' 2x4 Studs x1000" );
 productTable.Insert();
```

このコードは、アプリケーションにデータの初期セットを入力するために含まれています。運用アプリケーションでは、一般に、ソース・コードからデータの初期コピーは行わずに同期を実行します。

❖ 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　コードに同期情報の構造体を追加します。

- `// (2) declare variables` の直後に、次の行を追加します。

  ```
  auto ul_synch_info synch_info;
  ```

  この構造体は、同期を制御するパラメータを保持します。

2　明示的な挿入を同期呼び出しに置き換えます。

- 上記にリストされた **productTable** メソッドを削除します。

- その代わりに、次の行を追加します。

```
conn.InitSynchInfo( &synch_info );
 synch_info.user_name = UL_TEXT( "50" );
 synch_info.version = UL_TEXT( "custdb 9.0" );
 synch_info.stream = ULSocketStream();
 synch_info.stream_parms =
     UL_TEXT("host=localhost");
 conn.Synchronize( &synch_info );
```

  値 `50` は、Mobile Link のユーザ名を示します。

  文字列 `custdb` で、同期にはスクリプトのデフォルト・バージョンを使用するように、Mobile Link に命令しています。

  `ULSocketStream()` は、アプリケーションに対して、TCP/IP を介して同期をとるように命令しています。`host=localhost` は、Mobile Link サーバのホスト名を指定しています。ここでは現在のマシンの名前を指します。

3 アプリケーションのコンパイルとリンクを行います。

- [構築] － [APITutorial.exe を構築] を選択し、実行プログラムをコンパイルしてリンクします。設定内容に応じて、実行プログラム *APITutorial.exe* が *APITutorial* ディレクトリの Debug ディレクトリに作成されます。

4 サンプル・データベースに対して実行する Mobile Link サーバを起動します。

APITutorial ディレクトリのコマンド・プロンプトで、次のコマンドを入力します。

```
start dbmlsrv9 -c "dsn=UltraLite 9.0 Sample"
```

5 アプリケーションを実行します。

[構築] メニューから [APITutorial.exe を実行] を選択します。

データを受信するため、アプリケーションを接続し、同期を行います。コマンド・プロンプト・ウィンドウに情報が書き出されます。この出力は、次のようになります。

```
The ULData object is open
 Price: 400
 Price: 3000
 Price: 40
 Price: 75
 Price: 100
 Price: 400
 Price: 3000
 Price: 75
 Price: 40
 Price: 100
```

サンプル・データベースのリストア

チュートリアルを終了したら、再び使用できるように、サンプル・データをリストアしてください。「レッスン 1 ：はじめに」201 ページでは、Ultra Light 9.0 サンプル・データベースのコピーを作成しました。変更した *custdb.db* をそのコピーに置き換えてください。

❖ サンプル・データベースをリストアするには、次の手順に従います。

1. チュートリアル用のディレクトリから SQL Anywhere ディレクトリのサブディレクトリ *UltraLite¥Samples¥CustDB* に、*custdb.db* ファイルをコピーします。
2. 同じディレクトリで、トランザクション・ログ・ファイル *custdb.log* を削除します。

これで、サンプル・データベースがオリジナルの状態にリストアされました。

第 10 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

この章の内容

この章は、eMbedded Visual C++ を使用して Embedded SQL Ultra Light アプリケーションを開発するプロセスについて理解していただくためのチュートリアルです。

Ultra Light データベースの開発プロセスと補足情報の概要については、「Embedded SQL アプリケーションの開発」9 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については、「Embedded SQL を使用したアプリケーションの開発」73 ページを参照してください。

Embedded SQL の詳細については、「Embedded SQL API リファレンス」445 ページを参照してください。

概要

このチュートリアルでは、Embedded SQL ソース・ファイルを作成し、そのソース・ファイルを使用して簡単な Ultra Light アプリケーションを構築します。この Ultra Light アプリケーションはリモート・デバイスで実行できます。

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作成します。

  チュートリアル後半では、このディレクトリが *c:¥tutorial¥* であることを前提に説明します。

レッスン 1: eMbedded Visual C++ の設定

次のプロシージャは、Ultra Light 開発用に eMbedded Visual C++ を設定します。追加のライブラリやインクルード・パスが必要になることがあります。

❖ 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++ 3.0 を起動します。

   [スタート] メニューから、[プログラム] － [Microsoft Visual Tools] － [eMbedded Visual C++ 3.0] 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

   - [表示するディレクトリ] ドロップダウン・メニューの [インクルード ファイル] を選択します。次のディレクトリをインクルードして、Embedded SQL ヘッダ・ファイルにアクセスできるようにします。

     ```
     C:¥Program Files¥Sybase¥SQL Anywhere 9¥h
     ```

     SQL Anywhere をデフォルト以外のディレクトリにインストールしている場合には、インストール・ディレクトリの *¥h* サブディレクトリに置き換えます。

- [表示するディレクトリ] ドロップダウン・メニューの [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 エミュレータの場合は、次のように指定します。

  ```
  C:¥Program
  Files¥Sybase¥SQL Anywhere 9¥UltraLite¥
  ce¥emulator30¥lib
  ```

d. [OK] をクリックします。

レッスン 2: Embedded SQL ソース・ファイルに記述

次のプロシージャは、Ultra Light CustDB サンプル・データベースとの接続を確立するサンプル・プログラムを作成し、クエリを実行します。

❖ Embedded SQL Ultra Light 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. Microsoft eMbedded Visual C++ を起動します。

   [スタート] － [プログラム] － [Microsoft eMbedded Visual Tools] － [eMbedded Visual C++] を選択します。

2. 新しいワークスペース **Ultra Light** を作成します。

   - [ファイル] － [新規作成] を選択します。
   - [ワークスペース] タブをクリックします。
   - [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。

   - [ファイル] － [新規作成] を選択します。
   - [プロジェクト] タブをクリックします。
   - WCE Pocket PC 2002 アプリケーションを選択します。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。

- [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。

- [ファイル] ー [新規] を選択します。
- [ファイル] タブをクリックします。
- [C++ ソース ファイル] を選択します。
- [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。
- ファイルの名前は *sample.sqc* とします。[OK] をクリックします。
- 次のソース・コードをファイルにコピーします。

```
#include <stdio.h>
 #include <wingdi.h>
 #include <winuser.h>
 #include <string.h>
 #include "uliface.h"
 EXEC SQL INCLUDE SQLCA;
 int WINAPI WinMain( HINSTANCE hInstance,
     HINSTANCE hPrevInstance,
     LPSTR lpCmdLine,
     int nShowCmd)
 {
     /* Declare fields */
     EXEC SQL BEGIN DECLARE SECTION;
         long pid=1;
         long cost;
         char pname[31];
     EXEC SQL END DECLARE SECTION;
     /* Before working with data*/
     db_init(&sqlca);
     /* Connect to database */
     EXEC SQL CONNECT "DBA" IDENTIFIED BY "SQL";
     /* 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');
        EXEC SQL COMMIT;
        /* Fetch row from database */
        EXEC SQL SELECT price, prod_name
                INTO :cost, :pname
                FROM ULProduct
                WHERE prod_id= :pid;
        /* Error handling.  If the row does not
exist,
            or if     an error occurs, -1 is
returned */
        if((SQLCODE==SQLE_NOTFOUND)||(SQLCODE<0))
{
            return(-1);
        }

/* Print query results */
        wchar_t query[100];
        wchar_t result[10];
        wchar_t wpname[31];
        mbstowcs(wpname, pname, 31);
        wcscpy(query, L"Product id: ");
        _ltow(pid, result, 10);
        wcscat(query, result);
        wcscat(query, L" Price: ");
        _ltow(cost, result, 10);
        wcscat(query, result);
        wcscat(query, L" Product name: ");
        wcscat(query, wpname);
        wcscpy(result, L"Result");
        MessageBox(NULL, query, result, MB_OK);
        /* Preparing to exit:
        rollback any outstanding changes and
disconnect */
        EXEC SQL DISCONNECT;
        db_fini(&sqlca);
        return(0);
 }
```

- 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。

- [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。

  [Project Settings] ダイアログが表示されます。

- [設定の対象] ドロップダウン・メニューから [すべての構成] を選択します。

- [カスタム ビルド] タブで、次の文を [コマンド] ボックスに入力します。文は、すべて 1 行に入力するようにしてください。

  次の文は、*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++ に変換します。

  ```
  "%asany9%¥win32¥sqlpp.exe" -q -o WINDOWS -c
  "dsn=Ultralite 9.0 Sample" $(InputPath)
  sample.cpp
  ```

  SQL プリプロセッサの詳細については、『ASA プログラミング・ガイド』> 「SQL プリプロセッサ」を参照してください。

- [出力] ボックスで *sample.cpp* を指定します。

- [OK] をクリックして変更を送信します。

6 Adaptive Server Anywhere パーソナル・データベース・サーバを起動します。

データベース・サーバを起動すると、SQL プリプロセッサと Ultra Light アナライザの両方がリファレンス・データベースにアクセスします。サンプル・アプリケーションは、CustDB サンプル・データベース *custdb.db* をリファレンス・データベースと統合データベースとして使用します。

次のように、*custdb.db* を含む *Samples¥UltraLite¥CusDB* ディレクトリからコマンド・ラインを使用してデータベース・サーバを起動します。

```
dbeng9 custdb.db
```

[スタート] － [プログラム] － [SQL Anywhere 9] － [Ultra Light] － [Ultra Light のパーソナル・サーバのサンプル] を選択してもデータベース・サーバを起動できます。

7 *sample.sqc* ファイルの前処理を実行します。

サンプル・アプリケーションは、1 つのソース・ファイルだけで構成されているため、プリプロセッサは Ultra Light アナライザを自動的に実行し、生成されたソース・ファイルにさらに C/C++ コードを追加します。

- [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* フォルダに保存されます。

8 プロジェクトに *sample.cpp* ファイルを追加します。

- [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。
- *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。

サンプル・プログラムの説明

サンプル・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。

以下のリストに、このサンプル・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。

   サンプル・プログラムでは、標準的な I/O を使用します。そのため、*stdio.h* ヘッダ・ファイルはすでに含まれています。

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

   次のコマンドを使用します。

```
EXEC SQL INCLUDE SQLCA;
```

   この定義は、最初の Embedded SQL 文として定義する必要があるため、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。次のように、宣言セクションを記述します。

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

   ホスト変数については、「ホスト変数の使用」81 ページを参照してください。

4. 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);
```

5. 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、ログイン・ユーザ ID とパスワードを指定します。次に示すように、ユーザ ID **DBA**、パスワード **SQL** で接続します。

```
EXEC SQL CONNECT "DBA" IDENTIFIED BY "SQL";
```

6. データベース・テーブルにデータを挿入します。

   アプリケーションを初めて起動するときには、データベース・テーブルにデータは含まれていません。データは、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、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。

   同期機能を使用せずに、このサンプル・コードは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。

   同期機能を使用する場合、アプリケーションがクエリの実行に失敗したときに、その原因が同期プロセスであるのか、プログラム・エラーであるのかを判断できません。失敗の原因を突き止めるのは困難な作業です。一方、同期機能を使用せず、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。

   プログラムに誤りがないかテストしたら 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。

   Ultra Light アプリケーションへの同期の追加については、「レッスン 4 ：アプリケーションへの同期の追加」229 ページを参照してください。

7. SQL クエリを実行します。

   サンプル・プログラムは、結果のローを 1 つ返す選択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` に保管されます。

8. エラー処理を実行します。

   サンプル・プログラムは、エラー・コード `sqlcode` を返す選択要求を実行します。エラーが発生すると、このコードは負になります。クエリ結果がない場合には、SQL_NOTFOUND が返されます。サンプル・プログラムは、-1 を返してこれらのエラーを処理します。

9. データベースを切断します。

   切断する前に、未処理のまま残っている変更をロールバックするか、またはコミットしてください。

   切断するには、次のように DISCONNECT 文を使用します。

```
EXEC SQL DISCONNECT;
```

10. ライブラリ関数 *db_fini* を呼び出して、SQL の作業を終了します。

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

レッスン 3: Embedded SQL Ultra Light サンプル・アプリケーションの構築

次のプロシージャは、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 サンプル・アプリケーションを作成します。

❖ Embedded SQL Ultra Light 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. Adaptive Server Anywhere パーソナル・データベース・サーバが実行中であることを確認します。
2. プロジェクトを設定します。
   - **esql** を右クリックし、[Settings] を選択します。

     [Project Settings] ダイアログが表示されます。
   - [Settings For] ドロップダウン・メニューの [All Configurations] を選択します。
   - [Link] タブをクリックして、以下のランタイム・ライブラリを [Object/Library Modules] ボックスに追加します。

     `ulimp.lib`
   - [C/C++] タブをクリックします。[Category] ドロップダウン・メニューから [Preprocessor] を選択します。次の行が [Preprocessor definitions] の定義に含まれていることを確認してください。

     `__NT__`
   - [OK] をクリックしてダイアログを閉じます。
3. 実行プログラムを構築します。
   - [ビルド] － [Build 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。

4 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。

  画面が表示され、product テーブルの最初のローがリストされます。

レッスン 4 ：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、手動でデータを ULProduct テーブルに挿入したコードを、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命令に置き換えます。同期によってテーブルにデータが入力されるので、引き続き選択クエリを実行することができます。

TCP/IP を介する同期

TCP/IP のソケット接続を使用して、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。`ULSocketStream()` ストリームを使用して、`ULSynchronize` を呼び出します。

CustDB 統合データベースと同期するには、従業員 ID 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、または 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、同期中の割り込みからのリカバリに使用されます。

ULSynchronize 関数の詳細については、「ULSynchronize 関数」478 ページを参照してください。

同期機能を使用してサンプル・アプリケーションを実行する

sample.sqc に変更を加えたら、*sample.sqc* の前処理を実行し、*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. Adaptive Server Anywhere のデータベース・サーバが実行中であることを確認します。
2. INSERT コマンドを削除し、次のコードを追加します。*your-pc* を、使用するコンピュータの名前で置き換えます。

```
auto ul_synch_info synch_info;
 ULInitSynchInfo( &synch_info );
 synch_info.user_name = UL_TEXT("50");
 synch_info.version = UL_TEXT("custdb 9.0");
 synch_info.stream = ULSocketStream();
 synch_info.send_column_names = ul_true;
 synch_info.stream_parms = UL_TEXT("host=your-
pc;port=2439");
 ULSynchronize( &sqlca, &synch_info );
```

3 *sample.sqc* の前処理を実行します。

[ビルド] － [Compile] 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。

4 実行プログラムをビルドします。

[ビルド] － [Build esql.exe] を選択して、サンプル実行プログラムをビルドします。

5 Mobile Link 同期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。

```
dbmlsrv9 -c "DSN=UltraLite 9.0 Sample" -o ulsync.mls
-vcr -x tcpip -za
```

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、サンプル・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。サンプル・アプリケーションの選択クエリが実行され、クエリ結果のローが画面に表示されます。

第 11 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

この章の内容

この章では、Ultra Light ODBC アプリケーションの作成について説明します。

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」479 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、ほかの C/C++ インタフェースと機能を共有しません。特に、「Ultra Light C/C++ 共通 API リファレンス」247 ページにリストされている関数は、Ultra Light ODBC からは使用できません。

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"$(ASANY9)¥h"

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

 LibraryFolders= ¥
 /LIBPATH:"$(ASANY9)¥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 ASANY9 に依存します。

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・スキーマ・ファイルの作成

データベース・スキーマには、テーブル定義が格納されています。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。これは、ほかの Ultra Light コンポーネント・チュートリアルで使用されているスキーマと同じです。

この項では、Ultra Light スキーマ・ペインタを使用してスキーマ・ファイルを作成できることを前提としています。

データベース・スキーマの作成の詳細については、『Ultra Light データベース・ユーザーズ・ガイド』> 「レッスン 1：Ultra Light データベース・スキーマの作成」を参照してください。

❖ データベース・スキーマの作成

- Ultra Light スキーマ・ペインタを使用して、データベース・スキーマを作成します。

  Ultra Light スキーマ・ペインタを起動するには、[スタート] － [プログラム] － [SQL Anywhere Studio 9] － [Ultra Light] － [Ultra Light Schema Painter] を選択します。

  以下のようにスキーマを作成します。

  - **スキーマのファイル名** *c:¥tutorial¥ulodbc¥customer.usm*
  - **テーブル名** customer
  - **customer 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	char (15)	いいえ	なし
lname	char (20)	いいえ	なし
city	char (20)	はい	なし

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
phone	char (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」480 ページと 「SQLFreeHandle 関数」494 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(

"schema_file=customer.usm;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。

Ultra Light データベース・ファイル *customer.udb* とテンポラリ・ファイル *customer.~db* が作成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、「SQLConnect 関数」484 ページと 「SQLDisconnect 関数」488 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」490 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。

挿入したローが出力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、「SQLPrepare 関数」501 ページ、「SQLExecute 関数」491 ページ、「SQLBindCol 関数」481 ページ、「SQLFetch 関数」492 ページを参照してください。

これでチュートリアルが完了しました。

第 4 部　API リファレンス

第 4 部では、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 12 章

Ultra Light C/C++ 共通 API リファレンス

この章の内容

この章では、Embedded SQL、静的型 C++ API、または C++ コンポーネント・インタフェースで使用する関数とマクロについて説明します。これらの関数とマクロは、ODBC からは使用できません。

この章で説明するすべての関数は、SQLCA (SQL Communication Area) が必要です。詳細については、「Ultra Light C/C++ インタフェースの共通機能」129 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。

プロトタイプ

ul_error_action UL_GENNED_FN_MOD *error-callback-function*(
SQLCA * *sqlca*,
ul_error_kind *kind*,
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。「ULRegisterErrorCallback 関数」259 ページを参照してください。

- **sqlca** SQLCA (SQL communication 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。

  静的型 C++ API の場合、SQLCA はヘッダ・ファイルの中で **sqlca** として宣言されます。C++ コンポーネントの場合は、Sqlca.GetCA() メソッドを使用します。

- **kind** Ultra Light が通知しているエラーの種類。以下の定数のいずれかを表します。

  - **UL_ERROR_KIND_MEDIA_REMOVED** 現在の I/O 操作を正しく行うためには、データベースが格納されているメディア・カードを取り除き、再び挿入します。

    この種類のエラーに対する一般的なアクションでは、メディア・カードの挿入をユーザに要求します。

この種類のエラーの場合、バッファにはデータベース・ファイル名が保持されています。

- **UL_ERROR_KIND_SQLCODE** SQL エラー。SQLCA には、SQL コードが sqlca->sqlcode の形式で含まれています。SQL コードとその意味は、『ASA エラー・メッセージ』> 「Adaptive Server Anywhere SQLCODE 順にリストしたエラー・メッセージ」にリストされています。この種類のエラーの場合、返されるエラー・アクションは無視され、操作は常にキャンセルされます。

- **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・パラメータについては、『ASA エラー・メッセージ』> 「データベース・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。この値は、UL_ERROR_KIND_SQLCODE に特に適しています。

- **UL_ERROR_ACTION_TRY_AGAIN** エラーを引き起こした操作を再び試みます。

説明

この方法によるエラー処理の詳細については、
「ULRegisterErrorCallback 関数」259 ページを参照してください。

参照

- 「ULRegisterErrorCallback 関数」259 ページ
- 『ASA エラー・メッセージ』> 「Adaptive Server Anywhere SQLCODE 順にリストしたエラー・メッセージ」

ULRegisterSchemaUpgradeObserver のコールバック関数

Ultra Light データベース・スキーマのアップグレード時に、アプリケーションによる進行状況を表示できます。

プロトタイプ

ul_ret_void UL_GENNED_FN_MOD *upgrade-callback-function*(
p_ul_schema_upgrade_status *status*
)

パラメータ

- **upgrade-callback-function** 関数の名前。ULRegisterSchemaUpgradeObserver に名前を指定します。
- **status** スキーマ・アップグレード構造体へのポインタ。次のように定義されています。

```
typedef struct {
   ul_schema_upgrade_state state;
   ul_u_long               progress_counter;
   ul_u_long               final_progress_count;
   ul_u_long               upgrade_operations;
   ul_bool                 stop;
   ul_void *               user_data;
 }
```

ステータス・フィールドは次のとおりです。

- **state** 以下のいずれかを表します。
  - **UL_UPGRADE_STATE_STARTING** アップグレードが開始します。この段階では、ユーザは問題なく操作をキャンセルできます。
  - **UL_UPGRADE_STATE_UPGRADING** アップグレードが進行中です。
  - **UL_UPGRADE_STATE_ABORT** 修復可能なエラーの結果、またはユーザ・アクションの結果として、スキーマのアップグレードがキャンセルされます。古いデータベースは保持されます。

- **UL_UPGRADE_STATE_ERROR** 致命的なエラーが発生しました。データベースは使用不可能な状態です。
- **UL_UPGRADE_STATE_DONE** アップグレードが正常に終了しました。

- **progress_counter** 現在までの進行状況を表す近似値。値は 0 ～ final_progress_count の間の数値で、完了したパーセンテージをダイアログ・ボックスに表示できます。
- **final_progress_count** アップグレードが正常に終了したときの progress_counter の値。
- **upgrade_operations** アップグレードが進行している中で終了した作業量の近似値。値は 0 から開始し、アップグレードの進行に伴い増加します。progress_counter より頻繁に更新されます。ほかのスキーマ・アップグレードと比較するための相対的な計測値として使用できます。
- **stop** UL_UPGRADE_STATE_STARTING の状態の場合、スキーマのアップグレードをキャンセルするには、この値に ul_true を設定します。アプリケーションは、UL_UPGRADE_STATUS_ABORT の状態で 2 番目のコールバックを受け取ります。

  Embedded SQL と静的型 C++ API の場合、接続操作は失敗し、SQLCODE は SQLE_SCHEMA_UPGRADE_NOT_ALLOWED です。C++ コンポーネントの場合、UpgradeSchemaFromFile 呼び出しは false を返し、SQLCODE には SQLE_SCHEMA_UPGRADE_NOT_ALLOWED が設定されます。

  UL_UPGRADE_STATE_STARTING の状態でない場合に stop に ul_true を設定しても、stop 値は無視されます。
- **user_data** これは、ULRegisterSchemaUpgradeObserver または UpgradeSchemaFromFile に渡される user_data パラメータと同じです。

ULEnableFileDB 関数

Palm Computing Platform 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。

プロトタイプ

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

パラメータ

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

静的型 C++ API の場合、SQLCA はヘッダ・ファイルの中で **sqlca** として宣言されます。C++ コンポーネントの場合は、Sqlca.GetCA() メソッドを使用します。

説明

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

参照

「ULEnablePalmRecordDB 関数」256 ページ

ULEnableGenericSchema 関数 (旧式)

新バージョンのアプリケーションを配備するときにデータベース・スキーマをアップグレードします。

旧式な機能
この文は使用されなくなりました。代わりに ULRegisterSchemaUpgradeObserver を使用します。

プロトタイプ

void **ULEnableGenericSchema(** SQLCA * *sqlca* **);**

パラメータ

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

静的型 C++ API の場合、SQLCA はヘッダ・ファイルの中で **sqlca** として宣言されます。C++ コンポーネントの場合は、Sqlca.GetCA() メソッドを使用します。

説明

新しい Ultra Light アプリケーションがデバイスに配備されると、Ultra Light はデフォルトで空のデータベースを再作成するため、新しいアプリケーションが配備される前にデータベース内にあったデータは失われます。ULEnableGenericSchema を呼び出すと、既存のデータベースが新しいアプリケーションのスキーマにアップグレードされます。

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、**db_init** または **ULData.Open()** の前に呼び出してください。Palm Computing Platform は例外で、この場合はアップグレードの前に必ずしもすべてのカーソルを閉じる必要はありません。Palm Computing Platform でのアップグレードの直後に、LAUNCH_SUCCESS_FIRST 起動コードが返されます。

アップグレードの前にバックアップ
アップグレードを試行する前に、データベース・ファイルをコピーするか同期をとって、データをバックアップすることを強くおすすめします。

スキーマのアップグレード手順の詳細については、『Ultra Light データベース・ユーザーズ・ガイド』> 「スキーマのアップグレード方法」を参照してください。

ULEnablePalmRecordDB 関数

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。

プロトタイプ

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。

静的型 C++ API の場合、SQLCA はヘッダ・ファイルの中で **sqlca** として宣言されます。C++ コンポーネントの場合は、Sqlca.GetCA() メソッドを使用します。

説明

この関数は、C++ コンポーネント、C++ API アプリケーション、Embedded SQL アプリケーションで使用できます。

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」253 ページ

ULEnableStrongEncryption 関数

Ultra Light データベースを高度に暗号化します。

プロトタイプ

void **ULEnableStrongEncryption(** 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。

静的型 C++ API の場合、SQLCA はヘッダ・ファイルの中で **sqlca** として宣言されます。C++ コンポーネントの場合は、Sqlca.GetCA() メソッドを使用します。

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。この関数は、**db_init()** または **ULData.Open()** の前に呼び出してください。

参照

「データの暗号化」59 ページ

ULEnableUserAuthentication 関数

Ultra Light アプリケーションでのユーザ認証を有効にします。

プロトタイプ

void **ULEnableUserAuthentication(** 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。

静的型 C++ API の場合、SQLCA はヘッダ・ファイルの中で **sqlca** として宣言されます。C++ コンポーネントの場合は、Sqlca.GetCA() メソッドを使用します。

説明

この関数を呼び出さなければ、Ultra Light データベースにアクセスするときにユーザ ID もパスワードも必要ありません。この関数を使用して有効なユーザ ID とパスワードを指定します。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** 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されます。

参照

『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light のユーザ認証」

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。

プロトタイプ

```
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。

  静的型 C++ API の場合、SQLCA はヘッダ・ファイルの中で **sqlca** として宣言されます。C++ コンポーネントの場合は、Sqlca.GetCA() メソッドを使用します。

- **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」248 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** コールバック関数はアプリケーション内の任意の位置から呼び出されるため、アクセスするコンテキスト情報は、グローバルにアクセスできる必要があります。グローバル変数の代わりに、このフィールドを使用して、関数がアクセスするコンテキスト情報を提供できます。

  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。

  ここでデータ型を宣言し、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。SQL エラーのエラー・パラメータについては、『ASA エラー・メッセージ』> 「データベース・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 文字単位のバッファの長さ。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、パラメータは問題なくトランケートされます。

説明

この関数が呼び出されると、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しないため、コールバック関数がほかのエラー処理コードに置き換わることはありません。

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,
            100 );
        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_error_kind    kind,
     ul_void *    user_data,
     ul_char *    message_parameter ){
     ul_error_action rc;
     (void) user_data;

    switch( kind ) {
     case UL_ERROR_KIND_MEDIA_REMOVED:
         // TODO: Prompt user to re-insert media
         // (message_parameter contains the filename).
         rc = UL_ERROR_ACTION_ABORT; // use
UL_ERROR_ACTION_RETRY to retry
         break;

    case UL_ERROR_KIND_SQLCODE:
         switch( sqlca->sqlcode ){
         // The following errors are used for flow
control, and we don't want to
         // report them here:
         case SQLE_NOTFOUND:
         case SQLE_ULTRALITE_DATABASE_NOT_FOUND:
             break;
         case SQLE_CANNOT_ACCESS_SCHEMA_FILE:
           _tprintf(
            _TEXT( "Error %ld: UltraLite schema file %s
not found¥n" ),
            sqlca->sqlcode,
            message_parameter );
            break;
         case SQLE_COMMUNICATIONS_ERROR:
           _tprintf(
            _TEXT( "Error %ld: Communications error¥n"
),
            sqlca->sqlcode );
            break;
         default:
            _tprintf(
```

```
                _TEXT( "Error %ld: %s¥n" ),
                sqlca->sqlcode,
                message_parameter );
                break;
            }

        rc = UL_ERROR_ACTION_ABORT; // ignored for SQL
kind
            break;
        default:
            // future error kinds
            assert( 0 );
            rc = UL_ERROR_ACTION_ABORT; // default action
            break;
        }
        return rc;
    }
```

参照

- 『ASA エラー・メッセージ』> 「Adaptive Server Anywhere SQLCODE 順にリストしたエラー・メッセージ」
- 「ULRegisterErrorCallback のコールバック関数」248 ページ

ULRegisterSchemaUpgradeObserver 関数

プロトタイプ

```
ul_ret_void ULRegisterSchemaUpgradeObserver (
    SQLCA * sqlca,
    ul_schema_upgrade_observer_fn  callback,
    ul_void * user_data
    );
```

パラメータ

- **sqlca** SQLCA へのポインタ。

  静的型 C++ API の場合、SQLCA はヘッダ・ファイルの中で **sqlca** として宣言されます。C++ コンポーネントの場合は、Sqlca.GetCA() メソッドを使用します。

- **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」248 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** コールバック関数はアプリケーション内の任意の位置から呼び出されるため、アクセスするコンテキスト情報は、グローバルにアクセスできる必要があります。グローバル変数の代わりに、このフィールドを使用して、関数がアクセスするコンテキスト情報を提供できます。

  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。

  ここでデータ型を宣言し、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;
  ```

説明

スキーマのアップグレード処理には、時間がかかることがあります。observer コールバックを使用してアップグレード処理をモニタし、エンド・ユーザにフィードバックすることができます。たとえば、この機能を使用すると、進行状況を示すパーセンテージをダイアログ・ボックスに表示してユーザに提供できます。

db_init (Embedded SQL) または ULData::Open (静的型 C++ API) の呼び出し後、しかしデータベースに接続する前に ULRegisterSchemaUpgradeObserver を呼び出します。アップグレードは接続操作の中で行われ、そのときに、登録されているコールバック関数に対してコールバックが行われます。

C++ コンポーネントから ULRegisterSchemaUpgradeObserver を呼び出せますが、Connection.UpgradeSchemaFromFile メソッドを使用しても同じ操作ができます。アプリケーションが両方を呼び出した場合は、ULRegisterSchemaUpgradeObserver が使用されます。

データベース・スキーマをアップグレードするアプリケーションの中では、ULRegisterSchemaUpgradeObserver を呼び出してください。ユーザにフィードバックを提供しない場合は、コールバック関数として UL_NULL を渡します。ULRegisterSchemaUpgradeObserver を使用するとアプリケーションのサイズが大幅に大きくなるため、アプリケーションがデータベース・スキーマをアップグレードしない場合は、ULRegisterSchemaUpgradeObserver の呼び出しを省略できます。

参照

『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light データベース・スキーマのアップグレード」

ULStoreDefragFini 関数

プロトタイプ

ul_ret_void **ULStoreDefragFini(**
SQLCA * *sqlca*,
p_ul_store_defrag_info *dfi* **);**

説明

ULStoreDefragInit が返すデフラグメンテーション情報ブロックを破棄する関数です。

パラメータ

sqlca　SQLCA へのポインタ。

dfi　デフラグメンテーション情報のブロック。

参照

「Ultra Light データベースのデフラグメンテーション」135 ページ

「ULStoreDefragInit 関数」266 ページ

ULStoreDefragInit 関数

プロトタイプ

p_ul_store_defrag_info **ULStoreDefragInit(** SQLCA * *sqlca* **);**

説明

この関数は、データベースのデフラグメンテーション状態を管理するためのデフラグメンテーション情報ブロックを初期化し返します。

パラメータ

sqlca SQLCA へのポインタ。

検査結果

処理が正常に完了した場合には、デフラグメンテーション情報のブロック **p_ul_store_defrag_info** を返します。メモリ不足などで処理が正常に完了しなかった場合には、**UL_NULL** が返されます。

参照

「Ultra Light データベースのデフラグメンテーション」135 ページ

「ULStoreDefragFini 関数」265 ページ

ULStoreDefragStep 関数

プロトタイプ

ul_bool **ULStoreDefragStep(**
SQLCA * *sqlca*
p_ul_store_defrag_info *dfi* **);**

説明

この関数は 1 つのデータベースをデフラグメントします。

パラメータ

sqlca　SQLCA へのポインタ。

dfi　デフラグメンテーション情報のブロック。

検査結果

記憶領域全体がデフラグメントされた場合には **ul_true** が返されます。

記憶領域全体がデフラグメントされない場合には、**ul_false** が返されます。

エラーが発生すると、SQLCODE が設定されます。

参照

「Ultra Light データベースのデフラグメンテーション」135 ページ

「ULStoreDefragFini 関数」265 ページ

「ULStoreDefragInit 関数」266 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指令

この項では、Ultra Light C/C++ アプリケーションに対して指定するコンパイラ指令について説明します。特に指定のないかぎり、指令は Embedded SQL と C++ API の両方のアプリケーションに適用されます。

コンパイラ指令は、コンパイラのコマンド・ラインまたはユーザ・インタフェースのコンパイラ設定ダイアログ・ボックスに指定できます。ソース・コードの中で定義することもできます。

コンパイラのコマンド・ラインの場合、一般には、/D コマンド・ライン・オプションを使用してコンパイラ指令を設定します。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
 /D__NT__ /DUL_USE_DLL /DULB_USE_BIGINT_TYPES
 /DULB_USE_FLOAT_TYPES /DUL_ENABLE_USER_AUTH

 IncludeFolders= ¥
 /I"$(VCDIR)¥include" ¥
 /I"$(ASANY9)¥h"

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

ここで、*VCDIR* は Visual C++ ディレクトリであり、*ASANY9* は SQL Anywhere ディレクトリです。

ソース・コードの場合は、`#define` 文を使用して指令を指定します。

UL_AS_SYNCHRONIZE マクロ

関数

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。

適用先

ActiveSync のみを使用する Windows CE アプリケーション。

参照　『Ultra Light C/C++ ユーザーズ・ガイド』>「アプリケーションへの ActiveSync 同期の追加」

『Ultra Light C/C++ ユーザーズ・ガイド』>「アプリケーションへの ActiveSync 同期の追加」

UL_ENABLE_OBFUSCATION マクロ

関数　デフォルトでは、難読化は無効です。難読化を有効にするには、生成されたデータベースをコンパイルするときに、UL_ENABLE_OBFUSCATION を定義します。

適用先　生成されたデータベース・コード。

参照　「データの暗号化」59 ページ

UL_ENABLE_USER_AUTH マクロ

関数　C++ API アプリケーションの場合のみ、この指令を定義してユーザ認証を有効にします。この指令を使用しないと、C++ API Ultra Light アプリケーションではユーザ認証は行われません。

適用先　*ulapi.cpp* ファイル。

参照　『Ultra Light C/C++ ユーザーズ・ガイド』>「ユーザの認証」

UL_ENABLE_SEGMENTS マクロ

関数　コンパイラに対して、Palm Computing Platform アプリケーションのマルチセグメント・コードの生成を指示します。

適用先　生成されたデータベース・コード。

参照　『Ultra Light C/C++ ユーザーズ・ガイド』>「マルチセグメント・コード生成の有効化」

『Ultra Light C/C++ ユーザーズ・ガイド』> 「マルチセグメント・コード生成の有効化」

UL_OMIT_COLUMN_INFO マクロ

関数

生成コードが必要とする Palm OS データ・セグメント数を減少します。

使用法

生成されたファイルをコンパイルする前に、プリプロセッサ・シンボル UL_OMIT_COLUMN_INFO を定義します。CodeWarrior では、プレフィクス・ファイルに `#define` を追加してこの処理を行います。

マクロは、生成コードが必要とするセグメント数を減少します。セグメント数を確実に減少するために、Ultra Light ジェネレータが別に書き込む拡張スキーマ情報は省かれます。この結果、このマクロが定義されると、データベース・スキーマをアップグレードできず、send_column_names 同期パラメータも使用できません。

UL_STORE_PARMS マクロ

関数

キーワードと値の組み合わせを指定して、データベース・ストレージを設定します。

構文

#define UL_STORE_PARMS UL_TEXT("*keyword=value;…***")**

キーワードと値のリスト内のすべてのスペースが有効です。ただし、文字列の先頭のスペースと、セミコロンの直後のスペースを除きます。

使用法

すべての **db_init()** 呼び出しから参照できるように、UL_STORE_PARMS マクロはアプリケーションのソース・コードのヘッダで定義されます。

パラメータ

キーワードは、大文字と小文字を区別しません。値における大文字と小文字の区別は、値を解釈するアプリケーションに依存します。たとえば、ファイル名における大文字と小文字の区別は、オペレーティング・システムに依存します。

使用可能なパラメータのリストについては、『Ultra Light データベース・ユーザーズ・ガイド』> 「データベース・スキーマ・パラメータ」および 『Ultra Light データベース・ユーザーズ・ガイド』> 「追加接続パラメータ」を参照してください。

例

次の文は、キャッシュ・サイズを 128 KB に設定します。

```
#undef  UL_STORE_PARMS
 #define UL_STORE_PARMS  UL_TEXT("cache_size=128k")
     . . .
 db_init( &sqlca );
```

次の例に示すように、UL_STORE_PARMS を文字列に設定し、その文字列の値をプログラムで設定してから db_init を呼び出すことができます。UL_TEXT マクロと _stprintf 関数を使用して、文字を正しくコード化します。

```
char store_parms[32];
 #undef  UL_STORE_PARMS
 #define UL_STORE_PARMS  store_parms
 ...
     /* Set cache_size to the correct number of bytes.
*/
 ...
 _stprintf( store_parms, UL_TEXT("cache_size=%lu"),
     cache_size );
 db_init( &sqlca );
```

次に示す類似の例では、暗号化キーとしてパスワードを使用しており、これが、Ultra Light データベース・ファイルのロケーションとファイル名を設定します。コードでは、変数 filename に目的のパスとファイル名があらかじめ設定されていることが前提です。コードを実行してから、db_init を呼び出します。

```
TCHAR store_parms[100];
 #undef  UL_STORE_PARMS
 #define UL_STORE_PARMS  store_parms

//put the password in UL_STORE_PARMS to be used as
encryption key
 my_sprintf( store_parms, UL_TEXT("key=%s"), app_pword
);

 #ifdef UNDER_CE
```

```
    if( filename != NULL ) {
        my_strcat( store_parms, _T(";file_name=") );
        my_strcat( store_parms, filename );
    }
    #endif
```

参照

『Ultra Light データベース・ユーザーズ・ガイド』> 「データベース・スキーマ・パラメータ」

『Ultra Light データベース・ユーザーズ・ガイド』> 「追加接続パラメータ」

『Ultra Light C/C++ ユーザーズ・ガイド』> 「Ultra Light データベース・スキーマ・ファイルの作成」

「データの暗号化」59 ページ

UL_SYNC_ALL マクロ

関数

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

「publication 同期パラメータ」524 ページ

『Ultra Light C/C++ ユーザーズ・ガイド』> 「ULGetLastDownloadTime 関数」

『Ultra Light C/C++ ユーザーズ・ガイド』> 「ULCountUploadRows 関数」

「UL_SYNC_ALL_PUBS マクロ」272 ページ

UL_SYNC_ALL_PUBS マクロ

関数

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

「publication 同期パラメータ」524 ページ

『Ultra Light C/C++ ユーザーズ・ガイド』> 「ULGetLastDownloadTime 関数」

『Ultra Light C/C++ ユーザーズ・ガイド』> 「ULCountUploadRows 関数」

「UL_SYNC_ALL マクロ」272 ページ

UL_TEXT マクロ

関数

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。Embedded SQL アプリケーションと C++ API アプリケーションでは、このマクロを使用して定数文字列をすべて囲み、コンパイラがこれらのパラメータを正しく処理できるようにします。

UL_USE_DLL マクロ

関数

Windows CE アプリケーションと Windows アプリケーションの場合のみ、この指令を定義して、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 を使用します。

適用先

生成されたデータベース・コード。

UNDER_CE マクロ

関数

Windows CE 用の Ultra Light アプリケーションをコンパイルする場合のみ、このマクロを使用します。

デフォルトでは、このマクロは、すべての新規 eMbedded Visual C++ プロジェクトの中で定義されます。

参照

『Ultra Light C/C++ ユーザーズ・ガイド』> 「Windows CE 用 Ultra Light アプリケーションの開発」.

UNDER_PALM_OS マクロ

関数

Palm OS 用の Ultra Light アプリケーションをコンパイルする場合のみ、このマクロを使用します。

このマクロは、Ultra Light Palm OS アプリケーションに含まれる *ulpalmXX.h* ヘッダ・ファイルの中で、Ultra Light プラグインによって定義されます。詳細については、『Ultra Light C/C++ ユーザーズ・ガイド』> 「Ultra Light plug-in for CodeWarrior の使用」を参照してください。

参照

『Ultra Light C/C++ ユーザーズ・ガイド』> 「Palm Computing Platform 用 Ultra Light アプリケーションの開発」.

第 13 章

Ultra Light C++ コンポーネント API リファレンス

この章の内容

この章では Ultra Light C++ コンポーネント API について説明します。

クラス ULSqlca

構文

public **ULSqlca**

基本クラス

- 「クラス ULSqlcaBase」278 ページ

説明

「クラス ULSqlcaBase」278 ページ サブクラスに SQLCA 構造体が含まれるため、外部構造体は必要ありません。

これは、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」278 ページ
- 「GetCA 関数」279 ページ
- 「GetParameter 関数」279 ページ
- 「GetParameterCount 関数」279 ページ
- 「GetSQLCode 関数」280 ページ
- 「GetSQLCount 関数」280 ページ
- 「GetSQLErrorOffset 関数」280 ページ
- 「Initialize 関数」280 ページ
- 「LastCodeOK 関数」281 ページ
- 「LastFetchOK 関数」281 ページ
- 「ULSqlca 関数」276 ページ
- 「~ULSqlca 関数」277 ページ

ULSqlca 関数

構文

ULSqlca::ULSqlca()

説明

コンストラクタです。

~ULSqlca 関数

構文　**ULSqlca::~ULSqlca()**

説明　デストラクタです。

クラス ULSqlcaBase

構文

public **ULSqlcaBase**

派生クラス

- 「クラス ULSqlca」276 ページ
- 「クラス ULSqlcaWrap」282 ページ

説明

SQLCA を提供します。

ULSqlcaBase は、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 「クラス ULSqlca」276 ページ) を使用します。基本となる SQLCA オブジェクトが必ず存在しなければなりません。SQLCA は、他の関数を呼び出す前に初期化する必要があります。それぞれのスレッドには、固有の SQLCA が必要です。

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」278 ページ
- 「GetCA 関数」279 ページ
- 「GetParameter 関数」279 ページ
- 「GetParameterCount 関数」279 ページ
- 「GetSQLCode 関数」280 ページ
- 「GetSQLCount 関数」280 ページ
- 「GetSQLErrorOffset 関数」280 ページ
- 「Initialize 関数」280 ページ
- 「LastCodeOK 関数」281 ページ
- 「LastFetchOK 関数」281 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

ローの sqlca 構造体

GetParameter 関数

構文

size_t **ULSqlcaBase::GetParameter(**
ul_u_long *parm_num*
ul_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** ul_char のバッファ・サイズ

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 (NULL ターミネータを含む ul_char の数)。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。

これは、挿入、削除、または更新の操作によって影響を受けたローの数を示します。影響を受けたローがない場合はゼロです。

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　成功したときは true、失敗したときは false。システム・リソースが不足していると、基本のインタフェース・ライブラリの初期化に失敗することがありますが、この場合、このメソッドは失敗することがあります。

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 または警告である場合は true、エラーを示している場合は 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

クラス ULSqlcaWrap

構文

public **ULSqlcaWrap**

基本クラス

- 「クラス ULSqlcaBase」278 ページ

説明

既存の SQLCA オブジェクトにアタッチする 「クラス ULSqlcaBase」278 ページ サブクラスです。

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 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」278 ページ
- 「GetCA 関数」279 ページ
- 「GetParameter 関数」279 ページ
- 「GetParameterCount 関数」279 ページ
- 「GetSQLCode 関数」280 ページ
- 「GetSQLCount 関数」280 ページ
- 「GetSQLErrorOffset 関数」280 ページ
- 「Initialize 関数」280 ページ
- 「LastCodeOK 関数」281 ページ
- 「LastFetchOK 関数」281 ページ
- 「ULSqlcaWrap 関数」282 ページ
- 「~ULSqlcaWrap 関数」283 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」280 ページ 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。

クラス UltraLite_Connection

構文

public **UltraLite_Connection**

基本クラス

- 「クラス UltraLite_SQLObject_iface」329 ページ
- 「クラス UltraLite_Connection_iface」286 ページ

説明

Connection クラスです。

メンバ

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

- 「AddRef 関数」329 ページ
- 「ChangeEncryptionKey 関数」287 ページ
- 「Commit 関数」287 ページ
- 「CountUploadRows 関数」287 ページ
- 「GetConnection 関数」330 ページ
- 「GetConnectionNum 関数」288 ページ
- 「GetDatabaseID メソッド」288 ページ
- 「GetDatabaseProperty 関数」288 ページ
- 「GetIFace 関数」330 ページ
- 「GetLastDownloadTime 関数」288 ページ
- 「GetLastIdentity 関数」289 ページ
- 「GetNewUUID 関数」289 ページ
- 「GetPublicationMask 関数」289 ページ
- 「GetSchema 関数」290 ページ
- 「GetSqlca 関数」290 ページ
- 「GetSuspend 関数」290 ページ
- 「GetSynchResult 関数」290 ページ
- 「GetUtilityULValue 関数」290 ページ
- 「GlobalAutoincUsage 関数」291 ページ
- 「GrantConnectTo 関数」291 ページ
- 「InitSynchInfo 関数」291 ページ
- 「OpenTable 関数」292 ページ
- 「OpenTableWithIndex 関数」292 ページ
- 「PrepareStatement 関数」293 ページ
- 「Release 関数」330 ページ
- 「ResetLastDownloadTime 関数」293 ページ

- 「RevokeConnectFrom 関数」293 ページ
- 「Rollback 関数」294 ページ
- 「RollbackPartialDownload 関数」294 ページ
- 「SetDatabaseID 関数」294 ページ
- 「SetSuspend 関数」294 ページ
- 「SetSynchInfo 関数」295 ページ
- 「Shutdown 関数」295 ページ
- 「StartSynchronizationDelete 関数」295 ページ
- 「StopSynchronizationDelete 関数」295 ページ
- 「StrToUUID 関数」296 ページ
- 「Synchronize 関数」296 ページ
- 「UpgradeSchemaFromFile 関数」298 ページ
- 「UpgradeSchemaFromFile 関数」298 ページ
- 「UUIDToStr 関数」297 ページ
- 「UUIDToStr 関数」297 ページ

クラス UltraLite_Connection_iface

構文

public **UltraLite_Connection_iface**

派生クラス

- 「クラス UltraLite_Connection」284 ページ

説明

Connection 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」287 ページ
- 「Commit 関数」287 ページ
- 「CountUploadRows 関数」287 ページ
- 「GetConnectionNum 関数」288 ページ
- 「GetDatabaseID メソッド」288 ページ
- 「GetDatabaseProperty 関数」288 ページ
- 「GetLastDownloadTime 関数」288 ページ
- 「GetLastIdentity 関数」289 ページ
- 「GetNewUUID 関数」289 ページ
- 「GetPublicationMask 関数」289 ページ
- 「GetSchema 関数」290 ページ
- 「GetSqlca 関数」290 ページ
- 「GetSuspend 関数」290 ページ
- 「GetSynchResult 関数」290 ページ
- 「GetUtilityULValue 関数」290 ページ
- 「GlobalAutoincUsage 関数」291 ページ
- 「GrantConnectTo 関数」291 ページ
- 「InitSynchInfo 関数」291 ページ
- 「OpenTable 関数」292 ページ
- 「OpenTableWithIndex 関数」292 ページ
- 「PrepareStatement 関数」293 ページ
- 「ResetLastDownloadTime 関数」293 ページ
- 「RevokeConnectFrom 関数」293 ページ
- 「Rollback 関数」294 ページ
- 「RollbackPartialDownload 関数」294 ページ
- 「SetDatabaseID 関数」294 ページ
- 「SetSuspend 関数」294 ページ

- 「SetSynchInfo 関数」295 ページ
- 「Shutdown 関数」295 ページ
- 「StartSynchronizationDelete 関数」295 ページ
- 「StopSynchronizationDelete 関数」295 ページ
- 「StrToUUID 関数」296 ページ
- 「Synchronize 関数」296 ページ
- 「UpgradeSchemaFromFile 関数」298 ページ
- 「UpgradeSchemaFromFile 関数」298 ページ
- 「UUIDToStr 関数」297 ページ
- 「UUIDToStr 関数」297 ページ

ChangeEncryptionKey 関数

構文

virtual 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。

Commit 関数

構文

virtual 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

virtual 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

virtual ul_connection_num
UltraLite_Connection_iface::GetConnectionNum()

説明 接続の数を取得します。

GetDatabaseID メソッド

構文 virtual 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

virtual 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

virtual 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** 出力：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　virtual ul_u_big **UltraLite_Connection_iface::GetLastIdentity()**

説明　@@identity の値を取得します。

GetNewUUID 関数

構文

virtual 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

virtual 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名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、パブリケーション ID ではありません。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 virtual 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 virtual 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 virtual 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、そうでない場合は false

GetSynchResult 関数

構文 virtual bool **UltraLite_Connection_iface::GetSynchResult(**
p_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 virtual ULValue **UltraLite_Connection_iface::GetUtilityULValue()**

説明

新しい 「クラス ULValue」356 ページ を取得します。

「クラス ULValue」356 ページ 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

virtual ul_u_short **UltraLite_Connection_iface::GlobalAutoincUsage()**

説明

グローバル・オートインクリメント・カウンタの使用率を取得します。

GrantConnectTo 関数

構文

virtual bool **UltraLite_Connection_iface::GrantConnectTo(**
const ULValue & *uid*
const ULValue & *pwd*
)

パラメータ

- **uid** 接続する権限を付与されるユーザ ID
- **pwd** 接続するためにユーザ ID が指定する必要があるパスワード

説明

新しいユーザを追加したり、既存のユーザのパスワードを変更したりします。

InitSynchInfo 関数

構文

virtual void **UltraLite_Connection_iface::InitSynchInfo(**
p_ul_synch_info *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

virtual 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。

テーブルを初めて開いたときは、カーソルの位置は BeforeFirst() です。

OpenTableWithIndex 関数

構文

virtual 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のインデックスを使用して、テーブルを開きます。

テーブルを初めて開いたときは、カーソルの位置は BeforeFirst() です。

PrepareStatement 関数

構文

virtual 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

virtual 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

virtual 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 virtual 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 virtual 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 virtual 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。

SetSuspend 関数

構文 virtual 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。

true の場合、この接続はサスペンドされ、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。

SetSynchInfo 関数

構文

virtual bool **UltraLite_Connection_iface::SetSynchInfo(**
p_ul_synch_info *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

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

virtual 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。

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

virtual bool **UltraLite_Connection_iface::StartSynchronizationDelete()**

説明

この接続の START SYNCHRONIZATION DELETE です。

StopSynchronizationDelete 関数

構文

virtual bool **UltraLite_Connection_iface::StopSynchronizationDelete()**

説明

この接続の STOP SYNCHRONIZATION DELETE です。

StrToUUID 関数

構文

virtual 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

virtual bool **UltraLite_Connection_iface::Synchronize(**
p_ul_synch_info *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。

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

または

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

UUIDToStr 関数

構文

virtual 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

virtual 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。

UpgradeSchemaFromFile 関数

構文

virtual bool **UltraLite_Connection_iface::UpgradeSchemaFromFile(**
const ULValue & *options*
)

パラメータ

- **options** セミコロンで区切られたアップグレードのオプションのリスト

説明 外部のスキーマ定義ファイルを使用して、新しいスキーマにアップグレードします。

options は、セミコロンで区切られたアップグレードのオプションのリストです。

UpgradeSchemaFromFile 関数

構文

virtual bool **UltraLite_Connection_iface::UpgradeSchemaFromFile(**
const ULValue & *options*
ul_schema_upgrade_observer_fn *callback*
ul_void * *user_data*
)

パラメータ

- **options** オプションのリスト
- **callback** Upgrade observer コールバック関数
- **user_data** コールバック関数に渡されるユーザのデータ

説明 外部のスキーマ定義ファイルを使用して、新しいスキーマにアップグレードします。

options は、セミコロンで区切られたアップグレードのオプションのリストです。

クラス UltraLite_Cursor_iface

構文　public **UltraLite_Cursor_iface**

派生クラス

- 「クラス UltraLite_ResultSet」322 ページ
- 「クラス UltraLite_Table」336 ページ

説明　Cursor 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

- 「AfterLast 関数」299 ページ
- 「BeforeFirst 関数」299 ページ
- 「First 関数」300 ページ
- 「Get 関数」300 ページ
- 「GetRowCount 関数」300 ページ
- 「GetState 関数」300 ページ
- 「GetStreamReader 関数」301 ページ
- 「GetSuspend 関数」301 ページ
- 「IsNull 関数」301 ページ
- 「Last 関数」301 ページ
- 「Next 関数」302 ページ
- 「Previous 関数」302 ページ
- 「Relative 関数」302 ページ
- 「SetSuspend 関数」302 ページ

AfterLast 関数

構文　virtual 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　virtual 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。

First 関数

構文 virtual 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 virtual 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 virtual 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 virtual 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 enum UL_RS_STATE を参照してください。

GetStreamReader 関数

構文

virtual 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

チャンク内の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。

GetSuspend 関数

構文

virtual 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

この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

virtual 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

virtual 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

virtual bool **UltraLite_Cursor_iface::Next()**

説明

カーソルをロー 1 つ分進めます。

次のローがない場合は失敗します。この結果、カーソル位置は「AfterLast 関数」299 ページ となります。

Previous 関数

構文

virtual 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」299 ページ となります。

Relative 関数

構文

virtual 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。

SetSuspend 関数

構文

virtual void **UltraLite_Cursor_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の値を設定します。

true の場合、このカーソルはサスペンドされ、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。

クラス UltraLite_DatabaseManager

構文

public **UltraLite_DatabaseManager**

基本クラス

- 「クラス UltraLite_DatabaseManager_iface」305 ページ

説明

DatabaseManager クラスです。

メンバ

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

- 「CreateAndOpenDatabase 関数」305 ページ
- 「DropDatabase 関数」306 ページ
- 「OpenConnection 関数」306 ページ
- 「Shutdown 関数」307 ページ

クラス UltraLite_DatabaseManager_iface

構文

public **UltraLite_DatabaseManager_iface**

派生クラス

- 「クラス UltraLite_DatabaseManager」304 ページ

説明

DatabaseManager 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_DatabaseManager_iface のすべてのメンバ

- 「CreateAndOpenDatabase 関数」305 ページ
- 「DropDatabase 関数」306 ページ
- 「OpenConnection 関数」306 ページ
- 「Shutdown 関数」307 ページ

CreateAndOpenDatabase 関数

構文

virtual UltraLite_Connection *
UltraLite_DatabaseManager_iface::CreateAndOpenDatabase(
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** 作成および接続のパラメータ

説明

新しいデータベースを作成して、接続します。

この sqlca は新しい接続に関連付けられます。データベースがすでに存在する場合、この関数は失敗します。

戻り値

この関数が成功した場合、新しい接続オブジェクトが返されます。失敗した場合は、NULL が返されます。

DropDatabase 関数

構文

virtual 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

既存のデータベースを消去します。

消去するには、データベースが停止している必要があります。

OpenConnection 関数

構文

virtual 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 － この sqlca と接続名 CON のパラメータを使用して、すでに接続されています。
- SQLE_INVALID_LOGON － 指定したユーザ ID が存在しないか、パスワードが正しくありません。ユーザ認証が有効になるのは、ULEnableUserAuthentication 関数を呼び出してから接続した場合だけです。

- SQLE_INVALID_SQL_IDENTIFIER － 指定したユーザ ID またはパスワードが無効であるか、ユーザ ID が指定されていませんでした (UID= ユーザ ;PWD= パスワード を使用してください)。
- SQLE_TOO_MANY_CONNECTIONS － 開いた接続が多すぎるため、同時接続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 「クラス ULSqlca」276 ページ オブジェクトを使用します。

戻り値

この関数が成功した場合、新しい接続オブジェクトが返されます。失敗した場合は、NULL が返されます。

Shutdown 関数

構文

virtual 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)。

クラス UltraLite_DatabaseSchema

構文

public **UltraLite_DatabaseSchema**

基本クラス

- 「クラス UltraLite_SQLObject_iface」329 ページ
- 「クラス UltraLite_DatabaseSchema_iface」309 ページ

説明

DatabaseSchema クラスです。

メンバ

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

- 「AddRef 関数」329 ページ
- 「GetCollationName 関数」309 ページ
- 「GetConnection 関数」330 ページ
- 「GetIFace 関数」330 ページ
- 「GetPublicationCount 関数」309 ページ
- 「GetPublicationID 関数」310 ページ
- 「GetPublicationMask 関数」310 ページ
- 「GetPublicationName 関数」310 ページ
- 「GetSignature 関数」311 ページ
- 「GetTableCount 関数」311 ページ
- 「GetTableName 関数」311 ページ
- 「GetTableSchema 関数」311 ページ
- 「IsCaseSensitive 関数」312 ページ
- 「Release 関数」330 ページ

クラス UltraLite_DatabaseSchema_iface

構文　public **UltraLite_DatabaseSchema_iface**

派生クラス

- 「クラス UltraLite_DatabaseSchema」308 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」309 ページ
- 「GetPublicationCount 関数」309 ページ
- 「GetPublicationID 関数」310 ページ
- 「GetPublicationMask 関数」310 ページ
- 「GetPublicationName 関数」310 ページ
- 「GetSignature 関数」311 ページ
- 「GetTableCount 関数」311 ページ
- 「GetTableName 関数」311 ページ
- 「GetTableSchema 関数」311 ページ
- 「IsCaseSensitive 関数」312 ページ

GetCollationName 関数

構文　virtual 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 「クラス ULValue」356 ページ が返されます。

GetPublicationCount 関数

構文　virtual 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 ～ 「GetPublicationCount 関数」309 ページ です。

GetPublicationID 関数

構文

virtual 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

virtual 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。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。

パブリケーションが見つからない場合は、0 が返されます。

GetPublicationName 関数

構文

virtual 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。

GetSignature 関数

構文

virtual ULValue **UltraLite_DatabaseSchema_iface::GetSignature()**

説明

データベースのシグニチャを取得します。

GetTableCount 関数

構文

virtual ul_table_num **UltraLite_DatabaseSchema_iface::GetTableCount()**

説明

テーブルの数を取得します。

GetTableName 関数

構文

virtual 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。

テーブルが存在しない場合、返される 「クラス ULValue」356 ページ オブジェクトは空です。

GetTableSchema 関数

構文

virtual 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 ID または名前を指定して TableSchema オブジェクトを取得します。テーブルが存在しない場合は UL_NULL が返されます。

IsCaseSensitive 関数

構文

virtual 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。

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 が返されます。

クラス UltraLite_IndexSchema

構文

public **UltraLite_IndexSchema**

基本クラス

- 「クラス UltraLite_SQLObject_iface」329 ページ
- 「クラス UltraLite_IndexSchema_iface」314 ページ

説明

IndexSchema クラスです。

メンバ

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

- 「AddRef 関数」329 ページ
- 「GetColumnCount 関数」314 ページ
- 「GetColumnName 関数」314 ページ
- 「GetConnection 関数」330 ページ
- 「GetID 関数」315 ページ
- 「GetIFace 関数」330 ページ
- 「GetName 関数」315 ページ
- 「GetReferencedIndexName 関数」315 ページ
- 「GetReferencedTableName 関数」316 ページ
- 「GetTableName 関数」316 ページ
- 「IsColumnDescending 関数」316 ページ
- 「IsForeignKey 関数」316 ページ
- 「IsForeignKeyCheckOnCommit 関数」317 ページ
- 「IsForeignKeyNullable 関数」317 ページ
- 「IsPrimaryKey 関数」317 ページ
- 「IsUniqueIndex 関数」317 ページ
- 「IsUniqueKey 関数」317 ページ
- 「Release 関数」330 ページ

クラス UltraLite_IndexSchema_iface

構文 public **UltraLite_IndexSchema_iface**

派生クラス

- 「クラス UltraLite_IndexSchema」313 ページ

説明 IndexSchema 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_IndexSchema_iface のすべてのメンバ

- 「GetColumnCount 関数」314 ページ
- 「GetColumnName 関数」314 ページ
- 「GetID 関数」315 ページ
- 「GetName 関数」315 ページ
- 「GetReferencedIndexName 関数」315 ページ
- 「GetReferencedTableName 関数」316 ページ
- 「GetTableName 関数」316 ページ
- 「IsColumnDescending 関数」316 ページ
- 「IsForeignKey 関数」316 ページ
- 「IsForeignKeyCheckOnCommit 関数」317 ページ
- 「IsForeignKeyNullable 関数」317 ページ
- 「IsPrimaryKey 関数」317 ページ
- 「IsUniqueIndex 関数」317 ページ
- 「IsUniqueKey 関数」317 ページ

GetColumnCount 関数

構文 virtual 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

```
virtual 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。

カラムが存在しない場合、返される 「クラス ULValue」356 ページ オブジェクトは空です。カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND が返されます。

GetID 関数

構文

virtual 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

このインデックスの ID です。

GetName 関数

構文

virtual 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

virtual 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。

このメソッドは、外部キー専用です。インデックスが外部キーではない場合、返される 「クラス ULValue」356 ページ オブジェクトは空です。

GetReferencedTableName 関数

構文

virtual 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。インデックスが外部キーではない場合、返される 「クラス ULValue」356 ページ オブジェクトは空です。

GetTableName 関数

構文

virtual 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

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

パラメータ

- **column_name** カラムの名前

説明

カラムが降順に設定されている場合、true 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。

IsForeignKey 関数

構文

virtual bool **UltraLite_IndexSchema_iface::IsForeignKey()**

説明

インデックスが外部キーである場合、true を返します。

IsForeignKeyCheckOnCommit 関数

構文　virtual bool **UltraLite_IndexSchema_iface::IsForeignKeyCheckOnCommit()**

説明　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 を返します。

そうでない場合、参照整合性は挿入時にチェックされます。

IsForeignKeyNullable 関数

構文　virtual bool **UltraLite_IndexSchema_iface::IsForeignKeyNullable()**

説明　インデックスが一意の外部キー制約である場合、true を返します。

IsPrimaryKey 関数

構文　virtual bool **UltraLite_IndexSchema_iface::IsPrimaryKey()**

説明　インデックスがプライマリ・キーである場合、true を返します。

IsUniqueIndex 関数

構文　virtual bool **UltraLite_IndexSchema_iface::IsUniqueIndex()**

説明　インデックスがユニーク・インデックスである場合、true を返します。

IsUniqueKey 関数

構文　virtual bool **UltraLite_IndexSchema_iface::IsUniqueKey()**

説明　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、true を返します。

クラス UltraLite_PreparedStatement

構文

public **UltraLite_PreparedStatement**

基本クラス

- 「クラス UltraLite_SQLObject_iface」329 ページ
- 「クラス UltraLite_PreparedStatement_iface」319 ページ

説明

PreparedStatement クラスです。

メンバ

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

- 「AddRef 関数」329 ページ
- 「ExecuteQuery 関数」319 ページ
- 「ExecuteStatement 関数」319 ページ
- 「GetConnection 関数」330 ページ
- 「GetIFace 関数」330 ページ
- 「GetPlan 関数」320 ページ
- 「GetSchema 関数」320 ページ
- 「GetStreamWriter 関数」320 ページ
- 「HasResultSet 関数」320 ページ
- 「Release 関数」330 ページ
- 「SetParameter 関数」321 ページ
- 「SetParameterNull 関数」321 ページ

クラス UltraLite_PreparedStatement_iface

構文　public **UltraLite_PreparedStatement_iface**

派生クラス

- 「クラス UltraLite_PreparedStatement」318 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」319 ページ
- 「ExecuteStatement 関数」319 ページ
- 「GetPlan 関数」320 ページ
- 「GetSchema 関数」320 ページ
- 「GetStreamWriter 関数」320 ページ
- 「HasResultSet 関数」320 ページ
- 「SetParameter 関数」321 ページ
- 「SetParameterNull 関数」321 ページ

ExecuteQuery 関数

構文　virtual UltraLite_ResultSet * **UltraLite_PreparedStatement_iface::ExecuteQuery()**

説明　SQL クエリを実行します。

ResultSet オブジェクトが返されます。

ExecuteStatement 関数

構文　virtual ul_s_long **UltraLite_PreparedStatement_iface::ExecuteStatement()**

説明　SQL 文を実行します。

GetPlan 関数

構文

virtual size_t **UltraLite_PreparedStatement_iface::GetPlan(**
ul_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** ul_char のバッファ・サイズ

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。

GetSchema 関数

構文

virtual 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

virtual 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

virtual bool **UltraLite_PreparedStatement_iface::HasResultSet()**

説明　この SQL 文に結果セットがあるかどうかを調べます。

SetParameter 関数

構文

virtual 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

virtual 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。

クラス UltraLite_ResultSet

構文

public **UltraLite_ResultSet**

基本クラス

- 「クラス UltraLite_SQLObject_iface」329 ページ
- 「クラス UltraLite_ResultSet_iface」323 ページ
- 「クラス UltraLite_Cursor_iface」299 ページ

説明

ResultSet クラスです。

メンバ

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

- 「AddRef 関数」329 ページ
- 「AfterLast 関数」299 ページ
- 「BeforeFirst 関数」299 ページ
- 「First 関数」300 ページ
- 「Get 関数」300 ページ
- 「GetConnection 関数」330 ページ
- 「GetIFace 関数」330 ページ
- 「GetRowCount 関数」300 ページ
- 「GetSchema 関数」323 ページ
- 「GetState 関数」300 ページ
- 「GetStreamReader 関数」301 ページ
- 「GetSuspend 関数」301 ページ
- 「IsNull 関数」301 ページ
- 「Last 関数」301 ページ
- 「Next 関数」302 ページ
- 「Previous 関数」302 ページ
- 「Relative 関数」302 ページ
- 「Release 関数」330 ページ
- 「SetSuspend 関数」302 ページ

クラス UltraLite_ResultSet_iface

構文　public **UltraLite_ResultSet_iface**

派生クラス

- 「クラス UltraLite_ResultSet」322 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

- 「GetSchema 関数」323 ページ

GetSchema 関数

構文　virtual 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。

クラス UltraLite_ResultSetSchema

構文

public **UltraLite_ResultSetSchema**

基本クラス

- 「クラス UltraLite_SQLObject_iface」329 ページ
- 「クラス UltraLite_RowSchema_iface」325 ページ

説明

ResultSetSchema クラスです。

メンバ

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

- 「AddRef 関数」329 ページ
- 「GetColumnCount 関数」325 ページ
- 「GetColumnID 関数」325 ページ
- 「GetColumnName 関数」326 ページ
- 「GetColumnPrecision 関数」326 ページ
- 「GetColumnScale 関数」327 ページ
- 「GetColumnSize 関数」327 ページ
- 「GetColumnSQLType 関数」326 ページ
- 「GetColumnType 関数」328 ページ
- 「GetConnection 関数」330 ページ
- 「GetIFace 関数」330 ページ
- 「Release 関数」330 ページ

クラス UltraLite_RowSchema_iface

構文　public **UltraLite_RowSchema_iface**

派生クラス

- 「クラス UltraLite_ResultSetSchema」324 ページ
- 「クラス UltraLite_TableSchema」347 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

- 「GetColumnCount 関数」325 ページ
- 「GetColumnID 関数」325 ページ
- 「GetColumnName 関数」326 ページ
- 「GetColumnPrecision 関数」326 ページ
- 「GetColumnScale 関数」327 ページ
- 「GetColumnSize 関数」327 ページ
- 「GetColumnSQLType 関数」326 ページ
- 「GetColumnType 関数」328 ページ

GetColumnCount 関数

構文　virtual 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　virtual 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。カラムが存在しない場合は 0 が返されます。カラム名が存在しない場合は SQLE_COLUMN_NOT_FOUND が設定されます。

GetColumnName 関数

構文

virtual 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。カラムが存在しない場合、返される 「クラス ULValue」356 ページ オブジェクトは空です。

カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND が設定されます。

GetColumnPrecision 関数

構文

virtual 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。エラーは GetColumnScale と同じです。

GetColumnSQLType 関数

構文

virtual 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。カラムが存在しない場合は UL_SQLTYPE_BAD_INDEX が返されます。SQL エラーは設定されません。

GetColumnScale 関数

構文

virtual 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。

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 が返されます。カラム名が存在しない場合は SQLE_COLUMN_NOT_FOUND が設定されます。カラムが数値型ではない場合は SQLE_DATATYPE_NOT_ALLOWED が設定されます。

GetColumnSize 関数

構文

virtual 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。

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 が返されます。カラム名が存在しない場合は SQLE_COLUMN_NOT_FOUND が設定されます。カラムの型が UL_SQLTYPE_CHAR または UL_SQLTYPE_BINARY ではない場合は SQLE_DATATYPE_NOT_ALLOWED が設定されます。

GetColumnType 関数

構文

virtual 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。カラムが存在しない場合は UL_TYPE_BAD_INDEX が返されます。SQL エラーは設定されません。

クラス UltraLite_SQLObject_iface

構文

public **UltraLite_SQLObject_iface**

派生クラス

- 「クラス UltraLite_Connection」284 ページ
- 「クラス UltraLite_DatabaseSchema」308 ページ
- 「クラス UltraLite_IndexSchema」313 ページ
- 「クラス UltraLite_PreparedStatement」318 ページ
- 「クラス UltraLite_ResultSet」322 ページ
- 「クラス UltraLite_ResultSetSchema」324 ページ
- 「クラス UltraLite_StreamReader」331 ページ
- 「クラス UltraLite_StreamWriter」335 ページ
- 「クラス UltraLite_Table」336 ページ
- 「クラス UltraLite_TableSchema」347 ページ

説明

SQLObject 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_SQLObject_iface のすべてのメンバ

- 「AddRef 関数」329 ページ
- 「GetConnection 関数」330 ページ
- 「GetIFace 関数」330 ページ
- 「Release 関数」330 ページ

AddRef 関数

構文

virtual 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、「AddRef 関数」329 ページ への呼び出しと 「Release 関数」330 ページ への呼び出しを対にする必要があります。

GetConnection 関数

構文

virtual 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

virtual 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る

説明

今後の使用のために予約されています。

Release 関数

構文

virtual 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。「Release 関数」330 ページ は、少なくとも 1 回呼び出す必要があるほか、「AddRef 関数」329 ページ を使用した場合は、それぞれの「AddRef 関数」329 ページ と対になるようにして呼び出す必要があります。

クラス UltraLite_StreamReader

構文

public **UltraLite_StreamReader**

基本クラス

- 「クラス UltraLite_SQLObject_iface」329 ページ
- 「クラス UltraLite_StreamReader_iface」332 ページ

説明

StreamReader クラスです。

メンバ

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

- 「AddRef 関数」329 ページ
- 「GetByteChunk 関数」332 ページ
- 「GetConnection 関数」330 ページ
- 「GetIFace 関数」330 ページ
- 「GetLength 関数」333 ページ
- 「GetStringChunk 関数」333 ページ
- 「GetStringChunk 関数」334 ページ
- 「Release 関数」330 ページ
- 「SetReadPosition 関数」334 ページ

クラス UltraLite_StreamReader_iface

構文

public **UltraLite_StreamReader_iface**

派生クラス

- 「クラス UltraLite_StreamReader」331 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、文字列 (varchar) カラムおよびバイナリ・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」332 ページ
- 「GetLength 関数」333 ページ
- 「GetStringChunk 関数」333 ページ
- 「GetStringChunk 関数」334 ページ
- 「SetReadPosition 関数」334 ページ

GetByteChunk 関数

構文

virtual bool **UltraLite_StreamReader_iface::GetByteChunk(**
ul_byte * *data*
size_t *buffer_len*
size_t * *len_retn*
bool * *morebytes*
)

パラメータ

- **data** バイトのバッファ
- **buffer_len** バッファの長さ
- **len_retn** 出力：返される長さ
- **morebytes** 出力：さらに読み込むバイトがある場合は true

説明

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。

buffer_len のバイトをバッファ data にコピーします。
「SetReadPosition 関数」334 ページ が使用されないかぎり、前回の読み込みが終了したところからバイトが読み込まれます。

GetLength 関数

構文

virtual size_t **UltraLite_StreamReader_iface::GetLength(**
bool *fetching_as_wide*
)

パラメータ

- **fetching_as_wide** 使用されるフェッチの文字列型。バイナリの場合は false を使用します。

説明

文字列またはバイナリの値の長さを取得します。

バイナリの場合、および文字列が ANSI (fetch_is_wide = false) としてフェッチされる場合は、バイト数が返されます。UNICODE (fetch_is_wide = true) としてフェッチされる場合は、ユニコード文字の数が返されます。

GetStringChunk 関数

構文

virtual bool **UltraLite_StreamReader_iface::GetStringChunk(**
ul_wchar * *str*
size_t *buffer_len*
size_t * *len_retn*
bool * *morebytes*
)

パラメータ

- **str** ワイド文字のバッファ
- **buffer_len** バッファの長さ
- **len_retn** 出力：返される長さ
- **morebytes** 出力：さらに読み込む文字がある場合は true

説明

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。

buffer_len のワイド文字をバッファ str にコピーします。
「SetReadPosition 関数」334 ページ が使用されないかぎり、前回の読み込みが終了したところから文字が読み込まれます。

GetStringChunk 関数

構文

virtual bool **UltraLite_StreamReader_iface::GetStringChunk(**
char * *str*
size_t *buffer_len*
size_t * *len_retn*
bool * *morebytes*
)

パラメータ

- **str** 文字のバッファ
- **buffer_len** バッファの長さ
- **len_retn** 出力：返される長さ
- **morebytes** 出力：さらに読み込む文字がある場合は true

説明

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。

buffer_len のバイトをバッファ str にコピーします。
「SetReadPosition 関数」334 ページ が使用されないかぎり、前回の読み込みが終了したところから文字が読み込まれます。

SetReadPosition 関数

構文

virtual bool **UltraLite_StreamReader_iface::SetReadPosition(**
size_t *offset*
)

パラメータ

- **offset** オフセット。文字列の場合、オフセットは文字単位です。

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。

クラス UltraLite_StreamWriter

構文

public **UltraLite_StreamWriter**

基本クラス

- 「クラス UltraLite_SQLObject_iface」329 ページ

説明

StreamWriter クラスです。

メンバ

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

- 「AddRef 関数」329 ページ
- 「GetConnection 関数」330 ページ
- 「GetIFace 関数」330 ページ
- 「Release 関数」330 ページ

クラス UltraLite_Table

構文

public **UltraLite_Table**

基本クラス

- 「クラス UltraLite_SQLObject_iface」329 ページ
- 「クラス UltraLite_Table_iface」338 ページ
- 「クラス UltraLite_Cursor_iface」299 ページ

説明

Table クラスです。

メンバ

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

- 「AddRef 関数」329 ページ
- 「AfterLast 関数」299 ページ
- 「BeforeFirst 関数」299 ページ
- 「Delete 関数」338 ページ
- 「DeleteAllRows 関数」339 ページ
- 「Find 関数」339 ページ
- 「FindBegin 関数」340 ページ
- 「FindFirst 関数」340 ページ
- 「FindLast 関数」340 ページ
- 「FindNext 関数」341 ページ
- 「FindPrevious 関数」341 ページ
- 「First 関数」300 ページ
- 「Get 関数」300 ページ
- 「GetConnection 関数」330 ページ
- 「GetIFace 関数」330 ページ
- 「GetRowCount 関数」300 ページ
- 「GetSchema 関数」342 ページ
- 「GetState 関数」300 ページ
- 「GetStreamReader 関数」301 ページ
- 「GetStreamWriter 関数」342 ページ
- 「GetSuspend 関数」301 ページ
- 「Insert 関数」342 ページ
- 「InsertBegin 関数」342 ページ
- 「IsNull 関数」301 ページ
- 「Last 関数」301 ページ
- 「Lookup 関数」343 ページ

- 「LookupBackward 関数」343 ページ
- 「LookupBegin 関数」344 ページ
- 「LookupForward 関数」344 ページ
- 「Next 関数」302 ページ
- 「Previous 関数」302 ページ
- 「Relative 関数」302 ページ
- 「Release 関数」330 ページ
- 「Set 関数」345 ページ
- 「SetDefault 関数」345 ページ
- 「SetNull 関数」345 ページ
- 「SetSuspend 関数」302 ページ
- 「TruncateTable 関数」345 ページ
- 「Update 関数」346 ページ
- 「UpdateBegin 関数」346 ページ

クラス UltraLite_Table_iface

構文 public **UltraLite_Table_iface**

派生クラス

- 「クラス UltraLite_Table」336 ページ

説明 Table 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_Table_iface のすべてのメンバ

- 「Delete 関数」338 ページ
- 「DeleteAllRows 関数」339 ページ
- 「Find 関数」339 ページ
- 「FindBegin 関数」340 ページ
- 「FindFirst 関数」340 ページ
- 「FindLast 関数」340 ページ
- 「FindNext 関数」341 ページ
- 「FindPrevious 関数」341 ページ
- 「GetSchema 関数」342 ページ
- 「GetStreamWriter 関数」342 ページ
- 「Insert 関数」342 ページ
- 「InsertBegin 関数」342 ページ
- 「Lookup 関数」343 ページ
- 「LookupBackward 関数」343 ページ
- 「LookupBegin 関数」344 ページ
- 「LookupForward 関数」344 ページ
- 「Set 関数」345 ページ
- 「SetDefault 関数」345 ページ
- 「SetNull 関数」345 ページ
- 「TruncateTable 関数」345 ページ
- 「Update 関数」346 ページ
- 「UpdateBegin 関数」346 ページ

Delete 関数

構文 virtual bool **UltraLite_Table_iface::Delete()**

説明

現在の行を削除します。

カーソルの位置は、次の有効なロー上になります。

DeleteAllRows 関数

構文

virtual 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。

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」339 ページ 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

成功したときは true、失敗したとき (テーブルが開いていない、SQL エラーなど) は false

Find 関数

構文

virtual 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

FindFirst と同じです。

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 AfterLast() となり、false が返されます。

FindBegin 関数

構文

virtual bool **UltraLite_Table_iface::FindBegin()**

説明

カラムの設定に使用される検索モードを選択します。

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

virtual 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 AfterLast() となり、false が返されます。

FindLast 関数

構文

virtual 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

virtual 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 (順方向の) ローを取得します。

それ以上インデックスに一致するローがない場合は false を返します。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

virtual 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 (逆方向の) ローを取得します。

それ以上インデックスに一致するローがない場合は false を返します。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

virtual 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。

GetStreamWriter 関数

構文

virtual UltraLite_StreamWriter * **UltraLite_Table_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。

Insert 関数

構文

virtual 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」342 ページ を使用します。

InsertBegin 関数

構文

virtual bool **UltraLite_Table_iface::InsertBegin()**

説明

カラムの設定に使用され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

virtual 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

LookupForward と同じです。

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、それより大きい値の最初のローに配置されます。

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

virtual 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、それより小さい値の最後のローに配置されます。
- 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。

- 複合インデックスの場合、ncols はルックアップで使用するカラムの数を指定します。

LookupBegin 関数

構文

virtual bool **UltraLite_Table_iface::LookupBegin()**

説明

カラムの設定に使用されるルックアップ・モードを選択します。

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

virtual 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、それより大きい値の最初のローに配置されます。
- 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。

Set 関数

構文

virtual bool **UltraLite_Table_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

virtual bool **UltraLite_Table_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

virtual bool **UltraLite_Table_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。

TruncateTable 関数

構文

virtual bool **UltraLite_Table_iface::TruncateTable()**

説明

テーブルをトランケートします。

STOP SYNCHRONIZATION DELETE を一時的にアクティブにします。

Update 関数

構文

virtual bool **UltraLite_Table_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」346 ページ を使用します。

UpdateBegin 関数

構文

virtual bool **UltraLite_Table_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・キー内のカラムが修正されないことがあります。

クラス UltraLite_TableSchema

構文

public **UltraLite_TableSchema**

基本クラス

- 「クラス UltraLite_SQLObject_iface」329 ページ
- 「クラス UltraLite_TableSchema_iface」349 ページ
- 「クラス UltraLite_RowSchema_iface」325 ページ

説明

TableSchema クラスです。

メンバ

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

- 「AddRef 関数」329 ページ
- 「GetColumnCount 関数」325 ページ
- 「GetColumnDefault 関数」349 ページ
- 「GetColumnID 関数」325 ページ
- 「GetColumnName 関数」326 ページ
- 「GetColumnPrecision 関数」326 ページ
- 「GetColumnScale 関数」327 ページ
- 「GetColumnSize 関数」327 ページ
- 「GetColumnSQLType 関数」326 ページ
- 「GetColumnType 関数」328 ページ
- 「GetConnection 関数」330 ページ
- 「GetGlobalAutoincPartitionSize 関数」350 ページ
- 「GetID 関数」350 ページ
- 「GetIFace 関数」330 ページ
- 「GetIndexCount 関数」350 ページ
- 「GetIndexName 関数」351 ページ
- 「GetIndexSchema 関数」351 ページ
- 「GetName 関数」351 ページ
- 「GetOptimalIndex 関数」351 ページ
- 「GetPrimaryKey 関数」352 ページ
- 「GetUploadUnchangedRows 関数」352 ページ
- 「InPublication 関数」352 ページ
- 「IsColumnAutoinc 関数」353 ページ
- 「IsColumnCurrentDate 関数」353 ページ
- 「IsColumnCurrentTime 関数」353 ページ

- 「IsColumnCurrentTimestamp 関数」354 ページ
- 「IsColumnGlobalAutoinc 関数」354 ページ
- 「IsColumnInIndex 関数」354 ページ
- 「IsColumnNewUUID 関数」355 ページ
- 「IsColumnNullable 関数」355 ページ
- 「IsNeverSynchronized 関数」355 ページ
- 「Release 関数」330 ページ

クラス UltraLite_TableSchema_iface

構文　public **UltraLite_TableSchema_iface**

派生クラス

- 「クラス UltraLite_TableSchema」347 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」349 ページ
- 「GetGlobalAutoincPartitionSize 関数」350 ページ
- 「GetID 関数」350 ページ
- 「GetIndexCount 関数」350 ページ
- 「GetIndexName 関数」351 ページ
- 「GetIndexSchema 関数」351 ページ
- 「GetName 関数」351 ページ
- 「GetOptimalIndex 関数」351 ページ
- 「GetPrimaryKey 関数」352 ページ
- 「GetUploadUnchangedRows 関数」352 ページ
- 「InPublication 関数」352 ページ
- 「IsColumnAutoinc 関数」353 ページ
- 「IsColumnCurrentDate 関数」353 ページ
- 「IsColumnCurrentTime 関数」353 ページ
- 「IsColumnCurrentTimestamp 関数」354 ページ
- 「IsColumnGlobalAutoinc 関数」354 ページ
- 「IsColumnInIndex 関数」354 ページ
- 「IsColumnNewUUID 関数」355 ページ
- 「IsColumnNullable 関数」355 ページ
- 「IsNeverSynchronized 関数」355 ページ

GetColumnDefault 関数

構文

virtual ULValue **UltraLite_TableSchema_iface::GetColumnDefault(**
const ULValue & *column_id*
)

パラメータ

- **column_id** 1 から始まる序数

説明

カラムにデフォルト値がある場合、それを取得します。返される「クラス ULValue」356 ページ オブジェクトには、デフォルト値が文字列として含まれています。

カラムにデフォルト値が含まれていない場合、返される 「クラス ULValue」356 ページ オブジェクトは空です。カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND が設定されます。

GetGlobalAutoincPartitionSize 関数

構文

```
virtual bool UltraLite_TableSchema_iface::GetGlobalAutoincPartitionSize(
    const ULValue & column_id
    ul_u_big * size
)
```

パラメータ

- **column_id** 1 から始まる序数
- **size** 出力：カラムの分割サイズ

説明

グローバル・オートインクリメント・カラムの分割サイズを返します。

GetID 関数

構文

virtual 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

virtual ul_index_num **UltraLite_TableSchema_iface::GetIndexCount()**

説明

テーブル内のインデックス数を返します。

GetIndexName 関数

構文

virtual 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。

インデックスが存在しない場合、返される 「クラス ULValue」356 ページ オブジェクトは空です。

GetIndexSchema 関数

構文

virtual 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。

インデックスが存在しない場合は、UL_NULL が返されます。

GetName 関数

構文

virtual ULValue **UltraLite_TableSchema_iface::GetName()**

説明

テーブルの名前を取得します。

GetOptimalIndex 関数

構文

virtual ULValue **UltraLite_TableSchema_iface::GetOptimalIndex(**
const ULValue & *column_id*
)

パラメータ

- **column_id** 1 から始まる序数

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。

インデックスの名前を返します。

GetPrimaryKey 関数

構文

virtual 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。

GetUploadUnchangedRows 関数

構文

virtual bool **UltraLite_TableSchema_iface::GetUploadUnchangedRows()**

説明

テーブルが各ローをアップロードする場合は true を返します。

前回のダウンロードから現在までに修正されていない場合でも、ローはアップロードされます。

InPublication 関数

構文

virtual 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

テーブルがパブリケーションに含まれている場合は true 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

virtual bool **UltraLite_TableSchema_iface::IsColumnAutoinc(**
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、true 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。

IsColumnCurrentDate 関数

構文

virtual bool **UltraLite_TableSchema_iface::IsColumnCurrentDate(**
const ULValue & *column_id*
)

パラメータ

- **column_id** 1 から始まる序数

説明

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、true を返します。

IsColumnCurrentTime 関数

構文

virtual bool **UltraLite_TableSchema_iface::IsColumnCurrentTime(**
const ULValue & *column_id*
)

パラメータ

- **column_id** 1 から始まる序数

説明

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、true を返します。

IsColumnCurrentTimestamp 関数

構文

virtual bool **UltraLite_TableSchema_iface::IsColumnCurrentTimestamp(**
const ULValue & *column_id*
)

パラメータ

- **column_id** 1 から始まる序数

説明

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、true を返します。

IsColumnGlobalAutoinc 関数

構文

virtual bool **UltraLite_TableSchema_iface::IsColumnGlobalAutoinc(**
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

virtual 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」325 ページ を呼び出します。
- **index_id** インデックスを識別する 1 から始まる序数 テーブル内のインデックス数を取得するには、「GetIndexCount 関数」350 ページ を呼び出します。

説明

カラムがインデックスに含まれる場合は true を返します。ID は 1 から始まる必要があります。

カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND が設定されます。インデックスが存在しない場合、SQLE_INDEX_NOT_FOUND が設定されます。

IsColumnNewUUID 関数

構文

virtual bool **UltraLite_TableSchema_iface::IsColumnNewUUID(**
const ULValue & *column_id*
)

パラメータ

- **column_id** 1 から始まる序数

説明

カラムに、デフォルト値として UUID が存在する場合、true を返します。

IsColumnNullable 関数

構文

virtual bool **UltraLite_TableSchema_iface::IsColumnNullable(**
const ULValue & *column_id*
)

パラメータ

- **column_id** 1 から始まる序数

説明

カラムが null 入力可能であれば true 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。

IsNeverSynchronized 関数

構文

virtual bool **UltraLite_TableSchema_iface::IsNeverSynchronized()**

説明

テーブルが同期から省かれている場合に true を返します。

クラス ULValue

構文

public **ULValue**

説明

ULValue オブジェクトです。

ULValue クラスは、Ultra Light カーソルに格納されるデータ型のラッパです。このため、データ型を気にすることなく、データを格納できます。

ULValue 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、明示的にインスタンス化しなくても、ULValue をシームレスに使用できます。

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」357 ページ
- 「GetBinary 関数」358 ページ
- 「GetBinaryLength 関数」358 ページ
- 「GetString 関数」359 ページ
- 「GetString 関数」359 ページ
- 「GetStringLength 関数」360 ページ
- 「InDatabase 関数」360 ページ
- 「IsNull 関数」361 ページ
- 「operator bool 関数」368 ページ
- 「operator DECL_DATETIME 関数」368 ページ
- 「operator double 関数」369 ページ
- 「operator float 関数」369 ページ
- 「operator int 関数」369 ページ
- 「operator long 関数」369 ページ
- 「operator short 関数」369 ページ
- 「operator ul_s_big 関数」369 ページ
- 「operator ul_u_big 関数」370 ページ
- 「operator unsigned char 関数」370 ページ
- 「operator unsigned int 関数」370 ページ
- 「operator unsigned long 関数」370 ページ
- 「operator unsigned short 関数」370 ページ
- 「operator= 関数」371 ページ
- 「SetBinary 関数」361 ページ

- 「SetString 関数」361 ページ
- 「SetString 関数」362 ページ
- 「StringCompare 関数」362 ページ
- 「ULValue 関数」362 ページ
- 「ULValue 関数」363 ページ
- 「ULValue 関数」363 ページ
- 「ULValue 関数」363 ページ
- 「ULValue 関数」364 ページ
- 「ULValue 関数」364 ページ
- 「ULValue 関数」364 ページ
- 「ULValue 関数」364 ページ
- 「ULValue 関数」365 ページ
- 「ULValue 関数」365 ページ
- 「ULValue 関数」365 ページ
- 「ULValue 関数」366 ページ
- 「ULValue 関数」366 ページ
- 「ULValue 関数」366 ページ
- 「ULValue 関数」366 ページ
- 「ULValue 関数」367 ページ
- 「ULValue 関数」367 ページ
- 「ULValue 関数」367 ページ
- 「ULValue 関数」368 ページ
- 「ULValue 関数」368 ページ
- 「~ULValue 関数」371 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

バイナリ値を取得します。

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：実際に返されたバイト数

説明

バイナリ値を取得します。

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」357 ページ によって返されるバイナリ値の保持に必要なバイト数

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

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。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も、出力文字列は常に null で終了します。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。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も、出力文字列は常に null で終了します。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。

GetStringLength 関数

構文

size_t **ULValue::GetStringLength(**
bool *fetching_as_wide*
)

パラメータ

- **fetching_as_wide** 使用されるフェッチの文字列型

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_type[ len ];
 ( dst, len ); GetString
```

この使用方法では、複数要素のエンコードのために、文字型が異なると長さが異なる場合があります。`fetching_as_wide` パラメータは、「GetString 関数」359 ページ の呼び出しで使用される文字型を示すために使用します。

戻り値

「GetString 関数」359 ページ 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ワイド文字数。これは、文字列内の文字数であるとはかぎりません。

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 を返します。

IsNull 関数

構文

bool **ULValue::IsNull()**

説明

NULL 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 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に true を返します。

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

バイナリ値を設定します。

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 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 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 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 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** 比較する文字列

説明

文字列を比較します。

文字列、または ULValue 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 を返します。
- 現在の値が `value` より小さい場合は -1 を返します。
- 現在の値が `value` より大きい場合は 1 を返します。
- エラー時には、いずれかの ULValue オブジェクトの sqlca が未設定の場合は -2、いずれかの ULValue オブジェクトの文字列表現が UL_NULL である場合は -3 が返されます。

ULValue 関数

構文

ULValue::ULValue()

説明　ULValue を構成します。

ULValue 関数

構文

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

パラメータ

- **vSrc** ULValue として扱われる値

説明　const から ULValue を構成します。

ULValue 関数

構文

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

パラメータ

- **val** ULValue として扱われるブール値

説明　bool から ULValue を構成します。

ULValue 関数

構文

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

パラメータ

- **val** ULValue として扱われる short 値

説明　short から ULValue を構成します。

ULValue 関数

構文

ULValue::ULValue(
long *val*
)

パラメータ

- **val** ULValue として扱われる long 値

説明

long から ULValue を構成します。

ULValue 関数

構文

ULValue::ULValue(
int *val*
)

パラメータ

- **val** ULValue として扱われる int 値

説明

int から ULValue を構成します。

ULValue 関数

構文

ULValue::ULValue(
unsigned int *val*
)

パラメータ

- **val** ULValue として扱われる unsigned int 値

説明

unsigned int から ULValue を構成します。

ULValue 関数

構文

ULValue::ULValue(
float *val*
)

パラメータ

- **val** ULValue として扱われる float 値

説明　float から ULValue を構成します。

ULValue 関数

構文

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

パラメータ

- **val** ULValue として扱われる double 値

説明　double から ULValue を構成します。

ULValue 関数

構文

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

パラメータ

- **val** ULValue として扱われる unsigned char 値

説明　unsigned char から ULValue を構成します。

ULValue 関数

構文

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

パラメータ

- **val** ULValue として扱われる unsigned short 値

説明　unsigned short から ULValue を構成します。

ULValue 関数

構文

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

パラメータ

- **val** ULValue として扱われる unsigned long 値

説明

unsigned long から ULValue 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 として扱われる ul_u_big 値

説明

ul_u_big から ULValue 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 として扱われる ul_s_big 値

説明

ul_s_big から ULValue 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 として扱われる ul_binary 値

説明　ul_binary から ULValue を構成します。

ULValue 関数

構文

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

パラメータ

- **val** ULValue として扱われる datetime 値

説明　datetime から ULValue を構成します。

ULValue 関数

構文

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

パラメータ

- **val** ULValue として扱われる文字列へのポインタ

説明　文字列から ULValue を構成します。

ULValue 関数

構文

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

パラメータ

- **val** ULValue 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 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 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 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 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 を構成します。

operator DECL_DATETIME 関数

構文

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

説明

ULValue を datetime にキャストします。

operator bool 関数

構文

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

説明

ULValue を bool にキャストします。

operator double 関数

構文　**ULValue::operator double()**

説明　ULValue を double にキャストします。

operator float 関数

構文　**ULValue::operator float()**

説明　ULValue を float にキャストします。

operator int 関数

構文　**ULValue::operator int()**

説明　ULValue を int にキャストします。

operator long 関数

構文　**ULValue::operator long()**

説明　ULValue を long にキャストします。

operator short 関数

構文　**ULValue::operator short()**

説明　ULValue 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 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 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 を char にキャストします。

operator unsigned int 関数

構文 **ULValue::operator unsigned int()**

説明 ULValue 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 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 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 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 のデストラクタです。

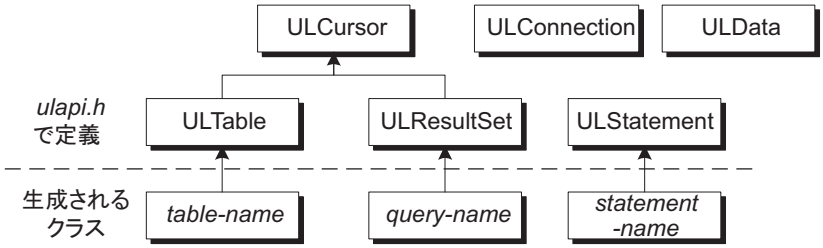
第 14 章

Ultra Light 静的型 C++ API リファレンス

この章の内容　この章では Ultra Light 静的型 C++ API について説明します。

C++ API クラス階層

次の図に、C++ API のクラスを示します。



クラスは、次のヘッダ・ファイルに記述されます。

- **generated-name.hpp**　特定の文またはテーブルのセット用に生成されたインタフェースは、生成された *.hpp* ファイルに定義されます。
- **ulapi.h**　基本クラスは *ulapi.h* に定義されます。このファイルは、SQL Anywhere インストール・ディレクトリの *h* サブディレクトリに格納されます。
- **ulglobal.h**　*ulapi.h* で使用する一部のデータ型やその他の定義について確認するには、*ulglobal.h* を参照します。このファイルは、SQL Anywhere インストール・ディレクトリの *h* サブディレクトリに格納されます。

静的型 C++ API から利用できる関数

C++ API アプリケーションは、クラス階層にない関数を使用することがあります。これらの関数は次のとおりです。

- 「ULEnableFileDB 関数」253 ページ .
- 「ULEnablePalmRecordDB 関数」256 ページ .
- 「ULEnableStrongEncryption 関数」257 ページ .
- 「ULEnableUserAuthentication 関数」258 ページ .
- 「ULRegisterErrorCallback 関数」259 ページ .

- 「ULRegisterSchemaUpgradeObserver 関数」263 ページ .

C++ API 言語要素

Ultra Light API メソッドと変数は、一連の Ultra Light データ型によって記述します。この項ではこれらのデータについて説明します。

Ultra Light データ型

- **an_SQL_code** SQL エラー・コードを保持するデータ型です。
- **ul_char** 文字を表すデータ型です。オペレーティング・システムがユニコードを使用する場合、**ul_char** は 1 文字につき 2 バイトを使用します。シングルバイトの文字セットの場合、**ul_char** は 1 文字につき 1 バイトを使用します。
- **ul_binary** バイナリ情報を 1 バイトで表すデータ型です。
- **ul_column_num** テーブルまたはクエリのカラムを指す番号を保持するデータ型です。テーブルまたはクエリの最初のカラムを指す場合は 1 になります。
- **ul_fetch_offset** ULCursor オブジェクト内の相対的な数値を保持するデータ型です。
- **ul_length** データ型の長さを保持するデータ型です。
- **DECL_DATETIME** 日付と時間の情報を SQLDATETIME 構造体で保持する型です。定義は次のとおりです。

```
typedef struct sqldatetime {
     unsigned short year; /* e.g. 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;
```

DECL_DATETIME は、Embedded SQL プログラミングでも使用されます。C++ API プログラミングでは、DECL_*type* という名前の他の Embedded SQL データ型は必要ありません。

- **UL_NULL**　SQL NULL を表す定数です。

ULConnection クラス

オブジェクト

データベース接続を表します。

説明

ULConnection オブジェクトは、Ultra Light データベース接続を表します。これは、接続のオープンとクローズ、接続がオープンされているかどうかの確認、現在の接続におけるデータベースの同期などを行うメソッドを提供します。

Embedded SQL では、**ULConnection** オブジェクトを開くことが EXEC SQL CONNECT 文に相当します。

Close メソッド

プロトタイプ

bool **Close ( )**

説明

アプリケーションをデータベースから切断し、ULConnection オブジェクトに関連するリソースを解放します。ULConnection オブジェクトを閉じると、アプリケーションは Ultra Light データベースと接続できません。

接続を閉じると、未処理の変更はロールバックされます。

Palm Computing Platform アプリケーションの接続オブジェクトは閉じないようにしてください。アプリケーションの再起動には **Reopen** メソッドを使用します。詳細については、「Reopen メソッド (旧式)」389 ページを参照してください。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例

次のサンプル・コードは、ULConnection オブジェクトを閉じます。

```
conn.Close();
```

参照

「Open メソッド」388 ページ

Commit メソッド

プロトタイプ

bool **Commit( )**

説明

未処理の変更をデータベースにコミットします。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例

次のサンプル・コードは、データベースに値を挿入し、変更をコミットします。

```
productTable.Open( &conn );
 productTable.SetProd_id( 2 );
 productTable.SetPrice( 3000 );
 productTable.SetProd_name( "8' 2x4 Studs x1000" );
 productTable.Insert();
 conn.Commit();
```

参照

「Rollback メソッド」391 ページ

CountUploadRows メソッド

プロトタイプ

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

説明

次回の同期でアップロードする必要のあるロー数を返します。

同期が必要かどうかを判定するときに使用する関数です。

パラメータ

publication-mask　チェック対象のパブリケーションのセット。値 0 はデータベース全体を示します。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
```

パブリケーション・マスクの詳細については、「publication 同期パラメータ」524 ページを参照してください。

threshold カウントするローの最大数を決定する値。呼び出しにかかる時間を制限します。値 0 は制限がないことを示します。値 1 は、同期の必要なローがあるかどうかを判別する場合に使用します。

戻り値 アップロードするローの数

GetCA メソッド

プロトタイプ SQLCA ***GetCA( )**

説明 現在の接続に関連する SQLCA を取り出します。

この機能は、単一のアプリケーションに Embedded SQL と C++ API を組み合わせて使用しているときに役立ちます。

戻り値 SQLCA へのポインタ。

例

```
ULConnection conn;
 conn.Open();
 conn.GetCA();
```

参照 『ASA プログラミング・ガイド』> 「SQLCA (SQL Communication Area)」

GetDatabaseID メソッド

プロトタイプ ul_u_long **ULGetDatabaseID( )**

説明 ULGetDatabaseID は、グローバル・オートインクリメントに使用する現在のデータベースの ID を返します。返される値は、直前の SetDatabaseID で設定された値、または ID が設定されていない場合は UL_INVALID_DATABASE_ID です。

GetLastIdentity メソッド

プロトタイプ ul_u_big **GetLastIdentity( )**

説明

直前に使用した identity の値を返します。この関数は、次の SQL 文と同義です。

```
SELECT @@identity
```

この関数は、グローバル・オートインクリメント・カラムで使うと特に便利です。

戻り値

直前に使用した identity の値

参照

『Mobile Link クライアント』> 「最後に割り当てられた値の割り出し」

『Mobile Link クライアント』> 「デフォルトのグローバル・オートインクリメント・カラムの宣言」

GetLastDownloadTime メソッド

プロトタイプ

bool **GetLastDownloadTime(**
ul_publication_mask *mask*,
DECL_DATETIME **value* **)**

説明

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示します。

パラメータ

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 はデータベース全体を示します。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
```

パブリケーション・マスクの詳細については、「publication 同期パラメータ」524 ページを参照してください。

value　投入する DECL_DATETIME 構造体へのポインタ。

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* が正常に投入されたことを示します。
- **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あることを示します。戻り値が false の場合は、*value* の内容が有効ではないことを示します。

GetSQLCode メソッド

プロトタイプ

an_SQL_code **GetSQLCode( )**

説明

SQLCODE の値を調べてデータベース操作の結果 (成功／失敗) を確認することで、エラー・チェック機能を提供します。SQLCODE は、Adaptive Server Anywhere で使用される標準コードです。

SQLCODE は、他の接続への操作など、後に続く Ultra Light データベース操作によってリセットされます。

戻り値

SQLCODE 値 (整数)

例

次のサンプル・コードは、SQLCODE を書き出します。同期呼び出しが失敗すると、値 -85 が返されます。

```
conn.Synchronize( &synch_info );
 sqlcode = conn.GetSQLCode();
 printf("sqlcode: %d¥n", sqlcode );
```

参照

『ASA エラー・メッセージ』> 「ASA エラー・メッセージ」

GetSynchResult メソッド

プロトタイプ

bool **GetSynchResult(** ul_synch_result * *synch-result* **);**

説明

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。

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 **GetSynchResult** 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 Computing Platform 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。**ULData.PalmLaunch** の呼び出しで設定された SQLCODE 値は、**ULData.PalmLaunch** 操作自体を表します。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、**ULData.PalmLaunch** が正常に終了してから **GetSynchResult** を呼び出します。

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;
```

個々のメンバには次の意味があります。

- **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、『ASA エラー・メッセージ』> 「Adaptive Server Anywhere SQLCODE 順にリストしたエラー・メッセージ」を参照してください。
- **stream_error**　最後の同期の通信ストリーム・エラー・コード。エラー・コードのリストについては、『ASA エラー・メッセージ』> 「Mobile Link 通信エラー・メッセージ」を参照してください。
- **upload_ok**　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。

- **ignored_rows** 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。
- **auth_status** 同期認証ステータス。詳細については、「auth_status パラメータ」511 ページを参照してください。
- **auth_value** Mobile Link 同期サーバが **auth_status** の結果を判断するために使用する値。詳細については、「auth_value 同期パラメータ」512 ページ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。詳細については、「observer 同期パラメータ」520 ページを参照してください。

戻り値

メソッドはブール値を返します。

true 成功を示します。

false 失敗を示します。

参照

「PalmLaunch メソッド [旧式]」417 ページ

GlobalAutoincUsage メソッド

プロトタイプ

ul_u_short **GlobalAutoincUsage()**

説明

利用可能なグローバル・オートインクリメントの値の使用済み比率 (%) を返します。

比率が 100% になったら、SetDatabaseID を使用して、使用中のアプリケーションに新しいグローバル・データベース ID 値を設定してください。

戻り値

利用可能なグローバル・オートインクリメント値の使用率

参照

「SetDatabaseID メソッド」392 ページ

GrantConnectTo メソッド

プロトタイプ　bool **GrantConnectTo(** *userid*, *password* **)**

パラメータ　**userid**　ユーザ ID を保持する文字配列。最大長は 16 文字です。

password　*userid* のパスワードを保持する文字配列。最大長は 16 文字です。

説明　指定されたパスワードを持つユーザ ID に、Ultra Light データベースへのアクセスを許可します。既存のユーザ ID が指定されていれば、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　『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light のユーザ認証」

「ユーザの認証」57 ページ

「RevokeConnectFrom メソッド」390 ページ

InitSynchInfo メソッド

プロトタイプ　an_SQL_code **InitSynchInfo(** ul_synch_info * *synch_info* **)**

説明　同期に使用する *synch_info* 構造体を初期化します。

戻り値　なし

例　次のサンプル・コードは、Ultra Light アプリケーションでデータを同期するときの呼び出しシーケンスで、**InitSynchInfo** メソッドが使用される場所を示しています。

```
auto ul_synch_info synch_info;
 conn.InitSynchInfo( &synch_info );
 conn.Synchronize( &synch_info );
```

参照

「Synchronize メソッド」393 ページ

IsOpen メソッド

プロトタイプ

bool **IsOpen ( )**

説明

ULConnection オブジェクトが現在開いているかどうかを確認します。

戻り値

true (1) ULConnection オブジェクトが開いている場合

false (0) ULConnection オブジェクトが開いていない場合

例

次のサンプル・コードは、接続を開く動作に成功したことを確認します。

```
ULConnection conn;
 conn.Open();
 if( conn.IsOpen() ){
     printf( "Connected to the database.¥n" );
 }
```

参照

「Open メソッド」388 ページ

LastCodeOK メソッド

プロトタイプ

bool **LastCodeOK ( )**

説明

最新の SQLCODE をチェックし、コードが警告または成功を示す場合には **true** を返します。最新の SQLCODE がエラーを示す場合には、**false** を返します。

このメソッドは、操作の成功または失敗を確認するのに便利です。数値の取得には、GetSQLCode を使用します。

SQLCODE は、他の接続への操作など、後に続く Ultra Light データベース操作によってリセットされます。

戻り値

true (1) 前回の SQLCode が 0 または警告の場合

false (0) 前回の SQLCode がエラーの場合

例

次のサンプル・コードは、接続を開く動作に成功したことを確認します。

```
ULConnection conn;
 conn.Open();
 if( conn.LastCodeOK() ){
     printf( "Connected to the database.¥n" );
 };
```

参照

「GetSQLCode メソッド」382 ページ

LastFetchOK メソッド

プロトタイプ

bool **LastFetchOK( )**

説明

最後に行われたローのフェッチについて、成功 (**true**) または失敗 (**false**) を確認するのに便利なメソッドです。

値は、他の接続への操作など、後に続く Ultra Light データベース操作によってリセットされます。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例

次のサンプル・コードは、テーブルの最後のローに移動し、そのローの値をフェッチします。さらに、フェッチが成功したことを確認します。

```
tb.Open( &conn );
 tb.Last();
 tb.GetID( iVal );
 if( tb.LastFetchOK() ){
     ... operations on success...
 }
```

参照

「AfterLast メソッド」397 ページ

「First メソッド」399 ページ

Open メソッド

プロトタイプ

bool **Open (** ULData* *db*,
ul_char* *userid*,
ul_char* *password*,
ul_char* *name* = **SQLNULL)**

bool **Open (** SQLCA * *sqlca*, char * *start-parms* **)**

説明

データベースへの接続を開きます。この呼び出しを成功させるには、**ULData** オブジェクトを開きます。

パラメータ

db 接続が確立された **ULData** オブジェクトへのポインタ。この引数は、通常、接続を確立する前に開かれた **ULData** オブジェクトのアドレスを指します。

userid ユーザ ID の引数。将来使用するために確保しておく予備のプレースホルダです。この引数は無視されます。

ユーザ ID および Ultra Light の詳細については、『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light のユーザ認証」を参照してください。

password パスワード・パラメータ。将来使用するために確保しておく予備のプレースホルダです。この引数は無視されます。

name 接続に使用するオプション名。1 つのアプリケーションから同じデータベースに複数の接続を確立する場合にのみ指定します。

start-parms keyword=value のペアがセミコロンで区切られたリストからなる接続文字列。

指定できるキーワードの詳細については、『Ultra Light データベース・ユーザーズ・ガイド』> 「接続パラメータ」を参照してください。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例

次のサンプル・コードは、Ultra Light データベースへの接続を開きます。

```
ULData db;
 ULConnection conn;

 db.Open();
 conn.Open( &db, "dummy", "dummy" );
```

参照

「Close メソッド」378 ページ

Reopen メソッド (旧式)

プロトタイプ

bool **Reopen ( )**

bool **Reopen(** ULData **db*, ul_char * *name* **= SQLNULL)**

説明

旧式な機能
Reopen メソッドはもはや不要で、使用されなくなりました。代わりに ULConnection::Open を使用します。

このメソッドは Palm Computing Platform でのみ使用できます。この呼び出しを成功させるには、**ULData** オブジェクトを再開します。

Palm アプリケーションを開発するときに、接続オブジェクトを閉じないようにしてください。Ultra Light アプリケーションに切り替えるときには **Reopen** メソッドを呼び出してください。このメソッドにより、アプリケーションで使用するデータベース・オブジェクト用のデータが準備されます。

db　接続が確立された **ULData** オブジェクトへのポインタ。この引数は、通常、接続を再開する前に開かれた **ULData** オブジェクトのアドレスを指します。

name　接続に使用するオプション名。1 つのアプリケーションから同じデータベースに複数の接続を確立する場合にのみ指定します。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例

次のサンプル・コードは、データベース・オブジェクトを再び開き、次に接続オブジェクトを再び開きます。

```
db.Reopen();
 conn.Reopen( &db );
```

参照

「Open メソッド」388 ページ

ResetLastDownloadTime メソッド

プロトタイプ

bool **ResetLastDownloadTime(** ul_publication_mask *publication-mask* **)**

説明

このメソッドは、値を再び投入し、アプリケーションを既知のクリーンな状態に戻すために使用できます。この関数は、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。

パラメータ

publication-mask チェック対象のパブリケーションのセット。値 0 はデータベース全体を示します。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
```

パブリケーション・マスクの詳細については、「publication 同期パラメータ」524 ページを参照してください。

例

次の例は、データベース内の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。

```
db.Reopen();
 conn.ResetLastDownloadTime( UL_SYNC_ALL );
```

参照

「GetLastDownloadTime メソッド」381 ページ

『Mobile Link 管理ガイド』> 「タイムスタンプベースの同期」

RevokeConnectFrom メソッド

プロトタイプ

bool **RevokeConnectFrom(** ul_char * *userid* **)**

説明

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。

パラメータ　**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　『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light のユーザ認証」

「ユーザの認証」57 ページ

「GrantConnectTo メソッド」385 ページ

Rollback メソッド

プロトタイプ　bool **Rollback( )**

説明　未処理の変更をデータベースにロールバックします。

戻り値　成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例　次のサンプル・コードは、値をデータベースに挿入し、その後で変更をロールバックしています。

```
productTable.Open( &conn );
 productTable.SetProd_id( 2 );
 productTable.SetPrice( 3000 );
 productTable.SetProd_name( "8' 2x4 Studs x1000" );
 productTable.Insert();
 conn.Rollback();
```

参照　「Commit メソッド」379 ページ

RollbackPartialDownload メソッド

失敗した同期から変更をロールバックします。

プロトタイプ　bool **RollbackPartialDownload ( )**

説明

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し、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)、RollbackPartialDownload 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

- 『Mobile Link 管理ガイド』> 「失敗したダウンロードの再開」
- 『Mobile Link クライアント』> 「Keep Partial Download 同期パラメータ」
- 『Mobile Link クライアント』> 「Partial Download Retained 同期パラメータ」
- 『Mobile Link クライアント』> 「Resume Partial Download 同期パラメータ」

SetDatabaseID メソッド

プロトタイプ

bool **SetDatabaseID(** ul_u_long *value* **)**

説明

グローバル・オートインクリメントのカラムに使用するデータベース ID の値を設定します。

パラメータ

value グローバル・オートインクリメントの値を生成するために使用する値。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

参照

『ASA データベース管理ガイド』> 「GLOBAL_DATABASE_ID オプション [データベース]」

「GlobalAutoincUsage メソッド」384 ページ

StartSynchronizationDelete メソッド

プロトタイプ

bool **StartSynchronizationDelete( )**

説明

この関数が呼び出されると、すべての削除操作がもう一度同期されます。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

参照

『ASA SQL リファレンス・マニュアル』> 「START SYNCHRONIZATION DELETE 文 [Mobile Link]」

「StopSynchronizationDelete メソッド」393 ページ

StopSynchronizationDelete メソッド

プロトタイプ

bool **StopSynchronizationDelete( )**

説明

この関数が呼び出されると、削除操作が同期されなくなります。領域を節約するために、Ultra Light データベースから古い情報を削除して、統合データベースにはそれを残しておく場合に使用すると便利です。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

参照

『ASA SQL リファレンス・マニュアル』> 「START SYNCHRONIZATION DELETE 文 [Mobile Link]」

「StartSynchronizationDelete メソッド」392 ページ

Synchronize メソッド

プロトタイプ

bool **Synchronize (** ul_synch_info * *synch_info* **)**

説明

Ultra Light データベースを同期します。

synch_info 構造体のメンバについては、「同期パラメータ」508 ページを参照してください。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例

次のサンプル・コード・フラグメントでは、Synchronize メソッドに情報を提供する方法を指定しています。

```
auto ul_synch_info synch_info;
 conn.InitSynchInfo( &synch_info );
 synch_info.user_name = UL_TEXT( "50" );
 synch_info.version = UL_TEXT( "custdb" );
 synch_info.stream = ULSocketStream();
 synch_info.stream_parms =
     UL_TEXT("host=localhost");
 conn.Synchronize( &synch_info );
```

参照

「同期パラメータ」508 ページ

ULCursor クラス

ULCursor クラスには、生成されたテーブル・オブジェクトと生成された結果セットのオブジェクトによって必要とされるメソッドが含まれます。

API 階層での位置付けについては、「C++ API クラス階層」374 ページを参照してください。

データ型列挙

この列挙には、定数として使用できる Ultra Light データ型がリストされています。これに含まれるメンバは次のとおりです。

列挙値	説明
BAD_INDEX	不適切な引数が指定された
S_LONG	符号付き 4 バイト整数
S_SHORT	符号付き 2 バイト整数
LONG	4 バイト整数
SHORT	2 バイト整数
TINY	1 バイト整数
BIT	ビット
TIMESTAMP_STRUCT	struct としてのタイムスタンプ情報
DATE	文字列としての日付と時刻情報
TIME	文字列としての時刻情報
S_BIG	符号付き 8 バイト整数
BIG	8 バイト整数
DOUBLE	倍精度の小数点数
REAL	実数

列挙値	説明
BINARY	指定した長さのバイナリ・データ
TCHAR	指定した長さの文字データ
NUMERIC	精度と桁数が指定された正確な数値データ
MAX_INDEX	予約

GetColumnType メソッドは、この列挙値のいずれかを返します。

参照 「GetColumnType メソッド」401 ページ

SQL データ型列挙

この列挙には、定数として使用できる Ultra Light SQL データ型がリストされています。これに含まれるメンバは次のとおりです。

```
enum {
          SQL_BAD_INDEX,
          SQL_S_LONG,
          SQL_U_LONG,
          SQL_LONG = SQL_U_LONG,
          SQL_S_SHORT,
          SQL_U_SHORT,
          SQL_SHORT = SQL_U_SHORT,
          SQL_S_BIG,
          SQL_U_BIG,
          SQL_BIG = SQL_U_BIG,
          SQL_TINY,
          SQL_BIT,
          SQL_TIMESTAMP,
          SQL_DATE,
          SQL_TIME,
          SQL_DOUBLE,
          SQL_REAL,
          SQL_NUMERIC,
          SQL_BINARY,
```

```
        SQL_CHAR,
        SQL_LONGVARCHAR,
        SQL_LONGBINARY,
        SQL_MAX_INDEX
    };
```

GetColumnSQLType メソッドは、この列挙のいずれかの値を返します。

参照　「GetColumnSQLType メソッド」402 ページ

AfterLast メソッド

プロトタイプ　bool **AfterLast( )**

説明　カーソル位置を、現在のテーブルまたは結果セット内の最後のローの後ろに移動します。

戻り値　成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例　次のサンプル・コードは、現在のローをテーブル **tb** の最後のローにします。

```
tb.AfterLast();
 tb.Previous();
```

参照　「BeforeFirst メソッド」397 ページ

「Last メソッド」404 ページ

BeforeFirst メソッド

プロトタイプ　bool **BeforeFirst( )**

説明　カーソル位置を、現在のテーブルまたは結果セットで最初のローの前に移動します。

戻り値　成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例

次のサンプル・コードは、現在のローをテーブル **tb** の最初のローにします。

```
tb.BeforeFirst();
 tb.Next();
```

参照

「AfterLast メソッド」397 ページ

「First メソッド」399 ページ

Close メソッド

プロトタイプ

bool **Close( )**

説明

アプリケーションの生成オブジェクトと関連するリソースを解放します。このメソッドは、テーブルに関するすべての処理が完了した後で、ULConnection と ULData オブジェクトを閉じる前に呼び出してください。

コミットされていない操作はすべて、**Close()** メソッドが呼び出されたときにロールバックされます。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例

次のサンプル・コードは、ULProduct テーブルに対して生成されたオブジェクトを閉じます。

```
tb.Close();
```

参照

「Open メソッド」441 ページ

Delete メソッド

プロトタイプ

bool **Delete( )**

説明

現在のテーブルまたは結果セットから現在のローを削除します。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。たとえば、複数のテーブルを表す SQL 文でこのメソッドを使用しようとすると、処理は失敗します。

例

次のサンプル・コードは、テーブル **tb** から最後のローを削除します。

```
tb.Open( &conn );
 tb.Last();
 tb.Delete();
```

参照

「Insert メソッド」403 ページ

「Update メソッド」409 ページ

First メソッド

プロトタイプ

bool **First( )**

説明

カーソル位置を、テーブルまたは結果セットの最初のローに移動します。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。たとえば、ローがない場合、メソッドは失敗します。

例

次のサンプル・コードは、テーブル **tb** から最初のローを削除します。

```
tb.Open( &conn );
 tb.First();
 tb.Delete();
```

参照

「BeforeFirst メソッド」397 ページ

「Last メソッド」404 ページ

Get メソッド

プロトタイプ

bool **Get(** ul_column_num *colnum*,
value-declaration,
bool* *isNull* = **UL_NULL)**

value-declaration:
ul_char * *ptr*, ul_length *length*
| p_ul_binary *name* , ul_length *length*
| DECL_DATETIME &*date-value*
| { DECL_BIGINT | DECL_UNSIGNED_BIGINT } &*bigint-value*
| [unsigned] long &*integer-value*
| unsigned char &*char-value*
| double & *double-value*
| float & *float-value*
| [unsigned] short &*short-value*

説明

指定のカラムから値を取得します。

colnum　2 バイト整数。最初のカラムの場合、カラム 1 になります。

value declaration　値の指定に必要な引数。指定方法はデータ型に依存します。文字とバイナリ・データは、呼び出しに指定するバッファの名前と長さに基づいてバッファにマップされます。その他のデータ型の場合は、適切な型の変数へのポインタが必要です。文字データの場合、ターミネータ用に使用する領域を*含む* C 配列の長さを指定するのは、長さパラメータです。

isNULL　カラムの値が NULL の場合、**isNull** は **true** に設定されます。この場合、引数 **value** を指定しても意味がありません。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

参照

「Get 生成メソッド」438 ページ

「Set メソッド」408 ページ

GetColumnCount メソッド

プロトタイプ　int **GetColumnCount( )**

説明　現在のテーブルまたは結果セットのカラム数を返します。

戻り値　カラム数を表す整数

例　次のサンプル・コードは、tb テーブル・オブジェクトを開き、変数 numCol にカラム数を設定します。

```
tb.Open( &conn );
 numCol = tb.GetColumnCount();
```

GetColumnSize メソッド

プロトタイプ　ul_length **GetColumnSize(** ul_column_num *column-index* **)**

説明　指定したカラムの情報を保持するために必要なバイト数を返します。

パラメータ　**column-index**　カラムの番号。テーブルの最初のカラム番号は 1 です。

戻り値　バイト数

例　次のサンプル・コードは、テーブル **tb** の第 3 カラムの情報を保持するために必要なバイト数を取得します。

```
tb.Open( &conn );
 colSize = tb.GetColumnSize( 3 );
```

参照　「GetColumnType メソッド」401 ページ

GetColumnType メソッド

プロトタイプ　int **GetColumnType(** ul_column_num *column-index* **)**

説明　指定したカラムの情報を保持するために必要なデータ型を返します。

パラメータ

column-index カラムの番号。テーブルまたは結果セットの最初のカラム番号は 1 です。

戻り値

Ultra Light データ型列挙のメンバであるカラム型。詳細については、「データ型列挙」395 ページを参照してください。

例

次のサンプル・コードは、テーブル **tb** の第 3 カラムのカラム型を取得します。

```
tb.Open( &conn );
 colType = tb.GetColumnType( 3 );
```

参照

「データ型列挙」395 ページ

「Get 生成メソッド」438 ページ

「GetColumnSQLType メソッド」402 ページ

GetColumnSQLType メソッド

プロトタイプ

int **GetColumnSQLType(** ul_column_num *column-index* **)**

説明

指定したカラムの SQL データ型を返します。

パラメータ

column-index カラムの番号。テーブルまたは結果セットの最初のカラム番号は 1 です。

戻り値

Ultra Light データ型列挙のメンバであるカラム型。詳細については、「データ型列挙」395 ページを参照してください。

例

次のサンプル・コードは、テーブル **tb** の第 3 カラムのカラム型を取得します。

```
tb.Open( &conn );
 colType = tb.GetColumnType( 3 );
```

参照

「データ型列挙」395 ページ

「Get 生成メソッド」438 ページ

「GetColumnType メソッド」401 ページ

GetSQLCode メソッド

ULConnection::GetSQLCode メソッドを呼び出す便宜的なメソッドです。

詳細については、「GetSQLCode メソッド」382 ページを参照してください。

Insert メソッド

プロトタイプ

bool **Insert( )**

説明

直前に実行された **Set** メソッドで指定された値を使って、テーブルにローを挿入します。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例

次のサンプル・コードは、現在の位置に基づいて、テーブルに新しいローを挿入します。

```
productTable.SetProd_id( 2 );
 productTable.SetPrice( 3000 );
 productTable.SetProd_name( "8' 2x4 Studs x1000" );
 productTable.Insert();
```

ローを挿入するときには、テーブル内のカラムごとに 1 つの値を指定します。

挿入後のカーソルの位置と挿入したローの位置の詳細については、「データへのアクセス」55 ページを参照してください。

参照

「Delete メソッド」398 ページ

「Update メソッド」409 ページ

IsOpen メソッド

プロトタイプ

bool **IsOpen ( )**

説明

ULCursor オブジェクトが現在開いているかどうかを確認します。

戻り値

true (1) ULCursor オブジェクトが開いている場合

false (0) ULCursor オブジェクトが開いていない場合

参照

「Open メソッド」405 ページ

Last メソッド

プロトタイプ

bool **Last( )**

説明

カーソル位置を、テーブルまたは結果セットの最後のローに移動します。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例

次のサンプル・コードは、カーソル位置をテーブル内の最後のローの後ろに移動します。

```
tb.Open( &conn );
 tb.Last();
 tb.Next();
```

参照

「AfterLast メソッド」397 ページ

「First メソッド」399 ページ

LastCodeOK メソッド

ULConnection::LastCodeOK メソッドを呼び出す便宜的なメソッドです。

詳細については、「LastCodeOK メソッド」386 ページを参照してください。

LastFetchOK メソッド

ULConnection::LastFetchOK メソッドを呼び出す便宜的なメソッドです。

詳細については、「LastFetchOK メソッド」405 ページを参照してください。

Next メソッド

プロトタイプ

bool **Next( )**

説明

カーソル位置を、テーブルまたは結果セットの次のローに移動します。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例

次のサンプル・コードは、カーソル位置をテーブル内の最初のローに移動します。

```
tb.Open( &conn );
 tb.BeforeFirst();
 tb.Next();
```

参照

「Previous メソッド」406 ページ

「Relative メソッド」407 ページ

Open メソッド

プロトタイプ

bool **Open(** ULConnection * *conn* **)**

説明

指定の接続にカーソルを開きます。オブジェクトがパラメータ付きの結果セットである場合には、結果セットを開く前にパラメータを設定します。

ULCursor の **ULTable** サブクラスの **Open** を使用している場合は、**ULTable** オブジェクトで同時に 2 つの接続を開かないでください。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例

次のサンプル・コードは、(カーソル・クラスを拡張する) 結果セット・オブジェクトを開き、カーソル位置を最初のローに移動します。

```
rs.Open( &conn );
 rs.BeforeFirst();
 rs.Next();
```

参照

「Close メソッド」398 ページ

「Open メソッド」433 ページ

Previous メソッド

プロトタイプ

bool **Previous( )**

説明

カーソル位置を、現在のテーブルまたは結果セット内の前のローに移動します。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例

次のサンプル・コードは、カーソル位置をテーブル内の最後のローに移動します。

```
tb.Open( &conn );
 tb.AfterLast();
 tb.Previous();
```

参照

「Next メソッド」405 ページ

「Relative メソッド」407 ページ

Relative メソッド

プロトタイプ

bool **Relative(** ul_fetch_offset *offset* **)**

説明

カーソル位置を、現在の位置を基準とした相対位置に移動します。指定した移動先にローがない場合には、false が返されます。その場合のカーソル位置は *offset* が正であるときには **AfterLast()**、負であるときには **BeforeFirst()** になります。

offset　移動するローの数です。負の値を指定すると、後方に移動します。

戻り値

true (1) ローが存在する場合

false (0) ローが存在しない場合

参照

「Next メソッド」405 ページ

「Previous メソッド」406 ページ

Reopen メソッド

プロトタイプ

bool **Reopen(** ULConnection **conn* **)**

説明

このメソッドは Palm Computing Platform でのみ使用できます。この呼び出しを成功させるには、**ULData** と **ULConnection** オブジェクトがあらかじめ開いている必要があります。

Palm アプリケーションの開発中にカーソル位置を維持する必要がある場合は、結果セットのオブジェクトを閉じないでください。Ultra Light アプリケーションに戻るときには **Reopen** メソッドを呼び出してください。

ULTable オブジェクトが **ULCursor** クラスから継承されていても、テーブル・オブジェクトでは **Reopen** を使用しないでください。既存の Palm アプリケーションではそのオブジェクトを閉じて、再入力時に開いてください。**ULTable** オブジェクトでは、カーソル位置が維持されません。

パラメータ

conn カーソルを定義する **ULConnection** オブジェクトへのポインタ。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例

次のサンプル・コードは、データベース・オブジェクトを再開し、さらに接続オブジェクト、結果セット・オブジェクトの順に再開します。

```
db.Reopen();
 conn.Reopen( &db );
 rs.Reopen( &conn );
```

参照

「Open メソッド」388 ページ

Set メソッド

プロトタイプ

bool **Set(** ul_column_num *colnum*, *value* **)**

value:
p_ul_binary *buffer-name*, ul_length *buffer-length*
| ul_char * *buffer-name*, ul_length *buffer-length* = 0
| DECL_DATETIME *date-value*
| DECL_UNSIGNED_BIGINT *bigint-value*
| unsigned char *char-value*
| double *double-value*
| float *float-value*
| [unsigned] long *long-value*
| [unsigned] short *short-value*

説明

現在のローについて、指定されたカラムに値を設定します。

colnum 2 バイト整数。最初のカラムの場合、カラム 1 になります。

value　文字とバイナリ・データの場合、バッファの名前と長さを指定します。その他のデータ型の場合は、適切な型の値を指定する必要があります。カラムに対して不適切なデータ型を指定すると、このメソッドは失敗します。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

参照

「Get メソッド」400 ページ

SetColumnNull メソッド

プロトタイプ

int **SetColumnNull(** ul_column_num *column-index* **)**

説明

カラムに SQL NULL を設定します。データは、Insert 文または Update 文を実行するまでは、実際には変更されません。また、その変更がコミットされないかぎり、永続化されません。

パラメータ

column-index　カラムの番号。テーブルの最初のカラム番号は 1 です。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

参照

「SetNull<Column> 生成メソッド」442 ページ

Update メソッド

プロトタイプ

bool **Update( )**

説明

直前に実行された **Set** メソッドで指定された値を使って、テーブルのローを更新します。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例

次のサンプル・コードは、**productTable** オブジェクトの現在のローに新規価格を設定し、Ultra Light データベース上の該当するローを更新します。

```
productTable.SetPrice( 400 );
 productTable.Update();
```

参照

「Delete メソッド」398 ページ

「Insert メソッド」403 ページ

ULData クラス

オブジェクト

Ultra Light データベースを表します。

プロトタイプ

ULData db;
　db.Open**()**;

説明

ULData クラスは、アプリケーションに対する Ultra Light データベースを表します。これにより、データベースを開くメソッドと閉じるメソッド、データベースがオープンされているどうかを確認するメソッドが提供されます。

データベースは、接続前または他の操作の実行前に開きます。また、データベース上でのすべての操作が終了したら、データベースを閉じてからアプリケーションを終了します。

マルチスレッド・アプリケーションでは、各スレッドが自身の **ULData** を作成する必要があります。**ULData** オブジェクトも、そこから継承された他のオブジェクト (**ULConnection** などのクラス) も、スレッド間で共有することはできません。

Embedded SQL では、**ULData** オブジェクトを開くことは **db_init** の呼び出しに相当します。

API 階層での位置付けについては、「C++ API クラス階層」374 ページを参照してください。

例

次のサンプル・コードは、ULData オブジェクトを宣言して開きます。

```
ULData db;
 db.Open();
```

Close メソッド

プロトタイプ

bool **Close ( )**

説明

オブジェクトに関連付けられたリソースを解放してから、アプリケーションを終了します。ULData オブジェクトを閉じると、接続を再開しないかぎり、データベース上で C++ API を使って操作を実行できません。

Palm Computing Platform
ULData.Close を呼び出す前に、ULSetSynchInfo を使用してアプリケーションの状態を保存します。アプリケーションが再起動された時点で、**Open** メソッドを使用します。詳細については、「Open メソッド」414 ページおよび「ULSetSynchInfo 関数」476 ページを参照してください。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例

次のサンプル・コードは、ULData オブジェクトを閉じます。

```
db.Close();
```

参照

「Open メソッド」414 ページ

Drop メソッド

プロトタイプ

bool **Drop (** SQLCA * *sqlca*, ul_char * *store-parms*)

説明

Ultra Light データベース・ファイルを削除します。

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。使用する場合は注意してください。

この関数は、データベース接続が開いている間は呼び出さないでください。この関数は、データベースを閉じた後、またはデータベースを開く前にだけ呼び出します (C++ API)。

Palm OS では、この関数は、データベースに接続していない場合のみ (ただし、**ULEnable** 関数が呼び出された直後) 呼び出します。

パラメータ

sqlca　SQLCA へのポインタ。

store-parms　接続パラメータの文字列。削除するファイル名を **file_name=***file.udb* という形式のキーワードと値の組み合わせで含みます。この引数として UL_STORE_PARMS マクロを使用すると便利な場合があります。UL_NULL の値は、デフォルトのデータベース・ファイル名を削除します。

詳細については、「UL_STORE_PARMS マクロ」270 ページを参照してください。

戻り値

- **true**　データベース・ファイルが正常に削除されたことを示します。
- **false**　GetSQLCode を使用して詳細なエラー情報を取得できます。よくある失敗の理由は、不正なファイル名が提供されたか、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場合です。

Initialize メソッド

プロトタイプ

bool **Initialize (** SQLCA * *ca* **)**

説明

データベースとの通信に備えて、Ultra Light を初期化します。このメソッドによってデータベースが起動されることはありません。通常、データベースは接続時 (ULConnection::Open) にアプリケーションによって起動されます。

この関数は、ULConnection::Open を含めて、ほかのすべてのデータ管理関数の前に使用します。

パラメータ

Open( )　ほとんどの Ultra Light アプリケーションで使用できるプロトタイプ。データベースを開くときには、UL_STORE_PARMS マクロを使って定義した永続ストレージ・パラメータが使用されます。

Open(SQLCA* ca)　アプリケーションが Embedded SQL と C++ API を併用しているときや、SQLCA がそのアプリケーションで有効になっているときに使用するプロトタイプ。このプロトタイプによって、C++ API を使用して同じデータにアクセスできます。

IsOpen メソッド

プロトタイプ

bool **IsOpen ( )**

説明

ULData オブジェクトが現在開いているかどうかを調べます。

戻り値

true (1) ULData オブジェクトが開いている場合

false (0) ULData オブジェクトが開いていない場合

例

次のサンプル・コードは、ULData オブジェクトを宣言して開きます。さらに、Open メソッドの処理が成功したことを確認します。

```
ULData db;
 db.Open();
 if( db.IsOpen() ){
     printf( "The ULData object is open¥n" );
 }
```

参照

「Open メソッド」414 ページ

Open メソッド

プロトタイプ

bool **Open ( )**

bool **Open(** SQLCA* *ca* **)**

bool **Open(** ul_char* *parms* **)**

bool **Open(** SQLCA* *ca* , ul_char* *parms* **)**

説明

データベースを操作できるように、アプリケーションを準備します。C++ API を使用するデータベース上の操作を実行する前に、ULData オブジェクトを開きます。このルールには、次に示す例外があります。

- Palm Computing Platform では、**ULData.Open** の前に **ULData.PalmLaunch** メソッドが呼び出されます。このライブラリがプログラムのために必要とするリソースは、この呼び出しで割り付けられて初期化されます。

Palm Computing Platform では、**ULData.PalmLaunch** から LAUNCH_SUCCESS_FIRST が返されるたびに、**ULData.Open** を呼び出します。詳細については、「PalmLaunch メソッド [旧式]」417 ページ を参照してください。

- データベース記憶領域を設定する関数を呼び出すことができます。この種の関数の名前は、**ULEnable** で始まります。

  データベースを開くときに永続ストレージ・パラメータを指定して、キャッシュ、暗号化、データベース・ファイル名を設定する特殊な使い方もできます。これらのパラメータの詳細については、『Ultra Light データベース・ユーザーズ・ガイド』>「Ultra Light データベース・プロパティの設定」を参照してください。

マルチスレッド・アプリケーションでは、各スレッドが自身の **ULData** オブジェクトを開く必要があります。**ULData** オブジェクトも、そこから継承された他のオブジェクト (**ULConnection** などのクラス) も、スレッド間で共有することはできません。

パラメータ

Open( )　ほとんどの Ultra Light アプリケーションで使用できるプロトタイプ。データベースを開くときには、UL_STORE_PARMS マクロを使って定義した永続ストレージ・パラメータが使用されます。

Open(SQLCA* ca)　アプリケーションが Embedded SQL と C++ API を併用しているときや、SQLCA がそのアプリケーションで有効になっているときに使用するプロトタイプ。このプロトタイプによって、C++ API を使用して同じデータにアクセスできます。

Open(ul_char* parms)　永続ストレージ・パラメータは、UL_STORE_PARMS マクロを使用して指定できます。このプロトタイプには、永続ストレージ・パラメータを指定する代替方法があります。この文字列では、割り当てをセミコロンで区切って指定します。形式は、*parameter=value* です。

Open(SQLCA *ca, ul_char* parms)　SQLCA と永続ストレージ・パラメータの両方を指定する呼び出し。

永続ストレージ・パラメータの詳細については、「UL_STORE_PARMS マクロ」270 ページを参照してください。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例

次のサンプル・コードは、ULData オブジェクトを宣言して開きます。

```
ULData db;
 db.Open();
```

参照

「Close メソッド」411 ページ

『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light データベース・プロパティの設定」

「UL_STORE_PARMS マクロ」270 ページ

PalmExit メソッド [旧式]

プロトタイプ

bool **PalmExit(** SQLCA **ca* **)**

bool **PalmExit(** ul_synch_info * *synch_info* **)**

説明

旧式な機能
この関数はもはや不要です。代わりに ULData::Close を使用します。

アプリケーションを閉じる直前にこのメソッドを呼び出し、アプリケーションの状態を保存します。

HotSync または Scout Sync 同期を使用するアプリケーションでは、このメソッドを使ってアップロード・ストリームの書き込みを行います。ユーザが HotSync または Scout Sync を使用して自分の Palm デバイスと PC のデータを同期した場合、アップロード・ストリームはそれぞれ Mobile Link HotSync コンジットまたは Mobile Link Scout コンジットによって読み込まれます。

Mobile Link HotSync コンジットは、TCP/IP または HTTP ストリームを介して Mobile Link 同期サーバと同期します。同期を調整するには、**synch_info.stream_parms** で一連のストリーム・パラメータまたはネットワーク・プロトコルのオプションを指定します。この他、*ClientParms* レジストリ・エントリを使用して、ストリームと stream

パラメータを指定する方法もあります。*ClientParms* レジストリ・エントリが存在しない場合は、デフォルト設定の `{stream=tcpip;host=localhost}` が使用されます。

パラメータ

sqlca　SQLCA へのポインタ。この引数は、アプリケーションで C++ API と Embedded SQL を併用しており、デフォルト以外の SQLCA を使用する場合にのみ指定します。

synch_info　同期構造体。

TCP/IP または HTTP 同期を使用している場合は、ul_synch_info 構造体の代わりに UL_NULL を指定します。これらのストリームを使用している場合は、**ULSynchronize** の呼び出しで代わりに同期情報が指定されます。

HotSync または Scout Sync 同期を使用している場合は、同期構造体を指定します。**stream** パラメータの値は無視されます。UL_NULL も指定できます。

synch_info 構造体のメンバについては、「同期パラメータ」508 ページを参照してください。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

PalmLaunch メソッド [旧式]

プロトタイプ

UL_PALM_LAUNCH_RET **PalmLaunch( )**;

UL_PALM_LAUNCH_RET **PalmLaunch(** ul_synch_info * *synch_info* **)**;

UL_PALM_LAUNCH_RET **PalmLaunch(** SQLCA* *ca* **)**;

UL_PALM_LAUNCH_RET **PalmLaunch(** SQLCA* *ca* ,
ul_synch_info * *synch_info* **)**;

```
typedef enum {
    LAUNCH_SUCCESS_FIRST,
    LAUNCH_SUCCESS,
    LAUNCH_FAIL
    } UL_PALM_LAUNCH_RET;
```

説明

旧式な機能
この関数はもはや不要です。代わりに ULData::Initialize を使用します。

この関数は、アプリケーションがアクティブになったときにアプリケーションの状態をリストアします。HotSync または Scout Sync 同期を使用しているアプリケーションでは、追加のタスクとして Mobile Link HotSync コンジットまたは Mobile Link Scout コンジットによって準備されたダウンロード・ストリームを処理します。

TCP/IP または HTTP 同期を使用している場合、**ul_synch_info** 同期構造体の stream パラメータに NULL 値を指定します。この情報は、**ULConnection.Synchronize** メソッドによって呼び出される同期構造体の代わりに指定します。

パラメータ

ca　SQLCA へのポインタ。この引数は、アプリケーションで C++ API と Embedded SQL を併用しており、デフォルト以外の SQLCA を使用する場合にのみ指定します。

synch_info　同期構造体。この構造体のメンバについては、「同期パラメータ」508 ページを参照してください。

TCP/IP または HTTP 同期を使用している場合、**stream** パラメータに NULL 値を指定します。

戻り値

UL_PALM_LAUNCH_RET 列挙のメンバが返されます。戻り値には次の意味があります。

- **LAUNCH_SUCCESS_FIRST**　アプリケーションが初めて正常に起動したときと、それ以降に Ultra Light データベースの内部状態を確立し直す必要があるときに、この値が返されます。通常、データベースの状態を確立し直す必要があるのは、重大な失敗の後だけです。

LAUNCH_SUCCESS_FIRST が返される場合は、**ULData** オブジェクトを開いてください。

- **LAUNCH_SUCCESS**　Palm ユーザが他のアプリケーションを使用し、その後アプリケーションが正常に起動すると、この値が返されます。
- **LAUNCH_FAIL**　起動に失敗したときに、この値が返されます。

例

次のサンプル・コードは、一般的な C++ API を示しています。

```
ULData db;
 ULEnablePalmRecordDB( & sqlca );
 switch( db.PalmLaunch( &synch_info ) ){
 case LAUNCH_SUCCESS_FIRST:
     if( !db.Open() ){
         // initialization failed: add error handling
here
         break;
     }
     // fall through
 case LAUNCH_SUCCESS:
     db.Reopen();
     // do work here
     break;
 case LAUNCH_FAIL:
     // error
     break;
 }
```

Reopen メソッド [旧式]

プロトタイプ

bool **Reopen ( )**

bool **Reopen(** SQLCA* *ca* **)**

説明

旧式な機能
この関数はもはや不要です。代わりに ULData::Open または ULData::Initialize を使用します。

このメソッドは Palm Computing Platform でのみ使用できます。

Palm アプリケーションを開発するときに、データベース・オブジェクトを閉じないようにしてください。Ultra Light アプリケーションに切り替えるときには **Reopen** メソッドを呼び出してください。このメソッドにより、アプリケーションで使用するデータベース・オブジェクト用のデータが準備されます。

パラメータ

Open( ) アプリケーションで C++ API と Embedded SQL のいずれも使用しない場合は、引数は不要です。

Open(SQLCA* ca) アプリケーションで Embedded SQL を使用する場合や、デフォルト以外の SQLCA を使用する場合にも、このメソッドによって C++ API を使って同じデータにアクセスできます。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例

次のサンプル・コードは、データベース・オブジェクトと接続オブジェクトを再開します。

```
db.Reopen();
 conn.Reopen( &db );
```

参照

「Open メソッド」414 ページ

StartDatabase メソッド

プロトタイプ

bool **StartDatabase (** ul_char const * *parms* **);**

説明

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。この関数は、静的型 C++ API 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。「Ultra Light C/C++ インタフェースの結合」132 ページを参照してください。

StartDatabase を呼び出す前に、ULData::Initialize() を呼び出してください。StartDatabase メソッドを使用してデータベースを起動する場合、UL_STORE_PARMS 内の情報は無視されます。

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、戻り値は true です。エラー情報は SQLCA に返されます。

パラメータ

- **parms** 起動するデータベースを指定する接続文字列。通常、データベース・ファイルのみが含まれます。

  ```
  DBF=mydatabase.udb
  ```

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

StopDatabase メソッド

プロトタイプ

bool **StopDatabase (** ul_char const * *parms* **);**

説明

データベースを停止します。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、静的型 C++ API 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。「Ultra Light C/C++ インタフェースの結合」132 ページを参照してください。

StartDatabase を呼び出した後に、ULData::Close() を呼び出してください。

データベースが正しく停止した場合、戻り値は true です。エラー情報は SQLCA に返されます。

パラメータ

- **parms** 停止するデータベースを指定する接続文字列。通常、データベース・ファイルのみが含まれます。

  ```
  DBF=mydatabase.udb
  ```

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

ULResultSet クラス

ULResultSet クラスは、**ULCursor** クラスを拡張し、生成されたすべての結果セットに必要なメソッドを提供します。

詳細については、「ULCursor クラス」395 ページおよび「生成された結果セットのクラス」432 ページを参照してください。

API 階層での位置付けについては、「C++ API クラス階層」374 ページを参照してください。

SetParameter メソッド

プロトタイプ

virtual bool **SetParameter(** int *argnum*, *value-reference* **)**

value-reference:
[unsigned] long & *value*
| p_ul_binary *value*
| unsigned char & *value*
| ul_char * *value*
| double & *value*
| float & *value*
| [unsigned] short & *value*
| DECL_DATETIME *value*
| DECL_BIGINT *value*
| DECL_UNSIGNED_BIGINT *value*

説明

次のクエリは、パラメータを使って結果セットを定義します。

```
SELECT id
 FROM mytable
 WHERE id < ?
```

このクエリに対応する C++ API で定義される結果セットのオブジェクトは、パラメータを持ちます。パラメータの値を設定してから、生成された結果オブジェクトを開いてください。

パラメータ

argnum　設定する引数の識別子。最初の引数は 1、2 番目の引数は 2、のようになります。

value-reference　パラメータ値の参照。上記のリストは、考えられるデータ型を示したものです。このパラメータはポインタとして渡されるため、使用されるまで有効になっています。使用されるまで開放しないようにします。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。不適切なデータ型のパラメータを指定すると、このメソッドは失敗します。

参照

「Open メソッド」433 ページ

ULTable クラス

ULTable クラスは、**ULCursor** クラスを拡張し、生成されたすべてのテーブル・オブジェクトに必要なメソッドを提供します。

同時に **ULTable** オブジェクトへ複数の接続はできません。

API 階層での位置付けについては、「C++ API クラス階層」374 ページを参照してください。

DeleteAllRows メソッド

プロトタイプ

ul_ret_void **DeleteAllRows( )**

説明

テーブル内のすべてのローを削除する関数です。

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ほうが便利なことがあります。**ULConnection::StopSynchronizationDelete** メソッドを使用すると、統合データベースからは削除しないで Ultra Light データベースからローを削除できます。

参照

「StartSynchronizationDelete メソッド」392 ページ

「StopSynchronizationDelete メソッド」393 ページ

Find メソッド

FindNext メソッドと同義です。

「FindNext メソッド」426 ページを参照してください。

FindFirst メソッド

プロトタイプ

bool **FindFirst(** ul_column_num *ncols* **)**

説明

テーブルを先頭から順方向に移動しながら、現在のインデックスの値かそのセットに完全に一致するローを検索します。

現在のインデックスは、テーブルのソート順の指定に使用されているインデックスです。このソート順は、生成されたテーブルの Open メソッドがアプリケーションによって呼び出されたときに指定されます。デフォルトのインデックスはプライマリ・キーです。詳細については、「Open メソッド」441 ページを参照してください。

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。処理が失敗すると、カーソル位置は **AfterLast()** となります。

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。たとえば、3 つのカラムのインデックスがあり、最初のカラムにのみ基づいて一致する値を検索する場合は、最初のカラムに値を設定し、*ncols* 値 1 を指定します。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

参照

「FindLast メソッド」425 ページ

「FindNext メソッド」426 ページ

「FindPrevious メソッド」427 ページ

「LookupBackward メソッド」429 ページ

「LookupForward メソッド」430 ページ

FindLast メソッド

プロトタイプ

bool **FindLast(** ul_column_num *ncols* **)**

説明

テーブルを最後から逆方向に移動しながら、現在のインデックスの値またはそのセットに一致する最初のローを検索します。

現在のインデックスは、テーブルのソート順の指定に使用されているインデックスです。このソート順は、生成されたテーブルの Open メソッドがアプリケーションによって呼び出されたときに指定されます。デフォルトのインデックスはプライマリ・キーです。詳細については、「Open メソッド」441 ページを参照してください。

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

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。たとえば、3 つのカラムのインデックスがあり、最初のカラムにのみ基づいて一致する値を検索する場合は、最初のカラムに値を設定し、*ncols* 値 1 を指定します。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

参照

「FindFirst メソッド」424 ページ

「FindNext メソッド」426 ページ

「FindPrevious メソッド」427 ページ

「LookupBackward メソッド」429 ページ

「LookupForward メソッド」430 ページ

FindNext メソッド

プロトタイプ

bool **FindNext(** ul_column_num *ncols* **)**

説明

現在の位置からテーブルを順方向に移動しながら、現在のインデックスの値かそのセットに完全に一致するローを検索します。

現在のインデックスは、テーブルのソート順の指定に使用されているインデックスです。このソート順は、生成されたテーブルの Open メソッドがアプリケーションによって呼び出されたときに指定されます。デフォルトのインデックスはプライマリ・キーです。詳細については、「Open メソッド」441 ページを参照してください。

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。処理が失敗すると、カーソル位置は **AfterLast()** となります。

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。たとえば、3 つのカラムのインデックスがあり、最初のカラムにのみ基づいて一致する値を検索する場合は、最初のカラムに値を設定し、*ncols* 値 1 を指定します。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

参照

「FindFirst メソッド」424 ページ

「FindLast メソッド」425 ページ

「FindPrevious メソッド」427 ページ

「LookupBackward メソッド」429 ページ

「LookupForward メソッド」430 ページ

FindPrevious メソッド

プロトタイプ

bool **FindPrevious(** ul_column_num *ncols* **)**

説明

現在の位置からテーブルを逆方向に移動しながら、現在のインデックスの値かそのセットに完全に一致するローを検索します。

現在のインデックスは、テーブルのソート順の指定に使用されているインデックスです。このソート順は、生成されたテーブルの Open メソッドがアプリケーションによって呼び出されたときに指定されます。デフォルトのインデックスはプライマリ・キーです。詳細については、「Open メソッド」441 ページを参照してください。

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

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。たとえば、3 つのカラムのインデックスがあり、最初のカラムにのみ基づいて一致する値を検索する場合は、最初のカラムに値を設定し、*ncols* 値 1 を指定します。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

参照

「FindFirst メソッド」424 ページ

「FindLast メソッド」425 ページ

「FindNext メソッド」426 ページ

「LookupBackward メソッド」429 ページ

「LookupForward メソッド」430 ページ

Lookup メソッド

LookupForward メソッドと同じです。

「LookupForward メソッド」430 ページを参照してください。

GetRowCount メソッド

プロトタイプ

ul_ul_long **GetRowCount( )**

説明

テーブルのロー数を返す関数です。

古いローを削除して領域を節約するタイミングを決定するときに使用します。**ULConnection::StartSynchronizationDelete** メソッドを使用すると、統合データベースからは削除しないで Ultra Light データベースから古いローを削除できます。

戻り値

テーブル内のロー数

参照

「StartSynchronizationDelete メソッド」392 ページ

「StopSynchronizationDelete メソッド」393 ページ

LookupBackward メソッド

プロトタイプ

bool **LookupBackward(** ul_column_num *ncols* **)**

説明

テーブルを最後から逆方向に移動しながら、現在のインデックスの値またはそのセットに一致する最初のローか、それより小さい値の最初のローを検索します。

現在のインデックスは、テーブルのソート順の指定に使用されているインデックスです。このソート順は、生成されたテーブルの Open メソッドがアプリケーションによって呼び出されたときに指定されます。デフォルトのインデックスはプライマリ・キーです。詳細については、「Open メソッド」441 ページを参照してください。

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最初のローか、それより少ない値の最初のローで停止します。検索が失敗した場合 (つまり、検索中の値より小さいローがない場合)、カーソル位置は **BeforeFirst()** です。

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。たとえば、3 つのカラムのインデックスがあり、最初のカラムにのみ基づいて一致する値を検索する場合は、最初のカラムに値を設定し、*ncols* 値 1 を指定します。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

参照

「FindFirst メソッド」424 ページ

「FindLast メソッド」425 ページ

「FindNext メソッド」426 ページ

「FindPrevious メソッド」427 ページ

「LookupForward メソッド」430 ページ

LookupForward メソッド

プロトタイプ

bool **LookupForward(** ul_column_num *ncols* **)**

説明

テーブルを最初から順方向に移動しながら、現在のインデックスの値またはそのセットに一致するか、その値より大きい値を持つ最初のローを検索します。

現在のインデックスは、テーブルのソート順の指定に使用されているインデックスです。このソート順は、生成されたテーブルの Open メソッドがアプリケーションによって呼び出されたときに指定されます。デフォルトのインデックスはプライマリ・キーです。詳細については、「Open メソッド」441 ページを参照してください。

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、それより大きい値の最初のローで停止します。検索が失敗した場合 (つまり、検索中の値より大きいローがない場合)、カーソル位置は **AfterLast()** です。

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。たとえば、3 つのカラムのインデックスがあり、最初のカラムにのみ基づいて一致する値を検索する場合は、最初のカラムに値を設定し、*ncols* 値 1 を指定します。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

参照

「FindFirst メソッド」424 ページ

「FindLast メソッド」425 ページ

「FindNext メソッド」426 ページ

「FindPrevious メソッド」427 ページ

「LookupBackward メソッド」429 ページ

生成された結果セットのクラス

オブジェクト

生成された結果セットのクラスは、アプリケーションに対するクエリ結果セットを表します。クラスの名前は、データベースに追加されたときに指定された文の名前に基づいて、Ultra Light ジェネレータによって生成されます。

プロトタイプ

生成された結果セットのオブジェクトを作成するには、宣言で生成した名前を使用します。

result-set rs;
    rs.**Open()**;

result-set: generated name

説明

Ultra Light ジェネレータは、結果セットを返す Ultra Light プロジェクトに指定された文ごとに、クラスを定義します。このクラスは **ULCursor** のメソッドを継承します。

API 階層での位置付けについては、「C++ API クラス階層」374 ページを参照してください。

参照

「ULCursor クラス」395 ページ

『Ultra Light データベース・ユーザーズ・ガイド』> 「ul_add_statement システム・プロシージャ」

Get<Column> 生成メソッド

プロトタイプ

bool **Get***column-name***(** type* *variable*, [ul_length* *length*,]
    bool* *isNull* = **UL_NULL)**

説明

column-name の値を取得します。データ型は、カラムのデータ型に依存します。

パラメータ

column-name　カラムの名前。

variable　カラムに対する、適切なデータ型の変数。このデータ型は、**GetColumnType** を使って取得できます。

length　可変長データに使用。文字データの場合、ターミネータ用に使用する領域を含む C 配列の長さを指定するのは、長さパラメータです。

isNull　NULL 値の場合、この引数は **true** になります。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

参照

「Set<Column> 生成メソッド」434 ページ

Open メソッド

プロトタイプ

bool **Open(** ULConnection * *conn*,
datatype *value*, … **)**

説明

Ultra Light ジェネレータは、結果セットを返す Ultra Light プロジェクトに指定された文ごとに、クラスを定義します。このクラスは **ULCursor** のメソッドを継承します。

結果セットにおけるプレースホルダごとに、値を指定します。

パラメータ

conn　結果セットを開く接続。

value　結果セット内のプレースホルダの値。

例

次のクエリには単一のプレースホルダが含まれます。

```
select prod_id, price, prod_name
 from "DBA".ulproduct
 where price < ?
```

ジェネレータは、(他のオブジェクトに加えて) 指定のオブジェクトに対して次のメソッドを書き出します。

```
bool Open( ULConnection* conn,
     long Price     );
 bool Open( ULConnection* conn );
 bool SetParameter( int index, long &value );
```

参照 「SetParameter メソッド」422 ページ

Set<Column> 生成メソッド

プロトタイプ bool **Set***column-name***( )**

説明 現在のカーソル位置の値を設定します。ローのデータは、Insert または Update 文が実行されるまで実際には変更されません。また、その変更がコミットされないかぎり、永続化されません。

パラメータ **column-name** リファレンス・データベースに格納されているカラム名を元に生成された名前。

戻り値 成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

参照 「Get<Column> 生成メソッド」432 ページ

「SetNull<Column> 生成メソッド」434 ページ

SetNull<Column> 生成メソッド

プロトタイプ bool **SetNull***column-name***( )**

説明 カラムに SQL NULL を設定します。データは、Insert 文または Update 文を実行するまでは、実際には変更されません。また、その変更がコミットされないかぎり、永続化されません。

パラメータ **column-name** リファレンス・データベースに格納されているカラム名を元に生成された名前。

戻り値 成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

参照 「Set<Column> 生成メソッド」434 ページ

生成文のクラス

Ultra Light ジェネレータは、INSERT、UPDATE、DELETE など、結果セットを返さない SQL 文に対して、生成文のクラスを定義します。クラスの名前には、リファレンス・データベースに文を追加したストアド・プロシージャ呼び出し ul_add_statement で指定した名前と同じ名前が提供されます。

生成文のクラスは、**ULStatement** クラスを継承します。このクラスは、独自のメソッドを持ちません。

API 階層での位置付けについては、「C++ API クラス階層」374 ページを参照してください。

Execute メソッド

プロトタイプ

bool **Execute(** ULConnection* *conn*,
[*datatype column-name*,…] **)**

説明

結果セットを返さない、指定の文を実行します。実行された変更は、コミットされないかぎり永続化されません。

ul_add_statement を使用して文を定義する場合は、その値に対するプレースホルダを指定してランタイムで提供します。生成されたプロトタイプは、値ごとにデータ型と名前を持ちます。

パラメータのセットを省略した場合は、SetNull または SetParameter を使って各値を設定してから、Execute メソッドを呼び出してください。

パラメータ

conn　文の実行に使う接続。

datatype　Ultra Light データ型列挙のメンバ。

column-name　カラムの名前。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

参照

『Ultra Light データベース・ユーザーズ・ガイド』> 「ul_add_statement システム・プロシージャ」

「SetNull メソッド」436 ページ

「SetParameter メソッド」436 ページ

SetNull メソッド

プロトタイプ

bool **SetNull(** ULConnection* *conn*,
ul_column_num *column-index* **)**

説明

指定したカラムを NULL に設定します。

パラメータ

conn 文の実行に使う接続。

column-index カラムの番号。テーブルの最初のカラム番号は 1 です。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

SetParameter メソッド

プロトタイプ

virtual bool **SetParameter(** int *argnum*, *value-reference* **)**

value-reference:
[unsigned] long & *value*
| p_ul_binary *value*
| unsigned char & *value*
| ul_char * *value*
| double & *value*
| float & *value*
| [unsigned] short & *value*
| DECL_DATETIME *value*
| DECL_BIGINT *value*
| DECL_UNSIGNED_BIGINT *value*

説明

このメソッドは、Execute メソッドが呼び出だされる前に、文のパラメータを設定します。

パラメータを NULL に設定するには、SetNull メソッドを使用します。

パラメータ

argnum　設定する引数の識別子。最初の引数は 1、2 番目の引数は 2、のようになります。

value-reference　パラメータ値の参照。上記のリストは、考えられるデータ型を示したものです。このパラメータはポインタとして渡されるため、使用されるまで有効になっています。使用されるまで開放しないようにします。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。不適切なデータ型のパラメータを指定すると、このメソッドは失敗します。

参照

「Execute メソッド」435 ページ

「SetNull メソッド」436 ページ

生成されたテーブル・クラス

オブジェクト

生成されたテーブル・クラスは、アプリケーションのデータベース・テーブルを表します。クラスの名前は、データベースのテーブル名に基づいて、Ultra Light ジェネレータによって生成されます。

プロトタイプ

Tablename tb;
    tb.**Open()**;

Tablename:
    generated name

説明

Ultra Light ジェネレータは、指定のパブリケーションにあるテーブルごとに、クラスを定義します。生成されたテーブル・クラスは、**ULTable** と **ULCursor** を継承します。クラスには、テーブルや文の名前に基づいた名前が付けられます。たとえば、Product というテーブルには、**Product** という名前のクラスが定義されます。

API 階層での位置付けについては、「C++ API クラス階層」374 ページを参照してください。

参照

「ULCursor クラス」395 ページ

「ULTable クラス」424 ページ

Get 生成メソッド

プロトタイプ

bool **Get (**
    ul_column_num *column-index*,
    value-declaration,
    bool* is-null = **UL_NULL);**

value-declaration:
    ul_char * *buffer-name*, ul_length *buffer-length*
    | p_ul_binary *buffer-name*, ul_length *buffer-length*
    | DECL_DATETIME & *date-value*
    | { DECL_BIGINT | DECL_UNSIGNED_BIGINT } & *bigint-value*
    | unsigned char & *char-value*
    | double & *double-value*

| float & *float-value*
| [unsigned] long & *integer-value*
| [unsigned] short & *short-value*

説明

インデックスにより指定されたカラムから値を取得します。

パラメータ

column-index　カラムの番号。テーブルの最初のカラム番号は 1 です。

value declaration　値の指定に必要な引数。指定方法はデータ型に依存します。文字とバイナリ・データは、呼び出しに指定するバッファの名前と長さに基づいてバッファにマップされます。その他のデータ型の場合は、適切な型の変数へのポインタが必要です。文字データの場合、ターミネータ用に使用する領域を*含む* C 配列の長さを指定するのは、長さパラメータです。

isNULL　カラムの値が NULL の場合、**isNull** は **true** に設定されます。この場合、引数 **value** を指定しても意味がありません。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例

次のサンプル・コードは、該当するデータ型に基づくローから値を取得する **switch** 文の一部を示しています。

```
switch( tb.GetColumnType( colIndex ) ) {
 case tb.S_LONG :
     ret = tb.Get( colIndex, longval );
     printf( "Long column: %d¥n", longval );
     break;
 ...
```

参照

「データ型列挙」395 ページ

「Get メソッド」400 ページ

「Get<Column> 生成メソッド」440 ページ

「GetColumnSize メソッド」401 ページ

Get<Column> 生成メソッド

プロトタイプ

bool **Get***column-name***(** type* *variable*, [ul_length* *length*,]
bool* *isNull* = **UL_NULL)**

説明

column-name の値を取得します。データ型は、カラムのデータ型に依存します。

パラメータ

column-name カラムの名前。

variable カラムに対する、適切なデータ型の変数。このデータ型は、**GetColumnType** を使って取得できます。

length 可変長データ型に使用。文字データの場合、ターミネータ用に使用する領域を*含む* C 配列の長さを指定するのは、長さパラメータです。

isNull NULL 値の場合、この引数は **true** になります。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

参照

「Get 生成メソッド」438 ページ

GetSize<Column> 生成メソッド

プロトタイプ

ul_length **GetSize***column-name***( )**

説明

指定されたカラムの値を保持するために必要な記憶領域を返します。

パラメータ

column-name リファレンス・データベースに格納されているカラム名を元に生成された名前。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

参照

「GetColumnType メソッド」401 ページ

Open メソッド

プロトタイプ

bool **Open(** ULConnection* *conn* **)**

bool **Open(** ULConnection* *conn*, ul_index_num *index* **)**

説明

生成されたテーブル・オブジェクトのデータを操作できるように、アプリケーションを準備します。

パラメータ

conn　ULConnection オブジェクトのアドレス。接続を開きます。

index　テーブル内のローの順序付けに使用する、オプションのインデックス番号。このインデックスは、生成されたインデックス列挙のメンバの 1 つです。デフォルトでは、テーブルはプライマリ・キーの値に基づいて順序付けられます。

詳細については、「インデックス列挙」442 ページを参照してください。

テーブルを開くと、カーソルは最初のローの前に置かれます。

戻り値

成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

例

次のサンプル・コードは、ULProduct テーブルに対して生成されたオブジェクトを宣言して開きます。

```
ULData db;
 ULConnection conn;
 ULProduct tb;
 db.Open();
 conn.Open( &db, "DBA", "SQL" );
 tb.Open( &conn );
```

参照

「Close メソッド」398 ページ

「インデックス列挙」442 ページ

Set<Column> 生成メソッド

プロトタイプ　bool **Set***column-name***( )**

説明　現在のカーソル位置の値を設定します。ローのデータは、Insert または Update 文が実行されるまで実際には変更されません。また、その変更がコミットされないかぎり、永続化されません。

パラメータ　**column-name**　リファレンス・データベースに格納されているカラム名を元に生成された名前。

戻り値　成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

参照　「SetColumnNull メソッド」409 ページ

SetNull<Column> 生成メソッド

プロトタイプ　bool **SetNull***column-name***( )**

説明　カラムに SQL NULL を設定します。データは、Insert 文または Update 文を実行するまでは、実際には変更されません。また、その変更がコミットされないかぎり、永続化されません。

パラメータ　**column-name**　リファレンス・データベースに格納されているカラム名を元に生成された名前。

戻り値　成功の場合は **true** (1) です。

失敗の場合は **false** (0) です。

参照　「SetColumnNull メソッド」409 ページ

インデックス列挙

プロトタイプ　**enum**{ *index-name*,… }

説明

列挙の各メンバは、生成中のテーブルにおけるインデックス名です。テーブルが開かれている場合、テーブルの順序付けを指定するときにインデックス名を使用できます。これにより、カーソル移動メソッドの動作を制御できます。

パラメータ

index-name　テーブルにあるインデックスの名前。プラマイリ・キーは **Primary** という名前です。他のインデックスはデータベースにそれぞれの名前があります。

参照

「Open メソッド」441 ページ

第 15 章

Embedded SQL API リファレンス

この章の内容

この章では、Ultra Light Embedded SQL アプリケーションで使用できる関数について説明します。Embedded SQL EXEC SQL 文と、アプリケーションに含むことができる SQL 文は説明に含まれません。これらの文については、「Embedded SQL を使用したアプリケーションの開発」73 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

プロトタイプ

unsigned short **db_fini(** SQLCA * *sqlca* **);**

説明

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。

db_fini が呼び出された後に、他の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。処理中にエラーが発生すると、SQLCA 内でエラー・コードが設定され、関数は 0 を返します。エラーがなければ、0 以外の値が返されます。

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」447 ページ

db_init 関数

プロトタイプ

unsigned short **db_init(**SQLCA * *sqlca* **) ;**

説明

Ultra Light ランタイム・ライブラリを初期化し、Ultra Light データベースが存在しない場合は新しく作成します。

この関数は、他のライブラリが呼び出される前に、そして Embedded SQL コマンドが実行される前に呼び出さなければなりません。このルールには、次に示す例外があります。

- データベース記憶領域を設定する関数を呼び出すことができます。この種の関数の名前は、**ULEnable** で始まります。

旧式な機能

db_init を呼び出した*直後*に、ULEnable 関数を呼び出すことをおすすめします。db_init の前に ULEnable 関数を呼び出す方法は使用されなくなりました。

永続ストレージの初期化中など、処理中にエラーが発生した場合は、SQLCA に渡されて 0 が返されます。エラーがなかった場合は、0 以外の値が返され、ESQL コマンドと関数の使用を開始できます。

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。

参照

「db_fini 関数」446 ページ

db_start_database 関数

プロトタイプ

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。詳細については、「SQLCA (SQL Communication Area) の初期化」77 ページを参照してください。

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"
```

使用可能な接続パラメータのリストについては、『Ultra Light データベース・ユーザーズ・ガイド』> 「接続パラメータ」を参照してください。

説明

データベースがまだ起動していない場合、データベースを起動します。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必要です。「Ultra Light C/C++ インタフェースの結合」132 ページを参照してください。

db_start_database 関数を使用してデータベースを起動する場合、UL_STORE_PARMS 内の情報は無視されます。

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、戻り値は true (非ゼロ値) であり、SQLCODE には 0 が設定されます。エラー情報は SQLCA に返されます。

db_stop_database 関数

プロトタイプ

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。詳細については、「SQLCA (SQL Communication Area) の初期化」77 ページを参照してください。

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成る、NULL で終了された文字列。通常、データベース・ファイルだけが必要です。次に例を示します。

```
"DBF=c:¥¥db¥¥mydatabase.db"
```

使用可能な接続パラメータのリストについては、『Ultra Light データベース・ユーザーズ・ガイド』> 「接続パラメータ」を参照してください。

説明

データベースを停止します。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。「Ultra Light C/C++ インタフェースの結合」132 ページを参照してください。

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。

戻り値の TRUE は、エラーがなかったことを表します。

ULActiveSyncStream 関数

プロトタイプ

ul_stream_defn **ULActiveSyncStream(** void **);**

説明

同期に適した ActiveSync ストリームを定義します。

ActiveSync ストリームは、Windows CE デバイスでのみ利用できます。

ULActiveSyncStream を使用する同期は、必ず ActiveSync ソフトウェアから開始します。アプリケーションは、**WindowProc** 関数で処理するメッセージを受け取ります。**ULIsSynchronizeMessage** を使用して、そのメッセージを同期命令として識別できます。

参照

「ULIsSynchronizeMessage 関数」466 ページ

「ULSynchronize 関数」478 ページ

「Synchronize メソッド」393 ページ

『Mobile Link クライアント』> 「ActiveSync プロトコル・オプション」

ULChangeEncryptionKey 関数

プロトタイプ

ul_bool **ULChangeEncryptionKey(** SQLCA **sqlca*, ul_char **new_key* **);**

説明

Ultra Light データベースの暗号化キーを変更します。

警告
キーを変更すると、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化され、新しいキーを使用して再度暗号化されます。この操作は回復不能です。操作の途中でアプリケーションが中断されると、データベースは無効になり、アクセスできなくなります。新しいデータベースを作成してください。

参照

「データの暗号化」105 ページ

ULClearEncryptionKey 関数

プロトタイプ

```
ul_bool ULClearEncryptionKey(
    ul_u_long * creator,
    ul_u_long * feature-num  );
```

説明

Palm Computing Platform 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。

この関数は暗号化キーをクリアします。

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。NULL 値には Ultra Light のデフォルト (機能番号 100) を使用します。

参照

「ULRetrieveEncryptionKey 関数」471 ページ

「ULSaveEncryptionKey 関数」474 ページ

『Ultra Light データベース・ユーザーズ・ガイド』> 「Palm OS に関する検討事項」

ULCountUploadRows 関数

プロトタイプ

ul_u_long **ULCountUploadRows (**
SQLCA * *sqlca*,
ul_publication_mask *publication-mask*,
ul_u_long *threshold* **);**

説明

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を返します。

この関数を使用して、ユーザに同期を要求します。

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 0 はデータベース全体を示します。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
```

パブリケーション・マスクの詳細については、『Mobile Link クライアント』> 「別々に同期するためのデータ・セットの設計」を参照してください。

threshold　カウントするローの最大数を決定する値。呼び出しにかかる時間を制限します。値 0 は制限がないことを示します。値 1 は、同期の必要なローがあるかどうかを判別する場合に使用します。

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 );
```

ULDropDatabase 関数

プロトタイプ

ul_bool **ULDropDatabase (** SQLCA * *sqlca*, ul_char * *store-parms* **);**

説明

Ultra Light データベース・ファイルを削除します。

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。使用する場合は注意してください。

この関数は、データベース接続が開いている間は呼び出さないでください。この関数は、**db_init** の前、または **db_fini** の後にだけ呼び出します。

Palm OS では、この関数は、データベースに接続していない場合のみ (ただし、**ULEnable** 関数が呼び出された直後) 呼び出します。

パラメータ

sqlca　SQLCA へのポインタ。

store-parms　接続パラメータの文字列。削除するファイル名を **file_name=***file.udb* という形式のキーワードと値の組み合わせで含みます。この引数として UL_STORE_PARMS マクロを使用すると便利な場合があります。UL_NULL の値は、デフォルトのデータベース・ファイル名を削除します。

詳細については、「UL_STORE_PARMS マクロ」270 ページを参照してください。

戻り値

- **ul_true**　データベース・ファイルが正常に削除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 フィールドで定義されています。よくある失敗の理由は、不正なファイル名が提供されたか、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場合です。

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。

```
#define UL_STORE_PARMS UL_TEXT("file_name=myfile.udb")
 if( ULDropDatabase(&sqlca, UL_STORE_PARMS ) ){
     // success
 };
```

ULGetDatabaseID 関数

プロトタイプ

ul_u_long **ULGetDatabaseID(** SQLCA * *sqlca* **)**

説明

ULGetDatabaseID は、グローバル・オートインクリメントに使用する現在のデータベースの ID を返します。返される値は、直前の SetDatabaseID 呼び出しで設定された値、または ID が設定されていない場合は UL_INVALID_DATABASE_ID です。

パラメータ

sqlca　SQLCA へのポインタ。

ULGetLastDownloadTime 関数

プロトタイプ

ul_bool **ULGetLastDownloadTime(**
SQLCA * *sqlca*,
ul_publication_mask *publication-mask*,
DECL_DATETIME * *value* **);**

説明

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。

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 はデータベース全体を示します。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
```

パブリケーション・マスクの詳細については、『Mobile Link クライアント』> 「別々に同期するためのデータ・セットの設計」を参照してください。

value 投入する DECL_DATETIME 構造体へのポインタ。

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* が正常に投入されたことを示します。
- **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あることを示します。戻り値が false の場合は、*value* の内容が有効ではないことを示します。

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

「UL_SYNC_ALL マクロ」272 ページ

「UL_SYNC_ALL_PUBS マクロ」272 ページ

ULGetSynchResult 関数

プロトタイプ

ul_bool **ULGetSynchResult(** ul_synch_result * *synch-result* **);**

説明

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。

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 Computing Platform 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。

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;
```

個々のメンバには次の意味があります。

- **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、『ASA エラー・メッセージ』> 「Adaptive Server Anywhere SQLCODE 順にリストしたエラー・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。

詳細については、「stream_error 同期パラメータ」534 ページを参照してください。

- **upload_ok**　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。
- **ignored_rows**　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。
- **auth_status**　同期認証ステータス。詳細については、「auth_status パラメータ」511 ページを参照してください。
- **auth_value**　Mobile Link 同期サーバが **auth_status** の結果を判断するために使用する値。詳細については、「auth_value 同期パラメータ」512 ページ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。詳細については、「observer 同期パラメータ」520 ページを参照してください。

戻り値

関数がブール値を返します。

true　成功を示します。

false　失敗を示します。

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

プロトタイプ

short **ULGlobalAutoincUsage(** SQLCA * *sqlca* **);**

説明

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。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。

戻り値

0 ～ 100 の範囲内の値が返されます。

参照

「ULSetDatabaseID 関数」475 ページ

ULGrantConnectTo 関数

プロトタイプ

void **ULGrantConnectTo(**
SQLCA * *sqlca*,
ul_char * *userid*,
ul_char * *password* **);**

説明

指定されたパスワードを持つユーザ ID に、Ultra Light データベースへのアクセスを許可します。既存のユーザ ID が指定されていれば、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。

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　*userid* のパスワードを保持する文字配列。最大長は 16 文字です。

参照

『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light のユーザ認証」

「ユーザの認証」102 ページ

「ULRevokeConnectFrom 関数」472 ページ

ULHTTPSStream 関数

プロトタイプ

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

説明

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。

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。

参照

「ULSynchronize 関数」478 ページ

「Synchronize メソッド」393 ページ

「stream 同期パラメータ」531 ページ

『Mobile Link クライアント』> 「HTTPS プロトコル・オプション」

ULHTTPStream 関数

プロトタイプ

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

説明

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。

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。

参照

「ULSynchronize 関数」478 ページ

「Synchronize メソッド」393 ページ

「stream 同期パラメータ」531 ページ

『Mobile Link クライアント』> 「HTTP プロトコル・オプション」

ULIsSynchronizeMessage 関数

プロトタイプ

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

説明

この関数は Windows CE 上で、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。

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。

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」171 ページ

ULPalmDBStream 関数 (旧式)

プロトタイプ

ul_stream_defn **ULPalmDBStream(** void **);**

説明

Palm Computing Platform で、HotSync と Scout Sync に適したストリームを定義します。

この関数は使用されなくなりました。HotSync 同期の場合、**stream** パラメータは不要です。また、UL_NULL も指定できます。

参照

『Mobile Link クライアント』> 「HotSync プロトコル・オプション」

「Synchronize メソッド」393 ページ

ULPalmExit 関数 (旧式)

プロトタイプ

ul_bool **ULPalmExit(** SQLCA * *sqlca*, ul_synch_info * *synch_info* **);**

説明

旧式な機能
ULPalmExit はもはや不要で、使用されなくなりました。アプリケーションを終了するには、代わりに ULSetSynchInfo と db_fini を使用します。

Palm Computing Platform 上で動作している Ultra Light アプリケーションの状態を保存し、HotSync 同期のアップロード・ストリームを記述します。

パラメータ

sqlca SQLCA へのポインタ。

synch_info 同期構造体。

TCP/IP または HTTP 同期を使用している場合は、ul_synch_info 構造体の代わりに UL_NULL を指定します。これらのストリームを使用している場合は、**ULSynchronize** の呼び出しで代わりに同期情報が指定されます。

HotSync または Scout Sync 同期を使用している場合は、同期構造体を指定します。**stream** パラメータの値は無視されます。UL_NULL も指定できます。

synch_info 構造体のメンバについては、「同期パラメータ・リファレンス」507 ページを参照してください。

戻り値

関数がブール値を返します。

true 成功を示します。

false 失敗を示します。

ULPalmLaunch 関数 (旧式)

プロトタイプ

```
UL_PALM_LAUNCH_RET ULPalmLaunch(
    SQLCA * sqlca,
    ul_synch_info * synch_info );

typedef enum {
    LAUNCH_SUCCESS_FIRST,
    LAUNCH_SUCCESS,
    LAUNCH_FAIL
    } UL_PALM_LAUNCH_RET;
```

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については、「同期パラメータ」508 ページを参照してください。

TCP/IP または HTTP 同期を使用している場合は、*synch_info* として UL_NULL を指定します。

説明

旧式な機能
ULPalmLaunch はもはや不要で、使用されなくなりました。アプリケーションを起動するには、代わりに db_init を使用します。

この関数は、Palm Computing Platform で Ultra Light アプリケーションの状態をリストアします。この関数は、すべての Ultra Light Palm アプリケーションで必要です。

参照

「Ultra Light Palm アプリケーションの状態のリストア」148 ページ

「ULEnableFileDB 関数」253 ページ

「ULEnablePalmRecordDB 関数」256 ページ

ULResetLastDownloadTime 関数

プロトタイプ

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

説明

この関数を使用して、値を再配置し、アプリケーションを既知のクリーンな状態に戻すことができます。この関数は、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。

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 0 はデータベース全体を示します。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
```

パブリケーション・マスクの詳細については、「publication 同期パラメータ」524 ページを参照してください。

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

「ULGetLastDownloadTime 関数」458 ページ

『Mobile Link 管理ガイド』> 「タイムスタンプベースの同期」

ULRetrieveEncryptionKey 関数

プロトタイプ

```
ul_bool ULRetrieveEncryptionKey(
    ul_char * key,
    ul_u_short len,
    ul_u_long * creator,
    ul_u_long * feature-num  );
```

説明

Palm Computing Platform 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。

この関数は、メモリから暗号化キーを取り出すときに使用します。

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。NULL 値には Ultra Light のデフォルト (機能番号 100) を使用します。

戻り値

- **true**　操作が成功したことを示します。
- **false**　操作が成功していないことを示します。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。

参照

「ULClearEncryptionKey 関数」452 ページ

「ULSaveEncryptionKey 関数」474 ページ

『Ultra Light データベース・ユーザーズ・ガイド』> 「Palm OS に関する検討事項」

ULRevokeConnectFrom 関数

プロトタイプ

void **ULRevokeConnectFrom(** SQLCA * *sqlca*, ul_char * *userid* **);**

説明

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。

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

『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light のユーザ認証」

「ユーザの認証」102 ページ

「ULGrantConnectTo 関数」463 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。

プロトタイプ

void **ULRollbackPartialDownload (** SQLCA * *sqlca* **)**

パラメータ

- **sqlca** SQLCA へのポインタ。

  静的型 C++ API の場合、SQLCA はヘッダ・ファイルの中で **sqlca** として宣言されます。C++ コンポーネントの場合は、Sqlca.GetCA() メソッドを使用します。

説明

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し、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

- 『Mobile Link 管理ガイド』> 「失敗したダウンロードの再開」
- 『Mobile Link クライアント』> 「Keep Partial Download 同期パラメータ」
- 『Mobile Link クライアント』> 「Partial Download Retained 同期パラメータ」
- 『Mobile Link クライアント』> 「Resume Partial Download 同期パラメータ」

ULSaveEncryptionKey 関数

プロトタイプ

```
ul_bool ULSaveEncryptionKey(
    ul_char * key,
    ul_u_long * creator,
    ul_u_long * feature-num  );
```

説明

Palm Computing Platform 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。

この関数は、Palm の動的メモリに暗号化キーを保存します。

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。NULL 値には Ultra Light のデフォルト (機能番号 100) を使用します。

戻り値

- **true**　操作が成功したことを示します。
- **false**　操作が成功していないことを示します。

参照

「ULClearEncryptionKey 関数」452 ページ

「ULRetrieveEncryptionKey 関数」471 ページ

『Ultra Light データベース・ユーザーズ・ガイド』> 「Palm OS に関する検討事項」

ULSetDatabaseID 関数

プロトタイプ

void **ULSetDatabaseID(** SQLCA * *sqlca*, ul_u_long *id* **);**

説明

データベースの ID 番号を設定します。

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

「ULGlobalAutoincUsage 関数」462 ページ

ULSetSynchInfo 関数

プロトタイプ

```
ul_bool ULSetSynchInfo(
        SQLCA * sqlca,
        ul_synch_info *  synch_info );
```

説明

Palm OS 上での HotSync 同期の場合は、ULSetSynchInfo を使用して、HotSync の実行時に使用する同期パラメータを格納します。通常、ULSetSynchInfo は、db_fini または ULData::Close を呼び出してアプリケーションを閉じる直前に呼び出されます。

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については、「同期パラメータ」508 ページを参照してください。

ULSocketStream 関数

プロトタイプ

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

説明

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。

参照

「ULSynchronize 関数」478 ページ

「Synchronize メソッド」393 ページ

ULSynchronize 関数

プロトタイプ

void **ULSynchronize(**
SQLCA * *sqlca*,
ul_synch_info * *synch_info* **);**

説明

Ultra Light アプリケーションで同期を開始します。

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。

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については、「同期パラメータ」508 ページを参照してください。

参照

『Mobile Link 管理ガイド』> 「Mobile Link 同期サーバのオプション」

『ASA SQL リファレンス・マニュアル』> 「START SYNCHRONIZATION DELETE 文 [Mobile Link]」

第 16 章

Ultra Light ODBC API リファレンス

この章の内容

この章では、Ultra Light でサポートされる ODBC インタフェース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

ハンドルを割り当てます。

プロトタイプ

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。

説明

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。ほか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

- SQLAllocHandle (Microsoft の「*ODBC Programmer's Reference*」)

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。

プロトタイプ

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_lnd** 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。

説明

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

- SQLBindCol (Microsoft の「*ODBC Programmer's Reference*」)

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。

プロトタイプ

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。

説明

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

- SQLBindParameter (Microsoft の「*ODBC Programmer's Reference*」)

SQLConnect 関数

Ultra Light ODBC の場合、データベースに接続します。

プロトタイプ

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、スキーマ・ファイルとデータベース・ファイル・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提供します。

  次は、ServerName パラメータの使用例です。

  ```
  (SQLTCHAR*)UL_TEXT(
       "schema_file=customer.usm;dbf=customer.udb"
  )
  ```

  接続パラメータの完全なリストについては、『Ultra Light データベース・ユーザーズ・ガイド』> 「接続パラメータ」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。

説明

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 データベース・ユーザーズ・ガイド』> 「接続パラメータ」を参照してください。

参照

- SQLConnect (Microsoft の『*ODBC Programmer's Reference*』)

SQLDescribeCol 関数

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。

プロトタイプ

```
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

- SQLDescribeCol (Microsoft の「*ODBC Programmer's Reference*」)

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。

プロトタイプ

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。

説明

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」484 ページ
- SQLDisconnect (Microsoft の『*ODBC Programmer's Reference*』)

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。

プロトタイプ

```
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

- SQLEndTran (Microsoft の「*ODBC Programmer's Reference*」)

SQLExecDirect 関数

Ultra Light ODBC の場合、SQL 文を実行します。

プロトタイプ

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。

説明

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

- SQLExecDirect (Microsoft の「*ODBC Programmer's Reference*」)

SQLExecute 関数

Ultra Light ODBC の場合、準備された SQL 文を実行します。

プロトタイプ

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。

説明

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」482 ページ
- 「SQLPrepare 関数」501 ページ

- SQLExecute (Microsoft の「*ODBC Programmer's Reference*」)

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。

プロトタイプ

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。

説明

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」493 ページ
- 「SQLBindCol 関数」481 ページ
- SQLFetch (Microsoft の「*ODBC Programmer's Reference*」)

SQLFetchScroll 関数

結果セットから指定し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。

プロトタイプ

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。

説明

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」492 ページ
- 「SQLBindCol 関数」481 ページ
- SQLFetchScroll (Microsoft の「*ODBC Programmer's Reference*」)

SQLFreeHandle 関数

Ultra Light ODBC 用に割り当てられているハンドルを解放します。

プロトタイプ

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。

説明

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」480 ページ
- SQLFreeHandle (Microsoft の「*ODBC Programmer's Reference*」)

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。

プロトタイプ

```
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」503 ページ
- SQLGetCursorName (Microsoft の「*ODBC Programmer's Reference*」)

SQLGetData 関数

結果セット内の 1 つのカラムのデータを取得します。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。

プロトタイプ

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。

参照

- SQLGetData (Microsoft の「*ODBC Programmer's Reference*」)

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。

プロトタイプ

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。エラー・コードのリストについては、『ASA エラー・メッセージ』> 「SQLSTATE 順にリストしたエラー・メッセージ」を参照してください。
- **NativeError** エラーの SQLCODE 値。エラー・コードのリストについては、『ASA エラー・メッセージ』> 「Adaptive Server Anywhere SQLCODE 順にリストしたエラー・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

- SQLGetDiagRec (Microsoft の「*ODBC Programmer's Reference*」)

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。

プロトタイプ

```
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

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

SQLNumResultCols 関数

Ultra Light ODBC の場合、結果セットのカラム数を返します。

プロトタイプ

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

- SQLNumResultCols (Microsoft の「*ODBC Programmer's Reference*」)

SQLPrepare 関数

Ultra Light ODBC の場合、実行する SQL 文を準備します。

プロトタイプ

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」491 ページ
- SQLPrepare (Microsoft の「*ODBC Programmer's Reference*」)

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。

プロトタイプ

```
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

- SQLRowCount (Microsoft の「*ODBC Programmer's Reference*」)

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。

プロトタイプ

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」495 ページ
- SQLSetCursorName (Microsoft の「*ODBC Programmer's Reference*」)

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。

プロトタイプ

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。

説明

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状態の保存」147 ページ
- 「SQLSetSuspend 関数」505 ページ

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。

プロトタイプ

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状態の保存」147 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。

プロトタイプ

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。「同期パラメータ・リファレンス」507 ページを参照してください。

説明

SQLSynchronize は ODBC の拡張機能です。Mobile Link 同期操作を開始します。

参照

- 「同期パラメータ・リファレンス」507 ページ .
- 『Mobile Link 管理ガイド』> 「Mobile Link 管理ガイド」

第 17 章

同期パラメータ・リファレンス

この章の内容

この章では、同期パラメータのリファレンス情報を提供します。

同期パラメータ

同期パラメータは、同期の呼び出しで引数として供給する構造体のメンバです。同期パラメータを格納する **ul_synch_info** 構造体は、次のように *ulglobal.h* で定義します。

```
struct ul_synch_info {
     ul_char *              user_name;
     ul_char *              password;
     ul_char *              new_password;
     ul_char *              version;
     p_ul_stream_defn       stream;
     ul_char *              stream_parms;
     p_ul_stream_defn       security;
     ul_char *              security_parms;
     ul_synch_observer_fn   observer;
     ul_void *              user_data;
     ul_publication_mask    publication;
     ul_bool                upload_only;
     ul_bool                download_only;
     ul_bool                send_download_ack;
     ul_bool                send_column_names;
     ul_bool                ping;
     ul_bool                checkpoint_store;
     ul_bool                disable_concurrency;
     ul_byte                num_auth_params;
     ul_char * *            auth_parms;
     ul_bool                keep_partial_download;
     ul_bool                resume_partial_download;

     // fields set on output
     ul_stream_error        stream_error;
     ul_bool                upload_ok;
     ul_bool                ignored_rows;
     ul_auth_status         auth_status;
     ul_s_long              auth_value;
     ul_bool                partial_download_retained;

     p_ul_synch_info        init_verify;
 };
```

init_verify フィールドは内部用に予約されています。

UL_TEXT マクロを使用して定数文字列を囲む

UL_TEXT マクロでは、定数文字列をシングルバイト文字列としてもワイド文字列としてもコンパイルできます。このマクロを使用して、**ul_synch_info** 構造体のメンバとして提供されている定数文字列をすべて囲み、コンパイラがこれらのパラメータを正しく処理できるようにします。

各同期パラメータの役割については、『Mobile Link クライアント』> 「同期パラメータ」を参照してください。

auth_parms パラメータ

機能

カスタム・ユーザ認証スクリプトにパラメータを供給します。

使用法

次のようにパラメータを設定します。

```
ul_char * Params[ 3 ] = { UL_TEXT( "parm1" ),
                          UL_TEXT( "parm2" ),
                          UL_TEXT( "parm3" ) };
 // ...
 info.num_auth_parms = 3;
 info.auth_parms = Params;
```

参照

- 『Mobile Link クライアント』> 「Authentication Parameters 同期パラメータ」
- 「num_auth_parms パラメータ」519 ページ
- 『Mobile Link 管理ガイド』> 「authenticate_parameters 接続イベント」
- 『Mobile Link 管理ガイド』> 「authenticate_user 接続イベント」

auth_status パラメータ

機能

Mobile Link のユーザ認証のステータスをレポートします。

使用法

このパラメータにアクセスするには、次のように入力します。

```
ul_synch_info info;
 // ...
 returncode = info.auth_status;
```

指定可能な値

同期後は、パラメータには次の値の 1 つが入ります。統合データベースでカスタム **authenticate_user** 同期スクリプトが異なる値を返す場合は、『Mobile Link 管理ガイド』> 「authenticate_user 接続イベント」で指定されているルールに従って値が解釈されます。

定数	値	説明
UL_AUTH_STATUS_UNKNOWN	0	認証ステータスが不明です。接続がまだ同期していない可能性があります。
UL_AUTH_STATUS_VALID	1000	ユーザ ID とパスワードは、同期時には有効でした。
UL_AUTH_STATUS_VALID_BUT_EXPIRES_SOON	2000	ユーザ ID とパスワードは、同期時には有効でしたが、まもなく有効期限が切れます。
UL_AUTH_STATUS_EXPIRED	3000	認証に失敗しました。ユーザ ID またはパスワードの有効期限が切れています。
UL_AUTH_STATUS_INVALID	4000	認証に失敗しました。不正なユーザ ID またはパスワードです。
UL_AUTH_STATUS_IN_USE	5000	認証に失敗しました。ユーザ ID はすでに使用されています。

参照

- 『Mobile Link クライアント』> 「Authentication Status 同期パラメータ」
- 『Mobile Link クライアント』> 「Mobile Link ユーザの認証」.

auth_value 同期パラメータ

機能

カスタム・ユーザ認証同期スクリプトからの戻り値をレポートします。

デフォルト

デフォルトの Mobile Link ユーザ認証メカニズムによって設定される値については、『Mobile Link 管理ガイド』> 「authenticate_user 接続イベント」で説明します。

使用法

このパラメータは読み込み専用です。

このパラメータにアクセスするには、次のように入力します。

```
ul_synch_info info;
 // ...
 returncode = info.auth_value;
```

参照

- 『Mobile Link クライアント』> 「Authentication Value 同期パラメータ」
- 『Mobile Link 管理ガイド』> 「authenticate_user 接続イベント」
- 『Mobile Link 管理ガイド』> 「authenticate_user_hashed 接続イベント」
- 「auth_status パラメータ」511 ページ

checkpoint_store 同期パラメータ

機能

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。

デフォルト

デフォルトでは、制限されたチェックポイントが実行されます。

使用法

次のようにパラメータを設定します。

```
ul_synch_info info;
 // ...
 info.checkpoint_store = ul_true ;
```

参照

『Mobile Link クライアント』> 「Checkpoint Store 同期パラメータ」

disable_concurrency 同期パラメータ

機能

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。

デフォルト

デフォルトで、データ・アクセスは可能です。ダウンロード時のデータ・アクセスは読み込み、書き込みともに可能です。それ以外は読み込み専用です。

使用法

次のようにパラメータを設定します。

```
ul_synch_info info;
 // ...
 info.checkpoint_store = ul_false ;
```

参照

- 『Mobile Link クライアント』> 「Disable Concurrency 同期パラメータ」
- 『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light での同時実行性について」

download_only 同期パラメータ

機能

この同期中は、Ultra Light データベースから変更をアップロードしません。

デフォルト

このパラメータはオプションのブール値であり、デフォルトは false です。

使用法

次のようにパラメータを設定します。

```
ul_synch_info info;
 // ...
 info.download_only = ul_true;
```

参照

- 『Mobile Link クライアント』> 「Download Only 同期パラメータ」
- 『Mobile Link クライアント』> 「Ultra Light データベースへの読み込み専用テーブルの組み込み」
- 「upload_only 同期パラメータ」540 ページ

keep_partial_download 同期パラメータ

機能

ダウンロード・エラーが発生した場合、すべての変更をロールバックするのではなく、部分的に行われたダウンロードを保持します。

デフォルト

このパラメータはオプションのブール値であり、デフォルトは false です。

使用法

次のようにパラメータを設定します。

```
ul_synch_info info;
 // ...
 info.keep_partial_download = ul_true;
```

参照

- 『Mobile Link 管理ガイド』> 「失敗したダウンロードの再開」
- 『Mobile Link クライアント』> 「Keep Partial Download 同期パラメータ」

ignored_rows 同期パラメータ

機能

同期中にスクリプトがないため、Mobile Link 同期サーバによってローが無視された場合、レポートを行います。

このパラメータは読み込み専用です。

new_password 同期パラメータ

機能

ユーザ名に対する新しい Mobile Link パスワードを設定します。

デフォルト

デフォルト値はありません。

使用法

次のようにパラメータを設定します。

```
ul_synch_info info;
 // ...
 info.password = UL_TEXT( "myoldpassword" );
 info.new_password = UL_TEXT( "mynewpassword" );
```

参照

- 『Mobile Link クライアント』> 「New Password 同期パラメータ」
- 『Mobile Link クライアント』> 「Mobile Link ユーザの認証」.

num_auth_parms パラメータ

機能

カスタム認証スクリプトに渡される認証パラメータ文字列の数。

デフォルト

カスタム認証スクリプトに渡されるパラメータはありません。

使用法

auth_parms とともにパラメータを使用して、カスタム認証スクリプトに情報を提供します。

詳細については、「auth_parms パラメータ」510 ページを参照してください。

参照

- 『Mobile Link クライアント』> 「Number of Authentication Parameters パラメータ」
- 「auth_parms パラメータ」510 ページ
- 『Mobile Link 管理ガイド』> 「authenticate_parameters 接続イベント」
- 『Mobile Link 管理ガイド』> 「authenticate_user 接続イベント」

observer 同期パラメータ

機能

同期をモニタするコールバック関数へのポインタです。

参照

- 『Mobile Link クライアント』> 「Observer 同期パラメータ」
- 「同期のモニタとキャンセル」65 ページ
- 「user_data 同期パラメータ」541 ページ

partial_download_retained 同期パラメータ

機能

同期のダウンロード時の失敗の結果として、部分的なダウンロードが保持されたことを示します。

デフォルト

ダウンロード・エラーが発生して部分的なダウンロードが保持された場合、同期時にパラメータが設定されます。

参照

- 『Mobile Link 管理ガイド』> 「失敗したダウンロードの再開」
- 『Mobile Link クライアント』> 「Partial Download Retained 同期パラメータ」

password 同期パラメータ

機能

user_name に対する Mobile Link パスワードを指定する文字列です。このユーザ名とパスワードは他のデータベース・ユーザ ID やパスワードとは別のもので、アプリケーションを Mobile Link 同期サーバに対して識別し、認証するために使用されます。

デフォルト

デフォルト値はありません。

使用法

次のようにパラメータを設定します。

```
ul_synch_info info;
 // ...
 info.password = UL_TEXT( "mypassword" );
```

参照

- 『Mobile Link クライアント』> 「Password 同期パラメータ」
- 『Mobile Link クライアント』> 「Mobile Link ユーザの認証」.

ping 同期パラメータ

機能

Ultra Light クライアントと Mobile Link 同期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。

Mobile Link 同期サーバは、ping 要求を受信すると、統合データベースに接続し、ユーザを認証し、ユーザ認証ステータスと値をクライアントに送信します。

ping に成功した場合、Mobile Link サーバは情報メッセージを発行します。ping に失敗した場合は、エラー・メッセージを発行します。

Mobile Link サーバがコマンド・ライン・オプション -zu+ を指定して実行されていると、Mobile Link ユーザ名が ml_user システム・テーブルに見つからない場合は Mobile Link サーバがユーザを ml_user に追加します。

Mobile Link 同期サーバに次のスクリプトが存在する場合、ping 要求に対してこれらのスクリプトを実行できます。

- begin_connection
- authenticate_user
- authenticate_user_hashed
- end_connection

デフォルト

このパラメータはオプションのブール値であり、デフォルトは false です。

使用法

次のようにパラメータを設定します。

```
ul_synch_info info;
 // ...
 info.ping = ul_true;
```

参照

- 『Mobile Link クライアント』> 「Ping 同期パラメータ」
- 『Mobile Link クライアント』> 「-pi オプション」

publication 同期パラメータ

機能

同期させるパブリケーションを指定します。

デフォルト

パブリケーションを指定しない場合、すべてのデータが同期されます。

使用法

ulgen -v コマンド・ライン・オプションで指定したパブリケーションが Ultra Light ジェネレータによって UL_PUB_pubname という名前の大文字の定数として識別されます。pubname は、-v オプションに指定する名前です。

たとえば、次のコマンド・ラインは、定数 UL_PUB_SALES で識別されるパブリケーションを生成します。

```
ulgen -v sales ...
```

同期時には、publication パラメータを「**パブリケーション・マスク**」、つまり、パブリケーション定数を OR で結合したリストに設定します。次に例を示します。

```
ul_synch_info info;
 // ...
 info.publication = UL_PUB_MYPUB1 | UL_PUB_MYPUB2 ;
```

特殊なパブリケーション・マスク **UL_SYNC_ALL** は、パブリケーションに含まれるかどうかに関係なく、データベース内のすべてのテーブルを表します。マスク **UL_SYNC_ALL_PUBS** は、データベース内のパブリケーションに含まれるすべてのテーブルを表します。

参照

- 『Mobile Link クライアント』> 「publication 同期パラメータ」
- 『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light ジェネレータ」
- 『Mobile Link クライアント』> 「別々に同期するためのデータ・セットの設計」

resume_partial_download 同期パラメータ

機能　ダウンロード時に通信障害によって中断された同期を再開します。

デフォルト　このパラメータはブール値であり、デフォルトは false です。

使用法　次のようにパラメータを設定します。

```
ul_synch_info info;
 // ...
 info.resume_partial_download = ul_true;
```

参照

- 『Mobile Link 管理ガイド』> 「失敗したダウンロードの再開」
- 『Mobile Link クライアント』> 「Resume Partial Download 同期パラメータ」

security 同期パラメータ

機能

Mobile Link 同期サーバとのメッセージ交換に Certicom 暗号化テクノロジを使用するように、Ultra Light クライアントを設定します。

別途ライセンスを取得できるオプションが必要
Certicom テクノロジを使用するには、別途ライセンスを取得できる SQL Anywhere Studio セキュリティ・オプションを入手する必要があります。このセキュリティ・オプションは、輸出規制対象品目です。このオプションの詳細については、『SQL Anywhere Studio の紹介』> 「SQL Anywhere Studio へようこそ」を参照してください。

デフォルト

security パラメータは、デフォルトでトランスポート・レイヤ・セキュリティなしに対応する null に設定されます。

使用法

同期ストリームに加えてセキュリティ・ストリームを指定します。指定可能な値は次のとおりです。

- **ULSecureCerticomTLSStream()** Certicom によって提供される楕円曲線暗号化トランスポート・レイヤ・セキュリティ。
- **ULSecureRSATLSStream()** Certicom によって提供される RSA トランスポート・レイヤ・セキュリティ。

リンクする必要があるライブラリなどの詳細については、『Mobile Link クライアント』> 「Security 同期パラメータ」を参照してください。

参照

- 『Mobile Link クライアント』> 「Security 同期パラメータ」
- 『Mobile Link 管理ガイド』> 「Mobile Link トランスポート・レイヤ・セキュリティ」.

security_parms 同期パラメータ

機能

トランスポート・レイヤ・セキュリティを使用する場合に必要なパラメータを設定します。このパラメータは、**security** パラメータとともに使用してください。

詳細については、「security 同期パラメータ」526 ページを参照してください。

使用法

ULSecureCerticomTLSStream() と ULSecureRSATLSStream() セキュリティ・パラメータは、次のオプションのパラメータで構成される文字列を使用します。オプションのパラメータは、セミコロンで区切られた文字列で指定します。

- **certificate_company**　証明書に記されている組織フィールドがこの値と一致する場合にだけ、Ultra Light アプリケーションはサーバ証明書を受け入れます。デフォルトでは、このフィールドはチェックされていません。
- **certificate_unit**　証明書に記されている部署フィールドがこの値と一致する場合にだけ、Ultra Light アプリケーションはサーバ証明書を受け入れます。デフォルトでは、このフィールドはチェックされていません。
- **certificate_name**　証明書に記されている共通名フィールドがこの値と一致する場合にだけ、Ultra Light アプリケーションはサーバ証明書を受け入れます。デフォルトでは、このフィールドはチェックされていません。

次に例を示します。

```
ul_synch_info info;
 ...
 info.stream = ULSocketStream();
 info.security = ULSecureCerticomTLSStream();
 info.security_parms =
      UL_TEXT( "certificate_company=Sybase" )
      UL_TEXT( ";" )
      UL_TEXT( "certificate_unit=Sales" );
```

security_parms パラメータは文字列です。デフォルトは null です。

セキュリティ機能のある同期を実行する場合は、Ultra Light ジェネレータの `-r` コマンド・ライン・オプションも使用してください。詳細については、『Ultra Light データベース・ユーザーズ・ガイド』> 「Ultra Light ジェネレータ」を参照してください。

参照

- 『Mobile Link クライアント』> 「Security Parameters 同期パラメータ」

send_column_names 同期パラメータ

機能

send_column_names を **ul_true** に設定すると、Ultra Light は各カラム名を Mobile Link 同期サーバに送信します。デフォルトでは、Ultra Light はカラム名を送信しません。

このパラメータは、通常は同期スクリプトを自動的に生成するために Mobile Link 同期サーバで `-za` または `-ze` スイッチとともに使用されます。

参照

- 『Mobile Link クライアント』> 「Send Column Names 同期パラメータ」
- 『Mobile Link 管理ガイド』> 「-za オプション」

send_download_ack 同期パラメータ

機能

このブール・パラメータを **true** に設定すると、Mobile Link 同期サーバに対して、クライアントからダウンロード確認が提供されることを通知します。

クライアントがダウンロード確認を送信する場合、Mobile Link 同期サーバのワーカ・スレッドは、クライアントがダウンロードを適用するまで待機します。クライアントがダウンロード確認を送信しない場合、Mobile Link 同期サーバは、次の同期のため、まもなく解放されます。

デフォルト

デフォルト設定は **false** です。

参照

- 『Mobile Link クライアント』> 「Send Download Acknowledgement 同期パラメータ」

stream 同期パラメータ

機能

同期に使用するように Mobile Link 同期ストリームを設定します。

詳細については、「stream_parms 同期パラメータ」538 ページを参照してください。

デフォルト

パラメータにはデフォルト値がないので、明示的に設定してください。

使用法

```
//Embedded SQL
 ul_synch_info info;
 ...
 info.stream = ULSocketStream();

//Static C++ API
 Connection conn;
 auto ul_synch_info info;
 ...
 conn.InitSynchInfo( &info );
 info.stream = ULSocketStream();
```

ストリームのタイプでパラメータが必要な場合は、**stream_parms** パラメータを使用してパラメータを渡します。パラメータが不要なときは、**stream_parms** パラメータを null に設定します。

stream 関数には以下の種類がありますが、すべてのターゲット・プラットフォームですべての関数を使用できるわけではありません。

ストリーム	説明
ULActiveSyncStream()	ActiveSync 同期 (Windows CE のみ) ストリーム・パラメータのリストについては、『Mobile Link クライアント』> 「ActiveSync プロトコル・オプション」を参照してください。

ストリーム	説明
ULHTTPStream()	HTTP を介して同期 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 同期サーバは 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。 ストリーム・パラメータのリストについては、『Mobile Link クライアント』> 「HTTP プロトコル・オプション」を参照してください。
ULHTTPSStream()	HTTPS 同期ストリームを介して同期 HTTPS ストリームは、基本のプロトコルとして SSL または TLS を使用します。これは、インターネット・プロトコル (HTTP と TCP/IP) 上で動作します。 HTTPS ストリームでは、Certicom が提供するテクノロジを使用する必要があります。Certicom テクノロジを使用するには、別途ライセンスを取得できる SQL Anywhere Studio セキュリティ・オプションを入手する必要があります。このセキュリティ・オプションは、輸出規制対象品目です。このオプションの詳細については、『SQL Anywhere Studio の紹介』>「SQL Anywhere Studio へようこそ」を参照してください。 ストリーム・パラメータのリストについては、『Mobile Link クライアント』> 「HTTPS プロトコル・オプション」を参照してください。
ULSocketStream()	TCP/IP を介して同期 ストリーム・パラメータのリストについては、『Mobile Link クライアント』> 「TCP/IP プロトコル・オプション」を参照してください。

参照

- 『Mobile Link クライアント』> 「Stream Type 同期パラメータ」

stream_error 同期パラメータ

機能

通信エラー・レポート情報を保持する構造体を設定します。

アクセス・メソッド

この機能は Java アプリケーションには使用できません。

デフォルト

パラメータにはデフォルト値がないので、明示的に設定してください。

説明

stream_error フィールドは **ul_stream_error** 型の構造体です。

```
typedef struct ss_error {
     ss_stream_id         stream_id;
     ss_stream_context    stream_context;
     ss_error_code        stream_error_code;
     asa_uint32           system_error_code;
     rp_char              *error_string;
     asa_uint32           error_string_length;
 } ss_error, *p_ss_error;
```

この構造体は、SQL Anywhere ディレクトリの *h* サブディレクトリにある *sserror.h* 内で定義されています。

ul_stream_error のフィールドは、次のとおりです。

- **stream_id** エラーをレポートするネットワーク・レイヤ。この列挙は *sserror.h* にリストされています。有効な定数を次に示します。

定数	値
STREAM_ID_TCPIP	0
STREAM_ID_PALM_CONDUIT	3
STREAM_ID_PALM_SS	4
STREAM_ID_HTTP	7
STREAM_ID_HTTPS	8
STREAM_ID_SECURE	10
STREAM_ID_CERTICOM	12

定数	値
STREAM_ID_JAVA_CERTICOM	13
STREAM_ID_CERTICOM_SSL	14
STREAM_ID_CERTICOM_TLS	15
STREAM_ID_WIRESTRM	16
STREAM_ID_ACTIVESYNC	23
STREAM_ID_RSA_TLS	24
STREAM_ID_JAVA_RSA	25

- **stream_context**　開く、読み込み、または書き込みなど、実行される基本的なネットワーク・オペレーション。詳細については、*sserror.h* を参照してください。

- **stream_error_code**　ストリーム自体によってレポートされるエラー。**stream_error_code** は、**ss_error_code** 型です。ストリーム・エラー・コードは、いずれもプレフィクス STREAM_ERROR_ で始まります。たとえば、書き込みエラーの場合は STREAM_ERROR_WRITE です。

  エラー番号のリストについては、『ASA エラー・メッセージ』>「Mobile Link 通信エラー・メッセージ」を参照してください。エラー・コードのサフィックスについては、*sserror.h* を参照してください。

  このバージョンでは、各番号に関連した定数を探すには、*sserror.h* 内でプレフィックス DO_STREAM_Error が付いている行の番号を数えます。たとえば、エラー番号 10 の定数を探すには、次のように *sserror.h* 内で 10 番目の DO_STREAM_ERROR エントリを探します。

  ```
  DO_STREAM_ERROR( WRITE )
  ```

  したがって、このエラーに関連した定数は STREAM_ERROR_WRITE です。

- **system_error_code** システム固有のエラー・コード。エラー・コードの詳細については、プラットフォームのマニュアルを参照してください。Windows プラットフォームの場合は、Microsoft Developer Network (MSDN) マニュアルを参照してください。

  Windows における一般的なシステム・エラーを次に示します。

  - **10048 (WSAADDRINUSE)** アドレスがすでに使用されています。
  - **10053 (WSAECONNABORTED)** ソフトウェアが接続のアボートを引き起こしました。
  - **10054 (WSAECONNRESET)** 通信の他方の側がソケットを閉じました。
  - **10060 (WSAETIMEDOUT)** 接続がタイムアウトしました。
  - **10061 (WSAECONNREFUSED)** 接続が拒否されました。通常、Mobile Link サーバが稼動していないか、指定されたポートで受信していません。

  完全なリストについては、Microsoft Developer Network Web サイトを参照してください。

- **error_string** アプリケーションが提供するエラー・メッセージ。文字列は空の場合もあれば、空でない場合もあります。空でない error_string は、stream_error_code とともに情報を提供します。たとえば、書き込みエラー (エラー・コード 9) の場合、エラー文字列は、書き込もうとしたバイト数を示します。

使用法

次のように SQLE_COMMUNICATIONS_ERROR をチェックします。

```
Connection conn;
 auto ul_synch_info info;
 ...
 conn.InitSynchInfo( &info );
 info.stream_error.error_string = error_buff;
 info.stream_error.error_string_length =
```

```
                    sizeof( error_buff );
if( !conn.Synchronize( &synch_info ) ){
    if( SQLCODE == SQLE_COMMUNICATIONS_ERROR ){
        printf( error_buff );
    // more error handline here
```

参照

- 『Mobile Link クライアント』> 「Stream Error 同期パラメータ」

stream_parms 同期パラメータ

機能

同期ストリームを設定するネットワーク・プロトコルのオプションを設定します。

オプションの割り当てのリストで、セミコロンで区切られます。各割り当ては *keyword*=*value* の形式で、許可されるキーワード・セットは同期ストリームよって異なります。

各ストリームに使用できるオプションのリストについては、次の各項を参照してください。

- 『Mobile Link クライアント』> 「ActiveSync プロトコル・オプション」
- 『Mobile Link クライアント』> 「HotSync プロトコル・オプション」
- 『Mobile Link クライアント』> 「HTTP プロトコル・オプション」
- 『Mobile Link クライアント』> 「HTTPS プロトコル・オプション」
- 『Mobile Link クライアント』> 「TCP/IP プロトコル・オプション」

デフォルト

これはオプションの文字列パラメータです。デフォルトは null です。

使用法

次のようにパラメータを設定します。

```
ul_synch_info info;
 // ...
 info.stream_parms= UL_TEXT( "host=myserver;port=2439"
);
```

参照

- 『Mobile Link クライアント』> 「Stream Parameters 同期パラメータ」
- 『Mobile Link クライアント』> 「Ultra Light 同期クライアントのネットワーク・プロトコルのオプション」.

upload_ok 同期パラメータ

機能

Mobile Link アップロードのステータスをレポートします。Mobile Link 同期サーバが、この情報をクライアントに提供します。

このパラメータは読み込み専用です。

使用法

同期後、**upload_ok** パラメータには、アップロードが成功した場合は **true** が入ります。それ以外の場合は **false** が入ります。

このパラメータにアクセスするには、次のように入力します。

```
ul_synch_info info;
 // ...
 returncode = info.upload_ok;
```

参照

- 『Mobile Link クライアント』> 「Upload OK 同期パラメータ」

upload_only 同期パラメータ

機能

現在の同期中にダウンロードは発生しないことを示します。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。true に設定すると、クライアントは Mobile Link 同期サーバからのアップロード確認を待ってから、同期セッションを正常終了します。

デフォルト

このパラメータはオプションのブール値であり、デフォルトは false です。

使用法

次のように true に設定します。

```
ul_synch_info info;
 // ...
 info.upload_only = ul_true;
```

参照

- 『Mobile Link クライアント』> 「Upload Only 同期パラメータ」
- 『Mobile Link クライアント』> 「優先度の高い変更の同期」
- 「download_only 同期パラメータ」515 ページ

user_data 同期パラメータ

機能

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。

使用法

アプリケーション固有の情報を使用できるようにするには、同期 observer コールバック関数 **observer** を実装する際に、**user_data** を使用して情報を提供します。

参照

- 『Mobile Link クライアント』> 「User Data 同期パラメータ」
- 「observer 同期パラメータ」520 ページ

user_name 同期パラメータ

機能

ユーザ名を指定する文字列であり、Mobile Link 同期サーバが Mobile Link クライアントをユニークに識別するために使用します。Mobile Link では、この値を使用して、ダウンロードする内容の決定、同期ステータスの記録、同期中の割り込みからの復帰を行います。

デフォルト

これは必須の文字列パラメータです。

使用法

次のようにパラメータを設定します。

```
ul_synch_info info;
 // ...
 info.user_name= UL_TEXT( "mluser" );
```

参照

- 『Mobile Link クライアント』> 「User Name 同期パラメータ」
- 『Mobile Link クライアント』> 「Mobile Link ユーザの認証」.
- 『Mobile Link クライアント』> 「Mobile Link ユーザ」.

version 同期パラメータ

機能

統合データベースの同期スクリプトは、それぞれバージョン文字列でマーク付けされます。たとえば、異なる文字列バージョンによって特定される 2 種類の **download_cursor** スクリプトがあります。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。

デフォルト

これは文字列パラメータです。デフォルトでは、Mobile Link のデフォルト・バージョン文字列です。

使用法

次のようにパラメータを設定します。

```
ul_synch_info info;
 // ...
 info.version = UL_TEXT( "default" );
```

参照

- 『Mobile Link クライアント』> 「Version 同期パラメータ」
- 『Mobile Link 管理ガイド』> 「スクリプト・バージョン」.

索引

記号

#define
Ultra Light アプリケーション 268
~ULSqlcaWrap 関数
Ultra Light C++ コンポーネント API 283
~ULValue 関数
Ultra Light C++ コンポーネント API 371
10054
同期ストリーム・システム・エラー (Ultra Light クライアント) 536
10 進数 Ultra Light Embedded SQL データ型 83
16 ビット符号付き整数 Ultra Light Embedded SQL データ型 82
32 ビット符号付き整数 Ultra Light Embedded SQL データ型 82
4 バイト浮動小数点数 Ultra Light Embedded SQL データ型 83
8 バイト浮動小数点数 Ultra Light Embedded SQL データ型 83

A

ActiveSync
MFC Ultra Light アプリケーション 173
ULIsSynchronizeMessage 関数 466
Ultra Light アプリケーションへの追加 171
Ultra Light メッセージ 268
WindowProc 関数 172
クラス名 168
サポートされるバージョン 171
説明 171
AddRef 関数
Ultra Light C++ コンポーネント API 329
AES 暗号化アルゴリズム
Ultra Light Embedded SQL データベース 105
Ultra Light 静的型 C++ API データベース 59
AfterLast 関数
Ultra Light C++ コンポーネント API 299
AfterLast メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 397
an_SQL_code Ultra Light データ型
Ultra Light 静的型 C++ API 376
auth_parms 同期パラメータ
説明 (Ultra Light C/C++) 510
auth_status 同期パラメータ
説明 (Ultra Light C/C++) 511
auth_value 同期パラメータ
説明 (Ultra Light C/C++) 512
autoCommit モード
Ultra Light C++ コンポーネント 38

B

BeforeFirst 関数
Ultra Light C++ コンポーネント API 299
BeforeFirst メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 397

C

C++ API 3
cache_size 永続ストレージ・パラメータ
説明 270
Certicom
Power PC では使用不可 155
ChangeEncryptionKey 関数
Ultra Light C++ コンポーネント API 287
changeEncryptionKey メソッド
Ultra Light Embedded SQL 106

Ultra Light 静的型 C++ API 60
checkpoint_store 同期パラメータ
Mobile Link 同期 513
ClientParms レジストリ・エントリ
Mobile Link コンジット 154
CLOSE 文
Ultra Light Embedded SQL 97
Close メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 378
Close メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 398
Close メソッド (ULData クラス)
Palm Computing Platform で使用しない 411
Ultra Light 静的型 C++ API 411
CodeWarrior
Ultra Light C/C++ 開発 139
Ultra Light のステーショナリ 140
Ultra Light プラグインのインストール 139
Ultra Light プラグインの使用 142
Ultra Light プロジェクトの作成 140
拡張モード Ultra Light アプリケーション 145
プロジェクトの変換 142
Commit 関数
Ultra Light C++ コンポーネント API 287
GetReferencedIndexName 関数
Ultra Light C++ コンポーネント API 315
commit メソッド
Ultra Light C++ コンポーネント 38
Commit メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 379
Connection オブジェクト
Ultra Light C++ コンポーネント 19
CONNECT 文
Ultra Light Embedded SQL 79
CountUploadRows 関数
Ultra Light C++ コンポーネント API 287
CountUploadRows メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 379
CreateAndOpenDatabase 関数
Ultra Light C++ コンポーネント API 305
CustDB アプリケーション
Palm Computing Platform 用に構築 143
Windows CE での構築 163

D

DatabaseManager オブジェクト
Ultra Light C++ コンポーネント 19
DatabaseSchema オブジェクト
Ultra Light C++ コンポーネント 39
db_fini 関数
Palm Computing Platform で使用しない 446
Ultra Light Embedded SQL 構文 446
db_init 関数
Ultra Light Embedded SQL 構文 447
db_start_database 関数
Ultra Light Embedded SQL 構文 448
db_stop_database 関数
Ultra Light Embedded SQL 構文 449
DECL_BINARY マクロ
Ultra Light Embedded SQL 82
DECL_DATETIME データ型
Ultra Light 静的型 C++ API 376
DECL_DATETIME マクロ
Ultra Light Embedded SQL 82
DECL_DECIMAL マクロ
Ultra Light Embedded SQL 82
DECL_FIXCHAR マクロ
Ultra Light Embedded SQL 82
DECL_VARCHAR マクロ
Ultra Light Embedded SQL 82
DECLARE 文
Ultra Light Embedded SQL 97
DeleteAllRows 関数
Ultra Light C++ コンポーネント API 339
DeleteAllRows メソッド (ULTable クラス)
Ultra Light 静的型 C++ API 424
Delete 関数
Ultra Light C++ コンポーネント API 338
Delete メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 398
disable_concurrency 同期パラメータ
Ultra Light C/C++ 514

DML
Ultra Light C++ コンポーネント 24
download_only 同期パラメータ
説明 (Ultra Light C/C++) 515
DropDatabase 関数
Ultra Light C++ コンポーネント API 306
Drop メソッド (ULData クラス)
Ultra Light 静的型 C++ API 412
DT_BINARY Ultra Light Embedded SQL データ型 86
DT_LONGVARCHAR Ultra Light Embedded SQL データ型 86

E

Embedded SQL
Ultra Light C++ コンポーネントとの結合 132
Ultra Light 関数 445
Ultra Light サンプル・プログラム 219
Ultra Light チュートリアル 216
Ultra Light データ・アクセス 73
Ultra Light の使用 445
Ultra Light ホスト変数 81
Ultra Light 利点 8
カーソル (Ultra Light) 97
データのフェッチ (Ultra Light) 96
Embedded SQL ライブラリ関数
ULActiveSyncStream (Ultra Light) 450
ULChangeEncryptionKey (Ultra Light) 451
ULClearEncryptionKey (Ultra Light) 452
ULCountUploadRows (Ultra Light) 453
ULDropDatabase (Ultra Light) 455
ULEnableFileDB (Ultra Light) 253
ULEnableGenericSchema (Ultra Light) 254
ULEnablePalmRecordDB (Ultra Light) 256
ULEnableStrongEncryption (Ultra Light) 257
ULEnableUserAuthentication (Ultra Light) 258
ULGetDatabaseID (Ultra Light) 457
ULGetLastDownloadTime (Ultra Light) 458
ULGetSynchResult (Ultra Light) 460
ULGlobalAutoincUsage (Ultra Light) 462
ULGrantConnectTo (Ultra Light) 463
ULHTTPSStream (Ultra Light) 464
ULHTTPStream (Ultra Light) 465
ULIsSynchronizeMessage (Ultra Light) 466
ULPalmDBStream (Ultra Light) 467
ULPalmExit (Ultra Light) 468
ULPalmLaunch (Ultra Light) 469
ULResetLastDownloadTime (Ultra Light) 470
ULRetrieveEncryptionKey (Ultra Light) 471
ULRevokeConnectFrom (Ultra Light) 472
ULSaveEncryptionKey (Ultra Light) 474
ULSetDatabaseID (Ultra Light) 475
ULSetSynchInfo (Ultra Light) 476
ULSocketStream (Ultra Light) 477
ULSynchronize (Ultra Light) 478
eMbedded Visual C++
Ultra Light の取得 160
EXEC SQL
Ultra Light Embedded SQL の開発 76
ExecuteQuery 関数
Ultra Light C++ コンポーネント API 319
ExecuteStatement 関数
Ultra Light C++ コンポーネント API 319
Execute メソッド (生成文のクラス)
Ultra Light 静的型 C++ API 435

F

FETCH 文
Ultra Light Embedded SQL 96, 97
Finalize 関数
Ultra Light C++ コンポーネント API 278
FindBegin 関数
Ultra Light C++ コンポーネント API 340
FindFirst 関数
Ultra Light C++ コンポーネント API 340
FindFirst メソッド (ULTable クラス)
Ultra Light 静的型 C++ API 424
FindLast 関数
Ultra Light C++ コンポーネント API 340
FindLast メソッド (ULTable クラス)
Ultra Light 静的型 C++ API 425
FindNext 関数
Ultra Light C++ コンポーネント API 341
FindNext メソッド (ULTable クラス)

Ultra Light 静的型 C++ API 426
FindPrevious 関数
Ultra Light C++ コンポーネント API 341
FindPrevious メソッド (ULTable クラス)
Ultra Light 静的型 C++ API 427
Find 関数
Ultra Light C++ コンポーネント API 339
find メソッド
Ultra Light C++ コンポーネント 34
Find メソッド (ULTable クラス)
Ultra Light 静的型 C++ API 424
First 関数
Ultra Light C++ コンポーネント API 300
First メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 399

G

GetBinaryLength 関数
Ultra Light C++ コンポーネント API 358
GetBinary 関数
Ultra Light C++ コンポーネント API 357, 358
GetByteChunk 関数
Ultra Light C++ コンポーネント API 332
GetCA 関数
Ultra Light C++ コンポーネント API 279
GetCA メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 380
GetCollationName 関数
Ultra Light C++ コンポーネント API 309
GetColumnCount 関数
Ultra Light C++ コンポーネント API 314, 325
GetColumnCount メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 401
GetColumnDefault 関数
Ultra Light C++ コンポーネント API 349
GetColumnID 関数
Ultra Light C++ コンポーネント API 325
GetColumnName 関数
Ultra Light C++ コンポーネント API 314, 326
GetColumnPrecision 関数
Ultra Light C++ コンポーネント API 326
GetColumnScale 関数
Ultra Light C++ コンポーネント API 327
GetColumnSize 関数
Ultra Light C++ コンポーネント API 327
GetColumnSize メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 401
GetColumnSQLType 関数
Ultra Light C++ コンポーネント API 326
GetColumnSQLType メソッド (ULCursor クラス) 402
GetColumnType 関数
Ultra Light C++ コンポーネント API 328
GetColumnType メソッド (ULCursor クラス) 401
GetColumn メソッド (生成された結果セットのクラス)
Ultra Light 静的型 C++ API 432
GetColumn メソッド (生成されたテーブル・クラス)
Ultra Light 静的型 C++ API 440
GetConnectionNum 関数
Ultra Light C++ コンポーネント API 288
GetConnection 関数
Ultra Light C++ コンポーネント API 330
GetDatabaseID メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 380
GetDatabaseProperty 関数
Ultra Light C++ コンポーネント API 288
GetGlobalAutoincPartitionSize 関数
Ultra Light C++ コンポーネント API 350
GetID 関数
Ultra Light C++ コンポーネント API 315, 350
GetIFace 関数
Ultra Light C++ コンポーネント API 330
GetIndexCount 関数
Ultra Light C++ コンポーネント API 350
GetIndexName 関数
Ultra Light C++ コンポーネント API 351
GetIndexSchema 関数
Ultra Light C++ コンポーネント API 351

GetLastDownloadTime 関数
Ultra Light C++ コンポーネント API 288
GetLastDownloadTime メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 381
GetLastIdentity 関数
Ultra Light C++ コンポーネント API 289
GetLastIdentity メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 380
GetLength 関数
Ultra Light C++ コンポーネント API 333
GetName 関数
Ultra Light C++ コンポーネント API 315, 351
GetNewUUID 関数
Ultra Light C++ コンポーネント API 289
GetOptimalIndex 関数
Ultra Light C++ コンポーネント API 351
GetParameterCount 関数
Ultra Light C++ コンポーネント API 279
GetParameter 関数
Ultra Light C++ コンポーネント API 279
GetPlan 関数
Ultra Light C++ コンポーネント API 320
GetPrimaryKey 関数
Ultra Light C++ コンポーネント API 352
GetPublicationCount 関数
Ultra Light C++ コンポーネント API 309
GetPublicationID 関数
Ultra Light C++ コンポーネント API 310
GetPublicationMask 関数
Ultra Light C++ コンポーネント API 289, 310
GetPublicationName 関数
Ultra Light C++ コンポーネント API 310
GetReferencedTableName 関数
Ultra Light C++ コンポーネント API 316
GetRowCount 関数
Ultra Light C++ コンポーネント API 300
GetRowCount メソッド (ULTable クラス)
Ultra Light 静的型 C++ API 428
GetSchema 関数
Ultra Light C++ コンポーネント API 290, 320, 323, 342
GetSignature 関数
Ultra Light C++ コンポーネント API 311
GetSizeColumn メソッド (生成されたテーブル・クラス)
Ultra Light 静的型 C++ API 440
GetSqlca 関数
Ultra Light C++ コンポーネント API 290
GetSQLCode 関数
Ultra Light C++ コンポーネント API 280
GetSQLCode メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 382
GetSQLCode メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 403
GetSQLCount 関数
Ultra Light C++ コンポーネント API 280, 288
GetSQLErrorOffset 関数
Ultra Light C++ コンポーネント API 280
GetState 関数
Ultra Light C++ コンポーネント API 300
GetStreamReader 関数
Ultra Light C++ コンポーネント API 301
GetStreamWriter 関数
Ultra Light C++ コンポーネント API 320, 342
GetStringChunk 関数
Ultra Light C++ コンポーネント API 333, 334
GetStringLength 関数
Ultra Light C++ コンポーネント API 360
GetString 関数
Ultra Light C++ コンポーネント API 359
GetSuspend 関数
Ultra Light C++ コンポーネント API 290, 301
GetSynchResult 関数
Ultra Light C++ コンポーネント API 290
GetSynchResult メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 382
GetTableCount 関数
Ultra Light C++ コンポーネント API 311
GetTableName 関数
Ultra Light C++ コンポーネント API 311, 316
GetTableSchema 関数

Ultra Light C++ コンポーネント API 311
GetUploadUnchangedRows 関数
Ultra Light C++ コンポーネント API 352
GetUtilityULValue 関数
Ultra Light C++ コンポーネント API 290
Get 関数
Ultra Light C++ コンポーネント API 300
Get メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 400
Get メソッド (生成されたテーブル・クラス)
Ultra Light 静的型 C++ API 438
GlobalAutoincUsage 関数
Ultra Light C++ コンポーネント API 291
GlobalAutoincUsage メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 384
GrantConnectTo 関数
Ultra Light C++ コンポーネント API 291
grantConnectTo メソッド
Ultra Light C++ コンポーネント開発 42
GrantConnectTo メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 385

H

HasResultSet 関数
Ultra Light C++ コンポーネント API 320
HotSync 同期
Palm Computing Platform 153
hpp ファイル
Ultra Light 静的型 C++ API 374
HTTPS 同期
Ultra Light の Palm Computing Platform 155
HTTP 同期
Ultra Light の Palm Computing Platform 155

I

ignored_rows 同期パラメータ
Ultra Light C/C++ 517
INCLUDE 文
SQLCA (Ultra Light) 77
InDatabase 関数
Ultra Light C++ コンポーネント API 360
IndexSchema オブジェクト
Ultra Light C++ コンポーネント開発 39
Initialize 関数
Ultra Light C++ コンポーネント API 280
Initialize メソッド (ULData クラス)
Ultra Light 静的型 C++ API 413
InitSynchInfo 関数
Ultra Light C++ コンポーネント API 291
InitSynchInfo メソッド
説明 62
InitSynchInfo メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 385
InPublication 関数
Ultra Light C++ コンポーネント API 352
InsertBegin 関数
Ultra Light C++ コンポーネント API 342
Insert 関数
Ultra Light C++ コンポーネント API 342
Insert メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 403
IsCaseSensitive 関数
Ultra Light C++ コンポーネント API 312
IsColumnAutoinc 関数
Ultra Light C++ コンポーネント API 353
IsColumnCurrentDate 関数
Ultra Light C++ コンポーネント API 353
IsColumnCurrentTimestamp 関数
Ultra Light C++ コンポーネント API 354
IsColumnCurrentTime 関数
Ultra Light C++ コンポーネント API 353
IsColumnDescending 関数
Ultra Light C++ コンポーネント API 316
IsColumnGlobalAutoinc 関数
Ultra Light C++ コンポーネント API 354
IsColumnInIndex 関数
Ultra Light C++ コンポーネント API 354

IsColumnNewUUID 関数
Ultra Light C++ コンポーネント API 355
IsColumnNullable 関数
Ultra Light C++ コンポーネント API 355
IsForeignKeyCheckOnCommit 関数
Ultra Light C++ コンポーネント API 317
IsForeignKeyNullable 関数
Ultra Light C++ コンポーネント API 317
IsForeignKey 関数
Ultra Light C++ コンポーネント API 316
IsNeverSynchronized 関数
Ultra Light C++ コンポーネント API 355
IsNull 関数
Ultra Light C++ コンポーネント API 301, 361
IsOpen メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 386
IsOpen メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 404
IsOpen メソッド (ULData クラス)
Ultra Light 静的型 C++ API 414
IsPrimaryKey 関数
Ultra Light C++ コンポーネント API 317
IsUniqueIndex 関数
Ultra Light C++ コンポーネント API 317
IsUniqueKey 関数
Ultra Light C++ コンポーネント API 317

K

keep_partial_download 同期パラメータ
Ultra Light C/C++ 516

L

LastCodeOK 関数
Ultra Light C++ コンポーネント API 281
LastCodeOK メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 386
LastCodeOK メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 404
LastFetchOK 関数
Ultra Light C++ コンポーネント API 281
LastFetchOK メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 387, 405
Last 関数
Ultra Light C++ コンポーネント API 301
Last メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 404
LAUNCH_SUCCESS_FIRST
Ultra Light 静的型 C++API 418
LookupBackward 関数
Ultra Light C++ コンポーネント API 343
LookupBackward メソッド (ULTable クラス)
Ultra Light 静的型 C++ API 429
LookupBegin 関数
Ultra Light C++ コンポーネント API 344
LookupForward 関数
Ultra Light C++ コンポーネント API 344
LookupForward メソッド (ULTable クラス)
Ultra Light 静的型 C++ API 430
Lookup 関数
Ultra Light C++ コンポーネント API 343
lookup メソッド
Ultra Light C++ コンポーネント 34
Lookup メソッド (ULTable クラス)
Ultra Light 静的型 C++ API 428

M

makefiles
Ultra Light Embedded SQL 125
Metrowerks CodeWarrior
Ultra Light プロジェクトの作成 140
MFC
Ultra Light 用の ActiveSync 173
moveFirst メソッド (Table オブジェクト)
Ultra Light C++ コンポーネント開発 27, 30
moveNext メソッド (Table オブジェクト)
Ultra Light C++ コンポーネント開発 27, 30

N

new_password 同期パラメータ
Ultra Light C/C++ 518
Next 関数
Ultra Light C++ コンポーネント API 302
Next メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 405
NULL
Ultra Light インジケータ変数 93
Ultra Light 静的型 C++ API 400
NULL で終了された TCHAR 文字列 Ultra Light SQL データ型 84
NULL で終了された UNICODE 文字列 Ultra Light SQL データ型 83
NULL で終了された WCHAR 文字列 Ultra Light SQL データ型 83
NULL で終了された文字列 Ultra Light Embedded SQL データ型 83
NULL で終了されたワイド文字列 Ultra Light SQL データ型 83
num_auth_parms 同期パラメータ
num_auth_parms (Ulrta Light C/C++) 519

O

observer 同期パラメータ
Ultra Light Embedded SQL の例 115
Ultra Light 静的型 C++ API の例 68
説明 (Ultra Light C/C++) 520
openByIndex メソッド (Table オブジェクト)
Ultra Light C++ コンポーネント開発 27
OpenConnection 関数
Ultra Light C++ コンポーネント API 306
OpenTableWithIndex 関数
Ultra Light C++ コンポーネント API 292
OpenTable 関数
Ultra Light C++ コンポーネント API 292
OPEN 文
Ultra Light Embedded SQL 97
open メソッド (Table オブジェクト)
Ultra Light C++ コンポーネント開発 27
Open メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 388
Open メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 405
Open メソッド (ULData クラス)
Ultra Light 静的型 C++ API 414
Open メソッド (生成された結果セットのクラス)
Ultra Light 静的型 C++ API 433
Open メソッド (生成されたテーブル・クラス)
Ultra Light 静的型 C++ API 441
operator= 関数
Ultra Light C++ コンポーネント API 371
operator bool 関数
Ultra Light C++ コンポーネント API 368
operator DECL_DATETIME 関数
Ultra Light C++ コンポーネント API 368
operator double 関数
Ultra Light C++ コンポーネント API 369
operator float 関数
Ultra Light C++ コンポーネント API 369
operator int 関数
Ultra Light C++ コンポーネント API 369
operator long 関数
Ultra Light C++ コンポーネント API 369
operator short 関数
Ultra Light C++ コンポーネント API 369
operator ul_s_big 関数
Ultra Light C++ コンポーネント API 369
operator ul_u_big 関数
Ultra Light C++ コンポーネント API 370
operator unsigned char 関数
Ultra Light C++ コンポーネント API 370
operator unsigned int 関数
Ultra Light C++ コンポーネント API 370
operator unsigned long 関数
Ultra Light C++ コンポーネント API 370
operator unsigned short 関数
Ultra Light C++ コンポーネント API 370

P

Palm Computing Platform
Ultra Light C++ のアプリケーション 138
Ultra Light アプリケーションのインストール 156
Ultra Light 静的型 C++ API 389, 407, 419
Ultra Light の HotSync 同期 153
Ultra Light の HTTP 同期 155
Ultra Light の TCP/IP 同期 155
セキュリティ 155
セグメント 149, 150, 151
バージョン 4.0 253, 256
ファイルベースのデータ・ストア 253
プラットフォーム稼働条件 138
レコードベースのデータ・ストア 256
PalmExit メソッド (ULData クラス)
Ultra Light 静的型 C++ API 416
PalmLaunch メソッド (ULData クラス)
Ultra Light 静的型 C++ API 417
partial_download_retained 同期パラメータ
Ultra Light C/C++ 521
password 同期パラメータ
説明 (Ultra Light C/C++) 522
PATH 環境変数
HotSync 138
PilotMain 関数
Ultra Light アプリケーション 148, 153
ping 同期パラメータ
説明 (Ultra Light C/C++) 523
preparedStatement クラス
Ultra Light C++ コンポーネント 24
PrepareStatement 関数
Ultra Light C++ コンポーネント API 293
Previous 関数
Ultra Light C++ コンポーネント API 302
Previous メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 406
PublicationSchema オブジェクト
Ultra Light C++ コンポーネント開発 40
publication 同期パラメータ
説明 (Ultra Light C/C++) 524

R

Relative 関数
Ultra Light C++ コンポーネント API 302
Relative メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 407
Release 関数
Ultra Light C++ コンポーネント API 330
Reopen メソッド
Ultra Light C/C++ 147
Reopen メソッド (ULConnection クラス)
旧式な関数 389
Reopen メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 407
Reopen メソッド (ULData クラス)
旧式な関数 419
ResetLastDownloadTime 関数
Ultra Light C++ コンポーネント API 293
ResetLastDownloadTime メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 390
resume_partial_download 同期パラメータ
Ultra Light C/C++ 525
RevokeConnectFrom 関数
Ultra Light C++ コンポーネント API 293
RevokeConnectFrom メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 390
revokeConnectionFrom メソッド
Ultra Light C++ コンポーネント開発 42
RollbackPartialDownload 関数
Ultra Light C++ コンポーネント API 294
Ultra Light 静的型 C++ API 391
Rollback 関数
Ultra Light C++ コンポーネント API 294
rollback メソッド
Ultra Light C++ コンポーネント 38
Rollback メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 391

S

security_parms 同期パラメータ
説明 (Ultra Light C/C++) 527
security 同期パラメータ
説明 (Ultra Light C/C++) 526
SELECT 文
Ultra Light C++ コンポーネント開発 27
シングルロー (Ultra Light Embedded SQL) 96
send_column_names 同期パラメータ
説明 (Ultra Light C/C++) 529
send_download_ack 同期パラメータ
説明 (Embedded SQL) 530
説明 (静的型 C++ API) 530
SetBinary 関数
Ultra Light C++ コンポーネント API 361
SetColumnNull メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 409
SetColumn メソッド (生成された結果セットのクラス)
Ultra Light 静的型 C++ API 434
SetColumn メソッド (生成されたテーブル・クラス)
Ultra Light 静的型 C++ API 442
SET CONNECTION 文
Ultra Light Embedded SQL の複数の接続 80
SetDatabaseID 関数
Ultra Light C++ コンポーネント API 294
SetDatabaseID メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 392
SetDefault 関数
Ultra Light C++ コンポーネント API 345
SetNullColumn メソッド (生成された結果セットのクラス)
Ultra Light 静的型 C++ API 434
SetNullColumn メソッド (生成されたテーブル・クラス)
Ultra Light 静的型 C++ API 442
SetNull 関数
Ultra Light C++ コンポーネント API 345
SetNull メソッド (生成文のクラス)
Ultra Light 静的型 C++ API 436
SetParameterNull 関数
Ultra Light C++ コンポーネント API 321
SetParameter 関数
Ultra Light C++ コンポーネント API 321
SetParameter メソッド (ULResultSet クラス)
Ultra Light 静的型 C++ API 422
SetParameterNext メソッド (ULResultSet クラス)
Ultra Light 静的型 C++ API 436
SetParameter メソッド (生成文のクラス)
Ultra Light 静的型 C++ API 436
SetReadPosition 関数
Ultra Light C++ コンポーネント API 334
SetString 関数
Ultra Light C++ コンポーネント API 361, 362
SetSuspend 関数
Ultra Light C++ コンポーネント API 294, 302
SetSynchInfo 関数
Ultra Light C++ コンポーネント API 295
Set 関数
Ultra Light C++ コンポーネント API 345
Set メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 408
Shutdown 関数
Ultra Light C++ コンポーネント API 295, 307
sqlaid SQLCA フィールド
Ultra Light Embedded SQL 77
SQLAllocHandle 関数
Ultra Light ODBC インタフェース 480
SQL Anywhere Studio
マニュアル xii
SQLBindCol 関数
Ultra Light ODBC インタフェース 481
SQLBindParameter 関数
Ultra Light ODBC インタフェース 482
SQLCA
Ultra Light C/C++ 130
Ultra Light Embedded SQL 77
フィールド (Ultra Light) 77
複数 (Ultra Light Embedded SQL) 79
sqlcabc SQLCA フィールド
Ultra Light Embedded SQL 77

SQLCODE
Ultra Light C++ コンポーネントのエラー処理 41
Ultra Light C/C++ エラー処理 248
sqlcode SQLCA フィールド
Ultra Light Embedded SQL 77
SQL Communications Area
Ultra Light C/C++ 130
Ultra Light Embedded SQL 77
SQLConnect 関数
Ultra Light ODBC インタフェース 484
SQLDescribeCol 関数
Ultra Light ODBC インタフェース 486
SQLDisconnect 関数
Ultra Light ODBC インタフェース 488
SQLEndTran 関数
Ultra Light ODBC インタフェース 489
sqlerrd SQLCA フィールド
Ultra Light Embedded SQL 78
sqlerrmc SQLCA フィールド
Ultra Light Embedded SQL 78
sqlerrml SQLCA フィールド
Ultra Light Embedded SQL 78
sqlerrp SQLCA フィールド
Ultra Light Embedded SQL 78
SQLExecDirect 関数
Ultra Light ODBC インタフェース 490
SQLExecute 関数
Ultra Light ODBC インタフェース 491
SQLFetchScroll 関数
Ultra Light ODBC インタフェース 493
SQLFetch 関数
Ultra Light ODBC インタフェース 492
SQLFreeHandle 関数
Ultra Light ODBC インタフェース 494
SQLGetCursorName 関数
Ultra Light ODBC インタフェース 495
SQLGetData 関数
Ultra Light ODBC インタフェース 496
SQLGetDiagRec 関数
Ultra Light ODBC インタフェース 497
SQLGetInfo 関数
Ultra Light ODBC インタフェース 499
SQLNumResultCols 関数
Ultra Light ODBC インタフェース 500
sqlpp ユーティリティ
Ultra Light Embedded SQL アプリケーション 118
SQLPrepare 関数
Ultra Light ODBC インタフェース 501
SQLRowCount 関数
Ultra Light ODBC インタフェース 502
SQLSetConnectionName 関数
Ultra Light ODBC インタフェース 504
SQLSetCursorName 関数
Ultra Light ODBC インタフェース 503
SQLSetSuspend 関数
Ultra Light ODBC インタフェース 505
sqlstate SQLCA フィールド
Ultra Light Embedded SQL 78
SQLSynchronize 関数
Ultra Light ODBC インタフェース 506
sqlwarn SQLCA フィールド
Ultra Light Embedded SQL 78
SQL プリプロセッサ
Ultra Light Embedded SQL アプリケーション 118
Ultra Light サンプル 121
StartDatabase メソッド (ULData クラス)
Ultra Light 静的型 C++ API 420
StartSynchronizationDelete 関数
Ultra Light C++ コンポーネント API 295
StartSynchronizationDelete メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 392
StopDatabase メソッド (ULData クラス)
Ultra Light 静的型 C++ API 421
StopSynchronizationDelete 関数
Ultra Light C++ コンポーネント API 295
StopSynchronizationDelete メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API 393
stream_error 同期パラメータ
ul_stream_error 構造体 (Ultra Light C/C++) 534
説明 (Ultra Light C/C++) 534
stream_parms 同期パラメータ

説明 (Ultra Light C/C++)　538
stream 同期パラメータ
説明 (Embedded SQL)　531
説明 (静的型 C++ API)　531
StringCompare 関数
Ultra Light C++ コンポーネント API　362
StrToUUID 関数
Ultra Light C++ コンポーネント API　296
Synchronize 関数
Ultra Light C++ コンポーネント API　296
Synchronize メソッド (ULConnection クラス)
Ultra Light 静的型 C++ API　393
sysAppLaunchCmdNormalLaunch
Ultra Light アプリケーション　148, 153
system_error_code 値
同期ストリーム・エラー (Ultra Light クライアント)　536

T

TableSchema オブジェクト
Ultra Light C++ コンポーネント開発　39
Table オブジェクト
Ultra Light C++ コンポーネント開発　27
TCP/IP 同期
Ultra Light の Palm Computing Platform　155
TruncateTable 関数
Ultra Light C++ コンポーネント API　345

U

UL_AS_SYNCHRONIZE マクロ
ActiveSync Ultra Light メッセージ　268
ul_binary Ultra Light データ型
Ultra Light 静的型 C++ API　376
ul_char Ultra Light データ型
Ultra Light 静的型 C++ API　376
ul_column_num Ultra Light データ型
Ultra Light 静的型 C++ API　376
UL_ENABLE_OBFUSCATION マクロ
説明　269
UL_ENABLE_SEGMENTS マクロ
説明　269
UL_ENABLE_USER_AUTH マクロ
説明　269
ul_fetch_offset Ultra Light データ型
Ultra Light 静的型 C++ API　376
ul_length Ultra Light データ型
Ultra Light 静的型 C++ API　376
UL_NULL　377
UL_OMIT_COLUMN_INFO マクロ
説明　270
UL_STORE_PARMS マクロ
説明　270
ul_stream_error 構造体
説明 (Ultra Light C/C++)　534
UL_SYNC_ALL_PUBS マクロ
説明　272
パブリケーション・マスク　524
UL_SYNC_ALL マクロ
説明　272
パブリケーション・マスク　524
ul_synch_info 構造体
Ultra Light C/C++　508
説明　62, 108
ul_synch_status 構造体
Ultra Light Embedded SQL　113
静的型 C++ API　65
UL_TEXT マクロ
説明　273
UL_USE_DLL マクロ
説明　273
ULActiveSyncStream 関数
Ultra Light Embedded SQL 構文　450
Windows CE　171
同期ストリームの設定 (Ultra Light C/C++)　531
ulapi.h
Ultra Light 静的型 C++ API　374
ULChangeEncryptionKey 関数
Ultra Light Embedded SQL 構文　451
Ultra Light Embedded SQL の使用　106
Ultra Light 静的型 C++ API の使用　60

ULClearEncryptionKey 関数
Ultra Light Embedded SQL 構文 452
ULConduitStream 関数
同期ストリームの設定 (Ultra Light C/C++) 531
ULConnection クラス
ResetLastDownloadTime メソッド 390
RevokeConnectFrom メソッド 390
Ultra Light 静的型 C++ API 54, 378
ULCountUploadRows 関数
Ultra Light Embedded SQL 構文 453
ULCursor クラス
Ultra Light 静的型 C++ API 395, 438
ULData クラス
Ultra Light 静的型 C++ API 54, 411
UL_AUTH_STATUS_EXPIRED auth_status 値
説明 511
UL_AUTH_STATUS_IN_USE auth_status 値
説明 511
UL_AUTH_STATUS_INVALID auth_status 値
説明 511
UL_AUTH_STATUS_UNKNOWN auth_status 値
説明 511
UL_AUTH_STATUS_VALID_BUT_EXPIRES_SOON auth_status 値
説明 511
UL_AUTH_STATUS_VALID auth_status 値
説明 511
ULDropDatabase 関数
Ultra Light Embedded SQL 構文 455
ULEnableFileDB 関数
Ultra Light C/C++ 構文 253
Ultra Light 静的型 C++ API 52, 374
ULEnableGenericSchema 関数
Ultra Light C/C++ 構文 254
旧式な機能 254
ULEnablePalmRecordDB 関数
Ultra Light C/C++ 構文 256
Ultra Light 静的型 C++ API 52, 374
ULEnableStrongEncryption 関数
Ultra Light C/C++ 構文 257
Ultra Light 静的型 C++ API 52, 374
ULEnableUserAuthentication 関数
Ultra Light C/C++ 構文 258
Ultra Light 静的型 C++ API 52, 374
説明 57, 102
ulgen ユーティリティ
Ultra Light 静的型 C++ API 70
ULGetDatabaseID 関数
Ultra Light Embedded SQL 構文 457
ULGetLastDownloadTime 関数
Ultra Light Embedded SQL 構文 458
ULGetSynchResult 関数
Ultra Light Embedded SQL 構文 460
ulglobal.h
ul_synch_info 構造体 (Ultra Light C/C++) 508
Ultra Light 静的型 C++ API 374
ULGlobalAutoincUsage 関数
Ultra Light Embedded SQL 構文 462
ULGrantConnectTo 関数
Ultra Light Embedded SQL 構文 463
ULHTTPSStream 関数
Ultra Light Embedded SQL 構文 464
Windows CE 175
同期ストリームの設定 (Ultra Light C/C++) 531
ULHTTPStream 関数
Ultra Light Embedded SQL 構文 465
Windows CE 175
同期ストリームの設定 (Ultra Light C/C++) 531
ULInitDatabaseManager 関数
Ultra Light C++ コンポーネント 20
ULInitSynchInfo 関数
説明 108
ULIsSynchronizeMessage 関数
ActiveSync 171
Ultra Light Embedded SQL 構文 466
ULPalmDBStream 関数
Ultra Light Embedded SQL 構文 467
ULPalmExit 関数
旧式な関数 468
ULPalmLaunch 関数
旧式な関数 469

ULRegisterErrorCallback 関数
Ultra Light C/C++ 構文 248, 259
Ultra Light 静的型 C++ API 374
ULRegisterSchemaUpgradeObserver 関数
Ultra Light C/C++ 構文 251, 263
Ultra Light 静的型 C++ API 374
ULResetLastDownloadTime 関数
Ultra Light Embedded SQL 構文 470
ULResultSet クラス
Ultra Light 静的型 C++ API 422
ULRetrieveEncryptionKey 関数
Ultra Light Embedded SQL 構文 471
ULRevokeConnectFrom 関数
Ultra Light Embedded SQL 構文 472
ULRollbackPartialDownload 関数
Ultra Light Embedded SQL 473
ulrt9.dll
Ultra Light C++ コンポーネント・アプリケーションのリンク 46
ulrtcw9.dll
Ultra Light アプリケーションのリンク 46
ulrtw9.dll
Ultra Light アプリケーションのリンク 46
ULSaveEncryptionKey 関数
Ultra Light Embedded SQL 構文 474
ULSecureCerticomTLSStream
説明 (Ultra Light C/C++) 526
ULSecureCerticomTLSStream 関数
Ultra Light plug-in for CodeWarrior 143
ULSecureRSATLSStream
説明 (Ultra Light C/C++) 526
ULSecureRSATLSStream 関数
Ultra Light plug-in for CodeWarrior 143
ULSetDatabaseID 関数
Ultra Light Embedded SQL 構文 475
ULSetSynchInfo 関数
Ultra Light Embedded SQL 構文 476
使用 153
ULSocketStream 関数
Ultra Light Embedded SQL 構文 477
Windows CE 175
同期ストリームの設定 (Ultra Light C/C++) 531
ULSqlcaBase クラス
Ultra Light C++ コンポーネント API 278
ULSqlcaWrap 関数
Ultra Light C++ コンポーネント API 282
ULSqlcaWrap クラス
Ultra Light C++ コンポーネント API 282
~ULSqlca 関数
Ultra Light C++ コンポーネント API 277
ULSqlca 関数
Ultra Light C++ コンポーネント API 276
ULSqlca クラス
Ultra Light C++ コンポーネント API 276
ULStoreDefragFini 関数
Ultra Light C/C++ 構文 265
ULStoreDefragInit 関数
Ultra Light C/C++ 構文 266
ULStoreDefragStep 関数
Ultra Light C/C++ 構文 267
ULSynchronize 関数
Palm Computing Platform 上のシリアル・ポート 155
Ultra Light Embedded SQL チュートリアル 229
構文 478
ULTable オブジェクト
再開する 148
ULTable クラス
Ultra Light 静的型 C++ API 424
Ultra Light
静的型 C++ API 374
静的型開発 50
Ultra Light C++
サポートされるプラットフォーム 11
Ultra Light C++ API
Ultra Light データベースの難読化 106
Ultra Light C++ コンポーネント
暗号化 43
開発 15, 129
スキーマ・ファイルの作成 17
チュートリアル 179
データベース・スキーマのアップグレード 18

テーブル API によるデータ操作　30
動的 SQL によるデータ操作　24
利点　4
Ultra Light C/C++
Palm OS　147
アーキテクチャ　12
インタフェースの結合　132
説明　3
Ultra Light Embedded SQL
権限　76
同期　107
Ultra Light ODBC インタフェース
SQLAllocHandle 関数　480
SQLBindCol 関数　481
SQLBindParameter 関数　482
SQLConnect 関数　484
SQLDescribeCol 関数　486
SQLDisconnect 関数　488
SQLEndTran 関数　489
SQLExecDirect 関数　490
SQLExecute 関数　491
SQLFetchScroll 関数　493
SQLFetch 関数　492
SQLFreeHandle 関数　494
SQLGetCursorName 関数　495
SQLGetData 関数　496
SQLGetDiagRec 関数　497
SQLGetInfo 関数　499
SQLNumResultCols 関数　500
SQLPrepare 関数　501
SQLRowCount 関数　502
SQLSetConnectionName 関数　504
SQLSetCursorName 関数　503
SQLSetSuspend 関数　505
SQLSynchronize 関数　506
Ultra Light plug-in for CodeWarrior
インストール　139
使用　142
プロジェクトの作成　140
プロジェクトの変換　142
Ultra Light 静的型 C++ API
Palm Computing Platform　389, 407, 419
Reopen メソッド　147
Ultra Light データベースの難読化　60
アプリケーションのコンパイル　71
アプリケーションのリンク　71
開発　7, 50
クエリ・クラス　55
クラス階層　374
クラスの生成　70
チュートリアル　198
テーブル・クラス　55
テーブルの定義　51
同期　61
プロジェクトの定義　51
プロジェクトへの文の追加　51
ヘッダ・ファイル　374
利点　5
Ultra Light データベース
Embedded SQL の暗号化　105
Palm Computing Platform への配備　157
Windows CE　166
静的型 C++ API の暗号化　59
Ultra Light ネームスペース
Ultra Light C++ コンポーネント　16
Ultra Light プロジェクト
CodeWarrior　140
Ultra Light ランタイム・ライブラリ
Ultra Light C++ コンポーネント　45
配備　167
UltraLite_Connection_iface クラス
Ultra Light C++ コンポーネント API　286
UltraLite_Connection クラス
Ultra Light C++ コンポーネント API　284
UltraLite_Cursor_iface クラス
Ultra Light C++ コンポーネント API　299
UltraLite_DatabaseManager_iface クラス
Ultra Light C++ コンポーネント API　305
UltraLite_DatabaseManager クラス
Ultra Light C++ コンポーネント API　304
UltraLite_DatabaseSchema_iface クラス
Ultra Light C++ コンポーネント API　309
UltraLite_DatabaseSchema クラス
Ultra Light C++ コンポーネント API　308

UltraLite_IndexSchema_iface クラス
Ultra Light C++ コンポーネント API 314
UltraLite_IndexSchema クラス
Ultra Light C++ コンポーネント API 313
UltraLite_PreparedStatement_iface クラス
Ultra Light C++ コンポーネント API 319
UltraLite_PreparedStatement クラス
Ultra Light C++ コンポーネント API 318
UltraLite_ResultSet_iface クラス
Ultra Light C++ コンポーネント API 323
UltraLite_ResultSetSchema クラス
Ultra Light C++ コンポーネント API 324
UltraLite_ResultSet クラス
Ultra Light C++ コンポーネント API 322
UltraLite_RowSchema_iface クラス
Ultra Light C++ コンポーネント API 325
UltraLite_SQLObject_iface クラス
Ultra Light C++ コンポーネント API 329
UltraLite_StreamReader_iface クラス
Ultra Light C++ コンポーネント API 332
UltraLite_StreamReader クラス
Ultra Light C++ コンポーネント API 331
UltraLite_StreamWriter クラス
Ultra Light C++ コンポーネント API 335
UltraLite_Table_iface クラス
Ultra Light C++ コンポーネント API 338
UltraLite_TableSchema_iface クラス
Ultra Light C++ コンポーネント API 349
UltraLite_TableSchema クラス
Ultra Light C++ コンポーネント API 347
UltraLite_Table クラス
Ultra Light C++ コンポーネント API 336
UltraLite_ クラス
Ultra Light ネームスペースの使用 16
ULValue 関数
Ultra Light C++ コンポーネント API 362, 363, 364, 365, 366, 367, 368
ULValue クラス
Ultra Light C++ コンポーネント API 356
UNDER_CE コンパイラ指令
説明 273
UNDER_PALM_OS コンパイラ指令
説明 274
UpdateBegin 関数
Ultra Light C++ コンポーネント API 346
Update 関数
Ultra Light C++ コンポーネント API 346
Update メソッド (ULCursor クラス)
Ultra Light 静的型 C++ API 409
UpgradeSchemaFromFile 関数
Ultra Light C++ コンポーネント API 298
UpgradeSchemaFromFile メソッド
Ultra Light C++ コンポーネント開発 18
upload_ok 同期パラメータ
説明 (Ultra Light C/C++) 539
upload_only 同期パラメータ
説明 (Ultra Light C/C++) 540
user_data 同期パラメータ
説明 (Ultra Light C/C++) 541
user_name 同期パラメータ
説明 (Ultra Light C/C++) 542
usm ファイル
Ultra Light C++ コンポーネント 17
UUIDToStr 関数
Ultra Light C++ コンポーネント API 297

V

version 同期パラメータ
説明 (Ultra Light C/C++) 543
Visual C++
Windows CE 開発 160

W

WindowProc 関数
ActiveSync 172, 466
Windows CE
Ultra Light の同期 171
開発 160
プラットフォーム稼働条件 160
winsock.lib
WindowsCE アプリケーション 160

あ

アイコン
マニュアルで使用 xvii
値
Ultra Light C++ コンポーネントのアクセス 32
アップグレード
Ultra Light C++ コンポーネントのデータベース・スキーマ 18
Ultra Light データベース 254
アップロード専用同期
upload_only 同期パラメータ (Ultra Light C/C++) 540
アプリケーション
Palm Computing Platform での Ultra Light の配備 156
Ultra Light Embedded SQL アプリケーションの記述 73
Ultra Light Embedded SQL サンプル・アプリケーションの構築 227
Ultra Light Embedded SQL の記述 219
Ultra Light Embedded SQL の構築 118
Ultra Light Embedded SQL のコンパイル 118
Ultra Light Embedded SQL の前処理 118
暗号化
Embedded SQL を使用する Ultra Light データベース 105
Ultra Light C++ コンポーネント開発 43
Ultra Light C/C++ の ULEnableStrongEncryption 関数 257
Ultra Light Embedded SQL データベース 105
Ultra Light Embedded SQL のキーの変更 106
Ultra Light 暗号化キーの変更 (Embedded SQL) 451
Ultra Light 静的型 C++ API 52
Ultra Light 静的型 C++ API データベース 59
Ultra Light 静的型 C++ API のキーの変更 60
静的型 C++ API を使用する Ultra Light データベース 59

い

依存性
Ultra Light Embedded SQL 125
インジケータ変数
NULL (Ultra Light) 94
Ultra Light Embedded SQL 93
インストール
Palm Computing Platform Ultra Light 156
Ultra Light plug-in for CodeWarrior 139
Windows CE Ultra Light 160
インタフェースの結合
Ultra Light C/C++ 132
インデックス
Ultra Light C++ コンポーネントのスキーマ情報 39
インデックス列挙 (生成されたテーブル・クラス)
Ultra Light 静的型 C++ API 442
インポート・ライブラリ
Ultra Light C++ コンポーネント 45

う

ウィザード
Ultra Light 静的型 C++ のパブリケーション作成 202

え

永続ストレージ
cache_size パラメータ 270
Windows CE 166
エミュレータ
Ultra Light C/C++ アプリケーションの WindowsCE 167
エラー
sqlcode SQLCA フィールド (Ultra Light) 77
SQLCODE (Ultra Light) 77
Ultra Light C++ コンポーネントの処理 41
Ultra Light Embedded SQL の通信エラーコード (Ultra Light) 77

エラー処理
Ultra Light C++ コンポーネント 41
Ultra Light C/C++ 248, 259
エラー・チェック
Ultra Light ODBC インタフェース 497

お

オブジェクト
ULConnection 378
ULCursor 395
ULData 411
ULResultSet 422
ULTable 424
生成された結果セット 432
生成されたテーブル 438
生成文 435

か

カーソル
Ultra Light Embedded SQL 97
開発
Ultra Light C++ コンポーネント 15, 129
Ultra Light 静的型 C++ 7
開発ツール
Ultra Light Embedded SQL 125
Ultra Light Embedded SQL の設定 125
Ultra Light Embedded SQL の前処理 125
Ultra Light Embedded SQL 用の設定 125
開発プラットフォーム
Ultra Light C++ 11
開発プロセス
Ultra Light Embedded SQL 9
拡張モード
Palm OS Ultra Light アプリケーション 145
Ultra Light plug-in for CodeWarrior 143
関数
Ultra Light Embedded SQL 445

き

規則
表記 xv
キャスト
Ultra Light C++ コンポーネントのデータ型 33

く

クエリ
Ultra Light 静的型 C++ API 51
シングルロー (Ultra Light Embedded SQL) 96
クラス名
ActiveSync 同期 168
グローバル・オートインクリメント
ULGlobalAutoincUsage 関数 462
ULSetDatabaseID (Ultra Light Embedded SQL) 475
Ultra Light 静的型 C++ API 384, 392
グローバル・データベース識別子
Ultra Light Embedded SQL 475
Ultra Light 静的型 C++ API 392

け

結果セット
Ultra Light C++ コンポーネント 28
結果セット・スキーマ
Ultra Light C++ コンポーネント 28
検索モード
Ultra Light C++ コンポーネント 32

こ

更新
Ultra Light C++ コンポーネントのロー 35
更新モード
Ultra Light C++ コンポーネント 32
構築
Ultra Light Embedded SQL アプリケーション

118
Ultra Light Embedded SQL サンプル・アプリケーション 227
Ultra Light 静的型 C++ API アプリケーション 70
構築プロセス
Ultra Light Embedded SQL アプリケーション 118
単一ファイルの Embedded SQL アプリケーション 122
高度な暗号化
Ultra Light Embedded SQL 105
Ultra Light 静的型 C++ API 59
Ultra Light データベース 257
コミット
Ultra Light C++ コンポーネント 38
コンパイラ
Windows CE 160
コンパイラ・オプション
Ultra Light C++ コンポーネント開発 46
コンパイラ指令
Ultra Light アプリケーション 268
UNDER_CE 273
UNDER_PALM_OS 274
コンパイル
Palm Computing Platform 138
Ultra Light Embedded SQL アプリケーション 118
Ultra Light 静的型 C++ API アプリケーション 70, 71

さ

再起動可能なダウンロード
keep_partial_download 同期パラメータ 516
partial_download_retained 同期パラメータ 521
resume_partial_download 同期パラメータ 525
Ultra Light Embedded SQL 473
Ultra Light 静的型 C++ API 391
最終ダウンロード・タイムスタンプ
ULGetLastDownloadTime 関数 458
Ultra Light データベースでのリセット 390, 470
削除
Ultra Light C++ コンポーネントのロー 37
サポート
ニュースグループ xx
サポートされるプラットフォーム
Ultra Light C++ 11
サンプル・アプリケーション
Palm Computing Platform 用に構築 143
Windows CE での構築 163

し

準備文
Ultra Light C++ コンポーネント 24
状態の保存
Palm OS 上の Ultra Light 147
指令
Ultra Light アプリケーション 268

す

スキーマ
Ultra Light C++ コンポーネントでのアップグレード 18
Ultra Light C++ コンポーネントのアクセス 39
スキーマのアップグレード
Ultra Light データベース 254
スキーマの更新
Ultra Light C/C++ コールバック関数 251
スキーマの変更
Ultra Light C/C++ コールバック関数 251
スキーマ・ファイル
Ultra Light C++ コンポーネントでのアップグレード 18
Ultra Light C++ コンポーネントでの作成 17
スクリプト・バージョン
version 同期パラメータ (Ultra Light C/C++) 543
スクロール

Ultra Light C++ コンポーネント 30
ストリーム定義関数
GetSynchResult メソッド 382
ULActiveSyncStream (Ultra Light Embedded SQL) 450
ULHTTPSStream (Ultra Light Embedded SQL) 464
ULHTTPStream (Ultra Light) 465
ULPalmDBStream (Embedded SQL) 467
ULSetDatabaseID (Embedded SQL) 475
ULSocketStream (Ultra Light Embedded SQL) 477
スレッド
Ultra Light Embedded SQL 79
マルチスレッド Ultra Light C++ コンポーネント・アプリケーション 23

せ

生成された結果セットのクラス
Ultra Light 静的型 C++ API 435
セキュリティ
Power PC では使用不可 155
security_column_names 同期パラメータ (Ultra Light C/C++) 529
security_parms 同期パラメータ (Ultra Light C/C++) 527
security 同期パラメータ (Ultra Light C/C++) 526
Ultra Light C/C++ アプリケーション 155
Ultra Light Embedded SQL の暗号化キーの変更 106
Ultra Light Embedded SQL の難読化 105
Ultra Light 静的型 C++ API の暗号化キーの変更 60
Ultra Light 静的型 C++ API の難読化 59
Ultra Light データベースの暗号化 59, 105
データベースの難読化 269
セグメント
Palm Computing Platform 149, 151, 152, 269
Ultra Light が必要とするセグメントの減少 270
説明 149, 151
マルチセグメント・コードの生成 150
明示的な割り当て 151
ユーザ定義コード 152
接続
Ultra Light C++ コンポーネント・データベース 19
Ultra Light C/C++ の SQLCA 130
Ultra Light Embedded SQL 79
設定
Ultra Light Embedded SQL 用の開発ツール 125
宣言
Ultra Light ホスト変数 81
宣言セクション
Ultra Light Embedded SQL 81

そ

挿入
Ultra Light C++ コンポーネントのロー 36
挿入モード
Ultra Light C++ コンポーネント 32

た

ターゲット・プラットフォーム
Ultra Light C++ 11
タイムスタンプ構造体 Ultra Light Embedded SQL データ型 84
ダウンロード確認
send_download_ack 同期パラメータ (Embedded SQL) 530
send_download_ack 同期パラメータ (静的型 C++ API) 530
ダウンロード専用同期
download_only 同期パラメータ (Ultra Light C/C++) 515

ち

チュートリアル

Ultra Light C++ コンポーネント 179
Ultra Light Embedded SQL 216
Ultra Light 静的型 C++ 198

つ

通信エラー
Ultra Light Embedded SQL 111

て

データ型
Ultra Light C++ コンポーネントのアクセス 32
Ultra Light C++ コンポーネントのキャスト 33
Ultra Light Embedded SQL 82
Ultra Light SQL 列挙 396
Ultra Light 列挙 395
データ操作
Ultra Light C++ コンポーネントのテーブル API 30
Ultra Light C++ コンポーネントの動的 SQL 24
データベース
Ultra Light C++ コンポーネントでの接続 19
Ultra Light C++ コンポーネントのスキーマ情報 39
データベース・スキーマ
Ultra Light C++ コンポーネントでのアップグレード 18
Ultra Light C++ コンポーネントのアクセス 39
データベースの生成
Ultra Light で名前を付ける 142
データベース・ファイル
Ultra Light C/C++ のデフラグメンテーション 135
Ultra Light Embedded SQL の暗号化 105
Ultra Light Embedded SQL の暗号化キーの変更 106
Ultra Light Embedded SQL の難読化 105
Ultra Light (Windows CE) 166
Ultra Light 静的型 C++ API の暗号化 59
Ultra Light 静的型 C++ API の暗号化キーの変更 60
Ultra Light 静的型 C++ API の難読化 59
難読化 269
テーブル
Ultra Light C++ コンポーネントのスキーマ情報 39
Ultra Light 静的型 C++ API の定義 51
テクニカル・サポート
ニュースグループ xx
デフラグメンテーション
Ultra Light C/C++ データベース 135

と

同期
Ultra Light C++ コンポーネント・チュートリアル 193
Ultra Light C++ の HTTP 155
Ultra Light C++ のトラブルシューティング 382
Ultra Light C/C++ の checkpoint_store パラメータ 513
Ultra Light C/C++ の disable_concurrency パラメータ 514
Ultra Light C/C++ の ignored_ rows パラメータ 517
Ultra Light Embedded SQL 107
Ultra Light Embedded SQL アプリケーションへの追加 107
Ultra Light Embedded SQL チュートリアル 229
Ultra Light Embedded SQL のキャンセル 112
Ultra Light Embedded SQL の初期同期 111
Ultra Light Embedded SQL のトラブルシューティング 460
Ultra Light Embedded SQL の変更のコミット 110
Ultra Light Embedded SQL のモニタ 112
Ultra Light Embedded SQL の呼び出し 109

Ultra Light Embedded SQL の例 108
Ultra Light ODBC インタフェース 506
Ultra Light 静的型 C++ API 61, 393
Ultra Light 静的型 C++ API アプリケーションへの追加 61
Ultra Light 静的型 C++ API のキャンセル 65
Ultra Light 静的型 C++ API の初期同期 64
Ultra Light 静的型 C++ API の変更のコミット 64
Ultra Light 静的型 C++ API のモニタ 65
Ultra Light 静的型 C++ API の呼び出し 63
Ultra Light 静的型 C++ API の例 62
Ultra Light 静的型 C++ チュートリアル 211
Ultra Light の HotSync 153
Ultra Light の Palm Computing Platform 153
Ultra Light の TCP/IP 155
Windows CE の Ultra Light 171

同期エラー
Ultra Light Embedded SQL の通信エラー 111

同期関数
ULSetSynchInfo (Ultra Light Embedded SQL) 476

同期ステータス
GetSynchResult メソッド 382
ULGetSynchResult 関数 460

同期ストリーム
stream_error 同期パラメータ (Ultra Light C/C++) 534
stream_parms 同期パラメータ (Ultra Light C/C++) 538
stream 同期パラメータ (Embedded SQL) 531
stream 同期パラメータ (静的型 C++ API) 531
ULActiveSyncStream (Ultra Light C/C++) 531
ULHTTPStream (Ultra Light C/C++) 531
ULSocketStream (Ultra Light C/C++) 531

同期の監視
observer 同期パラメータ (Ultra Light C/C++) 520

同期パラメータ
auth_parms (Ultra Light C/C++) 510
auth_status (Ultra Light C/C++) 511
auth_value (Ultra Light C/C++) 512
download_only (Ultra Light C/C++) 515
new_password (Ultra Light C/C++) 518
num_auth_parms (Ulrta Light C/C++) 519
observer (Ultra Light C/C++) 520
password (Ultra Light C/C++) 522
ping (Ultra Light C/C++) 523
publication (Ultra Light C/C++) 524
security_parms (Ultra Light C/C++) 527
security (Ultra Light C/C++) 526
send_column_names (Ultra Light C/C++) 529
send_download_ack (Embedded SQL) 530
send_download_ack (静的型 C++ API) 530
stream_error (Ultra Light C/C++) 534
stream_parms (Ultra Light C/C++) 538
stream (Embedded SQL) 531
stream (静的型 C++ API) 531
Ultra Light C/C++ の new_password 518
upload_ok (Ultra Light C/C++) 539
upload_only (Ultra Light C/C++) 540
user_data (Ultra Light C/C++) 541
user_name (Ultra Light C/C++) 542
version (Ultra Light C/C++) 543

動的 SQL
Ultra Light C++ コンポーネント開発 24
Ultra Light Embedded SQL への追加 132
Ultra Light 静的型 C++ API への追加 132

トラブルシューティング
ping 同期パラメータ (Ultra Light C/C++) 523
Ultra Light Embedded SQL の同期前の変更のコミット 110
Ultra Light 開発 64, 111
Ultra Light 静的型 C++ API の同期前の変更のコミット 64
upload_ok 同期パラメータ (Ultra Light C/C++) 539
前回の同期 382, 460

トランケーション
FETCH (Ultra Light) 94

トランザクション
Ultra Light C++ コンポーネント 38

トランザクション処理
Ultra Light C++ コンポーネント 38

トランスポート・レイヤ・セキュリティ
Power PC では使用不可 155

な

難読化
Embedded SQL を使用する Ultra Light データベース 106
Ultra Light C++ コンポーネント開発 43
Ultra Light Embedded SQL データベース 105
Ultra Light 静的型 C++ API データベース 59
Ultra Light データベース 269
コンパイラ指令 269
静的型 C++ API を使用する Ultra Light データベース 60

に

ニュースグループ
テクニカル・サポート xx

ね

ネットワーク・プロトコルのオプション
Ultra Light C/C++ 538

は

パーミッション
Ultra Light Embedded SQL 76
バイナリ Ultra Light Embedded SQL データ型 84
配備
Palm Computing Platform の Ultra Light 157
Palm Computing Platform の Ultra Light アプリケーション 156
Palm Computing Platform への Ultra Light 156
Ultra Light C/C++ アプリケーション (Windows CE) 167
Ultra Light データベース 254
パスワード
Ultra Light C++ コンポーネントの認証 42
Ultra Light C/C++ 同期 518
パック 10 進数 Ultra Light Embedded SQL データ型 83
パフォーマンス
Ultra Light cache_size パラメータ 270
パブリケーション
publication 同期パラメータ (Ultra Light C/C++) 524
Ultra Light C++ コンポーネントのスキーマ情報 40
[パブリケーション作成] ウィザード
Ultra Light 静的型 C++ の使用 202

ひ

表記
規則 xv
ヒント
Ultra Light 開発 64, 111

ふ

フィードバック
提供 xx
マニュアル xx
フェッチ
Ultra Light Embedded SQL 96
プラットフォーム
Ultra Light C++ でのサポート 11
プレフィクス・ファイル
CodeWarrior 150
説明 143
プログラム構造
Ultra Light Embedded SQL 76
プロジェクト
Ultra Light 静的型 C++ API 51
Ultra Light 静的型 C++ API の文の追加 51

へ

ヘッダ・ファイル
Ultra Light 静的型 C++ API 374

ほ

ホスト変数
Ultra Light Embedded SQL 81
Ultra Light Embedded SQL の式 89
Ultra Light の使用法 87
Ultra Light のスコープ 87

ま

前処理
Ultra Light Embedded SQL アプリケーション 118
Ultra Light Embedded SQL の開発ツールの設定 125
マクロ
UL_ENABLE_OBFUSCATION 269
UL_ENABLE_SEGMENTS 269
UL_ENABLE_USER_AUTH 269
UL_OMIT_COLUMN_INFO 270
UL_STORE_PARMS 270
UL_SYNC_ALL 272
UL_SYNC_ALL_PUBS 272
UL_TEXT 273
UL_USE_DLL 273
Ultra Light アプリケーション 268
マニュアル
SQL Anywhere Studio xii
マルチスレッド・アプリケーション
Ultra Light C++ コンポーネント 23
Ultra Light Embedded SQL 79
マルチセグメント・コード
Ultra Light の生成 150
マルチセグメント・コードの生成
Ultra Light 150
マルチロー・クエリ
カーソル (Ultra Light) 97

も

モード
Ultra Light C++ コンポーネント 32
文字列
UL_TEXT マクロ 273
文字列 Ultra Light Embedded SQL データ型
NULL で終了された文字列 83
可変長 84
固定長 84

ゆ

ユーザ
Ultra Light C++ コンポーネントの認証 42
ユーザ認証
auth_parms 同期パラメータ (Ultra Light C/C++) 510
auth_status 同期パラメータ (Ultra Light C/C++) 511
auth_value 同期パラメータ (Ultra Light C/C++) 512
Embedded SQL Ultra Light アプリケーション 102, 385, 463, 472
password 同期パラメータ (Ultra Light C/C++) 522
Ultra Light C++ コンポーネント開発 42
Ultra Light C/C++ アプリケーション 258
Ultra Light C/C++ の new_password 同期パラメータ 518
Ultra Light Embedded SQL 472
Ultra Light 静的型 C++ API 52
Ultra Light 静的型 C++ API アプリケーション 57
Ultra Light データベース 258, 385, 390, 463
Ultra Light のコンパイラ指令 269
user_name 同期パラメータ (Ultra Light C/C++) 542

ら

ライブラリ関数
RollbackPartialDownload (Ultra Light 静的型 C++ API) 391
ULActiveSyncStream (Ultra Light Embedded SQL) 450
ULChangeEncryptionKey (Ultra Light Embedded

SQL) 451
ULClearEncryptionKey (Ultra Light Embedded SQL) 452
ULCountUploadRows (Ultra Light Embedded SQL) 453
ULDropDatabase (Ultra Light Embedded SQL) 455
ULEnableFileDB (Ultra Light C/C++) 253
ULEnableGenericSchema (Ultra Light C/C++) 254
ULEnablePalmRecordDB (Ultra Light C/C++) 256, 257
ULEnableUserAuthentication (Ultra Light C/C++) 258
ULGetDatabaseID (Ultra Light Embedded SQL) 457
ULGetLastDownloadTime (Ultra Light Embedded SQL) 458
ULGetSynchResult (Ultra Light Embedded SQL) 460
ULGlobalAutoincUsage (Ultra Light Embedded SQL) 462
ULGrantConnectTo (Ultra Light Embedded SQL) 463
ULHTTPSStream (Ultra Light Embedded SQL) 464
ULHTTPStream (Ultra Light Embedded SQL) 465
ULIsSynchronizeMessage (Ultra Light Embedded SQL) 466
ULPalmDBStream (Ultra Light Embedded SQL) 467
ULPalmExit (Ultra Light Embedded SQL) 468
ULPalmLaunch (Ultra Light Embedded SQL) 469
ULRegisterErrorCallback (Ultra Light C/C++) 248, 259
ULRegisterSchemaUpgradeObserver (Ultra Light C/C++) 251, 263
ULResetLastDownloadTime (Ultra Light Embedded SQL) 470
ULRetrieveEncryptionKey (Ultra Light Embedded SQL) 471
ULRevokeConnectFrom (Ultra Light Embedded SQL) 472
ULRollbackPartialDownload (Ultra Light Embedded SQL) 473
ULSaveEncryptionKey (Ultra Light Embedded SQL) 474
ULSetDatabaseID (Ultra Light Embedded SQL) 475
ULSetSynchInfo (Ultra Light Embedded SQL) 476
ULSocketStream (Ultra Light Embedded SQL) 477
ULStoreDefragFini (Ultra Light C/C++) 265
ULStoreDefragInit (Ultra Light C/C++) 266
ULStoreDefragStep (Ultra Light C/C++) 267
ULSynchronize (Ultra Light Embedded SQL) 478
Ultra Light Embedded SQL 445
ランタイム・ライブラリ
Ultra Light C++ コンポーネント 45
Windows CE 161, 273

り

利点
Ultra Light C++ コンポーネント 4
Ultra Light Embedded SQL 8
Ultra Light 静的型 C++ API 5
リンク
Ultra Light C++ コンポーネント・アプリケーション 45
Ultra Light 静的型 C++ API アプリケーション 71
Ultra Light による Ultra Light アプリケーションのリンク 161

る

ルックアップ・モード
Ultra Light C++ コンポーネント 32

れ

レジストリ
ClientParms レジストリ・エントリ 154

ろ

ロー
Ultra Light C++ コンポーネント・チュートリアルのアクセス 191
Ultra Light C++ コンポーネントの現在値へのアクセス 32
ロールバック
Ultra Light C++ コンポーネント 38