



Ultra Light C/C++ プログラミング

バージョン 16.0

2013 年 2 月

バージョン 16.0
2013 年 2 月

© 2013 SAP AG or an SAP affiliate company. All rights reserved.

次の条件に従うかぎり、このマニュアルの全部または一部を使用、印刷、再生、配布することができます。1) マニュアルの全部または一部にかかわらず、すべてのコピーにこの情報またはマニュアル内のその他の著作権と商標の表示を含めること。2) マニュアルに変更を加えないこと。3) SAP 以外の人間がマニュアルの著者または情報源であるかのように示す行為をしないこと。ここに記載された情報は事前の通知なしに変更されることがあります。

SAP AG およびディストリビュータが販売しているソフトウェア製品には、他のソフトウェアベンダー独自のソフトウェアコンポーネントが含まれているものがあります。国内製品の仕様は変わることがあります。

これらの資料は SAP AG および関連会社 (SAP グループ) が情報のみを目的として提供するものであり、いかなる種類の表明または保証も行いうるものではなく、SAP グループはこの資料に関する誤りまたは脱落について責任を負わないものとします。SAP グループの製品およびサービスに関する保証は、かかる製品およびサービスに付属している明確な保証文書がある場合、そこで明記されている保証に限定されます。ここに記載されているいかなる内容も、追加保証を構成するものとして解釈されるものではありません。

ここに記載された SAP および他の SAP 製品とサービス、ならびに対応するロゴは、ドイツおよび他の国における SAP AG の商標または登録商標です。<http://www.sap.com/corporate-en/legal/copyright/index.epx#trademark> を参照してください。

目次

はじめに	v
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	1
アプリケーション開発	3
Ultra Light C++ アプリケーション開発	3
Embedded SQL を使用した Ultra Light C++ アプリケーション開発	36
Windows Mobile 向け Ultra Light アプリケーション開発	66
チュートリアル : C++ API を使用した Windows アプリケーションの構築	75
レッスン 1 : データベースの作成とデータベースへの接続	75
レッスン 2 : データベースへのデータの挿入	79
レッスン 3 : テーブルのローの選択とリスト	81
レッスン 4 : アプリケーションへの同期の追加	82
チュートリアルのコードリスト	83
API リファレンス	87
Ultra Light C/C++ 共通 API リファレンス	87
Ultra Light C/C++ API リファレンス	110
Ultra Light Embedded SQL API リファレンス	231
索引	279

はじめに

このマニュアルでは、Ultra Light C および Ultra Light C++ のプログラミングインタフェースについて説明します。Ultra Light を使用すると、ハンドヘルドデバイス、iPhone や iPad などのモバイルデバイス、埋め込みデバイスのデータベースアプリケーションを開発し、これらのデバイスに配備できます。

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、Linux、または Mac デスクトップ (開発プラットフォーム)
- サポートされている Microsoft または GNU の C/C++ コンパイラ

ターゲットプラットフォーム

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

- Windows Mobile 5.0 以降
- Windows XP 以降
- Linux
- Embedded Linux
- iOS 3 以降 (iPhone および iPad)
- Mac

サポートされるターゲットプラットフォームの詳細については、<http://www.sybase.com/detail?id=1002288> を参照してください。

アプリケーション開発

この項では、Ultra Light C/C++ API における開発上の注意について説明します。

C/C++ インタフェースによって、Ultra Light 開発者に以下の利点を提供されます。

- 専有容量が小さく、高パフォーマンスで、ネイティブな同期を実装したデータベースストア
- C または C++ 言語の優れた機能、効率、柔軟性
- アプリケーションを Windows Mobile、Windows デスクトッププラットフォーム、Linux デスクトップ、Embedded Linux、iPhone、および iPad に展開する機能

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

参照

- [「Ultra Light データベースの作成方法」『Ultra Light データベース管理とリファレンス』](#)

Ultra Light C++ アプリケーション開発

Ultra Light C++ アプリケーション開発のクイックスタートガイド

以下の手順は、通常、Ultra Light C++ API を使用したアプリケーションを作成するときに使用されます。

1. ULDatabaseManager オブジェクトを初期化します。
2. (省略可) Ultra Light ランタイムライブラリの機能を有効にします。
3. Ultra Light データベースを使用します。既存のデータベースへの接続の確立、新しいデータベースの作成、既存のデータベースの削除、または既存のデータベースのファイルに破損がないかどうかの検証を実行できます。
4. ULDatabaseManager オブジェクトをファイナライズします。

ULDatabaseManager オブジェクトは、アプリケーション内で 1 回のみ初期化し、アプリケーションが終了するときにファイナライズしてください。ULDatabaseManager クラスのすべてのメソッドは静的です。Ultra Light アプリケーションでのエラー情報を取得するには、ULError クラスを使用します。

参照

- [ULDatabaseManager クラス \[Ultra Light C++\]137 ページ](#)
- [「Ultra Light C++ アプリケーションの構築と配備」23 ページ](#)

iPhone と Mac OS X での考慮事項

開発環境

iPhone と Mac OS X の開発環境は Xcode です。

ビルド設定

Ultra Light のヘッダファイルとライブラリを参照するには、SQL Anywhere インストールフォルダの場所にユーザ定義のビルド設定セットを作成すると便利です。たとえば、`SQLANY_ROOT` を `/Applications/SQLAnywhere16` に設定します。この設定を作成するには、プロジェクトエディタの [ビルド] ウィンドウ枠を開き、[ユーザ定義の設定を追加] をクリックして、名前と値を入力します。

インクルードファイル

Ultra Light のインクルードファイルが検索されるようにするには、`$(SQLANY_ROOT)/sdk/include` を [ユーザヘッダ検索パス] の (`USER_HEADER_SEARCH_PATHS`) ビルド設定に追加します。

サポートされない Mobile Link クライアントネットワークプロトコルオプション

iPhone 用、Mac OS X 用、またはその両方の Ultra Light は、以下の Mobile Link クライアントネットワークプロトコルオプションをサポートしません。

- certificate_company
- certificate_unit
- client_port
- identity
- identity_password
- network_leave_open
- network_name

暗号化

Mobile Link サーバを使用して Mac OS X と iPhone の Ultra Light クライアントを同期させるとき、エンドツーエンド暗号化を使用するには、(PEM パブリックキーファイルではなく) パブリックキーを PEM でコード化された X509 証明書にカプセル化し、E2EE プライベートキーを指定する必要があります。E2EE プライベートキーを使用して PEM でコード化された X509 証明書を作成する場合は、証明書作成ユーティリティ `createcert` を使用することをおすすめします。E2EE プライベートキーを取得できたら、Mobile Link サーバの起動時に `-x` オプションを使用して、キーを `e2ee_private_key` オプションに割り当てます。Mobile Link サーバで Ultra Light クライアントを同期させるには、Ultra Light 同期ユーティリティ `ulsync` を使用して、E2EE パブリックキーを `e2ee_public_key` 接続オプションに割り当てます。iPhone と iPhone 以外のクライアントを同時に使用するときは、証明書からパブリックキーを抽出する必要があります。iPhone Ultra Light クライアントを作成する際に、`trusted_certificates` または `e2ee_public_key` オプションが割り当てられている場合には、Ultra Light 同期ユーティリティは、iPhone 開発パッケージのメインリソースバンドル (`mainBundle`) の証明書を検索します。そのため、Xcode プロジェクトの `Resources` フォルダに証明書をインクルードしておく必要があります。

次の標準暗号化方式はサポートされていません。

- FIPS 認定の暗号化

iPhone アプリケーションのデバッグ

Xcode デバッガ (GDB) では、ステップスルーと `longjmp()` 呼び出しでのブレークがサポートされています。通常、アプリケーションでは `longjmp` を使用しませんが、Ultra Light ランタイムライブラリでは内部的に使用されます (エラーが通知されたときなど)。これにより、アプリケーションコードのトレース中に Ultra Light 呼び出しが出現した場合に、問題が発生することがあります。Ultra Light 呼び出しのステップオーバーで、デバッガからエラーを受け取った場合は、プログラムを再起動して、問題のある行の後にブレークポイントを設定し、問題のある行をステップオーバーする代わりに **Continue** コマンドを使用します。デバッガは次のブレークポイントで停止しますが、`longjmp` 呼び出しに関連する問題を回避ができるため、同じ効果があります。この問題が発生する可能性が最も高いのは、**OpenConnection** を使用して既存のデータベースを開く場合、またはデータベースが存在しないかどうかを判別する場合です (データベースが存在しない場合、エラーが通知されます)。

参照

- 「e2ee_public_key」『Mobile Link クライアント管理』
- 「Mobile Link クライアントネットワークプロトコルオプション」『Mobile Link クライアント管理』
- 「データベースセキュリティ」『Ultra Light データベース管理とリファレンス』
- 「証明書作成ユーティリティ (createcert)」『SQL Anywhere サーバ データベース管理』
- 「-x mlsvr16 オプション」『Mobile Link サーバ管理』
- 「Ultra Light 同期ユーティリティ (ulsync)」『Ultra Light データベース管理とリファレンス』

Ultra Light 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。

ULDatabaseManager クラスは、データベースへの接続を開くために使用されます。ULDatabaseManager クラスは、接続が確立されると Null ではない ULConnection オブジェクトを返します。以下のタスクを実行するには、ULConnection オブジェクトを使用します。

- トランザクションのコミットまたはロールバック。
- データの Mobile Link サーバとの同期。
- データベース内のテーブルへのアクセス。
- SQL 文の処理。
- アプリケーション内のエラーの処理。

データベースファイルに対して書き込み可能なパスが指定されていることを確認します。たとえば、`NSSearchPathForDirectoriesInDomains` メソッドを使用して、`NSDocumentDirectory` を問い合わせます。

注意

サンプルコードは、`%SQLANYXSAMP16%¥UltraLite¥CustDB¥` ディレクトリにあります。

マルチスレッドアプリケーション

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

参照

- [ULConnection クラス \[Ultra Light C++\]111 ページ](#)
- [ULDatabaseManager クラス \[Ultra Light C++\]137 ページ](#)
- [ULError クラス \[Ultra Light C++\]151 ページ](#)

Ultra Light データベースへの接続

ULDatabaseManager オブジェクトを使用して、*sample.udb* という名前の Ultra Light データベースを作成するか、またはこのデータベースに接続します。

前提条件

この作業を実行するための前提条件はありません。

◆ タスク

1. ULDatabaseManager オブジェクトを初期化し、次のコードを使用して Ultra Light の機能を有効にします。

```
if( !ULDatabaseManager::Init() ) {  
    return 0;  
}  
ULDatabaseManager::EnableAesDBEncryption();  
  
// Use ULDatabaseManager.Fini() when terminating the app.
```

2. 次のコードを使用して、既存のデータベースへの接続を開きます。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。

```
ULConnection * conn;  
ULError ulerr;  
  
conn = ULDatabaseManager::OpenConnection( "dbf=sample.udb;dbkey=aBcD1234", &ulerr );  
if( conn == NULL ) {  
    if( ulerr.GetSQLCode() == SQLE_ULTRALITE_DATABASE_NOT_FOUND ) {  
        conn = ULDatabaseManager::CreateDatabase( "dbf=sample.udb;dbkey=aBcD1234", &ulerr );  
        if( conn == NULL ) {  
            // write code that uses ulerr to determine what happened  
            return 0;  
        }  
        // add code to create the schema for your database  
    } else {  
        // write code that uses ulerr to determine what happened  
        return 0;  
    }  
}  
assert( conn != NULL );
```

この手順では、接続が成功しなかった場合にエラー情報が保持される ULError オブジェクトを宣言します。

結果

sample.udb データベースへの接続が確立されます。

SQL 文を使用したデータの作成と修正

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

この項では、SQL を使用して次の操作を行う方法を説明します。

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

この項では、SQL 言語については説明しません。

参照

- 「Ultra Light SQL 文」『Ultra Light データベース管理とリファレンス』
- 「`ULTable` クラスを使用したデータの作成と修正」14 ページ

INSERT、UPDATE、DELETE を使用したデータ修正

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

参照

- `ULPreparedStatement` クラス [Ultra Light C++]159 ページ

テーブルへのローの挿入

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

前提条件

この作業を実行するための前提条件はありません。

◆ タスク

1. 次のコードで `ULPreparedStatement` を宣言します。

```
ULPreparedStatement * prepStmt;
```

2. 実行する SQL 文を準備します。

次のコードは、INSERT 文の実行を準備します。

```
prepStmt = conn->PrepareStatement("INSERT INTO MyTable(MyColumn1) VALUES (?");
```

3. 文を準備するときにエラーをチェックします。

たとえば、SQL 構文のエラーのチェックには次のコードが便利です。

```
if( prepStmt == NULL ) {  
    const ULError * ulerr;  
    ulerr = conn->GetLastError();  
    // write code to handle the error  
    return;  
}
```

4. 準備文で ? 文字を置き換える値を設定します。

次のコードは、エラーチェック時に ? 文字を "some value" に設定します。たとえば、パラメータの順序が準備文内のパラメータ数の範囲外である場合は、エラーがキャッチされます。

```
if( !prepStmt->SetParameterString(1, "some value") ) {  
    const ULError * ulerr;  
    ulerr = conn->GetLastError();  
    // write code to handle the error  
    return;  
}
```

5. 準備文を実行し、データをデータベースに挿入します。

次のコードは、文の実行後に発生する可能性があるエラーをチェックします。たとえば、ユニークインデックス内に重複したインデックスの値が見つかり、エラーが返されます。

```
bool success;  
success = prepStmt->ExecuteStatement();  
if( !success ) {  
    const ULError * ulerr;  
    ulerr = conn->GetLastError();  
    // write code to handle the error  
} else {  
    // Use the following line if you are interested in the number of rows inserted ...  
    ul_u_long rowsInserted = prepStmt->GetRowsAffectedCount();  
}
```

6. 準備文のリソースをクリーンアップします。

次のコードは、準備文のオブジェクトで使用されたリソースを解放します。このオブジェクトには、Close メソッドを呼び出した後はアクセスしないでください。

```
prepStmt->Close();
```

7. データをデータベースにコミットします。

次のコードは、データをデータベースに保存し、データの損失を防ぎます。アプリケーションがコミット呼び出しを完了する前に、デバイスアプリケーションが予期せず終了すると、手順 5 のデータが失われます。

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

結果

MyColumn1 の値が文字列 "some value" に設定されている場合、MyTable に新しいローが追加されます。

テーブル内のローの削除

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

前提条件

この作業を実行するための前提条件はありません。

◆ タスク

1. 次のコードで ULPreparedStatement を宣言します。

```
ULPreparedStatement * prepStmt;
```

2. 実行する SQL 文を準備します。

次のコードは、DELETE 文の実行を準備します。

```
prepStmt = conn->PrepareStatement("DELETE FROM MyTable(MyColumn1) VALUES (?)");
```

3. 文を準備するときにエラーをチェックします。

たとえば、SQL 構文のエラーのチェックには次のコードが便利です。

```
if( prepStmt == NULL ) {
    const ULError * ulerr;
    ulerr = conn->GetLastError();
    // write code to handle the error
    return;
}
```

4. 準備文で ? 文字を置き換える値を設定します。

次のコードは、エラーチェック時に ? 文字を 7 に設定します。たとえば、パラメータの順序が準備文内のパラメータ数の範囲外である場合は、エラーがキャッチされます。

```
ul_s_long value_to_delete = 7;
if( !prepStmt->SetParameterInt(1, value_to_delete) ) {
    const ULError * ulerr;
    ulerr = conn->GetLastError();
    // write code to handle the error.
    return;
}
```

5. 準備文を実行し、データベースからデータを削除します。

次のコードは、文の実行後に発生する可能性があるエラーをチェックします。たとえば、外部キーが参照しているローを削除しようとする、エラーが返されます。

```
bool success;
success = prepStmt->ExecuteStatement();
if( !success ) {
    const ULError * ulerr;
    ulerr = conn->GetLastError();
    // write code to handle the error
} else {
    // Use the following line if you are interested in the number of rows deleted ...
    ul_u_long rowsDeleted = prepStmt->GetRowsAffectedCount();
}
```

6. 準備文のリソースをクリーンアップします。

次のコードは、準備文のオブジェクトで使用されたリソースを解放します。このオブジェクトには、Close メソッドを呼び出した後はアクセスしないでください。

```
prepStmt->Close();
```

7. データをデータベースにコミットします。

次のコードは、データをデータベースに保存し、データの損失を防ぎます。アプリケーションがコミット呼び出しを完了する前に、デバイスアプリケーションが予期せず終了すると、手順 5 のデータが失われます。

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

結果

MyTable の MyColumn 値が 7 である場合、このテーブルのローエントリが削除されます。

テーブル内のローの更新

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

前提条件

この作業を実行するための前提条件はありません。

◆ タスク

1. 次のコードで ULPreparedStatement を宣言します。

```
ULPreparedStatement * prepStmt;
```

2. 実行する SQL 文を準備します。

次のコードは、UPDATE 文の実行を準備します。

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


3. 文を準備するときにエラーをチェックします。

たとえば、SQL 構文のエラーのチェックには次のコードが便利です。

```
if( prepStmt == NULL ) {
    const ULError * ulerr;
    ulerr = conn->GetLastError();
    // write code to handle the error
    return;
}
```

4. 準備文で ? 文字を置き換える値を設定します。

次のコードは、エラーチェック時に ? 文字を整数値に設定します。たとえば、パラメータの順序が準備文内のパラメータ数の範囲外である場合は、エラーがキャッチされます。

```
bool success;
success = prepStmt->SetParameterInt( 1, 25 );
if( success ) {
    success = prepStmt->SetParameterInt( 2, -1 );
}
if( !success ) {
    const ULError * ulerr;
    ulerr = conn->GetLastError();
    // write code to handle the error
    return;
}
```

5. 準備文を実行し、データベースのデータを更新します。

次のコードは、文の実行後に発生する可能性があるエラーをチェックします。たとえば、ユニークインデックス内に重複したインデックスの値が見つかったら、エラーが返されます。

```
success = prepStmt->ExecuteStatement();
if( !success ) {
    const ULError * ulerr;
    ulerr = conn->GetLastError();
    // write code to handle the error
} else {
    // if you are interested in the number of rows updated ...
    ul_u_long rowsUpdated = prepStmt->GetRowsAffectedCount();
}
```

6. 準備文のリソースをクリーンアップします。

次のコードは、準備文のオブジェクトで使用されたリソースを解放します。このオブジェクトには、Close メソッドを呼び出した後はアクセスしないでください。

```
prepStmt->Close();
```

7. データをデータベースにコミットします。

次のコードは、データをデータベースに保存し、データの損失を防ぎます。アプリケーションがコミット呼び出しを完了する前に、デバイスアプリケーションが予期せず終了すると、手順 5 のデータが失われます。

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

結果

このシナリオでは、MyColumn 値が -1 である場合、My Table のローエントリが更新されます。この値は 25 に更新されます。

SELECT を使用したデータの検索

SELECT 文を実行して Ultra Light データベースから情報を取得し、返される結果セットを処理します。

前提条件

この作業を実行するための前提条件はありません。

◆ タスク

1. 次のコードを使用して、必要な変数を宣言します。

```
ULPreparedStatement * prepStmt;  
ULResultSet * resultSet;
```

2. 実行する SQL 文を準備します。

次のコードは、SELECT 文の実行を準備します。

```
prepStmt = conn->PrepareStatement("SELECT MyColumn1 FROM MyTable");
```

3. 文を準備するときにエラーをチェックします。

たとえば、SQL 構文のエラーのチェックには次のコードが便利です。

```
if( prepStmt == NULL ) {  
    const ULError * ulerr;  
    ulerr = conn->GetLastError();  
    // write code to handle the error  
    return;  
}
```

4. SQL を実行し、クエリの結果を移動するために使用できる結果セットオブジェクトを返します。

```
resultSet = prepStmt->ExecuteQuery();  
if( resultSet == NULL ) {  
    const ULError * ulerr;  
    ulerr = conn->GetLastError();  
    // write code to handle the error  
    prepStmt->Close();  
    return;  
}
```

5. Next メソッドを呼び出すことによって、ローをトラバースします。結果を文字列として保存し、バッファに格納します。

Next メソッドによって、結果セットの次のローに移動します。呼び出しが `true` を返す場合、ULResultSet オブジェクトはローに位置付けられます。それ以外の場合 (呼び出しが `false` を返す場合)、すべてのローがトラバースされます。

```
while( resultSet->Next() ) {
    char buffer[ 100 ];
    resultSet->GetString( 1, buffer, 100 );
    printf( "MyColumn = %s¥n", buffer );
}
```

- 準備文オブジェクトと結果セットオブジェクトのリソースをクリーンアップします。

Close メソッドを呼び出した後に、準備文オブジェクトにアクセスしないでください。

```
resultSet->Close();
prepStmt->Close();
```

結果

SELECT 文の結果に含まれる文字列はその後コマンドプロンプトに出力されます。

参照

- [ULPreparedStatement.ExecuteQuery メソッド \[Ultra Light C++\]162 ページ](#)

スキーマの説明の作成と取得

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

例

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

```
const char * name;
int column_count;
const ULResultSetSchema & rss = prepStmt->GetResultSetSchema();
int column_count = rss.GetColumnCount();
for( int i = 1; i < column_count; i++ ) {
    name = rss.GetColumnName( i );
    printf( "id = %d, name = %s¥n", i, name );
}
```

この例では、必要な変数が宣言され、ULResultSetSchema オブジェクトが割り当てられます。結果セットオブジェクト自体から ULResultSetSchema オブジェクトを取得することは可能ですが、この例は、文が準備されてクエリが実行される前に、スキーマを利用する方法を示しています。結果セット内のローの数がカウントされ、各カラムの名前が表示されます。

参照

- [ULPreparedStatement.GetResultSetSchema メソッド \[Ultra Light C++\]164 ページ](#)

SQL 結果セットのナビゲーション

ULResultSet クラスに関連付けられている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。

結果セットクラスは、結果セットをナビゲーションする次のメソッドを提供します。

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

参照

- [ULResultSet クラス \[Ultra Light C++\]169 ページ](#)

ULTable クラスを使用したデータの作成と修正

Ultra Light アプリケーションは、SQL 文を実行するか ULTable クラス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、ULTable クラスを使用したデータアクセスについて説明します。

この項では、ULTable クラス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

警告

ローのプライマリキーは更新しないでください。ローを削除し、代わりに新しいローを追加します。

参照

- [「SQL 文を使用したデータの作成と修正」 7 ページ](#)

ローナビゲーション

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

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

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

参照

- [ULTable クラス \[Ultra Light C++\]210 ページ](#)

例

次の例は、MyTable テーブルを開き、各ローの MyColumn カラムの値を表示します。

```
char buffer[ 100 ];
ul_column_num column_id;
ULTable *tbl = conn->OpenTable( "MyTable" );
if( tbl == NULL ) {
    const ULError * ulerr;
    ulerr = conn->GetLastError();
    // write code to handle the error
    return;
}
column_id = tbl->GetTableSchema().GetColumnID( "MyColumn" );
if( column_id == 0 ) {
    // the column "MyColumn" likely does not exist. Handle the error.
    tbl->Close();
    return;
}
while( tbl->Next() ) {
    tbl->GetString( column_id, buffer, 100 );
    printf( "%s\n", buffer );
}
tbl->Close();
```

ULTable 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

次の例は、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。

```
ULTable * tbl = conn->OpenTable( "MyTable", "ix_col" );
```

Ultra Light のモード

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

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

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

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。

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

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

例

次のコードは、新しいローの挿入を示しています。

```
ULTable * tbl = conn->OpenTable("MyTable");  
bool success;  
tbl->InsertBegin(); // enter "Insert mode"  
tbl->SetInt("id", 3);
```

```
tbl->SetString("lname", "Smith");  
tbl->SetString("fname", "Mary");  
success = tbl->Insert();  
conn->Commit();  
tbl->Close();
```

この例では、tbl 変数は MyTable を開くように設定されています。各カラムの値は、現在のローバッファに設定されます。カラムは名前または ID で参照できます。Insert メソッドによって、テンポラリローバッファ値がデータベースに挿入されます。結果がコミットされて表示されます。リソースは Close メソッドによって解放されます。

ローの更新

Update メソッドを使用してテーブル内のローを更新します。

前提条件

この作業を実行するための前提条件はありません。

◆ タスク

1. 更新するローに移動します。

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

2. 更新モードを開始します。

たとえば、次の指示は、テーブル tbl 上で更新モードを開始します。

```
tbl->UpdateBegin();
```

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

```
tbl->SetInt("id", 3);
```

4. Update を実行します。

```
tbl->Update();
```

警告

Find メソッドと Update メソッドを使用する場合は、検索条件に含まれるカラムを更新した後、ポインタが予期した位置にない場合があります。複数のローを更新する場合は、SQL 文を使用することが適切である場合があります。

結果

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。

次の手順

データベースに追加の SQL 操作を実行してから、`database.executed` への変更をコミットするために `Commit` メソッドを実行します。

参照

- 「トランザクション管理」 20 ページ

ロー検索の検索モードとルックアップモード

Ultra Light には、データを操作するための操作モードがいくつかあります。検索では、これらのモードのうち、検索とルックアップの 2 つを使用できます。ULTable 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必要があります。

- **find メソッド** ULTable 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** ULTable 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。

例

この例は、次の SQL 文を使用して作成された `MyTable` という名前のテーブルを使用します。

```
CREATE TABLE MyTable( id int primary key, lname char(100), fname char(100) )
CREATE INDEX ix_lname ON MyTable ( lname )
```

次のコードは、`lname` カラムが "Smith" であるローの `fname` カラムの内容をすべて表示します。

```
ULTable * tbl = conn->OpenTable( "MyTable", "ix_lname" );
char buffer[ 100 ];
bool found;
tbl->FindBegin(); // enter "Find mode"
tbl->SetString( "lname", "Smith" ); // set pointer row buffer to "Smith"
found = tbl->FindFirst();
while( found ) {
    tbl->GetString( 3, buffer, 100 );
    printf( "%s\n", buffer );
    found = tbl->FindNext();
}
tbl->Close();
```

この例では、`tbl` 変数は `ix_lname` インデックスを使用して `MyTable` を開くように設定されています。これにより、`lname` カラムと同じ順序でローが返されます。ULTable オブジェクトは、検索を実行するとき、ローバッファ内の値を使用します。このバッファは "Smith" として指定されています。このことは、`SetString` メソッドによって定義されています。`FindFirst` メソッドは、`lname` が "Smith" に設定されている最初のローからトラバーサルが開始されることを示します。`lname`

が "Smith" に設定されているローがない場合、ポインタはテーブルの最後のローの後に位置付けられます。GetString メソッドによって fname が取得されます。これは、fname のカラム ID が 3 であるためです。次に、結果が表示され、リソースが解放されます。

参照

- [ULTable クラス \[Ultra Light C++\]210 ページ](#)

現在のローの値へのアクセス

ULTable 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

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

カラム値の取得

ULTable 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、カラム名またはカラム ID を引数として取ります。

次の例は、テーブルの最初のカラムが age であることを前提として、開いているテーブルから age 値を取得する 2 つの方法を示します。

```
ul_s_long age1 = tbl->GetInt( 1 );
ul_s_long age2 = tbl->GetInt( "age" );
assert( age1 == age2 );
```

値をループで取得する場合、値の取得にカラム ID を使用するバージョンの方が、パフォーマンスが優れています。

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、カラム名またはカラム ID と値を引数として取ります。

たとえば、次のコードは、テーブルの最初のカラムが lname であることを前提として、文字列カラム lname と fname を持つローに文字列値を設定する 2 つの方法を示しています。

```
tbl->SetString( 1, last_name );
tbl->SetString( "fname", first_name );
```

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

```
// This code is incorrect
tbl->BeforeFirst();
tbl = tbl.GetInt( 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。

参照

- [「CAST 関数 \[データ型変換\]」『Ultra Light データベース管理とリファレンス』](#)
- [「CONVERT 関数 \[データ型変換\]」『Ultra Light データベース管理とリファレンス』](#)

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。

削除するローにカーソルを移動してから、ULTable.Delete メソッドを実行する方法で、ローを削除します。

参照

- [ULResultSet.Delete メソッド \[Ultra Light C++\]175 ページ](#)

例

次のコードは、テーブル内の最初のローを削除する方法を示しています。

```
tbl->First();  
tbl->Delete();
```

トランザクション管理

トランザクションは、データベースを変更するために最初の文によって暗黙的に開始されますが、明示的にコミットまたはロールバック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、ULConnection.Commit メソッドを使用します。

トランザクションをロールバックするには、ULConnection.Rollback メソッドを使用します。

参照

- [ULConnection.Commit メソッド \[Ultra Light C++\]115 ページ](#)
- [ULConnection.Rollback メソッド \[Ultra Light C++\]130 ページ](#)
- [「Ultra Light でのトランザクション処理」『Ultra Light データベース管理とリファレンス』](#)

スキーマ情報へのアクセス

結果セットまたはデータベース構造の定義は、プログラムを使用して取得できます。これらの記述はスキーマ情報と呼ばれます。この情報は、Ultra Light C API スキーマクラスから使用できます。

注意

Ultra Light C API を使用してスキーマを変更することはできません。スキーマ情報の取得のみが可能です。

次のスキーマオブジェクトと情報にアクセスできます。

- **ULResultSetSchema** クエリ、またはテーブル内のデータの定義です。識別子、名前、各カラムの型情報、およびテーブル内のカラム数を公開します。ULResultSetSchema クラスは、以下のクラスから取得できます。
 - ULPreparedStatement
 - ULResultSet
 - ULTable
- **ULDatabaseSchema** データベース内のテーブルとパブリケーションの数と名前、日付と時刻の形式などのグローバルプロパティを公開します。ULDatabaseSchema クラスは、ULConnection クラスから取得できます。
- **ULTableSchema** カラムとインデックスの設定に関する情報を公開します。ULTableSchema クラスのカラム情報は、ULResultSetSchema クラスから利用できる情報を補完します。たとえば、カラムにデフォルト値があるかどうか、または NULL 値を許可するかどうかを判別できます。ULTableSchema クラスは、ULTable クラスから取得できます。
- **ULIndexSchema** インデックス内のカラムに関する情報を返します。ULIndexSchema クラスは、ULTableSchema クラスから取得できます。

ポインタとして返される ULDatabaseSchema、ULTableSchema、および ULIndexSchema クラスと異なり、ULResultSetSchema クラスは定数参照として返されます。定数参照を返すクラスは閉じられませんが、ポインタとして返されるクラスは閉じる必要があります。

次のコードは、スキーマクラスの正しい閉じ方と不正な閉じ方を示しています。

```
// This code demonstrates proper use of the ULResultSetSchema class:
const ULResultSetSchema & rss = prepStmt->GetResultSetSchema();
c_count = prepStmt->GetSchema().GetColumnCount();

// This code demonstrates proper use of the ULDatabaseSchema class:
ULDatabaseSchema * dbs = conn->GetResultSetSchema();
t_count = dbs->GetTableCount();
dbs->Close(); // This line is required.

// This code demonstrates improper use of the ULDatabaseSchema class
// because the object needs to be closed using the Close method:
t_count = conn->GetResultSetSchema()->GetTableCount();
```

参照

- [ULPreparedStatement クラス \[Ultra Light C++\]159 ページ](#)
- [ULResultSet クラス \[Ultra Light C++\]169 ページ](#)
- [ULTable クラス \[Ultra Light C++\]210 ページ](#)
- [ULConnection クラス \[Ultra Light C++\]111 ページ](#)

エラー処理

Ultra Light C++ API には、エラー情報の取得に使用する `ULError` オブジェクトが含まれています。API のさまざまなメソッドが、メソッド呼び出しが成功したかどうかを示すブール値を返します。場合によっては、エラーが発生したときに `NULL` が返されます。`ULConnection` オブジェクトには、`ULError` オブジェクトを返す `GetLastError` メソッドが含まれています。

エラーを診断するには、`SQLCode` を使用します。`SQLCode` に加えて、`GetParameterCount` メソッドと `GetParameter` メソッドを使用すると、エラーに関する追加情報を提供する追加のパラメータが存在するかどうかを判別できます。

Ultra Light では、明示的なエラー処理に加えて、エラーコールバック関数をサポートしています。コールバック関数を登録すると、Ultra Light エラーが発生するたびに関数が呼び出されます。コールバック関数がアプリケーションフローを制御することはありませんが、すべてのエラーを通知することができます。コールバック関数を使用すると、アプリケーションの開発中やデバッグ中は特に効果的です。

参照

- [「チュートリアル：C++ API を使用した Windows アプリケーションの構築」75 ページ](#)
- [ULSetErrorCallback メソッド \[Ultra Light Embedded SQL\]259 ページ](#)
- [「SQL Anywhere のエラーメッセージ \(Sybase エラーコード順\)」『エラーメッセージ』](#)

Mobile Link データ同期

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

Ultra Light C++ API は、TCP/IP、TLS、HTTP、HTTPS 通信による同期をサポートします。同期は、Ultra Light アプリケーションによって開始されます。接続オブジェクトのメソッドとプロパティは、同期を制御するために使用できます。

参照

- [「Ultra Light クライアント」『Ultra Light データベース管理とリファレンス』](#)
- [ul_sync_info 構造体 \[Ultra Light C および Embedded SQL データタイプ\]98 ページ](#)
- [「Ultra Light の同期パラメータ」『Ultra Light データベース管理とリファレンス』](#)

Ultra Light データベース接続の切断

使用しなくなったソフトウェアリソースを開放し、アプリケーションがデータベースに接続されているあいだ Ultra Light データベースファイルが使用中のままにならないようにします。

前提条件

この作業を実行するための前提条件はありません。

◆ タスク

1. Close メソッドを呼び出して、リソースを解放します。

アプリケーションでデータベースへの接続が必要でなくなったら、次のコードを使用します。

```
if( conn != NULL ) {
    conn->Close( &ulerr );
}
```

2. Fini メソッドを呼び出して、ULDatabaseManager オブジェクトをファイナライズします。

アプリケーションを閉じる場合、次のコードを使用します。

```
ULDatabaseManager.Fini();
```

結果

データベースへの接続が閉じられ、リソースが開放されます。

参照

- [ULConnection.Close メソッド \[Ultra Light C++\]115 ページ](#)
- [ULDatabaseManager.Fini メソッド \[Ultra Light C++\]145 ページ](#)

Ultra Light C++ アプリケーションの構築と配備

Ultra Light エンジンを使用しない C/C++ アプリケーションを作成する場合、静的な Ultra Light ランタイムライブラリにリンクさせます (この方法では、すべての Ultra Light コードが確実にアプリケーションにリンクされます)。あるいは、Windows および Windows Mobile でインポートライブラリにリンクさせて、アプリケーションの起動時に Ultra Light ランタイムコードを動的にロードすることもできます。

リンカオプションおよびコンパイラオプションによる Linux 開発用ランタイムの構築とリンク

libulrt.a のリンカ/コンパイラオプションは、以下のとおりです。

```
-L<${SQLANY16}/ultralite/linux/x86/586/lib -lulrt -julbase
```

エンジンのオプション :

```
-L<${SQLANY16}/ultralite/linux/x86/586/lib -lulrtc -julbase
```

ヘッダコマンドラインのオプション :

```
-I<${SQLANY16}/sdk/include
```

Windows Mobile 用の Ultra Light アプリケーションの配備 (静的リンク)

適切な作成パラメータ、接続パラメータ、同期パラメータ、プロトコルオプション、リンクライブラリ、メソッド呼び出し、配備ファイルを指定して、Ultra Light C++ アプリケーションが Windows デバイス上や Windows Mobile デバイス上で正常に実行されるようにします。

前提条件

この作業を実行するための前提条件はありません。

◆ タスク

- 次のパラメータを指定します。
 - 難読化を使用している場合は、データベース作成時に作成パラメータ **obfuscate=1** を設定します。
 - AES または FIPS 140-2 AES 暗号化を使用している場合は、データベースの作成時または接続時に接続パラメータ **DBKEY=encryption-key** を設定します。
- Ultra Light アプリケーションで同期を使用する場合は、パラメータを適切に設定してください。

同期タイプ	パラメータの設定
TCP/IP	Stream 同期パラメータを tcpip に設定します。
HTTP	Stream 同期パラメータを http に設定します。
RSA TLS	Stream 同期パラメータを tls に設定します。
RSA HTTPS	Stream 同期パラメータを https に設定します。
RSA FIPS 140-2 TLS	Stream 同期パラメータを tls に設定します。 プロトコルオプション fips=yes を設定します。
RSA FIPS 140-2 HTTPS	Stream 同期パラメータを https に設定します。 プロトコルオプション fips=yes を設定します。

- RSA または RSA FIPS 140-2 エンドツーエンド暗号化を使用している場合は、プロトコルオプション **e2ee_public_key=key-file** を設定します。

4. ZLIB 圧縮を使用している場合は、プロトコルオプション **compression=zlib** を設定します。
5. 次のファイルに対してリンクします。

- *ulrt.lib*

- *ulbase.lib*

- RSA TLS、または RSA HTTPS の同期を使用している場合は *ulrsa.lib*

Windows Mobile の場合は、これらのファイルは %SQLANY16%\UltraLite\CE\Arm.50\Lib にあります。Windows の場合は、これらのファイルは %SQLANY16%\UltraLite\Windows\x64\Lib\VS9 または %SQLANY16%\UltraLite\Windows\x86\Lib\VS9 にあります。

6. Ultra Light アプリケーションで次のメソッドを呼び出します。

- AES 暗号化を使用している場合は `ULDatabaseManager.EnableAesDBEncryption` メソッド。

- FIPS 140-2 AES 暗号化を使用している場合は
`ULDatabaseManager.EnableAesFipsDBEncryption` メソッド。

7. 次のメソッドが、Ultra Light アプリケーションで使用される同期タイプに対して呼び出されるようにします。

同期タイプ	パラメータの設定
TCP/IP	<code>EnableTcpipSynchronization</code> メソッドを呼び出します。
HTTP	<code>EnableHttpSynchronization</code> メソッドを呼び出します。
RSA TLS	<code>EnableTlsSynchronization</code> メソッドと <code>EnableRsaSyncEncryption</code> メソッドを呼び出します。
RSA HTTPS	<code>EnableHttpsSynchronization</code> メソッドと <code>EnableRsaSyncEncryption</code> メソッドを呼び出します。
RSA FIPS 140-2 TLS	<code>EnableTlsSynchronization</code> メソッドと <code>EnableRsaFipsEncryption</code> メソッドを呼び出します。
RSA FIPS 140-2 HTTPS	<code>EnableHttpsSynchronization</code> メソッドと <code>EnableRsaFipsSyncEncryption</code> メソッドを呼び出します。

8. 次のファイルを配備します。

- FIPS 140-2 AES 暗号化を使用している場合は、*ulfips16.dll* と *sbgse2.dll*。

- RSA FIPS 140-2 TLS、または RSA FIPS 140-2 HTTPS 同期を使用している場合は *sbgse2.dll* と *mlcrsafips16.dll*。

Windows Mobile の場合は、ファイルは %SQLANY16%\UltraLite\CE\Arm.50 にあります。
Windows の場合は、ファイルは %SQLANY16%\UltraLite\Windows\x64 または %SQLANY16%\UltraLite\Windows\x86 にあります。

結果

Ultra Light C++ アプリケーションは静的リンクを使用し、配備先の Windows デスクトップまたは Windows Mobile デバイスで正常に稼動します。

次の手順

アプリケーションを配備した Windows デスクトップまたは Windows Mobile デバイスに Ultra Lite データベースを配備するか、配備したアプリケーションと一緒に新しいデータベースを作成します。

参照

- 「Ultra Light アプリケーションのビルドと配備の仕様」『Ultra Light データベース管理とリファレンス』
- 「Ultra Light および Ultra Light Java Edition データベースの配備に関する制限事項」『Ultra Light データベース管理とリファレンス』

Windows Mobile 用の Ultra Light アプリケーションの配備 (動的リンク)

適切な作成パラメータ、接続パラメータ、同期パラメータ、プロトコルオプション、リンクライブラリ、メソッド呼び出し、配備ファイルを指定して、Ultra Light C++ アプリケーションが Windows デバイス上や Windows Mobile デバイス上で正常に実行されるようにします。

前提条件

この作業を実行するための前提条件はありません。

◆ タスク

1. 次のパラメータを指定します。
 - 難読化を使用している場合は、データベース作成時に作成パラメータ **obfuscate=1** を設定します。
 - AES または FIPS 140-2 AES 暗号化を使用している場合は、データベースの作成時または接続時に接続パラメータ **DBKEY=encryption-key** を設定します。
2. Ultra Light アプリケーションで同期を使用する場合は、パラメータを適切に設定してください。

同期タイプ	パラメータの設定
TCP/IP	Stream 同期パラメータを tcpip に設定します。
HTTP	Stream 同期パラメータを http に設定します。
RSA TLS	Stream 同期パラメータを tls に設定します。
RSA HTTPS	Stream 同期パラメータを https に設定します。
RSA FIPS 140-2 TLS	Stream 同期パラメータを tls に設定します。 プロトコルオプション fips=yes を設定します。
RSA FIPS 140-2 HTTPS	Stream 同期パラメータを https に設定します。 プロトコルオプション fips=yes を設定します。

3. RSA または RSA FIPS 140-2 エンドツーエンド暗号化を使用している場合は、プロトコルオプション **e2ee_public_key=key-file** を設定します。
4. ZLIB 圧縮を使用している場合は、プロトコルオプション **compression=zlib** を設定します。
5. 次のファイルに対してリンクします。

● *ulbase.lib*

● *ulimp.lib*

Windows Mobile の場合は、これらのファイルは %SQLANY16%\UltraLite\CE\Arm.50\Lib にあります。Windows の場合は、これらのファイルは %SQLANY16%\UltraLite\Windows\x64\Lib\VS9 または %SQLANY16%\UltraLite\Windows\x86\Lib\VS9 にあります。

6. *ulimp.lib* ライブラリにリンクする場合は、コンパイル時に UL_USE_DLL プリプロセッサマクロを定義してください。たとえば、次のように指定します。

-DUL_USE_DLL

7. Ultra Light アプリケーションで次のメソッドを呼び出します。

● AES 暗号化を使用している場合は ULDatabaseManager.EnableAesDBEncryption メソッド。

● FIPS 140-2 AES 暗号化を使用している場合は
ULDatabaseManager.EnableAesFipsDBEncryption メソッド。

8. 次のメソッドが、Ultra Light アプリケーションで使用される同期タイプに対して呼び出されるようにします。

同期タイプ	パラメータの設定
TCP/IP	EnableTcpipSynchronization メソッドを呼び出します。
HTTP	EnableHttpSynchronization メソッドを呼び出します。
RSA TLS	EnableTlsSynchronization メソッドと EnableRsaSyncEncryption メソッドを呼び出します。
RSA HTTPS	EnableHttpsSynchronization メソッドと EnableRsaSyncEncryption メソッドを呼び出します。
RSA FIPS 140-2 TLS	EnableTlsSynchronization メソッドと EnableRsaFipsEncryption メソッドを呼び出します。
RSA FIPS 140-2 HTTPS	EnableHttpsSynchronization メソッドと EnableRsaFipsSyncEncryption メソッドを呼び出します。

9. 次のファイルを配備します。

- *ulrt16.dll*
- ZLIB 圧縮を使用している場合は、*mlczlib16.dll*。
- RSA TLS、RSA HTTPS、または RSA E2EE を使用している場合は、*mlcrsa16.dll*。
- FIPS 140-2 AES 暗号化を使用している場合は、*ulfips16.dll* と *sbgse2.dll*。
- RSA FIPS 140-2 TLS、または RSA FIPS 140-2 HTTPS 同期を使用している場合は *sbgse2.dll* と *mlcrsafips16.dll*。

Windows Mobile の場合は、ファイルは %SQLANY16%\UltraLite\CE\Arm.50 にあります。
 Windows の場合は、ファイルは %SQLANY16%\UltraLite\Windows\x64 または %SQLANY16%\UltraLite\Windows\x86 にあります。

結果

Ultra Light C++ アプリケーションは動的リンクを使用し、配備先の Windows デスクトップまたは Windows Mobile デバイスで正常に稼動します。

次の手順

アプリケーションを配備した Windows デスクトップまたは Windows Mobile デバイスに Ultra Lite データベースを配備するか、配備したアプリケーションと一緒に新しいデータベースを作成します。

参照

- 「Ultra Light アプリケーションのビルドと配備の仕様」『Ultra Light データベース管理とリファレンス』
- 「Ultra Light および Ultra Light Java Edition データベースの配備に関する制限事項」『Ultra Light データベース管理とリファレンス』

Windows Mobile 用の Ultra Light アプリケーションの配備 (Ultra Light エンジン)

適切な作成パラメータ、接続パラメータ、同期パラメータ、プロトコルオプション、リンクライブラリ、メソッド呼び出し、配備ファイルを指定して、Ultra Light C++ アプリケーションが Windows デバイス上や Windows Mobile デバイス上で正常に実行されるようにします。

前提条件

この作業を実行するための前提条件はありません。

◆ タスク

1. 次のパラメータを指定します。
 - 難読化を使用している場合は、データベース作成時に作成パラメータ **obfuscate=1** を設定します。
 - AES または FIPS 140-2 AES 暗号化を使用している場合は、データベースの作成時または接続時に接続パラメータ **DBKEY=encryption-key** を設定します。
2. Ultra Light アプリケーションで同期を使用する場合は、パラメータを適切に設定してください。

同期タイプ	パラメータの設定
TCP/IP	Stream 同期パラメータを tcpip に設定します。
HTTP	Stream 同期パラメータを http に設定します。
RSA TLS	Stream 同期パラメータを tls に設定します。
RSA HTTPS	Stream 同期パラメータを https に設定します。

同期タイプ	パラメータの設定
RSA FIPS 140-2 RSA TLS	Stream 同期パラメータを tls に設定します。 プロトコルオプション fips=yes を設定します。
RSA FIPS 140-2 RSA HTTPS	Stream 同期パラメータを https に設定します。 プロトコルオプション fips=yes を設定します。

3. RSA または RSA FIPS 140-2 エンドツーエンド暗号化を使用している場合は、プロトコルオプション **e2ee_public_key=key-file** を設定します。
4. ZLIB 圧縮を使用している場合は、プロトコルオプション **compression=zlib** を設定します。
5. 次のファイルに対してリンクします。

● **ulrtc.lib**

● **ulbase.lib**

Windows Mobile の場合は、これらのファイルは %SQLANY16%\UltraLite¥CE¥Arm.50¥Lib にあります。Windows の場合は、これらのファイルは %SQLANY16%\UltraLite¥Windows¥x64¥Lib ¥VS9 または %SQLANY16%\UltraLite¥Windows¥x86¥Lib ¥VS9 にあります。

6. 次のメソッドが、Ultra Light アプリケーションで使用される同期タイプに対して呼び出されるようにします。

同期タイプ	パラメータの設定
TCP/IP	EnableTcpipSynchronization メソッドを呼び出します。
HTTP	EnableHttpSynchronization メソッドを呼び出します。
RSA TLS	EnableTlsSynchronization メソッドと EnableRsaSyncEncryption メソッドを呼び出します。
RSA HTTPS	EnableHttpsSynchronization メソッドと EnableRsaSyncEncryption メソッドを呼び出します。

同期タイプ	パラメータの設定
RSA FIPS 140-2 TLS	EnableTlsSynchronization メソッドと EnableRsaFipsEncryption メソッドを呼び出します。
RSA FIPS 140-2 HTTPS	EnableHttpsSynchronization メソッドと EnableRsaFipsSyncEncryption メソッドを呼び出します。

7. 次のファイルを配備します。

- *ulengl6.exe*
- ZLIB 圧縮を使用している場合は、*mlczlib16.dll*。
- RSA TLS、RSA HTTPS、または RSA E2EE を使用している場合は、*mlcrsa16.dll*。
- FIPS 140-2 AES 暗号化を使用している場合は、*ulfips16.dll* と *sbgse2.dll*。
- RSA FIPS 140-2 TLS、RSA FIPS 140-2 HTTPS、または RSA FIPS 140-2 E2EE を使用している場合は、*sbgse2.dll* と *mlcrsafips16.dll*。

Windows Mobile の場合は、ファイルは %SQLANY16%\UltraLite\CE\Arm.50 にあります。
Windows の場合は、ファイルは %SQLANY16%\UltraLite\Windows\x64 または %SQLANY16%\UltraLite\Windows\x86 にあります。

結果

Ultra Light C++ アプリケーションは Ultra Light エンジンを使用し、配備先の Windows デスクトップまたは Windows Mobile デバイスで正常に稼動します。

次の手順

アプリケーションを配備した Windows デスクトップまたは Windows Mobile デバイスに Ultra Lite データベースを配備するか、配備したアプリケーションと一緒に新しいデータベースを作成します。

参照

- 「Ultra Light アプリケーションのビルドと配備の仕様」『Ultra Light データベース管理とリファレンス』
- 「Ultra Light および Ultra Light Java Edition データベースの配備に関する制限事項」『Ultra Light データベース管理とリファレンス』

Mac OS X または iOS 用 Ultra Light アプリケーションの配備

適切な作成パラメータ、接続パラメータ、同期パラメータ、プロトコルオプション、メソッド呼び出し、配備ファイルを指定して、Ultra Light アプリケーションが Mac コンピュータ、iPhone、または iPad で正常に実行されるようにします。

前提条件

この作業を実行するための前提条件はありません。

◆ タスク

1. Mac OS X の場合、次のランタイムライブラリファイルを Xcode プロジェクトに追加します。

- `/Applications/SQLAnywhere16/System/ultralite/macosx/x86_64/libulrt.a`

- `/Applications/SQLAnywhere16/System/ultralite/macosx/x86_64/libulbase.a`

2. iOS の場合、Ultra Light ランタイムライブラリにリンクするには、次のいずれかを行います。

`install-dir/ultralite/iphone/libulrt.a` を Xcode の **Frameworks** グループに追加します。

「または」

[Other Linker Flags] (`OTHER_LDFLAGS`) のビルド設定に次の指定を追加します。

```
-L$(SQLANY_ROOT)/ultralite/iphone  
-lulrt
```

ここで、`SQLANY_ROOT` は、SQL Anywhere インストールディレクトリに設定されたカスタムビルド設定を表します。

Ultra Light ランタイムは、インストール後にビルドしてください。 `install-dir/ultralite/iphone/readme.txt` の指示に従ってください。

3. Xcode プロジェクトに適切なフレームワークを追加します。

- Mac OS X の場合、`CoreFoundation.framework`、`CoreServices.framework`、および `Security.framework`

- iOS の場合、`CFNetwork.framework` および `Security.framework`

4. 次のパラメータを指定します。

- 難読化を使用している場合は、データベース作成時に作成パラメータ `obfuscate=1` を設定します。

- AES 暗号化を使用する場合は、データベース作成時または接続時に接続パラメータ `DBKEY=encryption-key` を設定します。

5. Ultra Light アプリケーションで同期を使用する場合は、パラメータを適切に設定してください。

同期タイプ	パラメータの設定
TCP/IP	Stream 同期パラメータを tcpip に設定します。

同期タイプ	パラメータの設定
HTTP	Stream 同期パラメータを http に設定します。
RSA TLS	Stream 同期パラメータを tls に設定します。
RSA HTTPS	Stream 同期パラメータを https に設定します。

6. RSA E2EE 暗号化を使用している場合は、プロトコルオプション **e2ee_public_key=key-file** を設定します。
7. ZLIB 圧縮を使用している場合は、プロトコルオプション **compression=zlib** を設定します。
8. AES 暗号化を使用している場合は `ULDatabaseManager.EnableAesDBEncryption` メソッドを呼び出します。
9. 次のメソッドが、Ultra Light アプリケーションで使用される同期タイプに対して呼び出されるようにします。

同期タイプ	パラメータの設定
TCP/IP	<code>EnableTcpipSynchronization</code> メソッドを呼び出します。
HTTP	<code>EnableHttpSynchronization</code> メソッドを呼び出します。
RSA TLS	<code>EnableTlsSynchronization</code> メソッドと <code>EnableRsaSyncEncryption</code> メソッドを呼び出します。
RSA HTTPS	<code>EnableHttpsSynchronization</code> メソッドと <code>EnableRsaSyncEncryption</code> メソッドを呼び出します。

結果

Ultra Light アプリケーションは、Mac OS X デスクトップまたは iOS デバイスで正常に稼動します。

次の手順

アプリケーションが配備された Mac デスクトップ、iPhone、または iPad に Ultra Light データベースを配備するか、または配備したアプリケーションで新しいデータベースを作成します。

参照

- 「Ultra Light アプリケーションのビルドと配備の仕様」『Ultra Light データベース管理とリファレンス』
- 「Ultra Light および Ultra Light Java Edition データベースの配備に関する制限事項」『Ultra Light データベース管理とリファレンス』

Linux 用 UltraLite アプリケーションの配備

適切な作成パラメータ、接続パラメータ、同期パラメータ、プロトコルオプション、メソッド呼び出し、配備ファイルを指定して、Ultra Light アプリケーションが Linux で正常に実行されるようにします。

前提条件

この作業を実行するための前提条件はありません。

◆ タスク

1. 次のパラメータを指定します。
 - 難読化を使用している場合は、データベース作成時に作成パラメータ **obfuscate=1** を設定します。
 - AES 暗号化を使用する場合は、データベース作成時または接続時に接続パラメータ **DBKEY=encryption-key** を設定します。
2. Ultra Light アプリケーションで同期を使用する場合は、パラメータを適切に設定してください。

同期タイプ	パラメータの設定
TCP/IP	Stream 同期パラメータを tcpip に設定します。
HTTP	Stream 同期パラメータを http に設定します。
RSA TLS	Stream 同期パラメータを tls に設定します。
RSA HTTPS	Stream 同期パラメータを https に設定します。

3. RSA または RSA FIPS 140-2 エンドツーエンド暗号化を使用している場合は、プロトコルオプション **e2ee_public_key=key-file** を設定します。
4. ZLIB 圧縮を使用している場合は、プロトコルオプション **compression=zlib** を設定します。
5. 次のファイルに対してリンクします。

- *libulrt.a*
- *libulbase.a*
- RSA TLS、RSA HTTPS、または RSA E2EE を使用している場合は、*libulrsa.a*

これらのファイルは、*/opt/sqlanywhere16/ultralite/linux/x86/586/lib* にあります。

6. AES 暗号化を使用している場合は `ULDatabaseManager.EnableAesDBEncryption` メソッドを呼び出します。
7. 次のメソッドが、Ultra Light アプリケーションで使用される同期タイプに対して呼び出されるようにします。

同期タイプ	パラメータの設定
TCP/IP	<code>EnableTcpipSynchronization</code> メソッドを呼び出します。
HTTP	<code>EnableHttpSynchronization</code> メソッドを呼び出します。
RSA TLS	<code>EnableTlsSynchronization</code> メソッドと <code>EnableRsaSyncEncryption</code> メソッドを呼び出します。
RSA HTTPS	<code>EnableHttpsSynchronization</code> メソッドと <code>EnableRsaSyncEncryption</code> メソッドを呼び出します。

結果

Ultra Light アプリケーションは配備先の Linux コンピュータで正常に稼動します。

次の手順

アプリケーションが配備された Linux コンピュータに Ultra Light データベースを配備するか、または配備したアプリケーションで新しいデータベースを作成します。

参照

- 「Ultra Light アプリケーションのビルドと配備の仕様」『Ultra Light データベース管理とリファレンス』
- 「Ultra Light および Ultra Light Java Edition データベースの配備に関する制限事項」『Ultra Light データベース管理とリファレンス』

Embedded SQL を使用した Ultra Light C++ アプリケーション開発

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

参照

- 「Ultra Light Embedded SQL API リファレンス」 231 ページ
- 「Ultra Light SQL プリプロセッサユーティリティ (sqlpp)」 『Ultra Light データベース管理とリファレンス』

Ultra Light Embedded SQL アプリケーション開発のクイックスタートガイド

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

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

1. Ultra Light データベースを設計します。
2. 通常 *.sqlc* という拡張子の付いた Embedded SQL ソースファイルにソースコードを記述します。

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

```
EXEC SQL BEGIN DECLARE SECTION
    int cost
    char pname[31];
EXEC SQL END DECLARE SECTION

EXEC SQL SELECT price, prod_name
    INTO :cost, :pname
    FROM ULProduct
    WHERE prod_id= :pid;
```

3. *.sqlc* ファイルの前処理を実行します。

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

4. *.cpp* ファイルをコンパイルします。
5. *.cpp* ファイルをリンクします。

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

参照

- 「Embedded SQL アプリケーションの構築」 64 ページ
- 「Embedded SQL を使用した Ultra Light C++ アプリケーション開発」 36 ページ

Embedded SQL の例

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

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

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

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

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

初期化

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

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

```
EXEC SQL INCLUDE SQLCA;
```

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

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

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

終了の準備

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

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

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

エラー処理

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

参照

- [「db_init メソッド」 231 ページ](#)

Embedded SQL プログラム構造

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

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

```
EXEC SQL INCLUDE SQLCA;
```

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

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

SQL Communications Area の初期化

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

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

SQLDA 型はヘッダファイル `%SQLANY16%¥SDK¥Include¥sqlca.h` に定義されています。

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

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

SQLCA にはエラーコードが入る

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

SQLCA のフィールド

SQLCA には、次のフィールドがあります。

- **sqlcaid** SQLCA 構造体の ID として文字列 SQLCA が格納される 8 バイトの文字フィールド。このフィールドはデバッグ時にメモリの中身を見るときに役立ちます。
- **sqlcabc** long integer。SQLCA 構造体の長さ (バイト単位) が入ります。
- **sqlcode** long integer。データベースが検出した要求エラーのエラーコードが入ります。エラーコードの定義はヘッダファイル `%SQLANY16%¥SDK¥Include¥sqlerr.h` にあります。エラーコードは、0 (ゼロ) は成功、正の値は警告、負の値はエラーを示します。

SQLCODE マクロを使用してこのフィールドに直接アクセスできます。

- **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 (`%I`) 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。

参照

- [「SQL Anywhere のエラーメッセージ」『エラーメッセージ』](#)

Embedded SQL を使用した Ultra Light データベース接続

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

CONNECT 文の形式は次のとおりです。

EXEC SQL CONNECT USING

```
'uid=user-name;pwd=password;dbf=database-filename';
```

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

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。

単一の SQLCA の使用

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。

参照

- [「Ultra Light 接続パラメータ」『Ultra Light データベース管理とリファレンス』](#)
- [「CONNECT 文 \[ESQL\] \[Interactive SQL\]」『SQL Anywhere サーバ SQL リファレンス』](#)

複数の SQLCA を使用した複数のデータベース接続の管理

Embedded SQL 文の SET SQLCA を使用して、SQL プリプロセッサにデータベース要求で特定の SQLCA を使用することを伝えます。

前提条件

この作業を実行するための前提条件はありません。

◆ タスク

1. **db_init** への呼び出しによって、プログラムで使用される各 SQLCA を初期化します。
2. プログラムの先頭かヘッダファイルに置いて、SQLCA 参照がそのタスク固有のデータを指すようにします。

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

この文はコードをまったく生成しないので、パフォーマンスに影響を与えません。

結果

プリプロセッサ内部の状態が変更され、指定の文字列で SQLCA を参照するようにします。

次の手順

db_fini メソッドへの呼び出しによってアプリケーションを閉じると、リソースがクリーンアップされます。

参照

- 「db_init メソッド」 231 ページ
- 「SET SQLCA 文 [ESQL]」 『SQL Anywhere サーバ SQL リファレンス』
- 「SET CONNECTION 文 [Interactive SQL] [ESQL]」 『SQL Anywhere サーバ SQL リファレンス』

ホスト変数

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

ホスト変数の宣言

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

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

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

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

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

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

データ型

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

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

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

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

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

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

● 16 ビット符号付き整数

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

● 32 ビット符号付き整数

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

● 4 バイト浮動小数点数


```
float f;
```

● 8 バイト浮動小数点数

```
double d;
```

● パック 10 進数

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

● ブランクが埋め込まれ、NULL で終了された文字列

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

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

注意

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

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

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

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

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

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

● ブランクが埋め込まれた固定長文字列

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

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

```
DECL_VARCHAR(n) a; /* n >= 1 */
typedef struct VARCHAR {
```

```

a_sql_ulen 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 {
    a_sql_ulen len;
    unsigned char array[1];
} BINARY;

```

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

```

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

```

SQLDATETIME 構造体は、型が DATE、TIME、TIMESTAMP (または、いずれかの型に変換できるもの) のフィールドを取り出すのに使用されます。アプリケーションは、日付に関して独自のフォーマットで処理をすることがありますが、この構造体を使ってデータをフェッチすると、以後の操作が簡単になります。この構造体の中のデータをフェッチすると、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 メンバーは無視されます。

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

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

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

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

これらの構造体は `%SQLANY16%¥SDK¥Include¥sqlca.h` ファイルに定義されています。VARCHAR 型、BINARY 型、TYPE_DECIMAL 型は、データ格納領域が長さ 1 の文字配列のため、ホスト変数の宣言には向いていません。しかし、動的な変数の割り付けや他の変数の型変換を行うのには有効です。

データベースの DATE 型と TIME 型

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

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

参照

- [「データベースオプション」『SQL Anywhere サーバデータベース管理』](#)

ホスト変数の使用法

ホスト変数は次の場合に使用できます。

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

ホスト変数は、テーブル名、カラム名の代わりには**使用できません**。

ホスト変数のスコープ

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

プリプロセッサはすべてのホスト変数をグローバルとみなす

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

ホスト変数ごとにユニークな名前を付けることが最善の方法であるといえます。

例

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

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

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

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

```

        WHERE emp_number = :emp_id;
    }

```

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

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

ホスト変数としての式

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

```

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

```

上記の構文は使用できませんが、次の方法で式を使用できます。

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

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

```

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

```

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

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

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

同じ方法で、他の lvalue をホスト変数として使用できます。

●ポインタの間接参照

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

●配列参照

```
my_array[ i ]
```

●任意の複雑な lvalue

C++ でのホスト変数

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ーへのアクセスに、前の項で説明した方法を使用できます。

```
EXEC SQL INCLUDE SQLCA;
#include "my_class.hpp"
```

```

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

```

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

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

インジケータ変数

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 `a_sql_len` 型のホスト変数で、SQL 文では通常のホスト変数の直後に書きます。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 );
```

フェッチ時または実行時にデータベースサーバからローを受信しなかった場合 (エラーが発生したか、結果セットの末尾に到達した場合)、インジケータの値は変更されません。

注意

32 ビット長および 64 ビット長とインジケータを今後使用できるように、Embedded SQL での `short int` の使用は推奨されなくなりました。`a_sql_len` を代わりに使用します。

インジケータ変数の値

次の表は、インジケータ変数の使用法をまとめたものです。

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

NULL を扱うためのインジケータ変数

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

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

NULL は、カラムに定義されるどのデータ型の値とも同じではありません。インジケータ変数がデータベースに NULL 値を渡すか、NULL の結果を受け取る必要があります。

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];
a_sql_len ind_phone;
EXEC SQL END DECLARE SECTION;
```



```

/* set values of employee number, name,
   initials, and phone number */
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。

参照

- [「SQL Communications Area の初期化」 39 ページ](#)

データのフェッチ

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

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

シングルローフェッチ

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

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

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

参照

- [「SQL Communications Area の初期化」 39 ページ](#)

例

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

```
EXEC SQL BEGIN DECLARE SECTION;
long int   emp_id;
char       name[41];
char       sex;
char       birthdate[15];
a sql len  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。

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。

次の手順でカーソルを管理します。

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。
4. 警告 SQLE_NOTFOUND が返されるまで、ローをフェッチします。エラーコードと警告のコードは、SQL Communications Area 構造体で定義される変数 SQLCODE で返されます。
5. CLOSE 文を使ってカーソルを閉じます。

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];
a_sql_len 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;
}
```

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

先頭からの 絶対ロー		末尾からの 絶対ロー
0	最初のローの前	$-n - 1$
1		$-n$
2		$-n + 1$
3		$-n + 2$
$n - 2$		-3
$n - 1$		-2
n		-1
$n + 1$	最後のローの後	0

カーソル内のローの順序

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

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

カーソルの再配置

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

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

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

カーソルにローを挿入するには、PUT 文を使用します。

更新後のカーソル位置

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

テンポラリテーブルがカーソルに使用されている場合、基本となるテーブルに挿入されたローは、カーソルが閉じられて再び開かれるまでまったく表示されません。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。

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。

参照

- 「FETCH 文 [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』
- 「カーソルの原則」『SQL Anywhere サーバ プログラミング』
- 「ヒント：クエリ処理におけるワークテーブルの使用 (All-rows 最適化ゴールの使用)」『SQL Anywhere サーバ SQL の使用法』

ユーザ認証

完全なサンプルは %SQLANYSAMPLE%\UltraLiteYesqlauth ディレクトリにあります。次のコードは %SQLANYSAMPLE%\UltraLiteYesqlauth\sample.sqc の一部です。

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

```
// new user added: remove DBA
ULRevokeConnectFrom( &sqlca, UL_TEXT("DBA") );
}
EXEC SQL DISCONNECT;
}
// Prompt for password
printf("Enter user ID and password\n");
scanf( "%s %s", uid, pwd );
EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
```

このコードでは、次のタスクを実行します。

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

参照

- [ULGrantConnectTo メソッド \[Ultra Light Embedded SQL\]253 ページ](#)
- [ULRevokeConnectFrom メソッド \[Ultra Light Embedded SQL\]257 ページ](#)

Ultra Light Embedded SQL でのデータの暗号化

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。

暗号化

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

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

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

暗号化された Ultra Light への一般的な接続手順は次のとおりです。

1. EXEC SQL CONNECT 文に指定されている接続文字列で暗号化キーを指定します。
2. 暗号化キーは、key= 接続文字列パラメータの形で指定します。

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

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ベースを開こうとして、間違ったキーが渡されると、`db_init` が `ul_false` を返し、`SQLCODE -840` が設定されます。

暗号化キーを変更します。

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。

`ULChangeEncryptionKey` メソッドの引数として新しいキーを入力します。

難読化

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

参照

- [ULChangeEncryptionKey メソッド \[Ultra Light Embedded SQL\]237 ページ](#)
- [「Ultra Light データベース作成パラメータ」『Ultra Light データベース管理とリファレンス』](#)

Embedded SQL アプリケーションの同期の設定

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。

Embedded SQL API の構造体のメンバーは、UltraLite C++ API と似ています。

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

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、メソッド呼び出しの引数として渡され、同期されます。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機能を追加するには、通常、次の手順を行います。

1. 同期パラメータが格納された構造体を初期化します。
2. アプリケーションのパラメータ値を割り当てます。
3. 同期メソッドを呼び出し、構造体またはオブジェクトを引数として指定します。

同期するときに、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。

参照

- 「同期パラメータの初期化」 58 ページ
- 「Ultra Light のネットワークプロトコルのオプション」『Ultra Light データベース管理とリファレンス』
- 「同期の呼び出し」 58 ページ
- `ul_sync_info` 構造体 [Ultra Light C および Embedded SQL データタイプ]98 ページ
- 「Ultra Light の同期パラメータ」『Ultra Light データベース管理とリファレンス』

同期パラメータの初期化

同期パラメータは、`ul_sync_info` 構造体に格納されます。

`ul_sync_info` 構造体のメンバーは、初期化時には未定義です。構造体のメンバーの初期値にパラメータを設定し、特別なメソッドを呼び出します。同期パラメータは、Ultra Light ヘッダファイル `%SQLANYI6%¥SDK¥Include¥ulglobal.h` で宣言された構造体で定義されます。

参照

- `ULInitSyncInfo` メソッド [Ultra Light Embedded SQL]254 ページ

例

次の例は、`ULInitSyncInfo` メソッドで同期パラメータを初期化する方法を示しています。

```
ul_sync_info synch_info;  
ULInitSyncInfo( &synch_info );
```

例

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

```
ul_sync_info synch_info;  
ULInitSyncInfo( &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 );
```

同期の呼び出し

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

同期処理が機能するのは、Ultra Light アプリケーションを実行するデバイスが Mobile Link サーバと通信できる場合だけです。プラットフォームによっては、デバイスを、クレードルに置くか

または適切なケーブルを使用してサーバコンピュータに接続して、物理的に接続する必要があります。同期を完了できない場合は、エラー処理コードをアプリケーションに追加します。

同期を呼び出すには、ULInitSyncInfo メソッドを呼び出して、同期パラメータを初期化してから、ULSynchronize メソッドを呼び出して、同期化します。

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。

コミットされた変更と同期

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

参照

- 「Download Only 同期パラメータ」『Ultra Light データベース管理とリファレンス』

アプリケーションの初期データ

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。

ヒント

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

参照

- 「Mobile Link 開発のヒント」『Mobile Link サーバ管理』

同期通信エラー

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_MOBILINK_COMMUNICATIONS_ERROR ) {
    printf( " Stream error information:\n"
           "  stream_error_code = %ld\t(ss_error_code)\n"
           "  error_string    = %s\n"
           "  system_error_code = %ld\n",
           (long)info.stream_error.stream_error_code,
           info.stream_error.error_string,
```

```
        (long)info.stream_error.system_error_code );  
    }
```

SQL_MOBILINK_COMMUNICATIONS_ERROR は、通信エラーを表す一般的なエラーコードです。

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

同期のモニタとキャンセル

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

同期のモニタ

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

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

```
ULInitSyncInfo( &info );  
info.user_name = m_EmplIDStr;  
...  
//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

同期をモニタするコールバック関数は、ul_sync_status 構造体をパラメータとして取ります。

ul_sync_status 構造体には次のようなメンバーが含まれます。

```
struct ul_sync_status {  
    struct {  
        ul_u_long bytes;  
        ul_u_long inserts;  
        ul_u_long updates;  
        ul_u_long deletes;  
    } sent;  
    struct {  
        ul_u_long bytes;  
        ul_u_long inserts;  
        ul_u_long updates;  
        ul_u_long deletes;  
    } received;  
    p_ul_sync_info info;  
    ul_sync_state state;
```

```

    ul_u_short    db_table_count;
    ul_u_short    table_id;
    char          table_name[];
    ul_wchar      table_name_w2[];
    ul_u_short    sync_table_count;
    ul_u_short    sync_table_index;
    ul_sync_state state;
    ul_bool       stop;
    ul_u_short    flags;
    ul_void *     user_data;
    SQLCA *       sqlca;
    ul_u_long     current_download_row_count;
    ul_u_long     total_download_row_count;
}

```

- **sent.inserts** これまでにアップロードされた挿入済みローの数。
- **sent.updates** これまでにアップロードされた更新済みローの数。
- **sent.deletes** これまでにアップロードされた削除済みローの数。
- **sent.bytes** これまでにアップロードされたバイト数。
- **received.inserts** これまでにダウンロードされた挿入済みローの数。
- **received.updates** これまでにダウンロードされた更新済みローの数。
- **received.deletes** これまでにダウンロードされた削除済みローの数。
- **received.bytes** これまでにダウンロードされたバイト数。
- **info** `ul_sync_info` 構造体にポインタを返します。
- **db_table_count** データベース内のテーブルの数を返します。
- **table_id** 現在アップロードまたはダウンロードされているテーブルの番号 (1 から始まります) を返します。同期されないテーブルがある場合には、この番号で値がスキップされることがあります。また、番号が必ず増加するとはかぎりません。
- **table_name[]** 現在のテーブルの名前を返します。
- **table_name_w2[]** 現在のテーブルの名前 (ワイド文字バージョン) を返します。このフィールドには、Windows (デスクトップまたは Mobile) 環境の場合のみ値が入ります。
- **sync_table_count** 同期中のテーブルの数を返します。
- **sync_table_index** アップロードまたはダウンロードされているテーブルの番号を返します。この番号は 1 から始まり **sync_table_count** の値で終わります。同期されていないテーブルがある場合には、この番号で値がスキップされることがあります。
- **state** 以下のステータスのいずれかを表します。
 - **UL_SYNC_STATE_STARTING** 同期処理はまだ行われていません。

- **UL_SYNC_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_SYNC_STATE_SENDING_HEADER** 同期ストリームが開かれ、ヘッダが送信されようとしています。
- **UL_SYNC_STATE_SENDING_TABLE** テーブルが送信されています。
- **UL_SYNC_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_SYNC_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_SYNC_STATE_RECEIVING_UPLOAD_ACK** アップロード完了の確認を受信しています。
- **UL_SYNC_STATE_RECEIVING_TABLE** テーブルを受信しています。
- **UL_SYNC_STATE_RECEIVING_DATA** スキーマ情報またはデータを受信しています。
- **UL_SYNC_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_SYNC_STATE_SENDING_DOWNLOAD_ACK** ダウンロード完了の確認が送信されています。
- **UL_SYNC_STATE_DISCONNECTING** 同期ストリームが閉じられようとしています。
- **UL_SYNC_STATE_DONE** 同期は正常に完了しました。
- **UL_SYNC_STATE_ERROR** 同期は完了しましたが、エラーが発生しました。
- **UL_SYNC_STATE_ROLLING_BACK_DOWNLOAD** ダウンロード中にエラーが発生し、ダウンロードがロールバックされています。
- **stop** 同期を中断するには、このメンバーを **true** に設定します。SQL 例外の **SQL_INTERRUPTED** が設定され、通信エラーが発生したかのように同期が停止します。**observer** は、適切なクリーンアップを実行するように、**常に DONE** または **ERROR** のステータスで呼び出されます。
- **flags** 現在の状態に関連する追加情報を示す、現在の同期フラグを返します。
- **user_data** 引数として **ULSetSynchronizationCallback** 関数に渡されるユーザデータオブジェクトを返します。
- **sqlca** 接続のアクティブな **SQLCA** へのポインタを返します。
- **current_download_row_count** これまでにダウンロードされたローの数を返します。この数には、**received.inserts**、**received.updates**、または **received.deletes** に含まれていない、重複するローが含まれます。

- **total_download_row_count** ダウンロードで受信するローの合計数を返します。この数には、received.inserts、received.updates、または received.deletes に含まれていない、重複するローが含まれます。

例

次のコードは、ごく簡単な observer 関数を示しています。

```
extern void __stdcall ObserverFunc(
    p_ul_sync_status status )
{
    switch( status->state ) {
        case UL_SYNC_STATE_STARTING:
            printf( "Starting\n" );
            break;
        case UL_SYNC_STATE_CONNECTING:
            printf( "Connecting\n" );
            break;
        case UL_SYNC_STATE_SENDING_HEADER:
            printf( "Sending Header\n" );
            break;
        case UL_SYNC_STATE_SENDING_TABLE:
            printf( "Sending Table %d of %d\n",
                status->tableIndex + 1,
                status->tableCount );
            break;
        case UL_SYNC_STATE_RECEIVING_UPLOAD_ACK:
            printf( "Receiving Upload Ack\n" );
            break;
        case UL_SYNC_STATE_RECEIVING_TABLE:
            printf( "Receiving Table %d of %d\n",
                status->tableIndex + 1,
                status->tableCount );
            break;
        case UL_SYNC_STATE_SENDING_DOWNLOAD_ACK:
            printf( "Sending Download Ack\n" );
            break;
        case UL_SYNC_STATE_DISCONNECTING:
            printf( "Disconnecting\n" );
            break;
        case UL_SYNC_STATE_DONE:
            printf( "Done\n" );
            break;
    }
    ...
}
```

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

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

CustDB の例

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

CustDB サンプルコードは `%SQLANYSAMPI6%\UltraLite\CustDB` フォルダにあります。observer 関数は `CustDB` フォルダのプラットフォーム固有のサブフォルダに含まれています。

参照

- [ul_sync_info 構造体 \[Ultra Light C および Embedded SQL データタイプ\]98 ページ](#)
- [ul_sync_status 構造体 \[Ultra Light C および Embedded SQL データタイプ\]103 ページ](#)
- 「同期処理」『Mobile Link クイックスタート』

Embedded SQL アプリケーションの構築

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

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

一般的な構築手順

Ultra Light Embedded SQL アプリケーションの構築時に使用される一般的な手順は次のとおりです。

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

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

警告

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

2. 各 C++ ソースファイルを、選択したターゲットプラットフォームに合わせてコンパイルします。次のファイルを含めます。

- SQL プリプロセッサが生成した各 C++ ファイル
- アプリケーションに必要な追加の C または C++ ソースファイル

3. これら**すべて**のオブジェクトファイルを Ultra Light ランタイムライブラリとともにリンクします。

このプロセスを使用する makefile は、`%SQLANYSAMPI6%\UltraLite\ESQLSecurity` ディレクトリに保存されています。

注意

別途ライセンスが必要な必須コンポーネント

FIPS 認定の暗号化には別のライセンスが必要です。強力な暗号化テクノロジーはすべて、輸出規制対象品目です。

「別途ライセンスが必要なコンポーネント」『SQL Anywhere 16 紹介』を参照してください。

参照

- 「Ultra Light SQL プリプロセッサユーティリティ (sqlpp)」『Ultra Light データベース管理とリファレンス』

Embedded SQL 開発用の開発ツールの設定

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 (通常は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。

参照

- 「Ultra Light SQL プリプロセッサユーティリティ (sqlpp)」『Ultra Light データベース管理とリファレンス』

SQL プリプロセッサの実行

Visual C++ 用のプリプロセッサを実行する命令を追加することにより、SQL プリプロセッサを依存ベースの構築環境に組み込みます。

前提条件

この作業を実行するための前提条件はありません。

◆ タスク

1. `.sqlc` ファイルを開発プロジェクトに追加します。

開発プロジェクトは、開発ツールで定義されています。

2. 各 .sqc ファイルのカスタム構築規則を追加します。

- カスタム構築規則により SQL プリプロセッサを実行してください。Visual C++ の場合、構築規則には次のコマンドを含めてください (すべて 1 行に入力)。

```
"%SQLANY16%\Bin32\sqlpp.exe" -q -u $(InputPath) $(InputName).cpp
```

SQLANY16 は SQL Anywhere インストールフォルダを指す環境変数です。

- コマンドの出力を \$(InputName).cpp に設定します。

3. .sqc ファイルをコンパイルし、生成された .cpp ファイルを開発プロジェクトに追加します。

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

4. 生成された .cpp ファイルごとに、プリプロセッサの定義を設定します。

- [全般] または [プリプロセッサ] で、[プリプロセッサ] の定義に UL_USE_DLL を追加します。
- [プリプロセッサ] に、\$(SQLANY16)\SDK\Include と、インクルードパスに必要なインクルードフォルダを、カンマ区切りのリストとして追加します。

結果

Visual C++ 開発用に SQL プリプロセッサが設定されます。

Windows Mobile 向け Ultra Light アプリケーション開発

Microsoft Visual Studio 2005 以降は、Windows Mobile 環境用のアプリケーション開発に使用できます。

Windows Mobile を対象にしたアプリケーションでは、wchar_t のデフォルト設定を使用し、`%Program Files%\SQLAny16\ultralite\ce\arm.50\lib\` 内にある Ultra Light ランタイムライブラリでリンクする必要があります。

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

参照

- 「サポートされるプラットフォーム」『SQL Anywhere 16 紹介』

CustDB のサンプルアプリケーション

CustDB は、Visual Studio ソリューションとして提供される簡単な販売管理アプリケーションです。CustDB は SQL Anywhere インストールディレクトリの `%SQLANY16%\UltraLite\` ディレクトリにあります。

注意

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

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

参照

- [「CustDB サンプルデータベースアプリケーション」『SQL Anywhere 16 紹介』](#)
- [「Embedded SQL アプリケーションの構築」 64 ページ](#)

CustDB サンプルアプリケーションの構築

CustDB のサンプルアプリケーションを構築すると、Ultra Light データベースとのインタフェースとしてのアプリケーションの機能がわかります。

前提条件

この作業を実行するための前提条件はありません。

◆ タスク

1. Visual Studio を起動します。
2. プロジェクトファイルを開きます。このファイルは、`%SQLANYXSAMP16%\UltraLite\CustDB` ディレクトリにあります。
3. **[ビルド] » [構成マネージャの設定]** をクリックしてターゲットプラットフォームを設定します。

使用するアクティブソリューションプラットフォームを設定します。

4. アプリケーションを構築します。

● **[ビルド] » [ソリューションの配置]** をクリックして CustDB をビルドおよび展開します。

アプリケーションがビルドされると、リモートデバイスに自動的にアップロードされます。

5. Mobile Link サーバを起動します。

● Mobile Link サーバを起動するには、**[スタート] » [プログラム] » [SQL Anywhere 16] » [Mobile Link] » [同期サーバのサンプル]** をクリックします。

6. CustDB アプリケーションを実行します。

CustDB アプリケーションを実行する前に、custdb データベースをデバイスのルートフォルダにコピーする必要があります。%SQLANYSAMPI6%\UltraLite\CustDB\custdb.udb というデータベースファイルをデバイスのルートにコピーします。

デバイスまたはシミュレータ上で、¥Program Files の下のフォルダにある CustDB.exe を実行します。

結果

CustDB アプリケーションがロードされます。

永続的なデータ

Ultra Light データベースは Windows Mobile ファイルシステムに格納されます。デフォルトファイルは、¥UltraLiteDB¥ul_store.udb です。この設定は、ファイルベースの永続ストアのフルパス名を指定する **file_name** 接続パラメータを使用して上書きできます。

Ultra Light ランタイムは **file_name** パラメータへの代入を行いません。ファイル名を有効化するためにフォルダの作成が必要な場合は、**db_init** を呼び出す前にフォルダ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フォルダ名を追加します。次に例を示します。

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

アプリケーションに対するクラス名の割り当て

MFC を使用している場合は、アプリケーションに固有のクラス名を割り当てます。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。

前提条件

この作業を実行するための前提条件はありません。

◆ タスク

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。ダイアログを作成する前に、このコードを実行する必要があります。

```
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 `MyAppDlg` 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` メソッドに、次のコードを追加します。

```
m_pMainWnd = &dlg;
```

このコードにより、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。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。
6. プロジェクトのリソースファイルを変更します。
7. メインダイアログのリソース ID を見つけます。

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

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

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

8. `.rc` ファイルを保存します。

9. 同期メッセージを取得するコードを追加します。

結果

アプリケーション固有のクラス名が作成されます。

参照

- 「メインダイアログクラスでの **ActiveSync** 同期の追加」 71 ページ

Windows Mobile の同期

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

- ActiveSync
- TCP/IP
- HTTP

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

参照

- 「ActiveSync の同期設定」 70 ページ
- 「Windows Mobile からの TCP/IP、HTTP、HTTPS 同期」 73 ページ
- 「Windows Mobile からの TCP/IP、HTTP、HTTPS 同期」 73 ページ
- 「Ultra Light の同期パラメータ」『Ultra Light データベース管理とリファレンス』

ActiveSync の同期設定

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

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

ActiveSync を使用する場合、同期を開始できるのは、ActiveSync 自体だけです。デバイスがクレドルにある場合や、[ActiveSync] ウィンドウで **[同期]** 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 プロバイダは **wParam** パラメータを使用します。**wParam** の値が 1 の場合、アプリケーションが ActiveSync 用 Mobile Link プロバイダによって起動されたことを示します。同期の完了後、アプリケーションは自動的に停止しなければなりません。アプリケーションが

ActiveSync 用 Mobile Link プロバイダに呼び出されたときに、アプリケーションがすでに動作していた場合、**wParam** の値は 0 になります。アプリケーションが動作を継続する必要がある場合は、**wParam** パラメータを無視できます。

プロバイダがサポートされているプラットフォームを確認するには、[SQL Anywhere Components by Platform](#) を参照してください。

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

参照

- [「Ultra Light の ActiveSync プロバイダの配備」『Ultra Light データベース管理とリファレンス』](#)

ActiveSync 同期の設定 (Windows API の場合)

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

次の例は、メッセージの処理方法を示しています。

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

DoSync は実際に **ULSynchronize** を呼び出すメソッドです。

参照

- [ULIsSynchronizeMessage メソッド \[Ultra Light Embedded SQL\]254 ページ](#)

メインダイアログクラスでの ActiveSync 同期の追加

メインダイアログクラスで同期メッセージを取得します。

前提条件

アプリケーションを開発するには、Microsoft Foundation Classes を使用する必要があります。

◆ タスク

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。

1. 登録したメッセージを追加し、メッセージハンドラを宣言します。

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

```
static UINT WM_ULTRALITE_SYNC_MESSAGE =  
::RegisterWindowMessage( UL_AS_SYNCHRONIZE );  
BEGIN_MESSAGE_MAP(CMyAppDlg, CDialog)  
//{{AFX_MSG_MAP(CMyAppDlg)  
//}}AFX_MSG_MAP  
ON_REGISTERED_MESSAGE( WM_ULTRALITE_SYNC_MESSAGE,  
    OnDoUltraLiteSync )  
END_MESSAGE_MAP()
```

2. メッセージハンドラを実装します。

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

結果

メインダイアログクラスによって同期が実行されます。

参照

- 「アプリケーションに対するクラス名の割り当て」 68 ページ
- `ULIsSynchronizeMessage` メソッド [Ultra Light Embedded SQL]254 ページ

アプリケーションクラスでの ActiveSync 同期の追加

アプリケーションクラスで同期メッセージを取得します。

前提条件

アプリケーションを開発するには、Microsoft Foundation Classes を使用する必要があります。

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。

◆ タスク

1. アプリケーションクラスのクラスウィザードを開きます。
2. [メッセージ] リストで、**PreTranslateMessage** を強調表示して [関数の追加] をクリックします。
3. [コードの編集] をクリックします。

PreTranslateMessage メソッドが表示されます。

4. 次のように PreTranslateMessage メソッドを変更します。

```
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 メソッドが ULSynchronize メソッドを呼び出す。

結果

PreTranslateMessage メソッドが同期を実行します。

参照

- 「アプリケーションに対するクラス名の割り当て」 68 ページ
- ULSynchronize メソッド [Ultra Light Embedded SQL]262 ページ
- ULIsSynchronizeMessage メソッド [Ultra Light Embedded SQL]254 ページ

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

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

チュートリアル：C++ API を使用した Windows アプリケーションの構築

このチュートリアルでは、Ultra Light C++ アプリケーションを開発するプロセスを説明します。アプリケーションは Windows デスクトップオペレーティングシステム用に開発され、コマンドプロンプトで実行されます。

このチュートリアルは、Microsoft Visual C++ を使用した開発を基にしていますが、任意の C++ 開発環境を使用できます。

このチュートリアルは、コードをコピーして貼り付ける場合、約 30 分で終了します。この章の最後には、このチュートリアルで説明しているプログラムの完全なソースコードを掲載しています。

前提知識と経験

このチュートリアルは、次のことを前提にしています。

- C++ プログラミング言語に精通している。
- C++ コンパイラがマシンにインストールされている。
- データベース作成ウィザードを使用して Ultra Light データベースを作成する方法を知っている。

このチュートリアルの目的は、Ultra Light C++ アプリケーションの開発プロセスについて、知識を得ることです。

参照

- 「データベース作成ウィザードでの Ultra Light データベースの作成」『Ultra Light データベース管理とリファレンス』

レッスン 1：データベースの作成とデータベースへの接続

このレッスンでは、Ultra Light データベースを作成します。次に、作成したデータベースにアクセスする C++ アプリケーションを記述、コンパイル、実行します。

前提条件

このレッスンでは、必要なソフトウェアをすでにインストールしていることを前提としています。「チュートリアル：C++ API を使用した Windows アプリケーションの構築」75 ページを参照してください。

◆ タスク

1. **VCINSTALLDIR** 環境変数に Visual C++ インストール環境のルートフォルダを設定します (変数がまだ存在しない場合)。
2. **%VCINSTALLDIR%¥VC¥atlmfc¥src¥atl** を **INCLUDE** 環境変数に追加します。
3. このチュートリアルで作成されるファイルを保存するフォルダを作成します。

以降、このチュートリアルではこのフォルダが **C:¥tutorial¥cpp¥**であることを前提に説明します。別の名前のフォルダを作成した場合は、**C:¥tutorial¥cpp¥**の代わりにそのフォルダを使用してください。

4. Sybase Central で Ultra Light を使用して、デフォルトの特性を持つデータベース **ULCustomer.udb** を新しいフォルダに作成します。
5. **ULCustomer** という名前のテーブルをデータベースに追加します。次の ULCustomer テーブル仕様を使用します。

カラム名	データ型 (サイズ)	カラムの NULL 値の許可	デフォルト値	プライマリキー
cust_id	integer	いいえ	オートインクリメント	昇順
cust_name	varchar(30)	いいえ	なし	

6. Sybase Central でデータベースから切断します。そうしないと、実行ファイルが接続できません。
7. Microsoft Visual C++ で、**[ファイル]** » **[新規作成]** をクリックします。
8. **[ファイル]** タブで、**[C++ ソース ファイル]** をクリックします。
9. チュートリアルフォルダに、そのファイルを **customer.cpp** として保存します。
10. Ultra Light ライブラリをインクルードします。

以下のコードを **customer.cpp** にコピーします。

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

#include "ulcpp.h"

#define MAX_NAME_LEN 100
#define MAX_ERROR_LEN 256
```

11. データベースに接続するための接続パラメータを定義します。

次のコードでは、接続パラメータはハードコードされています。実際のアプリケーションでは、ロケーションは実行時に指定されることもあります。


```
static ULConnection * open_conn( void ) {
    ULConnection * conn = ULDatabaseManager::OpenConnection( ConnectionParms );
    if( conn == UL_NULL ) {
        _tprintf("Unable to open existing database.¥n");
    }
    return conn;
}
```

14. 次のタスクを実行する main メソッドを実装します。

- エラー処理メソッドを登録します。
- 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() {
    ULConnection * conn;

    ULDatabaseManager::Init();
    ULDatabaseManager::SetErrorCallback( MyErrorCallBack, NULL );

    conn = open_conn();
    if ( conn == UL_NULL ) {
        ULDatabaseManager::Fini();
        return 1;
    }

    // Main processing code goes here ...
    do_insert( conn );
    do_select( conn );
    do_sync( conn );

    conn->Close();
    ULDatabaseManager::Fini();
    return 0;
}
```

15. ソースファイルのコンパイルとリンクを行います。

ソースファイルのコンパイル方法は、コンパイラによって異なります。以下の手順は、`makefile` で Microsoft Visual C++ コマンドラインコンパイラを使用する場合です。

- a. コマンドプロンプトを開き、チュートリアルフォルダに変更します。
- b. `makefile` という名前の `makefile` を作成します。
- c. `makefile` で、フォルダをインクルードパスに追加します。

```
IncludeFolders=/I"${SQLANY16}¥SDK¥Include"
```

- d. `makefile` で、フォルダをライブラリパスに追加します。

```
LibraryFolders=/LIBPATH:"${SQLANY16}¥UltraLite¥Windows¥x86¥Lib¥vs8"
```

- e. `makefile` で、ライブラリをリンカオプションに追加します。

```
Libraries=ulimp.lib
```

Ultra Light ランタイムライブラリの名前は `ulimp.lib` です。

- f. `makefile` で、コンパイラオプションを設定します。次のように、1 行でこれらのオプションを入力してください。

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

- g. 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)
```

- h. 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
```

- i. vsvars32.bat を実行します。

```
%VCINSTALLDIR%¥Tools¥vsvars32.bat
```

- j. makefile を実行します。

```
nmake
```

customer.exe という実行ファイルが作成されます。

16. 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。

結果

アプリケーションがデータベースに接続したら、切断します。エラーメッセージが表示されなければ、アプリケーションは正常に実行されています。

次の手順

[「レッスン 2 : データベースへのデータの挿入」 79 ページ](#)に進みます。

参照

- 「データベース作成ウィザードでの Ultra Light データベースの作成」『Ultra Light データベース管理とリファレンス』
- 「Ultra Light 接続パラメータ」『Ultra Light データベース管理とリファレンス』
- 「エラー処理」 22 ページ

レッスン 2 : データベースへのデータの挿入

このレッスンでは、データベースにデータを追加します。

前提条件

このレッスンは、受講者がこれまでのすべてのレッスンを終了していることを前提としています。「[レッスン 1 : データベースの作成とデータベースへの接続](#)」 75 ページを参照してください。

◆ タスク

1. 以下のメソッドを、*customer.cpp* の main メソッドの直前に追加します。

```
static bool do_insert( ULConnection * conn )
{
    ULTable * table = conn->OpenTable( "ULCustomer" );
    if( table == UL_NULL ) {
        _tprintf( "Table not found: ULCustomer¥n" );
        return false;
    }
    if( table->GetRowCount() == 0 ) {
        _tprintf( "Inserting one row.¥n" );
        table->InsertBegin();
        table->SetString( "cust_name", "New Customer" );
        table->Insert();
        conn->Commit();
    } else {
        _tprintf( "The table has %lu rows¥n", table->GetRowCount() );
    }
    table->Close();
    return true;
}
```

このメソッドでは次のタスクを実行します。

- `connection->OpenTable()` メソッドを使用してテーブルを開きます。テーブルの操作を実行するには、Table オブジェクトを開いてください。
- テーブルが空の場合は、ローを 1 つ追加します。このコードでは、ローを挿入するために、`InsertBegin` メソッドを使用して挿入モードに変更し、必要な各カラムに値を設定し、挿入を実行してデータベースにローを追加します。
- テーブルが空でない場合は、テーブルのロー数をレポートします。
- Table オブジェクトを閉じ、関連付けられているリソースを解放します。
- 操作が正常に完了したことを示すブール値が返されます。

2. 作成した `do_insert` メソッドを呼び出します。

次の行を、`main()` メソッドの `conn->Close` 呼び出しの直前に追加します。

```
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。

結果

アプリケーションが実行され、ULCustomer テーブルにデータを挿入できます。

次の手順

[「レッスン 3 : テーブルのローの選択とリスト」 81 ページに進みます。](#)

レッスン 3 : テーブルのローの選択とリスト

このレッスンでは、テーブルからローを取得し、それらをコマンドラインで出力します。

前提条件

このレッスンは、受講者がこれまでのすべてのレッスンを終了していることを前提としています。「[レッスン 1 : データベースの作成とデータベースへの接続](#)」75 ページを参照してください。

◆ タスク

1. 以下のメソッドを、*customer.cpp* の *do_insert* メソッドの直後に追加します。このメソッドでは、次のタスクを実行します。

- Table オブジェクトを開きます。
- カラム識別子を取得します。
- 現在の位置を、テーブル内にある最初のローの前に設定します。

テーブルに対するすべての操作は、現在の位置で実行されます。現在の位置は、最初のローの前、テーブルのいずれかのローの上、または最後のローの後ろです。この例のように、デフォルトでは、ローはプライマリキー値 (*cust_id*) に基づいて順序付けられています。別の順序でローを並べ替えるには、Ultra Light データベースにインデックスを追加し、そのインデックスを使用してテーブルを開きます。

- 各ローに対して、*cust_id* と *cust_name* の値が書き出されます。ループは、最後のローの後に *Next* メソッドが *false* を返すまで繰り返されます。
- Table オブジェクトを閉じます。

```
static bool do_select( ULConnection * conn )
{
    ULTable * table = conn->OpenTable( "ULCustomer" );
    if( table == UL_NULL ) {
        return false;
    }
    ULTableSchema * schema = table->GetTableSchema();
    if( schema == UL_NULL ) {
        table->Close();
        return false;
    }
    ul_column_num id_cid =
        schema->GetColumnID( "cust_id" );
    ul_column_num cname_cid =
        schema->GetColumnID( "cust_name" );
    schema->Close();

    _tprintf( "\n\nTable 'ULCustomer' row contents:\n" );
    while( table->Next() ) {
        ul_char cname[ MAX_NAME_LEN ];
        table->GetString( cname_cid, cname, MAX_NAME_LEN );
        _tprintf( "id=%d, name=%s\n", (int)table->GetInt(id_cid), cname );
    }
    table->Close();
    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。

結果

ULCustomer テーブル内のすべての顧客 ID と顧客名のリストが出力されます。

次の手順

[「レッスン 4 : アプリケーションへの同期の追加」 82 ページ](#)に進みます。

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、アプリケーションに同期コードを追加し、Mobile Link サーバを起動し、アプリケーションを実行して、統合されたデータベースと同期させます。

前提条件

このレッスンは、受講者がこれまでのすべてのレッスンを終了していることを前提としています。「[レッスン 1 : データベースの作成とデータベースへの接続](#)」75 ページを参照してください。

内容と備考

前のレッスンで作成した Ultra Light データベースは、Ultra Light 16 サンプルデータベースと同期します。Ultra Light 16 の Sample データベースの ULCustomer テーブルのカラムには、作成したローカル Ultra Light データベースの customer テーブルのカラムが含まれます。

◆ タスク

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。

- `EnableTcpipSynchronization` を呼び出して、TCP/IP 通信を有効にします。同期は、HTTP、HTTPS、および TLS を使用しても実行できます。
- スクリプトバージョンを設定します。Mobile Link 同期は、統合データベースに保存されているスクリプトによって制御されます。スクリプトバージョンは、使用するスクリプトセットを識別します。
- Mobile Link ユーザ名を設定します。この値は、Mobile Link サーバでの認証に使用されます。アプリケーションによっては、Mobile Link のユーザ ID と Ultra Light データベースのユーザ ID を同じに設定しますが、これらの ID はあくまでも別のものです。
- `download_only` パラメータを `true` に設定します。デフォルトでは、Mobile Link 同期は双方向です。このアプリケーションでは、テーブルのローがサンプルデータベースにアップロードされないように、ダウンロード専用同期を使用します。


```
static bool do_sync( ULConnection * conn )
{
    ul_sync_info info;
    ul_stream_error * se = &info.stream_error;

    ULDatabaseManager::EnableTcpipSynchronization();
    conn->InitSyncInfo( &info );
    info.stream = "TCP/IP";
    info.version = "custdb 12.0";
    info.user_name = "50";
    info.download_only = true;
    if( !conn->Synchronize( &info ) ) {
        _tprintf( "Synchronization error %n" );
        _tprintf( " stream_error_code is '%lu'%n", se->stream_error_code );
        _tprintf( " system_error_code is '%ld'%n", se->system_error_code );
        _tprintf( " error_string is " );
        _tprintf( "%s", se->error_string );
        _tprintf( "%n" );
        return false;
    }
    return true;
}
```

2. do_select メソッド呼び出し後に、次のコード行を main メソッドに配置します。

```
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。
4. Mobile Link サーバを起動します。

コマンドプロンプトで次のコマンドを実行します。

```
mlsrv16 -c "dsn=SQL Anywhere 16 CustDB;uid=ml_server;pwd=sql" -v -vr -vs -zu+ -o
custdbASA.log
```

-zu+ オプションを指定すると、ユーザの自動追加が行われます。-v+ オプションを指定すると、すべてのメッセージについて冗長ロギングがオンになります。

5. 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。

結果

Mobile Link サーバのメッセージウィンドウでは、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。

参照

- 「Ultra Light クライアント」『Ultra Light データベース管理とリファレンス』
- 「Mobile Link サーバオプション」『Mobile Link サーバ管理』

チュートリアルのコードリスト

これまでの項で説明したチュートリアルプログラムの完全なコードを次に示します。

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

#include "ulcpp.h"

#define MAX_NAME_LEN 100
#define MAX_ERROR_LEN 256

static ul_char const * ConnectionParms =
    "UID=DBA;PWD=sql;DBF=c:\¥¥tutorial¥¥cpp¥¥ULCustomer.udb";

ul_error_action UL_CALLBACK_FN MyErrorCallBack(
    const UL_Error * error,
    ul_void *      user_data )
{
    ul_error_action rc;
    an_sql_code code = error->GetSQLCode();

    (void) user_data;

    switch( code ){
        // The following error is used for flow control - don't report it here
        case SQLE_NOTFOUND:
            rc = UL_ERROR_ACTION_CONTINUE;
            break;

        default:
            if (code >= 0) { // warning or success
                rc = UL_ERROR_ACTION_DEFAULT;
            } else { // negative is real error
                ul_char etext[ MAX_ERROR_LEN ];
                error->GetString( etext, MAX_ERROR_LEN );
                _tprintf( "Error %ld: %s¥n", code, etext );
                rc = UL_ERROR_ACTION_CANCEL;
            }
            break;
    }
    return rc;
}

static ULConnection * open_conn( void ) {
    ULConnection * conn = ULDatabaseManager::OpenConnection( ConnectionParms );
    if( conn == UL_NULL ) {
        _tprintf( "Unable to open existing database.¥n" );
    }
    return conn;
}

static bool do_insert( ULConnection * conn ) {
    ULTable * table = conn->OpenTable( "ULCustomer" );
    if( table == UL_NULL ) {
        _tprintf( "Table not found: ULCustomer¥n" );
        return false;
    }
    if( table->GetRowCount() == 0 ) {
        _tprintf( "Inserting one row.¥n" );
        table->InsertBegin();
        table->SetString( "cust_name", "New Customer" );
        table->Insert();
        conn->Commit();
    } else {
        _tprintf( "The table has %lu rows¥n",
            table->GetRowCount() );
    }
}
```

```

        table->Close();
        return true;
    }

    static bool do_select( ULConnection * conn )
    {
        ULTable * table = conn->OpenTable( "ULCustomer" );
        if( table == UL_NULL ) {
            return false;
        }
        ULTableSchema * schema = table->GetTableSchema();
        if( schema == UL_NULL ) {
            table->Close();
            return false;
        }
        ul_column_num id_cid =
            schema->GetColumnID( "cust_id" );
        ul_column_num cname_cid =
            schema->GetColumnID( "cust_name" );

        schema->Close();

        _tprintf( "\n\nTable 'ULCustomer' row contents:\n" );

        while( table->Next() ) {
            ul_char cname[ MAX_NAME_LEN ];

            table->GetString( cname_cid, cname, MAX_NAME_LEN );

            _tprintf( "id=%d, name=%s\n", (int)table->GetInt(id_cid), cname );
        }
        table->Close();
        return true;
    }

    static bool do_sync( ULConnection * conn )
    {
        ul_sync_info info;
        ul_stream_error * se = &info.stream_error;

        ULDatabaseManager::EnableTcpipSynchronization();
        conn->InitSyncInfo( &info );
        info.stream = "TCPIP";
        info.version = "custdb 12.0";
        info.user_name = "50";
        info.download_only = true;
        if( !conn->Synchronize( &info ) ) {
            _tprintf( "Synchronization error\n" );
            _tprintf( " stream_error_code is %lu\n", se->stream_error_code );
            _tprintf( " system_error_code is %ld\n", se->system_error_code );
            _tprintf( " error_string is "" );
            _tprintf( "%s", se->error_string );
            _tprintf( ""\n" );
            return false;
        }
        return true;
    }

    int main()
    {
        ULConnection * conn;

        ULDatabaseManager::Init();
        ULDatabaseManager::SetErrorCallback( MyErrorCallBack, NULL );
    }

```

```
conn = open_conn();
if( conn == UL_NULL ){
    ULDatabaseManager::Fini();
    return 1;
}

// Main processing code goes here ...
do_insert( conn );
do_select( conn );
do_sync( conn );

conn->Close();
ULDatabaseManager::Fini();
return 0;
}
```

API リファレンス

この項では、Ultra Light C/C++ API を提供します。

Ultra Light C/C++ 共通 API リファレンス

この項では、Embedded SQL または C++ インタフェースで使える関数とマクロについて説明します。この項で説明するほとんどの関数には、SQLCA (SQL Communications Area) が必要です。

ヘッダファイル

- `ulglobal.h`

Ultra Light C/C++ アプリケーションのマクロとコンパイラディレクティブ

特に指定のないかぎり、ディレクティブは Embedded SQL と C++ API の両方のアプリケーションに適用されます。

コンパイラディレクティブは、次の場所で指定できます。

- コンパイラのコマンドライン。一般にディレクティブは `/D` オプションを使用して設定します。たとえば、ユーザ認証を使用する Ultra Light アプリケーションをコンパイルする場合、Microsoft Visual C++ の `makefile` は、次のようになります。

```
CompileOptions=/c /DPRWIN32 /Od /Zi /DWIN32  
/DUL_USE_DLL
```

```
IncludeFolders= ¥  
/I"$(VCDIR)¥include" ¥  
/I"$(SQLANY16)¥SDK¥Include"
```

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

`VCDIR` は Visual C++ ディレクトリ、`SQLANY16` は SQL Anywhere インストールディレクトリです。

- ユーザインタフェースのコンパイラ設定ウィンドウ。
- ソースコード。ディレクティブは `#define` 文を使用して指定します。

UL_USE_DLL マクロ

静的ランタイムライブラリではなく、ランタイムライブラリ DLL を使用するようにアプリケーションを設定します。

備考

Windows Mobile アプリケーションと Windows アプリケーションに適用されます。

UNDER_CE マクロ

デフォルトで、このマクロはすべての新しい Visual C++ スマートデバイスプロジェクトで定義されています。

備考

Windows Mobile アプリケーションに適用されます。

参照

- [「Windows Mobile 向け Ultra Light アプリケーション開発」 66 ページ](#)

例

```
#D UNDER_CE
```

UL_RS_STATE 列挙体

可能な結果セットまたはカーソルステータスを指定します。

構文

```
public enum UL_RS_STATE
```

メンバー

メンバー名	説明
UL_RS_STATE_ERROR	エラー。
UL_RS_STATE_UNPREPARED	準備されていません。
UL_RS_STATE_ON_ROW	有効なローの上。
UL_RS_STATE_BEFORE_FIRST	最初のローの前。
UL_RS_STATE_AFTER_LAST	最後のローの後。
UL_RS_STATE_COMPLETED	閉じています。

ul_column_sql_type 列挙体

カラムの SQL タイプを表します。

構文

```
public enum ul_column_sql_type
```

メンバー

メンバー名	説明
UL_SQLTYPE_BAD_INDEX	指定されたインデックス位置のカラムが存在しないことを表します。
UL_SQLTYPE_S_LONG	カラムに、signed long が含まれることを表します。
UL_SQLTYPE_U_LONG	カラムに、unsigned long が含まれることを表します。
UL_SQLTYPE_S_SHORT	カラムに、signed short が含まれることを表します。
UL_SQLTYPE_U_SHORT	カラムに、unsigned short が含まれることを表します。
UL_SQLTYPE_S_BIG	カラムに、64 ビット符号付き整数が含まれることを表します。
UL_SQLTYPE_U_BIG	カラムに、64 ビット符号なし整数が含まれることを表します。
UL_SQLTYPE_TINY	カラムに、8 ビット符号なし整数が含まれることを表します。
UL_SQLTYPE_BIT	カラムに、1 ビットフラグが含まれることを表します。
UL_SQLTYPE_TIMESTAMP	カラムに、タイムスタンプ情報が含まれることを表します。
UL_SQLTYPE_DATE	カラムに、日付情報が含まれることを表します。
UL_SQLTYPE_TIME	カラムに、時刻の情報が含まれることを表します。
UL_SQLTYPE_DOUBLE	カラムに、倍精度の浮動小数点数 (8 バイト) が含まれることを表します。
UL_SQLTYPE_REAL	カラムに、単精度の浮動小数点数 (4 バイト) が含まれることを表します。

メンバー名	説明
UL_SQLTYPE_NUMERIC	カラムに、精度と桁数が指定された正確な数値データが含まれることを表します。
UL_SQLTYPE_BINARY	カラムに、指定された最大長のバイナリデータが含まれることを表します。
UL_SQLTYPE_CHAR	カラムに、指定された長さの文字列データが含まれることを表します。
UL_SQLTYPE_LONGVARCHAR	カラムに、可変長の文字列データが含まれることを表します。
UL_SQLTYPE_LONGBINARY	カラムに、可変長のバイナリデータが含まれることを表します。
UL_SQLTYPE_UUID	カラムに UUID が含まれることを表します。
UL_SQLTYPE_ST_GEOMETRY	カラムに、ポイント形式の空間データが含まれることを表します。
UL_SQLTYPE_TIMESTAMP_WITH_TIME_ZONE	カラムに、タイムスタンプ情報とタイムゾーン情報が含まれることを表します。

備考

値は SQL カラム型に対応します。

ul_column_storage_type 列挙体

カラムのホスト変数タイプを表します。

構文

```
public enum ul_column_storage_type
```

メンバー

メンバー名	説明
UL_TYPE_BAD_INDEX	無効な値を表します。
UL_TYPE_S_LONG	ul_s_long (32 ビット符号付き整数) を表します。
UL_TYPE_U_LONG	ul_u_long (32 ビット符号なし整数) を表します。

メンバー名	説明
UL_TYPE_S_SHORT	ul_s_short (16 ビット符号付き整数) を表します。
UL_TYPE_U_SHORT	ul_u_short (16 ビット符号なし整数) を表します。
UL_TYPE_S_BIG	ul_s_big (64 ビット符号付き整数) を表します。
UL_TYPE_U_BIG	ul_u_big (64 ビット符号なし整数) を表します。
UL_TYPE_TINY	ul_u_byte (8 ビット符号なし) を表します。
UL_TYPE_BIT	ul_byte (8 ビット符号なし、1 ビット使用) を表します。
UL_TYPE_DOUBLE	ul_double (double) を表します。
UL_TYPE_REAL	ul_real (float) を表します。
UL_TYPE_BINARY	ul_binary (2 バイト長の後にバイト配列) を表します。
UL_TYPE_TIMESTAMP_STRUCT	DECL_DATETIME を表します。
UL_TYPE_TCHAR	文字配列 (文字列バッファ) を表します。
UL_TYPE_CHAR	char 配列 (文字列バッファ) を表します。
UL_TYPE_WCHAR	ul_wchar (UTF16) 配列を表します。
UL_TYPE_GUID	GUID 構造体を表します。

備考

これらの値は、カラムに必要なホスト変数の型を識別し、Ultra Light で値をフェッチする方法を示すために使用します。

ul_error_action 列挙体

コールバックから返される可能性があるエラーアクションを指定します。

構文

```
public enum ul_error_action
```

メンバー

メンバー名	説明
UL_ERROR_ACTION_DEFAULT	エラーコールバックがない場合と同じように動作します。
UL_ERROR_ACTION_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_TRY_AGAIN	エラーを引き起こした操作をリトライします。
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。

備考

すべてのアクションがすべてのエラーコードに適用されるわけではありません。

ul_sync_state 列挙体

同期の現在の処理を示します。

構文

```
public enum ul_sync_state
```

メンバー

メンバー名	説明
UL_SYNC_STATE_STARTING	同期を開始しています。初期パラメータの検証が完了し、同期の結果が保存されます。
UL_SYNC_STATE_CONNECTING	Mobile Link サーバに接続しています。
UL_SYNC_STATE_SENDING_HEADER	同期接続が確立されました。初期データを送信しようとしています。
UL_SYNC_STATE_SENDING_TABLE	テーブルを送信しようとしています。
UL_SYNC_STATE_SENDING_DATA	スキーマ情報またはローデータが送信されています。
UL_SYNC_STATE_FINISHING_UPLOAD	アップロード処理が完了しました。処理情報をコミットしようとしています。

メンバー名	説明
UL_SYNC_STATE_RECEIVING_UPLOAD_ACK	サーバからデータを読み込もうとしています。アップロード確認を最初に読み込みます。
UL_SYNC_STATE_RECEIVING_TABLE	テーブルを受信しようとしています。
UL_SYNC_STATE_RECEIVING_DATA	最後に識別されたテーブルのデータを受信しています。
UL_SYNC_STATE_COMMITTING_DOWNLOAD	ダウンロード処理が完了しました。ダウンロードされたローをコミットしようとしています。
UL_SYNC_STATE_ROLLING_BACK_DOWNLOAD	ダウンロード中にエラーが発生し、ダウンロードがロールバックされています。
UL_SYNC_STATE_SENDING_DOWNLOAD_ACK	ダウンロード完了の確認が送信されています。
UL_SYNC_STATE_DISCONNECTING	サーバとの接続を切断しようとしています。
UL_SYNC_STATE_DONE	同期は正常に完了しました。
UL_SYNC_STATE_ERROR	同期は完了しましたが、エラーが発生しました。

備考

この一覧の順序は同期ステータスが発生する順序とは一致しません。

ul_validate_status_id 列挙体

Ultra Light 検証ツールの可能なステータス ID を指定します。

構文

```
public enum ul_validate_status_id
```

メンバー

メンバー名	説明	値
UL_VALID_NO_ERROR	エラーは発生しませんでした。	0
UL_VALID_START	検証を開始します。	1

メンバー名	説明	値
UL_VALID_END	検証を終了します。 parm1 は、成功または失敗を示す結果の sqlcode を追跡します。	2
UL_VALID_CHECKING_PAGE	データベースページのチェック中、定期的にステータスメッセージを送信します。 Parm1 は、ページに関連付けられている数字を追跡します。順序は定義されていません。	10
UL_VALID_CHECKING_TABLE	テーブルをチェックしています。 parm1 は、テーブル名を追跡します。	20
UL_VALID_CHECKING_INDEX	インデックスをチェックしています。 parm1 はテーブル名を格納し、parm2 はインデックス名を格納します。	21
UL_VALID_HASH_REPORT	インデックスハッシュの使用についてレポートしています。 (開発バージョンのみ) parm1 はテーブル名を追跡し、parm2 はインデックス名を追跡し、parm3 は参照できるローの数を追跡し、parm4 はユニークなハッシュ値の数を追跡し、parm5 は1つのハッシュエントリが表示される最大回数を追跡します。	30

メンバー名	説明	値
UL_VALID_REDUNDANT_INDEX	冗長なインデックスが見つかりました。 (開発バージョンのみ) parm1 はテーブル名を追跡し、parm2 は冗長なインデックス名を追跡し、parm3 は冗長を発生させたインデックスの名前を追跡します。	31
UL_VALID_DUPLICATE_INDEX	2つのインデックスが同じです。 (開発バージョンのみ) parm1 はテーブル名を追跡し、parm2 は最初のインデックスの名前を追跡し、parm3 は2番目のインデックスの名前を追跡します。	32
UL_VALID_DATABASE_ERROR	データベースにアクセスするときにエラーが発生しました。 詳細については、SQLCODE を参照してください。	100
UL_VALID_STARTUP_ERROR	データベースの起動中にエラーが発生しました。 (低レベルアクセスの場合)	101
UL_VALID_CONNECT_ERROR	データベースへの接続中にエラーが発生しました。	102
UL_VALID_INTERRUPTED	検証プロセスが中断されました。	103
UL_VALID_CORRUPT_PAGE_TABLE	ページテーブルが破損しています。	110
UL_VALID_FAILED_CHECKSUM	ページチェックサムが失敗しました。 Parm1 は、ページに関連付けられている数字を追跡します。	111

メンバー名	説明	値
UL_VALID_CORRUPT_PAGE	ページが破損しています。 Parm1 は、ページに関連付けられている数字を追跡します。	112
UL_VALID_ROWCOUNT_MISMATCH	インデックス内のローの数がテーブルのロー数と異なります。 parm1 はテーブル名を追跡し、parm2 はインデックス名を追跡します。	120
UL_VALID_BAD_ROWID	インデックス内に無効なロー識別子があります。 parm1 はテーブル名を追跡し、parm2 はインデックス名を追跡します。	121

ul_binary 構造体

データベース内のテーブルからバイナリ値を設定およびフェッチします。

構文

```
public typedef struct ul_binary
```

メンバー

メンバー名	タイプ	説明
data	ul_byte	設定する実際のデータ (挿入用) またはフェッチされた実際のデータ (選択用)。
len	ul_length	値内のバイト数。

ul_error_info 構造体

Ultra Light エラーに関する完全な情報を格納します。

構文

```
public typedef struct ul_error_info
```

メンバー

メンバー名	タイプ	説明
sqlcode	an_sql_code	SQLCODE 値。
sqlcount	ul_s_long	SQLCOUNT 値。

参照

- [ULErrorInfoString メソッド \[Ultra Light Embedded SQL\]247 ページ](#)
- [ULErrorInfoURL メソッド \[Ultra Light Embedded SQL\]247 ページ](#)
- [ULErrorInfoInitFromSqlca メソッド \[Ultra Light Embedded SQL\]246 ページ](#)
- [ULErrorInfoParameterCount メソッド \[Ultra Light Embedded SQL\]247 ページ](#)
- [ULErrorInfoParameterAt メソッド \[Ultra Light Embedded SQL\]246 ページ](#)

ul_stream_error 構造体

同期の通信ストリームエラー情報を格納します。

構文

```
public typedef struct ul_stream_error
```

メンバー

メンバー名	タイプ	説明
error_string	char	stream_error_code 値のための文字列です。利用可能な場合は、追加情報が含まれます。
stream_error_code	ss_error_code	特定のストリームエラー。 取り得る値については、ss_error_code 列挙体を参照してください。
system_error_code	asa_int32	システム固有のエラーコード。 エラーコードの詳細については、プラットフォームのマニュアルを参照してください。

ul_sync_info 構造体

同期データを格納します。

構文

```
public typedef struct ul_sync_info
```

メンバー

メンバー名	タイプ	説明
additional_parms	const char *	追加のパラメータを指定した、名前と値のペアの文字列 "name=value;"。
auth_parms	const char *	Mobile Link イベントの認証パラメータの配列です。
auth_status	ul_auth_status	Mobile Link のユーザ認証のステータスです。 Mobile Link サーバが、この情報をクライアントに提供します。
auth_value	ul_s_long	カスタム Mobile Link のユーザ認証スクリプトの結果です。 Mobile Link サーバが、この情報をクライアントに提供し、認証ステータスを判断できるようにします。
download_only	ul_bool	現在の同期中は、Ultra Light データベースから変更をアップロードしません。
ignored_rows	ul_bool	無視されたローのステータスです。同期中にスクリプトがないために Mobile Link サーバによってローが 1 つでも無視されると、この読み込み専用フィールドが true をレポートします。
init_verify	ul_sync_info *	検証を初期化します。

メンバー名	タイプ	説明
keep_partial_download	ul_bool	同期時の通信エラーが原因でダウンロードが失敗すると、このパラメータは、変更をロールバックしないで部分的なダウンロードを保持するかどうかを制御します。
new_password	const char *	ユーザ名に対する新しい Mobile Link パスワードを指定する文字列です。このパラメータはオプションです。
num_auth_parms	ul_byte	Mobile Link イベントの認証パラメータに渡される認証パラメータの数。
observer	ul_sync_observer_fn	同期をモニタするコールバック関数またはイベントハンドラへのポインタです。このパラメータはオプションです。
partial_download_retained	ul_bool	同期時の通信エラーが原因でダウンロードが失敗すると、このパラメータは、変更をロールバックしないで、ダウンロードされたこの変更が適用されたかどうかを示します。
password	const char *	ユーザ名に対する既存の Mobile Link パスワードを指定する文字列です。このパラメータはオプションです。
ping	ul_bool	Ultra Light クライアントと Mobile Link サーバ間の通信を確認します。このパラメータが true に設定されている場合は、同期は行われません。
publications	const char *	同期に含めるデータを示す、パブリケーションをカンマで区切ったリストです。

メンバー名	タイプ	説明
resume_partial_download	ul_bool	失敗したダウンロードを再開します。同期によって変更はアップロードされず、失敗したダウンロードで変更のみがダウンロードされます。
send_download_ack	ul_bool	クライアントがダウンロード確認を提供するかどうかを Mobile Link サーバに指示します。
stream	const char *	同期に使用する Mobile Link ネットワークプロトコルです。
stream_error	ul_stream_error	通信エラーレポート情報を保持する構造体です。
stream_parms	const char *	選択されたネットワークプロトコルを設定するオプションです。
upload_ok	ul_bool	Mobile Link サーバにアップロードされたデータのステータスです。アップロードに成功すると、true をレポートします。
upload_only	ul_bool	現在の同期中は、統合データベースから変更をダウンロードしません。これにより、特に低速の通信リンクでは、通信時間を節約できます。
user_data	ul_void *	アプリケーション固有の情報を同期 observer で使用できるようにします。このパラメータはオプションです。
user_name	const char *	Mobile Link サーバがユニークな Mobile Link ユーザを識別するために使用する文字列です。

メンバー名	タイプ	説明
version	const char *	Ultra Light アプリケーションは、バージョン文字列により、同期スクリプトのセットから選択できます。

備考

同期パラメータは、Ultra Light データベースと Mobile Link サーバ間の同期の動作を制御します。Stream Type 同期パラメータ、User Name 同期パラメータ、Version 同期パラメータが必要です。これらが設定されていない場合、同期メソッドはエラー (SQLE_SYNC_INFO_INVALID またはこれと同じもの) を返します。Download Only、Ping、または Upload Only は一度に 1 つのみ指定できます。これらのパラメータが 1 つ以上 true に設定されると、同期メソッドはエラー (SQLE_SYNC_INFO_INVALID またはこれと同じもの) を返します。

参照

- [「Mobile Link クライアントネットワークプロトコルオプション」](#)『[Mobile Link クライアント管理](#)』

ul_sync_result 構造体

アプリケーションで適切なアクションを実行できるようにするために、同期の結果を格納します。

構文

```
public typedef struct ul_sync_result
```

メンバー

メンバー名	タイプ	説明
auth_status	ul_auth_status	同期認証ステータスです。
auth_value	ul_s_long	Mobile Link サーバが auth_status の結果を判断するために使用する値です。
error_status	ul_error_info	最後の同期のエラーステータスです。
ignored_rows	ul_bool	アップロードされたローが無視された場合は true、それ以外の場合は false。

メンバー名	タイプ	説明
partial_download_retained	ul_bool	部分的なダウンロードが保持されたことを通知する値です。「keep_partial_download」を参照してください。
received	ul_sync_stats_download	ダウンロード統計です。
sent	ul_sync_stats_upload	アップロード統計です。
stream_error	ul_stream_error	通信ストリームエラー情報です。
timestamp	SQLDATETIME	最後の同期の時刻と日付です。
upload_ok	ul_bool	アップロードが成功した場合は true、それ以外の場合は false。

ul_sync_stats_download 構造体

同期ストリームのダウンロードに関する統計情報をレポートします。

構文

```
public typedef struct ul_sync_stats_download
```

メンバー

メンバー名	タイプ	説明
bytes	ul_u_long	現在までに受信されたバイト数です。
deletes	ul_u_long	現在までに受信したローのうち削除されたものの数です。
ignored_deletes	ul_u_long	現在のダウンロードで受信されたローのうちテーブルに存在しないものの数です。
ignored_updates	ul_u_long	現在のダウンロードで受信されたローのうちテーブルにすでに存在しているものの数です。

メンバー名	タイプ	説明
inserts	ul_u_long	現在までに挿入されたローの数です。
truncate_deletes	ul_u_long	トランケート操作によって削除されたローの数です。
updates	ul_u_long	現在までに受信したローのうち更新されたものの数です。

ul_sync_stats_upload 構造体

同期ストリームのアップロードに関する統計情報をレポートします。

構文

```
public typedef struct ul_sync_stats_upload
```

メンバー

メンバー名	タイプ	説明
bytes	ul_u_long	現在までに送信されたバイト数です。
deletes	ul_u_long	現在までに送信された削除済みのローの数です。
inserts	ul_u_long	現在までに挿入されたローの数です。
updates	ul_u_long	現在までに送信された更新済みのローの数です。

ul_sync_status 構造体

同期の進行状況のモニタリングデータを返します。

構文

```
public typedef struct ul_sync_status
```

メンバー

メンバー名	タイプ	説明
current_download_row_count	ul_u_long	これまでにダウンロードされたローの数。この数には、received.inserts、received.updates、またはreceived.deletesに含まれていない、重複するローが含まれます。
db_table_count	ul_u_short	データベース内のテーブルの数です。
flags	ul_u_short	現在の同期フラグです。現在の状態に関連した追加情報を示します。
info	ul_sync_info *	ul_sync_info_a 構造体へのポインタ。
received	ul_sync_stats_download	ダウンロード統計です。
sent	ul_sync_stats_upload	アップロード統計です。
sqlca	SQLCA *	接続のアクティブな SQLCA です。
state	ul_sync_state	サポートされている多くのステータスの 1 つです。
stop	ul_bool	同期をキャンセルするブール式です。値 true は同期がキャンセルされたことを意味します。
sync_table_count	ul_u_short	同期中のテーブルの数です。
sync_table_index	ul_u_short	1 .. sync_table_count
table_id	ul_u_short	現在アップロードまたはダウンロードされているテーブルの ID です (1 から始まります)。同期されないテーブルがある場合には、この番号で値がスキップされることがあります。また、番号が必ず増加するとはかぎりません。

メンバー名	タイプ	説明
table_name	char	現在のテーブルの名前。
table_name_w2	ul_wchar	現在のテーブルの名前。
total_download_row_count	ul_u_long	ダウンロードで受信するローの合計数。この数には、received.inserts、received.updates、received.deletes のいずれにも含まれなし重複ローが含まれます。このフィールドは、同期によって最初のテーブルの UL_SYNC_STATE_RECEIVING_TABLE ステータスが入力されるまで設定されません。
user_data	ul_void *	ULSetSynchronizationCallback メソッドに渡されたユーザーデータ、または ul_sync_info 構造体で設定されたユーザーデータです。

参照

- [ul_sync_state 列挙体 \[Ultra Light C および Embedded SQL データタイプ\]92 ページ](#)

ul_validate_data 構造体

コールバックの検証ステータス情報を格納します。

構文

```
public typedef struct ul_validate_data
```

メンバー

メンバー名	タイプ	説明
i	ul_u_long	整数としてのパラメータです。
parm_count	ul_u_short	構造体内のパラメータ数です。

メンバー名	タイプ	説明
parm_type	enumeration	指定可能なパラメータタイプです。
parms	struct ul_validate_data::@3	パラメータの配列です。
s	char	文字列としてのパラメータです (ワイド文字ではありません)。
status_id	ul_validate_status_id	検証プロセスでレポートされる対象を示します。
stop	ul_bool	検証をキャンセルするブール式です。値 true は検証がキャンセルされたことを意味します。
type	parm_type	格納されるパラメータのタイプです。
user_data	ul_void *	検証ルーチンに渡されるユーザ定義のデータポインタです。

ULVF_DATABASE 変数

データベースを検証するために使用します。

構文

```
#define ULVF_DATABASE
```

備考

ページのチェックサムやその他のチェックを使用してデータベースページを検証します。

参照

- [ULDatabaseManager.ValidateDatabase メソッド \[Ultra Light C++\]147 ページ](#)
- [ULConnection.ValidateDatabase メソッド \[Ultra Light C++\]136 ページ](#)

ULVF_EXPRESS 変数

完全ではないが、高速な検証を実行するために使用します。

構文

```
#define ULVF_EXPRESS
```

備考

このフラグは他のフラグと組み合わせることで動作を変更させます。

参照

- [ULDatabaseManager.ValidateDatabase メソッド \[Ultra Light C++\]147 ページ](#)
- [ULConnection.ValidateDatabase メソッド \[Ultra Light C++\]136 ページ](#)

ULVF_FULL_VALIDATE 変数

データベース上ですべてのタイプの検証を実行します。

構文

```
#define ULVF_FULL_VALIDATE
```

参照

- [ULDatabaseManager.ValidateDatabase メソッド \[Ultra Light C++\]147 ページ](#)
- [ULConnection.ValidateDatabase メソッド \[Ultra Light C++\]136 ページ](#)

ULVF_IDX_HASH 変数

インデックスのハッシュの有効性をレポートします (開発バージョンのみ)。

構文

```
#define ULVF_IDX_HASH
```

備考

テーブルとインデックスのローカウントが一致しているかどうかをチェックします。

参照

- [ULDatabaseManager.ValidateDatabase メソッド \[Ultra Light C++\]147 ページ](#)
- [ULConnection.ValidateDatabase メソッド \[Ultra Light C++\]136 ページ](#)

ULVF_IDX_REDUNDANT 変数

冗長なインデックスをチェックします (開発バージョンのみ)。

構文

```
#define ULVF_IDX_REDUNDANT
```

備考

テーブルとインデックスのローカウントが一致しているかどうかをチェックします。

参照

- [ULDatabaseManager.ValidateDatabase メソッド \[Ultra Light C++\]147 ページ](#)
- [ULConnection.ValidateDatabase メソッド \[Ultra Light C++\]136 ページ](#)

ULVF_INDEX 変数

インデックスを検証するために使用します。

構文

```
#define ULVF_INDEX
```

備考

インデックスの整合性をチェックします。

参照

- [ULDatabaseManager.ValidateDatabase メソッド \[Ultra Light C++\]147 ページ](#)
- [ULConnection.ValidateDatabase メソッド \[Ultra Light C++\]136 ページ](#)

ULVF_TABLE 変数

テーブルを検証するために使用します。

構文

```
#define ULVF_TABLE
```

備考

テーブルとインデックスのローカウントが一致しているかどうかをチェックします。

参照

- [ULDatabaseManager.ValidateDatabase メソッド \[Ultra Light C++\]147 ページ](#)
- [ULConnection.ValidateDatabase メソッド \[Ultra Light C++\]136 ページ](#)

UL_AS_SYNCHRONIZE 変数

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。

構文

```
#define UL_AS_SYNCHRONIZE
```

備考

これは、ActiveSync を使用する Windows Mobile アプリケーションのみに適用されます。

参照

- [「ActiveSync の同期設定」 70 ページ](#)

UL_SYNC_ALL 変数

どのパブリケーションにも含まれていないテーブルも含め、データベース内の "no sync" のマークが付いていないすべてのテーブルを同期させます。

構文

```
#define UL_SYNC_ALL
```

参照

- [ul_sync_info 構造体 \[Ultra Light C および Embedded SQL データタイプ\]98 ページ](#)
- [UL_SYNC_ALL_PUBS 変数 \[Ultra Light C および Embedded SQL データタイプ\]109 ページ](#)

UL_SYNC_ALL_PUBS 変数

パブリケーション内のすべてのテーブルを同期させます。

構文

```
#define UL_SYNC_ALL_PUBS
```

参照

- [ul_sync_info 構造体 \[Ultra Light C および Embedded SQL データタイプ\]98 ページ](#)
- [UL_SYNC_ALL 変数 \[Ultra Light C および Embedded SQL データタイプ\]109 ページ](#)

UL_SYNC_STATUS_FLAG_IS_BLOCKING 変数

ul_sync_status.flags フィールドでビット配列を定義して、Mobile Link サーバからの応答を待機中、同期はブロックされていることを示します。

構文

```
#define UL_SYNC_STATUS_FLAG_IS_BLOCKING
```

備考

この状況が続く間は、同じ状態を示す同期進行状況メッセージが定期的に生成されます。

UL_TEXT 変数

シングルス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。

構文

```
#define UL_TEXT
```

備考

アプリケーションをコンパイルして文字列の Unicode 表現と非 Unicode 表現を使用する場合は、このマクロを使用してすべての定数文字列を囲みます。このマクロは、あらゆる環境やプラットフォームで文字列を適切に定義します。

UL_VALID_IS_ERROR 変数

指定された `ul_validate_status_id` がエラーステータスの場合、`true` と評価します。

構文

```
#define UL_VALID_IS_ERROR
```

UL_VALID_IS_INFO 変数

指定された `ul_validate_status_id` が情報ステータスの場合、`true` と評価します。

構文

```
#define UL_VALID_IS_INFO
```

Ultra Light C/C++ API リファレンス

次に、よく使用される API オブジェクトの一部を示します。

- **ULDatabaseManager** データベース接続を管理するメソッド (`CreateDatabase`、`OpenConnection` など) を提供します。
- **ULConnection** Ultra Light データベースへの接続を表します。ULConnection オブジェクトは 1 つまたは複数作成できます。
- **ULTable** データベースのテーブルへの直接アクセスを提供します。
- **ULPreparedStatement**、**ULResultSet**、**ULResultSetSchema** 動的 SQL 文の作成、クエリの記述、INSERT、UPDATE、DELETE 文の実行、プログラムによるデータベースの結果セットの制御を行います。

ヘッダファイル

- `ulcpp.h`

参照

- [ULDatabaseManager クラス \[Ultra Light C++\]137 ページ](#)
- [ULConnection クラス \[Ultra Light C++\]111 ページ](#)
- [ULTable クラス \[Ultra Light C++\]210 ページ](#)
- [ULPreparedStatement クラス \[Ultra Light C++\]159 ページ](#)
- [ULResultSet クラス \[Ultra Light C++\]169 ページ](#)
- [ULResultSetSchema クラス \[Ultra Light C++\]205 ページ](#)

ULConnection クラス

Ultra Light データベースへの接続を表します。

構文

```
public class ULConnection
```

メンバー

継承されたメンバーを含む ULConnection クラスのすべてのメンバー。

名前	説明
CancelGetNotification メソッド	指定された名前に一致する、すべてのキューに登録されている保留中の取得通知コールをキャンセルします。
ChangeEncryptionKey メソッド	Ultra Light データベースのデータベース暗号化キーを変更します。
Checkpoint メソッド	チェックポイント操作を実行し、保留中になっているコミット済みトランザクションをデータベースにフラッシュします。
Close メソッド	この接続と、残りの関連オブジェクトを破棄します。
Commit メソッド	現在のトランザクションをコミットします。
CountUploadRows メソッド	同期するためにアップロードする必要があるローの数を数えます。
CreateNotificationQueue メソッド	この接続のイベント通知キューを作成します。
DeclareEvent メソッド	登録およびトリガされるイベントを宣言します。
DestroyNotificationQueue メソッド	指定されたイベント通知キューを破棄します。

名前	説明
ExecuteScalar メソッド	SQL SELECT 文を直接実行し、単一の結果を返します。
ExecuteScalarV メソッド	代入値のリストを指定して、SQL SELECT 文の文字列を実行します。
ExecuteStatement メソッド	SQL 文の文字列を直接実行します。
GetChildObjectCount メソッド	接続で現在開いている子オブジェクトの数を取得します。
GetDatabaseProperty メソッド	データベースプロパティの値を取得します。
GetDatabasePropertyInt メソッド	データベースプロパティの整数値を取得します。
GetDatabaseSchema メソッド	データベースのスキーマを問い合わせるために使用されるオブジェクトポインタを返します。
GetLastDownloadTime メソッド	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。
GetLastError メソッド	最後の呼び出しに関連付けられているエラー情報を返します。
GetLastIdentity メソッド	@@identity の値を取得します。
GetNotification メソッド	イベント通知を読み込みます。
GetNotificationParameter メソッド	GetNotification メソッドによって読み込まれたイベント通知のパラメータを取得します。
GetSqlca メソッド	この接続に関連付けられている SQLCA を取得します。
GetSyncResult メソッド	前回の同期結果を取得します。
GetUserPointer メソッド	SetUserPointer メソッドによって最後に設定されたポインタ値を取得します。
GlobalAutoincUsage メソッド	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。

名前	説明
GrantConnectTo メソッド	指定されたパスワードを持つ新しいまたは既存のユーザ ID に、Ultra Light データベースへのアクセスを許可します。
InitSyncInfo メソッド	同期情報の構造体を初期化します。
OpenTable メソッド	テーブルを開きます。
PrepareStatement メソッド	SQL 文の準備を行います。
RegisterForEvent メソッド	イベントの通知を受け取るためのキューを登録または登録解除します。
ResetLastDownloadTime メソッド	アプリケーションが以前にダウンロードされたデータを再同期するように、パブリケーションの最終ダウンロード時間をリセットします。
RevokeConnectFrom メソッド	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。
Rollback メソッド	現在のトランザクションをロールバックします。
RollbackPartialDownload メソッド	失敗した同期からの変更をロールバックします。
SendNotification メソッド	指定された名前と一致するすべてのキューに通知を送信します。
SetDatabaseOption メソッド	指定されたデータベースオプションを設定します。
SetDatabaseOptionInt メソッド	データベースオプションを設定します。
SetSynchronizationCallback メソッド	同期の実行中に呼び出されるようコールバックを設定します。
SetSyncInfo メソッド	指定された <code>ul_sync_info</code> 構造体に基づいて、指定された名前を使用して同期プロファイルを作成します。
SetUserPointer メソッド	呼び出すアプリケーションによって接続に使用される任意のポインタ値を設定します。
StartSynchronizationDelete メソッド	この接続の START SYNCHRONIZATION DELETE を設定します。

名前	説明
StopSynchronizationDelete メソッド	この接続の STOP SYNCHRONIZATION DELETE を設定します。
Synchronize メソッド	Ultra Light アプリケーションで同期を開始します。
SynchronizeFromProfile メソッド	指定されたプロファイルとマージパラメータを使用して、データベースを同期します。
TriggerEvent メソッド	ユーザ定義のイベントをトリガして、登録されたすべてのキューに通知を送信します。
ValidateDatabase メソッド	この接続のデータベースを検証します。

CancelGetNotification メソッド

指定された名前に一致する、すべてのキューに登録されている保留中の取得通知コールをキャンセルします。

構文

```
public virtual ul_u_long CancelGetNotification(const char * queueName)
```

パラメータ

- **queueName** キューの名前。

戻り値

影響を受けるキューの数。(必ずしも、ブロックされた読み込みの数ではありません)

ChangeEncryptionKey メソッド

Ultra Light データベースのデータベース暗号化キーを変更します。

構文

```
public virtual bool ChangeEncryptionKey(const char * newKey)
```

パラメータ

- **newkey** データベースの新しい暗号化キー。

戻り値

成功した場合は true、失敗した場合は false。

備考

このメソッドを呼び出すアプリケーションでは、データベースが同期されていること、または信頼できるバックアップコピーが作成されていることを、先に確認しておく必要があります。
ChangeEncryptionKey メソッドは、完了まで実行する必要がある操作であるため、信頼できるバックアップがあることが重要です。データベース暗号化キーを変更すると、まずデータベースのすべてのローは古いキーを使用して復号され、次に新しいキーを使用して再度暗号化されて、書き込まれます。この操作は元に戻せません。暗号化変更処理が完了しなかった場合、データベースは無効な状態のままになり、再度アクセスすることはできなくなります。

Checkpoint メソッド

チェックポイント操作を実行し、保留中になっているコミット済みトランザクションをデータベースにフラッシュします。

構文

```
public virtual bool Checkpoint()
```

戻り値

成功した場合は **true**、失敗した場合は **false**。

備考

Checkpoint メソッドを呼び出しても、現在のトランザクションすべてがコミットされるわけではありません。このメソッドは、パフォーマンスを向上させるために後回しにされた自動トランザクションチェックポイントとともに (**commit_flush** 接続パラメータを使用して) 使用されます。

Checkpoint メソッドを使用すると、保留中のコミット済みトランザクションがすべてデータベースの記憶領域に書き込まれることが保証されます。

Close メソッド

この接続と、残りの関連オブジェクトを破棄します。

構文

```
public virtual void Close(ULError * error)
```

パラメータ

- **error** エラー情報を受信するためのオプションの **ULError** オブジェクト。

Commit メソッド

現在のトランザクションをコミットします。

構文

```
public virtual bool Commit()
```

戻り値

成功した場合は `true`、失敗した場合は `false`。

CountUploadRows メソッド

同期するためにアップロードする必要があるローの数を数えます。

構文

```
public virtual ul_u_long CountUploadRows (  
    const char * pubList,  
    ul_u_long threshold  
)
```

パラメータ

- **pubList** チェック対象となるパブリケーションのカンマ区切りのリストを含む文字列。空の文字列 (`UL_SYNC_ALL` マクロ) は、「非同期」とマーク付けされたものを除くすべてのテーブルを表します。アスタリスクのみの文字列 (`UL_SYNC_ALL_PUBS` マクロ) は、いずれかのパブリケーションで参照されているすべてのテーブルを表します。一部のテーブルは、どのパブリケーションの一部でもないため、この値が "*" の場合は含まれません。
- **threshold** カウントするローの最大数を判断します。呼び出しの所要時間を制限します。`threshold` が 0 の場合、制限はありません (つまり、同期する必要のあるすべてのローをカウントします)。また、`threshold` が 1 の場合、同期の必要なローがあるかどうかを簡単に判別するために使用できます。

戻り値

指定されたパブリケーションのセットまたはデータベース全体のいずれかで、同期を必要とするローの数。

備考

このメソッドを使用すると、ユーザは同期、または自動バックグラウンド同期が行われるタイミングの決定を求められます。

次の呼び出しでは、データベース全体をチェックして、同期させるローの総数を確認します。

```
count = conn->CountUploadRows( UL_SYNC_ALL, 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 = conn->CountUploadRows( "PUB1,PUB2", 1000 );
```

次の呼び出しでは、パブリケーション PUB1 と PUB2 で同期させる必要のあるローがあるかどうかをチェックされます。

```
anyToSync = conn->CountUploadRows( "PUB1,PUB2", 1 ) != 0;
```

CreateNotificationQueue メソッド

この接続のイベント通知キューを作成します。

構文

```
public virtual bool CreateNotificationQueue(  
    const char * name,  
    const char * parameters  
)
```

パラメータ

- **name** 新しいキューの名前。
- **parameters** 予約済み。NULL に設定されます。

戻り値

成功した場合は true、失敗した場合は false。

備考

キュー名は、接続ごとにスコープされるため、別々の接続で同じ名前を持つキューを作成できません。イベント通知が送信されると、データベース内で一致する名前を持つすべてのキューが、個別のインスタンスの通知を受け取ります。名前では、大文字と小文字が区別されません。RegisterForEvent メソッドを呼び出したときに、キューが指定されていない場合は、接続ごとにデフォルトのキューが作成されます。その名前がすでに存在する場合や有効でない場合は、エラーが発生して呼び出しが失敗します。

参照

- [ULConnection.RegisterForEvent メソッド \[Ultra Light C++\]129 ページ](#)

DeclareEvent メソッド

登録およびトリガされるイベントを宣言します。

構文

```
public virtual bool DeclareEvent(const char * eventName)
```

パラメータ

- **eventName** 新しいユーザ定義イベントの名前。

戻り値

イベントが正常に宣言された場合は true。正常に宣言されず、名前がすでに使用されているか無効な場合は false。

備考

Ultra Light では、データベースまたは環境での操作によってトリガされるシステムイベントの一部が事前に定義されています。このメソッドは、ユーザ定義イベントを宣言します。ユーザ定義

イベントは、`TriggerEvent` メソッドでトリガされます。イベント名は、ユニークにする必要があります。名前では、大文字と小文字が区別されません。

参照

- [ULConnection.TriggerEvent メソッド \[Ultra Light C++\]135 ページ](#)

DestroyNotificationQueue メソッド

指定されたイベント通知キューを破棄します。

構文

```
public virtual bool DestroyNotificationQueue(const char * name)
```

パラメータ

- **name** 破棄するキューの名前。

戻り値

成功した場合は `true`、失敗した場合は `false`。

備考

キューに未読の通知がある場合は、警告が表示されます。未読通知は破棄されます。接続のデフォルトのイベントキューが作成されている場合、接続が閉じると破棄されます。

ExecuteScalar メソッド

SQL SELECT 文を直接実行し、単一の結果を返します。

構文

```
public virtual bool ExecuteScalar(  
    void * dstPtr,  
    size_t dstSize,  
    ul_column_storage_type dstType,  
    const char * sql,  
    ...  
)
```

パラメータ

- **dstPtr** 値を受信するための必要な型の変数へのポインタ。
- **dstSize** 値を受信するための変数のサイズ (適用できる場合)。
- **dstType** 取得する値の型。この値は、変数の型と一致する必要があります。
- **sql** SELECT 文。オプションで '?' パラメータが含まれます。
- **...** 代入する文字列 (`char *`) パラメータ値。

戻り値

クエリが正常に実行され、値が正常に取得される場合は **true**。それ以外の場合で値がフェッチされないときは **false**。SQLCODE エラーコードをチェックして、**false** が返される理由を特定します。警告またはエラー (SQLE_NOERROR) が示されない場合、選択した値は NULL になります。

備考

dstPtr 値は、dstType 値に一致する正しい型の変数を指す必要があります。dstSize パラメータは、文字列やバイナリなどの可変サイズの値にのみ必要とされ、それ以外の場合は無視されます。パラメータ値の変数リストは、文のパラメータに対応している必要があり、すべての値は文字列であると想定されます。(内部的には、Ultra Light は、文で必要とされるときにパラメータ値をキャストします)

次の型がサポートされます。

- **UL_TYPE_BIT/UL_TYPE_TINY** 変数の型 ul_byte (8 ビット、符号なし) を使用します。
- **UL_TYPE_U_SHORT/UL_TYPE_S_SHORT** 変数の型 ul_u_short/ul_s_short (16 ビット) を使用します。
- **UL_TYPE_U_LONG/UL_TYPE_S_LONG** 変数の型 ul_u_long/ul_s_long (32 ビット) を使用します。
- **UL_TYPE_U_BIG/UL_TYPE_S_BIG** 変数の型 ul_u_big/ul_s_big (64 ビット) を使用します。
- **UL_TYPE_DOUBLE** 変数の型 ul_double (double) を使用します。
- **UL_TYPE_REAL** 変数の型 ul_real (float) を使用します。
- **UL_TYPE_BINARY** 変数の型 ul_binary を使用し、dstSize を指定します (GetBinary() の場合と同様)。
- **UL_TYPE_TIMESTAMP_STRUCT** 変数の型 DECL_DATETIME を使用します。
- **UL_TYPE_CHAR** 変数の型 char [] (文字バッファ) を使用し、dstSize にバッファのサイズを設定します (GetString() の場合と同様)。
- **UL_TYPE_WCHAR** 変数の型 ul_wchar [] (ワイド文字バッファ) を使用し、dstSize にバッファのサイズを設定します (GetString() の場合と同様)。
- **UL_TYPE_TCHAR** どちらのバージョンのメソッドが呼び出されているかに応じて、UL_TYPE_CHAR または UL_TYPE_WCHAR と同様。

次の例は、整数のフェッチを示しています。

```
ul_u_long val;
ok = conn->ExecuteScalar( &val, 0, UL_TYPE_U_LONG,
    "SELECT count(*) FROM t WHERE col LIKE ?", "ABC%" );
```

次の例は、文字列のフェッチを示しています。

```
char val[40];
ok = conn->ExecuteScalar( &val, sizeof(val), UL_TYPE_CHAR,
    "SELECT uuidtostr( newid() )" );
```

参照

- [ul_column_storage_type](#) 列挙体 [Ultra Light C および Embedded SQL データタイプ]90 ページ

ExecuteScalarV メソッド

代入値のリストを指定して、SQL SELECT 文の文字列を実行します。

構文

```
public virtual bool ExecuteScalarV(
    void * dstPtr,
    size_t dstSize,
    ul_column_storage_type dstType,
    const char * sql,
    va_list args
)
```

パラメータ

- **dstPtr** 値を受信するための必要な型の変数へのポインタ。
- **dstSize** 値を受信するための変数のサイズ (適用できる場合)。
- **dstType** 取得する値の型。この値は、変数の型と一致する必要があります。
- **sql** SELECT 文。オプションで '?' パラメータが含まれます。
- **args** 代入する文字列 (char *) 値のリスト。

戻り値

クエリが正常に実行され、値が正常に取得される場合は **true**。それ以外の場合で値がフェッチされないときは **false**。SQLCODE エラーコードをチェックして、**false** が返される理由を特定します。警告またはエラー (SQLE_NOERROR) が示されない場合、選択した値は NULL になります。

備考

dstPtr 値は、dstType 値に一致する正しい型の変数を指す必要があります。dstSize パラメータは、文字列やバイナリなどの可変サイズの値にのみ必要とされ、それ以外の場合は無視されます。パラメータ値の変数リストは、文のパラメータに対応している必要があり、すべての値は文字列であると想定されます。(内部的には、Ultra Light は、文で必要とされるときにパラメータ値をキャストします)

次の型がサポートされます。

- **UL_TYPE_BIT/UL_TYPE_TINY** 変数の型 ul_byte (8 ビット、符号なし) を使用します。
- **UL_TYPE_U_SHORT/UL_TYPE_S_SHORT** 変数の型 ul_u_short/ul_s_short (16 ビット) を使用します。

- **UL_TYPE_U_LONG/UL_TYPE_S_LONG** 変数の型 `ul_u_long/ul_s_long` (32 ビット) を使用します。
- **UL_TYPE_U_BIG/UL_TYPE_S_BIG** 変数の型 `ul_u_big/ul_s_big` (64 ビット) を使用します。
- **UL_TYPE_DOUBLE** 変数の型 `ul_double` (double) を使用します。
- **UL_TYPE_REAL** 変数の型 `ul_real` (float) を使用します。
- **UL_TYPE_BINARY** 変数の型 `ul_binary` を使用し、**dstSize** を指定します (`GetBinary()` の場合と同様)。
- **UL_TYPE_TIMESTAMP_STRUCT** 変数の型 `DECL_DATETIME` を使用します。
- **UL_TYPE_CHAR** 変数の型 `char []` (文字バッファ) を使用し、**dstSize** にバッファのサイズを設定します (`GetString()` の場合と同様)。
- **UL_TYPE_WCHAR** 変数の型 `ul_wchar []` (ワイド文字バッファ) を使用し、**dstSize** にバッファのサイズを設定します (`GetString()` の場合と同様)。
- **UL_TYPE_TCHAR** どちらのバージョンのメソッドが呼び出されているかに応じて、`UL_TYPE_CHAR` または `UL_TYPE_WCHAR` と同様。

参照

- [ul_column_storage_type](#) 列挙体 [Ultra Light C および Embedded SQL データタイプ]90 ページ

ExecuteStatement メソッド

SQL 文の文字列を直接実行します。

構文

```
public virtual bool ExecuteStatement(const char * sql)
```

パラメータ

- **sql** 実行する SQL スクリプト。

戻り値

成功した場合は `true`、失敗した場合は `false`。

備考

このメソッドを使用して、`SELECT` 文を直接実行し、単一の結果を取り出します。

`PrepareStatement` メソッドを使用して、変数パラメータと共に文を繰り返し実行するか、複数の結果をフェッチします。

参照

- [ULConnection.PrepareStatement](#) メソッド [Ultra Light C++]128 ページ

GetChildObjectCount メソッド

接続で現在開いている子オブジェクトの数を取得します。

構文

```
public virtual ul_u_long GetChildObjectCount()
```

戻り値

現在開いている子オブジェクトの数。

備考

このメソッドを使用してオブジェクトのリークを検出できます。

GetDatabaseProperty メソッド

データベースプロパティの値を取得します。

構文

```
public virtual const char * GetDatabaseProperty(const char * propName)
```

パラメータ

- **propName** 要求されているプロパティの名前。

戻り値

正常に実行された場合は、データベースプロパティの値が格納された文字列バッファへのポインタ。失敗した場合は NULL。

備考

戻り値は静的バッファを指します。静的バッファの内容は、それ以降の Ultra Light の呼び出しによって変更される可能性があるため、値を保存しておきたい場合はその値のコピーを作成してください。

参照

- 「Ultra Light データベースプロパティ」『Ultra Light データベース管理とリファレンス』

例

次の例では、CharSet データベースプロパティの値を取得する方法を示します。

```
const char * charset = GetDatabaseProperty( "CharSet" );
```

GetDatabasePropertyInt メソッド

データベースプロパティの整数値を取得します。

構文

```
public virtual ul_u_long GetDatabasePropertyInt(const char * propName)
```

パラメータ

- **propName** 要求されているプロパティの名前。

戻り値

成功した場合は、プロパティの整数値。それ以外の場合は 0。

参照

- 「[Ultra Light データベースプロパティ](#)」『[Ultra Light データベース管理とリファレンス](#)』

例

次の例では、ConnCount データベースプロパティの値を取得する方法を示します。

```
unsigned connectionCount = GetDatabasePropertyInt( "ConnCount" );
```

GetDatabaseSchema メソッド

データベースのスキーマを問い合わせるために使用されるオブジェクトポインタを返します。

構文

```
public virtual ULDatabaseSchema * GetDatabaseSchema()
```

戻り値

データベースのスキーマを問い合わせるために使用される ULDatabaseSchema オブジェクト。

GetLastDownloadTime メソッド

指定したパブリケーションが最後にダウンロードされた時刻を取得します。

構文

```
public virtual bool GetLastDownloadTime(  
    const char * publication,  
    DECL_DATETIME * value  
)
```

パラメータ

- **publication** パブリケーション名。
- **value** 投入する DECL_DATETIME 構造体へのポインタ。値 January 1, 1900 は、パブリケーションがまだ同期されていないか、時間がリセットされたことを示します。

戻り値

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場合は true。それ以外の場合は false。

備考

次の呼び出しでは、'pub1' パブリケーション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;  
ok = conn->GetLastDownloadTime( "pub1", &dt );
```

GetLastError メソッド

最後の呼び出しに関連付けられているエラー情報を返します。

構文

```
public virtual const ULError * GetLastError()
```

戻り値

最後の呼び出しに関連付けられている情報を含む ULError オブジェクトへのポインタ。

備考

アドレスが返されるエラーオブジェクトは、接続が開いている間には有効ですが、以降の呼び出しで自動的に更新されるわけではありません。GetLastError を呼び出して、更新されたステータス情報を取得する必要があります。

参照

- [ULError クラス \[Ultra Light C++\]151 ページ](#)

GetLastIdentity メソッド

@@identity の値を取得します。

構文

```
public virtual ul_u_big GetLastIdentity()
```

戻り値

オートインクリメントカラムまたはグローバルオートインクリメントカラムに最後に挿入された値。

備考

この値は、データベースのオートインクリメントカラムまたはグローバルオートインクリメントカラムに最後に挿入された値です。データベースが停止している場合、この値は記録されません。このため、オートインクリメントの値が挿入される前にこのメソッドを呼び出すと、0 が返されます。

注意

最後に挿入された値が別の接続の値である可能性があります。

GetNotification メソッド

イベント通知を読み込みます。

構文

```
public virtual const char * GetNotification(  
    const char * queueName,  
    ul_u_long waitms  
)
```

パラメータ

- **queueName** 読み取るキュー。または、デフォルト接続キューの場合は NULL。
- **waitms** 返す前に、待機(ブロック)する時間(ミリ秒単位)。

戻り値

読み込まれたイベントの名前。エラーが発生した場合は NULL。

備考

この呼び出しは、通知が受信されるか、または指定された待機期間が終了するまでブロックします。無期限に待機するには、waitms パラメータを UL_READ_WAIT_INFINITE に設定します。待機をキャンセルするには、指定したキューに別の通知を送信するか、CancelGetNotification メソッドを使用します。通知を読み込んだ後に GetNotificationParameter メソッドを使用して、追加のパラメータを名前を取得します。

参照

- [ULConnection.CancelGetNotification メソッド \[Ultra Light C++\]114 ページ](#)
- [ULConnection.GetNotificationParameter メソッド \[Ultra Light C++\]125 ページ](#)

GetNotificationParameter メソッド

GetNotification メソッドによって読み込まれたイベント通知のパラメータを取得します。

構文

```
public virtual const char * GetNotificationParameter(  
    const char * queueName,  
    const char * parameterName  
)
```

パラメータ

- **queueName** 読み取るキュー。デフォルト接続キューの場合は NULL。
- **parameterName** 読み込むパラメータの名前(または "*")。

戻り値

パラメータ値。エラーが発生した場合は NULL。

備考

指定されたキューで最近読み込まれた通知のパラメータのみが使用可能です。パラメータは名前によって取得されます。パラメータ名を "*" と指定すると、パラメータ文字列全体が取得されます。

参照

- [ULConnection.GetNotification メソッド \[Ultra Light C++\]125 ページ](#)

GetSqlca メソッド

この接続に関連付けられている SQLCA を取得します。

構文

```
public virtual SQLCA * GetSqlca()
```

戻り値

この接続の SQLCA オブジェクトへのポインタ。

GetSyncResult メソッド

前回の同期結果を取得します。

構文

```
public virtual bool GetSyncResult(ul_sync_result * syncResult)
```

パラメータ

- **syncResult** 投入する ul_sync_result 構造体へのポインタ。

戻り値

成功した場合は true、失敗した場合は false。

参照

- [ul_sync_result 構造体 \[Ultra Light C および Embedded SQL データタイプ\]101 ページ](#)

GetUserPointer メソッド

SetUserPointer メソッドによって最後に設定されたポインタ値を取得します。

構文

```
public virtual void * GetUserPointer()
```

参照

- [ULConnection.SetUserPointer メソッド \[Ultra Light C++\]133 ページ](#)

GlobalAutoincUsage メソッド

グローバルオートインクリメントのデフォルト値を持つすべてのカラムで、デフォルト値が使用されている比率 (%) を取得します。

構文

```
public virtual ul_u_short GlobalAutoincUsage()
```

戻り値

グローバルオートインクリメントの値のカウンタによる使用済み比率 (%)。

備考

このデフォルト値を使用するカラムがデータベース内に複数含まれている場合は、すべてのカラムに対してこの値が計算され、最大値が返されます。たとえば、戻り値 99 は、少なくとも 1 つのカラムではデフォルト値が残されているが、きわめて少ないことを示します。

GrantConnectTo メソッド

指定されたパスワードを持つ新しいまたは既存のユーザ ID に、Ultra Light データベースへのアクセスを許可します。

構文

```
public virtual bool GrantConnectTo(const char * uid, const char * pwd)
```

パラメータ

- **uid** ユーザ ID を保持する文字配列。最大長は 31 文字です。
- **pwd** ユーザ ID のパスワードを保持する文字配列。

戻り値

成功した場合は true、失敗した場合は false。

備考

このメソッドは、既存のユーザ ID を指定したときに、既存のユーザのパスワードを更新します。

参照

- [ULConnection.RevokeConnectFrom メソッド \[Ultra Light C++\]130 ページ](#)

InitSyncInfo メソッド

同期情報の構造体を初期化します。

構文

```
public virtual void InitSyncInfo(ul_sync_info * info)
```

パラメータ

- **info** 同期パラメータを保持する `ul_sync_info` 構造体へのポインタ。

備考

このメソッドを呼び出してから、`ul_sync_info` structure のフィールドの値を設定するようにしてください。

OpenTable メソッド

テーブルを開きます。

構文

```
public virtual ULTable * OpenTable(  
    const char * tableName,  
    const char * indexName  
)
```

パラメータ

- **tableName** 開くテーブルの名前。
- **indexName** テーブルを開く場合に使用するインデックスの名前。プライマリキーを使用してテーブルを開く場合は NULL、順序付けなしでテーブルを開く場合は空の文字列を渡します。

戻り値

呼び出しが成功した場合は `ULTable` オブジェクト。失敗した場合は NULL。

備考

アプリケーションがテーブルを初めて開いたときは、カーソルの位置が最初のローの前に設定されます。

PrepareStatement メソッド

SQL 文の準備を行います。

構文

```
public virtual ULPreparedStatement * PrepareStatement(const char * sql)
```

パラメータ

- **sql** 準備する SQL 文。

戻り値

成功した場合は `ULPreparedStatement` オブジェクト、それ以外の場合は NULL。

RegisterForEvent メソッド

イベントの通知を受け取るためのキューを登録または登録解除します。

構文

```
public virtual bool RegisterForEvent(
    const char * eventName,
    const char * objectName,
    const char * queueName,
    bool register_not_unreg
)
```

パラメータ

- **eventName** 登録するシステム定義またはユーザ定義のイベント。
- **objectName** イベントを適用するオブジェクト。(テーブル名など)。
- **queueName** NULL は、デフォルトの接続キューの使用を表します。
- **register_not_unreg** 登録する場合は true、登録解除する場合は false を設定します。

戻り値

正常に登録できた場合は true。正常に登録できず、キューまたはイベントが存在しない場合は false。

備考

キュー名が指定されていない場合は、デフォルトの接続キューが暗黙で指定され、必要に応じて作成されます。特定のシステムイベントでは、そのイベントが適用されるオブジェクト名を指定できます。たとえば、TableModified イベントではテーブル名を指定できます。SendNotification メソッドとは異なり、登録された特定のキューのみイベントの通知を受信します。別の接続に、同じ名前前の他のキューがある場合、それらは、同様に明示的に登録されていないかぎり、通知を受信しません。

事前に定義されたシステムイベントは次のとおりです。

- **TableModified** テーブルのローが挿入、更新、または削除されたときにトリガされます。要求の影響を受けるローの数にかかわらず、要求ごとに 1 つの通知が送信されます。object_name パラメータは、モニタするテーブルを指定します。値 "*" は、データベース内のすべてのテーブルを意味します。このイベントには、table_name というパラメータがあり、このパラメータの値は変更されたテーブルの名前です。
- **Commit** コミットが完了した後にトリガされます。このイベントにはパラメータはありません。
- **SyncComplete** 同期が完了した後にトリガされます。このイベントにはパラメータはありません。

ResetLastDownloadTime メソッド

アプリケーションが以前にダウンロードされたデータを再同期するように、パブリケーションの最終ダウンロード時間をリセットします。

構文

```
public virtual bool ResetLastDownloadTime(const char * pubList)
```

パラメータ

- **pubList** リセットするパブリケーションのカンマ区切りのリストを含む文字列。空の文字列は、「非同期」とマーク付けされたものを除くすべてのテーブルを意味します。アスタリスクのみの文字列 ("*") は、すべてのパブリケーションを表します。一部のテーブルは、どのパブリケーションの一部でもないため、この値が "*" の場合は含まれません。

戻り値

成功した場合は true、失敗した場合は false。

備考

次のメソッド呼び出しは、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。

```
conn->ResetLastDownloadTime( "" );
```

RevokeConnectFrom メソッド

Ultra Light データベースからユーザ ID のアクセス権を取り消します。

構文

```
public virtual bool RevokeConnectFrom(const char * uid)
```

パラメータ

- **uid** 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。

戻り値

成功した場合は true、失敗した場合は false。

Rollback メソッド

現在のトランザクションをロールバックします。

構文

```
public virtual bool Rollback()
```

戻り値

成功した場合は true、失敗した場合は false。

RollbackPartialDownload メソッド

失敗した同期からの変更をロールバックします。

構文

```
public virtual bool RollbackPartialDownload()
```

戻り値

成功した場合は **true**、失敗した場合は **false**。

備考

再開可能なダウンロードを使用 (**Keep Partial Download** オプションを有効にして同期) しているときに、同期のダウンロードフェーズ中に通信エラーが発生すると、Ultra Light は、ダウンロードされた変更を保持します (このため、同期は、中断した時点から再開可能です)。ダウンロードを再開しない場合は、このメソッドを使用して、この部分的なダウンロードを破棄します。

このメソッドは、再開可能なダウンロードの使用時にのみ効果があります。

SendNotification メソッド

指定された名前と一致するすべてのキューに通知を送信します。

構文

```
public virtual ul_u_long SendNotification(  
    const char * queueName,  
    const char * eventName,  
    const char * parameters  
)
```

パラメータ

- **queueName** 対象となるキューの名前 (または "*")。
- **eventName** 通知の ID。
- **parameters** オプションのパラメータオプションのリスト。

戻り値

送信済みの通知の数。(一致するキューの数)

備考

これに含まれるのは、現在の接続におけるキューです。この呼び出しはブロックしません。特別なキュー名 "*" を使用して、すべてのキューに送信します。指定されたイベント名は、システム定義またはユーザ定義のイベントと対応する必要はありません。読み込まれたイベント名は、通知を識別するためにそのまま渡され、送信者と受信者に対してしか意味を持たないからです。

parameters 値には、「名前=値」のペアをセミコロンで区切ったオプションリストを指定します。通知が読み込まれた後、パラメータの値が **GetNotificationParameter** メソッドによって読み込まれます。

参照

- [ULConnection.GetNotificationParameter メソッド \[Ultra Light C++\]125 ページ](#)

SetDatabaseOption メソッド

指定されたデータベースオプションを設定します。

構文

```
public virtual bool SetDatabaseOption(  
    const char * optName,  
    const char * value  
)
```

パラメータ

- **optName** 設定されるオプションの名前。
- **value** オプションの新しい値。

戻り値

成功した場合は true、失敗した場合は false。

参照

- [「Ultra Light データベースオプション」『Ultra Light データベース管理とリファレンス』](#)

SetDatabaseOptionInt メソッド

データベースオプションを設定します。

構文

```
public virtual bool SetDatabaseOptionInt(  
    const char * optName,  
    ul_u_long value  
)
```

パラメータ

- **optName** 設定されるオプションの名前。
- **value** オプションの新しい値。

戻り値

成功した場合は true、失敗した場合は false。

SetSynchronizationCallback メソッド

同期の実行中に呼び出されるようコールバックを設定します。

構文

```
public virtual void SetSynchronizationCallback(  
    ul_sync_observer_fn callback,  
    void * userData  
)
```

パラメータ

- **callback** ul_sync_observer_fn コールバック。
- **userData** コールバックに渡されるユーザコンテキスト情報。

SetSyncInfo メソッド

指定された ul_sync_info 構造体に基づいて、指定された名前を使用して同期プロファイルを作成します。

構文

```
public virtual bool SetSyncInfo(  
    char const * profileName,  
    ul_sync_info * info  
)
```

パラメータ

- **profileName** 同期プロファイルの名前。
- **info** 同期パラメータを保持する ul_sync_info 構造体へのポインタ。

戻り値

成功した場合は true、失敗した場合は false。

備考

同じ名前の同期プロファイルがすでにある場合は、この同期プロファイルで置き換えられます。構造体に NULL ポインタを指定することによって、指定されたプロファイルが削除されます。

SetUserPointer メソッド

呼び出すアプリケーションによって接続に使用される任意のポインタ値を設定します。

構文

```
public virtual void * SetUserPointer(void * ptr)
```

戻り値

前に設定されていたポインタ値。

備考

これを使用して、アプリケーションデータを接続に関連付けることができます。

StartSynchronizationDelete メソッド

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

構文

```
public virtual bool StartSynchronizationDelete()
```

戻り値

成功した場合は true、失敗した場合は false。

StopSynchronizationDelete メソッド

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

構文

```
public virtual bool StopSynchronizationDelete()
```

戻り値

成功した場合は true、失敗した場合は false。

Synchronize メソッド

Ultra Light アプリケーションで同期を開始します。

構文

```
public virtual bool Synchronize(ul_sync_info * info)
```

パラメータ

- **info** 同期パラメータを保持する ul_sync_info 構造体へのポインタ。

戻り値

成功した場合は true、失敗した場合は false。

備考

このメソッドで、Mobile Link サーバとの同期を開始します。このメソッドは、同期が完了するまで戻りませんが、同期中に、別の接続を使用する追加スレッドがデータベースにアクセスし続ける場合があります。

このメソッドを呼び出す前に、ULDatabaseManager クラスの各メソッドで使用しているプロトコルと暗号化を有効にしてください。たとえば、"HTTP" を使用している場合は、ULDatabaseManager.EnableHttpSynchronization メソッドを呼び出します。

```
ul_sync_info info;  
conn->InitSyncInfo( &info );  
info.user_name = "my_user";  
info.version = "myapp_1_2";  
info.stream = "HTTP";
```

```
info.stream_parms = "host=myserver.com";  
conn->Synchronize( &info );
```

参照

- [ULDatabaseManager.EnableHttpSynchronization](#) メソッド [Ultra Light C++]142 ページ
- 『[Mobile Link クライアントネットワークプロトコルオプション](#)』『[Mobile Link クライアント管理](#)』

SynchronizeFromProfile メソッド

指定されたプロファイルとマージパラメータを使用して、データベースを同期します。

構文

```
public virtual bool SynchronizeFromProfile(  
    const char * profileName,  
    const char * mergeParms,  
    ul_sync_observer_fn observer,  
    void * userData  
)
```

パラメータ

- **profileName** 同期するプロファイルの名前。
- **mergeParms** 同期で使用するマージパラメータ。
- **observer** ステータス更新の送信先となる observer コールバック。
- **userData** コールバックに渡されるユーザコンテキストデータ。

戻り値

成功した場合は true、失敗した場合は false。

備考

このメソッドは、SYNCHRONIZE 文を実行するのと同じです。

参照

- [ULConnection.Synchronize](#) メソッド [Ultra Light C++]134 ページ
- 『[SYNCHRONIZE 文](#) [Ultra Light]』『[Ultra Light データベース管理とリファレンス](#)』

TriggerEvent メソッド

ユーザ定義のイベントをトリガして、登録されたすべてのキューに通知を送信します。

構文

```
public virtual ul_u_long TriggerEvent(  
    const char * eventName,  
    const char * parameters  
)
```

パラメータ

- **eventName** トリガするシステム定義またはユーザ定義のイベントの名前。
- **parameters** オプションのパラメータオプションのリスト。

戻り値

送信済みのイベント通知の数。

備考

parameters 値には、「名前=値」のペアをセミコロンで区切ったオプションリストを指定します。通知が読み込まれた後、パラメータの値が `GetNotificationParameter()` によって読み込まれます。

参照

- [ULConnection.GetNotificationParameter メソッド \[Ultra Light C++\]125 ページ](#)

ValidateDatabase メソッド

この接続のデータベースを検証します。

構文

```
public virtual bool ValidateDatabase (  
    ul_u_short flags,  
    ul_validate_callback_fn fn,  
    void * user_data,  
    const char * tableName  
)
```

パラメータ

- **flags** 検証のタイプを制御するフラグ。下の例を参照してください。
- **fn** 検証の進行状況の情報を受け取る関数。
- **user_data** コールバックにより呼び出し元に送り返すユーザデータ。
- **tableName** 省略可。検証する特定のテーブル。

戻り値

成功した場合は `true`、失敗した場合は `false`。

備考

このルーチンに渡されるフラグに応じて、テーブル、インデックス、およびデータベースページを検証できます。検証中に情報を受け取るには、コールバック関数を実装し、アドレスをこのルーチンに渡します。検証対象を特定のテーブルに限定するには、テーブルの名前または ID を最後のパラメータとして渡します。

`flags` パラメータは、次のいずれかの値の組み合わせです。

- ULVF_TABLE
- ULVF_INDEX
- ULVF_DATABASE
- ULVF_EXPRESS
- ULVF_FULL_VALIDATE

参照

- [ULVF_TABLE 変数 \[Ultra Light C および Embedded SQL データタイプ\]108 ページ](#)
- [ULVF_INDEX 変数 \[Ultra Light C および Embedded SQL データタイプ\]108 ページ](#)
- [ULVF_DATABASE 変数 \[Ultra Light C および Embedded SQL データタイプ\]106 ページ](#)
- [ULVF_EXPRESS 変数 \[Ultra Light C および Embedded SQL データタイプ\]106 ページ](#)
- [ULVF_FULL_VALIDATE 変数 \[Ultra Light C および Embedded SQL データタイプ\]107 ページ](#)

例

The following example demonstrates table and index validation in express mode:
flags = ULVF_TABLE | ULVF_INDEX | ULVF_EXPRESS;

ULDatabaseManager クラス

接続とデータベースを管理します。

構文

```
public class ULDatabaseManager
```

メンバー

継承されたメンバーを含む ULDatabaseManager クラスのすべてのメンバー。

名前	説明
CreateDatabase メソッド	新しいデータベースを作成します。
DropDatabase メソッド	現在実行していない既存のデータベースを消去します。
EnableAesDBEncryption メソッド	AES データベース暗号化を有効にします。
EnableAesFipsDBEncryption メソッド	FIPS 140-2 認定 AES データベース暗号化を有効にします。
EnableAllSynchronization メソッド	TCPIP、HTTP、TLS、HTTPS の 4 種類すべての同期を有効にします。
EnableHttpsSynchronization メソッド	HTTPS 同期を有効にします。

名前	説明
EnableHttpSynchronization メソッド	HTTP 同期を有効にします。
EnableRsaE2ee メソッド	RSA エンドツーエンド暗号化を有効にします。
EnableRsaFipsE2ee メソッド	FIPS 140-2 認定 RSA エンドツーエンド暗号化を有効にします。
EnableRsaFipsSyncEncryption メソッド	SSL ストリームまたは TLS ストリームの FIPS 140-2 認定 RSA 同期暗号化を有効にします。
EnableRsaSyncEncryption メソッド	RSA 同期暗号化を有効にします。
EnableTcpipSynchronization メソッド	TCP/IP 同期を有効にします。
EnableTlsSynchronization メソッド	TLS 同期を有効にします。
EnableZlibSyncCompression メソッド	同期ストリームの ZLIB 圧縮を有効にします。
Fini メソッド	Ultra Light ランタイムをファイナライズします。
Init メソッド	Ultra Light ランタイムを初期化します。
OpenConnection メソッド	既存のデータベースへの新しい接続を開きます。
SetErrorCallback メソッド	エラーの発生時に呼び出されるようコールバックを設定します。
ValidateDatabase メソッド	データベースで低レベルのインデックス検証を実行します。

備考

スレッド対応環境で [Init](#) メソッドを呼び出してから、他の呼び出しを行う必要があります。終了したら、同様にスレッド対応環境で [Fini](#) メソッドを呼び出してください。

注意

このクラスは静的です。このクラスのインスタンスを作成しないでください。

CreateDatabase メソッド

新しいデータベースを作成します。

構文

```
public static ULConnection * CreateDatabase (
    const char * connParms,
    const char * createParms,
    ULError * error
)
```

パラメータ

- **connParms** セミコロンで区切った接続パラメータ文字列で、キーワード=値のペアで設定されます。接続文字列には、データベースの名前を含める必要があります。ここに含まれるパラメータは、データベースの接続時に指定されるパラメータセットと同じです。
- **createParms** データベース作成パラメータをセミコロンで区切った文字列。キーワードと値のペアとして設定されます。例：page_size=2048;obfuscate=yes
- **error** エラー情報を受信するためのオプションの ULError オブジェクト。

戻り値

データベースが正常に作成された場合は、新しいデータベースへの ULConnection オブジェクトが返されます。メソッドが失敗した場合は、NULL が返されます。通常、失敗の原因は、無効なファイル名やアクセスの拒否です。

備考

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

connParms パラメータは、ファイル名や暗号化キーなど、データベースへのアクセス時に必ず適用される一連の標準接続パラメータです。

createParms パラメータは、チェックサムレベル、ページサイズ、照合、時刻と日付の形式など、データベースの作成時にのみ意味を持つ一連のパラメータです。

次のコードは、CreateDatabase メソッドを使用して、ファイル *mydb.udb* に Ultra Light データベースを作成する方法を示します。

```
ULConnection * conn;
conn = ULDatabaseManager::CreateDatabase( "DBF=mydb.udb", "checksum_level=2" );
if( conn != NULL ){
    // success
} else {
    // unable to create
}
```

参照

- 「Ultra Light 接続パラメータ」『Ultra Light データベース管理とリファレンス』
- 「Ultra Light 作成パラメータ」『Ultra Light データベース管理とリファレンス』

DropDatabase メソッド

現在実行していない既存のデータベースを消去します。

構文

```
public static bool DropDatabase(const char * parms, ULError * error)
```

パラメータ

- **parms** データベース識別パラメータ。(接続文字列)
- **error** エラー情報を受信するためのオプションの ULError オブジェクト。

戻り値

データベースが正常に削除された場合は **true**、正常に削除されなかった場合は **false**。

EnableAesDBEncryption メソッド

AES データベース暗号化を有効にします。

構文

```
public static void EnableAesDBEncryption()
```

備考

このメソッドを呼び出して、AES データベース暗号化を使用します。DBKEY 接続パラメータを使用して、暗号化パスフレーズを指定します。このメソッドを呼び出してからデータベース接続を開くようにしてください。

参照

- [「Ultra Light DBKEY 接続パラメータ」『Ultra Light データベース管理とリファレンス』](#)

EnableAesFipsDBEncryption メソッド

FIPS 140-2 認定 AES データベース暗号化を有効にします。

構文

```
public static void EnableAesFipsDBEncryption()
```

備考

このメソッドを呼び出して、FIPS AES データベース暗号化を使用します。DBKEY 接続パラメータを使用して、暗号化パスフレーズを指定します。

データベース作成パラメータ文字列には、'fips=yes' を指定する必要があります。このメソッドを呼び出してからデータベース接続を開くようにしてください。

注意

別途ライセンスが必要な必須コンポーネント

FIPS 認定の暗号化には別のライセンスが必要です。強力な暗号化テクノロジーはすべて、輸出規制対象品目です。

「別途ライセンスが必要なコンポーネント」『SQL Anywhere 16 紹介』を参照してください。

参照

- [ULDatabaseManager.EnableAesDBEncryption](#) メソッド [Ultra Light C++]140 ページ
- 「Ultra Light DBKEY 接続パラメータ」『Ultra Light データベース管理とリファレンス』

EnableAllSynchronization メソッド

TCPIP、HTTP、TLS、HTTPS の 4 種類すべての同期を有効にします。

構文

```
public static void EnableAllSynchronization()
```

備考

このメソッドを呼び出してから `Synchronize` メソッドを呼び出すようにしてください。

同期を開始するときは、**stream** パラメータを "TCPIP"、"HTTP"、"TLS"、または "HTTPS" に設定します。TLS または HTTPS を使用する場合は、ネットワークプロトコル証明書のオプションも設定します。

参照

- 「Mobile Link クライアントネットワークプロトコルオプション」『Mobile Link クライアント管理』

EnableHttpsSynchronization メソッド

HTTPS 同期を有効にします。

構文

```
public static void EnableHttpsSynchronization()
```

備考

このメソッドを呼び出してから `Synchronize` メソッドを呼び出すようにしてください。

同期を開始するときは、**stream** パラメータを "HTTPS" に設定します。また、ネットワークプロトコル証明書オプションも設定します。

参照

- [「Mobile Link クライアントネットワークプロトコルオプション」](#)『[Mobile Link クライアント管理](#)』

EnableHttpSynchronization メソッド

HTTP 同期を有効にします。

構文

```
public static void EnableHttpSynchronization()
```

備考

このメソッドを呼び出してから `Synchronize` メソッドを呼び出すようにしてください。

同期を開始するときは、`stream` パラメータを "HTTP" に設定します。

参照

- [「Mobile Link クライアントネットワークプロトコルオプション」](#)『[Mobile Link クライアント管理](#)』

EnableRsaE2ee メソッド

RSA エンドツーエンド暗号化を有効にします。

構文

```
public static void EnableRsaE2ee()
```

備考

このメソッドを呼び出してから `Synchronize` メソッドを呼び出すようにしてください。

エンドツーエンド暗号化を使用するには、`e2ee_public_key` ネットワークプロトコルオプションを設定します。

参照

- [「Mobile Link クライアントネットワークプロトコルオプション」](#)『[Mobile Link クライアント管理](#)』

EnableRsaFipsE2ee メソッド

FIPS 140-2 認定 RSA エンドツーエンド暗号化を有効にします。

構文

```
public static void EnableRsaFipsE2ee()
```

備考

このメソッドを呼び出してから `Synchronize` メソッドを呼び出すようにしてください。

エンドツーエンド暗号化を使用するには、`e2ee_public_key` ネットワークプロトコルオプションを設定します。この場合は、`fips` オプションを "yes" に設定する必要があります。

参照

- [「Mobile Link クライアントネットワークプロトコルオプション」『Mobile Link クライアント管理』](#)

EnableRsaFipsSyncEncryption メソッド

SSL ストリームまたは TLS ストリームの FIPS 140-2 認定 RSA 同期暗号化を有効にします。

構文

```
public static void EnableRsaFipsSyncEncryption()
```

備考

このメソッドを呼び出してから `Synchronize` メソッドを呼び出すようにしてください。

これは、FIPS RSA 暗号化に対して `stream` パラメータを "TLS" または "HTTPS" に設定するときが必要です。この場合は、`fips` オプションを "yes" に設定する必要があります。

参照

- [ULDatabaseManager.EnableRsaSyncEncryption メソッド \[Ultra Light C++\]143 ページ](#)
- [「Mobile Link クライアントネットワークプロトコルオプション」『Mobile Link クライアント管理』](#)

EnableRsaSyncEncryption メソッド

RSA 同期暗号化を有効にします。

構文

```
public static void EnableRsaSyncEncryption()
```

備考

このメソッドを呼び出してから `Synchronize` メソッドを呼び出すようにしてください。

これは、RSA 暗号化に対して `stream` パラメータを "TLS" または "HTTPS" に設定するときが必要です。

参照

- [「Mobile Link クライアントネットワークプロトコルオプション」『Mobile Link クライアント管理』](#)

EnableTcpipSynchronization メソッド

TCP/IP 同期を有効にします。

構文

```
public static void EnableTcpipSynchronization()
```

備考

このメソッドを呼び出してから **Synchronize** メソッドを呼び出すようにしてください。

同期を開始するときは、**stream** パラメータを "TCPIP" に設定します。

参照

- [「Mobile Link クライアントネットワークプロトコルオプション」『Mobile Link クライアント管理』](#)

EnableTlsSynchronization メソッド

TLS 同期を有効にします。

構文

```
public static void EnableTlsSynchronization()
```

備考

このメソッドを呼び出してから **Synchronize** メソッドを呼び出すようにしてください。

同期を開始するときは、**stream** パラメータを "TLS" に設定します。また、ネットワークプロトコル証明書オプションも設定します。

参照

- [「Mobile Link クライアントネットワークプロトコルオプション」『Mobile Link クライアント管理』](#)

EnableZlibSyncCompression メソッド

同期ストリームの ZLIB 圧縮を有効にします。

構文

```
public static void EnableZlibSyncCompression()
```

備考

このメソッドを呼び出してから **Synchronize** メソッドを呼び出すようにしてください。

圧縮を使用するには、**compression** ネットワークプロトコルオプションを "zlib" に設定します。

参照

- 『Mobile Link クライアントネットワークプロトコルオプション』『Mobile Link クライアント管理』

Finis メソッド

Ultra Light ランタイムをファイナライズします。

構文

```
public static void Finis()
```

備考

このメソッドは、アプリケーションの終了時に、単一のスレッドで1度だけ呼び出す必要があります。このメソッドはスレッド対応ではありません。

Init メソッド

Ultra Light ランタイムを初期化します。

構文

```
public static bool Init()
```

戻り値

成功した場合は **true**、失敗した場合は **false**。また、メソッドを複数回呼び出した場合にも、**false** が返されます。

備考

このメソッドは、その他の呼び出しを行う前に、単一のスレッドで1度だけ呼び出す必要があります。このメソッドはスレッド対応ではありません。

通常、メモリが使用可能であるかぎり、このメソッドは失敗しません。

OpenConnection メソッド

既存のデータベースへの新しい接続を開きます。

構文

```
public static ULConnection * OpenConnection(  
    const char * connParms,  
    ULError * error,  
    void * reserved  
)
```

パラメータ

- **connParms** 接続文字列。

- **error** エラー情報を返すためのオプションの `ULError` オブジェクト。
- **reserved** 内部用に予約されています。除外または `NULL` に設定されます。

戻り値

メソッドが成功した場合は、新しい `ULConnection` オブジェクト。成功しなかった場合は `NULL`。

備考

接続文字列は、どのデータベースに接続するかを示す `option=value` 接続パラメータ (セミコロンで区切られた) および接続に使用するオプションのセットです。たとえば、暗号化パスフレーズを安全に取得した後に得られる接続文字列は、`"DBF=mydb.udb;DBKEY=iyntTZld9OEa#&G"` のようになります。

エラー情報を取得するには、`ULError` オブジェクトへのポインタを渡します。次に、可能性のあるエラーのリストを示します。

- **SQLE_INVALID_PARSE_PARAMETER** `connParms` が正しくフォーマットされていません。
- **SQLE_UNRECOGNIZED_OPTION** 接続オプション名のスペルを間違えた可能性があります。
- **SQLE_INVALID_OPTION_VALUE** 接続オプション値が正しく指定されていません。
- **SQLE_ULTRALITE_DATABASE_NOT_FOUND** 指定されたデータベースが見つかりませんでした。
- **SQLE_INVALID_LOGON** 無効なユーザ ID または間違ったパスワードを入力しました。
- **SQLE_TOO_MANY_CONNECTIONS** 同時データベース接続の最大数を超過しました。

参照

- 「Ultra Light 接続文字列とパラメータ」『Ultra Light データベース管理とリファレンス』
- 「Ultra Light 接続パラメータ」『Ultra Light データベース管理とリファレンス』

SetErrorCallback メソッド

エラーの発生時に呼び出されるようコールバックを設定します。

構文

```
public static void SetErrorCallback(  
    ul_cpp_error_callback_fn callback,  
    void * userData  
)
```

パラメータ

- **callback** コールバック関数。
- **userData** コールバックに渡されるユーザコンテキスト情報。

備考

このメソッドはスレッド対応ではありません。

ValidateDatabase メソッド

データベースで低レベルのインデックス検証を実行します。

構文

```
public static bool ValidateDatabase(  
    const char * connParms,  
    ul_u_short flags,  
    ul_validate_callback_fn fn,  
    void * userData,  
    ULError * error  
)
```

パラメータ

- **connParms** データベースへの接続に使用されるパラメータ。
- **flags** 検証のタイプを制御するフラグ。次の例を参照してください。
- **fn** 検証の進行状況の情報を受け取る関数。
- **userData** コールバックにより呼び出し元に送り返すユーザデータ。
- **error** エラー情報を受信するためのオプションの ULError オブジェクト。

戻り値

検証が成功した場合は **true**、そうでない場合は **false**。

備考

flags パラメータは、次のいずれかの値の組み合わせです。

- **ULVF_TABLE**
- **ULVF_INDEX**
- **ULVF_DATABASE**
- **ULVF_EXPRESS**
- **ULVF_FULL_VALIDATE**

参照

- [ULVF_TABLE 変数 \[Ultra Light C および Embedded SQL データタイプ\]108 ページ](#)
- [ULVF_INDEX 変数 \[Ultra Light C および Embedded SQL データタイプ\]108 ページ](#)
- [ULVF_DATABASE 変数 \[Ultra Light C および Embedded SQL データタイプ\]106 ページ](#)
- [ULVF_EXPRESS 変数 \[Ultra Light C および Embedded SQL データタイプ\]106 ページ](#)
- [ULVF_FULL_VALIDATE 変数 \[Ultra Light C および Embedded SQL データタイプ\]107 ページ](#)

例

次の例は、エクスプレスモードでのテーブルとインデックスの検証を示します。

```
flags = ULVF_TABLE | ULVF_INDEX | ULVF_EXPRESS;
```

ULDatabaseSchema クラス

Ultra Light データベースのスキーマを表します。

構文

```
public class ULDatabaseSchema
```

メンバー

継承されたメンバーを含む ULDatabaseSchema クラスのすべてのメンバー。

名前	説明
Close メソッド	このオブジェクトを破棄します。
GetConnection メソッド	ULConnection オブジェクトを取得します。
GetNextPublication メソッド	データベース内の次のパブリケーションの名前を取得します。
GetNextTable メソッド	データベース内の次のテーブル (スキーマ) を取得します。
GetPublicationCount メソッド	データベース内のパブリケーション数を取得します。
GetTableCount メソッド	データベース内のテーブルの数を返します。
GetTableSchema メソッド	指定したテーブルのスキーマを返します。

Close メソッド

このオブジェクトを破棄します。

構文

```
public virtual void Close()
```

GetConnection メソッド

ULConnection オブジェクトを取得します。

構文

```
public virtual ULConnection * GetConnection()
```

戻り値

このオブジェクトに関連付けられている ULConnection。

GetNextPublication メソッド

データベース内の次のパブリケーションの名前を取得します。

構文

```
public virtual const char * GetNextPublication(  
    ul_publication_iter * iter  
)
```

パラメータ

- **iter** 繰り返し変数へのポインタ。

戻り値

次のパブリケーションの名前。この値は静的バッファを指します。静的バッファの内容は、それ以降の Ultra Light の呼び出しによって変更される可能性があるため、値を保持したい場合は、そのコピーを作成してください。反復が完了すると NULL が返されます。

備考

最初の呼び出しの前に、iter 値を ul_publication_iter_start 定数に初期化します。

参照

- [ul_publication_iter_start 変数 \[Ultra Light C++\]230 ページ](#)

GetNextTable メソッド

データベース内の次のテーブル (スキーマ) を取得します。

構文

```
public virtual ULTableSchema * GetNextTable(ul_table_iter * iter)
```

パラメータ

- **iter** 繰り返し変数へのポインタ。

戻り値

反復が完了したときに ULTableSchema オブジェクトまたは NULL。

備考

最初の呼び出しの前に、iter 値を ul_table_iter_start 定数に初期化します。

参照

- [ul_table_iter_start 変数 \[Ultra Light C++\]230 ページ](#)

GetPublicationCount メソッド

データベース内のパブリケーション数を取得します。

構文

```
public virtual ul_publication_count GetPublicationCount()
```

戻り値

データベース内のパブリケーションの数です。

備考

パブリケーション ID の範囲は 1 から、このメソッドが返す数字までです。

GetTableCount メソッド

データベース内のテーブルの数を返します。

構文

```
public virtual ul_table_num GetTableCount()
```

戻り値

テーブルの数を表す整数。

GetTableSchema メソッド

指定したテーブルのスキーマを返します。

構文

```
public virtual ULTableSchema * GetTableSchema(const char * tableName)
```

パラメータ

- **tableName** テーブル名。

戻り値

特定のテーブルの場合は ULTableSchema オブジェクト。それ以外で、テーブルが存在しない場合は UL_NULL。

ULError クラス

Ultra Light ランタイムから返されたエラーを管理します。

構文

```
public class ULError
```

メンバー

継承されたメンバーを含む ULError クラスのすべてのメンバー。

名前	説明
ULError コンストラクタ	ULError オブジェクトを構築します。
Clear メソッド	現在のエラーをクリアします。
GetErrorInfo メソッド	基本となる <code>ul_error_info</code> オブジェクトへのポインタを返します。
GetParameter メソッド	指定されたバッファに、指定されたエラーパラメータをコピーします。
GetParameterCount メソッド	エラーパラメータの数を返します。
GetSQLCode メソッド	最後の操作の <code>SQLCODE</code> エラーコードを返します。
GetSQLCount メソッド	最後の操作とその操作の結果によって異なる値を返します。
GetString メソッド	現在のエラーの説明を返します。
GetURL メソッド	このエラーの資料ページの URL を返します。
IsOK メソッド	エラーコードをテストします。

ULError コンストラクタ

ULError オブジェクトを構築します。

構文

```
public ULError()
```

Clear メソッド

現在のエラーをクリアします。

構文

```
public void Clear()
```

備考

現在のエラーは、ほとんどの呼び出しで自動的にクリアされます。そのため、通常、このメソッドがアプリケーションによって呼び出されることはありません。

GetErrorInfo メソッド

基本となる ul_error_info オブジェクトへのポインタを返します。

オーバーロードリスト

名前	説明
GetErrorInfo() メソッド	基本となる ul_error_info オブジェクトへのポインタを返します。
GetErrorInfo() メソッド	基本となる ul_error_info オブジェクトへのポインタを返します。

GetErrorInfo() メソッド

基本となる ul_error_info オブジェクトへのポインタを返します。

構文

```
public const ul_error_info * GetErrorInfo()
```

戻り値

基本となる ul_error_info オブジェクトへのポインタ。

参照

- [ul_error_info 構造体 \[Ultra Light C および Embedded SQL データタイプ\]](#)96 ページ

GetErrorInfo() メソッド

基本となる ul_error_info オブジェクトへのポインタを返します。

構文

```
public ul_error_info * GetErrorInfo()
```

戻り値

基本となる ul_error_info オブジェクトへのポインタ。

参照

- [ul_error_info 構造体 \[Ultra Light C および Embedded SQL データタイプ\]96 ページ](#)

GetParameter メソッド

指定されたバッファに、指定されたエラーパラメータをコピーします。

構文

```
public size_t GetParameter(ul_u_short parmNo, char * dst, size_t len)
```

パラメータ

- **parmNo** 1 から始まるパラメータ番号。
- **dst** パラメータを受け取るバッファ。
- **len** バッファのサイズ。

戻り値

パラメータの格納に必要なサイズ。または、序数が無効の場合は 0。戻り値が len 値より大きい場合、パラメータはトランケートされます。

備考

バッファが小さすぎて文字列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。

GetParameterCount メソッド

エラーパラメータの数を返します。

構文

```
public ul_u_short GetParameterCount()
```

戻り値

エラーパラメータ数。

GetSQLCode メソッド

最後の操作の SQLCODE エラーコードを返します。

構文

```
public an_sql_code GetSQLCode()
```

戻り値

sqlcode 値。

GetSQLCount メソッド

最後の操作とその操作の結果によって異なる値を返します。

構文

```
public ul_s_long GetSQLCount()
```

戻り値

適用される場合は、最後の操作の値。それ以外で、適用されない場合は -1。

備考

次のリストは、考えられる操作と、返される結果の概要を示します。

- **正常に実行された INSERT、UPDATE、または DELETE 操作** 文によって影響を受けたローの数を返します。
- **SQL 文の構文エラー (SQLE_SYNTAX_ERROR)** 文内のおおよそのエラー検出位置を返します。

GetString メソッド

現在のエラーの説明を返します。

構文

```
public size_t GetString(char * dst, size_t len)
```

パラメータ

- **dst** エラーの説明を受信するバッファ。
- **len** バッファのサイズ (配列の要素数)。

戻り値

文字列の格納に必要なサイズ。戻り値が len 値より大きい場合、文字列はトランケートされます。

備考

文字列には、エラーコードとすべてのパラメータが含まれます。ULError.GetURL メソッドで返された URL をロードすると、エラーの詳細な説明が得られます。

バッファが小さすぎて文字列がトランケートされる場合であっても、出力文字列は常に NULL で終了します。

参照

- [ULError.GetURL メソッド \[Ultra Light C++\]155 ページ](#)

GetURL メソッド

このエラーの資料ページの URL を返します。

構文

```
public size_t GetURL(char * buffer, size_t len, const char * reserved)
```

パラメータ

- **buffer** URL を受け取るバッファ。
- **len** バッファのサイズ。
- **reserved** 今後の使用のために予約されているため、デフォルトの NULL を渡す必要があります。

戻り値

URL の格納に必要なサイズ。戻り値が **len** より大きい場合、URL はトランケートされます。

IsOK メソッド

エラーコードをテストします。

構文

```
public bool IsOK()
```

戻り値

現在のコードが `SQLE_NOERROR` か、警告の場合は `true`。それ以外で、現在のコードがエラーを示している場合は、`false`。

ULIndexSchema クラス

Ultra Light テーブルのインデックスのスキーマを表します。

構文

```
public class ULIndexSchema
```

メンバー

継承されたメンバーを含む `ULIndexSchema` クラスのすべてのメンバー。

名前	説明
Close メソッド	このオブジェクトを破棄します。
GetColumnCount メソッド	インデックス内のカラム数を取得します。
GetColumnName メソッド	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。
GetConnection メソッド	ULConnection オブジェクトを取得します。
GetIndexColumnID メソッド	1 から始まるインデックスカラム ID を名前から取得します。
GetIndexFlags メソッド	インデックスプロパティフラグのビットフィールドを取得します。
GetName メソッド	インデックスの名前を取得します。
GetReferencedIndexName メソッド	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。
GetReferencedTableName メソッド	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。
GetTableName メソッド	この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 メソッド	カラムが降順かどうかを調べます。

Close メソッド

このオブジェクトを破棄します。

構文

```
public virtual void Close()
```

GetColumnCount メソッド

インデックス内のカラム数を取得します。

構文

```
public virtual ul_column_num GetColumnCount()
```

戻り値

インデックス内のカラム数。

GetColumnName メソッド

インデックス内のカラムの位置を指定して、カラムの名前を取得します。

構文

```
public virtual const char * GetColumnName(ul_column_num col_id_in_index)
```

パラメータ

- **col_id_in_index** インデックス内のカラムの位置を示す 1 から始まる順序数。

戻り値

カラム名。この値は静的バッファを指します。静的バッファの内容は、それ以降の Ultra Light の呼び出しによって変更される可能性があるため、値を保持したい場合は、そのコピーを作成してください。

GetConnection メソッド

ULConnection オブジェクトを取得します。

構文

```
public virtual ULConnection * GetConnection()
```

戻り値

このオブジェクトに関連付けられている接続。

GetIndexColumnID メソッド

1 から始まるインデックスカラム ID を名前から取得します。

構文

```
public virtual ul_column_num GetIndexColumnID(const char * columnName)
```

パラメータ

- **columnName** カラムの名前。

戻り値

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

GetIndexFlags メソッド

インデックスプロパティフラグのビットフィールドを取得します。

構文

```
public virtual ul_index_flag GetIndexFlags()
```

参照

- [ul_index_flag 列挙体 \[Ultra Light C++\]228 ページ](#)

GetName メソッド

インデックスの名前を取得します。

構文

```
public virtual const char * GetName()
```

戻り値

インデックス名。この値は静的バッファを指します。静的バッファの内容は、それ以降の Ultra Light の呼び出しによって変更される可能性があるため、値を保持したい場合は、そのコピーを作成してください。

GetReferencedIndexName メソッド

関連付けられているプライマリインデックスの名前を取得します。

構文

```
public virtual const char * GetReferencedIndexName()
```

戻り値

参照先インデックスの名前。この値は静的バッファを指します。静的バッファの内容は、それ以降の Ultra Light の呼び出しによって変更される可能性があるため、値を保持したい場合は、そのコピーを作成してください。

備考

このメソッドは、外部キーにのみ適用されます。

GetReferencedTableName メソッド

関連付けられているプライマリテーブルの名前を取得します。

構文

```
public virtual const char * GetReferencedTableName()
```

戻り値

参照先テーブルの名前。この値は静的バッファを指します。静的バッファの内容は、それ以降の Ultra Light の呼び出しによって変更される可能性があるため、値を保持したい場合は、そのコピーを作成してください。

備考

このメソッドは、外部キーにのみ適用されます。

GetTableName メソッド

このインデックスが含まれるテーブルの名前を取得します。

構文

```
public virtual const char * GetTableName()
```

戻り値

このインデックスが含まれるテーブルの名前。この値は静的バッファを指します。静的バッファの内容は、それ以降の Ultra Light の呼び出しによって変更される可能性があるため、値を保持したい場合は、そのコピーを作成してください。

IsColumnDescending メソッド

カラムが降順かどうかを調べます。

構文

```
public virtual bool IsColumnDescending(ul_column_num cid)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。

戻り値

カラムが降順の場合は true、昇順の場合は false。

ULPreparedStatement クラス

準備された SQL 文を表します。

構文

```
public class ULPreparedStatement
```

メンバー

継承されたメンバーを含む ULPreparedStatement クラスのすべてのメンバー。

名前	説明
AppendParameterByteChunk メソッド	複数のチャンクに分解される、サイズの大きい Binary パラメータを設定します。
AppendParameterStringChunk メソッド	複数のチャンクに分解される、サイズの大きい String パラメータを設定します。
Close メソッド	このオブジェクトを破棄します。

名前	説明
ExecuteQuery メソッド	SQL SELECT 文をクエリとして実行します。
ExecuteStatement メソッド	SQL INSERT 文、DELETE 文、UPDATE 文のように、結果セットを返さない文を実行します。
GetConnection メソッド	接続オブジェクトを取得します。
GetParameterCount メソッド	この文の入力パラメータの数を取得します。
GetParameterID メソッド	1 から始まるパラメータ名の順序数を取得します。
GetParameterType メソッド	パラメータの記憶タイプまたはホスト変数の型を取得します。
GetPlan メソッド	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。
GetResultSetSchema メソッド	結果セットのスキーマを取得します。
GetRowsAffectedCount メソッド	最後の文の影響を受けるローの数を取得します。
HasResultSet メソッド	SQL 文に結果セットがあるかどうかを調べます。
SetParameterBinary メソッド	パラメータを <code>ul_binary</code> 値に設定します。
SetParameterDateTime メソッド	パラメータを <code>DECL_DATETIME</code> 値に設定します。
SetParameterDouble メソッド	パラメータを <code>double</code> 値に設定します。
SetParameterFloat メソッド	パラメータを <code>float</code> 値に設定します。
SetParameterGuid メソッド	パラメータを <code>GUID</code> 値に設定します。
SetParameterInt メソッド	パラメータを <code>integer</code> 値に設定します。
SetParameterIntWithType メソッド	パラメータを、指定された整数型の <code>integer</code> 値に設定します。
SetParameterNull メソッド	パラメータを <code>NULL</code> に設定します。
SetParameterString メソッド	パラメータを <code>string</code> 値に設定します。

AppendParameterByteChunk メソッド

複数のチャンクに分解される、サイズの大きい Binary パラメータを設定します。

構文

```
public virtual bool AppendParameterByteChunk (  
    ul_column_num pid,  
    const ul_byte * value,  
    size_t valueSize  
)
```

パラメータ

- **pid** パラメータの、1 から始まる序数。
- **value** 追加するバイトのチャンク。
- **valueSize** バッファのサイズ。

戻り値

成功した場合は true、失敗した場合は false。

AppendParameterStringChunk メソッド

複数のチャンクに分解される、サイズの大きい String パラメータを設定します。

構文

```
public virtual bool AppendParameterStringChunk (  
    ul_column_num pid,  
    const char * value,  
    size_t len  
)
```

パラメータ

- **pid** パラメータの、1 から始まる序数。
- **value** 追加する文字列のチャンク。
- **len** 省略可。文字列のチャンクの長さ (バイト単位)、または文字列のチャンクが NULL で終了する場合は UL_NULL_TERMINATED_STRING に設定されます。

戻り値

成功した場合は true、失敗した場合は false。

Close メソッド

このオブジェクトを破棄します。

構文

```
public virtual void Close()
```

ExecuteQuery メソッド

SQL SELECT 文をクエリとして実行します。

構文

```
public virtual ULResultSet * ExecuteQuery()
```

戻り値

クエリの結果 (ローのセット) を含む ULResultSet オブジェクト。

ExecuteStatement メソッド

SQL INSERT 文、DELETE 文、UPDATE 文のように、結果セットを返さない文を実行します。

構文

```
public virtual bool ExecuteStatement()
```

戻り値

成功した場合は true、失敗した場合は false。

GetConnection メソッド

接続オブジェクトを取得します。

構文

```
public virtual ULConnection * GetConnection()
```

戻り値

この準備文に関連付けられている ULConnection オブジェクト。

GetParameterCount メソッド

この文の入力パラメータの数を取得します。

構文

```
public virtual ul_u_short GetParameterCount()
```

戻り値

この文の入力パラメータの数。

GetParameterID メソッド

1 から始まるパラメータ名の順序数を取得します。

構文

```
public virtual ul_column_num GetParameterID(const char * name)
```

パラメータ

- **name** ホスト変数名。

戻り値

1 から始まるパラメータ名の順序数。

GetParameterType メソッド

パラメータの記憶タイプまたはホスト変数の型を取得します。

構文

```
public virtual ul_column_storage_type GetParameterType(  
    ul_column_num pid  
)
```

パラメータ

- **pid** パラメータの、1 から始まる序数。

戻り値

指定したパラメータのタイプ。

参照

- [ul_column_storage_type 列挙体 \[Ultra Light C および Embedded SQL データタイプ\]90 ページ](#)

GetPlan メソッド

クエリ実行プランのテキストベースの記述を取得します。

構文

```
public virtual size_t GetPlan(char * dst, size_t dstSize)
```

パラメータ

- **dst** プランテキストの宛先のバッファ。NULL を渡し、プランの保持に必要なバッファのサイズを特定します。
- **dstSize** 宛先のバッファのサイズ。

戻り値

バッファにコピーされるバイト数。それ以外で、dst 値が NULL の場合は、プランの格納に必要なバイト数 (NULL ターミネータを含まない)。

備考

このメソッドは、主に開発中の使用を目的とします。

プランがない場合は、空の文字列を返します。準備された文が SQL クエリの場合には、プランが存在します。

関連するクエリの実行前にプランが取得された場合は、クエリの実行に使用される操作がプランに表示されます。また、クエリの実行後にプランが取得された場合は、各操作で生成されるロー数也表示されます。このプランを使用して、クエリの実行に関する理解を深めることができます。

GetResultSetSchema メソッド

結果セットのスキーマを取得します。

構文

```
public virtual const ULResultSetSchema & GetResultSetSchema()
```

戻り値

結果セットのスキーマに関する情報を取得するために使用できる ULResultSetSchema オブジェクト。

GetRowsAffectedCount メソッド

最後の文の影響を受けるローの数を取得します。

構文

```
public virtual ul_s_long GetRowsAffectedCount()
```

戻り値

最後の文の影響を受けるローの数。ローの数を使用できない場合 (たとえば、文によってデータではなくスキーマが変更される場合)、戻り値は -1 になります。

HasResultSet メソッド

SQL 文に結果セットがあるかどうかを調べます。

構文

```
public virtual bool HasResultSet()
```

戻り値

この文が実行されたときに結果セットが生成される場合は **true**。それ以外で、結果セットが生成されない場合は **false**。

SetParameterBinary メソッド

パラメータを **ul_binary** 値に設定します。

構文

```
public virtual bool SetParameterBinary(  
    ul_column_num pid,  
    const p_ul_binary value  
)
```

パラメータ

- **pid** パラメータの、1 から始まる序数。
- **value** **ul_binary** 値。

戻り値

成功した場合は **true**、失敗した場合は **false**。

SetParameterDateTime メソッド

パラメータを **DECL_DATETIME** 値に設定します。

構文

```
public virtual bool SetParameterDateTime(  
    ul_column_num pid,  
    DECL_DATETIME * value  
)
```

パラメータ

- **pid** パラメータの、1 から始まる序数。
- **value** **DECL_DATETIME** 値。

戻り値

成功した場合は **true**、失敗した場合は **false**。

SetParameterDouble メソッド

パラメータを **double** 値に設定します。

構文

```
public virtual bool SetParameterDouble(  
    ul_column_num pid,  
    ul_double value  
)
```

パラメータ

- **pid** パラメータの、1 から始まる序数。
- **value** double 値。

戻り値

成功した場合は true、失敗した場合は false。

SetParameterFloat メソッド

パラメータを float 値に設定します。

構文

```
public virtual bool SetParameterFloat(ul_column_num pid, ul_real value)
```

パラメータ

- **pid** パラメータの、1 から始まる序数。
- **value** float 値。

戻り値

成功した場合は true、失敗した場合は false。

SetParameterGuid メソッド

パラメータを GUID 値に設定します。

構文

```
public virtual bool SetParameterGuid(ul_column_num pid, GUID * value)
```

パラメータ

- **pid** パラメータの、1 から始まる序数。
- **value** GUID 値。

戻り値

成功した場合は true、失敗した場合は false。

SetParameterInt メソッド

パラメータを integer 値に設定します。

構文

```
public virtual bool SetParameterInt(ul_column_num pid, ul_s_long value)
```

パラメータ

- **pid** パラメータの、1 から始まる序数。
- **value** integer 値。

戻り値

成功した場合は true、失敗した場合は false。

SetParameterIntWithType メソッド

パラメータを、指定された整数型の integer 値に設定します。

構文

```
public virtual bool SetParameterIntWithType(  
    ul_column_num pid,  
    ul_s_big value,  
    ul_column_storage_type type  
)
```

パラメータ

- **pid** パラメータの、1 から始まる序数。
- **value** integer 値。
- **type** この値を処理するときの整数型。

戻り値

成功した場合は true、失敗した場合は false。

備考

次に、value パラメータに使用できる integer 値のリストを示します。

- **UL_TYPE_BIT**
- **UL_TYPE_TINY**
- **UL_TYPE_S_SHORT**
- **UL_TYPE_U_SHORT**
- **UL_TYPE_S_LONG**

- `UL_TYPE_U_LONG`

- `UL_TYPE_S_BIG`

- `UL_TYPE_U_BIG`

参照

- [ul_column_storage_type 列挙体 \[Ultra Light C および Embedded SQL データタイプ\]](#)90 ページ

SetParameterNull メソッド

パラメータを NULL に設定します。

構文

```
public virtual bool SetParameterNull(ul_column_num pid)
```

パラメータ

- **pid** パラメータの、1 から始まる序数。

戻り値

成功した場合は `true`、失敗した場合は `false`。

SetParameterString メソッド

パラメータを `string` 値に設定します。

構文

```
public virtual bool SetParameterString(  
    ul_column_num pid,  
    const char * value,  
    size_t len  
)
```

パラメータ

- **pid** パラメータの、1 から始まる序数。

- **value** `string` 値。

- **len** 省略可。文字列の長さ (バイト単位)、または文字列が NULL で終了する場合は `UL_NULL_TERMINATED_STRING` に設定されます。このパラメータが 32K を超える場合は、`SQLE_INVALID_PARAMETER` に設定されます。サイズの大きい文字列の場合は、代わりに `AppendParameterStringChunk` メソッドが呼び出されます。

戻り値

成功した場合は `true`、失敗した場合は `false`。

参照

- [ULPreparedStatement.AppendParameterStringChunk メソッド \[Ultra Light C++\]161 ページ](#)

ULResultSet クラス

Ultra Light データベースの結果セットを表します。

構文

```
public class ULResultSet
```

派生クラス

- [ULTable クラス \[Ultra Light C++\]210 ページ](#)

メンバー

継承されたメンバーを含む ULResultSet クラスのすべてのメンバー。

名前	説明
AfterLast メソッド	カーソルを最後のローの後に移動します。
AppendByteChunk メソッド	カラムにバイトを追加します。
AppendStringChunk メソッド	文字列のチャンクをカラムに追加します。
BeforeFirst メソッド	カーソルを最初のローの前に移動します。
Close メソッド	このオブジェクトを破棄します。
Delete メソッド	現在のローを削除し、次の有効なローに移動します。
DeleteNamed メソッド	現在のローを削除し、次の有効なローに移動します。
First メソッド	カーソルを最初のローに移動します。
GetBinary メソッド	カラムから値を <code>ul_binary</code> 値としてフェッチします。
GetBinaryLength メソッド	カラムの値のバイナリ長を取得します。
GetByteChunk メソッド	カラムからバイナリのチャンクを取得します。
GetConnection メソッド	接続オブジェクトを取得します。

名前	説明
GetDateTime メソッド	カラムから値を DECL_DATETIME としてフェッチします。
GetDouble メソッド	カラムから値を double としてフェッチします。
GetFloat メソッド	カラムから値を float としてフェッチします。
GetGuid メソッド	カラムから値を GUID としてフェッチします。
GetInt メソッド	カラムから値を integer としてフェッチします。
GetIntWithType メソッド	カラムからの値を、指定された整数型としてフェッチします。
GetResultSetSchema メソッド	結果セットに関する情報を取得するために使用できるオブジェクトを返します。
GetRowCount メソッド	テーブルのローの数を取得します。
GetState メソッド	カーソルの内部ステータスを取得します。
GetString メソッド	カラムから値を NULL で終了する文字列としてフェッチします。
GetStringChunk メソッド	カラムから文字列のチャンクを取得します。
GetStringLength メソッド	カラムの値の文字列長を取得します。
IsNull メソッド	カラムが NULL かどうかをチェックします。
Last メソッド	カーソルを最後のローに移動します。
Next メソッド	カーソルをロー 1 つ分進めます。
Previous メソッド	カーソルをロー 1 つ分戻します。
Relative メソッド	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。
SetBinary メソッド	カラムを ul_binary 値に設定します。
SetDateTime メソッド	カラムを DECL_DATETIME 値に設定します。
SetDefault メソッド	カラムをそのデフォルト値に設定します。

名前	説明
SetDouble メソッド	カラムを <code>double</code> 値に設定します。
SetFloat メソッド	カラムを <code>float</code> 値に設定します。
SetGuid メソッド	カラムを <code>GUID</code> 値に設定します。
SetInt メソッド	カラムを <code>integer</code> 値に設定します。
SetIntWithType メソッド	カラムを、指定された整数型の <code>integer</code> 値に設定します。
SetNull メソッド	カラムを <code>NULL</code> に設定します。
SetString メソッド	カラムを <code>string</code> 値に設定します。
Update メソッド	現在の行を更新します。
UpdateBegin メソッド	カラムの設定に使用される更新モードを選択します。

AfterLast メソッド

カーソルを最後のローの後に移動します。

構文

```
public virtual bool AfterLast()
```

戻り値

成功した場合は `true`、失敗した場合は `false`。

AppendByteChunk メソッド

カラムにバイトを追加します。

オーバーロードリスト

名前	説明
AppendByteChunk(const char *, const ul_byte *, size_t) メソッド	カラムにバイトを追加します。
AppendByteChunk(ul_column_num, const ul_byte *, size_t) メソッド	カラムにバイトを追加します。

AppendByteChunk(const char *, const ul_byte *, size_t) メソッド

カラムにバイトを追加します。

構文

```
public virtual bool AppendByteChunk (  
    const char * cname,  
    const ul_byte * value,  
    size_t valueSize  
)
```

パラメータ

- **cname** カラム名。
- **value** 追加するバイトのチャンク。
- **valueSize** バイトのチャンクのサイズ (バイト単位)。

戻り値

成功した場合は true、失敗した場合は false。

備考

AppendBinaryChunk メソッド呼び出しにより現時点で書き込み済みのカラムの末尾に、指定のバイトが追加されます。

参照

- [ULResultSet.AppendByteChunk メソッド \[Ultra Light C++\]171 ページ](#)

AppendByteChunk(ul_column_num, const ul_byte *, size_t) メソッド

カラムにバイトを追加します。

構文

```
public virtual bool AppendByteChunk (  
    ul_column_num cid,  
    const ul_byte * value,  
    size_t valueSize  
)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。
- **value** 追加するバイトのチャンク。
- **valueSize** バイトのチャンクのサイズ (バイト単位)。

戻り値

成功した場合は true、失敗した場合は false。

備考

AppendBinaryChunk メソッド呼び出しにより現時点で書き込み済みのカラムの末尾に、指定のバイトが追加されます。

参照

- [ULResultSet.AppendByteChunk メソッド \[Ultra Light C++\]171 ページ](#)

AppendStringChunk メソッド

文字列のチャンクをカラムに追加します。

オーバーロードリスト

名前	説明
AppendStringChunk(const char *, const char *, size_t) メソッド	文字列のチャンクをカラムに追加します。
AppendStringChunk(ul_column_num, const char *, size_t) メソッド	文字列のチャンクをカラムに追加します。

AppendStringChunk(const char *, const char *, size_t) メソッド

文字列のチャンクをカラムに追加します。

構文

```
public virtual bool AppendStringChunk(  
    const char * cname,  
    const char * value,  
    size_t len  
)
```

パラメータ

- **cname** カラムの名前。
- **value** 追加する文字列のチャンク。
- **len** 省略可。文字列のチャンクの長さ (バイト単位)、または文字列が NULL で終了する場合は UL_NULL_TERMINATED_STRING 定数。

戻り値

成功した場合は true、失敗した場合は false。

備考

このメソッドは、AppendStringChunk メソッド呼び出しにより現時点で書き込み済みの文字列の末尾に、指定の文字列を追加します。

参照

- [ULResultSet.AppendStringChunk メソッド \[Ultra Light C++\]173 ページ](#)

AppendStringChunk(ul_column_num, const char *, size_t) メソッド

文字列のチャンクをカラムに追加します。

構文

```
public virtual bool AppendStringChunk(  
    ul_column_num cid,  
    const char * value,  
    size_t len  
)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。
- **value** 追加する文字列のチャンク。
- **len** 省略可。文字列のチャンクの長さ (バイト単位)、または文字列が NULL で終了する場合は UL_NULL_TERMINATED_STRING 定数。

戻り値

成功した場合は true、失敗した場合は false。

備考

このメソッドは、AppendStringChunk メソッド呼び出しにより現時点で書き込み済みの文字列の末尾に、指定の文字列を追加します。

参照

- [ULResultSet.AppendStringChunk メソッド \[Ultra Light C++\]173 ページ](#)

BeforeFirst メソッド

カーソルを最初のローの前に移動します。

構文

```
public virtual bool BeforeFirst()
```

戻り値

成功した場合は true、失敗した場合は false。

Close メソッド

このオブジェクトを破棄します。

構文

```
public virtual void Close()
```

Delete メソッド

現在のローを削除し、次の有効なローに移動します。

構文

```
public virtual bool Delete()
```

戻り値

成功した場合は `true`、失敗した場合は `false`。

DeleteNamed メソッド

現在のローを削除し、次の有効なローに移動します。

構文

```
public virtual bool DeleteNamed(const char * tableName)
```

パラメータ

- **tableName** テーブル名またはその相関 (同じテーブル名を共有する複数のカラムがデータベースに存在する場合に必要)。

戻り値

成功した場合は `true`、失敗した場合は `false`。

First メソッド

カーソルを最初のローに移動します。

構文

```
public virtual bool First()
```

戻り値

成功した場合は `true`、失敗した場合は `false`。

GetBinary メソッド

カラムから値を `ul_binary` 値としてフェッチします。

オーバロードリスト

名前	説明
GetBinary(const char *, p_ul_binary, size_t) メソッド	カラムから値を ul_binary 値としてフェッチします。
GetBinary(ul_column_num, p_ul_binary, size_t) メソッド	カラムから値を ul_binary 値としてフェッチします。

GetBinary(const char *, p_ul_binary, size_t) メソッド

カラムから値を ul_binary 値としてフェッチします。

構文

```
public virtual bool GetBinary(
    const char * cname,
    p_ul_binary dst,
    size_t len
)
```

パラメータ

- **cname** カラムの名前。
- **dst** ul_binary の結果。
- **len** ul_binary オブジェクトのサイズ。

戻り値

値が正常にフェッチされた場合は true。

GetBinary(ul_column_num, p_ul_binary, size_t) メソッド

カラムから値を ul_binary 値としてフェッチします。

構文

```
public virtual bool GetBinary(
    ul_column_num cid,
    p_ul_binary dst,
    size_t len
)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。
- **dst** ul_binary の結果。
- **len** ul_binary オブジェクトのサイズ。

戻り値

値が正常にフェッチされた場合は `true`。

GetBinaryLength メソッド

カラムの値のバイナリ長を取得します。

オーバーロードリスト

名前	説明
GetBinaryLength(const char *) メソッド	カラムの値のバイナリ長を取得します。
GetBinaryLength(ul_column_num) メソッド	カラムの値のバイナリ長を取得します。

GetBinaryLength(const char *) メソッド

カラムの値のバイナリ長を取得します。

構文

```
public virtual size_t GetBinaryLength(const char * cname)
```

パラメータ

- **cname** カラム名。

戻り値

binary としてのカラムの値のサイズ。

GetBinaryLength(ul_column_num) メソッド

カラムの値のバイナリ長を取得します。

構文

```
public virtual size_t GetBinaryLength(ul_column_num cid)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。

戻り値

binary としてのカラムの値のサイズ。

GetByteChunk メソッド

カラムからバイナリのチャンクを取得します。

オーバロードリスト

名前	説明
GetByteChunk(const char *, ul_byte *, size_t, size_t) メソッド	カラムからバイナリのチャンクを取得します。
GetByteChunk(ul_column_num, ul_byte *, size_t, size_t) メソッド	カラムからバイナリのチャンクを取得します。

GetByteChunk(const char *, ul_byte *, size_t, size_t) メソッド

カラムからバイナリのチャンクを取得します。

構文

```
public virtual size_t GetByteChunk(  
    const char * cname,  
    ul_byte * dst,  
    size_t len,  
    size_t offset  
)
```

パラメータ

- **cname** カラムの名前。
- **dst** バイトを保持するバッファ。
- **len** バッファのサイズ (バイト単位)。
- **offset** 値内でのオフセット (読み込み開始位置)、または前回の読み込みが終了したところから続行する場合は `UL_BLOB_CONTINUE` 定数。

戻り値

宛先のバッファにコピーされたバイト数。dst 値が NULL の場合は、残りのバイト数が返されます。カラムが NULL のときは、dst パラメータに空の文字列が返されます。IsNull メソッドを使用して、NULL と空の文字列を区別してください。

備考

0 が返される場合は、値の最後に到達しています。

参照

- [ULResultSet.IsNull メソッド \[Ultra Light C++\]192 ページ](#)

GetByteChunk(ul_column_num, ul_byte *, size_t, size_t) メソッド

カラムからバイナリのチャンクを取得します。

構文

```
public virtual size_t GetByteChunk (
    ul_column_num cid,
    ul_byte *dst,
    size_t len,
    size_t offset
)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。
- **dst** バイトを保持するバッファ。
- **len** バッファのサイズ (バイト単位)。
- **offset** 値内でのオフセット (読み込み開始位置)、または前回の読み込みが終了したところから続行する場合は UL_BLOB_CONTINUE 定数。

戻り値

宛先のバッファにコピーされたバイト数。dst 値が NULL の場合は、残りのバイト数が返されます。カラムが NULL のときは、dst パラメータに空の文字列が返されます。IsNull メソッドを使用して、NULL と空の文字列を区別してください。

備考

0 が返される場合は、値の最後に到達しています。

参照

- [ULResultSet.IsNull メソッド \[Ultra Light C++\]192 ページ](#)

GetConnection メソッド

接続オブジェクトを取得します。

構文

```
public virtual ULConnection * GetConnection ()
```

戻り値

この結果セットに関連付けられている ULConnection オブジェクト。

GetDateTime メソッド

カラムから値を DECL_DATETIME としてフェッチします。

オーバードリスト

名前	説明
GetDateTime(const char *, DECL_DATETIME *) メソッド	カラムから値を DECL_DATETIME としてフェッチします。
GetDateTime(ul_column_num, DECL_DATETIME *) メソッド	カラムから値を DECL_DATETIME としてフェッチします。

GetDateTime(const char *, DECL_DATETIME *) メソッド

カラムから値を DECL_DATETIME としてフェッチします。

構文

```
public virtual bool GetDateTime(const char * cname, DECL_DATETIME * dst)
```

パラメータ

- **cname** カラムの名前。
- **dst** DECL_DATETIME 値。

戻り値

値が正常にフェッチされた場合は true。

GetDateTime(ul_column_num, DECL_DATETIME *) メソッド

カラムから値を DECL_DATETIME としてフェッチします。

構文

```
public virtual bool GetDateTime(ul_column_num cid, DECL_DATETIME * dst)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。
- **dst** DECL_DATETIME 値。

戻り値

値が正常にフェッチされた場合は true。

GetDouble メソッド

カラムから値を double としてフェッチします。

オーバーロードリスト

名前	説明
GetDouble(const char *) メソッド	カラムから値を double としてフェッチします。
GetDouble(ul_column_num) メソッド	カラムから値を double としてフェッチします。

GetDouble(const char *) メソッド

カラムから値を double としてフェッチします。

構文

```
public virtual ul_double GetDouble(const char * cname)
```

パラメータ

- **cname** カラム名。

戻り値

double としてのカラム値。

GetDouble(ul_column_num) メソッド

カラムから値を double としてフェッチします。

構文

```
public virtual ul_double GetDouble(ul_column_num cid)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。

戻り値

double としてのカラム値。

GetFloat メソッド

カラムから値を float としてフェッチします。

オーバーロードリスト

名前	説明
GetFloat(const char *) メソッド	カラムから値を float としてフェッチします。

名前	説明
GetFloat(<i>ul_column_num</i>) メソッド	カラムから値を float としてフェッチします。

GetFloat(const char *) メソッド

カラムから値を float としてフェッチします。

構文

```
public virtual ul_real GetFloat(const char * cname)
```

パラメータ

- **cname** カラム名。

戻り値

float としてのカラム値。

GetFloat(*ul_column_num*) メソッド

カラムから値を float としてフェッチします。

構文

```
public virtual ul_real GetFloat(ul_column_num cid)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。

戻り値

float としてのカラム値。

GetGuid メソッド

カラムから値を GUID としてフェッチします。

オーバーロードリスト

名前	説明
GetGuid(const char *, GUID *) メソッド	カラムから値を GUID としてフェッチします。
GetGuid(<i>ul_column_num</i>, GUID *) メソッド	カラムから値を GUID としてフェッチします。

GetGuid(const char *, GUID *) メソッド

カラムから値を GUID としてフェッチします。

構文

```
public virtual bool GetGuid(const char * cname, GUID * dst)
```

パラメータ

- **cname** カラムの名前。
- **dst** GUID 値。

戻り値

値が正常にフェッチされた場合は true。

GetGuid(ul_column_num, GUID *) メソッド

カラムから値を GUID としてフェッチします。

構文

```
public virtual bool GetGuid(ul_column_num cid, GUID * dst)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。
- **dst** GUID 値。

戻り値

値が正常にフェッチされた場合は true。

GetInt メソッド

カラムから値を integer としてフェッチします。

オーバーロードリスト

名前	説明
GetInt(const char *) メソッド	カラムから値を integer としてフェッチします。
GetInt(ul_column_num) メソッド	カラムから値を integer としてフェッチします。

GetInt(const char *) メソッド

カラムから値を integer としてフェッチします。

構文

```
public virtual ul_s_long GetInt(const char * cname)
```

パラメータ

- **cname** カラム名。

戻り値

Integer としてのカラム値。

GetInt(ul_column_num) メソッド

カラムから値を integer としてフェッチします。

構文

```
public virtual ul_s_long GetInt(ul_column_num cid)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。

戻り値

Integer としてのカラム値。

GetIntWithType メソッド

カラムからの値を、指定された整数型としてフェッチします。

オーバーロードリスト

名前	説明
GetIntWithType(const char *, ul_column_storage_type) メソッド	カラムからの値を、指定された整数型としてフェッチします。
GetIntWithType(ul_column_num, ul_column_storage_type) メソッド	カラムからの値を、指定された整数型としてフェッチします。

GetIntWithType(const char *, ul_column_storage_type) メソッド

カラムからの値を、指定された整数型としてフェッチします。

構文

```
public virtual ul_s_big GetIntWithType (
    const char * cname,
    ul_column_storage_type type
)
```

パラメータ

- **cname** カラムの名前。
- **type** フェッチするときの整数型。

戻り値

Integer としてのカラム値。

備考

次に、type パラメータに使用できる integer 値のリストを示します。

- UL_TYPE_BIT
- UL_TYPE_TINY
- UL_TYPE_S_SHORT
- UL_TYPE_U_SHORT
- UL_TYPE_S_LONG
- UL_TYPE_U_LONG
- UL_TYPE_S_BIG
- UL_TYPE_U_BIG

参照

- [ul_column_storage_type 列挙体 \[Ultra Light C および Embedded SQL データタイプ\]90 ページ](#)

GetIntWithType(ul_column_num, ul_column_storage_type) メソッド

カラムからの値を、指定された整数型としてフェッチします。

構文

```
public virtual ul_s_big GetIntWithType (
    ul_column_num cid,
    ul_column_storage_type type
)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。

- **type** フェッチするときの整数型。

戻り値

integer としてのカラム値。

備考

次に、type パラメータに使用できる integer 値のリストを示します。

- **UL_TYPE_BIT**
- **UL_TYPE_TINY**
- **UL_TYPE_S_SHORT**
- **UL_TYPE_U_SHORT**
- **UL_TYPE_S_LONG**
- **UL_TYPE_U_LONG**
- **UL_TYPE_S_BIG**
- **UL_TYPE_U_BIG**

参照

- [ul_column_storage_type](#) 列挙体 [Ultra Light C および Embedded SQL データタイプ]90 ページ

GetResultSetSchema メソッド

結果セットに関する情報を取得するために使用できるオブジェクトを返します。

構文

```
public virtual const ULResultSetSchema & GetResultSetSchema()
```

戻り値

結果セットに関する情報を取得するために使用できる ULResultSetSchema オブジェクト。

GetRowCount メソッド

テーブルのローの数を取得します。

構文

```
public virtual ul_u_long GetRowCount(ul_u_long threshold)
```

パラメータ

- **threshold** カウントするローの数の制限。無限を表すには 0 を設定します。

戻り値

テーブル内のローの数。

備考

このメソッドは、"SELECT COUNT(*) FROM table" 文を実行するのと同じです。

GetState メソッド

カーソルの内部ステータスを取得します。

構文

```
public virtual UL_RS_STATE GetState()
```

戻り値

カーソルのステータス

参照

- [UL_RS_STATE 列挙体 \[Ultra Light C および Embedded SQL データタイプ\]88 ページ](#)

GetString メソッド

カラムから値を NULL で終了する文字列としてフェッチします。

オーバーロードリスト

名前	説明
GetString(const char *, char *, size_t) メソッド	カラムから値を NULL で終了する文字列としてフェッチします。
GetString(ul_column_num, char *, size_t) メソッド	カラムから値を NULL で終了する文字列としてフェッチします。

GetString(const char *, char *, size_t) メソッド

カラムから値を NULL で終了する文字列としてフェッチします。

構文

```
public virtual bool GetString(  
    const char * cname,  
    char * dst,  
    size_t len  
)
```

パラメータ

- **cname** カラムの名前。
- **dst** string 値を保持するバッファ。トランケートされる場合であっても、文字列は NULL で終了します。
- **len** バッファのサイズ (バイト単位)。

戻り値

値が正常にフェッチされた場合は **true**。

備考

値全体を保持できるほど大きくない場合、文字列はバッファ内でトランケートされます。

GetString(*ul_column_num*, *char **, *size_t*) メソッド

カラムから値を NULL で終了する文字列としてフェッチします。

構文

```
public virtual bool GetString(ul_column_num cid, char * dst, size_t len)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。
- **dst** string 値を保持するバッファ。トランケートされる場合であっても、文字列は NULL で終了します。
- **len** バッファのサイズ (バイト単位)。

戻り値

値が正常にフェッチされた場合は **true**。

備考

値全体を保持できるほど大きくない場合、文字列はバッファ内でトランケートされます。

GetStringChunk メソッド

カラムから文字列のチャンクを取得します。

オーバーロードリスト

名前	説明
GetStringChunk(const char *, char *, size_t, size_t) メソッド	カラムから文字列のチャンクを取得します。

名前	説明
GetStringChunk(ul_column_num, char *, size_t, size_t) メソッド	カラムから文字列のチャンクを取得します。

GetStringChunk(const char *, char *, size_t, size_t) メソッド

カラムから文字列のチャンクを取得します。

構文

```
public virtual size_t GetStringChunk(  
    const char * cname,  
    char * dst,  
    size_t len,  
    size_t offset  
)
```

パラメータ

- **cname** カラムの名前。
- **dst** 文字列のチャンクを保持するバッファ。トランケートされる場合であっても、文字列は NULL で終了します。
- **len** バッファのサイズ (バイト単位)。
- **offset** 値内でのオフセット (読み込み開始位置)、または前回の読み込みが終了したところから続行する場合は UL_BLOB_CONTINUE 定数。

戻り値

宛先のバッファにコピーされたバイト数 (NULL ターミネータを含まない)。dst 値を NULL に設定した場合は、文字列の残りのバイト数が返されます。カラムが NULL のときは、dst パラメータに空の文字列が返されます。IsNull メソッドを使用して、NULL と空の文字列を区別してください。

備考

0 が返される場合は、値の最後に到達しています。

参照

- [ULResultSet.IsNull メソッド \[Ultra Light C++\]192 ページ](#)

GetStringChunk(ul_column_num, char *, size_t, size_t) メソッド

カラムから文字列のチャンクを取得します。

構文

```
public virtual size_t GetStringChunk(  
    ul_column_num cid,
```

```
        char * dst,  
        size_t len,  
        size_t offset  
    )
```

パラメータ

- **cid** 1 から始まるカラムの順序数。
- **dst** 文字列のチャンクを保持するバッファ。トランケートされる場合であっても、文字列は NULL で終了します。
- **len** バッファのサイズ (バイト単位)。
- **offset** 値内でのオフセット (読み込み開始位置)、または前回の読み込みが終了したところから続行する場合は `UL_BLOB_CONTINUE` 定数に設定します。

戻り値

宛先のバッファにコピーされたバイト数 (NULL ターミネータを含まない)。dst 値を NULL に設定した場合は、文字列の残りのバイト数が返されます。カラムが NULL のときは、dst パラメータに空の文字列が返されます。IsNull メソッドを使用して、NULL と空の文字列を区別してください。

備考

0 が返される場合は、値の最後に到達しています。

参照

- [ULResultSet.IsNull メソッド \[Ultra Light C++\]192 ページ](#)

GetStringLength メソッド

カラムの値の文字列長を取得します。

オーバーロードリスト

名前	説明
GetStringLength(const char *) メソッド	カラムの値の文字列長を取得します。
GetStringLength(ul_column_num) メソッド	カラムの値の文字列長を取得します。

GetStringLength(const char *) メソッド

カラムの値の文字列長を取得します。

構文

```
public virtual size_t GetStringLength(const char * cname)
```

パラメータ

- **cname** カラム名。

戻り値

いずれかの `GetString` メソッドによって返される文字列を保持するために必要なバイト数または文字数 (NULL ターミネータを含まない)。

備考

次の例は、カラムの文字列長の取得方法を示します。

```
len = result_set->GetStringLength( cid );  
dst = new char[ len + 1 ];  
result_set->GetString( cid, dst, len + 1 );
```

ワイド文字の場合の使用方法を次に示します。

```
len = result_set->GetStringLength( cid );  
dst = new ul_wchar[ len + 1 ];  
result_set->GetString( cid, dst, len + 1 );
```

参照

- [ULResultSet.GetString メソッド \[Ultra Light C++\]187 ページ](#)

GetStringLength(ul_column_num) メソッド

カラムの値の文字列長を取得します。

構文

```
public virtual size_t GetStringLength(ul_column_num cid)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。

戻り値

いずれかの `GetString` メソッドによって返される文字列を保持するために必要なバイト数または文字数 (NULL ターミネータを含まない)。

備考

次の例は、カラムの文字列長の取得方法を示します。

```
len = result_set->GetStringLength( cid );  
dst = new char[ len + 1 ];  
result_set->GetString( cid, dst, len + 1 );
```

ワイド文字の場合の使用方法を次に示します。

```
len = result_set->GetStringLength( cid );  
dst = new ul_wchar[ len + 1 ];  
result_set->GetString( cid, dst, len + 1 );
```

参照

- [ULResultSet.GetString メソッド \[Ultra Light C++\]187 ページ](#)

IsNull メソッド

カラムが NULL かどうかをチェックします。

オーバーロードリスト

名前	説明
IsNull(const char *) メソッド	カラムが NULL かどうかをチェックします。
IsNull(ul_column_num) メソッド	カラムが NULL かどうかをチェックします。

IsNull(const char *) メソッド

カラムが NULL かどうかをチェックします。

構文

```
public virtual bool IsNull(const char * cname)
```

パラメータ

- **cname** カラム名。

戻り値

カラムの値が NULL の場合は true。

IsNull(ul_column_num) メソッド

カラムが NULL かどうかをチェックします。

構文

```
public virtual bool IsNull(ul_column_num cid)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。

戻り値

カラムの値が NULL の場合は true。

Last メソッド

カーソルを最後のローに移動します。

構文

```
public virtual bool Last()
```

戻り値

成功した場合は **true**、失敗した場合は **false**。

Next メソッド

カーソルをロー 1 つ分進めます。

構文

```
public virtual bool Next()
```

戻り値

カーソルが正常に進められる場合は、**true**。**true** が返されても、カーソルが次のローに正常に移動したときに、エラーが送信されることがあります。たとえば **SELECT** 式の評価中に変換エラーが発生する可能性があります。この場合、カラム値を取得するときにもエラーが返されます。カーソルを進められなかった場合は、**false** が返されます。たとえば、次のローが存在しない可能性があります。この場合、移動後のカーソル位置は最後のローの後ろに設定されます。

Previous メソッド

カーソルをロー 1 つ分戻します。

構文

```
public virtual bool Previous()
```

戻り値

カーソルをロー 1 つ分戻せた場合は、**true**。カーソルを戻せなかった場合は、**false**。移動後のカーソル位置は、最初のローの前に設定されます。

Relative メソッド

カーソルを、現在のカーソルの位置から、**offset** で指定したロー数分移動します。

構文

```
public virtual bool Relative(ul_fetch_offset offset)
```

パラメータ

- **offset** 移動するローの数。

戻り値

成功した場合は **true**、失敗した場合は **false**。

SetBinary メソッド

カラムを ul_binary 値に設定します。

オーバロードリスト

名前	説明
SetBinary(const char *, p_ul_binary) メソッド	カラムを ul_binary 値に設定します。
SetBinary(ul_column_num, p_ul_binary) メソッド	カラムを ul_binary 値に設定します。

SetBinary(const char *, p_ul_binary) メソッド

カラムを ul_binary 値に設定します。

構文

```
public virtual bool SetBinary(const char * cname, p_ul_binary value)
```

パラメータ

- **cname** カラムの名前。
- **value** ul_binary 値。NULL を渡すことは、SetNull メソッドを呼び出すことと同じです。

戻り値

成功した場合は true、失敗した場合は false。

SetBinary(ul_column_num, p_ul_binary) メソッド

カラムを ul_binary 値に設定します。

構文

```
public virtual bool SetBinary(ul_column_num cid, p_ul_binary value)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。
- **value** ul_binary 値。NULL を渡すことは、SetNull メソッドを呼び出すことと同じです。

戻り値

成功した場合は true、失敗した場合は false。

SetDateTime メソッド

カラムを DECL_DATETIME 値に設定します。

オーバロードリスト

名前	説明
SetDateTime(const char *, DECL_DATETIME *) メソッド	カラムを DECL_DATETIME 値に設定します。
SetDateTime(ul_column_num, DECL_DATETIME *) メソッド	カラムを DECL_DATETIME 値に設定します。

SetDateTime(const char *, DECL_DATETIME *) メソッド

カラムを DECL_DATETIME 値に設定します。

構文

```
public virtual bool SetDateTime(  
    const char * cname,  
    DECL_DATETIME * value  
)
```

パラメータ

- **cname** カラムの名前。
- **value** DECL_DATETIME 値。NULL を渡すことは、SetNull メソッドを呼び出すことと同じです。

戻り値

成功した場合は true、失敗した場合は false。

SetDateTime(ul_column_num, DECL_DATETIME *) メソッド

カラムを DECL_DATETIME 値に設定します。

構文

```
public virtual bool SetDateTime(  
    ul_column_num cid,  
    DECL_DATETIME * value  
)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。
- **value** DECL_DATETIME 値。NULL を渡すことは、SetNull メソッドを呼び出すことと同じです。

戻り値

成功した場合は true、失敗した場合は false。

SetDefault メソッド

カラムをそのデフォルト値に設定します。

オーバーロードリスト

名前	説明
SetDefault(const char *) メソッド	カラムをそのデフォルト値に設定します。
SetDefault(ul_column_num) メソッド	カラムをそのデフォルト値に設定します。

SetDefault(const char *) メソッド

カラムをそのデフォルト値に設定します。

構文

```
public virtual bool SetDefault(const char * cname)
```

パラメータ

- **cname** カラム名。

戻り値

成功した場合は true、失敗した場合は false。

SetDefault(ul_column_num) メソッド

カラムをそのデフォルト値に設定します。

構文

```
public virtual bool SetDefault(ul_column_num cid)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。

戻り値

成功した場合は true、失敗した場合は false。

SetDouble メソッド

カラムを double 値に設定します。

オーバードリスト

名前	説明
SetDouble(const char *, ul_double) メソッド	カラムを double 値に設定します。
SetDouble(ul_column_num, ul_double) メソッド	カラムを double 値に設定します。

SetDouble(const char *, ul_double) メソッド

カラムを double 値に設定します。

構文

```
public virtual bool SetDouble(const char * cname, ul_double value)
```

パラメータ

- **cname** カラムの名前。
- **value** double 値。

戻り値

成功した場合は true、失敗した場合は false。

SetDouble(ul_column_num, ul_double) メソッド

カラムを double 値に設定します。

構文

```
public virtual bool SetDouble(ul_column_num cid, ul_double value)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。
- **value** double 値。

戻り値

成功した場合は true、失敗した場合は false。

SetFloat メソッド

カラムを float 値に設定します。

オーバロードリスト

名前	説明
SetFloat(const char *, ul_real) メソッド	カラムを float 値に設定します。
SetFloat(ul_column_num, ul_real) メソッド	カラムを float 値に設定します。

SetFloat(const char *, ul_real) メソッド

カラムを float 値に設定します。

構文

```
public virtual bool SetFloat(const char * cname, ul_real value)
```

パラメータ

- **cname** カラムの名前。
- **value** float 値。

戻り値

成功した場合は true、失敗した場合は false。

SetFloat(ul_column_num, ul_real) メソッド

カラムを float 値に設定します。

構文

```
public virtual bool SetFloat(ul_column_num cid, ul_real value)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。
- **value** float 値。

戻り値

成功した場合は true、失敗した場合は false。

SetGuid メソッド

カラムを GUID 値に設定します。

オーバロードリスト

名前	説明
SetGuid(const char *, GUID *) メソッド	カラムを GUID 値に設定します。
SetGuid(ul_column_num, GUID *) メソッド	カラムを GUID 値に設定します。

SetGuid(const char *, GUID *) メソッド

カラムを GUID 値に設定します。

構文

```
public virtual bool SetGuid(const char * cname, GUID * value)
```

パラメータ

- **cname** カラムの名前。
- **value** GUID 値。NULL を渡すことは、SetNull メソッドを呼び出すことと同じです。

戻り値

成功した場合は true、失敗した場合は false。

SetGuid(ul_column_num, GUID *) メソッド

カラムを GUID 値に設定します。

構文

```
public virtual bool SetGuid(ul_column_num cid, GUID * value)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。
- **value** GUID 値。NULL を渡すことは、SetNull メソッドを呼び出すことと同じです。

戻り値

成功した場合は true、失敗した場合は false。

SetInt メソッド

カラムを integer 値に設定します。

オーバロードリスト

名前	説明
SetInt(const char *, ul_s_long) メソッド	カラムを integer 値に設定します。
SetInt(ul_column_num, ul_s_long) メソッド	カラムを integer 値に設定します。

SetInt(const char *, ul_s_long) メソッド

カラムを integer 値に設定します。

構文

```
public virtual bool SetInt(const char * cname, ul_s_long value)
```

パラメータ

- **cname** カラム名。
- **value** 符号付き整数値。

戻り値

成功した場合は true、失敗した場合は false。

SetInt(ul_column_num, ul_s_long) メソッド

カラムを integer 値に設定します。

構文

```
public virtual bool SetInt(ul_column_num cid, ul_s_long value)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。
- **value** 符号付き整数値。

戻り値

成功した場合は true、失敗した場合は false。

SetIntWithType メソッド

カラムを、指定された整数型の integer 値に設定します。

オーバーロードリスト

名前	説明
SetIntWithType(const char *, ul_s_big, ul_column_storage_type) メソッド	カラムを、指定された整数型の integer 値に設定します。
SetIntWithType(ul_column_num, ul_s_big, ul_column_storage_type) メソッド	カラムを、指定された整数型の integer 値に設定します。

SetIntWithType(const char *, ul_s_big, ul_column_storage_type) メソッド

カラムを、指定された整数型の integer 値に設定します。

構文

```
public virtual bool SetIntWithType(  
    const char * cname,  
    ul_s_big value,  
    ul_column_storage_type type  
)
```

パラメータ

- **cname** カラムの名前。
- **value** integer 値。
- **type** この値を処理するときの整数型。

戻り値

成功した場合は true、失敗した場合は false。

備考

次に、value パラメータに使用できる integer 値のリストを示します。

- UL_TYPE_BIT
- UL_TYPE_TINY
- UL_TYPE_S_SHORT
- UL_TYPE_U_SHORT
- UL_TYPE_S_LONG
- UL_TYPE_U_LONG
- UL_TYPE_S_BIG
- UL_TYPE_U_BIG

参照

- [ul_column_storage_type](#) 列挙体 [Ultra Light C および Embedded SQL データタイプ]90 ページ

SetIntWithType([ul_column_num](#), [ul_s_big](#), [ul_column_storage_type](#)) メソッド

カラムを、指定された整数型の integer 値に設定します。

構文

```
public virtual bool SetIntWithType(  
    ul\_column\_num cid,  
    ul\_s\_big value,  
    ul\_column\_storage\_type type  
)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。
- **value** integer 値。
- **type** この値を処理するときの整数型。

戻り値

成功した場合は true、失敗した場合は false。

備考

次に、value パラメータに使用できる integer 値のリストを示します。

- [UL_TYPE_BIT](#)
- [UL_TYPE_TINY](#)
- [UL_TYPE_S_SHORT](#)
- [UL_TYPE_U_SHORT](#)
- [UL_TYPE_S_LONG](#)
- [UL_TYPE_U_LONG](#)
- [UL_TYPE_S_BIG](#)
- [UL_TYPE_U_BIG](#)

参照

- [ul_column_storage_type](#) 列挙体 [Ultra Light C および Embedded SQL データタイプ]90 ページ

SetNull メソッド

カラムを NULL に設定します。

オーバーロードリスト

名前	説明
SetNull(const char *) メソッド	カラムを NULL に設定します。
SetNull(ul_column_num) メソッド	カラムを NULL に設定します。

SetNull(const char *) メソッド

カラムを NULL に設定します。

構文

```
public virtual bool SetNull(const char * cname)
```

パラメータ

● **cname** カラム名。

戻り値

成功した場合は true、失敗した場合は false。

SetNull(ul_column_num) メソッド

カラムを NULL に設定します。

構文

```
public virtual bool SetNull(ul_column_num cid)
```

パラメータ

● **cid** 1 から始まるカラムの順序数。

戻り値

成功した場合は true、失敗した場合は false。

SetString メソッド

カラムを string 値に設定します。

オーバーロードリスト

名前	説明
SetString(const char *, const char *, size_t) メソッド	カラムを string 値に設定します。
SetString(ul_column_num, const char *, size_t) メソッド	カラムを string 値に設定します。

SetString(const char *, const char *, size_t) メソッド

カラムを string 値に設定します。

構文

```
public virtual bool SetString(
    const char * cname,
    const char * value,
    size_t len
)
```

パラメータ

- **cname** カラムの名前。
- **value** string 値。NULL を渡すことは、SetNull メソッドを呼び出すことと同じです。
- **len** 省略可。文字列の長さ (バイト単位)、または文字列が NULL で終了する場合は UL_NULL_TERMINATED_STRING 定数。設定された len 値が 32K より大きい場合は、SQLE_INVALID_PARAMETER 定数が設定されます。サイズの大きい文字列の場合は、代わりに AppendStringChunk メソッドが呼び出されます。

戻り値

成功した場合は true、失敗した場合は false。

参照

- [ULResultSet.AppendStringChunk メソッド \[Ultra Light C++\]173 ページ](#)

SetString(ul_column_num, const char *, size_t) メソッド

カラムを string 値に設定します。

構文

```
public virtual bool SetString(
    ul_column_num cid,
    const char * value,
    size_t len
)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。
- **value** string 値。NULL を渡すことは、SetNull メソッドを呼び出すことと同じです。
- **len** 省略可。文字列の長さ (バイト単位)、または文字列が NULL で終了する場合は UL_NULL_TERMINATED_STRING 定数。設定された **len** 値が 32K より大きい場合は、SQLE_INVALID_PARAMETER 定数が設定されます。サイズの大きい文字列の場合は、代わりに AppendStringChunk メソッドが呼び出されます。

戻り値

成功した場合は true、失敗した場合は false。

参照

- [ULResultSet.AppendStringChunk メソッド \[Ultra Light C++\]173 ページ](#)

Update メソッド

現在の行を更新します。

構文

```
public virtual bool Update()
```

戻り値

成功した場合は true、失敗した場合は false。

UpdateBegin メソッド

カラムの設定に使用される更新モードを選択します。

構文

```
public virtual bool UpdateBegin()
```

戻り値

成功した場合は true、失敗した場合は false。

備考

更新モードの場合、プライマリー内のカラムの修正はできません。

ULResultSetSchema クラス

Ultra Light の結果セットのスキーマを表します。

構文

```
public class ULResultSetSchema
```

派生クラス

- [ULTableSchema クラス \[Ultra Light C++\]219 ページ](#)

メンバー

継承されたメンバーを含む **ULResultSetSchema** クラスのすべてのメンバー。

名前	説明
GetColumnCount メソッド	結果セットまたはテーブル内のカラム数を取得します。
GetColumnID メソッド	1 から始まるカラム ID を名前から取得します。
GetColumnName メソッド	1 から始まる ID を指定してカラムの名前を取得します。
GetColumnPrecision メソッド	数値カラムの精度を取得します。
GetColumnScale メソッド	数値カラムの位取りを取得します。
GetColumnSize メソッド	カラムのサイズを取得します。
GetColumnSQLType メソッド	カラムの SQL の型を取得します。
GetColumnType メソッド	カラムの記憶タイプまたはホスト変数の型を取得します。
GetConnection メソッド	ULConnection オブジェクトを取得します。
IsAliased メソッド	結果セット内のカラムにエイリアスが付与されているかどうかを示します。

GetColumnCount メソッド

結果セットまたはテーブル内のカラム数を取得します。

構文

```
public virtual ul_column_num GetColumnCount()
```

戻り値

結果セットまたはテーブル内のカラム数。

GetColumnID メソッド

1 から始まるカラム ID を名前から取得します。

構文

```
public virtual ul_column_num GetColumnID(const char * columnName)
```

パラメータ

- **columnName** カラムの名前。

戻り値

カラムが存在しない場合は 0。それ以外で、カラム名が存在しない場合は `SQLE_COLUMN_NOT_FOUND`。

GetColumnName メソッド

1 から始まる ID を指定してカラムの名前を取得します。

構文

```
public virtual const char * GetColumnName(  
    ul_column_num cid,  
    ul_column_name_type type  
)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。
- **type** カラム名の必要な型。

戻り値

存在する場合は、カラム名を格納する文字列バッファへのポインタ。このポインタは静的バッファを指します。静的バッファの内容は、それ以降の **Ultra Light** の呼び出しによって変更される可能性があるため、値を一時的に保持したい場合はその値のコピーを作成する必要があります。カラムが存在しない場合は `NULL` が返され、`SQLE_COLUMN_NOT_FOUND` が設定されます。

備考

選択した型、および `SELECT` 文でのカラムの宣言方法によっては、カラム名が `[table- name]. [column-name]` という形式で返されることがあります。

`type` パラメータを使用して、どの型のカラム名を返すかを指定します。

参照

- [ul_column_name_type 列挙体 \[Ultra Light C++\]226 ページ](#)

GetColumnPrecision メソッド

数値カラムの精度を取得します。

構文

```
public virtual size_t GetColumnPrecision (ul_column_num cid)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。

戻り値

カラムが数値型ではないか、カラムが存在しない場合は、0 を返します。カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND が設定されます。カラム型が数値ではない場合は、SQLE_DATATYPE_NOT_ALLOWED が設定されます。

GetColumnScale メソッド

数値カラムの位取りを取得します。

構文

```
public virtual size_t GetColumnScale (ul_column_num cid)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。

戻り値

カラムが数値型ではないか、カラムが存在しない場合は、0 を返します。カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND が設定されます。カラム型が数値ではない場合は、SQLE_DATATYPE_NOT_ALLOWED が設定されます。

GetColumnSize メソッド

カラムのサイズを取得します。

構文

```
public virtual size_t GetColumnSize (ul_column_num cid)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。

戻り値

カラムが存在しないか、カラムの型が可変長ではない場合は、0 を返します。カラム名が存在しない場合は、SQLE_COLUMN_NOT_FOUND が設定されます。カラム型が UL_SQLTYPE_CHAR か UL_SQLTYPE_BINARY でない場合は、SQLE_DATATYPE_NOT_ALLOWED が設定されます。

GetColumnSQLType メソッド

カラムの SQL の型を取得します。

構文

```
public virtual ul_column_sql_type GetColumnSQLType(ul_column_num cid)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。

戻り値

カラムが存在しない場合は、UL_SQLTYPE_BAD_INDEX。

参照

- [ul_column_sql_type 列挙体 \[Ultra Light C および Embedded SQL データタイプ\]88 ページ](#)

GetColumnType メソッド

カラムの記憶タイプまたはホスト変数の型を取得します。

構文

```
public virtual ul_column_storage_type GetColumnType(ul_column_num cid)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。

戻り値

カラムが存在しない場合は、UL_TYPE_BAD_INDEX。

参照

- [ul_column_storage_type 列挙体 \[Ultra Light C および Embedded SQL データタイプ\]90 ページ](#)

GetConnection メソッド

ULConnection オブジェクトを取得します。

構文

```
public virtual ULConnection * GetConnection()
```

戻り値

この結果セットスキーマに関連付けられている ULConnection オブジェクト。

IsAliased メソッド

結果セット内のカラムにエイリアスが付与されているかどうかを示します。

構文

```
public virtual bool IsAliased(ul_column_num cid)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。

戻り値

カラムのエイリアスが使用されている場合は true、そうでない場合は false。

ULTable クラス

Ultra Light データベース内のテーブルを表します。

構文

```
public class ULTable : ULResultSet
```

基本クラス

- [ULResultSet クラス \[Ultra Light C++\]169 ページ](#)

メンバー

継承されたメンバーを含む ULTable クラスのすべてのメンバー。

名前	説明
AfterLast メソッド	カーソルを最後のローの後に移動します。
AppendByteChunk メソッド	カラムにバイトを追加します。
AppendStringChunk メソッド	文字列のチャンクをカラムに追加します。
BeforeFirst メソッド	カーソルを最初のローの前に移動します。
Close メソッド	このオブジェクトを破棄します。
Delete メソッド	現在のローを削除し、次の有効なローに移動します。
DeleteAllRows メソッド	テーブルからすべてのローを削除します。
DeleteNamed メソッド	現在のローを削除し、次の有効なローに移動します。

名前	説明
Find メソッド	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実行します。
FindBegin メソッド	検索モードを開始することで、テーブルで新規に検索呼び出しを実行する準備を行います。
FindFirst メソッド	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実行します。
FindLast メソッド	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実行します。
FindNext メソッド	インデックスに完全に一致する次のローを取得します。
FindPrevious メソッド	インデックスに完全に一致する前のローを取得します。
First メソッド	カーソルを最初のローに移動します。
GetBinary メソッド	カラムから値を <code>ul_binary</code> 値としてフェッチします。
GetBinaryLength メソッド	カラムの値のバイナリ長を取得します。
GetByteChunk メソッド	カラムからバイナリのチャンクを取得します。
GetConnection メソッド	接続オブジェクトを取得します。
GetDateTime メソッド	カラムから値を <code>DECL_DATETIME</code> としてフェッチします。
GetDouble メソッド	カラムから値を <code>double</code> としてフェッチします。
GetFloat メソッド	カラムから値を <code>float</code> としてフェッチします。
GetGuid メソッド	カラムから値を <code>GUID</code> としてフェッチします。
GetInt メソッド	カラムから値を <code>integer</code> としてフェッチします。

名前	説明
GetIntWithType メソッド	カラムからの値を、指定された整数型としてフェッチします。
GetResultSetSchema メソッド	結果セットに関する情報を取得するために使用できるオブジェクトを返します。
GetRowCount メソッド	テーブルのローの数を取得します。
GetState メソッド	カーソルの内部ステータスを取得します。
GetString メソッド	カラムから値を NULL で終了する文字列としてフェッチします。
GetStringChunk メソッド	カラムから文字列のチャンクを取得します。
GetStringLength メソッド	カラムの値の文字列長を取得します。
GetTableSchema メソッド	テーブルに関するスキーマ情報の取得に使用できる ULTableSchema オブジェクトを返します。
Insert メソッド	新しいローをテーブルに挿入します。
InsertBegin メソッド	設定されたカラムに対する挿入モードを選択します。
IsNull メソッド	カラムが NULL かどうかをチェックします。
Last メソッド	カーソルを最後のローに移動します。
Lookup メソッド	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実行します。
LookupBackward メソッド	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実行します。
LookupBegin メソッド	テーブルで新規に検索を実行する準備を行います。
LookupForward メソッド	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実行します。
Next メソッド	カーソルをロー 1 つ分進めます。

名前	説明
Previous メソッド	カーソルをロー 1 つ分戻します。
Relative メソッド	カーソルを、現在のカーソルの位置から、 <code>offset</code> で指定したロー数分移動します。
SetBinary メソッド	カラムを <code>ul_binary</code> 値に設定します。
SetDateTime メソッド	カラムを <code>DECL_DATETIME</code> 値に設定します。
SetDefault メソッド	カラムをそのデフォルト値に設定します。
SetDouble メソッド	カラムを <code>double</code> 値に設定します。
SetFloat メソッド	カラムを <code>float</code> 値に設定します。
SetGuid メソッド	カラムを <code>GUID</code> 値に設定します。
SetInt メソッド	カラムを <code>integer</code> 値に設定します。
SetIntWithType メソッド	カラムを、指定された整数型の <code>integer</code> 値に設定します。
SetNull メソッド	カラムを <code>NULL</code> に設定します。
SetString メソッド	カラムを <code>string</code> 値に設定します。
TruncateTable メソッド	テーブルをトランケートし、 <code>STOP SYNCHRONIZATION DELETE</code> を一時的にアクティブにします。
Update メソッド	現在の行を更新します。
UpdateBegin メソッド	カラムの設定に使用される更新モードを選択します。

DeleteAllRows メソッド

テーブルからすべてのローを削除します。

構文

```
public virtual bool DeleteAllRows()
```

戻り値

成功した場合は `true`、失敗した場合は `false`。たとえば、テーブルが開いていない場合や SQL エラーが発生した場合は、`false` が返されます。

備考

アプリケーションによっては、テーブル内のローをすべて削除してから、新しいデータセットをテーブルにダウンロードする方が便利ことがあります。接続で `stop sync` プロパティが設定されている場合は、削除されたローが同期されません。

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。このような挿入は、他の接続が `DeleteAllRows` メソッドを呼び出した後にロールバックを実行した場合にも削除されません。

インデックスを使用しないでこのテーブルを開いた場合、テーブルは読み込み専用とみなされ、データを削除できません。

Find メソッド

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実行します。

構文

```
public virtual bool Find(ul_column_num ncols)
```

パラメータ

- **ncols** 複合インデックスの場合に、検索中に使用するカラムの数。

戻り値

インデックスの値に一致するローがない場合は、カーソルの位置が最後のローの後ろに設定され、`false` が返されます。

備考

検索する値を指定するには、インデックスのカラムごとに値を設定します。カーソルは、インデックスの値と完全に一致した最初のローで停止します。

FindBegin メソッド

検索モードを開始することで、テーブルで新規に検索呼び出しを実行する準備を行います。

構文

```
public virtual bool FindBegin()
```

戻り値

成功した場合は `true`、失敗した場合は `false`。

備考

テーブルを開くのに使用されたインデックス内のカラムのみを設定できます。インデックスを使用しないでテーブルを開いた場合は、このメソッドを呼び出せません。

FindFirst メソッド

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実行します。

構文

```
public virtual bool FindFirst(ul_column_num ncols)
```

パラメータ

- **ncols** 複合インデックスの場合に、検索中に使用するカラムの数。

戻り値

インデックスの値に一致するローがない場合は、カーソルの位置が最後のローの後ろに設定され、false が返されます。

備考

検索する値を指定するには、インデックスのカラムごとに値を設定します。カーソルは、インデックスの値と完全に一致した最初のローで停止します。

FindLast メソッド

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実行します。

構文

```
public virtual bool FindLast(ul_column_num ncols)
```

パラメータ

- **ncols** 複合インデックスの場合に、検索中に使用するカラムの数。

戻り値

インデックスの値に一致するローがない場合は、カーソルの位置が最初のローの前に設定され、false が返されます。

備考

検索する値を指定するには、インデックスのカラムごとに値を設定します。カーソルは、インデックスの値と完全に一致した最初のローで停止します。

FindNext メソッド

インデックスに完全に一致する次のローを取得します。

構文

```
public virtual bool FindNext(ul_column_num ncols)
```

パラメータ

- **ncols** 複合インデックスの場合に、検索中に使用するカラムの数。

戻り値

それ以上インデックスに一致するローがない場合は、**false**。この場合、カーソルは最後のローの後ろに配置されます。

FindPrevious メソッド

インデックスに完全に一致する前のローを取得します。

構文

```
public virtual bool FindPrevious(ul_column_num ncols)
```

パラメータ

- **ncols** 複合インデックスの場合に、検索中に使用するカラムの数。

戻り値

それ以上インデックスに一致するローがない場合は、**false**。この場合、カーソルは最初のローの前に配置されます。

GetTableSchema メソッド

テーブルに関するスキーマ情報の取得に使用できる ULTableSchema オブジェクトを返します。

構文

```
public virtual ULTableSchema * GetTableSchema()
```

戻り値

テーブルに関するスキーマ情報の取得に使用できる ULTableSchema オブジェクト。

Insert メソッド

新しいローをテーブルに挿入します。

構文

```
public virtual bool Insert()
```

戻り値

成功した場合は **true**、失敗した場合は **false**。

InsertBegin メソッド

設定されたカラムに対する挿入モードを選択します。

構文

```
public virtual bool InsertBegin()
```

戻り値

成功した場合は **true**、失敗した場合は **false**。

備考

Set メソッド呼び出しによって代わりの値が指定されない場合、すべてのカラムは挿入時に初期値に設定されます。

Lookup メソッド

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実行します。

構文

```
public virtual bool 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

戻り値

ルックアップ後のカーソル位置が最後のローの後ろに設定された場合は **false**。

備考

検索する値を指定するには、インデックスのカラムごとに値を設定します。カーソルは、インデックスの値に一致するか、それより少ない値の最後のローで停止します。複合インデックスの場合、ncols パラメータはルックアップで使用するカラムの数を指定します。

LookupBackward メソッド

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実行します。

構文

```
public virtual bool LookupBackward(ul_column_num ncols)
```

パラメータ

- **ncols** 複合インデックスの場合にルックアップで使用するカラムの数。

戻り値

ルックアップ後のカーソル位置が最初のローの前に設定された場合は **false**。

備考

検索する値を指定するには、インデックスのカラムごとに値を設定します。カーソルは、インデックスの値に一致するか、それより少ない値の最後のローで停止します。複合インデックスの場合、ncols パラメータはルックアップで使用するカラムの数を指定します。

LookupBegin メソッド

テーブルで新規に検索を実行する準備を行います。

構文

```
public virtual bool LookupBegin()
```

戻り値

成功した場合は true、失敗した場合は false。

備考

テーブルを開くのに使用されたインデックス内のカラムのみを設定できます。インデックスを使用しないでテーブルを開いた場合は、このメソッドを呼び出せません。

LookupForward メソッド

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実行します。

構文

```
public virtual bool LookupForward(ul_column_num ncols)
```

パラメータ

- **ncols** 複合インデックスの場合にルックアップで使用するカラムの数。

戻り値

ルックアップ後のカーソル位置が最後のローの後ろに設定された場合は false。

備考

検索する値を指定するには、インデックスのカラムごとに値を設定します。カーソルは、インデックスの値に一致するか、それより少ない値の最後のローで停止します。複合インデックスの場合、ncols パラメータはルックアップで使用するカラムの数を指定します。

TruncateTable メソッド

テーブルをトランケートし、STOP SYNCHRONIZATION DELETE を一時的にアクティブにします。

構文

```
public virtual bool TruncateTable()
```


戻り値

成功した場合は true、失敗した場合は false。

ULTableSchema クラス

Ultra Light のテーブルのスキーマを表します。

構文

```
public class ULTableSchema : ULResultSetSchema
```

基本クラス

- [ULResultSetSchema クラス \[Ultra Light C++\]205 ページ](#)

メンバー

継承されたメンバーを含む ULTableSchema クラスのすべてのメンバー。

名前	説明
Close メソッド	このオブジェクトを破棄します。
GetColumnCount メソッド	結果セットまたはテーブル内のカラム数を取得します。
GetColumnDefault メソッド	カラムのデフォルト値が存在する場合は取得します。
GetColumnDefaultType メソッド	カラムのデフォルトの型を取得します。
GetColumnID メソッド	1 から始まるカラム ID を名前から取得します。
GetColumnName メソッド	1 から始まる ID を指定してカラムの名前を取得します。
GetColumnPrecision メソッド	数値カラムの精度を取得します。
GetColumnScale メソッド	数値カラムの位取りを取得します。
GetColumnSize メソッド	カラムのサイズを取得します。
GetColumnSQLType メソッド	カラムの SQL の型を取得します。
GetColumnType メソッド	カラムの記憶タイプまたはホスト変数の型を取得します。
GetConnection メソッド	ULConnection オブジェクトを取得します。

名前	説明
GetGlobalAutoincPartitionSize メソッド	分割サイズを取得します。
GetIndexCount メソッド	テーブル内のインデックス数を取得します。
GetIndexSchema メソッド	名前を指定してインデックスのスキーマを取得します。
GetName メソッド	テーブルの名前を取得します。
GetNextIndex メソッド	テーブル内の次のインデックス (スキーマ) を取得します。
GetOptimalIndex メソッド	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。
GetPrimaryKey メソッド	テーブルのプライマリキーを取得します。
GetPublicationPredicate メソッド	文字列としてのパブリケーション述部を取得します。
GetTableSyncType メソッド	テーブルの同期タイプを取得します。
InPublication メソッド	テーブルが、指定されたパブリケーションに含まれているかどうかをチェックします。
IsAliased メソッド	結果セット内のカラムにエイリアスが付与されているかどうかを示します。
IsColumnInIndex メソッド	カラムが、指定されたインデックスに含まれているかどうかをチェックします。
IsColumnNullable メソッド	指定されたカラムが NULL 入力可であるかどうかをチェックします。

Close メソッド

このオブジェクトを破棄します。

構文

```
public virtual void Close()
```

GetColumnDefault メソッド

カラムのデフォルト値が存在する場合は取得します。

構文

```
public virtual const char * GetColumnDefault(ul_column_num cid)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。

戻り値

デフォルト値。カラムにデフォルト値がない場合は、空の文字列を返します。この値は静的バッファを指します。静的バッファの内容は、それ以降の Ultra Light の呼び出しによって変更される可能性があるため、値を保持したい場合は、そのコピーを作成してください。

GetColumnDefaultType メソッド

カラムのデフォルトの型を取得します。

構文

```
public virtual ul_column_default_type GetColumnDefaultType(  
    ul_column_num cid  
)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。

戻り値

カラムのデフォルトの型。

参照

- [ul_column_default_type 列挙体 \[Ultra Light C++\]225 ページ](#)

GetGlobalAutoincPartitionSize メソッド

分割サイズを取得します。

構文

```
public virtual bool GetGlobalAutoincPartitionSize(  
    ul_column_num cid,  
    ul_u_big * size  
)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。
- **size** 出力パラメータ。カラムの分割サイズ。テーブルのすべてのグローバルオートインクリメントカラムは、同じグローバルオートインクリメントの分割サイズを共有します。

戻り値

成功した場合は `true`、失敗した場合は `false`。

GetIndexCount メソッド

テーブル内のインデックス数を取得します。

構文

```
public virtual ul_index_num GetIndexCount()
```

戻り値

テーブル内のインデックス数。

備考

インデックスの ID とカウントは、スキーマのアップグレード中に変更されることがあります。インデックスを正しく識別するには、名前でアクセスするか、キャッシュされている ID とカウントをスキーマのアップグレード後に再表示します。

GetIndexSchema メソッド

名前を指定してインデックスのスキーマを取得します。

構文

```
public virtual ULIndexSchema * GetIndexSchema(const char * indexName)
```

パラメータ

● **indexName** インデックス名。

戻り値

指定されたインデックスの `ULIndexSchema` オブジェクト、または、このオブジェクトが存在しない場合は `NULL`。

GetName メソッド

テーブルの名前を取得します。

構文

```
public virtual const char * GetName()
```

戻り値

テーブル名。この値は静的バッファを指します。静的バッファの内容は、それ以降の `Ultra Light` の呼び出しによって変更される可能性があるため、値を保持したい場合は、そのコピーを作成してください。

GetNextIndex メソッド

テーブル内の次のインデックス (スキーマ) を取得します。

構文

```
public virtual ULIndexSchema * GetNextIndex(ul_index_iter * iter)
```

パラメータ

- **iter** 繰り返し変数へのポインタ。

戻り値

ULIndexSchema オブジェクト。または、反復が完了したときは NULL。

備考

最初の呼び出しの前に、iter 値を ul_index_iter_start 定数に初期化します。

参照

- [ul_index_iter_start 変数 \[Ultra Light C++\]230 ページ](#)

GetOptimalIndex メソッド

カラム値を検索するのに最適なインデックスを特定します。

構文

```
public virtual const char * GetOptimalIndex(ul_column_num cid)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。

戻り値

インデックスの名前、またはカラムがインデックス付けされていない場合は NULL。この値は静的バッファを指します。静的バッファの内容は、それ以降の Ultra Light の呼び出しによって変更される可能性があるため、値をしばらく保持したい場合は、そのコピーを作成してください。

GetPrimaryKey メソッド

テーブルのプライマリキーを取得します。

構文

```
public virtual ULIndexSchema * GetPrimaryKey()
```

戻り値

テーブルのプライマリキーの ULIndexSchema オブジェクト。

GetPublicationPredicate メソッド

文字列としてのパブリケーション述部を取得します。

構文

```
public virtual const char * GetPublicationPredicate(  
    const char * pubName  
)
```

パラメータ

- **pubName** パブリケーションの名前

戻り値

指定されたパブリケーションのパブリケーション述部文字列。この値は静的バッファを指します。静的バッファの内容は、それ以降の **Ultra Light** の呼び出しによって変更される可能性があるため、値を保持したい場合は、そのコピーを作成してください。

GetTableSyncType メソッド

テーブルの同期タイプを取得します。

構文

```
public virtual ul_table_sync_type GetTableSyncType()
```

戻り値

テーブル同期タイプ。

備考

このメソッドは、テーブルが同期に参加する方法、および **CREATE TABLE** 文の **SYNCHRONIZE** 制約句によってテーブルを作成するときの定義方法を示します。

参照

- [ul_table_sync_type 列挙体 \[Ultra Light C++\]229 ページ](#)

InPublication メソッド

テーブルが、指定されたパブリケーションに含まれているかどうかをチェックします。

構文

```
public virtual bool InPublication(const char * pubName)
```

パラメータ

- **pubName** パブリケーションの名前。

戻り値

テーブルがパブリケーションに含まれている場合は `true`、含まれていない場合は `false`。

IsColumnInIndex メソッド

カラムが、指定されたインデックスに含まれているかどうかをチェックします。

構文

```
public virtual bool IsColumnInIndex(  
    ul_column_num cid,  
    const char * indexName  
)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。
- **indexName** インデックス名。

戻り値

カラムがインデックスに含まれている場合は `true`、含まれていない場合は `false`。

IsColumnNullable メソッド

指定されたカラムが NULL 入力可であるかどうかをチェックします。

構文

```
public virtual bool IsColumnNullable(ul_column_num cid)
```

パラメータ

- **cid** 1 から始まるカラムの順序数。

戻り値

カラムが NULL 入力可能である場合は `true`、そうでない場合は `false` を返します。

ul_column_default_type 列挙

カラムのデフォルト型を識別します。

構文

```
public enum ul_column_default_type
```

メンバー

メンバー名	説明
ul_column_default_none	カラムにデフォルト値はありません。
ul_column_default_autoincrement	カラムのデフォルトは AUTOINCREMENT です。
ul_column_default_global_autoincrement	カラムのデフォルトは GLOBAL AUTOINCREMENT です。
ul_column_default_current_timestamp	カラムのデフォルトは CURRENT TIMESTAMP です。
ul_column_default_current_utc_timestamp	カラムのデフォルトは CURRENT UTC TIMESTAMP です。
ul_column_default_current_time	カラムのデフォルトは CURRENT TIME です。
ul_column_default_current_date	カラムのデフォルトは CURRENT DATE です。
ul_column_default_newid	カラムのデフォルトは NEWID() です。
ul_column_default_other	カラムのデフォルトはユーザ指定の定数です。

参照

- [ULTableSchema.GetColumnDefaultType メソッド \[Ultra Light C++\]221 ページ](#)

ul_column_name_type 列挙

結果セットの記述時にカラムの名前を取得する方法を制御する値を指定します。

構文

```
public enum ul_column_name_type
```

メンバー

メンバー名	説明
ul_name_type_sql	SELECT 文の場合は、エイリアスまたは関連名を返します。 テーブルの場合は、カラム名を返します。

メンバー名	説明
ul_name_type_sql_column_only	SELECT 文の場合は、エイリアスまたは相関名を返し、指定されたテーブルの名前を除外します。 テーブルの場合は、カラム名を返します。
ul_name_type_base_table	確認できる場合は、基本となるテーブルの名前を返します。 このテーブルがデータベーススキーマに存在しない場合は、空の文字列を返します。
ul_name_type_base_column	確認できる場合は、基本となるカラムの名前を返します。 このカラムがデータベーススキーマに存在しない場合は、空の文字列を返します。
ul_name_type_qualified	ULResultSetSchema.GetColumnName メソッドと一緒に使用したときに、基本となる修飾カラム名を確認できる場合は、このカラム名を返します。 返される名前は、次の値のいずれかであり、この順序で確認されます。 1. 表現される相関テーブル 2. 表現されるテーブルのカラムの名前 3. カラムのエイリアス名 4. 空の文字列

メンバー名	説明
ul_name_type_base	GetColumnName メソッドと一緒に使用したときにテーブルの名前で修飾されたカラムの名前が返されることを示します。 取得されるカラム名がクエリのベーステーブルに関連付けられている場合は、ベーステーブル名がカラムの修飾子として使用されます (つまり、<code>base_table_name.column_name</code> 値が返されます)。取得されるカラム名がクエリの関連テーブルのカラムを表している場合は、関連名がカラムの修飾子として使用されます (つまり、<code>correl_table_name.col_name</code> 値が返されます)。カラムにエイリアスがある場合は、エイリアスを使用しているカラムの修飾名が返されますが、エイリアスは、修飾名の一部ではありません。それ以外の場合は、空の文字列が返されます。

参照

- [ULResultSetSchema.GetColumnName メソッド \[Ultra Light C++\]207 ページ](#)

ul_index_flag 列挙

インデックスのプロパティを識別するフラグ (ビットフィールド)。

構文

```
public enum ul_index_flag
```

メンバー

メンバー名	説明
ul_index_flag_primary_key	インデックスはプライマリキーです。
ul_index_flag_unique_key	インデックスはプライマリキーであるか、一意性制約に対して作成されたインデックスです (NULL は許可されません)。
ul_index_flag_unique_index	インデックスは UNIQUE フラグで作成されました (またはプライマリキーです)。
ul_index_flag_foreign_key	インデックスは外部キーです。
ul_index_flag_foreign_key_nullable	外部キーは NULL を許可します。

メンバー名	説明
ul_index_flag_foreign_key_check_on_commit	参照整合性チェックがコミット時に (挿入時や更新時ではなく) 実行されます。

参照

- [ULIndexSchema.GetIndexFlags メソッド \[Ultra Light C++\]157 ページ](#)

ul_table_sync_type 列挙

テーブルの同期タイプを識別します。

構文

```
public enum ul_table_sync_type
```

メンバー

メンバー名	説明
ul_table_sync_on	変更されたすべてのローが同期されます (デフォルトの動作)。 このイニシャライザは、CREATE TABLE 文の SYNCHRONIZE ON 句に対応します。
ul_table_sync_off	テーブルは同期されません。 このイニシャライザは、CREATE TABLE 文の SYNCHRONIZE OFF 句に対応します。
ul_table_sync_upload_all_rows	未変更のローを含め、常にすべてのローをアップロードします。 このイニシャライザは、CREATE TABLE 文の SYNCHRONIZE ALL 句に対応します。
ul_table_sync_download_only	変更がアップロードされることはありません。 このイニシャライザは、CREATE TABLE 文の SYNCHRONIZE DOWNLOAD 句に対応します。

参照

- [ULTableSchema.GetTableSyncType メソッド \[Ultra Light C++\]224 ページ](#)

UL_BLOB_CONTINUE 変数

ULResultSet.GetStringChunk メソッドまたは ULResultSet.GetByteChunk メソッドを使用してデータを読み込む場合に使用します。

構文

```
#define UL_BLOB_CONTINUE
```

備考

この値は、読み込むデータのチャンクが、最後のチャンクが読み込まれた位置から続いている必要があることを示します。

参照

- [ULResultSet.GetStringChunk メソッド \[Ultra Light C++\]188 ページ](#)
- [ULResultSet.GetByteChunk メソッド \[Ultra Light C++\]177 ページ](#)

ul_index_iter_start 変数

テーブル内のインデックス反復を初期化する場合に GetNextIndex メソッドで使用されます。

構文

```
#define ul_index_iter_start
```

参照

- [ULTableSchema.GetNextIndex メソッド \[Ultra Light C++\]223 ページ](#)

ul_publication_iter_start 変数

データベースでのパブリケーション反復を初期化する場合に GetNextPublication メソッドで使用されます。

構文

```
#define ul_publication_iter_start
```

参照

- [ULDatabaseSchema.GetNextPublication メソッド \[Ultra Light C++\]149 ページ](#)

ul_table_iter_start 変数

データベースでのテーブル反復を初期化する場合に GetNextTable メソッドで使用されます。

構文

```
#define ul_table_iter_start
```

参照

- [ULDatabaseSchema.GetNextTable メソッド \[Ultra Light C++\]149 ページ](#)

Ultra Light Embedded SQL API リファレンス

この項では、Embedded SQL アプリケーションで Ultra Light 機能をサポートする関数について説明します。

使用できる SQL 文の概要については、「[Embedded SQL を使用した Ultra Light C++ アプリケーション開発](#)」36 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。

ヘッダファイル

- `mlfiletransfer.h`
- `ulprotos.h`

db_fini メソッド

Ultra Light ランタイムライブラリで使用したリソースを解放します。

構文

```
unsigned short db_fini( SQLCA * sqlca );
```

戻り値

- 処理中にエラーが発生した場合は 0。エラー情報は SQLCA に設定されます。
- エラーが発生しなかった場合は 0 以外。

備考

db_fini が呼び出された後に、他の Ultra Light ライブラリ呼び出しをしたり、Embedded SQL コマンドを実行したりしないでください。

使用する SQLCA ごとに 1 回ずつ db_fini を呼び出します。

参照

- 「[db_init メソッド](#)」231 ページ

db_init メソッド

Ultra Light ランタイムライブラリを初期化します。

構文

```
unsigned short db_init( SQLCA * sqlca );
```

戻り値

- 処理中にエラーが発生した場合は 0 (たとえば永続ストアの初期化時)。エラー情報は SQLCA に設定されます。
- エラーが発生しなかった場合は 0 以外。Embedded SQL コマンドと関数の使用を開始できます。

備考

この関数は、他の Ultra Light ライブラリを呼び出す前、および Embedded SQL コマンドを実行する前に呼び出す必要があります。

通常は、この関数を 1 回だけ呼び出して、ヘッダファイル *sqlca.h* に定義されているグローバル変数 *sqlca* のアドレスを渡してください。アプリケーションに複数の実行パスがある場合、複数の *db_init* を呼び出すことができます。ただし、それぞれに別個の *sqlca* ポインタが必要です。この別個の SQLCA ポインタには、ユーザが定義したものを使用するか、*db_fini* で解放されたグローバル SQLCA を使用します。

マルチスレッドアプリケーションでは、各スレッドは、別個の SQLCA を獲得するために *db_init* を呼び出します。同じ SQLCA を使用する接続とトランザクションは、1 つのスレッドで続けて実行してください。

SQLCA を初期化すると、その前の ULEnable 関数呼び出しによる設定がすべてリセットされます。SQLCA を再初期化する場合は、アプリケーションに必要な ULEnable 関数をすべて発行する必要があります。

参照

- [「db_fini メソッド」 231 ページ](#)

MLFTEnableRsaE2ee メソッド

RSA エンドツーエンド暗号化機能を指定できるようにします。

構文

```
public void MLFTEnableRsaE2ee(ml_file_transfer_info * info)
```

パラメータ

- **info** ファイル転送情報が格納された構造体。

参照

- [ml_file_transfer_info structure \[Ultra Light Embedded SQL\]270 ページ](#)

MLFTEnableRsaEncryption メソッド

RSA 暗号化機能を指定できるようにします。

構文

```
public void MLFTEnableRsaEncryption(ml_file_transfer_info * info)
```

パラメータ

- **info** ファイル転送情報が格納された構造体。

参照

- [ml_file_transfer_info structure \[Ultra Light Embedded SQL\]270 ページ](#)

MLFTEnableRsaFipsE2ee メソッド

RSAFIPS エンドツーエンド暗号化機能を指定できるようにします。

構文

```
public void MLFTEnableRsaFipsE2ee(ml_file_transfer_info * info)
```

パラメータ

- **info** ファイル転送情報が格納された構造体。

参照

- [ml_file_transfer_info structure \[Ultra Light Embedded SQL\]270 ページ](#)

MLFTEnableRsaFipsEncryption メソッド

RSAFIPS 暗号化機能を指定できるようにします。

構文

```
public void MLFTEnableRsaFipsEncryption(ml_file_transfer_info * info)
```

パラメータ

- **info** ファイル転送情報が格納された構造体。

参照

- [ml_file_transfer_info structure \[Ultra Light Embedded SQL\]270 ページ](#)

MLFTEnableZlibCompression メソッド

ZLIB 暗号化機能を指定できるようにします。

構文

```
public void MLFTEnableZlibCompression(ml_file_transfer_info * info)
```

パラメータ

- **info** ファイル転送情報が格納された構造体。

参照

- [ml_file_transfer_info structure \[Ultra Light Embedded SQL\]270 ページ](#)

MLFileDownload メソッド

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

構文

```
public bool MLFileDownload(ml_file_transfer_info * info)
```

パラメータ

- **info** ファイル転送情報が格納された構造体。

備考

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

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

参照

- [ml_file_transfer_info structure \[Ultra Light Embedded SQL\]270 ページ](#)

例

次の例に、MLFileDownload メソッドの使用方法を示します。

```
ml_file_transfer_info info;  
MLInitFileTransferInfo( &info );  
MLFTEnableZlibCompression( &info );  
info.filename = "myfile";  
info.username = "user1";  
info.password = "pwd";  
info.version = "ver1";  
info.stream = "HTTP";  
info.stream_parms = "host=myhost.com;compression=zlib";  
if ( ! MLFileDownload( &info ) ) {  
    // file download failed
```



```
}  
MLFiniParamFileTransferInfo( &info );
```

MLFileUpload メソッド

Mobile Link インタフェースを使用して、Mobile Link サーバからファイルをアップロードします。

構文

```
public bool MLFileUpload(ml_file_transfer_info * info)
```

パラメータ

- **info** ファイル転送情報が格納された構造体。

備考

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

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

参照

- [ml_file_transfer_info structure \[Ultra Light Embedded SQL\]270 ページ](#)

例

次の例に、MLFileUpload メソッドの使用方法を示します。

```
ml_file_transfer_info info;  
MLInitFileTransferInfo( &info );  
MLFTEnableZlibCompression( &info );  
info.filename = "myfile";  
info.username = "user1";  
info.password = "pwd";  
info.version = "ver1";  
info.stream = "HTTP";  
info.stream_parms = "host=myhost.com;compression=zlib";  
if( ! MLFileUpload( &info ) ) {  
    // file upload failed  
}  
MLFiniParamFileTransferInfo( &info );
```

MLFiniParamFileTransferInfo メソッド

ml_file_transfer_info 構造体が初期化されている場合、この構造体で割り当てられているリソースをファイナライズします。

構文

```
public void MLFinifileTransferInfo(ml_file_transfer_info * info)
```

パラメータ

- **info** ファイル転送情報が格納された構造体。

備考

このメソッドは、ファイルのアップロードまたはダウンロードが完了した後で呼び出してください。

参照

- [ml_file_transfer_info structure \[Ultra Light Embedded SQL\]270 ページ](#)

MLInitFileTransferInfo メソッド

ml_file_transfer_info 構造体を初期化します。

構文

```
public bool MLInitFileTransferInfo(ml_file_transfer_info * info)
```

パラメータ

- **info** ファイル転送情報が格納された構造体。

備考

このメソッドは、ファイルのアップロードまたはダウンロードを開始する前に呼び出してください。

参照

- [ml_file_transfer_info structure \[Ultra Light Embedded SQL\]270 ページ](#)

ULCancelGetNotification メソッド

指定された名前に一致する、すべてのキューに登録されている保留中の取得通知コールをキャンセルします。

構文

```
public ul_u_long ULCancelGetNotification(  
    SQLCA * sqlca,  
    char const * queue_name  
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。

- **queue_name** キューの名前。

戻り値

影響を受けるキューの数 (ブロックされた読み込み数とは異なります)。

ULChangeEncryptionKey メソッド

Ultra Light データベースの暗号化キーを変更します。

構文

```
public ul_bool ULChangeEncryptionKey (  
    SQLCA * sqlca,  
    char const * new_key  
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **new_key** 新しい暗号化キー。

備考

このメソッドを呼び出すアプリケーションでは、データベースが同期されていること、または信頼できるバックアップコピーが作成されていることを、先に確認しておく必要があります。このメソッドは、実行を継続する必要がある操作であるため、信頼できるバックアップがあることが重要です。データベース暗号化キーを変更すると、まずデータベースのすべてのローは古いキーを使用して復号され、次に新しいキーを使用して再度暗号化されて、書き込まれます。この操作は元に戻せません。暗号化変更処理が完了しなかった場合、データベースは無効な状態のままになり、再度アクセスすることはできなくなります。

ULCheckpoint メソッド

チェックポイント操作を実行し、保留中になっているコミット済みトランザクションをデータベースにフラッシュします。

構文

```
public ul_ret_void ULCheckpoint (SQLCA * sqlca)
```

パラメータ

- **sqlca** SQLCA へのポインタ。

備考

このメソッドを呼び出しても、現在のトランザクションすべてがコミットされるわけではありません。このメソッドは、パフォーマンスを向上させるために後回しされた自動トランザクションチェックポイントとともに使用されます。

このメソッドを使用すると、保留中のコミット済みトランザクションがすべてデータベースの記憶領域に書き込まれることが保証されます。

ULCountUploadRows メソッド

同期するためにアップロードする必要があるローの数を数えます。

構文

```
public ul_u_long ULCountUploadRows(  
    SQLCA * sqlca,  
    char const * pub_list,  
    ul_u_long threshold  
)
```

パラメータ

- **sqlca** SQL へのポインタ。
- **pub_list** チェック対象となるパブリケーションのカンマ区切りのリストを含む文字列。空の文字列 (UL_SYNC_ALL マクロ) は、「非同期」とマーク付けされたものを除くすべてのテーブルを表します。アスタリスクのみの文字列 (UL_SYNC_ALL_PUBS マクロ) は、いずれかのパブリケーションで参照されているすべてのテーブルを表します。一部のテーブルは、どのパブリケーションの一部でもないため、pub_list string が "*" の場合は含まれません。
- **threshold** カウントするローの最大数を判断します。呼び出しの所要時間を制限します。threshold が 0 の場合、制限はありません (つまり、同期する必要のあるすべてのローがメソッドによってカウントされます)。また、threshold が 1 の場合、同期の必要なローがあるかどうかを簡単に判別するために使用できます。

戻り値

指定されたパブリケーションのセットまたはデータベース全体のいずれかで、同期を必要とするローの数。

備考

このメソッドを使用すると、ユーザは同期、または自動バックグラウンド同期が行われるタイミングの決定を求められます。

次の呼び出しでは、データベース全体をチェックして、同期させるローの総数を確認します。

```
count = ULCountUploadRows( sqlca, UL_SYNC_ALL, 0 );
```

次の呼び出しでは、最大 1000 のローに対して PUB1 パブリケーションと PUB2 パブリケーションがチェックされます。

```
count = ULCountUploadRows( sqlca, UL_TEXT("PUB1,PUB2"), 1000 );
```

次の呼び出しでは、PUB1 パブリケーションと PUB2 パブリケーションで同期させる必要のあるローがあるかどうかチェックされます。

```
count = ULCountUploadRows( sqlca, UL_TEXT("PUB1,PUB2"), 1 );
```

ULCreateDatabase メソッド

Ultra Light データベースを作成します。

構文

```
public ul_bool ULCreateDatabase (
    SQLCA * sqlca,
    char const * connect_parms,
    char const * create_parms,
    void * reserved
)
```

パラメータ

- **sqlca** 初期化済み SQLCA へのポインタ。
- **connect_parms** セミコロンで区切った接続パラメータ文字列で、キーワード=値のペアで設定されます。接続文字列には、データベースの名前を含める必要があります。ここに含まれるパラメータは、データベースの接続時に指定されるパラメータセットと同じです。
- **create_parms** セミコロンで区切った作成パラメータ文字列です。page_size=2048、obfuscate=yes などのように、キーワード=値のペアで設定されます。
- **reserved** このパラメータは、今後の使用のために予約されています。

戻り値

データベースが正常に作成された場合は **ul_true**。それ以外の場合は **ul_false** を返します。通常、**ul_false** は、無効なファイル名やアクセスの拒否によって発生します。

備考

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

connect_parms パラメータは、データベースがアクセスされるたびに適用される接続パラメータのリストです。ファイル名、ユーザ ID、パスワード、オプションの暗号化キーは、その例です。

create_parms パラメータはパラメータのリストで、データベースの作成時にのみ意味を持ちます。難読化、ページサイズ、時間形式や日付形式は、その例です。

アプリケーションでこのメソッドを呼び出すことができるのは、SQLCA の初期化後です。

次のコードは、ULCreateDatabase メソッドを使用して、ファイル *C:\myfile.udb* に Ultra Light データベースを作成する方法を示します。

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

参照

- [「Ultra Light 接続パラメータ」『Ultra Light データベース管理とリファレンス』](#)
- [「Ultra Light 作成パラメータ」『Ultra Light データベース管理とリファレンス』](#)

ULCreateNotificationQueue メソッド

この接続のイベント通知キューを作成します。

構文

```
public ul_bool ULCreateNotificationQueue (  
    SQLCA * sqlca,  
    char const * name,  
    char const * parameters  
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **name** 新しいキューの名前。
- **parameters** 現在使われていません。NULL に設定されます。

戻り値

成功した場合は true、失敗した場合は false。

備考

キュー名は、接続ごとにスコープされるため、別々の接続で同じ名前を持つキューを作成できません。イベント通知が送信されると、データベース内で一致する名前を持つすべてのキューが、個別のインスタンスの通知を受け取ります。名前では、大文字と小文字が区別されません。ULRegisterForEvent メソッドを呼び出したときに、キューが指定されていない場合は、接続ごとにデフォルトのキューが作成されます。その名前がすでに存在する場合や有効でない場合は、エラーが発生して呼び出しが失敗します。

ULDeclareEvent メソッド

登録およびトリガされるイベントを宣言します。

構文

```
public ul_bool ULDeclareEvent (SQLCA * sqlca, char const * event_name)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **event_name** 新しいユーザ定義イベントの名前。

戻り値

イベントが正常に宣言された場合は **true**。正常に宣言されず、名前がすでに使用されているか無効な場合は **false**。

備考

Ultra Light では、データベースまたは環境での操作によってトリガされるシステムイベントの一部が事前に定義されています。この関数は、ユーザ定義イベントを宣言します。ユーザ定義イベントは、**ULTriggerEvent** メソッドでトリガされます。イベント名は、ユニークにする必要があります。名前では、大文字と小文字が区別されません。

参照

- [ULTriggerEvent メソッド \[Ultra Light Embedded SQL\]263 ページ](#)

ULDeleteAllRows メソッド

テーブルからすべてのローを削除します。

構文

```
public ul_ret_void ULDeleteAllRows (SQLCA * sqlca, ul_table_num number)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **number** トランケートするテーブルの ID。

戻り値

成功した場合は **true**、失敗した場合は **false**。たとえばテーブルが開いていない場合や、SQL エラーが発生した場合などです。

備考

アプリケーションによっては、テーブル内のローをすべて削除してから、新しいデータセットをテーブルにダウンロードする方が便利ことがあります。接続で **stop sync** プロパティが設定されている場合は、削除されたローが同期されません。

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、別の接続が **DeleteAllRows** メソッドを呼び出した後にロールバックを行った場合は、その接続からのコミットされていない削除は削除されません。

インデックスを使用しないでこのテーブルを開いた場合、テーブルは読み込み専用とみなされ、データを削除できません。

ULDestroyNotificationQueue メソッド

指定されたイベント通知キューを破棄します。

構文

```
public ul_bool ULDestroyNotificationQueue (  
    SQLCA * sqlca,  
    char const * name  
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **name** 破棄するキューの名前。

戻り値

成功した場合は true、失敗した場合は false。

備考

キューに未読の通知がある場合は、警告が表示されます。未読通知は破棄されます。接続のデフォルトのイベントキューが作成されている場合、接続が閉じると破棄されます。

ULEnableAesDBEncryption メソッド

AES データベース暗号化を有効にします。

構文

```
public ul_ret_void ULEnableAesDBEncryption(SQLCA * sqlca)
```

パラメータ

- **sqlca** 初期化済み SQLCA へのポインタ。

備考

このメソッドは、C++ API アプリケーションと Embedded SQL アプリケーションで使用できます。この関数を呼び出してから `ULInitDatabaseManager` メソッドを呼び出すようにしてください。

注意

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

ULEnableAesFipsDBEncryption メソッド

FIPS 140-2 認定 AES データベース暗号化を有効にします。

構文

```
public ul_ret_void ULEnableAesFipsDBEncryption(SQLCA * sqlca)
```

パラメータ

- **sqlca** 初期化済み SQLCA へのポインタ。

備考**注意**

このメソッドを呼び出すと、適切なルーチンがアプリケーションにインクルードされ、アプリケーションのコードサイズがその分だけ増加します。

このメソッドは、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。このメソッドを呼び出してから **Synchronize** メソッドを呼び出すようにしてください。同期タイプを有効にする呼び出しを実行する前に同期しようとする、SQLE_METHOD_CANNOT_BE_CALLED エラーが発生します。

注意

別途ライセンスが必要な必須コンポーネント

FIPS 認定の暗号化には別のライセンスが必要です。強力な暗号化テクノロジーはすべて、輸出規制対象品目です。

「別途ライセンスが必要なコンポーネント」『SQL Anywhere 16 紹介』を参照してください。

参照

- **ULEnableAesDBEncryption** メソッド [Ultra Light Embedded SQL]242 ページ

ULEnableHttpSynchronization メソッド

HTTP 同期を有効にします。

構文

```
public ul_ret_void ULEnableHttpSynchronization(SQLCA * sqlca)
```

パラメータ

- **sqlca** SQLCA へのポインタ。

備考

このメソッドは、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。このメソッドを呼び出してから **Synchronize** メソッドを呼び出すようにしてください。同期タイプを有効にする呼び出しを実行する前に同期しようとする、SQLE_METHOD_CANNOT_BE_CALLED エラーが発生します。

ULEnableRsaE2ee メソッド

RSA エンドツーエンド暗号化を有効にします。

構文

```
public ul_ret_void ULEnableRsaE2ee(SQLCA * sqlca)
```

パラメータ

- **sqlca** SQLCA へのポインタ。

ULEnableRsaFipsE2ee メソッド

FIPS 140-2 認定 RSA エンドツーエンド暗号化を有効にします。

構文

```
public ul_ret_void ULEnableRsaFipsE2ee(SQLCA * sqlca)
```

パラメータ

- **sqlca** SQLCA へのポインタ。

ULEnableRsaFipsSyncEncryption メソッド

SSL ストリームまたは TLS ストリームの RSA FIPS 暗号化を有効にします。

構文

```
public ul_ret_void ULEnableRsaFipsSyncEncryption(SQLCA * sqlca)
```

パラメータ

- **sqlca** SQLCA へのポインタ。

備考

これは、ストリームパラメータを TLS または HTTPS に設定するときに必要です。

このメソッドは、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。このメソッドを呼び出してから **Synchronize** メソッドを呼び出すようにしてください。同期タイプを有効にする呼び出しを実行する前に同期しようとする、**SQL_METHOD_CANNOT_BE_CALLED** エラーが発生します。

参照

- [ULEnableRsaSyncEncryption メソッド \[Ultra Light Embedded SQL\]244 ページ](#)

ULEnableRsaSyncEncryption メソッド

SSL ストリームまたは TLS ストリームの RSA 暗号化を有効にします。

構文

```
public ul_ret_void ULEnableRsaSyncEncryption(SQLCA * sqlca)
```

パラメータ

- **sqlca** SQLCA へのポインタ。

備考

これは、ストリームパラメータを TLS または HTTPS に設定するときが必要です。

このメソッドは、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。このメソッドを呼び出してから **Synchronize** メソッドを呼び出すようにしてください。同期タイプを有効にする呼び出しを実行する前に同期しようとすると、SQLE_METHOD_CANNOT_BE_CALLED エラーが発生します。

参照

- [ULEnableRsaFipsSyncEncryption メソッド \[Ultra Light Embedded SQL\]244 ページ](#)

ULEnableTcpipSynchronization メソッド

TCP/IP 同期を有効にします。

構文

```
public ul_ret_void ULEnableTcpipSynchronization(SQLCA * sqlca)
```

パラメータ

- **sqlca** SQLCA へのポインタ。

備考

このメソッドは、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。このメソッドを呼び出してから **Synchronize** メソッドを呼び出すようにしてください。同期タイプを有効にする呼び出しを実行する前に同期しようとすると、SQLE_METHOD_CANNOT_BE_CALLED エラーが発生します。

ULEnableZlibSyncCompression メソッド

同期ストリームの ZLIB 圧縮を有効にします。

構文

```
public ul_ret_void ULEnableZlibSyncCompression(SQLCA * sqlca)
```

パラメータ

- **sqlca** 初期化済み SQLCA へのポインタ。

備考

このメソッドは、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。このメソッドを呼び出してから `Synchronize` メソッドを呼び出すようにしてください。同期タイプを有効にする呼び出しを実行する前に同期しようとすると、`SQL_METHOD_CANNOT_BE_CALLED` エラーが発生します。

ULErrorInfoInitFromSqlca メソッド

SQLCA to the `ul_error_info` オブジェクトからエラー情報をコピーします。

構文

```
public void ULErrorInfoInitFromSqlca (  
    ul_error_info * errinf,  
    SQLCA const * sqlca  
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **errinf** `ul_error_info` オブジェクト。

ULErrorInfoParameterAt メソッド

序数を指定してエラーパラメータを取得します。

構文

```
public size_t ULErrorInfoParameterAt (  
    ul_error_info const * errinf,  
    ul_u_short parmNo,  
    char * buffer,  
    size_t bufferSize  
)
```

パラメータ

- **errinf** `ul_error_info` オブジェクト。
- **parmNo** パラメータの、1 から始まる序数。
- **buffer** パラメータ文字列を受け取るバッファ。
- **bufferSize** バッファのサイズ。

戻り値

パラメータの格納に必要なサイズ (バイト単位)。または、序数が無効の場合は 0。戻り値が `bufferSize` 値より大きい場合、パラメータはトランケートされています。

ULErrorInfoParameterCount メソッド

エラーパラメータの数を取得します。

構文

```
public ul_u_short ULErrorInfoParameterCount(  
    ul_error_info const * errinf  
)
```

パラメータ

- **errinf** ul_error_info オブジェクト。

戻り値

エラーパラメータ数。

ULErrorInfoString メソッド

エラーの説明を取得します。

構文

```
public size_t ULErrorInfoString(  
    ul_error_info const * errinf,  
    char * buffer,  
    size_t bufferSize  
)
```

パラメータ

- **errinf** ul_error_info オブジェクト。
- **buffer** エラーの説明を受信するバッファ。
- **bufferSize** バッファのサイズ (バイト数)。

戻り値

文字列の格納に必要なサイズ (バイト単位)。戻り値が **len** 値より大きい場合、文字列はトランケートされています。

ULErrorInfoURL メソッド

このエラーの資料ページの URL を取得します。

構文

```
public size_t ULErrorInfoURL(  
    ul_error_info const * errinf,  
    char * buffer,  
    size_t bufferSize,  

```

```
        char const * reserved  
    )
```

パラメータ

- **errinf** ul_error_info オブジェクト。
- **buffer** URL を受け取るバッファ。
- **bufferSize** バッファのサイズ (バイト数)。
- **reserved** 今後の使用のために予約されています。

戻り値

URL の格納に必要なサイズ (バイト単位)。戻り値が **len** 値より大きい場合、URL はトランケートされています。

ULGetDatabaseID メソッド

グローバルオートインクリメントカラムに使用される現在のデータベース ID を取得します。

構文

```
public ul_u_long ULGetDatabaseID (SQLCA * sqlca)
```

パラメータ

- **sqlca** SQLCA へのポインタ。

戻り値

SetDatabaseID メソッドへの最後の呼び出しで設定された値。または、これまでに ID が設定されていない場合は **UL_INVALID_DATABASE_ID**。

ULGetDatabaseProperty メソッド

データベースプロパティの値を取得します。

構文

```
public void ULGetDatabaseProperty (  
    SQLCA * sqlca,  
    ul_database_property_id id,  
    char * dst,  
    size_t buffer_size,  
    ul_bool * null_indicator  
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。

- **id** データベースプロパティの識別子。
- **dst** プロパティの値を格納する文字配列。
- **buffer_size** 文字配列 **dst** のサイズ。
- **null_indicator** データベースパラメータが NULL であることを示すインジケータ。

ULGetErrorParameter メソッド

序数のパラメータ番号を使用してエラーパラメータを取得します。

構文

```
public size_t ULGetErrorParameter(  
    SQLCA const * sqlca,  
    ul_u_long parm_num,  
    char * buffer,  
    size_t size  
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **parm_num** 序数のパラメータ番号。
- **buffer** エラーパラメータを格納するバッファへのポインタ。
- **size** バッファのサイズ (バイト数)。

戻り値

このメソッドは、指定されたバッファにコピーされる文字数を返します。

参照

- [ULGetErrorParameterCount メソッド \[Ultra Light Embedded SQL\]249 ページ](#)

ULGetErrorParameterCount メソッド

エラーパラメータの数を取得します。

構文

```
public ul_u_long ULGetErrorParameterCount(SQLCA const * sqlca)
```

パラメータ

- **sqlca** SQLCA へのポインタ。

戻り値

エラーパラメータ数。結果が 0 の場合を除き、1 からこの戻り値までの値を使用して、ULGetErrorParameter メソッドを呼び出し、対応するエラーパラメータ値を取得できます。

参照

- [ULGetErrorParameter メソッド \[Ultra Light Embedded SQL\]249 ページ](#)

ULGetIdentity メソッド

@identity の値を取得します。

構文

```
public ul_u_big ULGetIdentity(SQLCA * sqlca)
```

パラメータ

- **sqlca** SQLCA へのポインタ。

戻り値

オートインクリメントカラムまたはグローバルオートインクリメントカラムに最後に挿入された値。

ULGetLastDownloadTime メソッド

指定したパブリケーションが最後にダウンロードされた時刻を取得します。

構文

```
public ul_bool ULGetLastDownloadTime(  
    SQLCA * sqlca,  
    char const * pub_name,  
    DECL_DATETIME * value  
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **pub_name** 最終ダウンロード時間を取得するパブリケーション名を含む文字列。
- **value** 投入する DECL_DATETIME 構造体へのポインタ。たとえば、値 January 1, 1990 は、パブリケーションがまだ同期されていないことを示します。

戻り値

pub-name 値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場合は true。それ以外の場合は、false を返します。

備考

次の呼び出しでは、UL_PUB_PUB1 パブリケーションがダウンロードされた日付と時刻とともに dt 構造体が投入されます。

```
DECL_DATETIME dt;  
ret = ULGetLastDownloadTime( &sqlca, UL_TEXT("UL_PUB_PUB1"), &dt );
```

ULGetNotification メソッド

イベント通知を読み込みます。

構文

```
public ul_bool ULGetNotification(  
    SQLCA * sqlca,  
    char const * queue_name,  
    char * event_name_buf,  
    ul_length event_name_buf_len,  
    ul_u_long wait_ms  
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **queue_name** 読み取るキュー。または、デフォルト接続キューの場合は NULL。
- **event_name_buf** イベント名を保持しているバッファ。
- **event_name_buf_len** バッファのサイズ (バイト単位)。
- **wait_ms** 返す前に、待機 (ブロック) する時間 (ミリ秒単位)。

戻り値

成功した場合は true、失敗した場合は false。

備考

この呼び出しは、通知が受信されるか、または指定された待機期間が終了するまでブロックします。wait_ms parameter に UL_READ_WAIT_INFINITE を渡し、無期限に待機します。待機をキャンセルするには、指定したキューに別の通知を送信するか、ULCancelGetNotification メソッドを使用します。通知を読み込んだら、ULGetNotificationParameter メソッドを使用して、追加のパラメータを名前で取得します。

参照

- [ULCancelGetNotification メソッド \[Ultra Light Embedded SQL\]236 ページ](#)
- [ULGetNotificationParameter メソッド \[Ultra Light Embedded SQL\]251 ページ](#)

ULGetNotificationParameter メソッド

ULGetNotification メソッドによって読み込まれたイベント通知のパラメータを取得します。

構文

```
public ul_bool ULGetNotificationParameter(
    SQLCA * sqlca,
    char const * queue_name,
    char const * parameter_name,
    char * value_buf,
    ul_length value_buf_len
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **queue_name** 読み取るキュー。デフォルト接続キューの場合は NULL。
- **parameter_name** 読み込むパラメータの名前 (または "*")。
- **value_buf** パラメータ値を保持しているバッファ。
- **value_buf_len** バッファのサイズ (バイト単位)。

戻り値

成功した場合は true、失敗した場合は false。

備考

指定されたキューで最近読み込まれた通知のパラメータのみが使用可能です。パラメータは名前によって取得されます。パラメータ名を "*" と指定すると、パラメータ文字列全体が取得されます。

ULGetSyncResult メソッド

前回の同期結果を取得します。

構文

```
public ul_bool ULGetSyncResult(
    SQLCA * sqlca,
    ul_sync_result * sync_result
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **sync_result** 同期の結果を保持する ul_sync_result 構造体へのポインタ。

戻り値

成功した場合は true、失敗した場合は false。

参照

- [ul_sync_result 構造体 \[Ultra Light C および Embedded SQL データタイプ\]101 ページ](#)

ULGlobalAutoincUsage メソッド

グローバルオートインクリメントのデフォルト値を持つすべてのカラムで、デフォルト値が使用されている比率 (%) を取得します。

構文

```
public ul_u_short ULGlobalAutoincUsage (SQLCA * sqlca)
```

パラメータ

- **sqlca** SQLCA へのポインタ。

戻り値

グローバルオートインクリメントの値のカウンタによる使用済み比率 (%)。

備考

このデフォルト値を使用するカラムがデータベース内に複数含まれている場合は、すべてのカラムに対してこの値が計算され、最大値が返されます。たとえば、戻り値 99 は、少なくとも 1 つのカラムではデフォルト値が残されているが、きわめて少ないことを示します。

参照

- [ULSetDatabaseID メソッド \[Ultra Light Embedded SQL\]258 ページ](#)

ULGrantConnectTo メソッド

指定されたパスワードを持つ新しいまたは既存のユーザ ID に、Ultra Light データベースへのアクセスを許可します。

構文

```
public ul_ret_void ULGrantConnectTo (  
    SQLCA * sqlca,  
    char const * uid,  
    char const * pwd  
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **uid** ユーザ ID を保持する文字配列。
- **pwd** ユーザ ID のパスワードを保持する文字配列。

備考

このメソッドは、既存のユーザ ID を指定したときに、既存のユーザのパスワードを更新します。

参照

- [ULRevokeConnectFrom メソッド \[Ultra Light Embedded SQL\]257 ページ](#)

ULInitSyncInfo メソッド

同期情報の構造体を初期化します。

構文

```
public ul_ret_void ULInitSyncInfo(ul_sync_info * info)
```

パラメータ

- **info** 同期構造体。

ULIsSynchronizeMessage メソッド

メッセージが ActiveSync に対する Mobile Link プロバイダからの同期メッセージであるかどうかを確認し、そのメッセージを処理するコードを呼び出すことができます。

構文

```
public ul_bool ULIsSynchronizeMessage(ul_u_long number)
```

備考

同期メッセージの処理が完了したときに、ULSignalSyncIsComplete メソッドを呼び出す必要があります。

このメソッドの呼び出しを、使用しているアプリケーションの WindowProc 関数にインクルードしてください。ActiveSync を使用する Windows Mobile に適用されます。

以下のコードは、ULIsSynchronizeMessage メソッド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

- [ULSignalSyncIsComplete メソッド \[Ultra Light Embedded SQL\]261 ページ](#)

ULLibraryVersion メソッド

Ultra Light ランタイムライブラリのバージョン番号を返します。

構文

```
public char const * ULLibraryVersion(void)
```

戻り値

Ultra Light ランタイムライブラリのバージョン番号。

ULRSALibraryVersion メソッド

RSA 暗号化ライブラリのバージョン番号を返します。

構文

```
public char const * ULRSALibraryVersion(void)
```

戻り値

RSA 暗号化ライブラリのバージョン番号。

ULRegisterForEvent メソッド

イベントの通知を受け取るためのキューを登録または登録解除します。

構文

```
public ul_bool ULRegisterForEvent(  
    SQLCA * sqlca,  
    char const * event_name,  
    char const * object_name,  
    char const * queue_name,  
    ul_bool register_not_unreg  
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **event_name** 登録するシステム定義またはユーザ定義のイベント。

- **object_name** イベントを適用するオブジェクト (テーブル名など)。
- **queue_name** 接続キュー名。NULL は、デフォルトの接続キューを表します。
- **register_not_unreg** 登録する場合は true、登録解除する場合は false。

戻り値

正常に登録できた場合は true、キューまたはイベントが存在しない場合は false。

備考

キュー名が指定されていない場合は、デフォルトの接続キューが暗黙で指定され、必要に応じて作成されます。特定のシステムイベントでは、そのイベントが適用されるオブジェクト名を指定できます。たとえば、**TableModified** イベントではテーブル名を指定できます。**ULSendNotification** メソッドとは異なり、登録された特定のキューのみイベントの通知を受信します。別の接続に、同じ名前の他のキューがある場合、それらは、同様に明示的に登録されていないかぎり、通知を受信しません。

事前に定義されたシステムイベントは次のとおりです。

- **TableModified** テーブルのローが挿入、更新、または削除されたときにトリガされます。要求の影響を受けるローの数にかかわらず、要求ごとに 1 つの通知が送信されます。
object_name パラメータは、モニタするテーブルを指定します。値 "*" は、データベース内のすべてのテーブルを意味します。このイベントには、**table_name** というパラメータがあり、このパラメータの値は変更されたテーブルの名前です。
- **Commit** コミットが完了した後にトリガされます。このイベントにはパラメータはありません。
- **SyncComplete** 同期が完了した後にトリガされます。このイベントにはパラメータはありません。

ULResetLastDownloadTime メソッド

アプリケーションが以前にダウンロードされたデータを再同期するように、パブリケーションの最終ダウンロード時間をリセットします。

構文

```
public ul_ret_void ULResetLastDownloadTime (  
    SQLCA * sqlca,  
    char const * pub_list  
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **pub_list** リセットするパブリケーションのカンマ区切りのリストを含む文字列。空の文字列は、「非同期」とマーク付けされたものを除くすべてのテーブルを割り当てます。アスタリ

スキのみの文字列 ("*") は、すべてのパブリケーションを割り当てます。一部のテーブルは、どのパブリケーションの一部でもないため、`pub_list string` が "*" の場合は含まれません。

備考

次のメソッド呼び出しは、すべてのテーブルの最終ダウンロード時間をリセットします。

```
ULResetLastDownloadTime( &sqlca, UL_TEXT("*") );
```

ULRevokeConnectFrom メソッド

Ultra Light データベースからユーザ ID のアクセス権を取り消します。

構文

```
public ul_ret_void ULRevokeConnectFrom(SQLCA * sqlca, char const * uid)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **uid** 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。

ULRollbackPartialDownload メソッド

失敗した同期からの変更をロールバックします。

構文

```
public ul_ret_void ULRollbackPartialDownload(SQLCA * sqlca)
```

パラメータ

- **sqlca** SQLCA へのポインタ。

備考

同期のダウンロード時に通信エラーが発生した場合、Ultra Light ではダウンロードした変更を適用できるため、アプリケーションでは同期が中断した時点から同期を再開することができます。ダウンロードした変更が不要である (ダウンロードが中断した時点での再開を望まない) 場合は、`ULRollbackPartialDownload` メソッドを使用することで、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。

ULSendNotification メソッド

指定された名前と一致するすべてのキューに通知を送信します。

構文

```
public ul_u_long ULSendNotification(  
    SQLCA * sqlca,  
    char const * queue_name,
```

```
        char const * event_name,  
        char const * parameters  
    )
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **queue_name** 接続キュー名。NULL は、デフォルトの接続キューを表します。
- **event_name** 登録するシステム定義またはユーザ定義のイベント。
- **parameters** 現在使われていません。NULL に設定されます。

戻り値

送信済みの通知の数 (一致するキューの数)。

備考

これに含まれるのは、現在の接続におけるキューです。この呼び出しはブロックしません。特別なキュー名 "*" を使用して、すべてのキューに送信します。指定されたイベント名は、システム定義またはユーザ定義のイベントと対応する必要はありません。読み込まれたイベント名は、通知を識別するためにそのまま渡され、送信者と受信者に対してしか意味を持たないからです。

パラメータの値には、「名前=値」のペアをセミコロンで区切ったオプションリストを指定します。通知が読み込まれた後、パラメータの値が `ULGetNotificationParameter` メソッドによって読み込まれます。

参照

- [ULGetNotificationParameter メソッド \[Ultra Light Embedded SQL\]251 ページ](#)

ULSetDatabaseID メソッド

データベースの ID 番号を設定します。

構文

```
public ul_ret_void ULSetDatabaseID(SQLCA * sqlca, ul_u_long value)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **value** 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 メソッド \[Ultra Light Embedded SQL\]253 ページ](#)

ULSetDatabaseOptionString メソッド

文字列値からデータベースオプションを設定します。

構文

```
public void ULSetDatabaseOptionString (  
    SQLCA * sqlca,  
    ul_database_option_id id,  
    char const * value  
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **id** 設定するデータベースオプションの識別子。
- **value** データベースオプションの値。

ULSetDatabaseOptionULong メソッド

数値のデータベースオプションを設定します。

構文

```
public void ULSetDatabaseOptionULong (  
    SQLCA * sqlca,  
    ul_database_option_id id,  
    ul_u_long value  
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **id** 設定するデータベースオプションの識別子。
- **value** データベースオプションの値。

ULSetErrorCallback メソッド

エラーの発生時に呼び出されるようコールバックを設定します。

構文

```
public ul_ret_void ULSetErrorCallback (  
    SQLCA * sqlca,  
    ul_error_callback_fn_a callback,  
    ul_void * user_data,  
    char * buffer,  
    size_t len  
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **callback** コールバック関数。
- **user_data** コールバックに渡されるユーザコンテキスト情報。
- **buffer** コールバックが呼び出されたときにエラーパラメータを保持する、ユーザが提供するバッファ。
- **len** バッファのサイズ (バイト数)。

参照

- [「エラー処理」 22 ページ](#)

ULSetSyncInfo メソッド

指定された `ul_sync_info` 構造体に基づいて、指定された名前を使用して同期プロファイルを作成します。

構文

```
public ul_bool ULSetSyncInfo(  
    SQLCA * sqlca,  
    char const * profile_name,  
    ul_sync_info * sync_info  
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **profile_name** 同期プロファイルの名前。
- **sync_info** 同期パラメータを保持する `ul_sync_info` 構造体へのポインタ。

戻り値

成功した場合は `true`、失敗した場合は `false`。

備考

同じ名前の同期プロファイルがすでにある場合は、この同期プロファイルで置き換えられます。構造体に `NULL` ポインタを指定することによって、指定されたプロファイルが削除されます。

ULSetSynchronizationCallback メソッド

同期の実行中に呼び出されるようコールバックを設定します。

構文

```
public ul_ret_void ULSetSynchronizationCallback(  
    SQLCA * sqlca,  
    ul_sync_observer_fn callback,  
    ul_void * user_data  
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **callback** コールバック。
- **user_data** コールバックに渡されるユーザコンテキスト情報。

ULSignalSyncIsComplete メソッド

同期メッセージの処理が完了したことを示します。

構文

```
public ul_ret_void ULSignalSyncIsComplete()
```

備考

ActiveSync プロバイダで登録したアプリケーションは、同期メッセージの処理が完了したときに、WNDPROC でこのメソッドを呼び出す必要があります。

ULStartSynchronizationDelete メソッド

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

構文

```
public ul_ret_void ULStartSynchronizationDelete(SQLCA * sqlca)
```

パラメータ

- **sqlca** SQLCA へのポインタ。

戻り値

成功した場合は true、失敗した場合は false。

ULStaticFini メソッド

Embedded SQL アプリケーションで Ultra Light ランタイムのファイナライズを実行します。

構文

```
public void ULStaticFini()
```

備考

このメソッドは、アプリケーションごとに 1 回だけ呼び出してください。その後、他の Ultra Light メソッドを呼び出すことはできません。

ULStaticInit メソッド

Embedded SQL アプリケーションで Ultra Light ランタイムの初期化を実行します。

構文

```
public void ULStaticInit()
```

備考

このメソッドは、他の Ultra Light メソッドを呼び出す前に、アプリケーションごとに 1 回だけ呼び出してください。

ULStopSynchronizationDelete メソッド

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

構文

```
public ul_bool ULStopSynchronizationDelete(SQLCA * sqlca)
```

パラメータ

- **sqlca** SQLCA へのポインタ。

戻り値

成功した場合は true、失敗した場合は false。

ULSynchronize メソッド

Ultra Light アプリケーションで同期を開始します。

構文

```
public ul_ret_void ULSynchronize(SQLCA * sqlca, ul_sync_info * info)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **info** 同期パラメータを保持する ul_sync_info 構造体へのポインタ。

備考

TCP/IP または HTTP 同期では、ULSynchronize メソッドが同期を開始します。同期中のエラーで handle_error スクリプトによって処理されないものは、SQL エラーとしてレポートされます。

アプリケーションプログラムでは、このメソッドの SQLCODE 戻り値をテストする必要があります。

次の例は、データベースの同期を示しています。

```
ul_sync_info info;  
ULInitSyncInfo( &info );  
info.user_name = UL_TEXT( "user_name" );  
info.version = UL_TEXT( "test" );  
ULSynchronize( &sqlca, &info );
```

ULSynchronizeFromProfile メソッド

指定されたプロファイルとマージパラメータを使用して、データベースを同期します。

構文

```
public ul_ret_void ULSynchronizeFromProfile(  
    SQLCA * sqlca,  
    char const * profile_name,  
    char const * merge_parms,  
    ul_sync_observer_fn observer,  
    ul_void * user_data  
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **profile_name** 同期するプロファイルの名前。
- **merge_parms** 同期で使用するマージパラメータ。
- **observer** ステータス更新の送信先となる observer コールバック。
- **user_data** コールバックに渡されるユーザコンテキストデータ。

備考

このメソッドは、SYNCHRONIZE 文を実行するのと同じです。

参照

- [「SYNCHRONIZE 文 \[Ultra Light\]」](#)『Ultra Light データベース管理とリファレンス』

ULTriggerEvent メソッド

ユーザ定義のイベントをトリガして、登録されたすべてのキューに通知を送信します。

構文

```
public ul_u_long ULTriggerEvent(  
    SQLCA * sqlca,  
    char const * event_name,
```

```
        char const * parameters  
    )
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **event_name** 登録するシステム定義またはユーザ定義のイベント。
- **parameters** 現在使われていません。NULL に設定されます。

戻り値

送信済みのイベント通知の数。

備考

パラメータの値には、「名前=値」のペアをセミコロンで区切ったオプションリストを指定します。通知が読み込まれた後、パラメータの値が `ULGetNotificationParameter` メソッドによって読み込まれます。

参照

- [ULGetNotificationParameter メソッド \[Ultra Light Embedded SQL\]251 ページ](#)

ULTruncateTable メソッド

テーブルをトランケートし、STOP SYNCHRONIZATION DELETE 文を一時的にアクティブにします。

構文

```
public ul_ret_void ULTruncateTable (SQLCA * sqlca, ul_table_num number)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **number** トランケートするテーブルの ID。

戻り値

成功した場合は true、失敗した場合は false。

ULValidateDatabase メソッド

この接続のデータベースを検証します。

構文

```
public ul_bool ULValidateDatabase (  
    SQLCA * sqlca,  
    char const * start_parms,  
    ul_table_num table_id,
```

```
    ul_u_short flags,  
    ul_validate_callback_fn callback_fn,  
    void * user_data  
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **start_parms** データベースの開始に使用されるパラメータ
- **table_id** 検証する特定のテーブルの ID。
- **flags** 検証のタイプを制御するフラグ。
- **callback_fn** 検証の進行状況の情報を受け取る関数。
- **user_data** コールバックにより呼び出し元に送り返すユーザデータ。

戻り値

成功した場合は **true**、失敗した場合は **false**。

備考

このルーチンに渡されるフラグに応じて、低レベルのストアかインデックス (または両方) を検証できます。検証中に情報を受け取るには、コールバック関数を実装し、アドレスをこのルーチンに渡します。検証対象を特定のテーブルに限定するには、テーブルの名前または ID を最後のパラメータとして渡します。

flags パラメータは、次のいずれかの値の組み合わせです。

- **ULVF_TABLE**
- **ULVF_INDEX**
- **ULVF_DATABASE**
- **ULVF_EXPRESS**
- **ULVF_FULL_VALIDATE**

参照

- [ULVF_TABLE 変数 \[Ultra Light C および Embedded SQL データタイプ\]108 ページ](#)
- [ULVF_INDEX 変数 \[Ultra Light C および Embedded SQL データタイプ\]108 ページ](#)
- [ULVF_DATABASE 変数 \[Ultra Light C および Embedded SQL データタイプ\]106 ページ](#)
- [ULVF_EXPRESS 変数 \[Ultra Light C および Embedded SQL データタイプ\]106 ページ](#)
- [ULVF_FULL_VALIDATE 変数 \[Ultra Light C および Embedded SQL データタイプ\]107 ページ](#)

ULValidateDatabaseTableName メソッド

この接続のデータベースを検証します。

構文

```
public ul_bool ULValidateDatabaseTableName (  
    SQLCA * sqlca,  
    char const * start_parms,  
    char const * table_name,  
    ul_u_short flags,  
    ul_validate_callback_fn callback_fn,  
    void * user_data  
)
```

パラメータ

- **sqlca** SQLCA へのポインタ。
- **start_parms** データベースの開始に使用されるパラメータ
- **table_name** 検証する特定のテーブルの名前。
- **flags** 検証のタイプを制御するフラグ。
- **callback_fn** 検証の進行状況の情報を受け取る関数。
- **user_data** コールバックにより呼び出し元に送り返すユーザデータ。

戻り値

成功した場合は true、失敗した場合は false。

備考

このルーチンに渡されるフラグに応じて、低レベルのストアかインデックス (または両方) を検証できます。検証中に情報を受け取るには、コールバック関数を実装し、アドレスをこのルーチンに渡します。検証対象を特定のテーブルに限定するには、テーブルの名前または ID を最後のパラメータとして渡します。

flags パラメータは、次のいずれかの値の組み合わせです。

- ULVF_TABLE
- ULVF_INDEX
- ULVF_DATABASE
- ULVF_EXPRESS
- ULVF_FULL_VALIDATE

参照

- [ULVF_TABLE 変数 \[Ultra Light C および Embedded SQL データタイプ\]108 ページ](#)
- [ULVF_INDEX 変数 \[Ultra Light C および Embedded SQL データタイプ\]108 ページ](#)
- [ULVF_DATABASE 変数 \[Ultra Light C および Embedded SQL データタイプ\]106 ページ](#)
- [ULVF_EXPRESS 変数 \[Ultra Light C および Embedded SQL データタイプ\]106 ページ](#)
- [ULVF_FULL_VALIDATE 変数 \[Ultra Light C および Embedded SQL データタイプ\]107 ページ](#)

ul_database_option_id 列挙体

ユーザが設定できる可能なデータベースオプションを設定します。

構文

```
public enum ul_database_option_id
```

メンバー

メンバー名	説明
ul_option_global_database_id	グローバルデータベース ID は、unsigned long integer を使用して設定されます。
ul_option_ml_remote_id	リモート ID は文字列を使用して設定されます。
ul_option_commit_flush_timeout	データベースのコミットフラッシュタイムアウトは、ミリ秒単位で測定される時間スレッシュホルドを表す整数として設定されます。
ul_option_commit_flush_count	データベースのコミットフラッシュカウントは、コミットカウントスレッシュホルドを表す整数として設定されます。
ul_option_isolation_level	接続の独立性レベルは文字列として設定されます。 (read_committed/read_uncommitted)
ul_option_cache_allocation	データベースファイルキャッシュを変更するために設定します。 割り当てるキャッシュの量の最小から最大までのサイズの幅を、0 ～ 100 までの整数値で設定します。

備考

これらのデータベースオプションは `ULConnection.SetDatabaseOption` メソッドで使用されます。

参照

- [ULConnection.SetDatabaseOption メソッド \[Ultra Light C++\]132 ページ](#)

ul_database_property_id 列挙体

ユーザが取得できる可能なデータベースプロパティを設定します。

構文

```
public enum ul_database_property_id
```

メンバー

メンバー名	説明
ul_property_date_format	日付形式。 (date_format)
ul_property_date_order	日付順。 (date_order)
ul_property_nearest_century	基準年。 (nearest_century)
ul_property_precision	精度。 (precision)
ul_property_scale	位取り。 (scale)
ul_property_time_format	時間形式。 (time_format)
ul_property_timestamp_format	タイムスタンプ形式。 (timestamp_format)
ul_property_timestamp_increment	タイムスタンプインクリメント。 (timestamp_increment)
ul_property_name	名前。 (Name)

メンバー名	説明
ul_property_file	ファイル。 (File)
ul_property_encryption	暗号化。 (Encryption)
ul_property_global_database_id	グローバルデータベース ID。 (global_database_id)
ul_property_ml_remote_id	リモート ID。 (ml_remote_id)
ul_property_char_set	文字セット。 (CharSet)
ul_property_collation	照合順。 (Collation)
ul_property_page_size	ページサイズ。 (PageSize)
ul_property_case_sensitive	大文字と小文字の区別。 (CaseSensitive)
ul_property_conn_count	接続数。 (ConnCount)
ul_property_max_hash_size	デフォルトの最大インデックスハッシュ。 (MaxHashSize)
ul_property_checksum_level	データベースのチェックサムレベル。 (ChecksumLevel)
ul_property_checkpoint_count	データベースのチェックポイント数。 (CheckpointCount)

メンバー名	説明
ul_property_commit_flush_timeout	データベースのコミットフラッシュタイムアウト。 (commit_flush_timeout)
ul_property_commit_flush_count	データベースのコミットフラッシュカウント。 (commit_flush_count)
ul_property_isolation_level	接続の独立性レベル。 (isolation_level)
ul_property_timestamp_with_time_zone_format	タイムゾーン付きタイムスタンプ形式。 (timestamp_with_time_zone_format)
ul_property_cache_allocation	現在のデータベースサイズキャッシュのサイズを最小から最大までのパーセンテージで表したものです。

備考

これらのプロパティは `ULConnection.GetDatabaseProperty` メソッドで使用されます。

参照

- [ULConnection.GetDatabaseProperty メソッド \[Ultra Light C++\]122 ページ](#)

ml_file_transfer_info 構造体

ファイルのアップロードまたはダウンロードのパラメータが格納された構造体。

構文

```
public typedef struct ml_file_transfer_info
```

メンバー

メンバー名	タイプ	説明
auth_parms	const char *	Mobile Link イベントの認証パラメータにパラメータを渡します。 詳細については、「Additional Parameters 同期パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。
auth_status	asa_uint16	Mobile Link イベントの認証パラメータにパラメータを渡します。 詳細については、「Additional Parameters 同期パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。
auth_value	asa_uint32	カスタム Mobile Link のユーザ認証スクリプトの結果をレポートします。 Mobile Link サーバが、この情報をクライアントに提供します。 詳細については、「Authentication Value 同期パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。

メンバー名	タイプ	説明
enable_resume	bool	true に設定すると、MLFileDownload メソッドは、通信エラーまたはユーザのキャンセルによって中断した以前のダウンロードを再開します。 サーバ上のファイルがローカルの部分ファイルより新しい場合、部分ファイルが破棄され、新しいバージョンがあらためてダウンロードされます。デフォルトは true です。
error	mlft_stream_error	発生したエラーに関する情報が格納されます。
file_auth_code	asa_uint16	サーバ上の authenticate_file_transfer スクリプト (オプション) のリターンコードが格納されます。
filename	const char *	Mobile Link を実行しているサーバから転送されるファイルの名前。 Mobile Link は username サブフォルダを検索してから、デフォルトのルートフォルダを検索します。 詳細については、「-ftr mlsrv16 オプション」『Mobile Link サーバ管理』を参照してください。
local_filename	const char *	ダウンロードファイルのローカル名。 このパラメータが空の場合、ファイル名の値が使用されます。

メンバー名	タイプ	説明
local_path	const char *	ダウンロードファイルの格納先となるローカルパス。 このパラメータが空の場合 (デフォルト)、ダウンロードファイルは現在のフォルダに格納されます。 Windows Mobile では、dest_path が空の場合、ファイルはデバイスのルート (¥) フォルダに格納されます。 デスクトップコンピュータでは、dest_path が空の場合、ファイルはユーザの現在のフォルダに格納されます。
num_auth_parms	asa_uint8	Mobile Link イベントの認証パラメータに渡される認証パラメータの数。 詳細については、「Number of Authentication Parameters パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。
observer	ml_file_transfer_observer_fn	'observer' フィールドを使用することで、ファイルのダウンロードの進捗状況を確認するコールバックを実現できます。 詳細については、後述のコールバック関数の説明を参照してください。
password	const char *	Mobile Link ユーザ名のパスワード。
remote_key	const char *	Mobile Link リモートキー。

メンバー名	タイプ	説明
stream	const char *	プロトコルには、TCPIP、TLS、HTTP、HTTPS のいずれか1つを指定します。 このフィールドは必須です。 詳細については、「Stream Type 同期パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。
stream_parms	const char *	指定されたストリームのプロトコルのオプション。 詳細については、「Ultra Light のネットワークプロトコルのオプション」『Ultra Light データベース管理とリファレンス』を参照してください。
transferred_file	asa_uint16	ファイルが正常に転送された場合は1、エラーが発生した場合は0。 MLFileUpload の呼び出し時にファイルがすでに最新の状態になっていると、エラーが発生します。この関数は、false ではなく true を返します。
user_data	void *	同期 observer で使用できるようにした、アプリケーション固有の情報。 詳細については、「User Data 同期パラメータ」『Ultra Light データベース管理とリファレンス』を参照してください。
username	const char *	Mobile Link ユーザ名。 このフィールドは必須です。

メンバー名	タイプ	説明
version	const char *	Mobile Link スクリプトのバージョン。 このフィールドは必須です。

ml_file_transfer_status 構造体

ファイルのアップロードまたはダウンロードの進行中の、ステータスまたは進行状況情報が格納された構造体。

構文

```
public typedef struct ml_file_transfer_status
```

メンバー

メンバー名	タイプ	説明
bytes_transferred	asa_uint64	現時点でダウンロード済みのファイルのサイズを示します。再開されたダウンロードの場合は、以前の同期も含まれます。
file_size	asa_uint64	ダウンロード中のファイルの合計サイズ (バイト単位) を示します。
flags	asa_uint16	追加情報が格納されます。 MLFileDownload メソッドがネットワーク呼び出しをブロックしており、observer メソッドが前回呼び出されたときからダウンロードステータスが変わっていない場合は、MLFT_STATUS_FLAG_IS_BLOCKING が設定されます。
info	ml_file_transfer_info *	MLFileDownload メソッドに渡された information オブジェクトへのポインタ。 このポインタを使用して user_data パラメータにアクセスできます。

メンバー名	タイプ	説明
resumed_at_size	asa_uint64	ダウンロードの再開で使用され、現在のダウンロードの開始点を示します。
stop	asa_uint8	true に設定すると、現在のダウンロードがキャンセルされます。 enable_resume パラメータが設定された場合にかぎり、MLFileDownload メソッドを後で呼び出したときにそのダウンロードを再開できます。

mlft_stream_error 構造体

ファイルのアップロードまたはダウンロードの進行中の、ステータスまたは進行状況情報が格納された構造体。

構文

```
public typedef struct mlft_stream_error
```

メンバー

メンバー名	タイプ	説明
error_string	char	system_error_code 値のローカライズされた説明 (システムから利用できる場合)、または stream_error_code 値の追加情報。
stream_error_code	ss_error_code	特定のストリームエラー。 取り得る値のリストについては、<code>%SQLANY16%¥SDK¥Include¥sserror.h</code> ヘッダファイルの <code>ss_error_code</code> 列挙を参照してください。
system_error_code	asa_int32	システム固有のエラーコード。

参照

- [「Mobile Link 通信エラーメッセージ \(エラーコード順\)」『エラーメッセージ』](#)

mlft_stream_error_w 構造体

ファイルのアップロードまたはダウンロードの進行中の、ステータスまたは進行状況情報が格納された構造体。

構文

```
public typedef struct mlft_stream_error_w
```

メンバー

メンバー名	タイプ	説明
error_string	wchar_t	system_error_code 値のローカライズされた説明 (システムから利用できる場合)、または stream_error_code 値の追加情報。
stream_error_code	ss_error_code	特定のストリームエラー。 取り得る値のリストについては、 <code>%SQLANY12%¥SDK¥Include¥sserror.h</code> ヘッダファイルの <code>ss_error_code</code> 列挙を参照してください。
system_error_code	asa_int32	システム固有のエラーコード。

備考**注意**

この構造体プロトタイプは、`mlft_stream_error` 構造体を参照して Win32 プラットフォームで UNICODE マクロを定義するときに内部的に使用されます。通常は、Ultra Light アプリケーションを作成する場合に直接この構造体を参照することはありません。

参照

- [mlft_stream_error structure \[Ultra Light Embedded SQL\]276 ページ](#)
- [「Mobile Link 通信エラーメッセージ \(エラーコード順\)」『エラーメッセージ』](#)

MLFT_STATUS_FLAG_IS_BLOCKING 変数

ml_file_transfer_status.flags フィールドでビット配列を定義して、Mobile Link サーバからの応答を待機中、ファイル転送はブロックされていることを示します。

構文

```
#define MLFT_STATUS_FLAG_IS_BLOCKING
```

備考

これが発生した場合には、同じファイル転送進行状況メッセージが定期的に生成されます。

索引

記号

#define

Ultra Light アプリケーション, 87

10 進数 Ultra Light Embedded SQL データ型

説明, 43

16 ビット符号付き整数 Ultra Light Embedded SQL データ型

説明, 42

4 バイト浮動小数点数 Ultra Light Embedded SQL データ型

説明, 42

8 バイト浮動小数点数 Ultra Light Embedded SQL データ型

説明, 43

A

ActiveSync

Ultra Light MFC 要件, 71

Ultra Light Windows Mobile アプリケーション, 70

WindowProc 関数, 71

Windows Mobile 用 Ultra Light の同期, 70

Windows Mobile 用 Ultra Light のバージョン, 70
クラス名, 68

AES 暗号化アルゴリズム

Ultra Light Embedded SQL データベース, 56

AfterLast メソッド

ULResultSet クラス [Ultra Light C++ API], 171

Android

データベースとの接続の切断, 22

AppendByteChunk メソッド

ULResultSet クラス [Ultra Light C++ API], 171

AppendParameterByteChunk メソッド

ULPreparedStatement クラス [Ultra Light C++ API], 161

AppendParameterStringChunk メソッド

ULPreparedStatement クラス [Ultra Light C++ API], 161

AppendStringChunk メソッド

ULResultSet クラス [Ultra Light C++ API], 173

AutoCommit モード

Ultra Light C++ 開発, 20

B

BeforeFirst メソッド

ULResultSet クラス [Ultra Light C++ API], 174

BlackBerry

データベースとの接続の切断, 22

C

CancelGetNotification メソッド

ULConnection クラス [Ultra Light C++ API], 114

changeEncryptionKey メソッド

Ultra Light Embedded SQL, 57

ChangeEncryptionKey メソッド

ULConnection クラス [Ultra Light C++ API], 114

Checkpoint メソッド

ULConnection クラス [Ultra Light C++ API], 115

Clear メソッド

ULError クラス [Ultra Light C++ API], 152

CLOSE 文

Ultra Light Embedded SQL, 52

Close メソッド

ULConnection クラス [Ultra Light C++ API], 115

ULDatabaseSchema クラス [Ultra Light C++ API], 148

ULIndexSchema クラス [Ultra Light C++ API], 156

ULPreparedStatement クラス [Ultra Light C++ API], 161

ULResultSet クラス [Ultra Light C++ API], 174

ULTableSchema クラス [Ultra Light C++ API], 220

Commit メソッド

ULConnection クラス [Ultra Light C++ API], 115

Ultra Light C++ トランザクション, 20

Connection オブジェクト

Ultra Light C++, 5

CONNECT 文

Ultra Light Embedded SQL, 40

CountUploadRows メソッド

ULConnection クラス [Ultra Light C++ API], 116

CreateDatabase メソッド

ULDatabaseManager クラス [Ultra Light C++ API], 138

CreateNotificationQueue メソッド

ULConnection クラス [Ultra Light C++ API], 117

CustDB アプリケーション

Ultra Light Windows Mobile 用の構築, 66

D

DatabaseManager オブジェクト
 Ultra Light C++, 5
db_fini メソッド [Ultra Light Embedded SQL API]
 説明, 231
db_init メソッド [Ultra Light Embedded SQL API]
 説明, 231
DECL_BINARY マクロ
 Ultra Light Embedded SQL, 42
DECL_DATETIME マクロ
 Ultra Light Embedded SQL, 42
DECL_DECIMAL マクロ
 Ultra Light Embedded SQL, 42
DECL_FIXCHAR マクロ
 Ultra Light Embedded SQL, 42
DECL_VARCHAR マクロ
 Ultra Light Embedded SQL, 42
DeclareEvent メソッド
 ULConnection クラス [Ultra Light C++ API], 117
DECLARE 文
 Ultra Light Embedded SQL, 52
DeleteAllRows メソッド
 ULTable クラス [Ultra Light C++ API], 213
DeleteNamed メソッド
 ULResultSet クラス [Ultra Light C++ API], 175
Delete メソッド
 ULResultSet クラス [Ultra Light C++ API], 175
DestroyNotificationQueue メソッド
 ULConnection クラス [Ultra Light C++ API], 118
DML
 Ultra Light C++, 7
DropDatabase メソッド
 ULDatabaseManager クラス [Ultra Light C++
 API], 139
DT_LONGBINARY Ultra Light Embedded SQL
データ型
 説明, 44
DT_LONGVARCHAR Ultra Light Embedded SQL
データ型
 説明, 44

E

e2ee_public_key
 iPhone, 4
 Mac OS X, 4
E2EE プライベートキー
 iPhone, 4

 Mac OS X, 4
EnableAesDBEncryption メソッド
 ULDatabaseManager クラス [Ultra Light C++
 API], 140
EnableAesFipsDBEncryption メソッド
 ULDatabaseManager クラス [Ultra Light C++
 API], 140
EnableAllSynchronization メソッド
 ULDatabaseManager クラス [Ultra Light C++
 API], 141
EnableHttpsSynchronization メソッド
 ULDatabaseManager クラス [Ultra Light C++
 API], 141
EnableHttpSynchronization メソッド
 ULDatabaseManager クラス [Ultra Light C++
 API], 142
EnableRsaE2ee メソッド
 ULDatabaseManager クラス [Ultra Light C++
 API], 142
EnableRsaFipsE2ee メソッド
 ULDatabaseManager クラス [Ultra Light C++
 API], 142
EnableRsaFipsSyncEncryption メソッド
 ULDatabaseManager クラス [Ultra Light C++
 API], 143
EnableRsaSyncEncryption メソッド
 ULDatabaseManager クラス [Ultra Light C++
 API], 143
EnableTcpipSynchronization メソッド
 ULDatabaseManager クラス [Ultra Light C++
 API], 144
EnableTlsSynchronization メソッド
 ULDatabaseManager クラス [Ultra Light C++
 API], 144
EnableZlibSyncCompression メソッド
 ULDatabaseManager クラス [Ultra Light C++
 API], 144
EXEC SQL
 Ultra Light Embedded SQL の開発, 38
ExecuteQuery メソッド
 ULPreparedStatement クラス [Ultra Light C++
 API], 162
ExecuteScalarV メソッド
 ULConnection クラス [Ultra Light C++ API], 120
ExecuteScalar メソッド
 ULConnection クラス [Ultra Light C++ API], 118
ExecuteStatement メソッド
 ULConnection クラス [Ultra Light C++ API], 121

ULPreparedStatement クラス [Ultra Light C++ API], 162

F

FETCH 文

Ultra Light Embedded SQL シングルロックエリ,
51

Ultra Light Embedded SQL マルチロックエリ,
52

FindBegin メソッド

ULTable クラス [Ultra Light C++ API], 214

FindFirst メソッド

ULTable クラス [Ultra Light C++ API], 215

FindLast メソッド

ULTable クラス [Ultra Light C++ API], 215

FindNext メソッド

ULTable クラス [Ultra Light C++ API], 215

FindPrevious メソッド

ULTable クラス [Ultra Light C++ API], 216

find メソッド

Ultra Light C++, 18

Find メソッド

ULTable クラス [Ultra Light C++ API], 214

Finis メソッド

ULDatabaseManager クラス [Ultra Light C++ API], 145

First メソッド

ULResultSet クラス [Ultra Light C++ API], 175

G

GetBinaryLength メソッド

ULResultSet クラス [Ultra Light C++ API], 177

GetBinary メソッド

ULResultSet クラス [Ultra Light C++ API], 175

GetByteChunk メソッド

ULResultSet クラス [Ultra Light C++ API], 177

GetChildObjectCount メソッド

ULConnection クラス [Ultra Light C++ API], 122

GetColumnCount メソッド

ULIndexSchema クラス [Ultra Light C++ API],
156

ULResultSetSchema クラス [Ultra Light C++ API], 206

GetColumnDefaultType メソッド

ULTableSchema クラス [Ultra Light C++ API],
221

GetColumnDefault メソッド

ULTableSchema クラス [Ultra Light C++ API],
220

GetColumnID メソッド

ULResultSetSchema クラス [Ultra Light C++ API], 207

GetColumnName メソッド

ULIndexSchema クラス [Ultra Light C++ API],
157

ULResultSetSchema クラス [Ultra Light C++ API], 207

GetColumnPrecision メソッド

ULResultSetSchema クラス [Ultra Light C++ API], 208

GetColumnScale メソッド

ULResultSetSchema クラス [Ultra Light C++ API], 208

GetColumnSize メソッド

ULResultSetSchema クラス [Ultra Light C++ API], 208

GetColumnSQLType メソッド

ULResultSetSchema クラス [Ultra Light C++ API], 209

GetColumnType メソッド

ULResultSetSchema クラス [Ultra Light C++ API], 209

GetConnection メソッド

ULDatabaseSchema クラス [Ultra Light C++ API], 149

ULIndexSchema クラス [Ultra Light C++ API],
157

ULPreparedStatement クラス [Ultra Light C++ API], 162

ULResultSetSchema クラス [Ultra Light C++ API], 209

ULResultSet クラス [Ultra Light C++ API], 179

GetDatabasePropertyInt メソッド

ULConnection クラス [Ultra Light C++ API], 122

GetDatabaseProperty メソッド

ULConnection クラス [Ultra Light C++ API], 122

GetDatabaseSchema メソッド

ULConnection クラス [Ultra Light C++ API], 123

GetDateTime メソッド

ULResultSet クラス [Ultra Light C++ API], 179

GetDouble メソッド

ULResultSet クラス [Ultra Light C++ API], 180

GetErrorInfo メソッド

ULError クラス [Ultra Light C++ API], 152

GetFloat メソッド

- ULResultSet クラス [Ultra Light C++ API], 181
- GetGlobalAutoincPartitionSize メソッド
 - ULTableSchema クラス [Ultra Light C++ API], 221
- GetGuid メソッド
 - ULResultSet クラス [Ultra Light C++ API], 182
- GetIndexColumnID メソッド
 - ULIndexSchema クラス [Ultra Light C++ API], 157
- GetIndexCount メソッド
 - ULTableSchema クラス [Ultra Light C++ API], 222
- GetIndexFlags メソッド
 - ULIndexSchema クラス [Ultra Light C++ API], 157
- GetIndexSchema メソッド
 - ULTableSchema クラス [Ultra Light C++ API], 222
- GetIntWithType メソッド
 - ULResultSet クラス [Ultra Light C++ API], 184
- GetInt メソッド
 - ULResultSet クラス [Ultra Light C++ API], 183
- GetLastDownloadTime メソッド
 - ULConnection クラス [Ultra Light C++ API], 123
- GetLastError メソッド
 - ULConnection クラス [Ultra Light C++ API], 124
- GetLastIdentity メソッド
 - ULConnection クラス [Ultra Light C++ API], 124
- GetName メソッド
 - ULIndexSchema クラス [Ultra Light C++ API], 158
 - ULTableSchema クラス [Ultra Light C++ API], 222
- GetNextIndex メソッド
 - ULTableSchema クラス [Ultra Light C++ API], 223
- GetNextPublication メソッド
 - ULDatabaseSchema クラス [Ultra Light C++ API], 149
- GetNextTable メソッド
 - ULDatabaseSchema クラス [Ultra Light C++ API], 149
- GetNotificationParameter メソッド
 - ULConnection クラス [Ultra Light C++ API], 125
- GetNotification メソッド
 - ULConnection クラス [Ultra Light C++ API], 125
- GetOptimalIndex メソッド
 - ULTableSchema クラス [Ultra Light C++ API], 223
- GetParameterCount メソッド
 - ULError クラス [Ultra Light C++ API], 153
- ULPreparedStatement クラス [Ultra Light C++ API], 162
- GetParameterID メソッド
 - ULPreparedStatement クラス [Ultra Lite C++ API], 163
- GetParameterType メソッド
 - ULPreparedStatement クラス [Ultra Light C++ API], 163
- GetParameter メソッド
 - ULError クラス [Ultra Light C++ API], 153
- GetPlan メソッド
 - ULPreparedStatement クラス [Ultra Light C++ API], 163
- GetPrimaryKey メソッド
 - ULTableSchema クラス [Ultra Light C++ API], 223
- GetPublicationCount メソッド
 - ULDatabaseSchema クラス [Ultra Light C++ API], 150
- GetPublicationPredicate メソッド
 - ULTableSchema クラス [Ultra Light C++ API], 224
- GetReferencedIndexName メソッド
 - ULIndexSchema クラス [Ultra Light C++ API], 158
- GetReferencedTableName メソッド
 - ULIndexSchema クラス [Ultra Light C++ API], 158
- GetResultSetSchema メソッド
 - ULPreparedStatement クラス [Ultra Light C++ API], 164
 - ULResultSet クラス [Ultra Light C++ API], 186
- GetRowCount メソッド
 - ULResultSet クラス [Ultra Light C++ API], 186
- GetRowsAffectedCount メソッド
 - ULPreparedStatement クラス [Ultra Light C++ API], 164
- GetSqlca メソッド
 - ULConnection クラス [Ultra Light C++ API], 126
- GetSQLCode メソッド
 - ULError クラス [Ultra Light C++ API], 153
- GetSQLCount メソッド
 - ULError クラス [Ultra Light C++ API], 154
- GetState メソッド

ULResultSet クラス [Ultra Light C++ API], 187
GetStringChunk メソッド
ULResultSet クラス [Ultra Light C++ API], 188
GetStringLength メソッド
ULResultSet クラス [Ultra Light C++ API], 190
GetString メソッド
ULError クラス [Ultra Light C++ API], 154
ULResultSet クラス [Ultra Light C++ API], 187
GetSyncResult メソッド
ULConnection クラス [Ultra Light C++ API], 126
GetTableCount メソッド
ULDatabaseSchema クラス [Ultra Light C++ API], 150
GetTableName メソッド
ULIndexSchema クラス [Ultra Light C++ API], 159
GetTableSchema メソッド
ULDatabaseSchema クラス [Ultra Light C++ API], 150
ULTable クラス [Ultra Light C++ API], 216
GetTableSyncType メソッド
ULTableSchema クラス [Ultra Light C++ API], 224
GetURL メソッド
ULError クラス [Ultra Light C++ API], 155
GetUserPointer メソッド
ULConnection クラス [Ultra Lite C++ API], 126
GlobalAutoincUsage メソッド
ULConnection クラス [Ultra Light C++ API], 127
GrantConnectTo メソッド
ULConnection クラス [Ultra Light C++ API], 127

H

HasResultSet メソッド
ULPreparedStatement クラス [Ultra Light C++ API], 164

I

INCLUDE 文
Ultra Light SQLCA, 39
InitSyncInfo メソッド
ULConnection クラス [Ultra Light C++ API], 127
Init メソッド
ULDatabaseManager クラス [Ultra Light C++ API], 145
InPublication メソッド

ULTableSchema クラス [Ultra Light C++ API], 224
InsertBegin メソッド
ULTable クラス [Ultra Light C++ API], 217
Insert メソッド
ULTable クラス [Ultra Light C++ API], 216
iOS
Ultra Light C++ アプリケーション, 3
Ultra Light C++ 開発, 23
アプリケーション開発, 3
エラーの処理, 22
データベーススキーマ, 20
データベースの作成と接続, 5
iPhone
Ultra Light C++ のインクルードファイル, 4
Ultra Light C++ のビルド設定, 4
Ultra Light ランタイムライブラリ, 31
IsAliased メソッド
ULResultSetSchema クラス [Ultra Light C++ API], 210
IsColumnDescending メソッド
ULIndexSchema クラス [Ultra Light C++ API], 159
IsColumnInIndex メソッド
ULTableSchema クラス [Ultra Light C++ API], 225
IsColumnNullable メソッド
ULTableSchema クラス [Ultra Light C++ API], 225
IsNull メソッド
ULResultSet クラス [Ultra Light C++ API], 192
IsOK メソッド
ULError クラス [Ultra Light C++ API], 155

L

Last メソッド
ULResultSet クラス [Ultra Light C++ API], 192
libulbase.a
Ultra Light C++ 開発, 23, 31, 34
libulrsa.a
Ultra Light C++ 開発, 23, 34
libulrt.a
Ultra Light C++ 開発, 23, 31, 34
libulrt.lib
Ultra Light C++ 開発, 23
Linux
Ultra Light ランタイムライブラリ, 23

LookupBackward メソッド
 ULTable クラス [Ultra Light C++ API], 217
LookupBegin メソッド
 ULTable クラス [Ultra Light C++ API], 218
LookupForward メソッド
 ULTable クラス [Ultra Light C++ API], 218
lookup メソッド
 Ultra Light C++, 18
Lookup メソッド
 ULTable クラス [Ultra Light C++ API], 217

M

Mac OS X
 Ultra Light C++ アプリケーション, 3
 Ultra Light C++ 開発, 23
 Ultra Light C++ のインクルードファイル, 4
 Ultra Light ランタイムライブラリ, 31
makefiles
 Ultra Light Embedded SQL, 65
MFC
 Ultra Light アプリケーションでの ActiveSync
 要件, 71
MFC アプリケーション
 Windows Mobile 用 Ultra Light, 68
ml_file_transfer_info 構造体 [Ultra Light Embedded
SQL API]
 説明, 270
ml_file_transfer_status 構造体 [Ultra Light
Embedded SQL API]
 説明, 275
mlcrsa16.dll
 Ultra Light C++ 開発, 23, 26, 29
mlcrsafips16.dll
 Ultra Light C++ 開発, 23, 24, 26, 29
mlczlib16.dll
 Ultra Light C++ 開発, 23, 26, 29
MLFileDownload メソッド [Ultra Light Embedded
SQL API]
 説明, 234
mlfiletransfer.h ヘッダファイル
 Ultra Light Embedded SQL API, 231
MLFileUpload メソッド [Ultra Light Embedded
SQL API]
 説明, 235
MLFiniFileTransferInfo メソッド [Ultra Light
Embedded SQL API]
 説明, 235

mlft_stream_error_w 構造体 [Ultra Light Embedded
SQL API]
 説明, 277
mlft_stream_error 構造体 [Ultra Light Embedded
SQL API]
 説明, 276
MLFTEnableRsaE2ee メソッド [Ultra Light
Embedded SQL API]
 説明, 232
MLFTEnableRsaEncryption メソッド [Ultra Light
Embedded SQL API]
 説明, 233
MLFTEnableRsaFipsE2ee メソッド [Ultra Light
Embedded SQL API]
 説明, 233
MLFTEnableRsaFipsEncryption メソッド [Ultra
Light Embedded SQL API]
 説明, 233
MLFTEnableZlibCompression メソッド [Ultra Light
Embedded SQL API]
 説明, 233
MLInitFileTransferInfo メソッド [Ultra Light
Embedded SQL API]
 説明, 236

N

next メソッド
 Ultra Light C++ データ取得の例, 12
Next メソッド
 ULResultSet クラス [Ultra Light C++ API], 193
NULL
 Ultra Light インジケータ変数, 49
NULL で終了された TCHAR 文字列 Ultra Light
SQL データ型
 説明, 43
NULL で終了された Unicode 文字列 Ultra Light
SQL データ型
 説明, 43
NULL で終了された WCHAR 文字列 Ultra Light
SQL データ型
 説明, 43
NULL で終了された文字列 Ultra Light Embedded
SQL データ型
 説明, 43
NULL で終了されたワイド文字列 Ultra Light SQL
データ型
 説明, 43

O

observer 同期パラメータ
 Ultra Light Embedded SQL の例, 63
OpenConnection メソッド
 ULDatabaseManager クラス [Ultra Light C++ API], 145
OpenTable メソッド
 ULConnection クラス [Ultra Light C++ API], 128
OPEN 文
 Ultra Light Embedded SQL, 52

P

PEM でコード化された X509 証明書
 iPhone, 4
PEM でコード化された証明書
 Mac OS X, 4
PrepareStatement メソッド
 ULConnection クラス [Ultra Light C++ API], 128
previous メソッド
 Ultra Light C++ データ取得の例, 12
Previous メソッド
 ULResultSet クラス [Ultra Light C++ API], 193

R

RegisterForEvent メソッド
 ULConnection クラス [Ultra Light C++ API], 129
Relative メソッド
 ULResultSet クラス [Ultra Light C++ API], 193
ResetLastDownloadTime メソッド
 ULConnection クラス [Ultra Light C++ API], 130
RevokeConnectFrom メソッド
 ULConnection クラス [Ultra Light C++ API], 130
RollbackPartialDownload メソッド
 ULConnection クラス [Ultra Light C++ API], 131
Rollback メソッド
 ULConnection クラス [Ultra Light C++ API], 130
 Ultra Light C++ トランザクション, 20

S

sbgse2.dll
 Ultra Light C++ 開発, 23, 24, 26, 29
SELECT 文
 Ultra Light C++ データ取得の例, 12
 Ultra Light Embedded SQL シングルロー, 51
SendNotification メソッド
 ULConnection クラス [Ultra Light C++ API], 131
SetBinary メソッド

 ULResultSet クラス [Ultra Light C++ API], 194
SET CONNECTION 文
 Ultra Light Embedded SQL の複数の接続, 40
SetDatabaseOptionInt メソッド
 ULConnection クラス [Ultra Light C++ API], 132
SetDatabaseOption メソッド
 ULConnection クラス [Ultra Light C++ API], 132
SetDateTime メソッド
 ULResultSet クラス [Ultra Light C++ API], 194
SetDefault メソッド
 ULResultSet クラス [Ultra Light C++ API], 196
SetDouble メソッド
 ULResultSet クラス [Ultra Light C++ API], 196
SetErrorCallback メソッド
 ULDatabaseManager クラス [Ultra Light C++ API], 146
SetFloat メソッド
 ULResultSet クラス [Ultra Light C++ API], 197
SetGuid メソッド
 ULResultSet クラス [Ultra Light C++ API], 198
SetIntWithType メソッド
 ULResultSet クラス [Ultra Light C++ API], 200
SetInt メソッド
 ULResultSet クラス [Ultra Light C++ API], 199
SetNull メソッド
 ULResultSet クラス [Ultra Light C++ API], 203
SetParameterBinary メソッド
 ULPreparedStatement クラス [Ultra Light C++ API], 165
SetParameterDateTime メソッド
 ULPreparedStatement クラス [Ultra Light C++ API], 165
SetParameterDouble メソッド
 ULPreparedStatement クラス [Ultra Light C++ API], 165
SetParameterFloat メソッド
 ULPreparedStatement クラス [Ultra Light C++ API], 166
SetParameterGuid メソッド
 ULPreparedStatement クラス [Ultra Light C++ API], 166
SetParameterIntWithType メソッド
 ULPreparedStatement クラス [Ultra Light C++ API], 167
SetParameterInt メソッド
 ULPreparedStatement クラス [Ultra Light C++ API], 167
SetParameterNull メソッド

- ULPreparedStatement クラス [Ultra Light C++ API], 168
- SetParameterString メソッド
 - ULPreparedStatement クラス [Ultra Light C++ API], 168
- SetString メソッド
 - ULResultSet クラス [Ultra Light C++ API], 203
- SetSynchronizationCallback メソッド
 - ULConnection クラス [Ultra Light C++ API], 132
- SetSyncInfo メソッド
 - ULConnection クラス [Ultra Light C++ API], 133
- SetUserPointer メソッド
 - ULConnection クラス [UltraLite C++ API], 133
- SQLCA
 - Ultra Light Embedded SQL, 39
 - Ultra Light Embedded SQL での複数の SQLCA, 40
 - Ultra Light フィールド, 39
- sqlcabc SQLCA フィールド
 - Ultra Light Embedded SQL, 39
- sqlcaid SQLCA フィールド
 - Ultra Light Embedded SQL, 39
- SQLCODE
 - Ultra Light C++ のエラー処理, 22
- sqlcode SQLCA フィールド
 - Ultra Light Embedded SQL, 39
- SQL Communications Area
 - Ultra Light Embedded SQL, 39
- sqlerrd SQLCA フィールド
 - Ultra Light Embedded SQL, 40
- sqlerrmc SQLCA フィールド
 - Ultra Light Embedded SQL, 39
- sqlerrml SQLCA フィールド
 - Ultra Light Embedded SQL, 39
- sqlerrp SQLCA フィールド
 - Ultra Light Embedded SQL, 40
- sqlpp ユーティリティ
 - Ultra Light Embedded SQL アプリケーション, 64
 - Ultra Light 使用法, 64
- sqlstate SQLCA フィールド
 - Ultra Light Embedded SQL, 40
- sqlwarn SQLCA フィールド
 - Ultra Light Embedded SQL, 40
- SQL 結果セットのナビゲーション
 - Ultra Light C++, 14
- SQL プリプロセッサユーティリティ (sqlpp)

- Ultra Light Embedded SQL アプリケーション, 64
- StartSynchronizationDelete メソッド
 - ULConnection クラス [Ultra Light C++ API], 134
- StopSynchronizationDelete メソッド
 - ULConnection クラス [Ultra Light C++ API], 134
- SynchronizeFromProfile メソッド
 - ULConnection クラス [Ultra Light C++ API], 135
- Synchronize メソッド
 - ULConnection クラス [Ultra Light C++ API], 134

T

- TriggerEvent メソッド
 - ULConnection クラス [Ultra Light C++ API], 135
- TruncateTable メソッド
 - ULTable クラス [Ultra Light C++ API], 218

U

- ul_binary 構造体 [Ultra Light C および Embedded SQL データ型 API]
 - 説明, 96
- ul_column_default_type 列挙体 [UltraLite C++ API]
 - 説明, 225
- ul_column_name_type 列挙体 [UltraLite C++ API]
 - 説明, 226
- ul_column_sql_type 列挙体 [Ultra Light C および Embedded SQL データ型 API]
 - 説明, 88
- ul_column_storage_type 列挙体 [Ultra Light C および Embedded SQL データ型 API]
 - 説明, 90
- ul_database_option_id 列挙体 [Ultra Light Embedded SQL API]
 - 説明, 267
- ul_database_property_id 列挙体 [Ultra Light Embedded SQL API]
 - 説明, 268
- ul_error_action 列挙体 [Ultra Light C および Embedded SQL データ型 API]
 - 説明, 91
- ul_error_info 構造体 [Ultra Light C および Embedded SQL データ型 API]
 - 説明, 96
- ul_index_flag 列挙体 [UltraLite C++ API]
 - 説明, 228
- UL_RS_STATE 列挙体 [Ultra Light C および Embedded SQL データ型 API]

説明, 88	ULCheckpoint メソッド [Ultra Light Embedded SQL API]
ul_stream_error 構造体 [Ultra Light C および Embedded SQL データ型 API]	説明, 237
説明, 97	ULConnection クラス [UltraLite C++ API]
ul_sync_info 構造体	CancelGetNotification メソッド, 114
説明, 58	ChangeEncryptionKey メソッド, 114
ul_sync_info 構造体 [Ultra Light C および Embedded SQL データ型 API]	Checkpoint メソッド, 115
説明, 98	Close メソッド, 115
ul_sync_result 構造体 [Ultra Light C および Embedded SQL データ型 API]	Commit メソッド, 115
説明, 101	CountUploadRows メソッド, 116
ul_sync_state 列挙体 [Ultra Light C および Embedded SQL データ型 API]	CreateNotificationQueue メソッド, 117
説明, 92	DeclareEvent メソッド, 117
ul_sync_stats_download 構造体 [Ultra Light C および Embedded SQL データ型 API]	DestroyNotificationQueue メソッド, 118
説明, 102	ExecuteScalarV メソッド, 120
ul_sync_stats_upload 構造体 [Ultra Light C および Embedded SQL データ型 API]	ExecuteScalar メソッド, 118
説明, 103	ExecuteStatement メソッド, 121
ul_sync_status 構造体	GetChildObjectCount メソッド, 122
Ultra Light Embedded SQL, 60	GetDatabasePropertyInt メソッド, 122
ul_sync_status 構造体 [Ultra Light C および Embedded SQL データ型 API]	GetDatabaseProperty メソッド, 122
説明, 103	GetDatabaseSchema メソッド, 123
ul_table_sync_type 列挙体 [UltraLite C++ API]	GetLastDownloadTime メソッド, 123
説明, 229	GetLastError メソッド, 124
UL_USE_DLL マクロ	GetLastIdentity メソッド, 124
説明, 87	GetNotificationParameter メソッド, 125
ul_validate_data 構造体 [Ultra Light C および Embedded SQL データ型 API]	GetNotification メソッド, 125
説明, 105	GetSqlca メソッド, 126
ul_validate_status_id 列挙体 [Ultra Light C および Embedded SQL データ型 API]	GetSyncResult メソッド, 126
説明, 93	GetUserPointer メソッド, 126
ULActiveSyncStream 関数	GlobalAutoincUsage メソッド, 127
Windows Mobile の使用法, 70	GrantConnectTo メソッド, 127
ulbase.lib	InitSyncInfo メソッド, 127
Ultra Light C++ 開発, 23, 24, 26, 29	OpenTable メソッド, 128
ULCancelGetNotification メソッド [Ultra Light Embedded SQL API]	PrepareStatement メソッド, 128
説明, 236	RegisterForEvent メソッド, 129
ULChangeEncryptionKey 関数 [UL ESQL]	ResetLastDownloadTime メソッド, 130
使用, 57	RevokeConnectFrom メソッド, 130
ULChangeEncryptionKey メソッド [Ultra Light Embedded SQL API]	RollbackPartialDownload メソッド, 131
説明, 237	Rollback メソッド, 130
	SendNotification メソッド, 131
	SetDatabaseOptionInt メソッド, 132
	SetDatabaseOption メソッド, 132
	SetSynchronizationCallback メソッド, 132
	SetSyncInfo メソッド, 133
	SetUserPointer メソッド, 133
	StartSynchronizationDelete メソッド, 134
	StopSynchronizationDelete メソッド, 134
	SynchronizeFromProfile メソッド, 135
	Synchronize メソッド, 134

- TriggerEvent メソッド, 135
- ValidateDatabase メソッド, 136
- 説明, 111
- ULCountUploadRows メソッド [Ultra Light Embedded SQL API]
 - 説明, 238
- ulcpp.h ヘッダファイル
 - Ultra Light C/C++ API リファレンス, 110
- ULCreateDatabase メソッド [Ultra Light Embedded SQL API]
 - 説明, 239
- ULCreateNotificationQueue メソッド [Ultra Light Embedded SQL API]
 - 説明, 240
- ULDatabaseManager クラス [UltraLite C++ API]
 - CreateDatabase メソッド, 138
 - DropDatabase メソッド, 139
 - EnableAesDBEncryption メソッド, 140
 - EnableAesFipsDBEncryption メソッド, 140
 - EnableHttpsSynchronization メソッド, 141
 - EnableHttpSynchronization メソッド, 142
 - EnableRsaE2ee メソッド, 142
 - EnableRsaFipsE2ee メソッド, 142
 - EnableRsaFipsSyncEncryption メソッド, 143
 - EnableRsaSyncEncryption メソッド, 143
 - EnableTcpipSynchronization メソッド, 144
 - EnableTlsSynchronization メソッド, 144
 - EnableZlibSyncCompression メソッド, 144
 - Fini メソッド, 145
 - Init メソッド, 145
 - OpenConnection メソッド, 145
 - SetErrorCallback メソッド, 146
 - ValidateDatabase メソッド, 147
 - 説明, 137
- ULDatabaseManager クラス [UltraLite C++ API]
 - EnableAllSynchronization メソッド, 141
- ULDatabaseSchema オブジェクト
 - Ultra Light C++ 開発, 21
- ULDatabaseSchema クラス [UltraLite C++ API]
 - Close メソッド, 148
 - GetConnection メソッド, 149
 - GetNextPublication メソッド, 149
 - GetNextTable メソッド, 149
 - GetPublicationCount メソッド, 150
 - GetTableCount メソッド, 150
 - GetTableSchema メソッド, 150
 - 説明, 148
- ULDeclareEvent メソッド [Ultra Light Embedded SQL API]
 - 説明, 240
- ULDeleteAllRows メソッド [Ultra Light Embedded SQL API]
 - 説明, 241
- ULDestroyNotificationQueue メソッド [Ultra Light Embedded SQL API]
 - 説明, 242
- ULEnableAesDBEncryption メソッド [Ultra Light Embedded SQL API]
 - 説明, 242
- ULEnableAesFipsDBEncryption メソッド [Ultra Light Embedded SQL API]
 - 説明, 242
- ULEnableHttpSynchronization メソッド [Ultra Light Embedded SQL API]
 - 説明, 243
- ULEnableRsaE2ee メソッド [Ultra Light Embedded SQL API]
 - 説明, 244
- ULEnableRsaFipsE2ee メソッド [Ultra Light Embedded SQL API]
 - 説明, 244
- ULEnableRsaFipsSyncEncryption メソッド [Ultra Light Embedded SQL API]
 - 説明, 244
- ULEnableRsaSyncEncryption メソッド [Ultra Light Embedded SQL API]
 - 説明, 244
- ULEnableTcpipSynchronization メソッド [Ultra Light Embedded SQL API]
 - 説明, 245
- ULEnableZlibSyncCompression メソッド [Ultra Light Embedded SQL API]
 - 説明, 245
- uleng16.exe
 - Ultra Light C++ 開発, 23, 29
- ULErrorInfoInitFromSqlca メソッド [Ultra Light Embedded SQL API]
 - 説明, 246
- ULErrorInfoParameterAt メソッド [Ultra Light Embedded SQL API]
 - 説明, 246
- ULErrorInfoParameterCount メソッド [Ultra Light Embedded SQL API]
 - 説明, 247

ULErrorInfoString メソッド [Ultra Light Embedded SQL API]
説明, 247

ULErrorInfoURL メソッド [Ultra Light Embedded SQL API]
説明, 247

ULError クラス [UltraLite C++ API]
Clear メソッド, 152
GetErrorInfo メソッド, 152
GetParameterCount メソッド, 153
GetParameter メソッド, 153
GetSQLCode メソッド, 153
GetSQLCount メソッド, 154
GetString メソッド, 154
GetURL メソッド, 155
IsOK メソッド, 155
ULError コンストラクタ, 151
説明, 151

ULError コンストラクタ
ULError クラス [Ultra Light C++ API], 151

ulfips16.dll
Ultra Light C++ 開発, 23, 24, 26, 29

ULGetDatabaseID メソッド [Ultra Light Embedded SQL API]
説明, 248

ULGetDatabaseProperty メソッド [Ultra Light Embedded SQL API]
説明, 248

ULGetErrorParameterCount メソッド [Ultra Light Embedded SQL API]
説明, 249

ULGetErrorParameter メソッド [Ultra Light Embedded SQL API]
説明, 249

ULGetIdentity メソッド [Ultra Light Embedded SQL API]
説明, 250

ULGetLastDownloadTime メソッド [Ultra Light Embedded SQL API]
説明, 250

ULGetNotificationParameter メソッド [Ultra Light Embedded SQL API]
説明, 251

ULGetNotification メソッド [Ultra Light Embedded SQL API]
説明, 251

ULGetSyncResult メソッド [Ultra Light Embedded SQL API]
説明, 252

ULGlobalAutoincUsage メソッド [Ultra Light Embedded SQL API]
説明, 253

ULGrantConnectTo メソッド [Ultra Light Embedded SQL API]
説明, 253

ulimp.libb
Ultra Light C++ 開発, 23, 26

ULIndexSchema オブジェクト
Ultra Light C++ 開発, 21

ULIndexSchema クラス [UltraLite C++ API]
Close メソッド, 156
GetColumnCount メソッド, 156
GetColumnName メソッド, 157
GetConnection メソッド, 157
GetIndexColumnID メソッド, 157
GetIndexFlags メソッド, 157
GetName メソッド, 158
GetReferencedIndexName メソッド, 158
GetReferencedTableName メソッド, 158
GetTableName メソッド, 159
IsColumnDescending メソッド, 159
説明, 155

ULInitSyncInfo 関数 [UL ESQL]
説明, 58

ULInitSyncInfo メソッド [Ultra Light Embedded SQL API]
説明, 254

ULIsSynchronizeMessage 関数 [UL ESQL]
ActiveSync の使用法, 70

ULIsSynchronizeMessage メソッド [Ultra Light Embedded SQL API]
説明, 254

ULLibraryVersion メソッド [Ultra Light Embedded SQL API]
説明, 255

ULPreparedStatement クラス
Ultra Light C++, 7

ULPreparedStatement クラス [UltraLite C++ API]
AppendParameterByteChunk メソッド, 161
AppendParameterStringChunk メソッド, 161
Close メソッド, 161
ExecuteQuery メソッド, 162
ExecuteStatement メソッド, 162
GetConnection メソッド, 162
GetParameterCount メソッド, 162
GetParameterID メソッド, 163

- GetParameterType メソッド, 163
- GetPlan メソッド, 163
- GetResultSetSchema メソッド, 164
- GetRowsAffectedCount メソッド, 164
- HasResultSet メソッド, 164
- SetParameterBinary メソッド, 165
- SetParameterDateTime メソッド, 165
- SetParameterDouble メソッド, 165
- SetParameterFloat メソッド, 166
- SetParameterGuid メソッド, 166
- SetParameterIntWithType メソッド, 167
- SetParameterInt メソッド, 167
- SetParameterNull メソッド, 168
- SetParameterString メソッド, 168
- 説明, 159
- ulprotos.h ヘッダファイル
 - Ultra Light Embedded SQL API, 231
- ULRegisterForEvent メソッド [Ultra Light Embedded SQL API]
 - 説明, 255
- ULResetLastDownloadTime メソッド [Ultra Light Embedded SQL API]
 - 説明, 256
- ULResultSetSchema オブジェクト
 - Ultra Light C++, 21
- ULResultSetSchema クラス [UltraLite C++ API]
 - GetColumnCount メソッド, 206
 - GetColumnID メソッド, 207
 - GetColumnName メソッド, 207
 - GetColumnPrecision メソッド, 208
 - GetColumnScale メソッド, 208
 - GetColumnSize メソッド, 208
 - GetColumnSQLType メソッド, 209
 - GetColumnType メソッド, 209
 - GetConnection メソッド, 209
 - IsAliased メソッド, 210
 - 説明, 205
- ULResultSet クラス [UltraLite C++ API]
 - AfterLast メソッド, 171
 - AppendByteChunk メソッド, 171
 - AppendStringChunk メソッド, 173
 - BeforeFirst メソッド, 174
 - Close メソッド, 174
 - DeleteNamed メソッド, 175
 - Delete メソッド, 175
 - First メソッド, 175
 - GetBinaryLength メソッド, 177
 - GetBinary メソッド, 175
 - GetByteChunk メソッド, 177
 - GetConnection メソッド, 179
 - GetDateTime メソッド, 179
 - GetDouble メソッド, 180
 - GetFloat メソッド, 181
 - GetGuid メソッド, 182
 - GetIntWithType メソッド, 184
 - GetInt メソッド, 183
 - GetResultSetSchema メソッド, 186
 - GetRowCount メソッド, 186
 - GetState メソッド, 187
 - GetStringChunk メソッド, 188
 - GetStringLength メソッド, 190
 - GetString メソッド, 187
 - IsNull メソッド, 192
 - Last メソッド, 192
 - Next メソッド, 193
 - Previous メソッド, 193
 - Relative メソッド, 193
 - SetBinary メソッド, 194
 - SetDateTime メソッド, 194
 - SetDefault メソッド, 196
 - SetDouble メソッド, 196
 - SetFloat メソッド, 197
 - SetGuid メソッド, 198
 - SetIntWithType メソッド, 200
 - SetInt メソッド, 199
 - SetNull メソッド, 203
 - SetString メソッド, 203
 - UpdateBegin メソッド, 205
 - Update メソッド, 205
 - 説明, 169
- ULRevokeConnectFrom メソッド [Ultra Light Embedded SQL API]
 - 説明, 257
- ULRollbackPartialDownload メソッド [Ultra Light Embedded SQL API]
 - 説明, 257
- ulrsa.lib
 - Ultra Light C++ 開発, 23, 24
- ULRSALibraryVersion メソッド [Ultra Light Embedded SQL API]
 - 説明, 255
- ulrt.lib
 - Ultra Light C++ 開発, 23, 24
- ulrt16.dll
 - Ultra Light C++ 開発, 23, 26
- ulrtc.lib

Ultra Light C++ 開発, 23, 29

ULSendNotification メソッド [Ultra Light Embedded SQL API]
説明, 257

ULSetDatabaseID メソッド [Ultra Light Embedded SQL API]
説明, 258

ULSetDatabaseOptionString メソッド [Ultra Light Embedded SQL API]
説明, 259

ULSetDatabaseOptionULong メソッド [Ultra Light Embedded SQL API]
説明, 259

ULSetErrorCallback メソッド [Ultra Light Embedded SQL API]
説明, 259

ULSetSynchronizationCallback メソッド [Ultra Light Embedded SQL API]
説明, 260

ULSetSyncInfo メソッド [Ultra Light Embedded SQL API]
説明, 260

ULSignalSyncIsComplete メソッド [Ultra Light Embedded SQL API]
説明, 261

ULStartSynchronizationDelete メソッド [Ultra Light Embedded SQL API]
説明, 261

ULStaticFini メソッド [Ultra Light Embedded SQL API]
説明, 261

ULStaticInit メソッド [Ultra Light Embedded SQL API]
説明, 262

ULStopSynchronizationDelete メソッド [Ultra Light Embedded SQL API]
説明, 262

ULSynchronizeFromProfile メソッド [Ultra Light Embedded SQL API]
説明, 263

ULSynchronize メソッド [Ultra Light Embedded SQL API]
説明, 262

ULTableSchema オブジェクト
Ultra Light C++ 開発, 21

ULTableSchema クラス [UltraLite C++ API]
Close メソッド, 220
GetColumnDefaultType メソッド, 221
GetColumnDefault メソッド, 220
GetGlobalAutoincPartitionSize メソッド, 221
GetIndexCount メソッド, 222
GetIndexSchema メソッド, 222
GetName メソッド, 222
GetNextIndex メソッド, 223
GetOptimalIndex メソッド, 223
GetPrimaryKey メソッド, 223
GetPublicationPredicate メソッド, 224
GetTableSyncType メソッド, 224
InPublication メソッド, 224
IsColumnInIndex メソッド, 225
IsColumnNullable メソッド, 225
説明, 219

ULTable オブジェクト
Ultra Light C++ データ取得の例, 12

ULTable クラス
Ultra Light C++ の概要, 14

ULTable クラス [UltraLite C++ API]
DeleteAllRows メソッド, 213
FindBegin メソッド, 214
FindFirst メソッド, 215
FindLast メソッド, 215
FindNext メソッド, 215
FindPrevious メソッド, 216
Find メソッド, 214
GetTableSchema メソッド, 216
InsertBegin メソッド, 217
Insert メソッド, 216
LookupBackward メソッド, 217
LookupBegin メソッド, 218
LookupForward メソッド, 218
Lookup メソッド, 217
TruncateTable メソッド, 218
説明, 210

Ultra Light エンジン
Ultra Light C++ 開発, 29

Ultra Light C/C++
INCLUDE 環境変数, 75
Ultra Light データベースの難読化, 57
アプリケーションチュートリアル, 75
アーキテクチャ, 110
サポートされるプラットフォーム, 1
説明, 3
チュートリアル, 75

Ultra Light C/C++ API リファレンス
ulcpp.h ヘッダファイル, 110

Ultra Light C/C++ 共通 API

- アルファベット順の一覧, 87
- Ultra Light C++
 - iPhone のインクルードファイル, 4
 - iPhone のビルド設定, 4
 - Mac OS X のインクルードファイル, 4
 - SQL を使用したデータ修正, 7
 - 開発, 3, 24, 26, 29, 31, 34
 - クイックスタート, 3
 - スキーマ情報へのアクセス, 20
 - データ検索, 12
 - データ修正, 7
 - データの同期, 22
 - トランザクション処理, 20
- Ultra Light C および Embedded SQL データ型 API
 - ul_binary 構造体, 96
 - ul_column_sql_type 列挙体, 88
 - ul_column_storage_type 列挙体, 90
 - ul_error_action 列挙体, 91
 - ul_error_info 構造体, 96
 - UL_RS_STATE 列挙体, 88
 - ul_stream_error 構造体, 97
 - ul_sync_info 構造体, 98
 - ul_sync_result 構造体, 101
 - ul_sync_state 列挙体, 92
 - ul_sync_stats_download 構造体, 102
 - ul_sync_stats_upload 構造体, 103
 - ul_sync_status 構造体, 103
 - ul_validate_data 構造体, 105
 - ul_validate_status_id 列挙体, 93
- Ultra Light Embedded SQL
 - CustDB アプリケーションの構築, 66
 - アプリケーションの開発, 36
 - 関数, 231
 - カーソル, 52
 - クイックスタート, 36
 - 権限, 39
 - 使用, 231
 - データのフェッチ, 51
 - 同期, 57
 - ホスト変数, 41
- Ultra Light Embedded SQL API
 - db_fini メソッド, 231
 - db_init メソッド, 231
 - ml_file_transfer_info 構造体, 270
 - ml_file_transfer_status 構造体, 275
 - MLFileDownload メソッド, 234
 - MLFileUpload メソッド, 235
 - MLFiniFileTransferInfo メソッド, 235
 - mlft_stream_error_w 構造体, 277
 - mlft_stream_error 構造体, 276
 - MLFTEnableRsaE2ee メソッド, 232
 - MLFTEnableRsaEncryption メソッド, 233
 - MLFTEnableRsaFipsE2ee メソッド, 233
 - MLFTEnableRsaFipsEncryption メソッド, 233
 - MLFTEnableZlibCompression メソッド, 233
 - MLInitFileTransferInfo メソッド, 236
 - ul_database_option_id 列挙体, 267
 - ul_database_property_id 列挙体, 268
 - ULCancelGetNotification メソッド, 236
 - ULChangeEncryptionKey メソッド, 237
 - ULCheckpoint メソッド, 237
 - ULCountUploadRows メソッド, 238
 - ULCreateDatabase メソッド, 239
 - ULCreateNotificationQueue メソッド, 240
 - ULDeclareEvent メソッド, 240
 - ULDeleteAllRows メソッド, 241
 - ULDestroyNotificationQueue メソッド, 242
 - ULEnableAesDBEncryption メソッド, 242
 - ULEnableAesFipsDBEncryption メソッド, 242
 - ULEnableHttpSynchronization メソッド, 243
 - ULEnableRsaE2ee メソッド, 244
 - ULEnableRsaFipsE2ee メソッド, 244
 - ULEnableRsaFipsSyncEncryption メソッド, 244
 - ULEnableRsaSyncEncryption メソッド, 244
 - ULEnableTcpipSynchronization メソッド, 245
 - ULEnableZlibSyncCompression メソッド, 245
 - ULErrorInfoInitFromSqlca メソッド, 246
 - ULErrorInfoParameterAt メソッド, 246
 - ULErrorInfoParameterCount メソッド, 247
 - ULErrorInfoString メソッド, 247
 - ULErrorInfoURL メソッド, 247
 - ULGetDatabaseID メソッド, 248
 - ULGetDatabaseProperty メソッド, 248
 - ULGetErrorParameterCount メソッド, 249
 - ULGetErrorParameter メソッド, 249
 - ULGetIdentity メソッド, 250
 - ULGetLastDownloadTime メソッド, 250
 - ULGetNotificationParameter メソッド, 251
 - ULGetNotification メソッド, 251
 - ULGetSyncResult メソッド, 252
 - ULGlobalAutoincUsage メソッド, 253
 - ULGrantConnectTo メソッド, 253
 - ULInitSyncInfo メソッド, 254
 - ULIsSynchronizeMessage メソッド, 254
 - ULLibraryVersion メソッド, 255
 - ulprotos.h ヘッダファイル, 231

- ULRegisterForEvent メソッド, 255
- ULResetLastDownloadTime メソッド, 256
- ULRevokeConnectFrom メソッド, 257
- ULRollbackPartialDownload メソッド, 257
- ULRSALibraryVersion メソッド, 255
- ULSendNotification メソッド, 257
- ULSetDatabaseID メソッド, 258
- ULSetDatabaseOptionString メソッド, 259
- ULSetDatabaseOptionULong メソッド, 259
- ULSetErrorCallback メソッド, 259
- ULSetSynchronizationCallback メソッド, 260
- ULSetSyncInfo メソッド, 260
- ULSignalSyncIsComplete メソッド, 261
- ULStartSynchronizationDelete メソッド, 261
- ULStaticFini メソッド, 261
- ULStaticInit メソッド, 262
- ULStopSynchronizationDelete メソッド, 262
- ULSynchronizeFromProfile メソッド, 263
- ULSynchronize メソッド, 262
- ULTriggerEvent メソッド, 263
- ULTruncateTable メソッド, 264
- ULValidateDatabaseTableName メソッド, 265
- ULValidateDatabase メソッド, 264
- Ultra Light for C/C++
 - Windows アプリケーションの構築, 75
- Ultra Light データベース
 - Embedded SQL の暗号化, 56
 - Ultra Light C++ API 情報へのアクセス, 20
 - Ultra Light C++ での接続, 5
 - Windows Mobile 用 Ultra Light, 68
- Ultra Light ランタイム
 - Ultra Light C++ ライブラリ, 23, 24, 26, 29
 - Windows Mobile ライブラリの配備, 23, 24, 26, 29
- Ultra Light ランタイムライブラリ
 - iPhone, 31
 - Linux, 23
 - Mac OS X, 31
- UltraLite C++ API
 - ul_column_default_type 列挙体, 225
 - ul_column_name_type 列挙体, 226
 - ul_index_flag 列挙体, 228
 - ul_table_sync_type 列挙体, 229
 - ULConnection クラス, 111
 - ULDatabaseManager クラス, 137
 - ULDatabaseSchema クラス, 148
 - ULError クラス, 151
 - ULIndexSchema クラス, 155

- ULPreparedStatement クラス, 159
- ULResultSetSchema クラス, 205
- ULResultSet クラス, 169
- ULTableSchema クラス, 219
- ULTable クラス, 210
- ULTriggerEvent メソッド [Ultra Light Embedded SQL API]
 - 説明, 263
- ULTruncateTable メソッド [Ultra Light Embedded SQL API]
 - 説明, 264
- ULValidateDatabaseTableName メソッド [Ultra Light Embedded SQL API]
 - 説明, 265
- ULValidateDatabase メソッド [Ultra Light Embedded SQL API]
 - 説明, 264
- UNDER_CE コンパイラディレクティブ
 - 説明, 88
- UpdateBegin メソッド
 - ULResultSet クラス [Ultra Light C++ API], 205
- Update メソッド
 - ULResultSet クラス [Ultra Light C++ API], 205

V

- ValidateDatabase メソッド
 - ULConnection クラス [Ultra Light C++ API], 136
 - ULDatabaseManager クラス [Ultra Light C++ API], 147
- Visual C++
 - Windows Mobile 用 Ultra Light の開発, 66

W

- WindowProc 関数
 - ActiveSync の使用法, 71
- Windows Mobile
 - Ultra Light アプリケーション開発の概要, 66
 - Ultra Light アプリケーションの同期, 70
 - Ultra Light クラス名, 68
 - Ultra Light 同期のメニュー制御, 73
 - Ultra Light プラットフォーム稼働条件, 66
- winsock.lib
 - Ultra Light Windows Mobile アプリケーション, 66

X

- x509 証明書

- Mac OS X, 4
- X509 証明書
 - iPhone, 4

あ

値

- Ultra Light C++ API のアクセス, 19
- アプリケーション
 - iOS の開発, 3
 - Ultra Light Embedded SQL アプリケーションの記述, 36
 - Ultra Light Embedded SQL の構築, 64
 - Ultra Light Embedded SQL のコンパイル, 64
 - Ultra Light Embedded SQL の前処理, 64
- 暗号化
 - Embedded SQL を使用する Ultra Light データベース, 56
 - iPhone, 4
 - Mac OS X, 4
 - PEM でコード化された X509 証明書, 4
 - Ultra Light Embedded SQL データベース, 56
 - Ultra Light Embedded SQL のキーの変更, 57
- アーキテクチャ
 - Ultra Light C/C++, 110

い

- 依存性
 - Ultra Light Embedded SQL, 65
- インジケータ変数
 - Ultra Light Embedded SQL, 49
 - Ultra Light NULL, 50
- インストール
 - Ultra Light Windows Mobile アプリケーション, 66
- インデックス
 - Ultra Light C++ API のスキーマ情報, 21
- インポートライブラリ
 - Ultra Light C++, 23, 24, 26, 29

え

- 永続ストレージ
 - Windows Mobile 用 Ultra Light, 68
- エミュレータ
 - Windows Mobile 用 Ultra Light, 23
- エラー
 - Ultra Light C++ API の処理, 22
 - Ultra Light Embedded SQL の通信エラー, 59

- Ultra Light SQLCODE, 39
- Ultra Light sqlcode SQLCA フィールド, 39
- Ultra Light コード, 39
- エラー処理
 - Ultra Light C++, 22
- エンドツーエンド暗号化
 - iPhone, 4
 - Mac OS X, 4

お

- オフセット
 - Ultra Light C++ 相対, 15

か

- 開発
 - Ultra Light C++, 3
- 開発ツール
 - Ultra Light Embedded SQL, 65
- 開発プラットフォーム
 - Ultra Light C++, 1
- 開発プロセス
 - Ultra Light Embedded SQL, 36
- カラム
 - Ultra Light C++ API での値の取得, 19
 - Ultra Light C++ API での値の変更, 19
- 簡易暗号化
 - Ultra Light データベースの簡易暗号化, 57
- 環境変数
 - INCLUDE, 75
- 関数
 - Ultra Light Embedded SQL, 231
- 管理
 - Ultra Light C++ トランザクション, 20
- カーソル
 - Ultra Light 位置, 53
 - Ultra Light 更新後の位置, 55
 - Ultra Light 再配置, 54
 - Ultra Light 順序, 54
 - Ultra Light 複数ローのフェッチ, 52

き

- キャスト
 - Ultra Light C++ API のデータ型, 20
- 強力な暗号化
 - Ultra Light Embedded SQL, 56

く

クエリ

- Ultra Light Embedded SQL シングルロックエリ, 51
- Ultra Light Embedded SQL マルチロックエリ, 52

クラス名

- ActiveSync 同期, 68

け

結果セット

- Ultra Light C++ API のスキーマ情報, 21
- Ultra Light C++ によるナビゲーション, 14

結果セットスキーマ

- Ultra Light C++, 13

検索

- Ultra Light C++ によるロー, 18

検索モード

- Ultra Light C++, 16

こ

更新

- Ultra Light C++ API テーブルロー, 17

更新モード

- Ultra Light C++, 16

構築

- Ultra Light Embedded SQL アプリケーション, 64

構築プロセス

- Embedded SQL アプリケーション, 64
- Ultra Light Embedded SQL アプリケーション, 64

コミット

- Ultra Light C++ トランザクション, 20
- Ultra Light Embedded SQL を使用した変更, 59

コミットされていないトランザクション

- Ultra Light Embedded SQL, 59

コンパイラ

- Windows Mobile 用 Ultra Light アプリケーション, 23, 26

コンパイラオプション

- Ultra Light C++ 開発, 23, 26
- Windows Mobile 用 Ultra Light アプリケーション, 23, 26

コンパイラディレクティブ

- Ultra Light アプリケーション, 87
- UNDER_CE, 88

コンパイル

- Ultra Light Embedded SQL アプリケーション, 64
- Windows Mobile 用 Ultra Light アプリケーション, 23, 26

さ

削除

- Ultra Light C++ API テーブルロー, 20
- サポートされるプラットフォーム
- Ultra Light C++, 1
- サンプルアプリケーション
- Ultra Light Windows Mobile 用の構築, 66

し

準備文

- Ultra Light C++, 7

す

スキーマ

- Ultra Light C++ API でのアクセス, 20
- Ultra Light C++ API のスキーマ情報, 21
- スキーマ情報へのアクセス
- Ultra Light C++ による, 20
- スクロール
- Ultra Light C++ API, 15
- スレッド
- Ultra Light C++ API マルチスレッドアプリケーション, 6
- Ultra Light Embedded SQL, 40

せ

静的ライブラリ

- Ultra Light C++ アプリケーション, 23, 24

セキュリティ

- Ultra Light Embedded SQL の暗号化キーの変更, 57
- Ultra Light Embedded SQL の難読化, 56
- Ultra Light データベースの暗号化, 56

接続

- Ultra Light Embedded SQL, 40
- Ultra Light データベース, 5

設定

- Ultra Light Embedded SQL 用の開発ツール, 65

宣言

- Ultra Light ホスト変数, 41

宣言セクション

Ultra Light Embedded SQL の宣言, 41

そ

相対オフセット

Ultra Light C++ API, 15

挿入

Ultra Light C++ API テーブルロー, 16

挿入モード

Ultra Light C++, 16

た

タイムスタンプ構造体 Ultra Light Embedded SQL
データ型

説明, 44

ターゲットプラットフォーム

Ultra Light C++, 1

ち

チュートリアル

Ultra Light C++ API, 75

つ

通信エラー

Ultra Light Embedded SQL, 59

て

ディレクティブ

Ultra Light アプリケーション, 87

データ型

Ultra Light C++ API のアクセスとキャスト, 19

Ultra Light Embedded SQL, 42

データ修正

SQL を使用した Ultra Light C++, 7

データ操作

Ultra Light C++ API, 14

データへのアクセス

Ultra Light C++ API, 14

データベーススキーマ

Ultra Light C++ API でのアクセス, 20

データベーステーブルからデータを選択

Ultra Light C++, 12

データベースファイル

Ultra Light 暗号化と難読化 (Embedded SQL), 56

Windows Mobile 用 Ultra Light, 68

テーブル

Ultra Light C++ API でのスキーマ情報, 21

と

同期

Ultra Light C++, 22

Ultra Light C++ API チュートリアル, 75

Ultra Light Embedded SQL, 57

Ultra Light Embedded SQL アプリケーションへの追加, 57

Ultra Light Embedded SQL のキャンセル, 60

Ultra Light Embedded SQL の初期同期, 59

Ultra Light Embedded SQL のモニタ, 60

Ultra Light Embedded SQL の呼び出し, 58

Ultra Light Embedded SQL の例, 58

Ultra Light Embedded SQL 変更のコミット, 59

Windows Mobile 用 Ultra Light の概要, 70

Windows Mobile 用 Ultra Light のメニュー制御,
73

同期エラー

Ultra Light Embedded SQL の通信エラー, 59

同期のキャンセル

Ultra Light Embedded SQL での, 60

動的ライブラリ

Ultra Light C++ アプリケーション, 23, 26

トラブルシューティング

Ultra Light C++ のエラー処理, 22

Ultra Light Embedded SQL を使用した同期, 59

Ultra Light SQL プリプロセッサでの参照式の
使用, 47

Ultra Light 開発, 59

トランケーション

Ultra Light FETCH, 51

トランザクション

Embedded SQL を使用した Ultra Light でのコ
ミット, 59

Ultra Light C++ 管理, 20

トランザクション処理

Ultra Light C++ 管理, 20

な

ナビゲーション

Ultra Light C++ API, 15

難読化

Embedded SQL を使用する Ultra Light データ
ベース, 57

Ultra Light Embedded SQL データベース, 56

ね

ネットワークプロトコル

Windows Mobile 用 Ultra Light, 73
ネームスペース
Ultra Light C++ の例, 75

は

バイナリ Ultra Light Embedded SQL データ型
説明, 44
配備
Windows Mobile 用 Ultra Light, 23, 24, 26, 29
インプロセスバージョンの Ultra Light, 23
パック 10 進数 Ultra Light Embedded SQL データ型
説明, 43
パフォーマンス
Ultra Light INSERT 文の使用, 59
Ultra Light で効率的にメモリを使用するための DLL の使用, 23, 24, 29
パーミッション
Ultra Light Embedded SQL, 39

ひ

ヒント
Ultra Light 開発, 59

ふ

フェッチ
Ultra Light Embedded SQL, 51
プラットフォーム
Ultra Light C++ でのサポート, 1
プラットフォーム稼働条件
Windows Mobile 用 Ultra Light, 66
プログラム構造
Ultra Light Embedded SQL, 38
プロトコル
Windows Mobile 用 Ultra Light, 73

ほ

ホスト変数
Ultra Light Embedded SQL, 41
Ultra Light Embedded SQL の式, 47
Ultra Light の使用法, 45
Ultra Light のスコープ, 45

ま

前処理
Ultra Light Embedded SQL アプリケーション,
64

Ultra Light Embedded SQL 開発ツールの設定,
65
マクロ
UL_USE_DLL, 87
Ultra Light アプリケーション, 87
マルチスレッドアプリケーション
Ultra Light C++, 6
Ultra Light Embedded SQL, 40
マルチロックエリ
Ultra Light カーソル, 52

も

文字列 Ultra Light Embedded SQL データ型
可変長, 43
固定長, 43
説明, 43
モニタのキャンセル
Ultra Light Embedded SQL での, 60
モード
Ultra Light C++, 16

ゆ

ユーザ認証
Ultra Light Embedded SQL アプリケーション,
55

ら

ライブラリ
C++ での Ultra Light のコンパイルとリンク,
23, 24, 26, 29
iOS 用 Ultra Light アプリケーション, 23, 31
Linux 用 Ultra Light アプリケーション, 23, 34
Mac OS X 用 Ultra Light アプリケーション, 23,
31
Ultra Light C++ でのリンクの例, 75
Windows Mobile 用 Ultra Light DLL, 23, 24, 29
Windows Mobile 用 Ultra Light アプリケーショ
ン, 23, 24, 26, 29
ライブラリ関数
Ultra Light Embedded SQL, 231
ランタイムライブラリ
Ultra Light C++, 23, 24, 26, 29
Ultra Light for C++, 23, 24, 26, 29, 31, 34
Windows Mobile, 87
Windows Mobile 用 Ultra Light アプリケーショ
ン, 23, 24, 26, 29

り

リンク

- Ultra Light C++ アプリケーション, 23, 24, 26, 29
- Windows Mobile 用 Ultra Light アプリケーション, 23, 24, 26, 29

る

ルックアップモード

- Ultra Light C++, 16

ろ

ロー

- C++ API による Ultra Light 削除, 20
- C++ API による Ultra Light 挿入, 16
- Ultra Light C++ API チュートリアルのアクセス, 75
- Ultra Light C++ API による更新, 17
- Ultra Light C++ 現在値のテーブルアクセス, 19
- Ultra Light C++ テーブルナビゲーション, 15

ロールバック

- Ultra Light C++ トランザクション, 20