SYBASE iAnywhere

UltraLite
C 及 C++ 编程

2009 年 2 月

11.0.1 版

版权和商标

目录

关于本手册 ix

关于 SQL Anywhere 文档 x

UltraLite for C/C++ 开发人员 1

开发嵌入式 SQL 应用程序 2
系统要求和支持的平台 3
UltraLite C++ 组件体系结构 4
了解 SQL 通信区 5
创建数据库 6

应用程序开发 7

使用 UltraLite C++ API 开发应用程序 9
使用 UltraLite 命名空间 10
连接到数据库 11
使用 SQL 访问数据 13
使用 Table API 访问数据 17
管理事务 22
访问模式信息 23
处理错误 24
验证用户 25
对数据进行加密 26
同步数据 27
编译和链接应用程序 28
使用嵌入式 SQL 开发应用程序 29
嵌入式 SQL 示例 30
初始化 SQL 通信区 32
连接到数据库 34
使用主机变量 35
读取数据 44
验证用户 48
对数据进行加密 50

向应用程序添加同步 51
构建嵌入式 SQL 应用程序 58
开发用于 Palm OS 的 UltraLite 应用程序 61
安装用于 CodeWarrior 的 UltraLite 插件 62
在 CodeWarrior 中创建 UltraLite 项目 63
将现有 CodeWarrior 项目转换为 UltraLite 应用程序 64
使用用于 CodeWarrior 的 UltraLite 插件 65
在 CodeWarrior 中构建 CustDB 示例应用程序 66
构建扩展模式应用程序 67
在 UltraLite Palm 应用程序中维护状态（不建议使用） 68
注册 Palm 创建者 ID 70
将 HotSync 同步添加到 Palm 应用程序 71
向 Palm 应用程序添加 TCP/IP、HTTP 或 HTTPS 同步 73
部署 Palm 应用程序 74
开发用于 Windows Mobile 的 UltraLite 应用程序 77
选择链接运行时库的方式 78
构建 CustDB 示例应用程序 79
存储持久数据 81
部署 Windows Mobile 应用程序 82
部署使用 ActiveSync 的应用程序 83
为应用程序指派类名 84
在 Windows Mobile 上进行同步 86
示例 eMbedded Visual C++ 项目 89

API 参考 91

UltraLite C/C++ 公共 API 参考 93
ULRegisterErrorCallback 的回调函数 94
ULRegisterSQLPassthroughCallback 的回调函数 95
MLFileTransfer 函数 97
ULCreateDatabase 函数 100
ULEnableEccSyncEncryption 函数 102
ULEnableFIPSStrongEncryption 函数 103
ULEnableHttpSynchronization 函数 104
ULEnableHttpsSynchronization 函数 105
ULEnableRsaFipsSyncEncryption 函数 106

ULEnableRsaSyncEncryption 函数 107
ULEnableStrongEncryption 函数 108
ULEnableTcpipSynchronization 函数 109
ULEnableTlsSynchronization 函数 110
ULEnableZlibSyncCompression 函数 111
ULInitDatabaseManager 112
ULInitDatabaseManagerNoSQL 113
ULRegisterErrorCallback 函数 114
ULRegisterSQLPassthroughCallback 116
ULRegisterSynchronizationCallback 118
用于 UltraLite C/C++ 应用程序的宏和编译器指令 119
UltraLite C++ 组件 API 123
ul_sql_passthrough_status 结构 125
ul_stream_error 结构 126
ul_synch_info_a 结构 127
ul_synch_info_w2 结构 129
ul_synch_result 结构 131
ul_synch_stats 结构 132
ul_synch_status 结构 133
ul_validate_data 结构 135
ULSqlca 类 136
ULSqlcaBase 类 138
ULSqlcaWrap 类 143
UltraLite_Connection 类 145
UltraLite_Connection_iface 类 148
UltraLite_Cursor_iface 类 172
UltraLite_DatabaseManager 类 179
UltraLite_DatabaseManager_iface 类 180
UltraLite_DatabaseSchema 类 183
UltraLite_DatabaseSchema_iface 类 184
UltraLite_IndexSchema 类 187
UltraLite_IndexSchema_iface 类 188
UltraLite_PreparedStatement 类 193
UltraLite_PreparedStatement_iface 类 194
UltraLite_ResultSet 类 198

UltraLite_ResultSet_iface 类 199
UltraLite_ResultSetSchema 类 200
UltraLite_RowSchema_iface 类 201
UltraLite_SQLObject_iface 类 206
UltraLite_StreamReader 类 208
UltraLite_StreamReader_iface 类 209
UltraLite_StreamWriter 类 212
UltraLite_Table 类 213
UltraLite_Table_iface 类 215
UltraLite_TableSchema 类 221
UltraLite_TableSchema_iface 类 223
ULValue 类 232
嵌入式 SQL API 参考 249
db_fini 函数 251
db_init 函数 252
db_start_database 函数 253
db_stop_database 函数 254
ULChangeEncryptionKey 函数 255
ULCheckpoint 函数 256
ULClearEncryptionKey 函数 257
ULCountUploadRows 函数 258
ULDropDatabase 函数 259
ULExecuteNextSQLPassthroughScript 260
ULExecuteSQLPassthroughScripts 261
ULGetDatabaseID 函数 262
ULGetDatabaseProperty 函数 263
ULGetErrorParameter 函数 264
ULGetErrorParameterCount 函数 265
ULGetLastDownloadTime 函数 266
GetSQLPassthroughScriptCount 267
ULGetSynchResult 函数 268
ULGlobalAutoincUsage 函数 270
ULGrantConnectTo 函数 271
ULInitSynchInfo 函数 272
ULIsSynchronizeMessage 函数 273

ULResetLastDownloadTime 函数 274
ULRetrieveEncryptionKey 函数 275
ULRevokeConnectFrom 函数 276
ULRollbackPartialDownload 函数 277
ULSaveEncryptionKey 函数 278
ULSetDatabaseID 函数 279
ULSetDatabaseOptionString 函数 280
ULSetDatabaseOptionULong 函数 281
ULSetSynchInfo 函数 282
ULSignalSyncIsComplete 函数 283
ULSynchronize 函数 284
UltraLite ODBC API 参考 285
SQLAllocHandle 函数 286
SQLBindCol 函数 287
SQLBindParameter 函数 288
SQLConnect 函数 289
SQLDescribeCol 函数 290
SQLDisconnect 函数 291
SQLEndTran 函数 292
SQLExecDirect 函数 293
SQLExecute 函数 294
SQLFetch 函数 295
SQLFetchScroll 函数 296
SQLFreeHandle 函数 297
SQLGetCursorName 函数 298
SQLGetData 函数 299
SQLGetDiagRec 函数 300
SQLGetInfo 函数 301
SQLNumResultCols 函数 302
SQLPrepare 函数 303
SQLRowCount 函数 304
SQLSetConnectionName 函数 305
SQLSetCursorName 函数 306
SQLSetSuspend 函数（不建议使用） 307
SQLSynchronize 函数 308

教程：使用 C++ API 构建应用程序 309

第 1 课：创建数据库并连接到数据库 310
第 2 课：将数据插入数据库 314
第 3 课：选择并列出表中的行 315
第 4 课：将同步添加到应用程序 317
教程的代码列表 319

术语表 323

术语表 325

索引 353

关于本手册

主题

本手册介绍 UltraLite C 和 C++ 编程接口。利用 UltraLite，可以开发数据库应用程序，并将它们部署到手持式设备、移动设备或嵌入式设备。

读者

本手册面向希望利用 UltraLite 关系数据库的性能、资源效率、稳健性和安全性进行数据存储和同步的 C 和 C++ 应用程序开发人员。

关于 SQL Anywhere 文档

完整的 SQL Anywhere 文档以四种形式提供，但所包含信息均相同。

- **HTML 帮助** 联机帮助文档包含完整的 SQL Anywhere 文档，其中包括手册和 SQL Anywhere 工具的上下文相关帮助。

  如果使用 Microsoft Windows 操作系统，则联机帮助文档以 HTML 帮助 (CHM) 格式提供。若要访问此文档，请选择 [开始] » [程序] » [**SQL Anywhere 11**] » [文档] » [联机手册]。

  管理工具使用同一联机文档来实现帮助功能。

- **Eclipse** 在 Unix 平台上以 Eclipse 格式提供完整的联机帮助。要访问文档，请从 SQL Anywhere 11 安装的 *bin32* 或 *bin64* 目录下运行 *sadoc*。

- **DocCommentXchange** DocCommentXchange 是一个用于访问和讨论 SQL Anywhere 文档的社区。

  使用 DocCommentXchange 可以执行以下任务：

  - 查看文档
  - 检查是否有用户对文档各部分所做出的阐明
  - 提供建议和修正意见以在将来的版本中为所有用户改进文档

  访问 http://dcx.sybase.com。

- **PDF** 整套 SQL Anywhere 手册会以一组 Portable Document Format (PDF) 文件的形式提供。您必须有 PDF 阅读器才能查看信息。要下载 Adobe Reader，请访问 http://get.adobe.com/reader/。

  若要在 Microsoft Windows 操作系统上访问 PDF 文档，请选择 [开始] » [程序] » [**SQL Anywhere 11**] » 文档 » [联机手册 - PDF 格式]。

  要在 Unix 操作系统上访问 PDF 文档，请使用 Web 浏览器打开 *install-dir/documentation/zh/pdf/index.html*。

关于文档集中的手册

SQL Anywhere 文档由以下手册组成：

- **SQL Anywhere 11 - 简介** 本手册介绍 SQL Anywhere 11，一个提供数据管理和数据交换技术的综合数据包，通过它可以为服务器环境、台式机环境、移动环境以及远程办公环境快速开发由数据库驱动的应用程序。

- **SQL Anywhere 11 - 更改和升级** 本手册介绍 SQL Anywhere 11 以及该软件以前版本中的新功能。

- **SQL Anywhere 服务器 - 数据库管理** 本手册介绍如何运行、管理及配置 SQL Anywhere 数据库。它介绍了数据库连接、数据库服务器、数据库文件、备份过程、安全性、高可用性、使用复制服务器进行复制以及管理实用程序和选项。

- **SQL Anywhere 服务器 - 编程** 本手册介绍如何使用 C、C++、Java、PHP、Perl、Python 和 .NET 编程语言（例如 Visual Basic 和 Visual C#）建立和部署数据库应用程序。其中介绍了各种编程接口，如 ADO.NET 和 ODBC。
- **SQL Anywhere 服务器 - SQL 参考** 本手册提供了系统过程和目录（系统表和视图）的参考信息。也介绍了 SQL 语言（搜索条件、语法、数据类型和函数）的 SQL Anywhere 实现。
- **SQL Anywhere 服务器 - SQL 的用法** 本手册介绍如何设计和创建数据库；如何导入、导出和修改数据；如何检索数据以及如何建立存储过程和触发器。
- **MobiLink - 入门** 本手册介绍基于会话的关系数据库同步系统 MobiLink。MobiLink 技术支持双向复制并且非常适用于移动计算环境。
- **MobiLink - 客户端管理** 本手册介绍如何设置、配置和同步 MobiLink 客户端。MobiLink 客户端可以是 SQL Anywhere 或者 UltraLite 数据库。本手册同时也介绍了 Dbmlsync API，通过它可无缝地将同步集成到 C++ 或 .NET 客户端应用程序中。
- **MobiLink - 服务器管理** 本手册说明如何设置和管理 MobiLink 应用程序。
- **MobiLink - 服务器启动的同步** 本手册介绍 MobiLink 服务器启动的同步，这种功能允许 MobiLink 服务器启动同步或在远程设备上进行操作。
- **QAnywhere** 本手册介绍 QAnywhere，一个用于移动、无线、台式机和膝上型客户端的消息传递平台。
- **SQL Remote** 本手册介绍用于移动计算的 SQL Remote 数据复制系统，此系统支持使用电子邮件或文件传输等间接链接共享 SQL Anywhere 统一数据库和多个 SQL Anywhere 远程数据库之间的数据。
- **UltraLite - 数据库管理和参考** 本手册介绍适用于小型设备的 UltraLite 数据库系统。
- **UltraLite - C 及 C++ 编程** 本手册介绍 UltraLite C 和 C++ 编程接口。利用 UltraLite，可以开发数据库应用程序，并将它们部署到手持式设备、移动设备或嵌入式设备。
- **UltraLite - M-Business Anywhere 编程** 本手册介绍 UltraLite for M-Business Anywhere。利用 UltraLite for M-Business Anywhere，用户可以开发基于 Web 的数据库应用程序，并将它们部署到运行 Palm OS、Windows Mobile 或 Windows 的手持式设备、移动设备或嵌入式设备。
- **UltraLite - .NET 编程** 本手册介绍 UltraLite.NET。利用 UltraLite.NET，您可以开发数据库应用程序，并将它们部署到计算机、手持式设备、移动设备或嵌入式设备。
- **UltraLiteJ** 本手册介绍 UltraLiteJ。利用 UltraLiteJ，可以在支持 Java 的环境中开发和部署数据库应用程序。UltraLiteJ 支持 BlackBerry 智能手机和 Java SE 环境。UltraLiteJ 基于 iAnywhere UltraLite 数据库产品。
- **错误消息** 本手册提供了 SQL Anywhere 错误消息及其诊断信息的完整列表。

文档约定

本节列出了本文档中使用的约定。

操作系统

SQL Anywhere 可以在各种平台上运行。在大多数情况下，该软件在所有平台上的行为都是相同的，但也有变动或限制。这些变动或限制通常基于基础操作系统（Windows、Unix），很少基于特定变型（AIX、Windows Mobile）或版本。

为了简化对操作系统的提及，本文档按如下方式对支持的操作系统进行分组：

- **Windows** Microsoft Windows 系列包括 Windows Vista 和 Windows XP（主要用于服务器、台式计算机和膝上型计算机），以及 Windows Mobile（用于移动设备）。

  除非另外指定，否则当本文档提及 Windows 时，是指所有基于 Windows 的平台，包括 Windows Mobile。

- **Unix** 除非另外指定，否则当本文档提及 Unix 时，是指所有基于 Unix 的平台，包括 Linux 和 Mac OS X。

目录和文件名

大部分情况下，对目录和文件名的引用在所有支持的平台上都是类似的，只需在不同形式之间进行简单的转换。这时需使用 Windows 约定。在细节更为复杂的情况下，文档显示所有相关形式。

下面是文档编写中用于简化目录和文件名的约定：

- **大写和小写目录名** 在 Windows 和 Unix 上，目录和文件名可以包括大写和小写字母。创建目录和文件时，文件系统会保留字母大小写。

  在 Windows 上，对目录和文件的提及*不区分大小写*。混合使用大小写的目录和文件名很常见，但使用所有小写字母来提及目录和文件的形式也很常见。SQL Anywhere 安装包含诸如 *Bin32* 和 *Documentation* 的目录。

  在 Unix 上，对目录和文件的提及*区分大小写*。混合使用大小写的目录和文件名不常见。大多数的目录和文件名全部使用小写字母。SQL Anywhere 安装包含诸如 *bin32* 和 *documentation* 的目录。

  本文档采用 Windows 形式的目录名。大多数情况下，在 Unix 上可以将大小写混合形式的目录名转换成小写字母的等效目录名。

- **分隔目录和文件名的斜线** 文档使用反斜线作为目录分隔符。例如，PDF 格式的文档位于 *install-dir\Documentation\zh\PDF*（Windows 形式）。

  在 Unix 上，用正斜线替换反斜线。PDF 文档位于 *install-dir/documentation/zh/pdf* 下。

- **可执行文件** 文档使用 Windows 约定显示可执行文件名（带有诸如 *.exe* 或 *.bat* 后缀）。在 Unix 上，可执行文件名没有后缀。

  例如，在 Windows 上，网络数据库服务器是 *dbsrv11.exe*。在 Unix 上是 *dbsrv11*。

- ***install-dir*** 在安装过程中，选择 SQL Anywhere 的安装位置。创建环境变量 SQLANY11，用来表示此位置。文档中以 *install-dir* 表示此位置。

  例如，本文档将此文件表示为 *install-dir*\readme.txt。在 Windows 上，这等同于 *%SQLANY11%\readme.txt*。在 Unix 上，这等同于 *$SQLANY11/readme.txt* 或 *${SQLANY11}/readme.txt*。

有关 *install-dir* 缺省位置的详细信息，请参见“SQLANY11 环境变量”一节 《SQL Anywhere 服务器 - 数据库管理》。

- ***samples-dir*** 在安装过程中，选择 SQL Anywhere 随附的示例的安装位置。创建环境变量 SQLANYSAMP11，用来表示此位置。文档中以 *samples-dir* 表示此位置。

  要在 *samples-dir* 中打开 Windows 资源管理器窗口，请在 [**开始**] 菜单中，选择 [**程序**] » [**SQL Anywhere 11**] » [**示例应用程序和项目**]。

  有关 *samples-dir* 缺省位置的详细信息，请参见“SQLANYSAMP11 环境变量”一节 《SQL Anywhere 服务器 - 数据库管理》。

命令提示符和命令 shell 语法

大多数操作系统都提供一种或多种使用命令 shell 或命令提示符来输入命令和参数的方法。Windows 命令提示符包括 Command Prompt（DOS 提示符）和 4NT。Unix 命令 shell 包括 Korn shell 和 bash。每个 shell 都具有一些功能，其能力不仅仅局限于简单命令。这些功能通过特殊字符来驱动。特殊字符和功能随 shell 的不同而不同。如果没有正确使用这些特殊字符，通常会导致语法错误或意外行为。

本文档以普通形式提供命令行示例。如果这些示例中包含 shell 的特殊字符，则命令需要根据特定 shell 进行修改。修改方法不在本文档所述范围之内，但通常是在包含这些特殊字符的参数两旁加上引号，或是在特殊字符前面使用转义字符。

下面是命令行语法的一些示例，不同的平台可能会有不同的形式：

- **括号和大括号** 有些命令行选项需要一个参数，该参数将以列表形式接受详细的值指定。该列表通常用括号或大括号括起来。本文档使用括号。例如：

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

  如果括号导致出现语法问题，用大括号替代：

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

  如果两种形式都将产生语法问题，应按照 shell 的要求，用引号将整个参数括起来：

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

- **引号** 如果必须在参数值中指定引号，该引号可能会与用于括参数的引号的传统用法发生冲突。例如，要指定值中包含双引号的加密密钥，则可能必须用引号括起密钥，然后转义嵌入的引号：

  ```
  -ek "my \"secret\" key"
  ```

  在许多 shell 中，密钥的值为 `my "secret" key`。

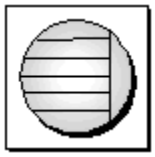
- **环境变量** 本文档介绍设置环境变量。在 Windows shell 中，环境变量使用语法 *%ENVVAR%* 来指定。在 Unix shell 中，环境变量使用语法 *$ENVVAR* 或 *${ENVVAR}* 来指定。

图标

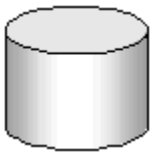
本文档中使用了下列图标。

- 客户端应用程序。

- 数据库服务器，如 Sybase SQL Anywhere。



- 数据库。在某些高水平的图中，可以使用此图标表示数据库和管理该数据库的数据库服务器。



- 复制或同步中间件。用于帮助在数据库之间共享数据。例如 MobiLink 服务器和 SQL Remote 消息代理。

- 编程接口。

联系文档小组

我们欢迎您就本帮助文档提出意见、建议和反馈信息。

要提交意见和建议，请发送电子邮件到 SQL Anywhere 文档小组，地址为 iasdoc@sybase.com。虽然我们不对这些电子邮件进行回复，但您的反馈会帮助我们改进文档，因此我们真诚地欢迎您提出宝贵的意见和建议。

DocCommentXchange

也可以使用 DocCommentXchange 将意见或建议直接置于帮助主题中。DocCommentXchange (DCX) 是一个用于访问和讨论 SQL Anywhere 文档的社区。使用 DocCommentXchange 可以执行以下任务：

- 查看文档
- 检查是否有用户对文档各部分所做出的阐明
- 提供建议和修正意见以在将来的版本中为所有用户改进文档

访问 http://dcx.sybase.com。

查找详细信息并请求技术支持

附加信息和资源可从 Sybase iAnywhere 开发人员社区获得，网址是 http://www.sybase.com/developer/library/sql-anywhere-techcorner。

如果您有问题或是需要帮助，可将邮件发布到下面所列的 Sybase iAnywhere 新闻组。

当您向这些新闻组发布邮件时，请务必提供问题的详细信息，包括 SQL Anywhere 版本的内部版本号。可以通过运行以下命令找到此信息：**dbeng11 -v**。 **dbeng11 -v**.

新闻组位于 *forums.sybase.com* 新闻服务器上。

这些新闻组包括：

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

有关 Web 开发问题，请访问 http://groups.google.com/group/sql-anywhere-web-development。

新闻组免责声明

iAnywhere Solutions 没有义务为其新闻组提供解决方案、信息或建议，除提供系统操作员监控服务和确保新闻组的运行和可用性外，iAnywhere Solutions 也没有义务提供任何其它服务。

如果时间允许，iAnywhere 技术顾问以及其他员工也会对新闻组服务提供帮助。他们是在自愿的基础上提供帮助的，所以可能无法定期提供解决方案和信息。他们可以提供多少帮助取决于他们的工作量。

UltraLite for C/C++ 开发人员

目录

开发嵌入式 SQL 应用程序 **2**
系统要求和支持的平台 **3**
UltraLite C++ 组件体系结构 **4**
了解 SQL 通信区 **5**
创建数据库 **6**

C 和 C++ 接口为面向小型设备的 UltraLite 开发人员带来了以下益处：

- 高性能的小型数据库存储。
- C 或 C++ 语言的强大功能、高效和灵活性。
- 在 Windows Mobile、Palm OS 和 Windows 桌面操作系统平台上部署应用程序的功能。

有关 UltraLite 数据库功能的详细信息，请参见“创建和配置 UltraLite 数据库”《UltraLite - 数据库管理和参考》。

使用 C++ 的 UltraLite 开发人员可以有两个选择：

- UltraLite C++ API。
- ODBC 编程接口（组件接口）。

使用 C 的 UltraLite 开发人员必须使用嵌入式 SQL 或 ODBC 编程接口。

开发嵌入式 SQL 应用程序

在开发嵌入式 SQL 应用程序时，您要将 SQL 语句与标准 C 或 C++ 源代码混合使用。要开发嵌入式 SQL 应用程序，您应该熟悉 C 或 C++ 编程语言。

嵌入式 SQL 应用程序的开发过程如下：

1. 创建 UltraLite 数据库。
2. 在嵌入式 SQL 源文件（该文件的扩展名通常为 *.sqc*）中写入您的源代码。

   当源代码中需要数据访问时，请使用要执行的 SQL 语句，并以 EXEC SQL 关键字为前缀。例如：

   ```
   EXEC SQL SELECT price, prod_name
      INTO :cost, :pname
      FROM ULProduct
      WHERE prod_id= :pid;
   if((SQLCODE==SQLE_NOTFOUND)||(SQLCODE<0)) {
     return(-1);
   }
   ```

3. 预处理 *.sqc* 文件。

   SQL Anywhere 包含一个 SQL 预处理器 (sqlpp)，该处理器读取 *.sqc* 文件并生成 *.cpp* 文件。这些文件保存对 UltraLite 运行时库的函数调用。

4. 编译 *.cpp* 文件。
5. 链接 *.cpp* 文件。

   必须将这些文件和 UltraLite 运行时库链接起来。

有关嵌入式 SQL 开发的详细信息，请参见“构建嵌入式 SQL 应用程序”一节第 58 页。

系统要求和支持的平台

开发平台

要使用 UltraLite C++ 开发应用程序，需要以下各项：

- Microsoft Windows 桌面操作系统作为开发平台。
- 受支持的 C/C++ 编译器。

目标平台

UltraLite C/C++ 支持以下目标平台：

- Windows Mobile 3.0 或更高版本
- Palm OS 4.0 或更高版本

有关所支持目标平台的详细信息，请参见 http://www.sybase.com/detail?id=1062617。

UltraLite C++ 组件体系结构

在 *uliface.h* 头文件中定义 UltraLite C++ 组件接口。以下列表介绍了一些常用的对象：

- **DatabaseManager** 要为每个应用程序创建一个 DatabaseManager 对象。
- **Connection** 表示与 UltraLite 数据库的连接。您可以创建一个或多个 Connection 对象。
- **Table** 提供对数据库中数据的访问。
- **PreparedStatement、ResultSet 和 ResultSetSchema** 创建动态 SQL 语句，进行查询和执行 INSERT、UPDATE 和 DELETE 语句，以及实现对数据库结果集的程序化控制。
- **SyncParms** 将 UltraLite 数据库与 MobiLink 服务器同步。

有关访问 API 参考的详细信息，请参见“UltraLite C++ API 参考”第 123 页。

了解 SQL 通信区

所有 UltraLite C/C++ 接口使用同一个 UltraLite 运行时引擎。这些 API 分别提供对相同基础功能的访问。

所有 UltraLite C/C++ 接口共享相同的基本数据结构，用于在运行库和应用程序之间调度数据。这个数据结构就是 SQL 通信区（或 SQLCA）。每个 SQLCA 都有一个当前连接，不同的线程不能共享公用 SQLCA。

您的应用程序代码必须在连接到数据库之前执行以下任务：

- 初始化 SQLCA。这是为了准备您的应用程序与 UltraLite 运行时通信。
- 注册错误回调函数。
- 启动数据库。此操作可以作为打开连接的一部分来执行。

下列函数是执行这些任务的等效方法。

任务	接口	功能
初始化 SQLCA	嵌入式 SQL	db_init
	C++	ULSqlca::Initialize
初始化 SQLCA 并启动数据库	嵌入式 SQL	db_init db_start_database
	C++	数据库作为 UltraLite_DatabaseManager 中连接函数的一部分启动

创建数据库

可使用以下任一方法创建 UltraLite 数据库：

- Sybase Central 中的 [**创建数据库向导**]。
- 命令行实用程序（如 ulcreate 或 ulinit）。
- 调用 ULCreateDataBase 函数。

使用 Sybase Central，可以交互方式创建数据库，并相应定义所需表和其它模式相关项。

ulcreate 实用程序创建一个不定义任何表的空数据库。通过调用 ULCreateDatabase 创建数据库的应用程序还需要执行 SQL CREATE 语句来创建表和索引定义。

显式命名数据库

不同的接口可以对数据库使用不同的缺省文件名。如果混合接口，最好在创建或连接数据库时始终显式命名数据库。可以使用 DBN= 连接参数来完成此操作。请参见“UltraLite DBN 连接参数”一节 《UltraLite - 数据库管理和参考》。

应用程序开发

本节提供针对 UltraLite C/C++ 程序员的开发说明。

使用 UltraLite C++ API 开发应用程序 9
使用嵌入式 SQL 开发应用程序 29
开发用于 Palm OS 的 UltraLite 应用程序 61
开发用于 Windows Mobile 的 UltraLite 应用程序 77

使用 UltraLite C++ API 开发应用程序

目录

使用 UltraLite 命名空间 10
连接到数据库 11
使用 SQL 访问数据 13
使用 Table API 访问数据 17
管理事务 22
访问模式信息 23
处理错误 24
验证用户 25
对数据进行加密 26
同步数据 27
编译和链接应用程序 28

使用 UltraLite 命名空间

UltraLite C++ 接口提供一组名称以 UltraLite_ 为前缀的类（如 UltraLite_Connection 和 UltraLite_DatabaseManager）。其中每个类的大部分函数都使用附加字符串 _iface 的基础接口实现函数。例如，UltraLite_Connection 类从 UltraLite_Connection_iface 实现函数。

当显式使用 UltraLite 命名空间时，可使用较短的名称引用每个类。如果您使用 UltraLite 命名空间，则不必将连接声明为 UltraLite_Connection 对象，而是可以将其声明为 Connection 对象：

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

作为此体系结构的结果，本章中的代码示例使用如 DatabaseManager、Connection 和 TableSchema 的类型，但详细信息的链接可能分别指向 UltraLite_DatabaseManager_iface、UltraLite_Connection_iface 和 UltraLite_TableSchema_iface。

连接到数据库

UltraLite 应用程序必须先连接到数据库，然后才能对数据库中的数据进行操作。本节将介绍如何连接到 UltraLite 数据库。

可以在 *samples-dir\UltraLite\CustDB* 目录中找到示例代码。

Connection 对象的属性

- **提交行为** UltraLite C++ API 中没有 AutoCommit 模式。每个事务后面必须跟上一条 Connection->Commit() 语句。请参见“管理事务”一节第 22 页。
- **用户验证** 可以使用授予和撤消连接权限的方法更改应用程序的用户 ID 和口令（分别从其缺省值 **DBA** 和 **sql** 更改为其它值）。每个数据库最多可以有四个用户 ID。请参见“验证用户”一节第 25 页。
- **同步** 通过使用 Connection 对象的方法可将 UltraLite 数据库与统一数据库同步。请参见“同步数据”一节第 27 页。
- **表** 使用 Connection 对象的方法可以访问 UltraLite 数据库表。请参见“使用 Table API 访问数据”一节第 17 页。
- **预准备语句** 提供了一些用来处理 SQL 语句的执行的方法。请参见“使用 SQL 访问数据”一节第 13 页和“UltraLite_PreparedStatement 类”一节第 193 页。

连接到 UltraLite 数据库

♦ **连接到 UltraLite 数据库**

1. 使用 UltraLite 命名空间。

   使用 UltraLite 命名空间将允许您为 C++ 接口中的类使用简单名称。

   ```
   using namespace UltraLite;
   ```

2. 创建并初始化一个 DatabaseManager 对象和一个 UltraLite SQL 通信区 (ULSqlca)。ULSqlca 是一个处理应用程序和数据库之间通信的结构。

   DatabaseManager 对象位于对象层次的根部。对于每个应用程序，只应创建一个 DatabaseManager 对象。通常，最好将 DatabaseManager 对象声明为应用程序范围内的全局对象。

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

   如果应用程序不需要 SQL 支持并且直接链接 UltraLite 运行时，则它可以调用 ULInitDatabaseManagerNoSQL 来初始化 ULSqlca。此变量减少了应用程序的大小。

   请参见“UltraLite_DatabaseManager_iface 类”一节第 180 页。

3. 打开与现有数据库的连接，或者，如果指定的数据库文件不存在，创建一个新数据库。请参见“OpenConnection 函数”一节第 181 页。

可以使用初始空数据库部署 UltraLite 应用程序，或者，如果该数据库尚不存在，该应用程序可以创建 UltraLite 数据库。部署初始数据库是最简单的解决方案；否则该应用程序必须调用 ULCreateDatabase 函数创建该数据库并且必须创建应用程序所需的所有表。请参见“ULCreateDatabase 函数”一节第 100 页。

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

多线程应用程序

每个连接以及从中创建的所有对象都应该由单个线程使用。如果应用程序需要使用多个线程访问 UltraLite 数据库，则每个线程都需要一个单独的连接。

使用 SQL 访问数据

UltraLite 应用程序可以执行 SQL 语句或使用 Table API 访问表数据。本节介绍如何使用 SQL 语句访问数据。

有关使用 Table API 的详细信息，请参见“使用 Table API 访问数据”一节第 17 页。

本节讲解如何使用 SQL 执行以下任务：

- 插入、删除和更新行。
- 将行检索到一个结果集。
- 滚动浏览结果集中的行。

本节不介绍 SQL 语言。有关 SQL 语言的详细信息，请参见“UltraLite SQL 语句” 《UltraLite - 数据库管理和参考》。

数据操作：Insert、Delete 和 Update

使用 UltraLite，可以使用 ExecuteStatement 方法（PreparedStatement 类的一个成员）执行 SQL 数据操作。

请参见“UltraLite_PreparedStatement 类”一节第 193 页。

引用预准备语句中的参数
UltraLite 使用 ? 字符表示查询参数。对于任何 INSERT、UPDATE 或 DELETE 语句，每个 ? 都是根据其在预准备语句中的顺序位置引用的。例如，第一个 ? 引用为参数 1，第二个引用为参数 2。

♦ 插入一行

1. 声明 PreparedStatement。

   ```
   PreparedStatement * prepStmt;
   ```

   请参见“PrepareStatement 函数”一节第 162 页。

2. 将一条 SQL 语句指派给 PreparedStatement 对象。

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

3. 为该语句指派输入参数值。

   以下代码显示一个字符串参数。

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

4. 执行预准备语句。

   返回值表示受该语句影响的行数。

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

5. 提交更改。

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

♦ 删除一行

1. 声明 PreparedStatement。

```
PreparedStatement * prepStmt;
```

2. 将一条 SQL 语句指派给 PreparedStatement 对象。

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

3. 为该语句指派输入参数值。

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

4. 执行该语句。

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

5. 提交更改。

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

♦ 更新一行

1. 声明 PreparedStatement。

```
PreparedStatement * prepStmt;
```

2. 将一条语句指派给 PreparedStatement 对象。

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

3. 为该语句指派输入参数值。

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

4. 执行该语句。

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

5. 提交更改。

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

数据检索：SELECT

使用 SELECT 语句可从数据库中检索信息。执行 SELECT 语句时，PreparedStatement.ExecuteQuery 方法返回一个 ResultSet 对象。

请参见“UltraLite_PreparedStatement_iface 类”一节第 194 页。

♦ 执行 SELECT 语句

1. 创建预准备语句对象。

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

2. 执行该语句。

   在以下代码中，SELECT 查询的结果包含一个字符串，该字符串输出到命令提示符。

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

浏览 SQL 结果集

可以使用与 ResultSet 对象关联的方法浏览结果集。

结果集对象提供了以下方法来浏览结果集：

- **AfterLast** 定位到紧接在最后一行后面的位置。
- **BeforeFirst** 定位到紧接在第一行前面的位置。
- **First** 移至第一行。
- **Last** 移至最后一行。
- **Next** 移至下一行。
- **Previous** 移至上一行。
- **Relative(offset)** 根据带符号的偏移值的指定，相对于当前行移动特定行数。如果偏移值为正，则相对于游标在结果集中的当前位置在结果集中向前移动。如果偏移值为负，则在结果集中向后移动。如果偏移值为零，则不移动当前位置，但可以重新填充行缓冲区。

请参见“UltraLite_ResultSet_iface 类”一节第 199 页。

结果集模式说明

ResultSet->GetSchema 方法允许您检索有关结果集的模式信息，例如：列名、总列数、列小数位数、列大小以及列 SQL 类型。

示例

以下示例演示如何使用 ResultSet.GetSchema 方法在命令提示符中显示模式信息。

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

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

请参见“GetSchema 函数”一节第 158 页。

使用 Table API 访问数据

UltraLite 应用程序可以执行 SQL 语句或使用 Table API 访问表数据。本节介绍如何使用 Table API 访问数据。

有关通过执行 SQL 语句访问数据的详细信息，请参见“使用 SQL 访问数据”一节第 13 页。

本节讲解如何使用 Table API 执行以下任务：

- 滚动浏览表中的行。
- 访问当前行中的值。
- 使用 Find 和 Lookup 方法来查找表中的行。
- 插入、删除和更新行。

浏览表中的行

UltraLite C++ API 提供了几种浏览表的方法，以执行各种浏览任务。

表对象提供了以下方法来浏览表：

- **AfterLast** 定位到紧接在最后一行后面的位置。
- **BeforeFirst** 定位到紧接在第一行前面的位置。
- **First** 移至第一行。
- **Last** 移至最后一行。
- **Next** 移至下一行。
- **Previous** 移至上一行。
- **Relative(offset)** 根据带符号的偏移值的指定，相对于当前行移动特定行数。如果偏移值为正，则相对于游标在结果集中的当前位置在结果集中向前移动。如果偏移值为负，则在结果集中向后移动。如果偏移值为零，则不移动当前位置，但可以重新填充行缓冲区。

请参见“UltraLite_Table_iface 类”一节第 215 页。

示例

以下代码打开名为 MyTable 的表并显示每一行名为 MyColumn 的列的值。

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

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


打开表对象时，应用程序可以访问表中的行。缺省情况下，这些行按主键值排序，但可在打开表时指定一个索引，以便以特定的顺序访问行。

示例

以下代码片段用于移动到按 ix_col 索引指定的顺序排序的 MyTable 表的第一行。

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

请参见“UltraLite_Table_iface 类”一节第 215 页。

UltraLite 模式

UltraLite 模式确定如何使用缓冲区中的值。您可以将 UltraLite 模式设置为以下模式之一：

- **插入模式** 调用 insert 方法时，将缓冲区中的数据作为新行添加到表中。
- **更新模式** 调用 update 方法时，缓冲区中的数据将替换当前行。
- **查找模式** 在调用查找方法之一时，会定位其值与缓冲区中的数据完全匹配的行。
- **查寻模式** 调用查寻方法之一时，会定位其值与缓冲区中的数据匹配或大于缓冲区中的数据的行。

通过调用用于设置模式的相应方法即可设置该模式。例如，InsertBegin、BeginUpdate、FindBegin 等。

访问当前行

Table 对象始终位于以下位置之一：

- 在表的第一行之前。
- 在表的某一行上。
- 在表的最后一行之后。

如果 Table 对象位于某一行上，可以在适合该数据类型的一组方法中使用一种方法来检索或修改该行中各列的值。

检索列值

Table 对象提供了一组用于检索列值的方法。这些方法都将列 ID 作为参数。

以下代码片段用于检索 lname 列的值，该值是一个字符串。

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


以下代码检索 cust_id 列的值，该值是一个整数。

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

修改列值

除用于检索值的方法外，还有用于设置值的方法。这些方法将列 ID 和值作为参数。

例如，以下代码将 lname 列的值设置为 Kaminski。

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

通过设置列值，您不用直接变更数据库中的数据。即使是在该表的第一行之前或最后一行之后，您也可以给属性指派值。当前行未定义时，不要尝试访问数据。例如，尝试读取以下示例中的列值是错误的：

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

转换值

所选方法应与要指派的数据类型匹配。UltraLite 自动转换兼容的数据库数据类型，这样您就可以使用 GetString 方法将一个整数值读取到字符串变量中，以及执行类似的操作。请参见“显式转换数据类型”一节 《UltraLite - 数据库管理和参考》。

搜索行

UltraLite 具有用于处理数据的不同操作模式。您可以将其中两种模式（查找和查寻）用于搜索。Table 对象具有对应于这些模式的方法，用于定位表中的特定行。

注意
使用 Find 方法和 Lookup 方法搜索的列必须在用于打开该表的索引中。

- **Find 方法** 按照打开 Table 对象时指定的排序顺序，移动到与指定的搜索值完全匹配的第一行。如果找不到搜索值，则应用程序将定位到第一行之前或最后一行之后。
- **Lookup 方法** 按照打开 Table 对象时指定的排序顺序，移动到与指定的搜索值匹配或大于指定的搜索值的第一行。

♦ **搜索行**

1. 进入查找或查寻模式。

   调用表对象上的方法来设置模式。例如，以下代码将进入查找模式。

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

2. 设置搜索值。

设置当前行中的搜索值。设置这些值仅影响保存当前行的缓冲区，而不影响数据库。例如，以下代码段用于将缓冲区中的值设置为 Kaminski。

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

3. 搜索行。

   调用适当的方法来执行搜索。例如，以下代码在当前索引中查找与指定值完全匹配的第一行。

   对于多列索引，将始终使用第一列的值，但可以忽略其它列。

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

4. 搜索该行的下一个实例。

   调用适当的方法来执行搜索。对于查找操作，FindNext 查找索引中参数的下一个实例。对于查寻操作，MoveNext 查找下一个实例。

请参见“UltraLite_Table_iface 类”一节第 215 页。

更新行

以下过程更新一行。

♦ **更新一行**

1. 移动到想要更新的行。

   可以通过滚动浏览表或使用 find 和 lookup 方法在表中进行搜索来移动到某一行。

2. 进入更新模式。

   例如，以下指令在表 tbl 上进入更新模式。

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

3. 为要更新的行设置新值。例如，以下指令将缓冲区中的 ID 列设置为 3。

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

4. 执行更新。

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

完成更新操作后，当前行就是已更新的行。

UltraLite C++ API 只能使用 `conn->Commit()` 向数据库提交更改。请参见“管理事务”一节第 22 页。

小心
不要更新行的主键：而是删除该行并添加新行。

插入行

插入行的步骤与更新行的步骤非常相似，区别在于无需在执行插入操作之前在表中定位行。由于始终根据索引在数据库中插入数据，因此在表中插入行的顺序无关紧要。

示例

以下代码段用于插入一个新行。

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

如果没有设置其中一列的值，并且该列有缺省值，则使用该缺省值。如果该列没有缺省值，将使用以下条目之一：

- 对于可为空的列，添加空值。
- 对于禁止使用空值的数字列，添加 0。
- 对于禁止使用空值的字符列，添加空字符串。
- 若要显式将一个值设置为 NULL，可使用 SetNull 方法。

删除行

删除行的步骤比插入或更新行的步骤更简单。

以下过程删除一行。

♦ 删除一行

1. 移到想要删除的行。
2. 执行 Table.Delete 方法。

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


管理事务

UltraLite C++ API 不支持 AutoCommit 模式。事务必须显式提交或回退。

♦ **提交事务**

- 执行 Conn->Commit 语句。

  请参见“Commit 函数”一节第 151 页。

♦ **回退事务**

- 执行 Conn->Rollback 语句。

  请参见“Rollback 函数”一节第 163 页。

有关 UltraLite 中事务管理的详细信息，请参见“UltraLite 事务处理”一节 《UltraLite - 数据库管理和参考》。

访问模式信息

API 中的对象表示表、列、索引和同步发布。每个对象都有一个 GetSchema 方法，用于访问与该对象的结构有关的信息。

您无法通过 API 修改模式。只能检索关于模式的信息。

您可以访问以下模式对象和信息：

- **DatabaseSchema** 提供数据库中表的数量和名称，以及日期和时间格式等全局属性。

  要获取 DatabaseSchema 对象，请使用 Conn->GetSchema。

  请参见“GetSchema 函数”一节第 158 页。

- **TableSchema** 提供此表的列和索引的数量和名称。

  要获取 TableSchema 对象，请使用 tbl->GetSchema。

  请参见“GetSchema 函数”一节第 158 页。

- **IndexSchema** 返回有关索引中的列的信息。由于索引没有与其直接关联的数据，因此没有单独的 Index 类，而只有一个 IndexSchema 类。

  要获取 IndexSchema 对象，请调用 table_schema->GetIndexSchema 或 table_schema->GetPrimaryKey 方法。

  请参见“UltraLite_Table_iface 类”一节第 215 页。

处理错误

每次数据库操作后，都应使用 ULSqlca 对象的方法检查错误。例如，LastCodeOK 检查操作是否成功，而 GetSQLCode 返回 SQLCode 的数字值。有关这些值含义的详细信息，请参见“按 Sybase 错误代码排序的 SQL Anywhere 错误消息”一节 《错误消息》。

除了显式错误处理外，UltraLite 还支持一个错误回调函数。如果注册一个回调函数，UltraLite 就可以在发生 UltraLite 错误时调用该函数。此回调函数不控制应用程序流，但确实能使您获知所有的错误。使用回调函数在应用程序开发和调试期间尤其有用。有关使用回调函数的详细信息，请参见“教程：使用 C++ API 构建应用程序”第 309 页。

有关示例回调函数，请参见“ULRegisterErrorCallback 的回调函数”一节第 94 页和“ULRegisterErrorCallback 函数”一节第 114 页。

有关 UltraLite C++ API 抛出的错误代码的列表，请参见“按 Sybase 错误代码排序的 SQL Anywhere 错误消息”一节 《错误消息》。

验证用户

UltraLite 数据库最多可以定义四个用户 ID。使用缺省用户 ID 和口令（分别为 **DBA** 和 **sql**）创建 UltraLite 数据库。与 UltraLite 数据库的所有连接必须提供用户 ID 和口令。建立连接后，可以更改口令及添加和删除用户 ID。

不能直接更改用户 ID。可以添加一个用户 ID，然后删除现有用户 ID。

♦ 添加用户或更改现有用户的口令

1. 以现有用户身份连接到数据库。
2. 使用 conn->GrantConnectTo 方法，授予具有所需口令的用户连接权限。

   无论要添加新用户还是要更改现有用户的口令，此过程均相同。

   请参见“GrantConnectTo 函数”一节第 159 页。

♦ 删除现有用户

1. 以现有用户身份连接到数据库。
2. 使用 conn->RevokeConnectFrom 方法撤消用户的连接权限。

   请参见“RevokeConnectFrom 函数”一节第 163 页。

对数据进行加密

可选择使用 UltraLite C++ API 对 UltraLite 数据库进行加密或模糊处理。加密为数据库中的数据提供了非常安全的表示，而模糊处理提供了级别过于简单的安全性，旨在防止随意查看数据库的内容。

有关背景信息，请参见“为 UltraLite 选择数据库创建参数”一节 《UltraLite - 数据库管理和参考》。

加密

要创建加密的数据库，请通过指定连接字符串中的 **key=** 连接参数来指定加密密钥。调用 CreateDatabase 方法时，您就在创建数据库的同时用指定的密钥加密了数据库。

对数据库进行加密后，所有与数据库的连接必须指定正确的加密密钥。否则，连接将失败。

请参见“UltraLite DBKEY 连接参数”一节 《UltraLite - 数据库管理和参考》。

模糊处理

要对数据库进行模糊处理，请指定 `obfuscate=1` 作为一个数据库创建参数。

请参见“保护 UltraLite 数据库”一节 《UltraLite - 数据库管理和参考》。

同步数据

UltraLite 应用程序可将数据与中央数据库同步。同步需要使用 SQL Anywhere 附带的 MobiLink 同步软件。

UltraLite C++ API 支持 TCP/IP、TLS、HTTP 和 HTTPS 同步。同步由 UltraLite 应用程序启动。在所有情况中，您都可以使用连接对象的方法和属性来控制同步。

有关同步的详细信息，请参见“UltraLite 客户端”《UltraLite - 数据库管理和参考》。

有关用于同步的 ul_synch_info 结构的详细信息，根据您使用的是 ASCII 字符还是宽字符，请参见“ul_synch_info_a 结构”一节第 127 页或“ul_synch_info_w2 结构”一节第 129 页。

有关同步参数的详细信息，请参见“UltraLite 的同步参数”一节 《UltraLite - 数据库管理和参考》。

编译和链接应用程序

使用 UltraLite C++ API 时，一些平台可以使用一组运行时库。对于 Windows Mobile 和 Windows 平台，这些运行时库包括允许对同一个数据库进行多进程访问的数据库引擎。

在 *install-dir\UltraLite\Palm*、*install-dir\UltraLite\ce*、*install-dir\UltraLite\win32* 和 *install-dirx64* 目录中提供了运行时库。

用于 Palm OS 的运行时库

为 Palm OS 上的应用程序提供了以下库。这些库位于 *install-dir\UltraLite\Palm\68k\lib\cw* 中。

- **ulrt.lib** 静态库。
- **ulbase.lib** 包含单独的动态链接库（dynamic link library，简称 DLL）中无法提供的额外函数的库。C/C++ 应用程序应链接到此库，以确保访问 UltraLite 功能。

用于 Windows Mobile 的运行时库

Windows Mobile 库位于 *install-dir\UltraLite\ce\arm.50\Lib* 目录中。

为 Windows Mobile 提供了以下动态库：

- **ulbase.lib** 包含单独的动态链接库（dynamic link library，简称 DLL）中无法提供的额外函数的库。C/C++ 应用程序应链接到此库，以确保访问 UltraLite 功能。
- **ulrt.lib** 在链接到此库时，确保指定以下编译选项：

  ```
  /DUNICODE
  ```

- **ulrtc.lib** Unicode 字符集静态库，与用于多进程访问 UltraLite 数据库的 UltraLite 引擎一起使用。

  在链接到此库时，确保指定以下编译选项：

  ```
  /DUNICODE
  ```

用于 Windows 桌面操作系统的运行时库

install-dir\UltraLite\win32\386\Lib\vs8 和 *install-dir\UltraLite\x64\Lib\vs8* 目录中包含了 Windows 桌面操作系统所支持的库。包括以下库：

- **ulbase.lib** 包含单独的动态链接库（dynamic link library，简称 DLL）中无法提供的函数的库。C/C++ 应用程序应链接到此库，以确保访问 UltraLite 功能。
- **ulrt11.dll** ANSI 字符集动态链接库。要使用此库，请将应用程序与导入库 (*ulimp.lib*) 链接。

  在链接到此库时，确保指定以下编译选项：

  ```
  /DUL_USE_DLL
  ```


使用嵌入式 SQL 开发应用程序

目录

嵌入式 SQL 示例 30
初始化 SQL 通信区 32
连接到数据库 34
使用主机变量 35
读取数据 44
验证用户 48
对数据进行加密 50
向应用程序添加同步 51
构建嵌入式 SQL 应用程序 58

本节介绍如何为嵌入式 SQL UltraLite 应用程序编写数据库访问代码。

有关 UltraLite C/C++ 开发过程的概述，请参见“UltraLite for C/C++ 开发人员”第 1 页。

有关嵌入式 SQL 参考信息，请参见“嵌入式 SQL API 参考”第 249 页。

有关 SQL 预处理器的详细信息，请参见“UltraLite 实用程序的 SQL 预处理器 (sqlpp)”一节《UltraLite - 数据库管理和参考》。

嵌入式 SQL 示例

嵌入式 SQL 是由 C/C++ 程序代码与伪代码组合而成的环境。可穿插传统 C/C++ 代码的伪代码是 SQL 语句的子集。预处理器将嵌入式 SQL 语句转换为函数调用，这些函数调用是被编译以创建该应用程序的实际代码的一部分。

下面是一个非常简单的嵌入式 SQL 程序示例。它演示了通过更改雇员 195 的姓氏对 UltraLite 数据库记录进行更新。

```
#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 );
}
```

尽管此示例过于简单而没什么用处，但它说明了所有嵌入式 SQL 应用程序所共有的以下方面：

- 每个 SQL 语句前面都有关键字 EXEC SQL。
- 每个 SQL 语句都以分号结尾。
- 一些嵌入式 SQL 语句不属于标准 SQL。INCLUDE SQLCA 语句就是一个例子。
- 除 SQL 语句外，嵌入式 SQL 还提供执行一些特定任务的库函数。函数 db_init 和 db_fini 就是两个库函数调用的示例。

初始化

以上示例代码说明了在使用 UltraLite 数据库中的数据前必须包括的初始化语句：

1. 使用以下命令定义 SQL 通信区 (SQLCA)：

   ```
   EXEC SQL INCLUDE SQLCA;
   ```

   此定义必须是第一条嵌入式 SQL 语句，所以很自然地，它的位置应在包含列表的末尾处。

   如果应用程序中有多个 *.sqc* 文件，则每个文件中都必须有此行。

2. 第一个数据库操作必须是调用名为 db_init 的嵌入式 SQL 库函数。该函数初始化 UltraLite 运行时库。在进行此调用前仅可执行嵌入式 SQL 定义语句。

   请参见“db_init 函数”一节第 252 页。

3. 必须使用 SQL CONNECT 语句连接到 UltraLite 数据库。

准备退出

以上示例代码说明了在准备退出时必需执行的一系列调用：

1. 提交或回退任何未完成的更改。
2. 断开与数据库的连接。
3. 调用名为 db_fini 的库函数结束 SQL 工作。

退出时，将自动回退所有未提交的数据库更改。

错误处理

在此示例中实际上没有任何 SQL 和 C 代码间的交互。C 代码仅控制程序流。WHENEVER 语句用于错误检查。在本示例中，在任何 SQL 语句引起错误之后执行错误处理操作 GOTO。

嵌入式 SQL 程序的结构

所有嵌入式 SQL 语句都以 EXEC SQL 开头，并以分号结尾。在嵌入式 SQL 语句的中间允许使用常规 C 语言注释。

使用嵌入式 SQL 的每个 C 程序都必须在源文件中任何其它嵌入式 SQL 语句之前包含以下语句。

```
EXEC SQL INCLUDE SQLCA;
```

程序中的第一个嵌入式 SQL 可执行语句必须是 SQL CONNECT 语句。CONNECT 语句提供用于建立到 UltraLite 数据库的连接的连接参数。

一些嵌入式 SQL 命令不生成任何可执行的 C 代码，或不涉及与数据库的通信。在执行 CONNECT 语句之前仅允许执行这些命令。最主要的是 INCLUDE 语句和指定错误处理方法的 WHENEVER 语句。

初始化 SQL 通信区

SQL 通信区 (SQLCA) 是一个内存区域，用于在应用程序和数据库之间相互传递统计信息和错误。SQLCA 用作应用程序到数据库的通信链接的句柄。它被显式传递给与数据库通信的所有数据库库函数，并隐式地在所有嵌入式 SQL 语句中传递。

UltraLite 在生成的代码中定义一个 SQLCA 全局变量。预处理器生成全局 SQLCA 变量的外部引用。该外部引用名为 sqlca，类型为 SQLCA。实际的全局变量在导入库中声明。

在头文件 *install-dir\SDK\Include\sqlca.h* 中定义 SQLCA 类型。

声明 SQLCA (`EXEC SQL INCLUDE SQLCA;`) 之后，必须先通过调用 db_init 并将 SQLCA 传递给它来初始化通信区，然后应用程序才可以对数据库执行操作：

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

SQLCA 提供错误代码

可引用 SQLCA 以测试特定错误代码。当数据库请求引起错误时，sqlcode 字段将含有一个错误代码。宏是为引用 sqlca 中的 sqlcode 字段和某些其它字段而定义的。

SQLCA 字段

SQLCA 包含以下字段：

- **sqlcaid** 8 字节字符字段，包含作为 SQLCA 结构标识的字符串 SQLCA。在您查看内存内容时，该字段可帮助进行调试。
- **sqlcabc** 包含 SQLCA 结构的长度（以字节为单位）的长整数。
- **sqlcode** 数据库对请求检测到错误时，包含错误代码的长整数。错误代码的定义位于头文件 *install-dir\SDK\Include\sqlerr.h* 中。成功操作的错误代码是 0（零），正值表示警告，负值表示错误。

  可使用 SQLCODE 宏直接访问此字段。

  有关错误代码的列表，请参见“SQL Anywhere 错误消息” 《错误消息》。
- **sqlerrml** sqlerrmc 字段中信息的长度。

  UltraLite 应用程序不使用此字段。
- **sqlerrmc** 可以包含准备插入到错误消息中的一个或多个字符串。一些错误消息包含占位符字符串 (*%1*)，它将被此字段中的文本替换。

  UltraLite 应用程序不使用此字段。
- **sqlerrp** 保留。
- **sqlerrd** 长整数的实用程序数组。
- **sqlwarn** 保留。

  UltraLite 应用程序不使用此字段。

- **sqlstate** SQLSTATE 状态值。

  UltraLite 应用程序不使用此字段。

连接到数据库

要从嵌入式 SQL 应用程序连接到 UltraLite 数据库，请在初始化 SQLCA 之后将 EXEC SQL CONNECT 语句包含到代码中。

CONNECT 语句具有以下格式：

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

连接字符串（括在单引号中）可包括其它的数据库连接参数。

有关数据库连接参数的详细信息，请参见“UltraLite 连接参数” 《UltraLite - 数据库管理和参考》。

有关 CONNECT 语句的详细信息，请参见“CONNECT 语句 [ESQL] [Interactive SQL]”一节《SQL Anywhere 服务器 - SQL 参考》。

管理多个连接

如果希望应用程序中有多个数据库连接，您可以使用多个 SQLCA 或单个 SQLCA 来管理连接。

使用多个 SQLCA

- **管理多个 SQLCA**

1. 在程序中使用的每个 SQLCA 都必须调用 **db_init** 进行初始化，并在结尾处调用 **db_fini** 清除它。

   请参见“db_init 函数”一节第 252 页。

2. 嵌入式 SQL 语句 SET SQLCA 用于通知 SQL 预处理器对于数据库请求使用特定的 SQLCA。在程序顶部或头文件中，通常会使用如下的语句将 SQLCA 引用设置为指向任务特定的数据：

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

   此语句不生成任何代码，且不影响性能。它更改预处理器内的状态，以便对 SQLCA 的任何引用都使用给定的字符串。

   有关创建 SQLCA 的详细信息，请参见“SET SQLCA 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用单个 SQLCA

也可以不使用多个 SQLCA，而使用单个 SQLCA 来管理多个到数据库的连接。

每个 SQLCA 只有一个活动连接或当前连接，但可以更改该连接。在执行命令前，可使用 SET CONNECTION 语句指定应该在哪个连接上执行命令。

请参见“SET CONNECTION 语句 [Interactive SQL] [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用主机变量

嵌入式 SQL 应用程序使用主机变量以向数据库发送值或者从数据库接收值。主机变量是声明部分中供 SQL 预处理器识别的 C 变量。

声明主机变量

通过将其放置在声明部分中来定义主机变量。用 BEGIN DECLARE SECTION 和 END DECLARE SECTION 语句把常规的 C 变量声明围起来以声明主机变量。

每当在 SQL 语句中使用主机变量时，必须在变量名之前添加冒号 (:)，以便使 SQL 预处理器知道您正在引用（已声明的）主机变量，并将其与语句中允许使用的其它标识符区别开来。

可以使用主机变量代替任何 SQL 语句中的常量值。当数据库服务器执行命令时，从每个主机变量中读取主机变量的值，或将其写入主机变量中。主机变量不能用于代替表或列的名称。

SQL 预处理器不扫描 C 语言代码（声明部分内的除外）。在声明部分内允许使用变量的初始化程序，但不允许使用 **typedef** 类型和结构。

以下示例代码使用 INSERT 命令说明如何使用主机变量。这些变量由程序填充，然后插入到数据库中：

```
/* 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 );
```

嵌入式 SQL 中的数据类型

要在程序和数据库服务器间传输信息，每个数据项都必须有一个数据类型。可使用任何一种受支持的类型创建主机变量。

仅有有限数量的 C 数据类型可作为主机变量。而且，某些主机变量类型没有对应的 C 类型。

在头文件 *sqlca.h* 中定义的宏可用于声明 VARCHAR、FIXCHAR、BINARY、DECIMAL 或 SQLDATETIME 类型的主机变量。这些宏的用法如下：

```
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;
```

预处理器能够识别声明部分中的这些宏，并可将变量视为相应的类型。

以下数据类型受嵌入式 SQL 编程接口的支持：

- **16 位有符号整数。**

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

- **32 位有符号整数。**

```
long int l;
unsigned long int l;
```

- **4 字节浮点数**

```
float f;
```

- **8 字节浮点数**

```
double d;
```

- **压缩十进制数**

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

- **以 NULL 终止并以空白填充的字符串**

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

因为 C 语言数组还必须保存 NULL 终止符，所以 char a[n] 数据类型映射为 CHAR(n - 1) SQL 数据类型（可保存 – 1 个字符）。

指向 char、WCHAR 和 TCHAR 的指针

SQL 预处理器假定**字符指针**指向一个大小为 2049 字节的字符数组，此数组可安全地保存 2048 个字符以及 NULL 终止符。换句话说，将 char* 数据类型映射为 CHAR(2048) SQL 类型。如果不是这样，则应用程序可能会发生内存出错。

如果使用的是 16 位编译器，则要求 2049 个字节可能会使程序堆栈溢出。这时，可改用已声明数组（即使是作为函数的一个参数），以使 SQL 预处理器知道该数组的大小。WCHAR 和 TCHAR 的行为与 char 类似。

- **以 NULL 终止的 UNICODE 或宽字符字符串** 每个字符占用两个字节的空间，所以可能包含 UNICODE 字符。

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

- **以 NULL 终止的、依赖于系统的字符串** 对于使用 UNICODE 作为其字符集的系统（例如 Windows Mobile）而言，TCHAR 等效于 WCHAR；否则，TCHAR 等效于 char。TCHAR 数据类型设计为自动支持这两种系统中的字符串。

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

- **以空格填充的固定长度字符串**


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

- **带有两字节长度字段的可变长度字符串** 在向数据库服务器提供信息时，您必须设置长度字段。从数据库服务器读取信息时，服务器将设置长度字段（不填充空白）。

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

- **带有两字节长度字段的可变长度二进制数据** 在向数据库服务器提供信息时，您必须设置长度字段。在从数据库服务器读取信息时，服务器将设置长度字段。

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

- **SQLDATETIME 结构，对于时间戳的每一部分都有一个字段**

```
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 结构用于检索 DATE、TIME 和 TIMESTAMP 类型（或者可转换为这些类型之一的任意类型）的字段。应用程序常常具有它们自己的格式和日期操作代码。从此结构中读取数据使编程人员可以更轻松地操作该数据。注意，您也可以使用任何字符类型来读取和更新 DATE、TIME 和 TIMESTAMP 字段。

如果您使用 SQLDATETIME 结构将日期、时间或时间戳输入到数据库中，则会忽略 day_of_year 和 day_of_week 成员。

有关 date_format、time_format、timestamp_format 和 date_order 数据库选项的详细信息，请参见“数据库选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

- **DT_LONGVARCHAR** 可变长度长整型字符数据。该宏定义一个如下结构：

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

DECL_LONGVARCHAR 结构可用于表示大于 32KB 的数据。可以一次读取全部数据，也可以使用 GET DATA 语句逐段读取。可以将全部数据一次提供给服务器，也可以通过使用 SET 语句附加到数据库变量中来逐段提供。该数据不是以空值终止的。

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

- **DT_LONGBINARY**　长二进制数据。该宏定义一个如下结构：

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

  DECL_LONGBINARY 结构可用于表示大于 32KB 的数据。可以一次读取全部数据，也可以使用 GET DATA 语句逐段读取。可以将全部数据一次提供给服务器，也可以通过使用 SET 语句附加到数据库变量中来逐段提供。

这些结构在 *install-dir\SDK\Include\sqlca.h* 文件中定义。VARCHAR、BINARY 和 TYPE_DECIMAL 类型包含一个单字符数组，它们对于声明主机变量没有帮助。但是，它们对于动态分配变量或强制转换其它变量的类型十分有用。

DATE 和 TIME 数据库类型

对于各种 DATE 和 TIME 数据库类型，没有相应的嵌入式 SQL 接口数据类型。这些数据库类型可使用 SQLDATETIME 结构或使用字符串来读取和更新。

没有表示 LONG VARCHAR 和 LONG BINARY 数据库类型的嵌入式 SQL 接口数据类型。

主机变量的用法

主机变量可用于以下环境中：

- SELECT、INSERT、UPDATE 或 DELETE 语句中任何允许使用数字或字符串常量的位置。
- SELECT 或 FETCH 语句的 INTO 子句中。
- 在 CONNECT、DISCONNECT 和 SETCONNECT 语句中，主机变量可用于代替用户 ID、口令、连接名或数据库名称。

主机变量*不能*用于代替表名或列名。

主机变量的作用域

主机变量的声明部分可出现在通常可以声明 C 变量的任何地方，包括 C 函数的参数声明部分。C 语言变量有其常规作用域（在定义它们的块中可用）。但是，因为 SQL 预处理器不扫描 C 代码，所以它与 C 语言块无关。

预处理器假定所有的主机变量都是全局的

就 SQL 预处理器而言，主机变量在其声明之后的源模块中都是全局已知的。主机变量不得同名。此规则的唯一例外情况是当两个主机变量类型（包括任何必需的长度）相同时，它们可以同名。

最好是每个主机变量都具有唯一名称。

示例

因为 SQL 预处理器不能分析 C 代码，因此它假定所有主机变量（不管它们是在哪里声明的）在其声明之后都是全局已知的。

```
// 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;
}
```

虽然以上代码可以使用，但它却容易造成混淆，因为 SQL 预处理器在处理 *setManagerID* 中的语句时，要依赖于 *getManagerID* 内部的声明。此段代码应按如下所示重写：

```
// 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 预处理器会看到在 #if 指令中包含的主机变量声明，因为它将忽略这些指令。另一方面，它忽略过程中的声明，因为它们不在 DECLARE SECTION 内部。相反，C 编译器会忽略 #if 指令内部的声明，并使用过程内部的声明。

这些声明能够正常工作，仅仅是因为将同名的变量声明为完全相同的类型。

将表达式用作主机变量

由于 SQL 预处理器不能识别指针或引用表达式，因此主机变量必须使用简单名称。例如，以下语句*不能正常工作*，因为 SQL 预处理器不能理解点运算符。同样的语法在 SQL 中具有不同的含义。

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

虽然不允许使用以上语法，但仍然可以通过以下方法来使用表达式：

- 用 #if 0 预处理器指令括起 SQL 声明部分。因为 SQL 预处理器忽略这些预处理器指令，所以它将读取该声明并将它们用于模块的其余部分。
- 定义与主机变量同名的宏。由于 #if 指令的存在，C 编译器将看不到 SQL 的声明部分，所以不会出现冲突。必须确保将宏解释为相同类型的主机变量。

以下代码演示了此方法，使 SQL 预处理器看不到 *host_value* 表达式。

```
#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
```

因为 SQLPP 处理器忽略条件编译的指令，所以 *host_value* 被视为 *long* 类型的主机变量，随后，在将该名称用作主机变量时将其发出。C/C++ 编译器处理发出的文件，并将用 *my_s.host_field* 替换所有如此使用的该名称。

准备好以上声明后，就可继续按以下方法访问 *host_field* 了。

```
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 );
}
```

可以使用相同方法将其它左值用作主机变量：

- 间接指针

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

- 数组引用

```
my_array[ I ]
```

- 任意复杂的左值

在 C++ 中使用主机变量

在 C++ 类的内部使用主机变量时会出现类似情况。在单独的头文件中声明类通常比较方便。例如，此头文件可能包含以下 *my_class* 声明。

```
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;
```

在此示例中，每个方法都是在嵌入式 SQL 源文件中实现的。只有简单的变量才可用作主机变量。前一节介绍的技术可用于访问类的数据成员。

```
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 );
}
```

以上示例为 SQL 预处理器声明 `this_host_member`，但宏使 C++ 将其转换为 `this->host_member`。预处理器并不知道该变量的类型。许多 C/C++ 编译器不允许重复的声明。`#if` 指令使编译器看不到第二个声明，但 SQL 预处理器可以看到它。

尽管多个声明可能十分有用，但必须确保每个声明将相同的变量名指派给相同的类型。由于预处理器不能完全分析 C 语言，因此它假定每个主机变量在其声明之后都是全局已知的。

使用指示符变量

指示符变量是保存有关特定主机变量补充信息的 C 变量。读取或放置数据时可使用主机变量。使用指示符变量处理 NULL 值。

指示符变量是 short int 类型的主机变量。要检测或指定 NULL 值，可将指示符变量放在 SQL 语句中紧随常规主机变量之后的位置上。

示例

例如，在下面的 INSERT 语句中，**:ind_phone** 是一个指示符变量：

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

指示符变量的值

下表提供了指示符变量用法的概览：

指示符的值	向数据库提供值	从数据库接收值
0	主机变量的值	读取的非 NULL 值。

指示符的值	向数据库提供值	从数据库接收值
-1	NULL 值	读取的 NULL 值

使用指示符变量处理 NULL

不要将 SQL 的 NULL 概念与同名的 C 语言常量相混淆。在 SQL 语言中，NULL 代表未知属性或不适用的信息。C 语言常量表示不指向内存位置的指针值。

在 SQL Anywhere 文档中使用 NULL 时，它指的是以上给出的 SQL 数据库含义。该 C 语言常量称为 null 指针（小写）。

NULL 不同于列的已定义类型的任何值。为了将 NULL 值传递到数据库或接回 NULL 结果，除了常规主机变量外还需要其它条件。指示符变量就用于此目的。

插入 NULL 时使用指示符变量

INSERT 语句可按以下方式使用指示符变量：

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

如果指示符变量的值为 -1，则写入 NULL。如果它具有值 0，则写入 employee_phone 的实际值。

在读取 NULL 时使用指示符变量

从数据库中接收数据时也可使用指示符变量。它们用于指示读取了 NULL 值（指示符为负数）。如果从数据库中读取了 NULL 值，但未提供指示符变量，则生成 SQLE_NO_INDICATOR 错误。

有关在 SQLCA 结构中返回的错误和警告的详细信息，请参见“初始化 SQL 通信区”一节 第 32 页。

读取数据

在嵌入式 SQL 中使用 SELECT 语句实现数据的读取。这包括两种情况：

1. SELECT 语句不返回行或只返回一行。
2. SELECT 语句返回多个行。

读取一行

单行查询最多从数据库中检索一行。单行查询 SELECT 语句可在选择列表之后和 FROM 子句之前有一个 INTO 子句。INTO 子句包含一个主机变量的列表，用来接收每个选择列表项的值。主机变量和选择列表项的数目必须相同。主机变量可以和指示符变量一起使用，以指示 NULL 结果。

当执行 SELECT 语句时，数据库服务器检索结果并将其放在主机变量中。

- 如果查询返回了多行，则数据库服务器返回 SQLE_TOO_MANY_RECORDS 错误。
- 如果查询没有返回任何行，则返回 SQLE_NOTFOUND 警告。

有关在 SQLCA 结构中返回的错误和警告的详细信息，请参见“初始化 SQL 通信区”一节 第 32 页。

示例

例如，如果成功地从 employee 表读取了一行，则以下代码段返回 1；如果该行不存在，则返回 0；如果发生错误，则返回 -1。

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


读取多行

如果查询的结果集中有多行，可使用游标从中检索行。游标是 SQL 查询结果集的句柄或标识符以及该结果集中的位置。

有关游标的介绍，请参见“使用游标”一节 《SQL Anywhere 服务器 - 编程》。

♦ 在嵌入式 SQL 中管理游标

1. 使用 DECLARE 语句声明特定 SELECT 语句的游标。
2. 使用 OPEN 语句打开游标。
3. 使用 FETCH 语句从游标中一次检索一行。
   - 一直读取行，直到返回 SQLE_NOTFOUND 警告。在变量 SQLCODE（在 SQL 通信区结构中定义）中返回错误和警告代码。
4. 使用 CLOSE 语句关闭游标。

UltraLite 应用程序中的游标始终是使用 WITH HOLD 选项打开的。它们从不自动关闭。必须使用 CLOSE 语句显式关闭每个游标。

以下是游标用法的简单示例：

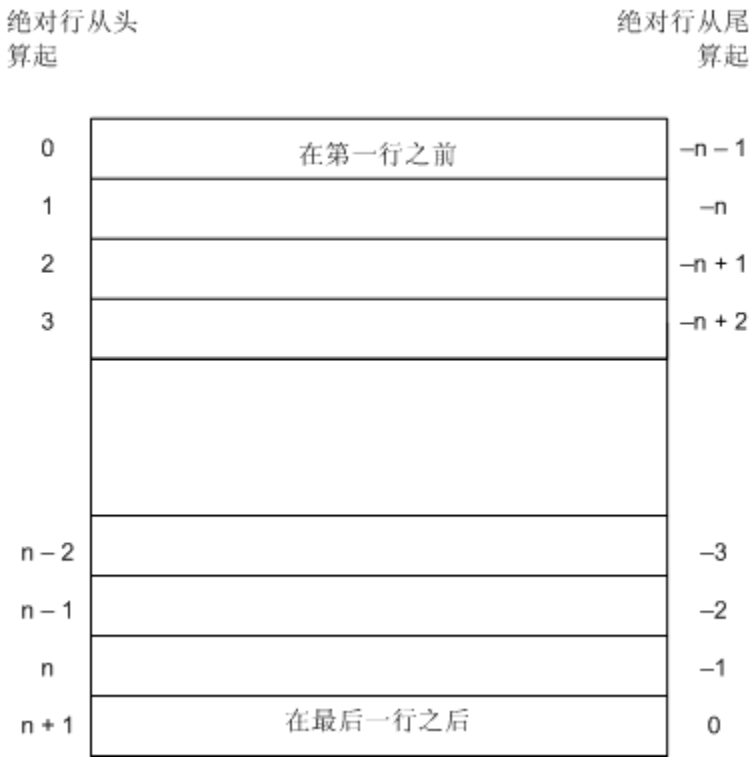
```
void print_employees( void )
{
   int status;
   EXEC SQL BEGIN DECLARE SECTION;
   char name[50];
   char sex;
   char birthdate[15];
   short int ind_birthdate;
   EXEC SQL END DECLARE SECTION;
   /* 1. Declare the cursor. */
   EXEC SQL DECLARE C1 CURSOR FOR
      SELECT emp_fname || ' ' || emp_lname,
                sex, birth_date
      FROM "DBA".employee
      ORDER BY emp_fname, emp_lname;
   /* 2. Open the cursor. */
   EXEC SQL OPEN C1;
   /* 3. Fetch each row from the cursor. */
   for( ;; ) {
      EXEC SQL FETCH C1 INTO :name, :sex,
            :birthdate:ind_birthdate;
      if( SQLCODE == SQLE_NOTFOUND ) {
         break; /* no more rows */
      } else if( SQLCODE < 0 ) {
         break; /* the FETCH caused an error */
      }
      if( ind_birthdate < 0 ) {
         strcpy( birthdate, "UNKNOWN" );
      }
      printf( "Name: %s Sex: %c Birthdate:
               %s\n",name, sex, birthdate );
   }
   /* 4. Close the cursor. */
   EXEC SQL CLOSE C1;
}
```

有关 FETCH 语句的详细信息，请参见“FETCH 语句 [ESQL] [SP]”一节 《SQL Anywhere 服务器 - SQL 参考》。

游标定位

游标定位在以下三个位置之一：

- 在一行上
- 在第一行之前
- 在最后一行之后



游标中行的顺序

通过在定义游标的 SELECT 语句中包括 ORDER BY 子句，可以控制游标中行的顺序。如果忽略此子句，则行的顺序将是不可预知的。

如果没有显式定义顺序，则只能保证在返回 SQLE_NOTFOUND 前，重复读取会将结果集中的每一行返回一次且只返回一次。

重新定位游标

打开游标时，它被定位在第一行之前。FETCH 语句将自动使游标的位置往下移。试图用 FETCH 读取最后一行之后的内容将导致 SQLE_NOTFOUND 错误，可以方便地将该错误用作行顺序处理已完成的信号。

还可将游标重新定位在相对查询结果开始或结尾的绝对位置，也可以相对于当前位置移动游标。UPDATE 和 DELETE 语句有特殊的*定位*版本，可用于更新或删除游标当前位置处的行。如果游标位于第一行之前或最后一行之后，则返回 SQLE_NOTFOUND 错误。

要在使用显式定位时避免不可预知的结果，可在定义游标的 SELECT 语句中包括 ORDER BY 子句。

可以使用 PUT 语句将行插入到游标中。

更新后定位游标

在对某打开的游标所访问的任何信息进行更新之后，最好再次读取并显示这些行。如果游标用于显示单行，则 FETCH RELATIVE 0 将重新读取当前行。如果当前行已被删除，则从游标中读取下一行（或者如果没有其它行，则返回 SQLE_NOTFOUND）。

将临时表用于游标时，在关闭并重新打开游标之前，在基础表中插入的行根本不显示。对于大多数编程人员来说，如果不检查 SQL 预处理器生成的代码，或者如果对于在何种条件下使用临时表不十分了解，则很难确定 SELECT 语句中是否涉及临时表。通常可在 ORDER BY 子句中使用的列上建立索引来避免使用临时表。

有关临时表的详细信息，请参见“在查询处理中使用工作表（使用 All-rows 优化目标）”一节《SQL Anywhere 服务器 - SQL 的用法》。

对非临时表的插入、更新和删除可能影响游标定位。因为 UltraLite 形成游标行时为一次一行（当不使用临时表时），新插入的行中的数据（或新删除的行中删除的数据）可能会影响后面的 FETCH 操作。在从单个表中选择（一部分）行的简单情况下，当插入或更新的行满足 SELECT 语句的选择标准时，该行将出现在游标的结果集中。类似地，以前提供给结果集中的新删除行将不再位于结果集中。

验证用户

UltraLite 数据库都是用缺省用户 ID (DBA) 和缺省口令 (sql) 创建的；必须首先以该初始用户的身份进行连接。必须从现有连接添加新用户。

用户 ID 不能更改；但可添加新用户 ID，然后再删除现有的用户 ID。每个 UltraLite 数据库最多允许四个用户 ID。

在 Palm OS 上，如果希望在用户每次从其它应用程序返回原应用程序时都对用户进行验证，则必须在 PilotMain 例程中包括要求输入用户和口令信息的提示。

用户验证示例

samples-dir\UltraLite\esqlauth 目录中提供了一个完整的示例。以下代码从 *samples-dir\UltraLite\esqlauth\sample.sqc* 中获取。

```
//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;
```

该代码执行以下任务：

1. 通过调用 db_init 启动数据库功能。
2. 尝试使用缺省用户 ID 和口令进行连接。
3. 如果连接尝试成功，则添加新用户。
4. 如果成功添加了新用户，则从 UltraLite 数据库中删除 DBA 用户。
5. 断开连接。这样就向数据库中添加了已更新的用户 ID 和口令。
6. 使用已更新的用户 ID 和口令进行连接。

请参见：

- “ULGrantConnectTo 函数”一节第 271 页
- “ULRevokeConnectFrom 函数”一节第 276 页

对数据进行加密

可以使用 UltraLite 嵌入式 SQL 对 UltraLite 数据库进行加密或模糊处理。

请参见“对数据进行加密”一节第 50 页。

加密

创建 UltraLite 数据库时（例如通过 Sybase Central 创建时），可指定一个可选的加密密钥。加密密钥用于加密数据库。数据库被加密后，随后的所有连接尝试都必须提供该加密密钥。将对照原始的加密密钥检查提供的密钥，仅当密钥匹配时，才会连接成功。

选择一个无法被轻易猜到的加密密钥值。该密钥可以是任意长度，但通常密钥越长越好，因为较短的密钥容易被猜到。组合使用数字、字母和特殊字符会减少他人猜中密钥的几率。

密钥中不要包含分号。不要将密钥本身放在引号中，否则引号将被视为密钥一部分。

♦ 连接到加密的 UltraLite 数据库

1. 在 EXEC SQL CONNECT 语句使用的连接字符串中指定加密密钥。

   使用 key= 连接字符串参数指定加密密钥。

   在每次要连接到该数据库时都必须提供此密钥。丢失或忘记密钥会导致数据库完全无法访问。
2. 处理使用错误密钥打开加密数据库的尝试。

   如果尝试打开加密数据库时提供的密钥不正确，则 db_init 返回 ul_false，并且设置 SQLCODE -840。

更改加密密钥

您可以更改数据库的加密密钥。必须在应用程序已使用现有密钥连接到数据库之后，才能进行更改。

♦ 更改 UltraLite 数据库的加密密钥

- 调用 ULChangeEncryptionKey 函数，提供新密钥作为参数。

  在调用此函数之前，应用程序必须已使用旧密钥连接到了数据库。

  请参见“ULChangeEncryptionKey 函数”一节第 255 页。

模糊处理

♦ 对 UltraLite 数据库进行模糊处理

- 顶替数据库加密的一种方法是指定对数据库进行模糊处理。模糊处理是对数据库中的数据进行简单的掩蔽，用于防止使用低级文件查看实用程序浏览数据库中的数据。模糊处理是一个只能在创建数据库时指定的数据库创建选项。

  请参见“为 UltraLite 选择数据库创建参数”一节 《UltraLite - 数据库管理和参考》 。

向应用程序添加同步

同步是众多 UltraLite 应用程序的核心功能。本节介绍如何向应用程序添加同步。

使 UltraLite 应用程序与最新的统一数据库保持一致的同步逻辑并未包含在应用程序本身中。存储在统一数据库中的同步脚本以及 MobiLink 服务器和 UltraLite 运行时库，一起控制上载更改时如何处理这些更改并确定要下载哪些更改。

概述

每个同步的具体信息由一组同步参数控制。这些参数被收集在一个结构中，后者然后将作为函数调用中的参数提供以进行同步。该方法的基本步骤在各种开发模型中都相同。

♦ 向应用程序添加同步

1. 初始化保存同步参数的结构。

   请参见“初始化同步参数”一节第 51 页。

2. 为应用程序指派参数值。

   请参见“UltraLite 同步流的网络协议选项”一节 《UltraLite - 数据库管理和参考》。

3. 调用同步函数，并提供结构或对象作为函数的参数。

   请参见“调用同步”一节第 52 页。

必须确保同步时没有未提交的更改。

同步参数

在 C/C++ 组件一章中对 ul_synch_info 结构进行了介绍；但该结构的成员同样为嵌入式 SQL 开发所共有。根据您使用的是 ASCII 字符还是宽字符，请参见“ul_synch_info_a 结构”一节第 127 页或“ul_synch_info_w2 结构”一节第 129 页。

有关同步参数的一般说明，请参见“UltraLite 的同步参数”一节 《UltraLite - 数据库管理和参考》。

初始化同步参数

同步参数存储在结构中。

结构的成员在初始化时未进行定义。必须调用特定函数将参数设置为它们的初始值。同步参数在 UltraLite 头文件 *install-dir\SDK\Include\ulglobal.h* 中声明的结构内定义。

♦ 初始化同步参数（嵌入式 SQL）

- 调用 ULInitSynchInfo 函数。例如：

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


设置同步参数

以下代码启动 TCP/IP 同步。MobiLink 用户名是 Betty Best，口令为 TwentyFour，脚本版本为 default，而运行 MobiLink 服务器的主机是 test.internal，端口为 2439：

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

以下代码用于 Palm Computing Platform 上的应用程序，在用户退出应用程序时调用。它允许进行 HotSync 同步，所使用的 MobiLink 用户名为 50、口令为空、脚本版本为 custdb。HotSync 管道通过 TCP/IP 与 MobiLink 服务器进行通信，该服务器运行在管道所在的计算机 (localhost) 上，并使用缺省端口 (2439)：

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

调用同步

如何调用同步的具体过程取决于您的目标平台和同步流。

仅当运行 UltraLite 应用程序的设备能够与 Mobilink 服务器进行通信时，同步过程才能进行。对于某些平台，这意味着需要将设备放在其底座中或者使用适当的电缆将其连接到服务器计算机，以便物理连接此设备。需要在应用程序中加入错误处理代码，以便在无法执行同步时使用。

♦ **调用同步（TCP/IP、TLS、HTTP 或 HTTPS 流）**

- 调用 ULInitSynchInfo 可初始化同步参数，调用 ULSynchronize 可进行同步。

♦ **调用同步 (HotSync)**

- 调用 ULInitSynchInfo 可初始化同步参数，调用 ULSetSynchInfo 可在退出应用程序之前管理同步。

  请参见“ULSetSynchInfo 函数”一节第 282 页。

同步调用需要一个结构来保存一组描述同步具体信息的参数。所使用的特定参数取决于流。

在同步前提交所有更改

同步 UltraLite 数据库时，它不能有未提交的更改。在尝试同步 UltraLite 数据库时，如果在任意连接中有未提交的事务，则同步失败，同时抛出异常，并给出 SQLE_UNCOMMITTED_TRANSACTIONS 错误。此错误代码也在 MobiLink 服务器日志中显示。

有关仅下载同步的详细信息，请参见“Download Only 同步参数”一节 《UltraLite - 数据库管理和参考》。

向应用程序中添加初始数据

许多 UltraLite 应用程序需要使用数据才能开始工作。可通过同步将数据下载到应用程序中。您可能要向应用程序中添加逻辑，以确保第一次运行它时，它首先下载所有必需的数据，然后再执行任何其它操作。

性能提示
如果分阶段开发应用程序，则更容易找出错误。开发原型时，可在应用程序中临时使用 INSERT 语句以提供用于测试和演示的数据。在原型正常工作后，用代码替换临时 INSERT 语句以执行同步。

有关同步开发提示的详细信息，请参见“MobiLink 开发提示”一节 《MobiLink - 服务器管理》。

处理同步通信错误

以下代码说明了如何从嵌入式 SQL 应用程序中处理通信错误：

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

SQLE_COMMUNICATIONS_ERROR 是通信错误的一般错误代码。在 stream_error 同步参数的成员中为应用程序提供了更多有关特定错误的信息。

为使 UltraLite 较小，运行库报告数字而不是报告消息。

监控和取消同步

本节介绍如何从 UltraLite 应用程序监控和取消同步。

监控同步

- 在同步结构 (ul_synch_info) 的 observer 成员中指定回调函数的名称。

- 调用同步函数或方法启动同步。
- 每当同步状态发生更改时，UltraLite 将调用回调函数。下一节将对同步状态进行介绍。

以下代码显示在嵌入式 SQL 应用程序中可以如何实现这一系列任务：

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

处理同步状态信息

监控同步的回调函数将 ul_synch_status 结构作为参数。有关详细信息，请参见“ul_synch_status 结构”一节第 133 页。

ul_synch_status 结构具有以下成员：

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

- **sent.inserts**　到目前为止上载的已插入行数。
- **sent.updates**　到目前为止上载的已更新行数。
- **sent.deletes**　到目前为止上载的已删除行数。
- **sent.bytes**　到目前为止上载的字节数。
- **received.inserts**　到目前为止下载的已插入行数。

- **received.updates** 到目前为止下载的已更新行数。
- **received.deletes** 到目前为止下载的已删除行数。
- **received.bytes** 到目前为止下载的字节数。
- **info** 指向 ul_synch_info 结构的指针。请参见“ul_synch_info_a 结构”一节第 127 页。
- **db_tableCount** 返回数据库中表的数目。
- **table_id** 正在上载或下载的当前表编号（相对于 1）。当没有同步所有表时此数字可能会跳过某些值，并且不一定会增加。
- **table_name[]** 当前表的名称。
- **table_name_w2[]** 当前表的名称（宽字符版本）。此字段只能在 Windows（桌面操作系统和 Mobile）环境中填充。
- **sync_table_count** 返回正在同步的表的数目。
- **sync_table_index** 正在上载或下载的表数，从 1 开始以 **sync_table_count** 值结束。当没有同步所有表时此数字可能会跳过某些值。
- **state** 以下状态之一：
  - **UL_SYNCH_STATE_STARTING** 尚未采取同步操作。
  - **UL_SYNCH_STATE_CONNECTING** 已构建同步流，但尚未打开。
  - **UL_SYNCH_STATE_SENDING_HEADER** 已打开同步流，即将发送标头。
  - **UL_SYNCH_STATE_SENDING_TABLE** 正在发送表。
  - **UL_SYNCH_STATE_SENDING_DATA** 正在发送模式信息或数据。
  - **UL_SYNCH_STATE_FINISHING_UPLOAD** 上载阶段已完成，正在执行提交。
  - **UL_SYNCH_STATE_RECEIVING_UPLOAD_ACK** 正在接收关于上载已完成的确认。
  - **UL_SYNCH_STATE_RECEIVING_TABLE** 正在接收表。
  - **UL_SYNCH_STATE_RECEIVING_DATA** 正在接收模式信息或数据。
  - **UL_SYNCH_STATE_COMMITTING_DOWNLOAD** 下载阶段已完成，正在执行提交。
  - **UL_SYNCH_STATE_SENDING_DOWNLOAD_ACK** 正在发送关于下载已完成的确认。
  - **UL_SYNCH_STATE_DISCONNECTING** 同步流即将关闭。
  - **UL_SYNCH_STATE_DONE** 同步已成功完成。
  - **UL_SYNCH_STATE_ERROR** 同步已完成，但有错误。
  - **UL_SYNCH_STATE_ROLLING_BACK_DOWNLOAD** 下载过程中出现了错误，正在回退下载。

  有关同步过程的详细信息，请参见“同步过程”一节 《MobiLink - 入门》。
- **stop** 将此成员设置为 true 可中断同步。已设置了 SQL 异常 SQLE_INTERRUPTED，且同步已经停止，就像发生了通信错误一样。观察器被调用时*始终*处于 DONE 或 ERROR 状态，以便它可以进行正确的清除。

- **flags** 返回指示有关当前状态的其它信息的当前同步标志。
- **user_data** 返回作为 ULRegisterSynchronizationCallback 函数的参数而传递的用户数据对象。
- **sqlca** 指向连接的活动 SQLCA 的指针。

示例

以下代码说明了一个非常简单的观察器函数：

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

当同步两个表时，此观察器产生以下输出：

```
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 示例

CustDB 示例应用程序中包括一个观察器函数的示例。CustDB 中的该观察器函数提供一个窗口，该窗口显示同步进度，并允许用户取消同步。用户界面组件使该观察器函数成为特定于平台的观察器函数。

CustDB 示例代码位于 *samples-dir\UltraLite\CustDB* 目录中。观察器函数位于 *CustDB* 目录的特定于平台的子目录中。

构建嵌入式 SQL 应用程序

本节介绍 UltraLite 嵌入式 SQL 应用程序的常规构建过程。

本节假设您熟悉整个嵌入式 SQL 开发模型。

常规构建过程

示例代码

可以在 *samples-dir\UltraLite\ESQLSecurity* 目录中找到一个使用此过程的生成文件。

需要单独授予许可的组成部分
ECC 加密和 FIPS 认证的加密需要单独的许可。所有高度加密技术受出口法规约束。

请参见“单独授权的组件”一节 《SQL Anywhere 11 - 简介》。

过程

♦ **构建 UltraLite 嵌入式 SQL 应用程序**

1. 对*每个*嵌入式 SQL 源文件运行 SQL 预处理器。

   SQL 预处理器是 sqlpp 命令行实用程序。它预处理嵌入式 SQL 源文件，生成要编译到应用程序中的 C++ 源文件。

   有关 SQL 预处理器的详细信息，请参见“UltraLite 实用程序的 SQL 预处理器 (sqlpp)”一节 《UltraLite - 数据库管理和参考》。

   小心
   sqlpp 会覆盖输出文件而不考虑其内容。应确保输出文件名不与任何源文件名相同。缺省情况下，*sqlpp* 通过将源文件的后缀更改为 *.cpp* 来构造输出文件名。如果您没有把握，应在源文件的名称之后显式地指定输出文件名。

2. 为所选目标平台编译*每个* C++ 源文件。包括：
   - 每个由 SQL 预处理器生成的 C++ 文件
   - 应用程序所需的任何额外的 C 或 C++ 源文件
3. 链接*所有*这些对象文件以及 UltraLite 运行时库。

为嵌入式 SQL 开发配置开发工具

大多数开发工具使用依赖性模型（有时表示为一个 makefile），在这种模型下，每个源文件上的时间戳与目标文件（大多数情况下是 obj 文件）上的时间戳进行比较，以确定是否需要重新生成目标文件。

对于 UltraLite 开发，对开发项目中任何 SQL 语句的更改都意味着需要将已生成的代码进行重新生成。这些更改不反映在任何单个源文件上的时间戳中，因为 SQL 语句存储在参考数据库中。

本节介绍如何将 UltraLite 应用程序（具体来说就是 SQL 预处理器）开发集成到基于依赖性的构建环境中。提供的特定指导是针对 Visual C++ 的，您可能需要根据自己的开发工具对其进行相应的修改。

CodeWarrior 的 UltraLite 插件会自动向 Palm Computing Platform 开发人员提供此处所述的技术。有关此插件的详细信息，请参见“开发用于 Palm OS 的 UltraLite 应用程序”第 61 页。

SQL 预处理

第一组指导介绍了如何向开发工具中添加指令来运行 SQL 预处理器。

♦ 向基于依赖性的开发工具添加嵌入式 SQL 预处理

1. 将 *.sqc* 文件添加到开发项目中。

   在开发工具中定义开发项目。

2. 为每个 *.sqc* 文件添加一条自定义构建规则。

   - 自定义构建规则应运行 SQL 预处理器。在 Visual C++ 中，构建规则应该包含以下命令（在一行中输入）：

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

     其中 SQLANY11 是指向 SQL Anywhere 安装目录的环境变量。

     有关 SQL 预处理器命令行的完整说明，请参见“UltraLite 实用程序的 SQL 预处理器 (sqlpp)”一节 《UltraLite - 数据库管理和参考》。

   - 将命令的输出设置为 **$(InputName).cpp**。

3. 编译 *.sqc* 文件，并将所生成的 *.cpp* 文件添加到开发项目中。

   即使所生成的文件不是源文件，也应将它们添加到您的项目中，这样才能建立依赖性和构建选项。

4. 为每个生成的 *.cpp* 文件设置预处理器定义。

   - 在 [General] 或 [Preprocessor] 下，向预处理器定义添加 UL_USE_DLL。
   - 在 [Preprocessor] 下，以逗号分隔列表的形式向包含路径添加 *$(SQLANY11)\SDK\Include* 以及所需的其它包含文件夹。

开发用于 Palm OS 的 UltraLite 应用程序

目录

安装用于 CodeWarrior 的 UltraLite 插件 62
在 CodeWarrior 中创建 UltraLite 项目 63
将现有 CodeWarrior 项目转换为 UltraLite 应用程序 64
使用用于 CodeWarrior 的 UltraLite 插件 65
在 CodeWarrior 中构建 CustDB 示例应用程序 66
构建扩展模式应用程序 67
在 UltraLite Palm 应用程序中维护状态（不建议使用） 68
注册 Palm 创建者 ID 70
将 HotSync 同步添加到 Palm 应用程序 71
向 Palm 应用程序添加 TCP/IP、HTTP 或 HTTPS 同步 73
部署 Palm 应用程序 74

您可以使用 CodeWarrior 插件通过嵌入式 SQL 或 UltraLite C++ 简化 UltraLite 应用程序的创建。

该插件在 *install-dir\UltraLite\Palm\68k\cwplugin* 目录中提供。请阅读该目录中的 *readme.txt* 文件。

CodeWarrior 包括 Palm SDK 的一个版本。根据特定目标设备的不同，您可能希望将 Palm SDK 升级为比开发工具中所包括版本更新的版本。

有关受支持的平台的列表，请参见“支持的平台”一节 《SQL Anywhere 11 - 简介》。

有关支持的目标操作系统的列表，请参见“支持的平台”一节 《SQL Anywhere 11 - 简介》。

安装用于 CodeWarrior 的 UltraLite 插件

在 UltraLite 安装过程中，会将用于 CodeWarrior 的 UltraLite 插件的文件放在磁盘上，但是如果没有附加的安装步骤，则无法使用该插件。

♦ 安装用于 CodeWarrior 的 UltraLite 插件

1. 在命令提示符下，更改为 *install-dir\UltraLite\Palm\68k\cwplugin* 目录。
2. 运行 *install.bat* 以将相应的文件复制到您的 CodeWarrior 安装目录中。*install.bat* 文件需要两个参数：
   - CodeWarrior 目录
   - CodeWarrior 版本。

   例如，以下位于一行中的命令会将 CodeWarrior 9 的插件安装在缺省的 CodeWarrior 安装目录中。

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

   如果路径包含任何嵌入的空格，需要将目录用双引号括起来。

卸载 CodeWarrior 插件

可以使用 *uninstall.bat* 从 CodeWarrior 中卸载 UltraLite 插件。*uninstall.bat* 文件需要的参数与上文所述的 *install.bat* 文件需要的参数相同。

在 CodeWarrior 中创建 UltraLite 项目

♦ 在 CodeWarrior 中创建 UltraLite 项目

1. 启动 CodeWarrior。
2. 创建一个新项目：
   a. 从 [**CodeWarrior**] 菜单选择 [**File**] » [**New**]。
   b. 单击 [**Projects**] 选项卡。
   c. 选择 [**Palm OS Application Stationery**]。
   d. 选择项目的名称和位置，然后单击 [**OK**]。
3. 选择 UltraLite 模板。

   UltraLite 插件将以下选项添加到模板列表中：

   - Palm OS UltraLite C++ App
   - Palm OS UltraLite ESQL App

   选择要使用的开发模型，然后单击 [**OK**] 创建项目。

   该模板是嵌入式 SQL 的标准 C 模板，同时也是 C++ 的标准 C++ 模板。
4. 如果要使用嵌入式 SQL，请在 [**UltraLite Preprocessor Panel**] 上为项目配置相关设置。如果要使用 C++，将忽略这些设置。
   a. 在项目窗口 (*.mcp*) 中，单击工具栏上的 [**Settings**] 图标。
   b. 在左窗格的树中，选择 [**Target**] » [**UltraLite Preprocessor**]。输入项目的设置。

预处理

构建嵌入式 SQL 项目时，UltraLite 插件调用 *sqlpp* 将 *.sqc* 文件预处理为 *.c/.cpp* 文件，并且还会将 ESQL 语句转换为 UltraLite 函数调用。

当您在 CodeWarrior 环境中构建 UltraLite 嵌入式 SQL 或 C++ 应用程序时，该插件不会将访问路径添加到 *install-dir\SDK\Include* 和 *install-dir\UltraLite\Palm\68k\lib\cw* 以及 UltraLite 库的 *ulrt.lib* 和 *ulbase.lib* 中。该插件只会将运行 SQL 预处理器时所生成的文件添加到 UltraLite 嵌入式 SQL 应用程序的 CodeWarrior 项目中。

将现有 CodeWarrior 项目转换为 UltraLite 应用程序

如果安装用于 CodeWarrior 的 UltraLite 插件，则在您首次打开旧项目时将要求您转换这些项目。在此转换中，CodeWarrior 设置缺省的 SQL 预处理器设置，并将它们保存在项目文件中。这不会中断那些不使用 SQL 预处理器的项目。如果希望进一步将项目转换为自动调用 SQL 预处理器，则需要执行以下操作：

1. 将 *.sqc* 文件的文件映射条目添加到 [Target] 设置的 [File Mappings] 面板中。

   文件类型是 **TEXT**，编译器的类型是 **UltraLite Preprocessor**。必须取消选中这些文件的所有标志。

2. 对于嵌入式 SQL 应用程序，从 [Files] 视图中删除所有生成的文件。在构建 *.sqc* 文件时，会自动生成并重新添加这些文件。

3. 删除 *.ulg* 文件的所有文件映射。

4. 验证对必需的库文件的引用。

使用用于 CodeWarrior 的 UltraLite 插件

对于嵌入式 SQL，CodeWarrior 的 UltraLite 插件会将 UltraLite 预处理步骤集成到 CodeWarrior 编译模型中。它确保在需要时运行 SQL 预处理器。

使用前缀文件

前缀文件是 CodeWarrior 项目中的所有源文件都必须包括的头文件。应使用 *install-dir\SDK\Include\ulpalmos.h* 作为前缀文件。

如果您有自己的前缀文件，它必须包括 *ulpalmos.h*。*ulpalmos.h* 文件用于定义 UltraLite Palm 应用程序所需的宏，还用于设置 UltraLite 所需的 CodeWarrior 编译器选项。

加密同步
如果要使用 TLS 或 HTTPS 协议实现加密的同步，必须将 *ulrsa.lib*、*ulecc.lib* 或 *ulfips.lib* 和 *gselst.lib* 添加到 CodeWarrior UltraLite 项目中。

在 CodeWarrior 中构建 CustDB 示例应用程序

CustDB 是一个简单的销售状态应用程序。

有关示例数据库模式的图示，请参见“关于 CustDB 示例数据库”一节 《SQL Anywhere 11 - 简介》。

应用程序的文件位于 *samples-dir\UltraLite\CustDB* 目录中。通用文件位于 *CustDB* 目录中。用于 Palm OS 的 CodeWarrior 特定文件位于以下目录中：

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

本节说明如何使用 CodeWarrior 9 构建 CustDB 应用程序。

♦ 使用 CodeWarrior 构建 CustDB 示例应用程序

1. 启动 CodeWarrior IDE。
2. 打开 CustDB 项目文件：
   - 选择 [**File**] » [**Open**]。
   - 打开项目文件 *samples-dir\UltraLite\CustDB\cw\custdb.mcp*。
3. 要构建目标应用程序 (*custdb.prc*)，选择 [**Project**] » [**Make**]。

构建扩展模式应用程序

CodeWarrior 支持一种称为扩展模式的代码生成模式，该模式改进了全局数据的内存使用。如果要使用 CodeWarrior 版本 9，可以对扩展模式使用一个从 A5 开始的跳转表。为此，必须使用 UltraLite 运行时库和 UltraLite 基库的扩展模式版本。这些库的扩展模式版本位于以下位置：

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

扩展模式对将超过 64 KB 全局数据限制的大型应用程序可能有用。扩展模式的限制在于只能通过 HotSync 使用加密同步，因为 UltraLite 的同步安全库不使用扩展模式。

在 UltraLite Palm 应用程序中维护状态（不建议使用）

通过挂起连接而不是关闭连接，可以在应用程序关闭时保存表和游标的状态。

仅当连接对象处于打开状态时，才存储打开的表的当前状态。

每当关闭 UltraLite 应用程序或用户切换到另一个应用程序时，UltraLite 都保存任何打开的游标和表的状态。

1. 当用户返回到应用程序时，请调用适当的打开方法：
   - 对于嵌入式 SQL，请调用以下函数：
     - db_init
     - EXEC SQL CONNECT
   - 对于 C++，请调用以下函数：
     - ULSqlca.Initialize
     - ULInitDatabaseManager
     - OpenConnection
2. 通过检查 SQLCODE 是否为 SQLE_CONNECTION_RESTORED，确认连接是否已恢复正常。
3. 对于游标对象（包括生成的结果集类的实例），可以执行以下操作之一：
   - 请确保在用户离开应用程序时关闭对象，并在下次需要该对象时调用 Open。如果选择此选项，不恢复对象的当前位置。
   - 在用户离开应用程序时不关闭对象，并在下次需要访问该对象时调用 Reopen。这样就保留了对象的当前位置，但是当用户使用其它应用程序时，该应用程序会占用 Palm 中的更多内存。
4. 对于表对象（包括生成的表类的实例），无法保存位置。在用户关闭应用程序之前，必须先关闭表对象；在用户再次需要它们时可调用 Open 打开它们。请不要对表对象使用 Reopen。

关闭连接将回退任何未提交的事务。如果不关闭连接对象，将保存（而非提交）任何未完成的事务，以便在重新启动应用程序时，这些事务将出现并可以提交或回退它们。不会同步未提交的更改。

在 UltraLite Palm 应用程序中恢复状态（不建议使用）

在 Palm OS 上重新启动应用程序时，UltraLite 将恢复那些在应用程序最近一次关闭时未显式关闭的所有游标或表的状态。

在 Palm OS 上保存、检索和清除加密密钥

在 Palm OS 上，每当用户切换到其它应用程序，系统都会自动关闭原来的应用程序。因此，如果在 Palm OS 上加密 UltraLite 数据库，则*每当*用户切换回应用程序时，都会提示用户重新输入密钥。

♦ 避免重新输入加密密钥

1. 将加密密钥作为 Palm 功能保存在动态内存中。

   功能的索引是根据创建者和功能编号编制的。随后，应用程序会接受其创建者 ID 或 NULL 以及功能编号或 NULL，以保存和检索加密密钥。

2. 编写应用程序，使其在重新启动时检索密钥。

   注意
   因为在设备重置时会清除加密密钥，所以，此时密钥的检索将失败。在这种情况下会提示用户重新输入密钥。

   以下示例代码（嵌入式 SQL）演示如何保存和检索加密密钥：

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

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

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

3. 使用菜单项清除加密密钥，从而保证设备安全。

   以下代码示例说明了满足这一目标的方法：

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

另请参见

- 对于 UltraLite.NET： “ULConnectionParms 类”一节 《UltraLite - .NET 编程》
- 对于 UltraLite C++： “UltraLite_Connection_iface 类”一节第 148 页
- “UltraLite DBKEY 连接参数”一节 《UltraLite - 数据库管理和参考》
- 对于 UltraLite for M-Business Anywhere： “数据库加密和模糊处理”一节 《UltraLite - M-Business Anywhere 编程》
- “UltraLite obfuscate 创建参数”一节 《UltraLite - 数据库管理和参考》

注册 Palm 创建者 ID

用于 Palm OS 的 UltraLite 应用程序与所有 Palm OS 应用程序一样，都需要创建者 ID。创建应用程序时将此创建者 ID 指派给应用程序，同时，如果要使用 HotSync 同步，则需要使用 HotSync 管理器注册该创建者 ID 以供 MobiLink 同步使用。

有关将创建者 ID 指派给应用程序的信息，请参见您的开发工具文档。有关使用 HotSync 管理器注册创建者 ID 的详细信息，请参见“Palm OS 上的 HotSync”一节 《UltraLite - 数据库管理和参考》。

创建者 ID 是一个由 1 到 4 个字符组成的字符串。第一个字符应为大写字母，因为 Palm OS 保留第一个小写字母以供 PALM OS 系统使用。

将 HotSync 同步添加到 Palm 应用程序

HotSync 同步在关闭 UltraLite 应用程序时发生。它由 HotSync 启动。

如果使用 HotSync，请先调用 ULSetSynchInfo 进行同步，然后再关闭应用程序。请不要使用 ULSynchronize 或 ULConnection.Synchronize 进行 HotSync 同步。

要从应用程序启用 HotSync 同步，必须添加执行以下步骤的代码：

1. 准备一个 ul_synch_info 结构。
2. 调用 ULSetSynchInfo 函数，提供 ul_synch_info 结构作为参数。

   在用户离开 UltraLite 应用程序时调用此函数。请务必确保提交了所有未完成的操作，然后再调用 db_fini。ul_synch_info.stream 参数将被忽略，因此无需设置它。

   例如：

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

ULSetSynchInfo( &sqlca, &info );

if( !db.Close( ) ) {
  return( false );
}
```

3. 调用 db_fini。

请参见“在 UltraLite Palm 应用程序中维护状态（不建议使用）”一节第 68 页和“UltraLite 的同步参数”一节 《UltraLite - 数据库管理和参考》。

UltraLite 应用程序的 HotSync 同步需要 UltraLite HotSync 管道。如果在关闭 Palm 应用程序时有未提交的事务并在此时进行同步，则管道会因数据库中有未提交的更改而导致同步失败。

指定流参数

ul_synch_info 结构中的同步流参数控制着与 MobiLink 服务器的通信。对于 HotSync 同步，UltraLite 应用程序不直接与 MobiLink 服务器通信，而是与 HotSync 管道通信。

通过提供同步流参数，可以使用以下方式之一控制 MobiLink 管道的行为：

- 在传递给 ULSetSynchInfo 的 ul_synch_info 结构的 stream_parms 成员中提供所需信息。

  有关可用值的列表，请参见“UltraLite 同步流的网络协议选项”一节 《UltraLite - 数据库管理和参考》。

- 为 stream_parms 成员提供一个空值。随后，MobiLink 管道会在其所在计算机的 *ClientParms* 注册表条目中，搜索有关如何连接到 MobiLink 服务器的信息。

  注册表项中指定流和流参数的形式与 ul_synch_info 结构 stream_parms 字段中的相同。

请参见“UltraLite 同步参数和网络协议选项” 《UltraLite - 数据库管理和参考》。

另请参见

- “Palm OS 上的 HotSync”一节 《UltraLite - 数据库管理和参考》

向 Palm 应用程序添加 TCP/IP、HTTP 或 HTTPS 同步

本节说明如何将 TCP/IP、HTTP 或 HTTPS 同步添加到 Palm 应用程序中。

有关如何向 UltraLite 应用程序添加同步的一般说明，请参见“向 UltraLite 应用程序添加同步”一节 《UltraLite - 数据库管理和参考》。

Palm OS 上的传送层安全性

传送层安全性可用于使用 CodeWarrior 生成的 Palm 应用程序。

有关传送层安全性的详细信息，请参见“加密 MobiLink 客户端/服务器通信”一节 《SQL Anywhere 服务器 - 数据库管理》。

Palm 设备可以使用 TCP/IP、HTTP 或 HTTPS 通信进行同步，方法是将 ul_synch_info 结构的 stream 成员设置为适当的流，然后调用 ULSynchronize 或 ULConnection.Synchronize 来执行同步。

在使用 TCP/IP、HTTP 或 HTTPS 同步时，db_init 或 db_fini 在退出和激活应用程序时分别保存和恢复应用程序的状态，但不参与同步。

在关闭应用程序之前，使用 ULSetSynchInfo 设置同步信息，提供 ul_synch_info 结构作为参数。

在 Palm 设备上使用 TCP/IP、HTTP 或 HTTPS 同步时，必须在 ul_synch_info 结构的 stream_parms 成员中显式指定主机名或 IP 地址。指定 NULL 缺省为 `localhost`，它代表设备，而不是主机。

有关 ul_synch_info 结构的详细信息，请参见“UltraLite 同步流的网络协议选项”一节 《UltraLite - 数据库管理和参考》。

部署 Palm 应用程序

本节说明部署 Palm 应用程序的以下几个方面：

- 部署应用程序。

  请参见“部署应用程序”一节第 74 页。
- 部署用于 HotSync 的 UltraLite 同步管道。

  请参见“Palm OS 上的 HotSync”一节 《UltraLite - 数据库管理和参考》。
- 部署 UltraLite 数据库的初始副本。

  请参见“部署 UltraLite 数据库”一节第 74 页。

在 Palm 设备上安装 UltraLite 应用程序就像安装任何其它 Palm OS 应用程序一样。

部署应用程序

♦ 在 Palm 设备上安装应用程序

1. 打开 Palm Desktop Organizer Software 中附带的 [Install Tool]。
2. 选择 [**Add**] 并指定已编译应用程序（*.prc* 文件）的位置。
3. 关闭 [Install Tool]。
4. 使用 HotSync 实用程序将应用程序复制到 Palm 设备。

部署 MobiLink 同步管道

对于使用 HotSync 同步的应用程序，每个最终用户的台式计算机上都必须安装 MobiLink 同步管道。

有关安装 MobiLink 同步管道的详细信息，请参见“Palm OS 上的 HotSync”一节 《UltraLite - 数据库管理和参考》。

部署 UltraLite 数据库

如果在没有数据库的情况下部署应用程序，该应用程序必须包含相对复杂的代码以创建数据库。建议先在 Windows 桌面操作系统上创建一个初始数据库，然后将数据库文件复制到 Palm 设备。Sybase Central（或 ulcreate 实用程序）可用于创建一个初始数据库。然后，用户在第一次同步时必须获取数据的初始副本。可以使用 uldbutil 实用程序将 UltraLite 数据库备份到 PC 上。要使用包含数据的初始数据库部署多个 UltraLite 数据库，可以先执行初始同步，然后备份 UltraLite 数据库。数据库可以部署在其它设备上，这样就不需要执行初始同步了。

请参见“Palm OS 的 UltraLite 数据管理实用程序 (ULDBUtil)”一节 《UltraLite - 数据库管理和参考》。

如果您使用 HotSync 同步，则每个终端用户也必须将同步管道安装到他们的台式计算机上。

有关安装同步管道的详细信息，请参见“部署 UltraLite HotSync 管道”一节 《UltraLite - 数据库管理和参考》。

如果使用 HotSync 部署数据库，HotSync 会在数据库上设置一个备份位。设置此备份位后，每次同步时，整个数据库都会备份到台式计算机上。这种情况通常并不适合于 UltraLite 数据库。当 UltraLite 应用程序启动时，它检查 Palm 数据存储以查看其备份位是否设置为 true。如果设置为 true，则将其清除。如果未设置，则不做任何更改。

如果希望备份位保持设置为 true，可以在数据库连接字符串中设置存储参数 palm_allow_backup。

开发用于 Windows Mobile 的 UltraLite 应用程序

目录

选择链接运行时库的方式 78
构建 CustDB 示例应用程序 79
存储持久数据 81
部署 Windows Mobile 应用程序 82
部署使用 ActiveSync 的应用程序 83
为应用程序指派类名 84
在 Windows Mobile 上进行同步 86
示例 eMbedded Visual C++ 项目 89

Microsoft eMbedded Visual C++ 可用于针对 Windows Mobile 环境开发应用程序。此开发环境可以作为 eMbedded Visual Tools 的一部分从 Microsoft 获取。

可以从 Microsoft Developer Network 网站下载 eMbedded Visual C++，网址是 http://msdn.microsoft.com/。

面向 Windows Mobile 的应用程序应使用 wchar_t 的缺省设置，并针对 *install-dir\ultralite\ce\arm.50\lib* \ 中的 UltraLite 运行时库进行链接。

有关 Windows Mobile 开发所支持的主机平台和开发工具的列表，以及支持的目标 Windows Mobile 平台的列表，请参见“支持的平台”一节 《SQL Anywhere 11 - 简介》 。

可以在大多数 Windows Mobile 目标平台上的模拟器下测试应用程序。

选择链接运行时库的方式

Windows Mobile 支持动态链接库。在链接时，可以选择使用导入库将 UltraLite 应用程序链接到运行时 DLL，或者使用 UltraLite 运行时库静态链接应用程序。

性能提示

如果目标设备上只有一个 UltraLite 应用程序，则静态链接库使用较少的内存。如果目标设备上有多个 UltraLite 应用程序，则使用 DLL 可能更节约内存。

如果通过较慢的链接重复地将 UltraLite 应用程序下载到某个设备，则可能需要使用 DLL，这样可以在初始下载后将下载的可执行文件的大小最小化。

♦ 使用 UltraLite 运行时 DLL 构建和部署应用程序

1. 预处理代码，然后使用 UL_USE_DLL 编译输出。
2. 使用 UltraLite 导入库链接应用程序。
3. 将应用程序可执行文件和 UltraLite 运行时 DLL 都复制到目标设备。

构建 CustDB 示例应用程序

CustDB 是一个简单的销售状态应用程序。它位于 SQL Anywhere 安装目录的 *samples-dir\UltraLite* 子目录中。通用文件位于 *CustDB* 子目录中。特定于 Windows Mobile 的文件位于 *CustDB* 的 *EVC* 子目录中。

CustDB 应用程序作为 eMbedded Visual C++ 3.0 项目提供。

有关示例数据库模式的图示，请参见“关于 CustDB 示例数据库”一节 《SQL Anywhere 11 - 简介》。

♦ 构建 CustDB 示例应用程序

1. 启动 eMbedded Visual C++。
2. 打开对应于您的 eMbedded Visual C++ 版本的项目文件：
   - 对于 eVC 3.0 是 *samples-dir\UltraLite\CustDB\EVC\EVCCustDB.vcp*。
   - 对于 eVC 4.0 是 *samples-dir\UltraLite\CustDB\EVC40\EVCCustDB.vcp*。
3. 选择 [**Build**] » [**Set Active Platform**] 来设置目标平台。
   - 按照您的需要设置平台。
4. 选择 [**Build**] » [**Set Active Configuration**] 来选择配置。
   - 按照您的需要设置活动配置。
5. 如果只是构建用于 Pocket PC x86em 模拟器平台的 CustDB，可以：
   - 选择 [**Project**] » [**Settings**]。
   - 在 [**Link**] 选项卡上的 [**Object/library Modules**] 字段中，将 UltraLite 运行时库条目更改为 *emulator30* 目录，而不是更改为 *emulator* 目录。
6. 构建该应用程序：
   - 按 **F7** 或选择 [**Build**] » [**Build EVCCustDB.exe**] 来构建 CustDB。

     在 eMbedded Visual C++ 构建该应用程序后，它自动尝试将该应用程序上载到远程设备。
7. 启动 MobiLink 服务器：
   - 要启动 MobiLink 服务器，从 [开始] 菜单中选择 [**程序**] » [**SQL Anywhere 11**] » [**MobiLink**] » [**同步服务器示例**]。
8. 运行 CustDB 应用程序：

   运行 CustDB 应用程序之前，必须将 custdb 数据库复制到设备的根文件夹。将名为 *samples-dir\UltraLite\CustDB\custdb.udb* 的数据库文件复制到设备的根目录。

   按 Ctrl+F5 或选择 [**Build**] » [**Execute CustDB.exe**]。

文件夹位置和环境变量
示例项目在任何可能的位置都会使用环境变量。您可能需要对项目进行调整才能正确地构建应用程序。如果遇到问题，请尝试在 Microsoft Visual C++ 文件夹中搜索丢失的文件，并添加相应的目录设置。

对于嵌入式 SQL，构建过程使用 SQL 预处理器 *sqlpp* 将文件 *CustDB.sqc* 预处理为文件 *CustDB.cpp*。对于其中所有嵌入式 SQL 都可限制为一个源模块的较小 UltraLite 应用程序来说，此单步过程非常有用。在比较大的 UltraLite 应用程序中，需要使用多个 *sqlpp* 调用。

请参见“构建嵌入式 SQL 应用程序”一节第 58 页。

存储持久数据

UltraLite 数据库存储在 Windows Mobile 文件系统中。缺省文件是 *\UltraLiteDB\ul_<project>.udb*，其中 *project* 被截断为八个字符。可以使用 **file_name** 连接参数指定基于文件的持久存储区的完整路径名，从而覆盖此选项。

UltraLite 运行库不对 **file_name** 参数进行替换。如果必须创建目录才能使文件名有效，那么应用程序必须确保在调用 **db_init** 之前创建所有目录。

例如，可以通过扫描存储卡并在名称前面加上与存储卡相对应的目录名的方式来利用闪存存储卡。例如，

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


部署 Windows Mobile 应用程序

在编译用于 Windows Mobile 的 UltraLite 应用程序时，可以静态或动态地链接 UltraLite 运行时库。如果动态地进行链接，则必须将所使用平台的 UltraLite 运行时库复制到目标设备中。

♦ 使用 UltraLite 运行时 DLL 构建和部署应用程序

1. 预处理代码，然后使用 UL_USE_DLL 编译输出。
2. 使用 UltraLite 导入库链接应用程序。
3. 将应用程序可执行文件和 UltraLite 运行时 DLL 都复制到目标设备。

   UltraLite 运行时 DLL 位于 SQL Anywhere 安装目录的 *\ultralite\ce* 子目录下芯片特定的目录中。

要为 Windows Mobile 模拟器部署 UltraLite 运行时 DLL，请将该 DLL 放在 Windows Mobile 工具目录下的相应子目录中。以下目录是 Pocket PC 模拟器的缺省设置：

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


部署使用 ActiveSync 的应用程序

使用 ActiveSync 同步的应用程序必须向 ActiveSync 注册并复制到设备上。请参见“使用 ActiveSync 管理器注册应用程序”一节 《UltraLite - 数据库管理和参考》。

还必须安装用于 ActiveSync 的 MobiLink 提供程序。请参见“为 UltraLite 部署 ActiveSync 提供程序”一节 《UltraLite - 数据库管理和参考》。

为应用程序指派类名

在注册与 ActiveSync 一起使用的应用程序时，必须提供窗口类名。指派类名的工作是在开发阶段进行的，应用程序开发工具文档是关于该主题的信息的主要来源。

Microsoft 基础类 (MFC) 对话框具有一个一般类名 **Dialog**，系统中的所有对话框都共享该名称。本节介绍在使用 MFC 和 eMbedded Visual C++ 时，如何为应用程序指派不同的类名。

♦ 使用 eMbedded Visual C++ 为 MFC 应用程序指派窗口类名

1. 基于缺省类为对话框创建和注册自定义窗口类。

   向应用程序的启动代码中添加以下代码。这些代码必须在创建任何对话框之前执行：

   ```
   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" );
   }
   ```

   其中 *MY_APP_CLASS* 是应用程序的唯一类名。

2. 确定哪个对话框是应用程序的主对话框。

   如果项目是用 [MFC 应用程序向导] 创建的，则它可能是名为 **CMyAppDlg** 的对话框。

3. 查找并记录主对话框的资源 ID。

   资源 ID 是常规格式为 IDD_MYAPP_DIALOG 的常数。

4. 确保主对话框在应用程序运行时始终处于打开状态。

   向应用程序的 **InitInstance** 函数中添加以下行。该行能够确保在主对话框 **dlg** 关闭后，应用程序也会关闭。

   ```
   m_pMainWnd = &dlg;
   ```

   有关详细信息，请参见有关 **CWinThread::m_pMainWnd** 的 Microsoft 文档。

   如果该对话框在应用程序打开期间不处于打开状态，则必须还更改其它对话框的窗口类。

5. 保存所做更改。

   如果 Embedded Visual C++ 处于打开状态，则保存所做更改，然后关闭项目和工作区。

6. 修改项目的资源文件。

   - 使用文本编辑器（如记事本）打开资源文件（其扩展名为 *.rc*）。

   找到主对话框的资源 ID。

   - 将主对话框的定义更改为使用新窗口类，如下例所示。应进行的*唯*一更改是添加 **CLASS** 行：

     ```
     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
```

其中 *MY_APP_CLASS* 是以前使用的窗口类的名称。

- 保存 *.rc* 文件。

7. 重新打开 eMbedded Visual C++ 并装载该项目。
8. 添加捕获同步消息的代码。

   请参见“添加 ActiveSync 同步 (MFC)”一节第 87 页。

在 Windows Mobile 上进行同步

Windows Mobile 上的 UltraLite 应用程序可通过以下流类型进行同步：

- **ActiveSync** 请参见“向应用程序添加 ActiveSync 同步”一节第 86 页。
- **TCP/IP** 请参见“在 Windows Mobile 上进行 TCP/IP、HTTP 或 HTTPS 同步”一节第 88 页。
- **HTTP** 请参见“在 Windows Mobile 上进行 TCP/IP、HTTP 或 HTTPS 同步”一节第 88 页。

在 Windows Mobile 上，当初始化时，*user_name* 和 *stream_parms* 参数必须用 **UL_TEXT()** 宏括起来，因为编译环境为 Unicode 宽字符。

有关同步参数的详细信息，请参见“UltraLite 的同步参数”一节 《UltraLite - 数据库管理和参考》。

向应用程序添加 ActiveSync 同步

ActiveSync 是 Microsoft 开发的软件，用于处理运行 Windows 的台式计算机与已连接的 Windows Mobile 手持式设备之间的数据同步。UltraLite 支持 ActiveSync 3.5 版本和更高版本。

本节介绍如何向应用程序添加 ActiveSync 提供程序，以及如何在最终用户的计算机上注册与 ActiveSync 一起使用的应用程序。

如果使用 ActiveSync，则只有 ActiveSync 本身才能启动同步操作。当设备放置在底座中或者从 ActiveSync 窗口选择 [同步] 命令时，ActiveSync 自动启动同步操作。如果应用程序尚未运行，MobiLink 提供程序将启动应用程序，并向该应用程序发送消息。

有关设置 ActiveSync 同步的详细信息，请参见“部署使用 ActiveSync 的应用程序”一节第 83 页。

ActiveSync 提供程序使用 **wParam** 参数。**wParam** 的值为 1 表示用于 ActiveSync 的 MobiLink 提供程序已启动了应用程序。应用程序在完成同步后必须自行关闭。如果当用于 ActiveSync 的 MobiLink 提供程序调用应用程序时，该应用程序已经在运行，则 **wParam** 为 0。如果应用程序想继续运行，它可以忽略 **wParam** 参数。

要确定哪个平台支持该提供程序，请参见 SQL Anywhere 组件支持平台。

添加同步的具体步骤取决于是直接使用 Windows API 还是使用 Microsoft 基础类。下面介绍这两种开发模型。

添加 ActiveSync 同步 (Windows API)

如果直接针对 Windows API 进行编程，则必须在应用程序的 **WindowProc** 函数中处理来自 MobiLink 提供程序的消息，使用 **ULIsSynchronizeMessage** 函数确定应用程序是否已收到消息。

下面是如何处理该消息的一个示例：

```
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;
}
```

其中 **DoSync** 是实际调用 ULSynchronize 的函数。

请参见“ULIsSynchronizeMessage 函数”一节第 273 页。

添加 ActiveSync 同步 (MFC)

如果使用 Microsoft 基础类开发应用程序，则可在主对话框类或应用程序类中捕获同步消息。下面介绍这两种方法。

应用程序必须创建和注册自定义窗口类名才能发出通知。请参见“为应用程序指派类名”一节第 84 页。

♦ 在主对话框类中添加 ActiveSync 同步

1. 添加注册消息，并声明消息处理程序。

   在主对话框的源文件（名称为 *CMyAppDlg.cpp* 格式）中查找消息映射。使用 **static** 添加注册消息，并使用 ON_REGISTERED_MESSAGE 声明消息处理程序，如下面的示例中所示：

   ```
   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. 实现消息处理程序。

   将具有以下签名的方法添加到主对话框类。每当用于 ActiveSync 的 MobiLink 提供程序请求应用程序进行同步时，该方法都会自动执行。此方法应调用 **ULSynchronize**。

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

   此函数的返回值应为 0。

   有关处理同步消息的详细信息，请参见“ULIsSynchronizeMessage 函数”一节第 273 页。

♦ **在应用程序类中添加 ActiveSync 同步**

1. 打开应用程序类的 [**Class Wizard**]。
2. 在 [**Messages**] 列表中，选中 [**PreTranslateMessage**]，然后单击 [**Add Function**]。
3. 单击 [**Edit Code**]。[PreTranslateMessage] 函数出现。将它更改为如下内容：

```
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);
}
```

其中 **DoSync** 是实际调用 ULSynchronize 的函数。

有关处理同步消息的详细信息，请参见“ULIsSynchronizeMessage 函数”一节第 273 页。

在 Windows Mobile 上进行 TCP/IP、HTTP 或 HTTPS 同步

对于 TCP/IP、HTTP 或 HTTPS 同步，应用程序控制何时发生同步。这种应用程序应提供一个菜单项或用户界面控件，以便用户能够请求进行同步。

示例 eMbedded Visual C++ 项目

samples-dir\UltraLite\CEStarter 目录下有一个 eMbedded Visual C++ 示例项目。工作区文件是 *samples-dir\UltraLite\CEStarter\ul_wceapplication.vcw*。

当准备使用 eMbedded Visual C++ 开发 UltraLite 应用程序时，应对项目设置进行以下更改。CEStarter 应用程序中已经进行了这些更改。

- 编译器设置：
  - 向包含路径添加 *$(SQLANY11)\SDK\Include*。
  - 定义相应的编译器指令。例如，应当为 eMbedded Visual C++ 项目定义 UNDER_CE 宏。
- 链接器设置：
  - 添加 "*$(SQLANY11)\ultralite\ce\processor\lib\ulrt.lib*"

    其中 *processor* 是应用程序的目标处理器。
  - 添加 *winsock.lib*。
- *.sqc* 文件（仅限于嵌入式 SQL）：
  - 将 *ul_database.sqc* 和 *ul_database.cpp* 添加到项目。
  - 为 *.sqc* 文件添加以下自定义构建步骤：

    ```
    "$(SQLANY11)\Bin32\sqlpp" -q $(InputPath) ul_database.cpp
    ```
  - 将输出文件设置为 *ul_database.cpp*。
  - 禁止 *ul_database.cpp* 使用预编译头。

API 参考

本节为 UltraLite C/C++ 程序员提供 API 参考资料。

UltraLite C/C++ 公共 API 参考 93
UltraLite C++ 组件 API 123
嵌入式 SQL API 参考 249
UltraLite ODBC API 参考 285

UltraLite C/C++ 公共 API 参考

目录

ULRegisterErrorCallback 的回调函数 94
ULRegisterSQLPassthroughCallback 的回调函数 95
MLFileTransfer 函数 97
ULCreateDatabase 函数 100
ULEnableEccSyncEncryption 函数 102
ULEnableFIPSStrongEncryption 函数 103
ULEnableHttpSynchronization 函数 104
ULEnableHttpsSynchronization 函数 105
ULEnableRsaFipsSyncEncryption 函数 106
ULEnableRsaSyncEncryption 函数 107
ULEnableStrongEncryption 函数 108
ULEnableTcpipSynchronization 函数 109
ULEnableTlsSynchronization 函数 110
ULEnableZlibSyncCompression 函数 111
ULInitDatabaseManager 112
ULInitDatabaseManagerNoSQL 113
ULRegisterErrorCallback 函数 114
ULRegisterSQLPassthroughCallback 116
ULRegisterSynchronizationCallback 118
用于 UltraLite C/C++ 应用程序的宏和编译器指令 119

本节列出可与嵌入式 SQL 或 C++ 接口一起使用的函数和宏。本节中介绍的大多数函数都需要用到“了解 SQL 通信区”一节第 5 页中介绍的 SQL 通信区。

ULRegisterErrorCallback 的回调函数

处理 UltraLite 运行时向您的应用程序发出的错误。

有关使用此技术处理错误的说明，请参见“ULRegisterErrorCallback 函数”一节第 114 页。

语法

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

参数

- **error-callback-function** 函数的名称。必须将此名称提供给 ULRegisterErrorCallback。
- **sqlca** 指向 SQL 通信区 (SQLCA) 的指针。

  SQLCA 将 SQL 代码包含在 `sqlca->sqlcode` 中。已从 SQLCA 中检索出所有错误参数，并存储在 *buffer* 中。

  此 sqlca 指针不一定指向应用程序中的 SQLCA，因此不能用于回调到 UltraLite。它仅用于将 SQL 代码传递给回调函数。

  在 C++ 组件中，请使用 Sqlca.GetCA 方法。
- **user_data** 提供给 ULRegisterErrorCallback 的用户数据。UltraLite 不会以任何方式更改此数据。由于可以从应用程序的任何位置调用回调函数，因此 user_data 参数可以代替创建全局变量。
- **buffer** 注册回调函数时提供的缓冲区。UltraLite 用一个字符串填充缓冲区，该字符串中包含错误消息的任何替代参数。为了使 UltraLite 尽可能小，UltraLite 不提供错误消息本身。替代参数取决于特定的错误。有关 SQL 错误的错误参数的详细信息，请参见“SQL Anywhere 错误消息” 《错误消息》。

返回值

返回以下操作之一：

- **UL_ERROR_ACTION_CANCEL** 取消引发错误的操作。
- **UL_ERROR_ACTION_CONTINUE** 继续执行操作，忽略引发错误的操作。
- **UL_ERROR_ACTION_DEFAULT** 行为与没有错误回调时一样。
- **UL_ERROR_ACTION_TRY_AGAIN** 重试引发错误的操作。

另请参见

- “ULRegisterErrorCallback 函数”一节第 114 页
- “按 Sybase 错误代码排序的 SQL Anywhere 错误消息”一节 《错误消息》

ULRegisterSQLPassthroughCallback 的回调函数

此回调函数可在 SQL 直通脚本执行期间提供脚本执行进度（状态）。

语法

void ul_sql_passthrough_observer_fn(*ul_sql_passthrough_status * status*);

参数

- **ul_sql_passthrough_status** 提供脚本的当前执行状态，包括状态、要执行的脚本数、当前正在执行的脚本以及注册调用中提供的任何用户数据。

示例

进度观察器回调函数定义如下：

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

ul_sql_passthrough_status 结构定义如下：

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

进度观察器回调函数通过以下方法注册：

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

以下为示例回调函数以及调用 ULRegisterSQLPassthroughCallback 的代码：

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

```
        }
    }

    int main() {
        ULSqlca sqlca;

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

返回值

返回以下操作之一 (*** Are these valid? ***)：

- **UL_ERROR_ACTION_CANCEL** 取消引发错误的操作。
- **UL_ERROR_ACTION_CONTINUE** 继续执行，忽略引发错误的操作。
- **UL_ERROR_ACTION_DEFAULT** 行为与没有错误回调时一样。
- **UL_ERROR_ACTION_TRY_AGAIN** 重试引发错误的操作。

另请参见

- “ULRegisterErrorCallback 函数”一节第 114 页
- “按 Sybase 错误代码排序的 SQL Anywhere 错误消息”一节 《错误消息》

MLFileTransfer 函数

使用 MobiLink 接口从 MobiLink 服务器下载文件。

语法

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

参数

info 包含文件传输信息的结构。

ML 文件传输参数

ML 文件传输参数是作为一个参数传送到 MLFileTransfer 函数的结构的成员。ml_file_transfer_info 结构在头文件 *mlfiletransfer.h* 中定义。按如下方法指定此结构中的各个字段：

filename 必需。从运行 MobiLink 的服务器传输的文件名。在缺省情况下搜索根目录之前，MobiLink 首先搜索 *username* 子目录。请参见“-ftr 选项”一节 《MobiLink - 服务器管理》。

如果找不到文件，则在错误字段中设置错误信息。文件名中不能包括任何驱动器或路径信息，否则 MobiLink 找不到它。

dest_path 用于存储下载文件的本地路径。如果此参数为空（缺省值），则将下载文件存储在当前目录中。

- 在 Windows Mobile 上，如果 dest_path 为空，则将文件存储在设备的根 (\) 目录中。
- 在桌面操作系统中，如果 dest_path 为空，则将文件存储在用户的当前目录中。
- 在 Palm OS 上，当下载到设备外部存储器上时，将 dest_path 加上前缀 **vfs:**。然后应该按平台文件命名约定指定路径。请参见“Palm OS”一节 《UltraLite - 数据库管理和参考》。

  如果 dest_path 字段为空，则 MLFileTransfer 假定它正在下载 Palm 记录数据库 (*.pdb*)。

dest_filename 所下载文件的本地名称。如果此参数为空，则使用文件名中的值。

stream 必需。协议可以是以下各项之一：TCPIP、TLS、HTTP 或 HTTPS。请参见“Stream Type 同步参数”一节 《UltraLite - 数据库管理和参考》。

stream_parms 给定流的协议选项。请参见“UltraLite 同步流的网络协议选项”一节 《UltraLite - 数据库管理和参考》。

username 必需。MobiLink 用户名。

password MobiLink 用户名的口令。

version 必需。MobiLink 脚本版本。

observer 可以提供一个回调以通过 'observer' 字段观察文件下载进度。有关详细信息，请参见后面的回调函数说明。

user_data 使应用程序特定的信息可用于同步观察器。请参见“User Data 同步参数”一节 《UltraLite - 数据库管理和参考》。

force_download 若设置为 true，则即使时间戳指示文件已经存在也下载文件。若设置为 false，则仅当服务器版本和本地版本不同时下载文件。在这种情况下，文件的服务器版本覆盖客户端版本。在下载文件之前，放弃客户端上名称相同的所有以前文件。MLFileTransfer 通过计算每个文件的加密散列值比较文件的服务器和客户端版本；只有文件内容相同时散列值才相同。

enable_resume 如果设置为 true，MLFileTransfer 恢复以前由于通信错误或用户将其取消而中断的下载。如果服务器上的文件比部分本地文件新，则放弃部分文件并从头下载新版本。force_download 参数替换此参数。

num_auth_parms 传送到 MobiLink 事件中的验证参数的验证参数的数目。请参见“Number of Authentication Parameters 参数”一节 《UltraLite - 数据库管理和参考》。

auth_parms 向 MobiLink 事件中的验证参数提供参数。请参见“Authentication Parameters 同步参数”一节 《UltraLite - 数据库管理和参考》。

downloaded_file 设置为以下各项之一：

- 1，如果成功下载文件。
- 0，如果出现错误。如果在调用 MLFileTransfer 时文件已为最新，则出现错误。在这种情况下，函数返回 true 而不是 false。对于 Palm OS，下载记录数据库 (*.pdb*) 文件时，MLFileTransfer 将始终下载此文件，而无论此文件是否为最新。

auth_status 报告 MobiLink 用户验证的状态。MobiLink 服务器将此信息提供给客户端。请参见“Authentication Status 同步参数”一节 《UltraLite - 数据库管理和参考》。

auth_value 报告自定义 MobiLink 用户验证脚本的结果。MobiLink 服务器将此信息提供给客户端。请参见“Authentication Value 同步参数”一节 《UltraLite - 数据库管理和参考》。

file_auth_code 包含服务器可选 authenticate_file_transfer 脚本的返回代码。

error 包含有关出现的任何错误的信息。

返回值

- **ul_true** 文件下载成功。
- **ul_false** 文件下载不成功。可在 ml_file_transfer_info 结构的错误字段中提供错误信息。可恢复未完成的文件传输。

注释

您必须设置要传输的文件的源位置。此位置必须指定为 MobiLink 服务器上的 MobiLink 用户目录（或在此服务器上的缺省目录中）。您也可以设置此文件的预定目标位置和文件名。

例如，您可以通过编程让您的应用程序从 MobiLink 服务器下载新数据库或替换的数据库。您可以为特定用户自定义文件，因为搜索的第一个位置是特定用户的子目录。您也可以维护服务器上根文件夹中的缺省文件，因为如果在用户文件夹中找不到指定文件，则使用此位置。

回调函数

通过观察器参数观察文件传输进度的回调具有以下原型：

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


传送到回调函数的 ml_file_transfer_status 对象定义如下：

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

file_size　所下载文件的总大小（以字节为单位）。

bytes_received　指示迄今为止已下载多少文件，如果是恢复下载，还包括以前的同步。

resumed_at_size　与下载恢复一起使用，指示当前下载的恢复点。

info　指向传送到 MLFileTransfer 的 info 对象。通过此指针可以访问 user_data 参数。

flags　提供额外信息。如果 MLFileTransfer 阻塞网络调用并且下载状态自上次调用观察器函数以来没有更改，则设置 MLFT_STATUS_FLAG_IS_BLOCKING 值。

stop　可设置为 true，以取消当前下载。您可以在对 MLFileTransfer 的后续调用中恢复此次下载，但只有在已设置 enable_resume 参数的情况下才可能。

ULCreateDatabase 函数

创建 UltraLite 数据库。

语法

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

参数

sqlca　指向已初始化的 SQLCA 的指针。

connection-parms　以分号分隔、以 "关键字值" 对形式设置的连接参数的字符串。连接字符串必须包含数据库名称。这些参数与可以在连接到数据库时指定的一组参数相同。有关完整列表，请参见“UltraLite 连接参数”《UltraLite - 数据库管理和参考》。

归类
数据库的所需归类序列。可通过调用相应函数获取归类序列。例如：

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

通过在所需归类的名称前加上 **ULGetCollation_** 前缀构成函数名。有关所有可用归类函数的列表，请参见 *install-dir\SDK\Include\ulgetcoll.h*。在调用任何一个 **ULGetCollation_** 函数的程序中必须包括此文件。

creation-parms
以分号分隔、设置为 "关键字值" 对形式的数据库创建参数的字符串。例如：

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

有关完整列表，请参见“为 UltraLite 选择数据库创建参数”一节《UltraLite - 数据库管理和参考》。

保留　此参数保留供将来使用。

返回值

- **ul_true**　表示成功创建了数据库。
- **ul_false**　详细的错误消息由 SQLCA 中的 SQLCODE 字段定义。通常情况下，此错误是由文件名无效或访问被拒绝造成的。

注释

创建数据库时通过以下两组参数提供信息：

- connection-parms 是标准连接参数，只要访问数据库，就需要这些参数（例如，文件名、用户 id、口令、可选的加密密钥等等）。

- creation-parms 是只有在创建数据库时才相关的参数（例如，模糊处理、页面大小、时间和日期格式等等）

初始化 SQLCA 后应用程序就可以调用此函数。

示例

以下代码说明使用 ULCreateDatabase 创建一个文件 *C:\myfile.udb* 形式的 UltraLite 数据库。

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

ULEnableEccSyncEncryption 函数

为 SSL 或 TLS 流启用 ECC 加密。流参数设置为 TLS 或 HTTPS 时，这是必需的。在这种情况下，还必须将同步参数 tls_type 设置为 ECC。

语法

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

参数

sqlca 指向已初始化的 SQLCA 的指针。

在 C++ API 中使用 Sqlca.GetCA 方法。

注释

此函数可用在 C++ API 应用程序和嵌入式 SQL 应用程序中。必须在调用 Synchronize 函数之前调用此函数。如果不通过先行调用启用同步类型就尝试同步，则会出现错误 SQLE_METHOD_CANNOT_BE_CALLED。

需要单独授予许可的组成部分
ECC 加密和 FIPS 认证的加密需要单独的许可。所有高度加密技术受出口法规约束。

请参见“单独授权的组件”一节 《SQL Anywhere 11 - 简介》。

另请参见

- “ULEnableZlibSyncCompression 函数”一节第 111 页
- “ULEnableRsaFipsSyncEncryption 函数”一节第 106 页

ULEnableFIPSStrongEncryption 函数

为数据库启用基于 FIPS 的高度加密。调用此函数可将适当的加密例程包括在应用程序中并会增加应用程序代码的大小。

语法

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

参数

sqlca 指向已初始化的 SQLCA 的指针。

在 C++ API 中使用 Sqlca.GetCA 方法。

注释

此函数可用在 C++ API 应用程序和嵌入式 SQL 应用程序中。必须在调用 Synchronize 函数之前调用此函数。如果不通过先行调用启用同步类型就尝试同步，则会出现错误 SQLE_METHOD_CANNOT_BE_CALLED。

需要单独授予许可的组成部分
ECC 加密和 FIPS 认证的加密需要单独的许可。所有高度加密技术受出口法规约束。

请参见“单独授权的组件”一节 《SQL Anywhere 11 - 简介》。

另请参见

- “ULEnableStrongEncryption 函数”一节第 108 页

ULEnableHttpSynchronization 函数

启用 HTTP 同步。

语法

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

参数

sqlca 指向已初始化的 SQLCA 的指针。

在 C++ API 中使用 Sqlca.GetCA 方法。

注释

此函数可用在 C++ API 应用程序和嵌入式 SQL 应用程序中。必须在调用 Synchronize 函数之前调用此函数。如果不通过先行调用启用同步类型就尝试同步，则会出现错误 SQLE_METHOD_CANNOT_BE_CALLED。

ULEnableHttpsSynchronization 函数

为 HTTP 启用 SSL 同步流。

语法

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

参数

sqlca 指向已初始化的 SQLCA 的指针。

在 C++ API 中使用 Sqlca.GetCA 方法。

注释

此函数可用在 C++ API 应用程序和嵌入式 SQL 应用程序中。必须在调用 Synchronize 函数之前调用此函数。如果不通过先行调用启用同步类型就尝试同步，则会出现错误 SQLE_METHOD_CANNOT_BE_CALLED。

示例

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

ULEnableRsaFipsSyncEncryption 函数

为 SSL 或 TLS 流启用 RSA FIPS 加密。流参数设置为 TLS 或 HTTPS 时，这是必需的。在这种情况下，还必须将同步参数 tls_type 设置为 RSA。

语法

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

参数

sqlca 指向已初始化的 SQLCA 的指针。

在 C++ API 中使用 Sqlca.GetCA 方法。

注释

此函数可用在 C++ API 应用程序和嵌入式 SQL 应用程序中。必须在调用 Synchronize 函数之前调用此函数。如果不通过先行调用启用同步类型就尝试同步，则会出现错误 SQLE_METHOD_CANNOT_BE_CALLED。

另请参见

- “ULEnableRsaSyncEncryption 函数”一节第 107 页
- “ULEnableEccSyncEncryption 函数”一节第 102 页

ULEnableRsaSyncEncryption 函数

为 SSL 或 TLS 流启用 RSA 加密。流参数设置为 TLS 或 HTTPS 时，这是必需的。在这种情况下，还必须将同步参数 tls_type 设置为 RSA。

语法

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

参数

sqlca 指向已初始化的 SQLCA 的指针。

在 C++ API 中使用 Sqlca.GetCA 方法。

注释

此函数可用在 C++ API 应用程序和嵌入式 SQL 应用程序中。必须在调用 Synchronize 函数之前调用此函数。如果不通过先行调用启用同步类型就尝试同步，则会出现错误 SQLE_METHOD_CANNOT_BE_CALLED。

另请参见

- “ULEnableEccSyncEncryption 函数”一节第 102 页
- “ULEnableRsaFipsSyncEncryption 函数”一节第 106 页

ULEnableStrongEncryption 函数

启用高度加密。

语法

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

参数

sqlca 指向已初始化的 SQLCA 的指针。

在 C++ API 中使用 Sqlca.GetCA 方法。

注释

此函数可用在 C++ API 应用程序和嵌入式 SQL 应用程序中。必须在 db_init 或 ULInitDatabaseManager 之前调用此函数。

> **注意**
> 调用此函数可将加密例程包括在应用程序中并会增加应用程序代码的大小。

ULEnableTcpipSynchronization 函数

启用 TCP/IP 同步。

语法

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

参数

sqlca 指向已初始化的 SQLCA 的指针。

在 C++ API 中使用 Sqlca.GetCA 方法。

注释

此函数可用在 C++ API 应用程序和嵌入式 SQL 应用程序中。必须在调用 Synchronize 函数之前调用此函数。如果不通过先行调用启用同步类型就尝试同步，则会出现错误 SQLE_METHOD_CANNOT_BE_CALLED。

ULEnableTlsSynchronization 函数

启用 TLS 同步。

语法

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

参数

sqlca 指向已初始化的 SQLCA 的指针。

在 C++ API 中使用 Sqlca.GetCA 方法。

注释

此函数可用在 C++ API 应用程序和嵌入式 SQL 应用程序中。必须在调用 Synchronize 函数之前调用此函数。如果不通过先行调用启用同步类型就尝试同步，则会出现错误 SQLE_METHOD_CANNOT_BE_CALLED。

ULEnableZlibSyncCompression 函数

为同步流启用 ZLIB 压缩。

语法

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

参数

sqlca 指向已初始化的 SQLCA 的指针。

在 C++ API 中使用 Sqlca.GetCA 方法。

注释

此函数可用在 C++ API 应用程序和嵌入式 SQL 应用程序中。必须在调用 Synchronize 函数之前调用此函数。如果不通过先行调用启用同步类型就尝试同步，则会出现错误 SQLE_METHOD_CANNOT_BE_CALLED。

ULInitDatabaseManager

初始化 UltraLite 数据库管理器。

语法

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

参数

sqlca 指向已初始化的 SQLCA 的指针。

在 C++ API 中使用 Sqlca.GetCA 方法。

返回值

- 指向数据库管理器的指针。
- 如果函数失败则为 NULL。

注释

如果以前没有初始化数据库管理器而且没有发出 Shutdown，则此函数失败。

ULInitDatabaseManagerNoSQL

初始化 UltraLite 数据库管理器并排除对 SQL 语句处理的支持（这可以大大减少应用程序运行时的大小）。

语法

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

参数

sqlca　指向已初始化的 SQLCA 的指针。

在 C++ API 中使用 Sqlca.GetCA 方法。

返回值

- 指向数据库管理器的指针。
- 如果函数失败则为 NULL。

注释

如果以前没有初始化数据库管理器而且没有发出 Shutdown，则此函数失败。

应用程序必须通过 Table API 访问数据，不能使用 SQL 语句。如果数据库模式包含发布谓语，则不能使用此调用；请使用 ULInitDatabaseManager 代替。

ULRegisterErrorCallback 函数

注册处理错误的回调函数。

语法

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

参数

- **sqlca** 指向 SQL 通信区的指针。

  在 C++ API 中使用 Sqlca.GetCA 方法。

- **callback** 回调函数的名称。有关此函数的原型的详细信息，请参见“ULRegisterErrorCallback 的回调函数”一节第 94 页。

  回调值 UL_NULL 禁用任何以前注册的回调函数。

- **user_data** 全局变量的一个替代，使任何上下文信息在全局范围内均可访问。这是必需的，因为您可以从应用程序中的任何位置调用回调函数。UltraLite 不修改提供的数据，它只是在回调函数被调用时将这些数据传递给回调函数。

  您可以声明任何数据类型，然后在回调函数中将其转换为正确的类型。例如，可以在回调函数中添加一行以下形式的代码：

  ```
  MyContextType * context = (MyContextType *)user_data;
  ```

- **buffer** 一个保存错误消息的替代参数的字符数组，包括空终止符。为了使 UltraLite 尽可能小，UltraLite 不提供错误消息。替代参数取决于特定的错误。有关完整列表，请参见“SQL Anywhere 错误消息”《错误消息》。

  只要 UltraLite 是活动的，buffer 就必须存在。如果您不想接收参数信息，则提供 UL_NULL。

- **len** buffer（前述参数）的长度，以 ul_char 字符为单位。值为 100 时足以容纳大多数错误参数。如果 buffer 太小，则截断参数。

注释

调用此函数后，只要 UltraLite 发出错误信号，就会调用用户提供的回调函数。因此，应在初始化 SQLCA 之后立即调用 ULRegisterErrorCallback。

使用此回调技术处理错误在开发过程中尤其有用，因为它能够确保就出现的任何错误和所有错误向您的应用程序发出通知。但是，此回调函数不控制执行流，所以在对 UltraLite 函数的所有调用完成之后，应用程序应该检查 SQLCA 中的 SQLCODE 字段。

示例

下面的代码为 UltraLite C++ 组件应用程序注册一个回调函数：


```
int main() {
    ul_char buffer[100];
    DatabaseManager * dm;
    Connection * conn;
    Sqlca.Initialize();
    ULRegisterErrorCallback(
        Sqlca.GetCA(),
        MyErrorCallBack,
        UL_NULL,
        buffer,
        sizeof (buffer) );
    dm = ULInitDatabaseManager( Sqlca );
    ...
}
```

以下是一个示例回调函数：

```
ul_error_action UL_GENNED_FN_MOD MyErrorCallBack(
    SQLCA *     Sqlca,
    ul_void *   user_data,
    ul_char *   message_param )
{
    ul_error_action rc = 0;
    (void) user_data;

    switch( Sqlca->sqlcode ) {
        // The following error is used for flow control - don't report it
here
        case SQLE_NOTFOUND:
            break;
        case SQLE_ULTRALITE_DATABASE_NOT_FOUND:
          _tprintf( _TEXT( "Error %ld: Database file %s not found\n" ), Sqlca-
>sqlcode, message_param );
            break;
        default:
            _tprintf( _TEXT( "Error %ld: %s\n" ), Sqlca->sqlcode,
message_param );
        break;
     }
    return rc;
}
```

另请参见

- “按 Sybase 错误代码排序的 SQL Anywhere 错误消息”一节 《错误消息》
- “ULRegisterErrorCallback 的回调函数”一节第 94 页

ULRegisterSQLPassthroughCallback

注册提供进行中状态的回调函数。

语法

```
void ULRegisterSQLPassthroughCallback (
   SQLCA * sqlca,
   ul_sql_passthrough_observer_fncallback,
   ul_void *   user_data
);
```

参数

- **sqlca** 指向 SQL 通信区的指针。

  在 C++ API 中使用 Sqlca.GetCA 方法。

- **callback** 回调函数的名称。

  回调值 UL_NULL 禁用任何以前注册的回调函数。

- **user_data** 全局变量的一个替代，使任何上下文信息在全局范围内均可访问。这是必需的，因为您可以从应用程序中的任何位置调用回调函数。UltraLite 不修改提供的数据，它只是在回调函数被调用时将这些数据传递给回调函数。

  您可以声明任何数据类型，然后在回调函数中将其转换为正确的类型。例如，可以在回调函数中添加一行以下形式的代码：

  ```
  MyContextType * context = (MyContextType *) user_data;
  ```

示例

以下为示例回调函数以及调用 ULRegisterSQLPassthroughCallback 的代码：

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

int main() {
    ULSqlca sqlca;

    sqlca.Initialize();
    ULRegisterSQLPassthroughCallback( sqlca.GetCA(), passthroughCallback,
NULL );
```



```
    DatabaseManager * dm = ULInitDatabaseManager( sqlca );
    ...
}
```

ULRegisterSynchronizationCallback

通过 SQL SYNCHRONIZE 语句执行同步时，注册一个要调用的函数。如果使用 UltraLite 定义和注册同步回调函数，则无论何时执行 SYNCHRONIZE 语句，都会将该同步的进度信息传递给回调函数。如果未注册任何回调，则将取消进度信息。

语法

```
void ULRegisterSynchronizationCallback (
   SQLCA * sqlca,
    ul_synch_observer_fncallback,
   ul_void *   user_data
);
```

参数

- **sqlca** 指向 SQL 通信区的指针。

  在 C++ API 中使用 Sqlca.GetCA 方法。

- **callback** 回调函数的名称。

  回调值 UL_NULL 禁用任何以前注册的回调函数。

- **user_data** 全局变量的一个替代，使任何上下文信息在全局范围内均可访问。这是必需的，因为您可以从应用程序中的任何位置调用回调函数。UltraLite 不修改提供的数据，它只是在回调函数被调用时将这些数据传递给回调函数。

  您可以声明任何数据类型，然后在回调函数中将其转换为正确的类型。例如，可以在回调函数中添加一行以下形式的代码：

  ```
  MyContextType * context = (MyContextType *) user_data;
  ```

另请参见

- “UltraLite SYNCHRONIZE 语句”一节 《UltraLite - 数据库管理和参考》

用于 UltraLite C/C++ 应用程序的宏和编译器指令

除非另外说明，否则指令对嵌入式 SQL 应用程序和 C++ API 应用程序均适用。

可以提供编译器指令：

- 在编译器命令行上。通常使用 /D 选项设置指令。例如，要编译具有用户验证的 UltraLite 应用程序，Microsoft Visual C++ 编译器的生成文件可能如下所示：

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

IncludeFolders= \
/I"$(VCDIR)\include" \
/I"$(SQLANY11)\SDK\Include"

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

  VCDIR 是 Visual C++ 目录，*SQLANY11* 是 SQL Anywhere 安装目录。

- 在用户界面的编译器设置窗口中。
- 在源代码中。使用 `#define` 语句提供指令。

UL_AS_SYNCHRONIZE 宏

提供用于指示 ActiveSync 同步的回调消息的名称。

注释

仅适用于使用 ActiveSync 的 Windows Mobile 应用程序。

另请参见

- “向应用程序添加 ActiveSync 同步”一节第 86 页

UL_SYNC_ALL 宏

提供一个发布列表字符串，用于引用数据库中的所有表（包括那些未被发布显示引用的表）。此列表中不包括标记为 "no sync" 的表。

另请参见

- “ul_synch_info_a 结构”一节第 127 页
- “ul_synch_info_w2 结构”一节第 129 页
- “ULGetLastDownloadTime 函数”一节第 266 页
- “ULCountUploadRows 函数”一节第 258 页
- “UL_SYNC_ALL_PUBS 宏”一节第 120 页

UL_SYNC_ALL_PUBS 宏

提供一个发布列表，用于引用在发布中引用的所有数据库表。

另请参见

- “ul_synch_info_a 结构”一节第 127 页
- “ul_synch_info_w2 结构”一节第 129 页
- “ULGetLastDownloadTime 函数”一节第 266 页
- “ULCountUploadRows 函数”一节第 258 页
- “UL_SYNC_ALL 宏”一节第 119 页

UL_TEXT 宏

准备要编译为单字节字符串或宽字符字符串的常量字符串。如果打算编译应用程序以使用字符串的 Unicode 和非 Unicode 表示，则使用此宏将所有常量字符串括起来。此宏正确定义所有环境和平台中的字符串。

UL_USE_DLL 宏

将应用程序设置为使用运行时库 DLL，而不是使用静态运行时库。

注释

适用于 Windows Mobile 和 Windows 应用程序。

UNDER_CE 宏

缺省情况下，由 Microsoft eMbedded Visual C++ 编译器在所有新的 eMbedded Visual C++ 项目中定义此宏。

注释

适用于 Windows Mobile 应用程序。

示例

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

另请参见

- “开发用于 Windows Mobile 的 UltraLite 应用程序”第 77 页

UNDER_PALM_OS 宏

此宏由 UltraLite 插件在 UltraLite Palm OS 应用程序所包括的 *ulpalmos.h* 头文件中定义。请参见“使用用于 CodeWarrior 的 UltraLite 插件”一节第 65 页。

注释

仅适用于 Palm OS 的编译器指令。

另请参见

- “开发用于 Palm OS 的 UltraLite 应用程序”第 61 页

UltraLite C++ API 参考

目录

ul_sql_passthrough_status 结构 125
ul_stream_error 结构 126
ul_synch_info_a 结构 127
ul_synch_info_w2 结构 129
ul_synch_result 结构 131
ul_synch_stats 结构 132
ul_synch_status 结构 133
ul_validate_data 结构 135
ULSqlca 类 136
ULSqlcaBase 类 138
ULSqlcaWrap 类 143
UltraLite_Connection 类 145
UltraLite_Connection_iface 类 148
UltraLite_Cursor_iface 类 172
UltraLite_DatabaseManager 类 179
UltraLite_DatabaseManager_iface 类 180
UltraLite_DatabaseSchema 类 183
UltraLite_DatabaseSchema_iface 类 184
UltraLite_IndexSchema 类 187
UltraLite_IndexSchema_iface 类 188
UltraLite_PreparedStatement 类 193
UltraLite_PreparedStatement_iface 类 194
UltraLite_ResultSet 类 198
UltraLite_ResultSet_iface 类 199
UltraLite_ResultSetSchema 类 200
UltraLite_RowSchema_iface 类 201
UltraLite_SQLObject_iface 类 206
UltraLite_StreamReader 类 208
UltraLite_StreamReader_iface 类 209
UltraLite_StreamWriter 类 212
UltraLite_Table 类 213

UltraLite_Table_iface 类 215
UltraLite_TableSchema 类 221
UltraLite_TableSchema_iface 类 223
ULValue 类 232

ul_sql_passthrough_status 结构

SQL 直通状态信息。

语法

public **ul_sql_passthrough_status**

属性

名称	类型	说明
cur_script	ul_u_long	正在执行的当前脚本（从 1 开始）。
script_count	ul_u_long	将要执行的脚本的总数。
sqlca	SQLCA *	指向 SQL 通信区的指针。请参见“GetCA 函数”一节 第 139 页。
state	ul_sql_passthrough_state	当前状态。请参见“处理同步状态信息”一节第 54 页。
stop	ul_bool	设置为 true 可停止脚本执行。
user_data	ul_void *	注册调用中提供的用户数据。请参见“ULRegisterSQLPassthroughCallback”一节第 116 页。

ul_stream_error 结构

同步通信流错误信息。

语法

public **ul_stream_error**

属性

名称	类型	说明
error_string	char	stream_error_code 字段数组。
stream_error_code	ss_error_code	非必需字段。值始终为 0。
system_error_code	asa_int32	系统特定的错误代码。有关错误代码的详细信息，请参见平台文档。对于 Windows 平台，请参见 Microsoft Developer Network Web 站点。

注释

以下是 Windows 上的常见系统错误：

- 10048 (WSAADDRINUSE) 地址已经在使用。
- 10053 (WSAECONNABORTED) 软件导致连接中止。
- 10054 (WSAECONNRESET) 通信的另一端关闭了套接字。
- 10060 (WSAETIMEDOUT) 连接超时。
- 10061 (WSAECONNREFUSED) 连接被拒绝。通常情况下，这表示 MobiLink 服务器没有运行或者没有在指定的端口上监听。请参见 Microsoft Developer Network Web 站点。

ul_synch_info_a 结构

用于描述同步数据的结构。

语法

public **ul_synch_info_a**

属性

名称	类型	说明
additional_parms	const char *	一串附加同步参数，编码为分号分隔的 "关键字=值" 对的列表。此字段通常包含不经常使用的同步参数。请参见“Additional Parameters 同步参数”一节 《UltraLite - 数据库管理和参考》。
auth_parms	char **	MobiLink 事件中的验证参数的数组。
auth_status	ul_auth_status	MobiLink 用户验证的状态。MobiLink 服务器将此信息提供给向客户端。
auth_value	ul_s_long	自定义 MobiLink 用户验证脚本的结果。MobiLink 服务器将此信息提供给客户端以确定验证状态。
download_only	ul_bool	在当前同步期间不从 UltraLite 数据库上载任何更改。
ignored_rows	ul_bool	忽略的行的状态。如果 MobiLink 服务器在同步期间由于缺少脚本而忽略了任何行，则此只读字段将报告 true。
init_verify	ul_synch_info_a *	初始化验证。
keep_partial_download	ul_bool	在同步期间当下载由于通信错误而失败时，此参数控制 UltraLite 是否保存部分下载而不回退更改。
new_password	char *	用来指定与用户名相关联的新的 MobiLink 口令的字符串。此参数是可选的。
num_auth_parms	ul_byte	传送到 MobiLink 事件中的验证参数的验证参数的数目。
observer	ul_synch_observer_fn	指向用于监控同步的回调函数或事件处理程序的指针。此参数是可选的。
partial_download_retained	ul_bool	在同步期间当下载由于通信错误而失败时，此参数将表示 UltraLite 是否应用这些已下载的更改而不回退这些更改。
password	char *	用来指定与用户名相关联的现有 MobiLink 口令的字符串。此参数是可选的。

名称	类型	说明
ping	ul_bool	确认 UltraLite 客户端和 MobiLink 服务器之间的通信。当此参数设置为 true 时，不进行同步。
publications	const char *	以逗号分隔的发布列表，表示同步中包含什么数据。
resume_partial_download	ul_bool	重新开始失败的下载。同步不上载更改；它仅下载那些要在失败的下载中下载的更改。
send_column_names	ul_bool	指示应在上载中被发送到 MobiLink 服务器的列名的应用程序。
send_download_ack	ul_bool	指示 MobiLink 服务器此客户端是否将提供下载确认。
stream	const char *	用于同步的 MobiLink 网络协议。
stream_error	ul_stream_error	用于保存通信错误报告信息的结构。
stream_parms	char *	用于配置所选网络协议的选项。
upload_ok	ul_bool	上载到 MobiLink 服务器的数据的状态。如果上载成功，此字段将报告 true。
upload_only	ul_bool	在当前同步期间不从统一数据库下载任何更改。这样可以节省通信时间，在通信链接很慢时尤其如此。
user_data	ul_void *	使应用程序特定的信息可用于同步观察器。此参数是可选的。
user_name	char *	MobiLink 服务器用于标识唯一 MobiLink 用户的字符串。
version	char *	版本字符串使 UltraLite 应用程序可以从一组同步脚本中进行选择。

注释

同步参数控制 UltraLite 数据库和 MobiLink 服务器之间的同步行为。Stream Type 同步参数、User Name 同步参数和 Version 同步参数是必需的。如果未设置这些参数，同步函数将返回错误（SQLE_SYNC_INFO_INVALID 或与其等效的信息）。您一次只能指定仅下载、Ping 或仅上载中的一个参数。如果将这些参数中的多个参数设置为 true，则同步函数将返回错误（SQLE_SYNC_INFO_INVALID 或与其等效的信息）。

ul_synch_info_w2 结构

用于描述同步的宽字符结构。

语法

public **ul_synch_info_w2**

属性

名称	类型	说明
additional_parms	const ul_wchar *	一串附加同步参数，编码为分号分隔的 "关键字=值" 对的列表。此字段通常包含不经常使用的同步参数。请参见“Additional Parameters 同步参数”一节 《UltraLite - 数据库管理和参考》。
auth_parms	ul_wchar **	MobiLink 事件中的验证参数的数组。
auth_status	ul_auth_status	MobiLink 用户验证的状态。MobiLink 服务器将此信息提供给向客户端。
auth_value	ul_s_long	MobiLink 服务器将此信息提供给客户端以确定验证状态。
download_only	ul_bool	在当前同步期间不从 UltraLite 数据库上载任何更改。
ignored_rows	ul_bool	忽略的行的状态。如果 MobiLink 服务器在同步期间由于缺少脚本而忽略了任何行，则此只读字段将报告 true。
init_verify	ul_synch_info_w2 *	初始化验证。
keep_partial_download	ul_bool	在同步期间当下载由于通信错误而失败时，此参数控制 UltraLite 是否保存部分下载而不回退更改。
new_password	ul_wchar *	用来指定与用户名相关联的新的 MobiLink 口令的字符串。此参数是可选的。
num_auth_parms	ul_byte	传送到 MobiLink 事件中的验证参数的验证参数的数目。
observer	ul_synch_observer_fn	指向用于监控同步的回调函数或事件处理程序的指针。此参数是可选的。
partial_download_retained	ul_bool	在同步期间当下载由于通信错误而失败时，此参数将表示 UltraLite 是否应用这些已下载的更改而不回退这些更改。

名称	类型	说明
password	ul_wchar *	用来指定与用户名相关联的现有 MobiLink 口令的字符串。此参数是可选的。
ping	ul_bool	确认 UltraLite 客户端和 MobiLink 服务器之间的通信。当此参数设置为 true 时，不进行同步。
publications	const ul_wchar *	以逗号分隔的发布列表，表示同步中包含什么数据。
resume_partial_download	ul_bool	重新开始失败的下载。同步不上载更改；它仅下载那些要在失败的下载中下载的更改。
send_column_names	ul_bool	指示应在上载中被发送到 MobiLink 服务器的列名的应用程序。
send_download_ack	ul_bool	指示 MobiLink 服务器此客户端是否将提供下载确认。
stream	const char *	用于同步的 MobiLink 网络协议。
stream_error	ul_stream_error	用于保存通信错误报告信息的结构。
stream_parms	ul_wchar *	用于配置所选网络协议的选项。
upload_ok	ul_bool	上载到 MobiLink 服务器的数据的状态。如果上载成功，此字段将报告 true。
upload_only	ul_bool	在当前同步期间不从统一数据库下载任何更改。这样可以节省通信时间，在通信链接很慢时尤其如此。
user_data	ul_void *	使应用程序特定的信息可用于同步观察器。此参数是可选的。
user_name	ul_wchar *	MobiLink 服务器用于标识唯一 MobiLink 用户的字符串。
version	ul_wchar *	版本字符串使 UltraLite 应用程序可以从一组同步脚本中进行选择。

注释

同步参数控制 UltraLite 数据库和 MobiLink 服务器之间的同步行为。Stream Type 同步参数、User Name 同步参数和 Version 同步参数是必需的。如果未设置这些参数，同步函数将返回错误（SQLE_SYNC_INFO_INVALID 或与其等效的信息）。您一次只能指定仅下载、Ping 或仅上载中的一个参数。如果将这些参数中的多个参数设置为 true，则同步函数将返回错误（SQLE_SYNC_INFO_INVALID 或与其等效的信息）。请参见“ul_synch_info_a 结构”一节 第 127 页。

ul_synch_result 结构

用于保存同步结果，以便在应用程序中可以采取相应操作的结构。

语法

public **ul_synch_result**

属性

名称	类型	说明
auth_status	ul_auth_status	同步验证状态。
auth_value	ul_s_long	MobiLink 同步服务器用于确定 auth_status 结果的值。
ignored_rows	ul_bool	如果忽略了已上载的行，则设置为 true；否则，设置为 false。
partial_download_retained	ul_bool	表示已保留了部分下载的值。请参见 keep_partial_download。
sql_code	an_sql_code	上一次同步中的 SQL 代码。
sql_error_string	char	与 sql_code 中错误代码相关联的错误文本。
status	ul_synch_status	观察器函数使用的状态信息。请参见观察器。
stream_error	ul_stream_error	通信流错误信息。
timestamp	SQLDATETIME	上一次同步的时间和日期。
upload_ok	ul_bool	如果上载成功，则设置为 true；否则，设置为 false。

ul_synch_stats 结构

报告同步流的统计信息。

语法

public **ul_synch_stats**

属性

名称	类型	说明
bytes	ul_u_long	当前发送的字节数。
deletes	ul_u_long	当前发送的删除行的数量。
inserts	ul_u_long	当前插入的行数。
updates	ul_u_long	当前发送的更新行的数量。

ul_synch_status 结构

返回同步进度监控数据。

语法

public **ul_synch_status**

属性

名称	类型	说明
db_table_count	ul_u_short	返回数据库中的表的数目。
flags	ul_u_short	返回指示有关当前状态的其它信息的当前同步标志。
info	ul_synch_info_a *	指向 ul_synch_info_a 结构的指针。请参见“ul_synch_info_a 结构”一节第 127 页。
received	ul_synch_stats	返回下载统计信息。
sent	ul_synch_stats	返回上载统计信息。
sqlca	SQLCA *	连接的活动 SQLCA。
state	ul_synch_state	许多受支持的状态之一。请参见“处理同步状态信息”一节第 54 页。
stop	ul_bool	取消同步的布尔值。值为 true 表示同步已取消。
sync_table_count	ul_u_short	返回正在同步的表的数目。
sync_table_index	ul_u_short	值域为从 1 到 sync_table_count 中指定的数。
table_id	ul_u_short	正在上载或下载的当前表 ID（从 1 开始）。当没有同步所有表时此数字可能会跳过某些值，不一定会增加。
table_name	char	当前表的名称。
table_name_w2	char	当前表的名称（宽字符格式）。此字段只能在 Windows 桌面操作系统和 Mobile 平台上填充。

名称	类型	说明
user_data	ul_void *	传递给 ULRegisterSynchronizationCallback 或者在 ul_synch_info 结构中设置的用户数据。

ul_validate_data 结构

验证状态信息。

语法

public **ul_validate_data**

属性

名称	类型	说明
I	ul_u_long	整数参数。
parm_count	ul_u_short	结构中参数的数量。
parm_type	enum	表示参数数组中每个参数的类型（整型或字符串型）。
parms	struct ul_validate_data::@12	参数数组。
s	char	字符串参数（注意：这不是宽字符）。
status_id	ul_validate_status_id	说明在校验过程中要报告的内容。
stop	ul_bool	取消验证的布尔值。值为 true 表示校验已取消。
type	parm_type	存储参数的类型。
user_data	ul_void *	传递给校验例程的用户定义的数据指针。

ULSqlca 类

“ULSqlcaBase 类”一节第 138 页包含 SQLCA 结构，因此不需要外部结构。

语法

public **ULSqlca**

基类

- “ULSqlcaBase 类”一节第 138 页

成员

ULSqlca 的所有成员（包括所有继承成员）。

- “Finalize 函数”一节第 138 页
- “GetCA 函数”一节第 139 页
- “GetParameter 函数”一节第 139 页
- “GetParameter 函数”一节第 139 页
- “GetParameterCount 函数”一节第 140 页
- “GetSQLCode 函数”一节第 140 页
- “GetSQLCount 函数”一节第 140 页
- “GetSQLErrorOffset 函数”一节第 141 页
- “Initialize 函数”一节第 141 页
- “LastCodeOK 函数”一节第 141 页
- “LastFetchOK 函数”一节第 142 页
- “ULSqlca 函数”一节第 136 页
- “~ULSqlca 函数”一节第 136 页

注释

大多数 C++ 组件应用程序都使用此类。必须先初始化 SQLCA，然后才能调用任何其它函数。每一线程都需要有自己的 SQLCA。

ULSqlca 函数

此函数为 SQLCA 构造函数。

语法

ULSqlca::ULSqlca()

~ULSqlca 函数

此函数为 SQLCA 析构函数。

语法

ULSqlca::~ULSqlca()

ULSqlcaBase 类

定义接口库和应用程序之间的通信区。

语法

public **ULSqlcaBase**

派生类

- “ULSqlca 类”一节第 136 页
- “ULSqlcaWrap 类”一节第 143 页

成员

ULSqlcaBase 的所有成员（包括所有继承成员）。

- “Finalize 函数”一节第 138 页
- “GetCA 函数”一节第 139 页
- “GetParameter 函数”一节第 139 页
- “GetParameter 函数”一节第 139 页
- “GetParameterCount 函数”一节第 140 页
- “GetSQLCode 函数”一节第 140 页
- “GetSQLCount 函数”一节第 140 页
- “GetSQLErrorOffset 函数”一节第 141 页
- “Initialize 函数”一节第 141 页
- “LastCodeOK 函数”一节第 141 页
- “LastFetchOK 函数”一节第 142 页

注释

使用此类的子类（通常为“ULSqlca 类”一节第 136 页）创建通信区。此 API 总是需要一个基础 SQLCA 对象。必须先初始化 SQLCA，然后才能调用任何其它函数。每一线程都需要有自己的 SQLCA。

Finalize 函数

结束此 SQLCA。

语法

void **ULSqlcaBase::Finalize()**

注释

除非再次初始化此通信区，否则您无法使用它。

GetCA 函数

获取 SQLCA 结构，以便直接访问其它字段。

语法

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

返回值

原始 SQLCA 结构。

GetParameter 函数

获取错误参数字符串。

语法

size_t **ULSqlcaBase::GetParameter(**
  ul_u_long *parm_num*,
  char * *buffer*,
  size_t *size*
)

参数

- **parm_num**　从 1 开始的参数编号。
- **buffer**　接收参数字符串的缓冲区。
- **size**　缓冲区的大小（以字符为单位）。

返回值

- 如果此函数成功，则返回值为保存整个参数字符串所需的缓冲区大小。
- 如果此函数失败，则返回值为 0。如果给出的参数编号无效（超出范围），则此函数将失败。

注释

即使缓冲区太小而导致参数被截断，输出参数字符串也总是以空值终止。参数编号从 1 开始。

GetParameter 函数

获取错误参数字符串。

语法

size_t **ULSqlcaBase::GetParameter(**
  ul_u_long *parm_num*,
  ul_wchar * *buffer*,
  size_t *size*
)

参数

- **parm_num** 从 1 开始的参数编号。
- **buffer** 接收参数字符串的缓冲区。
- **size** 缓冲区大小（以 ul_wchar 为单位）。

返回值

- 如果此函数成功，则返回值为保存整个参数字符串所需的缓冲区大小。
- 如果此函数失败，则返回值为 0。如果给出的参数编号无效（超出范围），则此函数将失败。

注释

即使缓冲区太小而导致参数被截断，输出参数字符串也总是以空值终止。参数编号从 1 开始。

GetParameterCount 函数

获取上一操作的错误参数计数。

语法

ul_u_long **ULSqlcaBase::GetParameterCount()**

返回值

当前错误的参数数目。

GetSQLCode 函数

获取上一操作的错误代码 (SQLCODE)。

语法

an_sql_code **ULSqlcaBase::GetSQLCode()**

返回值

sqlcode 值。

GetSQLCount 函数

获取上一操作的 sql 计数变量 (SQLCOUNT)。

语法

an_sql_code **ULSqlcaBase::GetSQLCount()**

返回值

受 INSERT、DELETE 或 UPDATE 操作影响的行数。如果未影响任何行，则返回 0。

GetSQLErrorOffset 函数

获取动态 SQL 语句中的错误偏移。

语法

ul_s_long **ULSqlcaBase::GetSQLErrorOffset()**

返回值

- 如果适用，则返回值是与当前错误相对应的相关动态 SQL 语句（被传递给 PrepareStatement 函数）中的偏移。
- 如果不适用，则返回值为 -1。

Initialize 函数

初始化此 SQLCA。

语法

bool **ULSqlcaBase::Initialize()**

返回值

- 如果 SQLCA 已初始化，则返回 True。
- 如果 SQLCA 初始化失败，则返回 False。如果基本接口库初始化失败，则此方法可能会失败。当系统资源耗尽时可能会发生库失败。

注释

必须先初始化此 SQLCA，然后才能执行任何其它操作。

LastCodeOK 函数

测试上一操作的错误代码。

语法

bool **ULSqlcaBase::LastCodeOK()**

返回值

- 如果 sqlcode 为 SQLE_NOERROR 或警告，则返回 TRUE。
- 如果 sqlcode 指示错误，则返回 FALSE。

LastFetchOK 函数

测试上一读取操作的错误代码。

语法

bool **ULSqlcaBase::LastFetchOK()**

返回值

- 如果 sqlcode 指示某一行被上一操作成功读取，则返回 TRUE。
- 如果 sqlcode 指示该行未被读取，则返回 FALSE。

注释

仅在执行读取操作后立即使用此函数。

ULSqlcaWrap 类

附加到现有 SQLCA 对象上的“ULSqlcaBase 类”一节第 138 页。

语法

public **ULSqlcaWrap**

基类

- “ULSqlcaBase 类”一节第 138 页

成员

ULSqlcaWrap 的所有成员（包括所有继承成员）。

- “Finalize 函数”一节第 138 页
- “GetCA 函数”一节第 139 页
- “GetParameter 函数”一节第 139 页
- “GetParameter 函数”一节第 139 页
- “GetParameterCount 函数”一节第 140 页
- “GetSQLCode 函数”一节第 140 页
- “GetSQLCount 函数”一节第 140 页
- “GetSQLErrorOffset 函数”一节第 141 页
- “Initialize 函数”一节第 141 页
- “LastCodeOK 函数”一节第 141 页
- “LastFetchOK 函数”一节第 142 页
- “ULSqlcaWrap 函数”一节第 143 页
- “~ULSqlcaWrap 函数”一节第 144 页

注释

它可以和先前初始化的 SQLCA 对象一起使用（在这种情况下，不要再次调用 Initialize 函数）。必须先初始化 SQLCA，然后才能调用任何其它函数。每一线程都需要有自己的 SQLCA。

ULSqlcaWrap 函数

构造函数。

语法

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

参数

- **sqlca**　要使用的 SQLCA 对象。

注释

可在创建此对象之前初始化给定的 SQLCA 对象。在这种情况下，不要再次调用“Initialize 函数”一节第 141 页。

~ULSqlcaWrap 函数

析构函数。

语法

ULSqlcaWrap::~ULSqlcaWrap()

UltraLite_Connection 类

表示与 UltraLite 数据库的连接。

语法

public **UltraLite_Connection**

基类

- “UltraLite_SQLObject_iface 类”一节第 206 页
- “UltraLite_Connection_iface 类”一节第 148 页

成员

UltraLite_Connection 的所有成员（包括所有继承成员）。

- “AddRef 函数”一节第 206 页
- “CancelGetNotification 函数”一节第 150 页
- “ChangeEncryptionKey 函数”一节第 150 页
- “Checkpoint 函数”一节第 151 页
- “Commit 函数”一节第 151 页
- “CountUploadRows 函数”一节第 151 页
- “CreateNotificationQueue 函数”一节第 151 页
- “DeclareEvent 函数”一节第 152 页
- “DestroyNotificationQueue 函数”一节第 153 页
- “ExecuteNextSQLPassthroughScript 函数”一节第 153 页
- “ExecuteSQLPassthroughScripts 函数”一节第 154 页
- “GetConnection 函数”一节第 206 页
- “GetConnectionNum 函数”一节第 154 页
- “GetDatabaseID 函数”一节第 154 页
- “GetDatabaseProperty 函数”一节第 154 页
- “GetDatabaseProperty 函数”一节第 155 页
- “GetIFace 函数”一节第 207 页
- “GetLastDownloadTime 函数”一节第 155 页
- “GetLastIdentity 函数”一节第 155 页
- “GetNewUUID 函数”一节第 156 页
- “GetNewUUID 函数”一节第 156 页
- “GetNotification 函数”一节第 156 页
- “GetNotificationParameter 函数”一节第 157 页
- “GetSchema 函数”一节第 158 页
- “GetSqlca 函数”一节第 158 页
- “GetSQLPassthroughScriptCount 函数”一节第 158 页
- “GetSuspend 函数（不建议使用）”一节第 158 页
- “GetSynchResult 函数”一节第 158 页
- “GetUtilityULValue 函数”一节第 159 页
- “GlobalAutoincUsage 函数”一节第 159 页
- “GrantConnectTo 函数”一节第 159 页
- “InitSynchInfo 函数”一节第 160 页
- “InitSynchInfo 函数”一节第 160 页
- “OpenTable 函数”一节第 160 页
- “OpenTableEx 函数”一节第 161 页
- “OpenTableWithIndex 函数”一节第 161 页
- “PrepareStatement 函数”一节第 162 页
- “RegisterForEvent 函数”一节第 162 页
- “Release 函数”一节第 207 页
- “ResetLastDownloadTime 函数”一节第 163 页
- “RevokeConnectFrom 函数”一节第 163 页
- “Rollback 函数”一节第 163 页
- “RollbackPartialDownload 函数”一节第 164 页
- “SendNotification 函数”一节第 164 页

- “SetDatabaseID 函数”一节第 165 页
- “SetDatabaseOption 函数”一节第 165 页
- “SetDatabaseOption 函数”一节第 165 页
- “SetSuspend 函数（不建议使用）”一节第 166 页
- “SetSynchInfo 函数”一节第 166 页
- “SetSynchInfo 函数”一节第 166 页
- “Shutdown 函数”一节第 167 页
- “StartSynchronizationDelete 函数”一节第 167 页
- “StopSynchronizationDelete 函数”一节第 167 页
- “StrToUUID 函数”一节第 168 页
- “StrToUUID 函数”一节第 168 页
- “Synchronize 函数”一节第 168 页
- “Synchronize 函数”一节第 169 页
- “SynchronizeFromProfile 函数”一节第 169 页
- “TriggerEvent 函数”一节第 170 页
- “UUIDToStr 函数”一节第 170 页
- “UUIDToStr 函数”一节第 171 页
- “ValidateDatabase 函数”一节第 171 页

UltraLite_Connection_iface 类

connection 接口。

语法

public **UltraLite_Connection_iface**

派生类

- “UltraLite_Connection 类”一节第 145 页

成员

UltraLite_Connection_iface 的所有成员（包括所有继承成员）。

- “CancelGetNotification 函数”一节第 150 页
- “ChangeEncryptionKey 函数”一节第 150 页
- “Checkpoint 函数”一节第 151 页
- “Commit 函数”一节第 151 页
- “CountUploadRows 函数”一节第 151 页
- “CreateNotificationQueue 函数”一节第 151 页
- “DeclareEvent 函数”一节第 152 页
- “DestroyNotificationQueue 函数”一节第 153 页
- “ExecuteNextSQLPassthroughScript 函数”一节第 153 页
- “ExecuteSQLPassthroughScripts 函数”一节第 154 页
- “GetConnectionNum 函数”一节第 154 页
- “GetDatabaseID 函数”一节第 154 页
- “GetDatabaseProperty 函数”一节第 154 页
- “GetDatabaseProperty 函数”一节第 155 页
- “GetLastDownloadTime 函数”一节第 155 页
- “GetLastIdentity 函数”一节第 155 页
- “GetNewUUID 函数”一节第 156 页
- “GetNewUUID 函数”一节第 156 页
- “GetNotification 函数”一节第 156 页
- “GetNotificationParameter 函数”一节第 157 页
- “GetSchema 函数”一节第 158 页
- “GetSqlca 函数”一节第 158 页
- “GetSQLPassthroughScriptCount 函数”一节第 158 页
- “GetSuspend 函数（不建议使用）”一节第 158 页
- “GetSynchResult 函数”一节第 158 页
- “GetUtilityULValue 函数”一节第 159 页
- “GlobalAutoincUsage 函数”一节第 159 页
- “GrantConnectTo 函数”一节第 159 页
- “InitSynchInfo 函数”一节第 160 页
- “InitSynchInfo 函数”一节第 160 页
- “OpenTable 函数”一节第 160 页
- “OpenTableEx 函数”一节第 161 页
- “OpenTableWithIndex 函数”一节第 161 页
- “PrepareStatement 函数”一节第 162 页
- “RegisterForEvent 函数”一节第 162 页
- “ResetLastDownloadTime 函数”一节第 163 页
- “RevokeConnectFrom 函数”一节第 163 页
- “Rollback 函数”一节第 163 页
- “RollbackPartialDownload 函数”一节第 164 页
- “SendNotification 函数”一节第 164 页
- “SetDatabaseID 函数”一节第 165 页
- “SetDatabaseOption 函数”一节第 165 页
- “SetDatabaseOption 函数”一节第 165 页
- “SetSuspend 函数（不建议使用）”一节第 166 页

- “SetSynchInfo 函数”一节第 166 页
- “SetSynchInfo 函数”一节第 166 页
- “Shutdown 函数”一节第 167 页
- “StartSynchronizationDelete 函数”一节第 167 页
- “StopSynchronizationDelete 函数”一节第 167 页
- “StrToUUID 函数”一节第 168 页
- “StrToUUID 函数”一节第 168 页
- “Synchronize 函数”一节第 168 页
- “Synchronize 函数”一节第 169 页
- “SynchronizeFromProfile 函数”一节第 169 页
- “TriggerEvent 函数”一节第 170 页
- “UUIDToStr 函数”一节第 170 页
- “UUIDToStr 函数”一节第 171 页
- “ValidateDatabase 函数”一节第 171 页

CancelGetNotification 函数

取消与给定名称匹配的所有队列上任何待执行 get-notification 调用。

语法

```
ul_u_long UltraLite_Connection_iface::CancelGetNotification(
  const ULValue & queue_name
)
```

参数

- **queue_name** 要取消的队列的名称。

返回值

受影响的队列数（不一定是受阻塞读取的数目）。

另请参见

- “DeclareEvent 函数”一节第 152 页
- “DestroyNotificationQueue 函数”一节第 153 页
- “GetNotification 函数”一节第 156 页
- “RegisterForEvent 函数”一节第 162 页
- “SendNotification 函数”一节第 164 页
- “TriggerEvent 函数”一节第 170 页

ChangeEncryptionKey 函数

更改加密密钥。

语法

```
bool UltraLite_Connection_iface::ChangeEncryptionKey(
  const ULValue & new_key
)
```

参数

- **new_key** 数据库的新加密密钥值。

Checkpoint 函数

对数据库执行检查点操作。

语法

```
bool UltraLite_Connection_iface::Checkpoint()
```

Commit 函数

提交当前事务。

语法

```
bool UltraLite_Connection_iface::Commit()
```

CountUploadRows 函数

确定需要上载的行数。

语法

```
ul_u_long UltraLite_Connection_iface::CountUploadRows(
  const ULValue & pub_list,
  ul_u_long threshold
)
```

参数

- **pub_list** 以逗号分隔的要考虑的发布的列表。
- **threshold** 对要计数的行数的限制。

CreateNotificationQueue 函数

创建此连接的事件通知队列。

语法

```
bool UltraLite_Connection_iface::CreateNotificationQueue(
  const ULValue & name,
  const ULValue & parameters
)
```

参数

- **name** 用于新队列的名称。
- **parameters** 创建参数；目前未使用，设置为 NULL。

注释

队列名的使用范围仅限于每个连接，所以不同的连接可以创建具有相同名称的队列。发送事件通知时，数据库中具有匹配名称的所有队列都接收通知（单独实例）。名称不区分大小写。如果没指定任何队列，则调用“RegisterForEvent 函数”一节第 162 页时按需为每个连接创建缺省队列。如果名称已存在或无效，则调用失败并出现错误。

另请参见

- “CancelGetNotification 函数”一节第 150 页
- “DeclareEvent 函数”一节第 152 页
- “DestroyNotificationQueue 函数”一节第 153 页
- “GetNotification 函数”一节第 156 页

DeclareEvent 函数

声明稍后可以进行注册和触发的事件。

语法

```
bool UltraLite_Connection_iface::DeclareEvent(
  const ULValue & event_name
)
```

参数

- **event_name** 用户定义的新事件的名称

返回值

如果成功声明事件，则返回 ul_true；如果名称已经使用或无效，则返回 ul_false。

注释

UltraLite 预定义一些由对数据库或环境的操作触发的系统事件。此函数会声明用户定义的事件。用户定义的事件由“TriggerEvent 函数”一节第 170 页触发。事件的名称必须是唯一的。名称不区分大小写。

另请参见

- “CancelGetNotification 函数”一节第 150 页
- “DestroyNotificationQueue 函数”一节第 153 页
- “GetNotification 函数”一节第 156 页
- “RegisterForEvent 函数”一节第 162 页
- “SendNotification 函数”一节第 164 页
- “TriggerEvent 函数”一节第 170 页

DestroyNotificationQueue 函数

取消给定的事件通知队列。

语法

```
bool UltraLite_Connection_iface::DestroyNotificationQueue(
  const ULValue & name
)
```

参数

- **name** 要取消的队列的名称

注释

如果未读通知仍然在队列中，则会发出警告。未读通知将被放弃。如果已创建连接的缺省事件队列，则在连接关闭时会将其取消。

另请参见

- “CancelGetNotification 函数”一节第 150 页
- “DeclareEvent 函数”一节第 152 页
- “GetNotification 函数”一节第 156 页
- “RegisterForEvent 函数”一节第 162 页
- “SendNotification 函数”一节第 164 页
- “TriggerEvent 函数”一节第 170 页

ExecuteNextSQLPassthroughScript 函数

执行下一个 SQL 直通脚本。

语法

```
bool UltraLite_Connection_iface::ExecuteNextSQLPassthroughScript()
```

返回值

执行脚本时如果发生任何错误，则返回 false

另请参见

- “ExecuteSQLPassthroughScripts 函数”一节第 154 页
- “GetSQLPassthroughScriptCount 函数”一节第 158 页

ExecuteSQLPassthroughScripts 函数

执行所有可用的 SQL 直通脚本。

语法

bool **UltraLite_Connection_iface::ExecuteSQLPassthroughScripts()**

返回值

执行脚本时如果发生错误，则返回 false

另请参见

- “ExecuteNextSQLPassthroughScript 函数”一节第 153 页
- “GetSQLPassthroughScriptCount 函数”一节第 158 页

GetConnectionNum 函数

获取连接号。

语法

ul_connection_num **UltraLite_Connection_iface::GetConnectionNum()**

GetDatabaseID 函数

获取用于全局自动增量列的数据库 ID。

语法

ul_u_long **UltraLite_Connection_iface::GetDatabaseID()**

GetDatabaseProperty 函数

获取数据库属性。

语法

```
ULValue UltraLite_Connection_iface::GetDatabaseProperty(
  ul_database_property_id id
)
```


参数

- **id** 所请求的属性的 ID。

返回值

所请求的属性的值。

GetDatabaseProperty 函数

获取数据库属性。

语法

```
ULValue UltraLite_Connection_iface::GetDatabaseProperty(
  const ULValue & prop_name
)
```

参数

- **prop_name** 所请求的属性的字符串名称。

返回值

所请求的属性的值。

GetLastDownloadTime 函数

获取上一次下载的时间。

语法

```
bool UltraLite_Connection_iface::GetLastDownloadTime(
  const ULValue & pub_list,
  DECL_DATETIME * value
)
```

参数

- **pub_list** 以逗号分隔的要考虑的发布的列表。
- **value** 上次下载时间。

GetLastIdentity 函数

获取 @@identity 值。

语法

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

GetNewUUID 函数

创建新的 UUID。

语法

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

参数

- **uuid** 新的 UUID 值。

GetNewUUID 函数

创建新的 UUID。

语法

bool **UltraLite_Connection_iface::GetNewUUID(**
  GUID * *uuid*
)

参数

- **uuid** 新的 UUID 值。

GetNotification 函数

读取事件通知。

语法

ULValue **UltraLite_Connection_iface::GetNotification(**
  const ULValue & *queue_name*,
  ul_u_long *wait_ms*
)

参数

- **queue_name** 要读取的队列，或者对于缺省连接队列为 NULL
- **wait_ms** 返回前的等待（阻塞）时间

返回值

所读取的事件的名称，或者在出错时为空字符串。

注释

此调用将阻塞直到收到通知或给定等待时期到期。要进行无限期的等待，将 UL_READ_WAIT_INFINITE 传递给 wait_ms。要取消等待，发送另一通知到给定队列或使用“CancelGetNotification 函数”一节第 150 页。读取通知后，使用 ReadNotificationParameter() 按名称来检索其它参数。

另请参见

- “CancelGetNotification 函数”一节第 150 页
- “DeclareEvent 函数”一节第 152 页
- “DestroyNotificationQueue 函数”一节第 153 页
- “RegisterForEvent 函数”一节第 162 页
- “SendNotification 函数”一节第 164 页
- “TriggerEvent 函数”一节第 170 页

GetNotificationParameter 函数

获取刚刚由“GetNotification 函数”一节第 156 页读取的事件通知的参数。

语法

```
ULValue UltraLite_Connection_iface::GetNotificationParameter(
  const ULValue & queue_name,
  const ULValue & parameter_name
)
```

参数

- **queue_name** 与“GetNotification 函数”一节第 156 页调用匹配的队列名称
- **parameter_name** 要读取的参数名称（或 "*"）

返回值

参数值，或在出错时为空字符串。

注释

只有给定队列中来自最近读取的通知的参数可用。参数会按名称检索。使用参数名称 "*" 可检索整个参数字符串。

另请参见

- “CancelGetNotification 函数”一节第 150 页
- “DeclareEvent 函数”一节第 152 页
- “DestroyNotificationQueue 函数”一节第 153 页
- “GetNotification 函数”一节第 156 页
- “RegisterForEvent 函数”一节第 162 页
- “SendNotification 函数”一节第 164 页
- “TriggerEvent 函数”一节第 170 页

GetSQLPassthroughScriptCount 函数

获取可运行的 SQL 直通脚本的数目。

语法

ul_u_long **UltraLite_Connection_iface::GetSQLPassthroughScriptCount()**

另请参见

- “ExecuteNextSQLPassthroughScript 函数”一节第 153 页
- “ExecuteSQLPassthroughScripts 函数”一节第 154 页

GetSchema 函数

获取数据库模式。

语法

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

GetSqlca 函数

获取与此连接关联的通信区。

语法

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

GetSuspend 函数（不建议使用）

获取 Suspend 属性。

语法

bool **UltraLite_Connection_iface::GetSuspend()**

返回值

- 如果此连接被暂停，则返回 true。
- 如果此连接未被暂停，则返回 false。

GetSynchResult 函数

获取上一次同步的结果。

语法

bool **UltraLite_Connection_iface::GetSynchResult(**
  ul_synch_result * *synch_result*
)

参数

- **synch_result** 指向保存同步结果的“ul_synch_result 结构”一节第 131 页结构的指针。

GetUtilityULValue 函数

获取一个新的“ULValue 类”一节第 232 页实例。

语法

ULValue **UltraLite_Connection_iface::GetUtilityULValue()**

注释

必须将“ULValue 类”一节第 232 页对象绑定到一个连接，才能使它的许多方法获得成功。

GlobalAutoincUsage 函数

获取计数器使用的全局自动增量值的百分比。

语法

ul_u_short **UltraLite_Connection_iface::GlobalAutoincUsage()**

GrantConnectTo 函数

授予给定用户连接权限。

语法

bool **UltraLite_Connection_iface::GrantConnectTo(**
  const ULValue & *uid*,
  const ULValue & *pwd*
)

参数

- **uid** 要授予其连接访问权限的用户 ID。
- **pwd** 已授权用户 ID 的口令。

注释

要创建新用户，请指定新用户 ID 和口令。

要更改口令，请指定现有的用户 ID，但为该用户设置一个新口令。

InitSynchInfo 函数

初始化同步信息结构。

语法

```
void UltraLite_Connection_iface::InitSynchInfo(
  ul_synch_info_a * info
)
```

参数

- **info** 指向保存同步参数的 ul_synch_info 结构的指针。

InitSynchInfo 函数

初始化同步信息结构。

语法

```
void UltraLite_Connection_iface::InitSynchInfo(
  ul_synch_info_w2 * info
)
```

参数

- **info** 指向保存同步参数的 ul_synch_info 结构的指针。

OpenTable 函数

打开表。

语法

```
UltraLite_Table * UltraLite_Connection_iface::OpenTable(
  const ULValue & table_id,
  const ULValue & persistent_name
)
```

参数

- **table_id** 表的名称或序号。
- **persistent_name** 用于暂停的实例名称。

注释

当应用程序首次打开表时，游标位置设置为 BeforeFirst()。

OpenTableEx 函数

打开表以检索行。

语法

```
UltraLite_Table * UltraLite_Connection_iface::OpenTableEx(
  const ULValue & table_id,
  ul_table_open_type open_type,
  const ULValue & parms,
  const ULValue & persistent_name
)
```

参数

- **table_id** 表的名称或序号。
- **open_type** 控制如何返回行。
- **parms** 取决于打开类型（如索引名称）的可选参数
- **persistent_name** 用于暂停的实例名称。

注释

行的顺序取决于用于打开表的索引。如果没有使用索引，行以任意顺序排列。当应用程序首次打开表时，游标位置设置为 BeforeFirst()。

不使用索引打开表能够提高性能。但是，当不使用索引时，返回的表无法用于变更数据并且无法执行查找。

OpenTableWithIndex 函数

打开表，使用指定的索引对行进行排序。

语法

```
UltraLite_Table * UltraLite_Connection_iface::OpenTableWithIndex(
  const ULValue & table_id,
  const ULValue & index_id,
  const ULValue & persistent_name
)
```

参数

- **table_id** 表的名称或序号。
- **index_id** 索引的名称或序号。
- **persistent_name** 用于暂停的实例名称。

注释

当应用程序首次打开表时，游标位置设置为 BeforeFirst()。

PrepareStatement 函数

准备 SQL 语句。

语法

```
UltraLite_PreparedStatement * UltraLite_Connection_iface::PrepareStatement(
  const ULValue & sql,
  const ULValue & persistent_name
)
```

参数

- **sql** 作为字符串的 SQL 语句。
- **persistent_name** 用于暂停的实例名称。

RegisterForEvent 函数

注册或注销（队列）以便接收事件的通知。

语法

```
bool UltraLite_Connection_iface::RegisterForEvent(
  const ULValue & event_name,
  const ULValue & object_name,
  const ULValue & queue_name,
  bool register_not_unreg
)
```

参数

- **event_name** 要注册的系统事件或用户定义的事件
- **object_name** 应用事件的对象（如表名）
- **queue_name** NULL 表示缺省连接队列
- **register_not_unreg** true 表示注册，false 表示注销

返回值

如果注册成功，则返回 true；如果队列或事件不存在，则返回 false。

注释

如果不提供队列名称，则表示采用缺省连接队列，如果需要，可以创建缺省连接队列。某些系统事件允许指定应用事件的对象名称。例如，TableModified 事件可以指定表名称。与“SendNotification 函数”一节第 164 页不同，仅特定的注册队列会收到事件的通知 - 不同连接中同名的其它队列则不会收到通知（除非它们也经过显式注册）。

预定义的系统事件为：

- TableModified - 插入、更新或删除表中的行时触发。每次请求时发送一个通知，无论多少行受到该请求的影响。object_name 参数指定要监控的表。"*" 值表示数据库中的所有表。此事件拥有名为 "table_name" 的参数，它的值是已修改表的名称。
- Commit - 任意提交完成后触发。此事件无参数。
- SyncComplete - 同步完成后触发。此事件无参数。

另请参见

- “CancelGetNotification 函数”一节第 150 页
- “DeclareEvent 函数”一节第 152 页
- “DestroyNotificationQueue 函数”一节第 153 页
- “GetNotification 函数”一节第 156 页
- “SendNotification 函数”一节第 164 页
- “TriggerEvent 函数”一节第 170 页

ResetLastDownloadTime 函数

重置指定发布的上次下载时间。

语法

```
bool UltraLite_Connection_iface::ResetLastDownloadTime(
  const ULValue & pub_list
)
```

参数

- **pub_list** 要重置的发布。

RevokeConnectFrom 函数

删除现有用户。

语法

```
bool UltraLite_Connection_iface::RevokeConnectFrom(
  const ULValue & uid
)
```

参数

- **uid** 要撤消连接权限的用户 ID。

Rollback 函数

回退当前事务。

语法

bool **UltraLite_Connection_iface::Rollback()**

RollbackPartialDownload 函数

回退部分下载。

语法

bool **UltraLite_Connection_iface::RollbackPartialDownload()**

SendNotification 函数

将通知发送到与给定名称匹配的所有队列。

语法

```
ul_u_long UltraLite_Connection_iface::SendNotification(
  const ULValue & queue_name,
  const ULValue & event_name,
  const ULValue & parameters
)
```

参数

- **queue_name** 目标队列名（或 "*"）。
- **event_name** 通知的标识
- **parameters** 参数选项列表或 NULL

返回值

已发送的通知数（匹配队列的数目）。

注释

这包括当前连接中任何此类队列。此调用不会阻塞。使用特殊队列名称 "*" 发送到所有队列。给定的事件名称不需要与任何系统或用户定义的事件对应；传递它仅仅是为读取时标识通知，并且它只对发送者和接收者有意义。这些参数变量会指定分号分隔的 "名称=值" 对选项列表。在读取通知后，参数值使用“GetNotificationParameter 函数”一节第 157 页读取。

另请参见

- “CancelGetNotification 函数”一节第 150 页
- “DeclareEvent 函数”一节第 152 页
- “DestroyNotificationQueue 函数”一节第 153 页
- “GetNotification 函数”一节第 156 页
- “RegisterForEvent 函数”一节第 162 页
- “TriggerEvent 函数”一节第 170 页

SetDatabaseID 函数

设置用于全局自动增量列的数据库 ID。

语法

```
bool UltraLite_Connection_iface::SetDatabaseID(
  ul_u_long value
)
```

参数

- **value** 确定全局自动增量列的初始值的数据库 ID。

SetDatabaseOption 函数

设置指定的数据库选项。

语法

```
bool UltraLite_Connection_iface::SetDatabaseOption(
  ul_database_option_id id,
  const ULValue & value
)
```

参数

- **id** 要设置的选项的 ID。
- **value** 选项的新值。

SetDatabaseOption 函数

设置指定的数据库选项。

语法

```
bool UltraLite_Connection_iface::SetDatabaseOption(
  const ULValue & option_name,
  const ULValue & value
)
```

参数

- **option_name** 所设置的选项的字符串名称。
- **value** 选项的新值。

SetSuspend 函数（不建议使用）

设置 Suspend 属性。

语法

```
void UltraLite_Connection_iface::SetSuspend(
  bool suspend
)
```

参数

- **suspend** 设置为 true 会暂停连接，以便重新打开数据库时可以恢复该连接的状态。

返回值

如果该连接被暂停，则返回 true。

注释

连接名（或缺少连接名）标识暂停的连接。

SetSynchInfo 函数

使用给定名称基于给定的 ul_synch_info 结构创建同步配置文件。

语法

```
bool UltraLite_Connection_iface::SetSynchInfo(
  char const * profile_name,
  ul_synch_info_a * info
)
```

参数

- **profile_name** 包含同步选项的同步配置文件的名称。如果 profile_name 为空值，则不会使用配置文件并且信息结构应包含所有用于同步的选项。
- **info** 指向保存同步参数的 ul_synch_info 结构的指针。

注释

这样会用此名称替换之前的任何同步配置文件。指定 ul_synch_info 的空指针将删除指定的配置文件。

另请参见

- “SynchronizeFromProfile 函数”一节第 169 页

SetSynchInfo 函数

使用给定名称基于给定的 ul_synch_info 结构创建同步配置文件。

语法

bool **UltraLite_Connection_iface::SetSynchInfo(**
ul_wchar const * *profile_name*,
ul_synch_info_w2 * *info*
)

参数

- **profile_name** 包含同步选项的同步配置文件的名称。如果 profile_name 为空值，则不会使用配置文件并且信息结构应包含所有用于同步的选项。
- **info** 指向保存同步参数的 ul_synch_info 结构的指针。

另请参见

- “SynchronizeFromProfile 函数”一节第 169 页

Shutdown 函数

取消此连接以及任何剩余的关联对象。

语法

void **UltraLite_Connection_iface::Shutdown()**

注释

如果未将此连接设置为暂停，则会回退它。

StartSynchronizationDelete 函数

设置此连接的 START SYNCHRONIZATION DELETE。

语法

bool **UltraLite_Connection_iface::StartSynchronizationDelete()**

另请参见

- “StopSynchronizationDelete 函数”一节第 167 页

StopSynchronizationDelete 函数

设置此连接的 STOP SYNCHRONIZATION DELETE。

语法

bool **UltraLite_Connection_iface::StopSynchronizationDelete()**

另请参见

- “StartSynchronizationDelete 函数”一节第 167 页

StrToUUID 函数

将字符串转换为二进制 UUID。

语法

```
bool UltraLite_Connection_iface::StrToUUID(
  p_ul_binary dst,
  size_t len,
  const ULValue & src
)
```

参数

- **dst** 要返回的 UUID 值。
- **len** ul_binary 数组的长度。
- **src** 保存要进行转换的 UUID 值的字符串。

StrToUUID 函数

将字符串转换为 GUID 结构。

语法

```
bool UltraLite_Connection_iface::StrToUUID(
  GUID * dst,
  const ULValue & src
)
```

参数

- **dst** 要返回的 GUID 值。
- **src** 保存要进行转换的 UUID 值的字符串。

Synchronize 函数

同步数据库。

语法

```
bool UltraLite_Connection_iface::Synchronize(
  ul_synch_info_a * info
)
```


参数

- **info** 指向保存同步参数的 ul_synch_info 结构的指针。

示例

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

Synchronize 函数

同步数据库。

语法

bool **UltraLite_Connection_iface::Synchronize(**
ul_synch_info_w2 * *info*
)

参数

- **info** 指向保存同步参数的 ul_synch_info 结构的指针。

另请参见

- “Synchronize 函数”一节第 168 页

SynchronizeFromProfile 函数

使用指定的配置文件和合并参数同步数据库。

语法

bool **UltraLite_Connection_iface::SynchronizeFromProfile(**
ULValue const & *profile_name*,
ULValue const & *merge_parms*
)

参数

- **profile_name** 要同步的配置文件名称。
- **merge_parms** 用于同步的合并参数

注释

这与执行 SYNCHRONIZE 语句相同。

TriggerEvent 函数

触发用户定义的事件（并将通知发送到所有已注册的队列）。

语法

```
ul_u_long UltraLite_Connection_iface::TriggerEvent(
  const ULValue & event_name,
  const ULValue & parameters
)
```

参数

- **event_name** 要触发的系统事件或用户定义的事件的名称
- **parameters** 参数选项列表或 NULL

返回值

已发送的事件通知的数目。

注释

这些参数值会指定分号分隔的 "名称=值" 对选项列表。在读取通知后，参数值使用 GetNotificationParameter 读取。

另请参见

- “CancelGetNotification 函数”一节第 150 页
- “DeclareEvent 函数”一节第 152 页
- “DestroyNotificationQueue 函数”一节第 153 页
- “GetNotification 函数”一节第 156 页
- “RegisterForEvent 函数”一节第 162 页
- “SendNotification 函数”一节第 164 页

UUIDToStr 函数

将 UUID 转换为 ANSI 字符串。

语法

```
bool UltraLite_Connection_iface::UUIDToStr(
  char * dst,
  size_t len,
  const ULValue & src
)
```

参数

- **dst** 要返回的字符串。
- **len** ul_binary 数组的长度。
- **src** 要转换为字符串的 UUID 值。

UUIDToStr 函数

将 UUID 转换为 Unicode 字符串。

语法

```
bool UltraLite_Connection_iface::UUIDToStr(
  ul_wchar * dst,
  size_t len,
  const ULValue & src
)
```

参数

- **dst** 要返回的 Unicode 字符串。
- **len** ul_binary 数组的长度。
- **src** 要转换为字符串的 UUID 值。

ValidateDatabase 函数

校验此连接上的数据库。

语法

```
bool UltraLite_Connection_iface::ValidateDatabase(
  ul_u_short flags,
  ul_validate_callback_fn fn,
  ul_void * user_data,
  const ULValue & table_id
)
```

参数

- **flags** 控制校验类型的标记
- **fn** 接收校验进度信息的函数
- **user_data** 通过回调发送回调用者的用户数据
- **table_id** 要校验的特定表（可选）

返回值

如果在校验的过程中出错，则返回 false。

注释

根据传递给此例程的标记不同，可校验的低级别存储和/或索引。要在校验期间接收信息，执行回调函数并且将地址传递给此例程。要限制对指定表的验证，将表名或 ID 作为最后的参数传递。

UltraLite_Cursor_iface 类

表示 UltraLite 数据库内的一个双向游标。

语法

public **UltraLite_Cursor_iface**

派生类

- “UltraLite_ResultSet 类”一节第 198 页
- “UltraLite_Table 类”一节第 213 页

成员

UltraLite_Cursor_iface 的所有成员（包括所有继承成员）。

- “AfterLast 函数”一节第 172 页
- “BeforeFirst 函数”一节第 173 页
- “Delete 函数”一节第 173 页
- “First 函数”一节第 173 页
- “Get 函数”一节第 173 页
- “GetRowCount 函数”一节第 174 页
- “GetState 函数”一节第 174 页
- “GetStreamReader 函数”一节第 174 页
- “GetStreamWriter 函数”一节第 175 页
- “GetSuspend 函数（不建议使用）”一节第 175 页
- “IsNull 函数”一节第 175 页
- “Last 函数”一节第 175 页
- “Next 函数”一节第 176 页
- “Previous 函数”一节第 176 页
- “Relative 函数”一节第 176 页
- “Set 函数”一节第 176 页
- “SetDefault 函数”一节第 177 页
- “SetNull 函数”一节第 177 页
- “SetSuspend 函数（不建议使用）”一节第 177 页
- “Update 函数”一节第 178 页
- “UpdateBegin 函数”一节第 178 页

注释

游标是来自表的行集或来自查询的结果集的行集。

AfterLast 函数

将游标移到最后一行的后面。

语法

bool **UltraLite_Cursor_iface::AfterLast()**

BeforeFirst 函数

将游标移到第一行的前面。

语法

bool **UltraLite_Cursor_iface::BeforeFirst()**

Delete 函数

删除当前行并将其移动到下一个有效行。

语法

bool **UltraLite_Cursor_iface::Delete()**

注释

如果此游标是未使用索引打开的表，那么数据将视为只读并且无法删除行。

First 函数

将游标移动到第一行。

语法

bool **UltraLite_Cursor_iface::First()**

Get 函数

从列中读取值。

语法

ULValue **UltraLite_Cursor_iface::Get(**
  const ULValue & *column_id*
)

参数

- **column_id** 列的名称或序号。

GetRowCount 函数

获取表中的行数。

语法

ul_u_long **UltraLite_Cursor_iface::GetRowCount(**
[ul_u_long *threshold*]
)

参数

- **threshold** 设置要计数的行的阈值的可选参数。如果应用了阈值并且大于零，则返回的最大行数为阈值。实际行数可以等于或大于阈值。

注释

在了解是否存在多于指定行数的行很重要的情况下，设置阈值以限制计数的行数很有用。例如，填充包含 25 个项目的列表的代码可以确定是否有必要允许用户选择查看其它行。

调用此方法等效于执行 "SELECT COUNT(*) FROM table"。

GetState 函数

获取游标的内部状态。

语法

UL_RS_STATE **UltraLite_Cursor_iface::GetState()**

注释

请参见 ulglobal.h 中的枚举 UL_RS_STATE

GetStreamReader 函数

获取用于读取块中的字符串或二进制列数据的流读取程序对象。

语法

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

参数

- **id** 列标识符，可以是从 1 开始的序号或列名。

GetStreamWriter 函数

获取一个用于将字符串/二进制数据流式写入列中的流写入器。

语法

```
UltraLite_StreamWriter * UltraLite_Cursor_iface::GetStreamWriter(
  const ULValue & column_id
)
```

参数

- **column_id** 列标识符，可以是从 1 开始的序号或列名。

GetSuspend 函数（不建议使用）

获取 Suspend 属性的值。

语法

```
bool UltraLite_Cursor_iface::GetSuspend()
```

返回值

- 如果此游标被暂停，则返回 true。
- 否则，返回 false。

IsNull 函数

检查列是否为 NULL。

语法

```
bool UltraLite_Cursor_iface::IsNull(
  const ULValue & column_id
)
```

参数

- **column_id** 列的名称或序号。

Last 函数

将游标移动到最后一行。

语法

```
bool UltraLite_Cursor_iface::Last()
```

Next 函数

将游标向前移动一行。

语法

bool **UltraLite_Cursor_iface::Next()**

返回值

- 如果游标向前移动成功，则返回 true。尽管返回 true，但当游标成功移动到下一行时也有可能发出错误信号。例如，在计算 SELECT 表达式时，可能发生转换错误。在这种情况下，检索列值时也会返回错误。
- 如果向前移动失败，则返回 false。例如，可能没有下一行。在这种情况下，游标的最终位置为“AfterLast 函数”一节第 172 页。

Previous 函数

将游标向后移动一行。

语法

bool **UltraLite_Cursor_iface::Previous()**

注释

如果失败，则游标的最终位置为“BeforeFirst 函数”一节第 173 页。

Relative 函数

将游标从当前游标位置移动由 offset 参数所指定的行。

语法

bool **UltraLite_Cursor_iface::Relative(**
ul_fetch_offset *offset*
)

参数

- **offset** 要移动的行数。

Set 函数

设置列值。

语法

```
bool UltraLite_Cursor_iface::Set(
  const ULValue & column_id,
  const ULValue & value
)
```

参数

- **column_id** 用于标识列的从 1 开始的序号。
- **value** 要为列设置的值。

SetDefault 函数

将列设置为其缺省值。

语法

```
bool UltraLite_Cursor_iface::SetDefault(
  const ULValue & column_id
)
```

参数

- **column_id** 用于标识列的从 1 开始的序号。

SetNull 函数

将列设置为空。

语法

```
bool UltraLite_Cursor_iface::SetNull(
  const ULValue & column_id
)
```

参数

- **column_id** 用于标识列的从 1 开始的序号。

SetSuspend 函数（不建议使用）

设置 Suspend 属性的值。

语法

```
void UltraLite_Cursor_iface::SetSuspend(
  bool suspend
)
```

参数

- **suspend** 如果该连接被暂停，则返回 true。此函数允许您在重新打开数据库时恢复数据库的状态。

返回值

- 如果游标被暂停并将在应用程序重新打开数据库时恢复，则返回 true。
- 如果游标未被暂停，则返回 false。

注释

当打开关联对象以标识挂起的游标时，使用持久名称参数。如果没有为此游标提供持久名称参数，将无法暂停此游标。

Update 函数

更新当前行。

语法

bool **UltraLite_Cursor_iface::Update()**

注释

表必须处于更新模式，此操作才能成功。使用“UpdateBegin 函数”一节第 178 页可切换到更新模式。

UpdateBegin 函数

选择用于设置列的更新模式。

语法

bool **UltraLite_Cursor_iface::UpdateBegin()**

注释

在更新模式下不能修改主键中的列。如果此游标是未使用索引打开的表，那么数据将视为只读并且无法修改。

UltraLite_DatabaseManager 类

管理同步监听器并允许删除 UltraLite 数据库。

语法

public **UltraLite_DatabaseManager**

基类

- “UltraLite_DatabaseManager_iface 类”一节第 180 页

成员

UltraLite_DatabaseManager 的所有成员（包括所有继承成员）。

- “CreateDatabase 函数”一节第 180 页
- “DropDatabase 函数”一节第 181 页
- “OpenConnection 函数”一节第 181 页
- “Shutdown 函数”一节第 182 页
- “ValidateDatabase 函数”一节第 182 页

UltraLite_DatabaseManager_iface 类

管理连接和数据库。

语法

public **UltraLite_DatabaseManager_iface**

派生类

- “UltraLite_DatabaseManager 类”一节第 179 页

成员

UltraLite_DatabaseManager_iface 的所有成员（包括所有继承成员）。

- “CreateDatabase 函数”一节第 180 页
- “DropDatabase 函数”一节第 181 页
- “OpenConnection 函数”一节第 181 页
- “Shutdown 函数”一节第 182 页
- “ValidateDatabase 函数”一节第 182 页

注释

创建数据库并建立与数据库的连接是使用 UltraLite 中必不可少的第一步。在对数据库尝试任何 DML 之前，应确保已正确连接。

CreateDatabase 函数

创建新数据库。

语法

```
bool UltraLite_DatabaseManager_iface::CreateDatabase(
  ULSqlcaBase & sqlca,
  ULValue const & access_parms,
  void const * coll,
  ULValue const & create_parms,
  void * reserved
)
```

参数

- **sqlca**　已初始化的 sqlca。
- **access_parms**　用于访问数据库的连接参数
- **coll**　归类序列
- **create_parms**　用于创建数据库的参数
- **保留**　保留（目前未使用）

DropDatabase 函数

消除已停止的现有数据库。

语法

```
bool UltraLite_DatabaseManager_iface::DropDatabase(
  ULSqlcaBase & sqlca,
  const ULValue & parms_string
)
```

参数

- **sqlca** 已初始化的 sqlca。
- **parms_string** 数据库标识参数。

注释

不能消除正在运行的数据库。

OpenConnection 函数

打开与现有数据库的新连接。

语法

```
UltraLite_Connection * UltraLite_DatabaseManager_iface::OpenConnection(
  ULSqlcaBase & sqlca,
  ULValue const & parms_string
)
```

参数

- **sqlca** 要与新连接相关联的已初始化的 sqlca。
- **parms_string** 连接字符串。

注释

给定 sqlca 与新连接相关联。

- SQLE_CONNECTION_ALREADY_EXISTS - 已存在具有给定 SQLCA 和名称（或没有名称）的连接。在连接前，必须断开现有的连接，或使用 CON 参数指定不同的连接名。
- SQLE_INVALID_LOGON - 提供的用户 ID 无效或口令不正确。
- SQLE_INVALID_SQL_IDENTIFIER - 通过 C 语言接口提供了无效的标识符。例如，可能为游标名称提供了 NULL 字符串。
- SQLE_TOO_MANY_CONNECTIONS - 已超过并发数据库连接数。

要获取错误信息，请使用相关的“ULSqlca 类”一节第 136 页对象。可能的错误包括：

返回值

- 如果函数成功，将返回新的连接对象。
- 如果函数失败，则返回 NULL。

Shutdown 函数

关闭所有数据库并释放数据库管理器。

语法

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

参数

- **sqlca** 已初始化的 sqlca。

注释

任何剩余的关联对象都将被取消。调用此函数后，无法再使用数据库管理器（也无法使用任何其它以前获取的对象）。

ValidateDatabase 函数

在数据库上执行低级校验和索引校验。

语法

```
bool UltraLite_DatabaseManager_iface::ValidateDatabase(
  ULSqlcaBase & sqlca,
  ULValue const & start_parms,
  ul_u_short flags,
  ul_validate_callback_fn fn,
  ul_void * user_data
)
```

参数

- **sqlca** 已初始化的 sqlca。
- **start_parms** 用于启动数据库的参数
- **flags** 控制校验类型的标记
- **fn** 接收校验进度信息的函数
- **user_data** 通过回调发送回调用者的用户数据

UltraLite_DatabaseSchema 类

表示 UltraLite 数据库的模式。

语法

public **UltraLite_DatabaseSchema**

基类

- “UltraLite_SQLObject_iface 类”一节第 206 页
- “UltraLite_DatabaseSchema_iface 类”一节第 184 页

成员

UltraLite_DatabaseSchema 的所有成员（包括所有继承成员）。

- “AddRef 函数”一节第 206 页
- “GetCollationName 函数”一节第 184 页
- “GetConnection 函数”一节第 206 页
- “GetIFace 函数”一节第 207 页
- “GetPublicationCount 函数”一节第 184 页
- “GetPublicationID 函数”一节第 185 页
- “GetPublicationName 函数”一节第 185 页
- “GetTableCount 函数”一节第 185 页
- “GetTableName 函数”一节第 185 页
- “GetTableSchema 函数”一节第 186 页
- “IsCaseSensitive 函数”一节第 186 页
- “Release 函数”一节第 207 页

UltraLite_DatabaseSchema_iface 类

DatabaseSchema 接口。

语法

public **UltraLite_DatabaseSchema_iface**

派生类

- “UltraLite_DatabaseSchema 类”一节第 183 页

成员

UltraLite_DatabaseSchema_iface 的所有成员（包括所有继承成员）。

- “GetCollationName 函数”一节第 184 页
- “GetPublicationCount 函数”一节第 184 页
- “GetPublicationID 函数”一节第 185 页
- “GetPublicationName 函数”一节第 185 页
- “GetTableCount 函数”一节第 185 页
- “GetTableName 函数”一节第 185 页
- “GetTableSchema 函数”一节第 186 页
- “IsCaseSensitive 函数”一节第 186 页

GetCollationName 函数

获取当前归类序列的名称。

语法

ULValue **UltraLite_DatabaseSchema_iface::GetCollationName()**

返回值

包含字符串的“ULValue 类”一节第 232 页。

GetPublicationCount 函数

获取数据库中的发布数目。

语法

ul_publication_count **UltraLite_DatabaseSchema_iface::GetPublicationCount()**

注释

发布 ID 的范围是从 1 到“GetPublicationCount 函数”一节第 184 页

GetPublicationID 函数

在已知发布的名称的情况下获取从 1 开始的发布 ID。

语法

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

参数

- **pub_id** 从 1 开始的序号。

GetPublicationName 函数

在已知发布的从 1 开始的索引 ID 的情况下获取发布的名称。

语法

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

参数

- **pub_id** 从 1 开始的序号。

GetTableCount 函数

返回数据库中表的数目。

语法

ul_table_num **UltraLite_DatabaseSchema_iface::GetTableCount()**

返回值

- 表示表的数目的整数。
- 如果未打开连接，则返回 0。

GetTableName 函数

已知从 1 开始的表 ID，获取表的名称。

语法

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

参数

- **tableID** 从 1 开始的序号。

返回值

由指定的表 ID 所标识的表的名称。

注释

表 ID 在模式升级过程中可发生变化。为了正确地标识表，请按名称访问它，或者在模式升级后刷新高速缓存中的 ID。如果表不存在，则返回的“ULValue 类”一节第 232 页对象为空。

GetTableSchema 函数

在已知表 ID（从 1 开始）或名称的情况下获取 TableSchema 对象。

语法

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

参数

- **table_id** 从 1 开始的序号。

返回值

如果表不存在，则返回 UL_NULL。

IsCaseSensitive 函数

获取数据库的区分大小写特性。

语法

bool **UltraLite_DatabaseSchema_iface::IsCaseSensitive()**

返回值

- 如果数据库区分大小写，则返回 true。
- 否则，返回 false。

注释

数据库是否区分大小写将影响表索引和结果集索引的排序方式。

UltraLite_IndexSchema 类

表示 UltraLite 表索引的模式。

语法

public **UltraLite_IndexSchema**

基类

- “UltraLite_SQLObject_iface 类”一节第 206 页
- “UltraLite_IndexSchema_iface 类”一节第 188 页

成员

UltraLite_IndexSchema 的所有成员（包括所有继承成员）。

- “AddRef 函数”一节第 206 页
- “GetColumnCount 函数”一节第 188 页
- “GetColumnName 函数”一节第 188 页
- “GetConnection 函数”一节第 206 页
- “GetID 函数”一节第 189 页
- “GetIFace 函数”一节第 207 页
- “GetName 函数”一节第 189 页
- “GetReferencedIndexName 函数”一节第 189 页
- “GetReferencedTableName 函数”一节第 190 页
- “GetTableName 函数”一节第 190 页
- “IsColumnDescending 函数”一节第 190 页
- “IsForeignKey 函数”一节第 190 页
- “IsForeignKeyCheckOnCommit 函数”一节第 191 页
- “IsForeignKeyNullable 函数”一节第 191 页
- “IsPrimaryKey 函数”一节第 191 页
- “IsUniqueIndex 函数”一节第 192 页
- “IsUniqueKey 函数”一节第 192 页
- “Release 函数”一节第 207 页

UltraLite_IndexSchema_iface 类

表示 IndexSchema 接口。

语法

public **UltraLite_IndexSchema_iface**

派生类

- “UltraLite_IndexSchema 类”一节第 187 页

成员

UltraLite_IndexSchema_iface 的所有成员（包括所有继承成员）。

- “GetColumnCount 函数”一节第 188 页
- “GetColumnName 函数”一节第 188 页
- “GetID 函数”一节第 189 页
- “GetName 函数”一节第 189 页
- “GetReferencedIndexName 函数”一节第 189 页
- “GetReferencedTableName 函数”一节第 190 页
- “GetTableName 函数”一节第 190 页
- “IsColumnDescending 函数”一节第 190 页
- “IsForeignKey 函数”一节第 190 页
- “IsForeignKeyCheckOnCommit 函数”一节第 191 页
- “IsForeignKeyNullable 函数”一节第 191 页
- “IsPrimaryKey 函数”一节第 191 页
- “IsUniqueIndex 函数”一节第 192 页
- “IsUniqueKey 函数”一节第 192 页

GetColumnCount 函数

获取索引中列的数目。

语法

ul_column_num **UltraLite_IndexSchema_iface::GetColumnCount()**

GetColumnName 函数

获取在索引中给定列位置的列的名称。

语法

ULValue **UltraLite_IndexSchema_iface::GetColumnName(**
  ul_column_num *col_id_in_index*
)

参数

- **col_id_in_index** 从 1 开始的序号，表示列在索引中的位置。

返回值

如果列不存在，则返回空的“ULValue 类”一节第 232 页对象。

注释

如果列名不存在，则返回 SQLE_COLUMN_NOT_FOUND。

GetID 函数

获取索引 ID。

语法

ul_index_num **UltraLite_IndexSchema_iface::GetID()**

返回值

索引的 ID。

GetName 函数

获取索引的名称。

语法

ULValue **UltraLite_IndexSchema_iface::GetName()**

GetReferencedIndexName 函数

获取关联的主索引名。

语法

ULValue **UltraLite_IndexSchema_iface::GetReferencedIndexName()**

返回值

如果索引不是外键，则返回空的“ULValue 类”一节第 232 页对象。

注释

此函数仅适用于外键。

GetReferencedTableName 函数

获取关联的主表名。

语法

ULValue **UltraLite_IndexSchema_iface::GetReferencedTableName()**

返回值

如果索引不是外键，则返回空的“ULValue 类”一节第 232 页对象。

注释

此方法仅适用于外键。

GetTableName 函数

获取包含索引的表的名称。

语法

ULValue **UltraLite_IndexSchema_iface::GetTableName()**

IsColumnDescending 函数

确定列是否为降序排序。

语法

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

参数

- **column_name**　列名称。

返回值

如果列为降序排序，则返回 true。

注释

如果列名不存在，则设置 SQLE_COLUMN_NOT_FOUND。

IsForeignKey 函数

检查索引是否为外键。

语法

bool **UltraLite_IndexSchema_iface::IsForeignKey()**

返回值

- 如果索引是外键，则返回 true。
- 如果索引不是外键，则返回 false。

注释

外键中的列可以引用另一个表的非空唯一索引。

IsForeignKeyCheckOnCommit 函数

检查是在提交时还是在插入和更新时执行外键的参照完整性。

语法

bool **UltraLite_IndexSchema_iface::IsForeignKeyCheckOnCommit()**

返回值

- 如果此外键在提交时检查参照完整性，则返回 true。
- 如果此外键在插入时检查参照完整性，则返回 false。

IsForeignKeyNullable 函数

检查外键是否可为空。

语法

bool **UltraLite_IndexSchema_iface::IsForeignKeyNullable()**

返回值

- 如果索引是唯一外键约束，则返回 true。
- 如果外键不可为空，则返回 False。

IsPrimaryKey 函数

检查索引是否为主键。

语法

bool **UltraLite_IndexSchema_iface::IsPrimaryKey()**

返回值

- 如果索引是主键，则返回 true。
- 如果索引不是主键，则返回 false。

注释

主键中的列不可以为空值。

IsUniqueIndex 函数

检查索引是否唯一。

语法

bool **UltraLite_IndexSchema_iface::IsUniqueIndex()**

返回值

- 如果索引是唯一的，则返回 true。
- 如果索引不是唯一的，则返回 false。

IsUniqueKey 函数

检查索引是否为唯一键。

语法

bool **UltraLite_IndexSchema_iface::IsUniqueKey()**

返回值

- 如果索引是主键或唯一约束，则返回 true。
- 如果索引既不是主键也不是唯一约束，则返回 false。

注释

唯一键中的列不可以为空值。

UltraLite_PreparedStatement 类

使用占位符来准备语句，然后在执行语句后向占位符赋值。

语法

public **UltraLite_PreparedStatement**

基类

- “UltraLite_SQLObject_iface 类”一节第 206 页
- “UltraLite_PreparedStatement_iface 类”一节第 194 页

成员

UltraLite_PreparedStatement 的所有成员（包括所有继承成员）。

- “AddRef 函数”一节第 206 页
- “ExecuteQuery 函数”一节第 194 页
- “ExecuteStatement 函数”一节第 194 页
- “GetConnection 函数”一节第 206 页
- “GetIFace 函数”一节第 207 页
- “GetPlan 函数”一节第 195 页
- “GetPlan 函数”一节第 195 页
- “GetSchema 函数”一节第 195 页
- “GetStreamWriter 函数”一节第 196 页
- “HasResultSet 函数”一节第 196 页
- “Release 函数”一节第 207 页
- “SetParameter 函数”一节第 196 页
- “SetParameterNull 函数”一节第 197 页

UltraLite_PreparedStatement_iface 类

PreparedStatement 接口。

语法

public **UltraLite_PreparedStatement_iface**

派生类

- “UltraLite_PreparedStatement 类”一节第 193 页

成员

UltraLite_PreparedStatement_iface 的所有成员（包括所有继承成员）。

- “ExecuteQuery 函数”一节第 194 页
- “ExecuteStatement 函数”一节第 194 页
- “GetPlan 函数”一节第 195 页
- “GetPlan 函数”一节第 195 页
- “GetSchema 函数”一节第 195 页
- “GetStreamWriter 函数”一节第 196 页
- “HasResultSet 函数”一节第 196 页
- “SetParameter 函数”一节第 196 页
- “SetParameterNull 函数”一节第 197 页

ExecuteQuery 函数

执行 SQL SELECT 语句（查询的形式）。

语法

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

返回值

该查询的结果集（一组行的形式）。

ExecuteStatement 函数

执行一个不返回结果集的语句，如 SQL INSERT、DELETE 或 UPDATE 语句。

语法

ul_s_long **UltraLite_PreparedStatement_iface::ExecuteStatement()**

GetPlan 函数

获取对查询执行计划的基于文本的说明。

语法

```
size_t UltraLite_PreparedStatement_iface::GetPlan(
  char * buffer,
  size_t size
)
```

参数

- **buffer** 接收计划说明的缓冲区。
- **size** 缓冲区的大小（以 ASCII 字符为单位）。

返回值

描述 UltraLite 将用来执行查询的访问计划的字符串。

注释

此函数主要供开发期间使用。

GetPlan 函数

获取对查询执行计划的基于文本的说明（宽字符形式）。

语法

```
size_t UltraLite_PreparedStatement_iface::GetPlan(
  ul_wchar * buffer,
  size_t size
)
```

参数

- **buffer** 接收计划说明的缓冲区。
- **size** 缓冲区大小（以 ul_wchar 为单位）。

返回值

描述 UltraLite 将用来执行查询的访问计划的字符串。

注释

此函数主要供开发期间使用。

GetSchema 函数

获取结果集的模式。

语法

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

GetStreamWriter 函数

获取一个用于将字符串/二进制数据流式写入参数的流写入器。

语法

```
UltraLite_StreamWriter * UltraLite_PreparedStatement_iface::GetStreamWriter(
  ul_column_num parameter_id
)
```

参数

- **parameter_id** 列标识符，可以是从 1 开始的序号或列名。

HasResultSet 函数

确定此 SQL 语句是否有结果集。

语法

bool **UltraLite_PreparedStatement_iface::HasResultSet()**

返回值

- 如果在执行此语句时生成结果集，则返回 true。
- 如果未生成结果集，则返回 false。

SetParameter 函数

设置 SQL 语句的参数。

语法

```
void UltraLite_PreparedStatement_iface::SetParameter(
  ul_column_num parameter_id,
  ULValue const & value
)
```

参数

- **parameter_id** 从 1 开始的参数序号。
- **value** 要设置参数的值。

SetParameterNull 函数

将参数设置为空。

语法

void **UltraLite_PreparedStatement_iface::SetParameterNull(**
ul_column_num *parameter_id*
)

参数

- **parameter_id** 从 1 开始的参数序号。

UltraLite_ResultSet 类

表示 UltraLite 数据库中可编辑的结果集。

语法

public **UltraLite_ResultSet**

基类

- “UltraLite_SQLObject_iface 类”一节第 206 页
- “UltraLite_ResultSet_iface 类”一节第 199 页
- “UltraLite_Cursor_iface 类”一节第 172 页

成员

UltraLite_ResultSet 的所有成员（包括所有继承成员）。

- “AddRef 函数”一节第 206 页
- “AfterLast 函数”一节第 172 页
- “BeforeFirst 函数”一节第 173 页
- “Delete 函数”一节第 173 页
- “DeleteNamed 函数”一节第 199 页
- “First 函数”一节第 173 页
- “Get 函数”一节第 173 页
- “GetConnection 函数”一节第 206 页
- “GetIFace 函数”一节第 207 页
- “GetRowCount 函数”一节第 174 页
- “GetSchema 函数”一节第 199 页
- “GetState 函数”一节第 174 页
- “GetStreamReader 函数”一节第 174 页
- “GetStreamWriter 函数”一节第 175 页
- “GetSuspend 函数（不建议使用）”一节第 175 页
- “IsNull 函数”一节第 175 页
- “Last 函数”一节第 175 页
- “Next 函数”一节第 176 页
- “Previous 函数”一节第 176 页
- “Relative 函数”一节第 176 页
- “Release 函数”一节第 207 页
- “Set 函数”一节第 176 页
- “SetDefault 函数”一节第 177 页
- “SetNull 函数”一节第 177 页
- “SetSuspend 函数（不建议使用）”一节第 177 页
- “Update 函数”一节第 178 页
- “UpdateBegin 函数”一节第 178 页

注释

可编辑结果集允许您执行定位的更新和删除。

UltraLite_ResultSet_iface 类

ResultSet 接口。

语法

public **UltraLite_ResultSet_iface**

派生类

- “UltraLite_ResultSet 类”一节第 198 页

成员

UltraLite_ResultSet_iface 的所有成员（包括所有继承成员）。

- “DeleteNamed 函数”一节第 199 页
- “GetSchema 函数”一节第 199 页

DeleteNamed 函数

删除当前行并将其移动到下一个有效行。

语法

```
bool UltraLite_ResultSet_iface:DeleteNamed(
  const ULValue & table_name
)
```

参数

- **table_name**　表名或其相关名（当数据库具有共享同一表名的多个列时，这是必需的）。

GetSchema 函数

获取此结果集的模式。

语法

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

UltraLite_ResultSetSchema 类

检索有关结果集的模式信息。例如，列名、总列数、列小数位数、列大小和列 SQL 类型。

语法

public **UltraLite_ResultSetSchema**

基类

- “UltraLite_SQLObject_iface 类”一节第 206 页
- “UltraLite_RowSchema_iface 类”一节第 201 页

成员

UltraLite_ResultSetSchema 的所有成员（包括所有继承成员）。

- “AddRef 函数”一节第 206 页
- “GetBaseColumnName 函数”一节第 201 页
- “GetColumnCount 函数”一节第 202 页
- “GetColumnID 函数”一节第 202 页
- “GetColumnName 函数”一节第 202 页
- “GetColumnPrecision 函数”一节第 203 页
- “GetColumnScale 函数”一节第 204 页
- “GetColumnSize 函数”一节第 204 页
- “GetColumnSQLName 函数”一节第 203 页
- “GetColumnSQLType 函数”一节第 203 页
- “GetColumnType 函数”一节第 205 页
- “GetConnection 函数”一节第 206 页
- “GetIFace 函数”一节第 207 页
- “Release 函数”一节第 207 页

UltraLite_RowSchema_iface 类

RowSchema 接口。

语法

public **UltraLite_RowSchema_iface**

派生类

- “UltraLite_ResultSetSchema 类”一节第 200 页
- “UltraLite_TableSchema 类”一节第 221 页

成员

UltraLite_RowSchema_iface 的所有成员（包括所有继承成员）。

- “GetBaseColumnName 函数”一节第 201 页
- “GetColumnCount 函数”一节第 202 页
- “GetColumnID 函数”一节第 202 页
- “GetColumnName 函数”一节第 202 页
- “GetColumnPrecision 函数”一节第 203 页
- “GetColumnScale 函数”一节第 204 页
- “GetColumnSize 函数”一节第 204 页
- “GetColumnSQLName 函数”一节第 203 页
- “GetColumnSQLType 函数”一节第 203 页
- “GetColumnType 函数”一节第 205 页

GetBaseColumnName 函数

获取结果集中某列的组合基本名和列名（即使此列具有相关名或别名）。

语法

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

参数

- **column_id**　从 1 开始的序号。

返回值

- 组合的“ULValue 类”一节第 232 页对象。
- 如果该列不是表的一部分，则返回空名称。

注释

如果列名不存在，则设置 SQLE_COLUMN_NOT_FOUND。

GetColumnCount 函数

获取表中的列数。

语法

ul_column_num **UltraLite_RowSchema_iface::GetColumnCount()**

GetColumnID 函数

获取从 1 开始的列 ID。

语法

ul_column_num **UltraLite_RowSchema_iface::GetColumnID(**
  const ULValue & *column_name*
)

参数

- **column_name** 列名称。

返回值

如果列不存在，则返回 0。

注释

如果列名不存在，则设置 SQLE_COLUMN_NOT_FOUND。

GetColumnName 函数

在已知列的 ID（从 1 开始）的情况下获取其名称。

语法

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

参数

- **column_id** 从 1 开始的序号。

返回值

如果列不存在，则返回空的“ULValue 类”一节第 232 页对象。

注释

此名称将成为 SELECT 语句的别名或相关名。

如果列名不存在，则设置 SQLE_COLUMN_NOT_FOUND

GetColumnPrecision 函数

获取数字列的精度。

语法

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

参数

- **column_id** 从 1 开始的序号。

返回值

如果列不是数字类型，或者如果列不存在，则返回 0。

注释

如果列名不存在，则设置 SQLE_COLUMN_NOT_FOUND。

如果列类型不是数字列，则设置 SQLE_DATATYPE_NOT_ALLOWED。

GetColumnSQLName 函数

获取结果集中的列的 SQL 名称。

语法

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

参数

- **column_id** 从 1 开始的序号。

返回值

组合的“ULValue 类”一节第 232 页对象。

注释

如果列有别名，则使用该名称。否则，如果结果集中的列与表中的列相对应，则使用该列名。不然的话，组合名为空。

如果列名不存在，则设置 SQLE_COLUMN_NOT_FOUND。

GetColumnSQLType 函数

获取列的 SQL 类型。

语法

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

参数

- **column_id** 从 1 开始的序号。

返回值

如果列不存在，则返回 UL_SQLTYPE_BAD_INDEX。

注释

请参见 ulprotos.h 中的 ul_column_sql_type。

GetColumnScale 函数

获取数字列的小数位数。

语法

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

参数

- **column_id** 从 1 开始的序号。

返回值

如果列不是数字类型，或者如果列不存在，则返回 0。

注释

如果列名不存在，则设置 SQLE_COLUMN_NOT_FOUND。

如果列类型不是数字列，则设置 SQLE_DATATYPE_NOT_ALLOWED。

GetColumnSize 函数

获取列的大小。

语法

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


参数

- **column_id** 从 1 开始的序号。

返回值

如果列不存在，或列类型不具有可变长度，则返回 0。

注释

如果列名不存在，则设置 SQLE_COLUMN_NOT_FOUND。

如果列类型不是 UL_SQLTYPE_CHAR 或 UL_SQLTYPE_BINARY，则设置 SQLE_DATATYPE_NOT_ALLOWED。

GetColumnType 函数

获取列的类型。

语法

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

参数

- **column_id** 从 1 开始的序号。

返回值

如果列不存在，则返回 UL_TYPE_BAD_INDEX。

注释

请参见 ulprotos.h 中的 ul_column_storage_type 枚举。

UltraLite_SQLObject_iface 类

SQLObject 接口。

语法

public **UltraLite_SQLObject_iface**

派生类

- “UltraLite_Connection 类”一节第 145 页
- “UltraLite_DatabaseSchema 类”一节第 183 页
- “UltraLite_IndexSchema 类”一节第 187 页
- “UltraLite_PreparedStatement 类”一节第 193 页
- “UltraLite_ResultSet 类”一节第 198 页
- “UltraLite_ResultSetSchema 类”一节第 200 页
- “UltraLite_StreamReader 类”一节第 208 页
- “UltraLite_StreamWriter 类”一节第 212 页
- “UltraLite_Table 类”一节第 213 页
- “UltraLite_TableSchema 类”一节第 221 页

成员

UltraLite_SQLObject_iface 的所有成员（包括所有继承成员）。

- “AddRef 函数”一节第 206 页
- “GetConnection 函数”一节第 206 页
- “GetIFace 函数”一节第 207 页
- “Release 函数”一节第 207 页

AddRef 函数

增加对象的内部引用计数。

语法

ul_ret_void **UltraLite_SQLObject_iface::AddRef()**

注释

要释放对象，必须使此函数的每个调用与“Release 函数”一节第 207 页调用相匹配。

GetConnection 函数

获取连接对象。

语法

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

返回值

与此对象关联的连接。

GetIFace 函数

留作将来使用。

语法

```
ul_void * UltraLite_SQLObject_iface::GetIFace(
  ul_iface_id iface
)
```

参数

- **iface** 留作将来使用。

Release 函数

释放对一个对象的引用。

语法

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

注释

当所有的引用都被删除后，对象立即被释放。必须至少调用此函数一次。如果使用“AddRef 函数”一节第 206 页，则还需要来自每个“AddRef 函数”一节第 206 页的匹配调用。

UltraLite_StreamReader 类

表示一个 UltraLite StreamReader。

语法

public **UltraLite_StreamReader**

基类

- “UltraLite_SQLObject_iface 类”一节第 206 页
- “UltraLite_StreamReader_iface 类”一节第 209 页

成员

UltraLite_StreamReader 的所有成员（包括所有继承成员）。

- “AddRef 函数”一节第 206 页
- “GetByteChunk 函数”一节第 209 页
- “GetConnection 函数”一节第 206 页
- “GetIFace 函数”一节第 207 页
- “GetLength 函数”一节第 210 页
- “GetStringChunk 函数”一节第 210 页
- “GetStringChunk 函数”一节第 210 页
- “Release 函数”一节第 207 页
- “SetReadPosition 函数”一节第 211 页

UltraLite_StreamReader_iface 类

StreamReader 接口。

语法

public **UltraLite_StreamReader_iface**

派生类

- “UltraLite_StreamReader 类”一节第 208 页

成员

UltraLite_StreamReader_iface 的所有成员（包括所有继承成员）。

- “GetByteChunk 函数”一节第 209 页
- “GetLength 函数”一节第 210 页
- “GetStringChunk 函数”一节第 210 页
- “GetStringChunk 函数”一节第 210 页
- “SetReadPosition 函数”一节第 211 页

注释

此接口支持读取/检索 VARCHAR 和 BINARY 列。

GetByteChunk 函数

通过将 buffer_len 字节复制到缓冲区数据中，从当前的 StreamReader 偏移获取字节块。

语法

```
bool UltraLite_StreamReader_iface::GetByteChunk(
  ul_byte * data,
  size_t buffer_len,
  size_t * len_retn,
  bool * morebytes
)
```

参数

- **data**　指向某个字节数组的指针。
- **buffer_len**　缓冲区或数组的长度。buffer_len 必须大于或等于 0。
- **len_retn**　输出参数。返回的长度。
- **morebytes**　输出参数。如果要读取更多字节，则为 True。

注释

除非使用“SetReadPosition 函数”一节第 211 页，否则将从上次读取停止处读取字节。

GetLength 函数

获取字符串/二进制值的长度。

语法

```
size_t UltraLite_StreamReader_iface::GetLength(
  bool fetch_as_chars
)
```

参数

- **fetch_as_chars** 对于字节，长度为 false；对于字符，长度为 true。

返回值

- 二进制值的字节数（对于二进制，将忽略 fetch_as_chars）。
- 字符串值的字符数或字节数。

GetStringChunk 函数

通过将 buffer_len 宽字符复制到缓冲区字符串中，从当前的 StreamReader 偏移获取字符串块。

语法

```
bool UltraLite_StreamReader_iface::GetStringChunk(
  ul_wchar * str,
  size_t buffer_len,
  size_t * len_retn,
  bool * morebytes
)
```

参数

- **str** 指向某个宽字符数组的指针。
- **buffer_len** 缓冲区的长度。
- **len_retn** 输出参数。返回的长度。
- **morebytes** 输出参数。如果要读取更多字符，则为 true。

注释

除非使用“SetReadPosition 函数”一节第 211 页，否则将从上次读取停止处读取字符。

GetStringChunk 函数

通过将 buffer_len 字节复制到缓冲区字符串中，从当前的 StreamReader 偏移获取字符串块。

语法

```
bool UltraLite_StreamReader_iface::GetStringChunk(
  char * str,
  size_t buffer_len,
  size_t * len_retn,
  bool * morebytes
)
```

参数

- **str** 指向某个字符数组的指针。
- **buffer_len** 缓冲区或数组的长度。buffer_len 必须大于或等于 0。
- **len_retn** 输出参数。返回的长度。
- **morebytes** 输出参数。如果要读取更多字符，则为 true。

注释

除非使用“SetReadPosition 函数”一节第 211 页，否则将从上次读取停止处读取字符。

SetReadPosition 函数

在数据中设置偏移以便进行下一次读取。

语法

```
bool UltraLite_StreamReader_iface::SetReadPosition(
  size_t offset,
  bool offset_in_chars
)
```

参数

- **offset** 偏移。
- **offset_in_chars** 如果偏移以字符为单位，则为 true。如果偏移以字节为单位，则为 False。

UltraLite_StreamWriter 类

表示一个 UltraLite StreamWriter。

语法

public **UltraLite_StreamWriter**

基类

- “UltraLite_SQLObject_iface 类”一节第 206 页

成员

UltraLite_StreamWriter 的所有成员（包括所有继承成员）。

- “AddRef 函数”一节第 206 页
- “GetConnection 函数”一节第 206 页
- “GetIFace 函数”一节第 207 页
- “Release 函数”一节第 207 页

UltraLite_Table 类

表示 UltraLite 数据库中的表。

语法

public **UltraLite_Table**

基类

- “UltraLite_SQLObject_iface 类”一节第 206 页
- “UltraLite_Table_iface 类”一节第 215 页
- “UltraLite_Cursor_iface 类”一节第 172 页

成员

UltraLite_Table 的所有成员（包括所有继承成员）。

- “AddRef 函数”一节第 206 页
- “AfterLast 函数”一节第 172 页
- “BeforeFirst 函数”一节第 173 页
- “Delete 函数”一节第 173 页
- “DeleteAllRows 函数”一节第 215 页
- “Find 函数”一节第 216 页
- “FindBegin 函数”一节第 216 页
- “FindFirst 函数”一节第 216 页
- “FindLast 函数”一节第 217 页
- “FindNext 函数”一节第 217 页
- “FindPrevious 函数”一节第 217 页
- “First 函数”一节第 173 页
- “Get 函数”一节第 173 页
- “GetConnection 函数”一节第 206 页
- “GetIFace 函数”一节第 207 页
- “GetRowCount 函数”一节第 174 页
- “GetSchema 函数”一节第 218 页
- “GetState 函数”一节第 174 页
- “GetStreamReader 函数”一节第 174 页
- “GetStreamWriter 函数”一节第 175 页
- “GetSuspend 函数（不建议使用）”一节第 175 页
- “Insert 函数”一节第 218 页
- “InsertBegin 函数”一节第 218 页
- “IsNull 函数”一节第 175 页
- “Last 函数”一节第 175 页
- “Lookup 函数”一节第 218 页
- “LookupBackward 函数”一节第 219 页
- “LookupBegin 函数”一节第 219 页
- “LookupForward 函数”一节第 219 页
- “Next 函数”一节第 176 页
- “Previous 函数”一节第 176 页
- “Relative 函数”一节第 176 页
- “Release 函数”一节第 207 页
- “Set 函数”一节第 176 页
- “SetDefault 函数”一节第 177 页
- “SetNull 函数”一节第 177 页
- “SetSuspend 函数（不建议使用）”一节第 177 页
- “TruncateTable 函数”一节第 220 页
- “Update 函数”一节第 178 页
- “UpdateBegin 函数”一节第 178 页

UltraLite_Table_iface 类

表示 table 接口。

语法

public **UltraLite_Table_iface**

派生类

- “UltraLite_Table 类”一节第 213 页

成员

UltraLite_Table_iface 的所有成员（包括所有继承成员）。

- “DeleteAllRows 函数”一节第 215 页
- “Find 函数”一节第 216 页
- “FindBegin 函数”一节第 216 页
- “FindFirst 函数”一节第 216 页
- “FindLast 函数”一节第 217 页
- “FindNext 函数”一节第 217 页
- “FindPrevious 函数”一节第 217 页
- “GetSchema 函数”一节第 218 页
- “Insert 函数”一节第 218 页
- “InsertBegin 函数”一节第 218 页
- “Lookup 函数”一节第 218 页
- “LookupBackward 函数”一节第 219 页
- “LookupBegin 函数”一节第 219 页
- “LookupForward 函数”一节第 219 页
- “TruncateTable 函数”一节第 220 页

DeleteAllRows 函数

从表中删除所有行。

语法

bool **UltraLite_Table_iface::DeleteAllRows()**

返回值

- 如果成功，则返回 true。
- 如果失败，则返回 false。例如，表未打开或发生 SQL 错误等等。

注释

在某些应用程序中，您可能需要在将一组新数据下载到表中之前删除表中的所有行。如果连接上设置了 stop sync 属性，则不同步已删除的行。

来自其它连接的任何未提交的插入不会被删除。另外，如果其它连接在其调用“DeleteAllRows 函数”一节第 215 页后执行回退，则来自其它连接的任何未提交的删除也不会被删除。

如果此表未使用索引打开，那么此表将视为只读并且无法删除数据。

Find 函数

此函数等效于“FindFirst 函数”一节第 216 页。根据当前索引，向前扫描整个表以执行精确匹配查寻。

语法

```
bool UltraLite_Table_iface::Find(
  ul_column_num ncols
)
```

参数

- **ncols** 对于复合索引，是指在查找中使用的列数。

FindBegin 函数

通过进入查找模式来准备在表上执行新的查找。

语法

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

注释

只可以设置用来打开表的索引中的列。如果此表未使用索引打开，将无法调用此方法。

FindFirst 函数

根据当前索引，向前扫描整个表以执行精确匹配查寻。

语法

```
bool UltraLite_Table_iface::FindFirst(
  ul_column_num ncols
)
```

参数

- **ncols** 对于复合索引，是指在查找中使用的列数。

注释

要指定要搜索的值，请为索引中的每一列设置列值。游标位于第一个与索引值完全匹配的行上。如果没有与索引值匹配的行，则游标位置为 AfterLast()，并且此函数返回 false。

FindLast 函数

根据当前索引，向后扫描整个表以执行精确匹配查寻。

语法

```
bool UltraLite_Table_iface::FindLast(
  ul_column_num ncols
)
```

参数

- **ncols** 对于复合索引，是指在查找中使用的列数。

注释

要指定要搜索的值，请为索引中的每一列设置列值。游标位于第一个与索引值完全匹配的行的左边。如果没有与索引值匹配的行，则游标位置为 BeforeFirst()，并且此函数返回 false。

FindNext 函数

获取与索引完全匹配的下一行。

语法

```
bool UltraLite_Table_iface::FindNext(
  ul_column_num ncols
)
```

参数

- **ncols** 对于复合索引，是指在查找中使用的列数。

返回值

如果没有其它行与该索引匹配，则返回 false。

FindPrevious 函数

获取与索引完全匹配的上一行。

语法

```
bool UltraLite_Table_iface::FindPrevious(
  ul_column_num ncols
)
```

参数

- **ncols** 对于复合索引，是指在查找中使用的列数。

返回值

如果没有其它行与该索引匹配，则返回 false。在这种情况下，游标定位在第一行之前。

GetSchema 函数

获取此表的模式对象。

语法

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

Insert 函数

在表中插入一个新行。

语法

bool **UltraLite_Table_iface::Insert()**

注释

表必须处于插入模式，此操作才能成功。使用“InsertBegin 函数”一节第 218 页可切换到插入模式。

InsertBegin 函数

选择用于设置列的插入模式。

语法

bool **UltraLite_Table_iface::InsertBegin()**

注释

在此模式下可以修改所有列。如果此表未使用索引打开，那么数据将视为只读并且无法插入行。

Lookup 函数

此函数等效于 LookupForward。根据当前索引，向前扫描整个表以执行查寻。

语法

bool **UltraLite_Table_iface::Lookup(**
  ul_column_num *ncols*
)

参数

- **ncols** 对于复合索引，是指在查找中使用的列数。

注释

如果游标的最终位置是 AfterLast()，则返回值为 false。

LookupBackward 函数

根据当前索引，向后扫描整个表以执行查寻。

语法

```
bool UltraLite_Table_iface::LookupBackward(
  ul_column_num ncols
)
```

参数

- **ncols** 对于复合索引，是指在查找中使用的列数。

返回值

如果游标的最终位置是 BeforeFirst()，则返回值为 false。

注释

要指定要搜索的值，请为索引中的每一列设置列值。游标位于最后一个与索引值匹配或小于该索引值的行上。对于复合索引，ncols 指定在查找中使用的列数。

LookupBegin 函数

通过进入查寻模式来准备在表上执行新的查找。

语法

```
bool UltraLite_Table_iface::LookupBegin()
```

注释

只可以设置用来打开表的索引中的列。如果此表未使用索引打开，将无法调用此方法。

LookupForward 函数

根据当前索引，向前扫描整个表以执行查寻。

语法

bool **UltraLite_Table_iface::LookupForward(**
ul_column_num *ncols*
)

参数

- **ncols** 对于复合索引，是指在查找中使用的列数。

返回值

如果游标的最终位置是 AfterLast()，则返回值为 false。

注释

要指定要搜索的值，请为索引中的每一列设置列值。游标位于最后一个与索引值匹配或小于该索引值的行上。对于复合索引，ncols 指定在查找中使用的列数。

TruncateTable 函数

截断该表并暂时激活 STOP SYNCHRONIZATION DELETE。

语法

bool **UltraLite_Table_iface::TruncateTable()**

注释

如果此表未使用索引打开，那么此表将视为只读并且无法删除数据。

UltraLite_TableSchema 类

表示表模式。

语法

public **UltraLite_TableSchema**

基类

- “UltraLite_SQLObject_iface 类”一节第 206 页
- “UltraLite_TableSchema_iface 类”一节第 223 页
- “UltraLite_RowSchema_iface 类”一节第 201 页

成员

UltraLite_TableSchema 的所有成员（包括所有继承成员）。

- “AddRef 函数”一节第 206 页
- “GetBaseColumnName 函数”一节第 201 页
- “GetColumnCount 函数”一节第 202 页
- “GetColumnDefault 函数”一节第 223 页
- “GetColumnID 函数”一节第 202 页
- “GetColumnName 函数”一节第 202 页
- “GetColumnPrecision 函数”一节第 203 页
- “GetColumnScale 函数”一节第 204 页
- “GetColumnSize 函数”一节第 204 页
- “GetColumnSQLName 函数”一节第 203 页
- “GetColumnSQLType 函数”一节第 203 页
- “GetColumnType 函数”一节第 205 页
- “GetConnection 函数”一节第 206 页
- “GetGlobalAutoincPartitionSize 函数”一节第 224 页
- “GetID 函数”一节第 224 页
- “GetIFace 函数”一节第 207 页
- “GetIndexCount 函数”一节第 224 页
- “GetIndexName 函数”一节第 225 页
- “GetIndexSchema 函数”一节第 225 页
- “GetName 函数”一节第 226 页
- “GetOptimalIndex 函数”一节第 226 页
- “GetPrimaryKey 函数”一节第 226 页
- “GetPublicationPredicate 函数”一节第 226 页
- “GetUploadUnchangedRows 函数”一节第 227 页
- “InPublication 函数”一节第 227 页
- “IsColumnAutoinc 函数”一节第 228 页
- “IsColumnCurrentDate 函数”一节第 228 页
- “IsColumnCurrentTime 函数”一节第 228 页
- “IsColumnCurrentTimestamp 函数”一节第 229 页
- “IsColumnGlobalAutoinc 函数”一节第 229 页
- “IsColumnInIndex 函数”一节第 230 页
- “IsColumnNewUUID 函数”一节第 230 页
- “IsColumnNullable 函数”一节第 231 页
- “IsNeverSynchronized 函数”一节第 231 页
- “Release 函数”一节第 207 页

UltraLite_TableSchema_iface 类

TableSchema 接口。

语法

public **UltraLite_TableSchema_iface**

派生类

- “UltraLite_TableSchema 类”一节第 221 页

成员

UltraLite_TableSchema_iface 的所有成员（包括所有继承成员）。

- “GetColumnDefault 函数”一节第 223 页
- “GetGlobalAutoincPartitionSize 函数”一节第 224 页
- “GetID 函数”一节第 224 页
- “GetIndexCount 函数”一节第 224 页
- “GetIndexName 函数”一节第 225 页
- “GetIndexSchema 函数”一节第 225 页
- “GetName 函数”一节第 226 页
- “GetOptimalIndex 函数”一节第 226 页
- “GetPrimaryKey 函数”一节第 226 页
- “GetPublicationPredicate 函数”一节第 226 页
- “GetUploadUnchangedRows 函数”一节第 227 页
- “InPublication 函数”一节第 227 页
- “IsColumnAutoinc 函数”一节第 228 页
- “IsColumnCurrentDate 函数”一节第 228 页
- “IsColumnCurrentTime 函数”一节第 228 页
- “IsColumnCurrentTimestamp 函数”一节第 229 页
- “IsColumnGlobalAutoinc 函数”一节第 229 页
- “IsColumnInIndex 函数”一节第 230 页
- “IsColumnNewUUID 函数”一节第 230 页
- “IsColumnNullable 函数”一节第 231 页
- “IsNeverSynchronized 函数”一节第 231 页

GetColumnDefault 函数

获取列的缺省值（如果存在）。

语法

```
ULValue UltraLite_TableSchema_iface::GetColumnDefault(
  const ULValue & column_id
)
```

参数

- **column_id** 列的 ID 号。该值必须是从 1 开始的序号。表中第一列的 ID 值为 1。

返回值

- 包含的缺省值为字符串形式。
- 如果列不包含缺省值，则返回结果为空。

注释

如果列名不存在，则设置 SQLE_COLUMN_NOT_FOUND。

GetGlobalAutoincPartitionSize 函数

获取分区大小。

语法

```
bool UltraLite_TableSchema_iface::GetGlobalAutoincPartitionSize(
  const ULValue & column_id,
  ul_u_big * size
)
```

参数

- **column_id** 列的 ID 号。该值必须是从 1 开始的序号。表中第一列的 ID 值为 1。
- **size** 输出参数。列的分区大小。给定表中的所有全局自动增量列都共享同一全局自动增量分区。

返回值

全局自动增量列的分区大小。

GetID 函数

获取表 ID。

语法

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

GetIndexCount 函数

获取表中的索引数。

语法

```
ul_index_num UltraLite_TableSchema_iface::GetIndexCount()
```


返回值

表中的索引数。

注释

索引 ID 和索引计数在模式升级过程中可能发生变化。为了正确地标识索引，请按名称访问它，或者在模式升级后刷新任何高速缓存的 ID 和计数。

GetIndexName 函数

在已知索引的从 1 开始的 ID 的情况下获取索引名称。

语法

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

参数

- **index_id** 从 1 开始的序号。

返回值

如果索引不存在，则返回的“ULValue 类”一节第 232 页对象为空。

注释

索引 ID 和索引计数在模式升级过程中可能发生变化。为了正确地标识索引，请按名称访问它，或者在模式升级后刷新任何高速缓存的 ID 和计数。

GetIndexSchema 函数

获取具有给定名称或 ID 的 IndexSchema 对象。

语法

```
UltraLite_IndexSchema * UltraLite_TableSchema_iface::GetIndexSchema(
  const ULValue & index_id
)
```

参数

- **index_id** 用于标识索引的名称或 ID 号。

返回值

如果索引不存在，则返回 UL_NULL。

GetName 函数

获取表的名称。

语法

ULValue **UltraLite_TableSchema_iface::GetName()**

返回值

字符串形式的表名。

GetOptimalIndex 函数

确定用于搜索列值的最佳索引。

语法

ULValue **UltraLite_TableSchema_iface::GetOptimalIndex(**
  const ULValue & *column_id*
)

参数

- **column_id** 列的 ID 号。该值必须是从 1 开始的序号。表中第一列的 ID 值为 1。

返回值

索引的名称。

GetPrimaryKey 函数

获取表的主键。

语法

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

GetPublicationPredicate 函数

获取字符串形式的发布谓语。

语法

ULValue **UltraLite_TableSchema_iface::GetPublicationPredicate(**
  const ULValue & *publication_name*
)

参数

- **publication_name** 发布的名称。

返回值

指定发布的发布谓语字符串。

注释

如果发布不存在，则设置 SQLE_PUBLICATION_NOT_FOUND。

GetUploadUnchangedRows 函数

检查数据库是否已被配置为上载未更改的行。

语法

bool **UltraLite_TableSchema_iface::GetUploadUnchangedRows()**

返回值

- 如果此表被标记为在同步过程中始终上载所有行，则返回 true。
- 如果此表被标记为仅上载已更改的行，则返回 false。

注释

设置为上载未更改和已更改的行的表有时被称为 allsync 表。

InPublication 函数

检查此表是否包含在指定发布中。

语法

```
bool UltraLite_TableSchema_iface::InPublication(
  const ULValue & publication_name
)
```

参数

- **publication_name** 发布的名称。

返回值

- 如果该表包含在发布中，则返回 true。
- 如果该表未包含在发布中，则返回 false。

注释

如果发布不存在，则设置 SQLE_PUBLICATION_NOT_FOUND。

IsColumnAutoinc 函数

检查指定列的缺省值是否设置为自动增量。

语法

```
bool UltraLite_TableSchema_iface::IsColumnAutoinc(
  const ULValue & column_id
)
```

参数

- **column_id** 列的 ID 号。该值必须是从 1 开始的序号。表中第一列的 ID 值为 1。

返回值

- 如果将列的缺省值设为自动增量，则返回 true。
- 如果列不自动增量，则返回 false。

注释

如果列名不存在，则设置 SQLE_COLUMN_NOT_FOUND。

IsColumnCurrentDate 函数

检查指定列的缺省值是否设置为当前日期。

语法

```
bool UltraLite_TableSchema_iface::IsColumnCurrentDate(
  const ULValue & column_id
)
```

参数

- **column_id** 列的 ID 号。该值必须是从 1 开始的序号。表中第一列的 ID 值为 1。

返回值

- 如果列具有当前日期缺省值，则返回 true。
- 如果列未缺省为当前日期，则返回 false。

IsColumnCurrentTime 函数

检查指定列的缺省值是否设置为当前时间。

语法

```
bool UltraLite_TableSchema_iface::IsColumnCurrentTime(
  const ULValue & column_id
)
```


参数

- **column_id** 列的 ID 号。该值必须是从 1 开始的序号。表中第一列的 ID 值为 1。

返回值

- 如果列具有当前时间缺省值，则返回 true。
- 否则，返回 false。

IsColumnCurrentTimestamp 函数

检查指定列的缺省值是否设置为当前时间戳。

语法

```
bool UltraLite_TableSchema_iface::IsColumnCurrentTimestamp(
  const ULValue & column_id
)
```

参数

- **column_id** 列的 ID 号。该值必须是从 1 开始的序号。表中第一列的 ID 值为 1。

返回值

- 如果列具有当前时间戳缺省值，则返回 true。
- 否则，返回 false。

IsColumnGlobalAutoinc 函数

检查指定列的缺省值是否设置为自动增量。

语法

```
bool UltraLite_TableSchema_iface::IsColumnGlobalAutoinc(
  const ULValue & column_id
)
```

参数

- **column_id** 列的 ID 号。该值必须是从 1 开始的序号。表中第一列的 ID 值为 1。

返回值

- 如果列为自动增量，则返回 true。
- 如果不是自动增量，则返回 False。

注释

如果列名不存在，则设置 SQLE_COLUMN_NOT_FOUND。

IsColumnInIndex 函数

检查此表是否包含在指定索引中。

语法

```
bool UltraLite_TableSchema_iface::IsColumnInIndex(
  const ULValue & column_id,
  const ULValue & index_id
)
```

参数

- **column_id** 用于标识列的从 1 开始的序号。通过调用“GetColumnCount 函数”一节第 202 页可以获取 column_id。
- **index_id** 用于标识索引的从 1 开始的序号。通过调用“GetIndexCount 函数”一节第 224 页可以获取表中的索引的数目。

返回值

- 如果该列包含在索引中，则返回 true。
- 如果该列未包含在索引中，则返回 false。

注释

如果列名不存在，则设置 SQLE_COLUMN_NOT_FOUND。

如果索引不存在，则设置 SQLE_INDEX_NOT_FOUND。

IsColumnNewUUID 函数

检查指定列的缺省值是否设置为新的 UUID。

语法

```
bool UltraLite_TableSchema_iface::IsColumnNewUUID(
  const ULValue & column_id
)
```

参数

- **column_id** 列的 ID 号。该值必须是从 1 开始的序号。表中第一列的 ID 值为 1。

返回值

- 如果列具有新的 UUID 缺省值，则返回 true。
- 如果列未缺省为新的 UUID，则返回 false。

IsColumnNullable 函数

检查指定列是否可为空。

语法

```
bool UltraLite_TableSchema_iface::IsColumnNullable(
  const ULValue & column_id
)
```

参数

- **column_id** 列的 ID 号。该值必须是从 1 开始的序号。表中第一列的 ID 值为 1。指定列是索引中的第一列，但索引可以有多列。

返回值

- 如果列可为空，则返回 true。
- 如果不能为空，则返回 false。

注释

如果列名不存在，则设置 SQLE_COLUMN_NOT_FOUND。

IsNeverSynchronized 函数

检查此表是否被标记为永不同步。

语法

```
bool UltraLite_TableSchema_iface::IsNeverSynchronized()
```

返回值

- 如果从同步中省略了表，则返回 true。被标记为永不同步的表永远不同步，即使它们包含在发布中。这些表有时称为 nosync 表。
- 如果该表包含为可同步表，则返回 false。

ULValue 类

语法

public **ULValue**

注释

“ULValue 类”一节第 232 页类。

“ULValue 类”一节第 232 页类是存储在 UltraLite 游标中的数据类型的包装。此类允许您存储数据而不必考虑数据类型，并可用于将值传入或传出 UltraLite C++ 组件。

“ULValue 类”一节第 232 页包含许多构造函数和类型转换运算符，因此在大多数情况下可以无缝地使用“ULValue 类”一节第 232 页，而不必显式实例化“ULValue 类”一节第 232 页。

可构造对象或通过任何基本 C++ 数据类型对其赋值。还可将其转换为基本 C++ 数据类型。

```
 x( 5 );        ULValue// Example of ULValue's constructor
 y = 5;         ULValue// Example of ULValue's assignment operator
int z = y;              // Example of ULValue's cast operator
```

此示例同样适用于字符串：

```
 x( UL_TEXT( ULValue"hello" ) );
 y = UL_TEXT( ULValue"hello" );
y.( buffer, BUFFER_LEN );    GetString// NOTE, there is no cast operator
```

不必显式构造“ULValue 类”一节第 232 页对象，因为编译器经常会自动执行此操作。例如，要读取列中的值，可使用以下内容：

```
int x = table->Get( UL_TEXT( "my_column" ) );
```

table->Get() 调用返回一个“ULValue 类”一节第 232 页对象。C++ 自动调用类型转换运算符以将其转换为整数。类似地，table->Get() 调用将“ULValue 类”一节第 232 页参数作为列标识符。这样便确定了要读取的列。C++ 自动将 "my_column" 字符串转换为“ULValue 类”一节第 232 页对象。

成员

ULValue 的所有成员（包括所有继承成员）。

- “GetBinary 函数”一节第 234 页
- “GetBinary 函数”一节第 234 页
- “GetBinaryLength 函数”一节第 235 页
- “GetCombinedStringItem 函数”一节第 235 页
- “GetCombinedStringItem 函数”一节第 235 页
- “GetString 函数”一节第 236 页
- “GetString 函数”一节第 236 页
- “GetStringLength 函数”一节第 236 页
- “InDatabase 函数”一节第 237 页
- “IsNull 函数”一节第 237 页
- “bool 运算符”一节第 246 页
- “DECL_DATETIME 运算符”一节第 245 页
- “double 运算符”一节第 246 页
- “float 运算符”一节第 246 页
- “GUID 运算符”一节第 245 页
- “int 运算符”一节第 246 页
- “long 运算符”一节第 246 页
- “short 运算符”一节第 247 页
- “ul_s_big 运算符”一节第 247 页
- “ul_u_big 运算符”一节第 247 页
- “unsigned char 运算符”一节第 247 页
- “unsigned int 运算符”一节第 247 页
- “unsigned long 运算符”一节第 247 页
- “unsigned short 运算符”一节第 248 页
- “operator= 函数”一节第 248 页
- “SetBinary 函数”一节第 238 页
- “SetString 函数”一节第 238 页
- “SetString 函数”一节第 238 页
- “StringCompare 函数”一节第 239 页
- “ULValue 函数”一节第 239 页
- “ULValue 函数”一节第 239 页
- “ULValue 函数”一节第 240 页
- “ULValue 函数”一节第 240 页
- “ULValue 函数”一节第 240 页
- “ULValue 函数”一节第 241 页
- “ULValue 函数”一节第 241 页
- “ULValue 函数”一节第 241 页
- “ULValue 函数”一节第 241 页
- “ULValue 函数”一节第 242 页
- “ULValue 函数”一节第 242 页
- “ULValue 函数”一节第 242 页
- “ULValue 函数”一节第 243 页
- “ULValue 函数”一节第 243 页
- “ULValue 函数”一节第 243 页

- “ULValue 函数”一节第 243 页
- “ULValue 函数”一节第 244 页
- “ULValue 函数”一节第 244 页
- “ULValue 函数”一节第 244 页
- “ULValue 函数”一节第 245 页
- “ULValue 函数”一节第 245 页
- “~ULValue 函数”一节第 248 页

GetBinary 函数

将当前值检索到二进制缓冲区中，并根据需要进行类型转换。

语法

```
void ULValue::GetBinary(
  p_ul_binary bin,
  size_t len
)
```

参数

- **bin** 要接收字节的二进制结构。
- **len** 缓冲区的长度。

注释

如果缓冲区太小，则值将被截断。最多有 len 个字符被复制到给定缓冲区中。

GetBinary 函数

将当前值检索到二进制缓冲区中，并根据需要进行类型转换。如果缓冲区太小，则该值将被截断。

语法

```
void ULValue::GetBinary(
  ul_byte * dst,
  size_t len,
  size_t * retr_len
)
```

参数

- **dst** 要接收字节的缓冲区。
- **len** 缓冲区的长度。
- **retr_len** 输出参数。实际返回的字节数。

注释

最多有 len 个字节被复制到给定缓冲区中。实际复制的字节数在 retr_len 中返回。

GetBinaryLength 函数

获取 Binary 值的长度。

语法

size_t **ULValue::GetBinaryLength()**

返回值

保存由“GetBinary 函数”一节第 234 页返回的二进制值所需的字节数。

GetCombinedStringItem 函数

将部分组合名检索到字符串缓冲区中，并根据需要进行类型转换。

语法

```
void ULValue::GetCombinedStringItem(
  ul_u_short selector,
  char * dst,
  size_t len
)
```

参数

- **selector** 选定的内部值。
- **dst** 要接收字符串值的缓冲区。
- **len** dst 的长度（以字节为单位）。

注释

如果值未组合，则复制空字符串。输出字符串始终以空值终止。如果缓冲区太小，则值将被截断。最多有 len 个字符被复制到给定的缓冲区，包括空终止符。

GetCombinedStringItem 函数

获取组合字符串值的选定部分。

语法

```
void ULValue::GetCombinedStringItem(
  ul_u_short selector,
  ul_wchar * dst,
  size_t len
)
```

参数

- **selector** 选定的内部值。

- **dst** 要接收字符串值的缓冲区。
- **len** dst 的长度（以宽字符为单位）。

GetString 函数

将当前值检索到字符串缓冲区中，并根据需要进行类型转换。

语法

```
void ULValue::GetString(
  char * dst,
  size_t len
)
```

参数

- **dst** 要接收字符串值的缓冲区。
- **len** dst 的长度（以字节为单位）。

注释

输出字符串始终以空值终止。如果缓冲区太小，则值将被截断。最多有 len 个字符被复制到给定的缓冲区，包括空终止符。

GetString 函数

将当前值检索到字符串缓冲区中，并根据需要进行类型转换。

语法

```
void ULValue::GetString(
  ul_wchar * dst,
  size_t len
)
```

参数

- **dst** 要接收字符串值的缓冲区。
- **len** dst 的长度（以宽字符为单位）。

GetStringLength 函数

获取字符串的长度。

语法

size_t **ULValue::GetStringLength(**
  bool *fetch_as_chars*
)

参数

- **fetch_as_chars** 对于字节，长度为 false；对于字符，长度为 true。

返回值

保存由“GetString 函数”一节第 236 页方法之一返回的字符串所需的字节数或字符数（不包括空终止符）。

示例

预期的用法如下：

```
len = v.GetStringLength();
dst = new char[ len + 1 ];
( dst, len + 1 ); GetString
```

对于宽字符应用程序，用法为：

```
len = v.GetStringLength( true );
dst = new ul_wchar[ len + 1 ];
( dst, len + 1 ); GetString
```

InDatabase 函数

检查值是否在数据库中。

语法

bool **ULValue::InDatabase()**

返回值

- 如果此对象引用游标字段，则返回 true。
- 否则，返回 false。

IsNull 函数

检查“ULValue 类”一节第 232 页对象是否为空。

语法

bool **ULValue::IsNull()**

返回值

- 如果此对象是一个空“ULValue 类”一节第 232 页对象，或者如果此对象引用了一个设置为 NULL 的游标字段，则返回 true。
- 否则，返回 false。

SetBinary 函数

设置该值以引用所提供的二进制缓冲区。

语法

```
void ULValue::SetBinary(
  ul_byte * src,
  size_t len
)
```

参数

- **src** 字节缓冲区。
- **len** 缓冲区的长度。

注释

在使用该值之前，不会从所提供的缓冲区中复制字节。

SetString 函数

将“ULValue 类”一节第 232 页转换为字符串。

语法

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

参数

- **val** 指向此“ULValue 类”一节第 232 页的以空值终止的字符串表示形式的指针。
- **len** 字符串的长度。

SetString 函数

将“ULValue 类”一节第 232 页转换为 UNICODE 字符串。

语法

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

参数

- **val** 指向此“ULValue 类”一节第 232 页的以空值终止的 unicode 字符串表示形式的指针。
- **len** 字符串的长度。

StringCompare 函数

比较“ULValue 类”一节第 232 页对象的字符串或字符串表示形式。

语法

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

参数

- **value** 比较字符串。

返回值

- 如果两个字符串相等，则返回 0。
- 如果当前值小于 value，则返回 -1。
- 如果当前值大于 value，则返回 1。
- 如果任一“ULValue 类”一节第 232 页对象的 sqlca 未设置，则返回 -3。
- 如果任一“ULValue 类”一节第 232 页对象的字符串表示为 UL_NULL，则返回 -2。

ULValue 函数

构造一个“ULValue 类”一节第 232 页。

语法

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

ULValue 函数

通过复制现有值构造“ULValue 类”一节第 232 页。

语法

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

参数

- **vSrc** 要视为“ULValue 类”一节第 232 页的值。

ULValue 函数

从布尔值构造“ULValue 类”一节第 232 页。

语法

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

参数

- **val** 要视为“ULValue 类”一节第 232 页的布尔值。

ULValue 函数

从短整型值构造“ULValue 类”一节第 232 页。

语法

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

参数

- **val** 要视为“ULValue 类”一节第 232 页的短整型值。

ULValue 函数

从长整型值构造“ULValue 类”一节第 232 页。

语法

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

参数

- **val** 要视为“ULValue 类”一节第 232 页的长整型值。

ULValue 函数

从 int 构造“ULValue 类”一节第 232 页。

语法

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

参数

- **val** 要视为“ULValue 类”一节第 232 页的 INTEGER 值。

ULValue 函数

从无符号 INTEGER 构造“ULValue 类”一节第 232 页。

语法

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

参数

- **val** 要视为“ULValue 类”一节第 232 页的无符号 INTEGER 值。

ULValue 函数

从 FLOAT 构造“ULValue 类”一节第 232 页。

语法

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

参数

- **val** 要视为“ULValue 类”一节第 232 页的 FLOAT 值。

ULValue 函数

从 DOUBLE 构造“ULValue 类”一节第 232 页。

语法

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

参数

- **val** 要视为“ULValue 类”一节第 232 页的 DOUBLE 值。

ULValue 函数

从无符号 CHAR 构造“ULValue 类”一节第 232 页。

语法

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

参数

- **val** 要视为“ULValue 类”一节第 232 页的无符号 CHAR 值。

ULValue 函数

从无符号 SHORT 构造“ULValue 类”一节第 232 页。

语法

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

参数

- **val** 要视为“ULValue 类”一节第 232 页的无符号 SHORT 值。

ULValue 函数

从无符号 LONG 构造“ULValue 类”一节第 232 页。

语法

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

参数

- **val** 要视为“ULValue 类”一节第 232 页的无符号 LONG 值。

ULValue 函数

从 ul_u_big 构造“ULValue 类”一节第 232 页。

语法

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

参数

- **val** 要视为“ULValue 类”一节第 232 页的 ul_u_big 值。

ULValue 函数

从 ul_s_big 构造“ULValue 类”一节第 232 页。

语法

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

参数

- **val** 要视为“ULValue 类”一节第 232 页的 ul_s_big 值。

ULValue 函数

从 ul_binary 构造“ULValue 类”一节第 232 页。

语法

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

参数

- **val** 要视为“ULValue 类”一节第 232 页的 ul_binary 值。

ULValue 函数

从日期时间构造“ULValue 类”一节第 232 页。

语法

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

参数

- **val** 要视为“ULValue 类”一节第 232 页的 DATETIME 值。

ULValue 函数

从 STRING 构造“ULValue 类”一节第 232 页。

语法

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

参数

- **val** 指向要视为“ULValue 类”一节第 232 页的字符串的指针。

ULValue 函数

从 UNICODE 字符串构造“ULValue 类”一节第 232 页。

语法

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

参数

- **val** 指向要视为“ULValue 类”一节第 232 页的 UNICODE 字符串的指针。

ULValue 函数

从字符缓冲区构造“ULValue 类”一节第 232 页。

语法

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


参数

- **val** 保存要视为“ULValue 类”一节第 232 页的字符串的缓冲区。
- **len** 缓冲区的长度。

ULValue 函数

从 unicode 字符缓冲区构造“ULValue 类”一节第 232 页。

语法

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

参数

- **val** 保存要视为“ULValue 类”一节第 232 页的字符串的缓冲区。
- **len** 缓冲区的长度。

ULValue 函数

从 GUID 结构构造“ULValue 类”一节第 232 页。

语法

```
ULValue::ULValue(
 GUID & val
)
```

参数

- **val** 要视为“ULValue 类”一节第 232 页的 GUID 值。

DECL_DATETIME 运算符

将“ULValue 类”一节第 232 页转换为日期时间。

语法

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

GUID 运算符

将“ULValue 类”一节第 232 页转换为 GUID 结构。

语法

ULValue::operator GUID()

bool 运算符

将“ULValue 类”一节第 232 页转换为布尔值。

语法

ULValue::operator bool()

double 运算符

将“ULValue 类”一节第 232 页转换为双精度值。

语法

ULValue::operator double()

float 运算符

将“ULValue 类”一节第 232 页转换为浮点值。

语法

ULValue::operator float()

int 运算符

将“ULValue 类”一节第 232 页转换为 int 值。

语法

ULValue::operator int()

long 运算符

将“ULValue 类”一节第 232 页转换为长整型值。

语法

ULValue::operator long()

short 运算符

将“ULValue 类”一节第 232 页转换为短整型值。

语法

ULValue::operator short()

ul_s_big 运算符

将“ULValue 类”一节第 232 页转换为有符号的 bigint 值。

语法

ULValue::operator ul_s_big()

ul_u_big 运算符

将“ULValue 类”一节第 232 页转换为无符号的 bigint 值。

语法

ULValue::operator ul_u_big()

unsigned char 运算符

将“ULValue 类”一节第 232 页转换为字符值。

语法

ULValue::operator unsigned char()

unsigned int 运算符

将“ULValue 类”一节第 232 页转换为无符号的 int 值。

语法

ULValue::operator unsigned int()

unsigned long 运算符

将“ULValue 类”一节第 232 页转换为无符号的长整型值。

语法

ULValue::operator unsigned long()

unsigned short 运算符

将“ULValue 类”一节第 232 页转换为无符号的短整型值。

语法

ULValue::operator unsigned short()

operator= 函数

替换 ULValues 的 = 运算符。

语法

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

参数

- **other** 要指派给“ULValue 类”一节第 232 页的值。

~ULValue 函数

“ULValue 类”一节第 232 页的析构函数。

语法

ULValue::~ULValue()

嵌入式 SQL API 参考

目录

db_fini 函数 251
db_init 函数 252
db_start_database 函数 253
db_stop_database 函数 254
ULChangeEncryptionKey 函数 255
ULCheckpoint 函数 256
ULClearEncryptionKey 函数 257
ULCountUploadRows 函数 258
ULDropDatabase 函数 259
ULExecuteNextSQLPassthroughScript 260
ULExecuteSQLPassthroughScripts 261
ULGetDatabaseID 函数 262
ULGetDatabaseProperty 函数 263
ULGetErrorParameter 函数 264
ULGetErrorParameterCount 函数 265
ULGetLastDownloadTime 函数 266
GetSQLPassthroughScriptCount 267
ULGetSynchResult 函数 268
ULGlobalAutoincUsage 函数 270
ULGrantConnectTo 函数 271
ULInitSynchInfo 函数 272
ULIsSynchronizeMessage 函数 273
ULResetLastDownloadTime 函数 274
ULRetrieveEncryptionKey 函数 275
ULRevokeConnectFrom 函数 276
ULRollbackPartialDownload 函数 277
ULSaveEncryptionKey 函数 278
ULSetDatabaseID 函数 279
ULSetDatabaseOptionString 函数 280
ULSetDatabaseOptionULong 函数 281
ULSetSynchInfo 函数 282

ULSignalSyncIsComplete 函数 283
ULSynchronize 函数 284

本节列出了在嵌入式 SQL 应用程序中支持 UltraLite 功能的函数。

有关可用的 SQL 语句的一般信息，请参见“使用嵌入式 SQL 开发应用程序”第 29 页。

使用 EXEC SQL INCLUDE SQLCA 命令，可包含本章中函数的原型。

db_fini 函数

释放 UltraLite 运行时库使用的资源。

语法

```
unsigned short db_fini(
SQLCA * sqlca
);
```

返回值

- 如果在处理过程中出现错误，则返回 0。在 SQLCA 中设置错误代码。
- 如果没有错误，则返回非零值。

注释

在调用 db_fini 之后不能进行任何其它 UltraLite 库调用或执行任何嵌入式 SQL 命令。

为每个正被使用的 SQLCA 调用一次 db_fini。

另请参见

- “db_init 函数”一节第 252 页

db_init 函数

初始化 UltraLite 运行时库。

语法

```
unsigned short db_init(
SQLCA * sqlca
);
```

返回值

- 如果在处理过程中（例如，在持久存储区的初始化过程中）出现错误，则返回 0。在 SQLCA 中设置错误代码。
- 如果没有错误，则返回非零值。可以开始使用嵌入式 SQL 命令和函数。

注释

在进行任何其它 UltraLite 库调用之前，以及执行任何嵌入式 SQL 命令之前，都必须调用此函数。

在大多数情况下，只应调用一次此函数，传递全局 sqlca 变量的地址（如 *sqlca.h* 头文件中定义的）。如果应用程序中有多个执行路径，只要每个路径具有单独的 sqlca 指针，您就可以使用多个 db_init 调用。这个单独的 SQLCA 指针可以是用户定义的指针，也可以是用 db_fini 释放了的全局 SQLCA。

在多线程应用程序中，每个线程都必须调用 db_init 来获得单独的 SQLCA。在单个线程上执行使用此 SQLCA 的后续连接和事务。

初始化 SQLCA 还会重置任何先前调用 ULEnable 函数时的设置。如果重新初始化 SQLCA，必须发出应用程序需要的所有 ULEnable 函数。

另请参见

- “db_fini 函数”一节第 251 页

db_start_database 函数

如果数据库尚未运行，则启动该数据库。

语法

```
unsigned int db_start_database(
SQLCA * sqlca,
char * parms
);
```

参数

sqlca 指向 SQLCA 结构的指针。

parms 以空值终止的字符串，其中包含以分号分隔的参数设置列表，每个参数设置的形式均为 **KEYWORD**=*value*。通常只需要一个文件名。例如：

```
"DBF=c:\\db\\mydatabase.db"
```

返回值

- 如果数据库已经运行或成功启动，则返回 true。在这种情况下，将 SQLCODE 设置为 0。
- 错误信息也在 SQLCA 中返回。

注释

在开发结合了嵌入式 SQL 和 C++ 组件的应用程序时，需要此函数。

另请参见

- “UltraLite 连接参数” 《UltraLite - 数据库管理和参考》
- “初始化 SQL 通信区”一节第 32 页

db_stop_database 函数

停止数据库。

语法

```
unsigned int db_stop_database(
SQLCA * sqlca,
char * parms
);
```

参数

sqlca 指向 SQLCA 结构的指针。

parms 以空值终止的字符串，其中包含以分号分隔的参数设置列表，每个参数设置的形式均为 **KEYWORD**=*value*。通常只需要一个数据库文件名。例如，

```
"DBF=c:\\db\\mydatabase.db"
```

返回值

- 如果没有错误，则返回 TRUE。

注释

通常不需要此函数，因为 UltraLite 在所有连接都关闭时会自动停止数据库。但是，当开发结合了嵌入式 SQL 和 C++ 组件的应用程序时，此函数可能很有用。

此函数不停止有连接的数据库。

另请参见

- “UltraLite 连接参数” 《UltraLite - 数据库管理和参考》
- “初始化 SQL 通信区”一节第 32 页

ULChangeEncryptionKey 函数

更改 UltraLite 数据库的加密密钥。

语法

```
ul_bool ULChangeEncryptionKey(
SQLCA *sqlca,
ul_char *new_key
);
```

注释

调用此函数的应用程序必须首先确保用户同步了数据库或者创建了数据库的可靠备份副本。拥有数据库的可靠备份很重要，因为 ULChangeEncryptionKey 是必须运行到结束的操作。数据库的加密密钥更改时，将首先使用旧密钥对数据库中的每一行解密，然后使用新密钥加密并重写。*此操作是不可恢复的*。如果加密更改操作没有完成，数据库将停留在无效状态，并且将无法再次访问。

另请参见

- “对数据进行加密”一节第 50 页

ULCheckpoint 函数

执行检查点操作：将所有待执行的已提交事务刷新到数据库中。通过调用 ULCheckpoint，将不会提交任何当前事务。ULCheckpoint 函数与推迟自动事务检查点联合使用，以增强性能。有关详细信息，请参见“刷新单个或分组的事务”一节 《UltraLite - 数据库管理和参考》。

语法

```
void ULCheckpoint(
SQLCA * sqlca
);
```

注释

ULCheckpoint 函数会确保已将所有待执行的已提交事务写入数据库存储中。

另请参见

- “UltraLite 事务处理”一节 《UltraLite - 数据库管理和参考》

ULClearEncryptionKey 函数

清除加密密钥。

语法

ul_bool **ULClearEncryptionKey(**
ul_u_long * *creator*,
ul_u_long * *feature-num* **);**

参数

creator 指向保存加密密钥的功能的创建者 ID 的指针。缺省值为 NULL。

feature-num 指向保存加密密钥的功能编号的指针。如果 feature-num 的值为 NULL，应用程序使用 UltraLite 的缺省值 100。

注释

在 Palm OS 上，加密密钥作为 Palm 功能保存在动态内存中。功能的索引是根据创建者和功能编号编制的。

另请参见

- “ULRetrieveEncryptionKey 函数”一节第 275 页
- “ULSaveEncryptionKey 函数”一节第 278 页

ULCountUploadRows 函数

计算进行同步需要上载的行数。

语法

ul_u_long **ULCountUploadRows (**
SQLCA * *sqlca*,
ul_char *pub-list*,
ul_u_long *threshold*
);

参数

sqlca 指向 SQLCA 的指针。

pub-list 包含以逗号分隔的要检查的发布列表的字符串。空字符串（UL_SYNC_ALL 宏）表示包括除标记为 "no sync" 以外的所有表。只包含一个星号的字符串（UL_SYNC_ALL_PUBS 宏）表示包括在任何发布中引用的所有表。某些表可能不属于任何发布，因而即使 pub-list 字符串为 "*"，也不会将这些表包括在内。

threshold 确定要计数的最大行数，从而限制调用所花费的时间。

- 阈值为 0 对应于没有限制（即计入所有需要同步的行）。
- 阈值 1 可用于快速确定是否有需要同步的行。

返回值

- 指定的一组发布中或整个数据库中需要同步的行数。

注释

使用此函数提示用户进行同步。

示例

以下调用在整个数据库中检查要同步的行数：

```
count = ULCountUploadRows( sqlca, UL_SYNC_ALL_PUBS, 0 );
```

以下调用检查发布 PUB1 和 PUB2，最多检查 1000 行：

```
count = ULCountUploadRows( sqlca, UL_TEXT("PUB1,PUB2"), 1000 );
```

以下调用检查在发布 PUB1 和 PUB2 中是否有任何需要同步的行：

```
count = ULCountUploadRows( sqlca, UL_TEXT("PUB1,PUB2"), 1 );
```

另请参见

- “UL_SYNC_ALL 宏”一节第 119 页
- “UL_SYNC_ALL_PUBS 宏”一节第 120 页

ULDropDatabase 函数

删除 UltraLite 数据库文件，同时删除任何关联的临时文件或工作文件。

语法

```
ul_bool ULDropDatabase (
SQLCA * sqlca,
ul_char * store-parms
);
```

参数

sqlca 指向 SQLCA 的指针。

store-parms 以空值终止的连接字符串，其中包含以分号分隔的参数设置列表，每个参数设置的形式均为 **KEYWORD**=*value*。

返回值

- **ul_true** 数据库文件已成功删除。
- **ul_false** 无法删除该数据库文件。详细的错误消息由 SQLCA 中的 sqlcode 字段定义。通常，失败的原因是文件名不正确，或者对该文件的访问被拒绝（可能由于某个应用程序打开了该文件）。

注释

仅在以下情况中调用此函数：

- 不存在打开的数据库连接时
- 调用 db_init 之前或调用 calling db_fini 之后。

在 Palm OS 上，仅在以下情况中调用此函数：

- 未连接到数据库时
- 调用任何 ULEnable 之后。

小心
此函数删除数据库文件和其中的所有数据。此操作是不可恢复的。因此，请谨慎使用此函数。

示例

以下调用将删除 UltraLite 数据库文件 *myfile.udb*。

```
if( ULDropDatabase(&sqlca, UL_TEXT("file_name=myfile.udb") ) ){
    // success
};
```

ULExecuteNextSQLPassthroughScript

执行下一个 SQL 直通脚本。

语法

```
bool ULExecuteNextSQLPassthroughScript(
SQLCA * sqlca
)
```

参数

sqlca 指向 SQLCA 的指针。

返回值

- 执行脚本时如果发生任何错误，则返回 false。

另请参见

- “UltraLite global_database_id 选项”一节 《UltraLite - 数据库管理和参考》

ULExecuteSQLPassthroughScripts

执行所有可用的 SQL 直通脚本。

语法

bool**ULExecuteSQLPassthroughScripts(**
SQLCA * *sqlca*
)

参数

sqlca 指向 SQLCA 的指针。

返回值

- 执行脚本时如果发生任何错误，则返回 false。

另请参见

- “UltraLite global_database_id 选项”一节 《UltraLite - 数据库管理和参考》

ULGetDatabaseID 函数

获取当前数据库 ID。

语法

ul_u_long **ULGetDatabaseID(**
SQLCA * *sqlca*
)

参数

sqlca 指向 SQLCA 的指针。

返回值

- 返回由上一次 SetDatabaseID 调用设置的值。
- 如果从未设置过此 ID，则返回 UL_INVALID_DATABASE_ID。

注释

用于全局自动增量的当前数据库 ID。

另请参见

- “UltraLite global_database_id 选项”一节 《UltraLite - 数据库管理和参考》

ULGetDatabaseProperty 函数

获取数据库属性的值。

语法

```
void ULGetDatabaseProperty (
SQLCA * sqlca,
ul_database_property_id id,
char * dst,
size_t buffer-size,
ul_bool * null-indicator
);
```

参数

sqlca 指向 SQLCA 的指针。

id 数据库属性的标识符。

dst 保存属性值的字符数组。

buffer-size 字符数组 *dst* 的大小。

null-indicator 数据库参数为空的指示符。

另请参见

- “UltraLite 数据库属性” 《UltraLite - 数据库管理和参考》

ULGetErrorParameter 函数

按顺序检索错误参数。

语法

```
size_t ULGetErrorParameter (
SQLCA * sqlca,
ul_u_long parm_num,
ul_char * buffer,
size_t size
);
```

参数

sqlca 指向 sqlca 的指针。

parm_num 参数序号。

buffer 指向包含错误参数的缓冲区的指针。

size 缓冲区的大小（以字节为单位）。

返回值

此函数返回复制到提供的缓冲区的字符数。

另请参见

- “ULGetErrorParameterCount 函数”一节第 265 页

ULGetErrorParameterCount 函数

获取错误参数的数量计数。

语法

```
ul_u_long ULGetErrorParameterCount (
SQLCA * sqlca
);
```

参数

sqlca 指向 SQLCA 的指针。

返回值

此函数返回错误参数的数目。除非结果为 0，否则此结果中从 1 开始的值可用于调用 ULGetErrorParameter 来检索对应的错误参数值。

另请参见

- “ULGetErrorParameter 函数”一节第 264 页

ULGetLastDownloadTime 函数

获得上次下载指定发布的时间。

语法

```
ul_bool ULGetLastDownloadTime (
SQLCA * sqlca,
ul_string pub-name,
DECL_DATETIME * value
);
```

参数

sqlca 指向 SQLCA 的指针。

pub-name 包含为其检索上次下载时间的发布名称的字符串。

value 指向要填充的 **DECL_DATETIME** 结构的指针。例如，值为 1990 年 1 月 1 日 说明仍必须同步该发布。

返回值

- **true** *value* 由 *pub-name* 指定的发布的上次下载时间成功填充。
- **false** *pub-name* 指定了多个发布，或者所指定的发布未定义。*value* 的内容没有意义。

示例

以下调用使用发布 UL_PUB_PUB1 的下载日期和时间填充 dt 结构：

```
DECL_DATETIME dt;
ret = ULGetLastDownloadTime( &sqlca, UL_TEXT("UL_PUB_PUB1"), &dt );
```

另请参见

- “UL_SYNC_ALL 宏”一节第 119 页
- “UL_SYNC_ALL_PUBS 宏”一节第 120 页

GetSQLPassthroughScriptCount

获取可运行的 SQL 直通脚本的数目。

语法

ul_u_long**ULExecuteNextSQLPassthroughScript(**
SQLCA * *sqlca*
)

参数

sqlca 指向 SQLCA 的指针。

返回值

- 返回可运行的 SQL 直通脚本的数目。

另请参见

- “UltraLite global_database_id 选项”一节 《UltraLite - 数据库管理和参考》

ULGetSynchResult 函数

包含最近进行的同步的结果的结构，以便在应用程序中可以采取相应的操作：

语法

```
ul_bool ULGetSynchResult(
ul_synch_result  * synch-result
);
```

参数

synch-result　一个用于保存同步结果的结构。此结构在 *ulglobal.h* 中定义如下：

```
typedef struct ul_synch_result {
    an_sql_code         sql_code;
    char                sql_error_string[];
    ul_stream_error     stream_error;
    ul_bool             upload_ok;
    ul_bool             ignored_rows;
    ul_auth_status      auth_status;
    ul_s_long           auth_value;
    SQLDATETIME         timestamp;
    ul_bool             partial_download_retained;
    ul_synch_status     status;
} ul_synch_result, * p_ul_synch_result;
```

各参数包括：

- **sql_code**　上一次同步中的 SQL 代码。有关 SQL 代码的列表，请参见“按 SQLSTATE 排序的 SQL Anywhere 错误消息”一节 《错误消息》。
- **sql_error_string**　与在字段 **sql_code** 中报告的错误相关联的错误消息文本。
- **stream_error**　ul_stream_error 类型的结构。请参见“Stream Error 同步参数”一节 《UltraLite - 数据库管理和参考》。
- **upload_ok**　如果上载成功则为 true；否则为 false。
- **ignored_rows**　如果忽略了上载的行，则为 true；否则，为 false。
- **auth_status**　同步验证状态。请参见“Authentication Status 同步参数”一节 《UltraLite - 数据库管理和参考》。
- **auth_value**　MobiLink 服务器用于确定 **auth_status** 结果的值。请参见“Authentication Value 同步参数”一节 《UltraLite - 数据库管理和参考》。
- **timestamp**　上一次同步的时间和日期。
- **partial_download_retained**　指示是否已保留部分下载的标记。
- **status**　观察器函数使用的状态信息。请参见“Observer 同步参数”一节 《UltraLite - 数据库管理和参考》。

返回值

- 如果操作成功，则返回 True。
- 如果操作失败，则返回 False。

注释

应用程序必须先分配一个 ul_synch_result 对象，然后才能将其传递给 ULGetSynchResult。该函数用上次同步的结果填充 ul_synch_result。这些结果永久存储在数据库中。

此函数在使用 HotSync 在 Palm OS 上同步应用程序时特别有用，因为同步发生在应用程序自身之外。连接中设置的 **SQLCODE** 值反映连接操作自身的结果。同步状态和结果仅写入 HotSync 日志。要获取扩展的同步结果信息，请在连接数据库时调用 ULGetSynchResult。

示例

下面的代码检查上一同步是否成功。

```
ul_synch_result synch_result;
memset( &synch_result, 0, sizeof( ul_synch_result ) );
db_init( &sqlca );
EXEC SQL CONNECT "dba" IDENTIFIED BY "sql";
if( !ULGetSynchResult( &sqlca, &synch_result ) ) {
    prMsg( "ULGetSynchResult failed" );
}
```

ULGlobalAutoincUsage 函数

获得在具有全局自动增量缺省值的所有列中使用的缺省值的百分比。

语法

```
ul_u_short ULGlobalAutoincUsage(
SQLCA * sqlca
);
```

返回值

- 范围在 0-100 之间的短整型值。

注释

如果数据库包含多个具有此缺省值的列，则将计算所有列的该百分比值，并返回最大值。例如，返回值为 99 表示至少为一列保留了极少的缺省值。

另请参见

- “ULSetDatabaseID 函数”一节第 279 页
- “UltraLite global_database_id 选项”一节 《UltraLite - 数据库管理和参考》

ULGrantConnectTo 函数

授予新的或现有的用户 ID 使用给定口令访问 UltraLite 数据库的权限。

语法

```
void ULGrantConnectTo(
SQLCA * sqlca,
ul_char * userid,
ul_char * password
);
```

参数

sqlca 指向 SQLCA 的指针。

userid 保存用户 ID 的字符数组。最大长度为 16 个字符。

password 保存该用户 ID 的口令的字符数组。最大长度为 16 个字符。

注释

如果指定了现有的用户 ID，则此函数更新该用户的口令。

另请参见

- “验证用户”一节第 48 页
- “ULRevokeConnectFrom 函数”一节第 276 页

ULInitSynchInfo 函数

初始化同步信息结构。

语法

```
void ULInitSynchInfo(
ul_synch_info *  synch_info
);
```

参数

synch_info　一个同步结构。有关此结构的成员的详细信息，请参见“UltraLite 的同步参数”一节 《UltraLite - 数据库管理和参考》。

ULIsSynchronizeMessage 函数

检查消息，看它是否是来自 ActiveSync 的 MobiLink 提供程序的同步消息，以便可以调用处理这种消息的代码。完成对同步消息的处理后，应调用 ULSignalSyncIsComplete 函数。

语法

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

注释

应将对此函数的调用包括在应用程序的 WindowProc 函数中。

适用于用于 ActiveSync 的 Windows Mobile。

示例

下面的代码片段说明如何使用 ULIsSynchronizeMessage 处理同步消息。

```
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;
}
```

另请参见

- “向应用程序添加 ActiveSync 同步”一节第 86 页
- “Windows Mobile 上的 ActiveSync”一节 《UltraLite - 数据库管理和参考》
- “ULSignalSyncIsComplete 函数”一节第 283 页

ULResetLastDownloadTime 函数

重置发布的上次下载时间，以便应用程序能够重新同步以前下载的数据。

语法

void **ULResetLastDownloadTime(**
SQLCA * *sqlca*,
ul_string *pub-list*
);

参数

sqlca 指向 SQLCA 的指针。

pub-list 包含以逗号分隔的要重置的发布列表的字符串。空字符串表示包括除标记为 "no sync" 以外的所有表。只包含一个星号 ("*") 的字符串表示包括所有发布。某些表可能不属于任何发布，因而即使 pub-list 字符串为 "*"，也不会将这些表包括在内。

示例

下面的函数调用为所有表重置上次下载时间：

```
ULResetLastDownloadTime( &sqlca, UL_TEXT("") );
```

另请参见

- “ULGetLastDownloadTime 函数”一节第 266 页
- “基于时间戳的下载”一节 《MobiLink - 服务器管理》
- “UL_SYNC_ALL 宏”一节第 119 页
- “UL_SYNC_ALL_PUBS 宏”一节第 120 页

ULRetrieveEncryptionKey 函数

从内存中检索加密密钥。

语法

```
ul_bool ULRetrieveEncryptionKey(
ul_char * key,
ul_u_short len,
ul_u_long * creator,
ul_u_long * feature-num
);
```

参数

key 指向缓冲区的指针，在该缓冲区中保存检索到的加密密钥。

len 缓冲区长度，该缓冲区保存加密密钥和一个终止空值字符。

creator 指向保存加密密钥的功能的创建者 ID 的指针。缺省值为 NULL。

feature-num 指向保存加密密钥的功能编号的指针。如果 feature-num 的值为 NULL，应用程序使用 UltraLite 的缺省值 100。

返回值

- 如果操作成功，则返回 true。
- 如果操作失败，则返回 false。如果未找到该功能，或者如果所提供的缓冲区长度不够保存密钥加上终止空字符，就会出现这种情况。

注释

在 Palm OS 上，加密密钥作为 Palm 功能保存在动态内存中。功能的索引是根据创建者和功能编号编制的。

适用于 Palm OS。

另请参见

- “ULClearEncryptionKey 函数”一节第 257 页
- “ULSaveEncryptionKey 函数”一节第 278 页

ULRevokeConnectFrom 函数

从 UltraLite 数据库撤消用户 ID 的访问权限。

语法

```
void ULRevokeConnectFrom(
SQLCA * sqlca,
ul_char * userid
);
```

参数

sqlca 指向 SQLCA 的指针。

userid 保存要从数据库访问中排除的用户 ID 的字符数组。最大长度为 16 个字符。

另请参见

- “验证用户”一节第 48 页
- “ULGrantConnectTo 函数”一节第 271 页

ULRollbackPartialDownload 函数

回退失败的同步所做的更改。

语法

void **ULRollbackPartialDownload(**
SQLCA * *sqlca*
);

参数

- **sqlca** 指向 SQL 通信区的指针。在 C++ API 中，请使用 Sqlca.GetCA 方法。

注释

在同步的下载阶段出现通信错误时，UltraLite 可以应用已下载的更改，以便应用程序可以从同步中断的位置恢复同步。如果不需要下载的更改（用户或应用程序不想从该位置恢复下载），则 ULRollbackPartialDownload 回退失败的下载事务。

另请参见

- “GetCA 函数”一节第 139 页
- “恢复失败的下载”一节 《MobiLink - 服务器管理》
- “Keep Partial Download 同步参数”一节 《UltraLite - 数据库管理和参考》
- “Partial Download Retained 同步参数”一节 《UltraLite - 数据库管理和参考》
- “Resume Partial Download 同步参数”一节 《UltraLite - 数据库管理和参考》

ULSaveEncryptionKey 函数

将加密密钥保存在 Palm 的动态内存中。

语法

```
ul_bool ULSaveEncryptionKey(
ul_char * key,
ul_u_long * creator,
ul_u_long * feature-num
);
```

参数

key 指向加密密钥的指针。

creator 指向保存加密密钥的功能的创建者 ID 的指针。缺省值为 NULL。

feature-num 指向保存加密密钥的功能编号的指针。如果 feature-num 的值为 NULL，应用程序使用 UltraLite 的缺省值 100。

返回值

- 如果操作成功，则返回 true。
- 如果操作失败，则返回 false。如果未找到该功能，或者如果所提供的缓冲区长度不够保存密钥加上终止空字符，就会出现这种情况。

注释

在 Palm OS 上，加密密钥作为 Palm 功能保存在动态内存中。功能的索引是根据创建者和功能编号编制的。系统不会备份它们，且在设备的任何一次重置时清除它们。

适用于 Palm OS 应用程序。

另请参见

- “ULClearEncryptionKey 函数”一节第 257 页
- “ULRetrieveEncryptionKey 函数”一节第 275 页

ULSetDatabaseID 函数

设置数据库标识编号。

语法

```
void ULSetDatabaseID(
SQLCA * sqlca,
ul_u_long id
);
```

参数

sqlca 指向 SQLCA 的指针。

id 在复制或同步设置中唯一标识特定数据库的正整数。

另请参见

- “UltraLite global_database_id 选项”一节 《UltraLite - 数据库管理和参考》
- “ULGlobalAutoincUsage 函数”一节第 270 页

ULSetDatabaseOptionString 函数

从字符串值设置数据库选项。

语法

```
void ULSetDatabaseOptionString (
SQLCA * sqlca,
ul_database_option_id id,
ul_char * value
);
```

参数

sqlca 指向 SQLCA 的指针。

id 要设置的数据库选项的标识符。

value 数据库选项的值。

ULSetDatabaseOptionULong 函数

设置数字数据库选项。

语法

```
void ULSetDatabaseOptionULong(
SQLCA * sqlca,
ul_database_option_id id,
ul_u_long * value
);
```

参数

sqlca 指向 SQLCA 的指针。

id 要设置的数据库选项的标识符。

value 数据库选项的值。

ULSetSynchInfo 函数

存储要和 HotSync 一起使用的同步参数。

语法

```
ul_bool ULSetSynchInfo(
SQLCA * sqlca,
PROFILE sync-profile-name
ul_synch_info *  synch_info
);
```

参数

sqlca　指向 SQLCA 的指针。

sync-profile-name　向其中存储同步信息的同步配置文件的名称。然后这就可作为 HotSync 的一部分通过名称引用。请参见“Palm OS 的 UltraLite HotSync 管道安装实用程序 (ulcond11)”一节《UltraLite - 数据库管理和参考》。

synch_info　一个同步结构。有关此结构的成员的详细信息，请参见“ul_synch_info_a 结构”一节第 127 页。

注释

通常情况下，在即将通过 db_fini 关闭应用程序之前调用 ULSetSynchInfo。

适用于配合 HotSync 使用的 Palm OS 应用程序。

另请参见

- “db_fini 函数”一节第 251 页

ULSignalSyncIsComplete 函数

调用此函数可指示对同步消息的处理已完成。

语法

void **ULSignalSyncIsComplete();**

注释

使用 ActiveSync 提供程序注册的应用程序需要在完成对同步消息的处理时在其 WNDPROC 中调用此方法。

另请参见

- “ULIsSynchronizeMessage 函数”一节第 273 页

ULSynchronize 函数

启动 UltraLite 应用程序中的同步。

语法

```
void ULSynchronize(
SQLCA * sqlca,
ul_synch_info * synch_info
);
```

参数

sqlca 指向 SQLCA 的指针。

synch_info 一个同步结构。同步的具体信息由一组同步参数控制。有关这些参数的详细信息，请参见“ul_synch_info_a 结构”一节第 127 页。

注释

对于 TCP/IP 或 HTTP 同步，ULSynchronize 函数将启动同步。在同步过程中出现的、没有被 handle_error 脚本处理的错误将作为 SQL 错误进行报告。应用程序应测试此函数的 SQLCODE 返回值。

UltraLite ODBC API 参考

目录

SQLAllocHandle 函数 286
SQLBindCol 函数 287
SQLBindParameter 函数 288
SQLConnect 函数 289
SQLDescribeCol 函数 290
SQLDisconnect 函数 291
SQLEndTran 函数 292
SQLExecDirect 函数 293
SQLExecute 函数 294
SQLFetch 函数 295
SQLFetchScroll 函数 296
SQLFreeHandle 函数 297
SQLGetCursorName 函数 298
SQLGetData 函数 299
SQLGetDiagRec 函数 300
SQLGetInfo 函数 301
SQLNumResultCols 函数 302
SQLPrepare 函数 303
SQLRowCount 函数 304
SQLSetConnectionName 函数 305
SQLSetCursorName 函数 306
SQLSetSuspend 函数（不建议使用） 307
SQLSynchronize 函数 308

本节介绍了 UltraLite 所支持的 ODBC 接口的组件。

本章并不是全面的 ODBC 参考，它旨在作为快速参考对 ODBC 的主要参考（即 Microsoft ODBC 程序员参考）予以补充。

SQLAllocHandle 函数

为 UltraLite ODBC 分配句柄。

语法

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLAllocHandle(**
SQLSMALLINT *HandleType*,
SQLHANDLE *InputHandle*,
SQLHANDLE * *OutputHandle* **)**;

参数

- **HandleType** 要分配的句柄类型。UltraLite 支持以下句柄类型：
  - SQL_HANDLE_ENV（环境句柄）
  - SQL_HANDLE_DBC（连接句柄）
  - SQL_HANDLE_STMT（语句句柄）
- **InputHandle** 要在其上下文中分配新句柄的句柄。对于连接句柄，这是环境句柄；对于语句句柄，这是连接句柄。
- **OutputHandle** 指向在其中返回新句柄的缓冲区的指针。

注释

ODBC 使用句柄提供数据库操作上下文。环境句柄提供与数据源（如其它接口中的 SQL 通信区）进行通信的上下文。连接句柄提供所有数据库操作的上下文。语句句柄管理结果集和数据修改。描述符句柄管理对结果集数据类型的处理。

SQLBindCol 函数

在 UltraLite ODBC 中，此函数用于将结果集列绑定到应用程序数据缓冲区。

语法

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLBindCol (
SQLHSTMT StatementHandle,
SQLUSMALLINT ColumnNumber,
SQLSMALLINT TargetType,
SQLPOINTER TargetValue,
SQLLEN BufferLength,
SQLLEN * StrLen_or_Ind );
```

参数

- **StatementHandle** 要返回结果集的语句的句柄。
- **ColumnNumber** 结果集中要绑定到应用程序数据缓冲区的列的编号。
- **TargetType** *TargetValue* 指针的数据类型标识符。
- **TargetValue** 指向要绑定到列的数据缓冲区的指针。
- **BufferLength** *TargetValue* 缓冲区的长度（以字节为单位）。
- **StrLen_or_Ind** 指向要绑定到列的长度或指示符缓冲区的指针。对于字符串，长度缓冲区存放了返回的实际字符串的长度（可能小于列允许的长度）。

注释

为了在您的应用程序和数据库之间交换信息，ODBC 会将应用程序中的缓冲区绑定到数据库对象（例如，列）。当执行查询以便在应用程序中标识一个供 UltraLite 存放指定列的值的缓冲区时，会使用 SQLBindCol。

SQLBindParameter 函数

在 UltraLite ODBC 中，此函数用于将缓冲区参数绑定到 SQL 语句中的参数标记。

语法

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLBindParameter (**
SQLHSTMT *StatementHandle*,
SQLUSMALLINT *ParameterNumber*,
SQLSMALLINT *ParamType*,
SQLSMALLINT *CType*,
SQLSMALLINT *SqlType*,
SQLULEN *ColDef*,
SQLSMALLINT *Scale*,
SQLPOINTER *rgbValue*,
SQLLEN *cbValueMax*,
SQLLEN * *StrLen_or_Ind* **)**;

参数

- **StatementHandle** 语句句柄。
- **ParameterNumber** 语句中的参数标记编号（从 1 开始顺序计数）。
- **ParamType** 参数类型。以下类型之一：
  - SQL_PARAM_INPUT
  - SQL_PARAM_INPUT_OUTPUT
  - SQL_PARAM_OUTPUT
- **CType** C 数据类型的参数。
- **SQLType** SQL 数据类型的参数。
- **ColDef** 列或参数标记表达式的大小。
- **Scale** 列或参数标记表达式的小数位数。
- **rgbValue** 指向参数数据缓冲区的指针。
- **cbValueMax** *rgbValue* 缓冲区的长度。
- **StrLen_or_Ind** 指向参数长度缓冲区的指针。

注释

为了在您的应用程序和数据库之间交换信息，ODBC 会将应用程序中的缓冲区绑定到数据库对象（例如，列）。当执行语句以便在应用程序中标识一个供 UltraLite 获取或设置查询中指定参数的值的缓冲区时，会使用 SQLBindParameter。

SQLConnect 函数

在 UltraLite ODBC 中，此函数用于连接数据库。

语法

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLConnect (**
SQLHDBC *ConnectionHandle*,
SQLTCHAR * *ServerName*,
SQLSMALLINT *NameLength1*,
SQLTCHAR * *UserName*,
SQLSMALLINT *NameLength2*,
SQLTCHAR * *Authentication*,
SQLSMALLINT *NameLength3* **)**;

参数

- **ConnectionHandle** 连接句柄。
- **ServerName** 一个连接字符串，定义应用程序要连接的数据库。UltraLite ODBC 不使用 ODBC 数据源，而是提供连接字符串，其中包含数据库连接参数以及可选的其它参数。

  下面是 ServerName 参数的示例：

  ```
  (SQLTCHAR*)UL_TEXT(
      "dbf=customer.udb"
  )
  ```

  有关连接参数的完整列表，请参见“UltraLite 连接参数” 《UltraLite - 数据库管理和参考》。
- **NameLength1** **ServerName* 的长度。
- **UserName** 连接时使用的用户 ID。该用户 ID 也可以在向 ServerName 参数提供的连接字符串中指定。
- **NameLength2** **UserName* 的长度。
- **Authentication** 连接时使用的口令。该口令也可以在向 ServerName 参数提供的连接字符串中指定。
- **NameLength3** **Authentication* 的长度。

注释

连接到数据库。有关 UltraLite 连接参数的详细信息，请参见“UltraLite 连接参数” 《UltraLite - 数据库管理和参考》。

SQLDescribeCol 函数

对于 UltraLite ODBC，返回结果集中某列的结果描述符。

此描述符包括列名、列大小、数据类型、小数位数以及是否为空。

语法

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLDescribeCol (
   SQLHSTMT StatementHandle,
   SQLUSMALLINT ColumnNumber,
   SQLTCHAR * ColumnName,
   SQLSMALLINT BufferLength,
   SQLSMALLINT * NameLength,
   SQLSMALLINT * DataType,
   SQLULEN * ColumnSize,
   SQLSMALLINT * DecimalDigits,
   SQLSMALLINT * Nullable );
```

参数

- **StatementHandle** 语句句柄。
- **ColumnNumber** 结果数据的列号（从 1 开始）。
- **ColumnName** 指向在其中返回列名的缓冲区的指针。
- **BufferLength** **ColumnName* 的长度（以字符为单位）。
- **NameLength** 指向缓冲区的指针，在该缓冲区中返回 **ColumnName* 中可返回的总字节数（不包括空终止字节）。
- **DataType** 指向缓冲区的指针，在该缓冲区中返回列的 SQL 数据类型。
- **ColumnSize** 指向缓冲区的指针，在该缓冲区中返回数据源中列的大小。
- **DecimalDigits** 指向缓冲区的指针，在该缓冲区中返回数据源中列的小数位数。
- **Nullable** 指向缓冲区的指针，在该缓冲区中返回一个值，指示该列是否允许 NULL 值。

SQLDisconnect 函数

在 UltraLite ODBC 中，此函数用于断开应用程序与数据库的连接。

语法

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLDisconnect (**
SQLHDBC *ConnectionHandle* **)**;

参数

- **ConnectionHandle** 要关闭的连接的句柄。

注释

调用了 SQLDisconnect 后，如果不打开一个新连接，就无法进一步操作数据库。

另请参见

- “SQLConnect 函数”一节第 289 页

SQLEndTran 函数

在 UltraLite ODBC 中，此函数用于提交或回退事务。

语法

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLEndTran (**
SQLSMALLINT *HandleType*,
SQLHANDLE *Handle*,
SQLSMALLINT *CompletionType* **)**;

参数

- **HandleType** 要分配的句柄类型。UltraLite 支持以下句柄类型：
  - SQL_HANDLE_ENV
  - SQL_HANDLE_DBC
  - SQL_HANDLE_STMT
- **Handle** 指示事务作用域的连接句柄。
- **CompletionType** 以下两个值之一：
  - SQL_COMMIT
  - SQL_ROLLBACK

SQLExecDirect 函数

在 UltraLite ODBC 中，此函数用于执行 SQL 语句。

语法

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLExecDirect (
SQLHSTMT StatementHandle,
SQLTCHAR * StatementText,
SQLINTEGER TextLength );
```

参数

- **StatementHandle** 语句句柄。
- **StatementText** SQL 语句的文本。
- **TextLength** **StatementText* 的长度。

注释

与 SQLExecute 不同，使用 SQLExecDirect 时，执行语句之前不需要准备语句。

对于重复执行的语句，SQLExecDirect 在执行速度上比 SQLExecute 慢。

另请参见

- “SQLExecute 函数”一节第 294 页

SQLExecute 函数

在 UltraLite ODBC 中，此函数用于执行 SQL 预准备语句。

语法

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLExecute (**
SQLHSTMT *StatementHandle* **)**;

参数

- **StatementHandle** 要执行的语句的句柄。

注释

必须先用 SQL Prepare 准备语句，才能执行它。如果语句有参数标记，则必须在执行前使用 SQLBindParameter 将它们绑定到变量。

使用 SQLExecDirect，无需事先准备就可以执行语句。对于重复执行的语句，SQLExecDirect 的性能比 SQLExecute 低。

另请参见

- “SQLBindParameter 函数”一节第 288 页
- “SQLPrepare 函数”一节第 303 页
- “SQLExecDirect 函数”一节第 293 页

SQLFetch 函数

在 UltraLite ODBC 中，此函数用于读取结果集中的下一行，并返回所有绑定列的数据。

语法

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLFetch (**
SQLHSTMT *StatementHandle* **)**;

参数

- **StatementHandle** 语句句柄。

注释

在读取行之前，必须已使用 SQLBindCol 将结果集中的列绑定到缓冲区。要读取结果集中下一行以外的某一行，请使用 SQLFetchScroll。

另请参见

- “SQLFetchScroll 函数”一节第 296 页
- “SQLBindCol 函数”一节第 287 页

SQLFetchScroll 函数

对于 UltraLite ODBC，此函数用于读取结果集中的指定行，并返回所有绑定列的数据。

语法

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLFetchScroll (**
SQLHSTMT *StatementHandle*,
SQLSMALLINT *FetchOrientation*,
SQLLEN *FetchOffset* **)**;

参数

- **StatementHandle** 语句句柄。
- **FetchOrientation** 读取类型。
- **FetchOffset** 要读取的行号。其解释取决于 *FetchOrientation* 的值。

注释

在读取行之前，必须已使用 SQLBindCol 将结果集中的列绑定到缓冲区。SQLFetchScroll 在不适合使用更直接的 SQLFetch 的情况下使用。

另请参见

- “SQLFetch 函数”一节第 295 页
- “SQLBindCol 函数”一节第 287 页

SQLFreeHandle 函数

对于 UltraLite ODBC，释放所分配句柄的资源。

语法

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLFreeHandle (**
SQLSMALLINT *HandleType*,
SQLHANDLE *Handle* **)**;

参数

- **HandleType** 要分配的句柄类型。UltraLite 支持以下句柄类型：
  - SQL_HANDLE_ENV
  - SQL_HANDLE_DBC
  - SQL_HANDLE_STMT
- **Handle** 要释放的句柄。

注释

当不再需要句柄时，应该对每个使用 SQLAllocHandle 分配的句柄调用 SQLFreeHandle。

另请参见

- “SQLAllocHandle 函数”一节第 286 页

SQLGetCursorName 函数

在 UltraLite ODBC 中，此函数用于返回与指定语句的游标关联的名称。

语法

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLGetCursorName (
  SQLHSTMT StatementHandle,
  SQLTCHAR * CursorName,
  SQLSMALLINT BufferLength,
  SQLSMALLINT * NameLength );
```

参数

- **StatementHandle** 语句句柄。
- **CursorName** 指向缓冲区的指针，在该缓冲区中返回与 *StatementHandle* 关联的游标名。
- **BufferLength** **CursorName* 的长度。
- **NameLength** 指向内存的指针，在该内存中返回 **CursorName* 中可返回的总字节数（不包括空终止字符）。

另请参见

- “SQLSetCursorName 函数”一节第 306 页

SQLGetData 函数

对于 UltraLite ODBC，检索结果集中单个列的数据。

语法

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLGetData (**
SQLHSTMT *StatementHandle*,
SQLUSMALLINT *ColumnNumber*,
SQLSMALLINT *TargetType*,
SQLPOINTER *TargetValue*,
SQLLEN *BufferLength*,
SQLLEN * *StrLen_or_Ind* **)**;

参数

- **StatementHandle** 语句句柄。
- **ColumnNumber** 结果集中要绑定的列的编号。
- **TargetType** 输出句柄。
- **TargetValue** 指向要绑定到列的数据缓冲区的指针。
- **BufferLength** *TargetValue* 缓冲区的长度（以字节为单位）。
- **StrLen_or_Ind** 指向要绑定到列的长度或指示符缓冲区的指针。

注释

SQLGetData 通常用于检索分成几部分的长度可变的数据。

SQLGetDiagRec 函数

在 UltraLite ODBC 中，此函数用于返回诊断状态记录的多个字段的当前值。

语法

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLGetDiagRec (
SQLSMALLINT HandleType,
SQLHANDLE Handle,
SQLSMALLINT RecNumber,
SQLTCHAR * Sqlstate,
SQLINTEGER * NativeError,
SQLTCHAR * MessageText,
SQLSMALLINT BufferLength,
SQLSMALLINT * TextLength );
```

参数

- **HandleType** 要分配的句柄类型。UltraLite 支持以下句柄类型：
  - SQL_HANDLE_ENV
  - SQL_HANDLE_DBC
  - SQL_HANDLE_STMT
- **Handle** 输入句柄。
- **RecNumber** 输出句柄。
- **Sqlstate** ANSI/ISO SQLSTATE 错误值。有关列表，请参见“按 SQLSTATE 排序的 SQL Anywhere 错误消息”一节 《错误消息》。
- **NativeError** SQLCODE 错误值。有关列表，请参见“按 SQLCODE 排序的 SQL Anywhere 错误消息”一节 《错误消息》。
- **MessageText** 错误消息或状态消息的文本。
- **BufferLength** **MessageText* 缓冲区的长度（以字节为单位）。
- **TextLength** 指向缓冲区的指针，在该缓冲区中返回 **MessageText* 中可返回的总字节数（不包括空终止字节）。

SQLGetInfo 函数

在 UltraLite ODBC 中，此函数用于返回有关当前 ODBC 驱动程序和数据源的一般信息。

语法

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLGetInfo (**
SQLHDBC *ConnectionHandle*,
SQLUSMALLINT *InfoType*,
SQLPOINTER * *InfoValue*,
SQLSMALLINT *BufferLength*,
SQLSMALLINT ODBCFAR * *StringLength* **)**;

参数

- **ConnectionHandle** 连接句柄。
- **InfoType** 返回的信息类型。唯一支持的类型是 SQL_DBMS_VER。返回的信息是一个标识软件当前版本的字符串。
- **InfoValue** 指向在其中返回信息的缓冲区的指针。
- **BufferLength** **InfoValue* 缓冲区的长度（以字节为单位）。
- **StringLength** 指向缓冲区的指针，在该缓冲区中返回 **InfoValue* 中可返回的总字节数（不包括字符数据的空终止字符）。

SQLNumResultCols 函数

在 UltraLite ODBC 中，此函数用于返回结果集中的列数。

语法

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLNumResultCols (**
SQLHSTMT *StatementHandle*,
SQLSMALLINT * *ColumnCount* **)**;

参数

- **StatementHandle** 语句句柄。
- **ColumnCount** 指向缓冲区的指针，在该缓冲区中返回结果集中的总列数。

SQLPrepare 函数

在 UltraLite ODBC 中，此函数用于准备要执行的 SQL 语句。

语法

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLPrepare (**
SQLHSTMT *StatementHandle*,
SQLTCHAR * *StatementText*,
SQLINTEGER *TextLength* **)**;

参数

- **StatementHandle** 语句句柄。
- **StatementText** 指向存放 SQL 语句文本的缓冲区的指针。
- **TextLength** **StatementText* 的长度。

另请参见

- “SQLExecute 函数”一节第 294 页

SQLRowCount 函数

在 UltraLite ODBC 中，此函数用于返回受 INSERT、UPDATE 或 DELETE 操作影响的行数。

语法

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLRowCount (
SQLHSTMT StatementHandle,
SQLLEN * RowCount );
```

参数

- **StatementHandle** 语句句柄。
- **RowCount** 指向返回行数的缓冲区的指针。

SQLSetConnectionName 函数

对于 UltraLite ODBC，此函数用于为挂起和恢复操作设置连接名。此函数是 UltraLite 特有的，不是 ODBC 标准的组成部分。

语法

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLSetConnectionName (
SQLHSTMT StatementHandle,
SQLTCHAR * ConnectionName,
SQLSMALLINT NameLength );
```

参数

- **StatementHandle** 语句句柄。
- **ConnectionName** 指向存放连接名的缓冲区的指针。
- **NameLength** *ConnectionName 的长度

注释

SQLSetConnectionName 与 SQLSetSuspend 一起用于提供挂起和恢复操作中使用的连接名。在打开连接之前设置连接名，以恢复应用程序状态。

另请参见

- “在 UltraLite Palm 应用程序中维护状态（不建议使用）”一节第 68 页
- “SQLSetSuspend 函数（不建议使用）”一节第 307 页

SQLSetCursorName 函数

在 UltraLite ODBC 中，此函数用于设置与 SQL 语句关联的游标名。

语法

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLSetCursorName (
SQLHSTMT StatementHandle,
SQLTCHAR * CursorName,
SQLSMALLINT NameLength );
```

参数

- **StatementHandle** 语句句柄。
- **CursorName** 指向存放游标名的缓冲区的指针。
- **NameLength** *CursorName 的长度。

另请参见

- “SQLGetCursorName 函数”一节第 298 页

SQLSetSuspend 函数（不建议使用）

对于 UltraLite ODBC，此函数指示关闭应用程序时是否应该保存打开游标的状态。此函数是 UltraLite 特有的，不是 ODBC 标准的组成部分。

语法

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLSetSuspend (
SQLSMALLINT HandleType,
SQLHSTMT StatementHandle,
SQLSMALLINT TrueFalse );
```

参数

- **HandleType** 要分配的句柄类型。UltraLite 支持以下句柄类型：
  - SQL_HANDLE_ENV
  - SQL_HANDLE_DBC
  - SQL_HANDLE_STMT
- **StatementHandle** 语句句柄。
- **TrueFalse** 输出句柄。

另请参见

- “在 UltraLite Palm 应用程序中维护状态（不建议使用）”一节第 68 页

SQLSynchronize 函数

对于 UltraLite ODBC，此函数用于使用 MobiLink 同步操作同步数据库中的数据。此函数是 UltraLite 特有的，不是 ODBC 标准的组成部分。

语法

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLSynchronize (
SQLHDBC ConnectionHandle,
ul_synch_info * SynchInfo );
```

参数

- **ConnectionHandle** 句柄。
- **SynchInfo** 用于存放同步信息的结构。请参见“UltraLite 同步参数和网络协议选项” 《UltraLite - 数据库管理和参考》。

注释

SQLSynchronize 是 ODBC 的扩展。它用于启动 MobiLink 同步操作。

另请参见

- “UltraLite 同步参数和网络协议选项” 《UltraLite - 数据库管理和参考》
- MobiLink - 服务器管理

教程：使用 C++ API 构建应用程序

目录

第 1 课：创建数据库并连接到数据库 **310**
第 2 课：将数据插入数据库 **314**
第 3 课：选择并列出表中的行 **315**
第 4 课：将同步添加到应用程序 **317**
教程的代码列表 **319**

本教程将指导您完成构建 UltraLite C++ 应用程序的全过程。此应用程序是为 Windows 桌面操作系统构建的，在命令提示符下运行。

虽然本教程以使用 Microsoft Visual C++ 的开发为基础，但您可以使用任何一种 C++ 开发环境。

如果您复制并粘贴代码，此教程需要大约 30 分钟时间。本教程的最后部分包含在本教程中介绍的程序的完整源代码。

能力和经验

本教程假定：

- 您熟悉 C++ 编程语言
- 您的计算机上安装了 C++ 编译器
- 您知道如何使用 [**创建数据库向导**] 创建 UltraLite 数据库。

  有关详细信息，请参见“使用 [创建数据库向导] 创建数据库”一节 《UltraLite - 数据库管理和参考》。

本教程的目标是掌握开发 UltraLite C++ 应用程序的过程。

第 1 课：创建数据库并连接到数据库

在第一个过程中，创建一个本地 UltraLite 数据库。然后编写、编译并运行一个访问创建的数据库的 C++ 应用程序。

♦ 创建 UltraLite 数据库

1. 将 *c:\vendor\visualstudio8\VC\atlmfc\src\atl* 添加到 INCLUDE 环境变量。
2. 创建一个目录，用于存放在本教程中要创建的文件。

   在本教程的剩余部分中，假定此目录是 *C:\tutorial\cpp*\。如果您创建具有不同名称的目录，则使用该目录代替 *C:\tutorial\cpp*\。
3. 在 Sybase Central 中使用 UltraLite，在新目录中创建一个名为 *ULCustomer.udb* 且具有以下特性的数据库。

   有关在 Sybase Central 中使用 UltraLite 的详细信息，请参见“使用 [创建数据库向导] 创建数据库”一节 《UltraLite - 数据库管理和参考》 。
4. 将名为 **ULCustomer** 的表添加到数据库。对 ULCustomer 表使用以下说明：

列名	数据类型（大小）	列是否允许 NULL 值？	缺省值	主键
cust_id	integer	否	自动增量	升序
cust_name	varchar(30)	否	无	

5. 在 Sybase Central 中断开与数据库的连接，否则可执行文件将无法进行连接。

♦ 连接到 UltraLite 数据库

1. 在 Microsoft Visual C++ 中，选择 [**File**] » [**New**]。
2. 在 [**Files**] 选项卡上，选择 [**C++ Source File**]。
3. 在教程目录中将文件另存为 *customer.cpp*。
4. 包括 UltraLite 库并使用 UltraLite 命名空间。

   将以下代码复制到 *customer.cpp*：

```
#include <tchar.h>
#include <stdio.h>
#include "uliface.h"
using namespace UltraLite;
#define MAX_NAME_LEN 100
ULSqlca Tutca;
```

   此代码段定义一个名为 Tutca 的 UltraLite SQL 通信区 (ULSqlca)。

   有关使用 UltraLite 命名空间来简化类声明的详细信息，请参见“使用 UltraLite 命名空间”一节 第 10 页。

5. 定义连接到数据库的连接参数。

   在此代码段中，连接参数是硬编码的。在实际的应用程序中，这些位置可在运行时指定。

   将以下代码复制到 *customer.cpp*。

```
static ul_char const * ConnectionParms =
  UL_TEXT( "UID=DBA;PWD=sql" )
  UL_TEXT( ";DBF=C:\\tutorial\\cpp\\ULCustomer.udb" );
```

   有关连接参数的详细信息，请参见“UltraLite_Connection_iface 类”一节第 148 页。

> **特殊字符处理**
> 文件名位置字符串中出现的反斜线字符必须使用前导反斜线字符进行转义。

6. 定义一个方法，用于处理应用程序中出现的数据库错误。

   UltraLite 提供通知应用程序出错的回调机制。在开发环境中，作为一种处理意外错误的机制，此函数很有用。生产应用程序通常包括用于处理所有常见错误情况的代码。应用程序可在每次调用 UltraLite 函数之后检查错误，也可以选择使用错误回调函数。

   下面是一个回调函数示例。

```
ul_error_action UL_GENNED_FN_MOD MyErrorCallBack(
 SQLCA *  Tutca,
 ul_void * user_data,
 ul_char * message_param )
{
    ul_error_action rc;

    (void) user_data;

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

        default:
            if (Tutca->sqlcode >= 0) { // warning or success
                rc = UL_ERROR_ACTION_DEFAULT;
            } else { // negative is real error
                _tprintf( _TEXT( "Error %ld: %s\n" ), Tutca->sqlcode,
message_param );
                rc = UL_ERROR_ACTION_CANCEL;
            }
        break;
     }
    return rc;
 }
```

   在 UltraLite 中，错误 SQLE_NOTFOUND 常用于控制应用程序流。发出此错误信号，标记结果集循环结束。上面编码的通用错误处理程序不为此错误情况输出消息。

   有关错误处理的详细信息，请参见“处理错误”一节第 24 页。

7. 定义打开到数据库的连接的方法。

   如果数据库文件不存在，则显示错误消息，否则建立连接。

```
Connection * open_conn( DatabaseManager * dm ) {
  Connection * conn = dm->OpenConnection( Tutca, ConnectionParms );

  if( conn == UL_NULL ) {
      _tprintf( _TEXT("Unable to open existing database.\n") );
   }
  return conn;
}
```

8. 实现 main 方法以执行以下任务：

   - 实例化 DatabaseManager 对象。所有 UltraLite 对象都创建自 DatabaseManager 对象。
   - 注册错误处理函数。
   - 打开与数据库的连接。
   - 关闭连接并关闭数据库管理器。

```
int main() {
  ul_char buffer[ MAX_NAME_LEN ];

  Connection * conn;

  Tutca.Initialize();

  ULRegisterErrorCallback(
    Tutca.GetCA(), MyErrorCallBack,
    UL_NULL, buffer, sizeof (buffer));

  DatabaseManager * dm = ULInitDatabaseManager( Tutca );

  conn = open_conn( dm );

  if( conn == UL_NULL ){
   dm->Shutdown( Tutca );
      Tutca.Finalize();
      return 1;
  }
  // main processing code to be inserted here
  dm->Shutdown( Tutca );
  Tutca.Finalize();
  return 0;
}
```

9. 编译并链接源文件。

   您编译源文件所用的方法取决于您的编译器。以下是对使用生成文件 (makefile) 的 Microsoft Visual C++ 命令行编译器的说明：

   - 打开命令提示符，然后更改至教程目录。
   - 创建一个名为 *makefile* 的生成文件。
   - 在生成文件中，将目录添加到包括路径。

     ```
     IncludeFolders=/I"$(SQLANY11)\SDK\Include"
     ```

   - 在生成文件中，将目录添加到库路径。

     ```
     LibraryFolders=/LIBPATH:"$(SQLANY11)\UltraLite\win32\386\Lib\vs8"
     ```

   - 在生成文件中，将库添加到链接器选项。

     ```
     Libraries=ulimp.lib
     ```


UltraLite 运行时库名为 *ulimp.lib*。

- 在生成文件中，设置编译器选项。您必须在一行上输入这些选项。

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

- 在生成文件中，添加链接应用程序的指令。

```
customer.exe: customer.obj
   link /NOLOGO /DEBUG customer.obj $(LibraryFolders) $(Libraries)
```

- 在生成文件中，添加编译应用程序的指令。

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

- 运行生成文件。

```
nmake
```

这将创建一个名为 *customer.exe* 的可执行文件。

10. 运行应用程序。

在命令提示符下，输入 **customer**。

第 2 课：将数据插入数据库

以下过程介绍如何向数据库添加数据。

♦ 向数据库添加行

1. 将以下过程添加到 *customer.cpp* 中紧靠 main 方法之前的地方：

```
bool do_insert( Connection * conn ) {
  Table * table = conn->OpenTable( UL_TEXT("ULCustomer") );
  if( table == UL_NULL ) {
      _tprintf( _TEXT("Table not found: ULCustomer\n") );
      return  false;
  }
  if( table->GetRowCount() == 0 ) {
      _tprintf( _TEXT("Inserting one row.\n") );
      table->InsertBegin();
      table->Set( UL_TEXT("cust_name"), UL_TEXT("New Customer") );
      table->Insert();
      conn->Commit();
  } else {
      _tprintf( _TEXT("The table has %lu rows\n"),
      table->GetRowCount() );
  }
  table->Release();
  return true;
}
```

   此过程执行以下任务。

   - 使用 connection->OpenTable() 方法打开表。必须打开 Table 对象才能操作表。
   - 如果表为空，向表添加一行。为插入行，使用 InsertBegin 方法将代码更改为插入模式，为每个必需的列设置值，并执行插入操作将此行添加到数据库。

     只有在关闭自动提交时才需要 commit 方法。缺省情况下，启用自动提交。但是为了获得更好的性能，或者为了进行多操作事务，可能会禁用自动提交。
   - 如果此表非空，则报告表中的行数。
   - 关闭 Table 对象，释放相关资源。
   - 返回一个布尔值，指示操作是否成功。

2. 调用已创建的 do_insert 方法。

   将如下行添加到 `main()` 方法中紧跟对 open_conn 的调用之后的地方。

```
do_insert(conn);
```

3. 通过运行 nmake 编译您的应用程序。
4. 通过在命令提示符下键入 **customer** 运行应用程序。

第 3 课：选择并列出表中的行

以下过程是从表中检索行，并将它们打印到命令行。

♦ 列出表中的行

1. 将以下方法添加到 *customer.cpp*。此方法执行以下任务：
   - 打开 Table 对象。
   - 检索列标识符。
   - 将当前位置设置在表的第一行之前。

     对表的任何操作都在当前位置执行。此位置可以在表的第一行之前、某一行中或最后一行之后。缺省情况下（如在本例中），行按照其主键值 (cust_id) 进行排序。要按其它方式排序行，可以向 UltraLite 数据库添加索引，然后使用此索引打开表。
   - 对于每一行，将写出 cust_id 和 cust_name 值。循环一直执行到 Next 方法返回 false（检索完最后一行后发生此情况）。
   - 关闭 Table 对象。

```
bool do_select( Connection * conn ) {
  Table * table = conn->OpenTable( _TEXT("ULCustomer") );
  if( table == UL_NULL ) {
    return false;
  }
  TableSchema * schema = table->GetSchema();
  if( schema == UL_NULL ) {
    table->Release();
    return false;
  }
  ul_column_num id_cid =
     schema->GetColumnID( UL_TEXT("cust_id") );
  ul_column_num cname_cid =
     schema->GetColumnID( UL_TEXT("cust_name") );

  schema->Release();

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

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

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

    _tprintf( _TEXT("id=%d, name=%s \n"), (int)table->Get(id_cid), cname);
  }
  table->Release();
  return true;
}
```

2. 将下行添加到 main 方法中紧跟对 insert 方法的调用之后的地方：

```
do_select(conn);
```

3. 通过运行 *nmake* 编译您的应用程序。

4. 通过在命令提示符下键入 *customer* 运行应用程序。

第 4 课：将同步添加到应用程序

本课介绍如何同步应用程序和计算机上运行的统一数据库。

以下过程将同步代码添加到应用程序，启动 MobiLink 服务器，然后运行应用程序进行同步。

在前几节课中创建的 UltraLite 数据库与 UltraLite 11 示例数据库同步。UltraLite 11 示例数据库中 ULCustomer 表的列包含本地 UltraLite 数据库中 customer 表的列。

本课假设您熟悉 MobiLink 同步。

♦ 将同步添加到应用程序

1. 将以下方法添加到 *customer.cpp*。此方法执行以下任务：
   - 通过调用 ULEnableTcpipSynchronization 将同步流设置为 TCP/IP。同步还可以通过 HTTP、HotSync 或 HTTPS 执行。请参见“UltraLite 客户端”《UltraLite - 数据库管理和参考》。
   - 设置脚本版本。MobiLink 同步由存储在统一数据库中的脚本控制。脚本版本确定要使用的脚本集。
   - 设置 MobiLink 用户名。此值用于 MobiLink 服务器处的验证。它与 UltraLite 数据库用户 ID 不同，尽管在某些应用程序中您可能希望赋予它们相同的值。
   - 将 download_only 参数设置为 true。缺省情况下，MobiLink 同步是双向的。此应用程序使用仅下载同步，因此表中的行不会上载到示例数据库中。

```
bool do_sync( Connection * conn ) {
  ul_synch_info info;
  ul_stream_error * se = &info.stream_error;

  ULEnableTcpipSynchronization( Tutca.GetCA() );
  conn->InitSynchInfo( &info );
  info.stream = ULSocketStream();
  info.version = UL_TEXT( "custdb 11.0" );
  info.user_name = UL_TEXT( "50" );
  info.download_only = true;
  if( !conn->Synchronize( &info ) ) {
      _tprintf( _TEXT("Synchronization error \n" ));
    _tprintf( _TEXT("  stream_error_code is '%lu'\n"), se-
>stream_error_code );
    _tprintf( _TEXT("  system_error_code is '%ld'\n"), se-
>system_error_code );
    _tprintf( _TEXT("  error_string is '") );
    _tprintf( _TEXT("%s"), se->error_string );
    _tprintf( _TEXT("'\n") );
    return false;
  }
  return true;
}
```

2. 将下行添加到 main 方法中，放在调用 insert 方法的语句和调用 select 方法的语句之间：

```
do_sync(conn);
```

3. 通过运行 *nmake* 编译您的应用程序。

♦ **同步数据**

1. 启动 MobiLink 服务器。

   在命令提示符下，运行以下命令：

   ```
   mlsrv11 -c "dsn=SQL Anywhere 11 CustDB" -v+ -zu+
   ```

   -zu+ 选项提供自动添加用户的功能。-v+ 选项为所有消息启用详细记录。

   有关此选项的详细信息，请参见“MobiLink 服务器选项”《MobiLink - 服务器管理》。

2. 通过在命令提示符下键入 *customer* 运行应用程序。

   MobiLink 服务器窗口显示指示同步进度的状态消息。如果同步成功，则最后一条消息显示 [Synchronization complete]。

教程的代码列表

以下是前几节中介绍的教程程序的完整代码。

```
#include <tchar.h>
#include <stdio.h>
#include "uliface.h"
using namespace UltraLite;
#define MAX_NAME_LEN 100
ULSqlca Tutca;

static ul_char const * ConnectionParms =
  UL_TEXT( "UID=DBA;PWD=sql;" )
  UL_TEXT( "DBF=C:\\temp\\ULCustomer.udb" );

ul_error_action UL_GENNED_FN_MOD MyErrorCallBack(
 SQLCA *  Tutca,
 ul_void * user_data,
 ul_char * message_param )
{
    ul_error_action rc;

    (void) user_data;

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

        default:
            if (Tutca->sqlcode >= 0) { // warning or success
                rc = UL_ERROR_ACTION_DEFAULT;
            } else { // negative is real error
                _tprintf( _TEXT( "Error %ld: %s\n" ), Tutca->sqlcode,
message_param );
                rc = UL_ERROR_ACTION_CANCEL;
            }
        break;
     }
    return rc;
 }
Connection * open_conn( DatabaseManager * dm ) {
  Connection * conn = dm->OpenConnection( Tutca, ConnectionParms );
  if( conn == UL_NULL ) {
      _tprintf( _TEXT("Unable to open existing database.\n") );
   }
  return conn;
}
// Open table, insert 1 row if table is currently empty

bool do_insert( Connection * conn ) {
  Table * table = conn->OpenTable( UL_TEXT("ULCustomer") );
  if( table == UL_NULL ) {
      _tprintf( _TEXT("Table not found: ULCustomer\n") );
      return  false;
  }
  if( table->GetRowCount() == 0 ) {
      _tprintf( _TEXT("Inserting one row.\n") );
      table->InsertBegin();
      table->Set( UL_TEXT("cust_name"), UL_TEXT("New Customer") );
      table->Insert();
```

```
        conn->Commit();

    } else {
        _tprintf( _TEXT("The table has %lu rows\n"),
        table->GetRowCount() );
    }
    table->Release();
    return true;
}// Open table, display data from all rows

bool do_select( Connection * conn ) {
    Table * table = conn->OpenTable( _TEXT("ULCustomer") );
    if( table == UL_NULL ) {
    return false;
    }
    TableSchema * schema = table->GetSchema();
    if( schema == UL_NULL ) {
      table->Release();
      return false;
    }
    ul_column_num id_cid = schema->GetColumnID( UL_TEXT("cust_id") );
    ul_column_num cname_cid = schema->GetColumnID( UL_TEXT("cust_name") );

    schema->Release();

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

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

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

      _tprintf( _TEXT("id=%d, name=%s \n"), (int)table->Get( id_cid ), cname );
    }
    table->Release();
    return true;
}
// sync database with MobiLink connection to reference database

bool do_sync( Connection * conn ) {
    ul_synch_info info;
    ul_stream_error * se = &info.stream_error;

    ULEnableTcpipSynchronization( Tutca.GetCA() );
    conn->InitSynchInfo( &info );
    info.stream = ULSocketStream();
    info.version = UL_TEXT( "custdb 11.0" );
    info.user_name = UL_TEXT( "50" );
    info.download_only = true;
    if( !conn->Synchronize( &info ) ) {
        _tprintf( _TEXT("Synchronization error \n" ));
      _tprintf( _TEXT("  stream_error_code is '%lu'\n"), se-
>stream_error_code );
      _tprintf( _TEXT("  system_error_code is '%ld'\n"), se-
>system_error_code );
      _tprintf( _TEXT("  error_string is '") );
      _tprintf( _TEXT("%s"), se->error_string );
      _tprintf( _TEXT("'\n") );
      return false;
    }
    return true;
}
```



```
int main() {
  ul_char buffer[ MAX_NAME_LEN ];

  Connection * conn;

  Tutca.Initialize();

  ULRegisterErrorCallback(
    Tutca.GetCA(), MyErrorCallBack,
    UL_NULL, buffer, sizeof (buffer));

  DatabaseManager * dm = ULInitDatabaseManager( Tutca );

  if( dm == UL_NULL ){
    //  You  may  have  mismatched  UNICODE  vs.  ANSI  runtimes.
 Tutca.Finalize();
    return 1;
  }

  conn = open_conn( dm );

  if( conn == UL_NULL ){
   dm->Shutdown( Tutca );
      Tutca.Finalize();
      return 1;
  }

  do_insert (conn);
  do_sync (conn);
  do_select (conn);


  dm->Shutdown( Tutca );
  Tutca.Finalize();
  return 0;
}
```

术语表

术语表 325

术语表

Adaptive Server Anywhere (ASA)

SQL Anywhere Studio 的关系数据库服务器组件，专供在移动和嵌入式环境中使用，或作为中小型企业的服务器使用。在版本 10.0.0 中，Adaptive Server Anywhere 更名为 SQL Anywhere 服务器，SQL Anywhere Studio 更名为 SQL Anywhere。

另请参见： “SQL Anywhere”一节第 342 页

包

Java 中相关类的集合。

被引用对象

一种对象（如表），该对象在另一个对象（如视图）的定义中被直接引用。

另请参见： “主键”一节第 352 页

编码

也称作字符编码，编码是一种方法，通过该方法可以将字符集中的每个字符映射到一个或多个字节的信息，这些信息通常以十六进制数字表示。编码的一个例子是 UTF-8。

另请参见：

- “字符集”一节第 352 页
- “代码页”一节第 327 页
- “归类”一节第 331 页

标识符

用于引用数据库对象（如表或列）的字符串。标识符可以包含 A 到 Z、a 到 z、0 到 9、下划线 (_)、at 符号 (@)、数字符号 (#) 或美元符号 ($) 中的任何字符。

并发

同时执行两个或更多个独立并且可能存在竞争关系的进程。SQL Anywhere 会自动使用锁定来隔离事务，并确保每个并发应用程序看到的数据集均一致。

另请参见：

- “事务”一节第 339 页
- “隔离级别”一节第 330 页

参考数据库

MobiLink 中一种用于 UltraLite 客户端开发的 SQL Anywhere 数据库。在开发过程中，可以将一个 SQL Anywhere 数据库同时作为参考数据库和统一数据库使用。通过其它产品建立的数据库无法用作参考数据库。

参照完整性

遵守数据一致性控制规则（具体而言，不同表中主键值与外键值之间的关系）。若要实现参照完整性，每个外键中的值必须与被引用表中行的主键值相符。

另请参见：

- “主键”一节第 352 页
- “外键”一节第 344 页

策略

QAnywhere 中指定应在何时进行消息传输的方式。

插件模块

Sybase Central 中一种用于访问和管理产品的方法。当您安装相应的产品时，插件通常会自动安装并注册 Sybase Central。通常，插件在 Sybase Central 主窗口中作为顶级容器出现，并且使用产品本身的名称，如 SQL Anywhere。

另请参见： “Sybase Central”一节第 343 页

查询

一条或一组 SQL 语句，用于访问和/或操作数据库中的数据。

另请参见： “SQL”一节第 342 页

冲突解决

在 MobiLink 中，冲突解决是指一种逻辑，它指定当两个用户修改不同远程数据库上同一行时的处理方法。

重定向器

一种 Web 服务器插件，用于为客户端与 MobiLink 服务器之间的请求和响应选择发送路径。此插件还实现了负荷平衡和故障转移机制。

抽取

SQL Remote 复制中从统一数据库卸载相应结构和数据的行为。此信息用于初始化远程数据库。

另请参见： “复制”一节第 329 页

触发器

一种特殊形式的存储过程，用户运行修改数据的查询时会自动执行该存储过程。

另请参见：

- “行级触发器”一节第 331 页
- “语句级触发器”一节第 349 页
- “完整性”一节第 345 页

传输规则

QAnywhere 中用于确定何时进行消息传输、传输哪些消息以及应在何时删除消息的逻辑。

窗口

作为分析功能执行对象的行组。一个窗口可以包含一行、多行或所有行的数据，这些数据已根据窗口定义中提供的分组规格进行了分区。窗口会进行移动，以包括为输入中的当前行执行计算所需的行数或行范围。窗口结构的主要优点是，不需要执行附加查询就可以有机会对结果进行分组和分析。

创建者 ID

UltraLite Palm OS 应用程序中一种在创建应用程序时指派的 ID。

存储过程

存储过程是数据库中存储的一组 SQL 指令，用于在数据库服务器上执行一组操作或查询。

代理表

一种本地表，它所包含的元数据可以像访问本地表一样访问远程数据库服务器上的表。

另请参见： “元数据”一节第 350 页

代理 ID

另请参见： “客户端消息存储库 ID”一节第 335 页

代码页

代码页是一种将字符集的字符映射到数字表示的编码，数字表示通常是 0 到 255 之间的一个整数。例如，Windows 代码页 1252 就是一个代码页。就本文档而言，代码页和编码这两个术语可以互换。

另请参见：

- “字符集”一节第 352 页
- “编码”一节第 325 页
- “归类”一节第 331 页

DBA 权限

使用户能够在数据库中执行管理活动的权限级别。DBA 用户在缺省情况下具有 DBA 权限。

另请参见： “数据库管理员 (DBA)”一节第 341 页

dbspace

用于创建更多数据存储空间的附加数据库文件。一个数据库可以包含在最多 13 个独立的文件（一个初始文件和 12 个 dbspace）中。每个表及其索引必须包含在单个数据库文件中。SQL 命令 CREATE DBSPACE 可将新文件添加到数据库中。

另请参见： “数据库文件”一节第 342 页

动态 SQL

执行前由程序以编程方式生成的 SQL。UltraLite 动态 SQL 是一种专用于小型设备的 SQL 变体。

对象树

Sybase Central 中数据库对象的层次。对象树的顶层显示您的 Sybase Central 版本所支持的全部产品。每种产品展开后会显示其自己的对象子树。

另请参见： “Sybase Central”一节第 343 页

EBF

快速错误修正软件。快速错误修正软件是含有一个或多个错误修正软件的软件子集。错误修正软件列在更新程序的发行说明中。错误修正软件更新可能只适用于具有相同版本号的已安装软件。已对该软件执行了一些测试，但该软件尚未进行完全测试。除非您自己已验证了软件的适用性，否则不要随应用程序分发这些文件。

发布

MobiLink 或 SQL Remote 中一种用于标识将要同步的数据的数据库对象。在 MobiLink 中，发布仅存在于客户端。一个发布包括多个项目。SQL Remote 用户可以通过预订发布来接收发布。MobiLink 用户可以通过创建发布的同步预订来同步发布。

另请参见：

- “复制”一节第 329 页
- “项目”一节第 347 页
- “发布更新”一节第 328 页

发布更新

SQL Remote 复制中对一个数据库中的一个或多个发布所做更改的列表。发布更新将作为复制消息的一部分定期发送到远程数据库。

另请参见：

- “复制”一节第 329 页
- “发布”一节第 328 页

发布者

SQL Remote 复制中数据库内可以与其它复制数据库交换复制消息的单个用户。

另请参见： “复制”一节第 329 页。

FILE

SQL Remote 复制中一种使用共享文件来交换复制消息的消息系统。它对测试以及在没有显式消息传送系统的情况下进行的安装很有用。

另请参见“复制”一节第 329 页。

分析树

查询的代数表示。

服务

在 Windows 操作系统上，服务是在运行应用程序的用户 ID 未登录时的应用程序运行方式。

服务器管理请求

一种 QAnywhere 消息，其格式设置为 XML 并发送到 QAnywhere 系统队列，作为一种管理服务器消息存储库或监控 QAnywhere 应用程序的方法。

服务器启动的同步

一种从 MobiLink 服务器启动 MobiLink 同步的方式。

服务器消息存储库

QAnywhere 中在消息传输到客户端消息存储库或 JMS 系统之前服务器上用于临时存储消息的关系数据库。消息通过服务器消息存储库在各客户端之间进行交换。

复制

在物理上不相同的数据库之间共享数据。Sybase 有三种复制技术：MobiLink、SQL Remote 和复制服务器。

复制代理

请参见： “LTM”一节第 336 页

复制服务器

Sybase 的一种基于连接的复制技术，用于与 SQL Anywhere 和 Adaptive Server Enterprise 一起使用。它专用于在一些数据库之间进行接近实时的复制。

另请参见： “LTM”一节第 336 页

复制频率

SQL Remote 复制中一项针对每个远程用户的设置，它决定发布者的消息代理向该远程用户发送复制消息的频率应为多少。

另请参见： “复制”一节第 329 页。

复制消息

SQL Remote 或复制服务器中一种在发布数据库与预订数据库之间发送的通信。消息包含复制系统所需的数据、直通语句及信息。

另请参见：

- “复制”一节第 329 页
- “发布更新”一节第 328 页

隔离级别

一个事务中的操作对其它并发事务中的操作的可见程度。隔离级别有四级，编号依次为 0 至 3。第 3 级提供最高级别的隔离。级别 0 为缺省设置。SQL Anywhere 还支持以下三个快照隔离级别：快照、语句快照和只读语句快照。

另请参见： “快照隔离”一节第 335 页

个人服务器

与客户端应用程序在同一台计算机上运行的数据库服务器。个人数据库服务器通常由单个用户在一台计算机上使用，但它可以支持来自该用户的几个并发连接。

工作表

一种内部存储区域，用于在查询优化过程中存储中间结果。

故障切换

在活动服务器、系统或网络出现故障或意外终止时切换到冗余或备用的服务器、系统或网络。故障转移会自动进行。

关系数据库管理系统 (RDBMS)

一种以相关表的形式存储数据的数据库管理系统。

另请参见： “数据库管理系统 (DBMS)”一节第 341 页

规范化

对数据库模式的改进，目的在于按照基于关系数据库理论的规则消除冗余并改善组织。

归类

定义数据库中文本属性的字符集与排序顺序的组合。对于 SQL Anywhere 数据库，缺省归类取决于运行服务器时所使用的操作系统和语言；例如，英语 Windows 系统上的缺省归类为 1252LATIN1。归类（也称作归类序列）用于对字符串进行比较和排序。

另请参见：

- “字符集”一节第 352 页
- “代码页”一节第 327 页
- “编码”一节第 325 页

行级触发器

每更改一行即执行一次的触发器。

另请参见：

- “触发器”一节第 327 页
- “语句级触发器”一节第 349 页

回退日志

对在每个未提交的事务执行过程中所做更改的记录。当收到 ROLLBACK 请求或者系统出现故障时，未提交的事务会从数据库中回退，将数据库返回其原先的状态。每个事务都有一个单独的回退日志，事务完成时日志会被删除。

另请参见： “事务”一节第 339 页

iAnywhere JDBC 驱动程序

iAnywhere JDBC 驱动程序提供了一个 JDBC 驱动程序，与纯 Java jConnect JDBC 驱动程序相比，该驱动程序拥有一些性能优势和功能优点，但它不是纯 Java 解决方案。建议在大多数情况下使用 iAnywhere JDBC 驱动程序。

另请参见：

- “JDBC”一节第 332 页
- “jConnect”一节第 332 页

InfoMaker

一种报告和数据维护工具，它用于创建复杂的表格、报告、图形、交叉表和表，并创建将这些报告用作构件块的应用程序。

Interactive SQL

一种 SQL Anywhere 应用程序，用于查询和更改数据库中的数据以及修改数据库的结构。Interactive SQL 不但提供了一个用于输入 SQL 语句的窗格，还提供了一些用于返回有关查询处理过程的信息和结果集的窗格。

JAR 文件

Java 档案文件。一种压缩的文件格式，由一个或多个用于 Java 应用程序的包的集合组成。它将安装和运行 Java 程序所需的全部资源都放在一个压缩文件中。

Java 类

Java 中的主要代码结构单元。它是组合在一起的过程和变量的集合，将过程和变量组合在一起的原因是它们都与某个特定的可识别类别有关。

jConnect

JavaSoft JDBC 标准的 Java 实现。它为 Java 开发人员提供多层和异类环境中的本地数据库访问。但在大多数情况下，iAnywhere JDBC 驱动程序是首选的 JDBC 驱动程序。

另请参见：

- “JDBC”一节第 332 页
- “iAnywhere JDBC 驱动程序”一节第 331 页

JDBC

Java 数据库连接。一种 SQL 语言编程接口，它允许 Java 应用程序访问关系数据。首选的 JDBC 驱动程序是 iAnywhere JDBC 驱动程序。

另请参见：

- “jConnect”一节第 332 页
- “iAnywhere JDBC 驱动程序”一节第 331 页

基表

永久性的数据表。有时为区别于临时表和视图，会将这种表称作**基表**。

另请参见：

- “临时表”一节第 335 页
- “视图”一节第 339 页

基于会话的同步

一种同步类型，这种同步会使数据表示在统一数据库和远程数据库都一致。MobiLink 基于会话。

基于脚本的上载

MobiLink 中一种将上载过程自定义为使用日志文件的替代方法的方式。

基于 SQL 的同步

MobiLink 中一种使用 MobiLink 事件将表数据与支持 MobiLink 的统一数据库进行同步的方式。对于基于 SQL 的同步，可以直接使用 SQL，也可以使用面向 Java 和 .NET 平台的 MobiLink 服务器 API 返回 SQL。

基于文件的下载

在 MobiLink 中同步数据的一种方式，其中下载以文件的方式进行分发，从而支持脱机分发同步更改。

集成登录

一种登录功能，它允许将同一个用户 ID 和口令用于操作系统登录、网络登录和数据库连接。

监听器

一个程序 (dblsn)，用于 MobiLink 服务器启动的同步。监听器安装在远程设备上，它们被配置为在接收到来自通告程序的信息时启动针对设备的操作。

另请参见： “服务器启动的同步”一节第 329 页

检查点

将对数据库的所有更改都保存到数据库文件中的时间点。在其它时间，所提交的更改仅保存到事务日志中。

检查约束

对列或列集强制实施指定条件的一种限制。

另请参见：

- “约束”一节第 351 页
- “外键约束”一节第 345 页
- “主键约束”一节第 352 页
- “唯一约束”一节第 346 页

脚本

MobiLink 中为处理 MobiLink 事件而编写的代码。脚本通过编程方式控制数据交换，以满足业务需要。

另请参见： “事件模型”一节第 339 页

脚本版本

MobiLink 中为创建同步而一起应用的一组同步脚本。

校验

测试数据库、表或索引是否受到特定类型的文件损坏。

校验和

随数据库页本身一起记录的计算出的数据库页位数。校验和能够确保数据库页写入磁盘时位数相符，因此数据库管理系统可以通过它来验证数据库页的完整性。如果计数相符，即认为数据库页已成功写入。

镜像日志

另请参见： “事务日志镜像”一节第 340 页

角色

概念性数据库建模中从一个角度描述某种关系的动词或短语。您可以用两个角色来描述每种关系。例如，"包含" 和 "隶属于" 便是角色。

角色名

外键的名称。由于它命名外表和主表之间的关系，因此称作角色名。缺省情况下，角色名就是表名，除非其它外键已经使用该名称（在这种情况下，缺省的角色名是表名后接一个三位的唯一数字）。也可以自己创建角色名。

另请参见： “外键”一节第 344 页

局部临时表

一种临时表，仅在复合语句执行期间或连接结束之前存在。当您只需要将数据集装载一次时，局部临时表非常有用。缺省情况下，行会在提交时被删除。

另请参见：

- “临时表”一节第 335 页
- “全局临时表”一节第 338 页

客户端/服务器

一种软件体系结构，在这种体系结构中，一个应用程序（客户端）从另一个应用程序（服务器）获取信息并向该应用程序发送信息。这两个应用程序常位于通过网络连接的不同计算机上。

客户端消息存储库

QAnywhere 中一种用于在远程设备上存储消息的 SQL Anywhere 数据库。

客户端消息存储库 ID

QAnywhere 中一种对客户端消息存储库进行唯一标识的 MobiLink 远程 ID。

快照隔离

一种为发出读请求的事务返回数据的已提交版本的隔离级别。SQL Anywhere 提供了以下三种快照隔离级别：快照、语句快照和只读语句快照。使用快照隔离时，读操作不会阻塞写操作。

另请参见： “隔离级别”一节第 330 页

连接

关系系统中的一种基本操作，它通过比较指定列中的值将两个或更多个表中的行链接在一起。

连接 ID

用于标识客户端应用程序与数据库之间给定连接的唯一编号。可以使用以下 SQL 语句来确定当前连接 ID：

```
SELECT CONNECTION_PROPERTY( 'Number' );
```

连接类型

SQL Anywhere 提供了四种类型的连接：交叉连接、键连接、自然连接和使用 ON 子句的连接。

另请参见： “连接”一节第 335 页

连接配置

连接到数据库所需的一组参数，如用户名、口令和服务器名称，它们在存储后即可方便地使用。

连接启动的同步

一种 MobiLink 服务器启动的同步，在这种同步下，连接发生变化时会启动同步。

另请参见： “服务器启动的同步”一节第 329 页

连接条件

一种影响连接结果的限制。您可以通过紧跟在连接语句的后面插入 ON 子句或 WHERE 子句来指定连接条件。对于自然连接和关键连接，SQL Anywhere 会生成连接条件。

另请参见：

- “连接”一节第 335 页
- “生成的连接条件”一节第 340 页

临时表

为临时存储数据而创建的表。有两种类型：全局临时表和局部临时表。

另请参见：

- “局部临时表”一节第 334 页
- “全局临时表”一节第 338 页

LTM

日志传送管理器（Log Transfer Manager，简称 LTM）也称作复制代理。LTM 是一个与 Replication Server 一起使用的程序，它读取数据库事务日志并将提交的更改发送到 Sybase 复制服务器。

请参见： “复制服务器”一节第 330 页

轮询

在 MobiLink 服务器启动的同步中，轻量级轮询器（例如 MobiLink 监听器）从通告程序请求推式通知的方式。

另请参见： “服务器启动的同步”一节第 329 页

逻辑索引

指向物理索引的引用（指针）。磁盘上不存储逻辑索引的索引结构。

命令文件

包含 SQL 语句的文本文件。命令文件可以手工建立，也可以通过数据库实用程序自动建立。例如，dbunload 实用程序会创建一个命令文件，其中包含重新创建给定数据库所需的 SQL 语句。

MobiLink

一种基于会话的同步技术，其设计用途是将 UltraLite 和 SQL Anywhere 远程数据库与统一数据库同步。

另请参见：

- “统一数据库”一节第 343 页
- “同步”一节第 343 页
- “UltraLite”一节第 344 页

MobiLink 服务器

运行 MobiLink 同步的计算机程序，即 mlsrv11。

MobiLink 监控器

一种用于监控 MobiLink 同步的图形化工具。

MobiLink 客户端

有两种 MobiLink 客户端。对于 SQL Anywhere 远程数据库，MobiLink 客户端是 dbmlsync 命令行实用程序。对于 UltraLite 远程数据库，MobiLink 客户端内置于 UltraLite 运行时库中。

MobiLink 系统表

MobiLink 同步所需的系统表。它们由 MobiLink 安装程序脚本安装到 MobiLink 统一数据库中。

MobiLink 用户

MobiLink 用户用于与 MobiLink 服务器进行连接。在远程数据库上创建 MobiLink 用户，然后在统一数据库中注册该用户。MobiLink 用户名完全独立于数据库用户名。

模式

数据库的结构，其中包括表、列和索引以及它们之间的关系。

内连接

一种连接，在这种连接中，仅当两个表都满足连接条件时行才会出现在结果集中。内连接是缺省设置。

另请参见：

- “连接”一节第 335 页
- “外连接”一节第 345 页

ODBC

开放式数据库连接。一种用于与数据库管理系统连接的标准 Windows 接口。ODBC 是 SQL Anywhere 所支持的几种接口之一。

ODBC 管理器

一种随 Windows 操作系统提供的 Microsoft 程序，用于设置 ODBC 数据源。

ODBC 数据源

用户要通过 ODBC 访问的数据的规范以及获取该数据时所需的信息。

PDB

Palm 数据库文件。

PowerDesigner

一种数据库建模应用程序。PowerDesigner 为设计数据库或数据仓库提供了结构化的方法。SQL Anywhere 包括 PowerDesigner 的 Physical Data Model 组件。

PowerJ

一种 Sybase 产品，用于开发 Java 应用程序。

QAnywhere

应用程序到应用程序的消息传递（包括移动设备到移动设备和移动设备与企业之间的消息传递），它使在移动或无线设备上运行的自定义程序能够与处在中央位置的服务器应用程序进行通信。

QAnywhere 代理

QAnywhere 中一种运行在客户端设备上的进程，用于监控客户端消息存储库和确定应在何时传输消息。

嵌入式 SQL

一种 C 语言程序编程接口。SQL Anywhere 嵌入式 SQL 是 ANSI 和 IBM 标准的实现。

轻量级轮询器

在 MobiLink 服务器启动的同步中，轮询来自 MobiLink 服务器的推式通知的设备应用程序。

另请参见： “服务器启动的同步”一节第 329 页

全局临时表

一种临时表，在被显式地删除之前，其数据定义对所有用户都可见。全局临时表允许用户各自打开一个表的相同实例。缺省情况下，行在提交时被删除，并且始终是在连接结束时被删除。

另请参见：

- “临时表”一节第 335 页
- “局部临时表”一节第 334 页

日志文件

SQL Anywhere 所维护的事务日志。该日志文件用于确保在出现系统或介质故障时可以恢复数据库、提高数据库性能以及使用 SQL Remote 实现数据复制。

另请参见：

- “事务日志”一节第 339 页
- “事务日志镜像”一节第 340 页
- “完全备份”一节第 345 页

散列

散列是一种将索引条目转化为键的索引优化。索引散列旨在通过将足够的行实际数据与其行 ID 包括在一起，以避免进行先查找行、后装载行然后再将行解出才能得出索引值的高开销操作。

上载

同步过程的一个阶段，在此阶段数据从远程数据库传送到统一数据库。

设备跟踪

在 MobiLink 服务器启动的同步中，允许使用标识设备的 MobiLink 用户名来对消息进行寻址的功能。

另请参见： “服务器启动的同步”一节第 329 页

实例化视图

实例化视图是指已计算并已存储在磁盘上的视图。实例化视图同时具有视图的特征（使用查询说明进行定义）和表的特征（可以对其执行大多数表操作）。

另请参见：

- “基表”一节第 332 页
- “视图”一节第 339 页

世代号

MobiLink 中的一种机制，用于强制远程数据库先上载数据，然后再应用任何其它下载文件。

另请参见： “基于文件的下载”一节第 333 页

事件模型

MobiLink 中组成同步的事件（如 begin_synchronization 和 download_cursor）序列。如果为事件创建了脚本，则会调用事件。

视图

一种作为对象存储在数据库中的 SELECT 语句。它使用户能够看到一个或多个表中的行子集或列子集。每当用户使用特定表或表组合的视图时，都将利用存储在这些表中的信息重新计算视图。视图对确保安全以及定制数据库信息的外观来使数据访问简单明了有帮助。

事务

组成一个逻辑工作单元的 SQL 语句序列。事务要么全部得到处理，要么根本不做处理。SQL Anywhere 支持事务处理，并内置了锁定功能，使并发事务能够访问数据库而又不损坏数据。事务要么以 COMMIT 语句结束，该语句使对数据的更改成为永久性更改；要么以 ROLLBACK 语句结束，该语句撤消在事务执行过程中所做的全部更改。

事务日志

一种按进行更改的顺序存储对数据库所做全部更改的文件。它会提高性能并支持在数据库文件损坏时恢复数据。

事务日志镜像

同时维护的事务日志文件的完全相同副本（可选）。每当数据库更改写入事务日志文件时，也会同时写入事务日志镜像文件。

镜像文件应该与事务日志保留在不同的设备上，这样在任意设备出现故障时，日志的其它副本会确保数据可以安全地恢复。

另请参见： “事务日志”一节第 339 页

事务完整性

MobiLink 中对整个同步系统事务的有保证维护。要么同步整个事务，要么不对事务的任何部分进行同步。

生成的连接条件

一种自动生成的对连接结果的限制。有两种类型：关键和自然。指定 KEY JOIN 或指定关键字 JOIN 但不使用关键字 CROSS、NATURAL 或 ON 时，会生成关键连接。对于关键连接，所生成的连接条件取决于表之间的外键关系。指定 NATURAL JOIN 时会生成自然连接；所生成的连接条件基于两个表中的公用列名。

另请参见：

- “连接”一节第 335 页
- “连接条件”一节第 335 页

受保护的功能

数据库服务器启动时由 -sf 选项指定的功能，该数据库服务器上运行的任何数据库都无法使用该功能。

授权选项

一种权限级别，它允许用户向其他用户授予权限。

数据操作语言 (DML)

用于操作数据库中数据的 SQL 语句子集。DML 语句可以检索、插入、更新和删除数据库中的数据。

数据定义语言 (DDL)

用于定义数据库中数据结构的 SQL 语句子集。DDL 语句可以创建、修改和删除数据库对象（如表和用户）。

数据类型

数据的格式，如 CHAR 或 NUMERIC。在 ANSI SQL 标准中，数据类型也可以包括对大小、字符集和归类的限制。

另请参见： “域”一节第 349 页

数据立方体

一种多维结果集，每一维都以不同的方式对相同的结果进行分组和排序。数据立方体提供了有关数据的综合性信息，如果不使用数据立方体，要获得同样的信息就必须进行自连接查询和相关子查询。数据立方体是 OLAP 功能的一部分。

数据库

通过主键和外键关联的表的集合。表包含数据库中的信息。表和键一起定义数据库的结构。数据库管理系统会访问此信息。

另请参见：

- “外键”一节第 344 页
- “主键”一节第 352 页
- “数据库管理系统 (DBMS)”一节第 341 页
- “关系数据库管理系统 (RDBMS)”一节第 330 页

数据库对象

包含或接收信息的数据库组件。表、索引、视图、过程和触发器便是数据库对象。

数据库服务器

对所有针对数据库信息的访问进行管理的计算机程序。SQL Anywhere 提供了两种类型的服务器：网络服务器和个人服务器。

数据库管理系统 (DBMS)

用于创建和使用数据库的程序的集合。

另请参见： “关系数据库管理系统 (RDBMS)”一节第 330 页

数据库管理员 (DBA)

具有维护数据库所需权限的用户。DBA 通常负责对数据库模式的所有更改以及管理用户和组。数据库管理员角色自动内置于数据库中，其用户 ID 为 DBA，口令是 sql。

数据库连接

客户端应用程序与数据库之间的通信渠道。必须具有有效的用户 ID 和口令才能建立连接。为用户 ID 授予的特权决定了在连接过程中可以执行的操作。

数据库名称

服务器装载数据库时为数据库指定的名称。缺省数据库名是初始数据库文件的文件名（不含扩展名）。

另请参见： “数据库文件”一节第 342 页

数据库所有者 (dbo)

一种特殊的用户，他拥有不归 SYS 所有的系统对象。

另请参见：

- “数据库管理员 (DBA)”一节第 341 页
- “SYS”一节第 343 页

数据库文件

数据库保存在一个或多个数据库文件中。其中一个是初始文件，后面的文件称作 dbspace。每个表（包括其索引）都必须包含在单个数据库文件中。

另请参见： “dbspace”一节第 328 页

死锁

一组事务会进入的一种特殊状态，在该状态下这些事务都不能继续执行。

SQL

用于与关系数据库进行通信的语言。ANSI 定义了 SQL 的标准，其最新标准是 SQL-2003。SQL 的非官方全称是结构化查询语言。

SQL Anywhere

SQL Anywhere 的关系数据库服务器组件，专供在移动和嵌入式环境中使用，或作为中小型企业的服务器使用。SQL Anywhere 也是包含 SQL Anywhere RDBMS、UltraLite RDBMS、MobiLink 同步软件和其它组件的软件包的名称。

SQL Remote

一种基于消息的数据复制技术，用于在统一数据库与远程数据库之间进行双向复制。统一数据库和远程数据库必须是 SQL Anywhere。

SQL 语句

包含用于将指令传递给 DBMS 的 SQL 关键字的字符串。

另请参见：

- “模式”一节第 337 页
- “SQL”一节第 342 页
- “数据库管理系统 (DBMS)”一节第 341 页

锁定

一种在同时执行多个事务的过程中保护数据完整性的并发控制机制。SQL Anywhere 会自动应用锁以防止两个连接同时更改同一数据，并防止其它连接读取正接受更改的数据。

您可以通过设置隔离级别来控制锁定。

另请参见：

- “隔离级别”一节第 330 页
- “并发”一节第 325 页
- “完整性”一节第 345 页

索引

一组已排序的、与基表中的一个或多个列关联的键和指针。在表中一个或多个列上设置索引可以提高性能。

Sybase Central

一种数据库管理工具，通过图形用户界面提供 SQL Anywhere 数据库设置、属性和实用程序。Sybase Central 也可用于管理其它 Sybase 产品，其中包括 MobiLink。

SYS

一种拥有大多数系统对象的特殊用户。无法以 SYS 身份登录。

统一数据库

在分布式数据库环境中，是指用于存储数据主副本的数据库。出现冲突或差异时，将把统一数据库视为具有数据的主副本。

另请参见：

- “同步”一节第 343 页
- “复制”一节第 329 页

通信流

MobiLink 中 MobiLink 客户端与 MobiLink 服务器之间进行通信时所使用的网络协议。

通告程序

一种由 MobiLink 服务器启动的同步使用的程序。通告程序集成在 MobiLink 服务器中。它们会检查统一数据库是否有推式请求，并发送推式通知。

另请参见：

- “服务器启动的同步”一节第 329 页
- “监听器”一节第 333 页

同步

利用 MobiLink 技术在数据库之间复制数据的过程。

在 SQL Remote 中，同步专指以初始数据集初始化远程数据库的过程。

另请参见：

- “MobiLink”一节第 336 页
- “SQL Remote”一节第 342 页

推式请求

在 MobiLink 服务器启动的同步中，通告程序通过检查它来确定推式通知是否需要发送到设备的结果集中的一行值。

另请参见： “服务器启动的同步”一节第 329 页

推式通知

QAnywhere 中一种从服务器传送到 QAnywhere 客户端的特殊消息，用于提示客户端启动消息传输。在 MobiLink 服务器启动的同步中，从通告程序传送到包含推式请求数据和内部信息的设备的特殊消息。

另请参见：

- “QAnywhere”一节第 338 页
- “服务器启动的同步”一节第 329 页

UltraLite

一种针对小型设备、移动设备和嵌入式设备进行了优化的数据库。所面向的平台包括手机、传呼机和个人记事本。

UltraLite 运行时

一种过程中关系数据库管理系统，其中包括一个内置 MobiLink 同步客户端。每个 UltraLite 编程接口使用的库以及 UltraLite 引擎中都包括 UltraLite 运行时。

外表

包含外键的表。

另请参见： “外键”一节第 344 页

外部登录

与远程服务器通信时使用的替代登录名和口令。缺省情况下，SQL Anywhere 每次代表其客户端连接到远程服务器时都会使用这些客户端的名称和口令。但是，您可以通过创建外部登录来替换这一缺省设置。外部登录是指与远程服务器通信时使用的替代登录名和口令。

外键

一个表中复制另一个表中主键值的一个或多个列。外键建立表间的关系。

另请参见：

- “主键”一节第 352 页
- “外表”一节第 344 页

外键约束

对单个列或一组列的限制，指定表中的数据与某个其它表中数据的关系。对列集施加外键约束可使这些列成为外键。

另请参见：

- “约束”一节第 351 页
- “检查约束”一节第 333 页
- “主键约束”一节第 352 页
- “唯一约束”一节第 346 页

外连接

一种保留表中所有行的连接。SQL Anywhere 支持左、右和完全外连接。左外连接保留表中位于连接运算符左侧的行，当右表中的行不满足连接条件时，它将返回空值。完全外连接保留两个表中的所有行。

另请参见：

- “连接”一节第 335 页
- “内连接”一节第 337 页

完全备份

对整个数据库和事务日志（可选）的备份。完全备份包含数据库中的所有信息，因此可以在系统或介质出现故障时提供保护。

另请参见： “增量备份”一节第 351 页

完整性

遵守完整性规则的情况，完整性规则确保数据正确并准确，而且数据库的关系结构保持不变。

另请参见： “参照完整性”一节第 326 页

网关

一种 MobiLink 对象，存储在 MobiLink 系统表或通告程序属性文件中，包含有关如何发送用于服务器启动同步的消息的信息。

另请参见： “服务器启动的同步”一节第 329 页

网络服务器

从共享公共网络的计算机接受连接的数据库服务器。

另请参见： “个人服务器”一节第 330 页

网络协议

通信类型，如 TCP/IP 或 HTTP。

维护版本

维护版本是一套完整的软件，它升级已安装的具有相同主版本号的较早版本的软件（版本号格式是 *major.minor.patch.build*）。升级程序的发行说明中列出了错误修正软件和其它更改。

唯一约束

对某个列或一组列的限制，它要求所有非空值都各不相同。一个表可以有多个唯一约束。

另请参见：

- “外键约束”一节第 345 页
- “主键约束”一节第 352 页
- “约束”一节第 351 页

谓语

一种条件表达式，可以选择性地将其与逻辑运算符 AND 和 OR 组合在一起，以组成 WHERE 或 HAVING 子句中的条件集。在 SQL 中，求值结果为 UNKNOWN 的谓语将解释为 FALSE。

位数组

位数组是一种用于有效率地存储位序列的数组数据结构。位数组与字符串类似，不同的是其各个部分由 0（零）和 1（一）而不是字符组成。位数组通常用于保存一串布尔值。

Windows

Microsoft Windows 操作系统系列，如 Windows Vista、Windows XP 和 Windows 200x。

Windows CE

请参见“Windows Mobile”一节第 346 页。

Windows Mobile

Microsoft 为移动设备制造的操作系统系列。

文件定义数据库

MobiLink 中一种用于创建下载文件的 SQL Anywhere 数据库。

另请参见： “基于文件的下载”一节第 333 页

物理索引

索引存储在磁盘上的实际索引结构。

系统表

一种表，由 SYS 或 dbo 拥有，用于保存元数据。系统表也称作数据字典表，由数据库服务器创建并维护。

系统对象

由 SYS 或 dbo 拥有的数据库对象。

系统视图

存在于每一个数据库中的一种视图，它以易于理解的格式表示系统表中包含的信息。

下载

同步过程的一个阶段，在此阶段数据从统一数据库传送到远程数据库。

相关名

查询的 FROM 子句中使用的表或视图的名称—要么是表或视图的原始名称，要么是在 FROM 子句中定义的替代名称。

项目

在 MobiLink 或 SQL Remote 中，项目是表示整个表或表中行和列子集的数据库对象。项目在发布中组合在一起。

另请参见：

- “复制”一节第 329 页
- “发布”一节第 328 页

消息存储库

QAnywhere 中客户端和服务器设备上存储消息的数据库。

另请参见：

- “客户端消息存储库”一节第 334 页
- “服务器消息存储库”一节第 329 页

消息类型

SQL Remote 复制中指定远程用户与统一数据库发布者通信方式的数据库对象。一个统一数据库可能定义了几种消息类型，这样一来，不同的远程用户就可以使用不同的消息系统与统一数据库进行通信。

另请参见：

- “复制”一节第 329 页
- “统一数据库”一节第 343 页

消息日志

可存储来自数据库服务器或 MobiLink 服务器等应用程序的消息的日志。此类信息还可以出现在消息窗口中或记录到文件中。消息日志包括信息性消息、错误、警告以及来自 MESSAGE 语句的消息。

消息系统

SQL Remote 复制中用于在统一数据库与远程数据库之间交换消息的协议。SQL Anywhere 包括对以下消息系统的支持：FILE、FTP 和 SMTP。

另请参见：

- “复制”一节第 329 页
- “FILE”一节第 329 页

卸载

卸载数据库时会将数据库的结构和/或数据导出到文本文件（如果是结构，则导出到 SQL 命令文件中；如果是数据，则导出到 ASCII 逗号分隔文件中）。使用卸载实用程序来卸载数据库。

此外，您也可以使用 UNLOAD 语句卸载数据的选定部分。

性能统计

反映数据库系统性能的值。例如，CURRREAD 统计表示数据库服务器已发出但尚未完成的文件读取次数。

业务规则

基于实际要求的准则。通常，业务规则通过检查约束、用户定义数据类型以及事务的正确使用来实现。

另请参见：

- “约束”一节第 351 页
- “用户定义数据类型”一节第 349 页

引用对象

一种对象（如视图），其定义直接引用数据库中的另一个对象（如表）。

另请参见： “外键”一节第 344 页

用户定义数据类型

请参见“域”一节第 349 页。

游标

指向结果集的已命名链接，用于通过编程接口访问和更新行。在 SQL Anywhere 中，游标支持在查询结果中进行向前和向后移动。游标由两部分组成：游标结果集（通常由 SELECT 语句定义）和游标位置。

另请参见：

- “游标结果集”一节第 349 页
- “游标位置”一节第 349 页

游标结果集

与游标关联的查询所得到的行集。

另请参见：

- “游标”一节第 349 页
- “游标位置”一节第 349 页

游标位置

指向游标结果集中一个行的指针。

另请参见：

- “游标”一节第 349 页
- “游标结果集”一节第 349 页

语句级触发器

在整个触发语句完成后执行的触发器。

另请参见：

- “触发器”一节第 327 页
- “行级触发器”一节第 331 页

域

内置数据类型的别名，其中包括适用的精度值和小数位值，还可以选择是否包括 DEFAULT 值和 CHECK 条件。SQL Anywhere 中预定义了一些域，如货币数据类型。也称作用户定义数据类型。

另请参见： “数据类型”一节第 340 页

预订

MobiLink 同步中发布与 MobiLink 用户之间的客户端数据库中的一个链接，它使发布所描述的数据能够得到同步。

SQL Remote 复制中发布与远程用户之间的一种链接，它使用户能够与统一数据库交换该发布上的更新。

另请参见：

- “发布”一节第 328 页
- “MobiLink 用户”一节第 337 页

元数据

数据的数据。元数据描述其它数据的性质和内容。

另请参见： “模式”一节第 337 页

原子事务

保证成功完成或保证根本不予完成的事务。如果错误使原子事务的一部分无法完成，则将回退事务以防止数据库处于不一致的状态。

REMOTE DBA 特权

在 SQL Remote 中，消息代理 (dbremote) 所需的权限级别。MobiLink 中 SQL Anywhere 同步客户端 (dbmlsync) 所需的权限级别。当消息代理或同步客户端作为具有该权限的用户建立连接时，它将具有完全的 DBA 访问权。如果不是通过消息代理或同步客户端进行连接，则该用户 ID 将不具有附加权限。

另请参见： “DBA 权限”一节第 328 页

远程 ID

SQL Anywhere 和 UltraLite 数据库中一种由 MobiLink 使用的唯一标识符。远程 ID 初始情况下设置为 NULL，在数据库第一次同步期间将设置为 GUID。

远程数据库

MobiLink 或 SQL Remote 中一种与统一数据库交换数据的数据库。远程数据库可以共享统一数据库中的全部或部分数据。

另请参见：

- “同步”一节第 343 页
- “统一数据库”一节第 343 页

约束

对特定数据库对象（如表或列）中所包含值的限制。例如，列可以具有唯一性约束，该约束要求该列中的所有值互不相同。表可以具有外键约束，该约束指定该表中的信息与某个其它表中数据的关系。

另请参见：

- “检查约束”一节第 333 页
- “外键约束”一节第 345 页
- “主键约束”一节第 352 页
- “唯一约束”一节第 346 页

运营公司

一种 MobiLink 对象，存储在 MobiLink 系统表或通告程序属性文件中，包含有关供服务器启动的同步使用的公共运营公司的信息。

另请参见： “服务器启动的同步”一节第 329 页

增量备份

仅包含事务日志的备份，通常在两次完全备份之间使用。

另请参见： “事务日志”一节第 339 页

争用

为获取资源而竞争的行为。例如，就数据库而言，如果有两个或更多个用户试图编辑数据库的同一行，就会为获得编辑该行的权利而发生争用。

正则表达式

正则表达式是字符、通配符和运算符的序列，用于定义某种模式以在字符串内进行搜索。

直方图

直方图是列统计信息最重要的组成部分，是一种表示数据分布的方式。SQL Anywhere 维护直方图以为优化程序提供有关列值分布情况的统计信息。

直接行处理

MobiLink 中一种用于将表数据同步到 MobiLink 支持的统一数据库以外的数据源的方法。使用直接行处理时，上载和下载都可以实现。

另请参见：

- “统一数据库”一节第 343 页
- “基于 SQL 的同步”一节第 333 页

主表

包含外键关系中的主键的表。

主键

其值唯一标识表中各行的一个列或多个列。

另请参见： “外键”一节第 344 页

主键约束

一种对主键列的唯一性约束。一个表只能有一个主键约束。

另请参见：

- “约束”一节第 351 页
- “检查约束”一节第 333 页
- “外键约束”一节第 345 页
- “唯一约束”一节第 346 页
- “完整性”一节第 345 页

子查询

嵌套在 SELECT、INSERT、UPDATE 或 DELETE 语句或者其它子查询中的 SELECT 语句。

有两种类型的子查询：相关子查询和嵌套子查询。

字符串

字符串是以单引号围起的字符序列。

字符集

字符集是一组符号，包括字母、数字、空格和其它符号。字符集的一个例子是 ISO-8859-1，又称作 Latin1。

另请参见：

- “代码页”一节第 327 页
- “编码”一节第 325 页
- “归类”一节第 331 页

索引

其它

~ULSqlcaWrap 函数
  ULSqlcaWrap 类 [UltraLite C++ API], 144
~ULSqlca 函数
  ULSqlca 类 [UltraLite C++ API], 136
~ULValue 函数
  ULValue 类 [UltraLite C++ API], 248
#define
  UltraLite 应用程序, 119

A

ActiveSync
  ULIsSynchronizeMessage 函数, 273
  UltraLite MFC 要求, 87
  UltraLite Windows Mobile 应用程序, 86
  UltraLite 消息, 119
  WindowProc 函数, 86
  用于 Windows Mobile 的 UltraLite 同步, 86
  用于 Windows Mobile 的 UltraLite 版本, 86
  类名, 84
AddRef 函数
  UltraLite_SQLObject_iface 类 [UltraLite C++ API], 206
AES 加密算法
  UltraLite 嵌入式 SQL 数据库, 50
AfterLast 函数
  UltraLite_Cursor_iface 类 [UltraLite C++ API], 172
ANSI
  UltraLite C++ 库, 28
API
  UltraLite Table API, 17
AutoCommit 模式
  UltraLite C++ 开发, 22
安全性
  Palm 上的加密, 68
  UltraLite C/C++ 应用程序, 73
  UltraLite 嵌入式 SQL 中的模糊处理, 50
  UltraLite 数据库加密, 50
  在 UltraLite 嵌入式 SQL 中更改加密密钥, 50
安装
  Palm OS UltraLite, 74
  UltraLite Windows Mobile 应用程序, 77
  用于 CodeWarrior 的 UltraLite 插件, 62

B

BeforeFirst 函数
  UltraLite_Cursor_iface 类 [UltraLite C++ API], 173
bool 运算符
  ULValue 类 [UltraLite C++ API], 246
帮助
  技术支持, xiv
包
  术语定义, 325
保存状态
  Palm OS 上的 UltraLite, 68
被引用对象
  术语定义, 325
编码
  术语定义, 325
编译
  UltraLite 嵌入式 SQL 应用程序, 58
  用于 Windows Mobile 的 UltraLite 应用程序, 78
编译器
  Palm OS, 61
  用于 Windows Mobile 的 UltraLite 应用程序, 78
编译器选项
  UltraLite C++ 开发, 28
编译器指令
  UltraLite 应用程序, 119
  UNDER_CE, 120
  UNDER_PALM_OS, 121
编译选项
  用于 Windows Mobile 的 UltraLite 应用程序, 78
标识符
  术语定义, 325
表
  UltraLite C++ API 模式信息, 23
表对象
  UltraLite C++ 数据检索示例, 15
并发
  术语定义, 325
部署
  Palm OS 上的 UltraLite, 74
  Palm OS 上的 UltraLite 应用程序, 74
  UltraLite for Windows Mobile, 82
  UltraLite 到 Palm OS, 74

C

C++ API
(*参见* UltraLite C/C++ API)
C++ 应用程序
(*参见* UltraLite C/C++)
CancelGetNotification 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 150
changeEncryptionKey 方法
UltraLite 嵌入式 SQL, 50
ChangeEncryptionKey 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 150
Checkpoint 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 151
CHECK 约束
术语定义, 333
重定向器
术语定义, 326
ClientParms 注册表条目
MobiLink 管道, 71
CLOSE 语句
UltraLite 嵌入式 SQL, 45
CodeWarrior
UltraLite C/C++ 开发, 61
UltraLite 模板, 63
使用 UltraLite 插件, 65
创建 UltraLite 项目, 63
安装 UltraLite 插件, 62
扩展模式 UltraLite 应用程序, 67
转换项目, 64
commit 方法
UltraLite C++ 事务, 22
Commit 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 151
Connection 对象
UltraLite C++, 11
CONNECT 语句
UltraLite 嵌入式 SQL, 34
CountUploadRows 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 151
CreateDatabase 函数
UltraLite_DatabaseManager_iface 类 [UltraLite C++ API], 180
CreateNotificationQueue 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 151
CustDB 应用程序
为 Windows Mobile 构建 UltraLite, 79
针对 Palm OS 的 UltraLite 构建, 66
参考数据库
术语定义, 326
参照完整性
术语定义, 326
策略
术语定义, 326
插件模块
术语定义, 326
插入
UltraLite C++ API 表行, 21
插入模式
UltraLite C++, 18
查询
UltraLite 嵌入式 SQL 单行查询, 44
UltraLite 嵌入式 SQL 多行查询, 45
术语定义, 326
查寻方法
UltraLite C++, 19
查寻模式
UltraLite C++, 18
查找方法
UltraLite C++, 19
查找模式
UltraLite C++, 18
查找详细信息并请求技术协助
技术支持, xv
程序结构
UltraLite 嵌入式 SQL, 31
持久存储
UltraLite for Windows Mobile, 81
冲突解决
术语定义, 326
抽取
术语定义, 326
触发器
术语定义, 327
处理错误
回调函数语法 (UltraLite C/C++), 94
ULRegisterErrorCallback (UltraLite C/C++), 114

传输规则
术语定义, 327
窗口 (OLAP)
术语定义, 327
创建 UltraLite 数据库
ULGetCollation_ 函数 (UltraLite C/C++), 100
数据库创建参数, 100
创建者 ID
Palm OS 应用程序, 70
关于, 70
术语定义, 327
存储过程
术语定义, 327
错误
UltraLite C++ API 处理, 24
UltraLite SQLCODE, 32
UltraLite sqlcode SQLCA 字段, 32
UltraLite 代码, 32
UltraLite 嵌入式 SQL 通信错误, 53
提供反馈, xiv
错误处理
回调函数语法 (UltraLite C/C++), 94
ULRegisterErrorCallback (UltraLite C/C++), 114
UltraLite C++, 24
错误检查
UltraLite ODBC 接口, 300

D

DatabaseManager 对象
UltraLite C++, 11
DatabaseSchema 对象
UltraLite C++, 23
db_fini 函数 [UL ESQL]
不要在 Palm Computing Platform 上使用, 251
语法, 251
db_init 函数 [UL ESQL]
语法, 252
db_start_database 函数 [UL ESQL]
语法, 253
db_stop_database 函数 [UL ESQL]
语法, 254
DBA 权限
术语定义, 328
DBMS
术语定义, 341
dbspaces
术语定义, 328
DCX
关于, x
DDL
术语定义, 340
DECL_BINARY 宏
UltraLite 嵌入式 SQL, 35
DECL_DATETIME 宏
UltraLite 嵌入式 SQL, 35
DECL_DATETIME 运算符
ULValue 类 [UltraLite C++ API], 245
DECL_DECIMAL 宏
UltraLite 嵌入式 SQL, 35
DECL_FIXCHAR 宏
UltraLite 嵌入式 SQL, 35
DECL_VARCHAR 宏
UltraLite 嵌入式 SQL, 35
DeclareEvent 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 152
DECLARE 语句
UltraLite 嵌入式 SQL, 45
DeleteAllRows 函数
UltraLite_Table_iface 类 [UltraLite C++ API], 215
DeleteNamed 函数
UltraLite_ResultSet_iface 类 [UltraLite C++ API], 199
Delete 函数
UltraLite_Cursor_iface 类 [UltraLite C++ API], 173
descUltraLite ODBC 接口
SQLDisconnect 函数, 291
DestroyNotificationQueue 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 153
DML
UltraLite C++, 13
术语定义, 340
DocCommentXchange (DCX)
关于, x
double 运算符
ULValue 类 [UltraLite C++ API], 246
DropDatabase 函数
UltraLite_DatabaseManager_iface 类 [UltraLite C++ API], 181
代理 ID
术语定义, 327
代理表

术语定义, 327
代码页
术语定义, 327
导入库
UltraLite C++, 28
动态 SQL
术语定义, 328
动态库
UltraLite C++ 应用程序, 28
读取
UltraLite 嵌入式 SQL, 44
对象树
术语定义, 328
多线程应用程序
UltraLite C++, 12
UltraLite 嵌入式 SQL, 34
多行查询
UltraLite 游标, 45

E

EBF
术语定义, 328
eMbedded Visual C++
用于 Windows Mobile 的 UltraLite 开发要求, 77
示例项目, 89
EXEC SQL
UltraLite 嵌入式 SQL 开发, 31
ExecuteNextSQLPassthroughScript 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 153
ExecuteQuery 函数
UltraLite_PreparedStatement_iface 类 [UltraLite C++ API], 194
ExecuteSQLPassthroughScripts 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 154
ExecuteStatement 函数
UltraLite_PreparedStatement_iface 类 [UltraLite C++ API], 194

F

FETCH 语句
UltraLite 嵌入式 SQL 单行查询, 44
UltraLite 嵌入式 SQL 多行查询, 45
FILE
术语定义, 329
FILE 消息类型
术语定义, 329
Finalize 函数
ULSqlcaBase 类 [UltraLite C++ API], 138
FindBegin 函数
UltraLite_Table_iface 类 [UltraLite C++ API], 216
FindFirst 函数
UltraLite_Table_iface 类 [UltraLite C++ API], 216
FindLast 函数
UltraLite_Table_iface 类 [UltraLite C++ API], 217
FindNext 函数
UltraLite_Table_iface 类 [UltraLite C++ API], 217
FindPrevious 函数
UltraLite_Table_iface 类 [UltraLite C++ API], 217
Find 函数
UltraLite_Table_iface 类 [UltraLite C++ API], 216
First 函数
UltraLite_Cursor_iface 类 [UltraLite C++ API], 173
float 运算符
ULValue 类 [UltraLite C++ API], 246
发布
术语定义, 328
发布更新
术语定义, 328
发布者
术语定义, 329
反馈
报告错误, xiv
提供, xiv
文档, xiv
请求更新, xiv
访问数据
UltraLite C++ Table API, 17
分析树
术语定义, 329
服务
术语定义, 329
服务器管理请求
术语定义, 329
服务器启动的同步
术语定义, 329
服务器消息存储库
术语定义, 329
复制
术语定义, 329
复制代理
术语定义, 329

复制服务器
术语定义, 330
复制频率
术语定义, 330
复制消息
术语定义, 330

G

GetBaseColumnName 函数
UltraLite_RowSchema_iface 类 [UltraLite C++ API], 201
GetBinary(p_ul_binary, size_t) 函数
ULValue 类 [UltraLite C++ API], 234
GetBinary(ul_byte *, size_t, size_t *) 函数
ULValue 类 [UltraLite C++ API], 234
GetBinaryLength 函数
ULValue 类 [UltraLite C++ API], 235
GetByteChunk 函数
UltraLite_StreamReader_iface 类 [UltraLite C++ API], 209
GetCA 函数
ULSqlcaBase 类 [UltraLite C++ API], 139
GetCollationName 函数
UltraLite_DatabaseSchema_iface 类 [UltraLite C++ API], 184
GetColumnCount 函数
UltraLite_IndexSchema_iface 类 [UltraLite C++ API], 188
UltraLite_RowSchema_iface 类 [UltraLite C++ API], 202
GetColumnDefault 函数
UltraLite_TableSchema_iface 类 [UltraLite C++ API], 223
GetColumnID 函数
UltraLite_RowSchema_iface 类 [UltraLite C++ API], 202
GetColumnName 函数
UltraLite_IndexSchema_iface 类 [UltraLite C++ API], 188
UltraLite_RowSchema_iface 类 [UltraLite C++ API], 202
GetColumnPrecision 函数
UltraLite_RowSchema_iface 类 [UltraLite C++ API], 203
GetColumnScale 函数
UltraLite_RowSchema_iface 类 [UltraLite C++ API], 204
GetColumnSize 函数
UltraLite_RowSchema_iface 类 [UltraLite C++ API], 204
GetColumnSQLName 函数
UltraLite_RowSchema_iface 类 [UltraLite C++ API], 203
GetColumnSQLType 函数
UltraLite_RowSchema_iface 类 [UltraLite C++ API], 203
GetColumnType 函数
UltraLite_RowSchema_iface 类 [UltraLite C++ API], 205
GetCombinedStringItem(ul_u_short, char *, size_t) 函数
ULValue 类 [UltraLite C++ API], 235
GetCombinedStringItem(ul_u_short, ul_wchar *, size_t) 函数
ULValue 类 [UltraLite C++ API], 235
GetConnectionNum 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 154
GetConnection 函数
UltraLite_SQLObject_iface 类 [UltraLite C++ API], 206
GetDatabaseID 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 154
GetDatabaseProperty(const ULValue &) 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 155
GetDatabaseProperty(ul_database_property_id) 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 154
GetGlobalAutoincPartitionSize 函数
UltraLite_TableSchema_iface 类 [UltraLite C++ API], 224
GetID 函数
UltraLite_IndexSchema_iface 类 [UltraLite C++ API], 189
UltraLite_TableSchema_iface 类 [UltraLite C++ API], 224
GetIFace 函数
UltraLite_SQLObject_iface 类 [UltraLite C++ API], 207
GetIndexCount 函数
UltraLite_TableSchema_iface 类 [UltraLite C++ API], 224

GetIndexName 函数
UltraLite_TableSchema_iface 类 [UltraLite C++ API], 225
GetIndexSchema 函数
UltraLite_TableSchema_iface 类 [UltraLite C++ API], 225
GetLastDownloadTime 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 155
GetLastIdentity 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 155
GetLength 函数
UltraLite_StreamReader_iface 类 [UltraLite C++ API], 210
GetName 函数
UltraLite_IndexSchema_iface 类 [UltraLite C++ API], 189
UltraLite_TableSchema_iface 类 [UltraLite C++ API], 226
GetNewUUID(GUID *) 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 156
GetNewUUID(p_ul_binary) 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 156
GetNotificationParameter 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 157
GetNotification 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 156
GetOptimalIndex 函数
UltraLite_TableSchema_iface 类 [UltraLite C++ API], 226
GetParameterCount 函数
ULSqlcaBase 类 [UltraLite C++ API], 140
GetParameter 函数
ULSqlcaBase 类 [UltraLite C++ API], 139
GetPlan(char *, size_t) 函数
UltraLite_PreparedStatement_iface 类 [UltraLite C++ API], 195
GetPlan(ul_wchar *, size_t) 函数
UltraLite_PreparedStatement_iface 类 [UltraLite C++ API], 195
GetPrimaryKey 函数
UltraLite_TableSchema_iface 类 [UltraLite C++ API], 226
GetPublicationCount 函数
UltraLite_DatabaseSchema_iface 类 [UltraLite C++ API], 184
GetPublicationID 函数
UltraLite_DatabaseSchema_iface 类 [UltraLite C++ API], 185
GetPublicationName 函数
UltraLite_DatabaseSchema_iface 类 [UltraLite C++ API], 185
GetPublicationPredicate 函数
UltraLite_TableSchema_iface 类 [UltraLite C++ API], 226
GetReferencedIndexName 函数
UltraLite_IndexSchema_iface 类 [UltraLite C++ API], 189
GetReferencedTableName 函数
UltraLite_IndexSchema_iface 类 [UltraLite C++ API], 190
GetRowCount 函数
UltraLite_Cursor_iface 类 [UltraLite C++ API], 174
GetSchema 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 158
UltraLite_PreparedStatement_iface 类 [UltraLite C++ API], 195
UltraLite_ResultSet_iface 类 [UltraLite C++ API], 199
UltraLite_Table_iface 类 [UltraLite C++ API], 218
GetSqlca 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 158
GetSQLCode 函数
ULSqlcaBase 类 [UltraLite C++ API], 140
GetSQLCount 函数
ULSqlcaBase 类 [UltraLite C++ API], 140
GetSQLErrorOffset 函数
ULSqlcaBase 类 [UltraLite C++ API], 141
GetSQLPassthroughScriptCount [UL ESQL] 语法, 267
GetSQLPassthroughScriptCount 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 158
GetState 函数

UltraLite_Cursor_iface 类 [UltraLite C++ API], 174
GetStreamReader 函数
UltraLite_Cursor_iface 类 [UltraLite C++ API], 174
GetStreamWriter 函数
UltraLite_Cursor_iface 类 [UltraLite C++ API], 175
UltraLite_PreparedStatement_iface 类 [UltraLite C++ API], 196
GetString(char *, size_t) 函数
ULValue 类 [UltraLite C++ API], 236
GetString(ul_wchar *, size_t) 函数
ULValue 类 [UltraLite C++ API], 236
GetStringChunk 函数
UltraLite_StreamReader_iface 类 [UltraLite C++ API], 210
GetStringLength 函数
ULValue 类 [UltraLite C++ API], 236
GetSuspend 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 158
UltraLite_Cursor_iface 类 [UltraLite C++ API], 175
GetSynchResult 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 158
GetTableCount 函数
UltraLite_DatabaseSchema_iface 类 [UltraLite C++ API], 185
GetTableName 函数
UltraLite_DatabaseSchema_iface 类 [UltraLite C++ API], 185
UltraLite_IndexSchema_iface 类 [UltraLite C++ API], 190
GetTableSchema 函数
UltraLite_DatabaseSchema_iface 类 [UltraLite C++ API], 186
GetUploadUnchangedRows 函数
UltraLite_TableSchema_iface 类 [UltraLite C++ API], 227
GetUtilityULValue 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 159
Get 函数
UltraLite_Cursor_iface 类 [UltraLite C++ API], 173
GlobalAutoincUsage 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 159
grantConnectTo 方法
UltraLite C++ 开发, 25
GrantConnectTo 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 159
GUID 运算符
ULValue 类 [UltraLite C++ API], 245
高度加密
UltraLite 嵌入式 SQL, 50
UltraLite 数据库, 108
隔离级别
术语定义, 330
个人服务器
术语定义, 330
更新
UltraLite C++ API 表行, 20
更新模式
UltraLite C++, 18
工作表
术语定义, 330
构建
UltraLite 嵌入式 SQL 应用程序, 58
构建过程
UltraLite 嵌入式 SQL 应用程序, 58
嵌入式 SQL 应用程序, 58
故障排除
UltraLite C++ 处理错误, 24
UltraLite C/C++ 使用 ULRegisterErrorCallback, 114
故障切换
术语定义, 330
关键连接
术语定义, 340
管理
UltraLite C++ 事务, 22
规范化
术语定义, 331
归类
术语定义, 331
滚动
UltraLite C++ Table API, 17

H

HasResultSet 函数

UltraLite_PreparedStatement_iface 类 [UltraLite C++ API], 196
HotSync 同步
Palm OS, 71
HTTPS 同步
UltraLite for Palm OS, 73
HTTP 同步
UltraLite for Palm OS, 73
回调
ULRegisterSQLPassthroughCallback (UltraLite C/C++), 116
ULRegisterSynchronizationCallback (UltraLite C/C++), 118
函数
UltraLite 嵌入式 SQL, 250
宏
UL_SYNC_ALL, 119
UL_SYNC_ALL_PUBS, 120
UL_TEXT, 120
UL_USE_DLL, 120
UltraLite 应用程序, 119
环境变量
INCLUDE, 309
命令 shell, xiii
命令提示符, xiii
回退
UltraLite C++ 事务, 22
回退日志
术语定义, 331
获取帮助
技术支持, xiv

I

iAnywhere JDBC 驱动程序
术语定义, 331
iAnywhere 开发人员社区
新闻组, xv
INCLUDE 语句
UltraLite SQLCA, 32
InDatabase 函数
ULValue 类 [UltraLite C++ API], 237
IndexSchema 对象
UltraLite C++ 开发, 23
InfoMaker
术语定义, 331
Initialize 函数
ULSqlcaBase 类 [UltraLite C++ API], 141
InitSynchInfo(ul_synch_info_a *) 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 160
InitSynchInfo(ul_synch_info_w2 *) 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 160
InPublication 函数
UltraLite_TableSchema_iface 类 [UltraLite C++ API], 227
InsertBegin 函数
UltraLite_Table_iface 类 [UltraLite C++ API], 218
Insert 函数
UltraLite_Table_iface 类 [UltraLite C++ API], 218
install-dir
文档用法, xii
Interactive SQL
术语定义, 332
int 运算符
ULValue 类 [UltraLite C++ API], 246
IsCaseSensitive 函数
UltraLite_DatabaseSchema_iface 类 [UltraLite C++ API], 186
IsColumnAutoinc 函数
UltraLite_TableSchema_iface 类 [UltraLite C++ API], 228
IsColumnCurrentDate 函数
UltraLite_TableSchema_iface 类 [UltraLite C++ API], 228
IsColumnCurrentTimestamp 函数
UltraLite_TableSchema_iface 类 [UltraLite C++ API], 229
IsColumnCurrentTime 函数
UltraLite_TableSchema_iface 类 [UltraLite C++ API], 228
IsColumnDescending 函数
UltraLite_IndexSchema_iface 类 [UltraLite C++ API], 190
IsColumnGlobalAutoinc 函数
UltraLite_TableSchema_iface 类 [UltraLite C++ API], 229
IsColumnInIndex 函数
UltraLite_TableSchema_iface 类 [UltraLite C++ API], 230
IsColumnNewUUID 函数
UltraLite_TableSchema_iface 类 [UltraLite C++ API], 230
IsColumnNullable 函数

UltraLite_TableSchema_iface 类 [UltraLite C++ API], 231
IsForeignKeyCheckOnCommit 函数
UltraLite_IndexSchema_iface 类 [UltraLite C++ API], 191
IsForeignKeyNullable 函数
UltraLite_IndexSchema_iface 类 [UltraLite C++ API], 191
IsForeignKey 函数
UltraLite_IndexSchema_iface 类 [UltraLite C++ API], 190
IsNeverSynchronized 函数
UltraLite_TableSchema_iface 类 [UltraLite C++ API], 231
IsNull 函数
UltraLite_Cursor_iface 类 [UltraLite C++ API], 175
ULValue 类 [UltraLite C++ API], 237
IsPrimaryKey 函数
UltraLite_IndexSchema_iface 类 [UltraLite C++ API], 191
IsUniqueIndex 函数
UltraLite_IndexSchema_iface 类 [UltraLite C++ API], 192
IsUniqueKey 函数
UltraLite_IndexSchema_iface 类 [UltraLite C++ API], 192

J

JAR 文件
术语定义, 332
Java 类
术语定义, 332
jConnect
术语定义, 332
JDBC
术语定义, 332
校验
术语定义, 334
校验和
术语定义, 334
基表
术语定义, 332
基于 SQL 的同步
术语定义, 333
基于会话的同步
术语定义, 332
基于脚本的上载
术语定义, 333
基于文件的下载
术语定义, 333
集成登录
术语定义, 333
技术支持
新闻组, xv
加密
Palm OS, 68
UltraLite C++ 开发, 26
UltraLite C/C++ 中的 ULEnableStrongEncryption 函数, 108
UltraLite 嵌入式 SQL 数据库, 50
使用嵌入式 SQL 的 UltraLite 数据库, 50
在 UltraLite 嵌入式 SQL 中存储加密密钥, 68
在 UltraLite 嵌入式 SQL 中更改密钥, 50
更改 UltraLite 的加密密钥（嵌入式 SQL）, 255
监听器
术语定义, 333
检查点
术语定义, 333
简单加密
UltraLite 数据库简单加密, 50
脚本
术语定义, 333
脚本版本
术语定义, 334
角色
术语定义, 334
角色名
术语定义, 334
教程
UltraLite C++ API, 309
截断
UltraLite FETCH, 43
结果集
UltraLite C++ 浏览, 15
结果集模式
UltraLite C++, 16
静态库
UltraLite C++ 应用程序, 28
镜像日志
术语定义, 334
局部临时表
术语定义, 334

K

可重新启动的下载
UltraLite 嵌入式 SQL, 277
开发
UltraLite C++, 9
开发工具
UltraLite 嵌入式 SQL, 58
开发过程
UltraLite 嵌入式 SQL, 2
开发平台
UltraLite C++, 3
开发人员社区
新闻组, xv
客户端/服务器
术语定义, 334
客户端消息存储库
术语定义, 334
客户端消息存储库 ID
术语定义, 335
口令
UltraLite C++ API 验证, 25
库
C++ 中的 UltraLite 编译和链接, 28
UltraLite DLL for Windows Mobile, 82
UltraLite Unicode 库, 28
UltraLite 链接用 C++ 编写的示例, 310
用于 Palm OS 的 UltraLite 应用程序, 67
用于 Windows Mobile 的 UltraLite 应用程序, 78
库函数
GetSQLPassthroughScriptCount（UltraLite 嵌入式 SQL）, 267
回调函数语法 (UltraLite C/C++), 94, 95
MLFileTransfer（UltraLite 嵌入式 SQL）, 97
ULChangeEncryptionKey（UltraLite 嵌入式 SQL）, 255
ULCheckpoint（UltraLite 嵌入式 SQL）, 256
ULClearEncryptionKey（UltraLite 嵌入式 SQL）, 257
ULCountUploadRows（UltraLite 嵌入式 SQL）, 258
ULCreateDatabase（UltraLite 嵌入式 SQL）, 100
ULDropDatabase（UltraLite 嵌入式 SQL）, 259
ULEnableEccSyncEncryption (UltraLite C/C++), 102
ULEnableFIPSStrongEncryption (UltraLite C/C++), 103
ULEnableHttpsSynchronization (UltraLite C/C++), 105
ULEnableHttpSynchronization (UltraLite C/C++), 104
ULEnableRsaFipsSyncEncryption (UltraLite C/C++), 106
ULEnableRsaSyncEncryption (UltraLite C/C++), 107
ULEnableStrongEncryption (UltraLite C/C++), 108
ULEnableTcpipSynchronization (UltraLite C/C++), 109
ULEnableTlsSynchronization (UltraLite C/C++), 110
ULEnableZlibSyncCompression (UltraLite C/C++), 111
ULExecuteNextSQLPassthroughScript（UltraLite 嵌入式 SQL）, 260
ULExecuteSQLPassthroughScripts（UltraLite 嵌入式 SQL）, 261
ULGetDatabaseID（UltraLite 嵌入式 SQL）, 262
ULGetDatabaseProperty （UltraLite 嵌入式 SQL）, 263
ULGetErrorParameterCount（UltraLite 嵌入式 SQL）, 265
ULGetErrorParameter（UltraLite 嵌入式 SQL）, 264
ULGetLastDownloadTime（UltraLite 嵌入式 SQL）, 266
ULGetSynchResult（UltraLite 嵌入式 SQL）, 268
ULGlobalAutoincUsage（UltraLite 嵌入式 SQL）, 270
ULGrantConnectTo（UltraLite 嵌入式 SQL）, 271
ULInitDatabaseManager (UltraLite C/C++), 112
ULInitDatabaseManagerNoSQL (UltraLite C/C++), 113
ULInitSynchInfo UltraLite（嵌入式 SQL）, 272
ULIsSynchronizeMessage（UltraLite 嵌入式 SQL）, 273
ULRegisterErrorCallback (UltraLite C/C++), 114
ULRegisterSQLPassthroughCallback (UltraLite C/C++), 116
ULRegisterSynchronizationCallback (UltraLite C/C++), 118

ULResetLastDownloadTime（UltraLite 嵌入式 SQL）, 274
ULRetrieveEncryptionKey（UltraLite 嵌入式 SQL）, 275
ULRevokeConnectFrom（UltraLite 嵌入式 SQL）, 276
ULRollbackPartialDownload（UltraLite 嵌入式 SQL）, 277
ULSaveEncryptionKey（UltraLite 嵌入式 SQL）, 278
ULSetDatabaseID（UltraLite 嵌入式 SQL）, 279
ULSetDatabaseOptionString（UltraLite 嵌入式 SQL）, 280
ULSetDatabaseOptionULong（UltraLite 嵌入式 SQL）, 281
ULSetSynchInfo（UltraLite 嵌入式 SQL）, 282
ULSignalSyncIsComplete（UltraLite 嵌入式 SQL）, 283
ULSynchronize（UltraLite 嵌入式 SQL）, 284
UltraLite 嵌入式 SQL, 250
快照隔离
术语定义, 335
扩展模式
Palm OS UltraLite 应用程序, 67

L

LastCodeOK 函数
ULSqlcaBase 类 [UltraLite C++ API], 141
LastFetchOK 函数
ULSqlcaBase 类 [UltraLite C++ API], 142
Last 函数
UltraLite_Cursor_iface 类 [UltraLite C++ API], 175
long 运算符
ULValue 类 [UltraLite C++ API], 246
LookupBackward 函数
UltraLite_Table_iface 类 [UltraLite C++ API], 219
LookupBegin 函数
UltraLite_Table_iface 类 [UltraLite C++ API], 219
LookupForward 函数
UltraLite_Table_iface 类 [UltraLite C++ API], 219
Lookup 函数
UltraLite_Table_iface 类 [UltraLite C++ API], 218
LTM
术语定义, 336
类名
ActiveSync 同步, 84
联机手册
PDF, x
连接
UltraLite C++ 数据库, 11
UltraLite C/C++ 中的 SQLCA, 5
UltraLite 嵌入式 SQL, 34
术语定义, 335
连接 ID
术语定义, 335
连接类型
术语定义, 335
连接配置
术语定义, 335
连接启动的同步
术语定义, 335
连接条件
术语定义, 335
链接
UltraLite C++ 应用程序, 28
用于 Windows Mobile 的 UltraLite 应用程序, 78
列
值的 UltraLite C++ API 修改, 19
值的 UltraLite C++ API 检索, 18
临时表
术语定义, 335
流定义函数
ULSetDatabaseID（嵌入式 SQL）, 279
轮询
术语定义, 336
逻辑索引
术语定义, 336
浏览
UltraLite C++ Table API, 17
浏览 SQL 结果集
UltraLite C++, 15

M

MFC
UltraLite 应用程序 ActiveSync 要求, 87
MFC 应用程序
UltraLite for Windows Mobile, 84
MLFileTransfer 函数 [UL ESQL]
语法, 97
MobiLink
术语定义, 336
MobiLink 服务器
术语定义, 336

MobiLink 监控器
术语定义, 336
MobiLink 客户端
术语定义, 337
MobiLink 系统表
术语定义, 337
MobiLink 用户
术语定义, 337
moveFirst 方法（表对象）
UltraLite C++ 数据检索示例, 15
moveNext 方法（表对象）
UltraLite C++ 数据检索示例, 15
命令 shell
大括号, xiii
引号, xiii
括号, xiii
环境变量, xiii
约定, xiii
命令提示符
大括号, xiii
引号, xiii
括号, xiii
环境变量, xiii
约定, xiii
命令文件
术语定义, 336
命名空间
UltraLite C++ 示例, 310
模糊处理
UltraLite C++ 开发, 26
UltraLite 嵌入式 SQL 数据库, 50
使用嵌入式 SQL 的 UltraLite 数据库, 50
模拟器
UltraLite for Windows Mobile, 82
模式
UltraLite C++, 18
UltraLite C++ API 访问, 23
术语定义, 337
目标平台
UltraLite C++, 3

N

Next 函数
UltraLite_Cursor_iface 类 [UltraLite C++ API], 176
NULL
UltraLite 指示符变量, 42
内连接
术语定义, 337

O

observer 同步参数
UltraLite 嵌入式 SQL 示例, 56
ODBC
术语定义, 337
ODBC 管理器
术语定义, 337
ODBC 数据源
术语定义, 337
OpenConnection 函数
UltraLite_DatabaseManager_iface 类 [UltraLite C++ API], 181
OpenTableEx 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 161
OpenTableWithIndex 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 161
OpenTable 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 160
open 方法（表对象）
UltraLite C++ 数据检索示例, 15
OPEN 语句
UltraLite 嵌入式 SQL, 45
operator= 运算符
ULValue 类 [UltraLite C++ API], 248

P

Palm OS
C/C++ 中的 UltraLite HTTP 同步, 73
C/C++ 中的 UltraLite TCP/IP 同步, 73
UltraLite C++ 应用程序, 61
UltraLite 中的 HotSync 同步, 71
UltraLite 使用 CodeWarrior 构建 CustDB 应用程序, 66
UltraLite 运行时库, 28
创建者 ID, 70
安全性, 73
安装 UltraLite 应用程序, 74
平台要求, 61
PATH 环境变量
HotSync, 61
PDB

术语定义, 337
PDF
文档, x
PowerDesigner
术语定义, 337
PowerJ
术语定义, 338
preparedStatement 类
UltraLite C++, 13
PrepareStatement 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 162
Previous 函数
UltraLite_Cursor_iface 类 [UltraLite C++ API], 176
配置
UltraLite 嵌入式 SQL 的开发工具, 58
偏移
UltraLite C++ 相对, 17
平台
在 UltraLite C++ 中受支持, 3
平台要求
UltraLite for Windows Mobile, 77

Q

QAnywhere
术语定义, 338
QAnywhere 代理
术语定义, 338
前缀文件
关于, 65
嵌入式 SQL
(*参见* UltraLite 嵌入式 SQL)
术语定义, 338
嵌入式 SQL 库函数 (*见* UltraLite 嵌入式 SQL 库函数)
权限
UltraLite 嵌入式 SQL, 31
全局临时表
术语定义, 338
全局数据库标识符
UltraLite 嵌入式 SQL, 279
全局自动增量
ULGlobalAutoincUsage 函数, 270
ULSetDatabaseID 函数（UltraLite 嵌入式 SQL）, 279

R

RDBMS
术语定义, 330
RegisterForEvent 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 162
Relative 函数
UltraLite_Cursor_iface 类 [UltraLite C++ API], 176
Release 函数
UltraLite_SQLObject_iface 类 [UltraLite C++ API], 207
REMOTE DBA 权限
术语定义, 350
Reopen 方法
UltraLite C/C++, 68
ResetLastDownloadTime 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 163
RevokeConnectFrom 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 163
revokeConnectionFrom 方法
UltraLite C++ 开发, 25
RollbackPartialDownload 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 164
rollback 方法
UltraLite C++ 事务, 22
Rollback 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 163
日志文件
术语定义, 338

S

samples-dir
文档用法, xii
SELECT 语句
UltraLite C++ 数据检索示例, 15
UltraLite 嵌入式 SQL 单行, 44
SendNotification 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 164
SetBinary 函数
ULValue 类 [UltraLite C++ API], 238

SET CONNECTION 语句
  UltraLite 嵌入式 SQL 中的多个连接, 34
SetDatabaseID 函数
  UltraLite_Connection_iface 类 [UltraLite C++ API], 165
SetDatabaseOption(const ULValue &, const ULValue &) 函数
  UltraLite_Connection_iface 类 [UltraLite C++ API], 165
SetDatabaseOption(ul_database_option_id, const ULValue &) 函数
  UltraLite_Connection_iface 类 [UltraLite C++ API], 165
SetDefault 函数
  UltraLite_Cursor_iface 类 [UltraLite C++ API], 177
SetNull 函数
  UltraLite_Cursor_iface 类 [UltraLite C++ API], 177
SetParameterNull 函数
  UltraLite_PreparedStatement_iface 类 [UltraLite C++ API], 197
SetParameter 函数
  UltraLite_PreparedStatement_iface 类 [UltraLite C++ API], 196
SetReadPosition 函数
  UltraLite_StreamReader_iface 类 [UltraLite C++ API], 211
SetString(const char *, size_t) 函数
  ULValue 类 [UltraLite C++ API], 238
SetString(const ul_wchar *, size_t) 函数
  ULValue 类 [UltraLite C++ API], 238
SetSuspend 函数
  UltraLite_Connection_iface 类 [UltraLite C++ API], 166
  UltraLite_Cursor_iface 类 [UltraLite C++ API], 177
SetSynchInfo(ul_wchar const *, ul_synch_info_a *) 函数
  UltraLite_Connection_iface 类 [UltraLite C++ API], 166
SetSynchInfo(ul_wchar const *, ul_synch_info_w2 *) 函数
  UltraLite_Connection_iface 类 [UltraLite C++ API], 166
Set 函数
  UltraLite_Cursor_iface 类 [UltraLite C++ API], 176
short 运算符
  ULValue 类 [UltraLite C++ API], 247
Shutdown 函数
  UltraLite_Connection_iface 类 [UltraLite C++ API], 167
  UltraLite_DatabaseManager_iface 类 [UltraLite C++ API], 182
SQL
  术语定义, 342
SQLAllocHandle 函数 [UL ODBC]
  语法, 286
SQL Anywhere
  文档, x
  术语定义, 342
SQLBindCol 函数 [UL ODBC]
  语法, 287
SQLBindParameter 函数 [UL ODBC]
  语法, 288
SQLCA
  UltraLite C/C++, 5
  UltraLite 字段, 32
  UltraLite 嵌入式 SQL, 32
  UltraLite 嵌入式 SQL 多个 SQLCA , 34
sqlcabc SQLCA 字段
  UltraLite 嵌入式 SQL, 32
sqlcaid SQLCA 字段
  UltraLite 嵌入式 SQL, 32
SQLCODE
  回调函数语法 (UltraLite C/C++), 94, 95
  UltraLite C++ 错误处理, 24
sqlcode SQLCA 字段
  UltraLite 嵌入式 SQL, 32
SQLConnect 函数 [UL ODBC]
  语法, 289
SQLDescribeCol 函数 [UL ODBC]
  语法, 290
SQLDisconnect 函数 [UL ODBC]
  语法, 291
SQLEndTran 函数 [UL ODBC]
  语法, 292
sqlerrd SQLCA 字段
  UltraLite 嵌入式 SQL, 32
sqlerrmc SQLCA 字段
  UltraLite 嵌入式 SQL, 32
sqlerrml SQLCA 字段

UltraLite 嵌入式 SQL, 32
sqlerrp SQLCA 字段
UltraLite 嵌入式 SQL, 32
SQLExecDirect 函数 [UL ODBC]
语法, 293
SQLExecute 函数 [UL ODBC]
语法, 294
SQLFetchScroll 函数 [UL ODBC]
语法, 296
SQLFetch 函数 [UL ODBC]
语法, 295
SQLFreeHandle 函数 [UL ODBC]
语法, 297
SQLGetCursorName 函数 [UL ODBC]
语法, 298
SQLGetData 函数 [UL ODBC]
语法, 299
SQLGetDiagRec 函数 [UL ODBC]
语法, 300
SQLGetInfo 函数 [UL ODBC]
语法, 301
SQLNumResultCols 函数 [UL ODBC]
语法, 302
sqlpp 实用程序
UltraLite 嵌入式 SQL 应用程序, 58
SQLPrepare 函数 [UL ODBC]
语法, 303
SQL Remote
术语定义, 342
SQLRowCount 函数 [UL ODBC]
语法, 304
SQLSetConnectionName 函数 [UL ODBC]
语法, 305
SQLSetCursorName 函数 [UL ODBC]
语法, 306
SQLSetSuspend 函数 [UL ODBC]
语法, 307
sqlstate SQLCA 字段
UltraLite 嵌入式 SQL, 33
SQLSynchronize 函数 [UL ODBC]
语法, 308
sqlwarn SQLCA 字段
UltraLite 嵌入式 SQL, 32
SQL 通信区
UltraLite C/C++, 5
UltraLite 嵌入式 SQL, 32
SQL 语句
术语定义, 342
SQL 预处理器
UltraLite 嵌入式 SQL 应用程序, 58
SQL 直通
回调函数语法 (UltraLite C/C++), 95
ULRegisterSQLPassthroughCallback (UltraLite C/C++), 116
StartSynchronizationDelete 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 167
StopSynchronizationDelete 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 167
StringCompare 函数
ULValue 类 [UltraLite C++ API], 239
StrToUUID(GUID, const ULValue &) 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 168
StrToUUID(p_ul_binary, size_t, const ULValue &) 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 168
Sybase Central
术语定义, 343
Synchronize(ul_synch_info_a *) 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 168
Synchronize(ul_synch_info_w2 *) 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 169
SynchronizeFromProfile 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 169
SYS
术语定义, 343
散列
术语定义, 338
删除
UltraLite C++ API 表行, 21
上次下载时间戳
ULGetLastDownloadTime 函数, 266
在 UltraLite 数据库中重置, 274
上载
术语定义, 339
设备跟踪
术语定义, 339
声明

UltraLite 主机变量, 35
声明部分
UltraLite 嵌入式 SQL 声明, 35
生成的连接条件
术语定义, 340
生成的数据库
在 UltraLite 中命名, 65
生成文件
UltraLite 嵌入式 SQL, 58
实例化视图
术语定义, 339
示例应用程序
为 Windows Mobile 构建 UltraLite, 79
针对 Palm OS 的 UltraLite 构建, 66
世代号
术语定义, 339
事件模型
术语定义, 339
事务
UltraLite C++ 管理, 22
在使用嵌入式 SQL 的 UltraLite 中提交, 53
术语定义, 339
事务处理
UltraLite C++ 管理, 22
事务日志
术语定义, 339
事务日志镜像
术语定义, 340
事务完整性
术语定义, 340
视图
术语定义, 339
授权选项
术语定义, 340
受保护的功能
术语定义, 340
术语表
SQL Anywhere 术语列表, 325
数据操作
UltraLite C++ Table API, 17
使用 SQL 的 UltraLite C++ API, 13
数据操作语言
术语定义, 340
数据库
术语定义, 341
数据库对象
术语定义, 341
数据库服务器
术语定义, 341
数据库管理员
术语定义, 341
数据库连接
术语定义, 341
数据库名称
术语定义, 341
数据库模式
UltraLite C++ API 访问, 23
数据库所有者
术语定义, 342
数据库文件
UltraLite for Windows Mobile, 81
UltraLite 加密和模糊处理（嵌入式 SQL）, 50
术语定义, 342
数据类型
UltraLite C++ API 访问于, 18
UltraLite C++ API 转换, 19
UltraLite 嵌入式 SQL, 35
术语定义, 340
数据立方体
术语定义, 341
死锁
术语定义, 342
搜索
使用 UltraLite C++ 搜索行, 19
索引
UltraLite C++ API 模式信息, 23
术语定义, 343
锁定
术语定义, 342

T

Table API
UltraLite C++ 简介, 17
TableSchema 对象
UltraLite C++ 开发, 23
TCP/IP 同步
C/C++ 中的 UltraLite for Palm OS, 73
TriggerEvent 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 170
TruncateTable 函数
UltraLite_Table_iface 类 [UltraLite C++ API], 220
提交
UltraLite C++ 事务, 22

使用嵌入式 SQL 提交 UltraLite 更改, 53
提示
UltraLite 开发, 53
通告程序
术语定义, 343
通信错误
UltraLite 嵌入式 SQL, 53
通信流
术语定义, 343
同步
C/C++ 中的 UltraLite HTTP, 73
C/C++ 中的 UltraLite TCP/IP, 73
CodeWarrior 加密库, 65
ULRegisterSynchronizationCallback (UltraLite C/C++), 118
UltraLite C++ API 教程, 317
UltraLite C++ 中的 ULEnableRsaFipsSyncEncryption, 106
UltraLite C/C++ 中的 ULEnableEccaSyncEncryption 函数, 102
UltraLite C/C++ 中的 ULEnableFIPSStrongEncryption 函数, 103
UltraLite C/C++ 中的 ULEnableHttpsSynchronization 函数, 105
UltraLite C/C++ 中的 ULEnableHttpSynchronization 函数, 104
UltraLite C/C++ 中的 ULEnableRsaSyncEncryption 函数, 107
UltraLite C/C++ 中的 ULEnableTcpipSynchronization 函数, 109
UltraLite C/C++ 中的 ULEnableTlsSynchronization 函数, 110
UltraLite C/C++ 中的 ULEnableZlibSyncCompression 函数, 111
UltraLite C/C++ 应用程序, 27
UltraLite for Windows Mobile 同步的菜单控件, 88
UltraLite for Windows Mobile 简介, 86
UltraLite ODBC 接口, 308
UltraLite 中的 HotSync, 71
UltraLite 中的 Palm OS, 71
UltraLite 嵌入式 SQL, 51
UltraLite 嵌入式 SQL 中的初始化, 53
UltraLite 嵌入式 SQL 中的疑难解答, 268
UltraLite 嵌入式 SQL 提交更改, 53
UltraLite 嵌入式 SQL 示例, 52
在 UltraLite 嵌入式 SQL 中取消, 53
在 UltraLite 嵌入式 SQL 中监控, 53
在 UltraLite 嵌入式 SQL 中调用, 52
术语定义, 343
添加到 UltraLite 嵌入式 SQL 应用程序, 51
同步错误
UltraLite 嵌入式 SQL 通信错误, 53
同步函数
ULInitSynchInfo（UltraLite 嵌入式 SQL）, 272
ULSetSynchInfo（UltraLite 嵌入式 SQL）, 282
ULSignalSyncIsComplete（UltraLite 嵌入式 SQL）, 283
同步数据
UltraLite 关于, 27
同步状态
ULGetSynchResult 函数, 268
统一数据库
术语定义, 343
图标
此帮助文档中使用的, xiii
推式请求
术语定义, 344
推式通知
术语定义, 344

U

UL_AS_SYNCHRONIZE 宏
ActiveSync UltraLite 消息, 119
ul_s_big 运算符
ULValue 类 [UltraLite C++ API], 247
ul_sql_passthrough_status 结构 [UltraLite C++ API]
说明, 125
ul_stream_error 结构 [UltraLite C++ API]
说明, 126
UL_SYNC_ALL_PUBS 宏
关于, 120
UL_SYNC_ALL 宏
关于, 119
ul_synch_info_a 结构 [UltraLite C++ API]
说明, 127
ul_synch_info_w2 结构 [UltraLite C++ API]
说明, 129
ul_synch_info 结构
关于, 51
ul_synch_result 结构 [UltraLite C++ API]
说明, 131
ul_synch_stats 结构 [UltraLite C++ API]
说明, 132

ul_synch_status 结构
  UltraLite 嵌入式 SQL, 54
ul_synch_status 结构 [UltraLite C++ API]
  说明, 133
UL_TEXT 宏
  关于, 120
ul_u_big 运算符
  ULValue 类 [UltraLite C++ API], 247
UL_USE_DLL 宏
  关于, 120
ul_validate_data 结构 [UltraLite C++ API]
  说明, 135
ULActiveSyncStream 函数
  Windows Mobile 用法, 86
ulbase.lib
  UltraLite C++ 开发, 28
ULChangeEncryptionKey 函数 [UL ESQL]
  使用, 50
  语法, 255
ULCheckpoint 函数 [UL ESQL]
  语法, 256
ULClearEncryptionKey 函数 [UL ESQL]
  使用, 68
  语法, 257
ULCountUploadRows 函数 [UL ESQL]
  语法, 258
ULCreateDatabase 函数 [UL ESQL]
  语法, 100
ULDropDatabase 函数 [UL ESQL]
  语法, 259
ULEnableEccSyncEncryption 函数 [UL C/C++]
  语法, 102
ULEnableFIPSStrongEncryption 函数 [UL C/C++]
  语法, 103
ULEnableHttpsSynchronization 函数 [UL C/C++]
  语法, 105
ULEnableHttpSynchronization 函数 [UL C/C++]
  语法, 104
ULEnableRsaFipsSyncEncryption 函数 [UL C/C++]
  语法, 106
ULEnableRsaSyncEncryption 函数 [UL C/C++]
  语法, 107
ULEnableStrongEncryption 函数 [UL C/C++]
  语法, 108
ULEnableTcpipSynchronization 函数 [UL C/C++]
  语法, 109
ULEnableTlsSynchronization 函数 [UL C/C++]
  语法, 110
ULEnableZlibSyncCompression 函数 [UL C/C++]
  语法, 111
ULExecuteNextSQLPassthroughScript [UL ESQL]
  语法, 260
ULExecuteSQLPassthroughScripts [UL ESQL]
  语法, 261
ULGetCollation_ 函数 (UltraLite C/C++)
  创建 UltraLite 数据库, 100
ULGetDatabaseID 函数 [UL ESQL]
  语法, 262
ULGetDatabaseProperty 函数 [UL ESQL]
  语法, 263
ULGetErrorParameterCount 函数 [UL ESQL]
  语法, 265
ULGetErrorParameter 函数 [UL ESQL]
  语法, 264
ULGetLastDownloadTime 函数 [UL ESQL]
  语法, 266
ULGetSynchResult 函数 [UL ESQL]
  语法, 268
ULGlobalAutoincUsage 函数 [UL ESQL]
  语法, 270
ULGrantConnectTo 函数 [UL ESQL]
  语法, 271
ULInitDatabaseManagerNoSQL 函数 [UL C/C++]
  语法, 113
  连接到数据库, 11
ULInitDatabaseManager 函数 [UL C/C++]
  语法, 112
  连接到数据库, 11
ULInitSynchInfo 函数 [UL ESQL]
  关于, 51
  语法, 272
ULIsSynchronizeMessage 函数 [UL ESQL]
  ActiveSync 用法, 86
  语法, 273
ULRegisterErrorCallback 函数 [UL C/C++]
  回调函数语法 (UltraLite C/C++), 94, 95
  语法, 114
ULRegisterSQLPassthroughCallback [UL C/C++]
  语法, 116
ULRegisterSynchronizationCallback [UL C/C++]
  语法, 118
ULResetLastDownloadTime 函数 [UL ESQL]
  语法, 274
ULRetrieveEncryptionKey 函数 [UL ESQL]

使用, 68
语法, 275
ULRevokeConnectFrom 函数 [UL ESQL]
语法, 276
ULRollbackPartialDownload 函数 [UL ESQL]
语法, 277
ulrt.lib
UltraLite C++ 开发, 28
ulrt11.dll
UltraLite C++ 开发, 28
ulrtc.lib
UltraLite C++ 开发, 28
ULSaveEncryptionKey 函数 [UL ESQL]
使用, 68
语法, 278
ULSetDatabaseID 函数 [UL ESQL]
语法, 279
ULSetDatabaseOptionString 函数 [UL ESQL]
语法, 280
ULSetDatabaseOptionULong 函数 [UL ESQL]
语法, 281
ULSetSynchInfo 函数 [UL ESQL]
使用, 71
语法, 282
ULSignalSyncIsComplete 函数 [UL ESQL]
语法, 283
ULSqlcaBase 类 [UltraLite C++ API]
Finalize 函数, 138
GetCA 函数, 139
GetParameter 函数, 139
GetParameterCount 函数, 140
GetSQLCode 函数, 140
GetSQLCount 函数, 140
GetSQLErrorOffset 函数, 141
Initialize 函数, 141
LastCodeOK 函数, 141
LastFetchOK 函数, 142
说明, 138
ULSqlcaWrap 函数
ULSqlcaWrap 类 [UltraLite C++ API], 143
ULSqlcaWrap 类 [UltraLite C++ API]
ULSqlcaWrap 函数, 143
~ULSqlcaWrap 函数, 144
说明, 143
ULSqlca 函数
ULSqlca 类 [UltraLite C++ API], 136
ULSqlca 类 [UltraLite C++ API]
ULSqlca 函数, 136
~ULSqlca 函数, 136
说明, 136
ULSynchronize 函数 [UL ESQL]
Palm OS 上的串行端口, 73
语法, 284
ULTable 对象
重新打开, 68
UltraLite
术语定义, 344
UltraLite_Connection_iface 类 [UltraLite C++ API]
CancelGetNotification 函数, 150
ChangeEncryptionKey 函数, 150
Checkpoint 函数, 151
Commit 函数, 151
CountUploadRows 函数, 151
CreateNotificationQueue 函数, 151
DeclareEvent 函数, 152
DestroyNotificationQueue 函数, 153
ExecuteNextSQLPassthroughScript 函数, 153
ExecuteSQLPassthroughScripts 函数, 154
GetConnectionNum 函数, 154
GetDatabaseID 函数, 154
GetDatabaseProperty(const ULValue &) 函数, 155
GetDatabaseProperty(ul_database_property_id) 函数, 154
GetLastDownloadTime 函数, 155
GetLastIdentity 函数, 155
GetNewUUID(GUID *) 函数, 156
GetNewUUID(p_ul_binary) 函数, 156
GetNotification 函数, 156
GetNotificationParameter 函数, 157
GetSchema 函数, 158
GetSqlca 函数, 158
GetSQLPassthroughScriptCount 函数, 158
GetSuspend 函数, 158
GetSynchResult 函数, 158
GetUtilityULValue 函数, 159
GlobalAutoincUsage 函数, 159
GrantConnectTo 函数, 159
InitSynchInfo(ul_synch_info_a *) 函数, 160
InitSynchInfo(ul_synch_info_w2 *) 函数, 160
OpenTable 函数, 160
OpenTableEx 函数, 161
OpenTableWithIndex 函数, 161
PrepareStatement 函数, 162
RegisterForEvent 函数, 162

ResetLastDownloadTime 函数, 163
RevokeConnectFrom 函数, 163
Rollback 函数, 163
RollbackPartialDownload 函数, 164
SendNotification 函数, 164
SetDatabaseID 函数, 165
SetDatabaseOption(const ULValue &, const ULValue &) 函数, 165
SetDatabaseOption(ul_database_option_id, const ULValue &) 函数, 165
SetSuspend 函数, 166
SetSynchInfo(ul_wchar const *, ul_synch_info_a *) 函数, 166
SetSynchInfo(ul_wchar const *, ul_synch_info_w2 *) 函数, 166
Shutdown 函数, 167
StartSynchronizationDelete 函数, 167
StopSynchronizationDelete 函数, 167
StrToUUID(GUID, const ULValue &) 函数, 168
StrToUUID(p_ul_binary, size_t, const ULValue &) 函数, 168
Synchronize(ul_synch_info_a *) 函数, 168
Synchronize(ul_synch_info_w2 *) 函数, 169
SynchronizeFromProfile 函数, 169
TriggerEvent 函数, 170
UUIDToStr(char *, size_t, const ULValue &) 函数, 170
UUIDToStr(ul_wchar *, size_t, const ULValue &) 函数, 171
ValidateDatabase 函数, 171
说明, 148
UltraLite_Connection 类 [UltraLite C++ API]
说明, 145
UltraLite_Cursor_iface 类 [UltraLite C++ API]
AfterLast 函数, 172
BeforeFirst 函数, 173
Delete 函数, 173
First 函数, 173
Get 函数, 173
GetRowCount 函数, 174
GetState 函数, 174
GetStreamReader 函数, 174
GetStreamWriter 函数, 175
GetSuspend 函数, 175
IsNull 函数, 175
Last 函数, 175
Next 函数, 176
Previous 函数, 176
Relative 函数, 176
Set 函数, 176
SetDefault 函数, 177
SetNull 函数, 177
SetSuspend 函数, 177
Update 函数, 178
UpdateBegin 函数, 178
说明, 172
UltraLite_DatabaseManager_iface 类 [UltraLite C++ API]
CreateDatabase 函数, 180
DropDatabase 函数, 181
OpenConnection 函数, 181
Shutdown 函数, 182
ValidateDatabase 函数, 182
说明, 180
UltraLite_DatabaseManager 类 [UltraLite C++ API]
说明, 179
UltraLite_DatabaseSchema_iface 类 [UltraLite C++ API]
GetCollationName 函数, 184
GetPublicationCount 函数, 184
GetPublicationID 函数, 185
GetPublicationName 函数, 185
GetTableCount 函数, 185
GetTableName 函数, 185
GetTableSchema 函数, 186
IsCaseSensitive 函数, 186
说明, 184
UltraLite_DatabaseSchema 类 [UltraLite C++ API]
说明, 183
UltraLite_IndexSchema_iface 类 [UltraLite C++ API]
GetColumnCount 函数, 188
GetColumnName 函数, 188
GetID 函数, 189
GetName 函数, 189
GetReferencedIndexName 函数, 189
GetReferencedTableName 函数, 190
GetTableName 函数, 190
IsColumnDescending 函数, 190
IsForeignKey 函数, 190
IsForeignKeyCheckOnCommit 函数, 191
IsForeignKeyNullable 函数, 191
IsPrimaryKey 函数, 191
IsUniqueIndex 函数, 192
IsUniqueKey 函数, 192

说明, 188
UltraLite_IndexSchema 类 [UltraLite C++ API]
说明, 187
UltraLite_PreparedStatement_iface 类 [UltraLite C++ API]
ExecuteQuery 函数, 194
ExecuteStatement 函数, 194
GetPlan(char *, size_t) 函数, 195
GetPlan(ul_wchar *, size_t) 函数, 195
GetSchema 函数, 195
GetStreamWriter 函数, 196
HasResultSet 函数, 196
SetParameter 函数, 196
SetParameterNull 函数, 197
说明, 194
UltraLite_PreparedStatement 类 [UltraLite C++ API]
说明, 193
UltraLite_ResultSet_iface 类 [UltraLite C++ API]
DeleteNamed 函数, 199
GetSchema 函数, 199
说明, 199
UltraLite_ResultSetSchema 类 [UltraLite C++ API]
说明, 200
UltraLite_ResultSet 类 [UltraLite C++ API]
说明, 198
UltraLite_RowSchema_iface 类 [UltraLite C++ API]
GetBaseColumnName 函数, 201
GetColumnCount 函数, 202
GetColumnID 函数, 202
GetColumnName 函数, 202
GetColumnPrecision 函数, 203
GetColumnScale 函数, 204
GetColumnSize 函数, 204
GetColumnSQLName 函数, 203
GetColumnSQLType 函数, 203
GetColumnType 函数, 205
说明, 201
UltraLite_SQLObject_iface 类 [UltraLite C++ API]
AddRef 函数, 206
GetConnection 函数, 206
GetIFace 函数, 207
Release 函数, 207
说明, 206
UltraLite_StreamReader_iface 类 [UltraLite C++ API]
GetByteChunk 函数, 209
GetLength 函数, 210
GetStringChunk 函数, 210
SetReadPosition 函数, 211
说明, 209
UltraLite_StreamReader 类 [UltraLite C++ API]
说明, 208
UltraLite_StreamWriter 类 [UltraLite C++ API]
说明, 212
UltraLite_Table_iface 类 [UltraLite C++ API]
DeleteAllRows 函数, 215
Find 函数, 216
FindBegin 函数, 216
FindFirst 函数, 216
FindLast 函数, 217
FindNext 函数, 217
FindPrevious 函数, 217
GetSchema 函数, 218
Insert 函数, 218
InsertBegin 函数, 218
Lookup 函数, 218
LookupBackward 函数, 219
LookupBegin 函数, 219
LookupForward 函数, 219
TruncateTable 函数, 220
说明, 215
UltraLite_TableSchema_iface 类 [UltraLite C++ API]
GetColumnDefault 函数, 223
GetGlobalAutoincPartitionSize 函数, 224
GetID 函数, 224
GetIndexCount 函数, 224
GetIndexName 函数, 225
GetIndexSchema 函数, 225
GetName 函数, 226
GetOptimalIndex 函数, 226
GetPrimaryKey 函数, 226
GetPublicationPredicate 函数, 226
GetUploadUnchangedRows 函数, 227
InPublication 函数, 227
IsColumnAutoinc 函数, 228
IsColumnCurrentDate 函数, 228
IsColumnCurrentTime 函数, 228
IsColumnCurrentTimestamp 函数, 229
IsColumnGlobalAutoinc 函数, 229
IsColumnInIndex 函数, 230
IsColumnNewUUID 函数, 230
IsColumnNullable 函数, 231
IsNeverSynchronized 函数, 231
说明, 223
UltraLite_TableSchema 类 [UltraLite C++ API]

说明, 221
UltraLite_Table 类 [UltraLite C++ API]
说明, 213
UltraLite_ 类
使用 UltraLite 命名空间, 10
UltraLite C/C++
INCLUDE 环境变量, 309
Palm OS, 68
Reopen 方法, 68
Table API 简介, 17
体系结构, 4
使用 SQL 的数据操作, 13
共同特性, 5
关于, 1
加密, 26
同步数据, 27
对 UltraLite 数据库进行模糊处理, 50
支持的平台, 3
教程, 309
访问模式信息, 23
验证, 25
UltraLite C/C++ 公用 API
按字母顺序排序的列表, 93
UltraLite C++
开发, 9
UltraLite C++ API
ul_sql_passthrough_status 结构, 125
ul_stream_error 结构, 126
ul_synch_info_a 结构, 127
ul_synch_info_w2 结构, 129
ul_synch_result 结构, 131
ul_synch_stats 结构, 132
ul_synch_status 结构, 133
ul_validate_data 结构, 135
ULSqlca 类, 136
ULSqlcaBase 类, 138
ULSqlcaWrap 类, 143
UltraLite_Connection 类, 145
UltraLite_Connection_iface 类, 148
UltraLite_Cursor_iface 类, 172
UltraLite_DatabaseManager 类, 179
UltraLite_DatabaseManager_iface 类, 180
UltraLite_DatabaseSchema 类, 183
UltraLite_DatabaseSchema_iface 类, 184
UltraLite_IndexSchema 类, 187
UltraLite_IndexSchema_iface 类, 188
UltraLite_PreparedStatement 类, 193
UltraLite_PreparedStatement_iface 类, 194
UltraLite_ResultSet 类, 198
UltraLite_ResultSet_iface 类, 199
UltraLite_ResultSetSchema 类, 200
UltraLite_RowSchema_iface 类, 201
UltraLite_SQLObject_iface 类, 206
UltraLite_StreamReader 类, 208
UltraLite_StreamReader_iface 类, 209
UltraLite_StreamWriter 类, 212
UltraLite_Table 类, 213
UltraLite_Table_iface 类, 215
UltraLite_TableSchema 类, 221
UltraLite_TableSchema_iface 类, 223
ULValue 类, 232
UltraLite ODBC 接口
SQLAllocHandle 函数, 286
SQLBindCol 函数, 287
SQLBindParameter 函数, 288
SQLConnect 函数, 289
SQLDescribeCol 函数, 290
SQLEndTran 函数, 292
SQLExecDirect 函数, 293
SQLExecute 函数, 294
SQLFetch 函数, 295
SQLFetchScroll 函数, 296
SQLFreeHandle 函数, 297
SQLGetCursorName 函数, 298
SQLGetData 函数, 299
SQLGetDiagRec 函数, 300
SQLGetInfo 函数, 301
SQLNumResultCols 函数, 302
SQLPrepare 函数, 303
SQLRowCount 函数, 304
SQLSetConnectionName 函数, 305
SQLSetCursorName 函数, 306
SQLSetSuspend 函数, 307
SQLSynchronize 函数, 308
按字母顺序排序的列表, 285
UltraLite SQL 以空值终止的 TCHAR 字符串数据类型
关于, 36
UltraLite SQL 以空值终止的 UNICODE 字符串数据类型
关于, 36
UltraLite SQL 以空值终止的 WCHAR 字符串数据类型
关于, 36

UltraLite SQL 以空值终止的宽字符串数据类型
关于, 36
UltraLite 插件
CodeWarrior 使用, 65
CodeWarrior 安装, 62
CodeWarrior 项目创建, 63
CodeWarrior 项目转换, 64
用于加密同步的 CodeWarrior 库, 65
UltraLite 命名空间
UltraLite C++, 10
UltraLite 嵌入式 SQL
主机变量, 35
使用, 250
函数, 250
同步, 51
开发应用程序, 29
授权, 31
构建 CustDB 应用程序, 79
游标, 45
读取数据, 44
UltraLite 嵌入式 SQL DT_BINARY 数据类型
关于, 38
UltraLite 嵌入式 SQL DT_LONGBINARY 数据类型
关于, 38
UltraLite 嵌入式 SQL DT_LONGVARCHAR 数据类型
关于, 37
UltraLite 嵌入式 SQL 的 16 位有符号整数数据类型
关于, 36
UltraLite 嵌入式 SQL 的 32 位有符号整数数据类型
关于, 36
UltraLite 嵌入式 SQL 的 4 字节浮点数据类型
关于, 36
UltraLite 嵌入式 SQL 的 8 字节浮点数据类型
关于, 36
UltraLite 嵌入式 SQL 的二进制数据类型
关于, 37
UltraLite 嵌入式 SQL 的十进制数据类型
关于, 36
UltraLite 嵌入式 SQL 的时间戳结构数据类型
关于, 37
UltraLite 嵌入式 SQL 的压缩十进制数据类型
关于, 36
UltraLite 嵌入式 SQL 的字符串数据类型
关于, 36
可变长度, 37
固定长度, 36
UltraLite 嵌入式 SQL 库函数
GetSQLPassthroughScriptCount, 267
MLFileTransfer, 97
ULChangeEncryptionKey, 255
ULCheckpoint, 256
ULClearEncryptionKey, 257
ULCountUploadRows, 258
ULCreateDatabase, 100
ULDropDatabase, 259
ULEnableEccSyncEncryption, 102
ULEnableFIPSStrongEncryption, 103
ULEnableHttpsSynchronization, 105
ULEnableHttpSynchronization, 104
ULEnableRsaFipsSyncEncryption, 106
ULEnableRsaSyncEncryption, 107
ULEnableStrongEncryption, 108
ULEnableTcpipSynchronization, 109
ULEnableTlsSynchronization, 110
ULEnableZlibSyncCompression, 111
ULExecuteNextSQLPassthroughScript, 260
ULExecuteSQLPassthroughScripts, 261
ULGetDatabaseID, 262
ULGetDatabaseProperty, 263
ULGetErrorParameter, 264
ULGetErrorParameterCount, 265
ULGetLastDownloadTime, 266
ULGetSynchResult, 268
ULGlobalAutoincUsage, 270
ULGrantConnectTo, 271
ULInitSynchInfo, 272
ULIsSynchronizeMessage, 273
ULResetLastDownloadTime, 274
ULRetrieveEncryptionKey, 275
ULRevokeConnectFrom, 276
ULRollbackPartialDownload 函数, 277
ULSaveEncryptionKey, 278
ULSetDatabaseID, 279
ULSetDatabaseOptionString, 280
ULSetDatabaseOptionULong, 281
ULSetSynchInfo, 282
ULSignalSyncIsComplete, 283
ULSynchronize, 284
UltraLite 嵌入式 SQL 以 NULL 终止的字符串数据类型
关于, 36
UltraLite 数据库
UltraLite C++ 信息访问, 23

UltraLite for Windows Mobile, 81
在 Palm OS 上部署, 74
在 UltraLite C++ 中连接, 11
在嵌入式 SQL 中加密, 50
UltraLite 项目
CodeWarrior, 63
UltraLite 应用程序
C/C++ 应用程序的同步, 27
UltraLite 运行时
UltraLite C++ 库, 28
术语定义, 344
部署 Windows Mobile 库, 82
ULValue(bool) 函数
ULValue 类 [UltraLite C++ API], 240
ULValue(const char *, size_t) 函数
ULValue 类 [UltraLite C++ API], 244
ULValue(const char *) 函数
ULValue 类 [UltraLite C++ API], 244
ULValue(const p_ul_binary) 函数
ULValue 类 [UltraLite C++ API], 243
ULValue(const ul_s_big &) 函数
ULValue 类 [UltraLite C++ API], 243
ULValue(const ul_u_big &) 函数
ULValue 类 [UltraLite C++ API], 243
ULValue(const ul_wchar *, size_t) 函数
ULValue 类 [UltraLite C++ API], 245
ULValue(const ul_wchar *) 函数
ULValue 类 [UltraLite C++ API], 244
ULValue(const ULValue &) 函数
ULValue 类 [UltraLite C++ API], 239
ULValue(DECL_DATETIME &) 函数
ULValue 类 [UltraLite C++ API], 243
ULValue(double) 函数
ULValue 类 [UltraLite C++ API], 241
ULValue(float) 函数
ULValue 类 [UltraLite C++ API], 241
ULValue(GUID &) 函数
ULValue 类 [UltraLite C++ API], 245
ULValue(int) 函数
ULValue 类 [UltraLite C++ API], 241
ULValue(long) 函数
ULValue 类 [UltraLite C++ API], 240
ULValue(short) 函数
ULValue 类 [UltraLite C++ API], 240
ULValue(unsigned char) 函数
ULValue 类 [UltraLite C++ API], 242
ULValue(unsigned int) 函数
ULValue 类 [UltraLite C++ API], 241
ULValue(unsigned long) 函数
ULValue 类 [UltraLite C++ API], 242
ULValue(unsigned short) 函数
ULValue 类 [UltraLite C++ API], 242
ULValue 函数
ULValue 类 [UltraLite C++ API], 239
ULValue 类 [UltraLite C++ API]
bool 运算符, 246
DECL_DATETIME 运算符, 245
double 运算符, 246
float 运算符, 246
GetBinary(p_ul_binary, size_t) 函数, 234
GetBinary(ul_byte *, size_t, size_t *) 函数, 234
GetBinaryLength 函数, 235
GetCombinedStringItem(ul_u_short, char *, size_t) 函数, 235
GetCombinedStringItem(ul_u_short, ul_wchar *, size_t) 函数, 235
GetString(char *, size_t) 函数, 236
GetString(ul_wchar *, size_t) 函数, 236
GetStringLength 函数, 236
GUID 运算符, 245
InDatabase 函数, 237
int 运算符, 246
IsNull 函数, 237
long 运算符, 246
operator= 运算符, 248
SetBinary 函数, 238
SetString(const char *, size_t) 函数, 238
SetString(const ul_wchar *, size_t) 函数, 238
short 运算符, 247
StringCompare 函数, 239
ul_s_big 运算符, 247
ul_u_big 运算符, 247
ULValue 函数, 239
ULValue(bool) 函数, 240
ULValue(const char *) 函数, 244
ULValue(const char *, size_t) 函数, 244
ULValue(const p_ul_binary) 函数, 243
ULValue(const ul_s_big &) 函数, 243
ULValue(const ul_u_big &) 函数, 243
ULValue(const ul_wchar *) 函数, 244
ULValue(const ul_wchar *, size_t) 函数, 245
ULValue(const ULValue &) 函数, 239
ULValue(DECL_DATETIME &) 函数, 243
ULValue(double) 函数, 241

ULValue(float) 函数, 241
ULValue(GUID &) 函数, 245
ULValue(int) 函数, 241
ULValue(long) 函数, 240
ULValue(short) 函数, 240
ULValue(unsigned char) 函数, 242
ULValue(unsigned int) 函数, 241
ULValue(unsigned long) 函数, 242
ULValue(unsigned short) 函数, 242
unsigned char 运算符, 247
unsigned int 运算符, 247
unsigned long 运算符, 247
unsigned short 运算符, 248
~ULValue 运算符, 248
说明, 232
UNDER_CE 编译器指令
关于, 120
UNDER_PALM_OS 编译器指令
关于, 121
UNICODE 字符
UltraLite C++ 库, 28
unsigned char 运算符
ULValue 类 [UltraLite C++ API], 247
unsigned int 运算符
ULValue 类 [UltraLite C++ API], 247
unsigned long 运算符
ULValue 类 [UltraLite C++ API], 247
unsigned short 运算符
ULValue 类 [UltraLite C++ API], 248
UpdateBegin 函数
UltraLite_Cursor_iface 类 [UltraLite C++ API], 178
Update 函数
UltraLite_Cursor_iface 类 [UltraLite C++ API], 178
UUIDToStr(char *, size_t, const ULValue &) 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 170
UUIDToStr(ul_wchar *, size_t, const ULValue &) 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 171

V

ValidateDatabase 函数
UltraLite_Connection_iface 类 [UltraLite C++ API], 171
UltraLite_DatabaseManager_iface 类 [UltraLite C++ API], 182
Visual C++
UltraLite for Windows Mobile 开发, 77

W

WindowProc 函数
ActiveSync, 273
ActiveSync 用法, 86
Windows
UltraLite 运行时库, 28
术语定义, 346
Windows Mobile
UltraLite 使用 eMbedded Visual C++ 构建 CustDB 应用程序, 79
UltraLite 同步菜单控件, 88
UltraLite 平台要求, 77
UltraLite 应用程序同步, 86
UltraLite 应用程序开发概述, 77
UltraLite 类名, 84
UltraLite 运行时库, 28
使用 ActiveSync 的 UltraLite 同步, 83
术语定义, 346
winsock.lib
UltraLite Windows Mobile 应用程序, 77
外表
术语定义, 344
外部登录
术语定义, 344
外键
术语定义, 344
外键约束
术语定义, 345
外连接
术语定义, 345
完全备份
术语定义, 345
完整性
术语定义, 345
网关
术语定义, 345
网络服务器
术语定义, 345
网络协议
UltraLite for Windows Mobile, 88
术语定义, 346
唯一约束

术语定义, 346
维护版本
术语定义, 346
未提交的事务
UltraLite 嵌入式 SQL, 53
位数组
术语定义, 346
谓语
术语定义, 346
文档
SQL Anywhere, x
约定, xi
文件定义数据库
术语定义, 346
物理索引
术语定义, 347

X

系统表
术语定义, 347
系统对象
术语定义, 347
系统视图
术语定义, 347
下载
术语定义, 347
线程
UltraLite C++ API 多线程应用程序, 12
UltraLite 嵌入式 SQL, 34
相对偏移
UltraLite C++ Table API, 17
相关名
术语定义, 347
项目
术语定义, 347
消息存储库
术语定义, 347
消息类型
术语定义, 347
消息日志
术语定义, 348
消息系统
术语定义, 348
协议
UltraLite for Windows Mobile, 88
卸载
术语定义, 348
新闻组
技术支持, xv
行
UltraLite C++ 表浏览, 17
使用 C++ API 进行 UltraLite 删除, 21
使用 C++ API 进行 UltraLite 插入, 21
使用 C++ API 进行 UltraLite 更新, 20
在 UltraLite C++ API 教程中访问, 315
当前的 UltraLite C++ 表访问, 18
行级触发器
术语定义, 331
性能
UltraLite 使用 DLL 以节约内存, 78
UltraLite 显式命名数据库, 6
UltraLite 避免加密密钥重复输入, 68
使用 INSERT 语句的 UltraLite, 53
性能统计
术语定义, 348

Y

业务规则
术语定义, 348
依赖性
UltraLite 嵌入式 SQL, 58
疑难解答
UltraLite 开发, 53
上一同步, 268
为 UltraLite 在 SQL 预处理器中使用表达式, 40
使用嵌入式 SQL 的 UltraLite 同步, 53
新闻组, xv
引用对象
术语定义, 348
应用程序
在 Palm OS 上部署 UltraLite, 74
构建 UltraLite 嵌入式 SQL , 58
编写 UltraLite 嵌入式 SQL, 29
编译 UltraLite 嵌入式 SQL, 58
预处理 UltraLite 嵌入式 SQL, 58
用户定义数据类型
术语定义, 349
用户验证
UltraLite C++ 开发, 25
UltraLite ULGrantConnectTo（UltraLite 嵌入式 SQL）, 271
UltraLite 嵌入式 SQL, 276
UltraLite 嵌入式 SQL 应用程序, 48
UltraLite 嵌入式 SQL 授权方法, 271

UltraLite 嵌入式 SQL 撤消方法, 276
游标
UltraLite 保存状态, 68
UltraLite 定位, 46
UltraLite 恢复状态, 68
UltraLite 更新后定位, 47
UltraLite 行的顺序, 46
UltraLite 读取多行, 45
UltraLite 重新定位, 47
术语定义, 349
游标结果集
术语定义, 349
游标位置
术语定义, 349
语句级触发器
术语定义, 349
域
术语定义, 349
预处理
UltraLite 嵌入式 SQL 应用程序, 58
UltraLite 嵌入式 SQL 开发工具设置, 58
预订
术语定义, 350
预准备语句
UltraLite C++, 13
元数据
术语定义, 350
原子事务
术语定义, 350
远程 ID
术语定义, 350
远程数据库
术语定义, 350
约定
命令 shell, xiii
命令提示符, xiii
文档, xi
文档中的文件名, xii
约束
术语定义, 351
运行时库
UltraLite C++, 28
UltraLite for C++, 28
Windows Mobile, 120
用于 Windows Mobile 的 UltraLite 应用程序, 78
运营公司
术语定义, 351

Z

增量备份
术语定义, 351
争用
术语定义, 351
正则表达式
术语定义, 351
支持
新闻组, xv
支持的平台
UltraLite C++, 3
直方图
术语定义, 351
直接行处理
术语定义, 351
值
UltraLite C++ API 访问于, 18
指令
UltraLite 应用程序, 119
指示符变量
UltraLite NULL, 43
UltraLite 嵌入式 SQL, 42
主表
术语定义, 352
主机变量
UltraLite 作用域, 38
UltraLite 使用, 38
UltraLite 嵌入式 SQL, 35
UltraLite 嵌入式 SQL 表达式, 40
主键
术语定义, 352
主键约束
术语定义, 352
主题
图标, xiii
注册表
ClientParms 注册表条目, 71
转换
UltraLite C++ API 数据类型, 19
子查询
术语定义, 352
自然连接
术语定义, 340
字符串
UL_TEXT 宏, 120
术语定义, 352
字符集

术语定义, 352

