SYBASE iAnywhere

SQL Anywhere® 服务器
编程

2009 年 2 月

11.0.1 版

版权和商标

目录

关于本手册

主题

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。

读者

本手册适用于编写直接使用 SQL Anywhere 接口之一的程序的应用程序开发人员。

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`。

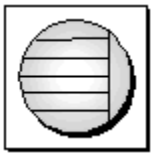
- **环境变量** 本文档介绍设置环境变量。在 Windows shell 中，环境变量使用语法 *%ENVVAR%* 来指定。在 Unix shell 中，环境变量使用语法 *$ENVVAR* 或 *${ENVVAR}* 来指定。

图标

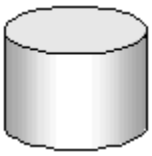
本文档中使用了下列图标。

- 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。

使用 SQL Anywhere 编程简介

本节向您介绍使用 SQL Anywhere 编程。

SQL Anywhere 数据访问编程接口

目录

SQL Anywhere .NET 支持

ADO.NET 是 Microsoft 的 ODBC、OLE DB 和 ADO 系列中最新的数据访问 API。它是 Microsoft .NET Framework 首选的数据访问组件，可用于访问关系数据库系统。

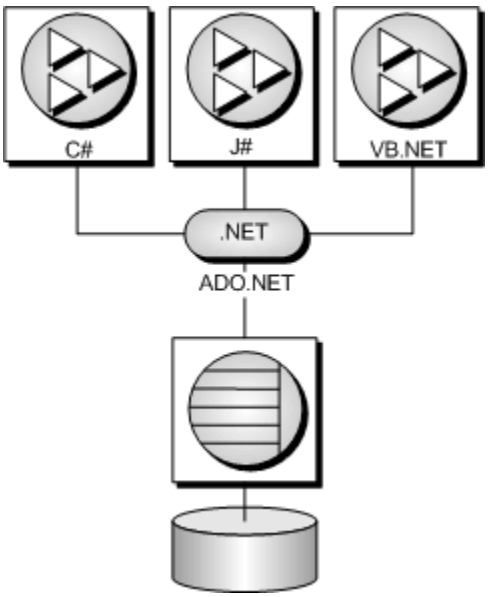
SQL Anywhere .NET 数据提供程序实现了 iAnywhere.Data.SQLAnywhere 命名空间，允许您使用支持 .NET 的任何语言（例如 C# 和 Visual Basic .NET）编写程序，并从 SQL Anywhere 数据库访问数据。

有关 .NET 数据访问的一般信息，请参见 Microsoft 的 .NET Data Access Architecture Guide。

ADO.NET 应用程序

您可以使用面向对象的语言开发 Internet 和 intranet 应用程序，然后使用 ADO.NET 数据提供程序将这些应用程序连接到 SQL Anywhere。

将此提供程序与内置的 XML 和 Web 服务功能、用于 MobiLink 同步的 .NET 脚本编写功能以及用于开发手持式数据库应用程序的 UltraLite .NET 组件组合起来，这样 SQL Anywhere 即可与 .NET Framework 进行集成。



另请参见

- “SQL Anywhere .NET 数据提供程序”第 103 页
- “iAnywhere.Data.SQLAnywhere 命名空间 (.NET 2.0)”一节第 168 页
- “教程：使用 SQL Anywhere .NET 数据提供程序”第 137 页

SQL Anywhere OLE DB 和 ADO 支持

SQL Anywhere 中附带了一个供 OLE DB 和 ADO 程序员使用的 OLE DB 提供程序。

OLE DB 是 Microsoft 开发的一组组件对象模型（Component Object Model，简称 COM）接口，它们为应用程序访问不同信息源中存储的数据提供了统一访问接口，并且还提供了实现其它数据库服务的能力。这些接口所支持的 DBMS 功能数与数据存储相符，从而使数据存储能够共享它的数据。

ADO 是通过 OLE DB 系统接口以编程方式访问、编辑以及更新多种数据源的对象模型。ADO 也是由 Microsoft 开发的。大多数使用 OLE DB 编程接口的开发人员在使用该编程接口时都是编写 ADO API 代码，而不是直接编写 OLE DB API 代码。

不要将 ADO 接口同 ADO.NET 混淆。ADO.NET 是一个单独的接口。有关详细信息，请参见“SQL Anywhere .NET 支持”一节第 4 页。

有关 OLE DB 和 ADO 编程的文档，请参见 Microsoft Developer Network。有关 OLE DB 和 ADO 开发的特定于 SQL Anywhere 的信息，请参见“SQL Anywhere OLE DB 和 ADO 开发”第 431 页。

SQL Anywhere ODBC 支持

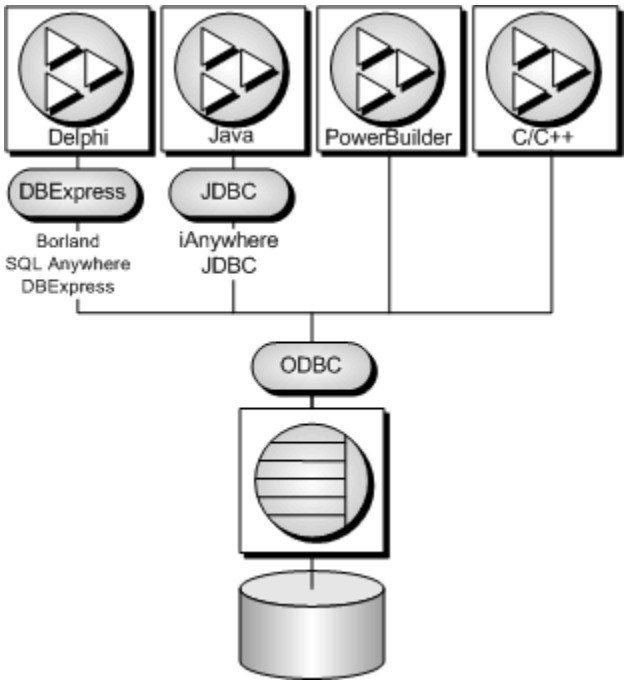
ODBC（开放式数据库连接，Open Database Connectivity）是 Microsoft 开发的一个标准调用层接口（Call Level Interface，简称 CLI）。它基于 [SQL 访问组 CLI] 规范。可以对任何提供 ODBC 驱动程序的数据源运行 ODBC 应用程序。如果您希望应用程序能够向具有 ODBC 驱动程序的其它数据源移植，将 ODBC 作为编程接口是很好的选择。

ODBC 是一个低层接口。几乎所有的 SQL Anywhere 功能都可用于此接口。ODBC 在 Windows 操作系统（Windows Mobile 除外）中以 DLL 形式提供。在 Unix 中则以库的形式提供。

ODBC 的主要文档是 [Microsoft ODBC 软件开发工具包]。

ODBC 应用程序

您可以使用多种开发工具和编程语言开发应用程序（如下图所示），并使用 ODBC API 访问 SQL Anywhere 数据库服务器。



例如，在与 SQL Anywhere 一起提供的应用程序中，InfoMaker 和 PowerDesigner Physical Data Model 使用 ODBC 连接到数据库。

另请参见

- “SQL Anywhere ODBC API”第 445 页

SQL Anywhere JDBC 支持

JDBC 是 Java 应用程序的调用层接口。JDBC 是由 Sun Microsystems 开发的，它给 Java 程序员提供了与各种关系数据库的统一接口，并且为创建各种更高级别的工具和接口提供了一个公共基础。JDBC 现在是 Java 的标准组成部分并被包括在 JDK 中。

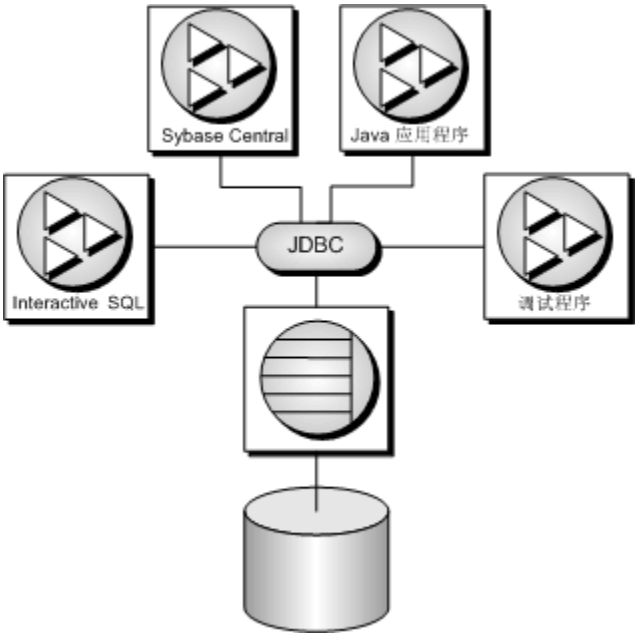
SQL Anywhere 包括一个纯 Java 的 JDBC 驱动程序，名为 jConnect。它还提供 iAnywhere JDBC 驱动程序，后者是一个类型 2 驱动程序。在“SQL Anywhere JDBC 驱动程序”第 481 页中对此二者都进行了介绍。

有关选择驱动程序的信息，请参见“选择 JDBC 驱动程序”一节第 482 页。

除了将 JDBC 用作客户端应用程序编程接口外，您还可以通过在数据库中使用 Java 来在数据库服务器内使用 JDBC 访问数据。

JDBC 应用程序

您可以开发使用 JDBC API 连接到 SQL Anywhere 的 Java 应用程序。与 SQL Anywhere 一起提供的某些应用程序使用 JDBC，如调试程序、Sybase Central 和 Interactive SQL。



Java 和 JDBC 也是用于开发 UltraLite 应用程序的重要编程语言。

另请参见

- “选择 JDBC 驱动程序”一节第 482 页
- “SQL Anywhere JDBC 驱动程序”第 481 页

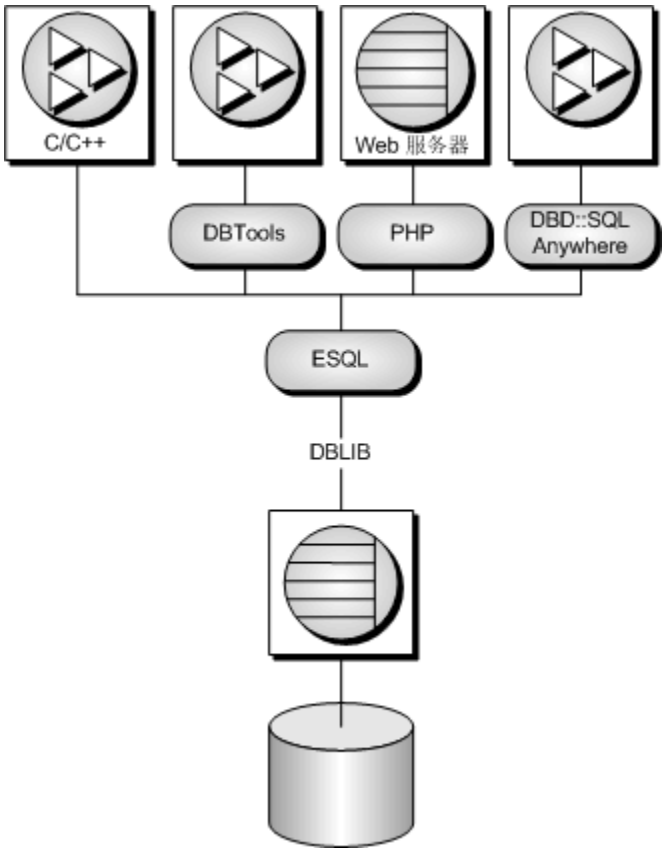
SQL Anywhere 嵌入式 SQL

嵌入在 C 或 C++ 资源文件中的 SQL 语句称为嵌入式 SQL。预处理器将这些语句翻译为对运行时库的调用。嵌入式 SQL 是一种 ISO/ANSI 和 IBM 标准。

嵌入式 SQL 能够向其它数据库和其它环境移植，并且它在所有操作环境中的功能是等同的。它是一个综合的低层接口，可提供某种数据库产品的所有可用功能。使用嵌入式 SQL 需要您了解 C 或 C++ 编程语言。

嵌入式 SQL 应用程序

您可以开发使用 SQL Anywhere 嵌入式 SQL 接口访问 SQL Anywhere 服务器的 C 或 C++ 应用程序。例如，命令行数据库工具就是以此方式开发的应用程序。



另请参见

- “SQL Anywhere 嵌入式 SQL”第 509 页

SQL Anywhere C 语言支持

SQL Anywhere C API 是 C/C++ 语言的数据访问 API。C API 规范定义了一组函数、变量和约定，这些函数、变量和约定提供了一致的数据库接口,该接口独立于实际所用数据库。使用 SQL Anywhere C API，您的 C/C++ 应用程序可以直接访问 SQL Anywhere 数据库服务器。

另请参见

- “SQL Anywhere C API 参考”第 589 页

SQL Anywhere Perl DBI 支持

DBD::SQLAnywhere 是用于 DBI 的 SQL Anywhere 数据库驱动程序，DBI 是用于 Perl 语言的数据访问 API。DBI API 规范定义了一组函数、变量和约定，这些函数、变量和约定提供了一致的数据库接口,该接口独立于实际所用数据库。使用 DBI 和 DBD::SQLAnywhere，您的 Perl 脚本可以直接访问 SQL Anywhere 数据库服务器。

另请参见

- “SQL Anywhere Perl DBD::SQLAnywhere DBI 模块”第 693 页

SQL Anywhere Python 支持

SQL Anywhere Python 数据库接口 (sqlanydb) 是用于 Python 语言的数据访问 API。Python 数据库 API 规范定义了一组方法，这些方法提供了一致的数据库接口,该接口独立于实际所用数据库。使用 sqlanydb 模块，您的 Python 脚本可以直接访问 SQL Anywhere 数据库服务器。

另请参见

- “SQL Anywhere Python 数据库支持”第 703 页

SQL Anywhere PHP 支持

PHP 提供了从许多常用数据库检索信息的能力。SQL Anywhere 包括一个可提供从 PHP 访问 SQL Anywhere 数据库的方法的模块。您可以使用 PHP 语言从 SQL Anywhere 数据库检索信息，并在您自己的 Web 站点上提供动态 Web 内容。

SQL Anywhere PHP 模块提供了一种从 PHP 访问数据库的内在方法。与其它 PHP 数据访问技术相比，您可能更喜欢此技术，因为它很简单，而且可帮助您避免使用其它技术可能发生的系统资源泄露。

另请参见

- “SQL Anywhere PHP API”第 711 页

SQL Anywhere Ruby 支持

SQL Anywher 支持三种不同的 Ruby API。第一种是 SQL Anywhere Ruby API。此 API 通过 SQL Anywhere C API 公开的接口提供了一个 Ruby 包装。第二种是对 ActiveRecord 的支持，ActiveRecord 是一个对象相关映射程序，作为 Ruby on Rails Web 开发框架的一部分而为人所知。第三种是对 Ruby DBI 的支持。SQL Anywhere 提供了 Ruby 数据库驱动程序（Ruby Database Driver，简称 DBD），可与 DBI 配合使用。

另请参见

- “SQL Anywhere for Ruby”第 777 页

SQL Anywhere Web 服务支持

SQL Anywhere Web 服务为客户端应用程序提供替代接口，来替代诸如 JDBC 和 ODBC 之类的数据访问 API。Web 服务可以从以多种语言编写并运行在多个平台上的客户端应用程序进行访问。即使 Perl 和 Python 之类的常见脚本编写语言都可以提供对 Web 服务的访问。

另请参见

- “SQL Anywhere Web 服务”第 817 页

Sybase Open Client 支持

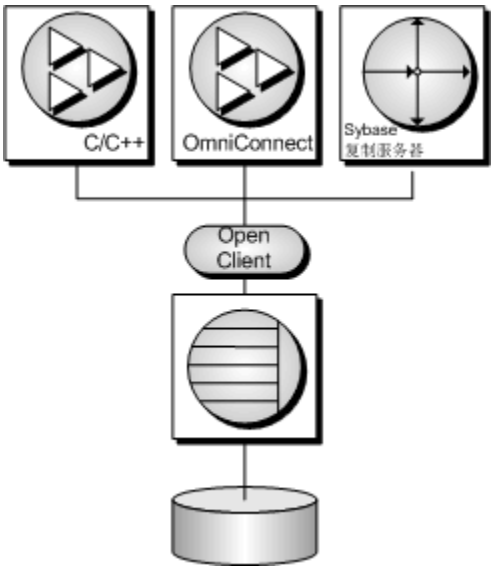
Sybase Open Client 为客户端应用程序、第三方产品及其它 Sybase 产品提供了与 SQL Anywhere 及其它 Open Server 进行通信所需要的接口。

何时使用 Open Client

如果您对 Adaptive Server Enterprise 的兼容性感到担心，或者您要使用其它支持 Open Client 接口的 Sybase 产品（如复制服务器），那么您应考虑使用 Open Client 接口。

Open Client 应用程序

您可以用 C 或 C++ 开发应用程序，然后使用 Open Client API 将这些应用程序连接到 SQL Anywhere。其它 Sybase 应用程序（例如 OmniConnect 或复制服务器）使用 Open Client。Open Client API 也是 Sybase Adaptive Server Enterprise 支持的接口。



另请参见

- “将 SQL Anywhere 用作 Open Server” 《SQL Anywhere 服务器 - 数据库管理》

SQL Anywhere Explorer

目录

SQL Anywhere Explorer 简介

SQL Anywhere Explorer 是能够从 Visual Studio 连接到 SQL Anywhere 和 UltraLite 数据库的组件。

有关 SQL Anywhere Explorer for UltraLite 的详细信息，请参见“SQL Anywhere Explorer for UltraLite”《UltraLite - .NET 编程》。

使用 SQL Anywhere Explorer

在 Visual Studio 中，可以使用 SQL Anywhere Explorer 创建与 SQL Anywhere 数据库的连接。在连接到数据库后，您可以：

- 查看数据库中的表、视图和过程
- 查看存储在表和视图中的数据
- 设计程序打开与 SQL Anywhere 数据库的连接，或者检索和操作数据
- 将数据库对象拖放到 C# 或 Visual Basic 代码或窗体，使 IDE 自动生成引用所选对象的代码

还可以在 [**Tools**] 菜单中选择相应命令，从 Visual Studio 打开 Sybase Central 和 Interactive SQL。

安装说明

如果在已装有 Visual Studio 的 Windows 计算机上安装 SQL Anywhere 软件，安装过程会检测到 Visual Studio 的存在，并运行必要的集成步骤。如果在安装 SQL Anywhere 之后安装 Visual Studio，或安装新版本的 Visual Studio，必须在命令提示符处完成将 SQL Anywhere 与 Visual Studio 集成的过程，操作步骤如下：

- 确保 Visual Studio 未运行。
- 在命令提示符处运行 *install-dir\Assembly\v2\SetupVSPackage.exe /install*。

在 Visual Studio 中使用数据库连接

使用 SQL Anywhere Explorer，在 [**Data Connections**] 节点显示 SQL Anywhere 数据库连接。必须创建数据连接才能查看表和视图中的数据。

您可以在 SQL Anywhere Explorer 中列出数据库表、视图、存储过程和函数并扩展各个表以列出它们的列。Visual Studio [**Properties**] 窗格中会显示在 [SQL Anywhere Explorer] 窗口中选择的对象的属性。

♦ 在 Visual Studio 中添加 SQL Anywhere 数据库连接

1. 选择 [**View**] » [**SQL Anywhere Explorer**]，打开 SQL Anywhere Explorer。
2. 在 [SQL Anywhere Explorer] 窗口中右击 [**Data Connections**]，然后选择 [**Add Connection**]。
3. 选择 [**SQL Anywhere**]，然后单击 [**OK**]。
4. 输入连接到数据库所需的适当值。
5. 单击 [**OK**]。

   这样就建立了一个到数据库的连接，且该连接被添加到 [**Data Connections**] 列表。

♦ **从 Visual Studio 中删除 SQL Anywhere 数据库连接**

1. 选择 [**View**] » [**SQL Anywhere Explorer**]，打开 SQL Anywhere Explorer。
2. 在 [SQL Anywhere Explorer] 窗口中右击要删除的数据连接，然后选择 [**Delete**]。

   这样该连接即从 [SQL Anywhere Explorer] 窗口删除。

配置 SQL Anywhere Explorer

Visual Studio [**Options**] 窗口包括可用来配置 SQL Anywhere Explorer 的设置。

♦ **访问 SQL Anywhere Explorer 选项**

1. 从 Visual Studio [**Tools**] 菜单中选择 [**Options**]。
2. 在 [**Options**] 窗口的左窗格中展开 [**SQL Anywhere**]。
3. 单击 [**General**] 可按需要配置 SQL Anywhere Explorer 常规选项。

   限制发送到输出窗口的查询结果　指定 [**Output**] 窗口中显示的行数。缺省值是 500。

   在服务器浏览器中显示系统对象　如果要在 Microsoft Server Explorer 中查看系统对象，请选中此选项。这不是 SQL Anywhere Explorer 选项，而是 Server Explorer 选项。系统对象包括归 "dbo" 用户所有的对象。

   排序对象　选择根据对象名称或对象所有者名称在 [SQL Anywhere Explorer] 窗口中对对象进行排序。

   将表或视图放到设计器时生成 UI 代码　为您拖放到 Windows 窗体设计器的表或视图生成代码。

   生成用于数据适配器的 Insert、Update 和 Delete 命令　将表或视图拖放到 C# 或 Visual Basic 文档时，生成用于数据适配器的 INSERT、UPDATE 和 DELETE 命令。

   生成数据适配器的表映射　将表拖放到 C# 或 Visual Basic 文档时，生成数据适配器的表映射。

使用 SQL Anywhere Explorer 添加数据库对象

在 Visual Studio 中，当您将某些数据库对象从 SQL Anywhere Explorer 拖放到 Visual Studio 设计器时，IDE 自动创建引用所选对象的新组件。可以通过在 Visual Studio 中选择 [**Tools**] » [**Options**]并打开 SQL Anywhere 节点，配置拖放操作的设置。

例如，如果将一个存储过程从 SQL Anywhere Explorer 拖放到 Windows 窗体，IDE 会自动创建预配置为调用该存储过程的 Command 对象。

下表列出了可以从 SQL Anywhere Explorer 拖放的对象，并介绍了将对象拖放到 Visual Studio 窗体设计器或代码编辑器时创建的组件。

项	结果
数据连接	创建数据连接。
表	创建适配器。
视图	创建适配器。
存储过程或函数	创建命令。

♦ 使用 SQL Anywhere Explorer 创建新数据组件

1. 打开想要添加数据组件的窗体或类。
2. 在 SQL Anywhere Explorer 中，选择要使用的对象。
3. 将对象从 SQL Anywhere Explorer 拖放到窗体设计器或代码编辑器。

使用 SQL Anywhere Explorer 处理表

使用 SQL Anywhere Explorer，可在 Visual Studio 中查看 SQL Anywhere 数据库中表和视图的属性和数据。

♦ 在 Visual Studio 中查看表或查看数据

1. 使用 SQL Anywhere Explorer 连接到 SQL Anywhere 数据库。
2. 在 [SQL Anywhere Explorer] 对话框中展开数据库，然后根据要查看的对象，将视图或将表展开。
3. 右击表或视图，然后选择 [**Retrieve Data**]。

   所选的表或视图中的数据显示在 Visual Studio 的 [**Output**] 窗口中。

使用 SQL Anywhere Explorer 处理过程和函数

如果对存储过程进行更改，则可以从 SQL Anywhere Explorer 刷新该过程以获取对列或参数的最新更改。

♦ 在 Visual Studio 中刷新过程

1. 连接到 SQL Anywhere 数据库。
2. 右击该过程，然后选择 [**Refresh**]。

   如果数据库中的过程进行了任何更改，则更新参数和列。

在应用程序中使用 SQL

目录

在应用程序中执行 SQL 语句

在应用程序中使用 SQL 语句的方式取决于您使用的应用程序开发工具和编程接口。

- **ADO.NET** 您可以使用多种 ADO.NET 对象执行 SQL 语句。SACommand 对象就是一个示例：

```
SACommand cmd = new SACommand(
     "DELETE FROM Employees WHERE EmployeeID = 105", conn );
cmd.ExecuteNonQuery();
```

  请参见“SQL Anywhere .NET 数据提供程序”第 103 页。

- **ODBC** 如果您直接对 ODBC 编程接口编写代码，那么您的 SQL 语句将以函数调用的形式出现。例如，下面的 C 函数调用将执行 DELETE 语句：

```
SQLExecDirect( stmt,
    "DELETE FROM Employees
     WHERE EmployeeID = 105",
    SQL_NTS );
```

  请参见“SQL Anywhere ODBC API”第 445 页。

- **JDBC** 如果使用的是 JDBC 编程接口，那么您可以通过调用语句对象的方法来执行 SQL 语句。例如：

```
stmt.executeUpdate(
    "DELETE FROM Employees
     WHERE EmployeeID = 105" );
```

  请参见“SQL Anywhere JDBC 驱动程序”第 481 页。

- **嵌入式 SQL** 如果您使用的是嵌入式 SQL，那么应在 C 语言 SQL 语句前用关键字 EXEC SQL 作为前缀。然后，代码在编译之前通过预处理器处理。例如：

```
EXEC SQL EXECUTE IMMEDIATE
 'DELETE FROM Employees
  WHERE EmployeeID = 105';
```

  请参见“SQL Anywhere 嵌入式 SQL”第 509 页。

- **Sybase Open Client** 如果使用的是 Sybase Open Client 接口，那么您的 SQL 语句以函数调用形式出现。例如，下面的两个调用将执行 DELETE 语句：

```
ret = ct_command( cmd, CS_LANG_CMD,
                  "DELETE FROM Employees
                   WHERE EmployeeID=105"
                 CS_NULLTERM,
                 CS_UNUSED);
ret = ct_send(cmd);
```

  请参见“Sybase Open Client API”第 807 页。

- **应用程序开发工具** 应用程序开发工具（例如，Sybase Enterprise Application Studio 软件包中的软件）提供了它们自己的 SQL 对象，这些对象在内部使用 ODBC (PowerBuilder) 或 JDBC (Power J)。

有关如何在应用程序中使用 SQL 的详细信息，请参见开发工具文档。如果使用的是 ODBC 或 JDBC，则请查阅软件开发工具包中有关这些接口的信息。

数据库服务器内的应用程序

在许多方面，存储过程和触发器都充当在数据库服务器内运行的应用程序或应用程序的组成部分。您也可以在存储过程中使用此处的许多技术。

有关存储过程和触发器的详细信息，请参见“使用过程、触发器和批处理”《SQL Anywhere 服务器 - SQL 的用法》。

数据库中的 Java 类可以采用与服务器之外的 Java 应用程序同样的方式来使用 JDBC 接口。本章讨论了 JDBC 的一些方面。有关使用 JDBC 的详细信息，请参见“SQL Anywhere JDBC 驱动程序” 第 481 页。

准备语句

每次向数据库发送语句时，数据库服务器必须执行以下步骤：

- 服务器必须分析语句并将其转换为内部形式。这有时称为**准备**语句。
- 服务器必须验证对数据库对象的所有引用的正确性，例如通过检查查询中提到的列是否确实存在。
- 如果语句涉及连接或子查询，则查询优化程序会生成访问计划。
- 在所有的这些步骤都已经执行之后，它会执行该语句。

重复使用预准备语句可以改善性能

如果要反复使用同一语句，例如，在表中插入多行，那么重复准备语句会产生很大不必要的开销。为消除这种开销，一些数据库编程接口提供了使用预准备语句的方法。**预准备语句**是包含一系列占位符的语句。当要执行语句时，您所要做的只是给占位符赋值，而不是再次重新准备整个语句。

使用预准备语句在执行多个类似的操作（如插入多行）时特别有用。

一般来讲，使用预准备语句需要执行下面的步骤：

1. **准备语句**

   在这一步，一般都要为语句提供一些占位符而非实际的值。

2. **反复执行预准备语句**

   在此步骤中，提供每次执行语句时要使用的值。不必每次都准备语句。

3. **删除语句**

   在此步骤中，释放与预准备语句关联的资源。一些编程接口会自动处理此步骤。

不要准备那些只使用一次的语句

通常，不应该准备那些将只执行一次的语句。单独的准备和执行会产生轻微的性能损失，并且它会给应用程序带来不必要的复杂性。

然而，在一些接口中，确实需要准备一个语句以将它与游标关联。

有关游标的信息，请参见“游标简介”一节第 29 页。

用来准备和执行语句的调用并不是 SQL 的组成部分，并且它们也因接口而异。SQL Anywhere 中的每一个编程接口都提供了使用预准备语句的方法。

如何使用预准备语句

本节简短概述如何使用预准备语句。总的过程都是相同的，但细节方面将因接口而异。通过将不同接口中使用预准备语句的方式加以比较，即可看出这一点。

♦ 使用预准备语句（通用）

1. 准备语句。
2. 绑定将在语句中保存值的参数。
3. 给语句中的绑定参数赋值。
4. 执行语句。
5. 根据需要重复执行步骤 3 和 4。
6. 完成之后删除语句。在 JDBC 中，Java 垃圾回收机制会删除语句。

♦ 使用预准备语句 (ADO.NET)

1. 创建一个保存语句的 SACommand 对象。

```
SACommand cmd = new SACommand(
     "SELECT * FROM Employees WHERE Surname=?", conn );
```

2. 为语句中的参数声明数据类型。

   使用 SACommand.CreateParameter 方法。

3. 使用 Prepare 方法准备此语句。

```
cmd.Prepare();
```

4. 执行语句。

```
SADataReader reader = cmd.ExecuteReader();
```

   有关使用 ADO.NET 准备语句的示例，请参见 *samples-dir\SQLAnywhere\ADO.NET\SimpleWin32* 中的源代码。

♦ 使用预准备语句 (ODBC)

1. 使用 SQLPrepare 准备语句。
2. 使用 SQLBindParameter 绑定语句参数。
3. 使用 SQLExecute 执行语句。
4. 使用 SQLFreeStmt 删除语句。

   有关使用 ODBC 准备语句的示例，请参见 *samples-dir\SQLAnywhere\ODBCPrepare* 中的源代码。

   有关 ODBC 预准备语句的详细信息，请参见 ODBC SDK 文档及“执行预准备语句”一节 第 463 页。

♦ 使用预准备语句 (JDBC)

1. 使用连接对象的 prepareStatement 方法准备语句。此步骤返回预准备语句对象。

2. 使用预准备语句对象的适当 **set***Type* 方法设置语句参数。这里的 *Type* 是指派的数据类型。
3. 使用预准备语句对象的适当方法执行语句。对于插入、更新和删除，使用 executeUpdate 方法。

   有关使用 JDBC 准备语句的示例，请参见源代码文件 *samples-dir\SQLAnywhere\JDBC \JDBCExample.java*。

   有关在 JDBC 中使用预准备语句的详细信息，请参见“使用预准备语句进行更有效的访问”一节第 499 页。

♦ **使用预准备语句（嵌入式 SQL）**

1. 使用 EXEC SQL PREPARE 语句准备语句。
2. 给语句中的参数赋值。
3. 使用 EXE SQL EXECUTE 语句执行语句。
4. 使用 EXEC SQL DROP 语句释放与该语句关联的资源。

   有关嵌入式 SQL 预准备语句的详细信息，请参见“PREPARE 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

♦ **使用预准备语句 (Open Client)：**

1. 使用带 CS_PREPARE 类型参数的 ct_dynamic 函数准备语句。
2. 使用 ct_param 设置语句参数。
3. 使用带 CS_EXECUTE 类型参数的 ct_dynamic 执行语句。
4. 使用带 CS_DEALLOC 类型参数的 ct_dynamic 释放与该语句关联的资源。

   有关在 Open Client 中使用预准备语句的详细信息，请参见“在 Open Client 应用程序中使用 SQL”一节第 812 页。

游标简介

当您在应用程序中执行查询时，结果集由若干行组成。通常，您执行查询之前并不知道应用程序要接收多少行。游标提供了在应用程序中处理查询结果集的方式。

使用游标的方式以及可供使用的游标种类取决于所使用的编程接口。有关每个接口可用的游标类型的列表，请参见“游标的可用性”一节第 37 页。

利用游标可在任何编程接口内执行下面的任务：

- 在查询的结果内循环。
- 在结果集内的任何点的基础数据上执行插入、更新和删除。

此外，一些编程接口可使您使用特殊功能调整结果集返回到应用程序的方式，为您的应用程序提供了实实在在的性能优点。

有关可通过不同编程接口使用的游标类型的详细信息，请参见“游标的可用性”一节第 37 页。

什么是游标？

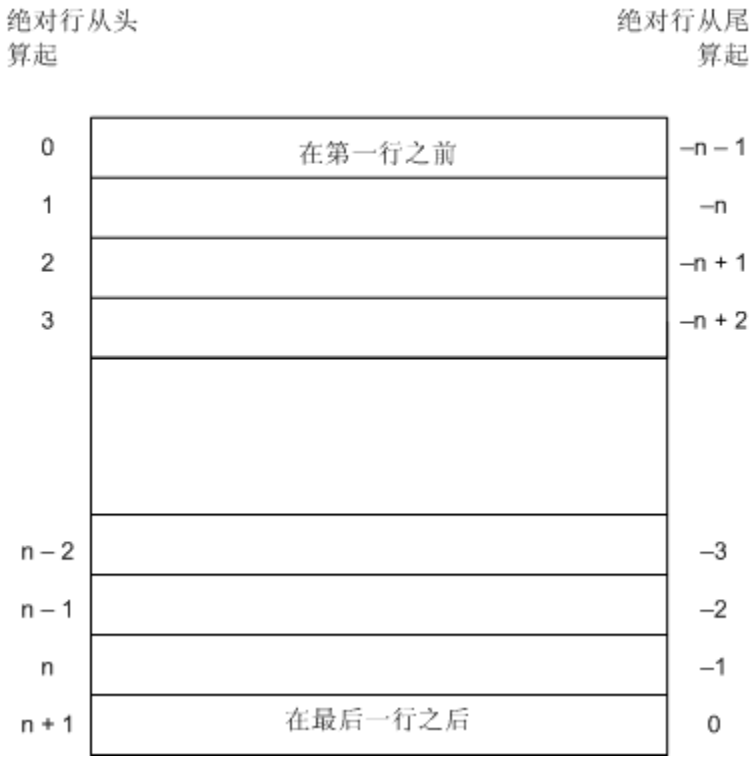
游标是与结果集相关联的名称。结果集可从 SELECT 语句或存储过程调用获得。

游标是结果集上的句柄。任何时候，游标在结果集内都有定义明确的位置。利用游标您可以检查并可能一次一行地操纵数据。SQL Anywhere 游标支持在查询结果中向前和向后移动。

游标位置

游标可以位于以下位置：

- 在结果集的第一行之前。
- 在结果集的某一行上。
- 在结果集的最后一行之后。



游标位置和结果集在数据库服务器中进行维护。客户端会一次一行或一次多行地**读取**行，以便显示和处理。不需要将整个结果集传输到客户端。

使用游标的优点

数据库应用程序中不一定需要使用游标，但使用游标确实具有多个好处。若不使用游标，就必须将整个结果集传输到客户端进行处理和显示，从而会带来以下问题：

- **客户端内存** 如果结果集较大，要在客户端上保存整个结果集，就会需要更多的内存。
- **响应时间** 游标可以在整个结果集汇编起来之前提供开头几行。如果不使用游标，则在整个结果集全部传递过来之前，您的应用程序不会显示任何行。
- **并发控制** 如果对数据进行多次更新而不在应用程序中使用游标，则必须向数据库服务器分别发送多条 SQL 语句来应用这些更改。如果结果集在客户端查询之后又发生了更改，就有可能带来并发问题。结果，就可能会导致更新丢失。

  游标作为其下数据的指针，因此会对您进行的任何更改都强制执行适当的并发约束。

使用游标

本节介绍如何使用游标执行不同类型的操作。

使用游标

在嵌入式 SQL 中使用游标与在其它接口中使用游标不同。

♦ 使用游标（ADO.NET、ODBC、JDBC 和 Open Client）

1. 准备和执行语句。

   使用接口的常用方法执行语句。您可以先准备语句，然后再执行该语句；也可以直接执行语句。

   使用 ADO.NET 时，只有 SACommand.ExecuteReader 方法才返回游标。此命令提供只读、只进游标。

2. 进行测试，看一看语句是否返回结果集。

   在执行创建结果集的语句时，游标被隐式打开。在打开游标时，游标定位在结果集的第一行之前。

3. 读取结果。

   虽然简单读取操作会将游标移到结果集中的下一行，但是 SQL Anywhere 允许在结果集内进行更复杂的移动。

4. 关闭游标。

   当您用完游标之后，将它关闭以释放关联的资源。

5. 释放语句。

   如果您使用了预准备语句，则请释放它以回收内存。

♦ 使用游标（嵌入式 SQL）

1. 准备语句。

   游标通常使用语句句柄而不是字符串。要使用句柄，您需要准备语句。

   有关准备语句的信息，请参见“准备语句”一节第 26 页。

2. 声明游标。

   每个游标都将引用单个 SELECT 或 CALL 语句。当声明游标时，应声明游标的名称和它所引用的语句。

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

3. 打开游标。请参见“OPEN 语句 [ESQL] [SP]”一节 《SQL Anywhere 服务器 - SQL 参考》。

   对于 CALL 语句，打开游标会执行过程直到即将获得第一行时为止。

4. 读取结果。

   虽然简单读取操作会将游标移到结果集中的下一行，但是 SQL Anywhere 允许在结果集内进行更复杂的移动。声明游标的方式决定了可以使用哪些读取操作。请参见“FETCH 语句 [ESQL] [SP]”一节 《SQL Anywhere 服务器 - SQL 参考》和“读取数据”一节第 548 页。

5. 关闭游标。

   当您用完游标之后，应将它关闭。这会释放与游标关联的所有资源。请参见“CLOSE 语句 [ESQL] [SP]”一节 《SQL Anywhere 服务器 - SQL 参考》。

6. 删除语句。

   要释放与语句关联的内存，必须删除语句。请参见“DROP STATEMENT 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关在嵌入式 SQL 中使用游标的详细信息，请参见“读取数据”一节第 548 页。

预取行

在某些情况下，接口库可能会在内部执行性能优化（如预取结果），因此客户端应用程序中的这些步骤可能不完全与软件操作一致。

游标定位

当游标打开之后，它位于第一行之前。您可以将游标移到以查询结果的开头或结尾作为参照的一个绝对位置，或者移到以当前游标位置作为参照的一个相对位置。如何更改游标位置以及可以执行什么操作的具体情况受编程接口约束。

在游标中可读取的行位置编号受整数的大小制约。您最多可以读取到第 2147483646 行，这个数字比可以在整数中保存的值小 1。在使用负数（从末尾开始计算行数）时，您最多可以读取到的行数比整数中可保存的最大负值大 1。

您可以使用特殊定位的更新操作和删除操作来更新或删除位于游标当前位置的行。如果游标定位在第一行之前或最后一行之后，则会返回错误，指示没有相应的游标行。

游标定位问题

对敏感性未定型游标的插入和某些更新会导致游标定位发生问题。SQL Anywhere 不将插入的行放在游标内可预测的位置，除非在 SELECT 语句上有 ORDER BY 子句。在有些情况下，插入的行要等到关闭并再次打开游标后才会出现。对于 SQL Anywhere，如果必须创建工作表才能打开游标，则会出现这种情况。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》。

UPDATE 语句可能导致行在游标中移动。如果游标有使用现有索引的 ORDER BY 子句（不必创建工作表），会发生这种情况。使用 STATIC SCROLL 游标会缓解这些问题，但需要更多的内存和处理。

在打开时配置游标

您可以在打开游标时配置游标行为的以下几个方面：

- **隔离级别** 您可以将游标上操作的隔离级别显式设置为不同于事务的当前隔离级别。为此，请设置 isolation_level 选项。请参见“isolation_level 选项 [数据库] [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》。
- **保存** 缺省情况下，嵌入式 SQL 中的游标在事务结束时关闭。游标 WITH HOLD 打开之后，您可以让它保持打开状态，直到连接结束，或者直到您将它显式关闭。缺省情况下，ADO.NET、ODBC、JDBC 和 Open Client 在事务结束时会让游标保持打开状态。

通过游标读取行

使用游标处理查询的结果集的最简单方式是在结果集的所有行中循环，直到没有行为止。

♦ **在结果集的行中循环**

1. 声明并打开游标（嵌入式 SQL），或者执行返回结果集的语句（ODBC、JDBC、Open Client），或者执行 SADataReader 对象 (ADO.NET)。
2. 继续读取下一行，直到收到 [**未找到行**] 错误。
3. 关闭游标。

此操作的第 2 步如何执行取决于您使用的接口。例如，

- **ADO.NET** 使用 SADataReader.NextResult 方法。请参见“NextResult 方法”一节第 325 页。
- **ODBC** SQLFetch、SQLExtendedFetch 或 SQLFetchScroll 让游标前进到下一行并返回数据。

  有关在 ODBC 中使用游标的详细信息，请参见“使用结果集”一节第 472 页。
- **JDBC** ResultSet 对象的 next 方法让游标前进并返回数据。

  有关在 JDBC 中使用 ResultSet 对象的详细信息，请参见“返回结果集”一节第 503 页。
- **嵌入式 SQL** FETCH 语句执行相同的操作。

  有关在嵌入式 SQL 中使用游标的详细信息，请参见“在嵌入式 SQL 中使用游标”一节第 549 页。
- **Open Client** ct_fetch 函数将游标前进到下一行并返回数据。

  有关在 Open Client 应用程序中使用游标的详细信息，请参见“使用游标”一节第 812 页。

读取多行

请不要将多行读取与预取行混淆。多行读取由应用程序执行，而预取则对应用程序透明，并可以提供类似的性能改进。一次读取多行可以提高性能。

多行读取

某些接口提供了一次将多行读取到数组中的下几个字段的方法。通常，您执行单独的读取操作越少，服务器必须响应的单个请求也就越少，性能也就越好。修改后的检索多行的 FETCH 语句有时也称为**宽读取**。使用多行读取的游标有时称为**块状游标**或**胖游标**。

使用多行读取

- 在 ODBC 中，您可以通过设置 SQL_ATTR_ROW_ARRAY_SIZE 或 SQL_ROWSET_SIZE 属性来设置每一次调用 SQLFetchScroll 或 SQLExtendedFetch 返回的行数。
- 在嵌入式 SQL 中，FETCH 语句使用 ARRAY 子句控制一次读取的行数。
- Open Client 和 JDBC 不支持多行读取。但它们可以使用预取。

用可滚动游标读取

ODBC 和嵌入式 SQL 提供了使用可滚动游标和动态可滚动游标的方法。这些方法使您能够在结果集中一次向前或向后移动多行。

JDBC 和 Open Client 接口不支持可滚动游标。

预取不适用于可滚动操作。例如，读取相反方向的某行并不能预取前面的多行。

通过游标修改行

游标的用途不仅仅是读取查询的结果集。您还可以在处理游标时修改数据库中的数据。这些操作通常称为**定位**插入、更新和删除操作，或者如果操作是插入操作，则称为 PUT 操作。

并非所有的查询结果集都允许定位更新和删除。如果您在不可更新的视图上执行查询，则基础表不会发生更改。此外，如果查询涉及连接，则必须指定您希望从哪一个表删除，或者您希望更新哪些列，何时执行操作。

只有在表中的某些非插入列允许 NULL 或有缺省值的情况下，才能通过游标执行插入。

将多行插入对值敏感的游标（由键集决定的游标）时，新插入的行出现在游标结果集的末尾处。即使这些行与查询的 WHERE 子句不匹配，或者 ORDER BY 子句通常将它们放置在结果集的其它位置，这些行也会出现在最后。此行为与编程接口无关。例如，当使用嵌入式 SQL PUT 语句或 ODBC SQLBulkOperations 函数时就会是这样。通过选择游标中最后一行可以找到最后插入行的自动增量列的值。例如，在嵌入式 SQL 中，可以使用 [`FETCH ABSOLUTE -1 cursor-name`] 来获取该值。此行为的结果是，对值敏感的游标的首次多行插入的开销会很大。

ODBC、JDBC、嵌入式 SQL 和 Open Client 允许使用游标进行数据修改，但 ADO.NET 不允许。对于 Open Client，您可以删除和更新行，但您只能在单表查询上插入行。

可以从哪个表中删除行？

如果您试图通过游标执行定位删除，那么请按如下所示的方法确定从哪个表删除行：

1. 如果 DELETE 语句中未包括 FROM 子句，那么游标必须只在单个表上。

2. 如果游标用于连接式查询（包括使用含有连接的视图），则必须使用 FROM 子句。只会删除指定表的当前行，连接中涉及的其它表不会受到影响。
3. 如果包括了 FROM 子句，但未指定表所有者，那么指定表为第一个与其值相匹配的相关名。请参见“FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》。
4. 如果给出了相关名，就用该相关名来确定指定表的名称。
5. 如果没有给出相关名，那么指定表的名称必须是游标中可明确标识的表名。
6. 如果给出的 FROM 子句中指定了表的所有者，那么指定表的名称必须像游标中的表名一样是明确可标识的。
7. 定位 DELETE 语句可以使用于在视图上打开的游标，只要视图是可更新的即可。

了解可更新语句

本节介绍 SELECT 语句中的子句如何影响可更新语句和游标。

只读语句的可更新性

在游标声明中指定 FOR READ ONLY 或在语句中包含 FOR READ ONLY 子句可使语句为只读语句。换句话说，FOR READ ONLY 子句或使用客户端 API 时适当的只读游标声明会覆盖任何其它可更新性说明。

如果 SELECT 语句的最外层块中包含 ORDER BY 子句且该语句未指定 FOR UPDATE，则游标为 READ ONLY。如果 SQL SELECT 语句指定 FOR XML，则游标为 READ ONLY。否则，游标是可更新的。

可更新语句和并发控制

对于可更新语句，SQL Anywhere 提供了优化和保守两种游标并发控制机制，以便确保滚动操作中结果集保持一致。虽然这两种机制各有各的语义和利弊，它们都可作为使用 INSENSITIVE 游标或快照隔离的替代方法。

FOR UPDATE 的说明可影响游标是否可更新。然而，在 SQL Anywhere 中，FOR UPDATE 语法对并发控制没有任何其它影响。如果使用其它参数指定 FOR UPDATE，SQL Anywhere 将按如下方式变更语句的处理过程，以合并两个并发控制选项之一：

- **保守** 对于在游标的结果集中读取的所有行，数据库服务器将获得意图行锁，以防止任何其它事务更新这些行。
- **优化** 数据库服务器使用的游标类型更改为由键集决定的游标（对行成员资格不敏感，对值敏感），以便在此事务或任何其它事务修改或删除结果中的行时通知应用程序。

通过带有 DECLARE CURSOR 或 FOR 语句的选项，或特定编程接口的并发设置 API，可在游标级别指定保守或优化并发。如果语句是可更新的，而游标未指定并发控制机制，则使用语句的说明。语法如下：

- **FOR UPDATE BY LOCK** 数据库服务器在结果集的读取行上获得意图行锁。这些锁是长期锁，会一直保留到事务 COMMIT 或 ROLLBACK 执行。

- **FOR UPDATE BY { VALUES | TIMESTAMP }** 数据库服务器通过利用键集决定的游标，这样，如果在滚动结果集过程中修改或删除了行，则会通过该游标通知应用程序。

有关详细信息，请参见“DECLARE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“FOR 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

限制可更新语句

FOR UPDATE (*column-list*) 强制实行以下限制：在随后的 UPDATE WHERE CURRENT OF 语句中只能修改指定的结果集属性。

取消游标操作

您可以通过接口函数取消请求。从 Interactive SQL，您可以通过单击工具栏上的 [中断 SQL 语句] 按钮（或者通过从 [SQL] 菜单中选择 [停止]）来取消请求。

如果您取消正在执行游标操作的请求，则游标的位置是不确定的。取消请求之后，您必须按照其绝对位置给游标定位，或者将它关闭。

选择游标类型

本节介绍 SQL Anywhere 游标和 SQL Anywhere 支持的编程接口的可用选项之间的映射。

有关 SQL Anywhere 游标的信息，请参见“SQL Anywhere 游标”一节第 39 页。

游标的可用性

并非所有的接口都为所有的游标类型提供支持。

- ADO.NET 只提供只进、只读游标。
- ADO/OLE DB 和 ODBC 支持所有的游标类型。

  有关详细信息，请参见“使用结果集”一节第 472 页。
- 嵌入式 SQL 支持所有的游标类型。
- 对于 JDBC
  - iAnywhere JDBC 驱动程序支持 JDBC 2.0 和 JDBC 3.0 规范并允许不敏感、敏感和只进敏感性未定游标的声明。
  - jConnect 5.5 和 6.0.5 支持不敏感、敏感和只进敏感性未定型游标的声明，方式与 iAnywhere JDBC 驱动程序相同。但 jConnect 的底层实现仅支持敏感性未定型游标语义。

    有关声明 JDBC 游标的详细信息，请参见“请求 SQL Anywhere 游标”一节第 51 页。
- Sybase Open Client 仅支持敏感性未定型游标。此外，在使用可更新的非唯一游标时，会产生严重的性能下降。

游标属性

您可以显式或者隐式从编程接口请求游标类型。不同的接口库提供不同的游标类型选择。例如，JDBC 和 ODBC 指定了不同的游标类型。

每个游标类型都由多个特性来定义：

- **唯一性** 对游标进行唯一性声明将强制查询返回唯一标识每一行所需要的所有列。通常，这意味着返回主键中的所有列。所需要的但未指定的任何列都要添加到结果集中。缺省游标类型是非唯一的。
- **可更新性** 声明为只读的游标不能在定位更新或删除操作中使用。缺省游标类型是可更新的。
- **可滚动性** 您可以这样声明游标：当您在结果集中移动时，它表现出不同的行为。某些游标可以只读取当前行或后面的行。其它一些游标可以在结果集中来回移动。
- **敏感性** 通过游标也许可以看到对数据库的更改，也许看不到。

这些特性可能会对性能以及数据库服务器内存的使用产生明显的副作用。

SQL Anywhere 可提供具有这些特性的各种组合的游标。请求指定类型的游标时，SQL Anywhere 会尝试匹配这些特性。

某些情况下，并非所有特性都可以提供。例如，SQL Anywhere 中的不敏感游标必须是只读的。如果您的应用程序请求可更新的不敏感游标，则提供给它的将是其它类型的游标（对值敏感）。

书签和游标

ODBC 提供**书签**或值，用以标识游标中的行。对于对值敏感的游标和不敏感游标，SQL Anywhere 支持书签。例如，这意味着：ODBC 游标类型 SQL_CURSOR_STATIC 和 SQL_CURSOR_KEYSET_DRIVEN 支持书签，而游标类型 SQL_CURSOR_DYNAMIC 和 SQL_CURSOR_FORWARD_ONLY 不支持书签。

块状游标

ODBC 提供了一种称为块状游标的游标类型。当您使用 BLOCK 游标时，可以使用 SQLFetchScroll 或 SQLExtendedFetch 读取一整块行，而不是单行。块状游标的行为与嵌入式 SQL ARRAY 读取完全相同。

SQL Anywhere 游标

任何游标一经打开，就会有一个关联的结果集。游标在一定的时间长度内保持打开状态。在这段时间内，与该游标关联的结果集可能被更改，要么是通过游标本身更改，要么是根据隔离级别要求被其它事务更改。有些游标允许看到对基础数据的更改，而其它游标却不会反映出这些更改。对基础数据的更改的敏感性会导致不同的游标行为或称为**游标敏感性**。

SQL Anywhere 为游标提供了各种敏感性特性。本节介绍什么是敏感性，并介绍游标的敏感性特性。

本节假定您阅读过“什么是游标？”一节第 29 页。

成员资格、顺序和值更改

对基础数据的更改会在下列几个方面影响游标的结果集：

- **成员资格** 结果集中的行集，如它们的主键值所标识的。
- **顺序** 结果集中行的顺序。
- **值** 结果集中行的值。

例如，请看以下带有雇员信息的简单表（EmployeeID 是主键列）：

EmployeeID	Surname
1	Whitney
2	Cobb
3	Chin

下列查询上的游标将按主键顺序返回表中的所有结果：

```
SELECT EmployeeID, Surname
FROM Employees
ORDER BY EmployeeID;
```

通过添加新行或删除行可更改结果集的成员资格。值可以通过更改表中的名称来进行更改。顺序可以通过更改雇员的主键值来进行更改。

可见的和不可见的更改

根据隔离级别要求，游标的结果集的成员资格、顺序和值可以在游标打开之后进行更改。根据所使用的游标类型不同，通过应用程序所看到的结果集可能会更改以反映出这些更改，也可能不会更改。

对基础数据的更改可能会通过游标**看到**，也可能**看不到**。可见的更改是反映在游标的结果集中的更改。如果对基础数据的更改不反映在通过游标所看到的结果集中，那么这种更改就是不可见的。

游标敏感性概述

SQL Anywhere 游标按其对基础数据的更改的敏感性来进行分类。特别是，游标敏感性按哪些更改可见来定义。

- **不敏感游标** 当游标打开之后，结果集是固定的。对基础数据的更改都不可见。请参见“不敏感游标”一节第 43 页。
- **敏感游标** 结果集可以在游标打开之后进行更改。对基础数据的所有更改都是可见的。请参见“敏感游标”一节第 44 页。
- **敏感性未定型游标** 更改可能会反映在通过游标看到的结果集的成员资格、顺序或值中，或者也可能根本没有反映。请参见“敏感性未定型游标”一节第 45 页。
- **对值敏感的游标** 对基础数据的顺序或值的更改是可见的。当游标打开之后结果集的成员资格是固定的。请参见“对值敏感的游标”一节第 46 页。

对游标的不同要求给执行设置了不同的约束，因此，对性能也会有不同的约束。请参见“游标敏感性和性能”一节第 47 页。

游标敏感性示例：删除的行

此示例使用一个简单查询来阐释这样的情形：删除结果集中的某一行后，不同的游标会如何响应。

请看以下事件序列：

1. 某个应用程序在以下针对示例数据库的查询中打开一个游标。

```
SELECT EmployeeID, Surname
FROM Employees
ORDER BY EmployeeID;
```

EmployeeID	Surname
102	Whitney
105	Cobb
160	Breault
...	...

2. 应用程序通过游标读取第一行 (102)。
3. 应用程序通过游标读取下一行 (105)。
4. 另外一个事务删除了 102 号雇员(Whitney) 并提交了更改。

在此情况下游标操作的结果取决于游标敏感性：

- **不敏感游标** 在通过游标所看到的结果的成员资格中或值中，不反映这一 DELETE：

操作	结果
读取前一行	返回该行的原始副本 (102)。
读取第一行（绝对读取）	返回该行的原始副本 (102)。
读取第二行（绝对读取）	返回未更改的行 (105)。

- **敏感游标** 结果集的成员资格已经更改，因此，行 105 现在是结果集中的第一行：

操作	结果
读取前一行	返回 [**未找到行**]。没有前一行。
读取第一行（绝对读取）	返回行 105。
读取第二行（绝对读取）	返回行 160。

- **对值敏感的游标** 结果集的成员资格是固定的，因此行 105 仍是结果集的第二行。DELETE 反映在游标的值中，并在结果集中创建了一个有效洞。

操作	结果
读取前一行	返回 [**没有当前的游标行**]。在游标中过去曾经是第一行的地方有一个洞。
读取第一行（绝对读取）	返回 [**没有当前的游标行**]。在游标中过去曾经是第一行的地方有一个洞。
读取第二行（绝对读取）	返回行 105。

- **敏感性未定型游标** 对于更改，结果集的成员资格和值是不确定的。读取前一行、第一行或第二行的响应取决于查询的特定优化方法：该方法是否涉及工作表的构造，另外，所读取的行是否已从客户端预读。

  敏感性未定型游标的优点是，对于许多应用程序，敏感性是不重要的。特别是，如果您要使用只进、只读游标，将不会看到基础更改。此外，如果您在高隔离级别运行，那么将不允许基础更改。

游标敏感性示例：更新的行

此示例使用一个简单查询阐释了这样的情形：更新了结果集中的某一行以致结果集的顺序改变时，不同的游标类型是如何响应的。

请看以下事件序列：

1. 某个应用程序在以下针对示例数据库的查询中打开一个游标。

```
SELECT EmployeeID, Surname
FROM Employees;
```

EmployeeID	Surname
102	Whitney
105	Cobb
160	Breault
...	...

2. 应用程序通过游标读取第一行 (102)。
3. 应用程序通过游标读取下一行 (105)。
4. 另外一个事务将 102 号雇员 (Whitney) 的员工 ID 更新为 165 并提交更改。

在此情况下，游标操作的结果取决于游标敏感性：

- **不敏感游标** 在通过游标看到的结果的成员资格或值中，没有反映出这一 UPDATE：

操作	结果
读取前一行	返回该行的原始副本 (102)。
读取第一行（绝对读取）	返回该行的原始副本 (102)。
读取第二行（绝对读取）	返回未更改的行 (105)。

- **敏感游标** 结果集的成员资格已经更改，因此，行 105 现在是结果集中的第一行：

操作	结果
读取前一行	返回 [**未找到行**]。结果集的成员资格已经更改，因此，第 105 行现在是结果集中的第一行。游标已经移到第一行之前的位置。
读取第一行（绝对读取）	返回行 105。
读取第二行（绝对读取）	返回行 160。

此外，如果该行自上次读取后已更改，则在敏感游标上读取时会返回 SQLE_ROW_UPDATED_WARNING 警告。警告只发出一次。后面再读取同一行时，不会产生警告。

同样，自上次读取某行后，如果通过游标在该行上进行定位更新或删除，就会返回 SQLE_ROW_UPDATED_SINCE_READ 错误。应用程序必须再次读取该行才能使敏感游标上的更新或删除生效。

对任何列的更新都会导致警告/错误，即使该列未被游标引用也是如此。例如，返回 Surname 的查询上的游标将报告更新，即使只有 Salary 列进行了修改也是这样。

- **对值敏感的游标** 结果集的成员资格是固定的，因此行 105 仍是结果集的第二行。UPDATE 反映在游标的值中，并在结果集中创建了一个有效 "洞"。

操作	结果
读取前一行	返回 [未找到行]。结果集的成员资格已经更改，因此，行 105 现在是结果集中的第一行。游标定位在该洞上：它在行 105 前面。
读取第一行（绝对读取）	返回 [没有当前的游标行]。结果集的成员资格已经更改，因此，行 105 现在是结果集中的第一行。游标定位在该洞上：它在行 105 前面。
读取第二行（绝对读取）	返回行 105。

- **敏感性未定型游标** 对于更改，结果集的成员资格和值是不确定的。读取前一行、第一行或第二行的响应取决于查询的特定优化方法：该方法是否涉及工作表的构造，另外，所读取的行是否已从客户端预读。

在批量操作模式下无警告或错误
更新警告和错误情况在批量操作模式（-b 数据库服务器选项）下不会发生。

不敏感游标

这些游标具有不敏感的成员资格、顺序和值。游标打开之后进行的任何更改都是不可见的。

不敏感游标只用于只读游标类型。

标准

不敏感游标对应于不敏感游标的 ISO/ANSI 标准定义，并对应于 ODBC 静态游标。

编程接口

接口	游标类型	注释
ODBC、ADO/OLE DB	静态	如果请求可更新的静态游标，则改为使用对值敏感的游标。
嵌入式 SQL	INSENSITIVE	
JDBC	INSENSITIVE	不敏感语义仅受 iAnywhere JDBC 驱动程序支持。
Open Client	不支持	

说明

不敏感游标会始终按任何可能存在的 ORDER BY 子句指定的顺序返回与查询的选择标准匹配的行。

当游标打开之后，不敏感游标的结果集将完全作为工作表实例化。这样会带来以下后果：

- 如果结果集非常大，那么管理结果集需要的磁盘空间和内存可能也非常大。
- 在整个结果集被汇编为工作表之前，没有行返回到应用程序。对于复杂的查询，这可能会导致经过一段延迟后第一行才返回到应用程序。
- 后面的行可以直接从工作表读取，因此可以快速地返回。客户端库可以一次预取多行，从而进一步改善性能。
- 不敏感游标不会受 ROLLBACK 或 ROLLBACK TO SAVEPOINT 的影响。

敏感游标

敏感游标可以用于只读或可更新的游标类型。

这些游标具有敏感的成员资格、顺序和值。

标准

敏感游标对应于敏感游标的 ISO/ANSI 标准定义，并对应于 ODBC 动态游标。

编程接口

接口	游标类型	注释
ODBC、ADO/OLE DB	动态	
嵌入式 SQL	SENSITIVE	当不需要工作表并且 prefetch 选项设置为 Off 时，也响应 DYNAMIC SCROLL 游标请求而予以提供。
JDBC	SENSITIVE	iAnywhere JDBC 驱动程序完全支持敏感游标。

说明

敏感游标禁用预取。通过游标可看到所有更改，包括通过游标以及从其它事务做出的更改。如果隔离级别较高，则可能会因锁定而隐藏一些在其它事务中做出的更改。

对游标成员资格、顺序以及所有列值的更改都是可见的。例如，如果敏感游标包含连接，而且，修改了某个基础表的某个值，那么，由该基行组成的所有结果行都会显示新值。结果集成员资格和顺序可能会在每一次读取时更改。

敏感游标会始终返回与查询的选择标准匹配的行，返回顺序为任何 ORDER BY 子句指定的顺序。更新可能会影响结果集的成员资格、顺序和值。

敏感游标的要求会对敏感游标的实现施加限制：

- 行不能被预取，因为对预取的行进行的更改通过游标将不可见。这可能会影响性能。
- 敏感游标必须在没有构建任何工作表的情况下实现，这是因为如果行存储为工作表，则无法通过游标看到对这些行的更改。
- 由于存在着不能有工作表这样的限制，所以优化器选择连接方法时也会受到限制，因而性能也可能会受到影响。
- 对于某些查询，优化器不能构建不包括工作表并会使游标敏感的计划。

  工作表通常用于对中间结果进行排序和分组。如果行可以通过索引进行访问，那么排序就不需要工作表。虽然无法准确说明都有哪些查询会使用工作表，但以下查询肯定要使用工作表：

  - UNION 查询，虽然 UNION ALL 查询不一定使用工作表。
  - 带有 ORDER BY 子句的语句，如果在 ORDER BY 列上没有索引。
  - 任何使用散列连接优化的查询。
  - 许多涉及 DISTINCT 或 GROUP BY 子句的查询。

  在这些情况下，SQL Anywhere 或者会向应用程序返回错误，或者会将游标类型更改为敏感性未定型游标并返回警告。

  有关查询优化和工作表使用的详细信息，请参见“查询优化与执行” 《SQL Anywhere 服务器 - SQL 的用法》。

敏感性未定型游标

这些游标在其成员资格、顺序或值方面没有明确定义的敏感性。在敏感性方面允许的这一灵活性可使敏感性未定型游标得以优化，进而改善性能。

敏感性未定型游标只用于只读游标类型。

标准

敏感性未定型游标对应于敏感性未定型游标的 ISO/ANSI 标准定义，并对应于带有非特定敏感性的 ODBC 游标。

编程接口

接口	游标类型
ODBC、ADO/OLE DB	未指定的敏感性
嵌入式 SQL	DYNAMIC SCROLL

说明

SQL Anywhere 可使用一些方法来优化查询并向应用程序返回行，请求敏感性未定型游标并不会给这些方法带来什么限制。因此，敏感性未定型游标提供的性能最佳。特别是，对于将中间结果实例化为工作表的任何措施，优化器均可自由使用，而且，客户端可以预取行。

SQL Anywhere 不保证用户能够看到对数据库基础行的更改。某些更改可能是可见的，而另外一些是不可见的。成员资格和顺序可能会在每一次读取时都更改。特别是，对基行的更新可能会导致只有部分更新的列反映在游标的结果中。

敏感性未定型游标不能保证返回与查询的选择和顺序匹配的行。行成员资格在游标打开时是固定的，但后续对基础值的更改将反映在结果中。

敏感性未定型游标会始终返回这样的行：这些行在建立游标成员资格时匹配客户的 WHERE 和 ORDER BY 子句。如果在游标打开之后列值被更改，那么可能返回不再匹配 WHERE 和 ORDER BY 子句的行。

对值敏感的游标

对于对值敏感的游标，会员资格是不敏感的，结果集的顺序和值是敏感的。

对值敏感的游标可以用于只读或可更新的游标类型。

标准

对值敏感的游标不符合 ISO/ANSI 标准定义。它们对应于 ODBC 键集决定的游标。

编程接口

接口	游标类型	注释
ODBC、ADO/ OLE DB	由键集决定	
嵌入式 SQL	SCROLL	
JDBC	INSENSITIVE 和 CONCUR_UPDATABLE	使用 iAnywhere JDBC 驱动程序时，如果请求可更新的 INSENSITIVE 游标，则会提供对值敏感的游标。
Open Client 和 jConnect	不支持	

说明

如果应用程序读取的行是由已经更改的数据库基础行构成的，那么，就必须给应用程序提供更新后的值，而且必须向应用程序发出 SQL_ROW_UPDATED 状态。如果应用程序试图读取的行是由被删除的数据库基础行构成的，那么，就必须向应用程序发出 SQL_ROW_DELETED 状态。

对主键值的更改会从结果集中删除行（作为删除对待，后面跟插入）。当结果集中的某一行被删除（从游标中或者游标之外）并插入带有同一主键值的一个新行时，会发生特殊情况。这将会导致在旧行出现的位置由新行替换旧行。

结果集中的行并不一定会与查询的选择或顺序指定匹配。由于行成员资格在打开时是固定的，因此，即使后续的更改使行不匹配 WHERE 子句或 ORDER BY，这样的更改也并不会更改行的成员资格，同样也不会更改行的位置。

所有的值对通过游标进行的更改都是敏感的。成员资格对通过游标进行的更改的敏感性受 ODBC 选项 SQL_STATIC_SENSITIVITY 的控制。如果该选项已打开，那么通过游标的插入会将行添加到游标。否则，它们就不是结果集的组成部分。通过游标的删除将会从结果集删除行，从而可防止洞返回 SQL_ROW_DELETED 状态。

对值敏感的游标可使用**键集表**。当游标打开之后，SQL Anywhere 将用组成结果集的每一行的标识信息填充工作表。当在结果集中滚动时，将使用键集表标识结果集的成员资格，但在必要时从基础表获取值。

对值敏感的游标的固定成员资格属性可使您的应用程序记住游标内的行位置，并确保这些位置不会改变。请参见“游标敏感性示例：删除的行”一节第 40 页。

- 如果自从打开游标之后某一行被更新或者可能已经更新，那么 SQL Anywhere 将在读取该行时返回 SQLE_ROW_UPDATED_WARNING。警告只生成一次：再次读取同一行时不再生成警告。

  只要更新行中的列，即使游标不引用所更新的列，也会产生警告。例如，即使只有 Birthdate 列被修改，Surname 和 GivenName 上的游标也将报告更新。当行锁定被禁用时，这些更新警告和错误情况在批量操作模式（-b 数据库服务器选项）下不会发生。请参见“批量操作的性能问题”一节 《SQL Anywhere 服务器 - SQL 的用法》和“上次读取后行已更新”一节 《错误消息》。

- 在自上次读取后被修改的行上执行定位更新或删除的尝试将会返回 SQLE_ROW_UPDATED_SINCE_READ 错误并会取消该语句。应用程序必须再次 FETCH 该行，然后才能允许 UPDATE 或 DELETE。

  只要更新行中的列，即使游标不引用所更新的列，也会产生错误。在批量操作模式下，不会发生错误。请参见“上次读取后行已更改 -- 操作被取消”一节 《错误消息》。

- 打开游标之后，如果通过游标或从另一事务中删除了某行，游标中就会出现一个**洞**。由于游标的成员资格是固定的，因此，会保留行位置，但 DELETE 操作会反映在行的更改的值中。如果您读取此洞上的行，您会收到 [**没有当前的游标行**] 错误，指出没有当前行，并且游标仍定位在该洞上。您可以通过使用敏感游标来避免洞，因为敏感游标的成员资格会随值一起改变。请参见“没有当前的游标行”一节 《错误消息》。

对于对值敏感的游标，行不能被预取。此要求可能会在某些情况下影响性能。

插入多行

通过对值敏感的游标插入多行时，新插入的行出现在该结果集的末尾处。请参见“通过游标修改行”一节第 34 页。

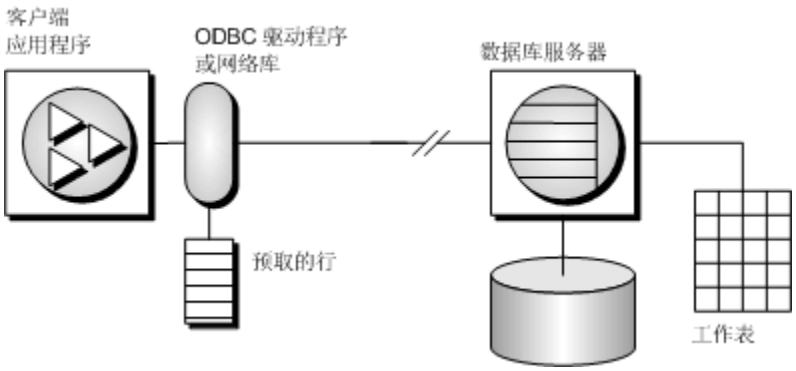
游标敏感性和性能

性能与游标的其它属性往往无法兼顾。特别是，如果使游标成为可更新游标，就会限制游标查询处理和传递，进而影响性能。此外，对游标敏感性提出要求也会约束游标性能。

要理解游标的可更新性和敏感性如何影响性能，就需要理解能通过游标看到的结果是如何从数据库传输到客户端应用程序的。

特别是，结果可能会由于性能原因存储在两个中间位置：

- **工作表** 无论是中间还是最终结果都可以作为工作表存储。对值敏感的游标会使用由主键值构成的工作表。查询特性也可能导致优化程序在其选择的执行计划中使用工作表。
- **预取** 通信的客户端可能会将许多行检索到客户端的缓冲区中，以避免为每一行单独将请求发送到数据库服务器。



敏感性和可更新性限制了中间位置的使用。

预取行

预取和多行读取是不同的。预取可以在没有来自客户端应用程序的显式指令的情况下执行。预取会将行从服务器检索到客户端上的缓冲区中，但客户端应用程序要先读取相应的行，然后才能使用这些行。

缺省情况下，只要应用程序读取一行，SQL Anywhere 客户端库就会预取多行。SQL Anywhere 客户端库会将其余的行存储在缓冲区中。

预取会通过减少客户端/服务器端通信的往返次数而提高性能，并且不用为每一行或行块向服务器发送单独的请求，就可以让许多行可供使用，因而可提高吞吐量。

有关控制预取的详细信息，请参见“prefetch 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

控制应用程序的预取

- prefetch 选项可以控制是否发生预取。可将单个连接的 prefetch 选项设为 Always、Conditional 或 Off。缺省情况下，它被设置为 Conditional。
- 在嵌入式 SQL 中，当您使用 BLOCK 子句打开单个 FETCH 操作上的游标时，您可以针对每个游标控制预取。

  应用程序可通过指定 BLOCK 子句来指定从服务器读取一次最多可以包含多少行。例如，如果一次读取并显示 5 行，则可以使用 BLOCK 5；如果指定 BLOCK 0，一次就会读取 1 个记录，并且会使 FETCH RELATIVE 0 总是从服务器重复读取行。

  虽然您还可以通过在应用程序上设置连接参数来将预取关闭，但是指定 BLOCK 0 比将 prefetch 选项设置为 Off 效率更高。请参见“prefetch 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

- 缺省情况下，预取对于值敏感型游标是禁用的。
- 在 Open Client 中，您可以在声明游标之后（但要在打开它之前）使用 ct_cursor 并使用 CS_CURSOR_ROWS 来控制预取行为。

在可能有助于提高性能的情况下，预取会动态增加预取的行数。这包括符合以下条件的游标：

- 使用一种支持的游标类型：
  - **ODBC 和 OLE DB**　FORWARD-ONLY 和 READ-ONLY（缺省）游标
  - **嵌入式 SQL**　DYNAMIC SCROLL（缺省）、NO SCROLL 和 INSENSITIVE 游标
  - **ADO.NET**　所有游标
- 只执行 FETCH NEXT 操作（无绝对读取、相对读取或向后读取）。
- 应用程序不会在两次读取之间更改主机变量类型，也不使用 GET DATA 语句按块获取列数据（支持使用一个 GET DATA 语句获取值）。

更新丢失

使用可更新的游标时，防止更新丢失很重要。更新丢失是这样一种情况：两个或多个事务更新同一行，但这些事务彼此之间都不知道其它事务进行的修改，因此第二个更改会覆盖第一个修改。下面示例说明了此问题：

1. 某个应用程序在以下针对示例数据库的查询中打开一个游标。

```
SELECT ID, Quantity
FROM Products;
```

ID	Quantity
300	28
301	54
302	75
...	...

2. 应用程序通过游标读取 ID = 300 的行。
3. 另外一个事务使用下面的语句更新该行：

```
UPDATE Products
SET Quantity = Quantity - 10
WHERE ID = 300;
```

4. 然后，应用程序通过游标将该行值更新为 `(Quantity - 5 )` 的值。
5. 该行的正确的最终值是 13。如果游标预读了该行，那么该行的新值是 23。另外一个事务的更新会丢失。

在数据库应用程序中，如果预先不对行值进行验证就进行更改，那么在任何一个隔离级别都有可能丢失更新。在较高隔离级别（2 和 3），可以使用锁（读取、意图和写入锁）来确保其它事务不能对应用程序已读取的行进行更改。但在隔离级别 0 和 1，更新丢失的可能性会更大：在隔离级别 0，不能获得读取锁来防止随后的数据更改；在隔离级别 1，只能锁定当前行。使用快照隔离时不会发生更新丢失，因为任何更改旧值的尝试都会导致更新冲突。同样，在隔离级别 1 使用预取也有可能导致丢失更新，因为在客户端的预取缓冲区中应用程序定位的结果集行可能与游标中服务器定位的当前行不同。

在隔离级别 1 使用游标时，为防止丢失更新，数据库服务器支持三种不同的并发控制机制，这三个机制可由应用程序指定：

1. 读取游标中的每一行时在该行上获得意图行锁。意图锁防止其它事务在同一行上获得意图锁或写入锁，从而防止并发更新。但是，意图锁不防碍读取行锁，因此意图锁不影响并发只读语句。
2. 使用对值敏感的游标。对值敏感的游标可用于跟踪基础行发生更改或删除的时间，以便应用程序可以采取相应措施。
3. 使用 FETCH FOR UPDATE，此方法可在特定行上获得意图行锁。

指定这些替代方法的方式取决于应用程序所使用的接口。对于适用于 SELECT 语句的前两个替代方法：

- 在 ODBC 中不会发生更新丢失，因为在声明可更新游标时应用程序必须为 SQLSetStmtAttr 函数指定游标并发参数。此参数是 SQL_CONCUR_LOCK、SQL_CONCUR_VALUES、SQL CONCUR_READ_ONLY 或 SQL_CONCUR_TIMESTAMP 之一。对于 SQL_CONCUR_LOCK，数据库服务器可获取行意图锁。对于 SQL_CONCUR_VALUES 和 SQL_CONCUR_TIMESTAMP，可使用对值敏感的游标。SQL_CONCUR_READ_ONLY 用于只读游标，是缺省值。
- 在 JDBC 中语句的并发设置类似于 ODBC 中的设置。iAnywhere JDBC 驱动程序支持 JDBC 并发值 RESULTSET_CONCUR_READ_ONLY 和 RESULTSET_CONCUR_UPDATABLE。第一个值对应于 ODBC 并发设置 SQL_CONCUR_READ_ONLY，指定只读语句。第二个值对应于 ODBC SQL_CONCUR_LOCK 设置，因此使用行意图锁来防止更新丢失。请注意，在 JDBC 3.0 规范中不能直接指定对值敏感的游标。
- 在 jConnect 中，在 API 级别支持可更新游标，但底层实现（使用 TDS）不支持通过游标进行更新。jConnect 将单独的 UPDATE 语句发送到数据库服务器来更新特定行。为避免更新丢失，应用程序必须在隔离级别为 2 或更高的情况下运行。应用程序还可以从游标发出单独的 UPDATE 语句，但必须确保 UPDATE 语句通过在其 WHERE 子句中设置相应的条件来验证自读取该行以后该行值未变化。
- 在嵌入式 SQL 中，可通过在 SELECT 语句本身或游标声明中包括语法来设置并发说明。在 SELECT 语句中，语法 SELECT ...FOR UPDATE BY LOCK 导致数据库服务器在结果集上获取意图行锁。

  或者，SELECT ...FOR UPDATE BY [VALUES | TIMESTAMP] 促使数据库服务器将游标类型更改为对值敏感的游标，这样如果自上次读取特定行以后通过游标对该行进行了更改，在使用 FETCH 语句时应用程序会收到警告 (SQLE_ROW_UPDATED_WARNING)，在使用 UPDATE WHERE CURRENT OF 语句时应用程序会收到错误 (SQLE_ROW_UPDATED_SINCE_READ)。如果删除行，应用程序也会收到错误 (SQLE_NO_CURRENT_ROW)。

嵌入式 SQL 和 ODBC 接口也支持 FETCH FOR UPDATE 功能，虽然细节方面因所使用的 API 而不同。

在嵌入式 SQL 中，应用程序使用 FETCH FOR UPDATE，而非 FETCH，促使在该行上获取意图锁。在 ODBC 中，应用程序使用 API 调用 SQLSetPos，并使用操作参数 SQL_POSITION 或 SQL_REFRESH 和锁类型参数 SQL_LOCK_EXCLUSIVE，以在行上获取意图锁。在 SQL Anywhere 中，这些锁是长期锁，会一直保持到提交或回退事务。

游标敏感性和隔离级别

游标敏感性和隔离级别都可解决并发控制问题，但处理方式不同，并且利弊也各不相同。

通过选择事务的隔离级别（通常在连接级别），您可以确定何时在数据库中的行上放置锁以及锁的类型。锁可以防止其它事务访问或修改数据库中的行。通常，维护的锁的数目越多，并发事务之间预期的并发级别越低。

但是，锁并不会阻止同一事务的其它部分进行更新。因此，维护多个可更新游标的单个事务无法依赖锁定来防止诸如更新丢失之类的问题。

快照隔离的目的是避免使用读取锁，其方法是确保每个事务都能查看到一致的数据库视图。显而易见的好处是无需依赖于完全可序列化事务（隔离级别 3）即可查询到一致的数据库视图，并且可以避免随使用隔离级别 3 发生的并发丢失。但是，快照隔离会带来巨大的开销，因为必须维护修改行的副本才能同时满足正在执行的并发快照事务的要求，以及尚未启动的快照事务的要求。由于存在这种副本维护，因此存在大量的更新时可能不适合使用快照隔离。请参见“选择快照隔离级别”一节 《SQL Anywhere 服务器 - SQL 的用法》。

另一方面，游标敏感性确定在游标的结果中可以看见（或不能看见）哪些更改。虽然事务的影响完全取决于指定的游标类型，但因为游标敏感性是基于游标指定的，所以游标敏感性既应用于其它事务的影响，也应用于同一事务的更新活动。通过设置游标敏感性，将不直接确定何时在数据库中的行上放置锁。而是由游标敏感性与隔离级别的组合来控制对特定应用程序可能发生的各种并发情况。

请求 SQL Anywhere 游标

当您从客户端应用程序请求游标类型时，SQL Anywhere 会提供一个游标。SQL Anywhere 游标不是按照编程接口中指定的类型进行定义的，而是按照结果集对基础数据中的更改的敏感性来定义的。SQL Anywhere 会根据您请求的游标类型提供一个行为与该类型相匹配的游标。

SQL Anywhere 按照客户端游标类型的请求来设置游标的敏感性。

ADO.NET

通过使用 SACommand.ExecuteReader 可使用只进、只读游标。SADataAdapter 对象不使用游标，而是使用客户端结果集。请参见“SACommand 类”一节第 195 页。

ADO/OLE DB 和 ODBC

下表说明了为响应不同的 ODBC 可滚动游标类型而设置的游标敏感性。

ODBC 可滚动游标类型	SQL Anywhere 游标
STATIC	不敏感
KEYSET-DRIVEN	对值敏感
DYNAMIC	敏感
MIXED	对值敏感

通过将游标类型设置为 SQL_CURSOR_KEYSET_DRIVEN，然后使用 SQL_ATTR_KEYSET_SIZE 为由键集决定的游标指定键集中的行数，可获取 MIXED 游标。如果键集大小为 0（缺省值），则游标完全由键集决定。如果键集大小大于 0，则游标是混合的（在键集内由键集决定，在键集外动态变化）。键集大小的缺省值为 0。键集大小大于 0 而小于行集大小 (SQL_ATTR_ROW_ARRAY_SIZE) 是错误的。

有关 SQL Anywhere 游标及其行为的信息，请参见“SQL Anywhere 游标”一节第 39 页。

有关如何在 ODBC 中请求游标类型的信息，请参见“选择 ODBC 游标特性”一节第 473 页。

例外

如果将 STATIC 游标请求为可更新，则提供的将是对值敏感的游标，而且会发出一个警告。

如果请求了 DYNAMIC 或 MIXED 游标，但若不使用工作表查询就无法执行，那么就会发出警告，并改为提供敏感性未定型游标。

JDBC

JDBC 2.0 和 3.0 规范支持三种游标类型：不敏感、敏感和只进敏感性未定。iAnywhere JDBC 驱动程序符合这些 JDBC 规范，也支持 JDBC ResultSet 对象的三种不同的游标类型。但是，在一些情况下，数据库服务器无法使用给定游标类型所需的语义构造访问计划。在这些情况下，数据库服务器或者返回错误，或者用其它游标类型替代。请参见“敏感游标”一节第 44 页。

使用 jConnect 时，尽管 jConnect 支持的 API 可按照 JDBC 2.0 规范创建不同的游标类型，但在数据库服务器上底层协议 (TDS) 却仅支持只进、只读敏感性未定型游标。因为 TDS 协议将语句的结果集缓存在块中，所以所有 jConnect 游标都是敏感性未定型。应用程序需要滚动支持可滚动性的不敏感或敏感游标类型时，会滚动这些缓存结果的块。如果应用程序向后滚动时越过高速缓存的结果集的开头，就会再次执行语句。在这种情况下，如果在两次语句执行之间更改了数据，就会导致数据不一致。

嵌入式 SQL

要从嵌入式 SQL 应用程序请求游标，您可以在 DECLARE 语句上指定游标类型。下表说明了为响应不同的请求设置的游标敏感性：

游标类型	SQL Anywhere 游标
NO SCROLL	敏感性未定
DYNAMIC SCROLL	敏感性未定
SCROLL	对值敏感
INSENSITIVE	不敏感
SENSITIVE	敏感

例外

如果将 DYNAMIC SCROLL 或 NO SCROLL 游标请求为 UPDATABLE，就会提供敏感的或对值敏感的游标。无法保证提供两个游标中的哪一个。这种不确定性正好符合敏感性未定行为的定义。

如果将 INSENSITIVE 游标请求为 UPDATABLE，就会提供一个对值敏感的游标。

如果请求 DYNAMIC SCROLL 游标，那么在 prefetch 数据库选项被设置为 Off 而且查询执行计划未涉及工作表的情况下，可能会提供敏感的游标。同样地，这种不确定性也符合敏感性未定行为的定义。

Open Client

与 jConnect 一样，Open Client 的底层协议 (TDS) 也仅支持只进、只读、敏感性未定游标。

描述结果集

有些应用程序会构建无法完全在应用程序中指定的 SQL 语句。例如，在有些情况下，在应用程序准确知道要检索什么信息之前（比如当某一报告应用程序让用户选择要显示的列时），语句取决于来自用户的响应。

在这种情况下，应用程序需要用一种方法来检索关于**结果集**的性质及内容方面的信息。关于结果集性质的信息（被称为**描述符**）用于标识数据结构，包括希望返回的列的数量和类型。一旦应用程序确定了结果集的性质，检索内容的过程就简单了。

此**结果集元数据**（关于数据的性质和内容的信息）通过描述符来操纵。获取和管理结果集元数据的过程称为**描述**。

由于游标通常会产生结果集，所以描述符和游标会紧密链接，只不过有些接口隐藏了描述符的使用，用户看不到。通常，需要描述符的语句是返回结果集的 SELECT 语句或存储过程。

与基于游标的操作一起使用描述符的顺序如下所示：

1. 分配描述符。此操作可以隐式进行，不过有些接口也允许显式分配。
2. 准备语句。
3. 描述语句。如果语句是存储过程调用或批处理，且结果集不是由过程定义中的结果子句定义的，那么描述应在打开游标之后发生。
4. 为语句声明和打开游标（嵌入式 SQL）或执行该语句。
5. 如果有必要，获取描述符并修改分配区域。此步骤通常隐式进行。
6. 读取和处理语句结果。
7. 释放描述符。
8. 关闭游标。
9. 删除语句。有些接口可以自动完成此步骤。

实现注意事项

- 在嵌入式 SQL 中，SQLDA（SQL 描述符区域）结构将保存描述符信息。请参见“SQL 描述符区域 (SQLDA)”一节第 540 页。
- 在 ODBC 中，使用 SQLAllocHandle 分配的描述符句柄提供了对描述符的字段的访问。您可以使用 SQLSetDescRec、SQLSetDescField、SQLGetDescRec 和 SQLGetDescField 对这些字段进行操作。

  或者，您还可以使用 SQLDescribeCol 和 SQLColAttributes 获取列信息。
- 在 Open Client 中，您可以使用 ct_dynamic 来准备语句，使用 ct_describe 来描述语句的结果集。不过，您还可以使用 ct_command 来发送 SQL 语句而不必事先准备它，并使用 ct_results 逐个地处理返回的行。这是在 Open Client 应用程序开发中比较常见的操作方式。
- 在 JDBC 中，java.sql.ResultSetMetaData 类提供了有关结果集的信息。

- 您还可以使用向数据库服务器发送数据的描述符（例如，使用 INSERT 语句）；然而，这与用于结果集的描述符是不同种类的描述符。

  有关 DESCRIBE 语句的输入和输出参数的详细信息，请参见“DESCRIBE 语句 [ESQL]”一节《SQL Anywhere 服务器 - SQL 参考》。

在应用程序中控制事务

事务是原子 SQL 语句集。要么执行事务中的所有语句，要么一个也不执行。本节介绍应用程序中的事务的一些方面。

有关事务的详细信息，请参见“使用事务和隔离级别” 《SQL Anywhere 服务器 - SQL 的用法》。

设置自动提交或手工提交模式

数据库编程接口可以在**手工提交**模式或**自动提交**模式下运行。

- **手工提交模式** 只有在您的应用程序执行显式提交操作时，或者，在数据库服务器执行自动提交时（例如，执行 ALTER TABLE 语句或其它数据定义语句时），才提交操作。手工提交模式有时也称为**链接模式**。

  要在应用程序中使用事务（包括嵌套事务和保存点），您必须在手工提交模式下操作。

- **自动提交模式** 每一语句都视为单独的事务。自动提交模式相当于在每条 SQL 语句的结尾附加一条 COMMIT 语句。自动提交模式有时也称为**非链接模式**。

自动提交模式会影响应用程序的性能和行为。如果应用程序需要事务完整性，则不要使用自动提交。

有关自动提交模式如何影响性能的信息，请参见“关闭自动提交模式”一节 《SQL Anywhere 服务器 - SQL 的用法》。

控制自动提交行为

控制应用程序的提交行为的方式取决于您使用的编程接口。自动提交的实现可以是客户端的或服务器端的，具体情况视接口而定。请参见“自动提交实现细节”一节第 57 页。

♦ 控制自动提交模式 (ADO.NET)

- 缺省情况下，ADO.NET 提供程序在自动提交模式下工作。若要使用显式事务，请使用 SAConnection.BeginTransaction 方法。请参见“事务处理”一节第 129 页。

♦ 控制自动提交模式 (OLE DB)

- 缺省情况下，OLE DB 提供程序在自动提交模式下工作。若要使用显式事务，请使用 ITransactionLocal::StartTransaction、ITransaction::Commit 和 ITransaction::Abort 方法。

♦ 控制自动提交模式 (ODBC)

- 缺省情况下，ODBC 在自动提交模式下工作。关闭自动提交的方式取决于您是在直接使用 ODBC 还是在使用应用程序开发工具。如果要直接对 ODBC 接口编程，则请设置 SQL_ATTR_AUTOCOMMIT 连接属性。

♦ **控制自动提交模式 (JDBC)**

- 缺省情况下，JDBC 在自动提交模式下工作。若要关闭自动提交，请使用连接对象的 setAutoCommit 方法：

```
conn.setAutoCommit( false );
```

♦ **控制自动提交模式（嵌入式 SQL）**

- 缺省情况下，嵌入式 SQL 应用程序在手工提交模式下工作。若要启用自动提交，请使用类似如下的语句将 chained 数据库选项（服务器端选项）设置为 Off：

```
SET OPTION chained='Off';
```

♦ **控制自动提交模式 (Open Client)**

- 缺省情况下，通过 Open Client 进行的连接在自动提交模式下工作。您可以更改此行为，方法是使用类似如下的语句在应用程序中将 chained 数据库选项（服务器端选项）设置为 On：

```
SET OPTION chained='On';
```

♦ **控制自动提交模式 (PHP)**

- 缺省情况下，PHP 在自动提交模式下工作。若要关闭自动提交，请使用 sqlanywhere_set_option 函数：

```
$result = sasql_set_option( $conn, "auto_commit", "Off" );
```

请参见“sasql_set_option”一节第 743 页。

♦ **控制自动提交模式（在服务器上）**

- 缺省情况下，数据库服务器在手工提交模式下工作。若要启用自动提交，请使用类似如下的语句将 chained 数据库选项（服务器端选项）设置为 Off：

```
SET OPTION chained='Off';
```

  如果使用的是在客户端上控制提交的接口，则设置 chained 数据库选项（服务器端选项）会影响应用程序的性能和/或行为。不建议设置服务器的链接模式。

请参见“设置自动提交或手工提交模式”一节第 56 页。

自动提交实现细节

根据您使用的接口以及控制自动提交行为的方式，自动提交模式的行为会稍有不同。

实现自动提交模式可以采用以下两种方式之一：

- **客户端自动提交** 当应用程序使用自动提交时，客户端库在每一个 SQL 语句执行之后发送 COMMIT 语句。

ADO.NET、ADO/OLE DB、ODBC 和 PHP 应用程序从客户端控制提交行为。

- **服务器端自动提交** 应用程序关闭链接模式时，数据库服务器提交每个 SQL 语句的结果。此行为在使用 JDBC 的情况下由 chained 数据库选项隐式控制。

  嵌入式 SQL、JDBC 和 Open Client 应用程序操纵服务器端提交行为（例如，这些应用程序设置 chained 选项）。

对于复合语句（如存储过程或触发器），在客户端和服务器端自动提交之间有一些区别。从客户端看，存储过程是单一语句，因此自动提交将在整个过程执行之后发送单一提交语句。从数据库服务器的角度来看，存储过程可以由许多 SQL 语句构成，因此服务器端自动提交将提交该过程内的每个 SQL 语句的结果。

不要混合使用客户端和服务器端实现
在 ADO.NET、ADO/OLE DB、ODBC 或 PHP 应用程序中，不要将设置 chained 选项与设置 autocommit 选项这两个操作混合使用。

控制隔离级别

可以使用 isolation_level 数据库选项设置当前连接的隔离级别。

某些接口（如 ODBC）允许您在连接时设置连接的隔离级别。以后，您可以使用 isolation_level 数据库选项将此级别重置。请参见“isolation_level 选项 [数据库] [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》。

游标和事务

通常，执行 COMMIT 时游标会关闭。但此行为也有两个例外：

- close_on_endtrans 数据库选项被设置为 Off。
- 游标打开为 WITH HOLD，这是 Open Client 和 JDBC 的缺省值。

如果这两种情况中任何一种为真，那么游标在 COMMIT 时依然打开。

ROLLBACK 和游标

如果事务回退，游标就会关闭，但那些使用 WITH HOLD 打开的游标除外。但是回退后游标的内容并不可靠。

ISO SQL3 标准草案声明：在回退时，所有游标（甚至是那些使用 WITH HOLD 打开的游标）都应关闭。您可以通过将 ansi_close_cursors_on_rollback 选项设置为 On 来获得此行为。

保存点

如果事务回退到保存点，并且如果 ansi_close_cursors_on_rollback 选项为 On，那么所有在 SAVEPOINT 后还处于打开状态的游标（甚至是那些打开为 WITH HOLD 的游标）将关闭。

游标和隔离级别

可以使用 SET OPTION 语句来更改 isolation_level 选项，以在事务期间更改连接的隔离级别。但是，此更改不影响打开的游标。

当 WITH HOLD 子句与快照、语句快照以及只读语句快照隔离级别一起使用时，在快照启动时所提交的所有行的快照是可见的。从在其内打开游标的事务的启动开始，由当前连接所完成的所有修改也是可见的。有关支持的隔离级别的详细信息，请参见“隔离级别和一致性”一节 《SQL Anywhere 服务器 - SQL 的用法》和“isolation_level 选项 [数据库] [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》。

三层计算和分布式事务

目录

三层计算和分布式事务简介

可以将 SQL Anywhere 用作数据库服务器或**资源管理器**，以参与由事务服务器协调的分布式事务。

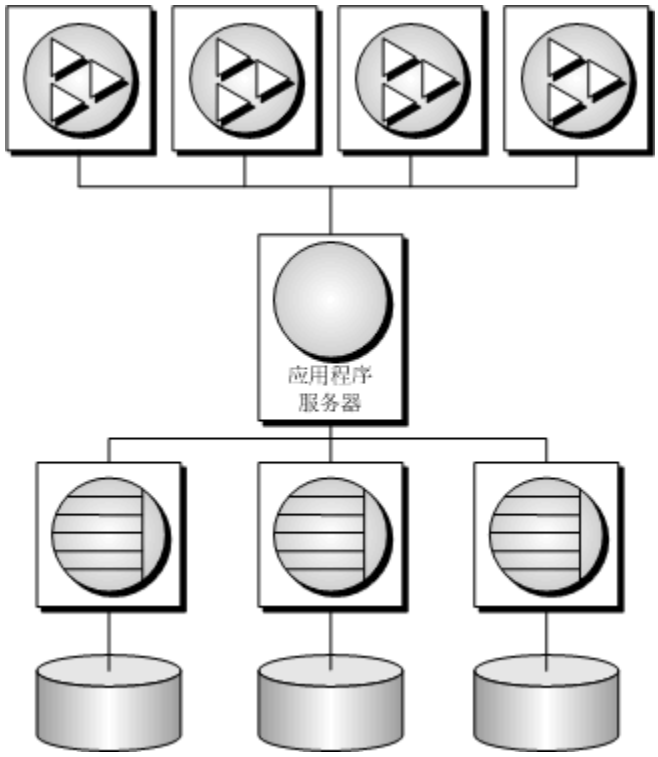
三层环境是一种常见的分布式事务环境，在此环境中应用程序服务器位于客户端应用程序和一组资源管理器之间。Sybase EAServer 及其它一些应用程序服务器也都是事务服务器。

Sybase EAServer 和 Microsoft Transaction Server 都使用 Microsoft 分布式事务处理协调器（Distributed Transaction Coordinator，简称 DTC）来协调事务。SQL Anywhere 提供了对 DTC 服务所控制的分布式事务的支持，因此可将 SQL Anywhere 用于上述两种应用程序服务器中的任意一种，也可用于任何其它基于 DTC 模型的产品。

将 SQL Anywhere 集成到三层环境中时，大多数工作都需要在应用程序服务器上完成。本章将介绍三层计算的概念和体系结构，并将概述 SQL Anywhere 的相关功能。但本章不介绍如何将应用程序服务器配置为与 SQL Anywhere 一起使用。有关详细信息，请参见应用程序服务器的文档。

三层计算体系结构

在三层计算中，应用程序逻辑保存在应用程序服务器（如 Sybase EAServer）上，该服务器位于资源管理器和客户端应用程序之间。在许多情况下，一个应用程序服务器可以访问多个资源管理器。在 Internet 环境中，客户端应用程序是基于浏览器的，而应用程序服务器通常是 Web 服务器扩展。



Sybase EAServer 以组件形式存储应用程序逻辑，并使这些组件可以供客户端应用程序使用。这些组件可以是 PowerBuilder 组件、JavaBean，也可以是 COM 组件。

有关详细信息，请参见您的 Sybase EAServer 文档。

三层计算中的分布式事务

客户端应用程序或应用程序服务器使用一个事务处理数据库（例如 SQL Anywhere）时，数据库外部并不需要事务逻辑；但是，如果使用多个资源管理器，事务控制就必须包括事务所涉及的各种资源。应用程序服务器会向它们的客户端应用程序提供事务逻辑，以保证操作集能够以原子方式执行。

许多事务服务器（包括 Sybase EAServer）都使用 Microsoft 分布式事务处理协调器（Distributed Transaction Coordinator，简称 DTC），向它们的客户端应用程序提供事务服务。DTC 使用 **OLE 事务**，而该事务又使用**两阶段提交**协议协调涉及多个资源管理器的事务。您必须安装 DTC，然后才能使用本章介绍的功能。

分布式事务中的 SQL Anywhere

SQL Anywhere 可以参与由 DTC 协调的事务，这意味着您可在使用事务服务器（例如 Sybase EAServer 或 Microsoft Transaction Server）的分布式事务中使用 SQL Anywhere 数据库。您还可直接在应用程序中使用 DTC 来协调多个资源管理器中的事务。

分布式事务词汇

本章假设您在一定程度上熟悉分布式事务。有关信息，请参见您的事务服务器文档。本节介绍一些常用的术语。

- **资源管理器**是那些对事务中涉及的数据进行管理的服务。

  在通过 OLE DB 或 ODBC 访问 SQL Anywhere 数据库服务器时，SQL Anywhere 数据库服务器可以用作分布式事务中的资源管理器。ODBC 驱动程序和 OLE DB 提供程序用作客户端计算机上的资源管理器代理。

- 应用程序组件并不直接与资源管理器通信，但可以与**资源分发器**通信，而资源分发器又管理与这些资源管理器的连接或连接池。

  SQL Anywhere 支持两种资源分发器：ODBC 驱动程序管理器和 OLE DB。

- 在事务组件请求数据库连接时（使用资源管理器），应用程序服务器会**征用**参与该事务的每个数据库连接。DTC 和资源分发器执行征用过程。

两阶段提交

分布式事务通过两阶段提交进行管理。当事务的工作完成时，事务管理器 (DTC) 会询问事务中征用的所有资源管理器是否准备提交该事务。此阶段称为**准备**提交。

如果所有资源管理器都作出准备提交的响应，则 DTC 会向每个资源管理器发送一个提交请求，并对其客户端作出事务已完成的响应。如果有一个或多个资源管理器不响应或者作出无法提交事务的响应，则事务的所有工作都将通过所有资源管理器进行回退。

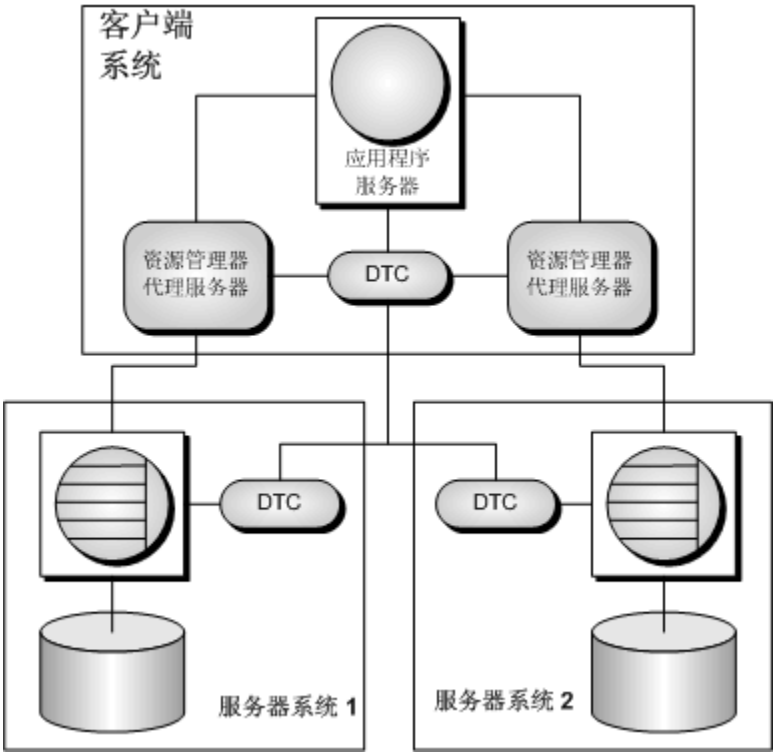
应用程序服务器如何使用 DTC

Sybase EAServer 和 Microsoft Transaction Server 都是组件服务器。应用程序逻辑以组件的形式保存，并且对客户端应用程序可用。

每个组件都有一个事务属性来指示组件参与事务的方式。创建组件的应用程序开发人员必须将事务的工作编为组件，即资源管理器连接，而资源管理器连接是每个资源管理器负责的对数据的操作。但是，应用程序开发人员不需要将事务管理逻辑添加到组件中。在设置了事务属性来指示组件需要事务管理之后，EAServer 即会使用 DTC 来征用事务并管理两阶段提交过程。

分布式事务体系结构

下图显示了分布式事务的体系结构。在此例中，资源管理器代理可以是 ODBC，也可以是 OLE DB。



此例中使用了一个资源分发器。应用程序服务器请求 DTC 准备事务。DTC 和资源分发器征用事务中的每个连接。每个资源管理器必须与 DTC 和数据库都保持联系，这样才能进行工作并在必要时向 DTC 通报其事务状态。

每台计算机上都必须运行 DTC 服务，才能操作分布式事务。可以通过 Windows [控制面板] 中的 [服务] 图标控制 DTC 服务；DTC 服务的名称为 **MSDTC**。

有关详细信息，请参见您的 DTC 或 EAServer 文档。

使用分布式事务

当 SQL Anywhere 在分布式事务中被征用时，它会将事务控制权交给事务服务器，而 SQL Anywhere 将确保它不会执行任何隐式事务管理。SQL Anywhere 参与分布式事务时会自动规定以下条件：

- 如果自动提交处于使用状态，则它自动关闭。
- 在分布式事务过程中不允许数据定义语句（其提交属于意外情况）。
- 由应用程序直接（而不是通过事务协调器）向 SQL Anywhere 发出的显式 COMMIT 或 ROLLBACK 会生成错误。但是该事务并不中止。
- 一个连接一次只能参与一个分布式事务。
- 当连接被征用到分布式事务中时一定不能有未提交的操作。

DTC 隔离级别

DTC 具有一组由应用程序服务器指定的隔离级别。这些隔离级别将按照以下方式映射到 SQL Anywhere 隔离级别：

DTC 隔离级别	SQL Anywhere 隔离级别
ISOLATIONLEVEL_UNSPECIFIED	0
ISOLATIONLEVEL_CHAOS	0
ISOLATIONLEVEL_READUNCOMMITTED	0
ISOLATIONLEVEL_BROWSE	0
ISOLATIONLEVEL_CURSORSTABILITY	1
ISOLATIONLEVEL_READCOMMITTED	1
ISOLATIONLEVEL_REPEATABLEREAD	2
ISOLATIONLEVEL_SERIALIZABLE	3
ISOLATIONLEVEL_ISOLATED	3

从分布式事务恢复

如果数据库服务器在未提交操作处于待执行状态时出现了故障，则它必须在启动时回退或者提交那些操作，以保持事务的原子特征。

如果在恢复过程中发现来自分布式事务的未提交操作，则数据库服务器将尝试连接到 DTC，并请求被重新征用到待执行或不确定的事务中。重新征用完成后，DTC 即会指示数据库服务器回退或提交未完成的操作。

如果重新征用过程失败，则 SQL Anywhere 将无法知道是应该提交还是应该回退不确定的操作，因而恢复将失败。如果要恢复此类状态下的数据库而不考虑数据的不确定状态，则可使用以下数据库服务器选项来强制恢复：

- **-tmf** 如果找不到 DTC，则未完成的操作将回退，并继续进行恢复。请参见“-tmf 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。
- **-tmt** 如果重新征用未能在指定的时间之前实现，则未完成的操作将回退，并继续进行恢复。请参见“-tmt 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

在 SQL Anywhere 中使用 EAServer

本节将概述在 EAServer 3.0 或更高版本中使用 SQL Anywhere 时需要执行的操作。有关详细信息，请参见 EAServer 文档。

配置 EAServer

在 Sybase EAServer 系统中安装的所有组件都共享同一事务处理协调器。

EAServer 3.0 以及更高版本提供了多个事务协调器以供用户选择。如果要将 SQL Anywhere 包括在事务中，必须将 DTC 用作事务协调器。本节介绍如何将 EAServer 3.0 配置为使用 DTC 作为其事务协调器。

EAServer 中的组件服务器名为 Jaguar。

♦ 配置 EAServer 以使用 Microsoft DTC 事务模型

1. 确保您的 Jaguar 服务器正在运行。

   在 Windows 上，Jaguar 服务器通常作为一个服务来运行。要手工启动已安装的 Jaguar 服务器（随 EAServer 3.0 提供），请选择 [**开始**] » [**程序**] » [**Sybase**] » [**EAServer**] » [**EAServer**]。

2. 启动 Jaguar Manager。

   在 Windows 桌面上，选择 [**开始**] » [**程序**] » [**Sybase**] » [**EAServer**] » [**Jaguar Manager**]。

3. 从 Jaguar Manager 连接到 Jaguar 服务器。

   从 Sybase Central 中，选择 [**工具**] » [**连接**] » [**Jaguar Manager**]。在连接窗口中，输入 **jagadmin** 作为 [**用户名**]，[**口令**] 字段保持空白，输入 **localhost** 作为 [**主机名**]。单击 [**确定**] 进行连接。

4. 为 Jaguar 服务器设置事务模型。

   在左窗格中，打开 [**Servers**] 文件夹。在右窗格中，右击要配置的服务器，然后选择 [**Server Properties**]。单击 [**Transactions**] 选项卡，然后选择 [**Microsoft DTC**] 作为事务模型。单击 [**OK**] 完成操作。

设置组件事务属性

在 EAServer 中，可实现一个在多个数据库上执行操作的组件。您可为此组件指派**事务属性**来定义它参与事务的方式。事务属性可具有以下值：

- **不支持**　组件的方法永远不会作为事务的一部分来执行。如果组件是由另一个在事务中执行的组件激活的，则新实例的工作将在现有事务之外执行。这是缺省设置。
- **支持事务**　组件可在事务的上下文中执行，但是在执行该组件的方法时不需要连接。如果组件是直接由基础客户端实例化的，EAServer 不会开始事务。如果组件 A 是由组件 B 实例化的，而组件 B 正在一个事务中执行，则组件 A 会在同一事务中执行。

- **需要事务** 组件总是在事务中执行。当组件由基础客户端直接实例化时，将开始新的事务。如果组件 A 是由组件 B 激活的，而组件 B 正在一个事务中执行，则组件 A 会在同一事务中执行；如果组件 B 不是在事务中执行的，则组件 A 会在一个新事务中执行。
- **需要新事务** 每当组件实例化时，都开始一个新的事务。如果组件 A 是由组件 B 激活的，而组件 B 正在一个事务中执行，则组件 A 将开始一个不受事务 B 的事务结果影响的新事务；如果组件 B 不是在事务中执行的，则组件 A 将在一个新事务中执行。

例如，在 Sybase Virtual University 示例应用程序（EAServer 中附带的 SVU 包）中，SVUEnrollment 组件的 enroll 方法执行两种不同的操作（预定课程座位，向学生收取课程费）。需要将这两种操作视为一个事务。

Microsoft Transaction Server 提供相同的属性值集。

♦ 设置组件的事务属性

1. 在 Jaguar Manager 中找到该组件。

   要找到 Jaguar 示例应用程序中的 SVUEnrollment 组件，请连接到 Jaguar 服务器，打开 [**Packages**] 文件夹，然后打开 SVU 程序包。该程序包中的组件列于右窗格中。

2. 设置组件的事务属性。

   右击该组件，然后选择 [**Component Properties**]。单击 [**Transaction**] 选项卡，从列表中选择事务属性值。单击 [**OK**] 完成操作。

   SVUEnrollment 组件已经标上了 [**Requires Transaction**] 标记。

设置了组件事务特性后，即可从该组件执行 SQL Anywhere 数据库操作，并可确保事务在您指定的级别进行处理。

数据库中的 Java

本节介绍 Java 和数据库中的 Java。

SQL Anywhere 中的 Java 支持

目录

Java 支持简介

SQL Anywhere 提供了用于在数据库服务器环境中执行 Java 类的机制。在数据库服务器中使用 Java 方法为向数据库添加编程逻辑提供了有效方式。

数据库中的 Java 支持具有下列优点：

- 您可以在应用程序的不同层级（客户端、中间层或服务器）中重复使用 Java 组件，哪个层级最适用，就将它们用于哪个层级。SQL Anywhere 成为了用于分布式计算的平台。
- 与 SQL 存储过程语言相比，Java 为在数据库中内置逻辑提供了一种功能更强大的语言。
- Java 可用于数据库服务器中，但不会危害数据库和服务器的完整性、安全性或可靠性。

SQLJ 标准

数据库中的 Java 基于 SQLJ 第 1 部分建议的标准 (ANSI/INCITS 331.1-1999) 构建而成。SQLJ 第 1 部分提供了有关将 Java 静态方法作为 SQL 存储过程和函数来调用的规范。

了解数据库中的 Java

下表列出了有关在数据库中使用 Java 的文档。

标题	作用
“SQL Anywhere 中的 Java 支持” 第 73 页（本章）	Java 概念以及如何在 SQL Anywhere 中应用它们。
“创建 Java 类以与 SQL Anywhere 配合使用”一节第 83 页	在数据库中使用 Java 的实践步骤。
“SQL Anywhere JDBC 驱动程序” 第 481 页	从 Java 类访问数据，包括分布式计算。

下表可帮助您根据您的兴趣和背景确定 Java 文档的哪些部分适合您。

如果……	请考虑阅读……
是希望立刻开始的 Java 开发人员。	“数据库中的 Java 的运行时环境”一节 第 79 页 “创建 Java 类以与 SQL Anywhere 配合使用”一节第 83 页
希望了解数据库中 Java 的主要特性。	“关于数据库中 Java 的问答”一节第 75 页
希望了解如何从 Java 访问数据。	“SQL Anywhere JDBC 驱动程序”第 481 页

关于数据库中 Java 的问答

本节介绍数据库中的 Java 的主要功能。

数据库中的 Java 有哪些主要功能？

下面所有要点的详细解释将在后面的几节中提供。

- **可以在数据库服务器中运行 Java** 外部 Java 虚拟机 (VM) 在数据库服务器中运行 Java 代码。
- **可以从 Java 访问数据** 内部 JDBC 驱动程序可让您从 Java 访问数据。
- **保留 SQL** 使用 Java 不会改变现有 SQL 语句的行为，也不会改变非 Java 关系数据库行为的其它方面。

如何在数据库中存储 Java 类？

Java 是一种面向对象的语言，因此它的指令（源代码）采用类的形式。要在数据库中执行 Java，应在数据库外编写 Java 指令并在数据库外将它们编译为已编译的类（**字节代码**），这些类是包含 Java 指令的二进制文件。

然后，将这些已编译的类安装到数据库中。安装之后，便可以在数据库服务器中将这些类作为存储过程来执行了。例如，下面的语句将创建到 Java 过程的接口：

```
CREATE PROCEDURE insertfix()
EXTERNAL NAME 'JDBCExample.InsertFixed()V'
LANGUAGE JAVA;
```

SQL Anywhere 是 Java 类的运行时环境，而不是 Java 开发环境。您需要一个 Java 开发环境（如 Sun Microsystems Java 开发工具包）来编写和编译 Java。您还需要具备 Java 运行时环境才能执行 Java 类。

有关详细信息，请参见“将 Java 类安装到数据库中”一节第 92 页。

Java 在数据库中是如何执行的？

SQL Anywhere 使用的是 **Java 虚拟机 (VM)**。Java VM 将解释已编译的 Java 指令并代表数据库服务器来运行它们。数据库服务器会在需要时自动启动 Java VM：您不必执行任何显式操作来启动或停止 Java VM。

数据库服务器中的 SQL 请求处理器已经进行了扩展，因此它可以向 Java VM 中发出调用来执行 Java 指令。另外，它还可以处理来自 Java VM 的请求以便能够从 Java 进行数据访问。

为什么用 Java？

Java 具有的许多功能使得它非常适合用在数据库中，这些功能包括：

- 编译时全面错误检查。
- 具有定义明确的错误处理方法的内置错误处理。
- 内置的垃圾回收（内存恢复）。
- 摈弃了许多容易出错的编程技术。
- 强大的安全功能。
- Java 代码是解释执行的，因此，如果操作不被 Java VM 接受，则将无法执行。

哪些平台支持数据库中的 Java？

数据库中的 Java 在所有 Unix 和 Windows 操作系统（Windows Mobile 除外）上均受支持。

如何结合使用 Java 和 SQL？

Java 方法被声明为存储过程，然后就可以象调用 SQL 存储过程一样调用它们。

您可以使用 Sun Microsystems Java 开发工具包附带的 Java API 中的许多类。您还可以使用 Java 开发人员创建和编译的类。

如何从 SQL 访问 Java？

可以将 Java 方法视为存储过程，可以从 SQL 进行调用。

必须创建一个用于运行方法的存储过程。例如：

```
CREATE PROCEDURE javaproc()
EXTERNAL NAME 'JDBCExample.MyMethod ()V'
LANGUAGE JAVA;
```

有关详细信息，请参见“CREATE PROCEDURE 语句（Web 服务）”一节 《SQL Anywhere 服务器 - SQL 参考》。

例如，SQL 函数 PI(*) 返回 pi 的值。Java API 类 java.lang.Math 有一个名为 PI 并返回同一值的并行字段。但是，java.lang.Math 还有一个名为 E 且返回自然对数的底的字段，以及一个按照 IEEE 754 标准的规定对两个参数进行余数运算计算的方法。

Java API 的其它成员甚至提供了更专门化的功能。例如，java.util.Stack 将生成一个可存储有序列表的后进先出队列；java.util.HashTable 会将各值映射到各键；而 java.util.StringTokenizer 会将字符串分解成单个的字单元。

如何在数据库中使用自己的 Java 类？

可以将您自己的 Java 类安装到数据库中。例如，您可以用 Java 设计、编写并用 Java 编译器编译用户创建的 Employees 类或 Package 类。

用户创建的 Java 类可以同时包含有关主题的信息和某些计算逻辑。一旦安装在数据库中，SQL Anywhere 即允许您在数据库的所有部分和操作中使用这些类并执行它们的功能（以类方法或实例方法的形式），就像调用存储过程那样容易。

Java 类和存储过程不同
Java 类不同于存储过程。存储过程是以 SQL 编写的，而 Java 类提供了功能更强大的语言，并且可以像调用存储过程一样轻松地从客户端应用程序中调用。

有关详细信息，请参见“将 Java 类安装到数据库中”一节第 92 页。

可以使用 Java 访问数据吗？

JDBC 接口是一个行业标准，专门为访问数据库系统而设计。根据设计，JDBC 类用于连接到数据库，使用 SQL 语句请求数据，并返回能够在客户端应用程序中进行处理的结果集。

通常情况下，客户端应用程序使用 JDBC 类，而数据库系统供应商提供 JDBC 驱动程序，JDBC 类使用该驱动程序建立连接。

可以通过 JDBC（使用 jConnect 或 iAnywhere JDBC 驱动程序）从客户端应用程序连接到 SQL Anywhere。SQL Anywhere 还提供了一个内部 JDBC 驱动程序，该驱动程序允许安装在数据库中的 Java 类使用执行 SQL 语句的 JDBC 类。请参见“SQL Anywhere JDBC 驱动程序”第 481 页。

是否可以将类从客户端移到服务器？

您可以创建能够在企业应用程序的不同层级之间移动的 Java 类。同一个 Java 类可以集成到客户端应用程序、中间层或者数据库中，哪个层级最适合，就集成到哪个层级中。

您可以将包含业务逻辑的类移到企业系统的任何层级（包括数据库服务器），这为您最恰当地利用资源提供了全面的灵活性。它还使企业客户能够以无与伦比的灵活性在多层体系结构中使用单一编程语言开发他们的应用程序。

用数据库中的 Java 不能做什么？

SQL Anywhere 是 Java 类的运行时环境，而不是 Java 开发环境。

您无法在数据库中执行下列任务：

- 编辑类源文件（*.*java* 文件）。
- 编译 Java 类源文件（*.*java* 文件）。
- 执行不受支持的 Java API，如小程序和可视类。
- 执行需要执行本地方法的 Java 方法。安装到数据库中的所有用户类都必须是百分之百的 Java 类。

必须使用 Java 应用程序开发工具编写和编译 SQL Anywhere 中使用的 Java 类，然后将这些类安装到数据库中以供使用。

Java 错误处理

Java 错误处理代码与用于普通处理的代码是分开的。

错误会生成一个表示错误的异常对象。这叫做**抛出异常**。如果未能在应用程序的某个层级捕获并适当处理抛出的异常，则该异常将终止 Java 程序。

无论是 Java API 类还是自定义创建的类都有可能抛出异常。事实上，用户可以创建自己的异常类，这些类会抛出其自定义创建的类。

如果发生异常的方法的主体中没有异常处理程序，则会继续沿着调用堆栈向上搜索异常处理程序。如果到达调用堆栈的顶部仍未找到异常处理程序，则会调用运行该应用程序的 Java 解释器的缺省异常处理程序，同时程序会终止。

在 SQL Anywhere 中，如果 SQL 语句调用 Java 方法，并抛出了未处理的异常，则会生成一个 SQL 错误。

数据库中的 Java 的运行时环境

本节介绍 SQL Anywhere 的 Java 运行时环境，以及它与标准的 Java 运行时环境之间的差异。

运行时 Java 类

运行时 Java 类属于低层类，在创建数据库或为数据库启用 Java 时会将这些类提供给数据库。这些类包含 Java API 的一个子集，它们是 Sun Java 开发工具包的组成部分。

运行时类提供了构建应用程序所基于的基本功能。数据库中的类始终可以使用这些运行时类。

可将运行时 Java 类合并到用户自己创建的类中：或者继承其功能，或者在某一方法的计算或操作中使用它。

示例

运行时 Java 类中包含了一些 Java API 类，其中包括：

- **基元 Java 数据类型** Java 中的所有基元（本地）数据类型都有一个对应的类。除了能够创建这些类型的对象之外，这些类还具有在通常情况下都很有用的其它功能。

  Java int 数据类型在 java.lang.Integer 中有对应的类。

- **实用程序包** 包 java.util.* 中包含许多类，这些类所具备的功能在 SQL Anywhere SQL 函数中尚未提供。

  现将这些类中的一部分列举如下：

  - **Hashtable** 将键映射到值。
  - **StringTokenizer** 将字符串分解成各个单独的字。
  - **Vector** 保存大小可动态变化的对象的数组。
  - **Stack** 保存后进先出的对象堆栈。

- **用于 SQL 操作的 JDBC** 包 java.SQL.* 包含了 Java 对象使用 SQL 语句从数据库中提取数据时所需要的类。

与用户定义的类不同，运行时类不存储在数据库中，而是存储在 Sun JRE 的安装位置。

Java 区分大小写

Java 语法会按您的预期工作，而 SQL 语法不会因 Java 类的存在而发生改变。即使同一 SQL 语句同时包含 Java 和 SQL 语法，情况也是如此。它是一个简单语句，但其含义却很深远。

Java 是区分大小写的。Java 类 FindOut 与类 Findout 是两个完全不同的类。SQL 在关键字和标识符方面不区分大小写。

即使将 Java 嵌入到不区分大小写的 SQL 语句中，Java 也会保留区分大小写的特性。语句的 Java 部分必须区分大小写，即使 Java 语法前面和后面的部分不区分大小写也是如此。

例如，下面的 SQL 语句能够成功执行，这是因为在 Java 对象、类和运算符中遵守了大小写约定，即使该语句中其余 SQL 部分的大小写有所变化。

```
SeLeCt java.lang.Math.random();
```

Java 和 SQL 中的字符串

一对双引号在 Java 中标识字符串常值，如下面的 Java 代码片段所示：

```
String str = "This is a string";
```

但在 SQL 中，如以下 SQL 语句所示，单引号标记字符串，而双引号表示标识符：

```
INSERT INTO TABLE DBA.t1
VALUES( 'Hello' );
```

在 Java 源代码中应始终使用双引号，在 SQL 语句中则始终使用单引号。

下面的 Java 代码段是有效的（如果在 Java 类内使用）。

```
String str = new java.lang.String(
             "Brand new object" );
```

打印到命令行

打印到标准输出是检查代码执行中的各个点的变量值和执行结果的快速方式。在遇到下面的 Java 代码段第二行中的方法时，它接受的字符串参数将打印到标准输出。

```
String str = "Hello world";
System.out.println( str );
```

在 SQL Anywhere 中，标准输出是数据库服务器消息窗口，因此字符串会在此处显示。在数据库内执行上面的 Java 代码相当于执行下面的 SQL 语句。

```
MESSAGE 'Hello world';
```

使用 main 方法

当类包含与下面的声明匹配的 main 方法时，大多数 Java 运行时环境（如 Sun Java 解释器）都会自动执行它。通常情况下，此静态方法只有是被 Java 解释器调用的类时才会执行。

```
public static void main( String args[ ] ) { }
```

在 Sun Java 运行时系统启动时，始终要保证首先调用此方法。

在 SQL Anywhere 中，Java 运行时系统始终可用。可以使用 SQL 语句以一种即席动态方式测试对象和方法的功能。这为测试 Java 类功能提供了一种灵活方法。

持久性

一旦 Java 类添加到数据库中，它将一直保留在数据库中，直到您使用 REMOVE JAVA 语句将其显式删除。

Java 类中的变量（如 SQL 变量）只在连接期间保留。

有关删除类的详细信息，请参见“REMOVE JAVA 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

SQL 语句中的 Java 转义字符

在 Java 代码中，可以使用转义字符在字符串中插入某些特殊字符。请看下面的代码，该代码在一个包含撇号的句子前面插入一个新行和制表符。

```
String str = "\n\t\This is an object\'s string literal";
```

只有在 Java 类使用 Java 转义字符时，SQL Anywhere 才允许使用 Java 转义字符。但在 SQL 内，必须遵守适用于 SQL 中的字符串的规则。

例如，要给使用 SQL 语句的字段传递字符串值，您可以使用下面的语句（包括 SQL 转义字符），但不能使用 Java 转义字符。

```
SET obj.str = '\nThis is the object''s string field';
```

有关 SQL 字符串处理规则的详细信息，请参见“字符串”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用 import 语句

在 Java 类声明中包含一个 import 语句以访问另一个包中的类是很常见的情况。您可以使用非限定类名引用导入的类。

例如，您可以采用两种方式引用 java.util 包的 Stack 类：

- 显式使用名称 java.util.Stack
- 使用名称 Stack，并包含下面的 import 语句：

```
import java.util.*;
```

还必须安装层次中更高层的类

由另一个类引用的类，无论是用完全限定名称显式引用的还是使用 import 语句隐式引用的，也都必须安装在数据库中。

import 语句在已编译的类内按预期工作。但在 SQL Anywhere 运行时环境中，不存在 import 语句的对等项。存储过程中所使用的所有类名都必须是完全限定的。例如，要创建 String 类型的变量，则使用这个完全限定的名称来引用该类：java.lang.String。

公共字段

在面向对象的编程中，将类字段定义为 private，且只通过 public 方法提供它们的值，这是一种常见做法。

本文档中使用的许多示例都使字段成为 public 字段，以使示例更紧凑和易于阅读。与访问 public 方法相比，在 SQL Anywhere 中使用 public 字段还提高了性能。

本文档中遵循的一般约定是：用户创建的用于 SQL Anywhere 的 Java 类在其字段中公开其主要值。方法中将包含可以作用于这些字段的计算自动化和逻辑。

创建 Java 类以与 SQL Anywhere 配合使用

以下几节介绍了创建 Java 方法并从 SQL 中调用这些方法时所涉及的步骤。其中展示了如何编译 Java 类以及如何将其安装到数据库中，使其可供在 SQL Anywhere 中使用。还介绍了如何从 SQL 语句访问该类及其成员和方法。

以下几节假定您已安装了 Java 开发工具包（Java Development Kit，简称 JDK），包括 Java 编译器 (javac) 和 Java VM。

samples-dir\SQLAnywhere\JavaInvoice 中提供了此示例的源代码和批处理文件。

在数据库中使用 Java 的第一步是编写 Java 代码并对其进行编译。此项任务在数据库外完成。

♦ 创建和编译类

1. 创建示例 Java 类源文件。

   为方便起见，在此提供了示例代码。可以将以下代码粘贴到 *Invoice.java* 中，或从 *samples-dir\SQLAnywhere\JavaInvoice* 获取该文件。

```
import java.io.*;

public class Invoice
{
    public static String lineItem1Description;
    public static double lineItem1Cost;

    public static String lineItem2Description;
    public static double lineItem2Cost;

    public static double totalSum() {
        double runningsum;
        double taxfactor = 1 + Invoice.rateOfTaxation();

        runningsum = lineItem1Cost + lineItem2Cost;
        runningsum = runningsum * taxfactor;

        return runningsum;
    }

    public static double rateOfTaxation()
    {
        double rate;
        rate = .15;

        return rate;
    }

    public static void init(
      String item1desc, double item1cost,
      String item2desc, double item2cost )
    {
        lineItem1Description = item1desc;
        lineItem1Cost = item1cost;
        lineItem2Description = item2desc;
        lineItem2Cost = item2cost;
    }

    public static String getLineItem1Description()
    {
```

```
        return lineItem1Description;
    }

    public static double getLineItem1Cost()
    {
        return lineItem1Cost;
    }

    public static String getLineItem2Description()
    {
        return lineItem2Description;
    }

    public static double getLineItem2Cost()
    {
        return lineItem2Cost;
    }

    public static boolean testOut( int[] param )
    {
        param[0] = 123;
        return true;
    }

    public static void main( String[] args )
    {
        System.out.print( "Hello" );
        for ( int i = 0; i  < args.length; i++ )
            System.out.print( " " + args[i] );
        System.out.println();
    }
}
```

2. 编译该文件以创建文件 *Invoice.class*。

   ```
   javac Invoice.java
   ```

   该类现已被编译并随时都可以安装到数据库中。

选择 Java VM

必须设置数据库服务器才能找到 Java VM。由于可为每个数据库指定不同的 Java VM，因此可使用 ALTER EXTERNAL ENVIRONMENT 语句指明 Java VM 的位置（路径）。

```
ALTER EXTERNAL ENVIRONMENT JAVA
LOCATION 'c:\\jdk1.5.0_06\\jre\\bin\\java.exe';
```

如果未设置此位置，则数据库服务器将按如下顺序搜索 Java VM 的位置：

- 检查 JAVA_HOME 环境变量。
- 检查 JAVAHOME 环境变量。
- 检查路径。
- 如果信息不在该路径中，则返回一个错误。

请参见“ALTER EXTERNAL ENVIRONMENT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

注意
JAVA_HOME 和 JAVAHOME 是通常在安装 Java VM 时创建的环境变量。如果这两个环境变量都不存在，可手工创建它们，然后将其指向 Java VM 的根目录。但在使用 ALTER EXTERNAL ENVIRONMENT 语句时不需要这样做。

♦ 指定 Java VM (Interactive SQL) 的位置

1. 启动 Interactive SQL 并连接到数据库。
2. 在 [SQL 语句] 窗格中键入以下语句：

   ```
   ALTER EXTERNAL ENVIRONMENT JAVA
   LOCATION 'path\\java.exe';
   ```

   其中，*path* 表示 Java VM 的位置（例如， *c:\\jdk1.5.0_06\\jre\\bin*）。

也可使用 ALTER EXTERNAL ENVIRONMENT 语句指定其连接可用于安装类和执行其它与 Java 相关管理任务的数据库用户。

```
ALTER EXTERNAL ENVIRONMENT JAVA
    USER user_name
```

有关详细信息，请参见“ALTER EXTERNAL ENVIRONMENT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用 java_vm_options 选项来指定启动 Java VM 所需的任何附加命令行选项。

```
SET OPTION PUBLIC.java_vm_options='java-options';
```

有关详细信息，请参见“java_vm_options 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

如果您想在数据库中使用 JAVA，但又没有安装 Java 运行时环境（Java Runtime Environment，简称 JRE），那么您可以安装和使用您想要的任何 Java JRE。一旦安装完毕，最好将 JAVA_HOME

或 JAVAHOME 环境变量设置为指向已安装的 JRE 的根。注意，大部分的 Java 安装程序在缺省情况下设置这些环境变量之一。JRE 安装完毕并正确设置了 JAVA_HOME 或 JAVAHOME 之后，您便可在不执行任何附加步骤的情况下在数据库中使用 Java。

安装示例 Java 类

必须先将 Java 类安装到数据库中，然后才能使用它们。可以从 Sybase Central 或 Interactive SQL 安装类。

♦ 将类安装到 SQL Anywhere 示例数据库 (Sybase Central)

1. 启动 Sybase Central 并连接到示例数据库。
2. 在左窗格中，展开 [**外部环境**] 文件夹。
3. 单击 [**Java**]。
4. 选择 [**文件**] » [**新建**] » [**Java 类**]。
5. 单击 [**浏览**] 浏览到 *Invoice.class* 的位置。
6. 单击 [**完成**]。

♦ 将类安装到 SQL Anywhere 示例数据库 (Interactive SQL)

1. 启动 Interactive SQL 并连接到示例数据库。
2. 在 Interactive SQL 的 [**SQL 语句**] 窗格中，键入以下语句：

   ```
   INSTALL JAVA NEW
   FROM FILE 'path\\Invoice.class';
   ```

   其中，*path* 是您已编译的类文件的位置。
3. 按 F5 键以执行该语句。

该类现在已安装到示例数据库中。

注意

- 此时，数据库操作中还未发生任何 Java 操作。现在类已安装到数据库中，随时都可以使用。
- 从现在起，对该类文件所进行的更改将 *不会* 自动反映在数据库中该类的副本中。如果您想要反映出这些更改，则必须在数据库中更新这些类。

有关安装类的详细信息以及有关更新已安装类的信息，请参见“将 Java 类安装到数据库中”一节 第 92 页。

使用 CLASSPATH 变量

Sun 的 Java 运行时环境和 Sun JDK Java 编译器使用 CLASSPATH 环境变量来查找 Java 代码内引用的类。CLASSPATH 变量提供了 Java 代码与实际文件路径或所引用类的 URL 位置之间的链接。例如，`import java.io.*` 允许在不使用完全限定名称的情况下引用 java.io 包中的所有类。下面的 Java 代码中只需要类名即可使用 java.io 包中的类。将要编译 Java 类声明的系统上的 CLASSPATH 环境变量必须包含 Java 目录的位置，即 java.io 包的根目录。

用于安装类的 CLASSPATH

CLASSPATH 变量可以在安装类期间用于查找文件。例如，下面的语句将一个用户创建的 Java 类安装到数据库中，但只指定了文件名称，而未指定其完整路径和名称。（请注意，此语句不涉及 Java 操作。）

```
INSTALL JAVA NEW
FROM FILE 'Invoice.class';
```

如果指定的文件位于 CLASSPATH 环境变量所指定的目录或 ZIP 文件中，则 SQL Anywhere 将成功找到该文件并安装该类。

访问 Java 类中的方法

若要访问该类中的 Java 方法，则必须创建存储过程或函数来充当该类中方法的包装。

♦ 使用 Interactive SQL 调用 Java 方法

1. 创建下面的 SQL 存储过程来调用示例类中的 Invoice.main 方法：

   ```
   CREATE PROCEDURE InvoiceMain( IN arg1 CHAR(50) )
   EXTERNAL NAME 'Invoice.main([Ljava/lang/String;)V'
   LANGUAGE JAVA;
   ```

   此存储过程将充当该 Java 方法的包装。

   有关此语句的语法的详细信息，请参见“CREATE PROCEDURE 语句（Web 服务）”一节《SQL Anywhere 服务器 - SQL 参考》。

2. 调用存储过程以调用 Java 方法：

   ```
   CALL InvoiceMain('to you');
   ```

   如果查看数据库服务器消息日志，您会看见那里写有 "Hello to you" 消息。数据库服务器已将输出从 System.out 重定向到那里。

访问 Java 对象的字段和方法

以下列举了更多有关如何调用 Java 方法、传递参数和返回值的示例。

♦ 为 Invoice 类中的方法创建存储过程/函数

1. 创建下面的 SQL 存储过程以将参数传递给 Invoice 类中的 Java 方法并从中检索返回值：

```
-- Invoice.init takes a string argument (Ljava/lang/String;)
-- a double (D), a string argument (Ljava/lang/String;), and
-- another double (D), and returns nothing (V)
CREATE PROCEDURE init( IN arg1 CHAR(50),
                       IN arg2 DOUBLE,
                       IN arg3 CHAR(50),
                       IN arg4 DOUBLE)
EXTERNAL NAME
  'Invoice.init(Ljava/lang/String;DLjava/lang/String;D)V'
LANGUAGE JAVA;
-- Invoice.rateOfTaxation take no arguments ()
-- and returns a double (D)
CREATE FUNCTION rateOfTaxation()
RETURNS DOUBLE
EXTERNAL NAME
  'Invoice.rateOfTaxation()D'
LANGUAGE JAVA;
-- Invoice.rateOfTaxation take no arguments ()
-- and returns a double (D)
CREATE FUNCTION totalSum()
RETURNS DOUBLE
EXTERNAL NAME
  'Invoice.totalSum()D'
LANGUAGE JAVA;
-- Invoice.getLineItem1Description take no arguments ()
-- and returns a string (Ljava/lang/String;)
CREATE FUNCTION getLineItem1Description()
RETURNS CHAR(50)
EXTERNAL NAME
  'Invoice.getLineItem1Description()Ljava/lang/String;'
LANGUAGE JAVA;
-- Invoice.getLineItem1Cost take no arguments ()
-- and returns a double (D)
CREATE FUNCTION getLineItem1Cost()
RETURNS DOUBLE
EXTERNAL NAME
  'Invoice.getLineItem1Cost()D'
LANGUAGE JAVA;
-- Invoice.getLineItem2Description take no arguments ()
-- and returns a string (Ljava/lang/String;)
CREATE FUNCTION getLineItem2Description()
RETURNS CHAR(50)
EXTERNAL NAME
  'Invoice.getLineItem2Description()Ljava/lang/String;'
LANGUAGE JAVA;
-- Invoice.getLineItem2Cost take no arguments ()
-- and returns a double (D)
CREATE FUNCTION getLineItem2Cost()
RETURNS DOUBLE
EXTERNAL NAME
  'Invoice.getLineItem2Cost()D'
LANGUAGE JAVA;
```

传递给 Java 方法的参数以及从 Java 方法检索的返回值的描述符具有以下含义：

字段类型	Java 数据类型
B	byte
C	char
D	double
F	float
I	int
J	long
L *class-name*;	类 *class-name* 的实例。类名必须是完全限定的，而且名称中的任何点都必须替换为 /。例如，**java/lang/String**。
S	short
V	void
Z	Boolean
[	数组的每个维度都使用一个。

有关这些语句的语法的详细信息，请参见“CREATE PROCEDURE 语句（Web 服务）”一节《SQL Anywhere 服务器 - SQL 参考》和“CREATE FUNCTION 语句（Web 服务）”一节《SQL Anywhere 服务器 - SQL 参考》。

2. 调用充当包装的存储过程以调用 Java 方法：

```
CALL init('Shirt',10.00,'Jacket',25.00);
SELECT getLineItem1Description() as Item1,
       getLineItem1Cost() as Item1Cost,
       getLineItem2Description() as Item2,
       getLineItem2Cost() as Item2Cost,
       rateOfTaxation() as TaxRate,
       totalSum() as Cost;
```

该查询将返回含有以下值的 6 个列：

Item1	Item1Cost	Item2	Item2Cost	TaxRate	Cost
Shirt	10	Jacket	25	0.15	40.25

将 Java 类安装到数据库中

可按照如下形式将 Java 类安装到数据库中：

- **单一类** 可从已编译的类文件中将单个类安装到数据库中。类文件的扩展名通常为 *.class*。
- **JAR** 如果某组类位于已压缩或未压缩的 JAR 文件中，您可以一次性地安装这组中的所有类。JAR 文件的扩展名通常为 *.jar* 或 *.zip*。SQL Anywhere 支持所有用 Sun JAR 实用程序创建的 JAR 压缩文件以及其它一些 JAR 压缩模式。

创建类

您在创建自己的类时会涉及许多步骤，每一步的具体情况会因您是否使用 Java 开发工具而有所不同，但一般都会包括以下步骤：

♦ 创建类

1. 定义类。

   编写定义类的 Java 代码。如果您使用的是 Sun Java SDK，则可以使用文本编辑器。如果您使用的是开发工具，该开发工具会提供相应说明。

   只使用支持的类
   用户类必须是百分之百的 Java 类。不允许使用本地方法。

2. 命名和保存类。

   将类声明（Java 代码）保存在扩展名为 *.java* 的文件中。确保文件名与类名相同，并且这两个名称的大小写一致。

   例如，名为 Utility 的类应保存在名为 *Utility.java* 的文件中。

3. 编译类。

   此步骤会将包含 Java 代码的类声明转化为一个不同的包含字节代码的新文件。新文件的名称与 Java 代码文件的名称相同，但是扩展名为 *.class*。您可以在 Java 运行时环境中运行已编译的 Java 类，而不必考虑编译它时所使用的平台或运行时环境的操作系统是什么。

   Sun JDK 包含一个 Java 编译器 *javac*。

安装类

为使 Java 类在数据库内可用，可从 Sybase Central 将该类安装到数据库中，也可以从 Interactive SQL 或另一个应用程序使用 INSTALL JAVA 语句进行安装。您必须知道要安装的类的路径和文件名。

您需要 DBA 权限来安装类。

♦ **安装类 (Sybase Central)**

1. 以 DBA 用户身份连接到数据库。
2. 打开 [**外部环境**] 文件夹。
3. 在此文件夹下，打开 [**Java**] 文件夹。
4. 右击右窗格，然后选择 [**新建**] » [**Java 类**]。
5. 请按照向导中的说明进行操作。

♦ **安装类 (SQL)**

1. 以 DBA 用户身份连接到数据库。
2. 执行以下语句：

```
INSTALL JAVA NEW
FROM FILE 'path\\ClassName.class';
```

path 是类文件所位于的目录，*ClassName.class* 是类文件的名称。

双反斜线可确保斜线不被视为转义字符。

例如，要安装名为 *Utility.class* 的文件（保存在目录 *c:\source* 中）中的类，请执行以下语句：

```
INSTALL JAVA NEW
FROM FILE 'c:\\source\\Utility.class';
```

如果使用相对路径，它必须相对于数据库服务器的当前工作目录。

有关详细信息，请参见“INSTALL JAVA 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

安装 JAR

一个有用且常见的做法是，将各组相关的类收集到各个包中，然后将一个或多个包存储在一个 **JAR 文件** 中。

JAR 文件的安装方式与类文件的安装方式相同。JAR 文件的扩展名可以为 JAR 或 ZIP。每个 JAR 文件在数据库中必须有一个名称。通常，您使用 JAR 文件的名称，但不带扩展名。例如，如果安装名为 *myjar.zip* 的 JAR 文件，则通常为它指定 *myjar* 这一 JAR 名。

有关详细信息，请参见“INSTALL JAVA 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

♦ **安装 JAR (Sybase Central)**

1. 以 DBA 用户身份连接到数据库。
2. 打开 [**外部环境**] 文件夹。
3. 在此文件夹下，打开 [**Java**] 文件夹。
4. 右击右窗格，然后选择 [**新建**] » [**JAR 文件**]。

5. 请按照向导中的说明进行操作。

♦ 安装 JAR (SQL)

1. 以 DBA 用户身份连接到数据库。
2. 执行以下语句：

```
INSTALL JAVA NEW
JAR 'jarname'
FROM FILE 'path\\JarName.jar';
```

更新类和 JAR 文件

您可以使用 Sybase Central 更新类和 JAR 文件，也可以通过在 Interactive SQL 或其它某个客户端应用程序中执行 INSTALL JAVA 语句来进行更新。

要更新类或 JAR，您必须具有 DBA 权限，并且磁盘上的某个文件中要有较新版本的已编译的类文件或 JAR 文件。

更新的类何时生效

只有在安装类之后建立的新连接或者在安装类之后首次使用类的新连接才使用新定义。一旦 Java VM 装载了某个类定义，它就会一直保留在内存中，直到连接关闭。

使用基于当前连接中的某个 Java 类或类的对象，那么，要使用新的类定义，就需要断开连接并重新连接。

♦ 更新类或 JAR (Sybase Central)

1. 以 DBA 用户身份连接到数据库。
2. 打开 [**外部环境**] 文件夹。
3. 在此文件夹下，打开 [**Java**] 文件夹。
4. 查找含有要更新的类或 JAR 文件的子文件夹。
5. 选择类或 JAR 文件并选择 [**文件**] » [**更新**]。
6. 在 [**更新**] 窗口中，指定要更新的类或 JAR 文件的名称和位置。可以单击 [**浏览**] 来搜索它。

提示

也可以通过右击类或 JAR 文件名并选择 [**更新**] 来更新 Java 类或 JAR 文件。

也可以单击 [**属性**] 窗口 [**常规**] 选项卡上的 [**立即更新**] 来更新 Java 类或 JAR 文件。

♦ 更新类或 JAR (SQL)

1. 以 DBA 用户身份连接到数据库。
2. 执行以下语句：

```
INSTALL JAVA UPDATE
[ JAR 'jarname' ]
FROM FILE 'filename';
```

如果要更新 JAR，必须输入在数据库中用于标识该 JAR 的名称。请参见“INSTALL JAVA 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

数据库中 Java 类的特殊功能

本节介绍在数据库中使用的 Java 类的功能。

调用 main 方法

通常都是通过对具有 main 方法的类运行 Java VM 来启动 Java 应用程序（在数据库外部）。

例如，文件 *samples-dir\SQLAnywhere\JavaInvoice\Invoice.java* 中的 Invoice 类具有一个 main 方法。当使用如下所示的命令从命令行执行该类时，执行的是 main 方法：

```
java Invoice
```

♦ 从 SQL 调用类的 main 方法

1. 声明将字符串数组作为参数的方法：

   ```
   public static void main( java.lang.String args[] )
   {
   ...
   }
   ```

2. 创建一个包装此方法的存储过程。

   ```
   CREATE PROCEDURE JavaMain( in arg char(50) )
   EXTERNAL NAME 'JavaClass.main([Ljava/lang/String;)V'
   LANGUAGE JAVA;
   ```

   有关详细信息，请参见“CREATE PROCEDURE 语句（Web 服务）”一节 《SQL Anywhere 服务器 - SQL 参考》。

3. 使用 CALL 语句调用 main 方法。

   ```
   call JavaMain( 'Hello world' );
   ```

   由于 SQL 语言的局限性，只能传递单个字符串。

在 Java 应用程序中使用线程

利用 java.lang.Thread 包的功能，可以在 Java 应用程序中使用多个线程。

您可在 Java 应用程序中同步、挂起、重新开始、中断或停止线程。

无此类方法例外

如果在调用 Java 方法时所提供的参数数量不正确，或者所使用的数据类型不正确，Java VM 将用一个 `java.lang.NoSuchMethodException` 错误做出响应。您应检查参数的数量和类型。

有关详细信息，请参见“访问 Java 对象的字段和方法”一节第 90 页。

从 Java 方法返回结果集

本节介绍如何从 Java 方法获得结果集。您必须编写一个向调用环境返回结果集的 Java 方法，并将此方法包装在一个被声明为 LANGUAGE JAVA 的 EXTERNAL NAME 的 SQL 存储过程中。

♦ 从一个 Java 方法返回结果集

1. 确保在一个公共类中将 Java 方法声明为公共的和静态的。
2. 对于您期望该方法返回的每个结果集，要确保该方法有一个类型为 java.sql.ResultSet[] 的参数。这些结果集参数都必须出现在参数列表的末尾处。
3. 在该方法中，首先创建一个 java.sql.ResultSet 实例，然后将其指派给其中一个 ResultSet[] 参数。
4. 创建一个类型为 EXTERNAL NAME LANGUAGE JAVA 的 SQL 存储过程。该类型的过程是 Java 方法的包装。您可以像对其它任何返回结果集的过程一样对 SQL 过程结果集使用游标。

   有关充当 Java 方法包装的存储过程的语法的详细信息，请参见“CREATE PROCEDURE 语句（Web 服务）”一节 《SQL Anywhere 服务器 - SQL 参考》。

示例

下面的简单类有一个方法，该方法执行查询并将结果集返回给调用环境。

```
import java.sql.*;

public class MyResultSet
{
  public static void return_rset( ResultSet[] rset1 )
       throws SQLException
  {
    Connection conn = DriverManager.getConnection(
                      "jdbc:default:connection" );
    Statement stmt = conn.createStatement();
    ResultSet rset =
      stmt.executeQuery (
                      "SELECT Surname " +
                      "FROM Customers" );
    rset1[0] = rset;
  }
}
```

您可以使用 CREATE PROCEDURE 语句公开结果集，该语句将指明从过程返回的结果集的数量以及 Java 方法的签名。

指明结果集的 CREATE PROCEDURE 语句可以定义如下：

```
CREATE PROCEDURE result_set()
  DYNAMIC RESULT SETS 1
  EXTERNAL NAME
     'MyResultSet.return_rset([Ljava/sql/ResultSet;)V'
  LANGUAGE JAVA
```

您可以对此过程打开游标，就像对任何返回结果集的 SQL Anywhere 过程一样。

字符串 `([Ljava/sql/ResultSet;)V` 是一个 Java 方法签名，它是参数和返回值的数量及类型的精简字符表示形式。

有关 Java 方法签名的详细信息，请参见“CREATE PROCEDURE 语句（Web 服务）”一节《SQL Anywhere 服务器 - SQL 参考》。

有关返回结果集的详细信息，请参见“返回结果集”一节第 503 页。

通过存储过程从 Java 返回值

您可以将使用 EXTERNAL NAME LANGUAGE JAVA 创建的存储过程用作 Java 方法的包装。本节介绍如何编写 Java 方法以在存储过程中利用 OUT 或 INOUT 参数。

Java 不显式支持 INOUT 或 OUT 参数。相反，您可以使用参数数组。例如，要使用整型 OUT 参数，请创建一个刚好由一个整数组成的数组：

```
public class Invoice
{
  public static boolean testOut( int[] param )
  {
    param[0] = 123;
    return true;
  }
}
```

以下过程使用 testOut 方法：

```
CREATE PROCEDURE testOut( OUT p INTEGER )
EXTERNAL NAME 'Invoice.testOut([I)Z'
LANGUAGE JAVA;
```

字符串 `([I)Z` 是一个 Java 方法签名，用于指明该方法有一个单个参数（是一个由整数构成的数组）并会返回一个布尔值。您必须相应定义该方法，从而使要用作 OUT 或 INOUT 参数的方法参数是一个对应于 OUT 或 INOUT 参数的 SQL 数据类型的 Java 数据类型的数组。

要对此进行测试，请使用未初始化的变量调用该存储过程。

```
CREATE VARIABLE zap INTEGER;
CALL testOut( zap );
SELECT zap;
```

结果集是 `123`。

有关语法（包括方法签名）的详细信息，请参见“CREATE PROCEDURE 语句（Web 服务）”一节 《SQL Anywhere 服务器 - SQL 参考》。

Java 的安全管理

Java 提供了安全管理器，您可以使用它们控制用户对应用程序的安全敏感功能（如文件访问和网络访问）的访问。您应利用 Java VM 所支持的安全管理功能。

启动和停止 Java VM

Java VM 将在执行第一个 Java 操作时自动装载。如果想要显式装载它以便为执行 Java 操作做好准备，则可以通过执行下面的语句来完成此任务：

```
START JAVA;
```

可在不使用 Java 时使用 STOP JAVA 语句卸载 Java VM。只有具有 DBA 权限的用户才能执行该语句。语法是：

```
STOP JAVA;
```

SQL Anywhere 数据访问 API

本节介绍 SQL Anywhere 的编程接口。

SQL Anywhere .NET 数据提供程序

目录

SQL Anywhere .NET 数据提供程序功能

通过三个不同的命名空间，SQL Anywhere 支持 Microsoft .NET Framework 2.0 或更高版本。

- **iAnywhere.Data.SQLAnywhere** ADO.NET 对象模型是通用数据访问模型。ADO.NET 组件设计用于代管数据操作中的数据访问。有两个可完成此任务的 ADO.NET 中央组件：DataSet 和 .NET Framework 数据提供程序。.NET Framework 数据提供程序是一组组件，其中包括 Connection、Command、DataReader 和 DataAdapter 对象。SQL Anywhere 包括一个 .NET Framework 数据提供程序，该数据提供程序可与 SQL Anywhere 数据库服务器直接通信，而不会增加 OLE DB 或 ODBC 开销。在 .NET 命名空间中，SQL Anywhere .NET 数据提供程序表示为 iAnywhere.Data.SQLAnywhere。

  Microsoft .NET Compact Framework 是适用于 Microsoft .NET 的智能设备开发框架。SQL Anywhere .NET Compact Framework 数据提供程序支持运行 Windows Mobile 的设备。

  SQL Anywhere .NET 数据提供程序命名空间在本文档中介绍。

- **System.Data.Oledb** 此命名空间支持 OLE DB 数据源。此命名空间是 Microsoft .NET Framework 的固有部分。System.Data.Oledb 可与 SQL Anywhere OLE DB 提供程序 SAOLEDB 一起使用，以访问 SQL Anywhere 数据库。

- **System.Data.Odbc** 此命名空间支持 ODBC 数据源。此命名空间是 Microsoft .NET Framework 的固有部分。System.Data.Odbc 可与 SQL Anywhere ODBC 驱动程序一起使用，以访问 SQL Anywhere 数据库。

在 Windows Mobile 上，仅支持 SQL Anywhere .NET 数据提供程序。

使用 SQL Anywhere .NET 数据提供程序具有几大优点：

- 在 .NET 环境中，SQL Anywhere .NET 数据提供程序提供对 SQL Anywhere 的本地访问。与其它受支持的提供程序不同，它直接与 SQL Anywhere 服务器进行通信而不需要使用 Bridge 技术。
- 因此，SQL Anywhere .NET 数据提供程序比 OLE DB 和 ODBC 数据提供程序速度更快。建议使用此数据提供程序访问 SQL Anywhere 数据库。

运行示例项目

SQL Anywhere .NET 数据提供程序中包含四个示例项目：

- **SimpleCE** 一个用于 Windows Mobile 的 .NET Compact Framework 示例项目，该项目演示了一个简单的列表框，当单击 [**Connect**] 时，会用 Employees 表中的姓名填充该列表框。
- **SimpleWin32** 一个用于 Windows 的 .NET Framework 示例项目，该项目演示了一个简单的列表框，当单击 [**Connect**] 时，会用 Employees 表中的姓名填充该列表框。
- **SimpleXML** 适用于 Windows 的 .NET Framework 示例项目，演示如何通过 ADO.NET 从 SQL Anywhere 获得 XML 数据。
- **TableViewer** 适用于 Windows 的 .NET Framework 示例项目，允许用户输入和执行 SQL 语句。

有关解释示例项目的教程，请参见“教程：使用 SQL Anywhere .NET 数据提供程序”第 137 页。

注意

如果未将 SQL Anywhere 安装在缺省安装目录 (*C:\Program Files\SQL Anywhere 11*) 中，则若在装载示例项目时引用数据提供程序 DLL，会出现错误。如果发生这种情况，请添加一个对 *iAnywhere.Data.SQLAnywhere.dll* 的引用。有一个版本的数据提供程序支持 .NET Framework 2.0 及更高版本。适用于 Windows 的数据提供程序位于 *install-dir\Assembly\v2\iAnywhere.Data.SQLAnywhere.dll* 中。适用于 Windows Mobile 的数据提供程序位于 *install-dir\ce\Assembly* 中。

有关添加对 DLL 的引用的说明，请参见“在项目中添加对数据提供程序 DLL 的引用”一节第 106 页。

在 Visual Studio 项目中使用 .NET 数据提供程序

SQL Anywhere .NET 数据提供程序可与 Visual Studio 2005 或更高版本搭配使用开发应用程序。若要使用 SQL Anywhere .NET 数据提供程序，Visual Studio 项目中必须包括以下两项：

- 对 SQL Anywhere .NET 数据提供程序 DLL 的引用
- 源代码中引用 SQL Anywhere .NET 数据提供程序类的一行语句

下面详细说明这些步骤。

有关安装和注册 SQL Anywhere .NET 数据提供程序的信息，请参见“部署 SQL Anywhere .NET 数据提供程序”一节第 132 页。

在项目中添加对数据提供程序 DLL 的引用

添加引用可通知 Visual Studio 要包含哪些 DLL 以找到 SQL Anywhere .NET 数据提供程序的代码。

♦ 在 Visual Studio 项目中添加对 SQL Anywhere .NET 数据提供程序的引用

1. 启动 Visual Studio 并打开项目。
2. 在 [**Solution Explorer**] 窗口中，右击 [**References**]，然后选择 [**Add Reference**]。
3. 在 [**.NET**] 选项卡上，单击 [**Browse**] 来查找 *iAnywhere.Data.SQLAnywhere.dll*。请注意，对于每个 Windows 和 Windows Mobile 平台，都提供单独的 DLL 版本。
   - 对于 Windows SQL Anywhere .NET 数据提供程序，缺省位置是 *install-dir\Assembly\v2*。
   - 对于 Windows Mobile SQL Anywhere .NET 数据提供程序，缺省位置是 *install-dir\ce\Assembly\v2*。
4. 选择 DLL，然后单击 [**Open**]。

   有关已安装的 DLL 的完整列表，请参见“SQL Anywhere .NET 数据提供程序必需的文件”一节第 132 页。
5. 可以验证 DLL 是否已添加到项目中。打开 [**Add Reference**] 窗口然后单击 [**.NET**] 选项卡。随即 *iAnywhere.Data.SQLAnywhere.dll* 出现在 [**Selected Components**] 列表中。单击 [**OK**] 关闭窗口。

   DLL 将添加到项目的 [**Solution Explorer**] 窗口的 [**References**] 文件夹中。

在源代码中使用数据提供程序类

为帮助使用 SQL Anywhere .NET 数据提供程序命名空间以及在此命名空间中定义的类型，应在源代码中添加一条指令。

♦ 在代码中使用数据提供程序命名空间

1. 启动 Visual Studio 并打开项目。
2. 在项目添加下面的指令行：

- 如果使用的是 C#，请将以下行添加到项目开始处的 `using` 指令列表中：

  ```
  using iAnywhere.Data.SQLAnywhere;
  ```

- 如果使用的是 Visual Basic，则应在项目开头处 `Public Class Form1` 行之前添加以下行：

  ```
  Imports iAnywhere.Data.SQLAnywhere
  ```

此指令虽然不是必需的指令，但通过它可使用 SQL Anywhere .NET 类的简写形式。例如：

```
SAConnection conn = new SAConnection()
```

在没有此指令的情况下，您仍然可以使用下面的语句：

```
iAnywhere.Data.SQLAnywhere.SAConnection
 conn = new iAnywhere.Data.SQLAnywhere.SAConnection()
```

连接到数据库

对数据执行任何操作之前，您的应用程序必须先连接到数据库。本节介绍如何通过编写代码连接到 SQL Anywhere 数据库。

有关详细信息，请参见“SAConnectionStringBuilder 类”一节第 257 页和“ConnectionName 属性”一节第 266 页。

♦ 连接到 SQL Anywhere 数据库

1. 分配 SAConnection 对象。

   以下代码创建一个名为 conn 的 SAConnection 对象：

   ```
   SAConnection conn = new SAConnection(connection-string)
   ```

   应用程序与数据库之间可以建立多个连接。某些应用程序只与 SQL Anywhere 数据库建立一个连接，并且该连接始终保持打开状态。为此，可以为连接声明一个全局变量：

   ```
   private SAConnection  _conn;
   ```

   有关详细信息，请参见 *samples-dir\SQLAnywhere\ADO.NET\TableViewer* 和“了解 Table Viewer 示例项目”一节第 143 页中的示例代码。

2. 指定用于连接到数据库的连接字符串。

   例如：

   ```
   "Data Source=SQL Anywhere 11 Demo"
   ```

   有关连接参数的完整列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

   可以不提供连接字符串，而是提示用户输入他们的用户 ID 和口令。

3. 打开一个与数据库的连接。

   以下代码尝试连接到数据库。它会在需要时自动启动数据库服务器。

   ```
   conn.Open();
   ```

4. 捕获连接错误。

   您的应用程序应设计为捕获尝试连接数据库时出现的任何错误。以下代码演示如何捕获错误并显示其消息：

   ```
   try {
       _conn = new SAConnection( txtConnectString.Text );
       _conn.Open();
     } catch( SAException ex ) {
       MessageBox.Show( ex.Errors[0].Source + " : "
        + ex.Errors[0].Message + " (" +
        ex.Errors[0].NativeError.ToString() + ")",
            "Failed to connect" );
   ```

   或者，可以使用 ConnectionString 属性设置连接字符串，而不是在创建 SAConnection 对象时传递连接字符串：


```
SAConnection _conn;
    _conn = new SAConnection();
    _conn.ConnectionString =
      "Data Source=SQL Anywhere 11 Demo";
    _conn.Open();
```

5. 关闭与数据库的连接。与数据库的连接会一直保持打开状态，直到使用 conn.Close() 方法将它们显式关闭。

Visual Basic 连接示例

以下 Visual Basic 代码打开一个与 SQL Anywhere 示例数据库的连接：

```
Private Sub Button1_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles Button1.Click
    ' Declare the connection object
    Dim myConn As New _
      iAnywhere.Data.SQLAnywhere.SAConnection()
    myConn.ConnectionString = _
      "Data Source=SQL Anywhere 11 Demo"
    myConn.Open()
    myConn.Close()
End Sub
```

连接池

SQL Anywhere .NET 数据提供程序支持连接池。连接池使应用程序可以通过将连接句柄保存到池中来重复使用现有连接，而不用重复新建与数据库的连接。缺省情况下，连接池是打开的。

池容量可使用 POOLING 选项在连接字符串中设置。最大池容量的缺省值是 100，最小池容量的缺省值是 0。可指定最小池容量和最大池容量。例如：

```
"Data Source=SQL Anywhere 11 Demo;POOLING=TRUE;Max Pool Size=50;Min Pool
Size=5"
```

当应用程序第一次尝试连接数据库时，它将在池中检查是否存在使用您所指定的连接参数的现有连接。如果找到匹配的连接，将使用该连接。否则将使用新的连接。断开连接时，连接将返回池中，以便可以重复使用该连接。

另请参见

- “ConnectionName 属性”一节第 266 页
- “AutoStop 连接参数 [ASTOP]”一节 《SQL Anywhere 服务器 - 数据库管理》

检查连接状态

一旦应用程序与数据库建立了连接，您就可以检查连接状态，确保在从数据库读取数据以进行更新之前连接处于打开状态。如果连接丢失或被占用，或者正在处理另一语句，可以将适当的消息返回给用户。

SAConnection 类具有一个可用于检查连接状态的状态属性。可能的状态值为 Open 和 Closed。

以下代码检查 Connection 对象是否已初始化，如果已初始化，则确保连接处于打开状态。如果连接没有打开，则会为用户返回一条消息。

```
if( _conn == null || _conn.State !=
  ConnectionState.Open ) {
  MessageBox.Show( "Connect to a database first",
   "Not connected" );
  return;
```

有关详细信息，请参见“State 属性”一节第 241 页。

访问和操作数据

如果使用 SQL Anywhere .NET 数据提供程序，则有两种方法访问数据：

- **SACommand 对象** 建议使用 SACommand 对象在 .NET 中访问和操作数据。

  SACommand 对象允许您执行直接从数据库检索或修改数据的 SQL 语句。使用 SACommand 对象可直接对数据库发出 SQL 语句以及调用存储过程。

  在 SACommand 对象中，SADataReader 用于从查询或存储过程返回只读结果集。SADataReader 每次仅返回一行，但这并不会降低性能，因为 SQL Anywhere 客户端的库使用预取缓冲每次预取多行。

  使用 SACommand 对象可以将更改组合成事务，而不是在自动提交模式下操作。使用 SATransaction 对象时，会将行锁定，这样其他用户便无法对其进行修改。

  有关详细信息，请参见“SACommand 类”一节第 195 页和“SADataReader 类”一节第 297 页。

- **SADataAdapter 对象** SADataAdapter 对象会将整个结果集检索到一个 DataSet 中。DataSet 是用于保存从数据库检索到的数据的断开连接的存储区。之后可以编辑 DataSet 中的数据，编辑完成后，SADataAdapter 对象利用对 DataSet 所做的更改更新数据库。使用 SADataAdapter 时，无法阻止其他用户修改 DataSet 中的行。您需要在应用程序中包含用于解决可能出现的任何冲突的逻辑。

  有关冲突的详细信息，请参见“解决使用 SADataAdapter 时的冲突”一节第 118 页。

  有关 SADataAdapter 对象的详细信息，请参见“SADataAdapter 类”一节第 286 页。

在 SACommand 对象中使用 SADataReader 而不使用 SADataAdapter 对象从数据库读取行时，对性能没有影响。

使用 SACommand 对象检索和操作数据

以下各节介绍如何使用 SADataReader 检索数据以及插入、更新或删除行。

使用 SACommand 对象获取数据

使用 SACommand 对象可针对 SQL Anywhere 数据库执行 SQL 语句或调用存储过程。可以使用下列方法中的任一种检索数据库中的数据：

- **ExecuteReader** 发出返回结果集的 SQL 查询。此方法使用只进、只读游标。可以沿一个方向快速循环遍历结果集中的行。

  有关详细信息，请参见“ExecuteReader 方法”一节第 214 页。

- **ExecuteScalar** 发出返回单个值的 SQL 查询。可以是结果集的第一行中的第一列，或返回集合值（如 COUNT 或 AVG）的 SQL 语句。此方法使用只进、只读游标。

有关详细信息，请参见“ExecuteScalar 方法”一节第 216 页。

使用 SACommand 对象时，可使用 SADataReader 检索基于连接的结果集。但是，只能对一个表中的数据进行更改（插入、更新或删除）。不能更新基于连接的数据集。

下面的说明使用 .NET 数据提供程序提供的 Simple 代码示例。

有关 Simple 代码示例的详细信息，请参见“了解 Simple 示例项目”一节第 140 页。

♦ 发出返回完整结果集的 SQL 查询

1. 声明并初始化一个 Connection 对象。

```
SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 11 Demo" );
```

2. 打开该连接。

```
try {
    conn.Open();
```

3. 添加一个 Command 对象以定义并执行一条 SQL 语句。

```
SACommand cmd = new SACommand(
     "SELECT Surname FROM Employees", conn );
```

   如果要调用存储过程，必须为该存储过程指定参数。

   有关详细信息，请参见“使用存储过程”一节第 127 页和“SAParameter 类”一节第 367 页。

4. 调用 ExecuteReader 方法以返回 DataReader 对象。

```
SADataReader reader = cmd.ExecuteReader();
```

5. 显示结果。

```
listEmployees.BeginUpdate();
while( reader.Read() ) {
     listEmployees.Items.Add( reader.GetString( 0 ) );
}
listEmployees.EndUpdate();
```

6. 关闭 DataReader 和 Connection 对象。

```
reader.Close();
conn.Close();
```

♦ 发出仅返回一个值的 SQL 查询

1. 声明并初始化一个 SAConnection 对象。

```
SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 11 Demo" );
```

2. 打开该连接。

```
conn.Open();
```


3. 添加一个 SACommand 对象，以定义并执行一条 SQL 语句。

```
SACommand cmd = new SACommand(
    "SELECT COUNT(*) FROM Employees WHERE Sex = 'M'",
    conn );
```

如果要调用存储过程，必须为该存储过程指定参数。

有关详细信息，请参见“使用存储过程”一节第 127 页。

4. 调用 ExecuteScalar 方法返回包含该值的对象。

```
int count = (int) cmd.ExecuteScalar();
```

5. 关闭 SAConnection 对象。

```
conn.Close();
```

使用 SADataReader 时，可使用多种 Get 方法来返回指定数据类型的结果。

有关详细信息，请参见“SADataReader 类”一节第 297 页。

Visual Basic DataReader 示例

下面的 Visual Basic 代码打开一个与 SQL Anywhere 示例数据库的连接，然后使用 DataReader 返回结果集中前五位雇员的姓氏：

```
Dim myConn As New SAConnection()
Dim myCmd As _
  New SACommand _
  ("SELECT Surname FROM Employees", myConn)
Dim myReader As SADataReader
Dim counter As Integer
myConn.ConnectionString = _
  "Data Source=SQL Anywhere 11 Demo"
myConn.Open()
myReader = myCmd.ExecuteReader()
counter = 0
Do While (myReader.Read())
  MsgBox(myReader.GetString(0))
  counter = counter + 1
  If counter >= 5 Then Exit Do
Loop
myConn.Close()
```

使用 SACommand 对象插入、更新和删除行

若要使用 SACommand 对象执行插入、更新或删除，请使用 ExecuteNonQuery 函数。ExecuteNonQuery 函数发出一个不返回结果集的查询（SQL 语句或存储过程）。请参见“ExecuteNonQuery 方法”一节第 213 页。

只能对一个表中的数据进行更改（插入、更新或删除）。不能更新基于连接的数据集。必须连接到数据库才能使用 SACommand 对象。

有关获取自动增量主键的主键值的信息，请参见“获取主键值”一节第 122 页。

如果希望设置 SQL 语句的隔离级别，则必须将 SACommand 对象用作 SATransaction 对象的一部分。不使用 SATransaction 对象修改数据时，.NET 数据提供程序在自动提交模式下运行，您所做的任何更改都将立即得到应用。请参见“事务处理”一节第 129 页。

♦ 发出插入行的语句

1. 声明并初始化一个 SAConnection 对象。

```
SAConnection conn = new SAConnection(
    c_connStr );
conn.Open();
```

2. 打开该连接。

```
conn.Open();
```

3. 添加一个 SACommand 对象，以定义并执行一条 INSERT 语句。

   可将 INSERT、UPDATE 或 DELETE 语句与 ExecuteNonQuery 方法一起使用。

```
SACommand insertCmd = new SACommand(
    "INSERT INTO Departments( DepartmentID, DepartmentName )
    VALUES( ?, ? )", conn );
```

   如果要调用存储过程，必须为该存储过程指定参数。

   有关详细信息，请参见“使用存储过程”一节第 127 页和“SAParameter 类”一节第 367 页。

4. 设置 SACommand 对象的参数。

   以下代码分别为 DepartmentID 和 DepartmentName 列定义参数。

```
SAParameter parm = new SAParameter();
parm.SADbType = SADbType.Integer;
insertCmd.Parameters.Add( parm );
parm = new SAParameter();
parm.SADbType = SADbType.Char;
insertCmd.Parameters.Add( parm );
```

5. 插入新值并调用 ExecuteNonQuery 方法，将更改应用到数据库。

```
insertCmd.Parameters[0].Value = 600;
insertCmd.Parameters[1].Value = "Eastern Sales";
int recordsAffected = insertCmd.ExecuteNonQuery();
insertCmd.Parameters[0].Value = 700;
insertCmd.Parameters[1].Value = "Western Sales";
recordsAffected = insertCmd.ExecuteNonQuery();
```

6. 显示结果并将结果绑定到屏幕上的网格中。

```
SACommand selectCmd = new SACommand(
    "SELECT * FROM Departments", conn );
SADataReader dr = selectCmd.ExecuteReader();

System.Windows.Forms.DataGrid dataGrid;
dataGrid = new System.Windows.Forms.DataGrid();
dataGrid.Location = new Point(10, 10);
dataGrid.Size = new Size(275, 200);
dataGrid.CaptionText = "iAnywhere SACommand Example";
this.Controls.Add(dataGrid);
```



```
dataGrid.DataSource = dr;
dataGrid.Show();
```

7. 关闭 SADataReader 和 SAConnection 对象。

```
dr.Close();
conn.Close();
```

♦ 发出更新行的语句

1. 声明并初始化一个 SAConnection 对象。

```
SAConnection conn = new SAConnection(
    c_connStr );
```

2. 打开该连接。

```
conn.Open();
```

3. 添加一个 SACommand 对象，以定义并执行一条 UPDATE 语句。

   可将 INSERT、UPDATE 或 DELETE 语句与 ExecuteNonQuery 方法一起使用。

```
SACommand updateCmd = new SACommand(
    "UPDATE Departments SET DepartmentName = 'Engineering'
    WHERE DepartmentID=100", conn );
```

   如果要调用存储过程，必须为该存储过程指定参数。

   有关详细信息，请参见“使用存储过程”一节第 127 页和“SAParameter 类”一节第 367 页。

4. 调用 ExecuteNonQuery 方法，将更改应用到数据库。

```
int recordsAffected = updateCmd.ExecuteNonQuery();
```

5. 显示结果并将结果绑定到屏幕上的网格中。

```
SACommand selectCmd = new SACommand(
    "SELECT * FROM Departments", conn );
SADataReader   dr = selectCmd.ExecuteReader();
dataGrid.DataSource = dr;
```

6. 关闭 SADataReader 和 SAConnection 对象。

```
dr.Close();
conn.Close();
```

♦ 发出删除行的语句

1. 声明并初始化一个 SAConnection 对象。

```
SAConnection conn = new SAConnection(
    c_connStr );
```

2. 打开该连接。

```
conn.Open();
```

3. 创建一个 SACommand 对象，以定义并执行一条 DELETE 语句。

   可将 INSERT、UPDATE 或 DELETE 语句与 ExecuteNonQuery 方法一起使用。

```
SACommand deleteCmd = new SACommand(
    "DELETE FROM Departments WHERE ( DepartmentID > 500 )", conn );
```

如果要调用存储过程，必须为该存储过程指定参数。

有关详细信息，请参见“使用存储过程”一节第 127 页和“SAParameter 类”一节第 367 页。

4. 调用 ExecuteNonQuery 方法，将更改应用到数据库。

```
int recordsAffected = deleteCmd.ExecuteNonQuery();
```

5. 关闭 SAConnection 对象。

```
conn.Close();
```

获取 DataReader 模式信息

您可以获得结果集中列的模式信息。

如果使用 SADataReader，则可以使用 GetSchemaTable 方法获取有关结果集的信息。GetSchemaTable 方法返回标准 .NET DataTable 对象，该对象提供有关结果集中所有列的信息，包括列属性。

有关 GetSchemaTable 方法的详细信息，请参见“GetSchemaTable 方法”一节第 317 页。

♦ **使用 GetSchemaTable 方法获取有关结果集的信息。**

1. 声明并初始化一个 Connection 对象。

```
SAConnection conn = new SAConnection(
    c_connStr );
```

2. 打开该连接。

```
conn.Open();
```

3. 用您要使用的 SELECT 语句来创建一个 SACommand 对象。将返回此查询的结果集的模式。

```
SACommand cmd = new SACommand(
    "SELECT * FROM Employees", conn );
```

4. 创建一个 SADataReader 对象，然后执行创建的 Command 对象。

```
SADataReader   dr = cmd.ExecuteReader();
```

5. 使用数据源模式填充 DataTable。

```
DataTable       schema = dr.GetSchemaTable();
```

6. 关闭 SADataReader 和 SAConnection 对象。

```
dr.Close();
conn.Close();
```

7. 将 DataTable 绑定到屏幕上的网格。

```
dataGrid.DataSource = schema;
```


使用 SADataAdapter 对象访问和操作数据

以下各节介绍如何使用 SADataAdapter 检索数据以及插入、更新或删除行。

使用 SADataAdapter 对象获取数据

使用 SADataAdapter 可查看整个结果集，方法是将一个 DataSet 绑定到显示网格，使用 Fill 方法将查询结果填充到此 DataSet 中。

使用 SADataAdapter 可以传递任何返回结果集的字符串（SQL 语句或存储过程）。使用 SADataAdapter 时，所有行都是使用只进、只读游标在一次操作中读取的。读取结果中的所有行后，将关闭游标。使用 SADataAdapter 可对 DataSet 进行更改。完成更改后，必须重新连接到数据库才能应用更改。

可以使用 SADataAdapter 对象检索基于连接的结果集。但是，只能对一个表中的数据进行更改（插入、更新或删除）。不能更新基于连接的结果集。

小心

对 DataSet 所做的所有更改都是在断开与数据库的连接的情况下完成的。这意味着应用程序未锁定数据库中的这些行。如果在您的更改应用到数据库之前其他用户更改了您正修改的数据，则在将 DataSet 的更改应用到数据库时，会出现一些冲突。您的应用程序必须设计为能够解决这样的冲突。

有关 SADataAdapter 的详细信息，请参见“SADataAdapter 类”一节第 286 页。

SADataAdapter 示例

下面的示例显示如何使用 SADataAdapter 填充 DataSet。

- **使用 SADataAdapter 对象检索数据**

1. 连接到数据库。
2. 创建一个新 DataSet。在本例中，DataSet 名为 Results。

```
DataSet ds =new DataSet ();
```

3. 创建一个新的 SADataAdapter 对象，以执行 SQL 语句并填充 DataSet。

```
SADataAdapter da=new SADataAdapter(
     txtSQLStatement.Text, _conn);
da.Fill(ds, "Results")
```

4. 将 DataSet 绑定到屏幕上的网格。

```
dgResults.DataSource = ds.Tables["Results"]
```

使用 SADataAdapter 对象插入、更新和删除行

SADataAdapter 会将结果集检索到一个 DataSet 中。DataSet 是表以及这些表之间的关系和约束的集合。DataSet 内置在 .NET Framework 中，与用于连接数据库的数据提供程序无关。

使用 SADataAdapter 时，必须连接到数据库以填充 DataSet 并使用对 DataSet 的更改更新数据库。不过，填充 DataSet 后，可以在 DataSet 与数据库断开的情况下对其进行修改。

如果不希望立即将更改应用到数据库，可以使用 WriteXML 方法将 DataSet（包括数据和/或模式）写入 XML 文件。这样，以后就可以通过使用 ReadXML 方法装载 DataSet 来应用更改。

有关详细信息，请参见 .NET Framework 文档中 WriteXML 和 ReadXML 的部分。

调用 Update 方法将 DataSet 的更改应用于数据库时，SADataAdapter 会对已经做出的更改进行分析，然后根据需要调用相应的 INSERT、UPDATE 或 DELETE 语句。使用 DataSet 时，只能对一个表中的数据进行更改（插入、更新或删除）。不能更新基于连接的数据集。如果其他用户锁定了您试图更新的行，则将抛出异常。

小心
对 DataSet 所做的所有更改都是在断开连接的情况下完成的。这意味着应用程序未锁定数据库中的这些行。如果在您的更改应用到数据库之前其他用户更改了您正修改的数据，则在将 DataSet 的更改应用到数据库时，会出现一些冲突。您的应用程序必须设计为能够解决这样的冲突。

解决使用 SADataAdapter 时的冲突

使用 SADataAdapter 时，不会锁定数据库的行。这意味着将来自 DataSet 的更改应用到数据库时可能会引起冲突。应用程序中应包含解决或记录产生的冲突的逻辑。

应用程序逻辑应解决的一些冲突如下：

- **唯一主键** 如果两个用户向一个表中插入新行，则每个行都必须有一个唯一的主键。对于包含自动增量主键的表，DataSet 中的值可能与数据源中的值不同步。

  有关获取自动增量主键的主键值的信息，请参见“获取主键值”一节第 122 页。
- **对同一值的更新** 如果两个用户修改同一值，则应用程序应包含用于决定哪个值正确的逻辑。
- **模式更改** 如果某个用户修改已在 DataSet 中更新的表的模式，则在将更改应用于数据库时，更新会失败。
- **数据并发** 并发应用程序应看到一致的数据集。SADataAdapter 不会锁定其读取的行，因此在您检索了 DataSet 并且脱机工作时，其他用户可以更新数据库中的值。

许多这样的潜在问题都可通过使用 SACommand、SADataReader 和 SATransaction 对象将更改应用到数据库来避免。建议使用 SATransaction 对象，因为它允许您设置事务的隔离级别，并将行锁定，这样其他用户便无法修改这些行。

有关使用事务将更改应用于数据库的详细信息，请参见“使用 SACommand 对象插入、更新和删除行”一节第 113 页。

若要简化冲突解决过程，可以自行设计 INSERT、UPDATE 或 DELETE 语句作为存储过程调用。通过在存储过程中包含 INSERT、UPDATE 和 DELETE 语句，可以捕获操作失败时的错误。除了语句外，还可以在存储过程中添加错误处理逻辑，以便在操作失败时采取适当行动，例如将错误记录到日志文件，或再次尝试操作。

♦ 使用 SADataAdapter 在表中插入行

1. 声明并初始化一个 SAConnection 对象。


```
SAConnection   conn = new SAConnection(
    c_connStr );
```

2. 打开该连接。

```
conn.Open();
```

3. 创建一个新的 SADataAdapter 对象。

```
SADataAdapter adapter = new SADataAdapter();
adapter.MissingMappingAction =
    MissingMappingAction.Passthrough;
adapter.MissingSchemaAction =
    MissingSchemaAction.Add;
```

4. 创建必需的 SACommand 对象并定义所有必需的参数。

   以下代码创建一条 SELECT 和一条 INSERT 语句，并定义 INSERT 语句的参数。

```
adapter.SelectCommand = new SACommand(
    "SELECT * FROM Departments", conn );
adapter.InsertCommand = new SACommand(
    "INSERT INTO Departments( DepartmentID, DepartmentName )
    VALUES( ?, ? )", conn );
adapter.InsertCommand.UpdatedRowSource =
    UpdateRowSource.None;
SAParameter parm = new SAParameter();
parm.SADbType = SADbType.Integer;
parm.SourceColumn = "DepartmentID";
parm.SourceVersion = DataRowVersion.Current;
adapter.InsertCommand.Parameters.Add(
    parm );
parm = new SAParameter();
parm.SADbType = SADbType.Char;
parm.SourceColumn = "DepartmentName";
parm.SourceVersion = DataRowVersion.Current;
adapter.InsertCommand.Parameters.Add( parm );
```

5. 用 SELECT 语句的结果填充 DataTable。

```
DataTable dataTable = new DataTable( "Departments" );
int rowCount = adapter.Fill( dataTable );
```

6. 将新行插入到 DataTable 并将更改应用于数据库。

```
DataRow row1 = dataTable.NewRow();
row1[0] = 600;
row1[1] = "Eastern Sales";
dataTable.Rows.Add( row1 );
DataRow row2 = dataTable.NewRow();
row2[0] = 700;
row2[1] = "Western Sales";
dataTable.Rows.Add( row2 );
recordsAffected = adapter.Update( dataTable );
```

7. 显示更新的结果。

```
dataTable.Clear();
rowCount = adapter.Fill( dataTable );
dataGrid.DataSource = dataTable;
```

8. 关闭该连接。

```
conn.Close();
```

♦ 使用 SADataAdapter 对象更新行

1. 声明并初始化一个 SAConnection 对象。

   ```
   SAConnection conn = new SAConnection( c_connStr );
   ```

2. 打开该连接。

   ```
   conn.Open();
   ```

3. 创建一个新的 SADataAdapter 对象。

   ```
   SADataAdapter adapter = new SADataAdapter();
   adapter.MissingMappingAction =
       MissingMappingAction.Passthrough;
   adapter.MissingSchemaAction =
       MissingSchemaAction.Add;
   ```

4. 创建 SACommand 对象并定义其参数。

   以下代码创建一条 SELECT 和一条 UPDATE 语句，并定义 UPDATE 语句的参数。

   ```
   adapter.SelectCommand = new SACommand(
       "SELECT * FROM Departments WHERE DepartmentID > 500",
       conn );
   adapter.UpdateCommand = new SACommand(
       "UPDATE Departments SET DepartmentName = ?
       WHERE DepartmentID = ?", conn );
   adapter.UpdateCommand.UpdatedRowSource =
       UpdateRowSource.None;
   SAParameter parm = new SAParameter();
   parm.SADbType = SADbType.Char;
   parm.SourceColumn = "DepartmentName";
   parm.SourceVersion = DataRowVersion.Current;
   adapter.UpdateCommand.Parameters.Add( parm );
   parm = new SAParameter();
   parm.SADbType = SADbType.Integer;
   parm.SourceColumn = "DepartmentID";
   parm.SourceVersion = DataRowVersion.Original;
   adapter.UpdateCommand.Parameters.Add( parm );
   ```

5. 用 SELECT 语句的结果填充 DataTable。

   ```
   DataTable dataTable = new DataTable( "Departments" );
   int rowCount = adapter.Fill( dataTable );
   ```

6. 使用行的更新值更新 DataTable，并将更改应用到数据库。

   ```
   foreach ( DataRow row in dataTable.Rows )
   {
   row[1] = ( string ) row[1] + "_Updated";
   }
   recordsAffected = adapter.Update( dataTable );
   ```

7. 将结果绑定到屏幕上的网格。

   ```
   dataTable.Clear();
   adapter.SelectCommand.CommandText =
       "SELECT * FROM Departments";
   ```



```
rowCount = adapter.Fill( dataTable );
dataGrid.DataSource = dataTable;
```

8. 关闭该连接。

```
conn.Close();
```

♦ 使用 SADataAdapter 对象删除表中的行

1. 声明并初始化一个 SAConnection 对象。

```
SAConnection conn = new SAConnection( c_connStr );
```

2. 打开该连接。

```
conn.Open();
```

3. 创建 SADataAdapter 对象。

```
SADataAdapter adapter = new SADataAdapter();
adapter.MissingMappingAction =
    MissingMappingAction.Passthrough;
adapter.MissingSchemaAction =
    MissingSchemaAction.AddWithKey;
```

4. 创建必需的 SACommand 对象并定义所有必需的参数。

   以下代码创建一条 SELECT 和一条 DELETE 语句，并定义 DELETE 语句的参数。

```
adapter.SelectCommand = new SACommand(
    "SELECT * FROM Departments WHERE DepartmentID > 500",
    conn );
adapter.DeleteCommand = new SACommand(
    "DELETE FROM Departments WHERE DepartmentID = ?",
    conn );
adapter.DeleteCommand.UpdatedRowSource =
    UpdateRowSource.None;
SAParameter parm = new SAParameter();
parm.SADbType = SADbType.Integer;
parm.SourceColumn = "DepartmentID";
parm.SourceVersion = DataRowVersion.Original;
adapter.DeleteCommand.Parameters.Add( parm );
```

5. 用 SELECT 语句的结果填充 DataTable。

```
DataTable dataTable = new DataTable( "Departments" );
int rowCount = adapter.Fill( dataTable );
```

6. 修改 DataTable 并将更改应用到数据库。

```
for each ( DataRow in dataTable.Rows )
{
    row.Delete();
}
recordsAffected = adapter.Update( dataTable )
```

7. 将结果绑定到屏幕上的网格。

```
dataTable.Clear();
rowCount = adapter.Fill( dataTable );
dataGrid.DataSource = dataTable;
```


8. 关闭该连接。

```
conn.Close();
```

获取 SADataAdapter 模式信息

使用 SADataAdapter 时，可以使用 FillSchema 方法获取有关 DataSet 中结果集的模式信息。FillSchema 方法返回标准 .NET DataTable 对象，该对象提供结果集中所有列的名称。

♦ 使用 FillSchema 方法获取 DataSet 模式信息

1. 声明并初始化一个 SAConnection 对象。

```
SAConnection   conn = new SAConnection(
    c_connStr );
```

2. 打开该连接。

```
conn.Open();
```

3. 用您要使用的 SELECT 语句来创建一个 SADataAdapter。将返回此查询的结果集的模式。

```
SADataAdapter  adapter = new SADataAdapter(
    "SELECT * FROM Employees", conn );
```

4. 创建一个新的 DataTable 对象（在本例中名为 Table），以使用模式进行填充。

```
DataTable       dataTable = new DataTable(
    "Table" );
```

5. 使用数据源模式填充 DataTable。

```
adapter.FillSchema( dataTable, SchemaType.Source );
```

6. 关闭 SAConnection 对象。

```
conn.Close();
```

7. 将 DataSet 绑定到屏幕上的网格。

```
dataGrid.DataSource = dataTable;
```

获取主键值

如果正在更新的表包含自动增量主键，请使用 UUID；或者，如果主键来自主键池，则可以使用存储过程获取该数据源生成的值。

使用 SADataAdapter 时，可通过此技术使用由数据源生成的主键值填充 DataSet 中的列。如果希望对 SACommand 对象使用此技术，则可以从参数获得主键列，也可以重新打开 DataReader。

示例

下面的示例使用名为 adodotnet_primarykey 的表，该表包含两列：ID 和 Name。该表的主键是 ID，其数据类型为 INTEGER 并包含一个自动增量值。Name 列的数据类型为 CHAR(40)。

这些示例调用以下存储过程从数据库中检索增量主键值。

```
CREATE PROCEDURE sp_adodotnet_primarykey( out p_id int, in p_name char(40) )
BEGIN
    INSERT INTO adodotnet_primarykey( name ) VALUES(
      p_name );
    SELECT @@IDENTITY INTO p_id;
END
```

♦ 使用 SACommand 对象插入一个包含自动增量主键的新行

1. 连接到数据库。

   ```
   SAConnection   conn = OpenConnection();
   ```

2. 创建一个新的 SACommand 对象以将新行插入 DataTable。在下面的代码中，行 `int id1 = ( int ) parmId.Value;` 校验该行的主键值。

   ```
   SACommand      cmd = conn.CreateCommand();
   cmd.CommandText = "sp_adodotnet_primarykey";
   cmd.CommandType = CommandType.StoredProcedure;
   SAParameter parmId = new SAParameter();
   parmId.SADbType = SADbType.Integer;
   parmId.Direction = ParameterDirection.Output;
   cmd.Parameters.Add( parmId );
   SAParameter parmName = new SAParameter();
   parmName.SADbType = SADbType.Char;
   parmName.Direction = ParameterDirection.Input;
   cmd.Parameters.Add( parmName );
   parmName.Value = "R & D --- Command";
   cmd.ExecuteNonQuery();
   int id1 = ( int ) parmId.Value;
   parmName.Value = "Marketing --- Command";
   cmd.ExecuteNonQuery();
   int id2 = ( int ) parmId.Value;
   parmName.Value = "Sales --- Command";
   cmd.ExecuteNonQuery();
   int id3 = ( int ) parmId.Value;
   parmName.Value = "Shipping --- Command";
   cmd.ExecuteNonQuery();
   int id4 = ( int ) parmId.Value;
   ```

3. 将结果绑定到屏幕上的网格，并将更改应用到数据库。

   ```
   cmd.CommandText = "SELECT * FROM " +
       adodotnet_primarykey";
   cmd.CommandType = CommandType.Text;
   SADataReader dr = cmd.ExecuteReader();
   dataGrid.DataSource = dr;
   ```

4. 关闭该连接。

   ```
   conn.Close();
   ```

♦ 使用 SADataAdapter 对象插入一个包含自动增量主键的新行

1. 创建一个新的 SADataAdapter。

   ```
   DataSet         dataSet = new DataSet();
   SAConnection    conn = OpenConnection();
   SADataAdapter   adapter = new SADataAdapter();
   ```

```
adapter.MissingMappingAction =
    MissingMappingAction.Passthrough;
adapter.MissingSchemaAction =
    MissingSchemaAction.AddWithKey;
```

2. 填充 DataSet 的数据和模式。SADataAdapter.Fill 方法调用 SelectCommand 完成此任务。如果不需要现有记录，还可以不使用 Fill 方法和 SelectCommand 手工创建 DataSet。

```
adapter.SelectCommand = new SACommand( "select * from  +
adodotnet_primarykey", conn );
```

3. 创建一个新的 SACommand 以从数据库获取主键值。

```
adapter.InsertCommand = new SACommand(
    "sp_adodotnet_primarykey", conn );
adapter.InsertCommand.CommandType =
    CommandType.StoredProcedure;
adapter.InsertCommand.UpdatedRowSource =
    UpdateRowSource.OutputParameters;
SAParameter parmId = new SAParameter();
parmId.SADbType = SADbType.Integer;
parmId.Direction = ParameterDirection.Output;
parmId.SourceColumn = "ID";
parmId.SourceVersion = DataRowVersion.Current;
adapter.InsertCommand.Parameters.Add( parmId );
SAParameter parmName = new SAParameter();
parmName.SADbType = SADbType.Char;
parmName.Direction = ParameterDirection.Input;
parmName.SourceColumn = "name";
parmName.SourceVersion = DataRowVersion.Current;
adapter.InsertCommand.Parameters.Add( parmName );
```

4. 填充 DataSet。

```
adapter.Fill( dataSet );
```

5. 将新行插入 DataSet。

```
DataRow row = dataSet.Tables[0].NewRow();
row[0] = -1;
row[1] = "R & D --- Adapter";
dataSet.Tables[0].Rows.Add( row );
row = dataSet.Tables[0].NewRow();
row[0] = -2;
row[1] = "Marketing --- Adapter";
dataSet.Tables[0].Rows.Add( row );
row = dataSet.Tables[0].NewRow();
row[0] = -3;
row[1] = "Sales --- Adapter";
dataSet.Tables[0].Rows.Add( row );
row = dataSet.Tables[0].NewRow();
row[0] = -4;
row[1] = "Shipping --- Adapter";
dataSet.Tables[0].Rows.Add( row );
```

6. 将 DataSet 中的更改应用到数据库。调用 Update 方法时，主键值会更改为从数据库获取的值。

```
adapter.Update( dataSet );
dataGrid.DataSource = dataSet.Tables[0];
```


向 DataTable 中添加新行并调用 Update 方法时，SADataAdapter 会调用 InsertCommand 并将输出参数映射到每个新行的主键列。Update 方法仅调用一次，但只要添加新行，Update 方法就会调用 InsertCommand。

7. 关闭与数据库的连接。

```
conn.Close();
```

处理 BLOB

读取长字符串值或二进制数据时，可以使用一些方法分段读取数据。对于二进制数据，可使用 GetBytes 方法；而对于字符串数据，则可使用 GetChars 方法。否则，BLOB 数据的处理方法与从数据库读取的任何其它数据的处理方法相同。

有关详细信息，请参见“GetBytes 方法”一节第 305 页和“GetChars 方法”一节第 307 页。

♦ 使用 GetChars 方法发出返回字符串的语句

1. 声明并初始化一个 Connection 对象。
2. 打开该连接。
3. 添加一个 Command 对象以定义并执行一条 SQL 语句。

```
SACommand cmd = new SACommand(
    "SELECT int_col, blob_col FROM test", conn );
```

4. 调用 ExecuteReader 方法以返回 DataReader 对象。

```
SADataReader reader = cmd.ExecuteReader();
```

下面的代码读取结果集中的两列。第一列是一个整数 (`GetInt32( 0 )`)，而第二列的类型为 LONG VARCHAR。GetChars 用于从 LONG VARCHAR 列一次读取 100 个字符。

```
int length = 100;
char[] buf = new char[ length ];
int intValue;
long dataIndex = 0;
long charsRead = 0;
long blobLength = 0;
while( reader.Read() ) {
    intValue = reader.GetInt32( 0 );
    while ( ( charsRead = reader.GetChars(
        1, dataIndex, buf, 0, length ) ) == ( long )
            length ) {
    dataIndex += length;
}
blobLength = dataIndex + charsRead;
}
```

5. 关闭 DataReader 和 Connection 对象。

```
reader.Close();
conn.Close();
```

获取时间值

.NET Framework 没有 Time 结构。如果希望从 SQL Anywhere 读取时间值，必须使用 GetTimeSpan 方法。使用此方法会将数据作为 .NET Framework TimeSpan 对象返回。

有关 GetTimeSpan 方法的详细信息，请参见“GetTimeSpan 方法”一节第 320 页。

♦ 使用 GetTimeSpan 方法转换时间值

1. 声明并初始化一个 Connection 对象。

```
SAConnection conn = new SAConnection(
    "Data Source=dsn-time-test;UID=DBA;PWD=sql" );
```

2. 打开该连接。

```
conn.Open();
```

3. 添加一个 Command 对象以定义并执行一条 SQL 语句。

```
SACommand cmd = new SACommand(
    "SELECT ID, time_col FROM time_test", conn )
```

4. 调用 ExecuteReader 方法以返回 DataReader 对象。

```
SADataReader reader = cmd.ExecuteReader();
```

   下面的代码使用 GetTimeSpan 方法将时间作为 TimeSpan 返回。

```
while ( reader.Read() )
{
    int ID = reader.GetInt32();
    TimeSpan time = reader.GetTimeSpan();
}
```

5. 关闭 DataReader 和 Connection 对象。

```
reader.Close();
conn.Close();
```


使用存储过程

可以将存储过程用于 .NET 数据提供程序。ExecuteReader 方法用于调用返回结果集的存储过程，而 ExecuteNonQuery 方法用于调用不返回结果集的存储过程。ExecuteScalar 方法用于调用仅返回单个值的存储过程。

调用存储过程前，必须先创建 SAParameter 对象。使用问号作为参数的占位符，如下所示：

```
sp_producttype( ?, ? )
```

有关 Parameter 对象的详细信息，请参见“SAParameter 类”一节第 367 页。

♦ 执行存储过程

1. 声明并初始化一个 SAConnection 对象。

   ```
   SAConnection conn = new SAConnection(
       "Data Source=SQL Anywhere 11 Demo" );
   ```

2. 打开该连接。

   ```
   conn.Open();
   ```

3. 添加一个 SACommand 对象，以定义并执行一条 SQL 语句。以下代码使用 CommandType 属性将语句标识为存储过程。

   ```
   SACommand cmd = new SACommand( "ShowProductInfo",
       conn );
   cmd.CommandType = CommandType.StoredProcedure;
   ```

   如果不指定 CommandType 属性，则必须使用问号作为参数的占位符，如下所示：

   ```
   SACommand cmd = new SACommand(
       "call ShowProductInfo(?)", conn );
   cmd.CommandType = CommandType.Text;
   ```

4. 添加一个 SAParameter 对象以定义存储过程的参数。必须为存储过程需要的每个参数创建一个新的 SAParameter 对象。

   ```
   SAParameter param = cmd.CreateParameter();
   param.SADbType = SADbType.Int32;
   param.Direction = ParameterDirection.Input;
   param.Value = 301;
   cmd.Parameters.Add( param );
   ```

   有关 Parameter 对象的详细信息，请参见“SAParameter 类”一节第 367 页。

5. 调用 ExecuteReader 方法以返回 DataReader 对象。Get 方法用于以指定的数据类型返回结果。

   ```
   SADataReader reader = cmd.ExecuteReader();
   reader.Read();
   int ID = reader.GetInt32(0);
   string name = reader.GetString(1);
   string descrip = reader.GetString(2);
   decimal price = reader.GetDecimal(6);
   ```

6. 关闭 SADataReader 和 SAConnection 对象。


```
reader.Close();
conn.Close();
```

调用存储过程的替代方法

上述说明的第 3 步介绍了两种调用存储过程的方法。另外一种不使用 Parameter 对象而调用存储过程的方法是从源代码调用存储过程，如下所示：

```
SACommand cmd = new SACommand(
    "call ShowProductInfo( 301 )", conn );
```

有关调用返回结果集或单个值的存储过程的信息，请参见“使用 SACommand 对象获取数据”一节第 111 页。

有关调用不返回结果集的存储过程的信息，请参见“使用 SACommand 对象插入、更新和删除行”一节第 113 页。

事务处理

利用 SQL Anywhere .NET 数据提供程序，可以使用 SATransaction 对象将语句组合到一起。每条语句都以 COMMIT 或 ROLLBACK 结尾，前者将使您对数据库的更改具有永久性，后者则取消事务中的所有操作。事务完成后，必须创建一个新的 SATransaction 对象以进行进一步更改。此行为有别于 ODBC 和嵌入式 SQL，使用后两者时，事务在执行 COMMIT 或 ROLLBACK 后继续存在，直到事务被关闭。

如果不创建事务，缺省情况下 SQL Anywhere .NET 数据提供程序将在自动提交模式下运行。每个插入、更新或删除后都有隐式 COMMIT，操作一旦完成，便已对数据库进行更改。在这种情况下，更改无法回退。

有关 SATransaction 对象的详细信息，请参见“SATransaction 类”一节第 424 页。

为事务设置隔离级别

缺省情况下对事务使用数据库隔离级别。但也可以选择在开始事务时使用 IsolationLevel 属性为事务指定隔离级别。指定的隔离级别适用于事务中执行的所有语句。SQL Anywhere .NET 数据提供程序支持快照隔离。

有关隔离级别的详细信息，请参见“隔离级别和一致性”一节 《SQL Anywhere 服务器 - SQL 的用法》。

输入 SELECT 语句时 SQL Anywhere 使用的锁定取决于事务的隔离级别。

有关锁定和隔离级别的详细信息，请参见“查询过程中的锁定”一节 《SQL Anywhere 服务器 - SQL 的用法》。

下面的示例使用 SATransaction 对象发出然后回退 SQL 语句。事务使用隔离级别 2 (RepeatableRead)，这会将正在修改的行写锁定，这样其他任何数据库用户都无法更新该行。

♦ 使用 SATransaction 对象发出语句

1. 声明并初始化一个 SAConnection 对象。

```
SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 11 Demo" );
```

2. 打开该连接。

```
conn.Open();
```

3. 发出 SQL 语句以更改 T 恤衫的价格。

```
string stmt = "UPDATE Products SET UnitPrice =
    2000.00 WHERE name = 'Tee shirt'";
```

4. 创建一个 SATransaction 对象，以使用 Command 对象发出 SQL 语句。

   使用事务允许您指定隔离级别。本例中使用了隔离级别 2 (RepeatableRead)，以便其他数据库用户无法更新行。

```
SATransaction trans = conn.BeginTransaction(
    IsolationLevel.RepeatableRead );
SACommand cmd = new SACommand( stmt, conn,
```

```
    trans );
int rows = cmd.ExecuteNonQuery();
```

5. 回退更改。

   ```
   trans.Rollback();
   ```

   SATransaction 对象使您可以提交或回退对数据库所做的更改。如果不使用事务，.NET 数据提供程序将在自动提交模式下运行，您将无法回退对数据库所做的任何更改。如果希望这些更改永久保留，可以使用下面的方法：

   ```
   trans.Commit();
   ```

6. 关闭 SAConnection 对象。

   ```
   conn.Close();
   ```

分布式事务处理

.NET 2.0 framework 引入了一个新命名空间 System.Transactions，其中包含用于编写事务性应用程序的类。客户端应用程序可以通过一个或多个参与者创建及参与分布式事务。客户端应用程序可以使用 TransactionScope 类隐式地创建事务。该连接对象可以检测周围存在的由 TransactionScope 创建的事务并且自动征用。客户端应用程序还可以创建一个 CommittableTransaction 并且调用 EnlistTransaction 方法进行征用。SQL Anywhere .NET 2.0 数据提供程序支持此功能。分布式事务具有很高的性能开销。建议对于非分布式事务使用数据库事务。

出错处理和 SQL Anywhere .NET 数据提供程序

您的应用程序必须设计为可处理出现的任何错误，包括 ADO.NET 错误。ADO.NET 错误在代码中的处理方法与应用程序中其它错误的处理方法相同。

只要执行中出现错误，SQL Anywhere .NET 数据提供程序就会抛出 SAException 对象。每个 SAException 对象都包括 SAError 对象列表，而这些错误对象包括错误消息和代码。

错误与冲突不同。冲突发生在将更改应用于数据库时。应用程序中应包括一个用于计算正确值或在发生冲突时记录冲突的过程。

有关处理冲突的详细信息，请参见“解决使用 SADataAdapter 时的冲突”一节第 118 页。

.NET 数据提供程序错误处理示例

下面的示例来自 Simple 示例项目。所有在执行期间发生并且源自 SQL Anywhere .NET 数据提供程序对象的错误均通过在窗口中显示错误消息来处理。以下代码捕获错误并显示其消息：

```
catch( SAException ex ) {
    MessageBox.Show( ex.Errors[0].Message );
}
```

连接错误处理示例

以下示例来自 Table Viewer 示例项目。如果应用程序尝试连接数据库时发生错误，以下代码将使用 try 和 catch 块捕获错误并显示其消息：

```
try {
    _conn = new SAConnection( txtConnectString.Text );
    _conn.Open();
  } catch( SAException ex ) {
    MessageBox.Show( ex.Errors[0].Source + " : "
     + ex.Errors[0].Message + " (" +
     ex.Errors[0].NativeError.ToString() + ")",
         "Failed to connect" );
```

如需更多错误处理示例，请参见“了解 Simple 示例项目”一节第 140 页和“了解 Table Viewer 示例项目”一节第 143 页。

有关错误处理的详细信息，请参见“SAFactory 类”一节第 345 页和“SAError 类”一节第 334 页。

部署 SQL Anywhere .NET 数据提供程序

以下各节介绍如何部署 SQL Anywhere .NET 数据提供程序。

SQL Anywhere .NET 数据提供程序系统要求

要使用 SQL Anywhere .NET 数据提供程序，必须在计算机或手持式设备上安装以下项目：

- .NET Framework 和/或 .NET Compact Framework 2.0 版或更高版本。
- Visual Studio 2005 或更高版本，或者 .NET 语言编译器，例如 C#（仅开发时需要）。

SQL Anywhere .NET 数据提供程序必需的文件

SQL Anywhere .NET 数据提供程序针对每个平台都提供了两个 DLL。

Windows 所需的文件

对于 Windows（除 Windows Mobile 外），需要以下 DLL：

- *install-dir\Assembly\v2\iAnywhere.Data.SQLAnywhere.dll*

文件 *iAnywhere.Data.SQLAnywhere.dll* 是 Visual Studio 项目引用的 DLL。.NET Framework 2.0 版或更高版本的应用程序需要该 DLL。

Windows Mobile 所需的文件

对于 Windows Mobile，需要以下 DLL：

- *install-dir\ce\Assembly\v2\iAnywhere.Data.SQLAnywhere.dll*

文件 *iAnywhere.Data.SQLAnywhere.dll* 是 Visual Studio 项目引用的 DLL。.NET Compact Framework 2.0 版或更高版本的应用程序需要该 DLL。

Visual Studio 将 .NET 数据提供程序 DLL (*iAnywhere.Data.SQLAnywhere.dll*) 与您的程序一同部署到设备。如果未使用 Visual Studio，则需要将数据提供程序 DLL 随您的程序一同复制到设备。它可以位于应用程序所在的目录中，也可以位于 Windows 目录中。

注册 SQL Anywhere .NET 数据提供程序 DLL

SQL Anywhere .NET 数据提供程序 DLL (*install-dir\Assembly\v2\iAnywhere.Data.SQLAnywhere.dll*) 需要在 Windows（Windows Mobile 除外）的 "全局程序集高速缓存" 中注册。全局程序集高速缓存列出了计算机上所有已注册的程序。在安装 .NET 数据提供程序时，.NET 数据提供程序安装程序会对其进行注册。在 Windows Mobile 上无需注册该 DLL。

如果要部署 .NET 数据提供程序，必须使用 .NET Framework 附带的 **gacutil** 实用程序注册 .NET 数据提供程序 DLL (*install-dir\Assembly\v2\iAnywhere.Data.SQLAnywhere.dll*)。

跟踪支持

SQL Anywhere .NET 提供程序支持使用 .NET 2.0 或更高版本的跟踪功能进行跟踪。请注意，Windows Mobile 上不支持跟踪。

缺省情况下，跟踪功能被禁用。要启用跟踪功能，请在应用程序的配置文件中指定跟踪源。这里是配置文件的一个示例：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.diagnostics>
<sources>
 <source name="iAnywhere.Data.SQLAnywhere"
         switchName="SASourceSwitch"
         switchType="System.Diagnostics.SourceSwitch">
  <listeners>
   <add name="ConsoleListener"
        type="System.Diagnostics.ConsoleTraceListener"/>
   <add name="EventListener"
        type="System.Diagnostics.EventLogTraceListener"
        initializeData="MyEventLog"/>
   <add name="TraceLogListener"
        type="System.Diagnostics.TextWriterTraceListener"
        initializeData="myTrace.log"
        traceOutputOptions="ProcessId, ThreadId, Timestamp"/>
   <remove name="Default"/>
  </listeners>
 </source>
</sources>
<switches>
 <add name="SASourceSwitch" value="All"/>
 <add name="SATraceAllSwitch" value="1" />
 <add name="SATraceExceptionSwitch" value="1" />
 <add name="SATraceFunctionSwitch" value="1" />
 <add name="SATracePoolingSwitch" value="1" />
 <add name="SATracePropertySwitch" value="1" />
</switches>
</system.diagnostics>
</configuration>
```

跟踪配置信息位于应用程序的 *bin\debug* 文件夹中，其名称为 *app.exe.config*。

可指定的 `traceOutputOptions` 包括以下内容：

- **Callstack** 编写由 Environment.StackTrace 属性的返回值表示的调用堆栈。
- **DateTime** 编写日期和时间。
- **LogicalOperationStack** 编写由 CorrelationManager.LogicalOperationStack 属性的返回值表示的逻辑操作堆栈。
- **None** 不编写任何元素。
- **ProcessId** 编写由 Process.Id 属性的返回值表示的过程标识。
- **ThreadId** 编写由当前线程的 Thread.ManagedThreadId 属性的返回值表示的线程标识。
- **Timestamp** 编写由 System.Diagnostics.Stopwatch.GetTimeStamp 方法的返回值表示的时间戳。

可通过设置特定的跟踪选项来限制跟踪的内容。缺省情况下，所有的跟踪选项设置均为 0。可设置的跟踪选项包括：

- **SATraceAllSwitch** 跟踪全部开关。指定此选项时，会启用所有的跟踪选项。您无需再设置任何其它选项，因为已选择了所有的选项。如果选择此选项，则无法禁用单独的选项。例如，以下语句将不会禁用异常跟踪。

  ```
  <add name="SATraceAllSwitch" value="1" />
  <add name="SATraceExceptionSwitch" value="0" />
  ```

- **SATraceExceptionSwitch** 记录所有异常。跟踪消息具有以下形式。

  ```
  <Type|ERR> message='message_text'[ nativeError=error_number]
  ```

  仅当具有 SAException 对象时，才会显示 nativeError=error_number 文本。

- **SATraceFunctionSwitch** 记录所有进入/退出函数作用域的操作。跟踪消息具有以下形式之一。

  ```
  enter_nnn <sa.class_name.method_name|API> [object_id#][parameter_names]
  leave_nnn
  ```

  nnn 是一个整数，它表示范围嵌套级别 1、2、3、……可选的 parameter_names 是由空格分隔的参数名列表。

- **SATracePoolingSwitch** 记录所有连接池。跟踪消息具有以下形式之一。

  ```
  <sa.ConnectionPool.AllocateConnection|CPOOL>
  connectionString='connection_text'
  <sa.ConnectionPool.RemoveConnection|CPOOL>
  connectionString='connection_text'
  <sa.ConnectionPool.ReturnConnection|CPOOL>
  connectionString='connection_text'
  <sa.ConnectionPool.ReuseConnection|CPOOL>
  connectionString='connection_text'
  ```

- **SATracePropertySwitch** 记录所有属性设置和检索。跟踪消息具有以下形式之一。

  ```
  <sa.class_name.get_property_name|API> object_id#
  <sa.class_name.set_property_name|API> object_id#
  ```

您可以使用 TableViewer 示例尝试应用程序跟踪。

♦ 配置应用程序以用于跟踪

1. 必须使用 .NET 2.0 或更高版本。

   启动 Visual Studio，然后打开 *samples-dir\SQLAnywhere\ADO.NET\TableViewer* 下的 TableViewer 项目文件 (*TableViewer.sln*)。

2. 将上面所示配置文件的副本置于应用程序的 *bin\debug* 文件夹中，并将其命名为 *TableViewer.exe.config*。

3. 从 [Debug] 菜单中，选择 [Start Debugging]。

应用程序执行完毕后，您会在 *samples-dir\SQLAnywhere\ADO.NET\TableViewer\bin\Debug\myTrace.log* 中找到一个跟踪输出文件。

Windows Mobile 上不支持跟踪。

有关详细信息，请参见 "Tracing Data Access"，网址为 http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnadonet/html/tracingdataaccess.asp。

教程：使用 SQL Anywhere .NET 数据提供程序

目录

.NET 数据提供程序教程简介

本章介绍如何使用 SQL Anywhere .NET 数据提供程序提供的 Simple 和 Table Viewer 示例项目。这些示例项目可与 Visual Studio 2005 或更高版本搭配使用。示例项目是用 Visual Studio 2005 开发的。如果您使用的是更高版本，可能需要运行 Visual Studio 中的 [**Upgrade Wizard**]。

使用 Simple 代码示例

本教程基于 SQL Anywhere 提供的 Simple 项目。

完整的应用程序位于您的 SQL Anywhere 示例目录 *samples-dir\SQLAnywhere\ADO.NET\SimpleWin32* 下。

有关 *samples-dir* 缺省位置的信息，请参见“示例目录”一节 《SQL Anywhere 服务器 - 数据库管理》。

Simple 项目阐述了以下功能：

- 使用 SAConnection 对象连接到数据库
- 使用 SACommand 对象执行查询
- 使用 SADataReader 对象获取结果
- 基本错误处理

有关此示例工作原理的详细信息，请参见“了解 Simple 示例项目”一节第 140 页。

♦ 在 Visual Studio 中运行 Simple 代码示例

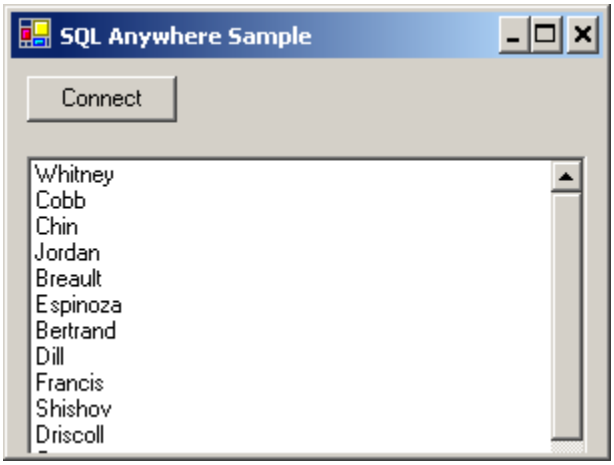
1. 启动 Visual Studio。
2. 选择 [**File**] » [**Open**] » [**Project**]。
3. 浏览至 *samples-dir\SQLAnywhere\ADO.NET\SimpleWin32*，然后打开 *Simple.sln* 项目。
4. 当在项目中使用 SQL Anywhere .NET 数据提供程序时，必须添加一个对数据提供程序 DLL 的引用。这在 Simple 代码示例中已经完成。可在以下位置查看对数据提供程序 DLL 的引用：
   - 在 [**Solution Explorer**] 窗口中打开 [**References**] 文件夹。
   - 您在列表中应该看到 iAnywhere.Data.SQLAnywhere。

     有关添加对数据提供程序 DLL 的引用的说明，请参见“在项目中添加对数据提供程序 DLL 的引用”一节第 106 页。
5. 还必须在源代码中添加一条 `using` 指令以引用数据提供程序类。这在 Simple 代码示例中已经完成。要查看 `using` 指令：
   - 打开项目的源代码。在 [**Solution Explorer**] 窗口中，右击 [*Form1.cs*]，然后选择 [**View Code**]。
   - 在顶部的 `using` 指令中，您应看到以下行：

     ```
     using iAnywhere.Data.SQLAnywhere;
     ```

     C# 项目要求使用这一行指令。如果您使用的是 Visual Basic .NET，则需要将 `Imports` 行添加到源代码中。
6. 选择 [**Debug**] » [**Start Without Debugging**] 或按下 Ctrl+F5 以运行 Simple 示例。
7. 在 [**SQL Anywhere Sample**] 窗口中单击 [**Connect**]。

   应用程序连接到 SQL Anywhere 示例数据库并将每个职员的姓氏放在窗口中，如下图所示：



8. 单击屏幕右上角的 **X**，关闭应用程序并断开与示例数据库的连接。这还会关闭数据库服务器。

您现在已经运行了该应用程序。下一节将介绍应用程序代码。

了解 Simple 示例项目

本节通过介绍 Simple 代码示例中的部分代码，来说明 SQL Anywhere .NET 数据提供程序的一些主要功能。Simple 代码示例使用 SQL Anywhere 示例数据库 *demo.db*，该数据库位于 SQL Anywhere 示例目录中。

有关 SQL Anywhere 示例目录位置的信息，请参见“示例目录”一节 《SQL Anywhere 服务器 - 数据库管理》。

有关示例数据库的信息（包括数据库中的表和它们之间的关系），请参见“SQL Anywhere 示例数据库” 《SQL Anywhere 11 - 简介》。

本节以一次多行的形式来介绍代码。此处并未包括示例中的所有代码。要查看所有代码，请打开 *samples-dir\SQLAnywhere\ADO.NET\SimpleWin32* 中的示例项目。

声明控件 下面的代码声明了一个名为 btnConnect 的按钮和一个名为 listEmployees 的列表框。

```
private System.Windows.Forms.Button btnConnect;
private System.Windows.Forms.ListBox listEmployees;
```

连接到数据库 btnConnect_Click 方法声明并初始化一个 SAConnection 连接对象。

```
private void btnConnect_Click(object sender,
   System.EventArgs e)
   SAConnection conn = new SAConnection(
    "Data Source=SQL Anywhere 11 Demo;UID=DBA;PWD=sql" );
```

SAConnection 对象在调用 Open 方法时使用连接字符串连接到 SQL Anywhere 示例数据库。

```
conn.Open();
```

有关 SAConnection 对象的详细信息，请参见“SAConnection 类”一节第 234 页。

定义查询 使用 SACommand 对象来执行 SQL 语句。以下代码使用 SACommand 构造函数声明并创建一个命令对象。此构造函数接受表示要执行的查询的字符串，以及表示在其上执行查询的连接的 SAConnection 对象。

```
SACommand cmd = new SACommand(
     "SELECT Surname FROM Employees", conn );
```

有关 SACommand 对象的详细信息，请参见“SACommand 类”一节第 195 页。

显示结果 使用 SADataReader 对象来获取查询的结果。以下代码使用 ExecuteReader 构造函数声明并创建一个 SADataReader 对象。此构造函数是以前声明的 SACommand 对象 cmd 的成员。ExecuteReader 将命令文本发送到连接以用于执行并构建一个 SADataReader。

```
SADataReader reader = cmd.ExecuteReader();
```

下面的代码循环检查保存在 SADataReader 对象中的行并将它们添加到列表框控件中。每次调用 Read 方法时，数据读取器都会从结果集中获取另一行。并在列表框中为读取到的每个行添加一个新项目。数据读取器使用带有参数 0 的 GetString 方法，从结果集的行中获取第一列。

```
listEmployees.BeginUpdate();
while( reader.Read() ) {
     listEmployees.Items.Add( reader.GetString( 0 ) );
}
listEmployees.EndUpdate();
```

有关 SADataReader 对象的详细信息，请参见“SADataReader 类”一节第 297 页。

完成 位于该方法末的以下代码将关闭数据读取器和连接对象。

```
reader.Close();
conn.Close();
```

错误处理 所有在执行期间发生并且源自 SQL Anywhere .NET 数据提供程序对象的错误均通过在窗口中显示错误消息来处理。以下代码捕获错误并显示其消息：

```
catch( SAException ex ) {
    MessageBox.Show( ex.Errors[0].Message );
}
```

有关 SAException 对象的详细信息，请参见“SAException 类”一节第 341 页。

使用 Table Viewer 代码示例

本教程基于 SQL Anywhere .NET 数据提供程序提供的 Table Viewer 项目。

完整的应用程序位于您的 SQL Anywhere 示例目录 *samples-dir\SQLAnywhere\ADO.NET\TableViewer* 下。

有关 *samples-dir* 缺省位置的信息，请参见“示例目录”一节 《SQL Anywhere 服务器 - 数据库管理》。

Table Viewer 项目比 Simple 项目更复杂。它阐述了以下功能：

- 使用 SAConnection 对象连接到数据库
- 使用 SACommand 对象执行查询
- 使用 SADataReader 对象获取结果
- 使用网格显示使用 DataGrid 对象的结果
- 更高级的错误处理与结果检查

有关此示例工作原理的详细信息，请参见“了解 Table Viewer 示例项目”一节第 143 页。

♦ 在 Visual Studio 中运行 Table Viewer 代码示例

1. 启动 Visual Studio。
2. 选择 [**File**] » [**Open**] » [**Project**]。
3. 浏览至 *samples-dir\SQLAnywhere\ADO.NET\TableViewer*，然后打开 *TableViewer.sln* 项目。
4. 如果想在项目中使用 SQL Anywhere .NET 数据提供程序，必须添加一个对数据提供程序 DLL 的引用。这在 Table Viewer 代码示例中已经完成。可在以下位置查看对数据提供程序 DLL 的引用：
   - 在 [**Solution Explorer**] 窗口中打开 [**References**] 文件夹。
   - 您在列表中应该看到 iAnywhere.Data.SQLAnywhere。

     有关添加对数据提供程序 DLL 的引用的说明，请参见“在项目中添加对数据提供程序 DLL 的引用”一节第 106 页。
5. 还必须在源代码中添加一条 `using` 指令以引用数据提供程序类。这在 Table Viewer 代码示例中已经完成。要查看 `using` 指令：
   - 打开项目的源代码。在 [**Solution Explorer**] 窗口中，右击 [*TableViewer.cs*]，然后选择 [**View Code**]。
   - 在顶部的 `using` 指令中，您应看到以下行：

     ```
     using iAnywhere.Data.SQLAnywhere;
     ```

     C# 项目要求使用这一行指令。如果您使用的是 Visual Basic，则需要将 `Imports` 行添加到源代码中。

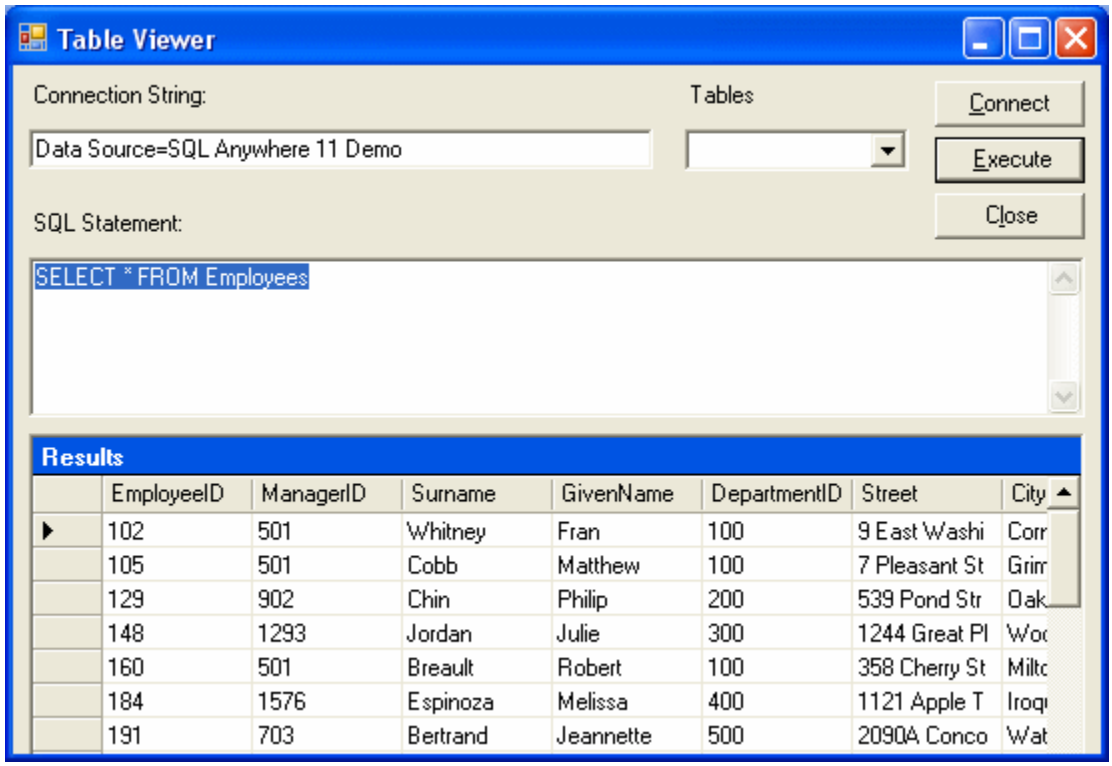
6. 选择 [**Debug**] » [**Start Without Debugging**] 或按下 Ctrl+F5 以运行 Table Viewer 示例。

   应用程序连接到该 SQL Anywhere 示例数据库。

7. 在 [**Table Viewer**] 窗口中单击 [**Connect**]。

8. 在 [**Table Viewer**] 窗口中单击 [**Execute**]。

   应用程序会检索示例数据库中的 Employees 表的数据，然后将查询结果放入 [**Results**] DataGrid 中，如下图所示：



   您还可以通过此应用程序执行其它 SQL 语句：在 [**SQL Statement**] 窗格中键入 SQL 语句，然后单击 [**Execute**]。

9. 单击窗口右上角的 **X**，关闭应用程序并断开与 SQL Anywhere 示例数据库的连接。这还会关闭数据库服务器。

您现在已经运行了该应用程序。下一节将介绍应用程序代码。

了解 Table Viewer 示例项目

本节通过介绍 Table Viewer 代码示例中的部分代码，来说明 SQL Anywhere .NET 数据提供程序的一些主要功能。Table Viewer 项目使用 SQL Anywhere 示例数据库 *demo.db*，该数据库位于您的 SQL Anywhere 示例目录中。

有关 SQL Anywhere 示例目录位置的信息，请参见“示例目录”一节 《SQL Anywhere 服务器 - 数据库管理》。

有关示例数据库的信息（包括数据库中的表和它们之间的关系），请参见“SQL Anywhere 示例数据库” 《SQL Anywhere 11 - 简介》。

本节以一次多行的形式来介绍代码。此处并未包括示例中的所有代码。要查看所有代码，请打开 *samples-dir\SQLAnywhere\ADO.NET\TableViewer* 中的示例项目。

声明控件 下面的代码声明名为 label1 和 label2 的两个标签、名为 txtConnectString 的 TextBox、名为 btnConnect 的按钮、名为 txtSQLStatement 的 TextBox、名为 btnExecute 的按钮，以及名为 dgResults 的 DataGrid。

```
private System.Windows.Forms.Label label1;
private System.Windows.Forms.TextBox txtConnectString;
private System.Windows.Forms.Label label2;
private System.Windows.Forms.Button btnConnect;
private System.Windows.Forms.TextBox txtSQLStatement;
private System.Windows.Forms.Button btnExecute;
private System.Windows.Forms.DataGrid dgResults;
```

声明连接对象 SAConnection 类型用于声明未初始化的 SQL Anywhere 连接对象。SAConnection 对象用于表示与 SQL Anywhere 数据源的唯一连接。

```
private SAConnection _conn;
```

有关 SAConnection 类的详细信息，请参见“SAConnection 类”一节第 234 页。

连接到数据库 txtConnectString 对象的 Text 属性的缺省值为 "Data Source=SQL Anywhere 11 Demo"。应用程序用户可通过向 txtConnectString 文本框键入新值来替换此值。可通过打开标签为 [Windows Form Designer Generated Code] 的 *TableViewer.cs* 内的区域或部分，来查看此缺省值是如何设置的。本节中，您会发现以下代码行。

```
this.txtConnectString.Text = "Data Source=SQL Anywhere 11 Demo";
```

随后，SAConnection 对象使用该连接字符串连接到数据库。以下代码使用 SAConnection 构造函数创建带有连接字符串的新连接对象。它随后会使用 Open 方法建立连接。

```
_conn = new SAConnection( txtConnectString.Text );
_conn.Open();
```

有关 SAConnection 构造函数的详细信息，请参见“SAConnection 成员”一节第 235 页。

定义查询 txtSQLStatement 对象的 Text 属性的缺省值为 "SELECT * FROM Employees"。应用程序用户可通过向 txtSQLStatement 文本框键入新值来替换此值。

使用 SACommand 对象来执行 SQL 语句。以下代码使用 SACommand 构造函数声明并创建一个命令对象。此构造函数接受表示要执行的查询的字符串，以及表示在其上执行查询的连接的 SAConnection 对象。

```
SACommand cmd = new SACommand( txtSQLStatement.Text.Trim(),
                                                _conn );
```

有关 SACommand 对象的详细信息，请参见“SACommand 类”一节第 195 页。

显示结果 使用 SADataReader 对象来获取查询的结果。以下代码使用 ExecuteReader 构造函数声明并创建一个 SADataReader 对象。此构造函数是以前声明的 SACommand 对象 cmd 的成员。ExecuteReader 将命令文本发送到连接以用于执行并构建一个 SADataReader。

```
SADataReader dr = cmd.ExecuteReader();
```

下面的代码将 SADataReader 对象连接到 DataGrid 对象，这样会使结果列出现在屏幕上。随后会关闭 SADataReader 对象。

```
dgResults.DataSource = dr;
dr.Close();
```

有关 SADataReader 对象的详细信息，请参见“SADataReader 类”一节第 297 页。

错误处理 如果应用程序尝试连接数据库或者填充 [Tables] 组合框时发生错误，以下代码将捕获错误并显示其消息：

```
try {
  _conn = new SAConnection( txtConnectString.Text );
  _conn.Open();

  SACommand cmd = new SACommand(
    "SELECT table_name FROM SYS.SYSTAB where creator = 101", _conn );
  SADataReader   dr = cmd.ExecuteReader();

  comboBoxTables.Items.Clear();
  while ( dr.Read() ) {
    comboBoxTables.Items.Add( dr.GetString( 0 ) );
  }
  dr.Close();
} catch( SAException ex ) {
  MessageBox.Show( ex.Errors[0].Source + " : " +
       ex.Errors[0].Message + " (" +
       ex.Errors[0].NativeError.ToString() + ")",
       "Failed to connect" );
}
```

有关 SAException 对象的详细信息，请参见“SAException 类”一节第 341 页。

SQL Anywhere ASP.NET 提供程序

目录

SQL Anywhere ASP.NET 提供程序替换 SQL Server 的标准 ASP.NET 提供程序，允许在 SQL Anywhere 数据库上运行 Web 站点。共有五个提供程序：

- **成员资格提供程序** 成员资格提供程序提供验证和授权服务。成员资格提供程序用于创建新用户和口令，以及校验用户的标识。
- **角色提供程序** 角色提供程序提供了创建角色、将用户添加到角色以及删除角色的方法。角色提供程序用于将用户指派到组和管理权限。
- **配置文件提供程序** 配置文件提供程序提供了读取、存储和检索用户信息的方法。配置文件提供程序用于保存用户首选项。
- **Web 部件个性化提供程序** Web 部件个性化提供程序提供了装载和存储 Web 页的个性化内容和布局的方法。使用 Web 部件个性化提供程序可允许用户创建个性化的 Web 站点视图。
- **健康监视提供程序** 健康监视提供程序提供了监视已部署的 Web 应用程序的状态的方法。健康监视提供程序用于监视应用程序性能、识别发生故障的应用程序或系统，以及记录和查看重要事件。

SQL Anywhere ASP.NET 提供程序使用的 SQL Anywhere 数据库服务器模式与标准 ASP.NET 提供程序使用的模式相同。用于操作和存储数据的方法也相同。

完成设置 SQL Anywhere ASP.NET 提供程序后，可以使用 Visual Studio ASP.NET Web 站点管理工具创建和管理用户及角色。也可以使用 Visual Studio Login、LoginView 和 PasswordRecovery 工具增加 Web 站点的安全性。使用静态包装类访问更高级的提供程序功能，或创建您自己的登录控件。

将 SQL Anywhere ASP.NET 提供程序模式添加到数据库

要实施 SQL Anywhere ASP.NET 提供程序，可以创建新数据库或将该模式添加到现有数据库。

要将该模式添加到现有 SQL Anywhere 数据库，请运行 *SASetupAspNet.exe*。执行后，*SASetupAspNet.exe* 将连接到现有 SQL Anywhere 数据库并创建 SQL Anywhere ASP.NET 提供程序所需的表和存储过程。所有 SQL Anywhere ASP.NET 提供程序资源都以 *aspnet_* 为前缀。要最大程度地减少与现有数据库资源的命名冲突，可以任意数据库用户身份安装提供程序数据库资源。

可以使用向导或命令行运行 *SASetupAspNet.exe*。要访问向导，可运行应用程序，或执行不带参数的命令行语句。使用命令行访问 *SASetupAspNet.exe* 时，使用问号 (-?) 参数显示配置数据库的详细帮助。

设置数据库连接

建议指定具有 DBA 权限的用户的连接字符串。具有 DBA 权限的用户可以为没有必需权限的其他用户创建资源。也可以指定具有 RESOURCE 权限的用户的连接字符串。RESOURCE 特权允许用户创建数据库对象（例如表、视图、存储过程和触发器）。RESOURCE 特权不能通过组成员资格继承，只能由拥有 DBA 特权的用户授予。

指定资源所有者

可以使用向导和命令行指定新资源的所有者。缺省情况下，新资源的所有者是 DBA。指定 SQL Anywhere ASP.NET 提供程序的连接字符串时，将用户指定为 DBA。您不需要向该用户授予任何权限；DBA 拥有资源，并拥有对表和存储过程的完全权限。

选择功能和保留数据

可以添加或删除特定功能。常用组件会自动安装。对于已卸载的功能，选择 [**删除**] 无任何影响；对于已安装的功能，选择 [**添加**] 将重新安装该功能。缺省情况下，将保留与所选功能关联的表中的数据。如果用户对某个表的模式进行较大更改，可能无法自动保留存储在该表中的数据。如果需要彻底重新安装，可以关闭数据保留功能。

建议将成员资格提供程序与角色提供程序一起安装。如果成员资格提供程序不与角色提供程序一起安装，Visual Studio ASP.NET Web 站点管理工具的有效性会降低。

注册连接字符串

有两种方法注册连接字符串：

- 在 ODBC 数据源管理器中注册 ODBC 数据源，并通过名称进行引用。
- 指定完整的 SQL Anywhere 连接字符串。例如：

```
connectionString="ENG=MyServer;DBN=MyDatabase;UID=DBA;PWD=sql"
```

将 <connectionStrings> 元素添加到 *web.config* 文件时，连接字符串及其提供程序可以被应用程序引用。可以在单个位置上实现更新。

注册连接字符串的 XML 代码示例

```
<connectionStrings>
   <add name="MyConnectionString"
      connectionString="DSN=MyDataSource"
      providerName="iAnywhere.Data.SQLAnywhere"/>
</connectionStrings>
```

注册 SQL Anywhere ASP.NET 提供程序

必须将 Web 应用程序配置为使用 SQL Anywhere ASP.NET 提供程序，而不使用缺省提供程序。要注册 SQL Anywhere ASP.NET 提供程序：

- 将对 iAnywhere.Web.Security 程序集的引用添加到 Web 站点。
- 将每个提供程序的条目添加到 *web.config* 文件中的 `<system.web>` 元素。
- 将 SQL Anywhere ASP.NET 提供程序的名称添加到应用程序的 `defaultProvider` 属性。

提供程序数据库可以存储多个应用程序的数据。对于每个应用程序，每个 SQL Anywhere ASP.NET 提供程序的 `applicationName` 属性都必须相同。如果未指定 `applicationName` 值，则将相同的名称指派到提供程序数据库中的每个提供程序。

要引用以前注册的连接字符串，请将 `connectionString` 属性替换为 `connectionStringName` 属性。

注册成员资格提供程序的 XML 代码示例

```
<membership defaultProvider="SAMembershipProvider">
   <providers>
      <add name="SAMembershipProvider"
         type="iAnywhere.Web.Security.SAMembershipProvider"
         connectionStringName="MyConnectionString"
         applicationName="MyApplication"
         commandTimeout="30"
                  enablePasswordReset="true"
         enablePasswordRetrieval="false"
         maxInvalidPasswordAttempts="5"
         minRequiredNonalphanumericCharacters="1"
         minRequiredPasswordLength="7"
         passwordAttemptWindow="10"
         passwordFormat="Hashed"
         requiresQuestionAndAnswer="true"
         requiresUniqueEmail="true"
         passwordStrengthRegularExpression="" />
   </providers>
</membership>
```

有关列说明，请参见“成员资格提供程序 XML 属性”一节第 152 页。

注册角色提供程序的 XML 代码示例

```
<roleManager enabled="true" defaultProvider="SARoleProvider">
   <providers>
      <add name="SARoleProvider"
         type="iAnywhere.Web.Security.SARoleProvider"
         connectionStringName="MyConnectionString"
         applicationName="MyApplication"
         commandTimeout="30" />
   </providers>
</roleManager>
```

有关列说明，请参见“角色提供程序表模式”一节第 153 页。

注册配置文件提供程序的 XML 代码示例

```
<profile defaultProvider="SAProfileProvider">
   <providers>
      <add name="SAProfileProvider"
         type="iAnywhere.Web.Security.SAProfileProvider"
         connectionStringName="MyConnectionString"
         applicationName="MyApplication"
         commandTimeout="30" />
   </providers
      <properties>
         <add name="UserString" type="string"
            serializeAs="Xml" />
         <add name="UserObject" type="object"
            serializeAs="Binary" />
   </properties>
</profile>
```

有关列说明，请参见“配置文件提供程序表模式”一节第 154 页。

注册个性化提供程序的 XML 代码示例

```
<webParts>
   <personalization defaultProvider="SAPersonalizationProvider">
      <providers>
               <add name="SAPersonalizationProvider"
         type="iAnywhere.Web.Security.SAPersonalizationProvider"
         connectionStringName="MyConnectionString"
         applicationName="MyApplication"
         commandTimeout="30" />
            </providers>
   </personalization>
</webParts>
```

有关列说明，请参见“Web 部件个性化提供程序表模式”一节第 155 页。

注册健康监视提供程序的 XML 代码示例

有关设置健康状况监控的详细信息，请参见 Microsoft 网页 "How To: Use Health Monitoring in ASP.NET 2.0 (http://msdn.microsoft.com/en-us/library/ms998306.aspx)。

```
<healthMonitoring enabled="true">
                  ...
   <providers>
      <add name="SAWebEventProvider"
         type="iAnywhere.Web.Security.SAWebEventProvider"
         connectionStringName="MyConnectionString"
         commandTimeout="30"
         bufferMode="Notification"
         maxEventDetailsLength="Infinite" /
   </providers>
               ...
</healthMonitoring>
```

有关列说明，请参见“健康监视提供程序表模式”一节第 156 页。

成员资格提供程序 XML 属性

列名	说明
name	提供程序的名称。
type	`iAnywhere.Web.Security.SAMembershipProvider`
connectionStringName	在 `<connectionStrings>` 元素中指定的连接字符串的名称。
connectionString	连接字符串，为可选。如果未指定 `connectionStringName`，则为必需。
applicationName	提供程序数据所关联的应用程序名。
commandTimeout	服务器调用的超时值，以秒为单位。
enablePasswordReset	有效条目为 true 或 false。
enablePasswordRetrieval	有效条目为 true 或 false。
maxInvalidPasswordAttempts	有效条目为 true 或 false。
minRequiredNonalphanumericCharacters	有效口令中必须出现的特殊字符的最小数量。
minRequiredPasswordLength	口令所需的最小长度。
passwordAttemptWindow	时间窗口，用于跟踪连续多次未能提供有效口令或口令提示答案的情况。
passwordFormat	有效条目为 Clear、Hashed 或 Encrypted。
requiresQuestionAndAnswer	有效条目为 true 或 false。
requiresUniqueEmail	有效条目为 true 或 false。
passwordStrengthRegularExpression	用于计算口令的正则表达式。

角色提供程序表模式

SARoleProvider 将角色信息存储在提供程序数据库的 aspnet_Roles 表中。与 SARoleProvider 关联的命名空间为 iAnywhere.Web.Security。Roles 表中的每条记录均与一个角色相对应。

SARoleProvider 使用 aspnet_UsersInRoles 表将角色映射到用户。

列名	说明
name	提供程序的名称。
type	`iAnywhere.Web.Security.SARoleProvider`
connectionStringName	在 `<connectionStrings>` 元素中指定的连接字符串的名称。
connectionString	连接字符串，为可选。如果未指定 `connectionStringName`，则为必需。
applicationName	提供程序数据所关联的应用程序名。
commandTimeout	服务器调用的超时值，以秒为单位。

配置文件提供程序表模式

SAProfileProvider 将配置文件数据存储在提供程序数据库的 aspnet_Profile 表中。与 SAProfileProvider 关联的命名空间为 iAnywhere.Web.Security。Profile 表中的每条记录均与一个用户的持久配置文件属性相对应。

列名	说明
name	提供程序的名称。
type	`iAnywhere.Web.Security.SAProfileProvider`
connectionStringName	在 `<connectionStrings>` 元素中指定的连接字符串的名称。
connectionString	连接字符串，为可选。如果未指定 `connectionStringName`，则为必需。
applicationName	提供程序数据所关联的应用程序名。
commandTimeout	服务器调用的超时值，以秒为单位。

Web 部件个性化提供程序表模式

SAPersonalizationProvider 将个性化用户内容保存在提供程序数据库的 aspnet_Paths 表中。与 SAPersonalizationProvider 关联的命名空间为 iAnywhere.Web.Security。

SARoleProvider 使用 aspnet_PersonalizationPerUser 和 aspnet_PersonalizationAllUsers 表定义保存 Web 部件个性化状态的路径。PathID 列指向 aspnet_Paths 表中的同名列。

列名	说明
name	提供程序的名称。
type	`iAnywhere.Web.Security.SAPersonalizationProvider`
connectionStringName	在 `<connectionStrings>` 元素中指定的连接字符串的名称。
connectionString	连接字符串，为可选。如果未指定 `connectionStringName`，则为必需。
applicationName	提供程序数据所关联的应用程序名。
commandTimeout	服务器调用的超时值，以秒为单位。

健康监视提供程序表模式

SAWebEventProvider 将 Web 事件记录在提供程序数据库的 aspnet_WebEvent_Events 表中。与 SAWebEventProvider 关联的命名空间为 iAnywhere.Web.Security。WebEvents_Events 表中的每条记录均与一个 Web 事件相对应。

有关设置健康状况监控的详细信息，请参见 Microsoft 网页 "How To: Use Health Monitoring in ASP.NET 2.0 (http://msdn.microsoft.com/en-us/library/ms998306.aspx)。

列名	说明
name	提供程序的名称。
type	`iAnywhere.Web.Security.SAWebEventProvider`
connectionStringName	在 `<connectionStrings>` 元素中指定的连接字符串的名称。
connectionString	连接字符串，为可选。如果未指定 `connectionStringName`，则为必需。
commandTimeout	服务器调用的超时值，以秒为单位。
maxEventDetailsLength	每个事件的详细信息字符串的最大长度或 Infinite

教程：使用 Visual Studio 开发简单的 .NET 数据库应用程序

目录

第 1 课：创建表查看器

本教程以 Visual Studio 和 .NET Framework 为基础。完整的应用程序可在 ADO.NET 项目 *samples-dir\SQLAnywhere\ADO.NET\SimpleViewer\SimpleViewer.sln* 中找到。

在本教程中，您将使用 Microsoft Visual Studio、Server Explorer 和 SQL Anywhere .NET 数据提供程序创建一个访问 SQL Anywhere 示例数据库中某个表格的应用程序，它可以检查行并执行更新。

♦ 使用 Visual Studio 开发数据库应用程序

1. 启动 Visual Studio。
2. 从 Visual Studio [**File**] 菜单选择 [**New**] » [**Project**]。

   随即出现 [**New Project**] 窗口。

   a. 在 [**New Project**] 窗口的左侧窗格中，选择 [**Visual Basic**] 或 [**Visual C#**] 编程语言。

   b. 从 **Windows** 子类中，选择 [**Windows Application**] (VS 2005) 或 [**Windows Forms Application**] (VS 2008)。

   c. 在项目 [**Name**] 字段中，键入 [**MySimpleViewer**]。

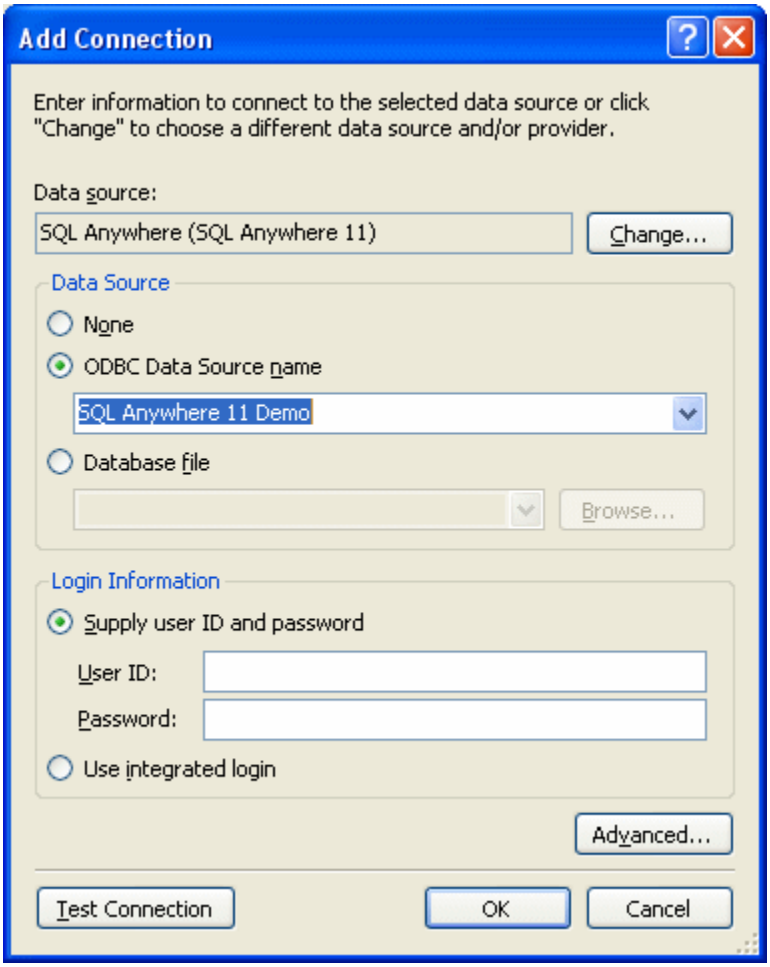
   d. 单击 [**OK**] 创建新项目。
3. 从 Visual Studio [**View**] 菜单选择 [**Server Explorer**]。
4. 在 [**Server Explorer**] 窗口中，右击 [**Data Connections**]，然后选择 [**Add Connection**]。

   名为 **SQL Anywhere.demo11** 的新连接随即出现在 [**Server Explorer**] 窗口中。
5. 在 [**Add Connection**] 窗口中：

   a. 如果从未对其它项目使用 [**Add Connection**]，则您会看到一个数据源列表。从显示的数据源列表中选择 [**SQL Anywhere**]。

   如果之前已经使用 [**Add Connection**]，则单击 [**Change**] 以将数据源更改为 [**SQL Anywhere**]。

   b. 在 [**Data Source**] 下面，选择 [**ODBC Data Source Name**]，然后键入 [**SQL Anywhere 11 Demo**]。



c. 单击 [**Test Connection**] 以验证您可以连接到示例数据库。

d. 单击 [**OK**]。

6. 在 [**Server Explorer**] 窗口中展开 **SQL Anywhere.demo11** 连接，直到看到表名称为止。

   a. 右击 Products 表，然后选择 [**Show Table Data**]。

      这样，窗口中会出现 Products 表的行和列。

   b. 关闭表数据窗口。

7. 从 Visual Studio 的 [**Data**] 菜单中，选择 [**Add New Data Source**]。

8. 在 [**Data Source Configuration Wizard**] 中，执行以下操作：

   a. 在 [**Data Source Type**] 页面上，选择 [**Database**]，然后单击 [**Next**]。

   b. 在 [**Data Connection**] 页面上，选择 [**SQL Anywhere.demo11**]，然后单击 [**Next**]。

   c. 在 [**Save The Connection String**] 页面上，确认已选择 [**Yes, Save The Connection As**]，然后单击 [**Next**]。

d. 在 [**Choose Your Database Objects**] 页面上，选择 [**Tables**]，然后单击 [**Finish**]。

9. 从 Visual Studio 的 [**Data**] 菜单中，选择 [**Show Data Sources**]。

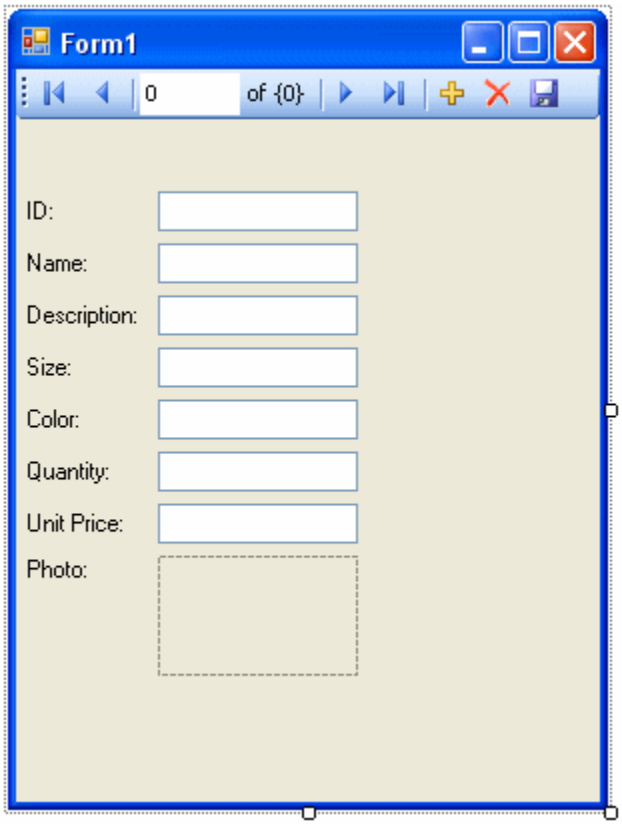
   随即出现 [**Data Sources**] 窗口。

   在 [**Data Sources**] 窗口中展开 Products 表。

   a. 从下拉列表中单击 [Products] 并选择 [**Details**]。

   b. 从下拉列表中单击 [Photo] 并选择 [**Picture Box**]。

   c. 单击 [Products] 并将其拖动到您的窗体 (Form1) 中。



   数据集控件和几个带标签的文本字段出现在窗体上。

10. 在窗体上，选择 [Photo] 旁边的图片框。

    a. 将框形状更改为方形。

    b. 单击图片框右上角的右箭头键。

       [**Picture Box Tasks**] 窗口随即打开。

    c. 从 [**Size Mode**] 下拉列表中，选择 [**Zoom**]。

    d. 要关闭 [**Picture Box Tasks**] 窗口，请单击窗口外的任何位置。

11. 构建并运行项目。
    a. 从 Visual Studio [**Build**] 菜单中，选择 [**Build Solution**]。
    b. 从 Visual Studio [**Debug**] 菜单中，选择 [**Start Debugging**]。

       应用程序连接到 SQL Anywhere 示例数据库，并在文本框和图片框中显示 Products 表的第一行。

    c. 可以使用控件上的按钮滚动浏览结果集中的行。
    d. 可以通过在滚动控件中输入行号直接转到结果集中的该行。
    e. 可以使用文本框更新结果集中的值并通过单击磁盘图标进行保存。

       您现在已经使用 Visual Studio、Server Explorer 和 SQL Anywhere .NET 数据提供程序创建了一个简单但强大的 .NET 应用程序。

12. 关闭应用程序，然后保存您的项目。

第 2 课：添加同步数据控件

本教程是“第 1 课：创建表查看器”一节第 158 页中所述教程的后续。完整的应用程序可在 ADO.NET 项目 *samples-dir\SQLAnywhere\ADO.NET\SimpleViewer\SimpleViewer.sln* 中找到。

在本教程中，您将在前一教程期间所开发的窗体中添加 datagrid 控件。在您浏览结果集时，此控件会自动进行更新。

♦ 添加 datagrid 控件

1. 启动 Visual Studio 并装载您在“第 1 课：创建表查看器”一节第 158 页中创建的 MySimpleViewer 项目。
2. 在 [**Data Sources**] 窗口中右击 DataSet1，并选择 [**Edit DataSet With Designer**]。
3. 右击 [**DataSet Designer**] 窗口中的空白区域，并选择 [**Add**] » [**TableAdapter**]。
4. 在 [**TableAdapter Configuration Wizard**] 中：
   a. 在 [**Choose Your Data Connection**] 页面上，单击 [**Next**]。
   b. 在 [**Choose A Command Type**] 页面上，确保选择了 [**Use SQL Statements**]，然后单击 [**Next**]。
   c. 在 [**Enter A SQL Statement**] 页面上，单击 [**Query Builder**]。
   d. 在 [**Add Table**] 窗口上，单击 [**Views**] 选项卡，选择 [**ViewSalesOrders**]，然后单击 [**Add**]。
   e. 单击 [**Close**] 关闭 [**Add Table**] 窗口。
5. 展开 [**Query Builder**] 窗口，使窗口中的所有部分都可见。
   a. 展开 [**ViewSalesOrders**] 窗口，使所有复选框可见。
   b. 选择 [**Region**]。
   c. 选择 [**Quantity**]。
   d. 选择 [**ProductID**]。
   e. 在 [**ViewSalesOrders**] 窗口下面的网格中，清除 [ProductID] 列的 [**Output**] 下面的复选框。
   f. 对于 ProductID 列，在 [**Filter**] 单元格中键入问号 (?)。这将为 ProductID 生成 WHERE 子句。

   此时构建出的 SQL 查询如下：

```
SELECT   Region, Quantity
FROM     GROUPO.ViewSalesOrders
WHERE    (ProductID = :Param1)
```

6. 按如下步骤修改 SQL 查询：
   a. 将 `Quantity` 更改为 `SUM(Quantity) AS TotalSales`。
   b. 将 `GROUP BY Region` 添加到 WHERE 子句后的查询末尾。

   修改后的 SQL 查询此时如下所示：

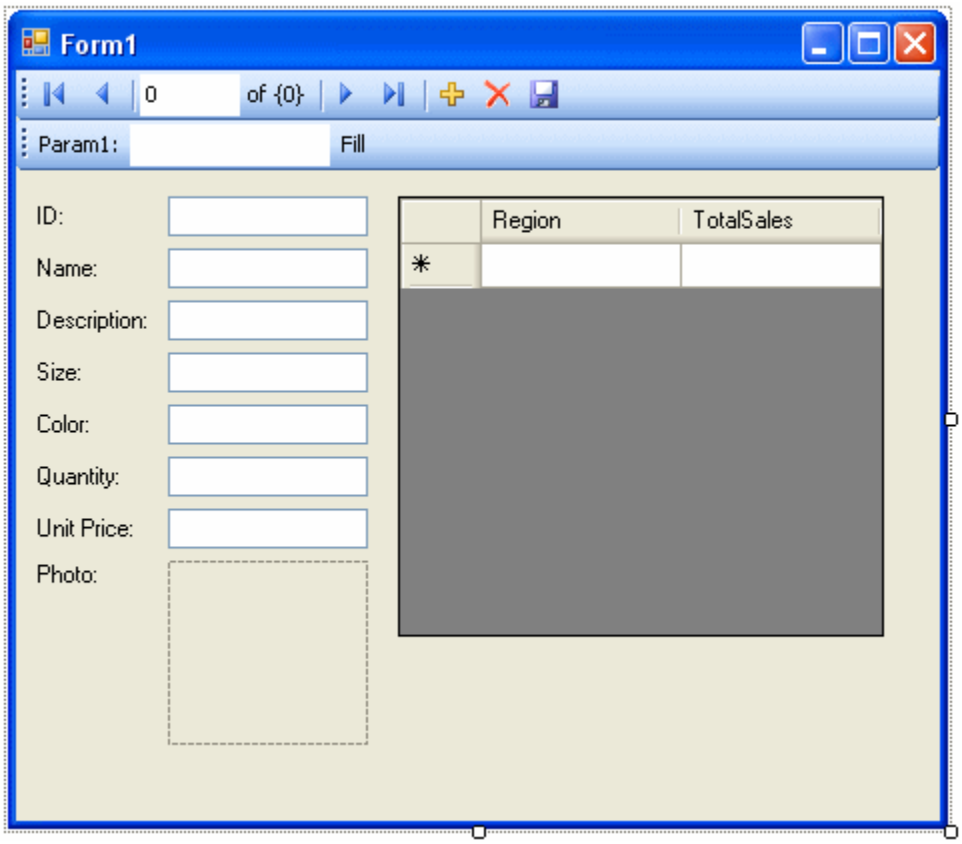
```
SELECT   Region, SUM(Quantity) as TotalSales
FROM     GROUPO.ViewSalesOrders
```



```
WHERE     (ProductID = :Param1)
GROUP BY Region
```

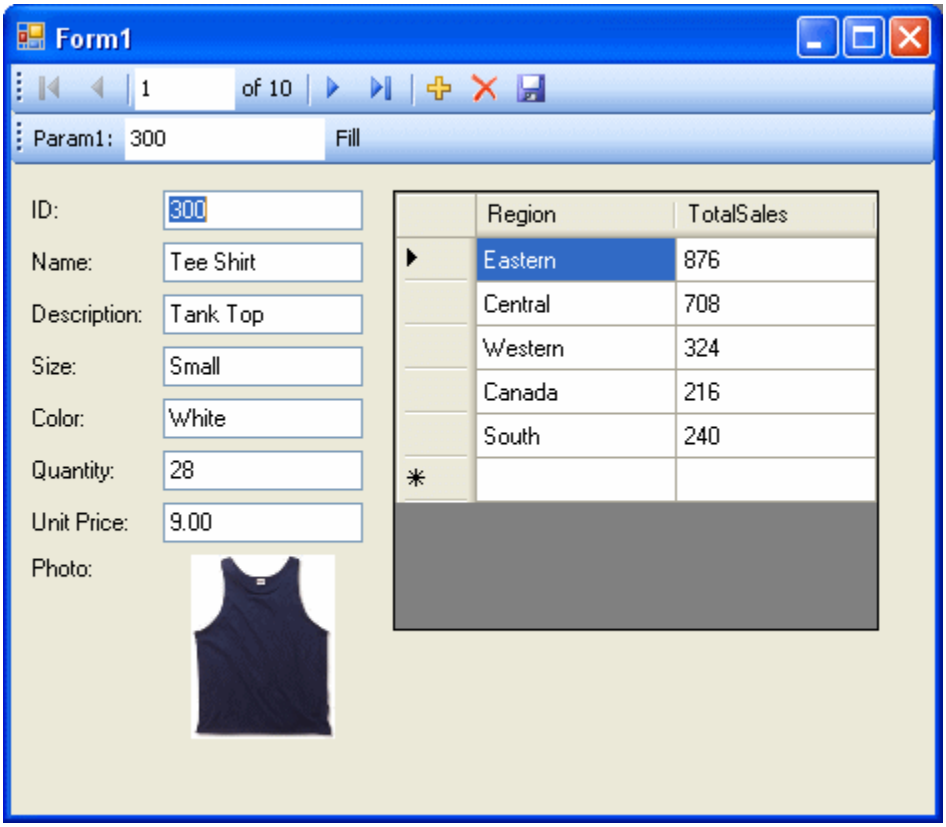
7. 单击 [**确定**]。
8. 单击 [**完成**]。

   名为 **ViewSalesOrders** 的新 **TableAdapter** 已添加到 [**DataSet Designer**] 窗口。
9. 单击窗体设计选项卡 (Form1)。
   - 向右拉伸窗体以为新控件留出空间。
10. 在 [**Data Sources**] 窗口中展开 ViewSalesOrders。
    a. 单击 [ViewSalesOrders] 并从下拉列表中选择 [**DataGridView**]。
    b. 单击 [ViewSalesOrders] 并将其拖动到您的窗体 (Form1) 中。



   datagrid 视图控件出现在窗体上。
11. 构建并运行项目。
    - 从 Visual Studio [**Build**] 菜单中，选择 [**Build Solution**]。
    - 从 Visual Studio [**Debug**] 菜单中，选择 [**Start Debugging**]。

- 在 [**Param1**] 文本框中，输入产品 ID 号（例如 300），然后单击 [**Fill**]。
  datagrid 视图针对所输入的产品 ID 按区域显示销售额的汇总。



还可以使用窗体上的其它控件在结果集的各行之间移动。

但如果两个控件可以互相保持同步会比较理想。以下几步说明了实现同步的方法。

12. 关闭应用程序，然后保存您的项目。
13. 删除窗体上的 [Fill] 条，因为您不需要它。
    - 在设计窗体 (Form1) 上，右击 [**Fill**] 一词右侧的 [Fill] 条并选择 [**Delete**]。
      [Fill] 条即从窗体上删除。
14. 按如下所示同步这两个控件。
    a. 在设计窗体 (Form1) 上，右击 [ID] 文本框，然后选择 [**Properties**]。
    b. 单击 [**Events**] 图标（它显示为一道闪电）。
    c. 向下滚动直到找到 [**TextChanged**] 事件。
    d. 单击 [**TextChanged**] 并从下拉列表中选择 [**FillToolStripButton_Click**]。如果您在使用 Visual Basic，则事件称为 **FillToolStripButton_Click**。

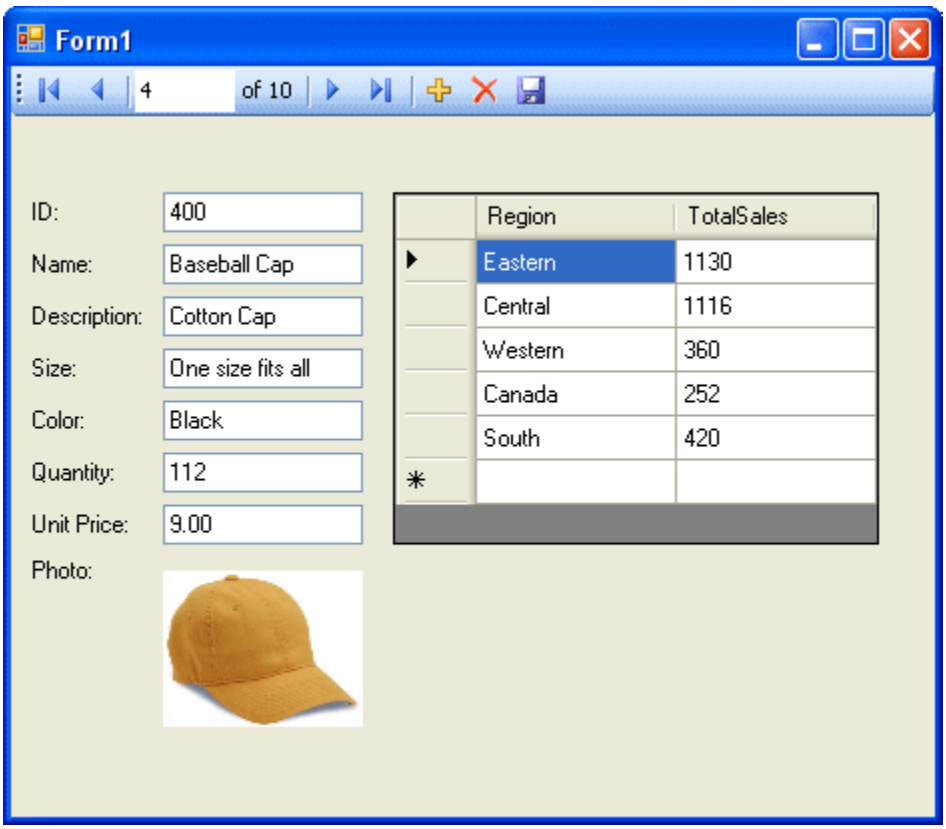
e. 双击 [**FillToolStripButton_Click**]，窗体的代码窗口即在 `fillToolStripButton_Click` 事件处理程序上打开。

f. 找到对 `param1ToolStripTextBox` 的引用并将其更改为 `iDTextBox`。如果您在使用 Visual Basic，则文本框称为 `IDTextBox`。

g. 重建并运行项目。

15. 应用程序窗体现在带有一个导航控件。

- 当您在结果集中移动时，datagrid 视图会针对当前产品按区域显示更新的销售额汇总。



您现在已添加了会随着您对结果集的浏览自动更新的控件。

16. 关闭应用程序，然后保存您的项目。

在这些教程中，您学到了如何利用 Microsoft Visual Studio、Server Explorer 和 SQL Anywhere .NET 数据提供程序的强大组合来创建数据库应用程序。

SQL Anywhere .NET 2.0 API 参考

目录

iAnywhere.Data.SQLAnywhere 命名空间 (.NET 2.0)

SABulkCopy 类

以有效率的方式将另一数据源中的数据批量装载到 SQL Anywhere 表中。此类无法继承。

语法

Visual Basic

```
Public NotInheritable Class SABulkCopy
  Implements IDisposable
```

C#

```
public sealed class SABulkCopy : IDisposable
```

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopy 类。

Implements: IDisposable

另请参见

- “SABulkCopy 成员”一节第 168 页

SABulkCopy 成员

公共构造函数

成员名称	说明
SABulkCopy 构造函数	初始化 SABulkCopy 对象。

公共属性

成员名称	说明
BatchSize 属性	获取或设置每个批处理中的行数。每个批处理结束时，该批处理中的行将发送到服务器。
BulkCopyTimeout 属性	获取或设置操作在超时之前完成所需的秒数。
ColumnMappings 属性	返回 SABulkCopyColumnMapping 项的集合。列映射定义数据源中的列与目标中的列之间的关系。
DestinationTableName 属性	获取或设置服务器上目标表的名称。

成员名称	说明
NotifyAfter 属性	获取或设置要在生成通知事件之前处理的行数。

公共方法

成员名称	说明
Close 方法	关闭 SABulkCopy 实例。
Dispose 方法	消除 SABulkCopy 实例。
WriteToServer 方法	将所提供的 DataRow 对象数组中的所有行复制到由 SABulkCopy 对象的 DestinationTableName 属性所指定的目标表中。

公共事件

成员名称	说明
SARowsCopied 事件	每次处理完 NotifyAfter 属性所指定的行数时便会发生此事件。

另请参见

- “SABulkCopy 类”一节第 168 页

SABulkCopy 构造函数

初始化 SABulkCopy 对象。

SABulkCopy(SAConnection) 构造函数

语法

Visual Basic

```
Public Sub New( _
  ByVal connection As SAConnection _
)
```

C#

```
public  SABulkCopy(
  SAConnection connection
);
```

参数

- **connection** 将用于执行批量复制操作的已打开的 SAConnection。如果连接未打开，则 WriteToServer 中会抛出异常。

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopy 类。

另请参见

- “SABulkCopy 类”一节第 168 页
- “SABulkCopy 成员”一节第 168 页
- “SABulkCopy 构造函数”一节第 169 页

SABulkCopy(String) 构造函数

初始化 SABulkCopy 对象。

语法

Visual Basic

```
Public Sub New( _
  ByVal connectionString As String _
)
```

C#

```
public  SABulkCopy(
  string  connectionString
);
```

参数

- **connectionString** 一个字符串，用于定义将被打开以供 SABulkCopy 实例使用的连接。连接字符串是以分号分隔的 "关键字=值" 对的列表。

注释

此语法在 WriteToServer 期间使用 connectionString 打开连接。连接在 WriteToServer 结束时关闭。

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopy 类。

另请参见

- “SABulkCopy 类”一节第 168 页
- “SABulkCopy 成员”一节第 168 页
- “SABulkCopy 构造函数”一节第 169 页

SABulkCopy(String, SABulkCopyOptions) 构造函数

初始化 SABulkCopy 对象。

语法

Visual Basic

```
Public Sub New( _
  ByVal connectionString As String, _
  ByVal copyOptions As SABulkCopyOptions _
)
```

C#

```
public  SABulkCopy(
  string  connectionString,
  SABulkCopyOptions copyOptions
);
```

参数

- **connectionString**　一个字符串，用于定义将被打开以供 SABulkCopy 实例使用的连接。连接字符串是以分号分隔的 "关键字=值" 对的列表。
- **copyOptions**　SABulkCopyOptions 枚举中的值组合，用于确定将哪些数据源行复制到目标表中。

注释

此语法在 WriteToServer 期间使用 connectionString 打开连接。连接在 WriteToServer 结束时关闭。copyOptions 参数具有以上所述的影响。

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopy 类。

另请参见

- “SABulkCopy 类”一节第 168 页
- “SABulkCopy 成员”一节第 168 页
- “SABulkCopy 构造函数”一节第 169 页

SABulkCopy(SAConnection, SABulkCopyOptions, SATransaction) 构造函数

初始化 SABulkCopy 对象。

语法

Visual Basic

```
Public Sub New( _
  ByVal connection As SAConnection, _
  ByVal copyOptions As SABulkCopyOptions, _
  ByVal externalTransaction As SATransaction _
)
```

C#

```
public  SABulkCopy(
  SAConnection connection,
```

```
    SABulkCopyOptions copyOptions,
    SATransaction externalTransaction
);
```

参数

- **connection** 将用于执行批量复制操作的已打开的 SAConnection。如果连接未打开，则 WriteToServer 中会抛出异常。
- **copyOptions** SABulkCopyOptions 枚举中的值组合，用于确定将哪些数据源行复制到目标表中。
- **externalTransaction** 一个现有的 SATransaction 实例，批量复制将在该实例下进行。如果 externalTransaction 不为 NULL，批量复制操作将在其内部进行。同时指定 externalTransaction 和 UseInternalTransaction 选项是错误的。

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopy 类。

另请参见

- “SABulkCopy 类”一节第 168 页
- “SABulkCopy 成员”一节第 168 页
- “SABulkCopy 构造函数”一节第 169 页

BatchSize 属性

获取或设置每个批处理中的行数。每个批处理结束时，该批处理中的行将发送到服务器。

语法

Visual Basic

```
Public Property BatchSize As Integer
```

C#

```
public int BatchSize { get; set; }
```

属性值

每个批处理中的行数。缺省值为 0。

注释

将此属性设置为零会使所有行都通过一个批处理进行发送。

将此属性设置为小于零的值是错误的。

如果在批处理执行期间更改了此值，当前批处理仍将使用更改前的值完成，但所有后续批处理都将使用新值。

另请参见

- “SABulkCopy 类”一节第 168 页
- “SABulkCopy 成员”一节第 168 页

BulkCopyTimeout 属性

获取或设置操作在超时之前完成所需的秒数。

语法

Visual Basic

Public Property **BulkCopyTimeout** As Integer

C#

public int **BulkCopyTimeout** { get; set; }

属性值

缺省值为 30 秒。

注释

值为零表示没有限制。应该避免使用该值，因为这可能会导致无限期的等待。

如果操作超时，将回退当前事务中的所有行，并抛出 SAException。

将此属性设置为小于零的值是错误的。

另请参见

- “SABulkCopy 类”一节第 168 页
- “SABulkCopy 成员”一节第 168 页

ColumnMappings 属性

返回 SABulkCopyColumnMapping 项的集合。列映射定义数据源中的列与目标中的列之间的关系。

语法

Visual Basic

Public Readonly Property **ColumnMappings** As SABulkCopyColumnMappingCollection

C#

public SABulkCopyColumnMappingCollection **ColumnMappings** { get;}

属性值

缺省情况下它是一个空集合。

注释

WriteToServer 执行期间无法对该属性进行修改。

如果执行 WriteToServer 时 ColumnMappings 为空，则数据源中的第一列将映射到目标中的第一列，其第二列将映射到目标的第二列，以此类推。这种映射的发生条件是：列类型可以转换、目标的列数至少与数据源的列数一样多且目标的任何额外列都可以为空。

另请参见

- “SABulkCopy 类”一节第 168 页
- “SABulkCopy 成员”一节第 168 页

DestinationTableName 属性

获取或设置服务器上目标表的名称。

语法

Visual Basic

Public Property **DestinationTableName** As String

C#

public string **DestinationTableName** { get; set; }

属性值

缺省值为空值引用。Visual Basic 中则是 Nothing。

注释

如果在 WriteToServer 执行期间更改了该值，这种更改不会产生任何影响。

如果在调用 WriteToServer 之前未设置该值，将会抛出 InvalidOperationException。

将该值设置为 NULL 或空字符串是错误的。

另请参见

- “SABulkCopy 类”一节第 168 页
- “SABulkCopy 成员”一节第 168 页

NotifyAfter 属性

获取或设置要在生成通知事件之前处理的行数。

语法

Visual Basic

Public Property **NotifyAfter** As Integer

C#

public int **NotifyAfter** { get; set; }

属性值

如果未设置该属性，将会返回零。

注释

WriteToServer 执行期间对 NotifyAfter 进行的更改到下次通知后才会生效。

将此属性设置为小于零的值是错误的。

NotifyAfter 与 BulkCopyTimeOut 的值相互排斥，因此，即使未将行发送到数据库或未提交行，也可以触发事件。

另请参见

- “SABulkCopy 类”一节第 168 页
- “SABulkCopy 成员”一节第 168 页
- “BulkCopyTimeout 属性”一节第 173 页

Close 方法

关闭 SABulkCopy 实例。

语法

Visual Basic

Public Sub **Close()**

C#

public void **Close();**

另请参见

- “SABulkCopy 类”一节第 168 页
- “SABulkCopy 成员”一节第 168 页

Dispose 方法

消除 SABulkCopy 实例。

语法

Visual Basic

NotOverridable Public Sub **Dispose()**

C#

```
public void Dispose();
```

另请参见

- “SABulkCopy 类”一节第 168 页
- “SABulkCopy 成员”一节第 168 页

WriteToServer 方法

将所提供的 DataRow 对象数组中的所有行复制到由 SABulkCopy 对象的 DestinationTableName 属性所指定的目标表中。

WriteToServer(DataRow[]) 方法

将所提供的 DataRow 对象数组中的所有行复制到由 SABulkCopy 对象的 DestinationTableName 属性所指定的目标表中。

语法

Visual Basic

```
Public Sub WriteToServer( _
  ByVal rows As DataRow() _
)
```

C#

```
public void WriteToServer(
  DataRow[] rows
);
```

参数

- **rows** 将要复制到目标表中的 System.Data.DataRow 对象的数组。

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopy 类。

另请参见

- “SABulkCopy 类”一节第 168 页
- “SABulkCopy 成员”一节第 168 页
- “WriteToServer 方法”一节第 176 页
- “DestinationTableName 属性”一节第 174 页

WriteToServer(DataTable) 方法

将提供的 DataTable 中的所有行复制到由 SABulkCopy 对象的 DestinationTableName 属性所指定的目标表中。

语法

Visual Basic

```
Public Sub WriteToServer( _
  ByVal table As DataTable _
)
```

C#

```
public void WriteToServer(
  DataTable table
);
```

参数

- **table** 其行将被复制到目标表中的一个 System.Data.DataTable。

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopy 类。

另请参见

- “SABulkCopy 类”一节第 168 页
- “SABulkCopy 成员”一节第 168 页
- “WriteToServer 方法”一节第 176 页
- “DestinationTableName 属性”一节第 174 页

WriteToServer(IDataReader) 方法

将提供的 IDataReader 中的所有行复制到由 SABulkCopy 对象的 DestinationTableName 属性所指定的目标表中。

语法

Visual Basic

```
Public Sub WriteToServer( _
  ByVal reader As IDataReader _
)
```

C#

```
public void WriteToServer(
  IDataReader reader
);
```

参数

- **reader** 一个 System.Data.IDataReader，它的行将复制到目标表中。

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopy 类。

另请参见

- “SABulkCopy 类”一节第 168 页
- “SABulkCopy 成员”一节第 168 页
- “WriteToServer 方法”一节第 176 页
- “DestinationTableName 属性”一节第 174 页

WriteToServer(DataTable, DataRowState) 方法

将提供的 DataTable 中的所有具有指定行状态的行复制到由 SABulkCopy 对象的 DestinationTableName 属性所指定的目标表中。

语法

Visual Basic

```
Public Sub WriteToServer( _
  ByVal table As DataTable, _
  ByVal rowState As DataRowState _
)
```

C#

```
public void WriteToServer(
  DataTable table,
  DataRowState rowState
);
```

参数

- **table** 其行将被复制到目标表中的一个 System.Data.DataTable。
- **rowState** System.Data.DataRowState 枚举中的一个值。只有与行状态匹配的行才会复制到目标表中。

注释

只会复制与行状态匹配的行。

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopy 类。

另请参见

- “SABulkCopy 类”一节第 168 页
- “SABulkCopy 成员”一节第 168 页
- “WriteToServer 方法”一节第 176 页
- “DestinationTableName 属性”一节第 174 页

SARowsCopied 事件

每次处理完 NotifyAfter 属性所指定的行数时便会发生此事件。

语法

Visual Basic

Public Event **SARowsCopied** As SARowsCopiedEventHandler

C#

public event SARowsCopiedEventHandler **SARowsCopied** ;

注释

接收到 SARowsCopied 事件并不意味着已将任何行发送到数据库服务器或已提交任何行。无法通过此事件调用 Close 方法。

另请参见

- “SABulkCopy 类”一节第 168 页
- “SABulkCopy 成员”一节第 168 页
- “NotifyAfter 属性”一节第 174 页

SABulkCopyColumnMapping 类

定义 SABulkCopy 实例的数据源中的列与该实例目标表中的列之间的映射。此类无法继承。

语法

Visual Basic

Public NotInheritable Class **SABulkCopyColumnMapping**

C#

public sealed class **SABulkCopyColumnMapping**

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopyColumnMapping 类。

另请参见

- “SABulkCopyColumnMapping 成员”一节第 180 页

SABulkCopyColumnMapping 成员

公共构造函数

成员名称	说明
SABulkCopyColumnMapping 构造函数	初始化一个新的“SABulkCopyColumnMapping 类”一节第 179 页实例。

公共属性

成员名称	说明
DestinationColumn 属性	获取或设置映射到的目标数据库表中列的名称。
DestinationOrdinal 属性	获取或设置映射到的目标表中列的顺序值。
SourceColumn 属性	获取或设置数据源中映射的列的名称。
SourceOrdinal 属性	获取或设置源列在数据源内的顺序位置。

另请参见

- “SABulkCopyColumnMapping 类”一节第 179 页

SABulkCopyColumnMapping 构造函数

初始化一个新的“SABulkCopyColumnMapping 类”一节第 179 页实例。

SABulkCopyColumnMapping() 构造函数

通过使用列顺序号或列名引用源列和目标列来创建新的列映射。

语法

Visual Basic

Public Sub **New()**

C#

public **SABulkCopyColumnMapping();**

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopyColumnMapping 类。

另请参见

- “SABulkCopyColumnMapping 类”一节第 179 页
- “SABulkCopyColumnMapping 成员”一节第 180 页
- “SABulkCopyColumnMapping 构造函数”一节第 180 页

SABulkCopyColumnMapping(Int32, Int32) 构造函数

通过使用列顺序号引用源列和目标列来创建新的列映射。

语法

Visual Basic

```
Public Sub New( _
  ByVal sourceColumnOrdinal As Integer, _
  ByVal destinationColumnOrdinal As Integer _
)
```

C#

```
public  SABulkCopyColumnMapping(
  int sourceColumnOrdinal,
  int destinationColumnOrdinal
);
```

参数

- **sourceColumnOrdinal** 源列在数据源内的顺序位置。数据源中第一列的顺序位置为零。
- **destinationColumnOrdinal** 目标列在目标表内的顺序位置。表中第一列的顺序位置为零。

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopyColumnMapping 类。

另请参见

- “SABulkCopyColumnMapping 类”一节第 179 页
- “SABulkCopyColumnMapping 成员”一节第 180 页
- “SABulkCopyColumnMapping 构造函数”一节第 180 页

SABulkCopyColumnMapping(Int32, String) 构造函数

通过使用列顺序号引用源列和使用列名引用目标列来创建新的列映射。

语法

Visual Basic

```
Public Sub New( _
  ByVal sourceColumnOrdinal As Integer, _
  ByVal destinationColumn As String _
)
```

C#

```
public  SABulkCopyColumnMapping(
  int sourceColumnOrdinal,
  string  destinationColumn
);
```

参数

- **sourceColumnOrdinal**　源列在数据源内的顺序位置。数据源中第一列的顺序位置为零。
- **destinationColumn**　目标列在目标表内的名称。

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopyColumnMapping 类。

另请参见

- “SABulkCopyColumnMapping 类”一节第 179 页
- “SABulkCopyColumnMapping 成员”一节第 180 页
- “SABulkCopyColumnMapping 构造函数”一节第 180 页

SABulkCopyColumnMapping(String, Int32) 构造函数

通过使用列名引用源列和使用列顺序号引用目标列来创建新的列映射。

语法

Visual Basic

```
Public Sub New( _
  ByVal sourceColumn As String, _
  ByVal destinationColumnOrdinal As Integer _
)
```

C#

```
public  SABulkCopyColumnMapping(
  string  sourceColumn,
  int destinationColumnOrdinal
);
```

参数

- **sourceColumn**　源列在数据源内的名称。
- **destinationColumnOrdinal**　目标列在目标表内的顺序位置。表中第一列的顺序位置为零。

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopyColumnMapping 类。

另请参见

- “SABulkCopyColumnMapping 类”一节第 179 页
- “SABulkCopyColumnMapping 成员”一节第 180 页
- “SABulkCopyColumnMapping 构造函数”一节第 180 页

SABulkCopyColumnMapping(String, String) 构造函数

通过使用列名引用源列和目标列来创建新的列映射。

语法

Visual Basic

```
Public Sub New( _
  ByVal sourceColumn As String, _
  ByVal destinationColumn As String _
)
```

C#

```
public  SABulkCopyColumnMapping(
  string  sourceColumn,
  string  destinationColumn
);
```

参数

- **sourceColumn** 源列在数据源内的名称。
- **destinationColumn** 目标列在目标表内的名称。

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopyColumnMapping 类。

另请参见

- “SABulkCopyColumnMapping 类”一节第 179 页
- “SABulkCopyColumnMapping 成员”一节第 180 页
- “SABulkCopyColumnMapping 构造函数”一节第 180 页

DestinationColumn 属性

获取或设置映射到的目标数据库表中列的名称。

语法

Visual Basic

```
Public Property DestinationColumn As String
```

C#

public string **DestinationColumn** { get; set; }

属性值

在 DestinationOrdinal 属性有优先级时，指定目标表中列的名称或空值引用（在 Visual Basic 中是 Nothing）的字符串。

注释

DestinationColumn 属性与 DestinationOrdinal 属性互相排斥。最近设置的值优先。

设置 DestinationColumn 属性会使 DestinationOrdinal 属性被设置为 -1；设置 DestinationOrdinal 属性会使 DestinationColumn 属性被设置为空值引用（在 Visual Basic 中是 Nothing）。

将 DestinationColumn 设置为空值或空字符串是错误的。

另请参见

- “SABulkCopyColumnMapping 类”一节第 179 页
- “SABulkCopyColumnMapping 成员”一节第 180 页
- “DestinationOrdinal 属性”一节第 184 页

DestinationOrdinal 属性

获取或设置映射到的目标表中列的顺序值。

语法

Visual Basic

Public Property **DestinationOrdinal** As Integer

C#

public int **DestinationOrdinal** { get; set; }

属性值

指定映射到的目标表中列的顺序号的整数，未设置该属性时值为 -1。

注释

DestinationColumn 属性与 DestinationOrdinal 属性互相排斥。最近设置的值优先。

设置 DestinationColumn 属性会使 DestinationOrdinal 属性被设置为 -1；设置 DestinationOrdinal 属性会使 DestinationColumn 属性被设置为空值引用（在 Visual Basic 中是 Nothing）。

另请参见

- “SABulkCopyColumnMapping 类”一节第 179 页
- “SABulkCopyColumnMapping 成员”一节第 180 页
- “DestinationColumn 属性”一节第 183 页

SourceColumn 属性

获取或设置数据源中映射的列的名称。

语法

Visual Basic

Public Property **SourceColumn** As String

C#

public string **SourceColumn** { get; set; }

属性值

在 SourceOrdinal 属性有优先级时，指定数据源中列的名称或空值引用（在 Visual Basic 中是 Nothing）的字符串。

注释

SourceColumn 属性与 SourceOrdinal 属性相互排斥。最近设置的值优先。

设置 SourceColumn 属性会使 SourceOrdinal 属性被设置为 -1；设置 SourceOrdinal 属性会使 SourceColumn 属性被设置为空值引用（在 Visual Basic 中是 Nothing）。

将 SourceColumn 设置为空值或空字符串是错误的。

另请参见

- “SABulkCopyColumnMapping 类”一节第 179 页
- “SABulkCopyColumnMapping 成员”一节第 180 页
- “SourceOrdinal 属性”一节第 185 页

SourceOrdinal 属性

获取或设置源列在数据源内的顺序位置。

语法

Visual Basic

Public Property **SourceOrdinal** As Integer

C#

public int **SourceOrdinal** { get; set; }

属性值

指定数据源中列的顺序号的整数，未设置该属性时值为 -1。

注释

SourceColumn 属性与 SourceOrdinal 属性相互排斥。最近设置的值优先。

设置 SourceColumn 属性会使 SourceOrdinal 属性被设置为 -1；设置 SourceOrdinal 属性会使 SourceColumn 属性被设置为空值引用（在 Visual Basic 中是 Nothing）。

另请参见

- “SABulkCopyColumnMapping 类”一节第 179 页
- “SABulkCopyColumnMapping 成员”一节第 180 页
- “SourceColumn 属性”一节第 185 页

SABulkCopyColumnMappingCollection 类

从 System.Collections.CollectionBase 继承的 SABulkCopyColumnMapping 对象的集合。此类无法继承。

语法

Visual Basic

```
Public NotInheritable Class SABulkCopyColumnMappingCollection
  Inherits CollectionBase
```

C#

```
public sealed class SABulkCopyColumnMappingCollection : CollectionBase
```

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopyColumnMappingCollection 类。

另请参见

- “SABulkCopyColumnMappingCollection 成员”一节第 186 页

SABulkCopyColumnMappingCollection 成员

公共属性

成员名称	说明
Capacity（继承自 CollectionBase）	获取或设置 CollectionBase 可以包含的元素数量。
Count（继承自 CollectionBase）	获取包含在 CollectionBase 实例中的元素的数量。此属性不能被替换。
Item 属性	获取指定索引处的 SABulkCopyColumnMapping 对象。

公共方法

成员名称	说明
Add 方法	将指定的 SABulkCopyColumnMapping 对象添加到集合中。
Clear（继承自 CollectionBase）	删除 CollectionBase 实例的所有对象。此方法不能被替换。
Contains 方法	获取表示集合中是否存在指定 SABulkCopyColumnMapping 对象的值。
CopyTo 方法	从特定索引处开始将 SABulkCopyColumnMappingCollection 的元素复制到 SABulkCopyColumnMapping 项的数组中。
GetEnumerator（继承自 CollectionBase）	返回通过 CollectionBase 实例进行迭代的枚举器。
IndexOf 方法	获取或设置集合内指定 SABulkCopyColumnMapping 对象的索引。
Remove 方法	从 SABulkCopyColumnMappingCollection 中删除指定的 SABulkCopyColumnMapping 元素。
RemoveAt 方法	从集合中删除指定索引处的映射。

另请参见

- “SABulkCopyColumnMappingCollection 类”一节第 186 页

Item 属性

获取指定索引处的 SABulkCopyColumnMapping 对象。

语法

Visual Basic

```
Public Readonly Property Item ( _
   ByVal index As Integer _
) As SABulkCopyColumnMapping
```

C#

```
public SABulkCopyColumnMapping this [
   int index
] { get;}
```

参数

- **index** 要查找的 SABulkCopyColumnMapping 对象的从零开始索引。

属性值

返回 SABulkCopyColumnMapping 对象。

另请参见

- “SABulkCopyColumnMappingCollection 类”一节第 186 页
- “SABulkCopyColumnMappingCollection 成员”一节第 186 页

Add 方法

将指定的 SABulkCopyColumnMapping 对象添加到集合中。

Add(SABulkCopyColumnMapping) 方法

将指定的 SABulkCopyColumnMapping 对象添加到集合中。

语法

Visual Basic

```
Public Function Add( _
  ByVal bulkCopyColumnMapping As SABulkCopyColumnMapping _
) As SABulkCopyColumnMapping
```

C#

```
public SABulkCopyColumnMapping Add(
  SABulkCopyColumnMapping bulkCopyColumnMapping
);
```

参数

- **bulkCopyColumnMapping** 对要添加到集合中的映射进行说明的 SABulkCopyColumnMapping 对象。

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopyColumnMappingCollection 类。

另请参见

- “SABulkCopyColumnMappingCollection 类”一节第 186 页
- “SABulkCopyColumnMappingCollection 成员”一节第 186 页
- “Add 方法”一节第 188 页
- “SABulkCopyColumnMapping 类”一节第 179 页

Add(Int32, Int32) 方法

通过使用顺序号指定源列和目标列来创建新的 SABulkCopyColumnMapping 对象，并将该映射添加到集合中。

语法

Visual Basic

```
Public Function Add( _
  ByVal sourceColumnOrdinal As Integer, _
  ByVal destinationColumnOrdinal As Integer _
) As SABulkCopyColumnMapping
```

C#

```
public SABulkCopyColumnMapping Add(
  int sourceColumnOrdinal,
  int destinationColumnOrdinal
);
```

参数

- **sourceColumnOrdinal** 源列在数据源内的顺序位置。
- **destinationColumnOrdinal** 目标列在目标表内的顺序位置。

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopyColumnMappingCollection 类。

另请参见

- “SABulkCopyColumnMappingCollection 类”一节第 186 页
- “SABulkCopyColumnMappingCollection 成员”一节第 186 页
- “Add 方法”一节第 188 页

Add(Int32, String) 方法

通过使用列顺序号引用源列和使用列名引用目标列来创建新的 SABulkCopyColumnMapping 对象，并将该映射添加到集合中。

语法

Visual Basic

```
Public Function Add( _
  ByVal sourceColumnOrdinal As Integer, _
  ByVal destinationColumn As String _
) As SABulkCopyColumnMapping
```

C#

```
public SABulkCopyColumnMapping Add(
  int sourceColumnOrdinal,
```

```
  string  destinationColumn
);
```

参数

- **sourceColumnOrdinal** 源列在数据源内的顺序位置。
- **destinationColumn** 目标列在目标表内的名称。

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopyColumnMappingCollection 类。

另请参见

- “SABulkCopyColumnMappingCollection 类”一节第 186 页
- “SABulkCopyColumnMappingCollection 成员”一节第 186 页
- “Add 方法”一节第 188 页

Add(String, Int32) 方法

通过使用列名引用源列和使用列顺序号引用目标列来创建新的 SABulkCopyColumnMapping 对象，并将该映射添加到集合中。

通过使用列顺序号或列名引用源列和目标列来创建新的列映射。

语法

Visual Basic

```
Public Function Add( _
  ByVal sourceColumn As String, _
  ByVal destinationColumnOrdinal As Integer _
) As SABulkCopyColumnMapping
```

C#

```
public SABulkCopyColumnMapping Add(
  string  sourceColumn,
  int destinationColumnOrdinal
);
```

参数

- **sourceColumn** 源列在数据源内的名称。
- **destinationColumnOrdinal** 目标列在目标表内的顺序位置。

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopyColumnMappingCollection 类。

另请参见

- “SABulkCopyColumnMappingCollection 类”一节第 186 页
- “SABulkCopyColumnMappingCollection 成员”一节第 186 页
- “Add 方法”一节第 188 页

Add(String, String) 方法

通过使用列名指定源列和目标列来创建新的 SABulkCopyColumnMapping 对象，并将该映射添加到集合中。

语法

Visual Basic

```
Public Function Add( _
  ByVal sourceColumn As String, _
  ByVal destinationColumn As String _
) As SABulkCopyColumnMapping
```

C#

```
public SABulkCopyColumnMapping Add(
  string  sourceColumn,
  string  destinationColumn
);
```

参数

- **sourceColumn** 源列在数据源内的名称。
- **destinationColumn** 目标列在目标表内的名称。

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopyColumnMappingCollection 类。

另请参见

- “SABulkCopyColumnMappingCollection 类”一节第 186 页
- “SABulkCopyColumnMappingCollection 成员”一节第 186 页
- “Add 方法”一节第 188 页

Contains 方法

获取表示集合中是否存在指定 SABulkCopyColumnMapping 对象的值。

语法

Visual Basic

```
Public Function Contains( _
```

```
  ByVal value As SABulkCopyColumnMapping _
) As Boolean
```

C#

```
public bool Contains(
  SABulkCopyColumnMapping value
);
```

参数

- **value** 有效的 SABulkCopyColumnMapping 对象。

返回值

如果集合中存在指定的映射，则为 true；否则为 false。

另请参见

- “SABulkCopyColumnMappingCollection 类”一节第 186 页
- “SABulkCopyColumnMappingCollection 成员”一节第 186 页

CopyTo 方法

从特定索引处开始将 SABulkCopyColumnMappingCollection 的元素复制到 SABulkCopyColumnMapping 项的数组中。

语法

Visual Basic

```
Public Sub CopyTo( _
  ByVal array As SABulkCopyColumnMapping(), _
  ByVal index As Integer _
)
```

C#

```
public void CopyTo(
  SABulkCopyColumnMapping[] array,
  int index
);
```

参数

- **array** 作为从 SABulkCopyColumnMappingCollection 复制的元素目标的一维 SABulkCopyColumnMapping 数组。该数组必须具有从零开始的索引。
- **index** 数组中复制开始时所在的从零开始索引。

另请参见

- “SABulkCopyColumnMappingCollection 类”一节第 186 页
- “SABulkCopyColumnMappingCollection 成员”一节第 186 页

IndexOf 方法

获取或设置集合内指定 SABulkCopyColumnMapping 对象的索引。

语法

Visual Basic

```
Public Function IndexOf( _
  ByVal value As SABulkCopyColumnMapping _
) As Integer
```

C#

```
public int IndexOf(
  SABulkCopyColumnMapping value
);
```

参数

- **value** 要搜索的 SABulkCopyColumnMapping 对象。

返回值

返回列映射的从零开始索引；如果未在集合中找到列映射，则返回 -1。

另请参见

- “SABulkCopyColumnMappingCollection 类”一节第 186 页
- “SABulkCopyColumnMappingCollection 成员”一节第 186 页

Remove 方法

从 SABulkCopyColumnMappingCollection 中删除指定的 SABulkCopyColumnMapping 元素。

语法

Visual Basic

```
Public Sub Remove( _
  ByVal value As SABulkCopyColumnMapping _
)
```

C#

```
public void Remove(
  SABulkCopyColumnMapping value
);
```

参数

- **value** 要从集合中删除的 SABulkCopyColumnMapping 对象。

另请参见

- “SABulkCopyColumnMappingCollection 类”一节第 186 页
- “SABulkCopyColumnMappingCollection 成员”一节第 186 页

RemoveAt 方法

从集合中删除指定索引处的映射。

语法

Visual Basic

```
Public Sub RemoveAt( _
  ByVal index As Integer _
)
```

C#

```
public void RemoveAt(
  int index
);
```

参数

- **index** 要从集合中删除的 SABulkCopyColumnMapping 对象的从零开始索引。

另请参见

- “SABulkCopyColumnMappingCollection 类”一节第 186 页
- “SABulkCopyColumnMappingCollection 成员”一节第 186 页

SABulkCopyOptions 枚举

指定一个或多个与 SABulkCopy 实例配合使用的选项的逐位标志。

语法

Visual Basic

```
Public Enum SABulkCopyOptions
```

C#

```
public enum SABulkCopyOptions
```

注释

构造 SABulkCopy 对象时，可以使用 SABulkCopyOptions 枚举来指定 WriteToServer 方法的行为方式。

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SABulkCopyOptions 类。

不支持 CheckConstraints 和 KeepNulls 选项。

成员

成员名称	说明	值
Default	仅指定此值会引发缺省行为的使用。缺省情况下将会启用触发器。	0
DoNotFireTriggers	指定此值时不会触发触发器。禁用触发器需要 DBA 权限。将在 WriteToServer 启动时为连接禁用触发器，并在该方法执行结束时恢复原值。	1
KeepIdentity	指定此值时将保留要复制到标识列中的源值。缺省情况下，将在目标表中生成新标识值。	2
TableLock	指定此值时，将使用 LOCK TABLE table_name WITH HOLD IN SHARE MODE 命令锁定表。此锁将一直保持到连接关闭时。	4
UseInternalTransaction	指定此值时每个批量复制操作批处理都在事务内执行。未指定此值时，将不使用事务。如果指定此选项，并为构造函数提供 SATransaction 对象，则会发生 System.ArgumentException。	8

另请参见

- “SABulkCopy 类”一节第 168 页

SACommand 类

对 SQL Anywhere 数据库执行的 SQL 语句或存储过程。此类无法继承。

语法

Visual Basic

```
Public NotInheritable Class SACommand
  Inherits DbCommand
  Implements ICloneable
```

C#

```
public sealed class SACommand : DbCommand,
  ICloneable
```

注释

Implements:ICloneable

有关详细信息，请参见“访问和操作数据”一节第 111 页。

另请参见

- “SACommand 成员”一节第 196 页

SACommand 成员

公共构造函数

成员名称	说明
SACommand 构造函数	初始化一个新的“SACommand 类”一节第 195 页实例。

公共属性

成员名称	说明
CommandText 属性	获取或设置 SQL 语句或存储过程的文本。
CommandTimeout 属性	获取或设置终止执行命令的尝试并生成错误之前等待的时间（以秒为单位）。
CommandType 属性	获取或设置 SACommand 所表示的命令类型。
Connection 属性	获取或设置应用 SACommand 对象的连接对象。
DesignTimeVisible 属性	获取或设置表示是否应使 SACommand 在 [Windows 窗体设计器] 控件中可见的值。缺省值为 true。
Parameters 属性	当前语句的参数集合。在 CommandText 中使用问号来表示参数。
Transaction 属性	指定 SACommand 执行时所处的 SATransaction 对象。
UpdatedRowSource 属性	获取或设置当 SADataAdapter 的 Update 方法使用命令结果时将命令结果应用于 DataRow 的方法。

公共方法

成员名称	说明
BeginExecuteNonQuery 方法	启动由此 SACommand 说明的 SQL 语句或存储过程的异步执行。
BeginExecuteReader 方法	启动由此 SACommand 说明的 SQL 语句或存储过程的异步执行，并从数据库服务器中检索一个或多个结果集。
Cancel 方法	取消 SACommand 对象的执行。

成员名称	说明
CreateParameter 方法	提供用于为 SACommand 对象提供参数的 SAParameter 对象。
EndExecuteNonQuery 方法	完成 SQL 语句或存储过程的异步执行。
EndExecuteReader 方法	完成 SQL 语句或存储过程的异步执行，并返回所请求的 SADataReader。
ExecuteNonQuery 方法	执行不返回结果集的语句，如 INSERT、UPDATE、DELETE 或数据定义语句。
ExecuteReader 方法	执行返回结果集的 SQL 语句。
ExecuteScalar 方法	执行返回单个值的语句。如果在返回多个行和列的查询上调用该方法，则只会返回第一行的第一列。
Prepare 方法	在数据源上准备或编译 SACommand。
ResetCommandTimeout 方法	将 CommandTimeout 属性重置为其缺省值 30 秒。

另请参见

- “SACommand 类”一节第 195 页

SACommand 构造函数

初始化一个新的“SACommand 类”一节第 195 页实例。

SACommand() 构造函数

初始化 SACommand 对象。

语法

Visual Basic

```
Public Sub New()
```

C#

```
public  SACommand();
```

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页
- “SACommand 构造函数”一节第 197 页

SACommand(String) 构造函数

初始化 SACommand 对象。

语法

Visual Basic

```
Public Sub New( _
  ByVal cmdText As String _
)
```

C#

```
public  SACommand(
  string  cmdText
);
```

参数

- **cmdText** SQL 语句或存储过程的文本。如果是参数化语句，请使用问号 (?) 占位符来传递参数。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页
- “SACommand 构造函数”一节第 197 页

SACommand(String, SAConnection) 构造函数

对 SQL Anywhere 数据库执行的 SQL 语句或存储过程。

语法

Visual Basic

```
Public Sub New( _
  ByVal cmdText As String, _
  ByVal connection As SAConnection _
)
```

C#

```
public  SACommand(
  string  cmdText,
  SAConnection connection
);
```

参数

- **cmdText** SQL 语句或存储过程的文本。如果是参数化语句，请使用问号 (?) 占位符来传递参数。
- **connection** 当前连接。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页
- “SACommand 构造函数”一节第 197 页

SACommand(String, SAConnection, SATransaction) 构造函数

对 SQL Anywhere 数据库执行的 SQL 语句或存储过程。

语法

Visual Basic

```
Public Sub New( _
  ByVal cmdText As String, _
  ByVal connection As SAConnection, _
  ByVal transaction As SATransaction _
)
```

C#

```
public  SACommand(
  string  cmdText,
  SAConnection connection,
  SATransaction transaction
);
```

参数

- **cmdText**　SQL 语句或存储过程的文本。如果是参数化语句，请使用问号 (?) 占位符来传递参数。
- **connection**　当前连接。
- **transaction**　SAConnection 执行时所处的 SATransaction 对象。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页
- “SACommand 构造函数”一节第 197 页
- “SATransaction 类”一节第 424 页

CommandText 属性

获取或设置 SQL 语句或存储过程的文本。

语法

Visual Basic

```
Public Overrides Property CommandText As String
```

C#

public override string **CommandText** { get; set; }

属性值

要执行的 SQL 语句或存储过程的名称。缺省值为空字符串。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页
- “SACommand() 构造函数”一节第 197 页

CommandTimeout 属性

获取或设置终止执行命令的尝试并生成错误之前等待的时间（以秒为单位）。

语法

Visual Basic

Public Overrides Property **CommandTimeout** As Integer

C#

public override int **CommandTimeout** { get; set; }

属性值

缺省值为 30 秒。

注释

值为 0 表示没有限制。应该避免使用该值，因为这可能会导致执行命令的尝试进行无限期的等待。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页

CommandType 属性

获取或设置 SACommand 所表示的命令类型。

语法

Visual Basic

Public Overrides Property **CommandType** As CommandType

C#

public override CommandType **CommandType** { get; set; }

属性值

CommandType 值之一。缺省值为 CommandType.Text。

注释

所支持的命令类型如下：

- CommandType.StoredProcedure 指定此 CommandType 时，命令文本必须为存储过程的名称，而且所提供的任何参数都必须以 SAParameter 对象形式出现。
- CommandType.Text 这是缺省值。

当 CommandType 属性设置为 StoredProcedure 时，CommandText 属性应设置为该存储过程的名称。当您调用其中一个 Execute 方法时，该命令就会执行此存储过程。

使用问号 (?) 占位符传递参数。例如：

```
SELECT * FROM Customers WHERE ID = ?
```

将 SAParameter 对象添加到 SAParameterCollection 的顺序必须与该参数问号占位符的位置直接对应。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页

Connection 属性

获取或设置应用 SACommand 对象的连接对象。

语法

Visual Basic

Public Property **Connection** As SAConnection

C#

public SAConnection **Connection** { get; set; }

属性值

缺省值为空值引用。Visual Basic 中则是 Nothing。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页

DesignTimeVisible 属性

获取或设置表示是否应使 SACommand 在 [Windows 窗体设计器] 控件中可见的值。缺省值为 true。

语法

Visual Basic

Public Overrides Property **DesignTimeVisible** As Boolean

C#

public override bool **DesignTimeVisible** { get; set; }

属性值

如果应使此 SACommand 实例可见，则为 true；否则为 false。缺省值为 false。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页

Parameters 属性

当前语句的参数集合。在 CommandText 中使用问号来表示参数。

语法

Visual Basic

Public Readonly Property **Parameters** As SAParameterCollection

C#

public SAParameterCollection **Parameters** { get;}

属性值

SQL 语句或存储过程的参数。缺省值为空集合。

注释

当 CommandType 设置为 Text 时，请使用问号占位符传递参数。例如：

```
SELECT * FROM Customers WHERE ID = ?
```

将 SAParameter 对象添加到 SAParameterCollection 的顺序必须与命令文本中该参数问号占位符的位置直接对应。

当集合中的参数与要执行的查询的要求不匹配时，可能会导致错误或抛出异常。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页
- “SAParameterCollection 类”一节第 380 页

Transaction 属性

指定 SACommand 执行时所处的 SATransaction 对象。

语法

Visual Basic

Public Property **Transaction** As SATransaction

C#

public SATransaction **Transaction** { get; set; }

属性值

缺省值为空值引用。在 Visual Basic 中为 Nothing。

注释

如果 Transaction 属性已设置为特定值且正在执行命令，则无法设置该属性。如果将 transaction 属性设置为某个 SATransaction 对象，而该 SATransaction 对象并未连接到 SACommand 对象所连接到的 SAConnection 对象，则您下次尝试执行语句时将会抛出异常。

有关详细信息，请参见“事务处理”一节第 129 页。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页
- “SATransaction 类”一节第 424 页

UpdatedRowSource 属性

获取或设置当 SADataAdapter 的 Update 方法使用命令结果时将命令结果应用于 DataRow 的方法。

语法

Visual Basic

Public Overrides Property **UpdatedRowSource** As UpdateRowSource

C#

public override UpdateRowSource **UpdatedRowSource** { get; set; }

属性值

UpdatedRowSource 的值之一。缺省值为 UpdateRowSource.OutputParameters。如果该命令自动生成，则此属性为 UpdateRowSource.None。

注释

不支持返回结果集和输出参数的 UpdatedRowSource.Both。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页

BeginExecuteNonQuery 方法

启动由此 SACommand 说明的 SQL 语句或存储过程的异步执行。

BeginExecuteNonQuery() 方法

启动由此 SACommand 说明的 SQL 语句或存储过程的异步执行。

语法

Visual Basic

```
Public Function BeginExecuteNonQuery() As IAsyncResult
```

C#

```
public IAsyncResult BeginExecuteNonQuery();
```

返回值

可只用于进行轮询、只用于等待结果或作轮询和等待结果这两种用途的 IAsyncResult；调用 EndExecuteNonQuery(IAsyncResult)（返回受影响的行数）时也需要此值。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页
- “BeginExecuteNonQuery 方法”一节第 204 页
- “EndExecuteNonQuery 方法”一节第 209 页

BeginExecuteNonQuery(AsyncCallback, Object) 方法

在已知回调过程和状态信息的情况下启动由此 SACommand 说明的 SQL 语句或存储过程的异步执行。

语法

Visual Basic

```
Public Function BeginExecuteNonQuery( _
  ByVal callback As AsyncCallback, _
  ByVal stateObject As Object _
) As IAsyncResult
```

C#

```
public IAsyncResult BeginExecuteNonQuery(
  AsyncCallback callback,
  object stateObject
);
```

参数

- **callback** 命令执行完成时调用的 AsyncCallback 委派。传递空值（在 Microsoft Visual Basic 中是 Nothing）表示不需要回调。
- **stateObject** 传递给回调过程的用户定义状态对象。使用 IAsyncResult.AsyncState 从回调过程中检索此对象。

返回值

可只用于进行轮询、只用于等待结果或作轮询和等待结果这两种用途的 IAsyncResult；调用 EndExecuteNonQuery(IAsyncResult)（返回受影响的行数）时也需要此值。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页
- “BeginExecuteNonQuery 方法”一节第 204 页
- “EndExecuteNonQuery 方法”一节第 209 页

BeginExecuteReader 方法

启动由此 SACommand 说明的 SQL 语句或存储过程的异步执行，并从数据库服务器中检索一个或多个结果集。

BeginExecuteReader() 方法

启动由此 SACommand 说明的 SQL 语句或存储过程的异步执行，并从数据库服务器中检索一个或多个结果集。

语法

Visual Basic

```
Public Function BeginExecuteReader() As IAsyncResult
```

C#

```
public IAsyncResult BeginExecuteReader();
```

返回值

可只用于进行轮询、只用于等待结果或作轮询和等待结果这两种用途的 IAsyncResult；调用 EndExecuteReader(IAsyncResult)（返回可用于检索返回行的 SADataReader 对象）时也需要此值。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页
- “BeginExecuteReader 方法”一节第 205 页
- “EndExecuteReader 方法”一节第 211 页
- “SADataReader 类”一节第 297 页

BeginExecuteReader(CommandBehavior) 方法

启动由此 SACommand 说明的 SQL 语句或存储过程的异步执行，并从服务器中检索一个或多个结果集。

语法

Visual Basic

```
Public Function BeginExecuteReader( _
  ByVal behavior As CommandBehavior _
) As IAsyncResult
```

C#

```
public IAsyncResult BeginExecuteReader(
  CommandBehavior behavior
);
```

参数

- **behavior** 说明查询结果及其对连接的影响的 CommandBehavior 标志的逐位组合。

返回值

可只用于进行轮询、只用于等待结果或作轮询和等待结果这两种用途的 IAsyncResult；调用 EndExecuteReader(IAsyncResult)（返回可用于检索返回行的 SADataReader 对象）时也需要此值。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页
- “BeginExecuteReader 方法”一节第 205 页
- “EndExecuteReader 方法”一节第 211 页
- “SADataReader 类”一节第 297 页

BeginExecuteReader(AsyncCallback, Object) 方法

在已知回调过程和状态信息的情况下，启动由此 SACommand 对象说明的 SQL 语句的异步执行，并检索结果集。

语法

Visual Basic

```
Public Function BeginExecuteReader( _
  ByVal callback As AsyncCallback, _
  ByVal stateObject As Object _
) As IAsyncResult
```

C#

```
public IAsyncResult BeginExecuteReader(
  AsyncCallback callback,
  object stateObject
);
```

参数

- **callback** 命令执行完成时调用的 AsyncCallback 委派。传递空值（在 Microsoft Visual Basic 中是 Nothing）表示不需要回调。
- **stateObject** 传递给回调过程的用户定义状态对象。使用 IAsyncResult.AsyncState 从回调过程中检索此对象。

返回值

可只用于进行轮询、只用于等待结果或作轮询和等待结果这两种用途的 IAsyncResult；调用 EndExecuteReader(IAsyncResult)（返回可用于检索返回行的 SADataReader 对象）时也需要此值。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页
- “BeginExecuteReader 方法”一节第 205 页
- “EndExecuteReader 方法”一节第 211 页
- “SADataReader 类”一节第 297 页

BeginExecuteReader(AsyncCallback, Object, CommandBehavior) 方法

启动由此 SACommand 说明的 SQL 语句或存储过程的异步执行，并从服务器中检索一个或多个结果集。

语法

Visual Basic

```
Public Function BeginExecuteReader( _
  ByVal callback As AsyncCallback, _
  ByVal stateObject As Object, _
```

```
  ByVal behavior As CommandBehavior _
) As IAsyncResult
```

C#

```
public IAsyncResult BeginExecuteReader(
  AsyncCallback callback,
  object stateObject,
  CommandBehavior behavior
);
```

参数

- **callback** 命令执行完成时调用的 AsyncCallback 委派。传递空值（在 Microsoft Visual Basic 中是 Nothing）表示不需要回调。
- **stateObject** 传递给回调过程的用户定义状态对象。使用 IAsyncResult.AsyncState 从回调过程中检索此对象。
- **behavior** 说明查询结果及其对连接的影响的 CommandBehavior 标志的逐位组合。

返回值

可只用于进行轮询、只用于等待结果或作轮询和等待结果这两种用途的 IAsyncResult；调用 EndExecuteReader(IAsyncResult)（返回可用于检索返回行的 SADataReader 对象）时也需要此值。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页
- “BeginExecuteReader 方法”一节第 205 页
- “EndExecuteReader 方法”一节第 211 页
- “SADataReader 类”一节第 297 页

Cancel 方法

取消 SACommand 对象的执行。

语法

Visual Basic

```
Public Overrides Sub Cancel()
```

C#

```
public override void Cancel();
```

注释

如果没有要取消的操作，则不会发生任何情况。如果有一个命令正在执行，而取消尝试失败，则不会产生异常。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页

CreateParameter 方法

提供用于为 SACommand 对象提供参数的 SAParameter 对象。

语法

Visual Basic

```
Public Function CreateParameter() As SAParameter
```

C#

```
public SAParameter CreateParameter();
```

返回值

一个新的参数，作为 SAParameter 对象。

注释

存储过程和某些其它 SQL 语句可以带参数，这些参数在语句文本中以问号 (?) 表示。

CreateParameter 方法提供了一个 SAParameter 对象。您可以设置 SAParameter 上的属性来为该参数指定值、数据类型等等。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页
- “SAParameter 类”一节第 367 页

EndExecuteNonQuery 方法

完成 SQL 语句或存储过程的异步执行。

语法

Visual Basic

```
Public Function EndExecuteNonQuery( _
  ByVal asyncResult As IAsyncResult _
) As Integer
```

C#

```
public int EndExecuteNonQuery(
  IAsyncResult asyncResult
);
```

参数

- **asyncResult** 调用 SACommand.BeginExecuteNonQuery 而返回的 IAsyncResult。

返回值

受影响的行数（与 SACommand.ExecuteNonQuery 的行为相同）。

注释

每次调用 BeginExecuteNonQuery 时都必须调用一次 EndExecuteNonQuery。必须在 BeginExecuteNonQuery 返回之后执行该调用。ADO.NET 不是线程安全的；确保 BeginExecuteNonQuery 已返回是您的责任。传递给 EndExecuteNonQuery 的 IAsyncResult 必须与从正在完成的 BeginExecuteNonQuery 调用返回的 IAsyncResult 相同。通过调用 EndExecuteNonQuery 来结束对 BeginExecuteReader 的调用是错误的，反之亦然。

如果在执行命令时出现错误，则调用 EndExecuteNonQuery 时会抛出异常。

等待执行完成有四种方法：

(1) 调用 EndExecuteNonQuery。

在命令完成之前会阻止调用 EndExecuteNonQuery。例如：

```
SAConnection conn = new SAConnection("DSN=SQL Anywhere 11 Demo");
conn.Open();
SACommand cmd = new SACommand(
  "UPDATE Departments"
    + " SET DepartmentName = 'Engineering'"
    + " WHERE DepartmentID=100",
  conn );
IAsyncResult res = cmd.BeginExecuteNonQuery();
// perform other work
// this will block until the command completes
int rowCount reader = cmd.EndExecuteNonQuery( res );
```

(2) 轮询 IAsyncResult 的 IsCompleted 属性。

可以轮询 IAsyncResult 的 IsCompleted 属性。例如：

```
SAConnection conn = new SAConnection("DSN=SQL Anywhere 11 Demo");
conn.Open();
SACommand cmd = new SACommand(
  "UPDATE Departments"
    + " SET DepartmentName = 'Engineering'"
    + " WHERE DepartmentID=100",
    conn
  );
IAsyncResult res = cmd.BeginExecuteNonQuery();
while( !res.IsCompleted ) {
// do other work
}
// this will not block because the command is finished
int rowCount = cmd.EndExecuteNonQuery( res );
```

(3) 使用 IAsyncResult.AsyncWaitHandle 属性获取同步对象。

可以使用 IAsyncResult.AsyncWaitHandle 属性获取同步对象，并就此进行等待。例如：

```
SAConnection conn = new SAConnection("DSN=SQL Anywhere 11 Demo");
conn.Open();
```



```
SACommand cmd = new SACommand(
  "UPDATE Departments"
    + " SET DepartmentName = 'Engineering'"
    + " WHERE DepartmentID=100",
    conn
  );
IAsyncResult res = cmd.BeginExecuteNonQuery();
// perform other work
WaitHandle wh = res.AsyncWaitHandle;
wh.WaitOne();
// this will not block because the command is finished
int rowCount = cmd.EndExecuteNonQuery( res );
```

(4) 调用 BeginExecuteNonQuery 时指定回调函数。

可在调用 BeginExecuteNonQuery 时指定回调函数。例如：

```
private void callbackFunction( IAsyncResult ar )
{
   SACommand cmd = (SACommand) ar.AsyncState;
   // this won't block since the command has completed
            int rowCount = cmd.EndExecuteNonQuery();
}
// elsewhere in the code
private void DoStuff()
{
      SAConnection conn = new SAConnection("DSN=SQL Anywhere 11 Demo");
      conn.Open();
            SACommand cmd = new SACommand(
        "UPDATE Departments"
    + " SET DepartmentName = 'Engineering'"
    + " WHERE DepartmentID=100",
    conn
  );
   IAsyncResult res = cmd.BeginExecuteNonQuery( callbackFunction, cmd );
   // perform other work.  The callback function will be
   // called when the command completes
}
```

回调函数在单独的线程中执行，因此与在线程化程序中更新用户界面有关的常见告诫也适用。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页
- “BeginExecuteNonQuery() 方法”一节第 204 页

EndExecuteReader 方法

完成 SQL 语句或存储过程的异步执行，并返回所请求的 SADataReader。

语法

Visual Basic

```
Public Function EndExecuteReader( _
  ByVal asyncResult As IAsyncResult _
) As SADataReader
```

C#

```
public SADataReader EndExecuteReader(
  IAsyncResult asyncResult
);
```

参数

- **asyncResult** 调用 SACommand.BeginExecuteReader 而返回的 IAsyncResult。

返回值

可用于检索所请求行的 SADataReader 对象（与 SACommand.ExecuteReader 的行为相同）。

注释

每次调用 BeginExecuteReader 时都必须调用一次 EndExecuteReader。必须在 BeginExecuteReader 返回之后执行该调用。ADO.NET 不是线程安全的；确保 BeginExecuteReader 已返回是您的责任。传递给 EndExecuteReader 的 IAsyncResult 必须与从正在完成的 BeginExecuteReader 调用返回的 IAsyncResult 相同。通过调用 EndExecuteReader 来结束对 BeginExecuteNonQuery 的调用是错误的，反之亦然。

如果在执行命令时出现错误，则调用 EndExecuteReader 时会抛出异常。

等待执行完成有四种方法：

(1) 调用 EndExecuteReader。

在命令完成之前会阻止调用 EndExecuteReader。例如：

```
SAConnection conn = new SAConnection("DSN=SQL Anywhere 11 Demo");
conn.Open();
SACommand cmd = new SACommand( "SELECT * FROM Departments",
  conn );
IAsyncResult res = cmd.BeginExecuteReader();
// perform other work
// this will block until the command completes
SADataReader reader = cmd.EndExecuteReader( res );
```

(2) 轮询 IAsyncResult 的 IsCompleted 属性。

可以轮询 IAsyncResult 的 IsCompleted 属性。例如：

```
SAConnection conn = new SAConnection("DSN=SQL Anywhere 11 Demo");
conn.Open();
SACommand cmd = new SACommand( "SELECT * FROM Departments",
  conn );
IAsyncResult res = cmd.BeginExecuteReader();
while( !res.IsCompleted ) {
// do other work
}
// this will not block because the command is finished
SADataReader reader = cmd.EndExecuteReader( res );
```

(3) 使用 IAsyncResult.AsyncWaitHandle 属性获取同步对象。

可以使用 IAsyncResult.AsyncWaitHandle 属性获取同步对象，并就此进行等待。例如：

```
SAConnection conn = new SAConnection("DSN=SQL Anywhere 11 Demo");
conn.Open();
```



```
SACommand cmd = new SACommand( "SELECT * FROM Departments",
  conn );
IAsyncResult res = cmd.BeginExecuteReader();
// perform other work
WaitHandle wh = res.AsyncWaitHandle;
wh.WaitOne();
// this will not block because the command is finished
SADataReader reader = cmd.EndExecuteReader( res );
```

(4) 调用 BeginExecuteReader 时指定回调函数。

可在调用 BeginExecuteReader 时指定回调函数。例如：

```
private void callbackFunction( IAsyncResult ar )
{
   SACommand cmd = (SACommand) ar.AsyncState;
   // this won't block since the command has completed
            SADataReader reader = cmd.EndExecuteReader();
}
// elsewhere in the code
private void DoStuff()
{
      SAConnection conn = new SAConnection("DSN=SQL Anywhere 11 Demo");
      conn.Open();
            SACommand cmd = new SACommand( "SELECT * FROM Departments",
        conn );
   IAsyncResult res = cmd.BeginExecuteReader( callbackFunction, cmd );
   // perform other work.  The callback function will be
   // called when the command completes
}
```

回调函数在单独的线程中执行，因此与在线程化程序中更新用户界面有关的常见告诫也适用。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页
- “BeginExecuteReader() 方法”一节第 205 页
- “SADataReader 类”一节第 297 页

ExecuteNonQuery 方法

执行不返回结果集的语句，如 INSERT、UPDATE、DELETE 或数据定义语句。

语法

Visual Basic

Public Overrides Function **ExecuteNonQuery()** As Integer

C#

public override int **ExecuteNonQuery();**

返回值

受影响的行数。

注释

您可以不使用 DataSet，而是使用 ExecuteNonQuery 来更改数据库中的数据。通过执行 UPDATE、INSERT 或 DELETE 语句来达到这个目的。

尽管 ExecuteNonQuery 不返回任何行，系统仍会以数据填充输出参数或映射到参数的返回值。

如果是 UPDATE、INSERT 和 DELETE 语句，返回值将是受该命令影响的行数。如果是所有其它类型的语句和回退，返回值将是 -1。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页
- “ExecuteReader() 方法”一节第 214 页

ExecuteReader 方法

执行返回结果集的 SQL 语句。

ExecuteReader() 方法

执行返回结果集的 SQL 语句。

语法

Visual Basic

```
Public Function ExecuteReader() As SADataReader
```

C#

```
public SADataReader ExecuteReader();
```

返回值

作为 SADataReader 对象的结果集。

注释

该语句是当前 SACommand 对象，视需要带有 CommandText 和 Parameters。SADataReader 对象是只读的只进结果集。对于可修改的结果集，请使用 SADataAdapter。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页
- “ExecuteReader 方法”一节第 214 页
- “ExecuteNonQuery 方法”一节第 213 页
- “SADataReader 类”一节第 297 页
- “SADataAdapter 类”一节第 286 页
- “CommandText 属性”一节第 199 页
- “Parameters 属性”一节第 202 页

ExecuteReader(CommandBehavior) 方法

执行返回结果集的 SQL 语句。

语法

Visual Basic

```
Public Function ExecuteReader( _
  ByVal behavior As CommandBehavior _
) As SADataReader
```

C#

```
public SADataReader ExecuteReader(
  CommandBehavior behavior
);
```

参数

- **behavior** CloseConnection、Default、KeyInfo、SchemaOnly、SequentialAccess、SingleResult 或 SingleRow 之一。

  有关此参数的详细信息，请参见 .NET Framework 文档中的 "CommandBehavior 枚举"。

返回值

作为 SADataReader 对象的结果集。

注释

该语句是当前 SACommand 对象，视需要带有 CommandText 和 Parameters。SADataReader 对象是只读的只进结果集。对于可修改的结果集，请使用 SADataAdapter。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页
- “ExecuteReader 方法”一节第 214 页
- “ExecuteNonQuery 方法”一节第 213 页
- “SADataReader 类”一节第 297 页
- “SADataAdapter 类”一节第 286 页
- “CommandText 属性”一节第 199 页
- “Parameters 属性”一节第 202 页

ExecuteScalar 方法

执行返回单个值的语句。如果在返回多个行和列的查询上调用该方法，则只会返回第一行的第一列。

语法

Visual Basic

Public Overrides Function **ExecuteScalar()** As Object

C#

public override object **ExecuteScalar();**

返回值

结果集第一行的第一列；如果结果集为空，则为空值引用。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页

Prepare 方法

在数据源上准备或编译 SACommand。

语法

Visual Basic

Public Overrides Sub **Prepare()**

C#

public override void **Prepare();**

注释

如果您在调用 Prepare 之后调用 ExecuteNonQuery、ExecuteReader 或 ExecuteScalar 方法之一，则任何大于 Size 属性所指定值的参数值都会被自动截断为该参数的初始指定大小，并且不返回任何截断错误。

只有以下数据类型的值才会被截断：

- CHAR
- VARCHAR
- LONG VARCHAR
- TEXT
- NCHAR
- NVARCHAR
- LONG NVARCHAR
- NTEXT
- BINARY
- LONG BINARY
- VARBINARY
- IMAGE

如果因未指定 Size 属性而使用了缺省值，就不会将数据截断。

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页
- “ExecuteNonQuery 方法”一节第 213 页
- “ExecuteReader() 方法”一节第 214 页
- “ExecuteScalar 方法”一节第 216 页

ResetCommandTimeout 方法

将 CommandTimeout 属性重置为其缺省值 30 秒。

语法

Visual Basic

Public Sub **ResetCommandTimeout()**

C#

public void **ResetCommandTimeout();**

另请参见

- “SACommand 类”一节第 195 页
- “SACommand 成员”一节第 196 页

SACommandBuilder 类

一种生成单表 SQL 语句的方法，这些语句会使对 DataSet 进行的更改与关联数据库中的数据一致。此类无法继承。

语法

Visual Basic

```
Public NotInheritable Class SACommandBuilder
  Inherits DbCommandBuilder
```

C#

```
public sealed class SACommandBuilder : DbCommandBuilder
```

另请参见

- “SACommandBuilder 成员”一节第 218 页

SACommandBuilder 成员

公共构造函数

成员名称	说明
SACommandBuilder 构造函数	初始化一个新的“SACommandBuilder 类”一节第 218 页实例。

公共属性

成员名称	说明
CatalogLocation（继承自 DbCommandBuilder）	设置或获取 DbCommandBuilder 实例的 CatalogLocation。
CatalogSeparator（继承自 DbCommandBuilder）	设置或获取用作 DbCommandBuilder 实例的目录分隔符的字符串。
ConflictOption（继承自 DbCommandBuilder）	指定 DbCommandBuilder 即将使用哪一个 ConflictOption。
DataAdapter 属性	指定要为其生成语句的 SADataAdapter。
QuotePrefix（继承自 DbCommandBuilder）	获取或设置要在指定名称包含空格或保留标识之类字符的数据库对象（例如，表或列）时使用的一个或多个起始字符。
QuoteSuffix（继承自 DbCommandBuilder）	获取或设置要在指定名称包含空格或保留标识之类字符的数据库对象（例如，表或列）时使用的一个或多个起始字符。

成员名称	说明
SchemaSeparator（继承自 DbCommandBuilder）	获取和设置要用作模式标识符与其它任何标识符之间分隔符的字符。
SetAllValues（继承自 DbCommandBuilder）	指定在更新语句中是包含所有列值，还是只包含更改的列值。

公共方法

成员名称	说明
DeriveParameters 方法	填充指定 SACommand 对象的 Parameters 集合。它用于 SACommand 中指定的存储过程。
GetDeleteCommand 方法	返回调用 SADataAdapter.Update 时生成的、对数据库执行 DELETE 操作的 SACommand 对象。
GetInsertCommand 方法	返回调用 Update 时生成的、对数据库执行 INSERT 操作的 SACommand 对象。
GetUpdateCommand 方法	返回调用 Update 时生成的、对数据库执行 UPDATE 操作的 SACommand 对象。
QuoteIdentifier 方法	返回不带引号标识符的正确加引号形式，该标识符中的所有嵌入式引号都已经过正确转义。
RefreshSchema（继承自 DbCommandBuilder）	清除与此 DbCommandBuilder 关联的命令。
UnquoteIdentifier 方法	返回带引号标识符的正确的不加引号形式，该标识符中的所有嵌入式引号都已经过正确的取消转义。

另请参见

- “SACommandBuilder 类”一节第 218 页

SACommandBuilder 构造函数

初始化一个新的“SACommandBuilder 类”一节第 218 页实例。

SACommandBuilder() 构造函数

初始化 SACommandBuilder 对象。

语法

Visual Basic

```
Public Sub New()
```

C#

```
public  SACommandBuilder();
```

另请参见

- “SACommandBuilder 类”一节第 218 页
- “SACommandBuilder 成员”一节第 218 页
- “SACommandBuilder 构造函数”一节第 219 页

SACommandBuilder(SADataAdapter) 构造函数

初始化 SACommandBuilder 对象。

语法

Visual Basic

```
Public Sub New( _
  ByVal adapter As SADataAdapter _
)
```

C#

```
public  SACommandBuilder(
  SADataAdapter adapter
);
```

参数

- **adapter**　要为其生成一致性语句的 SADataAdapter 对象。

另请参见

- “SACommandBuilder 类”一节第 218 页
- “SACommandBuilder 成员”一节第 218 页
- “SACommandBuilder 构造函数”一节第 219 页

DataAdapter 属性

指定要为其生成语句的 SADataAdapter。

语法

Visual Basic

```
Public Property DataAdapter As SADataAdapter
```


C#

```
public SADataAdapter DataAdapter { get; set; }
```

属性值

一种 SADataAdapter 对象。

注释

创建新的 SACommandBuilder 实例时，将释放所有与此 SADataAdapter 关联的现有 SACommandBuilder。

另请参见

- “SACommandBuilder 类”一节第 218 页
- “SACommandBuilder 成员”一节第 218 页

DeriveParameters 方法

填充指定 SACommand 对象的 Parameters 集合。它用于 SACommand 中指定的存储过程。

语法

Visual Basic

```
Public Shared Sub DeriveParameters( _
  ByVal command As SACommand _
)
```

C#

```
public static void DeriveParameters(
  SACommand command
);
```

参数

- **command** 要为其派生参数的 SACommand 对象。

注释

DeriveParameters 会覆盖 SACommand 的任何现有参数信息。

DeriveParameters 要求对数据库服务器进行额外调用。如果事先知道参数信息，则通过显式地设置该信息来填充 Parameters 集合的做法会更有效率。

另请参见

- “SACommandBuilder 类”一节第 218 页
- “SACommandBuilder 成员”一节第 218 页

GetDeleteCommand 方法

返回调用 SADataAdapter.Update 时生成的、对数据库执行 DELETE 操作的 SACommand 对象。

GetDeleteCommand(Boolean) 方法

返回调用 SADataAdapter.Update 时生成的、对数据库执行 DELETE 操作的 SACommand 对象。

语法

Visual Basic

```
Public Function GetDeleteCommand( _
  ByVal useColumnsForParameterNames As Boolean _
) As SACommand
```

C#

```
public SACommand GetDeleteCommand(
  bool useColumnsForParameterNames
);
```

参数

- **useColumnsForParameterNames** 如果为 true，则在可能的情况下生成与列名匹配的参数名。如果为 false，则生成 @p1、@p2 等等。

返回值

自动生成的、执行删除操作所需的 SACommand 对象。

注释

GetDeleteCommand 方法返回要执行的 SACommand 对象，因此它在提供信息或进行疑难解答上可能会有帮助。

也可以将 GetDeleteCommand 用作修改过的命令的基础。例如，您可能会调用 GetDeleteCommand 并修改 CommandTimeout 值，然后在 SADataAdapter 上显式地设置该值。

当应用程序调用 Update 或 GetDeleteCommand 时，会首先生成 SQL 语句。首先生成 SQL 语句后，如果应用程序以任何方式更改了该语句，则其必须显式地调用 RefreshSchema。否则，GetDeleteCommand 将仍然使用来自上一条语句的信息。

另请参见

- “SACommandBuilder 类”一节第 218 页
- “SACommandBuilder 成员”一节第 218 页
- “GetDeleteCommand 方法”一节第 222 页
- DbCommandBuilder.RefreshSchema

GetDeleteCommand() 方法

返回调用 SADataAdapter.Update 时生成的、对数据库执行 DELETE 操作的 SACommand 对象。

语法

Visual Basic

```
Public Function GetDeleteCommand() As SACommand
```

C#

```
public SACommand GetDeleteCommand();
```

返回值

自动生成的、执行删除操作所需的 SACommand 对象。

注释

GetDeleteCommand 方法返回要执行的 SACommand 对象，因此它在提供信息或进行疑难解答上可能会有帮助。

也可以将 GetDeleteCommand 用作修改过的命令的基础。例如，您可能会调用 GetDeleteCommand 并修改 CommandTimeout 值，然后在 SADataAdapter 上显式地设置该值。

当应用程序调用 Update 或 GetDeleteCommand 时，会首先生成 SQL 语句。首先生成 SQL 语句后，如果应用程序以任何方式更改了该语句，则其必须显式地调用 RefreshSchema。否则，GetDeleteCommand 将仍然使用来自上一条语句的信息。

另请参见

- “SACommandBuilder 类”一节第 218 页
- “SACommandBuilder 成员”一节第 218 页
- “GetDeleteCommand 方法”一节第 222 页
- DbCommandBuilder.RefreshSchema

GetInsertCommand 方法

返回调用 Update 时生成的、对数据库执行 INSERT 操作的 SACommand 对象。

GetInsertCommand(Boolean) 方法

返回调用 Update 时生成的、对数据库执行 INSERT 操作的 SACommand 对象。

语法

Visual Basic

```
Public Function GetInsertCommand( _
  ByVal useColumnsForParameterNames As Boolean _
) As SACommand
```

C#

```
public SACommand GetInsertCommand(
  bool useColumnsForParameterNames
);
```

参数

- **useColumnsForParameterNames** 如果为 true，则在可能的情况下生成与列名匹配的参数名。如果为 false，则生成 @p1、@p2 等等。

返回值

自动生成的、执行插入操作所需的 SACommand 对象。

注释

GetInsertCommand 方法返回要执行的 SACommand 对象，因此它在提供信息或进行疑难解答上可能会有帮助。

也可以将 GetInsertCommand 用作修改过的命令的基础。例如，您可能会调用 GetInsertCommand 并修改 CommandTimeout 值，然后在 SADataAdapter 上显式地设置该值。

当应用程序调用 Update 或 GetInsertCommand 时，将首先生成 SQL 语句。首先生成 SQL 语句后，如果应用程序以任何方式更改了该语句，则其必须显式地调用 RefreshSchema。否则，GetInsertCommand 将仍然使用来自上一条语句的信息，而这些信息可能是不正确的。

另请参见

- “SACommandBuilder 类”一节第 218 页
- “SACommandBuilder 成员”一节第 218 页
- “GetInsertCommand 方法”一节第 223 页
- “GetDeleteCommand() 方法”一节第 223 页

GetInsertCommand() 方法

返回调用 Update 时生成的、对数据库执行 INSERT 操作的 SACommand 对象。

语法

Visual Basic

```
Public Function GetInsertCommand() As SACommand
```

C#

```
public SACommand GetInsertCommand();
```

返回值

自动生成的、执行插入操作所需的 SACommand 对象。

注释

GetInsertCommand 方法返回要执行的 SACommand 对象，因此它在提供信息或进行疑难解答上可能会有帮助。

也可以将 GetInsertCommand 用作修改过的命令的基础。例如，您可能会调用 GetInsertCommand 并修改 CommandTimeout 值，然后在 SADataAdapter 上显式地设置该值。

当应用程序调用 Update 或 GetInsertCommand 时，将首先生成 SQL 语句。首先生成 SQL 语句后，如果应用程序以任何方式更改了该语句，则其必须显式地调用 RefreshSchema。否则，GetInsertCommand 将仍然使用来自上一条语句的信息，而这些信息可能是不正确的。

另请参见

- “SACommandBuilder 类”一节第 218 页
- “SACommandBuilder 成员”一节第 218 页
- “GetInsertCommand 方法”一节第 223 页
- “GetDeleteCommand() 方法”一节第 223 页

GetUpdateCommand 方法

返回调用 Update 时生成的、对数据库执行 UPDATE 操作的 SACommand 对象。

GetUpdateCommand(Boolean) 方法

返回调用 Update 时生成的、对数据库执行 UPDATE 操作的 SACommand 对象。

语法

Visual Basic

```
Public Function GetUpdateCommand( _
  ByVal useColumnsForParameterNames As Boolean _
) As SACommand
```

C#

```
public SACommand GetUpdateCommand(
  bool useColumnsForParameterNames
);
```

参数

- **useColumnsForParameterNames** 如果为 true，则在可能的情况下生成与列名匹配的参数名。如果为 false，则生成 @p1、@p2 等等。

返回值

自动生成的、执行更新操作所需的 SACommand 对象。

注释

GetUpdateCommand 方法返回要执行的 SACommand 对象，因此它在提供信息或进行疑难解答上可能会有帮助。

也可以将 GetUpdateCommand 用作修改过的命令的基础。例如，您可能会调用 GetUpdateCommand 并修改 CommandTimeout 值，然后在 SADataAdapter 上显式地设置该值。

当应用程序调用 Update 或 GetUpdateCommand 时，将首先生成 SQL 语句。首先生成 SQL 语句后，如果应用程序以任何方式更改了该语句，则其必须显式地调用 RefreshSchema。否则，GetUpdateCommand 将仍然使用来自上一条语句的信息，而这些信息可能是不正确的。

另请参见

- “SACommandBuilder 类”一节第 218 页
- “SACommandBuilder 成员”一节第 218 页
- “GetUpdateCommand 方法”一节第 225 页
- DbCommandBuilder.RefreshSchema

GetUpdateCommand() 方法

返回调用 Update 时生成的、对数据库执行 UPDATE 操作的 SACommand 对象。

语法

Visual Basic

```
Public Function GetUpdateCommand() As SACommand
```

C#

```
public SACommand GetUpdateCommand();
```

返回值

自动生成的、执行更新操作所需的 SACommand 对象。

注释

GetUpdateCommand 方法返回要执行的 SACommand 对象，因此它在提供信息或进行疑难解答上可能会有帮助。

也可以将 GetUpdateCommand 用作修改过的命令的基础。例如，您可能会调用 GetUpdateCommand 并修改 CommandTimeout 值，然后在 SADataAdapter 上显式地设置该值。

当应用程序调用 Update 或 GetUpdateCommand 时，将首先生成 SQL 语句。首先生成 SQL 语句后，如果应用程序以任何方式更改了该语句，则其必须显式地调用 RefreshSchema。否则，GetUpdateCommand 将仍然使用来自上一条语句的信息，而这些信息可能是不正确的。

另请参见

- “SACommandBuilder 类”一节第 218 页
- “SACommandBuilder 成员”一节第 218 页
- “GetUpdateCommand 方法”一节第 225 页
- DbCommandBuilder.RefreshSchema

QuoteIdentifier 方法

返回不带引号标识符的正确加引号形式，该标识符中的所有嵌入式引号都已经过正确转义。

语法

Visual Basic

```
Public Overrides Function QuoteIdentifier( _
   ByVal unquotedIdentifier As String _
) As String
```

C#

```
public override string  QuoteIdentifier(
   string  unquotedIdentifier
);
```

参数

- **unquotedIdentifier**　表示不带引号标识符的字符串，该字符串将被加上引号。

返回值

返回一个字符串，该字符串表示已对嵌入式引号正确转义的不带引号标识符的加引号形式。

另请参见

- “SACommandBuilder 类”一节第 218 页
- “SACommandBuilder 成员”一节第 218 页

UnquoteIdentifier 方法

返回带引号标识符的正确的不加引号形式，该标识符中的所有嵌入式引号都已经过正确的取消转义。

语法

Visual Basic

```
Public Overrides Function UnquoteIdentifier( _
   ByVal quotedIdentifier As String _
) As String
```

C#

```
public override string  UnquoteIdentifier(
   string  quotedIdentifier
);
```

参数

- **quotedIdentifier**　表示会删除带引号标识符的嵌入式引号的字符串。

返回值

返回字符串，此字符串表示已对嵌入式引号正确取消转义的带引号标识符的不加引号形式。

另请参见

- “SACommandBuilder 类”一节第 218 页
- “SACommandBuilder 成员”一节第 218 页

SACommLinksOptionsBuilder 类

为创建和管理 SAConnection 类所使用的连接字符串的 CommLinks 选项部分提供了一种简单的方法。此类无法继承。

语法

Visual Basic

Public NotInheritable Class **SACommLinksOptionsBuilder**

C#

public sealed class **SACommLinksOptionsBuilder**

注释

无法在 .NET Compact Framework 2.0 中使用 SACommLinksOptionsBuilder 类。

有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》 。

另请参见

- “SACommLinksOptionsBuilder 成员”一节第 228 页

SACommLinksOptionsBuilder 成员

公共构造函数

成员名称	说明
SACommLinksOptionsBuilder 构造函数	初始化一个新的“SACommLinksOptionsBuilder 类”一节第 228 页实例。

公共属性

成员名称	说明
All 属性	获取或设置 ALL CommLinks 选项。
ConnectionString 属性	获取或设置正在构建的连接字符串。
SharedMemory 属性	获取或设置 SharedMemory 协议。
TcpOptionsBuilder 属性	获取或设置用于创建 TCP 选项字符串的 TcpOptionsBuilder 对象。
TcpOptionsString 属性	获取或设置 TCP 选项的字符串。

公共方法

成员名称	说明
GetUseLongNameAsKeyword 方法	获取表示是否可以在连接字符串中使用长连接参数名的布尔值。
SetUseLongNameAsKeyword 方法	设置表示连接字符串中是否使用长连接参数名的布尔值。缺省情况下使用长连接参数名。
ToString 方法	将 SACommLinksOptionsBuilder 对象转换为字符串表示。

另请参见

- “SACommLinksOptionsBuilder 类”一节第 228 页

SACommLinksOptionsBuilder 构造函数

初始化一个新的“SACommLinksOptionsBuilder 类”一节第 228 页实例。

SACommLinksOptionsBuilder() 构造函数

初始化 SACommLinksOptionsBuilder 对象。

语法

Visual Basic

```
Public Sub New()
```

C#

```
public SACommLinksOptionsBuilder();
```

注释

无法在 .NET Compact Framework 2.0 中使用 SACommLinksOptionsBuilder 类。

示例

以下语句初始化 SACommLinksOptionsBuilder 对象。

```
SACommLinksOptionsBuilder commLinks =
 new SACommLinksOptionsBuilder( );
```

另请参见

- “SACommLinksOptionsBuilder 类”一节第 228 页
- “SACommLinksOptionsBuilder 成员”一节第 228 页
- “SACommLinksOptionsBuilder 构造函数”一节第 229 页

SACommLinksOptionsBuilder(String) 构造函数

初始化 SACommLinksOptionsBuilder 对象。

语法

Visual Basic

```
Public Sub New( _
  ByVal options As String _
)
```

C#

```
public  SACommLinksOptionsBuilder(
  string  options
);
```

参数

- **options**　一个 SQL Anywhere CommLinks 连接参数字符串。

  有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

注释

无法在 .NET Compact Framework 2.0 中使用 SACommLinksOptionsBuilder 类。

示例

以下语句初始化 SACommLinksOptionsBuilder 对象。

```
SACommLinksOptionsBuilder commLinks =
 new SACommLinksOptionsBuilder("TCPIP(DoBroadcast=ALL;Timeout=20)" );
```


另请参见

- “SACommLinksOptionsBuilder 类”一节第 228 页
- “SACommLinksOptionsBuilder 成员”一节第 228 页
- “SACommLinksOptionsBuilder 构造函数”一节第 229 页

All 属性

获取或设置 ALL CommLinks 选项。

语法

Visual Basic

Public Property **All** As Boolean

C#

public bool **All** { get; set; }

注释

首先尝试使用共享内存协议进行连接，然后使用所有其余的可用通信协议进行连接。如果无法确定使用哪个（些）协议，则使用此设置。

无法在 .NET Compact Framework 2.0 中使用 SACommLinksOptionsBuilder 类。

另请参见

- “SACommLinksOptionsBuilder 类”一节第 228 页
- “SACommLinksOptionsBuilder 成员”一节第 228 页

ConnectionString 属性

获取或设置正在构建的连接字符串。

语法

Visual Basic

Public Property **ConnectionString** As String

C#

public string **ConnectionString** { get; set; }

注释

无法在 .NET Compact Framework 2.0 中使用 SACommLinksOptionsBuilder 类。

另请参见

- “SACommLinksOptionsBuilder 类”一节第 228 页
- “SACommLinksOptionsBuilder 成员”一节第 228 页

SharedMemory 属性

获取或设置 SharedMemory 协议。

语法

Visual Basic

Public Property **SharedMemory** As Boolean

C#

public bool **SharedMemory** { get; set; }

注释

无法在 .NET Compact Framework 2.0 中使用 SACommLinksOptionsBuilder 类。

另请参见

- “SACommLinksOptionsBuilder 类”一节第 228 页
- “SACommLinksOptionsBuilder 成员”一节第 228 页

TcpOptionsBuilder 属性

获取或设置用于创建 TCP 选项字符串的 TcpOptionsBuilder 对象。

语法

Visual Basic

Public Property **TcpOptionsBuilder** As SATcpOptionsBuilder

C#

public SATcpOptionsBuilder **TcpOptionsBuilder** { get; set; }

另请参见

- “SACommLinksOptionsBuilder 类”一节第 228 页
- “SACommLinksOptionsBuilder 成员”一节第 228 页

TcpOptionsString 属性

获取或设置 TCP 选项的字符串。

语法

Visual Basic

Public Property **TcpOptionsString** As String

C#

public string **TcpOptionsString** { get; set; }

另请参见

- “SACommLinksOptionsBuilder 类”一节第 228 页
- “SACommLinksOptionsBuilder 成员”一节第 228 页

GetUseLongNameAsKeyword 方法

获取表示是否可以在连接字符串中使用长连接参数名的布尔值。

语法

Visual Basic

Public Function **GetUseLongNameAsKeyword()** As Boolean

C#

public bool **GetUseLongNameAsKeyword();**

返回值

如果使用长连接参数名来生成连接字符串，则为 true；否则为 false。

注释

SQL Anywhere 连接参数的名称有长、短两种形式。例如，要在连接字符串中指定 ODBC 数据源的名称，可以使用以下两个值之一：DataSourceName 或 DSN。缺省情况下使用长连接参数名来生成连接字符串。

另请参见

- “SACommLinksOptionsBuilder 类”一节第 228 页
- “SACommLinksOptionsBuilder 成员”一节第 228 页
- “SetUseLongNameAsKeyword 方法”一节第 233 页

SetUseLongNameAsKeyword 方法

设置表示连接字符串中是否使用长连接参数名的布尔值。缺省情况下使用长连接参数名。

语法

Visual Basic

```
Public Sub SetUseLongNameAsKeyword( _
  ByVal useLongNameAsKeyword As Boolean _
)
```

C#

```
public void SetUseLongNameAsKeyword(
  bool useLongNameAsKeyword
);
```

参数

- **useLongNameAsKeyword** 表示连接字符串中是否使用长连接参数名的布尔值。

另请参见

- “SACommLinksOptionsBuilder 类”一节第 228 页
- “SACommLinksOptionsBuilder 成员”一节第 228 页
- “GetUseLongNameAsKeyword 方法”一节第 233 页

ToString 方法

将 SACommLinksOptionsBuilder 对象转换为字符串表示。

语法

Visual Basic

```
Public Overrides Function ToString() As String
```

C#

```
public override string  ToString();
```

返回值

正在构建的选项字符串。

另请参见

- “SACommLinksOptionsBuilder 类”一节第 228 页
- “SACommLinksOptionsBuilder 成员”一节第 228 页

SAConnection 类

表示与 SQL Anywhere 数据库的连接。此类无法继承。

语法

Visual Basic

```
Public NotInheritable Class SAConnection
  Inherits DbConnection
```


C#

public sealed class **SAConnection** : DbConnection

注释

有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

另请参见

- “SAConnection 成员”一节第 235 页

SAConnection 成员

公共构造函数

成员名称	说明
SAConnection 构造函数	初始化一个新的“SAConnection 类”一节第 234 页实例。

公共属性

成员名称	说明
ConnectionString 属性	提供数据库连接字符串。
ConnectionTimeout 属性	获取连接尝试超时并产生错误前等待的秒数。
DataSource 属性	获取数据库服务器的名称。
Database 属性	获取当前数据库的名称。
InitString 属性	连接建立后立即执行的命令。
ServerVersion 属性	获取包含客户端连接到的 SQL Anywhere 实例版本的字符串。
State 属性	表示 SAConnection 对象的状态。

公共方法

成员名称	说明
BeginTransaction 方法	返回事务对象。将与事务对象关联的命令作为单个事务执行。通过调用 Commit 或 Rollback 方法终止事务。
ChangeDatabase 方法	更改打开的 SAConnection 的当前数据库。
ChangePassword 方法	将连接字符串中指示的用户口令更改为所提供的新口令。

成员名称	说明
ClearAllPools 方法	清空所有连接池。
ClearPool 方法	清空与指定连接关联的连接池。
Close 方法	关闭数据库连接。
CreateCommand 方法	初始化 SACommand 对象。
EnlistDistributedTransaction 方法	在指定事务中以分布式事务形式征用。
EnlistTransaction 方法	在指定事务中以分布式事务形式征用。
GetSchema 方法	返回所支持模式集合的列表。
Open 方法	使用由 SAConnection.ConnectionString 指定的属性设置打开数据库连接。

公共事件

成员名称	说明
InfoMessage 事件	SQL Anywhere 数据库服务器返回警告或信息性消息时发生。
StateChange 事件	SAConnection 对象状态变化时发生。

另请参见

- “SAConnection 类”一节第 234 页

SAConnection 构造函数

初始化一个新的“SAConnection 类”一节第 234 页实例。

SAConnection() 构造函数

初始化 SAConnection 对象。对数据库进行任何操作前必须打开该连接。

语法

Visual Basic

Public Sub **New()**

C#

```
public SAConnection();
```

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页
- “SAConnection 构造函数”一节第 236 页

SAConnection(String) 构造函数

初始化 SAConnection 对象。随后必须打开该连接，才能对数据库执行任何操作。

语法

Visual Basic

```
Public Sub New( _
  ByVal connectionString As String _
)
```

C#

```
public SAConnection(
  string connectionString
);
```

参数

- **connectionString** 一个 SQL Anywhere 连接字符串。连接字符串是以分号分隔的 "关键字=值" 对的列表。

  有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

示例

以下语句为到名为 policies 的数据库的连接初始化一个 SAConnection 对象，该数据库在名为 hr 的 SQL Anywhere 数据库服务器上运行。该连接使用的用户 ID 为 admin，口令为 money。

```
SAConnection conn = new SAConnection(
"UID=admin;PWD=money;ENG=hr;DBN=policies" );
conn.Open();
```

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页
- “SAConnection 构造函数”一节第 236 页
- “SAConnection 类”一节第 234 页

ConnectionString 属性

提供数据库连接字符串。

语法

Visual Basic

Public Overrides Property **ConnectionString** As String

C#

public override string **ConnectionString** { get; set; }

注释

ConnectionString 设计为尽可能与 SQL Anywhere 连接字符串的格式匹配，但有以下例外情况：Persist Security Info 值设置为 false（缺省值）时，返回的连接字符串与用户设置的 ConnectionString 去除安全性信息后的内容相同。除非将 Persist Security Info 设置为 true，否则 SQL Anywhere SQL Anywhere .NET 数据提供程序不会在返回的连接字符串中持久性地保存口令。

使用 ConnectionString 属性可以连接到多种类型的数据源。

只有关闭连接时才能设置 ConnectionString 属性。许多连接字符串值都有相应的只读属性。设置连接字符串后，将会更新所有这些属性，除非检测到错误。如果检测到错误，则不会更新任何属性。SAConnection 属性仅返回 ConnectionString 中包含的那些设置。

如果在关闭的连接上重置 ConnectionString，则所有连接字符串值和相关属性（包括口令）都会被重置。

设置该属性时会对连接字符串执行预备校验。应用程序调用 Open 方法时会完全校验连接字符串。如果连接字符串包含无效或不受支持的属性，则会产生运行时异常。

值可用单引号或双引号进行分隔。可以在连接字符串内使用单引号或双引号，但前提是使用某一种引号时须使用另一种引号作为分隔符。例如，name="value's" 或 name= 'value"s' 是允许的，而 name='value's' 或 name= ""value"" 则不允许。除非空白字符置于值或引号内，否则将被忽略。"关键字=值" 对必须以分号分隔。作为值一部分的分号必须以引号分隔。不支持转义序列，该值类型也没有实用性。名称不区分大小写。如果属性名在连接字符串中出现多次，将使用其最后一次出现时所关联的值。

基于用户输入构建连接字符串（如从窗口检索用户 ID 和口令并将其附加到连接字符串）时应该小心谨慎。应用程序不应允许用户将额外的连接字符串参数嵌入到这些值中。

连接池的缺省值为 true (pooling=true)。

示例

以下语句为名为 SQL Anywhere 11 Demo 的 ODBC 数据源设置连接字符串并打开该连接。

```
SAConnection conn = new SAConnection();
conn.ConnectionString = "DSN=SQL Anywhere 11 Demo";
conn.Open();
```


另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页
- “SAConnection 类”一节第 234 页
- “Open 方法”一节第 255 页

ConnectionTimeout 属性

获取连接尝试超时并产生错误前等待的秒数。

语法

Visual Basic

Public Overrides Readonly Property **ConnectionTimeout** As Integer

C#

public override int **ConnectionTimeout** { get;}

属性值

15 秒

示例

以下语句显示 ConnectionTimeout 的值。

```
MessageBox.Show( conn.ConnectionTimeout.ToString( ) );
```

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页

DataSource 属性

获取数据库服务器的名称。

语法

Visual Basic

Public Overrides Readonly Property **DataSource** As String

C#

public override string **DataSource** { get;}

注释

如果打开了连接，则 SAConnection 对象返回 ServerName 服务器属性。否则，SAConnection 对象将按以下顺序在连接字符串中查找：EngineName、ServerName、ENG。

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页
- “SAConnection 类”一节第 234 页

Database 属性

获取当前数据库的名称。

语法

Visual Basic

```
Public Overrides Readonly Property Database As String
```

C#

```
public override string  Database { get;}
```

注释

如果打开了连接，则 SAConnection 返回当前数据库的名称。否则，SAConnection 将按以下顺序在连接字符串中查找：DatabaseName、DBN、DataSourceName、DataSource、DSN、DatabaseFile、DBF。

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页

InitString 属性

连接建立后立即执行的命令。

语法

Visual Basic

```
Public Property InitString As String
```

C#

```
public string  InitString { get; set; }
```


注释

连接打开后将立即执行 InitString。

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页

ServerVersion 属性

获取包含客户端连接到的 SQL Anywhere 实例版本的字符串。

语法

Visual Basic

```
Public Overrides Readonly Property ServerVersion As String
```

C#

```
public override string  ServerVersion { get;}
```

属性值

SQL Anywhere 实例的版本。

注释

版本是 ##.##.####，其中前两位是主要版本，接下来的两位是次要版本，最后四位是发行版本。附加字符串的格式为 major.minor.build，其中 major 和 minor 为两位数字，build 为四位数字。

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页

State 属性

表示 SAConnection 对象的状态。

语法

Visual Basic

```
Public Overrides Readonly Property State As ConnectionState
```

C#

```
public override ConnectionState State { get;}
```

属性值

一种 ConnectionState 枚举。

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页

BeginTransaction 方法

返回事务对象。将与事务对象关联的命令作为单个事务执行。通过调用 Commit 或 Rollback 方法终止事务。

BeginTransaction() 方法

返回事务对象。将与事务对象关联的命令作为单个事务执行。通过调用 Commit 或 Rollback 方法终止事务。

语法

Visual Basic

Public Function **BeginTransaction()** As SATransaction

C#

public SATransaction **BeginTransaction();**

返回值

表示新事务的 SATransaction 对象。

注释

若要将命令与事务对象关联，请使用 SACommand.Transaction 属性。

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页
- “BeginTransaction 方法”一节第 242 页
- “SATransaction 类”一节第 424 页
- “Transaction 属性”一节第 203 页

BeginTransaction(IsolationLevel) 方法

返回事务对象。将与事务对象关联的命令作为单个事务执行。通过调用 Commit 或 Rollback 方法终止事务。

语法

Visual Basic

```
Public Function BeginTransaction( _
  ByVal isolationLevel As IsolationLevel _
) As SATransaction
```

C#

```
public SATransaction BeginTransaction(
  IsolationLevel isolationLevel
);
```

参数

- **isolationLevel** SAIsolationLevel 枚举的一个成员。缺省值为 ReadCommitted。

返回值

表示新事务的 SATransaction 对象。

注释

若要将命令与事务对象关联，请使用 SACommand.Transaction 属性。

示例

```
SATransaction tx = conn.BeginTransaction(
    SAIsolationLevel.ReadUncommitted );
```

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页
- “BeginTransaction 方法”一节第 242 页
- “SATransaction 类”一节第 424 页
- “Transaction 属性”一节第 203 页
- “SAIsolationLevel 枚举”一节第 355 页

BeginTransaction(SAIsolationLevel) 方法

返回事务对象。将与事务对象关联的命令作为单个事务执行。通过调用 Commit 或 Rollback 方法终止事务。

语法

Visual Basic

```
Public Function BeginTransaction( _
  ByVal isolationLevel As SAIsolationLevel _
) As SATransaction
```

C#

```
public SATransaction BeginTransaction(
  SAIsolationLevel isolationLevel
);
```

参数

- **isolationLevel** SAIsolationLevel 枚举的一个成员。缺省值为 ReadCommitted。

返回值

表示新事务的 SATransaction 对象。

有关详细信息，请参见“事务处理”一节第 129 页。

有关详细信息，请参见“典型的不一致类型”一节 《SQL Anywhere 服务器 - SQL 的用法》。

注释

若要将命令与事务对象关联，请使用 SACommand.Transaction 属性。

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页
- “BeginTransaction 方法”一节第 242 页
- “SATransaction 类”一节第 424 页
- “Transaction 属性”一节第 203 页
- “SAIsolationLevel 枚举”一节第 355 页
- “Commit 方法”一节第 427 页
- “Rollback() 方法”一节第 428 页
- “Rollback(String) 方法”一节第 428 页

ChangeDatabase 方法

更改打开的 SAConnection 的当前数据库。

语法

Visual Basic

```
Public Overrides Sub ChangeDatabase( _
  ByVal database As String _
)
```

C#

```
public override void ChangeDatabase(
  string  database
);
```

参数

- **database** 要使用的数据库而非当前数据库的名称。

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页

ChangePassword 方法

将连接字符串中指示的用户口令更改为所提供的新口令。

语法

Visual Basic

```
Public Shared Sub ChangePassword( _
  ByVal connectionString As String, _
  ByVal newPassword As String _
)
```

C#

```
public static void ChangePassword(
  string  connectionString,
  string  newPassword
);
```

参数

- **connectionString** 包含连接至想要连接的数据库服务器所需信息的连接字符串。连接字符串可以包含用户 ID 和当前口令。
- **newPassword** 要设置的新口令。此口令必须遵守服务器上设置的任何口令安全策略，包括最小长度、对特殊字符的要求等等。

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页

ClearAllPools 方法

清空所有连接池。

语法

Visual Basic

```
Public Shared Sub ClearAllPools()
```

C#

```
public static void ClearAllPools();
```

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页

ClearPool 方法

清空与指定连接关联的连接池。

语法

Visual Basic

```
Public Shared Sub ClearPool( _
  ByVal connection As SAConnection _
)
```

C#

```
public static void ClearPool(
  SAConnection connection
);
```

参数

- **connection** 要从池中清除的 SAConnection 对象。

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页
- “SAConnection 类”一节第 234 页

Close 方法

关闭数据库连接。

语法

Visual Basic

```
Public Overrides Sub Close()
```

C#

```
public override void Close();
```

注释

Close 方法会回退所有待执行的事务。然后它会将连接释放到连接池，如果连接池被禁用，则关闭连接。如果处理 StateChange 事件时调用 Close，则不会触发任何其它 StateChange 事件。应用程序可以多次调用 Close。

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页

CreateCommand 方法

初始化 SACommand 对象。

语法

Visual Basic

```
Public Function CreateCommand() As SACommand
```

C#

```
public SACommand CreateCommand();
```

返回值

SACommand 对象。

注释

命令对象与 SAConnection 对象关联。

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页
- “SACommand 类”一节第 195 页
- “SAConnection 类”一节第 234 页

EnlistDistributedTransaction 方法

在指定事务中以分布式事务形式征用。

语法

Visual Basic

```
Public Sub EnlistDistributedTransaction( _
  ByVal transaction As ITransaction _
)
```

C#

```
public void EnlistDistributedTransaction(
  ITransaction transaction
);
```

参数

- **transaction** 对要在其中征用的现有 System.EnterpriseServices.ITransaction 的引用。

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页

EnlistTransaction 方法

在指定事务中以分布式事务形式征用。

语法

Visual Basic

```
Public Overrides Sub EnlistTransaction( _
  ByVal transaction As Transaction _
)
```

C#

```
public override void EnlistTransaction(
  Transaction transaction
);
```

参数

- **transaction** 对要在其中征用的现有 System.Transactions.Transaction 的引用。

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页

GetSchema 方法

返回所支持模式集合的列表。

GetSchema() 方法

返回所支持模式集合的列表。

语法

Visual Basic

```
Public Overrides Function GetSchema() As DataTable
```


C#

```
public override DataTable GetSchema();
```

注释

有关可用元数据的说明，请参见 GetSchema(string,string[])。

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页
- “GetSchema 方法”一节第 248 页
- “GetSchema(String, String[]) 方法”一节第 250 页

GetSchema(String) 方法

返回此 SAConnection 对象的指定元数据集合的相关信息。

语法

Visual Basic

```
Public Overrides Function GetSchema( _
  ByVal collection As String _
) As DataTable
```

C#

```
public override DataTable GetSchema(
  string  collection
);
```

参数

- **collection**　元数据集合的名称。如果未提供名称，则使用 MetaDataCollections。

注释

有关可用元数据的说明，请参见 GetSchema(string,string[])。

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页
- “GetSchema 方法”一节第 248 页
- “GetSchema(String, String[]) 方法”一节第 250 页
- “SAConnection 类”一节第 234 页

GetSchema(String, String[]) 方法

返回此 SAConnection 对象的数据源模式信息，并且如果已指定，则使用指定的模式名称字符串和指定的限制值字符串数组。

语法

Visual Basic

```
Public Overrides Function GetSchema( _
  ByVal collection As String, _
  ByVal restrictions As String() _
) As DataTable
```

C#

```
public override DataTable GetSchema(
  string  collection,
  string [] restrictions
);
```

返回值

包含模式信息的 DataTable。

注释

这些方法用于查询数据库服务器以获取各种元数据。每种类型的元数据均被赋予了一个集合名称，必须传递该名称方可接收该数据。缺省集合名称为 MetaDataCollections。

通过不使用任何参数或使用模式集合名称 **MetaDataCollections** 调用 GetSchema 方法，可以查询 SQL Anywhere .NET 数据提供程序以确定所支持的模式集合列表。这会返回一个 DataTable，其中含有所支持模式集合的列表 (CollectionName)、每个集合所支持的限制数 (NumberOfRestrictions)，以及它们所使用的标识符部分的个数 (NumberOfIdentifierParts)。

集合	元数据
Columns	返回有关数据库中所有列的信息。
DataSourceInformation	返回有关数据库服务器的信息。
DataTypes	返回所支持数据类型的列表。
ForeignKeys	返回有关数据库中所有外键的信息。
IndexColumns	返回有关数据库中所有索引列的信息。
Indexes	返回有关数据库中所有索引的信息。
MetaDataCollections	返回所有集合名称的列表。
ProcedureParameters	返回有关数据库中所有过程参数的信息。

集合	元数据
Procedures	返回有关数据库中所有过程的信息。
ReservedWords	返回 SQL Anywhere 所用保留字的列表。
Restrictions	返回有关 GetSchema 中所用限制的信息。
Tables	返回有关数据库中所有表的信息。
UserDefinedTypes	返回有关数据库中所有用户定义数据类型的信息。
Users	返回有关数据库中所有用户的信息。
ViewColumns	返回有关数据库中视图内部所有列的信息。
Views	返回有关数据库中所有视图的信息。

这些集合名称也可作为只读属性在 SAMetaDataCollectionNames 类中使用。

可通过在调用 GetSchema 时指定限制数组来过滤返回的结果。

通过以下调用可以查询每个集合可用的限制：

```
GetSchema( "Restrictions" )
```

如果集合需要四个限制，则限制参数必须为 NULL 或具有四个值的字符串。

要根据特定限制进行过滤，请将用以过滤的字符串置于其在数组中的适当位置，并将任何未使用的位置均保留为 NULL。例如，Tables 集合具有三个限制：Owner、Table 和 TableType。

按 table_name 过滤 Table 集合：

```
GetSchema( "Tables", new string[ ] { NULL, "my_table", NULL } )
```

这将返回有关名为 my_table 的所有表的信息。

```
GetSchema( "Tables", new string[ ] { "DBA", "my_table", NULL } )
```

这将返回有关用户 DBA 所拥有的名为 my_table 的所有表的信息。

下面是各个集合所返回的列的汇总。如果可通过指定对某个列的限制来减少集合中所返回的行数，则该列的限制名称会显示在括号中。限制的指定顺序便是它们在下表中的显示顺序。

Columns 集合

- table_schema (Owner)
- table_name (Table)
- column_name (Column)
- ordinal_position
- column_default
- is_nullable
- data_type
- precision
- scale
- column_size

DataSourceInformation 集合

- CompositeIdentifierSeparatorPattern
- DataSourceProductName
- DataSourceProductVersion
- DataSourceProductVersionNormalized
- GroupByBehavior
- IdentifierPattern
- IdentifierCase
- OrderByColumnsInSelect
- ParameterMarkerFormat
- ParameterMarkerPattern
- ParameterNameMaxLength
- ParameterNamePattern
- QuotedIdentifierPattern
- QuotedIdentifierCase
- StatementSeparatorPattern
- StringLiteralPattern
- SupportedJoinOperators

DataTypes 集合

- TypeName
- ProviderDbType
- ColumnSize
- CreateFormat
- CreateParameters
- DataType
- IsAutoIncrementable
- IsBestMatch
- IsCaseSensitive
- IsFixedLength
- IsFixedPrecisionScale
- IsLong
- IsNullable
- IsSearchable
- IsSearchableWithLike
- IsUnsigned
- MaximumScale
- MinimumScale
- IsConcurrencyType
- IsLiteralSupported
- LiteralPrefix
- LiteralSuffix

ForeignKeys 集合

- table_schema (Owner)
- table_name (Table)
- column_name (Column)

IndexColumns 集合

- table_schema (Owner)
- table_name (Table)
- index_name (Name)
- column_name (Column)
- order

Indexes 集合

- table_schema (Owner)
- table_name (Table)
- index_name (Name)
- primary_key
- is_unique

MetaDataCollections 集合

- CollectionName
- NumberOfRestrictions
- NumberOfIdentifierParts

ProcedureParameters 集合

- procedure_schema (Owner)
- procedure_name (Name)
- parmeter_name (Parameter)
- data_type
- parameter_type
- is_input
- is_output

Procedures 集合

- procedure_schema (Owner)
- procedure_name (Name)

ReservedWords 集合

- reserved_word

Restrictions 集合

- CollectionName
- RestrictionName
- RestrictionDefault
- RestrictionNumber

Tables 集合

- table_schema (Owner)
- table_name (Table)
- table_type (TableType)

UserDefinedTypes 集合

- data_type
- default
- precision
- scale

Users 集合

- user_name (UserName)
- resource_auth
- database_auth
- schedule_auth
- user_group

ViewColumns 集合

- view_schema (Owner)
- view_name (Name)
- column_name (Column)

Views 集合

- view_schema (Owner)
- view_name (Name)

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页
- “GetSchema 方法”一节第 248 页
- “SAConnection 类”一节第 234 页

Open 方法

使用由 SAConnection.ConnectionString 指定的属性设置打开数据库连接。

语法

Visual Basic

Public Overrides Sub **Open()**

C#

public override void **Open();**

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页
- “ConnectionString 属性”一节第 238 页

InfoMessage 事件

SQL Anywhere 数据库服务器返回警告或信息性消息时发生。

语法

Visual Basic

Public Event **InfoMessage** As SAInfoMessageEventHandler

C#

public event SAInfoMessageEventHandler **InfoMessage** ;

注释

事件处理程序会收到包含与此事件有关的数据的 SaInfoMessageEventArgs 类型的参数。以下 SAaInfoMessageEventArgs 属性提供此事件所特有的信息：NativeError、Errors、Message、MessageType 和 Source。

有关详细信息，请参见关于 OleDbConnection.InfoMessage 事件的 .NET Framework 文档。

事件数据

- **MessageType** 返回消息的类型。此类型可以是以下各项之一：Action、Info、Status 或 Warning。
- **Errors** 返回从数据源发送的消息的集合。
- **Message** 返回从数据源发送的错误的完整文本。
- **Source** 返回 SQL Anywhere .NET 数据提供程序的名称。
- **NativeError** 返回由数据库返回的 SQL 代码。

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页

StateChange 事件

SAConnection 对象状态变化时发生。

语法

Visual Basic

Public Overrides Event **StateChange** As StateChangeEventHandler

C#

public event override StateChangeEventHandler **StateChange** ;

注释

事件处理程序会收到包含与此事件有关的数据的 StateChangeEventArgs 类型的参数。以下 StateChangeEventArgs 属性提供此事件所特有的信息：CurrentState 和 OriginalState。

有关详细信息，请参见关于 OleDbConnection.StateChange 事件的 .NET Framework 文档。

事件数据

- **CurrentState** 获取连接的新状态。触发事件时，连接对象将已处于新状态。
- **OriginalState** 获取连接的原始状态。

另请参见

- “SAConnection 类”一节第 234 页
- “SAConnection 成员”一节第 235 页

SAConnectionStringBuilder 类

为创建和管理 SAConnection 类所使用的连接字符串的内容提供了一种简单的方法。此类无法继承。

语法

Visual Basic

```
Public NotInheritable Class SAConnectionStringBuilder
  Inherits SAConnectionStringBuilderBase
```

C#

```
public sealed class SAConnectionStringBuilder : SAConnectionstring BuilderBase
```

注释

SAConnectionStringBuilder 类继承 SAConnectionStringBuilderBase，而后者继承 DbConnectionStringBuilder。

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SAConnectionStringBuilder 类。

Inherits: “SAConnectionStringBuilderBase 类”一节第 279 页

有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

另请参见

- “SAConnectionStringBuilder 成员”一节第 257 页

SAConnectionStringBuilder 成员

公共构造函数

成员名称	说明
SAConnectionStringBuilder 构造函数	初始化一个新的“SAConnectionStringBuilder 类”一节第 257 页实例。

公共属性

成员名称	说明
AppInfo 属性	获取或设置 AppInfo 连接属性。

成员名称	说明
AutoStart 属性	获取或设置 AutoStart 连接属性。
AutoStop 属性	获取或设置 AutoStop 连接属性。
BrowsableConnectionString（继承自 DbConnectionStringBuilder）	获取或设置一个值，指示 DbConnectionStringBuilder.ConnectionString 在 Visual Studio 设计器中是否可见。
Charset 属性	获取或设置 Charset 连接属性。
CommBufferSize 属性	获取或设置 CommBufferSize 连接属性。
CommLinks 属性	获取或设置 CommLinks 属性。
Compress 属性	获取或设置 Compress 连接属性。
CompressionThreshold 属性	获取或设置 CompressionThreshold 连接属性。
ConnectionLifetime 属性	获取或设置 ConnectionLifetime 连接属性。
ConnectionName 属性	获取或设置 ConnectionName 连接属性。
ConnectionReset 属性	获取或设置 ConnectionReset 连接属性。
ConnectionString（继承自 DbConnectionStringBuilder）	获取或设置与 DbConnectionStringBuilder 相关联的连接字符串。
ConnectionTimeout 属性	获取或设置 ConnectionTimeout 连接属性。
Count（继承自 DbConnectionStringBuilder）	获取 DbConnectionStringBuilder.ConnectionString 中当前包含的键数。
DataSourceName 属性	获取或设置 DataSourceName 连接属性。
DatabaseFile 属性	获取或设置 DatabaseFile 连接属性。
DatabaseKey 属性	获取或设置 DatabaseKey 连接属性。
DatabaseName 属性	获取或设置 DatabaseName 连接属性。
DatabaseSwitches 属性	获取或设置 DatabaseSwitches 连接属性。
DisableMultiRowFetch 属性	获取或设置 DisableMultiRowFetch 连接属性。
Elevate 属性	获取或设置 Elevate 连接属性。
EncryptedPassword 属性	获取或设置 EncryptedPassword 连接属性。

成员名称	说明
Encryption 属性	获取或设置 Encryption 连接属性。
Enlist 属性	获取或设置 Enlist 连接属性。
FileDataSourceName 属性	获取或设置 FileDataSourceName 连接属性。
ForceStart 属性	获取或设置 ForceStart 连接属性。
IdleTimeout 属性	获取或设置 IdleTimeout 连接属性。
Integrated 属性	获取或设置 Integrated 连接属性。
IsFixedSize（继承自 DbConnectionStringBuilder）	获取表示 DbConnectionStringBuilder 是否有固定大小的值。
IsReadOnly（继承自 DbConnectionStringBuilder）	获取表示 DbConnectionStringBuilder 是否为只读的值。
Item 属性（继承自 SAConnectionStringBuilderBase）	获取或设置连接关键字的值。
Kerberos 属性	获取或设置 Kerberos 连接属性。
Keys 属性（继承自 SAConnectionStringBuilderBase）	获取包含 SAConnectionStringBuilder 中的键的 System.Collections.ICollection。
Language 属性	获取或设置 Language 连接属性。
LazyClose 属性	获取或设置 LazyClose 连接属性。
LivenessTimeout 属性	获取或设置 LivenessTimeout 连接属性。
LogFile 属性	获取或设置 LogFile 连接属性。
MaxPoolSize 属性	获取或设置 MaxPoolSize 连接属性。
MinPoolSize 属性	获取或设置 MinPoolSize 连接属性。
NewPassword 属性	获取或设置 NewPassword 连接属性。
Password 属性	获取或设置 Password 连接属性。
PersistSecurityInfo 属性	获取或设置 PersistSecurityInfo 连接属性。
Pooling 属性	获取或设置 Pooling 连接属性。

成员名称	说明
PrefetchBuffer 属性	获取或设置 PrefetchBuffer 连接属性。
PrefetchRows 属性	获取或设置 PrefetchRows 连接属性。缺省值是 200。
RetryConnectionTimeout 属性	获取或设置 RetryConnectionTimeout 属性。
ServerName 属性	获取或设置 ServerName 连接属性。
StartLine 属性	获取或设置 StartLine 连接属性。
Unconditional 属性	获取或设置 Unconditional 连接属性。
UserID 属性	获取或设置 UserID 连接属性。
Values（继承自 DbConnectionStringBuilder）	获取包含 DbConnectionStringBuilder 中的值的 ICollection。

公共方法

成员名称	说明
Add（继承自 DbConnectionStringBuilder）	将具有指定键和值的条目添加到 DbConnectionStringBuilder 中。
Clear（继承自 DbConnectionStringBuilder）	清除 DbConnectionStringBuilder 实例的内容。
ContainsKey 方法（继承自 SAConnectionStringBuilderBase）	确定 SAConnectionStringBuilder 对象是否包含特定关键字。
EquivalentTo（继承自 DbConnectionStringBuilder）	将此 DbConnectionStringBuilder 对象中的连接信息与所提供对象中的连接信息进行比较。
GetKeyword 方法（继承自 SAConnectionStringBuilderBase）	获取指定 SAConnectionStringBuilder 属性的关键字。
GetUseLongNameAsKeyword 方法（继承自 SAConnectionStringBuilderBase）	获取表示是否可以在连接字符串中使用长连接参数名的布尔值。
Remove 方法（继承自 SAConnectionStringBuilderBase）	从 SAConnectionStringBuilder 实例中删除具有指定键的条目。

成员名称	说明
SetUseLongNameAsKeyword 方法（继承自 SAConnectionStringBuilderBase）	设置表示连接字符串中是否使用长连接参数名的布尔值。缺省情况下使用长连接参数名。
ShouldSerialize 方法（继承自 SAConnectionStringBuilderBase）	表示此 SAConnectionStringBuilder 实例中是否存在指定的键。
ToString（继承自 DbConnectionStringBuilder）	返回与此 DbConnectionStringBuilder 关联的连接字符串。
TryGetValue 方法（继承自 SAConnectionStringBuilderBase）	从此 SAConnectionStringBuilder 中检索与所提供的键对应的值。

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页

SAConnectionStringBuilder 构造函数

初始化一个新的“SAConnectionStringBuilder 类”一节第 257 页实例。

SAConnectionStringBuilder() 构造函数

初始化 SAConnectionStringBuilder 类的一个新实例。

语法

Visual Basic

```
Public Sub New()
```

C#

```
public  SAConnectionStringBuilder();
```

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SAConnectionStringBuilder 类。

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页
- “SAConnectionStringBuilder 构造函数”一节第 261 页

SAConnectionStringBuilder(String) 构造函数

初始化 SAConnectionStringBuilder 类的一个新实例。

语法

Visual Basic

```
Public Sub New( _
  ByVal connectionString As String _
)
```

C#

```
public  SAConnectionStringBuilder(
  string  connectionString
);
```

参数

- **connectionString**　对象的内部连接信息的基础。分解为 "关键字=值" 对。

  有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SAConnectionStringBuilder 类。

示例

以下语句为到名为 policies 的数据库的连接初始化一个 SAConnection 对象，该数据库在名为 hr 的 SQL Anywhere 数据库服务器上运行。该连接使用的用户 ID 为 admin，口令为 money。

```
SAConnectionStringBuilder conn = new
SAConnectionStringBuilder("UID=admin;PWD=money;ENG=hr;DBN=policies" );
```

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页
- “SAConnectionStringBuilder 构造函数”一节第 261 页

AppInfo 属性

获取或设置 AppInfo 连接属性。

语法

Visual Basic

```
Public Property AppInfo As String
```

C#

```
public string  AppInfo { get; set; }
```

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

AutoStart 属性

获取或设置 AutoStart 连接属性。

语法

Visual Basic

Public Property **AutoStart** As String

C#

public string **AutoStart** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

AutoStop 属性

获取或设置 AutoStop 连接属性。

语法

Visual Basic

Public Property **AutoStop** As String

C#

public string **AutoStop** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

Charset 属性

获取或设置 Charset 连接属性。

语法

Visual Basic

Public Property **Charset** As String

C#

public string **Charset** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

CommBufferSize 属性

获取或设置 CommBufferSize 连接属性。

语法

Visual Basic

Public Property **CommBufferSize** As Integer

C#

public int **CommBufferSize** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

CommLinks 属性

获取或设置 CommLinks 属性。

语法

Visual Basic

Public Property **CommLinks** As String

C#

public string **CommLinks** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

Compress 属性

获取或设置 Compress 连接属性。

语法

Visual Basic

Public Property **Compress** As String

C#

public string **Compress** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

CompressionThreshold 属性

获取或设置 CompressionThreshold 连接属性。

语法

Visual Basic

Public Property **CompressionThreshold** As Integer

C#

public int **CompressionThreshold** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

ConnectionLifetime 属性

获取或设置 ConnectionLifetime 连接属性。

语法

Visual Basic

Public Property **ConnectionLifetime** As Integer

C#

public int **ConnectionLifetime** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

ConnectionName 属性

获取或设置 ConnectionName 连接属性。

语法

Visual Basic

```
Public Property ConnectionName As String
```

C#

```
public string  ConnectionName { get; set; }
```

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

ConnectionReset 属性

获取或设置 ConnectionReset 连接属性。

语法

Visual Basic

```
Public Property ConnectionReset As Boolean
```

C#

```
public bool ConnectionReset { get; set; }
```

属性值

包含模式信息的 DataTable。

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

ConnectionTimeout 属性

获取或设置 ConnectionTimeout 连接属性。

语法

Visual Basic

Public Property **ConnectionTimeout** As Integer

C#

public int **ConnectionTimeout** { get; set; }

示例

以下语句显示 ConnectionTimeout 属性的值。

```
MessageBox.Show( connString.ConnectionTimeout.ToString() );
```

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

DataSourceName 属性

获取或设置 DataSourceName 连接属性。

语法

Visual Basic

Public Property **DataSourceName** As String

C#

public string **DataSourceName** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

DatabaseFile 属性

获取或设置 DatabaseFile 连接属性。

语法

Visual Basic

Public Property **DatabaseFile** As String

C#

public string **DatabaseFile** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

DatabaseKey 属性

获取或设置 DatabaseKey 连接属性。

语法

Visual Basic

Public Property **DatabaseKey** As String

C#

public string **DatabaseKey** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

DatabaseName 属性

获取或设置 DatabaseName 连接属性。

语法

Visual Basic

Public Property **DatabaseName** As String

C#

public string **DatabaseName** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

DatabaseSwitches 属性

获取或设置 DatabaseSwitches 连接属性。

语法

Visual Basic

Public Property **DatabaseSwitches** As String

C#

public string **DatabaseSwitches** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

DisableMultiRowFetch 属性

获取或设置 DisableMultiRowFetch 连接属性。

语法

Visual Basic

Public Property **DisableMultiRowFetch** As String

C#

public string **DisableMultiRowFetch** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

Elevate 属性

获取或设置 Elevate 连接属性。

语法

Visual Basic

Public Property **Elevate** As String

C#

public string **Elevate** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

EncryptedPassword 属性

获取或设置 EncryptedPassword 连接属性。

语法

Visual Basic

Public Property **EncryptedPassword** As String

C#

public string **EncryptedPassword** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

Encryption 属性

获取或设置 Encryption 连接属性。

语法

Visual Basic

Public Property **Encryption** As String

C#

public string **Encryption** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

Enlist 属性

获取或设置 Enlist 连接属性。

语法

Visual Basic

Public Property **Enlist** As Boolean

C#

public bool **Enlist** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

FileDataSourceName 属性

获取或设置 FileDataSourceName 连接属性。

语法

Visual Basic

Public Property **FileDataSourceName** As String

C#

public string **FileDataSourceName** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

ForceStart 属性

获取或设置 ForceStart 连接属性。

语法

Visual Basic

Public Property **ForceStart** As String

C#

public string **ForceStart** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

IdleTimeout 属性

获取或设置 IdleTimeout 连接属性。

语法

Visual Basic

Public Property **IdleTimeout** As Integer

C#

public int **IdleTimeout** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

Integrated 属性

获取或设置 Integrated 连接属性。

语法

Visual Basic

Public Property **Integrated** As String

C#

public string **Integrated** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

Kerberos 属性

获取或设置 Kerberos 连接属性。

语法

Visual Basic

Public Property **Kerberos** As String

C#

public string **Kerberos** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

Language 属性

获取或设置 Language 连接属性。

语法

Visual Basic

Public Property **Language** As String

C#

public string **Language** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

LazyClose 属性

获取或设置 LazyClose 连接属性。

语法

Visual Basic

Public Property **LazyClose** As String

C#

public string **LazyClose** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

LivenessTimeout 属性

获取或设置 LivenessTimeout 连接属性。

语法

Visual Basic

Public Property **LivenessTimeout** As Integer

C#

public int **LivenessTimeout** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

LogFile 属性

获取或设置 LogFile 连接属性。

语法

Visual Basic

Public Property **LogFile** As String

C#

public string **LogFile** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

MaxPoolSize 属性

获取或设置 MaxPoolSize 连接属性。

语法

Visual Basic

Public Property **MaxPoolSize** As Integer

C#

public int **MaxPoolSize** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

MinPoolSize 属性

获取或设置 MinPoolSize 连接属性。

语法

Visual Basic

Public Property **MinPoolSize** As Integer

C#

public int **MinPoolSize** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

NewPassword 属性

获取或设置 NewPassword 连接属性。

语法

Visual Basic

Public Property **NewPassword** As String

C#

public string **NewPassword** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

Password 属性

获取或设置 Password 连接属性。

语法

Visual Basic

Public Property **Password** As String

C#

public string **Password** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

PersistSecurityInfo 属性

获取或设置 PersistSecurityInfo 连接属性。

语法

Visual Basic

Public Property **PersistSecurityInfo** As Boolean

C#

public bool **PersistSecurityInfo** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

Pooling 属性

获取或设置 Pooling 连接属性。

语法

Visual Basic

Public Property **Pooling** As Boolean

C#

public bool **Pooling** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

PrefetchBuffer 属性

获取或设置 PrefetchBuffer 连接属性。

语法

Visual Basic

Public Property **PrefetchBuffer** As Integer

C#

public int **PrefetchBuffer** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

PrefetchRows 属性

获取或设置 PrefetchRows 连接属性。缺省值是 200。

语法

Visual Basic

Public Property **PrefetchRows** As Integer

C#

public int **PrefetchRows** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

RetryConnectionTimeout 属性

获取或设置 RetryConnectionTimeout 属性。

语法

Visual Basic

Public Property **RetryConnectionTimeout** As Integer

C#

public int **RetryConnectionTimeout** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

ServerName 属性

获取或设置 ServerName 连接属性。

语法

Visual Basic

Public Property **ServerName** As String

C#

public string **ServerName** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

StartLine 属性

获取或设置 StartLine 连接属性。

语法

Visual Basic

Public Property **StartLine** As String

C#

public string **StartLine** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

Unconditional 属性

获取或设置 Unconditional 连接属性。

语法

Visual Basic

Public Property **Unconditional** As String

C#

public string **Unconditional** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

UserID 属性

获取或设置 UserID 连接属性。

语法

Visual Basic

Public Property **UserID** As String

C#

public string **UserID** { get; set; }

另请参见

- “SAConnectionStringBuilder 类”一节第 257 页
- “SAConnectionStringBuilder 成员”一节第 257 页

SAConnectionStringBuilderBase 类

SAConnectionStringBuilder 类的基类。此类属于抽象类，因此无法将其实例化。

语法

Visual Basic

MustInherit Public Class **SAConnectionStringBuilderBase**
  Inherits DbConnectionStringBuilder

C#

public abstract class **SAConnectionStringBuilderBase** : DbConnectionstring Builder

另请参见

- “SAConnectionStringBuilderBase 成员”一节第 279 页

SAConnectionStringBuilderBase 成员

公共属性

成员名称	说明
BrowsableConnectionString（继承自 DbConnectionStringBuilder）	获取或设置一个值，指示 DbConnectionStringBuilder.ConnectionString 在 Visual Studio 设计器中是否可见。
ConnectionString（继承自 DbConnectionStringBuilder）	获取或设置与 DbConnectionStringBuilder 相关联的连接字符串。

成员名称	说明
Count（继承自 DbConnectionStringBuilder）	获取 DbConnectionStringBuilder.ConnectionString 中当前包含的键数。
IsFixedSize（继承自 DbConnectionStringBuilder）	获取表示 DbConnectionStringBuilder 是否有固定大小的值。
IsReadOnly（继承自 DbConnectionStringBuilder）	获取表示 DbConnectionStringBuilder 是否为只读的值。
Item 属性	获取或设置连接关键字的值。
Keys 属性	获取包含 SAConnectionStringBuilder 中的键的 System.Collections.ICollection。
Values（继承自 DbConnectionStringBuilder）	获取包含 DbConnectionStringBuilder 中的值的 ICollection。

公共方法

成员名称	说明
Add（继承自 DbConnectionStringBuilder）	将具有指定键和值的条目添加到 DbConnectionStringBuilder 中。
Clear（继承自 DbConnectionStringBuilder）	清除 DbConnectionStringBuilder 实例的内容。
ContainsKey 方法	确定 SAConnectionStringBuilder 对象是否包含特定关键字。
EquivalentTo（继承自 DbConnectionStringBuilder）	将此 DbConnectionStringBuilder 对象中的连接信息与所提供对象中的连接信息进行比较。
GetKeyword 方法	获取指定 SAConnectionStringBuilder 属性的关键字。
GetUseLongNameAsKeyword 方法	获取表示是否可以在连接字符串中使用长连接参数名的布尔值。
Remove 方法	从 SAConnectionStringBuilder 实例中删除具有指定键的条目。
SetUseLongNameAsKeyword 方法	设置表示连接字符串中是否使用长连接参数名的布尔值。缺省情况下使用长连接参数名。
ShouldSerialize 方法	表示此 SAConnectionStringBuilder 实例中是否存在指定的键。
ToString（继承自 DbConnectionStringBuilder）	返回与此 DbConnectionStringBuilder 关联的连接字符串。

成员名称	说明
TryGetValue 方法	从此 SAConnectionStringBuilder 中检索与所提供的键对应的值。

另请参见

- “SAConnectionStringBuilderBase 类”一节第 279 页

Item 属性

获取或设置连接关键字的值。

语法

Visual Basic

```
Public Overrides Default Property Item ( _
  ByVal keyword As String _
) As Object
```

C#

```
public override object this [
  string  keyword
] { get; set; }
```

参数

- **keyword**　连接关键字的名称。

属性值

表示指定连接关键字的值的对象。

注释

如果关键字或类型无效，则会抛出异常。关键字不区分大小写。

设置该值时，传递 NULL 会将该值清除。

另请参见

- “SAConnectionStringBuilderBase 类”一节第 279 页
- “SAConnectionStringBuilderBase 成员”一节第 279 页

Keys 属性

获取包含 SAConnectionStringBuilder 中的键的 System.Collections.ICollection。

语法

Visual Basic

```
Public Overrides Readonly Property Keys As ICollection
```

C#

```
public override ICollection Keys { get;}
```

属性值

包含 SAConnectionStringBuilder 中的键的 System.Collections.ICollection。

另请参见

- “SAConnectionStringBuilderBase 类”一节第 279 页
- “SAConnectionStringBuilderBase 成员”一节第 279 页

ContainsKey 方法

确定 SAConnectionStringBuilder 对象是否包含特定关键字。

语法

Visual Basic

```
Public Overrides Function ContainsKey( _
  ByVal keyword As String _
) As Boolean
```

C#

```
public override bool ContainsKey(
  string  keyword
);
```

参数

- **keyword**　要在 SAConnectionStringBuilder 中查找的关键字。

返回值

如果已设置与关键字关联的值，则为 true；否则为 false。

示例

以下语句确定 SAConnectionStringBuilder 对象是否包含 UserID 关键字。

```
connectString.ContainsKey("UserID")
```

另请参见

- “SAConnectionStringBuilderBase 类”一节第 279 页
- “SAConnectionStringBuilderBase 成员”一节第 279 页

GetKeyword 方法

获取指定 SAConnectionStringBuilder 属性的关键字。

语法

Visual Basic

```
Public Function GetKeyword( _
  ByVal propName As String _
) As String
```

C#

```
public string  GetKeyword(
  string  propName
);
```

参数

- **propName**　SAConnectionStringBuilder 属性的名称。

返回值

指定 SAConnectionStringBuilder 属性的关键字。

另请参见

- “SAConnectionStringBuilderBase 类”一节第 279 页
- “SAConnectionStringBuilderBase 成员”一节第 279 页

GetUseLongNameAsKeyword 方法

获取表示是否可以在连接字符串中使用长连接参数名的布尔值。

语法

Visual Basic

```
Public Function GetUseLongNameAsKeyword() As Boolean
```

C#

```
public bool GetUseLongNameAsKeyword();
```

返回值

如果使用长连接参数名来生成连接字符串，则为 true；否则为 false。

注释

SQL Anywhere 连接参数的名称有长、短两种形式。例如，要在连接字符串中指定 ODBC 数据源的名称，可以使用以下两个值之一：DataSourceName 或 DSN。缺省情况下使用长连接参数名来生成连接字符串。

另请参见

- “SAConnectionStringBuilderBase 类”一节第 279 页
- “SAConnectionStringBuilderBase 成员”一节第 279 页
- “SetUseLongNameAsKeyword 方法”一节第 284 页

Remove 方法

从 SAConnectionStringBuilder 实例中删除具有指定键的条目。

语法

Visual Basic

```
Public Overrides Function Remove( _
   ByVal keyword As String _
) As Boolean
```

C#

```
public override bool Remove(
   string  keyword
);
```

参数

- **keyword**　要从此 SAConnectionStringBuilder 中的连接字符串删除的 "键/值" 对的键。

返回值

如果连接字符串内存在过键且已被删除，则为 true；如果键不曾存在，则为 false。

另请参见

- “SAConnectionStringBuilderBase 类”一节第 279 页
- “SAConnectionStringBuilderBase 成员”一节第 279 页

SetUseLongNameAsKeyword 方法

设置表示连接字符串中是否使用长连接参数名的布尔值。缺省情况下使用长连接参数名。

语法

Visual Basic

```
Public Sub SetUseLongNameAsKeyword( _
   ByVal useLongNameAsKeyword As Boolean _
)
```

C#

```
public void SetUseLongNameAsKeyword(
```

```
  bool useLongNameAsKeyword
);
```

参数

- **useLongNameAsKeyword** 表示连接字符串中是否使用长连接参数名的布尔值。

另请参见

- “SAConnectionStringBuilderBase 类”一节第 279 页
- “SAConnectionStringBuilderBase 成员”一节第 279 页
- “GetUseLongNameAsKeyword 方法”一节第 283 页

ShouldSerialize 方法

表示此 SAConnectionStringBuilder 实例中是否存在指定的键。

语法

Visual Basic

```
Public Overrides Function ShouldSerialize( _
  ByVal keyword As String _
) As Boolean
```

C#

```
public override bool ShouldSerialize(
  string  keyword
);
```

参数

- **keyword** 要在 SAConnectionStringBuilder 中查找的键。

返回值

如果 SAConnectionStringBuilder 包含具有指定键的条目，则为 true；否则为 false。

另请参见

- “SAConnectionStringBuilderBase 类”一节第 279 页
- “SAConnectionStringBuilderBase 成员”一节第 279 页

TryGetValue 方法

从此 SAConnectionStringBuilder 中检索与所提供的键对应的值。

语法

Visual Basic

```
Public Overrides Function TryGetValue( _
```

```
  ByVal keyword As String, _
  ByVal value As Object _
) As Boolean
```

C#

```
public override bool TryGetValue(
  string  keyword,
  object value
);
```

参数

- **keyword** 要检索的项目的键。
- **value** 与关键字对应的值。

返回值

如果在连接字符串内找到了关键字，则为 true；否则为 false。

另请参见

- “SAConnectionStringBuilderBase 类”一节第 279 页
- “SAConnectionStringBuilderBase 成员”一节第 279 页

SADataAdapter 类

表示用来填充 DataSet 和更新数据库的一组命令和一个数据库连接。此类无法继承。

语法

Visual Basic

```
Public NotInheritable Class SADataAdapter
  Inherits DbDataAdapter
```

C#

```
public sealed class SADataAdapter : DbDataAdapter
```

注释

DataSet 提供了一种脱机处理数据的方法。SADataAdapter 提供了一些将 DataSet 与一组 SQL 语句相关联的方法。

Implements:IDbDataAdapter、IDataAdapter、ICloneable

有关详细信息，请参见“使用 SADataAdapter 对象访问和操作数据”一节第 117 页和“访问和操作数据”一节第 111 页。

另请参见

- “SADataAdapter 成员”一节第 287 页

SADataAdapter 成员

公共构造函数

成员名称	说明
SADataAdapter 构造函数	初始化一个新的“SADataAdapter 类”一节第 286 页实例。

公共属性

成员名称	说明
AcceptChangesDuringFill（继承自 DataAdapter）	获取或设置一个值，指示在任何 Fill 操作期间将 DataRow 添加到 DataTable 之后是否对其调用 DataRow.AcceptChanges。
AcceptChangesDuringUpdate（继承自 DataAdapter）	获取或设置在 DataAdapter.Update 期间是否调用 DataRow.AcceptChanges。
ContinueUpdateOnError（继承自 DataAdapter）	获取或设置一个值，指定在行更新过程中遇到错误时是否产生异常。
DeleteCommand 属性	指定在调用 Update 方法来删除数据库中与 DataSet 中已删除的行对应的行时对数据库执行的 SACommand 对象。
FillLoadOption（继承自 DataAdapter）	获取或设置决定适配器如何从 DbDataReader 填充 DataTable 的 LoadOption。
InsertCommand 属性	指定在调用 Update 方法来向数据库中添加与 DataSet 中插入的行对应的行时对数据库执行的 SACommand。
MissingMappingAction（继承自 DataAdapter）	决定当进来的数据没有匹配的表或列时所执行的操作。
MissingSchemaAction（继承自 DataAdapter）	决定当现有 DataSet 模式与进入的数据不匹配时采取何种动作。
ReturnProviderSpecificTypes（继承自 DataAdapter）	获取或设置 Fill 方法应返回特定于提供程序的值还是与 CLS 兼容的公用值。
SelectCommand 属性	指定在 Fill 或 FillSchema 过程中使用的 SACommand，它用于从数据库获取要复制到 DataSet 中的结果集。
TableMappings 属性	指定提供源表与 DataTable 之间主映射的集合。
UpdateBatchSize 属性	获取或设置每次到服务器的往返过程中处理的行数。
UpdateCommand 属性	指定在调用 Update 方法来更新数据库中与 DataSet 中已更新的行对应的行时对数据库执行的 SACommand。

公共方法

成员名称	说明
Fill（继承自 DbDataAdapter）	添加或刷新 DataSet 中的行。
FillSchema（继承自 DbDataAdapter）	将名为 "Table" 的 DataTable 添加到指定的 DataSet，并配置模式，使其与基于指定 SchemaType 的数据源中的模式相匹配。
GetFillParameters 方法	返回执行 SELECT 语句时所设置的参数。
ResetFillLoadOption（继承自 DataAdapter）	将 DataAdapter.FillLoadOption 重置为其缺省状态，并使 DataAdapter.Fill 履行 DataAdapter.AcceptChangesDuringFill。
ShouldSerializeAcceptChangesDuringFill（继承自 DataAdapter）	决定是否应保持 DataAdapter.AcceptChangesDuringFill。
ShouldSerializeFillLoadOption（继承自 DataAdapter）	决定是否应保持 DataAdapter.FillLoadOption。
Update（继承自 DbDataAdapter）	针对指定 DataRow 对象数组中每一插入、更新或删除的行调用相应的 INSERT、UPDATE 或 DELETE 语句。

公共事件

成员名称	说明
FillError（继承自 DataAdapter）	填充操作过程中出现错误时返回。
RowUpdated 事件	更新过程中对数据源执行命令后发生。尝试进行更新时便会触发该事件。
RowUpdating 事件	更新过程中对数据源执行命令前发生。尝试进行更新时便会触发该事件。

另请参见

- “SADataAdapter 类”一节第 286 页

SADataAdapter 构造函数

初始化一个新的“SADataAdapter 类”一节第 286 页实例。

SADataAdapter() 构造函数

初始化 SADataAdapter 对象。

语法

Visual Basic

```
Public Sub New()
```

C#

```
public  SADataAdapter();
```

另请参见

- “SADataAdapter 类”一节第 286 页
- “SADataAdapter 成员”一节第 287 页
- “SADataAdapter 构造函数”一节第 288 页
- “SADataAdapter(SACommand) 构造函数”一节第 289 页
- “SADataAdapter(String, SAConnection) 构造函数”一节第 290 页
- “SADataAdapter(String, String) 构造函数”一节第 290 页

SADataAdapter(SACommand) 构造函数

用指定的 SELECT 语句初始化 SADataAdapter 对象。

语法

Visual Basic

```
Public Sub New( _
  ByVal selectCommand As SACommand _
)
```

C#

```
public  SADataAdapter(
  SACommand selectCommand
);
```

参数

- **selectCommand**　DbDataAdapter.Fill 过程中使用的 SACommand 对象，用于从数据源中选择要放置在 DataSet 中的记录。

另请参见

- “SADataAdapter 类”一节第 286 页
- “SADataAdapter 成员”一节第 287 页
- “SADataAdapter 构造函数”一节第 288 页
- “SADataAdapter() 构造函数”一节第 288 页
- “SADataAdapter(String, SAConnection) 构造函数”一节第 290 页
- “SADataAdapter(String, String) 构造函数”一节第 290 页

SADataAdapter(String, SAConnection) 构造函数

用指定的 SELECT 语句和连接初始化 SADataAdapter 对象。

语法

Visual Basic

```
Public Sub New( _
  ByVal selectCommandText As String, _
  ByVal selectConnection As SAConnection _
)
```

C#

```
public  SADataAdapter(
  string  selectCommandText,
  SAConnection selectConnection
);
```

参数

- **selectCommandText**　用于设置 SADataAdapter 对象的 SADataAdapter.SelectCommand 属性的 SELECT 语句。
- **selectConnection**　定义到数据库的连接的 SAConnection 对象。

另请参见

- “SADataAdapter 类”一节第 286 页
- “SADataAdapter 成员”一节第 287 页
- “SADataAdapter 构造函数”一节第 288 页
- “SADataAdapter() 构造函数”一节第 288 页
- “SADataAdapter(SACommand) 构造函数”一节第 289 页
- “SADataAdapter(String, String) 构造函数”一节第 290 页
- “SelectCommand 属性”一节第 292 页
- “SAConnection 类”一节第 234 页

SADataAdapter(String, String) 构造函数

用指定的 SELECT 语句和连接字符串初始化 SADataAdapter 对象。

语法

Visual Basic

```
Public Sub New( _
  ByVal selectCommandText As String, _
  ByVal selectConnectionString As String _
)
```

C#

```
public  SADataAdapter(
```

```
  string  selectCommandText,
  string  selectConnectionString
);
```

参数

- **selectCommandText** 用于设置 SADataAdapter 对象的 SADataAdapter.SelectCommand 属性的 SELECT 语句。
- **selectConnectionString** SQL Anywhere 数据库的连接字符串。

另请参见

- “SADataAdapter 类”一节第 286 页
- “SADataAdapter 成员”一节第 287 页
- “SADataAdapter 构造函数”一节第 288 页
- “SADataAdapter() 构造函数”一节第 288 页
- “SADataAdapter(SACommand) 构造函数”一节第 289 页
- “SADataAdapter(String, SAConnection) 构造函数”一节第 290 页
- “SelectCommand 属性”一节第 292 页

DeleteCommand 属性

指定在调用 Update 方法来删除数据库中与 DataSet 中已删除的行对应的行时对数据库执行的 SACommand 对象。

语法

Visual Basic

Public Property **DeleteCommand** As SACommand

C#

public SACommand **DeleteCommand** { get; set; }

注释

如果 Update 过程中未设置此属性，且 DataSet 中具有主键信息，则可以通过设置 SelectCommand 和使用 SACommandBuilder 来自动生成 DeleteCommand。在这种情况下，SACommandBuilder 会生成您未设置的所有其它命令。此生成逻辑要求 SelectCommand 中存在键列信息。

将 DeleteCommand 指派给现有 SACommand 对象时，不会复制 SACommand 对象。DeleteCommand 会保留对现有 SACommand 的引用。

另请参见

- “SADataAdapter 类”一节第 286 页
- “SADataAdapter 成员”一节第 287 页
- “SelectCommand 属性”一节第 292 页

InsertCommand 属性

指定在调用 Update 方法来向数据库中添加与 DataSet 中插入的行对应的行时对数据库执行的 SACommand。

语法

Visual Basic

Public Property **InsertCommand** As SACommand

C#

public SACommand **InsertCommand** { get; set; }

注释

SACommandBuilder 不需要键列便可生成 InsertCommand。

将 InsertCommand 指派给现有 SACommand 对象时，不会复制 SACommand。InsertCommand 会保留对现有 SACommand 的引用。

如果此命令返回行，这些行可能会添加到 DataSet 中，具体视 SACommand 对象 UpdatedRowSource 属性的设置而定。

另请参见

- “SADataAdapter 类”一节第 286 页
- “SADataAdapter 成员”一节第 287 页

SelectCommand 属性

指定在 Fill 或 FillSchema 过程中使用的 SACommand，它用于从数据库获取要复制到 DataSet 中的结果集。

语法

Visual Basic

Public Property **SelectCommand** As SACommand

C#

public SACommand **SelectCommand** { get; set; }

注释

将 SelectCommand 指派给以前创建的 SACommand 时，不会复制 SACommand。SelectCommand 会保留对以前创建的 SACommand 对象的引用。

如果 SelectCommand 不返回任何行，则不会有任何表添加到 DataSet 中，也不会抛出任何异常。

SELECT 语句也可以在 SADataAdapter 构造函数中进行指定。

另请参见

- “SADataAdapter 类”一节第 286 页
- “SADataAdapter 成员”一节第 287 页

TableMappings 属性

指定提供源表与 DataTable 之间主映射的集合。

语法

Visual Basic

Public Readonly Property **TableMappings** As DataTableMappingCollection

C#

public DataTableMappingCollection **TableMappings** { get;}

注释

缺省值为空集合。

在使更改保持一致时，SADataAdapter 会使用 DataTableMappingCollection 集合将数据源使用的列名与 DataSet 使用的列名相关联。

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 TableMappings 属性。

另请参见

- “SADataAdapter 类”一节第 286 页
- “SADataAdapter 成员”一节第 287 页

UpdateBatchSize 属性

获取或设置每次到服务器的往返过程中处理的行数。

语法

Visual Basic

Public Overrides Property **UpdateBatchSize** As Integer

C#

public override int **UpdateBatchSize** { get; set; }

注释

缺省值是 1。

将该值设置为某个大于 1 的值会使 SADataAdapter.Update 执行批处理中的所有插入语句。删除和更新操作依旧按顺序执行，但之后的插入操作是以与 UpdateBatchSize 值相等的批处理大小来执行的。将该值设置为 0 会使 Update 通过一个批处理发送插入语句。

将该值设置为某个大于 1 的值会使 SADataAdapter.Fill 执行批处理中的所有插入语句。删除和更新操作依旧按顺序执行，但之后的插入操作是以与 UpdateBatchSize 值相等的批处理大小来执行的。

将该值设置为 0 会使 Fill 通过一个批处理发送插入语句。

将它设置为小于 0 是错误的。

如果将 UpdateBatchSize 设置为一以外的值，且将 InsertCommand 属性设置为的值不是 INSERT 语句，则调用 Fill 时会抛出异常。

此行为与 SqlDataAdapter 不同，它会对所有类型的命令进行批处理。

另请参见

- “SADataAdapter 类”一节第 286 页
- “SADataAdapter 成员”一节第 287 页

UpdateCommand 属性

指定在调用 Update 方法来更新数据库中与 DataSet 中已更新的行对应的行时对数据库执行的 SACommand。

语法

Visual Basic

```
Public Property UpdateCommand As SACommand
```

C#

```
public SACommand UpdateCommand { get; set; }
```

注释

如果 Update 过程中未设置此属性，且 SelectCommand 中具有主键信息，则可以通过设置 SelectCommand 属性和使用 SACommandBuilder 来自动生成 UpdateCommand。SACommandBuilder 随后会生成您未设置的任何其它命令。此生成逻辑要求 SelectCommand 中存在键列信息。

将 UpdateCommand 指派给以前创建的 SACommand 时，不会复制 SACommand。UpdateCommand 会保留对以前创建的 SACommand 对象的引用。

如果执行此命令时返回行，这些行可能会合并到 DataSet 中，具体视 SACommand 对象 UpdatedRowSource 属性的设置而定。

另请参见

- “SADataAdapter 类”一节第 286 页
- “SADataAdapter 成员”一节第 287 页

GetFillParameters 方法

返回执行 SELECT 语句时所设置的参数。

语法

Visual Basic

Public Function **GetFillParameters()** As SAParameter

C#

public SAParameter **GetFillParameters();**

返回值

包含由用户设置的参数的 IDataParameter 对象数组。

另请参见

- “SADataAdapter 类”一节第 286 页
- “SADataAdapter 成员”一节第 287 页

RowUpdated 事件

更新过程中对数据源执行命令后发生。尝试进行更新时便会触发该事件。

语法

Visual Basic

Public Event **RowUpdated** As SARowUpdatedEventHandler

C#

public event SARowUpdatedEventHandler **RowUpdated** ;

注释

事件处理程序会收到包含与此事件有关的数据的 SARowUpdatedEventArgs 类型的参数。

有关详细信息，请参见关于 OleDbDataAdapter.RowUpdated 事件的 .NET Framework 文档。

事件数据

- **Command** 获取在调用 DataAdapter.Update 时执行的 SACommand。
- **RecordsAffected** 返回通过执行 SQL 语句所更改、插入或删除的行数。
- **Command** 获取调用 DbDataAdapter.Update 时执行的 IDbCommand。
- **Errors** 获取执行 RowUpdatedEventArgs.Command 时 .NET Framework 数据提供程序所产生的任何错误。
- **Row** 获取通过 DbDataAdapter.Update 发送的 DataRow。

- **RowCount** 获取一批更新记录中处理的行数。
- **StatementType** 获取所执行 SQL 语句的类型。
- **Status** 获取 RowUpdatedEventArgs.Command 的 UpdateStatus。
- **TableMapping** 获取通过 DbDataAdapter.Update 发送的 DataTableMapping。

另请参见

- “SADataAdapter 类”一节第 286 页
- “SADataAdapter 成员”一节第 287 页

RowUpdating 事件

更新过程中对数据源执行命令前发生。尝试进行更新时便会触发该事件。

语法

Visual Basic

Public Event **RowUpdating** As SARowUpdatingEventHandler

C#

public event SARowUpdatingEventHandler **RowUpdating** ;

注释

事件处理程序会收到包含与此事件有关的数据的 SARowUpdatingEventArgs 类型的参数。

有关详细信息，请参见关于 OleDbDataAdapter.RowUpdating 事件的 .NET Framework 文档。

事件数据

- **Command** 指定要在执行 Update 时执行的 SACommand。
- **Command** 获取要在 DbDataAdapter.Update 操作期间执行的 IDbCommand。
- **Errors** 获取执行 RowUpdatedEventArgs.Command 时 .NET Framework 数据提供程序所产生的任何错误。
- **Row** 获取将作为插入、更新或删除操作的一部分发送给服务器的 DataRow。
- **StatementType** 获取要执行的 SQL 语句的类型。
- **Status** 获取或设置 RowUpdatedEventArgs.Command 的 UpdateStatus。
- **TableMapping** 获取要通过 DbDataAdapter.Update 发送的 DataTableMapping。

另请参见

- “SADataAdapter 类”一节第 286 页
- “SADataAdapter 成员”一节第 287 页

SADataReader 类

来自查询或存储过程的只读只进结果集。此类无法继承。

语法

Visual Basic

```
Public NotInheritable Class SADataReader
  Inherits DbDataReader
  Implements IListSource
```

C#

```
public sealed class SADataReader : DbDataReader,
  IListSource
```

注释

没有用于 SADataReader 的构造函数。要获取 SADataReader 对象，请执行 SACommand：

```
SACommand cmd = new SACommand(
    "SELECT EmployeeID FROM Employees", conn );
SADataReader reader = cmd.ExecuteReader();
```

只能在 SADataReader 中向前移动。如果您需要更为灵活的对象来操作结果，请使用 SADataAdapter。

SADataReader 会根据需要检索行，而 SADataAdapter 必须检索结果集的所有行，然后您才能对对象执行任何操作。如果结果集较大，这种差异会使 SADataReader 的响应时间比 SADataAdapter 短很多。

Implements:IDataReader、IDisposable、IDataRecord、IListSource

有关详细信息，请参见“访问和操作数据”一节第 111 页。

另请参见

- “SADataReader 成员”一节第 297 页
- “ExecuteReader() 方法”一节第 214 页

SADataReader 成员

公共属性

成员名称	说明
Depth 属性	获取表示当前行嵌套深度的值。最外层的表深度为 0。
FieldCount 属性	获取结果集中的列数。
HasRows 属性	获取表示 SADataReader 是包含一行还是多行的值。

成员名称	说明
IsClosed 属性	获取表示 SADataReader 是否已关闭的值。
Item 属性	返回以本地格式表示的列值。在 C# 中，此属性是 SADataReader 类的索引器。
RecordsAffected 属性	通过执行该 SQL 语句而更改、插入或删除的行数。
VisibleFieldCount（继承自 DbDataReader）	获取 DbDataReader 中未隐藏的字段数。

公共方法

成员名称	说明
Close 方法	关闭 SADataReader。
Dispose（继承自 DbDataReader）	释放 DbDataReader 的当前实例使用的所有资源。
GetBoolean 方法	以布尔值形式返回指定列的值。
GetByte 方法	以字节形式返回指定列的值。
GetBytes 方法	以给定缓冲区偏移为起点，将字节流作为数组从指定的列偏移读入到缓冲区中。
GetChar 方法	以字符形式返回指定列的值。
GetChars 方法	以给定缓冲区偏移为起点，将字符流作为数组从指定的列偏移读入到缓冲区中。
GetData 方法	不支持此方法。调用时它会抛出 InvalidOperationException。
GetDataTypeName 方法	返回源数据类型的名称。
GetDateTime 方法	以 DateTime 对象形式返回指定列的值。
GetDecimal 方法	以 Decimal 对象形式返回指定列的值。
GetDouble 方法	以双精度浮点数形式返回指定列的值。
GetEnumerator 方法	返回迭代通过 SADataReader 对象的 IEnumerator。
GetFieldType 方法	返回作为对象的数据类型的 Type。
GetFloat 方法	以单精度浮点数形式返回指定列的值。

成员名称	说明
GetGuid 方法	以全局唯一标识符（global unique identifier，简称 GUID）形式返回指定列的值。
GetInt16 方法	以 16 位有符号整数形式返回指定列的值。
GetInt32 方法	以 32 位有符号整数形式返回指定列的值。
GetInt64 方法	以 64 位有符号整数形式返回指定列的值。
GetName 方法	返回指定列的名称。
GetOrdinal 方法	在已知列名的情况下返回列顺序号。
GetProviderSpecificFieldType（继承自 DbDataReader）	返回指定列特定于提供程序的字段类型。
GetProviderSpecificValue（继承自 DbDataReader）	以 Object 实例的形式获取指定列的值。
GetProviderSpecificValues（继承自 DbDataReader）	获取当前行的集合中所有特定于提供程序的属性列。
GetSchemaTable 方法	返回说明 SADataReader 的列元数据的 DataTable。
GetString 方法	以字符串形式返回指定列的值。
GetTimeSpan 方法	以 TimeSpan 对象形式返回指定列的值。
GetUInt16 方法	以 16 位无符号整数形式返回指定列的值。
GetUInt32 方法	以 32 位无符号整数形式返回指定列的值。
GetUInt64 方法	以 64 位无符号整数形式返回指定列的值。
GetValue 方法	以对象形式返回指定列的值。
GetValues 方法	获取当前行中的所有列。
IsDBNull 方法	返回表示列是否包含 NULL 值的值。
NextResult 方法	读取批处理 SQL 语句的结果时，将 SADataReader 移到下一结果。
Read 方法	读取结果集的下一行，并将 SADataReader 移动到该行。
myDispose 方法	释放与该对象关联的资源。

另请参见

- “SADataReader 类”一节第 297 页
- “ExecuteReader() 方法”一节第 214 页

Depth 属性

获取表示当前行嵌套深度的值。最外层的表深度为 0。

语法

Visual Basic

Public Overrides Readonly Property **Depth** As Integer

C#

public override int **Depth** { get;}

属性值

当前行的嵌套深度。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

FieldCount 属性

获取结果集中的列数。

语法

Visual Basic

Public Overrides Readonly Property **FieldCount** As Integer

C#

public override int **FieldCount** { get;}

属性值

当前记录中的列数。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

HasRows 属性

获取表示 SADataReader 是包含一行还是多行的值。

语法

Visual Basic

Public Overrides Readonly Property **HasRows** As Boolean

C#

public override bool **HasRows** { get;}

属性值

如果 SADataReader 包含一行或多行，则为 true；否则为 false。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

IsClosed 属性

获取表示 SADataReader 是否已关闭的值。

语法

Visual Basic

Public Overrides Readonly Property **IsClosed** As Boolean

C#

public override bool **IsClosed** { get;}

属性值

如果 SADataReader 已关闭，则返回 true，否则返回 false。

注释

关闭 SADataReader 后，只能调用 IsClosed 和 RecordsAffected 属性。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

Item 属性

返回以本地格式表示的列值。在 C# 中，此属性是 SADataReader 类的索引器。

Item(Int32) 属性

返回以本地格式表示的列值。在 C# 中，此属性是 SADataReader 类的索引器。

语法

Visual Basic

```
Public Overrides Default Readonly Property Item ( _
   ByVal index As Integer _
) As Object
```

C#

```
public override object this [
   int index
] { get;}
```

参数

- **index** 列顺序号。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页
- “Item 属性”一节第 302 页

Item(String) 属性

返回以本地格式表示的列值。在 C# 中，此属性是 SADataReader 类的索引器。

语法

Visual Basic

```
Public Overrides Default Readonly Property Item ( _
   ByVal name As String _
) As Object
```

C#

```
public override object this [
   string  name
] { get;}
```

参数

- **name** 列名称。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页
- “Item 属性”一节第 302 页

RecordsAffected 属性

通过执行该 SQL 语句而更改、插入或删除的行数。

语法

Visual Basic

Public Overrides Readonly Property **RecordsAffected** As Integer

C#

public override int **RecordsAffected** { get;}

属性值

更改、插入或删除的行数。如果没有任何行受到影响或语句失败，此值将为 0；如果是 SELECT 语句，则此值为 -1。

注释

更改、插入或删除的行数。如果没有任何行受到影响或语句失败，此值将为 0；如果是 SELECT 语句，则此值为 -1。

此属性的值是累积的。例如，如果在批处理模式中插入两个记录，则 RecordsAffected 的值将是二。

关闭 SADataReader 后，只能调用 IsClosed 和 RecordsAffected 属性。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

Close 方法

关闭 SADataReader。

语法

Visual Basic

Public Overrides Sub **Close()**

C#

public override void **Close();**

注释

使用完 SADataReader 之后，必须显式地调用 Close 方法。

当在自动提交模式下运行时，会发出作为关闭 SADataReader 的副作用的 COMMIT。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

GetBoolean 方法

以布尔值形式返回指定列的值。

语法

Visual Basic

```
Public Overrides Function GetBoolean( _
  ByVal ordinal As Integer _
) As Boolean
```

C#

```
public override bool GetBoolean(
  int ordinal
);
```

参数

- **ordinal** 表示获取值的来源列的顺序号。编号从零开始。

返回值

列的值。

注释

不会进行任何转换，因此检索的数据必须已经是布尔值。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页
- “GetOrdinal 方法”一节第 316 页
- “GetFieldType 方法”一节第 312 页

GetByte 方法

以字节形式返回指定列的值。

语法

Visual Basic

```
Public Overrides Function GetByte( _
  ByVal ordinal As Integer _
) As Byte
```

C#

```
public override byte GetByte(
  int ordinal
);
```

参数

- **ordinal** 表示获取值的来源列的顺序号。编号从零开始。

返回值

列的值。

注释

不会进行任何转换，因此检索的数据必须已经是字节。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

GetBytes 方法

以给定缓冲区偏移为起点，将字节流作为数组从指定的列偏移读入到缓冲区中。

语法

Visual Basic

```
Public Overrides Function GetBytes( _
  ByVal ordinal As Integer, _
  ByVal dataIndex As Long, _
  ByVal buffer As Byte(), _
  ByVal bufferIndex As Integer, _
  ByVal length As Integer _
) As Long
```

C#

```
public override long GetBytes(
  int ordinal,
  long dataIndex,
  byte[] buffer,
  int bufferIndex,
  int length
);
```

参数

- **ordinal** 表示获取值的来源列的顺序号。编号从零开始。
- **dataIndex** 从中读取字节的列值内的索引。
- **buffer** 存储数据的数组。
- **bufferIndex** 数组中作为数据复制起点的索引。
- **length** 要复制到指定缓冲区中的最大长度。

返回值

读取的字节数。

注释

GetBytes 返回字段中的可用字节数。在大多数情况下，这是字段的实际长度。然而，如果 GetBytes 已经用于从字段中获取字节，则返回的数字可能比字段的实际长度小。例如，当 SADataReader 将一个较大的数据结构读入缓冲区时，就可能出现这种情况。

如果传递的缓冲区是空值引用（在 Visual Basic 中是 Nothing），GetBytes 会以字节为单位返回该字段的长度。

不会进行任何转换，因此检索的数据必须已经是字节数组。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

GetChar 方法

以字符形式返回指定列的值。

语法

Visual Basic

```
Public Overrides Function GetChar( _
   ByVal ordinal As Integer _
) As Char
```

C#

```
public override char GetChar(
   int ordinal
);
```

参数

- **ordinal** 表示获取值的来源列的顺序号。编号从零开始。

返回值

列的值。

注释

不会进行任何转换，因此检索的数据必须已经是字符。

在调用此方法之前，调用 SADataReader.IsDBNull 方法以检查是否存在空值。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页
- “IsDBNull 方法”一节第 325 页
- “IsDBNull 方法”一节第 325 页

GetChars 方法

以给定缓冲区偏移为起点，将字符流作为数组从指定的列偏移读入到缓冲区中。

语法

Visual Basic

```
Public Overrides Function GetChars( _
   ByVal ordinal As Integer, _
   ByVal dataIndex As Long, _
   ByVal buffer As Char(), _
   ByVal bufferIndex As Integer, _
   ByVal length As Integer _
) As Long
```

C#

```
public override long GetChars(
   int ordinal,
   long dataIndex,
   char[] buffer,
   int bufferIndex,
   int length
);
```

参数

- **ordinal** 从零开始的列顺序号。
- **dataIndex** 行内作为读取操作起点的索引。
- **buffer** 接收所复制数据的缓冲区。
- **bufferIndex** 缓冲区开始执行读取操作的索引。
- **length** 要读取的字符数。

返回值

实际读取的字符数。

注释

GetChars 返回字段中的可用字符数。在大多数情况下，这是字段的实际长度。然而，如果 GetChars 已用于从字段中获取字符，则返回的数字可能比字段的实际长度小。例如，当 SADataReader 将一个较大的数据结构读入缓冲区时，就可能出现这种情况。

如果传递的缓冲区是空值引用（在 Visual Basic 中是 Nothing），GetChars 会以字符为单位返回该字段的长度。

不会进行任何转换，因此检索的数据必须已经是字符数组。

有关处理 BLOB 的信息，请参见“处理 BLOB”一节第 125 页。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

GetData 方法

不支持此方法。调用时它会抛出 InvalidOperationException。

语法

Visual Basic

```
Public Function GetData( _
   ByVal i As Integer _
) As IDataReader
```

C#

```
public IDataReader GetData(
   int i
);
```

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页
- InvalidOperationException

GetDataTypeName 方法

返回源数据类型的名称。

语法

Visual Basic

```
Public Overrides Function GetDataTypeName( _
  ByVal index As Integer _
) As String
```

C#

```
public override string  GetDataTypeName(
  int index
);
```

参数

- **index** 从零开始的列顺序号。

返回值

后端数据类型的名称。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

GetDateTime 方法

以 DateTime 对象形式返回指定列的值。

语法

Visual Basic

```
Public Overrides Function GetDateTime( _
  ByVal ordinal As Integer _
) As Date
```

C#

```
public override DateTime GetDateTime(
  int ordinal
);
```

参数

- **ordinal** 从零开始的列顺序号。

返回值

指定列的值。

注释

不会进行任何转换，因此检索的数据必须已经是 DateTime 对象。

在调用此方法之前，调用 SADataReader.IsDBNull 方法以检查是否存在空值。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页
- “IsDBNull 方法”一节第 325 页

GetDecimal 方法

以 Decimal 对象形式返回指定列的值。

语法

Visual Basic

```
Public Overrides Function GetDecimal( _
  ByVal ordinal As Integer _
) As Decimal
```

C#

```
public override decimal GetDecimal(
  int ordinal
);
```

参数

- **ordinal** 表示获取值的来源列的顺序号。编号从零开始。

返回值

指定列的值。

注释

不会进行任何转换，因此检索的数据必须已经是 Decimal 对象。

在调用此方法之前，调用 SADataReader.IsDBNull 方法以检查是否存在空值。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页
- “IsDBNull 方法”一节第 325 页

GetDouble 方法

以双精度浮点数形式返回指定列的值。

语法

Visual Basic

```
Public Overrides Function GetDouble( _
  ByVal ordinal As Integer _
) As Double
```

C#

```
public override double GetDouble(
  int ordinal
);
```

参数

- **ordinal** 表示获取值的来源列的顺序号。编号从零开始。

返回值

指定列的值。

注释

不会进行任何转换，因此检索的数据必须已经是双精度浮点数。

在调用此方法之前，调用 SADataReader.IsDBNull 方法以检查是否存在空值。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页
- “IsDBNull 方法”一节第 325 页

GetEnumerator 方法

返回迭代通过 SADataReader 对象的 IEnumerator。

语法

Visual Basic

```
Public Overrides Function GetEnumerator() As IEnumerator
```

C#

```
public override IEnumerator GetEnumerator();
```

返回值

SADataReader 对象的 IEnumerator。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页
- “SADataReader 类”一节第 297 页

GetFieldType 方法

返回作为对象的数据类型的 Type。

语法

Visual Basic

```
Public Overrides Function GetFieldType( _
   ByVal index As Integer _
) As Type
```

C#

```
public override Type GetFieldType(
   int index
);
```

参数

- **index** 从零开始的列顺序号。

返回值

说明对象数据类型的 Type。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

GetFloat 方法

以单精度浮点数形式返回指定列的值。

语法

Visual Basic

```
Public Overrides Function GetFloat( _
   ByVal ordinal As Integer _
) As Single
```

C#

```
public override float  GetFloat(
```



```
  int ordinal
);
```

参数

- **ordinal** 表示获取值的来源列的顺序号。编号从零开始。

返回值

指定列的值。

注释

不会进行任何转换，因此检索的数据必须已经是单精度浮点数。

在调用此方法之前，调用 SADataReader.IsDBNull 方法以检查是否存在空值。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页
- “IsDBNull 方法”一节第 325 页

GetGuid 方法

以全局唯一标识符（global unique identifier，简称 GUID）形式返回指定列的值。

语法

Visual Basic

```
Public Overrides Function GetGuid( _
  ByVal ordinal As Integer _
) As Guid
```

C#

```
public override Guid GetGuid(
  int ordinal
);
```

参数

- **ordinal** 表示获取值的来源列的顺序号。编号从零开始。

返回值

指定列的值。

注释

检索的数据必须已经是全局唯一的标识符或 binary(16) 值。

在调用此方法之前，调用 SADataReader.IsDBNull 方法以检查是否存在空值。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页
- “IsDBNull 方法”一节第 325 页

GetInt16 方法

以 16 位有符号整数形式返回指定列的值。

语法

Visual Basic

```
Public Overrides Function GetInt16( _
   ByVal ordinal As Integer _
) As Short
```

C#

```
public override short GetInt16(
   int ordinal
);
```

参数

- **ordinal** 表示获取值的来源列的顺序号。编号从零开始。

返回值

指定列的值。

注释

不会进行任何转换，因此检索的数据必须已经是 16 位有符号整数。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

GetInt32 方法

以 32 位有符号整数形式返回指定列的值。

语法

Visual Basic

```
Public Overrides Function GetInt32( _
   ByVal ordinal As Integer _
) As Integer
```


C#

```
public override int GetInt32(
  int ordinal
);
```

参数

- **ordinal** 表示获取值的来源列的顺序号。编号从零开始。

返回值

指定列的值。

注释

不会进行任何转换，因此检索的数据必须已经是 32 位有符号整数。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

GetInt64 方法

以 64 位有符号整数形式返回指定列的值。

语法

Visual Basic

```
Public Overrides Function GetInt64( _
  ByVal ordinal As Integer _
) As Long
```

C#

```
public override long GetInt64(
  int ordinal
);
```

参数

- **ordinal** 表示获取值的来源列的顺序号。编号从零开始。

返回值

指定列的值。

注释

不会进行任何转换，因此检索的数据必须已经是 64 位有符号整数。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

GetName 方法

返回指定列的名称。

语法

Visual Basic

```
Public Overrides Function GetName( _
  ByVal index As Integer _
) As String
```

C#

```
public override string  GetName(
  int index
);
```

参数

- **index**　从零开始的列索引。

返回值

指定列的名称。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

GetOrdinal 方法

在已知列名的情况下返回列顺序号。

语法

Visual Basic

```
Public Overrides Function GetOrdinal( _
  ByVal name As String _
) As Integer
```

C#

```
public override int GetOrdinal(
  string  name
);
```


参数

- **name** 列名称。

返回值

从零开始的列顺序号。

注释

GetOrdinal 首先执行区分大小写的查找，如果失败，再进行不区分大小写的搜索。

GetOrdinal 不区分日语假名长度。

因为基于顺序号的查找比命名查找效率更高，因此在循环内调用 GetOrdinal 效率不高。可以通过调用一次 GetOrdinal，并将结果指派给一个整数变量以供在循环内使用来节约时间。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

GetSchemaTable 方法

返回说明 SADataReader 的列元数据的 DataTable。

语法

Visual Basic

```
Public Overrides Function GetSchemaTable() As DataTable
```

C#

```
public override DataTable GetSchemaTable();
```

返回值

描述列元数据的 DataTable。

注释

此方法按以下顺序返回关于每个列的元数据：

DataTable 列	说明
ColumnName	列的名称；如果该列没有名称，则为空值引用（在 Visual Basic 中是 Nothing）。如果该列在 SQL 查询中使用别名，则返回该别名。请注意，结果集中并非所有列都有名称，并且并非所有列名都是唯一的。
ColumnOrdinal	列的 ID。该值在 [0, FieldCount -1] 范围内。

DataTable 列	说明
ColumnSize	对于指定大小的列，为列中值的最大长度。对于其它列，为以字节表示的数据类型大小。
NumericPrecision	数值列的精度；如果该列不是数值列，则为 DBNull。
NumericScale	数值列的小数位数；如果该列不是数值列，则为 DBNull。
IsUnique	如果该列在提取它的表 (BaseTableName) 中是非计算唯一列，则为 true。
IsKey	如果该列是从结果集的唯一键中一起提取出来的一组列中的一列，则为 true。IsKey 设置为 true 的列集不必是作为行在结果集内唯一标识的最小集。
BaseServerName	SADataReader 使用的 SQL Anywhere 数据库服务器的名称。
BaseCatalogName	数据库中包含该列的目录的名称。此值始终为 DBNull。
BaseColumnName	数据库表 BaseTableName 中该列的原始名称；如果该列为计算列或无法确定此信息，则为 DBNull。
BaseSchemaName	数据库中包含该列的模式的名称。
BaseTableName	数据库中包含该列的表的名称；如果该列为计算列或无法确定此信息，则为 DBNull。
DataType	最适合此类型列的 .NET 数据类型。
AllowDBNull	如果该列可以为空，则为 true；如果该列不可以为空或无法确定此信息，则为 false。
ProviderType	列的类型。
IsAliased	如果列名是别名，则为 true；如果不是别名，则为 false。
IsExpression	如果该列是表达式，则为 true；如果是列值，则为 false。
IsIdentity	如果该列是标识列，则为 true；如果该列不是标识列，则为 false。
IsAutoIncrement	如果该列是自动增量列或全局自动增量列，则为 true；否则（或无法确定此信息时）为 false。
IsRowVersion	如果该列包含无法写入的持久性行标识符，并且该标识符除了标识行以外没有其它有意义的价值，则为 true。
IsHidden	如果列是隐藏的，则为 true；否则为 false。

DataTable 列	说明
IsLong	如果该列属于 long varchar、long nvarchar 或 long binary 列，则为 true；否则为 false。
IsReadOnly	如果该列为只读，则为 true；如果该列可修改或无法确定其访问权限，则为 false。

有关这些列的详细信息，请参见关于 SqlDataReader.GetSchemaTable 的 .NET Framework 文档。

有关详细信息，请参见“获取 DataReader 模式信息”一节第 116 页。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

GetString 方法

以字符串形式返回指定列的值。

语法

Visual Basic

```
Public Overrides Function GetString( _
   ByVal ordinal As Integer _
) As String
```

C#

```
public override string  GetString(
   int ordinal
);
```

参数

- **ordinal**　表示获取值的来源列的顺序号。编号从零开始。

返回值

指定列的值。

注释

不会进行任何转换，因此检索的数据必须已经是字符串。

在调用此方法之前，调用 SADataReader.IsDBNull 方法以检查 NULL 值。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页
- “IsDBNull 方法”一节第 325 页

GetTimeSpan 方法

以 TimeSpan 对象形式返回指定列的值。

语法

Visual Basic

```
Public Function GetTimeSpan( _
   ByVal ordinal As Integer _
) As TimeSpan
```

C#

```
public TimeSpan GetTimeSpan(
   int ordinal
);
```

参数

- **ordinal** 表示获取值的来源列的顺序号。编号从零开始。

返回值

指定列的值。

注释

列的数据类型必须是 SQL Anywhere TIME。数据将转换为 TimeSpan。TimeSpan 的 Days 属性始终设置为 0。

在调用此方法之前，调用 SADataReader.IsDBNull 方法以检查 NULL 值。

有关详细信息，请参见“获取时间值”一节第 126 页。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页
- “IsDBNull 方法”一节第 325 页

GetUInt16 方法

以 16 位无符号整数形式返回指定列的值。

语法

Visual Basic

```
Public Function GetUInt16( _
  ByVal ordinal As Integer _
) As UInt16
```

C#

```
public ushort GetUInt16(
  int ordinal
);
```

参数

- **ordinal** 表示获取值的来源列的顺序号。编号从零开始。

返回值

指定列的值。

注释

不会进行任何转换，因此检索的数据必须已经是 16 位无符号整数。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

GetUInt32 方法

以 32 位无符号整数形式返回指定列的值。

语法

Visual Basic

```
Public Function GetUInt32( _
  ByVal ordinal As Integer _
) As UInt32
```

C#

```
public uint GetUInt32(
  int ordinal
);
```

参数

- **ordinal** 表示获取值的来源列的顺序号。编号从零开始。

返回值

指定列的值。

注释

不会进行任何转换，因此检索的数据必须已经是 32 位无符号整数。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

GetUInt64 方法

以 64 位无符号整数形式返回指定列的值。

语法

Visual Basic

```
Public Function GetUInt64( _
   ByVal ordinal As Integer _
) As UInt64
```

C#

```
public ulong GetUInt64(
   int ordinal
);
```

参数

- **ordinal** 表示获取值的来源列的顺序号。编号从零开始。

返回值

指定列的值。

注释

不会进行任何转换，因此检索的数据必须已经是 64 位无符号整数。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

GetValue 方法

以对象形式返回指定列的值。

GetValue(Int32) 方法

以对象形式返回指定列的值。

语法

Visual Basic

```
Public Overrides Function GetValue( _
  ByVal ordinal As Integer _
) As Object
```

C#

```
public override object GetValue(
  int ordinal
);
```

参数

- **ordinal** 表示获取值的来源列的顺序号。编号从零开始。

返回值

对象形式的指定列的值。

注释

对于值为 NULL 的数据库列，此方法返回 DBNull。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页
- “GetValue 方法”一节第 322 页

GetValue(Int32, Int64, Int32) 方法

以对象形式返回指定列的子串值。

语法

Visual Basic

```
Public Function GetValue( _
  ByVal ordinal As Integer, _
  ByVal index As Long, _
  ByVal length As Integer _
) As Object
```

C#

```
public object GetValue(
  int ordinal,
  long index,
  int length
);
```

参数

- **ordinal** 表示获取值的来源列的顺序号。编号从零开始。
- **index** 要获得的子串值的索引（从零开始）。
- **length** 要获得的子串值的长度。

返回值

以对象形式返回的子串值。

注释

对于值为 NULL 的数据库列，此方法返回 DBNull。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页
- “GetValue 方法”一节第 322 页

GetValues 方法

获取当前行中的所有列。

语法

Visual Basic

```
Public Overrides Function GetValues( _
   ByVal values As Object() _
) As Integer
```

C#

```
public override int GetValues(
   object[] values
);
```

参数

- **values** 保存结果集中整个行的对象数组。

返回值

数组中的对象数。

注释

对大多数应用程序而言，GetValues 方法提供了一种有效率的检索所有列而不是分别检索每个列的方法。

您可以传递一个包含的列数比生成的行中包含的列数少的 Object 数组。只有 Object 数组保存的数据才会复制到数组中。您还可以传递长度比生成的行中包含的列数大的 Object 数组。

对于值为 NULL 的数据库列，此方法返回 DBNull。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

IsDBNull 方法

返回表示列是否包含 NULL 值的值。

语法

Visual Basic

```
Public Overrides Function IsDBNull( _
   ByVal ordinal As Integer _
) As Boolean
```

C#

```
public override bool IsDBNull(
   int ordinal
);
```

参数

- **ordinal** 从零开始的列顺序号。

返回值

如果指定的列值与 DBNull 等价，则返回 true。否则返回 false。

注释

在调用有类型 get 方法（如 GetByte、GetChar 等）之前，请调用此方法检查是否有 NULL 列值，以避免抛出异常。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

NextResult 方法

读取批处理 SQL 语句的结果时，将 SADataReader 移到下一结果。

语法

Visual Basic

```
Public Overrides Function NextResult() As Boolean
```

C#

public override bool **NextResult();**

返回值

如果还有其它结果集，则返回 true。否则返回 false。

注释

用于处理多个结果，这些结果可以通过执行批处理 SQL 语句来生成。

缺省情况下，数据读取器位于第一个结果上。

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

Read 方法

读取结果集的下一行，并将 SADataReader 移动到该行。

语法

Visual Basic

Public Overrides Function **Read()** As Boolean

C#

public override bool **Read();**

返回值

如果还有其它行，则返回 true。否则返回 false。

注释

SADataReader 的缺省位置是在第一个记录之前。因此，必须调用 Read 才能开始对任何数据进行访问。

示例

以下代码使用结果中一列的值填充列表框。

```
while( reader.Read() )
{
    listResults.Items.Add(
        reader.GetValue( 0 ).ToString() );
}
listResults.EndUpdate();
reader.Close();
```


另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

myDispose 方法

释放与该对象关联的资源。

语法

Visual Basic

Public Sub **myDispose()**

C#

public void **myDispose();**

另请参见

- “SADataReader 类”一节第 297 页
- “SADataReader 成员”一节第 297 页

SADataSourceEnumerator 类

提供枚举本地网络内所有可用 SQL Anywhere 数据库服务器实例的机制。此类无法继承。

语法

Visual Basic

Public NotInheritable Class **SADataSourceEnumerator**
  Inherits DbDataSourceEnumerator

C#

public sealed class **SADataSourceEnumerator** : DbDataSourceEnumerator

注释

没有用于 SADataSourceEnumerator 的构造函数。

无法在 .NET Compact Framework 2.0 中使用 SADataSourceEnumerator 类。

另请参见

- “SADataSourceEnumerator 成员”一节第 328 页

SADataSourceEnumerator 成员

公共属性

成员名称	说明
Instance 属性	获取 SADataSourceEnumberator 的实例，该实例可用于检索所有可见 SQL Anywhere 数据库服务器的相关信息。

公共方法

成员名称	说明
GetDataSources 方法	检索包含所有可见 SQL Anywhere 数据库服务器相关信息的 DataTable。

另请参见

- “SADataSourceEnumerator 类”一节第 327 页

Instance 属性

获取 SADataSourceEnumberator 的实例，该实例可用于检索所有可见 SQL Anywhere 数据库服务器的相关信息。

语法

Visual Basic

```
Public Shared Readonly Property Instance As SADataSourceEnumerator
```

C#

```
public const SADataSourceEnumerator Instance { get;}
```

另请参见

- “SADataSourceEnumerator 类”一节第 327 页
- “SADataSourceEnumerator 成员”一节第 328 页

GetDataSources 方法

检索包含所有可见 SQL Anywhere 数据库服务器相关信息的 DataTable。

语法

Visual Basic

```
Public Overrides Function GetDataSources() As DataTable
```


C#

public override DataTable **GetDataSources();**

注释

返回的表包含四列：ServerName、IPAddress、PortNumber 和 DataBaseNames。对于每个可用的数据库服务器，表中都有一行与其对应。

示例

以下代码使用每台可用数据库服务器的信息填充 DataTable。

```
DataTable servers = SADataSourceEnumerator.Instance.GetDataSources();
```

另请参见

- “SADataSourceEnumerator 类”一节第 327 页
- “SADataSourceEnumerator 成员”一节第 328 页

SADbType 枚举

枚举 SQL Anywhere .NET 数据库数据类型。

语法

Visual Basic

Public Enum **SADbType**

C#

public enum **SADbType**

注释

下表列出了与每种 SADbType 兼容的 .NET 类型。对于整型而言，表列始终都可以使用较小的整型来进行设置；但只要实际值在该类型取值范围之内，也可以使用较大的整型来进行设置。

SADbType	兼容的 .NET 类型	C# 内置类型	Visual Basic 内置类型
BigInt	System.Int64	long	Long
Binary, **VarBinary**	System.Byte[]，或 System.Guid（如果大小为 16）	byte[ ]	Byte()
Bit	System.Boolean	bool	Boolean
Char, **VarChar**	System.String	String	String

SADbType	兼容的 .NET 类型	C# 内置类型	Visual Basic 内置类型
Date	System.DateTime	DateTime（无内置类型）	Date
DateTime, **TimeStamp**	System.DateTime	DateTime（无内置类型）	Date
Decimal, **Numeric**	System.String	decimal	Decimal
Double	System.Double	double	Double
Float, **Real**	System.Single	float	Single
Image	System.Byte[]	byte[ ]	Byte()
Integer	System.Int32	int	Integer
LongBinary	System.Byte[]	byte[ ]	Byte()
LongNVarChar	System.String	String	String
LongVarChar	System.String	String	String
Money	System.String	decimal	Decimal
NChar	System.String	String	String
NText	System.String	String	String
Numeric	System.String	decimal	Decimal
NVarChar	System.String	String	String
SmallDateTime	System.DateTime	DateTime（无内置类型）	Date
SmallInt	System.Int16	short	Short
SmallMoney	System.String	decimal	Decimal
SysName	System.String	String	String
Text	System.String	String	String
Time	System.TimeSpan	TimeSpan（无内置类型）	TimeSpan（无内置类型）

SADbType	兼容的 .NET 类型	C# 内置类型	Visual Basic 内置类型
TimeStamp	System.DateTime	DateTime（无内置类型）	Date
TinyInt	System.Byte	byte	Byte
UniqueIdentifier	System.Guid	Guid（无内置类型）	Guid（无内置类型）
UniqueIdentifierStr	System.String	String	String
UnsignedBigInt	System.UInt64	ulong	UInt64（无内置类型）
UnsignedInt	System.UInt32	uint	UInt64（无内置类型）
UnsignedSmallInt	System.UInt16	ushort	UInt64（无内置类型）
Xml	System.System.Xml	String	String

长度为 16 的二进制列与 UniqueIdentifier 类型完全兼容。

成员

成员名称	说明	值
BigInt	有符号 64 位整数。	1
Binary	二进制数据，具有指定的最大长度。枚举值 Binary 和 VarBinary 互为对方的别名。	2
Bit	1 位标志。	3
Char	字符数据，具有指定长度。此类型始终支持 Unicode 字符。类型 Char 与 VarChar 完全兼容。	4
Date	日期信息。	5
DateTime	时间戳信息（日期、时间）。枚举值 DateTime 和 TimeStamp 互为对方的别名。	6
Decimal	精确数字数据，具有指定的精度和小数位数。枚举值 Decimal 和 Numeric 互为对方的别名。	7
Double	双精度浮点数（8 个字节）。	8

成员名称	说明	值
Float	单精度浮点数（4 个字节）。枚举值 Float 和 Real 互为对方的别名。	9
Image	存储任意长度的二进制数据。	10
Integer	无符号 32 位整数。	11
LongBinary	二进制数据，具有可变长度。	12
LongNVarchar	NCHAR 字符集中的字符数据，具有可变长度。此类型始终支持 Unicode 字符。	13
LongVarbit	具有可变长度的位数组。	14
LongVarchar	字符数据，具有可变长度。此类型始终支持 Unicode 字符。	15
Money	货币数据。	16
NChar	存储 Unicode 字符数据，最多可存储 8191 个字符。	17
NText	存储任意长度的 Unicode 字符数据。	18
Numeric	精确数字数据，具有指定的精度和小数位数。枚举值 Decimal 和 Numeric 互为对方的别名。	19
NVarChar	存储 Unicode 字符数据，最多可存储 8191 个字符。	20
Real	单精度浮点数（4 个字节）。枚举值 Float 和 Real 互为对方的别名。	21
SmallDateTime	一个域，以 TIMESTAMP 形式实现。	22
SmallInt	有符号 16 位整数。	23
SmallMoney	存储不足一百万货币单位的货币数据。	24
SysName	存储任意长度的字符数据。	25
Text	存储任意长度的字符数据。	26
Time	时间信息。	27
TimeStamp	时间戳信息（日期、时间）。枚举值 DateTime 和 TimeStamp 互为对方的别名。	28

成员名称	说明	值
TinyInt	无符号 8 位整数。	29
UniqueIdentifier	通用唯一标识符（Universally Unique Identifier，简称 UUID/GUID）。	30
UniqueIdentifierStr	一个域，以 CHAR(36) 形式实现。UniqueIdentifierStr 用于在映射 Microsoft SQL Server uniqueidentifier 列时进行远程数据访问。	31
UnsignedBigInt	无符号 64 位整数。	32
UnsignedInt	无符号 32 位整数。	33
UnsignedSmallInt	无符号 16 位整数。	34
VarBinary	二进制数据，具有指定的最大长度。枚举值 **Binary** 和 **VarBinary** 互为对方的别名。	35
VarBit	位数组的长度是从 1 到 32767 位之间的值。	36
VarChar	字符数据，具有指定的最大长度。此类型始终支持 Unicode 字符。类型 Char 与 VarChar 完全兼容。	37
Xml	XML 数据。此类型可存储任意长度的字符数据，也用来存储 XML 文档。	38

另请参见

- “GetFieldType 方法”一节第 312 页
- “GetDataTypeName 方法”一节第 308 页

SADefault 类

表示具有缺省值的参数。此类属于静态类，因此无法继承，也无法将其实例化。

语法

Visual Basic

Public NotInheritable Class **SADefault**

C#

public sealed class **SADefault**

注释

没有用于 SADefault 的构造函数。

```
SAParameter parm = new SAParameter();
parm.Value = SADefault.Value;
```

另请参见

- “SADefault 成员”一节第 334 页

SADefault 成员

公共字段

成员名称	说明
Value 字段	获取缺省参数的值。此字段是静态只读字段。此字段为只读字段。

另请参见

- “SADefault 类”一节第 333 页

Value 字段

获取缺省参数的值。此字段是静态只读字段。此字段为只读字段。

语法

Visual Basic

Public Shared Readonly **Value** As SADefault

C#

public const SADefault **Value** ;

另请参见

- “SADefault 类”一节第 333 页
- “SADefault 成员”一节第 334 页

SAError 类

收集与数据源返回的警告或错误有关的信息。此类无法继承。

语法

Visual Basic

Public NotInheritable Class **SAError**

C#

public sealed class **SAError**

注释

没有用于 SAError 的构造函数。

有关错误处理的信息，请参见“出错处理和 SQL Anywhere .NET 数据提供程序”一节第 131 页。

另请参见

- “SAError 成员”一节第 335 页

SAError 成员

公共属性

成员名称	说明
Message 属性	返回对错误的简短说明。
NativeError 属性	返回数据库特定的错误信息。
Source 属性	返回生成该错误的提供程序的名称。
SqlState 属性	遵循 ANSI SQL 标准的 SQL Anywhere 五字符 SQLSTATE。

公共方法

成员名称	说明
ToString 方法	错误消息的完整文本。

另请参见

- “SAError 类”一节第 334 页

Message 属性

返回对错误的简短说明。

语法

Visual Basic

Public Readonly Property **Message** As String

C#

public string **Message** { get;}

另请参见

- “SAError 类”一节第 334 页
- “SAError 成员”一节第 335 页

NativeError 属性

返回数据库特定的错误信息。

语法

Visual Basic

Public Readonly Property **NativeError** As Integer

C#

public int **NativeError** { get;}

另请参见

- “SAError 类”一节第 334 页
- “SAError 成员”一节第 335 页

Source 属性

返回生成该错误的提供程序的名称。

语法

Visual Basic

Public Readonly Property **Source** As String

C#

public string **Source** { get;}

另请参见

- “SAError 类”一节第 334 页
- “SAError 成员”一节第 335 页

SqlState 属性

遵循 ANSI SQL 标准的 SQL Anywhere 五字符 SQLSTATE。

语法

Visual Basic

Public Readonly Property **SqlState** As String

C#

public string **SqlState** { get;}

另请参见

- “SAError 类”一节第 334 页
- “SAError 成员”一节第 335 页

ToString 方法

错误消息的完整文本。

语法

Visual Basic

Public Overrides Function **ToString()** As String

C#

public override string **ToString();**

示例

返回值是一个前为 **SAError:**，后跟消息形式的字符串。例如：

```
SAError:UserId or Password not valid.
```

另请参见

- “SAError 类”一节第 334 页
- “SAError 成员”一节第 335 页

SAErrorCollection 类

收集 SQL Anywhere .NET 数据提供程序生成的所有错误。此类无法继承。

语法

Visual Basic

```
Public NotInheritable Class SAErrorCollection
  Implements ICollection, IEnumerable
```

C#

```
public sealed class SAErrorCollection : ICollection, IEnumerable
```

注释

没有用于 SAErrorCollection 的构造函数。SAErrorCollection 通常是从 SAException.Errors 属性获取的。

Implements:ICollection、IEnumerable

有关错误处理的信息，请参见“出错处理和 SQL Anywhere .NET 数据提供程序”一节第 131 页。

另请参见

- “SAErrorCollection 成员”一节第 338 页
- “Errors 属性”一节第 342 页
- SqlClientFactory.CanCreateDataSourceEnumerator

SAErrorCollection 成员

公共属性

成员名称	说明
Count 属性	返回集合中的错误数。
Item 属性	返回指定索引处的错误。

公共方法

成员名称	说明
CopyTo 方法	将 SAErrorCollection 的元素复制到数组中（从数组内的给定索引开始）。
GetEnumerator 方法	返回迭代通过 SAErrorCollection 的枚举器。

另请参见

- “SAErrorCollection 类”一节第 337 页
- “Errors 属性”一节第 342 页
- SqlClientFactory.CanCreateDataSourceEnumerator

Count 属性

返回集合中的错误数。

语法

Visual Basic

```
NotOverridable Public Readonly Property Count As Integer
```

C#

```
public int Count { get;}
```

另请参见

- “SAErrorCollection 类”一节第 337 页
- “SAErrorCollection 成员”一节第 338 页

Item 属性

返回指定索引处的错误。

语法

Visual Basic

```
Public Readonly Property Item ( _
   ByVal index As Integer _
) As SAError
```

C#

```
public SAError this [
   int index
] { get;}
```

参数

- **index** 要检索的错误的从零开始索引。

属性值

包含指定索引处的错误的 SAError 对象。

另请参见

- “SAErrorCollection 类”一节第 337 页
- “SAErrorCollection 成员”一节第 338 页
- “SAError 类”一节第 334 页

CopyTo 方法

将 SAErrorCollection 的元素复制到数组中（从数组内的给定索引开始）。

语法

Visual Basic

```
NotOverridable Public Sub CopyTo( _
  ByVal array As Array, _
  ByVal index As Integer _
)
```

C#

```
public void CopyTo(
  Array array,
  int index
);
```

参数

- **array** 接收所复制元素的数组。
- **index** 数组的起始索引。

另请参见

- “SAErrorCollection 类”一节第 337 页
- “SAErrorCollection 成员”一节第 338 页

GetEnumerator 方法

返回迭代通过 SAErrorCollection 的枚举器。

语法

Visual Basic

```
NotOverridable Public Function GetEnumerator() As IEnumerator
```

C#

```
public IEnumerator GetEnumerator();
```

返回值

SAErrorCollection 的 IEnumerator。

另请参见

- “SAErrorCollection 类”一节第 337 页
- “SAErrorCollection 成员”一节第 338 页

SAException 类

SQL Anywhere 返回警告或错误时抛出的异常。

语法

Visual Basic

Public Class **SAException**
  Inherits DbException

C#

public class **SAException** : DbException

注释

没有用于 SAException 的构造函数。SAException 对象通常是在捕获中进行声明。例如：

```
...
catch( SAException ex )
{
    MessageBox.Show( ex.Errors[0].Message, "Error" );
}
```

有关错误处理的信息，请参见“出错处理和 SQL Anywhere .NET 数据提供程序”一节第 131 页。

另请参见

- “SAException 成员”一节第 341 页

SAException 成员

公共属性

成员名称	说明
Data（继承自 Exception）	获取可提供有关异常的附加用户定义信息的键/值对集合。
ErrorCode（继承自 ExternalException）	获取错误的 HRESULT。
Errors 属性	返回包含一个或多个“SAError 类”一节第 334 页对象的集合。
HelpLink（继承自 Exception）	获取或设置可转到与此异常相关联的帮助文件的链接。
InnerException（继承自 Exception）	获取导致当前异常的异常实例。
Message 属性	返回说明错误的文本。

成员名称	说明
NativeError 属性	返回数据库特定的错误信息。
Source 属性	返回生成该错误的提供程序的名称。
StackTrace（继承自 Exception）	获取在抛出当前异常时调用堆栈上的框架的字符串表示形式。
TargetSite（继承自 Exception）	获取抛出当前异常的方法。

公共方法

成员名称	说明
GetBaseException（继承自 Exception）	当在派生类中被替换时，返回 Exception，它是导致产生一个或多个后续异常的根本原因。
GetObjectData 方法	使用有关该异常的信息来设置 SerializationInfo。替换 Exception.GetObjectData。
GetType（继承自 Exception）	获取当前实例的运行时类型。
ToString（继承自 Exception）	创建并返回当前异常的字符串表示形式。

另请参见

- “SAException 类”一节第 341 页

Errors 属性

返回包含一个或多个“SAError 类”一节第 334 页对象的集合。

语法

Visual Basic

Public Readonly Property **Errors** As SAErrorCollection

C#

public SAErrorCollection **Errors** { get;}

注释

SAErrorCollection 对象总是至少包含 SAError 对象的一个实例。

另请参见

- “SAException 类”一节第 341 页
- “SAException 成员”一节第 341 页
- “SAErrorCollection 类”一节第 337 页
- “SAError 类”一节第 334 页

Message 属性

返回说明错误的文本。

语法

Visual Basic

Public Overrides Readonly Property **Message** As String

C#

public override string **Message** { get;}

注释

此方法返回单个字符串，其中包含一连串 Errors 集合中所有 SAError 对象的所有 Message 属性。除最后一个消息外，所有其它消息后都跟有一个回车。

另请参见

- “SAException 类”一节第 341 页
- “SAException 成员”一节第 341 页
- “SAError 类”一节第 334 页

NativeError 属性

返回数据库特定的错误信息。

语法

Visual Basic

Public Readonly Property **NativeError** As Integer

C#

public int **NativeError** { get;}

另请参见

- “SAException 类”一节第 341 页
- “SAException 成员”一节第 341 页

Source 属性

返回生成该错误的提供程序的名称。

语法

Visual Basic

```
Public Overrides Readonly Property Source As String
```

C#

```
public override string  Source { get;}
```

另请参见

- “SAException 类”一节第 341 页
- “SAException 成员”一节第 341 页

GetObjectData 方法

使用有关该异常的信息来设置 SerializationInfo。替换 Exception.GetObjectData。

语法

Visual Basic

```
Public Overrides Sub GetObjectData( _
  ByVal info As SerializationInfo, _
  ByVal context As StreamingContext _
)
```

C#

```
public override void GetObjectData(
  SerializationInfo info,
  StreamingContext context
);
```

参数

- **info**　保存有关抛出的异常的序列化对象数据的 SerializationInfo。
- **context**　包含有关源或目标的上下文信息的 StreamingContext。

另请参见

- “SAException 类”一节第 341 页
- “SAException 成员”一节第 341 页

SAFactory 类

表示一组方法，这些方法用于创建 iAnywhere.Data.SQLAnywhere 提供程序对数据源类实现的实例。此类属于静态类，因此无法继承，也无法将其实例化。

语法

Visual Basic

```
Public NotInheritable Class SAFactory
  Inherits DbProviderFactory
  Implements IServiceProvider
```

C#

```
public sealed class SAFactory : DbProviderFactory,
  IServiceProvider
```

注释

没有用于 SAFactory 的构造函数。

ADO.NET 2.0 添加了两个新类 DbProviderFactories 和 DbProviderFactory，以简化提供程序无关代码的编写。要将它们与 SQL Anywhere 配合使用，请将 iAnywhere.Data.SQLAnywhere 指定为传递给 GetFactory 的提供程序无关名。例如：

```
' Visual Basic
Dim factory As DbProviderFactory =
  DbProviderFactories.GetFactory( "iAnywhere.Data.SQLAnywhere" )
Dim conn As DbConnection =
  factory.CreateConnection()
// C#
DbProviderFactory factory =
        DbProviderFactories.GetFactory("iAnywhere.Data.SQLAnywhere" );
DbConnection conn = factory.CreateConnection();
```

在此示例中，以 SAConnection 对象形式创建了 conn。

有关 ADO.NET 2.0 中提供程序工厂和通用编程的说明，请参见 http://msdn2.microsoft.com/zh-cn/library/ms379620.aspx。

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SAFactory 类。

Inherits: DbProviderFactory

另请参见

- “SAFactory 成员”一节第 346 页

SAFactory 成员

公共字段

成员名称	说明
Instance 字段	表示 SAFactory 类的单个实例。此字段为只读字段。

公共属性

成员名称	说明
CanCreateDataSourceEnumerator 属性	始终返回 true，它表示可以创建 SADataSourceEnumerator 对象。

公共方法

成员名称	说明
CreateCommand 方法	返回强类型的 DbCommand 实例。
CreateCommandBuilder 方法	返回强类型的 DbCommandBuilder 实例。
CreateConnection 方法	返回强类型的 DbConnection 实例。
CreateConnectionStringBuilder 方法	返回强类型的 DbConnectionStringBuilder 实例。
CreateDataAdapter 方法	返回强类型的 DbDataAdapter 实例。
CreateDataSourceEnumerator 方法	返回强类型的 DbDataSourceEnumerator 实例。
CreateParameter 方法	返回强类型的 DbParameter 实例。
CreatePermission 方法	返回强类型的 CodeAccessPermission 实例。

另请参见

- “SAFactory 类”一节第 345 页

Instance 字段

表示 SAFactory 类的单个实例。此字段为只读字段。

语法

Visual Basic

Public Shared Readonly **Instance** As SAFactory

C#

public const SAFactory **Instance** ;

注释

SAFactory 是一个单个类，这意味着只有此类的这个实例可以存在。

通常不会直接使用此字段，而是使用 DbProviderFactories.GetFactory 获取对此 SAFactory 实例的引用。有关示例，请参见 SAFactory 说明。

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SAFactory 类。

另请参见

- “SAFactory 类”一节第 345 页
- “SAFactory 成员”一节第 346 页
- “SAFactory 类”一节第 345 页

CanCreateDataSourceEnumerator 属性

始终返回 true，它表示可以创建 SADataSourceEnumerator 对象。

语法

Visual Basic

Public Overrides Readonly Property **CanCreateDataSourceEnumerator** As Boolean

C#

public override bool **CanCreateDataSourceEnumerator** { get;}

属性值

一个类型为 DbCommand 的新 SACommand 对象。

另请参见

- “SAFactory 类”一节第 345 页
- “SAFactory 成员”一节第 346 页
- “SADataSourceEnumerator 类”一节第 327 页
- “SACommand 类”一节第 195 页

CreateCommand 方法

返回强类型的 DbCommand 实例。

语法

Visual Basic

Public Overrides Function **CreateCommand()** As DbCommand

C#

public override DbCommand **CreateCommand();**

返回值

一个类型为 DbCommand 的新 SACommand 对象。

另请参见

- “SAFactory 类”一节第 345 页
- “SAFactory 成员”一节第 346 页
- “SACommand 类”一节第 195 页

CreateCommandBuilder 方法

返回强类型的 DbCommandBuilder 实例。

语法

Visual Basic

Public Overrides Function **CreateCommandBuilder()** As DbCommandBuilder

C#

public override DbCommandBuilder **CreateCommandBuilder();**

返回值

一个类型为 DbCommand 的新 SACommand 对象。

另请参见

- “SAFactory 类”一节第 345 页
- “SAFactory 成员”一节第 346 页
- “SACommand 类”一节第 195 页

CreateConnection 方法

返回强类型的 DbConnection 实例。

语法

Visual Basic

Public Overrides Function **CreateConnection()** As DbConnection

C#

public override DbConnection **CreateConnection();**

返回值

一个类型为 DbCommand 的新 SACommand 对象。

另请参见

- “SAFactory 类”一节第 345 页
- “SAFactory 成员”一节第 346 页
- “SACommand 类”一节第 195 页

CreateConnectionStringBuilder 方法

返回强类型的 DbConnectionStringBuilder 实例。

语法

Visual Basic

Public Overrides Function **CreateConnectionStringBuilder()** As DbConnectionStringBuilder

C#

public override DbConnectionstring Builder **CreateConnectionStringBuilder();**

返回值

一个类型为 DbCommand 的新 SACommand 对象。

另请参见

- “SAFactory 类”一节第 345 页
- “SAFactory 成员”一节第 346 页
- “SACommand 类”一节第 195 页

CreateDataAdapter 方法

返回强类型的 DbDataAdapter 实例。

语法

Visual Basic

Public Overrides Function **CreateDataAdapter()** As DbDataAdapter

C#

public override DbDataAdapter **CreateDataAdapter();**

返回值

一个类型为 DbCommand 的新 SACommand 对象。

另请参见

- “SAFactory 类”一节第 345 页
- “SAFactory 成员”一节第 346 页
- “SACommand 类”一节第 195 页

CreateDataSourceEnumerator 方法

返回强类型的 DbDataSourceEnumerator 实例。

语法

Visual Basic

Public Overrides Function **CreateDataSourceEnumerator()** As DbDataSourceEnumerator

C#

public override DbDataSourceEnumerator **CreateDataSourceEnumerator();**

返回值

一个类型为 DbCommand 的新 SACommand 对象。

另请参见

- “SAFactory 类”一节第 345 页
- “SAFactory 成员”一节第 346 页
- “SACommand 类”一节第 195 页

CreateParameter 方法

返回强类型的 DbParameter 实例。

语法

Visual Basic

Public Overrides Function **CreateParameter()** As DbParameter

C#

public override DbParameter **CreateParameter();**

返回值

一个类型为 DbCommand 的新 SACommand 对象。

另请参见

- “SAFactory 类”一节第 345 页
- “SAFactory 成员”一节第 346 页
- “SACommand 类”一节第 195 页

CreatePermission 方法

返回强类型的 CodeAccessPermission 实例。

语法

Visual Basic

```
Public Overrides Function CreatePermission( _
  ByVal state As PermissionState _
) As CodeAccessPermission
```

C#

```
public override CodeAccessPermission CreatePermission(
  PermissionState state
);
```

参数

- **state** PermissionState 枚举的一个成员。

返回值

一个类型为 DbCommand 的新 SACommand 对象。

另请参见

- “SAFactory 类”一节第 345 页
- “SAFactory 成员”一节第 346 页
- “SACommand 类”一节第 195 页

SAInfoMessageEventArgs 类

为 InfoMessage 事件提供数据。此类无法继承。

语法

Visual Basic

```
Public NotInheritable Class SAInfoMessageEventArgs
  Inherits EventArgs
```

C#

```
public sealed class SAInfoMessageEventArgs : EventArgs
```

注释

没有用于 SAInfoMessageEventArgs 的构造函数。

另请参见

- “SAInfoMessageEventArgs 成员”一节第 352 页

SAInfoMessageEventArgs 成员

公共属性

成员名称	说明
Errors 属性	返回从数据源发送的消息的集合。
Message 属性	返回从数据源发送的错误的完整文本。
MessageType 属性	返回消息的类型。此类型可以是以下各项之一：Action、Info、Status 或 Warning。
NativeError 属性	返回由数据库返回的 SQL 代码。
Source 属性	返回 SQL Anywhere .NET 数据提供程序的名称。

公共方法

成员名称	说明
ToString 方法	检索 InfoMessage 事件的字符串表示。

另请参见

- “SAInfoMessageEventArgs 类”一节第 351 页

Errors 属性

返回从数据源发送的消息的集合。

语法

Visual Basic

```
Public Readonly Property Errors As SAErrorCollection
```

C#

```
public SAErrorCollection Errors { get;}
```


另请参见

- “SAInfoMessageEventArgs 类”一节第 351 页
- “SAInfoMessageEventArgs 成员”一节第 352 页

Message 属性

返回从数据源发送的错误的完整文本。

语法

Visual Basic

Public Readonly Property **Message** As String

C#

public string **Message** { get;}

另请参见

- “SAInfoMessageEventArgs 类”一节第 351 页
- “SAInfoMessageEventArgs 成员”一节第 352 页

MessageType 属性

返回消息的类型。此类型可以是以下各项之一：Action、Info、Status 或 Warning。

语法

Visual Basic

Public Readonly Property **MessageType** As SAMessageType

C#

public SAMessageType **MessageType** { get;}

另请参见

- “SAInfoMessageEventArgs 类”一节第 351 页
- “SAInfoMessageEventArgs 成员”一节第 352 页

NativeError 属性

返回由数据库返回的 SQL 代码。

语法

Visual Basic

Public Readonly Property **NativeError** As Integer

C#

public int **NativeError** { get;}

另请参见

- “SAInfoMessageEventArgs 类”一节第 351 页
- “SAInfoMessageEventArgs 成员”一节第 352 页

Source 属性

返回 SQL Anywhere .NET 数据提供程序的名称。

语法

Visual Basic

Public Readonly Property **Source** As String

C#

public string **Source** { get;}

另请参见

- “SAInfoMessageEventArgs 类”一节第 351 页
- “SAInfoMessageEventArgs 成员”一节第 352 页

ToString 方法

检索 InfoMessage 事件的字符串表示。

语法

Visual Basic

Public Overrides Function **ToString()** As String

C#

public override string **ToString();**

返回值

表示 InfoMessage 事件的字符串。

另请参见

- “SAInfoMessageEventArgs 类”一节第 351 页
- “SAInfoMessageEventArgs 成员”一节第 352 页

SAInfoMessageEventHandler 委派

表示处理 SAConnection 对象的 SAConnection.InfoMessage 事件的方法。

语法

Visual Basic

```
Public Delegate Sub SAInfoMessageEventHandler( _
  ByVal obj As Object, _
  ByVal args As SAInfoMessageEventArgs _
)
```

C#

```
public delegate void SAInfoMessageEventHandler(
  object obj,
  SAInfoMessageEventArgs args
);
```

另请参见

- “SAConnection 类”一节第 234 页
- “InfoMessage 事件”一节第 255 页

SAIsolationLevel 枚举

指定 SQL Anywhere 隔离级别。此类会扩展 IsolationLevel。

语法

Visual Basic

```
Public Enum SAIsolationLevel
```

C#

```
public enum SAIsolationLevel
```

注释

SQL Anywhere .NET 数据提供程序支持所有 SQL Anywhere 隔离级别，包括快照隔离级别。要使用快照隔离，请指定 SAIsolationLevel.Snapshot、SAIsolationLevel.ReadOnlySnapshot 或 SAIsolationLevel.StatementSnapshot 之一作为 BeginTransaction 的参数。BeginTransaction 已经重载，因此它可以带有 IsolationLevel 或 SAIsolationLevel。这两个枚举中的值基本相同，只是

ReadOnlySnapshot 和 StatementSnapshot 仅存在于 SAIsolationLevel 中。名为 SAIsolationLevel 的 SATransaction 中有一个用于获取 SAIsolationLevel 的新属性。

有关详细信息，请参见“快照隔离”一节 《SQL Anywhere 服务器 - SQL 的用法》。

成员

成员名称	说明	值
Chaos	不支持此隔离级别。	16
ReadCommitted	将行为设置为等同于隔离级别 1。	4096
ReadOnlySnapshot	对于只读语句，从读取数据库的第一行时开始，使用已提交数据的快照。	16777217
ReadUncommitted	将行为设置为等同于隔离级别 0。	256
RepeatableRead	将行为设置为等同于隔离级别 2。	65536
Serializable	将行为设置为等同于隔离级别 3。	1048576
Snapshot	从事务读取、插入、更新或删除第一行时开始，使用已提交数据的快照。	16777216
StatementSnapshot	从语句读取第一行开始，使用已提交数据的快照。事务内的每个语句看到的都是不同时间的数据快照。	16777218
Unspecified	不支持此隔离级别。	-1

SAMessageType 枚举

标识消息的类型。此类型可以是以下各项之一：Action、Info、Status 或 Warning。

语法

Visual Basic

Public Enum **SAMessageType**

C#

public enum **SAMessageType**

成员

成员名称	说明	值
Action	ACTION 类型的消息。	2
Info	INFO 类型的消息。	0
Status	STATUS 类型的消息。	3
Warning	WARNING 类型的消息。	1

SAMetaDataCollectionNames 类

提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以检索元数据集合的常量列表。此类无法继承。

语法

Visual Basic

Public NotInheritable Class **SAMetaDataCollectionNames**

C#

public sealed class **SAMetaDataCollectionNames**

注释

此字段为常量并是只读字段。

另请参见

- “SAMetaDataCollectionNames 成员”一节第 357 页
- “GetSchema(String, String[]) 方法”一节第 250 页

SAMetaDataCollectionNames 成员

公共字段

成员名称	说明
Columns 字段	提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 Columns 集合的常量。此字段为只读字段。
DataSourceInformation 字段	提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 DataSourceInformation 集合的常量。此字段为只读字段。

成员名称	说明
DataTypes 字段	提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 DataTypes 集合的常量。此字段为只读字段。
ForeignKeys 字段	提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 ForeignKeys 集合的常量。此字段为只读字段。
IndexColumns 字段	提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 IndexColumns 集合的常量。此字段为只读字段。
Indexes 字段	提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 Indexes 集合的常量。此字段为只读字段。
MetaDataCollections 字段	提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 MetaDataCollections 集合的常量。此字段为只读字段。
ProcedureParameters 字段	提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 ProcedureParameters 集合的常量。此字段为只读字段。
Procedures 字段	提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 Procedures 集合的常量。此字段为只读字段。
ReservedWords 字段	提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 ReservedWords 集合的常量。此字段为只读字段。
Restrictions 字段	提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 Restrictions 集合的常量。此字段为只读字段。
Tables 字段	提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 Tables 集合的常量。此字段为只读字段。
UserDefinedTypes 字段	提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 UserDefinedTypes 集合的常量。此字段为只读字段。
Users 字段	提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 Users 集合的常量。此字段为只读字段。
ViewColumns 字段	提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 ViewColumns 集合的常量。此字段为只读字段。
Views 字段	提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 Views 集合的常量。此字段为只读字段。

另请参见

- “SAMetaDataCollectionNames 类”一节第 357 页
- “GetSchema(String, String[]) 方法”一节第 250 页

Columns 字段

提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 Columns 集合的常量。此字段为只读字段。

语法

Visual Basic

Public Shared Readonly **Columns** As String

C#

public const string **Columns** ;

示例

以下代码使用 Columns 集合填充 DataTable。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Columns );
```

另请参见

- “SAMetaDataCollectionNames 类”一节第 357 页
- “SAMetaDataCollectionNames 成员”一节第 357 页
- “GetSchema(String, String[]) 方法”一节第 250 页

DataSourceInformation 字段

提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 DataSourceInformation 集合的常量。此字段为只读字段。

语法

Visual Basic

Public Shared Readonly **DataSourceInformation** As String

C#

public const string **DataSourceInformation** ;

示例

以下代码使用 DataSourceInformation 集合填充 DataTable。

```
DataTable schema =
GetSchema( SAMetaDataCollectionNames.DataSourceInformation );
```

另请参见

- “SAMetaDataCollectionNames 类”一节第 357 页
- “SAMetaDataCollectionNames 成员”一节第 357 页
- “GetSchema(String, String[]) 方法”一节第 250 页

DataTypes 字段

提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 DataTypes 集合的常量。此字段为只读字段。

语法

Visual Basic

Public Shared Readonly **DataTypes** As String

C#

public const string **DataTypes** ;

示例

以下代码使用 DataTypes 集合填充 DataTable。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.DataTypes );
```

另请参见

- “SAMetaDataCollectionNames 类”一节第 357 页
- “SAMetaDataCollectionNames 成员”一节第 357 页
- “GetSchema(String, String[]) 方法”一节第 250 页

ForeignKeys 字段

提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 ForeignKeys 集合的常量。此字段为只读字段。

语法

Visual Basic

Public Shared Readonly **ForeignKeys** As String

C#

public const string **ForeignKeys** ;

示例

以下代码使用 ForeignKeys 集合填充 DataTable。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.ForeignKeys );
```

另请参见

- “SAMetaDataCollectionNames 类”一节第 357 页
- “SAMetaDataCollectionNames 成员”一节第 357 页
- “GetSchema(String, String[]) 方法”一节第 250 页

IndexColumns 字段

提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 IndexColumns 集合的常量。此字段为只读字段。

语法

Visual Basic

Public Shared Readonly **IndexColumns** As String

C#

public const string **IndexColumns** ;

示例

以下代码使用 IndexColumns 集合填充 DataTable。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.IndexColumns );
```

另请参见

- “SAMetaDataCollectionNames 类”一节第 357 页
- “SAMetaDataCollectionNames 成员”一节第 357 页
- “GetSchema(String, String[]) 方法”一节第 250 页

Indexes 字段

提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 Indexes 集合的常量。此字段为只读字段。

语法

Visual Basic

Public Shared Readonly **Indexes** As String

C#

public const string **Indexes** ;

示例

以下代码使用 Indexes 集合填充 DataTable。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Indexes );
```

另请参见

- “SAMetaDataCollectionNames 类”一节第 357 页
- “SAMetaDataCollectionNames 成员”一节第 357 页
- “GetSchema(String, String[]) 方法”一节第 250 页

MetaDataCollections 字段

提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 MetaDataCollections 集合的常量。此字段为只读字段。

语法

Visual Basic

Public Shared Readonly **MetaDataCollections** As String

C#

public const string **MetaDataCollections** ;

示例

以下代码使用 MetaDataCollections 集合填充 DataTable。

```
DataTable schema =
GetSchema( SAMetaDataCollectionNames.MetaDataCollections );
```

另请参见

- “SAMetaDataCollectionNames 类”一节第 357 页
- “SAMetaDataCollectionNames 成员”一节第 357 页
- “GetSchema(String, String[]) 方法”一节第 250 页

ProcedureParameters 字段

提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 ProcedureParameters 集合的常量。此字段为只读字段。

语法

Visual Basic

Public Shared Readonly **ProcedureParameters** As String

C#

public const string **ProcedureParameters** ;

示例

以下代码使用 ProcedureParameters 集合填充 DataTable。

```
DataTable schema =
GetSchema( SAMetaDataCollectionNames.ProcedureParameters );
```


另请参见

- “SAMetaDataCollectionNames 类”一节第 357 页
- “SAMetaDataCollectionNames 成员”一节第 357 页
- “GetSchema(String, String[]) 方法”一节第 250 页

Procedures 字段

提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 Procedures 集合的常量。此字段为只读字段。

语法

Visual Basic

Public Shared Readonly **Procedures** As String

C#

public const string **Procedures** ;

示例

以下代码使用 Procedures 集合填充 DataTable。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Procedures );
```

另请参见

- “SAMetaDataCollectionNames 类”一节第 357 页
- “SAMetaDataCollectionNames 成员”一节第 357 页
- “GetSchema(String, String[]) 方法”一节第 250 页

ReservedWords 字段

提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 ReservedWords 集合的常量。此字段为只读字段。

语法

Visual Basic

Public Shared Readonly **ReservedWords** As String

C#

public const string **ReservedWords** ;

示例

以下代码使用 ReservedWords 集合填充 DataTable。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.ReservedWords );
```

另请参见

- “SAMetaDataCollectionNames 类”一节第 357 页
- “SAMetaDataCollectionNames 成员”一节第 357 页
- “GetSchema(String, String[]) 方法”一节第 250 页

Restrictions 字段

提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 Restrictions 集合的常量。此字段为只读字段。

语法

Visual Basic

Public Shared Readonly **Restrictions** As String

C#

public const string **Restrictions** ;

示例

以下代码使用 Restrictions 集合填充 DataTable。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Restrictions );
```

另请参见

- “SAMetaDataCollectionNames 类”一节第 357 页
- “SAMetaDataCollectionNames 成员”一节第 357 页
- “GetSchema(String, String[]) 方法”一节第 250 页

Tables 字段

提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 Tables 集合的常量。此字段为只读字段。

语法

Visual Basic

Public Shared Readonly **Tables** As String

C#

public const string **Tables** ;

示例

以下代码使用 Tables 集合填充 DataTable。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Tables );
```


另请参见

- “SAMetaDataCollectionNames 类”一节第 357 页
- “SAMetaDataCollectionNames 成员”一节第 357 页
- “GetSchema(String, String[]) 方法”一节第 250 页

UserDefinedTypes 字段

提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 UserDefinedTypes 集合的常量。此字段为只读字段。

语法

Visual Basic

Public Shared Readonly **UserDefinedTypes** As String

C#

public const string **UserDefinedTypes** ;

示例

以下代码使用 Users 集合填充 DataTable。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.UserDefinedTypes );
```

另请参见

- “SAMetaDataCollectionNames 类”一节第 357 页
- “SAMetaDataCollectionNames 成员”一节第 357 页
- “GetSchema(String, String[]) 方法”一节第 250 页

Users 字段

提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 Users 集合的常量。此字段为只读字段。

语法

Visual Basic

Public Shared Readonly **Users** As String

C#

public const string **Users** ;

示例

以下代码使用 Users 集合填充 DataTable。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Users );
```

另请参见

- “SAMetaDataCollectionNames 类”一节第 357 页
- “SAMetaDataCollectionNames 成员”一节第 357 页
- “GetSchema(String, String[]) 方法”一节第 250 页

ViewColumns 字段

提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 ViewColumns 集合的常量。此字段为只读字段。

语法

Visual Basic

Public Shared Readonly **ViewColumns** As String

C#

public const string **ViewColumns** ;

示例

以下代码使用 ViewColumns 集合填充 DataTable。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.ViewColumns );
```

另请参见

- “SAMetaDataCollectionNames 类”一节第 357 页
- “SAMetaDataCollectionNames 成员”一节第 357 页
- “GetSchema(String, String[]) 方法”一节第 250 页

Views 字段

提供用于与 SAConnection.GetSchema(String,String[]) 方法配合使用以表示 Views 集合的常量。此字段为只读字段。

语法

Visual Basic

Public Shared Readonly **Views** As String

C#

public const string **Views** ;

示例

以下代码使用 Views 集合填充 DataTable。

```
DataTable schema = GetSchema( SAMetaDataCollectionNames.Views );
```


另请参见

- “SAMetaDataCollectionNames 类”一节第 357 页
- “SAMetaDataCollectionNames 成员”一节第 357 页
- “GetSchema(String, String[]) 方法”一节第 250 页

SAParameter 类

表示 SACommand 的参数以及它到 DataSet 列的映射（可选）。此类无法继承。

语法

Visual Basic

```
Public NotInheritable Class SAParameter
  Inherits DbParameter
  Implements ICloneable
```

C#

```
public sealed class SAParameter : DbParameter,
  ICloneable
```

注释

Implements:IDbDataParameter、IDataParameter、ICloneable

另请参见

- “SAParameter 成员”一节第 367 页

SAParameter 成员

公共构造函数

成员名称	说明
SAParameter 构造函数	初始化一个新的“SAParameter 类”一节第 367 页实例。

公共属性

成员名称	说明
DbType 属性	获取和设置参数的 DbType。
Direction 属性	获取和设置一个值，该值表示参数是仅输入、仅输出、双向还是存储过程返回值参数。
IsNullable 属性	获取和设置表示参数是否接受空值的值。

成员名称	说明
Offset 属性	获取和设置 Value 属性的偏移。
ParameterName 属性	获取和设置 SAParameter 的名称。
Precision 属性	获取和设置表示 Value 属性时最多可以使用的位数。
SADbType 属性	参数的 SADbType。
Scale 属性	获取和设置 Value 精确到的小数位数。
Size 属性	获取和设置列中数据的最大大小（以字节为单位）。
SourceColumn 属性	获取和设置映射到 DataSet 并用于装载或返回值的源列的名称。
SourceColumnNullMapping 属性	获取和设置表示源列是否可以为空的值。这使 SACommandBuilder 能够为可以为空的列正确生成 Update 语句。
SourceVersion 属性	获取和设置要在装载 Value 时使用的 DataRowVersion。
Value 属性	获取和设置参数的值。

公共方法

成员名称	说明
ResetDbType 方法	重置与此 SAParameter 关联的类型（DbType 和 SADbType 的值）。
ToString 方法	返回包含 ParameterName 的字符串。

另请参见

- “SAParameter 类”一节第 367 页

SAParameter 构造函数

初始化一个新的“SAParameter 类”一节第 367 页实例。

SAParameter() 构造函数

初始化具有空值（在 Visual Basic 中是 Nothing）的 SAParameter 对象。

语法

Visual Basic

```
Public Sub New()
```

C#

```
public  SAParameter();
```

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页
- “SAParameter 构造函数”一节第 368 页

SAParameter(String, Object) 构造函数

使用指定的参数名和值来初始化 SAParameter 对象。不建议使用此构造函数；提供它的目的是与其它数据提供程序兼容。

语法

Visual Basic

```
Public Sub New( _
  ByVal parameterName As String, _
  ByVal value As Object _
)
```

C#

```
public  SAParameter(
  string  parameterName,
  object value
);
```

参数

- **parameterName**　参数的名称。
- **value**　表示参数值的对象。

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页
- “SAParameter 构造函数”一节第 368 页

SAParameter(String, SADbType) 构造函数

使用指定的参数名和数据类型来初始化 SAParameter 对象。

语法

Visual Basic

```
Public Sub New( _
  ByVal parameterName As String, _
  ByVal dbType As SADbType _
)
```

C#

```
public  SAParameter(
  string  parameterName,
  SADbType dbType
);
```

参数

- **parameterName** 参数的名称。
- **dbType** SADbType 值之一。

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页
- “SAParameter 构造函数”一节第 368 页
- “SADbType 属性”一节第 376 页

SAParameter(String, SADbType, Int32) 构造函数

使用指定的参数名和数据类型来初始化 SAParameter 对象。

语法

Visual Basic

```
Public Sub New( _
  ByVal parameterName As String, _
  ByVal dbType As SADbType, _
  ByVal size As Integer _
)
```

C#

```
public  SAParameter(
  string  parameterName,
  SADbType dbType,
  int size
);
```

参数

- **parameterName** 参数的名称。
- **dbType** SADbType 值之一

- **size** 参数的长度。

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页
- “SAParameter 构造函数”一节第 368 页

SAParameter(String, SADbType, Int32, String) 构造函数

初始化具有指定参数名、数据类型和长度的 SAParameter 对象。

语法

Visual Basic

```
Public Sub New( _
  ByVal parameterName As String, _
  ByVal dbType As SADbType, _
  ByVal size As Integer, _
  ByVal sourceColumn As String _
)
```

C#

```
public  SAParameter(
  string  parameterName,
  SADbType dbType,
  int size,
  string  sourceColumn
);
```

参数

- **parameterName** 参数的名称。
- **dbType** SADbType 值之一
- **size** 参数的长度。
- **sourceColumn** 要映射的源列的名称。

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页
- “SAParameter 构造函数”一节第 368 页

SAParameter(String, SADbType, Int32, ParameterDirection, Boolean, Byte, Byte, String, DataRowVersion, Object) 构造函数

初始化具有指定参数名、数据类型、长度、方向、为空性、数字精度、数字小数位数、源列、源版本和值的 SAParameter 对象。

语法

Visual Basic

```
Public Sub New( _
  ByVal parameterName As String, _
  ByVal dbType As SADbType, _
  ByVal size As Integer, _
  ByVal direction As ParameterDirection, _
  ByVal isNullable As Boolean, _
  ByVal precision As Byte, _
  ByVal scale As Byte, _
  ByVal sourceColumn As String, _
  ByVal sourceVersion As DataRowVersion, _
  ByVal value As Object _
)
```

C#

```
public  SAParameter(
  string  parameterName,
  SADbType dbType,
  int size,
  ParameterDirection direction,
  bool isNullable,
  byte precision,
  byte scale,
  string  sourceColumn,
  DataRowVersion sourceVersion,
  object value
);
```

参数

- **parameterName** 参数的名称。
- **dbType** SADbType 值之一
- **size** 参数的长度。
- **direction** ParameterDirection 值之一。
- **isNullable** 如果此字段的值可以为空，则为 true；否则为 false。
- **precision** Value 精确到的小数点左右两侧的总位数。
- **scale** Value 精确到的总小数位数。
- **sourceColumn** 要映射的源列的名称。
- **sourceVersion** DataRowVersion 值之一。

- **value** 表示参数值的对象。

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页
- “SAParameter 构造函数”一节第 368 页

DbType 属性

获取和设置参数的 DbType。

语法

Visual Basic

Public Overrides Property **DbType** As DbType

C#

public override DbType **DbType** { get; set; }

注释

SADbType 和 DbType 链接在一起。因此，如果设置 DbType，就会将 SADbType 更改为支持的 SADbType。

该值必须是 SADbType 枚举器的成员。

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页

Direction 属性

获取和设置一个值，该值表示参数是仅输入、仅输出、双向还是存储过程返回值参数。

语法

Visual Basic

Public Overrides Property **Direction** As ParameterDirection

C#

public override ParameterDirection **Direction** { get; set; }

属性值

ParameterDirection 值之一。

注释

如果 ParameterDirection 为 output，而执行关联的 SACommand 并未返回值，则 SAParameter 将包含空值。读取最后一个结果集中的最后一行后，将会更新 Output、InputOut 和 ReturnValue 参数。

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页

IsNullable 属性

获取和设置表示参数是否接受空值的值。

语法

Visual Basic

```
Public Overrides Property IsNullable As Boolean
```

C#

```
public override bool IsNullable { get; set; }
```

注释

如果接受空值，则此属性为 true；否则为 false。缺省值为 false。使用 DBNull 类处理空值。

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页

Offset 属性

获取和设置 Value 属性的偏移。

语法

Visual Basic

```
Public Property Offset As Integer
```

C#

```
public int Offset { get; set; }
```

属性值

值的偏移。缺省值为 0。

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页

ParameterName 属性

获取和设置 SAParameter 的名称。

语法

Visual Basic

Public Overrides Property **ParameterName** As String

C#

public override string **ParameterName** { get; set; }

属性值

缺省值为空字符串。

注释

SQL Anywhere .NET 数据提供程序使用以问号 (?) 标记的定位参数而不是命名参数。

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页

Precision 属性

获取和设置表示 Value 属性时最多可以使用的位数。

语法

Visual Basic

Public Property **Precision** As Byte

C#

public byte **Precision** { get; set; }

属性值

此属性的值为表示 Value 属性时最多可以使用的位数。缺省值为 0，表示数据提供程序为 Value 属性设置精度。

注释

Precision 属性仅用于小数和数字类型的输入参数。

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页

SADbType 属性

参数的 SADbType。

语法

Visual Basic

```
Public Property SADbType As SADbType
```

C#

```
public SADbType SADbType { get; set; }
```

注释

SADbType 和 DbType 链接在一起。因此，如果设置 SADbType，就会将 DbType 更改为支持的 DbType。

该值必须是 SADbType 枚举器的成员。

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页

Scale 属性

获取和设置 Value 精确到的小数位数。

语法

Visual Basic

```
Public Property Scale As Byte
```

C#

```
public byte Scale { get; set; }
```

属性值

Value 精确到的小数位数。缺省值为 0。

注释

Scale 属性仅用于小数和数字类型的输入参数。

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页

Size 属性

获取和设置列中数据的最大大小（以字节为单位）。

语法

Visual Basic

```
Public Overrides Property Size As Integer
```

C#

```
public override int Size { get; set; }
```

属性值

此属性的值为列中数据的最大大小（以字节为单位）。缺省值由参数值推导而来。

注释

此属性的值为列中数据的最大大小（以字节为单位）。缺省值由参数值推导而来。

Size 属性用于二进制或字符串类型。

对可变长度数据类型而言，Size 属性说明要传输到服务器的最大数据量。例如，Size 属性可以用于将某字符串值发送到服务器的数据量限制为前一百个字节。

如果未进行显式设置，将根据指定参数值的实际大小推导出该大小。如果是固定宽度的数据类型，将会忽略 Size 的值。可以检索它以获得信息，它返回将参数值传输到服务器时提供程序使用的最大字节数。

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页

SourceColumn 属性

获取和设置映射到 DataSet 并用于装载或返回值的源列的名称。

语法

Visual Basic

Public Overrides Property **SourceColumn** As String

C#

public override string **SourceColumn** { get; set; }

属性值

一个字符串，指定映射到 DataSet 并用于装载或返回值的源列的名称。

注释

当 SourceColumn 设置为空字符串以外的其它值时，将从包含 SourceColumn 名称的列检索参数的值。如果 Direction 设置为 Input，将从 DataSet 获取该值。如果 Direction 设置为 Output，将从数据源获取该值。如果 Direction 设置为 InputOutput，则从以上两处获取该值。

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页

SourceColumnNullMapping 属性

获取和设置表示源列是否可以为空的值。这使 SACommandBuilder 能够为可以为空的列正确生成 Update 语句。

语法

Visual Basic

Public Overrides Property **SourceColumnNullMapping** As Boolean

C#

public override bool **SourceColumnNullMapping** { get; set; }

注释

如果源列可以为空，则返回 true；否则返回 false。

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页

SourceVersion 属性

获取和设置要在装载 Value 时使用的 DataRowVersion。

语法

Visual Basic

Public Overrides Property **SourceVersion** As DataRowVersion

C#

public override DataRowVersion **SourceVersion** { get; set; }

注释

该属性在 Update 操作过程中由 UpdateCommand 使用，它决定将参数值设置为 Current 还是 Original。这使主键得以更新。InsertCommand 和 DeleteCommand 会忽略此属性。此属性设置为 Item 属性使用的 DataRow 的版本，或 DataRow 对象的 GetChildRows 方法。

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页

Value 属性

获取和设置参数的值。

语法

Visual Basic

Public Overrides Property **Value** As Object

C#

public override object **Value** { get; set; }

属性值

指定参数值的 Object。

注释

如果是输入参数，该值将绑定到发送给服务器的 SACommand。如果是输出或返回值参数，将在完成 SACommand 时和关闭 SADataReader 后设置该值。

将空参数值发送到服务器时，必须指定 DBNull（而非空值）。系统中的空值是没有值的空对象。DBNull 用于表示空值。

如果应用程序指定了数据库类型，当 SQL Anywhere .NET 数据提供程序将数据发送到服务器时，绑定的值将转换为该类型。如果提供程序支持 IConvertible 接口，则会尝试转换任何类型的值。如果指定的类型与值不兼容，可能会导致转换错误。

通过设置 Value，DbType 和 SADbType 属性都可以推导出来。

Update 会覆盖 Value 属性。

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页

ResetDbType 方法

重置与此 SAParameter 关联的类型（DbType 和 SADbType 的值）。

语法

Visual Basic

```
Public Overrides Sub ResetDbType()
```

C#

```
public override void ResetDbType();
```

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页

ToString 方法

返回包含 ParameterName 的字符串。

语法

Visual Basic

```
Public Overrides Function ToString() As String
```

C#

```
public override string  ToString();
```

返回值

参数的名称。

另请参见

- “SAParameter 类”一节第 367 页
- “SAParameter 成员”一节第 367 页

SAParameterCollection 类

表示所有 SACommand 对象的参数，以及其到 DataSet 列的映射（可选）。此类无法继承。

语法

Visual Basic

Public NotInheritable Class **SAParameterCollection**
  Inherits DbParameterCollection

C#

public sealed class **SAParameterCollection** : DbParameterCollection

注释

没有用于 SAParameterCollection 的构造函数。从 SACommand 对象的 SACommand.Parameters 属性获取 SAParameterCollection 对象。

另请参见

- “SAParameterCollection 成员”一节第 381 页
- “SACommand 类”一节第 195 页
- “Parameters 属性”一节第 202 页
- “SAParameter 类”一节第 367 页
- “SAParameterCollection 类”一节第 380 页

SAParameterCollection 成员

公共属性

成员名称	说明
Count 属性	返回集合中 SAParameter 对象的数量。
IsFixedSize 属性	获取表示 SAParameterCollection 是否有固定大小的值。
IsReadOnly property	获取表示 SAParameterCollection 是否为只读的值。
IsSynchronized 属性	获取表示是否同步了 SAParameterCollection 对象的值。
Item 属性	获取和设置指定索引处的 SAParameter 对象。
SyncRoot 属性	获取可用于同步对 SAParameterCollection 的访问的对象。

公共方法

成员名称	说明
Add 方法	将 SAParameter 对象添加到此集合中。
AddRange 方法	将值数组添加到 SAParameterCollection 的尾部。

成员名称	说明
AddWithValue 方法	在此集合的末尾添加一个值。
Clear 方法	删除该集合中的所有项目。
Contains 方法	表示集合中是否存在 SAParameter 对象。
CopyTo 方法	将 SAParameter 对象从 SAParameterCollection 复制到指定的数组。
GetEnumerator 方法	返回迭代通过 SAParameterCollection 的枚举器。
IndexOf 方法	返回 SAParameter 对象在集合中的位置。
Insert 方法	在集合中指定索引处插入 SAParameter 对象。
Remove 方法	从集合中删除指定的 SAParameter 对象。
RemoveAt 方法	从集合中删除指定的 SAParameter 对象。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SACommand 类”一节第 195 页
- “Parameters 属性”一节第 202 页
- “SAParameter 类”一节第 367 页
- “SAParameterCollection 类”一节第 380 页

Count 属性

返回集合中 SAParameter 对象的数量。

语法

Visual Basic

```
Public Overrides Readonly Property Count As Integer
```

C#

```
public override int Count { get;}
```

属性值

集合中 SAParameter 对象的数量。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页
- “SAParameter 类”一节第 367 页
- “SAParameterCollection 类”一节第 380 页

IsFixedSize 属性

获取表示 SAParameterCollection 是否有固定大小的值。

语法

Visual Basic

```
Public Overrides Readonly Property IsFixedSize As Boolean
```

C#

```
public override bool IsFixedSize { get;}
```

属性值

如果此集合具有固定大小，则为 true，否则为 false。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页

IsReadOnly property

获取表示 SAParameterCollection 是否为只读的值。

语法

Visual Basic

```
Public Overrides Readonly Property IsReadOnly As Boolean
```

C#

```
public override bool IsReadOnly { get;}
```

属性值

如果此集合为只读，则返回 true；否则返回 false。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页

IsSynchronized 属性

获取表示是否同步了 SAParameterCollection 对象的值。

语法

Visual Basic

```
Public Overrides Readonly Property IsSynchronized As Boolean
```

C#

```
public override bool IsSynchronized { get;}
```

属性值

如果此集合是同步集合，则返回 true；否则返回 false。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页

Item 属性

获取和设置指定索引处的 SAParameter 对象。

Item(Int32) 属性

获取和设置指定索引处的 SAParameter 对象。

语法

Visual Basic

```
Public Property Item ( _
   ByVal index As Integer _
) As SAParameter
```

C#

```
public SAParameter this [
   int index
] { get; set; }
```

参数

- **index** 要检索的参数的从零开始索引。

属性值

指定索引处的 SAParameter。

注释

在 C# 中，此属性是 SAParameterCollection 对象的索引器。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页
- “Item 属性”一节第 384 页
- “SAParameter 类”一节第 367 页
- “SAParameterCollection 类”一节第 380 页

Item(String) 属性

获取和设置指定索引处的 SAParameter 对象。

语法

Visual Basic

```
Public Property Item ( _
   ByVal parameterName As String _
) As SAParameter
```

C#

```
public SAParameter this [
   string  parameterName
] { get; set; }
```

参数

- **parameterName**　要检索的参数的名称。

属性值

具有指定名称的 SAParameter 对象。

注释

在 C# 中，此属性是 SAParameterCollection 对象的索引器。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页
- “Item 属性”一节第 384 页
- “SAParameter 类”一节第 367 页
- “SAParameterCollection 类”一节第 380 页
- “Item(Int32) 属性”一节第 302 页
- “GetOrdinal 方法”一节第 316 页
- “GetValue(Int32) 方法”一节第 322 页
- “GetFieldType 方法”一节第 312 页

SyncRoot 属性

获取可用于同步对 SAParameterCollection 的访问的对象。

语法

Visual Basic

```
Public Overrides Readonly Property SyncRoot As Object
```

C#

```
public override object SyncRoot { get;}
```

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页

Add 方法

将 SAParameter 对象添加到此集合中。

Add(Object) 方法

将 SAParameter 对象添加到此集合中。

语法

Visual Basic

```
Public Overrides Function Add( _
  ByVal value As Object _
) As Integer
```

C#

```
public override int Add(
  object value
);
```

参数

- **value** 要添加到集合中的 SAParameter 对象。

返回值

新 SAParameter 对象的索引。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页
- “Add 方法”一节第 386 页
- “SAParameter 类”一节第 367 页

Add(SAParameter) 方法

将 SAParameter 对象添加到此集合中。

语法

Visual Basic

```
Public Function Add( _
   ByVal value As SAParameter _
) As SAParameter
```

C#

```
public SAParameter Add(
   SAParameter value
);
```

参数

- **value** 要添加到集合中的 SAParameter 对象。

返回值

新的 SAParameter 对象。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页
- “Add 方法”一节第 386 页

Add(String, Object) 方法

将使用指定参数名和值创建的 SAParameter 对象添加到此集合中。

语法

Visual Basic

```
Public Function Add( _
   ByVal parameterName As String, _
   ByVal value As Object _
) As SAParameter
```

C#

```
public SAParameter Add(
  string parameterName,
  object value
);
```

参数

- **parameterName** 参数的名称。
- **value** 要添加到连接中的参数的值。

返回值

新的 SAParameter 对象。

注释

鉴于系统对 0 和 0.0 常量的特殊处理方法以及对重载方法的解析方式，我们强烈建议您在使用此方法时显式地将常量值转换为类型对象。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页
- “Add 方法”一节第 386 页
- “SAParameter 类”一节第 367 页

Add(String, SADbType) 方法

将使用指定参数名和数据类型创建的 SAParameter 对象添加到此集合中。

语法

Visual Basic

```
Public Function Add( _
  ByVal parameterName As String, _
  ByVal saDbType As SADbType _
) As SAParameter
```

C#

```
public SAParameter Add(
  string parameterName,
  SADbType saDbType
);
```

参数

- **parameterName** 参数的名称。
- **saDbType** SADbType 值之一。

返回值

新的 SAParameter 对象。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页
- “Add 方法”一节第 386 页
- “SADbType 枚举”一节第 329 页
- “Add(SAParameter) 方法”一节第 387 页
- “Add(String, Object) 方法”一节第 387 页

Add(String, SADbType, Int32) 方法

将使用指定的参数名、数据类型和长度创建的 SAParameter 对象添加到此集合中。

语法

Visual Basic

```
Public Function Add( _
   ByVal parameterName As String, _
   ByVal saDbType As SADbType, _
   ByVal size As Integer _
) As SAParameter
```

C#

```
public SAParameter Add(
   string  parameterName,
   SADbType saDbType,
   int size
);
```

参数

- **parameterName** 参数的名称。
- **saDbType** SADbType 值之一。
- **size** 参数的长度。

返回值

新的 SAParameter 对象。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页
- “Add 方法”一节第 386 页
- “SADbType 枚举”一节第 329 页
- “Add(SAParameter) 方法”一节第 387 页
- “Add(String, Object) 方法”一节第 387 页

Add(String, SADbType, Int32, String) 方法

将使用指定的参数名、数据类型、长度和源列名创建的 SAParameter 对象添加到此集合中。

语法

Visual Basic

```
Public Function Add( _
   ByVal parameterName As String, _
   ByVal saDbType As SADbType, _
   ByVal size As Integer, _
   ByVal sourceColumn As String _
) As SAParameter
```

C#

```
public SAParameter Add(
   string  parameterName,
   SADbType saDbType,
   int size,
   string  sourceColumn
);
```

参数

- **parameterName** 参数的名称。
- **saDbType** SADbType 值之一。
- **size** 列的长度。
- **sourceColumn** 要映射的源列的名称。

返回值

新的 SAParameter 对象。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页
- “Add 方法”一节第 386 页
- “SADbType 枚举”一节第 329 页
- “Add(SAParameter) 方法”一节第 387 页
- “Add(String, Object) 方法”一节第 387 页

AddRange 方法

将值数组添加到 SAParameterCollection 的尾部。

AddRange(Array) 方法

将值数组添加到 SAParameterCollection 的尾部。

语法

Visual Basic

```
Public Overrides Sub AddRange( _
  ByVal values As Array _
)
```

C#

```
public override void AddRange(
  Array values
);
```

参数

- **values** 要添加的值。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页
- “AddRange 方法”一节第 391 页

AddRange(SAParameter[]) 方法

将值数组添加到 SAParameterCollection 的尾部。

语法

Visual Basic

```
Public Sub AddRange( _
```

```
  ByVal values As SAParameter() _
)
```

C#

```
public void AddRange(
  SAParameter[] values
);
```

参数

- **values** 要添加到此集合尾部的 SAParameter 对象数组。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页
- “AddRange 方法”一节第 391 页

AddWithValue 方法

在此集合的末尾添加一个值。

语法

Visual Basic

```
Public Function AddWithValue( _
  ByVal parameterName As String, _
  ByVal value As Object _
) As SAParameter
```

C#

```
public SAParameter AddWithValue(
  string  parameterName,
  object value
);
```

参数

- **parameterName** 参数的名称。
- **value** 要添加的值。

返回值

新的 SAParameter 对象。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页

Clear 方法

删除该集合中的所有项目。

语法

Visual Basic

```
Public Overrides Sub Clear()
```

C#

```
public override void Clear();
```

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页

Contains 方法

表示集合中是否存在 SAParameter 对象。

Contains(Object) 方法

表示集合中是否存在 SAParameter 对象。

语法

Visual Basic

```
Public Overrides Function Contains( _
  ByVal value As Object _
) As Boolean
```

C#

```
public override bool Contains(
  object value
);
```

参数

- **value** 要查找的 SAParameter 对象。

返回值

如果集合包含 SAParameter 对象，则为 true。否则为 false。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页
- “Contains 方法”一节第 393 页
- “SAParameter 类”一节第 367 页
- “Contains(String) 方法”一节第 394 页

Contains(String) 方法

表示集合中是否存在 SAParameter 对象。

语法

Visual Basic

```
Public Overrides Function Contains( _
  ByVal value As String _
) As Boolean
```

C#

```
public override bool Contains(
  string  value
);
```

参数

- **value**　要搜索的参数的名称。

返回值

如果集合包含 SAParameter 对象，则为 true。否则为 false。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页
- “Contains 方法”一节第 393 页
- “SAParameter 类”一节第 367 页
- “Contains(Object) 方法”一节第 393 页

CopyTo 方法

将 SAParameter 对象从 SAParameterCollection 复制到指定的数组。

语法

Visual Basic

```
Public Overrides Sub CopyTo( _
  ByVal array As Array, _
```



```
  ByVal index As Integer _
)
```

C#

```
public override void CopyTo(
  Array array,
  int index
);
```

参数

- **array** 要将 SAParameter 对象复制到的数组。
- **index** 数组的起始索引。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页
- “SAParameter 类”一节第 367 页
- “SAParameterCollection 类”一节第 380 页

GetEnumerator 方法

返回迭代通过 SAParameterCollection 的枚举器。

语法

Visual Basic

```
Public Overrides Function GetEnumerator() As IEnumerator
```

C#

```
public override IEnumerator GetEnumerator();
```

返回值

SAParameterCollection 对象的 IEnumerator。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页
- “SAParameterCollection 类”一节第 380 页

IndexOf 方法

返回 SAParameter 对象在集合中的位置。

IndexOf(Object) 方法

返回 SAParameter 对象在集合中的位置。

语法

Visual Basic

```
Public Overrides Function IndexOf( _
  ByVal value As Object _
) As Integer
```

C#

```
public override int IndexOf(
  object value
);
```

参数

- **value** 要查找的 SAParameter 对象。

返回值

SAParameter 对象在集合中的位置（从零开始）。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页
- “IndexOf 方法”一节第 395 页
- “SAParameter 类”一节第 367 页
- “IndexOf(String) 方法”一节第 396 页

IndexOf(String) 方法

返回 SAParameter 对象在集合中的位置。

语法

Visual Basic

```
Public Overrides Function IndexOf( _
  ByVal parameterName As String _
) As Integer
```

C#

```
public override int IndexOf(
  string  parameterName
);
```


参数

- **parameterName** 要查找的参数的名称。

返回值

SAParameter 对象在集合中的索引（从零开始）。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页
- “IndexOf 方法”一节第 395 页
- “SAParameter 类”一节第 367 页
- “IndexOf(Object) 方法”一节第 396 页

Insert 方法

在集合中指定索引处插入 SAParameter 对象。

语法

Visual Basic

```
Public Overrides Sub Insert( _
  ByVal index As Integer, _
  ByVal value As Object _
)
```

C#

```
public override void Insert(
  int index,
  object value
);
```

参数

- **index** 集合内要插入参数的索引（从零开始）。
- **value** 要添加到集合中的 SAParameter 对象。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页

Remove 方法

从集合中删除指定的 SAParameter 对象。

语法

Visual Basic

```
Public Overrides Sub Remove( _
  ByVal value As Object _
)
```

C#

```
public override void Remove(
  object value
);
```

参数

- **value** 要从集合中删除的 SAParameter 对象。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页

RemoveAt 方法

从集合中删除指定的 SAParameter 对象。

RemoveAt(Int32) 方法

从集合中删除指定的 SAParameter 对象。

语法

Visual Basic

```
Public Overrides Sub RemoveAt( _
  ByVal index As Integer _
)
```

C#

```
public override void RemoveAt(
  int index
);
```

参数

- **index** 要删除的参数的索引（从零开始）。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页
- “RemoveAt 方法”一节第 398 页
- “RemoveAt(String) 方法”一节第 399 页

RemoveAt(String) 方法

从集合中删除指定的 SAParameter 对象。

语法

Visual Basic

```
Public Overrides Sub RemoveAt( _
  ByVal parameterName As String _
)
```

C#

```
public override void RemoveAt(
  string  parameterName
);
```

参数

- **parameterName**　要删除的 SAParameter 对象的名称。

另请参见

- “SAParameterCollection 类”一节第 380 页
- “SAParameterCollection 成员”一节第 381 页
- “RemoveAt 方法”一节第 398 页
- “RemoveAt(Int32) 方法”一节第 398 页

SAPermission 类

使 SQL Anywhere .NET 数据提供程序能够确保用户具有访问 SQL Anywhere 数据源所需的安全级别。此类无法继承。

语法

Visual Basic

```
Public NotInheritable Class SAPermission
  Inherits DBDataPermission
```

C#

```
public sealed class SAPermission : DBDataPermission
```

注释

Base classes DBDataPermission

另请参见

- “SAPermission 成员”一节第 400 页

SAPermission 成员

公共构造函数

成员名称	说明
SAPermission 构造函数	初始化一个新的 SAPermission 类实例。

公共属性

成员名称	说明
AllowBlankPassword（继承自 DBDataPermission）	获取表示是否允许使用空口令的值。

公共方法

成员名称	说明
Add（继承自 DBDataPermission）	将指定连接字符串的访问权限添加到 DBDataPermission 的现有状态。
Assert（继承自 CodeAccessPermission）	声明调用代码可通过调用此方法的代码来访问受权限请求保护的资源，即使堆栈中处于更高级别的调用者尚未被赋予访问该资源的权限也是如此。使用 CodeAccessPermission.Assert 会产生安全问题。
Copy（继承自 DBDataPermission）	创建并返回当前权限对象的完全相同副本。
Demand（继承自 CodeAccessPermission）	在运行时，如果调用堆栈中处于较高级别的所有调用者尚未被赋予由当前实例指定的权限，则强制 SecurityException。
Deny（继承自 CodeAccessPermission）	防止调用堆栈中处于较高级别的调用者使用可调用此方法的代码，来访问由当前实例指定的资源。
Equals（继承自 CodeAccessPermission）	确定所指定的 CodeAccessPermission 对象是否与当前的 CodeAccessPermission 相等。

成员名称	说明
FromXml（继承自 DBDataPermission）	从 XML 编码重新构建具有指定状态的安全对象。
GetHashCode（继承自 CodeAccessPermission）	为 CodeAccessPermission 对象获取一个适合在散列算法和诸如散列表之类的数据结构中使用的散列代码。
Intersect（继承自 DBDataPermission）	返回表示当前权限对象与指定权限对象交集的新权限对象。
IsSubsetOf（继承自 DBDataPermission）	返回表示当前权限对象是否为指定权限对象子集的值。
IsUnrestricted（继承自 DBDataPermission）	返回指示是否可以在不了解任何权限语义的情况下将权限表示为不受限制的值。
PermitOnly（继承自 CodeAccessPermission）	防止调用堆栈中处于较高级别的调用者使用调用此方法的代码，来访问除当前实例所指定的资源之外的所有资源。
ToString（继承自 CodeAccessPermission）	创建并返回当前权限对象的字符串表示形式。
ToXml（继承自 DBDataPermission）	创建安全对象及其当前状态的 XML 编码。
Union（继承自 DBDataPermission）	返回取值为当前及指定权限对象并集的新权限对象。

另请参见

- “SAPermission 类”一节第 399 页

SAPermission 构造函数

初始化一个新的 SAPermission 类实例。

语法

Visual Basic

```
Public Sub New( _
  ByVal state As PermissionState _
)
```

C#

```
public  SAPermission(
  PermissionState state
);
```

参数

- **state** PermissionState 值之一。

另请参见

- “SAPermission 类”一节第 399 页
- “SAPermission 成员”一节第 400 页

SAPermissionAttribute 类

将安全操作与自定义安全特性相关联。此类无法继承。

语法

Visual Basic

```
Public NotInheritable Class SAPermissionAttribute
  Inherits DBDataPermissionAttribute
```

C#

```
public sealed class SAPermissionAttribute : DBDataPermissionAttribute
```

另请参见

- “SAPermissionAttribute 成员”一节第 402 页

SAPermissionAttribute 成员

公共构造函数

成员名称	说明
SAPermissionAttribute 构造函数	初始化一个新的 SAPermissionAttribute 类实例。

公共属性

成员名称	说明
Action（继承自 SecurityAttribute）	获取或设置安全操作。
AllowBlankPassword（继承自 DBDataPermissionAttribute）	获取或设置表示是否允许使用空口令的值。
ConnectionString（继承自 DBDataPermissionAttribute）	获取或设置允许的连接字符串。

成员名称	说明
KeyRestrictionBehavior（继承自 DBDataPermissionAttribute）	指明由 DBDataPermissionAttribute.KeyRestrictions 标识的连接字符串参数列表是否是允许使用的唯一连接字符串参数。
KeyRestrictions（继承自 DBDataPermissionAttribute）	获取或设置允许或禁止使用的连接字符串参数。
TypeId（继承自 Attribute）	如果使用派生类实现，则为此 Attribute 获取唯一标识符。
Unrestricted（继承自 SecurityAttribute）	获取或设置用于表示是否声明了对该属性所保护的资源的完全权限（不受限制的权限）的值。

公共方法

成员名称	说明
CreatePermission 方法	返回一个根据 attribute 属性配置的 SAPermission 对象。
Equals（继承自 Attribute）	返回一个用于指示此实例是否与指定对象相等的值。
GetHashCode（继承自 Attribute）	返回此实例的散列代码。
IsDefaultAttribute（继承自 Attribute）	如果在某个派生类中被替换，则表示此实例的值是否为该派生类的缺省值。
Match（继承自 Attribute）	如果在某个派生类中被替换，则返回一个用于指示此实例是否与指定对象相等的值。
ShouldSerializeConnectionString（继承自 DBDataPermissionAttribute）	指明属性是否应将连接字符串序列化。
ShouldSerializeKeyRestrictions（继承自 DBDataPermissionAttribute）	指明属性是否应将键限制集序列化。

另请参见

- “SAPermissionAttribute 类”一节第 402 页

SAPermissionAttribute 构造函数

初始化一个新的 SAPermissionAttribute 类实例。

语法

Visual Basic

```
Public Sub New( _
  ByVal action As SecurityAction _
)
```

C#

```
public  SAPermissionAttribute(
  SecurityAction action
);
```

参数

- **action** SecurityAction 值之一，表示可以使用声明性安全执行的操作。

另请参见

- “SAPermissionAttribute 类”一节第 402 页
- “SAPermissionAttribute 成员”一节第 402 页

CreatePermission 方法

返回一个根据 attribute 属性配置的 SAPermission 对象。

语法

Visual Basic

```
Public Overrides Function CreatePermission() As IPermission
```

C#

```
public override IPermission CreatePermission();
```

另请参见

- “SAPermissionAttribute 类”一节第 402 页
- “SAPermissionAttribute 成员”一节第 402 页

SARowsCopiedEventArgs 类

表示传递给 SARowsCopiedEventHandler 的参数集。此类无法继承。

语法

Visual Basic

```
Public NotInheritable Class SARowsCopiedEventArgs
```


C#

public sealed class **SARowsCopiedEventArgs**

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SARowsCopiedEventArgs 类。

另请参见

- “SARowsCopiedEventArgs 成员”一节第 405 页

SARowsCopiedEventArgs 成员

公共构造函数

成员名称	说明
SARowsCopiedEventArgs 构造函数	创建一个新的 SARowsCopiedEventArgs 对象实例。

公共属性

成员名称	说明
Abort 属性	获取或设置表示是否应中止批量复制操作的值。
RowsCopied 属性	获取在当前批量复制操作期间复制的行数。

另请参见

- “SARowsCopiedEventArgs 类”一节第 404 页

SARowsCopiedEventArgs 构造函数

创建一个新的 SARowsCopiedEventArgs 对象实例。

语法

Visual Basic

```
Public Sub New( _
  ByVal rowsCopied As Long _
)
```

C#

```
public  SARowsCopiedEventArgs(
  long rowsCopied
);
```

参数

- **rowsCopied** 表示在当前批量复制操作期间复制的行数的 64 位整数值。

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SARowsCopiedEventArgs 类。

另请参见

- “SARowsCopiedEventArgs 类”一节第 404 页
- “SARowsCopiedEventArgs 成员”一节第 405 页

Abort 属性

获取或设置表示是否应中止批量复制操作的值。

语法

Visual Basic

```
Public Property Abort As Boolean
```

C#

```
public bool Abort { get; set; }
```

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SARowsCopiedEventArgs 类。

另请参见

- “SARowsCopiedEventArgs 类”一节第 404 页
- “SARowsCopiedEventArgs 成员”一节第 405 页

RowsCopied 属性

获取在当前批量复制操作期间复制的行数。

语法

Visual Basic

```
Public Readonly Property RowsCopied As Long
```

C#

```
public long RowsCopied { get;}
```

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SARowsCopiedEventArgs 类。

另请参见

- “SARowsCopiedEventArgs 类”一节第 404 页
- “SARowsCopiedEventArgs 成员”一节第 405 页

SARowsCopiedEventHandler 委派

表示处理 SABulkCopy 的 SABulkCopy.SARowsCopied 事件的方法。

语法

Visual Basic

```
Public Delegate Sub SARowsCopiedEventHandler( _
  ByVal sender As Object, _
  ByVal rowsCopiedEventArgs As SARowsCopiedEventArgs _
)
```

C#

```
public delegate void SARowsCopiedEventHandler(
  object sender,
  SARowsCopiedEventArgs rowsCopiedEventArgs
);
```

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SARowsCopiedEventHandler 委派。

另请参见

- “SABulkCopy 类”一节第 168 页

SARowUpdatedEventArgs 类

为 RowUpdated 事件提供数据。此类无法继承。

语法

Visual Basic

```
Public NotInheritable Class SARowUpdatedEventArgs
  Inherits RowUpdatedEventArgs
```

C#

```
public sealed class SARowUpdatedEventArgs : RowUpdatedEventArgs
```

另请参见

- “SARowUpdatedEventArgs 成员”一节第 408 页

SARowUpdatedEventArgs 成员

公共构造函数

成员名称	说明
SARowUpdatedEventArgs 构造函数	初始化一个新的 SARowUpdatedEventArgs 类实例。

公共属性

成员名称	说明
Command 属性	获取在调用 DataAdapter.Update 时执行的 SACommand。
Errors（继承自 RowUpdatedEventArgs）	获取执行 RowUpdatedEventArgs.Command 时 .NET Framework 数据提供程序所产生的任何错误。
RecordsAffected 属性	返回通过执行 SQL 语句所更改、插入或删除的行数。
Row（继承自 RowUpdatedEventArgs）	获取通过 DbDataAdapter.Update 发送的 DataRow。
RowCount（继承自 RowUpdatedEventArgs）	获取一批更新记录中处理的行数。
StatementType（继承自 RowUpdatedEventArgs）	获取所执行 SQL 语句的类型。
Status（继承自 RowUpdatedEventArgs）	获取 RowUpdatedEventArgs.Command 的 UpdateStatus。
TableMapping（继承自 RowUpdatedEventArgs）	获取通过 DbDataAdapter.Update 发送的 DataTableMapping。

公共方法

成员名称	说明
CopyToRows（继承自 RowUpdatedEventArgs）	将对修改行的引用复制到所提供的数组中。

另请参见

- “SARowUpdatedEventArgs 类”一节第 407 页

SARowUpdatedEventArgs 构造函数

初始化一个新的 SARowUpdatedEventArgs 类实例。

语法

Visual Basic

```
Public Sub New( _
  ByVal row As DataRow, _
  ByVal command As IDbCommand, _
  ByVal statementType As StatementType, _
  ByVal tableMapping As DataTableMapping _
)
```

C#

```
public  SARowUpdatedEventArgs(
  DataRow row,
  IDbCommand command,
  StatementType statementType,
  DataTableMapping tableMapping
);
```

参数

- **row** 通过 Update 发送的 DataRow。
- **command** 调用 Update 时执行的 IDbCommand。
- **statementType** 指定所执行查询类型的 StatementType 值之一。
- **tableMapping** 通过 Update 发送的 DataTableMapping。

另请参见

- “SARowUpdatedEventArgs 类”一节第 407 页
- “SARowUpdatedEventArgs 成员”一节第 408 页

Command 属性

获取在调用 DataAdapter.Update 时执行的 SACommand。

语法

Visual Basic

```
Public Readonly Property Command As SACommand
```

C#

```
public SACommand Command { get;}
```

另请参见

- “SARowUpdatedEventArgs 类”一节第 407 页
- “SARowUpdatedEventArgs 成员”一节第 408 页

RecordsAffected 属性

返回通过执行 SQL 语句所更改、插入或删除的行数。

语法

Visual Basic

```
Public Readonly Property RecordsAffected As Integer
```

C#

```
public int RecordsAffected { get;}
```

属性值

更改、插入或删除的行数；如果没有任何行受到影响或语句失败，则值为 0；如果是 SELECT 语句，则值为 -1。

另请参见

- “SARowUpdatedEventArgs 类”一节第 407 页
- “SARowUpdatedEventArgs 成员”一节第 408 页

SARowUpdatedEventHandler 委派

表示处理 SADataAdapter 的 RowUpdated 事件的方法。

语法

Visual Basic

```
Public Delegate Sub SARowUpdatedEventHandler( _
   ByVal sender As Object, _
   ByVal e As SARowUpdatedEventArgs _
)
```

C#

```
public delegate void SARowUpdatedEventHandler(
   object sender,
   SARowUpdatedEventArgs e
);
```


SARowUpdatingEventArgs 类

为 RowUpdating 事件提供数据。此类无法继承。

语法

Visual Basic

```
Public NotInheritable Class SARowUpdatingEventArgs
  Inherits RowUpdatingEventArgs
```

C#

```
public sealed class SARowUpdatingEventArgs : RowUpdatingEventArgs
```

另请参见

- “SARowUpdatingEventArgs 成员”一节第 411 页

SARowUpdatingEventArgs 成员

公共构造函数

成员名称	说明
SARowUpdatingEventArgs 构造函数	初始化一个新的 SARowUpdatingEventArgs 类实例。

公共属性

成员名称	说明
Command 属性	指定要在执行 Update 时执行的 SACommand。
Errors（继承自 RowUpdatingEventArgs）	获取执行 RowUpdatedEventArgs.Command 时 .NET Framework 数据提供程序所产生的任何错误。
Row（继承自 RowUpdatingEventArgs）	获取将作为插入、更新或删除操作的一部分发送给服务器的 DataRow。
StatementType（继承自 RowUpdatingEventArgs）	获取要执行的 SQL 语句的类型。
Status（继承自 RowUpdatingEventArgs）	获取或设置 RowUpdatedEventArgs.Command 的 UpdateStatus。
TableMapping（继承自 RowUpdatingEventArgs）	获取要通过 DbDataAdapter.Update 发送的 DataTableMapping。

另请参见

- “SARowUpdatingEventArgs 类”一节第 411 页

SARowUpdatingEventArgs 构造函数

初始化一个新的 SARowUpdatingEventArgs 类实例。

语法

Visual Basic

```
Public Sub New( _
  ByVal row As DataRow, _
  ByVal command As IDbCommand, _
  ByVal statementType As StatementType, _
  ByVal tableMapping As DataTableMapping _
)
```

C#

```
public  SARowUpdatingEventArgs(
  DataRow row,
  IDbCommand command,
  StatementType statementType,
  DataTableMapping tableMapping
);
```

参数

- **row** 要更新的 DataRow。
- **command** 要在更新期间执行的 IDbCommand。
- **statementType** 指定所执行查询类型的 StatementType 值之一。
- **tableMapping** 通过 Update 发送的 DataTableMapping。

另请参见

- “SARowUpdatingEventArgs 类”一节第 411 页
- “SARowUpdatingEventArgs 成员”一节第 411 页

Command 属性

指定要在执行 Update 时执行的 SACommand。

语法

Visual Basic

```
Public Property Command As SACommand
```


C#

```
public SACommand Command { get; set; }
```

另请参见

- “SARowUpdatingEventArgs 类”一节第 411 页
- “SARowUpdatingEventArgs 成员”一节第 411 页

SARowUpdatingEventHandler 委派

表示处理 SADataAdapter 的 RowUpdating 事件的方法。

语法

Visual Basic

```
Public Delegate Sub SARowUpdatingEventHandler( _
  ByVal sender As Object, _
  ByVal e As SARowUpdatingEventArgs _
)
```

C#

```
public delegate void SARowUpdatingEventHandler(
  object sender,
  SARowUpdatingEventArgs e
);
```

SATcpOptionsBuilder 类

为创建和管理 SAConnection 对象所使用的连接字符串的 TCP 选项部分提供了一种简单的方法。此类无法继承。

语法

Visual Basic

```
Public NotInheritable Class SATcpOptionsBuilder
  Inherits SAConnectionStringBuilderBase
```

C#

```
public sealed class SATcpOptionsBuilder : SAConnectionstring BuilderBase
```

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SATcpOptionsBuilder 类。

另请参见

- “SATcpOptionsBuilder 成员”一节第 414 页
- “SAConnection 类”一节第 234 页

SATcpOptionsBuilder 成员

公共构造函数

成员名称	说明
SATcpOptionsBuilder 构造函数	初始化一个新的“SATcpOptionsBuilder 类”一节第 413 页实例。

公共属性

成员名称	说明
Broadcast 属性	获取或设置 Broadcast 选项。
BroadcastListener 属性	获取或设置 BroadcastListener 选项。
BrowsableConnectionString（继承自 DbConnectionStringBuilder）	获取或设置一个值，指示 DbConnectionStringBuilder.ConnectionString 在 Visual Studio 设计器中是否可见。
ClientPort 属性	获取或设置 ClientPort 选项。
ConnectionString（继承自 DbConnectionStringBuilder）	获取或设置与 DbConnectionStringBuilder 相关联的连接字符串。
Count（继承自 DbConnectionStringBuilder）	获取 DbConnectionStringBuilder.ConnectionString 中当前包含的键数。
DoBroadcast 属性	获取或设置 DoBroadcast 选项。
Host 属性	获取或设置 Host 选项。
IPV6 属性	获取或设置 IPV6 选项。
IsFixedSize（继承自 DbConnectionStringBuilder）	获取表示 DbConnectionStringBuilder 是否有固定大小的值。
IsReadOnly（继承自 DbConnectionStringBuilder）	获取表示 DbConnectionStringBuilder 是否为只读的值。
Item 属性（继承自 SAConnectionStringBuilderBase）	获取或设置连接关键字的值。

成员名称	说明
Keys 属性（继承自 SAConnectionStringBuilderBase）	获取包含 SAConnectionStringBuilder 中的键的 System.Collections.ICollection。
LDAP 属性	获取或设置 LDAP 选项。
LocalOnly 属性	获取或设置 LocalOnly 选项。
MyIP 属性	获取或设置 MyIP 选项。
ReceiveBufferSize 属性	获取或设置 ReceiveBufferSize 选项。
SendBufferSize 属性	获取或设置 Send BufferSize 选项。
ServerPort 属性	获取或设置 ServerPort 选项。
TDS 属性	获取或设置 TDS 选项。
Timeout 属性	获取或设置 Timeout 选项。
Values（继承自 DbConnectionStringBuilder）	获取包含 DbConnectionStringBuilder 中的值的 ICollection。
VerifyServerName 属性	获取或设置 VerifyServerName 选项。

公共方法

成员名称	说明
Add（继承自 DbConnectionStringBuilder）	将具有指定键和值的条目添加到 DbConnectionStringBuilder 中。
Clear（继承自 DbConnectionStringBuilder）	清除 DbConnectionStringBuilder 实例的内容。
ContainsKey 方法（继承自 SAConnectionStringBuilderBase）	确定 SAConnectionStringBuilder 对象是否包含特定关键字。
EquivalentTo（继承自 DbConnectionStringBuilder）	将此 DbConnectionStringBuilder 对象中的连接信息与所提供对象中的连接信息进行比较。
GetKeyword 方法（继承自 SAConnectionStringBuilderBase）	获取指定 SAConnectionStringBuilder 属性的关键字。

成员名称	说明
GetUseLongNameAsKeyword 方法（继承自 SAConnectionStringBuilderBase）	获取表示是否可以在连接字符串中使用长连接参数名的布尔值。
Remove 方法（继承自 SAConnectionStringBuilderBase）	从 SAConnectionStringBuilder 实例中删除具有指定键的条目。
SetUseLongNameAsKeyword 方法（继承自 SAConnectionStringBuilderBase）	设置表示连接字符串中是否使用长连接参数名的布尔值。缺省情况下使用长连接参数名。
ShouldSerialize 方法（继承自 SAConnectionStringBuilderBase）	表示此 SAConnectionStringBuilder 实例中是否存在指定的键。
ToString 方法	将 TcpOptionsBuilder 对象转换为字符串表示。
TryGetValue 方法（继承自 SAConnectionStringBuilderBase）	从此 SAConnectionStringBuilder 中检索与所提供的键对应的值。

另请参见

- “SATcpOptionsBuilder 类”一节第 413 页
- “SAConnection 类”一节第 234 页

SATcpOptionsBuilder 构造函数

初始化一个新的“SATcpOptionsBuilder 类”一节第 413 页实例。

SATcpOptionsBuilder() 构造函数

初始化 SATcpOptionsBuilder 对象。

语法

Visual Basic

```
Public Sub New()
```

C#

```
public  SATcpOptionsBuilder();
```


注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SATcpOptionsBuilder 类。

示例

以下语句初始化 SATcpOptionsBuilder 对象。

```
SATcpOptionsBuilder options = new SATcpOptionsBuilder( );
```

另请参见

- “SATcpOptionsBuilder 类”一节第 413 页
- “SATcpOptionsBuilder 成员”一节第 414 页
- “SATcpOptionsBuilder 构造函数”一节第 416 页

SATcpOptionsBuilder(String) 构造函数

初始化 SATcpOptionsBuilder 对象。

语法

Visual Basic

```
Public Sub New( _
  ByVal options As String _
)
```

C#

```
public  SATcpOptionsBuilder(
  string  options
);
```

参数

- **options**　SQL Anywhere TCP 连接参数选项字符串。

  有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

注释

Restrictions: 无法在 .NET Compact Framework 2.0 中使用 SATcpOptionsBuilder 类。

示例

以下语句初始化 SATcpOptionsBuilder 对象。

```
SATcpOptionsBuilder options = new SATcpOptionsBuilder( );
```

另请参见

- “SATcpOptionsBuilder 类”一节第 413 页
- “SATcpOptionsBuilder 成员”一节第 414 页
- “SATcpOptionsBuilder 构造函数”一节第 416 页

Broadcast 属性

获取或设置 Broadcast 选项。

语法

Visual Basic

Public Property **Broadcast** As String

C#

public string **Broadcast** { get; set; }

另请参见

- “SATcpOptionsBuilder 类”一节第 413 页
- “SATcpOptionsBuilder 成员”一节第 414 页

BroadcastListener 属性

获取或设置 BroadcastListener 选项。

语法

Visual Basic

Public Property **BroadcastListener** As String

C#

public string **BroadcastListener** { get; set; }

另请参见

- “SATcpOptionsBuilder 类”一节第 413 页
- “SATcpOptionsBuilder 成员”一节第 414 页

ClientPort 属性

获取或设置 ClientPort 选项。

语法

Visual Basic

Public Property **ClientPort** As String

C#

public string **ClientPort** { get; set; }

另请参见

- “SATcpOptionsBuilder 类”一节第 413 页
- “SATcpOptionsBuilder 成员”一节第 414 页

DoBroadcast 属性

获取或设置 DoBroadcast 选项。

语法

Visual Basic

```
Public Property DoBroadcast As String
```

C#

```
public string  DoBroadcast { get; set; }
```

另请参见

- “SATcpOptionsBuilder 类”一节第 413 页
- “SATcpOptionsBuilder 成员”一节第 414 页

Host 属性

获取或设置 Host 选项。

语法

Visual Basic

```
Public Property Host As String
```

C#

```
public string  Host { get; set; }
```

另请参见

- “SATcpOptionsBuilder 类”一节第 413 页
- “SATcpOptionsBuilder 成员”一节第 414 页

IPV6 属性

获取或设置 IPV6 选项。

语法

Visual Basic

Public Property **IPV6** As String

C#

public string **IPV6** { get; set; }

另请参见

- “SATcpOptionsBuilder 类”一节第 413 页
- “SATcpOptionsBuilder 成员”一节第 414 页

LDAP 属性

获取或设置 LDAP 选项。

语法

Visual Basic

Public Property **LDAP** As String

C#

public string **LDAP** { get; set; }

另请参见

- “SATcpOptionsBuilder 类”一节第 413 页
- “SATcpOptionsBuilder 成员”一节第 414 页

LocalOnly 属性

获取或设置 LocalOnly 选项。

语法

Visual Basic

Public Property **LocalOnly** As String

C#

public string **LocalOnly** { get; set; }

另请参见

- “SATcpOptionsBuilder 类”一节第 413 页
- “SATcpOptionsBuilder 成员”一节第 414 页

MyIP 属性

获取或设置 MyIP 选项。

语法

Visual Basic

Public Property **MyIP** As String

C#

public string **MyIP** { get; set; }

另请参见

- “SATcpOptionsBuilder 类”一节第 413 页
- “SATcpOptionsBuilder 成员”一节第 414 页

ReceiveBufferSize 属性

获取或设置 ReceiveBufferSize 选项。

语法

Visual Basic

Public Property **ReceiveBufferSize** As Integer

C#

public int **ReceiveBufferSize** { get; set; }

另请参见

- “SATcpOptionsBuilder 类”一节第 413 页
- “SATcpOptionsBuilder 成员”一节第 414 页

SendBufferSize 属性

获取或设置 Send BufferSize 选项。

语法

Visual Basic

Public Property **SendBufferSize** As Integer

C#

public int **SendBufferSize** { get; set; }

另请参见

- “SATcpOptionsBuilder 类”一节第 413 页
- “SATcpOptionsBuilder 成员”一节第 414 页

ServerPort 属性

获取或设置 ServerPort 选项。

语法

Visual Basic

Public Property **ServerPort** As String

C#

public string **ServerPort** { get; set; }

另请参见

- “SATcpOptionsBuilder 类”一节第 413 页
- “SATcpOptionsBuilder 成员”一节第 414 页

TDS 属性

获取或设置 TDS 选项。

语法

Visual Basic

Public Property **TDS** As String

C#

public string **TDS** { get; set; }

另请参见

- “SATcpOptionsBuilder 类”一节第 413 页
- “SATcpOptionsBuilder 成员”一节第 414 页

Timeout 属性

获取或设置 Timeout 选项。

语法

Visual Basic

Public Property **Timeout** As Integer

C#

public int **Timeout** { get; set; }

另请参见

- “SATcpOptionsBuilder 类”一节第 413 页
- “SATcpOptionsBuilder 成员”一节第 414 页

VerifyServerName 属性

获取或设置 VerifyServerName 选项。

语法

Visual Basic

Public Property **VerifyServerName** As String

C#

public string **VerifyServerName** { get; set; }

另请参见

- “SATcpOptionsBuilder 类”一节第 413 页
- “SATcpOptionsBuilder 成员”一节第 414 页

ToString 方法

将 TcpOptionsBuilder 对象转换为字符串表示。

语法

Visual Basic

Public Overrides Function **ToString()** As String

C#

public override string **ToString();**

返回值

正在构建的选项字符串。

另请参见

- “SATcpOptionsBuilder 类”一节第 413 页
- “SATcpOptionsBuilder 成员”一节第 414 页

SATransaction 类

表示 SQL 事务。此类无法继承。

语法

Visual Basic

```
Public NotInheritable Class SATransaction
  Inherits DbTransaction
```

C#

```
public sealed class SATransaction : DbTransaction
```

注释

没有用于 SATransaction 的构造函数。要获取 SATransaction 对象，请使用一种 BeginTransaction 方法。要将命令与事务关联，请使用 SACommand.Transaction 属性。

有关详细信息，请参见“事务处理”一节第 129 页和“使用 SACommand 对象插入、更新和删除行”一节第 113 页。

另请参见

- “SATransaction 成员”一节第 424 页
- “BeginTransaction() 方法”一节第 242 页
- “BeginTransaction(SAIsolationLevel) 方法”一节第 243 页
- “Transaction 属性”一节第 203 页

SATransaction 成员

公共属性

成员名称	说明
Connection 属性	与事务关联的 SAConnection 对象；如果事务不再有效，则为空值引用（在 Visual Basic 中是 Nothing）。
IsolationLevel 属性	指定此事务的隔离级别。
SAIsolationLevel 属性	指定此事务的隔离级别。

公共方法

成员名称	说明
Commit 方法	提交数据库事务。
Dispose（继承自 DbTransaction）	释放 DbTransaction 使用的非托管资源。
Rollback 方法	将事务从待执行状态回退。
Save 方法	在事务中创建一个用于回退部分事务的保存点，并指定保存点的名称。

另请参见

- “SATransaction 类”一节第 424 页
- “BeginTransaction() 方法”一节第 242 页
- “BeginTransaction(SAIsolationLevel) 方法”一节第 243 页
- “Transaction 属性”一节第 203 页

Connection 属性

与事务关联的 SAConnection 对象；如果事务不再有效，则为空值引用（在 Visual Basic 中是 Nothing）。

语法

Visual Basic

```
Public Readonly Property Connection As SAConnection
```

C#

```
public SAConnection Connection { get;}
```

注释

单个应用程序可以有多个数据库连接，每个连接可以具有零个或多个事务。此属性让您可以确定与由 BeginTransaction 创建的特定事务关联的连接对象。

另请参见

- “SATransaction 类”一节第 424 页
- “SATransaction 成员”一节第 424 页

IsolationLevel 属性

指定此事务的隔离级别。

语法

Visual Basic

Public Overrides Readonly Property **IsolationLevel** As IsolationLevel

C#

public override IsolationLevel **IsolationLevel** { get;}

属性值

此事务的隔离级别。此类型可以是以下各项之一：

- ReadCommitted
- ReadUncommitted
- RepeatableRead
- Serializable
- Snapshot
- ReadOnlySnapshot
- StatementSnapshot

缺省值是 ReadCommitted。

另请参见

- “SATransaction 类”一节第 424 页
- “SATransaction 成员”一节第 424 页

SAIsolationLevel 属性

指定此事务的隔离级别。

语法

Visual Basic

Public Readonly Property **SAIsolationLevel** As SAIsolationLevel

C#

public SAIsolationLevel **SAIsolationLevel** { get;}

属性值

此事务的 IsolationLevel。此类型可以是以下各项之一：

- Chaos
- Read ReadCommitted
- ReadOnlySnapshot
- ReadUncommitted
- RepeatableRead
- Serializable
- Snapshot
- StatementSnapshot
- Unspecified

缺省值是 ReadCommitted。

注释

不支持并行事务。因此，IsolationLevel 将应用于整个事务。

另请参见

- “SATransaction 类”一节第 424 页
- “SATransaction 成员”一节第 424 页

Commit 方法

提交数据库事务。

语法

Visual Basic

Public Overrides Sub **Commit()**

C#

public override void **Commit();**

另请参见

- “SATransaction 类”一节第 424 页
- “SATransaction 成员”一节第 424 页

Rollback 方法

将事务从待执行状态回退。

Rollback() 方法

将事务从待执行状态回退。

语法

Visual Basic

```
Public Overrides Sub Rollback()
```

C#

```
public override void Rollback();
```

注释

此事务只能从待执行状态回退（调用 BeginTransaction 之后，但在调用 Commit 之前）。

另请参见

- “SATransaction 类”一节第 424 页
- “SATransaction 成员”一节第 424 页
- “Rollback 方法”一节第 427 页

Rollback(String) 方法

将事务从待执行状态回退。

语法

Visual Basic

```
Public Sub Rollback( _
  ByVal savePoint As String _
)
```

C#

```
public void Rollback(
  string  savePoint
);
```

参数

- **savePoint**　要回退到的保存点的名称。

注释

此事务只能从待执行状态回退（调用 BeginTransaction 之后，但在调用 Commit 之前）。

另请参见

- “SATransaction 类”一节第 424 页
- “SATransaction 成员”一节第 424 页
- “Rollback 方法”一节第 427 页

Save 方法

在事务中创建一个用于回退部分事务的保存点，并指定保存点的名称。

语法

Visual Basic

```
Public Sub Save( _
  ByVal savePoint As String _
)
```

C#

```
public void Save(
  string  savePoint
);
```

参数

- **savePoint**　要回退到的保存点的名称。

另请参见

- “SATransaction 类”一节第 424 页
- “SATransaction 成员”一节第 424 页

SQL Anywhere OLE DB 和 ADO 开发

目录

OLE DB 介绍

OLE DB 是 Microsoft 的数据访问模型。它使用组件对象模型 (COM) 接口，与 ODBC 不同的是，OLE DB 假定数据源不使用 SQL 查询处理器。

SQL Anywhere 包括一个名为 **SAOLEDB** 的 **OLE DB 提供程序**。该提供程序可用于当前 Windows 平台。该提供程序不可用于 Windows Mobile 平台。

还可以将用于 ODBC 的 Microsoft OLE DB 提供程序 (MSDASQL) 和 SQL Anywhere ODBC 驱动程序结合使用来访问 SQL Anywhere。

使用 SQL Anywhere OLE DB 提供程序具有以下几个优点：

- 使用 OLE DB/ODBC Bridge 无法利用某些功能，例如，通过游标更新。
- 如果使用 SQL Anywhere OLE DB 提供程序，则在部署中不需要 ODBC。
- MSDASQL 允许 OLE DB 客户端使用任何 ODBC 驱动程序，但不保证您能使用每个 ODBC 驱动程序的全部功能。而使用 SQL Anywhere 提供程序则可以从 OLE DB 编程环境访问 SQL Anywhere 的全部功能。

支持的平台

SQL Anywhere OLE DB 提供程序设计为使用 OLE DB 2.5 和更高版本。

有关受支持的平台的列表，请参见 http://www.sybase.com/detail?id=1062623。

分布式事务

OLE DB 驱动程序可在分布式事务环境中用作资源管理器。

有关详细信息，请参见“三层计算和分布式事务”第 61 页。

利用 SQL Anywhere 进行 ADO 编程

ADO（ActiveX 数据对象）是一种通过 Automation 接口公开的数据访问对象模型，它允许客户端应用程序在事先不了解对象的情况下，在运行时发现对象的方法和属性。Automation 接口允许脚本语言（如 Visual Basic）使用标准的数据访问对象模型。ADO 使用 OLE DB 提供数据访问。

使用 SQL Anywhere OLE DB 提供程序可以从 ADO 编程环境访问 SQL Anywhere 的全部功能。

本节介绍使用 Visual Basic 的 ADO 时如何执行基本任务。它不是使用 ADO 进行编程的完整指南。

本节中的代码示例可在 *samples-dir\SQLAnywhere\VBSampler\vbsampler.sln* 项目文件中找到。

有关在 ADO 中进行编程的信息，请参见开发工具的文档。

用 Connection 对象连接到数据库

本节介绍一个用于连接到数据库的简单的 Visual Basic 例程。

示例代码

通过将名为 Command1 的命令按钮放置到窗体上，并将该例程粘贴到它的 Click 事件中，可以尝试执行该例程。运行程序并单击该按钮，即可建立连接然后断开连接。

```
Private Sub cmdTestConnection_Click( _
        ByVal eventSender As System.Object, _
        ByVal eventArgs As System.EventArgs) _
        Handles cmdTestConnection.Click

    ' Declare variables
    Dim myConn As New ADODB.Connection
    Dim myCommand As New ADODB.Command
    Dim cAffected As Integer

    On Error GoTo HandleError

    ' Establish the connection
    myConn.Provider = "SAOLEDB"
    myConn.ConnectionString = _
        "Data Source=SQL Anywhere 11 Demo"
    myConn.Open()
    MsgBox("Connection succeeded")
    myConn.Close()
    Exit Sub

HandleError:
    MsgBox(ErrorToString(Err.Number))
    Exit Sub
End Sub
```

注意

该示例执行下列任务：

- 声明例程中使用的变量。
- 使用 SQL Anywhere OLE DB 提供程序与示例数据库建立连接。

- 使用 Command 对象执行一条简单的语句，用以在数据库服务器消息窗口中显示消息。
- 关闭连接。

SAOLEDB 提供程序在安装过程中自行注册。注册过程包括在注册表的 COM 部分创建注册表条目，以便 ADO 可以在 SAOLEDB 提供程序被调用时找到 DLL。如果更改 DLL 的位置，则必须重新注册。

♦ 注册 OLE DB 提供程序

1. 打开命令提示符。
2. 转换到安装 OLE DB 提供程序的目录。
3. 输入以下命令以注册该提供程序：

```
regsvr32 dboledb11.dll
regsvr32 dboledba11.dll
```

有关使用 OLE DB 连接数据库的详细信息，请参见“使用 OLE DB 连接到数据库”一节 《SQL Anywhere 服务器 - 数据库管理》。

用 Command 对象执行语句

本节介绍一个向数据库发送简单 SQL 语句的简单例程。

示例代码

通过将名为 Command2 的命令按钮放置到窗体上，并将该例程粘贴到它的 Click 事件中，可以尝试执行该例程。运行程序并单击该按钮，即可建立连接，在数据库服务器消息窗口中显示消息，然后断开连接。

```
Private Sub cmdUpdate_Click( _
        ByVal eventSender As System.Object, _
        ByVal eventArgs As System.EventArgs) _
        Handles cmdUpdate.Click

    ' Declare variables
    Dim myConn As New ADODB.Connection
    Dim myCommand As New ADODB.Command
    Dim cAffected As Integer

    On Error GoTo HandleError

    ' Establish the connection
    myConn.Provider = "SAOLEDB"
    myConn.ConnectionString = _
        "Data Source=SQL Anywhere 11 Demo"
    myConn.Open()

    'Execute a command
    myCommand.CommandText = _
        "UPDATE Customers SET GivenName='Liz' WHERE ID=102"
    myCommand.ActiveConnection = myConn
    myCommand.Execute(cAffected)
    MsgBox(CStr(cAffected) & " rows affected.", _
```



```
            MsgBoxStyle.Information)

        myConn.Close()
        Exit Sub

    HandleError:
        MsgBox(ErrorToString(Err.Number))
        Exit Sub
    End Sub
```

注意

建立连接之后，示例代码创建一个 Command 对象，将其 CommandText 属性设置为更新语句，并将其 ActiveConnection 属性设置为当前连接。然后执行更新语句，并在窗口中显示受更新操作影响的行数。

在本示例中，更新将在执行之后被发送到数据库并被提交。

有关在 ADO 内使用事务的信息，请参见“使用事务”一节第 438 页。

还可以通过游标执行更新。

有关详细信息，请参见“通过游标更新数据”一节第 437 页。

用 Recordset 对象查询数据库

ADO Recordset 对象表示查询的结果集。可以使用它查看数据库中的数据。

示例代码

通过将名为 cmdQuery 的命令按钮放置到窗体上，并将该例程粘贴到它的 Click 事件中，可以尝试执行该例程。运行程序并单击该按钮，即可建立连接，在数据库服务器消息窗口中显示消息，执行查询并在窗口中显示前几行，然后断开连接。

```
Private Sub cmdQuery_Click( _
        ByVal eventSender As System.Object, _
        ByVal eventArgs As System.EventArgs) _
        Handles cmdQuery.Click

    ' Declare variables
    Dim i As Integer
    Dim myConn As New ADODB.Connection
    Dim myCommand As New ADODB.Command
    Dim myRS As New ADODB.Recordset

    On Error GoTo ErrorHandler

    ' Establish the connection
    myConn.Provider = "SAOLEDB"
    myConn.ConnectionString = _
        "Data Source=SQL Anywhere 11 Demo"
    myConn.CursorLocation = _
        ADODB.CursorLocationEnum.adUseServer
    myConn.Mode = _
        ADODB.ConnectModeEnum.adModeReadWrite
    myConn.IsolationLevel = _
        ADODB.IsolationLevelEnum.adXactCursorStability
    myConn.Open()
```

```
        'Execute a query
        myRS = New ADODB.Recordset
        myRS.CacheSize = 50
        myRS.let_Source("SELECT * FROM Customers")
        myRS.let_ActiveConnection(myConn)
        myRS.CursorType = ADODB.CursorTypeEnum.adOpenKeyset
        myRS.LockType = ADODB.LockTypeEnum.adLockOptimistic
        myRS.Open()

        'Scroll through the first few results
        myRS.MoveFirst()
        For i = 1 To 5
            MsgBox(myRS.Fields("CompanyName").Value, _
                MsgBoxStyle.Information)
            myRS.MoveNext()
        Next

        myRS.Close()
        myConn.Close()
        Exit Sub

    ErrorHandler:
        MsgBox(ErrorToString(Err.Number))
        Exit Sub
    End Sub
```

注意

在本例中，Recordset 对象保存对 Customers 表执行查询的结果。For 循环滚动浏览前几行并显示每一行的 CompanyName 值。

这是一个从 ADO 使用游标的简单示例。

有关从 ADO 使用游标的更高级示例，请参见“使用 Recordset 对象”一节第 436 页。

使用 Recordset 对象

使用 SQL Anywhere 时，ADO Recordset 代表一个游标。可以通过在打开 Recordset 对象之前声明 Recordset 对象的 CursorType 属性来选择游标类型。所选游标类型控制可对 Recordset 执行的操作，并会对性能产生影响。

游标类型

ADO 有其自己对游标类型的命名约定。SQL Anywhere 支持的游标类型集在“游标属性”一节第 37 页中加以说明。

下面列出可用的游标类型、相应的游标类型常量以及与它们等价的 SQL Anywhere 类型：

ADO 游标类型	ADO 常量	SQL Anywhere 类型
动态游标	adOpenDynamic	动态滚动游标
键集游标	adOpenKeyset	滚动游标

ADO 游标类型	ADO 常量	SQL Anywhere 类型
静态游标	adOpenStatic	不敏感游标
只进游标	adOpenForwardOnly	非滚动游标

有关如何选择适合应用程序的游标类型的信息，请参见“选择游标类型”一节第 37 页。

示例代码

下列代码集为 ADO Recordset 对象设置游标类型：

```
Dim myRS As New ADODB.Recordset
myRS.CursorType = ADODB.CursorTypeEnum.adOpenDynamic
```

通过游标更新数据

SQL Anywhere OLE DB 提供程序允许通过游标更新结果集。该功能不能通过 MSDASQL 提供程序实现。

更新记录集

可以通过 Recordset 来更新数据库。

```
Private Sub cmdUpdateThroughCursor_Click( _
        ByVal eventSender As System.Object, _
        ByVal eventArgs As System.EventArgs) _
        Handles cmdUpdateThroughCursor.Click

    ' Declare variables
    Dim i As Integer
    Dim myConn As New ADODB.Connection
    Dim myRS As New ADODB.Recordset
    Dim SQLString As String

    On Error GoTo HandleError

    ' Connect
    myConn.Provider = "SAOLEDB"
    myConn.ConnectionString = _
        "Data Source=SQL Anywhere 11 Demo"
    myConn.Open()
    myConn.BeginTrans()
    SQLString = "SELECT * FROM Customers"
    myRS.Open(SQLString, myConn, _
        ADODB.CursorTypeEnum.adOpenDynamic, _
        ADODB.LockTypeEnum.adLockBatchOptimistic)

    If myRS.BOF And myRS.EOF Then
        MsgBox("Recordset is empty!", 16, "Empty Recordset")
    Else
        MsgBox("Cursor type: " & CStr(myRS.CursorType), _
            MsgBoxStyle.Information)
        myRS.MoveFirst()
        For i = 1 To 3
            MsgBox("Row: " & CStr(myRS.Fields("ID").Value), _
                MsgBoxStyle.Information)
```

```
                If i = 2 Then
                    myRS.Update("City", "Toronto")
                    myRS.UpdateBatch()
                End If
                myRS.MoveNext()
            Next i
            myRS.Close()
        End If
        myConn.CommitTrans()
        myConn.Close()
        Exit Sub

    HandleError:
        MsgBox(ErrorToString(Err.Number))
        Exit Sub

    End Sub
```

注意

如果对 Recordset 使用了 adLockBatchOptimistic 设置，则 myRS.Update 方法不会对数据库本身做任何更改。而是会更新 Recordset 的本地副本。

myRS.UpdateBatch 方法对数据库服务器进行更新，但是不提交更改，因为它位于事务内部。如果 UpdateBatch 方法是在事务外部调用的，则更改将被提交。

myConn.CommitTrans 方法提交所做的更改。Recordset 对象在此之前已被关闭，因此不存在数据的本地副本是否发生更改的问题。

使用事务

缺省情况下，使用 ADO 对数据库进行的任何更改都会在执行时被提交。这包括显式更新以及对 Recordset 执行的 UpdateBatch 方法。但是，上一节内容却说明了可以对 Connection 对象使用 BeginTrans、RollbackTrans 或 CommitTrans 方法以使用事务。

事务隔离级别作为 Connection 对象的属性进行设置。IsolationLevel 属性可具有下列值：

ADO 隔离级别	常量	SQL Anywhere 级别
未指定	adXactUnspecified	不适用。设置为 0
混沌	adXactChaos	不支持。设置为 0
浏览	adXactBrowse	0
未提交的读操作	adXactReadUncommitted	0
游标稳定性	adXactCursorStability	1
已提交的读操作	adXactReadCommitted	1
可重复的读操作	adXactRepeatableRead	2

ADO 隔离级别	常量	SQL Anywhere 级别
已隔离	adXactIsolated	3
可序列化	adXactSerializable	3
快照	2097152	4
语句快照	4194304	5
只读语句快照	8388608	6

有关隔离级别的详细信息，请参见“隔离级别和一致性”一节 《SQL Anywhere 服务器 - SQL 的用法》。

使用 OLE DB 设置 Microsoft 链接服务器

可创建一个能使用 SQL Anywhere OLE DB 提供程序来获得对 SQL Anywhere 数据库的访问权限的 Microsoft 链接服务器。可使用 Microsoft 的表引用语法（由 4 部分组成）或 Microsoft 的 OPENQUERY SQL 函数来发出 SQL 查询。下面就是一个由 4 部分组成的语法的示例。

```
SELECT * FROM SADATABASE..GROUPO.Customers
```

在本例中，SADATABASE 是链接服务器的名称，GROUPO 是 SQL Anywhere 数据库中的表所有者，而 Customers 是 SQL Anywhere 数据库中的表名。由于分类名不是 SQL Anywhere 数据库的功能，因此被省略（由两个连续点表示）。

另一种形式是使用 Microsoft 的 OPENQUERY 函数。

```
SELECT * FROM OPENQUERY( SADATABASE, 'SELECT * FROM Customers' )
```

在 OPENQUERY 语法中，第二个 SELECT 语句 ('SELECT * FROM Customers') 被传递到 SQL Anywhere 服务器中执行。

要对使用 SQL Anywhere OLE DB 提供程序的链接服务器进行设置，必须遵循几个步骤。

♦ 设置链接服务器

1. 填充 [**常规**] 页。

   [**常规**] 页上的 [**链接服务器**] 字段应包含**链接服务器**的名称（如上面使用的 SADATABASE）。应选择 [**其他数据源**] 选项，并应从列表中选择 [SQL Anywhere OLE DB 访问接口]。[**产品名称**] 字段应当包含 ODBC 数据源名（例如，SQL Anywhere 11 Demo）.[**访问接口字符串**] 字段可包含其它连接参数，如用户 ID 和口令（例如，uid=DBA;pwd=sql）。其它字段（如 [**常规**] 页上的 [**数据源**]）应当留空。

2. 选择 [**允许进程内**] 访问接口选项。

   完成该操作的方法因 Microsoft SQL Server 的版本而异。在 SQL Server 2000 中有一个 [**访问接口选项**] 按钮，可将您引至能选择此选项的页面。在 SQL Server 2005 中，当您右击 [**链接服务器/访问接口**] 树形视图中的 [SAOLEDB 访问接口] 并选择 [**属性**] 时，会出现一个全局 [**允许进程内**] 复选框。如果未选择 [**进程内**] 选项，则查询将失败。

3. 选择 [**RPC**] 和 [**RPC 输出**] 选项。

   完成该操作的方法因 Microsoft SQL Server 的版本而异。在 SQL Server 2000 中，必须为这两个选项选中两个复选框。这些复选框位于 [**服务器选项**] 页上。在 SQL Server 2005 中，这些选项具有 True/False 设置。确保将它们设置为 True。如果想要在 SQL Anywhere 数据库中执行存储过程/函数调用，并成功地传递参数（传入和传出），则必须设置**远程过程调用**（Remote Procedure Call，简称 **RPC**）选项。

支持的 OLE DB 接口

OLE DB API 由一组接口组成。下表介绍 SQL Anywhere OLE DB 驱动程序中对每个接口的支持。

接口	作用	限制
IAccessor	定义客户端内存和数据存储值之间的捆绑。	不支持 DBACCESSOR_PASSBYREF。不支持 DBACCESSOR_OPTIMIZED。
IAlterIndex IAlterTable	变更表、索引和列。	不支持。
IChapteredRowset	分段的行集允许以单独的段访问行集中的行。	不支持。SQL Anywhere 不支持分段的行集。
IColumnsInfo	获得有关行集的列的简单信息。	支持。
IColumnsRowset	获得有关行集内可选元数据列的信息，并获得列元数据的行集。	支持。
ICommand	执行 SQL 语句。	不支持调用。ICommandProperties：带有 DBPROPSET_PROPERTIESINERROR 的 GetProperties，用于查找尚未设置的属性。
ICommandPersist	保持命令对象（而非任何活动行集）的状态。随后可使用 PROCEDURES 或 VIEWS 行集枚举这些持久性命令对象。	支持。
ICommandPrepare	准备命令。	支持。
ICommandProperties	为由命令创建的行集设置 [行集] 属性。最常用于指定行集应当支持的接口。	支持。
ICommandText	为 ICommand 设置 SQL 语句文本。	只支持 DBGUID_DEFAULT SQL 方言。
ICommandWithParameters	为命令设置或获取参数信息。	不支持以标量值的矢量形式存储的参数。 不支持 BLOB 参数。

接口	作用	限制
IConvertType		支持。
IDBAsynchNotify IDBAsyncStatus	异步处理。 在数据源初始化、填充行集等的异步处理时，向客户端通知事件。	不支持。
IDBCreateCommand	从会话创建命令。	支持。
IDBCreateSession	从数据源对象创建会话。	支持。
IDBDataSourceAdmin	创建/破坏/修改数据源对象：由客户端使用的 COM 对象。该接口不用于管理数据存储区（数据库）。	不支持。
IDBInfo	查找该提供程序所特有的关键字的信息（即，查找非标准的 SQL 关键字）。 还查找有关文字、文本匹配查询中使用的特殊字符、以及其它文字信息。	支持。
IDBInitialize	初始化数据源对象和枚举器。	支持。
IDBProperties	管理数据源对象或枚举器的属性。	支持。
IDBSchemaRowset	以标准格式（行集）获取有关系统表的信息。	支持。
IErrorInfo IErrorLookup IErrorRecords	支持 ActiveX 错误对象。	支持。
IGetDataSource	将接口指针返回会话的数据源对象。	支持。
IIndexDefinition	在数据存储区创建或删除索引。	不支持。
IMultipleResults	从命令检索多个结果（行集或行计数）。	支持。
IOpenRowset	按照数据库表的名称访问数据库表的非 SQL 方式。	支持。 支持按照表名打开表，不支持按照 GUID 打开表。
IParentRowset	访问分段/分层行集。	不支持。

接口	作用	限制
IRowset	访问行集。	支持。
IRowsetChange	允许更改行集数据，并将所做更改反映回数据存储区。 未实现用于 BLOB 的 InsertRow/SetData。	支持。
IRowsetChapterMember	访问分段/分层行集。	不支持。
IRowsetCurrentIndex	动态更改行集的索引。	不支持。
IRowsetFind	在行集中查找与指定值匹配的行。	不支持。
IRowsetIdentity	比较行的句柄。	不支持。
IRowsetIndex	访问数据库索引。	不支持。
IRowsetInfo	查找有关行集属性的信息或者查找创建了行集的对象。	支持。
IRowsetLocate	使用书签定位到行集的行上。	支持。
IRowsetNotify	为行集事件提供 COM 回调接口。	支持。
IRowsetRefresh	获取事务可看到的数据的最新值。	不支持。
IRowsetResynch	旧的 OLEDB 1.x 接口，被 IRowsetRefresh 替代。	不支持。
IRowsetScroll	滚动浏览行集以抓取行数据。	不支持。
IRowsetUpdate	延迟对行集数据的更改，直到 Update 被调用。	支持。
IRowsetView	使用现有行集上的视图。	不支持。
ISequentialStream	检索 BLOB 列。	只支持读取。 不支持 对该接口执行 SetData。
ISessionProperties	获得会话属性信息。	支持。
ISourcesRowset	获得数据源对象和枚举器的行集。	支持。

接口	作用	限制
ISQLErrorInfo ISupportErrorInfo	支持 ActiveX 错误对象。	支持。
ITableDefinition ITableDefinitionWithConstraints	用约束来创建、删除和变更表。	支持。
ITransaction	提交或中止事务。	并非所有的标志都受支持。
ITransactionJoin	支持分布式事务。	并非所有的标志都受支持。
ITransactionLocal	处理会话事务。 并非所有的标志都受支持。	支持。
ITransactionOptions	获得或设置事务的选项。	支持。
IViewChapter	使用现有行集上的视图，专用于对行应用后处理过滤/排序。	不支持。
IViewFilter	将行集的内容限制为与条件集相匹配的行。	不支持。
IViewRowset	在打开行集时，将行集的内容限制为与条件集相匹配的行。	不支持。
IViewSort	对视图应用排序顺序。	不支持。

SQL Anywhere ODBC API

目录

ODBC 简介

开放式数据库连接 (ODBC) 接口是一个应用程序编程接口，由 Microsoft Corporation 定义为 Windows 操作系统上数据库管理系统的标准接口。ODBC 是基于调用的接口。

要编写用于 SQL Anywhere 的 ODBC 应用程序，您需要：

- SQL Anywhere。
- 能够为您的环境创建程序的 C 编译器。
- Microsoft ODBC 软件开发工具包。在 Microsoft 开发人员网络上可以找到该工具包，该工具包提供测试 ODBC 应用程序时所用的文档及其它工具。

支持的平台

除了支持 Windows 上的 ODBC API 之外，SQL Anywhere 还支持 Unix 和 Windows Mobile 上的 ODBC API。由于有了多平台 ODBC 支持，便携式数据库应用程序开发变得更加容易。

有关在分布式事务中征用 ODBC 驱动程序的信息，请参见“三层计算和分布式事务”第 61 页。

另请参见

- Microsoft 开放式数据库连接 (ODBC)

> **注意**
> 有些已经具有 ODBC 支持的应用程序开发工具提供了它们自己的编程接口，这类接口隐藏了 ODBC 接口。SQL Anywhere 文档不介绍如何使用这些工具。

对 ODBC 的支持

SQL Anywhere 提供对 ODBC 3.5 的支持，这是作为 Microsoft Data Access Kit 2.7 的一部分提供的。

ODBC 支持的级别

ODBC 功能是根据符合的级别进行组织的。这些功能为**核心**、**级别 1** 或**级别 2**，级别 2 是 ODBC 支持的最完整级别。Microsoft ODBC 程序员参考中列出了这些功能。

SQL Anywhere 支持的功能

SQL Anywhere 对 ODBC 3.5 规范的支持如下所示：

- **核心支持**　SQL Anywhere 支持所有 [核心] 级别功能。
- **级别 1 支持**　除了 ODBC 函数的异步执行以外，SQL Anywhere 支持所有 [级别 1] 功能。

  SQL Anywhere 支持多个线程共享一个连接。SQL Anywhere 序列化来自不同线程的请求。
- **级别 2 支持**　SQL Anywhere 支持除了以下几项以外的所有其它 [级别 2] 功能：
  - 表和视图的由三个部分构成的名称。这不适用于 SQL Anywhere。

- 指定的个别语句的 ODBC 函数异步执行。
- 使登录请求和 SQL 查询超时的能力。

ODBC 向后兼容性

使用旧版本 ODBC 开发的应用程序能继续与 SQL Anywhere 和更新版本的 ODBC 驱动程序管理器一起工作。旧应用程序不具备新的 ODBC 功能。

ODBC 驱动程序管理器

Microsoft Windows 包括一个 ODBC 驱动程序管理器。对于 Unix，ODBC 驱动程序管理器随 SQL Anywhere 提供。

创建 ODBC 应用程序

本节介绍如何编译和链接简单的 ODBC 应用程序。

包括 ODBC 头文件

每个调用 ODBC 函数的 C 源文件都必须包括一个平台特定的 ODBC 头文件。每个特定于平台的头文件都包括主要的 ODBC 头文件 *odbc.h*，该头文件定义了编写 ODBC 程序所需的所有函数、数据类型和常量定义。

♦ 将 ODBC 头文件包括在 C 源文件中

1. 向源文件添加 include 行，该行引用相应的平台特定的头文件。要使用的行如下所示：

操作系统	Include 行
Windows	#include "ntodbc.h"
Unix	#include "unixodbc.h"
Windows Mobile	#include "ntodbc.h"

2. 将包含头文件的目录添加到您的编译器的包括路径中。

   特定于平台的头文件和 *odbc.h* 都安装在 SQL Anywhere 安装目录的 *SDK\Include* 子目录下。

3. 为 Unix 创建 ODBC 应用程序时，可能需要为 32 位应用程序定义宏 "UNIX" 或为 64 位应用程序定义宏 "UNIX64"，以便获得正确的数据对齐方式和大小。如果您使用的是以下受支持的编译器之一，则不需要此步骤：

   - 适用于任何受支持平台的 GNU C/C++ 编译器
   - 适用于 Linux 的 Intel C/C++ 编译器 (icc)
   - 适用于 Linux 或 Solaris 的 SunPro C/C++ 编译器
   - 适用于 AIX 的 VisualAge C/C++ 编译器
   - 适用于 HP-UX 的 C/C++ 编译器 (cc/aCC)

在 Windows 上链接 ODBC 应用程序

本节不适用于 Windows Mobile。

有关 Windows Mobile 的信息，请参见“在 Windows Mobile 上链接 ODBC 应用程序”一节 第 449 页。

链接应用程序时，必须链接到适当的导入库文件，才能访问 ODBC 函数。导入库为 ODBC 驱动程序管理器 *odbc32.dll* 定义入口点。驱动程序管理器进而装载 SQL Anywhere ODBC 驱动程序 *dbodbc11.dll*。

♦ **链接 ODBC 应用程序 (Windows)**

1. 将包含特定于平台的导入库的目录添加到 LIB 环境变量的库目录列表中。

   通常，导入库存储在 Microsoft 平台 SDK 的 *Lib* 目录结构下：

操作系统	导入库
Windows（32 位）	*Lib\odbc32.lib*
Windows（64 位）	*Lib\x64\odbc32.lib*

   下面是一个命令提示符的示例。

```
set LIB=%LIB%;C:\mssdk\v6.1\lib
```

2. 要使用 Microsoft 编译和链接工具编译和链接一个简单的 ODBC 应用程序，可以在命令提示符处发出单个命令。以下示例编译并链接存储在 *odbc.c* 中的应用程序。

```
cl odbc.c /Ic:\sa11\SDK\Include odbc32.lib
```

在 Windows Mobile 上链接 ODBC 应用程序

在 Windows Mobile 操作系统上没有 ODBC 驱动程序管理器。导入库 (*dbodbc11.lib*) 直接定义进入 SQL Anywhere ODBC 驱动程序 *dbodbc11.dll* 的入口点。此文件位于 SQL Anywhere 安装目录的 *SDK\Lib\CE\Arm.50* 子目录中。

由于 Windows Mobile 没有 ODBC 驱动程序管理器，因此必须在提供给 SQLDriverConnect 函数的链接字符串中以 "DRIVER= 参数" 的形式指定 SQL Anywhere ODBC 驱动程序 DLL 的位置。以下是一个示例。

```
szConnStrIn = "driver=ospath\\dbodbc11.dll;dbf=\\samples-dir\\demo.db"
```

这里，*ospath* 是 Windows Mobile 设备上 Windows 目录的完整路径。例如：

```
\\Windows
```

♦ **链接 ODBC 应用程序 (Windows Mobile)**

● 将包含平台特定的导入库的目录添加到库目录列表中。

有关支持的 Windows Mobile 版本列表，请参见 http://www.sybase.com/detail?id=1062617 下的 "SQL Anywhere 支持的 PC 平台" 表。

示例程序 (*odbc_sample.cpp*) 使用一个名为 *SQL Anywhere 11 Demo.dsn* 的文件数据源（FileDSN 连接参数）。该文件在您将 SQL Anywhere for Windows Mobile 安装到设备上时被放置在 Windows Mobile 设备的根目录下。您可以通过 ODBC 数据源管理器在桌面系统上创建文件数据源，但是必须为桌面环境设置文件数据源，然后对其进行编辑使之符合 Windows Mobile 环境。在进行适当的编辑后，可将它们复制到 Windows Mobile 设备。

有关 *samples-dir* 缺省位置的信息，请参见“示例目录”一节 《SQL Anywhere 服务器 - 数据库管理》。

Windows Mobile 和 Unicode

SQL Anywhere 使用称为 UTF-8 的编码方式，这是一种多字节字符编码技术，可用于进行 Unicode 编码。

SQL Anywhere ODBC 驱动程序支持 ASCII（8 位）字符串或 Unicode 代码（宽字符）字符串。UNICODE 宏控制 ODBC 函数将使用 ASCII 还是 Unicode 字符串。如果您必须在已定义 UNICODE 宏的情况下创建应用程序，而您又要使用 ASCII ODBC 函数，则还必须定义 SQL_NOUNICODEMAP 宏。

示例文件 *samples-dir\SQLAnywhere\C\odbc.c* 说明了如何使用 Unicode ODBC 功能。

在 Unix 上链接 ODBC 应用程序

ODBC 驱动程序管理器随 SQL Anywhere 提供，并且有第三方驱动程序管理器可供使用。本节说明介绍创建不使用 ODBC 驱动程序管理器的 ODBC 应用程序。

ODBC 驱动程序

ODBC 驱动程序是一个共享对象或者共享库。针对单线程应用程序和多线程应用程序，提供的 SQL Anywhere ODBC 驱动程序的版本是不同的。提供通用的 SQL Anywhere ODBC 驱动程序，此驱动程序将检测到正在使用的线程模型，并管理对适当单线程库或多线程库的调用。

ODBC 驱动程序就是下面的文件：

操作系统	线程模型	ODBC 驱动程序
（除 Mac OS X 和 HP-UX 以外的所有 Unix）	通用	*libdbodbc11.so (libdbodbc11.so.1)*
（除 Mac OS X 和 HP-UX 以外的所有 Unix）	单线程	*libdbodbc11_n.so (libdbodbc11_n.so.1)*
（除 Mac OS X 和 HP-UX 以外的所有 Unix）	多线程	*libdbodbc11_r.so (libdbodbc11_r.so.1)*
HP-UX	通用	*libdbodbc11.sl (libdbodbc11.sl.1)*
HP-UX	单线程	*libdbodbc11_n.sl (libdbodbc11_n.sl.1)*
HP-UX	多线程	*libdbodbc11_r.sl (libdbodbc11_r.sl.1)*
Mac OS X	通用	*libdbodbc11.dylib*
Mac OS X	单线程	*libdbodbc11_n.dylib*
Mac OS X	多线程	*libdbodbc11_r.dylib*

库是作为符号链接进行安装的，这些链接指向带有版本号（显示在括号中）的共享库。

此外，还为 Mac OS X 提供了以下软件包：

操作系统	线程模型	ODBC 驱动程序
Mac OS X	单线程	*dbodbc11.bundle*
Mac OS X	多线程	*dbodbc11_r.bundle*

♦ **链接 ODBC 应用程序 (Unix)**

1. 将应用程序链接到通用 ODBC 驱动程序 *libdbodbc11*。
2. 部署应用程序时，请确保用户的库路径中有适当的（或所有）ODBC 驱动程序版本（非线程或线程）。

数据源信息

如果 SQL Anywhere 未检测到 ODBC 驱动程序管理器，它会将系统信息文件用于数据源信息。请参见“在 Unix 上使用 ODBC 数据源”一节 《SQL Anywhere 服务器 - 数据库管理》。

在 Unix 上使用 SQL Anywhere ODBC 驱动程序管理器

SQL Anywhere 包括用于 Unix 的 ODBC 驱动程序管理器。*libdbodm11* 共享对象可在所有受支持的 Unix 平台上用作 ODBC 驱动程序管理器。iAnywhere ODBC 驱动程序管理器可用于装载 3.0 或更高版本的任何 ODBC 驱动程序。驱动程序管理器不在 ODBC 1.0/2.0 调用和 ODBC 3.x 调用之间执行映射；因此，使用 iAnywhere ODBC 驱动程序管理器的应用程序必须将它们对 ODBC 功能集的使用限制为 3.0 或更高版本。同时，线程应用程序和非线程应用程序都可以使用 iAnywhere ODBC 驱动程序管理器。

iAnywhere ODBC 驱动程序管理器可以为任何给定连接执行对 ODBC 调用的跟踪。要开启跟踪，用户可以使用 TraceLevel 和 TraceLog 指令。这些指令可以是连接字符串的一部分（在使用 SQLDriverConnect 的情况下），也可以位于 DSN 条目内。TraceLog 是执行连接的跟踪输出的日志文件，而 TraceLevel 则是所需的跟踪信息量。跟踪级别是：

- **NONE** 不输出跟踪信息。
- **MINIMAL** 输出中包括常规名称和参数。
- **LOW** 输出中包括常规名称和参数，以及返回值。
- **MEDIUM** 输出中包括常规名称、参数、返回值和执行日期与时间。
- **HIGH** 输出中包含常规名称、参数、返回值、执行日期与时间，以及参数类型。

用户还可以获得为 Unix 提供的第三方 ODBC 驱动程序管理器。有关这些驱动程序管理器的使用信息，请查阅其随附的文档。

使用 unixODBC 驱动程序管理器

unixODBC 在版本 2.2.13 之前发行的版本错误地实现了由 Microsoft 所定义的 64 位 ODBC 规范中的某些方面。将 unixODBC 驱动程序管理器与 SQL Anywhere 64 位 ODBC 驱动程序一起使用时，这些差异会导致问题。

要避免这些问题，您应了解这些差异。差异之一是 SQLLEN 和 SQLULEN 的定义。在 Microsoft 64 位 ODBC 规范中，这些定义是 64 位类型的，并且是 SQL Anywhere 64 位 ODBC 驱动程序应使用的 64 位类型。而 unixODBC 的某些实现方式将这两种类型定义为 32 位，在连接到 SQL Anywhere 64 位 ODBC 驱动程序时，这会导致问题。

要避免在 64 位平台上出现问题，您必须做三件事。

1. 您应包括 SQL Anywhere ODBC 头文件 *unixodbc.h*，而不是包括 unixODBC 标头（例如 *sql.h* 和 *sqlext.h*）。这将保证您有正确的 SQLLEN 和 SQLULEN 定义。在 unixODBC 2.2.13 中，由于头文件的发布，该问题得到了更正。
2. 您必须确保已使用了所有参数的正确类型。正确头文件的使用以及 C/C++ 编译器强大的类型检查功能会在这方面有所帮助。还必须确保您已为所有由 SQL Anywhere 驱动程序通过指针间接设置的变量使用了正确的类型。请参见“64 位 ODBC 注意事项”一节第 466 页。
3. 您不得使用 2.2.13 发行版以前的 unixODBC 驱动程序管理器版本。相反，您可以直接链接到 SQL Anywhere ODBC 驱动程序。例如，确保 libodbc 共享对象链接到 SQL Anywhere 驱动程序。

   ```
   libodbc.so.1 -> libdbodbc11_r.so.1
   ```

   或者，也可以在平台上使用 SQL Anywhere 驱动程序管理器（如果该平台具有该程序）。请参见“在 Unix 上使用 SQL Anywhere ODBC 驱动程序管理器”一节第 451 页。

有关详细信息，请参见“在 Unix 上链接 ODBC 应用程序”一节第 450 页。

ODBC 示例

SQL Anywhere 中附带了几个 ODBC 示例。可在 *samples-dir\SQLAnywhere* 子目录中找到这些示例。

目录中这些以 ODBC 开头的示例说明单独和简单的 ODBC 任务，例如连接到数据库和执行语句。*samples-dir\SQLAnywhere\C\odbc.c* 中提供了一个完整的 ODBC 示例程序。此程序执行的操作与同一目录中嵌入式 SQL 动态游标示例程序执行的操作相同。

有关关联的嵌入式 SQL 程序的说明，请参见“示例嵌入式 SQL 程序”一节第 516 页。

创建示例 ODBC 程序

ODBC 示例程序位于 *samples-dir\SQLAnywhere\C* 中。该程序包括一个可用于编译和链接此文件夹中所有示例应用程序的批处理文件（对于 Unix 是 shell 脚本）。

♦ 创建示例 ODBC 程序

1. 打开命令提示符，将目录更改为 *samples-dir\SQLAnywhere\C* 目录。
2. 运行 *build.bat* 或 *build64.bat* 批处理文件。

   对于 x64 平台版本，您可能需要设置正确的编译和链接环境。以下示例构建了一个适用于 x64 平台的示例程序。

   ```
   set mssdk=c:\MSSDK\v6.1
   build64
   ```

♦ 构建适用于 Unix 的 ODBC 示例程序

1. 打开命令 shell，将目录更改为 *samples-dir\SQLAnywhere\C* 目录。
2. 运行 *build.sh* shell 脚本。

运行示例 ODBC 程序

♦ 运行 ODBC 示例

1. 启动该程序：
   - 对于 32 位 Windows，运行文件 *samples-dir\SQLAnywhere\C\odbcwin.exe*。
   - 对于 64 位 Windows，运行文件 *samples-dir\SQLAnywhere\C\odbcx64.exe*。
   - 对于 Unix，运行文件 *samples-dir\SQLAnywhere\C\odbc*。
2. 选择表：
   - 选择示例数据库中的一个表。例如，可以输入 **Customers** 或 **Employees**。

ODBC 句柄

ODBC 应用程序使用一小组**句柄**来定义基本功能，例如数据库连接和 SQL 语句。句柄是一个 32 位的值。

下面的句柄基本上用于所有 ODBC 应用程序：

- **环境** 环境句柄提供一个在其中访问数据的全局上下文。每个 ODBC 应用程序必须在启动时分配一个（只一个）环境句柄，在结束时必须将其释放。

  下面的代码说明如何分配环境句柄：

  ```
  SQLHENV env;
  SQLRETURN rc;
  rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL
     _NULL_HANDLE, &env );
  ```

- **连接** 连接是由 ODBC 驱动程序和数据源指定的。应用程序可以具有几个与它的环境相关联的连接。分配连接句柄并不会建立连接；必须首先分配连接句柄，然后才能在建立连接时使用它。

  下面的代码说明如何分配连接句柄：

  ```
  SQLHDBC  dbc;
  SQLRETURN rc;
  rc = SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
  ```

- **语句** 语句句柄提供对 SQL 语句以及与之相关联的任何信息（如结果集和参数）的访问。每个连接可以有多个语句。在游标操作（读取数据）和单个语句执行（例如 INSERT、UPDATE 和 DELETE）中都使用语句。

  下面的代码说明如何分配语句句柄：

  ```
  SQLHSTMT stmt;
  SQLRETURN rc;
  rc = SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
  ```

分配 ODBC 句柄

ODBC 程序所需的句柄类型现列示如下：

项	句柄类型
环境	SQLHENV
连接	SQLHDBC
语句	SQLHSTMT
描述符	SQLHDESC

♦ **使用 ODBC 句柄：**

1. 调用 SQLAllocHandle 函数。

   SQLAllocHandle 采用以下参数：

   - 要分配的项所属类型的标识符
   - 父项的句柄
   - 要分配的句柄的位置指针

     有关完整说明，请参见 Microsoft *ODBC 程序员参考*中的 SQLAllocHandle。
2. 在随后的函数调用中使用该句柄。
3. 使用 SQLFreeHandle 释放对象。

   SQLFreeHandle 采用以下参数：

   - 要释放的项所属类型的标识符
   - 要释放的项的句柄

     有关完整说明，请参见 Microsoft《ODBC 程序员参考》中的 SQLFreeHandle。

示例

下面的代码段将分配和释放一个环境句柄：

```
SQLHENV env;
SQLRETURN retcode;
retcode = SQLAllocHandle(
      SQL_HANDLE_ENV,
      SQL_NULL_HANDLE,
      &env );
if( retcode == SQL_SUCCESS
    || retcode == SQL_SUCCESS_WITH_INFO ) {
  // success: application Code here
}
SQLFreeHandle( SQL_HANDLE_ENV, env );
```

有关返回代码和错误处理的详细信息，请参见“处理错误”一节第 478 页。

第一个 ODBC 示例

以下是一个简单的 ODBC 程序，该程序连接到 SQL Anywhere 示例数据库，然后立即断开连接。

可在 *samples-dir\SQLAnywhere\ODBCConnect\odbcconnect.cpp* 中找到此示例。

```
#include <stdio.h>
#include "ntodbc.h"

int main(int argc, char* argv[])
{
  SQLHENV  env;
  SQLHDBC  dbc;
  SQLRETURN retcode;

  retcode = SQLAllocHandle( SQL_HANDLE_ENV,
```

```
            SQL_NULL_HANDLE,
            &env );
  if (retcode == SQL_SUCCESS
      || retcode == SQL_SUCCESS_WITH_INFO) {
    printf( "env allocated\n" );
    /* Set the ODBC version environment attribute */
    retcode = SQLSetEnvAttr( env,
                  SQL_ATTR_ODBC_VERSION,
                  (void*)SQL_OV_ODBC3, 0);
      retcode = SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
      if (retcode == SQL_SUCCESS
          || retcode == SQL_SUCCESS_WITH_INFO) {
        printf( "dbc allocated\n" );
        retcode = SQLConnect( dbc,
          (SQLCHAR*) "SQL Anywhere 11 Demo", SQL_NTS,
          (SQLCHAR* ) "DBA", SQL_NTS,
          (SQLCHAR*) "sql", SQL_NTS );
        if (retcode == SQL_SUCCESS
           || retcode == SQL_SUCCESS_WITH_INFO) {
          printf( "Successfully connected\n" );
        }
        SQLDisconnect( dbc );
      }
      SQLFreeHandle( SQL_HANDLE_DBC, dbc );
    }
    SQLFreeHandle( SQL_HANDLE_ENV, env );
  return 0;
}
```


选择 ODBC 连接函数

ODBC 提供了一组连接函数。使用哪一个连接函数，取决于您希望如何部署和使用应用程序：

- **SQLConnect** 最简单的连接函数。

  SQLConnect 接收数据源名，以及用户 ID 和口令（可选）。如果您将数据源名硬编码到应用程序中，则可能想要使用 SQLConnect。

  有关详细信息，请参见 Microsoft *ODBC 程序员参考*中的 SQLConnect。

- **SQLDriverConnect** 使用连接字符串连接数据源。

  SQLDriverConnect 允许应用程序使用数据源外部的、特定于 SQL Anywhere 的连接信息。另外，您可以使用 SQLDriverConnect 来请求 SQL Anywhere 驱动程序提示用户提供连接信息。

  SQLDriverConnect 还可用来在不指定数据源的情况下进行连接。

  有关详细信息，请参见 Microsoft *ODBC 程序员参考*中的 SQLDriverConnect。

- **SQLBrowseConnect** 使用连接字符串（如 SQLDriverConnect）连接到数据源。

  SQLBrowseConnect 允许您的应用程序创建一些自己的、具有以下功能的窗口：提示用户提供连接信息，以及浏览由特定驱动程序（这里是 SQL Anywhere 驱动程序）使用的数据源。

  有关详细信息，请参见 Microsoft *ODBC 程序员参考*中的 SQLBrowseConnect。

有关可在连接字符串中使用的连接参数的完整列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

建立连接

您的应用程序必须先建立连接，然后才能执行任何数据库操作。

♦ **建立 ODBC 连接：**

1. 分配 ODBC 环境。

   例如：

   ```
   SQLHENV    env;
   SQLRETURN retcode;
   retcode = SQLAllocHandle( SQL_HANDLE_ENV,
     SQL_NULL_HANDLE, &env );
   ```

2. 声明 ODBC 版本。

   通过声明该应用程序遵循 ODBC 版本 3，SQLSTATE 值和某些其它与版本相关的功能都将被设置为适当的行为。例如：

   ```
   retcode = SQLSetEnvAttr( env,
     SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);
   ```

3. 如果需要，可以汇编这些数据源或连接字符串。

根据您的应用程序的情况，您可以对数据源或连接字符串进行硬编码，也可以将它们存储在外部以获得更大的灵活性。

4. 分配 ODBC 连接项。

   例如：

```
retcode = SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
```

5. 设置任何必须在连接前设置的连接属性。

   有些连接特性必须在建立连接前或建立连接后进行设置，而其它连接特性则既可在之前设置也可在之后设置。SQL_AUTOCOMMIT 特性是一个既可在之前设置也可在之后设置的特性：

```
retcode = SQLSetConnectAttr( dbc,
   SQL_AUTOCOMMIT,
   (SQLPOINTER)SQL_AUTOCOMMIT_OFF, 0 );
```

   有关详细信息，请参见“设置连接属性”一节第 458 页。

6. 调用 ODBC 连接函数。

   例如：

```
if (retcode == SQL_SUCCESS
    || retcode == SQL_SUCCESS_WITH_INFO) {
   printf( "dbc allocated\n" );
   retcode = SQLConnect( dbc,
      (SQLCHAR*) "SQL Anywhere 11 Demo", SQL_NTS,
      (SQLCHAR* ) "DBA", SQL_NTS,
      (SQLCHAR*) "sql", SQL_NTS );
   if (retcode == SQL_SUCCESS
         || retcode == SQL_SUCCESS_WITH_INFO){
   // successfully connected.
```

可在 *samples-dir\SQLAnywhere\ODBCConnect\odbcconnect.cpp* 中找到完整示例。

注意

- **SQL_NTS**　每个传递到 ODBC 的字符串都有相应的长度。如果长度未知，则可传递 SQL_NTS，表示它是一个结尾标记为空值字符 (\0) 的**以空值终止的字符串**。
- **SQLSetConnectAttr**　缺省情况下，ODBC 在自动提交模式下工作。通过将 SQL_AUTOCOMMIT 设置为 false 可关闭此模式。

  有关详细信息，请参见“设置连接属性”一节第 458 页。

设置连接属性

SQLSetConnectAttr 函数用于控制连接的详细信息。例如，以下语句可关闭 ODBC 自动提交行为。

```
retcode = SQLSetConnectAttr( dbc, SQL_AUTOCOMMIT,
   (SQLPOINTER)SQL_AUTOCOMMIT_OFF, 0 );
```

有关详细信息（包括连接属性列表），请参见 Microsoft *ODBC 程序员参考*中的 SQLSetConnectAttr。

通过连接参数可以控制连接的许多方面。有关信息，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

获取连接属性

SQLGetConnectAttr 函数用于获取连接的详细信息。例如，以下语句返回连接状态。

```
retcode = SQLGetConnectAttr( dbc, SQL_ATTR_CONNECTION_DEAD,
   (SQLPOINTER)&closed, SQL_IS_INTEGER, 0 );
```

通过 SQLGetConnectAttr 函数获取 SQL_ATTR_CONNECTION_DEAD 属性时，如果连接已经删除（即使从连接删除时起未向服务器发出任何请求），则会返回值 SQL_CD_TRUE。无需向服务器发出请求即可确定连接是否已经删除，并且可在几秒钟内检测到删除的连接。有多个原因可删除连接，例如空闲超时。

有关详细信息（包括连接属性列表），请参见 Microsoft *ODBC 程序员参考*中的 SQLGetConnectAttr。

ODBC 应用程序中的线程和连接

可开发用于 SQL Anywhere 的多线程 ODBC 应用程序。建议您对每个线程使用不同的连接。

可对多个线程使用一个连接。但是，数据库服务器不允许一个连接同时有多个处于活动状态的请求。如果一个线程执行一个耗时较长的语句，则所有其它线程都必须等待该请求完成。

SQL Anywhere 连接属性

SQL Anywhere ODBC 驱动程序支持某些扩展的连接属性。

- **SA_REGISTER_MESSAGE_CALLBACK** 可使用 SQL MESSAGE 语句将消息从服务器发送到客户端应用程序。可创建消息处理程序例程来拦截这些消息。消息处理程序的回调原型如下：

  ```
  void SQL_CALLBACK message_handler(
  SQLHDBC sqlany_dbc,
  unsigned char  msg_type,
  long code,
  unsigned short  length,
  char * message
  );
  ```

  msg_type 的以下可能值在 *sqldef.h* 中定义。

  - **MESSAGE_TYPE_INFO** 消息类型为 INFO。
  - **MESSAGE_TYPE_WARNING** 消息类型为 WARNING。
  - **MESSAGE_TYPE_ACTION** 消息类型为 ACTION。
  - **MESSAGE_TYPE_STATUS** 消息类型为 STATUS。

  与消息关联的 SQLCODE 可以在 *code* 中提供。如果不可用，则 *code* 参数值为 0。

  消息的长度包含在 *length* 中。

  消息的指针包含在 *message* 中。请注意，*message* 不是以空值终止的。必须设计您的应用程序以处理这一问题。以下是一个示例。

  ```
  memcpy( mybuff, msg, len );
  mybuff[ len ] = '\0';
  ```

  要在 ODBC 中注册消息处理程序，请按如下方式调用 SQLSetConnectAttr 函数：

  ```
  rc = SQLSetConnectAttr(
     hdbc,
     SA_REGISTER_MESSAGE_CALLBACK,
     (SQLPOINTER) &message_handler, SQL_IS_POINTER );
  ```

  要在 ODBC 中注销消息处理程序，请按如下方式调用 SQLSetConnectAttr 函数：

  ```
  rc = SQLSetConnectAttr(
     hdbc,
     SA_REGISTER_MESSAGE_CALLBACK,
     NULL, SQL_IS_POINTER );
  ```

- **SA_GET_MESSAGE_CALLBACK_PARM** 要检索将传递给消息处理程序回调例程的 SQLHDBC 连接句柄的值，请将 SQLGetConnectAttr 与 SA_GET_MESSAGE_CALLBACK_PARM 参数配合使用。

  ```
  SQLHDBC callback_hdbc = NULL;
  rc = SQLGetConnectAttr(
     hdbc,
     SA_GET_MESSAGE_CALLBACK_PARM,
     (SQLPOINTER) &callback_hdbc, 0, 0 );
  ```


返回值将与传递给消息处理程序回调例程的参数值相同。

- **SA_REGISTER_VALIDATE_FILE_TRANSFER_CALLBACK** 用于注册文件传输校验回调函数。在允许进行任何传输前，ODBC 驱动程序会调用校验回调函数（如果存在）。如果在执行间接语句（从存储过程内部）期间，请求进行客户端数据传输，则除非客户端应用程序注册了校验回调函数，否则 ODBC 驱动程序将不允许进行传输。下面更详尽地介绍了进行校验调用的条件。

  回调原型如下：

  ```
  int SQL_CALLBACK file_transfer_callback(
  void * sqlca,
  char * file_name,
  int is_write
  );
  ```

  file_name 参数是要读取或写入的文件的名称。如果请求读取（从客户端传输到服务器），则 *is_write* 参数为 0，如果请求写入，则该参数为非零值。如果不允许进行文件传输，则回调函数应返回 0，否则返回非零值。

  为确保数据安全，服务器会跟踪请求文件传输的语句的源。服务器会确定语句是否是从客户端应用程序直接接收的。从客户端启动数据传输时，服务器会将语句源的相关信息发送到客户端软件。对于嵌入式 SQL 客户端库而言，仅当是由于执行客户端应用程序直接发送的语句而请求数据传输时，它才会允许无条件传输数据。否则，应用程序必须注册上文所述的校验回调函数，如果未注册该函数，则传输会被拒绝，而且语句失败并出现一个错误。请注意，如果客户端语句调用的某个存储过程在数据库中已经存在，则对该存储过程本身的执行不被视为是客户端启动的语句所完成的。但是，如果客户端应用程序显式地创建一个临时存储过程，则服务器会将对该存储过程的执行视为是由客户端启动的。同样，如果客户端应用程序执行一个批处理语句，则该批处理语句的执行被视为是直接由客户端应用程序完成的。

- **SA_SQL_ATTR_TXN_ISOLATION** 该属性用于设置扩展的事务隔离级别。以下示例设置一个快照隔离级别：

  ```
  SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
  SQLSetConnectAttr( dbc, SQL_ATTR_TXN_ISOLATION,
        SA_SQL_TXN_SNAPSHOT, SQL_IS_UINTEGER );
  ```

  有关详细信息，请参见“选择 ODBC 事务隔离级别”一节第 472 页。

执行 SQL 语句

ODBC 包括几个用于执行 SQL 语句的函数：

- **直接执行** SQL Anywhere 将分析 SQL 语句，拟定访问计划，然后执行该语句。分析和访问计划的拟定称为**准备**语句。
- **预准备执行** 语句的准备与执行是分开进行的。对于重复执行的语句来说，这样做可避免重复准备，从而提高性能。

  有关详细信息，请参见“执行预准备语句”一节第 463 页。

直接执行语句

SQLExecDirect 函数准备并执行 SQL 语句。该语句可以包括参数。

以下代码段说明如何执行不带参数的语句。SQLExecDirect 函数带有一个语句句柄、一个 SQL 字符串和一个长度或终止指示符（在本例中是一个以空值终止的字符串指示符）。

本节中描述的过程比较简单，但不够灵活。应用程序无法采用来自用户的任何输入来修改语句。有关更灵活的语句构造方法，请参见“执行包含绑定参数的语句”一节第 462 页。

♦ 在 ODBC 应用程序中执行 SQL 语句

1. 使用 **SQLAllocHandle** 为该语句分配一个句柄。

   例如，以下语句在具有句柄 dbc 的连接上分配了一个类型为 SQL_HANDLE_STMT、名称为 stmt 的句柄：

   ```
   SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
   ```

2. 调用 SQLExecDirect 函数执行语句：

   例如，以下几行代码声明并执行一个语句。deletestmt 的声明通常出现在函数的开头：

   ```
   SQLCHAR deletestmt[ STMT_LEN ] =
     "DELETE FROM Departments WHERE DepartmentID = 201";
   SQLExecDirect( stmt, deletestmt, SQL_NTS) ;
   ```

有关错误检查的完整示例，请参见 *samples-dir\SQLAnywhere\ODBCExecute\odbcexecute.cpp*。

有关 SQLExecDirect 的详细信息，请参见 Microsoft *ODBC 程序员参考*中的 SQLExecDirect。

执行包含绑定参数的语句

本节介绍如何使用绑定参数为语句参数设置运行时的值，来构造和执行 SQL 语句。

♦ 在 ODBC 应用程序中执行包含绑定参数的 SQL 语句

1. 使用 SQLAllocHandle 为该语句分配一个句柄。

例如，以下语句在具有句柄 dbc 的连接上分配了一个类型为 SQL_HANDLE_STMT、名称为 stmt 的句柄：

```
SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
```

2. 使用 SQLBindParameter 为语句捆绑参数。

   例如，以下几行代码声明一些变量，这些变量用于存储部门 ID、部门名称和管理人员 ID 以及语句字符串的值。然后，这些代码将参数绑定到使用 stmt 语句句柄执行的语句的第一个、第二个和第三个参数上。

```
#defined DEPT_NAME_LEN 40
SQLLEN cbDeptID = 0,
   cbDeptName = SQL_NTS, cbManagerID = 0;
SQLCHAR deptName[ DEPT_NAME_LEN + 1 ];
SQLSMALLINT deptID, managerID;
SQLCHAR insertstmt[ STMT_LEN ] =
  "INSERT INTO Departments "
  "( DepartmentID, DepartmentName, DepartmentHeadID )"
  "VALUES (?, ?, ?)";
SQLBindParameter( stmt, 1, SQL_PARAM_INPUT,
    SQL_C_SSHORT, SQL_INTEGER, 0, 0,
    &deptID, 0, &cbDeptID);
SQLBindParameter( stmt, 2, SQL_PARAM_INPUT,
    SQL_C_CHAR, SQL_CHAR, DEPT_NAME_LEN, 0,
    deptName, 0,&cbDeptName);
SQLBindParameter( stmt, 3, SQL_PARAM_INPUT,
    SQL_C_SSHORT, SQL_INTEGER, 0, 0,
    &managerID, 0, &cbManagerID);
```

3. 为参数赋值。

   例如，下面几行代码为第 2 步代码段的参数赋值。

```
deptID = 201;
strcpy( (char * ) deptName, "Sales East" );
managerID = 902;
```

   通常，设置这些变量是为了响应用户操作。

4. 使用 SQLExecDirect 执行语句。

   例如，以下几行代码执行语句句柄 stmt 上的 insertstmt 中具有的语句字符串。

```
SQLExecDirect( stmt, insertstmt, SQL_NTS) ;
```

此外，还可以为准备语句使用绑定参数，从而为需要执行多次的语句带来更好的性能。有关详细信息，请参见“执行预准备语句”一节第 463 页。

上述代码段不包括错误检查。有关包括错误检查的完整示例，请参见 *samples-dir\SQLAnywhere\ODBCExecute\odbcexecute.cpp*。

有关 SQLExecDirect 的详细信息，请参见 Microsoft *ODBC 程序员参考*中的 SQLExecDirect。

执行预准备语句

预准备语句可提高重复使用的语句的性能。ODBC 为使用预准备语句提供了一整套函数。

有关预准备语句的简介，请参见“准备语句”一节第 26 页。

♦ 执行预准备 SQL 语句

1. 使用 SQLPrepare 准备语句。

   例如，下面的代码段说明如何准备 INSERT 语句：

```
SQLRETURN   retcode;
SQLHSTMT    stmt;
retcode = SQLPrepare( stmt,
            "INSERT INTO Departments
             ( DepartmentID, DepartmentName, DepartmentHeadID )
            VALUES (?, ?, ?,)",
          SQL_NTS);
```

   在此示例中：

   - **retcode** 包含返回代码，应测试这些代码来检查操作是成功还是失败。
   - **stmt** 提供语句句柄，供以后引用。
   - **?** 问号是表示语句参数的占位符。

2. 使用 SQLBindParameter 设置语句参数值。

   例如，以下函数调用设置 DepartmentID 变量的值：

```
SQLBindParameter( stmt,
                  1,
                  SQL_PARAM_INPUT,
                  SQL_C_SSHORT,
                  SQL_INTEGER,
                  0,
                  0,
                  &sDeptID,
                  0,
                  &cbDeptID);
```

   在此示例中：

   - **stmt** 是语句句柄。
   - **1** 指示此调用设置第一个占位符的值。
   - **SQL_PARAM_INPUT** 指示该参数是输入语句。
   - **SQL_C_SHORT** 指示该应用程序中正在使用的 C 数据类型。
   - **SQL_INTEGER** 指示数据库中正在使用的 SQL 数据类型。

     接下来的两个参数指示列的精度和小数位数：对于整数，这两个参数均为零。
   - **&sDeptID** 是指向参数值缓冲区的指针。
   - **0** 指示缓冲区的长度，以字节计。
   - **&cbDeptID** 是指向参数值长度缓冲区的指针。

3. 捆绑其它两个参数并给 sDeptId 赋值。
4. 执行该语句：


```
retcode = SQLExecute( stmt);
```

第 2 至 4 步可重复执行多次。

5. 删除语句。

   删除语句将释放与语句本身相关联的资源。使用 SQLFreeHandle 可以删除语句。

有关包括错误检查的完整示例，请参见 *samples-dir\SQLAnywhere\ODBCPrepare\odbcprepare.cpp*。

有关 SQLPrepare 的详细信息，请参见 Microsoft *ODBC 程序员参考*中的 SQLPrepare。

64 位 ODBC 注意事项

使用 SQLBindCol、SQLBindParameter 或 SQLGetData 这类的 ODBC 函数时，某些参数的类型在函数原型中被设置为 SQLLEN 或 SQLULEN。视您所查看的 Microsoft《ODBC API 参考》文档的日期而定，您可能会看到被描述为 SQLINTEGER 或 SQLUINTEGER 的相同参数。

SQLLEN 和 SQLULEN 数据项在 64 位 ODBC 应用程序中为 64 位，在 32 位 ODBC 应用程序中为 32 位。SQLINTEGER 和 SQLUINTEGER 数据项在所有平台上均为 32 位。

为说明该问题，从旧版 Microsoft ODBC API 参考中摘选了以下 ODBC 函数原型。

```
SQLRETURN SQLGetData(
     SQLHSTMT     StatementHandle,
     SQLUSMALLINT ColumnNumber,
     SQLSMALLINT  TargetType,
     SQLPOINTER   TargetValuePtr,
     SQLINTEGER   BufferLength,
     SQLINTEGER  *StrLen_or_IndPtr);
```

将该函数原型与在 Microsoft Visual Studio 版本 8 的 *sql.h* 中找到的实际函数原型相比较。

```
SQLRETURN  SQL_API SQLGetData(
    SQLHSTMT      StatementHandle,
    SQLUSMALLINT  ColumnNumber,
    SQLSMALLINT   TargetType,
    SQLPOINTER    TargetValue,
    SQLLEN        BufferLength,
    SQLLEN       *StrLen_or_Ind);
```

正如您所见，BufferLength 和 StrLen_or_Ind 参数的类型现在被设置为 SQLLEN，而不是 SQLINTEGER。对于 64 位平台，它们是 64 位，而不是 Microsoft 文档中所述的 32 位。

为避免出现跨平台编译问题，SQL Anywhere 提供了自己的 ODBC 头文件。对于 Windows 平台，您应包括 *ntodbc.h* 头文件。对于 Unix 平台（如 Linux），您应包括 *unixodbc.h* 头文件。使用这些头文件可确保与目标平台上的相应 SQL Anywhere ODBC 驱动程序兼容。

下表列出了一些在 64 位和 32 位平台上具有相同或不同存储大小的常见 ODBC 类型。

ODBC API	64 位平台	32 位平台
SQLINTEGER	32 位	32 位
SQLUINTEGER	32 位	32 位
SQLLEN	64 位	32 位
SQLULEN	64 位	32 位
SQLSETPOSIROW	64 位	16 位
SQL_C_BOOKMARK	64 位	32 位
BOOKMARK	64 位	32 位

如果对数据变量和参数的声明不正确，则您可能会遇到不正确的软件行为。

下表汇总了引入 64 位支持平台以来发生更改的 ODBC API 函数原型。已对受影响的参数进行注释。Microsoft 文档中的参数名与函数原型中使用的实际参数名不同时，前者将显示在括号中。参数名是指 Microsoft Visual Studio 版本 8 的头文件中使用的参数名。

ODBC API	参数 (文档中的参数名)
SQLBindCol	SQLLEN BufferLength SQLLEN *Strlen_or_Ind
SQLBindParam	SQLULEN LengthPrecision SQLLEN *Strlen_or_Ind
SQLBindParameter	SQLULEN cbColDef (ColumnSize) SQLLEN cbValueMax (BufferLength) SQLLEN *pcbValue (Strlen_or_IndPtr)
SQLColAttribute	SQLLEN *NumericAttribute
SQLColAttributes	SQLLEN *pfDesc
SQLDescribeCol	SQLULEN *ColumnSize (ColumnSizePtr)
SQLDescribeParam	SQLULEN *pcbParamDef (ParameterSizePtr)
SQLExtendedFetch	SQLLEN irow (FetchOffset) SQLULEN *pcrow (RowCountPtr)
SQLFetchScroll	SQLLEN FetchOffset
SQLGetData	SQLLEN BufferLength SQLLEN *Strlen_or_Ind (Strlen_or_IndPtr)
SQLGetDescRec	SQLLEN *Length (LengthPtr)
SQLParamOptions	SQLULEN crow SQLULEN *pirow
SQLPutData	SQLLEN Strlen_or_Ind
SQLRowCount	SQLLEN *RowCount (RowCountPtr)
SQLSetConnectOption	SQLULEN Value

ODBC API	参数 (文档中的参数名)
SQLSetDescRec	SQLLEN Length SQLLEN *StringLength (StringLengthPtr) SQLLEN *Indicator (IndicatorPtr)
SQLSetParam	SQLULEN LengthPrecision SQLLEN *Strlen_or_Ind (Strlen_or_IndPtr)
SQLSetPos	SQLSETPOSIROW irow (RowNumber)
SQLSetScrollOptions	SQLLEN crowKeyset
SQLSetStmtOption	SQLULEN Value

某些通过指针传递到 ODBC API 并从其返回的值已经更改，以符合 64 位应用程序的需要。例如，以下的 SQLSetStmtAttr 和 SQLSetDescField 函数值不再是 SQLINTEGER/SQLUINTEGER。此规则同样适用于 SQLGetStmtAttr 和 SQLGetDescField 函数的相应参数。

ODBC API	Value/ValuePtr 变量的类型
SQLSetStmtAttr(SQL_ATTR_FETCH_BOOKMARK_PTR)	SQLLEN * 值
SQLSetStmtAttr(SQL_ATTR_KEYSET_SIZE)	SQLULEN 值
SQLSetStmtAttr(SQL_ATTR_MAX_LENGTH)	SQLULEN 值
SQLSetStmtAttr(SQL_ATTR_MAX_ROWS)	SQLULEN 值
SQLSetStmtAttr(SQL_ATTR_PARAM_BIND_OFFSET_PTR)	SQLULEN * 值
SQLSetStmtAttr(SQL_ATTR_PARAMS_PROCESSED_PTR)	SQLULEN * 值
SQLSetStmtAttr(SQL_ATTR_PARAMSET_SIZE)	SQLULEN 值
SQLSetStmtAttr(SQL_ATTR_ROW_ARRAY_SIZE)	SQLULEN 值
SQLSetStmtAttr(SQL_ATTR_ROW_BIND_OFFSET_PTR)	SQLULEN * 值
SQLSetStmtAttr(SQL_ATTR_ROW_NUMBER)	SQLULEN 值
SQLSetStmtAttr(SQL_ATTR_ROWS_FETCHED_PTR)	SQLULEN * 值
SQLSetDescField(SQL_DESC_ARRAY_SIZE)	SQLULEN 值
SQLSetDescField(SQL_DESC_BIND_OFFSET_PTR)	SQLLEN * 值

ODBC API	Value/ValuePtr 变量的类型
SQLSetDescField(SQL_DESC_ROWS_PROCESSED_PTR)	SQLULEN * 值
SQLSetDescField(SQL_DESC_DISPLAY_SIZE)	SQLLEN 值
SQLSetDescField(SQL_DESC_INDICATOR_PTR)	SQLLEN * 值
SQLSetDescField(SQL_DESC_LENGTH)	SQLLEN 值
SQLSetDescField(SQL_DESC_OCTET_LENGTH)	SQLLEN 值
SQLSetDescField(SQL_DESC_OCTET_LENGTH_PTR)	SQLLEN * 值

有关详细信息，请参见 Microsoft 文章 MDAC 2.7 中的 ODBC 64 位 API 更改。

数据对齐要求

使用 SQLBindCol、SQLBindParameter 或 SQLGetData 时，将为列或参数指定 C 数据类型。在某些平台上，提供给每列的存储（内存）必须正确对齐，才能读取或存储指定类型的值。下表列出了诸如 Sun Sparc、Itanium-IA64 和基于 ARM 的设备之类的处理器的内存对齐要求。

C 数据类型	需要的对齐方式
SQL_C_CHAR	无
SQL_C_BINARY	none
SQL_C_GUID	none
SQL_C_BIT	none
SQL_C_STINYINT	none
SQL_C_UTINYINT	none
SQL_C_TINYINT	none
SQL_C_NUMERIC	none
SQL_C_DEFAULT	none
SQL_C_SSHORT	2
SQL_C_USHORT	2
SQL_C_SHORT	2
SQL_C_DATE	2
SQL_C_TIME	2
SQL_C_TIMESTAMP	2
SQL_C_TYPE_DATE	2
SQL_C_TYPE_TIME	2
SQL_C_TYPE_TIMESTAMP	2
SQL_C_WCHAR	2（在所有平台上，缓冲区大小必须是 2 的倍数）
SQL_C_SLONG	4
SQL_C_ULONG	4

C 数据类型	需要的对齐方式
SQL_C_LONG	4
SQL_C_FLOAT	4
SQL_C_DOUBLE	8
SQL_C_SBIGINT	8
SQL_C_UBIGINT	8

x86、x64 和 PowerPC 平台不需要内存对齐。x64 平台包括 Advanced Micro Devices (AMD) AMD64 处理器和 Intel Extended Memory 64 Technology (EM64T) 处理器。

使用结果集

ODBC 应用程序使用游标来操纵和更新结果集。SQL Anywhere 为不同种类的游标和游标操作提供广泛的支持。

有关游标的介绍，请参见“使用游标”一节第 31 页。

选择 ODBC 事务隔离级别

可使用 SQLSetConnectAttr 为连接设置事务隔离级别。确定 SQL Anywhere 提供的事务隔离级别的特性包括以下几种：

- **SQL_TXN_READ_UNCOMMITTED** 将隔离级别设置为 0。在设置此特性值后，它会隔离任何从其他用户所做的更改读取的数据，且其他用户所做的更改将不可见。其他用户重新执行读取语句时将会受到影响。它不支持可重复的读取。这是隔离级别的缺省值。
- **SQL_TXN_READ_COMMITTED** 将隔离级别设置为 1。在设置此特性值后，它不会隔离任何从其他用户所做的更改读取的数据，且其他用户所做的更改将可见。其他用户重新执行读取语句时将会受到影响。它不支持可重复的读取。
- **SQL_TXN_REPEATABLE_READ** 将隔离级别设置为 2。在设置此特性值后，它会隔离任何从其他用户所做的更改读取的数据，且其他用户所做的更改将不可见。其他用户重新执行读取语句时将会受到影响。它支持可重复的读取。
- **SQL_TXN_SERIALIZABLE** 将隔离级别设置为 3。在设置此特性值后，它会隔离任何从其他用户所做的更改读取的数据，且其他用户所做的更改将不可见。其他用户重新执行读取语句时将不会受到影响。它支持可重复的读取。
- **SA_SQL_TXN_SNAPSHOT** 将隔离级别设置为快照。设置此特性后，它将为整个事务提供数据库的单个视图。
- **SA_SQL_TXN_STATEMENT_SNAPSHOT** 将隔离级别设置为语句快照。设置此特性后，它所提供的一致性要比快照隔离差，但是在由于长时间运行事务而导致临时文件中版本存储所用的空间过大的情况下，它将非常有用。
- **SA_SQL_TXN_READONLY_STATEMENT_SNAPSHOT** 将隔离级别设置为只读语句快照。设置此特性后，它所提供的一致性要比语句快照隔离差，但可避免出现更新冲突的可能性。因此，它最适用于移植那些最初打算在不同隔离级别下运行的应用程序。

有关详细信息，请参见 Microsoft *ODBC 程序员参考*中的 SQLSetConnectAttr。

示例

以下片段使用了快照隔离级别：

```
SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
SQLSetConnectAttr( dbc, SQL_ATTR_TXN_ISOLATION,
      SA_SQL_TXN_SNAPSHOT, SQL_IS_UINTEGER );
```


选择 ODBC 游标特性

执行语句和操纵结果集的 ODBC 函数使用游标执行它们的任务。应用程序每次执行 SQLExecute 或 SQLExecDirect 函数时，都会隐式打开游标。

如果游标在结果集中只正向移动而不更新结果集，对于这种应用情况，游标行为相对比较简单。缺省情况下，ODBC 应用程序会请求此行为。ODBC 定义一个只读的只进游标，SQL Anywhere 针对这种情况提供了已优化性能的游标。

有关只进游标的简单示例，请参见“检索数据”一节第 473 页。

如果游标需要在结果集中向前和向后双向滚动（例如在很多图形用户界面应用程序中），对于这种应用情况，游标行为将会更加复杂。当应用程序返回到一个由某种别的应用程序更新的行时，它会怎样？ODBC 定义了多种**可滚动游标**，从而使您可以内置适合您的应用程序的行为。SQL Anywhere 提供了一整套游标，可以满足各种 ODBC 可滚动游标类型的要求。

通过调用定义语句属性的 SQLSetStmtAttr 函数，您可以设置所需的 ODBC 游标特性。您必须先调用 SQLSetStmtAttr，然后才能执行创建结果集的语句。

您可以使用 SQLSetStmtAttr 来设置很多游标特性。确定 SQL Anywhere 提供的游标类型的特性包括以下几种：

- **SQL_ATTR_CURSOR_SCROLLABLE** 对于可滚动游标，设置为 SQL_SCROLLABLE；对于只进游标，设置为 SQL_NONSCROLLABLE。SQL_NONSCROLLABLE 是缺省设置。
- **SQL_ATTR_CONCURRENCY** 设置为下列值之一：
  - **SQL_CONCUR_READ_ONLY** 不允许更新。SQL_CONCUR_READ_ONLY 是缺省设置。
  - **SQL_CONCUR_LOCK** 使用足以确保该行能够得到更新的最低锁定级别。
  - **SQL_CONCUR_ROWVER** 使用优化的并发控制，比较数据行的版本（例如，SQLBase ROWID 或 Sybase TIMESTAMP）。
  - **SQL_CONCUR_VALUES** 使用优化的并发控制，比较值。

有关详细信息，请参见 Microsoft *ODBC 程序员参考*中的 SQLSetStmtAttr。

示例

下面的代码段请求一个只读的、可滚动游标：

```
SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
SQLSetStmtAttr( stmt, SQL_ATTR_CURSOR_SCROLLABLE,
      SQL_SCROLLABLE, SQL_IS_UINTEGER );
```

检索数据

若要从数据库检索行，可使用 SQLExecute 或 SQLExecDirect 执行 SELECT 语句。这将为该语句打开一个游标。

然后，使用 SQLFetch 或 SQLFetchScroll 通过游标读取行。这些函数从结果集读取下一个数据行集，并为所有绑定列返回数据。使用 SQLFetchScroll，可在绝对位置或相对位置，或者通过书签来指定行集。SQLFetchScroll 替换 ODBC 2.0 规范中旧的 SQLExtendedFetch。

应用程序使用 SQLFreeHandle 释放语句后，它将关闭游标。

若要从游标读取值，您的应用程序可以使用 SQLBindCol 或 SQLGetData。如果使用 SQLBindCol，则会在每次读取时自动检索值。如果使用 SQLGetData，您必须在每次读取后为每一列调用它。

SQLGetData 用来逐段读取 LONG VARCHAR 或 LONG BINARY 之类的列的值。另一种方法，还可以将 SQL_ATTR_MAX_LENGTH 语句属性设置为一个很大的值，使之能够包含列的整个值。SQL_ATTR_MAX_LENGTH 的缺省值为 256 KB。

SQL Anywhere ODBC 驱动程序实现 SQL_ATTR_MAX_LENGTH 的方式与 ODBC 规范的目标方式不同。SQL_ATTR_MAX_LENGTH 的用意是作为截断较大读取的机制来使用。它可以 "预览" 模式实现，其中仅显示数据的第一部分。例如：不是从服务器向客户端应用程序传输 4 MB blob，而是只传输前 500 字节（通过设置 SQL_ATTR_MAX_LENGTH 为 500）。SQL Anywhere ODBC 驱动程序不支持此实现。

以下代码段在查询时打开游标，并通过游标检索数据。为了使示例更容易读，这里省略了错误检查。该代码段节选自一个完整示例，可在 *samples-dir\SQLAnywhere\ODBCSelect\odbcselect.cpp* 中找到。

```
SQLINTEGER cbDeptID = 0, cbDeptName = SQL_NTS, cbManagerID = 0;
SQLCHAR deptName[ DEPT_NAME_LEN + 1 ];
SQLSMALLINT deptID, managerID;
SQLHENV  env;
SQLHDBC  dbc;
SQLHSTMT stmt;
SQLRETURN retcode;
SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env );
SQLSetEnvAttr( env,
            SQL_ATTR_ODBC_VERSION,
            (void*)SQL_OV_ODBC3, 0);
SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
SQLConnect( dbc,
            (SQLCHAR*) "SQL Anywhere 11 Demo", SQL_NTS,
            (SQLCHAR*) "DBA", SQL_NTS,
            (SQLCHAR*) "sql", SQL_NTS );
SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
SQLBindCol( stmt, 1,
            SQL_C_SSHORT, &deptID, 0, &cbDeptID);
SQLBindCol( stmt, 2,
            SQL_C_CHAR, deptName,
            sizeof(deptName), &cbDeptName);
SQLBindCol( stmt, 3,
            SQL_C_SSHORT, &managerID, 0, &cbManagerID);
SQLExecDirect( stmt, (SQLCHAR * )
"SELECT DepartmentID, DepartmentName, DepartmentHeadID FROM Departments "
                "ORDER BY DepartmentID", SQL_NTS );
while( ( retcode = SQLFetch( stmt ) ) != SQL_NO_DATA ){
   printf( "%d %20s %d\n", deptID, deptName, managerID );
}
SQLFreeHandle( SQL_HANDLE_STMT, stmt );
SQLDisconnect( dbc );
SQLFreeHandle( SQL_HANDLE_DBC, dbc );
SQLFreeHandle( SQL_HANDLE_ENV, env );
```


在游标中可读取的行位置编号受整数的大小制约。您最多可以读取到第 2147483646 行，这个数字比可以在 32 位整数中保存的值小 1。在使用负数（从末尾开始计算行数）时，您最多可以读取到的行数比整数中可保存的最大负值大 1。

通过游标更新和删除行

Microsoft 的《ODBC 程序员参考》建议您使用 SELECT ...FOR UPDATE 来指示查询可以使用定位操作进行更新。您不必在 SQL Anywhere 中使用 FOR UPDATE 子句：只要满足以下条件，即可自动更新 SELECT 语句：

- 基础的查询支持更新。

  就是说，只要结果中各列内的数据修改语句有意义，定位的数据修改语句就可在游标上执行。

  ansi_update_constraints 数据库选项将查询的类型限制为可更新的类型。

  有关详细信息，请参见“ansi_update_constraints 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》。

- 游标类型支持更新。

  如果您使用的是只读游标，则不能更新结果集。

ODBC 提供两种方法来执行定位的更新和删除：

- 使用 SQLSetPos 函数。

  根据提供的参数（SQL_POSITION、SQL_REFRESH、SQL_UPDATE 和 SQL_DELETE），SQLSetPos 设置游标位置，让应用程序刷新、更新或删除结果集中的数据。

  这是 SQL Anywhere 中使用的方法。

- 使用 SQLExecute 发送定位的 UPDATE 和 DELETE 语句。这种方法不得用于 SQL Anywhere 中。

使用书签

ODBC 提供**书签**，书签是用于游标中的行的值。对于对值敏感的游标和不敏感游标，SQL Anywhere 支持书签。例如，这意味着：ODBC 游标类型 SQL_CURSOR_STATIC 和 SQL_CURSOR_KEYSET_DRIVEN 支持书签，而游标类型 SQL_CURSOR_DYNAMIC 和 SQL_CURSOR_FORWARD_ONLY 不支持书签。

在 ODBC 3.0 之前，数据库只能指定它是否支持书签：没有为每个游标类型提供此信息的接口。数据库服务器没有办法指示支持的游标书签类型。对于 ODBC 2 应用程序，SQL Anywhere 会返回信息说明它的确支持书签。这样，虽然您尝试将书签用于动态游标时不会遇到什么阻碍，但是您不应当这样组合使用。

调用存储过程

本节介绍如何创建和调用存储过程以及如何处理来自 ODBC 应用程序的结果。

有关存储过程和触发器的完整说明，请参见“使用过程、触发器和批处理”《SQL Anywhere 服务器 - SQL 的用法》。

过程和结果集

有两种类型的过程：返回结果集的过程和不返回结果集的过程。您可以使用 SQLNumResultCols 来指出差异：如果过程不返回结果集，则结果列数为零。如果有结果集，则可像任何其它游标一样使用 SQLFetch 或 SQLExtendedFetch 来读取值。

过程的参数应当使用参数标记（问号）来传递。使用 SQLBindParameter 可为每个参数标记指派存储区域，无论是 INPUT、OUTPUT 还是 INOUT 参数都可以。

要处理多个结果集，ODBC 必须描述当前正在执行的游标，而不是描述由过程定义的结果集。因此，ODBC 并不总是按照存储过程定义中的 RESULT 子句中的定义来描述列名称。要避免出现此问题，您可以在过程结果集游标中使用列别名。

示例

此示例创建和调用不返回结果集的过程。该过程采用一个 INOUT 参数，并会增加它的值。在此示例中，变量 num_col 的值为零，这是因为该过程不返回结果集。为了使示例更容易读，这里省略了错误检查。

```
HDBC dbc;
HSTMT stmt;
long I;
SWORD num_col;

/* Create a procedure */
SQLAllocStmt( dbc, &stmt );
SQLExecDirect( stmt,
      "CREATE PROCEDURE Increment( INOUT a INT )" \
      " BEGIN" \
         " SET a = a + 1" \
      " END", SQL_NTS );

/* Call the procedure to increment 'I' */
I = 1;
SQLBindParameter( stmt, 1, SQL_C_LONG, SQL_INTEGER, 0,
            0, &I, NULL );
SQLExecDirect( stmt, "CALL Increment( ? )",
            SQL_NTS );
SQLNumResultCols( stmt, &num_col );
do_something( I );
```

示例

此示例调用一个返回结果集的过程。在此示例中，变量 **num_col** 的值为 2，因为该过程返回的结果集具有两列。同样，错误检查已被省略，这样该示例便更容易阅读。

```
HDBC dbc;
HSTMT stmt;
SWORD num_col;
RETCODE retcode;
char ID[ 10 ];
```



```
char Surname[ 20 ];

/* Create the procedure */
SQLExecDirect( stmt,
      "CREATE PROCEDURE employees()" \
      " RESULT( ID CHAR(10), Surname CHAR(20))"\
      " BEGIN" \
      " SELECT EmployeeID, Surname FROM Employees" \
      " END", SQL_NTS );

/* Call the procedure - print the results */
SQLExecDirect( stmt, "CALL employees()", SQL_NTS );
SQLNumResultCols( stmt, &num_col );
SQLBindCol( stmt, 1, SQL_C_CHAR, &ID,
            sizeof(ID), NULL );
SQLBindCol( stmt, 2, SQL_C_CHAR, &Surname,
            sizeof(Surname), NULL );

for( ;; ) {
   retcode = SQLFetch( stmt );
   if( retcode == SQL_NO_DATA_FOUND ) {
      retcode = SQLMoreResults( stmt );
   if( retcode == SQL_NO_DATA_FOUND ) break;
}  else {
      do_something( ID, Surname );
   }
}
```

处理错误

ODBC 中的错误是使用来自每个 ODBC 函数调用的返回值和 SQLError 函数或 SQLGetDiagRec 函数的返回值进行报告的。SQLError 函数用于 ODBC 版本 3 之前的版本（但不包括版本 3）。自版本 3 起，已不建议使用 SQLError 函数，此函数已被 SQLGetDiagRec 函数取代。

每个 ODBC 函数都返回一个 SQLRETURN，它是以下状态代码之一：

状态代码	说明
SQL_SUCCESS	无错误。
SQL_SUCCESS_WITH_INFO	该函数完成，但是对 SQLError 的调用将显示警告。 这种状态最常见的情况是：返回的值太长，应用程序提供的缓冲区不够用。
SQL_ERROR	函数未完成，因为出现了错误。调用 SQLError 可获取有关此问题的详细信息。
SQL_INVALID_HANDLE	作为参数传递的环境、连接或语句句柄无效。 如果在释放句柄后再使用该句柄，或者句柄为空值指针，则通常会发生这种情况。
SQL_NO_DATA_FOUND	没有可用信息。 使用这种状态的最常见情况是在从游标进行读取时，这种状态表示游标中没有更多行。
SQL_NEED_DATA	参数需要数据。 这是一项高级特性，SQLParamData 和 SQLPutData 下面的 ODBC SDK 文档对其作了说明。

每个环境、连接和语句句柄都可能有与之相关联的一个或多个错误或警告。每个对 SQLError 或 SQLGetDiagRec 的调用都会返回有关一个错误的信息，然后删除有关该错误的信息。如果您不调用 SQLError 或 SQLGetDiagRec 来删除所有错误，则会在执行下一个将同一句柄作为参数来传递的函数调用时，删除这些错误。

每个对 SQLError 的调用都会传递三个句柄，这些句柄分别用于环境、连接和语句。第一个调用使用 SQL_NULL_HSTMT 获取与连接相关联的错误。同样，使用 SQL_NULL_DBC 和 SQL_NULL_HSTMT 的调用将获取任何与环境句柄相关联的错误。

每个对 SQLGetDiagRec 的调用将传递环境、连接或语句句柄。第一个调用传递句柄类型 SQL_HANDLE_DBC 以获取与连接关联的错误。第二个调用传递句柄类型 SQL_HANDLE_STMT，以获取与刚执行的语句相关联的错误。

如果有要报告的错误（*不是*SQL_ERROR），SQLError 和 SQLGetDiagRec 将返回 SQL_SUCCESS；如果没有其它要报告的错误，则将返回 SQL_NO_DATA_FOUND。

示例 1

以下代码段使用了 SQLError，并返回代码：

```
/* Declare required variables */
SQLHDBC dbc;
SQLHSTMT stmt;
SQLRETURN retcode;
UCHAR errmsg[100];
/* Code omitted here */
retcode = SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt );
if( retcode == SQL_ERROR ){
   SQLError( env, dbc, SQL_NULL_HSTMT, NULL, NULL,
         errmsg, sizeof(errmsg), NULL );
   /* Assume that print_error is defined */
   print_error( "Allocation failed", errmsg );
   return;
}

/* Delete items for order 2015 */
retcode = SQLExecDirect( stmt,
       "DELETE FROM SalesOrderItems WHERE ID=2015",
       SQL_NTS );
if( retcode == SQL_ERROR ) {
   SQLError( env, dbc, stmt, NULL, NULL,
            errmsg, sizeof(errmsg), NULL );
   /* Assume that print_error is defined */
   print_error( "Failed to delete items", errmsg );
   return;
}
```

示例 2

以下代码段使用了 SQLGetDiagRec，并返回代码：

```
/* Declare required variables */
SQLHDBC dbc;
SQLHSTMT stmt;
SQLRETURN retcode;
SQLSMALLINT errmsglen;
SQLINTEGER errnative;
UCHAR errmsg[255];
UCHAR errstate[5];
/* Code omitted here */
retcode = SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt );
if( retcode == SQL_ERROR ){
   SQLGetDiagRec(SQL_HANDLE_DBC, dbc, 1, errstate,
       &errnative, errmsg, sizeof(errmsg), &errmsglen);
    /* Assume that print_error is defined */
   print_error( "Allocation failed",
errstate, errnative, errmsg );
   return;

}
/* Delete items for order 2015 */
retcode = SQLExecDirect( stmt,
       "DELETE FROM SalesOrderItems WHERE ID=2015",
       SQL_NTS );
if( retcode == SQL_ERROR ) {
   SQLGetDiagRec(SQL_HANDLE_STMT, stmt,
       recnum, errstate,
       &errnative, errmsg, sizeof(errmsg), &errmsglen);
   /* Assume that print_error is defined */
```

```
        print_error("Failed to delete items",
            errstate, errnative, errmsg );
        return;
    }
```


SQL Anywhere JDBC 驱动程序

目录

JDBC 简介

您既可从客户端应用程序使用 JDBC，也可从数据库内部使用 JDBC。在数据库中加入编程逻辑时，使用 JDBC 的 Java 类也可以起到 SQL 存储过程的作用，而且功能比 SQL 存储过程更强。

JDBC 为 Java 应用程序提供了 SQL 接口：如果您要从 Java 访问关系数据，则可利用 JDBC 调用进行此访问。

短语**客户端应用程序**指在用户计算机上运行的应用程序和在中层应用程序服务器上运行的逻辑。

这些示例说明在 SQL Anywhere 中使用 JDBC 时的一些独特功能。有关 JDBC 编程的详细信息，请参见讲解 JDBC 编程的书籍。

可通过以下方式配合使用 JDBC 和 SQL Anywhere：

- **客户端的 JDBC** Java 客户端应用程序可对 SQL Anywhere 发出 JDBC 调用。通过 JDBC 驱动程序建立连接。

  SQL Anywhere 支持并包括两个 JDBC 驱动程序：iAnywhere JDBC 驱动程序和用于纯 Java 应用程序的 jConnect 驱动程序，其中前者是类型 2 JDBC 驱动程序，后者是类型 4 JDBC 驱动程序。
- **数据库中的 JDBC** 数据库中安装的 Java 类可使用内部 JDBC 驱动程序进行 JDBC 调用，以此来访问和修改数据库中的数据。

JDBC 资源

- **示例源代码** 本章示例的源代码位于 *samples-dir\SQLAnywhere\JDBC* 目录下。
- **JDBC 规范** 您可以在 Java SE Technologies - Database 中找到有关 JDBC Data Access API 的更多内容。
- **必需的软件** 使用 jConnect 驱动程序需要 TCP/IP。

  可以在 jConnect for JDBC 获得 jConnect 驱动程序。

  有关 jConnect 驱动程序及其位置的详细信息，请参见“使用 jConnect JDBC 驱动程序”一节 第 487 页。

选择 JDBC 驱动程序

SQL Anywhere 支持以下 JDBC 驱动程序：

- **iAnywhere JDBC 驱动程序** 此驱动程序使用命令序列客户端/服务器协议与 SQL Anywhere 进行通信。它的行为与 ODBC、嵌入式 SQL 和 OLE DB 应用程序是一致的。连接到 SQL Anywhere 数据库时，建议使用 iAnywhere JDBC 驱动程序。
- **jConnect** 此驱动程序是 100% 纯 Java 驱动程序。它使用 TDS 客户端/服务器协议与 SQL Anywhere 进行通信。

  可以从 jConnect for JDBC 获得 jConnect 及 jConnect 的文档。

在选择使用哪个驱动程序时，应当考虑下列因素：

- **特性** iAnywhere JDBC 驱动程序和 jConnect 6.0.5 都与 JDBC 3.0 兼容。但连接到 SQL Anywhere 数据库时，iAnywhere JDBC 驱动程序提供可完全滚动的游标。而仅当连接到 Adaptive Server Enterprise 数据库时，jConnect JDBC 驱动程序才提供可完全滚动的游标。

  可在 JDBC Downloads 获得 JDBC 3.0 API 文档。有关 iAnywhere 支持的 JDBC API 方法的汇总，请参见“iAnywhere JDBC 3.0 API 支持”一节第 508 页。

- **纯 Java** jConnect 驱动程序是纯 Java 解决方案。iAnywhere JDBC 驱动程序需要 SQL Anywhere ODBC 驱动程序，它不是纯 Java 解决方案。
- **性能** 在多数情况下，iAnywhere JDBC 驱动程序所提供的性能要比 jConnect 驱动程序好一些。
- **兼容性** jConnect 驱动程序使用的 TDS 协议可与 Adaptive Server Enterprise 共享。该驱动程序行为的某些方面受此协议的控制，并被配置为与 Adaptive Server Enterprise 兼容。

有关 iAnywhere JDBC 驱动程序和 jConnect 的平台可用性的信息，请参见 http://www.sybase.com/detail?id=1062623。

有关配合使用 jConnect 和 Windows Mobile 的信息，请参见“在 Windows Mobile 上使用 jConnect”一节 《SQL Anywhere 服务器 - 数据库管理》。

JDBC 程序结构

JDBC 应用程序中通常会发生以下事件序列：

1. **创建连接对象**

   调用 DriverManager 类的 getConnection 类方法即可创建 Connection 对象，并与数据库建立连接。

2. **生成语句对象**

   Connection 对象会生成 Statement 对象。

3. **传递 SQL 语句**

   将在数据库环境内部执行的 SQL 语句传递到 Statement 对象。如果该语句是一个查询，则此操作会返回 ResultSet 对象。

   ResultSet 对象包含 SQL 语句返回的数据，但一次只显示一行（与游标的工作方式类似）。

4. **遍历结果集的行**

   ResultSet 对象的 next 方法可执行两种操作：

   - 当前行（通过 ResultSet 对象显示的结果集内的行）前进一行。
   - 返回布尔值，指示要前进到的目标行是否存在。

5. **为每行检索值**

   通过识别 ResultSet 对象中每一列的名称或位置的方式来检索每一列的值。可以使用 getData 方法来获取当前行中某列的值。

Java 对象可使用 JDBC 对象与数据库进行交互，并获取数据以供自身使用。

客户端与服务器端 JDBC 连接的区别

客户端 JDBC 与数据库服务器中的 JDBC 之间的区别是它们与数据库环境建立连接的方式不同。

- **客户端** 在客户端 JDBC 中，建立连接需要 iAnywhere JDBC 驱动程序或 jConnect JDBC 驱动程序。将参数传递给 DriverManager.getConnection 就建立了连接。从客户端应用程序的角度看，数据库环境是外部应用程序。
- **服务器端** 在数据库服务器中使用 JDBC 时，连接已经存在。字符串 "jdbc:default:connection" 会传递给 DriverManager.getConnection，这可使 JDBC 应用程序在当前用户连接中工作。这是一个快速、有效而安全的操作，因为客户端应用程序已通过了数据库安全检查建立了连接。用户 ID 和口令一经提供，便不需要再次提供。服务器端 JDBC 驱动程序只能连接到当前连接的数据库。

您可以编写 JDBC 类，以便使用一个条件语句构造 URL，让 JDBC 类既能在客户端运行，又能在服务器端运行。外部连接需要主机名和端口号，而内部连接需要 "jdbc:default:connection"。

使用 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。

有关选择使用哪个 JDBC 驱动程序的信息，请参见“选择 JDBC 驱动程序”一节第 482 页。

装载 iAnywhere JDBC 驱动程序

在您的应用程序中使用 iAnywhere JDBC 驱动程序之前，您必须装载该相应的驱动程序。使用以下语句装载 iAnywhere JDBC 3.0 驱动程序：

```
DriverManager.registerDriver( (Driver)
    Class.forName(
    "ianywhere.ml.jdbcodbc.jdbc3.IDriver").newInstance()
    );
```

使用 newInstance 方法能解决某些浏览器中的问题。

- 由于类是使用 Class.forName 装载的，所以不必使用 import 语句导入包含 iAnywhere JDBC 驱动程序的程序包。
- 在运行应用程序时，*jodbc.jar* 必须位于类路径中。

```
set classpath=%classpath%;install-dir\java\jodbc.jar
```

所需文件

iAnywhere JDBC 驱动程序的 Java 组件包括在 *jodbc.jar* 文件中，该文件安装在 SQL Anywhere 安装目录的 *Java* 子目录中。对于 Windows，本机组件是 *dbjodbc11.dll*，它位于 SQL Anywhere 安装目录的 *bin32* 或 *bin64* 子目录中；对于 Unix，本机组件是 *libdbjodbc11.so*。该组件必须位于系统路径中。当部署使用此驱动程序的应用程序时，必须同时部署 ODBC 驱动程序文件。

将 URL 提供给驱动程序

要通过 iAnywhere JDBC 驱动程序连接到某个数据库，您需要提供该数据库的 URL。例如：

```
Connection con = DriverManager.getConnection(
      "jdbc:ianywhere:DSN=SQL Anywhere 11 Demo" );
```

该 URL 包含 **jdbc:ianywhere:**，后跟一个标准的 ODBC 连接字符串。该连接字符串通常是 ODBC 数据源，但除了该数据源以外，您还可以加上各个明确的连接参数（由分号分隔），或者用它们来代替该数据源。有关可在连接字符串中使用的参数的详细信息，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

如果不使用数据源，则应指定要使用的 ODBC 驱动程序，方法是在连接字符串中包括 DRIVER 参数：

```
Connection con = DriverManager.getConnection(
      "jdbc:ianywhere:driver=SQL Anywhere 11;..." );
```

使用 jConnect JDBC 驱动程序

SQL Anywhere 支持一种版本的 jConnect：jConnect 6.0.5。可在 用于 jConnect for JDBC 单独下载 jConnect 驱动程序。jConnect 文档也位于该同一页面上。

如果要在小程序中使用 JDBC，则必须使用 jConnect JDBC 驱动程序连接到 SQL Anywhere 数据库。

jConnect 驱动程序文件

SQL Anywhere 支持以下版本的 jConnect：

- **jConnect 6.0.5** 此版本的 jConnect 用于开发 JDK 1.4 或更高版本的应用程序。jConnect 6.0.5 与 JDBC 3.0 兼容。jConnect 6.0.5 是作为名为 *jconn3.jar* 的 JAR 文件提供的。

> **注意**
> 就本文档而言，提供的所有说明和代码示例均假定您是在开发 JDK 1.5 应用程序，同时在使用 jConnect 6.0.5 驱动程序。
>
> *install-dir\java* 文件夹中有 *jconn3.jar* 的一个副本。但是，推荐您使用随 jConnect 6.0.5 提供的文件的版本，因为该版本与您已安装的 jConnect 版本是最新的。

为 jConnect 设置类路径

要想让应用程序使用 jConnect，则在编译和运行时，jConnect 类必须位于类路径中，以便 Java 编译器和 Java 运行时能够找到必要的文件。

以下命令用于将 jConnect 6.0.5 驱动程序添加到现有的 CLASSPATH 环境变量中，其中 *path* 是 jConnect 安装目录。

```
set classpath=path\jConnect-6_0\classes\jconn3.jar;%classpath%
```

导入 jConnect 类

jConnect 6.0.5 中的类全部位于 com.sybase.jdbc3.jdbc 中。您必须在每个源文件的开头导入这些类：

```
import com.sybase.jdbc3.jdbc.*
```

加密口令

SQL Anywhere 支持用于 jConnect 连接的口令加密。

在数据库中安装 jConnect 系统对象

如果要用 jConnect 访问系统表信息（数据库元数据），必须将 jConnect 系统对象添加到数据库中。

您可以在创建数据库时将 jConnect 系统对象添加到数据库，也可以在以后升级数据库时再添加。可以从 Sybase Central 或使用 dbupgrad 实用程序升级数据库。

Windows Mobile
不要将 jConnect 系统对象添加到 Windows Mobile 数据库中。请参见“在 Windows Mobile 上使用 jConnect”一节 《SQL Anywhere 服务器 - 数据库管理》。

小心
在升级前，应始终对该数据库文件进行备份。如果在升级现有文件时升级失败，则这些文件将无法使用。有关备份数据库的信息，请参见“备份和数据恢复” 《SQL Anywhere 服务器 - 数据库管理》。

♦ **向数据库添加 jConnect 系统对象 (Sybase Central)**

1. 以 DBA 用户身份从 Sybase Central 连接到数据库。
2. 选择数据库（必要时），然后从 [**工具**] 菜单选择 SQL Anywhere 11，接着选择 [**升级数据库**]。
3. 请按照 [**升级数据库向导**] 中的说明进行操作。

♦ **向数据库添加 jConnect 系统对象 (dbupgrad)**

- 在命令提示符下，运行以下命令：

  ```
  dbupgrad -c "connection-string"
  ```

  在此命令中，*connection-string* 是以 DBA 用户身份来启用对数据库和服务器的访问的适当连接字符串。

装载 jConnect 驱动程序

在应用程序中使用 jConnect 之前，请先使用以下语句来装载驱动程序：

```
DriverManager.registerDriver( (Driver)
    Class.forName(
    "com.sybase.jdbc3.jdbc.SybDriver").newInstance()
    );
```

使用 newInstance 方法能解决某些浏览器中的问题。

- 由于类是使用 Class.forName 装载的，所以不必使用 import 语句导入包含 jConnect 驱动程序的程序包。
- 要使用 jConnect 6.0.5，在运行应用程序时 *jconn3.jar* 必须位于类路径中。*jconn3.jar* 位于 jConnect 6.0.5 安装目录的 *classes* 子目录中（通常为 *jConnect-6_0\classes*）。

将 URL 提供给驱动程序

要通过 jConnect 连接到某个数据库，您需要提供该数据库的 URL。例如：

```
Connection con = DriverManager.getConnection(
      "jdbc:sybase:Tds:localhost:2638", "DBA", "sql");
```


URL 按照以下方式组成：

```
jdbc:sybase:Tds:host:port
```

各组成部分如下：

- **jdbc:sybase:Tds** 使用 TDS 应用程序协议的 jConnect JDBC 驱动程序。
- **host** 运行服务器的计算机的 IP 地址或名称。如果要建立同一主机连接，则可使用 localhost，它表示您登录的计算机系统。
- **port** 数据库服务器监听时使用的端口号。分配给 SQL Anywhere 的端口号是 2638。如果没有特殊原因，请使用该端口号。

连接字符串长度必须小于 253 个字符。

指定服务器上的数据库

每个 SQL Anywhere 数据库服务器一次都能装载一个或多个数据库。如果通过 jConnect 连接时提供的 URL 只指定了服务器，而没有指定数据库，则尝试连接的数据库是服务器上的缺省数据库。

您可以通过下列任一方式提供 URL 的扩展形式，由此而指定特定数据库。

使用 ServiceName 参数

```
jdbc:sybase:Tds:host:port?ServiceName=database
```

使用问号，后面跟随一系列赋值，这是向 URL 提供参数的标准方式。ServiceName 是否大写并不重要，但 = 号两边不能有空格。*database* 参数是数据库名，而不是服务器名。数据库名不得包括路径或文件后缀。例如：

```
Connection con = DriverManager.getConnection(
      "jdbc:sybase:Tds:localhost:2638?ServiceName=demo", "DBA", "sql");
```

使用 RemotePWD 参数

存在一个用于将其它连接参数传递到服务器的解决方案。

这一技术允许您使用 RemotePWD 字段来提供其它连接参数，如数据库名或数据库文件。您可用 put 方法将 RemotePWD 设置成一个 [属性] 字段。

以下代码对如何使用该字段进行了说明。

```
import java.util.Properties;
.
.
.
DriverManager.registerDriver( (Driver)
    Class.forName(
    "com.sybase.jdbc3.jdbc.SybDriver").newInstance()
    );

Properties props = new Properties();
props.put( "User", "DBA" );
props.put( "Password", "sql" );
props.put( "RemotePWD", ",DatabaseFile=mydb.db" );
```

```
Connection con = DriverManager.getConnection(
    "jdbc:sybase:Tds:localhost:2638", props );
```

如示例中所示，在 DatabaseFile 连接参数前必须加上一个逗号。利用 DatabaseFile 参数，您可使用 jConnect 启动服务器上的数据库。缺省情况下，数据库启动时的设置为 autostop=YES。如果向 DatabaseFile (DBF) 或 DatabaseName (DBN) 连接参数指定 utility_db（例如，DBN=utility_db），则会自动启动实用程序数据库。

有关实用程序数据库的详细信息，请参见“使用实用程序数据库”一节 《SQL Anywhere 服务器 - 数据库管理》。

用于 jConnect 连接的数据库选项集

应用程序使用 jConnect 驱动程序连接数据库时，将调用 sp_tsql_environment 存储过程：sp_tsql_environment 过程将一些数据库选项设置为与 Adaptive Server Enterprise 行为兼容。

另请参见

- “Open Client 和 jConnect 连接的特性”一节 《SQL Anywhere 服务器 - 数据库管理》
- “sp_tsql_environment 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》

从 JDBC 客户端应用程序连接

使用 iAnywhere JDBC 驱动程序时，数据库元数据始终可用。

如果希望从使用 jConnect 的 JDBC 应用程序访问数据库系统表（数据库元数据），必须向数据库添加一组 jConnect 系统对象。这些过程会缺省安装到所有数据库。dbinit -i 选项会阻止此安装。

有关向数据库添加 jConnect 系统对象的详细信息，请参见“使用 jConnect JDBC 驱动程序”一节第 487 页。

以下完整的 Java 应用程序是一个命令行程序，它连接到正在运行的数据库，在命令行中输出一组信息，然后终止。

对于任何 JDBC 应用程序，要使用数据库数据，第一步都必须建立连接。

此示例中所示的外部连接是常规的客户端/服务器连接。有关如何从运行于数据库服务器内部的 Java 类创建内部连接的信息，请参见“从服务器端的 JDBC 类建立连接”一节第 493 页。

连接示例代码

以下是用来建立连接的方法的源代码。源代码可在 *samples-dir\SQLAnywhere\JDBC* 目录的 *JDBCConnect.java* 文件中找到。如所显示的那样，该示例使用 JDBC 3.0 版本的 iAnywhere JDBC 驱动程序连接到数据库。要使用 jConnect 6.0.5 驱动程序，请将 `ianywhere.ml.jdbcodbc.jdbc3.IDriver` 替换为 `com.sybase.jdbc3.jdbc.SybDriver`。如果要使用 jConnect 6.0.5 驱动程序，还必须更改连接字符串。替换代码以注释的形式包含在源代码中。

```
import java.io.*;
import java.sql.*;

public class JDBCConnect
{
  public static void main( String args[] )
  {
    try
    {
      // Select the JDBC driver. May throw a SQLException.
      // Choices are jConnect 6.0 driver
      // or iAnywhere JDBC 3.0 driver.
      // Currently, we use the iAnywhere JDBC 3.0 driver.
      DriverManager.registerDriver( (Driver)
        Class.forName(
        // "com.sybase.jdbc3.jdbc.SybDriver").newInstance()
        "ianywhere.ml.jdbcodbc.jdbc3.IDriver").newInstance()
        );

      // Create a connection. Choices are TDS using jConnect,
      // Sun's JDBC-ODBC bridge, or the iAnywhere JDBC driver.
      // Currently, we use the iAnywhere JDBC driver.
      Connection con = DriverManager.getConnection(
        // "jdbc:sybase:Tds:localhost:2638", "DBA", "sql");
        // "jdbc:odbc:driver=SQL Anywhere 11;uid=DBA;pwd=sql" );
        "jdbc:ianywhere:driver=SQL Anywhere 11;uid=DBA;pwd=sql" );

      // Create a statement object, the container for the SQL
      // statement. May throw a SQLException.
      Statement stmt = con.createStatement();
```

```
        // Create a result set object by executing the query.
        // May throw a SQLException.
        ResultSet rs = stmt.executeQuery(
          "SELECT ID, GivenName, Surname FROM Customers");

        // Process the result set.
        while (rs.next())
        {
          int value = rs.getInt(1);
          String FirstName = rs.getString(2);
          String LastName = rs.getString(3);
          System.out.println(value+" "+FirstName+" "+LastName);
        }
        rs.close();
        stmt.close();
        con.close();
      }
      catch (SQLException sqe)
      {
        System.out.println("Unexpected exception : " +
                   sqe.toString() + ", sqlstate = " +
                   sqe.getSQLState());
        System.exit(1);
      }
      catch (Exception e)
      {
        e.printStackTrace();
        System.exit(1);
      }

      System.exit(0);
    }
  }
```

连接示例如何工作

外部连接示例是一个 Java 命令行程序。

导入包

该应用程序需要两个程序包，这两个程序包将在 *JDBCConnect.java* 的前几行中导入：

- java.io 包中包含在输出到控制台窗口时所需的 Sun Microsystems io 类。
- java.sql 包中包含所有 JDBC 应用程序都需要的 Sun Microsystems JDBC 类。

main 方法

每个 Java 应用程序都需要一个包含 main 方法的类，该方法是在程序启动时调用的方法。在以上的简单示例中，JDBCConnect.main 是应用程序中唯一的公共方法。

JDBCConnect.main 方法执行以下任务：

1. 装载 JDBC 3.0 驱动程序（由驱动程序字符串 "ianywhere.ml.jdbcodbc.jdbc3.IDriver" 标识）。

   Class.forName 用于装载驱动程序。使用 newInstance 方法能解决某些浏览器中的问题。

2. 使用 iAnywhere JDBC 驱动程序 URL 与缺省的运行数据库建立连接。如果要改为使用 jConnect 驱动程序，则使用 URL "jdbc:sybase:Tds:localhost:2638"（如注释中所示），其中 "DBA" 和 "sql" 分别作为用户 ID 和口令。

   DriverManager.getConnection 使用指定的 URL 建立连接。
3. 创建包含 SQL 语句的语句对象。
4. 通过执行 SQL 查询创建结果集对象。
5. 迭代通过结果集，输出列信息。
6. 结束每个结果集、语句和连接对象。

运行连接示例

♦ **创建并执行外部连接示例应用程序**

1. 在命令提示符下，更改为 *samples-dir\SQLAnywhere\JDBC* 目录。
2. 利用以下命令，使用本地计算机上的示例数据库启动数据库服务器：

   ```
   dbeng11 samples-dir\demo.db
   ```
3. 设置 CLASSPATH 环境变量。

   ```
   set classpath=%classpath%;install-dir\java\jodbc.jar
   ```

   如果要改为使用 jConnect 驱动程序，则使用以下命令（其中 *path* 是 jConnect 的安装目录）：

   ```
   set classpath=path\jConnect-6_0\classes\jconn3.jar;%classpath%
   ```
4. 运行以下命令编译示例：

   ```
   javac JDBCConnect.java
   ```
5. 运行以下命令来运行示例：

   ```
   java JDBCConnect
   ```
6. 确认命令提示符下出现一列带有客户名称的标识号。

   如果连接尝试失败，则显示的是错误消息。请确认是否已执行了所需的全部步骤。请检查类路径是否正确。不正确的设置可能会导致无法找到类。

从服务器端的 JDBC 类建立连接

JDBC 中的 SQL 语句是使用 Connection 对象的 createStatement 方法构建的。即使是在服务器内部运行的类，也需要建立连接以创建 Connection 对象。

从服务器端的 JDBC 类建立连接比建立外部连接更为直接。由于用户已经连接到数据库，所以类只使用当前连接。

服务器端连接的示例代码

以下是服务器端连接示例的源代码。它是 *samples-dir\SQLAnywhere\JDBC\JDBCConnect.java* 中源代码的修改后版本。

```
import java.io.*;
import java.sql.*;

public class JDBCConnect2
{
  public static void main( String args[] )
  {
    try
    {
      // Open the connection. May throw a SQLException.
      Connection con = DriverManager.getConnection(
          "jdbc:default:connection" );

      // Create a statement object, the container for the SQL
      // statement. May throw a SQLException.
      Statement stmt = con.createStatement();
      // Create a result set object by executing the query.
      // May throw a SQLException.
      ResultSet rs = stmt.executeQuery(
        "SELECT ID, GivenName, Surname FROM Customers");

      // Process the result set.
      while (rs.next())
      {
        int value = rs.getInt(1);
        String FirstName = rs.getString(2);
        String LastName = rs.getString(3);
        System.out.println(value+" "+FirstName+" "+LastName);
      }
      rs.close();
      stmt.close();
      con.close();
    }
    catch (SQLException sqe)
    {
      System.out.println("Unexpected exception : " +
                sqe.toString() + ", sqlstate = " +
                sqe.getSQLState());
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

服务器端连接示例的不同之处

服务器端连接示例与客户端连接示例几乎是相同的，但有以下几个例外：

1. 不需要装载驱动程序管理器。已将以下代码从示例中删除。

```
DriverManager.registerDriver( (Driver)
  Class.forName(
  // "com.sybase.jdbc3.jdbc.SybDriver").newInstance()
```



```
    "ianywhere.ml.jdbcodbc.IDriver").newInstance()
    );
```

2. 它使用当前连接与缺省的运行数据库建立连接。已将 getConnection 调用中的 URL 更改为：

```
Connection con = DriverManager.getConnection(
          "jdbc:default:connection" );
```

3. `System.exit()` 语句已删除。

运行服务器端连接示例

♦ **创建并执行内部连接示例应用程序**

1. 在命令提示符下，更改为 *samples-dir\SQLAnywhere\JDBC* 目录。
2. 利用以下命令，使用本地计算机上的示例数据库启动数据库服务器：

```
dbeng11 samples-dir\demo.db
```

3. 对于服务器端 JDBC，不必设置 CLASSPATH 环境变量。
4. 输入以下命令编译示例：

```
javac JDBCConnect2.java
```

5. 使用 Interactive SQL 将类安装到示例数据库。运行以下语句：

```
INSTALL JAVA NEW
FROM FILE 'samples-dir\SQLAnywhere\JDBC\JDBCConnect2.class'
```

   您也可使用 Sybase Central 安装该类。与示例数据库保持连接，打开 [**外部环境**] 下的 [**Java**] 子文件夹，然后选择 [**文件**] » [**新建**] » [**Java 类**]。然后按向导中的说明进行操作。

6. 定义一个名为 JDBCConnect 的存储过程，该存储过程将充当类中 JDBCConnect2.main 方法的包装：

```
CREATE PROCEDURE JDBCConnect()
  EXTERNAL NAME 'JDBCConnect2.main([Ljava/lang/String;)V'
  LANGUAGE JAVA;
```

7. 如下所示调用 JDBCConnect2.main 方法：

```
call JDBCConnect();
```

   在会话中首次调用某个 Java 类时，必须装载 Java 虚拟机。这可能需要几秒钟的时间。

8. 确认在数据库服务器消息窗口中出现一列带有客户名称的标识号。

   如果连接尝试失败，则显示的是错误消息。请确认是否已执行了所需的全部步骤。

有关 JDBC 连接的几点说明

- **自动提交行为** JDBC 规范要求在每个数据修改语句后缺省执行 COMMIT。当前，客户端 JDBC 行为是提交（自动提交为 true），服务器端行为是不提交（自动提交为 false）。要在客户端和服务器端应用程序中均获得相同的行为，您可以使用诸如以下的语句：

  ```
  con.setAutoCommit( false );
  ```

  在此语句中，con 是当前连接对象。您也可以将自动提交设置为 true。

- **连接缺省值** 从服务器端的 JDBC 对 `getConnection( "jdbc:default:connection" )` 的调用中，只有第一个调用使用缺省值创建新连接。后续调用返回当前连接的包装，所有连接属性均保持不变。如果在初始连接中将自动提交设置为 false，则同一 Java 代码中，任何后续 getConnection 调用所返回的连接中的自动提交均会设置为 false。

  您可能希望确保关闭连接时会使连接属性恢复为缺省值，这样，所获得的后续连接就会采用标准的 JDBC 值。以下代码可实现这一点：

  ```
  Connection con =
      DriverManager.getConnection("jdbc:default:connection");

  boolean oldAutoCommit = con.getAutoCommit();
  try {
       // main body of code here
  }
  finally {
      con.setAutoCommit( oldAutoCommit );
  }
  ```

  此处的讨论不仅适用于自动提交，也适用于其它连接属性，如事务隔离级别和只读模式。

  有关 getTransactionIsolation、setTransactionIsolation 和 isReadOnly 方法的详细信息，请参见关于 java.sql.Connection 接口的文档。

使用 JDBC 访问数据

相对于传统的 SQL 存储过程，用于保存数据库中的部分或全部类的 Java 应用程序具有更大的优点。不过，在入门阶段，最好并行使用 SQL 存储过程来证实 JDBC 的功能，这样或许会有所帮助。在下面的示例中，您编写了向 Departments 表中插入行的 Java 类。

与其它接口一样，JDBC 中的 SQL 语句可以为**静态**，也可以为**动态**。静态 SQL 语句在 Java 应用程序中构造，然后会发送到数据库。数据库服务器会分析该语句，选择执行计划，然后执行该语句。语法分析与选择执行计划统称为**准备**语句。

如果某条类似的语句必须被执行多次（例如，向一个表中插入多次），使用静态 SQL 会带来很大的开销，因为每次都必须执行准备步骤。

与此相反，动态 SQL 语句包含占位符。只需用这些占位符准备一次语句，就可以多次执行语句，省去了额外的准备开销。动态 SQL 将在“使用预准备语句进行更有效的访问”一节第 499 页中进行讨论。

准备示例

示例代码

本节中的代码段取自 *samples-dir\SQLAnywhere\JDBC\JDBCExample.java* 中的完整类。

- **安装 JDBCExample 类**

1. 编译 *JDBCExample.java* 源代码。
2. 使用 Interactive SQL，以 DBA 身份连接到示例数据库。
3. 通过在 Interactive SQL 中执行以下语句将 *JDBCExample.class* 文件安装到示例数据库中（*samples-dir* 表示 SQL Anywhere 示例目录）：

```
INSTALL JAVA NEW
FROM FILE 'samples-dir\SQLAnywhere\JDBC\JDBCExample.class'
```

您也可使用 Sybase Central 安装该类。与示例数据库保持连接，打开 [**外部环境**] 下的 [**Java**] 子文件夹，然后选择 [**文件**] » [**新建**] » [**Java 类**]。请按照向导中的说明进行操作。

使用 JDBC 执行插入、更新和删除

Statement 对象执行静态 SQL 语句。您可使用 Statement 对象的 executeUpdate 方法执行 INSERT、UPDATE 和 DELETE 等 SQL 语句，这些语句不返回结果集。诸如 CREATE TABLE 的语句及其它数据定义语句也可通过使用 executeUpdate 来执行。

使用 iAnywhere JDBC 驱动程序执行成批插入时，建议使用较小的列大小。建议不要使用成批插入方式向 long binary 或 long varchar 列中插入大的二进制或字符数据，这样可能会导致性能下降。这是因为 iAnywhere JDBC 驱动程序只有分配较大的内存大小才能保存成批插入的各行。在所有其它情况下，与逐个插入相比，成批插入会提供更佳的性能。

如果不希望您的应用程序使用成批插入方式向 long binary 或 long varchar 列中插入大数据，则任何成批插入列的缺省最大字段大小为 256K。如果您的应用程序需要成批插入的列数据超过 256K，则必须先在 Statement.setMaxFieldSize() 方法中指定更大的最大字段大小，然后再执行成批插入。

以下代码段说明如何执行 INSERT 语句。它将已传递到 InsertStatic 方法的 Statement 对象用作一个参数。

```
public static void InsertStatic( Statement stmt )
{
  try
  {
    int iRows = stmt.executeUpdate(
      "INSERT INTO Departments (DepartmentID, DepartmentName)"
      + " VALUES (201, 'Eastern Sales')" );
    // Print the number of rows inserted
    System.out.println(iRows + " rows inserted");
  }
  catch (SQLException sqe)
  {
    System.out.println("Unexpected exception : " +
              sqe.toString() + ", sqlstate = " +
              sqe.getSQLState());
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}
```

可用源代码
此代码段是位于 *samples-dir\SQLAnywhere\JDBC* 目录中的 JDBCExample 类的一部分。

注意

- executeUpdate 方法返回一个用于反映操作所影响到的行数的整数。在此情况下，成功的 INSERT 会返回值 1。
- 作为服务器端的类运行时，`System.out.println` 的输出会显示在数据库服务器消息窗口中。

♦ 运行 JDBC 的 Insert 示例：

1. 使用 Interactive SQL，以 DBA 身份连接到示例数据库。
2. 确保已安装 JDBCExample 类。

   有关安装 Java 示例类的详细信息，请参见“准备示例”一节第 497 页。
3. 定义一个名为 JDBCExample 的存储过程，该存储过程充当类中 JDBCExample.main 方法的包装：

   ```
   CREATE PROCEDURE JDBCExample( IN arg CHAR(50) )
     EXTERNAL NAME 'JDBCExample.main([Ljava/lang/String;)V'
     LANGUAGE JAVA;
   ```
4. 如下所示调用 JDBCExample.main 方法：

   ```
   CALL JDBCExample( 'insert' );
   ```


参数字符串 'insert' 用以调用 InsertStatic 方法。

5. 确认 Departments 表中已添加一行。

```
SELECT * FROM Departments;
```

示例程序会在数据库服务器消息窗口中显示 Departments 表的更新内容。

6. 在示例类中有一个称为 DeleteStatic 的类似方法，用于显示如何删除刚刚添加的行。如下所示调用 JDBCExample.main 方法：

```
CALL JDBCExample( 'delete' );
```

参数字符串 'delete' 用以调用 DeleteStatic 方法。

7. 确认该行已从 Departments 表中删除。

```
SELECT * FROM Departments;
```

示例程序会在数据库服务器消息窗口中显示 Departments 表的更新内容。

使用预准备语句进行更有效的访问

如果使用 Statement 接口，则先对您发送到数据库的每条语句进行分析，生成访问计划，然后再执行该语句。这些在实际执行语句之前的步骤被称为**准备**语句。

如果使用 PreparedStatement 接口，将会在性能方面获得一些益处。这样您就可以使用占位符来准备语句，然后在执行语句时向占位符赋值。

使用预准备语句在执行多个类似的操作（如插入多行）时特别有用。

有关准备语句的详细信息，请参见“准备语句”一节第 26 页。

示例

以下示例说明如何使用 PreparedStatement 接口，不过，插入单行并未有效地利用了预准备语句。

JDBCExample 类的以下 InsertDynamic 方法用于执行一个预准备语句：

```
public static void InsertDynamic( Connection con,
                String ID, String name )
{
  try {
    // Build the INSERT statement
    // ? is a placeholder character
    String sqlStr = "INSERT INTO Departments " +
            "( DepartmentID, DepartmentName ) " +
            "VALUES ( ? , ? )";

    // Prepare the statement
    PreparedStatement stmt =
      con.prepareStatement( sqlStr );

    // Set some values
    int idValue = Integer.valueOf( ID );
    stmt.setInt( 1, idValue );
    stmt.setString( 2, name );
```

```
        // Execute the statement
        int iRows = stmt.executeUpdate();

        // Print the number of rows inserted
        System.out.println(iRows + " rows inserted");
      }
      catch (SQLException sqe)
      {
        System.out.println("Unexpected exception : " +
                   sqe.toString() + ", sqlstate = " +
                   sqe.getSQLState());
      }
      catch (Exception e)
      {
        e.printStackTrace();
      }
    }
```

可用源代码
此代码段是位于 *samples-dir\SQLAnywhere\JDBC* 目录中的 JDBCExample 类的一部分。

注意

- executeUpdate 方法返回一个用于反映操作所影响到的行数的整数。在此情况下，成功的 INSERT 会返回值 1。
- 作为服务器端的类运行时，`System.out.println` 的输出会显示在数据库服务器消息窗口中。

♦ **运行 JDBC 的 Insert 示例：**

1. 使用 Interactive SQL，以 DBA 身份连接到示例数据库。
2. 确保已安装 JDBCExample 类。

   有关安装 Java 示例类的详细信息，请参见“准备示例”一节第 497 页。
3. 定义一个名为 JDBCInsert 的存储过程，该存储过程充当类中 JDBCExample.Insert 方法的包装：

   ```
   CREATE PROCEDURE JDBCInsert( IN arg1 INTEGER, IN arg2 CHAR(50) )
     EXTERNAL NAME 'JDBCExample.Insert(ILjava/lang/String;)V'
     LANGUAGE JAVA;
   ```
4. 如下所示调用 JDBCExample.Insert 方法：

   ```
   CALL JDBCInsert( 202, 'Southeastern Sales' );
   ```

   Insert 方法用以调用 InsertDynamic 方法。
5. 确认 Departments 表中已添加一行。

   ```
   SELECT * FROM Departments;
   ```

   示例程序会在数据库服务器消息窗口中显示 Departments 表的更新内容。
6. 在示例类中有一个称为 DeleteDynamic 的类似方法，用于显示如何删除刚刚添加的行。

   定义一个名为 JDBCDelete 的存储过程，该存储过程充当类中 JDBCExample.Delete 方法的包装：


```
CREATE PROCEDURE JDBCDelete( in arg1 integer )
  EXTERNAL NAME 'JDBCExample.Delete(I)V'
  LANGUAGE JAVA;
```

7. 如下所示调用 JDBCExample.Delete 方法：

```
CALL JDBCDelete( 202 );
```

   Delete 方法用以调用 DeleteDynamic 方法。

8. 确认该行已从 Departments 表中删除。

```
SELECT * FROM Departments;
```

   示例程序会在数据库服务器消息窗口中显示 Departments 表的更新内容。

使用预准备语句进行宽插入

PreparedStatement.addBatch() 方法对于执行成批插入（或大范围插入）十分有用。以下是使用此方法的一些指导。

1. 应使用 Connection.prepareStatement() 方法之一准备 INSERT 语句。

```
// Build the INSERT statement
String sqlStr = "INSERT INTO Departments " +
            "( DepartmentID, DepartmentName ) " +
            "VALUES ( ? , ? )";
// Prepare the statement
PreparedStatement stmt =
     con.prepareStatement( sqlStr );
```

2. 应设置准备的插入语句的参数并对其进行批处理，如下所示：

```
// loop to batch "n" sets of parameters
for( i=0; i < n; i++ )
{
    // Note "stmt" is the original prepared insert statement from step 1.
    stmt.setSomeType( 1, param_1 );
    stmt.setSomeType( 2, param_2 );
    .
    .
    .
    // Note that there are "m" parameters in the statement.
    stmt.setSomeType( m , param_m );

    // Add the set of parameters to the batch and
    // move to the next row of parameters.
    stmt.addBatch();
}
```

   示例：

```
for( i=0; i < 5; i++ )
{
    stmt.setInt( 1, idValue );
    stmt.setString( 2, name );
    stmt.addBatch();
}
```


3. 然后，便可使用 PreparedStatement.executeUpdate() 方法执行批处理。

应注意，系统仅支持 PreparedStatement.addBatch() 方法，且必须调用 PreparedStatement.executeUpdate() 方法才能执行批处理。不支持语句对象的任何批处理方法（即 Statement.addBatch()、Statement.clearBatch()、Statement.executeBatch()），因为这些方法完全是可选项，而且并非十分有用。对于此类静态批处理，最好是在单个字符串上调用 Statement.execute() 或 Statement.executeQuery() 方法（批处理语句被包装在 BEGIN...END 内）。

注意

- 批处理中不支持 BLOB 参数。
- 虽然支持字符串/二进制参数，但字符串/二进制参数的大小是一个问题。缺省情况下，字符串/二进制参数的大小被限制为 255 个字符或 510 个字节。限制的原因是缘于基础 ODBC 协议，这里不加以讨论。有关进一步的信息，最好是查看关于传递 ODBC 中的参数数组的文档。但是，如果应用程序需要在一个批内传递较大的字符串或二进制参数，则已提供了另一种方法 - setBatchStringSize，以增加字符串/二进制参数的大小。请注意，必须在首次 addBatch() 调用之前调用此方法。如果在首次 addBatch() 方法调用之后调用此方法，则将忽略新的大小设置。因此，调用此方法更改字符串/二进制参数的大小时，应用程序需要事先知道该参数的最大字符串或二进制值将是多少。

要使用 setBatchStringSize 方法，必须修改上述 "代码"，如下所示：

```
// You need to cast "stmt" to an IPreparedStatment object
// to change the size of string/binary parameters.
ianywhere.ml.jdbcodbc.IPreparedStatement _stmt =
    (ianywhere.ml.jdbcodbc.IPreparedStatement)stmt;

// Now, for example, change the size of string parameter 4
// from the default 255 characters to 300 characters.
// Note that string parameters are measured in "characters".
_stmt.setBatchStringSize( 4, 300 );

// Change the size of binary parameter 6
// from the default 510 bytes to 750 bytes.
// Note that binary parameters are measured in "bytes".
_stmt.setBatchStringSize( 6, 750 );

// loop to batch "n" sets of parameters
// where n should not be too large
for( i=0; i < n; i++ )
{
    // stmt is the prepared insert statement from step 1
    stmt.setSomeType( 1, param_1 );
    stmt.setSomeType( 2, param_2 );
    .
    .
    .
    // Note that there are "m" parameters in the statement.
    stmt.setSomeType( m , param_m );

    // Add the set of parameters to the batch and
    // move to the next row of parameters.
    stmt.addBatch();
}
```

修改参数的最大字符串/二进制大小时应非常小心。如果将最大值设置得过高，则额外的内存分配开销可能会抵销通过使用批处理所获得的任何性能。此外，应用程序还可能不会事先知道某个特定

参数的最大字符串或二进制值是多少。因此，建议不要更改某个参数的最大字符串或二进制大小，而是一直使用批处理方法，直到遇到大于当前/缺省的最大值的字符串或二进制大小。那时，应用程序会调用 executeBatch() 来执行当前批处理的参数，然后调用并执行常规设置和 executeUpdate() 方法，直到较大的字符串/二进制参数已处理，接着在遇到较小的字符串/二进制参数时切换回批处理模式。

返回结果集

本节介绍如何从 Java 方法获得一个或多个结果集。

您必须编写一个用于向调用环境返回一个或多个结果集的 Java 方法，并将此方法包装在 SQL 存储过程中。以下代码段说明如何才能向调用 SQL 代码返回多个结果集。它使用三个 executeQuery 语句获得三个不同的结果集。

```
public static void Results( ResultSet[] rset )
       throws SQLException
{
    // Demonstrate returning multiple result sets

    Connection con = DriverManager.getConnection(
                    "jdbc:default:connection" );
    rset[0] = con.createStatement().executeQuery(
        "SELECT * FROM Employees" +
        "   ORDER BY EmployeeID" );
    rset[1] = con.createStatement().executeQuery(
        "SELECT * FROM Departments" +
        "   ORDER BY DepartmentID" );
    rset[2] = con.createStatement().executeQuery(
        "SELECT i.ID,i.LineID,i.ProductID,i.Quantity," +
        "       s.OrderDate,i.ShipDate," +
        "       s.Region,e.GivenName||' '||e.Surname" +
        "   FROM SalesOrderItems AS i" +
        "   JOIN SalesOrders AS s" +
        "   JOIN Employees AS e" +
        "   WHERE s.ID=i.ID" +
        "         AND s.SalesRepresentative=e.EmployeeID" );
    con.close();
}
```

可用源代码
此代码段是位于 *samples-dir\SQLAnywhere\JDBC* 目录中的 JDBCExample 类的一部分。

注意

- 此服务器端 JDBC 示例使用当前连接（使用 getConnection）与缺省的运行数据库建立连接。
- executeQuery 方法返回结果集。

♦ 运行 JDBC 结果集示例

1. 使用 Interactive SQL，以 DBA 身份连接到示例数据库。
2. 确保已安装 JDBCExample 类。

   有关安装 Java 示例类的详细信息，请参见“准备示例”一节第 497 页。

3. 定义一个名为 JDBCResults 的存储过程，该存储过程充当类中 JDBCExample.Results 方法的包装：

```
CREATE PROCEDURE JDBCResults()
  DYNAMIC RESULT SETS 3
  EXTERNAL NAME 'JDBCExample.Results([Ljava/sql/ResultSet;)V'
  LANGUAGE JAVA;
```

4. 设置以下 Interactive SQL 选项，以便您可以查看查询的所有结果：
   a. 从 [**工具**] 菜单中选择 [**选项**]。
   b. 单击 [**SQL Anywhere**]。
   c. 单击 [**结果**] 选项卡。
   d. 将 [**要显示的最大行数**] 的值设置为 **5000**。
   e. 选择 [**显示所有结果集**]。
   f. 单击 [**确定**]。
5. 如下所示调用 JDBCExample.Results 方法：

```
CALL JDBCResults();
```

6. 分别检查以下这三个结果选项卡：[结果集 1]、[结果集 2] 和 [结果集 3]。

其它 JDBC 说明

- **访问权限** 如同数据库中的所有 Java 类一样，包含有 JDBC 语句的类可由任何用户来访问，只要 GRANT EXECUTE 语句已授予它们用以执行充当 Java 方法包装的存储过程的权限。
- **执行权限** 执行 Java 类时需要具有执行这些类的连接权限。此行为与存储过程的行为不同，执行存储过程需要具有所有者权限。

使用 JDBC 转义语法

您可以在任何 JDBC 应用程序（包括 Interactive SQL）中使用 JDBC 转义语法。此转义语法允许您调用存储过程而不管您正在使用哪种数据库管理系统。转义语法的一般格式为

{{ **keyword** *parameters* }}

在 Interactive SQL 中，大括号*必须*成对使用。连续的括号之间不得有空格："{{" 是允许的，但 "{ {" 是不允许的。同样，您不能在语句中使用新行字符。由于转义语法不是由 Interactive SQL 执行的，所以在存储过程中不能使用转义语法。

您可以使用转义语法访问由 JDBC 驱动程序实现的函数库，这些函数包括数字、字符串、时间、日期和系统函数。

例如，要以与数据库管理系统无关的方式获得当前用户的名称，您将执行以下语句：

```
SELECT {{ FN USER() }}
```

可用的函数取决于您正在使用的 JDBC 驱动程序。以下两个表分别列出了 iAnywhere JDBC 驱动程序和 jConnect 驱动程序支持的函数。

iAnywhere JDBC 驱动程序支持的函数

数字函数	字符串函数	系统函数	时间/日期函数
ABS	ASCII	IFNULL	CURDATE
ACOS	CHAR	USERNAME	CURTIME
ASIN	CONCAT		DAYNAME
ATAN	DIFFERENCE		DAYOFMONTH
ATAN2	INSERT		DAYOFWEEK
CEILING	LCASE		DAYOFYEAR
COS	LEFT		HOUR
COT	LENGTH		MINUTE
DEGREES	LOCATE		MONTH
EXP	LOCATE_2		MONTHNAME
FLOOR	LTRIM		NOW
LOG	REPEAT		QUARTER
LOG10	RIGHT		SECOND

数字函数	字符串函数	系统函数	时间/日期函数
MOD	RTRIM		WEEK
PI	SOUNDEX		YEAR
POWER	SPACE		
RADIANS	SUBSTRING		
RAND	UCASE		
ROUND			
SIGN			
SIN			
SQRT			
TAN			
TRUNCATE			

jConnect 支持的函数

数字函数	字符串函数	系统函数	时间/日期函数
ABS	ASCII	DATABASE	CURDATE
ACOS	CHAR	IFNULL	CURTIME
ASIN	CONCAT	USER	DAYNAME
ATAN	DIFFERENCE	CONVERT	DAYOFMONTH
ATAN2	LCASE		DAYOFWEEK
CEILING	LENGTH		HOUR
COS	REPEAT		MINUTE
COT	RIGHT		MONTH
DEGREES	SOUNDEX		MONTHNAME
EXP	SPACE		NOW
FLOOR	SUBSTRING		QUARTER

数字函数	字符串函数	系统函数	时间/日期函数
LOG	UCASE		SECOND
LOG10			TIMESTAMPADD
PI			TIMESTAMPDIFF
POWER			YEAR
RADIANS			
RAND			
ROUND			
SIGN			
SIN			
SQRT			
TAN			

使用转义语法的语句应该可以用于 SQL Anywhere、Adaptive Server Enterprise、Oracle、SQL Server 或连接到的其它数据库管理系统。

例如，要使用 SQL 转义语法通过 sa_db_info 过程获得数据库属性，您将在 Interactive SQL 中执行以下语句：

```
{{CALL sa_db_info( 0 ) }}
```

iAnywhere JDBC 3.0 API 支持

支持 JDBC 3.0 规范的所有必需的类和方法。不支持 java.sql.Blob 接口的一些可选方法。这些可选的方法有：

- long position(Blob pattern, long start);
- long position(byte[] pattern, long start);
- OutputStream setBinaryStream(long pos)
- int setBytes(long pos, byte[] bytes)
- int setBytes(long pos, byte[] bytes, int offset, int len);
- void truncate(long len);

SQL Anywhere 嵌入式 SQL

目录

嵌入式 SQL 简介

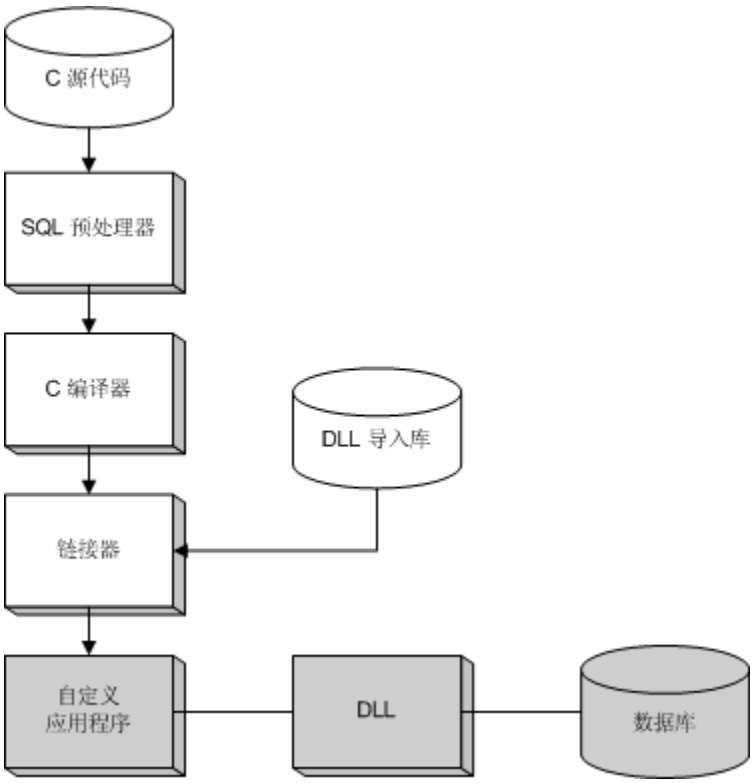
[嵌入式 SQL] 是用于 C 和 C++ 编程语言的数据库编程接口。它由混杂在（嵌入于） C 或 C++ 源代码中的 SQL 语句组成。这些 SQL 语句先由 **SQL 预处理器**转换为 C 或 C++ 源代码，然后您可以编译这些源代码。

在运行时，嵌入式 SQL 应用程序使用名为 DBLIB 的 SQL Anywhere **接口库**与数据库服务器通信。在大多数平台上，DBLIB 是一个动态链接库 (**DLL**) 或共享对象。

- 在 Windows 操作系统上，接口库是 *dblib11.dll*。
- 在 UNIX 操作系统上，接口库可以是 *libdblib11.so*、*libdblib11.sl* 或 *libdblib11.a*（视操作系统而定）。
- 在 Mac OS X 上，接口库是 *libdblib11.dylib.1*。

SQL Anywhere 提供了两种嵌入式 SQL。静态嵌入式 SQL 使用起来比较简单，但它不如动态嵌入式 SQL 灵活。

开发过程概述



在对程序成功地进行预处理和编译后，就可以将其与 DBLIB 的**导入库**链接在一起，形成可执行文件。在运行数据库服务器时，这个可执行文件使用 DBLIB 与数据库服务器交互。在对程序进行预处理时，不必运行数据库服务器。

对于 Windows，Microsoft Visual C++、Watcom C/C++ 和 Borland C++ 各自有单独的导入库。

在 Windows 中，通过使用导入库可以开发调用 DLL 中函数的应用程序。SQL Anywhere 还提供了另外一种方法来避免使用导入库（建议使用此方法）。有关详细信息，请参见“在 Windows 中动态装载 DBLIB”一节第 514 页。

运行 SQL 预处理器

SQL 预处理器是一个名为 *sqlpp* 的可执行文件。

SQLPP 命令行如下：

sqlpp [*options*] *sql-filename* [*output-filename*]

在运行 C 或 C++ 编译器之前，SQL 预处理器处理含有嵌入式 SQL 的 C 程序。预处理器将 SQL 语句转换为 C/C++ 语言源代码，并置于输出文件中。含有嵌入式 SQL 的源程序的扩展名通常为 *.sqc*。缺省的输出文件名为 *sql-filename*，扩展名为 *.c*。如果 *sql-filename* 已经具有 *.c* 扩展名，则缺省情况下输出文件的扩展名为 *.cc*。

重新处理嵌入式 SQL
如果重建应用程序以使用新的主版本数据库接口库，则必须使用相同版本的 SQL 预处理器对嵌入式 SQL 文件进行预处理。

有关命令行选项的完整列表，请参见“SQL 预处理器”一节第 563 页。

支持的编译器

C 语言 SQL 预处理器已经可以与下列编译器联合使用：

操作系统	编译器	版本
Windows	Watcom C/C++	9.5 或更高版本
Windows	Microsoft Visual C++	6.0 或更高版本
Windows	Borland C++	4.5
Windows Mobile	Microsoft Visual C++	2005
Windows Mobile	Microsoft eMbedded Visual C++	3.0, 4.0
Unix	GNU 编译器或本地编译器	

嵌入式 SQL 头文件

所有头文件都安装在 SQL Anywhere 安装目录的 *SDK\Include* 子目录中。

文件名	说明
sqlca.h	包括在所有嵌入式 SQL 程序中的主头文件。此文件包括 [SQL 通信区域] (SQLCA) 的结构定义和所有嵌入式 SQL 数据库接口函数的原型。
sqlda.h	包括在使用动态 SQL 的嵌入式 SQL 程序中的 [SQL 描述符区域] 结构定义。
sqldef.h	嵌入式 SQL 接口数据类型的定义。此文件还包含从 C 程序启动数据库服务器所需的结构定义和返回代码。
sqlerr.h	在 SQLCA 的 sqlcode 字段中返回的错误代码的定义。
sqlstate.h	在 SQLCA 的 sqlstate 字段中返回的 ANSI/ISO SQL 标准错误状态的定义。
pshpk1.h、*pshpk4.h*、*poppk.h*	这些头文件可确保正确地处理结构压缩。

导入库

在 Windows 平台上，所有导入库都安装在 SQL Anywhere 安装目录的 *SDK\Lib* 子目录下。Windows 导入库存储在 *SDK\Lib\x86* 和 *SDK\Lib\x64* 子目录中。Windows Mobile 导入库安装在 *SDK\Lib\CE\Arm.50* 子目录中。

在 Unix 平台上，所有导入库都安装在 SQL Anywhere 安装目录的 *lib32* 和 *lib64* 子目录下。

在 Mac OS X 平台上，所有导入库都安装在 SQL Anywhere 安装目录的 *System/lib32* 和 *System/lib64* 子目录下。

操作系统	编译器	导入库
Windows	Microsoft Visual C++	*dblibtm.lib*
Windows Mobile	Microsoft Visual C++ 2005	*dblib11.lib*
Windows Mobile	Microsoft eMbedded Visual C++	*dblib11.lib*
Unix（非线程应用程序）	所有编译器	*libdblib11.so*、*libdbtasks11.so*、*libdblib11.sl*、*libdbtasks11.sl*

操作系统	编译器	导入库
Unix（线程应用程序）	所有编译器	*libdblib11_r.so*、*libdbtasks11_r.so*、*libdblib11_r.sl*、*libdbtasks11_r.sl*
Mac OS X（线程应用程序）	所有编译器	*libdblib11.dylib*、*libdbtasks11.dylib*
Mac OS X（线程应用程序）	所有编译器	*libdblib11_r.dylib*、*libdbtasks11_r.dylib*

libdbtasks11 库由 *libdblib11* 库调用。有些编译器会自动查找 *libdbtasks11*。对于其它编译器，则需要您显式指定 libdbtasks11。

简单示例

下面是一个非常简单的嵌入式 SQL 程序示例。

```
#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 Employees
  SET Surname =  'Plankton'
  WHERE EmployeeID = 195;
 EXEC SQL COMMIT WORK;
 EXEC SQL DISCONNECT;
 db_fini( &sqlca );
 return( 0 );
error:
 printf( "update unsuccessful -- sqlcode = %ld\n",
  sqlca.sqlcode );
 db_fini( &sqlca );
 return( -1 );
}
```

本示例连接到数据库，更新 195 号雇员的姓氏，提交更改，然后退出。实际上 SQL 和 C 代码之间没有交互作用。在本示例中 C 代码仅用于控制流。WHENEVER 语句用于错误检查。错误处理（此示例中的 GOTO）会在任何引起错误的 SQL 语句之后执行。

有关读取数据的说明，请参见“读取数据”一节第 548 页。

嵌入式 SQL 程序的结构

SQL 语句置于（嵌入）常规 C 或 C++ 代码内。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;
```

使用嵌入式 SQL 的每个 C 程序都必须先初始化一个 SQLCA：

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

C 程序所执行的前几个嵌入式 SQL 语句之一必须是 CONNECT 语句。CONNECT 语句用于建立与数据库服务器的连接，以及指定连接期间用于授权执行的所有语句的用户 ID。

有些嵌入式 SQL 语句不生成任何 C 代码，或不涉及与数据库的通信。因此，允许在 CONNECT 语句之前使用这些语句。最主要的是 INCLUDE 语句和指定错误处理方法的 WHENEVER 语句。

使用嵌入式 SQL 的每个 C 程序都必须完成任何已初始化的 SQLCA。

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

在 Windows 中动态装载 DBLIB

开发使用 DLL 中的函数的应用程序通常的做法是：将应用程序链接到包含所需函数定义的**导入库**。

本节介绍了在开发 SQL Anywhere 应用程序时不使用导入库的方法。可以使用安装目录的 *SDK\C* 子目录下的 *esqldll.c* 模块动态装载 DBLIB，而不必链接导入库。

可以使用类似的方法在 Unix 平台上动态装载 DBLIB。

♦ 动态装载接口 DLL

1. 您的程序必须调用 db_init_dll 才能装载 DLL，而且必须调用 db_fini_dll 才能释放 DLL。必须在调用数据库接口中的任何函数之前调用 db_init_dll，而且在调用 db_fini_dll 之后不能调用接口中的任何函数。

   您还必须调用 db_init 和 db_fini 库函数。

2. 您必须在 EXEC SQL INCLUDE SQLCA 语句之前使用 #include 将 *esqldll.h* 头文件包含进来或在嵌入式 SQL 程序中包含 #include <*sqlca.h*> 行。*esqldll.h* 头文件包括 *sqlca.h*。

3. 必须定义一个 SQL OS 宏。*sqlca.h* 所包括的头文件 *sqlos.h* 会尝试确定适合的宏并对其进行定义。但是，平台和编译器的某些组合可能导致此操作失败。在这种情况下，您必须将 #define 添加到此文件的顶部，或者使用编译器选项进行定义。必须为 Windows 定义的宏如下所示。

宏	平台
_SQL_OS_WINDOWS	所有 Windows 操作系统

4. 编译 *esqldll.c*。

5. 将对象模块 *esqldll.obj* 与嵌入式 SQL 应用程序对象链接在一起，而不链接到导入库。

示例

您可以在 *samples-dir\SQLAnywhere\ESQLDynamicLoad* 目录中找到说明如何动态装载接口库的示例程序。源代码在 *sample.sqc* 中。

示例嵌入式 SQL 程序

示例嵌入式 SQL 程序随 SQL Anywhere 的安装提供，它们位于 *samples-dir\SQLAnywhere\C* 目录中。对于 Windows Mobile，*samples-dir\SQLAnywhere\CE\esql_sample* 目录中还有一个附加示例。

- 静态游标嵌入式 SQL 示例 *cur.sqc* 演示如何使用静态 SQL 语句。
- 动态游标嵌入式 SQL 示例 *dcur.sqc* 演示如何使用动态 SQL 语句。

为了减少示例程序重复的代码量，已经将主线和数据打印函数置于单独的文件中。对于字符模式系统，该文件为 *mainch.c*；对于窗口环境，该文件为 *mainwin.c*。

每个示例程序都提供了以下三个从主线调用的例程：

- **WSQLEX_Init** 连接到数据库并打开游标。
- **WSQLEX_Process_Command** 处理来自用户的命令，根据需要操作游标。
- **WSQLEX_Finish** 关闭游标并断开与数据库的连接。

主线的功能是：

1. 调用 WSQLEX_Init 例程。
2. 执行循环，从用户获取命令并调用 WSQL_Process_Command，直到用户退出。
3. 调用 WSQLEX_Finish 例程。

连接到数据库这一操作是通过提供相应用户 ID 和口令的嵌入式 SQL CONNECT 语句完成的。

除了这些示例之外，您还可以找到作为 SQL Anywhere 的一部分、演示可用于特定平台的功能的其它程序和源文件。

构建示例程序

用于构建示例程序的文件随示例代码提供。

- 对于 Windows 系统，使用 *build.bat* 或 *build64.bat* 来编译示例程序。

  对于 x64 平台版本，您可能需要设置正确的编译和链接环境。以下示例构建了一个适用于 x64 平台的示例程序。

  ```
  set mssdk=c:\MSSDK\v6.1
  build64
  ```

- 对于 Unix，使用 shell 脚本 *build.sh*。
- 对于 Windows Mobile，使用适用于 Microsoft Visual C++ 的 *esql_sample.sln* 项目文件。此文件位于 *samples-dir\SQLAnywhere\CE\esql_sample* 中。

这会构建以下示例。

- **CUR** 一个嵌入式 SQL 静态游标示例

- **DCUR** 一个嵌入式 SQL 动态游标示例
- **ODBC** 一个 ODBC 示例，“ODBC 示例”一节第 453 页中进行了介绍。

运行示例程序

可执行文件及相应的源代码位于 *samples-dir\SQLAnywhere\C* 目录中。对于 Windows Mobile，*samples-dir\SQLAnywhere\CE\esql_sample* 目录中还有一个附加示例。

♦ 运行静态游标示例程序

1. 启动 SQL Anywhere 示例数据库 *demo.db*。
2. 对于 32 位 Windows，运行文件 *curwin.exe*。

   对于 64 位 Windows，运行文件 *curx64.exe*。

   对于 Unix，运行文件 *cur*。
3. 按照屏幕上的说明进行操作。

   各种不同的命令可操作数据库游标并在屏幕上显示查询结果。输入要执行的命令的字母。某些系统可能要求您在键入字母之后按 Enter 键。

♦ 运行动态游标示例程序

1. 对于 32 位 Windows，运行文件 *dcurwin.exe*。

   对于 64 位 Windows，运行文件 *dcurx64.exe*。

   对于 Unix，运行文件 *dcur*。
2. 每个示例程序都提供一个控制台类型的用户界面并提示您输入命令。输入以下连接字符串以连接到示例数据库：

   ```
   DSN=SQL Anywhere 11 Demo
   ```
3. 每个示例程序都提示您选择一个表。选择示例数据库中的一个表。例如，可以输入 **Customers** 或 **Employees**。
4. 按照屏幕上的说明进行操作。

   各种不同的命令可操作数据库游标并在屏幕上显示查询结果。输入所要执行命令的字母。某些系统可能要求您在键入字母之后按 Enter 键。

Windows 示例

Windows 版本的示例程序使用 Windows 图形用户界面。但是，为了使用户界面代码相对简单，已经进行了一些简化。特别是，这些应用程序除了重新打印提示之外，不会针对 WM_PAINT 消息重绘其 Windows。

静态游标示例

本示例演示如何使用游标。此处使用的特定游标从示例数据库中的 Employees 表检索特定信息。游标是静态声明的，也就是说，检索信息的实际 SQL 语句会被硬编码到源程序中。这是了解游标工作方式的一个良好起点。动态游标示例采用第一个示例并将其转换为使用动态 SQL 语句。请参见“动态游标示例”一节第 518 页。

有关可以在何处找到源代码以及如何构建此示例程序的信息，请参见“示例嵌入式 SQL 程序”一节第 516 页。

open_cursor 例程声明特定 SQL 查询的游标并打开此游标。

显示信息页是由 print 例程完成的。它会循环 *pagesize* 次，从游标中读取一行并将它显示输出。注意，fetch 例程检查警告条件（如 [**未找到行**]）并显示这些条件出现时的相应消息。另外，此程序还会将游标重新定位到出现在当前数据页顶部的行之前的行。

move、top 和 bottom 例程使用适当形式的 FETCH 语句来定位游标。注意，此形式的 FETCH 语句实际上不获取数据，它只定位游标。另外，move 作为一个通用的相对定位例程，被设计成按照参数的符号上下移动游标。

在用户退出时，会关闭游标，而且还会释放数据库连接。游标由 ROLLBACK WORK 语句关闭，而连接由 DISCONNECT 释放。

动态游标示例

本示例演示如何使用动态 SQL SELECT 语句的游标。它与静态游标示例稍有不同。如果还未查看静态游标示例，则在查看此示例前先查看静态游标示例会很有帮助。请参见“静态游标示例”一节第 518 页。

有关可以在何处找到源代码以及如何构建此示例程序的信息，请参见“示例嵌入式 SQL 程序”一节第 516 页。

dcur 程序允许用户选择要使用 n 命令查看的表。然后，该程序会根据屏幕大小尽可能多地显示表中的信息。

在运行此程序时，它会提示输入以下形式的连接字符串：

```
UID=DBA;PWD=sql;DBF=samples-dir\demo.db
```

含有嵌入式 SQL 的 C 程序位于 *samples-dir\SQLAnywhere\C* 目录中。对于 Windows Mobile，*samples-dir\SQLAnywhere\CE\esql_sample* 目录中提供了一个动态游标示例。除了 connect、open_cursor 和 print 函数之外，此程序看起来很像静态游标示例。

connect 函数使用嵌入式 SQL 接口函数 db_string_connect 连接到数据库。此函数还提供额外功能，可支持用于连接数据库的连接字符串。

open_cursor 例程首先构建 SELECT 语句

```
SELECT * FROM table-name
```

其中 *table-name* 是传递给例程的参数。然后，它使用此字符串准备动态 SQL 语句。

嵌入式 SQL DESCRIBE 语句用于以 SELECT 语句的结果填充 SQLDA 结构。

SQLDA 的大小
最初推测的是 SQLDA 的大小 (3)。如果此值不够大，则会使用数据库服务器返回的选择列表的实际大小来分配一个大小正确的 SQLDA。

然后，用缓冲区填充 SQLDA 结构以存放代表查询结果的字符串。fill_s_sqlda 例程将 SQLDA 中的所有数据类型转换为 DT_STRING 并分配适当大小的缓冲区。

然后，为此语句声明并打开游标。用于移动和关闭游标的其余例程保持不变。

fetch 例程稍有不同：它将结果放在 SQLDA 结构中，而不是放在一组主机变量中。print 例程有很大改动，可按屏幕宽度显示来自 SQLDA 结构的结果。print 例程还使用 SQLDA 的名称字段来显示每列的标题。

嵌入式 SQL 数据类型

要在程序和数据库服务器间传送信息，每个数据都必须有一个数据类型。嵌入式 SQL 数据类型常量以 DT_ 为前缀，并且可以在 *sqldef.h* 头文件中找到。您可以创建任何一种受支持类型的主机变量。您还可以在 SQLDA 结构中使用这些类型向数据库中传入和从中传出数据。

可以使用 *sqlca.h* 中列出的 DECL_ 宏定义这些数据类型的变量。例如，可使用 DECL_BIGINT 来声明保存 BIGINT 值的变量。

以下数据类型受嵌入式 SQL 编程接口的支持：

- **DT_BIT** 8 位有符号整数。
- **DT_SMALLINT** 16 位有符号整数。
- **DT_UNSSMALLINT** 16 位无符号整数。
- **DT_TINYINT** 8 位有符号整数。
- **DT_BIGINT** 64 位有符号整数。
- **DT_UNSBIGINT** 64 位无符号整数。
- **DT_INT** 32 位有符号整数。
- **DT_UNSINT** 32 位无符号整数。
- **DT_FLOAT** 4 字节浮点数。
- **DT_DOUBLE** 8 字节浮点数。
- **DT_DECIMAL** 压缩十进制数（专有格式）。

```
typedef struct TYPE_DECIMAL {
  char array[1];
} TYPE_DECIMAL;
```

- **DT_STRING** 在 CHAR 字符集中以空值终止的字符串。如果数据库是使用填补空白的字符串进行的初始化，则该字符串是填补空白的。
- **DT_NSTRING** 在 NCHAR 字符集中以空值终止的字符串。如果数据库是使用填补空白的字符串进行的初始化，则该字符串便是用空白填补的。
- **DT_DATE** 以空值终止的字符串，是一个有效日期。
- **DT_TIME** 以空值终止的字符串，是一个有效时间。
- **DT_TIMESTAMP** 以空值终止的字符串，是一个有效时间戳。
- **DT_FIXCHAR** 在 CHAR 字符集中填补空白的定长字符串。最大长度为 32767（以字节为单位）。该数据不以空值终止。
- **DT_NFIXCHAR** 在 NCHAR 字符集中填补空白的定长字符串。最大长度为 32767（以字节为单位）。该数据不以空值终止。
- **DT_VARCHAR** 在 CHAR 字符集中具有双字节长度字段的变长字符串。最大长度为 32765 个字节。发送数据时，必须设置长度字段。读取数据时，数据库服务器设置长度字段。该数据不是以空值终止或以空白填补的。


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

- **DT_NVARCHAR** 在 NCHAR 字符集中具有双字节长度字段的变长字符串。最大长度为 32765 个字节。发送数据时，必须设置长度字段。读取数据时，数据库服务器设置长度字段。该数据不是以空值终止或以空白填补的。

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

- **DT_LONGVARCHAR** 在 CHAR 字符集中长变长字符串。

```
typedef struct LONGVARCHAR {
 a_sql_uint32 array_len;  /* number of allocated bytes in array */
 a_sql_uint32 stored_len; /* number of bytes stored in array
                           * (never larger than array_len) */
 a_sql_uint32 untrunc_len;/* number of bytes in untruncated expression
                           * (may be larger than array_len) */
 char  array[1];          /* the data */
} LONGVARCHAR, LONGNVARCHAR, LONGBINARY;
```

LONGVARCHAR 结构可用于大于 32767 个字节的数据。对于较大的数据，可以一次全部读取，也可以使用 GET DATA 语句逐段读取。对于较大的数据，可以一次就全部提供给服务器，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或以空白填补的。

有关详细信息，请参见“发送和检索 Long 型值”一节第 555 页。

- **DT_LONGNVARCHAR** 在 NCHAR 字符集中长变长字符串。该宏定义一个如下结构：

```
typedef struct LONGVARCHAR {
 a_sql_uint32 array_len;  /* number of allocated bytes in array */
 a_sql_uint32 stored_len; /* number of bytes stored in array
                           * (never larger than array_len) */
 a_sql_uint32 untrunc_len;/* number of bytes in untruncated expression
                           * (may be larger than array_len) */
 char  array[1];          /* the data */
} LONGVARCHAR, LONGNVARCHAR, LONGBINARY;
```

LONGNVARCHAR 结构可用于大于 32767 个字节的数据。对于较大的数据，可以一次全部读取，也可以使用 GET DATA 语句逐段读取。对于较大的数据，可以一次就全部提供给服务器，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或以空白填补的。

有关详细信息，请参见“发送和检索 Long 型值”一节第 555 页。

- **DT_BINARY** 具有双字节长度字段的变长二进制数据。最大长度为 32765 个字节。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，服务器设置长度字段。

```
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：

```
typedef struct LONGVARCHAR {
 a_sql_uint32 array_len;  /* number of allocated bytes in array */
 a_sql_uint32 stored_len; /* number of bytes stored in array
                           * (never larger than array_len) */
 a_sql_uint32 untrunc_len;/* number of bytes in untruncated expression
                           * (may be larger than array_len) */
 char  array[1];          /* the data */
} LONGVARCHAR, LONGNVARCHAR, LONGBINARY;
```

  LONGBINARY 结构可用于大于 32767 个字节的数据。对于较大的数据，可以一次全部读取，也可以使用 GET DATA 语句逐段读取。对于较大的数据，可以一次就全部提供给服务器，也可以通过使用 SET 语句附加到数据库变量中来逐段提供。

  有关详细信息，请参见“发送和检索 Long 型值”一节第 555 页。

- **DT_TIMESTAMP_STRUCT** SQLDATETIME 结构，对于时间戳的每一部分都有一个字段。

```
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 结构可用于检索 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。从该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。

  请参见：

  - “date_format 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》
  - “date_order 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》
  - “time_format 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》
  - “timestamp_format 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》

- **DT_VARIABLE** 以空值终止的字符串。字符串必须是 SQL 变量的名称，数据库服务器使用该变量的值。此数据类型仅用于为数据库服务器提供数据。在从数据库服务器读取数据时，不能使用它。

这些结构在 *sqlca.h* 文件中定义。由于 VARCHAR、NVARCHAR、BINARY、DECIMAL 和 LONG 数据类型包含一个单字符数组，因此，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。这些数据库类型都是使用 SQLDATETIME 结构或字符串读取和更新的。

有关详细信息，请参见“GET DATA 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》和“SET 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用主机变量

主机变量是供 SQL 预处理器识别的 C 语言变量。主机变量可用于将值发送到数据库服务器或从数据库服务器接收值。

主机变量非常易于使用，但是它们具有一些限制。动态 SQL 是一种向数据库服务器和从数据库服务器中传递信息的更常用的方法，它使用被称为 [SQL 描述符区域] (SQLDA) 的结构。SQL 预处理器为使用主机变量的每个语句自动生成 SQLDA。

不能在批处理中使用主机变量。

有关动态 SQL 的信息，请参见“静态和动态 SQL”一节第 537 页。

声明主机变量

主机变量是通过将它们放入**声明部分**来定义的。按照 ANSI 嵌入式 SQL 标准，主机变量是通过用以下内容围绕常规 C 变量声明定义的：

```
EXEC SQL BEGIN DECLARE SECTION;
/* C variable declarations */
EXEC SQL END DECLARE SECTION;
```

然后，可以使用这些主机变量代替任意 SQL 语句中的值常量。在数据库服务器执行语句时，会使用主机变量的值。注意，主机变量不能用于代替表或列的名称：此参数需要动态 SQL。在 SQL 语句中，变量名以冒号 (:) 为前缀，以便与语句中允许使用的其它标识符区别开。

在 SQL 预处理器中，只在 DECLARE SECTION 内部扫描 C 语言代码。因此，在 DECLARE SECTION 内，不允许使用 TYPEDEF 类型和结构，但允许使用变量的初始化程序。

示例

下面的示例代码阐释了主机变量在 INSERT 语句中是如何使用的。这些变量由程序填充，然后插入到数据库中：

```
EXEC SQL BEGIN DECLARE SECTION;
long employee_number;
char employee_name[50];
char employee_initials[8];
char employee_phone[15];
EXEC SQL END DECLARE SECTION;
/* program fills in variables with appropriate values
*/
EXEC SQL INSERT INTO Employees
VALUES (:employee_number, :employee_name,
:employee_initials, :employee_phone );
```

有关更广泛的示例，请参见“静态游标示例”一节第 518 页。

C 主机变量类型

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。

sqlca.h 头文件中定义的宏可用于声明以下类型的主机变量：NCHAR、VARCHAR、NVARCHAR、LONGVARCHAR、LONGNVARCHAR、BINARY、LONGBINARY、DECIMAL、FIXCHAR、NFIXCHAR、DATETIME (SQLDATETIME)、BIT、BIGINT 或 UNSIGNED BIGINT。它们的使用方法如下所示：

```
EXEC SQL BEGIN DECLARE SECTION;
DECL_NCHAR                  v_nchar[10];
DECL_VARCHAR( 10 )          v_varchar;
DECL_NVARCHAR( 10 )         v_nvarchar;
DECL_LONGVARCHAR( 32768 )   v_longvarchar;
DECL_LONGNVARCHAR( 32768 )  v_longnvarchar;
DECL_BINARY( 4000 )         v_binary;
DECL_LONGBINARY( 128000 )   v_longbinary;
DECL_DECIMAL( 30, 6 )       v_decimal;
DECL_FIXCHAR( 10 )          v_fixchar;
DECL_NFIXCHAR( 10 )         v_nfixchar;
DECL_DATETIME               v_datetime;
DECL_BIT                    v_bit;
DECL_BIGINT                 v_bigint;
DECL_UNSIGNED_BIGINT        v_ubigint;
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。建议不要使用 DECIMAL（DT_DECIMAL、DECL_DECIMAL）类型，因为十进制数值的格式为专有格式。

下表列出了允许用于主机变量的 C 变量类型及其对应的嵌入式 SQL 接口数据类型。

C 数据类型	嵌入式 SQL 接口类型	说明
`short si;` `short int si;`	DT_SMALLINT	16 位有符号整数。
`unsigned short int usi;`	DT_UNSSMALLINT	16 位无符号整数。
`long l;` `long int l;`	DT_INT	32 位有符号整数。
`unsigned long int ul;`	DT_UNSINT	32 位无符号整数。
`DECL_BIGINT ll;`	DT_BIGINT	64 位有符号整数。
`DECL_UNSIGNED_BIGINT ull;`	DT_UNSBIGINT	64 位无符号整数。
`float f;`	DT_FLOAT	4 字节浮点数。
`double d;`	DT_DOUBLE	8 字节浮点数。
`char a[n]; /*n>=1*/`	DT_STRING	在 CHAR 字符集中以空值终止的字符串。如果数据库是使用填补空白的字符串进行的初始化，则该字符串是填补空白的。此变量占有 n-1 个字符加上空终止符。

C 数据类型	嵌入式 SQL 接口类型	说明
`char *a;`	DT_STRING	在 CHAR 字符集中以空值终止的字符串。此变量指向一个可保存最多 32766 个字节加上空终止符的区域。
`DECL_NCHAR a[n]; /*n>=1*/`	DT_NSTRING	在 NCHAR 字符集中以空值终止的字符串。如果数据库是使用填补空白的字符串进行的初始化，则该字符串便是用空白填补的。此变量占有 n-1 个字符加上空终止符。
`DECL_NCHAR *a;`	DT_NSTRING	在 NCHAR 字符集中以空值终止的字符串。此变量指向一个可保存最多 32766 个字节加上空终止符的区域。
`DECL_VARCHAR(n) a;`	DT_VARCHAR	在 CHAR 字符集中具有双字节长度字段的变长字符串。不是以空值终止的或填补空白的。n 的最大值为 32765（字节）。
`DECL_NVARCHAR(n) a;`	DT_NVARCHAR	在 NCHAR 字符集中具有双字节长度字段的变长字符串。不是以空值终止的或填补空白的。n 的最大值为 32765（字节）。
`DECL_LONGVARCHAR(n) a;`	DT_LONGVARCHAR	在 CHAR 字符集中具有三个 4 字节长度字段的变长长字符串。不是以空值终止的或以空白填补的。
`DECL_LONGNVARCHAR(n) a;`	DT_LONGNVARCHAR	在 NCHAR 字符集中具有三个 4 字节长度字段的变长长字符串。不是以空值终止的或以空白填补的。
`DECL_BINARY(n) a;`	DT_BINARY	具有长度为 2 个字节的字段的变长二进制数据。n 的最大值为 32765（字节）。
`DECL_LONGBINARY(n) a;`	DT_LONGBINARY	具有三个长度为 4 个字节的字段的长型变长二进制数据。

C 数据类型	嵌入式 SQL 接口类型	说明
`char a; /*n=1*/` `DECL_FIXCHAR(n) a;`	DT_FIXCHAR	在 CHAR 字符集中固定长度的字符串。是填补空白的但不是以空值终止的。n 的最大值为 32767（字节）。
`DECL_NCHAR a; /*n=1*/` `DECL_NFIXCHAR(n) a;`	DT_NFIXCHAR	在 NCHAR 字符集中固定长度的字符串。是填补空白的但不是以空值终止的。n 的最大值为 32767（字节）。
`DECL_DATETIME a;`	DT_TIMESTAMP_STRUCT	SQLDATETIME 结构

字符集

对于 DT_FIXCHAR、DT_STRING、DT_VARCHAR 和 DT_LONGVARCHAR，字符数据采用应用程序的 CHAR 字符集，此字符集通常是应用程序区域设置的字符集。应用程序可使用 CHARSET 连接参数或通过调用 db_change_char_charset 函数来更改 CHAR 字符集。

对于 DT_NFIXCHAR、DT_NSTRING、DT_NVARCHAR 和 DT_LONGNVARCHAR，数据采用应用程序的 NCHAR 字符集。缺省情况下，应用程序的 NCHAR 字符集与 CHAR 字符集相同。应用程序可以通过调用 db_change_nchar_charset 函数来更改 NCHAR 字符集。

有关地区和字符集的详细信息，请参见“了解区域设置”一节 《SQL Anywhere 服务器 - 数据库管理》。

有关更改 CHAR 字符集的详细信息，请参见“CharSet 连接参数 [CS]”一节 《SQL Anywhere 服务器 - 数据库管理》或“db_change_char_charset 函数”一节第 570 页。

有关更改 NCHAR 字符集的详细信息，请参见“db_change_nchar_charset 函数”一节第 571 页。

数据长度

无论使用的是 CHAR 还是 NCHAR 字符集，所有数据长度均以字节为单位指定。

如果服务器和应用程序之间发生字符集转换，则应用程序负责确保缓冲区足够大以便处理转换的数据，并在数据被截断时发出附加 GET DATA 语句。

字符指针

数据库接口认为一个声明为**字符指针** (*char * a*) 的主机变量的长度为 32767 个字节。用于从数据库检索信息的任何字符指针类型的主机变量都必须指向一个缓冲区，该缓冲区的大小要足以容纳可能从数据库返回的任何值。

这具有相当大的潜在危险，因为可能会有人更改数据库中列的定义，使列比编写程序时更大。这可能会导致随机内存损坏问题。使用声明数组会比较好，甚至是作为函数的参数，其中数组是作为字符指针传递的。此技术可让嵌入式 SQL 语句知道数组的大小。

主机变量的范围

标准主机变量声明部分可以出现在通常声明 C 变量的任意位置，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，所以它不会考虑 C 语言块。

就 SQL 预处理器而言，主机变量对源文件来说是全局的；两个主机变量不能同名。

主机变量的用法

主机变量可用于以下环境中：

- 位于任何允许使用数字或字符串常量的位置的 SELECT、INSERT、UPDATE 和 DELETE 语句。
- SELECT 和 FETCH 语句的 INTO 子句。
- 主机变量也可用来代替语句名、游标名或嵌入式 SQL 特定语句中的选项名。
- 对于 CONNECT、DISCONNECT 和 SET CONNECT 语句，可以用主机变量代替服务器名、数据库名、连接名、用户 ID、口令或连接字符串。
- 对于 SET OPTION 和 GET OPTION，可以用主机变量代替用户 ID、选项名或选项值。
- 不能用主机变量代替任何语句中的表名或列名。

SQLCODE 和 SQLSTATE 主机变量

ISO/ANSI 标准允许嵌入式 SQL 源文件在声明部分中声明以下特殊主机变量：

```
long SQLCODE;
char SQLSTATE[6];
```

如果使用这些变量，则在进行数据库请求的任何嵌入式 SQL 语句（DECLARE SECTION、INCLUDE、WHENEVER SQLCODE 等之外的 EXEC SQL 语句）之后设置这些变量。

SQLCODE 和 SQLSTATE 主机变量必须在生成数据库请求的每个嵌入式 SQL 语句的范围内可见。

有关详细信息，请参见“SQL 预处理器”一节第 563 页中 sqlpp -k 选项的说明。

下面是有效的嵌入式 SQL：

```
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
long SQLCODE;
EXEC SQL END DECLARE SECTION;
sub1() {
 EXEC SQL BEGIN DECLARE SECTION;
 char SQLSTATE[6];
 EXEC SQL END DECLARE SECTION;
 exec SQL CREATE TABLE ...
}
```

下面是无效的嵌入式 SQL：

```
EXEC SQL INCLUDE SQLCA;
sub1() {
 EXEC SQL BEGIN DECLARE SECTION;
 char SQLSTATE[6];
```



```
 EXEC SQL END DECLARE SECTION;
 exec SQL CREATE TABLE...
}
sub2() {
 exec SQL DROP TABLE...
 // No SQLSTATE in scope of this statement
}
```

指示符变量

指示符变量是在您读取或保存数据时存放补充信息的 C 变量。指示符变量有以下几种不同的用法：

- **NULL 值** 使应用程序可以处理 NULL 值。
- **字符串截断** 使应用程序可以处理必须截断读取值以适合主机变量的情况。
- **转换错误** 保存错误消息。

指示符变量是在 SQL 语句中紧跟常规主机变量放置的 short int 类型的主机变量。例如，在下面的 INSERT 语句中，:ind_phone 是一个指示符变量：

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

如果在读取或执行操作中，未从数据库服务器接收到任何行（由于出错或者到达结果集的末尾），则指示符值保持不变。

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 语言常量表示未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。因此，要将 NULL 值传递到数据库或接收回 NULL 结果，除了常规主机变量外，还需要其它额外的条件。**指示符变量**正是用于此目的。

插入 NULL 时使用指示符变量

INSERT 语句可以包括一个指示符变量，如下所示：

```
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;
/*
This program fills in the employee number,
name, initials, and phone number.
*/
```

```
if( /* Phone number is unknown */ ) {
 ind_phone = -1;
} else {
 ind_phone = 0;
}
EXEC SQL INSERT INTO Employees
 VALUES (:employee_number, :employee_name,
 :employee_initials, :employee_phone:ind_phone );
```

如果指示符变量值为 -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，而又没有提供指示符变量，就会生成错误 (SQLE_NO_INDICATOR)。

将指示符变量用于截断值

指示符变量指示某些读取值是否为适合主机变量而被截断了。这使应用程序可以正确地处理截断问题。

如果一个值在读取时被截断，则指示符变量会被设为正值，其中包含了在被截断前数据库值的实际长度。如果值的长度大于 32767 个字节，则指示符变量值为 32767。

将指示符变量用于转换错误

缺省情况下，conversion_error 数据库选项设置为 On，任何数据类型转换失败都将导致一个错误，且不返回行。

您可以使用指示符变量来告知哪列导致了数据类型转换失败。如果您将数据库选项 conversion_error 设置为 Off，则任何数据类型转换失败都将给出 CANNOT_CONVERT 警告，而不是错误。如果遇到转换错误的列具有一个指示符变量，则将该变量的值设置为 -2。

如果您在向数据库中插入数据时将 conversion_error 选项设置为 Off，则发生转换失败时就会插入 NULL 值。

指示符变量值概览

下表提供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	检索的值被截断—指示符变量中的实际长度
0	主机变量的值	读取成功，或 conversion_error 设置为 On
-1	NULL 值	NULL 结果

指示符的值	向数据库提供值	从数据库接收值
-2	NULL 值	转换错误（仅当 conversion_error 设置为 Off 时）。SQLCODE 指示一个 CANNOT_CONVERT 警告
< -2	NULL 值	NULL 结果

有关检索长整型值的详细信息，请参见“GET DATA 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

SQL 通信区域 (SQLCA)

SQL 通信区域 (SQLCA) 是一个内存区域，每个数据库请求都会利用这个区域将统计信息和错误从应用程序传递到数据库服务器再回传到应用程序。SQLCA 用作应用程序到数据库的通信链接的句柄。它会被传递到需要与数据库服务器进行通信的所有数据库库函数中。它在所有嵌入式 SQL 语句中被隐式传递。

全局 SQLCA 变量在接口库中定义。预处理器会为全局 SQLCA 变量生成外部引用，并且会为该变量的指针生成外部引用。该外部引用名为 sqlca，类型为 SQLCA。指针名为 sqlcaptr。实际的全局变量在导入库中声明。

SQLCA 由 *sqlca.h* 头文件定义，该文件包括在安装目录的 *SDK\Include* 子目录中。

SQLCA 提供错误代码

可引用 SQLCA 来测试特定错误代码。当数据库请求有错误时，sqlcode 和 sqlstate 字段包含错误代码。某些 C 宏是为引用 sqlcode 字段、sqlstate 字段和某些其它字段而定义的。

SQLCA 字段

SQLCA 中的字段具有以下含义：

- **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（136 字节）的长型整数。
- **sqlcode** 数据库在请求上检测到错误时，指定错误代码的长整数。错误代码的定义可在头文件 *sqlerr.h* 中找到。错误代码是 0（零）表示操作成功，正数表示警告，负数表示错误。

  有关错误代码的完整列表，请参见错误消息。
- **sqlerrml** sqlerrmc 字段中信息的长度。
- **sqlerrmc** 要插入到错误消息中的零个或多个字符串。某些错误消息包含一个或多个占位符字符串（*%1*、*%2*、……），这些占位符字符串可由此字段中的字符串替换。

  例如，如果生成 [**未找到表**] 错误，则 sqlerrmc 包含表名，该表名会插入到错误消息中的相应位置。

  有关错误消息的完整列表，请参见错误消息。
- **sqlerrp** 保留。
- **sqlerrd** 长整数的实用程序数组。
- **sqlwarn** 保留。
- **sqlstate** SQLSTATE 状态值。除了 SQLCODE 值外，ANSI SQL 标准还定义了 SQL 语句的此种类型的返回值。SQLSTATE 值始终是一个由五个字符组成且以空值终止的字符串，它分为双字符类（前两个字符）和三字符子类。每个字符都可以是从 0 到 9 的数字或从 A 到 Z 的大写字母字符。

以 0 到 4 或 A 到 H 开头的任何类或子类都是由 SQL 标准定义的，其它类和子类则是各实现自行定义的。SQLSTATE 值 '00000' 表示还没有错误或警告。

有关更多的 SQLSTATE 值，请参见“按 SQLSTATE 排序的 SQL Anywhere 错误消息”一节 《错误消息》。

sqlerror 数组

sqlerror 字段数组具有以下元素。

- **sqlerrd[1] (SQLIOCOUNT)** 完成某条语句所需的实际输入/输出操作数。

  数据库服务器执行每条语句之前不会将此值设置为零。在执行一个语句序列之前，您的程序可以将此变量设置为零。执行完最后一条语句之后，该数字为整个语句序列的输入/输出操作的总数。

- **sqlerrd[2] (SQLCOUNT)** 此字段的值取决于要执行的语句。

  - **INSERT、UPDATE、PUT 和 DELETE 语句** 受语句影响的行数。
  - **OPEN 语句** 在游标 OPEN 上，此字段由游标中的实际行数（大于*或等于*0 的值）或它的估计值（绝对值为估计值的负数）填充。如果数据库服务器不统计该值即可计算出行数，则该值就是实际行数。也可以使用 row_counts 选项，将数据库配置为始终返回实际的行数。
  - **FETCH 游标语句** 如果返回 SQLE_NOTFOUND 警告，则填充 SQLCOUNT 字段。它包含 FETCH RELATIVE 或 FETCH ABSOLUTE 语句超出可能的游标位置（游标可以位于某一行上、第一行之前或最后一行之后）范围之外的行数。在宽读取的情况下，SQLCOUNT 是实际读取的行数，它小于或等于请求的行数。在宽读取过程中，只在未返回行的情况下才设置 SQLE_NOTFOUND。

    有关宽读取的详细信息，请参见“一次读取多个行”一节第 551 页。

    如果未找到行但位置有效，则值为 0，例如，当定位到游标的最后一行上时执行 FETCH RELATIVE 1。如果所尝试的读取超出了游标的末尾，则为正值；如果所尝试的读取位于游标开头的前面，则为负值。
  - **GET DATA 语句** SQLCOUNT 字段保存值的实际长度。
  - **DESCRIBE 语句** 在用于说明可能具有多个结果集的过程的 WITH VARIABLE RESULT 子句中，SQLCOUNT 设置为以下值之一：

    - **0** 结果集可能更改：应该在每个 OPEN 语句后重新描述过程调用。
    - **1** 结果集是固定的。不需要重新进行描述。

    如果出现语法错误 SQLE_SYNTAX_ERROR，此字段包含语句内检测到错误的大致字符位置。

- **sqlerrd[3] (SQLIOESTIMATE)** 完成语句所需的输入/输出操作的估计数。OPEN 或 EXPLAIN 语句将为此字段赋一个值。

多线程代码或重入代码的 SQLCA 管理

您可以在多线程代码或重入代码中使用嵌入式 SQL 语句。但是，如果您使用单个连接，则对于每个连接您只能有一个活动请求。在多线程应用程序中，除非使用信号控制访问，否则在每个线程上不应使用到数据库的同一连接。

对于在希望使用数据库的每个线程上使用单独的连接没有限制。运行时库使用 SQLCA 来区分不同的线程上下文。因此，希望并行使用数据库的每个线程都必须具有自己的 SQLCA。

任何给定的数据库连接都只能从一个 SQLCA 访问，但取消指令除外，此指令必须从单独的线程发出。

有关取消请求的信息，请参见“实现请求管理”一节第 562 页。

以下是一个多线程嵌入式 SQL 重入代码的示例。

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <ctype.h>
#include <stdlib.h>
#include <process.h>
#include <windows.h>
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;

#define TRUE 1
#define FALSE 0

// multithreading support

typedef struct a_thread_data {
    SQLCA sqlca;
    int num_iters;
    int thread;
    int done;
} a_thread_data;

// each thread's ESQL test

EXEC SQL SET SQLCA "&thread_data->sqlca";

static void PrintSQLError( a_thread_data * thread_data )
/*****************************************************/
{
  char               buffer[200];

  printf( "%d: SQL error %d -- %s ... aborting\n",
      thread_data->thread,
      SQLCODE,
      sqlerror_message( &thread_data->sqlca,
                buffer, sizeof( buffer ) ) );
    exit( 1 );
}

EXEC SQL WHENEVER SQLERROR { PrintSQLError( thread_data ); };

static void do_one_iter( void * data )
{
  a_thread_data *  thread_data = (a_thread_data *)data;
  int    i;
```



```
  EXEC SQL BEGIN DECLARE SECTION;
  char    user[ 20 ];
  EXEC SQL END DECLARE SECTION;

  if( db_init( &thread_data->sqlca ) != 0 ) {
    for( i = 0; i < thread_data->num_iters; i++ ) {
     EXEC SQL CONNECT "dba" IDENTIFIED BY "sql";
     EXEC SQL SELECT USER INTO :user;
     EXEC SQL DISCONNECT;
    }
    printf( "Thread %d did %d iters successfully\n",
        thread_data->thread, thread_data->num_iters );
    db_fini( &thread_data->sqlca );
  }
  thread_data->done = TRUE;
}

int main()
{
  int num_threads = 4;
  int thread;
  int num_iters = 300;
  int num_done = 0;
  a_thread_data *thread_data;
  thread_data = (a_thread_data *)malloc( sizeof( a_thread_data ) *
num_threads );
  for( thread = 0; thread < num_threads; thread++ ) {
    thread_data[ thread ].num_iters = num_iters;
    thread_data[ thread ].thread = thread;
    thread_data[ thread ].done = FALSE;
    if( _beginthread( do_one_iter,
      8096,
      (void *)&thread_data[thread] ) <= 0 ) {
      printf( "FAILED creating thread.\n" );
      return( 1 );
    }
  }
    while( num_done != num_threads ) {
    Sleep( 1000 );
    num_done = 0;
    for( thread = 0; thread < num_threads; thread++ ) {
      if( thread_data[ thread ].done == TRUE ) {
        num_done++;
      }
    }
  }
  return( 0 );
}
```

使用多个 SQLCA

- **在您的应用程序中管理多个 SQLCA**

1. 不要在 SQL 预处理器上使用生成非重入代码的选项 (-r-)。由于无法使用静态初始化的全局变量，因此重入代码稍大且速度稍慢。不过，这些影响是很小的。
2. 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。

3. 嵌入式 SQL 语句 SET SQLCA 用于通知 SQL 预处理器对数据库请求使用不同的 SQLCA。通常，会在程序顶部或头文件中使用类似 `EXEC SQL SET SQLCA 'task_data->sqlca';` 的语句，将 SQLCA 引用设置为指向任务特定的数据。由于此语句不生成任何代码，因而不影响性能。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 的信息，请参见“SET SQLCA 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

何时使用多个 SQLCA

您可以在任一受支持的嵌入式 SQL 环境中使用多个 SQLCA 支持，但仅在重入代码中要求这样做。

下面的列表详细说明必须使用多个 SQLCA 的环境：

- **多线程应用程序** 每个线程都必须具有自己的 SQLCA。当 DLL 使用嵌入式 SQL 且被应用程序中的多个线程调用时，也必须如此。
- **动态链接库和共享库** DLL 只有一个数据段。数据库服务器在处理一个应用程序发出的请求时，也可能会优先处理另一个应用程序向该数据库服务器发出的请求。如果您的 DLL 使用全局 SQLCA，则这两个应用程序会同时使用它。每个 Windows 应用程序都必须具有自己的 SQLCA。
- **具有一个数据段的 DLL** 可以将 DLL 创建为只有一个数据段，或者对于每个应用程序有一个数据段。如果您的 DLL 只有一个数据段，则无法使用全局 SQLCA，其原因与 DLL 无法使用全局 SQLCA 的原因相同。每个应用程序必须具有自己的 SQLCA。

用多个 SQLCA 进行连接管理

您无需使用多个 SQLCA 以连接到多个数据库或具有到单个数据库的多个连接。

每个 SQLCA 都可以具有一个未命名的连接。每个 SQLCA 都具有一个活动的或当前的连接，请参见“SET CONNECTION 语句 [Interactive SQL] [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

在所给定数据库连接上进行的所有操作都必须使用建立数据库连接时所使用的同一个 SQLCA。

记录锁定
不同连接上的操作受常规记录锁定机制的影响，并且可能导致互相阻塞，甚至可能导致死锁。有关锁定的信息，请参见“使用过程、触发器和批处理” 《SQL Anywhere 服务器 - SQL 的用法》。

静态和动态 SQL

有以下两种方式可以将 SQL 语句嵌入到 C 程序中：

- 静态语句
- 动态语句

在此之前，我们一直在讨论静态 SQL。本节将对静态 SQL 和动态 SQL 进行比较。

静态 SQL 语句

通过在所有标准 SQL 数据操作语句和数据定义语句之前加上 EXEC SQL，并在语句之后加上分号 (;)，可以将所有这些语句嵌入到 C 程序中。这些语句称为**静态**语句。

静态语句可以包含对主机变量的引用。此前的所有示例中使用的都是静态嵌入式 SQL 语句。请参见“使用主机变量”一节第 524 页。

主机变量只能用来代替字符串或数字常量，不能用来代替列名或表名；执行那些操作需要使用动态语句。

动态 SQL 语句

在 C 语言中，字符串存储在字符数组中。动态语句是用 C 语言字符串构造的。然后，可以使用 PREPARE 和 EXECUTE 语句执行这些语句。这些 SQL 语句无法按与静态语句相同的方式引用主机变量，因为在执行 C 程序时无法通过名称访问 C 语言变量。

要在语句与 C 语言变量之间传递信息，请使用名为 **SQL 描述符区域 (SQLDA)** 的数据结构。如果您在 EXECUTE 语句的 USING 子句中指定一组主机变量，则 SQL 预处理器会为您建立此结构。这些变量按位置对应于预准备语句相应位置中的占位符。

有关 SQLDA 的信息，请参见“SQL 描述符区域 (SQLDA)”一节第 540 页。

将**占位符**放在语句中是为了指示要访问主机变量的位置。占位符可以是问号 (?)，或者也可以是静态语句中的主机变量引用（主机变量名之前有一个冒号）。在后一种情况下，在语句的实际文本中使用的主机变量名仅充当占位符，指示对 SQL 描述符区域的引用。

用于向数据库传递信息的主机变量称为**绑定变量**。

示例

例如：

```
EXEC SQL BEGIN DECLARE SECTION;
char comm[200];
char Street[30];
char City[20];
short int cityind;
long empnum;
EXEC SQL END DECLARE SECTION;

...
```

```
sprintf( comm,
    "UPDATE %s SET Street = :?, City = :?"
    "WHERE EmployeeID = :?",
    tablename );
EXEC SQL PREPARE S1 FROM :comm;
EXEC SQL EXECUTE S1 USING :Street, :City:cityind, :empnum;
```

此方法要求您知道语句中共有多少个主机变量。但程序员对这一点往往并不清楚。因此，可以建立您自己的 SQLDA 结构，并在 EXECUTE 语句的 USING 子句中指定此 SQLDA。

DESCRIBE BIND VARIABLES 语句返回在预准备语句中找到的绑定变量的主机变量名。这使 C 程序可以更容易地管理主机变量。一般的方法如下：

```
EXEC SQL BEGIN DECLARE SECTION;
char comm[200];
EXEC SQL END DECLARE SECTION;
...
sprintf( comm, "UPDATE %s set Street = :Street,
    City = :City"
   " WHERE EmployeeID = :empnum",
   tablename );
EXEC SQL PREPARE S1 FROM :comm;
/* Assume that there are no more than 10 host variables.
 * See next example if you cannot put a limit on it. */
sqlda = alloc_sqlda( 10 );
EXEC SQL DESCRIBE BIND VARIABLES FOR S1 INTO sqlda;
/* sqlda->sqld will tell you how many
   host variables there were. */
/* Fill in SQLDA_VARIABLE fields with
   values based on name fields in sqlda. */
...
EXEC SQL EXECUTE S1 USING DESCRIPTOR sqlda;
free_sqlda( sqlda );
```

SQLDA 的内容

SQLDA 包含一组变量描述符。每个描述符说明对应的 C 程序变量的属性或者数据库存储数据的位置或检索数据的位置：

- 数据类型
- 如果 *type* 是字符串类型，则为长度
- 内存地址
- 指示符变量

有关 SQLDA 结构的完整说明，请参见“SQL 描述符区域 (SQLDA)”一节第 540 页。

指示符变量和 NULL

指示符变量用于将 NULL 值传递到数据库或从数据库检索 NULL 值。数据库服务器还使用指示符变量指示在数据库操作过程中遇到的截断条件。当提供的空间不足，无法接收数据库值时，会将指示符变量设置为正值。

有关详细信息，请参见“指示符变量”一节第 529 页。

动态 SELECT 语句

可以动态准备仅返回单个行的 SELECT 语句，后跟带 INTO 子句的 EXECUTE 以检索单行结果。但是，返回多个行的 SELECT 语句是使用动态游标管理的。

使用动态游标时，将结果放在 FETCH 语句（FETCH INTO 和 FETCH USING DESCRIPTOR）指定的主机变量列表或 SQLDA 中。由于 C 程序员通常不知道选择列表项数，因此最常使用 SQLDA。DESCRIBE SELECT LIST 语句建立具有选择列表项的类型的 SQLDA。然后，使用 fill_sqlda 或 fill_s_sqlda 函数为这些值分配空间，由 FETCH USING DESCRIPTOR 语句检索信息。

典型的情况如下：

```
EXEC SQL BEGIN DECLARE SECTION;
char comm[200];
EXEC SQL END DECLARE SECTION;
int actual_size;
SQLDA * sqlda;
...
sprintf( comm, "SELECT * FROM %s", table_name );
EXEC SQL PREPARE S1 FROM :comm;
/* Initial guess of 10 columns in result.
   If it is wrong, it is corrected right
   after the first DESCRIBE by reallocating
   sqlda and doing DESCRIBE again. */
sqlda = alloc_sqlda( 10 );
EXEC SQL DESCRIBE SELECT LIST FOR S1
  INTO sqlda;
if( sqlda->sqld > sqlda->sqln )
{
  actual_size = sqlda->sqld;
  free_sqlda( sqlda );
  sqlda = alloc_sqlda( actual_size );
  EXEC SQL DESCRIBE SELECT LIST FOR S1
    INTO sqlda;
}
fill_sqlda( sqlda );
EXEC SQL DECLARE C1 CURSOR FOR S1;
EXEC SQL OPEN C1;
EXEC SQL WHENEVER NOTFOUND {break};
for( ;; )
{
  EXEC SQL FETCH C1 USING DESCRIPTOR sqlda;
  /* do something with data */
}
EXEC SQL CLOSE C1;
EXEC SQL DROP STATEMENT S1;
```

使用后删除语句
为避免占用不必要的资源，请确保在使用语句后将其删除。

有关使用动态选择语句的游标的完整示例，请参见“动态游标示例”一节第 518 页。

有关上述函数的详细信息，请参见“库函数参考”一节第 566 页。

SQL 描述符区域 (SQLDA)

SQLDA（SQL 描述符区域）是一个用于动态 SQL 语句的接口结构。此结构可将有关主机变量和 SELECT 语句结果的信息传入和传出数据库。SQLDA 在头文件 *sqlda.h* 中定义。

数据库接口库或 DLL 中提供了可用于管理 SQLDA 的函数。有关说明，请参见“库函数参考”一节第 566 页。

当主机变量与静态 SQL 语句一起使用时，预处理器会为这些主机变量构造 SQLDA。实际上，被传入传出数据库服务器的正是此 SQLDA。

SQLDA 头文件

sqlda.h 的内容如下：

```
#ifndef _SQLDA_H_INCLUDED
#define _SQLDA_H_INCLUDED
#define II_SQLDA

#include "sqlca.h"

#if defined( _SQL_PACK_STRUCTURES )
#include "pshpk1.h"
#endif

#define SQL_MAX_NAME_LEN  30

#define _sqldafar
typedef short int   a_sql_type;
struct sqlname
{
  short int length; /* length of char data */
  char  data[ SQL_MAX_NAME_LEN ]; /* data */
};
struct sqlvar
{   /* array of variable descriptors */
  short int  sqltype; /* type of host variable */
  short int  sqllen;  /* length of host variable */
  void      *sqldata; /* address of variable */
  short int *sqlind;  /* indicator variable pointer */
  struct sqlname  sqlname;
};
struct sqlda
{
  unsigned char sqldaid[8]; /* eye catcher "SQLDA" */
  a_sql_int32   sqldabc; /* length of sqlda structure */
  short int     sqln;    /* descriptor size in number of entries */
  short int     sqld;    /* number of variables found by DESCRIBE */
  struct sqlvar sqlvar[1]; /* array of variable descriptors */
};

typedef struct sqlda    SQLDA;
typedef struct sqlvar   SQLVAR, SQLDA_VARIABLE;
typedef struct sqlname  SQLNAME, SQLDA_NAME;
#ifndef SQLDASIZE
#define SQLDASIZE(n)    ( sizeof( struct sqlda ) +  \
                          (n-1) * sizeof( struct sqlvar) )
#endif
```



```
#if defined( _SQL_PACK_STRUCTURES )
#include "poppk.h"
#endif
#endif
```

SQLDA 字段

SQLDA 的各字段的含义如下：

字段	说明
sqldaid	8 字节字符字段，包含用于标识 SQLCA 结构的字符串 **SQLDA**。在您查看内存内容时，该字段可帮助进行调试。
sqldabc	包含 SQLDA 结构的长度的长型整数。
sqln	sqlvar 数组中包含的变量描述符数。
sqld	有效的变量描述符（包含说明主机变量的信息）的数目。在向数据库服务器提供数据时，此字段由 DESCRIBE 语句设置，在向数据库服务器提供数据时，有时由程序员设置。
sqlvar	一组类型为 struct sqlvar 的描述符，每个描述符描述一个主机变量。

SQLDA 主机变量说明

SQLDA 中的每个 sqlvar 结构都说明一个主机变量。sqlvar 结构的各字段的含义如下：

- **sqltype** 此描述符描述的变量的类型。请参见“嵌入式 SQL 数据类型”一节第 520 页。

  低序位指示是否允许使用 NULL 值。有效的类型和常量定义可以在 *sqldef.h* 头文件中找到。

  此字段由 DESCRIBE 语句填充。在向数据库服务器提供数据或从中检索数据时，您可以将此字段设置为任何类型。任何必要的类型转换都会自动完成。

- **sqllen** 变量的长度。该长度的实际含义取决于类型信息和 SQLDA 的使用方式。

  对于 LONG VARCHAR、LONG NVARCHAR 和 LONG BINARY 数据类型，使用 DT_LONGVARCHAR、DT_LONGNVARCHAR 或 DT_LONGBINARY 数据类型结构的 array_len 字段，而不是使用 sqllen 字段。

  有关长度字段的详细信息，请参见“SQLDA sqllen 字段值”一节第 542 页。

- **sqldata** 指向此变量所占用的内存的指针。此内存必须对应于 sqltype 和 sqllen 字段。

  有关存储格式，请参见“嵌入式 SQL 数据类型”一节第 520 页。

  对于 UPDATE 和 INSERT 语句，如果 sqldata 指针为空指针，操作中就不会涉及此变量。对于 FETCH，如果 sqldata 指针是空指针，则不返回数据。换句话说，sqldata 指针返回的列是**未绑定列**。

如果 DESCRIBE 语句使用 LONG NAMES，则此字段保存结果集列的长名称。另外，如果 DESCRIBE 语句是 DESCRIBE USER TYPES 语句，则此字段保存用户定义的数据类型的长名称，而不是列的长名称。如果该类型是基类型，则此字段为空。

- **sqlind** 指向指示符值的指针。指示符值是 short int。负的指示符值表示是 NULL 值。正的指示符值表示此变量已经被 FETCH 语句截断，且指示符值包含截断前的数据长度。如果将 conversion_error 数据库选项设置为 Off，则值 -2 表示出现了转换错误。请参见“conversion_error 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》。

  有关详细信息，请参见“指示符变量”一节第 529 页。

  如果 sqlind 指针是空指针，则说明没有适用于此主机变量的指示符变量。

  DESCRIBE 语句也使用 sqlind 字段来指示参数类型。如果类型是用户定义的数据类型，则此字段将设置为 DT_HAS_USERTYPE_INFO。在此情况下，您可能希望执行 DESCRIBE USER TYPES 以获取有关用户定义的数据类型的信息。

- **sqlname** 类似 VARCHAR 的结构，如下所示：

```
struct sqlname {
   short int  length;
   char  data[ SQL_MAX_NAME_LEN ];
};
```

  它由 DESCRIBE 语句填充，在其它情况下不使用它。对于以下两种格式的 DESCRIBE 语句，此字段具有不同的含义：

  - **SELECT LIST** 名称数据缓冲区由选择列表中对应项的列标题填充。
  - **BIND VARIABLES** 名称数据缓冲区由曾用作绑定变量的主机变量的名称填充，或者，如果使用了未命名的参数标记，则用 "?" 填充。

  在 DESCRIBE SELECT LIST 语句中，出现的任何指示符变量都会用一个指示选择列表项是否可更新的标志来填充。有关此标志的详细信息可以在 *sqldef.h* 头文件中找到。

  如果 DESCRIBE 语句是 DESCRIBE USER TYPES 语句，则此字段保存用户定义数据类型的长名称，而不是列的长名称。如果该类型是基类型，则此字段为空。

SQLDA sqllen 字段值

在 SQLDA 中，sqlvar 结构的 sqllen 字段长度在与数据库服务器进行以下种类的交互作用时使用：

- **说明值** DESCRIBE 语句获取有关主机变量的信息（这些主机变量是存储从数据库检索的数据或将数据传递到数据库时所必需的）。请参见“说明值”一节第 543 页。
- **检索值** 从数据库检索值。请参见“检索值”一节第 546 页。
- **发送值** 向数据库发送信息。请参见“发送值”一节第 544 页。

本节将介绍这些交互作用。

下表逐一详细说明了这些交互作用。这些表列出了在 *sqldef.h* 头文件中找到的接口常量类型（**DT_** 类型）。这些常量应放在 SQLDA sqltype 字段中。

有关 sqltype 字段值的信息，请参见“嵌入式 SQL 数据类型”一节第 520 页。

在静态 SQL 中，也会使用 SQLDA，但它是由 SQL 预处理器生成并完全填充的。在这种静态情况下，这些表给出静态 C 语言主机变量类型和接口常量之间的对应关系。

说明值

下表为各种数据库类型指明由 DESCRIBE 语句返回的 sqllen 和 sqltype 结构成员的值（SELECT LIST 和 BIND VARIABLE DESCRIBE 语句）。在用户定义的数据库数据类型的情况下，则对基类型进行说明。

您的程序可以使用从 DESCRIBE 返回的类型和长度，也可以使用另一种类型。数据库服务器在任意两种类型之间执行类型转换。由 sqldata 字段指向的内存必须对应于 sqltype 和 sqllen 字段。嵌入式 SQL 类型通过对 sqltype 和 DT_TYPES 进行按位“与”操作 (sqltype & DT_TYPES) 来获得。

有关嵌入式 SQL 数据类型的信息，请参见“嵌入式 SQL 数据类型”一节第 520 页。

数据库字段类型	返回的嵌入式 SQL 类型	说明时返回的长度（以字节为单位）
BIGINT	DT_BIGINT	8
BINARY(n)	DT_BINARY	n
BIT	DT_BIT	1
CHAR(n)	DT_FIXCHAR	n
DATE	DT_DATE	最长的带格式字符串的长度
DECIMAL(p,s)	DT_DECIMAL	SQLDA 中长度字段的高字节设置为 p，低字节设置为 s
DOUBLE	DT_DOUBLE	8
FLOAT	DT_FLOAT	4
INT	DT_INT	4
LONG BINARY	DT_LONGBINARY	32767
LONG NVARCHAR	DT_LONGNVARCHAR[1]	32767
LONG VARCHAR	DT_LONGVARCHAR	32767
NCHAR(n)	DT_NFIXCHAR[1]	客户端的 NCHAR 字符集中最大字符长度的 n 倍

数据库字段类型	返回的嵌入式 SQL 类型	说明时返回的长度（以字节为单位）
NVARCHAR(n)	DT_NVARCHAR[1]	客户端的 NCHAR 字符集中最大字符长度的 n 倍
REAL	DT_FLOAT	4
SMALLINT	DT_SMALLINT	2
TIME	DT_TIME	最长的带格式字符串的长度
TIMESTAMP	DT_TIMESTAMP	最长的带格式字符串的长度
TINYINT	DT_TINYINT	1
UNSIGNED BIGINT	DT_UNSBIGINT	8
UNSIGNED INT	DT_UNSINT	4
UNSIGNED SMALLINT	DT_UNSSMALLINT	2
VARCHAR(n)	DT_VARCHAR	n

[1] 在嵌入式 SQL 中，缺省情况下 NCHAR、NVARCHAR 和 LONG NVARCHAR 被分别描述为 DT_FIXCHAR、DT_VARCHAR 和 DT_LONGVARCHAR。如果调用了 db_change_nchar_charset 函数，则这些类型被分别描述为 DT_NFIXCHAR、DT_NVARCHAR 和 DT_LONGNVARCHAR。请参见“db_change_nchar_charset 函数”一节第 571 页。

发送值

下表指明当您向数据库服务器提供数据时指定 SQLDA 中值的长度的方式。

在这种情况下，只允许使用表中显示的数据类型。在向数据库提供信息时，将 DT_DATE、DT_TIME 和 DT_TIMESTAMP 类型视为与 DT_STRING 相同；值必须是具有适当日期格式且以空值终止的字符串。

嵌入式 SQL 数据类型	设置长度的程序操作
DT_BIGINT	不需要任何操作。
DT_BINARY(n)	采用 BINARY 结构中的字段的长度。
DT_BIT	不需要任何操作。
DT_DATE	由终止的 \0 决定的长度。
DT_DOUBLE	不需要任何操作。

嵌入式 SQL 数据类型	设置长度的程序操作
DT_FIXCHAR(n)	SQLDA 中的长度字段决定字符串的长度。
DT_FLOAT	不需要任何操作。
DT_INT	不需要任何操作。
DT_LONGBINARY	忽略长度字段。请参见“发送 LONG 数据”一节 第 557 页。
DT_LONGNVARCHAR	忽略长度字段。请参见“发送 LONG 数据”一节 第 557 页。
DT_LONGVARCHAR	忽略长度字段。请参见“发送 LONG 数据”一节 第 557 页。
DT_NFIXCHAR(n)	SQLDA 中的长度字段决定字符串的长度。
DT_NSTRING	长度由终止的 \0 决定。如果 ansi_blanks 选项为 On 且数据库是以空白填充的，则 SQLDA 中的长度字段必须设置为包含该值的缓冲区的长度（至少为该值的长度加上终止空字符的空间）。
DT_NVARCHAR	采用 NVARCHAR 结构中的字段的长度。
DT_SMALLINT	不需要任何操作。
DT_STRING	长度由终止的 \0 决定。如果 ansi_blanks 选项为 On 且数据库是以空白填充的，则 SQLDA 中的长度字段必须设置为包含该值的缓冲区的长度（至少为该值的长度加上终止空字符的空间）。
DT_TIME	由终止的 \0 决定的长度。
DT_TIMESTAMP	由终止的 \0 决定的长度。
DT_TIMESTAMP_STRUCT	不需要任何操作。
DT_UNSBIGINT	不需要任何操作。
DT_UNSINT	不需要任何操作。
DT_UNSSMALLINT	不需要任何操作。
DT_VARCHAR(n)	采用 VARCHAR 结构中的字段的长度。
DT_VARIABLE	由终止的 \0 决定的长度。

检索值

下表指明在使用 SQLDA 从数据库中检索数据时长度字段的值。在检索数据时，从不修改 sqllen 字段。

在这种情况下，只允许使用表中显示的接口数据类型。在从数据库检索信息时，使用 DT_DATE、DT_TIME 和 DT_TIMESTAMP 数据类型与使用 DT_STRING 的方式是相同的。值会被设置为当前日期格式的字符串。

嵌入式 SQL 数据类型	在接收时程序必须将长度字段设置为	在读取值之后数据库返回长度信息的方式
DT_BIGINT	不需要任何操作。	不需要任何操作。
DT_BINARY(n)	BINARY 结构的最大长度 (n+2)。n 的最大值为 32765。	将 BINARY 结构的 len 字段设置为实际长度（以字节为单位）。
DT_BIT	不需要任何操作。	不需要任何操作。
DT_DATE	缓冲区的长度。	\0 位于字符串的末尾。
DT_DOUBLE	不需要任何操作。	不需要任何操作。
DT_FIXCHAR(n)	缓冲区的长度（以字节为单位）。n 的最大值为 32767。	通过填补空白至缓冲区的长度。
DT_FLOAT	不需要任何操作。	不需要任何操作。
DT_INT	不需要任何操作。	不需要任何操作。
DT_LONGBINARY	忽略长度字段。请参见“检索 LONG 数据”一节第 556 页。	忽略长度字段。请参见“检索 LONG 数据”一节第 556 页。
DT_LONGNVARCHAR	忽略长度字段。请参见“检索 LONG 数据”一节第 556 页。	忽略长度字段。请参见“检索 LONG 数据”一节第 556 页。
DT_LONGVARCHAR	忽略长度字段。请参见“检索 LONG 数据”一节第 556 页。	忽略长度字段。请参见“检索 LONG 数据”一节第 556 页。
DT_NFIXCHAR(n)	缓冲区的长度（以字节为单位）。n 的最大值为 32767。	通过填补空白至缓冲区的长度。
DT_NSTRING	缓冲区的长度。	\0 位于字符串的末尾。
DT_NVARCHAR(n)	NVARCHAR 结构的最大长度 (n+2)。n 的最大值为 32765。	将 NVARCHAR 结构的 len 字段设置为字符串的实际长度（以字节为单位）。
DT_SMALLINT	不需要任何操作。	不需要任何操作。

嵌入式 SQL 数据类型	在接收时程序必须将长度字段设置为	在读取值之后数据库返回长度信息的方式
DT_STRING	缓冲区的长度。	\0 位于字符串的末尾。
DT_TIME	缓冲区的长度。	\0 位于字符串的末尾。
DT_TIMESTAMP	缓冲区的长度。	\0 位于字符串的末尾。
DT_TIMESTAMP_ STRUCT	不需要任何操作。	不需要任何操作。
DT_UNSBIGINT	不需要任何操作。	不需要任何操作。
DT_UNSINT	不需要任何操作。	不需要任何操作。
DT_UNSSMALLINT	不需要任何操作。	不需要任何操作。
DT_VARCHAR(n)	VARCHAR 结构的最大长度 (n +2)。n 的最大值为 32765。	将 VARCHAR 结构的 len 字段设置为字符串的实际长度（以字节为单位）。

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：

- **最多返回一行的 SELECT 语句** 使用 INTO 子句将返回的值直接指派给主机变量。请参见“最多返回一行的 SELECT 语句”一节第 548 页。
- **SELECT 语句可能返回多行** 使用游标管理结果集的行。请参见“在嵌入式 SQL 中使用游标”一节第 549 页。

最多返回一行的 SELECT 语句

单行查询最多从数据库中检索一行。单行查询 SELECT 语句在选择列表之后和 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。如果查询结果包含多个行，则数据库服务器会返回一个错误。

如果查询结果中没有选定的行，则返回 [**未找到行**] 警告。在 SQLCA 结构中返回的错误和警告。请参见“SQL 通信区域 (SQLCA)”一节第 532 页。

示例

如果成功地从 Employees 表中读取了一行，则以下代码段返回 1；如果该行不存在，则返回 0；如果出现错误，则返回 -1。

```
EXEC SQL BEGIN DECLARE SECTION;
long   ID;
char   name[41];
char   Sex;
char   birthdate[15];
short int ind_birthdate;
EXEC SQL END DECLARE SECTION;
...
int find_employee( long Employees )
{
  ID = Employees;
  EXEC SQL SELECT GivenName ||
    ' ' || Surname, Sex, BirthDate
    INTO :name, :Sex,
      :birthdate:ind_birthdate
    FROM Employees
    WHERE EmployeeID = :ID;
  if( SQLCODE == SQLE_NOTFOUND )
  {
    return( 0 ); /* Employees not found */
  }
  else if( SQLCODE < 0 )
  {
    return( -1 ); /* error */
  }
  else
  {
    return( 1 ); /* found */
```



```
    }
}
```

在嵌入式 SQL 中使用游标

游标用于从其结果集中具有多个行的查询中检索行。**游标**是 SQL 查询的句柄或标识符，也是结果集中的一个位置。

有关游标的介绍，请参见“使用游标”一节第 31 页。

♦ 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 语句一次一行地从游标中检索结果。
4. 读取行，直到返回 [**未找到行**] 警告。

   在 SQLCA 结构中返回错误和警告。请参见“SQL 通信区域 (SQLCA)”一节第 532 页。
5. 使用 CLOSE 语句关闭游标。

缺省情况下，在事务（COMMIT 或 ROLLBACK 上）的结尾会自动关闭游标。用 WITH HOLD 子句打开的游标对于后续事务保持打开状态，直到它们被显式关闭。

以下是游标用法的简单示例：

```
void print_employees( void )
{
  EXEC SQL BEGIN DECLARE SECTION;
  char name[50];
  char Sex;
  char birthdate[15];
  short int ind_birthdate;
  EXEC SQL END DECLARE SECTION;
  EXEC SQL DECLARE C1 CURSOR FOR
    SELECT GivenName || ' ' || Surname,
      Sex, BirthDate
    FROM Employees;
  EXEC SQL OPEN C1;
  for( ;; )
  {
    EXEC SQL FETCH C1 INTO :name, :Sex,
        :birthdate:ind_birthdate;
    if( SQLCODE == SQLE_NOTFOUND )
    {
      break;
    }
    else if( SQLCODE < 0 )
    {
      break;
    }

    if( ind_birthdate < 0 )
    {
      strcpy( birthdate, "UNKNOWN" );
```

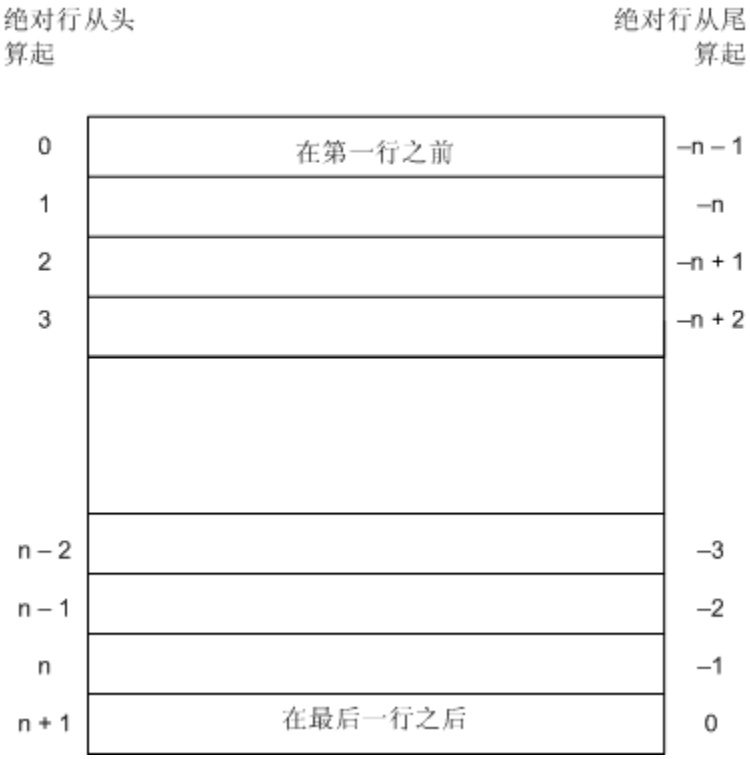
```
        }
        printf( "Name: %s Sex: %c Birthdate:
             %s.n",name, Sex, birthdate );
    }
    EXEC SQL CLOSE C1;
}
```

有关使用游标的完整示例，请参见“静态游标示例”一节第 518 页和“动态游标示例”一节第 518 页。

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



当游标打开之后，它位于第一行之前。可使用 FETCH 语句来移动游标位置。可以将游标定位到相对查询结果开头或结尾的一个绝对位置。也可以相对当前游标位置移动它。请参见“FETCH 语句 [ESQL] [SP]”一节 《SQL Anywhere 服务器 - SQL 参考》。

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，则会返回错误，指示游标中没有相应的行。

可以使用 PUT 语句向游标中插入行。请参见“PUT 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

游标定位问题

插入 DYNAMIC SCROLL 游标并对其进行某些更新会导致与游标定位有关的问题。除非 SELECT 语句中有 ORDER BY 子句，否则数据库服务器不将插入的行放在游标内可预知的位置。在有些情况下，插入的行要等到关闭并再次打开游标后才会出现。

对于 SQL Anywhere，如果必须创建临时表才能打开游标，则会出现这种情况。

有关说明，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》。

UPDATE 语句可导致行在游标中移动。如果游标具有一个使用现有索引（未创建临时表）的 ORDER BY 子句，则会出现这种情况。

一次读取多个行

可以将 FETCH 语句修改为一次读取多行，这样可能会改善性能。这种方式称为**宽读取**或**数组读取**。

SQL Anywhere 还支持宽放置和宽插入。请参见“PUT 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》和“EXECUTE 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

要在嵌入式 SQL 中使用宽读取，请将 fetch 语句包括在代码中，如下所示：

```
EXEC SQL FETCH ... ARRAY nnn
```

其中 ARRAY *nnn* 是 FETCH 语句的最后一项。读取计数 *nnn* 可以是一个主机变量。SQLDA 中的变量数必须是 *nnn* 和每行的列数的乘积。第一行放在 SQLDA 变量 0 和（每行的列数）-1 之间，依此类推。

SQLDA 的每一行中的每一列的类型必须相同，否则会返回 SQLDA_INCONSISTENT 错误。

服务器在 SQLCOUNT 中返回读取的记录数，除非有错误或警告，否则该记录数始终大于零。在宽读取时，在没有错误的情况下，SQLCOUNT 为一 (1) 指示已经读取一个有效行。

示例

下面的示例代码说明如何使用宽读取。也可在 *samples-dir\SQLAnywhere\esqlwidefetch\widefetch.sqc* 中找到此代码。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sqldef.h"
EXEC SQL INCLUDE SQLCA;

EXEC SQL WHENEVER SQLERROR { PrintSQLError();
        goto err; };

static void PrintSQLError()
{
  char buffer[200];
```

```
  printf( "SQL error %d -- %s\n",
     SQLCODE,
     sqlerror_message( &sqlca,
              buffer,
              sizeof( buffer ) ) );
}
static SQLDA * PrepareSQLDA(
  a_sql_statement_number stat0,
  unsigned  width,
  unsigned  *cols_per_row )

/* Allocate a SQLDA to be used for fetching from
   the statement identified by "stat0". "width"
   rows are retrieved on each FETCH request.
   The number of columns per row is assigned to
   "cols_per_row". */
{
  int           num_cols;
  unsigned      row, col, offset;
  SQLDA *       sqlda;
  EXEC SQL BEGIN DECLARE SECTION;
  a_sql_statement_number  stat;
  EXEC SQL END DECLARE SECTION;
  stat = stat0;
  sqlda = alloc_sqlda( 100 );
  if( sqlda == NULL ) return( NULL );
  EXEC SQL DESCRIBE :stat INTO sqlda;
  *cols_per_row = num_cols = sqlda->sqld;
  if( num_cols * width > sqlda->sqln )
  {
    free_sqlda( sqlda );
    sqlda = alloc_sqlda( num_cols * width );
    if( sqlda == NULL ) return( NULL );
    EXEC SQL DESCRIBE :stat INTO sqlda;
  }
  // copy first row in SQLDA setup by describe
  // to following (wide) rows
  sqlda->sqld = num_cols * width;
  offset = num_cols;
  for( row = 1; row < width; row++ )
  {
    for( col = 0;
      col < num_cols;
      col++, offset++ )
    {
      sqlda->sqlvar[offset].sqltype =
          sqlda->sqlvar[col].sqltype;
      sqlda->sqlvar[offset].sqllen =
          sqlda->sqlvar[col].sqllen;
      // optional: copy described column name
      memcpy( &sqlda->sqlvar[offset].sqlname,
              &sqlda->sqlvar[col].sqlname,
              sizeof( sqlda->sqlvar[0].sqlname ) );
    }
  }
  fill_s_sqlda( sqlda, 40 );
  return( sqlda );
err:
  return( NULL );
}
static void PrintFetchedRows(
  SQLDA * sqlda,
  unsigned cols_per_row )
```



```
{
  /* Print rows already wide fetched in the SQLDA */
  long      rows_fetched;
  int       row, col, offset;

  if( SQLCOUNT == 0 )
  {
    rows_fetched = 1;
  }
  else
  {
    rows_fetched = SQLCOUNT;
  }
  printf( "Fetched %d Rows:\n", rows_fetched );
  for( row = 0; row < rows_fetched; row++ )
  {
    for( col = 0; col < cols_per_row; col++ )
    {
      offset = row * cols_per_row + col;
      printf( " \"%s\"",
        (char *)sqlda->sqlvar[offset].sqldata );
     }
     printf( "\n" );
  }
}
static int DoQuery(
  char * query_str0,
  unsigned fetch_width0 )
{
  /* Wide Fetch "query_str0" select statement
   * using a width of "fetch_width0" rows" */
  SQLDA *                 sqlda;
  unsigned                cols_per_row;
  EXEC SQL BEGIN DECLARE SECTION;
  a_sql_statement_number  stat;
  char *                  query_str;
  unsigned                fetch_width;
  EXEC SQL END DECLARE SECTION;

  query_str = query_str0;
  fetch_width = fetch_width0;

  EXEC SQL PREPARE :stat FROM :query_str;
  EXEC SQL DECLARE QCURSOR CURSOR FOR :stat
       FOR READ ONLY;
  EXEC SQL OPEN QCURSOR;
  sqlda = PrepareSQLDA( stat,
    fetch_width,
    &cols_per_row );
  if( sqlda == NULL )
  {
    printf( "Error allocating SQLDA\n" );
    return( SQLE_NO_MEMORY );
  }
  for( ;; )
  {
    EXEC SQL FETCH QCURSOR INTO DESCRIPTOR sqlda
      ARRAY :fetch_width;
    if( SQLCODE != SQLE_NOERROR ) break;
    PrintFetchedRows( sqlda, cols_per_row );
  }
  EXEC SQL CLOSE QCURSOR;
  EXEC SQL DROP STATEMENT :stat;
  free_filled_sqlda( sqlda );
```

```
err:
  return( SQLCODE );
}
void main( int argc, char *argv[] )
{
  /* Optional first argument is a select statement,
   * optional second argument is the fetch width */
  char *query_str =
    "SELECT GivenName, Surname FROM Employees";
  unsigned fetch_width = 10;

  if( argc > 1 )
  {
    query_str = argv[1];
    if( argc > 2 )
    {
      fetch_width = atoi( argv[2] );
      if( fetch_width < 2 )
      {
        fetch_width = 2;
      }
    }
  }
  db_init( &sqlca );
  EXEC SQL CONNECT "DBA" IDENTIFIED BY "sql";

  DoQuery( query_str, fetch_width );

  EXEC SQL DISCONNECT;
err:
  db_fini( &sqlca );
}
```

有关使用宽读取的注意事项

- 在函数 PrepareSQLDA 中，SQLDA 内存是使用 alloc_sqlda 函数分配的。这样就为指示符变量留出了空间，而不用使用 alloc_sqlda_noind 函数。
- 如果读取的行数小于请求的行数，但又不是零（例如在游标的末尾），则通过设置指示符值可将对应于未读取的行的 SQLDA 项作为 NULL 返回。如果没有指示符变量，将生成错误（SQLE_NO_INDICATOR：未给 NULL 结果提供指示符变量）。
- 如果正在读取的行已经更新，并且生成了 SQLE_ROW_UPDATED_WARNING 警告，那么，读取到导致警告的行时就会停止。将返回处理到该点的所有行（包括导致警告的行）的值。SQLCOUNT 包含读取的行数，其中包括导致警告的行。所有剩余的 SQLDA 项都被标记为 NULL。
- 如果正在读取的行已经被删除或锁定，并且生成了 SQLE_NO_CURRENT_ROW 或 SQLE_LOCKED 错误，则 SQLCOUNT 包含出错前读取的行数。这不包括导致错误的行。SQLDA 不包含任何行的值，因为出现错误时不返回 SQLDA 值。如果必要，可使用 SQLCOUNT 值来重新定位游标，以便读取行。

发送和检索 Long 型值

在嵌入式 SQL 应用程序中，发送和检索 LONG VARCHAR、LONG NVARCHAR 和 LONG BINARY 值的方法与发送和检索其它数据类型的值的方法不同。标准 SQLDA 字段包含的数据不得超过 32767 字节，因为保存长度信息的字段（sqldata、sqllen 和 sqlind）是 16 位的值。将这些值更改为 32 位值会破坏现有的应用程序。

说明 LONG VARCHAR、LONG NVARCHAR 和 LONG BINARY 值的方法与说明其它数据类型的值的方法相同。

有关如何检索和发送值的信息，请参见“检索 LONG 数据”一节第 556 页和“发送 LONG 数据”一节第 557 页。

静态 SQL 结构

使用单独的字段来保存 LONG BINARY、LONG VARCHAR 和 LONG NVARCHAR 数据类型的已分配长度、已存储长度和未截断长度。静态 SQL 数据类型在 *sqlca.h* 中定义如下：

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

动态 SQL 结构

对于动态 SQL，根据需要将 sqltype 字段设置为 DT_LONGVARCHAR、DT_LONGNVARCHAR 或 DT_LONGBINARY。关联的 LONGVARCHAR、LONGNVARCHAR 和 LONGBINARY 结构如下：

```
typedef struct LONGVARCHAR {
    a_sql_uint32    array_len;
    a_sql_uint32    stored_len;
    a_sql_uint32    untrunc_len;
    char            array[1];
} LONGVARCHAR, LONGNVARCHAR, LONGBINARY;
```

结构成员定义

对于静态和动态 SQL 结构，结构成员定义如下：

- **array_len** （发送和检索。）为结构中数组部分分配的字节数。

- **stored_len** （发送和检索。）数组中存储的字节数。总是小于或等于 array_len 和 untrunc_len。
- **untrunc_len** （仅检索。）值不被截断的情况下数组中将存储的字节数。总是大于或等于 stored_len。如果发生截断，则此值大于 array_len。

检索 LONG 数据

本节介绍如何从数据库检索 LONG 值。有关背景信息，请参见“发送和检索 Long 型值”一节 第 555 页。

使用静态 SQL 时与使用动态 SQL 时的检索过程不同。

♦ 接收 LONG VARCHA、LONG NVARCHAR 或 LONG BINARY 值（静态 SQL）

1. 根据需要声明类型为 DECL_LONGVARCHAR、DECL_LONGNVARCHAR 或 DECL_LONGBINARY 的主机变量。array_len 成员将自动填充。
2. 使用 FETCH、GET DATA 或 EXECUTE INTO 检索数据。SQL Anywhere 设置以下信息：
   - **指示符变量** 指示符变量在值为 NULL 时为负，在未发生截断时为 0，在发生截断时为未截断值的字节数（不超过 32767 的正数）。

     有关详细信息，请参见“指示符变量”一节第 529 页。
   - **stored_len** 数组中存储的字节数。总是小于或等于 array_len 和 untrunc_len。
   - **untrunc_len** 值不被截断的情况下为数组中将存储的字节数。总是大于或等于 stored_len。如果发生截断，则此值大于 array_len。

♦ 将值接收到 LONGVARCHAR、LONGNVARCHAR 或 LONGBINARY 结构中（动态 SQL）

1. 根据需要将 sqltype 字段设置为 DT_LONGVARCHAR、DT_LONGNVARCHAR 或 DT_LONGBINARY。
2. 将 sqldata 字段设置为指向 LONGVARCHAR、LONGNVARCHAR 或 LONGBINARY 主机变量结构。

   可以使用 LONGVARCHARSIZE(*n*)、LONGNVARCHARSIZE(*n*) 或 LONGBINARYSIZE(*n*) 宏来确定在数组字段中容纳 *n* 字节的数据而要分配的总字节数。
3. 将主机变量结构的 array_len 字段设置为分配给数组字段的字节数。
4. 使用 FETCH、GET DATA 或 EXECUTE INTO 检索数据。SQL Anywhere 设置以下信息：
   - ***** **sqlind** sqlda 字段在值为 NULL 时为负，在未发生截断时为 0，在发生截断时为未截断值的字节数（不超过 32767 的正数）。
   - **stored_len** 数组中存储的字节数。总是小于或等于 array_len 和 untrunc_len。
   - **untrunc_len** 值不被截断的情况下数组中将存储的字节数。总是大于或等于 stored_len。如果发生截断，则此值大于 array_len。

下面的代码段阐释了使用动态嵌入式 SQL 来检索 LONG VARCHAR 数据的机制。它并不是为实际的应用程序而准备的：

```
#define DATA_LEN 128000
void get_test_var()
{
  LONGVARCHAR *longptr;
  SQLDA       *sqlda;
  SQLVAR      *sqlvar;

  sqlda = alloc_sqlda( 1 );
  longptr = (LONGVARCHAR *)malloc(
              LONGVARCHARSIZE( DATA_LEN ) );
  if( sqlda == NULL || longptr == NULL )
  {
    fatal_error( "Allocation failed" );
  }

  // init longptr for receiving data
  longptr->array_len = DATA_LEN;

  // init sqlda for receiving data
  // (sqllen is unused with DT_LONG types)
  sqlda->sqld = 1; // using 1 sqlvar
  sqlvar = &sqlda->sqlvar[0];
  sqlvar->sqltype = DT_LONGVARCHAR;
  sqlvar->sqldata = longptr;
  printf( "fetching test_var\n" );
  EXEC SQL PREPARE select_stmt FROM 'SELECT test_var';
  EXEC SQL EXECUTE select_stmt INTO DESCRIPTOR sqlda;
  EXEC SQL DROP STATEMENT select_stmt;
  printf( "stored_len: %d, untrunc_len: %d, "
          "1st char: %c, last char: %c\n",
        longptr->stored_len,
        longptr->untrunc_len,
        longptr->array[0],
        longptr->array[DATA_LEN-1] );
  free_sqlda( sqlda );
  free( longptr );
}
```

发送 LONG 数据

本节说明如何将 LONG 值从嵌入式 SQL 应用程序发送到数据库。有关背景信息，请参见“发送和检索 Long 型值”一节第 555 页。

使用静态 SQL 时与使用动态 SQL 时的检索过程不同。

♦ 发送 LONG 值（静态 SQL）

1. 根据需要声明类型为 DECL_LONGVARCHAR、DECL_LONGNVARCHAR 或 DECL_LONGBINARY 的主机变量。
2. 如果您要发送 NULL，请将指示符变量设置为负值。

   有关详细信息，请参见“指示符变量”一节第 529 页。
3. 将主机变量结构的 stored_len 字段设置为数组字段中数据的字节数。

4. 通过打开游标或执行语句发送数据。

下面的代码段说明使用静态嵌入式 SQL 发送 LONG VARCHAR 的机制。它并不是为实际的应用程序而准备的。

```
#define DATA_LEN 12800
EXEC SQL BEGIN DECLARE SECTION;
// SQLPP initializes longdata.array_len
DECL_LONGVARCHAR(128000) longdata;
EXEC SQL END DECLARE SECTION;

void set_test_var()
{
  // init longdata for sending data
  memset( longdata.array, 'a', DATA_LEN );
  longdata.stored_len = DATA_LEN;

  printf( "Setting test_var to %d a's\n", DATA_LEN );
  EXEC SQL SET test_var = :longdata;
}
```

♦ 发送 LONG 值（动态 SQL）

1. 根据需要将 sqltype 字段设置为 DT_LONGVARCHAR、DT_LONGNVARCHAR 或 DT_LONGBINARY。
2. 如果您要发送 NULL，请将 *** sqlind** 设置为负值。
3. 如果不是要发送 NULL，将 sqldata 字段设置为指向 LONGVARCHAR、LONGNVARCHAR 或 LONGBINARY 主机变量结构。

   可以使用 LONGVARCHARSIZE(*n*)、LONGNVARCHARSIZE(*n*) 或 LONGBINARYSIZE(*n*) 宏来确定在数组字段中容纳 *n* 字节的数据而要分配的总字节数。
4. 将主机变量结构的 array_len 字段设置为分配给数组字段的字节数。
5. 将主机变量结构的 stored_len 字段设置为数组字段中数据的字节数。它一定不能大于 array_len。
6. 通过打开游标或执行语句发送数据。

使用简单的存储过程

您可以在嵌入式 SQL 中创建和调用存储过程。

您可以像嵌入任何其它数据定义语句（如 CREATE TABLE）那样嵌入 CREATE PROCEDURE。您还可以嵌入 CALL 语句以执行存储过程。下面的代码段说明如何在嵌入式 SQL 中创建和执行存储过程：

```
EXEC SQL CREATE PROCEDURE pettycash(
 IN Amount DECIMAL(10,2) )
BEGIN
 UPDATE account
 SET balance = balance - Amount
 WHERE name = 'bank';

 UPDATE account
 SET balance = balance + Amount
 WHERE name = 'pettycash expense';
END;
EXEC SQL CALL pettycash( 10.72 );
```

如果想要将主机变量值传递到存储过程，或检索输出变量，请准备并执行 CALL 语句。下面的代码段说明主机变量的用法。USING 和 INTO 子句都在 EXECUTE 语句中使用。

```
EXEC SQL BEGIN DECLARE SECTION;
double  hv_expense;
double  hv_balance;
EXEC SQL END DECLARE SECTION;

// Code here
EXEC SQL CREATE PROCEDURE pettycash(
   IN expense  DECIMAL(10,2),
   OUT endbalance DECIMAL(10,2) )
BEGIN
  UPDATE account
  SET balance = balance - expense
  WHERE name = 'bank';
  UPDATE account
  SET balance = balance + expense
  WHERE name = 'pettycash expense';

  SET endbalance = ( SELECT balance FROM account
         WHERE name = 'bank' );
END;

EXEC SQL PREPARE S1 FROM 'CALL pettycash( ?, ? )';
EXEC SQL EXECUTE S1 USING :hv_expense INTO :hv_balance;
```

有关详细信息，请参见“EXECUTE 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》和“PREPARE 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

具有结果集的存储过程

数据库过程也可以包含 SELECT 语句。声明该过程的方法是这样的：使用 RESULT 子句来指定结果集中列的编号、名称和类型。结果集列与输出参数不同。对于具有结果集的过程，在游标声明中可以使用 CALL 语句代替 SELECT 语句：

```
EXEC SQL BEGIN DECLARE SECTION;
 char hv_name[100];
EXEC SQL END DECLARE SECTION;

EXEC SQL CREATE PROCEDURE female_employees()
  RESULT( name char(50) )
BEGIN
  SELECT GivenName || Surname FROM Employees
  WHERE Sex = 'f';
END;

EXEC SQL PREPARE S1 FROM 'CALL female_employees()';

EXEC SQL DECLARE C1 CURSOR FOR S1;
EXEC SQL OPEN C1;
for(;;)
{
  EXEC SQL FETCH C1 INTO :hv_name;
  if( SQLCODE != SQLE_NOERROR ) break;
  printf( "%s\\n", hv_name );
}
EXEC SQL CLOSE C1;
```

在本示例中，使用 OPEN 语句而不是 EXECUTE 语句调用该过程。OPEN 语句会使该过程在到达 SELECT 语句之前一直执行。此时，C1 是数据库过程内的 SELECT 语句的游标。您可以使用所有形式的 FETCH 语句（向后和向前滚动），直到完成它为止。CLOSE 语句用于终止该过程的执行。

如果在该过程中在 SELECT 语句之后还有一条语句，则不会执行该语句。要执行 SELECT 之后的语句，请使用 RESUME cursor-name 语句。RESUME 语句可返回警告 SQLE_PROCEDURE_COMPLETE，或返回指示存在另一游标的 SQLE_NOERROR。下面的示例阐释了一个双选择过程：

```
EXEC SQL CREATE PROCEDURE people()
  RESULT( name char(50) )
BEGIN
 SELECT GivenName || Surname
 FROM Employees;

 SELECT GivenName || Surname
 FROM Customers;
END;

EXEC SQL PREPARE S1 FROM 'CALL people()';
EXEC SQL DECLARE C1 CURSOR FOR S1;
EXEC SQL OPEN C1;
while( SQLCODE == SQLE_NOERROR )
{
  for(;;)
  {
    EXEC SQL FETCH C1 INTO :hv_name;
    if( SQLCODE != SQLE_NOERROR ) break;
    printf( "%s\\n", hv_name );
  }
  EXEC SQL RESUME C1;
}
EXEC SQL CLOSE C1;
```

CALL 语句的动态游标

以上这些示例使用了静态游标。也可以将完全动态的游标用于 CALL 语句。

有关动态游标的说明，请参见“动态 SELECT 语句”一节第 539 页。

DESCRIBE 语句完全适用于过程调用。DESCRIBE OUTPUT 生成的 SQLDA 具有每个结果集列的说明。

如果过程没有结果集，则 SQLDA 具有过程的每个 INOUT 或 OUT 参数的说明。DESCRIBE INPUT 语句生成的 SQLDA 具有过程的每个 IN 或 INOUT 参数的说明。

DESCRIBE ALL

DESCRIBE ALL 说明 IN、INOUT、OUT 和 RESULT 等集合参数。DESCRIBE ALL 使用 SQLDA 中的指示符变量提供其它信息。

当描述 CALL 语句时，会在指示符变量中设置 DT_PROCEDURE_IN 和 DT_PROCEDURE_OUT 位。DT_PROCEDURE_IN 指示 IN 或 INOUT 参数，而 DT_PROCEDURE_OUT 则指示 INOUT 或 OUT 参数。过程的 RESULT 列则清除这两个位。

在说明 OUTPUT 之后，可以使用这些位来区分具有结果集的语句（需要使用 OPEN、FETCH、RESUME 和 CLOSE）和不具有结果集的语句（需要使用 EXECUTE）。

有关完整说明，请参见“DESCRIBE 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

多个结果集

如果具有一个返回多个结果集的过程，则在结果集改变形状时必须在每条 RESUME 语句之后重新说明。

您需要说明游标（而不是语句），以重新说明游标的当前位置。

嵌入式 SQL 编程技巧

本节包含了一组可供嵌入式 SQL 程序开发人员使用的技巧。

实现请求管理

接口 DLL 的缺省行为是让应用程序在执行其它函数之前等待每个数据库请求的完成。可以使用请求管理函数更改这一行为。例如，在使用 Interactive SQL 时，操作系统在 Interactive SQL 等待数据库响应时仍处于活动状态，这时 Interactive SQL 可以执行一些其它任务。

通过提供回调函数，您可以在数据库请求正在进行时完成应用程序的活动。在此回调函数中，您不能发出除 db_cancel_request 外的其它数据库请求。您可以在消息处理程序中使用 db_is_working 函数来确定是否有正在进行的数据库请求。

db_register_a_callback 函数用于注册您的应用程序回调函数。

另请参见

- “db_register_a_callback 函数”一节第 576 页
- “db_cancel_request 函数”一节第 570 页
- “db_is_working 函数”一节第 574 页

备份函数

db_backup 函数在嵌入式 SQL 应用程序中提供对联机备份的支持。备份实用程序就利用了此函数。只有在 SQL Anywhere 备份实用程序无法满足您的备份要求时，才需要编写程序来使用此函数。

建议使用 BACKUP 语句
尽管此函数提供了一种向应用程序添加备份功能的方法，但建议您使用 BACKUP 语句来完成此任务。请参见“BACKUP 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

您也可以使用数据库工具 DBBackup 函数直接访问备份实用程序。请参见“DBBackup 函数”一节第 915 页。

另请参见

- “db_backup 函数”一节第 566 页

SQL 预处理器

SQL 预处理器会在编译器运行之前处理包含嵌入式 SQL 的 C 或 C++ 程序。

语法

sqlpp [*options*] *input-file* [*output-file*]

选项	说明
-d	生成减小数据空间大小的代码。数据结构在使用之前执行时会得到重用和初始化。这会增大代码的大小。
-e *level*	将任何未包含在指定标准中的静态嵌入式 SQL 标记为错误。*level* 值表示要使用的标准。例如，`sqlpp -e c03 ...` 标记任何未包含在核心 SQL/2003 标准中的语法。所支持的 *level* 值为： ● **c03** 标记不是核心 SQL/2003 语法的语法 ● **p03** 标记非完整 SQL/2003 语法的语法 ● **c99** 标记不是核心 SQL/1999 语法的语法 ● **p99** 标记非完整 SQL/1999 语法的语法 ● **e92** 标记非入门级 SQL/1992 语法的语法 ● **i92** 标记非中级 SQL/1992 语法的语法 ● **f92** 标记非完整 SQL/1992 语法的语法 ● **t** 标记非标准主机变量类型 ● **u** 标记 UltraLite 不支持的语法 为了与以前的 SQL Anywhere 版本兼容，也可指定 e、i 和 f（分别对应 e92、i92 和 f92）。
-h *width*	将 sqlpp 输出的最大行长度限制为 *width*。续行符是反斜杠 (\)，而 *width* 的最小值是 10。
-k	通知预处理器要编译的程序包括 SQLCODE 的用户声明。定义必须是 LONG 类型，但不必在声明部分内。
-n	在 C 文件中生成行号信息。该信息包括生成的 C 代码中适当位置处的 *#line* 指令。如果您使用的编译器支持 *#line* 指令，使用此选项可使编译器按照 SQC 文件（其中带有嵌入式 SQL）的行号报错，而不是用 SQL 预处理器生成的 C 文件的行号报错。此外，*#line* 指令由源代码级调试程序间接使用，以便您可以在查看 SQC 源文件时进行调试。

选项	说明
-o *操作系统*	指定目标操作系统。支持的操作系统有： ● **WINDOWS** Microsoft Windows ● **UNIX** 当创建 32 位 Unix 应用程序时可使用此选项。 ● **UNIX64** 当创建 64 位 Unix 应用程序时可使用此选项。
-q	安静模式—不显示消息。
-r-	生成非重入代码。有关重入代码的详细信息，请参见“多线程代码或重入代码的 SQLCA 管理”一节第 534 页。
-s *len*	设置预处理器放入 C 文件的最大大小的字符串。对于长度大于此值的字符串，将使用一组字符（*'a'*、*'b'*、*'c'* 等）对其进行初始化。大多数 C 编译器都对可以处理的字符串大小有限制。此选项用于设置其上限。缺省值是 500。
-u	为 UltraLite 生成代码。有关详细信息，请参见“嵌入式 SQL API 参考” 《UltraLite - C 及 C++ 编程》。
-w *level*	将任何未包含在指定标准中的静态嵌入式 SQL 标记为警告。*level* 值表示要使用的标准。例如，`sqlpp -w c03 ...` 标记任何未包含在核心 SQL/2003 语法中的 SQL 语法。所支持的 *level* 值为： ● **c03** 标记不是核心 SQL/2003 语法的语法 ● **p03** 标记非完整 SQL/2003 语法的语法 ● **c99** 标记不是核心 SQL/1999 语法的语法 ● **p99** 标记非完整 SQL/1999 语法的语法 ● **e92** 标记非入门级 SQL/1992 语法的语法 ● **i92** 标记非中级 SQL/1992 语法的语法 ● **f92** 标记非完整 SQL/1992 语法的语法 ● **t** 标记非标准主机变量类型 ● **u** 标记 UltraLite 不支持的语法 为了与以前的 SQL Anywhere 版本兼容，也可指定 e、i 和 f（分别对应 e92、i92 和 f92）。
-x	将多字节字符串更改为转义序列，以便它们可以通过编译器。

选项	说明
-z *cs*	指定归类序列。要查看建议使用的归类序列的列表，请在命令提示符处输入 dbinit -l。 归类序列用于帮助预处理器理解在程序源代码中使用的字符，例如，识别出适合在标识符中使用的字母字符。如果未指定 -z，则预处理器将尝试根据操作系统和 SALANG 与 SACHARSETE 环境变量确定要使用的合理归类。请参见“SACHARSET 环境变量”一节 《SQL Anywhere 服务器 - 数据库管理》和“SALANG 环境变量”一节 《SQL Anywhere 服务器 - 数据库管理》。
input-file	要处理的包含嵌入式 SQL 的 C 或 C++ 程序。
output-file	由 SQL 预处理器创建的 C 语言源文件。

说明

SQL 预处理器将 *input-file* 中的 SQL 语句转换为 C 语言源代码并将源代码放入 *output-file* 中。含有嵌入式 SQL 的源程序的扩展名通常为 *.sqc*。缺省的输出文件名为 *input-file*，扩展名为 *.c*。如果 *input-file* 的扩展名为 *.c*，则缺省的输出文件扩展名是 *.cc*。

另请参见

- “嵌入式 SQL 简介”一节第 510 页
- “sql_flagger_error_level 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》
- “sql_flagger_warning_level 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》
- “SQLFLAGGER 函数 [Miscellaneous]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “sa_ansi_standard_packages 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “SQL 预处理器错误消息” 《错误消息》

库函数参考

SQL 预处理器生成对接口库或 DLL 中的函数的调用。除了 SQL 预处理器生成的调用外，还提供了一组库函数方便数据库操作。EXEC SQL INCLUDE SQLCA 语句包括这些函数的原型。

本节包含这些不同函数的参考说明。

DLL 入口点

除了原型具有适合于 DLL 的修改程序外，DLL 入口点都是相同的。

可以使用 *sqlca.h* 中定义的 **_esqlentry_** 以可移植的方式声明入口点。它解析为值 __stdcall。

alloc_sqlda 函数

原型

struct sqlda * **alloc_sqlda(** unsigned *numvar* **);**

说明

分配一个具有 *numvar* 个变量描述符的 SQLDA。将该 SQLDA 的 sqln 字段初始化为 *numvar*。为指示符变量分配空间，将指示符变量设置为指向此空间，并将指示符值初始化为零。如果无法分配内存，则返回空指针。建议您使用此函数代替 alloc_sqlda_noind 函数。

alloc_sqlda_noind 函数

原型

struct sqlda * **alloc_sqlda_noind(** unsigned *numvar* **);**

说明

分配一个具有 *numvar* 个变量描述符的 SQLDA。将该 SQLDA 的 sqln 字段初始化为 *numvar*。不为指示符变量分配空间；将指示符指针设置为空指针。如果无法分配内存，则返回空指针。

db_backup 函数

原型

void **db_backup(**
SQLCA * *sqlca*,
int *op*,
int file_num,
unsigned long *page_num*,
struct sqlda * *sqlda*);

授权

必须以具有 DBA 权限、REMOTE DBA 权限 (SQL Remote) 或 BACKUP 权限的用户身份进行连接。

说明

建议使用 BACKUP 语句
尽管此函数提供了一种向应用程序添加备份功能的方法，但建议您使用 BACKUP 语句来完成此任务。请参见“BACKUP 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

执行的操作取决于 *op* 参数的值：

- **DB_BACKUP_START** 必须先调用此函数，然后才能开始备份。对于任何给定的数据库服务器，一个数据库一次只能运行一个备份。在备份完成之前禁用数据库检查点（直到使用 *op* 值 DB_BACKUP_END 调用 db_backup）。如果备份无法启动，则 SQLCODE 为 SQLE_BACKUP_NOT_STARTED。否则，将 *sqlca* 的 SQLCOUNT 字段设置为数据库页的大小。一次一页地对备份进行处理。

  忽略 *file_num*、*page_num* 和 *sqlda* 参数。

- **DB_BACKUP_OPEN_FILE** 打开由 *file_num* 指定的数据库文件，这允许使用 DB_BACKUP_READ_PAGE 备份指定文件的页。根数据库文件、事务日志文件的有效文件编号分别为：0 到 DB_BACKUP_MAX_FILE 和 0 到 DB_BACKUP_TRANS_LOG_FILE。如果指定的文件不存在，则 SQLCODE 为 SQLE_NOTFOUND。否则，SQLCOUNT 包含文件中的页数，SQLIOESTIMATE 包含一个标识数据库文件创建时间的 32 位值 (POSIX time_t)，操作系统文件名位于 SQLCA 的 *sqlerrmc* 字段中。

  忽略 *page_num* 和 *sqlda* 参数。

- **DB_BACKUP_READ_PAGE** 读取由 *file_num* 指定的数据库文件的一页。使用 DB_BACKUP_OPEN_FILE 操作对 db_backup 调用成功后会在 SQLCOUNT 中返回一个页数，*page_num* 值应介于 0 到此页数减去一得到的数之间。否则，SQLCODE 将设置为 SQLE_NOTFOUND。*sqlda* 描述符应使用一个指向缓冲区的 DT_BINARY 或 DT_LONG_BINARY 类型的变量建立。使用 DB_BACKUP_START 操作调用 db_backup 时会在 SQLCOUNT 字段中返回一个大小值，该缓冲区应足以保存这一大小的二进制数据。

  DT_BINARY 数据包含一个后跟实际二进制数据的两字节长度值，因此缓冲区必须比页大小大两个字节。

  应用程序必须保存缓冲区
  此调用会在缓冲区中制作指定数据库页的一个副本，但应由应用程序将缓冲区保存到某种备份介质。

- **DB_BACKUP_READ_RENAME_LOG** 此操作与 DB_BACKUP_READ_PAGE 只有一点不同：事务日志的最后一页返回之后，数据库服务器会重命名事务日志并启动一个新的事务日志。

  如果数据库服务器无法在当前时间重命名日志（例如，在 7.0.x 版或更早版本的数据库中可能存在未完成的事务），则会设置 SQLE_BACKUP_CANNOT_RENAME_LOG_YET 错误。在这种情况下，不要使用返回的页，而是要重新发出请求，直到收到 SQLE_NOERROR，然后写入页。继续读取页，直到收到 SQLE_NOTFOUND 条件。

可能会在多个页上多次返回 SQLE_BACKUP_CANNOT_RENAME_LOG_YET 错误。您应该在重试循环中增加延迟，以便不会因请求过多而降低服务器的速度。

当您收到 SQLE_NOTFOUND 条件时，事务日志已经成功备份且文件已经重命名。旧日志文件的名称在 SQLCA 的 *sqlerrmc* 字段中返回。

您应在 db_backup 调用后检查 sqlda->sqlvar[0].sqlind 的值。如果此值大于零，则最后一个日志页已经写入且日志文件已经重命名。新名称仍然在 sqlca.sqlerrmc 中，但 SQLCODE 值是 SQLE_NOERROR。

此后，您不应再次调用 db_backup（除非要关闭文件并完成备份），否则，您会获得备份日志文件的第二份副本并收到 SQLE_NOTFOUND。

- **DB_BACKUP_CLOSE_FILE** 处理完一个文件后必须调用此函数以关闭由 *file_num* 指定的数据库文件。

  忽略 *page_num* 和 *sqlda* 参数。

- **DB_BACKUP_END** 备份结束时必须调用此函数。在此备份结束之前，任何其它备份都无法启动。会再次启用检查点。

  忽略 *file_num*、*page_num* 和 *sqlda* 参数。

- **DB_BACKUP_PARALLEL_START** 开始并行备份。与 DB_BACKUP_START 类似，对于任何给定的数据库服务器，一个数据库一次只能运行一个备份。在备份完成之前禁用数据库检查点（直到使用 *op* 值 DB_BACKUP_END 调用 db_backup）。如果备份无法启动，则将收到 SQLE_BACKUP_NOT_STARTED。否则，将 sqlca 的 SQLCOUNT 字段设置为数据库页的大小。

  file_num 参数指示数据库服务器在事务日志的最后一页返回之后重命名事务日志并启动一个新的日志。如果此值非零，则将重命名或重新启动事务日志。否则，不会重命名或重新启动事务日志。此参数消除了对 DB_BACKUP_READ_RENAME_LOG 操作的需要，并行备份操作期间不允许进行该操作。

  page_num 参数通知数据库服务器客户端缓冲区的最大大小（以数据库页为单位）。在服务器端，并行备份读取程序会尝试读取连续页块—此值可告知服务器应为这些块分配多大的内存空间：传递 *N* 值可告知服务器：客户端一次可以从服务器接收最多 *N* 个数据库页。如果服务器无法为 *N* 页块分配足够的内存，服务器可能会返回小于 *N* 的页块。如果客户端在调用 DB_BACKUP_PARALLEL_START 之前不清楚数据库页的大小，则可通过 DB_BACKUP_INFO 操作将此值提供给服务器。必须在首次调用前提供此值，以便检索备份页 (DB_BACKUP_PARALLEL_READ)。

  注意
  如果使用 db_backup 开始并行备份，db_backup 不会创建写入程序线程。db_backup 的调用程序必须接收数据并充当写入程序。

- **DB_BACKUP_INFO** 此参数为数据库服务器提供关于并行备份的其它信息。*file_num* 参数指示所提供信息的类型，*page_num* 参数提供值。您可以使用 DB_BACKUP_INFO 指定以下其它信息：

  - **DB_BACKUP_INFO_PAGES_IN_BLOCK** *page_num* 参数包含应在同一块内发回的最大页数。

- **DB_BACKUP_INFO_CHKPT_LOG** 这是相当于 BACKUP 语句的 WITH CHECKPOINT LOG 选项的客户端。*page_num* 的值 DB_BACKUP_CHKPT_COPY 指示 COPY，而值 DB_BACKUP_CHKPT_NOCOPY 指示 NO COPY。如果未提供此值，则缺省为 COPY。

- **DB_BACKUP_PARALLEL_READ** 此操作从数据库服务器上读取页块。调用此操作前，使用 DB_BACKUP_OPEN_FILE 操作打开所有要备份的文件。DB_BACKUP_PARALLEL_READ 会忽略 *file_num* 和 *page_num* 参数。

  sqlda 描述符应使用一个指向缓冲区的 DT_LONGBINARY 类型的变量建立。缓冲区大小应足以保存 *N* 页（在 DB_BACKUP_START_PARALLEL 操作或 DB_BACKUP_INFO 操作中指定）大小的二进制数据。有关此数据类型的详细信息，请参见“嵌入式 SQL 数据类型”一节第 520 页中的 DT_LONGBINARY。

  服务器为特定数据库文件返回连续数据库页块。块内第一页的页码在 SQLCOUNT 字段中返回。页所属的文件号在 SQLIOESTIMATE 字段中返回，此值与在 DB_BACKUP_OPEN_FILE 调用中使用的文件号之一相匹配。返回的数据大小存储在 DT_LONGBINARY 变量的 *stored_len* 字段中，并且总是数据库页面大小的倍数。当此调用所返回的数据内包含某个给定文件的连续页块时，单独的数据块不一定会按顺序返回，某一数据库文件的所有页也不一定会在另一数据库文件的页之前全部返回。调用程序应该准备接收不按顺序的另一单个文件的一部分或在任何给定调用中打开的任何数据库文件的一部分。

  应用程序应该重复调用此操作，直到读取的数据大小为 0 或者 sqlda->sqlvar[0].sqlind 的值大于 0。如果备份启动时重命名/重新启动了事务日志，则 SQLERROR 可能设置为 SQLE_BACKUP_CANNOT_RENAME_LOG_YET。在这种情况下，不要使用返回的页，而是要重新发出请求，直到收到 SQLE_NOERROR，然后写入数据。可能会在多个页上多次返回 SQLE_BACKUP_CANNOT_RENAME_LOG_YET 错误。您应该在重试循环中增加延迟，以便不会因请求过多而降低数据库服务器的速度。继续读取页，直到满足上述两个条件之一。

dbbackup 实用程序使用下面的算法。注意，这*不是* C 代码，不包括错误检查。

```
sqlda->sqld = 1;
sqlda->sqlvar[0].sqltype = DT_LONGBINARY

/* Allocate LONGBINARY value for page buffer. It MUST have */
/* enough room to hold the requested number (128) of database pages */
sqlda->sqlvar[0].sqldata = allocated buffer

/* Open the server files needing backup */
for file_num = 0 to DB_BACKUP_MAX_FILE
  db_backup( ... DB_BACKUP_OPEN_FILE, file_num ... )
  if SQLCODE == SQLE_NO_ERROR
    /* The file exists */
    num_pages = SQLCOUNT
    file_time = SQLE_IO_ESTIMATE
    open backup file with name from sqlca.sqlerrmc
end for

/* read pages from the server, write them locally */
while TRUE
  /* file_no and page_no are ignored */
  db_backup( &sqlca, DB_BACKUP_PARALLEL_READ, 0, 0, &sqlda );

  if SQLCODE != SQLE_NO_ERROR
     break;

  if buffer->stored_len == 0 || sqlda->sqlvar[0].sqlind > 0
```

```
        break;

    /* SQLCOUNT contains the starting page number of the block    */
    /* SQLIOESTIMATE contains the file number the pages belong to */
    write block of pages to appropriate backup file
  end while

  /* close the server backup files */
  for file_num = 0 to DB_BACKUP_MAX_FILE
      /* close backup file */
      db_backup( ... DB_BACKUP_CLOSE_FILE, file_num ... )
  end for

  /* shut down the backup */
  db_backup( ... DB_BACKUP_END ... )

  /* cleanup */
  free page buffer
```

db_cancel_request 函数

原型

int **db_cancel_request(** SQLCA * *sqlca* **);**

说明

取消当前活动的数据库服务器请求。此函数会进行检查，以确保在发送取消请求之前数据库服务器请求是活动的。如果该函数返回 1，则发送取消请求；如果它返回 0，则不发送请求。

一个非零返回值不表示请求被取消。在几个临界时刻，取消请求和来自数据库或服务器的响应会发生交错。在这些情况下，即使函数仍然返回 TRUE，取消请求也没有效果。

可以异步调用 db_cancel_request 函数。数据库接口库中只有此函数和 db_is_working 可以使用可能正被另一请求使用的 SQLCA 进行异步调用。

如果您取消正在执行游标操作的请求，则游标的位置是不确定的。在取消之后，您必须按游标的绝对位置定位该游标或关闭它。

db_change_char_charset 函数

原型

unsigned int **db_change_char_charset(**
SQLCA * *sqlca*,
char * charset **);**

说明

更改用于此连接的应用程序的 CHAR 字符集。使用 FIXCHAR、VARCHAR、LONGVARCHAR 和 STRING 类型发送和读取的数据采用 CHAR 字符集。

如果更改成功则返回 1，否则返回 0。

有关建议字符集的列表，请参见“建议的字符集和归类”一节 《SQL Anywhere 服务器 - 数据库管理》。

db_change_nchar_charset 函数

原型

unsigned int **db_change_nchar_charset(**
SQLCA * *sqlca*,
char * charset **);**

说明

更改用于此连接的应用程序的 NCHAR 字符集。使用 NFIXCHAR、NVARCHAR、LONGNVARCHAR 和 NSTRING 主机变量类型发送和读取的数据采用 NCHAR 字符集。

如果没有调用 db_change_nchar_charset 函数，将使用 CHAR 字符集发送和读取所有数据。通常，要发送和读取 Unicode 数据的应用程序应该将 NCHAR 字符集设置为 UTF-8。

如果调用此函数，charset 参数通常为 "UTF-8"。NCHAR 字符集不能设置为 UTF-16。

如果更改成功则返回 1，否则返回 0。

在嵌入式 SQL 中，缺省情况下将 NCHAR、NVARCHAR 和 LONG NVARCHAR 分别描述为 DT_FIXCHAR、DT_VARCHAR 和 DT_LONGVARCHAR。如果调用了 db_change_nchar_charset 函数，则这些类型被分别描述为 DT_NFIXCHAR、DT_NVARCHAR 和 DT_LONGNVARCHAR。

有关建议字符集的列表，请参见“建议的字符集和归类”一节 《SQL Anywhere 服务器 - 数据库管理》。

db_delete_file 函数

原型

void **db_delete_file(**
SQLCA * *sqlca*,
char * *filename* **);**

授权

必须连接到具有 DBA 权限或 REMOTE DBA 权限 (SQL Remote) 的用户 ID。

说明

db_delete_file 函数请求数据库服务器删除 *filename*。可以在备份和重命名事务日志之后使用它来删除旧事务日志。请参见“db_backup 函数”一节第 566 页中的 DB_BACKUP_READ_RENAME_LOG。

您必须连接到具有 DBA 权限的用户 ID。

db_find_engine 函数

原型

unsigned short **db_find_engine(**
SQLCA * *sqlca*,
char * *name* **);**

说明

返回无符号的短整型值，指示有关名为 *name* 的本地数据库服务器的状态信息。如果共享内存中找不到具有指定名称的服务器，则返回值为 0。非零值表示本地服务器当前正在运行。

如果为 *name* 指定了空指针，则返回有关缺省数据库服务器的信息。

返回值中的每个位都指示某一信息。代表不同信息段的位的常量在 *sqldef.h* 头文件中定义。其含义如下。

- **DB_ENGINE** 始终设置该标志。
- **DB_CLIENT** 始终设置该标志。
- **DB_CAN_MULTI_DB_NAME** 该标志已过时。
- **DB_DATABASE_SPECIFIED** 始终设置该标志。
- **DB_ACTIVE_CONNECTION** 始终设置该标志。
- **DB_CONNECTION_DIRTY** 该标志已过时。
- **DB_CAN_MULTI_CONNECT** 该标志已过时。
- **DB_NO_DATABASES** 如果服务器未启动数据库，则设置该标志。

db_fini 函数

原型

int **db_fini(** SQLCA * *sqlca* **);**

说明

此函数释放由数据库接口或 DLL 使用的资源。在调用 db_fini 之后不能进行任何其它库调用或执行任何嵌入式 SQL 语句。如果在处理过程中出现错误，则在 SQLCA 中设置错误代码且函数返回 0。如果没有错误，则返回非零值。

需要为每个使用的 SQLCA 调用一次 db_fini。

有关在 UltraLite 应用程序中使用 db_init 的信息，请参见“db_fini 函数”一节 《UltraLite - C 及 C++ 编程》。

db_get_property 函数

原型

unsigned int **db_get_property(**
SQLCA * *sqlca*,
a_db_property *property*,
char * *value_buffer*,
int *value_buffer_size* **);**

说明

此函数用于获得数据库接口或连接到的服务器的信息。

参数如下：

- **a_db_property** 请求的属性，DB_PROP_CLIENT_CHARSET、DB_PROP_SERVER_ADDRESS 或 DB_PROP_DBLIB_VERSION。
- **value_buffer** 此参数填充的是以空值终止的字符串形式的属性值。
- **value_buffer_size** 字符串 value_buffer 的最大长度，其中包括终止的空字符。

支持以下属性：

- **DB_PROP_CLIENT_CHARSET** 此属性值可获取客户端字符集（如 "windows-1252"）。
- **DB_PROP_SERVER_ADDRESS** 此属性值获取当前连接的服务器网络地址，作为可打印字符串。共享内存协议始终会为地址返回空字符串。TCP/IP 协议会返回非空字符串地址。
- **DB_PROP_DBLIB_VERSION** 此属性值可获取数据库接口库的版本（如 "11.0.0.1297"）。

如果成功则返回 1，否则返回 0。

db_init 函数

原型

int **db_init(** SQLCA * *sqlca* **);**

说明

此函数初始化数据库接口库。此函数必须在调用任何其它库之前或执行任何嵌入式 SQL 语句之前进行调用。在进行此调用时将分配和初始化接口库对您的程序所要求的资源。

如果成功则返回 1，否则返回 0。

在程序结尾处使用 db_fini 来释放资源。如果在处理过程中出现错误，则在 SQLCA 中返回这些错误且返回 0。如果没有错误，则返回非零值，您便可开始使用嵌入式 SQL 语句和函数。

多数情况下，只应调用一次此函数（传递 *sqlca.h* 头文件中定义的全局 sqlca 变量的地址）。如果您要使用嵌入式 SQL 来编写具有多个线程的 DLL 或应用程序，则每使用一个 SQLCA，就要调用一次 db_init。

有关详细信息，请参见“多线程代码或重入代码的 SQLCA 管理”一节第 534 页。

有关在 UltraLite 应用程序中使用 db_init 的信息，请参见“db_init 函数”一节 《UltraLite - C 及 C++ 编程》。

db_is_working 函数

原型

unsigned short **db_is_working(** SQLCA * *sqlca* **);**

说明

如果您的应用程序有一个使用了给定的 sqlca 的数据库请求正在进行中，则返回 1；若没有这样的请求在进行中，则返回 0。

可以异步调用此函数。数据库接口库中只有此函数和 db_cancel_request 可以使用 SQLCA（可能正被另一请求使用）进行异步调用。

db_locate_servers 函数

原型

unsigned int**db_locate_servers(**
SQLCA * *sqlca*,
SQL_CALLBACK_PARM *callback_address*,
void * *callback_user_data* **);**

说明

提供对 dblocate 实用程序所显示的信息的编程式访问，列出本地网络上正在监听 TCP/IP 的所有 SQL Anywhere 数据库服务器。

回调函数必须具有以下原型：

int (*)(SQLCA * *sqlca*,
a_server_address * *server_addr*,
void * *callback_user_data*);

对于找到的每台服务器都要调用回调函数。如果回调函数返回 0，则 db_locate_servers 停止遍历服务器。

传递到回调函数的 sqlca 和 callback_user_data 是那些传递到 db_locate_servers 中的函数。第二个参数是指向 a_server_address 结构的指针。a_server_address 在 *sqlca.h* 中定义，其定义如下：

```
typedef struct a_server_address {
    a_sql_uint32 port_type;
```



```
    a_sql_uint32 port_num;
    char         *name;
    char         *address;
} a_server_address;
```

- **port_type** 它在此时始终为 PORT_TYPE_TCP（在 *sqlca.h* 中定义为 6）。
- **port_num** 它是此服务器正在监听的 TCP 端口号。
- **name** 指向包含服务器名称的缓冲区。
- **address** 指向包含服务器 IP 地址的缓冲区。

如果成功则返回 1，否则返回 0。

另请参见

- “服务器枚举实用程序 (dblocate)”一节 《SQL Anywhere 服务器 - 数据库管理》

db_locate_servers_ex 函数

原型

unsigned int **db_locate_servers_ex(**
SQLCA * *sqlca*,
SQL_CALLBACK_PARM *callback_address*,
void * *callback_user_data*,
unsigned int *bitmask*);

说明

提供对 dblocate 实用程序所显示的信息的编程式访问，列出本地网络上正在监听 TCP/IP 的所有 SQL Anywhere 数据库服务器，但提供用于选择传递给回调函数的地址的掩码参数。

回调函数必须具有以下原型：

int (*)(SQLCA * *sqlca*,
a_server_address * *server_addr*,
void * *callback_user_data*);

对于找到的每台服务器都要调用回调函数。如果回调函数返回 0，则 db_locate_servers_ex 停止遍历服务器。

传递到回调函数的 sqlca 和 callback_user_data 是那些传递到 db_locate_servers 中的函数。第二个参数是指向 a_server_address 结构的指针。a_server_address 在 *sqlca.h* 中定义，其定义如下：

```
typedef struct a_server_address {
    a_sql_uint32   port_type;
    a_sql_uint32   port_num;
    char           *name;
    char           *address;
    char           *dbname;
} a_server_address;
```

- **port_type** 它在此时始终为 PORT_TYPE_TCP（在 *sqlca.h* 中定义为 6）。

- **port_num** 它是此服务器正在监听的 TCP 端口号。
- **name** 指向包含服务器名称的缓冲区。
- **address** 指向包含服务器 IP 地址的缓冲区。
- **dbname** 指向包含数据库名称的缓冲区。

支持三种位掩码标志：

- DB_LOOKUP_FLAG_NUMERIC
- DB_LOOKUP_FLAG_ADDRESS_INCLUDES_PORT
- DB_LOOKUP_FLAG_DATABASES

这些标志在 *sqlca.h* 中定义，可用 OR 连起来。

DB_LOOKUP_FLAG_NUMERIC 确保传递给回调函数的地址是 IP 地址，而不是主机名。

DB_LOOKUP_FLAG_ADDRESS_INCLUDES_PORT 指定传递给回调函数的地址包括 a_server_address 结构中的 TCP/IP 端口号。

DB_LOOKUP_FLAG_DATABASES 指定为找到的每个数据库调用一次回调函数，或者，如果数据库服务器不支持发送数据库信息（9.0.2 及更早版本的数据库服务器），指定为找到的每个数据库服务器调用一次回调函数。

如果成功则返回 1，否则返回 0。

有关详细信息，请参见“服务器枚举实用程序 (dblocate)”一节 《SQL Anywhere 服务器 - 数据库管理》。

db_register_a_callback 函数

原型

```
void db_register_a_callback(
SQLCA * sqlca,
a_db_callback_index index,
( SQL_CALLBACK_PARM ) callback );
```

说明

此函数注册回调函数。

如果您不注册 DB_CALLBACK_WAIT 回调，则缺省操作是不执行任何操作。您的应用程序块，等待数据库响应。必须为 MESSAGE TO CLIENT 语句注册一个回调。请参见“MESSAGE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

要删除回调，请传递一个空指针作为 *callback* 函数。

对于 *index* 参数，允许使用以下值：

- **DB_CALLBACK_DEBUG_MESSAGE** 对每条调试信息都要调用一次提供的函数，而且会有一个包含该调试信息文本的以空值终止的字符串传递给该函数。调试消息是记录到 LogFile 文

件的消息。为了将调试消息传递给此回调函数，必须使用 LogFile 连接参数。该字符串通常在紧邻终止的空字符之前有一个换行符 (\n)。回调函数的原型如下：

```
void SQL_CALLBACK debug_message_callback(
SQLCA * sqlca,
char * message_string );
```

有关详细信息，请参见“Logfile 连接参数 [LOG]”一节 《SQL Anywhere 服务器 - 数据库管理》。

- **DB_CALLBACK_START** 原型如下：

  ```
  void SQL_CALLBACK start_callback( SQLCA * sqlca );
  ```

  此函数在数据库请求发送到服务器之前进行调用。DB_CALLBACK_START 只在 Windows 上使用。

- **DB_CALLBACK_FINISH** 原型如下：

  ```
  void SQL_CALLBACK finish_callback( SQLCA * sqlca );
  ```

  此函数在接口 DLL 收到对数据库请求的响应之后进行调用。DB_CALLBACK_FINISH 仅在 Windows 操作系统上使用。

- **DB_CALLBACK_CONN_DROPPED** 原型如下：

  ```
  void SQL_CALLBACK conn_dropped_callback (
  SQLCA * sqlca,
  char * conn_name );
  ```

  此函数在数据库服务器要通过 DROP CONNECTION 语句删除连接（因活动超时或因该数据库服务器正在关闭）时进行调用。连接名称 *conn_name* 会传递给此函数，以便使您能够区分连接。如果没有命名连接，则它的值为 NULL。

- **DB_CALLBACK_WAIT** 原型如下：

  ```
  void SQL_CALLBACK wait_callback( SQLCA * sqlca );
  ```

  在数据库服务器或客户端库忙于处理您的数据库请求的同时，接口库反复调用此函数。

  您可以按如下所示注册此回调函数：

  ```
  db_register_a_callback( &sqlca,
     DB_CALLBACK_WAIT,
     (SQL_CALLBACK_PARM)&db_wait_request );
  ```

- **DB_CALLBACK_MESSAGE** 使用此函数后，应用程序可以对处理请求期间从服务器接收的消息进行处理。

  回调原型如下：

  ```
  void SQL_CALLBACK message_callback(
  SQLCA * sqlca,
  unsigned char msg_type,
  an_sql_code code,
  unsigned short length,
  char *  msg
  );
  ```

msg_type 参数说明了消息的重要性，而您可能需要以不同方式处理不同的消息。可供使用的消息类型有 MESSAGE_TYPE_INFO、MESSAGE_TYPE_WARNING、MESSAGE_TYPE_ACTION 和 MESSAGE_TYPE_STATUS。这些常量在 *sqldef.h* 中定义。*code* 字段可提供与消息关联的 SQLCODE，否则该值为 0。*length* 字段指定消息的长度。消息*不*是以空值终止的。

例如，Interactive SQL 回调在 [消息] 选项卡中显示 STATUS 和 INFO 消息，而在窗口中显示类型为 ACTION 和 WARNING 的消息。如果应用程序不注册此回调，会有一个缺省回调，它导致将所有消息写入服务器日志文件（如果正在调试并指定了日志文件）。另外，MESSAGE_TYPE_WARNING 和 MESSAGE_TYPE_ACTION 类型的消息会以与操作系统相关的方式更为突出地显示。

如果某消息回调未被应用程序注册，则在指定了 LogFile 连接参数后，发送至客户端的消息将被保存到日志文件中。此外，发送至客户端的 ACTION 或 STATUS 消息在 Windows 操作系统下会出现窗口中，而在 Unix 操作系统下则被记录到 stderr。

- **DB_CALLBACK_VALIDATE_FILE_TRANSFER** 用于注册文件传输校验回调函数。在允许进行任何传输前，客户端库会调用校验回调函数（如果存在）。如果在执行间接语句（从存储过程内部）期间，请求进行客户端数据传输，则除非客户端应用程序注册了校验回调函数，否则客户端库将不允许进行传输。下面更详尽地介绍了进行校验调用的条件。

  回调原型如下：

  ```
  int SQL_CALLBACK file_transfer_callback(
  SQLCA * sqlca,
  char * file_name,
  int is_write
  );
  ```

  file_name 参数是要读取或写入的文件的名称。如果请求读取（从客户端传输到服务器），则 *is_write* 参数为 0，如果请求写入，则该参数为非零值。如果不允许进行文件传输，则回调函数应返回 0，否则返回非零值。

  为确保数据安全，服务器会跟踪请求文件传输的语句的源。服务器会确定语句是否是从客户端应用程序直接接收的。从客户端启动数据传输时，服务器会将语句源的相关信息发送到客户端软件。对于嵌入式 SQL 客户端库而言，仅当是由于执行客户端应用程序直接发送的语句而请求数据传输时，它才会允许无条件传输数据。否则，应用程序必须注册上文所述的校验回调函数，如果未注册该函数，则传输会被拒绝，而且语句失败并出现一个错误。请注意，如果客户端语句调用的某个存储过程在数据库中已经存在，则对该存储过程本身的执行不被视为是客户端启动的语句所完成的。但是，如果客户端应用程序显式地创建一个临时存储过程，则服务器会将对该存储过程的执行视为是由客户端启动的。同样，如果客户端应用程序执行一个批处理语句，则该批处理语句的执行被视为是直接由客户端应用程序完成的。

db_start_database 函数

原型

unsigned int **db_start_database(** SQLCA * *sqlca*, char * *parms* **);**

参数

- **sqlca** 指向 SQLCA 结构的指针。有关信息，请参见“SQL 通信区域 (SQLCA)”一节 第 532 页。
- **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*。例如：

  ```
  "UID=DBA;PWD=sql;DBF=c:\\db\\mydatabase.db"
  ```

有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

说明

在现有服务器上启动数据库（如果可能）。否则，启动新的服务器。“查找数据库服务器”一节 《SQL Anywhere 服务器 - 数据库管理》中说明了启动数据库的步骤。

如果数据库已运行或已成功启动，则返回值为真（非零）并且将 SQLCODE 设置为 0。错误消息在 SQLCA 中返回。

如果在参数中提供了用户 ID 和口令，则它们将被忽略。

启动和停止数据库所需的权限在服务器命令行上设置。有关信息，请参见“-gd 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

db_start_engine 函数

原型

unsigned int **db_start_engine(** SQLCA * *sqlca*, char * *parms* **);**

参数

- **sqlca** 指向 SQLCA 结构的指针。有关信息，请参见“SQL 通信区域 (SQLCA)”一节 第 532 页。
- **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*。例如，

  ```
  "UID=DBA;PWD=sql;DBF=c:\\db\\mydatabase.db"
  ```

  有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

说明

如果数据库服务器尚未运行，则将其启动。

有关此函数执行步骤的说明，请参见“查找数据库服务器”一节 《SQL Anywhere 服务器 - 数据库管理》。

如果数据库已运行或已成功启动，则返回值为 TRUE（非零）并且将 SQLCODE 设置为 0。错误消息在 SQLCA 中返回。

下面对 db_start_engine 的调用会启动数据库服务器并将其命名为 demo，虽然使用了 DBF 连接参数，但不会装载数据库：

```
db_start_engine( &sqlca,
    "DBF=samples-dir\\demo.db;START=dbeng11" );
```

如果希望启动服务器和数据库，请在 StartLine (START) 连接参数中包含数据库文件：

```
db_start_engine( &sqlca,
   "ENG=eng_name;START=dbeng11 samples-dir\\demo.db" );
```

此调用会启动服务器，将其命名为 eng_name，并启动该服务器上的 SQL Anywhere 示例数据库。

db_start_engine 函数在尝试启动某个服务器之前会先尝试连接到它，以免出现试图启动已在运行的服务器的情况。

ForceStart (FORCE) 连接参数仅由 db_start_engine 函数使用。当设置为 YES 时，在尝试启动服务器之前不尝试连接到服务器。这样，下面的一对命令就能够按预期方式工作：

1. 启动名为 server_1 的数据库服务器：

   ```
   start dbeng11 -n server_1 demo.db
   ```

2. 强制启动一台新服务器并连接到它：

   ```
   db_start_engine( &sqlda,
     "START=dbeng11 -n server_2 mydb.db;ForceStart=YES" )
   ```

如果没有使用 ForceStart (FORCE)，而且没有 ServerName (ENG) 参数，则第二个命令会尝试连接到 server_1。db_start_engine 函数不从 StartLine (START) 参数的 -n 选项获取服务器名。

db_stop_database 函数

原型

unsigned int **db_stop_database(** SQLCA * *sqlca*, char * *parms* **);**

参数

- **sqlca** 指向 SQLCA 结构的指针。有关信息，请参见“SQL 通信区域 (SQLCA)”一节 第 532 页。
- **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*。例如：

  ```
  "UID=DBA;PWD=sql;DBF=c:\\db\\mydatabase.db"
  ```

  有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

说明

在由 ServerName (ENG) 标识的服务器上停止由 DatabaseName (DBN) 标识的数据库。如果未指定 ServerName，则使用缺省服务器。

缺省情况下，此函数不停止存在现有连接的数据库。如果 Unconditional 为 yes，则不管是否存在现有连接都会停止数据库。

返回值 TRUE 指示没有错误。

启动和停止数据库所需的权限在服务器命令行上设置。有关信息，请参见“-gd 服务器选项”一节《SQL Anywhere 服务器 - 数据库管理》。

db_stop_engine 函数

原型

unsigned int **db_stop_engine(** SQLCA * *sqlca*, char * *parms* **);**

参数

- **sqlca** 指向 SQLCA 结构的指针。有关信息，请参见“SQL 通信区域 (SQLCA)”一节 第 532 页。
- **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*。例如，

  ```
  "UID=DBA;PWD=sql;DBF=c:\\db\\mydatabase.db"
  ```

  有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

说明

停止数据库服务器的执行。此函数执行的步骤有：

- 查找名称与 ServerName (ENG) 参数匹配的本地数据库服务器。如果未指定 ServerName，则查找缺省的本地数据库服务器。
- 如果找不到匹配的服务器，则此函数成功返回。
- 向服务器发送一个请求，让服务器执行检查点操作并关闭所有数据库。
- 卸载数据库服务器。

缺省情况下，此函数不停止有现有连接的数据库。如果 Unconditional 为 yes，则不管是否存在现有连接都会停止数据库服务器。

C 程序可以使用此函数，而不用正在生成的 dbstop。返回值 TRUE 指示没有错误。

是否可以使用 db_stop_engine 取决于在 -gk 服务器选项上的权限设置。请参见“-gk 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

db_string_connect 函数

原型

unsigned int **db_string_connect(** SQLCA * *sqlca*, char * *parms* **);**

参数

- **sqlca** 指向 SQLCA 结构的指针。有关信息，请参见“SQL 通信区域 (SQLCA)”一节 第 532 页。

- **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*。例如：

  ```
  "UID=DBA;PWD=sql;DBF=c:\\db\\mydatabase.db"
  ```

  有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

说明

提供了嵌入式 SQL CONNECT 语句所具有的功能以外的其它功能。

有关此函数所用算法的说明，请参见“对连接进行故障排除”一节 《SQL Anywhere 服务器 - 数据库管理》。

如果成功建立连接，则返回值为 TRUE（非零），否则返回值为 FALSE（零）。有关启动服务器、启动数据库或进行连接的错误消息均在 SQLCA 中返回。

db_string_disconnect 函数

原型

unsigned int **db_string_disconnect(**
SQLCA * *sqlca*,
char * *parms* **);**

参数

- **sqlca** 指向 SQLCA 结构的指针。有关信息，请参见“SQL 通信区域 (SQLCA)”一节 第 532 页。
- **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*。例如：

  ```
  "UID=DBA;PWD=sql;DBF=c:\\db\\mydatabase.db"
  ```

  有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

说明

此函数断开由 ConnectionName 参数标识的连接。忽略所有其它参数。

如果在字符串中未指定 ConnectionName 参数，则断开未命名的连接。它等效于嵌入式 SQL DISCONNECT 语句。如果连接成功结束，则返回值为 TRUE。错误消息在 SQLCA 中返回。

如果数据库是使用 AutoStop=yes 参数启动的且没有其它到该数据库的连接，则此函数关闭该数据库。如果服务器是使用 AutoStop=yes 参数启动的且没有其它正在运行的数据库，它也会停止该服务器。

db_string_ping_server 函数

原型

```
unsigned int db_string_ping_server(
SQLCA * sqlca,
char * connect_string,
unsigned int connect_to_db );
```

说明

- **connect_string** *connect_string* 是一个常规连接字符串，它可能包含服务器和数据库信息，也可能不包含。
- **connect_to_db** 如果 *connect_to_db* 为非零值 (TRUE)，此函数会尝试连接到服务器上的数据库。只有在连接字符串足以连接到指定服务器上的指定数据库时，才返回 TRUE。
- **connect_to_db** 如果 *connect_to_db* 为零，此函数只尝试定位服务器。只有在连接字符串足以定位服务器时，才返回 TRUE。它不尝试连接到数据库。

db_time_change 函数

原型

```
unsigned int db_time_change(
SQLCA * sqlca);
```

说明

sqlca 指向 SQLCA 结构的指针。有关信息，请参见“SQL 通信区域 (SQLCA)”一节第 532 页。此函数允许客户端通知服务器客户端上的时间已发生更改。此函数将重新计算时区调整并将其发送给服务器。在 Windows 平台上，建议应用程序在接收到 WM_TIMECHANGE 消息时调用此函数。这样可确保 UTC 时间戳与时间更改、时区更改或夏令时更改保持一致。

如果成功则返回 TRUE，否则返回 FALSE。

fill_s_sqlda 函数

原型

```
struct sqlda * fill_s_sqlda(
struct sqlda * sqlda,
unsigned int maxlen );
```

说明

与 fill_sqlda 相同，只不过它将 *sqlda* 中的所有数据类型更改为类型 DT_STRING。将分配足够的空间，以保存最初由 SQLDA 指定的类型的字符串表示，最大为 *maxlen* 字节。会相应修改 SQLDA 中的长度字段 (sqllen)。如果成功则返回 *sqlda*；如果没有足够的可用内存则返回空指针。

SQLDA 应使用 free_filled_sqlda 函数释放。

fill_sqlda 函数

原型

struct sqlda * **fill_sqlda(** struct sqlda * *sqlda* **);**

说明

为在 *sqlda* 的每个描述符中说明的每个变量分配空间，并将此内存的地址指派给对应描述符的 sqldata 字段。为描述符中指明的数据库类型和长度分配足够的空间。如果成功则返回 *sqlda*；如果没有足够的可用内存则返回空指针。

SQLDA 应使用 free_filled_sqlda 函数释放。

free_filled_sqlda 函数

原型

void **free_filled_sqlda(** struct sqlda * *sqlda* **);**

说明

释放分配给每个 sqldata 指针的内存和分配给 SQLDA 自身的空间。不释放任何空指针。

只有在 fill_sqlda 或 fill_s_sqlda 用来分配 SQLDA 的 sqldata 字段时才应该调用此函数。

调用此函数会导致自动调用 free_sqlda，因此由 alloc_sqlda 分配的任何描述符都被释放。

free_sqlda 函数

原型

void **free_sqlda(** struct sqlda * *sqlda* **);**

说明

释放分配给此 *sqlda* 的空间并释放指示符变量空间，这些空间是在 fill_sqlda 中分配的。不会释放每个 sqldata 指针引用的内存。

free_sqlda_noind 函数

原型

void **free_sqlda_noind(** struct sqlda * *sqlda* **);**

说明

释放分配给此 *sqlda* 的空间。不会释放每个 sqldata 指针引用的内存。将忽略指示符变量指针。

sql_needs_quotes 函数

原型

unsigned int **sql_needs_quotes(** SQLCA **sqlca*, char * *str* **);**

说明

返回 TRUE 或 FALSE 值，指示当字符串用作 SQL 标识符时是否需要用双引号引起来。此函数向数据库服务器发出请求以确定是否需要引号。相关信息存储在 sqlcode 字段中。

返回值/代码的组合有三种情况：

- **return = FALSE, sqlcode = 0** 该字符串不需要引号。
- **return = TRUE** sqlcode 始终是 SQLE_WARNING，该字符串需要引号。
- **return = FALSE** 如果 sqlcode 不是 SQLE_WARNING，则此测试不能确定是否需要引号。

sqlda_storage 函数

原型

unsigned int **sqlda_storage(** struct sqlda * *sqlda*, int *varno* **);**

说明

返回无符号的 32 位整数值，该值表示存储 sqlda->sqlvar[varno] 中所述变量的任意值时所需的存储量。

sqlda_string_length 函数

原型

unsigned int **sqlda_string_length(** struct sqlda * *sqlda*, int *varno* **);**

说明

返回无符号的 32 位整数值，该值表示保存变量 sqlda->sqlvar[varno]（不管其类型是什么）所需的 C 字符串（DT_STRING 类型）的长度。

sqlerror_message 函数

原型

char * **sqlerror_message(** SQLCA * *sqlca***,** char * *buffer***,** int *max* **);**

说明

返回到包含错误消息的字符串的指针。错误消息包含 SQLCA. 中的错误代码的文本。如果没有指出错误，则返回空指针。将错误消息置于提供的缓冲区中，如有必要则将其截至长度 *max*。

嵌入式 SQL 语句汇总

EXEC SQL
所有嵌入式 SQL 语句都必须以 EXEC SQL 开头且以分号 (;) 结尾。

有两组嵌入式 SQL 语句。标准 SQL 语句的用法是：只需将其置于 C 程序中，并在其前后分别加上 EXEC SQL 和分号 (;) 即可。CONNECT、DELETE、SELECT、SET 和 UPDATE 有一些附加格式只能在嵌入式 SQL 中使用。具有附加格式的语句则属于特定于嵌入式 SQL 语句的第二个类别。

有关标准 SQL 语句的说明，请参见“SQL 语句” 《SQL Anywhere 服务器 - SQL 参考》。

有几个 SQL 语句是特定于嵌入式 SQL 的并只能在 C 程序中使用。请参见“SQL 语言元素” 《SQL Anywhere 服务器 - SQL 参考》。

可以从嵌入式 SQL 应用程序使用标准的数据操作语句和数据定义语句。另外，以下语句也是专用于嵌入式 SQL 编程的语句：

- **ALLOCATE DESCRIPTOR** 为描述符分配内存。请参见“ALLOCATE DESCRIPTOR 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **CLOSE** 关闭游标。请参见“CLOSE 语句 [ESQL] [SP]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **CONNECT** 连接到数据库。请参见“CONNECT 语句 [ESQL] [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **DEALLOCATE DESCRIPTOR** 回收描述符占用的内存。请参见“DEALLOCATE DESCRIPTOR 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **Declaration section** 为数据库通信声明主机变量。请参见“声明部分 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **DECLARE CURSOR** 声明游标。请参见“DECLARE CURSOR 语句 [ESQL] [SP]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **DELETE（已定位）** 删除游标中当前位置的行。请参见“DELETE（定位）语句 [ESQL] [SP]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **DESCRIBE** 描述特定 SQL 语句的主机变量。请参见“DESCRIBE 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **DISCONNECT** 断开与数据库服务器的连接。请参见“DISCONNECT 语句 [ESQL] [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **DROP STATEMENT** 释放预准备语句所使用的资源。请参见“DROP STATEMENT 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **EXECUTE** 执行特定的 SQL 语句。请参见“EXECUTE 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **EXPLAIN** 解释特定游标的优化策略。请参见“EXPLAIN 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

- **FETCH** 从游标读取行。请参见“FETCH 语句 [ESQL] [SP]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **GET DATA** 从游标读取 Long 型值。请参见“GET DATA 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **GET DESCRIPTOR** 检索有关 SQLDA 中变量的信息。请参见“GET DESCRIPTOR 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **GET OPTION** 获得特定数据库选项的设置。请参见“GET OPTION 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **INCLUDE** 包括要进行 SQL 预处理的文件。请参见“INCLUDE 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **OPEN** 打开游标。请参见“OPEN 语句 [ESQL] [SP]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **PREPARE** 准备特定的 SQL 语句。请参见“PREPARE 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **PUT** 向游标中插入行。请参见“PUT 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **SET CONNECTION** 更改活动连接。请参见“SET CONNECTION 语句 [Interactive SQL] [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **SET DESCRIPTOR** 描述 SQLDA 中的变量并将数据置于 SQLDA 中。请参见“SET DESCRIPTOR 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **SET SQLCA** 使用一个 SQLCA（非缺省全局 SQLCA）。请参见“SET SQLCA 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **UPDATE（已定位）** 更新游标当前所在的行。请参见“UPDATE（定位）语句 [ESQL] [SP]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **WHENEVER** 指定 SQL 语句中出现错误时要采取的操作。请参见“WHENEVER 语句 [ESQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

SQL Anywhere C API 参考

目录

SQL Anywhere C API 1.0 版简介

SQL Anywhere C 应用程序编程接口 (API) 可用于为多种解释编程语言简化 C 和 C++ 包装驱动程序的创建过程，其中包括 PHP、Perl、Python 和 Ruby。SQL Anywhere C API 处于 DBLIB 包的最上层，采用嵌入式 SQL 实现。

虽然 SQL Anywhere C API 不能替代 DBLIB，但可以简化使用 C 和 C++ 创建应用程序的过程。使用 SQL Anywhere C API 无需对嵌入式 SQL 具有很深的了解。有关实现的详细信息，请参见 *sqlany_imp.sqc*。

API 分布

API 在 Microsoft Windows 系统上作为动态链接库 (DLL) (*dbcapi.dll*) 构建，而在 Unix 系统上作为共享对象 (*libdbcapi.so*) 构建。DLL 静态链接到在其上构建自身的 SQL Anywhere 版本的 DBLIB 包。装载 *dbcapi.dll* 文件时，操作系统会装载对应的 *dblibX.dll* 文件。使用 *dbcapi.dll* 的应用程序可直接链接到该文件，亦可动态装载它。

SQL Anywhere C API 数据类型和入口点的描述在主头文件 (*sacapi.h*) 中提供。

线程支持

SQL Anywhere C API 库觉察不到线程；该库不能执行任何需要互斥机制的任务。为使该库能够在线程应用程序中运行，一个连接中只允许一个请求。根据此规则，在访问任何连接特定的资源时，应用程序将负责实现互斥。其中包括连接处理、预准备语句和结果集对象。

动态装载接口库

动态装载 DLL 的代码包含在头文件 *sacapidll.h* 中。应用程序必须包括 *sacapidll.h* 头文件并在 *sacapidll.c* 中编译。可使用 sqlany_initialize_interface 动态装载 DLL 和查找入口点。

sacapidll.h

动态装载 SQL Anywhere C API DLL。

语法

```
#define function(x) x ## _func x
```

注释

必须将 *sacapidll.h* 包括在源文件中并在 *sacapidll.c* 中编译。

sqlany_initialize_interface 函数

初始化 SQLAnywhereInterface 对象并动态装载 DLL。

语法

```
int sqlany_initialize_interface ( SQLAnywhereInterface * api, const char * optional_path_to_dll )
```

参数

- **api** 要初始化的 API 结构的名称。
- **optional_path_to_dll** 可选参数，指定到 DLL API 的路径。

注释

此函数尝试动态装载 SQL Anywhere C API DLL 并查找 DLL 的所有入口点。填充 SQLAnywhereInterface 结构的字段时应指向 DLL 中的相应函数。如果可选路径参数为 NULL，则检查环境变量 SQLANY_DLL_PATH。如果已设置该变量，则库会尝试装载由该环境变量指定的 DLL。如果上述方法失败，则接口将尝试直接装载 DLL（这依赖于环境设置是否正确）。

返回值

- 成功初始化则为 1，失败则为 0。

另请参见

- “connecting.cpp”一节第 624 页
- “dbcapi_isql.cpp”一节第 625 页
- “fetching_a_result_set.cpp”一节第 628 页
- “fetching_multiple_from_sp.cpp”一节第 630 页
- “preparing_statements.cpp”一节第 632 页
- “send_retrieve_full_blob.cpp”一节第 634 页
- “send_retrieve_part_blob.cpp”一节第 636 页

sqlany_finalize_interface 函数

卸载库并解除 SQLAnywhereInterface 结构的初始化。

语法

void **sqlany_finalize_interface**(SQLAnywhereInterface * *api*)

参数

- **api** 要初始化的 API 结构的名称。

注释

使用 sqlany_finalize_interface 结束并释放与 SQL Anywhere C API DLL 相关联的资源。

另请参见

- “connecting.cpp”一节第 624 页
- “dbcapi_isql.cpp”一节第 625 页
- “fetching_a_result_set.cpp”一节第 628 页
- “fetching_multiple_from_sp.cpp”一节第 630 页
- “preparing_statements.cpp”一节第 632 页
- “send_retrieve_full_blob.cpp”一节第 634 页
- “send_retrieve_part_blob.cpp”一节第 636 页

sacapi.h

装载 SQL Anywhere C API DLL 并查找初始化实例时的所有入口点。

注释

网络环境中只需要此元素的一个实例。

成员

sacapi.h 文件引用的所有成员，包括所有继承的成员。

- “a_sqlany_bind_param_info 结构”一节第 612 页
- “a_sqlany_column_info 结构”一节第 613 页
- “a_sqlany_data_info 结构”一节第 614 页
- “a_sqlany_data_value 结构”一节第 615 页
- “SQLAnywhereInterface 结构”一节第 616 页

sqlany_affected_rows 函数

返回受执行预准备语句影响的行数。

语法

sacapi_i32 **sqlany_affected_rows**(a_sqlany_stmt * *stmt*)

参数

- **stmt** 预准备并成功执行的语句，其中不返回任何结果集。例如，执行了 INSERT、UPDATE 或 DELETE 语句。

返回值

- **sacapi_i32** 受影响的行数，或失败时返回 -1

另请参见

- “sqlany_execute 函数”一节第 597 页

sqlany_bind_param 函数

将用户提供的缓冲区作为参数绑定到预准备语句。

语法

sacapi_bool **sqlany_bind_param**(a_sqlany_stmt * *stmt*, sacapi_u32 *index*, a_sqlany_bind_param * *param*)

参数

- **stmt** 使用 sqlany_prepare 成功预准备的语句。

- **index** 参数的索引。该数字必须在 0 到 sqlany_num_params() - 1 之间。
- **param** 对要绑定参数 a_sqlany_bind_param 结构的说明。

返回值

成功时为 1，不成功时为 0。

另请参见

- “sqlany_describe_bind_param 函数”一节第 596 页

sqlany_clear_error 函数

清除上次存储的错误代码。

语法

void **sqlany_clear_error**(a_sqlany_connection * *conn*)

参数

- **conn** 从 sqlany_new_connection 返回的连接对象。

另请参见

- “sqlany_new_connection 函数”一节第 606 页

sqlany_client_version 函数

返回当前的客户端版本。

语法

sacapi_bool **sqlany_client_version**(char * *buffer*, size_t *len*)

参数

- **buffer** 将在其中填充客户端版本字符串的缓冲区。
- **len** 缓冲区的长度。

返回值

成功时为 1，不成功时为 0。

sqlany_commit 函数

提交当前事务。

语法

sacapi_bool **sqlany_commit**(a_sqlany_connection * *conn*)

参数

- **conn** 要在其上执行提交操作的连接对象。

返回值

成功时为 1，不成功时为 0。

另请参见

- “sqlany_rollback 函数”一节第 608 页

sqlany_connect 函数

使用指定的连接对象和连接字符串创建与某个 SQL Anywhere 数据库服务器的连接。

语法

sacapi_bool **sqlany_connect**(a_sqlany_connection * *conn*, const char * *str*)

参数

- **conn** 由 sqlany_new_connection 创建的连接对象。
- **str** 一个 SQL Anywhere 连接字符串。

返回值

如果成功建立连接则为 1，连接失败则为 0。使用 sqlany_error 检索错误代码和消息。

另请参见

- “sqlany_new_connection 函数”一节第 606 页
- “sqlany_error 函数”一节第 597 页
- “连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》
- “SQL Anywhere 数据库连接” 《SQL Anywhere 服务器 - 数据库管理》

示例

```
a_sqlany_connection * conn;
  conn = sqlany_new_connection();
  if( !sqlany_connect( conn, "uid=dba;pwd=sql" ) ) {
       char           reason[SACAPI_ERROR_SIZE];
       sacapi_i32     code;
       code = sqlany_error( conn, reason, sizeof(reason) );
       printf( "Connection failed. Code: %d Reason: %s\n", code, reason );
  } else {
       printf( "Connected successfully!\n" );
       sqlany_disconnect( conn );
  }
  sqlany_free_connection( conn );
```

sqlany_describe_bind_param 函数

描述预准备语句的绑定参数。

语法

sacapi_bool **sqlany_describe_bind_param**(a_sqlany_stmt * *stmt*,
sacapi_u32 *index*, a_sqlany_bind_param * *param*)

参数

- **stmt** 使用 sqlany_prepare 成功预准备的语句。
- **index** 参数的索引。该数字必须在 0 到 sqlany_num_params() - 1 之间。
- **param** 在其中填充信息的 a_sqlany_bind_param 结构。

注释

此函数允许调用者确定有关预准备语句参数的信息。预准备语句（存储过程或 DML）的类型确定所提供的信息量。始终会提供参数的方向（输入、输出或输入-输出）。

返回值

成功则为 1，失败则为 0。

另请参见

- “sqlany_bind_param 函数”一节第 593 页
- “sqlany_prepare 函数”一节第 607 页

sqlany_disconnect 函数

断开 SQL Anywhere 连接。回退所有未提交的事务。

语法

sacapi_bool **sqlany_disconnect**(a_sqlany_connection * *conn*)

参数

- **conn** 已使用 sqlany_connect 建立连接的连接对象。

返回值

成功则为 1，失败则为 0。

另请参见

- “sqlany_connect 函数”一节第 595 页
- “sqlany_new_connection 函数”一节第 606 页

sqlany_error 函数

返回存储在连接对象中的上一错误代码和消息。

语法

sacapi_i32 **sqlany_error**(a_sqlany_connection * *conn*, char * *buffer*, size_t *size*)

参数

- **conn** 从 sqlany_new_connection 返回的连接对象。
- **buffer** 将在其中填充错误消息的缓冲区。
- **size** 提供的缓冲区大小。

返回值

上一错误代码。正值表示警告，负值表示错误，0 表示成功执行。

另请参见

- “sqlany_connect 函数”一节第 595 页
- “按 SQLCODE 排序的 SQL Anywhere 错误消息”一节 《错误消息》

sqlany_execute 函数

执行预准备语句。

语法

sacapi_bool **sqlany_execute**(a_sqlany_stmt * *stmt*)

参数

- **stmt** 使用 sqlany_prepare 成功预准备的语句。

返回值

成功则为 1，失败则为 0。

注释

可使用 sqlany_num_cols 验证语句是否返回结果集。

另请参见

- “sqlany_prepare 函数”一节第 607 页

示例

```
// This example shows how to execute a statement that does not return a result
set
   a_sqlany_stmt *       stmt;
   int                   I;
   a_sqlany_bind_param   param;
```

```
stmt = sqlany_prepare( conn, "insert into moe(id,value) values( ?,? )" );
if( stmt ) {
    sqlany_describe_bind_param( stmt, 0, &param );
    param.value.buffer = (char *)&I;
    param.value.type   = A_VAL32;
    sqlany_bind_param( stmt, 0, &param );

    sqlany_describe_bind_param( stmt, 1, &param );
    param.value.buffer = (char *)&I;
    param.value.type   = A_VAL32;
    sqlany_bind_param( stmt, 1, &param );

    for( I = 0; I < 10; I++ ) {
        if( !sqlany_execute( stmt ) ) {
            // call sqlany_error()
        }
    }
    sqlany_free_stmt( stmt );
}
```

sqlany_execute_direct 函数

执行由字符串参数指定的 SQL 语句。

语法

a_sqlany_stmt * **sqlany_execute_direct**(a_sqlany_connection * *conn*, const char * *sql*)

参数

- **conn** 已使用 sqlany_connect 建立连接的连接对象。
- **sql** SQL 字符串。SQL 字符串不应包含诸如 **?** 之类的参数。

注释

如果要准备和执行语句，或者要取代 sqlany_prepare 后接 sqlany_execute，则可使用此函数，。不要使用此函数执行带参数的 SQL 语句。

返回值

函数成功执行时返回语句句柄，函数执行不成功时返回 NULL。

另请参见

- “sqlany_fetch_absolute 函数”一节第 599 页
- “sqlany_fetch_next 函数”一节第 600 页
- “sqlany_num_cols 函数”一节第 606 页
- “sqlany_get_column 函数”一节第 601 页

示例

```
stmt = sqlany_execute_direct( conn, "select * from employees" ) ) {
  if( stmt ) {
      while( sqlany_fetch_next( stmt ) ) {
          int i;
```



```
            for( i = 0; i < sqlany_num_cols( stmt ); i++ ) {
                // Get i'th column data
            }
        }
        sqlany_free_stmt( stmt  );
    }
```

sqlany_execute_immediate 函数

立即执行指定的 SQL 语句，不返回结果集。

语法

sacapi_bool **sqlany_execute_immediate**(a_sqlany_connection * *conn*, const char * *sql*)

参数

- **conn** 已使用 sqlany_connect 建立连接的连接对象。
- **sql** 表示要执行的 SQL 语句的字符串。

返回值

成功则为 1，失败则为 0。

另请参见

- “sqlany_error 函数”一节第 597 页

sqlany_fetch_absolute 函数

将结果集中的当前行移动到指定的行编号处，然后读取该行的数据。

语法

sacapi_bool **sqlany_fetch_absolute**(a_sqlany_stmt * *stmt*, sacapi_i32 *row_num*)

参数

- **stmt** 由 sqlany_execute 或 sqlany_execute_direct 执行的语句对象。
- **row_num** 要读取的行编号。第一行为 1，最后一行为 -1。

返回值

如果读取成功则为 1，读取失败时为 0。

另请参见

- “sqlany_error 函数”一节第 597 页
- “sqlany_execute 函数”一节第 597 页
- “sqlany_execute_direct 函数”一节第 598 页
- “sqlany_fetch_next 函数”一节第 600 页

sqlany_fetch_next 函数

返回结果集的下一行。此函数首先前移行指针，然后读取新行的数据。

语法

sacapi_bool **sqlany_fetch_next**(a_sqlany_stmt * *stmt*)

参数

- **stmt** 由 sqlany_execute 或 sqlany_execute_direct 执行的语句对象。

返回值

如果读取成功则为 1，读取失败时为 0。

另请参见

- “sqlany_error 函数”一节第 597 页
- “sqlany_execute 函数”一节第 597 页
- “sqlany_execute_direct 函数”一节第 598 页
- “sqlany_fetch_absolute 函数”一节第 599 页

sqlany_fini 函数

释放由 API 分配的资源。

语法

void **sqlany_fini**(void)

另请参见

- “sqlany_init 函数”一节第 604 页

sqlany_free_connection 函数

释放与连接对象关联的资源。

语法

void **sqlany_free_connection**(a_sqlany_connection * *conn*)

参数

- **conn** 由 sqlany_new_connection 创建的连接对象。

另请参见

- “sqlany_new_connection 函数”一节第 606 页

sqlany_free_stmt 函数

释放与预准备语句对象关联的资源。

语法

void **sqlany_free_stmt**(a_sqlany_stmt * *stmt*)

参数

- **stmt** 成功执行 sqlany_prepare 或 sqlany_execute_direct 所返回的语句对象。

另请参见

- “sqlany_prepare 函数”一节第 607 页
- “sqlany_execute_direct 函数”一节第 598 页

sqlany_get_bind_param_info 函数

检索使用 sqlany_bind_param 绑定的参数的相关信息。

语法

sacapi_bool **sqlany_get_bind_param_info**(a_sqlany_stmt * *stmt*, sacapi_u32 *index*,
a_sqlany_bind_param_info * *info*)

参数

- **stmt** 使用 sqlany_prepare 成功预准备的语句。
- **index** 参数的索引。该数字必须在 0 到 sqlany_num_params() - 1 之间。
- **info** 将在其中填充绑定参数信息的 sqlany_bind_param_info 缓冲区。

返回值

成功则为 1，失败则为 0。

另请参见

- “sqlany_bind_param 函数”一节第 593 页
- “sqlany_describe_bind_param 函数”一节第 596 页
- “sqlany_prepare 函数”一节第 607 页

sqlany_get_column 函数

用为指定列读取的值填充提供的缓冲区。

语法

sacapi_bool **sqlany_get_column**(a_sqlany_stmt * *stmt*, sacapi_u32 *col_index*,
a_sqlany_data_value * *value*)

参数

- **stmt** 由 sqlany_execute 或 sqlany_execute_direct 执行的语句对象。
- **col_index** 要检索的列编号。列编号在 0 到 sqlany_num_cols() - 1 之间。
- **value** 要使用为列 col_index 读取的数据进行填充的 a_sqlany_data_value 对象。

注释

对于 A_BINARY 和 A_STRING * 数据类型，**value->buffer** 指向与结果集相关联的内部缓冲区。不要依赖于指针 **buffer** 的内容，也不要变更它，因为它在读取新行或释放结果集对象时会改变。将数据从指针复制到缓冲区。

length 字段指示 **value->buffer** 指向的有效字符数。**value->buffer** 中返回的数据不是以空值终止的。此函数读取来自 SQL Anywhere 数据库服务器的所有返回值。例如，如果某列包含一个 2 GB 的 blob，sqlany_get_column 函数将尝试分配足以容纳该值的内存空间。如不想分配内存，请使用 sqlany_get_data。

返回值

成功则为 1，失败则为 0。如果有任何参数无效，或者如果没有足够的内存用于检索来自 SQL Anywhere 数据库服务器的完整值，则会引起失败。

另请参见

- “sqlany_execute 函数”一节第 597 页
- “sqlany_execute_direct 函数”一节第 598 页
- “sqlany_fetch_absolute 函数”一节第 599 页
- “sqlany_fetch_next 函数”一节第 600 页

sqlany_get_column_info 函数

将列信息添加到 a_sqlany_column_info 结构。

语法

sacapi_bool **sqlany_get_column_info**(a_sqlany_stmt * *stmt*, sacapi_u32 *col_index*, a_sqlany_column_info * *info*)

参数

- **stmt** 由 sqlany_prepare 或 sqlany_execute_direct 创建的语句对象。
- **col_index** 列编号在 0 到 sqlany_num_cols - 1 之间。
- **info** 将在其中填充列信息的列信息结构。

返回值

成功时为 1，如果列索引超出范围，或者如果语句未返回结果集，则为 0。

另请参见

- “sqlany_execute 函数”一节第 597 页
- “sqlany_execute_direct 函数”一节第 598 页
- “sqlany_prepare 函数”一节第 607 页

sqlany_get_data 函数

将为指定列读取的数据检索到提供的缓冲区内存。

语法

sacapi_i32 **sqlany_get_data**(a_sqlany_stmt * *stmt*, sacapi_u32 *col_index*,
size_t *offset*, void * *buffer*, size_t *size*)

参数

- **stmt** 由 sqlany_execute 或 sqlany_execute_direct 执行的语句对象。
- **col_index** 要检索的列编号。列编号在 0 到 sqlany_num_cols() - 1 之间。
- **offset** 要获取的数据的起始偏移量。
- **buffer** 将在其中填充列内容的缓冲区。缓冲区指针必须按照复制到其中的数据类型正确对齐。
- **size** 缓冲区的大小（以字节为单位）。如果所指定的大小大于 2 GB，则该函数会失败。

返回值

成功复制到所提供缓冲区中的字节数。此数字不会超过 2 GB。0 表示没有数据还要复制。-1 表示失败。

另请参见

- “sqlany_execute 函数”一节第 597 页
- “sqlany_execute_direct 函数”一节第 598 页
- “sqlany_fetch_absolute 函数”一节第 599 页
- “sqlany_fetch_next 函数”一节第 600 页

sqlany_get_data_info 函数

检索由上一读取操作所读取的数据的相关信息。

语法

sacapi_bool **sqlany_get_data_info**(a_sqlany_stmt * *stmt*, sacapi_u32 *col_index*,
a_sqlany_data_info * *info*)

参数

- **stmt** 由 sqlany_execute 或 sqlany_execute_direct 执行的语句对象。
- **col_index** 列编号在 0 到 sqlany_num_cols() - 1 之间。

- **info** 将在其中填充所读取数据的元数据的数据信息缓冲区。

返回值

成功则为 1，失败则为 0。如果任何提供的参数无效，则 sqlany_get_data_info 函数会失败。

另请参见

- “sqlany_execute 函数”一节第 597 页
- “sqlany_execute_direct 函数”一节第 598 页
- “sqlany_fetch_absolute 函数”一节第 599 页
- “sqlany_fetch_next 函数”一节第 600 页

sqlany_get_next_result 函数

前进到多结果集查询中的下一结果集。

语法

sacapi_bool **sqlany_get_next_result**(a_sqlany_stmt * *stmt*)

参数

- **stmt** 由 sqlany_execute 或 sqlany_execute_direct 执行的语句对象。

返回值

如果语句成功前进到下一结果集则返回 1，否则返回 0。

另请参见

- “sqlany_execute 函数”一节第 597 页
- “sqlany_execute_direct 函数”一节第 598 页

示例

```
   stmt = sqlany_execute_direct( conn, "call
my_multiple_results_procedure()" );
    if( result ) {
        do {
            while( sqlany_fetch_next( stmt ) ) {
                // get column data
            }
        } while( sqlany_get_next_result( stmt ) );
        sqlany_free_stmt( stmt );
    }
```

sqlany_init 函数

初始化接口。

语法

sacapi_bool **sqlany_init**(const char * *app_name*, sacapi_u32 *api_version*, sacapi_u32 * *version_available*)

参数

- **app_name** 由 sqlany_execute 或 sqlany_execute_direct 执行的语句对象。
- **api_version** 已编译应用程序的版本（使用 **SQLANY_CURRENT_API_VERSION**）。
- **version_available** 最大的支持 API 版本。

返回值

成功则为 1，否则为 0。

另请参见

- “sqlany_fini 函数”一节第 600 页

示例

```
SQLAnywhereInterface  api;
a_sqlany_connection   *conn;
unsigned int          max_api_ver;

if( !sqlany_initialize_interface( &api, NULL ) ) {
    printf( "Could not initialize the interface!\n" );
    exit( 0 );
}

if( !api.sqlany_init( "MyAPP", SQLANY_CURRENT_API_VERSION, &max_api_ver )) {
    printf( "Failed to initialize the interface! Supported version = %d\n",
max_api_ver );
    sqlany_finalize_interface( &api );
    return -1;
}
```

sqlany_make_connection 函数

基于已提供的 DBLIB SQLCA 指针创建连接对象。

语法

a_sqlany_connection * **sqlany_make_connection**(void * *arg*)

参数

- **arg** 指向 DBLIB SQLCA 对象的 void * 指针。

另请参见

- “sqlany_new_connection 函数”一节第 606 页

sqlany_new_connection 函数

创建连接对象。

语法

a_sqlany_connection * **sqlany_new_connection**(void)

注释

在建立数据库连接之前必须创建 API 连接对象。可以从连接对象中检索错误。一个连接中每次只能处理一个请求。此外，一次不允许多个线程访问同一连接对象。当多个线程同时尝试访问一个连接对象时，会发生未定义行为或失败。

返回值

连接对象。

另请参见

- “sqlany_connect 函数”一节第 595 页
- “sqlany_disconnect 函数”一节第 596 页

sqlany_num_cols 函数

返回结果集中的列数。

语法

sacapi_i32 **sqlany_num_cols**(a_sqlany_stmt * *stmt*)

参数

- **stmt** 由 sqlany_prepare 或 sqlany_execute_direct 创建的语句对象。

返回值

结果集中的列数，若失败则返回 -1。

另请参见

- “sqlany_execute 函数”一节第 597 页
- “sqlany_execute_direct 函数”一节第 598 页
- “sqlany_prepare 函数”一节第 607 页

sqlany_num_params 函数

返回预期用于预准备语句的参数数目。

语法

sacapi_i32 **sqlany_num_params**(a_sqlany_stmt * *stmt*)

参数

- **stmt** 成功执行 sqlany_prepare 所返回的语句对象。

返回值

预期的参数数目，或 -1（如果语句对象无效）。

另请参见

- “sqlany_prepare 函数”一节第 607 页

sqlany_num_rows 函数

返回结果集中的行数。

语法

sacapi_i32 **sqlany_num_rows**(a_sqlany_stmt * *stmt*)

参数

- **stmt** 由 sqlany_execute 或 sqlany_execute_direct 执行的语句对象。

注释

缺省情况下，此函数仅返回一个估计值。要返回准确计数，请在连接中设置 ROW_COUNTS 选项。有关详细信息，请参见“row_counts 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

返回值

结果集中的行数。如果行数是估计值，则返回负数，且估计值为所返回整数的绝对值。如果行数为准确值，则返回值为正。

另请参见

- “sqlany_execute 函数”一节第 597 页
- “sqlany_execute_direct 函数”一节第 598 页

sqlany_prepare 函数

准备所提供的 SQL 字符串

语法

a_sqlany_stmt * **sqlany_prepare**(a_sqlany_connection * *conn*, const char * *sql*)

参数

- **conn** 已使用 sqlany_connect 建立连接的连接对象。
- **sql** 要准备的 SQL 语句。

返回值

SQL Anywhere 语句对象的句柄。

注释

由 sqlany_execute 执行与该语句对象相关联的语句。可使用 sqlany_free_stmt 释放与语句对象关联的资源。

另请参见

- “sqlany_execute 函数”一节第 597 页
- “sqlany_free_stmt 函数”一节第 601 页
- “sqlany_num_params 函数”一节第 606 页
- “sqlany_describe_bind_param 函数”一节第 596 页
- “sqlany_bind_param 函数”一节第 593 页

示例

```
char * str;
a_sqlany_stmt * stmt;

str = "select * from employees where salary >= ?";
stmt = sqlany_prepare( conn, str );
if( stmt == NULL ) {
    // Failed to prepare statement, call sqlany_error() for more info
}
```

sqlany_reset 函数

将语句重置为其准备状态。

语法

sacapi_bool **sqlany_reset**(a_sqlany_stmt * *stmt*)

参数

- **stmt** 使用 sqlany_prepare 成功预准备的语句。

返回值

成功则为 1，失败则为 0。

另请参见

- “sqlany_prepare 函数”一节第 607 页

sqlany_rollback 函数

回退当前事务。

语法

sacapi_bool **sqlany_rollback**(a_sqlany_connection * *conn*)

参数

- **conn** 要在其上执行回退操作的连接对象。

返回值

成功则为 1，否则为 0。

另请参见

- “sqlany_commit 函数”一节第 594 页

sqlany_send_param_data 函数

作为绑定参数的一部分发送数据。

语法

sacapi_bool **sqlany_send_param_data**(a_sqlany_stmt * *stmt*, sacapi_u32 *index*, char * *buffer*, size_t *size*)

参数

- **stmt** 使用 sqlany_prepare 成功预准备的语句。
- **index** 参数的索引。它应是 0 和 sqlany_num_params() - 1 之间的一个数字。
- **buffer** 要发送的数据。
- **size** 要发送的字节数。

返回值

成功则为 1，失败则为 0。

另请参见

- “sqlany_prepare 函数”一节第 607 页

sqlany_sqlstate 函数

检索当前的 SQL 状态。

语法

size_t **sqlany_sqlstate**(a_sqlany_connection * *conn*, char * *buffer*, size_t *size*)

参数

- **conn** 从 sqlany_new_connection 返回的连接对象。

- **buffer** 将在其中填充当前 5 字符 SQL 状态的缓冲区。
- **size** 缓冲区大小。

返回值

复制到缓冲区中的字节数。

另请参见

- “sqlany_error 函数”一节第 597 页
- “按 SQLSTATE 排序的 SQL Anywhere 错误消息”一节 《错误消息》

a_sqlany_bind_param 结构

绑定要执行的预准备语句参数。

语法

public **a_sqlany_bind_param**

属性

名称	类型	说明
direction	a_sqlany_bind_param	数据方向（输入、输出、输入-输出）。
name	a_sqlany_bind_param	绑定参数的名称。
value	a_sqlany_bind_param	要设置的值。

另请参见

- “sqlany_execute 函数”一节第 597 页

示例

要查看 a_sqlany_bind_param 结构引用语法的示例，请参见：

- “preparing_statements.cpp”一节第 632 页
- “send_retrieve_full_blob.cpp”一节第 634 页
- “send_retrieve_part_blob.cpp”一节第 636 页

a_sqlany_bind_param_info 结构

用于绑定要执行的预准备语句参数。

语法

```
typedef struct a_sqlany_bind_param_info
{
    char *              name;
    a_sqlany_data_direction     direction;
    a_sqlany_data_value         input_value;
    a_sqlany_data_value         output_value;
} a_sqlany_bind_param_info;
```

属性

名称	类型	说明
direction	a_sqlany_data_direction	参数方向。
input_value	a_sqlany_data_value	绑定输入值的相关信息。
name	char *	指向参数名称的指针。
output_value	a_sqlany_data_value	绑定输出值的相关信息。

另请参见

- “sqlany_execute 函数”一节第 597 页

示例

要查看 a_sqlany_bind_param_info 结构引用语法的示例，请参见：

- “preparing_statements.cpp”一节第 632 页
- “send_retrieve_full_blob.cpp”一节第 634 页
- “send_retrieve_part_blob.cpp”一节第 636 页

a_sqlany_column_info 结构

用于返回列元数据信息。

语法

```
typedef struct a_sqlany_column_info
{
    char *          name;
    a_sqlany_data_type       type;
    a_sqlany_native_type    native_type;
    unsigned short       precision;
    unsigned short       scale;
    size_t           max_size;
    sacapi_bool          nullable;
} a_sqlany_column_info;
```

属性

名称	类型	说明
max_size	size_t	此列中的数据值可承载数据的最大大小。
name	char *	列的名称（以空值终止）。只要结果集对象未释放，即可引用该字符串。
native_type	a_sqlany_native_type	数据库中列的本地类型。
nullable	sacapi_bool	列中的值是否可为空。
precision	unsigned short	精度。
scale	unsigned short	小数位数。
type	a_sqlany_data_type	列数据类型。

注释

可使用 sqlany_get_column_info 填充此结构。

示例

要查看使用 a_sqlany_column_info 结构的示例，请参见：

- “dbcapi_isql.cpp”一节第 625 页

a_sqlany_data_info 结构

用于返回结果集中列值的相关信息。

语法

```
typedef struct a_sqlany_data_info
{
   a_sqlany_data_type    type;
   sacapi_bool        is_null;
   size_t         data_size;
} a_sqlany_data_info;
```

属性

名称	类型	说明
data_size	size_t	可读取的总字节数。此字段仅在成功执行读取操作后才有效。
is_null	sacapi_bool	指示上一读取的数据是否为 NULL。此字段仅在成功执行读取操作后才有效。
type	a_sqlany_data_type	列中数据的类型。

注释

可使用 sqlany_get_data_info 将读取操作上次检索内容的相关信息用来填充此结构。

另请参见

- “sqlany_get_data_info 函数”一节第 603 页

示例

要查看使用 a_sqlany_data_info 结构的示例，请参见：

- “send_retrieve_part_blob.cpp”一节第 636 页

a_sqlany_data_value 结构

用于返回数据值的属性描述。

语法

```
typedef struct a_sqlany_data_value
{
    char *          buffer;
    size_t          buffer_size;
    size_t *        length;
    a_sqlany_data_type    type;
    sacapi_bool *    is_null;
} a_sqlany_data_value;
```

参数

名称	类型	说明
buffer	char *	指向用户提供的数据缓冲区的指针。
buffer_size	size_t	缓冲区大小。
is_null	sacapi_bool *	指向指示上一读取数据是否为 NULL 的指示符的指针。
length	size_t *	指向缓冲区中有效字节数的指针。必须小于 buffer_size。
type	a_sqlany_data_type	数据的类型。

示例

要查看使用 a_sqlany_data_value 结构的示例，请参见：

- “dbcapi_isql.cpp”一节第 625 页
- “fetching_a_result_set.cpp”一节第 628 页
- “send_retrieve_full_blob.cpp”一节第 634 页
- “send_retrieve_part_blob.cpp”一节第 636 页
- “preparing_statements.cpp”一节第 632 页

SQLAnywhereInterface 结构

SQL Anywhere C API 接口结构。

语法

```
typedef struct SQLAnywhereInterface {
    /** DLL handle.
     */
    void    * dll_handle;

    /** Flag to know if initialized or not.
     */
    int         initialized;

    /** Pointer to ::sqlany_init() function.
     */
    function( sqlany_init );

    /** Pointer to ::sqlany_fini() function.
     */
    function( sqlany_fini );

    /** Pointer to ::sqlany_new_connection() function.
     */
    function( sqlany_new_connection );

    /** Pointer to ::sqlany_free_connection() function.
     */
    function( sqlany_free_connection );

    /** Pointer to ::sqlany_make_connection() function.
     */
    function( sqlany_make_connection );

    /** Pointer to ::sqlany_connect() function.
     */
    function( sqlany_connect );

    /** Pointer to ::sqlany_disconnect() function.
     */
    function( sqlany_disconnect );

    /** Pointer to ::sqlany_execute_immediate() function.
     */
    function( sqlany_execute_immediate );

    /** Pointer to ::sqlany_prepare() function.
     */
    function( sqlany_prepare );

    /** Pointer to ::sqlany_free_stmt() function.
     */
    function( sqlany_free_stmt );

    /** Pointer to ::sqlany_num_params() function.
```



```
 */
function( sqlany_num_params );

/** Pointer to ::sqlany_describe_bind_param() function.
 */
function( sqlany_describe_bind_param );

/** Pointer to ::sqlany_bind_param() function.
 */
function( sqlany_bind_param );

/** Pointer to ::sqlany_send_param_data() function.
 */
function( sqlany_send_param_data );

/** Pointer to ::sqlany_reset() function.
 */
function( sqlany_reset );

/** Pointer to ::sqlany_get_bind_param_info() function.
 */
function( sqlany_get_bind_param_info );

/** Pointer to ::sqlany_execute() function.
 */
function( sqlany_execute );

/** Pointer to ::sqlany_execute_direct() function.
 */
function( sqlany_execute_direct );

/** Pointer to ::sqlany_fetch_absolute() function.
 */
function( sqlany_fetch_absolute );

/** Pointer to ::sqlany_fetch_next() function.
 */
function( sqlany_fetch_next );

/** Pointer to ::sqlany_get_next_result() function.
 */
function( sqlany_get_next_result );

/** Pointer to ::sqlany_affected_rows() function.
 */
function( sqlany_affected_rows );

/** Pointer to ::sqlany_num_cols() function.
 */
function( sqlany_num_cols );

/** Pointer to ::sqlany_num_rows() function.
 */
function( sqlany_num_rows );

/** Pointer to ::sqlany_get_column() function.
 */
```

```
    function( sqlany_get_column );

    /** Pointer to ::sqlany_get_data() function.
     */
    function( sqlany_get_data );

    /** Pointer to ::sqlany_get_data_info() function.
     */
    function( sqlany_get_data_info );

    /** Pointer to ::sqlany_get_column_info() function.
     */
    function( sqlany_get_column_info );

    /** Pointer to ::sqlany_commit() function.
     */
    function( sqlany_commit );

    /** Pointer to ::sqlany_rollback() function.
     */
    function( sqlany_rollback );

    /** Pointer to ::sqlany_client_version() function.
     */
    function( sqlany_client_version );

    /** Pointer to ::sqlany_error() function.
     */
    function( sqlany_error );

    /** Pointer to ::sqlany_sqlstate() function.
     */
    function( sqlany_sqlstate );

    /** Pointer to ::sqlany_clear_error() function.
     */
    function( sqlany_clear_error );

} SQLAnywhereInterface;
```

注释

应用程序环境中只需要此结构的一个实例。此结构由 **sqlany_initialize_interface** 函数初始化。此结构尝试动态装载 SQL Anywhere C API 动态链接库或共享对象，并查找 DLL 的所有入口点。填充 SQLAnywhereInterface 结构中的字段时应指向 DLL 中的相应函数。

另请参见

- “sqlany_initialize_interface 函数”一节第 591 页

a_sqlany_data_direction 枚举

数据方向枚举。

语法

```
typedef enum a_sqlany_data_direction
{
   DD_INVALID       = 0x0,
   DD_INPUT         = 0x1,
   DD_OUTPUT        = 0x2,
   DD_INPUT_OUTPUT  = 0x3
} a_sqlany_data_direction;
```

属性

名称	类型	说明
DD_INVALID	a_sqlany_data_direction	无效的数据方向。
DD_INPUT	a_sqlany_data_direction	仅用于输入的主机变量。
DD_OUTPUT	a_sqlany_data_direction	仅用于输出的主机变量。
DD_INPUT_OUTPUT	a_sqlany_data_direction	用于输入和输出的主机变量。

a_sqlany_data_type 枚举

指定传入或检索的数据类型。

语法

```
typedef enum a_sqlany_data_type
{
   A_INVALID_TYPE,
   A_BINARY,
   A_STRING,
   A_DOUBLE,
   A_VAL64,
   A_UVAL64,
   A_VAL32,
   A_UVAL32,
   A_UVAL16,
   A_VAL8,
   A_UVAL8
} a_sqlany_data_type
```

参数

名称	类型	说明
A_INVALID_TYPE	a_sqlany_data_type	无效数据类型。
A_BINARY	a_sqlany_data_type	二进制数据。直接处理二进制数据，不执行任何字符集转换。
A_STRING	a_sqlany_data_type	字符串数据。对其执行字符集转换的数据。
A_DOUBLE	a_sqlany_data_type	双字节数据。包括浮点值。
A_VAL64	a_sqlany_data_type	64 位整数。
A_UVAL64	a_sqlany_data_type	64 位无符号整数。
A_VAL32	a_sqlany_data_type	32 位整数。
A_UVAL32	a_sqlany_data_type	32 位无符号整数。
A_VAL16	a_sqlany_data_type	16 位整数。
A_UVAL16	a_sqlany_data_type	16 位无符号整数。
A_VAL8	a_sqlany_data_type	8 位整数。
A_UVAL8	a_sqlany_data_type	8 位无符号整数。

a_sqlany_native_type 枚举

嵌入式 SQL (ESQL) 数据类型的枚举。

语法

```
typedef enum a_sqlany_native_type
{
   DT_NOTYPE          = 0,
   DT_DATE            = 384,
   DT_TIME            = 388,
   DT_TIMESTAMP       = 392,
   DT_VARCHAR         = 448,
   DT_FIXCHAR         = 452,
   DT_LONGVARCHAR     = 456,
   DT_STRING          = 460,
   DT_DOUBLE          = 480,
   DT_FLOAT           = 482,
   DT_DECIMAL         = 484,
   DT_INT             = 496,
   DT_SMALLINT        = 500,
   DT_BINARY          = 524,
   DT_LONGBINARY      = 528,
   DT_TINYINT         = 604,
   DT_BIGINT          = 608,
   DT_UNSINT          = 612,
   DT_UNSSMALLINT     = 616,
   DT_UNSBIGINT       = 620,
   DT_BIT             = 624,
   DT_LONGNVARCHAR    = 640
} a_sqlany_native_type;
```

另请参见

- “a_sqlany_column_info 结构”一节第 613 页
- “sqlany_get_column_info 函数”一节第 602 页
- “嵌入式 SQL 数据类型”一节第 520 页

sacapi_error_size 常量

返回错误缓冲区大小。

语法

#define **SACAPI_ERROR_SIZE**

sqlany_current_api_version 常量

指示当前的 API 级别。

语法

#define**SQLANY_CURRENT_API_VERSION**

SQL Anywhere C API 示例

connecting.cpp

这是一个如何创建连接对象并将其连接到 SQL Anywhere 的示例。

```
// *************************************************************************
// Copyright 1994-2008 iAnywhere Solutions, Inc.  All rights reserved.
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original iAnywhere code.
//
// *************************************************************************
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "sacapidll.h"

int main( int argc, char * argv[] )
{
    SQLAnywhereInterface  api;
    a_sqlany_connection   *conn;
    unsigned int          max_api_ver;

    if( !sqlany_initialize_interface( &api, NULL ) ) {
        printf( "Could not initialize the interface!\n" );
        exit( 0 );
    }

    if( !api.sqlany_init( "MyAPP", SQLANY_CURRENT_API_VERSION,
&max_api_ver )) {
        printf( "Failed to initialize the interface! Supported version = %d
\n", max_api_ver );
        sqlany_finalize_interface( &api );
        return -1;
    }

    /* A connection object needs to be created first */
    conn = api.sqlany_new_connection();

    if( !api.sqlany_connect( conn, "uid=dba;pwd=sql" ) ) {
        /* failed to connect */
        char buffer[SACAPI_ERROR_SIZE];
        int  rc;
        rc = api.sqlany_error( conn, buffer, sizeof(buffer) );
        printf( "Failed to connect: error code=%d error message=%s\n",
            rc, buffer );
    } else {
        printf( "Connected successfully!\n" );
        api.sqlany_disconnect( conn );
    }

    /* Must free the connection object or there will be a memory leak */
    api.sqlany_free_connection( conn );

    api.sqlany_fini();
```



```
    sqlany_finalize_interface( &api );

    return 0;
}
```

dbcapi_isql.cpp

此示例说明如何使用 dbcapi 编写 ISQL 应用程序。

```
// ************************************************************************
// Copyright 1994-2008 iAnywhere Solutions, Inc.  All rights reserved.
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original iAnywhere code.
//
// ************************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>

#include "sacapidll.h"

#define max( x, y) ( x >= y ? x : y )

SQLAnywhereInterface api;
a_sqlany_connection *conn;


void print_blob( char * buffer, size_t length )
/*********************************************/
{
    size_t I;

    if( length == 0 ) {
        return;
    }
    printf( "0x" );
    I = 0;
    while( I < length ) {
        printf( "%.2X", (unsigned char)buffer[i] );
        I++;
    }
}


void execute( char * query )
/**************************/
{
    a_sqlany_stmt * stmt;
    int             err_code;
    char            err_mesg[SACAPI_ERROR_SIZE];
    int             I;
    int             num_rows;
    int             length;
```

```
    stmt = api.sqlany_execute_direct( conn, query );
    if( stmt == NULL ) {
        err_code = api.sqlany_error( conn, err_mesg, sizeof(err_mesg) );
        printf( "Failed: [%d] '%s'\n", err_code, err_mesg );
        return;
    }
    if( api.sqlany_error( conn, NULL, 0 ) > 0 ) {
        err_code = api.sqlany_error( conn, err_mesg, sizeof(err_mesg) );
        printf( "Warning: [%d] '%s'\n", err_code, err_mesg );
    }
    if( api.sqlany_num_cols( stmt ) == 0 ) {
        printf( "Executed successfully.\n" );
        if( api.sqlany_affected_rows( stmt ) > 0 ) {
            printf( "%d affected rows.\n",
api.sqlany_affected_rows( stmt ) );
        }
        api.sqlany_free_stmt( stmt );
        return;
    }

    // first output column header
    length = 0;
    for( I = 0; I < api.sqlany_num_cols( stmt ); I++ ) {
        a_sqlany_column_info    column_info;

        if( I > 0 ) {
            printf( "," );
            length += 1;
        }
        api.sqlany_get_column_info( stmt, I, &column_info );
        printf( "%s", column_info.name );
        length += (int)strlen( column_info.name );
    }
    printf( "\n" );
    for( I = 0; I < length; I++ ) {
        printf( "-" );
    }
    printf( "\n" );
    num_rows = 0;
    while( api.sqlany_fetch_next( stmt ) ) {
        num_rows++;
        for( I = 0; I < api.sqlany_num_cols( stmt ); I++ ) {
            a_sqlany_data_value dvalue;

            api.sqlany_get_column( stmt, I, &dvalue );
            if( I > 0 ) {
                printf( "," );
            }
            if( *(dvalue.is_null) ) {
                printf( "(NULL)" );
                continue;
            }
            switch( dvalue.type ) {
                case A_BINARY:
                    print_blob( dvalue.buffer, *(dvalue.length) );
                    break;
                case A_STRING:
                  printf( "'%.*s'", *(dvalue.length), (char *)dvalue.buffer,
*(dvalue.length) );
                    break;
                case A_VAL64:
                    printf( "%lld", *(long long*)dvalue.buffer);
                    break;
                case A_UVAL64:
```



```
                    printf( "%lld", *(unsigned long long*)dvalue.buffer);
                    break;
                case A_VAL32:
                    printf( "%d", *(int*)dvalue.buffer );
                    break;
                case A_UVAL32:
                    printf( "%u", *(unsigned int*)dvalue.buffer );
                    break;
                case A_VAL16:
                    printf( "%d", *(short*)dvalue.buffer );
                    break;
                case A_UVAL16:
                    printf( "%u", *(unsigned short*)dvalue.buffer );
                    break;
                case A_VAL8:
                    printf( "%d", *(char*)dvalue.buffer );
                    break;
                case A_UVAL8:
                    printf( "%d", *(char*)dvalue.buffer );
                    break;
                case A_DOUBLE:
                    printf( "%f", *(double*)dvalue.buffer );
                    break;
            }
        }
        printf( "\n" );
    }
    for( I = 0; I < length; I++ ) {
        printf( "-" );
    }
    printf( "\n" );

    printf( "%d rows returned\n", num_rows );
    if( api.sqlany_error( conn, NULL, 0 ) != 100 ) {
        char buffer[256];
        int  code = api.sqlany_error( conn, buffer, sizeof(buffer) );
        printf( "Failed: [%d] '%s'\n", code, buffer );
    }
    printf( "\n" );

    fflush( stdout );
    api.sqlany_free_stmt( stmt );
}


int main( int argc, char * argv[] )
/*********************************/
{
    unsigned int  max_api_ver;
    char          buffer[256];
    int           len;
    int           ch;

    if( argc < 1 ) {
        printf( "Usage: %s -c <connection_string>\n", argv[0] );
        exit( 0 );
    }
    if( !sqlany_initialize_interface( &api, NULL ) ) {
        printf( "Failed to initialize the interface!\n" );
        exit( 0 );
    }
   if( !api.sqlany_init( "isql", SQLANY_CURRENT_API_VERSION, &max_api_ver ))
{
        printf( "Failed to initialize the interface! Supported version = %d
```

```
\n", max_api_ver );
        sqlany_finalize_interface( &api );
        return -1;
    }
    conn = api.sqlany_new_connection();

    if( !api.sqlany_connect( conn, argv[1] ) ) {
        int code = api.sqlany_error( conn, buffer, sizeof(buffer) );
        printf( "Could not connect: [%d] %s\n", code, buffer );
        goto done;
    }

    printf( "Connected successfully!\n" );
    while( 1 ) {
        printf( "\n%s> ", argv[0] );
        fflush( stdout );

        len = 0;
        while( len < (sizeof(buffer) - 1) ) {
            ch = fgetc( stdin );
            if( ch == '\0' || ch == '\n' || ch == '\r' || ch == -1 ) {
                break;
            }
            buffer[len] = (char)tolower( ch );
            len++;
        }
        buffer[len] = '\0';
        if(buffer[0] == '\0' ) {
            break;
        }
        if( strcmp( buffer, "quit" ) == 0 ) {
            break;
        }
        execute( buffer );
    }

    api.sqlany_disconnect( conn );

done:
    api.sqlany_free_connection( conn );
    api.sqlany_fini();
    sqlany_finalize_interface( &api );
}
```

fetching_a_result_set.cpp

此示例说明如何从结果集中获取数据。

```
// *****************************************************************
// Copyright 1994-2008 iAnywhere Solutions, Inc.  All rights reserved.
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original iAnywhere code.
//
// *****************************************************************
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

```
#include "sacapidll.h"

int main( )
{
    SQLAnywhereInterface  api;
    a_sqlany_connection * conn;
    a_sqlany_stmt       * stmt;
    unsigned int          max_api_ver;

    if( !sqlany_initialize_interface( &api, NULL ) ) {
        printf( "Could not initialize the interface!\n" );
        exit( 0 );
    }

    if( !api.sqlany_init( "MyAPP", SQLANY_CURRENT_API_VERSION,
&max_api_ver ) ) {
        printf( "Failed to initialize the interface! Supported version=%d\n",
max_api_ver );
        sqlany_finalize_interface( &api );
        return -1;
    }

    /* A connection object needs to be created first */
    conn = api.sqlany_new_connection();

    if( !api.sqlany_connect( conn, "uid=dba;pwd=sql" ) ) {
        api.sqlany_free_connection( conn );
        api.sqlany_fini();
        sqlany_finalize_interface( &api );
        exit( -1 );
    }

    printf( "Connected successfully!\n" );


    if( (stmt = api.sqlany_execute_direct( conn, "select * from systable" )) !
= NULL ) {
        int                     num_rows = 0;
        a_sqlany_data_value     value;

        while( api.sqlany_fetch_next( stmt ) ) {

            num_rows++;
            printf( "\nRow [%d] data .......\n", num_rows );
            for( int I = 0; I < api.sqlany_num_cols( stmt ); I++ ) {
                int rc = api.sqlany_get_column( stmt, I, &value );

                if( rc < 0 ) {
                    printf( "Truncation of column %d\n", I );
                }
                if( *(value.is_null) ) {
                    printf( "Received a NULL value\n" );
                    continue;
                }

                switch( value.type ) {
                    case A_BINARY:
                        printf( "Binary value of length %d.\n",
*(value.length) );
                        break;
                    case A_STRING:
                        printf( "String value [%.*s] of length %d.\n",
                                *(value.length), (char *)value.buffer,
*(value.length) );
```

```
                        break;
                    case A_VAL64:
                        printf( "A_VAL64 value [%lld].\n", *(long long
*)value.buffer );
                        break;
                    case A_UVAL64:
                        printf( "A_UVAL64 value [%lld].\n", *(unsigned long
long *)value.buffer );
                        break;
                    case A_VAL32:
                        printf( "A_VAL32 value [%d].\n",
*(int*)value.buffer );
                        break;
                    case A_UVAL32:
                        printf( "A_UVAL32 value [%d].\n", *(unsigned
int*)value.buffer );
                        break;
                    case A_VAL16:
                        printf( "A_VAL16 value [%d].\n",
*(short*)value.buffer );
                        break;
                    case A_UVAL16:
                        printf( "A_UVAL16 value [%d].\n", *(unsigned
short*)value.buffer );
                        break;
                    case A_VAL8:
                        printf( "A_VAL8 value [%d].\n", *(char
*)value.buffer );
                        break;
                    case A_UVAL8:
                        printf( "A_UVAL8 value [%d].\n", *(unsigned char
*)value.buffer );
                        break;
                }
                /* do some processing with the data ... */
            }
        }

        /* Must free the result set object when done with it */
        api.sqlany_free_stmt( stmt );
    }
    api.sqlany_disconnect( conn );

    /* Must free the connection object or there will be a memory leak */
    api.sqlany_free_connection( conn );

    api.sqlany_fini();

    sqlany_finalize_interface( &api );

    return 0;
}
```

fetching_multiple_from_sp.cpp

此示例说明如何从存储过程中读取多个结果集。

```
// *************************************************************
// Copyright 1994-2008 iAnywhere Solutions, Inc.  All rights reserved.
// This sample code is provided AS IS, without warranty or liability
// of any kind.
```



```
//
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original iAnywhere code.
//
// *********************************************************************
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "sacapidll.h"

int main( )
{
    SQLAnywhereInterface  api;
    a_sqlany_connection * conn;
    a_sqlany_stmt       * stmt;
    unsigned int          max_api_ver;

    if( !sqlany_initialize_interface( &api, NULL ) ) {
        printf( "Could not initialize the interface!\n" );
        exit( 0 );
    }

    if( !api.sqlany_init( "MyAPP", SQLANY_CURRENT_API_VERSION,
&max_api_ver )) {
        printf( "Failed to initialize the interface! Supported version=%d\n",
max_api_ver );
        sqlany_finalize_interface( &api );
        return -1;
    }

    /* A connection object needs to be created first */
    conn = api.sqlany_new_connection();

    if( !api.sqlany_connect( conn, "uid=dba;pwd=sql" ) ) {
        api.sqlany_free_connection( conn );
        api.sqlany_fini();
        sqlany_finalize_interface( &api );
        exit( -1 );
    }

    printf( "Connected successfully!\n" );

    api.sqlany_execute_immediate( conn, "drop procedure myproc" );
    api.sqlany_execute_immediate( conn,
            "create procedure myproc( ) \n"
            "begin                \n"
            "    select 1, 2;     \n"
            "    select 3, 4, 5;\n"
            "end                  \n" );


    if( (stmt = api.sqlany_execute_direct( conn, "call myproc()" )) != NULL )
{
        do {
            /* fetch one row at a time */
            while( api.sqlany_fetch_next( stmt ) ) {

                /* sqlany_num_cols() will be updated everytime the result set
shape changes */
                for( int I = 0; I < api.sqlany_num_cols( stmt ); I++ ) {
                    /* process data here ... */
                }
            }
```

```
            /* Check to see if there are other result sets */
        } while( api.sqlany_get_next_result( stmt ) );

        /* Must free the result set object when done with it */
        api.sqlany_free_stmt( stmt );
    }
    api.sqlany_disconnect( conn );

    /* Must free the connection object or there will be a memory leak */
    api.sqlany_free_connection( conn );

    api.sqlany_fini();

    sqlany_finalize_interface( &api );

    return 0;
}
```

preparing_statements.cpp

此示例说明如何准备和执行语句。

```
// *************************************************************************
// Copyright 1994-2008 iAnywhere Solutions, Inc.  All rights reserved.
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original iAnywhere code.
//
// *************************************************************************
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "sacapidll.h"
#include <assert.h>

int main( )
{
    SQLAnywhereInterface  api;
    a_sqlany_connection * conn;
    a_sqlany_stmt       * stmt;
    unsigned int          max_api_ver;

    if( !sqlany_initialize_interface( &api, NULL ) ) {
        printf( "Could not initialize the interface!\n" );
        exit( 0 );
    }

    if( !api.sqlany_init( "MyAPP", SQLANY_CURRENT_API_VERSION,
&max_api_ver )) {
        printf( "Failed to initialize the interface! Supported version=%d\n",
max_api_ver );
        sqlany_finalize_interface( &api );
        return -1;
    }

    /* A connection object needs to be created first */
    conn = api.sqlany_new_connection();
```

```
    if( !api.sqlany_connect( conn, "pktdump=c:\\temp\
\pktdump;uid=dba;pwd=sql" ) ) {
        api.sqlany_free_connection( conn );
        api.sqlany_fini();
        sqlany_finalize_interface( &api );
        exit( -1 );
    }

    printf( "Connected successfully!\n" );

    api.sqlany_execute_immediate( conn, "drop procedure myproc" );
    api.sqlany_execute_immediate( conn,
            "create procedure myproc ( IN prefix char(10),       \n"
            "                          INOUT buffer varchar(256),\n"
            "                          OUT str_len int,          \n"
            "                          IN suffix char(10) ) \n"
            "begin                                               \n"
            "    set buffer = prefix || buffer || suffix;\n"
            "    select length( buffer ) into str_len;   \n"
            "end                                                 \n" );
    stmt = api.sqlany_prepare( conn, "call myproc( ?, ?, ?, ? )" );

    if( stmt ) {

        a_sqlany_bind_param     param;
        char                    buffer[256] = "-some_string-";
        int                     str_len;
        size_t                  buffer_size = strlen(buffer);
        size_t                  prefix_length = 6;
        size_t                  suffix_length = 6;

        assert( api.sqlany_describe_bind_param( stmt, 0, &param ) );
        param.value.buffer = "PREFIX";
        param.value.length = &prefix_length;
        assert( api.sqlany_bind_param( stmt, 0, &param ) );

        assert( api.sqlany_describe_bind_param( stmt, 1, &param ) );
        param.value.buffer = buffer;
        param.value.length = &buffer_size;
        //params[1].value.type      = A_STRING;           // already set by
sqlany_describe_bind_param()
        //params[1].direction       = INPUT_OUTPUT;       // already set by
sqlany_describe_bind_param()
        param.value.buffer_size = sizeof(buffer);         // IMPORTANT: this
field must be set for
                                                          // OUTPUT and
INPUT_OUTPUT parameters so that
                                                         // the library knows
how much data can be written
                                                          // into the buffer
        assert( api.sqlany_bind_param( stmt, 1, &param ) );

        assert( api.sqlany_describe_bind_param( stmt, 2, &param ) );
        param.value.buffer      = (char *)&str_len;
        param.value.is_null     = NULL;                   // use NULL if not
interested in nullability
        //param.value.type          = A_VAL32;            // already set by
sqlany_describe_bind_param()
        //param.direction           = OUTPUT_ONLY;        // already set by
sqlany_describe_bind_param()
        //param.value.buffer_size = sizeof(str_len);      // for non string
or binary buffers, buffer_size is not needed
        assert( api.sqlany_bind_param( stmt, 2, &param ) );
```

```
        assert( api.sqlany_describe_bind_param( stmt, 3, &param ) );
        param.value.buffer      = "SUFFIX";
        param.value.length      = &suffix_length;
        //params.value.type     = A_STRING;               // already set by
sqlany_describe_bind_param()
        assert( api.sqlany_bind_param( stmt, 3, &param ) );

       /* We are not expecting a result set so the result set parameter could
be NULL */
        if( api.sqlany_execute( stmt ) ) {
            printf( "Complete string is %s and is %d chars long \n", buffer,
str_len );
            assert( str_len == (6+13+6) );

            buffer_size = str_len;
            api.sqlany_execute( stmt );
            printf( "Complete string is %s and is %d chars long \n", buffer,
str_len );
            assert( str_len == 6+(6+13+6)+6 );
        } else {
            char buffer[SACAPI_ERROR_SIZE];
            int  rc;
            rc = api.sqlany_error( conn, buffer, sizeof(buffer));
            printf( "Failed to execute! [%d] %s\n", rc, buffer );
        }

        /* Free the statement object or there will be a memory leak */
        api.sqlany_free_stmt( stmt );
    }

    api.sqlany_disconnect( conn );

    /* Must free the connection object or there will be a memory leak */
    api.sqlany_free_connection( conn );

    api.sqlany_fini();

    sqlany_finalize_interface( &api );

    return 0;
}
```

send_retrieve_full_blob.cpp

此示例说明如何在一个块中插入和检索 blob。

```
// *************************************************************************
// Copyright 1994-2008 iAnywhere Solutions, Inc.  All rights reserved.
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original iAnywhere code.
//
// *************************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include "sacapidll.h"
```



```
int main( )
{
    SQLAnywhereInterface api;
    a_sqlany_connection *conn;
    a_sqlany_stmt       *stmt;
    unsigned int         I;
    unsigned char       *data;
    size_t               size = 1024*1024; // 1MB blob
    int                  code;
    a_sqlany_data_value  value;
    int                  num_cols;
    unsigned int         max_api_ver;
    a_sqlany_bind_param  param;

    if( !sqlany_initialize_interface( &api, NULL ) ) {
        printf( "Could not initialize the interface!\n" );
        exit( 0 );
    }

    assert( api.sqlany_init( "my_php_app", SQLANY_CURRENT_API_VERSION,
&max_api_ver ) );
    conn = api.sqlany_new_connection();

    if( !api.sqlany_connect( conn, "uid=dba;pwd=sql" ) ) {
        char buffer[SACAPI_ERROR_SIZE];
        code = api.sqlany_error( conn, buffer, sizeof(buffer) );
        printf( "Could not connection[%d]:%s\n", code, buffer );
        goto clean;
    }

    printf( "Connected successfully!\n" );

    api.sqlany_execute_immediate( conn, "drop table my_blob_table" );
    assert( api.sqlany_execute_immediate( conn, "create table my_blob_table
(size integer, data long binary)" ) != 0);

    stmt = api.sqlany_prepare( conn, "insert into my_blob_table( size, data )
values( ?, ?)" );
    assert( stmt != NULL );

    data = (unsigned char *)malloc( size );
    // initialize the buffer
    for( I = 0; I < size; I++ ) {
        data[i] = I % 256;
    }


    // initialize the parameters
    api.sqlany_describe_bind_param( stmt, 0, &param );
    param.value.buffer = (char *)&size;
    param.value.type   = A_VAL32;              // This needs to be set as the
server does not
    // know what data will be inserting.
    api.sqlany_bind_param( stmt, 0, &param );

    api.sqlany_describe_bind_param( stmt, 1, &param );
    param.value.buffer = (char *)data;
    param.value.length = &size;
    param.value.type   = A_BINARY;               // This needs to be set for
the same reason as above.
    api.sqlany_bind_param( stmt, 1, &param );

    assert( api.sqlany_execute( stmt ) );
```

```
    api.sqlany_free_stmt( stmt );

    api.sqlany_commit( conn );

    stmt = api.sqlany_execute_direct( conn, "select * from my_blob_table" );
    assert( stmt != NULL );

    assert( api.sqlany_fetch_next( stmt ) == 1 );

    num_cols = api.sqlany_num_cols( stmt );

    assert( num_cols == 2 );

    api.sqlany_get_column( stmt, 0, &value );

    assert( *((int*)value.buffer) == size );
    assert( value.type == A_VAL32 );

    api.sqlany_get_column( stmt, 1, &value );

    assert( value.type == A_BINARY );
    assert( *(value.length) == size );

    for( I = 0; I < (*value.length); I++ ) {
        assert( (unsigned char)(value.buffer[i]) == data[i]);
    }

    assert( api.sqlany_fetch_next( stmt ) == 0 );
    api.sqlany_free_stmt( stmt );

    api.sqlany_disconnect( conn );

clean:
    api.sqlany_free_connection( conn );

    api.sqlany_fini();

    sqlany_finalize_interface( &api );
    printf( "Success!\n" );
}
```

send_retrieve_part_blob.cpp

此示例说明如何在多个块中插入 blob，以及如何在多个块中检索它。

```
// *************************************************************
// Copyright 1994-2008 iAnywhere Solutions, Inc.  All rights reserved.
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original iAnywhere code.
//
// *************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include "sacapidll.h"
```

```
int main( )
{
    SQLAnywhereInterface api;
    a_sqlany_connection *conn;
    a_sqlany_stmt       *stmt;
    unsigned int         I;
    unsigned char       *data;
    unsigned int         size = 1024*1024; // 1MB blob
    int                  code;
    a_sqlany_data_value  value;
    int                  num_cols;
    unsigned char        retrieve_buffer[4096];
    a_sqlany_data_info   dinfo;
    int                  bytes_read;
    size_t               total_bytes_read;
    unsigned int         max_api_ver;

    if( !sqlany_initialize_interface( &api, NULL ) ) {
        printf( "Could not initialize the interface!\n" );
        exit( 0 );
    }

    assert( api.sqlany_init( "my_php_app", SQLANY_CURRENT_API_VERSION,
&max_api_ver ) );
    conn = api.sqlany_new_connection();

    if( !api.sqlany_connect( conn, "uid=dba;pwd=sql" ) ) {
        char buffer[SACAPI_ERROR_SIZE];
        code = api.sqlany_error( conn, buffer, sizeof(buffer) );
        printf( "Could not connection[%d]:%s\n", code, buffer );
        goto clean;
    }

    printf( "Connected successfully!\n" );

    api.sqlany_execute_immediate( conn, "drop table my_blob_table" );
    assert( api.sqlany_execute_immediate( conn, "create table my_blob_table
(size integer, data long binary)" ) != 0);

    // 1. Starting to insert blob operation
    stmt = api.sqlany_prepare( conn, "insert into my_blob_table( size, data)
values( ?, ? )" );
    assert( stmt != NULL );


    // 1.1 We must first bind the parameters
    a_sqlany_bind_param param;

    api.sqlany_describe_bind_param( stmt, 0, &param );
    param.value.buffer = (char *)&size;
    param.value.type   = A_VAL32;
    param.value.is_null= NULL;
    param.direction    = DD_INPUT;
    api.sqlany_bind_param( stmt, 0, &param );

    api.sqlany_describe_bind_param( stmt, 1, &param );
    param.value.buffer = NULL;
    param.value.type   = A_BINARY;
    param.value.is_null= NULL;
    param.direction    = DD_INPUT;
    api.sqlany_bind_param( stmt, 1, &param );

    data = (unsigned char *)malloc( size );
```

```
        for( I = 0; I < size; I++ ) {
            data[i] = I % 256;
        }

        // 1.2 upload the blob data to the server in chunks
        for( I = 0; I < size; I += 4096 ) {
            if( !api.sqlany_send_param_data( stmt, 1, (char *)&data[i], 4096 )) {
                char buffer[SACAPI_ERROR_SIZE];
                code = api.sqlany_error( conn, buffer, sizeof(buffer) );
                printf( "Could not send param[%d]:%s\n", code, buffer );
            }
        }

        // 1.3 actually do the row insert operation
        assert( api.sqlany_execute( stmt ) == 1 );

        api.sqlany_commit( conn );

        api.sqlany_free_stmt( stmt );


        // 2. Now let's retrieve the blob
        stmt = api.sqlany_execute_direct( conn, "select * from my_blob_table" );
        assert( stmt != NULL );

        assert( api.sqlany_fetch_next( stmt ) == 1 );

        num_cols = api.sqlany_num_cols( stmt );

        assert( num_cols == 2 );

        api.sqlany_get_column( stmt, 0, &value );

        assert( I == size );
        assert( value.type == A_VAL32 );

        api.sqlany_get_data_info( stmt, 1, &dinfo );

        assert( dinfo.type == A_BINARY );
        assert( dinfo.data_size == size );
        assert( dinfo.is_null == 0 );

        // 2.1 Retrieve data in 4096 byte chunks
        total_bytes_read = 0;
        while( 1 ) {
            bytes_read = api.sqlany_get_data( stmt, 1, total_bytes_read,
    retrieve_buffer, sizeof(retrieve_buffer) );
            if( bytes_read <= 0 ) {
                break;
            }
            // verify the buffer contents
            for( I = 0; I < (unsigned int)bytes_read; I++ ) {
                assert( retrieve_buffer[i] == data[total_bytes_read+I] );
            }
            total_bytes_read += bytes_read;
        }
        assert( total_bytes_read == size );

        free(data );

        assert( api.sqlany_fetch_next( stmt ) == 0 );

        api.sqlany_free_stmt( stmt );
```



```
    api.sqlany_disconnect( conn );

clean:
    api.sqlany_free_connection( conn );

    api.sqlany_fini();

    sqlany_finalize_interface( &api );

    printf( "Success!\n" );
}
```

SQL Anywhere 外部函数 API

目录

从过程调用外部库

可以从存储过程或函数调用外部库中的函数。可以调用 DLL（Windows 操作系统）和共享对象 (Unix) 中的函数。不能在 Windows Mobile 上调用外部函数。

本节介绍如何使用外部库调用 API。外部存储过程示例以及构建含这些过程的 DLL 所需的文件位于以下文件夹中：*samples-dir\SQLAnywhere\ExternalProcedures*。有关 *samples-dir* 位置的信息，请参见“示例目录”一节 《SQL Anywhere 服务器 - 数据库管理》。

小心
从过程调用的外部库共享服务器的内存。如果您从过程调用外部库并且该外部库包含内存处理错误，则可能会造成该服务器崩溃或损坏数据库。在生产数据库上部署库之前，务必对这些库进行彻底测试。

本节中所述的 API 代替了早期的 API。建议不要使用早期的 API。针对早期 API 所编写的库（在 7.0.x 之前的版本中使用）仍然是支持的，但在进行新的开发时，建议您使用新的 API。请注意，对所有 Unix 平台和所有 64 位平台（包括 64 位 Windows），必须使用新 API。

SQL Anywhere 包括一组可以利用此功能来实现某些操作（例如发送 MAPI 电子邮件消息）的系统过程。请参见“MAPI 和 SMTP 过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

创建具有外部调用的过程和函数

本节提供了一些具有外部调用的过程和函数的示例。

需要 DBA 权限
要创建引用外部库的过程或函数，您必须具有 DBA 权限。这一要求比创建其它过程或函数所需的 RESOURCE 权限更为严格。

语法

可以创建一个调用库（动态链接库（Dynamic Link Library，简称 DLL）或共享对象）中的 C/C++ 函数的 SQL 存储过程，如下所示：

```
CREATE PROCEDURE coverProc( parameter-list )
  EXTERNAL NAME 'myFunction@myLibrary'
  LANGUAGE C_ESQL32;
```

以这种方式定义存储过程或函数时，将创建一个连接到外部 DLL 中函数的桥。该存储过程或函数不能执行任何其它任务。

同样，还可以创建一个调用库中的 C/C++ 函数的 SQL 存储函数，如下所示：

```
CREATE FUNCTION coverFunc( parameter-list )
  RETURNS data-type
  EXTERNAL NAME 'myFunction@myLibrary'
  LANGUAGE C_ESQL32;
```

在这些语句中，EXTERNAL NAME 子句指示函数名及函数所在的库。在此示例中，`myFunction` 是库中函数的导出名称；`myLibrary` 是库的名称（例如，*myLibrary.dll* 或 *myLibrary.so*）。

LANGUAGE 子句指示将在外部环境中调用函数。LANGUAGE 子句可以指定 C_ESQL32、C_ESQL64、C_ODBC32 或 C_ODBC64 之一。后缀 **32** 或 **64** 指示函数将被编译为 32 位或 64 位应用程序。**ODBC** 标志指示应用程序使用 ODBC API。**ESQL** 标志指示应用程序可能使用嵌入式 SQL API、SQL Anywhere C API、任何其它非 ODBC API，或者根本不使用 API。

如果忽略 LANGUAGE 子句，则包含此函数的库将被装载到数据库服务器的地址空间中。调用时，外部函数将作为服务器的一部分执行。在这种情况下，如果此函数导致故障，则会使数据库服务器终止。因此，建议在外部环境中装载和执行函数。如果函数在外部环境中导致故障，数据库服务器仍可继续运行。

parameter-list 中的参数必须在类型和顺序上与库函数所需的参数相符。库函数使用在“外部函数原型”一节第 645 页中介绍的 API 来访问过程参数。

外部函数所返回的任何值或结果集都可以由存储过程或函数返回给调用环境。

不允许其它语句

引用外部函数的存储过程或函数不可以包括任何其它语句：其唯一目的在于获取函数的参数、调用函数，并将任何值和从函数所返回的参数返回到调用环境。您可以采用与其它过程相同的方式在过程调用中使用 IN、INOUT 或 OUT 参数：将输入值传递给外部函数，由函数所修改的任何参数都将在 OUT 或 INOUT 参数中或者作为存储函数的 RETURNS 结果返回给调用环境。

系统相关的调用

可以指定与操作系统相关的调用，使过程在一个操作系统上运行时调用一个函数，在另一个操作系统上运行时调用另一个函数（大概类似）。此类调用的语法需要将操作系统名称作为函数名称的前缀。操作系统标识符必须是 Unix。下面是一个示例。

```
CREATE FUNCTION func ( parameter-list )
  RETURNS data-type
  EXTERNAL NAME 'Unix:function-name@library.so;function-name@library.dll';
```

如果函数列表不包含运行服务器所在的操作系统的条目，但该列表确实包含未指定操作系统的条目，则数据库服务器调用该条目中的函数。

另请参见

- “CREATE PROCEDURE 语句（Web 服务）”一节 《SQL Anywhere 服务器 - SQL 参考》
- “CREATE FUNCTION 语句（Web 服务）”一节 《SQL Anywhere 服务器 - SQL 参考》
- “SQL Anywhere 外部环境支持”第 661 页

外部函数原型

本节描述以 C 或 C++ 语言编写的函数所使用的 API。

该 API 由 SQL Anywhere 安装目录 *SDK\Include* 子目录中名为 *extfnapi.h* 的头文件定义。此头文件处理外部函数原型的与平台相关的功能。

函数原型

函数的名称必须与 CREATE PROCEDURE 或 CREATE FUNCTION 语句所引用的名称匹配。假设已执行了以下 CREATE FUNCTION 语句。

```
CREATE FUNCTION cover-name ( parameter-list )
  RETURNS data-type
  EXTERNAL NAME 'function-name@library.dll'
  LANGUAGE C_ESQL32;
```

C/C++ 函数声明必须如下所示：

void *function-name*(**an_extfn_api** **api*, **void** **argument-handle*)

函数必须返回 void，其参数必须有两个，一个是用于调用一组回调函数的结构指针，另一个是 SQL 过程所提供参数的句柄。

extfn_use_new_api 方法

要将外部库是使用外部函数调用 API 编写的这一情况通知给数据库服务器，外部库必须导出以下函数：

语法

a_sql_uint32 extfn_use_new_api( );

返回值

该函数返回 32 位无符号整数。返回值必须是在 *extfnapi.h* 中定义的 API 版本号 EXTFN_API_VERSION。返回值为 0 表示正在使用的是旧 API。

注释

如果不通过库导出该函数，则数据库服务器就会认为正在使用旧 API。对所有 Unix 平台和所有 64 位平台（包括 64 位 Windows），必须使用新 API。

此函数的典型实现方式如下：

```
a_sql_uint32 extfn_use_new_api( void )
{
    return( EXTFN_API_VERSION );
}
```

另请参见

- “an_extfn_api 结构”一节第 646 页

extfn_cancel 方法

要将外部库支持取消处理这一情况通知给数据库服务器，外部库必须导出以下函数：

语法

void extfn_cancel(void *cancel_handle);

参数

- **cancel_handle** 指向要操纵的变量的指针。

注释

无论何时取消当前正在执行的 SQL 语句，数据库服务器都会异步调用此函数。

函数使用 cancel_handle 设置一个标志，来将 SQL 语句已取消这一情况通知外部库函数。

如果不通过库导出该函数，则数据库服务器就会认为取消处理是不被支持的。

此函数的典型实现方式如下：

```
void extfn_cancel( void *cancel_handle )
{
    *(short *)cancel_handle = 1;
}
```

另请参见

- “an_extfn_api 结构”一节第 646 页

an_extfn_api 结构

用于与调用 SQL 环境通信。

语法

```
typedef struct an_extfn_api {
    short (SQL_CALLBACK *get_value)(
                void *          arg_handle,
                a_sql_uint32    arg_num,
                an_extfn_value *value
                );
    short (SQL_CALLBACK *get_piece)(
                void *          arg_handle,
                a_sql_uint32    arg_num,
                an_extfn_value *value,
                a_sql_uint32    offset
                );
    short (SQL_CALLBACK *set_value)(
                void *          arg_handle,
                a_sql_uint32    arg_num,
```



```
                an_extfn_value *value
                short           append
                );
    void (SQL_CALLBACK *set_cancel)(
                void *          arg_handle,
                void *          cancel_handle
                );
} an_extfn_api;
```

属性

- **get_value** 使用此回调函数获取指定参数的值。以下示例获取参数 1 的值。

```
result = extapi->get_value( arg_handle, 1, &arg )
if( result == 0 || arg.data == NULL )
{
    return; // no parameter or parameter is NULL
}
```

- **get_piece** 使用此回调函数获取指定参数的下一个值块（如果存在）。以下示例获取参数 1 的其余部分。

```
cmd = (char *)malloc( arg.len.total_len + 1 );
offset = 0;
for( ; result != 0; )
{
     if( arg.data == NULL ) break;
     memcpy( &cmd[offset], arg.data, arg.piece_len );
     offset += arg.piece_len;
     cmd[offset] = '\0';
     if( arg.piece_len == 0 ) break;
     result = extapi->get_piece( arg_handle, 1, &arg, offset );
}
```

- **set_value** 使用此回调函数设置指定参数的值。以下示例为 FUNCTION 的 RETURNS 子句设置返回值（参数 0）。

```
an_extfn_value      retval;
int ret = -1;

// set up the return value struct
retval.type = DT_INT;
retval.data = (void*) &ret;
retval.piece_len = retval.len.total_len =
    (a_sql_uint32) sizeof( int );
extapi->set_value( arg_handle, 0, &retval, 0 );
```

- **set_cancel** 使用此回调函数建立一个指向可由 **extfn_cancel** 方法设置的变量的指针。示例如下。

```
short               canceled = 0;
extapi->set_cancel( arg_handle, &canceled );
```

注释

调用程序会将指向 an_extfn_api 结构的指针传递到外部函数。下面是一个示例：

```
extern "C" __declspec( dllexport )
void my_external_proc( an_extfn_api *extapi, void *arg_handle )
{
    short               result;
```

```
    short               canceled;
    an_extfn_value      arg;

    canceled = 0;
    extapi->set_cancel( arg_handle, &canceled );

    result = extapi->get_value( arg_handle, 1, &arg );
    if( canceled || result == 0 || arg.data == NULL )
    {
        return; // no parameter or parameter is NULL
    }
.
.
.
}
```

每当使用上述的任意一个回调函数时，都必须传递回作为第二个参数传递到外部函数的参数句柄。

另请参见

- “an_extfn_value 结构”一节第 648 页
- “extfn_cancel 方法”一节第 646 页

an_extfn_value 结构

用于从调用 SQL 环境访问参数数据。

语法

```
typedef struct an_extfn_value {
    void *              data;
    a_sql_uint32        piece_len;
    union {
        a_sql_uint32 total_len;
        a_sql_uint32 remain_len;
    } len;
    a_sql_data_type  type;
} an_extfn_value;
```

属性

- **data** 指向此参数数据的指针。
- **piece_len** 参数本段的长度。其小于或等于 **total_len**。
- **total_len** 参数的总长度。对于字符串，它表示字符串的长度且不包括空终止符。调用 **get_value** 回调函数后，即会设置此属性。调用 **get_piece** 回调函数后，此属性不再有效。
- **remain_len** 当分段获取参数时，其为尚未获取的部分的长度。每次调用 **get_piece** 回调函数后，均会设置此属性。
- **type** 指示参数的类型。它是嵌入式 SQL 数据类型之一，如 **DT_INT**、**DT_FIXCHAR** 或 **DT_BINARY**。请参见“嵌入式 SQL 数据类型”一节第 520 页。

注释

假设外部函数接口是使用以下 SQL 语句描述的。


```
CREATE FUNCTION mystring( IN instr LONG VARCHAR )
RETURNS LONG VARCHAR
EXTERNAL NAME 'mystring@c:\\project\\mystring.dll';
```

以下代码段显示了如何访问类型为 **an_extfn_value** 的对象的属性。在本示例中，此函数 (mystring) 的输入参数 1 (instr) 应为 SQL LONGVARCHAR 字符串。

```
an_extfn_value      arg;

result = extapi->get_value( arg_handle, 1, &arg );
if( result == 0 || arg.data == NULL )
{
    return; // no parameter or parameter is NULL
}

if( arg.type != DT_LONGVARCHAR )
{
    return; // unexpected type of parameter
}

cmd = (char *)malloc( arg.len.total_len + 1 );
offset = 0;
for( ; result != 0; )
{
     if( arg.data == NULL ) break;
     memcpy( &cmd[offset], arg.data, arg.piece_len );
     offset += arg.piece_len;
     cmd[offset] = '\0';
     if( arg.piece_len == 0 ) break;
     result = extapi->get_piece( arg_handle, 1, &arg, offset );
}
```

另请参见

- “an_extfn_api 结构”一节第 646 页

an_extfn_result_set_info 结构

便于返回调用 SQL 环境的结果集。

语法

```
typedef struct an_extfn_result_set_info {
    a_sql_uint32                        number_of_columns;
    an_extfn_result_set_column_info     *column_infos;
    an_extfn_result_set_column_data     *column_data_values;
} an_extfn_result_set_info;
```

属性

- **number_of_columns** 结果集中的列数。
- **column_infos** 链接到结果集列的说明。请参见“an_extfn_result_set_column_info 结构”一节第 650 页。
- **column_data_values** 链接到结果集列数据的说明。请参见“an_extfn_result_set_column_data 结构”一节第 651 页。

注释

以下代码段显示了如何设置此类型对象的属性。

```
int columns = 2;
an_extfn_result_set_info    rs_info;

an_extfn_result_set_column_info *col_info =
    (an_extfn_result_set_column_info *)
    malloc( columns * sizeof(an_extfn_result_set_column_info) );

an_extfn_result_set_column_data *col_data =
    (an_extfn_result_set_column_data *)
    malloc( columns * sizeof(an_extfn_result_set_column_data) );

rs_info.number_of_columns   = columns;
rs_info.column_infos        = col_info;
rs_info.column_data_values  = col_data;
```

另请参见

- “an_extfn_result_set_column_info 结构”一节第 650 页
- “an_extfn_result_set_column_data 结构”一节第 651 页

an_extfn_result_set_column_info 结构

用于描述结果集。

语法

```
typedef struct an_extfn_result_set_column_info {
    char *                          column_name;
    a_sql_data_type                 column_type;
    a_sql_uint32                    column_width;
    a_sql_uint32                    column_index;
    short int                       column_can_be_null;
} an_extfn_result_set_column_info;
```

属性

- **column_name** 指向列的名称（以空值终止的字符串）。
- **column_type** 指示列的类型。它是嵌入式 SQL 数据类型之一，如 **DT_INT**、**DT_FIXCHAR** 或 **DT_BINARY**。请参见“嵌入式 SQL 数据类型”一节第 520 页。
- **column_width** 定义 char(n)、varchar(n) 和 binary(n) 声明的最大宽度，对于所有其它类型均设置为 0。
- **column_index** 列的顺序位置，从 1 开始。
- **column_can_be_null** 如果列可以为空，则设置为 1；否则，设置为 0。

注释

以下代码段显示了如何设置此类型对象的属性，以及如何描述调用 SQL 环境的结果集。

```
// set up column descriptions
// DepartmentID          INTEGER NOT NULL
col_info[0].column_name  = "DepartmentID";
```



```
col_info[0].column_type  = DT_INT;
col_info[0].column_width = 0;
col_info[0].column_index = 1;
col_info[0].column_can_be_null = 0;

// DepartmentName        CHAR(40) NOT NULL
col_info[1].column_name  = "DepartmentName";
col_info[1].column_type  = DT_FIXCHAR;
col_info[1].column_width = 40;
col_info[1].column_index = 2;
col_info[1].column_can_be_null = 0;

extapi->set_value( arg_handle,
                EXTFN_RESULT_SET_ARG_NUM,
                (an_extfn_value *)&rs_info,
                EXTFN_RESULT_SET_DESCRIBE );
```

另请参见

- “an_extfn_result_set_info 结构”一节第 649 页
- “an_extfn_result_set_column_data 结构”一节第 651 页

an_extfn_result_set_column_data 结构

用于返回列的数据值。

语法

```
typedef struct an_extfn_result_set_column_data {
    a_sql_uint32                    column_index;
    void *                          column_data;
    a_sql_uint32                    data_length;
    short                           append;
} an_extfn_result_set_column_data;
```

属性

- **column_index** 列的顺序位置，从 1 开始。
- **column_data** 指向包含列数据的缓冲区的指针。
- **data_length** 数据的实际长度。
- **append** 用于返回块中的列值。返回部分列值时，设置为 1；否则，设置为 0。

注释

以下代码段显示了如何设置此类型对象的属性，以及如何将结果集行返回到调用 SQL 环境。

```
int DeptNumber = 400;
char * DeptName = "Marketing";

col_data[0].column_index = 1;
col_data[0].column_data  = &DeptNumber;
col_data[0].data_length  = sizeof( DeptNumber );
col_data[0].append  = 0;

col_data[1].column_index = 2;
col_data[1].column_data  = DeptName;
```

```
col_data[1].data_length  = strlen(DeptName);
col_data[1].append  = 0;

extapi->set_value(  arg_handle,
                EXTFN_RESULT_SET_ARG_NUM,
                (an_extfn_value *)&rs_info,
                EXTFN_RESULT_SET_NEW_ROW_FLUSH );
```

另请参见

- “an_extfn_result_set_info 结构”一节第 649 页
- “an_extfn_result_set_column_info 结构”一节第 650 页

使用外部函数调用 API 方法

get_value 回调

```
short (SQL_CALLBACK *get_value)
(
    void *          arg_handle,
    a_sql_uint32    arg_num,
    an_extfn_value *value
);
```

get_value 回调函数可用于获取传递给作为外部函数接口的存储过程或函数的参数值。如果失败则返回 0；否则返回非零结果。调用 get_value 之后，an_extfn_value 结构的 total_len 字段会包含整个值的长度。piece_len 字段包含作为 get_value 调用结果所获取的那部分的长度。请注意，piece_len 将始终小于或等于 otal_len。小于时，可以调用另一个函数 **get_piece** 来获得其余部分。请注意，total_len 字段只在初始调用 get_value 后才有效。该字段会由通过调用 get_piece 而更改的 remain_len 字段覆盖。调用 get_value 后，如果打算在以后使用 total_len 字段的值，则应立即将其保留下来。

get_piece 回调

```
short (SQL_CALLBACK *get_piece)
(
    void *          arg_handle,
    a_sql_uint32    arg_num,
    an_extfn_value *value,
    a_sql_uint32    offset
);
```

如果不能一次性返回整个参数值，则可以反复调用 **get_piece** 函数来获取参数值的其余部分。

通过调用 get_value 和 get_piece 所返回的所有 piece_len 值的总和将等于调用 get_value 之后 total_len 字段所返回的初始值。调用 get_piece 之后，remain_len 字段会覆盖 total_len，它表示尚未获取的长度。

使用 get_value 和 get_piece 回调

下面的示例显示如何使用 get_value 和 get_piece 来获取字符串参数的值，如 long varchar 型参数。

假设按如下方式声明了外部函数的包装：

```
CREATE PROCEDURE mystring( IN instr LONG VARCHAR )
  EXTERNAL NAME 'mystring@mystring.dll';
```

为从 SQL 调用外部函数，我们将使用如下的语句。

```
call mystring('Hello world!');
```

下面是以 C 语言编写的 mystring 函数针对 Windows 操作系统的实现示例：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>
#include "extfnapi.h"
```

```
BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
      )
{
    return TRUE;
}

extern "C" __declspec( dllexport )
a_sql_uint32 extfn_use_new_api( void )
{
    return( EXTFN_API_VERSION );
}

extern "C" __declspec( dllexport )
void mystring( an_extfn_api *extapi, void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    unsigned            offset;
    char                *string;

    result = extapi->get_value( arg_handle, 1, &arg );
    if( result == 0 || arg.data == NULL )
    {
        return; // no parameter or parameter is NULL
    }
    string = (char *)malloc( arg.len.total_len + 1 );
    offset = 0;
    for( ; result != 0; ) {
         if( arg.data == NULL ) break;
         memcpy( &string[offset], arg.data, arg.piece_len );
         offset += arg.piece_len;
         string[offset] = '\0';
         if( arg.piece_len == 0 ) break;
         result = extapi->get_piece( arg_handle, 1, &arg, offset );
    }
    MessageBoxA( NULL, string,
          "SQL Anywhere",
          MB_OK | MB_TASKMODAL );
    free( string );
    return;
}
```

set_value 回调

```
short (SQL_CALLBACK *set_value)
(
    void *         arg_handle,
    a_sql_uint32   arg_num,
    an_extfn_value *value
    short          append
);
```

set_value 回调函数可用于设置 OUT 参数和存储函数的 RETURNS 结果。arg_num 值为 0 时表示设置 RETURNS 值。以下是一个示例。

```
an_extfn_value      retval;

retval.type = DT_LONGVARCHAR;
retval.data = result;
```



```
retval.piece_len = retval.len.total_len = (a_sql_uint32) strlen( result );
extapi->set_value( arg_handle, 0, &retval, 0 );
```

set_value 的 append 参数决定所提供的数据是替换 (false) 现有数据，还是附加到 (true) 现有数据之后。在用 append=TRUE 为同一参数调用 set_value 前，必须使用 append=FALSE 调用它。对于固定长度的数据类型，将忽略 append 参数。

要返回 NULL，请将 an_extfn_value 结构的数据字段设置为 NULL。

set_cancel 回调

```
void (SQL_CALLBACK *set_cancel)
(
    void *arg_handle,
    void *cancel_handle
);
```

外部函数可以获得 IN 或 INOUT 参数的值，并且可以设置 OUT 参数以及存储函数的 RETURNS 结果。但有一种情况下，获取的参数值可能不再有效，或者值的设置不再必要。这种情况发生在取消一个正在执行的 SQL 语句的时候。这可能是因为应用程序突然与数据库服务器断开连接。要处理这种情况，可以在一个名为 extfn_cancel 的库中定义一个特殊的入口点。定义此函数后，每当取消一个正在运行的 SQL 语句，服务器就会调用该函数。

extfn_cancel 函数通过一个可以以任何合适方式使用的句柄来调用。句柄的典型用途是间接设置一个标志来指示正在调用的 SQL 语句已取消。

传递的句柄值可以由外部库中的函数使用 set_cancel 回调函数来设置。这可由以下代码段来说明。

```
extern "C" __declspec( dllexport )
void extfn_cancel( void *cancel_handle )
{
    *(short *)cancel_handle = 1;
}

extern "C" __declspec( dllexport )
void mystring( an_extfn_api *api, void *arg_handle )
{
.
.
.
    short  canceled = 0;

    extapi->set_cancel( arg_handle, &canceled );
.
.
.
    if( canceled )
```

请注意，设置静态全局 "canceled" 变量是不恰当的，因为当所有连接的所有 SQL 语句都被取消时（通常不会这样），它会被错误的解释。这正是提供 set_cancel 回调函数的原因。在调用 set_cancel 之前务必要初始化 "canceled" 变量。

在外部函数的关键控制点检查 "canceled" 变量的设置是非常重要的。关键控制点包括通过调用外部库来调用 API 函数（如 get_value 和 set_value）的之前和之后。如果设置了变量（调用 extfn_cancel 的结果），外部函数可以采取适当的终止动作。基于前面示例的代码段如下：

```
if( canceled )
{
```

```
        free( string );
        return;
    }
```

注意

任何给定参数的 get_piece 函数只能紧接着相同参数的 get_value 函数被调用。

在 OUT 参数上调用 get_value 将返回设置为参数数据类型的 an_extfn_value 结构的类型字段，以及设置为 NULL 的 an_extfn_value 结构的数据字段。

SQL Anywhere 安装目录 *SDK\Include* 文件夹中的头文件 *extfnapi.h* 包含一些附加说明。

下表说明在 an_extfn_api 中定义的函数返回 false 所基于的条件：

函数	在以下条件为真时返回 0；否则返回 1
get_value()	● arg_num 无效；例如，arg_num 大于外部函数中参数的数目。
get_piece()	● arg_num 无效；例如，arg_num 与之前 get_value 调用所用的参数数目不相符。 ● 偏移超出用于 arg_num 参数值的总长度。 ● 在调用 get_value 之前调用它。
set_value()	● arg_num 无效；例如，arg_num 大于外部函数中参数的数目。 ● 参数 arg_num 只用于输入。 ● 提供的值的类型与参数 arg_num 的类型不匹配。

处理数据类型

数据类型

以下 SQL 数据类型可传递给外部库：

SQL 数据类型	sqldef.h	C 类型
CHAR	DT_FIXCHAR	具有指定长度的字符数据
VARCHAR	DT_VARCHAR	具有指定长度的字符数据
LONG VARCHAR、TEXT	DT_LONGVARCHAR	具有指定长度的字符数据
UNIQUEIDENTIFIERSTR	DT_FIXCHAR	具有指定长度的字符数据
XML	DT_LONGVARCHAR	具有指定长度的字符数据
NCHAR	DT_NFIXCHAR	UTF-8 字符数据，具有指定长度
NVARCHAR	DT_NVARCHAR	UTF-8 字符数据，具有指定长度
LONG NVARCHAR、NTEXT	DT_LONGNVARCHAR	UTF-8 字符数据，具有指定长度
UNIQUEIDENTIFIER	DT_BINARY	二进制数据，长度为 16 字节
BINARY	DT_BINARY	具有指定长度的二进制数据
VARBINARY	DT_BINARY	具有指定长度的二进制数据
LONG BINARY	DT_LONGBINARY	具有指定长度的二进制数据
TINYINT	DT_TINYINT	1 字节整数
[UNSIGNED] SMALLINT	DT_SMALLINT、DT_UNSMALLINT	[无符号的] 2 字节整数
[UNSIGNED] INT	DT_INT、DT_UNSINT	[无符号的] 4 字节整数

SQL 数据类型	sqldef.h	C 类型
[UNSIGNED] BIGINT	DT_BIGINT、DT_UNSBIGINT	[无符号的] 8 字节整数
REAL、FLOAT(1-24)	DT_FLOAT	单精度浮点数
DOUBLE、FLOAT(25-53)	DT_DOUBLE	双精度浮点数

不能使用任何日期或时间数据类型，也不能使用 DECIMAL 或 NUMERIC 数据类型（包括货币类型）。

要为 INOUT 或 OUT 参数提供值，请使用 set_value API 函数。要读取 IN 和 INOUT 参数，请使用 get_value API 函数。

确定参数的数据类型

调用 get_value 之后，an_extfn_value 结构的类型字段可用于获取参数的数据类型信息。以下示例代码段显示了如何标识参数的类型。

```
an_extfn_value      arg;
a_sql_data_type     data_type;

extapi->get_value( arg_handle, 1, &arg );
data_type = arg.type & DT_TYPES;
switch( data_type )
{
case DT_FIXCHAR:
case DT_VARCHAR:
case DT_LONGVARCHAR:
    break;
default:
    return;
}
```

有关数据类型的详细信息，请参见“使用主机变量”一节第 524 页。

UTF-8 类型

NCHAR、NVARCHAR、LONG NVARCHAR 和 NTEXT 等 UTF-8 数据类型都作为 UTF-8 编码字符串传递。Windows MultiByteToWideChar 等函数可用来将 UTF-8 字符串转换为宽字符 (Unicode) 字符串。

传递 NULL

对于所有参数，您都可以将 NULL 作为有效值传递。外部库中的函数可以提供 NULL 作为任何数据类型的返回值。

返回值

要设置外部函数中的返回值，请以值为 0 的 arg_num 参数调用 set_value 函数。如果不使用设置为 0 的 arg_num 调用 set_value，则该函数的结果为 NULL。

为调用存储函数设置返回值的数据类型同样非常重要。以下代码段显示了如何设置返回数据类型。


```
an_extfn_value      retval;

retval.type = DT_LONGVARCHAR;
retval.data = result;
retval.piece_len = retval.len.total_len = (a_sql_uint32) strlen( result );
extapi->set_value( arg_handle, 0, &retval, 0 );
```

卸载外部库

系统过程 dbo.sa_external_library_unload 可用于在外部库不使用时将其卸载。该过程使用一个可选 long varchar 参数。该参数用于指定要卸载的库的名称。如果未指定任何参数，则将卸载所有未使用的外部库。

以下是一个卸载外部函数库的示例。

```
call sa_external_library_unload('library.dll')
```

在开发一组外部函数时，该函数非常有用，因为您不必关闭数据库服务器来安装更新版本的库。

SQL Anywhere 外部环境支持

目录

外部环境概述

SQL Anywhere 支持六种外部运行时环境。它们包括以 C/C++ 编写的嵌入式 SQL 和 ODBC 应用程序，以及以 Java、Perl、PHP 或 C# 和 Visual Basic 等基于 Microsoft .NET Framework 公共语言运行库 (CLR) 的语言编写的应用程序。

SQL Anywhere 支持调用以 C 或 C++ 语言编写的编译后本地函数已有一段时间了。但是，如果这些过程由数据库服务器运行，动态链接库或共享对象始终由数据库服务器装载，本地函数也始终由数据库服务器调用。这样做的风险是，如果本地函数导致故障，则会使数据库服务器崩溃。所以，在数据库服务器的外部（即在外部环境中）运行已编译的本地函数，可以消除服务器的这些风险。

以下是对 SQL Anywhere 在外部环境支持方面的概述。

目录表

一个系统目录表用来存储标识和启动每个外部环境所需的信息。此表的定义如下：

```
SYS.SYSEXTERNENV (
  object_id            unsigned bigint not null,
  name                 varchar(128)    not null,
  scope                char(1)         not null,
  supports_result_sets char(1)         not null,
  location             long varchar    not null,
  options              long varchar    not null,
  user_id              unsigned int
)
```

- **object_id** 由数据库服务器生成的唯一标识符。
- **name** name 列标识外部环境或语言的名称。即 **java**、**perl**、**php**、**clr**、**c_esql32**、**c_esql64**、**c_odbc32** 或 **c_odbc64**。
- **scope** scope 列为 **C** 或 **D**，分别代表 CONNECTION（连接）或 DATABASE（数据库）。scope 列用来标识外部环境是与连接一一对应还是与数据库一一对应。

  对于与连接一一对应的外部环境（如 PERL、PHP、C_ESQL32、C_ESQL64、C_ODBC32 和 C_ODBC64），针对使用外部环境的每个连接会有一个外部环境实例。如果是与连接一一对应，则连接终止时外部环境即终止。

  对于与数据库一一对应的外部环境（如 JAVA 和 CLR），针对使用外部环境的每个数据库会有一个外部环境实例。如果是与数据库一一对应，则数据库停止时外部环境即终止。
- **supports_result_sets** supports_result_sets 列标识那些可以返回结果集的外部环境。除了 PERL 和 PHP，所有外部环境都可以返回结果集。
- **location** location 列标识数据库服务器计算机上可以找到外部环境可执行文件/二进制文件的位置。它包括可执行文件名/二进制文件名。此路径可以是完全限定路径，也可以是相对路径。如果是相对路径，则可执行文件/二进制文件必须位于数据库服务器可以找到的位置。
- **options** options 列标识命令行上启动与外部环境相关联的可执行文件所需的任何选项。您不能修改此列。
- **user_id** user_id 列标识数据库中具有 DBA 权限的用户 ID。最初启动外部环境时，必须建立返回到数据库的连接，以为外部环境的使用做好准备。缺省情况下，此连接会使用该 DBA 用户 ID 建立，但如果数据库管理员倾向于让外部环境使用另一个具有 DBA 权限的用户 ID，就会

转而由 SYS.SYSEXTERNENV 表中的 user_id 列指示该不同的用户 ID。然而大多数情况下，SYS.SYSEXTERNENV 中的此列为 NULL，数据库服务器会缺省使用这里的 DBA 用户 ID。

另一个系统目录表用来存储非 Java 外部对象。此表的表定义如下：

```
SYS.SYSEXTERNENVOBJECT (
  object_id    unsigned bigint not null,
  extenv_id    unsigned bigint not null,
  owner        unsigned int    not null,
  name         long varchar    not null,
  contents     long binary     not null,
  update_time  timestamp       not null
)
```

- **object_id** 由数据库服务器生成的唯一标识符。
- **extenv_id** extenv_id 标识外部环境类型（如 SYS.SYSEXTERNENV 中所存储的）。
- **owner** owner 列标识外部对象的创建者/所有者。
- **name** name 列是 INSTALL EXTERNAL OBJECT 语句中所指定的外部对象的名称。
- **contents** contents 列包含外部对象的内容。
- **update_time** update_time 列代表上次修改（或安装）对象的时间。

不再支持的选项

随着 SYS.SYSEXTERNENV 表的引入，现在已不再支持某些特定于 Java 的选项。不再支持的选项有：

```
java_location
java_main_userid
```

对于之前一直使用这些选项来标识使用哪个具体 Java VM 或使用哪个用户 ID 安装类及执行其它 Java 相关管理任务的应用程序，应转而使用 ALTER EXTERNAL ENVIRONMENT 语句，来在 SYS.SYSEXTERNENV 表中针对 Java 设置位置和 user_id 值。

SQL 语句

以下 SQL 语法用于设置或修改 SYS.SYSEXTERNENV 表中的值。

```
ALTER EXTERNAL ENVIRONMENT environment-name
    [ USER user-name ]
    [ LOCATION location-string ]
```

- **environment-name** 环境名是 SYS.SYSEXTERNENV 中代表环境名称的标识符，可以是 PERL、PHP、JAVA、CLR、C_ESQL32、C_ESQL64、C_ODBC32 或 C_ODBC64。
- **user-name** 用户名字符串标识数据库中具有 DBA 权限的用户。最初启动外部环境时，必须建立返回到数据库的连接，以为外部环境的使用做好准备。缺省情况下，此连接会使用该 DBA 用户 ID 建立，但如果数据库管理员倾向于让外部环境使用另一个具有 DBA 权限的用户 ID，则 user-name 会指示将使用的不同用户 ID。大多数情况下，不需要指定此选项。

- **location-string** 位置字符串标识数据库服务器计算机上可以找到外部环境可执行文件/二进制文件的位置。它包括可执行文件名/二进制文件名。此路径可以是完全限定路径，也可以是相对路径。如果是相对路径，则可执行文件/二进制文件必须位于数据库服务器可以找到的位置。

某个外部环境被设置为在数据库服务器上使用后，即可以在数据库中安装对象并在外部环境内创建使用这些对象的存储过程和函数。这些对象、存储过程和存储函数的安装、创建与使用与当前安装 Java 类和创建及使用 Java 存储过程和函数的方法非常类似。

要添加针对外部环境的注释，可以执行以下语句：

```
COMMENT ON EXTERNAL ENVIRONMENT environment-name
  IS comment-string
```

要从文件或表达式将外部对象（例如 Perl 脚本）安装到数据库中，需要执行如下的 INSTALL EXTERNAL OBJECT 语句：

```
INSTALL EXTERNAL OBJECT object-name-string
  [ update-mode ]
  FROM { FILE file-path | VALUE expression }
  ENVIRONMENT environment-name
```

- **object-name-string** 对象名字符串是数据库中用来标识安装的对象的名称。
- **update-mode** 更新模式为 NEW 或 UPDATE。如果忽略更新模式，则使用 NEW。
- **file-path** 文件路径是数据库服务器计算机上从中安装对象的位置。
- **environment-name** 环境名为 JAVA、PERL、PHP、CLR、C_ESQL32、C_ESQL64、C_ODBC32 或 C_ODBC64。

要添加针对所安装外部对象的注释，可以执行以下语句：

```
COMMENT ON EXTERNAL ENVIRONMENT OBJECT object-name-string
  IS comment-string
```

要从数据库中删除所安装的外部对象，需要使用 REMOVE EXTERNAL OBJECT 语句：

```
REMOVE EXTERNAL OBJECT object-name-string
```

- **object-name-string** 对象名字符串与在相应 INSTALL EXTERNAL OBJECT 语句中指定的字符串为同一字符串。

外部对象安装在数据库中后，即可在外部存储过程和函数定义中使用（类似于当前创建 Java 存储过程和函数的机制）。

```
CREATE PROCEDURE procedure-name(...)
  EXTERNAL NAME '...'
  LANGUAGE environment-name

CREATE FUNCTION function-name(...)
  RETURNS ...
  EXTERNAL NAME '...'
  LANGUAGE environment-name
```


- **environment-name** 环境名为 JAVA、PERL、PHP、CLR、C_ESQL32、C_ESQL64、C_ODBC32 或 C_ODBC64。

这些存储过程和函数创建后，即可像数据库中任何其它存储过程或函数一样使用。遇到外部环境存储过程或函数时，数据库服务器会自动启动外部环境（如果尚未启动），并发送获取外部环境所需的一切信息，以便从数据库读取外部对象并执行。根据需要，会返回执行后产生的任何结果集或返回值。

如果要根据要求启动或停止外部环境，可以使用 START EXTERNAL ENVIRONMENT 和 STOP EXTERNAL ENVIRONMENT 语句（类似于当前的 START JAVA 和 STOP JAVA 语句）：

```
START EXTERNAL ENVIRONMENT environment-name
STOP EXTERNAL ENVIRONMENT environment-name
```

- **environment-name** 环境名为 JAVA、PERL、PHP、CLR、C_ESQL32、C_ESQL64、C_ODBC32 或 C_ODBC64。

有关详细信息，请参见：

- “ALTER EXTERNAL ENVIRONMENT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “INSTALL EXTERNAL OBJECT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “REMOVE EXTERNAL OBJECT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “COMMENT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “CREATE FUNCTION 语句（Web 服务）”一节 《SQL Anywhere 服务器 - SQL 参考》
- “CREATE PROCEDURE 语句（Web 服务）”一节 《SQL Anywhere 服务器 - SQL 参考》
- “START EXTERNAL ENVIRONMENT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “STOP EXTERNAL ENVIRONMENT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

CLR 外部环境

SQL Anywhere 支持 CLR 存储过程和函数。CLR 存储过程或函数的行为与 SQL 存储过程或函数的基本相同，只是过程或函数的代码以 C# 或 Visual Basic 等 .NET 语言编写并在数据库服务器外（即在单独的 .NET 可执行文件内）执行。每个数据库只有一个此 .NET 可执行文件的实例。执行 CLR 函数和存储过程的所有连接都使用同一个 .NET 可执行实例，但每个连接的命名空间是独立的。静态在连接期间内会一直保持，但在连接之间不共享。仅支持 .NET 版本 2.0。

要调用外部 CLR 函数或过程，需要在定义相应存储过程或函数时指定 EXTERNAL NAME 字符串，确定装载哪个 DLL 以及调用程序集中的哪个函数。此外，在定义存储过程或函数时，还必须指定 LANGUAGE CLR。以下是一个声明示例：

```
CREATE PROCEDURE clr_stored_proc(
    IN p1 INT,
    IN p2 UNSIGNED SMALLINT,
    OUT p3 LONG VARCHAR)
EXTERNAL NAME 'MyCLRTest.dll::MyCLRTest.Run( int, ushort, out string )'
LANGUAGE CLR;
```

在此示例中，名为 `clr_stored_proc` 的存储过程在执行时会装载 DLL *MyCLRTest.dll* 并调用函数 MyCLRTest.Run。clr_stored_proc 过程使用三个 SQL 参数，即两个分别为 INT 类型和 UNSIGNED SMALLINT 类型的 IN 参数和一个 LONG VARCHAR 类型的 OUT 参数。在 .NET 端，这三个参数转换为 int 类型和 ushort 类型的输入参数和 string 类型的输出参数。除了 out 参数，CLR 函数还可以具有 ref 参数。如果相应的存储过程有一个 INOUT 参数，则用户必须声明一个 ref CLR 参数。

下表列出了各种 CLR 参数类型以及对应的建议 SQL 数据类型：

CLR 类型	建议的 SQL 数据类型
bool	bit
byte	tinyint
short	smallint
ushort	unsigned smallint
int	int
uint	unsigned int
long	bigint
ulong	unsigned bigint
decimal	numeric
float	real

CLR 类型	建议的 SQL 数据类型
double	double
DateTime	timestamp
string	long varchar
byte[ ]	long binary

DLL 的声明可以使用相对路径也可以使用绝对路径。如果指定路径为相对路径，则外部 .NET 可执行文件会在该路径及其它位置搜索 DLL。可执行文件不会在全局程序集高速缓存 (GAC) 中搜索 DLL。

与现有 Java 存储过程和函数一样，CLR 存储过程和函数可以发出返回到数据库的服务器端请求，还可以返回结果集。而且，像 Java 一样，任何输出到 Console.Out 和 Console.Error 的信息都会自动重定向到数据库服务器消息窗口。

关于如何发出服务器端请求以及如何从 CLR 函数或存储过程返回结果集的详细信息，请参见位于 *samples-dir\SQLAnywhere\ExternalEnvironments\CLR* 目录中的示例。

在数据库中使用 CLR，需确保数据库服务器能够找到并启动 CLR 可执行文件。通过执行以下语句可以验证数据库服务器是否能够找到并启动 CLR 可执行文件：

```
START EXTERNAL ENVIRONMENT CLR;
```

如果数据库服务器未能启动 CLR，则可能是数据库服务器无法找到 CLR 可执行文件。CLR 可执行文件是 *dbextclr11.exe*。应确保此文件位于 *install-dir\Bin32* 或 *install-dir\Bin64* 文件夹中，具体哪个文件夹取决于所用数据库服务器的版本。

请注意，除了验证数据库服务器可以启动 CLR 可执行文件外，START EXTERNAL ENVIRONMENT CLR 语句并不是必需的。通常，进行 CLR 存储过程或函数的调用会自动启动 CLR。

类似地，停止 CLR 的实例时 STOP EXTERNAL ENVIRONMENT CLR 语句也不是必需的，因为连接终止时实例会自动消失。然而，如果要彻底离开 CLR 并且想要释放一些资源，STOP EXTERNAL ENVIRONMENT CLR 语句则可以为您的连接释放 CLR 实例。

与 Perl、PHP 和 Java 外部环境不同，CLR 环境不需要在数据库中安装任何内容。因此，使用 CLR 外部环境前，无需执行任何 INSTALL 语句。

下面是一个可以在外部环境中运行、以 C# 编写的函数的示例。

```
public class StaticTest
{
    private static int val = 0;

    public static int GetValue() {
        val += 1;
        return val;
    }
}
```

此函数编译到动态链接库后，即可从外部环境调用。数据库服务器会启动名为 *dbextclr11.exe* 的可执行映像，用以为您装载该动态链接库。SQL Anywhere 附带了多种版本的该可执行文件。例如，

在 Windows 中，可以同时拥有 32 位和 64 位的可执行文件。一个用于 32 位版本的数据库服务器，另一个用于 64 位版本的数据库服务器。

要使用 Microsoft C# 编译器将此应用程序构建到动态链接库中，请使用如下的命令。假定上面示例的源代码位于名为 *StaticTest.cs* 的文件中。

```
csc /target:library /out:clrtest.dll StaticTest.cs
```

此命令将编译后的代码放在名为 *clrtest.dll* 的 DLL 中。要调用编译后的 C# 函数 GetValue，应使用 Interactive SQL 按如下方式定义包装：

```
CREATE FUNCTION stc_get_value()
RETURNS INT
EXTERNAL NAME 'clrtest.dll::StaticTest.GetValue() int'
LANGUAGE CLR;
```

对于 CLR，EXTERNAL NAME 字符串以 SQL 的单独一行指定。为了能够找到 DLL，您可能需要在 EXTERNAL NAME 字符串中包含该 DLL 的路径。对于相关程序集（例如，如果 *myLib.dll* 的代码调用的函数在 *myOtherLib.dll* 中或以某种方式与其相关），则由 .NET Framework 决定装载依赖性。CLR 外部环境将对指定程序集的装载进行处理，但可能需要执行额外的步骤才能确保装载相关程序集。一种解决方案是，通过使用与 .NET Framework 一起安装的 Microsoft **gacutil** 实用程序，在全局程序集高速缓存 (**GAC**) 中注册所有依赖性。对于自定义开发库，gacutil 要求先使用强命名密钥对库进行签名，然后才能在 **GAC** 中注册这些库。

要执行该编译后 C# 函数示例，请执行以下语句。

```
SELECT stc_get_value();
```

每次调用 C# 函数都会生成一个新的整数结果。返回值的顺序是 1、2、3 以此类推。

有关在数据库支持中使用 CLR 的更详细信息和示例，请参见位于 *samples-dir\SQLAnywhere \ExternalEnvironments\CLR* 目录中的示例。

ESQL 和 ODBC 外部环境

SQL Anywhere 支持调用以 C 或 C++ 语言编写的编译后本地函数已有一段时间了。但是，如果这些过程由数据库服务器运行，动态链接库或共享对象始终由数据库服务器装载，本地函数也始终由数据库服务器调用。尽管让数据库服务器进行这些本地调用是最为有效的，但如果本地函数存在问题，就可能会导致严重的后果。具体来说，如果本地函数进入一个无限循环，数据库服务器就可能会挂起，而且如果本地函数引发故障，就会使数据库服务器崩溃。因此，现在您可以选择在数据库服务器的外部（即在外部环境中）运行编译后的本地函数。在外部环境中运行编译后的本地函数有几大优点：

1. 即使编译后的本地函数存在问题，数据库服务器也不会挂起或崩溃。
2. 本地函数可以使用 ODBC、嵌入式 SQL (ESQL) 或 SQL Anywhere C API，而且无需建立连接即可进行返回到数据库服务器的服务器端调用。
3. 本地函数可以将结果集返回给数据库服务器。
4. 在外部环境中，32 位数据库服务器可以与 64 位编译后本地函数进行通信，反之亦然。请注意，如果编译后的本地函数直接装载到数据库服务器的地址空间中，则无法实现上述的通信。32 位的库只能由 32 位服务器装载，同样 64 位的库也只能由 64 位服务器装载。

在外部环境（而非数据库服务器）中运行编译后的本地函数会略微影响性能。

此外，编译后的本地函数必须使用本地函数调用 API 将信息传递到本地函数，并从本地函数返回信息。此 API 将在“SQL Anywhere 外部函数 API”第 641 页中介绍。

要在外部环境而非数据库服务器中运行编译后的本地 C 函数，存储过程或函数应使用其后的 LANGUAGE 属性的 EXTERNAL NAME 子句来定义，其中，LANGUAGE 属性指定 C_ESQL32、C_ESQL64、C_ODBC32 或 C_ODBC64。

与 Perl、PHP 和 Java 外部环境不同的是，不需要在数据库中安装任何源代码或编译后的对象。因此，使用 ESQL 和 ODBC 外部环境前，无需执行任何 INSTALL 语句。

下面是一个可以在数据库服务器或外部环境中运行、以 C++ 编写的函数的示例。

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}

// Note: extfn_use_new_api used only for
// execution in the database server

extern "C" __declspec( dllexport )
a_sql_uint32 extfn_use_new_api( void )
{
    return( EXTFN_API_VERSION );
```

```
}

extern "C" __declspec( dllexport )
void SimpleCFunction(
    an_extfn_api *api,
    void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;
    int *               intp;
    int                 i, j, k;

    j = 1000;
    k = 0;
    for( i = 1; i <= 4; i++ )
    {
        result = api->get_value( arg_handle, i, &arg );
        if( result == 0 || arg.data == NULL ) break;
        if( arg.type & DT_TYPES != DT_INT ) break;
        intp = (int *) arg.data;
        k += *intp * j;
        j = j / 10;
    }
    retval.type = DT_INT;
    retval.data = (void*)&k;
    retval.piece_len = retval.len.total_len =
       (a_sql_uint32) sizeof( int );
    api->set_value( arg_handle, 0, &retval, 0 );
    return;
}
```

此函数在编译到动态链接库或共享对象中后，即可从外部环境调用。数据库服务器会启动名为 *dbexternc11* 的可执行映像，此可执行映像会为您装载该动态链接库或共享对象。SQL Anywhere 附带了多种版本的该可执行文件。例如，在 Windows 中，可以同时拥有 32 位和 64 位的可执行文件。

请注意，32 位或 64 位版本的数据库服务器都可以使用，任何一个版本都可以启动 32 位或 64 位版本的 *dbexternc11*。这是使用外部环境的优点之一。请注意，*dbexternc11* 一旦由数据库服务器启动就会始终运行，直到连接终止或执行了 STOP EXTERNAL ENVIRONMENT 语句（包含正确的环境名）。每个执行外部环境调用的连接都将获得其自身的 *dbexternc11* 副本。

要调用编译后的本地函数 SimpleCFunction，包装应如下定义：

```
CREATE FUNCTION SimpleCDemo(
  IN arg1 INT,
  IN arg2 INT,
  IN arg3 INT,
  IN arg4 INT )
RETURNS INT
EXTERNAL NAME 'SimpleCFunction@c:\\c\\extdemo.dll'
LANGUAGE C_ODBC32;
```

这与在将编译后本地函数加载到数据库服务器地址空间时该函数的描述方式几乎完全相同。唯一的区别是使用了 LANGUAGE C_ODBC32 子句。该子句说明 SimpleCDemo 是一个在外部环境中运行的函数，它使用 32 位 ODBC 调用。C_ESQL32、C_ESQL64、C_ODBC32 或 C_ODBC64 的语言规范告诉数据库服务器：在发出服务器端请求时，外部 C 函数是发出 32 位还是 64 位调用，以及这些调用是 ODBC、ESQL 调用还是 SQL Anywhere C API 调用。

如果本地函数不使用 ODBC、ESQL 或 SQL Anywhere C API 调用中的任何一个执行服务器端请求，则 C_ODBC32 或 C_ESQL32 可用于 32 位应用程序，C_ODBC64 或 C_ESQL64 可用于 64 位应用程序。这是以上所示的外部 C 函数中的情况。它未使用这些 API 中的任何一个。

要执行该编译后本地函数示例，请执行以下语句。

```
SELECT SimpleCDemo(1,2,3,4);
```

要使用服务器端 ODBC，C/C++ 代码必须使用缺省数据库连接。要获取数据库连接的句柄，请以 EXTFN_CONNECTION_HANDLE_ARG_NUM 参数调用 get_value。该参数会指示数据库服务器返回当前外部环境连接，而不是打开一个新连接。

```
#include <windows.h>
#include <stdio.h>
#include "odbc.h"
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}

extern "C" __declspec( dllexport )
void ServerSideFunction( an_extfn_api *api, void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;
    SQLRETURN           ret;

    ret = -1;
    // set up the return value struct
    retval.type = DT_INT;
    retval.data = (void*) &ret;
    retval.piece_len = retval.len.total_len =
        (a_sql_uint32) sizeof( int );

    result = api->get_value( arg_handle,
                    EXTFN_CONNECTION_HANDLE_ARG_NUM,
                    &arg );
    if( result == 0 || arg.data == NULL )
    {
        api->set_value( arg_handle, 0, &retval, 0 );
        return;
    }

    HDBC dbc = (HDBC)arg.data;
    HSTMT stmt = SQL_NULL_HSTMT;
    ret = SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
    if( ret != SQL_SUCCESS ) return;
    ret = SQLExecDirect( stmt,
            (SQLCHAR *) "INSERT INTO odbcTab "
                "SELECT table_id, table_name "
                "FROM SYS.SYSTAB", SQL_NTS );
    if( ret == SQL_SUCCESS )
    {
        SQLExecDirect( stmt,
            (SQLCHAR *) "COMMIT", SQL_NTS );
    }
```

```
    SQLFreeHandle( SQL_HANDLE_STMT, stmt );

    api->set_value( arg_handle, 0, &retval, 0 );
    return;
}
```

如果以上 ODBC 代码存储在文件 *extodbc.cpp* 中，则可以使用以下命令为 Windows 构建该文件（假设 SQL Anywhere 软件安装到文件夹 *c:\sa11* 中并且已安装 Microsoft Visual C++）。

```
cl extodbc.cpp /LD /Ic:\sa11\sdk\include odbc32.lib
```

下面的示例将创建一个表，定义用来调用编译后本地函数的存储过程包装，然后调用该本地函数来填充该表。

```
CREATE TABLE odbcTab(c1 int, c2 char(128));

CREATE FUNCTION ServerSideODBC( )
RETURNS INT
EXTERNAL NAME 'ServerSideFunction@extodbc.dll'
LANGUAGE C_ODBC32;

SELECT ServerSideODBC();

// The following statement should return two identical rows
SELECT COUNT(*) FROM odbcTab
UNION ALL
SELECT COUNT(*) FROM SYS.SYSTAB;
```

同样，要使用服务器端 ESQL，C/C++ 代码也必须使用缺省数据库连接。要获取数据库连接的句柄，请以 EXTFN_CONNECTION_HANDLE_ARG_NUM 参数调用 get_value。该参数会指示数据库服务器返回当前外部环境连接，而不是打开一个新连接。

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

#include "sqlca.h"
#include "sqlda.h"
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}

EXEC SQL INCLUDE SQLCA;
static SQLCA *_sqlc;
EXEC SQL SET SQLCA "_sqlc";
EXEC SQL WHENEVER SQLERROR { ret = _sqlc->sqlcode; };

extern "C" __declspec( dllexport )
void ServerSideFunction( an_extfn_api *api, void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;

    EXEC SQL BEGIN DECLARE SECTION;
    char *stmt_text =
        "INSERT INTO esqlTab "
```



```
            "SELECT table_id, table_name "
            "FROM SYS.SYSTAB";
    char *stmt_commit =
        "COMMIT";
    EXEC SQL END DECLARE SECTION;

    int ret = -1;

    // set up the return value struct
    retval.type = DT_INT;
    retval.data = (void*) &ret;
    retval.piece_len = retval.len.total_len =
        (a_sql_uint32) sizeof( int );

    result = api->get_value( arg_handle,
                    EXTFN_CONNECTION_HANDLE_ARG_NUM,
                    &arg );
    if( result == 0 || arg.data == NULL )
    {
        api->set_value( arg_handle, 0, &retval, 0 );
        return;
    }
    ret = 0;
    _sqlc = (SQLCA *)arg.data;

    EXEC SQL EXECUTE IMMEDIATE :stmt_text;
    EXEC SQL EXECUTE IMMEDIATE :stmt_commit;

    api->set_value( arg_handle, 0, &retval, 0 );
}
```

如果以上嵌入式 SQL 代码存储在文件 *extesql.sqc* 中，则可以使用以下命令为 Windows 构建该文件（假设 SQL Anywhere 软件安装到文件夹 *c:\sa11* 中并且已安装 Microsoft Visual C++）。

```
sqlpp extesql.sqc extesql.cpp
cl extesql.cpp /LD /Ic:\sa11\sdk\include c:\sa11\sdk\lib\x86\dblibtm.lib
```

下面的示例将创建一个表，定义用来调用编译后本地函数的存储过程包装，然后调用该本地函数来填充该表。

```
CREATE TABLE esqlTab(c1 int, c2 char(128));

CREATE FUNCTION ServerSideESQL( )
RETURNS INT
EXTERNAL NAME 'ServerSideFunction@extesql.dll'
LANGUAGE C_ESQL32;

SELECT ServerSideESQL();

// The following statement should return two identical rows
SELECT COUNT(*) FROM esqlTab
UNION ALL
SELECT COUNT(*) FROM SYS.SYSTAB;
```

与在前面的示例中一样，要使用服务器端 SQL Anywhere C API 调用，C/C++ 代码必须使用缺省数据库连接。要获取数据库连接的句柄，请以 EXTFN_CONNECTION_HANDLE_ARG_NUM 参数调用 get_value。该参数会指示数据库服务器返回当前外部环境连接，而不是打开一个新连接。以下示例显示了这样一个框架：获得连接句柄、初始化 C API 环境并将连接句柄转换成可与 SQL Anywhere C API 一起使用的连接对象 (**a_sqlany_connection**)。

```
#include <windows.h>
#include "sacapidll.h"
```

```
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}

extern "C" __declspec( dllexport )
void ServerSideFunction( an_extfn_api *extapi, void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;
    unsigned            offset;
    char                *cmd;

    SQLAnywhereInterface  capi;
    a_sqlany_connection * sqlany_conn;
    unsigned int          max_api_ver;

    result = extapi->get_value( arg_handle,
                    EXTFN_CONNECTION_HANDLE_ARG_NUM,
                    &arg );

    if( result == 0 || arg.data == NULL )
    {
        return;
    }
    if( !sqlany_initialize_interface( &capi, NULL ) )
    {
        return;
    }
    if( !capi.sqlany_init( "MyAPP",
            SQLANY_CURRENT_API_VERSION,
            &max_api_ver ) )
    {
        sqlany_finalize_interface( &capi );
        return;
    }
    sqlany_conn = sqlany_make_connection( arg.data );

    // processing code goes here

    capi.sqlany_fini();

    sqlany_finalize_interface( &capi );
    return;
}
```

如果以上 C 代码存储在文件 *extcapi.c* 中，则可以使用以下命令为 Windows 构建该文件（假设 SQL Anywhere 软件安装到文件夹 *c:\sa11* 中并且已安装 Microsoft Visual C++）。

```
cl /LD /Tp extcapi.c /Tp c:\sa11\SDK\C\sacapidll.c /Ic:\sa11\SDK\Include c:
\sa11\SDK\Lib\X86\dbcapi.lib
```

下面的示例将定义用来调用编译后的本地函数的存储过程包装，然后调用该本地函数。

```
CREATE FUNCTION ServerSideC()
RETURNS INT
EXTERNAL NAME 'ServerSideFunction@extcapi.dll'
```



```
LANGUAGE C_ESQL32;

SELECT ServerSideC();
```

上例中的 LANGUAGE 属性指定 C_ESQL32。将为 64 位应用程序使用 C_ESQL64。必须使用嵌入式 SQL 语言属性，因为 SQL Anywhere C API 构建在与 ESQL 相同的层（库）上。

如前所述，每个执行外部环境调用的连接都将启动其自身的 *dbexternc11* 副本。首次执行外部环境调用时，此可执行应用程序由服务器自动装载。但是，可以使用 START EXTERNAL ENVIRONMENT 语句预装载 *dbexternc11*。这在想要避免在首次执行外部环境调用时出现的略微延迟的情况下很有用。以下是该语句的示例。

```
START EXTERNAL ENVIRONMENT C_ESQL32
```

预装载 *dbexternc11* 在另一种情况也很有用，即想要调试外部函数时。可以使用调试程序连接到正在运行的 *dbexternc11* 过程，并在外部函数中设置断点。

更新动态链接库或共享对象时，STOP EXTERNAL ENVIRONMENT 语句很有用。它将为当前连接终止本地库装载程序 *dbexternc11*，从而释放对动态链接库或共享对象的访问。如果多个连接在使用同一个动态链接库或共享对象，则它们的每一个 *dbexternc11* 副本都必须终止。必须在 STOP EXTERNAL ENVIRONMENT 语句中指定相应的外部环境名称。以下是该语句的示例。

```
STOP EXTERNAL ENVIRONMENT C_ESQL32
```

要从外部函数返回结果集，编译后的本地函数必须使用本地函数调用 API。此 API 将在“SQL Anywhere 外部函数 API”第 641 页中进行详细介绍。以下是一些用于返回结果集的重点代码段。

以下代码段显示了如何设置结果集信息结构。它包含列计数、指向列信息结构数组的指针，以及指向列数据值结构数组的指针。本示例也使用 SQL Anywhere C API。

```
an_extfn_result_set_info    rs_info;

int columns = capi.sqlany_num_cols( sqlany_stmt );

an_extfn_result_set_column_info *col_info =
    (an_extfn_result_set_column_info *)
    malloc( columns * sizeof(an_extfn_result_set_column_info) );

an_extfn_result_set_column_data *col_data =
    (an_extfn_result_set_column_data *)
    malloc( columns * sizeof(an_extfn_result_set_column_data) );

rs_info.number_of_columns   = columns;
rs_info.column_infos        = col_info;
rs_info.column_data_values  = col_data;
```

以下代码段显示了如何描述结果集。它使用 SQL Anywhere C API 为先前由 C API 执行的 SQL 查询获取列信息。从 SQL Anywhere C API 获得的各列信息将被转换为用于描述结果集的列名称、类型、宽度、索引以及空值指示符。

```
a_sqlany_column_info        info;
for( int i = 0; i < columns; i++ )
{
    if( sqlany_get_column_info( sqlany_stmt, i, &info ) )
    {
        // set up a column  description
        col_info[i].column_name  = info.name;
        col_info[i].column_type  = info.native_type;
```

```
            switch( info.native_type )
            {
                case DT_DATE:       // DATE is converted to string by C API
                case DT_TIME:       // TIME is converted to string by C API
                case DT_TIMESTAMP:  // TIMESTAMP is converted to string by C API
                case DT_DECIMAL:    // DECIMAL is converted to string by C API
                    col_info[i].column_type  = DT_FIXCHAR;
                    break;
                case DT_FLOAT:      // FLOAT is converted to double by C API
                    col_info[i].column_type  = DT_DOUBLE;
                    break;
                case DT_BIT:        // BIT is converted to tinyint by C API
                    col_info[i].column_type  = DT_TINYINT;
                    break;
            }
            col_info[i].column_width = info.max_size;
            col_info[i].column_index = i + 1; // column indices are origin 1
            col_info[i].column_can_be_null = info.nullable;
        }
    }
    // send the result set description
    if( extapi->set_value( arg_handle,
                           EXTFN_RESULT_SET_ARG_NUM,
                           (an_extfn_value *)&rs_info,
                           EXTFN_RESULT_SET_DESCRIBE ) == 0 )
    {
        // failed
        free( col_info );
        free( col_data );
        return;
    }
```

一旦描述了结果集，就可以返回结果集行。以下代码段显示了如何返回结果集的行。它使用 SQL Anywhere C API 为先前由 C API 执行的 SQL 查询读取行。由 SQL Anywhere C API 返回的行被发送回调用环境，一次发回一行。返回各行之前，必须先填充列数据值结构的数组。列数据值结构包括列索引、指向数据值的指针、数据长度和附加标志。

```
    a_sqlany_data_value *value = (a_sqlany_data_value *)
        malloc( columns * sizeof(a_sqlany_data_value) );

    while( capi.sqlany_fetch_next( sqlany_stmt ) )
    {
        for( int i = 0; i < columns; i++ )
        {
            if( capi.sqlany_get_column( sqlany_stmt, i, &value[i] ) )
            {
                col_data[i].column_index = i + 1;
                col_data[i].column_data  = value[i].buffer;
                col_data[i].data_length  = (a_sql_uint32)*(value[i].length);
                col_data[i].append  = 0;
                if( *(value[i].is_null) )
                {
                    // Received a NULL value
                    col_data[i].column_data = NULL;
                }
            }
        }
        if( extapi->set_value(  arg_handle,
                                EXTFN_RESULT_SET_ARG_NUM,
                                (an_extfn_value *)&rs_info,
                                EXTFN_RESULT_SET_NEW_ROW_FLUSH ) == 0 )
        {
            // failed
```



```
            free( value );
            free( col_data );
            free( col_data );
            extapi->set_value( arg_handle, 0, &retval, 0 );
            return;
        }
    }
```

有关更详细信息，请参见“SQL Anywhere 外部函数 API”第 641 页。

有关如何发出服务器端请求以及如何从外部函数返回结果集的详细信息，请参见 *samples-dir\SQLAnywhere\ExternalEnvironments\ExternC* 中的示例。

Java 外部环境

SQL Anywhere 支持 Java 存储过程和函数。Java 存储过程或函数的行为与 SQL 存储过程或函数的基本相同，只是过程或函数的代码以 Java 编写并且在数据库服务器外（即在 Java 虚拟机环境内）执行。应该注意的是，每个数据库对应于一个 Java VM 实例，而不是每个连接对应于一个实例。Java 存储过程可以返回结果集。

在数据库支持中使用 Java 有几个前提条件：

1. 必须在数据库服务器计算机上安装 Java 运行时环境的副本。
2. SQL Anywhere 数据库服务器必须能够找到 Java 可执行文件 (Java VM)。

要在数据库中使用 Java，需确保数据库服务器能够找到并启动 Java 可执行文件。通过执行以下语句可验证这一点：

```
START EXTERNAL ENVIRONMENT JAVA;
```

如果数据库服务器未能启动 Java，问题的原因可能是数据库服务器不能找到 Java 可执行文件。这种情况下，应执行 ALTER EXTERNAL ENVIRONMENT 语句来显式设置 Java 可执行文件的位置。务必要包含可执行文件名。

```
ALTER EXTERNAL ENVIRONMENT JAVA
  LOCATION 'java-path';
```

例如：

```
ALTER EXTERNAL ENVIRONMENT JAVA
  LOCATION 'c:\\jdk1.6.0\\jre\\bin\\java.exe';
```

请注意，除了验证数据库服务器可以启动 Java VM 外，START EXTERNAL ENVIRONMENT JAVA 语句并不是必需的。通常，进行 Java 存储过程或函数的调用会自动启动 Java VM。

类似地，停止 Java 的实例时 STOP EXTERNAL ENVIRONMENT JAVA 语句也不是必需的，因为在数据库的所有连接终止时该实例会自动消失。然而，如果要彻底离开 Java 并且想要释放一些资源，STOP EXTERNAL ENVIRONMENT JAVA 语句则可以减少 Java VM 的使用量。

验证了数据库服务器可以启动 Java VM 可执行文件后，接下来要做的事就是在数据库中安装所需的 Java 类代码。使用 INSTALL JAVA 语句即可完成。例如，可以执行以下语句来将 Java 类从文件安装到数据库。

```
INSTALL JAVA
NEW
FROM FILE 'java-class-file';
```

也可以将 Java JAR 文件安装到数据库中。

```
INSTALL JAVA
NEW
JAR 'jar-name'
FROM FILE 'jar-file';
```

可以从变量安装 Java 类，如下所示：

```
CREATE VARIABLE JavaClass LONG VARCHAR;
SET JavaClass = xp_read_file('java-class-file')
```



```
INSTALL JAVA
NEW
FROM JavaClass;
```

可以从变量安装 Java JAR 文件，如下所示：

```
CREATE VARIABLE JavaJar LONG VARCHAR;
SET JavaJar = xp_read_file('jar-file')
INSTALL JAVA
NEW
JAR 'jar-name'
FROM JavaJar;
```

要从数据库中删除 Java 类，请使用 REMOVE JAVA 语句，如下所示：

```
REMOVE JAVA CLASS 'java-class'
```

要从数据库中删除 Java JAR，请使用 REMOVE JAVA 语句，如下所示：

```
REMOVE JAVA JAR 'jar-name'
```

要修改现有 Java 类，可以使用 INSTALL JAVA 语句的 UPDATE 子句，如下所示：

```
INSTALL JAVA
UPDATE
FROM FILE 'java-class-file'
```

也可以在数据库中更新现有 Java JAR 文件。

```
INSTALL JAVA
UPDATE
JAR 'jar-name'
FROM FILE 'jar-file';
```

可以从变量更新 Java 类，如下所示：

```
CREATE VARIABLE JavaClass LONG VARCHAR;
SET JavaClass = xp_read_file('java-class-file')
INSTALL JAVA
UPDATE
FROM JavaClass;
```

可以从变量更新 Java JAR 文件，如下所示：

```
CREATE VARIABLE JavaJar LONG VARCHAR;
SET JavaJar = xp_read_file('jar-file')
INSTALL JAVA
UPDATE
FROM JavaJar;
```

Java 类安装在数据库中后，接下来可以创建存储过程和函数，以与 Java 方法连接。EXTERNAL NAME 字符串包含了调用 Java 方法以及返回 OUT 参数和返回值所需的信息。EXTERNAL NAME 子句的 LANGUAGE 属性必须指定 JAVA。EXTERNAL NAME 子句的格式是：

EXTERNAL NAME '*java-call*' LANGUAGE JAVA

java-call :
[*package-name*.]*class-name.method-name method-signature*

method-signature :
([*field-descriptor*, ...]) *return-descriptor*

field-descriptor and *return-descriptor* :
Z
| **B**
| **S**
| **I**
| **J**
| **F**
| **D**
| **C**
| **V**
| **[***descriptor*
| **L***class-name*;

Java 方法签名是参数类型和返回值类型的压缩字符表示形式。如果参数的数目比方法签名中指出的数目少，则差值必须等于 DYNAMIC RESULT SETS 中指定的个数，并且方法签名中超出过程参数列表中参数的每个参数必须具有方法签名 `[Ljava/SQL/ResultSet;`。

field-descriptor 和 *return-descriptor* 具有以下含义：

字段类型	Java 数据类型
B	byte
C	char
D	double
F	float
I	int
J	long
L *class-name*;	类 *class-name* 的实例。类名必须是完全限定的，而且名称中的任何点都必须替换为 /。例如 java/lang/String
S	short
V	void
Z	Boolean
[	数组的每个维度都使用一个

例如，

```
double some_method(
  boolean a,
  int b,
  java.math.BigDecimal c,
  byte [][] d,
```



```
  java.sql.ResultSet[] rs ) {
}
```

可以有以下签名：

```
'(ZILjava/math/BigDecimal;[[B[Ljava/SQL/ResultSet;)D'
```

以下过程创建 Java 方法的接口。Java 方法不返回任何值 (V)。

```
CREATE PROCEDURE insertfix()
EXTERNAL NAME 'JDBCExample.InsertFixed()V'
LANGUAGE JAVA;
```

以下过程创建具有字符串 ([Ljava/lang/String;) 输入参数的 Java 方法的接口。Java 方法不返回任何值 (V)。

```
CREATE PROCEDURE InvoiceMain( IN arg1 CHAR(50) )
EXTERNAL NAME 'Invoice.main([Ljava/lang/String;)V'
LANGUAGE JAVA;
```

以下过程创建 Java 方法 Invoice.init（采用一个字符串参数 (Ljava/lang/String;)、一个双精度参数 (D)、另一个字符串参数 (Ljava/lang/String;) 和另一个双精度参数 (D)）的接口，并且不返回任何值 (V)。

```
CREATE PROCEDURE init( IN arg1 CHAR(50),
                       IN arg2 DOUBLE,
                       IN arg3 CHAR(50),
                       IN arg4 DOUBLE)
EXTERNAL NAME 'Invoice.init(Ljava/lang/String;DLjava/lang/String;D)V'
LANGUAGE JAVA
```

有关调用 Java 方法的详细信息，请参见“访问 Java 类中的方法”一节第 89 页。

有关返回结果集的详细信息，请参见“从 Java 方法返回结果集”一节第 97 页。

下面的 Java 示例使用一个字符串参数并将其写入数据库服务器消息窗口：

```
import java.io.*;

public class Hello
{
    public static void main( String[] args )
    {
        System.out.print( "Hello" );
               for ( int i = 0; i  < args.length; i++ )
                   System.out.print( " " + args[i] );
        System.out.println();
    }
}
```

以上 Java 代码位于 *Hello.java* 文件中，并使用 Java 编译器进行编译。所生成的类文件将装载到数据库中，如下所示。

```
INSTALL JAVA
NEW
FROM FILE 'Hello.class';
```

使用 Interactive SQL，用于在 `Hello` 类中连接方法 `main` 的存储过程将按如下方式创建：

```
CREATE PROCEDURE HelloDemo( IN name LONG VARCHAR )
EXTERNAL NAME 'Hello.main([Ljava/lang/String;)V'
LANGUAGE JAVA;
```

请注意，`main` 的参数以 java.lang.String 的数组形式进行描述。使用 Interactive SQL，通过执行以下 SQL 语句来测试该接口。

```
CALL HelloDemo('SQL Anywhere');
```

如果检查数据库服务器消息窗口，将会找到此处写入的消息。所有到 System.out 的输出将重定向到服务器消息窗口。

有关在数据库支持中使用 Java 的详细信息和示例，请参见“SQL Anywhere 中的 Java 支持” 第 73 页。

PERL 外部环境

SQL Anywhere 支持 Perl 存储过程和函数。Perl 存储过程或函数的行为与 SQL 存储过程或函数基本相同，只是过程或函数的代码以 Perl 编写并在数据库服务器外（即在 Perl 可执行实例内）执行。值得注意的是，对于使用 Perl 存储过程和函数的每个连接，会有一个单独的 Perl 可执行文件实例。这一点不同于 Java 存储过程和函数。如果是 Java，针对每个数据库有一个 Java VM 实例，而不是每个连接一个实例。Perl 和 Java 间的另一个主要差异是，Perl 存储过程不返回结果集，而 Java 存储过程可以返回结果集。

在数据库支持中使用 Perl 有几个前提条件：

1. 必须将 Perl 安装在数据库服务器计算机上，并且 SQL Anywhere 数据库服务器必须能够找到 Perl 可执行文件。
2. 必须在数据库服务器计算机上安装 DBD::SQLAnywhere 驱动程序。
3. 在 Windows 上还必须安装 Microsoft Visual Studio。它对于安装 DBD::SQLAnywhere 驱动程序是必要的，因此这是一个前提条件。

有关安装 DBD::SQLAnywhere 驱动程序的详细信息，请参见“SQL Anywhere Perl DBD::SQLAnywhere DBI 模块”第 693 页。

除了以上前提条件外，数据库管理员还必须安装 SQL Anywhere Perl External Environment 模块。安装外部环境模块：

♦ 安装外部环境模块 (Windows)

- 在 SQL Anywhere 安装目录的 *SDK\PerlEnv* 子目录中运行以下命令：

```
perl Makefile.PL
nmake
nmake install
```

♦ 安装外部环境模块 (Unix)

- 在 SQL Anywhere 安装目录的 *sdk/perlenv* 子目录中运行以下命令：

```
perl Makefile.PL
make
make install
```

构建和安装 Perl 外部环境模块后，数据库支持中的 Perl 即可使用。请注意，数据库支持中的 Perl 仅适用于 SQL Anywhere 版本 11 或更高版本数据库。如果装载 SQL Anywhere 10 数据库，则在数据库支持中尝试使用 Perl 时会返回指示不支持外部环境的错误消息。

要在数据库中使用 Perl，需确保数据库服务器能够找到并启动 Perl 可执行文件。通过执行以下语句可验证这一点：

```
START EXTERNAL ENVIRONMENT PERL;
```

如果数据库服务器未能启动 Perl，导致问题的原因可能是数据库服务器无法找到 Perl 可执行文件。这种情况下，应执行 ALTER EXTERNAL ENVIRONMENT 语句来显式设置 Perl 可执行文件的位置。务必要包含可执行文件名。


```
ALTER EXTERNAL ENVIRONMENT PERL
  LOCATION 'perl-path';
```

例如：

```
ALTER EXTERNAL ENVIRONMENT PERL
  LOCATION 'c:\\Perl\\bin\\perl.exe';
```

请注意，除了验证数据库服务器可以启动 Perl 外，START EXTERNAL ENVIRONMENT PERL 语句并不是必需的。通常，进行 Perl 存储过程或函数的调用会自动启动 Perl。

类似地，停止 Perl 的实例时，STOP EXTERNAL ENVIRONMENT PERL 语句也不是必需的，因为连接终止时实例会自动消失。然而，如果要彻底离开 Perl 并且想要释放一些资源，则 STOP EXTERNAL ENVIRONMENT PERL 语句可以为您的连接释放该 Perl 实例。

验证数据库服务器可以启动 Perl 可执行文件后，要做的下一件事就是在数据库中安装所需的 Perl 代码。使用 INSTALL 语句即可完成。例如，可以执行以下语句来将 Perl 脚本从文件安装到数据库。

```
INSTALL EXTERNAL OBJECT 'perl-script'
  NEW
  FROM FILE 'perl-file'
  ENVIRONMENT PERL;
```

也可以从表达式构建和安装 Perl 代码，如下所示：

```
INSTALL EXTERNAL OBJECT 'perl-script'
  NEW
  FROM VALUE 'perl-statements'
  ENVIRONMENT PERL;
```

还可以从变量构建和安装 Perl 代码，如下所示：

```
CREATE VARIABLE PerlVariable LONG VARCHAR;
SET PerlVariable = 'perl-statements';
INSTALL EXTERNAL OBJECT 'perl-script'
  NEW
  FROM VALUE PerlVariable
  ENVIRONMENT PERL;
```

要从数据库中删除 Perl 代码，请使用 REMOVE 语句，如下所示：

```
REMOVE EXTERNAL OBJECT 'perl-script'
```

要修改现有 Perl 代码，可以使用 INSTALL EXTERNAL OBJECT 语句的 UPDATE 子句，如下所示：

```
INSTALL EXTERNAL OBJECT 'perl-script'
  UPDATE
  FROM FILE 'perl-file'
  ENVIRONMENT PERL

INSTALL EXTERNAL OBJECT 'perl-script'
  UPDATE
  FROM VALUE 'perl-statements'
  ENVIRONMENT PERL

SET PerlVariable = 'perl-statements';
INSTALL EXTERNAL OBJECT 'perl-script'
  UPDATE
```



```
  FROM VALUE PerlVariable
  ENVIRONMENT PERL
```

Perl 代码安装在数据库中后，接下来可以创建所需的 Perl 存储过程和函数。创建 Perl 存储过程和函数时，LANGUAGE 始终是 PERL，EXTERNAL NAME 字符串包含调用 Perl 子例程和返回 OUT 参数及返回值所需的信息。每次调用时 Perl 代码可使用以下全局变量：

- **$sa_perl_return**　用于设置函数调用的返回值。
- **$sa_perl_argN**　其中 N 是正整数 [0 .. n]。用于将 SQL 参数传递给 Perl 代码。例如，`$sa_perl_arg0` 指参数 0，而 `$sa_perl_arg1` 指参数 1，以此类推。
- **$sa_perl_default_connection**　用于进行服务器端的 Perl 调用。
- **$sa_output_handle**　用于将 Perl 代码的输出发送给数据库服务器消息窗口。

创建 Perl 存储过程时，其输入和输出参数以及返回值可以采用任何一组数据类型。但是，在进行 Perl 调用时，所有非二进制数据类型会映射到字符串，而二进制数据映射到数值数组。下面是一个简单的 Perl 示例：

```
INSTALL EXTERNAL OBJECT 'SimplePerlExample'
  NEW
  FROM VALUE 'sub SimplePerlSub{
    return( ($_[0] * 1000) +
            ($_[1] * 100) +
            ($_[2] * 10) +
           $_[3] );
  }'
  ENVIRONMENT PERL;

CREATE FUNCTION SimplePerlDemo(
    IN thousands INT,
    IN hundreds INT,
    IN tens INT,
    IN ones INT)
  RETURNS INT
  EXTERNAL NAME '<file=SimplePerlExample>
    $sa_perl_return = SimplePerlSub(
      $sa_perl_arg0,
      $sa_perl_arg1,
      $sa_perl_arg2,
      $sa_perl_arg3)'
  LANGUAGE PERL;

// The number 1234 should appear
SELECT SimplePerlDemo(1,2,3,4);
```

下面的 Perl 示例使用一个字符串参数并将其写入数据库服务器消息窗口：

```
INSTALL EXTERNAL OBJECT 'PerlConsoleExample'
  NEW
  FROM VALUE 'sub WriteToServerConsole { print $sa_output_handle $_[0]; }'
  ENVIRONMENT PERL;

CREATE PROCEDURE PerlWriteToConsole( IN str LONG VARCHAR)
  EXTERNAL NAME '<file=PerlConsoleExample>
    WriteToServerConsole( $sa_perl_arg0 )'
  LANGUAGE PERL;
```

```
// 'Hello world' should appear in the database server messages window
CALL PerlWriteToConsole( 'Hello world' );
```

要使用服务器端 Perl，Perl 代码必须使用 $sa_perl_default_connection 变量。下面的示例将创建一个表，然后调用 Perl 存储过程来填充该表：

```
CREATE TABLE perlTab(c1 int, c2 char(128));

INSTALL EXTERNAL OBJECT 'ServerSidePerlExample'
  NEW
  FROM VALUE 'sub ServerSidePerlSub
    { $sa_perl_default_connection->do(
        "INSERT INTO perlTab SELECT table_id, table_name FROM SYS.SYSTAB" );
      $sa_perl_default_connection->do(
        "COMMIT" );
    }'
  ENVIRONMENT PERL;

CREATE PROCEDURE PerlPopulateTable()
  EXTERNAL NAME '<file=ServerSidePerlExample> ServerSidePerlSub()'
  LANGUAGE PERL;

CALL PerlPopulateTable();

// The following should return 2 identical rows
SELECT count(*) FROM perlTab
UNION ALL
SELECT count(*) FROM SYS.SYSTAB;
```

有关在数据库支持中使用 Perl 的更详细信息和示例，请参见位于 *samples-dir\SQLAnywhere\ExternalEnvironments\Perl* 目录中的示例。

PHP 外部环境

SQL Anywhere 支持 PHP 存储过程和函数。PHP 存储过程或函数的行为与 SQL 存储过程或函数基本相同，只是过程或函数的代码以 PHP 编写并且在数据库服务器外（即在 PHP 可执行实例内）执行。对于使用 PHP 存储过程和函数的每个连接，会有一个单独的 PHP 可执行文件实例。这一点与 Java 存储过程和函数有很大不同。如果是 Java，针对每个数据库有一个 Java VM 实例，而不是每个连接一个实例。PHP 和 Java 间的另一个主要差异是 PHP 存储过程不返回结果集，而 Java 存储过程可以返回结果集。PHP 仅返回 LONG VARCHAR 类型的对象，它是 PHP 脚本的输出。

在数据库支持中使用 PHP 有两个前提条件：

1. 必须将 PHP 的副本安装在数据库服务器计算机上，并且 SQL Anywhere 数据库服务器必须能够找到 PHP 可执行文件。
2. 在数据库服务器计算机上必须安装 SQL Anywhere PHP 驱动程序（随 SQL Anywhere 一起提供）。请参见“安装和配置 SQL Anywhere PHP”一节第 713 页。

除了以上两个前提条件外，数据库管理员还必须安装 SQL Anywhere PHP External Environment 模块。有些 PHP 版本的预建模块随 SQL Anywhere 发行包一起提供。要安装预建模块，将相应的驱动程序复制模块到 PHP 扩展目录中（可以在 *php.ini* 中找到）。在 Unix 中，还可以使用符号链接。

♦ 安装外部环境模块 (Windows)

1. 找到 PHP 安装目录中的 *php.ini* 文件，并在文本编辑器中将其打开。找到指定 **extension_dir** 目录位置的行。如果未将 **extension_dir** 设置到任何特定目录，则为获得更好的系统安全，最好将其设置为指向一个单独的目录。
2. 将所需的外部环境 PHP 模块从 SQL Anywhere 安装目录复制到 PHP 扩展目录。以下是一个可使用的模型：

   ```
   copy install-dir\Bin32\php-5.2.6_sqlanywhere_extenv11.dll
     php-dir\ext
   ```

3. 请确保您同时还将 SQL Anywhere PHP 驱动程序从 SQL Anywhere 安装目录安装到了 PHP 扩展目录。此文件名遵循模式 *php-5.x.y_sqlanywhere.dll*，其中 x 和 y 为版本号。它们应该与在步骤 2 中所复制的文件的版本号相匹配。

♦ 安装外部环境模块 (Unix)

1. 找到 PHP 安装目录中的 *php.ini* 文件，并在文本编辑器中将其打开。找到指定 **extension_dir** 目录位置的行。如果未将 **extension_dir** 设置到任何特定目录，则为获得更好的系统安全，最好将其设置为指向一个单独的目录。
2. 将所需的外部环境 PHP 模块从 SQL Anywhere 安装目录复制到 PHP 安装目录。以下是一个可使用的模型：

   ```
   cp install-dir/bin32/php-5.2.6_sqlanywhere_extenv11.so
     php-dir/ext
   ```

3. 请确保您同时还将 SQL Anywhere PHP 驱动程序从 SQL Anywhere 安装目录安装到了 PHP 扩展目录。此文件名遵循模式 *php-5.x.y_sqlanywhere.so*，其中 x 和 y 为版本号。它们应该与在步骤 2 中所复制的文件的版本号相匹配。

数据库支持中的 PHP 仅适用于 SQL Anywhere 版本 11 或更高版本数据库。如果装载 SQL Anywhere 10 数据库，则在数据库支持中尝试使用 PHP 时会返回指示不支持外部环境的错误消息。

要在数据库中使用 PHP，数据库服务器必须能够找到并启动 PHP 可执行文件。通过执行以下语句可以验证数据库服务器是否能够找到并启动 PHP 可执行文件：

```
START EXTERNAL ENVIRONMENT PHP;
```

如果消息指示找不到 "外部可执行文件"，则该问题的原因是数据库服务器无法找到 PHP 可执行文件。这种情况下，应执行 ALTER EXTERNAL ENVIRONMENT 语句来显式设置 PHP 可执行文件的位置（包括可执行文件名），或者确保具有 PHP 可执行文件的目录包含在 PATH 环境变量中。

```
ALTER EXTERNAL ENVIRONMENT PHP
  LOCATION 'php-path';
```

例如：

```
ALTER EXTERNAL ENVIRONMENT PHP
  LOCATION 'c:\\php\\php-5.2.6-win32\\php.exe';
```

要恢复缺省设置，请执行以下语句：

```
ALTER EXTERNAL ENVIRONMENT PHP
  LOCATION 'php';
```

如果一条消息指出找不到 "主线程"，请检查以下各项：

- 请确保 *php-5.x.y_sqlanywhere* 和 *php-5.x.y_sqlanywhere_extenv11* 模块都位于 **extension_dir** 指示的目录中。检查上述安装步骤。
- 请确保可以找到 *phpenv.php*。检查以确定 SQL Anywhere *bin32* 文件夹在 PATH 中。
- 对于 Windows，请确保可以找到 32 位 DLL（*dbcapi.dll*、*dblib11.dll*、*dbicu11.dll*、*dbicudt11.dll*、*dblgen11.dll* 和 *dbextenv11.dll*）。检查以确定 SQL Anywhere *bin32* 文件夹在 PATH 中。
- 对于 Linux、Unix 和 Mac OS X，请确保可以找到 32 位共享对象（*libdbcapi_r*、*libdblib11_r*、*libdbicu11_r*、*libdbicudt11*、*dblgen11.res* 和 *libdbextenv11_r*）。检查以确定 SQL Anywhere *bin32* 文件夹在 PATH 中。
- 请确保未设置环境变量 **PHPRC**，或确保它指向要使用的 PHP 版本。

除了验证数据库服务器可以启动 PHP 外，START EXTERNAL ENVIRONMENT PHP 语句并不是必需的。通常，进行 PHP 存储过程或函数的调用会自动启动 PHP。

类似地，停止 PHP 的实例时 STOP EXTERNAL ENVIRONMENT PHP 语句也不是必需的，因为在连接终止时实例会自动消失。然而，如果要彻底离开 PHP 并且想要释放一些资源，则 STOP EXTERNAL ENVIRONMENT PHP 语句则可以为您的连接释放该 PHP 实例。

验证数据库服务器可以启动 PHP 可执行文件后，要做的下一件事就是在数据库中安装所需的 PHP 代码。使用 INSTALL 语句即可完成。例如，可以执行以下语句来将特定 PHP 脚本安装到数据库。

```
INSTALL EXTERNAL OBJECT 'php-script'
  NEW
  FROM FILE 'php-file'
  ENVIRONMENT PHP;
```

也可以从表达式构建和安装 PHP 代码，如下所示：

```
INSTALL EXTERNAL OBJECT 'php-script'
  NEW
```



```
  FROM VALUE 'php-statements'
  ENVIRONMENT PHP;
```

还可以从变量构建和安装 PHP 代码，如下所示：

```
CREATE VARIABLE PHPVariable LONG VARCHAR;
SET PHPVariable = 'php-statements';
INSTALL EXTERNAL OBJECT 'php-script'
  NEW
  FROM VALUE PHPVariable
  ENVIRONMENT PHP;
```

要从数据库中删除 PHP 代码，请使用 REMOVE 语句，如下所示：

```
REMOVE EXTERNAL OBJECT 'php-script';
```

要修改现有 PHP 代码，可以使用 INSTALL 语句的 UPDATE 子句，如下所示：

```
INSTALL EXTERNAL OBJECT 'php-script'
  UPDATE
  FROM FILE 'php-file'
  ENVIRONMENT PHP;

INSTALL EXTERNAL OBJECT 'php-script'
  UPDATE
  FROM VALUE 'php-statements'
  ENVIRONMENT PHP;

SET PHPVariable = 'php-statements';
INSTALL EXTERNAL OBJECT 'php-script'
  UPDATE
  FROM VALUE PHPVariable
  ENVIRONMENT PHP;
```

PHP 代码安装在数据库中后，接下来可以继续创建所需的 PHP 存储过程和函数。创建 PHP 存储过程和函数时，LANGUAGE 始终是 PHP，EXTERNAL NAME 字符串包含调用 PHP 子例程和返回 OUT 参数所需的信息。

参数通过 $argv 数组传递给 PHP 脚本，这与 PHP 从命令行获取参数的方式类似（即 $argv[1] 为第一个参数）。要设置输出参数，请将其赋值给相应的 $argv 元素。返回值始终是脚本的输出（LONG VARCHAR 数据类型）。

创建 PHP 存储过程时，其输入和输出参数可以采用任何一组数据类型。但为了在 PHP 脚本内部使用，这些参数会转换为（或者反向转换）布尔值、整数、双精度值或字符串。返回值始终是 LONG VARCHAR 类型的对象。以下是一个简单的 PHP 示例：

```
INSTALL EXTERNAL OBJECT 'SimplePHPExample'
  NEW
  FROM VALUE '<? function SimplePHPFunction(
    $arg1, $arg2, $arg3, $arg4 )
    { return ($arg1 * 1000) +
      ($arg2 * 100) +
      ($arg3 * 10) +
      $arg4;
    } ?>'
  ENVIRONMENT PHP;

CREATE FUNCTION SimplePHPDemo(
  IN thousands INT,
  IN hundreds INT,
```

```
  IN tens INT,
  IN ones INT)
RETURNS LONG VARCHAR
EXTERNAL NAME '<file=SimplePHPExample> print SimplePHPFunction(
    $argv[1], $argv[2], $argv[3], $argv[4]);'
LANGUAGE PHP;

// The number 1234 should appear
SELECT SimplePHPDemo(1,2,3,4);
```

对于 PHP，EXTERNAL NAME 字符串以 SQL 的单独一行指定。

要使用服务器端 PHP，PHP 代码可以使用缺省数据库连接。要获取数据库连接的句柄，请以空字符串参数（" 或 ""）调用 sasql_pconnect。空字符串参数会指示 SQL Anywhere PHP 驱动程序返回当前外部环境连接，而不是打开一个新连接。下面的示例将创建一个表，然后调用 PHP 存储过程来填充该表。

```
CREATE TABLE phpTab(c1 int, c2 char(128));

INSTALL EXTERNAL OBJECT 'ServerSidePHPExample'
  NEW
  FROM VALUE '<? function ServerSidePHPSub() {
    $conn = sasql_pconnect( '''' );
    sasql_query( $conn,
    "INSERT INTO phpTab
       SELECT table_id, table_name FROM SYS.SYSTAB" );
    sasql_commit( $conn );
  } ?>'
  ENVIRONMENT PHP;

CREATE PROCEDURE PHPPopulateTable()
EXTERNAL NAME '<file=ServerSidePHPExample> ServerSidePHPSub()'
LANGUAGE PHP;

CALL PHPPopulateTable();

// The following should return 2 identical rows
SELECT count(*) FROM phpTab
UNION ALL
SELECT count(*) FROM SYS.SYSTAB;
```

对于 PHP，EXTERNAL NAME 字符串以 SQL 的单独一行指定。请注意，在上面的示例中，由于引号在 SQL 中的分析方式，单引号都是双写的。如果 PHP 源代码是在文件中，单引号则不用双写。

要将错误返回给数据库服务器，可抛出一个 PHP 异常。下面举例说明如何实现这一目的。

```
CREATE TABLE phpTab(c1 int, c2 char(128));

INSTALL EXTERNAL OBJECT 'ServerSidePHPExample'
  NEW
  FROM VALUE '<? function ServerSidePHPSub() {
    $conn = sasql_pconnect( '''' );
    if( !sasql_query( $conn,
      "INSERT INTO phpTabNoExist
         SELECT table_id, table_name FROM SYS.SYSTAB" )
    ) throw new Exception(
      sasql_error( $conn ),
      sasql_errorcode( $conn )
    );
    sasql_commit( $conn );
```



```
    } ?>'
    ENVIRONMENT PHP;

  CREATE PROCEDURE PHPPopulateTable()
    EXTERNAL NAME
      '<file=ServerSidePHPExample> ServerSidePHPSub()'
    LANGUAGE PHP;

  CALL PHPPopulateTable();
```

上面的示例将以指示无法找到 phpTabNoExist 表的错误 SQLE_UNHANDLED_EXTENV_EXCEPTION 终止。

有关在数据库支持中使用 PHP 的更详细信息和示例，请参见位于 *samples-dir\SQLAnywhere \ExternalEnvironments\PHP* 目录中的示例。

SQL Anywhere Perl DBD::SQLAnywhere DBI 模块

目录

DBD::SQLAnywhere 简介

通过 DBD::SQLAnywhere 接口，可从使用 Perl 编写的脚本访问 SQL Anywhere 数据库。DBD::SQLAnywhere 是由 Tim Bunce 编写的用于 Perl 的数据库独立接口 (DBI) 模块的驱动程序。在安装 DBI 模块和 DBD::SQLAnywhere 后，就可以使用 Perl 来访问和更改 SQL Anywhere 数据库中的信息了。

使用 Perl ithread 模式时，DBD::SQLAnywhere 驱动程序是线程安全的。

要求

DBD::SQLAnywhere 接口要求以下组件。

- Perl 5.6.0 或更高版本。在 Windows 上，要求 ActivePerl 5.6.0 build 616 或更高版本。
- DBI 1.34 或更高版本。
- C 编译器。在 Windows 上只支持 Microsoft Visual C++ 编译器。

以下各节提供有关安装 Perl、DBI 和 DBD::SQLAnywhere 驱动程序软件方面的帮助。

在 Windows 上安装 DBD::SQLAnywhere

♦ 准备计算机

1. 安装 ActivePerl 5.6.0 或更高版本。您可以使用 ActivePerl 安装程序安装 Perl 并配置计算机。无需重新编译 Perl。
2. 安装 Microsoft Visual Studio 并配置环境。

   如果没有选择在安装时配置环境，则必须正确设置 PATH、LIB 和 INCLUDE 环境变量才能继续。Microsoft 为此提供了一个批处理文件。例如，Visual Studio 2005 或 2008 安装目录的 *vc\bin* 子目录中提供了名为 *vcvars32.bat* 的批处理文件。打开一个新的系统命令提示符并运行此批处理文件，然后再继续。

♦ 在 Windows 上安装 DBI Perl 模块

1. 在命令提示符处，转到 ActivePerl 安装目录的 *bin* 子目录。

   强烈建议使用该系统命令提示符，因为下面的步骤可能无法从其它 shell 运行。
2. 通过 Perl Module Manager，输入以下命令。

   ```
   ppm query dbi
   ```

   如果 ppm 无法运行，请检查是否正确地安装了 Perl。

   该命令应生成两行如下所示的文本。在此情况下，该信息指示 ActivePerl version 5.8.1 build 807 正在运行且 DBI 版本 1.38 已安装。

   ```
   Querying target 1 (ActivePerl 5.8.1.807)
     1. DBI [1.38] Database independent interface for Perl
   ```

   对于更新版本的 Perl，则可能会显示如下所示的表。这种情况下，该信息指示已安装了 DBI 1.58 版。

name	version	abstract	area
DBI	1.58	Database independent interface for Perl	perl

   如果没有安装 DBI，则必须安装。为此，请在 ppm 提示符处输入以下命令。

   ```
   ppm install dbi
   ```

♦ 在 Windows 上安装 DBD::SQLAnywhere

1. 在命令提示符处，转到 SQL Anywhere 安装目录的 *SDK\Perl* 子目录。
2. 输入以下命令生成并测试 DBD::SQLAnywhere。

   ```
   perl Makefile.PL

   nmake
   ```

如果出于任何原因您需要从头开始，则可以运行 **nmake clean** 命令删除所有部分生成的目标。

3. 要测试 DBD::SQLAnywhere，请将示例数据库文件复制到 *SDK\Perl* 目录下，然后进行测试。

   ```
   copy "samples-dir\demo.db" .
   ```

   有关 *samples-dir* 缺省位置的信息，请参见“示例目录”一节 《SQL Anywhere 服务器 - 数据库管理》。

   ```
   dbeng11 demo
   ```

   ```
   nmake test
   ```

   如果测试没有运行，请确保路径中包含 SQL Anywhere 安装目录的 *bin32* 或 *bin64* 子目录。

4. 要完成安装，请在同一提示符下执行以下命令。

   ```
   nmake install
   ```

现在就可以使用 DBD::SQLAnywhere 接口了。

在 Unix 和 Mac OS X 上安装 DBD::SQLAnywhere

下面的过程介绍如何在支持的 Unix 平台（包括 Mac OS X）上安装 DBD::SQLAnywhere 接口。

♦ 准备计算机

1. 安装 ActivePerl 5.6.0 build 616 或更高版本。
2. 安装 C 编译器。

♦ 在 Unix 和 Mac OS X 上安装 DBI Perl 模块

1. 从 www.cpan.org 下载 DBI 模块源。
2. 将该文件的内容抽取到一个新目录中。
3. 在命令提示符处，更改为该新目录并执行以下命令以生成 DBI 模块。

   ```
   perl Makefile.PL
   make
   ```

   如果出于任何原因您需要从头开始，则可以使用 **make clean** 命令删除所有部分生成的目标。
4. 使用下面的命令测试 DBI 模块。

   ```
   make test
   ```
5. 要完成安装，请在同一提示符下执行以下命令。

   ```
   make install
   ```
6. 您现在可以选择删除 DBI 源树。不再需要该源树。

♦ 在 Unix 和 Mac OS X 上安装 DBD::SQLAnywhere

1. 确保设置了适用于 SQL Anywhere 的环境。

   根据您使用的 shell，输入相应的命令以从 SQL Anywhere 安装目录执行 SQL Anywhere 配置脚本：

在此 shell 中……	... 使用此命令
sh、ksh 或 bash	. *bin/sa_config.sh*
csh 或 tcsh	source *bin/sa_config.csh*

2. 在 shell 提示符处，转到 SQL Anywhere 安装目录的 *sdk/perl* 子目录。
3. 在命令提示符处，运行以下命令来生成 DBD::SQLAnywhere。

   ```
   perl Makefile.PL
   make
   ```

如果出于任何原因您需要从头开始，则可以使用 **make clean** 命令删除所有部分生成的目标。

4. 要测试 DBD::SQLAnywhere，请将示例数据库文件复制到 *sdk/perl* 目录下，然后进行测试。

```
cp samples-dir/demo.db .
dbeng11 demo
make test
```

如果测试没有运行，请确保路径中包含 SQL Anywhere 安装目录的 *bin32* 或 *bin64* 子目录。

5. 要完成安装，请在同一提示符下执行以下命令。

```
make install
```

现在就可以使用 DBD::SQLAnywhere 接口了。

编写使用 DBD::SQLAnywhere 的 Perl 脚本

本节将概述如何编写使用 DBD::SQLAnywhere 接口的 Perl 脚本。DBD::SQLAnywhere 是 DBI 模块的驱动程序。DBI 模块的完整文档可从 dbi.perl.org 联机获得。

装载 DBI 模块

要在 Perl 脚本中使用 DBD::SQLAnywhere 接口，必须先告诉 Perl 您打算使用 DBI 模块。为此，请在文件的顶部包括下面的命令行。

```
use DBI;
```

此外，强烈建议您在严格模式下运行 Perl。例如强制要求显式变量定义的语句，可大大减少由于常见失误（如键入错误）而产生的莫名其妙的错误。

```
#!/usr/local/bin/perl -w
#
use DBI;
use strict;
```

必要时 DBI 模块会自动装载 DBD 驱动程序（包括 DBD::SQLAnywhere）。

打开和关闭连接

通常，打开一个到数据库的连接，然后通过该连接运行一系列 SQL 语句来执行所有需要的操作。要打开连接，请使用 connect 方法。返回值是一个到数据库连接的句柄，使用该句柄可以在连接上执行后继操作。

connect 方法的参数如下所示：

1. "DBI:SQLAnywhere:"和其它连接参数以分号分隔。
2. 用户名。除非该字符串为空，否则 ";UID=*value*" 将附加到连接字符串。
3. 口令值。除非该字符串为空，否则 ";PWD=*value*" 将附加到连接字符串。
4. 指向散列缺省值的指针。诸如 AutoCommit、RaiseError 和 PrintError 的设置可用该方法设置。

以下代码示例打开然后关闭到 SQL Anywhere 示例数据库的连接。必须先启动数据库服务器和示例数据库然后才能运行此脚本。

```
#!/usr/local/bin/perl -w
#
use DBI;
use strict;
my $database = "demo";
my $data_src = "DBI:SQLAnywhere:ENG=$database;DBN=$database";
my $uid      = "DBA";
my $pwd      = "sql";
my %defaults = (
     AutoCommit => 1, # Autocommit enabled.
     PrintError => 0  # Errors not automatically printed.
   );
```

```
my $dbh = DBI->connect($data_src, $uid, $pwd, \%defaults)
  or die "Cannot connect to $data_src: $DBI::errstr\n";
$dbh->disconnect;
exit(0);
__END__
```

或者，您可以选择将用户名或口令值附加到数据源字符串，而不是将它们作为独立参数提供。如果要这样做，请为相应参数提供空白字符串。例如，使用下面的语句替换打开连接的语句可修改上面的脚本：

```
$data_src .= ";UID=$uid";
$data_src .= ";PWD=$pwd";
my $dbh = DBI->connect($data_src, '', '', \%defaults)
  or die "Cannot connect to $data_src: $DBI::errstr\n";
```

选择数据

获得了打开的连接的句柄后，您可以访问和修改存储在数据库中的数据。可能最简单的操作是检索某些行并输出它们。

必须先准备好返回行集的 SQL 语句，然后才能执行它们。prepare 方法为该语句返回一个句柄。使用该句柄执行语句，然后检索关于结果集和结果集的行的元信息。

```
#!/usr/local/bin/perl -w
#
use DBI;
use strict;
my $database = "demo";
my $data_src = "DBI:SQLAnywhere:ENG=$database;DBN=$database";
my $uid      = "DBA";
my $pwd      = "sql";
my $sel_stmt = "SELECT ID, GivenName, Surname
                FROM Customers
                ORDER BY GivenName, Surname";
my %defaults = (
     AutoCommit => 0, # Require explicit commit or rollback.
     PrintError => 0
   );
my $dbh = DBI->connect($data_src, $uid, $pwd, \%defaults)
  or die "Cannot connect to $data_src: $DBI::errstr\n";
&db_query($sel_stmt, $dbh);
$dbh->rollback;
$dbh->disconnect;
exit(0);

sub db_query {
      my($sel, $dbh) = @_;
      my($row, $sth) = undef;
      $sth = $dbh->prepare($sel);
      $sth->execute;
      print "Fields:     $sth->{NUM_OF_FIELDS}\n";
      print "Params:     $sth->{NUM_OF_PARAMS}\n\n";
      print join("\t\t", @{$sth->{NAME}}), "\n\n";
      while($row = $sth->fetchrow_arrayref) {
         print join("\t\t", @$row), "\n";
      }
      $sth = undef;
}
__END__
```


直到 Perl 语句句柄被破坏，预准备语句才会从数据库服务器上删除。要破坏语句句柄，请重新使用变量或将其设置为 undef。调用 finish 方法不会删除句柄。实际不应调用 finish 方法，除非决定不完成读取结果集的操作。

要检测句柄泄漏，缺省情况下 SQL Anywhere 数据库服务器将允许的游标和准备好的语句数量限制为每个连接最多 50 个。如果超过这些限制，资源调控器自动生成错误。如果收到此错误，请检查未被释放的语句句柄。请谨慎使用 prepare_cached，因为语句句柄并没有被释放。

如有必要，可通过设置 max_cursor_count 和 max_statement_count 选项来修改这些限制。请参见“max_cursor_count 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》和“max_statement_count 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

插入行

插入行需要打开的连接的句柄。最简单的方法是使用参数化的 INSERT 语句，这意味着问号用作值的占位符。首先准备好语句，然后对每一新行执行一次。新行的值作为参数提供给 execute 方法。

以下示例程序插入两个新的客户。尽管行值显示为文字字符串，您可能希望从文件中读取这些值。

```
#!/usr/local/bin/perl -w
#
use DBI;
use strict;
my $database = "demo";
my $data_src = "DBI:SQLAnywhere:ENG=$database;DBN=$database";
my $uid      = "DBA";
my $pwd      = "sql";
my $ins_stmt = "INSERT INTO Customers (ID, GivenName, Surname,
                        Street, City, State, Country, PostalCode,
                        Phone, CompanyName)
                  VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)";
my %defaults = (
     AutoCommit => 0, # Require explicit commit or rollback.
     PrintError => 0
   );
my $dbh = DBI->connect($data_src, $uid, $pwd, \%defaults)
     or die "Can't connect to $data_src: $DBI::errstr\n";
&db_insert($ins_stmt, $dbh);
$dbh->commit;
$dbh->disconnect;
exit(0);

sub db_insert {
      my($ins, $dbh) = @_;
      my($sth) = undef;
      my @rows = (
        "801,Alex,Alt,5 Blue Ave,New York,NY,USA,10012,5185553434,BXM",
        "802,Zach,Zed,82 Fair St,New York,NY,USA,10033,5185552234,Zap"
       );
      $sth = $dbh->prepare($ins);
      my $row = undef;
      foreach $row ( @rows ) {
        my @values = split(/,/, $row);
        $sth->execute(@values);
      }
}
__END__
```

SQL Anywhere Python 数据库支持

目录

sqlanydb 简介

通过 sqlanydb 接口，可从使用 Python 编写的脚本访问 SQL Anywhere 数据库。sqlanydb 模块及其扩展模块执行由 Marc-Andr¨| Lemburg 编写的 Python 数据库 API 规范 v2.0。安装完 sqlanydb 模块后，即可使用 Python 访问和更改 SQL Anywhere 数据库中的信息。

有关 Python 数据库 API 规范 v2.0 的信息，请访问 Python Database API specification v2.0。

将 Python 用于多线程时，sqlanydb 模块是线程安全的。

要求

sqlanydb 模块要求具有以下组件：

- Python 2.4 或更高版本（建议使用 2.5 或更高版本）。
- 需要 ctypes 模块。要测试 ctypes 模块是否存在，请打开命令提示窗口并运行 Python。

  在 Python 提示符处，输入以下语句：

  ```
  import ctypes
  ```

  如果看到错误消息，则表明 ctypes 不存在。以下是一个示例。

  ```
  >>> import ctypes
  Traceback (most recent call last):
    File "<stdin>", line 1, in ?
  ImportError: No module named ctypes
  ```

  如果 Python 安装目录中没有 ctypes，则安装 ctypes。可以访问 http://sourceforge.net/project/showfiles.php?group_id=71702，在 SourceForge.net 文件部分中找到安装程序，也可以使用 Peak EasyInstall 自动下载并安装程序。要下载 Peak EasyInstall，请访问 http://peak.telecommunity.com/DevCenter/EasyInstall。

以下几节可帮助您安装 Python 和 sqlanydb 模块。

在 Windows 上安装 sqlanydb

♦ 准备计算机

1. 安装 Python 2.4 或更高版本。
2. 如果 ctypes 模块丢失，则安装 ctypes 模块。

♦ 在 Windows 上安装 sqlanydb 模块

1. 在系统命令提示符处，转到 SQL Anywhere 安装目录的 *SDK\Python* 子目录。
2. 运行以下命令安装 sqlanydb。

   ```
   python setup.py install
   ```

3. 要测试 sqlanydb，请将示例数据库文件复制到 *SDK\Python* 目录中，然后进行测试。

   ```
   copy "samples-dir\demo.db" .
   dbeng11 demo
   python Scripts\test.py
   ```

   如果测试没有运行，请确保路径中包含 SQL Anywhere 安装目录的 *bin32* 或 *bin64* 子目录。

现在即可使用 sqlanydb 模块。

在 Unix 和 Mac OS X 上安装 sqlanydb

以下步骤介绍了如何在支持的 Unix 平台（包括 Mac OS X）中安装 sqlanydb 模块。

♦ 准备计算机

1. 安装 Python 2.4 或更高版本。
2. 如果 ctypes 模块丢失，则安装 ctypes 模块。

♦ 在 Unix 和 Mac OS X 上安装 sqlanydb 模块

1. 确保设置了适用于 SQL Anywhere 的环境。

   根据您使用的 shell，输入相应的命令以从 SQL Anywhere 安装目录执行 SQL Anywhere 配置脚本：

在此 shell 中……	... 使用此命令
sh、ksh 或 bash	*. bin/sa_config.sh*
csh 或 tcsh	source *bin/sa_config.csh*

2. 在 shell 提示符处，转到 SQL Anywhere 安装目录的 *sdk/python* 子目录。
3. 输入以下命令安装 sqlanydb。

   ```
   python setup.py install
   ```

4. 要测试 sqlanydb，请将示例数据库文件复制到 *sdk/python* 目录中，然后进行测试。

   ```
   cp samples-dir/demo.db .
   dbeng11 demo
   python scripts/test.py
   ```

   如果测试没有运行，请确保路径中包含 SQL Anywhere 安装目录的 *bin32* 或 *bin64* 子目录。

现在即可使用 sqlanydb 模块。

编写使用 sqlanydb 的 Python 脚本

本节概述如何编写使用 sqlanydb 接口的 Python 脚本。从 Python Database API specification v2.0 可在线获取有关 API 的完整文档。

装载 sqlanydb 模块

要在 Python 脚本中使用 sqlanydb 模块，必须首先通过在文件顶部添加以下行来装载该模块。

```
import sqlanydb
```

打开和关闭连接

通常，打开一个到数据库的连接，然后通过该连接运行一系列 SQL 语句来执行所有需要的操作。要打开连接，请使用 connect 方法。返回值是一个到数据库连接的句柄，使用该句柄可以在连接上执行后继操作。

connect 方法的参数指定为一系列以逗号分隔的 [关键字=值] 对。

```
sqlanydb.connect( keyword=value, ...)
```

下面是一些常用连接参数：

- **DataSourceName="*dsn*"** 此连接参数的简写形式是 **DSN**="*dsn*"。例如，DataSourceName="SQL Anywhere 11 Demo"。
- **UserID="*user-id*"** 此连接参数的简写形式是 **UID**="*user-id*"。例如，UserID="DBA"。
- **Password="*passwd*"** 此连接参数的简写形式是 **PWD**="*passwd*"。例如，Password="sql"。
- **DatabaseFile="*db-file*"** 此连接参数的简写形式是 **DBF**="*db-file*"。例如，DatabaseFile="demo.db"

有关连接参数的完整列表，请参见“连接参数和网络协议选项”《SQL Anywhere 服务器 - 数据库管理》。

以下代码示例打开然后关闭到 SQL Anywhere 示例数据库的连接。必须先启动数据库服务器和示例数据库然后才能运行此脚本。

```
import sqlanydb

# Create a connection object
con = sqlanydb.connect( userid="DBA",
                        password="sql" )
# Close the connection
con.close()
```

为避免手工启动数据库服务器，可以用经过配置的数据源启动服务器，如下例所示。

```
import sqlanydb

# Create a connection object
```

```
con = sqlanydb.connect( DSN="SQL Anywhere 11 Demo" )

# Close the connection
con.close()
```

选择数据

获得了打开的连接的句柄后，您可以访问和修改存储在数据库中的数据。可能最简单的操作是检索某些行并输出它们。

cursor 方法用于在打开的连接上创建游标。execute 方法用于创建结果集。fetchall 方法用于获取此结果集中的行。

```
import sqlanydb

# Create a connection object, then use it to create a cursor
con = sqlanydb.connect( userid="DBA",
                        password="sql" )
cursor = con.cursor()

# Execute a SQL string
sql = "SELECT * FROM Employees"
cursor.execute(sql)

# Get a cursor description which contains column names
desc = cursor.description
print len(desc)

# Fetch all results from the cursor into a sequence,
# display the values as column name=value pairs,
# and then close the connection
rowset = cursor.fetchall()
for row in rowset:
    for col in range(len(desc)):
        print "%s=%s" % (desc[col][0], row[col] )
    print
cursor.close()
con.close()
```

插入行

在表中插入行最简单的方法是使用非参数化 INSERT 语句，这意味着值作为 SQL 语句的一部分来指定。会为每个新行构建和执行一条新语句。在前面的示例中，需要游标才能执行 SQL 语句。

以下示例程序在示例数据库中插入两名新客户。断开连接前，它将事务提交到数据库。

```
import sqlanydb

# Create a connection object, then use it to create a cursor
con = sqlanydb.connect( userid="DBA", pwd="sql" )
cursor = con.cursor()
cursor.execute("DELETE FROM Customers WHERE ID > 800")

rows = ((801,'Alex','Alt','5 Blue Ave','New York','NY',
        'USA','10012','5185553434','BXM'),
        (802,'Zach','Zed','82 Fair St','New York','NY',
        'USA','10033','5185552234','Zap'))
```



```
# Set up a SQL INSERT
parms = ("'%s'," * len(rows[0]))[:-1]
sql = "INSERT INTO Customers VALUES (%s)" % (parms)
print sql % rows[0]
cursor.execute(sql % rows[0])
print sql % rows[1]
cursor.execute(sql % rows[1])
cursor.close()
con.commit()
con.close()
```

另一种方法是使用参数化的 INSERT 语句，这意味着问号用作值的占位符。executemany 方法用于为每个行集成员执行 INSERT 语句。新行的值作为单个参数提供给 executemany 方法。

```
import sqlanydb

# Create a connection object, then use it to create a cursor
con = sqlanydb.connect( userid="DBA", pwd="sql" )
cursor = con.cursor()
cursor.execute("DELETE FROM Customers WHERE ID > 800")

rows = ((801,'Alex','Alt','5 Blue Ave','New York','NY',
        'USA','10012','5185553434','BXM'),
        (802,'Zach','Zed','82 Fair St','New York','NY',
        'USA','10033','5185552234','Zap'))

# Set up a parameterized SQL INSERT
parms = ("?," * len(rows[0]))[:-1]
sql = "INSERT INTO Customers VALUES (%s)" % (parms)
print sql
cursor.executemany(sql, rows)
cursor.close()
con.commit()
con.close()
```

虽然两个示例似乎都是将行数据插入表中的较为合适的方法，但后一种方法更佳，原因有两点。如果数据值是通过提示来输入而获取的，在注入包含 SQL 语句的游荡数据时，第一个示例容易受到影响。第一个示例需要为每个要插入到表中的行调用 execute 方法。而第二个示例只调用一次 executemany 方法，就可将所有行插入到表中。

SQL Anywhere PHP API

目录

SQL Anywhere PHP 模块简介

PHP 是 **PHP: Hypertext Preprocessor**（超文本预处理器）的缩写，是一种开放源代码的脚本编写语言。虽然可以将其用作通用目的脚本编写语言，但是其设计目的是成为一种用来编写可嵌入到 HTML 文档中的脚本的方便语言。与客户端经常执行的用 JavaScript 编写的脚本不同，PHP 脚本由 Web 服务器处理，然后将生成的 HTML 输出发送到客户端。PHP 的语法是由其它流行语言（如 Java 和 Perl）的语法派生出来的。

为了使其成为一种可以开发动态 Web 页的方便的语言，PHP 提供了从许多常见数据库（如 SQL Anywhere）检索信息的功能。SQL Anywhere 中有两个可以从 PHP 访问 SQL Anywhere 数据库的模块。您可以使用这些模块和 PHP 语言来编写独立脚本，并创建依赖于存储在 SQL Anywhere 数据库中的信息的动态 Web 页。

为 Windows、Linux 和 Solaris 提供 PHP 模块的预建版本，并将其安装在 SQL Anywhere 安装目录的操作系统特定的二进制子目录中。SQLAnywhere PHP 模块的源代码安装在 SQL Anywhere 安装目录的 *sdk\php* 子目录中。

还可以在 SQL Anywhere PHP 模块中在线找到其它信息。

要求

还必须安装以下组件才能使用 SQL Anywhere PHP 模块：

- 您的平台的 PHP 5 二进制文件（可以从 http://www.php.net 下载）。SQL Anywhere 为 PHP 版本 5.1.1 到 5.2.6 提供预建的 PHP 模块。在写本书时，PHP 版本 5.2.6 是最新的稳定版本。对于 Windows 平台，PHP 的线程安全版必须与 SQL Anywhere PHP 模块一起使用。
- Web 服务器（如果想在 Web 服务器中运行 PHP 脚本）。请注意，SQL Anywhere 可以作为 Web 服务器使用。

  还可以使用其它 Web 服务器，例如 Apache HTTP 服务器。SQL Anywhere 可以与 Web 服务器在同一台计算机上运行，也可以不在同一台计算机上运行。
- 对于 Windows，使用 SQL Anywhere 客户端软件 *dblib11.dll* 和 *dbcapi.dll*。
- 对于 Linux/Unix，使用 SQL Anywhere 客户端软件 *libdblib11.so* 和 *libdbcapi.so*。
- 对于 Mac OS X，使用 SQL Anywhere 客户端软件 *libdblib11.dylib* 和 *libdbapi.dylib*。

有关安装 PHP 和 Apache HTTP 服务器的其它信息，请参见 Serving Content from SQL Anywhere Databases Using Apache and PHP。

以下几节可帮助您安装 SQL Anywhere PHP 模块。

安装和配置 SQL Anywhere PHP

下面几节介绍如何安装和配置 SQL Anywhere PHP 模块。

选择要使用的 PHP 模块

在 Windows 上，SQL Anywhere 包括 PHP 版本 5.1.1 到 5.2.6 的线程安全模块。PHP 的线程安全版必须与 SQL Anywhere PHP 模块一起使用。受支持的 PHP 版本的模块文件名遵循以下模式：

```
php-5.x.y_sqlanywhere.dll
```

在 Linux 和 Solaris 上，SQL Anywhere 包含 PHP 版本 5.1.1 到 5.2.6 的 64 位和 32 位版本模块。而且，它还包含线程模块和非线程模块。如果您正在使用 PHP 的 CGI 版本或 Apache 1.x，则请使用非线程模块。如果您使用 Apache 2.x，请使用线程模块。受支持的 PHP 版本的模块文件名遵循以下模式：

```
php-5.x.y_sqlanywhere[_r].so
```

"5.x.y" 代表 PHP 版本（如 5.2.6）。对于 Linux 和 Solaris，PHP 模块的线程版本已将 *_r* 附加到文件名。Windows 版本被作为动态链接库来实现，而 Linux/Solaris 版本则被作为共享对象来实现。

在 Windows 上安装 PHP 模块

要在 Windows 上使用 SQL Anywhere PHP 模块，您必须从 SQL Anywhere 安装目录中复制 DLL，并将其添加到 PHP 安装目录中。或者，您可以将一个条目添加到 PHP 初始化文件中以装载模块，从而不需要在每个脚本中手工装载该模块。

♦ 在 Windows 上安装 PHP 模块

1. 找到 PHP 安装目录中的 *php.ini* 文件，并在文本编辑器中将其打开。找到指定 **extension_dir** 目录位置的行。如果未将 **extension_dir** 设置到任何特定目录，则为获得更好的系统安全，最好将其设置为指向一个单独的目录。
2. 将文件 *php-5.x.y_sqlanywhere.dll* 从 SQL Anywhere 安装目录的 *Bin32* 子目录复制到由 *php.ini* 文件中的 **extension_dir** 条目指定的目录中。

   注意
   字符串 *5.x.y* 是对应于已安装版本的 PHP 版本号。

   如果您的 PHP 版本比 SQL Anywhere 提供的 SQL Anywhere PHP 模块新，请尝试使用所提供的最新模块。注意，5.2.x 版本的 SQL Anywhere PHP 模块不能与 5.3.x 版本的 PHP 一起工作。
3. 将以下行添加到 *php.ini* 文件的动态扩展部分，以自动装载 SQL Anywhere PHP 驱动程序。

   ```
   extension=php-5.x.y_sqlanywhere.dll
   ```

   其中，*5.x.y* 是在上一步中所复制的 SQL Anywhere PHP 模块的版本号。

   保存并关闭 *php.ini*。

如果不自动装载 PHP 驱动程序，也可以在每个需要 PHP 驱动程序的脚本中手工装载。请参见“配置 SQL Anywhere PHP 模块”一节第 716 页。

4. 确保路径中包含 SQL Anywhere 安装目录的 *Bin32* 子目录。SQL Anywhere PHP 扩展 DLL 需要您的路径中包含 *Bin32* 目录。
5. 在命令提示符处运行以下命令以启动 SQL Anywhere 示例数据库。

   ```
   dbeng11 samples-dir\demo.db
   ```

   该命令使用示例数据库启动数据库服务器。
6. 在命令提示符处，转到 SQL Anywhere 安装目录的 *SDK\PHP\Examples* 子目录。请确保将 **php** 可执行目录包含在您的路径中。输入以下命令：

   ```
   php test.php
   ```

   应该会出现类似以下内容的消息。如果无法识别此 PHP 命令，请验证它是否在路径中。

   ```
   Installation successful
   Using php-5.2.6_sqlanywhere.dll
   Connected successfully
   ```

   如果未装载 SQL Anywhere PHP 驱动程序，则可将命令 "php -i" 用于有关 PHP 设置的有帮助的信息。在此命令的输出中搜索 **extension_dir** 和 **sqlanywhere**。
7. 当您完成时，请单击数据库服务器消息窗口中的 [关闭] 来停止 SQL Anywhere 数据库服务器。

有关详细信息，请参见“创建 PHP 测试页”一节第 718 页。

在 Linux/Solaris 上安装 PHP 模块

要在 Linux 或 Solaris 上使用 SQL Anywhere PHP 模块，您必须从 SQL Anywhere 安装目录中复制共享对象，并将其添加到 PHP 安装目录中。或者，可以将一个条目添加到 PHP 初始化文件 *php.ini* 中以装载模块，从而不需要在每个脚本中手工装载该模块。

♦ 在 Linux/Solaris 上安装 PHP 模块

1. 找到 PHP 安装目录中的 *php.ini* 文件，并在文本编辑器中将其打开。找到指定 **extension_dir** 目录位置的行。如果未将 **extension_dir** 设置到任何特定目录，则为获得更好的系统安全，最好将其设置为指向一个单独的目录。
2. 将共享对象从 SQL Anywhere 安装目录的 *lib32* 或 *lib64* 子目录复制到 *php.ini* 文件中的 **extension_dir** 条目指定的目录中。共享对象的选择将取决于已安装的 PHP 的版本以及该版本是 32 位还是 64 位。

   注意
   如果您的 PHP 版本比 SQL Anywhere 提供的共享对象新，请尝试使用所提供的最新共享对象。注意，5.2.x 版本的 SQL Anywhere PHP 模块不能与 5.3.x 版本的 PHP 一起工作。

   有关使用哪个共享对象版本的信息，请参见“选择要使用的 PHP 模块”一节第 713 页。

3. 将以下行添加到 *php.ini* 文件的动态扩展部分，以自动装载 SQL Anywhere PHP 驱动程序。该条目必须标识您复制的共享对象

   ```
   extension=php-5.x.y_sqlanywhere.so
   ```

   或线程安全共享对象，

   ```
   extension=php-5.x.y_sqlanywhere_r.so
   ```

   其中 *5.x.y* 是在上一步中所复制的 PHP 共享对象的版本号。

   保存并关闭 *php.ini*。

   如果不自动装载 PHP 驱动程序，也可以在每个需要 PHP 驱动程序的脚本中手工装载。请参见“配置 SQL Anywhere PHP 模块”一节第 716 页。

4. 尝试使用 PHP 模块前，请校验是否为 SQL Anywhere 设置了合适的 PHP 执行环境。根据您正在使用的 shell，必须为您的 Web 服务器的环境编辑配置脚本并添加相应的命令以从 SQL Anywhere 安装目录执行 SQL Anywhere 配置脚本：

在此 shell 中……	**... 使用此命令**
sh、ksh 或 bash	. /bin32/sa_config.sh
csh 或 tcsh	source /bin32/sa_config.csh

   SQL Anywhere PHP 扩展 DLL 的 32 位版本需要您的路径中包含 *bin32* 目录。SQL Anywhere PHP 扩展 DLL 的 64 位版本需要您的路径中包含 *bin64* 目录。

   对于不同的 Web 服务器和 Linux 分布，应在其中插入该命令行的配置文件是不同的。以下是指示分布上的 Apache 服务器的一些示例：

   - **RedHat/Fedora/CentOS** /etc/sysconfig/httpd
   - **Debian/Ubuntu** /etc/apache2/envvars

   必须在编辑了 Web 服务器的环境配置后重新启动它。

5. 在命令提示符处输入以下命令以启动 SQL Anywhere 示例数据库：

   ```
   dbeng11 samples-dir/demo.db
   ```

6. 在命令提示符处，更改为 SQL Anywhere 安装目录的 *sdk/php/examples* 子目录。输入以下命令：

   ```
   php test.php
   ```

   应该会出现类似以下内容的消息。如果无法识别此 php 命令，请验证它是否在路径中。

   ```
   Installation successful
   Using php-5.2.6_sqlanywhere.so
   Connected successfully
   ```

7. 当您完成时，停止数据库服务器。

有关详细信息，请参见“创建 PHP 测试页”一节第 718 页。

在 UNIX 和 Mac OS X 上构建 PHP 模块

要在 Unix 或 Mac OS X 的其它版本上使用 SQL Anywhere PHP 模块，必须利用在 SQL Anywhere 安装目录的 *sdk/php* 子目录中所安装的源代码来构建 PHP 模块。请参见“在 UNIX 和 Mac OS X 上构建 SQL Anywhere PHP 模块”一节第 772 页。

配置 SQL Anywhere PHP 模块

可以通过设置 PHP 初始化文件 *php.ini* 中的值来控制 SQL Anywhere PHP 驱动程序的行为。支持以下条目：

- **extension** 导致 PHP 在每次启动时自动装载 SQL Anywhere PHP 模块。您可以选择是否将此条目添加到 PHP 初始化文件中，但是如果您不添加该条目，则必须在编写的每个脚本的开头添加几行代码以确保装载此模块。以下条目用于 Windows 平台。

  ```
  extension=php-5.x.y_sqlanywhere.dll
  ```

  在 Linux 平台上，使用以下条目之一。第二个条目是线程安全。

  ```
  extension=php-5.x.y_sqlanywhere.so

  extension=php-5.x.y_sqlanywhere_r.so
  ```

  在这些条目中，*5.x.y* 标识 PHP 版本。

  如果启动 PHP 时不总是自动装载该 SQL Anywhere 模块，则您必须在编写的每个脚本的开头添加以下几行代码。该代码可确保装载 SQL Anywhere PHP 模块。

  ```
  # Ensure that the SQL Anywhere PHP module is loaded
  if( !extension_loaded('sqlanywhere') ) {
      # Find out which version of PHP is running
      $version = phpversion();
      $module_name = 'php-'.$version.'_sqlanywhere';
      if( strtoupper(substr(PHP_OS, 0, 3) == 'WIN' )) {
          $module_ext = '.dll';
      } else {
          $module_ext = '.so';
      }
      dl( $module_name.$module_ext );
  }
  ```

- **allow_persistent** 设置为 On 时，允许持久性连接。设置为 Off 时，不允许持久性连接。缺省值为 On。

  ```
  sqlanywhere.allow_persistent=On
  ```

- **max_persistent** 设置持久性连接的最大数量。缺省值为 -1，这意味着不加限制。

  ```
  sqlanywhere.max_persistent=-1
  ```

- **max_connections** 设置可以通过 SQL Anywhere PHP 模块同时打开的连接的最大数量。缺省值为 -1，这意味着不加限制。

  ```
  sqlanywhere.max_connections=-1
  ```


- **auto_commit** 指定数据库服务器是否自动执行提交操作。设置为 On 时，会在每个语句执行后立刻执行提交。设置为 Off 时，应根据需要使用 sasql_commit 或 sasql_rollback 函数来手工结束事务。缺省值为 On。

  ```
  sqlanywhere.auto_commit=On
  ```

- **row_counts** 设置为 On 时，返回受操作影响的确切行数；设置为 Off 时，返回估计值。缺省值为 Off。

  ```
  sqlanywhere.row_counts=Off
  ```

- **verbose_errors** 设置为 On 时，返回详细错误和警告消息。否则，必须调用 sasql_error 或 sasql_errorcode 函数以获取进一步的错误信息。缺省值为 On。

  ```
  sqlanywhere.verbose_errors=On
  ```

有关详细信息，请参见“sasql_set_option”一节第 743 页。

在 Web 页中运行 PHP 测试脚本

本节介绍如何编写 PHP 测试脚本来查询示例数据库并显示有关 PHP 的信息。

创建 PHP 测试页

以下说明适用于所有配置。

为了测试 PHP 的设置是否正确，下面的过程介绍如何创建并运行一个调用 phpinfo 的 Web 页。PHP 函数 phpinfo 可生成一个系统设置信息页。输出会显示 PHP 是否正常工作。

有关安装 PHP 的信息，请参见 http://us2.php.net/install。

♦ 创建 PHP 信息测试页

1. 在 Web 内容的根目录中创建一个名为 *info.php* 的文件。

   如果您不能确定要使用哪个目录，请检查 Web 服务器的配置文件。在 Apache 安装目录中，内容目录通常名为 *htdocs*。如果使用的是 Mac OS X，Web 内容的目录名称则可能取决于所用帐户：

   - 如果您是 Mac OS X 系统上的系统管理员，则使用 */Library/WebServer/Documents*。
   - 如果您是 Mac OS X 用户，则将文件置于 */Users/your-user-name/Sites/* 中。

2. 将以下代码插入到此文件中：

   ```
   <?php phpinfo(); ?>
   ```

   或者，一旦正确安装并配置了 PHP，也可以通过在命令提示符处执行以下命令来创建一个测试 Web 页。

   ```
   php -i > info.html
   ```

   这可确认安装的 PHP 和 Web 服务器是否能够一起正常工作。

3. 在命令提示符处运行以下命令以启动 SQL Anywhere 示例数据库（如果您尚未这样做）：

   ```
   dbeng11 samples-dir\demo.db
   ```

4. 测试 PHP 和 Web 服务器是否能与 SQL Anywhere 一起正常工作：

   a. 将 *connect.php* 文件从 PHP 示例目录复制到 Web 内容的根目录中。

   b. 从 Web 浏览器访问 *connect.php* 页。

      此时应出现消息 [`Connected successfully`]。

♦ 创建使用 SQL Anywhere PHP 模块的查询页

1. 在 Web 内容的根目录下，创建一个名为 *sa_test.php* 且包含以下 PHP 代码的文件。
2. 将以下 PHP 代码插入到此文件中：

   ```
   <?php
     $conn = sasql_connect( "UID=DBA;PWD=sql" );
   ```



```
  $result = sasql_query( $conn, "SELECT * FROM Employees" );
  sasql_result_all( $result );
  sasql_free_result( $result );
  sasql_disconnect( $conn );
?>
```

访问测试 Web 页

以下过程介绍安装并配置 PHP 和 SQL Anywhere PHP 模块后如何通过 Web 浏览器查看测试页。

♦ 查看 Web 页

1. 重新启动 Web 服务器。

   例如，要启动 Apache Web 服务器，可从 Apache 安装目录的 *bin* 子目录运行以下命令：

   ```
   apachectl start
   ```

2. 在 Linux 或 Mac OS X 上，使用提供的脚本之一来设置 SQL Anywhere 环境变量。

   根据您使用的 shell，输入相应的命令从 SQL Anywhere 安装目录找到 SQL Anywhere 配置脚本：

在此 shell 中……	**... 使用此命令**
sh、ksh 或 bash	`. /bin32/sa_config.sh`
csh 或 tcsh	`source /bin32/sa_config.csh`

3. 启动 SQL Anywhere 数据库服务器。

   例如，要访问上述测试 Web 页，请使用以下命令启动 SQL Anywhere 示例数据库。

   ```
   dbeng11 samples-dir\demo.db
   ```

4. 要通过与服务器运行在同一计算机上的浏览器访问测试页，请输入以下 URL：

对于此测试页……	**... 使用的 URL**
info.php	`http://localhost/info.php`
sa_test.php	`http://localhost/sa_test.php`

   如果每项配置均正确，则 sa_test 页会显示 Employees 表的内容。

编写 PHP 脚本

本节介绍如何编写使用 SQL Anywhere PHP 模块访问 SQL Anywhere 数据库的 PHP 脚本。

上述示例及其它示例的源代码位于 SQL Anywhere 安装目录的 *SDK\PHP\Examples* 子目录中。

连接到数据库

要建立一个到数据库的连接，将一个标准的 SQL Anywhere 连接字符串作为 sasql_connect 函数的参数传递给数据库服务器。<?php 和 ?> 标记指示 Web 服务器应让 PHP 执行这两个标记之间的代码，并用 PHP 输出替换这些代码。

本示例的源代码包含在 SQL Anywhere 安装目录中一个名为 *connect.php* 的文件中。

```
<?php
  # Connect using the default user ID and password
  $conn = sasql_connect( "UID=DBA;PWD=sql" );
  if( ! $conn ) {
      echo "Connection failed\n";
  } else {
      echo "Connected successfully\n";
      sasql_close( $conn );
  }?>
```

第一个代码块校验是否装载了 PHP 模块。如果已将该行添加到 PHP 初始化文件中以自动装载该模块，则此处就不再需要这个代码块了。如果未将 PHP 配置为在启动时自动装载 SQL Anywhere PHP 模块，则必须将此代码添加到其它示例脚本中。

第二个代码块尝试建立一个连接。要使此代码成功执行，必须正在运行 SQL Anywhere 示例数据库。

从数据库检索数据

PHP 脚本在 Web 页中的一个用途是检索并显示数据库中包含的信息。以下示例介绍了一些非常有用的技术。

简单选择查询

以下 PHP 代码介绍了一种在 Web 页中包含 SELECT 语句的结果集的便捷方法。此示例用来连接到 SQL Anywhere 示例数据库并返回一个客户列表。

此代码可以嵌入到 Web 页中，条件是 Web 服务器已配置为可以执行 PHP 脚本。

本示例的源代码包含在 SQL Anywhere 安装目录中一个名为 *query.php* 的文件中。

```
<?php
  # Connect using the default user ID and password
  $conn = sasql_connect( "UID=DBA;PWD=sql" );
  if( ! $conn ) {
      echo "sasql_connect failed\n";
  } else {
      echo "Connected successfully\n";
      # Execute a SELECT statement
      $result = sasql_query( $conn, "SELECT * FROM Customers" );
```



```
        if( ! $result ) {
            echo "sasql_query failed!";
        } else {
            echo "query completed successfully\n";
            # Generate HTML from the result set
            sasql_result_all( $result );
            sasql_free_result( $result );
        }
        sasql_close( $conn );
    }
?>
```

sasql_result_all 函数可读取结果集中的所有行，并生成一个 HTML 输出表来显示这些行。sasql_free_result 函数释放用来存储结果集的资源。

按列名读取

某些情况下，您可能不想显示结果集中的所有数据，或是希望以其它方式显示这些数据。下面的示例说明如何对结果集的输出格式进行更好地控制。PHP 允许您以选择的任何方式显示任意数量的信息。

本示例的源代码包含在 SQL Anywhere 安装目录中一个名为 *fetch.php* 的文件中。

```
<?php
  # Connect using the default user ID and password
  $conn = sasql_connect( "UID=DBA;PWD=sql" );
  if( ! $conn ) {
      die ("Connection failed");
  } else {
      # Connected successfully.
  }
  # Execute a SELECT statement
  $result = sasql_query( $conn, "SELECT * FROM Customers" );
  if( ! $result ) {
      echo "sasql_query failed!";
      return 0;
  } else {
      echo "query completed successfully\n";
  }
  # Retrieve meta information about the results
  $num_cols = sasql_num_fields( $result );
  $num_rows = sasql_num_rows( $result );
  echo "Num of rows = $num_rows\n";
  echo "Num of cols = $num_cols\n";
  while( ($field = sasql_fetch_field( $result )) ) {
      echo "Field # : $field->id \n";
      echo "\tname   : $field->name \n";
      echo "\tlength : $field->length \n";
      echo "\ttype   : $field->type \n";
  }
  # Fetch all the rows
  $curr_row = 0;
  while( ($row = sasql_fetch_row( $result )) ) {
      $curr_row++;
      $curr_col = 0;
      while( $curr_col < $num_cols ) {
          echo "$row[$curr_col]\t|";
          $curr_col++;
      }
      echo "\n";
  }
  # Clean up.
```

```
  sasql_free_result( $result );
  sasql_disconnect( $conn );
?>
```

sasql_fetch_array 函数从表中返回单行。可以按列名和列索引检索这些数据。

asql_fetch_assoc 函数从表中返回单行作为一个联合数组。可以使用列名作为索引来检索这些数据。以下是一个示例。

```
<?php
  # Connect using the default user ID and password
  $conn = sasql_connect("UID=DBA;PWD=sql");

  /* check connection */
  if( sasql_errorcode() ) {
      printf("Connect failed: %s\n", sasql_error());
      exit();
  }

  $query = "SELECT Surname, Phone FROM Employees ORDER by EmployeeID";

  if( $result = sasql_query($conn, $query) ) {

      /* fetch associative array */
      while( $row = sasql_fetch_assoc($result) ) {
          printf ("%s (%s)\n", $row["Surname"], $row["Phone"]);
      }

      /* free result set */
      sasql_free_result($result);
  }

  /* close connection */
  sasql_close($conn);
?>
```

在 PHP 接口中还提供了其它两个类似的方法：sasql_fetch_row 返回一个仅能按列索引加以搜索的行，而 sasql_fetch_object 则返回一个仅能按列名搜索的行。

有关 sasql_fetch_object 函数的示例，请参见 *fetch_object.php* 示例脚本。

嵌套结果集

向数据库发送一条 SELECT 语句时，会返回一个结果集。sasql_fetch_row 和 sasql_fetch_array 函数从结果集的各行检索数据，然后将每行作为可以进一步查询的列数组返回。

本示例的源代码包含在 SQL Anywhere 安装目录中一个名为 *nested.php* 的文件中。

```
<?php
  # Connect using the default user ID and password
  $conn = sasql_connect( "UID=DBA;PWD=sql" );
  if( ! $conn ) {
      die ("Connection failed");
  } else {
      # Connected successfully.
  }
  # Retrieve the data and output HTML
  echo "<BR>\n";
  $query1 = "SELECT table_id, table_name FROM SYSTAB";
  $result = sasql_query( $conn, $query1 );
  if( $result ) {
    $num_rows = sasql_num_rows( $result );
```



```
    echo "Returned : $num_rows <BR>\n";
    $I = 1;
    while( ($row = sasql_fetch_array( $result ))) {
      echo "$I:  table_id:$row[table_id]" .
           " --- table_name:$row[table_name] <br>\n";
      $query2 = "SELECT table_id, column_name " .
      "FROM SYSTABCOL" .
      "WHERE table_id = '$row[table_id]'";
      echo " $query2 <br>\n";
      echo "  Columns: ";
      $result2 = sasql_query( $conn, $query2 );
      if( $result2 ) {
     while(($detailed = sasql_fetch_array($result2))) {
         echo " $detailed[column_name]";
     }
     sasql_free_result( $result2 );
      } else {
     echo "******FAILED********";
      }
      echo "<br>\n";
      $I++;
    }
  }
  echo "<BR>\n";
  sasql_disconnect( $conn );
?>
```

在上例中，SQL 语句从 SYSTAB 中为每个表选择表 ID 和名称。sasql_query function 函数返回一个行数组。该脚本使用 sasql_fetch_array 函数迭代通过各行，以从数组检索行。一个内部迭代遍历每行中的列并显示其值。

Web 表单

PHP 可以从 Web 表单获取用户输入，将用户输入作为 SQL 查询传递给数据库服务器，并显示返回的结果。以下示例介绍了一个简单的 Web 表单，该表单使用户能够使用 SQL 语句查询示例数据库，并将结果显示在一个 HTML 表中。

本示例的源代码包含在 SQL Anywhere 安装目录中一个名为 *webisql.php* 的文件中。

```
<?php
  echo "<HTML>\n";
  $qname = $_POST["qname"];
  $qname = str_replace( "\\", "", $qname );
  echo "<form method=post action=webisql.php>\n";
  echo "<br>Query: <input type=text Size=80 name=qname value=\"$qname\">\n";
  echo "<input type=submit>\n";
  echo "</form>\n";
  echo "<HR><br>\n";
  if( ! $qname ) {
      echo "No Current Query\n";
      return;
  }
  # Connect to the database
  $con_str = "UID=DBA;PWD=sql;ENG=demo;LINKS=tcpip";
  $conn = sasql_connect( $con_str );
  if( ! $conn ) {
      echo "sasql_connect failed\n";
      echo "</html>\n";
      return 0;
  }
```

```
  $qname = str_replace( "\\", "", $qname );
  $result = sasql_query( $conn, $qname );
  if( ! $result ) {
        echo "sasql_query failed!";
  } else {
        // echo "query completed successfully\n";
        sasql_result_all( $result, "border=1" );
      sasql_free_result( $result );
  }
  sasql_disconnect( $conn );
  echo "</html>\n";
?>
```

通过基于用户输入的值的公式化自定义 SQL 查询，可以扩展这一设计，从而处理复杂的 Web 表单。

处理 BLOB

SQL Anywhere 数据库可以将任何类型的数据作为一个二进制大对象 (BLOB) 存储。如果 Web 浏览器可以读取该类型的数据，则 PHP 脚本可以从数据库方便地检索数据，并将其显示在动态生成的页上。

BLOB 字段通常用于存储非文本数据（如 GIF 或 JPG 格式的图像）。许多类型的数据无需使用第三方软件或通过数据类型转换，即可传递到 Web 浏览器。以下示例说明了将一个图像添加到数据库中，然后对其进行再次检索以在 Web 浏览器中显示的过程。

此示例类似于 SQL Anywhere 安装目录中的文件 *image_insert.php* 和 *image_retrieve.php* 中的示例代码。这些示例也说明了如何使用 BLOB 列存储图像。

```
<?php
  $conn = sasql_connect( "UID=DBA;PWD=sql" )
        or die("Can not connect to database");
  $create_table = "CREATE TABLE images (ID INTEGER PRIMARY KEY, img IMAGE)";
  sasql_query( $conn, $create_table);
  $insert = "INSERT INTO images VALUES (99,
xp_read_file('ianywhere_logo.gif'))";
  sasql_query( $conn, $insert );
  $query = "SELECT img FROM images WHERE ID = 99";
  $result = sasql_query($conn, $query);
  $data = sasql_fetch_row($result);
  $img = $data[0];
  header("Content-type: image/gif");
  echo $img;
  sasql_disconnect($conn);
?>
```

为了能够将二进制数据从数据库直接发送到 Web 浏览器，脚本必须使用标头函数设置数据的 MIME 类型。在此示例中，系统会告诉浏览器该数据是一个 GIF 图像，以便浏览器正确显示该数据。

SQL Anywhere PHP API 参考

PHP API 支持以下函数：

连接

- “sasql_close”一节第 727 页
- “sasql_connect”一节第 727 页
- “sasql_disconnect”一节第 729 页
- “sasql_error”一节第 729 页
- “sasql_errorcode”一节第 730 页
- “sasql_insert_id”一节第 735 页
- “sasql_message”一节第 736 页
- “sasql_pconnect”一节第 738 页
- “sasql_set_option”一节第 743 页

查询

- “sasql_affected_rows”一节第 726 页
- “sasql_next_result”一节第 737 页
- “sasql_query”一节第 739 页
- “sasql_real_query”一节第 741 页
- “sasql_store_result”一节第 753 页
- “sasql_use_result”一节第 754 页

结果集

- “sasql_data_seek”一节第 728 页
- “sasql_fetch_array”一节第 731 页
- “sasql_fetch_assoc”一节第 731 页
- “sasql_fetch_field”一节第 732 页
- “sasql_fetch_object”一节第 733 页
- “sasql_fetch_row”一节第 733 页
- “sasql_field_count”一节第 734 页
- “sasql_free_result”一节第 735 页
- “sasql_num_rows”一节第 738 页
- “sasql_result_all”一节第 741 页

事务

- “sasql_commit”一节第 727 页
- “sasql_rollback”一节第 742 页

语句

杂类

sasql_affected_rows

原型

int **sasql_affected_rows(** sasql_conn *$conn* **)**

说明

返回受上一 SQL 语句影响的行数。此函数通常用于 INSERT、UPDATE 或 DELETE 语句。对于 SELECT 语句，使用 sasql_num_rows 函数。

参数

$conn 由连接函数返回的连接资源。

返回值

受影响的行数。

相关函数

sasql_commit

原型

bool **sasql_commit(** sasql_conn *$conn* **)**

说明

结束 SQL Anywhere 数据库上的一个事务，并使该事务期间所做的所有更改永久生效。仅当 auto_commit 选项为 Off 时才有效。

参数

$conn　由连接函数返回的连接资源。

返回值

TRUE（成功时）或 FALSE（失败时）。

相关函数

- “sasql_rollback”一节第 742 页
- “sasql_set_option”一节第 743 页
- “sasql_pconnect”一节第 738 页
- “sasql_disconnect”一节第 729 页

sasql_close

原型

bool **sasql_close(** sasql_conn *$conn* **)**

说明

关闭先前打开的数据库连接。

参数

$conn　由连接函数返回的连接资源。

返回值

TRUE（成功时）或 FALSE（失败时）。

sasql_connect

原型

sasql_conn **sasql_connect(** string *$con_str* **)**

说明

建立与 SQL Anywhere 数据库的连接。

参数

$con_str 一个由 SQL Anywhere 识别的连接字符串。

返回值

一个正的 SQL Anywhere 连接资源（成功时），或一个错误和 0（失败时）。

相关函数

- “sasql_pconnect”一节第 738 页
- “sasql_disconnect”一节第 729 页

另请参见

- “连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》
- “SQL Anywhere 数据库连接” 《SQL Anywhere 服务器 - 数据库管理》

sasql_data_seek

原型

bool **sasql_data_seek(** sasql_result *$result*, int *row_num* **)**

说明

将游标定位到使用 sasql_query 打开的 *$result* 的第 *row_num* 行上。

参数

$result 由 sasql_query 函数返回的结果资源。

row_num 结果资源中表示游标的新位置的整数。例如，指定 0 将游标移动到结果集的第一行，指定 5 将游标移动到第六行。负数表示相对于结果集结尾的行。例如，-1 将游标移动到结果集的最后一行，-2 将游标移动到倒数第二行。

返回值

TRUE（成功时）或 FALSE（错误时）。

相关函数

- “sasql_fetch_field”一节第 732 页
- “sasql_fetch_array”一节第 731 页
- “sasql_fetch_assoc”一节第 731 页
- “sasql_fetch_row”一节第 733 页
- “sasql_fetch_object”一节第 733 页
- “sasql_query”一节第 739 页

sasql_disconnect

原型

bool **sasql_disconnect(** sasql_conn *$conn* **)**

说明

关闭一个已经用 sasql_connect 打开的连接。

参数

$conn 由连接函数返回的连接资源。

返回值

TRUE（成功时）或 FALSE（错误时）。

相关函数

- “sasql_connect”一节第 727 页
- “sasql_pconnect”一节第 738 页

sasql_error

原型

string **sasql_error(** [sasql_conn *$conn*] **)**

说明

返回最近执行的 SQL Anywhere PHP 函数的错误文本。错误消息按连接进行存储。如果未指定 *$conn*，则 sasql_error 返回没有可用连接的最后一条错误消息。例如，如果您调用 sasql_connect，但连接失败，则可调用不含 *$conn* 参数的 sasql_error 以获取错误消息。如果要获取相应的 SQL Anywhere 错误代码值，则使用 sasql_errorcode 函数。

参数

$conn 由连接函数返回的连接资源。

返回值

描述错误的字符串。有关错误消息的列表，请参见“SQL Anywhere 错误消息” 《错误消息》。

相关函数

- “sasql_errorcode”一节第 730 页
- “sasql_sqlstate”一节第 754 页
- “sasql_set_option”一节第 743 页
- “sasql_stmt_errno”一节第 747 页
- “sasql_stmt_error”一节第 747 页

sasql_errorcode

原型

int **sasql_errorcode(** [sasql_conn *$conn*] **)**

说明

返回最近执行的 SQL Anywhere PHP 函数的错误代码。错误代码按连接进行存储。如果未指定 *$conn*，则 sasql_errorcode 返回没有可用连接的最后一个错误代码。例如，如果您调用 sasql_connect，但连接失败，则可调用不含 *$conn* 参数的 sasql_errorcode 以获取错误代码。如果要获取相应的错误消息，请使用 sasql_error 函数。

参数

$conn 由连接函数返回的连接资源。

返回值

一个整数，表示 SQL Anywhere 错误代码。错误代码 0 表示成功。正的错误代码表示成功但有警告。负的错误代码表示失败。有关错误代码的列表，请参见“按 SQLCODE 排序的 SQL Anywhere 错误消息”一节 《错误消息》。

相关函数

- “sasql_connect”一节第 727 页
- “sasql_pconnect”一节第 738 页
- “sasql_error”一节第 729 页
- “sasql_sqlstate”一节第 754 页
- “sasql_set_option”一节第 743 页
- “sasql_stmt_errno”一节第 747 页
- “sasql_stmt_error”一节第 747 页

sasql_escape_string

原型

string **sasql_escape_string(** sasql_conn *$conn*, string *$str* **)**

说明

转义所提供的字符串中的所有特殊字符。要转义的特殊字符是 \r、\n、'、"、;、\ 和 NULL 字符。此函数是 sasql_real_escape_string 的别名。

参数

$conn 由连接函数返回的连接资源。

$string 要转义的字符串。

返回值

已转义字符串。

相关函数

- “sasql_real_escape_string”一节第 740 页
- “sasql_connect”一节第 727 页

sasql_fetch_array

原型

array **sasql_fetch_array(** sasql_result *$result* **[,** int *$result_type* **])**

说明

从结果集读取一行。该行作为一个可以按列名或列索引进行索引的数组返回。

参数

$result 由 sasql_query 函数返回的结果资源。

$result_type 此可选参数是一个常量，它指示应该从当前行数据中生成何种类型的数组。参数的可能值是常量 SASQL_ASSOC、SASQL_NUM 或 SASQL_BOTH。它缺省为 SASQL_BOTH。

通过使用 SASQL_ASSOC 常量，此函数的行为将与 sasql_fetch_assoc 函数相同，而 SASQL_NUM 的行为则与 sasql_fetch_row 函数相同。最后一个选项 SASQL_BOTH 将创建一个具备这两种属性的单一数组。

返回值

一个表示结果集中的一行的数组，或 FALSE（无法获取行时）。

相关函数

- “sasql_data_seek”一节第 728 页
- “sasql_fetch_assoc”一节第 731 页
- “sasql_fetch_field”一节第 732 页
- “sasql_fetch_row”一节第 733 页
- “sasql_fetch_object”一节第 733 页

sasql_fetch_assoc

原型

array **sasql_fetch_assoc(** sasql_result *$result* **)**

说明

从结果集中读取一行作为联合数组。

参数

$result 由 sasql_query 函数返回的结果资源。

返回值

代表从结果集中读取的行的字符串的联合数组，数组中的各个关键字代表结果集中的一个列的名称或 FALSE（如果结果集中不再有行）。

相关函数

- “sasql_data_seek”一节第 728 页
- “sasql_fetch_field”一节第 732 页
- “sasql_fetch_field”一节第 732 页
- “sasql_fetch_row”一节第 733 页
- “sasql_fetch_object”一节第 733 页

sasql_fetch_field

原型

object **sasql_fetch_field(** sasql_result *$result* **[,** int *$field_offset* **])**

说明

返回一个对象，其中包含关于某个特定列的信息。

参数

$result 由 sasql_query 函数返回的结果资源。

$field_offset 一个整数，表示您希望在其中检索信息的列/字段。列的编号以零为基准；要获取第一列，应指定值 0。如果忽略此参数，则返回下一个字段对象。

返回值

一个具有以下属性的对象：

- **id** 包含字段号
- **name** 包含字段名
- **numeric** 指出该字段是否是数值
- **length** 返回字段的本地存储大小。
- **type** 返回字段类型
- **native_type** 返回字段的本地类型。这些值是 DT_FIXCHAR、DT_DECIMAL 或 DT_DATE。请参见“嵌入式 SQL 数据类型”一节第 520 页。
- **precision** 返回字段的数字精度。仅为 native_type 等于 DT_DECIMAL 的字段设置此属性。
- **scale** 返回字段的数字小数位数。仅为 native_type 等于 DT_DECIMAL 的字段设置此属性。

相关函数

- “sasql_data_seek”一节第 728 页
- “sasql_fetch_array”一节第 731 页
- “sasql_fetch_assoc”一节第 731 页
- “sasql_fetch_row”一节第 733 页
- “sasql_fetch_object”一节第 733 页

sasql_fetch_object

原型

object **sasql_fetch_object(** sasql_result *$result* **)**

说明

从结果集中读取一行作为对象。

参数

$result 由 sasql_query 函数返回的结果资源。

返回值

代表从结果集中读取的行的对象，其中的每个属性名称都匹配着一个结果集的列名称或 FALSE（如果结果集中不再有行）。

相关函数

- “sasql_data_seek”一节第 728 页
- “sasql_fetch_field”一节第 732 页
- “sasql_fetch_array”一节第 731 页
- “sasql_fetch_assoc”一节第 731 页
- “sasql_fetch_row”一节第 733 页

sasql_fetch_row

原型

array **sasql_fetch_row(** sasql_result *$result* **)**

说明

从结果集读取一行。该行作为一个仅可按列索引进行索引的数组返回。

参数

$result 由 sasql_query 函数返回的结果资源。

返回值

一个表示结果集中的一行的数组，或 FALSE（无法获取行时）。

相关函数

- “sasql_data_seek”一节第 728 页
- “sasql_fetch_field”一节第 732 页
- “sasql_fetch_array”一节第 731 页
- “sasql_fetch_assoc”一节第 731 页
- “sasql_fetch_object”一节第 733 页

sasql_field_count

原型

int **sasql_field_count(** sasql_conn *$conn* **)**

说明

返回最后一个结果所包含的列（字段）数。

参数

$conn 由连接函数返回的连接资源。

返回值

一个正的列数，或 FALSE（如果 *$conn* 无效）。

sasql_field_seek

原型

bool **sasql_field_seek(** sasql_result *$result*, int *$field_offset* **)**

说明

将字段游标设置为给定的偏移。下次调用 sasql_fetch_field 将检索与该偏移相关联的列的字段定义。

参数

$result 由 sasql_query 函数返回的结果资源。

$field_offset 一个整数，表示您希望在其中检索信息的列/字段。列的编号以零为基准；要获取第一列，应指定值 0。如果忽略此参数，则返回下一个字段对象。

返回值

TRUE（成功时）或 FALSE（错误时）。

sasql_free_result

原型

bool **sasql_free_result(** sasql_result *$result* **)**

说明

释放与从 sasql_query 返回的结果资源相关联的数据库资源。

参数

$result 由 sasql_query 函数返回的结果资源。

返回值

TRUE（成功时）或 FALSE（错误时）。

相关函数

- “sasql_query”一节第 739 页

sasql_get_client_info

原型

string **sasql_get_client_info( )**

说明

返回客户端的版本信息。

参数

无

返回值

代表 SQL Anywhere 客户端软件版本的字符串。返回的字符串采用的是 X.Y.Z.W 形式，其中 X 为主版本号，Y 为子版本号，Z 为补丁号，W 为内部版本号（如 10.0.1.3616）。

sasql_insert_id

原型

int **sasql_insert_id(** sasql_conn *$conn* **)**

说明

返回最后插入到 IDENTITY 列或 DEFAULT AUTOINCREMENT 列中的值，如果是最后插入到了不含 IDENTITY 或 DEFAULT AUTOINCREMENT 列的表中，则返回零。

提供 sasql_insert_id 函数是为了与 MySQL 数据库兼容。

参数

$conn 由连接函数返回的连接资源。

返回值

由前一个 INSERT 语句为 AUTOINCREMENT 列生成的 ID，如果最后的插入对 AUTOINCREMENT 列没有影响，则返回零。如果 *$conn* 无效，则该函数会返回 FALSE。

sasql_message

原型

bool **sasql_message(** sasql_conn *$conn*, string *$message* **)**

说明

将一条消息写入服务器控制台。

参数

$conn 由连接函数返回的连接资源。

$message 要写入服务器控制台的消息。

返回值

TRUE（成功时）或 FALSE（失败时）。

sasql_multi_query

原型

bool **sasql_multi_query(** sasql_conn *$conn*, string *$sql_str* **)**

说明

使用提供的连接资源准备并执行由 *$sql_str* 指定的一个或多个 SQL 查询。查询之间使用分号进行分隔。

第一个查询结果可使用 sasql_use_result 或 sasql_store_result 来检索或存储。sasql_field_count 可用于查看查询是否返回结果集。

可使用 sasql_next_result 和 sasql_use_result/sasql_store_result 处理所有的后继查询结果。

参数

$conn 由连接函数返回的连接资源。

$sql_str 由分号分隔的一个或多个 SQL 语句。

有关 SQL 语句的详细信息，请参见“SQL 语句” 《SQL Anywhere 服务器 - SQL 参考》。

返回值

TRUE（成功时）或 FALSE（失败时）。

相关函数

- “sasql_store_result”一节第 753 页
- “sasql_use_result”一节第 754 页
- “sasql_field_count”一节第 734 页

sasql_next_result

原型

bool **sasql_next_result(** sasql_conn *$conn* **)**

说明

从在 *$conn* 上执行的上一个查询中准备下一个结果集。

参数

$conn 由连接函数返回的连接资源。

返回值

如果没有其它结果集要检索，则返回 FALSE。如果有其它结果要检索，则返回 TRUE。调用 sasql_use_result 或 sasql_store_result 来检索下一结果集。

相关函数

- “sasql_use_result”一节第 754 页
- “sasql_store_result”一节第 753 页

sasql_num_fields

原型

int **sasql_num_fields(** sasql_result *$result* **)**

说明

返回 *$result* 中的行所包含的字段数。

参数

$result 由 sasql_query 函数返回的结果资源。

返回值

返回指定结果集中的字段数。

相关函数

- “sasql_num_rows”一节第 738 页
- “sasql_query”一节第 739 页

sasql_num_rows

原型

int **sasql_num_rows(** sasql_result *$result* **)**

说明

返回 *$result* 包含的行数。

参数

$result 由 sasql_query 函数返回的结果资源。

返回值

一个正数（如果行数是确切值），或一个负数（如果行数是估计值）。要获取确切的行数，必须针对数据库永久设置或针对连接临时设置数据库选项 row_counts。请参见“sasql_set_option”一节第 743 页。

相关函数

- “sasql_num_fields”一节第 737 页
- “sasql_query”一节第 739 页

sasql_pconnect

原型

sasql_conn **sasql_pconnect(** string *$con_str* **)**

说明

建立与 SQL Anywhere 数据库的持久性连接。由于 Apache 创建子进程的方式，您可能会在使用 sasql_pconnect 代替 sasql_connect 时发现性能得到了提高。持久性连接可以用与连接池相似的方式提高性能。如果数据库服务器具有的连接的数量有限（例如，个人数据库服务器限制为 10 个并发连接），则使用持久性连接时应当谨慎。持久性连接可以附加到每个子进程上，并且如果 Apache 中拥有的子进程数目超过可用连接的数目，您就会接收到连接错误。

参数

$con_str 一个由 SQL Anywhere 识别的连接字符串。

返回值

一个正的 SQL Anywhere 永久连接资源（成功时），或一个错误和 0（失败时）。

相关函数

- “sasql_connect”一节第 727 页
- “sasql_disconnect”一节第 729 页

另请参见

- “连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》
- “SQL Anywhere 数据库连接” 《SQL Anywhere 服务器 - 数据库管理》

sasql_prepare

原型

sasql_stmt **sasql_prepare(** sasql_conn *$conn*, string *$sql_str* **)**

说明

准备所提供的 SQL 字符串。

参数

$conn 由连接函数返回的连接资源。

$sql_str 要准备的 SQL 语句。此字符串可通过在适当位置嵌入问号来包含参数标记。

有关 SQL 语句的详细信息，请参见“SQL 语句” 《SQL Anywhere 服务器 - SQL 参考》。

返回值

语句对象或 FALSE（失败时）。

相关函数

- “sasql_stmt_param_count”一节第 751 页
- “sasql_stmt_bind_param”一节第 744 页
- “sasql_stmt_bind_param_ex”一节第 745 页
- “sasql_prepare”一节第 739 页
- “sasql_stmt_execute”一节第 748 页
- “sasql_connect”一节第 727 页
- “sasql_pconnect”一节第 738 页

sasql_query

原型

mixed **sasql_query(** sasql_conn *$conn*, string *$sql_str* [, int *$result_mode*] **)**

说明

准备并执行已经使用 sasql_connect 或 sasql_pconnect 打开并以 *$conn* 标识的连接上的 SQL 查询 *$sql_str*。

sasql_query 函数等效于调用两个函数：sasql_real_query 和 sasql_store_result 或 sasql_use_result 中的一个。

参数

$conn 由连接函数返回的连接资源。

$sql_str 一个 SQL Anywhere 支持的 SQL 语句。

$result_mode SASQL_USE_RESULT 或 SASQL_STORE_RESULT（缺省值）。

有关 SQL 语句的详细信息，请参见“SQL 语句” 《SQL Anywhere 服务器 - SQL 参考》。

返回值

失败时返回 FALSE；对于 INSERT、UPDATE、DELETE、CREATE，成功时返回 TRUE；对于 SELECT，返回 sasql_result。

相关函数

- “sasql_real_query”一节第 741 页
- “sasql_free_result”一节第 735 页
- “sasql_fetch_array”一节第 731 页
- “sasql_fetch_field”一节第 732 页
- “sasql_fetch_object”一节第 733 页
- “sasql_fetch_row”一节第 733 页

sasql_real_escape_string

原型

string **sasql_real_escape_string(** sasql_conn *$conn*, string *$str* **)**

说明

转义所提供的字符串中的所有特殊字符。要转义的特殊字符是 \r、\n、'、"、;、\ 和 NULL 字符。

参数

$conn 由连接函数返回的连接资源。

$string 要转义的字符串。

返回值

已转义字符串或 FALSE（错误时）。

相关函数

- “sasql_escape_string”一节第 730 页
- “sasql_connect”一节第 727 页

sasql_real_query

原型

bool **sasql_real_query(** sasql_conn *$conn*, string *$sql_str* **)**

说明

使用提供的连接资源针对数据库执行查询。可使用 sasql_store_result 或 sasql_use_result 检索或存储查询结果。sasql_field_count 函数可用于查看查询是否返回结果集。

请注意，sasql_query 函数等效于调用该函数和 sasql_store_result 或 sasql_use_result 中的一个。

参数

$conn 由连接函数返回的连接资源。

$sql_str 一个 SQL Anywhere 支持的 SQL 语句。

返回值

TRUE（成功时）或 FALSE（失败时）。

相关函数

- “sasql_query”一节第 739 页
- “sasql_store_result”一节第 753 页
- “sasql_use_result”一节第 754 页
- “sasql_field_count”一节第 734 页

sasql_result_all

原型

bool **sasql_result_all(** resource *$result*
[, *$html_table_format_string*
[, *$html_table_header_format_string*
[, *$html_table_row_format_string*
[, *$html_table_cell_format_string*
]]]] **)**

说明

读取 *$result* 的所有结果，并生成一个带可选格式字符串的 HTML 输出表。

参数

$result 由 sasql_query 函数返回的结果资源。

$html_table_format_string 一个适用于 HTML 表的格式字符串。例如，**"Border=1; Cellpadding=5"**。特殊值 none 不创建 HTML 表。在您要自定义列名或脚本时，这会很有用。如果您不想为此参数指定显式值，请对此参数值使用 NULL。

$html_table_header_format_string 一个适用于 HTML 表的列标题的格式字符串。例如，**"bgcolor=#FF9533"**。特殊值 none 不创建 HTML 表。在您要自定义列名或脚本时，这会很有用。如果您不想为此参数指定显式值，请对此参数值使用 NULL。

$html_table_row_format_string 一个适用于 HTML 表中的行的格式字符串。例如，**"onclick='alert('this')'"**。如果您希望交替使用不同的格式，请使用特殊标识 ><。标识的左侧指示对奇数行使用的格式，标识的右侧用于格式化偶数行。如果不在格式字符串中放入此标识，则所有行都使用相同的格式。如果您不想为此参数指定显式值，请对此参数值使用 NULL。

$html_table_cell_format_string 一个适用于 HTML 表行中的单元格的格式字符串。例如，**"onclick='alert('this')'"**。如果您不想为此参数指定显式值，请对此参数值使用 NULL。

返回值

TRUE（成功时）或 FALSE（失败时）。

相关函数

- “sasql_query”一节第 739 页

sasql_rollback

原型

bool **sasql_rollback(** sasql_conn *$conn* **)**

说明

结束 SQL Anywhere 数据库上的一个事务，并放弃该事务期间所做的所有更改。仅当 auto_commit 选项为 Off 时此函数才有效。

参数

$conn 由连接函数返回的连接资源。

返回值

TRUE（成功时）或 FALSE（失败时）。

相关函数

- “sasql_commit”一节第 727 页
- “sasql_set_option”一节第 743 页

sasql_set_option

原型

bool **sasql_set_option(** sasql_conn *$conn*, string *$option*, mixed *$value* **)**

说明

在指定连接上设置指定选项的值。可以为以下选项设置值：

名称	说明	缺省值
auto_commit	此选项设置为 on 时，数据库服务器在每执行一条语句后提交。	on
row_counts	此选项设置为 FALSE 时，sasql_num_rows 函数返回受影响的行数估计值。如果想获得准确计数，请将此选项设置为 TRUE。	FALSE
verbose_errors	此选项设置为 TRUE 时，PHP 驱动程序返回详细的错误信息。此选项设置为 FALSE 时，必须调用 sasql_error 或 sasql_errorcode 函数以获取更详细的错误信息。	TRUE

可以通过在 *php.ini* 文件中加入以下行，更改选项的缺省值。在本示例中，为 auto_commit 选项设置了缺省值。

```
sqlanywhere.auto_commit=0
```

参数

$conn 由连接函数返回的连接资源。

$option 要设置的选项的名称。

$value 新的选项值。

返回值

TRUE（成功时）或 FALSE（失败时）。

相关函数

- “sasql_commit”一节第 727 页
- “sasql_error”一节第 729 页
- “sasql_errorcode”一节第 730 页
- “sasql_num_rows”一节第 738 页
- “sasql_rollback”一节第 742 页

sasql_stmt_affected_rows

原型

int **sasql_stmt_affected_rows(** sasql_stmt *$stmt* **)**

说明

返回受执行语句影响的行数。

参数

$stmt 由 sasql_stmt_execute 执行的语句资源。

返回值

受影响的行数或 FALSE（失败时）。

相关函数

- “sasql_stmt_execute”一节第 748 页

sasql_stmt_bind_param

原型

bool **sasql_stmt_bind_param(** sasql_stmt *$stmt*, string *$types*, mixed **&***$var_1* **[**, mixed **&***$var_2* .. **]** **)**

说明

将 PHP 变量绑定到语句参数。

参数

$stmt 由 sasql_prepare 函数返回的预准备语句资源。

$types 指定相应绑定类型的包含一个或多个字符的字符串。此类型可以是以下各项之一：**s**（对于字符串）、**i**（对于整数）、**d**（对于双精度型）、**b**（对于 blobs）。$types 字符串的长度必须与跟在 $types 参数 ($var_1, $var_2, ...) 后的参数的数量一致。字符数也应该与预准备语句中的参数标记（问号）的数量一致。

$var_n 变量引用。

返回值

如果绑定成功则为 TRUE，否则为 FALSE。

相关函数

- “sasql_prepare”一节第 739 页
- “sasql_stmt_param_count”一节第 751 页
- “sasql_stmt_bind_param_ex”一节第 745 页
- “sasql_stmt_execute”一节第 748 页

sasql_stmt_bind_param_ex

原型

bool **sasql_stmt_bind_param_ex(** sasql_stmt *$stmt*, int *$param_number*, mixed **&***$var*, string *$type* [, bool *$is_null* [, int *$direction*]] **)**

说明

将一个 PHP 变量绑定到语句参数。

参数

$stmt 由 sasql_prepare 函数返回的预准备语句资源。

$param_number 参数数字。它应该是 1 和 sasql_stmt_param_count 之间的一个数字。

$var PHP 变量。仅允许对 PHP 变量的引用。

$type 变量的类型。此类型可以是以下各项之一：**s**（对于字符串）、**i**（对于整数）、**d**（对于双精度型）、**b**（对于 blobs）。

$is_null 不论变量值是否为 NULL。

$direction 可以是 SASQL_D_INPUT、SASQL_D_OUTPUT 或 SASQL_INPUT_OUTPUT。

返回值

如果绑定成功则为 TRUE，否则为 FALSE。

相关函数

- “sasql_prepare”一节第 739 页
- “sasql_stmt_param_count”一节第 751 页
- “sasql_stmt_bind_param”一节第 744 页
- “sasql_stmt_execute”一节第 748 页

sasql_stmt_bind_result

原型

bool **sasql_stmt_bind_result(** sasql_stmt *$stmt*, mixed **&***$var1* [, mixed **&***$var2* ..] **)**

说明

将一或多个 PHP 变量绑定到已执行的语句的结果列，并返回结果集。

参数

$stmt 由 sasql_stmt_execute 执行的语句资源。

$var1 对于由 sasql_stmt_fetch 返回的、将被绑定到结果集列的 PHP 变量的引用。

返回值

TRUE（成功时）或 FALSE（失败时）。

相关函数

- “sasql_stmt_execute”一节第 748 页
- “sasql_stmt_fetch”一节第 748 页

sasql_stmt_close

原型

bool **sasql_stmt_close(** sasql_stmt *$stmt* **)**

说明

关闭所提供的语句资源并释放任何与其相关联的资源。此函数还将释放由 sasql_stmt_result_metadata 返回的任何结果对象。

参数

$stmt 由 sasql_prepare 函数返回的预准备语句资源。

返回值

TRUE（成功时）或 FALSE（失败时）。

相关函数

- “sasql_stmt_result_metadata”一节第 752 页
- “sasql_prepare”一节第 739 页

sasql_stmt_data_seek

原型

bool **sasql_stmt_data_seek(** sasql_stmt *$stmt*, int *$offset* **)**

说明

此函数查找结果集中的指定偏移量。

参数

$stmt 语句资源。

$offset 结果集中的偏移量。它是 0 与 sasql_stmt_num_rows 减去 1 之间的一个数字。

返回值

TRUE（成功时）或 FALSE（失败时）。

相关函数

- “sasql_stmt_num_rows”一节第 751 页

sasql_stmt_errno

原型

int **sasql_stmt_errno(** sasql_stmt *$stmt* **)**

说明

返回最近使用指定的语句资源执行的语句函数的错误代码。

参数

$stmt 由 sasql_prepare 函数返回的预准备语句资源。

返回值

一个整数错误代码。有关错误代码的列表，请参见“按 SQLCODE 排序的 SQL Anywhere 错误消息”一节 《错误消息》。

相关函数

- “sasql_stmt_error”一节第 747 页
- “sasql_error”一节第 729 页
- “sasql_errorcode”一节第 730 页
- “sasql_prepare”一节第 739 页
- “sasql_stmt_result_metadata”一节第 752 页

sasql_stmt_error

原型

string **sasql_stmt_error(** sasql_stmt *$stmt* **)**

说明

返回最近使用指定的语句资源执行的语句函数的错误文本。

参数

$stmt 由 sasql_prepare 函数返回的预准备语句资源。

返回值

描述错误的字符串。有关错误消息的列表，请参见“SQL Anywhere 错误消息” 《错误消息》。

相关函数

- “sasql_stmt_errno”一节第 747 页
- “sasql_error”一节第 729 页
- “sasql_errorcode”一节第 730 页
- “sasql_prepare”一节第 739 页
- “sasql_stmt_result_metadata”一节第 752 页

sasql_stmt_execute

原型

bool **sasql_stmt_execute(** sasql_stmt *$stmt* **)**

说明

执行预准备语句。sasql_stmt_result_metadata 可用于检查语句是否返回结果集。

参数

$stmt 由 sasql_prepare 函数返回的预准备语句资源。应该在执行调用前绑定变量。

返回值

TRUE（成功时）或 FALSE（失败时）。

相关函数

- “sasql_prepare”一节第 739 页
- “sasql_stmt_param_count”一节第 751 页
- “sasql_stmt_bind_param”一节第 744 页
- “sasql_stmt_bind_param_ex”一节第 745 页
- “sasql_stmt_result_metadata”一节第 752 页
- “sasql_stmt_bind_result”一节第 745 页

sasql_stmt_fetch

原型

bool **sasql_stmt_fetch(** sasql_stmt *$stmt* **)**

说明

此函数为语句读取结果之外的一行，并将列置于使用 sasql_stmt_bind_result 绑定的变量中。

参数

$stmt 语句资源。

返回值

TRUE（成功时）或 FALSE（失败时）。

相关函数

- “sasql_stmt_bind_result”一节第 745 页

sasql_stmt_field_count

原型

int **sasql_stmt_field_count(** sasql_stmt *$stmt* **)**

说明

此函数返回语句结果集中的列数。

参数

$stmt 语句资源。

返回值

语句结果中的列数。如果语句不返回结果，则它返回 0。

相关函数

- “sasql_stmt_result_metadata”一节第 752 页

sasql_stmt_free_result

原型

bool **sasql_stmt_free_result(** sasql_stmt *$stmt* **)**

说明

此函数释放语句的高速缓存结果集。

参数

$stmt 使用 sasql_stmt_execute 执行的语句资源。

返回值

TRUE（成功时）或 FALSE（失败时）。

相关函数

- “sasql_stmt_execute”一节第 748 页
- “sasql_stmt_store_result”一节第 753 页

sasql_stmt_insert_id

原型

int **sasql_stmt_insert_id(** sasql_stmt *$stmt* **)**

说明

返回最后插入到 IDENTITY 列或 DEFAULT AUTOINCREMENT 列中的值，如果是最后插入到了不含 IDENTITY 或 DEFAULT AUTOINCREMENT 列的表中，则返回零。

参数

$stmt 由 sasql_stmt_execute 执行的语句资源。

返回值

由前一个 INSERT 语句为 IDENTITY 列或 DEFAULT AUTOINCREMENT 列生成的 ID 或者零（如果最后的插入不影响 IDENTITY 或 DEFAULT AUTOINCREMENT 列）。如果 $stmt 无效，此函数可返回 FALSE (0)。

相关函数

- “sasql_stmt_execute”一节第 748 页

sasql_stmt_next_result

原型

bool **sasql_stmt_next_result(** sasql_stmt *$stmt* **)**

说明

此函数可以从语句前进到下一结果。如果没有另一个结果集，则放弃当前已完成的结果，并删除相关联的结果集对象（由 sasql_stmt_result_metadata 返回）。

参数

$stmt 语句资源。

返回值

TRUE（成功时）或 FALSE（失败时）。

相关函数

- “sasql_stmt_result_metadata”一节第 752 页

sasql_stmt_num_rows

原型

int **sasql_stmt_num_rows(** sasql_stmt *$stmt* **)**

说明

返回结果集中的行数。只有在调用 sasql_stmt_store_result 函数以缓冲整个结果集之后，才能确定结果集的实际行数。如果尚未调用 sasql_stmt_store_result 函数，则返回 0。

参数

$stmt 调用由 sasql_stmt_execute 和 sasql_stmt_store_result 执行的语句资源。

返回值

结果中可用的行数或 0（失败时）。

相关函数

- “sasql_stmt_execute”一节第 748 页
- “sasql_stmt_store_result”一节第 753 页

sasql_stmt_param_count

原型

int **sasql_stmt_param_count(** sasql_stmt *$stmt* **)**

说明

返回提供的预准备语句句柄中的参数数目。

参数

$stmt 由 sasql_prepare 函数返回的语句资源。

返回值

参数数目或 FALSE（错误时）。

相关函数

- “sasql_prepare”一节第 739 页
- “sasql_stmt_bind_param”一节第 744 页
- “sasql_stmt_bind_param_ex”一节第 745 页

sasql_stmt_reset

原型

bool **sasql_stmt_reset(** sasql_stmt *$stmt* **)**

说明

此函数在描述之后重置状态的 *$stmt* 对象。对所有被绑定的变量解除绑定，并删除所有使用 sasql_stmt_send_long_data 发送的数据。

参数

$stmt　语句资源。

返回值

TRUE（成功时）或 FALSE（失败时）。

相关函数

- “sasql_stmt_send_long_data”一节第 752 页

sasql_stmt_result_metadata

原型

sasql_result **sasql_stmt_result_metadata(** sasql_stmt *$stmt* **)**

说明

为所提供的语句返回结果集对象。

参数

$stmt　准备和执行的语句资源。

返回值

sasql_result 对象或 FALSE（如果语句不返回任何结果）。

sasql_stmt_send_long_data

原型

bool **sasql_stmt_send_long_data(** sasql_stmt *$stmt*, int *$param_number*, string *$data* **)**

说明

允许用户发送块中的参数数据。用户在尝试发送任何数据之前，必须先调用 sasql_stmt_bind_param 或 sasql_stmt_bind_param_ex。绑定参数必须属于字符串或 blob 类型。重复调用此函数将会附加至先前发送的内容。

参数

$stmt 使用 sasql_prepare 准备的语句资源。

$param_number 参数数字。它必须是 0 和 (sasql_stmt_param_count() - 1) 之间的一个数字。

$data 要发送的数据。

返回值

TRUE（成功时）或 FALSE（失败时）。

相关函数

- “sasql_stmt_bind_param”一节第 744 页
- “sasql_stmt_bind_param_ex”一节第 745 页
- “sasql_prepare”一节第 739 页
- “sasql_stmt_param_count”一节第 751 页

sasql_stmt_store_result

原型

bool **sasql_stmt_store_result(** sasql_stmt *$stmt* **)**

说明

此函数允许客户端高速缓存语句的整个结果集。可以使用函数 sasql_stmt_free_result 来释放高速缓存的结果。

参数

$stmt 使用 sasql_stmt_execute 执行的语句资源。

返回值

TRUE（成功时）或 FALSE（失败时）。

相关函数

- “sasql_stmt_free_result”一节第 749 页
- “sasql_stmt_execute”一节第 748 页

sasql_store_result

原型

sasql_result **sasql_store_result(** sasql_conn *$conn* **)**

说明

传输在将与 sasql_data_seek 函数配合使用的数据库连接 *$conn* 上所进行的最后一次查询的结果集。

参数

$conn 由连接函数返回的连接资源。

返回值

FALSE（如果查询未返回结果对象）或结果集对象（包含所有行的结果）。该结果在客户端被高速缓存。

相关函数

- “sasql_data_seek”一节第 728 页
- “sasql_stmt_execute”一节第 748 页

sasql_sqlstate

原型

string **sasql_sqlstate(** sasql_conn *$conn* **)**

说明

返回最近的 SQLSTATE 字符串。SQLSTATE 指示最近执行的 SQL 语句是否产生了成功、错误或警告状态。由 5 个字符 "00000" 所构成的 SQLSTATE 代码表示没有错误。这些值由 ISO/ANSI SQL 标准定义。

参数

$conn 由连接函数返回的连接资源。

返回值

返回一个包含当前 SQLSTATE 代码的由 5 个字符构成的字符串。注意，"00000" 表示没有错误。有关 SQLSTATE 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》。

相关函数

- “sasql_error”一节第 729 页
- “sasql_errorcode”一节第 730 页

sasql_use_result

原型

sasql_result **sasql_use_result(** sasql_conn *$conn* **)**

说明

为在连接上执行的最后一次查询启动结果集检索。

参数

$conn 由连接函数返回的连接资源。

返回值

FALSE（如果查询未返回结果对象或结果集对象）。该结果在客户端未被高速缓存。

相关函数

- “sasql_data_seek”一节第 728 页
- “sasql_stmt_execute”一节第 748 页

不建议使用 PHP 函数

支持但不建议使用以下 PHP 函数。这些函数中的每一个函数都拥有一个名称以 sasql_ instead of sqlanywhere_ 开头的较新的等效项。

sqlanywhere_commit（不建议使用）

原型

bool **sqlanywhere_commit(** resource *link_identifier* **)**

说明

不建议使用此函数。应该改用以下 PHP 函数： “sasql_commit”一节第 727 页.

结束 SQL Anywhere 数据库上的一个事务，并使该事务期间所做的所有更改永久生效。仅当 auto_commit 选项为 Off 时才有效。

参数

link_identifier 由 sqlanywhere_connect 函数返回的链接标识符。

返回值

TRUE（成功时）或 FALSE（失败时）。

示例

本示例说明 sqlanywhere_commit 如何能够用于引起特定连接上的提交操作。

```
$result = sqlanywhere_commit( $conn );
```

相关函数

- “sasql_commit”一节第 727 页
- “sasql_pconnect”一节第 738 页
- “sasql_disconnect”一节第 729 页
- “sqlanywhere_pconnect（不建议使用）”一节第 767 页
- “sqlanywhere_disconnect（不建议使用）”一节第 757 页

sqlanywhere_connect（不建议使用）

原型

resource **sqlanywhere_connect(** string *con_str* **)**

说明

不建议使用此函数。应该改用以下 PHP 函数： “sasql_connect”一节第 727 页.

建立与 SQL Anywhere 数据库的连接。

参数

con_str 一个由 SQL Anywhere 识别的连接字符串。

返回值

一个正的 SQL Anywhere 链接标识符（成功时），或一个错误和 0（失败时）。

示例

本示例传递连接字符串中的 SQL Anywhere 数据库的用户 ID 和口令。

```
$conn = sqlanywhere_connect( "UID=DBA;PWD=sql" );
```

相关函数

- “sasql_connect”一节第 727 页
- “sasql_pconnect”一节第 738 页
- “sasql_disconnect”一节第 729 页
- “sqlanywhere_pconnect（不建议使用）”一节第 767 页
- “sqlanywhere_disconnect（不建议使用）”一节第 757 页

sqlanywhere_data_seek（不建议使用）

原型

bool **sqlanywhere_data_seek(** resource *result_identifier*, int *row_num* **)**

说明

不建议使用此函数。应该改用以下 PHP 函数： “sasql_data_seek”一节第 728 页.

将游标定位到使用 sqlanywhere_query 打开的 *result_identifier* 的第 *row_num* 行上。

参数

result_identifier 由 sqlanywhere_query 函数返回的结果资源。

row_num 一个整数，表示 result_identifier 中游标的新位置。例如，指定 0 将游标移动到结果集的第一行，指定 5 将游标移动到第六行。负数表示相对于结果集结尾的行。例如，-1 将游标移动到结果集的最后一行，-2 将游标移动到倒数第二行。

返回值

TRUE（成功时）或 FALSE（错误时）。

示例

本示例说明如何找到结果集中的第六条记录。

```
sqlanywhere_data_seek( $result, 5 );
```

相关函数

- “sasql_data_seek”一节第 728 页
- “sasql_fetch_field”一节第 732 页
- “sasql_fetch_array”一节第 731 页
- “sasql_fetch_row”一节第 733 页
- “sasql_fetch_object”一节第 733 页
- “sasql_query”一节第 739 页
- “sqlanywhere_fetch_field（不建议使用）”一节第 761 页
- “sqlanywhere_fetch_array（不建议使用）”一节第 760 页
- “sqlanywhere_fetch_row（不建议使用）”一节第 763 页
- “sqlanywhere_fetch_object（不建议使用）”一节第 762 页
- “sqlanywhere_query（不建议使用）”一节第 767 页

sqlanywhere_disconnect（不建议使用）

原型

bool **sqlanywhere_disconnect(** resource *link_identifier* **)**

说明

不建议使用此函数。应该改用以下 PHP 函数： “sasql_disconnect”一节第 729 页.

关闭一个用 sqlanywhere_connect 打开的连接。

参数

link_identifier 由 sqlanywhere_connect 函数返回的链接标识符。

返回值

TRUE（成功时）或 FALSE（错误时）。

示例

本示例关闭与数据库的连接。

```
sqlanywhere_disconnect( $conn );
```

相关函数

- “sasql_disconnect”一节第 729 页
- “sasql_connect”一节第 727 页
- “sasql_pconnect”一节第 738 页
- “sqlanywhere_connect（不建议使用）”一节第 756 页
- “sqlanywhere_pconnect（不建议使用）”一节第 767 页

sqlanywhere_error（不建议使用）

原型

string **sqlanywhere_error(** [resource *link_identifier*] **)**

说明

不建议使用此函数。应该改用以下 PHP 函数： “sasql_error”一节第 729 页.

返回最近执行的 SQL Anywhere PHP 函数的错误文本。错误消息按连接进行存储。如果未指定 *link_identifier*，则 sqlanywhere_error 返回没有可用连接的最后一条错误消息。例如，如果您调用 sqlanywhere_connect，但连接失败，则可调用不含 *link_identifier* 参数的 sqlanywhere_error 以获取错误消息。如果要获取相应的 SQL Anywhere 错误代码值，请使用 sqlanywhere_errorcode 函数。

参数

link_identifier sqlanywhere_connect 或 sqlanywhere_pconnect 返回的链接标识符。

返回值

描述错误的字符串。

示例

本示例试图从不存在的表中进行选择。sqlanywhere_query 函数返回 FALSE，sqlanywhere_error 函数返回错误消息。

```
$result = sqlanywhere_query( $conn, "SELECT * FROM
table_that_does_not_exist" );
if( ! $result ) {
      $error_msg = sqlanywhere_error( $conn );
      echo "Query failed. Reason: $error_msg";
}
```

相关函数

- “sasql_error”一节第 729 页
- “sasql_errorcode”一节第 730 页
- “sasql_set_option”一节第 743 页
- “sqlanywhere_errorcode（不建议使用）”一节第 759 页
- “sqlanywhere_set_option（不建议使用）”一节第 770 页

sqlanywhere_errorcode（不建议使用）

原型

bool **sqlanywhere_errorcode(** [resource *link_identifier*] **)**

说明

不建议使用此函数。应该改用以下 PHP 函数： “sasql_errorcode”一节第 730 页.

返回最近执行的 SQL Anywhere PHP 函数的错误代码。错误代码按连接进行存储。如果未指定 *link_identifier*，则 sqlanywhere_errorcode 返回没有可用连接的最后一个错误代码。例如，如果您调用 sqlanywhere_connect，但连接失败，则可调用不含 *link_identifier* 参数的 sqlanywhere_errorcode 以获取错误代码。如果要获取相应的错误消息，请使用 sqlanywhere_error 函数。

参数

link_identifier sqlanywhere_connect 或 sqlanywhere_pconnect 返回的链接标识符。

返回值

一个整数，表示 SQL Anywhere 错误代码。错误代码 0 表示成功。正的错误代码表示成功但有警告。负的错误代码表示失败。

示例

本示例说明如何从失败的 SQL Anywhere PHP 调用检索最后一个错误代码。

```
$result = sqlanywhere_query( $conn, "SELECT * from
table_that_does_not_exist" );
   if( ! $result ) {
       $error_code = sqlanywhere_errorcode( $conn );
       echo "Query failed: Error code: $error_code";
   }
```

相关函数

- “sasql_error”一节第 729 页
- “sasql_set_option”一节第 743 页
- “sqlanywhere_error（不建议使用）”一节第 758 页
- “sqlanywhere_set_option（不建议使用）”一节第 770 页

sqlanywhere_execute（不建议使用）

原型

bool **sqlanywhere_execute(** resource *link_identifier*, string *sql_str* **)**

说明

不建议使用此函数。

准备并执行使用 sqlanywhere_connect 或 sqlanywhere_pconnect 打开且由 *link_identifier* 标识的连接上的 SQL 查询 *sql_str*。此函数根据查询执行的结果返回 TRUE 或 FALSE。此函数适合于不返回结果集的查询。如果希望得到结果集，请改用 sqlanywhere_query 函数。

参数

link_identifier sqlanywhere_connect 或 sqlanywhere_pconnect 返回的链接标识符。

sql_str 一个 SQL Anywhere 支持的 SQL 语句。

返回值

如果查询执行成功，则返回 TRUE，否则返回 FALSE 和错误消息。

示例

本示例说明如何使用 sqlanywhere_execute 函数执行 DDL 语句。

```
if( sqlanywhere_execute( $conn, "CREATE TABLE my_test_table( INT id )" ) ) {
   // handle success
} else {
   // handle failure
}
```

相关函数

- “sasql_query”一节第 739 页
- “sqlanywhere_query（不建议使用）”一节第 767 页

sqlanywhere_fetch_array（不建议使用）

原型

array **sqlanywhere_fetch_array(** resource *result_identifier* **)**

说明

不建议使用此函数。应该改用以下 PHP 函数： “sasql_fetch_array”一节第 731 页.

从结果集读取一行。该行作为一个可以按列名或列索引进行索引的数组返回。

参数

result_identifier 由 sqlanywhere_query 函数返回的结果资源。

返回值

一个表示结果集中的一行的数组，或 FALSE（无法获取行时）。

示例

本示例说明如何检索结果集中的所有行。每一行都作为一个数组返回。

```
$result = sqlanywhere_query( $conn, "SELECT GivenName, Surname FROM
Employees" );
  While( ($row = sqlanywhere_fetch_array( $result )) ) {
```



```
        echo " GivenName = " . $row["GivenName"] . " \n" ;
              echo " Surname = $row[1] \n";
         }
```

相关函数

- “sasql_fetch_array”一节第 731 页
- “sasql_data_seek”一节第 728 页
- “sasql_fetch_field”一节第 732 页
- “sasql_fetch_row”一节第 733 页
- “sasql_fetch_object”一节第 733 页
- “sasql_query”一节第 739 页
- “sqlanywhere_data_seek（不建议使用）”一节第 756 页
- “sqlanywhere_fetch_field（不建议使用）”一节第 761 页
- “sqlanywhere_fetch_row（不建议使用）”一节第 763 页
- “sqlanywhere_fetch_object（不建议使用）”一节第 762 页
- “sqlanywhere_query（不建议使用）”一节第 767 页

sqlanywhere_fetch_field（不建议使用）

原型

object **sqlanywhere_fetch_field(** resource *result_identifier* [, *field_offset*] **)**

说明

不建议使用此函数。应该改用以下 PHP 函数： “sasql_fetch_field”一节第 732 页.

返回一个包含关于某个特定列的信息的对象。

参数

result_identifier 由 sqlanywhere_query 函数返回的结果资源。

field_offset 一个整数，表示您希望在其中检索信息的列/字段。列的编号以零为基准；要获取第一列，应指定值 0。如果忽略此参数，则返回下一个字段对象。

返回值

一个具有以下属性的对象：

- **id** 包含字段号/列号
- **name** 包含字段/列的名称
- **numeric** 指出该字段是否是数值
- **length** 返回字段长度
- **type** 返回字段类型

示例

本示例说明如何使用 sqlanywhere_fetch_field 检索结果集的所有列信息。

```
$result = sqlanywhere_query($conn, "SELECT GivenName, Surname FROM
Employees");
while( ($field = sqlanywhere_fetch_field( $result )) ) {
        echo " Field ID = $field->id \n";
        echo " Field name = $field->name \n";
  }
```

相关函数

- “sasql_data_seek”一节第 728 页
- “sasql_fetch_field”一节第 732 页
- “sasql_fetch_array”一节第 731 页
- “sasql_fetch_row”一节第 733 页
- “sasql_fetch_object”一节第 733 页
- “sasql_query”一节第 739 页
- “sqlanywhere_data_seek（不建议使用）”一节第 756 页
- “sqlanywhere_fetch_array（不建议使用）”一节第 760 页
- “sqlanywhere_fetch_row（不建议使用）”一节第 763 页
- “sqlanywhere_fetch_object（不建议使用）”一节第 762 页
- “sqlanywhere_query（不建议使用）”一节第 767 页

sqlanywhere_fetch_object（不建议使用）

原型

object **sqlanywhere_fetch_object(** resource *result_identifier* **)**

说明

不建议使用此函数。应该改用以下 PHP 函数： “sasql_fetch_object”一节第 733 页.

从结果集读取一行。该行作为一个仅可按列名进行索引的对象返回。

参数

result_identifier 由 sqlanywhere_query 函数返回的结果资源。

返回值

代表结果集中的行的对象或 FALSE（行不可用时）。

示例

本示例说明如何从结果集中一次检索一行作为一个对象。列名可用作对象成员以访问列值。

```
$result = sqlanywhere_query( $conn, "SELECT GivenName, Surname FROM
Employees" );
  While( ($row = sqlanywhere_fetch_object( $result )) ) {
        echo "$row->GivenName \n";   # output the data in the first column
only.
  }
```


相关函数

- “sasql_data_seek”一节第 728 页
- “sasql_fetch_field”一节第 732 页
- “sasql_fetch_array”一节第 731 页
- “sasql_fetch_row”一节第 733 页
- “sasql_fetch_object”一节第 733 页
- “sasql_query”一节第 739 页
- “sqlanywhere_query（不建议使用）”一节第 767 页
- “sqlanywhere_data_seek（不建议使用）”一节第 756 页
- “sqlanywhere_fetch_field（不建议使用）”一节第 761 页
- “sqlanywhere_fetch_array（不建议使用）”一节第 760 页
- “sqlanywhere_fetch_row（不建议使用）”一节第 763 页

sqlanywhere_fetch_row（不建议使用）

原型

array **sqlanywhere_fetch_row(** resource *result_identifier* **)**

说明

不建议使用此函数。应该改用以下 PHP 函数： “sasql_fetch_row”一节第 733 页.

从结果集读取一行。该行作为一个仅可按列索引进行索引的数组返回。

参数

result_identifier 由 sqlanywhere_query 函数返回的结果资源。

返回值

一个表示结果集中的一行的数组，或 FALSE（无法获取行时）。

示例

本示例说明如何从结果集中一次检索一行。

```
while( ($row = sqlanywhere_fetch_row( $result )) ) {
        echo "$row[0] \n";     # output the data in the first column only.
  }
```

相关函数

- “sasql_fetch_row”一节第 733 页
- “sasql_data_seek”一节第 728 页
- “sasql_fetch_field”一节第 732 页
- “sasql_fetch_array”一节第 731 页
- “sasql_fetch_object”一节第 733 页
- “sasql_query”一节第 739 页
- “sqlanywhere_data_seek（不建议使用）”一节第 756 页
- “sqlanywhere_fetch_field（不建议使用）”一节第 761 页
- “sqlanywhere_fetch_array（不建议使用）”一节第 760 页
- “sqlanywhere_fetch_object（不建议使用）”一节第 762 页
- “sqlanywhere_query（不建议使用）”一节第 767 页

sqlanywhere_free_result（不建议使用）

原型

bool **sqlanywhere_free_result(** resource *result_identifier* **)**

说明

不建议使用此函数。应该改用以下 PHP 函数： “sasql_free_result”一节第 735 页.

释放与从 sqlanywhere_query 返回的结果资源相关联的数据库资源。

参数

result_identifier 由 sqlanywhere_query 函数返回的结果资源。

返回值

TRUE（成功时）或 FALSE（错误时）。

示例

本示例说明如何释放结果标识符的资源。

```
sqlanywhere_free_result( $result );
```

相关函数

- “sasql_query”一节第 739 页
- “sasql_free_result”一节第 735 页
- “sqlanywhere_query（不建议使用）”一节第 767 页

sqlanywhere_identity（不建议使用）

原型

int **sqlanywhere_identity(** resource *link_identifier* **)**

int **sqlanywhere_insert_id(** resource *link_identifier* **)**

说明

不建议使用此函数。应该改用以下 PHP 函数： “sasql_insert_id”一节第 735 页.

返回最后插入到 IDENTITY 列或 DEFAULT AUTOINCREMENT 列中的值，如果是最后插入到了不含 IDENTITY 或 DEFAULT AUTOINCREMENT 列的表中，则返回零。

提供 sqlanywhere_insert_id 函数是为了与 MySQL 数据库兼容。

参数

link_identifier sqlanywhere_connect 或 sqlanywhere_pconnect 返回的链接标识符。

返回值

由前一个 INSERT 语句为 AUTOINCREMENT 列生成的 ID，如果最后的插入对 AUTOINCREMENT 列没有影响，则返回零。如果 *link_identifier* 无效，则该函数会返回 FALSE。

示例

本示例说明如何能够使用 sqlanywhere_identity 函数检索由指定连接最新插入到表中的自动增量值。

```
if( sqlanywhere_execute( $conn, "INSERT INTO my_auto_increment_table VALUES
( 1 ) " ) ) {
      $insert_id = sqlanywhere_insert_id( $conn );
      echo "Last insert id = $insert_id";
}
```

相关函数

- “sasql_insert_id”一节第 735 页
- “sqlanywhere_execute（不建议使用）”一节第 759 页

sqlanywhere_num_fields（不建议使用）

原型

int **sqlanywhere_num_fields(** resource *result_identifier* **)**

说明

不建议使用此函数。应该改用以下 PHP 函数： “sasql_field_count”一节第 734 页.

返回 *result_identifier* 包含的列（字段）的数量。

参数

result_identifier 由 sqlanywhere_query 函数返回的结果资源。

返回值

一个正的列数，或一个错误（如果 *result_identifier* 无效）。

示例

本示例返回表示结果集中有多少列的值。

```
$num_columns = sqlanywhere_num_fields( $result );
```

相关函数

- “sasql_field_count”一节第 734 页
- “sasql_query”一节第 739 页
- “sqlanywhere_query（不建议使用）”一节第 767 页

sqlanywhere_num_rows（不建议使用）

原型

int **sqlanywhere_num_rows(** resource *result_identifier* **)**

说明

不建议使用此函数。应该改用以下 PHP 函数： “sasql_num_rows”一节第 738 页.

返回 *result_identifier* 包含的行数。

参数

result_identifier 由 sqlanywhere_query 函数返回的结果资源。

返回值

一个正数（如果行数是确切值），或一个负数（如果行数是估计值）。要获取确切的行数，必须针对数据库永久设置或针对连接临时设置数据库选项 row_counts。请参见“sasql_set_option”一节第 743 页。

示例

本示例说明如何检索结果集中返回的估计行数。

```
$num_rows = sqlanywhere_num_rows( $result );
   if( $num_rows < 0 ) {
       $num_rows = abs( $num_rows );    # take the absolute value as an
estimate
   }
```

相关函数

- “sasql_num_rows”一节第 738 页
- “sasql_query”一节第 739 页
- “sqlanywhere_query（不建议使用）”一节第 767 页

sqlanywhere_pconnect（不建议使用）

原型

resource **sqlanywhere_pconnect(** string *con_str* **)**

说明

不建议使用此函数。应该改用以下 PHP 函数： “sasql_pconnect”一节第 738 页.

建立与 SQL Anywhere 数据库的持久性连接。由于 Apache 创建子进程的方式，您可能会在使用 sqlanywhere_pconnect 代替 sqlanywhere_connect 时发现性能得到了提高。持久性连接可以用与连接池相似的方式提高性能。如果数据库服务器具有的连接的数量有限（例如，个人数据库服务器限制为 10 个并发连接），则使用持久性连接时应当谨慎。持久性连接可以附加到每个子进程上，并且如果 Apache 中拥有的子进程数目超过可用连接的数目，您就会接收到连接错误。

参数

con_str 一个由 SQL Anywhere 识别的连接字符串。

返回值

一个正的 SQL Anywhere 持久性链接标识符（成功时），或一个错误和 0（失败时）。

示例

本示例说明如何检索结果集中的所有行。每一行都作为一个数组返回。

```
$conn = sqlanywhere_pconnect( "UID=DBA;PWD=sql" );
```

相关函数

- “sasql_pconnect”一节第 738 页
- “sasql_connect”一节第 727 页
- “sasql_disconnect”一节第 729 页
- “sqlanywhere_connect（不建议使用）”一节第 756 页
- “sqlanywhere_disconnect（不建议使用）”一节第 757 页

sqlanywhere_query（不建议使用）

原型

resource **sqlanywhere_query(** resource *link_identifier*, string *sql_str* **)**

说明

不建议使用此函数。应该改用以下 PHP 函数： “sasql_query”一节第 739 页.

准备并执行使用 sqlanywhere_connect 或 sqlanywhere_pconnect 打开且由 *link_identifier* 标识的连接上的 SQL 查询 *sql_str*。对于不返回结果集的查询，可以使用 sqlanywhere_execute 函数。

参数

link_identifier 由 sqlanywhere_connect 函数返回的链接标识符。

sql_str 一个 SQL Anywhere 支持的 SQL 语句。

有关 SQL 语句的详细信息，请参见“SQL 语句”《SQL Anywhere 服务器 - SQL 参考》。

返回值

一个表示结果资源的正值（成功时），或 0 和一条错误消息（失败时）。

示例

本示例对 SQL Anywhere 数据库执行查询 SELECT * FROM SYSTAB。

```
$result = sqlanywhere_query( $conn, "SELECT * FROM SYSTAB" );
```

相关函数

- “sasql_query”一节第 739 页
- “sasql_free_result”一节第 735 页
- “sasql_fetch_array”一节第 731 页
- “sasql_fetch_field”一节第 732 页
- “sasql_fetch_object”一节第 733 页
- “sasql_fetch_row”一节第 733 页
- “sqlanywhere_execute（不建议使用）”一节第 759 页
- “sqlanywhere_free_result（不建议使用）”一节第 764 页
- “sqlanywhere_fetch_array（不建议使用）”一节第 760 页
- “sqlanywhere_fetch_field（不建议使用）”一节第 761 页
- “sqlanywhere_fetch_object（不建议使用）”一节第 762 页
- “sqlanywhere_fetch_row（不建议使用）”一节第 763 页

sqlanywhere_result_all（不建议使用）

原型

bool **sqlanywhere_result_all(** resource *result_identifier* **[,** *html_table_format_string* **[,** *html_table_header_format_string* **[,** *html_table_row_format_string* **[,** *html_table_cell_format_string* **]]]])**

说明

不建议使用此函数。

读取 *result_identifier* 的所有结果，并生成一个带可选格式字符串的 HTML 输出表。

参数

result_identifier 由 sqlanywhere_query 函数返回的结果资源。

html_table_format_string 一个适用于 HTML 表的格式字符串。例如，**"Border=1; Cellpadding=5"**。特殊值 none 不创建 HTML 表。在您要自定义列名或脚本时，这会很有用。如果您不想为此参数指定显式值，请对此参数值使用 NULL。

html_table_header_format_string 一个适用于 HTML 表的列标题的格式字符串。例如，**"bgcolor=#FF9533"**。特殊值 none 不创建 HTML 表。在您要自定义列名或脚本时，这会很有用。如果您不想为此参数指定显式值，请对此参数值使用 NULL。

html_table_row_format_string 一个适用于 HTML 表中的行的格式字符串。例如，**"onclick='alert('this')'"**。如果您希望交替使用不同的格式，请使用特殊标识 ><。标识的左侧指示对奇数行使用的格式，标识的右侧用于格式化偶数行。如果不在格式字符串中放入此标识，则所有行都使用相同的格式。如果您不想为此参数指定显式值，请对此参数值使用 NULL。

html_table_cell_format_string 一个适用于 HTML 表行中的单元格的格式字符串。例如，**"onclick='alert('this')'"**。如果您不想为此参数指定显式值，请对此参数值使用 NULL。

返回值

TRUE（成功时）或 FALSE（失败时）。

示例

本示例说明如何使用 sqlanywhere_result_all 生成具有结果集中所有行的一个 HTML 表。

```
$result = sqlanywhere_query( $conn, "SELECT GivenName, Surname FROM
Employees" );
  sqlanywhere_result_all( $result );
```

本示例说明如何通过样式表对各行交替应用不同的格式。

```
$result = sqlanywhere_query( $conn, "SELECT GivenName, Surname FROM
Employees");
sqlanywhere_result_all( $result, "border=2", "bordercolor=#3F3986",
"bgcolor=#3F3986 style=\"color=#FF9533\"", 'class="even"><class="odd"');
```

相关函数

- “sasql_query”一节第 739 页
- “sqlanywhere_query（不建议使用）”一节第 767 页

sqlanywhere_rollback（不建议使用）

原型

bool **sqlanywhere_rollback(** resource *link_identifier* **)**

说明

不建议使用此函数。应该改用以下 PHP 函数： “sasql_rollback”一节第 742 页.

结束 SQL Anywhere 数据库上的一个事务，并放弃该事务期间所做的所有更改。仅当 auto_commit 选项为 Off 时此函数才有效。

参数

link_identifier 由 sqlanywhere_connect 函数返回的链接标识符。

返回值

TRUE（成功时）或 FALSE（失败时）。

示例

此示例使用 sqlanywhere_rollback 回退连接。

```
$result = sqlanywhere_rollback( $conn );
```

相关函数

- “sasql_rollback”一节第 742 页
- “sasql_commit”一节第 727 页
- “sasql_set_option”一节第 743 页
- “sqlanywhere_commit（不建议使用）”一节第 755 页
- “sqlanywhere_set_option（不建议使用）”一节第 770 页

sqlanywhere_set_option（不建议使用）

原型

bool **sqlanywhere_set_option(** resource *link_identifier*, string *option*, mixed *value* **)**

说明

不建议使用此函数。应该改用以下 PHP 函数： “sasql_set_option”一节第 743 页.

在指定连接上设置指定选项的值。可以为以下选项设置值：

名称	说明	缺省值
auto_commit	此选项设置为 on 时，数据库服务器在每执行一条语句后提交。	on
row_counts	此选项设置为 FALSE 时，sqlanywhere_num_rows 函数返回受影响的行数估计值。如果想获得准确计数，请将此选项设置为 TRUE。	FALSE
verbose_errors	此选项设置为 TRUE 时，PHP 驱动程序返回详细的错误信息。此选项设置为 FALSE 时，必须调用 sqlanywhere_error 或 sqlanywhere_errorcode 函数以获取更详细的错误信息。	TRUE

可以通过在 *php.ini* 文件中加入以下行，更改选项的缺省值。在本示例中，为 auto_commit 选项设置了缺省值。

```
sqlanywhere.auto_commit=0
```

参数

link_identifier 由 sqlanywhere_connect 函数返回的链接标识符。

option 要设置的选项的名称。

value 新的选项值。

返回值

TRUE（成功时）或 FALSE（失败时）。

示例

以下示例说明设置 auto_commit 选项值的不同方法。

```
$result = sqlanywhere_set_option( $conn, "auto_commit", "Off" );

$result = sqlanywhere_set_option( $conn, "auto_commit", 0 );

$result = sqlanywhere_set_option( $conn, "auto_commit", False );
```

相关函数

- “sasql_set_option”一节第 743 页
- “sasql_commit”一节第 727 页
- “sasql_error”一节第 729 页
- “sasql_errorcode”一节第 730 页
- “sasql_num_rows”一节第 738 页
- “sasql_rollback”一节第 742 页
- “sqlanywhere_commit（不建议使用）”一节第 755 页
- “sqlanywhere_error（不建议使用）”一节第 758 页
- “sqlanywhere_errorcode（不建议使用）”一节第 759 页
- “sqlanywhere_num_rows（不建议使用）”一节第 766 页
- “sqlanywhere_rollback（不建议使用）”一节第 769 页

在 UNIX 和 Mac OS X 上构建 SQL Anywhere PHP 模块

在 Unix 和 Mac OS X 上，要使用 SQL Anywhere PHP 模块将 PHP 连接到 SQL Anywhere，则必须将 SQL Anywhere PHP 模块文件添加到 PHP 的源树，然后重新编译 PHP。

要求

下面列出了要完成本文档中所详述的步骤需要在系统上安装的软件：

- 必须安装可与 Apache Web 服务器在同一台计算机上（或不同的计算机上）运行的 SQL Anywhere。
- 可从 http://download.sybase.com/ianywhere/php/2.0.3/src/sasql_php.zip 下载 SQL Anywhere PHP 模块的源代码。

  还需要安装有 **sqlpp** 和 *libdblib11.so* (Unix) 或 *libdblib11.dylib* (Mac OS X)（请查看您的 SQL Anywhere *lib32* 目录）。
- 需要 PHP 源代码（可以从 http://www.php.net 下载）。PHP 的 5.2.6 版本是最新的稳定版本。
- 需要 Apache Web 服务器源代码（可以从 http://httpd.apache.org 下载）。如果您打算使用 Apache 的预建版本，请确保您已安装了 **apache** 和 **apache-devel**。
- 如果您计划使用标准的 ODBC PHP 模块，则需要安装有 *libdbodbc11.so* (Unix) 或 *libdbodbc11.dylib* (Mac OS X)（请查看您的 SQL Anywhere *lib32* 目录）。

应该从您的 Unix 安装磁盘中安装以下二进制文件（如果还未安装它们），并且可以 RPM 的形式找到这些二进制文件：

- make
- automake
- autoconf
- libtool（对于 Mac OS X 是 glibtool）
- makeinfo
- bison
- gcc
- cpp
- glibc-devel
- kernel-headers
- flex

要执行特定的安装步骤，必须具有与安装 PHP 的用户相同的访问权限。大多数基于 Unix 的系统都会提供一个 **sudo** 命令，该命令允许那些权限不够的用户如同具有足够权限的用户那样执行特定的命令。

将 SQL Anywhere PHP 模块文件添加到 PHP 源树

1. 可以从 http://download.sybase.com/ianywhere/php/2.0.3/src/sasql_php.zip 下载 SQL Anywhere PHP 模块。
2. 将文件从保存 SQL Anywhere PHP 模块的目录中抽取到 PHP 源树的 *ext* 子目录中（Mac OS X 用户应该用 **gnutar** 替换 **tar**）：

   ```
   $ tar -xzf sasql_php.zip -C PHP-source-directory/ext/
   ```

   例如：

   ```
   $ tar -xzf sqlanywhere_php-1.0.8.tar.gz -C ~/php-5.2.6/ext
   ```

3. 让 PHP 接受该模块：

   ```
   $ cd PHP-source-directory/ext/sqlanywhere
   $ touch *
   $ cd ~/PHP-source-directory
   $ ./buildconf
   ```

   以下示例适用于 PHP 版本 5.2.6。必须将 **php-5.2.6** 设置为正在使用的 PHP 版本。

   ```
   $ cd ~/php-5.2.6/ext/sqlanywhere
   $ touch *
   $ cd ~/php-5.2.6
   $ ./buildconf
   ```

4. 验证 PHP 是否接受该模块：

   ```
   $ ./configure -help | egrep sqlanywhere
   ```

   如果已成功让 PHP 接受了 SQL Anywhere 模块，则应看到以下文本：

   ```
   --with-sqlanywhere=[DIR]
   ```

   如果未成功，则跟踪此命令的输出并将其发布到 **sybase.public.sqlanywhere.linux** 新闻组以寻求帮助。

编译 Apache 和 PHP

PHP 可作为 Web 服务器（如 Apache）或 CGI 可执行程序的共享模块进行编译。如果正在使用 PHP 不支持的 Web 服务器，或者希望在命令 shell 中而不在 Web 页上执行 PHP 脚本，则应将 PHP 作为 CGI 可执行文件进行编译。否则，如果希望安装 PHP 以与 Apache 配合使用，则需将它作为 Apache 模块进行编译。

有关如何将 PHP 作为 Apache 模块进行编译的信息，请参见“将 PHP 作为 Apache 模块进行编译”一节第 774 页。

有关如何将 PHP 作为 CGI 可执行文件进行编译的信息，请参见“将 PHP 作为 CGI 可执行文件进行编译”一节第 776 页。

将 PHP 作为 Apache 模块进行编译

以下说明中的前两步将会配置 Apache 以使它能识别共享模块。如果您的系统上已经安装了 Apache 的已编译版本，则直接进入第 3 步。请注意，Mac OS X 附带有预安装的 Apache Web 服务器。

♦ 将 PHP 作为 Apache 模块进行编译

1. 配置 Apache 以识别共享模块。

   从提取 Apache 文件的目录中执行以下命令（在一行中输入所有内容）：

   ```
   $ cd Apache-source-directory
   $ ./configure --enabled-shared=max --enable-module=most --
   prefix=/Apache-installation-directory
   ```

   以下示例适用于 Apache 版本 2.2.9。必须将 **apache_2.2.9** 设置为正在使用的 Apache 版本。

   ```
   $ cd ~/apache_2.2.9
   $ ./configure --enabled-shared=max --enable-module=most --
   prefix=/usr/local/web/apache
   ```

2. 重新编译和安装相关组件：

   ```
   $ make
   $ make install
   ```

   现在，可以编译 PHP 来将其作为 Apache 模块运行。

3. 确保设置了适用于 SQL Anywhere 的环境。

   根据您使用的 shell，从 SQL Anywhere 的安装目录（缺省情况下，为 */opt/sqlanywhere11*）输入相应的命令。在 Mac OS X 上，缺省目录为 */Applications/SQLAnywhere11/System*。

如果正在使用此 shell...	...使用此命令
sh、ksh、bash	`. ./bin32/sa_config.sh`
csh、tcsh	`source ./bin32/sa_config.csh`

4. 将 PHP 作为 Apache 模块进行配置以包括 SQL Anywhere PHP 模块。

   执行以下命令：

   ```
   $ cd PHP-source-directory
   $ ./configure --with-sqlanywhere --with- apxs=/Apache-installation-
   directory/bin/apxs
   ```

   以下示例适用于 PHP 版本 5.2.6。必须将 **php-5.2.6** 设置为正在使用的 PHP 版本。

   ```
   $ cd ~/php-5.2.6
   $ ./configure --with-sqlanywhere --with- apxs=/usr/local/web/apache/bin/
   apxs
   ```

   configure 脚本将尝试确定 SQL Anywhere 安装的版本及位置。在此命令的输出中，应看到与以下内容类似的几行：

```
checking for SQL Anywhere support... yes
checking     SQL Anywhere install dir... /opt/sqlanywhere11
checking     SQL Anywhere version... 11
```

5. 重新编译相关组件：

   ```
   $ make
   ```

6. 检查是否已正确链接到库。

   - Linux 用户（以下示例假定您正在使用 PHP 版本 5）：

     ```
     ldd ./.libs/libphp5.so
     ```

   - Mac OS X 用户：

     请参考 *httpd.conf* 配置文件以确定计算机上的 *libphp5.so* 位置。使用以下命令执行检查：

     ```
     otool -L $LIBPHP5_DIR/libphp5.so
     ```

     $LIBPHP5_DIR 是 *libphp5.so* 所在的目录（根据您的服务器配置）。

     此命令输出由 *libphp5.so* 使用的库的列表。验证 *libdblib11.so* 在该列表中。

7. 在 Apache 的 *lib* 目录中安装 PHP 二进制文件：

   ```
   $ make install
   ```

8. 执行验证。PHP 会自动执行此操作。只需确保 *httpd.conf* 配置文件经过验证以使 Apache 承认 *.php* 文件作为 PHP 脚本。

   httpd.conf 存储在 *Apache* 目录的 *conf* 子目录中：

   ```
   $ cd Apache-installation-directory/conf
   ```

   例如：

   ```
   $ cd /usr/local/web/apache/conf
   ```

   在编辑文件前制作 *httpd.conf* 的一个备份副本（可使用所选的文本编辑器替换 **pico**）：

   ```
   $ cp httpd.conf httpd.conf.backup
   $ pico httpd.conf
   ```

   在 *httpd.conf* 中添加以下几行或取消这几行的注释（这些行在文件中不在一起）：

   ```
   LoadModule php5_module   libexec/libphp5.so
   AddModule mod_php5.c
   AddType application/x-httpd-php .php
   AddType application/x-httpd-php-source .phps
   ```

   注意
   在 Mac OS X 上，*httpd_macosxserver.conf* 中的最后两行应添加或取消注释。

   开头两行会将 Apache 指向用于解释 PHP 代码的文件，而另外两行声明扩展名为 *.php* 或 *.phps* 的文件的文件类型，以便 Apache 能够识别并正确地处理它们。

有关测试和使用您的设置的信息，请参见“在 Web 页中运行 PHP 测试脚本”一节第 718 页。

将 PHP 作为 CGI 可执行文件进行编译

♦ **将 PHP 作为 CGI 可执行文件进行编译**

1. 确保设置了适用于 SQL Anywhere 的环境。

   有关为 SQL Anywhere 设置环境的说明，请遵照“将 PHP 作为 Apache 模块进行编译”一节第 774 页的第 4 步中的说明。

2. 将 PHP 作为 CGI 可执行文件进行配置以便与 SQL Anywhere PHP 模块一起使用。

   从提取 PHP 文件的目录中执行以下命令：

   ```
   $ cd PHP-source-directory
   $ ./configure --with-sqlanywhere
   ```

   例如：

   ```
   $ cd ~/php-5.2.6/
   $ ./configure --with-sqlanywhere
   ```

   配置脚本将尝试确定 SQL Anywhere 安装的版本及位置。如果检查此命令的输出，应看到与以下内容类似的几行：

   ```
   checking for SQL Anywhere support... yes
   checking     SQL Anywhere install dir... /opt/sqlanywhere10
   checking     SQL Anywhere version... 9
   ```

3. 编译可执行文件：

   ```
   $ make
   ```

4. 安装组件。

   ```
   $ make install
   ```

有关测试和使用 PHP 的信息，请参见“在 Web 页中运行 PHP 测试脚本”一节第 718 页。

SQL Anywhere for Ruby

目录

SQL Anywhere 中的 Ruby 支持

SQL Anywhere for Ruby 项目中提供三个单独的程序包。安装其中任何一个程序包最简单的方法是使用 **RubyGems**。要获得 RubyGems，请访问以下站点：http://rubyforge.org/projects/rubygems/。建议安装 1.3.1 版或更高版本。

SQL Anywhere Ruby 项目的主页为 http://sqlanywhere.rubyforge.org/。

SQL Anywhere 本地 Ruby 驱动程序

sqlanywhere 此程序包是低层驱动程序，允许 Ruby 代码与 SQL Anywhere 数据库结合。此程序包通过 SQL Anywhere C API 公开的接口提供了一个 Ruby 包装。此程序包使用 C 语言编写，可以源代码或预编译 gem 的形式提供，适用于 Windows 和 Linux。如果已安装 **RubyGems**，可以通过运行以下命令获得此程序包：

```
gem install sqlanywhere
```

请注意，此程序包是任何其它 SQL Anywhere Ruby 程序包的前提条件。有关详细信息，请参见：

- “SQL Anywhere Ruby API”一节第 786 页
- 下载源代码 (http://rubyforge.org/projects/sqlanywhere)
- RDoc (http://sqlanywhere.rubyforge.org/sqlanywhere/)
- Ruby 编程语言 (http://www.ruby-lang.org)
- RubyForge / Ruby Central (http://rubyforge.org/)

SQL Anywhere ActiveRecord 适配器

activerecord-sqlanywhere-adapter 此程序包是一个允许 ActiveRecord 与 SQL Anywhere 进行通信的适配器。ActiveRecord 是一个对象相关映射程序，作为 Ruby on Rails Web 开发框架的一部分而为人所知。此程序包使用纯 Ruby 编写，以源代码或 gem 格式提供。此适配器使用（并依赖于）**sqlanywhere** gem。如果已安装 **RubyGems**，可以通过运行以下命令安装此程序包及其依赖包：

```
gem install activerecord-sqlanywhere-adapter
```

有关详细信息，请参见：

- “SQL Anywhere 中的 Rails 支持”一节第 780 页
- 下载源代码 (http://rubyforge.org/projects/sqlanywhere)
- RDoc (http://sqlanywhere.rubyforge.org/activerecord-sqlanywhere-adapter)
- Ruby on Rails (http://www.rubyonrails.org/)

SQL Anywhere Ruby/DBI 驱动程序

dbi 此程序包是 Ruby 的 DBI 驱动程序。如果已安装 **RubyGems**，可以通过运行以下命令安装此程序包及其依赖包：

```
gem install dbi
```

dbd-sqlanywhere 此程序包是一个允许 Ruby/DBI 与 SQL Anywhere 进行通信的驱动程序。Ruby/DBI 是模仿 Perl 流行的 DBI 模块的通用数据库接口。此程序包使用纯 Ruby 编写，以源代码或

gem 格式提供。此驱动程序使用（并依赖于）**sqlanywhere** gem。如果已安装 **RubyGems**，可以通过运行以下命令安装此程序包及其依赖包：

```
gem install dbd-sqlanywhere
```

有关详细信息，请参见：

- “SQL Anywhere 的 Ruby DBI 驱动程序”一节第 782 页
- 下载源代码 (http://rubyforge.org/projects/sqlanywhere)
- RDoc (http://sqlanywhere.rubyforge.org/dbd-sqlanywhere)
- Ruby/DBI - 用于 Ruby 的直接数据库访问 (http://ruby-dbi.rubyforge.org/)

有关其中任何一个程序包的反馈信息，请使用 sqlanywhere-users@rubyforge.com 邮件列表。有关在 Web 环境中使用 SQL Anywhere 的一般问题，请使用 SQL Anywhere Web 开发论坛。有关 SQL Anywhere 及其用法的常见问题，请使用 sybase.public.sqlanywhere.general 新闻组。

SQL Anywhere 中的 Rails 支持

Rails 是用 Ruby 语言编写的 Web 开发框架。其优势在于 Web 应用程序开发。在尝试 Rails 开发之前，强烈建议您熟悉 Ruby 编程语言。可以考虑将 “SQL Anywhere Ruby API”一节第 786 页 包括在对 Ruby 的熟悉中。

如果您准备好投入 Rails 开发，还需要完成以下几件事情。

前提条件

- **RubyGems** 您应该安装 RubyGems。它使 Ruby 程序包的安装变得更加容易。编写本文档时，Rails 开发要求 1.3.1 版。Ruby on Rails (http://www.rubyonrails.org/) Web 站点将引导您安装正确版本。
- **Ruby** 需要在系统上安装 Ruby 解释器。Ruby on Rails (http://www.rubyonrails.org/) Web 站点会推荐要安装的版本。
- **Rails** 使用 RubyGems，只需一个命令行就可以安装所有 Rails 及其依赖包：

  ```
  gem install rails
  ```

- **activerecord-sqlanywhere-adapter** 如果尚未安装 SQL Anywhere ActiveRecord 支持，则必须安装才能使用 SQL Anywhere 进行 Rails 开发。使用 RubyGems，只需一个命令行就可以安装所有 SQL Anywhere ActiveRecord 支持及其依赖包：

  ```
  gem install activerecord-sqlanywhere-adapter
  ```

开始之前

在安装必备组件之后，还必须执行最后几个步骤，然后才能使用 SQL Anywhere 开始 Rails 开发。这些步骤是将 SQL Anywhere 添加到 Rails 支持的数据库管理系统集所必需的。

1. 必须在 Rails *configs\databases* 目录中创建 *sqlanywhere.yml* 文件。如果已将 Ruby 安装在 *\Ruby* 路径中，并安装了 2.2.2 版本的 Rails，则此文件的路径为 *\Ruby\lib\ruby\gems\1.8\gems\rails-2.2.2\configs\databases*。此文件的内容应该为：

   ```
   #
   # SQL Anywhere database configuration
   #
   # This configuration file defines the patten used for
   # database filenames. If your application is called "blog",
   # then the database names will be blog_development,
   # blog_test, blog_production. The specified username and
   # password should permit DBA access to the database.
   #

   development:
     adapter: sqlanywhere
     database: <%= app_name %>_development
     username: DBA
     password: sql

   # Warning: The database defined as "test" will be erased and
   # re-generated from your development database when you run "rake".
   # Do not set this db to the same as development or production.
   test:
     adapter: sqlanywhere
   ```



```
  database: <%= app_name %>_test
  username: DBA
  password: sql

production:
  adapter: sqlanywhere
  database: <%= app_name %>_production
  username: DBA
  password: sql
```

2. 必须更新 Rails *app_generator.rb* 文件。使用上述步骤 1 中的相同假设，此文件位于 *\Ruby\lib\ruby\gems\1.8\gems\rails-2.2.2\lib\rails_generator\generators\applications\app* 路径中。编辑 *app_generator.rb* 文件，并找到以下行：

   ```
   DATABASES = %w(mysql oracle postgresql sqlite2 sqlite3 frontbase ibm_db)
   ```

   将 **sqlanywhere** 添加到如下列表。

   ```
   DATABASES = %w(sqlanywhere mysql oracle postgresql sqlite2 sqlite3
   frontbase ibm_db)
   ```

   如果需要，还可以更改 **DEFAULT_DATABASE** 设置（在下一行上），如下所示：

   ```
   DEFAULT_DATABASE = 'sqlanywhere'
   ```

   现在保存文件并退出。

了解 Rails

建议您从 Ruby on Rails Web 站点上优秀的 Rails 入门教程开始。在此教程中，将介绍用于初始化 **blog** 项目的命令。以下是用于初始化与 SQL Anywhere 一起使用的 **blog** 项目的命令。

```
rails blog -d sqlanywhere
```

如果已更改 **DEFAULT_DATABASE** 设置，则不需要 `-d sqlanywhere` 选项

另外，请注意，**blog** 教程需要建立三个数据库。初始化项目之后，可以切换到项目的根目录，并按如下方式创建三个数据库。

```
dbinit blog_development
dbinit blog_test
dbinit blog_production
```

必须按如下方式启动数据库服务器和三个数据库，然后才能继续。

```
dbsrv11 blog_development.db blog_production.db blog_test.db
```

现在您可以利用该教程探究 Ruby on Rails Web 开发了。

有关 Ruby on Rails Web 开发框架的详细信息，请访问 Ruby on Rails (http://www.rubyonrails.org/) Web 站点。

SQL Anywhere 的 Ruby DBI 驱动程序

本节将概述如何编写使用 SQL Anywhere DBI 驱动程序的 Ruby 应用程序。可从 http://ruby-dbi.rubyforge.org/ 在线获取 DBI 模块的完整文档。

装载 DBI 模块

要在 Ruby 应用程序中使用 DBI:SQLAnywhere 接口，必须先告诉 Ruby 您打算使用 Ruby DBI 模块。为此，请在 Ruby 源代码文件的顶部附近包括以下行。

```
require 'dbi'
```

DBI 模块会根据需要自动装载 SQL Anywhere 数据库驱动程序 (DBD) 接口。

打开和关闭连接

通常，打开一个到数据库的连接，然后通过该连接运行一系列 SQL 语句来执行所有需要的操作。要打开连接，请使用 connect 函数。返回值是一个到数据库连接的句柄，使用该句柄可以在连接上执行后继操作。

对 connect 函数的调用采用一般形式：

```
dbh = DBI.connect('DBI:SQLAnywhere:server-name', user-id, password, options)
```

- ***server-name*** 是想要连接到的数据库服务器的名称。也可以按 "option1=value1;option2=value2;..." 的格式指定连接字符串。
- ***user-id*** 是有效用户 ID。除非此字符串为空，否则将 ";UID=value" 附加到连接字符串。
- ***password*** 是用户 ID 的相应口令。除非此字符串为空，否则将 ";PWD=value" 附加到连接字符串。
- ***options*** 是一列附加连接参数，如 DatabaseName、DatabaseFile 和 ConnectionName。这些参数以 "option1=value1;option2=value2;..." 的格式附加到连接字符串。

在使用 connect 函数之前，请启动数据库服务器和示例数据库。

```
dbeng11 samples-dir\demo.db
```

以下代码示例打开然后关闭到 SQL Anywhere 示例数据库的连接。下面示例中的字符串 "demo" 是服务器名。

```
require 'dbi'
DBI.connect('DBI:SQLAnywhere:demo', 'DBA', 'sql') do |dbh|
    if dbh.ping
       print "Successfully Connected\n"
       dbh.disconnect()
    end
end
```

也可以指定一个连接字符串替换服务器名。例如，在上面的脚本中，可以通过替换 connect 函数的第一个参数进行更改，如下所示：

```
require 'dbi'
DBI.connect('DBI:SQLAnywhere:ENG=demo;DBN=demo;UID=DBA;PWD=sql') do |dbh|
    if dbh.ping
```



```
        print "Successfully Connected\n"
        dbh.disconnect()
    end
end
```

因为在连接字符串中指定了用户 ID 和口令，所以不必指定 connect 函数的参数 2 和 3。但是，如果传递附加连接参数的散列值，则为用户 ID 和口令参数指定空字符串 ('')。

以下示例说明如何将附加连接参数作为散列键/值对传递到 connect 函数。

```
require 'dbi'
DBI.connect('DBI:SQLAnywhere:demo', 'DBA', 'sql',
        { :ConnectionName => "RubyDemo",
          :DatabaseFile => "demo.db",
          :DatabaseName => "demo" }
        ) do |dbh|
    if dbh.ping
        print "Successfully Connected\n"
        dbh.disconnect()
    end
end
```

选择数据

获得了打开的连接的句柄后，您可以访问和修改存储在数据库中的数据。可能最简单的操作是检索某些行并输出它们。

必须先执行 SQL 语句。如果语句返回结果集，可以使用得到的语句句柄检索有关结果集和结果集行的元信息。以下示例从元数据获得列名，并显示获取的每一行的列名和值。

```
require 'dbi'

def db_query( dbh, sql )
    sth = dbh.execute(sql)
    print "# of Fields:  #{sth.column_names.size}\n"
    sth.fetch do |row|
        print "\n"
        sth.column_info.each_with_index do |info, i|
            unless info["type_name"] == "LONG VARBINARY"
                print "#{info["name"]}=#{row[i]}\n"
            end
        end
    end
    sth.finish
end

begin
    dbh = DBI.connect('DBI:SQLAnywhere:demo', 'DBA', 'sql')
    db_query(dbh, "SELECT * FROM Products")
rescue DBI::DatabaseError => e
    puts "An error occurred"
    puts "Error code: #{e.err}"
    puts "Error message: #{e.errstr}"
    puts "Error SQLSTATE: #{e.state}"
ensure
    dbh.disconnect if dbh
end
```

显示的输出的前几行如下所示。

```
# of Fields:  8
```

```
ID=300
Name=Tee Shirt
Description=Tank Top
Size=Small
Color=White
Quantity=28
UnitPrice=9.00

ID=301
Name=Tee Shirt
Description=V-neck
Size=Medium
Color=Orange
Quantity=54
UnitPrice=14.00
```

完成后调用 finish 释放语句句柄十分重要。如果不释放句柄，则可能收到如下所示的错误：

```
Resource governor for 'prepared statements' exceeded
```

要检测句柄泄漏，缺省情况下 SQL Anywhere 数据库服务器将允许的游标和准备好的语句数量限制为每个连接最多 50 个。如果超过这些限制，资源调控器自动生成错误。如果收到此错误，请检查未被释放的语句句柄。请谨慎使用 prepare_cached，因为语句句柄并没有被释放。

如有必要，可通过设置 max_cursor_count 和 max_statement_count 选项来修改这些限制。请参见 “max_cursor_count 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》和 “max_statement_count 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

插入行

插入行需要打开的连接的句柄。插入行最简单的方法是使用参数化的 INSERT 语句，这意味着问号用作值的占位符。首先准备好语句，然后对每一新行执行一次。新行的值作为参数提供给 execute 方法。

```
require 'dbi'

def db_query( dbh, sql )
    sth = dbh.execute(sql)
    print "# of Fields:  #{sth.column_names.size}\n"
    sth.fetch do |row|
        print "\n"
        sth.column_info.each_with_index do |info, i|
            unless info["type_name"] == "LONG VARBINARY"
                print "#{info["name"]}=#{row[i]}\n"
            end
        end
    end
    sth.finish
end

def db_insert( dbh, rows )
    sql = "INSERT INTO Customers (ID, GivenName, Surname,
                Street, City, State, Country, PostalCode,
                Phone, CompanyName)
           VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)"
    sth = dbh.prepare(sql);
    rows.each do |row|
        sth.execute(row[0],row[1],row[2],row[3],row[4],
                    row[5],row[6],row[7],row[8],row[9])
    end
end
```



```
begin
    dbh = DBI.connect('DBI:SQLAnywhere:demo', 'DBA', 'sql')
    rows = [
        [801,'Alex','Alt','5 Blue Ave','New York','NY','USA',
            '10012','5185553434','BXM'],
        [802,'Zach','Zed','82 Fair St','New York','NY','USA',
            '10033','5185552234','Zap']
       ]
    db_insert(dbh, rows)
    dbh.commit
    db_query(dbh, "SELECT * FROM Customers WHERE ID > 800")
rescue DBI::DatabaseError => e
    puts "An error occurred"
    puts "Error code: #{e.err}"
    puts "Error message: #{e.errstr}"
    puts "Error SQLSTATE: #{e.state}"
ensure
    dbh.disconnect if dbh
end
```

SQL Anywhere Ruby API

SQL Anywhere 提供了 SQL Anywhere C API 的低层接口。以下几节中介绍的 API 允许快速开发 SQL 应用程序。要演示 Ruby 应用程序开发能力，可考虑以下 Ruby 程序示例。该程序装载 SQL Anywhere Ruby 扩展，连接到 demo 数据库，列出 Products 表中的列值，断开连接并终止。

```
begin
    require 'rubygems'
    gem 'sqlanywhere'
    unless defined? SQLAnywhere
        require 'sqlanywhere'
    end
end
api = SQLAnywhere::SQLAnywhereInterface.new()
SQLAnywhere::API.sqlany_initialize_interface( api )
api.sqlany_init()
conn = api.sqlany_new_connection()
api.sqlany_connect( conn, "DSN=SQL Anywhere 11 Demo" )
stmt = api.sqlany_execute_direct( conn, "SELECT * FROM Products" )
num_rows = api.sqlany_num_rows( stmt )
num_rows.times {
    api.sqlany_fetch_next( stmt )
    num_cols = api.sqlany_num_cols( stmt )
    for col in 1..num_cols do
        info = api.sqlany_get_column_info( stmt, col - 1 )
        unless info[3]==1 # Don't do binary
            rc, value = api.sqlany_get_column( stmt, col - 1 )
            print "#{info[2]}=#{value}\n"
        end
    end
    print "\n"
}
api.sqlany_free_stmt( stmt )
api.sqlany_disconnect(conn)
api.sqlany_free_connection(conn)
api.sqlany_fini()
SQLAnywhere::API.sqlany_finalize_interface( api )
```

此 Ruby 程序输出的结果集的前两行如下所示：

```
ID=300
Name=Tee Shirt
Description=Tank Top
Size=Small
Color=White
Quantity=28
UnitPrice=9.00

ID=301
Name=Tee Shirt
Description=V-neck
Size=Medium
Color=Orange
Quantity=54
UnitPrice=14.00
```

以下几节介绍每个支持的函数。

sqlany_affected_rows

返回受执行预准备语句影响的行数。

语法

sqlany_affected_rows (*$stmt*)

参数

- **$stmt** 预准备并成功执行的语句，其中不返回任何结果集。例如，执行了 INSERT、UPDATE 或 DELETE 语句。

返回值

返回标量值（受影响的行数）或返回 -1（失败时）。

另请参见

- “sqlany_execute”一节第 791 页

示例

```
affected = api.sqlany_affected( stmt )
```

sqlany_bind_param

将用户提供的缓冲区作为参数绑定到预准备语句。

语法

sqlany_bind_param (*$stmt*, *$index*, *$param*)

参数

- **$stmt** 成功执行 sqlany_prepare 所返回的语句对象。
- **$index** 参数的索引。该数字必须在 0 到 sqlany_num_params() - 1 之间。
- **$param** 检索自 sqlany_describe_bind_param 的填充的绑定对象。

返回值

返回标量值，成功时为 1，不成功时为 0。

另请参见

- “sqlany_describe_bind_param”一节第 789 页

示例

```
stmt = api.sqlany_prepare(conn, "UPDATE Contacts
     SET Contacts.ID = Contacts.ID + 1000
     WHERE Contacts.ID >= ?" )
rc, param = api.sqlany_describe_bind_param( stmt, 0 )
print "Param name = ", param.get_name(), "\n"
```

```
print "Param dir = ", param.get_direction(), "\n"
param.set_value(50)
rc = api.sqlany_bind_param( stmt, 0, param )
```

sqlany_clear_error

清除上次存储的错误代码。

语法

sqlany_clear_error (*$conn*)

参数

- **$conn** 从 sqlany_new_connection 返回的连接对象。

返回值

返回 nil。

另请参见

- “sqlany_new_connection”一节第 800 页

示例

```
api.sqlany_clear_error( conn )
```

sqlany_client_version

返回当前的客户端版本。

语法

sqlany_client_version ( )

返回值

返回标量值，即客户端版本字符串。

示例

```
buffer = api.sqlany_client_version()
```

sqlany_commit

提交当前事务。

语法

sqlany_commit (*$conn*)

参数

- **$conn** 要在其上执行提交操作的连接对象。

返回值

返回标量值，成功时为 1，不成功时为 0。

另请参见

- “sqlany_rollback”一节第 803 页

示例

```
rc = api.sqlany_commit( conn )
```

sqlany_connect

使用指定的连接对象和连接字符串创建与某个 SQL Anywhere 数据库服务器的连接。

语法

sqlany_connect (*$conn*, *$str*)

参数

- **$conn** 由 sqlany_new_connection 创建的连接对象。
- **$str** 一个 SQL Anywhere 连接字符串。

返回值

返回标量值，成功建立连接为 1，连接失败则为 0。使用 sqlany_error 检索错误代码和消息。

另请参见

- “sqlany_new_connection”一节第 800 页
- “sqlany_error”一节第 791 页
- “连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》
- “SQL Anywhere 数据库连接” 《SQL Anywhere 服务器 - 数据库管理》

示例

```
# Create a connection
conn = api.sqlany_new_connection()

# Establish a connection
status = api.sqlany_connect( conn, "UID=DBA;PWD=sql" )
print "Connection status = #{status}\n"
```

sqlany_describe_bind_param

描述预准备语句的绑定参数。

语法

sqlany_describe_bind_param (*$stmt*, *$index*)

参数

- **$stmt** 使用 sqlany_prepare 成功预准备的语句。
- **$index** 参数的索引。该数字必须在 0 到 sqlany_num_params() - 1 之间。

返回值

返回 2 个元素的数组，其中第一个参数中 1 表示成功，0 表示失败；第二个是描述性参数。

注释

此函数允许调用者确定有关预准备语句参数的信息。预准备语句（存储过程或 DML）的类型确定所提供的信息量。始终会提供参数的方向（输入、输出或输入-输出）。

另请参见

- “sqlany_bind_param”一节第 787 页
- “sqlany_prepare”一节第 802 页

示例

```
stmt = api.sqlany_prepare(conn, "UPDATE Contacts
     SET Contacts.ID = Contacts.ID + 1000
     WHERE Contacts.ID >= ?" )
rc, param = api.sqlany_describe_bind_param( stmt, 0 )
print "Param name = ", param.get_name(), "\n"
print "Param dir = ", param.get_direction(), "\n"
param.set_value(50)
rc = api.sqlany_bind_param( stmt, 0, param )
```

sqlany_disconnect

断开 SQL Anywhere 连接。回退所有未提交的事务。

语法

sqlany_disconnect (*$conn*)

参数

- **$conn** 已使用 sqlany_connect 建立连接的连接对象。

返回值

返回标量值，成功时为 1，不成功时为 0。

另请参见

- “sqlany_connect”一节第 789 页
- “sqlany_new_connection”一节第 800 页

示例

```
# Disconnect from the database
status = api.sqlany_disconnect( conn )
print "Disconnect status = #{status}\n"
```

sqlany_error

返回存储在连接对象中的上一错误代码和消息。

语法

sqlany_error (*$conn*)

参数

- **$conn** 从 sqlany_new_connection 返回的连接对象。

返回值

返回 2 个元素的数组，其中第一个参数表示 SQL 错误代码，第二个参数表示错误消息字符串。

对于错误代码，正值表示警告，负值表示错误，0 表示成功执行。

另请参见

- “sqlany_connect”一节第 789 页
- “按 SQLCODE 排序的 SQL Anywhere 错误消息”一节 《错误消息》

示例

```
code, msg = api.sqlany_error( conn )
print "Code=#{code} Message=#{msg}\n"
```

sqlany_execute

执行预准备语句。

语法

sqlany_execute (*$stmt*)

参数

- **$stmt** 使用 sqlany_prepare 成功预准备的语句。

返回值

返回标量值，成功时为 1，不成功时为 0。

注释

可使用 sqlany_num_cols 验证语句是否返回结果集。

另请参见

- “sqlany_prepare”一节第 802 页

示例

```
stmt = api.sqlany_prepare(conn, "UPDATE Contacts
     SET Contacts.ID = Contacts.ID + 1000
     WHERE Contacts.ID >= ?" )
rc, param = api.sqlany_describe_bind_param( stmt, 0 )
param.set_value(50)
rc = api.sqlany_bind_param( stmt, 0, param )
rc = api.sqlany_execute( stmt )
```

sqlany_execute_direct

执行由字符串参数指定的 SQL 语句。

语法

sqlany_execute_direct (*$conn*, *$sql*)

参数

- **$conn** 已使用 sqlany_connect 建立连接的连接对象。
- **$sql** SQL 字符串。SQL 字符串不应包含诸如 **?** 之类的参数。

返回值

返回语句对象或 nil（失败时）。

注释

如果要在一个步骤中准备和执行语句，则使用此函数。不要使用此函数执行带参数的 SQL 语句。

另请参见

- “sqlany_fetch_absolute”一节第 793 页
- “sqlany_fetch_next”一节第 794 页
- “sqlany_num_cols”一节第 800 页
- “sqlany_get_column”一节第 797 页

示例

```
stmt = api.sqlany_execute_direct( conn, "SELECT * FROM Employees" )
rc = api.sqlany_fetch_next( stmt )
rc, employeeID = api.sqlany_get_column( stmt, 0 )
rc, managerID  = api.sqlany_get_column( stmt, 1 )
rc, surname = api.sqlany_get_column( stmt, 2 )
rc, givenName = api.sqlany_get_column( stmt, 3 )
rc, departmentID = api.sqlany_get_column( stmt, 4 )
print employeeID, ",", managerID, ",",
     surname, ",", givenName, ",", departmentID, "\n"
```


sqlany_execute_immediate

立即执行指定的 SQL 语句，不返回结果集。这对不返回结果集的语句很有用。

语法

sqlany_execute_immediate (*$conn*, *$sql*)

参数

- **$conn** 已使用 sqlany_connect 建立连接的连接对象。
- **$sql** SQL 字符串。SQL 字符串不应包含诸如 **?** 之类的参数。

返回值

返回标量值，成功时为 1，不成功时为 0。

另请参见

- “sqlany_error”一节第 791 页

示例

```
rc = api.sqlany_execute_immediate(conn, "UPDATE Contacts
     SET Contacts.ID = Contacts.ID + 1000
     WHERE Contacts.ID >= 50" )
```

sqlany_fetch_absolute

将结果集中的当前行移动到指定的行编号处，然后读取该行的数据。

语法

sqlany_fetch_absolute (*$stmt*, *$row_num*)

参数

- **$stmt** 由 sqlany_execute 或 sqlany_execute_direct 执行的语句对象。
- **$row_num** 要读取的行编号。第一行为 1，最后一行为 -1。

返回值

返回标量值，成功时为 1，不成功时为 0。

另请参见

- “sqlany_error”一节第 791 页
- “sqlany_execute”一节第 791 页
- “sqlany_execute_direct”一节第 792 页
- “sqlany_fetch_next”一节第 794 页

示例

```
stmt = api.sqlany_execute_direct( conn, "SELECT * FROM Employees" )
# Fetch the second row
rc = api.sqlany_fetch_absolute( stmt, 2 )
rc, employeeID = api.sqlany_get_column( stmt, 0 )
rc, managerID  = api.sqlany_get_column( stmt, 1 )
rc, surname = api.sqlany_get_column( stmt, 2 )
rc, givenName = api.sqlany_get_column( stmt, 3 )
rc, departmentID = api.sqlany_get_column( stmt, 4 )
print employeeID, ",", managerID, ",",
     surname, ",", givenName, ",", departmentID, "\n"
```

sqlany_fetch_next

返回结果集的下一行。此函数首先前移行指针，然后读取新行的数据。

语法

sqlany_fetch_next (*$stmt*)

参数

- **$stmt** 由 sqlany_execute 或 sqlany_execute_direct 执行的语句对象。

返回值

返回标量值，成功时为 1，不成功时为 0。

另请参见

- “sqlany_error”一节第 791 页
- “sqlany_execute”一节第 791 页
- “sqlany_execute_direct”一节第 792 页
- “sqlany_fetch_absolute”一节第 793 页

示例

```
stmt = api.sqlany_execute_direct( conn, "SELECT * FROM Employees" )
# Fetch the second row
rc = api.sqlany_fetch_next( stmt )
rc, employeeID = api.sqlany_get_column( stmt, 0 )
rc, managerID  = api.sqlany_get_column( stmt, 1 )
rc, surname = api.sqlany_get_column( stmt, 2 )
rc, givenName = api.sqlany_get_column( stmt, 3 )
rc, departmentID = api.sqlany_get_column( stmt, 4 )
print employeeID, ",", managerID, ",",
     surname, ",", givenName, ",", departmentID, "\n"
```

sqlany_fini

释放由 API 分配的资源。

语法

sqlany_fini ( )

返回值

返回 nil。

另请参见

- “sqlany_init”一节第 799 页

示例

```
# Disconnect from the database
api.sqlany_disconnect( conn )

# Free the connection resources
api.sqlany_free_connection( conn )

# Free resources the api object uses
api.sqlany_fini()

# Close the interface
SQLAnywhere::API.sqlany_finalize_interface( api )
```

sqlany_free_connection

释放与连接对象关联的资源。

语法

sqlany_free_connection (*$conn*)

参数

- **$conn** 由 sqlany_new_connection 创建的连接对象。

返回值

返回 nil。

另请参见

- “sqlany_new_connection”一节第 800 页

示例

```
# Disconnect from the database
api.sqlany_disconnect( conn )

# Free the connection resources
api.sqlany_free_connection( conn )

# Free resources the api object uses
api.sqlany_fini()

# Close the interface
SQLAnywhere::API.sqlany_finalize_interface( api )
```

sqlany_free_stmt

释放与语句对象关联的资源。

语法

sqlany_free_stmt (*$stmt*)

参数

- **$stmt** 成功执行 sqlany_prepare 或 sqlany_execute_direct 所返回的语句对象。

返回值

返回 nil。

另请参见

- “sqlany_prepare”一节第 802 页
- “sqlany_execute_direct”一节第 792 页

示例

```
stmt = api.sqlany_prepare(conn, "UPDATE Contacts
     SET Contacts.ID = Contacts.ID + 1000
     WHERE Contacts.ID >= ?" )
rc, param = api.sqlany_describe_bind_param( stmt, 0 )
param.set_value(50)
rc = api.sqlany_bind_param( stmt, 0, param )
rc = api.sqlany_execute( stmt )
rc = api.sqlany_free_stmt( stmt )
```

sqlany_get_bind_param_info

检索使用 sqlany_bind_param 绑定的参数的相关信息。

语法

sqlany_get_bind_param_info (*$stmt*, *$index*)

参数

- **$stmt** 使用 sqlany_prepare 成功预准备的语句。
- **$index** 参数的索引。该数字必须在 0 到 sqlany_num_params() - 1 之间。

返回值

返回 2 个元素的数组，其中第一个参数中 1 表示成功，0 表示失败；第二个是描述性参数。

另请参见

- “sqlany_bind_param”一节第 787 页
- “sqlany_describe_bind_param”一节第 789 页
- “sqlany_prepare”一节第 802 页

示例

```
# Get information on first parameter (0)
rc, param_info = api.sqlany_get_bind_param_info( stmt, 0 )
print "Param_info direction = ", param_info.get_direction(), "\n"
print "Param_info output = ", param_info.get_output(), "\n"
```

sqlany_get_column

返回为指定列读取的值。

语法

sqlany_get_column (*$stmt*, *$col_index*)

参数

- **$stmt** 由 sqlany_execute 或 sqlany_execute_direct 执行的语句对象。
- **$col_index** 要检索的列编号。列编号在 0 到 sqlany_num_cols() - 1 之间。

返回值

返回 2 个元素的数组，其中第一个参数中 1 表示成功，0 表示失败；第二个是列值。

另请参见

- “sqlany_execute”一节第 791 页
- “sqlany_execute_direct”一节第 792 页
- “sqlany_fetch_absolute”一节第 793 页
- “sqlany_fetch_next”一节第 794 页

示例

```
stmt = api.sqlany_execute_direct( conn, "SELECT * FROM Employees" )
# Fetch the second row
rc = api.sqlany_fetch_next( stmt )
rc, employeeID = api.sqlany_get_column( stmt, 0 )
rc, managerID  = api.sqlany_get_column( stmt, 1 )
rc, surname = api.sqlany_get_column( stmt, 2 )
rc, givenName = api.sqlany_get_column( stmt, 3 )
rc, departmentID = api.sqlany_get_column( stmt, 4 )
print employeeID, ",", managerID, ",",
     surname, ",", givenName, ",", departmentID, "\n"
```

sqlany_get_column_info

获取指定结果集列的列信息。

语法

sqlany_get_column_info (*$stmt*, *$col_index*)

参数

- **$stmt** 由 sqlany_execute 或 sqlany_execute_direct 执行的语句对象。
- **$col_index** 列编号在 0 到 sqlany_num_cols() - 1 之间。

返回值

返回 9 个元素的数组，其中包含用于描述结果集中一列的信息。第一个元素包含 1（成功时）或 0（失败时）。数组元素在下表中进行介绍。

元素号	类型	说明
0	Integer	成功则为 1，失败则为 0。
1	Integer	列索引（0 到 sqlany_num_cols() - 1）。
2	String	列名。
3	Integer	列类型。请参见“列类型”一节第 803 页。
4	Integer	列本地类型。请参见“本地列类型”一节第 804 页。
5	Integer	列精度（用于数字类型）。
6	Integer	列小数位数（用于数字类型）。
7	Integer	列大小。
8	Integer	列可为空（1 = 可为空，0 = 不可为空）。

另请参见

- “sqlany_execute”一节第 791 页
- “sqlany_execute_direct”一节第 792 页
- “sqlany_prepare”一节第 802 页

示例

```
# Get column info for first column (0)
rc, col_num, col_name, col_type, col_native_type, col_precision, col_scale,
    col_size, col_nullable = api.sqlany_get_column_info( stmt, 0 )
```

sqlany_get_next_result

前进到多结果集查询中的下一结果集。

语法

sqlany_get_next_result (*$stmt*)

参数

- **$stmt** 由 sqlany_execute 或 sqlany_execute_direct 执行的语句对象。

返回值

返回标量值，成功时为 1，不成功时为 0。

另请参见

- “sqlany_execute”一节第 791 页
- “sqlany_execute_direct”一节第 792 页

示例

```
stmt = api.sqlany_prepare(conn, "call two_results()" )
rc = api.sqlany_execute( stmt )
# Fetch from first result set
rc = api.sqlany_fetch_absolute( stmt, 3 )
# Go to next result set
rc = api.sqlany_get_next_result( stmt )
# Fetch from second result set
rc = api.sqlany_fetch_absolute( stmt, 2 )
```

sqlany_init

初始化接口。

语法

sqlany_init ( )

返回值

返回 2 个元素的数组，其中第一个参数中 1 表示成功，0 表示失败，第二个是 Ruby 接口版本。

另请参见

- “sqlany_fini”一节第 794 页

示例

```
# Load the SQLAnywhere gem
begin
    require 'rubygems'
    gem 'sqlanywhere'
    unless defined? SQLAnywhere
        require 'sqlanywhere'
    end
end
# Create an interface
api = SQLAnywhere::SQLAnywhereInterface.new()
# Initialize the interface (loads the DLL/SO)
SQLAnywhere::API.sqlany_initialize_interface( api )
# Initialize our api object
api.sqlany_init()
```

sqlany_new_connection

创建连接对象。

语法

sqlany_new_connection ()

返回值

返回标量值，即连接对象。

注释

必须先创建连接对象，然后才能建立数据库连接。可以从连接对象中检索错误。一个连接中每次只能处理一个请求。

另请参见

- “sqlany_connect”一节第 789 页
- “sqlany_disconnect”一节第 790 页

示例

```
# Create a connection
conn = api.sqlany_new_connection()

# Establish a connection
status = api.sqlany_connect( conn, "UID=DBA;PWD=sql" )
print "Status=#{status}\n"
```

sqlany_num_cols

返回结果集中的列数。

语法

sqlany_num_cols (*$stmt*)

参数

- **$stmt** 由 sqlany_execute 或 sqlany_execute_direct 执行的语句对象。

返回值

返回标量值，即结果集中的列数或 -1（失败时）。

另请参见

- “sqlany_execute”一节第 791 页
- “sqlany_execute_direct”一节第 792 页
- “sqlany_prepare”一节第 802 页

示例

```
stmt = api.sqlany_execute_direct( conn, "SELECT * FROM Employees" )
# Get number of result set columns
num_cols = api.sqlany_num_cols( stmt )
```

sqlany_num_params

返回预期用于预准备语句的参数数目。

语法

sqlany_num_params (*$stmt*)

参数

- **$stmt** 成功执行 sqlany_prepare 所返回的语句对象。

返回值

返回标量值，即预准备语句中的参数数或 -1（失败时）。

另请参见

- “sqlany_prepare”一节第 802 页

示例

```
stmt = api.sqlany_prepare(conn, "UPDATE Contacts
      SET Contacts.ID = Contacts.ID + 1000
      WHERE Contacts.ID >= ?" )
num_params = api.sqlany_num_params( stmt )
```

sqlany_num_rows

返回结果集中的行数。

语法

sqlany_num_rows (*$stmt*)

参数

- **$stmt** 由 sqlany_execute 或 sqlany_execute_direct 执行的语句对象。

返回值

返回标量值，即结果集中的行数。如果行数是估计值，则返回负数，且估计值为所返回整数的绝对值。如果行数为准确值，则返回值为正。

注释

缺省情况下，此函数仅返回一个估计值。要返回准确计数，请在连接中设置 ROW_COUNTS 选项。有关详细信息，请参见“row_counts 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

只能针对返回多个结果集的语句中的第一个结果集返回结果集中的行数计数。如果使用 sqlany_get_next_result 移动到下一个结果集，sqlany_num_rows 将仍返回第一个结果集中的行数。

另请参见

- “sqlany_execute”一节第 791 页
- “sqlany_execute_direct”一节第 792 页

示例

```
stmt = api.sqlany_execute_direct( conn, "SELECT * FROM Employees" )
# Get number of rows in result set
num_rows = api.sqlany_num_rows( stmt )
```

sqlany_prepare

准备所提供的 SQL 字符串

语法

sqlany_prepare (*$conn*, *$sql*)

参数

- **$conn** 已使用 sqlany_connect 建立连接的连接对象。
- **$sql** 要准备的 SQL 语句。

返回值

返回标量值，即语句对象或 nil（失败时）。

注释

由 sqlany_execute 执行与该语句对象相关联的语句。可使用 sqlany_free_stmt 释放与语句对象关联的资源。

另请参见

- “sqlany_execute”一节第 791 页
- “sqlany_free_stmt”一节第 796 页
- “sqlany_num_params”一节第 801 页
- “sqlany_describe_bind_param”一节第 789 页
- “sqlany_bind_param”一节第 787 页

示例

```
stmt = api.sqlany_prepare(conn, "UPDATE Contacts
     SET Contacts.ID = Contacts.ID + 1000
     WHERE Contacts.ID >= ?" )
rc, param = api.sqlany_describe_bind_param( stmt, 0 )
param.set_value(50)
rc = api.sqlany_bind_param( stmt, 0, param )
rc = api.sqlany_execute( stmt )
```


sqlany_rollback

回退当前事务。

语法

sqlany_rollback (*$conn*)

参数

- **$conn** 要在其上执行回退操作的连接对象。

返回值

返回标量值，成功时为 1，不成功时为 0。

另请参见

- “sqlany_commit”一节第 788 页

示例

```
rc = api.sqlany_rollback( conn )
```

sqlany_sqlstate

检索当前的 SQL 状态。

语法

sqlany_sqlstate (*$conn*)

参数

- **$conn** 从 sqlany_new_connection 返回的连接对象。

返回值

返回标量值，即当前 5 个字符的 SQL 状态。

另请参见

- “sqlany_error”一节第 791 页
- “按 SQLSTATE 排序的 SQL Anywhere 错误消息”一节 《错误消息》

示例

```
sql_state = api.sqlany_sqlstate( conn )
```

列类型

下面的 Ruby 类定义了由一些 SQL Anywhere Ruby 函数返回的列类型。

```
class Types
    A_INVALID_TYPE = 0
    A_BINARY       = 1
    A_STRING       = 2
    A_DOUBLE       = 3
    A_VAL64        = 4
    A_UVAL64       = 5
    A_VAL32        = 6
    A_UVAL32       = 7
    A_VAL16        = 8
    A_UVAL16       = 9
    A_VAL8         = 10
    A_UVAL8        = 11
end
```

本地列类型

下表定义了由一些 SQL Anywhere 函数返回的本地列类型。

本地类型值	本地类型
384	DT_DATE
388	DT_TIME
390	DT_TIMESTAMP_STRUCT
392	DT_TIMESTAMP
448	DT_VARCHAR
452	DT_FIXCHAR
456	DT_LONGVARCHAR
460	DT_STRING
480	DT_DOUBLE
482	DT_FLOAT
484	DT_DECIMAL
496	DT_INT
500	DT_SMALLINT
524	DT_BINARY
528	DT_LONGBINARY
600	DT_VARIABLE

本地类型值	本地类型
604	DT_TINYINT
608	DT_BIGINT
612	DT_UNSINT
616	DT_UNSSMALLINT
620	DT_UNSBIGINT
624	DT_BIT
628	DT_NSTRING
632	DT_NFIXCHAR
636	DT_NVARCHAR
640	DT_LONGNVARCHAR

Sybase Open Client API

目录

Open Client 体系结构

注意
本章介绍面向 SQL Anywhere 的 Sybase Open Client 编程接口。Sybase Open Client 应用程序开发的主要文档为 Open Client 文档，您可从 Sybase 获得。本章介绍的是专门面向 SQL Anywhere 的功能，而不是 Sybase Open Client 应用程序编程的详尽指南。

Sybase Open Client 具有两种组件：编程接口和网络服务。

DB-Library 和 Client Library。

Sybase Open Client 提供了两个核心编程接口以供编写客户端应用程序时使用：DB-Library 和 Client-Library。

Open Client DB-Library 为早期版本的 Open Client 应用程序提供支持，是与 Client-Library 完全分开的编程接口。Sybase Open Client 产品附带的 *Open Client DB-Library/C 参考手册*中对 DB-Library 进行了介绍。

Client-Library 程序还依赖于 CS-Library，在 Client-Library 和 Server-Library 应用程序中都使用 CS-Library 提供的例程。Client-Library 应用程序还可以使用 Bulk-Library 中的例程来促进数据的高速传输。

CS-Library 和 Bulk-Library 都包含在 Sybase Open Client 中，两者分开使用。

网络服务

Open Client 网络服务包括 Sybase Net-Library，Sybase Net-Library 为特定网络协议（如 TCP/IP 和 DECnet）提供支持。Net-Library 接口对于应用程序编程人员来说是不可见的。但在某些平台上，针对不同的系统网络配置，应用程序可能需要另外一种 Net-Library 驱动程序。根据您的主机平台，Net-Library 驱动程序由系统的 Sybase 配置指定，或者在您编译和链接程序时指定。

有关驱动程序配置的说明，请参见 *Open Client/Server 配置指南*。

有关构建 Client-Library 程序的说明，请参见 *Open Client/Server 程序员补充材料*。

建立 Open Client 应用程序的要求

要运行 Open Client 应用程序，则必须在运行该应用程序的计算机上安装和配置 Sybase Open Client 组件。这些组件可以作为您安装的其它 Sybase 产品的一部分进行安装，您也可以随 SQL Anywhere 一同安装这些库，这取决于许可协议条款的规定。

Open Client 应用程序不需要在运行数据库服务器的计算机上有任何 Open Client 组件。

要生成 Open Client 应用程序，您需要 Open Client 的开发版本，这可由 Sybase 提供。

缺省情况下，SQL Anywhere 数据库创建后不区分大小写，而 Adaptive Server Enterprise 数据库区分大小写。

有关与 SQL Anywhere 一同运行 Open Client 应用程序的详细信息，请参见“将 SQL Anywhere 用作 Open Server” 《SQL Anywhere 服务器 - 数据库管理》。

数据类型映射

Open Client 有其自己的内部数据类型，它们在某些细节上与 SQL Anywhere 中提供的数据类型有所区别。出于这个原因，SQL Anywhere 会在 Open Client 应用程序使用的数据类型和 SQL Anywhere 中的数据类型之间对某些数据类型进行内部映射。

要构建 Open Client 应用程序，您需要 Open Client 的开发版本。要使用 Open Client 应用程序，则必须在运行它的计算机上安装和配置 Open Client 运行时版本。

SQL Anywhere 服务器不需要任何外部通信运行时环境即可支持 Open Client 应用程序。

每个 Open Client 数据类型都映射到 SQL Anywhere 中对等的数据类型。所有的 Open Client 数据类型均受支持

在 Open Client 中没有直接对等项的 SQL Anywhere 数据类型

下表列出了在 SQL Anywhere 中受支持但在 Open Client 中没有直接对等项的数据类型的映射关系。

SQL Anywhere 数据类型	Open Client 数据类型
unsigned short	int
unsigned int	bigint
unsigned bigint	numeric(20.0)
date	smalldatetime
time	smalldatetime
string	varchar
timestamp	datetime

数据类型映射中的范围限制

某些数据类型在 SQL Anywhere 中与在 Open Client 中的范围不同。这种情况下，在检索或插入数据过程中可能会发生溢出错误。

下表列出了可映射到 SQL Anywhere 数据类型但对可能值范围具有某些限制的 Open Client 应用程序数据类型。

在多数情况下，Open Client 数据类型会映射到可能值范围更大的 SQL Anywhere 数据类型。因此，可能会出现这种情况：向 SQL Anywhere 传递的值被接受并存储在数据库中，而这个值由于过大而无法被 Open Client 应用程序读取。

数据类型	Open Client 下限	Open Client 上限	SQL Anywhere 可能值范围下限	SQL Anywhere 可能值范围上限
MONEY	-922 377 203 685 477.5808	922 377 203 685 477.5807	-1e15 + 0.0001	1e15 - 0.0001
SMALLMONEY	-214 748.3648	214 748.3647	-214 748.3648	214 748.3647
DATETIME	Jan 1, 1753	Dec 31, 9999	Jan 1, 0001	Dec 31, 9999
SMALLDATETIME	Jan 1, 1900	June 6, 2079	March 1, 1600	Dec 31, 7910

示例

例如，Open Client MONEY 和 SMALLMONEY 数据类型不会跨越它们底层的 SQL Anywhere 实现的整个数值范围。因此，SQL Anywhere 列中的值有可能会超出 Open Client 数据类型 MONEY 的界限。当客户端通过 SQL Anywhere 读取这种违规值时，就会产生错误。

时间戳

当 Open Client TIMESTAMP 值在 SQL Anywhere 中传递时，该数据类型的 SQL Anywhere 实现与它的 Adaptive Server Enterprise 实现不同。在 SQL Anywhere 中，该值被映射到 SQL Anywhere DATETIME 数据类型。在 SQL Anywhere 中，缺省值是 NULL，而且不保证其唯一性。而 Adaptive Server Enterprise 可确保该值会单调递增，从而确保其唯一性。

相比之下，SQL Anywhere TIMESTAMP 数据类型包含年、月、日、小时、分钟、秒和毫秒信息。另外，DATETIME 数据类型的可能值范围要大于由 SQL Anywhere 映射来的 Open Client 数据类型的可能值范围。

在 Open Client 应用程序中使用 SQL

本节简要介绍了在 Open Client 应用程序中使用 SQL 的相关信息，特别重点说明了专门针对 SQL Anywhere 的问题。

有关各概念的介绍，请参见“在应用程序中使用 SQL”第 23 页。有关完整说明，请参见您的 Open Client 文档。

执行 SQL 语句

可通过将 SQL 语句包含在 Client Library 函数调用中将其发送到数据库。例如，下面的两个调用将执行 DELETE 语句：

```
ret = ct_command(cmd, CS_LANG_CMD,
                  "DELETE FROM Employees
                   WHERE EmployeeID=105"
                 CS_NULLTERM,
                 CS_UNUSED);
ret = ct_send(cmd);
```

有关 Open Client 函数的详细信息，请参见 Open Client 15.0 Client-Library/C 参考手册。

使用预准备语句

ct_dynamic 函数用于管理预准备语句。此函数采用 *type* 参数描述您要执行的操作。

♦ **在 Open Client 中使用预准备语句**

1. 使用带 CS_PREPARE *type* 参数的 ct_dynamic 函数准备语句。
2. 使用 ct_param 函数设置语句参数。
3. 使用带 CS_EXECUTE *type* 参数的 ct_dynamic 函数执行语句。
4. 使用带 CS_DEALLOC *type* 参数的 ct_dynamic 函数释放与该语句关联的资源。

有关在 Open Client 中使用预准备语句的详细信息，请参见您的 Open Client 文档。

使用游标

ct_cursor 函数用于管理游标。此函数采用 *type* 参数描述您要执行的操作。

支持的游标类型

并非 SQL Anywhere 支持的所有游标类型都能通过 Open Client 接口使用。您不能通过 Open Client 使用滚动游标、动态滚动游标或不敏感游标。

唯一性和可更新性是游标的两个属性。游标可以唯一（每个行都带有主键或唯一性信息，无论应用程序是否使用它），也可以不唯一。游标可以是只读游标，也可以是可更新游标。如果游标是可更新游标且不唯一，则性能可能会受到影响，因为在这种情况下，无论 CS_CURSOR_ROWS 设置是什么，都不会对任何行执行预取。

使用游标的步骤

与其它某些接口不同（例如嵌入式 SQL），Open Client 会将游标与一个以字符串表示的 SQL 语句相关联。嵌入式 SQL 首先准备一个语句，然后游标使用该语句句柄进行声明。

♦ 在 Open Client 中使用游标

1. 要在 Open Client 中声明游标，请使用将 CS_CURSOR_DECLARE 作为 *type* 参数的 ct_cursor。
2. 声明游标之后，可以使用将 CS_CURSOR_ROWS 作为 *type* 参数的 ct_cursor 函数来控制每次从服务器读取一行时要预取多少行到客户端。

   在客户端存储预取的行会减少对服务器的调用，从而提高总体吞吐量，同时缩短周转时间。预取的行不会立即传递到应用程序，而是会存储在客户端的缓冲区内备用。

   PREFETCH 数据库选项的设置控制其它接口的行预取。Open Client 连接会忽略该设置。对于非唯一的可更新游标，将忽略 CS_CURSOR_ROWS 设置。
3. 要在 Open Client 中打开游标，请使用将 CS_CURSOR_OPEN 作为 *type* 参数的 ct_cursor 函数。
4. 使用 ct_fetch 函数可将每行读取到应用程序中。
5. 要关闭游标，请使用带 CS_CURSOR_CLOSE 的 ct_cursor。
6. 在 Open Client 中，您还需要释放与游标相关联的资源。使用带 CS_CURSOR_DEALLOC 的 ct_cursor 函数可以完成此任务。您还可以使用带附加参数 CS_DEALLOC 的 CS_CURSOR_CLOSE 通过单个步骤执行这些操作。

通过游标修改行

借助 Open Client，只要游标针对的是单个表，您就可以在游标中删除或更新行。用户必须具有更新表的权限，且游标必须标记为可更新。

♦ 通过游标修改行

- 您可以不执行读取，而是分别使用带 CS_CURSOR_DELETE 或 CS_CURSOR_UPDATE 的 ct_cursor 来删除或更新游标的当前行。

在 Open Client 应用程序中不能通过游标插入行。

在 Open Client 中描述查询结果

Open Client 处理结果集的方式与其它某些 SQL Anywhere 接口不同。

在嵌入式 SQL 和 ODBC 中，**描述**查询或存储过程是为了设置正确数目和类型的变量来接收结果。描述是对语句其本身执行。

在 Open Client 中，您不需要描述语句。从服务器返回的每个行都可以带有对其内容的描述。如果使用 ct_command 函数和 ct_send 函数执行语句，则可以使用 ct_results 函数来处理查询中所返回行的各个方面。

如果不想用这种逐行方式来处理结果集，则可以使用 ct_dynamic 函数来准备 SQL 语句，并使用 ct_describe 函数来描述其结果集。这种方法更接近于其它接口中对 SQL 语句的描述方式。

SQL Anywhere 的已知 Open Client 限制

通过 Open Client 接口，您基本上就可以像使用 Adaptive Server Enterprise 数据库那样使用 SQL Anywhere 数据库。但还是有一些限制，其中包括以下方面：

- SQL Anywhere 不支持 Adaptive Server Enterprise 提交服务。
- 客户端/服务器连接的**功能**决定了该连接所允许的客户端请求和服务器响应的类型。不支持以下功能：
  - CS_CSR_ABS
  - CS_CSR_FIRST
  - CS_CSR_LAST
  - CS_CSR_PREV
  - CS_CSR_REL
  - CS_DATA_BOUNDARY
  - CS_DATA_SENSITIVITY
  - CS_OPT_FORMATONLY
  - CS_PROTO_DYNPROC
  - CS_REG_NOTIF
  - CS_REQ_BCP
- 不支持安全性组件，例如 SSL。但是，支持口令加密。
- Open Client 应用程序可使用 TCP/IP 连接到 SQL Anywhere。

  有关功能的详细信息，请参见 *Open Server Server-Library C 参考手册*。
- 如果将 CS_DATAFMT 与 CS_DESCRIBE_INPUT 一起使用，在某个参数化变量作为输入发送到 SQL Anywhere 时，CS_DATAFMT 不会返回列的数据类型。

SQL Anywhere Web 服务

目录

Web 服务简介

SQL Anywhere 包含的内置 HTTP 服务器允许您提供 Web 服务，并访问其它 SQL Anywhere 数据库中的 Web 服务以及通过 Internet 提供的标准 Web 服务。SOAP 是针对这方面用途所使用的标准，但 SQL Anywhere 中的内置 HTTP 服务器也使您可以处理来自客户端应用程序的标准 HTTP 和 HTTPS 请求。

Web 服务一词包涵了多方面含义。通常，它指的是能促进计算机间数据传送和互操作性的软件。实质上，Web 服务通过 Internet 提供业务逻辑段。**简单对象访问协议**（Simple Object Access Protocol，简称 SOAP）是一个基于 XML 的简单协议，它允许各应用程序通过 HTTP 交换信息。

SOAP 提供了一种利用 Internet 在应用程序（例如用 Java 或 Microsoft .NET 语言 [如 Visual C#] 编写的那些应用程序）之间进行通信的方法。SOAP 消息定义服务器提供的服务。通常，使用 HTTP 进行实际的数据传送来实现 XML 文档交换，这些 XML 文档的结构应能够有效地对相关信息进行编码。参与 SOAP 通信的任何应用程序（如客户端或服务器）都称为 SOAP 节点或 SOAP 端点。此类应用程序可传输、接收或处理 SOAP 消息。可以使用 SQL Anywhere 创建 SOAP 节点。

有关 SOAP 标准的详细信息，请参见 http://www.w3.org/TR/2000/NOTE-SOAP-20000508/。

Web 服务和 SQL Anywhere

在 SQL Anywhere 环境中，Web 服务一词表示 SQL Anywhere 能够监听和处理标准 SOAP 请求。SQL Anywhere 中的 Web 服务为客户端应用程序提供了除 JDBC 和 ODBC 这样的传统接口之外的替代接口。可以从多种语言编写的并在多种平台上运行的客户端应用程序访问 Web 服务。即使是 Perl 和 Python 这样的常见脚本编写语言也可提供对 Web 服务的访问。可使用 CREATE SERVICE 语句在数据库中创建 Web 服务。

SQL Anywhere 还可用作 SOAP 或 HTTP 客户端，允许数据库内运行的应用程序访问通过 Internet 提供的或其它 SQL Anywhere 数据库提供的标准 Web 服务。此客户端功能通过存储函数和过程来访问。

此外，Web 服务一词还指使用内置 Web 服务器处理来自客户端的 HTTP 请求的应用程序。这些应用程序的工作方式通常类似于传统数据库支持的 Web 应用程序，但可能更紧凑和更容易编写，因为数据和整个应用程序都可驻留在一个数据库内。在此类应用程序中，Web 服务通常返回 HTML 格式的文档。支持 GET、HEAD 和 POST 方法。

数据库内 Web 服务的集合共同定义了可用的 URL。每种服务都提供一组 Web 页。通常，这些 Web 页的内容由您编写并存储在数据库中的过程生成，然而这些过程可以是单个语句，或者允许用户执行其自己的语句。如果使用可使数据库服务器监听 HTTP 请求的选项启动数据库服务器，便可以使用这些 Web 服务。

因为处理 Web 服务请求的 HTTP 服务器嵌入在数据库中，所以可保证良好的性能。使用 Web 服务的应用程序很容易部署，因为除数据库和数据库服务器之外无需其它任何组件。

Web 服务快速入门

以下过程介绍了如何新建一个数据库、如何在启用 HTTP 服务器的情况下启动 SQL Anywhere 数据库，以及如何使用任何常用 Web 浏览器访问此数据库。

♦ 创建和访问简单 HTML Web 服务

1. 在命令提示符处，运行以下命令以启动个人 Web 服务器。将 *samples-dir* 替换为示例数据库的实际位置。`-xs http(port=80)` 选项将指示数据库服务器监听 HTTP 请求。如果端口 80 上已有 Web 服务器在运行，则将另一个端口号（例如 8080）用于此演示。

   ```
   dbeng11 -xs http(port=80) samples-dir\demo.db
   ```

   HTTP 通信链接的很多属性都由 -xs 选项的参数来控制。请参见“-xs 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

2. 使用相应的 -xs 选项参数启动数据库服务器，并使用 CREATE SERVICE 语句创建 Web 服务以响应进来的请求。

   启动 Interactive SQL。以 DBA 身份连接到 SQL Anywhere 示例数据库。执行以下语句。

   ```
   CREATE SERVICE HTMLtable
   TYPE 'HTML'
   AUTHORIZATION OFF
   USER DBA
   AS SELECT * FROM Customers;
   ```

   此语句创建一个名为 HTMLtable 的 Web 服务。这个简单的 Web 服务会返回语句 SELECT * FROM Customers 的结果，并将输出自动转换为 HTML 格式。因为关闭了授权，所以从 Web 浏览器访问表无需任何权限。请参见“创建 Web 服务”一节第 823 页和“CREATE SERVICE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

3. 启动 Web 浏览器。

4. 浏览至 URL http://localhost:80/demo/HTMLtable。使用启动数据库服务器时指定的端口号。

   Web 浏览器将显示数据库服务器返回的 HTML 文档的主体。缺省情况下，会将结果集的格式设置为 HTML 表。

♦ 创建和访问简单的 XML Web 服务

1. 在命令提示符处，运行以下命令以启动个人 Web 服务器。将 *samples-dir* 替换为示例数据库的实际位置。`-xs http(port=80)` 选项将指示数据库服务器监听 HTTP 请求。如果端口 80 上已有 Web 服务器在运行，则将另一个端口号（例如 8080）用于此演示。

   ```
   dbeng11 -xs http(port=80) samples-dir\demo.db
   ```

   HTTP 通信链接的很多属性都由 -xs 选项的参数来控制。请参见“-xs 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

2. 使用相应的 -xs 选项参数启动数据库服务器，并使用 CREATE SERVICE 语句创建 Web 服务以响应进来的请求。

   启动 Interactive SQL。以 DBA 身份连接到 SQL Anywhere 示例数据库。执行以下语句：

```
CREATE SERVICE XMLtable
TYPE 'XML'
AUTHORIZATION OFF
USER DBA
AS SELECT * FROM Customers;
```

此语句创建一个名为 XMLtable 的 Web 服务。这个简单的 Web 服务会返回语句 SELECT * FROM Customers 的结果，并将输出自动转换为 XML 格式。因为关闭了授权，所以从 Web 浏览器访问表无需任何权限。请参见“创建 Web 服务”一节第 823 页和“CREATE SERVICE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

3. 启动 Web 浏览器。
4. 浏览至 URL http://localhost:80/demo/XMLtable。使用启动数据库服务器时指定的端口号。
   - **localhost:80** 定义要使用的 Web 主机名和端口号。
   - **demo** 定义要使用的数据库名。您使用的是 *demo.db*。
   - **XMLtable** 定义要使用的服务名。

   Web 浏览器将显示数据库服务器返回的 XML 文档的主体。由于未包含任何格式设置信息，您将看到原始 XML，包括标记和属性。
5. 也可以从常用编程语言访问 XMLtable 服务。例如，以下 C# 短程序使用 XMLtable Web 服务：

```
using System.Xml;

static void Main(string[] args)
{
  XmlTextReader reader =
    new XmlTextReader( "http://localhost:80/demo/XMLtable" );

  while( reader.Read() )
  {
    switch( reader.NodeType )
    {
    case XmlNodeType.Element:
      if( reader.Name == "row" )
      {
        Console.Write(reader.GetAttribute("ID")+" ");
        Console.WriteLine(reader.GetAttribute("Surname"));
      }
      break;
    }
  }
  reader.Close();
}
```

6. 此外，如以下示例所示，还可从 Python 访问同一 Web 服务：

```
import xml.sax

class DocHandler( xml.sax.ContentHandler ):
  def startElement( self, name, attrs ):
    if name == 'row':
      table_id = attrs.getValue( 'ID' )
      table_name = attrs.getValue( 'Surname' )
      print '%s %s' % ( table_id, table_name )

parser = xml.sax.make_parser()
```



```
parser.setContentHandler( DocHandler() )
parser.parse( 'http://localhost:80/demo/XMLtable' )
```

将此代码保存在名为 *DocHandler.py* 的文件中。要运行该应用程序，请输入类似以下的命令：

```
python DocHandler.py
```

♦ 创建和访问简单的 JSON web 服务

1. 在命令提示符处，运行以下命令以启动个人 Web 服务器。将 *samples-dir* 替换为示例数据库的实际位置。`-xs http(port=80)` 选项将指示数据库服务器监听 HTTP 请求。如果端口 80 上已有 Web 服务器在运行，则将另一个端口号（例如 8080）用于此演示。

   ```
   dbeng11 -xs http(port=80) samples-dir\demo.db
   ```

   HTTP 通信链接的很多属性都由 -xs 选项的参数来控制。请参见“-xs 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

2. 使用相应的 -xs 选项参数启动数据库服务器，并使用 CREATE SERVICE 语句创建 Web 服务以响应进来的请求。

   启动 Interactive SQL。以 DBA 身份连接到 SQL Anywhere 示例数据库。执行以下语句：

   ```
   CREATE SERVICE JSONtable
   TYPE 'JSON'
   AUTHORIZATION OFF
   USER DBA
   AS SELECT * FROM Customers;
   ```

   此语句创建一个名为 JSONtable 的 Web 服务。这个简单的 Web 服务会返回语句 SELECT * FROM Customers 的结果，并将输出自动转换为 JavaScript Object Notation 格式。因为关闭了授权，所以从 Web 浏览器访问表无需任何权限。请参见“创建 Web 服务”一节第 823 页和“CREATE SERVICE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

3. 启动 Web 浏览器。

4. 浏览至 URL http://localhost:80/demo/JSONtable。使用启动数据库服务器时指定的端口号。

   - **localhost:80** 定义要使用的 Web 主机名和端口号。
   - **demo** 定义要使用的数据库名。您使用的是 *demo.db*。
   - **JSONtable** 定义要使用的服务名。

   Web 浏览器将允许您保存数据库服务器返回的 JSON 响应文档。将响应保存到文件中。

5. 如果您使用文本编辑器来查看包含响应文档的文件，您将在结果集中看到以下的数组符号。

   ```
   [
           {
                   "ID": 101,
                   "Surname": "Devlin",
                   "GivenName": "Michaels",
                   "Street": "114 Pioneer Avenue",
                   "City": "Kingston",
                   "State": "NJ",
                   "Country": "USA",
                   "PostalCode": "07070",
                   "Phone": "2015558966",
   ```

```
                "CompanyName": "The Power Group"
        },
        {
                "ID": 102,
                "Surname": "Reiser",
                "GivenName": "Beth",
                "Street": "33 Whippany Road",
                "City": "Rockwood",
                "State": "NY",
                "Country": "USA",
                "PostalCode": "10154",
                "Phone": "2125558725",
                "CompanyName": "AMF Corp."
        },
.
.
.
        {
                "ID": 665,
                "Surname": "Thompson",
                "GivenName": "William",
                "Street": "19 Washington Street",
                "City": "Bancroft",
                "State": "NY",
                "Country": "USA",
                "PostalCode": "11700",
                "Phone": "5165552549",
                "CompanyName": "The Apple Farm"
        }
]
```

其它入门资源

示例包含在 *samples-dir\SQLAnywhere\HTTP* 目录中。

CodeXchange 上提供了其它示例，网址是 http://www.sybase.com/developer/codexchange。

另请参见

- “HTML_DECODE 函数 [Miscellaneous]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “HTML_ENCODE 函数 [Miscellaneous]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “HTTP_DECODE 函数 [HTTP]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “HTTP_ENCODE 函数 [HTTP]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “HTTP_HEADER 函数 [HTTP]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “HTTP_VARIABLE 函数 [HTTP]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “NEXT_HTTP_HEADER 函数 [HTTP]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “NEXT_HTTP_VARIABLE 函数 [HTTP]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “NEXT_SOAP_HEADER 函数 [SOAP]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “SOAP_HEADER 函数 [SOAP]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “sa_http_header_info 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “sa_http_variable_info 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “sa_set_http_header 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “sa_set_http_option 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “sa_set_soap_header 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》

创建 Web 服务

创建并存储于数据库中的 Web 服务定义了哪些 URL 有效以及它们的作用。单个数据库可以定义多个 Web 服务。可以在不同的数据库中定义 Web 服务，以使这些 Web 服务看起来像是单个 Web 站点的一部分。

以下语句允许您创建、变更和删除 Web 服务：

- CREATE SERVICE
- ALTER SERVICE
- DROP SERVICE
- COMMENT ON SERVICE

CREATE SERVICE 语句的一般语法如下：

CREATE SERVICE *service-name* **TYPE '***service-type***'** [*attributes*] [**AS** *statement*]

服务名

由于服务名构成了用于访问服务的 URL 的一部分，因此它们在允许包含什么字符方面有很大的灵活性。除了标准的字母数字字符外，还允许使用以下字符： - _ . ! * '( )

此外，除了那些用于命名 DISH 服务的服务名，其它服务名可以包含斜线 "/"，但有一些限制，因为此字符是标准 URL 分隔符，会影响 SQL Anywhere 对 URL 的解释。斜线不能用作服务名的第一个字符。此外，服务名不能包含两个连续的斜线。

服务名中允许使用的字符在仅适用于 DISH 服务的 GROUP 名称中也允许使用。

服务类型

支持以下服务类型：

- **'SOAP'** 结果集作为 SOAP 响应返回。数据格式由 FORMAT 子句决定。对 SOAP 服务的请求必须是有效 SOAP 请求，而不仅是简单的 HTTP 请求。
- **'DISH'** DISH 服务 (Determine SOAP Handler) 充当 GROUP 子句标识的 SOAP 服务的代理，并为每个此类 SOAP 服务都生成一个 WSDL（Web 服务描述语言）文档。
- **'HTML'** 将语句或过程的结果集自动设置为包含表的 HTML 文档格式。
- **'XML'** 结果集以 XML 格式返回。如果结果集已是 XML 格式，则不应用其它任何格式设置。如果它还不是 XML 格式，则将其格式自动设置为 XML。其作用与在 SELECT 语句中使用 FOR XML RAW 子句的作用类似。
- **'JSON'** 结果集以 JavaScript Object Notation (JSON) 格式返回。JSON 与 XML 的结构相似，但 JSON 的结构更精简。有关 JSON 的详细信息，请访问 http://www.json.org。
- **'RAW'** 不对结果集另外进行任何格式设置就将其发送到客户端。可以通过在过程中显式生成所需的标记来生成格式化文档。

在所有服务类型中，RAW 可使您最大程度地控制输出。但它的确需要您执行更多操作，因为您必须显式输出所有必要的标记。可通过向服务的语句应用 FOR XML 子句来调整 XML 服务的输出。SOAP 服务的输出可使用 CREATE 或 ALTER SERVICE 语句的 FORMAT 属性进行调整。请参见“CREATE SERVICE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

语句

语句是当某人访问服务时调用的命令（通常是存储过程）。如果您定义了语句，则这是唯一可以通过此服务运行的语句。该语句是 SOAP 服务所必需的，而 DISH 服务将忽略该语句。缺省值是 NULL，表示没有语句。

可以创建不含语句的服务。语句从 URL 中获取。在测试服务或者需要一种通用的信息访问方式时，以这种方式配置的服务会很有用。为此，可以完全省略该语句，或者使用短语 AS NULL 替换该语句。

未定义语句的服务会带来严重的安全风险，因为这类服务允许 Web 客户端执行任意命令。创建此类服务时，必须启用授权，这会强制所有客户端都提供有效的用户名和口令。虽然如此，但生产系统中仍应该只运行定义了语句的服务。

属性

通常，所有属性都是可选的。但有些属性是互相依赖的。可以使用以下属性：

- **AUTHORIZATION** 此属性控制哪些用户可以使用该服务。缺省设置为 ON。如果未提供任何语句，则必须将 AUTHORIZATION 设为 ON。此外，授权设置还影响对用户名（由 USER 属性定义）的解释。
- **SECURE** 当设置为 ON 时，只允许安全连接。HTTP 端口上接收的所有连接都将自动重定向到 HTTPS 端口。缺省值为 OFF，即同时启用 HTTP 和 HTTPS 请求，前提是在启动数据库服务器时使用相应选项启用了这些端口。请参见“-xs 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。
- **USER** USER 子句控制可以使用哪些数据库用户帐户处理服务请求。但对此设置的解释取决于 AUTHORIZATION 是 ON 还是 OFF。

  当 AUTHORIZATION 设置为 ON 时，所有客户端在连接时都必须提供有效的用户名和口令。当 AUTHORIZATION 设置为 ON 时，USER 选项可以为 NULL、数据库用户名或数据库组的名称。如果为 NULL，则任何数据库用户都能连接并发出请求。请求使用该用户的帐户和权限运行。如果指定了组名，则只有属于该组的用户可以运行请求。其他所有数据库用户都被拒绝使用该服务。

  如果 AUTHORIZATION 设置为 OFF，则必须提供一条语句。此外，还必须提供用户名。所有请求都使用该用户的帐户和权限运行。这样，如果服务器连接到公共网络，则应将指定用户帐户的权限设置到最小程度，以限制由恶意使用造成的损失。
- **GROUP** GROUP 子句仅适用于 DISH 服务，用于确定 DISH 服务公开哪些 SOAP 服务。DISH 服务仅公开其名称以该 DISH 服务的组名开头的 SOAP 服务。因此，组名是公开的 SOAP 服务的公用前缀。例如，指定 GROUP *xyz* 则仅公开 SOAP 服务 *xyz/aaaa*、*xyz/bbbb* 或 *xyz/cccc*，而不公开 *abc/aaaa* 或 *xyzaaaa*。如果没有指定组名，则 DISH 服务公开数据库中的所有 SOAP 服务。组名和服务名中允许使用相同的字符。

  SOAP 服务可由多个 DISH 服务公开。特别是，此特性允许单个 SOAP 服务以多种格式提供数据。除非在 SOAP 服务中指定，否则服务类型将从 DISH 服务继承。因此，可以创建不声明格

式类型的 SOAP 服务，然后将其包含在多个 DISH 服务中，每个 DISH 服务指定一个不同的格式。

- **FORMAT** FORMAT 子句仅适用于 DISH 和 SOAP 服务，用于控制 SOAP 或 DISH 响应的输出格式。可以使用与各种类型的 SOAP 客户端（如 .NET 或 JAX-WS）兼容的输出格式。如果未指定 SOAP 服务的格式，则从该服务的 DISH 服务声明继承格式。如果 DISH 服务也没有声明格式，则格式缺省设置为与 .NET 客户端兼容的 DNET。可以通过定义多个 DISH 服务（每个都使用一个不同的 FORMAT 类型）将没有声明格式的 SOAP 服务与不同类型的 SOAP 客户端同用。
- **URL [PATH]** URL 或 URL PATH 子句控制对 URL 的解释，并仅适用于 XML、HTML 和 RAW 服务类型。特别是，它决定了是否接受 URL 路径以及在接受的情况下如何处理 URL 路径。如果服务名以字符 "/" 结尾，则必须将 URL 设置为 OFF。请参见“CREATE SERVICE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

启动监听 Web 请求的数据库服务器

如果要让数据库服务器通过 HTTP 或 HTTPS 监听 Web 服务请求，则必须在启动服务器时在命令行上指定要监听的 Web 请求的类型。缺省情况下，数据库服务器不监听 Web 服务请求，这样就使客户端无法访问数据库中可能定义的任何服务。

还可以在命令行上指定 HTTP 或 HTTPS 服务的各种属性，例如在哪个端口上进行监听。

还必须在数据库中创建 Web 服务。请参见“创建 Web 服务”一节第 823 页。

可使用 -xs 选项启用协议。HTTP 和 HTTPS 是两种可用的 Web 服务协议。协议名后面的括号中的可选参数允许您自定义对每种 Web 服务的访问。

此选项的一般语法如下：

-xs { *protocol* [**(***option***=***value*; ...**)**], ... }

启动多个 Web 服务器
如果想要同时启动多个 Web 服务器，则必须更改其它 Web 服务器的端口，因为它们的缺省端口都相同。

协议

可以选择以下 Web 服务协议值：

- **http** 监听 HTTP 连接。
- **https** 监听 HTTPS 连接。支持使用 SSL 3.0 版和 TLS 1.0 版的 HTTPS 连接。
- **none** 不监听 Web 服务请求。这是缺省设置。

选项

以下是一些可用选项：

- **FIPS** 指定 **FIPS=Y** 可监听 HTTPS FIPS 连接。
- **ServerPort [PORT]** 监听 Web 请求所使用的端口。缺省情况下，SQL Anywhere 在端口 80 上监听 HTTP 请求，在端口 443 上监听安全 HTTP (HTTPS) 请求。FIPS 认可的 HTTPS 连接的缺省端口与 HTTPS 的相同。

  例如，如果端口 80 上已有 Web 服务器在运行，则可以使用以下选项启动一个在端口 8080 上监听 Web 请求的数据库服务器：

```
dbeng11 mywebapp.db -xs http(port=8080)
```

  作为另一个实例，以下命令使用 SQL Anywhere 随附的示例标识文件（您必须安装了 RSA 或 FIPS 认可的 RSA 加密才能拥有此文件）来启动安全 Web 服务器。此命令应在单独的一行中输入。

```
dbeng11 -xs https(identity=rsaserver.id;
                  identity_password=test)
```


> **小心**
> 示例标识文件仅适用于测试和开发阶段。由于它是 SQL Anywhere 的标准部分，因此不提供任何保护。请先将其替换为您自己的证书，然后再部署应用程序。

- **DatabaseName [DBN]** 指定处理 Web 请求时要使用的数据库名称，或者使用 REQUIRED 或 AUTO 关键字指定是否需要在 URL 中使用数据库名称。

  如果此参数设置为 REQUIRED，则 URL 必须指定数据库名称。

  如果此参数设置为 AUTO，则 URL 可以指定数据库名，但不必这么做。如果 URL 中不包含数据库名，将使用服务器上的缺省数据库来处理 Web 请求。

  如果此参数设置为数据库的名称，则使用该数据库处理所有 Web 请求。URL 中不得包含数据库名。

- **LocalOnly [LOCAL]** 当设置为 YES 时，此参数将使网络数据库服务器拒绝来自不同计算机上运行的客户端的所有连接。此选项对个人数据库服务器没有任何影响，因为此类服务器从不接受来自其它计算机的 Web 服务请求。缺省值是 NO，表示接受来自客户端的请求，而无论客户端的位置在哪里。

- **LogFile [LOG]** 数据库服务器将有关 Web 服务请求的信息写入到的文件的名称。

- **LogFormat [LF]** 控制写入日志文件的消息的格式以及消息中显示哪些字段。如果消息以字符串形式显示，则在写入每条消息时将使用当前值替换代码（例如 @T）。

  缺省值为 **@T - @W - @I - @P - "@M @U @V" - @R - @L - @E**，这将生成如下所示的消息：

  ```
  06/15 01:30:08.114 - 0.686 - 127.0.0.1 - 80
   - "GET /web/ShowTable HTTP/1.1" - 200 OK - 55133 -
  ```

  日志文件的格式与 Apache 兼容，因此可以使用相同的工具进行分析。

  有关字段代码的详细信息，请参见“LogFormat 协议选项 [LF]”一节 《SQL Anywhere 服务器 - 数据库管理》。

- **LogOptions [LOPT]** 允许您指定用于控制将哪些消息（或哪些类型的消息）写入日志文件的关键字和错误号。请参见“LogOptions 协议选项 [LOPT]”一节 《SQL Anywhere 服务器 - 数据库管理》。

有关可用选项的完整列表以及这些选项的详细信息，请参见“网络协议选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

了解如何解释 URL

统一资源定位符（即 URL）可标识 SOAP 或 HTTP Web 服务提供的文档，例如 HTML 页。SQL Anywhere 中使用的 URL 式样就是您浏览 Web 时经常使用的式样。通过数据库服务器进行浏览的用户无需知道他们的请求没有通过传统的独立 Web 服务器进行处理。

尽管使用标准格式，但 SQL Anywhere 数据库服务器解释 URL 的方式不同于标准 Web 服务器。您在启动数据库服务器时指定的选项也会影响对它们的解释。

URL 的一般语法如下：

{ **http** | **https** }*://*[*user*:*password*@]*host*[:*port*][*/dbn*]*/service-name*[*path* | *?searchpart*]

以下是一个示例 URL： `http://localhost:80/demo/XMLtable`.

用户和口令

如果 Web 服务需要验证，则可以将用户名和口令作为 URL 的一部分直接传递，方式是用冒号将它们分隔并将它们放在主机名前面（与电子邮件地址非常相似）。

主机和端口

与所有标准 HTTP 请求一样，URL 的开始部分包含主机名或 IP 地址，还可以包含端口号。IP 地址（或主机名）及端口应当是服务器正在监听的 IP 地址（或主机名）及端口。IP 地址是运行 SQL Anywhere 的计算机中网卡的地址。端口号将是启动数据库服务器时使用 -xs 选项指定的端口号。如果未指定端口号，则使用该类型服务的缺省端口号。例如，服务器缺省情况下在端口 80 上监听 HTTP 请求。请参见“-xs 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

数据库名

位于斜线之间的下一个标识通常是数据库的名称。此数据库必须在服务器上运行，并且必须含有 Web 服务。

如果 URL 中未出现数据库名，并且没有使用 -xs 服务器选项的 DBN 连接参数指定数据库名，则使用缺省数据库。

只有在以下情况下才能省略数据库名：数据库服务器只运行一个数据库；或者使用 -xs 选项的 DBN 连接参数指定了数据库名。

服务名

URL 的下一部分是服务名。此服务必须存在于指定的数据库中。服务名可以跨出下一个斜线字符，因为 Web 服务名可以包含斜线字符。SQL Anywhere 会将 URL 的其余部分与定义的服务进行匹配。

如果 URL 未提供服务名称，则数据库服务器将查找名为 **root** 的服务。如果未定义指定的服务或 root 服务，则服务器返回 [`404 Not Found`] 错误。

参数

根据目标服务类型，可以通过不同方式提供参数。HTML、XML 和 RAW 服务的参数可以通过以下任何方式传递：

- 使用斜线附加到 URL

- 作为显式 URL 参数列表提供
- 作为 POST 请求中的 POST 数据提供

SOAP 服务的参数必须作为标准 SOAP 请求的一部分提供。以其它方式提供的值将被忽略。

URL 路径

若要访问参数值，必须为参数指定名称。这些主机变量名（以冒号 (:) 为前缀）可包含在构成部分 Web 服务定义的语句中。

例如，假设您定义以下存储过程：

```
CREATE PROCEDURE Display (IN ident INT )
BEGIN
    SELECT ID, GivenName, Surname FROM Customers
    WHERE ID = ident;
END;
```

调用存储过程的语句需要客户标识号。如下所示定义服务：

```
CREATE SERVICE DisplayCustomer
TYPE 'HTML'
URL PATH ELEMENTS
AUTHORIZATION OFF
USER DBA
AS CALL Display( :url1 );
```

它的 URL 实例为：http://localhost:80/demo/DisplayCustomer/105。

将参数 `105` 作为 `url1` 传递给服务。子句 URL PATH ELEMENTS 指明应将由斜线隔开的参数作为参数 `url1`、`url2`、`url3`（以此类推）进行传递。依此方法最多可传递 10 个参数。

由于 Display 过程只有一个参数，因此该服务可能已定义如下：

```
CREATE SERVICE DisplayCustomer
TYPE 'HTML'
URL PATH ON
AUTHORIZATION OFF
USER DBA
AS CALL Display( :url );
```

在本例中，会将参数 `105` 作为 `url` 传递给服务。子句 URL PATH ON 指明应将服务名后面的所有内容作为单个参数（称为 `url`）传递。因此，在以下 URL 中，会将字符串 105/106 作为 `url` 传递（且会产生 SQL 错误，因为 Display 存储过程要求整数值）。

http://localhost:80/demo/DisplayCustomer/105/106

有关变量的详细信息，请参见“处理变量”一节第 879 页。

也可使用 HTTP_VARIABLE 函数访问参数。请参见“HTTP_VARIABLE 函数 [HTTP]”一节《SQL Anywhere 服务器 - SQL 参考》。

URL searchpart

另一个传递参数的方法是通过 URL searchpart 机制传递。URL searchpart 由问号 (?) 和问号后面用 "和" 符号 (&) 隔开的 *name=value* 对组成。searchpart 将被附加到 URL 的结尾。以下示例表明了一般格式：

```
http://server/path/document?name1=value1&name2=value2
```

GET 请求是以这种方式进行格式设置的。如果存在，则定义指定的变量并对其赋予相应的值。

例如，调用存储过程 ShowSalesOrderDetail 的语句既需要客户标识号也需要产品标识号：

```
CREATE SERVICE ShowSalesOrderDetail
TYPE 'HTML'
URL PATH OFF
AUTHORIZATION OFF
USER DBA
AS CALL ShowSalesOrderDetail( :customer_id, :product_id );
```

它的 URL 实例为：http://localhost:80/demo/ShowSalesOrderDetail?customer_id=101&product_id=300。

如果将 URL PATH 设置为 ON 或 ELEMENTS，则会定义其它变量。但这两者在其它情况下通常都是相互独立的。可以通过将 URL PATH 设置为 ON 或 ELEMENTS 来允许在请求的 URL 中使用变量。以下示例说明了如何将这两者混合使用：

```
CREATE SERVICE ShowSalesOrderDetail2
TYPE 'HTML'
URL PATH ON
AUTHORIZATION OFF
USER DBA
AS CALL ShowSalesOrderDetail( :customer_id, :url );
```

在以下示例中，同时使用了 searchpart 和 URL 路径。为 `url` 指派值 300，为 `customer_id` 指派 101。

http://localhost:80/demo/ShowSalesOrderDetail2/300?customer_id=101

也可以使用 searchpart 来表示（仅按以下方式）。

http://localhost:80/demo/ShowSalesOrderDetail2/?customer_id=101&url=300

这样就产生一个问题，即为同一个变量同时指定两者时会出现什么情况。在以下的示例中，前面的 300 和后面的 302 被依次指派到 `url`，并且最后的指派优先。

http://localhost:80/demo/ShowSalesOrderDetail2/300?customer_id=101&url=302

有关变量的详细信息，请参见“处理变量”一节第 879 页。

也可使用 HTTP_VARIABLE 函数访问参数。请参见“HTTP_VARIABLE 函数 [HTTP]”一节《SQL Anywhere 服务器 - SQL 参考》。

创建 SOAP 和 DISH Web 服务

SOAP 和 DISH Web 服务是创建可由标准 SOAP 客户端（如使用 Microsoft .NET 或 JAX-WS 编写的客户端）访问的标准 SOAP Web 服务的方法。

SOAP 服务

SOAP 服务是在 SQL Anywhere 中构造能够接受和处理标准 SOAP 请求的 Web 服务所使用的机制。

要声明 SOAP 服务，请将服务类型指定为 SOAP。标准 SOAP 请求的主体是 SOAP 封装，表示具有特定格式的 XML 文档。SQL Anywhere 使用您提供的过程分析和处理这些请求。响应将自动设置为标准 SOAP 响应（也是 SOAP 封装）的格式并返回客户端。

用于创建 SOAP 服务的语句的语法如下所示：

CREATE SERVICE *service-name*
TYPE 'SOAP'
[**FORMAT** { **'DNET'** | **'CONCRETE'** | **'XML'** | **NULL** }]
[*common-attributes*]
AS *statement*

DISH 服务

DISH 服务充当 SOAP 服务组的代理。此外，它们会自动为其客户端构造用于描述当前公开的 SOAP 服务的 WSDL（Web 服务描述语言）文档。

创建 DISH 服务时，GROUP 子句中给定的名称决定了 DISH 服务公开哪些 SOAP 服务。每个以 DISH 服务名作为其名称前缀的 SOAP 服务都将公开。例如，指定 GROUP xyz 将公开 SOAP 服务 xyz/aaaa、xyz/bbbb 或 xyz/cccc。但不公开名为 abc/aaaa 或 xyzaaaa 的 SOAP 服务。SOAP 服务可由多个 DISH 服务公开。如果没有指定组名，则 DISH 服务公开数据库中的所有 SOAP 服务。DISH 组名中允许使用与 SOAP 服务名相同的字符。

用于创建 DISH 服务的语句的语法如下所示：

CREATE SERVICE *service-name*
TYPE 'DISH'
[**GROUP** { *group-name* | **NULL** }]
[**FORMAT** { **'DNET'** | **'CONCRETE'** | **'XML'** | **NULL** }]
[*common-attributes*]

SOAP 和 DISH 服务格式

可以使用 CREATE SERVICE 语句的 FORMAT 子句自定义 SOAP 服务数据载荷，以最好地适应各种类型的 SOAP 客户端，如 .NET 和 JAX-WS。FORMAT 子句会影响 DISH 服务返回的 WSDL 文档的内容以及 SOAP 响应中返回的数据载荷的格式。

缺省格式 DNET 是适于与 .NET SOAP 客户端应用程序一同使用的本机格式（此类应用程序要求使用 .NET 数据集格式）。

CONCRETE 格式适于与根据返回数据结构的格式自动生成接口的客户端（如 JAX-WS 和 .NET）一同使用。指定此格式时，SQL Anywhere 返回的 WSDL 文档将公开一个具体描述结果集的 SimpleDataset 元素。此元素是由一组行（每行包含一组列元素）组成的行集的包容层次。

XML 格式适于与 SOAP 客户端一同使用，此类客户端将 SOAP 响应作为一个大型字符串接受并使用 XML 分析程序查找和抽取所需元素和值。此格式通常是不同类型的 SOAP 客户端之间最便于移植的格式。

如果未指定 SOAP 服务的格式，则从该服务的 DISH 服务声明继承格式。如果 DISH 服务也没有声明格式，则格式缺省设置为与 .NET 客户端兼容的 DNET。可以通过定义多个 DISH 服务（每个都使用一个不同的 FORMAT 类型）将没有声明格式的 SOAP 服务与不同类型的 SOAP 客户端同用。

创建同类 DISH 服务

SOAP 服务无需指定格式类型（即，您可将格式类型设置为 NULL）。在这种情况下，则从充当 SOAP 服务代理的 DISH 服务继承格式。每个 SOAP 服务可有多个 DISH 服务充当代理，并且这些 DISH 服务无需是同一类型。这些事实表明，可以通过使用多个属于不同类型的 DISH 服务将单个 SOAP 服务与不同类型的 SOAP 客户端（如 .NET 和 JAX-WS）一同使用。此类 DISH 服务称为**同类**服务，因为它们公开相同 SOAP 服务的相同数据载荷，只不过采用不同的格式。

以下面两个都没有指定格式的 SOAP 服务为例：

```
CREATE SERVICE "abc/hello"
TYPE 'SOAP'
AS CALL hello(:student);

CREATE SERVICE "abc/goodbye"
TYPE 'SOAP'
AS CALL goodbye(:student);
```

因为这两个服务都不包含 FORMAT 子句，所以格式缺省为 NULL；它继承自充当代理的 DISH 服务。现在，假定有下面两个 DISH 服务：

```
CREATE SERVICE "abc_xml"
TYPE 'DISH'
GROUP "abc"
FORMAT 'XML';

CREATE SERVICE "abc_concrete"
TYPE 'DISH'
GROUP "abc"
FORMAT 'CONCRETE';
```

由于两个 DISH 服务都指定相同的组名 abc，因此它们充当相同 SOAP 服务（即名称具有前缀 "abc/" 的所有 SOAP 服务）的代理。

但如果两个 SOAP 服务中有任一个是通过 abc_xml DISH 服务访问的，则该 SOAP 服务将继承 XML 格式；如果是通过 abc_concrete SOAP 服务访问，则继承 CONCRETE 格式。

每当想要实现不同类型 SOAP 客户端对所创建的 SOAP Web 服务的访问时，通过同类 DISH 服务便可避免重复的服务。

教程：从 Microsoft .NET 访问 Web 服务

以下教程演示了如何使用 Visual C# 从 Microsoft .NET 访问 Web 服务。

♦ 创建 SOAP 和 DISH 服务

1. 在命令提示符处，运行以下命令以启动个人 Web 服务器。将 *samples-dir* 替换为示例数据库的实际位置。`-xs http(port=80)` 选项会告知数据库服务器接受来自端口 80 的 HTTP 请求。如果在端口 80 上已有 Web 服务器在运行，则将另一个端口号（如 8080）用于本教程，并在所有端口引用中使用 8080 来代替 80。

   ```
   dbeng11 -xs http(port=80) samples-dir\demo.db
   ```

2. 启动 Interactive SQL。以 DBA 身份连接到 SQL Anywhere 示例数据库。执行以下语句：

   a. 定义一个列出 Employees 表的 SOAP 服务。

      ```
      CREATE SERVICE "SASoapTest/EmployeeList"
      TYPE 'SOAP'
      AUTHORIZATION OFF
      SECURE OFF
      USER DBA
      AS SELECT * FROM Employees;
      ```

      由于授权已关闭，因此任何人都可使用此服务而无需提供用户名和口令。命令在用户 DBA 下运行。这种安排方式虽然简单，但不安全。

   b. 创建一个 DISH 服务以充当 SOAP 服务的代理并生成 WSDL 文档。

      ```
      CREATE SERVICE "SASoapTest_DNET"
      TYPE 'DISH'
      GROUP "SASoapTest"
      FORMAT 'DNET'
      AUTHORIZATION OFF
      SECURE OFF
      USER DBA;
      ```

      SOAP 和 DISH 服务必须为 DNET 格式。在本示例中，创建 SOAP 服务时省略了 FORMAT 子句。这样，SOAP 服务将从 DISH 服务继承 DNET 格式。

3. 启动 Microsoft Visual Studio。请注意，本示例使用 .NET Framework 2.0 中的函数。

   a. 使用 Visual C# 新建一个 [Windows Application] 项目。

      将出现一个空的窗体。

   b. 从 [**Project**] 菜单中，选择 [**Add Web Reference**]。

   c. 在 [**Add Web Reference**] 页面的 [**URL**] 字段中，输入以下 URL： **http://localhost:80/demo/SASoapTest_DNET**.

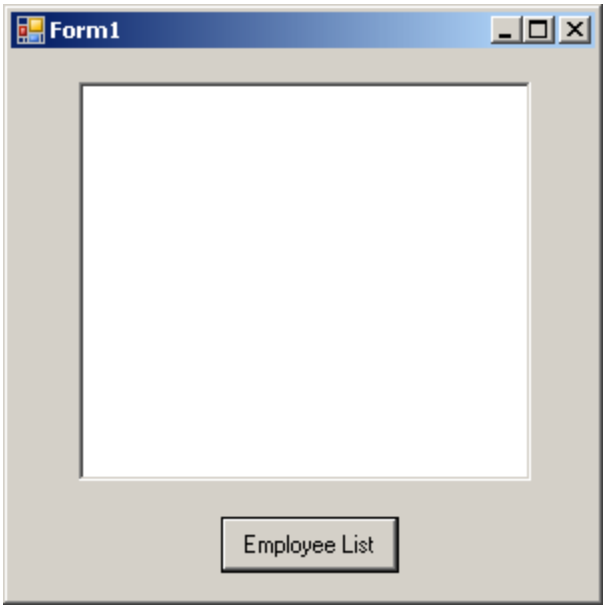
   d. 单击 [**Go**]。

      此时显示可供用于 SASoapTest_DNET 的方法的列表。您将看到 EmployeeList 方法。

   e. 单击 [**Add Reference**] 完成操作。

      [**Solution Explorer**] 窗口即会显示新的 Web 引用。

f. 在 Visual Studio [**Toolbox**] 中，向窗体添加一个列表框和一个按钮，如下图所示。



g. 将按钮文本重命名为 **Employee List**。

h. 双击 [**Employee List**]，然后为按钮的 click 事件添加以下代码。

```
int sqlCode;

listBox1.Items.Clear();

localhost.SASoapTest_DNET proxy = new localhost.SASoapTest_DNET();

DataSet results = proxy.EmployeeList(out sqlCode);
DataTableReader dr = results.CreateDataReader();
while (dr.Read())
{
  for (int i = 0; i < dr.FieldCount; i++)
  {
    string columnName = dr.GetName(i);
    try
    {
      string value = dr.GetString(i);
      listBox1.Items.Add(columnName + "=" + value);
    }
    catch ( InvalidCastException )
    {
      listBox1.Items.Add(columnName + "=(null)");
    }
  }
  listBox1.Items.Add("");
}
dr.Close();
```

i. 构建并运行程序。

列表框将以 column name=value 对的形式显示 EmployeeList 结果集。

会包括一个 try/catch 块来处理 NULL 列值（例如可以在 Employees 表的 TerminationDate 列中找到的列值）。

教程：从 JAX-WS 访问 Web 服务

以下教程演示了如何使用 Java API for XML Web Services (JAX-WS) 访问 Web 服务。

在开始之前，您需要有 Sun 提供的 JAX-WS 工具。如果您的系统上没有安装此软件包，您必须下载 JAX-WS 工具并安装它们。要下载 JAX-WS 工具，请访问 http://java.sun.com/webservices/。单击 JAX-WS 的链接。您将进入 Java API for XML Web Services 页面。单击 [**Download Now**] 链接。在您已经下载了软件包之后，将它安装在您的系统上。

此示例是使用 JAX-WS 2.1.3 for Windows 开发的。

应将从 JAX-WS 访问的 SQL Anywhere SOAP Web 服务声明为 CONCRETE 格式。

♦ 创建 SOAP 和 DISH 服务

1. 在命令提示符处，运行以下命令以启动个人 Web 服务器。将 *samples-dir* 替换为示例数据库的实际位置。`-xs http(port=80)` 选项会告知数据库服务器接受来自端口 80 的 HTTP 请求。如果在端口 80 上已有 Web 服务器在运行，则将另一个端口号（如 8080）用于本教程，并在所有端口引用中使用 8080 来代替 80。

   ```
   dbeng11 -xs http(port=80) samples-dir\demo.db
   ```

2. 启动 Interactive SQL，并以 DBA 身份连接到 SQL Anywhere 示例数据库。执行以下语句：

   a. 定义一个列出 Employees 表的某些列的存储过程。

   ```
   CREATE PROCEDURE ListEmployees()
   RESULT (
    EmployeeID            INTEGER,
    Surname               CHAR(20),
    GivenName             CHAR(20),
    StartDate             DATE,
    TerminationDate       DATE )
   BEGIN
     SELECT EmployeeID, Surname, GivenName,
       StartDate, TerminationDate
     FROM Employees;
   END;
   ```

   b. 定义一个调用此存储过程的 SOAP 服务。

   ```
   CREATE SERVICE "WS/EmployeeList"
   TYPE 'SOAP'
   FORMAT 'CONCRETE' EXPLICIT OFF
   DATATYPE OUT
   AUTHORIZATION OFF
   SECURE OFF
   USER DBA
   AS CALL ListEmployees();
   ```

   EXPLICIT 子句仅能用于 CONCRETE 类型的 SOAP 或 DISH 服务。在此示例中，EXPLICIT OFF 表示相应的 DISH 服务应生成说明通用 SimpleDataset 对象的 XML 模式。此选项仅影响生成的 WSDL 文档。稍后，我们将看一个使用 EXPLICIT ON 的示例。请参见“教程：将数据类型与 JAX-WS 一起使用”一节第 855 页。

   DATATYPE OUT 表示显式数据类型信息在 XML 结果集响应中生成。如果已指定 DATATYPE OFF，那么所有数据将作为字符串键入。此选项不影响生成 WSDL 文档。

由于授权已关闭，因此任何人都可使用此服务而无需提供用户名和口令。命令在用户 DBA 下运行。这种安排虽然简单，但不安全。

c. 创建一个 DISH 服务以充当 SOAP 服务的代理并生成 WSDL 文档。

```
CREATE SERVICE "WSDish"
TYPE 'DISH'
FORMAT 'CONCRETE'
GROUP "WS"
AUTHORIZATION OFF
SECURE OFF
USER DBA;
```

SOAP 和 DISH 服务必须都是 CONCRETE 格式。由于 EmployeeList 服务在 WS 组中，因此会包括 GROUP 子句。

3. 查看 DISH 服务自动创建的 WSDL。为此，请打开 Web 浏览器并浏览至以下 URL：http://localhost:80/demo/WSDish。DISH 服务会自动生成一个 WSDL 文档，该文档将出现在浏览器窗口中。

特别是要注意被公开的 SimpleDataset 对象，因为该服务是 CONCRETE 格式并且 EXPLICIT 为 OFF。在随后的步骤中，**wsimport** 应用程序使用此信息为该服务生成一个 SOAP 1.1 客户端接口。

```
<s:complexType name="SimpleDataset">
<s:sequence>
<s:element name="rowset">
  <s:complexType>
  <s:sequence>
  <s:element name="row" minOccurs="0" maxOccurs="unbounded">
    <s:complexType>
    <s:sequence>
      <s:any minOccurs="0" maxOccurs="unbounded" />
    </s:sequence>
    </s:complexType>
  </s:element>
  </s:sequence>
  </s:complexType>
</s:element>
</s:sequence>
</s:complexType>
```

♦ 为这些 Web 服务生成 JAX-WS 接口

1. 在本示例中，Java API for XML Web Services (JAX-WS) 和 Sun Java 1.6.0 JDK 安装在驱动器 *C:* 上。设置 PATH 环境变量，使其包含 JAX-WS 二进制文件和 JDK 二进制文件。二进制文件位于以下目录中。

```
c:\Sun\jaxws-ri\bin
c:\Sun\SDK\jdk\bin
```

2. 在命令提示符处，设置 CLASSPATH 环境变量。

```
SET classpath=.;C:\Sun\jaxws-ri\lib\jaxb-api.jar
;C:\Sun\jaxws-ri\lib\jaxws-rt.jar
```

3. 下一步是生成调用 Web 服务所需的接口。

在同一命令提示符处，创建一个新的项目目录，并使该目录成为您的当前目录。在此目录中执行以下命令。

```
wsimport -keep -Xendorsed "http://localhost:80/demo/WSDish"
```

该 wsimport 工具从给定的 URL 中检索 WSDL 文档，并生成为其定义接口的 Java 文件，然后编译 Java 文件。

keep 选项将指示 wsimport 不要删除 *.java* 文件。如果不使用此选项，则会在生成相应的 *.class* 文件后删除这些文件。如果保存这些文件，会更易于检查接口的组成。

Xendorsed 选项允许您使用 JDK6 版的 JAX-WS 2.1 API。

此命令完成后，会在您的当前目录中生成一个名为 *localhost\demo\ws* 的新子目录结构，其中包含以下 Java 文件，以及每个 Java 文件编译后的 *.class* 版本。

```
EmployeeList.java
EmployeeListResponse.java
FaultMessage.java
ObjectFactory.java
package-info.java
SimpleDataset.java
WSDish.java
WSDishSoapPort.java
```

♦ 使用生成的 JAX-WS 接口

1. 将以下 Java 源代码保存到 *SASoapDemo.java* 中。确保该文件位于含有 *localhost* 子目录（由 wsimport 工具生成）的同一目录中。

```
// SASoapDemo.java illustrates a web service client that
// calls the WSDish service and prints out the data.

import java.util.*;
import javax.xml.ws.*;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import localhost.demo.ws.*;

public class SASoapDemo
{
  @WebServiceRef( wsdlLocation= "http://localhost:8080/demo/WSDish" )
  public static void main( String[] args )
  {
    try {
      WSDish service = new WSDish();

      Holder<SimpleDataset> response = new Holder<SimpleDataset>();
      Holder<Integer> sqlcode = new Holder<Integer>();

      WSDishSoapPort port = service.getWSDishSoap();

      // This is the SOAP service call to EmployeeList
      port.employeeList( response, sqlcode );

      localhost.demo.ws.SimpleDataset result = response.value;
      SimpleDataset.Rowset rowset = result.getRowset();

      List<SimpleDataset.Rowset.Row> rows = rowset.getRow();
```



```
        for ( int i = 0; i < rows.size(); i++ ) {
          SimpleDataset.Rowset.Row row = rows.get( i );
          List<Object> cols = row.getAny();

          System.out.println( "Number of columns=" + cols.size() );

          for ( int j = 0; j < cols.size(); j++ ) {
            // Column data is contained as a SOAPElement
            Element col = (Element)cols.get(j);
            System.out.print(col.getLocalName() + "=" );
            Node node = col.getFirstChild();
            if( node == null ) {
              System.out.println( "(null)" );
            } else {
              System.out.println( node.getNodeValue() );
            }
          }
          System.out.println();
        }

      }
      catch (Exception x) {
        x.printStackTrace();
      }
    }
  }
```

如果您选择使用一个不同的端口号（如 8080）来启动 web 服务器，则必须将 [`import localhost`] 源行更改为如下形式：

```
import localhost._8080.demo.ws.*;
```

2. 编译 *SASoapDemo.java*。

```
javac SASoapDemo.java
```

3. 运行编译过的类文件。

```
java SASoapDemo
```

当应用程序向 web 服务器发送请求时，它会收到一个包含 EmployeeListResult（带有包含几个行条目的行集）的 XML 结果集响应。此响应中还包含执行查询的 sqlcode 结果。响应的示例如下所示。

```
<tns:EmployeeListResponse>
 <tns:EmployeeListResult xsi:type='tns:SimpleDataset'>
  <tns:rowset>
    <tns:row>...</tns:row>
.
.
.
    <tns:row>...</tns:row>
   </tns:rowset>
 </tns:EmployeeListResult>
 <tns:sqlcode>0</tns:sqlcode>
</tns:EmployeeListResponse>
```

行集中的每行均以类似如下的格式发送。

```
<tns:row>
   <tns:EmployeeID xsi:type="xsd:int">1751</tns:EmployeeID>
```

```
    <tns:Surname xsi:type="xsd:string">Ahmed</tns:Surname>
    <tns:GivenName xsi:type="xsd:string">Alex</tns:GivenName>
    <tns:StartDate xsi:type="xsd:date">1994-07-12-04:00</tns:StartDate>
    <tns:TerminationDate xsi:type="xsd:date">2008-04-18-04:00
        </tns:TerminationDate>
</tns:row>
```

请注意，列名和数据类型包括在内。

使用代理
您可以通过使用记录 XML 消息流量的代理软件来观察上面所示的响应。代理软件置身于客户端应用程序和 Web 服务器之间。

SASoapDemo 应用程序将以 (type)column name=value 对的形式显示 EmployeeList 结果集。将生成如下所示的几行输出。

EmployeeList 结果集会显示为 column name=value 对。将生成如下所示的几行输出。

```
Number of columns=4
EmployeeID=102
Surname=Whitney
GivenName=Fran
StartDate=1984-08-28-04:00

Number of columns=4
EmployeeID=105
Surname=Cobb
GivenName=Matthew
StartDate=1985-01-01-05:00
.
.
.
Number of columns=4
EmployeeID=1740
Surname=Nielsen
GivenName=Robert
StartDate=1994-06-24-04:00

Number of columns=5
EmployeeID=1751
Surname=Ahmed
GivenName=Alex
StartDate=1994-07-12-04:00
TerminationDate=2008-04-18-04:00
```

请注意，仅当 TerminationDate 列值为非 NULL 时，才发送它。对于本示例，Employees 表的最后一行被改动，因此将设置一个非 NULL 终止日期。

还应注意，日期值包含一个 UTC 时间偏移值。在上面的示例数据中，服务器位于北美东部时区。如果不执行夏令时，该时区的时间比 UTC 时间早 5 个小时 (-05:00)；如果执行夏令时，该时区的时间比 UTC 时间早 4 个小时 (-04:00)。

使用提供 HTML 文档的过程

通常，编写一个可处理发送给特定服务的请求的过程是最简单的方式。这样的过程将返回一个 Web 页。或者，该过程也可接受作为 URL 的一部分进行传递的参数以对其输出进行自定义。

但以下示例要简单很多。它例示了服务可以简单到什么程度。此 Web 服务仅返回短语 "Hello world!"。

```
CREATE SERVICE hello
TYPE 'RAW'
AUTHORIZATION OFF
USER DBA
AS SELECT 'Hello world!';
```

使用 -xs 选项启动数据库服务器以启用对 Web 请求的处理，然后从任一 Web 浏览器请求 URL http://localhost/hello。"Hello world!" 将显示在另外的纯文本页面上。

HTML 页

上述页面在浏览器中以纯文本格式显示。这是因为缺省 HTTP Content-Type 是 text/plain。若要创建更标准的 Web 页（HTML 格式），则必须执行两个任务：

- 将 HTTP Content-Type 标头字段设置为 text/html，以使浏览器预期 HTML。
- 在输出中包含 HTML 标记。

可通过两种方法将标记写入到输出中。一种方法是使用 CREATE SERVICE 语句中的短语 TYPE 'HTML'。这样会指示 SQL Anywhere 数据库服务器为您添加 HTML 标记。在某些情况下（例如在返回表时），这种方法很有效。

另一种方法是使用 TYPE 'RAW' 并自己编写所有必要的标记。使用第二种方法可对输出进行最大程度的控制。请注意，指定 RAW 类型不一定表示输出不会采用 HTML 或 XML 格式。它只是告知 SQL Anywhere 可将返回值直接传递给客户端，而无需自己添加标记。

以下过程将生成一版更别致的 "Hello world"。为方便起见，主体工作在下面的过程中完成，该过程设置了 Web 页的格式。

内置过程 sa_set_http_header 用于设置 HTTP 标头类型，使浏览器能够正确解释结果。如果省略此语句，浏览器将显示所有 HTML 代码，而不是使用它们设置文档格式。

```
CREATE PROCEDURE hello_pretty_world ()
RESULT (html_doc XML)
BEGIN
  CALL dbo.sa_set_http_header( 'Content-Type', 'text/html' );
  SELECT HTML_DECODE(
    XMLCONCAT(
      '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">',
      XMLELEMENT('HTML',
      XMLELEMENT('HEAD',
          XMLELEMENT('TITLE', 'Hello Pretty World')
        ),
        XMLELEMENT('BODY',
          XMLELEMENT('H1', 'Hello Pretty World!'),
          XMLELEMENT('P',
            '(If you see the tags in your browser, check that '
            || 'the Content-Type header is set to text/html.)'
          )
```

```
        )
      )
    )
  );
END
```

以下语句将创建一个使用此过程的服务。该语句将调用上述过程，该过程会生成 "Hello Pretty World" Web 页。

```
CREATE SERVICE hello_pretty_world
TYPE 'RAW'
AUTHORIZATION OFF
USER DBA
AS CALL hello_pretty_world();
```

创建该过程和服务后，便可以访问 Web 页了。请确保数据库服务器是使用正确的 -xs 选项值启动的，然后在 Web 浏览器中打开 URL http://localhost/hello_pretty_world。

您将看到以简单的 HTML 页格式显示的结果，标题是 "Hello Pretty World"。您可以通过加入更多内容、使用更多标记、使用样式表或者加入在浏览器中运行的脚本来按照自己的需要详尽设计该 Web 页。在任何情况下，都要创建处理浏览器请求所必需的服务。

有关内置存储过程的详细信息，请参见“按字母顺排序的系统过程列表”一节 《SQL Anywhere 服务器 - SQL 参考》。

Root 服务

如果 URL 中不含服务名，则 SQL Anywhere 会查找名为 **root** 的 Web 服务。Root 页的作用与许多传统 Web 服务器中 *index.html* 页的作用类似。

Root 服务在创建主页时很方便，因为它们可以处理只含有 Web 站点地址的 URL 请求。在处理不存在的 URL 路径时，它们同样能够提供方便。例如，以下过程和服务会实现一个在您浏览至 URL http://localhost 时显示的简单 Web 页。它还用于处理浏览至并不存在的页面的情况。

```
CREATE PROCEDURE HomePage( IN url LONG VARCHAR )
RESULT (html_doc XML)
BEGIN
  CALL dbo.sa_set_http_header( 'Content-Type', 'text/html' );
  IF url IS NULL THEN
    SELECT HTML_DECODE(
      XMLCONCAT(
        '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">',
        XMLELEMENT('HTML',
          XMLELEMENT('HEAD',
            XMLELEMENT('TITLE', 'My Home Page')
          ),
          XMLELEMENT('BODY',
            XMLELEMENT('H1', 'My home on the web'),
            XMLELEMENT('P',
              'Thank you for visiting my web site!'
            )
          )
        )
      )
    )
  ELSE
    CALL dbo.sa_set_http_header('Status','404');
    SELECT '<H1>Status 404 - Document ' || url || ' not found</H1>'
  END IF
END
```


现在创建一个使用此过程的服务：

```
CREATE SERVICE root
TYPE 'RAW'
AUTHORIZATION OFF
SECURE OFF
URL PATH ON
USER DBA
AS CALL HomePage(:url);
```

只要未指定在启动数据库服务器时必须提供数据库名，便可以通过浏览至 URL http://localhost 来访问此 Web 页。请参见“启动监听 Web 请求的数据库服务器”一节第 826 页。通过指定 **URL PATH ON**，可确保不存在的 URL 路径指向此服务。

示例

更多详尽示例包含在 *samples-dir\SQLAnywhere\HTTP* 目录中。

使用数据类型

缺省情况下，参数输入的 XML 编码为字符串，SOAP 服务格式的结果集输出中不包含明确描述结果集中列的数据类型的信息。对于所有格式，参数数据类型都为字符串。对于 DNET 格式，在响应的模式部分中所有列的数据类型均设置为字符串。CONCRETE 和 XML 格式不包含响应中的数据类型信息。可使用 DATATYPE 子句处理此缺省行为。

SQL Anywhere 允许使用 DATATYPE 子句设置数据类型。数据类型信息包含在参数输入和结果集输出或所有 SOAP 服务格式的响应的 XML 编码中。这样一来便不再需要客户端代码将参数显式转换为字符串，从而简化了从 SOAP 工具箱进行参数传递。例如，可将整数作为整型进行传递。XML 编码的数据类型允许 SOAP 工具箱分析数据并将其归到相应类型。

以独占方式使用字符串数据类型时，应用程序需要隐式地了解结果集内每列的数据类型。当 Web 服务器请求数据类型时，无需执行此操作。在定义 Web 服务时，可使用 DATATYPE 子句控制是否包括数据类型信息。

DATATYPE { OFF | ON | IN | OUT }

- **OFF** 未使用 DATATYPE 选项时，此行为将是缺省行为。对于 DNET 输出格式，SQL Anywhere 数据类型将在 XML 模式字符串类型间相互转换。对于 CONCRETE 和 XML 格式，将不发出任何数据类型信息。
- **ON** 为输入参数和结果集响应发出数据类型信息。将 SQL Anywhere 数据类型在 XML 模式数据类型间相互转换。
- **IN** 仅发出输入参数的数据类型信息。
- **OUT** 仅发出结果集响应的数据类型信息。

下面是征用结果集响应数据类型的 Web 服务定义的示例。

```
CREATE SERVICE "SASoapTest/EmployeeList"
TYPE 'SOAP'
AUTHORIZATION OFF
SECURE OFF
USER DBA
DATATYPE OUT
AS SELECT * FROM Employees;
```

在此示例中，由于此服务没有参数，因此仅为结果集响应请求数据类型信息。

数据类型适用于所有定义为 'SOAP' 类型的 SQL Anywhere Web 服务。

输入参数的数据类型

只要将参数数据类型公开为其在 WSDL 中由 DISH 服务生成的真实数据类型就可支持输入参数的数据类型。

典型字符串参数定义（或非类型参数）应如下所示：

```
<s:element minOccurs="0" maxOccurs="1" name="a_varchar" nillable="true"
type="s:string" />
```

字符串参数可以为空，即它可以发生，也可以不发生。

对于确定类型的参数（例如整数），必须有该参数且不能为空。以下是一个示例。

```
<s:element minOccurs="1" maxOccurs="1" name="an_int" nillable="false"
type="s:int" />
```

输出参数的数据类型

所有 'SOAP' 类型的 SQL Anywhere Web 服务都可公开响应数据内的数据类型信息。将数据类型公开为行集列元素内的属性。

以下是一个来自 SOAP FORMAT 'CONCRETE' Web 服务的 SimpleDataSet 类型响应的示例。

```
<SOAP-ENV:Body>
  <tns:test_types_concrete_onResponse>
   <tns:test_types_concrete_onResult xsi:type='tns:SimpleDataset'>
    <tns:rowset>
      <tns:row>
       <tns:lvc xsi:type="xsd:string">Hello World</tns:lvc>
       <tns:i xsi:type="xsd:int">99</tns:i>
       <tns:ii xsi:type="xsd:long">99999999</tns:ii>
       <tns:f xsi:type="xsd:float">3.25</tns:f>
       <tns:d xsi:type="xsd:double">.55555555555555582</tns:d>
       <tns:bin xsi:type="xsd:base64Binary">AAAAZg==</tns:bin>
       <tns:date xsi:type="xsd:date">2006-05-29-04:00</tns:date>
      </tns:row>
     </tns:rowset>
   </tns:test_types_concrete_onResult>
  <tns:sqlcode>0</tns:sqlcode>
  </tns:test_types_concrete_onResponse>
</SOAP-ENV:Body>
```

以下是一个返回字符串格式 XML 数据的 SOAP FORMAT 'XML' Web 服务的响应示例。内部行集由编码的 XML 组成，为便于理解在此处以其解码形式显示。

```
<SOAP-ENV:Body>
 <tns:test_types_XML_onResponse>
  <tns:test_types_XML_onResult xsi:type='xsd:string'>
   <tns:rowset
     xmlns:tns="http://localhost/satest/dish"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <tns:row>
     <tns:lvc xsi:type="xsd:string">Hello World</tns:lvc>
     <tns:i xsi:type="xsd:int">99</tns:i>
     <tns:ii xsi:type="xsd:long">99999999</tns:ii>
     <tns:f xsi:type="xsd:float">3.25</tns:f>
     <tns:d xsi:type="xsd:double">.55555555555555582</tns:d>
     <tns:bin xsi:type="xsd:base64Binary">AAAAZg==</tns:bin>
     <tns:date xsi:type="xsd:date">2006-05-29-04:00</tns:date>
    </tns:row>
   </tns:rowset>
  </tns:test_types_XML_onResult>
  <tns:sqlcode>0</tns:sqlcode>
 </tns:test_types_XML_onResponse>
</SOAP-ENV:Body>
```

请注意，除了数据类型信息外，这些元素的命名空间和 XML 模式还提供 XML 分析程序进行后处理所必需的全部信息。如果结果集中不存在数据类型信息（DATATYPE OFF 或 IN），则忽略 xsi:type 和 XML 模式命名空间声明。

SOAP FORMAT 'DNET' Web 服务返回的 SimpleDataSet 类型的示例如下：

```
<SOAP-ENV:Body>
  <tns:test_types_dnet_outResponse>
```

```
    <tns:test_types_dnet_outResult xsi:type='sqlresultstream:SqlRowSet'>
     <xsd:schema id='Schema2'
        xmlns:xsd='http://www.w3.org/2001/XMLSchema'
        xmlns:msdata='urn:schemas-microsoft.com:xml-msdata'>
      <xsd:element name='rowset' msdata:IsDataSet='true'>
       <xsd:complexType>
        <xsd:sequence>
         <xsd:element name='row' minOccurs='0' maxOccurs='unbounded'>
          <xsd:complexType>
           <xsd:sequence>
            <xsd:element name='lvc' minOccurs='0' type='xsd:string' />
            <xsd:element name='ub' minOccurs='0' type='xsd:unsignedByte' />
            <xsd:element name='s' minOccurs='0' type='xsd:short' />
            <xsd:element name='us' minOccurs='0' type='xsd:unsignedShort' />
            <xsd:element name='i' minOccurs='0' type='xsd:int' />
            <xsd:element name='ui' minOccurs='0' type='xsd:unsignedInt' />
            <xsd:element name='l' minOccurs='0' type='xsd:long' />
            <xsd:element name='ul' minOccurs='0' type='xsd:unsignedLong' />
            <xsd:element name='f' minOccurs='0' type='xsd:float' />
            <xsd:element name='d' minOccurs='0' type='xsd:double' />
            <xsd:element name='bin' minOccurs='0' type='xsd:base64Binary' />
            <xsd:element name='bool' minOccurs='0' type='xsd:boolean' />
            <xsd:element name='num' minOccurs='0' type='xsd:decimal' />
            <xsd:element name='dc' minOccurs='0' type='xsd:decimal' />
            <xsd:element name='date' minOccurs='0' type='xsd:date' />
           </xsd:sequence>
          </xsd:complexType>
         </xsd:element>
        </xsd:sequence>
       </xsd:complexType>
      </xsd:element>
     </xsd:schema>
     <diffgr:diffgram xmlns:msdata='urn:schemas-microsoft-com:xml-msdata'
xmlns:diffgr='urn:schemas-microsoft-com:xml-diffgram-v1'>
      <rowset>
       <row>
        <lvc>Hello World</lvc>
        <ub>128</ub>
        <s>-99</s>
        <us>33000</us>
        <i>-2147483640</i>
        <ui>4294967295</ui>
        <l>-9223372036854775807</l>
        <ul>18446744073709551615</ul>
        <f>3.25</f>
        <d>.555555555555555582</d>
        <bin>QUJD</bin>
        <bool>1</bool>
        <num>123456.123457</num>
        <dc>-1.756000</dc>
        <date>2006-05-29-04:00</date>
       </row>
      </rowset>
     </diffgr:diffgram>
    </tns:test_types_dnet_outResult>
   <tns:sqlcode>0</tns:sqlcode>
   </tns:test_types_dnet_outResponse>
</SOAP-ENV:Body>
```


将 SQL Anywhere 类型映射到 XML 模式类型

SQL Anywhere 类型	XML 模式类型	XML 示例
CHAR	string	Hello World
VARCHAR	string	Hello World
LONG VARCHAR	string	Hello World
TEXT	string	Hello World
NCHAR	string	Hello World
NVARCHAR	string	Hello World
LONG NVARCHAR	string	Hello World
NTEXT	string	Hello World
UNIQUEIDENTIFIER	string	12345678-1234-5678-9012-123456789012
UNIQUEIDENTIFIERSTR	string	12345678-1234-5678-9012-123456789012
XML	此项由用户定义。假定参数是表示复杂类型（例如 base64Binary、SOAP 数组、struct）的有效 XML。	<inputHexBinary xsi:type="xsd:hexBinary"> 414243 </inputHexBinary> （解释为 'ABC'）
BIGINT	long	-9223372036854775807
UNSIGNED BIGINT	unsignedLong	18446744073709551615
BIT	boolean	1
VARBIT	string	11111111
LONG VARBIT	string	00000000000000001000000000000000
DECIMAL	decimal	-1.756000
DOUBLE	double	.555555555555555582
FLOAT	float	12.3456792831420898
INTEGER	int	-2147483640
UNSIGNED INTEGER	unsignedInt	4294967295

SQL Anywhere 类型	XML 模式类型	XML 示例
NUMERIC	decimal	123456.123457
REAL	float	3.25
SMALLINT	short	-99
UNSIGNED SMALLINT	unsignedShort	33000
TINYINT	unsignedByte	128
MONEY	decimal	12345678.9900
SMALLMONEY	decimal	12.3400
DATE	date	2006-11-21-05:00
DATETIME	dateTime	2006-05-21T09:00:00.000-08:00
SMALLDATETIME	dateTime	2007-01-15T09:00:00.000-08:00
TIME	time	14:14:48.980-05:00
TIMESTAMP	dateTime	2007-01-12T21:02:14.420-06:00
BINARY	base64Binary	AAAAZg==
IMAGE	base64Binary	AAAAZg==
LONG BINARY	base64Binary	AAAAZg==
VARBINARY	base64Binary	AAAAZg==

如果一个或多个参数属于 NCHAR、NVARCHAR、LONG NVARCHAR 或 NTEXT 类型，则响应输出采用 UTF8 格式。如果客户端数据库使用 UTF-8 字符编码，则不存在行为的变化（因为 NCHAR 和 CHAR 数据类型是相同的）。不过，如果数据库不使用 UTF-8 字符编码，则所有不属于 NCHAR 数据类型的参数都将转换为 UTF8。XML 声明编码和 Content-Type HTTP 标头的值将与使用的字符编码相对应。

将 XML 模式类型映射到 Java 类型

XML 模式类型	Java 数据类型
xsd:string	java.lang.String
xsd:integer	java.math.BigInteger
xsd:int	int

XML 模式类型	Java 数据类型
xsd:long	long
xsd:short	short
xsd:decimal	java.math.BigDecimal
xsd:float	float
xsd:double	double
xsd:boolean	boolean
xsd:byte	byte
xsd:QName	javax.xml.namespace.QName
xsd:dateTime	javax.xml.datatype.XMLGregorianCalendar
xsd:base64Binary	byte[ ]
xsd:hexBinary	byte[ ]
xsd:unsignedInt	long
xsd:unsignedShort	int
xsd:unsignedByte	short
xsd:time	javax.xml.datatype.XMLGregorianCalendar
xsd:date	javax.xml.datatype.XMLGregorianCalendar
xsd:g	javax.xml.datatype.XMLGregorianCalendar
xsd:anySimpleType	java.lang.Object
xsd:anySimpleType	java.lang.String
xsd:duration	javax.xml.datatype.Duration
xsd:NOTATION	javax.xml.namespace.QName

教程：将数据类型与 Microsoft .NET 一起使用

以下教程演示了使用 Visual C# 时如何从 Microsoft .NET 中使用所支持的 SQL Anywhere Web 服务数据类型。

♦ 创建 SOAP 和 DISH 服务

1. 在命令提示符处，运行以下命令以启动个人 Web 服务器。将 *samples-dir* 替换为示例数据库的实际位置。`-xs http(port=80)` 选项会告知数据库服务器接受来自端口 80 的请求。如果在端口 80 上已有 Web 服务器在运行，则将另一个端口号（如 8080）用于本教程。

   ```
   dbeng11 -xs http(port=80) samples-dir\demo.db
   ```

2. 启动 Interactive SQL。以 DBA 身份连接到 SQL Anywhere 示例数据库。执行以下语句：

   a. 定义一个列出 Employees 表的 SOAP 服务。

   ```
   CREATE SERVICE "SASoapTest/EmployeeList"
   TYPE 'SOAP'
   AUTHORIZATION OFF
   SECURE OFF
   USER DBA
   DATATYPE OUT
   AS SELECT * FROM Employees;
   ```

   在此示例中，指定了 DATATYPE OUT 以生成 XML 结果集响应中的数据类型信息。来自 Web 服务器的响应片段如下所示。请注意，该类型信息与数据库列的数据类型相匹配。

   ```
   <xsd:element name='EmployeeID' minOccurs='0' type='xsd:int' />
   <xsd:element name='ManagerID' minOccurs='0' type='xsd:int' />
   <xsd:element name='Surname' minOccurs='0' type='xsd:string' />
   <xsd:element name='GivenName' minOccurs='0' type='xsd:string' />
   <xsd:element name='DepartmentID' minOccurs='0' type='xsd:int' />
   <xsd:element name='Street' minOccurs='0' type='xsd:string' />
   <xsd:element name='City' minOccurs='0' type='xsd:string' />
   <xsd:element name='State' minOccurs='0' type='xsd:string' />
   <xsd:element name='Country' minOccurs='0' type='xsd:string' />
   <xsd:element name='PostalCode' minOccurs='0' type='xsd:string' />
   <xsd:element name='Phone' minOccurs='0' type='xsd:string' />
   <xsd:element name='Status' minOccurs='0' type='xsd:string' />
   <xsd:element name='SocialSecurityNumber' minOccurs='0'
   type='xsd:string' />
   <xsd:element name='Salary' minOccurs='0' type='xsd:decimal' />
   <xsd:element name='StartDate' minOccurs='0' type='xsd:date' />
   <xsd:element name='TerminationDate' minOccurs='0' type='xsd:date' />
   <xsd:element name='BirthDate' minOccurs='0' type='xsd:date' />
   <xsd:element name='BenefitHealthInsurance' minOccurs='0'
   type='xsd:boolean' />
   <xsd:element name='BenefitLifeInsurance' minOccurs='0'
   type='xsd:boolean' />
   <xsd:element name='BenefitDayCare' minOccurs='0' type='xsd:boolean' />
   <xsd:element name='Sex' minOccurs='0' type='xsd:string' />
   ```

   b. 创建一个 DISH 服务以充当 SOAP 服务的代理并生成 WSDL 文档。

   ```
   CREATE SERVICE "SASoapTest_DNET"
   TYPE 'DISH'
   GROUP "SASoapTest"
   FORMAT 'DNET'
   ```



```
AUTHORIZATION OFF
SECURE OFF
USER DBA;
```

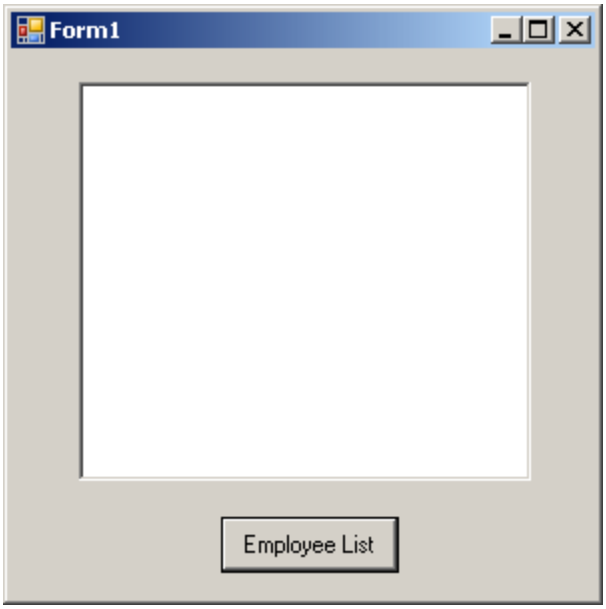
SOAP 和 DISH 服务必须为 DNET 格式。在本示例中，创建 SOAP 服务时省略了 FORMAT 子句。这样，SOAP 服务将从 DISH 服务继承 DNET 格式。

3. 启动 Microsoft Visual Studio。请注意，本示例使用 .NET Framework 2.0 中的函数。
   a. 使用 Visual C# 新建一个 [Windows Application] 项目。

      将出现一个空的窗体。
   b. 从 [**Project**] 菜单中，选择 [**Add Web Reference**]。
   c. 在 [**Add Web Reference**] 页面的 [**URL**] 字段中，输入以下 URL： **http://localhost:80/demo/SASoapTest_DNET**.
   d. 单击 [**Go**]。

      此时显示可供用于 SASoapTest_DNET 的方法的列表。您将看到 EmployeeList 方法。
   e. 单击 [**Add Reference**] 完成操作。

      [**Solution Explorer**] 窗口即会显示新的 Web 引用。
   f. 在 Visual Studio [**Toolbox**] 中，向窗体添加一个列表框和一个按钮，如下图所示。



   g. 将按钮文本重命名为 **Employee List**。
   h. 双击 [**Employee List**]，然后为按钮的 click 事件添加以下代码。

```
int sqlCode;

listBox1.Items.Clear();

localhost.SASoapTest_DNET proxy = new localhost.SASoapTest_DNET();
```

```
DataSet results = proxy.EmployeeList(out sqlCode);
DataTableReader dr = results.CreateDataReader();
while (dr.Read())
{
  for (int i = 0; i < dr.FieldCount; i++)
  {
    string columnName = "(" + dr.GetDataTypeName(i)
                      + ")"
                      + dr.GetName(i);
    if (dr.IsDBNull(i))
    {
      listBox1.Items.Add(columnName + "=(null)");
    }
    else {
      System.TypeCode typeCode =
          System.Type.GetTypeCode(dr.GetFieldType(i));
      switch (typeCode)
      {
        case System.TypeCode.Int32:
          Int32 intValue = dr.GetInt32(i);
          listBox1.Items.Add(columnName + "="
              + intValue);
          break;
        case System.TypeCode.Decimal:
          Decimal decValue = dr.GetDecimal(i);
          listBox1.Items.Add(columnName + "="
              + decValue.ToString("c"));
          break;
        case System.TypeCode.String:
          string stringValue = dr.GetString(i);
          listBox1.Items.Add(columnName + "="
              + stringValue);
          break;
        case System.TypeCode.DateTime:
          DateTime dateValue = dr.GetDateTime(i);
          listBox1.Items.Add(columnName + "="
              + dateValue);
          break;
        case System.TypeCode.Boolean:
          Boolean boolValue = dr.GetBoolean(i);
          listBox1.Items.Add(columnName + "="
              + boolValue);
          break;
        case System.TypeCode.DBNull:
          listBox1.Items.Add(columnName
              + "=(null)");
          break;
        default:
          listBox1.Items.Add(columnName
              + "=(unsupported)");
          break;
      }
    }
  }
  listBox1.Items.Add("");
}
dr.Close();
```

此示例旨在说明应用程序开发人员可以对可用的数据类型信息进行精细控制。

i. 构建并运行程序。

来自 Web 服务器的 XML 响应包括已设置格式的结果集。已设置格式的结果集的第一行如下所示。

```
<row>
 <EmployeeID>102</EmployeeID>
 <ManagerID>501</ManagerID>
 <Surname>Whitney</Surname>
 <GivenName>Fran</GivenName>
 <DepartmentID>100</DepartmentID>
 <Street>9 East Washington Street</Street>
 <City>Cornwall</City>
 <State>NY</State>
 <Country>USA</Country>
 <PostalCode>02192</PostalCode>
 <Phone>6175553985</Phone>
 <Status>A</Status>
 <SocialSecurityNumber>017349033</SocialSecurityNumber>
 <Salary>45700.000</Salary>
 <StartDate>1984-08-28-05:00</StartDate>
 <TerminationDate xsi:nil="true" />
 <BirthDate>1958-06-05-05:00</BirthDate>
 <BenefitHealthInsurance>1</BenefitHealthInsurance>
 <BenefitLifeInsurance>1</BenefitLifeInsurance>
 <BenefitDayCare>0</BenefitDayCare>
 <Sex>F</Sex>
</row>
```

有关 XML 结果集响应的几点注意事项。

- 所有列数据都转换为数据的字符串表示形式。
- 包含日期和/或时间信息的列包括来自 Web 服务器 UTC 的偏移。在此示例中，偏移为 -05:00，即与 UTC 西部（在此例中为北美东部标准时间）时间相差 5 小时。
- 只含有日期的列的格式如下：[yyyy-mm-dd-HH:MM] 或 [yyyy-mm-dd+HH:MM]。将向字符串后添加区域偏移（-HH:MM 或 +HH:MM）后缀。
- 只含有时间的列的格式如下：[hh:mm:ss.nnn-HH:MM] 或 [hh:mm:ss.nnn+HH:MM]。将向字符串后添加时区偏移（-HH:MM 或 +HH:MM）后缀。
- 含有日期和时间的列的格式如下：[yyyy-mm-ddThh:mm:ss.nnn-HH:MM] 或 [yyyy-mm-ddThh:mm:ss.nnn+HH:MM]。请注意，使用字母 'T' 分隔日期和时间。将向字符串后添加时区偏移（-HH:MM 或 +HH:MM）后缀。

列表框将以 (type)column name=value 对的形式显示 EmployeeList 结果集。处理结果集第一行的结果如下所示。

```
(Int32)EmployeeID=102
(Int32)ManagerID=501
(String)Surname=Whitney
(String)GivenName=Fran
(Int32)DepartmentID=100
(String)Street=9 East Washington Street
(String)City=Cornwall
(String)State=New York
(String)Country=USA
(String)PostalCode=02192
(String)Phone=6175553985
(String)Status=A
(String)SocialSecurityNumber=017349033
(String)Salary=$45,700.00
(DateTime)StartDate=28/08/1984 0:00:00 AM
(DateTime)TerminationDate=(null)
(DateTime)BirthDate=05/06/1958 0:00:00 AM
```



```
(Boolean)BenefitHealthInsurance=True
(Boolean)BenefitLifeInsurance=True
(Boolean)BenefitDayCare=False
(String)Sex=F
```

有关结果的几点注意事项。

- 将包含空值的列返回为 DBNull 类型。
- Salary 金额已被转换为客户端的货币形式。
- 假设包含日期但不包含时间值的列的时间为 00:00:00 或午夜。

教程：将数据类型与 JAX-WS 一起使用

以下教程演示了如何使用 Java API for XML Web Services (JAX-WS) 访问 Web 服务。

如果您已完成了更早版本的 JAX-WS 教程，那么这些步骤中的一些已经执行过了。

在开始之前，您需要有 Sun 提供的 JAX-WS 工具。如果您的系统上没有安装此软件包，您必须下载 JAX-WS 工具并安装它们。要下载 JAX-WS 工具，请访问 http://java.sun.com/webservices/。单击 JAX-WS 的链接。您将进入 Java API for XML Web Services 页面。单击 [**Download Now**] 链接。在您已经下载了软件包之后，将它安装在您的系统上。

此示例是使用 JAX-WS 2.1.3 for Windows 开发的。

应将从 JAX-WS 访问的 SQL Anywhere SOAP Web 服务声明为 CONCRETE 格式。

♦ 创建 SOAP 和 DISH 服务

1. 在命令提示符处，运行以下命令以启动个人 Web 服务器。将 *samples-dir* 替换为示例数据库的实际位置。`-xs http(port=80)` 选项会告知数据库服务器接受来自端口 80 的请求。如果在端口 80 上已有 Web 服务器在运行，则将另一个端口号（如 8080）用于本教程。

   ```
   dbeng11 -xs http(port=80) samples-dir\demo.db
   ```

2. 启动 Interactive SQL，并以 DBA 身份连接到 SQL Anywhere 示例数据库。执行以下语句：

   a. 定义一个列出 Employees 表的某些列的存储过程。

   ```
   CREATE PROCEDURE ListEmployees()
   RESULT (
    EmployeeID            INTEGER,
    Surname               CHAR(20),
    GivenName             CHAR(20),
    StartDate             DATE,
    TerminationDate       DATE )
   BEGIN
     SELECT EmployeeID, Surname, GivenName,
       StartDate, TerminationDate
     FROM Employees;
   END;
   ```

   b. 定义一个调用此存储过程的 SOAP 服务。

   ```
   CREATE SERVICE "WS/EmployeeList2"
   TYPE 'SOAP'
   FORMAT 'CONCRETE' EXPLICIT ON
   DATATYPE OUT
   AUTHORIZATION OFF
   SECURE OFF
   USER DBA
   AS CALL ListEmployees();
   ```

   EXPLICIT 子句仅能用于 CONCRETE 类型的 SOAP 或 DISH 服务。在此示例中，EXPLICIT ON 表示相应的 DISH 服务应生成说明 EmployeeList2Dataset 对象的 XML 模式。此选项仅影响生成的 WSDL 文档。在前面的 JAX-WS 教程中，我们看过使用 EXPLICIT OFF 的示例。请参见“教程：从 JAX-WS 访问 Web 服务”一节第 836 页。

   DATATYPE OUT 表示显式数据类型信息在 XML 结果集响应中生成。如果已指定 DATATYPE OFF，那么所有数据将作为字符串键入。此选项不影响生成 WSDL 文档。

由于授权已关闭，因此任何人都可使用此服务而无需提供用户名和口令。命令在用户 DBA 下运行。这种安排虽然简单，但不安全。

c. 创建一个 DISH 服务以充当 SOAP 服务的代理并生成 WSDL 文档。

```
CREATE SERVICE "WSDish"
TYPE 'DISH'
FORMAT 'CONCRETE'
GROUP "WS"
AUTHORIZATION OFF
SECURE OFF
USER DBA;
```

SOAP 和 DISH 服务必须都是 CONCRETE 格式。由于 EmployeeList2 服务在 WS 组中，因此会包括 GROUP 子句。

3. 查看 DISH 服务自动创建的 WSDL。为此，请打开 Web 浏览器并浏览至以下 URL：http://localhost:80/demo/WSDish。DISH 服务会自动生成一个 WSDL 文档，该文档将出现在浏览器窗口中。

特别是要观察被公开的 EmployeeList2Dataset 对象，因为该服务是 CONCRETE 格式并且 EXPLICIT 为 ON。在随后的步骤中，**wsimport** 应用程序使用此信息为该服务生成一个 SOAP 1.1 客户端接口。

```
<s:complexType name="EmployeeList2Dataset">
<s:sequence>
<s:element name="rowset">
  <s:complexType>
  <s:sequence>
  <s:element name="row" minOccurs="0" maxOccurs="unbounded">
    <s:complexType>
    <s:sequence>
    <s:element minOccurs="0" maxOccurs="1" name="EmployeeID"
        nillable="true" type="s:int" />
    <s:element minOccurs="0" maxOccurs="1" name="Surname"
        nillable="true" type="s:string" />
    <s:element minOccurs="0" maxOccurs="1" name="GivenName"
        nillable="true" type="s:string" />
    <s:element minOccurs="0" maxOccurs="1" name="StartDate"
        nillable="true" type="s:date" />
    <s:element minOccurs="0" maxOccurs="1" name="TerminationDate"
        nillable="true" type="s:date" />
    </s:sequence>
    </s:complexType>
  </s:element>
  </s:sequence>
  </s:complexType>
</s:element>
</s:sequence>
</s:complexType>
```

♦ 为这些 Web 服务生成 JAX-WS 接口

1. 在本示例中，Java API for XML Web Services (JAX-WS) 和 Sun Java 1.6.0 JDK 安装在驱动器 *C:* 上。设置 PATH 环境变量，使其包含 JAX-WS 二进制文件和 JDK 二进制文件。二进制文件位于以下目录中。

```
c:\Sun\jaxws-ri\bin
c:\Sun\SDK\jdk\bin
```


2. 在命令提示符处，设置 CLASSPATH 环境变量。

```
SET classpath=.;C:\Sun\jaxws-ri\lib\jaxb-api.jar
;C:\Sun\jaxws-ri\lib\jaxws-rt.jar
```

3. 下一步是生成调用 Web 服务所需的接口。

   在同一命令提示符处，创建一个新的项目目录，并使该目录成为您的当前目录。在此目录中执行以下命令。

```
wsimport -keep -Xendorsed "http://localhost:80/demo/WSDish"
```

   该 wsimport 工具从给定的 URL 中检索 WSDL 文档，并生成为其定义接口的 Java 文件，然后编译 Java 文件。

   keep 选项将指示 wsimport 不要删除 *.java* 文件。如果不使用此选项，则会在生成相应的 *.class* 文件后删除这些文件。如果保存这些文件，会更易于检查接口的组成。

   Xendorsed 选项允许您使用 JDK6 版的 JAX-WS 2.1 API。

   此命令完成后，会在您的当前目录中生成一个名为 *localhost\demo\ws* 的新子目录结构，其中包含以下 Java 文件，以及每个 Java 文件编译后的 *.class* 版本。

```
EmployeeList2.java
EmployeeList2Dataset.java
EmployeeList2Response.java
FaultMessage.java
ObjectFactory.java
package-info.java
WSDish.java
WSDishSoapPort.java
```

♦ 使用生成的 JAX-WS 接口

1. 将以下 Java 源代码保存到 *SASoapDemo2.java* 中。确保该文件位于含有 *localhost* 子目录（由 wsimport 工具生成）的同一目录中。

```
// SASoapDemo2.java illustrates a web service client that
// calls the WSDish service and prints out the data.

import java.util.*;
import javax.xml.ws.*;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import javax.xml.datatype.*;
import localhost.demo.ws.*;

public class SASoapDemo2
{
  public static void main( String[] args )
  {
    try {
      WSDish service = new WSDish();

      Holder<EmployeeList2Dataset> response =
          new Holder<EmployeeList2Dataset>();
      Holder<Integer> sqlcode = new Holder<Integer>();

      WSDishSoapPort port = service.getWSDishSoap();
```

```
            // This is the SOAP service call to EmployeeList2
            port.employeeList2( response, sqlcode );

            EmployeeList2Dataset result = response.value;
            EmployeeList2Dataset.Rowset rowset = result.getRowset();

            List<EmployeeList2Dataset.Rowset.Row> rows = rowset.getRow();

            String fieldType;
            String fieldName;
            String fieldValue;
            Integer fieldInt;
            XMLGregorianCalendar fieldDate;

            for ( int i = 0; i < rows.size(); i++ ) {
              EmployeeList2Dataset.Rowset.Row row = rows.get( i );

              fieldType = row.getEmployeeID().getDeclaredType().getSimpleName();
              fieldName = row.getEmployeeID().getName().getLocalPart();
              fieldInt = row.getEmployeeID().getValue();
              System.out.println( "(" + fieldType + ")" + fieldName +
                                  "=" + fieldInt );

              fieldType = row.getSurname().getDeclaredType().getSimpleName();
              fieldName = row.getSurname().getName().getLocalPart();
              fieldValue = row.getSurname().getValue();
              System.out.println( "(" + fieldType + ")" + fieldName +
                                  "=" + fieldValue );

              fieldType = row.getGivenName().getDeclaredType().getSimpleName();
              fieldName = row.getGivenName().getName().getLocalPart();
              fieldValue = row.getGivenName().getValue();
              System.out.println( "(" + fieldType + ")" + fieldName +
                                   "=" + fieldValue );

              fieldType = row.getStartDate().getDeclaredType().getSimpleName();
              fieldName = row.getStartDate().getName().getLocalPart();
              fieldDate = row.getStartDate().getValue();
              System.out.println( "(" + fieldType + ")" + fieldName +
                                  "=" + fieldDate );

              if ( row.getTerminationDate() == null ) {
                fieldType = "unknown";
                fieldName = "TerminationDate";
                fieldDate = null;
              } else {
                fieldType =
                  row.getTerminationDate().getDeclaredType().getSimpleName();
                fieldName = row.getTerminationDate().getName().getLocalPart();
                fieldDate = row.getTerminationDate().getValue();
              }
              System.out.println( "(" + fieldType + ")" + fieldName +
                                  "=" + fieldDate );
              System.out.println();
            }
          }
          catch (Exception x) {
            x.printStackTrace();
          }
        }
      }
```


如果您选择使用一个不同的端口号（如 8080）来启动 web 服务器，则必须将 [import localhost] 源行更改为如下形式：

```
import localhost._8080.demo.ws.*;
```

2. 编译 *SASoapDemo2.java*。

```
javac SASoapDemo2.java
```

3. 运行编译过的类文件。

```
java SASoapDemo2
```

当应用程序向 Web 服务器发送请求时，它会收到一个包含 EmployeeList2Result（带有含有几个行条目的行集）的 XML 结果集响应。此响应中还包含执行查询的 sqlcode 结果。响应的示例如下所示。

```
<tns:EmployeeList2Response>
 <tns:EmployeeList2Result xsi:type='tns:EmployeeList2Dataset'>
  <tns:rowset>
    <tns:row>...</tns:row>
.
.
.
    <tns:row>...</tns:row>
   </tns:rowset>
 </tns:EmployeeList2Result>
 <tns:sqlcode>0</tns:sqlcode>
</tns:EmployeeList2Response>
```

行集中的每行均以类似如下的格式发送。

```
<tns:row>
   <tns:EmployeeID xsi:type="xsd:int">1751</tns:EmployeeID>
   <tns:Surname xsi:type="xsd:string">Ahmed</tns:Surname>
   <tns:GivenName xsi:type="xsd:string">Alex</tns:GivenName>
   <tns:StartDate xsi:type="xsd:date">1994-07-12-04:00</tns:StartDate>
   <tns:TerminationDate xsi:type="xsd:date">2008-04-18-04:00
       </tns:TerminationDate>
</tns:row>
```

请注意，列名和数据类型包括在内。

使用代理
您可以通过使用记录 XML 消息流量的代理软件来观察上面所示的响应。代理软件置身于客户端应用程序和 Web 服务器之间。

SASoapDemo2 应用程序将以 (type)column name=value 对的形式显示 EmployeeList2 结果集。将生成如下所示的几行输出。

```
(Integer)EmployeeID=102
(String)Surname=Whitney
(String)GivenName=Fran
(XMLGregorianCalendar)StartDate=1984-08-28-04:00
(unknown)TerminationDate=null

(Integer)EmployeeID=105
(String)Surname=Cobb
```

```
(String)GivenName=Matthew
(XMLGregorianCalendar)StartDate=1985-01-01-05:00
(unknown)TerminationDate=null
.
.
.
(Integer)EmployeeID=1740
(String)Surname=Nielsen
(String)GivenName=Robert
(XMLGregorianCalendar)StartDate=1994-06-24-04:00
(unknown)TerminationDate=null

(Integer)EmployeeID=1751
(String)Surname=Ahmed
(String)GivenName=Alex
(XMLGregorianCalendar)StartDate=1994-07-12-04:00
(XMLGregorianCalendar)TerminationDate=2008-04-18-04:00
```

请注意，仅当 TerminationDate 列值为非 NULL 时，才发送它。Java 应用程序用于检测 TerminationDate 列何时不出现。对于本示例，Employees 表的最后一行被改动，因此将设置一个非 NULL 终止日期。

还应注意，日期值包含一个 UTC 时间偏移值。在上面的示例数据中，服务器位于北美东部时区。如果不执行夏令时，该时区的时间比 UTC 时间早 5 个小时 (-05:00)；如果执行夏令时，该时区的时间比 UTC 时间早 4 个小时 (-04:00)。

要进一步了解 SASoapDemo2 应用程序中使用的 Java 方法，值得研究一下 java.sun.com Web 站点 (http://java.sun.com/javaee/5/docs/api/) 上的 javax.xml.bind.JAXBElement 类文档。

使用 iAnywhere WSDL 编译器

假定有一个介绍 Web 服务的 WSDL 源，则 iAnywhere WSDL 编译器将生成一组包含在应用程序中的 Java 代理类、C# 代理类或 SQL Anywhere 的 SQL SOAP 客户端过程。

由 WSDL 编译器生成的 Java 或 C# 代理类用于与 QAnywhere 一起使用。这些类将 Web 服务操作以方法调用的方式公开。所生成的类包括：

- 主要服务绑定类（此类继承自移动 Web 服务运行库中的 ianywhere.qanywhere.ws.WSBase）。
- WSDL 文件中所指定的每个复杂类型的代理类。

有关生成的代理类的信息，请参见：

- .NET： “用于 Web 服务的 QAnywhere .NET (.NET 2.0)”一节 《QAnywhere》
- Java： “用于 Web 服务的 QAnywhere Java API”一节 《QAnywhere》

WSDL 编译器支持 WSDL 1.1 和 SOAP 1.1（通过 HTTP 和 HTTPS）。

语法

wsdlc [*options*] *wsdl-uri*

wsdl-uri:

它是对 WSDL（Web Services Description Language，简称 Web 服务描述语言）源（URL 或文件）的说明。

选项：

- **-h** 显示帮助文本。
- **-v** 显示详细信息。
- **-o *output-directory*** 指定生成文件的输出目录。
- **-l *language*** 指定生成文件的语言。该语言为 **java**、**cs**（对于 C#）或 **sql** 之一。这些选项必须以小写字母指定。
- **-d** 当联系 iAnywhere 客户支持时，显示可能有用的调试信息。

特定于 Java 的选项：

- **-p *package*** 指定一个程序包名。这使您可以覆盖缺省的程序包名。

特定于 C# 的选项：

- **-n *namespace*** 指定命名空间。使用此选项可在所选命名空间中封装生成的类。

特定于 SQL 的选项：

- **-f *filename*** （必需）指定写入 SQL 语句的 SQL 输出文件的名称。此操作将覆盖所有现有同名文件。
- **-p=*prefix*** 为已生成的函数或过程名指定前缀。缺省的前缀是后跟一个句点的服务名（例如，"WSDish."）。

- **-x** 生成过程定义而不是函数定义。

这是介绍 Web 服务的 WSDL 文件的名称。

WSDLC 不扩展表示结构或数组的复杂参数。这样的参数都被注释掉了，以允许数据库服务器自动创建给定的过程或函数而无需修改。但是，要使 SOAP 操作正常执行，必需分析并手工撰写这样的参数。该进程可能需要使用 SQL Anywhere XMLELEMENT 函数以及 XMLATTRIBUTES 参数来生成复杂的 XML 结构表示形式。

创建 Web 服务客户端函数和过程

SQL Anywhere 数据库可提供和使用 Web 服务。这些 Web 服务可以是通过 Internet 提供的标准 Web 服务，也可以由 SQL Anywhere 数据库提供（只要 Web 服务与客户端过程或函数不在同一数据库中即可）。

SQL Anywhere 可充当 HTTP 和 SOAP 两种 Web 服务客户端。此功能通过存储函数和存储过程提供。

使用以下 SQL 语句创建和操作客户端函数和过程：

- “CREATE FUNCTION 语句（Web 服务）”一节 《SQL Anywhere 服务器 - SQL 参考》
- “CREATE PROCEDURE 语句（Web 服务）”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER FUNCTION 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER PROCEDURE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “DROP FUNCTION 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “DROP PROCEDURE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

例如，用于创建 Web 服务客户端函数的 CREATE FUNCTION 和 CREATE PROCEDURE 语句的语法如下所示：

CREATE FUNCTION [*owner.*]*procedure-name* **(** [*parameter*, ...] **)**
RETURNS *data-type*
URL *url-string*
[*proc-attributes*]

CREATE PROCEDURE [*owner.*]*procedure-name* **(** [*parameter*, ...] **)**
URL *url-string*
[*proc-attributes*]

此语法的关键在于 URL 子句，该子句用于提供想要过程访问的 Web 服务的 URL。URL 子句的基本语法如下：

url-string :
'{ **HTTP** | **HTTPS** | **HTTPS_FIPS** }*://*[*user*:*password*@]*hostname*[:*port*][*/path*]'

可选的用户和口令信息允许您访问要求验证的 Web 服务。主机名可以是提供 Web 服务的计算机的名称或 IP 地址。

仅当服务器正在监听缺省端口号以外的端口号时，才需要端口号。HTTP 服务的缺省端口号为 80，HTTPS 服务的缺省端口号为 443。

路径用于标识服务器上的资源或 Web 服务。

请求可发送到任何 Web 服务，无论是另一个 SQL Anywhere 数据库提供的 Web 服务还是通过 Internet 提供的 Web 服务。如果 Web 服务由同一数据库服务器提供，它不得位于客户端函数所在的数据库中。尝试访问同一数据库中的 Web 服务将导致错误 [403 禁止]。

由于感叹号用于参数替代，因此出现在过程定义中任何位置的字符串中的感叹号都必须转义。请参见 “! 字符转义”一节第 872 页。

常见子句

可以用来提供关于过程调用的更多详细信息的其它子句如下所示：

proc-attributes :
[**TYPE** { **'HTTP**[**:**{ **GET** | **POST**[**:***MIME-type*] | **PUT**[**:***MIME-type*] | **DELETE** | **HEAD** }]' |
'SOAP[**:**{ **RPC** | **DOC** }]' }]
[**NAMESPACE** *namespace-string*]
[**CERTIFICATE** *certificate-string*]
[**CLIENTPORT** *clientport-string*]
[**PROXY** *proxy-string*]
[**SET** *protocol-option-string*]

TYPE 子句非常重要，因为它告知 SQL Anywhere 如何设置对 Web 服务提供商的请求的格式。可使用标准 SOAP 类型 RPC 和 DOC。可使用标准 HTTP 方法 GET 和 POST，分别指定为 HTTP:GET 和 HTTP:POST。指定 HTTP 则表示 HTTP:POST。

如果选择类型 SOAP，SQL Anywhere 会自动将请求的格式设置为 SOAP 请求所需标准格式的 XML 文档。由于 SOAP 请求始终为 XML 文档，因此如果选择了类型 SOAP，则始终会隐式使用 HTTP POST 请求将 SOAP 请求文档发送到服务器。指定 SOAP 隐含 SOAP:RPC。

Web 服务客户端函数和过程的名称

在生成外发 SOAP 请求时，将过程名用作 SOAP 操作名。此外，任何参数的名称也会出现在 SOAP 请求封装的标记名称中。由于 SOAP 服务器期望看到这些名称，因此正确地指定这些名称就称为定义 SOAP 存储过程的重要部分。除了应用于 SQL Anywhere 中过程名和函数名的一般规则外，这一事实也对 SOAP 过程和函数的名称进行了限制。

以下语句将创建一个名为 MyOperation 的 SOAP 存储过程：

```
CREATE PROCEDURE MyOperation ( a INTEGER, b CHAR(128) )
URL 'HTTP://localhost'
TYPE 'SOAP:DOC';
```

当调用此过程时（例如由下面的语句调用），会生成一个 SOAP 请求：

```
CALL MyOperation( 123, 'abc' );
```

该过程名将出现在请求主体的标记 [<m:MyOperation>] 中。该过程的两个参数 a 和 b 分别成为 [<m:a>] 和 [<m:b>]。

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:m="http://localhost">
  <SOAP-ENV:Body>
    <m:MyOperation>
      <m:a>123</m:a>
      <m:b>abc</m:b>
    </m:MyOperation>
  </SOAP-ENV:Body>
 </SOAP-ENV:Envelope>
```

命名空间 URI

所有 SOAP 请求都需要一个方法命名空间 URI。服务器端的 SOAP 处理器使用此 URI 来了解请求的消息主体中各种实体的名称。

创建 SOAP:DOC 或 SOAP:RPC 类型的 SOAP 函数或过程时，可能需要先指定一个命名空间 URI，然后调用才能成功。可从 WSDL 描述文档或服务的其它可用文档获得所需的命名空间值。

NAMESPACE 子句仅适用于 SOAP 函数和过程。缺省命名空间值为过程的 URI，直到（但不包括）可选的路径组成部分，并排除任何用户和口令值。

HTTPS 请求

要发布安全 HTTPS 请求，客户端必须有权访问服务器证书或用于为服务器证书签名的证书。此证书将告知 SQL Anywhere 如何加密请求。被定向到不安全服务器的请求可能会被重定向到安全服务器时，也需要证书值。

有两种方式可提供证书信息。可以将证书放置在文件中并提供文件名，或者将整个证书作为字符串值提供。您不能同时使用这两种方式。

证书属性作为字符串值提供时，字符串的构造是由分号分隔的各个 key=value 对：

certificate-string :
{ **file=***filename* | **certificate=***string* } ; **company=***company* ; **unit=***company-unit* ; **name=** *common-name*

可以使用以下键：

键	缩写	说明
file		证书的文件名。
certificate	cert	证书本身，采用 Base64 编码。
company	co	证书中指定的公司。
unit		证书中指定的公司单位。
name		证书中指定的公用名。

例如，以下语句所创建的过程会向与客户端位于同一计算机上的 Web 服务发出安全请求：

```
CREATE PROCEDURE test()
URL 'HTTPS://localhost/myservice'
CERTIFICATE 'file=C:\srv_cert.id;co=iAnywhere;
             unit=SA;name=JohnSmith';
```

由于没有提供 TYPE 子句，因此假定请求为 SOAP:RPC 类型。服务器的公共证书位于文件 *C:\srv_cert.id* 中。

客户端端口

通过防火墙访问 Web 服务时，有时需要告知 SQL Anywhere 在打开与服务器的连接时要使用哪些端口。通常，端口号是动态获取的，并且您应依赖该缺省行为，除非防火墙限制对特定范围端口的访问。

ClientPort 选项指定客户端应用程序通过 TCP/IP 进行通信所用的端口号。您可以指定单个端口号（或各端口号的组合），以及端口号的范围（如下例所示）：

```
CREATE PROCEDURE test ()
URL 'HTTPS://localhost/myservice'
CLIENTPORT '5040,5050-5060,5070';
```

最好指定一系列端口号或端口号的范围。如果指定单个端口号，则一次只能维护一个连接。实际上，即使是在关闭这个连接之后，也会有一个长达几分钟的超时期，在此期间无法建立与同一远程服务器和端口的任何新连接。如果指定端口号的列表和/或范围，应用程序会一直尝试这些端口号，直到找到一个可成功捆绑到的端口号。

此特性与 ClientPort 网络协议选项类似。请参见“ClientPort 协议选项 [CPORT]”一节 《SQL Anywhere 服务器 - 数据库管理》。

使用代理

有些 Web 服务请求可能需要通过代理服务器发出。如果是这样，则必须使用 PROXY 子句提供代理服务器的 URL。

该值的格式与 URL 子句的相同，但忽略了所有用户、口令或路径值：

proxy-string :
'{ HTTP | HTTPS }://[*user*:*password*@]*hostname*[**:***port*][*/path*]'

如果指定了代理服务器，SQL Anywhere 将设置请求的格式，并使用提供的代理 URL 将其发送到代理服务器。代理服务器会将请求转发到最终目标，获取响应，然后将响应转发回 SQL Anywhere。

记录 Web 服务客户端过程

来自 Web 服务客户端的信息（包括 HTTP 请求和传输数据）会记录到 **Web 服务客户端日志文件**。您可以通过启动数据库服务器以及 -zoc 服务器选项来记录到该文件，或者通过使用 sa_server_option 系统过程设置 WebClientLogging 服务器属性来记录到该文件。请参见“-zoc 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》和“sa_server_option 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

修改 HTTP 标头

使用 CREATE PROCEDURE 语句创建 Web 服务过程时，如果指定了没有冒号 (:) 和值的 HTTP HEADER 名称，HTTP 客户端应用程序将取消该标头。如果包括冒号但未提供值，HTTP 客户端应用程序可包括标头名称，但没有值。例如：

```
CREATE PROCEDURE suds(...)
TYPE 'SOAP:RPC'
URL '...'
HEADER 'SOAPAction\nDate\nFrom:';
```

在此示例中，由 SQL Anywhere 自动生成的 Action 和 Date HTTP 标头被取消，From 标头存在但没有值。

修改自动生成的标头可能会导致意外的结果。通常自动生成以下 HTTP 标头，未经认真考虑，请勿修改。

HTTP 标头	说明
Accept-Charset	始终自动生成。如果更改或删除，可能导致意外的数据转换错误。

HTTP 标头	说明
ASA-Id	始终自动生成。确保客户端未与自身连接（即与同一服务器连接），可能会导致死锁。
Authorization	当 URL 包含证书时自动生成。如果更改或删除，可能导致请求失败。仅支持 BASIC 授权。当通过 HTTPS 进行连接时只应包括用户和口令信息。
Connection	Connection:close 始终自动生成。客户端不支持持久连接。不应进行更改，否则连接可能会挂起。
Host	始终自动生成。如果 HTTP/1.1 客户端未提供 Host 标头，则需要 HTTP/1.1 服务器响应 400 错误请求。
Transfer-Encoding	在块模式下发布请求时自动生成。当客户端使用 CHUNK 模式时，删除此标头或删除块值将导致失败。
Content-Length	当发布请求且不在块模式下时自动生成。需要此标头告知服务器主体的内容长度。如果内容长度错误，则连接可能挂起或发生数据丢失。

处理返回值和结果集

可以使用存储函数或存储过程进行 Web 服务客户端调用。如果从函数进行调用，则函数的返回类型必须是字符数据类型，如 CHAR、VARCHAR 或 LONG VARCHAR。返回值是 HTTP 响应的主体。不包含任何标头信息。关于请求的附加信息（包括 HTTP 状态信息）由过程返回。这样，在需要访问此附加信息时最好使用过程。

SOAP 过程

来自 SOAP 函数的响应是一个包含 SOAP 响应的 XML 文档。

由于 SOAP 响应是结构化的 XML 文档，因此，SQL Anywhere 在缺省情况下会尝试利用此信息，并构造一个更实用的结果集。返回的响应文档中的每个顶级标记都将被抽取并用作列名。每个此类标记下的子树的内容将用作该列的行值。

例如，假设 SOAP 响应如下所示，SQL Anywhere 将构造出所显示的数据集：

```
<SOAP-ENV:Envelope
  xmlns:SOAPSDK1="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAPSDK2="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:SOAPSDK3="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
     <ElizaResponse xmlns:SOAPSDK4="SoapInterop">
        <Eliza>Hi, I'm Eliza. Nice to meet you.</Eliza>
     <ElizaResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Eliza
Hi, I'm Eliza.Nice to meet you.

在本示例中，响应文档使用 SOAP-ENV:Body 标记中出现的 ElizaResponse 标记分隔。

结果集的列数与顶级标记数量相同。因为 SOAP 响应中只有一个顶级标记，所以此结果集只有一列。这单个顶级标记 Eliza 将成为列名。

XML 处理工具

使用内置 Open XML 处理功能也可以访问 XML 结果集中的信息（包括 SOAP 响应）。

以下示例使用 OPENXML 过程来抽取 SOAP 响应的各部分。此示例使用 Web 服务将 SYSWEBSERVICE 表的内容公开为 SOAP 服务：

```
CREATE SERVICE get_webservices
TYPE 'SOAP'
AUTHORIZATION OFF
USER DBA
AS SELECT * FROM SYSWEBSERVICE;
```

以下存储函数（必须在另一个 SQL Anywhere 数据库中创建）发出对此 Web 服务的调用。该函数的返回值是整个 SOAP 响应文档。响应采用 .NET 数据集格式，因为 DNET 是缺省的 SOAP 服务格式。


```
CREATE FUNCTION get_webservices()
RETURNS LONG VARCHAR
URL 'HTTP://localhost/get_webservices'
TYPE 'SOAP:DOC';
```

以下语句演示了如何使用 OPENXML 过程抽取结果集的两列。*service_name* 和 *secure_required* 列分别指明哪些 SOAP 服务是安全的以及何处需要 HTTPS。

```
SELECT *
FROM openxml( get_webservices(), '//row' )
WITH ("Name"     char(128) 'service_name',
      "Secure?" char(1)   'secure_required' );
```

此语句通过选择 *row* 节点的子节点生效。WITH 子句基于相关的两个元素构造结果集。假定只存在 *get_webservices* 服务，此函数将返回以下结果集：

名称	Secure?
get_webservices	N

有关 SQL Anywhere 提供的 XML 处理工具的详细信息，请参见“在数据库中使用 XML”《SQL Anywhere 服务器 - SQL 的用法》。

其它类型的过程

其它类型的过程在一个两列式结果集中返回关于响应的所有信息。此结果集包含响应状态、标头信息和主体。第一列名为 Attribute，第二列名为 Value。两列的数据类型都是 LONG VARCHAR。

结果集为每个响应标头字段都提供一个数据行，并且分别为 HTTP 状态行（Status 属性）和响应主体（Body 属性）提供了一个数据行。

以下示例代表一个典型响应：

属性	值
Status	HTTP /1.0 200 OK
Body	<!DOCTYPE HTML ...><HTML> ...</HTML>
Content-Type	text/html
Server	GWS/2.1
Content-Length	2234
日期	Mon, 18 Oct 2004, 16:00:00 GMT

从结果集中选择

SELECT 语句用于从结果集中检索值。一旦被检索到，这些值就可存储在表中或用于设置变量。

```
CREATE PROCEDURE test( INOUT parm CHAR(128) )
URL 'HTTP://localhost/test'
TYPE 'HTTP';
```

由于其类型为 HTTP，因此该过程将返回上节所述的两列式结果集。第一列中是属性名，第二列中是属性值。关键字位于 HTTP 响应标头字段中。Body 属性包含消息主体，消息主体通常是一个 HTML 文档。

有一种方法是将结果集插入到表中，如下所示：

```
CREATE TABLE StoredResults(
     Attribute LONG VARCHAR,
     Value     LONG VARCHAR
);
```

可将结果集按如下所示插入到此表中：

```
INSERT INTO StoredResults SELECT *
FROM test('Storing into a table')
WITH (Attribute LONG VARCHAR, Value LONG VARCHAR);
```

可以根据 SELECT 语句的通常语法添加子句。例如，如果只需要结果集中特定的一行，可以添加 WHERE 子句将选择的结果仅限定为一行：

```
SELECT Value
FROM test('Calling test for the Status Code')
WITH (Attribute LONG VARCHAR, Value LONG VARCHAR)
WHERE Attribute = 'Status';
```

此语句仅从结果集中选择状态信息。它可用于验证调用是否成功。

使用参数

与其它类型的函数或过程一样，充当 Web 服务客户端的存储函数和存储过程可使用参数进行声明。除非在参数替代期间使用，否则这些参数值将作为 HTTP 或 SOAP 请求的一部分传递。

此外，调用存储函数或存储过程时，可使用参数替换该函数或过程主体中的占位符。如果不存在特定变量的占位符，参数及其值将作为请求的一部分传递。以这种方式用于替代的参数和值不会作为 Web 服务请求的一部分传递。

传递的参数

除非在参数替代期间使用，否则函数或过程的所有参数都将作为 Web 服务请求的一部分传递。传递的格式取决于 Web 服务请求的类型。

HTTP 请求

HTTP:GET 类型请求的参数被编码到 URL 中。例如，以下过程将声明两个参数：

```
CREATE PROCEDURE test ( a INTEGER, b CHAR(128) )
URL 'HTTP://localhost/myservice'
TYPE 'HTTP:GET';
```

如果此过程使用 123 和 'xyz' 这两个值调用，则用于该请求的 URL 如下所示：

```
HTTP://localhost/myservice?a=123&b=xyz
```

如果类型为 HTTP:POST，则参数及其相应值成为请求主体的一部分。对于这两个参数及其相应值，将在 HTTP 请求主体中的标头后显示以下文本：

```
a=123&b=xyz
```

SOAP 请求

按照 SOAP 规范的要求，传递给 SOAP 请求的参数将捆绑为请求主体的一部分：

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:m="http://localhost">
  <SOAP-ENV:Body>
    <m:test>
      <m:a>123</m:a>
      <m:b>abc</m:b>
    </m:test>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

参数替代

每次执行存储过程或存储函数时，将用该过程或函数的已声明参数自动替代过程或函数定义中的占位符。任何包含后面跟有已声明参数名称的感叹号 (!) 的子字符串都会被替换为该参数的值。

例如，以下过程定义允许将整个 URL 作为一个参数传递。每次调用此过程时可使用不同的值。

```
CREATE PROCEDURE test ( url CHAR(128) )
URL '!url'
TYPE 'HTTP:POST';
```

例如，接下来可以使用下面的过程：

```
CALL test ( 'HTTP://localhost/myservice' );
```

隐藏用户和口令值

参数替代的一个实用价值是避免将敏感值（如用户名和口令）作为 Web 服务客户端函数或过程定义的一部分。当此类值被指定为过程或函数定义中的文字时，它们将存储在系统表中，这样就使数据库的所有用户随时都可以访问它们。将这些值作为参数传递就避免了这一问题。

例如，以下过程定义将用户名和口令以纯文本形式包含在其中：

```
CREATE PROCEDURE test ()
URL 'HTTP://dba:sql@localhost/myservice';
```

为避免此问题，可以将用户名和口令声明为参数。这样就可以只在调用过程时才提供用户名和口令值。例如：

```
CREATE PROCEDURE test ( uid CHAR(128), pwd CHAR(128) )
URL 'HTTP://!uid:!pwd@localhost/myservice';
```

如下所示调用此过程：

```
CALL test ( 'dba', 'sql' );
```

作为另一个示例，可使用参数替代传递来自某文件的加密证书并将它们传递到存储过程或存储函数：

```
CREATE PROCEDURE secure( cert LONG VARCHAR )
URL 'https://localhost/secure'
TYPE 'HTTP:GET'
CERTIFICATE 'cert=!cert;company=test;unit=test;name=RSA Server';
```

调用此过程时，以字符串形式提供证书。在以下示例调用中，从某文件读取证书。这只是为了举例说明，因为可使用 CERTIFICATE 子句的 **file=** 关键字直接从文件读取证书。

```
CALL secure( xp_read_file('install-dir\bin32\rsaserver.id') );
```

! 字符转义

由于感叹号 (!) 用于标识 Web 服务客户端存储函数和存储过程上下文中要进行参数替代的占位符，因此，只要想在任何过程属性字符串中包含此字符，就必须对其进行转义。为此，要在感叹号前面加上另一个感叹号作为前缀。这样，Web 服务客户端或 Web 服务函数定义中的字符串内出现的所有 !! 都将替换为 !。

用作占位符的参数名必须只包含字母数字字符。此外，占位符后必须跟有一个非字母数字字符以避免多义性。没有匹配参数名的占位符将自动删除。例如，将不会用参数 size 替代以下过程中的占位符：

```
CREATE PROCEDURE orderitem ( size CHAR(18) )
URL 'HTTP://salesserver/order?size=!sizeXL'
TYPE 'SOAP:RPC';
```

但是，!sizeXL 会始终删除，因为它是一个没有匹配参数的有效占位符。

参数数据类型转换

不属于字符或二进制数据类型的参数值将在添加到请求之前先转换为字符串表示形式。此过程相当于将值设置为字符类型。该转换是在调用函数或过程时根据数据类型格式设置选项设置进行的。特别是，转换可能会受到 precision、scale 和 timestamp_format 之类选项的影响。

使用结构化数据类型

XML 返回值

使用函数或过程将 SQL Anywhere 服务器作为 Web 服务客户端与 Web 服务连接。

对于简单的返回数据类型，结果集内的字符串表示形式就足够了。在这种情况下，可能有必要使用存储过程。

当返回复杂的数据（如数组或结构）时，使用 Web 服务函数是较好的选择。对于函数声明，RETURN 子句可指定 XML 数据类型。可使用 OPENXML 对返回的 XML 进行分析以抽取有用的元素。

请注意，返回的 XML 数据（如 dateTime）将在结果集内逐字显示。例如，如果结果集内包括 TIMESTAMP 列，它的格式将为 XML dateTime 字符串 (2006-12-25T12:00:00.000-05:00) 而非字符串 (2006-12-25 12:00:00.000)。

XML 参数值

支持将 SQL Anywhere XML 数据类型用作 Web 服务函数和过程内的参数。对于简单类型，当生成 SOAP 请求主体时将自动构造参数元素。但对于 XML 类型的参数，将无法执行此操作，因为元素的 XML 表示形式可能需要提供其它数据的属性。因此，当为数据类型为 XML 的参数生成 XML 时，根元素名称必须与该参数名相对应。

```
<inputHexBinary xsi:type="xsd:hexBinary">414243</inputHexBinary>
```

该 XML 类型演示如何将参数作为 hexBinary XML 类型发送。SOAP 端点期望该参数名（在 XML 术语中称为根元素名称）为 "inputHexBinary"。

Cookbook 常量

构造复杂的结构和数组需要了解 SQL Anywhere 如何引用命名空间的知识。此处列出的前缀与为 SQL Anywhere SOAP 请求封装生成的命名空间的声明相对应。

SQL Anywhere XML 前缀	命名空间
xsd	http://www.w3.org/2001/XMLSchema
xsi	http://www.w3.org/2001/XMLSchema-instance
SOAP-ENC	http://schemas.xmlsoap.org/soap/encoding/
m	NAMESPACE 子句中定义的命名空间

复杂数据类型示例

以下三个示例演示了如何使用表示数组、结构和结构数组的参数创建 Web 服务客户端函数。该示例旨在向 Microsoft SOAP ToolKit 3.0 Round 2 互操作性测试服务器 (http://mssoapinterop.org/stkV3) 发布请求。Web 服务函数将分别与名为 echoFloatArray、echoStruct 和 echoStructArray 的 SOAP 操作（或 RPC 函数名称）进行通信。用于互操作性测试的公共命名空间为 "http://soapinterop.org/"，这样，只需将 URL 子句更改为所选的 SOAP 端点，给定函数便可针对替代互操作性服务器进行测试。

所有三个示例都使用一个表来生成 XML 数据。以下过程介绍如何设置该表。

```
CREATE LOCAL TEMPORARY TABLE SoapData
(
    seqno INT DEFAULT AUTOINCREMENT,
    i INT,
    f FLOAT,
    s LONG VARCHAR
) ON COMMIT PRESERVE ROWS;

INSERT INTO SoapData (i,f,s)
VALUES (99,99.999,'Ninety-Nine');

INSERT INTO SoapData (i,f,s)
VALUES (199,199.999,'Hundred and Ninety-Nine');
```

以下三个函数将 SOAP 请求发送到互操作性服务器。请注意，此示例向 Microsoft Interop 服务器发出请求：

```
CALL sa_make_object('function', 'echoFloatArray');
ALTER FUNCTION echoFloatArray( inputFloatArray XML )
RETURNS XML
URL 'http://mssoapinterop.org/stkV3/Interop.wsdl'
HEADER 'SOAPAction:"http://soapinterop.org/"'
NAMESPACE 'http://soapinterop.org/';

CALL sa_make_object('function', 'echoStruct');
ALTER FUNCTION echoStruct( inputStruct XML )
RETURNS XML
URL 'http://mssoapinterop.org/stkV3/Interop.wsdl'
HEADER 'SOAPAction:"http://soapinterop.org/"'
NAMESPACE 'http://soapinterop.org/';

CALL sa_make_object('function', 'echoStructArray');
ALTER FUNCTION echoStructArray( inputStructArray XML )
RETURNS XML
URL 'http://mssoapinterop.org/stkV3/Interop.wsdl'
HEADER 'SOAPAction:"http://soapinterop.org/"'
NAMESPACE 'http://soapinterop.org/';
```

最终将显示三个示例语句及其参数的 XML 表示形式：

1. 以下示例中的参数表示一个数组。

   ```
   SELECT echoFloatArray(
       XMLELEMENT( 'inputFloatArray',
          XMLATTRIBUTES( 'xsd:float[]' as "SOAP-ENC:arrayType" ),
           (
           SELECT XMLAGG( XMLELEMENT( 'number', f ) ORDER BY seqno )
           FROM SoapData
           )
       )
   );
   ```

   存储过程 echoFloatArray 将以下 XML 发送到互操作性服务器。

   ```
   <inputFloatArray SOAP-ENC:arrayType="xsd:float[2]">
   <number>99.9990005493164</number>
   <number>199.998992919922</number>
   </inputFloatArray>
   ```

   来自互操作性服务器的响应如下所示。

```
'<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<SOAP-ENV:Envelope
  xmlns:SOAPSDK1="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAPSDK2="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:SOAPSDK3="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAPSDK4:echoFloatArrayResponse
      xmlns:SOAPSDK4="http://soapinterop.org/">
      <Result SOAPSDK3:arrayType="SOAPSDK1:float[2]"
        SOAPSDK3:offset="[0]"
        SOAPSDK2:type="SOAPSDK3:Array">
        <SOAPSDK3:float>99.9990005493164</SOAPSDK3:float>
        <SOAPSDK3:float>199.998992919922</SOAPSDK3:float>
      </Result>
    </SOAPSDK4:echoFloatArrayResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>'
```

如果该响应存储在变量中，则可使用 OPENXML 进行分析。

```
SELECT * FROM openxml( resp,'//*:Result/*' )
WITH ( varFloat FLOAT 'text()' );
```

varFloat
99.9990005493
199.9989929199

2. 以下示例中的参数表示一个结构。

```
SELECT echoStruct(
    XMLELEMENT('inputStruct',
        (
        SELECT XMLFOREST( s as varString,
                          i as varInt,
                          f as varFloat )
        FROM SoapData
        WHERE seqno=1
        )
    )
);
```

存储过程 echoStruct 将以下 XML 发送到互操作性服务器。

```
<inputStruct>
  <varString>Ninety-Nine</varString>
  <varInt>99</varInt>
  <varFloat>99.9990005493164</varFloat>
</inputStruct>
```

来自互操作性服务器的响应如下所示。

```
'<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<SOAP-ENV:Envelope
  xmlns:SOAPSDK1="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAPSDK2="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:SOAPSDK3="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
```



```
  <SOAP-ENV:Body
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAPSDK4:echoStructResponse
      xmlns:SOAPSDK4="http://soapinterop.org/">
      <Result href="#id1"/>
    </SOAPSDK4:echoStructResponse>
    <SOAPSDK5:SOAPStruct
      xmlns:SOAPSDK5="http://soapinterop.org/xsd"
      id="id1"
      SOAPSDK3:root="0"
      SOAPSDK2:type="SOAPSDK5:SOAPStruct">
      <varString>Ninety-Nine</varString>
      <varInt>99</varInt>
      <varFloat>99.9990005493164</varFloat>
    </SOAPSDK5:SOAPStruct>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>'
```

如果该响应存储在变量中，则可使用 OPENXML 进行分析。

```
SELECT * FROM openxml( resp,'//*:Body/*:SOAPStruct' )
WITH (
varString LONG VARCHAR 'varString',
varInt INT 'varInt',
varFloat FLOAT 'varFloat' );
```

varString	varInt	varFloat
Ninety-Nine	99	99.9990005493

3. 以下示例中的参数表示一个结构数组。

```
SELECT echoStructArray(
    XMLELEMENT( 'inputStructArray',
        XMLATTRIBUTES( 'http://soapinterop.org/xsd' AS "xmlns:q2",
                       'q2:SOAPStruct[2]' AS "SOAP-ENC:arrayType" ),
            (
            SELECT XMLAGG(
                XMLElement('q2:SOAPStruct',
                XMLFOREST( s as varString,
                           i as varInt,
                           f as varFloat )
                )
            ORDER BY seqno
            )
        FROM SoapData
        )
    )
);
```

存储过程 echoFloatArray 将以下 XML 发送到互操作性服务器。

```
<inputStructArray xmlns:q2="http://soapinterop.org/xsd"
  SOAP-ENC:arrayType="q2:SOAPStruct[2]">
  <q2:SOAPStruct>
    <varString>Ninety-Nine</varString>
    <varInt>99</varInt>
    <varFloat>99.9990005493164</varFloat>
  </q2:SOAPStruct>
  <q2:SOAPStruct>
    <varString>Hundred and Ninety-Nine</varString>
```

```
    <varInt>199</varInt>
    <varFloat>199.998992919922</varFloat>
  </q2:SOAPStruct>
</inputStructArray>
```

来自互操作性服务器的响应如下所示。

```
'<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<SOAP-ENV:Envelope
  xmlns:SOAPSDK1="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAPSDK2="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:SOAPSDK3="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAPSDK4:echoStructArrayResponse
      xmlns:SOAPSDK4="http://soapinterop.org/">
      <Result xmlns:SOAPSDK5="http://soapinterop.org/xsd"
        SOAPSDK3:arrayType="SOAPSDK5:SOAPStruct[2]"
        SOAPSDK3:offset="[0]" SOAPSDK2:type="SOAPSDK3:Array">
        <SOAPSDK5:SOAPStruct href="#id1"/>
        <SOAPSDK5:SOAPStruct href="#id2"/>
      </Result>
    </SOAPSDK4:echoStructArrayResponse>
    <SOAPSDK6:SOAPStruct
      xmlns:SOAPSDK6="http://soapinterop.org/xsd"
      id="id1"
      SOAPSDK3:root="0"
      SOAPSDK2:type="SOAPSDK6:SOAPStruct">
      <varString>Ninety-Nine</varString>
      <varInt>99</varInt>
      <varFloat>99.9990005493164</varFloat>
    </SOAPSDK6:SOAPStruct>
    <SOAPSDK7:SOAPStruct
      xmlns:SOAPSDK7="http://soapinterop.org/xsd"
      id="id2"
      SOAPSDK3:root="0"
      SOAPSDK2:type="SOAPSDK7:SOAPStruct">
      <varString>Hundred and Ninety-Nine</varString>
      <varInt>199</varInt>
      <varFloat>199.998992919922</varFloat>
    </SOAPSDK7:SOAPStruct>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>'
```

如果该响应存储在变量中，则可使用 OPENXML 进行分析。

```
SELECT * FROM openxml( resp,'//*:Body/*:SOAPStruct' )
WITH (
varString LONG VARCHAR 'varString',
varInt INT 'varInt',
varFloat FLOAT 'varFloat' );
```

varString	varInt	varFloat
Ninety-Nine	99	99.9990005493
Hundred and Ninety-Nine	199	199.9989929199

处理变量

HTTP 请求中的变量来自两个来源之一。第一个来源是 URL，URL 中可能包含查询字符串，查询字符串中会包含各种 name=value 对。HTTP GET 请求是以这种方式进行格式设置的。以下是包含查询字符串的 URL 示例。

```
http://localhost/gallery?picture=sunset.jpg
```

第二个来源是通过 URL 路径提供。将 URL PATH 设置为 ON 或 ELEMENTS 都会使服务名后的路径部分被解释为变量值。此选项让 URL 看起来像是在请求特定目录中的某文件（与基于文件的传统 Web 站点上的情况一样），而不是在请求存储在数据库中的内容。以下是一个示例。

```
http://localhost/gallery/sunset.jpg
```

此 URL 看起来像是在请求目录 gallery 中的文件 *sunset.jpg*。实际上，gallery 服务将此字符串作为参数接收（或许使用它从数据库表中检索图片）。

HTTP 请求中传递的参数取决于 URL PATH 的设置。

- **OFF** 不允许在服务名后添加路径参数。
- **ON** 服务名后的所有路径元素都赋予给变量 URL。
- **ELEMENTS** 按照斜线字符将 URL 路径的其余部分分割成数量最多为 10 个的一组元素。这些值被赋予给变量 URL1、URL2、URL3、……、URL10。如果提供的值少于 10 个，则将其余变量设置为 NULL。指定 10 个以上变量会导致出错。

除了定义位置不同外，各变量间没有任何差异。所有 HTTP 变量的访问和使用方式都相同。例如，访问变量值（如 url1）的方式与访问显示为查询一部分的参数（如 **?picture=sunset.jpg**）的方式相同。

访问变量

访问变量的方式主要有两种。第一种方式是在服务声明语句中引用变量。例如，以下语句将多个变量的值传递给 ShowTable 存储过程：

```
CREATE SERVICE ShowTable
TYPE 'RAW'
AUTHORIZATION ON
AS CALL ShowTable( :user_name, :table_name, :limit, :start );
```

另一种方式是使用处理请求的存储过程中的内置函数 NEXT_HTTP_VARIABLE 和 HTTP_VARIABLE。如果不知道定义了哪些变量，可以使用 NEXT_HTTP_VARIABLE 查找。HTTP_VARIABLE 函数返回变量值。

NEXT_HTTP_VARIABLE 函数允许您遍历已定义变量的名称。首次调用该函数时，将传入 NULL 值。这将返回一个变量的名称。如果连续调用它，并且每次都传入上一个变量名，则将返回下一个变量名。在将最后一个变量的名称传递给此函数时，它将返回 NULL。

以这种方式遍历变量名可确保每个变量名都刚好返回一次。但值的返回顺序不一定与它们在请求中出现的顺序相同。此外，如果再次遍历这些变量名，它们可能会以另一个不同的顺序返回。

若要获取每个变量的值，请使用 HTTP_VARIABLE 函数。第一个参数是变量的名称。其它参数是可选参数。如果为同一变量提供了多个值，则函数在仅被提供一个参数时会只返回第一个值。提供整数作为第二个参数使您可以检索其它值。

第三个参数允许您从包含多个部分的请求中检索变量标头字段值。提供标头字段的名称以检索其值。例如，以下 SQL 语句将检索三个变量值，然后检索 image 变量的标头字段值。

```
SET v_id = HTTP_VARIABLE( 'ID' );
SET v_title = HTTP_VARIABLE( 'Title' );
SET v_descr = HTTP_VARIABLE( 'descr' );

SET v_name = HTTP_VARIABLE( 'image', NULL, 'Content-Disposition' );
SET v_type = HTTP_VARIABLE( 'image', NULL, 'Content-Type' );
SET v_image = HTTP_VARIABLE( 'image', NULL, '@BINARY' );
```

以下是使用 HTTP_VARIABLE 函数检索变量相关值的一个示例。它是上文中所述的 ShowSalesOrderDetail 服务的修改后版本。

```
CREATE PROCEDURE ShowDetail()
BEGIN
    DECLARE v_customer_id LONG VARCHAR;
    DECLARE v_product_id LONG VARCHAR;
    SET v_customer_id = HTTP_VARIABLE( 'customer_id' );
    SET v_product_id = HTTP_VARIABLE( 'product_id' );
    CALL ShowSalesOrderDetail( v_customer_id, v_product_id );
END;
```

以下是调用存储过程的服务：

```
CREATE SERVICE ShowDetail
TYPE 'HTML'
URL PATH OFF
AUTHORIZATION OFF
USER DBA
AS CALL ShowDetail();
```

若要测试该服务，请打开 Web 浏览器并输入以下 URL：

http://localhost:80/demo/ShowDetail?product_id=300&customer_id=101

有关参数传递的详细信息，请参见“了解如何解释 URL”一节第 828 页和“Web 服务函数”一节《SQL Anywhere 服务器 - SQL 参考》。

处理 HTTP 标头

服务器端

创建 HTTP Web 服务客户端过程时，此子句用于添加、修改或删除 HTTP 请求标头条目。标头说明与 RFC2616 超文本传输协议—HTTP/1.1 中指定的格式，以及用于 ARPA Internet 文本消息的 RFC822 标准极其相似，其中包括只能为 HTTP 标头指定可打印的 ASCII 字符，且这些字符不区分大小写这一事实。以下是关于 HTTP 标头说明的一些要点：

- 标头/值对可分别用 \n（指定换行 (<LF>)）或 \x0d\n（指定回车和换行 (<CR><LF>)）进行分隔。
- 由于标头与标头值之间是用冒号 (:) 分隔，因此标头中不能包含冒号。
- 标头后跟 :\n 或行尾时，指定一个不含任何值的标头，标头后不接冒号或值时，也是如此。例如，`HEADER 'Date'` 指定不应包含 Date 标头。取消标头或其值会引起意外的结果。请参见“修改 HTTP 标头”一节第 866 页。
- 支持合并较长的标头值，但前提是在 \n 后紧跟一个或多个空格。例如，以下 HEADER 说明和生成的 HTTP 输出在语义上是等效的：

```
HEADER 'heading1: This long value\n is a really long value for heading1\n
heading2:shortvalue'

HEADER 'heading1:This long value is a really long value for
heading1<CR><LF>
heading2:shortvalue<CR><LF>'
```

- 多个连续空格（包括可合并的空格）会生成单个空格。
- 此子句支持参数替代。

此示例显示如何添加静态的用户定义标头：

```
CREATE PROCEDURE http_client()
  URL 'http://localhost/getdata'
  TYPE 'http:get'
  HEADER 'UserHeader1:value1\nUserHeader2:value2';
```

此示例显示如何添加替代参数的用户定义标头：

```
CREATE PROCEDURE http_client( headers LONG VARCHAR )
  URL 'http://localhost/getdata'
  TYPE 'http:get'
  HEADER '!headers';

CALL http_client( 'NewHeader1:value1\nNewHeader2:value2' );
```

客户端

可将 NEXT_HTTP_HEADER 和 HTTP_HEADER 函数结合使用来获得 HTTP 请求中的标头。NEXT_HTTP_HEADER 函数将遍历包含在请求中的 HTTP 标头并返回下一个 HTTP 标头名。如果用 NULL 调用该函数，该函数会返回第一个标头的名称。通过向该函数传递上一个标头的名称来检索后续的标头。当用最后一个标头的名称调用该函数时，该函数返回 NULL。

重复调用该函数将会返回所有的标头字段且刚好都只返回一次，但返回顺序不一定与它们在 HTTP 请求中出现的顺序相同。

HTTP_HEADER 函数将返回指定 HTTP 标头字段的值或 NULL（如果不是从 HTTP 服务调用）。当通过 Web 服务处理 HTTP 请求时，将使用该函数。如果给定字段名的标头不存在，则会返回值 NULL。

下表列出了一些典型的 HTTP 标头和值。

标头名	标头值
Accept	image/gif、image/x-xbitmap、image/jpeg、image/pjpeg、application/x-shockwave-flash、application/vnd.ms-excel、application/vnd.ms-powerpoint、application/msword、*/*
Accept-Language	en-us
UA-CPU	x86
Accept-Encoding	gzip、deflate
User-Agent	Mozilla/4.0（兼容；MSIE 7.0；Windows NT 5.2；WOW64；SV1；.NET CLR 2.0.50727）
Host	localhost
Connection	Keep-Alive
@HttpMethod	GET
@HttpURI	/demo/ShowHTTPHeaders
@HttpVersion	HTTP/1.1

若要获取每个标头的值，请使用 NEXT_HTTP_HEADER 函数获取标头的名称，然后使用 HTTP_HEADER 函数获取其值。下面举例说明了如何进行此操作。

```
CREATE PROCEDURE HTTPHeaderExample()
RESULT ( html_string LONG VARCHAR )
BEGIN
    declare header_name long varchar;
    declare header_value long varchar;
    declare table_rows XML;
    set header_name  = NULL;
    set table_rows = NULL;
header_loop:
    LOOP
        SET header_name = NEXT_HTTP_HEADER( header_name );
        IF header_name IS NULL THEN
            LEAVE header_loop
        END IF;
        SET header_value = HTTP_HEADER( header_name );
        -- Format header name and value into an HTML table row
        SET table_rows = table_rows ||
            XMLELEMENT( name "tr",
                XMLATTRIBUTES( 'left' AS "align",
                               'top' AS "valign" ),
                XMLELEMENT( name "td", header_name ),
```



```
                XMLELEMENT( name "td", header_value ) );

    END LOOP;
    SELECT XMLELEMENT( name "table",
                XMLATTRIBUTES( '' AS "BORDER",
                              '10' AS "CELLPADDING",
                               '0' AS "CELLSPACING" ),
                XMLELEMENT( name "th",
                    XMLATTRIBUTES( 'left' AS "align",
                                   'top'  AS "valign" ),
                                   'Header Name' ),
                XMLELEMENT( name "th",
                    XMLATTRIBUTES( 'left' AS "align",
                                   'top'  AS "valign" ),
                                   'Header Value' ),
                table_rows );
END;
```

此示例将标头名和标头值转换为一个 HTML 表。可定义以下服务以表明此示例过程如何运行。

```
CREATE SERVICE ShowHTTPHeaders
TYPE 'RAW'
AUTHORIZATION OFF
USER DBA
AS CALL HTTPHeaderExample();
```

若要测试该服务，请打开 Web 浏览器并输入以下 URL：

http://localhost:80/demo/ShowHTTPHeaders

要设置正在处理的请求的状态码（或响应码），使用 @HttpStatus 特殊标头。请参见“sa_set_http_header 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关标头处理的详细信息，请参见“Web 服务函数”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用 SOAP 服务

为阐明 Web 服务的众多特性，先以一个简单的华氏温度到摄氏温度转换器作为示例服务开始介绍。

♦ 设置简单的 Web 服务服务器

1. 创建数据库。

```
dbinit ftc
```

2. 使用此数据库启动一个服务器。

```
dbsrv11 -xs http(port=8082) -n ftc ftc.db
```

3. 使用 Interactive SQL 连接到该服务器。

```
dbisql -c "UID=DBA;PWD=sql;ENG=ftc"
```

4. 使用 Interactive SQL 创建一个 Web 服务。

```
CREATE SERVICE FtoCService
TYPE 'SOAP'
FORMAT 'XML'
AUTHORIZATION OFF
USER DBA
AS CALL FToCConvertor( :temperature );
```

5. 定义此服务要调用的用来执行从华氏温度转换为摄氏温度所需计算的存储过程：

```
CREATE PROCEDURE FToCConvertor( temperature FLOAT )
BEGIN
    SELECT ROUND((temperature - 32.0) * 5.0 / 9.0, 5)
    AS answer;
END;
```

此时，已有一个 SQL Anywhere Web 服务服务器在运行且已准备好处理请求。该服务器在端口 8082 上监听 SOAP 请求。

那么您将如何测试该 SOAP 请求服务器？最简单的方法是使用另一个 SQL Anywhere 数据库服务器传达 SOAP 请求并检索响应。

♦ 发送和接收 SOAP 请求

1. 创建另一个数据库以供与第二台服务器一同使用。

```
dbinit ftc_client
```

2. 使用此数据库启动个人服务器。

```
dbeng11 ftc_client.db
```

3. 使用另一个 Interactive SQL 实例连接到个人服务器。

```
dbisql -c "UID=DBA;PWD=sql;ENG=ftc_client"
```

4. 使用 Interactive SQL 创建一个存储过程。

```
CREATE PROCEDURE FtoC( temperature FLOAT )
  URL 'http://localhost:8082/FtoCService'
  TYPE 'SOAP:DOC';
```

该 URL 子句用于引用 SOAP Web 服务。字符串 ['http://localhost:8082/FtoCService'] 用于指定要使用的 Web 服务的 URI。这是对监听端口 8082 的 Web 服务器的引用。

发出 Web 服务请求时使用的缺省格式是 'SOAP:RPC'。此示例中选择的格式为 'SOAP:DOC'，此格式与 'SOAP:RPC' 类似，但是它允许使用更丰富的一组数据类型。SOAP 请求始终作为 XML 文档发送。发送 SOAP 请求所用的机制是 'HTTP:POST'。

5. 您需要一个用于 FtoC 存储过程的包装，因此创建另一个存储过程。

```
CREATE PROCEDURE FahrenheitToCelsius( temperature FLOAT )
BEGIN
    DECLARE result LONG VARCHAR;
    DECLARE err INTEGER;
    DECLARE crsr CURSOR FOR
        CALL FtoC( temperature );

    OPEN crsr;
    FETCH crsr INTO result, err;
    CLOSE crsr;

    SELECT temperature, Celsius
    FROM OPENXML(result, '//tns:answer', 1, result)
         WITH ("Celsius" FLOAT 'text()');
END;
```

此存储过程充当对 Web 服务的调用的包装过程。FtoC 存储过程将返回一个此存储过程处理的结果集。该结果集是一个单独的 XML 字符串，与以下字符串类似。

```
<tns:rowset xmlns:tns="http://localhost/ftc/FtoCService">
 <tns:row>
  <tns:answer>100</tns:answer>
 </tns:row>
</tns:rowset>
```

OPENXML 函数用于分析返回的 XML，从而抽取摄氏温度值。

6. 调用该存储过程以发送请求并获得响应。

```
CALL FahrenheitToCelsius(212);
```

将出现华氏温度和对等的摄氏温度。

温度	摄氏
212	100

此时，已演示了一个在 SQL Anywhere Web 服务器上运行的简单 Web 服务。正如您所见，其它 SQL Anywhere 服务器可与此 Web 服务器进行通信。对这些服务器之间传送的 SOAP 请求和响应的内容几乎没有什么控制。在下一节中，您将了解如何通过添加您自己的 SOAP 标头来扩展这个简单的 Web 服务。

注意
Web 服务可由同一数据库服务器提供，但不得位于客户端函数所在的数据库中。尝试访问同一数据库中的 Web 服务将导致错误 [`403 Forbidden`]。

有关 SOAP 标头处理的信息，请参见“处理 SOAP 标头”一节第 887 页。

处理 SOAP 标头

在本节中，会将“使用 SOAP 服务”一节第 884 页中所介绍的简单 Web 服务扩展到处理 SOAP 标头。

如果您已执行上一节中概述的步骤，则可以跳过第 1 步到第 4 步，直接进行第 5 步。

♦ 创建 Web 服务服务器

1. 创建数据库。

```
dbinit ftc
```

2. 使用此数据库启动一个服务器。

```
dbsrv11 -xs http(port=8082) -n ftc ftc.db
```

3. 使用 Interactive SQL 连接到该服务器。

```
dbisql -c "UID=DBA;PWD=sql;ENG=ftc"
```

4. 使用 Interactive SQL 创建一个 Web 服务。

```
CREATE SERVICE FtoCService
TYPE 'SOAP'
FORMAT 'XML'
AUTHORIZATION OFF
USER DBA
AS CALL FToCConvertor( :temperature );
```

5. 定义此服务要调用的用来执行从华氏温度转换为摄氏温度所需计算的存储过程。与上一节中的示例不同，此示例包含其它语句用以处理特殊 SOAP 标头。如果您已完成了上一节中的示例，则将下面的 CREATE 更改为 ALTER，因为您现在要修改该存储过程。

```
CREATE PROCEDURE FToCConvertor( temperature FLOAT )
BEGIN
  DECLARE hd_key LONG VARCHAR;
  DECLARE hd_entry LONG VARCHAR;
  DECLARE alias LONG VARCHAR;
  DECLARE first_name LONG VARCHAR;
  DECLARE last_name LONG VARCHAR;
  DECLARE xpath LONG VARCHAR;
  DECLARE authinfo LONG VARCHAR;
  DECLARE namespace LONG VARCHAR;
  DECLARE mustUnderstand LONG VARCHAR;
header_loop:
  LOOP
    SET hd_key = NEXT_SOAP_HEADER( hd_key );
    IF hd_key IS NULL THEN
      -- no more header entries
      LEAVE header_loop;
    END IF;
    IF hd_key = 'Authentication' THEN
      SET hd_entry = SOAP_HEADER( hd_key );
      SET xpath = '/*:' || hd_key || '/*:userName';
      SET namespace = SOAP_HEADER( hd_key, 1,
                                   '@namespace' );
      SET mustUnderstand = SOAP_HEADER( hd_key, 1,
                                   'mustUnderstand' );
      BEGIN
```

```
            -- parse the XML returned in the SOAP header
            DECLARE crsr CURSOR FOR
              SELECT *
              FROM OPENXML( hd_entry, xpath )
                WITH ( alias LONG VARCHAR '@*:alias',
                first_name LONG VARCHAR '*:first/text()',
                last_name LONG VARCHAR '*:last/text()' );
            OPEN crsr;
            FETCH crsr INTO alias, first_name, last_name;
            CLOSE crsr;
          END;
          -- build a response header
          -- based on the pieces from the request header
          SET authinfo =
            XMLELEMENT( 'Authentication',
              XMLATTRIBUTES(
                namespace as xmlns,
                alias,
                mustUnderstand ),
                XMLELEMENT( 'first', first_name ),
                XMLELEMENT( 'last', last_name ) );
          CALL SA_SET_SOAP_HEADER( 'authinfo', authinfo );
        END IF;
      END LOOP header_loop;
      SELECT ROUND((temperature - 32.0) * 5.0 / 9.0, 5)
      AS answer;
    END;
```

可将 NEXT_SOAP_HEADER 和 SOAP_HEADER 函数结合使用来获得 SOAP 请求中的标头。NEXT_SOAP_HEADER 函数将遍历包含在请求中的 SOAP 标头并返回下一个 SOAP 标头名。如果用 NULL 调用该函数，该函数会返回第一个标头的名称。通过向 NEXT_SOAP_HEADER 函数传递上一个标头的名称来检索后续的标头。当用最后一个标头的名称调用该函数时，该函数返回 NULL。在该示例中执行 SOAP 标头检索的 SQL 代码如下所示。它会在最后返回 NULL 时退出循环。

```
SET hd_key = NEXT_SOAP_HEADER( hd_key );
    IF hd_key IS NULL THEN
      -- no more header entries
      LEAVE header_loop;
    END IF;
```

重复调用该函数将会返回所有的标头字段且仅返回一次，但不一定与其出现在 SOAP 请求中的顺序相同。

SOAP_HEADER 函数将返回指定的 SOAP 标头字段的值或 NULL（如果不是从 SOAP 服务调用）。当通过 Web 服务处理 SOAP 请求时，将使用该函数。如果给定字段名的标头不存在，则会返回值 NULL。

该示例将搜索名为 Authentication 的 SOAP 标头。当它找到此标头时，便会抽取整个 SOAP 标头的值以及 [@namespace] 和 [mustUnderstand] 属性的值。SOAP 标头值可能类似于下面的 XML 字符串：

```
<Authentication xmlns="SecretAgent" mustUnderstand="1">
  <userName alias="99">
    <first>Susan</first>
    <last>Hilton</last>
  </userName>
</Authentication>
```


对于此标头，[@namespace] 属性值将为：[SecretAgent]

同时，[mustUnderstand] 属性值将为：1

使用设置为 /*:Authentication/*:userName 的 XPath 字符串来通过 OPENXML 函数分析此 XML 字符串的内部。

```
SELECT *
FROM OPENXML( hd_entry, xpath )
  WITH ( alias LONG VARCHAR '@*:alias',
  first_name LONG VARCHAR '*:first/text()',
  last_name LONG VARCHAR '*:last/text()' );
```

通过使用上面所示的示例 SOAP 标头值，SELECT 语句将会创建一个如下所示的结果集：

alias	first_name	last_name
99	Susan	Hilton

在此结果集上会声明一个游标，且会将三个列值读取到三个变量中。此时，您获取了传递给 Web 服务的所有相关信息。您获取了华氏温度值以及一些通过 SOAP 标头传递给 Web 服务的其它属性。那么您要如何处理这些信息？

您可以检查所提供的名称和别名，以确定此人是否已被授权使用该 Web 服务。但此步骤并未在示例中表明。

存储过程中的下一步是以 SOAP 格式创建一个响应。可以按如下所示构建该 XML 响应：

```
SET authinfo =
  XMLELEMENT( 'Authentication',
    XMLATTRIBUTES(
      namespace as xmlns,
      alias,
      mustUnderstand ),
      XMLELEMENT( 'first', first_name ),
      XMLELEMENT( 'last', last_name ) );
```

这样会构建出以下 XML 字符串：

```
<Authentication xmlns="SecretAgent" alias="99"
                              mustUnderstand="1">
  <first>Susan</first>
  <last>Hilton</last>
</Authentication>
```

最后，为将 SOAP 响应返回到调用方，使用 SA_SET_SOAP_HEADER 存储过程：

```
CALL SA_SET_SOAP_HEADER( 'authinfo', authinfo );
```

与上一节的示例相同，最后一步是执行从华氏度转换到摄氏度的计算。

此时，您已经有一个运行的 SQL Anywhere Web 服务服务器可将温度从华氏度转换为摄氏度，这与上一节相同。但主要区别是，它还可以处理来自调用方的 SOAP 标头并将 SOAP 响应发送回调用方。

这仅是整个过程的一半。下一步是开发一个可发送 SOAP 请求并接收 SOAP 响应的示例客户端。

如果您已执行上一节中概述的步骤，则可以跳过第 1 步到第 3 步，直接进行第 4 步。

♦ 发送和接收 SOAP 标头

1. 创建另一个数据库以供与第二台服务器一同使用。

   ```
   dbinit ftc_client
   ```

2. 使用此数据库启动个人服务器。

   ```
   dbeng11 ftc_client.db
   ```

3. 使用另一个 Interactive SQL 实例连接到个人服务器。

   ```
   dbisql -c "UID=DBA;PWD=sql;ENG=ftc_client"
   ```

4. 使用 Interactive SQL 创建一个存储过程。

   ```
   CREATE PROCEDURE FtoC( temperature FLOAT,
       INOUT inoutheader LONG VARCHAR,
       IN inheader LONG VARCHAR )
     URL 'http://localhost:8082/FtoCService'
     TYPE 'SOAP:DOC'
     SOAPHEADER '!inoutheader!inheader';
   ```

   该 URL 子句用于引用 SOAP Web 服务。字符串 ['http://localhost:8082/FtoCService'] 用于指定要使用的 Web 服务的 URI。这是对监听端口 8082 的 Web 服务器的引用。

   发出 Web 服务请求时使用的缺省格式是 'SOAP:RPC'。此示例中选择的格式为 'SOAP:DOC'，此格式与 'SOAP:RPC' 类似，但是它允许使用更丰富的一组数据类型。SOAP 请求始终作为 XML 文档发送。发送 SOAP 请求所用的机制是 'HTTP:POST'。

   SQL Anywhere 客户端过程中的替代变量（inoutheader、inheader）必须是字母数字。如果 Web 服务客户端被声明为函数，则其所有参数都只能是 IN 模式（它们不能由被调用函数赋值）。因此，必须使用 OPENXML 或其它字符串函数抽取 SOAP 响应标头信息。

5. 您需要一个用于 FtoC 存储过程的包装，因此按如下所示创建另一个存储过程。与上一节中的示例不同，此示例包含其它语句，用于创建特殊 SOAP 标头、在 Web 服务调用中将其发送，以及处理来自 Web 服务器的响应。如果您已完成了上一节中的示例，则将下面的 CREATE 更改为 ALTER，因为您现在要修改该存储过程。

   ```
   CREATE PROCEDURE FahrenheitToCelsius( temperature FLOAT )
   BEGIN
     DECLARE io_header LONG VARCHAR;
     DECLARE in_header LONG VARCHAR;
     DECLARE result LONG VARCHAR;
     DECLARE err INTEGER;
     DECLARE crsr CURSOR FOR
       CALL FtoC( temperature, io_header, in_header );
     SET io_header =
       '<Authentication xmlns="SecretAgent" ' ||
                       'mustUnderstand="1">' ||
       '<userName alias="99">' ||
       '<first>Susan</first><last>Hilton</last>' ||
       '</userName>' ||
       '</Authentication>';
     SET in_header =
       '<Session xmlns="SomeSession">' ||
   ```



```
      '123456789' ||
      '</Session>';

    MESSAGE 'send, soapheader=' || io_header || in_header;
    OPEN crsr;
    FETCH crsr INTO result, err;
    CLOSE crsr;
    MESSAGE 'receive, soapheader=' || io_header;
    SELECT temperature, Celsius
    FROM OPENXML(result, '//tns:answer', 1, result)
           WITH ("Celsius" FLOAT 'text()');
END;
```

此存储过程充当对 Web 服务的调用的包装过程。此存储过程的功能已在上一节的示例基础上得到了增强。它创建两个 SOAP 标头。第一个 SOAP 标头如下所示。

```
<Authentication xmlns="SecretAgent"
                              mustUnderstand="1">
  <userName alias="99">
    <first>Susan</first>
    <last>Hilton</last>
  </userName></Authentication>
```

第二个 SOAP 标头如下所示。

```
<Session xmlns="SomeSession">123456789</Session>
```

打开游标时，会将 SOAP 请求发送到 Web 服务。

```
<Authentication xmlns="SecretAgent" alias="99"
                              mustUnderstand="1">
<first>Susan</first>
<last>Hilton</last>
</Authentication>
```

FtoC 存储过程返回一个此存储过程将处理的结果集。该结果集如下所示。

```
<tns:rowset xmlns:tns="http://localhost/ftc/FtoCService">
 <tns:row>
  <tns:answer>100</tns:answer>
 </tns:row>
</tns:rowset>
```

OPENXML 函数用于分析返回的 XML，从而抽取摄氏温度值。

6. 调用该存储过程以发送请求并获得响应：

```
CALL FahrenheitToCelsius(212);
```

将出现华氏温度和对等的摄氏温度。

温度	摄氏
212.0	100.0

可将 SQL Anywhere Web 服务客户端声明为函数或者过程。SQL Anywhere 客户端函数声明会将所有参数都有效限定于 in 模式（参数不能由被调用函数赋值）。调用 SQL Anywhere Web 服务函数将返回原始 SOAP 封装响应，而某过程会返回一个结果集。

SOAPHEADER 子句已被添加到 create/alter procedure/function 语句。SOAP 标头可声明为静态常量，也可使用参数替代机制进行动态设置。Web 服务客户端函数可定义一个或多个 in 模式替代参数，而 Web 服务客户端过程也可定义一个 inout 或 out 替代参数。因此，Web 服务客户端过程可返回 out（或 inout）替代参数中的响应 SOAP 标头。Web 服务函数必须分析响应 SOAP 封装以获得标头条目。

以下示例说明客户端如何指定发送多个带有参数的标头条目和接收响应 SOAP 标头数据。

```
CREATE PROCEDURE SoapClient(
    INOUT hd1 VARCHAR,
    IN hd2 VARCHAR,
    IN hd3 VARCHAR )
  URL 'localhost/some_endpoint'
  SOAPHEADER '!hd1!hd2!hd3';
```

注意

- hd1、hd2 和 hd3 均指定请求标头条目。hd1 还会返回所有响应标头条目的集合。
- 使用 SOAP 标头调用 SOAP 客户端时，SOAP 客户端将生成封装 SOAP 标头元素。如果 SOAPHEADER 值为 NULL，则不会生成任何 SOAP 标头元素。
- 如果未收到任何 SOAP 标头，则会将 hd1 设置为 NULL。
- 没有必要指定 hd1 的 INOUT 模式，因为参数的缺省模式就是 INOUT。
- 如果将多个替代参数指定为 OUT（或 INOUT）类型，则会导致下面的运行时错误：
  `'Expression has unsupported data type' SQLCODE=-624, ODBC 3 State-"HY000"`
- 仅可将一个显式用于 SOAPHEADER 的替代参数声明为 OUT。

限制

- 服务器端 SOAP 服务目前无法定义输入和输出 SOAP 标头要求。因此，SOAP 标头元数据在 DISH 服务的 WSDL 输出中不可用。这就意味着 SOAP 客户端工具箱无法为 SQL Anywhere SOAP 服务端点自动生成 SOAP 标头接口。
- 不支持 Soap 标头错误。

使用 MIME 类型

SQL Anywhere Web 服务客户端程序的 TYPE 子句或函数定义允许 MIME 类型说明。MIME 类型说明的值用于设置 Content-Type 请求标头并设置操作模式以允许仅调用单个参数填充请求的主体。处理完参数替换后进行 Web 服务存储过程（或函数）调用时，只能保留零个或一个参数。调用为空的或无参数的 Web 服务程序（替换后）将导致无主体的请求和内容长度为零。如果未指定 MIME 类型，将不更改该行为。参数名和值（允许多个参数）在 HTTP 请求的主体内进行 URL 编码。

一些典型 MIME 类型包括：

- text/plain
- text/html
- text/xml

以下步骤说明了 MIME 类型的设置。第一部分设置可用于测试 MIME 类型设置的 Web 服务。第二部分演示如何设置 MIME 类型。

♦ 创建 Web 服务服务器

1. 创建数据库。

```
dbinit echo
```

2. 使用此数据库启动一个服务器。

```
dbsrv11 -xs http(port=8082) -n echo echo.db
```

3. 使用 Interactive SQL 连接到该服务器。

```
dbisql -c "UID=DBA;PWD=sql;ENG=echo"
```

4. 使用 Interactive SQL 创建一个 Web 服务。

```
CREATE SERVICE EchoService
TYPE 'RAW'
USER DBA
AUTHORIZATION OFF
SECURE OFF
AS CALL Echo(:valueAsXML);
```

5. 定义此服务要调用的存储过程。

```
CREATE PROCEDURE Echo( parm LONG VARCHAR )
BEGIN
    SELECT parm;
END;
```

此时，已有一个 SQL Anywhere Web 服务服务器在运行且已准备好处理请求。该服务器正在监听端口 8082 上的 HTTP 请求。

要使用此 Web 服务器进行测试，可创建另一个 SQL Anywhere 数据库，然后启动并与之连接。以下步骤说明了如何执行此操作。

♦ 发送 HTTP 请求

1. 使用数据库创建实用程序另外创建一个要与 Web 服务客户端配合使用的数据库。

   ```
   dbinit echo_client
   ```

2. 继续使用 Interactive SQL，使用以下语句启动此数据库。

   ```
   START DATABASE 'echo_client.db'
   AS echo_client;
   ```

3. 现在，可使用以下语句连接到在该服务器回写时启动的数据库。

   ```
   CONNECT TO 'echo'
   DATABASE 'echo_client'
   USER 'DBA'
   IDENTIFIED BY 'sql';
   ```

4. 创建将与 EchoService Web 服务进行通信的存储过程。

   ```
   CREATE PROCEDURE setMIME(
     value LONG VARCHAR,
     mimeType LONG VARCHAR,
     urlSpec LONG VARCHAR
     )
   URL '!urlSpec'
   HEADER 'ASA-Id'
   TYPE 'HTTP:POST:!mimeType';
   ```

   该 URL 子句用于引用 Web 服务。出于说明目的，URL 将作为参数传递到 setMIME 过程。

   TYPE 子句指示 MIME 类型将作为参数传递到 setMIME 过程。发出 Web 服务请求时使用的缺省格式是 'SOAP:RPC'。为此 Web 服务请求所选择的格式是 'HTTP:POST'。

5. 调用该存储过程以发送请求并获得响应。所传递的值参数是 URL 编码形式的 `<hello>this is xml</hello>`。由于 SQL Anywhere Web 服务器可理解 URL 编码形式，所以媒体类型为 `application/x-www-form-urlencoded`。该 Web 服务的 URL 作为调用中的最后一个参数包括在内。

   ```
   CALL setMIME('valueAsXML=%3Chello%3Ethis%20is%20xml%3C/hello%3E',
       'application/x-www-form-urlencoded',
       'http://localhost:8082/EchoService');
   ```

   最后一个参数可指定在端口 8082 上监听的 Web 服务的 URI。

以下是发送到 Web 服务器的 HTTP 包的代表示例。

```
POST /EchoService HTTP/1.0
Date: Sun, 28 Jan 2007 04:04:44 GMT
Host: localhost
Accept-Charset: windows-1252, UTF-8, *
User-Agent: SQLAnywhere/11.0.0.1297
Content-Type: application/x-www-form-urlencoded; charset=windows-1252
Content-Length: 49
ASA-Id: 1055532613:echo_client:echo:968000
Connection: close

valueAsXML=%3Chello%3Ethis%20is%20xml%3C/hello%3E
```

以下是来自 Web 服务器的响应。

```
HTTP/1.1 200 OK
Server: SQLAnywhere/11.0.0.1297
Date: Sun, 28 Jan 2007 04:04:44 GMT
Expires: Sun, 28 Jan 2007 04:04:44 GMT
Content-Type: text/plain; charset=windows-1252
Connection: close

<hello>this is xml</hello>
```

Interactive SQL 显示的结果集如下所示。

属性	值
Status	HTTP /1.1 200 OK
Body	<hello>this is xml</hello>
Server	SQLAnywhere/11.0.0.1297
日期	Sun, 16 Dec 2007 04:04:44 GMT
Expires	Sun, 16 Dec 2007 04:04:44 GMT
Content-Type	text/plain; charset=windows-1252
Connection	close

使用 HTTP 会话

HTTP 连接可创建一个 HTTP 会话以维护 HTTP 请求之间的状态。

通过 **HTTP 会话**可以用最少的 SQL 应用程序代码来保持客户端（通常是 Web 浏览器）状态。会话上下文中的数据库连接在该会话的生存期期间会一直保持。每个用会话 ID 标记的新 HTTP 请求会进行序列化（排队），以便使用相同的数据库连接按顺序处理具有相同会话 ID 的每个请求。重复使用数据库连接为维护 HTTP 请求之间的状态信息提供了途径。相反，无会话的 HTTP 请求会为每个请求都新建一个数据库连接，而临时表和连接变量中的数据不能在各请求之间共享。

HTTP 会话管理对 URL 和 cookie 这两种状态管理技术都提供了支持。

在 *samples-dir\SQLAnywhere\HTTP\session.sql* 中提供了 HTTP 会话功能的工作示例。

创建 HTTP 会话

会话是通过调用 sa_set_http_option 系统过程利用 HTTP 选项 SessionID 在 Web 应用程序内创建而成。会话 ID 可以是任何非空字符串。在内部将生成一个由会话 ID 和数据库名组成的会话密钥，这样在装载多个数据库时，该会话密钥在各数据库中都保持唯一性。整个会话密钥的长度不得超过 128 个字符。在此示例中，将生成一个唯一的会话 ID，并将其传递到 sa_set_http_option。

```
DECLARE session_id VARCHAR(64);
DECLARE tm TIMESTAMP;
SET tm=now(*);
SET session_id = 'session_' ||
    CONVERT( VARCHAR, SECONDS(tm)*1000+DATEPART(millisecond,tm));
CALL sa_set_http_option('SessionID', session_id);
```

Web 应用程序可通过 SessionID 连接属性获得会话 ID。如果没有为连接定义会话 ID（即，连接是无会话连接），则此属性为空字符串。

```
DECLARE session_id VARCHAR(64);
SELECT CONNECTION_PROPERTY( 'SessionID' ) INTO session_id;
```

使用 sa_set_http_option 过程创建会话后，本地主机客户端便可访问具有指定会话 ID（例如，session_63315422814117），并在运行具有以下 URL 的服务 session_service 的数据库 *dbname* 内运行的会话。

```
http://localhost/dbname/session_service?sessionid=session_63315422814117
```

如果仅连接了一个数据库，则可省略数据库名。

```
http://localhost/session_service?sessionid=session_63315422814117
```

使用 cookie 的会话管理

使用字段名为 'Set-Cookie' 的 sa_set_http_header 系统过程支持 cookie 状态管理。在利用 cookie 进行状态管理时，不需要将会话 ID 包含在 URL 中。而是由客户端在其 HTTP cookie 标头中提供会话 ID。使用 cookie 进行状态管理的缺点是，在客户端可能已禁用 cookie 的非管制环境中，就不能依赖 cookie 支持。因此，Web 应用程序应同时支持 URL 和 cookie 会话状态管理。正如上一节中所

述，在客户端同时提供 URL 和 cookie 会话 ID 时，URL 会话 ID 优先。在会话到期或被显式删除时，则由 Web 应用程序负责删除 SessionID cookie（例如，sa_set_http_option('SessionID', NULL)）。

```
DECLARE session_id VARCHAR(64);
DECLARE tm TIMESTAMP;
SET tm=now(*);
SET session_id = 'session_' ||
    CONVERT( VARCHAR, SECONDS(tm)*1000+DATEPART(millisecond,tm));
CALL sa_set_http_option('SessionID', session_id);
CALL sa_set_http_header( 'Set-Cookie',
        'sessionid=' || session_id || ';' ||
        'max-age=60;' ||
        'path=/session;' );
```

失效会话的检测

SessionID 和 SessionCreateTime 连接属性对于确定当前连接是否在会话上下文中十分有用。如果其中任一连接属性查询返回空字符串，则该会话不存在。SessionCreateTime 属性提供了一个用于确定会话创建时间的度量。在执行 sa_set_http_option 调用后将立即定义该属性。SessionLastTime 属性用于提供会话的上次使用时间。更具体一点说，它是上一次处理的会话请求在终止时释放数据库连接的时间。从第一次创建会话开始，到创建会话的请求释放连接时为止，SessionLastTime 连接属性会作为空字符串返回。

删除或更改会话 ID

可通过使用新 SessionID 值调用 sa_set_http_option 系统过程将会话 ID 重设为另一个值。更改会话 ID 后将会删除旧会话并创建一个新会话，但它会重新使用当前的数据库连接，以便状态信息不会丢失。SessionID 可被设置为 NULL（或者空字符串），这样将会删除会话。不可将 SessionID 设置为现有会话的 ID（除了它自己的会话 ID，这样不会发生任何问题）。尝试将 SessionID 设置为现有会话的会话 ID 将会导致 "HTTP 选项 'SessionID' 的设置无效 SQLCODE=-939" 错误。

如果服务器接收到一连串指定同一个会话上下文的多个 HTTP 请求，则它会在其会话队列上将这些请求排队（序列化）。如果 SessionID 被其中一个（或多个）请求更改或删除，则会话队列中的任何待执行请求都会作为单个 HTTP 请求进行重新排队。每个 HTTP 请求都将无法获得该会话，因为它已不再存在。无法获得会话的 HTTP 请求在缺省情况下被视为无会话操作，并且会新建一个数据库连接。Web 应用程序可通过检查 SessionID 或 SessionCreateTime 连接属性的非空字符串值来验证请求是否在会话上下文中进行操作。当然，Web 应用程序可检查它使用的任何应用程序特定变量或临时表的状态。下面是一个示例：

```
IF VAREXISTS( 'state' ) = 0 THEN
    // first invocation by this connection
    CREATE VARIABLE state LONG VARCHAR;
END IF;
```

会话语义

HTTP 请求可创建一个 HTTP 会话上下文。此请求创建的会话总是会立即实例化，以便该会话将需要该会话上下文的任何后续 HTTP 请求进行排队。

不以会话开头但已创建其会话上下文的 HTTP 请求是其会话的创建者。在其会话上下文中发生任何更改或删除时，创建者请求可立即更改或删除该会话上下文。在会话上下文中开始的 HTTP 请求还可更改或删除其会话。对其会话的更改会立即创建一个完全可操作的待执行会话，只不过另一个 HTTP 请求无法取得所有权(而某个需要待执行会话的进来请求将代为在该会话上排队）。总而言之，更改或删除创建者请求的会话会立即修改当前会话上下文，而仅更改其会话的请求会修改它的待执行会话。HTTP 请求完成时，会检查自己是否具有待执行会话。如果存在待执行会话，则它将删除其当前会话并替换为待执行会话。被会话高速缓存的数据库连接会高效移动到新的会话上下文，且会保留所有状态数据（如临时表和创建的变量）。

无论在什么情况下，只要删除了 HTTP 会话，就会释放其队列中的任何请求，并允许这些请求在没有会话上下文的情况下执行。要求请求在会话上下文中运行的应用程序代码必须尝试通过调用 CONNECTION_PROPERTY('SessionID') 来获得一个有效的会话上下文。

```
DECLARE ses_id LONG VARCHAR;
SELECT CONNECTION_PROPERTY( 'SessionID' ) INTO ses_id;
```

如果某 HTTP 请求被故意取消或由于网络故障而取消，则会删除现有的待执行会话并保留原始会话上下文。创建者 HTTP 请求（无论是被取消还是正常终止）会立即更改会话状态。

删除连接和服务器关闭

显式删除在会话上下文中进行高速缓存处理的数据库连接会导致会话被删除。以这种方式删除会话会被视为是取消操作，并且从会话队列释放的任何 HTTP 请求都会处于已取消状态。这就确保了快速终止 HTTP 请求且适时地通知用户。

同样，服务器或数据库关闭会取消其相应的数据库连接，这可能会导致取消 HTTP 请求。

会话超时

http_session_timeout 公共数据库选项提供了可变的会话超时控制。该选项设置以分钟为单位。缺省情况下，公共设置为 30 分钟。最小值为 1 分钟，最大值为 525600 分钟（365 天）。Web 应用程序可从拥有会话的任何请求内更改超时标准。如果会话超时，则新的超时标准可能会影响后面排队的请求。由 Web 应用程序负责提供用于检测客户端是否在尝试访问不存在的会话的逻辑。它执行此任务的方式是，通过检查 SessionCreateTime 连接属性来确定它是否为有效的时间戳。如果 HTTP 请求与现有会话没有关联，则 SessionCreateTime 值将为空字符串。

会话范围

服务器重新启动期间会话不会持续。

授权

由于与会话关联的连接在连接生命期内都会保持它对数据库连接的持有状态，因此它同样也持有一个许可座席。Web 应用程序会将这一点考虑在内，从而确保适当删除失效会话，或设置相应的超时值。

有关 SQL Anywhere 授权的详细信息，请访问 http://www.sybase.com/detail?id=1056242。

会话错误

如果新请求尝试访问某会话而该会话上的待执行请求超过 16 个，或会话排队时出错，则会出现错误 [503 Service Unavailable]。

如果客户端 IP 地址或主机名与会话创建者的 IP 地址或主机名不匹配，则会发生错误 [403 Forbidden]。

如果请求规定的会话不存在，则该请求不会隐式生成错误。由 Web 应用程序负责检测此条件（通过检查 SessionID、SessionCreateTime 或 SessionLastTime 连接属性）并执行相应操作。

会话连接属性和选项汇总

连接属性

- **SessionID**

  ```
  SELECT CONNECTION_PROPERTY('SessionID') INTO ses_id;
  ```

  提供当前数据库上下文中的当前会话 ID。

- **SessionCreateTime**

  ```
  SELECT CONNECTION_PROPERTY('SessionCreateTime') INTO ses_create;
  ```

  提供会话创建时的时间戳。

- **SessionLastTime**

  ```
  SELECT CONNECTION_PROPERTY('SessionLastTime') INTO ses_last;
  ```

  提供上一个请求释放会话时的时间戳。

- **http_session_timeout**

  ```
  SELECT CONNECTION_PROPERTY('http_session_timeout') INTO ses_timeout;
  ```

  读取以分钟为单位的当前会话超时值。

HTTP 选项

- **'SessionID','value'**

  ```
  CALL sa_set_http_option( 'SessionID', 'my_app_session_1' );
  ```

  创建或更改当前 HTTP 请求的会话上下文。如果 my_app_session_1 归另一个 HTTP 请求所有，则返回一个错误。

- **'SessionID', NULL**

  ```
  CALL sa_set_http_option('SessionID', NULL );
  ```

如果请求来自会话创建者，则立即删除当前会话；否则，为会话设置删除标记。在请求没有会话时删除会话不会出错，也不会产生任何影响。

将会话更改为当前会话的 SessionID（没有待执行会话）不会出错，并且没有明显的影响。

将会话更改为另一个 HTTP 请求正在使用的 SessionID 会出错。

在某更改已为待执行状态时更改会话将导致待执行会话被删除，并创建新的待执行会话。

将带有待执行会话的会话更改回其原始 SessionID 将导致待执行会话被删除。

HTTP 会话超时

- **http_session_timeout**

```
SET TEMPORARY OPTION PUBLIC.http_session_timeout=100;
```

设置当前 HTTPS 会话超时值（以分钟为单位）。缺省值为 30 分钟，范围是 1 到 525600 分钟（365 天）。请参见“http_session_timeout 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

管理

Web 应用程序可能需要一种用于跟踪数据库服务器内活动会话使用情况的方法。使用 NEXT_CONNECTION 函数调用遍历活动数据库连接，并检查与会话相关的属性（如 SessionID）即可找到会话数据。以下 SQL 代码演示了此方法：

```
CREATE VARIABLE conn_id LONG VARCHAR;
CREATE VARIABLE the_sessionID LONG VARCHAR;
SELECT NEXT_CONNECTION( NULL, NULL ) INTO conn_id;
conn_loop:
LOOP
    IF conn_id IS NULL THEN
        LEAVE conn_loop;
    END IF;
    SELECT CONNECTION_PROPERTY( 'SessionID', conn_id )
        INTO the_sessionID;
    IF the_sessionID != '' THEN
        PRINT 'conn_id = %1!, SessionID = %2!', conn_id, the_sessionID;
    ELSE
        PRINT 'conn_id = %1!', conn_id;
    END IF;
    SELECT NEXT_CONNECTION( conn_id, NULL ) INTO conn_id;
END LOOP conn_loop;
PRINT '\n';
```

如果查看数据库服务器消息窗口，可能会看到类似以下的输出。

```
conn_id = 30
conn_id = 29, SessionID = session_63315442223323
conn_id = 28, SessionID = session_63315442220088
conn_id = 25, SessionID = session_63315441867629
```

显式删除属于某会话的连接会导致连接关闭并删除该会话。如果要被删除的连接当前正用于为 HTTP 请求提供服务，则将对该请求设置删除标记，并向其发送一个取消信号以终止该请求。当请求终止时，会话即被删除，并且连接关闭。如“删除或更改会话 ID”一节第 897 页中所述，删除

该会话会导致该会话队列上的待执行请求进行重新排队。如果连接当前处于非活动状态，则对会话设置删除标记，并将其重新排队为会话超时队列的起点。在下一个超时周期中（通常在 5 秒内）将删除该会话和连接。新 HTTP 请求不可以使用具有删除标记的任何会话。

在无条件停止数据库运行时，会删除每个数据库连接，这将导致该数据库上下文中的所有会话都被删除。这种情况是肯定要发生的，因为对于一个会话上下文必须存在一个有效的数据库连接，并且一个数据库连接一次仅可与一个会话相关联。会话和数据库连接必须在同一个数据库上下文内。

有关数据库上下文中会话的详细信息，请参见“创建 HTTP 会话”一节第 896 页中对会话密钥的说明。

使用字符集自动转换

缺省情况下，将对文本类型的外发结果集自动执行字符集转换。其它类型的结果集（如二进制对象）不会进行转换。请求的字符集将转换为数据库字符集，然后结果集会按需要从数据库字符集转换为客户端字符集，但结果集中的二进制列除外。当请求列出它可以处理的多个字符集时，服务器将从列表中选择第一个适用的字符集。

可通过设置 HTTP 选项 CharsetConversion 来启用或禁用字符集转换。允许的值为 ON 和 OFF。缺省值为 ON。以下语句关闭了字符集自动转换：

```
CALL sa_set_http_option( 'CharsetConversion', 'OFF' );
```

有关内置存储过程的详细信息，请参见“按字母顺排序的系统过程列表”一节 《SQL Anywhere 服务器 - SQL 参考》。

处理错误

当 Web 服务请求失败时，数据库服务器会生成标准错误，并显示在浏览器中。这些错误被指派了符合协议标准的编号。

如果服务是 SOAP 服务，则这些失败消息以 SOAP 1.1 版标准中定义的 SOAP 失败消息的形式返回到客户端：

- 如果处理请求的应用程序中的错误生成 SQLCODE，则将返回 faultcode 为 Client（还可能带有子类别，如 Procedure）的 SOAP 失败消息。SOAP 错误消息中的 faultstring 元素将设置为详细的错误说明，并且 detail 元素中会包含 SQLCODE 数字值。
- 如果发生传输协议错误，faultcode 将设置为 Client 或 Server（具体取决于错误），faultstring 将设置为 HTTP 传输消息（如 [`404 Not Found`]），并且详细信息元素中包含数字型的 HTTP 错误值。
- 由于返回 SQLCODE 值的应用程序错误而生成的 SOAP 错误消息返回时带有 HTTP 状态 [`200 OK`]。

如果无法将客户端标识为 SOAP 客户端，则在生成的 HTML 文档中返回相应的 HTTP 错误。

下面列出了一些可能遇到的典型错误：

编号	名称	SOAP 失败	说明
301	永久移走	Server	请求的页面已永久移走。服务器会自动将请求重定向到新的位置。
304	未修改	Server	根据请求中的信息，服务器已确定自上次请求后没有修改过请求的数据，因此无需将其再次发送。
307	临时重定向	Server	请求的页面已移动，但此更改不一定是永久性的。服务器会自动将请求重定向到新的位置。
400	错误请求	Client.BadRequest	HTTP 请求不完整或者格式有误。
401	需要授权	Client.Authorization	使用该服务需要授权，但未提供有效的用户名和口令。
403	禁止	Client.Forbidden	您无权访问该数据库。
404	未找到	Client.NotFound	服务器上没有运行指定的数据库，或者指定的 Web 服务不存在。
408	请求超时	Server.RequestTimeout	接收请求时超出了连接空闲时间上限。

编号	名称	SOAP 失败	说明
411	必需的 HTTP 长度	Client.LengthRequired	服务器要求客户端在请求中指定 Content-Length。通常在向服务器上载数据时发生这种情况。
413	实体太大	Server	请求超出允许的大小上限。
414	URI 太大	Server	URI 的长度超出了允许的长度上限。
500	内部服务器错误	Server	发生内部错误。无法处理该请求。
501	未实现	Server	HTTP 请求方法不是 GET、HEAD 或 POST。
502	错误的网关	Server	所请求的文档位于第三方服务器上，并且服务器收到来自第三方服务器的错误。
503	服务不可用	Server	连接数超出允许上限。

SQL Anywhere 数据库工具接口

本节介绍 SQL Anywhere 的数据库工具编程接口。

数据库工具接口

目录

数据库工具接口简介

SQL Anywhere 包括用于管理数据库的 Sybase Central 和一组实用程序。这些数据库管理实用程序执行各种任务，如备份数据库、创建数据库、将事务日志转换为 SQL 等。

支持的平台

所有数据库管理实用程序都使用一个名为**数据库工具库**的共享库。它是为 Windows 操作系统、Linux、Unix 和 Mac OS X 提供的。对于 Windows，此库的名称为 *dbtool11.dll*。对于 Linux 和 Unix，此库的名称为 *libdbtool11_r.so*。对于 Mac OS X，此库的名称为 *libdbtool11_r.dylib*。

您可以开发自己的数据库管理实用程序，也可以通过调用数据库工具库将数据库管理功能合并到您的应用程序中。本章介绍数据库工具库的接口。本章将假定您已熟悉如何从所使用的开发环境中调用库例程。

数据库工具库为每个数据库管理实用程序都提供了函数或入口点。此外，函数的调用必须是在使用其它数据库工具函数之前和完成其它数据库工具函数的使用之后。

Windows Mobile

为 Windows Mobile 提供了 *dbtool11.dll* 库，但是其中只包括 DBToolsInit、DBToolsFini、DBRemoteSQL 和 DBSynchronizeLog 的入口点。没有为 Windows Mobile 提供其它入口点。

dbtools.h 头文件

SQL Anywhere 提供的 dbtools 头文件列出了 DBTools 库的入口点以及用于将信息传入和传出该库的结构。*dbtools.h* 文件安装在 SQL Anywhere 安装目录下的 *SDK\Include* 子目录中。您应当查阅 *dbtools.h* 文件以获得有关入口点和结构成员的最新信息。

dbtools.h 头文件包含其它文件，如：

- **sqlca.h** 这是为解析各种宏（而非 SQLCA 本身）而提供的。
- **dllapi.h** 为与操作系统和语言相关的宏定义预处理器宏。
- **dbtlvers.h** 定义 DB_TOOLS_VERSION_NUMBER 预处理器宏和其它版本特定的宏。

sqldef.h 头文件

sqldef.h 头文件包括错误返回值。

dbrmt.h 头文件

随 SQL Anywhere 提供的 *dbrmt.h* 头文件介绍了 DBTools 库中的 DBRemoteSQL 入口点以及用于将信息传入和传出 DBRemoteSQL 入口点的结构。*dbrmt.h* 文件安装在 SQL Anywhere 安装目录下的 *SDK\Include* 子目录中。您应当查阅 *dbrmt.h* 文件以获得有关 DBRemoteSQL 入口点和结构成员的最新信息。

使用数据库工具接口

本节概述如何开发通过使用 DBTools 接口来管理数据库的应用程序。

使用导入库

要使用 DBTools 函数，您必须将应用程序链接到包含所需函数定义的 DBTools **导入库**。

对于 Unix 系统，不需要任何导入库。请直接链接到 *libdbtool11.so*（非线程）或 *libdbtool11_r.so*（线程）。

导入库

SQL Anywhere 为 Windows 和 Windows Mobile 提供了 DBTools 接口的导入库。对于 Windows 系统，可以在 SQL Anywhere 安装目录下的 *SDK\Lib\x86* 和 *SDK\Lib\x64* 子目录中找到导入库。对于 Windows Mobile 系统，可以在 SQL Anywhere 安装目录下的 *SDK\Lib\CE\Arm.50* 子目录中找到导入库。所提供的 DBTools 导入库如下所示：

编译器	库
Microsoft Windows	*dbtlstm.lib*
Microsoft Windows Mobile	*dbtool11.lib*

启动和完成 DBTools 库

在使用任何 DBTools 函数之前，您必须调用 DBToolsInit。使用完 DBTools 库后，您必须调用 DBToolsFini。

DBToolsInit 和 DBToolsFini 函数的主要用途是允许 DBTools 库装载 SQL Anywhere 消息库。消息库包含 DBTools 内部使用的所有本地化版本的错误消息和提示。如果未调用 DBToolsFini，消息库的引用计数不会减少，而且不会被卸载，因此请注意：一定要有一对匹配的 DBToolsInit/DBToolsFini 调用。

下列代码段阐释了如何初始化和清除 DBTools：

```
// Declarations
a_dbtools_info  info;
short           ret;

//Initialize the a_dbtools_info structure
memset( &info, 0, sizeof( a_dbtools_info) );
info.errorrtn = (MSG_CALLBACK)MyErrorRtn;

// initialize the DBTools library
ret = DBToolsInit( &info );
if( ret != EXIT_OKAY ) {
    // library initialization failed
    ...
}
```

```
// call some DBTools routines ...
...
// finalize the DBTools library
DBToolsFini( &info );
```

调用 DBTools 函数

所有工具都是通过先填写结构然后调用 DBTools 库中的函数（或**入口点**）来运行的。每个入口点都将指向单个结构的指针视为参数。

下面的示例展示了如何在 Windows 操作系统上使用 DBBackup 函数。

```
// Initialize the structure
a_backup_db backup_info;
memset( &backup_info, 0, sizeof( backup_info ) );

// Fill out the structure
backup_info.version = DB_TOOLS_VERSION_NUMBER;
backup_info.output_dir = "c:\\backup";
backup_info.connectparms ="UID=DBA;PWD=sql;DBF=demo.db";

backup_info.confirmrtn = (MSG_CALLBACK) ConfirmRtn ;
backup_info.errorrtn = (MSG_CALLBACK) ErrorRtn ;
backup_info.msgrtn = (MSG_CALLBACK) MessageRtn ;
backup_info.statusrtn = (MSG_CALLBACK) StatusRtn ;
backup_info.backup_database = TRUE;

// start the backup
DBBackup( &backup_info );
```

有关 DBTools 结构成员的信息，请参见“DBTools 结构”一节第 925 页。

使用回调函数

DBTools 结构中有几个类型为 MSG_CALLBACK 的元素。这些是指向回调函数的指针。

使用回调函数

回调函数允许 DBTools 函数将对操作的控制返回到用户的调用应用程序中。DBTools 库使用回调函数来处理因以下四种用途而由 DBTools 函数发送到用户的消息：

- **确认** 当需要由用户确认操作时调用。例如，如果备份目录不存在，则工具库会询问是否需要创建它。
- **错误消息** 在发生错误（如在执行某个操作时磁盘空间不足）时被调用以处理消息。
- **信息消息** 在工具要向用户显示某些消息（如正在卸载的当前表的名称）时调用。
- **状态信息** 针对工具调用以显示操作的状态（如卸载表时的完成百分比）。

将回调函数指派给结构

您可以将回调例程直接指派给结构。以下语句是使用备份结构的一个示例：

```
backup_info.errorrtn = (MSG_CALLBACK) MyFunction
```


MSG_CALLBACK 在随 SQL Anywhere 提供的 *dllapi.h* 头文件中定义。工具例程可以使用应当出现在相应用户界面中的消息回调调用应用程序，而无论该用户界面是窗口环境、基于字符的系统上的标准输出还是其它用户界面。

确认回调函数的示例

下面这个示例确认例程要求用户对提示回答 [是] 或 [否] 并返回用户的选择：

```
extern short _callback ConfirmRtn(
        char * question )
{
    int ret = IDNO;
    if( question != NULL ) {
        ret = MessageBox( HwndParent, question,
        "Confirm", MB_ICONEXCLAMATION|MB_YESNO );
    }
    return( ret == IDYES );
}
```

错误回调函数的示例

下面是一个错误消息处理例程的示例，它在一个窗口中显示错误消息。

```
extern short _callback ErrorRtn(
        char * errorstr )
{
    if( errorstr != NULL ) {
        MessageBox( HwndParent, errorstr, "Backup Error", MB_ICONSTOP|
MB_OK );
    }
    return( 0 );
}
```

消息回调函数的示例

消息回调函数的公用执行会将消息输出到屏幕上：

```
extern short _callback MessageRtn(
        char * messagestr )
{
    if( messagestr != NULL ) {
                    OutputMessageToWindow( messagestr );
    }
    return( 0 );
}
```

状态回调函数的示例

状态回调例程在工具需要显示操作的状态（如卸载表时的完成百分比）时被调用。通常实现只将消息输出到屏幕上：

```
extern short _callback StatusRtn(
        char * statusstr )
{
    if( statusstr != NULL ) {
        OutputMessageToWindow( statusstr );
    return( 0 );
}
```

版本号和兼容性

每个结构都有一个指示版本号的成员。您应当使用该版本成员来保存应用程序开发所针对的 DBTools 库的版本。当您将 *dbtools.h* 头文件包括进来时便定义了 DBTools 库的当前版本。

♦ 将当前版本号指派给结构

- 在调用 DBTools 函数之前，将版本常量指派给结构的版本成员。下面这行代码将当前版本指派给备份结构：

```
backup_info.version = DB_TOOLS_VERSION_NUMBER;
```

兼容性

版本号允许应用程序继续针对较新版本的 DBTools 库工作。即使新成员已添加到 DBTools 结构中，DBTools 函数也使用由应用程序提供的版本号来允许应用程序工作。

如果更新了任何 DBTools 结构或是发布了更新版本的软件，版本号就会增大。如果您使用了 DB_TOOLS_VERSION_NUMBER 并用新版本的 DBTools 头文件重建您的应用程序，则必须部署新版本的 DBTools 库。如果您应用程序的功能没有变化，则您可能想要使用 *dbtlsvers.h* 中定义的一个版本特定的宏，以便不会出现库版本不匹配的情况。

使用位字段

许多 DBTools 结构都使用位字段来以紧凑方式保存布尔信息。例如，备份结构包括以下位字段：

```
a_bit_field     backup_database : 1;
a_bit_field     backup_logfile  : 1;
a_bit_field     no_confirm      : 1;
a_bit_field     quiet           : 1;
a_bit_field     rename_log      : 1;
a_bit_field     truncate_log    : 1;
a_bit_field     rename_local_log: 1;
a_bit_field     server_backup   : 1;
```

每个位字段的长度都是一位，这由结构声明中冒号右侧的 1 来指示。所使用的特定数据类型取决于指派给 a_bit_field 的值，此参数在 *dbtools.h* 的顶部设置且与操作系统相关。

将值 0 或 1 指派给位字段，以传递此结构中的布尔信息。

DBTools 的示例

可以在 *samples-dir\SQLAnywhere\DBTools* 目录中找到本示例以及对其进行编译的说明。示例程序本身在 *main.cpp* 中。该示例说明了如何使用 DBTools 库执行数据库备份。

```
#define WIN32

#include <stdio.h>
#include <string.h>
#include "windows.h"
#include "sqldef.h"
#include "dbtools.h"
```



```
extern short _callback ConfirmCallBack( char * str )
{
    if( MessageBox( NULL, str, "Backup",
         MB_YESNO|MB_ICONQUESTION ) == IDYES )
    {
        return 1;
    }
    return 0;
}
extern short _callback MessageCallBack( char * str )
{
    if( str != NULL )
    {
        fprintf( stdout, "%s\n", str );
    }
    return 0;
}
extern short _callback StatusCallBack( char * str )
{
    if( str != NULL )
    {
        fprintf( stdout, "%s\n", str );
    }
    return 0;
}
extern short _callback ErrorCallBack( char * str )
{
    if( str != NULL )
    {
        fprintf( stdout, "%s\n", str );
    }
    return 0;
}
typedef void (CALLBACK *DBTOOLSPROC)( void * );
typedef short (CALLBACK *DBTOOLSFUNC)( void * );

// Main entry point into the program.
int main( int argc, char * argv[] )
{
    a_dbtools_info  dbt_info;
    a_backup_db     backup_info;
    char            dir_name[ _MAX_PATH + 1];
    char            connect[ 256 ];
    HINSTANCE       hinst;
    DBTOOLSFUNC     dbbackup;
    DBTOOLSFUNC     dbtoolsinit;
    DBTOOLSPROC     dbtoolsfini;
    short           ret_code;

    // Always initialize to 0 so new versions
    // of the structure will be compatible.
    memset( &dbt_info, 0, sizeof( a_dbtools_info ) );
    dbt_info.errorrtn = (MSG_CALLBACK)MessageCallBack;;

    memset( &backup_info, 0, sizeof( a_backup_db ) );
    backup_info.version = DB_TOOLS_VERSION_NUMBER;
    backup_info.quiet = 0;
    backup_info.no_confirm = 0;
    backup_info.confirmrtn = (MSG_CALLBACK)ConfirmCallBack;
    backup_info.errorrtn = (MSG_CALLBACK)ErrorCallBack;
    backup_info.msgrtn = (MSG_CALLBACK)MessageCallBack;
    backup_info.statusrtn = (MSG_CALLBACK)StatusCallBack;
    if( argc > 1 )
    {
```

```
            strncpy( dir_name, argv[1], _MAX_PATH );
        }
        else
        {
            // DBTools does not expect (or like) a trailing slash
            strcpy( dir_name, "c:\\temp" );
        }
        backup_info.output_dir = dir_name;
        if( argc > 2 )
        {
            strncpy( connect, argv[2], 255 );
        }
        else
        {
            strcpy( connect, "DSN=SQL Anywhere 11 Demo" );
        }
        backup_info.connectparms = connect;
        backup_info.quiet = 0;
        backup_info.no_confirm = 0;
        backup_info.backup_database = 1;
        backup_info.backup_logfile = 1;
        backup_info.rename_log = 0;
        backup_info.truncate_log = 0;
        hinst = LoadLibrary( "dbtool11.dll" );
        if( hinst == NULL )
        {
            // Failed
            return EXIT_FAIL;
        }
        dbbackup = (DBTOOLSFUNC) GetProcAddress( (HMODULE)hinst,
            "_DBBackup@4" );
        dbtoolsinit = (DBTOOLSFUNC) GetProcAddress( (HMODULE)hinst,
            "_DBToolsInit@4" );
        dbtoolsfini = (DBTOOLSPROC) GetProcAddress( (HMODULE)hinst,
            "_DBToolsFini@4" );
        ret_code = (*dbtoolsinit)( &dbt_info );
        if( ret_code != EXIT_OKAY ) {
            return ret_code;
        }
        ret_code = (*dbbackup)( &backup_info );
        (*dbtoolsfini)( &dbt_info );
        FreeLibrary( hinst );
        return ret_code;
    }
```


DBTools 函数

DBBackup 函数

备份数据库。该函数由 dbbackup 实用程序使用。

原型

short **DBBackup (** const a_backup_db * **);**

参数

指向结构的指针。请参见“a_backup_db 结构”一节第 925 页。

返回值

一个返回代码，如“软件组件的退出代码”一节第 972 页中所列。

注释

DBBackup 函数管理所有的客户端数据库备份任务。

有关这些任务的说明，请参见“备份实用程序 (dbbackup)”一节 《SQL Anywhere 服务器 - 数据库管理》。

要执行服务器端备份，请使用 BACKUP DATABASE 语句。请参见“BACKUP 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

DBChangeLogName 函数

更改事务日志文件的名称。该函数由 dblog 实用程序使用。

原型

short **DBChangeLogName (** const a_change_log * **);**

参数

指向结构的指针。请参见“a_change_log 结构”一节第 927 页。

返回值

一个返回代码，如“软件组件的退出代码”一节第 972 页中所列。

注释

事务日志实用程序 (dblog) 的 -t 选项更改事务日志的名称。DBChangeLogName 向该函数提供编程接口。

有关 dblog 实用程序的说明，请参见“事务日志实用程序 (dblog)”一节 《SQL Anywhere 服务器 - 数据库管理》。

另请参见

- “ALTER DATABASE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

DBCreate 函数

创建数据库。该函数由 dbinit 实用程序使用。

原型

short **DBCreate (** const a_create_db * **);**

参数

指向结构的指针。请参见“a_create_db 结构”一节第 929 页。

返回值

一个返回代码，如“软件组件的退出代码”一节第 972 页中所列。

注释

有关 dbinit 实用程序的信息，请参见“初始化实用程序 (dbinit)”一节 《SQL Anywhere 服务器 - 数据库管理》。

另请参见

- “CREATE DATABASE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

DBCreatedVersion 函数

确定用于创建数据库文件的 SQL Anywhere 的版本（不尝试启动数据库）。目前，此函数仅区分版本 10 或 11 与版本 10 之前的数据库。

原型

short **DBCreatedVersion (** a_db_version_info * **);**

参数

指向结构的指针。请参见“a_db_version_info 结构”一节第 933 页。

返回值

一个返回代码，如“软件组件的退出代码”一节第 972 页中所列。

注释

如果返回代码指示成功，则 a_db_version_info 结构的 created_version 字段会包含一个 a_db_version 类型的值，该值指示创建数据库的 SQL Anywhere 的版本。有关可能的值的定义，请参见“a_db_version 枚举”一节第 965 页。

如果返回失败的代码，则不设置版本信息。

另请参见

- “CREATE DATABASE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “a_db_version_info 结构”一节第 933 页
- “a_db_version 枚举”一节第 965 页

DBErase 函数

消除数据库文件和/或事务日志文件。该函数由 dberase 实用程序使用。

原型

short **DBErase (** const an_erase_db * **);**

参数

指向结构的指针。请参见“an_erase_db 结构”一节第 935 页。

返回值

一个返回代码，如“软件组件的退出代码”一节第 972 页中所列。

注释

有关消除实用程序及其功能的信息，请参见“消除实用程序 (dberase)”一节 《SQL Anywhere 服务器 - 数据库管理》。

DBInfo 函数

返回有关数据库文件的信息。该函数由 dbinfo 实用程序使用。

原型

short **DBInfo (** const a_db_info * **);**

参数

指向结构的指针。请参见“a_db_info 结构”一节第 931 页。

返回值

一个返回代码，如“软件组件的退出代码”一节第 972 页中所列。

注释

有关信息实用程序及其功能的信息，请参见“信息实用程序 (dbinfo)”一节 《SQL Anywhere 服务器 - 数据库管理》。

另请参见

- “DBInfoDump 函数”一节第 918 页
- “DBInfoFree 函数”一节第 918 页
- “DB_PROPERTY 函数 [System]”一节 《SQL Anywhere 服务器 - SQL 参考》

DBInfoDump 函数

返回有关数据库文件的信息。该函数由应用了 -u 选项的 dbinfo 实用程序使用。

原型

short **DBInfoDump (** const a_db_info * **);**

参数

指向结构的指针。请参见“a_db_info 结构”一节第 931 页。

返回值

一个返回代码，如“软件组件的退出代码”一节第 972 页中所列。

注释

有关信息实用程序及其功能的信息，请参见“信息实用程序 (dbinfo)”一节 《SQL Anywhere 服务器 - 数据库管理》。

另请参见

- “DBInfo 函数”一节第 917 页
- “DBInfoFree 函数”一节第 918 页
- “sa_table_page_usage 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》

DBInfoFree 函数

调用 DBInfoDump 函数后释放资源。

原型

short **DBInfoFree (** const a_db_info * **);**

参数

指向结构的指针。请参见“a_db_info 结构”一节第 931 页。

返回值

一个返回代码，如“软件组件的退出代码”一节第 972 页中所列。

注释

有关信息实用程序及其功能的信息，请参见“信息实用程序 (dbinfo)”一节 《SQL Anywhere 服务器 - 数据库管理》。

另请参见

- “DBInfo 函数”一节第 917 页
- “DBInfoDump 函数”一节第 918 页

DBLicense 函数

修改或报告数据库服务器的授权信息。

原型

short **DBLicense (** const a_db_lic_info * **);**

参数

指向结构的指针。请参见“a_dblic_info 结构”一节第 934 页。

返回值

一个返回代码，如“软件组件的退出代码”一节第 972 页中所列。

注释

有关服务器授权实用程序及其功能的信息，请参见“服务器授权实用程序 (dblic)”一节 《SQL Anywhere 服务器 - 数据库管理》。

DBRemoteSQL 函数

访问 SQL Remote 消息代理。

原型

short **DBRemoteSQL(** const a _remote_sql * **);**

参数

指向结构的指针。请参见“a_remote_sql 结构”一节第 937 页。

返回值

一个返回代码，如“软件组件的退出代码”一节第 972 页中所列。

注释

有关您可以访问的功能的信息，请参见“消息代理 (dbremote)”一节 《SQL Remote》。

另请参见

- “SQL Remote 简介” 《SQL Remote》

DBSynchronizeLog 函数

将数据库与 MobiLink 服务器同步。

原型

short **DBSynchronizeLog(** const a _sync_db * **);**

参数

指向结构的指针。请参见“a_sync_db 结构”一节第 942 页。

返回值

一个返回代码，如“软件组件的退出代码”一节第 972 页中所列。

注释

有关您可以访问的功能的信息，请参见“启动同步”一节 《MobiLink - 客户端管理》 。

另请参见

- “dbmlsync 的 DBTools 接口” 《MobiLink - 客户端管理》

DBToolsFini 函数

当应用程序用完 DBTools 库后，计数器值减一并释放资源。

原型

short **DBToolsFini (** const a_dbtools_info * **);**

参数

指向结构的指针。请参见“a_dbtools_info 结构”一节第 935 页。

返回值

一个返回代码，如“软件组件的退出代码”一节第 972 页中所列。

注释

必须在任何使用 DBTools 接口的应用程序末尾调用 DBToolsFini 函数。否则会导致丢失内存资源。

另请参见

- “DBToolsInit 函数”一节第 921 页

DBToolsInit 函数

准备要使用的 DBTools 库。

原型

short **DBToolsInit(** const a_dbtools_info * **);**

参数

指向结构的指针。请参见“a_dbtools_info 结构”一节第 935 页。

返回值

一个返回代码，如“软件组件的退出代码”一节第 972 页中所列。

注释

DBToolsInit 函数的主要用途是装载 SQL Anywhere 消息库。消息库包含 DBTools 内部使用的本地化版本的错误消息和提示。

必须在使用 DBTools 接口的任何应用程序的开头以及在任何其它 DBTools 函数之前调用 DBToolsInit 函数。有关示例内容，请参见“DBTools 的示例”一节第 912 页。

另请参见

- “DBToolsFini 函数”一节第 920 页

DBToolsVersion 函数

返回 DBTools 库的版本号。

原型

short **DBToolsVersion (** void **);**

返回值

一个短整数，指示 DBTools 库的版本号。

注释

使用 DBToolsVersion 函数来检查 DBTools 库不比应用程序开发所针对的库旧。尽管应用程序可针对较新版本的 DBTools 运行，但是它们不能针对较旧版本的 DBTools 运行。

另请参见

- “版本号和兼容性”一节第 912 页

DBTranslateLog 函数

将事务日志文件翻译为 SQL。该函数由 dbtran 实用程序使用。

原型

short **DBTranslateLog (** const a_translate_log * **);**

参数

指向结构的指针。请参见“a_translate_log 结构”一节第 952 页。

返回值

一个返回代码，如“软件组件的退出代码”一节第 972 页中所列。

注释

有关翻译日志文件实用程序的信息，请参见“翻译日志文件实用程序 (dbtran)”一节 《SQL Anywhere 服务器 - 数据库管理》。

DBTruncateLog 函数

截断事务日志文件。该函数由 dbbackup 实用程序使用。

原型

short **DBTruncateLog (** const a_truncate_log * **);**

参数

指向结构的指针。请参见“a_truncate_log 结构”一节第 956 页。

返回值

一个返回代码，如“软件组件的退出代码”一节第 972 页中所列。

注释

有关备份实用程序的信息，请参见“备份实用程序 (dbbackup)”一节 《SQL Anywhere 服务器 - 数据库管理》。

另请参见

- “BACKUP 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

DBUnload 函数

卸载数据库。此函数由 dbunload 和 dbxtract 实用程序使用。

原型

short **DBUnload (** const an_unload_db * **);**

参数

指向结构的指针。请参见“an_unload_db 结构”一节第 957 页。

返回值

一个返回代码，如“软件组件的退出代码”一节第 972 页中所列。

注释

有关卸载实用程序的信息，请参见“卸载实用程序 (dbunload)”一节 《SQL Anywhere 服务器 - 数据库管理》。

有关抽取实用程序的信息，请参见“抽取实用程序 (dbxtract)”一节 《SQL Remote》。

DBUpgrade 函数

升级数据库文件。该函数由 dbupgrad 实用程序使用。

原型

short **DBUpgrade (** const an_upgrade_db * **);**

参数

指向结构的指针。请参见“an_upgrade_db 结构”一节第 961 页。

返回值

一个返回代码，如“软件组件的退出代码”一节第 972 页中所列。

注释

有关升级实用程序的信息，请参见“升级实用程序 (dbupgrad)”一节 《SQL Anywhere 服务器 - 数据库管理》。

另请参见

- “ALTER DATABASE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

DBValidate 函数

校验所有或部分数据库。该函数由 dbvalid 实用程序使用。

原型

short **DBValidate (** const a_validate_db * **);**

参数

指向结构的指针。请参见“a_validate_db 结构”一节第 962 页。

返回值

一个返回代码，如“软件组件的退出代码”一节第 972 页中所列。

注释

有关校验实用程序的信息，请参见“校验实用程序 (dbvalid)”一节 《SQL Anywhere 服务器 - 数据库管理》。

> **小心**
> 应在没有任何连接对数据库进行更改时对表或整个数据库进行校验；否则，可能会报告虚假错误，指出某种形式的数据库损坏，而实际上并没有任何损坏。

另请参见

- “VALIDATE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “sa_validate 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》

DBTools 结构

本节列出了用于与 DBTools 库交换信息的结构。这些结构按字母顺序列出。除 a_remote_sql 结构以外，所有这些结构都在 *dbtools.h* 中定义。a_remote_sql 结构在 *dbrmt.h* 中定义。

许多结构元素与相应实用程序上的命令行选项相对应。例如，有几个结构具有名为 quiet 的成员，它们可采用值 0 或 1。该成员与由许多实用程序使用的 quiet 操作 (-q) 选项相对应。

a_backup_db 结构

保存使用 DBTools 库执行备份任务所需的信息。

语法

```
typedef struct a_backup_db {
    unsigned short      version;
    const char *        output_dir;
    const char *        connectparms;
    MSG_CALLBACK        confirmrtn;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        statusrtn;
    a_bit_field         backup_database : 1;
    a_bit_field         backup_logfile  : 1;
    a_bit_field         no_confirm      : 1;
    a_bit_field         quiet           : 1;
    a_bit_field         rename_log      : 1;
    a_bit_field         truncate_log    : 1;
    a_bit_field         rename_local_log: 1;
    a_bit_field         server_backup   : 1;
    const char *        hotlog_filename;
    char                backup_interrupted;
    a_chkpt_log_type    chkpt_log_type;
    a_sql_uint32        page_blocksize;
} a_backup_db;
```

成员

成员	说明
Version	DBTools 版本号。
output_dir	输出目录的路径。例如： `"c:\backup"`

成员	说明

connectparms	连接到数据库所需的参数。它们采用如下所示的连接字符串形式：
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db"`
	数据库服务器将由连接字符串的 START 参数启动。例如：
	`"START=d:\sqlany11\bin32\dbeng11.exe"`
	一个包括 START 参数的完整连接字符串示例：
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db;START=d:\sqlany11\bin32\dbeng11.exe"`
	有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

confirmrtn	用于确认操作的回调例程。
errorrtn	用于处理错误消息的回调例程。
msgrtn	用于处理信息消息的回调例程。
statusrtn	用于处理状态消息的回调例程。
backup_database	备份数据库文件 (1) 或不备份 (0)。
backup_logfile	备份事务日志文件 (1) 或不备份 (0)。
no_confirm	在进行确认 (0) 或不进行确认 (1) 的情况下运行。
quiet	在不打印消息 (1) 或打印消息 (0) 的情况下运行。
rename_log	重命名事务日志。
truncate_log	删除事务日志。
rename_local_log	重命名事务日志的本地备份。
server_backup	设置为 1 时，指示使用 BACKUP DATABASE 在服务器上进行备份。相当于 dbbackup -s 选项。
hotlog_filename	活动备份文件的文件名。
backup_interrupted	指示该操作已被中断。

成员	说明
chkpt_log_type	控制检查点日志的复制。必须是 BACKUP_CHKPT_LOG_COPY、BACKUP_CHKPT_LOG_NOCOPY、BACKUP_CHKPT_LOG_RECOVER、BACKUP_CHKPT_LOG_AUTO 或 BACKUP_CHKPT_LOG_DEFAULT 其中之一。
page_blocksize	数据块中的页数。相当于 dbbackup -b 选项。如果设置为 0，则缺省值为 128。

另请参见

- “DBBackup 函数”一节第 915 页
- “a_db_version 枚举”一节第 965 页
- “使用回调函数”一节第 910 页

a_change_log 结构

保存使用 DBTools 库执行 dblog 任务所需的信息。

语法

```
typedef struct a_change_log {
    unsigned short      version;
    const char *        dbname;
    const char *        logname;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    a_bit_field         query_only              : 1;
    a_bit_field         quiet                   : 1;
    a_bit_field         change_mirrorname       : 1;
    a_bit_field         change_logname          : 1;
    a_bit_field         ignore_ltm_trunc        : 1;
    a_bit_field         ignore_remote_trunc     : 1;
    a_bit_field         set_generation_number   : 1;
    a_bit_field         ignore_dbsync_trunc     : 1;
    const char *        mirrorname;
    unsigned short      generation_number;
    char *              zap_current_offset;
    char *              zap_starting_offset;
    char *              encryption_key;
} a_change_log;
```

成员

成员	说明
version	DBTools 版本号。
dbname	数据库文件名。

成员	说明
logname	事务日志的名称。如果设置为 NULL，则没有日志。
errorrtn	用于处理错误消息的回调例程。
msgrtn	用于处理信息消息的回调例程。
query_only	如果为 1，则只显示事务日志的名称。如果为 0，则允许更改日志名。
quiet	在不打印消息 (1) 或打印消息 (0) 的情况下运行。
change_mirrorname	如果为 1，则允许更改日志镜像的名称。
change_logname	如果为 1，则允许更改事务日志的名称。
ignore_ltm_trunc	当使用日志传送管理器时，执行与 dbcc settrunc('ltm', 'gen_id', n) 复制服务器功能相同的功能。 有关 dbcc 的信息，请参见复制服务器的文档。
ignore_remote_trunc	用于 SQL Remote。重置为 delete_old_logs 选项保留的偏移，允许事务日志在不再需要时被删除。
set_generation_number	当使用日志传送管理器时，在备份被还原以便设置世代号之后使用。
ignore_dbsync_trunc	当使用 dbmlsync 时，重新设置用于 delete_old_logs 选项的偏移量，允许事务日志在不再需要时被删除。
mirrorname	事务日志镜像文件的新名称。
generation_number	新的世代号。与 set_generation_number 一起使用。
zap_current_offset	将当前的偏移量更改为指定的值。这只用于在卸载并重装之后重置事务日志以便与 dbremote 或 dbmlsync 设置相匹配。
zap_starting_offset	将起始偏移量更改为指定的值。这只用于在卸载并重装之后重置事务日志以便与 dbremote 或 dbmlsync 设置相匹配。
encryption_key	数据库文件的加密密钥。

另请参见

- “DBChangeLogName 函数”一节第 915 页
- “使用回调函数”一节第 910 页

a_create_db 结构

保存使用 DBTools 库创建数据库所需的信息。

语法

```
typedef struct a_create_db {
    unsigned short      version;
    const char          *dbname;
    const char          *logname;
    const char          *startline;
    unsigned short      page_size;
    const char          *default_collation;
    const char          *nchar_collation;
    const char          *encoding;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;

    a_bit_field         blank_pad               : 2;
    a_bit_field         respect_case            : 1;
    a_bit_field         encrypt                 : 1;
    a_bit_field         avoid_view_collisions   : 1;
    a_bit_field         jconnect                : 1;
    a_bit_field         checksum                : 1;
    a_bit_field         encrypted_tables        : 1;
    a_bit_field         case_sensitivity_use_default : 1;
    char                verbose;
    char                accent_sensitivity;
    const char          *mirrorname;
    const char          *data_store_type;
    const char          *encryption_key;
    const char          *encryption_algorithm;
    char                *dba_uid;
    char                *dba_pwd;
    unsigned int        db_size;
    int                 db_size_unit;

} a_create_db;
```

成员

成员	说明
version	DBTools 版本号。
dbname	数据库文件名。
logname	新事务日志的名称。

startline	用于启动数据库服务器的命令行。例如：
	`"d:\sqlany11\bin32\dbeng11.exe"`
	如果此成员为 NULL，则使用缺省启动行
	以下是缺省 START 参数：
	`"dbeng11 -gp page_size -c 10M"`

成员	说明
page_size	数据库的页面大小。
default_collation	数据库的归类。
nchar_collation	如果不为 NULL，则用于生成带有指定字符串的 NCHAR COLLATION 子句。
errorrtn	用于处理错误消息的回调例程。
msgrtn	用于处理信息消息的回调例程。
blank_pad	必须是 NO_BLANK_PADDING 或 BLANK_PADDING 之一。比较字符串时，将空白视为有效，并保存索引信息以反映这种情况。请参见“空白填充枚举”一节第 964 页。
respect_case	使字符串比较区分大小写，并保存索引信息以反映这种情况。
encrypt	设置后，将生成 ENCRYPTED ON，如果还设置了 encrypted_tables，将生成 ENCRYPTED TABLES ON 子句。
avoid_view_collisions	忽略 Watcom SQL 兼容视图 SYS.SYSCOLUMNS 和 SYS.SYSINDEXES 的生成。
jconnect	包括 jConnect 所需的系统过程。
checksum	设置为 1 表示 ON，设置为 0 表示 OFF。生成 CHECKSUM ON 或 CHECKSUM OFF 子句之一。
encrypted_tables	设置为 1 支持加密表。与 encrypt 一起使用，生成 ENCRYPTED TABLE ON 子句而不是 ENCRYPTED ON 子句。
case_sensitivity_use_default	如果设置该成员，则地区将使用缺省的区分大小写设置。这只会影响 UCA。设置该成员时，请不要向 CREATE DATABASE 语句中添加 CASE RESPECT 子句。
verbose	请参见“详细枚举”一节第 968 页。
accent_sensitivity	y、n 或 f（是、否、法语）之一。生成 ACCENT RESPECT、ACCENT IGNORE 或 ACCENT FRENCH 子句之一。
mirrorname	事务日志镜像的名称。
data_store_type	保留。使用 NULL。
encryption_key	数据库文件的加密密钥。与 encrypt 一起使用，生成 KEY 子句。

成员	说明
encryption_algorithm	加密算法（AES、AES256、AES_FIPS 或 AES256_FIPS）。与 encrypt 和 encryption_key 一起使用，生成 ALGORITHM 子句。
dba_uid	如果不为 NULL，则生成 DBA USER xxx 子句。
dba_pwd	如果不为 NULL，则生成 DBA PASSWORD xxx 子句。
db_size	如果不为 0，则生成 DATABASE SIZE 子句。
db_size_unit	与 db_size 一起使用，必须为 DBSP_UNIT_NONE、DBSP_UNIT_PAGES、DBSP_UNIT_BYTES、DBSP_UNIT_KILOBYTES、DBSP_UNIT_MEGABYTES、DBSP_UNIT_GIGABYTES、DBSP_UNIT_TERABYTES 其中之一。如果不为 DBSP_UNIT_NONE，它会生成相应的关键字（例如，当 db_size 为 10 且 db_size_unit 为 DBSP_UNIT_MEGABYTES 时，将生成 DATABASE SIZE 10 MB）。请参见“数据库大小单位枚举”一节第 965 页。

另请参见

- “DBCreate 函数”一节第 916 页
- “使用回调函数”一节第 910 页

a_db_info 结构

保存使用 DBTools 库返回 dbinfo 信息所需的信息。

语法

```
typedef struct a_db_info {
    unsigned short      version;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        statusrtn;
    unsigned short      dbbufsize;
    char *              dbnamebuffer;
    unsigned short      logbufsize;
    char *              lognamebuffer;
    unsigned short      mirrorbufsize;
    char *              mirrornamebuffer;
    unsigned short      collationnamebufsize;
    char *              collationnamebuffer;
    const char *        connectparms;
    a_bit_field         quiet    : 1;
    a_bit_field         page_usage : 1;
    a_sysinfo           sysinfo;
    a_table_info *      totals;
    a_sql_uint32        file_size;
    a_sql_uint32        free_pages;
    a_sql_uint32        bit_map_pages;
    a_sql_uint32        other_pages;
```



```
        a_bit_field         checksum : 1;
        a_bit_field         encrypted_tables : 1;
    } a_db_info;
```

成员

成员	说明
version	DBTools 版本号。
errorrtn	用于处理错误消息的回调例程。
msgrtn	用于处理信息消息的回调例程。
statusrtn	用于处理状态消息的回调例程。
dbbufsize	设置数据库文件名缓冲区的长度（例如，_MAX_PATH）。
dbnamebuffer	设置指向数据库文件名缓冲区的指针。
logbufsize	设置事务日志文件名缓冲区的长度（例如，_MAX_PATH）。
lognamebuffer	设置指向事务日志文件名缓冲区的指针。
mirrorbufsize	设置镜像文件名缓冲区的长度（例如，_MAX_PATH）。
mirrornamebuffer	设置指向镜像文件名缓冲区的指针。
collationnamebufsize	设置数据库归类名称和标签缓冲区的长度（最大长度为 129，其中包括空字符的空间）。
collationnamebuffer	设置指向数据库归类名称和标签缓冲区的指针。

connectparms	连接到数据库所需的参数。它们采用如下所示的连接字符串形式：
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db"`
	数据库服务器将由连接字符串的 START 参数启动。例如：
	`"START=d:\sqlany11\bin32\dbeng11.exe"`
	一个包括 START 参数的完整连接字符串示例：
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db;START=d:\sqlany11\bin32\dbeng11.exe"`
	有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

quiet	在不确认消息的情况下运行。
page_usage	如果报告页面使用统计，则为 1，否则为 0。

成员	说明
sysinfo	a_sysinfo 结构（请参见“a_sysinfo 结构”一节第 950 页）。
totals	指向 a_table_info 结构的指针（请参见“a_table_info 结构”一节第 951 页）。
file_size	数据库文件的大小。
free_pages	空闲页数。
bit_map_pages	数据库中位图页的数目。
other_pages	不属于表页、索引页、空闲页或位图页的页数。
checksum	如果为 1，则启用数据库页校验和；如果为 0，则禁用数据库页校验和。
encrypted_tables	如果为 1，则支持加密表，如果为 0，则禁用加密表。

另请参见

- “DBInfo 函数”一节第 917 页
- “使用回调函数”一节第 910 页

a_db_version_info 结构

保存有关用于创建数据库的 SQL Anywhere 版本的信息。

语法

```
typedef struct a_db_version_info {
        unsigned short  version;
        const char      *filename;
        a_db_version    created_version;
        MSG_CALLBACK      errorrtn;
        MSG_CALLBACK      msgrtn;
  } a_db_version_info;
```

成员

成员	说明
version	DBTools 版本号。
filename	要检查的数据库文件的名称。
created_version	设置为 a_db_version 类型的值，指示创建数据库文件的服务器版本。请参见“a_db_version 枚举”一节第 965 页。

成员	说明
errorrtn	用于处理错误消息的回调例程。
msgrtn	用于处理信息消息的回调例程。

另请参见

- “DBCreatedVersion 函数”一节第 916 页
- “a_db_version 枚举”一节第 965 页
- “使用回调函数”一节第 910 页

a_dblic_info 结构

保存包含许可信息的信息。使用此信息时必须严格遵守许可协议。

语法

```
typedef struct a_dblic_info {
    unsigned short      version;
    char                *exename;
    char                *username;
    char                *compname;
    a_sql_int32         nodecount;
    a_sql_int32         conncount;
    a_license_type      type;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    a_bit_field         quiet           : 1;
    a_bit_field         query_only      : 1;
    char                *installkey;
} a_dblic_info;
```

成员

成员	说明
version	DBTools 版本号。
exename	服务器可执行文件或许可文件的名称。
username	获得许可的用户名。
compname	获得许可的公司名。
nodecount	许可的节点数。
conncount	必须为 1000000L。
type	其值请参见 *lictype.h*。

成员	说明
errorrtn	用于处理错误消息的回调例程。
msgrtn	用于处理信息消息的回调例程。
quiet	在不打印消息 (1) 或打印消息 (0) 的情况下运行。
query_only	如果为 1，则只显示许可信息。如果为 0，则允许更改该信息。
installkey	仅供内部使用。设置为 NULL。

a_dbtools_info 结构

保存有关开始和完成使用 DBTools 库所需的信息。

语法

```
typedef struct a_dbtools_info {
    MSG_CALLBACK        errorrtn;
} a_dbtools_info;
```

成员

成员	说明
errorrtn	用于处理错误消息的回调例程。

另请参见

- “DBToolsFini 函数”一节第 920 页
- “DBToolsInit 函数”一节第 921 页
- “使用回调函数”一节第 910 页

an_erase_db 结构

保存有关使用 DBTools 库消除数据库所需的信息。

语法

```
typedef struct an_erase_db {
    unsigned short      version;
    const char *        dbname;
    MSG_CALLBACK        confirmrtn;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    a_bit_field         quiet : 1;
    a_bit_field         erase : 1;
    const char *        encryption_key;
} an_erase_db;
```

成员

成员	说明
version	DBTools 版本号。
dbname	要消除的数据库文件名。
confirmrtn	用于确认操作的回调例程。
errorrtn	用于处理错误消息的回调例程。
msgrtn	用于处理信息消息的回调例程。
quiet	在不打印消息 (1) 或打印消息 (0) 的情况下运行。
erase	在不进行确认 (1) 或进行确认 (0) 的情况下消除。
encryption_key	数据库文件的加密密钥。

另请参见

- “DBErase 函数”一节第 917 页
- “使用回调函数”一节第 910 页

a_name 结构

保存名称的链接列表。这由需要名称列表的其它结构使用。

语法

```
typedef struct a_name {
    struct a_name *next;
    char          name[1];
} a_name, * p_name;
```

成员

成员	说明
next	指向列表中下一个 a_name 结构的指针。
name	名称。

另请参见

- “a_translate_log 结构”一节第 952 页
- “a_validate_db 结构”一节第 962 页
- “an_unload_db 结构”一节第 957 页

a_remote_sql 结构

保存 dbremote 实用程序使用 DBTools 库时所需的信息。

语法

```
typedef struct a_remote_sql {
    short               version;
    MSG_CALLBACK        confirmrtn;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_QUEUE_CALLBACK  msgqueuertn;
    char *              connectparms;
    char *              transaction_logs;
    a_bit_field         receive : 1;
    a_bit_field         send : 1;
    a_bit_field         verbose : 1;
    a_bit_field         deleted : 1;
    a_bit_field         apply : 1;
    a_bit_field         batch : 1;
    a_bit_field         more : 1;
    a_bit_field         triggers : 1;
    a_bit_field         debug : 1;
    a_bit_field         rename_log : 1;
    a_bit_field         latest_backup : 1;
    a_bit_field         scan_log : 1;
    a_bit_field         link_debug : 1;
    a_bit_field         full_q_scan : 1;
    a_bit_field         no_user_interaction : 1;
    a_bit_field         _unused1 : 1;
    a_sql_uint32        max_length;
    a_sql_uint32        memory;
    a_sql_uint32        frequency;
    a_sql_uint32        threads;
    a_sql_uint32        operations;
    char *              queueparms;
    char *              locale;
    a_sql_uint32        receive_delay;
    a_sql_uint32        patience_retry;
    MSG_CALLBACK        logrtn;
    a_bit_field         use_hex_offsets : 1;
    a_bit_field         use_relative_offsets : 1;
    a_bit_field         debug_page_offsets : 1;
    a_sql_uint32        debug_dump_size;
    a_sql_uint32        send_delay;
    a_sql_uint32        resend_urgency;
    char *              include_scan_range;
    SET_WINDOW_TITLE_CALLBACK set_window_title_rtn;
    char *              default_window_title;
    MSG_CALLBACK        progress_msg_rtn;
    SET_PROGRESS_CALLBACK progress_index_rtn;
    char **             argv;
    a_sql_uint32        log_size;
    char *              encryption_key;
    const char *        log_file_name;
    a_bit_field         truncate_remote_output_file:1;
    char *              remote_output_file_name;
    MSG_CALLBACK        warningrtn;
    char *              mirror_logs;
} a_remote_sql;
```

成员

成员	说明
version	DBTools 版本号。
confirmrtn	指向打印给定消息并接受是或否响应的函数的指针，如果为是，则返回 TRUE，如果为否，则返回 FALSE。
errorrtn	指向打印给定错误消息的函数的指针。
msgrtn	指向打印给定信息性（非错误）消息的函数的指针。
msgqueuertn	指向一个函数的指针，该函数应休眠传递给它的毫秒数时间。当 DBRemoteSQL 正忙但想允许上层对消息进行处理时，使用 0 调用此函数。正常情况下，此例程应返回 MSGQ_SLEEP_THROUGH，或返回 MSGQ_SHUTDOWN_REQUESTED 以停止 SQL Remote 处理过程。

connectparms	连接到数据库所需的参数。对应 dbremote -c 选项。它们采用如下所示的连接字符串形式：
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db"`
	数据库服务器将由连接字符串的 START 参数启动。例如：
	`"START=d:\sqlany11\bin32\dbeng11.exe"`
	一个包括 START 参数的完整连接字符串示例：
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db;START=d:\sqlany11\bin32\dbeng11.exe"`
	有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

transaction_logs	指向一个字符串的指针，此字符串命名包含脱机事务日志的目录。与 dbremote 的 transaction_logs_directory 参数对应。
receive	如果 receive 为 true，表示收到消息。对应 dbremote -r 选项。 如果 receive 和 send 均为 false，则两者均被认为是 true。建议将两者均设置为 false。
send	如果 send 为 true，表示已发送消息。对应 dbremote -s 选项。 如果 receive 和 send 均为 false，则两者均被认为是 true。建议将两者均设置为 false。
verbose	如果为 true，则打印额外的信息。与 dbremote -v 选项对应。

成员	说明
deleted	应设置为 true。如果为 false，消息被应用后不会被删除。与 dbremote -p 选项对应。
apply	应设置为 true。如果为 false，则扫描消息，但不应用消息。与 dbremote -a 选项对应。
batch	如果为 true，则在应用消息和扫描日志后强制退出。与至少有一个具有 [always] 发送时间的用户的情况相同。如果为 false，则允许由远程用户的发送次数决定运行模式。
more	应设置为 true。
triggers	在大多数情况下，应设置为 false；否则，true 表示 DBRemoteSQL 复制触发器操作。与 dbremote -t 选项对应。
debug	如果设置为 true，则包括调试输出。
rename_log	如果设置为 true，则重命名并将重新启动日志。
latest_backup	如果设置为 true，则只处理备份日志。不发送来自活动日志的操作。与 dbremote -u 选项对应。
scan_log	保留；设置为 false。
link_debug	如果设置为 true，将打开对链接的调试。
full_q_scan	保留；设置为 false。
no_user_interaction	如果设置为 true，则不请求用户交互。
max_length	设置为消息可具有的最大长度（以字节为单位）。这会影响发送和接收。建议值为 50000。与 dbremote -l 选项对应。
memory	设置为在建立要发送的消息时所使用的内存缓冲区的最大大小（以字节为单位）。建议值至少为 2 * 1024 * 1024。与 dbremote -m 选项对应。
frequency	设置进来的消息的轮询频率。此值应设置为 max(1, receive_delay/60)。请参见下面的 receive_delay。
threads	设置应当用于应用消息的工作线程的数目。此值不能超过 50。与 dbremote -w 选项对应。
operations	当应用消息时会使用此值。在 DBRemoteSQL 至少具有此数量的未提交操作（插入、删除、更新）之前，提交将被忽略。与 dbremote -g 选项对应。

成员	说明
queueparms	保留；设置为 NULL。
locale	保留；设置为 NULL。
receive_delay	设置为在轮询之间等待新进来的消息的时间（以秒为单位）。建议值为 60。与 dbremote -rd 选项对应。
patience_retry	设置为 DBRemoteSQL 在假定所期待的消息丢失之前应等待的轮询进来的消息的次数。例如，如果 patience_retry 为 3，则 DBRemoteSQL 尝试最多三次来接收缺失的消息。然后，它会发送一个重新发送请求。建议值为 1。与 dbremote -rp 选项对应。
logrtn	指向将给定消息打印到日志文件的函数的指针。这些消息无需用户进行查看。
use_hex_offsets	如果想以十六进制表示法显示日志偏移，请将其设置为 true；否则，将使用十进制表示法。
use_relative_offsets	如果想以相对于当前日志文件开始位置的方式显示日志偏移，请将其设置为 true。如果想显示相对于开始时间的日志偏移，则将其设置为 false。
debug_page_offsets	保留；设置为 false。
debug_dump_size	保留；设置为 0。
send_delay	设置扫描日志文件以查找要发送的新操作的时间间隔（以秒为单位）。设置为零可允许 DBRemoteSQL 根据用户发送时间选择一个合适的值。与 dbremote -sd 选项对应。
resend_urgency	设置 DBRemoteSQL 在知道用户需要重新扫描之后，在执行完全日志扫描之前所等待的时间（以秒为单位）。设置为零可允许 DBRemoteSQL 根据用户发送时间及其收集的其它信息选择一个合适的值。与 dbremote -ru 选项对应。
include_scan_range	保留；设置为 NULL。
set_window_title_rtn	指向一个重置窗口标题的函数的指针（仅适用于 Windows）。标题可以是 "*database_name*（接收、扫描或发送）- *default_window_title*"。
default_window_title	指向缺省窗口标题字符串的指针。
progress_msg_rtn	指向显示进程消息的函数的指针。

成员	说明
progress_index_rtn	指向更新进度条状态的函数的指针。该函数具有两个无符号整型变量 *index* 和 *max*。第一次调用时，这两个值分别为最小和最大值（如 0、100）。后续调用时，第一个参数为当前索引值（例如，0 到 100 之间的值），而第二个参数始终为 0。
argv	指向所分析命令行的指针（字符串指针矢量）。如果不为 NULL，则 DBRemoteSQL 将调用消息例程来显示前缀不是 -c、-cq 或 -ek 的每个命令行参数。
log_size	当联机事务日志的大小大于此值时，DBRemoteSQL 将重命名并重新启动联机事务日志。与 dbremote -x 选项对应。
encryption_key	指向加密密钥的指针。与 dbremote -ek 选项对应。
log_file_name	指向 DBRemoteSQL 输出日志的名称的指针，消息回调会将其输出打印到该输出日志。如果 *send* 为 true，错误日志会被发送到统一数据库（除非此指针为 NULL）。
truncate_remote_output_file	设置为 true 将使得远程输出文件被截断，而非附加到远程输出文件。请参见下文。与 dbremote -rt 选项对应。
remote_output_file_name	指向 DBRemoteSQL 远程输出文件的名称的指针。与 dbremote -ro 或 -rt 选项对应。
warningrtn	指向打印给定警告消息的函数的指针。如果为 NULL，将改为调用 *errorrtn* 函数。
mirror_logs	指向目录名称的指针，该目录包含脱机镜像事务日志。与 dbremote -ml 选项对应。

dbremote 工具在处理任何命令行选项前会设置以下缺省值：

- version = DB_TOOLS_VERSION_NUMBER
- argv =（传递给应用程序的变量矢量）
- deleted = TRUE
- apply = TRUE
- more = TRUE
- link_debug = FALSE
- max_length = 50000
- memory = 2 * 1024 * 1024
- frequency = 1
- threads = 0
- receive_delay = 60
- send_delay = 0
- log_size = 0
- patience_retry = 1
- resend_urgency = 0
- log_file_name =（从命令行进行设置）
- truncate_remote_output_file = FALSE
- remote_output_file_name = NULL
- no_user_interaction = TRUE（如果没有可用的用户界面）
- errorrtn =（相应例程的地址）
- msgrtn =（相应例程的地址）
- confirmrtn =（相应例程的地址）
- msgqueuertn =（相应例程的地址）
- logrtn =（相应例程的地址）
- warningrtn =（相应例程的地址）
- set_window_title_rtn =（相应例程的地址）
- progress_msg_rtn =（相应例程的地址）
- progress_index_rtn =（相应例程的地址）

另请参见

- “DBRemoteSQL 函数”一节第 919 页
- “dbmlsync 的 DBTools 接口” 《MobiLink - 客户端管理》

a_sync_db 结构

保存 dbmlsync 实用程序使用 DBTools 库时所需的信息。

语法

```
typedef struct a_sync_db {
    unsigned short      version;
    char *              connectparms;
    char *              publication;
    const char *        offline_dir;
    char *              extended_options;
    char *              script_full_path;
    const char *        include_scan_range;
```



```
const char *        raw_file;
MSG_CALLBACK        confirmrtn;
MSG_CALLBACK        errorrtn;
MSG_CALLBACK        msgrtn;
MSG_CALLBACK        logrtn;
a_sql_uint32        debug_dump_size;
a_sql_uint32        dl_insert_width;
a_bit_field         verbose                 : 1;
a_bit_field         debug                   : 1;
a_bit_field         debug_dump_hex          : 1;
a_bit_field         debug_dump_char         : 1;
a_bit_field         debug_page_offsets      : 1;
a_bit_field         use_hex_offsets         : 1;
a_bit_field         use_relative_offsets    : 1;
a_bit_field         output_to_file          : 1;
a_bit_field         output_to_mobile_link   : 1;
a_bit_field         dl_use_put              : 1;
a_bit_field         kill_other_connections  : 1;
a_bit_field         retry_remote_behind     : 1;
a_bit_field         ignore_debug_interrupt  : 1;
SET_WINDOW_TITLE_CALLBACK set_window_title_rtn;
char *              default_window_title;
MSG_QUEUE_CALLBACK  msgqueuertn;
MSG_CALLBACK        progress_msg_rtn;
SET_PROGRESS_CALLBACK progress_index_rtn;
char **             argv;
char **             ce_argv;
a_bit_field         connectparms_allocated  : 1;
a_bit_field         entered_dialog          : 1;
a_bit_field         used_dialog_allocation  : 1;
a_bit_field         ignore_scheduling       : 1;
a_bit_field         ignore_hook_errors      : 1;
a_bit_field         changing_pwd            : 1;
a_bit_field         prompt_again            : 1;
a_bit_field         retry_remote_ahead      : 1;
a_bit_field         rename_log              : 1;
a_bit_field         hide_conn_str           : 1;
a_bit_field         hide_ml_pwd             : 1;
a_sql_uint32        dlg_launch_focus;
char *              mlpassword;
char *              new_mlpassword;
char *              verify_mlpassword;
a_sql_uint32        pub_name_cnt;
char **             pub_name_list;
USAGE_CALLBACK      usage_rtn;
a_sql_uint32        log_size;
a_sql_uint32        hovering_frequency;
a_bit_short         ignore_hovering         : 1;
a_bit_short         verbose_upload          : 1;
a_bit_short         verbose_upload_data     : 1;
a_bit_short         verbose_download        : 1;
a_bit_short         verbose_download_data   : 1;
a_bit_short         autoclose               : 1;
a_bit_short         ping                    : 1;
a_bit_short         _unused                 : 9;
char *              encryption_key;
a_syncpub *         upload_defs;
const char *        log_file_name;
char *              user_name;
a_bit_short         verbose_minimum         : 1;
a_bit_short         verbose_hook            : 1;
a_bit_short         verbose_row_data        : 1;
a_bit_short         verbose_row_cnts        : 1;
a_bit_short         verbose_option_info     : 1;
```

```
    a_bit_short         strictly_ignore_trigger_ops : 1;
    a_bit_short         _unused2                    : 10;
    a_sql_uint32        est_upld_row_cnt;
    STATUS_CALLBACK     status_rtn;
    MSG_CALLBACK        warningrtn;
    char **             ce_reproc_argv;
    a_bit_short         upload_only                 : 1;
    a_bit_short         download_only               : 1;
    a_bit_short         allow_schema_change         : 1;
    a_bit_short         dnld_gen_num                : 1;
    a_bit_short         _unused3                    :12;
    const char *        apply_dnld_file;
    const char *        create_dnld_file;
    char *              sync_params;
    const char *        dnld_file_extra;
    COMServer *         com_server;
    a_bit_short         trans_upload                : 1;
    a_bit_short         continue_download           : 1;
    a_bit_short         lite_blob_handling          : 1;
    a_sql_uint32        dnld_read_size;
    a_sql_uint32        dnld_fail_len;
    a_sql_uint32        upld_fail_len;
    a_bit_short         persist_connection          :1;
    a_bit_short         verbose_protocol            :1;
    a_bit_short         no_stream_compress          :1;
    a_bit_short         _unused4                    :13;
    const char *        encrypted_stream_opts;
    a_sql_uint32        no_offline_logscan;
    a_bit_short         server_mode :1;
    a_bit_short         allow_outside_connect :1;
    a_bit_short         prompt_for_encrypt_key:1;
    a_bit_short         com_server_mode:1;
    a_bit_short         verbose_server:1;
    a_bit_short         _unused5 :11;
    a_sql_uint32        server_port;
    char *              preload_dlls;
    char *              sync_profile;
    char *              sync_opt;
    a_syncpub *         last_upload_def;
} a_sync_db;
```

成员

成员	说明
version	DBTools 版本号。

成员	说明

connectparms	连接到数据库所需的参数。它们采用如下所示的连接字符串形式：
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db"`
	数据库服务器将由连接字符串的 START 参数启动。例如：
	`"START=d:\sqlany11\bin32\dbeng11.exe"`
	一个包括 START 参数的完整连接字符串示例：
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db;START=d:\sqlany11\bin32\dbeng11.exe"`
	有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

publication	不建议使用；使用 NULL。
offline_dir	日志目录，在选项后面的命令行上指定。
extended_options	扩展选项，由 -e 指定。
script_full_path	不建议使用；使用 NULL。
include_scan_range	保留；使用 NULL。
raw_file	保留；使用 NULL。
confirmrtn	保留；使用 NULL。
errorrtn	显示错误消息的函数。
msgrtn	将消息写入用户接口（或日志文件）的函数。
logrtn	将消息仅写入日志文件的函数。
debug_dump_size	保留；使用 0。
dl_insert_width	保留；使用 0。
verbose	不建议使用；使用 0。
debug	保留；使用 0。
debug_dump_hex	保留；使用 0。
debug_dump_char	保留；使用 0。

成员	说明
debug_page_offsets	保留；使用 0。
use_hex_offsets	保留；使用 0。
use_relative_offsets	保留；使用 0。
output_to_file	保留；使用 0。
output_to_mobile_link	保留；使用 1。
dl_use_put	保留；使用 0。
kill_other_connections	如果指定了 -d 选项，则为 TRUE。
retry_remote_behind	如果指定了 -r 或 -rb，则为 TRUE。
ignore_debug_interrupt	保留；使用 0。
set_window_title_rtn	通过对其调用可更改 dbmlsync 窗口标题的函数（仅限 Windows）。
default_window_title	将要在窗口标题中显示的程序的名称（如 DBMLSync）。
msgqueuertn	当 DBMLSync 要休眠时调用的函数。该参数指定休眠周期（以毫秒为单位）。正如 *dllapi.h* 中的定义，此函数应返回以下值。 ● MSGQ_SLEEP_THROUGH，指示例程休眠请求的毫秒数。多数情况下，这是应返回的值。 ● MSGQ_SHUTDOWN_REQUESTED，指示希望尽快终止同步。 ● MSGQ_SYNC_REQUESTED，指示例程休眠的时间少于请求的毫秒数，如果当前没有正在进行同步，则立即开始下一个同步。
progress_msg_rtn	用于改变状态窗口中进度条上的文本的函数。
progress_index_rtn	用于更新进度条状态的函数。
argv	用于此次运行的 argv 数组；此数组的最后一个元素必须为 NULL。
ce_argv	保留；使用 NULL。
connectparms_allocated	保留；使用 0。

成员	说明
entered_dialog	保留；使用 0。
used_dialog_allocation	保留；使用 0。
ignore_scheduling	如果指定了 -is，则为 TRUE。
ignore_hook_errors	如果指定了 -eh，则为 TRUE。
changing_pwd	如果指定了 -mn，则为 TRUE。
prompt_again	保留—使用 0。
retry_remote_ahead	如果指定了 -ra，则为 TRUE。
rename_log	如果指定了 -x，则为 TRUE，此时会重命名并重新启动日志文件。
hide_conn_str	除非指定 -vc，否则为 TRUE。
hide_ml_pwd	除非指定 -vp，否则为 TRUE。
dlg_launch_focus	保留；使用 0。
mlpassword	用 -mp 指定的 MobiLink 口令，否则为 NULL。
new_mlpassword	用 -mn 指定的新的 MobiLink 口令，否则为 NULL。
verify_mlpassword	保留；使用 NULL。
pub_name_cnt	不建议使用；使用 0。
pub_name_list	不建议使用；使用 NULL。
usage_rtn	保留；使用 NULL。
log_size	日志大小（以字节为单位），用 -x 指定；否则为 0。
hovering_frequency	用 -pp 设置的悬停频率（以秒为单位）。
ignore_hovering	如果指定了 -p，则为 True。
verbose_upload	如果指定了 -vu，则为 True。
verbose_upload_data	保留；使用 0。
verbose_download	保留；使用 0。

成员	说明
verbose_download_data	保留；使用 0。
autoclose	如果指定了 -k，则为 TRUE。
ping	如果指定了 -pi，则为 TRUE。
encryption_key	用 -ek 指定的数据库键。
upload_defs	将要一起上载的发布的链接列表—请参见 a_syncpub。
log_file_name	用 -o 或 -ot 指定的数据库服务器消息日志文件名。
user_name	用 -u 指定的 MobiLink 用户名。
verbose_minimum	如果指定了 -v，则为 TRUE。
verbose_hook	如果指定了 -vs，则为 TRUE。
verbose_row_data	如果指定了 -vr，则为 TRUE。
verbose_row_cnts	如果指定了 -vn，则为 TRUE。
verbose_option_info	如果指定了 -vo，则为 TRUE。
strictly_ignore_trigger_ops	保留；使用 0。
est_upld_row_cnt	预计要上载的行数，用 -urc 指定。
status_rtn	保留；使用 NULL。
warningrtn	显示警告消息的函数。
ce_reproc_argv	保留，使用 NULL。
upload_only	如果指定了 -uo，则为 True。
download_only	如果指定了 -ds，则为 TRUE。
allow_schema_change	如果指定了 -sc，则为 TRUE。
dnld_gen_num	如果指定了 -bg，则为 TRUE。
apply_dnld_file	用 -ba 指定的文件，否则为 NULL。
create_dnld_file	用 -bc 指定的文件，否则为 NULL。
sync_params	用户验证参数—使用 -ap 指定。

成员	说明
dnld_file_extra	用 -be 指定的字符串。
com_server	保留；使用 NULL。
trans_upload	如果指定了 -tu，则为 TRUE。
continue_download	如果指定了 -dc，则为 TRUE。
dnld_read_size	由 -drs 选项指定的值。
dnld_fail_len	保留；使用 0。
upld_fail_len	保留；使用 0。
persist_connection	如果在命令行上指定了 -pp，则为 TRUE。
verbose_protocol	保留；使用 0。
no_stream_compress	保留；使用 0。
encrypted_stream_opts	保留；使用 NULL。
no_offline_logscan	如果指定了 -do，则为 TRUE
server_mode	如果指定了 -sm，则为 TRUE
allow_outside_connect	保留；使用 0。
prompt_for_encrypt_key	保留；使用 0。
com_server_mode	保留；使用 0。
verbose_server	保留；使用 0。
server_port	-sp 选项的值。
preload_dlls	保留；使用 NULL。
sync_profile	由 -sp 选项指定的值。
sync_opt	保留；使用 NULL。
last_upload_def	保留；使用 NULL。

有些成员与可从 dbmlsync 命令行实用程序访问的功能相对应。应根据数据类型为未使用的成员指派值 0、FALSE 或 NULL。

有关更多说明，请参见 *dbtools.h* 头文件。

有关详细信息，请参见“dbmlsync 语法”一节 《MobiLink - 客户端管理》。

另请参见

- “dbmlsync 的 DBTools 接口” 《MobiLink - 客户端管理》
- “DBSynchronizeLog 函数”一节第 920 页

a_syncpub 结构

保存 dbmlsync 实用程序所需的信息。

语法

```
typedef struct a_syncpub {
    struct a_syncpub *  next;
    char *              pub_name;
    char *              ext_opt;
    a_bit_field         alloced_by_dbsync: 1;
} a_syncpub;
```

成员

成员	说明
a_syncpub	指向列表中下一个节点的指针，NULL 表示最后一个节点。
pub_name	为此 -n 选项指定的发布名称。命令行上 -n 后面就是这个字符串。
ext_opt	使用 -eu 选项指定的扩展选项。
alloced_by_dbsync	保留；使用 FALSE。

另请参见

- “dbmlsync 的 DBTools 接口” 《MobiLink - 客户端管理》

a_sysinfo 结构

保存有关 dbinfo 和 dbunload 实用程序使用 DBTools 库时所需的信息。

```
typedef struct a_sysinfo {
    a_bit_field     valid_data          : 1;
    a_bit_field     blank_padding       : 1;
    a_bit_field     case_sensitivity    : 1;
    a_bit_field     encryption          : 1;
    char            default_collation[11];
    unsigned short  page_size;
} a_sysinfo;
```


成员

成员	说明
valid_date	用于指示是否设置以下值的位字段。
blank_padding	如果在该数据库中使用空白填充，则为 1，否则为 0。
case_sensitivity	如果数据库区分大小写，则为 1，否则为 0。
encryption	如果对数据库进行了加密，则为 1，否则为 0。
default_collation	数据库的归类序列。
page_size	数据库的页面大小。

另请参见

- “a_db_info 结构”一节第 931 页

a_table_info 结构

保存有关 a_db_info 结构所需表的信息。

语法

```
typedef struct a_table_info {
    struct a_table_info *next;
    a_sql_uint32        table_id;
    a_sql_uint32        table_pages;
    a_sql_uint32        index_pages;
    a_sql_uint32        table_used;
    a_sql_uint32        index_used;
    char *              table_name;
    a_sql_uint32        table_used_pct;
    a_sql_uint32        index_used_pct;
} a_table_info;
```

成员

成员	说明
next	列表中的下一个表。
table_id	该表的 ID 号。
table_pages	表页数。
index_pages	索引页数。
table_used	表页中使用的字节数。

成员	说明
index_used	索引页中使用的字节数。
table_name	表的名称。
table_used_pct	表空间的使用百分比。
index_used_pct	索引空间的使用百分比。

另请参见

- “a_db_info 结构”一节第 931 页

a_translate_log 结构

保存有关使用 DBTools 库转换事务日志所需的信息。

语法

```
typedef struct a_translate_log {
    unsigned short      version;
    const char *        connectparms;
    const char *        logname;
    const char *        sqlname;
    const char *        encryption_key;
    const char *        logs_dir;
    p_name              userlist;
    a_sql_uint32        since_time;
    MSG_CALLBACK        confirmrtn;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        logrtn;
    MSG_CALLBACK        statusrtn;
    char                userlisttype;
    a_bit_field         quiet                   : 1;
    a_bit_field         remove_rollback         : 1;
    a_bit_field         ansi_sql                : 1;
    a_bit_field         since_checkpoint        : 1;
    a_bit_field         replace                 : 1;
    a_bit_field         include_trigger_trans   : 1;
    a_bit_field         comment_trigger_trans   : 1;
    a_bit_field         debug                   : 1;
    a_bit_field         debug_sql_remote        : 1;
    a_bit_field         debug_dump_hex          : 1;
    a_bit_field         debug_dump_char         : 1;
    a_bit_field         debug_page_offsets      : 1;
    a_bit_field         omit_comments           : 1;
    a_bit_field         use_hex_offsets         : 1;
    a_bit_field         use_relative_offsets    : 1;
    a_bit_field         include_audit           : 1;
    a_bit_field         chronological_order     : 1;
    a_bit_field         force_recovery          : 1;
    a_bit_field         include_subsets         : 1;
    a_bit_field         force_chaining          : 1;
    a_bit_field         generate_reciprocals    : 1;
    a_bit_field         match_mode              : 1;
```



```
    a_bit_field         show_undo               : 1;
    a_bit_field         extra_audit             : 1;
    a_sql_uint32        debug_dump_size;
    a_sql_uint32        recovery_ops;
    a_sql_uint32        recovery_bytes;
    const char *        include_source_sets;
    const char *        include_destination_sets;
    const char *        include_scan_range;
    const char *        repserver_users;
    const char *        include_tables;
    const char *        include_publications;
    const char *        queueparms;
    const char *        match_pos;
    a_bit_field         leave_output_on_error   : 1;
} a_translate_log;
```

成员

成员	说明
version	DBTools 版本号。

connectparms	连接到数据库所需的参数。它们采用如下所示的连接字符串形式：
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db"`
	数据库服务器将由连接字符串的 START 参数启动。例如：
	`"START=d:\sqlany11\bin32\dbeng11.exe"`
	一个包括 START 参数的完整连接字符串示例：
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db;START=d:\sqlany11\bin32\dbeng11.exe"`
	有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

logname	事务日志文件的名称。如果为 NULL，则没有日志。
sqlname	SQL 输出文件的名称。如果为 NULL，则此名称将基于事务日志文件的名称（-n 设置该字符串）。
encryption_key	指定数据库加密密钥（-ek 设置为字符串）。
logs_dir	事务日志目录（-m *dir* 设置字符串）；必须设置 sqlname 且 connect_parms 必须为 NULL。
userlist	用户名的链接列表。相当于 -u 用户 1,... 或 -x 用户 1,...选择或忽略列出用户的事务。
since_time	从给定时间（-j <时间> 设置该时间）之前最后一次检查点操作开始输出。从 0001 年 1 月 1 日以来的分钟数。
confirmrtn	用于确认操作的回调例程。

成员	说明
errorrtn	用于处理错误消息的回调例程。
msgrtn	用于处理信息消息的回调例程。
logrtn	只将消息写入日志文件的回调例程。
statusrtn	用于处理状态消息的回调例程。
userlisttype	除非您要包括或排除一组用户，否则设置为 DBTRAN_INCLUDE_ALL。DBTRAN_INCLUDE_SOME 对应 -u，DBTRAN_EXCLUDE_SOME 对应 -x。
quiet	设置为 TRUE 可在不打印消息的情况下运行 (-y)。
remove_rollback	通常设置为 TRUE；如果想在输出中包括回退事务，则设置为 FALSE（相当于 -a）。
ansi_sql	如果想生成 ANSI 标准的 SQL 事务，则设置为 TRUE（相当于 -s）。
since_checkpoint	如果想从最近的检查点开始输出，则设置为 TRUE（相当于 -f）。
replace	替换现有的 SQL 文件而无需确认（相当于 -y）。
include_trigger_trans	设置为 TRUE 可将触发器生成的事务包括在内（相当于 -g、-sr 或 -t）。
comment_trigger_trans	设置为 TRUE 可将触发器生成的事务作为注释包括在内（相当于 -z）。
debug	保留；设置为 FALSE。
debug_sql_remote	保留，使用 FALSE。
debug_dump_hex	保留，使用 FALSE。
debug_dump_char	保留，使用 FALSE。
debug_page_offsets	保留，使用 FALSE。
use_hex_offsets	保留，使用 FALSE。
use_relative_offsets	保留，使用 FALSE。
include_audit	保留，使用 FALSE。
chronological_order	保留，使用 FALSE。
force_recovery	保留，使用 FALSE。

成员	说明
include_subsets	保留，使用 FALSE。
force_chaining	保留，使用 FALSE。
generate_reciprocals	保留，使用 FALSE。
match_mode	保留，使用 FALSE。
show_undo	保留，使用 FALSE。
debug_dump_size	保留，使用 0。
recovery_ops	保留，使用 0。
recovery_bytes	保留，使用 0。
include_source_sets	保留，使用 NULL。
include_destination_sets	保留，使用 NULL。
include_scan_range	保留，使用 NULL。
repserver_users	保留，使用 NULL。
include_tables	保留，使用 NULL。
include_publications	保留，使用 NULL。
queueparms	保留，使用 NULL。
match_pos	保留，使用 NULL。
leave_output_on_error	如果要在检测到文件损坏时离开生成的 .SQL 文件，则设置为 TRUE（等效于 -k）

这些成员与可从 dbtran 实用程序访问的功能相对应。

有关更多说明，请参见 *dbtools.h* 头文件。

另请参见

- “DBTranslateLog 函数”一节第 921 页
- “a_name 结构”一节第 936 页
- “dbtran_userlist_type 枚举”一节第 966 页
- “使用回调函数”一节第 910 页

a_truncate_log 结构

保存有关使用 DBTools 库截断事务日志所需的信息。

语法

```
typedef struct a_truncate_log {
    unsigned short      version;
    const char *        connectparms;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    a_bit_field         quiet           : 1;
    a_bit_field         server_backup   : 1;
    char                truncate_interrupted;
} a_truncate_log;
```

成员

成员	说明
version	DBTools 版本号。

connectparms	连接到数据库所需的参数。它们采用如下所示的连接字符串形式：
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db"`
	数据库服务器将由连接字符串的 START 参数启动。例如：
	`"START=d:\sqlany11\bin32\dbeng11.exe"`
	一个包括 START 参数的完整连接字符串示例：
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db;START=d:\sqlany11\bin32\dbeng11.exe"`
	有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

errorrtn	用于处理错误消息的回调例程。
msgrtn	用于处理信息消息的回调例程。
quiet	在不打印消息 (1) 或打印消息 (0) 的情况下运行。
server_backup	设置为 1 时，指示使用 BACKUP DATABASE 在服务器上进行备份。相当于 dbbackup -s 选项。
truncate_interrupted	指示该操作已被中断。

另请参见

- “DBTruncateLog 函数”一节第 922 页
- “使用回调函数”一节第 910 页

an_unload_db 结构

保存有关使用 DBTools 库卸载数据库或者为 SQL Remote 抽取远程数据库所需的信息。指示由 dbxtract SQL Remote 抽取实用程序使用的字段。

语法

```
typedef struct an_unload_db {
    unsigned short      version;
    const char *        connectparms;
    const char *        temp_dir;
    const char *        reload_filename;
    char *              reload_connectparms;
    char *              reload_db_filename;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        statusrtn;
    MSG_CALLBACK        confirmrtn;
    char                unload_type;
    char                verbose;
    char                escape_char;
    char                unload_interrupted;
    a_bit_field         unordered               : 1;
    a_bit_field         no_confirm              : 1;
    a_bit_field         use_internal_unload     : 1;
    a_bit_field         refresh_mat_view        : 1;
    a_bit_field         table_list_provided     : 1;
    a_bit_field         exclude_tables          : 1;
    a_bit_field         preserve_ids            : 1;
    a_bit_field         replace_db              : 1;
    a_bit_short         escape_char_present     : 1;
    a_bit_short         use_internal_reload     : 1;
    a_bit_field         recompute               : 1;
    a_bit_field         make_auxiliary          : 1;
    a_bit_field         encrypted_tables        : 1;
    a_bit_field         remove_encrypted_tables : 1;
    a_bit_field         extract                 : 1;
    a_bit_field         start_subscriptions     : 1;
    a_bit_field         exclude_foreign_keys    : 1;
    a_bit_field         exclude_procedures      : 1;
    a_bit_field         exclude_triggers        : 1;
    a_bit_field         exclude_views           : 1;
    a_bit_field         isolation_set           : 1;
    a_bit_field         include_where_subscribe : 1;
    a_bit_field         exclude_hooks           : 1;
    a_bit_field         startline_name          : 1;
    a_bit_field         debug                   : 1;
    a_bit_field         compress_output         : 1;
    a_bit_field         schema_reload           : 1;
    a_bit_field         genscript               : 1;
    a_bit_field         runscript               : 1;
    a_bit_field         display_create          : 1;
    a_bit_field         display_create_dbinit   : 1;
    a_bit_field         preserve_identity_values: 1;
    const char *        ms_filename;
    int                 ms_reserve;
    int                 ms_size;
    long                notemp_size;
    p_name              table_list;
    a_sysinfo           sysinfo;
    const char *        remote_dir;
    const char *        subscriber_username;
```

```
    unsigned short      isolation_level;
    const char *        site_name;
    const char *        template_name;
    char *              reload_db_logname;
    const char *        encryption_key;
    const char *        encryption_algorithm;
    unsigned short      reload_page_size;
    const char *        locale;
    const char *        startline;
    const char *        startline_old;
} an_unload_db;
```

成员

成员	说明
version	DBTools 版本号。

connectparms	连接到数据库所需的参数。它们采用如下所示的连接字符串形式：
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db"`
	数据库服务器将由连接字符串的 START 参数启动。例如：
	`"START=d:\sqlany11\bin32\dbeng11.exe"`
	一个包括 START 参数的完整连接字符串示例：
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db;START=d:\sqlany11\bin32\dbeng11.exe"`
	有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

temp_dir	用于卸载数据文件的目录。
reload_filename	dbunload -r 选项，与 *reload.sql* 类似。
reload_connectparms	用于重装数据库的用户 ID、口令、数据库。
reload_db_filename	要创建的重装数据库的文件名。
errorrtn	用于处理错误消息的回调例程。
msgrtn	用于处理信息消息的回调例程。
statusrtn	用于处理状态消息的回调例程。
confirmrtn	用于确认操作的回调例程。
unload_type	请参见“dbunload 类型枚举”一节第 967 页。
verbose	请参见“详细枚举”一节第 968 页。

成员	说明
escape_char	当 escape_char_present 为 TRUE 时使用。
unload_interrupted	在卸载被中断的情况下设置。
unordered	dbunload -u 设置为 TRUE。
no_confirm	dbunload -y 设置为 TRUE。
use_internal_unload	dbunload -ii/-ix 设置为 TRUE。dbunload -xi/-xx 设置为 FALSE。
refresh_mat_view	dbunload -g 设置为 TRUE。
table_list_provided	dbunload -e *list* 或 -i 设置为 TRUE。
exclude_tables	dbunload -e 设置为 TRUE。dbunload -i（未提供文档）设置为 FALSE。
preserve_ids	dbunload 设置为 TRUE/-m 设置为 FALSE。
replace_db	dbunload -ar 设置为 TRUE。
escape_char_present	dbunload -p 设置为 TRUE。注意必须设置 escape_char。
use_internal_reload	通常设置为 TRUE；-ix/-xx 设置为 FALSE；-ii/-xi 设置为 TRUE。
recompute	dbunload -dc 设置为 TRUE。重新计算所有计算列。
make_auxiliary	dbunload -k 设置为 TRUE。创建辅助目录（与诊断跟踪一起使用）。
encrypted_tables	dbunload -et 设置为 TRUE。在新数据库中启用加密表（与 -an 或 -ar 一起使用）。
remove_encrypted_tables	dbunload -er 设置为 TRUE。从加密表中删除密钥。
extract	如果是 dbxtract，则为 TRUE，否则为 FALSE。
start_subscriptions	缺省情况下，dbxtract 为 TRUE，-b 则设置为 FALSE。
exclude_foreign_keys	dbxtract -xf 设置为 TRUE。
exclude_procedures	dbxtract -xp 设置为 TRUE。
exclude_triggers	dbxtract -xt 设置为 TRUE。
exclude_views	dbxtract -xv 设置为 TRUE。

成员	说明
isolation_set	dbxtract -l 设置为 TRUE。
include_where_subscribe	dbxtract -f 设置为 TRUE。
exclude_hooks	dbxtract -hx 设置为 TRUE。
startline_name	（内部使用）
debug	（内部使用）
compress_output	dbunload -cp 设置为 TRUE。
schema_reload	（内部使用）
genscript	（内部使用）
runscript	（内部使用）
display_create	-cm 设置为 TRUE
display_create_dbinit	-cm dbinit 设置为 TRUE
preserve_identity_values	dbunload -l 设置为 TRUE
ms_filename	（内部使用）
ms_reserve	（内部使用）
ms_size	（内部使用）
notemp_size	（内部使用）
table_list	选择性表列表。
sysinfo	（内部使用）
remote_dir	（类似 temp_dir），但用于在服务器端内部卸载。
subscriber_username	dbxtract 的参数。
isolation_level	dbxtract -l 设置为值。
site_name	对于 dbxtract：指定一个站点名称。
template_name	对于 dbxtract：指定一个模板名称。
reload_db_logname	该重装数据库的日志文件名。

成员	说明
encryption_key	-ek 设置为字符串。
encryption_algorithm	-ea 设置为 "AES"、"AES256"、"AES_FIPS" 或 "AES256_FIPS" 之一。
reload_page_size	dbunload -ap 设置为值。设置重建数据库的页面大小。
locale	（内部使用）区域设置（语言和字符集）。
startline	（内部使用）
startline_old	（内部使用）

这些成员与可从 dbunload 和 dbxtract 实用程序访问的功能相对应。

有关更多说明，请参见 *dbtools.h* 头文件。

另请参见

- “DBUnload 函数”一节第 922 页
- “a_name 结构”一节第 936 页
- “dbunload 类型枚举”一节第 967 页
- “详细枚举”一节第 968 页
- “使用回调函数”一节第 910 页

an_upgrade_db 结构

保存有关使用 DBTools 库升级数据库所需的信息。

语法

```
typedef struct an_upgrade_db {
    unsigned short      version;
    const char *        connectparms;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        statusrtn;
    a_bit_field         quiet           : 1;
    a_bit_field         jconnect        : 1;
} an_upgrade_db;
```

成员

成员	说明
version	DBTools 版本号。

成员	说明

connectparms	连接到数据库所需的参数。它们采用如下所示的连接字符串形式：
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db"`
	数据库服务器将由连接字符串的 START 参数启动。例如：
	`"START=d:\sqlany11\bin32\dbeng11.exe"`
	一个包括 START 参数的完整连接字符串示例：
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db;START=d:\sqlany11\bin32\dbeng11.exe"`
	有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

errorrtn	用于处理错误消息的回调例程。
msgrtn	用于处理信息消息的回调例程。
statusrtn	用于处理状态消息的回调例程。
quiet	在不打印消息 (1) 或打印消息 (0) 的情况下运行。
jconnect	升级数据库以包括 jConnect 过程。

另请参见

- “DBUpgrade 函数”一节第 923 页
- “使用回调函数”一节第 910 页

a_validate_db 结构

保存有关使用 DBTools 库校验数据库所需的信息。

语法

```
typedef struct a_validate_db {
    unsigned short      version;
    const char *        connectparms;
    p_name              tables;
    MSG_CALLBACK        errorrtn;
    MSG_CALLBACK        msgrtn;
    MSG_CALLBACK        statusrtn;
    a_bit_field         quiet : 1;
    a_bit_field         index : 1;
    a_validate_type     type;
} a_validate_db;
```

成员

成员	说明
version	DBTools 版本号。

connectparms	连接到数据库所需的参数。它们采用如下所示的连接字符串形式：
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db"`
	数据库服务器将由连接字符串的 START 参数启动。例如：
	`"START=d:\sqlany11\bin32\dbeng11.exe"`
	一个包括 START 参数的完整连接字符串示例：
	`"UID=DBA;PWD=sql;DBF=samples-dir\demo.db;START=d:\sqlany11\bin32\dbeng11.exe"`
	有关连接参数的列表，请参见“连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

tables	指向表名的链接列表的指针。
errorrtn	用于处理错误消息的回调例程。
msgrtn	用于处理信息消息的回调例程。
statusrtn	用于处理状态消息的回调例程。
quiet	在不打印消息 (1) 或打印消息 (0) 的情况下运行。
index	校验索引。
type	请参见“a_validate_type 枚举”一节第 967 页。

另请参见

- “DBValidate 函数”一节第 923 页
- “a_name 结构”一节第 936 页
- “a_validate_type 枚举”一节第 967 页
- 有关回调函数的详细信息，请参见“使用回调函数”一节第 910 页。

DBTools 枚举类型

本节列出由 DBTools 库使用的枚举类型。这些枚举按字母顺序列出。

空白填充枚举

在“a_create_db 结构”一节第 929 页中用于指定 blank_pad 的值。

语法

```
enum {
    NO_BLANK_PADDING,
    BLANK_PADDING
};
```

参数

值	说明
NO_BLANK_PADDING	不使用空白填充。
BLANK_PADDING	使用空白填充。

另请参见

- “a_create_db 结构”一节第 929 页

a_chkpt_log_type 枚举

在“a_backup_db 结构”一节第 925 页中用于控制检查点日志的复制。

语法

```
typedef enum {
    BACKUP_CHKPT_LOG_COPY = 0,
    BACKUP_CHKPT_LOG_NOCOPY,
    BACKUP_CHKPT_LOG_RECOVER,
    BACKUP_CHKPT_LOG_AUTO,
    BACKUP_CHKPT_LOG_DEFAULT
} a_chkpt_log_type;
```

参数

值	说明
BACKUP_CHKPT_LOG_COPY	用于生成 WITH CHECKPOINT LOG COPY 子句。
BACKUP_CHKPT_LOG_NOCOPY	用于生成 WITH CHECKPOINT LOG NOCOPY 子句。
BACKUP_CHKPT_LOG_RECOVER	用于生成 WITH CHECKPOINT LOG RECOVER 子句。

值	说明
BACKUP_CHKPT_LOG_AUTO	用于生成 WITH CHECKPOINT LOG AUTO 子句。
BACKUP_CHKPT_LOG_DEFAULT	用于忽略 WITH CHECKPOINT 子句。

另请参见

- “a_backup_db 结构”一节第 925 页

a_db_version 枚举

在“a_db_version_info 结构”一节第 933 页中用于指示最初创建数据库的 SQL Anywhere 版本。

语法

```
enum {
          VERSION_UNKNOWN,
          VERSION_PRE_10,
          VERSION_10,
          VERSION_11
    };
```

参数

值	说明
VERSION_UNKNOWN	无法确定创建数据库的 SQL Anywhere 的版本。
VERSION_PRE_10	数据库是使用 SQL Anywhere 10 之前版本创建的。
VERSION_10	数据库是使用 SQL Anywhere 10 创建的。
VERSION_11	数据库是使用 SQL Anywhere 11 创建的。

另请参见

- “DBCreatedVersion 函数”一节第 916 页
- “a_db_version_info 结构”一节第 933 页

数据库大小单位枚举

在“a_create_db 结构”一节第 929 页中用于指定 db_size_unit 的值。

语法

```
enum {
    DBSP_UNIT_NONE,
    DBSP_UNIT_PAGES,
    DBSP_UNIT_BYTES,
```

```
    DBSP_UNIT_KILOBYTES,
    DBSP_UNIT_MEGABYTES,
    DBSP_UNIT_GIGABYTES,
    DBSP_UNIT_TERABYTES
};
```

参数

值	说明
DBSP_UNIT_NONE	未指定单位。
DBSP_UNIT_PAGES	将大小指定为页面数。
DBSP_UNIT_BYTES	将大小指定为字节数。
DBSP_UNIT_KILOBYTES	将大小指定为千字节数。
DBSP_UNIT_MEGABYTES	将大小指定为兆字节数。
DBSP_UNIT_GIGAYTES	将大小指定为千兆字节数。
DBSP_UNIT_TERABYTES	将大小指定为千吉字节数。

另请参见

- “a_create_db 结构”一节第 929 页

dbtran_userlist_type 枚举

用户列表的类型，由“a_translate_log 结构”一节第 952 页使用。

语法

```
typedef enum dbtran_userlist_type {
    DBTRAN_INCLUDE_ALL,
    DBTRAN_INCLUDE_SOME,
    DBTRAN_EXCLUDE_SOME
} dbtran_userlist_type;
```

参数

值	说明
DBTRAN_INCLUDE_ALL	包括所有用户的操作。
DBTRAN_INCLUDE_SOME	只包括所提供用户列表中用户的操作。
DBTRAN_EXCLUDE_SOME	排除所提供用户列表中用户进行的操作。

另请参见

- “a_translate_log 结构”一节第 952 页

dbunload 类型枚举

正执行的卸载的类型，由“an_unload_db 结构”一节第 957 页使用。

语法

```
enum {
    UNLOAD_ALL,
    UNLOAD_DATA_ONLY,
    UNLOAD_NO_DATA,
    UNLOAD_NO_DATA_FULL_SCRIPT
};
```

参数

值	说明
UNLOAD_ALL	卸载数据和模式。
UNLOAD_DATA_ONLY	卸载数据，不卸载模式。相当于 dbunload -d 选项。
UNLOAD_NO_DATA	无数据。只卸载模式。相当于 dbunload -n 选项。
UNLOAD_NO_DATA_FULL_SCRIPT	无数据。在重装脚本中包括 LOAD/INPUT 语句。相当于 dbunload -nl 选项。

另请参见

- “an_unload_db 结构”一节第 957 页

a_validate_type 枚举

正执行的校验的类型，由“a_validate_db 结构”一节第 962 页使用。

语法

```
typedef enum {
    VALIDATE_NORMAL = 0,
    VALIDATE_DATA,
    VALIDATE_INDEX,
    VALIDATE_EXPRESS,
    VALIDATE_FULL,
    VALIDATE_CHECKSUM,
    VALIDATE_DATABASE,
    VALIDATE_COMPLETE
} a_validate_type;
```


参数

值	说明
VALIDATE_NORMAL	只用缺省检查进行校验。
VALIDATE_DATA	（已过时）
VALIDATE_INDEX	（已过时）
VALIDATE_EXPRESS	使用快速检查进行校验。相当于 dbvalid -fx 选项。
VALIDATE_FULL	（已过时）
VALIDATE_CHECKSUM	校验数据库校验和。相当于 dbvalid -s 选项。
VALIDATE_DATABASE	校验数据库。相当于 dbvalid -d 选项。
VALIDATE_COMPLETE	执行所有可能的校验活动。

另请参见

- “a_validate_db 结构”一节第 962 页
- “校验实用程序 (dbvalid)”一节 《SQL Anywhere 服务器 - 数据库管理》
- “VALIDATE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

详细枚举

指定输出的量。

语法

```
enum {
    VB_QUIET,
    VB_NORMAL,
    VB_VERBOSE
};
```

参数

值	说明
VB_QUIET	无输出。
VB_NORMAL	一般输出量。
VB_VERBOSE	详细输出，有助于进行调试。

另请参见

- “a_create_db 结构”一节第 929 页
- “an_unload_db 结构”一节第 957 页

退出代码

目录

软件组件的退出代码

所有的数据库工具都是作为 DLL 中的入口点提供的。这些入口点使用以下退出代码。SQL Anywhere 实用程序（dbbackup、dbspawn、dbeng11 等）也使用这些退出代码。

代码	状态	解释
0	EXIT_OKAY	成功
1	EXIT_FAIL	常规失败
2	EXIT_BAD_DATA	文件格式无效
3	EXIT_FILE_ERROR	文件未找到，无法打开
4	EXIT_OUT_OF_MEMORY	内存不足
5	EXIT_BREAK	被用户终止
6	EXIT_COMMUNICATIONS_FAIL	通信失败
7	EXIT_MISSING_DATABASE	缺少一个必需的数据库名
8	EXIT_PROTOCOL_MISMATCH	客户端/服务器协议不匹配
9	EXIT_UNABLE_TO_CONNECT	无法连接到数据库服务器
10	EXIT_ENGINE_NOT_RUNNING	数据库服务器未运行
11	EXIT_SERVER_NOT_FOUND	未找到数据库服务器
12	EXIT_BAD_ENCRYPT_KEY	加密密钥遗失或错误
13	EXIT_DB_VER_NEWER	必须升级服务器才能运行数据库
14	EXIT_FILE_INVALID_DB	文件不是数据库
15	EXIT_LOG_FILE_ERROR	日志文件遗失或其它错误
16	EXIT_FILE_IN_USE	文件正在使用
17	EXIT_FATAL_ERROR	发生致命错误或断言
255	EXIT_USAGE	命令行上的参数无效

这些退出代码包含在 *install-dir\sdk\include\sqldef.h* 文件中。

部署 SQL Anywhere

本节介绍 SQL Anywhere 的部署策略。

部署数据库和应用程序

目录

部署简介

当您完成了数据库应用程序时，必须将应用程序部署到最终用户。根据您的应用程序使用 SQL Anywhere 的方式（作为嵌入式数据库、以客户端/服务器形式等等），您可能必须将 SQL Anywhere 软件的组件与您的应用程序一起部署。您可能还必须部署配置信息（如数据源名称），以便使您的应用程序能够与 SQL Anywhere 进行通信。

检查许可协议
重新分发文件受到 Sybase 许可协议的制约。本文中的任何声明都不能替代许可协议中的任何内容。在考虑部署之前，请检查许可协议。

本章介绍下列部署步骤：

- 基于所选应用程序平台和体系结构确定必需的文件。
- 配置客户端应用程序。

本章的大部分内容介绍各个文件以及需要放置它们的位置。但是，建议通过使用 [**部署向导**] 或使用静默安装来部署 SQL Anywhere 组件。有关信息，请参见“使用 [部署向导]”一节第 981 页和“使用静默安装进行部署”一节第 985 页。

部署类型

需要部署的文件取决于所选的部署类型。下面是一些可用的部署模型：

- **客户端部署** 您只能部署面向最终用户的 SQL Anywhere 客户端部分，以便他们能够连接到集中放置的网络数据库服务器。
- **网络服务器部署** 您可以将网络服务器部署到多个办公室，然后将客户端部署到这些办公室中的每个用户。
- **嵌入式数据库部署** 您可以部署与个人数据库服务器一起运行的应用程序。在这种情况下，需要在最终用户的计算机上同时安装客户端和个人服务器。
- **SQL Remote 部署** SQL Remote 应用程序的部署是对嵌入式数据库部署模型的扩展。
- **MobiLink 部署** 有关部署 MobiLink 服务器的信息，请参见“部署 MobiLink 应用程序” 《MobiLink - 服务器管理》。
- **管理工具部署** 您可以部署 Interactive SQL、Sybase Central 和其它管理工具。

文件的分发方式

可通过两种方法来部署 SQL Anywhere：

- **使用 SQL Anywhere 安装程序** 您可以为最终用户提供安装程序。通过选择正确的选项，保证每个最终用户收到他们所需的文件。

这种解决方案对很多部署情况来说都是最简单的。在这种情况下，您还必须向最终用户提供一种连接到数据库服务器的方法（如 ODBC 数据源）。

有关详细信息，请参见“使用 [部署向导]”一节第 981 页或“使用静默安装进行部署”一节第 985 页。

- **开发自己的安装程序** 可能有许多原因促使您开发自己的包括 SQL Anywhere 文件的安装程序。这是一个更复杂的选择，本章中的大部分内容满足了正在自行开发安装程序的用户的需要。

  如果已为客户端应用程序体系结构所需的服务器类型和操作系统安装了 SQL Anywhere，则可在 SQL Anywhere 安装目录中具有相应名称的子目录中找到必需的文件。例如，安装目录的 *bin32* 子目录包含了在 32 位 Windows 操作系统中运行服务器所需的文件。

无论选择哪个选项，都不得违反许可协议中的条款。

了解安装目录和文件名

为了使部署的应用程序正确工作，数据库服务器和客户端应用程序都必须能够找到它们所需的文件。部署文件应当以与 SQL Anywhere 安装相同的方式来确定彼此之间的相对位置。

实际上，这意味着 Windows 上的大多数文件都属于同一目录。例如，在 Windows 上，客户端和数据库服务器所需的文件都安装在同一目录中，即 SQL Anywhere 安装目录的 *bin32* 子目录。

有关软件查找文件的位置的完整说明，请参见“SQL Anywhere 如何定位文件”一节 《SQL Anywhere 服务器 - 数据库管理》。

Linux、Unix 和 Mac OS X 部署问题

Unix 部署在下列几个方面不同于 Windows 部署：

- **目录结构** 对于 Linux、Unix 和 Mac OS X 安装，目录结构如下所示：

目录	内容
/opt/sqlanywhere11/bin32 和 */opt/sqlanywhere11/bin64*	可执行文件、许可文件
/opt/sqlanywhere11/lib32 和 */opt/sqlanywhere11/lib64*	共享对象和库
/opt/sqlanywhere11/res	字符串文件

  在 AIX 上，缺省根目录是 */usr/lpp/sqlanywhere11* 而不是 */opt/sqlanywhere11*。

  在 Mac OS X 上，缺省根目录是 */Applications/SQLAnywhere11/System* 而不是 */opt/sqlanywhere11*。

- **文件后缀** 在本章的表中，列出的共享对象带有后缀 *.so* 或 *.so.1*。随着更新版本的发布，版本号可大于 1。为简单起见，通常不会列出版本号。

  对于 AIX，后缀不包含版本号，因此它仅为 *.so*。

- **符号链接** 每个共享对象都安装为具有附加后缀 *.1*（一）的同名文件的符号链接 (symlink)。例如，*libdblib11.so* 是相同目录中文件 *libdblib11.so.1* 的符号连接。

  随着更新版本的发布，版本后缀 *.1* 可能更高，因此必须重定向符号链接。

  在 Mac OS X 上，应为任何一个想要从 Java 客户端应用程序直接装载的 dylib 创建 jnilib 符号链接。

- **线程和非线程应用程序** 大多数共享对象都以两种形式提供，其中之一是文件后缀前带有附加字符 *_r*。例如，除了 *libdblib11.so.1*，还有一个名为 *libdblib11_r.so.1* 的文件。在这种情况下，线程应用程序必须链接到名称中带有 *_r* 后缀的共享对象，而非线程应用程序必须链接到名称中不带 *_r* 后缀的共享对象。有时候，共享对象有第三种形式，即文件后缀之前带有 *_n*。此版本的共享对象用于非线程应用程序。

- **字符集转换** 如果您要使用数据库服务器字符集转换，则需要包括下列文件：
  - *libdbicu11.so.1*
  - *libdbicu11_r.so.1*
  - *libdbicudt11.so.1*
  - *sqlany.cvf*
- **环境变量** 在 Linux 和 Unix 上，必须设置环境变量，系统才能找到 SQL Anywhere 应用程序和库。建议使用与您的 shell 相适合的文件，即 *sa_config.sh* 或 *sa_config.csh* 文件（位于目录 */opt/sqlanywhere11/bin32* 和 */opt/sqlanywhere11/bin64* 下），来作为设置所需环境变量的模板。由这些文件设置的一些环境变量包括 PATH、LD_LIBRARY_PATH 和 SQLANYSH11。

有关 SQL Anywhere 如何查找文件的说明，请参见“SQL Anywhere 如何定位文件”一节 《SQL Anywhere 服务器 - 数据库管理》。

文件命名约定

SQL Anywhere 使用一致的文件命名约定来帮助标识和分组系统组件。

这些约定包括：

- **版本号** SQL Anywhere 的版本号由主要服务器组件（可执行文件、动态链接库、共享对象、许可文件等等）的文件名来表示。

  例如，文件 *dbeng11.exe* 是适用于 Windows 的版本 11 可执行文件。
- **Language** 语言资源库中使用的语言由它的文件名中的双字母代码表示。版本号前面的两个字符表示该库中使用的语言。例如，*dblgen11.dll* 是英语语言的消息资源库。这些双字母代码由 ISO 标准 639-1 指定。

  有关语言标签的详细信息，请参见“语言选择实用程序 (dblang)”一节 《SQL Anywhere 服务器 - 数据库管理》。

  有关 SQL Anywhere 中可用语言的列表，请参见“SQL Anywhere 的本地化版本”一节 《SQL Anywhere 服务器 - 数据库管理》。

标识其它文件类型

下表按照 SQL Anywhere 文件的文件扩展名来标识它们的平台和功能。SQL Anywhere 尽可能遵循标准的文件扩展名约定。

文件扩展名	平台	文件类型
.bat、*.cmd*	Windows	批处理命令文件
.chm、*.chw*	Windows	帮助系统文件
.dll	Windows	动态链接库
.exe	Windows	可执行文件

文件扩展名	平台	文件类型
.ini	全部	初始化文件
.lic	全部	许可文件
.lib	因开发工具而异	用于创建嵌入式 SQL 可执行文件的静态运行时库
.res	Linux、Unix、Mac OS X	面向非 Windows 环境的语言资源文件
.so	Linux、Unix	共享对象或者共享库文件。等效于 Windows DLL
.bundle、*.dylib*	Mac OS X	共享对象文件。等效于 Windows DLL

数据库文件名

SQL Anywhere 数据库由下列两个元素组成：

- **数据库文件** 此文件用于存储有组织格式的信息。缺省情况下，此文件使用 *.db* 文件扩展名。还可能存在其它 dbspace 文件。这些文件可以带有任何文件扩展名，也可以没有文件扩展名。
- **事务日志文件** 此文件用于记录对存储在数据库文件中的数据进行的所有更改。缺省情况下，此文件使用 *.log* 文件扩展名，如果此文件不存在但指定了要使用日志文件，就会由 SQL Anywhere 生成此文件。事务日志镜像的缺省扩展名为 *.mlg*。

上述文件由 SQL Anywhere 关系数据库管理系统更新、维护和管理。

使用 [部署向导]

SQL Anywhere [**部署向导**] 是创建用于 Windows 的 SQL Anywhere 32 位部署的首选工具。[**部署向导**] 可以创建包括以下部分或全部组件的安装程序文件：

- 客户端接口（例如 ODBC）
- SQL Anywhere 服务器（包括远程数据访问、数据库工具和加密）
- UltraLite 关系数据库
- MobiLink 服务器、客户端和加密
- QAnywhere 消息传递
- 管理工具（例如 Interactive SQL 和 Sybase Central）

[**部署向导**] 不支持创建 64 位软件组件的部署。

可以使用 [**部署向导**] 来创建 Microsoft Windows 安装程序包文件或 Microsoft Windows 安装程序合并模块文件：

- **Microsoft Windows 安装程序包文件** 包含安装应用程序所需的说明和数据的存储文件。安装程序包文件带有扩展名 *.msi*。
- **Microsoft Windows 安装程序合并模块文件** 简化类型的 Microsoft 安装程序包文件，其中包括安装共享组件所需的所有文件、资源、注册表条目和安装逻辑。合并模块带有扩展名 *.msm*。

  合并模块不能单独安装，因为它缺少某些存在于安装程序包文件中的重要数据库表。合并模块还包含其它只有合并模块才有的表。要安装由合并模块通过应用程序传送的信息，必须首先将模块合并到应用程序的安装程序包 (*.msi*) 文件中。合并模块包括以下组成部分：

  - 合并模块数据库，其中包含由合并模块传送的安装属性和安装逻辑。
  - 介绍模块的合并模块摘要信息流。
  - *MergeModule.CAB* cabinet 文件，该文件做为合并模块内部的流存储。此 cabinet 文件包含合并模块传送的组件所需的所有文件。每个由合并模块传送的文件都必须存储在 cabinet 文件中，该文件作为流嵌入到合并模块的结构存储中。在标准合并模块中，此 cabinet 文件的名称始终为：*MergeModule.CAB*。

注意
重新分发文件受到许可协议的制约。必须确认您拥有重新分发 SQL Anywhere 文件的适当许可。在继续之前，请检查许可协议。

♦ 创建部署文件

1. 启动 [**部署向导**]：
   - 从 [**开始**] 菜单中，选择 [**程序**] » [**SQL Anywhere 11**] » [**部署 SQL Anywhere For Windows**]。

     或者
   - 从 SQL Anywhere 安装目录的 *Deployment* 子目录，运行 *DeploymentWizard.exe*。

2. 按照向导中的说明进行操作。

[**部署向导**] 允许选择在 SQL Anywhere 中包含的组件的子集。每个组件均依赖于其它组件，因此由向导选择的文件可能包括来自其它类别的文件。

在 [**选择功能**] 中，可用类别有 [**数据库**]、[**同步**] 和 [**管理工具**]。

数据库

用于选择或取消选择它的所有子类别。

- **SQL Anywhere（32 位）** 用于选择或取消选择它的所有子类别。

  可以使用以下子类别：

  - **客户端接口** 用于选择或取消选择它的所有子类别。
    - **ODBC** SQL Anywhere ODBC 驱动程序。
    - **嵌入式 SQL** SQL Anywhere 嵌入式 SQL 库。
    - **OLEDB** SQL Anywhere OLE DB 提供程序。
    - **ADO.NET** SQL Anywhere .NET 提供程序。
    - **JDBC** SQL Anywhere JDBC 驱动程序。
    - **客户端工具** SQL Anywhere 客户端库（如 dblib11、dbtool11）及客户端实用程序（如 dblocate、dbping、dbisqlc 和 dbdsn）。
    - **客户端资源** SQL Anywhere 语言资源文件（如 dblgen11、dblgde11 和 dblges11）及 dblang 语言选择工具。
  - **SQL Anywhere 服务器** 用于选择或取消选择它的所有子类别。
    - **个人服务器** SQL Anywhere 个人服务器和许可文件。
    - **网络服务器** SQL Anywhere 网络服务器和许可文件。
    - **服务器工具** SQL Anywhere 服务器实用程序，如 dbbackup、dberase、dbinit、dblog、dbsvc、dbunload 等。
    - **卸载支持** 支持卸载版本 9 和更早版本的数据库。
- **UltraLite** 用于选择或取消选择它的所有子类别。

  可以使用以下子类别：

  - **UltraLite 引擎** UltraLite 引擎、实用程序和库，例如 uleng11、ulcond11、ulcreate、ulerase、ullgen11、ullgde11、ulrt11 和 ulunload。

同步

用于选择或取消选择它的所有子类别。

- **MobiLink** 用于选择或取消选择它的所有子类别。
  - **MobiLink 客户端** MobiLink 客户端工具和库（如 dblsn、dbmlsync、mlasinst、dbmlsynccli11）以及 MobiLink .NET 客户端提供程序。

- **MobiLink 服务器** MobiLink 服务器、工具和库（如服务器、ODBC 驱动程序、JDBC 驱动程序）以及 MobiLink .NET 提供程序。

- **QAnywhere** QAnywhere 应用程序到应用程序消息传递工具。
- **SQL Remote** SQL Remote 工具和库（包括 dbremote、dbxtract）以及消息传送库（如 dbsmtp11）。

管理工具

用于选择或取消选择它的所有子类别。

- **Sybase Central** Sybase Central 数据库管理器和插件。用于选择或取消选择它的所有子类别。
  - **SQL Anywhere 插件** SQL Anywhere 插件。
  - **MobiLink 插件** MobiLink 插件。
  - **UltraLite 插件** UltraLite 插件。
  - **QAnywhere 插件** QAnywhere 插件。
- **ISQL** Interactive SQL 工具。
- **DBConsole** 数据库服务器连接的管理和监控工具。

如果要确定在每个可选择的组件中包括了哪些文件，请在选择所有组件时创建 MSI 安装程序映像。系统会创建一个日志文件，其中详细记录了每个组件中所包括的文件。可使用文本编辑器来检查此文本文件。您将看到 "Feature:SERVER32_TOOLS" 和 "Feature:CLIENT32_TOOLS" 之类的标题。这些标题与 [部署向导] 组件密切对应。这会让您了解到每个组中包括哪些文件。

♦ **安装部署文件**

- 使用 Microsoft Windows 安装程序来安装部署文件。下面是示例命令：

```
msiexec /package sqlany11.msi
```

可使用以下形式的命令执行静默安装：

```
msiexec /qn /package sqlany11.msi SQLANYDIR=c:\sa11
```

- **/package <package-name>** 此参数通知 Microsoft Windows 安装程序安装指定的包（此示例中为 *sqlany11.msi*）。
- **/qn** 此参数通知 Microsoft Windows 安装程序进行无用户交互的后台操作。
- **SQLANYDIR** 此参数的值是软件要安装到的路径。

♦ **卸载部署**

- 也可以执行静默卸载。下面是可完成此操作的命令行示例。

```
msiexec /uninstall sqlany11.msi
```

也可以指定产品代码。

```
msiexec.exe /qn /uninstall {19972A31-72EF-126F-31C7-5CF249B8593F}
```

- **/qn** 此参数通知 Microsoft Windows 安装程序进行无用户交互的后台操作。
- **/uninstall <package-name> | <product-code>** 此参数通知 Microsoft Windows 安装程序卸载与指定 MSI 文件或产品代码相关联的产品。

有关如何进行静默安装的详细信息，请参见“使用静默安装进行部署”一节第 985 页。

使用静默安装进行部署

运行静默安装时无需用户输入且不指示用户安装正在进行。在 Windows 操作系统上，您可以按照 SQL Anywhere 静默安装的方式，从自己的安装程序中调用 SQL Anywhere 安装程序。

SQL Anywhere 安装程序 *setup.exe* 的选项如下：

- **/L:language_id** 语言标识符是表示安装语言的地区号码。例如，地区 ID 1033 表示美国英语，地区 ID 1031 表示德语，地区 ID 1036 表示法语，地区 ID 1041 表示日语，地区 ID 2052 表示简体中文。
- **/S** 此选项可隐藏初始化对话框。此选项应与 **/V** 配合使用。
- **/V** 指定 MSIEXEC（Microsoft Windows 安装程序工具）的参数。

以下命令行示例假定安装映像目录位于驱动器 *d:* 中磁盘的 *software\SQLAnywhere* 目录下。

```
d:\software\sqlanywhere\setup.exe /l:1033 /s "/v:/qn
    REGKEY=PEPEV-E96QE-A4000-00000-00000 INSTALLDIR=c:\sa11 DIR_SAMPLES=c:
\sa11\Samples"
```

注意
上述命令中的 setup.exe 与 SQLANY32.msi 和 SQLANY64.msi 文件位于同一个目录中。这两个文件的父目录中的 setup.exe 不支持静默安装。

可在命令行上指定以下选项。

- **REGKEY** 此参数值必须为有效的软件安装密钥。
- **INSTALLDIR** 此参数的值是要将软件安装到的路径。
- **DIR_SAMPLES** 此参数的值是示例程序所安装到的路径。
- **USERNAME** 此参数的值是要为此次安装记录的用户名（例如 USERNAME=\"John Smith\"）。
- **COMPANYNAME** 此参数的值是要为此次安装记录的公司名（例如 COMPANYNAME=\"Smith Holdings\"）。

以下示例包括所有选项：

```
d:\software\sqlanywhere\setup.exe /l:1033 /s "/v:/qn
    REGKEY=PEPEV-E96QE-A4000-00000-00000
    INSTALLDIR=c:\sa11
    DIR_SAMPLES=c:\sa11\Samples
    USERNAME=\"John Smith\"
    COMPANYNAME=\"Smith Holdings\""
```

尽管以上文本由于长度原因而分为多行显示，但它是作为单一文本行而指定的。注意使用反斜线字符来转义内部的引号。

也可以采用静默安装方式来安装文档。与此相对应的 *setup.exe* 位于 *d:\software\Documentation* 中。用来安装文档的命令行示例为：

```
d:\software\documentation\setup.exe /l:1033 /s "/v:/qn"
```

要生成 MSI 日志，请将以下内容添加到命令行的 /v: 之后。

```
/l*v! logfile
```

在上述示例中，logfile 是日志文件的完整路径和文件名。该路径必须已经存在。请注意，此开关将生成一个极其详细的日志，因此会明显延长执行安装所需的时间。有关如何减少日志文件输出的详细信息，请参见 MSI 文档 (http://msdn.microsoft.com/en-us/library/aa367988.aspx)。

除静默安装外，还可以执行静默卸载。下面是可完成此操作的命令行示例。

```
msiexec.exe /qn /uninstall {ECE263B0-6C8B-404C-B4AC-8FAB1C87AB4A}
```

在上面的示例中，直接调用了 Microsoft Windows 安装程序工具。

- **/qn** 此参数通知 Microsoft Windows 安装程序进行无用户交互的后台操作。
- **/uninstall <product-code>** 此参数通知 Microsoft Windows 安装程序卸载与指定产品代码相关联的产品。上面所示的代码适用于 SQL Anywhere 软件。

SQL Anywhere 的产品代码为：

- **{ECE263B0-6C8B-404C-B4AC-8FAB1C87AB4A}** SQL Anywhere 软件
- **{10964A7D-722B-4FE5-A16D-4977DCECEE95}** SQL Anywhere 文档

上面介绍的静默安装没有提到如何选择安装组件中的一部分。在 [**部署向导**] 中有关于这一主题的详细叙述。有关组件选择的信息，请参见“使用 [部署向导]” 一节第 981 页。

部署客户端应用程序

要部署针对网络数据库服务器运行的客户端应用程序，您必须为每位最终用户提供下列项目：

- **客户端应用程序** 应用程序软件本身与数据库软件无关，因此在此不予介绍。
- **数据库接口文件** 客户端应用程序需要它所使用的数据库接口文件（.NET、ADO、OLE DB、ODBC、JDBC、嵌入式 SQL 或 Open Client）。
- **连接信息** 每个客户端应用程序都需要数据库连接信息。

所需的接口文件和连接信息因应用程序正在使用的接口而异。以下各节分别介绍了每个接口。

部署客户端最简单的方法是使用 [**部署向导**]。有关详细信息，请参见“使用 [部署向导]”一节第 981 页。

部署 .NET 客户端

部署 .NET 程序集最简单的方法是使用 [**部署向导**]。有关详细信息，请参见“使用 [部署向导]”一节第 981 页。

如果最终用户将来要开发 .NET 应用程序，则应考虑将 SQL Anywhere 工具集成到 Microsoft Visual Studio 中。在客户端计算机上，必须执行以下操作。

- 确保 Visual Studio 未运行。
- 运行 *install-dir\Assembly\v2\SetupVSPackage.exe /install*。

如果您希望创建自己的安装，本节介绍用于部署到最终用户的文件。

每个 .NET 客户端计算机都必须具有下列项目：

- **可工作的 .NET 2.0（或更高版本）安装** Microsoft Corporation 提供了 Microsoft .NET 程序集及其重新分发指导。此处不再详细介绍。
- **SQL Anywhere .NET 数据提供程序** 下表显示了运行 SQL Anywhere .NET 提供程序所需的文件。这些文件应放在单一目录中。

  SQL Anywhere 安装将用于 .NET Framework 的 Windows 程序集置于 SQL Anywhere 安装目录的 *Assembly\v2* 子目录中。其它文件则置于 SQL Anywhere 安装目录的操作系统子目录中（如 *bin32* 或 *bin64*）。

  SQL Anywhere 安装将用于 .NET Compact Framework 的 Windows Mobile 程序集置于 *ce\Assembly\v2* 下。其它文件则置于 SQL Anywhere 安装目录的 Windows Mobile 子目录中（如 *ce\arm.50*）。

说明	Windows	Windows Mobile
.NET 驱动程序文件	*iAnywhere.Data.SQLAnywhere.dll*	*iAnywhere.Data.SQLAnywhere.dll*

说明	Windows	Windows Mobile
.NET 全局程序集高速缓存	N/A	*iAnywhere.Data.SQLAnywhere.gac*
语言资源库	*dblg[xx]11.dll*	*dblg[xx]11.dll*
[连接] 窗口	*dbcon11.dll*	N/A

上表显示了带有标志 **[xx]** 的文件。这表示消息文件有多个，每个消息文件支持不同的语言。如果要安装对不同语言的支持，必须添加这些语言的资源文件。

有关部署 SQL Anywhere .NET 提供程序的详细信息，请参见“部署 SQL Anywhere .NET 数据提供程序”一节第 132 页。

部署 OLE DB 和 ADO 客户端

部署 OLE DB 客户端库最简单的方法是使用 [**部署向导**]。有关详细信息，请参见“使用 [部署向导]”一节第 981 页。

如果您希望创建自己的安装，本节介绍了用于部署到最终用户的文件。

每个 OLE DB 客户端计算机都必须具有下列项目：

- **可工作的 OLE DB 安装** Microsoft Corporation 提供了 OLE DB 文件及其重新分发指导。此处不再详细介绍。
- **SQL Anywhere OLE DB 提供程序** 下表显示了运行 SQL Anywhere OLE DB 提供程序所需的文件。这些文件应放在单一目录中。SQL Anywhere 安装将它们全部置于 SQL Anywhere 安装目录的操作系统子目录中（如 *bin32* 或 *bin64*）。对于 Windows，有两个提供程序 DLL。第二个 DLL (*dboledba11*) 是用于提供模式支持的辅助 DLL。

说明	Windows
OLE DB 驱动程序文件	*dboledb11.dll*
OLE DB 驱动程序文件	*dboledba11.dll*
语言资源库	*dblg[xx]11.dll*
[连接] 窗口	*dbcon11.dll*
已提升的操作代理	*dbelevate11.exe*（仅限 Vista）

上表显示了一个带有标志 **[xx]** 的文件。这表示消息文件有多个，每个消息文件支持不同的语言。如果要安装对不同语言的支持，必须添加这些语言的资源文件。

OLE DB 提供程序需要许多注册表条目。通过使用 regsvr32 实用程序自行注册 *dboledb11.dll* 和 *dboledba11.dll* DLL，可创建这些注册表条目。

注意，对于 Windows Vista 或更高版本的 Windows，必须包含 SQL Anywhere 已提升的操作代理以支持注册或注销 DLL 时所需的权限提升。此文件仅为 OLE DB 提供程序安装或卸载过程所必需。

对于 Windows 客户端，建议使用 Microsoft MDAC 2.7 或更高版本。

自定义 OLE DB 提供程序

安装 OLE DB 提供程序时，必须修改 Windows 注册表。通常，使用内置于 OLE DB 提供程序的自行注册功能进行修改。例如，可以使用 Windows regsvr32 工具来进行此操作。注册表条目的标准集由提供程序创建。

在典型的连接字符串中，提供程序的属性是组成部分之一。要指示将使用的 SQL Anywhere OLE DB 提供程序，需要指定提供程序的名称。下面是 Visual Basic 示例：

```
connectString = "Provider=SAOLEDB;DSN=SQL Anywhere 11 Demo"
```

对于 ADO 和/或 OLE DB，有许多通过名称引用提供程序的其它方法。下面是一个 C++ 示例，其中不仅指定了提供程序名称而且还指定了要使用的版本。

```
hr = db.Open(_T("SAOLEDB.11"), &dbinit);
```

提供程序名称在注册表中查找。如果查看您的计算机系统上的注册表，您会在 HKEY_CLASSES_ROOT 中会找到 SAOLEDB 条目。

```
[HKEY_CLASSES_ROOT\SAOLEDB]
@="SQL Anywhere OLE DB Provider"
```

此条目有两个子项，分别包含此提供程序的类标识符 (ClsId) 以及当前版本 (CurVer)。下面是一个示例：

```
[HKEY_CLASSES_ROOT\SAOLEDB\Clsid]
@="{41dfe9f3-db91-11d2-8c43-006008d26a6f}"

[HKEY_CLASSES_ROOT\SAOLEDB\CurVer]
@="SAOLEDB.11"
```

还有若干更类似的条目。它们用于标识 OLE DB 提供程序的特定实例。如果在注册表的 HKEY_CLASSES_ROOT\CLSID 下查找 ClsId 并且查看子项，您将看到其中一个条目标识了提供程序 DLL 的位置。

```
[HKEY_CLASSES_ROOT\CLSID\
{41dfe9f3-db91-11d2-8c43-006008d26a6f}\
InprocServer32]

@="c:\\sa11\\bin64\\dboledb11.dll"
"ThreadingModel"="Both"
```

此处问题是结构非常单一。如果将 SQL Anywhere 软件从您的系统卸载，OLE DB 提供程序注册表条目将从注册表删除，随后提供程序 DLL 将从硬盘驱动器删除。任何依赖提供程序的应用程序将不再工作。

同样，如果来自不同供应商的应用程序都使用相同的 OLE DB 提供程序，则每次安装相同的提供程序时都将覆盖公用注册表设置。您希望应用程序使用的提供程序版本将由另一较新（或较旧！）的提供程序版本代替。

这种情况所引发的不稳定性无疑是不希望发生的。为解决此问题，可以自定义 SQL Anywhere OLE DB 提供程序。

在下面的练习中，您可以生成一组唯一的 GUID，选择唯一的提供程序名称和唯一的 DLL 名称。这三项操作有助于创建可与您的应用程序一起部署的唯一的 OLE DB 提供程序。

下面是创建自定义版本 OLE DB 提供程序所涉及的步骤。

♦ 自定义 OLE DB 提供程序

1. 制作下面所示的示例注册文件的副本。由于注册文件过长，因此将在这些步骤之后列出。文件名应当带有 *.reg* 后缀。注册表值的名称区分大小写。
2. 使用 Microsoft Visual Studio uuidgen 实用程序创建 4 个有顺序的 UUID (GUID)。

   ```
   uuidgen -n4 -s -x >oledbguids.txt
   ```

3. 按以下顺序指派 4 个 UUID 或 GUID：
   a. 提供程序类 ID（下面的 GUID1）。
   b. 枚举类 ID（下面的 GUID2）。
   c. ErrorLookup 类 ID（下面的 GUID3）。
   d. 提供程序帮助类 ID（下面的 GUID4）。此最后一个 GUID 不在 Windows Mobile 部署中使用。

   上述四项之间存在先后顺序（这就是在 uuidgen 命令行中 -x 所起的作用），这一点很重要。每个 GUID 都应出现与以下类似的内容。

名称	GUID
GUID1	41dfe9f3-db92-11d2-8c43-006008d26a6f
GUID2	41dfe9f4-db92-11d2-8c43-006008d26a6f
GUID3	41dfe9f5-db92-11d2-8c43-006008d26a6f
GUID4	41dfe9f6-db92-11d2-8c43-006008d26a6f

   请注意，递增的只是 GUID 的第一部分（例如，41dfe9f3）。
4. 使用编辑器的 [查找/替换] 功能将文本中所有 GUID1、GUID2、GUID3 和 GUID4 更改为相应的 GUID（例如，如果 41dfe9f3-db92-11d2-8c43-006008d26a6f 是由 uuidgen 生成的 GUID，则 GUID1 将用其替换）。
5. 确定提供程序名称。这是可以在您的应用程序的连接字符串中使用的名称（例如，Provider=SQLAny）。请不要使用以下任何名称。这些名称由 SQL Anywhere 使用。

版本 10 或更高版本	版本 9 或更早版本
SAOLEDB	ASAProv

版本 10 或更高版本	版本 9 或更早版本
SAErrorLookup	ASAErrorLookup
SAEnum	ASAEnum
SAOLEDBA	ASAProvA

6. 使用编辑器的 [查找/替换] 功能将所有出现的字符串 SQLAny 更改为您所选择的提供程序名称。其中包括所有那些 SQLAny 可能为较长字符串（例如，SQLAnyEnum）的子串的位置。

   假定您选择 Acme 作为提供程序名称。将出现在 HKEY_CLASSES_ROOT 注册表配置单元中的名称以及供比较的 SQL Anywhere 名称如下表所示。

SQL Anywhere 提供程序	您自定义的提供程序
SAOLEDB	Acme
SAErrorLookup	AcmeErrorLookup
SAEnum	AcmeEnum
SAOLEDBA	AcmeA

7. 以不同名称制作 SQL Anywhere 提供程序 DLL（*dboledb11.dll* 和 *dboledba11.dll*）的副本。请注意，没有用于 Windows Mobile 的 *dboledba11.dll*。

   ```
   copy dboledb11.dll myoledb11.dll
   copy dboledba11.dll myoledba11.dll
   ```

   特殊注册表项将由脚本基于您所选择的 DLL 名称创建。此名称应不同于标准的 DLL 名称（例如 *dboledb11.dll* 或 *dboledba11.dll*），这一点非常重要。如果您将提供程序 DLL 命名为 *myoledb11*，则提供程序将在 HKEY_CLASSES_ROOT 中查找具有相同名称的注册表条目。上述情况同样适用于提供程序模式辅助 DLL。如果您将 DLL 命名为 *myoledba11*，则提供程序将在 HKEY_CLASSES_ROOT 中查找具有相同名称的注册表条目。您选择的名称是唯一的并且不可能被任何其他人选择，这一点非常重要。下面是一些示例。

所选的 DLL 名称	相应的 HKEY_CLASSES_ROOT\名称
myoledb11.dll	HKEY_CLASSES_ROOT\myoledb11
myoledba11.dll	HKEY_CLASSES_ROOT\myoledba11
acmeOledb.dll	HKEY_CLASSES_ROOT\acmeOledb
acmeOledba.dll	HKEY_CLASSES_ROOT\acmeOledba
SAcustom.dll	HKEY_CLASSES_ROOT\SAcustom
SAcustomA.dll	HKEY_CLASSES_ROOT\SAcustomA

8. 使用编辑器的 [查找/替换] 功能将所有在注册表脚本中出现的 *myoledb11* 和 *myoledba11* 更改为您所选择的两个 DLL 名称。
9. 使用编辑器的 [查找/替换] 功能将所有在注册表脚本中出现的 *d:\\mypath\\bin32* 更改为 DLL 的安装位置。请务必使用一对斜线来表示单个斜线。此步骤必须在应用程序安装时自定义。
10. 将注册表脚本保存到磁盘并且运行它。
11. 尝试运行新的提供程序。切勿忘记将 ADO/OLE DB 应用程序更改为使用新的提供程序名称。

下面列出了将要修改的注册表脚本。

```
REGEDIT4
; Special registry entries for a private OLE DB provider.

[HKEY_CLASSES_ROOT\myoledb11]
@="Custom SQL Anywhere OLE DB Provider 11.0"

[HKEY_CLASSES_ROOT\myoledb11\Clsid]
@="{GUID1}"

; Data1 of the following GUID must be 3 greater than the
; previous, for example, 41dfe9f3 + 3 => 41dfe9ee.

[HKEY_CLASSES_ROOT\myoledba11]
@="Custom SQL Anywhere OLE DB Provider 11.0"

[HKEY_CLASSES_ROOT\myoledba11\Clsid]
@="{GUID4}"

; Current version (or version independent prog ID)
; entries (what you get when you have "SQLAny"
; instead of "SQLAny.11")

[HKEY_CLASSES_ROOT\SQLAny]
@="SQL Anywhere OLE DB Provider"

[HKEY_CLASSES_ROOT\SQLAny\Clsid]
@="{GUID1}"

[HKEY_CLASSES_ROOT\SQLAny\CurVer]
@="SQLAny.11"

[HKEY_CLASSES_ROOT\SQLAnyEnum]
@="SQL Anywhere OLE DB Provider Enumerator"

[HKEY_CLASSES_ROOT\SQLAnyEnum\Clsid]
@="{GUID2}"

[HKEY_CLASSES_ROOT\SQLAnyEnum\CurVer]
@="SQLAnyEnum.11"

[HKEY_CLASSES_ROOT\SQLAnyErrorLookup]
@="SQL Anywhere OLE DB Provider Extended Error Support"

[HKEY_CLASSES_ROOT\SQLAnyErrorLookup\Clsid]
@="{GUID3}"

[HKEY_CLASSES_ROOT\SQLAnyErrorLookup\CurVer]
@="SQLAnyErrorLookup.11"

[HKEY_CLASSES_ROOT\SQLAnyA]
```

```
@="SQL Anywhere OLE DB Provider Assist"

[HKEY_CLASSES_ROOT\SQLAnyA\Clsid]
@="{GUID4}"

[HKEY_CLASSES_ROOT\SQLAnyA\CurVer]
@="SQLAnyA.11"

; Standard entries (Provider=SQLAny.11)

[HKEY_CLASSES_ROOT\SQLAny.11]
@="Sybase SQL Anywhere OLE DB Provider 11.0"

[HKEY_CLASSES_ROOT\SQLAny.11\Clsid]
@="{GUID1}"

[HKEY_CLASSES_ROOT\SQLAnyEnum.11]
@="Sybase SQL Anywhere OLE DB Provider Enumerator 11.0"

[HKEY_CLASSES_ROOT\SQLAnyEnum.11\Clsid]
@="{GUID2}"

[HKEY_CLASSES_ROOT\SQLAnyErrorLookup.11]
@="Sybase SQL Anywhere OLE DB Provider Extended Error Support 11.0"

[HKEY_CLASSES_ROOT\SQLAnyErrorLookup.11\Clsid]
@="{GUID3}"

[HKEY_CLASSES_ROOT\SQLAnyA.11]
@="Sybase SQL Anywhere OLE DB Provider Assist 11.0"

[HKEY_CLASSES_ROOT\SQLAnyA.11\Clsid]
@="{GUID4}"

; SQLAny (Provider=SQLAny.11)

[HKEY_CLASSES_ROOT\CLSID\{GUID1}]
@="SQLAny.11"
"OLEDB_SERVICES"=dword:ffffffff

[HKEY_CLASSES_ROOT\CLSID\{GUID1}\ExtendedErrors]
@="Extended Error Service"

[HKEY_CLASSES_ROOT\CLSID\{GUID1}\ExtendedErrors\{GUID3}]
@="Sybase SQL Anywhere OLE DB Provider Error Lookup"

[HKEY_CLASSES_ROOT\CLSID\{GUID1}\InprocServer32]
@="d:\\mypath\\bin32\\myoledb11.dll"
"ThreadingModel"="Both"

[HKEY_CLASSES_ROOT\CLSID\{GUID1}\OLE DB Provider]
@="Sybase SQL Anywhere OLE DB Provider 11.0"

[HKEY_CLASSES_ROOT\CLSID\{GUID1}\ProgID]
@="SQLAny.11"

[HKEY_CLASSES_ROOT\CLSID\{GUID1}\VersionIndependentProgID]
@="SQLAny"

; SQLAnyErrorLookup

[HKEY_CLASSES_ROOT\CLSID\{GUID3}]
@="Sybase SQL Anywhere OLE DB Provider Error Lookup 11.0"
@="SQLAnyErrorLookup.11"
```

```
[HKEY_CLASSES_ROOT\CLSID\{GUID3}\InprocServer32]
@="d:\\mypath\\bin32\\myoledb11.dll"
"ThreadingModel"="Both"

[HKEY_CLASSES_ROOT\CLSID\{GUID3}\ProgID]
@="SQLAnyErrorLookup.11"

[HKEY_CLASSES_ROOT\CLSID\{GUID3}\VersionIndependentProgID]
@="SQLAnyErrorLookup"

; SQLAnyEnum

[HKEY_CLASSES_ROOT\CLSID\{GUID2}]
@="SQLAnyEnum.11"

[HKEY_CLASSES_ROOT\CLSID\{GUID2}\InprocServer32]
@="d:\\mypath\\bin32\\myoledb11.dll"
"ThreadingModel"="Both"

[HKEY_CLASSES_ROOT\CLSID\{GUID2}\OLE DB Enumerator]
@="Sybase SQL Anywhere OLE DB Provider Enumerator"

[HKEY_CLASSES_ROOT\CLSID\{GUID2}\ProgId]
@="SQLAnyEnum.11"

[HKEY_CLASSES_ROOT\CLSID\{GUID2}\VersionIndependentProgID]
@="SQLAnyEnum"

; SQLAnyA

[HKEY_CLASSES_ROOT\CLSID\{GUID4}]
@="SQLAnyA.11"

[HKEY_CLASSES_ROOT\CLSID\{GUID4}\InprocServer32]
@="d:\\mypath\\bin32\\myoledba11.dll"
"ThreadingModel"="Both"

[HKEY_CLASSES_ROOT\CLSID\{GUID4}\ProgID]
@="SQLAnyA.11"

[HKEY_CLASSES_ROOT\CLSID\{GUID4}\VersionIndependentProgID]
@="SQLAnyA"
```

部署 ODBC 客户端

部署 ODBC 客户端最简单的方法是使用 [**部署向导**]。有关详细信息，请参见“使用 [部署向导]”一节第 981 页。

每个 ODBC 客户端计算机都必须具有下列项目：

- **ODBC 驱动程序管理器** Microsoft 为 Windows 操作系统提供 ODBC 驱动程序管理器。SQL Anywhere 包括一个适用于 Linux、Unix 和 Mac OS X 的 ODBC 驱动程序管理器。没有适用于 Windows Mobile 的 ODBC 驱动程序管理器。ODBC 应用程序在没有驱动程序管理器的情况下也可运行，但是在可以使用驱动程序管理器的平台上建议不要这样做。
- **连接信息** 客户端应用程序必须能够访问连接到服务器所需的信息。此信息通常包括在 ODBC 数据源中。

- **SQL Anywhere ODBC 驱动程序** 必须包括在 ODBC 客户端应用程序部署中的文件在下面的 ODBC 驱动程序所需的文件中介绍。

ODBC 驱动程序所需的文件

下表显示运行 SQL Anywhere ODBC 驱动程序所需的文件。这些文件应放在单一目录中。SQL Anywhere 安装将它们全部置于 SQL Anywhere 安装目录的操作系统子目录中（如 *bin32* 或 *bin64*）。

用于 Linux、Unix 和 Mac OS X 平台的 ODBC 驱动程序多线程版本由 "MT" 表示。

平台	所需文件
Windows	*dbodbc11.dll* *dbcon11.dll* *dbicu11.dll* *dbicudt11.dll* *dblg[xx]11.dll* *dbelevate11.exe*
Windows Mobile	*dbodbc11.dll* *dbicu11.dll*（可选） *dbicudt11.dat*（可选） *dblg[xx]11.dll*
Linux、Solaris、HP-UX	*libdbodbc11.so.1* *libdbodbc11_n.so.1* *libdbodm11.so.1* *libdbtasks11.so.1* *libdbicu11.so.1* *libdbicudt11.so.1* *dblg[xx]11.res*

平台	所需文件
Linux、Solaris、HP-UX MT	*libdbodbc11.so.1* *libdbodbc11_r.so.1* *libdbodm11.so.1* *libdbtasks11_r.so.1* *libdbicu11_r.so.1* *libdbicudt11.so.1* *dblg[xx]11.res*
AIX	*libdbodbc11.so* *libdbodbc11_n.so* *libdbodm11.so* *libdbtasks11.so* *libdbicu11.so* *libdbicudt11.so* *dblg[xx]11.res*
AIX MT	*libdbodbc11.so* *libdbodbc11_r.so* *libdbodm11.so* *libdbtasks11_r.so* *libdbicu11_r.so* *libdbicudt11.so* *dblg[xx]11.res*
Mac OS X	*dbodbc11.bundle* *libdbodbc11.dylib* *libdbodbc11_n.dylib* *libdbodm11.dylib* *libdbtasks11.dylib* *libdbicu11.dylib* *libdbicudt11.dylib* *dblg[xx]11.res*

平台	所需文件
Mac OS X MT	*dbodbc11_r.bundle* *libdbodbc11.dylib* *libdbodbc11_r.dylib* *libdbodm11.dylib* *libdbtasks11_r.dylib* *libdbicu11_r.dylib* *libdbicudt11.dylib* *dblg[xx]11.res*

注意

- 对于 Linux 和 Solaris 平台，应创建指向 *.so.1* 文件的链接。此链接名应与删除了 ".1" 版本后缀的文件名匹配。
- 有几个适用于 Linux、Unix 和 Mac OS X 平台的 ODBC 驱动程序多线程 (MT) 版本。其文件名包含 "_r" 后缀。如果您的应用程序需要这些文件，则可以对其进行部署。
- 对于 Windows，驱动程序管理器包括在操作系统中。对于 Linux、Unix 和 Mac OS X 平台，SQL Anywhere 提供了驱动程序管理器。此文件名以 *libdbodm11* 开始。
- 注意，对于 Windows Vista 或更高版本的 Windows，必须包含 SQL Anywhere 已提升的操作代理 (*dbelevate11.exe*) 以支持注册或注销 ODBC 驱动程序所需的权限提升。此文件仅为 ODBC 驱动程序安装或卸载过程所必需。
- 还应包括语言资源库文件。上表显示了带有标志 **[xx]** 的文件。这表示消息文件有多个，每个消息文件支持不同的语言。如果要安装对不同语言的支持，必须添加这些语言的资源文件。
- 对于 Windows，在以下情况下需要 [连接] 窗口支持代码 (*dbcon11.dll*)：最终用户要创建各自的数据源；他们需要在连接到数据库时输入用户 ID 和口令；或者他们出于任何其它目的需要显示 [连接] 窗口。

配置 ODBC 驱动程序

除了将 ODBC 驱动程序文件复制到磁盘上以外，您的安装程序还必须创建一组注册表条目以便正确安装 ODBC 驱动程序。

Windows

SQL Anywhere 安装程序对 Windows 注册表进行更改以标识和配置 ODBC 驱动程序。如果您正为最终用户创建安装程序，则应当对注册表进行相同的设置。

完成此操作最简单的方法就是使用 ODBC 驱动程序的自行注册功能。在 Windows 上，可使用 regsvr32 实用程序；在 Windows Mobile 上，可使用 regsvrce 实用程序。注意，对于 64 位版本的 Windows，您可以注册 64 位和 32 位两种版本的 ODBC 驱动程序。通过使用 ODBC 驱动程序的自

行注册功能，可以确保创建正确的注册表条目。要注册 32 位和 64 位版本的 ODBC 驱动程序，请打开命令提示，然后发出以下命令。

```
regsvr32 install-dir\bin32\dbodbc11.dll
regsvr32 install-dir\bin64\dbodbc11.dll
```

可以使用 regedit 实用程序检查由 ODBC 驱动程序创建的注册表条目。

SQL Anywhere ODBC 驱动程序由以下注册表项中的一组注册表值向系统标识：

```
HKEY_LOCAL_MACHINE\
   SOFTWARE\
      ODBC\
         ODBCINST.INI\
            SQL Anywhere 11
```

如下所示为用于 32 位 Windows 的示例值：

值的名称	值的类型	值的数据
Driver	String	*install-dir\bin32\dbodbc11.dll*
Setup	String	*install-dir\bin32\dbodbc11.dll*

在以下键中也有一个注册表值：

```
HKEY_LOCAL_MACHINE\
   SOFTWARE\
      ODBC\
         ODBCINST.INI\
            ODBC Drivers
```

其值如下：

值的名称	值的类型	值的数据
SQL Anywhere 11	String	Installed

64 位 Windows

对于 64 位 Windows，32 位 ODBC 驱动程序注册表条目（"SQL Anywhere 11" 和 "ODBC Drivers"）位于以下项中：

```
HKEY_LOCAL_MACHINE\
   SOFTWARE\
      Wow6432Node\
         ODBC\
            ODBCINST.INI
```

要查看这些条目，必须使用 64 位版本的 regedit。如果在 64 位 Windows 上无法找到 Wow6432Node，则说明您正在使用 32 位版本的 regedit。

第三方 ODBC 驱动程序

如果您在非 Windows 操作系统上使用第三方 ODBC 驱动程序，请查阅该驱动程序的文档以了解如何配置 ODBC 驱动程序。

部署连接信息

ODBC 客户端连接信息通常以 ODBC 数据源形式部署。您可以按照下列方式之一来部署 ODBC 数据源：

- **编程方式** 将数据源的说明添加到最终用户的注册表或 ODBC 初始化文件中。
- **手工** 向最终用户提供说明，以便他们能够在自己的计算机上创建相应的数据源。

  在 Windows 上，应使用 [ODBC 管理器] 中的 [用户 DSN] 选项卡或 [系统 DSN] 选项卡手工创建数据源。SQL Anywhere ODBC 驱动程序显示用于输入设置的配置窗口。数据源设置包括数据库文件的位置、数据库服务器的名称以及任何启动参数和其它选项。

  在 Unix 平台上，可以使用 SQL Anywhere 的 dbdsn 实用程序来手工创建数据源。数据源设置包括数据库文件的位置、数据库服务器的名称以及任何启动参数和其它选项。

对于每种方式，本节都向您提供需要了解的信息。

数据源类型 (Windows)

有三种数据源：用户数据源、系统数据源和文件数据源。

用户数据源定义存储在注册表的某个部分中，其中包含了当前登录到系统的特定用户的设置。但是，系统数据源对于所有用户和 Windows 服务可用，无论用户是否登录到系统，Windows 服务都运行。给定一个名为 MyApp 的正确配置的系统数据源，任何用户都可以通过在 ODBC 连接字符串中提供 DSN=MyApp 来使用该 ODBC 连接。

文件数据源不保留在注册表中，而是保留在一个专门的目录中。连接字符串必须提供 FileDSN 连接参数才能使用文件数据源。

数据源注册表条目 (Windows)

每个用户数据源都由注册表条目向系统标识。确保可以为数据源定义创建正确的注册表条目最简单的方法是使用 SQL Anywhere dbdsn 实用程序进行创建。

否则，您必须在特定的注册表项中创建一组注册表值。

对于用户数据源，该注册表项如下所示：

```
HKEY_CURRENT_USER\
   SOFTWARE\
      ODBC\
         ODBC.INI\
            user-data-source-name
```

对于系统数据源，该项如下所示：

```
HKEY_LOCAL_MACHINE\
   SOFTWARE\
      ODBC\
         ODBC.INI\
            system-data-source-name
```

该项包含一组注册表值，每个值都与一个连接参数相对应。例如，对应于 SQL Anywhere 11 Demo 系统数据源名 (DSN) 的 SQL Anywhere 11 Demo 项包含以下针对 32 位 Windows 的设置：

值的名称	值的类型	值的数据
Autostop	String	YES
DatabaseFile	String	*samples-dir\demo.db*
Description	String	SQL Anywhere 11 Sample Database
Driver	String	*install-dir\bin32\dbodbc11.dll*
Password	String	sql
ServerName	String	demo11
StartLine	String	*install-dir\bin32\dbeng11.exe*
UserID	String	DBA

注意
建议在所部署应用程序的连接字符串中包含 ServerName 参数。这样可确保当计算机运行着多个 SQL Anywhere 数据库服务器时，应用程序可连接到正确的服务器，并且可以帮助防止出现与计时相关的连接故障。

在上述条目中，*install-dir* 是 SQL Anywhere 的安装目录。对于 64 位 Windows，*bin32* 将替换为 *bin64*。

另外，您必须将数据源名添加到注册表中的数据源列表中。对于用户数据源，使用以下注册表项：

```
HKEY_CURRENT_USER\
   SOFTWARE\
      ODBC\
         ODBC.INI\
            ODBC Data Sources
```

对于系统数据源，使用以下项：

```
HKEY_LOCAL_MACHINE\
   SOFTWARE\
      ODBC\
         ODBC.INI\
            ODBC Data Sources
```

该值将每个数据源与一个 ODBC 驱动程序相关联。值的名称是数据源的名称，值的数据是 ODBC 驱动程序的名称。例如，由 SQL Anywhere 安装的系统数据源名为 SQL Anywhere 11 Demo，其值如下所示：

值的名称	值的类型	值的数据
SQL Anywhere 11 Demo	String	SQL Anywhere 11

小心：ODBC 设置很容易查看
用户数据源配置可以包含敏感的数据库设置（如用户的 ID 和口令）。这些设置以纯文本形式存储在注册表中，可使用由 Microsoft 随操作系统提供的 Windows 注册表编辑器 *regedit.exe* 或 *regedt32.exe,* 来查看。您可以选择加密口令或要求用户在连接时输入口令。

必需的和可选的连接参数

可通过以下方式在 ODBC 连接字符串中标识数据源名称：

```
DSN=UserDataSourceName
```

在 Windows 上，如果在连接字符串中提供了 DSN 参数，则首先在 Windows 注册表中搜索当前用户数据源定义，然后再搜索系统数据源。只有在 ODBC 连接字符串中提供了 FileDSN 时才搜索文件数据源。

下表阐释了当数据源存在且以 DSN 或 FileDSN 参数形式包括在应用程序的连接字符串中时，用户和开发人员应该注意的一些问题。

当数据源……	连接字符串还必须标识……	用户必须提供……
包含 ODBC 驱动程序的名称和位置；数据库文件/服务器的名称；启动参数；用户 ID 和口令。	无额外信息	无额外信息。
包含 ODBC 驱动程序的名称和位置；数据库文件/服务器的名称；启动参数。	无额外信息	用户 ID 和口令（如果未在 DSN 中提供）。
只包含 ODBC 驱动程序的名称和位置。	数据库文件的名称 (DBF=) 和/或数据库服务器的名称 (ENG=)。或者，它可能包含其它连接参数，例如，Userid (UID=) 和 PASSWORD (PWD=)。	用户 ID 和口令（如果未在 DSN 或 ODBC 连接字符串中提供）。
不存在	要使用的 ODBC 驱动程序名称 (Driver=) 和数据库名称 (DBN=)、数据库文件 (DBF=) 和/或数据库服务器 (ENG=)。或者，它可能包含其它连接参数，例如，Userid (UID=) 和 PASSWORD (PWD=)。	用户 ID 和口令（如果未在 ODBC 连接字符串中提供）。

有关 ODBC 连接和配置的详细信息，请参见以下内容：

- “SQL Anywhere 数据库连接” 《SQL Anywhere 服务器 - 数据库管理》.
- 开放式数据库连接 (ODBC) SDK（可从 Microsoft 获得）。

部署嵌入式 SQL 客户端

部署嵌入式 SQL 客户端最简单的方法是使用 [**部署向导**]。有关详细信息，请参见“使用 [部署向导]”一节第 981 页。

部署嵌入式 SQL 客户端涉及到下列内容：

- **安装文件** 每台客户端计算机都必须具有 SQL Anywhere 嵌入式 SQL 客户端应用程序所需的文件。
- **连接信息** 客户端应用程序必须能够访问连接到服务器所需的信息。此信息可包括在 ODBC 数据源中。

为嵌入式 SQL 客户端安装文件

下表显示嵌入式 SQL 客户端所需的文件。

说明	Windows	Linux/Unix	Mac OS X
接口库	*dblib11.dll*	*libdblib11_r.so*	*libdblib11_r.dylib*
线程支持库	N/A	*libdbtasks11_r.so*	*libdbtasks11_r.dylib*
语言资源库	*dblg[xx]11.dll*	*dblg[xx]11.res*	*dblg[xx]11.res*
[连接] 窗口	*dbcon11.dll*	N/A	N/A

注意

- 上表显示了带有标志 **[xx]** 的文件。这表示消息文件有多个，每个消息文件支持不同的语言。如果要安装对不同语言的支持，必须添加这些语言的资源文件。
- 对于 Linux/Unix 上的非多线程应用程序，使用 *libdblib11.so* 和 *libdbtasks11.so*。
- 对于 Mac OS X 上的非多线程应用程序，使用 *libdblib11.dylib* 和 *libdbtasks11.dylib*。
- 如果客户端应用程序使用加密，那么还应当包括适当的加密支持（*dbecc11.dll*、*dbfips11.dll* 或 *dbrsa11.dll*）。
- 如果客户端应用程序使用 ODBC 数据源来保存连接参数，则最终用户必须有正常运行的 ODBC 安装。Microsoft ODBC SDK 中提供了有关部署 ODBC 的指导。

  有关部署 ODBC 信息的详细信息，请参见“部署 ODBC 客户端”一节第 994 页。
- 在以下情况下需要 [连接] 窗口支持 (*dbcon11.dll*)：最终用户将要创建各自的数据源；他们需要在连接到数据库中时输入用户 ID 和口令；或者他们出于任何其它目的而显示 [连接] 窗口。

连接信息

您可以按照下列某种方式来部署嵌入式 SQL 连接信息：

- **手工** 向最终用户提供关于在他们的计算机上创建相应数据源的说明。
- **文件** 以应用程序可以读取的格式分发包含连接信息的文件。
- **ODBC 数据源** 您可以使用 ODBC 数据源保存连接信息。

部署 JDBC 客户端

您必须安装 Java 运行时环境以便使用 JDBC。建议使用版本 1.6.0 或更高版本。

除了 Java 运行时环境，每个 JDBC 客户端都还需要 iAnywhere JDBC 驱动程序或 jConnect。

iAnywhere JDBC 驱动程序

要部署 iAnywhere JDBC 驱动程序，您必须部署下列文件：

- *jodbc.jar* 此文件必须在应用程序的类路径中。此文件位于 SQL Anywhere 安装目录的 *java* 文件夹中。
- *dbjodbc11.dll* 此文件必须位于系统路径中。在 Linux 和 Unix 环境中，此文件是名为 *libdbjodbc11.so* 的共享库。在 MAC OS X 中，此文件是名为 *libdbjodbc11.dylib* 的共享库。
- ODBC 驱动程序文件。有关详细信息，请参见“ODBC 驱动程序所需的文件”一节第 995 页。

jConnect JDBC 驱动程序

要部署 jConnect JDBC 驱动程序，您必须部署以下文件：

- jConnect 驱动程序文件。有关 jConnect 软件版本和 jConnect 文档的信息，请参见 jConnect for JDBC。
- 当使用 TDS 客户端时（Open Client 或基于 jConnect 的客户端），可以选择以明文或加密的形式发送连接口令。后者通过执行 TDS 加密的口令握手来完成。握手会用到专用/公共密钥加密。对于生成 RSA 专用/公共密钥对以及对解密加密的口令的支持包含在一个特殊的库中。SQL Anywhere 服务器在其系统路径中必须能够找到该库文件。对于 Windows，该文件名为 *dbrsakp11.dll*。此 DLL 同时有 64 位和 32 位两种版本。在 Linux 和 Unix 环境中，此文件是名为 *libdbrsakp11.so* 的共享库。在 MAC OS X 中，此文件是名为 *libdbrsakp11.dylib* 的共享库。如果不使用此功能，则无需该文件。

JDBC 数据库连接 URL

为了连接到数据库，您的 Java 应用程序需要一个 URL。此 URL 指定驱动程序、要使用的计算机以及数据库服务器所监听的端口。

有关 URL 的详细信息，请参见“将 URL 提供给驱动程序”一节第 488 页。

部署 Open Client 应用程序

要部署 Open Client 应用程序，每台客户端计算机都需要 Sybase Open Client 产品。您必须从 Sybase 单独购买 Open Client 软件。软件中包含其自身的安装指导。

当使用 TDS 客户端时（Open Client 或基于 jConnect 的客户端），可以选择以明文或加密的形式发送连接口令。后者通过执行 TDS 加密的口令握手来完成。握手会用到专用/公共密钥加密。对于生成 RSA 专用/公共密钥对以及对解密加密的口令的支持包含在一个特殊的库中。SQL Anywhere 服务器在其系统路径中必须能够找到该库文件。对于 Windows，该文件名为 *dbrsakp11.dll*。此 DLL 同时有 64 位和 32 位两种版本。在 Linux 和 Unix 环境中，此文件是名为 *libdbrsakp11.so* 的共享库。在 MAC OS X 中，此文件是名为 *libdbrsakp11_r.dylib* 的共享库。如果不使用此功能，则无需该文件。

Open Client 客户端的连接信息保存在接口文件中。有关接口文件的信息，请参见 Open Client 文档和“配置 Open Server”一节 《SQL Anywhere 服务器 - 数据库管理》。

部署管理工具

在许可协议的范围内，您可以部署一组管理工具，其中包括 Interactive SQL、Sybase Central 和 SQL Anywhere 控制台实用程序。

部署管理工具最简单的方法是使用 [**部署向导**]。有关详细信息，请参见“使用 [部署向导]”一节 第 981 页。

有关管理工具的系统要求的信息，请参见 http://www.sybase.com/detail?id=1062617。

初始化文件可以简化管理工具的部署。管理工具的每个启动程序可执行文件（Sybase Central、Interactive SQL 和控制台实用程序）都可以有一个相对应的 *.ini* 文件。这使得不再需要用于 JAR 文件位置的注册表条目和固定目录结构。这些 *ini* 文件位于同一目录中，且具有与可执行文件相同的文件名。

- **dbconsole.ini** 控制台实用程序初始化文件的名称。
- **dbisql.ini** Interactive SQL 初始化文件的名称。
- **scjview.ini** Sybase Central 初始化文件的名称。

初始化文件包含有关如何装载数据库管理工具的详细信息。例如，初始化文件可能包含以下行：

- **JRE_DIRECTORY=<path>** 这是所需 JRE 的位置。**JRE_DIRECTORY** 说明为必需。
- **VM_ARGUMENTS=<any required VM arguments>** VM 参数以分号 (;) 分隔。任何包含空格的路径值都应当括在引号中。可通过使用管理工具的 `-batch` 选项（例如 `scjview -batch`）和检查已创建的文件来发现 VM 参数。**VM_ARGUMENTS** 说明是可选的。
- **JAR_PATHS=<path1;path2;...>** 包含用于程序的 JAR 文件的目录分隔列表。各项以分号 (;) 分隔。**JAR_PATHS** 说明是可选的。
- **ADDITIONAL_CLASSPATH=<path1;path2;...>** 类路径值以分号 (;) 分隔。**ADDITIONAL_CLASSPATH** 说明是可选的。
- **LIBRARY_PATHS=<path1;path2;...>** DLL/共享对象的路径。各项以分号 (;) 分隔。**LIBRARY_PATHS** 说明是可选的。
- **APPLICATION_ARGUMENTS=<arg1;arg2;...>** 这些是所有应用程序的参数。各项以分号 (;) 分隔。可以通过使用管理工具的 `-batch` 选项（例如 `scjview -batch`）和检查已创建的文件来发现应用程序参数。**APPLICATION_ARGUMENTS** 说明是可选的。

以下是 Sybase Central 的示例初始化文件的内容。

```
JRE_DIRECTORY=c:\Sun\JRE160_x86
VM_ARGUMENTS=-Xmx200m
JAR_PATHS=c:\scj\jars;c:\scj\jhelp
ADDITIONAL_CLASSPATH=
LIBRARY_PATHS=c:\scj\bin
APPLICATION_ARGUMENTS=-screpository=C:\Documents and Settings\All Users
\Application Data\Sybase Central 6.0.0;-installdir=c:\scj
```


此处假定 32 位 Sun JRE 的一份副本位于 *c:\Sun\JRE160_x86* 中。同时假定 Sybase Central 可执行文件和共享库 (DLL)（如 *jsyblib600*）存储在 *c:\scj\bin* 中。SQL Anywhere JAR 文件存储在 *c:\scj\jars* 中。Sun JavaHelp 2.0 JAR 文件存储在 *c:\scj\jhelp* 中。

注意
当部署应用程序时，需要使用个人数据库服务器 (dbeng11)，通过 dbinit 实用程序来创建数据库。如果在没有运行其它数据库服务器时，从本地计算机的 Sybase Central 创建数据库，也需要使用个人数据库服务器。

在 Windows 上部署管理工具而不使用 InstallShield

本节介绍如何在不使用 InstallShield 的情况下，在 Windows 计算机上安装 Interactive SQL (dbisql)、Sybase Central（包括 SQL Anywhere、MobiLink、QAnywhere 和 UltraLite 插件）和 SQL Anywhere 控制台实用程序 (dbconsole)。目标读者是那些希望为这些管理工具创建安装程序的用户。

这些信息适用于除 Windows Mobile 以外的所有 Windows 平台。这里给出的说明特定于 11.0.1 版，不适用于此软件的早期版本或后期版本。

检查许可协议
重新分发文件受到许可协议的制约。本文中的任何声明都不能替代许可协议中的任何内容。在考虑部署之前，请检查许可协议。

开始之前

阅读本节前，您应该了解 Windows 注册表，包括 REGEDIT 应用程序。注册表值的名称区分大小写。

修改注册表存在危险
用户要自己承担修改注册表的风险。建议修改注册表前先备份系统。

部署管理工具时需要执行以下步骤：

1. 确定要部署的软件。
2. 复制所需文件。
3. 向 Windows 注册管理工具。
4. 更新系统路径。
5. 向 Sybase Central 注册插件。
6. 向 Windows 注册 SQL Anywhere ODBC 驱动程序。
7. 向 Windows 注册联机帮助文件。

以下各节将详细介绍每一步骤。

第 1 步：确定要部署的软件

可以安装以下软件包的任意组合：

- Interactive SQL
- 带有 SQL Anywhere 插件的 Sybase Central
- 带有 MobiLink 插件的 Sybase Central
- 带有 QAnywhere 插件的 Sybase Central
- 带有 UltraLite 插件的 Sybase Central
- SQL Anywhere 控制台实用程序 (dbconsole)

安装上述任何软件包时还需要以下组件：

- SQL Anywhere ODBC 驱动程序
- Java 运行时环境 (JRE) 1.6.0 版

注意
要在 Mac OS X 上检查 JRE 版本，请转到 [**Apple**] 菜单，然后选择 [**System Preferences**] » [**Software Updates**]。单击 [**Installed Updates**]，以获得已应用更新的列表。如果 Java 1.6.0 不在列表中，请访问 developer.apple.com/java/download/。

以下各节的说明进行了结构化安排，这样您可以安装六个软件包中的任意几个（或全部），而不会引起冲突。

第 2 步：复制所需文件

管理工具需要特定的目录结构。您可以将目录树放置在任何驱动器的任何目录中。在以下讨论中，将 *c:\sa11* 用作示例安装文件夹。软件必须安装到具有以下布局的目录树结构中：

目录	说明
sa11	根文件夹。虽然以下步骤假定您要在 *c:\sa11* 中安装，但您可以将该目录放置在任何位置（例如 *C:\Program Files\SQLAny11*）。
sa11\java	保存 Java 程序的 JAR 文件。
sa11\bin32	保存程序使用的本地 32 位 Windows 组件，包括启动应用程序的程序。
sa11\Sun\JavaHelp-2_0	JavaHelp 运行时库。
sa11\Sun\jre160_x86	32 位 Java 运行时环境。

x64
在大多数平台上，基于 Java 的管理工具是 32 位应用程序。除 Mac OS X 外，其它平台没有 64 位版本。32 位管理工具可在基于 x64 并支持 32 位 JRE 的平台上部署。

Itanium 64
没有任何基于 Java 的管理工具可在 Itanium (ia64) 平台上部署。但是，有一个 Interactive SQL 的本地版本，此版本不像 Java 版本那样具有丰富的功能。请参见“部署 dbisqlc”一节第 1028 页。

下表列出了每个软件包所需的文件。列出所需的文件，然后将其复制到上面介绍的目录结构中。一般说来，应从 SQL Anywhere 的已安装副本中获取这些文件。

文件	Interactive SQL	带有 SQL Anywhere 插件的 Sybase Central	带有 MobiLink 插件的 Sybase Central	带有 QAnywhere 插件的 Sybase Central	带有 UltraLite 插件的 Sybase Central	SQL Anywhere 控制台
documentation\[xx]\htmlhelp\sqlanywhere_[xx]11.chm	X	X	X	X	X	X
documentation\sqlanywhere_[xx]11.map	X	X	X	X	X	X
c:\windows\system32\keyHH.exe[1]	X	X	X	X	X	X
java\jodbc.jar	X	X	X	X	X	X
java\JComponents1101.jar	X	X	X	X	X	X
java\jlogon.jar	X	X	X	X	X	X
java\SCEditor600.jar	X	X	X	X	X	X
java\jsyblib600.jar	X	X	X	X	X	X
Sun\JavaHelp-2_0\jh.jar	X	X	X	X	X	X
Sun\jre160_x86\...	X	X	X	X	X	X
bin32\jsyblib600.dll	X	X	X	X	X	X
bin32\dblib11.dll	X	X	X	X	X	X
bin32\dbjodbc11.dll	X	X	X	X	X	X
bin32\dbodbc11.dll	X	X	X	X	X	X

文件	Interactive SQL	带有 SQL Anywhere 插件的 Sybase Central	带有 MobiLink 插件的 Sybase Central	带有 QAnywhere 插件的 Sybase Central	带有 UltraLite 插件的 Sybase Central	SQL Anywhere 控制台
bin32\dbcon11.dll	X	X	X	X	X	X
bin32\dblg[xx]11.dll	X	X	X	X	X	X
bin32\dbtool11.dll	X	X	X			
bin32\dbelevate11.exe（Vista 或更高版本）	X	X				X
bin32\dbisql.com	X					
bin32\dbisql.exe	X					
java\isql.jar	X	X	X		X	
java\saip11.jar	X	X	X		X	
bin32\scjview.exe		X	X	X	X	
bin32\scvw[xx]600.jar		X	X	X	X	
java\sybasecentral600.jar		X	X	X	X	
java\salib.jar		X	X	X	X	
java\saplugin.jar		X				
java\debugger.jar		X				
bin32\dbput11.dll		X	X			
java\apache_files.txt			X	X		
java\apache_license_1.1.txt			X	X		
java\apache_license_2.0.txt			X	X		
java\log4j.jar			X	X		
java\mlplugin.jar			X			
java\mldesign.jar			X	X		

文件	Interactive SQL	带有 SQL Anywhere 插件的 Sybase Central	带有 MobiLink 插件的 Sybase Central	带有 QAnywhere 插件的 Sybase Central	带有 UltraLite 插件的 Sybase Central	SQL Anywhere 控制台
java\stax-api-1.0.jar			X			
java\wstx-asl-3.2.6.jar			X			
java\velocity.jar			X			
java\velocity-dep.jar			X			
java\qaplugin.jar				X		
java\qaconnector.jar				X		
java\mlstream.jar				X		
bin32\qaagent.exe				X		
bin32\dbicu11.dll				X		
bin32\dbicudt11.dll				X		
bin32\dbghelp.dll				X		
bin32\dbinit.exe				X		
java\ulplugin.jar					X	
bin32\dbconsole.exe						X
java\DBConsole.jar						X

[1] 使用的操作系统不同，Windows 系统目录的确切名称也不同。

上表显示了带有标志 **[xx]** 的文件。这表示消息文件有多个，每个消息文件支持不同的语言。如果要安装对不同语言的支持，必须添加这些语言的资源文件。有关详细信息，请参见下一节国际消息和上下文相关帮助文件。

上面的一些文件路径以 "..." 结尾。这表明应该复制整个树，包括子目录。

管理工具要求使用 JRE 1.6.0。除非有特定需要，否则不应替代具有更高补丁版本的 JRE。从目录 *install-dir\Sun\jre160_x86* 中复制 32 位版本的 JRE 文件。复制整个 *jre160_x86* 树，包括其子目录。

作为参考，*sqlanywhere.jpr* 文件包含 Sybase Central 的 SQL Anywhere 插件的 jar 文件列表。

mobilink.jpr 文件包含 Sybase Central 的 MobiLink 插件的 jar 文件列表。

qanywhere.jpr 文件包含 Sybase Central 的 QAnywhere 插件的 jar 文件列表。部署 QAnywhere 插件时，dbinit 是必需的。有关部署数据库工具的信息，请参见“部署数据库实用程序”一节 第 1037 页。

ultralite.jpr 文件包含 Sybase Central 的 UltraLite 插件的 jar 文件列表。

国际消息和上下文相关帮助文件

管理工具的所有显示文本和上下文相关帮助都已从英语翻译为法语、德语、日语和简体中文。每种语言的资源保存在不同的文件中。英语文件的文件名中包含 **en**。法语文件的文件名与之类似，但使用的是 **fr** 而不是 **en**。德语文件名包含 **de**，日语文件名包含 **ja**，而中文文件名包含 **zh**。

如果要安装对不同语言的支持，必须添加其它语言的消息文件。翻译后的文件如下所示：

dblgen11.dll	英语
dblgde11.dll	德语
dblgfr11.dll	法语
dblgja11.dll	日语
dblgzh11.dll	简体中文

还必须添加其它语言的上下文相关帮助文件。可用的翻译后文件如下所示：

scvwen600.jar	英语
scvwde600.jar	德语
scvwfr600.jar	法语
scvwja600.jar	日语
scvwzh600.jar	简体中文

这些文件包含在 SQL Anywhere 的本地化版本中。

第 3 步：向 Windows 注册管理工具

您必须为管理工具设置以下注册表值。注册表值的名称区分大小写。

- 在 **HKEY_LOCAL_MACHINE\SOFTWARE\Sybase\Sybase Central\6.0.0** 中
  - **Language**　Sybase Central 所用语言的双字母代码。此参数必须是下列项之一：**EN**、**DE**、**FR**、**JA** 或 **ZH** 分别对应英语、德语、法语、日语和简体中文。
- 在 **HKEY_LOCAL_MACHINE\SOFTWARE\Sybase\SQL Anywhere\11.0** 中

- **Location** 包含 Sybase Central 文件的安装文件夹根路径的完全限定路径（缺省为 *C:\Program Files\SQL Anywhere 11*）。
- **Language** SQL Anywhere 所用语言的双字母代码。此参数必须是下列项之一：**EN**、**DE**、**FR**、**JA** 或 **ZH** 分别对应英语、德语、法语、日语和简体中文。

在 64 位 Windows 上，这些注册表条目在 32 位注册表 (**SOFTWARE\Wow6432Node\Sybase**) 中。

路径*不*应以反斜线结尾。

安装程序可通过创建然后执行一个 *.reg* 文件来封装所有这些信息。以下示例 *.reg* 文件使用我们的示例安装文件夹 *c:\sa11*：

```
REGEDIT4

[HKEY_LOCAL_MACHINE\SOFTWARE\Sybase\Sybase Central\6.0.0]
"Language"="EN"

[HKEY_LOCAL_MACHINE\SOFTWARE\Sybase\SQL Anywhere\11.0]
"Location"="c:\\sa11"
"Language"="EN"
```

在 *.reg* 文件中，文件路径中的反斜线必须由另一个反斜线进行转义。

第 4 步：更新系统路径

要运行管理工具，必须在路径中包含带有 *.exe* 和 *.dll* 文件的目录。必须将 *c:\sa11\bin32* 目录添加到系统路径中。

在 Windows 上，系统路径存储在以下注册表项中：

```
HKEY_LOCAL_MACHINE\
  SYSTEM\
    CurrentControlSet\
      Control\
        Session Manager\
          Environment\
            Path
```

部署 Interactive SQL 或 Sybase Central 时，请将以下目录添加到现有路径中：

```
c:\sa11\bin32
```

第 5 步：注册 Sybase Central 插件

此步骤涉及 Sybase Central 的配置。如果不安装 Sybase Central，则可以跳过此步骤。

Sybase Central 需要一个列出已安装插件的配置文件。安装程序必须创建此文件。请注意，它所包含的几个 JAR 文件的完整路径可能会依据软件的安装位置而更改。

文件名为 *.scRepository600*。在 Windows XP/200x 上，它位于 *%allusersprofile%\application data\Sybase Central 6.0.0* 文件夹中。在 Windows Vista 上，它位于 *%ProgramData%\Sybase Central 6.0.0* 文件夹中。此文件是一个纯文本文件，包含 Sybase Central 应装载的插件的一些基本信息。

在 Windows Vista 上，所有用户均应具有对包含 *.scRepository600* 文件的目录的读取访问权限。这可以通过使用以下命令来实现。要手工实现这一操作，请打开管理员命令提示符窗口（右击 [Command Prompt] 然后单击 [Run As Administrator]）。

```
icacls "%ProgramData%\Sybase Central 6.0.0" /grant everyone:F
```

SQL Anywhere 的提供程序信息是使用以下命令在存储库文件中创建的。

```
scjview.exe -register "C:\Program Files\SQL Anywhere 11\java\sqlanywhere.jpr"
```

sqlanywhere.jpr 文件的内容如下所示（为便于显示，一些条目已拆分为多行）。**AdditionalClasspath** 行必须在 *.jpr* 文件的一个单独的行上输入。

```
PluginName=SQL Anywhere 11
PluginId=sqlanywhere1100
PluginClass=ianywhere.sa.plugin.SAPlugin
PluginFile=C:\Program Files\SQL Anywhere 11\java\saplugin.jar
AdditionalClasspath=
    C:\Program Files\SQL Anywhere 11\java\isql.jar;
    C:\Program Files\SQL Anywhere 11\java\salib.jar;
    C:\Program Files\SQL Anywhere 11\java\JComponents1101.jar;
    C:\Program Files\SQL Anywhere 11\java\jlogon.jar;
    C:\Program Files\SQL Anywhere 11\java\debugger.jar;
    C:\Program Files\SQL Anywhere 11\java\jodbc.jar
ClassloaderId=SA1100
InitialLoadOrder=0
```

最初安装 SQL Anywhere 时，在 SQL Anywhere 安装目录的 *java* 文件夹中创建 *sqlanywhere.jpr* 文件。将其用作必须作为安装过程一部分而创建的 *.jpr* 文件的模型。此文件还有用于 MobiLink、QAnywhere 和 UltraLite 的版本，其名称分别为 *mobilink.jpr*、*qanywhere.jpr* 和 *ultralite.jpr*。它们也都位于 *java* 文件夹中。

下面是按照上述过程创建的 *.scRepository600* 文件的一部分。为便于显示，一些条目已拆分为多行。在此文件中，每个条目出现在一行中：

```
#  Version: 6.0.0.1154
#  Fri Feb 22 10:22:20 EST 2008
#
SCRepositoryInfo/Version=4
#
Providers/sqlanywhere1100/Version=11.0.1.1297
Providers/sqlanywhere1100/UseClassloader=true
Providers/sqlanywhere1100/ClassloaderId=SA1100
Providers/sqlanywhere1100/Classpath=
    C:\\Program Files\\SQL Anywhere 11\\java\\saplugin.jar
Providers/sqlanywhere1100/Name=SQL Anywhere 11
Providers/sqlanywhere1100/AdditionalClasspath=
    C:\\Program Files\\SQL Anywhere 11\\java\\isql.jar;
    C:\\Program Files\\SQL Anywhere 11\\java\\salib.jar;
    C:\\Program Files\\SQL Anywhere 11\\java\\JComponents1101.jar;
    C:\\Program Files\\SQL Anywhere 11\\java\\jlogon.jar;
    C:\\Program Files\\SQL Anywhere 11\\java\\debugger.jar;
    C:\\Program Files\\SQL Anywhere 11\\java\\jodbc.jar
Providers/sqlanywhere1100/Provider=ianywhere.sa.plugin.SAPlugin
Providers/sqlanywhere1100/ProviderId=sqlanywhere1100
Providers/sqlanywhere1100/InitialLoadOrder=0
#
```

注意

- 您的安装程序应使用上述方法创建一个与此类似的文件。唯一需要更改的是 Classpath 和 AdditionalClasspath 行中 JAR 文件的完全限定路径。
- 上面显示的 AdditionalClasspath 行经换行占用了其它行。在 *.scRepository600* 文件中，它们必须单独位于一行中。
- 在 *.scRepository600* 文件中，反斜线字符 (\) 用转义序列 \\ 表示。
- 第一行指示 *.scRepository600* 文件的版本。
- 以 # 开头的行是注释。

第 6 步：创建 Sybase Central 的连接配置文件

此步骤涉及 Sybase Central 的配置。如果不安装 Sybase Central，则可以跳过此步骤。

将 Sybase Central 安装到系统时，会在 *.scRepository600* 文件中创建 **SQL Anywhere 11 Demo** 的连接配置文件。如果您不想创建一个或多个连接配置文件，则可以跳过此步骤。

以下命令用于创建 **SQL Anywhere 11 Demo** 连接配置文件。将此作为您创建自己的连接配置文件的模型。

```
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Name" "SQL Anywhere
11 Demo"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/FirstTimeStart"
"false"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Description"
"Suitable Description"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/ProviderId"
"sqlanywhere1100"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Provider" "SQL
Anywhere 11"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Data/
ConnectionProfileSettings" "DSN\eSQL^0020Anywhere^002011^0020Demo;UID
\eDBA;PWD\e35c624d517fb"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Data/
ConnectionProfileName" "SQL Anywhere 11 Demo"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Data/
ConnectionProfileType" "SQL Anywhere"
```

连接配置文件字符串和值可从 *.scRepository600* 文件中抽取。使用 Sybase Central 定义连接配置文件，然后在 *.scRepository600* 文件中查找相对应的行。

下面是按照上述过程创建的 *.scRepository600* 文件的一部分。为便于显示，一些条目已拆分为多行。在此文件中，每个条目出现在一行中：

```
#  Version: 6.0.0.1154
#  Fri Feb 23 13:09:14 EST 2007
#
ConnectionProfiles/SQL Anywhere 11 Demo/Name=SQL Anywhere 11 Demo
ConnectionProfiles/SQL Anywhere 11 Demo/FirstTimeStart=false
ConnectionProfiles/SQL Anywhere 11 Demo/Description=Suitable Description
ConnectionProfiles/SQL Anywhere 11 Demo/ProviderId=sqlanywhere1100
ConnectionProfiles/SQL Anywhere 11 Demo/Provider=SQL Anywhere 11
ConnectionProfiles/SQL Anywhere 11 Demo/Data/ConnectionProfileSettings=
    DSN\eSQL^0020Anywhere^002011^0020Demo;
    UID\eDBA;
```



```
    PWD\e35c624d517fb
ConnectionProfiles/SQL Anywhere 11 Demo/Data/ConnectionProfileName=
    SQL Anywhere 11 Demo
ConnectionProfiles/SQL Anywhere 11 Demo/Data/ConnectionProfileType=
     SQL Anywhere
```

第 7 步：注册 SQL Anywhere ODBC 驱动程序

必须先安装 SQL Anywhere ODBC 驱动程序，然后 iAnywhere JDBC 驱动程序才能在管理工具中对其进行使用。

有关详细信息，请参见“配置 ODBC 驱动程序”一节第 997 页。

在 Linux、Solaris 和 Mac OS X 上部署管理工具

本节介绍如何在 Linux、Solaris 和 Mac OS X 计算机上安装 Interactive SQL (dbisql)、Sybase Central（包括 SQL Anywhere、MobiLink 和 QAnywhere 插件）以及 SQL Anywhere 控制台实用程序 (dbconsole)。目标读者是那些希望为这些管理工具创建安装程序的用户。

这里给出的说明特定于 11.0.1 版，不适用于此软件的早期版本或后期版本。

还请注意，dbisqlc 命令行实用程序在 Linux、Solaris、Mac OS X、HP-UX 和 AIX 上也受支持。请参见“部署 dbisqlc”一节第 1028 页。

检查许可协议
重新分发文件受到许可协议的制约。本文中的任何声明都不能替代许可协议中的任何内容。在考虑部署之前，请检查许可协议。

开始之前

开始前，必须在一台计算机上安装 SQL Anywhere 作为程序文件的源。这是用于部署的**参考安装位置**。

涉及的一般步骤如下所示：

1. 确定要部署的程序。
2. 复制所需文件。
3. 设置环境变量。
4. 注册 Sybase Central 插件。

以下各节将详细介绍每一步骤。

第 1 步：确定要部署的内容

可以安装以下软件包的任意组合：

- Interactive SQL

- 带有 SQL Anywhere 插件的 Sybase Central
- 带有 MobiLink 插件的 Sybase Central
- 带有 QAnywhere 插件的 Sybase Central
- SQL Anywhere 控制台实用程序 (dbconsole)

安装上述任何软件包时还需要以下组件：

- SQL Anywhere ODBC 驱动程序
- Java 运行时环境 (JRE) 1.6.0 版。在 Linux/Solaris 上为 32 位版本的 JRE。在 Mac OS X 上为 64 位版本的 JRE。

下节中的说明进行了结构化安排，这样您可以安装五个软件包中的任意几个（或全部），而不引起冲突。

第 2 步：复制所需文件

安装程序应复制 SQL Anywhere 安装程序所安装文件的子集。必须保持相同的目录结构。必须将所有文件安装在 */opt/sqlanywhere11/* 目录下。

从参考 SQL Anywhere 安装位置复制这些文件时，应保留有关这些文件的权限。一般说来，允许所有用户和组读取和执行所有文件。

下表列出了 Linux 和 Sun Solaris 上每个软件包所需的文件。列出所需的文件，然后将其复制到上面介绍的目录结构中。一般说来，应从 SQL Anywhere 的已安装副本中获取这些文件。

文件	Interactive SQL	带有 SQL Anywhere 插件的 Sybase Central	带有 MobiLink 插件的 Sybase Central	带有 QAnywhere 插件的 Sybase Central	带有 UltraLite 插件的 Sybase Central [1]	SQL Anywhere 控制台
java/jodbc.jar	X	X	X	X	X	X
java/JComponents1101.jar	X	X	X	X	X	X
java/jlogon.jar	X	X	X	X	X	X
java/SCEditor600.jar	X	X	X	X	X	X
java/jsyblib600.jar	X	X	X	X	X	X
lib32/libjsyblib600_r.so.1	X	X	X	X	X	X
sun/javahelp-2_0/jh.jar	X	X	X	X	X	X

文件	Interactive SQL	带有 SQL Anywhere 插件的 Sybase Central	带有 MobiLink 插件的 Sybase Central	带有 QAnywhere 插件的 Sybase Central	带有 UltraLite 插件的 Sybase Central [1]	SQL Anywhere 控制台
jre_1.6.0_linux_sun_i586/...（仅限 Linux）	X	X	X	X	X	X
jre_1.6.0_solaris_sun_sparc/...（仅限 Solaris）	X	X	X	X	X	X
lib32/libdblib11_r.so.1	X	X	X	X	X	X
lib32/libdbjodbc11.so.1	X	X	X	X	X	X
lib32/libdbodbc11_r.so.1	X	X	X	X	X	X
lib32/libdbodm11.so.1	X	X	X	X	X	X
lib32/libdbtasks11_r.so.1	X	X	X	X		X
res/dblg[xx]11.res	X	X	X	X	X	X
lib32/libdbtool11_r.so.1	X	X	X			
bin32/dbisql	X	X			X	
java/isql.jar	X	X	X		X	
bin32/scjview		X	X	X	X	
bin32/scvw[xx]600.jar		X	X	X	X	
java/sybasecentral600.jar		X	X	X	X	
java/salib.jar		X	X	X	X	
java/saplugin.jar		X			X	
java/debugger.jar		X				
lib32/libdbput11_r.so.1		X				
lib32/libmljodbc11.so.1			X			
java/apache_files.txt			X	X		

文件	Interactive SQL	带有 SQL Anywhere 插件的 Sybase Central	带有 MobiLink 插件的 Sybase Central	带有 QAnywhere 插件的 Sybase Central	带有 UltraLite 插件的 Sybase Central [1]	SQL Anywhere 控制台
java/apache_license_1.1.txt			X	X		
java/apache_license_2.0.txt			X	X		
java/log4j.jar			X	X		
java/mlplugin.jar			X			
java/mldesign.jar			X	X		
java/stax-api-1.0.jar			X			
java/wstx-asl-3.2.6.jar			X			
java/velocity.jar			X			
java/velocity-dep.jar			X			
java/qaplugin.jar				X		
java/qaconnector.jar				X		
java/mlstream.jar				X		
lib32/libdbicu11_r.so				X		
lib32/libdbicudt11.so				X		
bin32/dbinit				X		
java/ulplugin.jar					X	
lib32/libulscp11_r.so.1					X	
lib32/libulhltool11_r.so.1					X	
res/ulg[xx]11.res					X	
bin32/uleng11					X	
bin32/ulcreate					X	

文件	Interactive SQL	带有 SQL Anywhere 插件的 Sybase Central	带有 MobiLink 插件的 Sybase Central	带有 QAnywhere 插件的 Sybase Central	带有 UltraLite 插件的 Sybase Central [1]	SQL Anywhere 控制台
bin32/ulload					X	
bin32/ulunload					X	
bin32/ulsync					X	
bin32/ulinit					X	
bin32/ulvalid					X	
bin32/ulerase					X	
bin32/dbconsole						X
java/DBConsole.jar						X

[1] 只在 Linux 上支持 UltraLite。

上表显示了带有标志 **[xx]** 的文件。仅对于 Linux，消息文件有多个，每个消息文件支持不同的语言。如果要安装对不同语言的支持，必须添加这些语言的资源文件。有关详细信息，请参见下一节“国际消息和上下文相关帮助文件”一节第 1023 页。

下表列出了 Mac OS X 上每个软件包所需的文件。请制作一个所需文件的列表，然后将其复制到上面介绍的目录结构中。一般说来，应从 SQL Anywhere 的已安装副本中获取这些文件。

文件	Interactive SQL	带有 SQL Anywhere 插件的 Sybase Central	带有 MobiLink 插件的 Sybase Central	带有 QAnywhere 插件的 Sybase Central	SQL Anywhere 控制台
java/jodbc.jar	X	X	X	X	X
java/JComponents1101.jar	X	X	X	X	X
java/jlogon.jar	X	X	X	X	X
java/SCEditor600.jar	X	X	X	X	X
java/jsyblib600.jar	X	X	X	X	X
lib64/libjsyblib600_r.dylib	X	X	X	X	X

文件	Interactive SQL	带有 SQL Anywhere 插件的 Sybase Central	带有 MobiLink 插件的 Sybase Central	带有 QAnywhere 插件的 Sybase Central	SQL Anywhere 控制台
sun/javahelp-2_0/jh.jar	X	X	X	X	X
lib64/libdblib11_r.dylib	X	X	X	X	X
lib64/libdbjodbc11.dylib	X	X	X	X	X
lib64/libdbodbc11_r.dylib	X	X	X	X	X
lib64/libdbodm11.dylib	X	X	X	X	X
lib64/libdbtasks11_r.dylib	X	X	X	X	X
res/dblgen11.res	X	X	X	X	X
lib64/libdbtool11_r.dylib	X	X	X		
bin64/dbisql	X	X			
java/isql.jar	X	X	X		
bin64/scjview		X	X	X	
bin64/scvwen600.jar		X	X	X	
java/sybasecentral600.jar		X	X	X	
java/salib.jar		X	X	X	
java/saplugin.jar		X			
java/debugger.jar		X			
lib64/libdbput11_r.dylib		X			
libmljodbc11.dylib			X		
java/apache_files.txt			X	X	
java/apache_license_1.1.txt			X	X	
java/apache_license_2.0.txt			X	X	
java/log4j.jar			X	X	

文件	Interactive SQL	带有 SQL Anywhere 插件的 Sybase Central	带有 MobiLink 插件的 Sybase Central	带有 QAnywhere 插件的 Sybase Central	SQL Anywhere 控制台
java/mlplugin.jar			X		
java/mldesign.jar			X	X	
java/stax-api-1.0.jar			X		
java/wstx-asl-3.2.6.jar			X		
java/velocity.jar			X		
java/velocity-dep.jar			X		
java/qaplugin.jar				X	
java/qaconnector.jar				X	
java/mlstream.jar				X	
lib64/libdbicu11_r.dylib				X	
lib64/libdbicudt11.dylib				X	
bin32/dbinit				X	
bin64/dbconsole					X
java/DBConsole.jar					X

对于 Linux/Solaris，管理工具需要 32 位版本的 JRE 1.6.0。Mobilink 服务器需要 64 位版本的 JRE 1.6.0。对于 Mac OS X，管理工具需要 64 位版本。除非有特定需要，否则不应替代具有更高补丁版本的 JRE。不是所有的 JRE 平台版本均与 SQL Anywhere 捆绑。SQL Anywhere 附带的平台支持 x86/x64 上的 Linux 和 Solaris SPARC。其它平台版本必须从相应服务商获得。例如，如果使用 Linux 安装，则复制整个 *jre_1.6.0_linux_sun_i586* 树，包括子目录。

如果您所需要的平台随 SQL Anywhere 提供，则从 SQL Anywhere 11 的已安装副本中复制 JRE 文件。复制整个树，包括子目录。

sqlanywhere.jpr 文件包含 Sybase Central 的 SQL Anywhere 插件的 jar 文件列表。

mobilink.jpr 文件包含 Sybase Central 的 MobiLink 插件的 jar 文件列表。

qanywhere.jpr 文件包含 Sybase Central 的 QAnywhere 插件的 jar 文件列表。部署 QAnywhere 插件时，dbinit 是必需的。有关部署数据库工具的信息，请参见“部署数据库实用程序”一节 第 1037 页。

ultralite.jpr 文件包含 Sybase Central 的 UltraLite 插件的 jar 文件列表。

需要为上表中的软件包创建几个链接。以下几节中提供了详细信息。

Mac OS X

请注意，在 Mac OS X 上，共享对象的扩展名为 *.dylib*。需要为以下 dylibs 创建 Symlink（符号链接）：

```
libdbjodbc11.jnilib -> libdbjodbc11.dylib
libdblib11_r.jnilib -> libdblib11_r.dylib
libdbput11_r.jnilib -> libdbput11_r.dylib
libmljodbc11.jnilib -> libmljodbc11.dylib
```

Linux/Solaris 基组件文件

所有软件包都需要本节中列出的链接。

在 */opt/sqlanywhere11/lib32* 中创建以下符号链接：

```
libdbicu11_r.so -> libdbicu11_r.so.1
libdbicudt11.so -> libdbicudt11.so.1
libdbjodbc11.so -> libdbjodbc11.so.1
libjsyblib600_r.so -> libjsyblib600_r.so.1
libdbodbc11_r.so -> libdbodbc11_r.so.1
libdbodm11.so -> libdbodm11.so.1
libdbtasks11_r.so -> libdbtasks11_r.so.1
```

在 */opt/sqlanywhere11/sun* 中创建符号链接。用于 Linux 的符号链接是 32 位 JRE 的 *jre160_x86*。用于其它系统的符号链接是 *jre_160*。

```
jre160_x86 -> /opt/sqlanywhere11/sun/jre_1.6.0_linux_sun_i586 (Linux)
jre160 -> /opt/sqlanywhere11/sun/jre_1.6.0_solaris_sun_sparc (Solaris)
```

Linux/Solaris Interactive SQL 文件

在 */opt/sqlanywhere11/lib32* 中创建以下符号链接：

```
libdblib11_r.so -> libdblib11_r.so.1
libdbtool11_r.so -> libdbtool11_r.so.1
```

Linux/Solaris 带有 SQL Anywhere 插件的 Sybase Central

在 */opt/sqlanywhere11/lib32* 中创建以下符号链接：

```
libdblib11_r.so -> libdblib11_r.so.1
libdbput11_r.so -> libdbput11_r.so.1
libdbtool11_r.so -> libdbtool11_r.so.1
```

Linux/Solaris 带有 MobiLink 插件的 Sybase Central

在 */opt/sqlanywhere11/lib32* 中创建以下符号链接：

```
libdblib11_r.so -> libdblib11_r.so.1
libdbmlput11_r.so -> libdbmlput11_r.so.1
libdbtool11_r.so -> libdbtool11_r.so.1
```

对于 64 位 Linux，在 */opt/sqlanywhere11/sun* 中创建附加符号链接。用于 Linux 的符号链接是 64 位 JRE 的 *jre160_x64*。

```
jre160_x64 -> /opt/sqlanywhere11/sun/jre_1.6.0_linux_sun_x64 (Linux)
```


Linux/Solaris 带有 QAnywhere 插件的 Sybase Central

在 */opt/sqlanywhere11/lib32* 中创建以下符号链接：

```
libdblib11_r.so -> libdblib11_r.so.1
```

Linux/Solaris SQL Anywhere 控制台

在 */opt/sqlanywhere11/lib32* 中创建以下符号链接：

```
libdblib11_r.so -> libdblib11_r.so.1
```

国际消息和上下文相关帮助文件

仅对于 Linux 系统，管理工具的所有显示文本和上下文相关帮助都已从英语翻译为德语、法语、日语和简体中文。每种语言的资源位于不同的文件中。英语文件的文件名中包含 **en**。德语文件名包含 **de**，法语文件名包含 **fr**，日语文件名包含 **ja**，而中文文件名包含 **zh**。

如果要安装对不同语言的支持，必须添加其它语言的消息文件。翻译后的文件如下所示：

dblgen11.res	英语
dblgde11_iso_1.res、*dblgde11_utf8.res*	德语（仅限 Linux）
dblgja11_eucjis.res、*dblgja11_sjis.res* 、*dblgja11_utf8.res*	日语（仅限 Linux）
dblgzh11_cp936.res、*dblgzh11_eucgb.res*、*dblgzh11_utf8.res*	简体中文（仅限 Linux）

还必须添加其它语言的上下文相关帮助文件。可用的翻译后文件如下所示：

scvwen600.jar	英语
scvwde600.jar	德语
scvwfr600.jar	法语
scvwja600.jar	日语
scvwzh600.jar	简体中文

这些文件包含在 SQL Anywhere 的本地化版本中。

第 3 步：设置环境变量

要运行管理工具，必须定义或修改几个环境变量。此操作通常在 SQL Anywhere 安装程序创建的 *sa_config.sh* 文件中完成，但也可以灵活地通过最适合您的应用程序的方式完成。

1. 设置 PATH 包含下面的内容：

   ```
   /opt/sqlanywhere11/bin32
   ```

   （选择适合需要的）。

2. 设置 LD_LIBRARY_PATH 以包含下面的内容：

   对于 Linux：

   ```
   /opt/sqlanywhere11/jre_1.6.0_linux_sun_i586/lib/i386/client
   /opt/sqlanywhere11/jre_1.6.0_linux_sun_i586/lib/i386
   /opt/sqlanywhere11/jre_1.6.0_linux_sun_i586/lib/i386/native_threads
   ```

   对于 Solaris：

   ```
   /opt/sqlanywhere11/jre_1.6.0_solaris_sun_sparc/lib/sparc/client
   /opt/sqlanywhere11/jre_1.6.0_solaris_sun_sparc/lib/sparc
   /opt/sqlanywhere11/jre_1.6.0_solaris_sun_sparc/lib/sparc/native_threads
   ```

3. 设置以下环境变量：

   ```
   SQLANY11="/opt/sqlanywhere11"
   ```

第 4 步：注册 Sybase Central 插件

此步骤涉及 Sybase Central 的配置。如果不安装 Sybase Central，则可以跳过此步骤。

Sybase Central 需要一个列出已安装插件的配置文件。安装程序必须创建此文件。请注意，它所包含的几个 JAR 文件的完整路径可能会依据软件的安装位置而更改。

文件名为 *.scRepository600*。对于多数 Linux 和 Unix 系统，它位于 */opt/sqlanywhere11/bin32* 目录中。对于 Mac OS X，它位于 */opt/sqlanywhere11/bin64* 目录中。此文件是一个纯文本文件，包含 Sybase Central 应装载的插件的一些基本信息。

SQL Anywhere 的提供程序信息是使用以下命令在存储库文件中创建的。

```
scjview -register "/opt/sqlanywhere11/java/sqlanywhere.jpr"
```

sqlanywhere.jpr 文件的内容如下所示（为便于显示，一些条目已拆分为多行）。**AdditionalClasspath** 行必须在 *.jpr* 文件的一个单独的行上输入。

```
PluginName=SQL Anywhere 11
PluginId=sqlanywhere1100
PluginClass=ianywhere.sa.plugin.SAPlugin
PluginFile=\_opt\_sqlanywhere11\_java\_saplugin.jar
AdditionalClasspath=\_opt\_sqlanywhere11\_java\_isql.jar:
    \_opt\_sqlanywhere11\_java\_salib.jar:
    \_opt\_sqlanywhere11\_java\_JComponents1101.jar:
    \_opt\_sqlanywhere11\_java\_jlogon.jar:
    \_opt\_sqlanywhere11\_java\_debugger.jar:
    \_opt\_sqlanywhere11\_java\_jodbc.jar
ClassloaderId=SA1100
```

最初安装 SQL Anywhere 时，在 SQL Anywhere 安装目录的 *java* 文件夹中创建 *sqlanywhere.jpr* 文件。此文件用作必须作为安装过程一部分而创建的 *.jpr* 文件的模型。此文件还有用于 MobiLink 和 QAnywhere 的版本，其名称分别为 *mobilink.jpr* 和 *qanywhere.jpr*。它们也都位于 *java* 文件夹中。

下面是按照上述过程创建的示例 *.scRepository600* 文件。为便于显示，一些条目已拆分为多行。在此文件中，每个条目出现在一行中：

```
#  Version: 6.0.0.1154
#  Fri Feb 23 13:09:14 EST 2007
```



```
#
SCRepositoryInfo/Version=4
#
Providers/sqlanywhere1100/Version=11.0.1.1297
Providers/sqlanywhere1100/UseClassloader=true
Providers/sqlanywhere1100/ClassloaderId=SA1100
Providers/sqlanywhere1100/Classpath=
    \_opt\_sqlanywhere11\_java\_saplugin.jar
Providers/sqlanywhere1100/Name=SQL Anywhere 11
Providers/sqlanywhere1100/AdditionalClasspath=
    \_opt\_sqlanywhere11\_java\_isql.jar:
    \_opt\_sqlanywhere11\_java\_salib.jar:
    \_opt\_sqlanywhere11\_java\_JComponents1101.jar:
    \_opt\_sqlanywhere11\_java\_jlogon.jar:
    \_opt\_sqlanywhere11\_java\_debugger.jar:
    \_opt\_sqlanywhere11\_java\_jodbc.jar
Providers/sqlanywhere1100/Provider=ianywhere.sa.plugin.SAPlugin
Providers/sqlanywhere1100/ProviderId=sqlanywhere1100
Providers/sqlanywhere1100/InitialLoadOrder=0
#
```

注意

- 您的安装程序应使用上述方法创建一个与此类似的文件。唯一需要更改的是 Classpath 和 AdditionalClasspath 行中 JAR 文件的完全限定路径。
- 上面显示的 AdditionalClasspath 行经换行占用了其它行。在 *.scRepository600* 文件中，它们必须单独位于一行中。
- 在 *.scRepository600* 文件中，正斜线字符 (/) 用转义序列 _ 表示。
- 第一行指示 *.scRepository600* 文件的版本。
- 以 # 开头的行是注释。

有关部署数据库和数据库应用程序的详细信息，请参见“部署数据库和应用程序”第 975 页。

第 5 步：创建 Sybase Central 的连接配置文件

此步骤涉及 Sybase Central 的配置。如果不安装 Sybase Central，则可以跳过此步骤。

将 Sybase Central 安装到系统时，会在 *.scRepository600* 文件中创建 **SQL Anywhere 11 Demo** 的连接配置文件。如果您不想创建一个或多个连接配置文件，则可以跳过此步骤。

以下命令用于创建 **SQL Anywhere 11 Demo** 连接配置文件。将此作为您创建自己的连接配置文件的模型。

```
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Name" "SQL Anywhere
11 Demo"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/FirstTimeStart"
"false"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Description"
"Suitable Description"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/ProviderId"
"sqlanywhere1100"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Provider" "SQL
Anywhere 11"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Data/
ConnectionProfileSettings" "DSN\eSQL^0020Anywhere^002011^0020Demo;UID
```

```
\eDBA;PWD\e35c624d517fb"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Data/
ConnectionProfileName" "SQL Anywhere 11 Demo"
scjview -write "ConnectionProfiles/SQL Anywhere 11 Demo/Data/
ConnectionProfileType" "SQL Anywhere"
```

连接配置文件字符串和值可从 *.scRepository600* 文件中抽取。使用 Sybase Central 定义连接配置文件，然后在 *.scRepository600* 文件中查找相对应的行。

下面是按照上述过程创建的 *.scRepository600* 文件的一部分。为便于显示，一些条目已拆分为多行。在此文件中，每个条目出现在一行中：

```
#  Version: 6.0.0.1154
#  Fri Feb 23 13:09:14 EST 2007
#
ConnectionProfiles/SQL Anywhere 11 Demo/Name=SQL Anywhere 11 Demo
ConnectionProfiles/SQL Anywhere 11 Demo/FirstTimeStart=false
ConnectionProfiles/SQL Anywhere 11 Demo/Description=Suitable Description
ConnectionProfiles/SQL Anywhere 11 Demo/ProviderId=sqlanywhere1100
ConnectionProfiles/SQL Anywhere 11 Demo/Provider=SQL Anywhere 11
ConnectionProfiles/SQL Anywhere 11 Demo/Data/ConnectionProfileSettings=
    DSN\eSQL^0020Anywhere^002011^0020Demo;
    UID\eDBA;
    PWD\e35c624d517fb
ConnectionProfiles/SQL Anywhere 11 Demo/Data/ConnectionProfileName=
    SQL Anywhere 11 Demo
ConnectionProfiles/SQL Anywhere 11 Demo/Data/ConnectionProfileType=
     SQL Anywhere
```

配置管理工具

您可以控制管理工具显示或启用的功能。通过名为 *OEM.ini* 的初始化文件进行控制。此文件必须与管理工具使用的 JAR 文件位于同一目录中（例如，*C:\Program Files\SQL Anywhere 11\java*）。如果未找到此文件，则使用缺省值。此外，缺省值将用作 *OEM.ini* 中缺少的值。

以下是示例 *OEM.ini* 文件：

```
[errors]
# reportErrors type is boolean, default = true
reportErrors=true

[updates]
# checkForUpdates type is boolean, default = true
checkForUpdates=true

[dbisql]
disableExecuteAll=false
# lockedPreferences is assigned a comma-separated
# list of one or more of the following option names:
#    autoCommit
#    autoRefetch
#    commitOnExit
#    disableResultsEditing
#    executeToolbarButtonSemantics
#    fastLauncherEnabled
#    maximumDisplayedRows
#    showMultipleResultSets
#    showResultsForAllStatements
lockedPreferences=showMultipleResultSets,commitOnExit
```


所有以 # 字符开始的行都是注释行，将被忽略。指定的选项名称和值区分大小写。

如果 **reportErrors** 为 false，则管理工具不会显示软件崩溃时用于向 iAnywhere 提交错误信息的窗口。相反，将出现标准窗口。

如果 **checkForUpdates** 为 false，则管理工具不会自动检查 SQL Anywhere 软件更新，也不会为用户提供选项让其自行决定是否检查 SQL Anywhere 软件更新。

如果 **disableExecuteAll** 为 true，则 Interactive SQL 中的 [**SQL**] » [**执行**] 菜单项和 F5 加速键将被禁用。如果已将 [执行] 工具栏按钮配置为 [**执行所有语句**]，则该按钮也将被禁用。因此，最好在 Interactive SQL 中将 [执行] 工具栏按钮设置为 [**执行所选语句**]，然后在 *OEM.ini* 文件中设置 executeToolbarButtonSemantics 选项，以防止用户更改 [执行] 工具栏按钮。请参见“配置执行语句工具栏按钮”一节 《SQL Anywhere 服务器 - 数据库管理》。

防止用户更改 Interactive SQL 选项设置

在 *OEM.ini* 文件的 **[dbisql]** 部分，您可以锁定 Interactive SQL 选项设置以使用户无法对其进行更改。选项名称区分大小写。对于部分选项，您可以指定选项是针对 SQL Anywhere 数据库还是针对 UltraLite 数据库进行锁定。如果不指定数据库类型，则设置将针对所有数据库锁定。以下是一个示例：

```
[dbisql]
lockedPreferences=autoCommit
```

要只对一种类型的数据库锁定选项设置，可在 **lockedPreferences** 前加上数据库类型的名称（**SQLAnywhere** 或 **UltraLite**），并在其间以句点分隔。

例如，如果要针对 SQL Anywhere 数据库锁定 **autoCommit**，但不对 UltraLite 锁定，则可添加以下行：

```
[dbisql]
SQLAnywhere.lockedPreferences=autoCommit
```

您可以阻止用户更改以下 Interactive SQL 选项设置（**SQLAnywhere/UltraLite** 表示这些选项可以针对某一特定的数据库类型锁定）：

- **autoCommit (SQLAnywhere/UltraLite)** 阻止用户自定义 [**每条语句后提交**] 选项。请参见“auto_commit 选项 [Interactive SQL]”一节 《SQL Anywhere 服务器 - 数据库管理》。
- **autoRefetch (SQLAnywhere/UltraLite)** 阻止用户自定义 [**自动重新读取结果**] 选项。请参见“auto_refetch 选项 [Interactive SQL]”一节 《SQL Anywhere 服务器 - 数据库管理》。
- **commitOnExit (SQLAnywhere/UltraLite)** 阻止用户自定义 [**退出或断开连接时提交**] 选项。请参见“commit_on_exit 选项 [Interactive SQL]”一节 《SQL Anywhere 服务器 - 数据库管理》。
- **disableResultsEditing (SQLAnywhere/UltraLite)** 阻止用户自定义 [**禁用编辑**] 选项。
- **executeToolbarButtonSemantics** 阻止用户自定义 [**执行**] 工具栏按钮的行为。请参见“配置执行语句工具栏按钮”一节 《SQL Anywhere 服务器 - 数据库管理》。
- **fastLauncherEnabled** 阻止用户自定义快速启动程序选项。

- **maximumDisplayedRows (SQLAnywhere/UltraLite)** 阻止用户自定义 [**要显示的最大行数**] 选项。“isql_maximum_displayed_rows 选项 [Interactive SQL]”一节 《SQL Anywhere 服务器 - 数据库管理》.
- **showMultipleResultSets (SQLAnywhere/UltraLite)** 阻止用户自定义 [**仅显示第一个结果集**] 或 [**显示所有结果集**] 选项。请参见“isql_show_multiple_result_sets [Interactive SQL]”一节 《SQL Anywhere 服务器 - 数据库管理》。
- **showResultsForAllStatements (SQLAnywhere/UltraLite)** 阻止用户自定义 [**显示最后一条语句的结果**] 或 [**显示每条语句的结果**] 选项。

部署 dbisqlc

如果客户应用程序运行在资源有限的计算机上，则您可能想要部署 dbisqlc 可执行文件而不是 Interactive SQL (dbisql)。但是您应该注意，dbisqlc 未包含 Interactive SQL 的全部功能，并且不保证两者之间的兼容性。此外，不建议使用 dbisqlc，因为未向其添加任何新功能；目前没有计划要将它从产品中删除。

dbisqlc 可执行文件需要标准的嵌入式 SQL 客户端库。

有关 dbisqlc 的详细信息，请参见“dbisqlc 实用程序（不建议使用）”一节 《SQL Anywhere 服务器 - 数据库管理》。

有关 Interactive SQL (dbisql) 的详细信息，请参见“Interactive SQL 实用程序 (dbisql)”一节 《SQL Anywhere 服务器 - 数据库管理》。

部署数据库服务器

您可以通过为最终用户提供 SQL Anywhere 安装程序来部署数据库服务器。通过选择正确的选项，保证每个最终用户获得他们所需的文件。

部署个人数据库服务器或网络数据库服务器最简单的方法是使用 [**部署向导**]。有关详细信息，请参见“使用 [部署向导]”一节第 981 页。

要运行数据库服务器，您需要安装一组文件。下表列出了这些文件。这些文件的所有重新分发由许可协议的条款控制。重新分发数据库服务器文件之前，您必须确认自己是否具有执行此操作的权限。

Windows	Linux/Unix	Mac OS X
dbeng11.exe	*dbeng11*	*dbeng11*
dbeng11.lic	*dbeng11.lic*	*dbeng11.lic*
dbsrv11.exe	*dbsrv11*	*dbsrv11*
dbsrv11.lic	*dbsrv11.lic*	*dbsrv11.lic*
dbserv11.dll	*libdbserv11_r.so*、*libdbtasks11_r.so*	*libdbserv11_r.dylib*、*libdbtasks11_r.dylib*
dbscript11.dll	*libdbscript11_r.so*	*libdbscript11_r.dylib*
dblg[xx]11.dll	*dblg[xx]11.res*	*dblgen11.res*
dbghelp.dll	N/A	N/A
dbctrs11.dll	N/A	N/A
dbextf.dll [1]	*libdbextf.so* [1]	*libdbextf.dylib* [1]
dbicu11.dll[2]	*libdbicu11_r.so*[2]	*libdbicu11_r.dylib*[2]
dbicudt11.dll[2] [3]	*libdbicudt11.so*[2]	*libdbicudt11.dylib*[2]
sqlany.cvf	*sqlany.cvf*	*sqlany.cvf*
dbrsakp11.dll[4]	*libdbrsakp11_r.so*[4]	*libdbrsakp11_r.dylib*[4]
dbodbc11.dll [5]	*libdbodbc11.so* [5]	*libdbodbc11.dylib* [5]
N/A	*libdbodbc11_n.so* [5]	*libdbodbc11_n.dylib* [5]
N/A	*libdbodbc11_r.so* [5]	*libdbodbc11_r.dylib* [5]
dbjodbc11.dll [5]	*libdbjodbc11.so* [5]	*libdbjodbc11.dylib* [5]

Windows	Linux/Unix	Mac OS X
java\jconn3.jar [5]	*java/jconn3.jar* [5]	*java/jconn3.jar* [5]
java\jodbc.jar [5]	*java/jodbc.jar* [5]	*java/jodbc.jar* [5]
java\sajvm.jar [5]	*java/sajvm.jar* [5]	*java/sajvm.jar* [5]
java\cis.zip [6]	*java/cis.zip* [6]	*java/cis.zip* [6]
dbcis11.dll[7]	*libdbcis11.so*[7]	*libdbcis11.dylib*[7]
libsybbr.dll[8]	*libsybbr.so*[8]	*libsybbr.dylib*[8]

[1] 只有当使用系统扩展存储过程和函数 (xp_) 时才需要。

[2] 只有当数据库字符集为多字节或使用 UCA 归类序列时才需要。

[3] 在 Windows Mobile 上，要部署的文件名为 *dbicudt11.dat*。

[4] 仅为加密 TDS 连接所必需。

[5] 仅在数据库中使用 Java 时才需要。

[6] 仅在数据库和远程数据访问中使用 Java 时才需要。

[7] 仅在使用远程数据访问时才需要。

[8] 仅为档案备份所必需。

注意

- 您应根据自己的情况选择是部署个人数据库服务器 (dbeng11) 还是网络数据库服务器 (dbsrv11)。
- 部署数据库服务器时，您必须包括单独的对应许可文件（*dbeng11.lic* 或 *dbsrv11.lic*）。许可文件与服务器可执行文件位于相同的目录中。
- 上表显示了带有标志 **[xx]** 的文件。这表示消息文件有多个，每个消息文件支持不同的语言。如果要安装对不同语言的支持，必须添加这些语言的资源文件。
- 只有数据库服务器要在数据库功能中使用 Java 时才需要 Java VM jar 文件 (*sajvm.jar*)。
- 此表不包括运行实用程序（例如 dbbackup）所需的文件。

  有关部署实用程序的信息，请参见“部署管理工具”一节第 1005 页。

Windows 注册表条目

在 Windows 上，要确保服务器写入事件日志的消息的格式是正确的，请创建以下注册表项。

```
HKEY_LOCAL_MACHINE\
   SYSTEM\
      CurrentControlSet\
         Services\
            Eventlog\
               Application\
                  SQLANY 11.0
```


在该项中，添加一个名为 EventMessageFile 的 REG_SZ 值，并为其指派 *dblgen11.dll* 完全限定位置的数据值，例如 *C:\Program Files\SQL Anywhere 11\bin32\dblgen11.dll*。可以不考虑部署语言而指定英语 DLL *dblgen11.dll*。下面是注册表文件更改示例。

```
REGEDIT4
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog\Application
\SQLANY 11.0]
"EventMessageFile"="c:\\sa11\\bin32\\dblgen11.dll"
```

对于 64 位版本的服务器，注册表项为 `SQLANY64 11.0`。

在 Windows 上，要确保由 MESSAGE ... TO EVENT LOG 语句写入到事件日志的消息格式正确，请创建以下注册表项。

```
HKEY_LOCAL_MACHINE\
   SYSTEM\
      CurrentControlSet\
         Services\
            Eventlog\
               Application\
                  SQLANY 11.0 Admin
```

在该项中，添加一个名为 EventMessageFile 的 REG_SZ 值，并为其指派 *dblgen11.dll* 完全限定位置的数据值，例如 *C:\Program Files\SQL Anywhere 11\bin32\dblgen11.dll*。可以不考虑部署语言而指定英语 DLL *dblgen11.dll*。下面是注册表文件更改示例。

```
REGEDIT4
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog\Application
\SQLANY 11.0 Admin]
"EventMessageFile"="c:\\sa11\\bin32\\dblgen11.dll"
```

对于 64 位版本的服务器，注册表项为 `SQLANY64 11.0 Admin`。

可以通过设置注册表项取消 Windows 事件日志条目。该注册表项是：

```
Software\Sybase\SQL Anywhere\11.0\EventLogMask
```

可将其置于 HKEY_CURRENT_USER 或 HKEY_LOCAL_MACHINE 配置单元中。要控制事件日志条目，请创建名为 EventLogMask 的 REG_DWORD 值，并为其指派一个位掩码（其中包含不同 Windows 事件类型的内部位值）。SQL Anywhere 数据库服务器支持的三种类型为：

```
EVENTLOG_ERROR_TYPE         0x0001
EVENTLOG_WARNING_TYPE       0x0002
EVENTLOG_INFORMATION_TYPE   0x0004
```

例如，如果将 EventLogMask 项设置为 0，则根本不显示消息。最好设置为 1，这样可以不显示信息性消息和警告消息，但显示错误消息。缺省设置（没有输入项）是显示所有消息。下面是注册表文件更改示例。

```
REGEDIT4
[HKEY_LOCAL_MACHINE\SOFTWARE\Sybase\SQL Anywhere\11.0]
"EventLogMask"=dword:00000007
```


在 Windows 上注册 DLL

部署 SQL Anywhere 时，为使 SQL Anywhere 正常工作，有一些 DLL 文件必须进行注册。注意，对于 Windows Vista 或更高版本的 Windows，必须包含 SQL Anywhere 已提升的操作代理 (*dbelevate11.exe*) 以支持注册或注销 DLL 时所需的权限提升。

注册这些 DLL 的方法有许多，其中包括在安装脚本中进行注册或使用 Windows 上的 regsvr32 实用程序或 Windows Mobile 上的 regsvrce 实用程序进行注册。还可在批处理文件中包括一个命令（例如以下过程中的命令）。

♦ 注册 DLL

1. 打开命令提示符。
2. 转到安装 DLL 提供程序的目录。
3. 输入以下命令以注册提供程序（在此例中，注册 OLE DB 提供程序）：

```
regsvr32 dboledb11.dll
```

下表列出了部署 SQL Anywhere 时必须注册的 DLL：

文件	说明
dbctrs11.dll	SQL Anywhere 性能监控器计数器。
dbmlsynccom.dll	Dbmlsync 集成组件（非可视化组件）。
dbmlsynccomg.dll	Dbmlsync 集成组件（可视化组件）。
dbodbc11.dll	SQL Anywhere ODBC 驱动程序。
dboledb11.dll	SQL Anywhere OLE DB 提供程序。
dboledba11.dll	SQL Anywhere OLE DB 提供程序模式辅助模块。
Windows\system32\msxml4.dll	Microsoft XML 分析程序。

部署数据库

通过将数据库文件安装到最终用户的磁盘上来部署数据库文件。

只要数据库服务器完全关闭，您就无需用数据库文件部署事务日志文件。当最终用户开始运行数据库时，即创建一个新事务日志。

对于 SQL Remote 应用程序，数据库必须在正确的同步状态下创建，在此情况下无需事务日志。您可以使用抽取实用程序实现此目的。

有关抽取数据库的信息，请参见“抽取远程数据库”一节 《SQL Remote》。

国际化注意事项

在全球范围部署数据库时，必须考虑要使用数据库的地区。不同地区可能会有不同的排序顺序或文本比较规则。例如，待部署的数据库可能是使用 1252LATIN1 归类创建的，而这可能不适合未来要使用数据库的某些环境。

由于数据库的归类在其创建后就无法更改，您可以考虑在安装阶段创建数据库，然后以所需的模式和数据填充数据库。可以在安装期间使用 dbinit 实用程序创建数据库，或以实用程序数据库启动数据库服务器然后发出 CREATE DATABASE 语句创建数据库。随后，就可以使用 SQL 语句创建模式并进行任何其它设置初始数据库所必需的操作。

如果您决定使用 UCA 归类，则可以使用附加的归类定制选项，以便能够使用 dbinit 实用程序或 CREATE DATABASE 语句对字符排序和比较进行更精细的控制。这些选项采用 *keyword=value* 对的形式，包括在圆括号内，位于归类名后。例如，使用 CREATE DATABASE 语句时，可按照以下语法指定归类定制：

```
CHAR COLLATION 'UCA( locale=es;case=respect;accent=respect )'
```

或者，可以创建多个数据库模板，每个模板对应一个数据库要在其中使用的地区。在要部署数据库的地区数目相对较少时这种方法可能适合。您可以让安装程序选择要安装的数据库。

另请参见

- “CREATE DATABASE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “初始化实用程序 (dbinit)”一节 《SQL Anywhere 服务器 - 数据库管理》
- “选择归类”一节 《SQL Anywhere 服务器 - 数据库管理》

在只读介质上部署数据库

只要您在只读模式下运行数据库，就可以在只读介质（如 CD-ROM）上分发数据库。

有关在只读模式下运行数据库的详细信息，请参见“-r 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

如果需要对数据库进行更改，则必须将数据库从 CD-ROM 复制到可对其进行修改的位置（例如硬盘）。

部署外部环境支持

下表总结了为支持 SQL Anywhere 中的外部调用而必须部署的组件。

ESQL/ODBC 外部调用

组件	Windows	Linux/Unix	Mac OS X
ESQL 和 ODBC 启动程序	*dbexternc11.exe*	*dbexternc11*	*dbexternc11*
Bridge	*dbextenv11.dll*	*libdbextenv11_r.so*	*libdbextenv11_r.dylib*
SQL Anywhere C API 运行时库	*dbcapi.dll*	*libdbcapi_r.so*	*libdbcapi_r.dylib*

有关嵌入式 SQL 应用程序所需的其它文件，请参见“部署嵌入式 SQL 客户端”一节第 1002 页。

有关 ODBC 应用程序所需的其它文件，请参见“部署 ODBC 客户端”一节第 994 页。

Java 外部调用

组件	Windows	Linux/Unix	Mac OS X
Java 安装（第三方）	*java.exe*	*java*	*java*
启动程序	*sajvm.jar*	*sajvm.jar*	*sajvm.jar*
iAnywhere JDBC 驱动程序（服务器端调用）	*dbjodbc11.dll*	*libdbjodbc11.so*	*libdbjodbc11.dylib*

有关 JDBC 应用程序所需的其它文件，请参见“部署 JDBC 客户端”一节第 1003 页。

.NET CLR 外部调用

组件	Windows	Linux/Unix	Mac OS X
.NET 2.0 或更高版本	（由 Microsoft 提供）	N/A	N/A
.NET CLR Bridge	*dbextclr11.exe*	N/A	N/A
Bridge	*dbextenv11.dll*	*libdbextenv11_r.so*	*libdbextenv11_r.dylib*
.NET CLR 支持	*dbclrenv11.dll*	N/A	N/A

Perl 外部调用

组件	Windows	Linux/Unix	Mac OS X
Perl 安装（第三方）	*perl.exe*	*perl*	*perl*

组件	Windows	Linux/Unix	Mac OS X
Perl 启动程序	*perlenv.pl*	*perlenv.pl*	*perlenv.pl*
Bridge	*dbextenv11.dll*	*libdbextenv11_r.so*	*libdbextenv11_r.dylib*
SQL Anywhere C API 运行时库	*dbcapi.dll*	*libdbcapi_r.so*	*libdbcapi_r.dylib*

有关 Perl 应用程序所需的其它文件，请参见“DBD::SQLAnywhere 简介”一节第 694 页。

PHP 外部调用

组件	Windows	Linux/Unix	Mac OS X
PHP 安装（第三方）	*php.exe*	*php*	*php*
PHP 启动程序	*phpenv.php*	*phpenv.php*	*phpenv.php*
Bridge	*dbextenv11.dll*	*libdbextenv11_r.so*	*libdbextenv11_r.dylib*
PHP 5.1.x 外部调用	*php-5.1.[1-6]_sqlanywhere_extenv11.dll*	*php-5.1.[1-6]_sqlanywhere_extenv11_r.so* 或从源代码构建	从源代码构建。
PHP 5.2.x 外部调用	*php-5.2.[0-6]_sqlanywhere_extenv11.dll*	*php-5.2.[0-6]_sqlanywhere_extenv11_r.so* 或从源代码构建	从源代码构建。
SQL Anywhere C API 运行时库	*dbcapi.dll*	*libdbcapi_r.so*	*libdbcapi_r.dylib*
DBLIB（线程）	*dblib11.dll*	*libdblib11_r.so*	*libdblib11_r.dylib*

有关 PHP 应用程序所需的其它文件，请参见“SQL Anywhere PHP 模块简介”一节第 712 页。

部署安全

下表总结了 SQL Anywhere 中支持安全功能的组件。

安全组件	安全类型	所在模块	可授权
数据库加密	AES	*dbserv11.dll* *libdbserv11_r.so*	包括 [1]
数据库加密	FIPS 认可的 AES	*dbfips11.dll*	单独授权 [2]
传送层安全性	RSA	*dbrsa11.dll* *libdbrsa11.so* *libdbrsa11.dylib* *libdbrsa11_r.dylib*	包括 [1]
传输层安全性	FIPS 认可的 RSA	*dbfips11.dll*、*sbgse2.dll*	单独授权 [2]
传输层安全性	ECC	*dbecc11.dll* *libdbecc11.so*	单独授权 [2]

[1] AES 和 RSA 高度加密随附在 SQL Anywhere 中且不需要单独的许可，但是这些库未经过 FIPS 认证。

[2] 使用 ECC 或 FIPS 认证技术进行高度加密的软件必须单独订购。

部署嵌入式数据库应用程序

本节提供有关部署嵌入式数据库应用程序的信息，在这种情况下，应用程序和数据库都驻留在同一台计算机上。

嵌入式数据库应用程序包括下列项目：

- **客户端应用程序** 这包括 SQL Anywhere 客户端要求。

  有关部署客户端应用程序的信息，请参见“部署客户端应用程序”一节第 987 页。
- **数据库服务器** SQL Anywhere 个人数据库服务器。

  有关部署数据库服务器的信息，请参见“部署数据库服务器”一节第 1029 页。
- **SQL Remote** 如果您的应用程序使用 SQL Remote 复制，则必须部署 SQL Remote 消息代理。
- **数据库** 您必须部署保存着该应用程序所使用数据的数据库文件。

部署个人服务器

当您部署使用个人服务器的应用程序时，需要同时部署客户端应用程序组件和数据库服务器组件。

在客户端和服务器之间共享语言资源库 (*dblgen11.dll*)。您只需此文件的一个副本。

建议您遵循 SQL Anywhere 安装行为，并将客户端和服务器文件安装在同一目录中。

切记，如果您的应用程序在数据库中使用 Java，需提供 Java zip 文件和 Java DLL。

部署数据库实用程序

如果您需要将数据库实用程序（例如 dbbackup）和应用程序一起部署，则需要此实用程序可执行文件和下列附加文件：

说明	Windows	Linux/Unix	Mac OS X
数据库工具库	*dbtool11.dll*	*libdbtool11_r.so*、*libdbtasks11_r.so*	*libdbtool11_r.dylib*、*libdbtasks11_r.dylib*
接口库	*dblib11.dll*	*libdblib11_r.so*	*libdblib11_r.dylib*
语言资源库	*dblg[xx]11.dll*	*dblg[xx]11.res*	*dblg[xx]en11.res*
[连接] 窗口	*dbcon11.dll*		
版本 10 以前的物理存储库	*dboftsp.dll*	*libdboftsp_r.so*	N/A

注意

- 上表显示了带有标志 **[xx]** 的文件。这表示消息文件有多个，每个消息文件支持不同的语言。如果要安装对不同语言的支持，必须添加这些语言的资源文件。
- 对于 Linux 和 Unix 上的非多线程应用程序，使用 *libdbtasks11.so* 和 *libdblib11.so*。
- 对于 Mac OS X 上的非多线程应用程序，使用 *libdbtasks11.dylib* 和 *libdblib11.dylib*。
- 某些实用程序（dblog、dbtran、dberase）需要使用版本 10 以前的物理存储库，以访问版本 10.0.0 之前的日志文件。如果不准备部署这些实用程序，则无需此库。
- 使用 dbinit 实用程序创建数据库时，需要个人数据库服务器 (dbeng11)。如果在没有运行其它数据库服务器时，从本地计算机的 Sybase Central 创建数据库，也需要使用个人数据库服务器。请参见“部署数据库服务器”一节第 1029 页。
- dbunload 实用程序可能需要用到 *scripts* 目录的内容。

部署对 10.0.0 之前的数据库的卸载支持

如果在应用程序中，要将较旧版本的数据库转换为版本 11 格式，则除了需要使用数据库卸载实用程序 dbunload 外，您还需要以下附加文件：

说明	Windows	Linux/Unix	Mac OS X
对 10.0 之前的数据库的卸载支持	*dbunlspt.exe*	*dbunlspt*	*dbunlspt*
消息资源库	*dbus[xx].dll*	*dbus[xx].res*	*dbus[xx].res*

上表显示了带有标志 **[xx]** 的文件。这表示消息文件有多个，每个消息文件支持不同的语言。如果要安装对不同语言的支持，必须添加这些语言的资源文件。消息文件如下所示。

Windows 消息文件

dbusde.dll	德语
dbusen.dll	英语
dbuses.dll	西班牙语
dbusfr.dll	法语
dbusit.dll	意大利语
dbusja.dll	日语
dbusko.dll	朝鲜语

dbuslt.dll	立陶宛语
dbuspl.dll	波兰语
dbuspt.dll	葡萄牙语
dbusru.dll	俄语
dbustw.dll	繁体中文
dbusuk.dll	乌克兰语
dbuszh.dll	简体中文

Linux 消息文件

dbusde_iso_1.res、*dbusde_utf8.res*、*dbusen.res*	德语
dbusen.res	英语
dbusja_eucjis.res、*dbusja_sjis.res*、*dbusja_utf8.res*	日语
dbuszh_cp936.res、*dbuszh_eucgb.res*、*dbuszh_utf8.res*	中文

这些文件包含在 SQL Anywhere 的本地化版本中。

除这些文件外，还需要“部署数据库实用程序”一节第 1037 页中介绍的文件。

部署 SQL Remote

如果要部署 [SQL Remote 消息代理]，需要包括以下文件：

说明	Windows	Linux/Solaris	Mac OS X
消息代理	*dbremote.exe*	*dbremote*	*dbremote*
编码/解码库	*dbencod11.dll*	*libdbencod11_r.so.1*	*libdbencod11_r.dylib*
FILE 消息链接库 [1]	*dbfile11.dll*	*libdbfile11_r.so.1*	*libdbfile11_r.dylib*
FTP 消息链接库 [1]	*dbftp11.dll*	*libdbftp11_r.so.1*	*libdbftp11_r.dylib*
语言资源库	*dblg[xx]11.dll*	*dblg[xx]11.res*	*dblg[xx]11.res*

说明	Windows	Linux/Solaris	Mac OS X
接口库	*dblib11.dll*	*libdblib11_r.so.1*	*libdblib11_r.dylib*
SMTP 消息链接库 [1]	*dbsmtp11.dll*	*libdbsmtp11_r.so.1*	*libdbsmtp11_r.dylib*
数据库工具库	*dbtool11.dll*	*libdbtool11_r.so.1*	*libdbtool11_r.dylib*
线程支持库	N/A	*libdbtasks11_r.so.1*	*libdbtasks11_r.dylib*

[1] 只针对要使用的消息链接部署该库。

上表显示了带有标志 **[xx]** 的文件。这表示消息文件有多个，每个消息文件支持不同的语言。如果要安装对不同语言的支持，必须添加这些语言的资源文件。

建议您遵循 SQL Anywhere 安装行为，并将 SQL Remote 文件与 SQL Anywhere 文件安装在同一个目录中。

术语表

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 1060 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 1070 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 1070 页
- “代码页”一节第 1045 页
- “归类”一节第 1049 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 1057 页
- “隔离级别”一节第 1048 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 1070 页
- “外键”一节第 1062 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 1061 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 1060 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 1047 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 1049 页
- “语句级触发器”一节第 1067 页
- “完整性”一节第 1063 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 1068 页

代理 ID

另请参见： “客户端消息存储库 ID”一节第 1053 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 1070 页
- “编码”一节第 1043 页
- “归类”一节第 1049 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 1059 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 1060 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 1061 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 1047 页
- “项目”一节第 1065 页
- “发布更新”一节第 1046 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 1047 页
- “发布”一节第 1046 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 1047 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 1047 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 1054 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 1054 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 1047 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 1047 页
- “发布更新”一节第 1046 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 1053 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 1059 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 1070 页
- “代码页”一节第 1045 页
- “编码”一节第 1043 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 1045 页
- “语句级触发器”一节第 1067 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 1057 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 1050 页
- “jConnect”一节第 1050 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 1050 页
- “iAnywhere JDBC 驱动程序”一节第 1049 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 1050 页
- “iAnywhere JDBC 驱动程序”一节第 1049 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 1053 页
- “视图”一节第 1057 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 1047 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 1069 页
- “外键约束”一节第 1063 页
- “主键约束”一节第 1070 页
- “唯一约束”一节第 1064 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 1057 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 1058 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 1062 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 1053 页
- “全局临时表”一节第 1056 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 1048 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 1053 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 1047 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 1053 页
- “生成的连接条件”一节第 1058 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 1052 页
- “全局临时表”一节第 1056 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 1048 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 1047 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 1061 页
- “同步”一节第 1061 页
- “UltraLite”一节第 1062 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 1053 页
- “外连接”一节第 1063 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 1047 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 1053 页
- “局部临时表”一节第 1052 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 1057 页
- “事务日志镜像”一节第 1058 页
- “完全备份”一节第 1063 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 1047 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 1050 页
- “视图”一节第 1057 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 1051 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 1057 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 1053 页
- “连接条件”一节第 1053 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 1067 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 1062 页
- “主键”一节第 1070 页
- “数据库管理系统 (DBMS)”一节第 1059 页
- “关系数据库管理系统 (RDBMS)”一节第 1048 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 1048 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 1060 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 1059 页
- “SYS”一节第 1061 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 1046 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 1055 页
- “SQL”一节第 1060 页
- “数据库管理系统 (DBMS)”一节第 1059 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 1048 页
- “并发”一节第 1043 页
- “完整性”一节第 1063 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 1061 页
- “复制”一节第 1047 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 1047 页
- “监听器”一节第 1051 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 1054 页
- “SQL Remote”一节第 1060 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 1047 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 1056 页
- “服务器启动的同步”一节第 1047 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 1062 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 1070 页
- “外表”一节第 1062 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 1069 页
- “检查约束”一节第 1051 页
- “主键约束”一节第 1070 页
- “唯一约束”一节第 1064 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 1053 页
- “内连接”一节第 1055 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 1069 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 1044 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 1047 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 1048 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 1063 页
- “主键约束”一节第 1070 页
- “约束”一节第 1069 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 1064 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 1051 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 1047 页
- “发布”一节第 1046 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 1052 页
- “服务器消息存储库”一节第 1047 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 1047 页
- “统一数据库”一节第 1061 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 1047 页
- “FILE”一节第 1047 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 1069 页
- “用户定义数据类型”一节第 1067 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 1062 页

用户定义数据类型

请参见“域”一节第 1067 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 1067 页
- “游标位置”一节第 1067 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 1067 页
- “游标位置”一节第 1067 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 1067 页
- “游标结果集”一节第 1067 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 1045 页
- “行级触发器”一节第 1049 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 1058 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 1046 页
- “MobiLink 用户”一节第 1055 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 1055 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 1046 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 1061 页
- “统一数据库”一节第 1061 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 1051 页
- “外键约束”一节第 1063 页
- “主键约束”一节第 1070 页
- “唯一约束”一节第 1064 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 1047 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 1057 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 1061 页
- “基于 SQL 的同步”一节第 1051 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 1062 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 1069 页
- “检查约束”一节第 1051 页
- “外键约束”一节第 1063 页
- “唯一约束”一节第 1064 页
- “完整性”一节第 1063 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 1045 页
- “编码”一节第 1043 页
- “归类”一节第 1049 页

索引

其它

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

