DBMaster

SQL存储过程用户手册

SYSCOM

SYSCOM Computer Engineering Co./Corporate Headquarters

B1, 2-7F No. 115 Emei Street, Wanhua District,
Taipei City 108, Taiwan (R.O.C.)

www.dbmaker.com

www.dbmaker.com.tw/service

Document No.645049-237392/DBM54CN-M03312017-SLSP

发行日期: 2017-03-31

对于本手册中没有体现的关于产品最新功能的描述，请在安装完成SYSCOM DBMaster 软件后阅读 README.TXT 文件。

注册商标
SYSCOM，SYSCOM 图标和 DBMaster 是SYSCOM 公司的注册商标。
Microsoft, MS-DOS, Windows 和 Windows NT 是 Microsoft 公司的注册商标。
UNIX 是 The Open Group 的注册商标。
ANSI 是美国国家标准化组织的注册商标。

手册中提到的其他产品名称或许是它们各自持有者的注册商标，仅仅是为提供此信息。
SQL 是行业语言，并不为任何公司或任何组织所有。

注意事项
本手册中有关软件的描述，均以该软件所提供的使用许可为基础。

对于授权许可的详细信息，请与您的经销商联系。关于计算机产品的特殊用途的市场性与适用性，经销商不会给予任何说明和保证。因外界因素如地震、过热、过冷和潮湿而引起产品的任何损坏以及由于使用不正确的电压和不兼容的软硬件而引起的损失和损坏，经销商概不负责。

虽然该手册的内容已经过仔细核对，但错误再所难免。若手册再有改动，不另行通知。还请见谅。

目录

1 简介

欢迎使用SQL存储过程用户使用手册。*DBMaster*是一个功能强大且使用灵活的*SQL*数据库管理系统（*DBMS*），它支持交互式的结构化查询语言(SQL)，Microsoft开放式数据库连接(ODBC)标准接口，以及嵌入式的ESQL/C语言。其独特的开放式结构和ODBC界面允许您通过使用多种编程工具来自由地构建客户应用程序，或者使用现有的*ODBC*应用程序来查询数据库。

DBMaster可以很容易地从个人使用的“单用户数据库”升级到企业级的“分布式数据库”。无论您使用的是单一使用者数据库还是商用数据库，DBMaster都能提供给您最先进的安全性、完整性和可靠性管理。广泛的跨平台支持特性则在您需要作硬件升级时，提供给您最佳的调整弹性。

*DBMaster*提供出色的多媒体处理能力来存储、查询、恢复和操作各种类型的多媒体数据。*DBMaster*中的二进制大型对象*(BLOBs)*通过其较高的安全性以及溃损恢复机制确保了多媒体数据的完整性。在源应用程序中编辑独立文件时，文件对象*(FOs)*用来管理多媒体数据。

本手册包含了SQL存储过程的基本操作，并提供了系统说明以便您更好地管理数据库。用户手册的内容专为设计人员与数据库管理员编制，您仅需了解关系型数据库的工作方式即可。此外，用户还应具有在*Windows*或*UNIX*环境下操作的经验，本手册的信息同样适用于有经验的用户参考使用。

本手册展示了使用SQL存储过程维护数据库的不同命令与过程。尽管手册适用于*Windows*环境下的*DBMaster*，但是它依然具备*UNIX*平台下的性能。为了更加清晰，我们使用了统一的示例贯穿整个手册。

*SQL*是双重功效的语言。它既作为能够访问数据库的交互式工具（交互式*SQL*），同时也是应用程序用来访问数据库的数据库编程语言。

一般情况下，所有主流的RDBMS都提供各自的用户界面来使用*SQL*，例如*DBMaster*提供的是dmSQL/C。用户可以直接在该工具中输入*SQL*语法来访问和维护数据库。

DBMaster同时提供多种Java工具，详细内容请参考*其它相关文件*章节内容并选择您所需的手册。

1.1 其它相关文件

除了本书之外，我们还为您提供了其它用户手册和参考文献，帮助您更深入地了解DBMaster数据库管理系统。

- 有关DBMaster的性能和功能信息，请参考*DBMaster指南*。
- 有关设计、管理和维护DBMaster数据库的信息，请参考*数据库管理员手册*。
- 有关DBMaster管理的相关信息，请参考*服务器管理工具用户手册*。
- 有关DBMaster配置文件的信息，请参考*配置管理工具用户手册*。
- 有关DBMaster功能的相关信息，请参考*数据库管理工具用户手册*。
- 有关DCI COBOL接口的详细信息，请参考*DBMaster DCI用户手册*。
- 有关DBMaster SQL语言的语法和使用的相关信息，请参考*SQL命令与函数参考手册*。
- 有关dmSQL命令行工具的使用方法，请参考*dmSQL使用手册*。
- 有关ODBC API和JDBC API的信息，请参考*ODBC程序员参考手册*和*JDBC程序员参考手册*。
- 有关错误和警告信息的内容，请参考*错误信息参考手册*。
- 有关嵌入式ESQL/C语言的语法和使用，请参考*ESQL/C用户手册*。
- 有关Java存储过程的相关信息，请参考*Java存储过程用户手册*。

1.2 技术支持

在软件试用期间，Syscom电脑有限公司（“Syscom”）将为您提供30天的免费email支持和电话支持。当软件注册后，我们还会再为您提供30天的免费技术支持。如此一来，您就可以获得60天的免费支持。不仅如此，在您购买软件后，Syscom对任何问题都会以email的方式为您提供技术支持。

您除了可以获得免费的技术支持外，还可以以20%的零售价购买其它产品。要想获得更多的详细资料和价格信息，请与sales@dbmaker.com.tw保持联系。

您可以通过任何一种方式(普通信件、电话或email)与Syscom技术支持保持联系，请登录至：http://www.dbmaker.com.tw/service以获取详细信息。在与Syscom技术支持联系之前，请先查询当前数据库的常见问题解答。

无论您以何种方式与CASEMaker的技术支持联系时，请务必写上以下有效信息：

- 产品名称和版本号
- 注册号
- 注册的用户名和地址
- 供应商/发行者的地址
- 操作平台和计算机系统配置
- 错误发生前执行的动作
- 如果可以，请提供错误信息和编号
- 其它一些相关信息

1.3 文档协定

为方便用户的阅读和使用，本手册使用了标准的排版约定，注释、程序、示例和命令行都用缩进排版的方式进行了特别的设置。

协定	说明
斜体字	斜体字表示必须输入的信息占位符，例如用户名和表名。此字符应该用实际的名称来替换。有时，也会使用斜体字来介绍新的关键字，或强调重点。
黑体字	黑体字表示文件名、数据库名、表名称、字段名、用户名和其它数据库对象。它也用于强调程序执行步骤中的菜单命令。
关键字	文字段落中，SQL语言使用的关键字都是以大写字母出现的。
小符号	文档中出现的小写字符表示键盘上的按键，两个键名之间的加号（+）表示在按住第一个键不放的同时，再按第二个键。两个键名之间的逗号（，）表示释放第一个键以后，再按第二个键。
注意	包含一些重要的信息。
➲ 程序	表示后面跟随的是程序的执行步骤或连续的项目。很多任务都是通过这种方式描述，给用户提供一个逻辑顺序步骤得以效仿。
➲ **示例**	例子用来阐明描述，通常包括屏幕上出现的文本，用户也可以将这些例子输入到计算机中，通过屏幕看到运行结果。当然，示例还包括一些原型和语法。
命令行	包括文本，这些命令都可以输入计算机中，显示在屏幕上。通常用于显示SQL命令的输入输出或dmconfig.ini中的内容。

表 1-1文档协定

2 DBMaster存储过程概述

本章我们将对DBMaster ESQL\C、Java存储过程和SQL存储过程做一个简单的描述。

2.1 什么是存储过程

存储过程是一种保存并执行在服务器端的程序，一旦创建，就以一种可执行的格式存储在数据库中。如此一来，数据库引擎可避免重复SQL命令的编译和优化，从而提高了重复执行命令的执行效率。存储过程作为一个可执行命令，用在以下环境：交互式SQL命令、应用程序、触发器或其它存储过程。

利用存储过程可以实现很多目标，例如：提高数据库的执行性能、简化应用程序的代码、限制和监控数据库的访问等。

因为存储过程运行在DBMS中，所以它可以帮助减少应用程序的延迟。对于服务器端的操作，您无需执行四五条SQL语句，仅需执行一个存储过程即可。在减少网络往返次数的同时，对数据库性能有着很大的改善。

2.2 DBMaster支持的存储过程语言

DBMaster支持三种存储过程语言：ESQL/C, Java和SQL。

DBMaster4.3以前仅支持ESQL/C存储过程，但DBMaster 4.3及以后的版本不仅支持ESQL/C存储过程，还支持Java存储过程。DBMaster 5.2及以后的版本则支持一种新的存储过程——SQL存储过程(SQL SP)。

本书将集中介绍如何使用SQL 语言来开发一个存储过程。

ESQL/C存储过程

DBMaster ESQL/C为直接将SQL语句输入至C语言源代码中提供了方便。ESQL存储过程是一个ESQL/C程序，它可以执行任何C程序所允许的函数，包括调用其它的C函数和系统调用。因此系统必须安装一个C编译器。您可以编写C应用程序来通过ESQL命令访问DBMS，DBMaster的ESQL/C预处理程序为C编译器提供包含SQL命令的应用程序，然后预处理程序将SQL命令转换成C语句，通过C语言命令来执行数据库操作。

一个ESQL/C编写的存储过程包括**CREATE PROCEDURE**语句，必要的变量声明部分以及代码区。如果您的程序不需要使用主变量，那么变量声明部分可以省略。有关更多ESQL的信息，请参考*ESQL用户手册*的相关章节。

JAVA存储过程

在Java语言盛行的今天，开发人员对Java语言的运用程度会比ESQL更精通。DBMaster支持Java存储过程，为Java程序员利用他们更精通的语言来编写存储过程提供了方便。针对有经验的ESQL开发人员，可以利用Java语言的优势来扩展数据库应用功能，也可以重复利用现有的代码以提高效率。

每一个Java存储过程都作为一个Java方法来实现。当您调用它时，程序名和指定的参数都会通过JDBC连接而发送到数据库管理系统中，同时执行程序并将结果返回到连接。

Java存储程序是一种开放的，独立于数据库并可替代ESQL/C的程序。此外，Java存储过程使Java语言更加强大、丰富和面向对象。

有关更多Java存储过程的信息，请参考*Java存储过程用户手册*的相关章节。

SQL存储过程

通过SQL语句创建存储过程，和通过ESQL、JAVA语言创建存储过程相比，不失为一种更好的方法。

SQL存储过程是一种只通过SQL语句来逻辑实现的存储过程，一个SQL存储过程是存储在服务器端的一组SQL语句集，一旦执行SQL存储过程，客户端就不需要再保存单独的语句，而使用SQL存储过程来代替。

3 SQL存储过程基础

存储过程是一种特殊的用户自定义函数，一旦创建就以一种可执行的格式存储在数据库中。它可为快速查询转换、更新数据、产生基本报告创建简单的脚本；提高应用性能；模块化应用并改善整体数据库设计和数据库的安全性。如此一来，数据库引擎可避免重复SQL命令的编译和优化，从而提高了重复执行命令的执行效率。存储过程作为一个可执行命令，可用在以下环境：交互式SQL命令、应用程序、触发器或其它存储过程。

由于存储过程作为一个对象存储在数据库中，所以它可以被运行在数据库中的每一个应用程序所使用。几个应用程序可以使用相同的存储过程，以减少应用程序的开发时间。

在决定实现SQL存储过程之前，重要的是先了解什么是SQL存储过程，它们将如何实现。您也可以从下面列出的章节中了解更多有关SQL存储过程的知识，这样您就可以决定何时以及如何在您的数据库环境中使用它们。

- SQL存储过程优势
- 使用SQL存储过程的工具
- SQL存储过程功能
- SQL存储过程语法
- SQL存储过程使用

- 移植SQL存储过程
- SQL存储过程限制

3.1 SQL存储过程优势

SQL存储过程的很多有用程序都存储在数据库或数据库应用体系结构中。SQL存储过程可以改善数据库性能、简化应用程序的撰写、并限制或监测访问数据库。

SQL存储过程会使系统运行的更快

由于没有编译器，因此SQL存储过程不像外部语言（如ESQL/C）编写的程序运行起来那么快。但是提升速度的主要方法在于能否降低网络信息流量，如果您需要处理的是需要检查、循环、多语句但没有用户交互的重复任务，就可以使用保存在服务器上的存储过程来完成。这样在执行任务的每一步时，服务器和客户端之间就没有那么多的信息往来了。

SQL存储过程是可复用的组件

SQL SP可以跨平台，并支持Win32/64 Linux32/64系统。

如果您改变了主机语言，这将不会对存储过程产生影响，因为它采用的是数据库逻辑而不是应用程序。存储过程是可以移植的，当您使用 SQL 语言编写存储过程时，您就知道它可以运行在 DBMaster 支持的任何平台上，不需要您额外添加运行环境，也不需要为程序在操作系统中执行设置许可，或者为您不同型号的电脑配置不同的软件包。这就是与 java、ESQL/C 等外部语言相比，使用 SQL 语句的优势。

SQL存储过程将被保存

如果您编写一个程序，如显示银行事务处理中的支票撤消业务，那么想要了解支票的人就可以找到您的程序。它会以源代码的形式保存在数据库中，这将使数据和处理数据的进程关联。

SQL存储过程可以移植

DBMaster完全支持SQL 99标准，具有良好的可移植性。其它DBMS也同样支持这个标准，因此使用其它DBMS编写的代码很容易移植到DBMaster上。

SQL存储过程方便升级

SQL存储过程与目前DBMaster的ESQL在数据类型、创建、执行和删除方式上完全一样，这样就方便了现有用户的升级，而客户端程序不需要做任何改动。

SQL存储过程的其它优势

除此之外，SQL存储过程还有另外一些优势：

- 因为不使用额外的C编译器，所以不会出现由编译器影响的问题，比如安装路径、编译器卸载等。
- 使用方便，语法简单、清晰、结构性强。这个特点将在后面的章节*SQL 存储过程语法*中有所体现。
- 无需编译，创建速度快捷。

3.2 使用SQL存储过程的工具

利用开发工具能够更快更容易地创建并执行一个SQL存储过程，以下图形化工具和命令行工具都可用于创建并执行SQL存储过程。

- 数据库管理工具（JDBA ）
- dmSQL命令行工具

数据库管理工具（JDBA）

数据库管理工具是一个跨平台的图形用户界面(GUI)工具，它能帮助用户轻松地管理DBMaster中的数据库对象，是一个功能强大且使用灵活的SQL数据库管理系统。数据库管理工具隐藏了DBMS和查询语言的复杂性，并且提供了容易理解、使用方便的图形界面。用户无需学习SQL语句，就能操作数据库。数据库管理工具利用数据库的监测功能来提供了统计数据和信息。有关如何使用数据库管理工具请参考*数据库管理工具用户手册*。

dmSQL命令行工具

*dmSQL*是一个命令行工具，在开发SQL存储过程时，它可以帮助您做很多事情，包括查询、更改、载入和提取表数据、使用XML函数、执行java程序等。有关如何使用dmSQL命令行工具请参考*dmSQL用户手册*。

4 SQL存储过程功能

SQL存储过程有很丰富的功能，如下所示：

- 包含的存储过程语言语句和功能，可围绕传统的静态和动态SQL语句，支持控制流的逻辑实现。
- 容易实现，因为它们使用了一种简单、高级且功能强大的语言。
- SQL存储过程与等价的外部程序相比，可靠性更强。
- 遵循SQL99 ANSI/ISO/IEC SQL语言标准协议。
- 支持输入输出参数传递模式和以下数据类型：SMALLINT、INTEGER、BIGINT、FLOAT、DOUBLE、DECIMAL、REAL、DATE、TIME、TIMESTAMP、BINARY、CHAR、VARCHAR、NCHAR、NVARCHAR。
- 支持简单的，但功能强大的条件和错误处理模式。
- 允许您给调用者或客户端应用程序返回一个结果集。
- 您可以轻松地访问SQLSTATE和SQLCODE值作为特殊变量，但不支持为SQLSTATE和SQLCODE赋值。
- 支持Call语句调用。
- 嵌套的SQL存储过程可以调用其它语言实现的存储过程。
- 支持递归。
- 可以被触发器调用。

- 在存储过程中可以定义变量：cursor、condition、handle。
- 支持游标操作：OPEN、FETCH、CLOSE
- 对变量赋值支持SET语法。
- 支持控制流语句：IF、CASE、WHILE、LOOP、FOR、REPEAT、GOTO、RETURN。
- 支持TRACE，使用方法与ESQL一样。

SQL存储过程支持的功能不限于上面所列出的。当选择一个最佳方法来实现SQL存储过程时，它们在数据库架构、应用程序设计和数据库系统性能方面都发挥着重要的作用。

5 SQL存储过程语法

SQL存储过程语言（SQL SP）符合ANSI SQL99语言标准协议，它是一组SQL语句的集合，有关DBMaster的SQL命令可参考*SQL命令和函数参考手册*。在围绕传统的SQL查询和操作上，为实现控制流逻辑提供了必要的程序架构。

SQL存储过程的语法简单，包括变量的支持、条件语句、循环语句、转换控制语句、错误管理语句和结果集操作语句。

5.1 SQL存储过程架构

SQL存储过程由若干逻辑部分和格式组成，十分简单且容易掌握，目的在于简化程序的设计和语义。

SQL存储过程的核心是一个复合语句，该复合语句被BEGIN和END包围，下面是一个SQL存储过程语句的架构。

```
BEGIN                    #语句块头
Variable declarations
Condition declarations
Cursor declarations
Condition handler declarations
Assignment,flow of control. SQL statements and other compound statements
END;                     #语句块尾
```

这个例子显示了SQL存储过程如何通过一个或多个组件声明和语句来组成一个块，块允许在单个SQL存储过程中嵌套。组件声明的选项包括变量、条件和处理程序。然而，这些选项必须在流量控制、SQL和其它复合语句之前分配。请注意游标声明可能出现的块的任何地方。

SQL存储过程语法

以下命名规则适用于SQL存储过程中的所有参数名称和变量名称等：

- SQL存储过程名称最多为128个字符。
- SQL存储过程名称可以包括任意的字母字符、数字字符和下划线。
- 字符可以出现在任一位置。
- SQL存储过程名称不区分大小写。
- SQL存储过程名称必须唯一。

➲ 语法

SQL存储过程语法如下所示：

```
<SQL stored procedure syntax> ::=
CREATE [OR REPLACE] PROCEDURE <procedure_name>
```

```
[< procedure_parameters > [ { <comma> < procedure_parameters
> }... ]]
[RETURNS STATUS]
LANGUAGE SQL
BEGIN
  [stored_procedure_statement]
END;
<procedure_parameters> ::=
[IN | OUT | INPUT | OUTPUT] <parameter_name> <data_type>
```

上面是SQL存储过程的整体框架，一个完整的SQL存储过程必须包含完整的块头和块尾。块头用来描述SQL存储过程的名称及其参数，块尾只是该存储过程的结束标志。如：

```
CREATE [OR REPLACE] PROCEDURE project_name.module_name.sp_name
[ arg_list ]
LANGUAGE SQL
BEGIN
  ----------------->块头
  [ declare_list ]
  [ statement_list ]
  ----------------->块尾
END;
```

Arg_list是参数列表，声明过程中的输入参数和输出参数，如为空则没有参数。

```
<Arg_list> ::=
        [ Variable declarations ]
        [ Condition declarations ]
        [ Cursor declarations ]
        [ Condition handler declarations ]
```

declare_list是变量声明列表，声明过程中要用到的变量。

```
<declare_list> ::=
        [ Variable declarations ]
        [ Condition declarations ]
        [ Cursor declarations ]
        [ Condition handler declarations ]
```

statement_list是过程控制语句，主要由各种控制语句和SQL语句组成，完成过程对数据库的各种操作。

```
<statement_list> ::=
        [ sp_block ]
        [ procedure_control_statement ]
```

5.2 SQL存储过程中的参数

SQL存储过程支持的参数用于将SQL值传递进\出程序。

当执行逻辑对特殊输入或输入标值集有条件限定时，或当您需要返回一个或多个输出标值并且不希望返回一个结果集时，SQL存储过程中的参数将是十分有用的。

在设计和创建SQL存储过程时，要对其中参数的特征和限制有一个很好的理解。

- DBMaster支持对SQL存储过程中的大量输入输出参数进行选择性地使用。CREATE PROCEDURE语句程序签名部分的关键字IN/INPUT和OUT/OUTPUT用来表示参数的模式或特意用途。IN/INPUT和OUT/OUTPUT参数通过值来传递。
- 当为一个程序指定多个参数时，每一个参数都必须拥有唯一的名称。
- 如果程序内的一个声明变量与参数使用了相同的名称，那么该变量必须在程序中标记的原子嵌套块内被声明，否则DBMaster将检测引用一个含糊的名称。
- 在SQL存储过程中，参数支持的数据类型有：SMALLINT, BIGINT, INTEGER, FLOAT, DOUBLE, DECIMAL, REAL, DATE, TIME, TIMESTAMP, BINARY, CHAR, VARCHAR, NCHAR, NVARCHAR。

➲ 语法

SQL存储过程参数语法如下所示：

```
<procedure_parameters> ::=
[IN | OUT| INPUT | OUTPUT] <parameter_name> <data_type>
```

该子句是可选的。SQL 存储过程与普通函数相同，都是通过参数来传递数据。主函数可通过添加在'in/input'之后的参数将数据传递给 SQL 存储过程，SQL 存储过程也可以通过添加在'out/output'之后的参数将数据传递给主函数。

➲ **示例1**

下例名为myparams的SQL存储过程说明了IN/INPUT和OUT/OUTPUT参数模式的用法。

```
CREATE PROCEDURE myparams (IN p1 INT, OUT p2 INT, OUT p3 INT)
LANGUAGE SQL
BEGIN
  SET p2 = p1 + 1;
  SET p3 = 2 * p1;
END;
```

➲ **示例2**

空白参数的SQL存储过程如下所示：

```
CREATE PROCEDURE CRETB
LANGUAGE SQL
BEGIN
        CREATE TABLE TB_1(V1 INT,V2 BIGINT,V3 SMALLINT,V4 CHAR(10),
                          V5 VARCHAR(20),V6 FLOAT,V7 DOUBLE,
                          V8 BINARY（10）,V9 DECIMAL（20,4）,V10 TIME,
                          V11 DATE,V12 TIMESTAMP);
END;
```

存储过程CRETB的参数为空白，它的功能是创建一个名为TB_1的表。

➲ **示例3**

带有输入参数的SQL存储过程如下所示：

```
###############################################################################
#
#       Module Name = INS.SP
#       Purpose = Store Procedure testing program
#               1. Test IN parameter store procedure
#               2. Test all type which can use in Store Procedure
#       Function = 1. Create table INS
#       Use Database: "DBNAME" "SYSADM"  ""
#            table : INS(V1 int,V2 BIGINT,V3 smallint,V4 INT,
#                        V5 float,V6 DOUBLE, V7 DECIMAL(20,4),
#                        V8 binary(20), V9 CHAR (20), V10 VARCHAR (20),
#                        V11 NCHAR (40),V12 NVARCHAR (40),
```

```
#                          V13 DATE, V14 TIME, V15 TIMESTAMP,V16 REAL)
###############################################################################
#
CREATE PROCEDURE INS(IN V1 int, IN V2 BIGINT, IN V3 smallint, IN V4 INT,
                     IN V5 FLOAT, IN V6 DOUBLE,IN V7 DECIMAL(20,4),
                     IN V8 BINARY(20), IN V9 CHAR(20),
                     IN V10 VARCHAR(20),IN V11 NCHAR(40),
                     IN V12 NVARCHAR(40), IN V13 DATE,
                     IN V14 TIME, IN V15 TIMESTAMP, IN V16 REAL)
LANGUAGE SQL
BEGIN
        INSERT INTO ins VALUES(V1, V2,V3,V4,V5,V6,V7,V8,V9,V10,V11,
                               V12,V13,V14,V15,V16);
END;
```

存储过程INS的参数类型全部为INPUT，INS中的输入参数数据类型含盖了SQL存储过程所支持的所有数据类型，上例所示的功能是将输入的数据插入到已存在的INS表中。

注意 *本手册示例中以'#'开始的当前行都为注释。*

➲ **示例4**

带有输入输出参数的SQL存储过程如下所示：

```
###############################################################################
#       Module Name = DRPTB.SP
#       Purpose  = Store Procedure testing program
#                  1. Test DROP TABLE in Stored Procrdure
#       Function = 1. create table TB_1
#       Use Database : DBNAME
#        table : TB_1(V1 INT, V2 BIGINT, V3 SMALLINT, V4 CHAR(10),
#                     V5 VARCHAR(20),V6 FLOAT,V7 DOUBLE,V8 BINARY,
#                     V9 DECIMAL,V10 TIME,V11 DATE,V12 TIMESTAMP,)
###############################################################################
CREATE PROCEDURE DRPTB(IN V1 CHAR(20),OUT WORNING CHAR(40))
LANGUAGE SQL
BEGIN
        IF V1 = 'DROP TABLE' THEN
                DROP TABLE TB_1;
                SET WORNING ='NORMAL! TABLE TB_1 DROPED';
```

```
        ELSEIF V1 = 'CREATE TABLE' THEN
                CALL CRETB;
                SET WORNING ='NORMAL! CREATE A NEW TABLE NAMED TB_1';
        ELSE
                SET WORNING = 'YOU INPUT WRONG PARAMETER!';
        END IF;
END;
```

上例存储过程DRPTB中既有输入参数又有输出参数。在过程体中判断输入参数，若输入参数是'DROP TABLE'则存储过程执行DROP TABLE命令，并将输出参数设置成'NORMAL! TABLE TB_1 DROPED'。这样，可以根据输入参数来控制输出参数。

5.3 SQL存储过程中的变量

SQL存储过程中支持的本地变量允许您分配并返回支持SQL存储过程逻辑的SQL值。

SQL存储过程中的变量都通过DECLARE语句来定义，DECLARE语句用于在程序中定义本地变量项目：本地变量、条件、句柄和游标。DECLARE只允许出现在BEGIN ... END复合语句内，并且必须以此为起始，先于其它语句。声明要遵循一定的顺序：游标必须在声明句柄之前被声明，变量和条件必须在声明游标或句柄之前被声明。

➲ **语法**

SQL存储过程DECLARE变量语法如下所示：

```
<sp_declare_main> ::=
DECLARE <variable_name> [ { <comma> <variable_name> }... ] <data_type> [DEFAULT
<default_value>]
```

该子句用于声明本地变量。为了提供一个变量的默认值，包括默认子句，该值可作为表达式来指定。如果默认子句缺失，那么它可以不作为一个常量。初始值为null。

本地变量的范围在begin…end块之间声明，该变量可以被声明块内的嵌套块参照，除了那些使用相同名称声明一个变量的块。

➲ **示例1**

不通过默认值声明数据：

```
CREATE PROCEDURE DECALRES
LANGUAGE SQL
BEGIN
        declare v1 int;
        declare v2 bigint;
        declare v3 char(10);
        declare v4 varchar(10);
        declare v5 integer;
        declare v6 time;
```

```
        declare v7 date;
        declare v8 timestamp;
        declare v9 nchar(20);
        declare v10 binary(20);
        declare v11 decimal(4,8);
        declare v12 double;
        declare v13 float;
        declare v14 real;
END;
```

➲ **示例2**

通过默认值声明数据：

```
CREATE PROCEDURE DECLARES
LANGUAGE SQL
BEGIN
        declare v1 int default 50;
        declare v2 bigint default 7396;
        declare v3 char(10) default 'char';
        declare v4 varchar(10) default 'varchar';
        declare v5 integer default 20;
        declare v6 time default '11:11:11';
        declare v7 date default '2008-08-08';
        declare v8 timestamp default '2008-08-08 11:11:11';
        declare v9 nchar(20) default 'nchar';
        declare v10 binary(20) default 'binary';
        declare v11 decimal(4,8) default 1.123;
        declare v12 double default 123;
        declare v13 float default 6546;
        declare v14 real default 123.3;
END;
```

5.4 SQL存储过程中的游标

在SQL存储过程中，游标可用来定义结果集并在单行数据上执行复合逻辑，注意，结果集只是一组数据行。通过同样的方法，SQL存储过程还可以定义结果集并直接将它返回至SQL存储过程的调用者或客户端应用程序。

游标可视为一组数据行中指向一行的指针，在结果集中可以指向任一行，但是在任一指定时刻只能参照一行。

DECLARE CURSOR语句可用于定义游标，尽管交互式SQL语言提供一个界面使其看起来像是交互式执行，但这些语句只能嵌入到应用程序中。它不是一个可执行语句，不能动态准备。

➲ **在SQL存储过程中使用游标，您需要按照以下步骤执行：**

1. 声明一个游标以定义一个结果集。
2. 打开游标建立结果集。
3. 根据需要从游标获取数据到本地变量，一次一行。
4. 执行完毕关闭游标。

使用游标必须遵循以下SQL语句：

- DECLARE CURSOR
- OPEN
- FETCH
- CLOSE

➲ **语法**

DECLARE CURSOR语句语法如下所示：

```
<sp_declare_main> ::=
```

```
DECLARE <cursor_name> [[NO] SCROLL CURSOR [WITH RETURN] FOR { CALL
<procedure_name> | <select_statement> }
```

cursor_name

在源程序运行时指定创建的游标名称，该名称必须不同于声明在源程序中的其它游标名称。在使用之前，该游标必须是开启的。

[NO] SCROLL

如果DECLARE CURSOR语句的定义中没有使用SCROLL子句或是使用了NO SCROLL子句，那么FETCH语句将不能执行NEXT以外的任何操作。如果用户使用了SCROLL子句，那么FETCH语句中的所有行为都可以被使用。默认为不使用游标。

WITH RETURN

在SQL存储过程内，当SQL存储过程结束时，使用WITH RETURN子句声明的游标状态将仍然是开启的，且视为从SQL存储过程定义的结果集。当SQL存储过程结束时，SQL存储过程中所有其它开启的游标都将关闭。在一个外部存储过程（没有使用LANGUAGE SQL定义）中，所有游标默认为WITH RETURN TO CALL。因此，当存储过程结束时，所有开启的游标都将被认为是结果集。

Select_statement

标识游标的SELECT 语句。select-statement不能包括参数标签，但可以包括主变量的引用。在源程序中，主变量的声明必须先于DECLARE CURSOR语句。

call

指定游标给调用者返回一个结果集。例如，如果调用者是其它存储过程，那么结果集将返回给那个存储过程中。如果调用者是一个客户端应用程序， 结果集将返回给这个客户端应用程序。

➲ **示例1**

带有select_statements命令的游标如下所示：

```
CREATE PROCEDURE getbd_sql(IN name char(12), OUT bd DATE)
LANGUAGE SQL
```

```
BEGIN
        DECLARE cur CURSOR FOR SELECT birthday FROM birthd WHERE NAME = name;

        OPEN cur;
        FETCH cur INTO bd;
        CLOSE cur;
END;
```

➲ **示例2**

带有结果集的游标如下所示：

```
CREATE PROCEDURE t42_sql(argn TIMESTAMP)
LANGUAGE SQL
BEGIN
        DECLARE cur CURSOR WITH RETURN FOR select n,ts from t2 where ts < argn;
        OPEN cur;
END;
```

➲ **示例3**

下例演示了带有call的游标。

```
CREATE PROCEDURE call_test
LANGUAGE SQL
BEGIN
        DECLARE cur CURSOR WITH RETURN FOR select * from call_tb;
        OPEN cur;
END;
```

➲ **示例4**

下例演示了SQL存储过程call4将使用call_test来声明一个结果集。

```
CREATE PROCEDURE call4
LANGUAGE SQL
BEGIN
        DECLARE cur CURSOR WITH RETURN FOR call call_test;
        OPEN cur;
END;
```

➲ **示例5**

下例演示了SQL存储过程中只读游标的基本使用。

```
CREATE PROCEDURE sum_salaries(OUT sum INTEGER)
LANGUAGE SQL
BEGIN
        DECLARE p_sum INTEGER;
        DECLARE p_sal INTEGER;
        DECLARE c CURSOR FOR SELECT SALARY FROM EMPLOYEE;
        SET p_sum = 0;
        OPEN c;
        FETCH FROM c INTO p_sal;
        WHILE(SQLCODE = 0)
                        DO
                        SET p_sum = p_sum + p_sal;
                        FETCH FROM c INTO p_sal;
        END WHILE;
        CLOSE c;
        SET sum = p_sum;
END;
```

SQL存储过程中的FETCH语句

FETCH语句可将游标定位于结果表的下一行，并将那行的值分配给主变量。尽管交互式SQL语言提供一个界面使其看起来像是交互式执行，但这些语句只能嵌入到应用程序中。它不是一个可执行语句，不能动态准备。

➲ **语法**

FETCH语句语法如下所示：

```
<FETCH statement main> ::=
FETCH [NEXT|PRIOR|FIRST|LAST|ABSOLUTE n|RELATIVE n] FROM <cursor_name> INTO
<fetch_target_arg>
fetch_target_arg ::=
<variable_name> [ { <comma> <variable_name> }... ]
```

cursor-name

标识用于FETCH操作中的游标。cursor-name必须识别一个声明的游标。在源程序中， DECLARE CURSOR语句必须先于FETCH语句。当FETCH语句执行时，游标必须处于开启状态。

NEXT

在结果集中返回下一行，NEXT为FETCH语句的默认游标。

LAST

在结果集中，LAST命令用于将游标移动到最后一行并且返回最后一行。

FIRST

在结果集中，FIRST命令用于将游标移动到第一行并且返回第一行。

PRIOR

在结果集中，PRIOR命令用于返回上一行。

ABSOLUTE n

在结果集中，从第一行返回至第n行。

RELATIVE n

在结果集中，从当前行返回至第n行。

INTO fetch_target_arg

标识一个或多个变量，这些变量必须依照声明变量的规则来描述。结果行中的第一个值分配给列表中的第一个变量，第二个值分配给第二个变量…以此类推。

➲ **示例1**

fetch语句的基本使用。

```
CREATE PROCEDURE t43_sql(i SMALLINT, OUT ts1 TIMESTAMP)
LANGUAGE SQL
BEGIN
        DECLARE cur CURSOR FOR select TS from t2 where n = i;
        OPEN cur;
        FETCH cur INTO ts1;
```

```
        CLOSE cur;
END;
```

➲ **示例2**

下例演示了SQL存储过程中带有first、last、next、prior游标的fetch语句。

```
Table used in example procedure
dmSQL> SELECT * FROM TB_1;
        C1
====================
FIRST
NEXT
PRIOR
LAST

CREATE PROCEDURE FETCH_TEST(OUT FHTY_1 CHAR(20),OUT FHTY_2 CHAR(20),OUT FHTY_3
CHAR(20),OUT FHTY_4 CHAR(20),OUT FHTY_5 CHAR(20),OUT FHTY_6 CHAR(20))
LANGUAGE SQL
BEGIN
        DECLARE V1 CHAR(20);
        DECLARE V2 CHAR(20);
        DECLARE V3 CHAR(20);
        DECLARE V4 CHAR(20);
        DECLARE V5 CHAR(20);
        DECLARE V6 CHAR(20);
        DECLARE CUR SCROLL CURSOR FOR SELECT * FROM TB_1;
        OPEN CUR;
        FETCH FIRST FROM CUR INTO V1;
                IF V1 = 'FIRST' THEN
                        SET FHTY_1 = V1;
                ELSE
                        SET FHTY_1 = 'NULL';
                END IF;
        FETCH NEXT FROM CUR INTO V2;
                IF V2 = 'NEXT' THEN
                        SET FHTY_2 = V2;
                ELSE
                        SET FHTY_2 = 'NULL';
```

```
                END IF;
        FETCH NEXT FROM CUR INTO V3;
        FETCH PRIOR FROM CUR INTO V3;
                IF V3 = 'NEXT' THEN
                        SET FHTY_3 = V3;
                ELSE
                        SET FHTY_3 = 'NULL';
                END IF;
        FETCH RELATIVE 1 FROM CUR INTO V4;
                IF V4 = 'PRIOR' THEN
                        SET FHTY_4 = V4;
                ELSE
                        SET FHTY_4 = 'NULL';
                END IF;
        FETCH ABSOLUTE 1 FROM CUR INTO V5;
            IF V5 = 'FIRST' THEN
                        SET FHTY_5 = V5;
                ELSE
                        SET FHTY_5 = 'NULL';
                END IF;

        FETCH LAST FROM CUR INTO V6;
                IF V6 = 'LAST' THEN
                        SET FHTY_6 = V6;
                ELSE
                        SET FHTY_6 = 'NULL';
                END IF;
        CLOSE CUR;
END;
```

➲ **调用fetch_test的结果如下：**

```
dmSQL> CALL FETCH_TEST(?,?,?,?,?,?);
FHTY_1   : FIRST
FHTY_2   : NEXT
FHTY_3   : NEXT
FHTY_4   : PRIOR
FHTY_5   : FIRST
FHTY_6   : LAST
```

SQL存储过程中的DECLARE CONDITION语句

有一些条件可能需要特殊处理，这些条件可能会涉及到错误，并在程序中产生控制流。

➲ **语法**

DECLARE CONDITION语句语法如下所示：

```
<condition declaration> ::=
DECLARE <condition name> CONDITION  FOR <sqlstate value>
<sqlstate value> ::=
SQLSTATE [ VALUE ] <character string literal>
```

➲ **示例**

```
DECLARE con1 CONDITION FOR SQLSTATE '23000';
DECLARE con2 CONDITION FOR SQLSTATE VALUE '23001';
```

SQL存储过程中的DECLARE HANDLE语句

在模块或复合语句中，联合一个异常或完成条件的句柄。

用户可以重新定义DECLARE HANDLER语句，这样之前DECLARE HANDLER所定义的行为都将被重新设置。

➲ **语法**

DECLARE HANDLE语句语法如下所示：

```
<handler declaration> ::=
DECLARE <handler type> HANDLER FOR <condition value list> <handler action>
<handler type> ::=
CONTINUE
| EXIT
<handler action> ::= <SQL procedure statement>
<condition value list> ::= <condition value> [ { <comma> <condition value> }... ]
<condition value> ::=
<sqlstate value>
| <condition name>
| SQLEXCEPTION
| SQLWARNING
| NOT FOUND
```

➲ 示例1

```
create procedure sphdler
language sql
begin
        declare continue handler for sqlexception;
        drop table tb_1;
        declare exit handler for sqlexception;
        drop table tb_1;
end
```

➲ 示例2

```
DECLARE val INT;
DECLARE con1 CONDITION FOR SQLSTATE '23000';
DECLARE CONTINUE HANDLER FOR SQLSTATE '23000';
DECLARE CONTINUE HANDLER FOR con1 SET val = 100;
```

5.5 SQL存储过程中的赋值语句

赋值语句用于对SQL变量或SQL参数来赋值，可以通过SET语句和CURSOR FOR SELECT FROM语句给变量赋值。此外，变量声明时也可以设置默认值。文字、表达式、查询结果、特殊触发存器的值都可以分配给变量。变量值可以被分配给SQL存储过程参数，SQL存储过程中的其它变量，可在SQL存储过程语句日常执行的程序中作为参数被引用。

SET变量语句可将值分配给本地变量、输出参数和新的转换变量。它受事务的控制，SET赋值语法接受简单表达式和复杂表达式。

注意 *对于string类型变量赋值，赋值长度要小于1024个字节。*

➲ **示例1**

下例演示了赋值并找回变量值的方式。

```
CREATE PROCEDURE proc_vars()
LANGUAGE SQL
BEGIN
        DECLARE v_rcount INTEGER;
        DECLARE v_max DECIMAL (9,2);
        DECLARE v_adate, v_another DATE;
        DECLARE v_total INTEGER DEFAULT 0;            # (1)
        SET v_total = v_total + 1;                    # (2)
        DECLARE CUR CURSOR FOR SELECT * FROM TB_1;    # (3)
END;
```

当声明一个变量时，您可以使用DEFAULT子句指定一个默认值，如(1)行所示；(2)行演示了SET语句可用于指定一个单一变量值；(3)行演示了您可以使用CURSOR FOR SELECT FROM语句指定一个值。

注意 *DECLARE语句必须出现在SET定义之前，否则预处理程序编译出错。*

➲ **示例 2**

下例演示了如何在SET语句中赋值并找回变量值 。在SET赋值语句中，对　于超过double和integer数据类型表示范围的数值，如果被赋值的变量是对应的数据类型则可以接受，否则一律按double和integer的数值范围来截取。

```
DECLARE d1, d2 DECIMAL(20, 10);
DECLARE b1, b2 BIGINT;
SET d1 = 1234.43534534531; # exceeding the double data type range
SET d2 = 1234.43534534532;

SET b1 = 1234454654645645651;  # exceeding the integer data type range
SET b2 = 1234454654645645652;
```

以上**SET**等式的比较结果是d1小于d2，b1小于b2。

但如果像下面的 IF 语句，d1 和 d2，b1 和 b2 却是相等的，因为超出表示范围的部分都进行了截断操作：

```
IF 1234.43534534531 = 1234.43534534532 THEN …….     #当成double数据类型被截断
IF 1234454654645645651 = 1234454654645645652 THEN ……. #当成int数据类型被截断
```

简单表达式

简单表达式分为数字数据类型：INTEGER、BIGINT、SMALLINT、DOUBLE、FLOAT、DECIMAL、REAL；字符数据类型：CHAR、NCHAR、VARCHAR、NVARCHAR；BINARY数据类型和时间数据类型：DATE、TIME、TIMESTAMP。

简单表达式包括'+'、'-'、 '*'、 '/'和变量、常量、数值、字符串。简单表达式的执行效率比复杂表达式要高很多，比较适用于多次循环语句，可以极大地提高执行速度。有关SQL函数的相关说明可参考*SQL命令与函数参考手册*。

➲ **语法**

set语句语法如下所示：

```
SET <variable_name> ::= <variable_name> [{ <+ | - | * | / | || >
<variable_name> }...]
```

➲ 示例1

```
CREATE PROCEDURE SETTS(out rc1 int,out rc2 int, out rc3 char(10) )
LANGUAGE SQL
BEGIN
        DECLARE d1 INT DEFAULT 2;
        DECLARE d2,d3 INT DEFAULT 2;
        DECLARE c1,c2 char(10) DEFAULT '12345';
        SET d1 = 1 + d2 * d3*100/10;
        SET d2 = 6;
        SET c2 = c1;
        SET d3 = d1+d2;
        SET rc1 = d1;
        SET rc2 = d2-3;
        SET rc3 = c2;
END;
```

➲ 示例2

```
CREATE PROCEDURE OUTPUTS(OUTPUT V1 INT,OUTPUT V2 BIGINT,
                         OUTPUT V3 FLOAT,OUTPUT V4 DOUBLE,
                         OUTPUT V5 DECIMAL(8,4),
                         OUTPUT V6 BINARY(20),OUTPUT V7 CHAR(20),
                          OUTPUT V8 VARCHAR(20),OUTPUT V9 NCHAR(40),
                          OUTPUT V10 NVARCHAR(40),OUTPUT V11 DATE,
                          OUTPUT V12 TIME,OUTPUT V13 TIMESTAMP)
LANGUAGE SQL
BEGIN
        SET V1 = 1;
        SET V2 = 7396;
        SET V3 = 2.2;
        SET V4 = 3.3;
        SET V5 = 4.4;
        SET V6 = 'ASSIGNMENT';
        SET V7 = 'CHAR';
        SET V8 = 'VARCHAR';
        SET V9 = 'NCHAR';
        SET V10 = 'NVARCHAR';
        SET V11 = '2008-08-08';
        SET V12 = '11:11:11';
        SET V13 = '2008-08-08 11:11:11';
END;
```

复杂表达式

复杂表达式不仅包括简单表达式中所包括的赋值，还包括SQL函数，如内置函数和用户自定义函数。

内置函数可用于结果集的某些字段或某些限定记录的字段上。DBMaster支持的内置函数可参考SQL手册的第4章，每个函数的自变量和返回值都进行了详细地说明。

DBMaster支持用户创建自己的用户自定义函数(UDF)， UDF一旦创建，它就可以作为一个新的内置函数来使用。有关用户自定义函数的详细说明请参考*数据库管理员手册*的第14章。

➲ 示例

set语句用语法如下：

```
################################################################################
#       Module Name = SETTS.SP
#       Purpose  = Store Procedure testing program
#                  1. Test keyword "SET" in Store Procedure
#       Function = 1. decalare d1 d2 d3 c1 c2 for pass parameter
#       Use Database : DBNAME
################################################################################
CREATE PROCEDURE SETTS(out rc1 int,out rc2 int, out rc3 char(10) )
LANGUAGE SQL
BEGIN
         DECLARE d1 INT DEFAULT 2;
         DECLARE d2,d3 INT DEFAULT 2;
         DECLARE c1,c2 char(10) DEFAULT '12345';

         SET d1 = 1 + d2 * d3*100/10;
         SET d2 = 6;
         SET c2 = c1;
         SET d3 = d1+d2;
         SET rc1 = d1;
         SET rc2 = d2-3;
```

```
        SET rc3 = c2;
END;
```

5.6 SQL存储过程中的控制流语句

顺序执行是最基本的程序执行路径。通过这种方式，程序将从代码的第一行开始执行，紧接着下一行，直到执行代码的最后一条语句。这种方法很适合简单的任务，但往往缺乏有效性，因为它只能处理一个情况。程序往往需要能够决定如何应对不断变化的环境，通过控制代码的执行路径，一段特定的代码可以智能地处理多个情况。

SQL控制语句提供的变量支持和流程控制语句可用于控制语句执行的顺序。像IF和CASE这样的语句用于有条件地执行SQL控制语句块，而其它语句如WHILE和REPEAT，通常用于执行一组重复地语句，直到任务完成。

尽管有很多种SQL控制语句，我们将它分为以下几类：

- 变量关联语句（variable related statements）
- 条件语句（conditional statements）
- 循环语句（loop statements）
- Goto语句（Goto statements）
- 返回语句（Return statements）
- 传输控制语句（transfer of control statements）
- 标签和SQL存储过程复合语句（labels and SQL stored procedure compound statements）

变量关联语句

变量关联SQL语句用于声明变量并且为变量赋值，分为以下几类：

- SQL存储过程中的DECLARE <variable>语句
- SQL存储过程中的DECLARE <condition>语句

- SQL存储过程中的DECLARE <condition handler>语句
- SQL存储过程中的DECLARE CURSOR

这些语句为使用其它类型的SQL控制语句提供了必要的支持，并且SQL语句将对变量值非常有用。

条件语句

条件语句会根据条件符合的状态来定义执行什么逻辑，SQL存储过程支持以下两种类型的条件语句：

- CASE
- IF

这些语句非常相似，但其中IF语句是CASE语句的扩展。

SQL存储过程中的CASE语句

CASE语句会依据条件符合的状态有条件地输入一些逻辑，以下有两种类型的CASE语句：

- 简单的CASE语句：依据字面值输入一些逻辑
- 搜索CASE语句：依据表达式的值输入一些逻辑

CASE语句中的WHEN子句定义了符合控制流值的时间。

CASE语句为存储过程实现了一个复合的条件结构。如果search_condition的值为真，那么相应地SQL语句将被执行；如果没有搜索到符合的条件，那么ELSE子句中的SQL语句将被执行。每一个statement_list都由一个或多个语句组成。

➲ 语法

CASE语句语法如下所示：

```
<case_statement_main> ::=
CASE
<variable_case> | <condition_case>
ELSE
```

```
<sp_statement_main>
END CASE
<variable_case> ::= <variable_name> <variable_case_list>
<variable_case_list> ::= WHEN <var_value> THEN <sp_statement_main>
                          [ { < ; > WHEN <var_value> THEN
<sp_statement_main> }... ]
<condition_case> ::= WHEN <condition> THEN <sp_statement_main>
                        [ { < ; > WHEN <condition> THEN
                        <sp_statement_main> }... ]
```

➲ **示例1**

下例为带有WHEN子句的简单CASE语句SQL存储过程：

```
CREATE PROCEDURE UPDATE_DEPT (IN p_workdept char(3))
LANGUAGE SQL
BEGIN
        DECLARE v_workdept CHAR(3);
        SET v_workdept = p_workdept;
        CASE v_workdept
        WHEN 'A00' THEN
        UPDATE department SET deptname = 'D1';
        WHEN 'B01' THEN
        UPDATE department SET deptname = 'D2';
        ELSE
        UPDATE department SET deptname = 'D3';
        END CASE;
END;
```

➲ **示例2**

下例为带有WHEN子句的搜索CASE语句SQL存储过程：

```
CREATE PROCEDURE UPDATE_DEPT (IN p_workdept char(3))
LANGUAGE SQL
BEGIN
        DECLARE v_workdept CHAR(3);
        SET v_workdept = p_workdept;
        CASE
        WHEN v_workdept = 'A00' THEN
        UPDATE department SET deptname = 'D1';
        WHEN v_workdept = 'B01' THEN
```

```
        UPDATE department SET deptname = 'D2';
        ELSE
        UPDATE department SET deptname = 'D3';
        END CASE;
END;
```

上面所提供的例子是逻辑等价的，但重要的是注意带有WHEN子句的搜索CASE语句可以非常强大。任何支持的SQL表达式都可用在此处，这些表达式可以包含相关的变量、参数等。

➲ **示例3**

case用法如下所示：

```
################################################################################
#       Module Name = CASE_TEST.SP
#       Purpose  = Store Procedure testing program
#                     1. Test keyword "CASE" in Store Procedure
#       Use Database : "DBNAME" "SYSADM"  ""
################################################################################
CREATE PROCEDURE CASE_TEST (IN INVAL INT, OUT outval1 INT, OUT outval2 INT)
LANGUAGE SQL
BEGIN
        DECLARE VAL INT;
        SET VAL = INVAL;
        CASE VAL
                WHEN 1 THEN
                        SET OUTVAL1 = 1;
                WHEN 2 THEN
                        SET OUTVAL1 = 2;
                WHEN 3 THEN
                        SET OUTVAL1 = 3;
                ELSE
                        SET OUTVAL1 = 10;
        END CASE;
        CASE
                WHEN VAL = 1 THEN
                        SET OUTVAL2 = 11;
                WHEN VAL = 2 THEN
                        SET OUTVAL2 = 22;
```

```
                WHEN VAL = 3 THEN
                        SET OUTVAL2 = 33;
                ELSE
                        SET OUTVAL2 = 100;
        END CASE;
END;
```

SQL存储过程中的IF语句

IF语句会依据条件符合的状态有条件地输入一些逻辑，IF语句和带有WHEN子句的搜索CASE语句在逻辑上是等价的。

IF语句可选择使用ELSE IF子句和默认ELSE子句，END IF子句用来标识语句的结束。

IF语句实现了一个基本的条件结构，如果search_condition的值为真，那么相应地SQL语句将被执行；如果没有搜索到符合的条件，那么ELSE子句中的SQL语句将被执行。每一个statement_list都由一个或多个语句组成。

➲ **语法**

IF语句语法如下所示：

```
<IF statement main> ::=
IF <condition_value> THEN <sp_statement_main>
[ELSEIF <condition_value> THEN <sp_statement_main>]
[ELSE <sp_statement_main>]
END IF
```

➲ **示例1**

下例为包含IF...CALL语句的SQL存储过程：

```
#############################################################################
#       Module Name = IF_CALL.SP
#       Purpose = Store Procedure testing program
#                 1. Test keyword "IF" and "CALL" in Store Procedure
#       Function = 1. Store Procedure: CRETB CASE_TEST_2 INS
#       Use Database: "DBNAME" "SYSADM"  ""
#############################################################################
CREATE PROCEDURE IF_CALL (INPUT C1 CHAR (20))
```

```
LANGUAGE SQL
BEGIN
        DECLARE OBJ CHAR (10);
        IF C1 = 'CRETB' THEN
                CALL CRETB;
        ELSEIF C1 = 'CASE' THEN
                CALL CASE_TEST_2 (OBJ);
        ELSE
                CALL
INS(1,2,3,4,5,6,7,'binary','char','varchar',N'NCHAR',N'NVARCHAR',
'2008-01-01','11:11:11','2008-01-01 11:11:11');
        END IF;
END;
```

➲ **示例2**

下例为包含IF...SET语句的SQL存储过程：

```
#########################################################################
#       Module Name = SETTS.SP
#       Purpose  = Store Procedure testing program
#                  1. Test keyword "SET" and "IF" in Store Procedure
#       Function = 1. create table TB_1 and insert data into TB_1
#       Use Database: "DBNAME" "SYSADM"  ""
#                        table : TB_1(V1 INTEGER)
#########################################################################
CREATE PROCEDURE IFTEST_1(OUT SUM INTEGER)
LANGUAGE SQL
BEGIN
        DECLARE P_SUM INTEGER;
        DECLARE P_SAL INTEGER;
        DECLARE CUR CURSOR FOR SELECT  V1 FROM TB_1;
        SET P_SUM = 0;
        OPEN CUR;
        FETCH FROM CUR INTO P_SAL;
                IF P_SAL = 2 THEN
                SET P_SUM = 10;
                ELSE
                SET P_SUM = 20;
                END IF;
```

```
                FETCH NEXT FROM CUR INTO P_SAL;
                SET P_SUM = P_SUM+P_SAL;
                FETCH NEXT FROM CUR INTO P_SAL;
                SET P_SUM = P_SUM+P_SAL;
                FETCH NEXT FROM CUR INTO P_SAL;
                SET P_SUM = P_SUM+P_SAL;
                FETCH NEXT FROM CUR INTO P_SAL;
                SET P_SUM = P_SUM+P_SAL;
        CLOSE CUR;
        SET SUM = P_SUM;
END;
```

➲ **示例3**

下例为包含IF…ELSEIF…ELSE语句的SQL存储过程：

```
#########################################################################
#       Module Name = IF_UPDATE.SP
#       Purpose  = Store Procedure testing program
#                  1. Test keyword "IF" and "UPDATE" in Store Procedure
#       Function = 1. create table TB_1 and insert data into TB_1
#       Use Database : "DBNAME" "SYSADM"  ""
#        table : TB_1(V1 INT,V2 DOUBLE,V3 CHAR(6))
#########################################################################
CREATE PROCEDURE IF_UPDATE (IN C1 CHAR(6), IN C2 CHAR(20))
LANGUAGE SQL
BEGIN
        IF C2 = 'FIRST' THEN
                UPDATE TB_1 SET V2 = V2 * 1.10, V1 = 1000
                WHERE V3 = C1;
        ELSEIF C2 = 'SECOND' THEN
                UPDATE TB_1 SET V2 = V2 * 1.05, V1 = 500
                WHERE V3 = C1;
        ELSE
                UPDATE TB_1 SET V2 = V2 * 1.03, V1 = 0
                WHERE V3 = C1;
        END IF;
END;
```

➲ **示例4**

下例为包含IF…ELSEIF语句的SQL存储过程：

```
#########################################################################
#       Module Name = ELSEIFS.SP
#       Purpose  = Store Procedure testing program
#                  1. Test keyword "ELSEIF" in Store Procedure
#       Use Database: "DBNAME" "SYSADM"  ""
#########################################################################
CREATE PROCEDURE ELSEIFS(IN con INT, OUT c1 INT)
LANGUAGE SQL
BEGIN
         IF con = 1 THEN
                  SET c1 = 1;
                  INSERT INTO TB_1 VALUES(C1);
         ELSEIF con = 2 THEN
                  SET c1 = 2;
                  INSERT INTO TB_1 VALUES(C1);
         ELSEIF con = 3 THEN
                  SET c1 = 3;
                  INSERT INTO TB_1 VALUES(C1);
         ELSE
                  IF con = 5 THEN
                           SET c1 = 5;
                           INSERT INTO TB_1 VALUES(C1);
                  ELSEIF con = 6 THEN
                           SET c1 = 6;
                           INSERT INTO TB_1 VALUES(C1);
                  ELSE
                           SET c1 = 7;
                           INSERT INTO TB_1 VALUES(C1);
                  END IF;
         END IF;
END;
```

循环语句

循环语句为重复执行一些逻辑直到条件满足提供了支持，SQL控制语句中支持以下几种循环语句：

- FOR
- LOOP
- WHILE
- REPEAT

FOR语句与其它语句不同，因为它是用来迭代一个定义的结果集行，而其它语句用于迭代一连串的SQL语句，直到每一个条件都满足。

SQL存储过程中的FOR语句

FOR语句是一种特殊类型的循环语句，因为它们可以在定义的只读结果集行中进行迭代。当一个FOR语句执行后，会为每一个FOR-loop循环自动声明一个游标，如果执行返回操作时，结果集将会是游标指向的下一行。循环继续，直到结果集中没有行为止。

FOR语句简化了游标的执行，并且可以在执行的逻辑运算上方便地找回行集字段值。

只要CURSOR的取值不为0，那么FOR语句中的语句列将被重复。statement_list由一个或多个语句组成。FOR语句可以被标签，只有begin_label出现时，end_label才能够出现。如果两者都存在，它们必须是相同的。

➲ 语法

FOR语句语法如下所示：

```
FOR [<for loop variable name> AS]
[<cursor name>[<cursor sensitivity>] CURSOR FOR]
<cursor specification>
DO
    <sql statement list>
END FOR
```

For语句语法支持以下四种形式：

- FOR x AS select * from t1
- FOR x AS cur CURSOR FOR select * from t1

- FOR cur CURSOR FOR select * from t1
- FOR select * from t1

注意 *For语句中必须存在select语句。所查询的表无须在用户创建SQL存储过程时存在。X表示引用变量的范围，如用户可以使用x.c1、x.c2来表示从t1表中查询的结果。如果没有x，那么用户只能使用c1、c2，而不能使用引用形式。上面的cur表示for语句将产生一个名为cur的游标，该游标不能用于引用变量，但可以用于where current of xxxx或FETCH等有关游标的语法中。*

➲ **示例1**

计算表t1中所有数值的和并且将每条数据插入到表t2中，其中c1是t1表中的列名，所以在FOR语法中可以使用c1和x.c1这两种形式来表示。

```
CREATE TABLE t1 (c1 INT);
CREATE TABLE t2 (c1 INT);

CREATE PROCEDURE test1(OUT res INT)
LANGUAGE SQL
BEGIN
        SET res = 0;
        FOR x AS select * from t1
        DO
            SET res = res + c1;
            INSERT INTO t2 VALUES (x.c1);
        END FOR;
END;
```

➲ **示例2**

该例中的x为变量名，用于引用变量old，Cur为游标名，用于update或其它与游标相关的语句中。

```
CREATE TABLE t1 (c1 INT);

CREATE PROCEDURE test3
LANGUAGE SQL
BEGIN
        FOR x AS cur CURSOR FOR select c1 as old from t1 for update
```

```
        DO
            update t1 set c1 = x.old + 100 where current of cur;
        END FOR;
END;
```

➲ **示例3**

该例中没有回滚变量名，用户只能直接使用变量old。Cur是一个游标名，用于update语句或其它与游标相关的语句中。

```
CREATE TABLE t1 (c1 INT);

CREATE PROCEDURE test4
LANGUAGE SQL
BEGIN
        FOR cur CURSOR FOR select c1 as old from t1 for update
        DO
        update t1 set c1 = old + 100 where current of cur;
        END FOR;
END;
```

➲ **示例4**

和上例功能一样，但是由于这个FOR语句没有使用游标名或loop变量名，所以不能使用x.c1的形式来表示。

```
CREATE TABLE t1 (c1 INT);
CREATE TABLE t2 (c1 INT);

CREATE PROCEDURE test2(OUT res INT)
LANGUAGE SQL
BEGIN
        SET res = 0;
        FOR select * from t1
        DO
            SET res = res + c1;
            INSERT INTO t2 VALUES (c1);
        END FOR;
END;
```

SQL存储过程中的LOOP语句

LOOP语句是一种特殊类型的循环语句，因为它没有终止条件子句。它定义的一连串语句重复执行直到遇见另一个逻辑块，通常是一个传输控制语句，强制控制流跳出循环。

LOOP语句非常有用，当您的循环逻辑复杂时，您可能需要多次退出循环。但使用该语句时必须小心，以免造成无限的循环。

如果LOOP语句没有传输控制语句而单独使用，那么包含在循环中的一连串语句将被无限执行，否则可增加条件句柄以强制控制流的更改，或对出现的条件无法处理以强制SQL存储过程的返回。

LOOP实现了一个简单的循环结构，能够重复地执行由一个或多个语句组成的语句序列。循环中的语句重复执行直到退出循环，通常和LEAVE语句一起完成。LOOP语句可以被标签，只有begin_label出现时，end_label才能够出现。如果两者都存在，它们必须是相同的。

➲ 语法

LOOP语句语法如下所示：

```
<loop_statement_main> ::=
LOOP <sp_statement_main> END LOOP
```

➲ 示例

下例为包含LOOP语句的SQL存储过程：

```
CREATE PROCEDURE LOOP_IF (OUTPUT sum INTEGER)
LANGUAGE SQL
BEGIN
        DECLARE p_sum INTEGER;
        DECLARE P_SAL INTEGER;
        DECLARE CUR CURSOR FOR SELECT V1 FROM TB_1;
        SET p_sum = 0;
        SET p_sal = 0;
        OPEN CUR;
        FETCH FROM CUR INTO P_SAL;
        LOOP
                SET p_sum = p_sum + P_SAL;
                FETCH NEXT FROM CUR INTO P_SAL;
```

```
                IF P_SAL=NULL THEN
                        BREAK;
                END IF;
        END LOOP;
        CLOSE CUR;
        SET sum = p_sum;
END;
```

注意 *在LUA文件中，程序中的NULL将被转换成零。*

SQL存储过程中的WHILE语句

WHILE语句定义了一组执行语句，直到WHILE循环的开头判断为否时才停止。while-loop-condition（一个表达式）是在每个循环迭代之前来判断的。

只要search_condition为真，那么WHILE语句中的语句列将被重复。statement_list由一个或多个语句组成。WHILE语句可以被标签，只有begin_label出现时，end_label才能够出现。如果两者都存在，它们必须是相同的。

➲ **语法**

WHILE语句语法如下所示：

```
<while_statement_main> ::=
WHILE <condition_name> DO <sp_statement_main> END WHILE
```

➲ **示例1**

下例为包含简单WHILE循环的SQL存储过程：

```
CREATE PROCEDURE WHILE_LOOP(OUT SUM INTEGER)
LANGUAGE SQL
BEGIN
        DECLARE P_SUM INTEGER;
        DECLARE P_SAL INTEGER;
        DECLARE CUR CURSOR FOR SELECT V1 FROM TB_1;
        SET P_SUM = 0;
        OPEN CUR;
        FETCH FROM CUR INTO P_SAL;
                WHILE (P_SAL!= NULL) DO
```

```
                        SET P_SUM = P_SUM + P_SAL;
                        FETCH NEXT FROM CUR INTO P_SAL;
                END WHILE;
        CLOSE CUR;
        SET SUM = P_SUM;
END;
```

➲ **示例2**

while语句用法如下所示：

```
#############################################################################
#       Module Name = SUM_WHILE.SP
#       Purpose  = Store Procedure testing program
#                  1. Test keyword "WHILE" in Store Procedure
#       Function = 1. create test table TB_1 and TB_2
#                  2. insert test data into TB_1 and TB_2
#                  3. select data from TB_1 TB_2 and use crusor get data
#       Use Database : "DBNAME" "SYSADM"  ""
#       table: TB_1(V1 INTEGER) TB_2(V1 INTEGER)
#############################################################################
CREATE PROCEDURE SUM_WHILE(OUT SUM INTEGER)
LANGUAGE SQL
BEGIN
        DECLARE P_SUM INTEGER;
        DECLARE P_SAL INTEGER;
        DECLARE P_SAL1 INTEGER;
        DECLARE CUR CURSOR FOR SELECT  V1 FROM TB_1;
        DECLARE CUR1 CURSOR FOR SELECT  V1 FROM TB_2;
        SET P_SUM = 0;
        OPEN CUR;
        FETCH FROM CUR INTO P_SAL;
                WHILE(P_SAL != NULL)DO
                        OPEN CUR1;
                        FETCH FROM CUR1 INTO P_SAL1;
                                WHILE(P_SAL1 != NULL)DO
                                        SET P_SUM = P_SUM + P_SAL1;
                                        FETCH NEXT FROM CUR1 INTO P_SAL1;
                                END WHILE;
                        CLOSE CUR1;
```

```
                        FETCH NEXT FROM CUR INTO P_SAL;
                END WHILE;
        CLOSE CUR;
        SET SUM = P_SUM;
END;
The NULL in procedure will be translate as nil in lua file
```

SQL存储过程中的REPEAT语句

REPEAT语句定义了一组执行语句，直到REPEAT循环的末尾判断为真时才停止。repeat-loop-condition是在完成每个循环迭代之后来判断的。

如果WHILE语句的第一行while-loop-condition判断为否，那么将不会进入循环，此时REPEAT语句将变得十分有用。但值得注意的是while-loop逻辑可以被当做REPEAT语句进行重写。

REPEAT和UNTIL语句可用于创建一个循环，一直持续到满足某些逻辑条件为止。

REPEAT循环比较容易维护，因为十分明显的条件会导致循环终止。简单循环中的LEAVE语句可以出现在任何地方，而UNTIL语句总是和END REPEAT子句联合起来出现在循环的最末端。此外，我们并不需要为REPEAT循环指定标签，因为UNTIL条件总是特定于当前循环。但是，我们仍然推荐您使用REPEAT循环的标签以提高程序可读性，特别是在循环嵌套的情况下。

一个REPEAT循环始终保证至少运行一次。也就是说，在循环第一次执行后，UNTIL条件是第一次被判断的。除非满足某些条件，否则循环将一次也不运行。

➲ **示例**

下例为包含REPEAT语句的SQL存储过程：

```
CREATE PROCEDURE REPEATS
LANGUAGE SQL
BEGIN
        DECLARE V INTEGER DEFAULT 0;
          REPEAT
                INSERT INTO TB_1 VALUES(1,2,3,4);
```

```
                SET V = V+1;
        UNTIL V>10
        END REPEAT;
END;
```

Goto语句

Goto语句用于跳过一些语句并且可跳转到任何一条合适的语句上，但使用该语句时请慎重。如果goto语句使用不当，那么执行流将会被打乱成无序状态。如果使用的goto语句错误，那么相关的错误信息将会被存入SQL存储过程的错误信息文件中（sp_name.msg）。

➲ **语法**

Goto语句语法如下所示：

```
< goto statement syntax > ::=
goto <label name>;
```

<label name>：存储过程语句；

标签名称可以和输入/输出的变量名称相同，声明变量名称包括：共同声明变量、游标变量和条件变量。

➲ **示例1**

标签名称和输入的变量名称相同

```
create procedure gsp1(in p1 int, out p2 char(30))
language sql
begin
        set p2='more than or equal 10';
        if p1>=10 then goto p1;
        end if;
        set p2=' less than 10';
        p1: set p1=1;
end;
```

➲ **示例2**

标签名称和共同声明变量名称相同

```
create procedure gsp1(in p1 int, out p2 char(30))
```

```
language sql
begin
        set p2='more than or equal 10';
        if p1>=10 then goto p1;
        end if;
        set p2=' less than 10';
        p1: set p1=1;
end;
```

用户可以在不同范畴内定义相同名称的标签，对于SQLSP支持的标签、begin/end块和loop将会定义一个新范畴。

➲ **示例1**

在不同于开始/结束块的区域内定义相同名称的标签变量

```
create procedure gsp5(out p1 int)
language sql
begin
        lab: set p1=1;
         begin
             lab: set p1=2;
        end;
end;
```

➲ **示例2**

在不同于loop的区域内定义相同名称的标签变量

```
create procedure gsp6(in p1 int)
language sql
begin
        while p1<5 do
        lab: set p1=p1+1;
        end while;
        while p1<15 do
        lab: set p1=p1+1;
        end while;
end;
```

标签可定义在goto语句之前或之后。

➲ **示例1**

标签定义在goto语句之前

```
create procedure gsp10(in p1 int)
language sql
begin
        lab: set p1=p1+1;
        if p1<10 then goto lab;
end if;
end;
```

➲ **示例2**

标签定义在goto语句之后

```
create procedure gsp11(in p1 int, out p2 int)
language sql
begin
        if p1>10 then goto lab;
        end if;
        set p1= p1+1;
        lab: set p2=p1;
end;
```

goto语句可用在if语句、loop语句和条件句柄的句柄行为。

➲ **示例1**

下例为用于if语句的goto语句

```
create procedure gsp7(p1 int, out p2 int)
language sql
begin
        lab:set p1=p1+1;
        If p1<10 then goto lab;
        End if;
        set p2=p1;
end;
```

➲ **示例2**

下例为用于loop语句的goto语句

```
create procedure gsp8(p1 int, out p2 int)
```

```
language sql
begin
        L1:begin
        L2:loop
        set p1=p1+1;
        L3:if p1>5 then goto L4;
        end if L3;
        end loop L2;
        L4:set p2=p1;
        end L1;
end;
```

➲ **示例3**

下例为用于条件句柄的句柄行为中的goto语句

```
create procedure gsp9(out p1 int,out p2 int)
language sql
begin
        declare con1 condition for sqlstate 'HY019';
        declare continue handler for con1
        begin
            set p1=0;
            goto label1;
            set p1=1;
            label1: set p2=1;
end;
declare  i  int;
set i= 3147483647;
end;
```

当用户使用goto语句时，请注意以下错误示例。

➲ **示例1**

用于goto语句的标签必须提前定义，下例为没有经过定义的用于goto语句中的标签：

```
create procedure gsp12(in p1 int, out p2 int)
language sql
begin
        if p1>10 then goto lab;
```

```
        end if;
        set p1= p1+1;
        set p2=p1;
end;
```

➲ **示例2**

同一区域内的标签名称不能重复，下例所属同一区域内的标签名称重复：

```
create procedure gsp13(in p1 int, out p2 int)
language sql
begin
        if p1>10 then goto lab;
        end if;
        lab:set p1= p1+1;
        lab:set p2=p1;
end;
```

➲ **示例3**

下例标签不在可见范围：

```
create procedure gsp14(in p1 int, out p2 int)
language sql
begin
        if p1>10 then goto lab;
        end if;
        while p1<15 do
        lab:set p1= p1+1;
        end while;
        set p2=p1;
end;
```

在使用goto语句时，存在以下一些限制：

- 在条件句柄的句柄行为中使用goto语句，用户必须将句柄行为语句放置于一个新的begin/end块中。同时，goto语句和标签也都应该在一个新的块中。
- Goto语句不能在begin块和end块之间跳转。
- Goto语句不能在SQL存储过程的loop循环中跳转，但是在嵌套循环中，goto语句可以从内循环跳转到外循环。

- 在loop循环中，goto语句不能从外部循环跳转到内部循环，该语法仅DB2支持，DBMaster不支持。
- 对于begin/end块，goto语句不能在块内从外部跳转到内部。
- 标签不能用在空语句之前。

返回语句

通过return语句，SQL存储过程可以从正在执行程序的任何一处退出，然后将出现一条带有用户自定义代码的错误信息和dmSQL命令行工具中的信息。

➲ **语法**

Return语句语法如下所示：

```
< return statement syntax > ::=
return <code>, <statue>
```

➲ **示例1**

下例为条件语句中的返回语句：

```
dmSQL> @@create procedure ret_sp1(c1 int, out c2 int)
       2> language sql
       3> begin
       4> if c1 < 0 then
       5> return  -1, 'error';
       6> end if;
       7> set c2 = c1;
       8> end;@@
dmSQL >  call ret_sp1(-1, ?);
ERROR : [DBMaker] return user defined error code and message : -1, error
```

➲ **示例2**

下例为goto语句中的返回语句：

```
dmSQL> @@create procedure ret_sp2(c1 int, out c2 int)
       2> language sql
       3> begin
       4> declare v1 int default 0 ;
```

```
       5> if c1 < 0 then
       6> set v1 = -1;
       7> goto LEXIT;
       8> elseif c1 > 0 then
       9> set v1 = 0;
       10> goto LEXIT;
       11> end if;
       12> set c2 = c1;
       13> LEXIT :
       14> if v1 !=  0 then
       15> return -1, 'error';
       16> end if LEXIT;
       17> end;@@

dmSQL >  call ret_sp2(0, ?);
dmSQL >  call ret_sp2(-1, ?);
ERROR : [DBMaker] return user defined error code and message : -1, error
```

返回语句的格式必须为如下形式：return error_code, err_message，并且它们可以出现在SQL存储过程的任何地方。

用户可以通过任何一个整型常量来定义错误代码并通过单引号括起来的字符串定义错误信息，错误代码和错误信息都是通过用户自定义的错误代码和信息呈现出来的。

传输控制语句

传输控制语句可用来重定向SQL存储过程中的控制流，这种无条件的转移可使控制流从某点跳至另一点，先于或跟随传输控制语句。SQL存储过程中支持的传输控制语句有：

- ITERATE
- LEAVE

SQL存储过程中的传输控制语句可用于任何地方，然而ITERATE和LEAVE通常和LOOP语句或其它循环语句一起使用。

SQL存储过程中的ITERATE语句

ITERATE语句用于使控制流返回到一个标签LOOP语句的开始。

➲ 示例

下例为包含ITERATE 语句的SQL存储过程：

```
CREATE PROCEDURE ITERATOR
LANGUAGE SQL
BEGIN
        DECLARE v_deptno CHAR(3);
        DECLARE v_deptname VARCHAR(29);
        DECLARE c1 CURSOR FOR SELECT deptno, deptname FROM department ORDER BY
deptno;

        OPEN c1;
        LOOP
                FETCH c1 INTO v_deptno, v_deptname;
                IF v_deptno = NULL THEN LEAVE;
                ELSEIF v_deptno = 'D11' THEN INSERT INTO department (deptno,
                                        deptname) VALUES ('NEW', v_deptname);
                        ITERATE;
                ELSE
                        ITERATE;
                END IF;
        END LOOP;
        CLOSE c1;
END;
```

在这个例子中，当返回行中的字段值匹配某个值时，ITERATE语句可用于将控制流返回到LOOP语句定义的循环中。ITERATE语句的位置能确保没有值能插入到department表中。

SQL存储过程中的LEAVE语句

LEAVE语句可将程序控制转出一个循环或复合语句，该语句可嵌入到SQL存储过程中或动态复合语句中。它不是一个可执行语句，不能动态地准备。

➲ 示例

下例为包含LEAVE语句的SQL存储过程：

```
CREATE PROCEDURE ITEA(OUT C1 INT)
LANGUAGE SQL
BEGIN
        DECLARE V1 INT;
        DECLARE CUR CURSOR FOR SELECT * FROM T3;
        OPEN CUR;
        FETCH CUR INTO V1;
        LOOP
        IF V1 = 2 THEN
                SET C1 = 1;
                FETCH NEXT FROM CUR INTO V1;
                ITERATE;
        ELSEIF V1 != NULL THEN
                FETCH NEXT FROM CUR INTO V1;
                ITERATE;
        ELSE
                LEAVE;
        END IF;
        END LOOP;
        CLOSE CUR;
END;
```

标签和SQL存储过程复合语句

标签可随意命名SQL存储过程中的任何控制语句，包括复合语句和循环。在其它语句中引用标签，您可以强制执行流跳出复合语句或循环，或者再次跳进复合语句或循环的开始。标签可以通过ITERATE和LEAVE语句引用。

您可以为复合语句的结束提供一个相应的标签。如果提供了结束标签，那么该标签必须和用于程序开始的标签是一样的。

SQL存储过程体的每一个标签必须是唯一的。

如果具有相同名称的变量在多个复合语句中进行了声明，那么标签就可以避免名称的混淆。标签可以限定SQL变量的名称。

➲ 语法

以下是复合语句的语法：

```
[<label name> COLON] BEGIN [<any statement list>] END [<label name>]
```

➲ 示例

```
CREATE PROCEDURE test3
LANGUAGE SQL
L1: BEGIN
    ...
    L2: BEGIN
    ...
    END L2;
END L1;
```

注意 *<label name>必须在复合语句中对应出现，并且可以嵌套使用复合语句。*

➲ 示例

下例为带有简单标签的SQL存储过程：

```
CREATE PROCEDURE LABEL_1(OUT C1 char(20))
LANGUAGE SQL
BEGIN
        DECLARE V1 INT;
        DECLARE CUR CURSOR FOR SELECT * FROM T3;
        OPEN CUR;
        FETCH CUR INTO V1;
        label1: LOOP
        IF V1 = 2 THEN
                SET C1 = 'OK';
                FETCH NEXT FROM CUR INTO V1;
                ITERATE label1;
        ELSEIF V1 != NULL THEN
                FETCH NEXT FROM CUR INTO V1;
                ITERATE label1;
        ELSE
                LEAVE label1;
        END IF;
        END LOOP label1;
```

```
        CLOSE CUR;
END;
```

➲ 示例

下例为带有leave标签离开begin/end块的SQL存储过程：

```
CREATE PROCEDURE LEAVE_2(IN V1 INT, OUT V2 INT)
LANGUAGE SQL
BEGIN
lable1: LOOP
    IF V1 < 0 THEN
      SET V2 = -1;
      LEAVE lable1;
    END IF;

    lable2: BEGIN
      IF V1 > 100 THEN
        SET V2 = -2;
         LEAVE lable1;
      END IF;

      lable3: BEGIN
        IF V1 > 50 THEN
            SET V2 = -3;
            LEAVE lable1;
         END IF;
         SET V1 = V1+1;
      END lable3;

    END lable2;

  END LOOP lable1;
END;
```

普通变量的范围检查

普通变量即SQL存储过程支持的基本数据类型(INT, CHAR, FLOAT....)定义的变量。每个变量都有其使用范围，上一层定义的变量可以在下一层使用，但下一层的变量不能在上一层使用。

在存储过程中可以用BEGIN...END将复合语句括起来以表示新的一层。

➲ **示例1**

```
CREATE PROCEDURE test4(OUT res1 INT, OUT res2 INT, OUT res3 INT)
LANGUAGE SQL
L1: BEGIN
    DECLARE c1, c2 INT;
    SET c1 = 1; #在L1层设定一个值
    L2: BEGIN
        DECLARE c1 INT;
        SET c1 = 2;
        SET c2 = 3;
        SET res3 = c1; #c1在L2层被重新定义，屏蔽了L1层的值，所以res3 = 2
    END L2;
    SET res1 = c1; #当L2层的赋值语句结束后，c1又恢复了L1层的值，所以res1 = 1
    SET res2 = c2; #c2只在L1层被定义，L2层中的改变也将影响到L1层，所以res2 = 3
END L1;
```

注意 *FOR 语句也是一个新层。*

➲ **示例2**

```
CREATE PROCEDURE test5(OUT res INT)
LANGUAGE SQL
BEGIN
    DECLARE c1 INT;
    SET c1 = 10;
    SET res = 0;
    FOR select * from t1
   DO
        SET res = res + c1;
        INSERT INTO t2 VALUES (c1);
   END FOR;
   #从FOR语句出来后c1仍等于10，因为c1的影响范围只在FOR语句中
END;
```

SQL存储过程中的SQLCODE和SQLSTATE变量

要执行错误处理或帮助您调试SQL存储过程，您会发现测试SQLCODE或SQLSTATE的值非常有用，返回这些值作为输出参数或将这些值插入到表中以提供基本的追踪支持。

只要执行一条语句，DBMaster就会自动设置这些变量。如果一条语句为一个存在的处理增加了一个条件，那么在处理执行开始时就可以获得SQLSTATE和SQLCODE的变量值。但是只要处理的第一条语句执行，这些变量就会被重新设置。因此，在处理的第一条语句中，通常会将SQLSTATE和SQLCODE的值复制到本地变量中。在下例中，任一条件的CONTINUE句柄用来将SQLCODE变量复制到另一个名为retcode的变量中。变量retcode就可以在执行语句中应用以控制程序逻辑，或作为一个输出变量传递返回值。

当需要检查语句的运行结果时，请优先使用SQLCODE变量，然后使用SQLSTATE变量来检查详细的错误信息，这样可以更准确的了解语句的执行状况。

```
BEGIN
DECLARE retcode INTEGER DEFAULT 0;
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION, SQLWARNING, NOT FOUND SET retcode =
SQLCODE;
END
```

SQL存储过程中的SQLCODE变量定义

SQL存储过程中SQLCODE变量的返回值定义如下：

= 0　成功：表示运行结果成功

= 1　警告：表示运行结果不正常

= 100　没有发现数据：表示在运行结果中没有找到相应地数据

<0　DBMaster native code的负值

SQL存储过程中的SQLSTATE变量定义

目前，SQLSTATE的定义符合ODBC 3.0标准，请参照*错误信息参考手册*。

SQL存储过程中的条件句柄

SQL存储过程的条件句柄：当条件出现时，条件句柄将决定您的SQL存储过程的行为。您可以在SQL程序中为一般条件、命名条件或特殊的SQLSTATE取值声明一个或多个条件句柄。

如果您的SQL存储过程中的一条语句声明了SQLWARNING或NOT FOUND条件，并且为每一个条件声明了句柄，DBMaster会将控制权传递给相关的句柄。如果您没有为这种条件声明句柄，DBMaster会在SQL程序部分将控制权传递给下一条语句。如果SQLCODE和SQLSTATE变量已经声明，那么它们会包含条件句柄的相应取值。

如果您的SQL存储过程中的一条语句增加了SQLEXCEPTION条件，并且为特定的SQLSTATE或SQLEXCEPTION条件声明了句柄时，DBMaster会将控制权进行传递。如果SQLSTATE和SQLCODE变量已经声明，在成功执行条件句柄后，它们各自的取值会变为'00000' 和 0。

如果您的SQL存储过程的一条语句增加了SQLEXCEPTION条件，但没有为特定的SQLSTATE或SQLEXCEPTION条件声明句柄时，DBMaster会终止SQL存储过程并且返回到调用处。

5.7 SQL存储过程的返回结果集

游标可用于遍历多个重复的结果集行，在SQL存储过程中，游标也可以将结果集返回到调用程序中。

要从SQL存储过程中返回结果集，您必须执行以下操作：

1. 使用WITH RETURN子句DECLARE游标。
2. 在SQL存储过程中开启游标。
3. 为客户端程序保持游标的开启状态 – 不要关闭它。

➲ **示例1**

下例为带有一个简单返回的SQL存储过程：

```
CREATE PROCEDURE call_ret
LANGUAGE SQL
BEGIN
DECLARE cur CURSOR WITH RETURN FOR select * from tb;
OPEN cur;
END;
```

➲ **示例2**

```
################################################################################
#       Module Name = RETU.SP
#       Purpose  = Store Procedure testing program
#                  1. Test keyword "RETURN" in Store Procedure
#       Function = 1. decalar r1~r14 for pass parameter
#                  2. create table tb_1 and insert data into tb_1
#                  3. use cursor get data from tb_1
#       Use Database : DBNAME
#        table : TB_1(V1 int,V2 smallint,V3 INT,V4 FLOAT,V5 DOUBLE,
#                     V6 DECIMAL(20,4),V7 BINARY(10),V8 CHAR(20),
#                     V9 VARCHAR(20),V10 NCHAR(40),
#                     V11 NVARCHAR(40),V12 DATE,V13 TIME,V14 TIMESTAMP)
################################################################################
CREATE PROCEDURE RETU
```

```
LANGUAGE SQL
BEGIN
        DECLARE cur CURSOR WITH RETURN FOR select * from tb_1;
        OPEN CUR ;
END;
```

以下为调用RETU的返回结果：

```
dmSQL> call retu;
V1 V2 V3 V4  V5  V6   V7  V8   V9  V10        V11        V12  V13 V14
== == == ==  ==  ==   === ===  ==  ======     ======     === === ====
1  2  3  *00 *00 2.33 626* ch* va* 6e0063006* 6e0076006* 20* 11* 200*
1  2  3  *00 *00 2.33 626* ch* va* 6e0063006* 6e0076006* 20* 11* 200*
1  2  3  *00 *00 2.33 626* 汉* va* 6e0063006* 6e0076006* 20* 11* 200*
1  2  3  *00 *00 2.33 626* 汉* va* 6e0063006* 6e0076006* 20* 11* 200*
1  2  3  *00 *00 2.33 626* 汉* va* 6e0063006* 6e0076006* 20* 11* 200*
1  2  3  *00 *00 2.33 626* ch* 汉* 6e0063006* 6e0076006* 20* 11* 200*
1  2  3  *00 *00 2.33 626* ch* va* 6e0063006* 6e0076006* 20* 11* 200*
1  2  3  *00 *00 2.33 626* ch* va* 6e0063006* ba00ba00d* 20* 11* 200*
1  2  3  *00 *00 2.33 626* ch* va* 6e0063006* 6e0076006* 20* 11* 200*
1  2  3  *00 *00 2.33 626* ch* va* ba00ba00d* 6e0076006* 20* 11* 200*
1  2  3  *00 *00 2.33 626* ch* va* 6e0063006* 6e0076006* 20* 11* 200*
11 rows selected
```

5.8 SQL存储过程的返回状态

状态代码反应了一个存储过程的执行状态以及执行的成功与否，用户不能定义一个存储过程的状态代码。

状态代码：

-1: 存储过程执行错误

0: 存储过程执行正确

1: 存储过程执行警告

如果您想返回存储过程的状态，可以在LANGUAGE SQL之前添加'RETURN STATUS'代码。

➲ **示例**

下例为带有返回状态的SQL存储过程：

```
CREATE PROCEDURE ret_status RETURN STATUS
LANGUAGE SQL
BEGIN

END;
```

5.9 匿名存储过程

匿名存储过程（Anonymous Stored Procedures）是数据库临时创建和执行的一组SQL语句集，无需作为数据库对象永久地存储在DBMaster中。它只能临时存在于某单个SQL区块（SQL block）内，并且只能被创建者使用一次。

匿名存储过程属于一种特殊的SQL存储过程，类似于匿名SQL块（Anonymous SQL Block），可以让用户在client端一次执行多条SQL语句（batch of SQL），且支持包含变量、语法逻辑、游标等在内的全部SQL语法块。匿名存储过程不能使用参数和dmSQL命令行工具特有的命令（如：set等）。在dmSQL中编辑匿名SQL块前需先设定块定界符，否则将返回错误。有关匿名存储过程使用的变量和语法逻辑等，请参考*数据库管理员手册12章SQL存储过程*。

注意 *匿名存储过程的复合语句被“**BEGIN**”和“**END**”包围。*

同SQL存储过程相比，匿名存储过程没有名称，不能通过其它数据库对象来进行引用。也就是说，匿名存储过程的执行是即时的。当用户成功创建一个匿名存储过程时，DBMaster会立即执行该存储过程，当其执行完毕后，DBMaster会立即将其删除。匿名存储过程不会将信息保存到系统表SYSPROCINFO中，也不能永久地存储在DBMaster中以便重复使用。但是匿名存储过程缩短了代码更改和程序执行的时间间隔，从而提高了问题诊断、原型化和代码测试的执行效率，为任务的多次更改和执行提供了便利。

➲ **示例**

通过SQL块创建匿名存储过程：

```
dmSQL> set block delimiter @@;
dmSQL >@@
     2> BEGIN //note:this space have ","
     3> CREATE TABLE tab(c1 INT,c2 INT);
     4> INSERT INTO tab values(123,456);
     5>END;
```

```
     6>@@
dmSQL >SELECT * FROM tab;
    C1          C2
=========== ===========
        123         456
1 rows selected
dmSQL >@@
     2>BEGIN
     3> DECLARE c1 INT;
     4> DECLARE SET INT @a2 = 100;
     5> SET c1 = 200;
     6> INSERT INTO tab VALUES(@a2,c1);
     7>END;
     8>@@
dmSQL >SELECT * FROM tab;
    C1          C2
=========== ===========
        123         456
        100         200
2 rows selected
```

5.10 动态SQL存储过程

EXECUTE IMMEDIATE语句

您可以动态准备并执行一个EXECUTE IMMEDIATE语句。

➲ 语法

EXECUTE IMMEDIATE语句语法：

```
<execute immediate statement main> ::=
EXECUTE IMMEDIATE <SQL statement variable>
```

<SQL statement variable>声明的类型应该为字符串型。

➲ 示例

```
CREATE PROCEDURE dym_test1
LANGUAGE SQL
BEGIN
        DECLARE str char(128);
        SET str= 'insert into t1 values(5)';
        EXECUTE IMMEDIATE str;
END;
```

PREPARE语句

为执行预备一条语句。

➲ 语法

PREPARE语句语法如下所示：

```
<prepare statement main> ::=
PREPARE <SQL statement name> FROM <SQL statement variable>
```

➲ 示例

```
CREATE PROCEDURE dym_test2
LANGUAGE SQL
BEGIN
```

```
        DECLARE str char(128);
        SET str= 'insert into t1 values(5)';

        PREPARE stmt FROM str;
        EXECUTE stmt;
        DEALLOCATE PREPARE stmt;
END;
```

EXECUTE语句

将输入的SQL参数和输出对象与PREPARE语句进行结合并执行该语句。

➲ **语法**

EXECUTE语句语法如下所示：

```
<execute statement> ::=
EXECUTE <SQL statement name> [ <result using clause> ] [ <parameter using
clause> ]
<result using clause> ::= INTO <into argument> [ { <comma> <into argument> }... ]
<parameter using clause> ::= USING <using argument> [ { <comma> <using
argument> }... ]
<into argument> ::= SQL SP variable
<using argument> ::= SQL SP variable
```

➲ **示例1**

```
CREATE PROCEDURE dym_test3(IN val INT)
LANGUAGE SQL
BEGIN
        DECLARE str char(128);
        SET str= 'insert into t1 values(?)';

        PREPARE stmt FROM str;
        EXECUTE stmt USING val;
        DEALLOCATE PREPARE stmt;
END;
```

➲ **示例2**

```
CREATE PROCEDURE dym_test4(IN val INT, OUT co INT)
LANGUAGE SQL
```

```
BEGIN
        DECLARE str char(128);
        SET str= 'select COUNT(*) from t1 where c1=?';

        PREPARE stmt FROM str;
        EXECUTE stmt INTO co USING val;
        DEALLOCATE PREPARE stmt;
END;
```

DEALLOCATE PREPARE语句

通过PREPARE语句分配已经准备好的SQL语句。

➲ **语法**

DEALLOCATE PREPARE语句语法如下所示：

```
<deallocate prepared statement> ::= DEALLOCATE PREPARE <SQL statement name>
```

➲ **示例**

```
CREATE PROCEDURE dym_test2
LANGUAGE SQL
BEGIN
        DECLARE str char(128);
        SET str= 'insert into t1 values(5)';

        PREPARE stmt FROM str;
        EXECUTE stmt;
        DEALLOCATE PREPARE stmt;
END;
```

动态声明游标

声明一个与声明段名称相关联的游标，这个声明段名称会轮流与游标相关联。

➲ **语法**

Dynamic declare cursor语法如下所示：

```
<dynamic declare cursor> ::= DECLARE <dynamic cursor name> [[NO] SCROLL] CURSOR
[WITH RETURN] FOR <prepare statement name>
```

动态开放游标

使用游标输入动态参数并且开启游标。

➲ **语法**

Dynamic open cursor语法如下所示：

```
<dynamic open statement> ::= OPEN <dynamic cursor name> [ <input using clause> ]
```

➲ **示例1**

动态游标

```
CREATE PROCEDURE dym_test5(OUT sum INT)
LANGUAGE SQL
BEGIN
        DECLARE val INT;
        DECLARE str char(128);
        DECLARE CONTINUE HANDLER FOR NOT FOUND;

        SET str= 'select * from t1';
        SET sum = 0;

       PREPARE stmt FROM str;
       DECLARE cur CURSOR FOR stmt;

        OPEN cur;
        WHILE SQLCODE = 0 DO
                        SET sum = sum + val;
                        FETCH cur INTO val;
                END WHILE;
        CLOSE cur;

        DEALLOCATE PREPARE stmt;
END;
```

➲ **示例2**

包含输入参数的动态游标

```
CREATE PROCEDURE dym_test6(IN cd INT, OUT sum INT)
LANGUAGE SQL
BEGIN
        DECLARE val INT;
        DECLARE str char(128);
        DECLARE CONTINUE HANDLER FOR NOT FOUND;

        SET str= 'select * from t1 where c1=?';
        SET sum = 0;

        PREPARE stmt FROM str;
        DECLARE cur CURSOR FOR stmt;

        OPEN cur USING cd;
        WHILE SQLCODE = 0 DO
                        SET sum = sum + val;
                        FETCH cur INTO val;
                        END WHILE;
        CLOSE cur;

        DEALLOCATE PREPARE stmt;
END;
```

➲ **示例3**

返回动态游标的结果集

```
CREATE PROCEDURE dym_test7(IN cd INT)
LANGUAGE SQL
BEGIN
        DECLARE str char(128);
        SET str= 'select * from t1 where c1=?';

        PREPARE stmt FROM str;
         DECLARE cur CURSOR WITH RETURN FOR stmt;
```

```
        OPEN cur USING cd;
END;
```

5.11 临时存储过程

Temp SP是一个临时的存储过程，存在两种状态：local和global。Local Temp SP可以由创建它的连接来调用，但不能由其它连接调用，但其它连接可以创建一个相同名字的Local Temp SP。Global Temp SP与永久SP相同，但其不能被自动删除，当其它连接被赋予权限则可以调用Global Temp SP。Global Temp SP对所有连接都是公开的，即使没有执行权限的连接也可以调用它，当创建global temp sp的连接断开后，其它连接就再也不能调用它。

如果您在多用户模式下创建了一个临时存储过程，那么当连接断开后，这个临时存储过程将被删除。如果您在单用户模式下创建了一个临时存储过程，那么它可以在数据库启动时删除。当您断开创建临时存储过程的连接时，临时存储过程不会立即删除，而将在10秒后删除，用户也可以手动删除临时存储过程，当数据库关闭或崩溃时，数据库重启时将清除所有temp sp。

- Permanent sp和global temp sp名称不能相同。
- 不同的连接能够创建相同名称的local temp程序。
- Temp sp能够被同样名称的永久sp替换，但是永久sp不能被temp sp替换。
- SYSDAM创建名为XX的永久sp，其他用户同样也可以创建名为XX的任意类型sp。

➲ **语法**

```
CREATE [OR REPLACE] [GLOBAL\LOCAL] TEMP PROCEDURE <sp_name>
LANGUAGE SQL
BEGIN
<sp_body>
END;
```

➲ **示例1**

创建一个Temp存储过程tsp1并调用它：

```
dmSQL> set block delimiter @@;
dmSQL> connect to test sysadm;
dmSQL> @@
    2> /* default is local temp sp*/
    3> create temp procedure tsp1
    4> language sql
    5> begin
    6> end;
    7> @@
```

➲ **示例2**

通过out参数创建一个临时存储过程tsp2并调用它：

```
dmSQL> @@
   2> /* create local temp sp*/
   3> create local temp procedure tsp2(out c1 int)
   4> language sql
   5> begin
   6> set c1 = 1;
   7> end;
   8> @@
dmSQL> call tsp2(?);
```

➲ **示例3**

创建一个与tsp2同名的本地Temp存储过程并调用它：

```
dmSQL> use 2;
dmSQL>connect to test sysadm;
dmSQL> @@
    2> /* create local temp sp with the same name */
    3> create local temp procedure tsp2(out c1 int)
    4> language sql
    5> begin
    6> set c1 = 2;
    7> end;
    8> @@
dmSQL> call tsp2(?);
```

➲ **示例4**

创建一个global temp存储过程tsp3：

```
dmSQL> @@
    2> /* create global temp sp by sysadm */
    3> create global temp procedure tsp3
    4> language sql
    5> begin
    6> end;
    7> @@
```

➲ **示例5**

其它用户创建一个与tsp3同名的global temp存储过程：

```
dmSQL> use 3;
dmSQL> connect to test xu;
dmSQL> @@
    2> /* create global temp sp with the same name by sysadm */
    3> create global temp procedure tsp3
    4> language sql
    5> begin
    6> end;
    7> @@
```

5.12 数据处理

这节将通过一些简单的例子来介绍SQL存储过程的数据处理。

创建一个空的SQL存储过程

➲ 示例

创建一个空SQL存储过程：

```
CREATE PROCEDURE SQLSP
LANGUAGE SQL
BEGIN
END;
```

INSERT语句

下例演示了如何在SQL语句中使用insert语句。

➲ 示例

创建一个SQL存储过程，完成对表的插入数据操作：

```
###############################################################################
#       Module Name = INPUTS.SP
#       Purpose  = Store Procedure testing program
#                  1. Test INPUT parameter store procedure
#                  2. Test all type that can use in Store Procedure
#       Function = 1. create table INPUTS
#       Use Database : "DBNAME" "SYSADM"  ""
#        table : INPUTS(V1 int,V2 BIGINT,V3 smallint,V4 INT,V5 FLOAT,
#                       V6 DOUBLE, V7 DECIMAL(20,4),V8 CHAR(10),
#                       V9 CHAR(20),V10 VARCHAR(20), V11 NCHAR(40),
#                       V12 NVARCHAR(40),V13 DATE,V14 TIME,V15 TIMESTAMP)
###############################################################################
CREATE PROCEDURE INPUTS(INPUT V1 int, INPUT V2 BIGINT,
                        INPUT V3 smallint, INPUT V4 INT,
```

```
                        INPUT V5 FLOAT,INPUT V6 DOUBLE,
                        INPUT V7 DECIMAL(20,4),INPUT V8 BINARY(20),
                        INPUT V9 CHAR(20),INPUT V10 VARCHAR(20),
                        INPUT V11 NCHAR(40),INPUT V12 NVARCHAR(40),
                        INPUT V13 DATE,INPUT V14 TIME,
                        INPUT V15 TIMESTAMP,
                        INPUT V16 REAL)
LANGUAGE SQL
BEGIN
        INSERT INTO INPUTS VALUES(V1,V2,V3,V4,V5,V6,V7,V8,
                                  V9,V10,V11,V12,V13,V14,V15,V16);
END;
```

Select语句

下例演示了如何在SQL语句中使用select语句。

➲ 示例

```
###############################################################################
#       Module Name = OUTPUTS.SP
#       Purpose  = Store Procedure testing program
#                1. Test OUTPUT parameter store procedure
#                2. Test all type which can use in Store Procedure
#       Function = 1. create table OUTPUTS and insert data into OUTPUTS
#        2. use cursor get data from OUTPUTS and pass data  to OUTPUT parameter
#       Use Database : DBNAME
#        table : OUTPUTS(V1 int, V2 BIGINT, V3 smallint,V4 INT,
#                        V5 double,V6 DOUBLE, V7 DECIMAL(20,4),
#                        V8 BINARY(10),V9 CHAR(20), V10 VARCHAR(20),
#                        V11 NCHAR(40),V12 NVARCHAR(40),
#                        V13 DATE,V14 TIME,V15 TIMESTAMP,V16 REAL)
###############################################################################

CREATE PROCEDURE OUTPUTS(OUTPUT V1 int, OUTPUT V2 BIGINT,
                        OUTPUT V3 smallint, OUTPUT V4 INT,
                        OUTPUT V5 FLOAT,OUTPUT V6 DOUBLE,
                        OUTPUT V7 DECIMAL(8,4),OUTPUT V8 BINARY(20),
                        OUTPUT V9 CHAR(20),OUTPUT V10 VARCHAR(20),
                        OUTPUT V11 NCHAR(40),
```

```
                        OUTPUT V12 NVARCHAR(40),OUTPUT V13 DATE,
                        OUTPUT V14 TIME,OUTPUT V15 TIMESTAMP,OUTPUT V16 REAL)
LANGUAGE SQL
BEGIN
        DECLARE CUR CURSOR FOR select * from OUTPUTS;

        OPEN CUR;
        FETCH FROM CUR INTO V1,V2,V3,V4,V5,V6,V7,V8,V9,V10,
                            V11,V12,V13,V14,V15,V16;

        CLOSE CUR;
END;
```

Create语句

下例演示了如何在SQL语句中使用create语句。

➲ **示例1**

使用SQL存储过程来创建表：

```
CREATE PROCEDURE CRETB
LANGUAGE SQL
BEGIN

        CREATE TABLE TB_1(V1 int, V2 BIGINT, V3 smallint,V4 INT,
                          V5 FLOAT,V6 DOUBLE,V7 DECIMAL(8,2),V8 CHAR(20),
                          V9 CHAR(20),V10 VARCHAR(20),V11 CHAR(40),
                          V12 VARCHAR(40),V13 DATE,V14 TIME,
                          V15 TIMESTAMP,V16 REAL);
END;
```

➲ **示例2**

使用SQL存储过程来创建视图：

```
CREATE PROCEDURE CREVE
LANGUAGE SQL
BEGIN
        CREATE VIEW VE_1 AS SELECT * FROM TB_1;
END;
```

➲ **示例3**

使用SQL存储过程来创建用户自定义数据类型：

```
CREATE PROCEDURE CREDM
LANGUAGE SQL
BEGIN
        CREATE DOMAIN TYP_1 VARCHAR(35);
END;
```

➲ **示例4**

使用SQL存储过程来创建索引：

```
CREATE PROCEDURE CREIND
LANGUAGE SQL
BEGIN
        CREATE UNIQUE INDEX IND_1 ON TB_1(V1);
END;
```

Drop语句

下例演示了如何在SQL语句中用drop语句。

➲ **示例1**

使用SQL存储过程删除表：

```
CREATE PROCEDURE DRPTB(IN V1 CHAR(20),OUT WORNING CHAR(40))
LANGUAGE SQL
BEGIN
        IF V1 = 'DROP TABLE' THEN
                DROP TABLE TB_1;
                SET WORNING ='NORMAL! TABLE TB_1 DROPED';
        ELSEIF V1 = 'CREATE TABLE' THEN
                CALL CRETB;
                SET WORNING ='NORMAL! CREATE A NEW TABLE NAMED TB_1';
        ELSE
                SET WORNING = 'YOU INPUT WRONG PARAMETER!';
        END IF;
END;
```

➲ **示例2**

使用SQL存储过程删除视图：

```
CREATE PROCEDURE DRPVE(IN V1 CHAR(20),OUT WORNING CHAR(40))
LANGUAGE SQL
BEGIN
        IF V1 = 'DROP VIEW VE_1' THEN
                DROP VIEW VE_1;
                SET WORNING ='NORMAL! VIEW VE_1 DROPED';
        ELSEIF V1 = 'CREATE VIEW' THEN
                CALL CREVE;
                SET WORNING ='NORMAL! CREATE VIEW AS SELECT V1 FROM TB_1';
        ELSE
                SET WORNING = 'YOU INPUT WRONG PARAMETER!';
        END IF;
END;
```

跟踪SQL存储过程的执行

功能性跟踪用来帮助用户跟踪SQL存储过程调试的执行。开启并使用TRACE函数来确定跟踪变量的位置和打印信息。在SQL存储过程执行后，所有跟踪信息都将写入DBMaster bin 路径下的**_sptrace.log**文件里。

➲ **语法**

TRACE语句语法如下所示：

```
<TRACE statement main> ::=
TRACE [ON <trace_file> | OFF | <trace_detail>]
```

其中TRACE的形式为：

```
TRACE('V1=', V1);
TRACE('V2=', V2, 'V3=', V3);
```

表达式必须使用单引号（''）括起来，且支持所有数据类型，但其中BINARY、NCHAR、NVARCHAR显示为16进制格式。

您可以开启并使用TRACE功能来设置跟踪变量和输出结果。当存储过程执行后，所有跟踪信息都将存放于**_spusr.log**文件中，此文件位于客户机**dmconfig.ini**文件中**DB_SPLog**定义的目录下。

➲ **示例1**

```
CREATE PROCEDURE INPUTS(INPUT V1 int, INPUT V2 BIGINT,
                        INPUT V3 smallint,INPUT V4 INT,
                        INPUT V5 FLOAT,INPUT V6 DOUBLE,
                        INPUT V7 DECIMAL(20,4),
                        INPUT V8 BINARY(20),INPUT V9 CHAR(20),
                        INPUT V10 VARCHAR(20),INPUT V11 NCHAR(40),
                        INPUT V12 NVARCHAR(40),INPUT V13 DATE,
                        INPUT V14 TIME, INPUT V15 TIMESTAMP,INPUT V16 REAL)
LANGUAGE SQL
BEGIN
        TRACE ON;
        TRACE('V1=', V1);
        TRACE('V2=', V2);
        TRACE('V3=', V3);

        TRACE('V4=', V4);
        TRACE('V5=', V5);

        TRACE('V8=', V8);
        TRACE('V9=', V9);

        TRACE('V12=', V12);
        TRACE('V13=', V13);
        TRACE('V14=', V14);
        TRACE('V15=', V15);
        TRACE('V16=', V16);
        TRACE OFF;
        INSERT INTO INPUTS VALUES(V1,V2,V3,V4,V5,V6,V7,V8,V9,
                                              V10,V11,V12,V13,V14,V15,V16);
END;
```

经过调用输入
(1,7396,2,3,4,5,6,'binary','char','varchar','NCHAR','NVARCHAR','2008-

01-01','11:11:11','2008-01-01 11:11:11','123.456')后，_sptrace.log会增加下列记录：

```
INPUTS 47: Begin trace ====>
INPUTS 48: V1=1
INPUTS 49: V2=7396
INPUTS 50: V3=2
INPUTS 51: V4=3
INPUTS 52: V5=4
INPUTS 53: V8=62696e617279
INPUTS 54: V9=char
INPUTS 55: V12=4e00560041005200430048004100 5200
INPUTS 56: V13=2008-01-01
INPUTS 57: V14=11:11:11
INPUTS 58: V15=2008-01-01 11:11:11.000
INPUTS 59: V16=123.4560012817383
```

6 SQL存储过程

开发SQL存储过程与开发其它类型的存储过程类似，涵盖了从设计到部署阶段所需的所有步骤。

在这里，我们将从以下几个方面介绍SQL存储过程的使用。

- 创建SQL存储过程
- 执行SQL存储过程
- 删除SQL存储过程

为了帮助您开发SQL存储过程，我们列举了几个例子以供参考。

6.1 创建SQL存储过程

创建SQL存储过程需要明白设计需求、SQL存储过程的特征、如何使用这些特征以及任何可能影响到设计的限制。

创建SQL存储过程与创建任何一种数据库对象类似，都是由SQL语句组成。SQL存储过程可以通过DBMaster命令行工具(dmSQL)的CREATE PROCEDURE语句来创建，也可以直接通过图形化用户界面工具(JDBA)来实现。

从外部文件创建SQL存储过程

首先撰写一个SQL存储过程并将它保存至外部文件，然后通过DBMaster命令行工具dmSQL将该SQL存储过程保存至数据库中。

注意 *DBMaster的SQL存储过程只能通过调用外部文件（*.sp）的方式来创建，不能直接将它写在dmsql命令行工具中。*

➲ **语法**

创建SQL存储过程语法：

```
<CREATE SQL PROCEDURE syntax> ::=
CREATE PROCEDURE FROM <file_name>
```

➲ **示例**

通过外部文件创建一个SQL存储过程：

```
dmSQL> CREATE PROCEDURE FROM 'CRETB.SP';
dmSQL> CREATE PROCEUDRE FROM '.\SPDIR\CRETB.SP';
dmSQL> CREATE PROCEDURE FROM 'D:\DATABASE\SPDIR\CRETB.SP';
```

上面展示了如何使用dmSQL命令行工具从外部文件创建SQL存储过程。

从脚本创建存储过程

在DBMaster中，用户不仅可以从文件创建sp，还可以在dmSQL中创建SQL SP，调用并删除所属的sql sp，并且可以执行赋予权限的SQL SP。

SQL SP包含一个以上的SQL语法，并且每一个都以';'结束，所以dmSQL必须支持块分隔。块分隔可以是一组a-z、A-Z、@、%、^，并且由至少两个字符不超过七个字符组成。在块分隔里输入';'并不表示结束。另外，用户必须在dmSQL中写SQLSP之前设置块分隔。当未设置块分隔时将会返回错误信息。

➲ **语法**

创建SQL存储过程语法：

```
CREATE PROCEUDRE <sp_name>
LANGUAGE SQL
BEGIN
        <sp_body>
END;
```

➲ **示例**

从脚本创建一个SQL存储过程：

```
dmSQL> set block delimiter @@;
dmSQL> @@
    2> create procedure sp_in_script2
    3> language sql
    4> begin
    5> insert into t1 values(1);
    6> end;
    7> @@
dmSQL> set block delimiter;
```

注意 *在dmSQL中# 不是命令字符，因为表名里有可能含有# ，命令行我们支持--、 //、块命令里支持/**/。虽然从文件创建SQL SP时'#'是一个命令，我们还是建议用户在SQL SP文件中不要使用#符号'。*

注意 *如果创建存储过程的命令大小超过4K，dmSQL就不能将其保存为历史命令，因此用户无法使用历史命令。*

使用ODBC API创建

ODBC API支持创建永久的sp和temp sp，用户可以使用ODBC来写AP，它支持创建任何格式的SQL程序。

➲ **示例**

使用ODBC API创建一个SQL存储过程：

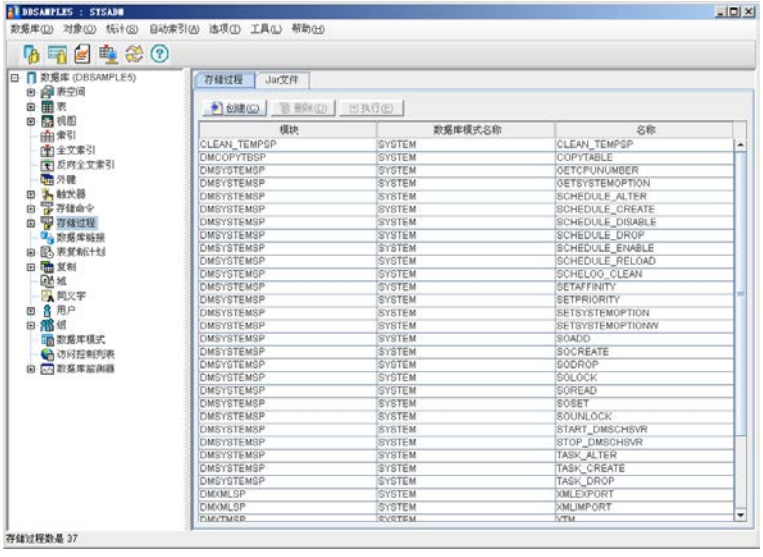
```
#define CRELTSP "create temp procedure tsp1 language sql begin end;"
#define CREGTSP "create global temp procedure tsp2 language sql begin end;"

int main()
{
        …
        SQLExecDriect(hstmt, CRELTSP, SQL_NTS);

        SQLPrepare(hstmt, CREGTSP, SQL_NTS);
         SQLExecute (hstmt);
                …
}
```

使用JDBA工具

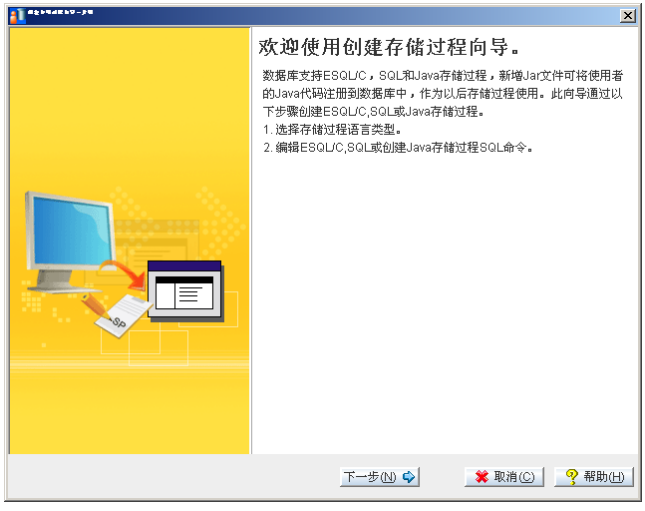
DBMaster提供了三种语言方式来创建存储过程：SQL、ESQL/C和Java。下例为使用SQL语言创建存储过程的范例。

➲ 创建一个SQL存储过程

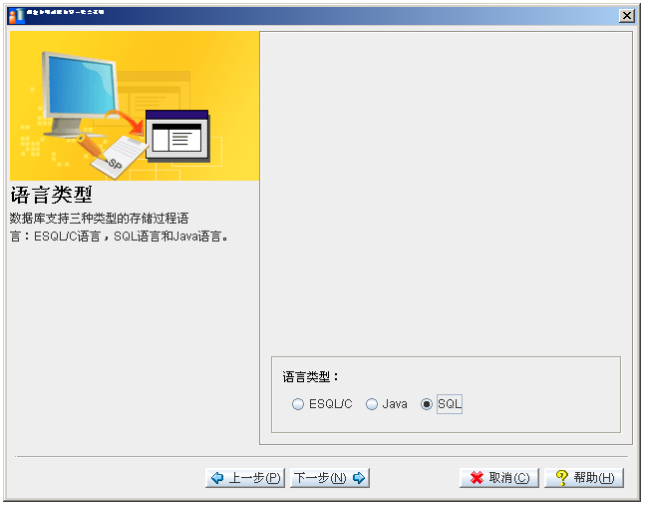
1. 点击树型图中的**存储过程**对象节点，进入**存储过程**页面。



2. 点击**创建**按钮，打开**创建存储过程向导**的**介绍**窗口。

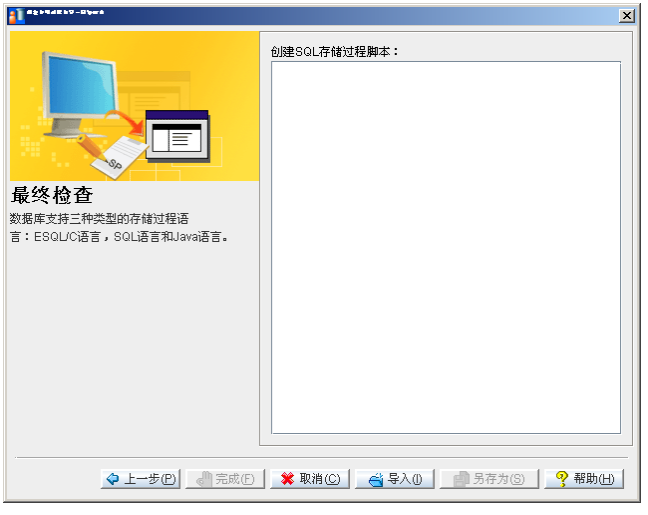


3. 点击**下一步**。打开选择**语言类型**窗口。

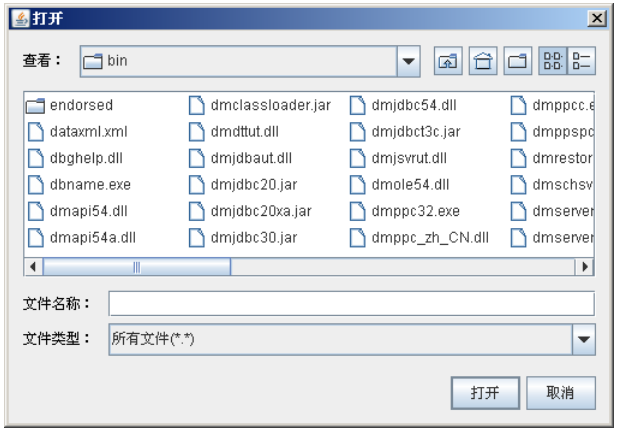


4. 点击**SQL**单选框，使用SQL语言创建存储过程。

5. 点击**下一步**，显示**最终检查**窗口。您可以在右侧控制台中输入SQL语句，或是点击**导入**按钮，导入文件来建立存储过程。



6. 点击**导入**按钮进入**打开**窗口，导入数据库服务器或网络驱动上的SPDIR目录。在**文件名**字段中键入文件路径，或通过浏览目录树找到正确的路径。



7. 点击**打开**按钮，打开选择的文件。

8. 如果导入的文件包括格式化（ASIIC）文本，或者您选择手动输入代码，**最终检查**窗口再次弹出。点击**另存为**按钮将存储过程存放于另一位置，点击**完成**按钮完成存储过程的编译和存储。

9. 如果存储过程编译正确，将弹出确认对话框。

10. 点击**确定**按钮，存储过程创建完成。

6.2 执行SQL存储过程

SQL存储过程可以通过图形化用户界面工具（JDBA Tool）的CALL语句来执行，也可以直接通过DBMaster命令行工具（dmSQL Tool）来执行。

执行SQL存储过程语法

CALL语句用来调用一个存储过程。该语句可以被嵌入到应用程序中，通过动态SQL语句或动态准备来发布。

您可以使用dmSQL工具或触发行为来调用一个存储过程。用户必须拥有SQL存储过程执行CALL PROCEDURE语句的权限。

➲ **语法**

CALL SQL存储过程语法如下所示：

```
<CALL SQL stored procedure> ::=
CALL <procedure_name> [<variable_name> [ { <comma> <variable_name> }... ]]
```

➲ **示例1**

调用其它SQL存储过程：

```
CREATE  TABLE call_tb(c1 INT);
CREATE PROCEDURE case_test_1(IN inval INT, OUT outval1 INT, OUT outval2 INT)
BEGIN
        DECLARE cur CURSOR WITH RETURN FOR select * from call_tb;
        OPEN cur;
END;

CREATE PROCEDURE call1(IN inval INT, OUT outval1 INT, OUT outval2 INT)
LANGUAGE SQL
BEGIN
        CALL CASE_TEST_1(inval, outval1, outval2);
END;
```

➲ **示例2**

调用其它ESQL存储过程：

```
EXEC SQL CREATE PROCEDURE ecret(INT ct output) RETURNS STATUS;
{
EXEC SQL BEGIN CODE SECTION;
        EXEC SQL SELECT count(*) FROM call_tb INTO :ct;
        exec sql RETURNS STATUS SQLCODE;
        EXEC SQL END CODE SECTION;
}

CREATE PROCEDURE call2(OUT st INT, OUT i2 INT)
LANGUAGE SQL
BEGIN
        SET st = CALL ecret(i2);
END;
```

➲ **示例3**

调用JAVA存储过程：

```
CREATE PROCEDURE CALLJAR
LANGUAGE SQL
BEGIN
       CALL INSERTS_3;            #  INSERTS_3 is a java sp
END;
```

➲ **示例4**

调用其它存储过程作为游标：

```
CREATE PROCEDURE call3(OUT i1 INT, OUT i2 INT)
LANGUAGE SQL
BEGIN
        DECLARE cur CURSOR FOR call call_test;
        OPEN cur;
        FETCH FROM CUR INTO i1, i2;
        CLOSE cur;
END;
```

➲ **示例5**

调用其它存储过程作为结果集：

```
CREATE PROCEDURE call4
LANGUAGE SQL
BEGIN
        DECLARE cur CURSOR WITH RETURN FOR call call_test;
        OPEN cur;
END;
```

➲ **示例6**

在dmSQL命令行工具中调用存储过程：

```
dmSQL> create table tb_1(name char(20),phone char(20));
dmSQL> select * from tb_1;
       NAME                 PHONE
==================== ====================

0 rows selected
dmSQL> CREATE PROCEDURE FROM 'D:\SPDIR\INSERT_1.SP';
dmSQL> CALL INSERT_1;
dmSQL> SELECT * FROM TB_1;

       NAME                 PHONE
====================    ====================
JONTH                     1234567

1 rows selected
```

➲ **示例7**

在标准ODBC 程序中，调用存储过程proc1：

```
dmSQL> create table odbc(c1 char(10),c2 char(10),c3 int);
dmSQL> insert into odbc values(?,?,?);
dmSQL/Val> 'linda','linda',1;
1 rows inserted
dmSQL/Val> end;

CREATE PROCEDURE proc1(in i1 char(10), in i2 char(10))
LANGUAGE SQL
BEGIN
        DECLARE cur CURSOR with return FOR select c3 from odbc where
        c1=i1 and c2=i2;
```

```
        OPEN cur;
END;
SQLPrepare(cmdp,(UCHAR*)"call proc1(?, ?)", SQL_NTS);

SQLBindParameter(cmdp, 1, SQL_PARAM_INPUT_OUTPUT, SQL_C_CHAR, SQL_CHAR,
20, 0, n1, 20, NULL);

SQLBindParameter(cmdp, 2, SQL_PARAM_INPUT_OUTPUT, SQL_C_CHAR, SQL_CHAR,
20, 0, n2, 20, NULL);

SQLBindCol(cmdp, 1, SQL_C_LONG, &i, sizeof(long), NULL);
SQLExecute(cmdp); /* get n2 */

while ((rc=SQLFetch(cmdp))!=SQL_NO_DATA_FOUND) /* fetch result set */
```

通过触发行为执行SQL存储过程

用户可通过触发行为调用存储过程，首先创建一张表，如tb_2(name char(20),phone char(20))，然后为tb_2创建一个触发器。

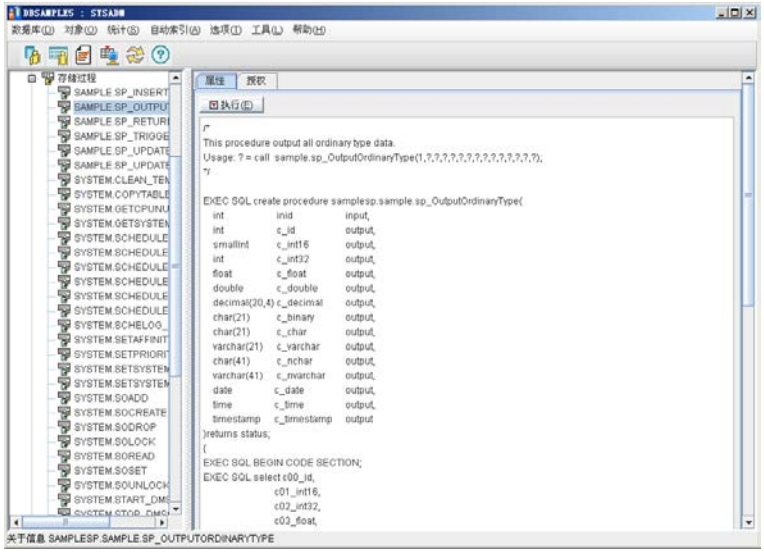
```
CREATE TRIGGER TRG_1 AFTER INSERT ON TB_2 FOR EACH ROW (CALL INSERT_1);
```

使用JDBA工具

SQL存储过程创建后，您可以直接执行或在应用程序中进行调用。

➲ **执行SQL存储过程**

1. 展开树型图中的**存储过程**对象节点，选择要执行的存储过程，进入**属性**页面。



注意 *双击树型图中的存储过程节点，也将显示同样窗口。*

2. 点击**执行**按钮，显示执行的存储过程。

3. 点击**确定**按钮。

6.3 删除SQL存储过程

您可以使用JDBA工具或dmSQL命令行工具删除一个SQL存储过程。

使用dmSQL命令行工具

➲ **语法**

删除SQL存储过程语法如下所示：

```
<DROP SQL stored procedure> ::=
DROP PROCEDURE <procedure_name>
```

➲ **示例**

在dmSQL命令行工具中删除SQL存储过程：

```
dmSQL>   DROP PROCEDURE PROC1;
dmSQL>   DROP PROCEDURE USER1.PROC1;
```

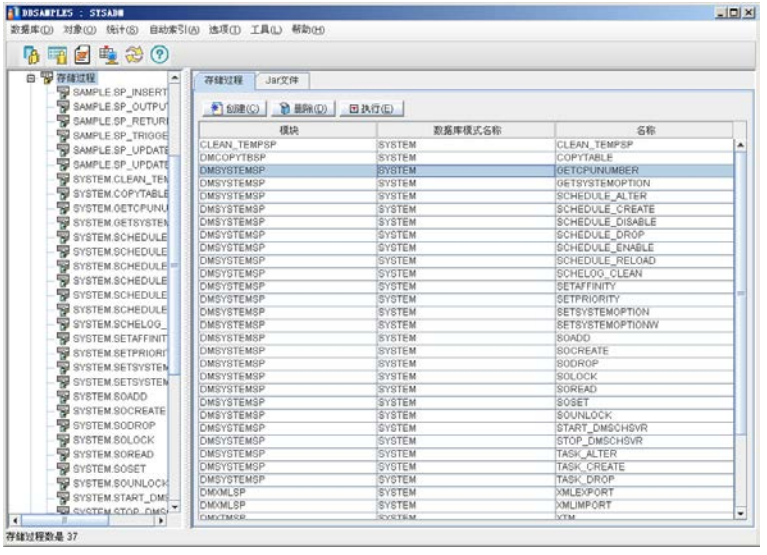
第一条语句删除了存储过程**proc1**，第二条语句删除了存储过程**user1.proc1**。

使用JDBA工具

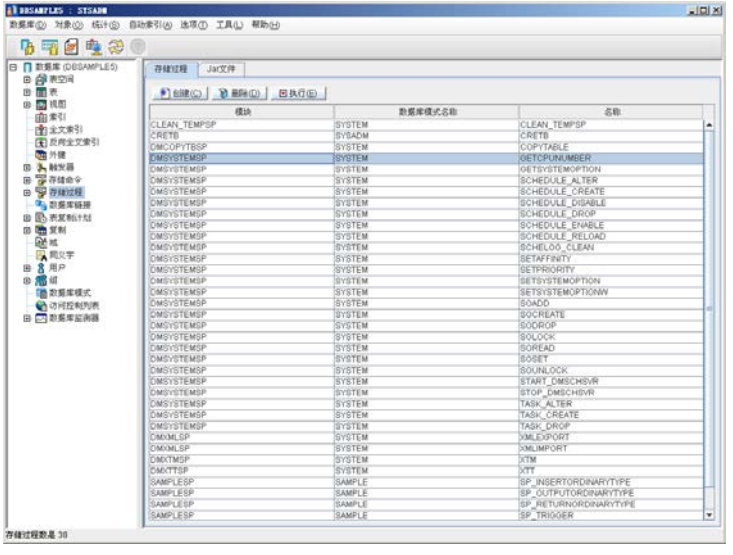
您也可以通过JDBA图形界面工具删除一个不需要的SQL存储过程。

➲ **删除SQL存储过程**

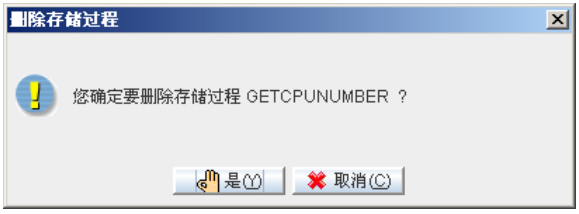
1. 点击目录树中的**存储过程**对象，数据库中的所有存储过程都将出现。



2. 选择一个SQL存储过程。



3. 点击**删除**按钮，弹出**删除存储过程**对话框。



4. 点击**确定**按钮，完成删除操作，或点击**取消**按钮，终止删除过程。

6.4 获得SQL存储过程信息

您可以通过dmSQL命令行工具查询SYSPROCINFO和SYSPROCPARAM系统表以获取SQL存储过程信息。

➲ 示例

```
dmSQL> select * from sysprocinfo;
dmSQL> select * from SYSPROCPARAM;
```

6.5 安全管理

只有存储过程的拥有者、DBA或具有更高权限的用户才能够执行存储过程，其它用户或用户所在的组必须被授予存储过程的执行权限才能够调用该存储过程。当然，也只有存储过程的拥有者、DBA或具有更高权限的用户才能够授予其他用户EXECUTE PROCEDURE权限。

➲ **语法**

GRANT EXECUTE语句语法如下所示：

```
<GRANT EXECUTE syntax> ::=
GRANT EXECUTE ON <COMMAND | PROCEDURE> <executable_name> TO
< <user_name> | <group_name> | <PUBLIC> > [ { <comma> < <user_name> |
<group_name> | <PUBLIC> > }... ]
```

存储过程的拥有者、DBA或具有更高权限的用户可以取消其他用户存储过程的执行权限。

➲ **语法**

REVOKE EXECUTE语句语法如下所示：

```
<REVOKE EXECUTE syntax> ::=
REVOKE EXECUTE ON <COMMAND | PROCEDURE> <executable_name> FROM
<<user_name> | <group_name> | <PUBLIC>> [ { <comma> [<user_name> | <group_name> |
<PUBLIC>] }... ]
```

➲ **示例1**

用户user1通过dmSQL取消用户**user1**创建的存储过程**proc1**，并将该存储过程的执行权限授予用户**user2**：

```
dmSQL> GRANT EXECUTE ON PROCEDURE proc1 TO user2;
```

➲ **示例2**

用户user1通过dmSQL取消用户**user1**创建的存储过程**proc1**，并将该存储过程的执行权限授予**PUBLIC**：

```
dmSQL> GRANT EXECUTE ON PROCEDURE proc1 TO PUBLIC;
```

➲ **示例3**

用户**user1**通过dmSQL取消**user2**对存储过程proc1的执行权限：

```
dmSQL> REVOKE EXECUTE ON PROCEDURE proc1 FROM user2;
```

➲ **示例4**

用户**user1**通过dmSQL取消**PUBLIC**对存储过程proc1的执行权限：

```
dmSQL> REVOKE EXECUTE ON PROCEDURE proc1 FROM PUBLIC;
```

6.6 SQL存储过程的配置设置

当存储过程创建时，一个相应的动态链接库也就随之创建，并存储在服务器上。默认状态下，动态链接库存放于DBMaster 数据库的工作目录下。数据库管理员可以通过关键字**DB_SPDir**来为存储过程的动态链接库文件指定不同的存放路径。

当存储过程被创建或执行时，客户端的用户可以通过关键字**DB_SPLog**来指定一目录，用于接受从服务器返回的错误信息文件和跟踪文件。

➲ **示例1**

在**dmconfig.ini**配置文件中，将存储过程的动态链接文件的默认路径设置为*/usr1/dbmaster/data/SP*：

```
DB_SPDIR=/usr1/DBMaster/data/SP
```

➲ **示例2**

在**dmconfig.ini**配置文件中，将存储过程日志文件的路径设置为*c:\usr\jerry\data\SP* ：

```
DB_SPLOG=c:\usr\jerry\data\SP
```

7 移植SQL存储过程

要将存储过程移植到其它数据库，可以使用 dmSQL 工具中的命令 unload/load procedure 来完成。

7.1 载出\载入存储过程

```
dmSQL>unload db to dbname;
```

载出过程[PROC | PROCEDURE]

```
dmSQL> unload procedure from call to 'd:\spdir\call';
```

执行上面命令后系统在*d:\spdir*下生成两个文件call.b0和call.so。call.b0用来存储BLOB数据、call.s0用来存储脚本。

载入过程[PROC | PROCEDURE]

```
dmSQL> load procedure from 'd:\spdir\call';
```

执行以上命令后，系统将从外部文件载入存储过程。

8 SQL存储过程限制

DBMaster SQL 存储过程中有一些不支持的语法，这些语法如下：

1. 如果简单表达式中有NULL，则整个表达式为NULL。如果比较表达式中有NULL，则该表达式为false。
2. 不支持**0x**形式表示的十六进制数据格式(0x是C语言的表示方式)，但支持十进制和指数形式表示的数据格式。

   ➲ **示例 1**：下例为不支持的数据格式：

   ```
   SET d1 = 0x1A;
   SET d2 = 0x124C;
   ```

   ➲ **示例 2**：下例为支持的数据格式：

   ```
   SET d1 = 12.3;
   SET d2 = 1.2345E2;
   ```

3. 不支持在SQL存储过程名称中包括*project_name.module_name*的形式，同时*user.SYSPROCINFO*表中的**MODULENAME**字段为空。
4. 用SET语句对字符型变量赋值，若长度超过变量所定义的长度，将会发生截断：

➲ 示例

c1的值为1234：

```
DECLARE c1 char(4);
SET c1 = '1234';
```

5. 用**SET**语句对变量进行赋值，会对变量的类型进行检查。不同属性类型之间不能赋值，也就是不能够把字符型数据赋给数值型变量，反之亦然；对于数值型之间的赋值，若要把小数赋给一个整型数，则会自动舍去小数部分。

6. 字符串只支持使用两个单引号（""）括起来，同时遵守如下规则：

- 字符串中连续两个""表示其中一个""是字符串中的内容。

➲ 示例

```
SET c1 = '12"34"56"78"9';  -- c1 = 12"34"56"78"9
```

- 字符串中可以有任意的双引号""，它只是字符串的一部分，不会被解释为字符串的分隔符。

➲ 示例

```
SET c1 = '12"34"56"78"9';  -- c1 = 12"34"56"78"9
```

- 字符串中可以包括反斜杠"\"。

➲ 示例

```
SET c1 = '12345\6789';  -- c1 = 12345\6789
```

但是若在反斜杠之后的字符是注释符**"#"**，则会将注释符转义为普通字符**"#"**。

➲ 示例

```
SET c1 = '12345\#6789';  -- c1 = 12345#6789
```

- 字符串中可以包括回车换行，如：

➲ 示例

```
SET c1 = '12345
6789';
```

则

```
c1 = 12345
6789
```

- 若在回车换行之前是反斜杠"\"，则表示将去掉回车换行，同时连接上下两行，如：

➲ 示例

```
SET c1 = '12345\
6789';
```

则

```
c1 = 123456789
```

7. 支持*SET n1, n2, …. = select c1, c2, … from t1*;语法，执行的时候只取select的第一条记录，如果被赋值的变量数与结果集中的字段数不相等，将按照下面的规则：

当变量数有n个，结果集字段数有m个，如果n<=m，则取结果集的前n个，否则取全部结果集，并把多余的变量赋值为NULL，同时还检查结果集字段元类型和对应的变量类型是否匹配，如果不匹配则提示错误信息。

最常用的方式就是用count()计数：

➲ 示例

```
SET sum = select count(*) from t1;
```

8. 取模运算需要用SQL函数MOD()来进行。

9. Dmconfig.ini配置文件的**DB_SPLog**参数只在client端有效，对于SQL 存储过程的**TRACE**信息无用。TRACE只接受默认路径，也就是server上的DBMaster执行目录，信息将默认写到该路径的**_sptrace.log**文件里。

10. SQL 存储过程的脚本可以为任意扩展名，甚至可以没有扩展名。

11. 当在SQL 存储过程中的语句存在和变量名相同的表名称或字段元名称时，需要将表名称或字段名称用双引号""""括起来以和变量名作出区别。

➲ 示例

```
DECLARE c1 INT;
Select "c1" from t1 where "c1" > c1;
```

12. 除了以下所列command，支持所有SQL command：

ABORT BACKUP

BEGIN BACKUP

BEGIN WORK

END BACKUP

13. 如果在执行SQL存储过程时产生WARNING或ERROR，默认情况下，WARNING会被忽略并继续执行下去，但最后的WARNING信息会返回给用户，ERROR则会停止执行并返回给用户，并且当一个SQL 存储过程中同时发生WARNING和ERROR时，默认情况下会只返回ERROR。

14. 不支持类似下面这种不确定的情况，建议用户避免这种情况出现：

➲ 示例

intc1、intMax、charc2、charc3都为变量：

```
INSERT INTO mytb(c1, c2) VALUES(intc1,
     CASE
        WHEN intc1 <= intMax THEN charc2
     ELSE charc3
END);
```

15. 不支持类似*set client_char_set 'big5'*的client语法，用户可以使用动态SQL来实现类似语法。

16. 下面是SQL 存储过程的保留关键字，变量名不可以设置为下面的关键字和*SQL命令与函数参考手册*里的保留关键字（可能会有交集）。

    ADD、ALTER、AND、AS、ASENSITIVE、BEGIN、BIGINT、BINARY、BREAK、CALL、CASE、CHAR、 CLOSE、COMMIT、CONDITION、CONNECT、CONT、 CONTINUE、CREATE、CURSOR、DATE、DEALLOCATE、DECIMAL、DECLARE、DEFAULT、DELETE、DISCONNECT、DO、DOUBLE、DROP、 DYNAMIC、ELSE、ELSEIF、END、EXECUTE、EXIT、 FALSE、FETCH、FIRST、FLOAT、FOR、FOUND、FROM、GO、GOTO、GRANT、HANDLER、HOLD、IF、IMMEDIATE、IN、INOUT、INPUT、INSENSITIVE、INSERT、INT、INTEGER、INTO、IS、ITERATE、LANGUAGE、LAST、LEAVE、LOOP、NCHAR、NCLOB、NEXT、NO、NOT、NULL、NVARCHAR、OFF、ON、OPEN、OR、OUT、UTPUT、PREPARE、 PRIOR、PROCEDURE、REPEAT、RESULT、RETURN、 RETURNS、ROLLBACK、SCROLL、 SELECT、SENSITIVE、SET、SETS、SHORT、SMALLINT、SQL、SQLCODE、SQLEXCEPTION、SQLSTATE、SQLWARNING、TATISTICS、STOP、THEN、TIME、TIMESTAMP、TO、TRACE、TRUE、UNTIL、UPDATE、USING、VALUE、VARBINARY、VARBPTR、VARCHAR、VARCPTR、WHEN、WHILE、WITH、WITHOUT

17. 在SQL 存储过程中可以使用反斜杠'\'，若其后紧跟着回车换行，则表示将'\'去掉，同时连接上下两行。

18. INOUT 参数不支持。

19. 在存储过程中不支持长度超过10,240字节的静态SQL语句。