IBM DB2 10.1
for Linux, UNIX, and Windows

SQL 过程语言：应用程序启用和支持

S151-1762-00

IBM DB2 10.1
for Linux, UNIX, and Windows

SQL 过程语言：应用程序启用和支持

S151-1762-00

注意

使用此信息及其支持的产品前，请先阅读第 473 页的附录 B，『声明』下的常规信息。

修订版声明

此文档包含 IBM 的所有权信息。它在许可协议中提供，且受版权法的保护。本出版物中包含的信息不包括对任何产品的保证，且提供的任何语句都不需要如此解释。

您可在线或通过当地的 IBM 代表处订购 IBM 出版物。

- 要在线订购出版物，请转至 IBM 出版物中心，网址为: http://www.ibm.com/shop/publications/order
- 要查找当地的 IBM 代表处，请转至 IBM 全球联系人目录，网址为: http://www.ibm.com/planetwide/

要从美国或加拿大的 DB2 市场和销售部订购 DB2 出版物，请致电 1-800-IBM-4YOU（426-4968）。

您发送信息给 IBM 后，即授予 IBM 非独占权限，IBM 可以按它认为适当的任何方式使用或分发您所提供的任何信息而无须对您承担任何责任。

目录

第 1 部分 SQL 过程语言 (SQL PL)

SQL 过程语言 (SQL PL) 是 SQL 的语言扩展，由可用于在 SQL 语句中实现过程逻辑的语句和语言元素组成。SQL PL 提供了用于声明变量和条件处理程序、对变量赋值以及实现过程逻辑的语句。

第 1 章 内联 SQL PL

内联 SQL PL 是可在复合 SQL（内联型）语句中使用的 SQL PL 功能的子集。复合 SQL（内联型）语句可独立执行，也可用于实现触发器、SQL 函数或 SQL 方法的主体。复合 SQL（内联型）语句处于交互方式以提供对基本 SQL 脚本语言的支持时，可通过 DB2® CLP 独立执行复合 SQL（内联型）语句。

内联 SQL PL 被描述为“内联”是因为该逻辑扩展至引用它们的 SQL 语句并使用这些 SQL 语句执行该逻辑。

以下 SQL PL 语句被视为内联 SQL PL 语句集的一部分:

- 与变量相关的语句
  - DECLARE <variable>
  - DECLARE <condition>
  - SET 语句（赋值语句）
- 条件语句
  - IF
  - CASE 表达式
- 循环语句
  - FOR
  - WHILE
- 控制转移语句
  - GOTO
  - ITERATE
  - LEAVE
  - RETURN
- 错误管理语句
  - SIGNAL
  - GET DIAGNOSTICS

SQL 过程中支持的其他 SQL PL 语句在复合 SQL（内联型）语句中不受支持。游标和条件处理程序在内联 SQL PL 中不受支持，因此 RESIGNAL 语句也不受支持。

因为内联 SQL PL 语句必须在复合 SQL（内联型）语句中执行，因此 PREPARE、EXECUTE 或 EXECUTE IMMEDIATE 语句不受支持。

而且，因为 ATOMIC 必须在动态预编译或执行的复合 SQL（内联型）语句中指定，所以成员语句必须全部成功落实或无一成功落实。因此，COMMIT 和 ROLLBACK 语句也不受支持。

至于 LOOP 和 REPEAT 语句，可使用 WHILE 语句来实现等价逻辑。

带有内联 SQL PL 的独立脚本编制包括在命令行处理器 (CLP) 脚本中或直接在 CLP 提示符下执行动态预编译或执行的复合 SQL（内联型）语句。动态预编译或执行的复合 SQL（内联型）语句由关键字 BEGIN 和 END 绑定，并且必须以非缺省终止符结束。它们可包含 SQL PL 和其他 SQL 语句。

因为内联 SQL PL 语句是在引用它们的 SQL 语句中扩展的，而不是分别编译的，所以有一些小小的性能注意事项，当您规划是在 SQL 过程中以 SQL PL 实现过程逻辑还是通过在动态预编译或执行的函数、触发器或复合 SQL（编译型）语句中使用内联 SQL PL 来实现逻辑过程时应注意这些事项。

第 2 章 SQL 过程中的 SQL PL

SQL PL 语句主要用于 SQL 过程。SQL 过程可包含用于查询和修改数据的基本 SQL 语句，但它们也可包含用于在其他 SQL 语句周围实现控制流逻辑的 SQL PL 语句。可在 SQL 过程中使用一组完整的 SQL PL 语句。

SQL 过程还支持参数、变量、赋值语句、功能强大的条件和错误处理机制、嵌套和递归调用、事务和保存点支持并且能够将多个结果集返回给过程调用程序或客户机应用程序。

SQL PL 在 SQL 过程中使用时使您能够以 SQL 高效编程。SQL PL 的高级语言以及 SQL 过程提供的其他功能使得您能够使用 SQL PL 快速轻松地进行编程。

以下是 SQL 过程中使用的 SQL PL 语句的简单示例:

```
CREATE PROCEDURE UPDATE_SAL (IN empNum CHAR(6),
        INOUT rating SMALLINT)
LANGUAGE  SQL
BEGIN
  IF rating = 1 THEN
   UPDATE employee
      SET salary = salary * 1.10, bonus = 1000
        WHERE empno = empNum;
  ELSEIF rating = 2 THEN
    UPDATE employee
      SET salary = salary * 1.05, bonus = 500
        WHERE empno = empNum;
  ELSE
    UPDATE employee
      SET salary = salary * 1.03, bonus = 0
        WHERE empno = empNum;
  END IF;
END
```

第 3 章 内联 SQL PL 和 SQL 函数、触发器及复合 SQL 语句

可在复合 SQL（编译型）语句、复合 SQL（内联型）语句、SQL 函数和触发器中执行内联 SQL PL 语句。

复合 SQL（内联型）语句是允许您将多个 SQL 语句组合成可选原子块的语句，可在原子块中声明变量及条件处理元素。这些语句由 DB2 编译为单个 SQL 语句，并且可包含内联 SQL PL 语句。

SQL 函数和触发器的主体可包含复合 SQL（内联型）语句，并且还可包括一些内联 SQL PL 语句。

复合 SQL（内联型）语句本身在创建简短脚本时很有用，这些脚本用于执行控制流很小但数据流很大的小型逻辑工作单元。在函数和触发器中，它们允许使用这些对象时执行更复杂的逻辑。

考虑以下包含 SQL PL 的复合 SQL（内联型）语句示例:

```
BEGIN ATOMIC
  FOR row AS
    SELECT pk, c1, discretize(c1) AS v FROM source
  DO
    IF row.v is NULL THEN
      INSERT INTO except VALUES(row.pk, row.c1);
    ELSE
      INSERT INTO target VALUES(row.pk, row.d);
    END IF;
  END FOR;
END
```

复合 SQL（内联型）语句由关键字 BEGIN 和 END 绑定。它包括使用包含在 SQL PL 中的 FOR 和 IF/ELSE 控制语句。FOR 语句用于通过已定义的一组行迭代。对于每行，将检查列值并根据该值有条件地将一组值插入到另一个表中。

考虑以下包含 SQL PL 的触发器示例:

```
CREATE TRIGGER validate_sched
NO CASCADE BEFORE INSERT ON c1_sched
FOR EACH ROW
MODE DB2SQL
Vs: BEGIN ATOMIC

  IF (n.ending IS NULL) THEN
    SET n.ending = n.starting + 1 HOUR;
  END IF;

  IF (n.ending > '21:00') THEN
    SIGNAL SQLSTATE '80000'  SET MESSAGE_TEXT =
           'Class ending time is after 9 PM';
  ELSE IF (n.DAY=1 or n.DAY-7) THEN
    SIGNAL SQLSTATE '80001' SET MESSAGE_TEXT =
           'Class cannot be scheduled on a weekend';
  END IF;
END vs;
```

此触发器将在对表 c1_sched 执行插入操作时激活，并且会使用 SQL PL 来检查并提供类结束时间（如果尚未提供），如果类结束时间在晚上 9 点以后或者该类安排在周末，那么会产生错误。考虑以下包含 SQL PL 的标量 SQL 函数的示例:

```
CREATE FUNCTION GetPrice (Vendor CHAR(20), Pid INT)
RETURNS  DECIMAL(10,3)
LANGUAGE SQL  MODIFIES SQL
BEGIN
  DECLARE price DECIMAL(10,3);

  IF Vendor = 'Vendor 1'
    THEN SET price = (SELECT ProdPrice FROM V1Table WHERE Id = Pid);
  ELSE IF Vendor = 'Vendor 2'
    THEN SET price = (SELECT Price FROM V2Table WHERE Pid = GetPrice.Pid);
  END IF;

  RETURN price;
END
```

此简单函数根据标识供应商的输入参数的值返回标量价格值。它还会使用 IF 语句。

对于需要输出参数的较复杂逻辑，SQL 过程可能更适合传递结果集或其他更高级过程元素。

第 4 章 SQL PL 数据类型

锚定位数据类型

锚定位数据类型是定义为另一对象数据类型的数据类型。如果底层对象数据类型更改，那么锚定位数据类型也会更改。

以下主题提供有关锚定位数据类型的更多信息:

锚定位数据类型的功能

锚点类型根据另一 SQL 对象（如列、全局变量、SQL 变量、SQL 参数或者表或视图的行）定义数据类型。

使用锚点类型定义进行定义的数据类型仍然依赖于它定位至的对象。锚点对象的数据类型中的任何更改将影响锚定位数据类型。如果定位至表或视图的行，那么锚定位数据类型为 ROW，其字段由锚点表或锚点视图的列定义。

声明变量时，如果要求变量与另一对象（例如，表中某列）的数据类型相同但您不知道确切数据类型，那么此数据类型很有用。

锚定位数据类型可以与下列其中一项的类型相同:

- 表中某行
- 视图中某行
- 游标变量行定义
- 表中某列
- 视图中某列
- 局部变量，包括局部游标变量或行变量
- 全局变量

仅当声明或创建下列其中一项时，才能指定锚定位数据类型:

- SQL 过程中的局部变量，包括行变量
- 编译型 SQL 函数中的局部变量，包括行变量
- 例程参数
- 使用 CREATE TYPE 语句的用户定义游标数据类型。
  - 不能在 DECLARE CURSOR 语句中对其进行引用。
- 函数返回数据类型
- 全局变量

要定义锚定位数据类型，请指定 ANCHOR DATA TYPE TO 子句（或更短格式的 ANCHOR 子句）来指定将要成为的数据类型。如果锚定位数据类型是行数据类型，那么必须指定 ANCHOR ROW OF 子句或其同义词之一。这些子句在以下语句中受支持:

- DECLARE
- CREATE TYPE

- CREATE VARIABLE
  - 在此版本中，只能将全局变量定位至其他全局变量、表中某列或视图中某列。

对锚定位数据类型的限制

声明此类型的变量之前或诊断与锚定位数据类型使用有关的问题时，应了解锚定位数据类型使用的限制。

使用锚点数据类型时存在以下限制:

- 不支持在内联 SQL 函数中使用锚定位数据类型。
- 锚定位数据类型不能引用昵称或昵称中的列。
- 锚定位数据类型不能引用类型表、类型表的列、类型视图和类型视图列。
- 锚定位数据类型不能引用已声明临时表或已声明临时表的列。
- 锚定位数据类型不能引用与弱类型游标相关联的行定义。
- 锚定位数据类型不能引用其代码页或整理不同于数据库代码页或数据整理的对象。

锚定位数据类型变量

锚定位变量是数据类型为锚定位数据类型的局部变量或参数。

锚定位变量在以下上下文中受支持:

- SQL 过程
  - 在 SQL 过程中，可将参数和局部变量指定为锚定位数据类型。
- 编译型 SQL 函数
  - 使用 CREATE FUNCTION 语句（该语句指定 BEGIN 子句而不是 BEGIN ATOMIC 子句）创建的 SQL 函数可包括锚定位数据类型的参数或局部变量规范。
- 模块变量
  - 可将锚定位变量指定为模块中定义的已发布或未发布变量。
- 全局变量
  - 可创建锚定位数据类型的全局变量。

锚定位变量通过使用 DECLARE 语句声明。

声明锚定位数据类型的局部变量

每当需要变量或参数的数据类型与对象将定位至的数据类型保持一致时，应声明锚定位数据类型的局部变量或参数。

开始之前

必须定义变量将定位至的数据类型的对象。

过程

1. 创建 DECLARE 语句
   a. 指定变量的名称。
   b. 指定 ANCHOR DATA TYPE TO 子句。
   c. 指定属于变量将定位至的数据类型的对象名称。
2. 从受支持的 DB2 接口执行 DECLARE 语句。

结果

如果 DECLARE 语句成功执行，那么会在数据库中定义具有指定锚点数据类型的变量。

示例

以下是锚定位数据类型声明的示例，其中变量 v1 定位至表 emp 中的列 c1 的数据类型:

```
DECLARE v1 ANCHOR DATA TYPE TO emp.c1;
```

下一步做什么

定义变量后，可对其指定值，可对其进行引用或作为例程的参数传递。

示例: 锚定位数据类型用法

锚定位数据类型用法示例是使用此数据类型时很有用的参考。

以下主题包括锚定位数据类型用法的示例:

示例: 锚定位数据类型的变量声明

声明变量时，锚定位数据类型声明的示例是很有用的参考。

以下是变量 v1 的声明示例，该变量的数据类型与表 staff 中列 name 的数据类型相同:

```
DECLARE v1 ANCHOR DATA TYPE TO staff.name;
```

以下是 CREATE TYPE 语句的示例，该语句定义类型 empRow1，该类型定位至表 employee 中定义的行:

```
CREATE TYPE empRow1 AS ROW ANCHOR DATA TYPE TO ROW OF employee;
```

对于声明为类型 empRow1 的变量，字段名称与表列名称相同。

如果列 name 的数据类型为 VARCHAR(128)，那么变量 v1 也将属于数据类型 VARCHAR(128)。

示例: SQL 例程中的锚定位数据类型用法

创建您自己的 SQL 例程时，可参考 SQL 例程中的锚定位数据类型用法示例。

下面的一组示例演示各种功能及 SQL 例程中的锚定位数据类型用法。所演示锚定位数据类型功能比包含它们的 SQL 例程的功能更多。

以下示例演示定位至表中某列的数据类型的已声明变量:

```
CREATE TABLE tab1(col1 INT, col2 CHAR)@

INSERT INTO tab1 VALUES (1,2)@

INSERT INTO tab1 VALUES (3,4)@

CREATE TABLE tab2 (col1a INT, col2a CHAR)@

CREATE PROCEDURE p1()
BEGIN
  DECLARE var1 ANCHOR tab1.col1;
  SELECT col1 INTO var1 FROM tab1 WHERE col2 = 2;
```

```
  INSERT INTO tab2 VALUES (var1, 'a');
END@

CALL p1()@
```

调用过程 p1 时，特定行的列 col1 的值会被选择到相同类型的变量 var1 中。

以下 CLP 脚本包括某个函数的示例，其中演示函数参数的锚定位数据类型用法:

```
-- Create a table with multiple columns
CREATE TABLE tt1 (c1 VARCHAR(18), c2 CHAR(8), c3 INT, c4 FLOAT)
@

INSERT INTO tt1 VALUES ('aaabbb', 'ab', 1, 1.1)
@

INSERT INTO tt1 VALUES ('cccddd', 'cd', 2, 2.2)
@

SELECT c1, c2, c3, c4 FROM tt1
@

-- Creation of the function
CREATE FUNCTION func_a(p1 ANCHOR tt1.c3)
 RETURNS INT
 BEGIN
   RETURN p1 + 1;
 END
@

-- Invocation of the function
SELECT c1, c2 FROM tt1 WHERE c3 = func_a(2)
@

-- Another invocation of the function
SELECT c1, c2 FROM tt1 WHERE c3 = func_a(1)
@

DROP FUNCTION func_a
@

DROP TABLE tt1
@
```

调用函数 func_a 时，该函数会使用锚定位数据类型参数的值执行基本操作。

行类型

行数据类型是包含指定字段的有序序列的用户定义的类型，每个字段具有关联数据类型。

行类型可用作 SQL PL 中全局变量、SQL 变量及 SQL 参数的类型，以灵活地操作数据行中的列（通常使用查询进行检索）。

行数据类型的功能

行数据类型的功能使其在简化 SQL PL 代码时非常有用。

仅支持行数据类型与 SQL 过程语言配合使用。它是由多个字段组成的结构，每个字段都有自己的名称和数据类型，可用于将行的列值存储在结果集或其他相似格式的数据中。

此数据类型可用于:

- 简化 SQL 过程语言应用程序中的逻辑编码。例如，数据库应用程序一次处理一个记录，并且需要参数和变量来临时存储记录。单个行数据类型可替换处理和存储记录值本来所需的多个参数和变量。这极大地简化了在应用程序和例程中将行值作为参数传递的能力。
- 使移植至使用支持类似数据类型的其他过程 SQL 语言所编写代码的 DB2 SQL PL 更加容易。
- 在包括以下各项的数据更改语句和查询中引用行数据: INSERT 语句、FETCH 语句、VALUES INTO 语句和 SELECT INTO 语句。

必须使用 CREATE TYPE (ROW) 语句创建行数据类型。创建后，可使用 DECLARE 语句在 SQL PL 上下文中声明已定义数据类型的变量。然后可使用这些变量来存储行类型的值。

可使用单点（“.”）表示法来显式指定和引用行字段值。

对行数据类型的限制

使用行数据类型之前或诊断可能与使用行数据类型有关的问题时，应注意使用行数据类型的限制。

行数据类型存在以下限制:

- 行数据类型中支持的最大字段数为 1012。
- 不能将行数据类型作为输入参数值从 CLP 传递至过程和函数。
- 不能将行数据类型作为输入-输出或输出参数值从过程和函数传递至 CLP。
- 不能直接比较行数据类型变量。要比较两个行类型变量，可比较每个字段。
- 不支持对行字段使用以下数据类型:
  - XML 数据类型
  - LONG VARCHAR
  - LONG VARGRAPHIC
  - 结构化数据类型
  - 行数据类型
  - 数组数据类型
- 不支持以下行类型的全局变量: 该行包含类型为 LOB 的一个或多个字段。
- 不支持使用 CAST 函数将参数值强制转换为行数据类型。

可能存在与使用数据类型、授权、执行 SQL、数据类型的使用范围或其他原因有关的其他常规限制。

行变量

行变量是基于用户定义行数据类型的变量。可声明行变量、对其指定值、将其设置为另一个值或作为参数传递至 SQL 过程或自 SQL 过程传递。行变量继承它们所基于的行数据类型的属性。行变量用于保存结果集中的数据行或被指定其他元组格式的数据。

可使用 DECLARE 语句在 SQL 过程中声明行变量。

创建行变量

要创建行变量，必须先创建行类型，然后声明行变量。

以下主题说明如何创建行数据类型和变量:

创建行数据类型

创建行数据类型是创建行变量的先决条件。

创建行数据类型之前:

- 读取: 第 12 页的『行类型』
- 读取: 第 13 页的『对行数据类型的限制』

可从支持执行 CREATE TYPE 语句的任何接口执行此任务。

要在数据库中创建行数据类型，必须从任何支持执行 SQL 语句的 DB2 接口成功执行 CREATE TYPE (ROWR) 语句。

1. 创建 CREATE TYPE (ROW) 语句:
   a. 指定类型的名称。
   b. 通过对行中每个字段指定名称和数据类型来对行指定行字段定义。

   以下是有关如何创建可与以下结果集相关联的行数据类型的示例，该结果集的格式与 empRow 行数据类型相同:

   ```
   CREATE TYPE empRow AS ROW (name VARCHAR(128), id VARCHAR(8));
   ```
2. 从受支持的 DB2 接口执行 CREATE TYPE 语句。

如果 CREATE TYPE 语句成功执行，那么会在数据库中创建行数据类型。如果该语句未成功执行，请验证该语句的语法并验证该数据类型是否尚未存在。

创建行数据类型之后，可根据此数据类型声明行变量。

声明行类型的局部变量

创建行数据类型后，可声明行类型的变量。

创建行数据类型之前:

- 读取: 第 12 页的『行类型』
- 读取: 第 13 页的『对行数据类型的限制』

只能在 SQL PL 上下文（包括支持执行 DECLARE 语句的 SQL 过程和函数）中声明行数据类型变量。

必须遵循下列步骤来声明局部行变量:

1. 创建 DECLARE 语句:
   a. 指定变量的名称。
   b. 指定将定义该变量的行数据类型。必须已在数据库中定义指定的行数据类型。

   以下是有关如何创建 DECLARE 语句的示例，该语句将定义类型为 empRow 的行变量:

   ```
   DECLARE r1 empRow;
   ```
2. 在受支持上下文中执行 DECLARE 语句。

如果 DECLARE 语句成功执行，那么会创建行变量。

创建行变量时，该行中的每个字段会初始化为 NULL 值。

可对行变量指定值、对其进行引用或作为参数传递。

将值指定给行变量

可以多种方式将值指定给类型为行的变量。可将一个行变量值指定给另一个行变量。可指定和引用变量字段值。可使用单点“.”表示法来引用行的字段值。

以下主题显示如何将值指定给行类型变量及行类型变量数组:

受支持的行数据类型赋值

支持对行和行字段指定各种值。

声明行变量或参数时，行中的每个字段的缺省值为 NULL，直到对其指定值。

可将下列类型的值指定给行变量:

- 行数据类型相同的另一行变量（使用 SET 语句）
  - 行变量值只能指定给行变量（如果它们是兼容类型）。如果两个行变量都属于同一行数据类型或者源行类型被定位至表或视图定义，那么这两个行变量可兼容。为使两个变量的类型可兼容，具有相同字段名称和字段数据类型是不够的。

    例如，如果创建了行数据类型 row1 和 row2，并且它们的定义完全相同，那么类型为 row1 的变量值不能指定给类型为 row2 的变量。类型为 row2 的变量值也不能指定给类型为 row1 的变量。但是，类型为 row1 的变量 v1 的值可指定给类型为 row1 的变量 v2。
- 元素数目与该行相同，并且元素数据类型与行字段相同的元组。
  - 以下是要指定给行的字面值元组的示例:

    ```
    SET v1 = (1, 'abc')
    ```
- 解析为行值的表达式
  - 解析为可指定给行变量的行值的表达式在 VALUES ... INTO 语句中是解析型表达式。以下是这类赋值的示例:

    ```
    VALUES (1, 'abc') INTO rv1
    ```
- 函数的返回类型（如果其行数据类型与目标变量相同）:
  - 以下是函数 foo 的返回类型与目标变量属于相同行数据类型的示例:

    ```
    SET v1 = foo()
    ```

    如果返回数据类型定义为锚定位数据类型，那么锚定位数据类型赋值规则适用。
- 查询的单个行结果集
  - 结果集的元素数目必须与该行相同，并且必须可将列指定给与该行字段相同的数据类型。以下是此类型赋值的示例:

    ```
    SET v1 = (select c1, c2 from T)
    ```
- NULL
  - 将 NULL 指定给行变量时，所有行字段都将设置为 NULL，但行变量本身仍然不为 NULL。

可将以下类型的值指定给行变量字段:

- 字面值
- 参数
- 变量
- 表达式
- NULL

可通过下列方式向行字段值指定值:

- 使用 SET 语句
- 使用解析为行值的 SELECT INTO 语句
- 使用解析为行值的 FETCH INTO 语句
- 使用解析为行值的 VALUES INTO 语句

可将 ROW 数据类型指定为 SQL 标量函数的返回类型。

使用 SET 语句将值指定给行变量

可使用 SET 语句将值指定给行变量。可将行值指定给行变量。可将行字段值或表达式指定给行字段。

如果行值和行变量属于同一用户定义数据类型，那么可使用 SET 语句将行值指定给行变量。

以下是有关如何将行值指定给同一格式的行变量的示例:

```
SET empRow = newHire;
```

行值 newHire 的格式与 empRow 变量相同，即，行字段的数目和类型完全相同:

```
empRow.lastName      /* VARCHAR(128) */
empRow.firstName     /* VARCHAR(128) */
empRow.id            /* VARCHAR(10)  */
empRow.hireDate      /* TIMESTAMP    */
empRow.dept          /* VARCHAR(3)   */

newHire.lastName     /* VARCHAR(128) */
newHire.firstName    /* VARCHAR(128) */
newHire.id           /* VARCHAR(10)  */
newHire.hireDate     /* TIMESTAMP    */
newHire.dept         /* VARCHAR(3)   */
```

如果尝试将行值指定给没有相同格式的变量，那么会发生错误。

可通过将值指定给行中的个别字段来指定行值。以下是有关如何使用 SET 语句将值指定给行 empRow 中的字段的示例:

```
SET empRow.lastName = 'Brown';           // Literal value assignment

SET empRow.firstName = parmFirstName;    // Parameter value of same type assignment

SET empRow.id = var1;                    // Local variable of same type assignment

SET empRow.hiredate = CURRENT_TIMESTAMP; // Special register expression assignment

SET empRow.dept = NULL;                  // NULL value assignment
```

可使用任何受支持的字段来初始化行值。

使用 SELECT、VALUES 或 FETCH 语句将行值指定给行变量

可使用 SELECT INTO 语句、VALUES INTO 语句或 FETCH INTO 语句将行值指定给类型为行的变量。源行值的字段值必须可指定给目标行变量的字段值。

以下是有关如何使用 SELECT 语句将单个行值指定给行变量 empRow 的示例:

```
SELECT * FROM employee
INTO empRow
WHERE id=5;
```

如果 SELTCT 查询解析为多个行值，那么会发生错误。

以下是有关如何使用 VALUES INTO 语句将行值指定给行变量 empEmpBasics 的示例:

```
VALUES (5, 'Jane Doe', 10000) INTO empBasics;
```

以下是有关如何使用 FETCH 语句将行值指定给行变量 empRow 的示例，该 FETCH 语句引用游标 cur1，该游标将带有可兼容字段值的行定义为变量 empRow:

```
FETCH cur1 INTO empRow;
```

可将其他变体与其中每个语句一起使用。

比较行变量和行字段值

即使行变量属于同一行数据类型，也不能直接比较行变量，但可比较个别行字段。

行类型中的个别字段可与其他值进行比较，并且该字段的数据类型的比较规则适用。

要比较两个行变量，必须比较个别对应字段值。

以下是使用 SQL PL 中的可比较字段定义比较两个行值的示例:

```
IF ROW1.field1 = ROW2.field1 AND
   ROW1.field2 = ROW2.field2 AND
   ROW1.field3 = ROW2.field3
THEN
   SET EQUAL = 1;
ELSE
   SET EQUAL = 0;
```

在该示例中，IF 语句用于执行过程逻辑，如果字段值相等，那么此逻辑将局部变量 EQUAL 设置为 1，如果字段值不相等，那么此逻辑将局部变量 EQUAL 设置为 0。

引用行值

可在 SQL 和 SQL 语句中引用行值和行字段值。

以下主题演示可引用行值的位置和引用方式:

引用行变量

可在支持行变量数据类型引用的位置通过名称引用行变量值。

受支持的行变量引用上下文包括下列各项:

- SET 语句的源或目标
- INSERT 语句
- SELECT INTO、VALUES INTO 或 FETCH 语句的目标

以下是要使用 SET 语句指定给具有相同定义的另一行变量的行变量示例:

```
-- Referencing row variables as source and
   target of a SET statement
SET v1 = v2;
```

以下是要在插入两行的 INSERT 语句中引用的行变量示例。行变量 v1 和 v2 具有字段定义，并且其类型与 INSERT 语句目标的表的列定义兼容:

```
-- Referencing row variables in an INSERT statement
INSERT INTO employee VALUES v1, v2;
```

以下是 FETCH 语句中引用的行变量的示例。行变量 empRow 的列定义与游标 c1 的关联数据集相同:

```
-- Referencing row variables in a FETCH statement
FETCH c1 INTO empRow;
```

以下是要在 SELECT 语句中引用的行变量 v3 的示例。employee 表中的两个列值将选择到变量 v3 的两个字段中:

```
-- Referencing row variables in a SELECT statement
SELECT id, name INTO v3 FROM employee;
```

在行变量中引用字段

可在多个上下文中引用字段值。

可在允许使用字段数据类型值的任何位置引用行字段值。支持在以下上下文中引用行字段:

- 允许使用字段数据类型值的位置包括但不限于:
  - 赋值的源（SET 语句）
  - 赋值的目标（SET 语句）
  - SELECT INTO、VALUES INTO 或 FETCH INTO 语句的目标。

为了在行变量中引用字段值，使用了单点表示法。与变量相关联的字段值如下所示:

```
<row-variable-name>.<field-name>
```

以下是有关如何访问变量 employee 的字段 id 的示例:

```
employee.id
```

下面是行变量字段值的受支持引用。

以下示例说明如何将字面值指定给行变量 v1 中的字段:

```
-- Literal assignment to a row variable field
SET v1.c1 = 5;
```

以下示例说明如何将字面值和表达式值指定给多个行变量字段:

```
-- Literal assignment to fields of row variable
SET (emp.id, emp.name) = (v1.c1 + 1, 'James');
```

以下示例说明如何在 INSERT 语句中引用字段值:

```
-- Field references in an INSERT statement
INSERT INTO employee
VALUES(v1.c1, 'Beth'),
      (emp.id, emp.name);
```

以下示例说明如何在 UPDATE 语句中引用字段值:

```
-- Field references in an UPDATE statement
UPDATE employee
SET name = 'Susan'
WHERE id = v1.c1;
```

以下示例说明如何在 SELECT INTO 语句中引用字段值:

```
-- Field references in a SELECT INTO statement
SELECT employee.firstname INTO v2.c1
FROM employee
WHERE name=emp.name;
```

在 INSERT 中引用行变量 语句

可在 INSERT 语句中使用行变量以追加或修改整个表行。

以下是在表 employee 中插入一行的 INSERT 语句的示例:

```
INSERT INTO employee VALUES empRow;
```

对于 INSERT 语句，行变量中的字段数必须与隐式或显式目标列列表中的列数相匹配。

上面显示的 INSERT 语句对表中每列插入相应的行字段值。因此上面显示的 INSERT 语句等价于以下 INSERT 语句:

```
INSERT INTO employee VALUES (emp.id,
                             emp.name,
                             emp.salary,
                             emp.phone);
```

将行作为例程参数传递

行类型值和行类型变量数组可作为过程和函数的参数传递。过程支持将这些数据类型用作 IN、OUT 和 INOUT 参数。

以下是将 CHAR 类型用作输入参数，修改输出行参数中的字段并返回的过程的示例。

```
CREATE PROCEDURE p(IN basicChar CHAR, OUT outEmpRow empRow)
BEGIN

  SET outEmpRow.field2 = basicChar;

END@
```

以下是调用过程的 CALL 语句的示例:

```
CALL p('1', myEmpRow)@
```

删除行数据类型

不再需要行数据类型或者要重复使用现有行数据类型的名称时删除行数据类型。

开始之前

必须先满足下列先决条件，才能删除行数据类型:

- 必须建立与数据库的连接。
- 数据库中必须存在行数据类型。

关于此任务

不再需要行数据类型或者要重复使用现有行数据类型的名称时删除行数据类型。可从支持执行 DROP 语句的任何接口删除行。

过程

1. 创建 DROP 语句，该语句应指定要删除的行数据类型的名称。
2. 从受支持的 DB2 接口执行 DROP 语句。

示例

以下是有关如何删除行数据类型 simpleRow 的示例。

```
DROP TYPE simpleRow;
```

下一步做什么

如果 DROP 语句成功执行，那么就从数据库中删除了该行数据类型。

示例: 行数据类型用法

行数据类型用法示例为理解行数据类型的使用方式和使用时间提供了很有用的参考。

以下主题演示如何使用行数据类型:

示例: CLP 脚本中的行数据类型用法

DB2 CLP 脚本中显示了行数据类型的一些基本功能，用于演示行数据类型的最常用用法。

以下 DB2 CLP 脚本演示行数据类型的用法及其相关操作。它包括下列各项的演示:

- 创建行数据类型
- 创建表
- 创建包括下列各项的过程:
  - 行数据类型声明
  - 将值插入到包括某些行字段值的类型中
  - 根据行字段值更新行值
  - 将一些值选择到行字段值中
  - 对行指定行值
  - 向参数指定行字段值
- 调用过程
- 删除行数据类型和表

```
-- Creating row types

CREATE TYPE row01 AS ROW (c1 INTEGER)@

CREATE TYPE empRow AS ROW (id INTEGER, name VARCHAR(10))@

CREATE TABLE employee (id INTEGER, name VARCHAR(10))@

CREATE procedure proc01 (OUT p0 INTEGER, OUT p1 INTEGER)
BEGIN
  DECLARE v1, v2 row01;
```

```
  DECLARE emp empRow;

  -- Assigning values to row fields
  SET v1.c1 = 5;
  SET (emp.id, emp.name) = (v1.c1 + 1, 'James');

  -- Using row fields in DML
  INSERT INTO employee
  VALUES (v1.c1, 'Beth'), (emp.id, emp.name);

  UPDATE employee
  SET name = 'Susan' where id = v1.c1;

  -- SELECT INTO a row field
  SELECT id INTO v2.c1
  FROM employee
  WHERE name = emp.name;

  -- Row level assignment
  SET v1 = v2;

  -- Assignment to parameters
  SET (p0, p1) = (v1.c1, emp.id);

END@

CALL proc01(?, ?)@

SELECT * FROM employee@

DROP procedure proc01@

DROP TABLE employee@

-- Dropping row types
DROP TYPE empRow@

DROP TYPE row01@
```

可通过 DB2 命令行发出以下命令来保存和运行此脚本:

```
DB2 -td@ -vf <filename>;
```

以下是运行脚本的输出:

```
CREATE TYPE row01 AS ROW (c1 INTEGER)
DB20000I  The SQL command completed successfully.

CREATE TYPE empRow AS ROW (id INTEGER, name VARCHAR(10))
DB20000I  The SQL command completed successfully.

CREATE TABLE employee (id INTEGER, name VARCHAR(10))
DB20000I  The SQL command completed successfully.

CREATE procedure proc01 (OUT p0 INTEGER, OUT p1 INTEGER)
   BEGIN DECLARE v1, v2 row01;
   DECLARE emp empRow;
   SET v1.c1 = 5;
   SET (emp.id, emp.name) = (v1.c1 + 1, 'James');
   INSERT INTO employee  VALUES (v1.c1, 'Beth'), (emp.id, emp.name);
   UPDATE employee  SET name = 'Susan' where id = v1.c1;
   SELECT id INTO v2.c1 FROM employee WHERE name = emp.name;
   SET v1 = v2;
   SET (p0, p1) = (v1.c1, emp.id);
END

DB20000I  The SQL command completed successfully.
```

```
CALL proc01(?, ?)

  Value of output parameters
  --------------------------
  Parameter Name  : P0
  Parameter Value : 6

  Parameter Name  : P1
  Parameter Value : 6

  Return Status = 0

SELECT * FROM employee

ID          NAME
----------- ----------
          5 Susan
          6 James

  2 record(s) selected.


DROP procedure proc01
DB20000I  The SQL command completed successfully.

DROP TABLE employee
DB20000I  The SQL command completed successfully.

DROP TYPE empRow
DB20000I  The SQL command completed successfully.

DROP TYPE row01
DB20000I  The SQL command completed successfully.
```

示例: SQL 过程中的行数据类型用法

可在 SQL 过程中使用行数据类型来检索记录数据并将其作为参数传递。

本主题包含 CLP 脚本的示例，其中包含多个 SQL 过程的定义，这些定义演示了行的多种用法的其中一些。

过程 ADD_EMP 将行数据类型用作输入参数，然后将其插入到表中。

过程 NEW_HIRE 使用 SET 语句将行指定给行变量，然后将行数据类型值作为调用另一过程的 CALL 语句中的参数传递。

过程 FIRE_EMP 会将一行表数据选择到行变量中，然后将行字段值插入到表中。

以下是 CLP 脚本，后跟以详细方式从 CLP 运行脚本的输出:

```
--#SET TERMINATOR @;
CREATE TABLE employee (id INT,
                       name VARCHAR(10),
                       salary DECIMAL(9,2))@

INSERT INTO employee VALUES (1, 'Mike', 35000),
                            (2, 'Susan', 35000)@

CREATE TABLE former_employee (id INT, name VARCHAR(10))@

CREATE TYPE empRow AS ROW ANCHOR ROW OF employee@
CREATE PROCEDURE ADD_EMP (IN newEmp empRow)
BEGIN
  INSERT INTO employee VALUES newEmp;
```

```
END@

CREATE PROCEDURE NEW_HIRE (IN newName VARCHAR(10))
BEGIN
  DECLARE newEmp empRow;
  DECLARE maxID INT;

  -- Find the current maximum ID;
  SELECT MAX(id) INTO maxID FROM employee;

  SET (newEmp.id, newEmp.name, newEmp.salary)
    = (maxID + 1, newName, 30000);

  -- Call a procedure to insert the new employee
  CALL ADD_EMP (newEmp);
END@

CREATE PROCEDURE FIRE_EMP (IN empID INT)
BEGIN
  DECLARE emp empRow;

  -- SELECT INTO a row variable
  SELECT * INTO emp FROM employee WHERE id = empID;

  DELETE FROM employee WHERE id = empID;

  INSERT INTO former_employee VALUES (emp.id, emp.name);
END@

CALL NEW_HIRE('Adam')@

CALL FIRE_EMP(1)@

SELECT * FROM employee@

SELECT * FROM former_employee@
```

以下是以详细方式从 CLP 运行脚本的输出:

```
CREATE TABLE employee (id INT, name VARCHAR(10), salary DECIMAL(9,2))
DB20000I  The SQL command completed successfully.

INSERT INTO employee VALUES (1, 'Mike', 35000), (2, 'Susan', 35000)
DB20000I  The SQL command completed successfully.

CREATE TABLE former_employee (id INT, name VARCHAR(10))
DB20000I  The SQL command completed successfully.

CREATE TYPE empRow AS ROW ANCHOR ROW OF employee
DB20000I  The SQL command completed successfully.

CREATE PROCEDURE ADD_EMP (IN newEmp empRow)
BEGIN
  INSERT INTO employee VALUES newEmp;
END
DB20000I  The SQL command completed successfully.

CREATE PROCEDURE NEW_HIRE (IN newName VARCHAR(10))
BEGIN
  DECLARE newEmp empRow;
  DECLARE maxID INT;

  -- Find the current maximum ID;
  SELECT MAX(id) INTO maxID FROM employee;

  SET (newEmp.id, newEmp.name, newEmp.salary) = (maxID + 1, newName, 30000);
```

```
  -- Call a procedure to insert the new employee
  CALL ADD_EMP (newEmp);
END
DB20000I  The SQL command completed successfully.

CREATE PROCEDURE FIRE_EMPLOYEE (IN empID INT)
BEGIN
  DECLARE emp empRow;

  -- SELECT INTO a row variable
  SELECT * INTO emp FROM employee WHERE id = empID;

  DELETE FROM employee WHERE id = empID;

  INSERT INTO former_employee VALUES (emp.id, emp.name);
END
DB20000I  The SQL command completed successfully.

CALL NEW_HIRE('Adam')

  Return Status = 0

CALL FIRE_EMPLOYEE(1)

  Return Status = 0

SELECT * FROM employee

ID          NAME       SALARY
----------- ---------- -----------
          2 Susan         35000.00
          3 Adam          30000.00

  2 record(s) selected.


SELECT * FROM former_employee

ID          NAME
----------- ----------
          1 Mike

  1 record(s) selected.
```

示例: SQL 函数中的行数据类型用法

可在 SQL 函数中使用行数据类型来构造、存储或修改记录数据。

可使用基于行数据类型的变量轻松保存格式与表相同的行值。用于此用途时，第一次使用时初始化行变量很有帮助。

以下是 DB2 CLP 脚本的示例，其中包含用于创建表、行数据类型以及包含行变量声明、行引用和 UDF 调用的函数的 SQL 语句:

```
CREATE TABLE t1 (deptNo VARCHAR(3),
                 reportNo VARCHAR(3),
                 deptName VARCHAR(29),
                 mgrNo VARCHAR (8),
                 location VARCHAR(128))@

 INSERT INTO t1 VALUES ('123', 'MM1', 'Sales-1', '0112345', 'Miami')@
 INSERT INTO t1 VALUES ('456', 'MM2', 'Sales-2', '0221345', 'Chicago')@
 INSERT INTO t1 VALUES ('789', 'MM3', 'Marketing-1', '0331299', 'Toronto')@

 CREATE TYPE deptRow AS ROW (r_deptNo VARCHAR(3),
                             r_reportNo VARCHAR(3),
```

```
                              r_depTName VARCHAR(29),
                              r_mgrNo VARCHAR (8),
                              r_location VARCHAR(128))@

CREATE FUNCTION getLocation(theDeptNo VARCHAR(3),
                             reportNo VARCHAR(3),
                             theName VARCHAR(29))
 RETURNS VARCHAR(128)
 BEGIN

   -- Declare a row variable
   DECLARE dept deptRow;

   -- Assign values to the fields of the row variable
   SET dept.r_deptno = theDeptNo;
   SET dept.r_reportNo = reportNo;
   SET dept.r_deptname = theName;
   SET dept.r_mgrno = '';
   SET dept.r_location = '';

   RETURN
     (SELECT location FROM t1 WHERE deptNo = dept.r_deptno);

 END@

VALUES (getLocation ('789', 'MM3','Marketing-1'))@
```

执行时，此 CLP 脚本会创建表、在表中插入行、创建行数据类型和 UDF。

函数 getLocation 是一个 SQL UDF，它声明行变量并使用输入参数值以向其字段指定值。它引用 SELECT 语句的行变量中的一个字段值，该字段定义该函数返回的标量值。

在脚本结尾执行 VALUES 语句时，会调用 UDF 并返回标量返回值。

以下是从 CLP 运行此脚本的输出:

```
CREATE TABLE t1 (deptNo VARCHAR(3), reportNo VARCHAR(3),
 deptName VARCHAR(29), mgrNo VARCHAR (8), location VARCHAR(128))
DB20000I  The SQL command completed successfully.

INSERT INTO t1 VALUES ('123', 'MM1', 'Sales-1', '0112345', 'Miami')
DB20000I  The SQL command completed successfully.

INSERT INTO t1 VALUES ('456', 'MM2', 'Sales-2', '0221345', 'Chicago')
DB20000I  The SQL command completed successfully.

INSERT INTO t1 VALUES ('789', 'MM3', 'Marketing-1', '0331299', 'Toronto')
DB20000I  The SQL command completed successfully.

CREATE TYPE deptRow AS ROW (r_deptNo VARCHAR(3), r_reportNo VARCHAR(3), r_depTNa
me VARCHAR(29), r_mgrNo VARCHAR (8), r_location VARCHAR(128))
DB20000I  The SQL command completed successfully.

CREATE FUNCTION getLocation(theDeptNo VARCHAR(3),
                            reportNo VARCHAR(3),
                            theName VARCHAR(29))
 RETURNS VARCHAR(128)
 BEGIN
   DECLARE dept deptRow;
   SET dept.r_deptno = theDeptNo;
   SET dept.r_reportNo = reportNo;
   SET dept.r_deptname = theName;
   SET dept.r_mgrno = '';
   SET dept.r_location = '';
 RETURN
   (SELECT location FROM t1 WHERE deptNo = dept.r_deptno);
```

```
 END
DB20000I  The SQL command completed successfully.

VALUES (getLocation ('789', 'MM3','Marketing-1'))

1

------------------------------------------------------------------------------
----------------------------------------------
Toronto


  1 record(s) selected.
```

数组类型

数组类型是一种用户定义数据类型，由单个数据类型的元素的有序集合组成。

*普通*数组类型的元素数目具有已定义上限，并且该类型使用有序位置作为数组下标。

*关联*数组类型的元素数据没有特定上限，并且每个元素具有关联下标值。下标值的数据类型可以是整数或字符串，但对于整个数组应该属于同一数据类型。

数组类型可用作 SQL PL 中全局变量、SQL 变量及 SQL 参数的类型，以灵活地操作单个数据类型的值的集合。

比较数组和关联数组

简单数组和关联数组有多处不同。了解它们之间的差别可帮助您选择使用正确的数据类型。

下表重点说明数组与关联数组之间的差别:

表 1. 比较数组和关联数组

数组	关联数组
简单数组的最大基数是定义简单数组时定义的。对下标 N 指定值时，下标在数组的当前基数与 N 之间的元素被隐式初始化为 NULL。	声明关联数组变量时，没有用户指定的最大基数，也未初始化任何元素。最大基数受可用内存限制。
简单数组的下标数据类型必须是整数值。	关联数组的下标类型可以是一组受支持数据类型。
简单数组中的下标值必须是一组连续的整数值。	在关联数组中，下标值可以是稀疏的。
针对简单数组的 CREATE TYPE 语句不需要指定数组基数。例如，在以下语句中未指定任何基数: `CREATE TYPE simple AS INTEGER ARRAY[];`	在关联数组的 CREATE TYPE 语句中，需要下标数据类型，但不需要指定数组基数。例如，在此语句中，下标数据类型的基数被指定为 INTEGER: `CREATE TYPE assoc AS INTEGER ARRAY[INTEGER];`

表 1. 比较数组和关联数组 (续)

数组	关联数组

对简单数组的第一次赋值会导致下标值在 1 与指定给数组的下标值之间的数组元素初始化。以下复合 SQL（编译型）语句包含简单数组变量的声明以对变量的赋值:
`BEGIN`
` DECLARE mySimpleA simple;`
`  SET mySimpleA[100] = 123;`
`END`
执行赋值语句后，mySimpleA 的基数为 100; 下标值在 1 到 99 的元素被隐式初始化为 NULL。
	对关联数组的第一次赋值导致具有单个下标值的单个元素初始化。以下复合 SQL（编译型）语句包含关联数组变量的声明以及对变量的赋值:
	`BEGIN`
	`  DECLARE myAssocA assoc;`
	`  SET myAssocA[100] = 123;`
	`END`
	执行赋值语句后，数组的基数为 1。

示例

常规数组数据类型

常规数组数据类型是包含数据元素的有序集合的结构，在此结构中，可通过每个元素在集合中的顺序位置引用该元素。

如果 N 是数组中的基数（元素数目），那么与每个元素相关联的有序位置（称为下标）是大于或等于 1 并且小于或等于 N 的整数值。数组中的所有元素都具有相同数据类型。

数组数据类型的功能

数组数据类型的许多功能使其很适合用于 SQL PL 逻辑。

数组类型是定义为另一数据类型的数组的数据类型。

每个数组类型都有最大基数，此基数在 CREATE TYPE 语句上指定。如果 A 是数组类型并且最大基数为 M，那么类型为 A 的值的基数可以是 0 到 M 之间的任何值。与编程语言（如 C）中的数组的最大基数不同，SQL 数组的最大基数与其物理表示无关。系统在运行时使用最大基数来确保下标在预定界限内。表示数组值所需的内存量通常与其基数成一定比例，而与其类型的最大基数不成比例。

引用数组时，数组中的所有值都存储在主存储器中。因此，包含大量数据的数组将消耗大量主存储器空间。

可通过指定元素的对应下标值来检索数组元素值。

想要存储单个数据类型的一组值时，数组数据类型很有用。可使用这一组值来极大简化将值传递至例程的过程，原因是可传递单个数组值来代替多个（可能很多）参数。

数组数据类型与关联数组数据类型不同。数组数据类型是简单值集合，而关联数组从概念上讲类似一组数组。即，关联数组是包含零个或零个以上子数组元素的有序数组，这样可通过主下标来访问数组元素，并通过子下标访问子数组元素。

对数组数据类型的限制

使用数组数据类型之前或诊断因为声明数组数据类型而导致的问题时，应注意数组数据类型的限制。

数组数据类型存在以下限制:

- 不支持在动态复合语句中使用数组数据类型。
- 不支持在 SQL 过程外部使用 ARRAY_AGG 函数。
- 不支持在 SQL 过程外部使用 UNNEST 函数。
- 不支持在 Java 过程以外的外部过程中使用数组数据类型的参数。
- 不支持将数组强制转换为用户定义数组数据类型以外的任何数组类型。
- 不支持包含并非对数组指定的数据类型的元素。
- 不支持强制转换基数大于目标数组的数组。
- 不支持在方法中使用数组作为参数或返回类型。
- 不支持在有源函数或模板函数中使用数组作为参数或返回类型。
- 不支持在外部标量函数或外部表函数中使用数组作为参数或返回类型。
- 不支持在 SQL 标量函数、SQL 表函数或 SQL 行函数中使用数组作为参数或返回类型。
- 不支持将 TRIM_ARRAY 函数的值指定或强制转换为数组以外的任何数据类型。
- 不支持将 ARRAY 构造函数或 ARRAY_AGG 函数的结果值指定或强制转换为数组以外的任何数据类型。
-

数组变量

数组变量是基于用户定义数组数据类型的变量。可声明数组变量、对其指定值、将其设置为另一个值或作为参数传递至 SQL 过程或自 SQL 过程传递。

数组变量继承它们所基于的数组数据类型的属性。数组变量用于保存同一数据类型的一组数据。

可使用 DECLARE 语句在 SQL 过程中声明局部数组变量。

可使用 CREATE VARIABLE 语句创建全局数组变量。

创建数组变量

要创建数组变量，必须先创建数组类型，然后声明局部数组变量或创建全局数组变量。

创建数组数据类型（CREATE TYPE 语句）:

在创建数组数据类型的变量之前，应创建数组数据类型。

开始之前

创建数组数据类型之前，请确保您具有执行 CREATE TYPE 语句所需的特权。

关于此任务

只能在支持 CREATE TYPE 语句的 SQL PL 上下文中创建数组数据类型。

限制

请参阅: 第 27 页的『对数组数据类型的限制』

过程

1. 定义 CREATE TYPE 语句
   a. 指定数组数据类型的名称。
   b. 指定 AS 关键字，后跟数组元素的数据类型的关键字名称。例如，INTEGER 和 VARCHAR。
   c. 指定 ARRAY 关键字以及数组中子下标的域。例如，如果指定 100，那么有效下标为 1 到 100。 此数字与数组的基数（数组中的元素数目）相同。
2. 从受支持接口执行 CREATE TYPE 语句。

结果

CREATE TYPE 语句应成功执行，并且应创建数组类型。

示例

示例 1:

```
CREATE TYPE simpleArray AS INTEGER ARRAY[100];
```

此数组数据类型最多可包含 100 个整数值，下标为 1 到 100 的整数值。

示例 2:

```
CREATE TYPE id_Phone AS VARCHAR(20) ARRAY[100];
```

此数组数据类型最多可包含 100 个存储为 VARCHAR(20) 数据类型值的电话值，下标为 1 到 100 的整数值。

下一步做什么

创建数组数据类型后，可声明数组变量。

声明数组类型的局部变量:

创建数组数据类型后，如果要临时存储或传递数组数据类型值，应声明数组数据类型变量。

开始之前

创建行类型局部变量之前:

- 读取: 数组数据类型
- 读取: 第 27 页的『对数组数据类型的限制』
- 读取: 第 28 页的『创建数组数据类型（CREATE TYPE 语句）』
- 确保您具有执行 DECLARE 语句所需的特权。

关于此任务

可在受支持上下文中声明数组数据类型，包括以下上下文: SQL 过程、SQL 函数和触发器。

过程

1. 定义 DECLARE 语句。
   a. 指定数组数据类型变量的名称。
   b. 指定创建数组数据类型时使用的数组数据类型的名称。

   如果使用以下 CREATE TYPE 语句声明数组数据类型:

   ```
   CREATE TYPE simpleArray AS INTEGER ARRAY[10];
   ```

   那么应按如下所示声明此数据类型的变量:

   ```
   DECLARE myArray simpleArray;
   ```

   如果使用以下 CREATE TYPE 语句声明数组数据类型:

   ```
   CREATE TYPE id_Phone AS VARCHAR(20) ARRAY[100];
   ```

   那么应按如下所示创建此数据类型的变量:

   ```
   DECLARE id_Phone_Toronto_List  id_Phone;
   ```

   此数组最多可包含 100 个存储为 VARCHAR(20) 数据类型值的电话值，下标为 1 到 100 的整数值。变量名称指示电话值为多伦多电话号码。
2. 将 DECLARE 语句包括在受支持上下文中。此操作可在 CREATE PROCEDURE、CREATE FUNCTION 或 CREATE TRIGGER 语句中进行。
3. 执行包含 DECLARE 语句的语句。

结果

该语句应成功执行。

如果该语句因为 DECLARE 语句存在错误而未能成功执行，请执行以下操作:

- 验证 DECLARE 语句的 SQL 语句语法并再次执行该语句。
- 验证是否尚未在同一上下文中声明同名的其他变量。
- 验证是否成功创建了数组数据类型。

下一步做什么

声明关联数组变量后，您可能想要对它们指定值。

将值指定给数组

可以多种方式将值指定给数组。

使用子下标和字面值来指定数组值:

可使用子下标和字面值将值指定给关联数组。

开始之前

- 读取: 第 27 页的『常规数组数据类型』
- 读取: 第 27 页的『对数组数据类型的限制』
- 执行 SET 语句所需的特权

关于此任务

您应先执行此任务，然后根据具有已指定值的变量有条件执行 SQL PL 或将该变量作为参数传递至例程。

过程

1. 定义 SET 语句。
   a. 指定数组变量名称。
   b. 指定赋值符号“=”。
   c. 指定 ARRAY 关键字和值对（值对必须用方括号括起来）。
2. 执行 SET 语句。

示例

以下是有关如何将元素值指定给名为 myArray 的数组的示例:

```
SET myArray[1]   =  123;
SET myArray[2]   =  124;
...
SET myArray[100] =  223;
```

下一步做什么

如果 SET 语句成功执行，那么表示成功定义了数组元素。要验证是否创建了数组，可尝试从该数组检索值。

如果 SET 语句未能成功执行，请执行以下操作:

- 验证 SET 语句的 SQL 语句语法并再次执行该语句。
- 验证是否成功创建了数据类型。

检索数组值

可以多种方式检索数组值。

使用下标检索数组值:

通过引用数组并指定子下标值来检索数组元素值。

开始之前

以下是完成此任务的先决条件:

- 读取: 第 27 页的『常规数组数据类型』
- 读取: 第 27 页的『对数组数据类型的限制』
- 执行 SET 语句或任何包含数组引用的 SQL 语句所需的特权

关于此任务

您应在 SQL PL 代码中执行此任务以访问数组中存储的值。您可访问赋值（SET）语句中的数组元素值或直接在表达式中访问。

过程

1. 定义 SET 语句。
   a. 指定数据类型与数组元素相同的变量名称。
   b. 指定赋值符号“=”。
   c. 指定数组的名称和下标值（下标值用方括号括起来）。
2. 执行 SET 语句。

示例

以下是检索数组值的 SET 语句的示例:

```
SET mylocalVar = myArray[1];
```

下一步做什么

如果 SET 语句成功执行，那么局部变量应包含数组元素值。

如果 SET 语句未能成功执行，请执行以下操作:

- 验证 SET 语句的 SQL 语句语法并再次执行该语句。
- 验证变量的数据类型是否与数组元素相同。
- 验证该数组是否已创建成功并且当前存在。

检索数组元素数目:

可通过使用 CARDINALITY 函数很轻松地检索简单数组中的数组元素数目，并且可使用 MAX_CARDINALITY 函数检索数组的最大允许大小。

开始之前

- 读取: 第 27 页的『常规数组数据类型』
- 读取: 第 27 页的『对数组数据类型的限制』
- 执行 SET 语句所需的特权

关于此任务

您应在 SQL PL 代码中执行此任务以访问数组中的元素数目的计数值。您可访问赋值（SET）语句中的数组元素值或直接访问表达式中的值。

过程

1. 定义 SET 语句。
   a. 声明并指定类型为整数的变量名称，该变量用于保存基数值。
   b. 指定赋值符号“=”。
   c. 指定 CARDINALITY 或 MAX_CARDINALTIY 函数的名称及数组名称（数组名称必须用括号括起来）。
2. 执行 SET 语句。

结果

如果 SET 语句成功执行，那么局部变量应包含数组中元素数目的计数值。

示例

以下是演示这些赋值的两个 SET 语句的示例:

```
SET card = CARDINALITY(arrayName);

SET maxcard = MAX_CARDINALITY(arrayName);
```

下一步做什么

如果 SET 语句未能成功执行，请执行以下操作:

- 验证 SET 语句的 SQL 语句语法并再次执行该语句。
- 验证局部变量是否为整数数据类型。
- 验证该数组是否已创建成功并且当前存在。

检索第一个和最后一个数组元素（FIRST 和 LAST 函数）:

可通过使用 FIRST 和 LAST 函数很轻松地检索简单数组中的第一个和最后一个元素。

开始之前

- 读取: 第 27 页的『常规数组数据类型』
- 读取: 第 27 页的『对数组数据类型的限制』
- 执行 SET 语句所需的特权

关于此任务

您应在 SQL PL 代码中执行此任务以快速访问数组中的第一个元素。

过程

定义 SET 语句:

1. 声明并指定类型与数组元素相同的变量。
2. 指定赋值符号“=”。
3. 指定 FIRST 或 LAST 函数的名称及数组名称（数组名称必须用括号括起来）。

结果

如果 SET 语句成功执行，那么局部变量应包含数组中第一个或最后一个（适当时）下标值。

示例

对于定义为如下所示的一组电话号码:

```
firstPhone index    0              1              2              3
           phone  '416-223-2233' '416-933-9333' '416-887-8887' '416-722-7227'
```

如果执行以下 SQL 语句:

```
SET firstPhoneIx = FIRST(phones);
```

变量 firstPhoneIx 的值将为 0。即使此位置中的元素值为 NULL 时也是这样。

以下 SET 语句访问数组第一个位置中的元素值:

```
SET firstPhone = A[FIRST(A)]
```

下一步做什么

如果 SET 语句未能成功执行，请执行以下操作:

- 验证 SET 语句的 SQL 语句语法并再次执行该语句。
- 验证局部变量是否为正确数据类型。
- 验证该数组是否已创建成功并且当前存在。

检索下一个和上一个数组元素:

可通过使用 PREV 和 NEXT 函数很轻松地检索简单数组中的下一个或上一个元素。

开始之前

- 读取: 第 27 页的『常规数组数据类型』
- 读取: 第 27 页的『对数组数据类型的限制』
- 执行 SET 语句所需的特权

关于此任务

您应在 SQL PL 代码中执行此任务以快速访问数组中的直接相邻元素值。

过程

1. 定义 SET 语句:
   a. 声明并指定类型与数组元素相同的变量。
   b. 指定赋值符号“=”。
   c. 指定 NEXT 或 PREV 函数的名称及数组名称（数组名称必须用括号括起来）。
2. 执行 SET 语句。

示例

对于定义为如下所示的一组电话号码:

```
  firstPhone index  0             1             2             3
               phone  '416-223-2233' '416-933-9333' '416-887-8887' '416-722-7227'
```

以下 SQL 语句将变量 firstPhone 设置为值 0。

```
SET firstPhone = FIRST(phones);
```

以下 SQL 语句将变量 nextPhone 设置为值 1。

```
SET nextPhone = NEXT(phones, firstPhone);
```

以下 SQL 语句将变量 phoneNumber 设置为数组中 nextPhone 之后下一个位置的电话号码值。这是下标值位置 2 处的数组元素值。

```
SET phoneNumber = phones[NEXT(phones, nextPhone)];
```

下一步做什么

如果 SET 语句未能成功执行，请执行以下操作:

- 验证 SET 语句的 SQL 语句语法并再次执行该语句。

- 验证局部变量是否为正确数据类型。
- 验证该数组是否已创建成功并且当前存在。

修剪数组（TRIM_ARRAY 函数）

修剪数组是想要从数组末尾除去不想要的数组元素时使用 TRIM_ARRAY 函数执行的任务。

开始之前

- 读取: 数组数据类型
- 读取: 对数组数据类型的限制
- 执行 SET 语句所需的特权

关于此任务

您应在 SQL PL 代码中执行此任务以迅速从数组末尾除去数组元素。

过程

1. 定义 SET 语句:
   a. 声明并指定类型与要修改的数组相同的数组变量，或重复使用同一数组变量。
   b. 指定赋值符号“=”。
   c. 指定 TRIM_ARRAY 函数的名称及数组名称和要修剪的元素数目（数组名称和元素数目必须用括号括起来）。
2. 执行 SET 语句。

结果

如果 SET 语句成功执行，那么数组 phones 应包含已更新值。

示例

对于定义为如下所示的一组电话号码:

```
phones  index  0       1       2       3
          phone  '416-223-2233' '416-933-9333' '416-887-8887' '416-722-7227'
```

执行以下语句后:

```
SET phones = TRIM_ARRAY ( phones,  2 );
```

数组 phones 将定义为如下所示:

```
phones  index  0       1
         phone  '416-223-2233' '416-933-9333'
```

下一步做什么

如果 SET 语句未能成功执行，请执行以下操作:

- 验证 SET 语句的 SQL 语句语法并再次执行该语句。
- 验证局部变量是否为正确数据类型。
- 验证该数组是否已创建成功并且当前存在。

删除数组元素（ARRAY_DELETE）

可使用 ARRAY_DELETE 函数来从数组中永久删除元素。

开始之前

- 读取: 数组数据类型
- 读取: 对数组数据类型的限制
- 执行 SET 语句所需的特权

关于此任务

您应在 SQL PL 代码中执行此任务以在数组中删除元素。

过程

1. 定义 SET 语句:
   a. 声明并指定类型与数组元素相同的变量。
   b. 指定赋值符号“=”。
   c. 指定 ARRAY_DELETE 函数的名称及数组名称和用于定义元素删除范围的子下标（数组名称和子下标必须用括号括起来）。
2. 执行 SET 语句。

结果

如果 SET 语句成功执行，那么数组 phones 应包含已更新值。

示例

对于定义为如下所示的一组电话号码:

```
phones  index  0             1             2             3
         phone  '416-223-2233' '416-933-9333' '416-887-8887' '416-722-7227'
```

执行以下 SQL 语句后:

```
SET phones = ARRAY_DELETE ( phones,  1, 2 );
```

数组 phones 将定义为如下所示:

```
phones  index  0             3
         phone  '416-223-2233' '416-722-7227'
```

下一步做什么

如果 SET 语句未能成功执行，请执行以下操作:

- 验证 SET 语句的 SQL 语句语法并再次执行该语句。
- 验证局部变量是否为正确数据类型。
- 验证该数组是否已创建成功并且当前存在。

确定数组元素是否存在

确定数组元素是否存在并且具有值是可使用 ARRAY_EXISTS 函数完成的任务。

开始之前

- 读取: 第 27 页的『常规数组数据类型』
- 读取: 第 27 页的『对数组数据类型的限制』
- 执行 IF 语句或引用了 ARRAY_EXISTS 函数的任何 SQL 语句所需的特权。

关于此任务

您应在 SQL PL 代码中执行此任务以确定数组中是否存在数组元素。

过程

1. 定义 IF 语句:
   a. 定义包括 ARRAY_EXISTS 函数的条件。
   b. 指定 THEN 子句并包括希望在符合条件时执行的任何逻辑，并添加想的任何 ELSE 子句值。
   c. 使用 END IF 子句关闭 IF 语句。
2. 执行 IF 语句。

示例

对于定义为如下所示的一组电话号码:

```
phones  index  0             1             2             3
         phone  '416-223-2233' '416-933-9333' '416-887-8887' '416-722-7227'
```

执行以下语句后，变量 x 将设置为 1。

```
IF (ARRAY_EXISTS(phones, 2)) THEN
  SET x = 1;
END IF;
```

下一步做什么

如果 SET 语句未能成功执行，请执行以下操作:

- 验证 SET 语句的 SQL 语句语法并再次执行该语句。
- 验证局部变量是否为正确数据类型。
- 验证该数组是否已创建成功并且当前存在。

SQL 过程中的数组支持

SQL 过程支持数组类型的参数和变量。在应用程序与存储过程之间或两个存储过程之间传递瞬态数据集合时，使用数组会很方便。

在 SQL 存储过程中，可将数组作为传统编程语言中的数组操作。而且，数组将在关系模型中按以下方式集成: 表示为数组的数据可轻松转换为表，并且表列中的数据可聚集成数组。下面的示例说明针对数组的一些操作。两个示例都是命令行处理器 (CLP) 脚本，它们使用百分比字符 (%) 作为语句终止符。

示例 1

此示例显示以下两个过程: sub 和 main。过程 main 使用数组构造函数创建包含 6 个整数的数组。然后将该数组传递至过程 sum，该过程会计算输入数组中所有元素之和，并将结果返回给 main。过程 sum 说明数组子下标及 CARDINALITY 函数的用法，该函数返回数组中的元素数目。

```
create type intArray as integer array[100] %

create procedure sum(in numList intArray, out total integer)
begin
declare i, n integer;

set n = CARDINALITY(numList);

set i = 1;
set total = 0;

while (i <= n) do
set total = total + numList[i];
set i = i + 1;
end while;

end %

create procedure main(out total integer)
begin
declare numList intArray;

set numList = ARRAY[1,2,3,4,5,6];

call sum(numList, total);

end %
```

示例 2

在此示例中，我们使用两种数组数据类型（intArray 和 stringArray）以及带有两个列（id 和 name）的 persons 表。过程 processPersons 向表添加三个额外人员，并返回人员姓名中包含字母“o”的数组（按标识排序）。要添加的三个人员的标识和名称表示为两个数组（ids 和 names）。这些数组用作 UNNEST 函数的自变量，此函数将数组变为包含两个列的表，其元素将插入到 persons 表中。最后，过程中的最后一个 SET 语句使用 ARRAY_AGG 聚集函数来计算输出参数的值。

```
create type intArray as integer array[100] %
create type stringArray as varchar(10) array[100] %

create table persons (id integer, name varchar(10)) %
insert into persons values(2, 'Tom') %
insert into persons values(4, 'Jill') %
insert into persons values(1, 'Joe') %
insert into persons values(3, 'Mary') %

create procedure processPersons(out witho stringArray)
begin
declare ids intArray;
declare names stringArray;

set ids = ARRAY[5,6,7];
set names = ARRAY['Bob', 'Ann', 'Sue'];

insert into persons(id, name)
(select T.i, T.n from UNNEST(ids, names) as T(i, n));
```

```
set witho = (select array_agg(name order by id)
from persons
where name like '%o%');
end %
```

关联数组数据类型

关联数组数据类型是用于表示没有预定义基数的常规数组的数据类型。关联数组包含同一数据类型的零个或零个以上元素的有序集合，其中每个元素按下标值排序并且通过下标值引用。

关联数组的下标值是唯一的，发球同一数据类型并且不必连续。

以下主题提供有关关联数组数据类型的更多信息:

关联数组的功能

关联数组数据类型用于表示关联数组。它有许多功能，因此非常实用。

关联数组数据类型支持以下关联数组属性:

- 未对关联数组指定预定义基数。这允许您继续向数组添加元素而不考虑最大大小，这在您不知道将来构成集合的元素数目时很有用。
- 数组下标值可以是非整数数据类型。VARCHAR 和 INTEGER 都是受支持的关联数组下标值。
- 下标值不必连续。与传统数组（按位置确定下标）相比，关联数组按另一数据类型的值确定下标，并且对于最低与最高下标值之间的所有可能下标值，不一定有对应的下标元素。如果要创建用于存储姓名和电话号码的集合，那么这一点很有用。可按任意顺序将数据对添加至集合并使用数据对中定义为下标的数据项进行排序。
- 关联数组中的元素按下标值的升序排序。元素的插入顺序无关紧要。
- 可使用直接引用或一组可用标量函数来访问和设置关联数组数据。
- 关联数组在 SQL PL 上下文中受支持。
- 关联数组可用来管理和传递同一种类的值集（以集合形式），而不必:
  - 将数据减少至标量值并通过一次一个元素方式进行处理，这可能导致网络流量问题。
  - 使用作为参数传递的游标。
  - 将数据减少为标量值并使用 VALUES 子句将它们重新构造为一个集合。

对关联数组数据类型的限制

使用数组数据类型之前或诊断因为声明数组数据类型而导致的问题时，应注意数组数据类型的限制。

数组数据类型存在以下限制:

- 只能在 SQL PL 上下文中声明、创建或引用关联数组。以下是可在其中使用此数据类型的 SQL PL 上下文列表:
  - 在模块中定义的 SQL 函数的参数。
  - 未在模块中定义但具有复合 SQL（编译型）语句作为函数体的 SQL 函数的参数。
  - 在模块中定义的 SQL 函数的返回类型。

- 未在模块中定义但具有复合 SQL（编译型）语句作为函数体的 SQL 函数的返回类型。
- SQL 过程的参数。
- 在模块中定义的 SQL 函数中声明的局部变量。
- 未在模块中定义但具有复合 SQL（编译型）语句作为函数体的 SQL 函数中声明的局部变量。
- SQL 过程中声明的局部变量。
- 复合 SQL（编译型）语句作为触发器体的触发器中声明的局部变量。
- 复合 SQL（编译型）语句的 SQL 语句中的表达式。
- SQL PL 上下文的 SQL 语句中的表达式。
- 全局变量。

在上述任一列出的 SQL PL 上下文之外使用无效。

- 关联数组不能是表列类型。
- 不允许 NULL 作为下标值。
- 关联数组的最大大小受系统资源限制。
- 关联数组不能成为 TRIM_ARRAY 函数的输入。关联数组值不能存储在表列中。
- 支持 MAX_CARDINALITY 函数与关联数组配合使用，但该函数始终返回空值，原因是关联数组没有指定的最大大小。

创建关联数组数据类型

创建关联数组数据类型的变量之前，应创建关联数组数据类型。关联数组数据类型通过执行 CREATE TYPE (array) 语句创建。

开始之前

确保您具有执行 CREATE TYPE 语句所需的特权。

关于此任务

关联数组数据类型只能在特定上下文中使用。

过程

1. 定义 CREATE TYPE 语句:
   a. 指定关联数组数据类型的名称。 该名称应指定清晰指定数组中存储的数据类型。例如，Products 很适合作为包含产品相关信息的数组的名称，其中数组下标是产品标识。又例如，y_coordinate 很适合作为数组下标是图形函数中 x 坐标值的数组的名称。
   a. 指定 AS 关键字，后跟数组元素的数据类型的关键字名称（如 INTEGER）。
   b. 指定 ARRAY 关键字。 在 ARRAY 子句的方括号中指定数组下标的数据类型。注意: 对于关联数组，元素数目或数组下标值的范围没有明确限制。
2. 从受支持接口执行 CREATE TYPE 语句。

示例

示例 1:
以下是 CREATE TYPE 语句的示例，该语句创建带有 20 个元素的数组 assocArray，数组下标的类型为 VARCHAR。

```
CREATE TYPE assocArray AS INTEGER ARRAY[VARCHAR(20)];
```

示例 2:
以下是基本关联数组定义的示例，该定义使用省份名称来作为下标，其中的元素是省会城市:

```
CREATE TYPE capitalsArray AS VARCHAR(12) ARRAY[VARCHAR(16)];
```

下一步做什么

如果该语句成功执行，那么会在数据库中创建数组数据类型并且可引用该数组数据类型。

创建数组数据类型后，您可能想要创建关联数据类型变量。

声明关联数组变量

创建关联数组数据类型后，为了能够临时存储或传递关联数组数据类型值，应该声明关联数组变量。局部变量通过使用 DECLARE 语句声明。全局变量通过使用 CREATE VARIABLE 语句创建。

开始之前

- 读取: 关联数组数据类型
- 读取: 对关联数组数据类型的限制
- 读取: 创建关联数组数据类型
- 对于全局变量，需要特权才能执行 CREATE VARIABLE 语句。对于局部变量，不需要任何特权就可执行 DECLARE 语句

关于此任务

可在受支持上下文中声明并使用关联数组变量以存储行数据集。

过程

1. 对局部变量定义 DECLARE 语句或对全局变量定义 CREATE TYPE 语句:
   a. 指定关联数组数据类型的名称。
   b. 指定创建关联数组数据类型时使用的关联数组数据类型的名称。
2. 从受支持接口执行 CREATE TYPE 语句。

示例

示例 1:
考虑定义为如下所示的关联数组数据类型:

```
CREATE TYPE Representative_Location AS VARCHAR(20) ARRAY[VARCHAR(30)];
```

要声明此数据类型的变量，应按如下所示使用 DECLARE 语句:

```
DECLARE RepsByCity Representative_Location;
```

此数组最多可包含关联数组元素值的最大数目，这些值存储为 VARCHAR(20) 数据类型值，它们按唯一变量字符数据类型值确定下标。变量名称指示一组销售代表名称按他们所代表的城市名称确定下标。在此数组中，一个销售代表只能代表一个城市（城市为数组下标值）。

示例 2:
考虑关联数组数据类型，此数据类型定义为存储为元素值，即首都名称（下标为省份名称）：

```
CREATE TYPE capitalsArray AS VARCHAR(12) ARRAY[VARCHAR(16)];
```

要创建此数据类型的变量，应按如下所示使用 CREATE VARIABLE 语句:

```
CREATE VARIABLE capitals capitalsArray;
```

此数组最多可包含关联数组元素值的最大数目，这些值存储为 VARCHAR(20) 数据类型值，它们按唯一变量字符数据类型值确定下标。变量名称指示一组销售代表名称按他们所代表的城市名称确定下标。在此数组中，一个销售代表只能代表一个城市（城市为数组下标值）。

下一步做什么

如果 DECLARE 语句或 CREATE VARIABLE 语句成功执行，那么表示已成功定义并且可引用数组数据类型。要验证是否已创建关联数组变量，可将值指定给该数组并尝试引用数组中的值。

如果 DECLARE 语句或 CREATE VARIABLE 语句未能成功执行，请验证 DECLARE 语句的 SQL 语言语法并再次执行该语句。请参阅 DECLARE 语句。

使用子下标和字面值来指定数组值

创建或声明关联数组变量后，可对其指定值。指定关联数组值的一种方法是直接指定。

开始之前

- 读取: 关联数组数据类型
- 读取: 对关联数组数据类型的限制
- 确保关联数组变量在当前使用范围内。

关于此任务

可使用赋值语句来对关联数组变量元素指定值，赋值语句中命名了数组，指定了下标值并且指定了对应元素值。

过程

1. 对关联数组变量定义赋值语句。
   - 指定变量名称、下标值和元素值。
   - 指定另一关联数组变量。
2. 从受支持接口执行赋值语句。

示例

示例 1:
以下是变量声明以及一系列用于在数组中定义值的赋值语句的示例:

```
DECLARE capitals capitalsArray;

SET capitals['British Columbia'] = 'Victoria';
SET capitals['Alberta'] = 'Edmonton';
SET capitals['Manitoba'] = 'Winnipeg';
SET capitals['Ontario'] = 'Toronto';
SET capitals['Nova Scotia'] = 'Halifax';
```

在 capitals 数组中，数组下标值是省份名称，关联数组元素值是对应省会城市的名称。关联数组按下标值的升序排序。对关联数组元素指定值时使用的顺序无关紧要。

示例 2:
还可对关联数组变量指定同一关联数组数据类型的关联数组变量值。可使用赋值语句来完成此操作。例如，考虑 capitalsA 和 capitalsB 这两个关联数组变量定义为如下所示:

```
DECLARE capitalsA capitalsArray;
DECLARE capitalsB capitalsArray;

SET capitalsA['British Columbia'] = 'Victoria';
SET capitalsA['Alberta'] = 'Edmonton';
SET capitalsA['Manitoba'] = 'Winnipeg';
SET capitalsA['Ontario'] = 'Toronto';
SET capitalsA['Nova Scotia'] = 'Halifax';
```

可通过执行以下赋值语句对变量 capitalsB 指定变量 capitalsA 的值:

```
SET capitalsB = capitalsA;
```

执行后 capitalsB 的值与 capitalsA 的值相同。

下一步做什么

如果赋值语句成功执行，那么会成功指定该值并且可引用新的变量值。

如果该语句执行失败，请验证并更正 SQL 语句的语法，同时验证是否定义了指定变量，然后再次执行该语句。

游标类型

游标类型可以是内置数据类型 CURSOR 或基于内置 CURSOR 数据类型的用户定义的类型。还可使用特定行类型来定义用户定义游标类型，以限制关联游标的结果行的属性。

游标类型与行数据结构（由行指定）相关联时，该游标类型称为强类型游标。只能将与该定义相匹配的结果集指定给强类型游标数据类型的变量并存储在该变量中。游标数据类型定义没有关联结果集定义时，该游标数据类型称为弱类型。可将任何结果集存储在弱类型化游标数据类型的变量中。

该游标数据类型仅支持与 SQL PL 配合使用。它主要用于创建可供游标变量声明使用的游标类型定义。

此数据类型可用于:

- 定义游标变量声明。
- 简化 SQL 过程语言应用程序中的逻辑编码。例如，数据库应用程序处理称为结果集的记录集，在某些情况下，可能需要在不同上下文中引用和处理相同结果集。在接口之间传递已定义结果集可能需要复杂逻辑。游标数据类型允许创建游标变量，这些变量可用于存储结果集、处理结果集以及将结果集作为参数传递。
- 使移植至具有相似数据类型的 DB2 SQL PL 代码更加容易。

必须使用 CREATE TYPE 语句创建游标数据类型。此操作完成后，可声明和引用此数据类型的变量。可对游标变量指定行数据结构定义、可打开游标变量、关闭游标变量、对游标变量指定来自另一游标变量的一组行、或将游标变量作为参数自 SQL 过程传递。

游标数据类型概述

此游标数据类型概述介绍游标数据类型的种类、可使用的范围，并提供与游标数据类型使用有关的限制和特权的相关信息。

游标数据类型的类型

有两种主要类型的游标数据类型：弱类型化游标数据类型和强类型游标数据类型。强类型或弱类型的属性是在创建数据类型时定义的。此属性保留在已创建的每种类型的变量中。

下面提供了强类型游标数据类型和弱类型化游标数据类型的特征:

强类型游标数据类型

强类型游标数据类型是使用行数据结构指定的结果集定义创建的数据类型。这些数据类型称为强类型，原因是将结果集值指定给这些数据类型时，可检查结果集的数据类型。可通过提供行类型定义来定义游标数据类型结果集定义。只有与定义相匹配的结果集才能指定给强类型游标数据类型并存储在其中。强类型检查是在指定时执行的，如果存在任何数据类型不匹配，那么会发生错误。

强类型游标数据类型的结果集定义可通过行数据类型定义或 SQL 语句定义提供。

以下是游标数据类型定义的示例，该示例定义为返回行格式与 rowType 数据类型相同的结果集:

```
CREATE TYPE cursorType AS rowType CURSOR@
```

只有包含的数据列的数据类型定义与 rowType 行定义相同的结果集可成功指定给声明为 cursorType 游标数据类型的变量。

以下是游标数据类型定义的示例，该示例定义为返回行格式与定义表 T1 的格式相同的结果集:

```
CREATE TABLE T1 (C1 INT)

 CREATE TYPE cursorType AS ANCHOR ROW OF t1 CURSOR;
```

只有包含的数据列的数据类型定义与表 t1 的列定义相同的结果集可成功指定给声明为 cursorType 游标数据类型的变量。

与强类型游标相关联的行定义可作为锚定位数据类型的定义引用。以下示例说明这一点:

```
CREATE TYPE r1 AS ROW (C1 INT);
CREATE TYPE c1 AS RTEST CURSOR;

DECLARE c1 CTEST;
DECLARE r1 ANCHOR ROW OF CV1;
```

定义了行数据类型 r1，并且定义了与 r1 的行定义相关联的游标类型 c1。后续 SQL 语句可能出现在 SQL 过程中的变量声明的示例。第二个变量声明用于变量 r1，它被定义为锚定位数据类型：它被定位至用于定义游标 cv1 的行类型。

弱类型化游标数据类型

弱类型化游标数据类型与任何行数据类型定义无关。将值指定给弱类型游标变量时未执行任何类型检查。

有一个内置弱类型化游标数据类型 CURSOR，可用于声明弱类型游标变量或参数。以下是基于内置弱类型化游标数据类型 CURSOR 的弱类型游标变量声明的示例：

```
DECLARE cv1 CURSOR;
```

弱类型游标变量在您必须存储行定义未知的结果集时很有用。
要将弱类型游标变量作为输出参数返回，必须打开该游标。
在此版本中基于弱类型化游标数据类型的变量不能作为锚定位数据类型引用。
您可以定义用户定义弱类型化游标数据类型。

游标变量的所有其他特征都是每种类型的游标变量的公共特征。

对游标数据类型的限制

对游标数据类型和游标变量的限制限制了游标变量功能及可定义和引用游标变量的位置。

实现游标数据类型和变量之前，应注意它们的限制。确定游标变量是否符合您的要求时，这些限制可能会很重要，并且在诊断与游标数据类型和变量使用有关的错误期间进行检查时很有用。

在此版本中，游标数据类型存在以下限制：

- 只能在 SQL 过程中将游标数据类型创建为局部类型。

在此版本中，游标变量存在以下限制：

- 不支持在应用程序中使用游标变量。只能在 SQL PL 上下文中引用和声明游标变量。
- 游标变量是只读游标。
- 通过使用游标变量访问的行不可更新。
- 游标变量不是可滚动游标。
- 强类型游标变量列不能作为锚定位数据类型引用。
- 不支持全局游标变量。
- 不能在游标变量定义中引用 XML 列。
- 不能使用 XQuery 语言语句来定义强类型游标结果集。

与游标数据类型使用相关的特权

与游标数据类型和变量相关的特定特权的存在是为了限制和控制哪些用户可创建游标数据类型和变量。

要创建游标数据类型，需要以下特权:

- 执行 CREATE TYPE 语句以创建游标数据类型的特权。

要根据现有游标数据类型声明游标变量，不需要任何特权。

要初始化游标变量、打开游标变量引用的游标或从打开的游标变量引用访存值，您需要执行 DECLARE CURSOR 语句所需的特权。

游标变量

游标变量是基于预定义游标数据类型的游标。可对游标变量取消初始化、初始化、指定值、将其设置为另一个值或作为参数自 SQL 过程传递。游标变量继承它们所基于的游标数据类型的属性。游标变量可以是强类型或弱类型。游标变量保存对游标数据类型所定义游标的上下文的引用。

可使用 DECLARE 语句在 SQL 过程中声明游标变量。

游标谓词

游标谓词是一些 SQL 关键字，可用于确定当前作用域内定义的游标的状态。它们提供了一种方法以轻松了解游标是否打开或是否存在与游标相关联的行。

可在 SQL 和 SQL PL 语句中游标状态可用作谓词条件的任何位置引用游标谓词。可使用的游标谓词包括:

IS OPEN

此谓词可用于确定游标是否处于打开状态。游标解析为函数和过程的参数的情况下，此谓词非常有用。尝试打开游标之前，可使用此谓词来确定游标是否已打开。

IS NOT OPEN

此谓词可用于确定游标是否已关闭。其值是 IS OPEN 的逻辑反义词。在尝试实际关闭游标之前确定游标是否关闭时，此谓词会非常有用。

IS FOUND

此谓词可用于确定执行 FETCH 语句后游标是否包含行。如果最后一个 FETCH 语句成功执行，那么 IS FOUND 谓词值为 true。如果最后一个 FETCH 语句执行后导致找不到行，那么结果为 false。在下列情况下，结果未知:

- 游标变量名称的值为空
- 游标变量名称的底层游标未打开
- 在执行第一个 FETCH 操作之前，针对底层游标对谓词进行了求值
- 最后一个 FETCH 操作返回了错误

IS FOUND 谓词在循环并在每次迭代时执行访存的 SQL PL 逻辑的一部分中会很有用。该谓词可用于确定是否还有要访存的行。它可以有效替代条件处理程序的使用，该条件处理程序用于检查没有其他要访存的行时发生的错误情况。

使用 IS FOUND 的替代方法是使用 IS NOT FOUND（这是 IS FOUND 的相反值）。

示例

以下脚本定义 SQL 过程，该过程引用这些谓词以及成功编译和调用过程所需的必备对象:

```
CREATE TABLE T1 (c1 INT, c2 INT, c3 INT)@

insert into t1 values (1,1,1),(2,2,2),(3,3,3) @

CREATE TYPE myRowType AS ROW(c1 INT, c2 INT, c3 INT)@

CREATE TYPE myCursorType AS myRowType CURSOR@


CREATE PROCEDURE p(OUT count INT)
LANGUAGE SQL
BEGIN
  DECLARE C1 cursor;
  DECLARE lvarInt INT;

  SET count = -1;
  SET c1 = CURSOR FOR SELECT c1 FROM t1;

  IF (c1 IS NOT OPEN) THEN
     OPEN c1;
  ELSE
     set count = -2;
  END IF;

  set count = 0;
  IF (c1 IS OPEN) THEN

    FETCH c1 into lvarInt;

    WHILE (c1 IS FOUND) DO
      SET count = count + 1;
      FETCH c1 INTO lvarInt;
    END WHILE;
  ELSE
     SET count = 0;
  END IF;

END@

CALL p()@
```

创建游标变量

要创建游标变量，必须先创建游标类型，然后根据该类型创建游标变量。

使用 CREATE TYPE 语句创建游标数据类型

创建游标数据类型是创建游标变量的先决条件。游标数据类型通过使用 CREATE TYPE (cursor) 语句创建。

开始之前

要执行此任务，您需要:

- 执行 CREATE TYPE (cursor) 语句的特权。
- 如果创建强类型游标数据类型，那么必须预编译行规范或以表、视图或游标中的现有行作为基础。

CREATE TYPE (cursor) 语句定义游标数据类型，可在 SQL PL 中用于声明该游标数据类型的参数和局部变量。如果在 CREATE TYPE (cursor) 语句中指定行类型名称子句，那么会创建强类型游标数据类型。省略行类型名称子句时会创建弱定义游标数据类型。

作为创建弱定义游标数据类型的替代项，可在声明游标变量时使用内置弱定义游标数据类型 CURSOR。

```
CREATE TYPE weakCursorType AS CURSOR@
```

如果要创建强类型游标数据类型，那么行数据类型定义必须存在，它将定义可与游标相关联的结果集。可从显式定义的行数据类型、表或视图或者强类型游标来派生行数据类型定义。以下是行类型定义的示例:

```
CREATE TYPE empRow AS ROW (name varchar(128), ID varchar(8))@
```

以下是可从中派生行类型定义的表定义的示例:

```
CREATE TABLE empTable AS ROW (name varchar(128), ID varchar(8))@
```

关于此任务

要在数据库中定义强类型游标数据类型，必须从任何支持执行 SQL 语句的 DB2 接口成功执行 CREATE TYPE (CURSOR) 语句。

过程

1. 创建 CREATE TYPE (CURSOR) 语句:
   a. 指定类型的名称。
   b. 通过执行下列其中一个操作来指定行定义：引用行数据类型的名称、指定该类型应定位至表或视图，或定位至与现有强游标类型相关联的结果集定义。
2. 从受支持的 DB2 接口执行 CREATE TYPE 语句。

结果

如果 CREATE TYPE 语句成功执行，那么会在数据库中创建游标数据类型。

示例

以下是有关如何创建可与以下结果集相关联的弱类型化游标数据类型的示例，该结果集的格式与 empRow 行数据类型相同:

```
CREATE TYPE cursorType AS empRow CURSOR@
```

以下是有关如何创建可与以下结果集相关联的游标数据类型的示例，该结果集的格式与表 empTable 相同:

```
CREATE TYPE cursorType AS ANCHOR ROW OF empTable@
```

下一步做什么

创建游标数据类型之后，可根据此数据类型声明游标变量。

声明游标类型的局部变量

创建游标数据类型后，可声明游标类型的局部变量。

开始之前

数据库中必须存在游标数据类型定义。游标数据类型通过成功执行 CREATE TYPE (CURSOR) 语句创建。以下是强类型游标类型定义的示例:

```
CREATE TYPE cursorType AS empRow CURSOR;
```

关于此任务

在此版本中，只能将游标变量声明为 SQL 过程中的局部变量。可声明强类型和弱类型游标变量。

过程

1. 创建 DECLARE 语句:
   a. 指定变量的名称。
   b. 指定将定义该变量的游标数据类型。如果游标变量将为弱类型，那么必须指定用户定义的弱类型化游标数据类型或内置弱类型化游标数据类型 CURSOR。如果游标变量将以弱类型化游标数据类型为基础，那么可立即初始化该变量。

   以下是有关如何创建 DECLARE 语句的示例，该语句将定义未初始化并且类型为 cursorType 的游标变量:

   ```
   DECLARE Cv1 cursorType@
   ```

   以下是有关如何创建 DECLARE 语句的示例，该语句将定义游标变量 Cv2，该变量的类型将定位至现有游标变量 Cv1 的类型:

   ```
   DECLARE Cv2 ANCHOR DATA TYPE TO Cv1@
   ```

   以下是有关如何创建 DECLARE 语句的示例，该语句将定义弱类型游标变量:

   ```
   DECLARE Cv1 CURSOR@
   ```
2. 在受支持上下文中执行 DECLARE 语句。

结果

如果 DECLARE 语句成功执行，那么会创建游标变量。

下一步做什么

创建此游标变量后，可对游标变量指定值、对其进行引用或作为参数传递。

将值指定给游标变量

可在不同时间以多种方式通过使用 SET 语句将结果集指定给游标变量。

关于此任务

将查询结果集指定给游标变量

可使用 SET 语句和 CURSOR FOR 关键字将 SELECT 查询的结果集指定给游标变量。以下是有关如何将与针对表 T 的查询相关联的结果集指定给游标变量 c1 的示例，该游标变量的行定义与该表完全相同。

如果 T 定义为如下所示:

```
CREATE TABLE T  (C1 INT, C2 INT, C3 INT);
```

如果 C1 是定义为如下所示的强类型游标变量:

```
CREATE TYPE simpleRow AS ROW (c1 INT, c2 INT, c3 INT);
CREATE TYPE simpleCur AS CURSOR RETURNS simpleRow;
DECLARE c1 simpleCur;
```

那么可按如下所示完成赋值:

```
SET c1 = CURSOR FOR SELECT * FROM T;
```

强类型检查将成功，原因是 c1 的定义与表 T 相兼容。如果 c1 是弱类型游标，那么此赋值也将成功，原因是不会执行任何数据类型检查。

将字面值指定给游标变量

可使用 SET 语句和 CURSOR FOR 关键字将 SELECT 查询的结果集指定给游标变量。以下是有关如何将与针对表 T 的查询相关联的结果集指定给游标变量 c1 的示例，该游标变量的行定义与该表完全相同。

使 T 成为定义为如下所示的表:

```
CREATE TABLE T  (C1 INT, C2 INT, C3 INT);
```

使 simpleRow 成为行类型，并使 simpleCur 成为游标类型，它们分别按如下所示创建:

```
CREATE TYPE simpleRow AS ROW (c1 INT, c2 INT, c3 INT);
CREATE TYPE simpleCur AS CURSOR RETURNS simpleRow;
```

使 c1 成为按如下所示在过程中声明的强类型游标变量:

```
DECLARE c1 simpleCur;
```

可按如下所示将字面值指定给游标 c1:

```
SET c1 = CURSOR FOR VALUES (1, 2, 3);
```

强类型检查将成功，原因是字面值与游标定义相兼容。以下是将失败的字面值赋值的示例，失败原因是字面值数据类型与游标类型定义不兼容:

```
SET c1 = CURSOR FOR VALUES ('a', 'b', 'c');
```

将游标变量值指定给游标变量值

仅当游标变量具有完全相同的结果集定义时，才能将一个游标变量值指定给另一个游标变量值。例如:

如果 c1 和 c2 是定义为如下所示的强类型游标变量:

```
CREATE TYPE simpleRow AS ROW (c1 INT, c2 INT, c3 INT);

CREATE TYPE simpleCur AS CURSOR RETURNS simpleRow

DECLARE c1 simpleCur;

DECLARE c2 simpleCur;
```

如果 c2 已定义为如下所示的值:

```
SET c2 = CURSOR FOR VALUES (1, 2, 3);
```

可按如下所示将 c2 的结果集指定给游标变量 c1:

```
 SET c1 = c2;
```

一旦对游标变量指定了值，那么可指定和引用游标变量和游标变量字段值。

引用游标变量

在与检索和访问结果集或调用过程并将游标变量作为参数传递有关的游标操作中，可通过多种方式引用游标变量。

关于此任务

以下语句可用于在 SQL PL 上下文中引用游标变量:

- CALL
- SET
- OPEN
- FETCH
- CLOSE

访问与游标变量相关联的结果集时，OPEN、FETCH 和 CLOSE 语句会经常一起使用。OPEN 语句用于初始化与游标变量相关联的结果集。成功执行此语句时，游标变量与结果集相关联，并且可访问结果集中的行。FETCH 语句用于专门检索游标变量当前访问的行中的列值。CLOSE 语句用于结束游标变量的处理。

以下是已创建行数据类型定义以及包含游标变量定义的 SQL 过程定义的示例。SQL 过程中演示了 OPEN、FETCH 和 CLOSE 语句如何与游标变量配合使用:

```
CREATE TYPE simpleRow AS ROW (c1 INT, c2 INT, c3 INT);

CREATE PROCEDURE P(OUT p1 INT, OUT p2 INT, PUT p3 INT, OUT pRow simpleRow)
LANGUAGE SQL
BEGIN

      CREATE TYPE simpleCur AS CURSOR RETURNS simpleRow
      DECLARE c1 simpleCur;
      DECLARE localVar1 INTEGER;
      DECLARE localVar2 INTEGER;
      DECLARE localVar3 INTEGER;
      DECLARE localRow simpleRow;

      SET c1 = CURSOR FOR SELECT * FROM T;

      OPEN C1;

      FETCH c1 INTO localVar1, localVar2, localVar3;

      FETCH c1 into localRow;

      SET p1 = localVar1;

      SET p2 = localVar2;

      SET p3 = localVar3;

      SET pRow = localRow;

      CLOSE c1;

END;
```

游标变量还可作为 CALL 语句中的参数引用。与其他参数一样，游标变量参数只需通过名称引用。以下是 SQL 过程中的 CALL 语句的示例，该语句引用作为输出参数的游标变量 curVar:

```
CALL P2(curVar);
```

确定游标的已访存行的数目

确定与游标相关联的行数可通过使用 cursor_rowCount 标量函数高效完成，它将游标变量用作参数，并返回对应打开该游标后访存的行数的整数值作为输出。

开始之前

必须先满足下列先决条件，才能使用 cursor_rowCount 函数:

- 必须创建游标数据类型。
- 必须声明游标数据类型的游标变量。
- 必须已执行引用该游标的 OPEN 语句。

关于此任务

可在 SQL PL 上下文中使用 cursor_rowCount 函数，每当在过程逻辑中需要访问至今为止对游标访存的行数计数或总访存行数计数时，应执行此任务。使用 cursor_rowCount 函数可简化访问已访存行计数的过程，此过程本来可能需要在循环过程逻辑中使用已声明变量和重复执行语句来维护此计数。

限制

cursor_rowCount 函数只能在 SQL PL 上下文中使用。

过程

1. 使用对 cursor_rowCount 标量函数的引用来创建 SQL 语句。 以下是 SET 语句的示例，此示例将 cursor_rowCount 标量函数的输出指定给局部变量 rows_fetched:

   ```
   SET rows_fetched = CURSOR_ROWCOUNT(curEmp)
   ```

2. 在受支持 SQL PL 上下文中加入包含 cursor_rowCount 函数引用的 SQL 语句。例如，这可能在 CREATE PROCEDURE 语句或 CREATE FUNCTION 语句中并编译该语句。

3.

结果

该语句应成功编译。

示例

以下是包括对 cursor_rowCount 函数的引用的 SQL 过程的示例:

```
connect to sample %

set serveroutput on %

create or replace procedure rowcount_test()
language sql
begin
  declare rows_fetched bigint;
  declare Designers cursor;
  declare first anchor to employee.firstnme;
  declare last anchor to employee.lastname;

  set Designers = cursor for select firstnme, lastname
    from employee where job = 'DESIGNER' order by empno asc;
```

```
  open Designers;

  fetch Designers into first, last;
  call dbms_output.put_line(last || ', ' || first);
  fetch Designers into first, last;
  call dbms_output.put_line(last || ', ' || first);

  set rows_fetched = CURSOR_ROWCOUNT(Designers);
  call dbms_output.put_line(rows_fetched || ' rows fetched.');

  close Designers;
end %

call rowcount_test() %

connect reset %
terminate %
```

下一步做什么

执行 SQL 过程或调用 SQL 函数。

示例: 游标变量用法

设计和实现游标变量时，参考游标变量用法的示例会很有用。

SQL 过程中的游标变量用法

可通过参考演示游标变量用法的示例来了解如何使用游标变量以及可在何处使用。

本示例显示下列内容:

- CREATE TYPE 语句，用于创建 ROW 数据类型
- CREATE TYPE 语句，用于根据行数据类型规范来创建强类型游标
- CREATE PROCEDURE 语句，用于创建具有输出游标参数的过程
- CREATE PROCEDURE 语句，用于创建调用另一过程并将游标作为输入参数传递的过程

运行此示例的先决条件是 SAMPLE 数据库必须存在。要创建样本数据库，请从 DB2 命令窗口发出以下命令:

```
db2sampl;
```

以下是示例 CLP 脚本，该脚本演示 SQL 过程中游标变量用法的核心功能。该脚本包含行数据类型定义、游标类型定义及两个 SQL 过程定义。过程 P_CALLER 包含游标变量定义及对过程 P 的调用。过程 P 定义游标，打开该游标并将其作为输出参数值传递。过程 P_CALLER 接收游标参数，将游标值访存到局部变量中，然后根据局部变量值设置以下两个输出参数值: edlvel 和 lastname。

```
--#SET TERMINATOR @
update command options using c off @
connect to sample @

CREATE TYPE myRowType AS ROW (edlevel SMALLINT, name VARCHAR(128))@

CREATE TYPE myCursorType AS myRowType CURSOR@

CREATE PROCEDURE P(IN pempNo VARCHAR(8), OUT pcv1 SYSCURSOR)
LANGUAGE SQL
BEGIN
  SET pcv1 = CURSOR FOR SELECT edlevel, lastname FROM employee WHERE empNo = pempNo;
```

```
  OPEN pcv1;

END@

CREATE PROCEDURE P_CALLER( IN pempNo VARCHAR(8) ,
                           OUT edlevel SMALLINT,
                           OUT lastname VARCHAR(128))
LANGUAGE SQL
BEGIN
  DECLARE rv1 myRowType;
  DECLARE c1 SYSCURSOR;

  CALL P (pempNo,c1);
  FETCH c1 INTO rv1;
  CLOSE c1;

  SET edlevel = rv1.edlevel;
  SET lastname = rv1.name;

END @

CALL P_CALLER('000180',?,?) @
```

运行此脚本后，会生成以下输出:

```
update command options using c off
DB20000I  The UPDATE COMMAND OPTIONS command completed successfully.

connect to sample

   Database Connection Information

 Database server        = DB2/LINUXX8664 9.7.0
 SQL authorization ID   = REGRESS5
 Local database alias   = SAMPLE


CREATE TYPE myRowType AS ROW (edlevel SMALLINT, name VARCHAR(128))
DB20000I  The SQL command completed successfully.

CREATE TYPE myCursorType AS CURSOR RETURNS myRowType
DB20000I  The SQL command completed successfully.

CREATE PROCEDURE P(IN pempNo VARCHAR(8),OUT pcv1 SYSCURSOR)
LANGUAGE SQL
BEGIN
  SET pcv1 = CURSOR FOR SELECT edlevel, lastname FROM employee WHERE empNo = pempNo;
  OPEN pcv1;

END
DB20000I  The SQL command completed successfully.

CREATE PROCEDURE P_CALLER( IN pempNo VARCHAR(8) ,
                           OUT edlevel SMALLINT,
                           OUT lastname VARCHAR(128))
LANGUAGE SQL
BEGIN
  DECLARE rv1 myRowType;
  DECLARE c1 SYSCURSOR;

  CALL P (pempNo,c1);
  FETCH c1 INTO rv1;
  CLOSE c1;

  SET EDLEVEL = rv1.edlevel;
  SET LASTNAME = rv1.name;
```

```
END
DB20000I  The SQL command completed successfully.

CALL P_CALLER('000180',?,?)

  Value of output parameters
  --------------------------
  Parameter Name  : EDLEVEL
  Parameter Value : 17

  Parameter Name  : LASTNAME
  Parameter Value : SCOUTTEN

  Return Status = 0
```

布尔数据类型

BOOLEAN 类型是一种内置数据类型，只能用于复合 SQL（编译型）语句中的局部变量、全局变量、参数或返回类型。布尔值表示 TRUE 或 FALSE 的真实值。布尔表达式或谓词可产生未知值，未知值表示为空值。

对布尔数据类型的限制

使用布尔数据类型之前或诊断因为使用布尔数据类型而导致的问题时，应注意布尔数据类型的限制。

布尔数据类型存在以下限制:

- 布尔数据类型只能作为以下对象引用:
  - SQL 函数中声明的局部变量
  - SQL 过程中声明的局部变量
  - 复合 SQL（编译型）语句作为触发器体的触发器中声明的局部变量
  - 复合 SQL（编译型）语句作为函数体的 SQL 函数的参数
  - 复合 SQL（编译型）语句作为过程体的 SQL 过程的参数
  - 返回类型
  - 模块中的全局变量
- 不能使用布尔数据类型来定义表或视图中的列数据类型。
- 关键字 TRUE 和 FALSE 不能作为值引用以插入到表中。
- 不能在外部例程或客户机应用程序中引用布尔数据类型。
- 布尔数据类型不能强制转换为其他数据类型。
- 布尔数据类型不能作为 SQL 过程的返回码值。
- 只能向布尔数据类型的变量指定下列其中一个值: TRUE、FALSE 或 NULL。不支持数字或其他数据类型赋值。
- 不支持将值选择或访存到布尔数据类型变量中。
- 不能在结果集中返回布尔数据类型。
- 布尔变量不能用作谓词。例如，不支持以下 SQL 子句:

  ```
  IF (gb) THEN ...
  ```

  仅支持在 UDF 的 SET 语句和 RETURN 语句中使用谓词。

如果这些限制导致您无法使用此数据类型，请考虑改为使用整数数据类型并向其指定值 1（表示 TRUE）、0（表示 FALSE）和 -1（表示 NULL）。

第 5 章 SQL 例程

SQL 例程是逻辑完全使用 SQL 语句（包括 SQL 过程语言 (SQL PL) 语句）实现的例程。它们的特征是，例程体逻辑包含在用于创建它们的 CREATE 语句中。这与外部例程相反，外部例程的例程逻辑是在库构建形式的编程源代码中实现的。一般来说，SQL 例程可包含和执行的 SQL 语句比外部外部例程少；但按最佳范例实现时，其中的每个 SQL 语句的功能都很强大，并且性能很高。

可创建 SQL 过程、SQL 函数和 SQL 方法。尽管它们都是通过 SQL 实现的，但每个例程功能类型具有不同功能。

SQL 例程概述

SQL 例程是逻辑完全使用 SQL 语句（包括 SQL 过程语言 (SQL PL) 语句）实现的例程。它们的特征是，例程体逻辑包含在用于创建它们的 CREATE 语句中。可创建 SQL 过程、SQL 函数和 SQL 方法。尽管它们都是通过 SQL 实现的，但每个例程功能类型具有不同功能。

在决定实现 SQL 例程之前，应先阅读“例程概述”来了解 SQL 例程的概念、实现方式以及用法。了解这些信息后，可从以下概念主题了解有关 SQL 例程的更多信息，以便在决定何时以及如何在数据库环境中使用 SQL 例程时做出正确决定:

- SQL 过程
- SQL 函数
- 用于开发 SQL 例程的工具
- SQL 过程语言 (SQL PL)
- SQL PL 与内联型 SQL PL 的比较
- SQL PL 语句和功能
- 受支持的内联型 SQL PL 语句和功能
- 确定何时使用 SQL 过程或 SQL 函数
- SQL 例程限制

了解 SQL 例程后，您可能想要执行下列其中一个任务:

- 开发 SQL 过程
- 开发 SQL 函数
- 开发 SQL 方法

SQL 例程的 CREATE 语句

SQL 例程通过对例程类型执行适当的 CREATE 语句创建。在 CREATE 语句中，还可指定例程体，对于 SQL 例程，该例程体必须完全由 SQL 或 SQL PL 语句组成。可使用 IBM® DB2 开发中心来帮助您创建、调试和运行 SQL 过程。还可使用 DB2 命令行处理器来创建 SQL 过程、函数和方法。

SQL 过程、函数和方法都必须有各自的 CREATE 语句。尽管这些语句的语法不同，但它们有一些公共元素。在每个语句中，必须指定例程名称、参数（如果将存在任何参数）以及返回类型。还可指定其他关键字，这些关键字将为 DB2 提供有关例程中包含的逻辑的信息。DB2 在调用时使用例程原型和其他关键字来标识例程，并执行具有必需的功能支持和可能的最佳性能的例程。

有关在 DB2 开发中心中或从命令行处理器中创建 SQL 过程或有关创建函数和方法的特定信息，请参阅相关主题。

确定何时使用 SQL 例程或外部例程

关于此任务

实现例程逻辑时，可选择实现 SQL 例程或外部例程。选择两种实现的其中任何一种都有原因。

要确定何时选择实现 SQL 例程或外部例程，请阅读下文以确定是否有任何因素可能限制您的选择。

过程

- 下列情况下选择实现 SQL 例程:
  - SQL PL 和 SQL 语句为实现所需逻辑提供足够支持。
  - 例程逻辑主要由查询或修改数据的 SQL 语句组成，并且会考虑性能。与查询或修改数据库数据的 SQL 语句数目相比，包含的控制流逻辑相对较少的逻辑的性能通常比 SQL 例程实现的性能更好。SQL PL 适合用于实现数据库操作中的过程逻辑，而不太适合用于编写复杂逻辑。
  - 可在外部例程实现中执行需要执行的 SQL 语句。
  - 您可能想要模块在操作系统环境间具有高可移植性并且使其对编程语言代码编译器和脚本解释器的依赖性降至最低。
  - 您想要迅速实现逻辑并轻松使用高级编程语言。
  - 您更愿意使用 SQL 而不是脚本编制或编程语言。
  - 您想要保护数据库管理系统中的逻辑。
  - 您想要将发行版升级或操作系统升级时产生的例程维护和例程包维护工作量降至最低。
  - 您想要将实现逻辑所需的代码量降至最低。
  - 您想要尽量降低内存管理、指针操作或其他常用编程陷阱所带来的风险以使所实现的代码的安全性达到最高。
  - 您想要受益于使用 SQL PL 时可用的特殊 SQL 高速缓存支持。
- 下列情况下选择实现外部过程:
  - 如果例程逻辑非常复杂并且几乎不包括 SQL 语句，同时需要考虑例程性能。涉及大量字符串操作的复杂算术算法或或不访问数据库之类的逻辑的性能通常比外部例程实现的性能好。
  - 如果可在外部例程实现中执行需要执行的 SQL 语句。
  - 例程逻辑将进行操作系统调用（这只能通过外部例程完成）。
  - 例程逻辑必须读取或写入文件（这只能通过外部例程完成）。

– 写入服务器文件系统。执行此任务时需谨慎。
– 调用驻留在数据库服务器上的应用程序或脚本。
– 发出在 SQL 过程中不受支持的特定 SQL 语句。
– 您更愿意使用 SQL PL 以外的编程语言进行编程。

结果

缺省情况下，如果 SQL 例程能够满足您的需要，请使用 SQL 例程。一般来说，需要实现复杂逻辑或访问数据库服务器上的文件和脚本时应选择使用外部例程。特别是 SQL PL 可快速轻松地学习和实现时。

确定何时使用 SQL 过程或 SQL 函数

关于此任务

面对通过 SQL 过程或 SQL 函数中的 SQL PL 实现逻辑这一选择时，您可能会因为某些原因而选择下列两种实现中的每一种。

过程

阅读下文以确定何时选择使用 SQL 过程或 SQL 函数。
下列情况下选择实现 SQL 函数:

- SQL 函数可满足功能需求，并且您预计以后不需要 SQL 过程提供的功能。
- 优先考虑性能，并且要包含在例程中的逻辑仅由查询组成或者仅返回单个结果集。

  它们仅包含查询或返回单个结果集时，SQL 函数的性能比逻辑上等价的 SQL 过程（因为 SQL 函数的编译方式）。

  在 SQL 过程中，SELECT 语句和全查询语句格式的静态查询是分别编译的，这样每个查询都成为创建 SQL 过程时程序包中查询存取方案的一个部分。直到重新创建 SQL 过程或程序包重新绑定至数据库，才会重新编译此程序包。这意味着查询的性能根据执行 SQL 过程之前对数据库管理器可用的信息确定，因此可能并非最优。而且，对于 SQL 过程，数据库管理器在执行过程流语句与查询或修改数据的 SQL 语句之间转换时会有小小开销。

  但是，SQL 函数会在引用它们的 SQL 语句中扩展并编译，这意味着每次编译该 SQL 语句时（取决于该语句动态执行的情况）会编译这些函数。因为 SQL 函数未直接与程序包相关联，所以数据库管理器在执行过程流语句与查询或修改数据的 SQL 语句之间转换时没有开销。

下列情况下选择实现 SQL 过程:

- 需要仅在 SQL 过程中受支持的 SQL PL 功能。这包括: 输出参数支持、使用游标、将多个结果集返回给调用程序的能力、完整条件处理支持、事务和保存点控制及其他功能。
- 您想要执行只能在 SQL 过程中执行的非 SQL PL 语句。
- 您想要修改数据，并且您需要的函数类型不支持修改数据。

结果

尽管并不总是那么显而易见，但通常可轻松地将 SQL 过程重写为执行等价逻辑的 SQL 函数。如果您在意每一次小小的性能改进，那么这可能是使性能达到最佳的有效方式。

确定何时使用 SQL 例程或动态预编译复合 SQL 语句

确定如何实现在使用 SQL 例程或动态预编译复合 SQL 语句之间进行选择时可能面对的基本 SQL PL 块和其他 SQL 语句。尽管 SQL 例程在内部使用复合 SQL 语句，但选择使用哪一项可能取决于其他因素。

性能

如果动态预编译复合 SQL 语句的功能可满足您的需要，那么最好使用动态预编译复合 SQL 语句，原因是动态预编译复合 SQL 语句中出现的 SQL 语句作为单个块编译和执行。而且，这些语句的性能比逻辑上等价的 SQL 过程的 CALL 语句更好。

创建 SQL 过程时，会编译该过程并且创建包。该包会包含从 SQL 过程编译时起用于访问数据的最佳执行路径。动态预编译复合 SQL 语句在执行时编译。对于这些语句而言，用于访问数据的最佳执行路径通过使用最新数据库信息确定，这可能意味着它们的存取方案可能比之前创建的逻辑上等价的 SQL 过程更好，从而使得它们的性能更好。

必需逻辑的复杂度

如果逻辑非常简单并且 SQL 语句相对较少，那么应考虑在动态预编译复合 SQL 语句（指定 ATOMIC）或 SQL 函数中使用 SQL PL。SQL 过程也可处理简单逻辑，但使用 SQL 过程会导致一些开销，如创建并调用该过程，如非必要，应尽量避免这些开销。

要执行的 SQL 语句数

如果只有一个或两个 SQL 语句要执行，那么使用 SQL 过程可能没什么优点。实际上可能对执行这些语句所需的总体性能有负面影响。在此情况下，最好在动态预编译复合 SQL 语句中使用内联 SQL PL。

原子性和事务控制

原子性是另一个要考虑的事项。复合 SQL（内联型）语句必须为原子语句。不支持在复合 SQL（内联型）语句中进行落实和回滚。如果需要事务控制或对回滚至保存点的支持，那么必须使用 SQL 过程。

安全性

安全性也可能成为要考虑的事项。SQL 过程只能由对过程具有 EXECUTE 特权的用户执行。如果需要限制可执行特定逻辑块的用户，那么这一点非常有用。还可管理执行动态预编译复合 SQL 语句的能力。但是，SQL 过程执行权限产生了额外的安全性控制层。

功能支持

如果需要返回一个或多个结果集，那么必须使用 SQL 过程。

模块性、寿命和重复使用

SQL 过程是永久存储在数据库中并且可由多个应用程序或脚本以一致方式引用的数据库对象。动态预编译复合 SQL 语句未存储在数据库中，因此无法稳定地重复使用它们包含的逻辑。

如果 SQL 过程能够满足您的需要，请使用 SQL 过程。一般来说，实现复杂逻辑或使用 SQL 过程支持但对动态预编译复合 SQL 语句不可用的功能时应选择使用 SQL 过程。

将 SQL 过程重写为 SQL 用户定义函数

关于此任务

为了在数据库管理系统中获得最佳性能，将简单 SQL 过程重写为 SQL 函数有时很有益（如果可能）。对于过程和函数而言，它们的例程体都是使用可能包含 SQL PL 的复合块实现的。在过程和函数中，相同 SQL PL 语句包括在由 BEGIN 和 END 关键字绑定的复合块中。

过程

将 SQL 过程转换为 SQL 函数时，有一些事项需要注意:

- 执行此操作的主要和唯一原因是逻辑仅查询数据时改进例程性能。
- 在标量函数中，您可能必须声明用于保存返回值的变量，以避免您不能直接向函数的任何输出参数指定值的事实。用户定义标量函数的输出值仅在函数的 RETURN 语句中指定。
- 如果 SQL 函数将修改数据，那么必须使用 MODIFIES SQL 子句显式创建该函数，以便它可包含修改数据的 SQL 语句。

示例

在以下示例中，将显示逻辑上等价的 SQL 过程和 SQL 标量函数。从功能上看，这两个例程在给定相同输入值的情况下提供相同输出值，但它们以略有不同的方式实现和调用。

```
CREATE PROCEDURE GetPrice (IN Vendor CHAR(20),
                           IN Pid INT,
                           OUT price DECIMAL(10,3))
LANGUAGE SQL
BEGIN

  IF Vendor = 'Vendor 1'
   THEN SET price = (SELECT ProdPrice FROM V1Table WHERE Id = Pid);
  ELSE IF Vendor = 'Vendor 2'
   THEN SET price = (SELECT Price FROM V2Table
                  WHERE Pid = GetPrice.Pid);
  END IF;
END
```

此过程接收两个输入参数值，并返回输出参数值，输出参数值是根据输入参数值有条件确定的。它使用 IF 语句。此 SQL 过程通过执行 CALL 语句调用。例如，可通过 CLP 执行以下操作:

```
CALL GetPrice( 'Vendor 1',  9456, ?)
```

SQL 过程可重写为逻辑上等价的 SQL 表函数，如下所示:

```
CREATE FUNCTION GetPrice (Vendor CHAR(20), Pid INT)
    RETURNS  DECIMAL(10,3)
LANGUAGE SQL  MODIFIES SQL
BEGIN
  DECLARE price DECIMAL(10,3);

  IF Vendor = 'Vendor 1'
    THEN SET price = (SELECT ProdPrice FROM V1Table WHERE Id = Pid);
  ELSE IF Vendor = 'Vendor 2'
    THEN SET price = (SELECT Price FROM V2Table
              WHERE Pid = GetPrice.Pid);
  END IF;

  RETURN price;
END
```

此函数接收两个输入参数值，并根据输入参数值有条件返回单个标量值。它需要声明并使用局部变量 price 来保存要返回的值直到函数返回，而 SQL 过程可将输出参数用作变量。从功能上看，这两个例程执行相同逻辑。

当然现在其中每个例程的执行接口不同。与简单地通过 CALL 语句调用 SQL 过程不同，SQL 函数必须在允许使用表达式的 SQL 语句中调用。在大多数情况下，这不是问题，如果有意立即操作例程返回的数据，这可能实际上有益。以下是有关如何调用 SQL 函数的两个示例。

可使用 VALUES 语句对它进行调用:

```
VALUES (GetPrice('Vendor 1', 9456))
```

还可在 SELECT 语句中对它进行调用，例如，可能从表中查询值并根据函数结果过滤行:

```
SELECT VName FROM Vendors WHERE GetPrice(Vname, Pid) < 10
```

SQL 过程

SQL 过程是完全通过 SQL 实现的过程，可用于封装可以像编程子例程一样调用的逻辑。在数据库或数据库应用程序体系结构中有许多实用的 SQL 过程应用程序。可使用 SQL 过程来创建简单脚本，以迅速查询、变换、更新数据、生成基本报告、改变应用程序性能、使应用程序模块化、改进整体数据库设计和数据库安全性。

SQL 过程的许多功能使得 SQL 过程成为功能强大的例程选件。

决定实现 SQL 过程之前，应先了解例程并参阅“SQL 过程概述”主题以了解 SQL 过程在 SQL 例程上下文中的概念以及如何实现和使用 SQL 过程。

SQL 过程的功能

SQL 过程具有许多功能。SQL 过程:

- 可包含 SQL 过程语言语句及功能，它们支持在传统静态和动态 SQL 语句周围实现控制流逻辑。
- 在整个 DB2 系列品牌的数据库产品中受支持，在这些产品中，在 DB2 版本 9 中受支持的大部分功能也受支持。
- 易于实现，原因是它们使用简单高级强类型语言。

- SQL 过程比等价的外部过程更可靠。
- 遵循 SQL99 ANSI/ISO/IEC SQL 标准。
- 支持输入、输出和输入/输出参数传递方式。
- 支持简单但功能强大的条件和错误处理模型。
- 允许将多个结果集返回至调用程序或客户机应用程序。
- 允许您轻松地将 SQLSTATE 和 SQLCODE 值作为特殊变量访问。
- 驻留在数据库中，并且自动备份和复原。
- 可在支持 CALL 语句的位置调用。
- 支持对其他 SQL 过程或使用其他语言实现的过程进行嵌套过程调用。
- 支持递归。
- 支持保存点及已执行 SQL 语句的回滚以提供大量事务控制。
- 可通过触发器调用。

SQL 过程提供广泛支持（不限于以上列出的功能部件）。根据最佳范例实现时，它们可充当数据库体系结构、数据库应用程序设计和数据库系统性能中的重要角色。

设计 SQL 过程

设计 SQL 过程要求了解您的需求、SQL 过程功能、如何使用 SQL 功能以及有关可能妨碍设计的任何限制。

SQL 过程的部分

要了解 SQL 过程，了解 SQL 过程的部分会有所帮助。以下只是 SQL 过程的某些部分:

- SQL 过程的结构
- SQL 过程中的参数
- SQL 过程中的变量
- SQL 过程中的 SQLCODE 和 SQLSTATE
- SQL 过程中的原子块和变量作用域
- SQL 过程中的游标
- SQL PL 中的逻辑元素
- SQL 过程中的条件和错误处理程序
- 可在 SQL 过程中执行的 SQL 语句

SQL 过程的结构

SQL 过程由若干逻辑部分组成，并且 SQL 过程开发要求您根据结构化格式实现这些部分。该格式非常直观，易于遵循并且旨在简化例程的设计和语义。

SQL 过程的核心是复合语句。复合语句由关键字 BEGIN 和 END 绑定。这些语句可以为 ATOMIC 或 NOT ATOMIC。缺省情况下，它们为 NOT ATOMIC。

在复合语句中，可使用 SQL 语句声明并引用多个可选 SQL PL 对象。下图说明 SQL 过程中复合语句的结构化格式:

```
label:  BEGIN
  Variable declarations
  Condition declarations
```

```
   Cursor declarations
   Condition handler declarations
   Assignment, flow of control, SQL statements and other compound statements
 END label
```

该图显示 SQL 过程可由一个或多个可选 ATOMIC 复合语句（块）组成，并且可在单个 SQL 过程中嵌套或顺序引入这些块。在每个原子块中，可选变量、条件和处理程序声明都有规定顺序。它们必须在引入使用 SQL 控制语句和其他 SQL 语句及游标声明实现的过程逻辑之前。可使用 SQL 过程体中包含的一组 SQL 语句在任意位置声明游标。

为明晰控制流，可像 SQL 过程原子块中包含的许多 SQL 控制语句那样标记这些原子块。这使得您能够轻松地精确引用变量和控制转移语句引用。

以下是 SQL 过程的示例，该过程演示了上面列出的每个元素:

```
CREATE PROCEDURE DEL_INV_FOR_PROD (IN prod INT, OUT err_buffer VARCHAR(128))
LANGUAGE SQL
DYNAMIC RESULT SETS 1
BEGIN

  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLCODE integer DEFAULT 0;
  DECLARE NO_TABLE CONDITION FOR SQLSTATE '42704';
  DECLARE cur1 CURSOR WITH RETURN TO CALLER
                  FOR SELECT * FROM Inv;

  A: BEGIN ATOMIC
       DECLARE EXIT HANDLER FOR NO_TABLE
         BEGIN
            SET ERR_BUFFER='Table Inv does not exist';
         END;

       SET err_buffer = '';

       IF (prod < 200)
         DELETE FROM Inv WHERE product = prod;
       ELSE IF (prod < 400)
         UPDATE Inv SET quantity = 0 WHERE product = prod;
       ELSE
         UPDATE Inv SET quantity = NULL WHERE product = prod;
       END IF;

    B:  OPEN cur1;

END
```

SQL 过程中的 NOT ATOMIC 复合语句

先前示例说明了 NOT ATOMIC 复合语句，它是 SQL 过程中使用的缺省类型。如果复合语句中发生了未处理的错误情况，那么将不回滚该错误之前完成的所有工作，但也不会落实这些工作。仅当使用 ROLLBACK 或 ROLLBACK TO SAVEPOINT 语句显式回滚工作单元时，才会回滚语句组。还可使用 COMMIT 语句来落实成功语句（如果这样做有用）。

以下是带有 NOT ATOMIC 复合语句的 SQL 过程的示例:

```
CREATE PROCEDURE not_atomic_proc ()
LANGUAGE SQL
SPECIFIC not_atomic_proc
 nap:  BEGIN NOT ATOMIC
```

```
  INSERT INTO c1_sched (class_code, day)
    VALUES ('R11:TAA', 1);

  SIGNAL SQLSTATE '70000';

  INSERT INTO c1_sched (class_code, day)
    VALUES ('R22:TBB', 1);

END nap
```

SIGNAL 语句执行时，它会显式产生未处理的错误。之后该过程立即返回。过程返回后，尽管发生了错误，但第一个 INSERT 语句仍会成功执行并将一行插入到 c1_sched 表中。该过程既不会落实，也不会回滚行插入操作，这将留待调用该 SQL 过程的完整工作单元完成。

SQL 过程中的 ATOMIC 复合语句

就如名称所示，ATOMIC 复合语句可视为单个整体。如果其中发生任何未处理错误，那么已执行至该点的所有语句也将被视为失败，并因此回滚。

ATOMIC 复合语句不能嵌套在其他 ATOMIC 复合语句中。

不能在 ATOMIC 复合语句中使用 SAVEPOINT 语句、COMMIT 语句或 ROLLBACK 语句。它们仅在 SQL 过程的 NOT ATOMIC 复合语句中受支持。

以下是带有 ATOMIC 复合语句的 SQL 过程的示例:

```
CREATE PROCEDURE atomic_proc ()
LANGUAGE SQL
SPECIFIC atomic_proc

ap:  BEGIN ATOMIC

   INSERT INTO c1_sched (class_code, day)
     VALUES ('R33:TCC', 1);

   SIGNAL SQLSTATE '70000';

   INSERT INTO c1_sched (class_code, day)
     VALUES ('R44:TDD', 1);

END ap
```

SIGNAL 语句执行时，它会显式产生未处理的错误。之后该过程立即返回。尽管成功执行产生的表没有对此过程插入的行，但第一个 INSERT 语句会回滚。

标签和 SQL 过程复合语句

可选择使用标签来命名 SQL 过程中的任何可执行语句，包括复合语句和循环。通过在其他语句中引用标签，可强制执行流跳出命令语句或循环，或者另外跳至复合语句或循环的开头。GOTO、ITERATE 和 LEAVE 语句可引用标签。

可选择对复合语句的 END 提供相应标签。如果已提供结尾标签，那么它必须与用于其开头的标签相同。

每个标签在 SQL 过程的主体中必须唯一。

如果在存储过程的多个复合语句中声明了同名变量，那么还可使用来标签来避免歧义。可使用标签来限定 SQL 变量的名称。

SQL 过程中的参数

SQL 过程支持用于将 SQL 值传递至过程或传递出过程的参数。

根据特定输入或输入标量值集合有条件实现逻辑时，或需要返回一个或多个输出标题值并且不希望返回结果集时，在 SQL 过程中使用参数会很有用。

设计或创建 SQL 过程时，应了解 SQL 过程中的参数功能和限制。

- DB2 支持在 SQL 过程中有选择地使用大量输入、输出和输入-输出参数。CREATE PROCEDURE 语句例程特征符部分中的关键字 IN、OUT 和 INOUT 指示参数的方式或期望用途。IN 和 OUT 参数按值传递，INOUT 参数按引用传递。
- 对过程指定多个参数时，每个参数必须具有唯一名称。
- 如果要在过程中声明同名的变量和参数，那么必须在过程中所嵌套的有标号原子块内声明该变量。否则，DB2 将检测哪些对象本来可能会成为有歧义的名称引用。
- 不管 SQL 过程参数的数据类型如何，都不能指定为 SQLSTATE 或 SQLCODE。

请参阅 CREATE PROCEDURE (SQL) 语句，以了解有关 SQL 过程中的参数引用的完整详细信息。

以下 SQL 过程 myparams 演示如何使用 IN、INOUT 和 OUT 参数方式。假定 SQL 过程是在 CLP 文件 myfile.db2 中定义的，并且我们将使用命令行。

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

参数标记

参数标记通常用问号（?）或冒号并后跟变量名（:*var1*）表示，在 SQL 语句中充当其值在执行语句期间获取的占位符。应用程序使参数标记与应用程序变量相关联。在执行语句期间，这些变量的值将分别替换每个参数标记。该过程期间可能会发生数据转换。

参数标记的优点

对于需要多次执行的 SQL 语句，预编译 SQL 语句一次，然后通过在运行时期间使用参数标记来替代输入值以重复使用查询方案通常很有利。在 DB2® 9 中，参数标记用下列两种方法的其中一种表示:

- 第一种样式带有“?”字符，用于动态 SQL 执行（动态嵌入式 SQL、CLI、Perl 等）。
- 第二种样式表示嵌入式 SQL 标准构造，其中变量名称以冒号为前缀（:*var1*）。此样式用于静态 SQL 执行，并且通常称为主变量。

使用任一样式指示应用程序变量将在 SQL 语句内被替代。参数标记是按编号引用的，并且按从左至右，从 1 开始的号码顺序编号。执行 SQL 语句之前，应用程序必须将变量存储区域绑定至 SQL 语句中指定的每个参数标记。此外，绑定变量必须是有效存储区域，并且在对数据库执行预编译语句时必须包含输入数据值。

以下示例说明包含两个参数标记的 SQL 语句。

```
SELECT * FROM customers WHERE custid = ? AND lastname = ?
```

受支持的类型

DB2 支持隐式类型参数标记，可在 SQL 语句的所选位置使用这些标记。表 1 列示对参数标记用法的限制。

表 2. 对参数标记用法的限制

隐式类型参数标记位置	数据类型
表达式: 单独在选择列表中	错误
表达式: 算术运算符的两个操作数	错误
谓词: IN 谓词的左端操作数	错误
谓词: 关系运算符的两个操作数	错误
函数: 聚集函数的操作数	错误

示例

DB2® 提供了一组丰富的标准接口，其中包括用于高效访问数据的 CLI/ODBC、JDBC 和 ADO.NET。以下代码段显示如何将预编译语句与每个数据访问 API 的参数标记配合使用。

考虑表 t1 的以下表模式，其中列 c1 是表 t1 的主键。

表 3. 示例表模式

列名	DB2 数据类型	是否可空
c1	INTEGER	false
c2	SMALLINT	true
c3	CHAR(20)	true
c4	VARCHAR(20)	true
c5	DECIMAL(8,2)	true
c6	DATE	true
c7	TIME	true
c8	TIMESTAMP	true
c9	BLOB(30)	true

以下示例演示如何使用预编译语句向表 t1 插入一行。

CLI 示例

```
void parameterExample1(void)
{
   SQLHENV henv;
   SQLHDBC hdbc;
   SQLHSTMT hstmt;
   SQLRETURN rc;
   TCHAR server[] = _T("C:\\mysample\\");
   TCHAR uid[] = _T("db2e");
   TCHAR pwd[] = _T("db2e");
   long p1 = 10;
   short p2 = 100;
   TCHAR p3[100];
   TCHAR p4[100];
   TCHAR p5[100];
   TCHAR p6[100];
```

```
TCHAR p7[100];
TCHAR p8[100];
char  p9[100];
long len = 0;

_tcscpy(p3, _T("data1"));
_tcscpy(p4, _T("data2"));
_tcscpy(p5, _T("10.12"));
_tcscpy(p6, _T("2003-06-30"));
_tcscpy(p7, _T("12:12:12"));
_tcscpy(p8, _T("2003-06-30-17.54.27.710000"));

memset(p9, 0, sizeof(p9));
p9[0] = 'X';
p9[1] = 'Y';
p9[2] = 'Z';

rc = SQLAllocEnv(&henv);
// check return code ...

rc = SQLAllocConnect(henv, &hdbc);
// check return code ...

rc = SQLConnect(hdbc, (SQLTCHAR*)server, SQL_NTS,
 (SQLTCHAR*)uid, SQL_NTS, (SQLTCHAR*)pwd, SQL_NTS);
// check return code ...

rc = SQLAllocStmt(hdbc, &hstmt);
// check return code ...

// prepare the statement
rc = SQLPrepare(hstmt, _T("INSERT INTO t1 VALUES (?,?,?,?,?,?,?,?,?)"), SQL_NTS);
// check return code ...

// bind input parameters
rc = SQLBindParameter(hstmt, (unsigned short)1, SQL_PARAM_INPUT,
 SQL_C_LONG, SQL_INTEGER, 4, 0, &p1, sizeof(p1), &len);
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)2, SQL_PARAM_INPUT, SQL_C_LONG,
  SQL_SMALLINT, 2, 0, &p2, sizeof(p2), &len);
// check return code ...

len = SQL_NTS;
rc = SQLBindParameter(hstmt, (unsigned short)3, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_CHAR, 0, 0, &p3[0], 100, &len);
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)4, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_VARCHAR, 0, 0, &p4[0], 100, &len);
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)5, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_DECIMAL, 8, 2, &p5[0], 100, &len);
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)6, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_TYPE_DATE, 0, 0, &p6[0], 100, &len);
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)7, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_TYPE_TIME, 0, 0, &p7[0], 100, &len);
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)8, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_TYPE_TIMESTAMP, 0, 0, &p8[0], 100, &len);
// check return code ...
```

```
   len = 3;
   rc = SQLBindParameter(hstmt, (unsigned short)9, SQL_PARAM_INPUT, SQL_C_BINARY,
     SQL_BINARY, 0, 0, &p9[0], 100, &len);
   // check return code ...

   // execute the prepared statement
   rc = SQLExecute(hstmt);
   // check return code ...

   rc = SQLFreeStmt(hstmt, SQL_DROP);
   // check return code ...

   rc = SQLDisconnect(hdbc);
   // check return code ...

   rc = SQLFreeConnect(hdbc);
   // check return code ...

   rc = SQLFreeEnv(henv);
   // check return code ...
```

C 示例

```
EXEC SQL BEGIN DECLARE SECTION;
  char hostVarStmt1[50];
  short hostVarDeptnumb;
EXEC SQL END DECLARE SECTION;

/* prepare the statement with a parameter marker */
strcpy(hostVarStmt1, "DELETE FROM org WHERE deptnumb = ?");
EXEC SQL PREPARE Stmt1 FROM :hostVarStmt1;

/* execute the statement for hostVarDeptnumb = 15 */
hostVarDeptnumb = 15;
EXEC SQL EXECUTE Stmt1 USING :hostVarDeptnumb;
```

JDBC 示例

```
public static void parameterExample1() {

   String driver = "com.ibm.db2e.jdbc.DB2eDriver";
   String url    = "jdbc:db2e:mysample";
   Connection conn = null;
   PreparedStatement pstmt = null;

   try
   {
      Class.forName(driver);

      conn = DriverManager.getConnection(url);

      // prepare the statement
      pstmt = conn.prepareStatement("INSERT INTO t1 VALUES
                                                  (?, ?, ?, ?, ?, ?, ?, ?, ?)");

      // bind the input parameters
      pstmt.setInt(1, 1);
      pstmt.setShort(2, (short)2);
      pstmt.setString(3, "data1");
      pstmt.setString(4, "data2");
      pstmt.setBigDecimal(5, new java.math.BigDecimal("12.34"));
      pstmt.setDate(6, new java.sql.Date(System.currentTimeMillis() ) );
      pstmt.setTime(7, new java.sql.Time(System.currentTimeMillis() ) );
      pstmt.setTimestamp (8, new java.sql.Timestamp(System.currentTimeMillis() ) );
      pstmt.setBytes(9, new byte[] { (byte)'X', (byte)'Y', (byte)'Z' } );
```

```
      // execute the statement
      pstmt.execute();

      pstmt.close();

      conn.close();
   }
   catch (SQLException sqlEx)
   {
      while(sqlEx != null)
      {
          System.out.println("SQLERROR: \n" + sqlEx.getErrorCode() +
             ", SQLState: " + sqlEx.getSQLState() +
             ", Message: " + sqlEx.getMessage() +
             ", Vendor: " + sqlEx.getErrorCode() );
         sqlEx = sqlEx.getNextException();
      }
   }
   catch (Exception ex)
   {
      ex.printStackTrace();
   }
}
```

ADO.NET 示例 [C#]

```
public static void ParameterExample1()
{
   DB2eConnection conn = null;
   DB2eCommand cmd  = null;
   String connString   = @"database=.\; uid=db2e; pwd=db2e";
   int i = 1;

   try
   {
      conn = new DB2eConnection(connString);

      conn.Open();

      cmd = new DB2eCommand("INSERT INTO t1 VALUES
                                     (?, ?, ?, ?, ?, ?, ?, ?, ?)", conn);

      // prepare the command
      cmd.Prepare();

      // bind the input parameters
      DB2eParameter p1 = new DB2eParameter("@p1", DB2eType.Integer);
      p1.Value = ++i;
      cmd.Parameters.Add(p1);

      DB2eParameter p2 = new DB2eParameter("@p2", DB2eType.SmallInt);
      p2.Value = 100;
      cmd.Parameters.Add(p2);

      DB2eParameter p3 = new DB2eParameter("@p3", DB2eType.Char);
      p3.Value = "data1";
      cmd.Parameters.Add(p3);

      DB2eParameter p4 = new DB2eParameter("@p4", DB2eType.VarChar);
      p4.Value = "data2";
      cmd.Parameters.Add(p4);

      DB2eParameter p5 = new DB2eParameter("@p5", DB2eType.Decimal);
      p5.Value = 20.25;
      cmd.Parameters.Add(p5);

      DB2eParameter p6 = new DB2eParameter("@p6", DB2eType.Date);
```

```
      p6.Value = DateTime.Now;
      cmd.Parameters.Add(p6);

      DB2eParameter p7 = new DB2eParameter("@p7", DB2eType.Time);
      p7.Value = new TimeSpan(23, 23, 23);
      cmd.Parameters.Add(p7);

      DB2eParameter p8 = new DB2eParameter("@p8", DB2eType.Timestamp);
      p8.Value = DateTime.Now;
      cmd.Parameters.Add(p8);

      byte []barr = new byte[3];
      barr[0] = (byte)'X';
      barr[1] = (byte)'Y';
      barr[2] = (byte)'Z';

      DB2eParameter p9 = new DB2eParameter("@p9", DB2eType.Blob);
      p9.Value = barr;
      cmd.Parameters.Add(p9);

      // execute the prepared command
      cmd.ExecuteNonQuery();
   }
   catch (DB2eException e1)
   {
      for (int i=0; i < e1.Errors.Count; i++)
      {
         Console.WriteLine("Error #" + i + "\n" +
            "Message: " + e1.Errors[i].Message + "\n" +
            "Native: " + e1.Errors[i].NativeError.ToString() + "\n" +
            "SQL: " + e1.Errors[i].SQLState + "\n");
      }
   }
   catch (Exception e2)
   {
       Console.WriteLine(e2.Message);
   }
   finally
   {
      if (conn != null && conn.State != ConnectionState.Closed)
      {
            conn.Close();
            conn = null;
      }
   }
}
```

SQL 过程中的变量（DECLARE 和 SET 语句）

SQL 过程中的局部变量支持允许您在 SQL 过程逻辑的支持下指定和检索值。

SQL 过程中的变量通过 DECLARE 语句定义。

可使用 SET 语句或 SELECT INTO 语句将值指定给变量，也可在声明变量时将其指定为缺省值。可对变量指定字面值、表达式、查询结果和专用寄存器值。

可将变量值指定给 SQL 过程参数或 SQL 过程中的其他变量，也可在例程内执行的 SQL 语句中将变量值作为参数引用。

以下示例演示指定和检索变量值的各种方法。

```
CREATE PROCEDURE proc_vars()
SPECIFIC 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)

  DECLARE v_rowsChanged BOOLEAN DEFAULT FALSE; -- (2)

  SET v_total = v_total + 1;                   -- (3)

  SELECT MAX(salary)                           -- (4)
    INTO v_max FROM employee;

  VALUES CURRENT_DATE INTO v_date;             -- (5)

  SELECT CURRENT DATE, CURRENT DATE            -- (6)
       INTO v_adate, v_another
  FROM SYSIBM.SYSDUMMY1;

  DELETE FROM T;
  GET DIAGNOSTICS v_rcount = ROW_COUNT;        -- (7)

  IF v_rcount > 0 THEN                         -- (8)
    SET is_done = TRUE;
  END IF;
END
```

声明变量时，可按行 (1) 中所示使用 DEFAULT 子句指定缺省值。行 (2) 显示布尔数据类型的变量的声明，该变量的缺省值为 FALSE。行 (3) 显示可用于指定单个变量值的 SET 语句。还可按行 (4) 中所示，通过将 SELECT 或 FETCH 语句与 INTO 子句配合执行来设置变量。行 (5) 和行 (6) 显示如何使用 VALUES INTO 语句来对函数或专用寄存器求值以及如何将该值指定给一个变量或多个变量。

还可将 GET DIAGNOSTICS 语句的结果指定给变量。可使用 GET DIAGNOSTICS 控制受影响行数（UDPDATE 对应 UPDATE 语句，DELETE 对应 DELETE 语句）或获取刚刚执行的 SQL 语句的返回状态。行 (7) 显示如何将刚才执行的 DELETE 语句修改的行数指定给变量。

行 (8) 演示如何使用逻辑块来确定要指定给变量的值。在此情况下，如果在先前执行 DELETE 语句和 GET DIAGNOSTICS 语句（导致对变量 v_rcount 指定大于零的值）时行数发生了更改，那么会对变量 is_done is 指定值 TRUE。

SQL 过程中的 SQLCODE 和 SQLSTATE 变量

为执行错误处理或帮助您调试 SQL 过程，您可能会发现，测试 SQLCODE 或 SQLSTATE 的值，然后将这些值作为输出参数或诊断消息字符串的一部分返回，或者将这些值插入到表中以提供基本跟踪支持非常有用。

要在 SQL 过程中使用 SQLCODE 和 SQLSTATE 值，必须在 SQL 过程体中声明以下 SQL 变量:

```
DECLARE SQLCODE INTEGER DEFAULT 0;
DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
```

每当执行语句时，DB2 会隐式设置这些变量。如果语句出现存在对应处理程序的条件，那么会在处理程序执行开始时提供 SQLSTATE 和 SQLCODE 变量的值。但是，执行处理程序中的第一个语句后这些变量会重置。因此，通常会将 SQLSTATE 和 SQLCODE

的值复制到处理程序第一个语句的局部变量中。在以下示例中，对应任意条件的 CONTINUE 处理程序用于将 SQLCODE 变量复制到另一变量 retcode 中。然后可在可执行语句中使用变量 retcode 来控制过程逻辑或将该值作为输出参数返回。

```
BEGIN
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE retcode INTEGER DEFAULT 0;

  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION, SQLWARNING, NOT FOUND
     SET retcode = SQLCODE;

  可执行语句
END
```

注: 在 SQL 过程中访问 SQLCODE 或 SQLSTATE 变量时，DB2 将 SQLCODE 的值设置为 0 并将 SQLSTATE 设置为“00000”（对于后续语句）。

SQL 过程中的复合语句和变量范围

SQL 过程可包含一个或多个编译型复合语句。可顺序引入这些语句，也可将它们嵌套在另一复合语句中。每个复合语句引入一个新作用域，在该作用域中，变量可能可用，也可能不可用。

使用标签来标识复合语句很重要，因为标签可用来限定和唯一标识复合语句内声明的变量。引用不同复合语句或嵌套复合语句中的变量时，这一点特别重要。

在以下示例中，变量 *a* 有两个声明。其中一个实例在 *lab1* 标记的外部复合语句中声明，第二个实例在 *lab2* 标记的内部复合语句中声明。编写时 DB2 假定赋值语句中对 *a* 的引用是 *lab2* 标记的复合块的局部作用域中的对象。但是，如果变量 *a* 的期望实例是使用 *lab1* 标记的复合语句块中声明的对象，那么为了在最内部的复合块中正确地引用该对象，必须使用该块的标记来限定该变量。即，它应该限定为 *lab1.a*。

```
CREATE PROCEDURE P1 ()
LANGUAGE SQL
  lab1: BEGIN
    DECLARE a INT DEFAULT 100;
    lab2: BEGIN
      DECLARE a INT DEFAULT NULL;

      SET a = a + lab1.a;

      UPDATE T1
       SET T1.b = 5
        WHERE T1.b = a;  <-- Variable a refers to lab2.a
                             unless qualified otherwise

    END lab2;
 END lab1
```

可通过在 BEGIN 关键字后添加关键字 ATOMIC 将 SQL 过程中最外部的复合语句声明为原子的。如果在执行包含原始复合语句的语句时发生任何错误，那么整个复合语句会回滚。

SQL 过程中的游标

在 SQL 过程中，游标使您能够定义结果集（一组数据行）并以行为基础对行执行复杂逻辑。通过使用相同机制，SQL 过程还可定义结果集并直接返回至 SQL 过程的调用程序或客户机应用程序。

可将游标视为一组行中的某行的指针。该游标一次只能引用一行，但可根据需要移至结果集的其他行。

要在 SQL 过程中使用游标，需要执行下列操作:

1. 声明定义结果集的游标。
2. 打开游标以建立结果集。
3. 根据需要通过游标将数据访存到局部变量中（一次一行）。
4. 完成时关闭游标

要使用游标，必须使用以下 SQL 语句:

- DECLARE CURSOR
- OPEN
- FETCH
- CLOSE

以下示例演示只读游标在 SQL 过程内的基本用法:

```
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;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';

   SET p_sum = 0;

   OPEN c;

   FETCH FROM c INTO p_sal;

   WHILE(SQLSTATE = '00000') DO
      SET p_sum = p_sum + p_sal;
      FETCH FROM c INTO p_sal;
   END WHILE;

   CLOSE c;

   SET sum = p_sum;

END%
```

以下是在 SQL 过程内游标的较复杂用法的示例。此示例演示游标和 SQL PL 语句的组合用法。

SQL 过程体中的 SQL PL 逻辑元素

顺序执行是程序执行可采用的最基本路径。通过此方法，程序从代码的第一行开始执行，然后执行下一行代码，然后继续直到已执行代码中的最后一个语句。此方法对非常简单的任务很有效，但因为它只能处理一种情况，所以用处不大。程序能够需要能够根据环境不断变化而做出相应响应。通过控制代码的执行路径，可使用特定代码块来以智能方式处理多种情况。

SQL PL 支持可用于控制语句执行顺序的变量和控制流语句。IF 和 CASE 之类的语句可用于有条件执行 SQL PL 语句块，而 WHILE 和 REPEAT 之类的其他语句通常用于重复执行一组语句直到任务完成。

尽管有多种类型的 SQL PL 语句，但它们只能分为几种类别:

- 与变量相关的语句
- 条件语句
- 循环语句
- 控制转移语句

SQL 过程中的变量相关语句: 变量相关 SQL 语句可用于声明变量及对变量指定值。以下是一些类型的变量相关语句:

- SQL 过程中的 DECLARE <variable> 语句
- SQL 过程中的 DECLARE <condition> 语句
- SQL 过程中的 DECLARE <condition handler> 语句
- SQL 过程中的 DECLARE CURSOR
- SQL 过程中的 SET (assignment-statement)

这些语句为使用其他类型 SQL PL 语句和 SQL 语句（这些语句将利用变量值）提供必要的支持。

SQL 过程中的条件语句: 条件语句用于根据某些条件的满足状态定义要执行的逻辑。SQL 过程中支持两种类型的条件语句:

- CASE
- IF

这些语句是相似的；但是，CASE 语句会扩展 IF 语句。

SQL 过程中的 CASE 语句: 可使用 CASE 语句来根据条件满足状态而有条件进入某个逻辑。有两种类型的 CASE 语句:

- 简单 CASE 语句: 用于根据字面值进入某个逻辑
- 搜索式 CASE 语句: 用于根据表达式的值进入某个逻辑

CASE 语句的 WHEN 子句定义满足条件时用于确定控制流的值。

以下是带有简单 CASE 语句 WHEN 子句的 CASE 语句的 SQL 过程示例:

```
CREATE PROCEDURE UPDATE_DEPT (IN p_workdept)
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
```

以下是带有搜索式 CASE 语句 WHEN 子句的 CASE 语句的示例:

```
CREATE PROCEDURE UPDATE_DEPT (IN p_workdept)
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
```

上面提供的示例逻辑上是等价的，但要注意的是，带有搜索式 CASE 语句 WHEN 子句的 CASE 语句功能非常强大。可在其中使用任何受支持 SQL 表达式。这些表达式可包含变量、参数、专用寄存器及其他对象的引用。

SQL 过程中的 IF 语句: 可使用 IF 语句来根据条件满足状态而有条件进入某个逻辑。IF 语句在逻辑上等价于带有搜索式 CASE 语句 WHEN 子句的 CASE 语句。

IF 语句支持使用可选 ELSE IF 子句和缺省 ELSE 子句。END IF 子句是指示语句结尾所必需的。

以下是包含 IF 语句的过程的示例:

```
CREATE PROCEDURE UPDATE_SAL (IN empNum CHAR(6),
                             INOUT rating SMALLINT)
LANGUAGE  SQL
BEGIN
  IF rating = 1 THEN
    UPDATE employee
    SET salary = salary * 1.10, bonus = 1000
      WHERE empno = empNum;
  ELSEIF rating = 2 THEN
    UPDATE employee
    SET salary = salary * 1.05, bonus = 500
      WHERE empno = empNum;
  ELSE
    UPDATE employee
    SET salary = salary * 1.03, bonus = 0
      WHERE empno = empNum;
  END IF;
END
```

SQL 过程中的循环语句: 循环语句支持重复执行某个逻辑直到满足条件。SQL PL 支持以下循环语句:

- FOR
- LOOP
- REPEAT
- WHILE

FOR 语句与其他语句不同，原因是它用于迭代已定义结果集的行，而其他语句用于迭代一系列 SQL 语句直到每项都满足条件。

可对所有循环控制语句定义标签以标识这些语句。

SQL 过程中的 FOR 语句: FOR 语句是特殊类型的循环语句，原因是它们用于迭代已定义只读结果集中的行。执行 FOR 语句时，将隐式声明游标以使对于 FOR 循环的每次迭代，访存的下一行是结果集。循环会持续直到结果集中没有余下任何行。

FOR 语句简化了游标的实现，并且使得检索可对其执行逻辑运算的一组行的一组列值变得很轻松。

以下是仅包含简单 FOR 语句的 SQL 过程的示例:

```
CREATE PROCEDURE P()
LANGUAGE SQL
BEGIN ATOMIC
  DECLARE fullname CHAR(40);

  FOR v AS cur1 CURSOR FOR
              SELECT firstnme, midinit, lastname FROM employee
  DO
    SET fullname = v.lastname || ',' || v.firstnme
                   ||' ' || v.midinit;
    INSERT INTO tnames VALUES (fullname);
  END FOR;
END
```

注意: 可使用 CONCAT 函数更好地实现此示例中显示的逻辑。此简单示例用于演示语法。

FOR 循环名称对为实现 FOR 语句而生成的隐式复合语句指定标签。它遵循复合语句的标签的规则。可使用 FOR 循环名称来限定 SELECT 语句返回的结果集中的列名。

游标名仅命名用于从结果集中查询行的游标。如果未指定此名称，那么 DB2 数据库管理器将自动在内部生成唯一游标名。

SELECT 语句的列名必须唯一，并且需要用于指定某个表（或多个表，如果执行 JOIN 或 UNION 之类的操作）的 FROM 子句。执行循环之前，引用的表和列必须存在。可引用全局临时表和已声明临时表。

FOR 循环支持定位式更新和删除以及搜索式更新和删除。为确保结果正确，FOR 循环游标规范必须包括 FOR UPDATE 子句。

不能在 FOR 循环外部引用在 FOR 语句支持下创建的游标。

SQL 过程中的 LOOP 语句: LOOP 语句是特殊类型的循环语句，原因是它没有终止条件子句。它会定义重复执行的一系列语句直到另一块逻辑（通常是控制转移语句）强制控制流跳至循环外部某点。

LOOP 语句通常与下列其中一个语句配合使用: LEAVE、GOTO、ITERATE 或 RETURN。这些语句可强制控制权跳至 SQL 过程中紧跟循环之后的指定位置、跳至循环的开头以开始另一次循环迭代或退出 SQL 过程。为指示使用这些语句时要将控制流传递至的位置，使用了标签。

当循环中有复杂逻辑（您可能需要用它来以多种方式退出）时，LOOP 语句很有用，但使用时应该十分谨慎以避免出现无限循环。

如果在没有控制转移语句的情况下单独使用 LOOP 语句，那么循环中包括的一系列语句将无限执行或者直到某个数据库条件发生（该条件引发强制控制流更改的条件处理程序）或不受控制的条件发生（该条件强制 SQL 过程返回）。

以下是包含 LOOP 语句的 SQL 过程的示例。它还会使用 ITERATE 和 LEAVE 语句。

```
CREATE  PROCEDURE ITERATOR()
LANGUAGE SQL
BEGIN
  DECLARE v_deptno CHAR(3); DECLARE v_deptname VARCHAR(29);
  DECLARE at_end INTEGER DEFAULT 0;
  DECLARE not_found CONDITION FOR SQLSTATE '02000';

  DECLARE c1 CURSOR FOR SELECT deptno, deptname
                        FROM department ORDER BY deptno;
  DECLARE CONTINUE HANDLER FOR not_found SET at_end = 1;
  OPEN c1;

  ins_loop: LOOP

    FETCH c1 INTO v_deptno, v_deptname;

    IF at_end = 1 THEN
      LEAVE ins_loop;
    ELSEIF v_dept = 'D11' THEN
      ITERATE ins_loop;
    END  IF;

    INSERT  INTO department (deptno, deptname)
      VALUES ('NEW', v_deptname);

  END LOOP;

  CLOSE c1;
END
```

SQL 过程中的 WHILE 语句: WHILE 语句定义 WHILE 循环开头条件求值为 false 之前执行的一组语句。每次循环迭代之前都会对 WHILE 循环条件（表达式）求值。

以下是带有简单 WHILE 循环的 SQL 过程的示例:

```
CREATE PROCEDURE sum_mn (IN p_start INT
                         ,IN p_end INT
                         ,OUT p_sum INT)
SPECIFIC sum_mn
LANGUAGE SQL
smn: BEGIN

DECLARE v_temp INTEGER DEFAULT 0;
DECLARE v_current INTEGER;

SET v_current = p_start;

WHILE (v_current <= p_end) DO
  SET v_temp = v_temp + v_current;
  SET v_current = v_current + 1;
END WHILE;
p_sum = v_current;
END smn;
```

注意: 可使用算术公式更好地实现此示例中显示的逻辑。此简单示例用于演示语法。

SQL 过程中的 REPEAT 语句: REPEAT 语句定义在 REPEAT 循环结束时条件求值为 true 之前执行的一组语句。REPEAT 循环条件是在每次循环迭代完成时求值的。

对于 WHILE 语句，如果 WHILE 循环条件在第 1 轮求值为 false，那么不会进入循环。REPEAT 语句也可替代使用；但是，不值得将 WHILE 循环逻辑重写为 REPEAT 语句。

以下是包括 REPEAT 语句的 SQL 过程:

```
CREATE PROCEDURE sum_mn2 (IN p_start INT
                         ,IN p_end INT
                         ,OUT p_sum INT)
SPECIFIC sum_mn2
LANGUAGE SQL
smn2: BEGIN

  DECLARE v_temp INTEGER DEFAULT 0;
  DECLARE v_current INTEGER;

  SET v_current = p_start;

  REPEAT
    SET v_temp = v_temp + v_current;
    SET v_current = v_current + 1;
  UNTIL (v_current > p_end)
  END REPEAT;
END
```

SQL 过程中的控制转移语句: 控制转移语句用于在 SQL 过程中重定向控制流。这一无条件转移可用于使控制流从一个点跳至另一个点，另一个点可以在控制语句转移语句之前或之后。SQL 过程中的控制转移语句包括:

- GOTO
- ITERATE
- LEAVE
- RETURN

可在 SQL 过程中的任意位置使用控制转移语句，但 ITERATE 和 LEAVE 通常与 LOOP 语句或其他循环语句一起使用。

SQL 过程中的 GOTO 语句: GOTO 语句是直观基本的控制流语句，会导致控制流发生无条件更改。它用于分流至使用 SQL 过程中定义的标签的特定用户定义位置。

使用 GOTO 语句通常被视为缺乏编程技巧，并且不推荐这样做。大量使用 GOTO 会导致代码可读性不好，特别是在过程变长时。此外，因为有更好的语句可用于控制执行路径，所以 GOTO 并非必需。没有需要使用 GOTO 的特定情况；使用它通常只是为了方便。

以下是包含 GOTO 语句的 SQL 过程的示例:

```
CREATE PROCEDURE adjust_salary ( IN p_empno CHAR(6),
                      IN p_rating INTEGER,
     OUT p_adjusted_salary DECIMAL (8,2) )
LANGUAGE SQL
BEGIN
  DECLARE new_salary DECIMAL (9,2);
  DECLARE service DATE;  -- start date

  SELECT salary, hiredate INTO v_new_salary, v_service
    FROM employee
       WHERE empno = p_empno;

  IF service > (CURRENT DATE - 1 year) THEN
    GOTO exit;
  END IF;

  IF p_rating = 1 THEN
    SET new_salary = new_salary + (new_salary * .10);
```

```
  END IF;

  UPDATE employee SET salary = new_salary WHERE empno = p_empno;

exit:
  SET p_adjusted_salary = v_new_salary;


END
```

此示例演示如何适当地使用 GOTO 语句：跳至接近过程或循环结尾以便不执行某个逻辑，但确保仍然执行某个其他逻辑。

使用 GOTO 语句时，您应该一些额外作用域注意事项:

- 如果在 FOR 语句中定义了 GOTO 语句，那么必须在同一 FOR 语句内定义标签，除非它在嵌套 FOR 语句或嵌套复合语句中。
- 如果在复合语句中定义了 GOTO 语句，那么必须在同一复合语句内定义标签，除非它在嵌套 FOR 语句或嵌套复合语句中。
- 如果在处理程序中定义了 GOTO 语句，那么必须在同一处理程序内定义标签并遵循其他作用域规则。
- 如果在处理程序外定义了 GOTO 语句，那么不能在处理程序内定义标签。
- 如果未在 GOTO 语句可访问的作用域内定义标签，那么会返回错误 (SQLSTATE 42736)。

SQL 过程中的 ITERATE 语句: ITERATE 语句用于使控制流返回至已标记 LOOP 语句的开头。

以下是包含 ITERATE 语句的 SQL 过程的示例:

```
CREATE  PROCEDURE ITERATOR()
LANGUAGE SQL
BEGIN
  DECLARE v_deptno CHAR(3); DECLARE v_deptname VARCHAR(29);
  DECLARE at_end INTEGER DEFAULT 0;
  DECLARE not_found CONDITION FOR SQLSTATE '02000';

  DECLARE c1 CURSOR FOR SELECT deptno, deptname
                        FROM department ORDER BY deptno;
  DECLARE CONTINUE HANDLER FOR not_found SET at_end = 1;
  OPEN c1;

  ins_loop: LOOP
   FETCH c1 INTO v_deptno, v_deptname;
      IF at_end = 1 THEN
        LEAVE ins_loop;
      ELSEIF v_dept = 'D11' THEN
        ITERATE ins_loop;
      END  IF;

      INSERT  INTO department (deptno, deptname)
      VALUES ('NEW', v_deptname);

  END LOOP;

  CLOSE c1;

END
```

在此示例中，ITERATE 语句用于在已访存行中的列值与特定值相匹配时将控制流返回至使用标签 ins_loop 定义的 LOOP 语句。ITERATE 语句的位置确保不会将任何值插入到 department 表中。

SQL 过程中的 LEAVE 语句: LEAVE 语句用于将控制流转移出循环或复合语句。

以下是包含 LEAVE 语句的 SQL 过程的示例:

```
  CREATE  PROCEDURE ITERATOR()
  LANGUAGE SQL
  BEGIN
    DECLARE v_deptno CHAR(3); DECLARE v_deptname VARCHAR(29);
    DECLARE at_end INTEGER DEFAULT 0;
    DECLARE not_found CONDITION FOR SQLSTATE '02000';

    DECLARE c1 CURSOR FOR SELECT deptno, deptname
                          FROM department ORDER BY deptno;
    DECLARE CONTINUE HANDLER FOR not_found SET at_end = 1;

    OPEN c1;

    ins_loop: LOOP

      FETCH c1 INTO v_deptno, v_deptname;

      IF at_end = 1 THEN
        LEAVE ins_loop;
      ELSEIF v_dept = 'D11' THEN
        ITERATE ins_loop;
      END IF;

      INSERT  INTO department (deptno, deptname)
      VALUES ('NEW', v_deptname);

    END LOOP;

    CLOSE c1;
END
```

在此示例中，LEAVE 语句用于退出使用标签 ins_loop 定义的 LOOP 语句。它嵌套在 IF 语句中，并且因此可在 IF 条件为 true 时有条件执行（游标中未发现其他行时，IF 条件变为 true）。LEAVE 语句的位置确保一旦出现 NOT FOUND 错误，不会再执行循环迭代。

SQL 过程中的 RETURN 语句: RETURN 语句用于通过将控制流返回至存储过程的调用程序来无条件立即终止 SQL 过程。

执行 RETURN 语句时必须返回整数值。如果未提供返回值，那么缺省值为 0。该值通常用于指示过程执行成功或失败。该值可以是字面值、变量或求值为整数的表达式。

可在存储过程中使用一个或多个 RETURN 语句。可在 SQL 过程体中的声明块后的任何位置使用 RETURN 语句。

要返回多个输出值，可改为使用参数。必须在执行 RETURN 语句之前设置参数值。

以下是使用 RETURN 语句的 SQL 过程的示例:

```
  CREATE PROCEDURE return_test (IN p_empno CHAR(6),
                                IN p_emplastname VARCHAR(15) )
  LANGUAGE SQL
  SPECIFIC return_test
  BEGIN
```

```
  DECLARE v_lastname VARCHAR (15);

  SELECT lastname INTO v_lastname
    FROM employee
  WHERE empno = p_empno;

  IF v_lastname = p_emplastname THEN
    RETURN 1;
  ELSE
    RETURN -1;
  END IF;

END rt
```

在示例中，如果参数 *p_emplastname* 与存储在表 employee 中的值相匹配，那么该过程返回 1。如果不匹配，那么该过程返回 -1。

SQL 过程中的条件处理程序: 条件处理程序确定条件发生时 SQL 过程的行为。可在 SQL 过程中针对一般条件、指定条件或特定 SQLSTATE 值声明一个或多个条件处理程序。

如果 SQL 过程中的语句出现 SQLWARNING 或 NOT FOUND 条件，并且您已对相应条件声明处理程序，那么 DB2 会将控制权转交给相应处理程序。如果未对此类条件声明处理程序，那么 DB2 会将控制权转义给 SQL 过程体中的下一个语句。如果已声明 SQLCODE 和 SQLSTATE 变量，那么它们将包含对应该条件的值。

如果 SQL 过程中的语句出现 SQLEXCEPTION 条件，并且您已对特定 SQLSTATE 或 SQLEXCEPTION 条件声明处理程序，那么 DB2 会将控制权转交给该处理程序。如果已声明 SQLSTATE 和 SQLCODE 变量，那么成功执行处理程序后它们的值将分别为“00000”和 0。

如果 SQL 过程中的语句出现 SQLEXCEPTION 条件，并且您未对特定 SQLSTATE 或 SQLEXCEPTION 条件声明处理程序，那么 DB2 会终止 SQL 过程并返回至调用程序。

条件处理程序的处理程序声明语法是在复合 SQL（过程）语句中描述的。

从 SQL 过程中返回结果集

在 SQL 过程中，可使用游标完成的工作超过通过结果集的行进行迭代完成的工作。还可使用游标将结果集返回至调用程序。

可通过 SQL 过程（存在嵌套过程调用时）或 C 语言版本的客户机应用程序（通过 CLI 应用程序编程接口、Java、CLI 或 .NET CLR 语言编写）来检索结果集。

开始之前

您必须具有创建 SQL 过程的权限。

过程

要从 SQL 过程中返回结果集，请执行下列操作:

1. 在 CREATE PROCEDURE 语句中指定 DYNAMIC RESULT SETS 子句
2. 使用 WITH RETURN 子句声明游标
3. 在 SQL 过程中打开游标

4. 使游标保持对客户机应用程序打开: 不要关闭游标

示例

以下是仅返回单个结果集的 SQL 过程的示例:

```
CREATE PROCEDURE  read_emp()
SPECIFIC read_emp
LANGUAGE SQL
DYNAMIC RESULT SETS 1

Re:  BEGIN

  DECLARE c_emp CURSOR WITH RETURN FOR
    SELECT salary, bonus, comm.
    FROM employee
    WHERE job != 'PRES';

  OPEN c_emp;

END Re
```

如果 SQL 过程返回之前使用 CLOSE 语句关闭了游标，那么游标结果集将不会返回至调用程序或客户机应用程序。

可使用多个游标从 SQL 过程返回多个结果集。要返回多个游标，必须完成下列操作:

- 在 CREATE PROCEDURE 语句中指定 DYNAMIC RESULT SETS 子句。指定可能返回的最大可能结果集数。实际返回的结果集数不能超过此数。
- 对通过指定 WITH RETURN 子句返回的每个结果集声明游标。
- 打开要返回的游标。
- 使游标保持对客户机应用程序打开: 不要关闭游标。

对于要返回的每个结果集，都需要一个游标。

结果集将按它们的打开顺序返回给调用程序。

创建返回结果集的 SQL 过程后，您可能想要调用它并检索结果集。

还可通过启用同一游标的多个实例来返回多个结果集。您必须使用 WITH RETURN TO CLIENT 来声明游标。

使用 WITH RETURN TO CLIENT 启用打开游标的多个实例的示例如下:

```
CREATE PROCEDURE PROC(IN a INT)
BEGIN
  DECLARE index INTEGER DEFAULT 1;
  WHILE index < a DO
    BEGIN
      DECLARE cur CURSOR WITH RETURN TO CLIENT FOR SELECT * FROM T WHERE pk = index;
      OPEN cur;
      SET index = index + 1;
    END;
  END WHILE;
END
@
```

在 SQL 例程中接收过程结果集

可从在 SQL 体例程中调用的过程接收结果集。

开始之前

必须知道被调用过程将返回的结果集数目。对于调用例程接收的每个结果集，都必须声明结果集。

过程

要接受 SQL 体例程中的过程结果集，请执行下列操作:

1. 对过程将返回的每个结果集声明结果集定位器。例如:

```
DECLARE result1 RESULT_SET_LOCATOR VARYING;
DECLARE result2 RESULT_SET_LOCATOR VARYING;
DECLARE result3 RESULT_SET_LOCATOR VARYING;
```

2. 调用该过程。例如:

```
CALL targetProcedure();
```

3. 将以上定义的结果集定位器变量与被调用过程“相关联”。例如:

```
ASSOCIATE RESULT SET LOCATORS(result1, result2, result3)
   WITH PROCEDURE targetProcedure;
```

4. 将被调用过程传递出的结果集游标分配给结果集定位器。例如:

```
ALLOCATE rsCur CURSOR FOR RESULT SET result1;
```

5. 从结果集访存行。例如:

```
FETCH rsCur INTO ...
```

创建 SQL 过程

创建 SQL 过程类似创建任何数据库对象，原因在于它执行 DDL SQL 语句。

通过执行 CREATE PROCEDURE 语句创建 SQL 过程。

关于此任务

可使用图形开发环境工具执行 CREATE PROCEDURE 语句，或通过直接从 DB2 命令行处理器 (CLP)、DB2 命令窗口或另一 DB2 接口执行该语句。

创建 SQL 过程时，可指定预编译程序和绑定程序生成过程包的方式、用于在 DB2 目录视图中设置 SQL 过程定义程序以及设置其他包选项的授权标识。

从命令行创建 SQL 过程

开始之前

- 用户必须具有对 SQL 过程执行 CREATE PROCEDURE 语句所需的特权。
- 执行过程的 SQL 过程体内包括的所有 SQL 语句的特权。
- SQL 过程的 CREATE PROCEDURE 语句中引用的数据库对象必须存在，才能执行该语句。

过程

- 对命令行处理器 (DB2 CLP) 选择缺省终止符以外的备用终止符，以在下一步中要预编译的脚本中使用，缺省终止符为分号（“;”）。

CLP 必须能够区分出现在例程体的 CREATE 语句内的 SQL 语句结尾和 CREATE PROCEDURE 语句本身的结尾。分号字符必须用于在 SQL 例程体中终止 SQL 语句，所选备用终止符应该用于终止 CREATE 语句及可能包含在 CLP 脚本中的任何其他 SQL 语句。

例如，在以下 CREATE PROCEDURE 语句中，“at;”符号（“@”）用作 DB2 CLP 脚本 myCLPscript.db2 的终止符:

```
CREATE PROCEDURE UPDATE_SALARY_IF
(IN employee_number CHAR(6), IN rating SMALLINT)
LANGUAGE SQL
BEGIN
  DECLARE not_found CONDITION FOR SQLSTATE '02000';
  DECLARE EXIT HANDLER FOR not_found
     SIGNAL SQLSTATE '20000' SET MESSAGE_TEXT = 'Employee not found';

  IF (rating = 1)
    THEN UPDATE employee
      SET salary = salary * 1.10, bonus = 1000
      WHERE empno = employee_number;
  ELSEIF (rating = 2)
    THEN UPDATE employee
      SET salary = salary * 1.05, bonus = 500
      WHERE empno = employee_number;
  ELSE UPDATE employee
      SET salary = salary * 1.03, bonus = 0
      WHERE empno = employee_number;
  END IF;
END
@
```

- 使用以下 CLP 命令从命令行运行包含 CREATE PROCEDURE 语句的 DB2 CLP 脚本:

  ```
  db2 -td terminating-character -vf CLP-script-name
  ```

  其中 *终止符* 是在要运行的 CLP 脚本文件 *CLP-脚本名* 中使用的终止符。

  DB2 CLP 选项 **-td** 指示将使用*终止符*重置 CLP 终止符缺省值。**-vf** 指示将使用 CLP 的可选冗余（**-v**）选项，这会导致脚本中的每个 SQL 语句或命令运行时显示到屏幕上，同时显示执行这些语句或命令所生成的输出。**-f** 选项指示命令的目标为文件。

  要运行第一步中显示的特定脚本，请从系统命令提示符发出以下命令:

  ```
  db2 -td@ -vf myCLPscript.db2
  ```

为编译型 SQL 对象定制预编译和绑定选项

SQL 过程、编译型函数、编译型触发器及复合 SQL（编译型）语句的预编译和绑定选项可通过 DB2 注册表变量或一些 SQL 过程例程定制。

关于此任务

要为编译型 SQL 对象定制预编译和绑定选项，请设置实例范围的 DB2 注册表变量 **DB2_SQLROUTINE_PREPOPTS**。例如:

```
db2set DB2_SQLROUTINE_PREPOPTS=options
```

可使用 SET_ROUTINE_OPTS 存储过程在过程级别更改这些选项。在当前会话中为创建 SQL 过程而设置的选项的值可通过 GET_ROUTINE_OPTS 函数获取。

用于编译给定例程的选项存储在系统目录表 ROUTINES.PRECOMPILE_OPTIONS 内对应该例程的行中。如果该例程重新生效，那么还会在例程重新生效期间使用这些已存储选项。

创建例程后，可使用 SYSPROC.ALTER_ROUTINE_PACKAGE 和 SYSPROC.REBIND_ROUTINE_PACKAGE 过程来改变编译选项。改变后的选项反映在 ROUTINES_PRECOMPILE_OPTIONS 系统目录表中。

注: 在 SQL 过程中对 FETCH 语句中引用的游标和 FOR 语句中的隐式游标禁用了游标分块。不管对 BLOCKING 绑定选项指定的值如何，将以优化高效方式检索数据（一次一行）。

示例

在此示例中使用的 SQL 过程将在以下 CLP 脚本中定义。这些脚本不在 sqlpl 样本目录中，但可通过将 CREATE 过程语句复制并粘贴到您自己的文件中来轻松创建这些文件。

这些样本使用表 expenses，可按如下所示在样本数据库中创建该表:

```
db2 connect to sample
db2 CREATE TABLE expenses(amount DOUBLE, date DATE)
db2 connect reset
```

开始时指定使用日期的 ISO 格式作为实例范围设置:

```
db2set DB2_SQLROUTINE_PREPOPTS="DATETIME ISO"
db2stop
db2start
```

必须停止然后重新启动 DB2 实例才能使更改生效。

然后连接至数据库:

```
db2 connect to sample
```

按如下所示在 CLP 脚本 maxamount.db2 中定义第一个过程:

```
CREATE PROCEDURE maxamount(OUT maxamnt DOUBLE)
BEGIN
  SELECT max(amount) INTO maxamnt FROM expenses;
END @
```

它将使用选项 **DATETIME ISO** 和 **ISOLATION UR** 创建:

```
db2 "CALL SET_ROUTINE_OPTS(GET_ROUTINE_OPTS() || ' ISOLATION UR')"
db2 -td@ -vf maxamount.db2
```

按如下所示在 CLP 脚本 fullamount.db2 中定义下一个过程:

```
CREATE PROCEDURE fullamount(OUT fullamnt DOUBLE)
BEGIN
  SELECT sum(amount) INTO fullamnt FROM expenses;
END @
```

它将使用选项 **ISOLATION CS** 创建（请注意，在此情况下，未使用实例范围的 **DATETIME ISO** 设置）:

```
CALL SET_ROUTINE_OPTS('ISOLATION CS')
db2 -td@ -vf fullamount.db2
```

按如下方式在 CLP 脚本 perday.db2 中定义示例中的最后一个过程:

```
CREATE PROCEDURE perday()
BEGIN
  DECLARE cur1 CURSOR WITH RETURN FOR
    SELECT date, sum(amount)
    FROM expenses
    GROUP BY date;

    OPEN cur1;
END @
```

最后一个 SET_ROUTINE_OPTS 调用使用 NULL 值作为自变量。这会复原 **DB2_SQLROUTINE_PREPOPTS** 注册表中指定的全局设置，所以最后一个过程将使用选项 **DATETIME ISO** 创建:

```
CALL SET_ROUTINE_OPTS(NULL)
db2 -td@ -vf perday.db2
```

改进 SQL 过程的性能

有关DB2 如何编译 SQL PL 和内联 SQL PL 的概述

在讨论如何改进 SQL 过程的性能之前，应先讨论 DB2 如何在执行 CREATE PROCEDURE 语句时编译这些过程。

创建 SQL 过程后，DB2 会将过程体中的 SQL 查询与过程逻辑分开。为使性能达到最佳，SQL 查询以静态方式编译为包中的若干段。对于静态编译的查询，一个段主要由 DB2 优化器为该查询选择的存取方案组成。包是若干段的集合。有关包和段的更多信息，请参阅《DB2 SQL 参考》。过程逻辑编译为动态链接的库。

执行过程期间，每次控制权从过程逻辑流至 SQL 语句时，DLL 与 DB2 引擎之间存在“上下文切换”。从 DB2 版本 8.1 开始，SQL 过程以“不受防护方式”运行。即，它们与 DB2 引擎在同一地址空间中运行。因此，我们在此处提到的上下文切换并非操作系统级别的完整上下文切换，而是 DB2 内的层切换。减少频繁调用的过程（如 OLTP 应用程序中的过程）或处理大量行的过程（如执行数据清理的过程）中的上下文切换数目对它们的性能有很大影响。

包含 SQL PL 的 SQL 过程是通过以静态方式将其 SQL 查询编译成包中的段来实现的，而内联 SQL PL 函数是通过将函数体直接插入到使用它的查询中来实现的。SQL 函数中的查询是一起编译的，就像函数体是单个查询一样。每次编译使用了该函数的语句时，都会进行此编译。与 SQL 过程中进行的操作不同，SQL 函数中的过程语句与数据流语句在同一层中执行。因此，每次控制权从过程流至数据流语句（或相反）时，不会进行上下文切换。

如果对逻辑没有副作用，请改为使用 SQL 函数

因为过程中的 SQL PL 与函数中的内联 SQL PL 在编译上存在差别，所以在仅查询 SQL 数据而不修改数据（即，对数据库内部或外部的数据没有副作用）时，假定过程代码块在函数中执行的速度比在过程中执行的速度快是合理的。

仅当 SQL 函数支持您需要执行的所有语句时，才能这样做。SQL 函数不能包含修改数据库的 SQL 语句。而且，只有一部分 SQL PL 在函数的内联 SQL PL 中可用。例如，不能在 SQL 函数中执行 CALL 语句、声明游标或返回结果集。

以下是 SQL 过程的示例，其中包含可用于转换至 SQL 函数以使性能达到最佳的 SQL PL:

```
CREATE PROCEDURE GetPrice (IN Vendor CHAR&(20&),
                           IN Pid INT, OUT price DECIMAL(10,3))
LANGUAGE SQL
BEGIN
  IF Vendor eq; ssq;Vendor 1ssq;
    THEN SET price eq; (SELECT ProdPrice
                        FROM V1Table
                        WHERE Id = Pid);
  ELSE IF Vendor eq; ssq;Vendor 2ssq;
    THEN SET price eq; (SELECT Price FROM V2Table
                        WHERE Pid eq; GetPrice.Pid);
  END IF;
END
```

以下是重写的 SQL 函数:

```
CREATE FUNCTION GetPrice (Vendor CHAR(20), Pid INT)
RETURNS  DECIMAL(10,3)
LANGUAGE SQL
BEGIN
  DECLARE price DECIMAL(10,3);
  IF Vendor = 'Vendor 1'
    THEN SET price = (SELECT ProdPrice
                      FROM V1Table
                      WHERE Id = Pid);
  ELSE IF Vendor = 'Vendor 2'
    THEN SET price = (SELECT Price FROM V2Table
                WHERE Pid = GetPrice.Pid);
  END IF;
  RETURN price;
END
```

请记住，调用函数与调用过程不同。要调用函数，请使用 VALUES 语句，或在表达式有效的位置（例如，在 SELECT 或 SET 语句中）调用函数。下列任何一项都是调用新函数的有效方式:

```
VALUES (GetPrice('IBM', 324))

SELECT VName FROM Vendors WHERE GetPrice(Vname, Pid) < 10

SET price = GetPrice(Vname, Pid)
```

如果 SQL PL 过程中只需要使用一个语句，那么应尽量避免在其中使用多个语句。

尽管通常从理念上讲应该编写简洁的 SQL，但实际操作时很容易忘记这一点。例如以下 SQL 语句:

```
INSERT INTO tab_comp VALUES (item1, price1, qty1);
INSERT INTO tab_comp VALUES (item2, price2, qty2);
INSERT INTO tab_comp VALUES (item3, price3, qty3);
```

可重写为单个语句:

```
INSERT INTO tab_comp VALUES (item1, price1, qty1),
                            (item2, price2, qty2),
                            (item3, price3, qty3);
```

多行插入所需时间大概是执行三个原始语句所需时间的 1/3。孤立地看，此改进可能显得微不足道，但如果代码片段重复执行（例如在循环或触发器体中），那么此改进会很重要。

同样，类似如下的 SET 语句序列:

```
SET A = expr1;
SET B = expr2;
SET C = expr3;
```

可写为单个 VALUES 语句:

```
VALUES expr1, expr2, expr3 INTO A, B, C;
```

此变换保留了原始序列的语义（如果任意两个语句之间没有依赖性）。为说明这一点，请考虑:

```
SET A = monthly_avg * 12;
SET B = (A / 2) * correction_factor;
```

将以上两个语句转换为:

```
VALUES (monthly_avg * 12, (A / 2) * correction_factor) INTO A, B;
```

不会保留原始语义，原因是 INTO 关键字之前的表达式以“并行”方式求值。这意味着指定给 *B* 的值并非以指定给 *A* 的值为基础，这是原始语句的期望语义。

将多个 SQL 语句减少为单个 SQL 表达式

与其他编程语言一样，SQL 语言提供两种类型的条件构造: 过程（IF 和 CASE 语句）及函数（CASE 表达式）。在大多数环境下，每种类型都可用于表达计算，使用哪种类型由您的喜好而定。但是，与使用 CASE 或 IF 语句编辑的逻辑相比，使用 CASE 表达式编写的逻辑更加紧凑和高效。

考虑以下 SQL PL 代码片段:

```
IF (Price <= MaxPrice) THEN
  INSERT INTO tab_comp(Id, Val) VALUES(Oid, Price);
ELSE
  INSERT INTO tab_comp(Id, Val) VALUES(Oid, MaxPrice);
END IF;
```

IF 子句中的条件仅用于决定将哪个值插入到 tab_comp.Val 列中。为避免在过程与数据流层之间进行上下文切换，可使用 CASE 表达式将相同逻辑表达为单个 INSERT 语句:

```
INSERT INTO tab_comp(Id, Val)
       VALUES(Oid,
            CASE
               WHEN (Price <= MaxPrice) THEN Price
               ELSE MaxPrice
            END);
```

值得注意的是，可在期望标量值的任何上下文使用 CASE 表达式。特别是可在赋值语句的右端使用 CASE 表达式。例如:

```
IF (Name IS NOT NULL) THEN
  SET ProdName = Name;
ELSEIF (NameStr IS NOT NULL) THEN
  SET ProdName = NameStr;
ELSE
  SET ProdName = DefaultName;
END IF;
```

可重写为:

```
SET ProdName = (CASE
                  WHEN (Name IS NOT NULL) THEN Name
                  WHEN (NameStr IS NOT NULL) THEN NameStr
                  ELSE  DefaultName
                END);
```

实际上，此特定示例表达了更好的解决方案:

```
SET ProdName = COALESCE(Name, NameStr, DefaultName);
```

不要低估花时间分析并考虑重写 SQL 带来的好处。性能提高带给您的好处大大超过您分析并重写过程所花时间产生的成本。

使用 SQL 的一次性设置语义

过程构造（如循环、赋值和游标）允许我们表达仅使用 SQL DML 语句无法表达的计算。但是，如果有可任意支配的过程语句，那么即使手边的计算实际上可仅使用 SQL DML 语句表达时，也会为它们带来风险。就像先前提到的那样，过程计算的性能比使用 DML 语句表达的等价计算的性能快几个数量级。考虑以下代码片段:

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab_comp;
OPEN cur1;
FETCH cur1 INTO v1, v2;
WHILE SQLCODE <> 100 DO
  IF (v1 > 20) THEN
    INSERT INTO tab_sel VALUES (20, v2);
  ELSE
    INSERT INTO tab_sel VALUES (v1, v2);
  END IF;
  FETCH cur1 INTO v1, v2;
END WHILE;
```

开始时可通过应用上一节“将多个 SQL 语句减少为单个 SQL 表达式”中讨论的变换来改进循环体:

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab_comp;
OPEN cur1;
FETCH cur1 INTO v1, v2;
WHILE SQLCODE <> 100 DO
  INSERT INTO tab_sel VALUES (CASE
                                WHEN v1 > 20 THEN 20
                                ELSE v1
                              END, v2);
  FETCH cur1 INTO v1, v2;
END WHILE;
```

但通过更仔细地检查，可将整个代码块编写为带有子 SELECT 的 INSERT:

```
INSERT INTO tab_sel (SELECT (CASE
                               WHEN col1 > 20 THEN 20
                               ELSE col1
                             END),
                             col2
                     FROM tab_comp);
```

在最初创建时，对于 SELECT 语句中的每行，过程与数据流层之间存在上下文切换。在最后一次创建时，根本没有上下文切换，并且优化器可全局优化完整计算。

另一方面，如果每个 INSERT 语句都以不同表为目标，那么这一显著简化不可行，如以下示例所示:

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab_comp;
OPEN cur1;
FETCH cur1 INTO v1, v2;
WHILE SQLCODE <> 100 DO
  IF (v1 > 20) THEN
    INSERT INTO tab_default VALUES (20, v2);
  ELSE
    INSERT INTO tab_sel VALUES (v1, v2);
  END IF;
  FETCH cur1 INTO v1, v2;
END WHILE;
```

但是，此处还可使用 SQL 的一次设置特点:

```
INSERT INTO tab_sel (SELECT col1, col2
                     FROM tab_comp
                     WHERE col1 <= 20);
INSERT INTO tab_default (SELECT col1, col2
                         FROM tab_comp
                         WHERE col1 > 20);
```

查看并改进现有过程逻辑的性能时，消除游标循环所花的时间会有所回报。

及时通知 DB2 优化器

创建过程时，其 SQL 查询编译为包中的若干段。除了别的以外，DB2 优化器还会根据表统计信息（例如，表大小或列中数据值的相对频率）以及编译查询时可用的索引来为查询选择执行方案。表进行显著更改时，最好由 DB2 再次收集这些表的统计信息。更新统计信息或创建新索引时，最好重新绑定与使用表的 SQL 过程相关联的包，以允许 DB2 创建使用最新统计信息和索引的方案。

可使用 **RUNSTATS** 命令来更新表统计信息。要重新绑定与 SQL 过程相关联的包，可使用 DB2 版本 8.1 中提供的 REBIND_ROUTINE_PACKAGE 内置过程。例如，可使用以下命令来重新绑定过程 MYSCHEMA.MYPROC 的包:

```
CALL SYSPROC.REBIND_ROUTINE_PACKAGE('P', 'MYSCHEMA.MYPROC', 'ANY')
```

其中“P”指示该包对应于过程，“ANY”指示会考虑对 SQL 路径中的所有函数和类型进行函数和类型解析。请参阅 **REBIND** 命令的命令参考条目来了解更多详细信息。

使用数组

可使用数组在应用程序与存储过程之间高效传递数据集合，以及在 SQL 过程中存储和操作瞬态数据集合而不必使用关系表。针对 SQL 过程中可用的数组的操作程序允许高效存储和检索数据。用于创建中等大小数组的应用程序性能比创建大型数组（规模为几兆字节）的应用程序性能好很多，原因是整个数组存储在主存储器中。请参阅*相关链接*部分以了解其他信息。

SQL 函数

SQL 函数是完全通过 SQL 实现的函数，可用于封装可以像编程子例程一样调用的逻辑。可创建 SQL 标量函数和 SQL 表函数。

在数据库或数据库应用程序体系结构中有许多有用的应用程序可用于 SQL 函数。SQL 函数可用于对列数据创建操作程序、扩展内置函数的支持、使应用程序逻辑更加模块化，并可用于改进整体数据库设计和数据库安全性。

以下主题提供有关 SQL 函数的更多信息:

SQL 函数的功能

SQL 函数具有许多常用功能:

SQL 函数:

- 可包含 SQL 过程语言语句及功能，它们支持在传统静态和动态 SQL 语句周围实现控制流逻辑。
- 在整个 DB2 系列品牌的数据库产品中受支持，在这些产品中，在 DB2 版本 9 中受支持的大部分功能也受支持。
- 易于实现，原因是它们使用简单高级强类型语言。
- SQL 函数比等价的外部函数更可靠。
- 支持输入参数。
- SQL 标量函数返回标量值。
- SQL 表函数返回表结果集。
- 支持简单但功能强大的条件和错误处理模型。
- 允许您轻松地将 SQLSTATE 和 SQLCODE 值作为特殊变量访问。
- 驻留在数据库中，并且作为备份和复原操作的一部分自动备份和复原。
- 可在 SQL 语句中支持使用表达式的任何位置调用。
- 运行对其他 SQL 函数或使用其他语言实现的函数进行嵌套函数调用。
- 支持递归（在编译型函数中使用动态 SQL 时）。
- 可通过触发器调用。
- 许多 SQL 语句可包括在 SQL 函数中，但它们例外。有关可在 SQL 函数中包括并执行的 SQL 语句的完整列表，请参阅: 可在例程中执行的 SQL 语句

SQL 函数提供广泛支持（不限于上面列示的各项）。根据最佳范例实现时，它们可充当数据库体系结构、数据库应用程序设计和数据库系统性能中的重要角色。

设计 SQL 函数

设计 SQL 函数是在数据库中创建 SQL 函数之前执行的任务。

要设计 SQL 函数，应熟悉 SQL 函数的功能。以下主题提供有关 SQL 函数设计概念的更多信息:

内联型 SQL 函数和编译型 SQL 函数

SQL 函数有两种实现类型: 内联型 SQL 函数和编译型 SQL 函数。

内联型 SQL 函数

内联型 SQL 函数是使用 CREATE FUNCTION 语句创建的 SQL 函数，其主体为 RETURN 语句或内联复合语句。内联复合语句是使用 BEGIN ATOMIC 和 END 关键字定义的。

内联型 SQL 函数可包含 SQL 语句和内联 SQL PL 语句: SQL PL 语句的子集。

编译型 SQL 函数

编译型 SQL 函数是使用 CREATE FUNCTION 语句创建的 SQL 函数，其主体为 RETURN 语句或编译型复合语句。编译型复合语句是使用 BEGIN 和 END 关键字定义的。

省略 ATOMIC 子句时，SQL 函数会被编译，因此可包括或引用的 SQL PL 功能比内联型 SQL 函数多。编译型 SQL 函数可包括下列在内联型 SQL 函数中不受支持的功能:

- SQL PL 语句，包括:
  - CASE 语句
  - REPEAT 语句
- 游标处理
- 动态 SQL
- 条件处理程序

SQL 函数限制

创建 SQL 函数之前或诊断与 SQL 函数的实现和使用有关的问题时，应注意 SQL 函数的限制。

SQL 函数存在以下限制:

- SQL 表函数不能包含编译型复合语句。
- 不能在分区数据库环境中调用包含编译型复合语句的 SQL 标量函数。
- 根据定义，SQL 函数不能包含使用 WITH RETURN 子句定义的游标。
- 不能在分区数据库环境中调用编译型 SQL 标量函数。
- 以下数据类型在编译型 SQL 函数中不受支持: 结构化数据类型、XML 数据类型、LONG VARCHAR 数据类型和 LONG VARGRAPHIC 数据类型。
- 在此版本中，不支持在编译型 SQL 函数中使用 DECLARE TYPE 语句。

创建 SQL 标量函数

设计数据库或开发应用程序时应创建 SQL 标量函数。封装一块可重复使用的逻辑以便可在多个应用程序的 SQL 语句或数据库对象中引用非常有用时，通常会创建 SOL 标量函数。

开始之前

创建 SQL 函数之前:

- 读取: 第 91 页的『SQL 函数』
- 读取: 第 92 页的『SQL 函数的功能』
- 确保您具有执行 CREATE FUNCTION (scalar) 语句的特权。

关于此任务

限制

请参阅: 『SQL 函数限制』

过程

1. 定义 CREATE FUNCTION (scalar) 语句:
   a. 指定函数的名称。
   b. 指定每个输入参数的名称和数据类型。
   c. 指定 RETURNS 关键字和标量返回值的数据类型。
   d. 指定 BEGIN 关键字以引入函数体。注意: 不建议对新函数使用 BEGIN ATOMIC 关键字。
   e. 指定函数体。指定 RETURN 子句和标量返回值或变量。
   f. 指定 END 关键字。
2. 从受支持接口执行 CREATE FUNCTION (scalar) 语句。

结果

CREATE FUNCTION (scalar) 语句应成功执行，并且应创建标量函数。

示例

示例 1

以下是编译型 SQL 函数的示例:

```
CREATE FUNCTION GetPrice (Vendor CHAR(20), Pid INT)
  RETURNS  DECIMAL(10,3)
  LANGUAGE SQL
  MODIFIES SQL
  BEGIN
    DECLARE price DECIMAL(10,3);

    IF Vendor = 'Vendor 1'
      THEN SET price = (SELECT ProdPrice FROM V1Table WHERE Id = Pid);
    ELSE IF Vendor = 'Vendor 2'
      THEN SET price = (SELECT Price
                        FROM V2Table
           WHERE Pid = GetPrice.Pid);
    END IF;

  RETURN price;
END
```

此函数接收两个输入参数值，并根据输入参数值有条件返回单个标量值。它需要声明并使用局部变量 price 来保存要返回的值直到函数返回。

示例 2

以下示例演示包含游标、条件处理程序语句和 REPEAT 语句的编译型 SQL 函数定义:

```
CREATE FUNCTION exit_func(a INTEGER)
  SPECIFIC exit_func
  LANGUAGE SQL
  RETURNS INTEGER
  BEGIN
    DECLARE val INTEGER DEFAULT 0;

    DECLARE myint INTEGER DEFAULT 0;

    DECLARE cur2 CURSOR FOR
      SELECT c2 FROM udfd1
        WHERE c1 <= a
        ORDER BY c1;
```

```
  DECLARE EXIT HANDLER FOR NOT FOUND
    BEGIN
      SIGNAL SQLSTATE '70001'
      SET MESSAGE_TEXT =
        'Exit handler for not found fired';
    END;

  OPEN cur2;

  REPEAT
    FETCH cur2 INTO val;
    SET myint = myint + val;
  UNTIL (myint >= a)
  END REPEAT;

  CLOSE cur2;

  RETURN myint;

END@
```

下一步做什么

创建标量函数后，您可能想要调用该函数以进行测试。

创建 SQL 表函数

可随时创建 SQL 表函数。

开始之前

创建 SQL 表函数之前，请确保您具有执行 CREATE FUNCTION（表）语句所需的特权。

限制

请参阅: 第 93 页的『SQL 函数限制』

过程

1. 定义 CREATE FUNCTION (table) 语句:
   a. 指定函数的名称。
   b. 指定每个输入参数的名称和数据类型。
   c. 指定例程属性。
   d. 指定 RETURNS TABLE 关键字。
   e. 指定用于引入函数体的 BEGIN ATOMIC 关键字。
   f. 指定函数体。
   g. 指定 RETURN 子句，其中指定用于定义要返回的结果集的查询并用方括号括起来。
   h. 指定 END 关键字。
2. 从受支持接口执行 CREATE FUNCTION (table) 语句。

结果

CREATE FUNCTION (table) 语句应成功执行并且应创建表函数。

示例

示例 1

以下是用于记录和审计对职员薪水数据所做更新的编译型 SQL 表函数的示例:

```
CREATE FUNCTION update_salary  (updEmpNum CHAR(4), amount INTEGER)
RETURNS TABLE (emp_lastname VARCHAR(10),
      emp_firstname VARCHAR(10),
      newSalary INTEGER)
LANGUAGE SQL
MODIFIES SQL DATA
NO EXTERNAL ACTION
NOT DETERMINISTIC
BEGIN ATOMIC

  INSERT INTO audit_table(user, table, action, time)
    VALUES (USER, 'EMPLOYEE',
    'Salary update. Values: ' || updEmpNum || ' ' || char(amount), CURRENT_TIMESTAMP);

  RETURN (SELECT lastname, firstname, salary
    FROM FINAL TABLE(UPDATE employee SET salary = salary + amount WHERE employee.empnum = updEmpNum));

END
```

此函数会更新 updEmpNum 所指定职员的薪水（更新金额由 amount 指定），并且也会在审计表 audit_table 中记录调用例程的用户、所修改表的名称以及用户所作修改的类型。在 FROM 子句中引用数据更改语句的 SELECT 语句用来返回所更新的行值。

示例 2

以下是编译型 SQL 表函数的示例:

```
CREATE TABLE t1(pk INT

CREATE TABLE t1_archive LIKE T1%

CREATE FUNCTION archive_tbl_t1(ppk INT)
  RETURNS TABLE(pk INT, c1 INT, date)
LANGUAGE SQL
MODIFIES SQL DATA
BEGIN ATOMIC

  DECLARE c1 INT;

  DECLARE date DATE;

  SET (c1, date) = (SELECT * FROM OLD TABLE(DELETE FROM t1 WHERE t1.pk = ppk));

  INSERT INTO T1_ARCHIVE VALUES (ppk, c1, date);

  RETURN VALUES (pk, c1, date);
END%
```

下一步做什么

创建表函数后，您可能想要调用该函数以进行测试。

第 6 章 复合语句

复合语句将其他语句组合成可执行块。复合语句可独立执行，也可包括在过程、函数、方法和触发器之类的数据库对象的定义中。因为存在不同差别和限制，所以它们的 SQL 语句不同。

复合语句可以是内联复合语句（以前称为动态复合语句）或编译型复合语句。以下各段显示了两种语句之间的差别。

内联复合语句

内联复合语句是原子的，并且使用 BEGIN ATOMIC 和 END 关键字定义，可在这两个关键字间定义和执行其他 SQL 语句。内联复合语句可包含变量声明、SQL 语句以及识别为内联 SQL PL 语句的 SQL PL 语句子集。

编译型复合语句

编译型复合语句不是原子的，并且使用 BEGIN 和 END 关键字定义，可在这两个关键字间定义和执行其他 SQL 语句。编译型复合语句可包含 SQL 语句和所有 SQL PL 语句。

如果要使用随编译型复合语句提供的其他功能，那么应选择使用编译型复合语句而不是内联复合语句。

复合语句主要用于创建可从 DB2 命令行处理器执行的简短脚本。还可使用它们来定义例程或触发器的主体。

创建复合语句

需要运行由 SQL 语句组成的脚本时，应创建并执行复合语句。

开始之前

创建复合语句之前:

- 读取: 第 6 章，『复合语句』
- 确保您具有执行复合语句所需的特权。

过程

1. 定义复合 SQL 语句。
2. 从受支持接口执行复合 SQL 语句。

结果

如果以动态方式执行，那么该 SQL 语句应成功执行。

示例

以下是包含 SQL PL 的内联型复合 SQL 语句的示例:

```
BEGIN
 FOR row AS
   SELECT pk, c1, discretize(c1) AS v FROM source
 DO
```

```
    IF row.v is NULL THEN
      INSERT INTO except VALUES(row.pk, row.c1);
    ELSE
      INSERT INTO target VALUES(row.pk, row.d);
    END IF;
  END FOR;
 END
```

复合语句由关键字 BEGIN 和 END 绑定。它包括使用包含在 SQL PL 中的 FOR 和 IF/ELSE 控制语句。FOR 语句用于通过已定义的一组行迭代。对于每行，将检查列值并根据该值有条件地将一组值插入到另一个表中。

第 2 部分 PL/SQL 支持

您可以使用 DB2 接口来编译和执行 PL/SQL（过程语言/结构化查询语言）语句。此支持能够降低启用现有 PL/SQL 解决方案的复杂性，以使这些解决方案能够与 DB2 数据服务器配合工作。

支持的接口包括:

- DB2 命令行处理器 (CLP)
- DB2 CLPPlus
- IBM Data Studio 完整客户机

缺省情况下，这些接口不允许执行 PL/SQL 语句。您必须在 DB2 数据服务器上启用 PL/SQL 语句执行支持。

第 7 章 PL/SQL 功能

您可以使用 DB2 接口来编译和执行 PL/SQL 语句和脚本。

可以执行下列 PL/SQL 语句:

- 匿名块; 例如 DECLARE...BEGIN...END
- CREATE OR REPLACE FUNCTION 语句
- CREATE OR REPLACE PACKAGE 语句
- CREATE OR REPLACE PACKAGE BODY 语句
- CREATE OR REPLACE PROCEDURE 语句
- CREATE OR REPLACE TRIGGER 语句
- DROP PACKAGE 语句
- DROP PACKAGE BODY 语句

可以从其他 PL/SQL 语句或 DB2 SQL PL 语句中调用 PL/SQL 过程和函数。可以使用 CALL 语句从 SQL PL 中调用 PL/SQL 过程。

在 PL/SQL 上下文中，支持下列语句和语言元素:

- 类型声明（在此版本中，只在程序包中支持类型声明。在过程、函数、触发器或匿名块中，不支持类型声明。）
  - 关联数组
  - 记录类型
  - VARRAY 类型
- 变量声明:
  - %ROWTYPE
  - %TYPE
- 基本语句、子句和语句属性:
  - 赋值语句
  - NULL 语句
  - RETURNING INTO 子句
  - 语句属性，包括 SQL%FOUND、SQL%NOTFOUND 和 SQL%ROWCOUNT
- 控制语句和结构:
  - CASE 语句:
    - 简单 CASE 语句
    - 搜索型 CASE 语句
  - 异常处理
  - EXIT 语句
  - FOR 语句
  - GOTO 语句
  - IF 语句

  - LOOP 语句
  - WHILE 语句
- 静态游标:
  - CLOSE 语句
  - 游标 FOR 循环语句
  - FETCH 语句（包括以 %ROWTYPE 变量为目标的 FETCH INTO）
  - OPEN 语句
  - 参数化游标
  - 游标属性
- REF CURSOR 支持:
  - 类型为 REF CURSOR 的变量和参数
  - 强 REF CURSOR
  - OPEN FOR 语句
  - 将 REF CURSOR 返回到 JDBC 应用程序
- 错误支持:
  - RAISE_APPLICATION_ERROR 过程
  - RAISE 语句
  - SQLCODE 函数
  - SQLERRM 函数

第 8 章 根据 CLP 脚本来创建 PL/SQL 过程和函数

可以根据 DB2 命令行处理器 (CLP) 脚本来创建 PL/SQL 过程和函数。

过程

1. 在 CLP 脚本文件中构造 PL/SQL 过程或函数定义。每个语句均以换行符和正斜杠字符（/）结束。 另外，还支持其他语句终止字符。
2. 保存文件。 在本示例中，文件名是 script.db2。
3. 从 CLP 中执行此脚本。 如果已使用正斜杠字符或分号来终止语句，请发出以下命令:

   ```
   db2 -td/ -vf script.db2
   ```

   如果在脚本文件中使用了另一个语句终止字符（例如 @ 字符），那么必须在命令字符串中指定该字符。例如:

   ```
   db2 -td@ -vf script.db2
   ```

结果

如果没有语法错误，那么该 CLP 脚本应该会执行成功。

示例

以下 CLP 脚本示例创建一个 PL/SQL 函数和过程，然后调用该 PL/SQL 过程。

```
CONNECT TO mydb
/

CREATE TABLE emp (
     name            VARCHAR2(10),
     salary          NUMBER,
     comm            NUMBER,
     tot_comp        NUMBER
)
/

INSERT INTO emp VALUES ('Larry', 1000, 50, 0)
/
INSERT INTO emp VALUES ('Curly', 200, 5, 0)
/
INSERT INTO emp VALUES ('Moe', 10000, 1000, 0)
/

CREATE OR REPLACE FUNCTION emp_comp (
     p_sal           NUMBER,
     p_comm          NUMBER )
RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END emp_comp
/

CREATE OR REPLACE PROCEDURE update_comp(p_name IN VARCHAR) AS
BEGIN
    UPDATE emp SET tot_comp = emp_comp(salary, comm)
      WHERE name = p_name;
```

```
END update_comp
/

CALL update_comp('Curly')
/

SELECT * FROM emp
/

CONNECT RESET
/
```

此脚本生成以下样本输出:

```
CALL update_comp('Curly')

  Return Status = 0

SELECT * FROM emp

NAME       SALARY             COMM               TOT_COMP
---------- ------...---------- ----...---------- --------...-------
Larry                    1000                 50                  0
Curly                     200                  5               4920
Moe                     10000               1000                  0

  3 record(s) selected.
```

下一步做什么

通过进行调用来测试新过程或函数。对于过程，请使用 CALL 语句。对于函数，请执行包含对那些函数的引用的查询或其他 SQL 语句。

第 9 章 PL/SQL 支持方面的限制

在执行 PL/SQL 编译之前以及在对 PL/SQL 编译或运行时问题进行故障诊断时，注意 PL/SQL 编译方面的限制至关重要。

在此版本中:

- 无法在分区数据库环境中创建 PL/SQL 触发器。
- 仅可在分区数据库环境的目录分区创建 PL/SQL 过程、函数、触发器和程序包。
- 数据库未被定义为 Unicode 数据库时，不支持在 PL/SQL 语句或 PL/SQL 上下文中使用 NCLOB 数据类型。在 Unicode 数据库中，NCLOB 数据类型将映射到 DB2 DBCLOB 数据类型。
- 不支持 XMLTYPE 数据类型。

第 10 章 PL/SQL 样本模式

大部分 PL/SQL 示例基于一个代表组织机构中的职员的 PL/SQL 样本模式。

以下脚本（plsql_sample.sql）用于定义该 PL/SQL 样本模式。

```
--
--  Script that creates the 'sample' tables, views, procedures,
--  functions, triggers, and so on.
--
--  Create and populate tables used in the documentation examples.
--
--  Create the 'dept' table
--
CREATE TABLE dept (
    deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname           VARCHAR2(14) NOT NULL CONSTRAINT dept_dname_uq UNIQUE,
    loc             VARCHAR2(13)
);
--
--  Create the 'emp' table
--
CREATE TABLE emp (
    empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename           VARCHAR2(10),
    job             VARCHAR2(9),
    mgr             NUMBER(4),
    hiredate        DATE,
    sal             NUMBER(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
    comm            NUMBER(7,2),
    deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk
                        REFERENCES dept(deptno)
);
--
--  Create the 'jobhist' table
--
CREATE TABLE jobhist (
    empno           NUMBER(4) NOT NULL,
    startdate       DATE NOT NULL,
    enddate         DATE,
    job             VARCHAR2(9),
    sal             NUMBER(7,2),
    comm            NUMBER(7,2),
    deptno          NUMBER(2),
    chgdesc         VARCHAR2(80),
    CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate),
    CONSTRAINT jobhist_ref_emp_fk FOREIGN KEY (empno)
        REFERENCES emp(empno) ON DELETE CASCADE,
    CONSTRAINT jobhist_ref_dept_fk FOREIGN KEY (deptno)
        REFERENCES dept (deptno) ON DELETE SET NULL,
    CONSTRAINT jobhist_date_chk CHECK (startdate <= enddate)
);
--
--  Create the 'salesemp' view
--
CREATE OR REPLACE VIEW salesemp AS
    SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job = 'SALESMAN';
--
--  Sequence to generate values for function 'new_empno'
--
CREATE SEQUENCE next_empno START WITH 8000 INCREMENT BY 1;
--
--  Issue PUBLIC grants
```

```
--
GRANT ALL ON emp TO PUBLIC;
GRANT ALL ON dept TO PUBLIC;
GRANT ALL ON jobhist TO PUBLIC;
GRANT ALL ON salesemp TO PUBLIC;
--
--  Load the 'dept' table
--
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');
INSERT INTO dept VALUES (30,'SALES','CHICAGO');
INSERT INTO dept VALUES (40,'OPERATIONS','BOSTON');
--
--  Load the 'emp' table
--
INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'17-DEC-80',800,NULL,20);
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'20-FEB-81',1600,300,30);
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'22-FEB-81',1250,500,30);
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'02-APR-81',2975,NULL,20);
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'28-SEP-81',1250,1400,30);
INSERT INTO emp VALUES (7698,'BLAKE','MANAGER',7839,'01-MAY-81',2850,NULL,30);
INSERT INTO emp VALUES (7782,'CLARK','MANAGER',7839,'09-JUN-81',2450,NULL,10);
INSERT INTO emp VALUES (7788,'SCOTT','ANALYST',7566,'19-APR-87',3000,NULL,20);
INSERT INTO emp VALUES (7839,'KING','PRESIDENT',NULL,'17-NOV-81',5000,NULL,10);
INSERT INTO emp VALUES (7844,'TURNER','SALESMAN',7698,'08-SEP-81',1500,0,30);
INSERT INTO emp VALUES (7876,'ADAMS','CLERK',7788,'23-MAY-87',1100,NULL,20);
INSERT INTO emp VALUES (7900,'JAMES','CLERK',7698,'03-DEC-81',950,NULL,30);
INSERT INTO emp VALUES (7902,'FORD','ANALYST',7566,'03-DEC-81',3000,NULL,20);
INSERT INTO emp VALUES (7934,'MILLER','CLERK',7782,'23-JAN-82',1300,NULL,10);
--
--  Load the 'jobhist' table
--
INSERT INTO jobhist VALUES (7369,'17-DEC-80',NULL,'CLERK',800,NULL,20,
  'New Hire');
INSERT INTO jobhist VALUES (7499,'20-FEB-81',NULL,'SALESMAN',1600,300,30,
  'New Hire');
INSERT INTO jobhist VALUES (7521,'22-FEB-81',NULL,'SALESMAN',1250,500,30,
  'New Hire');
INSERT INTO jobhist VALUES (7566,'02-APR-81',NULL,'MANAGER',2975,NULL,20,
  'New Hire');
INSERT INTO jobhist VALUES (7654,'28-SEP-81',NULL,'SALESMAN',1250,1400,30,
  'New Hire');
INSERT INTO jobhist VALUES (7698,'01-MAY-81',NULL,'MANAGER',2850,NULL,30,
  'New Hire');
INSERT INTO jobhist VALUES (7782,'09-JUN-81',NULL,'MANAGER',2450,NULL,10,
  'New Hire');
INSERT INTO jobhist VALUES (7788,'19-APR-87','12-APR-88','CLERK',1000,NULL,20,
  'New Hire');
INSERT INTO jobhist VALUES (7788,'13-APR-88','04-MAY-89','CLERK',1040,NULL,20,
  'Raise');
INSERT INTO jobhist VALUES (7788,'05-MAY-90',NULL,'ANALYST',3000,NULL,20,
  'Promoted to Analyst');
INSERT INTO jobhist VALUES (7839,'17-NOV-81',NULL,'PRESIDENT',5000,NULL,10,
  'New Hire');
INSERT INTO jobhist VALUES (7844,'08-SEP-81',NULL,'SALESMAN',1500,0,30,
  'New Hire');
INSERT INTO jobhist VALUES (7876,'23-MAY-87',NULL,'CLERK',1100,NULL,20,
  'New Hire');
INSERT INTO jobhist VALUES (7900,'03-DEC-81','14-JAN-83','CLERK',950,NULL,10,
  'New Hire');
INSERT INTO jobhist VALUES (7900,'15-JAN-83',NULL,'CLERK',950,NULL,30,
  'Changed to Dept 30');
INSERT INTO jobhist VALUES (7902,'03-DEC-81',NULL,'ANALYST',3000,NULL,20,
  'New Hire');
INSERT INTO jobhist VALUES (7934,'23-JAN-82',NULL,'CLERK',1300,NULL,10,
  'New Hire');
```

```
SET SQLCOMPAT PLSQL;
--
--  Procedure that lists all employees' numbers and names
--  from the 'emp' table using a cursor
--
CREATE OR REPLACE PROCEDURE list_emp
IS
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    CURSOR emp_cur IS
        SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
    OPEN emp_cur;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cur INTO v_empno, v_ename;
        EXIT WHEN emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_cur;
END;
/
--
--  Procedure that selects an employee row given the employee
--  number and displays certain columns
--
CREATE OR REPLACE PROCEDURE select_emp (
    p_empno         IN  NUMBER
)
IS
    v_ename         emp.ename%TYPE;
    v_hiredate      emp.hiredate%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_dname         dept.dname%TYPE;
    v_disp_date     VARCHAR2(10);
BEGIN
    SELECT ename, hiredate, sal, NVL(comm, 0), dname
        INTO v_ename, v_hiredate, v_sal, v_comm, v_dname
        FROM emp e, dept d
        WHERE empno = p_empno
          AND e.deptno = d.deptno;
    v_disp_date := TO_CHAR(v_hiredate, 'YYYY/MM/DD');
    DBMS_OUTPUT.PUT_LINE('Number    : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name      : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Hire Date : ' || v_disp_date);
    DBMS_OUTPUT.PUT_LINE('Salary    : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Commission: ' || v_comm);
    DBMS_OUTPUT.PUT_LINE('Department: ' || v_dname);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
        DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
        DBMS_OUTPUT.PUT_LINE(SQLCODE);
END;
/
--
--  Procedure that queries the 'emp' table based on
--  department number and employee number or name. Returns
--  employee number and name as IN OUT parameters and job,
--  hire date, and salary as OUT parameters.
--
CREATE OR REPLACE PROCEDURE emp_query (
```

```
    p_deptno        IN     NUMBER,
    p_empno         IN OUT NUMBER,
    p_ename         IN OUT VARCHAR2,
    p_job           OUT    VARCHAR2,
    p_hiredate      OUT    DATE,
    p_sal           OUT    NUMBER
)
IS
BEGIN
    SELECT empno, ename, job, hiredate, sal
        INTO p_empno, p_ename, p_job, p_hiredate, p_sal
        FROM emp
        WHERE deptno = p_deptno
          AND (empno = p_empno
           OR  ename = UPPER(p_ename));
END;
/
--
--  Procedure to call 'emp_query_caller' with IN and IN OUT
--  parameters. Displays the results received from IN OUT and
--  OUT parameters.
--
CREATE OR REPLACE PROCEDURE emp_query_caller
IS
    v_deptno        NUMBER(2);
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_hiredate      DATE;
    v_sal           NUMBER;
BEGIN
    v_deptno := 30;
    v_empno  := 0;
    v_ename  := 'Martin';
    emp_query(v_deptno, v_empno, v_ename, v_job, v_hiredate, v_sal);
    DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('More than one employee was selected');
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('No employees were selected');
END;
/
--
--  Function to compute yearly compensation based on semimonthly
--  salary
--
CREATE OR REPLACE FUNCTION emp_comp (
    p_sal           NUMBER,
    p_comm          NUMBER
) RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END;
/
--
--  After statement-level triggers that display a message after
--  an insert, update, or deletion to the 'emp' table. One message
--  per SQL command is displayed.
--
CREATE OR REPLACE TRIGGER user_ins_audit_trig
```

```
    AFTER INSERT ON emp
    FOR EACH ROW
DECLARE
    v_action        VARCHAR2(24);
BEGIN
    v_action := ' added employee(s) on ';
    DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action ||
      TO_CHAR(SYSDATE,'YYYY-MM-DD'));
END;
/
CREATE OR REPLACE TRIGGER user_upd_audit_trig
    AFTER UPDATE ON emp
    FOR EACH ROW
DECLARE
    v_action        VARCHAR2(24);
BEGIN
    v_action := ' updated employee(s) on ';
    DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action ||
      TO_CHAR(SYSDATE,'YYYY-MM-DD'));
END;
/
CREATE OR REPLACE TRIGGER user_del_audit_trig
    AFTER DELETE ON emp
    FOR EACH ROW
DECLARE
    v_action        VARCHAR2(24);
BEGIN
    v_action := ' deleted employee(s) on ';
    DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action ||
      TO_CHAR(SYSDATE,'YYYY-MM-DD'));
END;
/
--
--  Before row-level triggers that display employee number and
--  salary of an employee that is about to be added, updated,
--  or deleted in the 'emp' table
--
CREATE OR REPLACE TRIGGER emp_ins_sal_trig
    BEFORE INSERT ON emp
    FOR EACH ROW
DECLARE
    sal_diff       NUMBER;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Inserting employee ' || :NEW.empno);
    DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
END;
/
CREATE OR REPLACE TRIGGER emp_upd_sal_trig
    BEFORE UPDATE ON emp
    FOR EACH ROW
DECLARE
    sal_diff       NUMBER;
BEGIN
    sal_diff := :NEW.sal - :OLD.sal;
    DBMS_OUTPUT.PUT_LINE('Updating employee ' || :OLD.empno);
    DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
    DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
    DBMS_OUTPUT.PUT_LINE('..Raise     : ' || sal_diff);
END;
/
CREATE OR REPLACE TRIGGER emp_del_sal_trig
    BEFORE DELETE ON emp
    FOR EACH ROW
DECLARE
    sal_diff       NUMBER;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Deleting employee ' || :OLD.empno);
```

```
    DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
END;
/
--
--  Package specification for the 'emp_admin' package
--
CREATE OR REPLACE PACKAGE emp_admin
IS
    FUNCTION get_dept_name (
        p_deptno        NUMBER
    ) RETURN VARCHAR2;
    FUNCTION update_emp_sal (
        p_empno         NUMBER,
        p_raise         NUMBER
    ) RETURN NUMBER;
    PROCEDURE hire_emp (
        p_empno         NUMBER,
        p_ename         VARCHAR2,
        p_job           VARCHAR2,
        p_sal           NUMBER,
        p_hiredate      DATE,
        p_comm          NUMBER,
        p_mgr           NUMBER,
        p_deptno        NUMBER
    );
    PROCEDURE fire_emp (
        p_empno         NUMBER
    );
END emp_admin;
/
--
--  Package body for the 'emp_admin' package
--
CREATE OR REPLACE PACKAGE BODY emp_admin
IS
    --
    --  Function that queries the 'dept' table based on the department
    --  number and returns the corresponding department name
    --
    FUNCTION get_dept_name (
        p_deptno        IN NUMBER
    ) RETURN VARCHAR2
    IS
        v_dname         VARCHAR2(14);
    BEGIN
        SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
        RETURN v_dname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Invalid department number ' || p_deptno);
            RETURN '';
    END;
    --
    --  Function that updates an employee's salary based on the
    --  employee number and salary increment/decrement passed
    --  as IN parameters. Upon successful completion the function
    --  returns the new updated salary.
    --
    FUNCTION update_emp_sal (
        p_empno         IN NUMBER,
        p_raise         IN NUMBER
    ) RETURN NUMBER
    IS
        v_sal           NUMBER := 0;
    BEGIN
        SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
        v_sal := v_sal + p_raise;
```

```
        UPDATE emp SET sal = v_sal WHERE empno = p_empno;
        RETURN v_sal;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
            RETURN -1;
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
            DBMS_OUTPUT.PUT_LINE(SQLERRM);
            DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
            DBMS_OUTPUT.PUT_LINE(SQLCODE);
            RETURN -1;
    END;
    --
    --  Procedure that inserts a new employee record into the 'emp' table
    --
    PROCEDURE hire_emp (
        p_empno         NUMBER,
        p_ename         VARCHAR2,
        p_job           VARCHAR2,
        p_sal           NUMBER,
        p_hiredate      DATE,
        p_comm          NUMBER,
        p_mgr           NUMBER,
        p_deptno        NUMBER
    )
    AS
    BEGIN
        INSERT INTO emp(empno, ename, job, sal, hiredate, comm, mgr, deptno)
            VALUES(p_empno, p_ename, p_job, p_sal,
                   p_hiredate, p_comm, p_mgr, p_deptno);
    END;
    --
    --  Procedure that deletes an employee record from the 'emp' table based
    --  on the employee number
    --
    PROCEDURE fire_emp (
        p_empno         NUMBER
    )
    AS
    BEGIN
        DELETE FROM emp WHERE empno = p_empno;
    END;
END;
/

SET SQLCOMPAT DB2;
```

第 11 章 模糊化

模糊化对诸如例程、触发器、视图和 PL/SQL 程序包等数据库对象的 DDL 语句主体进行编码。使代码模糊化有助于保护您的知识产权，这是因为其他用户无法阅读此代码，但是 DB2 Database for Linux, UNIX, and Windows 仍然可以理解此代码。

DBMS_DDL 模块提供了两个例程，用于使例程、触发器、视图或 PL/SQL 程序包模糊化:

WRAP 函数

将例程、触发器、PL/SQL 程序包或者 PL/SQL 程序包主体定义作为自变量，然后生成一个字符串，其中包含初始头，后面跟着该语句的其余部分的模糊化版本。例如，输入为如下所示:

```
CREATE PROCEDURE P(a INT)
BEGIN
  INSERT INTO T1 VALUES (a);
END
```

使用 DBMS_DDL.WRAP 函数可能会导致:

```
CREATE PROCEDURE P(a INT) WRAPPED SQL09072
aBcDefg12AbcasHGJG6JKHhgkjFGHHkkkljljk878979HJHui99
```

DDL 语句的模糊化部分包含代码页不变量字符，确保它对于任何代码页都有效。

CREATE_WRAPPED 过程

与上面所描述的 WRAP 函数采用相同的输入，但它不会返回模糊化的文本，而是在数据库中创建一个对象。在内部不会使该对象模糊化，以便编译器可以处理该对象；但是，在诸如 SYSCAT.ROUTINES 或 SYSCAT.TRIGGERS 等目录视图中，会使 TEXT 列的内容模糊化。

可以在 CLP 脚本中使用已模糊化的语句，并且可以使用客户机接口将已模糊化的语句作为动态 SQL 语句来提交。

模糊化可用于下列语句:

- **db2look**（通过使用 **-wrap** 选项）
- CREATE FUNCTION
- CREATE PACKAGE
- CREATE PACKAGE BODY
- CREATE PROCEDURE
- CREATE TRIGGER
- CREATE VIEW
- ALTER MODULE

使用 **-wrap** 选项时，**db2look** 工具会模糊化所有上述语句。

第 12 章 块 (PL/SQL)

可以将 PL/SQL 块结构包括在 PL/SQL 过程、函数或触发器定义中，也可以将其作为匿名块语句独立执行。

PL/SQL 块结构和匿名块语句包含下列一个或多个节:

- 可选的声明节
- 必需的可执行节
- 可选的异常节

这些节可以包含 SQL 语句、PL/SQL 语句、数据类型和变量声明或者其他 PL/SQL 语言元素。

匿名块语句 (PL/SQL)

PL/SQL 匿名块语句是可以包含 PL/SQL 控制语句和 SQL 语句的可执行语句。它可以用来在脚本语言中实现过程逻辑。在 PL/SQL 上下文中，此语句可以由 DB2 数据服务器编译和执行。

匿名块语句并不保存在数据库中，并且，它可以包含多达 3 个节: 可选的声明节、必需的可执行节以及可选的异常节。

可选的声明节可以包含将由可执行节和异常节中的语句使用的变量、游标和类型的声明，并且被插入到可执行的 BEGIN-END 块之前。

可选的异常节可以被插入到 BEGIN-END 块的末尾附近。异常节必须以关键字 EXCEPTION 开始，并且一直持续到它所在的块的末尾为止。

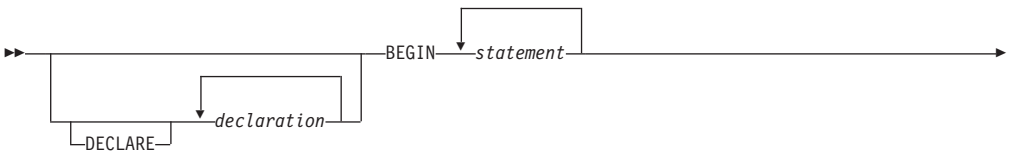
调用

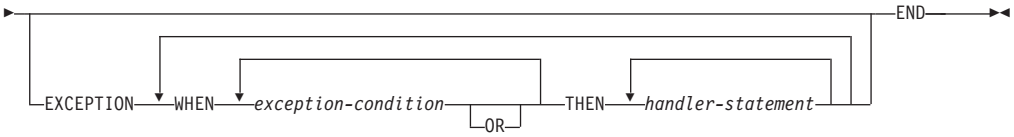
可以从交互式工具或命令行界面（例如 CLP）中执行此语句。另外，还可以在 PL/SQL 过程定义、函数定义或触发器定义中嵌入此语句。在这些上下文中，此语句被称为块结构，而不是被称为匿名块语句。

授权

调用匿名块不需要任何特权。但是，此语句的授权标识所拥有的特权必须包括调用匿名块中嵌入的 SQL 语句所必需的所有特权。

语法





描述

DECLARE
用于开始 DECLARE 语句的可选关键字，此关键字可用于声明数据类型、变量或游标。此关键字的使用取决于此块所在的上下文。

declaration
指定作用域限定于块的数据类型、变量、游标、异常或过程声明。每个声明都必须以分号终止。

BEGIN
用于引入可执行节的必需关键字，该节可以包含一个或多个 SQL 或 PL/SQL 语句。BEGIN-END 块可以包含嵌套的 BEGIN-END 块。

statement
指定 PL/SQL 或 SQL 语句。每个语句都必须以分号终止。

EXCEPTION
用于引入异常节的可选关键字。

WHEN *exception-condition*
指定用于测试一种或多种异常的条件表达式。

THEN *handler-statement*
指定所抛出的异常与 *exception-condition* 中的异常匹配时要执行的 PL/SQL 或 SQL 语句。每个语句都必须以分号终止。

END
用于结束块的必需关键字。

示例

以下示例说明 DB2 数据服务器可以编译的最简单的匿名块语句:

```
BEGIN
    NULL;
END;
```

以下示例说明您可以通过 DB2 CLP 以交互方式输入的匿名块:

```
SET SERVEROUTPUT ON;

BEGIN
  dbms_output.put_line( 'Hello' );
END;
```

以下示例说明您可以通过 DB2 CLP 以交互方式输入的带有声明节的匿名块:

```
SET SERVEROUTPUT ON;

DECLARE
```

```
   current_date DATE := SYSDATE;
BEGIN
     dbms_output.put_line( current_date );
END;
```

第 13 章 过程 (PL/SQL)

DB2 数据服务器支持编译和执行 PL/SQL 过程。PL/SQL 过程是数据库对象，它们包含可以从允许使用 CALL 语句或过程引用的上下文中调用的 PL/SQL 过程逻辑和 SQL 语句。

要创建 PL/SQL 过程，请执行 PL/SQL CREATE PROCEDURE 语句。通过使用 DB2 SQL DROP 语句，可以从数据库中删除此类过程。如果要替换某个过程的实现，那么不需要将其删除。您可以使用 CREATE PROCEDURE 语句并指定 OR REPLACE 选项来替换过程实现。

CREATE PROCEDURE 语句 (PL/SQL)

CREATE PROCEDURE 语句用于定义存储在数据库中的过程。

调用

可以从 DB2 命令行处理器 (CLP)、任何受支持的交互式 SQL 界面、应用程序或例程中执行此语句。

授权

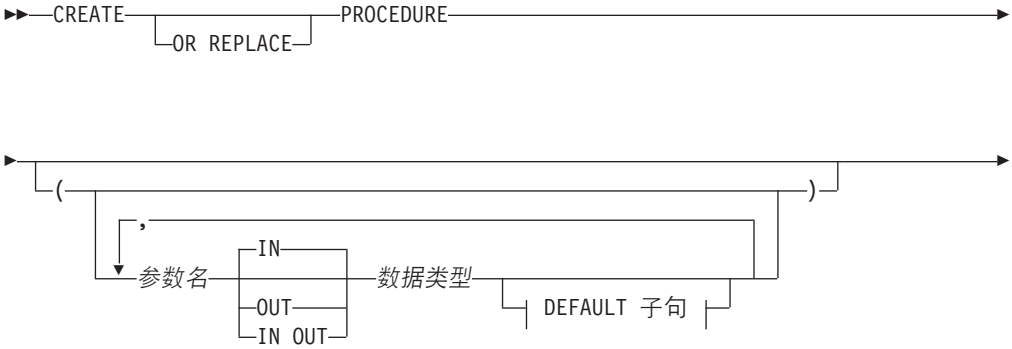
此语句的授权标识所拥有的特权必须至少包括下列其中一项:

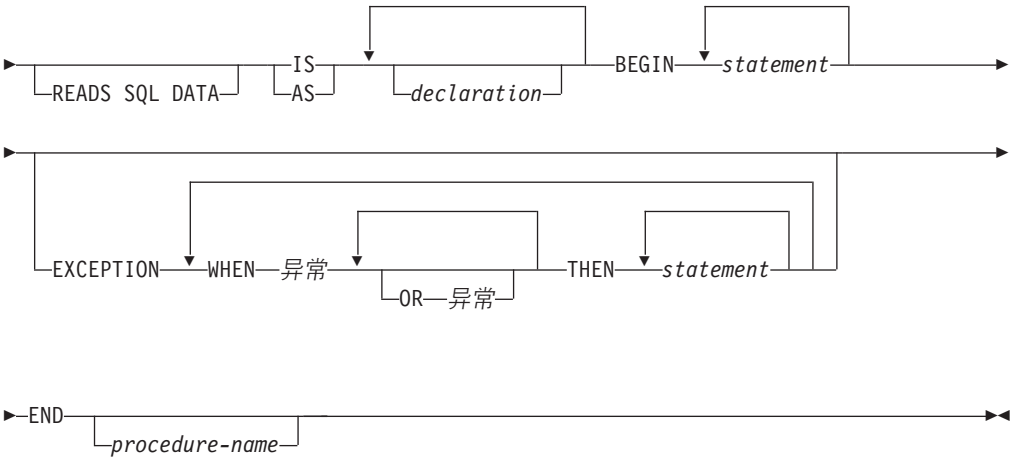
- 如果此过程的模式名不存在: 对数据库的 IMPLICIT_SCHEMA 权限
- 如果此过程的模式名引用现有的模式: 对该模式的 CREATEIN 特权
- DBADM 权限

此语句的授权标识所拥有的特权还必须包括调用过程体中指定的 SQL 语句所需的所有特权。

如果指定了 OR REPLACE，那么此语句的授权标识必须是匹配的过程的所有者 (SQLSTATE 42501)。

语法





描述

PROCEDURE *procedure-name*

指定过程的标识。*procedure-name* 的未限定格式是最大长度为 128 的 SQL 标识。在动态 SQL 语句中，使用 CURRENT SCHEMA 专用寄存器的值对未限定的对象名进行限定。在静态 SQL 语句中，QUALIFIER 预编译或绑定选项隐式地指定未限定的对象名的限定符。*procedure-name* 的限定格式是后面跟着句点字符和 SQL 标识的模式名。如果指定了两部分的名称，那么模式名不能以“SYS”开头；否则，将返回错误 (SQLSTATE 42939)。

名称（包括隐式或显式的限定符）与参数数目不能标识已在目录中描述的过程 (SQLSTATE 42723)。未限定的名称与参数数目在其模式中唯一，但不必跨模式唯一。

参数名

指定参数的名称。参数名对此过程必须唯一 (SQLSTATE 42734)。

数据类型

指定其中一种受支持的 PL/SQL 数据类型。

READS SQL DATA

指示此过程可以包含不修改 SQL 数据的 SQL 语句。此子句是 DB2 扩展。

IS 或 AS

引入过程体定义。

declaration

指定一个或多个变量、游标或 REF CURSOR 类型声明。

BEGIN

引入可执行块。BEGIN-END 块可以包含 EXCEPTION 节。

statement

指定 PL/SQL 或 SQL 语句。此语句必须以分号终止。

EXCEPTION

用于引入异常节的可选关键字。

WHEN *exception-condition*
指定用于测试一种或多种异常的条件表达式。

statement
指定 PL/SQL 或 SQL 语句。此语句必须以分号终止。

END
用于结束块的必需关键字。（可选）可以指定过程的名称。

备注

可以采用已模糊化的格式来提交 CREATE PROCEDURE 语句。在已模糊化的语句中，只有过程名可读。按照下面这样一种方式对该语句的其余内容进行编码：这些内容不可读，但是可由数据库服务器解码。可以通过调用 DBMS_DDL.WRAP 函数来生成模糊化的语句。

示例

以下示例演示没有参数的单一过程:

```
CREATE OR REPLACE PROCEDURE simple_procedure
IS
BEGIN
   DBMS_OUTPUT.PUT_LINE('That''s all folks!');
END simple_procedure;
```

以下示例演示一个过程，此过程接受一个 IN 参数和一个 OUT 参数，并包含标号具有标准 PL/SQL 格式（**<<标号>>**）的 GOTO 语句:

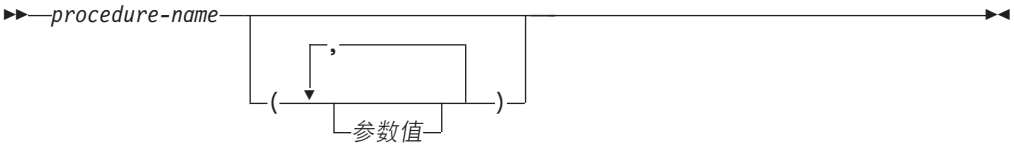
```
CREATE OR REPLACE PROCEDURE test_goto
( p1 IN INTEGER, out1 OUT VARCHAR2(30) )
IS
BEGIN
 <<LABEL2ABOVE>>
 IF p1 = 1 THEN
  out1 := out1 || 'one';
  GOTO LABEL1BELOW;
 END IF;
 if out1 IS NULL THEN
  out1 := out1 || 'two';
  GOTO LABEL2ABOVE;
 END IF;

 out1 := out1 || 'three';

 <<LABEL1BELOW>>
 out1 := out1 || 'four';

END test_goto;
```

过程引用 (PL/SQL)

在 PL/SQL 上下文中对 PL/SQL 过程进行的调用引用可由 DB2 数据服务器编译。

有效的 PL/SQL 过程引用由过程名以及随后的参数（如果有的话）组成。

语法



描述

procedure-name
指定过程的标识。

参数值
指定参数值。如果不传递任何参数，那么调用过程时，可以指定圆括号，也可以不指定圆括号。

示例

以下示例说明如何在 PL/SQL 上下文中调用 PL/SQL 过程:

```
BEGIN
    simple_procedure;
END;
```

在 DB2 数据库中创建 PL/SQL 过程之后，还可以使用 CALL 语句来调用该过程，此语句在 DB2 SQL 上下文中以及使用了受支持 DB2 应用程序编程接口的应用程序中受支持。

函数调用语法支持 (PL/SQL)

许多过程支持 PL/SQL 赋值语句中的函数调用语法。

这些过程包括:

- DBMS_SQL.EXECUTE
- DBMS_SQL.EXECUTE_AND_FETCH
- DBMS_SQL.FETCH_ROWS
- DBMS_SQL.IS_OPEN
- DBMS_SQL.LAST_ERROR_POSITION
- DBMS_SQL.LAST_ROW_COUNT
- DBMS_SQL.OPEN_CURSOR
- UTL_SMTP.CLOSE_DATA
- UTL_SMTP.COMMAND
- UTL_SMTP.COMMAND_REPLIES
- UTL_SMTP.DATA
- UTL_SMTP.EHLO
- UTL_SMTP.HELO
- UTL_SMTP.HELP
- UTL_SMTP.MAIL
- UTL_SMTP.NOOP
- UTL_SMTP.OPEN_DATA

- UTL_SMTP.QUIT
- UTL_SMTP.RCPT
- UTL_SMTP.RSET
- UTL_SMTP.VRFY

示例

```
DECLARE
  cursor1 NUMBER;
  rowsProcessed NUMBER;
BEGIN
  cursor1 := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(cursor1, 'INSERT INTO T1 VALUES (10)', DBMS_SQL.NATIVE);
  rowsProcessed := DBMS_SQL.EXECUTE(cursor1);
  DBMS_SQL.CLOSE_CURSOR(cursor1);
END;
/
```

```
DECLARE
  v_connection UTL_SMTP.CONNECTION;
  v_reply UTL_SMTP.REPLY;
BEGIN
  UTL_SMTP.OPEN_CONNECTION('127.0.0.1', 25, v_connection, 10, v_reply);
  UTL_SMTP.HELO(v_connection,'127.0.0.1');
  UTL_SMTP.MAIL(v_connection, 'sender1@ca.ibm.com');
  UTL_SMTP.RCPT(v_connection, 'receiver1@ca.ibm.com');
  v_reply := UTL_SMTP.OPEN_DATA (v_connection);
  UTL_SMTP.WRITE_DATA (v_connection, 'Test message');
  UTL_SMTP.CLOSE_DATA (v_connection);
  UTL_SMTP.QUIT(v_connection);
END;
/
```

第 14 章 函数 (PL/SQL)

DB2 数据服务器支持编译和执行 PL/SQL 函数。PL/SQL 函数是数据库对象，它们包含可以从允许使用表达式的上下文中调用的 PL/SQL 过程逻辑和 SQL 语句。对 PL/SQL 函数进行求值时，该函数将返回一个值，该值将被替换到该函数所嵌入在的表达式中。

要创建 PL/SQL 函数，请执行 CREATE FUNCTION 语句。通过使用 DB2 SQL DROP 语句，可以从数据库中删除此类函数。如果要替换某个函数的实现，那么不需要将其删除。您可以使用 CREATE FUNCTION 语句并指定 OR REPLACE 选项来替换函数实现。

CREATE FUNCTION 语句 (PL/SQL)

CREATE FUNCTION 语句用于定义存储在数据库中的 SQL 标量函数。标量函数在每次被调用时返回单一的值，并且，在 SQL 表达式有效的位置，标量函数通常都有效。PL/SQL 函数不支持输出参数。

调用

可以从 DB2 命令行处理器、任何受支持的交互式 SQL 界面、应用程序或例程中执行此语句。

授权

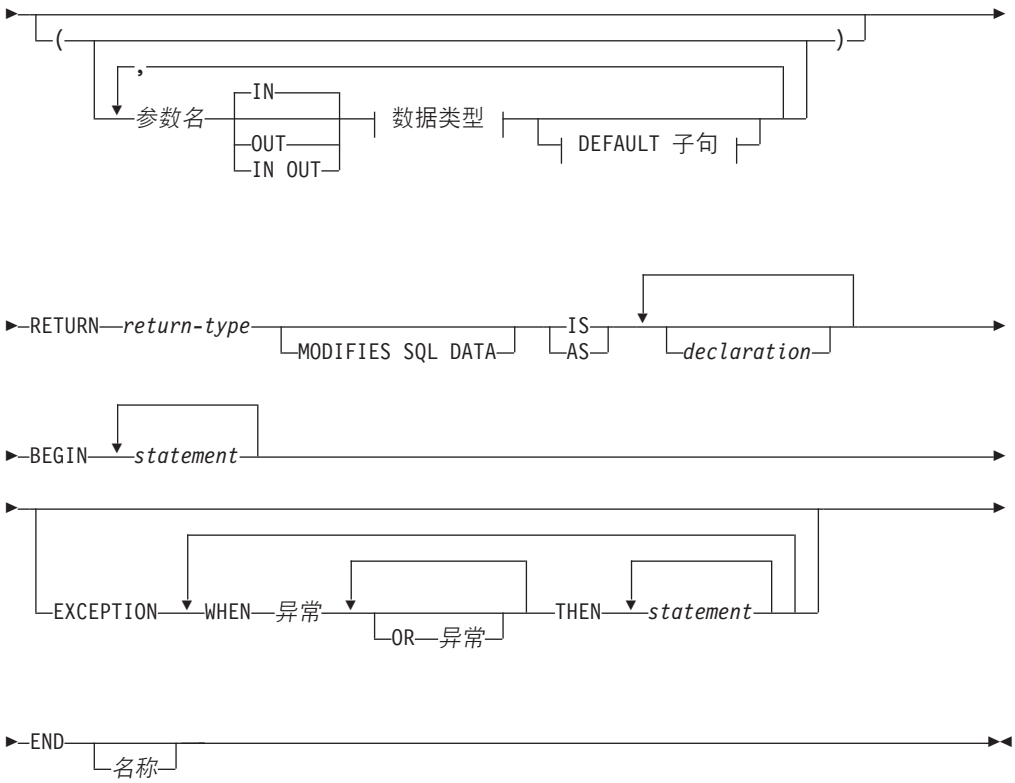
此语句的授权标识所拥有的特权必须至少包括下列其中一项:

- 如果此函数的模式名不存在: 对数据库的 IMPLICIT_SCHEMA 权限
- 如果此函数的模式名引用现有的模式: 对该模式的 CREATEIN 特权
- DBADM 权限

此语句的授权标识所拥有的特权还必须包括调用函数体中指定的 SQL 语句所需的所有特权。

如果指定了 OR REPLACE，那么此语句的授权标识必须是匹配的函数的所有者 (SQLSTATE 42501)。

语法



描述

CREATE FUNCTION 语句指定函数的名称、可选参数、函数的返回类型以及函数的主体。函数的主体是括在 BEGIN 和 END 关键字中的块。它可以包含可选的 EXCEPTION 节，该节定义发生已定义的异常条件时执行的操作。

OR REPLACE
指示当模式中已存在同名的函数时，新函数将替换现有函数。如果未指定此选项，那么新函数无法替换同一模式中现有的同名函数。

FUNCTION *名称*
指定函数的标识。

参数名
指定参数的名称。此名称不能与参数列表中的任何其他参数名相同 (SQLSTATE 42734)。

数据类型
指定其中一种受支持的 PL/SQL 数据类型。

RETURN *return-type*
指定函数所返回的标量值的数据类型。

MODIFIES SQL DATA
指示此函数可以发出除函数所不支持的语句外的任何 SQL 语句 (SQLSTATE 38002 或 42985)。

此子句是 DB2 扩展。当语句中指定了可能修改 SQL 数据的动态 SQL 语句时，必须使用此子句，否则，在函数调用期间，发出尝试修改 SQL 数据的动态语句将失败 (SQLSTATE 38002)。

IS 或 AS
引入用于定义函数体的块。

declaration
指定一个或多个变量、游标或 REF CURSOR 类型声明。

statement
指定一个或多个 PL/SQL 程序语句。每个语句都必须以分号终止。

异常
指定异常条件名。

备注

PL/SQL 函数不能执行任何将要更改数据库管理器未管理的对象的状态的操作。

可以采用已模糊化的格式来提交 CREATE FUNCTION 语句。在已模糊化的语句中，只有函数名可读。按照下面这样一种方式对该语句的其余内容进行编码：这些内容不可读，但是可由数据库服务器解码。可以通过调用 DBMS_DDL.WRAP 函数来生成模糊化的语句。

示例

以下示例演示没有参数的基本函数:

```
CREATE OR REPLACE FUNCTION simple_function
    RETURN VARCHAR2
IS
BEGIN
    RETURN 'That''s All Folks!';
END simple_function;
```

以下示例演示接受两个输入参数的函数:

```
CREATE OR REPLACE FUNCTION emp_comp (
     p_sal           NUMBER,
     p_comm          NUMBER )
RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END emp_comp;
```

函数引用 (PL/SQL)

在表达式受支持的位置，都可以引用 PL/SQL 函数。

语法

描述

函数名
指定函数的标识。

参数值
指定参数的值。

示例

以下示例说明如何从 PL/SQL 匿名块中调用 PL/SQL 样本模式中定义的函数 SIMPLE_FUNCTION:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(simple_function);
END;
```

以下示例说明如何在 SQL 语句中使用函数:

```
SELECT
   empno "EMPNO", ename "ENAME", sal "SAL", comm "COMM",
   emp_comp(sal, comm) "YEARLY COMPENSATION"
FROM emp
```

第 15 章 集合 (PL/SQL)

DB2 数据服务器支持使用 PL/SQL 集合。PL/SQL 集合是一组具有相同数据类型的有序数据元素。通过在圆括号中使用下标表示法，可以引用此集合中的各个数据项。

在 PL/SQL 上下文中，DB2 服务器既支持 VARRAY 集合类型也支持关联数组。

VARRAY 集合类型声明 (PL/SQL)

VARRAY 是一种集合类型，在此集合中，每个元素都由称为*数组下标*的正整数引用。VARRAY 的最大基数在类型定义中指定。

TYPE IS VARRAY 语句用于定义 VARRAY 集合类型。

语法

描述

VARRAY 类型
对此数组类型指定的标识。

n 此数组类型中的最大元素数目。

数据类型
受支持的数据类型，例如 NUMBER、VARCHAR2、RECORD、VARRAY 或关联数组类型。另外，还支持 %TYPE 属性和 %ROWTYPE 属性。

示例

以下示例从 EMP 表中读取职员姓名，将姓名存储在类型为 VARRAY 的数组变量中，然后显示结果。EMP 表包含一个名为 ENAME 的列。此代码从 DB2 脚本（script.db2）中执行。在执行此脚本（db2 -tvf script.db2）之前，应该从 DB2 命令窗口中发出下列命令:

```
db2set DB2_COMPATIBILITY_VECTOR=FFF
db2stop
db2start
```

此脚本包含以下代码:

```
SET SQLCOMPAT PLSQL;

connect to mydb
/

CREATE PACKAGE foo
AS
    TYPE emp_arr_typ IS VARRAY(5) OF VARCHAR2(10);
END;
/

SET SERVEROUTPUT ON
/
```

```
DECLARE
    emp_arr         foo.emp_arr_typ;
    CURSOR emp_cur IS SELECT ename FROM emp WHERE ROWNUM <= 5;
    i               INTEGER := 0;
BEGIN
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp.ename;
    END LOOP;
    FOR j IN 1..5 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j));
    END LOOP;
END;
/

DROP PACKAGE foo
/

connect reset
/
```

此脚本生成以下样本输出:

```
Curly
Larry
Moe
Shemp
Joe
```

CREATE TYPE (VARRAY) 语句 (PL/SQL)

CREATE TYPE (VARRAY) 语句用于定义 VARRAY 数据类型。

调用

可以从 DB2 命令行处理器 (CLP)、任何受支持的交互式 SQL 界面、应用程序或例程中执行此语句。

授权

此语句的授权标识所拥有的特权必须至少包括下列其中一项:

- 如果 VARRAY 类型的模式名不存在: 对数据库的 IMPLICIT_SCHEMA 权限
- 如果 VARRAY 类型的模式名引用现有的模式: 对该模式的 CREATEIN 特权
- DBADM 权限

语法

描述

OR REPLACE

指示当模式中已存在同名的用户定义数据类型时，新数据类型将替换现有数据类型。如果未指定此选项，那么新数据类型无法替换同一模式中现有的同名数据类型。

VARRAY 类型

指定 VARRAY 类型的标识。*VARRAY 类型* 的未限定格式是最大长度为 128 的 SQL 标识。使用 CURRENT SCHEMA 专用寄存器的值对未限定的对象名进行限定。*VARRAY 类型* 的限定格式是后面跟着句点字符和 SQL 标识的模式名。如果指定了两部分的名称，那么模式名不能以“SYS”开头；否则，将返回错误 (SQLSTATE 42939)。名称（包括隐式或显式的限定符）不能标识已在目录中描述的用户定义数据类型 (SQLSTATE 42723)。未限定的名称在其模式中唯一，但不必跨模式唯一。

n 指定此数组类型中的最大元素数目。给定系统上数组的最大基数由可供 DB2 应用程序使用的内存总量限制。因此，虽然可以创建大基数（最高可达 2,147,483,647）数组，但并非所有元素都可用。

数据类型

受支持的数据类型，例如 NUMBER、VARCHAR2、RECORD、VARRAY 或关联数组类型。另外，还支持 %TYPE 属性和 %ROWTYPE 属性。

示例

以下示例创建最多包含 10 个元素的 VARRAY 数据类型，其中，每个元素的数据类型均为 NUMBER:

```
CREATE TYPE NUMARRAY1 AS VARRAY (10) OF NUMBER
```

关联数组 (PL/SQL)

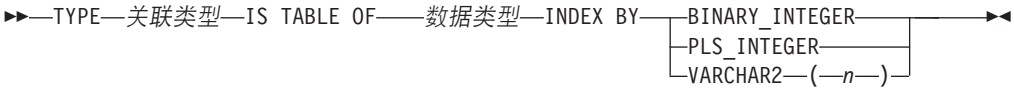
PL/SQL 关联数组是一种集合类型，它使唯一的键与值相关联。

关联数组具有下列特征:

- 必须先定义关联数组类型，然后才能声明具有该数组类型的数组变量。数据处理在数组变量中进行。
- 无需对此数组进行初始化；只需将值指定给数组元素。
- 对此数组中的元素数目没有已定义的限制；此数组可以在添加元素时动态地增大。
- 此数组可以是*稀疏*数组；即，对键进行的赋值可以有间隔。
- 尝试引用尚未进行赋值的数组元素将导致异常。

请使用 TYPE IS TABLE OF 语句来定义关联数组类型。

语法



描述

TYPE *关联类型*
指定数组类型的标识。

数据类型
指定受支持的数据类型，例如 VARCHAR2、NUMBER、RECORD、VARRAY 或关联数组类型。另外，还支持 %TYPE 属性和 %ROWTYPE 属性。

INDEX BY
指定关联数组按此子句所引入的其中一种数据类型建立索引。

BINARY INTEGER
整型数字数据。

PLS_INTEGER
整型数字数据。

VARCHAR2 (*n*)
最大长度为 *n* 的可变长度字符串。如果 %TYPE 属性所应用于的对象具有 BINARY_INTEGER、PLS_INTEGER 或 VARCHAR2 数据类型，那么还支持 %TYPE 属性。

要声明具有关联数组类型的变量，请指定 *数组名 关联类型*，其中*数组名* 表示对关联数组指定的标识，*关联类型* 表示先前声明的数组类型的标识。

要引用数组的特定元素，请指定 *数组名*(*n*)，其中*数组名* 表示先前声明的数组的标识，*n* 表示 INDEX BY 数据类型为*关联类型* 的值。如果根据记录类型来定义数组，那么该引用将变为 *数组名*(*n*).*字段*，其中*字段* 在数组类型的定义所基于记录类型中定义。要引用整个记录，请省略 *字段*。

示例

以下示例从 EMP 表中读取前 10 个职员姓名，将其存储在数组中，然后显示数组内容。

```
SET SERVEROUTPUT ON
/

CREATE OR REPLACE PACKAGE pkg_test_type1
IS
    TYPE emp_arr_typ IS TABLE OF VARCHAR2(10) INDEX BY BINARY_INTEGER;
END pkg_test_type1
/

DECLARE
    emp_arr         pkg_test_type1.emp_arr_typ;
    CURSOR emp_cur IS SELECT ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j));
    END LOOP;
END
/
```

此代码生成以下样本输出:

```
SMITH
ALLEN
WARD
JONES
MARTIN
BLAKE
CLARK
SCOTT
KING
TURNER
```

可以对此示例进行修改，以便在数组定义中使用记录类型。

```
SET SERVEROUTPUT ON
/

CREATE OR REPLACE PACKAGE pkg_test_type2
IS
    TYPE emp_rec_typ IS RECORD (
        empno       INTEGER,
        ename       VARCHAR2(10)
    );
END pkg_test_type2
/

CREATE OR REPLACE PACKAGE pkg_test_type3
IS
    TYPE emp_arr_typ IS TABLE OF pkg_test_type2.emp_rec_typ INDEX BY BINARY_INTEGER;
END pkg_test_type3
/

DECLARE
    emp_arr         pkg_test_type3.emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i).empno := r_emp.empno;
        emp_arr(i).ename := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' ||
            emp_arr(j).ename);
    END LOOP;
END
/
```

修改后的代码生成以下样本输出:

```
EMPNO    ENAME
-----    -------
1001     SMITH
1002     ALLEN
1003     WARD
1004     JONES
1005     MARTIN
1006     BLAKE
1007     CLARK
1008     SCOTT
1009     KING
1010     TURNER
```

可以进一步对此示例进行修改，以便使用 emp%ROWTYPE 属性来定义 emp_arr_typ，而不是使用 emp_rec_typ 记录类型。

```
SET SERVEROUTPUT ON
/

CREATE OR REPLACE PACKAGE pkg_test_type4
IS
    TYPE emp_arr_typ IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
END pkg_test_type4
/

DECLARE
    emp_arr         pkg_test_type4.emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i).empno := r_emp.empno;
        emp_arr(i).ename := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' ||
            emp_arr(j).ename);
    END LOOP;
END
/
```

在此情况下，样本输出与上一个示例完全相同。

最后，可以进行从 r_emp 到 emp_arr 的记录级赋值，而不是对记录的每个字段逐个赋值:

```
SET SERVEROUTPUT ON
/

CREATE OR REPLACE PACKAGE pkg_test_type5
IS
    TYPE emp_rec_typ IS RECORD (
        empno       INTEGER,
        ename       VARCHAR2(10)
    );
END pkg_test_type5
/

CREATE OR REPLACE PACKAGE pkg_test_type6
IS
    TYPE emp_arr_typ IS TABLE OF pkg_test_type5.emp_rec_typ INDEX BY BINARY_INTEGER;
END pkg_test_type6
/

DECLARE
    emp_arr         pkg_test_type6.emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp;
    END LOOP;
    FOR j IN 1..10 LOOP
```

```
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' ||
            emp_arr(j).ename);
    END LOOP;
END
/
```

集合方法 (PL/SQL)

集合方法可用于获取关于集合的信息或者修改集合。

在尝试运行表 4 中的示例之前，应该执行下列命令。

```
db2set DB2_COMPATIBILITY_VECTOR=ORA
db2stop
db2start
db2 connect to mydb
```

MYDB 数据库包含一个表 EMP，该表包含一列 ENAME（定义为 VARCHAR (10)）：

```
db2 select * from emp

ENAME
----------
Curly
Larry
Moe
Shemp
Joe

  5 record(s) selected.
```

表 4. DB2 数据服务器在 PL/SQL 上下文中支持的集合方法

集合方法	描述	示例

COUNT	返回集合中的元素数目。	CREATE PACKAGE foo
		AS
		  TYPE sparse_arr_typ IS TABLE OF NUMBER
		  INDEX BY BINARY_INTEGER;
		END;
		/
		
		SET SERVEROUTPUT ON
		/
		
		DECLARE
		  sparse_arr foo.sparse_arr_typ;
		BEGIN
		  sparse_arr(-10) := -10;
		  sparse_arr(0) := 0;
		  sparse_arr(10) := 10;
		  DBMS_OUTPUT.PUT_LINE('COUNT: ' ||
		  sparse_arr.COUNT);
		END;
		/

表 4. DB2 数据服务器在 PL/SQL 上下文中支持的集合方法 (续)

集合方法	描述	示例

DELETE	从集合中除去所有元素。	CREATE PACKAGE foo
		AS
		TYPE names_typ IS TABLE OF VARCHAR2(10)
		INDEX BY BINARY_INTEGER;
		END;
		/
		
		SET SERVEROUTPUT ON
		/
		
		DECLARE
		actor_names foo.names_typ;
		
		BEGIN
		
		actor_names(1) := 'Chris';
		actor_names(2) := 'Steve';
		actor_names(3) := 'Kate';
		actor_names(4) := 'Naomi';
		actor_names(5) := 'Peter';
		actor_names(6) := 'Philip';
		actor_names(7) := 'Michael';
		actor_names(8) := 'Gary';
		
		DBMS_OUTPUT.PUT_LINE('COUNT: ' ||
		actor_names.COUNT);
		
		actor_names.DELETE(2);
		DBMS_OUTPUT.PUT_LINE('COUNT: ' ||
		actor_names.COUNT);
		
		actor_names.DELETE(3, 5);
		DBMS_OUTPUT.PUT_LINE('COUNT: ' ||
		actor_names.COUNT);
		
		actor_names.DELETE;
		DBMS_OUTPUT.PUT_LINE('COUNT: ' ||
		actor_names.COUNT);
		
		END;
		/

DELETE (*n*)	从关联数组中除去元素 *n*。不能从 VARRAY 集合数组中除去各个元素。	请参阅“DELETE”。
DELETE (*n1*, *n2*)	从关联数组中除去所有从 *n1* 到 *n2* 的元素。不能从 VARRAY 集合数组中除去各个元素。	请参阅“DELETE”。

表 4. DB2 数据服务器在 PL/SQL 上下文中支持的集合方法 (续)

集合方法	描述	示例

EXISTS (*n*)	如果指定的元素存在，那么返回 TRUE。	CREATE PACKAGE foo
		AS
		  TYPE emp_arr_typ IS VARRAY(5) OF VARCHAR2(10);
		END;
		/
		
		SET SERVEROUTPUT ON
		/
		
		DECLARE
		  emp_arr foo.emp_arr_typ;
		  CURSOR emp_cur IS SELECT ename FROM emp
		  WHERE ROWNUM <= 5;
		  i INTEGER := 0;
		BEGIN
		  FOR r_emp IN emp_cur LOOP
		  i := i + 1;
		  emp_arr.EXTEND;
		  emp_arr(i) := r_emp.ename;
		  END LOOP;
		  emp_arr.TRIM;
		  FOR j IN 1..5 LOOP
		  IF emp_arr.EXISTS(j) = true THEN
		  DBMS_OUTPUT.PUT_LINE(emp_arr(j));
		  ELSE
		  DBMS_OUTPUT.PUT_LINE('THIS ELEMENT
		  HAS BEEN DELETED');
		  END IF;
		  END LOOP;
		END;
		/

EXTEND	对集合追加单一 NULL 元素。	请参阅“EXISTS (*n*)”。
EXTEND (*n*)	对集合追加 *n* 个 NULL 元素。	请参阅“EXISTS (*n*)”。
EXTEND (*n1*, *n2*)	对集合追加第 *n2* 个元素的 *n1* 个副本。	请参阅“EXISTS (*n*)”。

表 4. DB2 数据服务器在 PL/SQL 上下文中支持的集合方法 (续)

集合方法	描述	示例

FIRST	返回集合中的最小索引号。	CREATE PACKAGE foo
		AS
		  TYPE emp_arr_typ IS VARRAY(5) OF VARCHAR2(10);
		END;
		/
		
		SET SERVEROUTPUT ON
		/
		
		DECLARE
		  emp_arr foo.emp_arr_typ;
		  CURSOR emp_cur IS SELECT ename FROM emp
		  WHERE ROWNUM <= 5;
		  i INTEGER := 0;
		  k INTEGER := 0;
		  l INTEGER := 0;
		BEGIN
		
		  FOR r_emp IN emp_cur LOOP
		  i := i + 1;
		  emp_arr(i) := r_emp.ename;
		  END LOOP;
		
		  -- Use FIRST and LAST to specify the lower and
		  -- upper bounds of a loop range:
		  FOR j IN emp_arr.FIRST..emp_arr.LAST LOOP
		  DBMS_OUTPUT.PUT_LINE(emp_arr(j));
		  END LOOP;
		
		  -- Use NEXT(n) to obtain the subscript of
		  -- the next element:
		  k := emp_arr.FIRST;
		  WHILE k IS NOT NULL LOOP
		  DBMS_OUTPUT.PUT_LINE(emp_arr(k));
		  k := emp_arr.NEXT(k);
		  END LOOP;
		
		  -- Use PRIOR(n) to obtain the subscript of
		  -- the previous element:
		  l := emp_arr.LAST;
		  WHILE l IS NOT NULL LOOP
		  DBMS_OUTPUT.PUT_LINE(emp_arr(l));
		  l := emp_arr.PRIOR(l);
		  END LOOP;
		
		  DBMS_OUTPUT.PUT_LINE('COUNT: ' || emp_arr.COUNT);
		
		  emp_arr.TRIM;
		  DBMS_OUTPUT.PUT_LINE('COUNT: ' || emp_arr.COUNT);
		
		  emp_arr.TRIM(2);
		  DBMS_OUTPUT.PUT_LINE('COUNT: ' || emp_arr.COUNT);
		
		  DBMS_OUTPUT.PUT_LINE('Max. no. elements = ' ||
		  emp_arr.LIMIT);
		
		END;
		/

LAST	返回集合中的最大索引号。	请参阅“FIRST”。

表 4. DB2 数据服务器在 PL/SQL 上下文中支持的集合方法 (续)

集合方法	描述	示例
LIMIT	对于 VARRAY，返回最大元素数目；对于嵌套的表，返回 NULL。	请参阅“FIRST”。
NEXT (*n*)	返回紧跟在指定元素后面的元素的索引编号。	请参阅“FIRST”。
PRIOR (*n*)	返回正好在指定元素前面的元素的索引编号。	请参阅“FIRST”。
TRIM	从集合末尾除去单个元素。不能从关联数组集合类型中除去元素。	请参阅“FIRST”。
TRIM (*n*)	从集合末尾除去 *n* 个元素。不能从关联数组集合类型中除去元素。	请参阅“FIRST”。

第 16 章 变量 (PL/SQL)

在引用变量之前，必须对其进行声明。

通常，除全局变量或程序包级变量以外，必须在块的声明节中定义该块所使用的变量。声明节包含可以在块中的 PL/SQL 语句中使用的变量、游标和其他类型的定义。变量声明由对变量指定的名称以及该变量的数据类型组成。（可选）可以在变量声明中将变量初始化为缺省值。

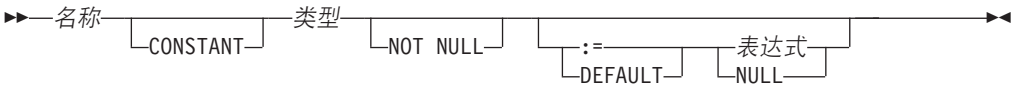
过程和函数可以带有用于传递输入值的参数。过程还可以带有用于传递输出值的参数或者用于同时传递输入值和输出值的参数。

PL/SQL 还包含可变数据类型，以便使用 %TYPE 和 %ROWTYPE 限定符与现有的列、行或游标的数据类型匹配。

变量声明 (PL/SQL)

通常，除全局变量或程序包级变量以外，必须在块的声明节中定义该块所使用的变量。变量声明由对变量指定的名称以及该变量的数据类型组成。（可选）可以在变量声明中将变量初始化为缺省值。

语法



描述

名称
指定要对此变量指定的标识。

CONSTANT
指定变量值是常量。必须指定缺省表达式，并且，不能在应用程序中将新值指定给此变量。

类型
指定此变量的数据类型。

NOT NULL
当前被 DB2 忽略。将变量声明指定为 NOT NULL 的例程已成功编译。但是，这类例程的行为仍然好像尚未指定为 NOT NULL 一样。不会执行运行时检查以禁止在声明为 NOT NULL 的变量中使用空值。如果您的应用程序需要限制 PL/SQL 变量中的空值，请参阅以下示例。

DEFAULT
指定变量的缺省值。每次进入该块时，都将对此缺省值进行求值。例如，如果将 SYSDATE 指定给类型为 DATE 的变量，那么该变量将解析为当前调用时间，而不是解析为先前预编译该过程或函数时的时间。

:= 赋值运算符是 DEFAULT 关键字的同义词。但是，如果在未指定 *表达式* 的情况下指定此运算符，那么此变量将初始化为值 NULL。

表达式
指定进入该块时要指定给此变量的初始值。

NULL
指定 SQL 值 NULL，它具有空值。

示例

1. 以下过程中的变量声明利用了由字符串和数字表达式组成的缺省值:

```
CREATE OR REPLACE PROCEDURE dept_salary_rpt (
    p_deptno        NUMBER
)
IS
    todays_date     DATE := SYSDATE;
    rpt_title       VARCHAR2(60) := 'Report For Department # ' || p_deptno
             || ' on ' || todays_date;
    base_sal        INTEGER := 35525;
    base_comm_rate  NUMBER := 1.33333;
    base_annual     NUMBER := ROUND(base_sal * base_comm_rate, 2);
BEGIN
    DBMS_OUTPUT.PUT_LINE(rpt_title);
    DBMS_OUTPUT.PUT_LINE('Base Annual Salary: ' || base_annual);
END;
```

通过调用此过程，可以获得以下样本输出:

```
CALL dept_salary_rpt(20);

Report For Department # 20 on 10-JUL-07 16:44:45
Base Annual Salary: 47366.55
```

2. 以下示例通过添加隐式检查来限制空值，方法是使用 IS NULL 或 IS NOT NULL 并根据需要处理错误情况:

```
create table T(col1 integer);
insert into T values null;

declare
  N integer not null := 0;
  null_variable exception;
begin
select col1 into N from T;
  if N is null then
    raise null_variable;
  end if;
exception
  when null_variable then
    -- 在此处处理错误条件。
    dbms_output.put_line('Null variable detected');
end;
```

参数方式 (PL/SQL)

PL/SQL 过程参数可以处于下列三种可能方式的一种: IN、OUT 或 IN OUT。PL/SQL 函数参数只能处于 IN 方式。

- 除非显式使用缺省值来初始化 IN 形参，否则该形参将初始化为用于调用该形参的实参。可以在被调用程序中引用 IN 参数；但是，被调用程序不能将新值指定给 IN 参数。在控制权返回到调用程序之后，实参始终包含调用前对其设置的值。

- OUT 形参将初始化为用于调用该形参的实参。被调用程序可以引用此形参以及将新值指定给此形参。如果被调用程序在未发生异常的情况下终止，那么实参将接收到上次对此形参设置的值。如果发生已处理的异常，那么实参将接收到对此形参设置的上一个值。如果发生未处理的异常，那么实参将保持执行调用前对其设置的值。
- 与 IN 参数相似，IN OUT 形参将初始化为用于调用该形参的实参。与 OUT 参数相似，被调用程序可以对 IN OUT 形参进行修改，如果被调用程序在未发生异常的情况下终止，那么形参的上一个值将被传递到调用程序的实参。如果发生已处理的异常，那么实参将接收到对此形参设置的上一个值。如果发生未处理的异常，那么实参将保持执行调用前对其设置的值。

表 5 对此行为作了摘要说明。

表 5. 参数方式

方式属性	IN	IN OUT	OUT
形参初始化为:	实参值	实参值	实参值
被调用程序可以对形参进行修改吗？	否	是	是
在被调用程序正常终止之后，实参包含:	执行调用前的原始实参值	形参的上一个值	形参的上一个值
在被调用程序中发生已处理的异常之后，实参包含:	执行调用前的原始实参值	形参的上一个值	形参的上一个值
在被调用程序中发生未处理的异常之后，实参包含:	执行调用前的原始实参值	执行调用前的原始实参值	执行调用前的原始实参值

数据类型 (PL/SQL)

DB2 数据服务器支持各种可用于在 PL/SQL 块中声明变量的数据类型。

表 6. PL/SQL 中的受支持标量数据类型

PL/SQL 数据类型	DB2 SQL 数据类型	描述
BINARY_INTEGER	INTEGER	整型数字数据
BLOB	BLOB（4096）	二进制数据
BLOB (*n*)	BLOB (*n*) *n* = 1 到 2147483647	二进制大对象数据
BOOLEAN	BOOLEAN	逻辑布尔值（true 或 false）
CHAR	CHAR (1)	长度为 1 的固定长度字符串数据
CHAR (*n*)	CHAR (*n*) *n* = 1 到 254	长度为 *n* 的固定长度字符串数据
CHAR VARYING (*n*)	VARCHAR (*n*)	最大长度为 *n* 的可变长度字符串数据
CHARACTER	CHARACTER (1)	长度为 1 的固定长度字符串数据
CHARACTER (*n*)	CHARACTER (*n*) *n* = 1 到 254	长度为 *n* 的固定长度字符串数据

表 6. PL/SQL 中的受支持标量数据类型 (续)

PL/SQL 数据类型	DB2 SQL 数据类型	描述
CHARACTER VARYING (*n*)	VARCHAR (*n*) *n* = 1 到 32672	最大长度为 *n* 的可变长度字符串数据
CLOB	CLOB (1M)	字符大对象数据
CLOB (*n*)	CLOB (*n*) *n* = 1 到 2147483647	长度为 *n* 的固定长度长字符串数据
DATE	DATE [1]	日期和时间数据（精度为秒）
DEC	DEC (9, 2)	十进制数字数据
DEC (*p*)	DEC (*p*) *p* = 1 到 31	精度为 *p* 的十进制数字数据
DEC (*p*, *s*)	DEC (*p*, *s*) *p* = 1 到 31; *s* = 1 到 31	精度为 *p* 并且小数位数为 *s* 的十进制数字数据
DECIMAL	DECIMAL (9, 2)	十进制数字数据
DECIMAL (*p*)	DECIMAL (*p*) *p* = 1 到 31	精度为 *p* 的十进制数字数据
DECIMAL (*p*, *s*)	DECIMAL (*p*, *s*) *p* = 1 到 31; *s* = 1 到 31	精度为 *p* 并且小数位数为 *s* 的十进制数字数据
DOUBLE	DOUBLE	双精度浮点数
DOUBLE PRECISION	DOUBLE PRECISION	双精度浮点数
FLOAT	FLOAT	浮点数字数据
FLOAT (*n*) *n* = 1 到 24	REAL	实数数字数据
FLOAT (*n*) *n* = 25 到 53	DOUBLE	双精度数字数据
INT	INT	带符号 4 字节整型数字数据
INTEGER	INTEGER	带符号 4 字节整型数字数据
LONG	CLOB (32760)	字符大对象数据
LONG RAW	BLOB (32760)	二进制大对象数据
LONG VARCHAR	CLOB (32760)	字符大对象数据
NATURAL	INTEGER	带符号 4 字节整型数字数据
NCHAR	GRAPHIC (127)	固定长度图形字符串数据
NCHAR (*n*) *n* = 1 到 2000	GRAPHIC (*n*) *n* = 1 到 127	长度为 *n* 的固定长度图形字符串数据
NCLOB [2]	DBCLOB (1M)	双字节字符大对象数据
NCLOB (*n*)	DBCLOB (2000)	最大长度为 *n* 的双字节长字符串数据
NVARCHAR2	VARGRAPHIC (2048)	可变长度图形字符串数据
NVARCHAR2 (*n*)	VARGRAPHIC (*n*)	最大长度为 *n* 的可变长度图形字符串数据
NUMBER	NUMBER [3]	精确数字数据
NUMBER (*p*)	NUMBER (*p*) [3]	最大精度为 *p* 的精确数字数据
NUMBER (*p*, *s*)	NUMBER (*p*, *s*) [3] *p* = 1 到 31	最大精度为 *p* 并且小数位数为 *s* 的精确数字数据

表 6. PL/SQL 中的受支持标量数据类型 (续)

PL/SQL 数据类型	DB2 SQL 数据类型	描述
NUMERIC	NUMERIC (9.2)	精确数字数据
NUMERIC (*p*)	NUMERIC (*p*) *p* = 1 到 31	最大精度为 *p* 的精确数字数据
NUMERIC (*p*, s)	NUMERIC (*p*, *s*) *p* = 1 到 31; *s* = 0 到 31	最大精度为 *p* 并且小数位数为 *s* 的精确数字数据
PLS_INTEGER	INTEGER	整型数字数据
RAW	BLOB (32767)	二进制大对象数据
RAW (*n*)	BLOB (*n*) *n* = 1 到 32767	二进制大对象数据
SMALLINT	SMALLINT	带符号 2 字节整型数据
TIMESTAMP (0)	TIMESTAMP (0)	包含时间戳记信息的日期数据
TIMESTAMP (*p*)	TIMESTAMP (*p*)	包含可选的小数秒和精度 *p* 的日期和时间数据
VARCHAR	VARCHAR (4096)	最大长度为 4096 个字符的可变长度字符串数据
VARCHAR (*n*)	VARCHAR (*n*)	最大长度为 *n* 个字符的可变长度字符串数据
VARCHAR2 (*n*)	VARCHAR2 (*n*) [4]	最大长度为 *n* 个字符的可变长度字符串数据

1. 如果为 DATE 数据类型设置了 **DB2_COMPATIBILITY_VECTOR** 注册表变量，那么 DATE 等同于 TIMESTAMP (0)。
2. 要了解特定数据库环境中有关 NCLOB 数据类型的限制，请参阅“PL/SQL 支持方面的限制”。
3. 当 **number_compat** 数据库配置参数设置为 ON 时，支持此数据类型。
4. 当 **varchar2_compat** 数据库配置参数设置为 ON 时，支持此数据类型。

除第 145 页的表 6 描述的标量数据类型以外，DB2 数据服务器还支持集合类型、记录类型和 REF CURSOR 类型。

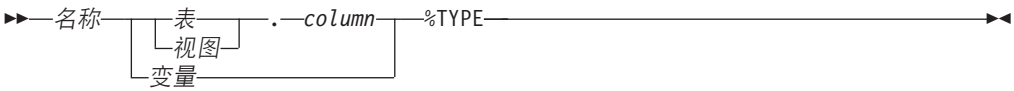
变量声明中的 %TYPE 属性 (PL/SQL)

DB2 数据服务器支持在 PL/SQL 变量声明和参数声明中使用的 %TYPE 属性。使用此属性将确保维持表列与 PL/SQL 变量之间的类型兼容性。

必须指定点分表示法中的限定列名或者先前声明的变量的名称作为 %TYPE 属性的前缀。此列或变量的数据类型将被指定给所声明的变量。即使此列或变量的数据类型发生更改，也不需要修改声明代码。

%TYPE 属性也可以与形参声明配合使用。

语法



描述

名称
指定所声明的变量或形参的标识。

表 指定表的标识，将引用该表中的列。

视图
指定视图的标识，将引用该视图中的列。

column
指定要引用的表列或视图列的标识。

变量
指定先前声明的变量的标识，将引用此变量。此变量不会继承任何其他列属性（例如可空属性）。

示例

在以下示例中，一个过程使用职员编号来查询 EMP 表、显示该职员的数据并查找该职员所属部门中所有职员的平均薪水，然后将所选职员的薪水与部门平均薪水进行比较。

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN NUMBER
)
IS
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_deptno        NUMBER(2);
    v_avgsal        NUMBER(7,2);
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO v_ename, v_job, v_hiredate, v_sal, v_deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || v_deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = v_deptno;
    IF v_sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the department '
            || 'average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the department '
            || 'average of ' || v_avgsal);
    END IF;
END;
```

您可以重新编写此过程，以便不在声明节中明确编码 EMP 表数据类型。

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN emp.empno%TYPE
)
IS
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_hiredate      emp.hiredate%TYPE;
```

```
    v_sal           emp.sal%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_avgsal        v_sal%TYPE;
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO v_ename, v_job, v_hiredate, v_sal, v_deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || v_deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = v_deptno;
    IF v_sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the department '
            || 'average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the department '
            || 'average of ' || v_avgsal);
    END IF;
END;
```

p_empno 参数是使用 %TYPE 属性定义的形参的示例。v_avgsal 变量是引用了另一个变量（而不是引用表列）的 %TYPE 属性的示例。

调用 EMP_SAL_QUERY 过程将生成以下样本输出:

```
CALL emp_sal_query(7698);

Employee # : 7698
Name       : BLAKE
Job        : MANAGER
Hire Date  : 01-MAY-81 00:00:00
Salary     : 2850.00
Dept #     : 30
Employee's salary is more than the department average of 1566.67
```

基于用户定义记录类型的记录变量 (PL/SQL)

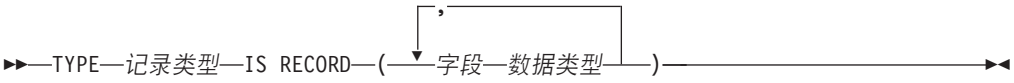
在 PL/SQL 上下文中，DB2 数据服务器支持基于用户定义记录类型定义的 PL/SQL 记录变量声明。

*记录类型*是由一个或多个标识及其相应数据类型组成的记录的定义。记录类型本身不能用来处理数据。您可以声明基于现有用户定义记录类型的 PL/SQL 记录变量，并可以使用 PL/SQL TYPE IS RECORD 语句来创建用户定义记录类型。只有在 CREATE PACKAGE 或 CREATE PACKAGE BODY 语句中，才支持记录类型定义。

记录变量（即，记录）是记录类型的实例。记录变量根据记录类型进行声明。记录的属性（例如它的字段名和字段类型）从记录类型继承。

请使用点分表示法来引用记录中的字段。例如，`record.field`。

语法



描述

TYPE *记录类型* **IS RECORD**
指定记录类型的标识。

字段
指定该记录类型的某个字段的标识。

数据类型
指定 *字段* 的相应数据类型。支持 %TYPE 属性、RECORD、VARRAY、关联数组类型和 %ROWTYPE 属性。

示例

以下示例演示引用了用户定义记录类型的程序包:

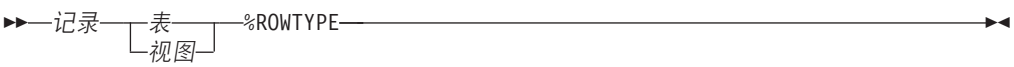
```
CREATE OR REPLACE PACKAGE pkg7a
IS
TYPE t1_typ IS RECORD (
  c1 T1.C1%TYPE,
  c2 VARCHAR(10)
);
END;
```

记录类型声明中的 %ROWTYPE 属性 (PL/SQL)

DB2 数据服务器支持 %ROWTYPE 属性，此属性用于声明类型为记录的 PL/SQL 变量，该记录的字段与表列或视图列相对应。PL/SQL 记录中的每个字段都采用表中相应列的数据类型。

*记录*是字段的有序集合，它具有名称。*字段*与变量类似；它具有标识和数据类型，但还属于某个记录，并且必须在将记录名用作限定符的情况下通过点分表示法进行引用。

语法



描述

记录
指定记录的标识。

表 指定一个表的标识，将使用该表的列定义来定义记录中的字段。

视图
指定一个视图的标识，将使用该视图的列定义来定义记录中的字段。

%ROWTYPE

指定根据与所标识表或视图相关联的列数据类型来派生记录字段数据类型。记录字段不会继承任何其他列属性（例如可空属性）。

示例

以下示例说明如何使用 %ROWTYPE 属性来创建名为 r_emp 的记录，以代替为 EMP 表中的列声明各个变量。

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN emp.empno%TYPE
)
IS
    r_emp           emp%ROWTYPE;
    v_avgsal        emp.sal%TYPE;
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO r_emp.ename, r_emp.job, r_emp.hiredate, r_emp.sal, r_emp.deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || r_emp.ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || r_emp.job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || r_emp.hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || r_emp.sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || r_emp.deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = r_emp.deptno;
    IF r_emp.sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the department '
            || 'average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the department '
            || 'average of ' || v_avgsal);
    END IF;
END;
```

第 17 章 基本语句 (PL/SQL)

可以在 PL/SQL 应用程序中使用的编程语句包括: 赋值、DELETE、EXECUTE IMMEDIATE、INSERT、NULL、SELECT INTO 和 UPDATE。

NULL 语句 (PL/SQL)

NULL 语句是不执行任何操作的可执行语句。每当需要可执行的语句，但不想执行 SQL 操作时，可以使用 NULL 语句作为占位符；例如，在 IF-THEN-ELSE 语句的分支中使用 NULL 语句。

语法

示例

以下示例说明 DB2 数据服务器可以编译的最简单的有效 PL/SQL 程序:

```
BEGIN
    NULL;
END;
```

以下示例演示 IF...THEN...ELSE 语句中的 NULL 语句:

```
CREATE OR REPLACE PROCEDURE divide_it (
    p_numerator     IN  NUMBER,
    p_denominator   IN  NUMBER,
    p_result        OUT NUMBER
)
IS
BEGIN
    IF p_denominator = 0 THEN
        NULL;
    ELSE
        p_result := p_numerator / p_denominator;
    END IF;
END;
```

赋值语句 (PL/SQL)

赋值语句将先前声明的变量或者 OUT 或 IN OUT 形参设置为某个表达式的值。

语法

描述

变量

指定先前声明的变量、OUT 形参或 IN OUT 形参的标识。

表达式
指定求值为单一值的表达式。此值的数据类型必须与 *变量* 的数据类型兼容。

示例

以下示例演示过程的可执行节中的赋值语句:

```
CREATE OR REPLACE PROCEDURE dept_salary_rpt (
    p_deptno       IN   NUMBER,
    p_base_annual  OUT  NUMBER
)
IS
    todays_date     DATE;
    rpt_title       VARCHAR2(60);
    base_sal        INTEGER;
    base_comm_rate  NUMBER;
BEGIN
    todays_date := SYSDATE;
    rpt_title := 'Report For Department # ' || p_deptno || ' on '
        || todays_date;
    base_sal := 35525;
    base_comm_rate := 1.33333;
    p_base_annual := ROUND(base_sal * base_comm_rate, 2);

    DBMS_OUTPUT.PUT_LINE(rpt_title);
    DBMS_OUTPUT.PUT_LINE('Base Annual Salary: ' || p_base_annual);
END
/
```

EXECUTE IMMEDIATE 语句 (PL/SQL)

EXECUTE IMMEDIATE 语句根据字符串形式的 SQL 语句来准备可执行形式的语句，然后执行该 SQL 语句。EXECUTE IMMEDIATE 结合了 PREPARE 和 EXECUTE 语句的基本功能。

调用

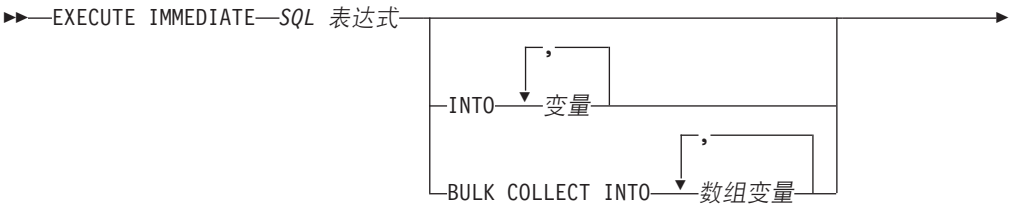
只能在 PL/SQL 上下文中指定此语句。

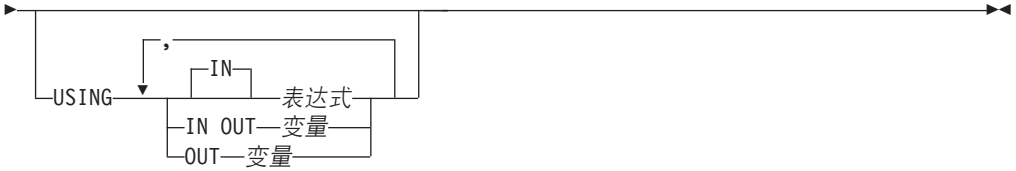
授权

授权规则就是为指定的 SQL 语句定义的那些授权规则。

语句的授权标识可能会受 DYNAMICRULES 绑定选项影响。

语法





描述

SQL 表达式

这是一个表达式，它返回所要执行的语句字符串。此表达式必须返回长度不超过最大语句大小（2097152 字节）的字符串类型。注意，CLOB(2097152) 可以包含具有最大大小的语句，但 VARCHAR 不能。

语句字符串必须是下列其中一个 SQL 语句:

- ALTER
- CALL
- COMMENT
- COMMIT
- 复合 SQL（编译型）
- 复合 SQL（直接插入型）
- CREATE
- DECLARE GLOBAL TEMPORARY TABLE
- DELETE
- DROP
- EXPLAIN
- FLUSH EVENT MONITOR
- FLUSH PACKAGE CACHE
- GRANT
- INSERT
- LOCK TABLE
- MERGE
- REFRESH TABLE
- RELEASE SAVEPOINT
- RENAME
- REVOKE
- ROLLBACK
- SAVEPOINT
- SELECT（仅当 EXECUTE IMMEDIATE 语句也指定了 BULK COLLECT INTO 子句时）
- SET COMPILATION ENVIRONMENT
- SET CURRENT DECFLOAT ROUNDING MODE
- SET CURRENT DEFAULT TRANSFORM GROUP
- SET CURRENT DEGREE

- SET CURRENT FEDERATED ASYNCHRONY
- SET CURRENT EXPLAIN MODE
- SET CURRENT EXPLAIN SNAPSHOT
- SET CURRENT IMPLICIT XMLPARSE OPTION
- SET CURRENT ISOLATION
- SET CURRENT LOCALE LC_TIME
- SET CURRENT LOCK TIMEOUT
- SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
- SET CURRENT MDC ROLLOUT MODE
- SET CURRENT OPTIMIZATION PROFILE
- SET CURRENT QUERY OPTIMIZATION
- SET CURRENT REFRESH AGE
- SET CURRENT SQL_CCFLAGS
- SET ROLE（仅当 DYNAMICRULES 运行行为对程序包生效时）
- SET ENCRYPTION PASSWORD
- SET EVENT MONITOR STATE（仅当 DYNAMICRULES 运行行为对程序包生效时）
- SET INTEGRITY
- SET PASSTHRU
- SET PATH
- SET SCHEMA
- SET SERVER OPTION
- SET SESSION AUTHORIZATION
- SET 变量
- TRANSFER OWNERSHIP（仅当 DYNAMICRULES 运行行为对程序包生效时）
- TRUNCATE（仅当 DYNAMICRULES 运行行为对程序包生效时）
- UPDATE

语句字符串不能包含语句终止符，但复合 SQL 语句除外，这些语句可以包含分号（;）来分隔复合块中的语句。复合 SQL 语句是在某些 CREATE 和 ALTER 语句中使用的，因此还可以包含分号。

执行 EXECUTE IMMEDIATE 语句时，将对指定的语句字符串进行解析并检查错误。如果该 SQL 语句无效，那么将不执行该语句，并且将抛出异常。

INTO *变量*

指定要从相应参数标记接收输出值的变量的名称。

BULK COLLECT INTO *数组变量*

标识一个或多个具有数组数据类型的变量。查询的每一行都按结果集的顺序被指定给每个数组中的元素，并按顺序指定数组下标。

- 如果只指定了一个 *数组变量*:
  - 如果 *数组变量* 元素的数据类型不是记录类型，那么 SELECT 列表必须正好包含一列，并且该列的数据类型必须可指定给数组元素数据类型。

- 如果 *数组变量* 元素的数据类型是记录类型，那么 SELECT 列表必须可指定给记录类型。

- 如果指定了多个数组变量:
  - *数组变量* 元素的数据类型不能是记录类型。
  - 对于 SELECT 列表中的每一列，都必须存在一个 *数组变量*。
  - SELECT 列表中每一列的数据类型都必须可指定给相应 *数组变量* 的数组元素数据类型。

如果 *数组变量* 的数据类型是普通数组，那么最大基数必须大于或等于查询所返回的行数。

仅当 *SQL 表达式* 是 SELECT 语句时，才能使用此子句。

USING

IN *表达式*
指定要传递给输入参数标记的值。IN 是缺省方式。

IN OUT *变量*
指定要向相应参数标记提供输入值或者从相应参数标记接收输出值的变量的名称。使用 INTO 或 BULK COLLECT INTO 子句时，不支持此选项。

OUT *变量*
指定从相应参数标记接收输出值的变量的名称。使用 INTO 或 BULK COLLECT INTO 子句时，不支持此选项。

所求值的表达式或变量的数目和顺序必须与 *SQL 表达式* 中的参数标记数目和顺序匹配，类型也必须兼容。

备注

- 语句高速缓存将影响 EXECUTE IMMEDIATE 语句的行为。

示例

```
CREATE OR REPLACE PROCEDURE proc1( p1 IN NUMBER, p2 IN OUT NUMBER, p3 OUT NUMBER )
IS
BEGIN
  p3 := p1 + 1;
  p2 := p2 + 1;
END;
/

EXECUTE IMMEDIATE 'BEGIN proc1( :1, :2, :3 ); END' USING IN p1 + 10, IN OUT p3,
  OUT p2;

EXECUTE IMMEDIATE 'BEGIN proc1( :1, :2, :3 ); END' INTO p3, p2 USING p1 + 10, p3;
```

SQL 语句 (PL/SQL)

可以使用在 PL/SQL 上下文中受支持的 SQL 语句来修改数据或者指定语句的执行方式。

表 7 列示了这些语句。这些语句在 PL/SQL 上下文中执行时的行为与相应 DB2 SQL 语句的行为相同。

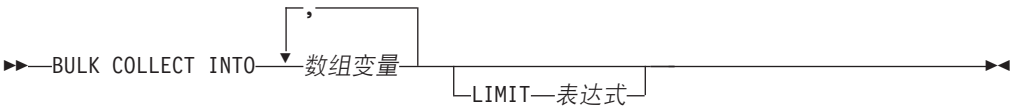
表 7. DB2 服务器可以在 PL/SQL 上下文中执行的 SQL 语句

命令	描述
DELETE	从表中删除行
INSERT	在表中插入行
MERGE	使用源（表引用的结果）中的数据来更新目标（表或视图）
SELECT INTO	从表中检索行
UPDATE	更新表中的行

BULK COLLECT INTO 子句 (PL/SQL)

在 INTO 关键字前带有可选的 BULK COLLECT 关键字的 SELECT INTO 语句将多行检索到一个数组中。

语法



描述

BULK COLLECT INTO *数组变量*

标识一个或多个具有数组数据类型的变量。结果的每一行都按结果集的顺序被指定给每个数组中的元素，并按顺序指定数组下标。

- 如果只指定了一个 *数组变量*:
  - 如果 *数组变量* 元素的数据类型不是记录类型，那么 SELECT 列表必须正好包含一列，并且该列的数据类型必须可指定给数组元素数据类型。
  - 如果 *数组变量* 元素的数据类型是记录类型，那么 SELECT 列表必须可指定给记录类型。
- 如果指定了多个数组变量:
  - *数组变量* 元素的数据类型不能是记录类型。
  - 对于 SELECT 列表中的每一列，都必须存在一个 *数组变量*。
  - SELECT 列表中每一列的数据类型都必须可指定给相应 *数组变量* 的数组元素数据类型。

如果 *数组变量* 的数据类型是普通数组，那么最大基数必须大于或等于查询所返回的行数。

LIMIT *表达式*

对要访存的行数提供上限。该表达式可以是数字文字、变量或复杂表达式，但是它不能依赖于 SELECT 语句中的任何列。

注意

- FETCH 语句和 EXECUTE IMMEDIATE 语句还支持 BULK COLLECT INTO 子句的变体。

示例

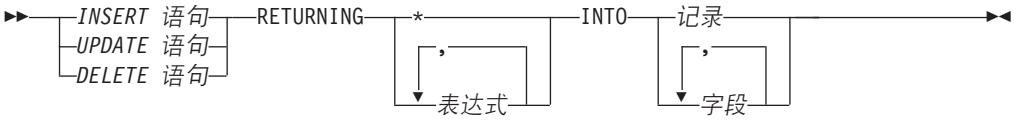
以下示例演示一个使用 BULK COLLECT INTO 子句从过程返回行数组的过程。此过程以及数组的类型在程序包中定义。

```
CREATE OR REPLACE PACAKGE bci_sample
IS
 TYPE emps_array IS VARRAY (30) OF VARCHAR2(6);

 PROCEDURE get_dept_empno (
  dno       IN   emp.deptno%TYPE,
  emps_dno  OUT  emps_array
  );
END bci_sample;

CREATE OR REPLACE PACKAGE BODY bci_sample
IS
 PROCEDURE get_dept_empno (
  dno       IN   emp.deptno%TYPE,
  emps_dno  OUT  emps_array
  )
 IS
  BEGIN
   SELECT empno BULK COLLECT INTO emps_dno LIMIT 20
    FROM emp
    WHERE deptno=dno;
  END get_dept_empno;
END bci_sample;
```

RETURNING INTO 子句 (PL/SQL)

DB2 数据服务器能够对追加了可选的 RETURNING INTO 子句的 INSERT、UPDATE 和 DELETE 语句进行编译。在 PL/SQL 上下文中使用此子句时，它将捕获分别通过执行 INSERT、UPDATE 或 DELETE 语句新添加、修改或删除的值。

语法



描述

INSERT 语句

指定有效的 INSERT 语句。如果此 INSERT 语句返回包含多行的结果集，那么将发生异常。

UPDATE 语句

指定有效的 UPDATE 语句。如果此 UPDATE 语句返回包含多行的结果集，那么将发生异常。

DELETE 语句

指定有效的 DELETE 语句。如果此 DELETE 语句返回包含多行的结果集，那么将发生异常。

RETURNING *

指定受 INSERT、UPDATE 或 DELETE 语句影响的行中所有的值都可用于赋值。

RETURNING *表达式*

指定要针对 INSERT、UPDATE 或 DELETE 语句所影响的行进行求值的表达式。求值结果将被指定给指定的记录或字段。

INTO *记录*

指定将返回的值存储到具有兼容的字段和数据类型的记录中。这些字段在数目、顺序和数据类型方面必须与 RETURNING 子句所指定的那些值匹配。如果结果集未包含任何行，那么记录中的字段将被设置为空值。

INTO *字段*

指定将返回的值存储到一组具有兼容的字段和数据类型的变量中。这些字段在数目、顺序和数据类型方面必须与 RETURNING 子句所指定的那些值匹配。如果结果集未包含任何行，那么这些字段将被设置为空值。

示例

以下示例演示一个使用了 RETURNING INTO 子句的过程:

```
CREATE OR REPLACE PROCEDURE emp_comp_update (
    p_empno         IN emp.empno%TYPE,
    p_sal           IN emp.sal%TYPE,
    p_comm          IN emp.comm%TYPE
)
IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_deptno        emp.deptno%TYPE;
BEGIN
    UPDATE emp SET sal = p_sal, comm = p_comm WHERE empno = p_empno
    RETURNING
        empno,
        ename,
        job,
        sal,
        comm,
        deptno
    INTO
        v_empno,
        v_ename,
        v_job,
        v_sal,
        v_comm,
        v_deptno;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Updated Employee # : ' || v_empno);
        DBMS_OUTPUT.PUT_LINE('Name               : ' || v_ename);
        DBMS_OUTPUT.PUT_LINE('Job                : ' || v_job);
        DBMS_OUTPUT.PUT_LINE('Department         : ' || v_deptno);
        DBMS_OUTPUT.PUT_LINE('New Salary         : ' || v_sal);
        DBMS_OUTPUT.PUT_LINE('New Commission     : ' || v_comm);
```

```
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

此过程返回以下样本输出:

```
EXEC emp_comp_update(9503, 6540, 1200);

Updated Employee # : 9503
Name               : PETERSON
Job                : ANALYST
Department         : 40
New Salary         : 6540.00
New Commission     : 1200.00
```

以下示例演示一个使用了带有记录类型的 RETURNING INTO 子句的过程:

```
CREATE OR REPLACE PROCEDURE emp_delete (
    p_empno         IN emp.empno%TYPE
)
IS
    r_emp           emp%ROWTYPE;
BEGIN
    DELETE FROM emp WHERE empno = p_empno
    RETURNING
        *
    INTO
        r_emp;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Deleted Employee # : ' || r_emp.empno);
        DBMS_OUTPUT.PUT_LINE('Name               : ' || r_emp.ename);
        DBMS_OUTPUT.PUT_LINE('Job                : ' || r_emp.job);
        DBMS_OUTPUT.PUT_LINE('Manager            : ' || r_emp.mgr);
        DBMS_OUTPUT.PUT_LINE('Hire Date          : ' || r_emp.hiredate);
        DBMS_OUTPUT.PUT_LINE('Salary             : ' || r_emp.sal);
        DBMS_OUTPUT.PUT_LINE('Commission         : ' || r_emp.comm);
        DBMS_OUTPUT.PUT_LINE('Department         : ' || r_emp.deptno);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

此过程返回以下样本输出:

```
EXEC emp_delete(9503);

Deleted Employee # : 9503
Name               : PETERSON
Job                : ANALYST
Manager            : 7902
Hire Date          : 31-MAR-05 00:00:00
Salary             : 6540.00
Commission         : 1200.00
Department         : 40
```

语句属性 (PL/SQL)

SQL%FOUND、SQL%NOTFOUND 和 SQL%ROWCOUNT 是可用于确定 SQL 语句的影响的 PL/SQL 属性。

- SQL%FOUND 属性包含一个布尔值，如果至少一行受 INSERT、UPDATE 或 DELETE 语句影响或者 SELECT INTO 语句已检索到一行，那么此属性返回 TRUE。以下示例演示了一个匿名块，在此块中，插入一行并显示状态消息。

```
BEGIN
    INSERT INTO emp (empno,ename,job,sal,deptno)
        VALUES (9001, 'JONES', 'CLERK', 850.00, 40);
    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Row has been inserted');
    END IF;
END;
```

- SQL%NOTFOUND 属性包含一个布尔值，如果没有任何行受 INSERT、UPDATE 或 DELETE 语句影响或者 SELECT INTO 语句未检索到任何行，那么此属性返回 TRUE。例如:

```
BEGIN
    UPDATE emp SET hiredate = '03-JUN-07' WHERE empno = 9000;
    IF SQL%NOTFOUND THEN
        DBMS_OUTPUT.PUT_LINE('No rows were updated');
    END IF;
END;
```

- SQL%ROWCOUNT 属性包含整数值，此值表示受 INSERT、UPDATE 或 DELETE 语句影响的行数。例如:

```
BEGIN
    UPDATE emp SET hiredate = '03-JUN-07' WHERE empno = 9001;
    DBMS_OUTPUT.PUT_LINE('# rows updated: ' || SQL%ROWCOUNT);
END;
```

第 18 章 控制语句 (PL/SQL)

控制语句是一些编程语句，它们使 PL/SQL 成为 SQL 的全面的过程式补充。

DB2 数据服务器能够编译许多 PL/SQL 控制语句。

IF 语句 (PL/SQL)

在 PL/SQL 上下文中，可以使用 IF 语句来根据特定条件执行 SQL 语句。

IF 语句的四种格式是:

- IF...THEN...END IF
- IF...THEN...ELSE...END IF
- IF...THEN...ELSE IF...END IF
- IF...THEN...ELSIF...THEN...ELSE...END IF

IF...THEN...END IF

此语句的语法是:

```
IF 布尔表达式 THEN
  语句
END IF;
```

IF...THEN 语句是 IF 的最简单格式。仅当条件求值为 TRUE 时，才会执行 THEN 与 END IF 之间的语句。在以下示例中，IF...THEN 语句用来测试和显示那些有佣金的职员。

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_comm          emp.comm%TYPE;
    CURSOR emp_cursor IS SELECT empno, comm FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    COMM');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
--
--  Test whether or not the employee gets a commission
--
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
            DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
            TO_CHAR(v_comm,'$99999.99'));
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;
```

此程序生成以下样本输出:

```
EMPNO    COMM
-----    -------
7499     $300.00
7521     $500.00
7654    $1400.00
```

IF...THEN...ELSE...END IF

此语句的语法是:

```
IF 布尔表达式 THEN
  语句
ELSE
  语句
END IF;
```

IF...THEN...ELSE 语句指定一组备用的语句，这些语句将在条件求值为 FALSE 时执行。以下示例对上一个示例作了修改，以便使用 IF...THEN...ELSE 语句在雇员没有佣金时显示文本“Non-commission”。

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_comm          emp.comm%TYPE;
    CURSOR emp_cursor IS SELECT empno, comm FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    COMM');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
--
--  Test whether or not the employee gets a commission
--
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
            DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
            TO_CHAR(v_comm,'$99999.99'));
        ELSE
            DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || 'Non-commission');
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;
```

此程序生成以下样本输出:

```
EMPNO    COMM
-----    -------
7369     Non-commission
7499  $   300.00
7521  $   500.00
7566     Non-commission
7654  $  1400.00
7698     Non-commission
7782     Non-commission
7788     Non-commission
7839     Non-commission
7844     Non-commission
7876     Non-commission
7900     Non-commission
7902     Non-commission
7934     Non-commission
```

IF...THEN...ELSE IF...END IF

此语句的语法是:

```
IF 布尔表达式 THEN
  IF 布尔表达式 THEN
    语句
ELSE
  IF 布尔表达式 THEN
    语句
END IF;
```

可以对 IF 语句进行嵌套，以便根据外层 IF 语句的条件求值为 TRUE 还是 FALSE 来调用备用 IF 语句。在以下示例中，外层 IF...THEN...ELSE 语句测试某位职员是否有佣金。内层 IF...THEN...ELSE 语句接着测试该职员的薪水总额是超出还是低于公司平均数。使用这种格式的 IF 语句时，实际上是在外层 IF 语句的 ELSE 部分中嵌套 IF 语句。因此，嵌套的每个 IF 都需要一个 END IF，并且，父 IF...ELSE 也需要一个 END IF。（注意，通过在游标声明的 SELECT 语句内使用 NVL 函数计算每位职员的年度佣金，可以大大简化此程序的逻辑；但是，本示例的用途是演示如何使用 IF 语句。）

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_avg           NUMBER(7,2);
    CURSOR emp_cursor IS SELECT empno, sal, comm FROM emp;
BEGIN
--
--  Calculate the average yearly compensation
--
    SELECT AVG((sal + NVL(comm,0)) * 24) INTO v_avg FROM emp;
    DBMS_OUTPUT.PUT_LINE('Average Yearly Compensation: ' ||
        TO_CHAR(v_avg,'$999,999.99'));
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    YEARLY COMP');
    DBMS_OUTPUT.PUT_LINE('-----    -----------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_sal, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
--
--  Test whether or not the employee gets a commission
--
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
--
--  Test whether the employee's compensation with commission exceeds
--  the company average
--
            IF (v_sal + v_comm) * 24 > v_avg THEN
                DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                    TO_CHAR((v_sal + v_comm) * 24,'$999,999.99') ||
                    ' Exceeds Average');
            ELSE
                DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                    TO_CHAR((v_sal + v_comm) * 24,'$999,999.99') ||
                    ' Below Average');
            END IF;
        ELSE
--
--  Test whether the employee's compensation without commission exceeds
--  the company average
--
            IF v_sal * 24 > v_avg THEN
                DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                    TO_CHAR(v_sal * 24,'$999,999.99') || ' Exceeds Average');
            ELSE
```

```
                DBMS_OUTPUT.PUT_LINE(v_empno || '   ' ||
                    TO_CHAR(v_sal * 24,'$999,999.99') || ' Below Average');
            END IF;
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;
```

此程序生成以下样本输出:

```
Average Yearly Compensation: $  53,528.57
EMPNO    YEARLY COMP
-----    -----------
7369  $  19,200.00 Below Average
7499  $  45,600.00 Below Average
7521  $  42,000.00 Below Average
7566  $  71,400.00 Exceeds Average
7654  $  63,600.00 Exceeds Average
7698  $  68,400.00 Exceeds Average
7782  $  58,800.00 Exceeds Average
7788  $  72,000.00 Exceeds Average
7839  $ 120,000.00 Exceeds Average
7844  $  36,000.00 Below Average
7876  $  26,400.00 Below Average
7900  $  22,800.00 Below Average
7902  $  72,000.00 Exceeds Average
7934  $  31,200.00 Below Average
```

IF...THEN...ELSIF...THEN...ELSE...END IF

此语句的语法是:

```
IF 布尔表达式 THEN
  语句
[ ELSIF 布尔表达式 THEN
  语句
[ ELSIF 布尔表达式 THEN
  语句 ] ...]
[ ELSE
  语句 ]
END IF;
```

IF...THEN...ELSIF...ELSE 语句提供了在一个语句中检查多种可能情况的方法。在形式上，此语句相当于嵌套的 IF...THEN...ELSE...IF...THEN 语句，但只需要一个 END IF。以下示例使用 IF...THEN...ELSIF...ELSE 语句按步长 $25,000 来计算各佣金范围的职员数。

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_comp          NUMBER(8,2);
    v_lt_25K        SMALLINT := 0;
    v_25K_50K       SMALLINT := 0;
    v_50K_75K       SMALLINT := 0;
    v_75K_100K      SMALLINT := 0;
    v_ge_100K       SMALLINT := 0;
    CURSOR emp_cursor IS SELECT empno, (sal + NVL(comm,0)) * 24 FROM emp;
BEGIN
    OPEN emp_cursor;
    LOOP
        FETCH emp_cursor INTO v_empno, v_comp;
        EXIT WHEN emp_cursor%NOTFOUND;
        IF v_comp < 25000 THEN
            v_lt_25K := v_lt_25K + 1;
        ELSIF v_comp < 50000 THEN
            v_25K_50K := v_25K_50K + 1;
        ELSIF v_comp < 75000 THEN
            v_50K_75K := v_50K_75K + 1;
```

```
        ELSIF v_comp < 100000 THEN
            v_75K_100K := v_75K_100K + 1;
        ELSE
            v_ge_100K := v_ge_100K + 1;
        END IF;
    END LOOP;
    CLOSE emp_cursor;
    DBMS_OUTPUT.PUT_LINE('Number of employees by yearly compensation');
    DBMS_OUTPUT.PUT_LINE('Less than 25,000 : ' || v_lt_25K);
    DBMS_OUTPUT.PUT_LINE('25,000 - 49,9999 : ' || v_25K_50K);
    DBMS_OUTPUT.PUT_LINE('50,000 - 74,9999 : ' || v_50K_75K);
    DBMS_OUTPUT.PUT_LINE('75,000 - 99,9999 : ' || v_75K_100K);
    DBMS_OUTPUT.PUT_LINE('100,000 and over : ' || v_ge_100K);
END;
```

此程序生成以下样本输出:

```
Number of employees by yearly compensation
Less than 25,000 : 2
25,000 - 49,9999 : 5
50,000 - 74,9999 : 6
75,000 - 99,9999 : 0
100,000 and over : 1
```

CASE 语句 (PL/SQL)

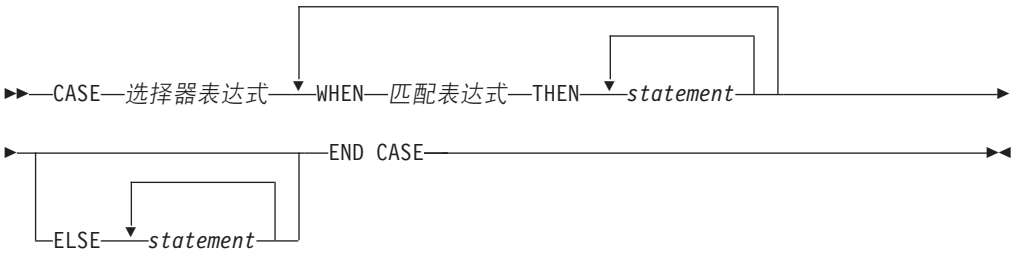
CASE 语句在指定的搜索条件为 true 时执行一条或多条语句。CASE 是独立的语句，它与必须作为表达式组成部分出现的 CASE 表达式不同。

CASE 语句有两种形式: 简单 CASE 语句和搜索型 CASE 语句。

简单 CASE 语句 (PL/SQL)

简单 CASE 语句尝试使表达式（被称为*选择器*）与一个或多个 WHEN 子句中指定的另一表达式匹配。发生匹配将导致执行一条或多条相应的语句。

语法



描述

CASE *选择器表达式*

指定一个表达式，此表达式的值的数据类型与每个 *匹配表达式* 兼容。如果 *选择器表达式* 的值与第一个 *匹配表达式* 匹配，那么将执行相应 THEN 子句中的语句。如果没有匹配项，那么将执行相应 ELSE 子句中的语句。如果没有匹配项，并且没有 ELSE 子句，那么将抛出异常。

WHEN *匹配表达式*

指定要在 CASE 语句中进行求值的表达式。如果 *选择器表达式* 与某个 *匹配表达式* 匹配，那么将执行相应 THEN 子句中的语句。

THEN

此关键字引入要在相应布尔表达式求值为 TRUE 时执行的语句。

statement

指定一个或多个 SQL 或 PL/SQL 语句，每个语句都以分号终止。

ELSE

此关键字引入 CASE 语句的缺省情况。

示例

以下示例使用简单 CASE 语句将部门名称和地点指定给基于部门编号的变量。

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_loc           dept.loc%TYPE;
    CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME     DEPTNO    DNAME     '
        || '     LOC');
    DBMS_OUTPUT.PUT_LINE('-----    -------   ------    ----------'
        || '     ---------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
        EXIT WHEN emp_cursor%NOTFOUND;
        CASE v_deptno
            WHEN 10 THEN v_dname := 'Accounting';
                         v_loc   := 'New York';
            WHEN 20 THEN v_dname := 'Research';
                         v_loc   := 'Dallas';
            WHEN 30 THEN v_dname := 'Sales';
                         v_loc   := 'Chicago';
            WHEN 40 THEN v_dname := 'Operations';
                         v_loc   := 'Boston';
            ELSE v_dname := 'unknown';
                         v_loc   := '';
        END CASE;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) ||
            '  ' || v_deptno || '      ' || RPAD(v_dname, 14) || ' ' ||
            v_loc);
    END LOOP;
    CLOSE emp_cursor;
END;
```

此程序返回以下样本输出:

```
EMPNO    ENAME     DEPTNO    DNAME          LOC
-----    -------   ------    ----------     ---------
7369     SMITH       20      Research       Dallas
7499     ALLEN       30      Sales          Chicago
7521     WARD        30      Sales          Chicago
7566     JONES       20      Research       Dallas
7654     MARTIN      30      Sales          Chicago
7698     BLAKE       30      Sales          Chicago
7782     CLARK       10      Accounting     New York
7788     SCOTT       20      Research       Dallas
7839     KING        10      Accounting     New York
```

```
7844     TURNER      30      Sales          Chicago
7876     ADAMS       20      Research       Dallas
7900     JAMES       30      Sales          Chicago
7902     FORD        20      Research       Dallas
7934     MILLER      10      Accounting     New York
```

搜索型 CASE 语句 (PL/SQL)

搜索型 CASE 语句使用一个或多个布尔表达式来确定所要执行的语句。

语法

描述

CASE

此关键字引入 CASE 语句中的第一个 WHEN 子句。

WHEN *布尔表达式*

指定一个表达式，当控制流进入定义此表达式的 WHEN 子句时，将对此表达式进行求值。如果 *布尔表达式* 求值为 TRUE，那么将执行相应 THEN 子句中的语句。如果 *布尔表达式* 未求值为 TRUE，那么将执行相应 ELSE 子句中的语句。

THEN

此关键字引入要在相应布尔表达式求值为 TRUE 时执行的语句。

statement

指定一个或多个 SQL 或 PL/SQL 语句，每个语句都以分号终止。

ELSE

此关键字引入 CASE 语句的缺省情况。

示例

以下示例使用搜索型 CASE 语句将部门名称和地点指定给基于部门编号的变量。

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_loc           dept.loc%TYPE;
    CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME     DEPTNO    DNAME     '
        || '     LOC');
    DBMS_OUTPUT.PUT_LINE('-----    -------   ------    ----------'
        || '     ---------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
        EXIT WHEN emp_cursor%NOTFOUND;
        CASE
            WHEN v_deptno = 10 THEN v_dname := 'Accounting';
                                    v_loc   := 'New York';
            WHEN v_deptno = 20 THEN v_dname := 'Research';
                                    v_loc   := 'Dallas';
            WHEN v_deptno = 30 THEN v_dname := 'Sales';
```

```
                                v_loc   := 'Chicago';
            WHEN v_deptno = 40 THEN v_dname := 'Operations';
                                v_loc   := 'Boston';
            ELSE v_dname := 'unknown';
                                v_loc   := '';
        END CASE;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) ||
            '  ' || v_deptno || '      ' || RPAD(v_dname, 14) || ' ' ||
            v_loc);
    END LOOP;
    CLOSE emp_cursor;
END;
```

此程序返回以下样本输出:

```
EMPNO     ENAME      DEPTNO     DNAME          LOC
-----     -------    ------     ----------     ---------
7369      SMITH        20       Research       Dallas
7499      ALLEN        30       Sales          Chicago
7521      WARD         30       Sales          Chicago
7566      JONES        20       Research       Dallas
7654      MARTIN       30       Sales          Chicago
7698      BLAKE        30       Sales          Chicago
7782      CLARK        10       Accounting     New York
7788      SCOTT        20       Research       Dallas
7839      KING         10       Accounting     New York
7844      TURNER       30       Sales          Chicago
7876      ADAMS        20       Research       Dallas
7900      JAMES        30       Sales          Chicago
7902      FORD         20       Research       Dallas
7934      MILLER       10       Accounting     New York
```

循环 (PL/SQL)

使用 EXIT、FOR、LOOP 和 WHILE 语句在 PL/SQL 程序中反复执行一系列命令。

FOR（游标变体）语句 (PL/SQL)

游标 FOR 循环语句打开先前声明的游标，访存游标结果集中所有的行，然后关闭该游标。

使用此语句来代替多个不同的 SQL 语句，以便打开游标、定义循环构造以检索结果集中的每一行、测试结果集是否结束并最终关闭游标。

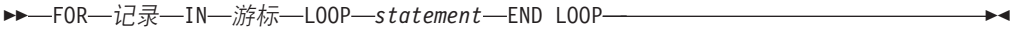
调用

可以从 PL/SQL 过程、函数、触发器或匿名块中调用此语句。

授权

不需要特定的权限即可在 SQL 语句中引用行表达式；但是，要成功地执行语句，需要处理游标所需的所有其他权限。

语法



描述

FOR

引入一个条件，要使 FOR 循环继续进行，此条件必须为 true。

记录

指定对隐式声明的记录（其定义为“游标%ROWTYPE”）指定的标识。

IN *游标*

指定先前声明的游标的名称。

LOOP 和 END LOOP

开始和结束循环，此循环包含要在循环的每次迭代期间执行的 SQL 语句。

statement

一个或多个 PL/SQL 语句。至少需要一个语句。

示例

以下示例演示包含游标 FOR 循环的过程:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
  CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
  DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
  DBMS_OUTPUT.PUT_LINE('-----    -------');
  FOR v_emp_rec IN emp_cur_1 LOOP
    DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || '     ' || v_emp_rec.ename);
  END LOOP;
END;
```

FOR（整数变体）语句 (PL/SQL)

使用 FOR 语句来多次执行一组 SQL 语句。

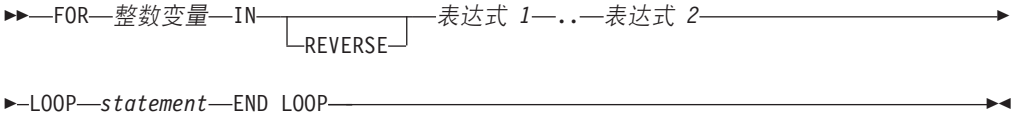
调用

可以在 PL/SQL 过程、函数或匿名块语句中嵌入此语句。

授权

调用 FOR 语句不需要任何特权；但是，此语句的授权标识必须拥有调用 FOR 语句中嵌入的 SQL 语句所必需的特权。

语法



描述

整数变量

自动定义的整数变量，在循环处理期间，将使用此变量。整数变量 的初始值是 表

达式 1。在初始迭代之后，*整数变量* 的值将在后续每次迭代开始时递增。进入循环时，将对 *表达式 1* 和 *表达式 2* 进行求值，当 *整数变量* 等于 *表达式 2* 时，循环处理停止。

IN 引入可选的 REVERSE 关键字和表达式，它们定义循环的整数变量的范围。

REVERSE
指定迭代方向是从 *表达式 2* 到 *表达式 1*。注意，无论是否指定了 REVERSE 关键字，*表达式 2* 的值都必须大于 *表达式 1*，这样才会处理循环中的语句。

表达式 1
指定循环的整数变量范围的初始值。如果指定了 REVERSE 关键字，那么*表达式 1* 指定循环的整数变量范围的结束值。

表达式 2
指定循环的整数变量范围的结束值。如果指定了 REVERSE 关键字，那么*表达式 2* 指定循环的整数变量范围的初始值。

statement
指定每次处理循环时执行的 PL/SQL 和 SQL 语句。

示例

以下示例演示匿名块中的基本 FOR 语句:

```
BEGIN
  FOR i IN 1 .. 10 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
  END LOOP;
END;
```

此示例生成以下输出:

```
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

如果起始值大于结束值，那么完全不执行循环体，但不会返回错误，如以下示例所示:

```
BEGIN
  FOR i IN 10 .. 1 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
  END LOOP;
END;
```

此示例不生成输出，这是因为，永远不会执行循环体。

以下示例使用 REVERSE 关键字:

```
BEGIN
  FOR i IN REVERSE 1 .. 10 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
  END LOOP;
END;
```

此示例生成以下输出:

```
Iteration # 10
Iteration # 9
Iteration # 8
Iteration # 7
Iteration # 6
Iteration # 5
Iteration # 4
Iteration # 3
Iteration # 2
Iteration # 1
```

FORALL 语句 (PL/SQL)

FORALL 语句对数组的所有元素或数组中某个范围的元素执行数据更改语句。

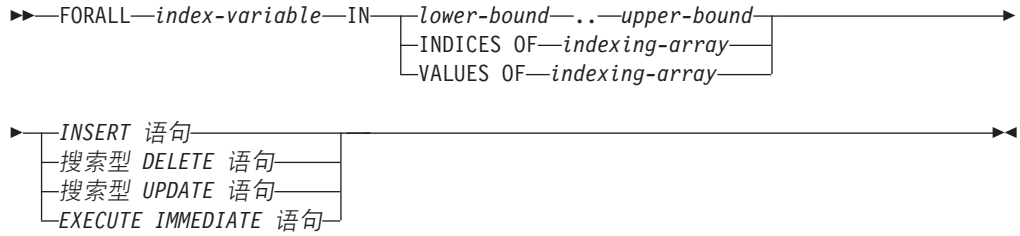
调用

只能在 PL/SQL 块中指定此语句。

授权

此语句的授权标识所拥有的特权必须包括调用 FORALL 语句中指定的数据更改语句所需的所有特权。

语法



描述

index-variable

标识用作数组下标的名称。它被隐式地声明为 INTEGER，并且只能在 FORALL 语句中引用。

lower-bound **..** *upper-bound*

标识可以指定给 *index-variable* 的下标值的范围，并且 *lower-bound* 小于 *upper-bound*。范围表示从 *lower-bound* 开始每次递增 1 并直到 *upper-bound* 为止（包括上限）的每个整数值。

INDICES OF *indexing-array*

标识 *indexing-array* 所标识的数组的数组下标值集合。如果 *indexing-array* 是关联数组，那么数组下标值必须可指定给 *index-variable* 并且不能是稀疏集合。

VALUES OF *indexing-array*

标识 *indexing-array* 所标识的数组的元素值集合。元素值必须可指定给 *index-variable*，并且不能是无序稀疏集合。

INSERT 语句

指定要对每个 *index-variable* 值执行的 INSERT 语句。

搜索型 DELETE 语句
指定要对每个 *index-variable* 值执行的搜索型 DELETE 语句。

搜索型 UPDATE 语句
指定要对每个 *index-variable* 值执行的搜索型 UPDATE 语句。

EXECUTE IMMEDIATE 语句
指定要对每个 *index-variable* 值执行的 EXECUTE IMMEDIATE 语句。

注意

- FORALL 语句处理不是原子处理。如果在 FORALL 语句中进行迭代时发生错误，那么不会以隐式方式回滚已处理的任何数据更改操作。当 FORALL 语句发生错误时，应用程序可以使用 ROLLBACK 语句来回滚整个事务。

示例

以下示例演示基本 FORALL 语句:

```
FORALL x
  IN in_customer_list.FIRST..in_customer_list.LAST
  DELETE FROM customer
    WHERE cust_id IN in_customer_list(x);
```

EXIT 语句 (PL/SQL)

EXIT 语句用于终止执行 PL/SQL 代码块中的循环。

调用

可以在 PL/SQL 过程、函数或匿名块中的 FOR、LOOP 或 WHILE 语句中嵌入此语句。

授权

调用 EXIT 语句不需要任何特权。但是，此语句的授权标识必须拥有调用 FOR、LOOP 或 WHILE 语句中嵌入的 SQL 语句所必需的特权。

语法

示例

以下示例演示匿名块中包含 EXIT 语句的基本 LOOP 语句:

```
DECLARE
  sum PLS_INTEGER := 0;
BEGIN
  LOOP
    sum := sum + 1;
    IF sum > 10 THEN
       EXIT;
    END IF;
  END LOOP;
END
```

LOOP 语句 (PL/SQL)

LOOP 语句在 PL/SQL 代码块中多次执行一系列语句。

调用

可以在 PL/SQL 过程、函数或匿名块语句中嵌入此语句。

授权

调用 LOOP 语句不需要任何特权。但是，此语句的授权标识必须拥有调用 LOOP 语句中嵌入的 SQL 语句所必需的特权。

语法

描述

statement
指定一个或多个 PL/SQL 或 SQL 语句。在此循环的每次迭代期间，都将执行这些语句。

示例

以下示例演示匿名块中的基本 LOOP 语句:

```
DECLARE
  sum INTEGER := 0;
BEGIN
  LOOP
    sum := sum + 1;
    IF sum > 10 THEN
       EXIT;
    END IF;
  END LOOP;
END
```

WHILE 语句 (PL/SQL)

WHILE 语句在指定的表达式为 true 时反复执行一组 SQL 语句。每次进入循环体时，都将立即对此条件进行求值。

调用

可以在 PL/SQL 过程、函数或匿名块语句中嵌入此语句。

授权

调用 WHILE 语句不需要任何特权；但是，此语句的授权标识必须拥有调用 WHILE 语句中嵌入的 SQL 语句所必需的特权。

语法

描述

表达式
指定一个表达式，在每次进入循环体之前，都将立即对此表达式进行求值以确定是

否执行该循环。如果此表达式在逻辑上为 true，那么将执行该循环。如果此表达式在逻辑上为 false，那么循环处理结束。可以使用 EXIT 语句在此表达式为 true 时终止循环。

statement
指定每次处理循环时执行的 PL/SQL 和 SQL 语句。

示例

以下示例演示匿名块中的基本 WHILE 语句:

```
DECLARE
  sum INTEGER := 0;
BEGIN
  WHILE sum < 11 LOOP
    sum := sum + 1;
  END LOOP;
END
```

此匿名块中的 WHILE 语句将执行到 *sum* 等于 11 为止；然后，循环处理结束，匿名块的处理将进行到完成。

CONTINUE 语句 (PL/SQL)

CONTINUE 语句终止 PL/SQL 代码块内某个循环的当前迭代，并且移至该循环的下一迭代。

调用

可以在 FOR、LOOP 或 WHILE 语句或者 PL/SQL 过程、函数或匿名块语句中嵌入此语句。

授权

调用 CONTINUE 语句不需要任何特权。但是，此语句的授权标识必须拥有调用 FOR、LOOP 或 WHILE 语句中嵌入的 SQL 语句所必需的特权。

语法

示例

以下示例显示匿名块中包含 CONTINUE 语句的基本 LOOP 语句:

```
BEGIN
  FOR i IN 1 .. 5 LOOP
    IF i = 3 THEN
       CONTINUE;
    END IF;
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
  END LOOP;
END;
```

此示例生成以下输出:

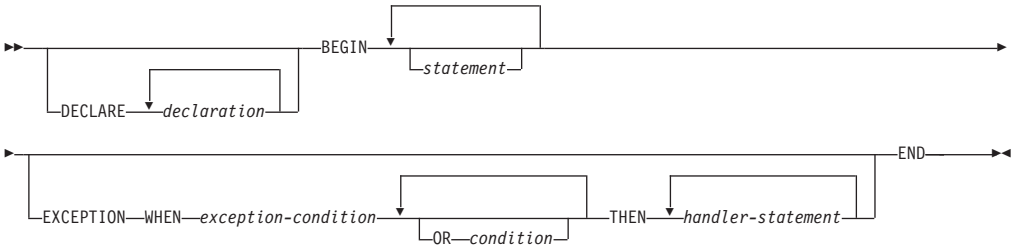
```
Iteration # 1
Iteration # 2
Iteration # 4
Iteration # 5
```

异常处理 (PL/SQL)

缺省情况下，在 PL/SQL 程序中遇到的任何错误都将停止执行程序。您可以使用 EXCEPTION 节来捕获错误以及从出错状态恢复。

异常处理程序的语法是 BEGIN 块的语法扩展。

语法



如果未发生错误，那么此块仅仅执行 *statement* 并将控制权传递到 END 之后的语句。但是，如果在执行 *statement* 时发生错误，那么会放弃对 *statement* 的进一步处理，并且控制权会传递到 EXCEPTION 部分。会在 WHEN 子句中搜索与所发生错误相匹配的第一个异常。如果找到匹配项，那么将执行相应的*处理语句* 并将控制权传递到 END 之后的语句。如果找不到匹配项，那么程序将停止执行。

如果执行 *处理语句* 时发生新的错误，那么它只能由外层的 EXCEPTION 子句捕获。

WHEN 子句中的异常可以是用户定义的，也可以是内置的。可在当前块或其外层块的 DECLARE 部分中定义用户定义的异常，也可在 PL/SQL 程序包的 DECLARE 部分中定义用户定义的异常。可在异常定义后面直接使用语法 PRAGMA EXCEPTION_INIT 或 PRAGMA DB2_EXCEPTION_INIT，从而指定与用户定义的异常对应的 Oracle sqlcode 或 DB2 sqlstate。

在以下示例中，DECLARE 部分包含三个指定异常的定义。块的主体是对过程 MyApp.Main 的调用。EXCEPTION 部分包含用于下列三个异常的处理程序:

1. exception1，与 Oracle sqlcode 或 DB2 sqlstate 不相关联。
2. exception2，与 Oracle sqlcode -942（未定义名称）相关联。
3. exception3，与 DB2 sqlstate 42601（语法错误）相关联。

```
DECLARE
  exception1 EXCEPTION;
  exception2 EXCEPTION;
  PRAGMA EXCEPTION_INIT(exception2,-942);
  exception3 EXCEPTION;
  PRAGMA DB2_EXCEPTION_INIT(exception3,'42601');
BEGIN
  MyApp.Main(100);
EXCEPTION
  WHEN exception1 THEN
```

```
      DBMS_OUTPUT.PUT_LINE('User-defined exception1 caught');
  WHEN exception2 THEN
      DBMS_OUTPUT.PUT_LINE('User-defined exception2 (Undefined name) caught');
  WHEN exception3 THEN
      DBMS_OUTPUT.PUT_LINE('User-defined exception3 (Syntax error) caught');
END
```

注: DB2 会接受有限数目的 Oracle sqlcode 作为 PRAGMA EXCEPTION_INIT 的自变量。请参阅 第 180 页的『Oracle-DB2 错误映射 (PL/SQL)』，以获取完整列表。

当捕获了使用 PRAGMA EXCEPTION_INIT 初始化的异常时，由 SQLCODE 函数返回的值是与该异常相关联的 DB2 sqlcode，而不是 Oracle 值。在以上示例中，当捕获了 exception2 时，由 SQLCODE 返回的值将为 -204，它是与 Oracle sqlcode -942 对应的 DB2 sqlcode。如果 Oracle-DB2 错误映射表中未列示在 PRAGMA EXCEPTION_INIT 中指定的 Oracle sqlcode，那么编译失败。可通过执行下列操作来避免发生此情况：将 PRAGMA EXCEPTION_INIT 替换为 PRAGMA DB2_EXCEPTION_INIT，并且指定与要识别的错误对应的 DB2 sqlstate。

表 8 对可使用的内置异常进行了概括。特殊异常名称 OTHERS 与每个异常都相匹配。条件名不区分大小写。

表 8. 内置异常名称

异常名称	描述
CASE_NOT_FOUND	在 CASE 语句中，没有任何情况求值为“true”，并且没有 ELSE 条件。
CURSOR_ALREADY_OPEN	试图打开已打开的游标。
DUP_VAL_ON_INDEX	索引键有重复的值。
INVALID_CURSOR	试图访问未打开的游标。
INVALID_NUMBER	数字值无效。
LOGIN_DENIED	用户名或密码无效。
NO_DATA_FOUND	没有满足选择标准的行。
NOT_LOGGED_ON	不存在数据库连接。
OTHERS	表示任何未被异常节中的先前条件捕获的异常。
SUBSCRIPT_BEYOND_COUNT	数组下标超出范围或不存在。
SUBSCRIPT_OUTSIDE_LIMIT	数组下标表达式的数据类型无法指定给数组下标类型。
TOO_MANY_ROWS	多行满足选择标准，但只允许返回一行。
VALUE_ERROR	值无效。
ZERO_DIVIDE	试图除零。

发出应用程序错误 (PL/SQL)

RAISE_APPLICATION_ERROR 过程根据由用户提供的错误代码和消息发出异常。此过程只有在 PL/SQL 上下文中才受支持。

语法

描述

错误号

特定于供应商的编号，在将此编号存储在名为 SQLCODE 的变量中之前，将它映射到 DB2 错误代码。RAISE_APPLICATION_ERROR 过程接受用户定义的 *错误号* 值 -20000 到 -20999。在错误消息中返回的 SQLCODE 是 SQL0438N。SQLSTATE 包含类“UD”以及三个字符，这三个字符与 *错误号* 值的最后三个数字相对应。

消息

用户定义的消息，其最大长度为 70 个字节。

keeperrorstack

用于指示是否应该保留错误堆栈的可选布尔值。当前，仅支持缺省值 *false*。

示例

以下示例使用 RAISE_APPLICATION_ERROR 过程来显示特定于“缺少职员信息”这种情况的错误代码和消息:

```
CREATE OR REPLACE PROCEDURE verify_emp (
    p_empno         NUMBER
)
IS
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_mgr           emp.mgr%TYPE;
    v_hiredate      emp.hiredate%TYPE;
BEGIN
    SELECT ename, job, mgr, hiredate
        INTO v_ename, v_job, v_mgr, v_hiredate FROM emp
        WHERE empno = p_empno;
    IF v_ename IS NULL THEN
        RAISE_APPLICATION_ERROR(-20010, 'No name for ' || p_empno);
    END IF;
    IF v_job IS NULL THEN
        RAISE_APPLICATION_ERROR(-20020, 'No job for' || p_empno);
    END IF;
    IF v_mgr IS NULL THEN
        RAISE_APPLICATION_ERROR(-20030, 'No manager for ' || p_empno);
    END IF;
    IF v_hiredate IS NULL THEN
        RAISE_APPLICATION_ERROR(-20040, 'No hire date for ' || p_empno);
    END IF;
    DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno ||
        ' validated without errors');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
END;

CALL verify_emp(7839);

SQLCODE: -438
SQLERRM: SQL0438N  Application raised error or warning with
 diagnostic text: "No manager for 7839".  SQLSTATE=UD030
```

RAISE 语句 (PL/SQL)

RAISE 语句发出先前定义的异常。

语法

描述

异常

指定先前定义的异常。

示例

以下示例显示根据调用中作为自变量提供的值发出 oddno 或 evenno 异常的过程:

```
CREATE OR REPLACE PROCEDURE raise_demo (inval NUMBER) IS
  evenno EXCEPTION;
  oddno  EXCEPTION;
BEGIN
  IF MOD(inval, 2) = 1 THEN
    RAISE oddno;
  ELSE
    RAISE evenno;
  END IF;
EXCEPTION
  WHEN evenno THEN
    dbms_output.put_line(TO_CHAR(inval) || ' is even');
  WHEN oddno THEN
    dbms_output.put_line(TO_CHAR(inval) || ' is odd');
END raise_demo;
/

SET SERVEROUTPUT ON
/

CALL raise_demo (11)
/
```

CALL 语句的输出将如下:

```
CALL raise_demo (11)

 Return Status = 0

11 is odd
```

Oracle-DB2 错误映射 (PL/SQL)

PL/SQL 错误代码和异常名称有相应的 DB2 错误代码和 SQLSTATE 值。

表 9 对这些错误代码、异常名称和 SQLSTATE 值进行摘要说明。

表 9. PL/SQL 错误代码和异常名称与 DB2 错误代码和 SQLSTATE 值的映射

plsqlCode	plsqlName	db2Code	db2State
-1	DUP_VAL_ON_INDEX	-803	23505
+100	NO_DATA_FOUND	+100	02000

表 9. PL/SQL 错误代码和异常名称与 DB2 错误代码和 SQLSTATE 值的映射 (续)

plsqlCode	plsqlName	db2Code	db2State
-1012	NOT_LOGGED_ON	-1024	08003
-1017	LOGIN_DENIED	-30082	08001
-1476	ZERO_DIVIDE	-801	22012
-1722	INVALID_NUMBER	-420	22018
-1001	INVALID_CURSOR	-501	24501
-1422	TOO_MANY_ROWS	-811	21000
-6502	VALUE_ERROR	-433	22001
-6511	CURSOR_ALREADY_OPEN	-502	24502
-6532	SUBSCRIPT_OUTSIDE_LIMIT	-20439	428H1
-6533	SUBSCRIPT_BEYOND_COUNT	-20439	2202E
-6592	CASE_NOT_FOUND	-773	20000
-54		-904	57011
-60		-911	40001
-310		-206	42703
-595		-390	42887
-597		-303	42806
-598		-407	23502
-600		-30071	58015
-603		-119	42803
-604		-119	42803
-610		-20500	428HR
-611		-117	42802
-612		-117	42802
-613		-811	21000
-615		-420	22018
-616		-420	22018
-617		-418	42610
-618		-420	22018
-619		-418	42610
-620		-171	42815
-622		-304	22003
-623		-604	42611
-904		-206	42703
-911		-7	42601
-942		-204	42704
-955		-601	42710
-996		-1022	57011
-1119		-292	57047
-1002		+231	02000
-1403		-100	02000

表 9. PL/SQL 错误代码和异常名称与 DB2 错误代码和 SQLSTATE 值的映射 (续)

plsqlCode	plsqlName	db2Code	db2State
-1430		-612	42711
-1436		-20451	560CO
-1438		-413	22003
-1450		-614	54008
-1578		-1007	58034
-2112		-811	21000
-2261		+605	01550
-2291		-530	23503
-2292		-532	23001
-3113		-30081	08001
-3114		-1024	08003
-3214		-20170	57059
-3297		-20170	57059
-4061		-727	56098
-4063		-727	56098
-4091		-723	09000
-6502		-304	22003
-6508		-440	42884
-6550		-104	42601
-6553		-104	42601
-14028		-538	42830
-19567		-1523	55039
-30006		-904	57011
-30041		-1139	54047

第 19 章 游标 (PL/SQL)

*游标*是由应用程序使用并具有名称的控制结构，用于指向并选择结果集中的数据行。您可以使用游标以每次一行的方式来读取并处理查询结果集，而不是立即执行整个查询。

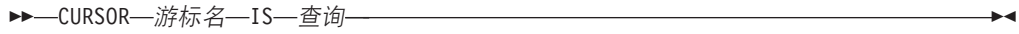
PL/SQL 上下文中的游标被视为 WITH HOLD 游标。有关 WITH HOLD 游标的更多信息，请参阅“DECLARE CURSOR 语句”。

DB2 数据服务器既支持 PL/SQL 静态游标也支持游标变量。

静态游标 (PL/SQL)

*静态游标*的相关联查询在编译时固定。声明游标是使用该游标的先决条件。DB2 服务器支持在 PL/SQL 上下文中使用 PL/SQL 语法来声明静态游标。

语法



描述

游标名
指定游标的标识，此标识可用于引用该游标及其结果集。

查询
指定一个 SELECT 语句，此语句确定游标的结果集。

示例

以下示例演示包含多个静态游标声明的过程:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
     CURSOR emp_cur_1 IS SELECT * FROM emp;

     CURSOR emp_cur_2 IS SELECT empno, ename FROM emp;

     CURSOR emp_cur_3 IS SELECT empno, ename
                         FROM emp
                         WHERE deptno = 10
                         ORDER BY empno;
BEGIN
    OPEN emp_cur_1;
        ...
END;
```

参数化游标 (PL/SQL)

参数化游标是指在打开时能够接受传入的参数值的静态游标。

以下示例包含一个参数化游标。此游标显示 EMP 表中每个薪水低于传入参数值所指定值的职员的姓名和薪水。

```
DECLARE
    my_record       emp%ROWTYPE;
    CURSOR c1 (max_wage NUMBER) IS
        SELECT * FROM emp WHERE sal < max_wage;
BEGIN
    OPEN c1(2000);
    LOOP
        FETCH c1 INTO my_record;
        EXIT WHEN c1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Name = ' || my_record.ename || ', salary = '
            || my_record.sal);
    END LOOP;
    CLOSE c1;
END;
```

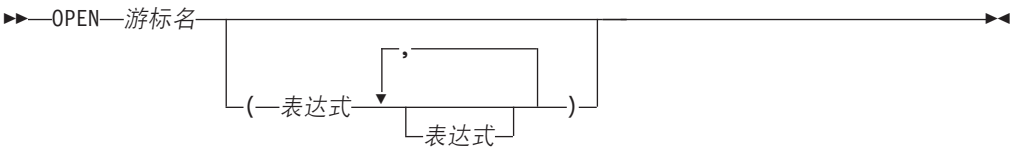
如果传递 2000 作为 *max_wage* 的值，那么将只返回那些薪水低于 2000 的职员的姓名和薪水数据:

```
Name = SMITH, salary = 800.00
Name = ALLEN, salary = 1600.00
Name = WARD, salary = 1250.00
Name = MARTIN, salary = 1250.00
Name = TURNER, salary = 1500.00
Name = ADAMS, salary = 1100.00
Name = JAMES, salary = 950.00
Name = MILLER, salary = 1300.00
```

打开游标 (PL/SQL)

直到打开游标之后，才能引用与该游标相关联的结果集。

语法



描述

游标名

指定 PL/SQL 上下文中先前声明的游标的标识。指定的游标不能已打开。

表达式

当 *游标名* 是参数化的游标时，请指定一个或多个可选的实参。实参数必须与相应形参的数目匹配。

示例

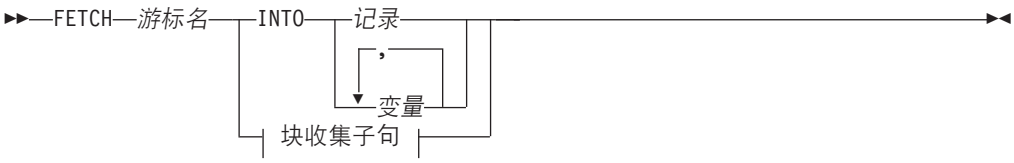
以下示例演示对 CURSOR_EXAMPLE 过程中的游标执行 OPEN 语句:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    CURSOR emp_cur_3 IS SELECT empno, ename
                        FROM emp
                        WHERE deptno = 10
                        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
        ...
END;
```

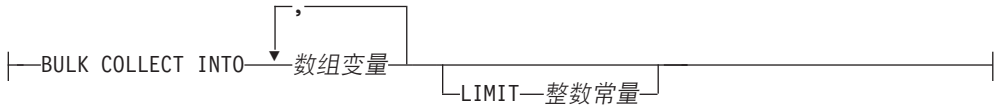
从游标访存行 (PL/SQL)

在 PL/SQL 上下文中，DB2 数据服务器支持用于从 PL/SQL 游标访存行的 FETCH 语句。

语法



块收集子句:



描述

游标名

静态游标或游标变量的名称。

记录

先前定义的记录的标识。这可以是用户定义的记录，也可以是使用 %ROWTYPE 属性从表派生的记录定义。

变量

用于存放所访存行中的字段数据的 PL/SQL 变量。您可以定义一个或多个变量，但是，它们在顺序和数目方面必须与游标声明所指定查询的选择列表中返回的字段匹配。选择列表中字段的数据类型与记录中字段的数据类型或者变量的数据类型必须匹配或可隐式地转换。

您可以显式地定义变量数据类型或者使用 %TYPE 属性来定义变量数据类型。

BULK COLLECT INTO *数组变量*

标识一个或多个具有数组数据类型的变量。结果的每一行都按结果集的顺序被指定给每个数组中的元素，并按顺序指定数组下标。

- 如果只指定了一个 *数组变量*:
  - 如果 *数组变量* 元素的数据类型不是记录类型，那么游标的结果行必须正好包含一列，并且该列的数据类型必须可指定给数组元素数据类型。
  - 如果 *数组变量* 元素的数据类型是记录类型，那么游标的结果行必须可指定给记录类型。
- 如果指定了多个数组变量:
  - *数组变量* 元素的数据类型不能是记录类型。
  - 对于游标的结果行中的每一列，都必须存在一个 *数组变量*。
  - 游标的结果行中每一列的数据类型都必须可指定给相应 *数组变量* 的数组元素数据类型。

如果 *数组变量* 的数据类型是普通数组，那么最大基数必须大于或等于查询所返回的行数，或者大于或等于 LIMIT 子句中指定的 *整数常量*。

LIMIT *整数常量*

指定存储在目标数组中的行的数目限制。游标位置将向前移动 *整数常量* 行或者移至结果集末尾。

示例

以下示例演示包含 FETCH 语句的过程。

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
    FETCH emp_cur_3 INTO v_empno, v_ename;
        ...
END;
```

如果使用 %TYPE 属性来定义目标变量的数据类型，那么在数据库列的数据类型更改时，不需要更改 PL/SQL 应用程序中的目标变量声明。以下示例演示包含使用 %TYPE 属性定义的变量的过程。

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
    FETCH emp_cur_3 INTO v_empno, v_ename;
        ...
END;
```

如果表中的所有列都按它们的定义顺序进行检索，那么可以使用 %ROWTYPE 属性来定义一个记录，FETCH 语句将把检索到的数据放入该记录。然后，可以使用点分表示法来访问该记录中的每个字录。以下示例所演示的过程包含一个使用了 %ROWTYPE 的记录定义。此记录被用作 FETCH 语句的目标。

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
     v_emp_rec       emp%ROWTYPE;
     CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
     OPEN emp_cur_1;
     FETCH emp_cur_1 INTO v_emp_rec;
     DBMS_OUTPUT.PUT_LINE('Employee Number: ' || v_emp_rec.empno);
     DBMS_OUTPUT.PUT_LINE('Employee Name  : ' || v_emp_rec.ename);
END;
```

关闭游标 (PL/SQL)

从游标的相关联结果集中检索所有行之后，必须关闭该游标。关闭游标后，就无法引用该结果集。

但是，可以重新打开该游标，并可以访存新结果集的行。

语法

描述

游标名

指定 PL/SQL 上下文中先前声明的已打开游标的标识。

示例

以下示例演示对 CURSOR_EXAMPLE 过程中的游标执行 CLOSE 语句:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_emp_rec       emp%ROWTYPE;
    CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur_1;
    FETCH emp_cur_1 INTO v_emp_rec;
    DBMS_OUTPUT.PUT_LINE('Employee Number: ' || v_emp_rec.empno);
    DBMS_OUTPUT.PUT_LINE('Employee Name  : ' || v_emp_rec.ename);
    CLOSE emp_cur_1;
END;
```

将 %ROWTYPE 与游标配合使用 (PL/SQL)

%ROWTYPE 属性用于定义一个记录，该记录的字段与从游标或游标变量中访存的所有列相对应。每个字段都将采用其相应的列的数据类型。

%ROWTYPE 属性以游标名或游标变量名为前缀。语法为 *记录* 游标%ROWTYPE，其中 *记录* 是对记录指定的标识，*游标* 是在当前作用域中显式声明的游标。

以下示例说明如何将游标与 %ROWTYPE 属性配合使用，以便检索关于 EMP 表中每个职员的部门信息。

```
CREATE OR REPLACE PROCEDURE emp_info
IS
    CURSOR empcur IS SELECT ename, deptno FROM emp;
    myvar           empcur%ROWTYPE;
BEGIN
    OPEN empcur;
    LOOP
        FETCH empcur INTO myvar;
        EXIT WHEN empcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE( myvar.ename || ' works in department '
            || myvar.deptno );
    END LOOP;
    CLOSE empcur;
END;
```

对此过程（CALL emp_info;）进行的调用将返回以下样本输出:

```
SMITH works in department 20
ALLEN works in department 30
WARD works in department 30
JONES works in department 20
MARTIN works in department 30
BLAKE works in department 30
CLARK works in department 10
SCOTT works in department 20
KING works in department 10
```

```
TURNER works in department 30
ADAMS works in department 20
JAMES works in department 30
FORD works in department 20
MILLER works in department 10
```

游标属性 (PL/SQL)

每个游标都有一组属性，这些属性使应用程序能够测试该游标的状态。

这些属性是 %ISOPEN、%FOUND、%NOTFOUND 和 %ROWCOUNT。

%ISOPEN

此属性用于确定游标是否处于打开状态。将游标作为参数传递到函数或过程时，在尝试打开该游标之前知道该游标是否已打开十分有用。

%FOUND

此属性用于确定执行 FETCH 语句后游标是否包含行。如果 FETCH 语句的执行成功，那么 %FOUND 属性的值为 true。如果 FETCH 语句的执行未成功，那么 %FOUND 属性的值为 false。在下列情况下，结果未知:

- *游标变量名* 的值为 null
- *游标变量名* 的底层游标未打开
- 在对底层游标执行第一个 FETCH 语句之前对 %FOUND 属性进行求值
- FETCH 语句的执行返回了错误

%FOUND 属性提供了使用条件处理程序来查找没有更多行可供访存时返回的错误的高效替代方法。

%NOTFOUND

此属性的逻辑与 %FOUND 属性相反。

%ROWCOUNT

此属性用于确定自从打开游标后访存的行数。

表 10 对与某些游标事件相关联的属性值作了摘要说明。

表 10. 游标属性值摘要

游标属性	%ISOPEN	%FOUND	%NOTFOUND	%ROWCOUNT
在 OPEN 之前	False	未定义	未定义	“游标未打开”异常
在 OPEN 之后并在第一个 FETCH 之前	True	未定义	未定义	0
在第一个成功的 FETCH 之后	True	True	False	1
在第 *n* 个成功的 FETCH 之后（最后一行）	True	True	False	*n*
在第 *n*+1 个 FETCH 之后（在最后一行之后）	True	False	True	*n*
在 CLOSE 之后	False	未定义	未定义	“游标未打开”异常

游标变量 (PL/SQL)

*游标变量*是包含指向查询结果集的指针的游标。结果集是通过使用游标变量执行 OPEN FOR 语句来确定的。

游标变量与静态游标不同，它不与特定查询相关联。您可以使用包含不同查询的不同 OPEN FOR 语句来多次打开同一个游标变量。每次打开游标变量时，都将创建新的结果集并使该结果集通过该游标变量可供使用。

SYS_REFCURSOR 游标变量 (PL/SQL)

DB2 服务器支持声明具有 SYS_REFCURSOR 内置数据类型的游标变量，此变量可以与任何结果集相关联。

SYS_REFCURSOR 数据类型被称为弱类型化 REF CURSOR 类型。REF CURSOR 类型的强类型化游标变量要求指定结果集。

语法

描述

游标变量名
指定游标变量的标识。

SYS_REFCURSOR
指定游标变量的数据类型是内置 SYS_REFCURSOR 数据类型。

示例

以下示例演示 SYS_REFCURSOR 变量声明:

```
DECLARE emprefcur SYS_REFCURSOR;
```

用户定义 REF CURSOR 类型变量 (PL/SQL)

DB2 服务器支持用户定义 REF CURSOR 数据类型和游标变量声明。

可以通过在 PL/SQL 上下文中执行 TYPE 声明来定义用户定义 REF CURSOR 类型。定义此类型之后，就可以声明具有此类型的游标变量。

语法

描述

TYPE *游标类型名*
指定游标数据类型的标识。

IS REF CURSOR
指定游标具有用户定义 REF CURSOR 数据类型。

RETURN *return-type*
指定与游标相关联的返回类型。如果指定了 *return-type*，那么此 REF CURSOR 类型是强类型化类型；否则，它是弱类型化类型。

示例

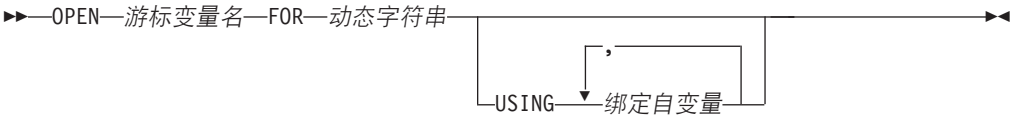
以下示例演示匿名块的声明节中的游标变量声明:

```
DECLARE
    TYPE emp_cur_type IS REF CURSOR RETURN emp%ROWTYPE;
    my_rec emp_cur_type;
BEGIN
    ...
END
```

包含游标变量的动态查询 (PL/SQL)

DB2 数据服务器通过 PL/SQL 上下文中的 OPEN FOR 语句来支持动态查询。

语法



描述

OPEN *游标变量名*
指定 PL/SQL 上下文中先前声明的游标变量的标识。

FOR *动态字符串*
指定包含 SELECT 语句（不带终止分号）的字符串文字或字符串变量。此语句可以包含指定的参数，例如 *:param1*。

USING *绑定自变量*
指定一个或多个绑定自变量，打开游标时，这些自变量的值将替换 *动态字符串* 中的占位符。

示例

以下示例演示使用了字符串文字的动态查询:

```
CREATE OR REPLACE PROCEDURE dept_query
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    OPEN emp_refcur FOR 'SELECT empno, ename FROM emp WHERE deptno = 30' ||
        ' AND sal >= 1500';
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
```

```
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

DEPT_QUERY 过程将生成以下示例输出:

```
CALL dept_query;

EMPNO    ENAME
-----    -------
7499     ALLEN
7698     BLAKE
7844     TURNER
```

可以对上一示例中的查询进行修改，以便使用绑定自变量来传递查询参数:

```
CREATE OR REPLACE PROCEDURE dept_query (
    p_deptno        emp.deptno%TYPE,
    p_sal           emp.sal%TYPE
)
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    OPEN emp_refcur FOR 'SELECT empno, ename FROM emp WHERE deptno = :dept'
        || ' AND sal >= :sal' USING p_deptno, p_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

以下 CALL 语句将生成与上一个示例相同的输出:

```
CALL dept_query(30, 1500);
```

最为灵活的方法是，使用字符串变量来传递 SELECT 语句:

```
CREATE OR REPLACE PROCEDURE dept_query (
    p_deptno        emp.deptno%TYPE,
    p_sal           emp.sal%TYPE
)
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    p_query_string  VARCHAR2(100);
BEGIN
    p_query_string := 'SELECT empno, ename FROM emp WHERE ' ||
        'deptno = :dept AND sal >= :sal';
    OPEN emp_refcur FOR p_query_string USING p_deptno, p_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

此版本的 DEPT_QUERY 过程将生成以下示例输出:

```
CALL dept_query(20, 1500);

EMPNO    ENAME
-----    -------
7566     JONES
7788     SCOTT
7902     FORD
```

示例: 从过程中返回 REF CURSOR (PL/SQL)

此示例演示如何定义和打开 REF CURSOR 变量并接着将其作为过程参数进行传递。

将游标变量指定为 IN OUT 参数，以便将结果集提供给过程调用者使用:

```
CREATE OR REPLACE PROCEDURE emp_by_job (
    p_job           VARCHAR2,
    p_emp_refcur    IN OUT SYS_REFCURSOR
)
IS
BEGIN
    OPEN p_emp_refcur FOR SELECT empno, ename FROM emp WHERE job = p_job;
END;
```

在以下匿名块中，通过将过程的 IN OUT 参数指定给该匿名块的声明节中先前声明的游标变量来调用 EMP_BY_JOB 过程。接着，使用此游标变量来访存结果集。

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE := 'SALESMAN';
    v_emp_refcur    SYS_REFCURSOR;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES WITH JOB ' || v_job);
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    emp_by_job(v_job, v_emp_refcur);
    LOOP
        FETCH v_emp_refcur INTO v_empno, v_ename;
        EXIT WHEN v_emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE v_emp_refcur;
END;
```

执行此匿名块时，将生成以下示例输出:

```
EMPLOYEES WITH JOB SALESMAN
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7844     TURNER
```

示例: 将游标操作模块化 (PL/SQL)

此示例演示对游标变量执行的各种操作可以如何进行模块化以形成不同的程序或 PL/SQL 组件。

以下示例演示一个过程，此过程打开一个游标变量，该游标变量的查询将检索 EMP 表中所有的行:

```
CREATE OR REPLACE PROCEDURE open_all_emp (
    p_emp_refcur    IN OUT SYS_REFCURSOR
)
IS
BEGIN
    OPEN p_emp_refcur FOR SELECT empno, ename FROM emp;
END;
```

以下示例中的过程打开一个游标变量，该游标变量的查询将检索给定部门的所有行:

```
CREATE OR REPLACE PROCEDURE open_emp_by_dept (
    p_emp_refcur    IN OUT SYS_REFCURSOR,
    p_deptno        emp.deptno%TYPE
)
IS
BEGIN
    OPEN p_emp_refcur FOR SELECT empno, ename FROM emp
        WHERE deptno = p_deptno;
END;
```

以下示例演示一个过程，此过程打开一个游标变量，该游标变量的查询将检索 DEPT 表中所有的行:

```
CREATE OR REPLACE PROCEDURE open_dept (
    p_dept_refcur    IN OUT SYS_REFCURSOR
)
IS
BEGIN
    OPEN p_dept_refcur FOR SELECT deptno, dname FROM dept;
END;
```

以下示例中的过程访存并显示包含职员编号和姓名的游标变量结果集:

```
CREATE OR REPLACE PROCEDURE fetch_emp (
    p_emp_refcur    IN OUT SYS_REFCURSOR
)
IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH p_emp_refcur INTO v_empno, v_ename;
        EXIT WHEN p_emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
END;
```

以下示例演示一个过程，此过程访存并显示包含部门编号和名称的游标变量结果集:

```
CREATE OR REPLACE PROCEDURE fetch_dept (
    p_dept_refcur   IN SYS_REFCURSOR
)
IS
    v_deptno        dept.deptno%TYPE;
    v_dname         dept.dname%TYPE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('DEPT   DNAME');
    DBMS_OUTPUT.PUT_LINE('----   ---------');
    LOOP
        FETCH p_dept_refcur INTO v_deptno, v_dname;
        EXIT WHEN p_dept_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_deptno || '     ' || v_dname);
    END LOOP;
END;
```

以下示例演示一个过程，此过程关闭游标变量:

```
CREATE OR REPLACE PROCEDURE close_refcur (
    p_refcur        IN OUT SYS_REFCURSOR
)
IS
BEGIN
    CLOSE p_refcur;
END;
```

以下示例演示一个匿名块，此匿名块执行这些过程:

```
DECLARE
    gen_refcur      SYS_REFCURSOR;
BEGIN
    DBMS_OUTPUT.PUT_LINE('ALL EMPLOYEES');
    open_all_emp(gen_refcur);
    fetch_emp(gen_refcur);
    DBMS_OUTPUT.PUT_LINE('****************');

    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #10');
    open_emp_by_dept(gen_refcur, 10);
    fetch_emp(gen_refcur);
    DBMS_OUTPUT.PUT_LINE('****************');

    DBMS_OUTPUT.PUT_LINE('DEPARTMENTS');
    open_dept(gen_refcur);
    fetch_dept(gen_refcur);
    DBMS_OUTPUT.PUT_LINE('*****************');

    close_refcur(gen_refcur);
END;
```

执行此匿名块时，将生成以下示例输出:

```
ALL EMPLOYEES
EMPNO    ENAME
-----    -------
7369     SMITH
7499     ALLEN
7521     WARD
7566     JONES
7654     MARTIN
7698     BLAKE
7782     CLARK
7788     SCOTT
7839     KING
7844     TURNER
7876     ADAMS
7900     JAMES
7902     FORD
7934     MILLER
****************
EMPLOYEES IN DEPT #10
EMPNO    ENAME
-----    -------
7782     CLARK
7839     KING
7934     MILLER
****************
DEPARTMENTS
DEPT   DNAME
----   ---------
10     ACCOUNTING
```

```
20      RESEARCH
30      SALES
40      OPERATIONS
*****************
```

第 20 章 触发器 (PL/SQL)

PL/SQL 触发器是具有名称的数据库对象，它封装并定义一组要在响应对表执行的插入、更新或删除操作时执行的操作。请使用 PL/SQL CREATE TRIGGER 语句来创建触发器。

触发器的类型 (PL/SQL)

DB2 数据服务器支持在 PL/SQL 上下文中使用行级和语句级触发器。

*行级触发器*对于触发事件所影响的每一行触发一次。例如，如果将删除定义为特定表的触发事件，并且单个 DELETE 语句从该表中删除五行，那么触发器会触发五次，即，对每一行触发一次。

*语句级触发器*仅对每个语句触发一次。在使用以上示例的情况下，如果将删除定义为特定表的触发事件，并且单个 DELETE 语句从该表中删除五行，那么触发器会触发一次。不能对 BEFORE 触发器或 INSTEAD OF 触发器指定语句级触发器粒度。

会在触发语句影响每一行之前或之后执行触发器代码块（除执行触发器代码块而不根据激活语句影响每一行的 INSTEAD OF 触发器外）。

触发器变量 (PL/SQL)

NEW 和 OLD 是特殊变量，您可以将其与 PL/SQL 触发器配合使用，而不必显式地定义这些变量。

- NEW 是一个伪记录名，它引用行级触发器中的插入和更新操作的新表行。它的用法是 `:NEW.column`，其中 *column* 是表中对其定义此触发器的列的名称。
  - 在*行级前触发器*中使用时，:NEW.*column* 的初始内容是所要插入的新行中的列值或者要替换旧行的行中的列值。
  - 在*行级后触发器*中使用时，新列值已存储在表中。
  - 当删除操作激活触发器时，该触发器中使用的 :NEW.*column* 为空。

  在触发器代码块中，可以像使用任何其他变量一样使用 :NEW.*column*。如果在行级前触发器的代码块中将值赋予 :NEW.*column*，那么所插入或更新的行将使用所赋予的值。
- OLD 是一个伪记录名，它引用行级触发器中的更新和删除操作的旧表行。它的用法是 `:OLD.column`，其中 *column* 是表中对其定义此触发器的列的名称。
  - 在*行级前触发器*中使用时，:OLD.*column* 的初始内容是所要删除的行中的列值或者要被新行替换的旧行中的列值。
  - 在*行级后触发器*中使用时，旧列值已不再存储在表中。
  - 当插入操作激活触发器时，该触发器中使用的 :OLD.*column* 为空。

  在触发器代码块中，可以像使用任何其他变量一样使用 :OLD.*column*。如果在行级前触发器的代码块中将值赋予 :OLD.*column*，那么所赋予的值不会影响触发器的操作。

触发器事件谓词 (PL/SQL)

只能在触发器中使用触发器事件谓词 UPDATING、DELETING 和 INSERTING 来识别已激活该触发器的事件。

DELETING
如果删除操作已激活触发器，那么为 True。否则，为 False。

INSERTING
如果插入操作已激活触发器，那么为 True。否则，为 False。

UPDATING
如果更新操作已激活触发器，那么为 True。否则，为 False。

在 WHEN 子句或 PL/SQL 语句中，可将这些谓词指定为单个搜索条件，也可将它们指定为复合搜索条件内的布尔因子。

事务和异常 (PL/SQL)

触发器始终作为执行触发语句的事务的组成部分执行。

如果在触发器代码块中未发生异常，那么仅当包含触发语句的事务落实时，才会落实该触发器中的数据操作语言（DML）的效果。如果该事务回滚，那么该触发器中的 DML 的效果也将回滚。

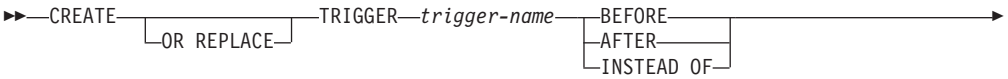
DB2 回滚只能在原子块中发生或者通过 UNDO 处理程序进行。除非应用程序强制回滚外层事务，否则触发语句本身不会回滚。

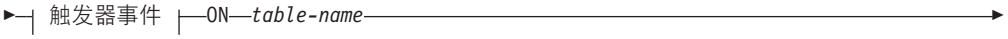
如果在触发器代码块中发生未处理的异常，那么调用语句将回滚。

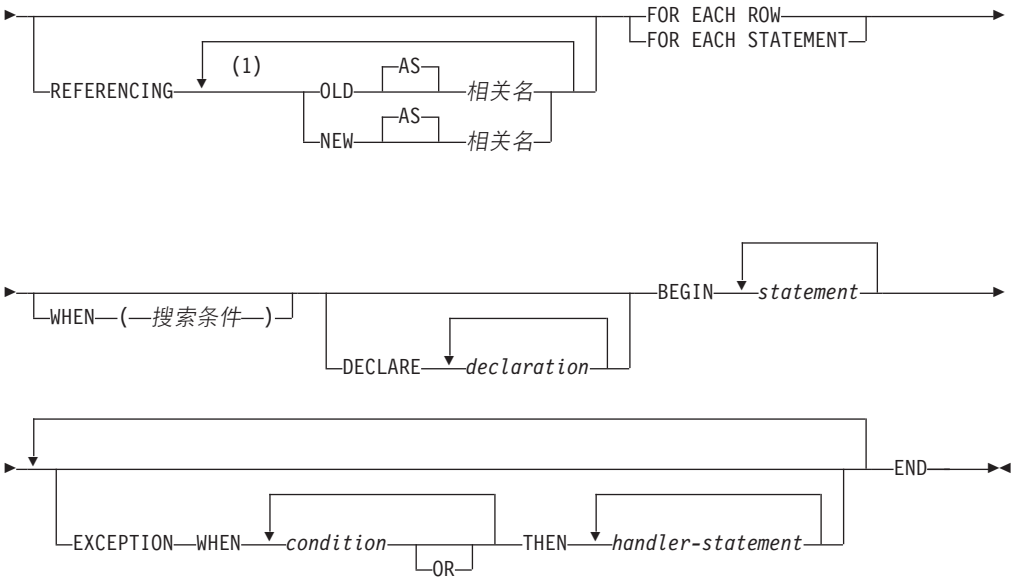
CREATE TRIGGER 语句 (PL/SQL)

CREATE TRIGGER 语句在数据库中定义 PL/SQL 触发器。

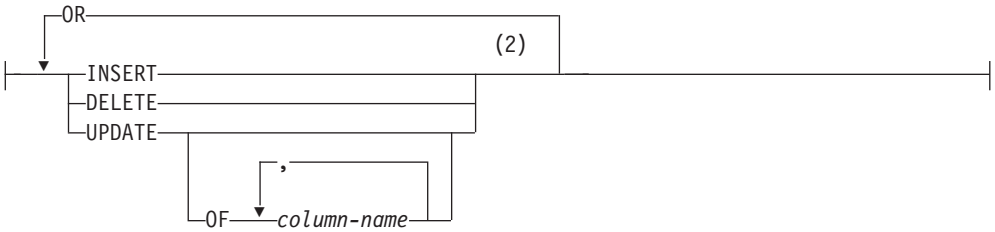
语法







触发器事件:



注:

1 OLD 和 NEW 只能指定一次。

2 不能对同一操作多次指定触发器事件。例如，允许使用 INSERT OR DELETE，但不允许使用 INSERT OR INSERT。

描述

OR REPLACE

指定如果当前服务器上已存在触发器定义，那么替换该定义。在目录中替换新定义之前，将有效地删除现有定义。如果当前服务器上不存在触发器定义，那么将忽略此选项。

trigger-name

指定触发器的名称。此名称（包括隐式或显式的模式名）不能标识已在目录中描述的触发器 (SQLSTATE 42710)。如果指定了两部分的名称，那么模式名不能以“SYS”开头 (SQLSTATE 42939)。

BEFORE

指定在将主题表的实际更新所引起的任何更改应用于数据库之前执行相关联的触发操作。

AFTER

指定在将主题表的实际更新所引起的更改应用于数据库之后执行相关联的触发操作。

INSTEAD OF
指定由相关联的触发操作替换对主题视图的操作。

触发器事件
指定每当对主题表应用其中一个事件时都要执行与触发器相关联的触发操作。可指定这些事件的任意组合，但每个事件（INSERT、DELETE 和 UPDATE）只能指定一次 (SQLSTATE 42613)。

INSERT
指定每当对主题表执行 INSERT 操作时执行与此触发器相关联的触发操作。

DELETE
指定每当对主题表执行 DELETE 操作时执行与此触发器相关联的触发操作。

UPDATE
指定每当对主题表中指定的列或暗指的列执行 UPDATE 操作时执行与此触发器相关联的触发操作。

如果未指定可选的 *列名* 列表，那么暗指该表的每一列。因此，省略 *列名* 列表意味着，更新该表的任何列都将激活触发器。

OF *column-name*,...
指定的每个 *column-name* 都必须是基本表的列 (SQLSTATE 42703)。如果此触发器是 BEFORE 触发器，那么指定的 *列名* 不能是除标识列以外的生成列 (SQLSTATE 42989)。*列名* 不能在*列名* 列表中多次出现 (SQLSTATE 42711)。仅当更新*列名* 列表中标识的列时，才将激活此触发器。不能对 INSTEAD OF 触发器指定此子句 (SQLSTATE 42613)。

ON *table-name*
指定 BEFORE 触发器或 AFTER 触发器定义的主题表。此名称必须指定基本表或者解析为基本表的别名（SQLSTATE 42704 或 42809）。此名称不能指定目录表 (SQLSTATE 42832)、具体化查询表 (SQLSTATE 42997)、已创建临时表、已声明临时表 (SQLSTATE 42995) 或昵称 (SQLSTATE 42809)。

REFERENCING
指定*转换变量*的相关名。相关名标识触发 SQL 操作所影响的行集中的特定行。通过使用 *相关名*（如下指定）对列进行限定，触发操作可以对触发 SQL 操作所影响的每一行进行处理。

OLD AS *correlation-name*
指定一个相关名，此名称标识触发 SQL 操作之前的行状态。如果触发器事件为 INSERT，那么行中的值为空值。

NEW AS *correlation-name*
指定一个相关名，此名称标识由触发 SQL 操作以及已执行的 BEFORE 触发器中的任何 SET 语句修改的行状态。如果触发器事件为 DELETE，那么行中的值为空值。

如果未调用 REFERENCING 子句，那么可选择使用触发器变量 NEW 和 OLD 而不必显式定义。

FOR EACH ROW
指定对触发 SQL 操作所影响的主题表中的每一行执行一次触发操作。

FOR EACH STATEMENT
指定对整个语句仅应用触发操作一次。

WHEN

(*search-condition***)**

指定具有 true 或 false 值或者未知值的条件。*search-condition* 使您能够确定是否应该执行特定的触发操作。仅当指定的搜索条件求值为 true 时，才会执行相关联的操作。

declaration

指定变量声明。

statement **或** *handler-statement*

指定 PL/SQL 程序语句。触发器主体可以包含嵌套的块。

condition

指定异常条件名，例如 NO_DATA_FOUND。

示例

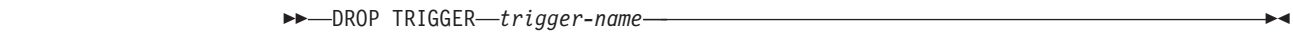
以下示例显示行级别的前触发器，对于每个隶属于部门 30 的新职员，在将该职员的记录插入到 EMP 表之前，此触发器会计算该职员的佣金。它还记录异常表中任何增幅超出 50% 的薪水增长情况:

```
CREATE TABLE emp (
    name            VARCHAR2(10),
    deptno          NUMBER,
    sal             NUMBER,
    comm            NUMBER
)
/

CREATE TABLE exception (
    name            VARCHAR2(10),
    old_sal         NUMBER,
    new_sal         NUMBER
)
/

CREATE OR REPLACE TRIGGER emp_comm_trig
    BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW
BEGIN
    IF (:NEW.deptno = 30 and INSERTING) THEN
        :NEW.comm := :NEW.sal * .4;
    END IF;

    IF (UPDATING and (:NEW.sal - :OLD.sal) > :OLD.sal * .5) THEN
       INSERT INTO exception VALUES (:NEW.name, :OLD.sal, :NEW.sal);
    END IF;
END
/
```

删除触发器 (PL/SQL)

通过使用 DROP TRIGGER 语句，可以从数据库中除去触发器。

语法



描述

trigger-name
指定所要删除的触发器的名称。

示例: 触发器 (PL/SQL)

DB2 数据服务器能够对 PL/SQL 触发器定义进行编译。这些示例帮助您创建有效的触发器以及诊断 PL/SQL 触发器编译错误。

行级前触发器

以下示例演示行级别的前触发器，对于每个隶属于部门 30 的新职员，在将该职员的记录插入到 EMP 表之前，此触发器计算该职员的佣金:

```
CREATE OR REPLACE TRIGGER emp_comm_trig
    BEFORE INSERT ON emp
    FOR EACH ROW
BEGIN
    IF :NEW.deptno = 30 THEN
        :NEW.comm := :NEW.sal * .4;
    END IF;
END;
```

此触发器计算两个新职员的佣金并将那些值作为新职员行的组成部分插入:

```
INSERT INTO emp VALUES (9005,'ROBERS','SALESMAN',7782,SYSDATE,3000.00,NULL,30);

INSERT INTO emp VALUES (9006,'ALLEN','SALESMAN',7782,SYSDATE,4500.00,NULL,30);

SELECT * FROM emp WHERE empno IN (9005, 9006);

     EMPNO ENAME      JOB              MGR HIREDATE         SAL       COMM     DEPTNO
---------- ---------- --------- ---------- --------- ---------- ---------- ----------
      9005 ROBERS     SALESMAN         7782 01-APR-05       3000       1200         30
      9006 ALLEN      SALESMAN         7782 01-APR-05       4500       1800         30
```

行级后触发器

以下示例演示了 3 个行级后触发器。

- 将新的职员行插入到 EMP 表时，一个触发器（EMP_INS_TRIG）将在 JOBHIST 表中为该职员添加新行，并在 EMPCHGLOG 表中添加包含操作描述的行。
- 更新现有的职员行时，第二个触发器（EMP_CHG_TRIG）将最新 JOBHIST 行（假定其 ENDDATE 为 null）的 ENDDATE 列设置为当前日期，并插入包含职员的新信息的新 JOBHIST 行。此触发器还在 EMPCHGLOG 表中添加包含操作描述的行。
- 从 EMP 表中删除职员行时，第三个触发器（EMP_DEL_TRIG）在 EMPCHGLOG 表中添加包含操作描述的行。

```
CREATE TABLE empchglog (
    chg_date        DATE,
    chg_desc        VARCHAR2(30)
);
CREATE OR REPLACE TRIGGER emp_ins_trig
    AFTER INSERT ON emp
    FOR EACH ROW
DECLARE
    v_empno         emp.empno%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_action        VARCHAR2(7);
    v_chgdesc       jobhist.chgdesc%TYPE;
```

```
BEGIN
    v_action := 'Added';
    v_empno := :NEW.empno;
    v_deptno := :NEW.deptno;
    INSERT INTO jobhist VALUES (:NEW.empno, SYSDATE, NULL,
        :NEW.job, :NEW.sal, :NEW.comm, :NEW.deptno, 'New Hire');

    INSERT INTO empchglog VALUES (SYSDATE,
        v_action || ' employee # ' || v_empno);
END;

CREATE OR REPLACE TRIGGER emp_chg_trig
    AFTER UPDATE ON emp
    FOR EACH ROW
DECLARE
    v_empno         emp.empno%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_action        VARCHAR2(7);
    v_chgdesc       jobhist.chgdesc%TYPE;
BEGIN
    v_action := 'Updated';
    v_empno := :NEW.empno;
    v_deptno := :NEW.deptno;
    v_chgdesc := '';
    IF NVL(:OLD.ename, '-null-') != NVL(:NEW.ename, '-null-') THEN
        v_chgdesc := v_chgdesc || 'name, ';
    END IF;
    IF NVL(:OLD.job, '-null-') != NVL(:NEW.job, '-null-') THEN
        v_chgdesc := v_chgdesc || 'job, ';
    END IF;
    IF NVL(:OLD.sal, -1) != NVL(:NEW.sal, -1) THEN
        v_chgdesc := v_chgdesc || 'salary, ';
    END IF;
    IF NVL(:OLD.comm, -1) != NVL(:NEW.comm, -1) THEN
        v_chgdesc := v_chgdesc || 'commission, ';
    END IF;
    IF NVL(:OLD.deptno, -1) != NVL(:NEW.deptno, -1) THEN
        v_chgdesc := v_chgdesc || 'department, ';
    END IF;
    v_chgdesc := 'Changed ' || RTRIM(v_chgdesc, ', ');
    UPDATE jobhist SET enddate = SYSDATE WHERE empno = :OLD.empno
        AND enddate IS NULL;
    INSERT INTO jobhist VALUES (:NEW.empno, SYSDATE, NULL,
        :NEW.job, :NEW.sal, :NEW.comm, :NEW.deptno, v_chgdesc);

    INSERT INTO empchglog VALUES (SYSDATE,
        v_action || ' employee # ' || v_empno);
END;

CREATE OR REPLACE TRIGGER emp_del_trig
    AFTER DELETE ON emp
    FOR EACH ROW
DECLARE
    v_empno         emp.empno%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_action        VARCHAR2(7);
    v_chgdesc       jobhist.chgdesc%TYPE;
BEGIN
    v_action := 'Deleted';
    v_empno := :OLD.empno;
    v_deptno := :OLD.deptno;

    INSERT INTO empchglog VALUES (SYSDATE,
        v_action || ' employee # ' || v_empno);
END;
```

在以下示例中，使用两个不同的 INSERT 语句来添加两个职员行，然后使用单一 UPDATE 语句来更新这两行。JOBHIST 表显示触发器对每个受影响的行执行的操作: 两个新职员的两个新聘用条目以及两个更改佣金记录。EMPCHGLOG 表还显示触发器共触发 4 次，即，对这两行执行的每项操作触发一次。

```
INSERT INTO emp VALUES (9003,'PETERS','ANALYST',7782,SYSDATE,5000.00,NULL,40);

INSERT INTO emp VALUES (9004,'AIKENS','ANALYST',7782,SYSDATE,4500.00,NULL,40);

UPDATE emp SET comm = sal * 1.1 WHERE empno IN (9003, 9004);

SELECT * FROM jobhist WHERE empno IN (9003, 9004);

     EMPNO STARTDATE ENDDATE   JOB              SAL       COMM     DEPTNO CHGDESC
---------- --------- --------- --------- ---------- ---------- ---------- ------------------
      9003 31-MAR-05 31-MAR-05 ANALYST         5000                    40 New Hire
      9004 31-MAR-05 31-MAR-05 ANALYST         4500                    40 New Hire
      9003 31-MAR-05           ANALYST         5000       5500         40 Changed commission
      9004 31-MAR-05           ANALYST         4500       4950         40 Changed commission

SELECT * FROM empchglog;

CHG_DATE  CHG_DESC
--------- ------------------------------
31-MAR-05 Added employee # 9003
31-MAR-05 Added employee # 9004
31-MAR-05 Updated employee # 9003
31-MAR-05 Updated employee # 9004
```

使用单一 DELETE 语句删除这两个职员之后，EMPCHGLOG 表显示触发器已触发两次，即，对删除的每个雇员触发一次:

```
DELETE FROM emp WHERE empno IN (9003, 9004);

SELECT * FROM empchglog;

CHG_DATE  CHG_DESC
--------- ------------------------------
31-MAR-05 Added employee # 9003
31-MAR-05 Added employee # 9004
31-MAR-05 Updated employee # 9003
31-MAR-05 Updated employee # 9004
31-MAR-05 Deleted employee # 9003
31-MAR-05 Deleted employee # 9004
```

第 21 章 程序包 (PL/SQL)

DB2 数据服务器支持 PL/SQL 程序包定义。PL/SQL 程序包是使用公共限定符（即，程序包名）进行引用的函数、过程、变量、游标、用户定义的类型以及记录的集合并具有名称。

程序包具有下列特征:

- 程序包提供了一种便利的方法来组织那些具有相关用途的函数和过程。使用程序包函数和过程所需的许可权取决于授予整个程序包的特权。
- 可以将程序包中的某些项声明为公用项。公用实体可视，并可能被其他对该程序包拥有 EXECUTE 特权的程序引用。对于公用函数和过程而言，只有它们的特征符可视。这些函数和过程的 PL/SQL 代码不可供其他程序访问；因此，利用此类程序包的应用程序只依赖于特征符中的信息。
- 可以将程序包中的其他项声明为私有项。私有实体只能由该程序包中的函数和过程引用和使用，而不可供外部应用程序引用和使用。

程序包组件 (PL/SQL)

程序包由两种主要组件组成: 程序包规范和软件包主体。

- *程序包规范*是公共接口，它由可以从程序包外部引用的元素组成。要创建程序包规范，请执行 CREATE PACKAGE 语句。
- *软件包主体*包含程序包规范中声明的所有过程和函数的实际实现以及专用类型、变量和游标的任何声明。要创建软件包主体，请执行 CREATE PACKAGE BODY 语句。

创建程序包 (PL/SQL)

通过创建程序包规范，您能够将相关的数据类型、过程和函数定义封装在数据库中的单一上下文中。

程序包是对模式的扩展，它们为它们所引用的对象提供名称空间支持。它们是可以在其中定义可执行代码的存储库。使用程序包涉及引用或执行在程序包规范中定义并在程序包中实现的对象。

创建程序包规范 (PL/SQL)

程序包规范确定可以从该程序包外部引用哪些程序包对象。可以从程序包外部引用的对象被称为该程序包的公用元素。

以下示例说明如何创建名为 EMP_ADMIN 并由两个函数和两个存储过程组成的程序包规范。

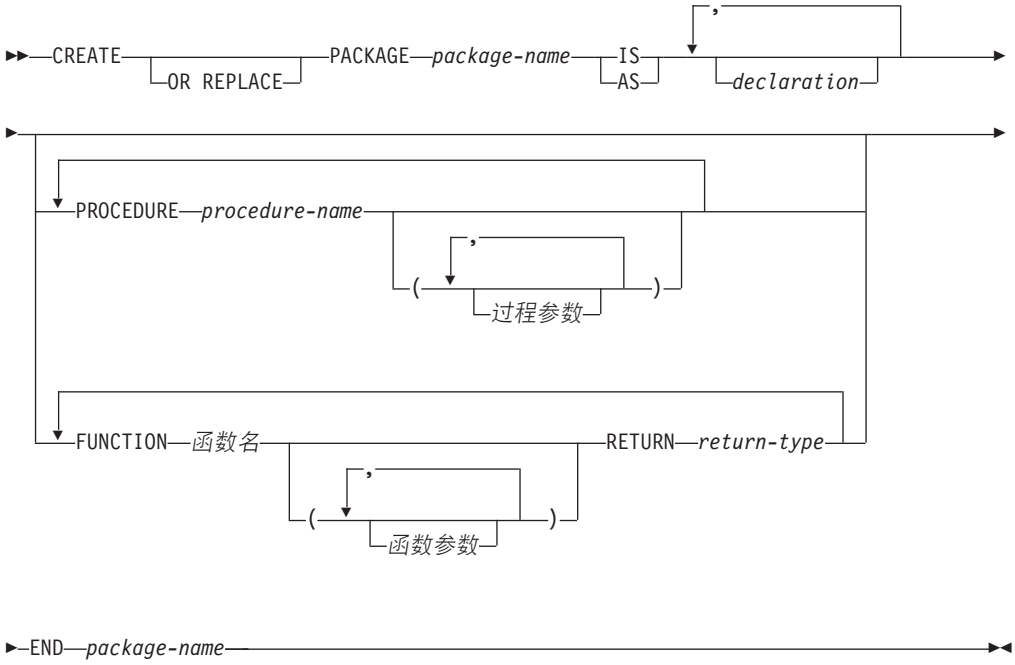
```
CREATE OR REPLACE PACKAGE emp_admin
IS

   FUNCTION get_dept_name (
      p_deptno        NUMBER DEFAULT 10
   )
   RETURN VARCHAR2;
```

```
   FUNCTION update_emp_sal (
      p_empno         NUMBER,
      p_raise         NUMBER
   )
   RETURN NUMBER;
   PROCEDURE hire_emp (
      p_empno         NUMBER,
      p_ename         VARCHAR2,
      p_job           VARCHAR2,
      p_sal           NUMBER,
      p_hiredate      DATE DEFAULT sysdate,
      p_comm          NUMBER DEFAULT 0,
      p_mgr           NUMBER,
      p_deptno        NUMBER DEFAULT 10
   );
   PROCEDURE fire_emp (
      p_empno         NUMBER
   );

END emp_admin;
```

CREATE PACKAGE 语句 (PL/SQL)

CREATE PACKAGE 语句创建程序包规范，后者定义程序包的接口。

语法



描述

package-name

指定程序包的标识。

declaration
指定公用项的标识。可以通过语法 *package-name.项名* 从程序包外部访问公用项。可以存在零个或零个以上的公用项。公用项声明必须在过程声明或函数声明之前。*声明* 可以是下列任何一项:

- 集合声明
- EXCEPTION 声明
- 记录声明
- REF CURSOR 和游标变量声明
- 集合、记录或 REF CURSOR 类型变量的 TYPE 定义
- 变量声明

procedure-name
指定公用过程的标识。可以通过语法 *package-name.procedure-name*() 从程序包外部调用公用过程。

过程参数
指定过程的形参的标识。

function-name
指定公用函数的标识。可以通过语法 *package-name.function-name*() 从程序包外部调用公用函数。

function-parameter
指定函数的形参的标识。可以使用缺省值对输入（IN 方式）参数进行初始化。

return-type
指定函数所返回的值的数据类型。

备注

可以采用已模糊化的格式来提交 CREATE PACKAGE 语句。在已模糊化的语句中，只有程序包名可读。按照下面这样一种方式对该语句的其余内容进行编码：这些内容不可读，但是可由数据库服务器解码。可以通过调用 DBMS_DDL.WRAP 函数来生成模糊化的语句。

创建软件包主体 (PL/SQL)

软件包主体包含程序包规范中声明的所有过程和函数的实现。

以下示例说明如何创建 EMP_ADMIN 程序包规范的软件包主体。

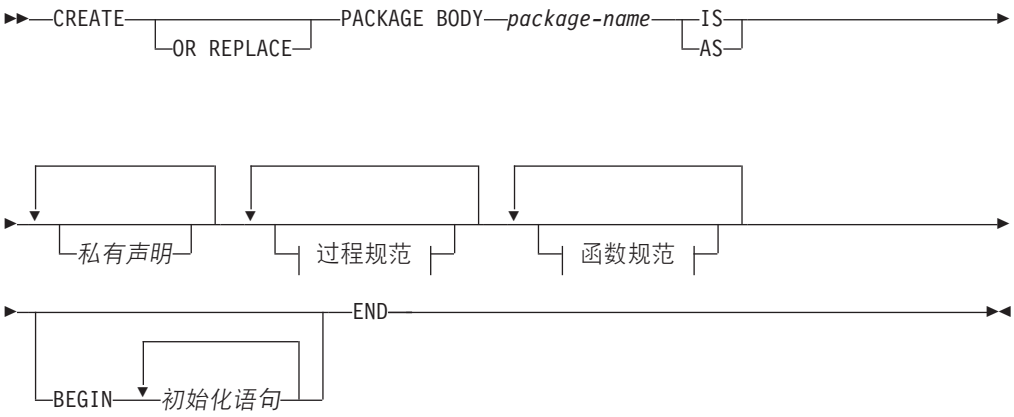
```
--
--  Package body for the 'emp_admin' package.
--
CREATE OR REPLACE PACKAGE BODY emp_admin
IS
   --
   --  Function that queries the 'dept' table based on the department
   --  number and returns the corresponding department name.
   --
   FUNCTION get_dept_name (
      p_deptno        IN NUMBER DEFAULT 10
   )
   RETURN VARCHAR2
   IS
      v_dname         VARCHAR2(14);
   BEGIN
```

```
      SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
      RETURN v_dname;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         DBMS_OUTPUT.PUT_LINE('Invalid department number ' || p_deptno);
         RETURN '';
   END;
   --
   --  Function that updates an employee's salary based on the
   --  employee number and salary increment/decrement passed
   --  as IN parameters.  Upon successful completion the function
   --  returns the new updated salary.
   --
   FUNCTION update_emp_sal (
      p_empno         IN NUMBER,
      p_raise         IN NUMBER
   )
   RETURN NUMBER
   IS
      v_sal           NUMBER := 0;
   BEGIN
      SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
      v_sal := v_sal + p_raise;
      UPDATE emp SET sal = v_sal WHERE empno = p_empno;
      RETURN v_sal;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
         RETURN -1;
      WHEN OTHERS THEN
         DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
         DBMS_OUTPUT.PUT_LINE(SQLERRM);
         DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
         DBMS_OUTPUT.PUT_LINE(SQLCODE);
         RETURN -1;
   END;
   --
   --  Procedure that inserts a new employee record into the 'emp' table.
   --
   PROCEDURE hire_emp (
      p_empno         NUMBER,
      p_ename         VARCHAR2,
      p_job           VARCHAR2,
      p_sal           NUMBER,
      p_hiredate      DATE    DEFAULT sysdate,
      p_comm          NUMBER  DEFAULT 0,
      p_mgr           NUMBER,
      p_deptno        NUMBER  DEFAULT 10
   )
   AS
   BEGIN
      INSERT INTO emp(empno, ename, job, sal, hiredate, comm, mgr, deptno)
         VALUES(p_empno, p_ename, p_job, p_sal,
                p_hiredate, p_comm, p_mgr, p_deptno);
   END;
   --
   --  Procedure that deletes an employee record from the 'emp' table based
   --  on the employee number.
   --
   PROCEDURE fire_emp (
      p_empno         NUMBER
   )
   AS
   BEGIN
      DELETE FROM emp WHERE empno = p_empno;
   END;
END;
```

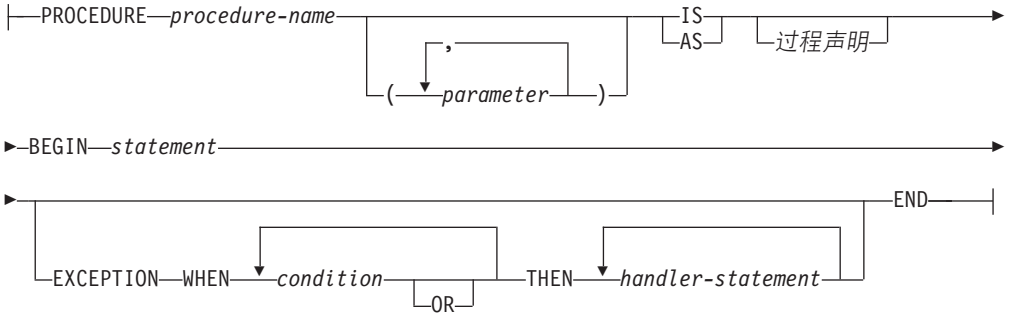
CREATE PACKAGE BODY 语句 (PL/SQL)

CREATE PACKAGE BODY 语句将创建一个程序包主体，该主体包含程序包规范中声明的所有过程和函数的实现以及私有类型、变量和游标的任何声明。

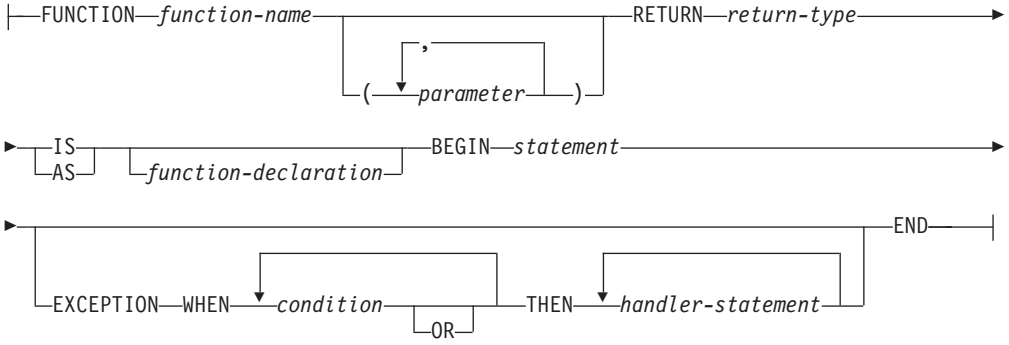
语法



过程规范:



函数规范:



描述

package-name

指定要创建其主体的程序包的名称。必须存在同名的程序包规范。

private-declaration
指定可以由程序包中的任何过程或函数访问的私有变量的名称。可以存在零个或零个以上私有变量。*private-declaration* 可以是下列任何一项:

- 变量声明
- 记录声明
- 集合声明
- REF CURSOR 和游标变量声明
- REF CURSOR 类型的记录、集合或变量的 TYPE 定义

procedure-name
指定程序包规范及其特征符中声明的公用过程的名称。特征符可以指定下列任何一项: 形参名称、数据类型、参数方式、形参的顺序或形参的数目。当过程名和程序包规范与公用过程的声明的特征符完全匹配时，*procedure-name* 将定义此公用过程的主体。

如果这些条件都不成立，那么 *procedure-name* 将定义新的私有过程。

parameter
指定过程的形参。

procedure-declaration
指定只能从过程 *procedure-name* 中访问的声明。这是 PL/SQL 语句。

statement
指定 PL/SQL 程序语句。

function-name
指定程序包规范及其特征符中声明的公用函数的名称。特征符可以指定下列任何一项: 形参名称、数据类型、参数方式、形参的顺序或形参的数目。当函数名和程序包规范与公用函数的声明的特征符完全匹配时，*function-name* 将定义此公用函数的主体。

如果这些条件都不成立，那么 *function-name* 将定义新的私有函数。

parameter
指定函数的形参。

return-type
指定函数所返回的值的数据类型。

function-declaration
指定只能从函数 *function-name* 中访问的声明。这是 PL/SQL 语句。

statement
指定 PL/SQL 程序语句。

initialization-statement
指定程序包主体的初始化节中的语句。如果指定了初始化节，那么它必须至少包含一条语句。对于每个用户会话，将在第一次引用该程序包时执行一次初始化节中的语句。

备注

可以采用已模糊化的格式来提交 CREATE PACKAGE BODY 语句。在已模糊化的语句中，只有程序包名可读。按照下面这样一种方式对该语句的其余内容进行编码: 这些

内容不可读，但是可由数据库服务器解码。可以通过调用 DBMS_DDL.WRAP 函数来生成模糊化的语句。

引用程序包对象 (PL/SQL)

有时，对程序包中定义的对象进行的引用必须由程序包名进行限定。

要引用程序包规范中声明的对象，请指定程序包名、句点字符和对象名。如果该程序包不是在当前模式中定义的，请同时指定模式名。例如:

```
程序包名.类型名
程序包名.项名
程序包名.子程序名
模式.程序包名.子程序名
```

示例

以下示例包含对程序包 EMP_ADMIN 中定义的函数 GET_DEPT_NAME 的引用:

```
select emp_admin.get_dept_name(10) from dept
```

包含用户定义的类型的程序包 (PL/SQL)

在程序包中，可以声明和引用用户定义的类型。

以下示例说明 EMP_RPT 程序包的程序包规范。此定义包含下列声明:

- 可公开访问的记录类型 EMPREC_TYP
- 可公开访问的弱类型化 REF CURSOR 类型 EMP_REFCUR
- 两个函数 GET_DEPT_NAME 和 OPEN_EMP_BY_DEPT; 后一个函数返回 REF CURSOR 类型 EMP_REFCUR
- 两个过程 FETCH_EMP 和 CLOSE_REFCUR; 这两个过程都声明了一个弱类型化 REF CURSOR 类型作为形参

```
CREATE OR REPLACE PACKAGE emp_rpt
IS
    TYPE emprec_typ IS RECORD (
        empno       NUMBER(4),
        ename       VARCHAR(10)
    );
    TYPE emp_refcur IS REF CURSOR;

    FUNCTION get_dept_name (
        p_deptno    IN NUMBER
    ) RETURN VARCHAR2;
    FUNCTION open_emp_by_dept (
        p_deptno    IN emp.deptno%TYPE
    ) RETURN EMP_REFCUR;
    PROCEDURE fetch_emp (
        p_refcur    IN OUT SYS_REFCURSOR
    );
    PROCEDURE close_refcur (
        p_refcur    IN OUT SYS_REFCURSOR
    );
END emp_rpt;
```

相关联程序包主体的定义包含下列私有变量声明:

- 静态游标 DEPT_CUR
- 关联数组类型 DEPTTAB_TYP

- 关联数组变量 T_DEPT
- 整数变量 T_DEPT_MAX
- 记录变量 R_EMP

```
CREATE OR REPLACE PACKAGE BODY emp_rpt
IS
    CURSOR dept_cur IS SELECT * FROM dept;
    TYPE depttab_typ IS TABLE of dept%ROWTYPE
        INDEX BY BINARY_INTEGER;
    t_dept          DEPTTAB_TYP;
    t_dept_max      INTEGER := 1;
    r_emp           EMPREC_TYP;

    FUNCTION get_dept_name (
        p_deptno    IN NUMBER
    ) RETURN VARCHAR2
    IS
    BEGIN
        FOR i IN 1..t_dept_max LOOP
            IF p_deptno = t_dept(i).deptno THEN
                RETURN t_dept(i).dname;
            END IF;
        END LOOP;
        RETURN 'Unknown';
    END;

    FUNCTION open_emp_by_dept(
        p_deptno    IN emp.deptno%TYPE
    ) RETURN EMP_REFCUR
    IS
        emp_by_dept EMP_REFCUR;
    BEGIN
        OPEN emp_by_dept FOR SELECT empno, ename FROM emp
            WHERE deptno = p_deptno;
        RETURN emp_by_dept;
    END;

    PROCEDURE fetch_emp (
        p_refcur    IN OUT SYS_REFCURSOR
    )
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
        DBMS_OUTPUT.PUT_LINE('-----    -------');
        LOOP
            FETCH p_refcur INTO r_emp;
            EXIT WHEN p_refcur%NOTFOUND;
            DBMS_OUTPUT.PUT_LINE(r_emp.empno || '     ' || r_emp.ename);
        END LOOP;
    END;

    PROCEDURE close_refcur (
        p_refcur    IN OUT SYS_REFCURSOR
    )
    IS
    BEGIN
        CLOSE p_refcur;
    END;
BEGIN
    OPEN dept_cur;
    LOOP
        FETCH dept_cur INTO t_dept(t_dept_max);
        EXIT WHEN dept_cur%NOTFOUND;
        t_dept_max := t_dept_max + 1;
```

```
    END LOOP;
    CLOSE dept_cur;
    t_dept_max := t_dept_max - 1;
END emp_rpt;
```

此程序包包含一个初始化节，该节使用私有静态游标 DEPT_CUR 来装入私有关联数组变量 T_DEPT。在函数 GET_DEPT_NAME 中，将 T_DEPT 用作部门名称查找表。函数 OPEN_EMP_BY_DEPT 返回一个 REF CURSOR 变量，此变量是给定部门的职员编号和姓名的结果集。然后，可以将这个 REF CURSOR 变量传递给过程 FETCH_EMP，以便检索和列示该结果集的各行。最后，可以使用过程 CLOSE_REFCUR 来关闭与此结果集相关联的 REF CURSOR 变量。

以下匿名块运行程序包函数和过程。在声明节中，使用公用的 REF CURSOR 类型 EMP_REFCUR 来声明游标变量 V_EMP_CUR。V_EMP_CUR 包含一个指针，该指针指向在程序包函数和过程之间传递的结果集。

```
DECLARE
    v_deptno        dept.deptno%TYPE DEFAULT 30;
    v_emp_cur       emp_rpt.EMP_REFCUR;
BEGIN
    v_emp_cur := emp_rpt.open_emp_by_dept(v_deptno);
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #' || v_deptno ||
        ': ' || emp_rpt.get_dept_name(v_deptno));
    emp_rpt.fetch_emp(v_emp_cur);
    DBMS_OUTPUT.PUT_LINE('**********************');
    DBMS_OUTPUT.PUT_LINE(v_emp_cur%ROWCOUNT || ' rows were retrieved');
    emp_rpt.close_refcur(v_emp_cur);
END;
```

此匿名块生成以下样本输出:

```
EMPLOYEES IN DEPT #30: SALES
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7698     BLAKE
7844     TURNER
7900     JAMES
**********************
6 rows were retrieved
```

以下匿名块说明另一种实现同一结果的方法。即，将逻辑直接编码到匿名块中，而不是使用程序包过程 FETCH_EMP 和 CLOSE_REFCUR。注意，这里使用公用记录类型 EMPREC_TYP 来声明记录变量 R_EMP。

```
DECLARE
    v_deptno        dept.deptno%TYPE DEFAULT 30;
    v_emp_cur       emp_rpt.EMP_REFCUR;
    r_emp           emp_rpt.EMPREC_TYP;
BEGIN
    v_emp_cur := emp_rpt.open_emp_by_dept(v_deptno);
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #' || v_deptno ||
        ': ' || emp_rpt.get_dept_name(v_deptno));
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH v_emp_cur INTO r_emp;
        EXIT WHEN v_emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(r_emp.empno || '     ' ||
            r_emp.ename);
    END LOOP;
```

```
    DBMS_OUTPUT.PUT_LINE('**********************');
    DBMS_OUTPUT.PUT_LINE(v_emp_cur%ROWCOUNT || ' rows were retrieved');
    CLOSE v_emp_cur;
END;
```

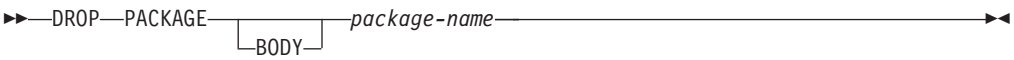
此匿名块生成以下样本输出:

```
EMPLOYEES IN DEPT #30: SALES
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7698     BLAKE
7844     TURNER
7900     JAMES
**********************
6 rows were retrieved
```

删除程序包 (PL/SQL)

可以删除不再需要的程序包。另外，如果要复用该程序包，那么可以选择只删除程序包主体。

语法



描述

BODY
指定只删除程序包主体。如果省略此关键字，那么将同时删除程序包规范和程序包主体。

package-name
指定程序包的名称。

示例

以下示例说明如何只删除名为 EMP_ADMIN 的程序包的主体:

```
DROP PACKAGE BODY emp_admin
```

以下示例说明如何同时删除程序包的规范和主体:

```
DROP PACKAGE emp_admin
```

第 3 部分 内置模块

内置模块提供易于使用的程序化接口，用于执行各种有用的操作。

例如，可使用内置模块来执行下列功能:

- 通过连接发送及接收消息和警报。
- 写入和读取操作系统的文件系统上的文件和目录。
- 生成包含各种监视器信息的报告。

可通过基于 SQL 的应用程序、DB2 命令行或命令脚本调用内置模块。

内置模块按数据库代码页设置变换字符串数据。

对于下列产品版本，内置模块不受支持:

- DB2 Express-C
- DB2 个人版

第 22 章 DBMS_ALERT 模块

DBMS_ALERT 模块提供一组过程，用于注册警报、发送警报和接收警报。

警报存储在 SYSTOOLS.DBMS_ALERT_INFO 中，SYSTOOLS.DBMS_ALERT_INFO 是您第一次对每个数据库引用此模块时在 SYSTOOLSPACE 中创建的。

此模块的模式为 SYSIBMADM。

DBMS_ALERT 模块包括以下内置例程。

表 11. DBMS_ALERT 模块中可用的内置例程

例程名称	描述
REGISTER 过程	注册当前会话以接收指定的警报。
REMOVE 过程	除去指定的警报的注册。
REMOVEALL 过程	除去所有警报的注册。
SIGNAL 过程	用信号表明指定警报的出现。
SET_DEFAULTS 过程	设置 WAITONE 和 WAITANY 过程的轮询时间间隔。
WAITANY 过程	等待任何已注册警报出现。
WAITONE 过程	等待指定的警报出现。

用法说明

当您想要针对特定事件发送警报时，DBMS_ALERT 模块中的过程非常有用。例如，您可能想要在因为一个或多个表发生更改而导致触发器激活时发送警报。

DBMS_ALERT 模块要求将数据库配置参数 CUR_COMMIT 设置为 ON。

示例

触发器 TRIG1 激活时，从连接 1 向连接 2 发送警报。首先，创建表和触发器。

```
CREATE TABLE T1 (C1 INT)@

CREATE TRIGGER TRIG1
AFTER INSERT ON T1
REFERENCING NEW AS NEW
FOR EACH ROW
BEGIN ATOMIC
CALL DBMS_ALERT.SIGNAL( 'trig1', NEW.C1 );
END@
```

从连接 1 发出 INSERT 语句。

```
INSERT INTO T1 values (10)@
-- Commit to send messages to the listeners (required in early program)
CALL DBMS_ALERT.COMMIT()@
```

从连接 2 进行注册以接收警报 trig1 并等待该警报出现。

```
CALL DBMS_ALERT.REGISTER('trig1')@
CALL DBMS_ALERT.WAITONE('trig1', ?, ?, 5)@
```

此示例生成以下输出:

```
Value of output parameters
--------------------------
Parameter Name  : MESSAGE
Parameter Value : -

Parameter Name  : STATUS
Parameter Value : 1

Return Status = 0
```

REGISTER 过程 - 注册以接收指定警报

REGISTER 过程注册当前会话以接收指定的警报。

语法

过程参数

name
类型为 VARCHAR(128) 的输入参数，用于指定警报的名称。

授权

对 DBMS_ALERT 模块的 EXECUTE 特权。

示例

使用 REGISTER 过程以针对警报 alert_test 进行注册，然后等待信号。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_name    VARCHAR(30) DEFAULT 'alert_test';
  DECLARE v_msg     VARCHAR(80);
  DECLARE v_status  INTEGER;
  DECLARE v_timeout INTEGER DEFAULT 5;
  CALL DBMS_ALERT.REGISTER(v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
  CALL DBMS_ALERT.WAITONE(v_name , v_msg , v_status , v_timeout);
  CALL DBMS_OUTPUT.PUT_LINE('Alert name   : ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
  CALL DBMS_ALERT.REMOVE(v_name);
END@

CALL proc1@
```

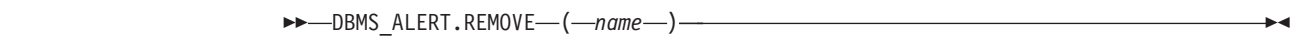
此示例生成以下输出:

```
Waiting for signal...
Alert name   : alert_test
Alert status : 1
```

REMOVE 过程 - 除去对指定警报的注册

REMOVE 过程从当前会话中除去对指定警报的注册。

语法



过程参数

name
类型为 VARCHAR(128) 的输入参数，用于指定警报的名称。

授权

对 DBMS_ALERT 模块的 EXECUTE 特权。

示例

使用 REMOVE 过程除去警报 alert_test。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_name    VARCHAR(30) DEFAULT 'alert_test';
  DECLARE v_msg     VARCHAR(80);
  DECLARE v_status  INTEGER;
  DECLARE v_timeout INTEGER DEFAULT 5;
  CALL DBMS_ALERT.REGISTER(v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
  CALL DBMS_ALERT.WAITONE(v_name , v_msg , v_status , v_timeout);
  CALL DBMS_OUTPUT.PUT_LINE('Alert name   : ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
  CALL DBMS_ALERT.REMOVE(v_name);
END@

CALL proc1@
```

此示例生成以下输出:

```
Waiting for signal...
Alert name   : alert_test
Alert status : 1
```

REMOVEALL 过程 - 除去对所有警报的注册

REMOVEALL 过程从当前会话中除去对所有警报的注册。

语法

授权

对 DBMS_ALERT 模块的 EXECUTE 特权。

示例

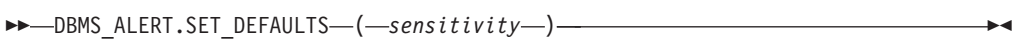
使用 REMOVEALL 过程除去对所有警报的注册。

```
  CALL DBMS_ALERT.REMOVEALL@
```

SET_DEFAULTS - 设置 WAITONE 和 WAITANY 的轮询时间间隔

SET_DEFAULTS 过程设置 WAITONE 和 WAITANY 过程使用的轮询时间间隔。

语法



过程参数

sensitivity
类型为 INTEGER 的输入参数，用于指定 WAITONE 和 WAITANY 过程检查信号的时间间隔（以秒计）。如果未指定值，那么缺省情况下时间间隔为 1 秒。

授权

对 DBMS_ALERT 模块的 EXECUTE 特权。

示例

使用 SET_DEFAULTS 过程来指定 WAITONE 和 WAITANY 过程的轮询时间间隔。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_name    VARCHAR(30) DEFAULT 'alert_test';
  DECLARE v_msg     VARCHAR(80);
  DECLARE v_status  INTEGER;
  DECLARE v_timeout INTEGER DEFAULT 20;
  DECLARE v_polling INTEGER DEFAULT 3;
  CALL DBMS_ALERT.REGISTER(v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
  CALL DBMS_ALERT.SET_DEFAULTS(v_polling);
  CALL DBMS_OUTPUT.PUT_LINE('Polling interval: ' || v_polling);
  CALL DBMS_ALERT.WAITONE(v_name , v_msg , v_status , v_timeout);
  CALL DBMS_ALERT.REMOVE(v_name);
END@

CALL proc1@
```

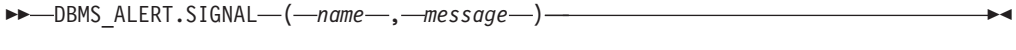
此示例生成以下输出:

```
Polling interval : 3
```

SIGNAL 过程 - 用信号表明指定警报的出现

SIGNAL 过程用信号表明指定警报的出现。信号包括随警报传递的消息。发出 SIGNAL 调用时，该消息将分发至侦听器（针对警报注册的进程）。

语法



过程参数

name
类型为 VARCHAR(128) 的输入参数，用于指定警报的名称。

message
类型为 VARCHAR(32672) 的输入参数，用于指定随此警报传递的信息。发生警报时，WAITANY 或 WAITONE 过程可能会返回此消息。

授权

对 DBMS_ALERT 模块的 EXECUTE 特权。

示例

使用 SIGNAL 过程来针对警报 alert_test 的实例发出信号。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_name   VARCHAR(30) DEFAULT 'alert_test';
  CALL DBMS_ALERT.SIGNAL(v_name,'This is the message from ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Issued alert for ' || v_name);
END@

CALL proc1@
```

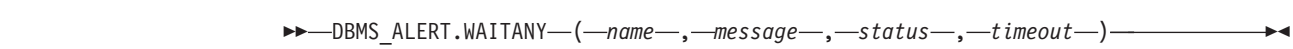
此示例生成以下输出:

```
Issued alert for alert_test
```

WAITANY 过程 - 等待任何已注册警报

WAITANY 过程等待任何已注册警报出现。

语法



过程参数

name
类型为 VARCHAR(128) 的输出参数，其中包含警报的名称。

message
类型为 VARCHAR(32672) 的输出参数，其中包含 SIGNAL 过程发送的消息。

status
类型为 INTEGER 的输出参数，其中包含过程返回的状态码。可能为以下值:

0 发生了警报。

1 发生了超时。

timeout
类型为 INTEGER 的输入参数，用于指定等待警报的时间量（以秒计）。

授权

对 DBMS_ALERT 模块的 EXECUTE 特权。

示例

通过一个连接运行 CLP 脚本 waitany.clp 以接收任何已注册警报。

```
waitany.clp:

SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_name    VARCHAR(30);
  DECLARE v_msg     VARCHAR(80);
  DECLARE v_status  INTEGER;
  DECLARE v_timeout INTEGER DEFAULT 20;
  CALL DBMS_ALERT.REGISTER('alert_test');
  CALL DBMS_ALERT.REGISTER('any_alert');
  CALL DBMS_OUTPUT.PUT_LINE('Registered for alert alert_test and any_alert');
  CALL DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
  CALL DBMS_ALERT.WAITANY(v_name , v_msg , v_status , v_timeout);
  CALL DBMS_OUTPUT.PUT_LINE('Alert name : ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Alert msg : ' || v_msg);
  CALL DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Alert timeout: ' || v_timeout || ' seconds');
  CALL DBMS_ALERT.REMOVEALL;
END@

call proc1@
```

通过另一个连接运行脚本 signal.clp 以对警报 any_alert 发出信号。

```
signal.clp:

SET SERVEROUTPUT ON@

CREATE PROCEDURE proc2
  BEGIN
  DECLARE v_name VARCHAR(30) DEFAULT 'any_alert';
  CALL DBMS_ALERT.SIGNAL(v_name,'This is the message from ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Issued alert for ' || v_name);
END@

CALL proc2@
```

脚本 signal.clp 生成以下输出:

```
Issued alert for any_alert
```

脚本 waitany.clp 生成以下输出:

```
Registered for alert alert_test and any_alert
Waiting for signal...
Alert name : any_alert
Alert msg : This is the message from any_alert
Alert status : 0
Alert timeout: 20 seconds
```

用法说明

如果调用 WAITANY 过程时未注册任何警报，那么该过程会返回 SQL0443N。

WAITONE 过程 - 等待指定警报

WAITONE 过程等待指定警报出现。

语法

过程参数

name
类型为 VARCHAR(128) 的输入参数，用于指定警报的名称。

message
类型为 VARCHAR(32672) 的输出参数，其中包含 SIGNAL 过程发送的消息。

status
类型为 INTEGER 的输出参数，其中包含过程返回的状态码。可能为以下值:

0 发生了警报。

1 发生了超时。

timeout
类型为 INTEGER 的输入参数，用于指定等待所指定警报的时间量（以秒计）。

授权

对 DBMS_ALERT 模块的 EXECUTE 特权。

示例

运行 CLP 脚本 `waitone.clp` 以接收警报 alert_test。

`waitone.clp`:

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_name    VARCHAR(30) DEFAULT 'alert_test';
  DECLARE v_msg     VARCHAR(80);
  DECLARE v_status  INTEGER;
  DECLARE v_timeout INTEGER DEFAULT 20;
  CALL DBMS_ALERT.REGISTER(v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
  CALL DBMS_ALERT.WAITONE(v_name , v_msg , v_status , v_timeout);
  CALL DBMS_OUTPUT.PUT_LINE('Alert name : ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Alert msg : ' || v_msg);
  CALL DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Alert timeout: ' || v_timeout || ' seconds');
  CALL DBMS_ALERT.REMOVE(v_name);
END@

CALL proc1@
```

通过另一连接运行脚本 `signalalert.clp` 以针对警报 alert_test 发出信号。

`signalalert.clp`:

```
SET SERVEROUTPUT ON@
```

```
CREATE PROCEDURE proc2
BEGIN
  DECLARE v_name VARCHAR(30) DEFAULT 'alert_test';
  CALL DBMS_ALERT.SIGNAL(v_name,'This is the message from ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Issued alert for ' || v_name);
END@

CALL proc2@
```

脚本 signalalert.clp 生成以下输出:

```
Issued alert for alert_test
```

脚本 waitone.clp 生成以下输出:

```
Waiting for signal...
Alert name : alert_test
Alert msg : This is the message from alert_test
Alert status : 0
Alert timeout: 20 seconds
```

第 23 章 DBMS_DDL 模块

DBMS_DDL 模块提供了用于模糊化 DDL 对象（如例程、触发器、视图或 PL/SQL 程序包）的功能。通过模糊化，可在不暴露过程逻辑的情况下，将 SQL 对象部署到数据库。

这些对象的 DDL 语句将同时在供应商提供的安装脚本以及 DB2 目录中模糊化。

此模块的模式为 SYSIBMADM。

DBMS_DDL 模块包括以下例程。

表 12. DBMS_DDL 模块中可用的内置例程

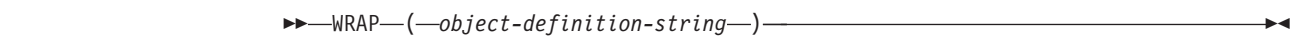
例程名称	描述
WRAP 函数	生成作为自变量而提供的 DDL 语句的模糊化版本。
CREATE_WRAPPED 过程	在数据库中以模糊化格式部署 DDL 语句。

WRAP 函数 - 模糊化 DDL 语句

WRAP 函数将可读的 DDL 语句变换为模糊化 DDL 语句。

语法

在模糊化 DDL 语句中，过程逻辑和嵌入式 SQL 语句按如下方式进行加密：无法轻松抽取逻辑中的知识产权。如果 DDL 语句对应于外部例程定义，那么将对参数列表后面的部分进行编码。



参数

object-definition-string

包含 DDL 语句文本的 CLOB(2M) 类型的字符串，可以为下列一项内容 (SQLSTATE 5UA0O):

- create procedure
- create function
- create package (PL/SQL)
- create package body (PL/SQL)
- create trigger
- create view
- alter module add function
- alter module publish function
- alter module add procedure

- alter module publish procedure

结果是 CLOB(2M) 类型的字符串，其中包含输入语句的已编码版本。编码由原始语句的前缀组成，并且包括例程签名或触发器、视图或包名称（后跟关键字 WRAPPED）。此关键字后跟有关执行函数的应用程序服务器的信息。信息的格式为 *pppvvrrm*，其中:

- *ppp* 使用字母 SQL 将产品标识为 DB2 Database for Linux®, UNIX®, and Windows®
- *vv* 是一个两位数版本标识，如“09”
- *rr* 是一个两位数发行版标识，如“07”。
- *m* 是一个一位字符修改级别标识，如“0”。

例如，版本 9.7 的修订包 2 将标识为“SQL09072”。此应用程序服务器信息后跟一个字母字符串（a-z 和 A-Z）、数字 (0-9)、下划线和冒号。除了在模糊化后仍可保持可读的前缀之外，不会对输入语句进行任何语法检查。

已编码的 DDL 语句长度通常超过该语句的纯文本格式。如果结果超过 SQL 语句的最大长度，那么将出现错误 (SQLSTATE 54001)。

注: 语句的编码意味着模糊化内容，并且不应被视为强加密形式。

授权

对 DBMS_DDL 模块的 EXECUTE 特权

示例

- 生成根据每小时工资来计算年薪（规定每周 40 小时）的函数的模糊化版本

```
VALUES(DBMS_DDL.WRAP('CREATE FUNCTION ' ||
                     'salary(wage DECFLOAT) ' ||
                     'RETURNS DECFLOAT ' ||
                     'RETURN wage * 40 * 52'))
The result of the previous statement would be something of the form:

CREATE FUNCTION salary(wage DECFLOAT) WRAPPED SQL09072 obfuscated-text
```

- 生成一个用于设置复杂缺省值的触发器的模糊化格式

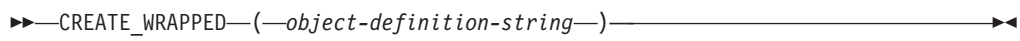
```
VALUES(DBMS_DDL.WRAP('CREATE OR REPLACE TRIGGER ' ||
                      'trg1 BEFORE INSERT ON emp ' ||
                      'REFERENCING NEW AS n ' ||
                      'FOR EACH ROW ' ||
                      'WHEN (n.bonus IS NULL) ' ||
                      'SET n.bonus = n.salary * .04'))

The result of the previous statement would be something of the form:

CREATE OR REPLACE TRIGGER trg1 WRAPPED SQL09072 obfuscated-text
```

CREATE_WRAPPED 过程 – 部署模糊化对象

CREATE_WRAPPED 过程会将纯文本 DDL 对象定义变换为模糊化 DDL 对象定义，然后在数据库中部署对象。

语法

在模糊化 DDL 语句中，过程逻辑和嵌入式 SQL 语句按如下方式进行编码：无法轻松抽取逻辑中的知识产权。



参数

object-definition-string

包含 DDL 语句文本的 CLOB(2M) 类型的字符串，可以为下列一项内容 (SQLSTATE 5UA0O):

- create procedure
- create function
- create package (PL/SQL)
- create package body (PL/SQL)
- create trigger
- create view
- alter module add function
- alter module publish function
- alter module add procedure
- alter module publish procedure

该过程将输入变换为模糊化 DDL 语句字符串，然后动态执行该 DDL 语句。正在使用在调用时生效的专用寄存器值（如 PATH 和 CURRENT SCHEMA）以及当前调用者的权限。

编码由原始语句的前缀组成，并且包括例程签名或触发器、视图或包名称（后跟关键字 **WRAPPED**）。此关键字后跟有关执行过程的应用程序服务器的信息。信息的格式为“*pppvvrrm*”，其中:

- *ppp* 使用字母 SQL 将产品标识为 DB2 Database for Linux®, UNIX®, and Windows®
- *vv* 是一个两位数版本标识，如“09”
- *rr* 是一个两位数发行版标识，如“07”。
- *m* 是一个一位字符修改级别标识，如“0”。

例如，版本 9.7 的修订包 2 将标识为“SQL09072”。此应用程序服务器信息后跟一个字母字符串（a-z 和 A-Z）、数字 (0-9)、下划线和冒号。除了在模糊化后仍可保持可读的前缀之外，不会对输入语句进行任何语法检查。

已编码的 DDL 语句长度通常超过该语句的纯文本格式。如果结果超过 SQL 语句的最大长度，那么将出现错误 (SQLSTATE 54001)。

注: 语句的编码意味着模糊化内容，并且不应被视为强加密形式。

授权

对 DBMS_DDL 模块的 EXECUTE 特权。

示例

- 创建一个模糊化函数，用以根据每小时工资来计算年薪（规定每周 40 小时）

```
CALL DBMS_DDL.CREATE_WRAPPED('CREATE FUNCTION ' ||
                    'salary(wage DECFLOAT) ' ||
                    'RETURNS DECFLOAT ' ||
                    'RETURN wage * 40 * 52');
SELECT text FROM SYSCAT.ROUTINES
 WHERE routinename = 'SALARY'
    AND routineschema = CURRENT SCHEMA;
```

在成功执行 CALL 语句后，与例程“SALARY&”相对应的行的 SYSCAT.ROUTINES.TEXT 列将类似于以下格式:

```
CREATE FUNCTION salary(wage DECFLOAT) WRAPPED SQL09072 obfuscated-text
```

- 创建一个用于设置复杂缺省值的模糊化触发器

```
CALL DBMS_DDL.CREATE_WRAPPED('CREATE OR REPLACE TRIGGER ' ||
                    'trg1 BEFORE INSERT ON emp ' ||
                    'REFERENCING NEW AS n ' ||
                    'FOR EACH ROW ' ||
                    'WHEN (n.bonus IS NULL) ' ||
                    'SET n.bonus = n.salary * .04');
SELECT text FROM SYSCAT.TRIGGERS
 WHERE trigname = 'TRG1'
   AND trigschema = CURRENT SCHEMA;
```

在成功执行 CALL 语句后，与触发器“TRG1”相对应的行的 SYSCAT.TRIGGERS.TEXT 列将类似于以下格式:

```
CREATE OR REPLACE TRIGGER trg1 WRAPPED SQL09072  obfuscated-text
```

第 24 章 DBMS_JOB 模块

DBMS_JOB 模块提供用于创建、调度和管理作业的过程。

DBMS_JOB 模块提供“管理任务调度程序”（ATS）的替代界面。通过将任务添加至 ATS 创建作业。实际任务名称通过使 DBMS_JOB.TASK_NAME_PREFIX 过程名称与指定作业标识并置构造，如 SAMPLE_JOB_TASK_1，其中 1 是作业标识。

作业运行先前存储在数据库中的存储过程。SUBMIT 过程用于创建和存储作业定义。系统会对每个作业指定作业标识，并且指定其关联存储过程和用于描述作业运行时间和方式的属性。

第一次在数据库中运行 SUBMIT 过程，如果必要，会创建 SYSTOOLSPACE 表空间。

要对 DBMS_JOB 例程启用作业调度，请运行:

```
db2set DB2_ATS_ENABLE=1
```

作业的运行时间和频率取决于两个交互参数: **next_date** 和 **interval**。**next_date** 参数是日期时间值，用于指定下次执行作业的日期和时间。**interval** 参数是字符串，包含求值为日期时间值的日期函数。在执行作业之前，会对 **interval** 参数中的表达式进行求值，产生的值将替换随作业存储的 **next_date** 值。然后执行作业。在此方式下，每次执行作业之前会重新对 **interval** 中的表达式求值，以便提供对应下次执行的 **next_date** 日期和时间。

按 **next_date** 参数指定的那样，已调度作业的第一次运行应设置为在当前时间之后经过至少 5 分钟开始，并且运行每个作业之间的时间间隔应至少为 5 分钟。

此模块的模式为 SYSIBMADM。

DBMS_JOB 模块包括以下内置例程。

表 13. DBMS_JOB 模块中可用的内置例程

例程名称	描述
BROKEN 过程	指定所给定作业是已中断还是未中断。
CHANGE 过程	更改作业的参数。
INTERVAL 过程	通过作业每次运行时重新计算的日期函数来设置执行频率。此值成为下一次执行的日期和时间。
NEXT_DATE 过程	设置下次要运行作业的日期和时间。
REMOVE 过程	从数据库中删除作业定义。
RUN 过程	强制执行作业，即使该作业被标记为中断。
SUBMIT 过程	创建作业并将作业定义存储在数据库中。
WHAT 过程	更改作业运行的存储过程。

表 14. DBMS_JOB 模块中可用的内置常量

常量名称	描述
ANY_INSTANCE	DBMS_JOB 例程的实例参数唯一支持的值。
TASK_NAME_PREFIX	此常量包含字符串，该字符串用作构造管理任务调度程序的任务名称的前缀。

用法说明

通过 DBMS_JOB 模块对每个数据库提交第一个作业时，会执行“管理任务调度程序”设置:

1. 创建 SYSTOOLSPACE 表空间（如果该表空间不存在）;
2. 创建 ATS 表和视图，如 SYSTOOLS.ADMIN_TASK_LIST。

要列示已调度作业，请运行:

```
db2 SELECT * FROM systools.admin_task_list
      WHERE name LIKE DBMS_JOB.TASK_NAME_PREFIX || '_%'
```

要查看作业执行状态，请运行:

```
db2 SELECT * FROM systools.admin_task_status
      WHERE name LIKE DBMS_JOB.TASK_NAME_PREFIX || '_%'
```

示例

示例 1: 以下示例使用存储过程 job_proc。此存储过程仅将时间戳记插入到包含单个 VARCHAR 列的 jobrun 表中。

```
CREATE TABLE jobrun (
    runtime         VARCHAR(40)
)@

CREATE OR REPLACE PROCEDURE job_proc
BEGIN
    INSERT INTO jobrun VALUES ('job_proc run at ' || TO_CHAR(SYSDATE,
        'yyyy-mm-dd hh24:mi:ss'));
END@
```

此示例生成以下输出:

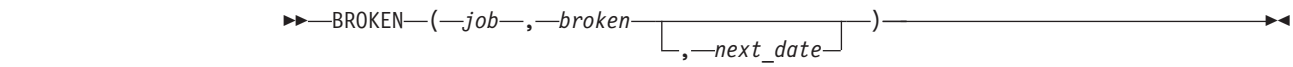
```
CREATE TABLE jobrun ( runtime         VARCHAR(40) )
DB20000I  The SQL command completed successfully.

CREATE OR REPLACE PROCEDURE job_proc
BEGIN
    INSERT INTO jobrun VALUES ('job_proc run at ' || TO_CHAR(SYSDATE,
        'yyyy-mm-dd hh24:mi:ss'));
END
DB20000I  The SQL command completed successfully.
```

BROKEN 过程 - 将作业的状态设置为已中断或未中断

BROKEN 过程将作业的状态设置为已中断或未中断。

除非使用 RUN 过程，否则不能执行已中断作业。

语法



参数

job

类型为 DECIMAL(20) 的输入参数，用于指定要设置为已中断或未中断的作业的标识。

broken

类型为 BOOLEAN 的输入参数，用于指定作业状态。如果设置为“true”，那么作业状态设置为 broken（已中断）。如果设置为“false”，那么作业状态设置为 not broken（未中断）。除非通过 RUN 过程，否则不能运行已中断作业。

next_date

类型为 DATE 的可选输入参数，用于指定作业的下次运行日期和时间。缺省值为 SYSDATE。

授权

对 DBMS_JOB 模块的 EXECUTE 特权。

示例

示例 1: 将作业标识为 104 的作业的状态设置为已中断:

```
CALL DBMS_JOB.BROKEN(104,true);
```

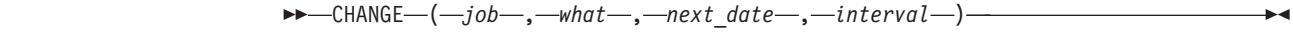
示例 2: 将状态更改为未中断:

```
CALL DBMS_JOB.BROKEN(104,false);
```

CHANGE 过程 - 修改作业属性

CHANGE 过程修改特定作业属性，包括可执行 SQL 语句、作业的下次运行日期和时间以及运行频率。

语法



参数

job

类型为 DECIMAL(20) 的输入参数，用于指定带有要修改的属性的作业的标识。

what

类型为 VARCHAR(1024) 的输入参数，用于指定可执行的 SQL 语句。如果要使现有值保持不变，请将此参数设置为 NULL。

next_date

类型为 TIMESTAMP(0) 的输入参数，用于指定作业的下次运行日期和时间。如果要使现有值保持不变，请将此参数设置为 NULL。

interval
类型为 VARCHAR(1024) 的输入参数，用于指定求值为作业的下次运行日期和时间的日期函数。如果要使现有值保持不变，请将此参数设置为 NULL。

授权

对 DBMS_JOB 模块的 EXECUTE 特权。

示例

示例 1: 将作业的下次运行时间更改为 2009 年 12 月 13 日。其他参数保持不变。

```
CALL DBMS_JOB.CHANGE(104,NULL,TO_DATE('13-DEC-09','DD-MON-YY'),NULL);
```

INTERVAL 过程 - 设置运行频率

INTERVAL 过程设置作业的运行频率。

语法

参数

job
类型为 DECIMAL(20) 的输入参数，用于指定要更改其运行频率的作业的标识。

interval
类型为 VARCHAR(1024) 的输入参数，用于指定求值为作业的下次运行日期和时间的日期函数。

授权

对 DBMS_JOB 模块的 EXECUTE 特权。

示例

示例 1: 将作业的运行频率更改为每周运行一次:

```
CALL DBMS_JOB.INTERVAL(104,'SYSDATE + 7');
```

NEXT_DATE 过程 - 设置作业的运行日期和时间

NEXT_DATE 过程设置作业的下次运行日期和时间。

语法

参数

job
类型为 DECIMAL(20) 的输入参数，用于指定要修改其下次运行日期的作业的标识。

next_date
类型为 TIMESTAMP(0) 的输入参数，用于指定作业的下次运行日期和时间。

授权

对 DBMS_JOB 模块的 EXECUTE 特权。

示例

示例 1: 将作业的下次运行时间更改为 2009 年 12 月 14 日:

```
CALL DBMS_JOB.NEXT_DATE(104, TO_DATE('14-DEC-09','DD-MON-YY'));
```

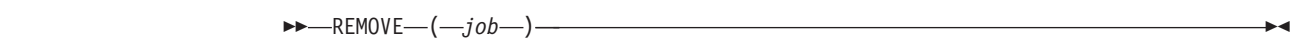
REMOVE 过程 - 从数据库中删除作业定义

REMOVE 过程从数据库中删除指定的作业。

为了将来再次执行作业，必须使用 SUBMIT 过程重新提交该作业。

注: 除去作业时，不会删除与该作业相关联的存储过程。

语法



参数

job
类型为 DECIMAL(20) 的输入参数，用于指定要从数据库中除去的作业的标识。

授权

对 DBMS_JOB 模块的 EXECUTE 特权。

示例

示例 1: 从数据库中除去作业:

```
CALL DBMS_JOB.REMOVE(104);
```

RUN 过程 - 强制已中断作业运行

RUN 过程强制作业运行，即使该作业处于已中断状态也是如此。

语法

参数

job
类型为 DECIMAL(20) 的输入参数，用于指定要运行的作业的标识。

授权

对 DBMS_JOB 模块的 EXECUTE 特权。

示例

示例 1: 强制作业运行:

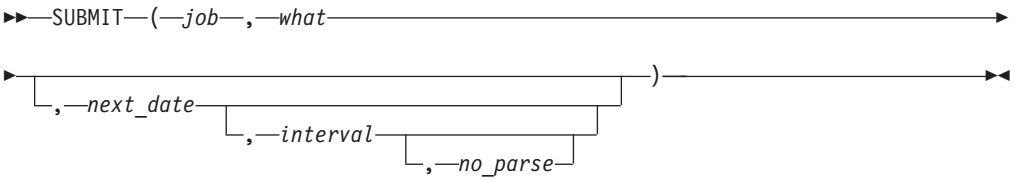
```
CALL DBMS_JOB.RUN(104);
```

SUBMIT 过程 - 创建作业定义并将其存储在数据库中

SUBMIT 过程创建作业定义并将其存储在数据库中。

作业由作业标识、要执行的存储过程、第一次执行作业的时间以及用于计算作业的下次运行日期和时间的日期函数组成。

语法



参数

job
类型为 DECIMAL(20) 的输出参数，用于指定已指定给作业的标识。

what
类型为 VARCHAR(1024) 的输入参数，用于指定动态可执行 SQL 语句的名称。

next_date
类型为 TIMESTAMP(0) 的可选输入参数，用于指定作业的下次运行日期和时间。缺省值为 SYSDATE。

interval
类型为 VARCHAR(1024) 的可选输入参数，用于指定一个日期函数，在对该函数求值后，将会提供下次执行后再次执行的日期和时间。如果 *interval* 设置为 NULL，那么该作业仅运行一次。NULL 是缺省值。

no_parse
类型为 BOOLEAN 的可选输入参数。如果设置为 true，那么不会在创建作业时对 SQL 语句进行语法检查；而是仅在第一次执行作业时执行语法检查。如果设置为 false，那么会在创建作业时对 SQL 语句进行语法检查。缺省值为 false。

授权

对 DBMS_JOB 模块的 EXECUTE 特权。

示例

示例 1: 以下示例使用存储过程 job_proc 创建作业。第一次执行该作业将大约在 5 分钟内，并且按 *interval* 参数的设置 (SYSDATE + 1) 一天运行一次。

```
SET SERVEROUTPUT ON@

BEGIN
  DECLARE jobid           INTEGER;
  CALL DBMS_JOB.SUBMIT(jobid,'CALL job_proc();',SYSDATE + 5 minutes, 'SYSDATE + 1');
  CALL DBMS_OUTPUT.PUT_LINE('jobid: ' || jobid);
END@
```

此命令的输出如下所示:

```
SET SERVEROUTPUT ON
DB20000I  SET SERVEROUTPUT 命令成功完成。

BEGIN
  DECLARE jobid           INTEGER;
  CALL DBMS_JOB.SUBMIT(jobid,'CALL job_proc();',SYSDATE + 5 minutes, 'SYSDATE + 1');
  CALL DBMS_OUTPUT.PUT_LINE('jobid: ' || jobid);
END
DB20000I  The SQL command completed successfully.

jobid: 1
```

WHAT 过程 - 更改作业运行的 SQL 语句

WHAT 过程更改由指定作业运行的 SQL 语句。

语法

参数

job

类型为 DECIMAL(20) 的输入参数，用于指定要对其更改动态可执行 SQL 语句的作业标识。

what

类型为 VARCHAR(1024) 的输入参数，用于指定动态执行的 SQL 语句。

授权

对 DBMS_JOB 模块的 EXECUTE 特权。

示例

示例 1: 将作业更改为运行 list_emp 过程:

```
CALL DBMS_JOB.WHAT(104,'list_emp;');
```

第 25 章 DBMS_LOB 模块

DBMS_LOB 模块能够操作大对象。

在描述各个过程和函数的下列各节中，如果大对象为 BLOB，那么长度和位移以字节计。如果大对象为 CLOB，那么长度和位移以字符计。

DBMS_LOB 模块最多支持 10 MB LOB 数据。

此模块的模式为 SYSIBMADM。

DBMS_LOB 模块包括下列例程。

表 15. DBMS_LOB 模块中可用的内置例程

例程名称	描述
APPEND 过程	将一个大对象追加至另一个大对象。
CLOSE 过程	关闭已打开的大对象。
COMPARE 函数	比较两个大对象。
CONVERTTOBLOB 过程	将字符数据转换为二进制。
CONVERTTOCLOB 过程	将二进制数据转换为字符。
COPY 过程	将一个大对象复制到另一个大对象。
ERASE 过程	擦除大对象。
GET_STORAGE_LIMIT 函数	获取大对象的存储限制。
GETLENGTH 函数	获取大对象的长度。
INSTR 函数	获取大对象中模式的第 n 个实例的位置（从位移开始）。
ISOPEN 函数	检查大对象是否已打开。
OPEN 过程	打开大对象。
READ 过程	读取大对象。
SUBSTR 函数	获取大对象的部分。
TRIM 过程	将大对象调整为指定长度。
WRITE 过程	将数据写至大对象。
WRITEAPPEND 过程	将缓冲区中的数据写至大对象结尾。

下表列示模块中可用的公用变量。

表 16. DBMS_LOB 公用变量

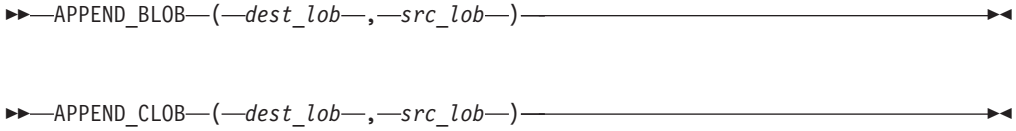
公用变量	数据类型	值
lob_readonly	INTEGER	0
lob_readwrite	INTEGER	1

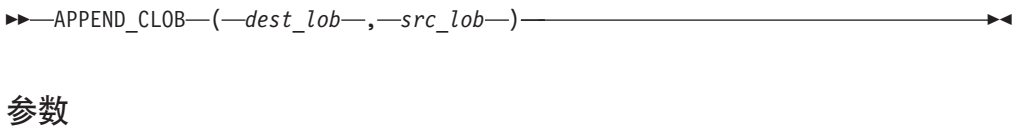
APPEND 过程 - 将一个大对象追加至另一个大对象

APPEND 过程能够将一个大对象追加至另一个大对象。

注: 两个大对象必须为同一类型。

语法





参数

dest_lob
类型为 BLOB(10M) 或 CLOB(10M) 的输入或输出参数，用于指定目标对象的大对象定位器。必须与 *src_lob* 为同一数据类型。

src_lob
类型为 BLOB(10M) 或 CLOB(10M) 的输入参数，用于指定源对象的大对象定位器。必须与 *dest_lob* 为同一数据类型。

授权

对 DBMS_LOB 模块的 EXECUTE 特权。

CLOSE 过程 - 关闭已打开的大对象

CLOSE 过程是空操作。

语法

参数

lob_loc
类型为 BLOB(10M) 或 CLOB(10M) 的输入或输出参数，用于指定要关闭的大对象的定位器。

授权

对 DBMS_LOB 模块的 EXECUTE 特权。

COMPARE 函数 - 比较两个大对象

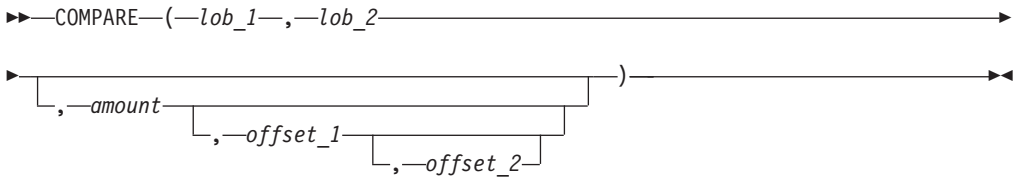
COMPARE 函数针对两个大对象的给定位移处的给定长度执行确切的每字节比较。

此函数返回:

- 零，如果两个大对象的指定位移处的指定长度完全相同
- 非零，如果这些对象不相同
- 空，如果 *amount*、*offset_1* 或 *offset_2* 小于零。

注: 被比较的大对象必须为同一数据类型。

语法



参数

lob_1
类型为 BLOB(10M) 或 CLOB(10M) 的输入参数，用于指定要比较的第一个大对象的定位器。必须与 *lob_2* 为同一数据类型。

lob_2
类型为 BLOB(10M) 或 CLOB(10M) 的输入参数，用于指定要比较的第二个大对象的定位器。必须与 *lob_1* 为同一数据类型。

amount
类型为 INTEGER 的可选输入参数。如果大对象的数据类型为 BLOB，那么将针对 *amount* 字节进行比较。如果大对象的数据类型为 CLOB，那么将针对 *amount* 字符进行比较。缺省值为大对象的最大大小。

offset_1
类型为 INTEGER 的可选输入参数，用于指定第一个大对象中开始比较的位置。第一个字节（或字符）为位移 1。缺省值为 1。

offset_2
类型为 INTEGER 的可选输入参数，用于指定第二个大对象中开始比较的位置。第一个字节（或字符）为位移 1。缺省值为 1。

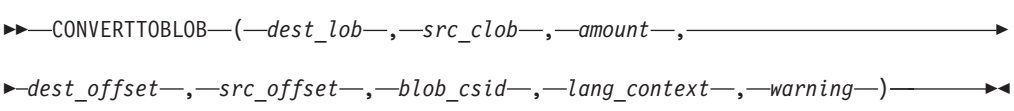
授权

对 DBMS_LOB 模块的 EXECUTE 特权。

CONVERTTOBLOB 过程 - 将字符数据转换为二进制

CONVERTTOBLOB 过程能够将字符数据转换为二进制。

语法



参数

dest_lob
类型为 BLOB(10M) 的输入或输出参数，用于指定字符数据要转换至其中的大对象定位器。

src_clob
类型为 CLOB(10M) 的输入参数，用于指定要转换的字符数据的大对象定位器。

amount
类型为 INTEGER 的输入参数，用于指定要转换的 *src_clob* 的字符数。

dest_offset
类型为 INTEGER 的输入或输出参数，用于指定目标 BLOB 中应开始写入源 CLOB 的位置（以字节计）。第一个字节为位移 1。

src_offset
类型为 INTEGER 的输入或输出参数，用于指定源 CLOB 中应开始至目标 BLOB 的转换的位置（以字符计）。第一个字符为位移 1。

blob_csid
类型为 INTEGER 的输入参数，用于指定目标 BLOB 的字符集标识。此值必须与数据库代码页相匹配。

lang_context
类型为 INTEGER 的输入参数，用于指定转换的语言上下文。此值必须为 0。

warning
类型为 INTEGER 的输出参数，始终返回 0。

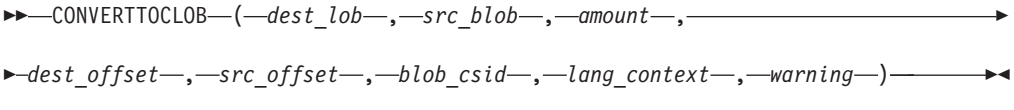
授权

对 DBMS_LOB 模块的 EXECUTE 特权。

CONVERTTOCLOB 过程 - 将二进制数据转换为字符

CONVERTTOCLOB 过程能够将二进制数据转换为字符。

语法



参数

dest_lob
类型为 CLOB(10M) 的输入或输出参数，用于指定二进制数据要转换至其中的大对象定位器。

src_clob
类型为 BLOB(10M) 的输入参数，用于指定要转换的二进制数据的大对象定位器。

amount
类型为 INTEGER 的输入参数，用于指定要转换的 *src_blob* 的字符数。

dest_offset
类型为 INTEGER 的输入或输出参数，用于指定目标 CLOB 中应开始写入源 BLOB 的位置（以字符计）。第一个字节为位移 1。

src_offset
类型为 INTEGER 的输入或输出参数，用于指定源 BLOB 中应开始至目标 CLOB 的转换的位置（以字节计）。第一个字符为位移 1。

blob_csid
类型为 INTEGER 的输入参数，用于指定源 BLOB 的字符集标识。此值必须与数据库代码页相匹配。

lang_context
类型为 INTEGER 的输入参数，用于指定转换的语言上下文。此值必须为 0。

warning
类型为 INTEGER 的输出参数，始终返回 0。

授权

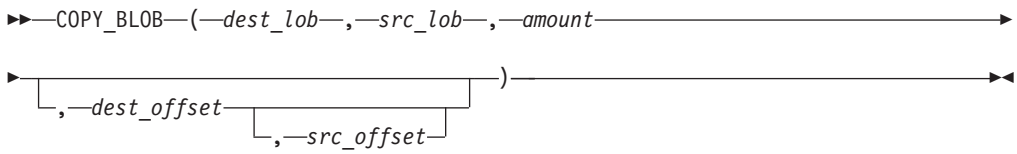
对 DBMS_LOB 模块的 EXECUTE 特权。

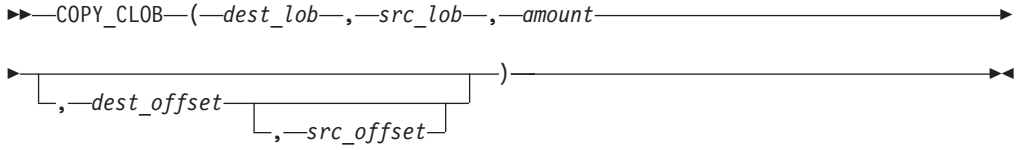
COPY 过程 - 将一个大对象复制到另一个大对象

COPY 过程能够将一个大对象复制到另一个大对象。

注: 源大对象和目标大对象必须为同一数据类型。

语法





参数

dest_lob
类型为 BLOB(10M) 或 CLOB(10M) 的输入或输出参数，用于指定 *src_lob* 要复制到的大对象的定位器。必须与 *src_lob* 为同一数据类型。

src_lob
类型为 BLOB(10M) 或 CLOB(10M) 的输入参数，用于指定要从中复制 *dest_lob* 的大对象的定位器。必须与 *dest_lob* 为同一数据类型。

amount
类型为 INTEGER 的输入参数，用于指定要复制的 *src_lob* 的字节或字符数。

dest_offset
类型为 INTEGER 的可选输入参数，用于指定目标大对象中应开始写入源大对象的位置。第一个位置为位移 1。缺省值为 1。

src_offset
类型为 INTEGER 的可选输入参数，用于指定源大对象中应开始复制到目标大对象的位置。第一个位置为位移 1。缺省值为 1。

授权

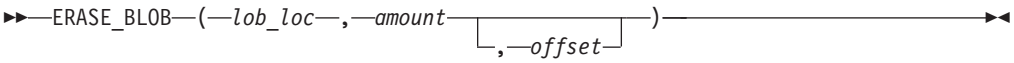
对 DBMS_LOB 模块的 EXECUTE 特权。

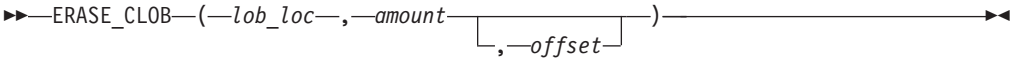
ERASE 过程 - 擦除大对象的部分

ERASE 过程能够擦除大对象的部分。

要擦除大对象意味着要将指定部分替换为零字节填充符（对于 BLOB）或空格（对于 CLOB）。大对象的实际大小不会改变。

语法





参数

lob_loc
类型为 BLOB(10M) 或 CLOB(10M) 的输入或输出参数，用于指定要擦除的大对象的定位器。

amount
类型为 INTEGER 的输入或输出参数，用于指定要擦除的字节或字符数。

offset
类型为 INTEGER 的可选输入参数，用于指定大对象中开始擦除的位置。第一个字节或字符为位置 1。缺省值为 1。

授权

对 DBMS_LOB 模块的 EXECUTE 特权。

GET_STORAGE_LIMIT 函数 - 返回对最大可允许大对象的限制

GET_STORAGE_LIMIT 函数返回对最大可允许大对象的限制。

该函数返回 INTEGER 值，该值反映此数据库中最大可允许大对象的大小。

语法

授权

对 DBMS_LOB 模块的 EXECUTE 特权。

GETLENGTH 函数 - 返回大对象的长度

GETLENGTH 函数返回大对象的长度。

该函数返回 INTEGER 值，该值反映大对象的长度，对于 BLOB，长度以字节计，对于 CLOB，长度以字符计。

语法

参数

lob_loc
类型为 BLOB(10M) 或 CLOB(10M) 的输入参数，用于指定要获取其长度的大对象的定位器。

授权

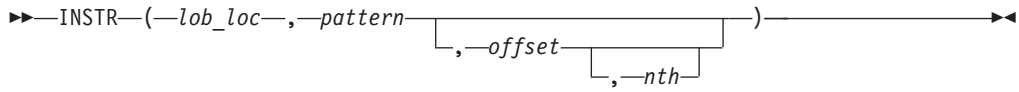
对 DBMS_LOB 模块的 EXECUTE 特权。

INSTR 函数 - 返回给定模式的第 *n* 个实例的位置

INSTR 函数返回大对象中给定模式的第 *n* 个实例的位置。

该函数返回 INTEGER 值，该值表示大对象中第 n 次（由 *nth* 指定）出现该模式的位置。此值从 *offset* 给定的位置开始。

语法



参数

lob_loc
类型为 BLOB 或 CLOB 的输入参数，用于指定要在其中搜索 *pattern* 的大对象的定位器。

pattern
类型为 BLOB(32767) 或 VARCHAR(32672) 的输入参数，用于指定要针对大对象进行比较的字节或字符的模式。注意，如果 *lob_loc* 为 BLOB，那么 *pattern* 必须为 BLOB；如果 *lob_loc* 为 CLOB，那么 *pattern* 必须为 VARCHAR。

offset
类型为 INTEGER 的可选输入参数，用于指定 *lob_loc* 中开始搜索 *pattern* 的位置。第一个字节或字符为位置 1。缺省值为 1。

nth
类型为 INTEGER 的可选参数，用于指定搜索 *pattern* 的次数（从 *offset* 给定的位置开始）。缺省值为 1。

授权

对 DBMS_LOB 模块的 EXECUTE 特权。

ISOPEN 函数 - 测试大对象是否已打开

ISOPEN 函数始终返回 INTEGER 值 1。

语法

参数

lob_loc
类型为 BLOB(10M) 或 CLOB(10M) 的输入参数，用于指定函数要测试的大对象的定位器。

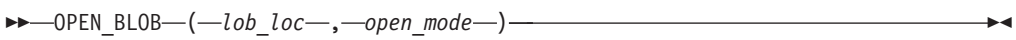
授权

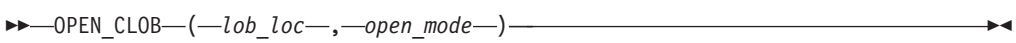
对 DBMS_LOB 模块的 EXECUTE 特权。

OPEN 过程 - 打开大对象

OPEN 过程是空操作。

语法





参数

lob_loc
类型为 BLOB(10M) 或 CLOB(10M) 的输入或输出参数，用于指定要打开的大对象的定位器。

open_mode
类型为 INTEGER 的输入参数，用于指定大对象的打开方式。设置为 0（lob_readonly）时以只读方式打开。设置为 1（lob_readwrite）时以读写方式打开。

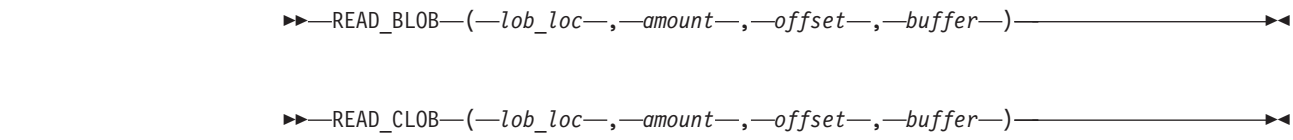
授权

对 DBMS_LOB 模块的 EXECUTE 特权。

READ 过程 - 读取大对象的部分

READ 过程能够将大对象的部分读取至缓冲区。

语法



参数

lob_loc
类型为 BLOB(10M) 或 CLOB(10M) 的输入参数，用于指定要读取的大对象的定位器。

amount
类型为 INTEGER 的输入或输出参数，用于指定要读取的字节或字符数。

offset
类型为 INTEGER 的输入参数，用于指定要开始读取的位置。第一个字节或字符为位置 1。

buffer
类型为 BLOB(32762) 或 VARCHAR(32672) 的输出参数，用于指定要接收大对象的变量。如果 *lob_loc* 为 BLOB，那么 *buffer* 必须为 BLOB。如果 *lob_loc* 为 CLOB，那么 *buffer* 必须为 VARCHAR。

授权

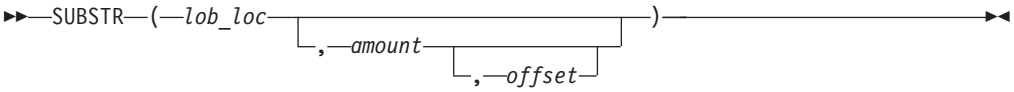
对 DBMS_LOB 模块的 EXECUTE 特权。

SUBSTR 函数 - 返回大对象的部分

SUBSTR 函数能够返回大对象的部分。

该函数返回函数读取的大对象的返回部分的 BLOB(32767)（对于 BLOB）或 VARCHAR（对于 CLOB）值。

语法



参数

lob_loc
类型为 BLOB(10M) 或 CLOB(10M) 的输入参数，用于指定要读取的大对象的定位器。

amount
类型为 INTEGER 的可选输入参数，用于指定要返回的字节或字符数。缺省值为 32,767。

offset
类型为 INTEGER 的可选输入参数，用于指定大对象中开始返回数据的位置。第一个字节或字符为位置 1。缺省值为 1。

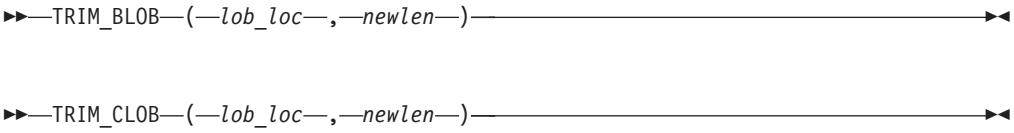
授权

对 DBMS_LOB 模块的 EXECUTE 特权。

TRIM 过程 - 将大对象截断为指定长度

TRIM 过程能够将大对象截断为指定长度。

语法



参数

lob_loc
类型为 BLOB(10M) 或 CLOB(10M) 的输入或输出参数，用于指定要调整的大对象的定位器。

newlen
类型为 INTEGER 的输入参数，用于指定大对象要调整至的新字节或字符数。

授权

对 DBMS_LOB 模块的 EXECUTE 特权。

WRITE 过程 - 将数据写至大对象

WRITE 过程能够将数据写至大对象。

大对象中位于指定位移并且具有给定长度的任何现有数据被缓冲区中的给定数据覆盖。

语法

```
►►—WRITE_BLOB—(—lob_loc—,—amount—,—offset—,—buffer—)———————————►◄

►►—WRITE_CLOB—(—lob_loc—,—amount—,—offset—,—buffer—)———————————►◄
```

参数

lob_loc
类型为 BLOB(10M) 或 CLOB(10M) 的输入或输出参数，用于指定要写入的大对象的定位器。

amount
类型为 INTEGER 的输入参数，用于指定 *buffer* 中要写至大对象的字节或字符数。

offset
类型为 INTEGER 的输入参数，用于指定在大对象开头开始写入操作的位移（以字节或字符计）。大对象的起始值为 1。

buffer
类型为 BLOB(32767) 或 VARCHAR(32672) 的输入参数，其中包含要写至大对象的数据。如果 *lob_loc* 为 BLOB，那么 *buffer* 必须为 BLOB。如果 *lob_loc* 为 CLOB，那么 *buffer* 必须为 VARCHAR。

授权

对 DBMS_LOB 模块的 EXECUTE 特权。

WRITEAPPEND 过程 - 将数据追加至大对象结尾

WRITEAPPEND 过程能够在大对象结尾添加数据。

语法

```
►►—WRITEAPPEND_BLOB—(—lob_loc—,—amount—,—buffer—)———————————►◄

►►—WRITEAPPEND_CLOB—(—lob_loc—,—amount—,—buffer—)———————————►◄
```

参数

lob_loc
类型为 BLOB 或 CLOB 的输入或输出参数，用于指定要对其追加数据的大对象的定位器。

amount
类型为 INTEGER 的输入参数，用于指定 *buffer* 中要追加至大对象的字节或字符数。

buffer

类型为 BLOB(32767) 或 VARCHAR(32672) 的输入参数，其中包含要追加至大对象的数据。如果 *lob_loc* 为 BLOB，那么 *buffer* 必须为 BLOB。如果 *lob_loc* 为 CLOB，那么 *buffer* 必须为 VARCHAR。

授权

对 DBMS_LOB 模块的 EXECUTE 特权。

第 26 章 DBMS_OUTPUT 模块

DBMS_OUTPUT 模块提供了一组过程，用于将消息（文本行）放到消息缓冲区中以及从消息缓冲区获取消息。当您在应用程序调试期间需要将消息写至标准输出时，这些过程很有用。

此模块的模式为 SYSIBMADM。

DBMS_OUTPUT 模块包括以下内置例程。

表 17. DBMS_OUTPUT 模块中可用的内置例程

例程名称	描述
DISABLE 过程	禁用消息缓冲区。
ENABLE 过程	启用消息缓冲区
GET_LINE 过程	从消息缓冲区获取文本行。
GET_LINES 过程	从消息缓冲区获取一行或多行文本并将文本放到集合中
NEW_LINE 过程	将行结束字符序列放到消息缓冲区中。
PUT 过程	将未包括行结束字符序列的字符串放到消息缓冲区中。
PUT_LINE 过程	将包括一个行结束字符序列的一行放到消息缓冲区中。

此模块中的过程允许您使用消息缓冲区。使用命令行处理器 (CLP) 命令 SET SERVEROUTPUT ON 将输出重定向至标准输出。

自治过程内不支持 DISABLE 和 ENABLE 过程。

自主过程是这样的一个过程：被调用时，它在新事务中以独立于原始事务的方式执行。

示例

在 proc1 中，使用 PUT 和 PUT_LINE 过程将一行文本放到消息缓冲区中。 proc1 第一次运行时，会指定 SET SERVEROUTPUT ON，并且消息缓冲区中的该行将显示在 CLP 窗口中。proc1 第二次运行时，会指定 SET SERVEROUTPUT OFF，并且不会在 CLP 窗口中显示来自消息缓冲区的任何行。

```
CREATE PROCEDURE proc1( P1 VARCHAR(10) )
BEGIN
  CALL DBMS_OUTPUT.PUT( 'P1 = ' );
  CALL DBMS_OUTPUT.PUT_LINE( P1 );
END@

SET SERVEROUTPUT ON@

CALL proc1( '10' )@

SET SERVEROUTPUT OFF@

CALL proc1( '20' )@
```

此示例生成以下输出:

```
CALL proc1( '10' )

  Return Status = 0

P1 = 10

SET SERVEROUTPUT OFF
DB20000I  SET SERVEROUTPUT 命令成功完成。

CALL proc1( '20' )

  Return Status = 0
```

DISABLE 过程 - 禁用消息缓冲区

DISABLE 过程会禁用消息缓冲区。

此过程运行后，会废弃消息缓冲区中的所有消息。对 PUT、PUT_LINE 或 NEW_LINE 过程的调用会被忽略，并且不会向发送方返回任何错误。

语法

授权

对 DBMS_OUTPUT 模块的 EXECUTE 特权。

示例

以下示例禁用当前会话的消息缓冲区:

```
CALL DBMS_OUTPUT.DISABLE@
```

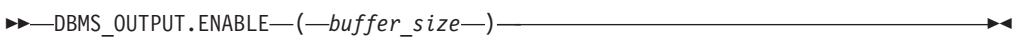
用法说明

要在禁用消息缓冲区后发送和接收消息，请使用 ENABLE 过程。

ENABLE 过程 - 启用消息缓冲区

ENABLE 过程会启用消息缓冲区。进行单个会话期间，应用程序可将消息放到消息缓冲区中以及从消息缓冲区获取消息。

语法



过程参数

buffer_size
类型为 INTEGER 的输入参数，用于指定消息缓冲区的最大长度（以字节计）。如果对 *buffer_size* 指定小于 2000 的值，那么缓冲区大小设置为 2000。如果该值为 NULL，那么缺省缓冲区大小为 20000。

授权

对 DBMS_OUTPUT 模块的 EXECUTE 特权。

示例

以下示例启用消息缓冲区:

```
CALL DBMS_OUTPUT.ENABLE( NULL )@
```

用法说明

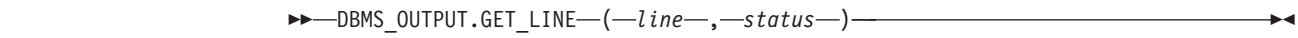
可调用 ENABLE 过程来提高现有消息缓冲区的大小。旧缓冲区中的所有消息将复制到增大后的缓冲区。

GET_LINE 过程 - 从消息缓冲区获取行

GET_LINE 过程从消息缓冲区获取文本行。该文本必须以行结束字符序列终止。

提示: 要将行结束字符序列添加至消息缓冲区，请使用 PUT_LINE 过程，或者在对 PUT 过程进行一系列调用后使用 NEW_LINE 过程。

语法



过程参数

line
类型为 VARCHAR(32672) 的输出参数，用于从消息缓冲区返回文本行。

status
类型为 INTEGER 的输出参数，用于指示是否从消息缓冲区返回了行:

- 0 指示已返回行
- 1 指示未返回行

授权

对 DBMS_OUTPUT 模块的 EXECUTE 特权。

示例

使用 GET_LINE 过程从消息缓冲区获取文本行。在此示例中，proc1 将一行文本放到消息缓冲区中。proc3 从消息缓冲区获取文本并将其插入到表 messages 中。然后会运行 proc2，但因为已禁用消息缓冲区，所以不会向消息缓冲区添加任何文本。SELECT 语句运行时，它仅返回 proc1 添加的文本。

```
CALL DBMS_OUTPUT.ENABLE( NULL )@

CREATE PROCEDURE proc1()
BEGIN
  CALL DBMS_OUTPUT.PUT_LINE( 'PROC1 put this line in the message buffer.' );
END@

CREATE PROCEDURE proc2()
BEGIN
```

```
  CALL DBMS_OUTPUT.PUT_LINE( 'PROC2 put this line in the message buffer.' );
END@

CREATE TABLE messages ( msg VARCHAR(100) )@

CREATE PROCEDURE proc3()
BEGIN
  DECLARE line VARCHAR(32672);
  DECLARE status INT;

  CALL DBMS_OUTPUT.GET_LINE( line, status );
  while status = 0 do
    INSERT INTO messages VALUES ( line );
    CALL DBMS_OUTPUT.GET_LINE( line, status );
  end while;
END@

CALL proc1@

CALL proc3@

CALL DBMS_OUTPUT.DISABLE@

CALL proc2@

CALL proc3@

SELECT * FROM messages@
```

此示例生成以下输出:

```
MSG
--------------------------------------------
PROC1 put this line in the message buffer.

  1 record(s) selected.
```

GET_LINES 过程 - 从消息缓冲区获取多行

GET_LINES 过程从消息缓冲区获取一行或多行文本并将该文本存储在集合中。该文本的每行必须以行结束字符序列终止。

提示: 要将行结束字符序列添加至消息缓冲区，请使用 PUT_LINE 过程，或者在对 PUT 过程进行一系列调用后使用 NEW_LINE 过程。

语法

过程参数

lines
类型为 DBMS_OUTPUT.CHARARR 的输出参数，用于从消息缓冲区返回文本行。类型 DBMS_OUTPUT.CHARARR 在内部定义为 VARCHAR(32672) ARRAY[2147483647] 数组。

numlines
类型为 INTEGER 的输入和输出参数。用作输入时，指定要从消息缓冲区检索的行数。用作输出时，指示从消息缓冲区检索到的实际行数。如果 *numlines* 的输出值小于输入值，那么消息缓冲区中没有其他行。

授权

对 DBMS_OUTPUT 模块的 EXECUTE 特权。

示例

使用 GET_LINES 过程从消息缓冲区获取文本行并将该文本存储在数组中。数组中的文本可插入到表中并可供查询。

```
CALL DBMS_OUTPUT.ENABLE( NULL )@

CREATE PROCEDURE proc1()
BEGIN
  CALL DBMS_OUTPUT.PUT_LINE( 'PROC1 put this line in the message buffer.' );
  CALL DBMS_OUTPUT.PUT_LINE( 'PROC1 put this line in the message buffer.' );
END@

CREATE PROCEDURE proc2()
BEGIN
  CALL DBMS_OUTPUT.PUT_LINE( 'PROC2 put this line in the message buffer.' );
END@

CREATE TABLE messages ( msg VARCHAR(100) )@

CREATE PROCEDURE proc3()
BEGIN
  DECLARE lines DBMS_OUTPUT.CHARARR;
  DECLARE numlines INT;
  DECLARE i INT;

  CALL DBMS_OUTPUT.GET_LINES( lines, numlines );
  SET i = 1;
  WHILE i <= numlines DO
    INSERT INTO messages VALUES ( lines[i] );
    SET i = i + 1;
  END WHILE;
END@

CALL proc1@

CALL proc3@

CALL DBMS_OUTPUT.DISABLE@

CALL proc2@

CALL proc3@

SELECT * FROM messages@
```

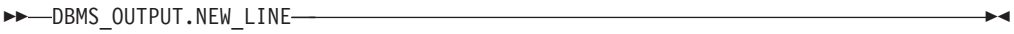
此示例生成以下输出:

```
MSG
----------------------------------------
PROC1 put this line in the message buffer.
PROC1 put this line in the message buffer

  2 record(s) selected.
```

NEW_LINE 过程 - 将行结束字符序列放到消息缓冲区中

NEW_LINE 过程 - 将行结束字符序列放到消息缓冲区中。

语法



授权

对 DBMS_OUTPUT 模块的 EXECUTE 特权。

示例

使用 NEW_LINE 过程将行结束字符序列写至消息缓冲区。在此示例中，因为指定了 SERVEROUTPUT ON，所以后跟行结束字符序列的文本显示为输出。但是，不会显示在消息缓冲区中却未后跟行结束字符的文本。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  CALL DBMS_OUTPUT.PUT( 'T' );
  CALL DBMS_OUTPUT.PUT( 'h' );
  CALL DBMS_OUTPUT.PUT( 'i' );
  CALL DBMS_OUTPUT.PUT( 's' );
  CALL DBMS_OUTPUT.NEW_LINE;
  CALL DBMS_OUTPUT.PUT( 'T' );
  CALL DBMS_OUTPUT.PUT( 'h' );
  CALL DBMS_OUTPUT.PUT( 'a' );
  CALL DBMS_OUTPUT.PUT( 't' );
END@

CALL proc1@

SET SERVEROUTPUT OFF@
```

此示例生成以下输出:

```
This
```

PUT 过程 - 将行的部分放在消息缓冲区中

PUT 过程将字符串放在消息缓冲区中。字符串结尾不会写入任何行结束字符序列。

语法

过程参数

item
类型为 VARCHAR(32672) 的输入参数，用于指定要写至消息缓冲区的文本。

授权

对 DBMS_OUTPUT 模块的 EXECUTE 特权。

示例

使用 PUT 过程将行的部分放在消息缓冲区中。在此示例中，NEW_LINE 过程将行结束字符序列添加至消息缓冲区。proc1 运行时，因为指定了 SET SERVEROUTPUT ON，所以会返回一行文本。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  CALL DBMS_OUTPUT.PUT( 'H' );
  CALL DBMS_OUTPUT.PUT( 'e' );
  CALL DBMS_OUTPUT.PUT( 'l' );
  CALL DBMS_OUTPUT.PUT( 'l' );
  CALL DBMS_OUTPUT.PUT( 'o' );
  CALL DBMS_OUTPUT.PUT( '.' );
  CALL DBMS_OUTPUT.NEW_LINE;
END@

CALL proc1@

SET SERVEROUTPUT OFF@
```

此示例生成以下输出:

```
Hello.
```

用法说明

使用 PUT 过程将文本添加至消息缓冲区之后，请使用 NEW_LINE 过程将行结束字符序列添加至消息缓冲区。否则，GET_LINE 和 GET_LINES 过程不会回返该文本，因为它并非完整行。

PUT_LINE 过程 - 将完整行放到消息缓冲区中

PUT_LINE 过程将包括行结束字符序列的单行放到消息缓冲区中。

语法

过程参数

item
类型为 VARCHAR(32672) 的输入参数，用于指定要写至消息缓冲区的文本。

授权

对 PUT_LINE 过程的 EXECUTE 特权。

示例

使用 PUT_LINE 过程将包括行结束字符序列的单行放到消息缓冲区中。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE PROC1()
BEGIN
  CALL DBMS_OUTPUT.PUT( 'a' );
```

```
  CALL DBMS_OUTPUT.NEW_LINE;
  CALL DBMS_OUTPUT.PUT_LINE( 'b' );
END@

CALL PROC1@

SET SERVEROUTPUT OFF@
```

此示例生成以下输出:

```
a
b
```

第 27 章 DBMS_PIPE 模块

DBMS_PIPE 模块提供一组例程，用于通过管道在连接至同一 DB2 例程中数据库的会话中或会话间发送消息。

此模块的模式为 SYSIBMADM。

DBMS_PIPE 模块包括以下内置例程。

表 18. DBMS_PIPE 模块中可用的内置例程

例程名称	描述
CREATE_PIPE 函数	显式创建专用或公用管道。
NEXT_ITEM_TYPE 函数	确定所接收消息中的下一项的数据类型。
PACK_MESSAGE 函数	将某项放在会话的本地消息缓冲区中。
PACK_MESSAGE_RAW 过程	将类型为 RAW 的项放在会话的本地消息缓冲区中。
PURGE 过程	除去指定管道中未接收到的消息。
RECEIVE_MESSAGE 函数	从指定的管道中获取消息。
REMOVE_PIPE 函数	删除显式创建的管道。
RESET_BUFFER 过程	重置本地消息缓冲区。
SEND_MESSAGE 过程	在指定的管道上发送消息。
UNIQUE_SESSION_NAME 函数	返回唯一会话名称。
UNPACK_MESSAGE 过程	从消息中检索下一数据项并将其指定给变量。

用法说明

管道是在过程调用期间隐式或显式创建的。*隐式管道*是在过程调用包含对不存在的管道名称的引用时创建的。例如，如果管道“mailbox”被传递至 SEND_MESSAGE 过程，并且该管道尚未存在，那么会创建新管道“mailbox”。*显式管道*是通过调用 CREATE_PIPE 函数并指定管道名称创建的。

管道可以是专用管道或公用管道。*专用管道*只能由创建该管道的用户访问。甚至管理员也不能访问另一用户创建的专用管道。*公用管道*可由对 DBMS_PIPE 模块具有访问权的任何用户访问。要对管道指定访问级别，请使用 CREATE_PIPE 函数并指定 *private* 参数的值：“false”指定该管道为公用管道，“true”指定该管道为专用管道。如果未指定任何值，那么缺省值是创建专用管道。所有隐式管道都是专用管道。

要通过管道发送消息，请调用 PACK_MESSAGE 函数以将个别数据项（行）放入到当前会话独有的本地消息缓冲区中。然后，调用 SEND_MESSAGE 过程以通过管道发送消息。

要接收消息，请调用 RECEIVE_MESSAGE 函数以从指定管道获取消息。该消息将写至接收会话的本地消息缓冲区。然后，调用 UNPACK_MESSAGE 过程以从本地消息缓冲区检索下一数据项并将其指定给指定的程序变量。如果管道包含多个消息，那么 RECEIVE_MESSAGE 函数会以 FIFO（先进先出）顺序获取消息。

每个会话保留单独的消息缓冲区，以保存 PACK_MESSAGE 函数创建的消息及 RECEIVE_MESSAGE 函数检索的消息。单独的消息缓冲区允许您在同一会话中构建和接收消息。但是，对 RECEIVE_MESSAGE 函数进行连续调用时，只有来自最后一个 RECEIVE_MESSAGE 调用的消息才会保存在本地消息缓冲区中。

示例

在连接 1 中，创建名为 pipe1 的管道。将消息放在会话的本地消息缓冲区中并通过 pipe1 发送该消息。

```
BEGIN
  DECLARE status INT;
  SET status = DBMS_PIPE.CREATE_PIPE( 'pipe1' );
  SET status = DBMS_PIPE.PACK_MESSAGE('message1');
  SET status = DBMS_PIPE.SEND_MESSAGE( 'pipe1' );
END@
```

在连接 2 中，接收消息，对其解包，然后将其显示至标准输出。

```
SET SERVEROUTPUT ON@

BEGIN
  DECLARE status   INT;
  DECLARE int1     INTEGER;
  DECLARE date1    DATE;
  DECLARE raw1     BLOB(100);
  DECLARE varchar1 VARCHAR(100);
  DECLARE itemType INTEGER;

  SET status = DBMS_PIPE.RECEIVE_MESSAGE( 'pipe1' );
  IF( status = 0 ) THEN
    SET itemType = DBMS_PIPE.NEXT_ITEM_TYPE();
    CASE itemType
      WHEN 6 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_INT( int1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'int1: ' || int1 );
      WHEN 9 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_CHAR( varchar1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'varchar1: ' || varchar1 );
      WHEN 12 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_DATE( date1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'date1:' || date1 );
      WHEN 23 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_RAW( raw1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'raw1: ' || VARCHAR(raw1) );
      ELSE
        CALL DBMS_OUTPUT.PUT_LINE( 'Unexpected value' );
    END CASE;
  END IF;
  SET status = DBMS_PIPE.REMOVE_PIPE( 'pipe1' );
END@
```

此示例生成以下输出:

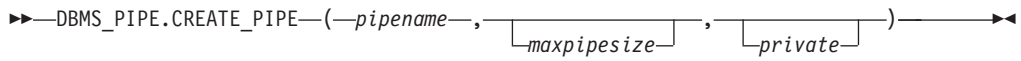
```
  varchar1: message1
```

CREATE_PIPE 函数 - 创建管道

CREATE_PIPE 函数显式创建具有指定名称的公用或专用管道。

有关显式公用和专用管道的更多信息，请参阅有关 DBMS_PIPE 模块的主题。

语法



返回值

如果成功创建该管道，那么此函数返回状态码 0。

函数参数

pipename
类型为 VARCHAR(128) 的输入参数，用于指定管道的名称。有关管道的更多信息，请参阅第 257 页的第 27 章，『DBMS_PIPE 模块』。

maxpipesize
类型为 INTEGER 的可选输入参数，用于指定管道的最大容量（以字节计）。缺省值为 8192 字节。

private
可选输入参数，用于指定管道的访问级别:

对于非分区数据库环境

值“0”或“FALSE”创建公用管道。

值“1”或“TRUE”创建专用管道。这是缺省值。

在分区数据库环境中

值“0”创建公用管道。

值“1”创建专用管道。这是缺省值。

授权

对 DBMS_PIPE 模块的 EXECUTE 特权。

示例

示例 1: 创建名为 messages 的专用管道:

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_status        INTEGER;
  SET v_status = DBMS_PIPE.CREATE_PIPE('messages');
  DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status: ' || v_status);
END@

CALL proc1@
```

此示例生成以下输出:

```
CREATE_PIPE status: 0
```

示例 2: 创建名为 mailbox 的公用管道:

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc2()
BEGIN
```

```
  DECLARE v_status INTEGER;
  SET v_status = DBMS_PIPE.CREATE_PIPE('mailbox',0);
  DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status: ' || v_status);
END@

CALL proc2@
```

此示例生成以下输出:

```
CREATE_PIPE status: 0
```

NEXT_ITEM_TYPE 函数 - 返回下一项的数据类型代码

NEXT_ITEM_TYPE 函数返回整数代码，该代码标识所接收消息中下一数据项的数据类型。

所接受消息存储在会话的本地消息缓冲区中。使用 UNPACK_MESSAGE 过程将每一项移出本地消息缓冲区，然后使用 NEXT_ITEM_TYPE 函数返回下一可用项的数据类型代码。如果消息中没有其他代码剩余，那么会返回代码 0。

语法

返回值

此函数返回下列表示数据类型的代码。

表 19. NEXT_ITEM_TYPE 数据类型代码

类型代码	数据类型
0	没有其他数据项
6	INTEGER
9	VARCHAR
12	DATE
23	BLOB

授权

对 DBMS_PIPE 模块的 EXECUTE 特权。

示例

在 proc1 中，将消息打包并发送。在 proc2 中，接收该消息，然后通过使用 NEXT_ITEM_TYPE 函数确定该消息的类型来对其解压缩。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE status INT;
  SET status = DBMS_PIPE.CREATE_PIPE( 'pipe1' );
  SET status = DBMS_PIPE.PACK_MESSAGE('message1');
  SET status = DBMS_PIPE.SEND_MESSAGE( 'pipe1' );
END@
```

```
CREATE PROCEDURE proc2()
BEGIN
  DECLARE status   INT;
  DECLARE num1     DECFLOAT;
  DECLARE date1    DATE;
  DECLARE raw1     BLOB(100);
  DECLARE varchar1 VARCHAR(100);
  DECLARE itemType INTEGER;

  SET status = DBMS_PIPE.RECEIVE_MESSAGE( 'pipe1' );
  IF( status = 0 ) THEN
    SET itemType = DBMS_PIPE.NEXT_ITEM_TYPE();
    CASE itemType
      WHEN 6 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_NUMBER( num1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'num1: ' || num1 );
      WHEN 9 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_CHAR( varchar1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'varchar1: ' || varchar1 );
      WHEN 12 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_DATE( date1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'date1:' || date1 );
      WHEN 23 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_RAW( raw1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'raw1: ' || VARCHAR(raw1) );
      ELSE
        CALL DBMS_OUTPUT.PUT_LINE( 'Unexpected value' );
    END CASE;
  END IF;
  SET status = DBMS_PIPE.REMOVE_PIPE( 'pipe1' );
END@

CALL proc1@

CALL proc2@
```

此示例生成以下输出:

```
varchar1: message1
```

PACK_MESSAGE 函数 - 将数据项放到本地消息缓冲区中

PACK_MESSAGE 函数将数据项放到会话的本地消息缓冲区中。

语法

过程参数

item

类型为 VARCHAR(4096)、DATE 或 DECFLOAT 的输入参数，其中包含表达式。此表达式返回的值将添加至会话的本地消息缓冲区。

提示: 要将类型为 RAW 的数据项放到本地消息缓冲区中，请使用 PACK_MESSAGE_RAW 过程。

授权

对 DBMS_PIPE 模块的 EXECUTE 特权。

示例

使用 PACK_MESSAGE 函数将有关 Sujata 的消息放在本地消息缓冲区中，然后使用 SEND_MESSAGE 过程在管道上发送该消息。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_status    INTEGER;
  DECLARE    status      INTEGER;
  SET status = DBMS_PIPE.PACK_MESSAGE('Hi, Sujata');
  SET status = DBMS_PIPE.PACK_MESSAGE('Can you attend a meeting at 3:00, today?');
  SET status = DBMS_PIPE.PACK_MESSAGE('If not, is tomorrow at 8:30 ok with you?');
  SET v_status = DBMS_PIPE.SEND_MESSAGE('pipe');
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END@

CALL proc1@
```

此示例生成以下输出:

```
  SEND_MESSAGE status: 0
```

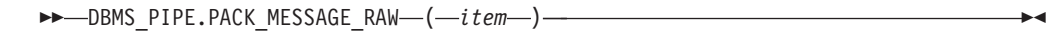
用法说明

在发出 SEND_MESSAGE 调用之前，必须至少调用一次 PACK_MESSAGE 函数或 PACK_MESSAGE_RAW 过程。

PACK_MESSAGE_RAW 过程 - 将类型为 RAW 的数据项放在本地消息缓冲区中

PACK_MESSAGE_RAW 过程将类型为 RAW 的数据项放在会话的本地消息缓冲区中。

语法



过程参数

item
类型为 BLOB(4096) 的输入参数，用于指定表达式。此表达式返回的值将添加至会话的本地消息缓冲区。

授权

对 DBMS_PIPE 模块的 EXECUTE 特权。

示例

使用 PACK_MESSAGE_RAW 过程将类型为 RAW 的数据项放到本地消息缓冲区中。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_raw           BLOB(100);
  DECLARE v_raw2   BLOB(100);
  DECLARE v_status        INTEGER;
```

```
  SET v_raw = BLOB('21222324');
  SET v_raw2 = BLOB('30000392');
  CALL DBMS_PIPE.PACK_MESSAGE_RAW(v_raw);
  CALL DBMS_PIPE.PACK_MESSAGE_RAW(v_raw2);
  SET v_status = DBMS_PIPE.SEND_MESSAGE('datatypes');
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END@

CALL proc1@
```

此示例生成以下输出:

```
SEND_MESSAGE status: 0
```

用法说明

在发出 SEND_MESSAGE 调用之前，必须至少调用一次 PACK_MESSAGE 函数或 PACK_MESSAGE_RAW 过程。

PURGE 过程 - 从管道中除去未接收消息

PURGE 过程除去指定隐式管道中的未接收消息。

提示: 使用 REMOVE_PIPE 函数来删除显式管道。

语法

过程参数

pipename
类型为 VARCHAR(128) 的输入参数，用于指定隐式管道的名称。

授权

对 DBMS_PIPE 模块的 EXECUTE 特权。

示例

在 proc1 中，在管道上发送以下两条消息: Message #1 和 Message #2。在 proc2 中，接收第一条消息，对其解包，然后清除管道。proc3 运行时，对 RECEIVE_MESSAGE 函数的调用超时并返回状态码 1，因为没有可用消息。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE   v_status        INTEGER;
  DECLARE   status          INTEGER;
  SET status = DBMS_PIPE.PACK_MESSAGE('Message #1');
  SET v_status = DBMS_PIPE.SEND_MESSAGE('pipe');
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
  SET status = DBMS_PIPE.PACK_MESSAGE('Message #2');
  SET v_status = DBMS_PIPE.SEND_MESSAGE('pipe');
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END@

CREATE PROCEDURE proc2()
```

```
BEGIN
  DECLARE   v_item          VARCHAR(80);
  DECLARE   v_status        INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
  CALL DBMS_PIPE.UNPACK_MESSAGE_CHAR(v_item);
  CALL DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
  CALL DBMS_PIPE.PURGE('pipe');
END@

CREATE PROCEDURE proc3()
BEGIN
  DECLARE    v_item          VARCHAR(80);
  DECLARE    v_status        INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
END@

CALL proc1@

CALL proc2@

CALL proc3@
```

此示例生成以下输出。

在 proc1 中:

```
SEND_MESSAGE status: 0
SEND_MESSAGE status: 0
```

在 proc2 中:

```
RECEIVE_MESSAGE status: 0
Item: Hi, Sujata
```

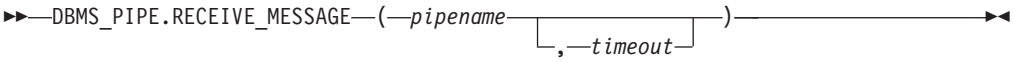
在 proc3 中:

```
RECEIVE_MESSAGE status: 1
```

RECEIVE_MESSAGE 函数 - 从指定管道获取消息

RECEIVE_MESSAGE 函数从指定管道获取消息。

语法



返回值

RECEIVE_MESSAGE 函数返回类型为 INTEGER 的下列状态码之一。

表 20. RECEIVE_MESSAGE 状态码

状态码	描述
0	成功
1	超时

函数参数

pipename
类型为 VARCHAR(128) 的输入参数，用于指定管道的名称。如果指定的管道不存在，那么会隐式创建该管道。有关管道的更多信息，请参阅第 257 页的第 27 章，『DBMS_PIPE 模块』。

timeout
类型为 INTEGER 的可选输入参数，用于指定等待时间（以秒计）。缺省值为 86400000 秒（1000 天）。

授权

对 DBMS_PIPE 模块的 EXECUTE 特权。

示例

在 proc1 中，发送一条消息。在 proc2 中，接收该消息并对其解包。如果在 1 秒内未接收到该消息，那么会超时。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE status INTEGER;
  SET status = DBMS_PIPE.CREATE_PIPE( 'pipe1' );
  SET status = DBMS_PIPE.PACK_MESSAGE('message1');
  SET status = DBMS_PIPE.SEND_MESSAGE( 'pipe1' );
END@

CREATE PROCEDURE proc2()
BEGIN
  DECLARE    v_item          VARCHAR(80);
  DECLARE    v_status        INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe1',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
  CALL DBMS_PIPE.UNPACK_MESSAGE(v_item);
  CALL DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
END@

CALL proc1@
CALL proc2@
```

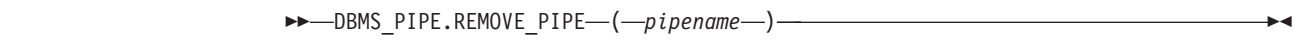
此示例生成以下输出:

```
RECEIVE_MESSAGE status: 0
Item: message1
```

REMOVE_PIPE 函数 - 删除管道

REMOVE_PIPE 函数删除显式创建的管道。使用此函数删除 CREATE_PIPE 函数创建的任意公用或专用管道。

语法



返回值

此函数返回类型为 INTEGER 的下列状态码之一。

表 21. REMOVE_PIPE 状态码

状态码	描述
0	已成功除去管道或管道不存在
NULL	抛出了异常

函数参数

pipename
类型为 VARCHAR(128) 的输入参数，用于指定管道的名称。

授权

对 DBMS_PIPE 模块的 EXECUTE 特权。

示例

在 proc1 中，在管道上发送以下两条消息：Message #1 和 Message #2。在 proc2 中，接收第一条消息，对其解包，然后删除管道。proc3 运行时，对 RECEIVE_MESSAGE 函数的调用超时并返回状态码 1，因为该管道不再存在。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_status        INTEGER;
  DECLARE    status          INTEGER;
  SET  v_status = DBMS_PIPE.CREATE_PIPE('pipe1');
  CALL DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status : ' || v_status);

  SET status = DBMS_PIPE.PACK_MESSAGE('Message #1');
  SET v_status = DBMS_PIPE.SEND_MESSAGE('pipe1');
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);

  SET status = DBMS_PIPE.PACK_MESSAGE('Message #2');
  SET v_status = DBMS_PIPE.SEND_MESSAGE('pipe1');
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END@

CREATE PROCEDURE proc2()
BEGIN
  DECLARE    v_item          VARCHAR(80);
  DECLARE    v_status        INTEGER;
  DECLARE    status          INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe1',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
  CALL DBMS_PIPE.UNPACK_MESSAGE(v_item);
  CALL DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
  SET status = DBMS_PIPE.REMOVE_PIPE('pipe1');
END@

CREATE PROCEDURE proc3()
BEGIN
  DECLARE    v_item          VARCHAR(80);
  DECLARE    v_status        INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe1',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
END@
```

```
CALL proc1@

CALL proc2@

CALL proc3@
```

此示例生成以下输出。

在 proc1 中:

```
CREATE_PIPE status : 0
SEND_MESSAGE status: 0
SEND_MESSAGE status: 0
```

在 proc2 中:

```
RECEIVE_MESSAGE status: 0
Item: Message #1
```

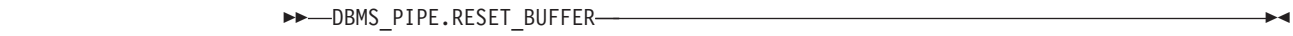
在 proc3 中:

```
RECEIVE_MESSAGE status: 1
```

RESET_BUFFER 过程 - 重置本地消息缓冲区

RESET_BUFFER 过程将会话的本地消息缓冲区的指针重置回至缓冲区的开头。重置缓冲区会导致后续 PACK_MESSAGE 调用覆盖消息缓冲区中，在 RESET_BUFFER 调用之前已存在的任何数据项。

语法



授权

对 DBMS_PIPE 模块的 EXECUTE 特权。

示例

在 proc1 中，使用 PACK_MESSAGE 函数将职员 Sujata 的消息放在本地消息缓冲区中。调用 RESET_BUFFER 过程以将该消息替换为 Bing 的消息，然后在管道上发送消息。在 proc2 中，接收 Bing 的消息并对该消息解包。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_status        INTEGER;
  DECLARE    status          INTEGER;
  SET status = DBMS_PIPE.PACK_MESSAGE('Hi, Sujata');
  SET status = DBMS_PIPE.PACK_MESSAGE('Can you attend a meeting at 3:00, today?');
  SET status = DBMS_PIPE.PACK_MESSAGE('If not, is tomorrow at 8:30 ok with you?');
  CALL DBMS_PIPE.RESET_BUFFER;
  SET status = DBMS_PIPE.PACK_MESSAGE('Hi, Bing');
  SET status = DBMS_PIPE.PACK_MESSAGE('Can you attend a meeting at 9:30, tomorrow?');
  SET v_status = DBMS_PIPE.SEND_MESSAGE('pipe');
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END@

CREATE PROCEDURE proc2()
```

```
BEGIN
  DECLARE    v_item          VARCHAR(80);
  DECLARE    v_status        INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
  CALL DBMS_PIPE.UNPACK_MESSAGE_CHAR(v_item);
  CALL DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
  CALL DBMS_PIPE.UNPACK_MESSAGE(v_item);
  CALL DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
END@

CALL proc1@

CALL proc2@
```

此示例生成以下输出:

在 proc1 中:

```
SEND_MESSAGE status: 0
```

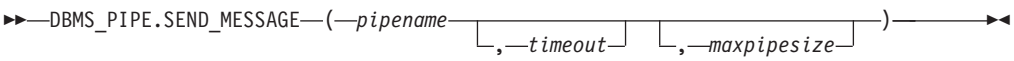
在 proc2 中:

```
RECEIVE_MESSAGE status: 0
Item: Hi, Bing
Item: Can you attend a meeting at 9:30, tomorrow?
```

SEND_MESSAGE 过程 - 将消息发送至指定管道

SEND_MESSAGE 过程将会话的本地消息缓冲区中的消息发送至指定管道。

语法



返回值

此过程返回类型为 INTEGER 的下列状态码之一。

表 22. SEND_MESSAGE 状态码

状态码	描述
0	成功
1	超时

过程参数

pipename

类型为 VARCHAR(128) 的输入参数，用于指定管道的名称。如果指定的管道不存在，那么会隐式创建该管道。有关管道的更多信息，请参阅第 257 页的第 27 章，『DBMS_PIPE 模块』。

timeout

类型为 INTEGER 的可选输入参数，用于指定等待时间（以秒计）。缺省值为 86400000 秒（1000 天）。

maxpipesize
类型为 INTEGER 的可选输入参数，用于指定管道的最大容量（以字节计）。缺省值为 8192 字节。

授权

对 DBMS_PIPE 模块的 EXECUTE 特权。

示例

在 proc1 中，发送一条消息。在 proc2 中，接收该消息并对其解包。如果在 1 秒内未接收到该消息，那么会超时。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE status INTEGER;
  SET status = DBMS_PIPE.CREATE_PIPE( 'pipe1' );
  SET status = DBMS_PIPE.PACK_MESSAGE('message1');
  SET status = DBMS_PIPE.SEND_MESSAGE( 'pipe1' );
END@

CREATE PROCEDURE proc2()
BEGIN
  DECLARE    v_item          VARCHAR(80);
  DECLARE    v_status        INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe1',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
  CALL DBMS_PIPE.UNPACK_MESSAGE(v_item);
  CALL DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
END@

CALL proc1@
CALL proc2@
```

此示例生成以下输出:

```
RECEIVE_MESSAGE status: 0
Item: message1
```

UNIQUE_SESSION_NAME 函数 - 返回唯一会话名称

UNIQUE_SESSION_NAME 函数返回当前会话的唯一名称。

可使用此函数创建与当前会话同名的管道。要创建此管道，请将 UNIQUE_SESSION_NAME 函数返回的值作为管道名称传递至 SEND_MESSAGE 过程。会创建与当前会话同名的隐式管道。

语法

返回值

此函数返回类型为 VARCHAR(128) 的值，该值表示当前会话的唯一名称。

授权

对 DBMS_PIPE 模块的 EXECUTE 特权。

示例

创建与当前会话同名的管道。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    status          INTEGER;
  DECLARE    v_session       VARCHAR(30);
  SET v_session = DBMS_PIPE.UNIQUE_SESSION_NAME;
  SET status = DBMS_PIPE.PACK_MESSAGE('message1');
  SET status = DBMS_PIPE.SEND_MESSAGE(v_session);
  CALL DBMS_OUTPUT.PUT_LINE('Sent message on pipe ' || v_session);
END@

CALL proc1@
```

此示例生成以下输出:

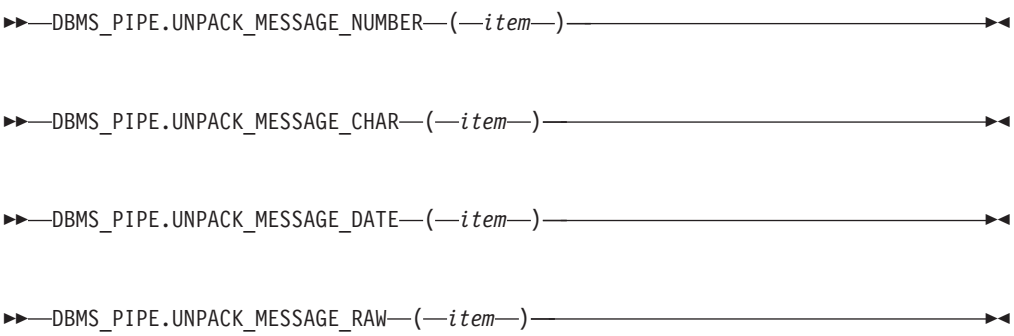
```
Sent message on pipe *LOCAL.myschema.080522010048
```

UNPACK_MESSAGE 过程 - 从本地消息缓冲区获取数据项

UNPACK_MESSAGE 过程从消息中检索下一个数据项并将其指定给变量。

调用其中一个 UNPACK_MESSAGE 过程之前，使用 RECEIVE_MESSAGE 过程将该消息放在本地消息缓冲区中。

语法



过程参数

item

下列其中一种类型的输出参数，用于指定从本地消息缓冲区接收数据项的变量。

例程	数据类型
UNPACK_MESSAGE_NUMBER	DECFLOAT
UNPACK_MESSAGE_CHAR	VARCHAR(4096)
UNPACK_MESSAGE_DATE	DATE
UNPACK_MESSAGE_RAW	BLOB(4096)

授权

对 DBMS_PIPE 模块的 EXECUTE 特权。

示例

在 proc1 中，将消息打包并发送。在 proc2 中，接收消息，根据该项的类型使用适当过程对其解包，然后将该消息显示至标准输出。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE status INT;
  SET status = DBMS_PIPE.CREATE_PIPE( 'pipe1' );
  SET status = DBMS_PIPE.PACK_MESSAGE('message1');
  SET status = DBMS_PIPE.SEND_MESSAGE( 'pipe1' );
END@

CREATE PROCEDURE proc2()
BEGIN
  DECLARE status   INT;
  DECLARE num1     DECFLOAT;
  DECLARE date1    DATE;
  DECLARE raw1     BLOB(100);
  DECLARE varchar1 VARCHAR(100);
  DECLARE itemType INTEGER;

  SET status = DBMS_PIPE.RECEIVE_MESSAGE( 'pipe1' );
  IF( status = 0 ) THEN
    SET itemType = DBMS_PIPE.NEXT_ITEM_TYPE();
    CASE itemType
      WHEN 6 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_NUMBER( num1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'num1: ' || num1 );
      WHEN 9 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_CHAR( varchar1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'varchar1: ' || varchar1 );
      WHEN 12 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_DATE( date1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'date1:' || date1 );
      WHEN 23 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_RAW( raw1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'raw1: ' || VARCHAR(raw1) );
      ELSE
        CALL DBMS_OUTPUT.PUT_LINE( 'Unexpected value' );
    END CASE;
  END IF;
  SET status = DBMS_PIPE.REMOVE_PIPE( 'pipe1' );
END@

CALL proc1@

CALL proc2@
```

此示例生成以下输出:

```
varchar1: message1
```

第 28 章 DBMS_SQL 模块

DBMS_SQL 模块提供一组用于执行动态 SQL 的过程，并因此支持各种数据操作语言（DML）或数据定义语言（DDL）语句。

此模块的模式为 SYSIBMADM。

DBMS_SQL 模块包括以下内置例程。

表 23. DBMS_SQL 模块中可用的内置例程

过程名称	描述
BIND_VARIABLE_BLOB 过程	提供 IN 或 INOUT 参数的输入 BLOB 值，并针对 INOUT 或 OUT 参数将输出值的数据类型定义为 BLOB。
BIND_VARIABLE_CHAR 过程	提供 IN 或 INOUT 参数的输入 CHAR 值，并针对 INOUT 或 OUT 参数将输出值的数据类型定义为 CHAR。
BIND_VARIABLE_CLOB 过程	提供 IN 或 INOUT 参数的输入 CLOB 值，并针对 INOUT 或 OUT 参数将输出值的数据类型定义为 CLOB。
BIND_VARIABLE_DATE 过程	提供 IN 或 INOUT 参数的输入 DATE 值，并针对 INOUT 或 OUT 参数将输出值的数据类型定义为 DATE。
BIND_VARIABLE_DOUBLE 过程	提供 IN 或 INOUT 参数的输入 DOUBLE 值，并针对 INOUT 或 OUT 参数将输出值的数据类型定义为 DOUBLE。
BIND_VARIABLE_INT 过程	提供 IN 或 INOUT 参数的输入 INTEGER 值，并针对 INOUT 或 OUT 参数将输出值的数据类型定义为 INTEGER。
BIND_VARIABLE_NUMBER 过程	提供 IN 或 INOUT 参数的输入 DECFLOAT 值，并针对 INOUT 或 OUT 参数将输出值的数据类型定义为 DECFLOAT。
BIND_VARIABLE_RAW 过程	提供 IN 或 INOUT 参数的输入 BLOB(32767) 值，并针对 INOUT 或 OUT 参数将输出值的数据类型定义为 BLOB(32767)。
BIND_VARIABLE_TIMESTAMP 过程	提供 IN 或 INOUT 参数的输入 TIMESTAMP 值，并针对 INOUT 或 OUT 参数将输出值的数据类型定义为 TIMESTAMP。
BIND_VARIABLE_VARCHAR 过程	提供 IN 或 INOUT 参数的输入 VARCHAR 值，并针对 INOUT 或 OUT 参数将输出值的数据类型定义为 VARCHAR。
CLOSE_CURSOR 过程	关闭游标。
COLUMN_VALUE_BLOB 过程	检索类型为 BLOB 的列的值。
COLUMN_VALUE_CHAR 过程	检索类型为 CHAR 的列的值。
COLUMN_VALUE_CLOB 过程	检索类型为 CLOB 的列的值。

表 23. DBMS_SQL 模块中可用的内置例程 (续)

过程名称	描述
COLUMN_VALUE_DATE 过程	检索类型为 DATE 的列的值。
COLUMN_VALUE_DOUBLE 过程	检索类型为 DOUBLE 的列的值。
COLUMN_VALUE_INT 过程	检索类型为 INTEGER 的列的值。
COLUMN_VALUE_LONG 过程	检索类型为 CLOB(32767) 的列的值。
COLUMN_VALUE_NUMBER 过程	检索类型为 DECFLOAT 的列的值。
COLUMN_VALUE_RAW 过程	检索类型为 BLOB(32767) 的列的值。
COLUMN_VALUE_TIMESTAMP 过程	检索类型为 TIMESTAMP 的列的值。
COLUMN_VALUE_VARCHAR 过程	检索类型为 VARCHAR 的列的值。
DEFINE_COLUMN_BLOB 过程	将列的数据类型定义为 BLOB。
DEFINE_COLUMN_CHAR 过程	将列的数据类型定义为 CHAR。
DEFINE_COLUMN_CLOB 过程	将列的数据类型定义为 CLOB。
DEFINE_COLUMN_DATE 过程	将列的数据类型定义为 DATE。
DEFINE_COLUMN_DOUBLE 过程	将列的数据类型定义为 DOUBLE。
DEFINE_COLUMN_INT 过程	将列的数据类型定义为 INTEGER。
DEFINE_COLUMN_LONG 过程	将列的数据类型定义为 CLOB(32767)。
DEFINE_COLUMN_NUMBER 过程	将列的数据类型定义为 DECFLOAT。
DEFINE_COLUMN_RAW 过程	将列的数据类型定义为 BLOB(32767)。
DEFINE_COLUMN_TIMESTAMP 过程	将列的数据类型定义为 TIMESTAMP。
DEFINE_COLUMN_VARCHAR 过程	将列的数据类型定义为 VARCHAR。
DESCRIBE_COLUMNS 过程	返回游标检索到的列的描述。
DESCRIBE_COLUMNS2 过程	与 DESCRIBE_COLUMNS 完全相同，但允许列名长度超过 32 个字符。
EXECUTE 过程	执行游标。
EXECUTE_AND_FETCH 过程	执行游标并访存一行。
FETCH_ROWS 过程	通过游标访存行。
IS_OPEN 过程	检查游标是否已打开。
LAST_ROW_COUNT 过程	返回访存的总行数。
OPEN_CURSOR 过程	打开游标。
PARSE 过程	解析 DDL 语句。
VARIABLE_VALUE_BLOB 过程	将 INOUT 或 OUT 参数的值作为 BLOB 进行检索。
VARIABLE_VALUE_CHAR 过程	将 INOUT 或 OUT 参数的值作为 CHAR 进行检索。
VARIABLE_VALUE_CLOB 过程	将 INOUT 或 OUT 参数的值作为 CLOB 进行检索。
VARIABLE_VALUE_DATE 过程	将 INOUT 或 OUT 参数的值作为 DATE 进行检索。
VARIABLE_VALUE_DOUBLE 过程	将 INOUT 或 OUT 参数的值作为 DOUBLE 进行检索。
VARIABLE_VALUE_INT 过程	将 INOUT 或 OUT 参数的值作为 INTEGER 进行检索。

表 23. DBMS_SQL 模块中可用的内置例程 (续)

过程名称	描述
VARIABLE_VALUE_NUMBER 过程	将 INOUT 或 OUT 参数的值作为 DECFLOAT 进行检索。
VARIABLE_VALUE_RAW 过程	将 INOUT 或 OUT 参数的值作为 BLOB (32767) 进行检索。
VARIABLE_VALUE_TIMESTAMP 过程	将 INOUT 或 OUT 参数的值作为 TIMESTAMP 进行检索。
VARIABLE_VALUE_VARCHAR 过程	将 INOUT 或 OUT 参数的值作为 VARCHAR 进行检索。

下表列示 DBMS_SQL 模块中可用的内置类型和常量。

表 24. DBMS_SQL 内置类型和常量

名称	类型或常量	描述
DESC_REC	类型	列信息的记录。
DESC_REC2	类型	列信息的记录。
DESC_TAB	类型	类型为 DESC_REC 的一组记录。
DESC_TAB2	类型	类型为 DESC_REC2 的一组记录。
NATIVE	常量	PARSE 过程的 language_flag 参数唯一支持的值。

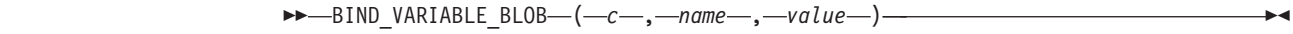
用法说明

当您想要构造并运行动态 SQL 语句时，DBMS_SQL 模块中的例程非常有用。例如，您可能想要执行“ALTER TABLE”或“DROP TABLE”之类的 DDL 或 DML 语句，实时构造并执行 SQL 语句，或调用使用 SQL 语句中的动态 SQL 的函数。

BIND_VARIABLE_BLOB 过程 - 将 BLOB 值绑定至变量

BIND_VARIABLE_BLOB 过程能够使 BLOB 值与 SQL 命令中的 IN、INOUT 或 OUT 参数相关联。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定带有绑定变量的 SQL 命令的游标标识。

name
类型为 VARCHAR(128) 的输入参数，用于指定 SQL 命令中的绑定变量的名称。

value
类型为 BLOB(2G) 的输入参数，用于指定要指定的值。

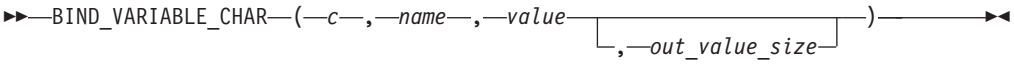
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

BIND_VARIABLE_CHAR 过程 - 将 CHAR 值绑定至变量

BIND_VARIABLE_CHAR 过程能够使 CHAR 值与 SQL 命令中的 IN、INOUT 或 OUT 参数相关联。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定带有绑定变量的 SQL 命令的游标标识。

name
类型为 VARCHAR(128) 的输入参数，用于指定 SQL 命令中的绑定变量的名称。

value
类型为 CHAR(254) 的输入参数，用于指定要指定的值。

out_value_size
类型为 INTEGER 的可选输入参数，用于指定 IN 或 INOUT 参数的长度限制以及 INOUT 或 OUT 参数的输出值最大长度。如果未指定此参数，那么采用 *value* 的长度。

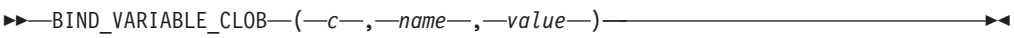
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

BIND_VARIABLE_CLOB 过程 - 将 CLOB 值绑定至变量

BIND_VARIABLE_CLOB 过程能够使 CLOB 值与 SQL 命令中的 IN、INOUT 或 OUT 参数相关联。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定带有绑定变量的 SQL 命令的游标标识。

name
类型为 VARCHAR(128) 的输入参数，用于指定 SQL 命令中的绑定变量的名称。

value
类型为 CLOB(2G) 的输入参数，用于指定要指定的值。

授权

对 DBMS_SQL 模块的 EXECUTE 特权。

BIND_VARIABLE_DATE 过程 - 将 DATE 值绑定至变量

BIND_VARIABLE_DATE 过程能够使 DATE 值与 SQL 命令中的 IN、INOUT 或 OUT 参数相关联。

语法

```
►►—BIND_VARIABLE_DATE—(—c—,—name—,—value—)———————————————————————►◄
```

参数

c 类型为 INTEGER 的输入参数，用于指定带有绑定变量的 SQL 命令的游标标识。

name
类型为 VARCHAR(128) 的输入参数，用于指定 SQL 命令中的绑定变量的名称。

value
类型为 DATE 的输入参数，用于指定要指定的值。

授权

对 DBMS_SQL 模块的 EXECUTE 特权。

BIND_VARIABLE_DOUBLE 过程 - 将 DOUBLE 值绑定至变量

BIND_VARIABLE_DOUBLE 过程能够使 DOUBLE 值与 SQL 命令中的 IN、INOUT 或 OUT 参数相关联。

语法

```
►►—BIND_VARIABLE_DOUBLE—(—c—,—name—,—value—)———————————————————————►◄
```

参数

c 类型为 INTEGER 的输入参数，用于指定带有绑定变量的 SQL 命令的游标标识。

name
类型为 VARCHAR(128) 的输入参数，用于指定 SQL 命令中的绑定变量的名称。

value
类型为 DOUBLE 的输入参数，用于指定要指定的值。

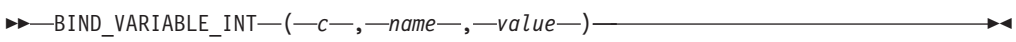
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

BIND_VARIABLE_INT 过程 - 将 INTEGER 值绑定至变量

BIND_VARIABLE_INT 过程能够使 INTEGER 值与 SQL 命令中的 IN 或 INOUT 绑定变量相关联。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定带有绑定变量的 SQL 命令的游标标识。

name
类型为 VARCHAR(128) 的输入参数，用于指定 SQL 命令中的绑定变量的名称。

value
类型为 INTEGER 的输入参数，用于指定要指定的值。

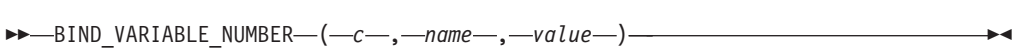
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

BIND_VARIABLE_NUMBER 过程 - 将 NUMBER 值绑定至变量

BIND_VARIABLE_NUMBER 过程能够使 NUMBER 值与 SQL 命令中的 IN、INOUT 或 OUT 参数相关联。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定带有绑定变量的 SQL 命令的游标标识。

name
类型为 VARCHAR(128) 的输入参数，用于指定 SQL 命令中的绑定变量的名称。

value
类型为 DECFLOAT 的输入参数，用于指定要指定的值。

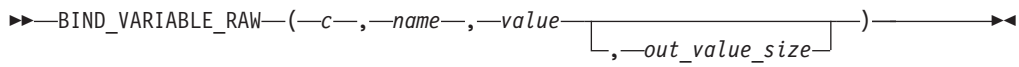
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

BIND_VARIABLE_RAW 过程 - 将 RAW 值绑定至变量

BIND_VARIABLE_RAW 过程能够使 RAW 值与 SQL 命令中的 IN、INOUT 或 OUT 参数相关联。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定带有绑定变量的 SQL 命令的游标标识。

name
类型为 VARCHAR(128) 的输入参数，用于指定 SQL 命令中的绑定变量的名称。

value
类型为 BLOB(32767) 的输入参数，用于指定要指定的值。

out_value_size
类型为 INTEGER 的可选输入参数，用于指定 IN 或 INOUT 参数的长度限制以及 INOUT 或 OUT 参数的输出值最大长度。如果未指定此参数，那么采用 *value* 的长度。

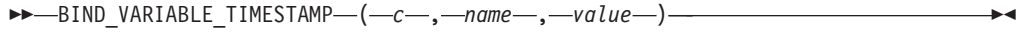
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

BIND_VARIABLE_TIMESTAMP 过程 - 将 TIMESTAMP 值绑定至变量

BIND_VARIABLE_TIMESTAMP 过程能够使 TIMESTAMP 值与 SQL 命令中的 IN、INOUT 或 OUT 参数相关联。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定带有绑定变量的 SQL 命令的游标标识。

name
类型为 VARCHAR(128) 的输入参数，用于指定 SQL 命令中的绑定变量的名称。

value
类型为 TIMESTAMP 的输入参数，用于指定要指定的值。

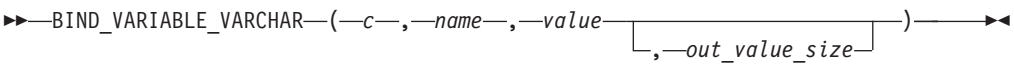
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

BIND_VARIABLE_VARCHAR 过程 - 将 VARCHAR 值绑定至变量

BIND_VARIABLE_VARCHAR 过程能够使 VARCHAR 值与 SQL 命令中的 IN、INOUT 或 OUT 参数相关联。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定带有绑定变量的 SQL 命令的游标标识。

name
类型为 VARCHAR(128) 的输入参数，用于指定 SQL 命令中的绑定变量的名称。

value
类型为 VARCHAR(32672) 的输入参数，用于指定要指定的值。

out_value_size
类型为 INTEGER 的输入参数，用于指定 IN 或 INOUT 参数的长度限制以及 INOUT 或 OUT 参数的输出值最大长度。如果未指定此参数，那么采用 *value* 的长度。

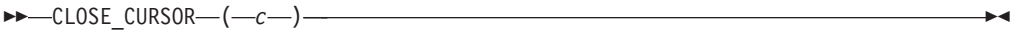
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

CLOSE_CURSOR 过程 - 关闭游标

CLOSE_CURSOR 过程关闭已打开游标。分配给游标的资源将被释放，并且不能再使用。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定要关闭的游标的标识。

授权

对 DBMS_SQL 模块的 EXECUTE 特权。

示例

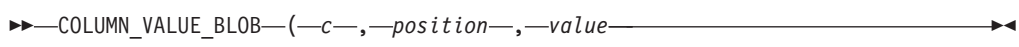
示例 1: 此示例演示正在关闭先前打开的游标。

```
DECLARE
    curid           INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
             .
             .
             .
    DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

COLUMN_VALUE_BLOB 过程 - 将 BLOB 列值返回到变量中

COLUMN_VALUE_BLOB 过程定义将从游标接收 BLOB 值的变量。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定将数据返回到所定义变量的游标的标识。

position
类型为 INTEGER 的输入参数，用于指定所返回数据在游标中的位置。游标中的第一个值为位置 1。

value
类型为 BLOB(2G) 的输出参数，用于指定接收先前访存调用中的游标所返回数据的变量。

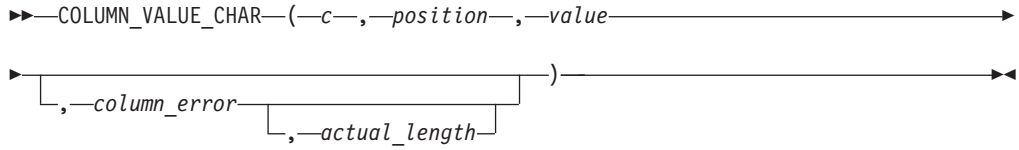
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

COLUMN_VALUE_CHAR 过程 - 将 CHAR 列值返回到变量中

COLUMN_VALUE_CHAR 过程定义将从游标接收 CHAR 值的变量。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定将数据返回到所定义变量的游标的标识。

position
类型为 INTEGER 的输入参数，用于指定所返回数据在游标中的位置。游标中的第一个值为位置 1。

value
类型为 CHAR 的输出参数，用于指定接收先前访存调用中的游标所返回数据的变量。

column_error
类型为 INTEGER 的可选输出参数，用于返回与列相关联的 SQLCODE（如果存在）。

actual_length
类型为 INTEGER 的可选输出参数，用于返回任何截断之前的数据实际长度。

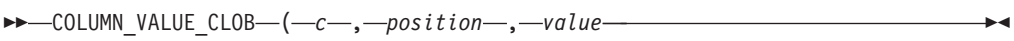
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

COLUMN_VALUE_CLOB 过程 - 将 CLOB 列值返回到变量中

COLUMN_VALUE_CLOB 过程定义将从游标接收 CLOB 值的变量。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定将数据返回到所定义变量的游标的标识。

position
类型为 INTEGER 的输入参数，用于指定所返回数据在游标中的位置。游标中的第一个值为位置 1。

value
类型为 CLOB(2G) 的输出参数，用于指定接收先前访存调用中的游标所返回数据的变量。

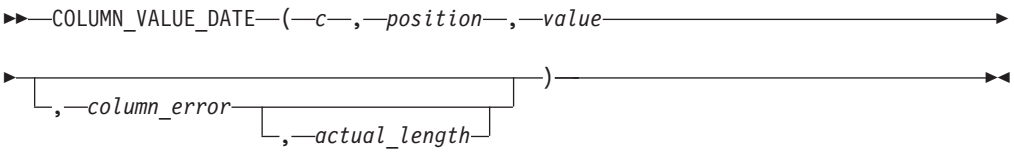
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

COLUMN_VALUE_DATE 过程 - 将 DATE 列值返回到变量中

COLUMN_VALUE_DATE 过程定义将从游标接收 DATE 值的变量。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定将数据返回到所定义变量的游标的标识。

position
类型为 INTEGER 的输入参数，用于指定所返回数据在游标中的位置。游标中的第一个值为位置 1。

value
类型为 DATE 的输出参数，用于指定接收先前访存调用中的游标所返回数据的变量。

column_error
类型为 INTEGER 的输出参数，用于返回与列相关联的 SQLCODE（如果存在）。

actual_length
类型为 INTEGER 的输出参数，用于返回任何截断之前的数据实际长度。

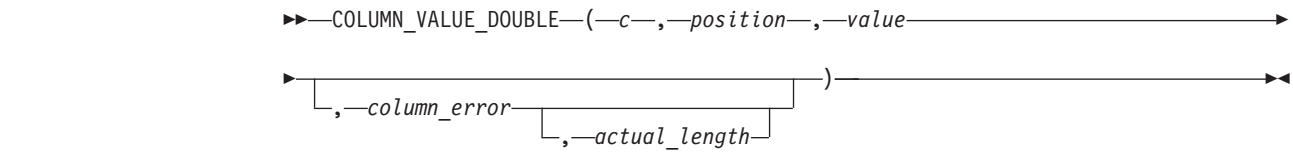
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

COLUMN_VALUE_DOUBLE 过程 - 将 DOUBLE 列值返回到变量中

COLUMN_VALUE_DOUBLE 过程定义将从游标接收 DOUBLE 值的变量。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定将数据返回到所定义变量的游标的标识。

position
类型为 INTEGER 的输入参数，用于指定所返回数据在游标中的位置。游标中的第一个值为位置 1。

value
类型为 DOUBLE 的输出参数，用于指定接收先前访存调用中的游标所返回数据的变量。

column_error
类型为 INTEGER 的输出参数，用于返回与列相关联的 SQLCODE（如果存在）。

actual_length
类型为 INTEGER 的输出参数，用于返回任何截断之前的数据实际长度。

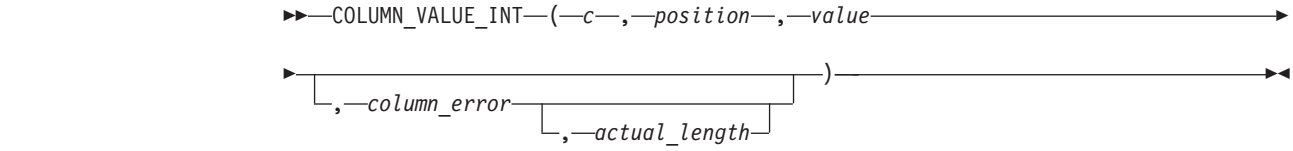
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

COLUMN_VALUE_INT 过程 - 将 INTEGER 列值返回到变量中

COLUMN_VALUE_INT 过程定义将从游标接收 INTEGER 值的变量。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定将数据返回到所定义变量的游标的标识。

position
类型为 INTEGER 的输入参数，用于指定所返回数据在游标中的位置。游标中的第一个值为位置 1。

value
类型为 INTEGER 的输出参数，用于指定接收先前访存调用中的游标所返回数据的变量。

column_error
类型为 INTEGER 的输出参数，用于返回与列相关联的 SQLCODE（如果存在）。

actual_length
类型为 INTEGER 的输出参数，用于返回任何截断之前的数据实际长度。

授权

对 DBMS_SQL 模块的 EXECUTE 特权。

COLUMN_VALUE_LONG 过程 - 将 LONG 列值返回到变量中

COLUMN_VALUE_LONG 过程定义将从游标接收 LONG 值的部分的变量。

语法

```
►►—COLUMN_VALUE_LONG—(—c—,—position—,—length—,————————————————►
►—offset—,—value—,—value_length—)————————————————————————————►◄
```

参数

c 类型为 INTEGER 的输入参数，用于指定将数据返回到所定义变量的游标的标识。

position
类型为 INTEGER 的输入参数，用于指定所返回数据在游标中的位置。游标中的第一个值为位置 1。

length
类型为 INTEGER 的输入参数，用于指定要从 *offset* 开始检索的 LONG 数据的期望字节数。

offset
类型为 INTEGER 的输入参数，用于指定 LONG 值中开始检索数据的位置。

value
类型为 CLOB(32760) 的输出参数，用于指定接收先前访存调用中的游标所返回数据的变量。

value_length
类型为 INTEGER 的输出参数，用于返回所返回数据的实际长度。

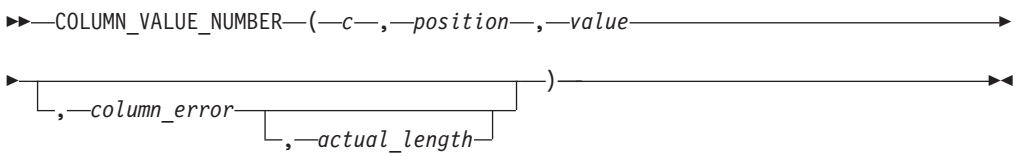
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

COLUMN_VALUE_NUMBER 过程 - 将 DECFLOAT 列值返回到变量中

COLUMN_VALUE_NUMBER 过程定义将从游标接收 DECFLOAT 值的变量。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定将数据返回到所定义变量的游标的标识。

position
类型为 INTEGER 的输入参数，用于指定所返回数据在游标中的位置。游标中的第一个值为位置 1。

value
类型为 DECFLOAT 的输出参数，用于指定接收先前访存调用中的游标所返回数据的变量。

column_error
类型为 INTEGER 的可选输出参数，用于返回与列相关联的 SQLCODE（如果存在）。

actual_length
类型为 INTEGER 的可选输出参数，用于返回任何截断之前的数据实际长度。

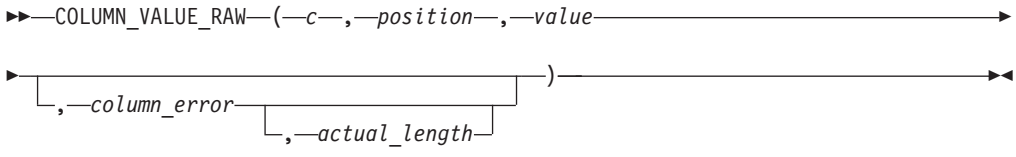
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

COLUMN_VALUE_RAW 过程 - 将 RAW 列值返回到变量中

COLUMN_VALUE_RAW 过程定义将从游标接收 RAW 值的变量。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定将数据返回到所定义变量的游标的标识。

position
类型为 INTEGER 的输入参数，用于指定所返回数据在游标中的位置。游标中的第一个值为位置 1。

value
类型为 BLOB(32767) 的输出参数，用于指定接收先前访存调用中的游标所返回数据的变量。

column_error
类型为 INTEGER 的可选输出参数，用于返回与列相关联的 SQLCODE（如果存在）。

actual_length
类型为 INTEGER 的可选输出参数，用于返回任何截断之前的数据实际长度。

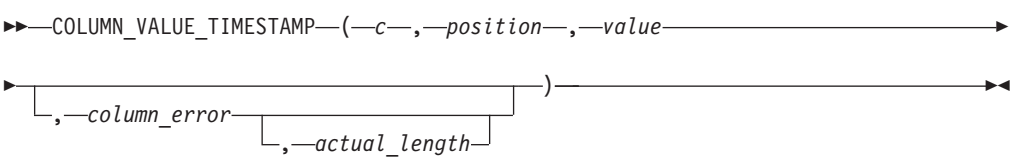
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

COLUMN_VALUE_TIMESTAMP 过程 - 将 TIMESTAMP 列值返回到变量中

COLUMN_VALUE_TIMESTAMP 过程定义将从游标接收 TIMESTAMP 值的变量。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定将数据返回到所定义变量的游标的标识。

position
类型为 INTEGER 的输入参数，用于指定所返回数据在游标中的位置。游标中的第一个值为位置 1。

value
类型为 TIMESTAMP 的输出参数，用于指定接收先前访存调用中的游标所返回数据的变量。

column_error
类型为 INTEGER 的输出参数，用于返回与列相关联的 SQLCODE（如果存在）。

actual_length
类型为 INTEGER 的输出参数，用于返回任何截断之前的数据实际长度。

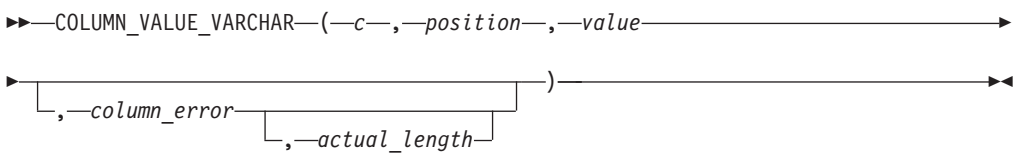
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

COLUMN_VALUE_VARCHAR 过程 - 将 VARCHAR 列值返回到变量中

COLUMN_VALUE_VARCHAR 过程定义将从游标接收 VARCHAR 值的变量。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定将数据返回到所定义变量的游标的标识。

position
类型为 INTEGER 的输入参数，用于指定所返回数据在游标中的位置。游标中的第一个值为位置 1。

value
类型为 VARCHAR(32672) 的输出参数，用于指定接收先前访存调用中的游标所返回数据的变量。

column_error
类型为 INTEGER 的输出参数，用于返回与列相关联的 SQLCODE（如果存在）。

actual_length
类型为 INTEGER 的输出参数，用于返回任何截断之前的数据实际长度。

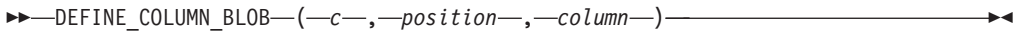
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

DEFINE_COLUMN_BLOB - 定义 SELECT 列表中的 BLOB 列

DEFINE_COLUMN_BLOB 过程定义 SELECT 列表中要在游标中返回和检索的 BLOB 列或表达式。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定与 SELECT 命令相关联的游标句柄。

position
类型为 INTEGER 的输入参数，用于指定要定义的列或表达式在 SELECT 列表中的位置。

column
类型为 BLOB(2G) 的输入参数。

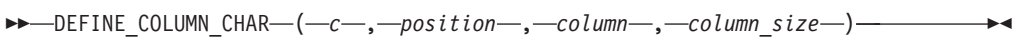
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

DEFINE_COLUMN_CHAR 过程 - 定义 SELECT 列表中的 CHAR 列

DEFINE_COLUMN_CHAR 过程定义 SELECT 列表中要在游标中返回和检索的 CHAR 列或表达式。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定与 SELECT 命令相关联的游标句柄。

position
类型为 INTEGER 的输入参数，用于指定要定义的列或表达式在 SELECT 列表中的位置。

column
类型为 CHAR(254) 的输入参数。

column_size
类型为 INTEGER 的输入参数，用于指定返回数据的最大长度。大小超过 *column_size* 的返回数据将截断为 *column_size* 个字符。

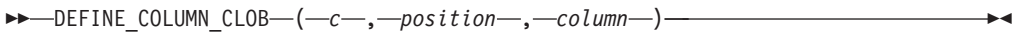
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

DEFINE_COLUMN_CLOB - 定义 SELECT 列表中的 CLOB 列

DEFINE_COLUMN_CLOB 过程定义 SELECT 列表中要在游标中返回和检索的 CLOB 列或表达式。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定与 SELECT 命令相关联的游标句柄。

position
类型为 INTEGER 的输入参数，用于指定要定义的列或表达式在 SELECT 列表中的位置。

column
类型为 CLOB(2G) 的输入参数。

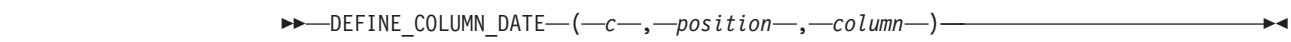
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

DEFINE_COLUMN_DATE - 定义 SELECT 列表中的 DATE 列

DEFINE_COLUMN_DATE 过程定义 SELECT 列表中要在游标中返回和检索的 DATE 列或表达式。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定与 SELECT 命令相关联的游标句柄。

position
类型为 INTEGER 的输入参数，用于指定要定义的列或表达式在 SELECT 列表中的位置。

column
类型为 DATE 的输入参数。

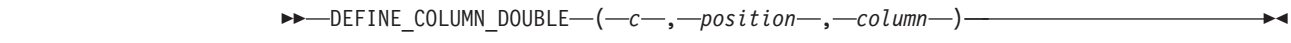
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

DEFINE_COLUMN_DOUBLE - 定义 SELECT 列表中的 DOUBLE 列

DEFINE_COLUMN_DOUBLE 过程定义 SELECT 列表中要在游标中返回和检索的 DOUBLE 列或表达式。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定与 SELECT 命令相关联的游标句柄。

position
类型为 INTEGER 的输入参数，用于指定要定义的列或表达式在 SELECT 列表中的位置。

column
类型为 DOUBLE 的输入参数。

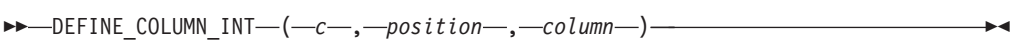
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

DEFINE_COLUMN_INT - 定义 SELECT 列表中的 INTEGER 列

DEFINE_COLUMN_INT 过程定义 SELECT 列表中要在游标中返回和检索的 INTEGER 列或表达式。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定与 SELECT 命令相关联的游标句柄。

position
类型为 INTEGER 的输入参数，用于指定要定义的列或表达式在 SELECT 列表中的位置。

column
类型为 INTEGER 的输入参数。

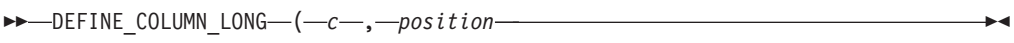
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

DEFINE_COLUMN_LONG 过程 - 定义 SELECT 列表中的 LONG 列

DEFINE_COLUMN_LONG 过程定义 SELECT 列表中要在游标中返回和检索的 LONG 列或表达式。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定与 SELECT 命令相关联的游标句柄。

position
类型为 INTEGER 的输入参数，用于指定要定义的列或表达式在 SELECT 列表中的位置。

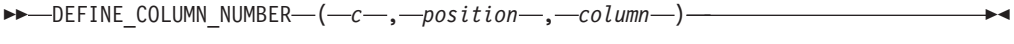
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

DEFINE_COLUMN_NUMBER 过程 - 定义 SELECT 列表中的 DECFLOAT 列

DEFINE_COLUMN_NUMBER 过程定义 SELECT 列表中要在游标中返回和检索的 DECFLOAT 列或表达式。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定与 SELECT 命令相关联的游标句柄。

position
类型为 INTEGER 的输入参数，用于指定要定义的列或表达式在 SELECT 列表中的位置。

column
类型为 DECFLOAT 的输入参数。

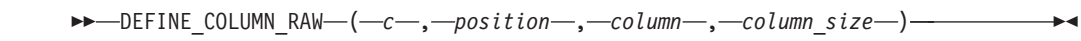
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

DEFINE_COLUMN_RAW 过程 - 定义 SELECT 列表中的 RAW 列或表达式

DEFINE_COLUMN_RAW 过程定义 SELECT 列表中要在游标中返回和检索的 RAW 列或表达式。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定与 SELECT 命令相关联的游标句柄。

position
类型为 INTEGER 的输入参数，用于指定要定义的列或表达式在 SELECT 列表中的位置。

column
类型为 BLOB(32767) 的输入参数。

column_size
类型为 INTEGER 的输入参数，用于指定返回数据的最大长度。大小超过 *column_size* 的返回数据将截断为 *column_size* 个字符。

授权

对 DBMS_SQL 模块的 EXECUTE 特权。

DEFINE_COLUMN_TIMESTAMP - 定义 SELECT 列表中的 TIMESTAMP 列

DEFINE_COLUMN_TIMESTAMP 过程定义 SELECT 列表中要在游标中返回和检索的 TIMESTAMP 列或表达式。

语法

参数

c 类型为 INTEGER 的输入参数，用于指定与 SELECT 命令相关联的游标句柄。

position
类型为 INTEGER 的输入参数，用于指定要定义的列或表达式在 SELECT 列表中的位置。

column
类型为 TIMESTAMP 的输入参数。

授权

对 DBMS_SQL 模块的 EXECUTE 特权。

DEFINE_COLUMN_VARCHAR 过程 - 定义 SELECT 列表中的 VARCHAR 列

DEFINE_COLUMN_VARCHAR 过程定义 SELECT 列表中要在游标中返回和检索的 VARCHAR 列或表达式。

语法

参数

c 类型为 INTEGER 的输入参数，用于指定与 SELECT 命令相关联的游标句柄。

position
类型为 INTEGER 的输入参数，用于指定要定义的列或表达式在 SELECT 列表中的位置。

column
类型为 VARCHAR(32672) 的输入参数。

column_size
类型为 INTEGER 的输入参数，用于指定返回数据的最大长度。大小超过 *column_size* 的返回数据将截断为 *column_size* 个字符。

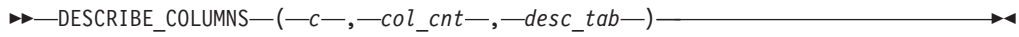
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

DESCRIBE_COLUMNS 过程 - 检索 SELECT 列表中的列的描述

DESCRIBE_COLUMNS 过程能够通过游标检索 SELECT 列表中的列的描述。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定要描述其列的游标的标识。

col_cnt
类型为 INTEGER 的输出参数，用于返回游标的 SELECT 列表中的列数。

desc_tab
类型为 DESC_TAB 的输出参数，用于描述列元数据。DESC_TAB 数组提供指定游标中的每列的信息。

授权

对 DBMS_SQL 模块的 EXECUTE 特权。

用法说明

此过程需要页大小为 4K 的用户临时表空间；否则它会返回 SQL0286N 错误。可使用此命令创建用户临时表空间:

```
CREATE USER TEMPORARY TABLESPACE DBMS_SQL_TEMP_TBS
```

DESC_TAB 是一组有关列信息的 DESC_REC 记录:

表 25. DESC_REC 记录中的 DESC_TAB 定义

记录名	描述
col_type	按 C 和 C++ 嵌入式 SQL 应用程序中的受支持 SQL 数据类型中定义的 SQL 数据类型。
col_max_len	该列的最大长度。
col_name	列名。
col_name_len	列名的长度。
col_schema	始终为 NULL。
col_schema_name_len	始终为 NULL。
col_precision	按数据库中定义的列的精度。如果 col_type 表示图形或 DBCLOB SQL 数据类型，那么此变量指示列可保存的最大双字节字符数。
col_scale	按数据库中定义的列的小数位（仅适用于 DECIMAL、NUMERIC 和 TIMESTAMP）。
col_charsetid	始终为 NULL。
col_charsetform	始终为 NULL。
col_null_ok	可空指示符。如果该列可空，那么其值为 1，否则为 0。

```
ALTER MODULE SYSIBMADM.DBMS_SQL PUBLISH TYPE DESC_REC AS ROW
(
  col_type INTEGER,
  col_max_len INTEGER,
  col_name VARCHAR(128),
  col_name_len INTEGER,
  col_schema_name VARCHAR(128),
  col_schema_name_len INTEGER,
```

```
  col_precision INTEGER,
  col_scale INTEGER,
  col_charsetid INTEGER,
  col_charsetform INTEGER,
  col_null_ok INTEGER
);

ALTER MODULE SYSIBMADM.DBMS_SQL PUBLISH TYPE DESC_TAB AS DESC_REC ARRAY[INTEGER];
```

示例

示例 1: 以下示例描述“EMP”表中的 empno、ename、hiredate 和 sal 列。

```
SET SERVEROUTPUT ON@

BEGIN
  DECLARE handle INTEGER;
  DECLARE col_cnt INTEGER;
  DECLARE col DBMS_SQL.DESC_TAB;
  DECLARE i INTEGER DEFAULT 1;
  DECLARE CUR1 CURSOR FOR S1;

  CALL DBMS_SQL.OPEN_CURSOR( handle );
  CALL DBMS_SQL.PARSE( handle,
      'SELECT empno, firstnme, lastname, salary
        FROM employee', DBMS_SQL.NATIVE );
  CALL DBMS_SQL.DESCRIBE_COLUMNS( handle, col_cnt, col );

  IF col_cnt > 0 THEN
    CALL DBMS_OUTPUT.PUT_LINE( 'col_cnt = ' || col_cnt );
    CALL DBMS_OUTPUT.NEW_LINE();
    fetchLoop: LOOP
      IF i > col_cnt THEN
        LEAVE fetchLoop;
      END IF;

      CALL DBMS_OUTPUT.PUT_LINE( 'i = ' || i );
      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_name = ' || col[i].col_name );
      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_name_len = ' ||
          NVL(col[i].col_name_len, 'NULL') );
      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_schema_name = ' ||
          NVL( col[i].col_schema_name, 'NULL' ) );

      IF col[i].col_schema_name_len IS NULL THEN
        CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_schema_name_len = NULL' );
      ELSE
        CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_schema_name_len = ' ||
            col[i].col_schema_name_len);
      END IF;

      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_type = ' || col[i].col_type );
      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_max_len = ' || col[i].col_max_len );
      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_precision = ' || col[i].col_precision );
      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_scale = ' || col[i].col_scale );

      IF col[i].col_charsetid IS NULL THEN
        CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_charsetid = NULL' );
      ELSE
        CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_charsetid = ' || col[i].col_charsetid );
      END IF;

      IF col[i].col_charsetform IS NULL THEN
        CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_charsetform = NULL' );
      ELSE
        CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_charsetform = ' || col[i].col_charsetform );
      END IF;
```

```
      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_null_ok = ' || col[i].col_null_ok );
      CALL DBMS_OUTPUT.NEW_LINE();
      SET i = i + 1;
    END LOOP;
  END IF;
END@
```

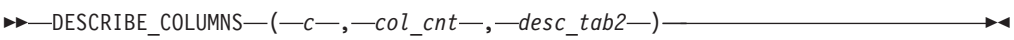
输出:

```
col_cnt = 4

i = 1
col[i].col_name = EMPNO
col[i].col_name_len = 5
col[i].col_schema_name = NULL
col[i].col_schema_name_len = NULL
col[i].col_type = 452
col[i].col_max_len = 6
col[i].col_precision = 6
col[i].col_scale = 0
col[i].col_charsetid = NULL
col[i].col_charsetform = NULL
col[i].col_null_ok = 0

i = 2
col[i].col_name = FIRSTNME
col[i].col_name_len = 8
col[i].col_schema_name = NULL
col[i].col_schema_name_len = NULL
col[i].col_type = 448
col[i].col_max_len = 12
col[i].col_precision = 12
col[i].col_scale = 0
col[i].col_charsetid = NULL
col[i].col_charsetform = NULL
col[i].col_null_ok = 0

i = 3
col[i].col_name = LASTNAME
col[i].col_name_len = 8
col[i].col_schema_name = NULL
col[i].col_schema_name_len = NULL
col[i].col_type = 448
col[i].col_max_len = 15
col[i].col_precision = 15
col[i].col_scale = 0
col[i].col_charsetid = NULL
col[i].col_charsetform = NULL
col[i].col_null_ok = 0

i = 4
col[i].col_name = SALARY
col[i].col_name_len = 6
col[i].col_schema_name = NULL
col[i].col_schema_name_len = NULL
col[i].col_type = 484
col[i].col_max_len = 5
col[i].col_precision = 9
col[i].col_scale = 2
col[i].col_charsetid = NULL
col[i].col_charsetform = NULL
col[i].col_null_ok = 1
```

DESCRIBE_COLUMNS2 过程 - 检索 SELECT 列表中的列名的描述

DESCRIBE_COLUMNS2 过程能够通过游标检索 SELECT 列表中的列的描述。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定要描述其列的游标的标识。

col_cnt
类型为 INTEGER 的输出参数，用于返回游标的 SELECT 列表中的列数。

desc_tab
类型为 DESC_TAB2 的输出参数，用于描述列元数据。DESC_TAB2 数组提供指定游标中的每列的信息

授权

对 DBMS_SQL 模块的 EXECUTE 特权。

用法说明

此过程需要页大小为 4K 的用户临时表空间；否则它会返回 SQL0286N 错误。可使用此命令创建用户临时表空间:

```
CREATE USER TEMPORARY TABLESPACE DBMS_SQL_TEMP_TBS
```

DESC_TAB2 是一组有关列信息的 DESC_REC2 记录:

表 26. DESC_REC2 记录中的 DESC_TAB2 定义

记录名	描述
col_type	按 C 和 C++ 嵌入式 SQL 应用程序中的受支持 SQL 数据类型中定义的 SQL 数据类型。
col_max_len	该列的最大长度。
col_name	列名。
col_name_len	列名的长度。
col_schema	始终为 NULL。
col_schema_name_len	始终为 NULL。
col_precision	按数据库中定义的列的精度。如果 col_type 表示图形或 DBCLOB SQL 数据类型，那么此变量指示列可保存的最大双字节字符数。
col_scale	按数据库中定义的列的小数位（仅适用于 DECIMAL、NUMERIC 和 TIMESTAMP）。
col_charsetid	始终为 NULL。
col_charsetform	始终为 NULL。
col_null_ok	可空指示符。如果该列可空，那么其值为 1，否则为 0。

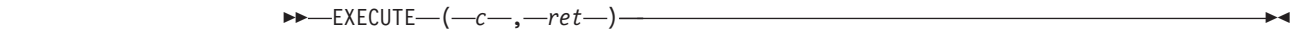
```
ALTER MODULE SYSIBMADM.DBMS_SQL PUBLISH TYPE DESC_REC2 AS ROW
(
col_type INTEGER,
col_max_len INTEGER,
col_name VARCHAR(128),
col_name_len INTEGER,
```

```
col_schema_name VARCHAR(128),
col_schema_name_len INTEGER,
col_precision INTEGER,
col_scale INTEGER,
col_charsetid INTEGER,
col_charsetform INTEGER,
col_null_ok INTEGER
);

ALTER MODULE SYSIBMADM.DBMS_SQL PUBLISH TYPE DESC_TAB2 AS DESC_REC2 ARRAY[INTEGER];
```

EXECUTE 过程 - 运行已解析 SQL 语句

EXECUTE 函数执行已解析 SQL 语句。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定要执行的已解析 SQL 语句的游标标识。

ret
类型为 INTEGER 的输出参数，如果 SQL 命令为 DELETE、INSERT 或 UPDATE，那么此参数返回处理的行数；否则返回 0。

授权

对 DBMS_SQL 模块的 EXECUTE 特权。

用法说明

可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

示例

示例 1: 以下匿名块在“DEPT”表中插入一行。

```
SET SERVEROUTPUT ON@

CREATE TABLE dept (
  deptno DECIMAL(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
  dname  VARCHAR(14) NOT NULL,
  loc    VARCHAR(13),
  CONSTRAINT dept_dname_uq UNIQUE( deptno, dname )
)@

BEGIN
  DECLARE curid INTEGER;
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status        INTEGER;

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  SET v_sql = 'INSERT INTO dept VALUES (50, ''HR'', ''LOS ANGELES'')';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@
```

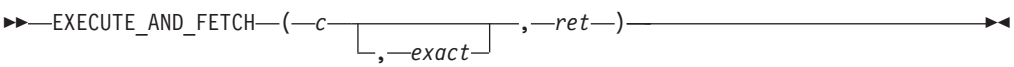
此示例生成以下输出:

```
SET SERVEROUTPUT ON
DB20000I  SET SERVEROUTPUT 命令成功完成。

CREATE TABLE dept
( deptno DECIMAL(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
  dname  VARCHAR(14) NOT NULL,
  loc    VARCHAR(13),
  CONSTRAINT dept_dname_uq UNIQUE( deptno, dname ) )
DB20000I  The SQL command completed successfully.

BEGIN
  DECLARE curid INTEGER;
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status        INTEGER;

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  SET v_sql = 'INSERT INTO dept VALUES (50, ''HR'', ''LOS ANGELES'')';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END
DB20000I  The SQL command completed successfully.

Number of rows processed: 1
```

EXECUTE_AND_FETCH 过程 - 运行已解析 SELECT 命令并访存一行

EXECUTE_AND_FETCH 过程执行已解析的 SELECT 命令并访存一行。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定要执行的 SELECT 命令的游标标识。

exact
类型为 INTEGER 的可选参数。如果设置为 1，并且结果集中的行数未正好等于 1，那么会抛出异常。如果设置为 0，那么不会抛出异常。缺省值为 0。如果 *exact* 设置为 1 并且结果集中没有任何行，那么会抛出 NO_DATA_FOUND (SQL0100W) 异常。如果 *exact* 设置为 1 并且结果集中有多行，那么会抛出 TOO_MANY_ROWS (SQL0811N) 异常。

ret
类型为 INTEGER 的输出参数，如果成功访存行，那么返回 1，如果未访存任何行，那么返回 0。

授权

对 DBMS_SQL 模块的 EXECUTE 特权。

用法说明

可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

示例

示例 1: 以下存储过程使用职员姓名来通过 EXECUTE_AND_FETCH 函数检索一个职员。如果找不到该职员，或者有多个同名职员，那么会抛出异常。

```
SET SERVEROUTPUT ON@

CREATE TABLE emp (
  empno    DECIMAL(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
  ename    VARCHAR(10),
  job      VARCHAR(9),
  mgr      DECIMAL(4),
  hiredate TIMESTAMP(0),
  sal      DECIMAL(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
  comm     DECIMAL(7,2) )@

INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'1980-12-17',800,NULL)@
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'1981-02-20',1600,300)@
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'1981-02-22',1250,500)@
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'1981-04-02',2975,NULL)@
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'1981-09-28',1250,1400)@

CREATE OR REPLACE PROCEDURE select_by_name(
IN p_ename ANCHOR TO emp.ename)
BEGIN
  DECLARE curid INTEGER;
  DECLARE v_empno ANCHOR TO emp.empno;
  DECLARE v_hiredate ANCHOR TO emp.hiredate;
  DECLARE v_sal ANCHOR TO emp.sal;
  DECLARE v_comm ANCHOR TO emp.comm;
  DECLARE v_disp_date VARCHAR(10);
  DECLARE v_sql VARCHAR(120);
  DECLARE v_status        INTEGER;
  SET v_sql = 'SELECT empno, hiredate, sal, NVL(comm, 0)
      FROM emp e WHERE ename = :p_ename ';
  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.BIND_VARIABLE_VARCHAR(curid, ':p_ename', UPPER(p_ename));
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.DEFINE_COLUMN_DATE(curid, 2, v_hiredate);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 3, v_sal);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 4, v_comm);
  CALL DBMS_SQL.EXECUTE_AND_FETCH(curid, 1 /*True*/, v_status);
  CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.COLUMN_VALUE_DATE(curid, 2, v_hiredate);
  CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 3, v_sal);
  CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 4, v_comm);
  SET v_disp_date = TO_CHAR(v_hiredate, 'MM/DD/YYYY');
  CALL DBMS_OUTPUT.PUT_LINE('Number    : ' || v_empno);
  CALL DBMS_OUTPUT.PUT_LINE('Name      : ' || UPPER(p_ename));
  CALL DBMS_OUTPUT.PUT_LINE('Hire Date : ' || v_disp_date);
  CALL DBMS_OUTPUT.PUT_LINE('Salary    : ' || v_sal);
  CALL DBMS_OUTPUT.PUT_LINE('Commission: ' || v_comm);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@

CALL select_by_name( 'MARTIN' )@
```

此示例生成以下输出:

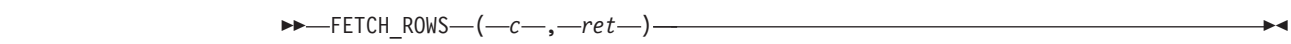
```
SET SERVEROUTPUT ON
DB20000I  SET SERVEROUTPUT 命令成功完成。

CREATE TABLE emp
 ( empno    DECIMAL(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
   ename    VARCHAR(10),
   job      VARCHAR(9),
```

```
   mgr      DECIMAL(4),
   hiredate TIMESTAMP(0),
   sal      DECIMAL(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
   comm     DECIMAL(7,2) )
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'1980-12-17',800,NULL)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'1981-02-20',1600,300)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'1981-02-22',1250,500)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'1981-04-02',2975,NULL)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'1981-09-28',1250,1400)
DB20000I  The SQL command completed successfully.

CREATE OR REPLACE PROCEDURE select_by_name(
IN p_ename ANCHOR TO emp.ename)
BEGIN
  DECLARE curid INTEGER;
  DECLARE v_empno ANCHOR TO emp.empno;
  DECLARE v_hiredate ANCHOR TO emp.hiredate;
  DECLARE v_sal ANCHOR TO emp.sal;
  DECLARE v_comm ANCHOR TO emp.comm;
  DECLARE v_disp_date VARCHAR(10);
  DECLARE v_sql VARCHAR(120);
  DECLARE v_status        INTEGER;
  SET v_sql = 'SELECT empno, hiredate, sal, NVL(comm, 0)
      FROM emp e WHERE ename = :p_ename ';
  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.BIND_VARIABLE_VARCHAR(curid, ':p_ename', UPPER(p_ename));
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.DEFINE_COLUMN_DATE(curid, 2, v_hiredate);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 3, v_sal);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 4, v_comm);
  CALL DBMS_SQL.EXECUTE_AND_FETCH(curid, 1 /*True*/, v_status);
  CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.COLUMN_VALUE_DATE(curid, 2, v_hiredate);
  CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 3, v_sal);
  CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 4, v_comm);
  SET v_disp_date = TO_CHAR(v_hiredate, 'MM/DD/YYYY');
  CALL DBMS_OUTPUT.PUT_LINE('Number    : ' || v_empno);
  CALL DBMS_OUTPUT.PUT_LINE('Name      : ' || UPPER(p_ename));
  CALL DBMS_OUTPUT.PUT_LINE('Hire Date : ' || v_disp_date);
  CALL DBMS_OUTPUT.PUT_LINE('Salary    : ' || v_sal);
  CALL DBMS_OUTPUT.PUT_LINE('Commission: ' || v_comm);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END
DB20000I  The SQL command completed successfully.

CALL select_by_name( 'MARTIN' )

  Return Status = 0

Number    : 7654
Name      : MARTIN
Hire Date : 09/28/1981
Salary    : 1250.00
Commission: 1400.00
```

FETCH_ROWS 过程 - 从游标检索行

FETCH_ROWS 函数从游标检索行

语法



参数

c 类型为 INTEGER 的输入参数，用于指定要从中访存行的游标的标识。

ret

类型为 INTEGER 的输出参数，如果成功访存行，那么返回 1，如果未访存任何行，那么返回 0。

授权

对 DBMS_SQL 模块的 EXECUTE 特权。

用法说明

可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

示例

示例 1: 以下示例从“EMP”表访存行并显示结果。

```
SET SERVEROUTPUT ON@

CREATE TABLE emp (
  empno    DECIMAL(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
  ename    VARCHAR(10),
  job      VARCHAR(9),
  mgr      DECIMAL(4),
  hiredate TIMESTAMP(0),
  sal      DECIMAL(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
  comm     DECIMAL(7,2) )@

INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'1980-12-17',800,NULL)@
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'1981-02-20',1600,300)@
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'1981-02-22',1250,500)@
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'1981-04-02',2975,NULL)@
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'1981-09-28',1250,1400)@

BEGIN
  DECLARE curid INTEGER;
  DECLARE v_empno DECIMAL(4);
  DECLARE v_ename VARCHAR(10);
  DECLARE v_hiredate DATE;
  DECLARE v_sal DECIMAL(7, 2);
  DECLARE v_comm DECIMAL(7, 2);
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status        INTEGER;
  DECLARE v_rowcount INTEGER;

  SET v_sql = 'SELECT empno, ename, hiredate, sal, ' || 'comm FROM emp';

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 1, v_empno);
```

```
  CALL DBMS_SQL.DEFINE_COLUMN_VARCHAR(curid, 2, v_ename, 10);
  CALL DBMS_SQL.DEFINE_COLUMN_DATE(curid, 3, v_hiredate);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 4, v_sal);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 5, v_comm);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME       HIREDATE    SAL
        COMM');
  CALL DBMS_OUTPUT.PUT_LINE('-----  ----------  ----------  --------
       ' || '--------');

  FETCH_LOOP: LOOP
    CALL DBMS_SQL.FETCH_ROWS(curid, v_status);

    IF v_status = 0 THEN
      LEAVE FETCH_LOOP;
    END IF;

    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 1, v_empno);
    CALL DBMS_SQL.COLUMN_VALUE_VARCHAR(curid, 2, v_ename);
    CALL DBMS_SQL.COLUMN_VALUE_DATE(curid, 3, v_hiredate);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 4, v_sal);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 5, v_comm);
    CALL DBMS_OUTPUT.PUT_LINE(v_empno || '   ' ||
            RPAD(v_ename, 10) || '  ' || TO_CHAR(v_hiredate,
            'yyyy-mm-dd') || ' ' || TO_CHAR(v_sal,
            '9,999.99') || ' ' || TO_CHAR(NVL(v_comm, 0),
            '9,999.99'));
  END LOOP FETCH_LOOP;

  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@
```

此示例生成以下输出:

```
SET SERVEROUTPUT ON
DB20000I  SET SERVEROUTPUT 命令成功完成。

CREATE TABLE emp (empno  DECIMAL(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename    VARCHAR(10), job      VARCHAR(9), mgr      DECIMAL(4),
    hiredate TIMESTAMP(0),
    sal      DECIMAL(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
    comm     DECIMAL(7,2) )
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'1980-12-17',800,NULL)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'1981-02-20',1600,300)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'1981-02-22',1250,500)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'1981-04-02',2975,NULL)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'1981-09-28',1250,1400)
DB20000I  The SQL command completed successfully.

BEGIN
  DECLARE curid INTEGER;
  DECLARE v_empno DECIMAL(4);
  DECLARE v_ename VARCHAR(10);
  DECLARE v_hiredate DATE;
  DECLARE v_sal DECIMAL(7, 2);
  DECLARE v_comm DECIMAL(7, 2);
  DECLARE v_sql VARCHAR(50);
```

```
  DECLARE v_status        INTEGER;
  DECLARE v_rowcount INTEGER;

  SET v_sql = 'SELECT empno, ename, hiredate, sal, ' || 'comm FROM emp';

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.DEFINE_COLUMN_VARCHAR(curid, 2, v_ename, 10);
  CALL DBMS_SQL.DEFINE_COLUMN_DATE(curid, 3, v_hiredate);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 4, v_sal);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 5, v_comm);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME       HIREDATE    SAL
      COMM');
  CALL DBMS_OUTPUT.PUT_LINE('-----  ----------  ----------  --------
      ' || '--------');

  FETCH_LOOP: LOOP
    CALL DBMS_SQL.FETCH_ROWS(curid, v_status);

    IF v_status = 0 THEN
      LEAVE FETCH_LOOP;
    END IF;

    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 1, v_empno);
    CALL DBMS_SQL.COLUMN_VALUE_VARCHAR(curid, 2, v_ename);
    CALL DBMS_SQL.COLUMN_VALUE_DATE(curid, 3, v_hiredate);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 4, v_sal);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 5, v_comm);
    CALL DBMS_OUTPUT.PUT_LINE(v_empno || '   ' || RPAD(v_ename,
        10) || '  ' || TO_CHAR(v_hiredate,
        'yyyy-mm-dd') || ' ' || TO_CHAR(v_sal,
        '9,999.99') || ' ' || TO_CHAR(NVL(v_comm,
         0), '9,999.99'));
  END LOOP FETCH_LOOP;

  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END
DB20000I  The SQL command completed successfully.

EMPNO  ENAME       HIREDATE    SAL       COMM
-----  ----------  ----------  --------  --------
7369   SMITH       1980-12-17    800.00      0.00
7499   ALLEN       1981-02-20  1,600.00    300.00
7521   WARD        1981-02-22  1,250.00    500.00
7566   JONES       1981-04-02  2,975.00      0.00
7654   MARTIN      1981-09-28  1,250.00  1,400.00
```

IS_OPEN 过程 - 检查游标是否已打开

IS_OPEN 函数能够测试给定游标是否已打开。

语法

参数

c 类型为 INTEGER 的输入参数，用于指定要测试的游标的标识。

ret
类型为 BOOLEAN 的输出参数，用于指示指定的文件已打开 (TRUE) 还是已关闭 (FALSE)。

授权

对 DBMS_SQL 模块的 EXECUTE 特权。

用法说明

可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

LAST_ROW_COUNT 过程 - 返回访存的累积行数

LAST_ROW_COUNT 过程返回已访存的行数。

语法

参数

ret
类型为 INTEGER 的输出参数，用于返回当前会话中至今访存的行数。对 DBMS_SQL.PARSE 的调用会使计数器重置。

授权

对 DBMS_SQL 模块的 EXECUTE 特权。

用法说明

可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

示例

示例 1: 以下示例使用 LAST_ROW_COUNT 过程来显示查询中访存的总行数。

```
SET SERVEROUTPUT ON@

CREATE TABLE emp (
  empno    DECIMAL(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
  ename    VARCHAR(10),
  job      VARCHAR(9),
  mgr      DECIMAL(4),
  hiredate TIMESTAMP(0),
  sal      DECIMAL(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
  comm     DECIMAL(7,2) )@

INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'1980-12-17',800,NULL)@
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'1981-02-20',1600,300)@
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'1981-02-22',1250,500)@
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'1981-04-02',2975,NULL)@
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'1981-09-28',1250,1400)@

BEGIN
  DECLARE curid INTEGER;
  DECLARE v_empno DECIMAL(4);
```

```
  DECLARE v_ename VARCHAR(10);
  DECLARE v_hiredate DATE;
  DECLARE v_sal DECIMAL(7, 2);
  DECLARE v_comm DECIMAL(7, 2);
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status        INTEGER;
  DECLARE v_rowcount INTEGER;

  SET v_sql = 'SELECT empno, ename, hiredate, sal, ' || 'comm FROM emp';

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.DEFINE_COLUMN_VARCHAR(curid, 2, v_ename, 10);
  CALL DBMS_SQL.DEFINE_COLUMN_DATE(curid, 3, v_hiredate);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 4, v_sal);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 5, v_comm);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME       HIREDATE    SAL
     COMM');
  CALL DBMS_OUTPUT.PUT_LINE('-----  ----------  ----------  --------
     ' || '--------');

  FETCH_LOOP: LOOP
    CALL DBMS_SQL.FETCH_ROWS(curid, v_status);

    IF v_status = 0 THEN
      LEAVE FETCH_LOOP;
    END IF;

    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 1, v_empno);
    CALL DBMS_SQL.COLUMN_VALUE_VARCHAR(curid, 2, v_ename);
    CALL DBMS_SQL.COLUMN_VALUE_DATE(curid, 3, v_hiredate);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 4, v_sal);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 5, v_comm);
    CALL DBMS_OUTPUT.PUT_LINE(v_empno || '   ' || RPAD(v_ename,
       10) || '  ' || TO_CHAR(v_hiredate,
       'yyyy-mm-dd') || ' ' || TO_CHAR(v_sal,
       '9,999.99') || ' ' || TO_CHAR(NVL(v_comm,
       0), '9,999.99'));
  END LOOP FETCH_LOOP;

  CALL DBMS_SQL.LAST_ROW_COUNT( v_rowcount );
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows: ' || v_rowcount);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@
```

此示例生成以下输出:

```
SET SERVEROUTPUT ON
DB20000I  SET SERVEROUTPUT 命令成功完成。

CREATE TABLE emp ( empno    DECIMAL(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
   ename    VARCHAR(10), job      VARCHAR(9),
   mgr      DECIMAL(4),
   hiredate TIMESTAMP(0),
   sal      DECIMAL(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
   comm     DECIMAL(7,2) )
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'1980-12-17',800,NULL)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'1981-02-20',1600,300)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'1981-02-22',1250,500)
```

```
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'1981-04-02',2975,NULL)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'1981-09-28',1250,1400)
DB20000I  The SQL command completed successfully.

BEGIN
  DECLARE curid INTEGER;
  DECLARE v_empno DECIMAL(4);
  DECLARE v_ename VARCHAR(10);
  DECLARE v_hiredate DATE;
  DECLARE v_sal DECIMAL(7, 2);
  DECLARE v_comm DECIMAL(7, 2);
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status        INTEGER;
  DECLARE v_rowcount INTEGER;

  SET v_sql = 'SELECT empno, ename, hiredate, sal, ' || 'comm FROM emp';

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.DEFINE_COLUMN_VARCHAR(curid, 2, v_ename, 10);
  CALL DBMS_SQL.DEFINE_COLUMN_DATE(curid, 3, v_hiredate);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 4, v_sal);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 5, v_comm);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME       HIREDATE    SAL
     COMM');
  CALL DBMS_OUTPUT.PUT_LINE('-----  ----------  ----------  --------
     ' || '--------');

  FETCH_LOOP: LOOP
    CALL DBMS_SQL.FETCH_ROWS(curid, v_status);

    IF v_status = 0 THEN
      LEAVE FETCH_LOOP;
    END IF;

    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 1, v_empno);
    CALL DBMS_SQL.COLUMN_VALUE_VARCHAR(curid, 2, v_ename);
    CALL DBMS_SQL.COLUMN_VALUE_DATE(curid, 3, v_hiredate);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 4, v_sal);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 5, v_comm);
    CALL DBMS_OUTPUT.PUT_LINE(
      v_empno || '   ' || RPAD(v_ename, 10) || '  ' || TO_CHAR(v_hiredate,
      'yyyy-mm-dd') || ' ' || TO_CHAR(v_sal,
      '9,999.99') || ' ' || TO_CHAR(NVL(v_comm,
      0), '9,999.99'));
  END LOOP FETCH_LOOP;

  CALL DBMS_SQL.LAST_ROW_COUNT( v_rowcount );
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows: ' || v_rowcount);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END
DB20000I  The SQL command completed successfully.

EMPNO  ENAME       HIREDATE    SAL       COMM
-----  ----------  ----------  --------  --------
7369   SMITH       1980-12-17    800.00      0.00
7499   ALLEN       1981-02-20  1,600.00    300.00
7521   WARD        1981-02-22  1,250.00    500.00
7566   JONES       1981-04-02  2,975.00      0.00
7654   MARTIN      1981-09-28  1,250.00  1,400.00
Number of rows: 5
```

OPEN_CURSOR 过程 - 打开游标

OPEN_CURSOR 过程创建新游标。

必须使用游标来解析和执行所有动态 SQL 语句。一旦打开了游标，就可再将该游标与相同或不同 SQL 语句配合使用。游标不必关闭然后重新打开就可再次使用。

语法

参数

c 类型为 INTEGER 的输出参数，用于指定新创建游标的标识。

授权

对 DBMS_SQL 模块的 EXECUTE 特权。

用法说明

可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

示例

示例 1: 以下示例创建新游标:

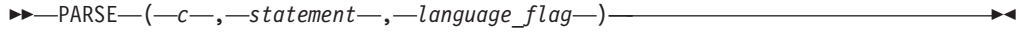
```
DECLARE
    curid           INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
             .
             .
             .
END;
```

PARSE 过程 - 解析 SQL 语句

PARSE 过程解析 SQL 语句。

如果 SQL 命令是 DDL 命令，那么会直接执行该命令而不需要运行 EXECUTE 过程。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定已打开游标的标识。

statement
要解析的 SQL 语句。

language_flag
提供此参数是为了与 Oracle 语法兼容。使用值 1 或 DBMS_SQL.native。

授权

对 DBMS_SQL 模块的 EXECUTE 特权。

用法说明

可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

示例

示例 1: 以下匿名块创建表 job。请注意，PARSE 过程会直接执行 DDL 语句而不需要执行单独的 EXECUTE 步骤。

```
SET SERVEROUTPUT ON@

BEGIN
  DECLARE curid INTEGER;
  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, 'CREATE TABLE job (jobno DECIMAL(3),
     ' || 'jname VARCHAR(9))', DBMS_SQL.native);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@
```

此示例生成以下输出:

```
SET SERVEROUTPUT ON
DB20000I  SET SERVEROUTPUT 命令成功完成。

BEGIN
  DECLARE curid INTEGER;
  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, 'CREATE TABLE job (jobno DECIMAL(3), ' ||
    'jname VARCHAR(9))', DBMS_SQL.native);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END
DB20000I  The SQL command completed successfully.
```

示例 2: 以下代码在表 job 中插入两行。

```
BEGIN
  DECLARE curid INTEGER;
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status        INTEGER;

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  SET v_sql = 'INSERT INTO job VALUES (100, ''ANALYST'')';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
  SET v_sql = 'INSERT INTO job VALUES (200, ''CLERK'')';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@
```

此示例生成以下输出:

```
BEGIN
  DECLARE curid INTEGER;
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status        INTEGER;

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  SET v_sql = 'INSERT INTO job VALUES (100, ''ANALYST'')';
```

```
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
  SET v_sql = 'INSERT INTO job VALUES (200, ''CLERK'')';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END
DB20000I  The SQL command completed successfully.

Number of rows processed: 1
Number of rows processed: 1
```

示例 3: 以下匿名块使用 DBMS_SQL 模块执行包含两个 INSERT 语句的块。请注意，块结尾包含终止分号，而先前示例中的各个 INSERT 语句没有终止分号。

```
BEGIN
  DECLARE curid INTEGER;
  DECLARE v_sql VARCHAR(100);
  DECLARE v_status        INTEGER;

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  SET v_sql = 'BEGIN ' || 'INSERT INTO job VALUES (300, ''MANAGER''); '
       || 'INSERT INTO job VALUES (400, ''SALESMAN''); ' || 'END;';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@
```

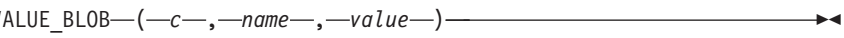
此示例生成以下输出:

```
BEGIN
  DECLARE curid INTEGER;
  DECLARE v_sql VARCHAR(100);
  DECLARE v_status        INTEGER;

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  SET v_sql = 'BEGIN ' || 'INSERT INTO job VALUES (300, ''MANAGER''); ' ||
    'INSERT INTO job VALUES (400, ''SALESMAN''); ' || 'END;';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END
DB20000I  The SQL command completed successfully.
```

VARIABLE_VALUE_BLOB 过程 - 返回 BLOB INOUT 或 OUT 参数的值

VARIABLE_VALUE_BLOB 过程能够返回 BLOB INOUT 或 OUT 参数的值。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定将返回绑定变量的游标的标识。

name
输入参数，用于指定绑定变量的名称。

value
类型为 BLOB(2G) 的输出参数，用于指定将接收值的变量。

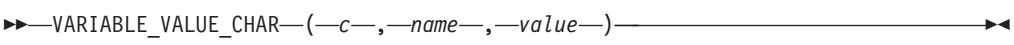
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

VARIABLE_VALUE_CHAR 过程 - 返回 CHAR INOUT 或 OUT 参数的值

VARIABLE_VALUE_CHAR 过程能够返回 CHAR INOUT 或 OUT 参数的值。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定将返回绑定变量的游标的标识。

name
输入参数，用于指定绑定变量的名称。

value
类型为 CHAR(254) 的输出参数，用于指定将接收值的变量。

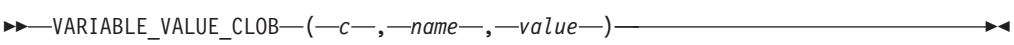
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

VARIABLE_VALUE_CLOB 过程 - 返回 CLOB INOUT 或 OUT 参数的值

VARIABLE_VALUE_CLOB 过程能够返回 CLOB INOUT 或 OUT 参数的值。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定将返回绑定变量的游标的标识。

name
输入参数，用于指定绑定变量的名称。

value
类型为 CLOB(2G) 的输出参数，用于指定将接收值的变量。

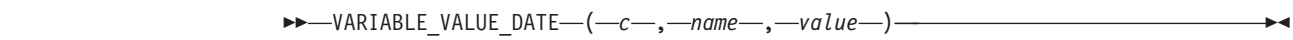
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

VARIABLE_VALUE_DATE 过程 - 返回 DATE INOUT 或 OUT 参数的值

VARIABLE_VALUE_DATE 过程能够返回 DATE INOUT 或 OUT 参数的值。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定将返回绑定变量的游标的标识。

name
输入参数，用于指定绑定变量的名称。

value
类型为 DATE 的输出参数，用于指定将接收值的变量。

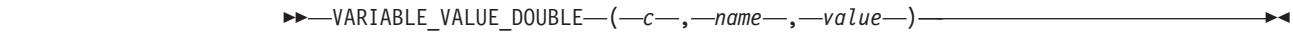
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

VARIABLE_VALUE_DOUBLE 过程 - 返回 DOUBLE INOUT 或 OUT 参数的值

VARIABLE_VALUE_DOUBLE 过程能够返回 DOUBLE INOUT 或 OUT 参数的值。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定将返回绑定变量的游标的标识。

name
输入参数，用于指定绑定变量的名称。

value
类型为 DOUBLE 的输出参数，用于指定将接收值的变量。

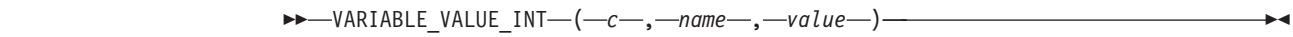
授权

对 DBMS_SQL 模块的 EXECUTE 特权。

VARIABLE_VALUE_INT 过程 - 返回 INTEGER INOUT 或 OUT 参数的值

VARIABLE_VALUE_INT 过程能够返回 INTEGER INOUT 或 OUT 参数的值。

语法



参数

c 类型为 INTEGER 的输入参数，用于指定将返回绑定变量的游标的标识。

name
输入参数，用于指定绑定变量的名称。

value
类型为 INTEGER 的输出参数，用于指定将接收值的变量。

授权

对 DBMS_SQL 模块的 EXECUTE 特权。

VARIABLE_VALUE_NUMBER 过程 - 返回 DECFLOAT INOUT 或 OUT 参数的值

VARIABLE_VALUE_NUMBER 过程能够返回 DECFLOAT INOUT 或 OUT 参数的值。

语法

```
►►—VARIABLE_VALUE_NUMBER—(—c—,—name—,—value—)————————————————————►◄
```

参数

c 类型为 INTEGER 的输入参数，用于指定将返回绑定变量的游标的标识。

name
输入参数，用于指定绑定变量的名称。

value
类型为 DECFLOAT 的输出参数，用于指定将接收值的变量。

授权

对 DBMS_SQL 模块的 EXECUTE 特权。

VARIABLE_VALUE_RAW 过程 - 返回 BLOB(32767) INOUT 或 OUT 参数的值

VARIABLE_VALUE_RAW 过程能够返回 BLOB(32767) INOUT 或 OUT 参数的值。

语法

```
►►—VARIABLE_VALUE_RAW—(—c—,—name—,—value—)————————————————————►◄
```

参数

c 类型为 INTEGER 的输入参数，用于指定将返回绑定变量的游标的标识。

name
输入参数，用于指定绑定变量的名称。

value
类型为 BLOB(32767) 的输出参数，用于指定将接收值的变量。

授权

对 DBMS_SQL 模块的 EXECUTE 特权。

VARIABLE_VALUE_TIMESTAMP 过程 - 返回 TIMESTAMP INOUT 或 OUT 参数的值

VARIABLE_VALUE_TIMESTAMP 过程能够返回 TIMESTAMP INOUT 或 OUT 参数的值。

语法

```
►►—VARIABLE_VALUE_TIMESTAMP—(—c—,—name—,—value—)————————————————►◄
```

参数

c 类型为 INTEGER 的输入参数，用于指定将返回绑定变量的游标的标识。

name
输入参数，用于指定绑定变量的名称。

value
类型为 TIMESTAMP 的输出参数，用于指定将接收值的变量。

授权

对 DBMS_SQL 模块的 EXECUTE 特权。

VARIABLE_VALUE_VARCHAR 过程 - 返回 VARCHAR INOUT 或 OUT 参数的值

VARIABLE_VALUE_VARCHAR 过程能够返回 VARCHAR INOUT 或 OUT 参数的值。

语法

```
►►—VARIABLE_VALUE_VARCHAR—(—c—,—name—,—value—)————————————————►◄
```

参数

c 类型为 INTEGER 的输入参数，用于指定将返回绑定变量的游标的标识。

name
输入参数，用于指定绑定变量的名称。

value
类型为 VARCHAR(32672) 的输出参数，用于指定将接收值的变量。

授权

对 DBMS_SQL 模块的 EXECUTE 特权。

第 29 章 DBMS_UTILITY 模块

DBMS_UTILITY 模块提供各种实用程序。

此模块的模式为 SYSIBMADM。

DBMS_UTILITY 模块包括下列例程。

表 27. DBMS_UTILITY 模块中可用的内置例程

例程名称	描述
ANALYZE_DATABASE 过程	分析数据库表、集群和索引。
ANALYZE_PART_OBJECT 过程	分析分区表或分区索引。
ANALYZE_SCHEMA 过程	分析模式表、集群和索引。
CANONICALIZE 过程	使字符串规范化（例如，除去空格）。
COMMA_TO_TABLE 过程	将用逗号定界的名称列表转换为名称表。
COMPILE_SCHEMA 过程	编译模式中的程序。
DB_VERSION 过程	获取数据库版本。
EXEC_DDL_STATEMENT 过程	执行 DDL 语句。
GET_CPU_TIME 函数	获取当前 CPU 时间。
GET_DEPENDENCY 过程	获取依赖于给定对象的对象。
GET_HASH_VALUE 函数	计算散列值。
GET_TIME 函数	获取当前时间。
NAME_RESOLVE 过程	解析给定名称。
NAME_TOKENIZE 过程	将给定名称解析为若干组成部分。
TABLE_TO_COMMA 过程	将名称表转换为用逗号定界的列表。
VALIDATE 过程	使无效数据库对象变为有效。

下表列示 DBMS_UTILITY 模块中可用的内置变量和类型。

表 28. DBMS_UTILITY 公用变量

公用变量	数据类型	描述
lname_array	TABLE	用于长名称列表。
uncl_array	TABLE	用于用户和名称列表。

LNAME_ARRAY 用于存储包括标准名称在内的长名称列表。

```
ALTER MODULE SYSIBMADM.DBMS_UTILITY PUBLISH TYPE LNAME_ARRAY AS VARCHAR(4000) ARRAY[];
```

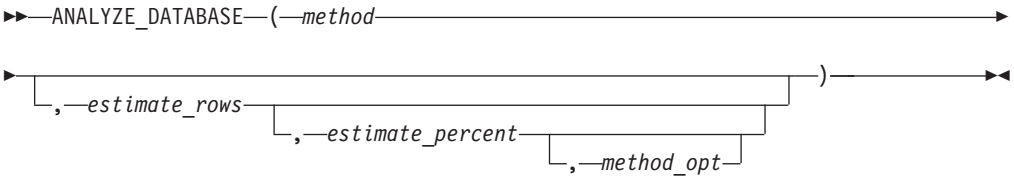
UNCL_ARRAY 用于存储用户和名称列表。

```
ALTER MODULE SYSIBMADM.DBMS_UTILITY PUBLISH TYPE UNCL_ARRAY  AS VARCHAR(227)  ARRAY[];
```

ANALYZE_DATABASE 过程 - 收集有关表、集群和索引的统计信息

ANALYZE_DATABASE 过程能够收集有关数据库中的表、集群和索引的统计信息。

语法



参数

method
类型为 VARCHAR(128) 的输入参数，用于指定要执行的分析功能的类型。有效值为:

- ESTIMATE - 收集基于 *estimate_rows* 中的指定行数或 *estimate_percent* 中的行百分比的估计统计信息;
- COMPUTE - 计算确切统计信息；或者
- DELETE - 从数据字典中删除统计信息。

estimate_rows
类型为 INTEGER 的可选输入参数，用于指定估计统计信息时所基于的行数。如果 *method* 为 ESTIMATE，那么必须指定 *estimate_rows* 或 *estimate_percent* 中的一个。缺省值为 NULL。

estimate_percent
类型为 INTEGER 的可选输入参数，用于指定估计统计信息时所基于的行百分比。如果 *method* 为 ESTIMATE，那么必须指定 *estimate_rows* 或 *estimate_percent* 中的一个。缺省值为 NULL。

method_opt
类型为 VARCHAR(1024) 的可选输入参数，用于指定要分析的对象类型。下列关键字的任意组合都有效:

- [FOR TABLE]
- [FOR ALL [INDEXED] COLUMNS] [SIZE n]
- [FOR ALL INDEXES]

缺省值为 NULL。

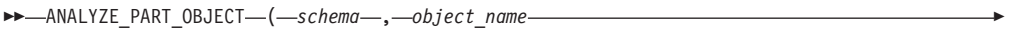
授权

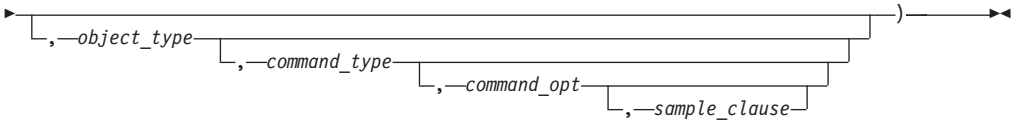
对 DBMS_UTILITY 模块的 EXECUTE 特权。

ANALYZE_PART_OBJECT 过程 - 收集有关分区表或分区索引的统计信息

ANALYZE_PART_OBJECT 过程能够收集有关分区表或索引的统计信息。

语法





参数

schema

类型为 VARCHAR(128) 的输入参数，用于指定要分析其对象的模式的名称。

object_name

类型为 VARCHAR(128) 的输入参数，用于指定要分析的分区对象的名称。

object_type

类型为 CHAR 的可选输入参数，用于指定要分析的对象的类型。有效值为:

- T - 表;
- I - 索引。

缺省值为 T。

command_type

类型为 CHAR 的可选输入参数，用于指定要执行的分析功能的类型。有效值为:

- E - 收集基于 *sample_clause* 子句中的指定行数或行百分比的估计统计信息;
- C - 计算确切统计信息；或者
- V - 验证分区的结构或完整性。

缺省值为 E。

command_opt

类型为 VARCHAR(1024) 的可选输入参数，用于指定计算统计信息时使用的选项。对于 *command_type* E 或 C，此参数可以是下列各项的任意组合:

- [FOR TABLE]
- [FOR ALL COLUMNS]
- [FOR ALL LOCAL INDEXES]

对于 *command_type* V，如果 *object_type* 为 T，那么此参数可为 CASCADE。缺省值为 NULL。

sample_clause

类型为 VARCHAR(128) 的可选输入参数。如果 *command_type* 为 E，那么此参数包含以下子句，该子句用于指定估计时所基于的行数或行百分比。

```
SAMPLE n { ROWS | PERCENT }
```

缺省值为 `SAMPLE 5 PERCENT`。

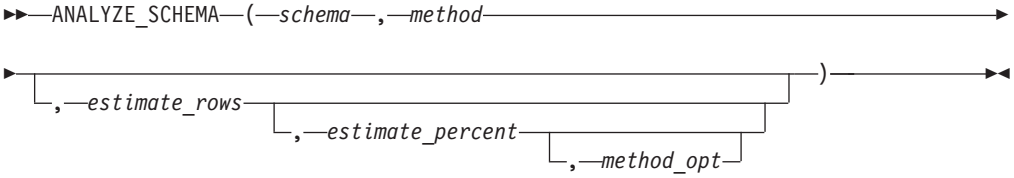
授权

对 DBMS_UTILITY 模块的 EXECUTE 特权。

ANALYZE_SCHEMA 过程 - 收集有关模式表、集群和索引的统计信息

ANALYZE_SCHEMA 过程能够收集有关指定模式中的表、集群和索引的统计信息。

语法



参数

schema
类型为 VARCHAR(128) 的输入参数，用于指定要分析其对象的模式的名称。

method
类型为 VARCHAR(128) 的输入参数，用于指定要执行的分析功能的类型。有效值为:

- ESTIMATE - 收集基于 *estimate_rows* 中的指定行数或 *estimate_percent* 中的行百分比的估计统计信息;
- COMPUTE - 计算确切统计信息；或者
- DELETE - 从数据字典中删除统计信息。

estimate_rows
类型为 INTEGER 的可选输入参数，用于指定估计统计信息时所基于的行数。如果 *method* 为 ESTIMATE，那么必须指定 *estimate_rows* 或 *estimate_percent* 中的一个。缺省值为 NULL。

estimate_percent
类型为 INTEGER 的可选输入参数，用于指定估计统计信息时所基于的行百分比。如果 *method* 为 ESTIMATE，那么必须指定 *estimate_rows* 或 *estimate_percent* 中的一个。缺省值为 NULL。

method_opt
类型为 VARCHAR(1024) 的可选输入参数，用于指定要分析的对象类型。下列关键字的任意组合都有效:

- [FOR TABLE]
- [FOR ALL [INDEXED] COLUMNS] [SIZE n]
- [FOR ALL INDEXES]

缺省值为 NULL。

授权

对 DBMS_UTILITY 模块的 EXECUTE 特权。

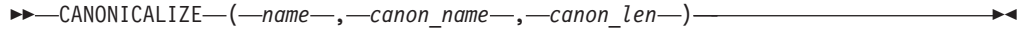
CANONICALIZE 过程 - 使字符串规范化

CANONICALIZE 过程对输入字符串执行各种操作。

CANONICALIZE 过程对输入字符串执行下列操作:

- 如果该字符串未加双引号，那么会验证它是否使用合法标识的字符。如果未使用合法标识的字符，那么会抛出异常。如果字符串加了双引号，那么允许使用所有字符。
- 如果字符串未加双引号并且未包含句点，那么会将所有字母字符转换为大写并消除前导空格和结尾空格。
- 如果字符串加了双引号并且未包含句点，那么会除去双引号。
- 如果字符串包含句点并且字符串的各个部分都未加双引号，那么会将字符串的每个部分转换为大写并将每个部分括在双引号中。
- 如果字符串包含句点并且字符串的某些部分加了双引号，那么返回时加了双引号的部分会保持不变（包括双引号），未加双引号的部分会转换为大写并括在双引号中。

语法



参数

name
类型为 VARCHAR(1024) 的输入参数，用于指定要规范化的字符串。

canon_name
类型为 VARCHAR(1024) 的输出参数，用于返回规范化字符串。

canon_len
类型为 INTEGER 的输入参数，用于指定 *name* 中从第一个字符开始规范化的字节数。

授权

对 DBMS_UTILITY 模块的 EXECUTE 特权。

示例

示例 1: 以下过程对其输入参数应用 CANONICALIZE 过程并显示结果。

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE canonicalize(
  IN p_name VARCHAR(4096),
  IN p_length INTEGER DEFAULT 30)
BEGIN
  DECLARE v_canon VARCHAR(100);

  CALL DBMS_UTILITY.CANONICALIZE(p_name, v_canon, p_length);
  CALL DBMS_OUTPUT.PUT_LINE('Canonicalized name ==>' || v_canon || '<==');
  CALL DBMS_OUTPUT.PUT_LINE('Length: ' || LENGTH(v_canon));
END@

CALL canonicalize('Identifier')@
CALL canonicalize('"Identifier"')@
CALL canonicalize('"_+142%"')@
CALL canonicalize('abc.def.ghi')@
CALL canonicalize('"abc.def.ghi"')@
CALL canonicalize('"abc".def."ghi"')@
CALL canonicalize('"abc.def".ghi')@
```

此示例生成以下输出:

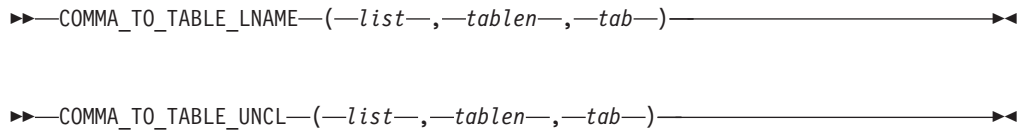
```
SET SERVEROUTPUT ON
DB20000I  SET SERVEROUTPUT 命令成功完成。

CREATE OR REPLACE PROCEDURE canonicalize(
  IN p_name VARCHAR(4096),
  IN p_length INTEGER DEFAULT 30)
BEGIN
  DECLARE v_canon VARCHAR(100);

  CALL DBMS_UTILITY.CANONICALIZE(p_name, v_canon, p_length);
  CALL DBMS_OUTPUT.PUT_LINE('Canonicalized name ==>' || v_canon || '<==');
  CALL DBMS_OUTPUT.PUT_LINE('Length: ' || LENGTH(v_canon));
END
DB20000I  The SQL command completed successfully.

CALL canonicalize('Identifier')

  Return Status = 0

Canonicalized name ==>IDENTIFIER<==
Length: 10

CALL canonicalize('"Identifier"')

  Return Status = 0

Canonicalized name ==>Identifier<==
Length: 10

CALL canonicalize('"_+142%"')

  Return Status = 0

Canonicalized name ==>_+142%<==
Length: 6

CALL canonicalize('abc.def.ghi')

  Return Status = 0

Canonicalized name ==>"ABC"."DEF"."GHI"<==
Length: 17

CALL canonicalize('"abc.def.ghi"')

  Return Status = 0

Canonicalized name ==>abc.def.ghi<==
Length: 11

CALL canonicalize('"abc".def."ghi"')

  Return Status = 0

Canonicalized name ==>"abc"."DEF"."ghi"<==
Length: 17

CALL canonicalize('"abc.def".ghi')

  Return Status = 0

Canonicalized name ==>"abc.def"."GHI"<==
Length: 15
```

COMMA_TO_TABLE 过程 - 将用逗号定界的名称列表转换为名称表

COMMA_TO_TABLE 过程 - 将用逗号定界的名称列表转换为名称数组。列表中的每个条目都将成为数组中的一个元素。

注: 这些名称必须格式化为有效标识。

语法



参数

列表
类型为 VARCHAR(32672) 的输入参数，用于指定用逗号定界的名称列表。

tablen
类型为 INTEGER 的输出参数，用于指定 *tab* 中的条目数。

tab
类型为 LNAME_ARRAY 或 UNCL_ARRAY 的输出参数，其中包含 *list* 中的各个名称的表。请参阅 LNAME_ARRAY 或 UNCL_ARRAY 以获取 *tab* 的描述。

授权

对 DBMS_UTILITY 模块的 EXECUTE 特权。

示例

示例 1: 以下过程使用 COMMA_TO_TABLE_LNAME 过程将名称列表转换为表。然后会显示表的条目。

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE comma_to_table(
  IN p_list VARCHAR(4096))
BEGIN
  DECLARE r_lname DBMS_UTILITY.LNAME_ARRAY;
  DECLARE v_length INTEGER;
  CALL DBMS_UTILITY.COMMA_TO_TABLE_LNAME(p_list, v_length, r_lname);
  BEGIN
    DECLARE i INTEGER DEFAULT 1;
    DECLARE loop_limit INTEGER;

    SET loop_limit = v_length;
    WHILE i <= loop_limit DO
      CALL DBMS_OUTPUT.PUT_LINE(r_lname[i]);
      SET i = i + 1;
    END WHILE;
  END;
END@

CALL comma_to_table('sample_schema.dept,sample_schema.emp,sample_schema.jobhist')@
```

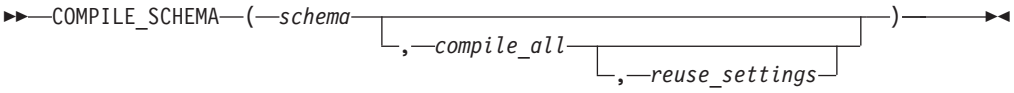
此示例生成以下输出:

```
SET SERVEROUTPUT ON
DB20000I  SET SERVEROUTPUT 命令成功完成。

CREATE OR REPLACE PROCEDURE comma_to_table(
  IN p_list VARCHAR(4096))
BEGIN
  DECLARE r_lname DBMS_UTILITY.LNAME_ARRAY;
  DECLARE v_length INTEGER;
  CALL DBMS_UTILITY.COMMA_TO_TABLE_LNAME(p_list, v_length, r_lname);
  BEGIN
    DECLARE i INTEGER DEFAULT 1;
    DECLARE loop_limit INTEGER;

    SET loop_limit = v_length;
    WHILE i <= loop_limit DO
      CALL DBMS_OUTPUT.PUT_LINE(r_lname[i]);
      SET i = i + 1;
    END WHILE;
  END;
END
DB20000I  The SQL command completed successfully.

CALL comma_to_table('sample_schema.dept,sample_schema.emp,sample_schema.jobhist')

  Return Status = 0

sample_schema.dept
sample_schema.emp
sample_schema.jobhist
```

COMPILE_SCHEMA 过程 - 编译模式中的所有函数、过程、触发器和包

COMPILE_SCHEMA 过程能够重新编译模式中的所有函数、过程、触发器和包。

语法



参数

schema
类型为 VARCHAR(128) 的输入参数，用于指定要重新编译其中的程序的模式。

compile_all
类型为 BOOLEAN 的可选输入参数，必须设置为 false，意味着该过程仅重新编译当前处于无效状态的程序。

reuse_settings
类型为 BOOLEAN 的可选输入参数，必须设置为 false，意味着该过程使用当前会话设置。

授权

对 DBMS_UTILITY 模块的 EXECUTE 特权。

DB_VERSION 过程 - 检索数据库版本

DB_VERSION 过程返回数据库的版本号。

语法

参数

版本
类型为 VARCHAR(1024) 的输出参数，用于返回数据库版本号。

兼容性
类型为 VARCHAR(1024) 的输出参数，用于返回数据库的兼容性设置。

授权

对 DBMS_UTILITY 模块的 EXECUTE 特权。

示例

示例 1: 以下匿名块显示数据库版本信息。

```
SET SERVEROUTPUT ON@

BEGIN
  DECLARE v_version VARCHAR(80);
  DECLARE v_compat VARCHAR(80);

  CALL DBMS_UTILITY.DB_VERSION(v_version, v_compat);
  CALL DBMS_OUTPUT.PUT_LINE('Version: ' || v_version);
  CALL DBMS_OUTPUT.PUT_LINE('Compatibility: ' || v_compat);
END@
```

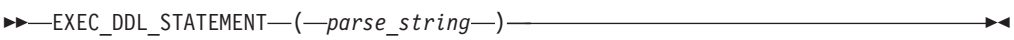
此示例生成以下输出:

```
SET SERVEROUTPUT ON
DB20000I  SET SERVEROUTPUT 命令成功完成。

BEGIN
  DECLARE v_version VARCHAR(80);
  DECLARE v_compat VARCHAR(80);

  CALL DBMS_UTILITY.DB_VERSION(v_version, v_compat);
  CALL DBMS_OUTPUT.PUT_LINE('Version: ' || v_version);
  CALL DBMS_OUTPUT.PUT_LINE('Compatibility: ' || v_compat);
END
DB20000I  The SQL command completed successfully.

Version: DB2 v9.7.0.0
Compatibility: DB2 v9.7.0.0
```

EXEC_DDL_STATEMENT 过程 - 运行 DDL 语句

EXEC_DDL_STATEMENT 过程能够执行 DDL 命令。

语法



参数

parse_string
类型为 VARCHAR(1024) 的输入参数，用于指定要执行的 DDL 命令。

授权

对 DBMS_UTILITY 模块的 EXECUTE 特权。

示例

示例 1: 以下匿名块创建表 job。

```
BEGIN
  CALL DBMS_UTILITY.EXEC_DDL_STATEMENT(
    'CREATE TABLE job (' ||
    'jobno DECIMAL(3),' ||
    'jname VARCHAR(9))' );
END@
```

GET_CPU_TIME 函数 - 检索当前 CPU 时间

GET_CPU_TIME 函数返回从某个任意时间点开始计算的 CPU 时间（以百分之一秒计）。

语法

授权

对 DBMS_UTILITY 模块的 EXECUTE 特权。

示例

示例 1: 以下 SELECT 命令检索当前 CPU 时间。

```
SELECT DBMS_UTILITY.GET_CPU_TIME FROM DUAL;

get_cpu_time
-------------
          603
```

示例 2: 通过获得两个 CPU 时间值之间的差别计算耗用时间。

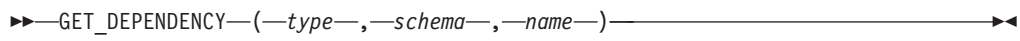
```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE proc1()

  BEGIN
     DECLARE cpuTime1 BIGINT;
     DECLARE cpuTime2 BIGINT;
     DECLARE cpuTimeDelta BIGINT;
     DECLARE i INTEGER;
```

```
     SET cpuTime1 = DBMS_UTILITY.GET_CPU_TIME();

     SET i = 0;
     loop1: LOOP
IF i > 10000 THEN
           LEAVE loop1;
END IF;
     SET i = i + 1;
END LOOP;
     SET cpuTime2 = DBMS_UTILITY.GET_CPU_TIME();

     SET cpuTimeDelta = cpuTime2 - cpuTime1;

     CALL DBMS_OUTPUT.PUT_LINE( 'cpuTimeDelta = ' || cpuTimeDelta );
     END
     @

  CALL proc1@
```

GET_DEPENDENCY 过程 - 列示依赖于给定对象的对象

GET_DEPENDENCY 过程能够列示依赖于给定对象的所有对象。

语法



参数

类型
类型为 VARCHAR(128) 的输入参数，用于指定类型为 *name* 的对象。有效值为 FUNCTION、INDEX、LOB、PACKAGE、PACKAGE BODY、PROCEDURE、SEQUENCE、TABLE、TRIGGER 和 VIEW。

schema
类型为 VARCHAR(128) 的输入参数，用于指定 *name* 中存在的模式的名称。

name
类型为 VARCHAR(128) 的输入参数，用于指定要获取其依赖性的对象的名称。

授权

对 DBMS_UTILITY 模块的 EXECUTE 特权。

示例

示例 1: 以下匿名块查找表 T1 以及函数 FUNC1 的依赖性。

```
SET SERVEROUTPUT ON@

CREATE TABLE SCHEMA1.T1 (C1 INTEGER)@

CREATE OR REPLACE FUNCTION SCHEMA2.FUNC1( parm1 INTEGER )
SPECIFIC FUNC1
RETURNS INTEGER
BEGIN
  RETURN parm1;
END@
```

```
CREATE OR REPLACE FUNCTION SCHEMA3.FUNC2()
SPECIFIC FUNC2
RETURNS INTEGER
BEGIN
  DECLARE retVal INTEGER;
  SELECT SCHEMA2.FUNC1(1) INTO retVal FROM SCHEMA1.T1;
END@

CALL DBMS_UTILITY.GET_DEPENDENCY('FUNCTION', 'SCHEMA2', 'FUNC1')@
CALL DBMS_UTILITY.GET_DEPENDENCY('TABLE', 'SCHEMA1', 'T1')@
```

此示例生成以下输出:

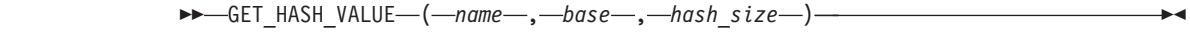
```
SET SERVEROUTPUT ON
DB20000I  SET SERVEROUTPUT 命令成功完成。

CREATE TABLE SCHEMA1.T1 (C1 INTEGER)
DB20000I  The SQL command completed successfully.

CREATE OR REPLACE FUNCTION SCHEMA2.FUNC1( parm1 INTEGER )
SPECIFIC FUNC1
RETURNS INTEGER
BEGIN
  RETURN parm1;
END
DB20000I  The SQL command completed successfully.

CREATE OR REPLACE FUNCTION SCHEMA3.FUNC2()
SPECIFIC FUNC2
RETURNS INTEGER
BEGIN
  DECLARE retVal INTEGER;
  SELECT SCHEMA2.FUNC1(1) INTO retVal FROM SCHEMA1.T1;
END
DB20000I  The SQL command completed successfully.

CALL DBMS_UTILITY.GET_DEPENDENCY('FUNCTION', 'SCHEMA2', 'FUNC1')

  Return Status = 0

DEPENDENCIES ON SCHEMA2.FUNC1
------------------------------------------------------------------
*FUNCTION SCHEMA2.FUNC1()
*   FUNCTION SCHEMA3 .FUNC2()

CALL DBMS_UTILITY.GET_DEPENDENCY('TABLE', 'SCHEMA1', 'T1')

  Return Status = 0

DEPENDENCIES ON SCHEMA1.T1
------------------------------------------------------------------
*TABLE SCHEMA1.T1()
*   FUNCTION SCHEMA3 .FUNC2()
```

GET_HASH_VALUE 函数 - 计算给定字符串的散列值

GET_HASH_VALUE 函数能够计算给定字符串的散列值。

该函数返回类型为 INTEGER 的已生成散列值，该值依赖于平台。

语法



参数

name
类型为 VARCHAR(32672) 的输入参数，用于指定要对其计算散列值的字符串。

base
类型为 INTEGER 的输入参数，用于指定要生成散列值的起始值。

hash_size
类型为 INTEGER 的输入参数，用于指定期望散列表的散列值数目。

授权

对 DBMS_UTILITY 模块的 EXECUTE 特权。

示例

示例 1: 以下示例返回两个字符串的散列值: 散列值的起始值为 100，最多 1024 个相异值。

```
SELECT DBMS_UTILITY.GET_HASH_VALUE('Peter',100,1024) AS HASH_VALUE FROM SYSIBM.SYSDUMMY1@
SELECT DBMS_UTILITY.GET_HASH_VALUE('Mary',100,1024) AS HASH_VALUE FROM SYSIBM.SYSDUMMY1@
```

此示例生成以下输出:

```
SELECT DBMS_UTILITY.GET_HASH_VALUE('Peter',100,1024) AS HASH_VALUE FROM SYSIBM.SYSDUMMY1

HASH_VALUE
--------------------
                 343

  1 record(s) selected.


SELECT DBMS_UTILITY.GET_HASH_VALUE('Mary',100,1024) AS HASH_VALUE FROM SYSIBM.SYSDUMMY1

HASH_VALUE
--------------------
                 760

  1 record(s) selected.
```

GET_TIME 函数 - 返回当前时间

GET_TIME 函数能够返回当前时间（以百分之一秒计）。

语法

授权

对 DBMS_UTILITY 模块的 EXECUTE 特权。

示例

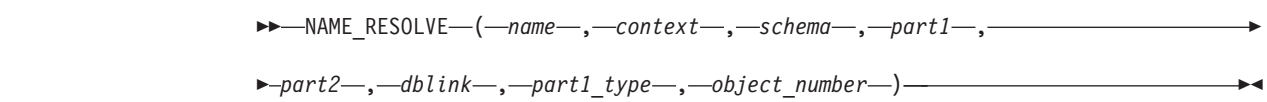
示例 1: 以下示例显示对 GET_TIME 函数的调用。

```
SELECT DBMS_UTILITY.GET_TIME FROM DUAL;

 get_time
---------
  1555860

SELECT DBMS_UTILITY.GET_TIME FROM DUAL;

 get_time
---------
  1556037
```

NAME_RESOLVE 过程 - 获取数据库对象的模式和其他成员资格信息

NAME_RESOLVE 过程能够获取数据库对象的模式和其他成员资格信息。同义词将解析为其基本对象。

语法



参数

name

类型为 VARCHAR(1024) 的输入参数，用于指定要解析的数据库对象的名称。可按以下格式指定:

```
[[ a.]b.]c[@dblink ]
```

context

类型为 INTEGER 的输入参数。设置为下列值:

- 1 - 解析函数、过程或模块名称;
- 2 - 解析表、视图、序列或同义词名称; 或者
- 3 - 解析触发器名称。

schema

类型为 VARCHAR(128) 的输出参数，用于指定包含 *name* 所指定对象的模式的名称。

part1

类型为 VARCHAR(128) 的输出参数，用于指定所解析表、视图、序列、触发器或模块的名称。

part2

类型为 VARCHAR(128) 的输出参数，用于指定所解析函数或过程（包括某个模块中的函数和过程）的名称。

dblink

类型为 VARCHAR(128) 的输出参数，用于指定数据库链接的名称（如果在 *name* 中指定了 `@dblink`）。

part1_type

类型为 INTEGER 的输出参数。返回下列值:

- 2 - 所解析对象为表;

- 4 - 所解析对象为视图;
- 6 - 所解析对象为序列;
- 7 - 所解析对象为存储过程;
- 8 - 所解析对象为存储函数;
- 9 - 所解析对象为模块或模块中的函数或过程; 或者
- 12 - 所解析对象为触发器。

object_number
类型为 INTEGER 的输出参数，用于指定所解析数据库对象的对象标识。

授权

对 DBMS_UTILITY 模块的 EXECUTE 特权。

示例

示例 1: 以下存储过程用于显示 NAME_RESOLVE 过程对各种数据库对象返回的值。

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE name_resolve(
  IN p_name VARCHAR(4096),
  IN p_context DECFLOAT )
BEGIN
  DECLARE v_schema VARCHAR(30);
  DECLARE v_part1 VARCHAR(30);
  DECLARE v_part2 VARCHAR(30);
  DECLARE v_dblink VARCHAR(30);
  DECLARE v_part1_type DECFLOAT;
  DECLARE v_objectid DECFLOAT;

  CALL DBMS_UTILITY.NAME_RESOLVE(p_name, p_context, v_schema, v_part1, v_part2,
    v_dblink, v_part1_type, v_objectid);
  CALL DBMS_OUTPUT.PUT_LINE('name      : ' || p_name);
  CALL DBMS_OUTPUT.PUT_LINE('context   : ' || p_context);
  CALL DBMS_OUTPUT.PUT_LINE('schema    : ' || v_schema);
  IF v_part1 IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('part1     : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('part1     : ' || v_part1);
  END IF;
  IF v_part2 IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('part2     : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('part2     : ' || v_part2);
  END IF;
  IF v_dblink IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('dblink    : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('dblink    : ' || v_dblink);
  END IF;
  CALL DBMS_OUTPUT.PUT_LINE('part1 type: ' || v_part1_type);
  CALL DBMS_OUTPUT.PUT_LINE('object id : ' || v_objectid);
END@

DROP TABLE S1.T1@
CREATE TABLE S1.T1 (C1 INT)@

CREATE OR REPLACE PROCEDURE S2.PROC1
BEGIN
END@
```

```
CREATE OR REPLACE MODULE S3.M1@
ALTER MODULE S3.M1 PUBLISH FUNCTION F1() RETURNS BOOLEAN
BEGIN
  RETURN TRUE;
END@

CALL NAME_RESOLVE( 'S1.T1', 2 )@
CALL NAME_RESOLVE( 'S2.PROC1', 2 )@
CALL NAME_RESOLVE( 'S2.PROC1', 1 )@
CALL NAME_RESOLVE( 'PROC1', 1 )@
CALL NAME_RESOLVE( 'M1', 1 )@
CALL NAME_RESOLVE( 'S3.M1.F1', 1 )@
```

此示例生成以下输出:

```
SET SERVEROUTPUT ON
DB20000I  SET SERVEROUTPUT 命令成功完成。

CREATE OR REPLACE PROCEDURE name_resolve(
  IN p_name VARCHAR(4096),
  IN p_context DECFLOAT )
BEGIN
  DECLARE v_schema VARCHAR(30);
  DECLARE v_part1 VARCHAR(30);
  DECLARE v_part2 VARCHAR(30);
  DECLARE v_dblink VARCHAR(30);
  DECLARE v_part1_type DECFLOAT;
  DECLARE v_objectid DECFLOAT;

  CALL DBMS_UTILITY.NAME_RESOLVE(p_name, p_context, v_schema, v_part1, v_part2,
    v_dblink, v_part1_type, v_objectid);
  CALL DBMS_OUTPUT.PUT_LINE('name      : ' || p_name);
  CALL DBMS_OUTPUT.PUT_LINE('context   : ' || p_context);
  CALL DBMS_OUTPUT.PUT_LINE('schema    : ' || v_schema);
  IF v_part1 IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('part1     : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('part1     : ' || v_part1);
  END IF;
  IF v_part2 IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('part2     : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('part2     : ' || v_part2);
  END IF;
  IF v_dblink IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('dblink    : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('dblink    : ' || v_dblink);
  END IF;
  CALL DBMS_OUTPUT.PUT_LINE('part1 type: ' || v_part1_type);
  CALL DBMS_OUTPUT.PUT_LINE('object id : ' || v_objectid);
END
DB20000I  The SQL command completed successfully.

DROP TABLE S1.T1
DB20000I  The SQL command completed successfully.

CREATE TABLE S1.T1 (C1 INT)
DB20000I  The SQL command completed successfully.

CREATE OR REPLACE PROCEDURE S2.PROC1
BEGIN
END
DB20000I  The SQL command completed successfully.

CREATE OR REPLACE MODULE S3.M1
DB20000I  The SQL command completed successfully.
```

```
ALTER MODULE S3.M1 PUBLISH FUNCTION F1() RETURNS BOOLEAN
BEGIN
  RETURN TRUE;
END
DB20000I  The SQL command completed successfully.

CALL NAME_RESOLVE( 'S1.T1', 2 )

  Return Status = 0

name      : S1.T1
context   : 2
schema    : S1
part1     : T1
part2     : NULL
dblink    : NULL
part1 type: 2
object id : 8

CALL NAME_RESOLVE( 'S2.PROC1', 2 )
SQL0204N  "S2.PROC1" is an undefined name.  SQLSTATE=42704

CALL NAME_RESOLVE( 'S2.PROC1', 1 )

  Return Status = 0

name      : S2.PROC1
context   : 1
schema    : S2
part1     : PROC1
part2     : NULL
dblink    : NULL
part1 type: 7
object id : 66611

CALL NAME_RESOLVE( 'PROC1', 1 )

  Return Status = 0

name      : PROC1
context   : 1
schema    : S2
part1     : NULL
part2     : PROC1
dblink    : NULL
part1 type: 7
object id : 66611

CALL NAME_RESOLVE( 'M1', 1 )

  Return Status = 0

name      : M1
context   : 1
schema    : S3
part1     : NULL
part2     : M1
dblink    : NULL
part1 type: 9
object id : 16

CALL NAME_RESOLVE( 'S3.M1.F1', 1 )

  Return Status = 0

name      : S3.M1.F1
```

```
context   : 1
schema    : S3
part1     : M1
part2     : F1
dblink    : NULL
part1 type: 9
object id : 16
```

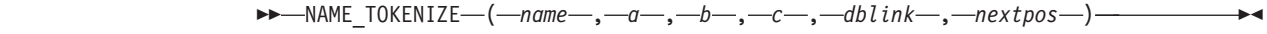
示例 2: 解析数据库链接访问的表。注意，NAME_RESOLVE 未检查远程数据库上的数据库对象的有效性。它只是回传了 *name* 参数中指定的部分。

```
BEGIN
    name_resolve('sample_schema.emp@sample_schema_link',2);
END;

name      : sample_schema.emp@sample_schema_link
context   : 2
schema    : SAMPLE_SCHEMA
part1     : EMP
part2     :
dblink    : SAMPLE_SCHEMA_LINK
part1 type: 0
object id : 0
```

NAME_TOKENIZE 过程 - 将给定名称解析为若干组成部分

NAME_TOKENIZE 过程将名称解析为若干部分。没有双引号的名称将转换为大写，并且会除去这些名称所带的双引号。

语法



参数

name
类型为 VARCHAR(1024) 的输入参数，用于指定包含以下格式的名称的字符串:

```
a[.b[.c]][@dblink ]
```

a 类型为 VARCHAR(128) 的输出参数，用于返回最左边的部分。

b 类型为 VARCHAR(128) 的输出参数，用于返回第二个部分（如果存在）。

c 类型为 VARCHAR(128) 的输出参数，用于返回第三个部分（如果存在）。

dblink
类型为 VARCHAR(32672) 的输出参数，用于返回数据库链接名称。

nextpos
类型为 INTEGER 的输出参数，用于指定 *name* 中解析的最后一个字符的位置。

授权

对 DBMS_UTILITY 模块的 EXECUTE 特权。

示例

示例 1: 以下存储过程用于显示 NAME_TOKENIZE 过程对各种名称返回的值。

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE name_tokenize(
  IN p_name VARCHAR(100) )
BEGIN
  DECLARE v_a VARCHAR(30);
  DECLARE v_b VARCHAR(30);
  DECLARE v_c VARCHAR(30);
  DECLARE v_dblink VARCHAR(30);
  DECLARE v_nextpos INTEGER;

  CALL DBMS_UTILITY.NAME_TOKENIZE(p_name, v_a, v_b, v_c, v_dblink, v_nextpos);
  CALL DBMS_OUTPUT.PUT_LINE('name      : ' || p_name);
  IF v_a IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('a      : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('a      : ' || v_a);
  END IF;
  IF v_b IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('b      : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('b      : ' || v_b);
  END IF;
  IF v_c IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('c      : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('c      : ' || v_c);
  END IF;
  IF v_dblink IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('dblink    : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('dblink    : ' || v_dblink);
  END IF;
  IF v_nextpos IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('nextpos: NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('nextpos: ' || v_nextpos);
  END IF;
END@

CALL name_tokenize( 'b' )@
CALL name_tokenize( 'a.b' )@
CALL name_tokenize( '"a".b.c' )@
CALL name_tokenize( 'a.b.c@d' )@
CALL name_tokenize( 'a.b."c"@"d"' )@
```

此示例生成以下输出:

```
SET SERVEROUTPUT ON
DB20000I  SET SERVEROUTPUT 命令成功完成。

CREATE OR REPLACE PROCEDURE name_tokenize(
  IN p_name VARCHAR(100) )
BEGIN
  DECLARE v_a VARCHAR(30);
  DECLARE v_b VARCHAR(30);
  DECLARE v_c VARCHAR(30);
  DECLARE v_dblink VARCHAR(30);
  DECLARE v_nextpos INTEGER;

  CALL DBMS_UTILITY.NAME_TOKENIZE(p_name, v_a, v_b, v_c, v_dblink, v_nextpos);
  CALL DBMS_OUTPUT.PUT_LINE('name      : ' || p_name);
  IF v_a IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('a      : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('a      : ' || v_a);
  END IF;
```

```
  IF v_b IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('b      : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('b      : ' || v_b);
  END IF;
  IF v_c IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('c      : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('c      : ' || v_c);
  END IF;
  IF v_dblink IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('dblink    : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('dblink    : ' || v_dblink);
  END IF;
  IF v_nextpos IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('nextpos: NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('nextpos: ' || v_nextpos);
  END IF;
END
DB20000I  The SQL command completed successfully.

CALL name_tokenize( 'b' )

  Return Status = 0

name   : b
a      : B
b      : NULL
c      : NULL
dblink    : NULL
nextpos: 1

CALL name_tokenize( 'a.b' )

  Return Status = 0

name   : a.b
a      : A
b      : B
c      : NULL
dblink    : NULL
nextpos: 3
CALL name_tokenize( '"a".b.c' )

  Return Status = 0

name   : "a".b.c
a      : a
b      : B
c      : C
dblink    : NULL
nextpos: 7

CALL name_tokenize( 'a.b.c@d' )

  Return Status = 0

name   : a.b.c@d
a      : A
b      : B
c      : C
dblink : D
nextpos: 7

CALL name_tokenize( 'a.b."c"@"d"' )
```

```
  Return Status = 0

name   : a.b."c"@"d"
a      : A
b      : B
c      : c
dblink : d
nextpos: 11
```

TABLE_TO_COMMA 过程 - 将名称表转换为用逗号定界的名称列表

TABLE_TO_COMMA 过程将名称数组转换为用逗号定界的名称列表。每个数组元素变为一个列表条目。

注: 这些名称必须格式化为有效标识。

语法

```
►►—TABLE_TO_COMMA_LNAME—(—tab—,—tablen—,—list—)———————————————————►◄

►►—TABLE_TO_COMMA_UNCL—(—tab—,—tablen—,—list—)———————————————————►◄
```

参数

tab
类型为 LNAME_ARRAY 或 UNCL_ARRAY 的输入参数，用于指定包含名称的数组。请参阅 LNAME_ARRAY 或 UNCL_ARRAY 以获取 *tab* 的描述。

tablen
类型为 INTEGER 的输出参数，用于返回 *list* 中的条目数。

列表
类型为 VARCHAR(32672) 的输出参数，用于从 *tab* 返回用逗号定界的名称列表。

授权

对 DBMS_UTILITY 模块的 EXECUTE 特权。

示例

示例 1: 以下示例先使用 COMMA_TO_TABLE_LNAME 过程将用逗号定界的列表转换为表。然后 TABLE_TO_COMMA_LNAME 过程会将该表转换为转换回所显示的用逗号定界的列表。

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE table_to_comma(
  IN p_list VARCHAR(100))
BEGIN
  DECLARE r_lname DBMS_UTILITY.LNAME_ARRAY;
  DECLARE v_length INTEGER;
  DECLARE v_listlen INTEGER;
  DECLARE v_list VARCHAR(80);

  CALL DBMS_UTILITY.COMMA_TO_TABLE_LNAME(p_list, v_length, r_lname);
  CALL DBMS_OUTPUT.PUT_LINE('Table Entries');
```

```
  CALL DBMS_OUTPUT.PUT_LINE('-------------');
  BEGIN
    DECLARE i INTEGER DEFAULT 1;
    DECLARE LOOP_LIMIT INTEGER;
    SET LOOP_LIMIT = v_length;

    WHILE i <= LOOP_LIMIT DO
      CALL DBMS_OUTPUT.PUT_LINE(r_lname(i));
      SET i = i + 1;
    END WHILE;
  END;
  CALL DBMS_OUTPUT.PUT_LINE('-------------');
  CALL DBMS_UTILITY.TABLE_TO_COMMA_LNAME(r_lname, v_listlen, v_list);
  CALL DBMS_OUTPUT.PUT_LINE('Comma-Delimited List: ' || v_list);
END@

CALL table_to_comma('sample_schema.dept,sample_schema.emp,sample_schema.jobhist')@
```

此示例生成以下输出:

```
SET SERVEROUTPUT ON
DB20000I  SET SERVEROUTPUT 命令成功完成。

CREATE OR REPLACE PROCEDURE table_to_comma(
  IN p_list VARCHAR(100))
BEGIN
  DECLARE r_lname DBMS_UTILITY.LNAME_ARRAY;
  DECLARE v_length INTEGER;
  DECLARE v_listlen INTEGER;
  DECLARE v_list VARCHAR(80);

  CALL DBMS_UTILITY.COMMA_TO_TABLE_LNAME(p_list, v_length, r_lname);
  CALL DBMS_OUTPUT.PUT_LINE('Table Entries');
  CALL DBMS_OUTPUT.PUT_LINE('-------------');
  BEGIN
    DECLARE i INTEGER DEFAULT 1;
    DECLARE LOOP_LIMIT INTEGER;
    SET LOOP_LIMIT = v_length;

    WHILE i <= LOOP_LIMIT DO
      CALL DBMS_OUTPUT.PUT_LINE(r_lname(i));
      SET i = i + 1;
    END WHILE;
  END;
  CALL DBMS_OUTPUT.PUT_LINE('-------------');
  CALL DBMS_UTILITY.TABLE_TO_COMMA_LNAME(r_lname, v_listlen, v_list);
  CALL DBMS_OUTPUT.PUT_LINE('Comma-Delimited List: ' || v_list);
END
DB20000I  The SQL command completed successfully.

CALL table_to_comma('sample_schema.dept,sample_schema.emp,sample_schema.jobhist')

  Return Status = 0

Table Entries
-------------
sample_schema.dept
sample_schema.emp
sample_schema.jobhist
-------------
Comma-Delimited List: sample_schema.dept,sample_schema.emp,sample_schema.jobhist
```

VALIDATE 过程 - 将无效例程更改为有效例程

VALIDATE 过程能够将无效例程的状态更改为有效。

语法

参数

object_id
类型为 INTEGER 的输入参数，用于指定要更改为有效状态的例程的标识。SYSCAT.ROUTINES 视图的 ROUTINEID 列包含所有例程标识。

授权

对 DBMS_UTILITY 模块的 EXECUTE 特权。

第 30 章 MONREPORT 模块

MONREPORT 模块提供用于检索各种监视数据和生成文本报告的一组过程。

此模块的模式为 SYSIBMADM。

MONREPORT 模块包括以下内置例程。

表 29. MONREPORT 模块中可用的内置例程

例程名称	描述
CONNECTION 过程	“连接”报告显示每个连接的监视器数据。
CURRENTAPPS 过程	“当前应用程序”报告显示每个连接的工作单元、代理程序和活动的当前瞬时处理状态。报告状态以连接总计状态信息开头，后跟对应每个连接的详细信息部分。
CURRENTSQL 过程	“当前 SQL”报告列示按各种度量计算的当前运行最频繁的活动。
DBSUMMARY 过程	“摘要”报告包含整个数据库以及每个连接、工作负载、服务类和数据库成员的关键性能指标的深入监视数据。
LOCKWAIT 过程	“锁定等待”报告包含有关当前正在进行的每个锁定等待的信息。详细信息包括锁定持有者和请求者详细信息以及所持有锁定和所请求锁定的特征。
PKGCACHE 过程	“程序包高速缓存”报告列示按各种度量计算的程序包高速缓存中累积的最常用语句。

用法说明

监视元素名称以大写显示（如 TOTAL_CPU_TIME）。要了解有关监视元素的更多信息，请在 *DB2 信息中心*中搜索监视器名称。

对于带有 *monitoring_interval* 输入的报告，报告中的负值不准确。这可能会在源数据计数器滚动期间发生。要确定准确的值，请在滚动完成后重新运行该报告。

注: 报告是使用模块中的 SQL 过程实现的，并且因此会受到程序包高速缓存配置的影响。如果发现运行报告时性能不佳，请检查程序包高速缓存配置以确保它对于您的工作负载而言是够用的。有关进一步信息，请参阅“pckcachesz - 程序包高速缓存大小配置参数”。

以下示例演示调用 MONREPORT 例程的各种方法。这些示例显示 MONREPORT.CONNECTION(*monitoring_interval*, *application_handle*) 过程。可通过以下方式处理不想对其输入值的可选参数:

- 可始终指定 NULL 或 DEFAULT。
- 对于字符输入，可指定空字符串（“ ”）。
- 如果是最后一个参数，那么可省略。

要生成包括对应每个连接的部分的报告（使用缺省监视时间间隔 10 秒），请对 MONREPORT.CONNECTION 过程执行以下调用:

```
call monreport.connection()
```

要生成仅包括与应用程序句柄为 32 的连接对应的部分的报告（使用缺省监视时间间隔 10 秒），可对 MONREPORT.CONNECTION 过程执行以下调用之一:

```
call monreport.connection(DEFAULT, 32)
```

```
call monreport.connection(10, 32)
```

要生成包括对应每个连接的部分的报告（使用监视时间间隔 60 秒），可对 MONREPORT.CONNECTION 过程执行以下调用之一:

```
call monreport.connection(60)
```

```
call monreport.connection(60, null)
```

缺省情况下，此模块中的报告以英语生成。要更改用于生成报告的语言，请更改 CURRENT LOCALE LC_MESSAGES 专用寄存器。例如，要以法语生成 CONNECTION 报告，请发出以下命令:

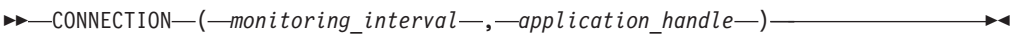
```
SET CURRENT LOCALE LC_MESSAGES = 'CLDR 1.5:fr_FR'
CALL MONREPORT.CONNECTION
```

确保数据库代码页支持生成报告时所用的语言。如果数据库代码页为 Unicode，那么可使用任何语言生成这些报告。

CONNECTION 过程 - 生成连接度量值的报告

CONNECTION 过程收集每个连接的监视器数据并产生文本格式的报告。

语法



参数

monitoring_interval
类型为 INTEGER 的可选输入参数，用于指定在报告所收集监视数据之前，收集这些数据的持续时间（以秒计）。例如，如果指定监视时间间隔为 30 秒，那么该例程会调用表函数，等待 30 秒并再次调用表函数。然后该例程会计算差别，这些差别反映时间间隔期间的更改。如果未指定 *monitoring_interval* 参数（或者指定了 NULL），那么缺省值为 10。有效输入的范围是整数值 0 到 3600（即，最长 1 小时）。

application_handle
类型为 BIGINT 的可选输入参数，用于指定标识连接的应用程序句柄。如果未指定 *application_handle* 参数（或者指定了 NULL），那么对于每个连接，报告都将包括一节。缺省值为 NULL。

授权

需要以下特权:

- 对 MONREPORT 模块的 EXECUTE 特权

示例

以下示例演示调用 CONNECTION 过程的各种方法。

此示例对所有连接生成一个报告，其中显示的数据对应 30 秒时间间隔:

```
call monreport.connection(30);
```

此示例对应用程序句柄为 34 的连接生成报告。系统将根据源表函数中累积的绝对总数（而不是根据当前时间间隔）显示数据:

```
call monreport.connection(0, 34);
```

下一示例对应用程序句柄为 34 的连接生成报告。显示的数据对应于 10 秒时间间隔。

```
call monreport.connection(DEFAULT, 34);
```

最终示例对所有连接生成缺省报告: 显示的数据对应时间间隔 10 秒:

```
call monreport.connection;
```

以下是缺省过程调用的报告输出的示例（所有连接，10 秒时间间隔）:

```
  Result set 1
  --------------

  TEXT
  ------------------------------------------------------------------------------
  ------------------------------------------------------------------------------
  Monitoring report - connection
  ------------------------------------------------------------------------------
  Database:               SAMPLE
  Generated:              04/06/2010 13:36:52
  Interval monitored:     10
  -- Command options --
  APPLICATION_HANDLE:     All

  ==============================================================================
  Part 1 - Summary of connections

  ------------------------------------------------------------------------------
       APPLICATION  TOTAL_         TOTAL_     ACT_COMPLETED  TOTAL_WAIT  CLIENT_IDLE
  #    _HANDLE      CPU_TIME       ACT_TIME   _TOTAL         _TIME       _WAIT_TIME
  ---  -----------  -------------  ---------  -------------  ----------  -----------
  1    180          0              0          0              0           0
  2    65711        116307         675        1              410         9884
  3    131323       116624         679        1              717         12895

  ==============================================================================
  Part 2 - Details for each connection

  connection #:1
  ------------------------------------------------------------------------------

  --Connection identifiers--
  Application identifiers
    APPLICATION_HANDLE                = 180
    APPLICATION_NAME                  = db2bp
    APPLICATION_ID                    = *N0.jwr.100406173420
  Authorization IDs
    SYSTEM_AUTHID                     = JWR
    SESSION_AUTHID                    = JWR
  Client attributes
    CLIENT_ACCTNG                     =
    CLIENT_USERID                     =
    CLIENT_APPLNAME                   =
```

```
  CLIENT_WRKSTNNAME                =
  CLIENT_PID                       = 29987
  CLIENT_PRDID                     = SQL09081
  CLIENT_PLATFORM                  = LINUXX8664
  CLIENT_PROTOCOL                  = LOCAL
-- Other connection details --
CONNECTION_START_TIME              = 2010-04-06-13.34.20.635181
NUM_LOCKS_HELD                     = 9

Work volume and throughput
-------------------------------------------------------------------------------
                                 Per second             Total
                                 ---------------------  ----------------------
TOTAL_APP_COMMITS                0                      0
ACT_COMPLETED_TOTAL              0                      0
APP_RQSTS_COMPLETED_TOTAL        0                      0

TOTAL_CPU_TIME                   = 0
TOTAL_CPU_TIME per request       = 0

Row processing
  ROWS_READ/ROWS_RETURNED        = 0 (0/0)
  ROWS_MODIFIED                  = 0

Wait times
-------------------------------------------------------------------------------

-- Wait time as a percentage of elapsed time --

                                       %    Wait time/Total time
                                       ---  ----------------------------------
For requests                           0    0/0
For activities                         0    0/0

-- Time waiting for next client request --

CLIENT_IDLE_WAIT_TIME              = 0
CLIENT_IDLE_WAIT_TIME per second   = 0

-- Detailed breakdown of TOTAL_WAIT_TIME --

                             %    Total
                             ---  ------------------------------------------
TOTAL_WAIT_TIME              100  3434

I/O wait time
  POOL_READ_TIME             23   805
  POOL_WRITE_TIME            8    280
  DIRECT_READ_TIME           3    131
  DIRECT_WRITE_TIME          3    104
  LOG_DISK_WAIT_TIME         10   344
LOCK_WAIT_TIME               0    18
AGENT_WAIT_TIME              0    0
Network and FCM
  TCPIP_SEND_WAIT_TIME       0    0
  TCPIP_RECV_WAIT_TIME       0    0
  IPC_SEND_WAIT_TIME         0    0
  IPC_RECV_WAIT_TIME         0    0
  FCM_SEND_WAIT_TIME         0    0
  FCM_RECV_WAIT_TIME         6    212
WLM_QUEUE_TIME_TOTAL         0    0
CF_WAIT_TIME                 32   1101
RECLAIM_WAIT_TIME            2    98
SMP_RECLAIM_WAIT_TIME        3    118

Component times
-------------------------------------------------------------------------------
```

```
-- Detailed breakdown of processing time --

                                   %                Total
                                   ---------------  -------------------------
Total processing                   100              0

Section execution
  TOTAL_SECTION_PROC_TIME          0                0
    TOTAL_SECTION_SORT_PROC_TIME   0                0
Compile
  TOTAL_COMPILE_PROC_TIME          0                0
  TOTAL_IMPLICIT_COMPILE_PROC_TIME 0                0
Transaction end processing
  TOTAL_COMMIT_PROC_TIME           0                0
  TOTAL_ROLLBACK_PROC_TIME         0                0
Utilities
  TOTAL_RUNSTATS_PROC_TIME         0                0
  TOTAL_REORGS_PROC_TIME           0                0
  TOTAL_LOAD_PROC_TIME             0                0

Buffer pool
------------------------------------------------------------------------------
Buffer pool hit ratios

Type            Ratio            Reads (Logical/Physical)
--------------  ---------------  ------------------------------------------
Data            84               1545/246
Index           73               1824/491
XDA             0                0/0
Temp data       0                0/0
Temp index      0                0/0
Temp XDA        0                0/0
GBP Data        94               (259 - 246)/259
GBP Index       96               (507 - 491)/507
GBP XDA         88               (1402 - 1366)/1776
LBP Data        18               (1268 - 4)/1545
LBP Index       29               (1287 - 0)/1824
LBP XDA         22               (1299 - 3)/1492

I/O
------------------------------------------------------------------------------
Buffer pool writes
  POOL_DATA_WRITES      = 0
  POOL_XDA_WRITES       = 0
  POOL_INDEX_WRITES     = 0
Direct I/O
  DIRECT_READS          = 0
  DIRECT_READ_REQS      = 0
  DIRECT_WRITES         = 0
  DIRECT_WRITE_REQS     = 0
Log I/O
  LOG_DISK_WAITS_TOTAL  = 0

Locking
------------------------------------------------------------------------------
                        Per activity                    Total
                        ------------------------------  ----------------------
LOCK_WAIT_TIME          0                               0
LOCK_WAITS              0                               0
LOCK_TIMEOUTS           0                               0
DEADLOCKS               0                               0
LOCK_ESCALS             0                               0

Routines
------------------------------------------------------------------------------
                              Per activity              Total
                              ------------------------  ------------------------
```

```
TOTAL_ROUTINE_INVOCATIONS     0                         0
TOTAL_ROUTINE_TIME            0                         0

TOTAL_ROUTINE_TIME per invocation   = 0

Sort
------------------------------------------------------------------------------
TOTAL_SORTS                         = 0
SORT_OVERFLOWS                      = 0
POST_THRESHOLD_SORTS                = 0
POST_SHRTHRESHOLD_SORTS             = 0

Network
------------------------------------------------------------------------------
Communications with remote clients
TCPIP_SEND_VOLUME per send          = 0          (0/0)
TCPIP_RECV_VOLUME per receive       = 0          (0/0)

Communications with local clients
IPC_SEND_VOLUME per send            = 0          (0/0)
IPC_RECV_VOLUME per receive         = 0          (0/0)

Fast communications manager
FCM_SEND_VOLUME per send            = 0          (0/0)
FCM_RECV_VOLUME per receive         = 0          (0/0)

Other
------------------------------------------------------------------------------
Compilation
  TOTAL_COMPILATIONS                = 0
  PKG_CACHE_INSERTS                 = 0
  PKG_CACHE_LOOKUPS                 = 0
Catalog cache
  CAT_CACHE_INSERTS                 = 0
  CAT_CACHE_LOOKUPS                 = 0
Transaction processing
  TOTAL_APP_COMMITS                 = 0
  INT_COMMITS                       = 0
  TOTAL_APP_ROLLBACKS               = 0
  INT_ROLLBACKS                     = 0
Log buffer
  NUM_LOG_BUFFER_FULL               = 0
Activities aborted/rejected
  ACT_ABORTED_TOTAL                 = 0
  ACT_REJECTED_TOTAL                = 0
Workload management controls
  WLM_QUEUE_ASSIGNMENTS_TOTAL       = 0
  WLM_QUEUE_TIME_TOTAL              = 0

DB2 utility operations
------------------------------------------------------------------------------
  TOTAL_RUNSTATS                    = 0
  TOTAL_REORGS                      = 0
  TOTAL_LOADS                       = 0

connection #:2
------------------------------------------------------------------------------

--Connection identifiers--
Application identifiers
  APPLICATION_HANDLE                = 65711
  APPLICATION_NAME                  = db2bp
  APPLICATION_ID                    = *N1.jwr.100406173430
Authorization IDs
  SYSTEM_AUTHID                     = JWR
  SESSION_AUTHID                    = JWR
Client attributes
```

```
  CLIENT_ACCTNG                    =
  CLIENT_USERID                    =
  CLIENT_APPLNAME                  =
  CLIENT_WRKSTNNAME                =
  CLIENT_PID                       = 30044
  CLIENT_PRDID                     = SQL09081
  CLIENT_PLATFORM                  = LINUXX8664
  CLIENT_PROTOCOL                  = LOCAL
-- Other connection details --
CONNECTION_START_TIME              = 2010-04-06-13.34.31.058344
NUM_LOCKS_HELD                     = 0

Work volume and throughput
-------------------------------------------------------------------------------
                                   Per second             Total
                                   ---------------------  ----------------------
TOTAL_APP_COMMITS                  0                      1
ACT_COMPLETED_TOTAL                0                      1
APP_RQSTS_COMPLETED_TOTAL          0                      2

TOTAL_CPU_TIME                     = 116307
TOTAL_CPU_TIME per request         = 58153

Row processing
  ROWS_READ/ROWS_RETURNED          = 0 (8/0)
  ROWS_MODIFIED                    = 5

Wait times
-------------------------------------------------------------------------------

-- Wait time as a percentage of elapsed time --

                                        %    Wait time/Total time
                                        ---  ---------------------------------
For requests                            58   410/696
For activities                          58   398/675

-- Time waiting for next client request --

CLIENT_IDLE_WAIT_TIME               = 9884
CLIENT_IDLE_WAIT_TIME per second    = 988

-- Detailed breakdown of TOTAL_WAIT_TIME --

                              %    Total
                              ---  ------------------------------------------
TOTAL_WAIT_TIME               100  410

I/O wait time
  POOL_READ_TIME              5    23
  POOL_WRITE_TIME             28   116
  DIRECT_READ_TIME            0    1
  DIRECT_WRITE_TIME           0    4
  LOG_DISK_WAIT_TIME          11   48
LOCK_WAIT_TIME                2    11
AGENT_WAIT_TIME               0    0
Network and FCM
  TCPIP_SEND_WAIT_TIME        0    0
  TCPIP_RECV_WAIT_TIME        0    0
  IPC_SEND_WAIT_TIME          0    1
  IPC_RECV_WAIT_TIME          0    0
  FCM_SEND_WAIT_TIME          0    0
  FCM_RECV_WAIT_TIME          1    5
WLM_QUEUE_TIME_TOTAL          0    0
CF_WAIT_TIME                  17   73
RECLAIM_WAIT_TIME             23   96
SMP_RECLAIM_WAIT_TIME         4    20
```

```
Component times
------------------------------------------------------------------------------
-- Detailed breakdown of processing time --

                                  %                Total
                                  ---------------  -------------------------
Total processing                  100              286

Section execution
  TOTAL_SECTION_PROC_TIME         96               276
    TOTAL_SECTION_SORT_PROC_TIME  0                0
Compile
  TOTAL_COMPILE_PROC_TIME         0                2
  TOTAL_IMPLICIT_COMPILE_PROC_TIME 0               0
Transaction end processing
  TOTAL_COMMIT_PROC_TIME          1                4
  TOTAL_ROLLBACK_PROC_TIME        0                0
Utilities
  TOTAL_RUNSTATS_PROC_TIME        0                0
  TOTAL_REORGS_PROC_TIME          0                0
  TOTAL_LOAD_PROC_TIME            0                0

Buffer pool
------------------------------------------------------------------------------
Buffer pool hit ratios

Type             Ratio            Reads (Logical/Physical)
---------------  ---------------  ------------------------------------------
Data             91               72/6
Index            100              46/0
XDA              0                0/0
Temp data        0                0/0
Temp index       0                0/0
Temp XDA         0                0/0
GBP Data         60               (10 - 6)/10
GBP Index        0                (8 - 0)/8
LBP Data         52               (34 - 0)/72
LBP Index        0                (46 - 0)/46

I/O
------------------------------------------------------------------------------
Buffer pool writes
  POOL_DATA_WRITES      = 36
  POOL_XDA_WRITES       = 0
  POOL_INDEX_WRITES     = 0
Direct I/O
  DIRECT_READS          = 1
  DIRECT_READ_REQS      = 1
  DIRECT_WRITES         = 4
  DIRECT_WRITE_REQS     = 1
Log I/O
  LOG_DISK_WAITS_TOTAL  = 13

Locking
------------------------------------------------------------------------------
                        Per activity                    Total
                        ------------------------------  ----------------------
LOCK_WAIT_TIME          11                              11
LOCK_WAITS              100                             1
LOCK_TIMEOUTS           0                               0
DEADLOCKS               0                               0
LOCK_ESCALS             0                               0

例程------------------------------------------------------------------------------
                              Per activity              Total
                              ------------------------  ------------------------
```

```
TOTAL_ROUTINE_INVOCATIONS     0                         0
TOTAL_ROUTINE_TIME            0                         0

TOTAL_ROUTINE_TIME per invocation   = 0

Sort
------------------------------------------------------------------------------
TOTAL_SORTS                         = 0
SORT_OVERFLOWS                      = 0
POST_THRESHOLD_SORTS                = 0
POST_SHRTHRESHOLD_SORTS             = 0

Network
------------------------------------------------------------------------------
Communications with remote clients
TCPIP_SEND_VOLUME per send          = 0          (0/0)
TCPIP_RECV_VOLUME per receive       = 0          (0/0)

Communications with local clients
IPC_SEND_VOLUME per send            = 54         (108/2)
IPC_RECV_VOLUME per receive         = 69         (138/2)

Fast communications manager
FCM_SEND_VOLUME per send            = 0          (0/0)
FCM_RECV_VOLUME per receive         = 432        (2592/6)

Other
------------------------------------------------------------------------------
Compilation
  TOTAL_COMPILATIONS                = 1
  PKG_CACHE_INSERTS                 = 2
  PKG_CACHE_LOOKUPS                 = 2
Catalog cache
  CAT_CACHE_INSERTS                 = 3
  CAT_CACHE_LOOKUPS                 = 8
Transaction processing
  TOTAL_APP_COMMITS                 = 1
  INT_COMMITS                       = 0
  TOTAL_APP_ROLLBACKS               = 0
  INT_ROLLBACKS                     = 0
Log buffer
  NUM_LOG_BUFFER_FULL               = 0
Activities aborted/rejected
  ACT_ABORTED_TOTAL                 = 0
  ACT_REJECTED_TOTAL                = 0
Workload management controls
  WLM_QUEUE_ASSIGNMENTS_TOTAL       = 0
  WLM_QUEUE_TIME_TOTAL              = 0

DB2 utility operations
------------------------------------------------------------------------------
  TOTAL_RUNSTATS                    = 0
  TOTAL_REORGS                      = 0
  TOTAL_LOADS                       = 0

connection #:3
------------------------------------------------------------------------------

--Connection identifiers--
Application identifiers
  APPLICATION_HANDLE                = 131323
  APPLICATION_NAME                  = db2bp
  APPLICATION_ID                    = *N2.jwr.100406173452
Authorization IDs
  SYSTEM_AUTHID                     = JWR
  SESSION_AUTHID                    = JWR
Client attributes
```

```
  CLIENT_ACCTNG                    =
  CLIENT_USERID                    =
  CLIENT_APPLNAME                  =
  CLIENT_WRKSTNNAME                =
  CLIENT_PID                       = 30510
  CLIENT_PRDID                     = SQL09081
  CLIENT_PLATFORM                  = LINUXX8664
  CLIENT_PROTOCOL                  = LOCAL
-- Other connection details --
CONNECTION_START_TIME              = 2010-04-06-13.34.52.398427
NUM_LOCKS_HELD                     = 0

Work volume and throughput
-------------------------------------------------------------------------------
                                 Per second             Total
                                 ---------------------  ----------------------
TOTAL_APP_COMMITS                0                      1
ACT_COMPLETED_TOTAL              0                      1
APP_RQSTS_COMPLETED_TOTAL        0                      2

TOTAL_CPU_TIME                   = 116624
TOTAL_CPU_TIME per request       = 58312

Row processing
  ROWS_READ/ROWS_RETURNED        = 0 (18/0)
  ROWS_MODIFIED                  = 4

Wait times
-------------------------------------------------------------------------------

-- Wait time as a percentage of elapsed time --

                                       %    Wait time/Total time
                                       ---  ---------------------------------
For requests                           82   717/864
For activities                         80   549/679

-- Time waiting for next client request --

CLIENT_IDLE_WAIT_TIME               = 12895
CLIENT_IDLE_WAIT_TIME per second    = 1289

-- Detailed breakdown of TOTAL_WAIT_TIME --

                             %    Total
                             ---  ------------------------------------------
TOTAL_WAIT_TIME              100  717

I/O wait time
  POOL_READ_TIME             2    16
  POOL_WRITE_TIME            18   136
  DIRECT_READ_TIME           0    3
  DIRECT_WRITE_TIME          0    2
  LOG_DISK_WAIT_TIME         10   77
LOCK_WAIT_TIME               3    27
AGENT_WAIT_TIME              0    0
Network and FCM
  TCPIP_SEND_WAIT_TIME       0    0
  TCPIP_RECV_WAIT_TIME       0    0
  IPC_SEND_WAIT_TIME         0    0
  IPC_RECV_WAIT_TIME         0    0
  FCM_SEND_WAIT_TIME         0    0
  FCM_RECV_WAIT_TIME         21   157
WLM_QUEUE_TIME_TOTAL         0    0
CF_WAIT_TIME                 9    66
RECLAIM_WAIT_TIME            12   92
SMP_RECLAIM_WAIT_TIME        16   119
```

```
Component times
------------------------------------------------------------------------------
-- Detailed breakdown of processing time --

                                  %                Total
                                  ---------------  -------------------------
Total processing                  100              147

Section execution
  TOTAL_SECTION_PROC_TIME         89               131
    TOTAL_SECTION_SORT_PROC_TIME  0                0
Compile
  TOTAL_COMPILE_PROC_TIME         4                6
  TOTAL_IMPLICIT_COMPILE_PROC_TIME 0               0
Transaction end processing
  TOTAL_COMMIT_PROC_TIME          1                2
  TOTAL_ROLLBACK_PROC_TIME        0                0
Utilities
  TOTAL_RUNSTATS_PROC_TIME        0                0
  TOTAL_REORGS_PROC_TIME          0                0
  TOTAL_LOAD_PROC_TIME            0                0

Buffer pool
------------------------------------------------------------------------------
Buffer pool hit ratios

Type             Ratio            Reads (Logical/Physical)
---------------  ---------------  -----------------------------------------
Data             91               47/4
Index            100              78/0
XDA              0                0/0
Temp data        0                0/0
Temp index       0                0/0
Temp XDA         0                0/0
GBP Data         26               (15 - 4)/15
GBP Index        0                (9 - 0)/9
LBP Data         6                (44 - 0)/47
LBP Index        48               (40 - 0)/78

I/O
------------------------------------------------------------------------------
Buffer pool writes
  POOL_DATA_WRITES      = 3
  POOL_XDA_WRITES       = 0
  POOL_INDEX_WRITES     = 35
Direct I/O
  DIRECT_READS          = 15
  DIRECT_READ_REQS      = 4
  DIRECT_WRITES         = 6
  DIRECT_WRITE_REQS     = 1
Log I/O
  LOG_DISK_WAITS_TOTAL  = 18

Locking
------------------------------------------------------------------------------
                        Per activity                  Total
                        ----------------------------  ----------------------
LOCK_WAIT_TIME          27                            27
LOCK_WAITS              200                           2
LOCK_TIMEOUTS           0                             0
DEADLOCKS               0                             0
LOCK_ESCALS             0                             0

Routines
------------------------------------------------------------------------------
                              Per activity            Total
```

```
                              ------------------------  ------------------------
TOTAL_ROUTINE_INVOCATIONS     0                         0
TOTAL_ROUTINE_TIME            0                         0

TOTAL_ROUTINE_TIME per invocation   = 0

Sort
-------------------------------------------------------------------------------
TOTAL_SORTS                         = 1
SORT_OVERFLOWS                      = 0
POST_THRESHOLD_SORTS                = 0
POST_SHRTHRESHOLD_SORTS             = 0

Network
-------------------------------------------------------------------------------
Communications with remote clients
TCPIP_SEND_VOLUME per send          = 0          (0/0)
TCPIP_RECV_VOLUME per receive       = 0          (0/0)

Communications with local clients
IPC_SEND_VOLUME per send            = 54         (108/2)
IPC_RECV_VOLUME per receive         = 73         (146/2)

Fast communications manager
FCM_SEND_VOLUME per send            = 0          (0/0)
FCM_RECV_VOLUME per receive         = 1086       (10864/10)

Other
-------------------------------------------------------------------------------
Compilation
  TOTAL_COMPILATIONS                = 1
  PKG_CACHE_INSERTS                 = 2
  PKG_CACHE_LOOKUPS                 = 2
Catalog cache
  CAT_CACHE_INSERTS                 = 0
  CAT_CACHE_LOOKUPS                 = 9
Transaction processing
  TOTAL_APP_COMMITS                 = 1
  INT_COMMITS                       = 0
  TOTAL_APP_ROLLBACKS               = 0
  INT_ROLLBACKS                     = 0
Log buffer
  NUM_LOG_BUFFER_FULL               = 0
Activities aborted/rejected
  ACT_ABORTED_TOTAL                 = 0
  ACT_REJECTED_TOTAL                = 0
Workload management controls
  WLM_QUEUE_ASSIGNMENTS_TOTAL       = 0
  WLM_QUEUE_TIME_TOTAL              = 0

DB2 utility operations
-------------------------------------------------------------------------------
  TOTAL_RUNSTATS                    = 0
  TOTAL_REORGS                      = 0
  TOTAL_LOADS                       = 0

628 record(s) selected.

Return Status = 0
```

CURRENTAPPS 过程 - 生成时间点应用程序处理度量的报告

CURRENTAPPS 过程收集有关每个连接的工作单元、代理程序和活动的当前瞬时处理状态的信息。

语法

授权

需要以下特权:

- 对 MONREPORT 模块的 EXECUTE 特权

以下示例演示调用 CURRENTAPPS 过程的各种方法:

```
call monreport.currentapps;
call monreport.currentapps();
```

CURRENTSQL 过程 - 生成总结活动的报告

CURRENTSQL 过程生成文本格式的报告，该报告总结当前运行的活动。

语法

参数

member
类型为 SMALLINT 的输入参数，用于确定是显示特定成员或分区的数据还是所有成员总计数据。如果未指定该参数（或者指定了 NULL），那么报告显示所有成员总计值。如果指定了有效成员编号，那么报告显示该成员的值。

授权

需要以下特权:

- 对 MONREPORT 模块的 EXECUTE 特权

以下示例演示调用 CURRENTSQL 过程的各种方法。第一个示例产生的报告显示所有成员聚集的活动度量:

```
call monreport.currentsql;
```

下一个示例产生的报告显示特定于成员编号 4 的活动性能的活动度量:

```
call monreport.currentsql(4);
```

DBSUMMARY 过程 - 生成系统和应用程序性能指标的摘要报告

DBSUMMARY 过程生成文本格式的监视报告，该报告总结系统和应用程序性能指标。

“数据库摘要”报告包含整个数据库以及每个连接、工作负载、服务类和数据库成员的关键性能指标的深入监视数据。

语法

参数

monitoring_interval

类型为 INTEGER 的可选输入参数，用于指定在报告所收集监视数据之前，收集这些数据的持续时间（以秒计）。例如，如果指定监视时间间隔为 30 秒，那么该例程会调用表函数，等待 30 秒并再次调用表函数。然后 DBSUMMARY 过程会计算差别，这些差别反映时间间隔期间的更改。如果未指定 *monitoring_interval* 参数（或者指定了 NULL），那么缺省值为 10。有效输入的范围是整数值 0 到 3600（即，最长 1 小时）。

授权

需要以下特权:

- 对 MONREPORT 模块的 EXECUTE 特权

示例

以下示例演示调用 DBSUMMARY 过程的各种方法。

第一个示例生成一个报告，其中显示的数据对应 30 秒时间间隔。

```
call monreport.dbsummary(30);
```

下一个示例生成一个报告，其中显示的数据对应 10 秒时间间隔（缺省值）:

```
call monreport.dbsummary;
```

此过程调用返回以下输出:

```
  Result set 1
  --------------

  TEXT
  ------------------------------------------------------------------------------
  ------------------------------------------------------------------------------
  Monitoring report - database summary
  ------------------------------------------------------------------------------
  Database:               SAMPLE
  Generated:                                  04/06/2010 13:35:24
  Interval monitored:                         10

  ==============================================================================
  Part 1 - System performance

  Work volume and throughput
  ------------------------------------------------------------------------------
                                    Per second             Total
                                    ---------------------  ----------------------
  TOTAL_APP_COMMITS                 0                      2
  ACT_COMPLETED_TOTAL               0                      9
  APP_RQSTS_COMPLETED_TOTAL         0                      6

  TOTAL_CPU_TIME                    = 2649800
  TOTAL_CPU_TIME per request        = 441633

  Row processing
```

```
  ROWS_READ/ROWS_RETURNED         = 97 (685/7)
  ROWS_MODIFIED                   = 117

Wait times
------------------------------------------------------------------------------

-- Wait time as a percentage of elapsed time --

                                      %    Wait time/Total time
                                      ---  ----------------------------------
For requests                          19   3434/17674
For activities                        10   1203/11613

-- Time waiting for next client request --

CLIENT_IDLE_WAIT_TIME               = 70566
CLIENT_IDLE_WAIT_TIME per second    = 7056

-- Detailed breakdown of TOTAL_WAIT_TIME --

                              %    Total
                              ---  ------------------------------------------
TOTAL_WAIT_TIME               100  3434

I/O wait time
  POOL_READ_TIME              23   805
  POOL_WRITE_TIME             8    280
  DIRECT_READ_TIME            3    131
  DIRECT_WRITE_TIME           3    104
  LOG_DISK_WAIT_TIME          10   344
LOCK_WAIT_TIME                0    18
AGENT_WAIT_TIME               0    0
Network and FCM
  TCPIP_SEND_WAIT_TIME        0    0
  TCPIP_RECV_WAIT_TIME        0    0
  IPC_SEND_WAIT_TIME          0    0
  IPC_RECV_WAIT_TIME          0    0
  FCM_SEND_WAIT_TIME          0    0
  FCM_RECV_WAIT_TIME          6    212
WLM_QUEUE_TIME_TOTAL          0    0
CF_WAIT_TIME                  32   1101
RECLAIM_WAIT_TIME             2    98
SMP_RECLAIM_WAIT_TIME         3    118

Component times
------------------------------------------------------------------------------
-- Detailed breakdown of processing time --

                                    %                Total
                                    ---------------  -------------------------
Total processing                    100              14240

Section execution
  TOTAL_SECTION_PROC_TIME           2                365
    TOTAL_SECTION_SORT_PROC_TIME    0                0
Compile
  TOTAL_COMPILE_PROC_TIME           0                17
  TOTAL_IMPLICIT_COMPILE_PROC_TIME  2                294
Transaction end processing
  TOTAL_COMMIT_PROC_TIME            0                36
  TOTAL_ROLLBACK_PROC_TIME          0                0
Utilities
  TOTAL_RUNSTATS_PROC_TIME          0                0
  TOTAL_REORGS_PROC_TIME            0                0
  TOTAL_LOAD_PROC_TIME              0                0

Buffer pool
```

```
------------------------------------------------------------------------------
Buffer pool hit ratios

Type             Ratio            Reads (Logical/Physical)
---------------  ---------------  ------------------------------------------
Data             84               1545/246
Index            73               1824/491
XDA              0                0/0
Temp data        0                0/0
Temp index       0                0/0
Temp XDA         0                0/0
GBP Data         94               (259 - 246)/259
GBP Index        96               (507 - 491)/507
GBP XDA          88               (1402 - 1366)/1776
LBP Data         18               (1268 - 4)/1545
LBP Index        29               (1287 - 0)/1824
Temp XDA         22               (1299 - 3)/1492

I/O
------------------------------------------------------------------------------
Buffer pool writes
  POOL_DATA_WRITES      = 39
  POOL_XDA_WRITES       = 0
  POOL_INDEX_WRITES     = 35
Direct I/O
  DIRECT_READS          = 1006
  DIRECT_READ_REQS      = 131
  DIRECT_WRITES         = 484
  DIRECT_WRITE_REQS     = 28
Log I/O
  LOG_DISK_WAITS_TOTAL  = 63

Locking
------------------------------------------------------------------------------
                         Per activity                    Total
                         ------------------------------  ----------------------
LOCK_WAIT_TIME           2                               18
LOCK_WAITS               22                              2
LOCK_TIMEOUTS            0                               0
DEADLOCKS                0                               0
LOCK_ESCALS              0                               0

Routines
------------------------------------------------------------------------------
                              Per activity             Total
                              -----------------------  ------------------------
TOTAL_ROUTINE_INVOCATIONS     0                        1
TOTAL_ROUTINE_TIME            1117                     10058

TOTAL_ROUTINE_TIME per invocation   = 10058

Sort
------------------------------------------------------------------------------
TOTAL_SORTS                         = 5
SORT_OVERFLOWS                      = 0
POST_THRESHOLD_SORTS                = 0
POST_SHRTHRESHOLD_SORTS             = 0

Network
------------------------------------------------------------------------------
Communications with remote clients
TCPIP_SEND_VOLUME per send          = 0          (0/0)
TCPIP_RECV_VOLUME per receive       = 0          (0/0)

Communications with local clients
IPC_SEND_VOLUME per send            = 137        (1101/8)
IPC_RECV_VOLUME per receive         = 184        (1106/6)
```

```
Fast communications manager
FCM_SEND_VOLUME per send            = 3475       (31277/9)
FCM_RECV_VOLUME per receive         = 2433       (131409/54)

Other
--------------------------------------------------------------------------------
Compilation
  TOTAL_COMPILATIONS                = 4
  PKG_CACHE_INSERTS                 = 11
  PKG_CACHE_LOOKUPS                 = 13
Catalog cache
  CAT_CACHE_INSERTS                 = 74
  CAT_CACHE_LOOKUPS                 = 112
Transaction processing
  TOTAL_APP_COMMITS                 = 2
  INT_COMMITS                       = 2
  TOTAL_APP_ROLLBACKS               = 0
  INT_ROLLBACKS                     = 0
Log buffer
  NUM_LOG_BUFFER_FULL               = 0
Activities aborted/rejected
  ACT_ABORTED_TOTAL                 = 0
  ACT_REJECTED_TOTAL                = 0
Workload management controls
  WLM_QUEUE_ASSIGNMENTS_TOTAL       = 0
  WLM_QUEUE_TIME_TOTAL              = 0

DB2 utility operations
--------------------------------------------------------------------------------
  TOTAL_RUNSTATS                    = 0
  TOTAL_REORGS                      = 0
  TOTAL_LOADS                       = 0

================================================================================
Part 2 - Application performance drill down

Application performance database-wide
--------------------------------------------------------------------------------
TOTAL_CPU_TIME          TOTAL_       TOTAL_APP_     ROWS_READ +
per request             WAIT_TIME %  COMMITS        ROWS_MODIFIED
----------------------  -----------  -------------  ----------------------------
441633                  19           2              802

Application performance by connection
--------------------------------------------------------------------------------
APPLICATION_   TOTAL_CPU_TIME        TOTAL_         TOTAL_APP_     ROWS_READ +
HANDLE         per request           WAIT_TIME %    COMMITS        ROWS_MODIFIED
-------------  -------------------   -----------    -------------  -------------
180            0                     0              0              0
65711          495970                46             1              566
131323         324379                43             1              222

Application performance by service class
--------------------------------------------------------------------------------
SERVICE_       TOTAL_CPU_TIME        TOTAL_         TOTAL_APP_     ROWS_READ +
CLASS_ID       per request           WAIT_TIME %    COMMITS        ROWS_MODIFIED
--------       -------------------   -----------    -------------  -------------
11             0                     0              0              0
12             0                     0              0              0
13             440427                19             2              802

Application performance by workload
--------------------------------------------------------------------------------
WORKLOAD_      TOTAL_CPU_TIME          TOTAL_       TOTAL_APP_     ROWS_READ +
NAME           per request             WAIT_TIME %  COMMITS        ROWS_MODIFIED
-------------  ----------------------  -----------  -------------  -------------
```

```
SYSDEFAULTADM  0                    0           0           0
SYSDEFAULTUSE  410174               45          2           788

================================================================================
Part 3 - Member level information

- I/O wait time is
  (POOL_READ_TIME + POOL_WRITE_TIME + DIRECT_READ_TIME + DIRECT_WRITE_TIME).

        TOTAL_CPU_TIME          TOTAL_        RQSTS_COMPLETED_  I/O
MEMBER  per request             WAIT_TIME %   TOTAL             wait time
------  ----------------------  -----------  ----------------  -----------------
0       17804                   0            9                 10
1       108455                  47           14                866
2       74762                   41           13                441


237 record(s) selected.

Return Status = 0
```

LOCKWAIT 过程 - 生成当前锁定等待的报告

“锁定等待”报告包含有关当前正在进行的每个锁定等待的信息。详细信息包括有关锁定持有者和请求者及所持有锁定和所请求锁定的特征的信息。

语法

授权

需要以下特权:

- 对 MONREPORT 模块的 EXECUTE 特权

以下示例演示调用 LOCKWAIT 过程的各种方法:

```
call monreport.lockwait;

call monreport.lockwait();
```

```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Monitoring report - current lock waits
--------------------------------------------------------------------------------
Database:               SAMPLE
Generated:              08/28/2009 07:16:26

================================================================================
Part 1 - Summary of current lock waits

--------------------------------------------------------------------------------

      REQ_APPLICATION  LOCK_MODE  HLD_APPLICATION  LOCK_  LOCK_OBJECT_TYPE
#     HANDLE           REQUESTED  _HANDLE          MODE
----  ---------------  ---------  ---------------  -----  ----------------------
1     26               U          21               U      ROW
2     25               U          21               U      ROW
3     24               U          21               U      ROW
4     23               U          21               U      ROW
5     22               U          21               U      ROW
6     27               U          21               U      ROW

================================================================================

...


390 record(s) selected.

Return Status = 0
```

图 1. 样本 MONREPORT.LOCKWAIT 输出 - 摘要部分

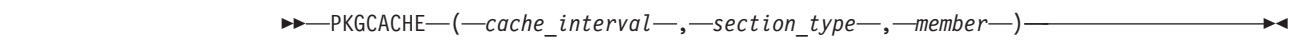
```
================================================================================
Part 2: Details for each current lock wait

lock wait #:1
--------------------------------------------------------------------------------

-- Lock details --

LOCK_NAME             = 0400050004000000000000052
LOCK_WAIT_START_TIME = 2009-08-28-07.15.31.013802
LOCK_OBJECT_TYPE      = ROW
TABSCHEMA             = TRIPATHY
TABNAME               = INVENTORY
ROWID                 = 4
LOCK_STATUS           = W
LOCK_ATTRIBUTES       = 0000000000000000
ESCALATION            = N

-- Requestor and holder application details --

Attributes            Requestor                      Holder
-------------------   ----------------------------   ----------------------------
APPLICATION_HANDLE    26                             21
APPLICATION_ID        *LOCAL.tripathy.090828111531   *LOCAL.tripathy.090828111435
APPLICATION_NAME      java                           java
SESSION_AUTHID        TRIPATHY                       TRIPATHY
MEMBER                0                              0
LOCK_MODE             -                              U
LOCK_MODE_REQUESTED   U                              -

-- Lock holder current agents --

AGENT_TID             = 41
REQUEST_TYPE          = FETCH
EVENT_STATE           = IDLE
EVENT_OBJECT          = REQUEST
EVENT_TYPE            = WAIT
ACTIVITY_ID           =
UOW_ID                =

-- Lock holder current activities --

ACTIVITY_ID           = 1
UOW_ID                = 1
LOCAL_START_TIME      = 2009-08-28-07.14.31.079757
ACTIVITY_TYPE         = READ_DML
ACTIVITY_STATE        = IDLE

STMT_TEXT             =
select * from inventory for update

-- Lock requestor waiting agent and activity --

AGENT_TID             = 39
REQUEST_TYPE          = FETCH
ACTIVITY_ID           = 1
UOW_ID                = 1
LOCAL_START_TIME      = 2009-08-28-07.15.31.012935
ACTIVITY_TYPE         = READ_DML
ACTIVITY_STATE        = EXECUTING

STMT_TEXT             =
select * from inventory for update
```

图 2. 样本 MONREPORT.LOCKWAIT 输出 - 详细信息部分

PKGCACHE 过程 - 生成程序包高速缓存度量值的摘要报告

“程序包高速缓存摘要”报告列示按各种度量计算的程序包高速缓存中累积的最常用语句。

语法



参数

cache_interval
类型为 INTEGER 的可选输入参数，用于指定报告仅应包括过去数分钟（由 *cache_interval* 值指定）内更新的程序包高速缓存条目的数据。例如，*cache_interval* 值 60 会根据过去 60 分钟内更新的程序包高速缓存条目产生报告。有效值为 0 到 10080 之间的整数（支持最长 7 天的时间间隔）。如果未指定该参数（或者指定了 NULL），那么报告包括程序包高速缓存条目的数据，不管它们何时添加或更新都是如此。

section_type
类型为 CHAR(1) 的可选输入参数，用于指定报告是否应包括静态 SQL 和/或动态 SQL 的数据。如果未指定该参数（或者指定了 NULL），那么报告包括两种类型的 SQL 的数据。有效值为 d 或 D（用于动态）以及 s 或 S（用于静态）。

member
类型为 SMALLINT 的可选输入参数，用于确定是显示特定成员或分区的数据还是所有成员总计数据。如果未指定该参数（或者指定了 NULL），那么报告显示所有成员总计值。如果指定了有效成员编号，那么报告显示该成员的值。

授权

需要以下特权:

- 对 MONREPORT 模块的 EXECUTE 特权

以下示例演示调用 PKGCACHE 过程的各种方法。第一个示例根据程序包高速缓存中的所有语句产生报告，并显示所有成员聚集的数据:

```
call monreport.pkgcache;
```

下一个示例根据程序包高速缓存中其度量在过去 30 分钟内进行了更新的动态和静态语句产生报告，并显示所有成员聚集的数据:

```
call monreport.pkgcache(30);
```

下一个示例根据程序包高速缓存中的所有动态语句产生报告，并显示所有成员聚集的数据:

```
call monreport.pkgcache(DEFAULT, 'd');
```

下一个示例根据程序包高速缓存中其度量在过去 30 分钟内进行了更新的动态和静态语句产生报告，并显示特定于成员编号 4 的数据:

```
call db2monreport.pkgcache(30, DEFAULT, 4);
```

第 31 章 UTL_DIR 模块

UTL_DIR 模块提供一组例程，用于维护与 UTL_FILE 模块一起使用的目录别名。

注: UTL_DIR 模块不发出任何直接的操作系统调用，例如 **mkdir** 或 **rmdir** 命令。对物理目录的维护不在此模块的范围之内。

此模块的模式为 SYSIBMADM。

UTL_DIR 模块包括以下内置例程。

表 30. UTL_DIR 模块中可用的内置例程

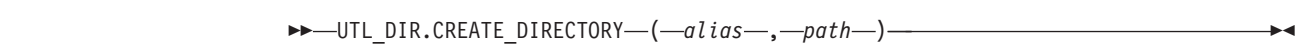
例程名称	描述
CREATE_DIRECTORY 过程	对指定路径创建目录别名。
CREATE_OR_REPLACE_DIRECTORY 过程	对指定路径创建或替换目录别名。
DROP_DIRECTORY 过程	删除指定的目录别名。
GET_DIRECTORY_PATH 过程	获取指定的目录别名的相应路径。

CREATE_DIRECTORY 过程 - 创建目录别名

CREATE_DIRECTORY 过程对指定路径创建目录别名。

目录信息存储在 SYSTOOLS.DIRECTORIES 中，SYSTOOLS.DIRECTORIES 是您第一次对每个数据库引用此模块时在 SYSTOOLSPACE 中创建的。

语法



过程参数

alias
类型为 VARCHAR(128) 的输入参数，用于指定目录别名。

path
类型为 VARCHAR(1024) 的输入参数，用于指定路径。

授权

对 UTL_DIR 模块的 EXECUTE 特权。

示例

创建目录别名，并在对 UTL_FILE.FOPEN 函数的调用中使用该别名。

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE proc1()
BEGIN
  DECLARE  v_filehandle    UTL_FILE.FILE_TYPE;
```

```
  DECLARE  isOpen          BOOLEAN;
  DECLARE  v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
  CALL UTL_DIR.CREATE_DIRECTORY('mydir', '/home/user/temp/mydir');
  SET v_filehandle = UTL_FILE.FOPEN('mydir',v_filename,'w');
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
    IF isOpen != TRUE THEN
      RETURN -1;
    END IF;
  CALL DBMS_OUTPUT.PUT_LINE('Opened file: ' || v_filename);
  CALL UTL_FILE.FCLOSE(v_filehandle);
END@

CALL proc1@
```

此示例生成以下输出:

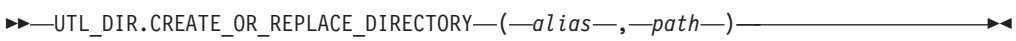
```
Opened file: myfile.csv
```

CREATE_OR_REPLACE_DIRECTORY 过程 - 创建或替换目录别名

CREATE_OR_REPLACE_DIRECTORY 过程对指定路径创建或替换目录别名。

目录信息存储在 SYSTOOLS.DIRECTORIES 中，SYSTOOLS.DIRECTORIES 是您第一次对每个数据库引用此模块时在 SYSTOOLSPACE 中创建的。

语法



过程参数

alias
类型为 VARCHAR(128) 的输入参数，用于指定目录别名。

path
类型为 VARCHAR(1024) 的输入参数，用于指定路径。

授权

对 UTL_DIR 模块的 EXECUTE 特权。

示例

示例 1: 创建目录别名。因为该目录已经存在，所以发生了错误。

```
CALL UTL_DIR.CREATE_DIRECTORY('mydir', 'home/user/temp/empdir')@
```

此示例生成以下输出:

```
SQL0438N
应用程序发生了错误或警告，诊断文本为:
"已定义目录别名"。SQLSTATE=23505
```

示例 2: 创建或替换目录别名。

```
CALL UTL_DIR.CREATE_OR_REPLACE_DIRECTORY('mydir', 'home/user/temp/empdir')@
```

此示例生成以下输出:

```
  Return Status = 0
```

DROP_DIRECTORY 过程 - 删除目录别名

DROP_DIRECTORY 过程删除指定的目录别名。

语法

过程参数

alias
类型为 VARCHAR(128) 的输入参数，用于指定目录别名。

授权

对 UTL_DIR 模块的 EXECUTE 特权。

示例

删除指定的目录别名。

```
CALL UTL_DIR.DROP_DIRECTORY('mydir')@
```

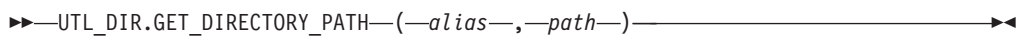
此示例生成以下输出:

```
  Return Status = 0
```

GET_DIRECTORY_PATH 过程 - 获取目录别名的路径

GET_DIRECTORY_PATH 过程返回目录别名的对应路径。

语法



过程参数

alias
类型为 VARCHAR(128) 的输入参数，用于指定目录别名。

path
类型为 VARCHAR(1024) 的输出参数，用于指定对目录别名定义的路径。

授权

对 UTL_DIR 模块的 EXECUTE 特权。

示例

获取对目录别名定义的路径。

```
CALL UTL_DIR.GET_DIRECTORY_PATH('mydir', ? )@
```

此示例生成以下输出:

```
Value of output parameters
--------------------------
Parameter Name  : PATH
Parameter Value : home/rhoda/temp/mydir

Return Status = 0
```

第 32 章 UTL_FILE 模块

UTL_FILE 模块提供一组例程，用于读取和写入数据库服务器的文件系统上的文件。

此模块的模式为 SYSIBMADM。

UTL_FILE 模块包括以下内置例程和类型。

表 31. UTL_FILE 模块中可用的内置例程

例程名称	描述
FCLOSE 过程	关闭指定的文件。
FCLOSE_ALL 过程	关闭所有打开的文件。
FCOPY 过程	将一个文件中的文本复制到另一个文件。
FFLUSH 过程	将未写入的数据写入文件
FOPEN 函数	打开文件。
FREMOVE 过程	除去文件。
FRENAME 过程	重命名文件。
GET_LINE 过程	从文件获取一行。
IS_OPEN 函数	确定指定的文件是否已打开。
NEW_LINE 过程	将行结束字符序列写至文件。
PUT 过程	将字符串写至文件。
PUT_LINE 过程	将一行写至文件。
PUTF 过程	将格式字符串写至文件。
UTL_FILE.FILE_TYPE	存储文件句柄。

以下是应用程序可能接收到的指定情况（它们被 Oracle 称为“异常”）列表。

表 32. 应用程序的指定情况

条件名	描述
access_denied	对文件的访问被操作系统拒绝。
charsetmismatch	使用 FOPEN_NCHAR 打开了文件，但后续 I/O 操作使用了非 CHAR 函数，如 PUTF 或 GET_LINE。
delete_failed	无法删除文件。
file_open	文件已打开。
internal_error	UTL_FILE 模块中出现未处理的内部错误。
invalid_filehandle	文件句柄不存在。
invalid_filename	路径中不存在具有指定名称的文件。
invalid_maxlinesize	FOPEN 的 MAX_LINESIZE 值无效。它必须在 1 与 32672 之间。
invalid_mode	FOPEN 中的 open_mode 参数无效。
invalid_offset	FSEEK 的 ABSOLUTE_OFFSET 参数无效。它必须大于 0 且小于文件中的总字节数。

表 32. 应用程序的指定情况 (续)

条件名	描述
invalid_operation	未能按请求打开或操作文件。
invalid_path	指定的路径不存在或对数据库不可视
read_error	无法读取该文件。
rename_failed	无法重命名该文件。
write_error	无法写至该文件。

用法说明

要引用文件系统上的目录，请使用目录别名。可通过调用 UTL_DIR.CREATE_DIRECTORY 或 UTL_DIR.CREATE_OR_REPLACE_DIRECTORY 过程来创建目录别名。例如，`CALL UTL_DIR.CREATE_DIRECTORY('mydir', 'home/user/temp/mydir')@`。

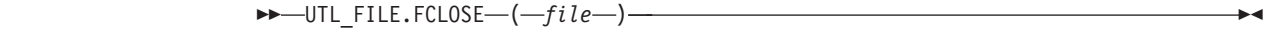
UTL_FILE 模块通过使用 DB2 实例标识来执行文件操作。因此，如果您要打开文件，请验证 DB2 实例标识是否具有适当的操作系统许可权。

UTL_FILE 模块仅在非分区数据库环境中受支持。

FCLOSE 过程 - 关闭已打开文件

FCLOSE 过程关闭指定的文件。

语法



过程参数

file
类型为 UTL_FILE.FILE_TYPE 的输入或输出参数，其中包含文件句柄。文件关闭时，此值设置为 0。

授权

对 UTL_FILE 模块的 EXECUTE 特权。

示例

打开文件，将某些文本写至该文件，然后关闭该文件。

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE proc1()
BEGIN
  DECLARE  v_filehandle    UTL_FILE.FILE_TYPE;
  DECLARE  isOpen          BOOLEAN;
  DECLARE  v_dirAlias      VARCHAR(50) DEFAULT 'mydir';
  DECLARE  v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
  CALL UTL_DIR.CREATE_OR_REPLACE_DIRECTORY('mydir', '/tmp');
  SET v_filehandle = UTL_FILE.FOPEN(v_dirAlias,v_filename,'w');
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
```

```
    IF isOpen != TRUE THEN
      RETURN -1;
    END IF;
  CALL UTL_FILE.PUT_LINE(v_filehandle,'Some text to write to the file.');
  CALL UTL_FILE.FCLOSE(v_filehandle);
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
    IF isOpen != TRUE THEN
      CALL DBMS_OUTPUT.PUT_LINE('Closed file: ' || v_filename);
    END IF;
END@

CALL proc1@
```

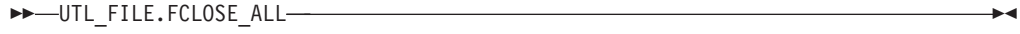
此示例生成以下输出:

```
Closed file: myfile.csv
```

FCLOSE_ALL 过程 - 关闭所有已打开文件

FCLOSE_ALL 过程关闭所有已打开文件。即使没有要关闭的已打开文件，该过程也会成功运行。

语法



授权

对 UTL_FILE 模块的 EXECUTE 特权。

示例

打开一些文件，将某些文本写至这些文件，然后关闭所有已打开文件。

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE proc1()
BEGIN
  DECLARE  v_filehandle    UTL_FILE.FILE_TYPE;
  DECLARE  v_filehandle2   UTL_FILE.FILE_TYPE;
  DECLARE  isOpen          BOOLEAN;
  DECLARE  v_dirAlias      VARCHAR(50) DEFAULT 'mydir';
  DECLARE  v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
  DECLARE  v_filename2     VARCHAR(20) DEFAULT 'myfile2.csv';
  CALL UTL_DIR.CREATE_OR_REPLACE_DIRECTORY('mydir', '/tmp');
  SET v_filehandle = UTL_FILE.FOPEN(v_dirAlias,v_filename,'w');
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
    IF isOpen != TRUE THEN
      RETURN -1;
    END IF;
  CALL UTL_FILE.PUT_LINE(v_filehandle,'Some text to write to a file.');
  SET v_filehandle2 = UTL_FILE.FOPEN(v_dirAlias,v_filename2,'w');
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle2 );
    IF isOpen != TRUE THEN
      RETURN -1;
    END IF;
  CALL UTL_FILE.PUT_LINE(v_filehandle2,'Some text to write to another file.');
  CALL UTL_FILE.FCLOSE_ALL;
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
    IF isOpen != TRUE THEN
      CALL DBMS_OUTPUT.PUT_LINE(v_filename || ' is now closed.');
    END IF;
```

```
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle2 );
    IF isOpen != TRUE THEN
      CALL DBMS_OUTPUT.PUT_LINE(v_filename2 || ' is now closed.');
    END IF;
END@

CALL proc1@
```

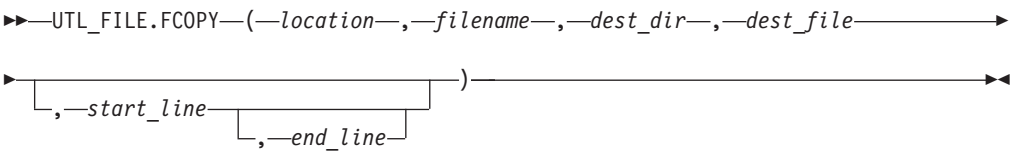
此示例生成以下输出:

```
myfile.csv is now closed.
myfile2.csv is now closed.
```

FCOPY 过程 - 将一个文件中的文本复制到另一个文件

FCOPY 过程将一个文件中的文本复制到另一个文件。

语法



过程参数

location
类型为 VARCHAR(128) 的输入参数，用于指定包含源文件的目录的别名。

filename
类型为 VARCHAR(255) 的输入参数，用于指定源文件的名称。

dest_dir
类型为 VARCHAR(128) 的输入参数，用于指定目标目录的别名。

dest_file
类型为 VARCHAR(255) 的输入参数，用于指定目标文件的名称。

start_line
类型为 INTEGER 的可选输入参数，用于指定源文件中要复制的第一行文本的行号。缺省值为 1。

end_line
类型为 INTEGER 的可选输入参数，用于指定源文件中要复制的最后一行文本的行号。如果此参数被省略或为空，那么该过程继续复制所有文本直到文件结尾。

授权

对 UTL_FILE 模块的 EXECUTE 特权。

示例

创建文件 empfile.csv 的副本，该文件包含 emp 表中的职员的逗号定界列表。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
```

```
  DECLARE    v_empfile       UTL_FILE.FILE_TYPE;
  DECLARE    v_dirAlias      VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_src_file      VARCHAR(20) DEFAULT 'empfile.csv';
  DECLARE    v_dest_file     VARCHAR(20) DEFAULT 'empcopy.csv';
  DECLARE    v_empline       VARCHAR(200);
  CALL UTL_FILE.FCOPY(v_dirAlias,v_src_file,v_dirAlias,v_dest_file);
END@

CALL proc1@
```

此示例生成以下输出:

```
  Return Status = 0
```

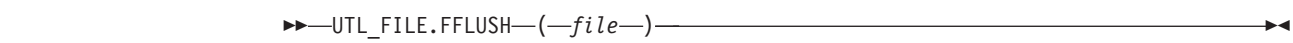
文件副本 empcopy.csv 包含以下数据:

```
10,CHRISTINE,I,HAAS,A00,3978,1/1/1965,PRES,18,F,8/24/1933,52750,1000,4220
20,MICHAEL,L,THOMPSON,B01,3476,10/10/1973,MANAGER,18,M,2/2/1948,41250,800,3300
30,SALLY,A,KWAN,C01,4738,4/5/1975,MANAGER,20,F,5/11/1941,38250,800,3060
50,JOHN,B,GEYER,E01,6789,8/17/1949,MANAGER,16,M,9/15/1925,40175,800,3214
60,IRVING,F,STERN,D11,6423,9/14/1973,MANAGER,16,M,7/7/1945,32250,500,2580
70,EVA,D,PULASKI,D21,7831,9/30/1980,MANAGER,16,F,5/26/1953,36170,700,2893
90,EILEEN,W,HENDERSON,E11,5498,8/15/1970,MANAGER,16,F,5/15/1941,29750,600,2380
100,THEODORE,Q,SPENSER,E21,972,6/19/1980,MANAGER,14,M,12/18/1956,26150,500,2092
```

FFLUSH 过程 - 将未写入的数据写入文件

FFLUSH 过程强制将写入缓冲区中的未写入数据写至文件。

语法



过程参数

file
类型为 UTL_FILE.FILE_TYPE 的输入参数，其中包含文件句柄。

授权

对 UTL_FILE 模块的 EXECUTE 特权。

示例

调用 NEW_LINE 过程后刷新每行。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_empfile_src    UTL_FILE.FILE_TYPE;
  DECLARE    v_empfile_tgt    UTL_FILE.FILE_TYPE;
  DECLARE    v_dirAlias       VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_src_file       VARCHAR(20) DEFAULT 'empfile.csv';
  DECLARE    v_dest_file      VARCHAR(20) DEFAULT 'empfilenew.csv';
  DECLARE    v_empline        VARCHAR(200);
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLSTATE1 CHAR(5) DEFAULT '00000';
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'SET SQLSTATE1 = SQLSTATE;

  SET v_empfile_src = UTL_FILE.FOPEN(v_dirAlias,v_src_file,'r');
  SET v_empfile_tgt = UTL_FILE.FOPEN(v_dirAlias,v_dest_file,'w');
```

```
  loop1: LOOP
    CALL UTL_FILE.GET_LINE(v_empfile_src,v_empline);
    IF SQLSTATE1 = '02000' THEN -- NO DATA FOUND
      LEAVE loop1;
    END IF;
    CALL UTL_FILE.PUT(v_empfile_tgt,v_empline);
    CALL UTL_FILE.NEW_LINE(v_empfile_tgt);
    CALL UTL_FILE.FFLUSH(v_empfile_tgt);
  END LOOP;
  CALL DBMS_OUTPUT.PUT_LINE('Updated file: ' || v_dest_file);
  CALL UTL_FILE.FCLOSE_ALL;
END@

CALL proc1@
```

此示例生成以下输出:

```
Updated file: empfilenew.csv
```

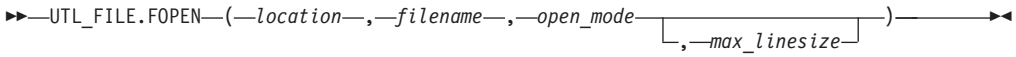
更新后的文件 empfilenew.csv 包含以下数据:

```
10,CHRISTINE,I,HAAS,A00,3978,1/1/1965,PRES,18,F,8/24/1933,52750,1000,4220

20,MICHAEL,L,THOMPSON,B01,3476,10/10/1973,MANAGER,18,M,2/2/1948,41250,800,3300

30,SALLY,A,KWAN,C01,4738,4/5/1975,MANAGER,20,F,5/11/1941,38250,800,3060

50,JOHN,B,GEYER,E01,6789,8/17/1949,MANAGER,16,M,9/15/1925,40175,800,3214

60,IRVING,F,STERN,D11,6423,9/14/1973,MANAGER,16,M,7/7/1945,32250,500,2580

70,EVA,D,PULASKI,D21,7831,9/30/1980,MANAGER,16,F,5/26/1953,36170,700,2893

90,EILEEN,W,HENDERSON,E11,5498,8/15/1970,MANAGER,16,F,5/15/1941,29750,600,2380

100,THEODORE,Q,SPENSER,E21,972,6/19/1980,MANAGER,14,M,12/18/1956,26150,500,2092
```

FOPEN 函数 - 打开文件

FOPEN 函数打开文件以执行 I/O。

语法



返回值

此函数返回类型为 UTL_FILE.FILE_TYPE 的值，用于指示已打开文件的句柄。

函数参数

location
类型为 VARCHAR(128) 的输入参数，用于指定包含该文件的目录的别名。

filename
类型为 VARCHAR(255) 的输入参数，用于指定该文件的名称。

open_mode
类型为 VARCHAR(10) 的输入参数，用于指定该文件的打开方式:

a 追加至文件

r 读取文件

w 写至文件

max_linesize

类型为 INTEGER 的可选输入参数，用于指定行的最大大小（以字符计）。缺省值为 1024 字节。 在读取方式下，如果尝试读取大小超过 *max_linesize* 的行，那么会抛出异常。在写入和追加方式下，如果尝试写入大小超过 *max_linesize* 的行，那么会抛出异常。行结束字符不会计入行大小。

授权

对 UTL_FILE 模块的 EXECUTE 特权。

示例

打开文件，将某些文本写至该文件，然后关闭该文件。

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE proc1()
BEGIN
  DECLARE  v_filehandle    UTL_FILE.FILE_TYPE;
  DECLARE  isOpen          BOOLEAN;
  DECLARE  v_dirAlias      VARCHAR(50) DEFAULT 'mydir';
  DECLARE  v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
  CALL UTL_DIR.CREATE_OR_REPLACE_DIRECTORY('mydir', '/tmp');
  SET v_filehandle = UTL_FILE.FOPEN(v_dirAlias,v_filename,'w');
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
    IF isOpen != TRUE THEN
      RETURN -1;
    END IF;
  CALL DBMS_OUTPUT.PUT_LINE('Opened file: ' || v_filename);
  CALL UTL_FILE.PUT_LINE(v_filehandle,'Some text to write to the file.');
  CALL UTL_FILE.FCLOSE(v_filehandle);
END@

CALL proc1@
```

此示例生成以下输出。

```
Opened file: myfile.csv
```

FREMOVE 过程 - 除去文件

FREMOVE 过程从系统中除去指定文件。如果该文件不存在，那么此过程会抛出异常。

语法

过程参数

location

类型为 VARCHAR(128) 的输入参数，用于指定包含该文件的目录的别名。

filename

类型为 VARCHAR(255) 的输入参数，用于指定该文件的名称。

授权

对 UTL_FILE 模块的 EXECUTE 特权。

示例

从系统中除去文件 myfile.csv。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_dirAlias      VARCHAR(50) DEFAULT 'mydir';
  DECLARE v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
  CALL UTL_FILE.FREMOVE(v_dirAlias,v_filename);
  CALL DBMS_OUTPUT.PUT_LINE('Removed file: ' || v_filename);
END@

CALL proc1@
```

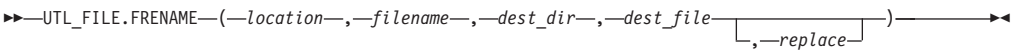
此示例生成以下输出:

```
Removed file: myfile.csv
```

FRENAME 过程 - 重命名文件

FRENAME 过程可重命名指定的文件。重命名文件可有效地将文件从一个位置移至另一个位置。

语法



过程参数

location
类型为 VARCHAR(128) 的输入参数，用于指定包含要重命名的文件的目录的别名。

filename
类型为 VARCHAR(255) 的输入参数，用于指定要重命名的文件的名称。

dest_dir
类型为 VARCHAR(128) 的输入参数，用于指定目标目录的别名。

dest_file
类型为 VARCHAR(255) 的输入参数，用于指定文件的新名称。

replace
类型为 INTEGER 的可选输入参数，用于指定是否替换目录 *dest_dir* 中的文件 *dest_file*（如果该文件已存在）：

1 替换现有文件。

0 该文件已存在时抛出异常。如果未对 *replace* 指定任何值，那么这是缺省值。

授权

对 UTL_FILE 模块的 EXECUTE 特权。

示例

重命名文件 empfile.csv，该文件包含 emp 表中的职员的逗号定界列表。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_dirAlias      VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_src_file      VARCHAR(20) DEFAULT 'oldemp.csv';
  DECLARE    v_dest_file     VARCHAR(20) DEFAULT 'newemp.csv';
  DECLARE    v_replace       INTEGER DEFAULT 1;
  CALL UTL_FILE.FRENAME(v_dirAlias,v_src_file,v_dirAlias,
        v_dest_file,v_replace);
  CALL DBMS_OUTPUT.PUT_LINE('The file ' || v_src_file ||
    ' has been renamed to ' || v_dest_file);
END@

CALL proc1@
```

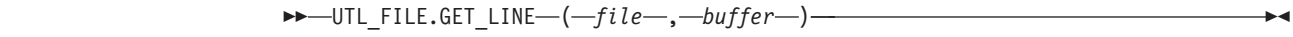
此示例生成以下输出:

```
The file oldemp.csv has been renamed to newemp.csv
```

GET_LINE 过程 - 从文件获取行

GET_LINE 过程从指定文件获取文本行。该文本行不包括行结束终止符。没有其他要读取的行时，该过程会抛出 NO_DATA_FOUND 异常。

语法



过程参数

file
类型为 UTL_FILE.FILE_TYPE 的输入参数，其中包含已打开文件的句柄。

buffer
类型为 VARCHAR(32672) 的输出参数，其中包含文件中的文本行。

授权

对 UTL_FILE 模块的 EXECUTE 特权。

示例

从头到尾读取并显示文件 empfile.csv 中的记录。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_empfile       UTL_FILE.FILE_TYPE;
  DECLARE    v_dirAlias      VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_filename      VARCHAR(20) DEFAULT 'empfile.csv';
```

```
  DECLARE    v_empline       VARCHAR(200);
  DECLARE    v_count         INTEGER DEFAULT 0;
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLSTATE1 CHAR(5) DEFAULT '00000';
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'SET SQLSTATE1 = SQLSTATE;

  SET v_empfile = UTL_FILE.FOPEN(v_dirAlias,v_filename,'r');

  loop1: LOOP
    CALL UTL_FILE.GET_LINE(v_empfile, v_empline);
    IF SQLSTATE1 = '02000' THEN -- NO DATA FOUND
      LEAVE loop1;
    END IF;
    CALL DBMS_OUTPUT.PUT_LINE(v_empline);
    SET v_count = v_count + 1;
  END LOOP;
  CALL DBMS_OUTPUT.PUT_LINE('End of file ' || v_filename || ' - ' || v_count
        || ' records retrieved');
  CALL UTL_FILE.FCLOSE(v_empfile);
END@

CALL proc1@
```

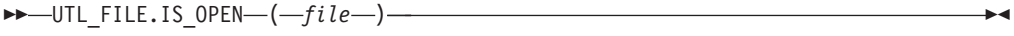
此示例生成以下输出:

```
10,CHRISTINE,I,HAAS,A00,3978,1/1/1965,PRES,18,F,8/24/1933,52750,1000,4220

20,MICHAEL,L,THOMPSON,B01,3476,10/10/1973,MANAGER,18,M,2/2/1948,41250,800,3300

30,SALLY,A,KWAN,C01,4738,4/5/1975,MANAGER,20,F,5/11/1941,38250,800,3060

50,JOHN,B,GEYER,E01,6789,8/17/1949,MANAGER,16,M,9/15/1925,40175,800,3214

60,IRVING,F,STERN,D11,6423,9/14/1973,MANAGER,16,M,7/7/1945,32250,500,2580

70,EVA,D,PULASKI,D21,7831,9/30/1980,MANAGER,16,F,5/26/1953,36170,700,2893

90,EILEEN,W,HENDERSON,E11,5498,8/15/1970,MANAGER,16,F,5/15/1941,29750,600,2380

100,THEODORE,Q,SPENSER,E21,972,6/19/1980,MANAGER,14,M,12/18/1956,26150,500,2092

End of file empfile.csv - 8 records retrieved
```

IS_OPEN 函数 - 确定指定的文件是否已打开

IS_OPEN 函数确定指定的文件是否已打开。

语法



返回值

此函数返回类型为 BOOLEAN 的值，用于指示指定的文件已打开 (TRUE) 还是已关闭 (FALSE)。

函数参数

file
类型为 UTL_FILE.FILE_TYPE 的输入参数，其中包含文件句柄。

授权

对 UTL_FILE 模块的 EXECUTE 特权。

示例

以下示例演示: 在将文本写至文件之前，您可调用 IS_OPEN 函数以检查该文件是否已打开。

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE proc1()
BEGIN
  DECLARE  v_filehandle    UTL_FILE.FILE_TYPE;
  DECLARE  isOpen          BOOLEAN;
  DECLARE  v_dirAlias      VARCHAR(50) DEFAULT 'mydir';
  DECLARE  v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
  CALL UTL_DIR.CREATE_OR_REPLACE_DIRECTORY('mydir', '/tmp');
  SET v_filehandle = UTL_FILE.FOPEN(v_dirAlias,v_filename,'w');
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
    IF isOpen != TRUE THEN
      RETURN -1;
    END IF;
  CALL UTL_FILE.PUT_LINE(v_filehandle,'Some text to write to the file.');
  CALL DBMS_OUTPUT.PUT_LINE('Updated file: ' || v_filename);
  CALL UTL_FILE.FCLOSE(v_filehandle);
END@

CALL proc1@
```

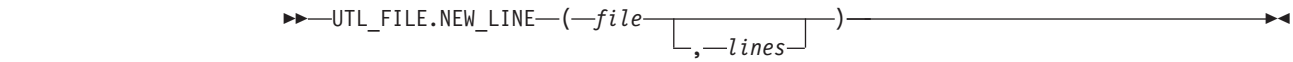
此示例生成以下输出。

```
Updated file: myfile.csv
```

NEW_LINE 过程 - 将行结束字符序列写至文件

NEW_LINE 过程将行结束字符序列写至指定文件。

语法



过程参数

file
类型为 UTL_FILE.FILE_TYPE 的输入参数，其中包含文件句柄。

lines
类型为 INTEGER 的可选输入参数，用于指定要写至文件的行结束字符序列的数目。缺省值为 1。

授权

对 UTL_FILE 模块的 EXECUTE 特权。

示例

写入包含带有三个空格的职员记录列表的文件。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_empfile_src    UTL_FILE.FILE_TYPE;
  DECLARE    v_empfile_tgt    UTL_FILE.FILE_TYPE;
  DECLARE    v_dirAlias       VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_src_file       VARCHAR(20) DEFAULT 'empfile.csv';
  DECLARE    v_dest_file      VARCHAR(20) DEFAULT 'empfilenew.csv';
  DECLARE    v_empline        VARCHAR(200);
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLSTATE1 CHAR(5) DEFAULT '00000';
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'SET SQLSTATE1 = SQLSTATE;

  SET v_empfile_src = UTL_FILE.FOPEN(v_dirAlias,v_src_file,'r');
  SET v_empfile_tgt = UTL_FILE.FOPEN(v_dirAlias,v_dest_file,'w');

  loop1: LOOP
    CALL UTL_FILE.GET_LINE(v_empfile_src,v_empline);
    IF SQLSTATE1 = '02000' THEN -- NO DATA FOUND
      LEAVE loop1;
    END IF;
    CALL UTL_FILE.PUT(v_empfile_tgt,v_empline);
    CALL UTL_FILE.NEW_LINE(v_empfile_tgt, 2);
  END LOOP;

  CALL DBMS_OUTPUT.PUT_LINE('Wrote to file: ' || v_dest_file);
  CALL UTL_FILE.FCLOSE_ALL;
END@

CALL proc1@
```

此示例生成以下输出:

```
Wrote to file: empfilenew.csv
```

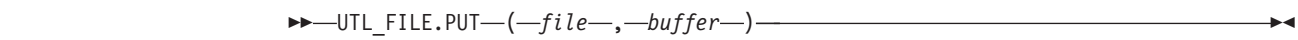
更新后的文件 empfilenew.csv 包含以下数据:

```
10,CHRISTINE,I,HAAS,A00,3978,1/1/1965,PRES,18,F,8/24/1933,52750,1000,4220


20,MICHAEL,L,THOMPSON,B01,3476,10/10/1973,MANAGER,18,M,2/2/1948,41250,800,3300


30,SALLY,A,KWAN,C01,4738,4/5/1975,MANAGER,20,F,5/11/1941,38250,800,3060


50,JOHN,B,GEYER,E01,6789,8/17/1949,MANAGER,16,M,9/15/1925,40175,800,3214


60,IRVING,F,STERN,D11,6423,9/14/1973,MANAGER,16,M,7/7/1945,32250,500,2580


70,EVA,D,PULASKI,D21,7831,9/30/1980,MANAGER,16,F,5/26/1953,36170,700,2893


90,EILEEN,W,HENDERSON,E11,5498,8/15/1970,MANAGER,16,F,5/15/1941,29750,600,2380


100,THEODORE,Q,SPENSER,E21,972,6/19/1980,MANAGER,14,M,12/18/1956,26150,500,2092
```

PUT 过程 - 将字符串写至文件

PUT 过程将字符串写至指定文件。字符串结尾不会写入任何行结束字符序列。

语法



过程参数

file
类型为 UTL_FILE.FILE_TYPE 的输入参数，其中包含文件句柄。

buffer
类型为 VARCHAR(32672) 的输入参数，用于指定要写至文件的文本。

授权

对 UTL_FILE 模块的 EXECUTE 特权。

示例

使用 PUT 过程将字符串添加至文件，然后使用 NEW_LINE 过程来添加行结束字符序列。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_empfile_src    UTL_FILE.FILE_TYPE;
  DECLARE    v_empfile_tgt    UTL_FILE.FILE_TYPE;
  DECLARE    v_dirAlias       VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_src_file       VARCHAR(20) DEFAULT 'empfile.csv';
  DECLARE    v_dest_file      VARCHAR(20) DEFAULT 'empfilenew.csv';
  DECLARE    v_empline        VARCHAR(200);
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLSTATE1 CHAR(5) DEFAULT '00000';
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'SET SQLSTATE1 = SQLSTATE;

  SET v_empfile_src = UTL_FILE.FOPEN(v_dirAlias,v_src_file,'r');
  SET v_empfile_tgt = UTL_FILE.FOPEN(v_dirAlias,v_dest_file,'w');

  loop1: LOOP
    CALL UTL_FILE.GET_LINE(v_empfile_src,v_empline);
    IF SQLSTATE1 = '02000' THEN -- NO DATA FOUND
      LEAVE loop1;
    END IF;
    CALL UTL_FILE.PUT(v_empfile_tgt,v_empline);
    CALL UTL_FILE.NEW_LINE(v_empfile_tgt, 2);
  END LOOP;

  CALL DBMS_OUTPUT.PUT_LINE('Wrote to file: ' || v_dest_file);
  CALL UTL_FILE.FCLOSE_ALL;
END@

CALL proc1@
```

此示例生成以下输出:

```
Wrote to file: empfilenew.csv
```

更新后的文件 empfilenew.csv 包含以下数据:

```
10,CHRISTINE,I,HAAS,A00,3978,1/1/1965,PRES,18,F,8/24/1933,52750,1000,4220
```

```
20,MICHAEL,L,THOMPSON,B01,3476,10/10/1973,MANAGER,18,M,2/2/1948,41250,800,3300

30,SALLY,A,KWAN,C01,4738,4/5/1975,MANAGER,20,F,5/11/1941,38250,800,3060

50,JOHN,B,GEYER,E01,6789,8/17/1949,MANAGER,16,M,9/15/1925,40175,800,3214

60,IRVING,F,STERN,D11,6423,9/14/1973,MANAGER,16,M,7/7/1945,32250,500,2580

70,EVA,D,PULASKI,D21,7831,9/30/1980,MANAGER,16,F,5/26/1953,36170,700,2893

90,EILEEN,W,HENDERSON,E11,5498,8/15/1970,MANAGER,16,F,5/15/1941,29750,600,2380

100,THEODORE,Q,SPENSER,E21,972,6/19/1980,MANAGER,14,M,12/18/1956,26150,500,2092
```

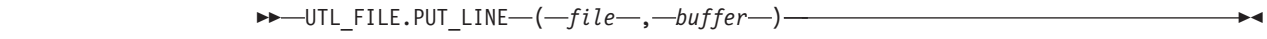
用法说明

使用 PUT 过程将字符串添加至文件后，请使用 NEW_LINE 过程将行结束字符序列添加至文件。

PUT_LINE 过程 - 将一行文本写至文件

PUT_LINE 过程将包括行结束字符序列的一行文本写至指定文件。

语法



过程参数

file
类型为 UTL_FILE.FILE_TYPE 的输入参数，其中包含该行将写至的文件的句柄。

buffer
类型为 VARCHAR(32672) 的输入参数，用于指定要写至文件的文本。

授权

对 UTL_FILE 模块的 EXECUTE 特权。

示例

使用 PUT_LINE 过程将文本行写至文件。

```
CALL proc1@
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_empfile_src    UTL_FILE.FILE_TYPE;
  DECLARE    v_empfile_tgt    UTL_FILE.FILE_TYPE;
  DECLARE    v_dirAlias       VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_src_file       VARCHAR(20) DEFAULT 'empfile.csv';
  DECLARE    v_dest_file      VARCHAR(20) DEFAULT 'empfilenew2.csv';
  DECLARE    v_empline        VARCHAR(200);
```

```
  DECLARE    v_count          INTEGER DEFAULT 0;
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLSTATE1 CHAR(5) DEFAULT '00000';
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'SET SQLSTATE1 = SQLSTATE;

  SET v_empfile_src = UTL_FILE.FOPEN(v_dirAlias,v_src_file,'r');
  SET v_empfile_tgt = UTL_FILE.FOPEN(v_dirAlias,v_dest_file,'w');

  loop1: LOOP
    CALL UTL_FILE.GET_LINE(v_empfile_src,v_empline);
    IF SQLSTATE1 = '02000' THEN -- NO DATA FOUND
      LEAVE loop1;
    END IF;
    SET v_count = v_count + 1;
    CALL UTL_FILE.PUT(v_empfile_tgt,'Record ' || v_count || ': ');
    CALL UTL_FILE.PUT_LINE(v_empfile_tgt,v_empline);
  END LOOP;
  CALL DBMS_OUTPUT.PUT_LINE('End of file ' || v_src_file || ' - ' || v_count
    || ' records retrieved');
  CALL UTL_FILE.FCLOSE_ALL;
END@

CALL proc1@
```

此示例生成以下输出:

```
End of file empfile.csv - 8 records retrieved
```

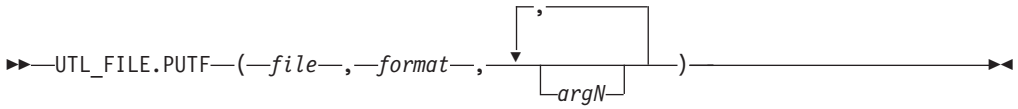
更新后的文件 empfilenew2.csv 包含以下数据:

```
Record 1: 10,CHRISTINE,I,HAAS,A00,3978,1/1/1965,PRES,18,F,8/24/1933,52750,1000,4220

Record 2: 20,MICHAEL,L,THOMPSON,B01,3476,10/10/1973,MANAGER,18,M,2/2/1948,41250,800,3300

Record 3: 30,SALLY,A,KWAN,C01,4738,4/5/1975,MANAGER,20,F,5/11/1941,38250,800,3060

Record 4: 50,JOHN,B,GEYER,E01,6789,8/17/1949,MANAGER,16,M,9/15/1925,40175,800,3214

Record 5: 60,IRVING,F,STERN,D11,6423,9/14/1973,MANAGER,16,M,7/7/1945,32250,500,2580

Record 6: 70,EVA,D,PULASKI,D21,7831,9/30/1980,MANAGER,16,F,5/26/1953,36170,700,2893

Record 7: 90,EILEEN,W,HENDERSON,E11,5498,8/15/1970,MANAGER,16,F,5/15/1941,29750,600,2380

Record 8: 100,THEODORE,Q,SPENSER,E21,972,6/19/1980,MANAGER,14,M,12/18/1956,26150,500,2092
```

PUTF 过程 - 将格式字符串写至文件

PUTF 过程将格式字符串写至指定的文件。

语法



过程参数

file

类型为 UTL_FILE.FILE_TYPE 的输入参数，其中包含文件句柄。

format

类型为 VARCHAR(1024) 的输入参数，用于指定格式化文本时使用的字符串。特殊字符序列 %s 将替换为 *argN* 的值。特殊字符序列 \n 指示换行。

argN

类型为 VARCHAR(1024) 的可选输入参数，用于指定格式字符串中要替换特殊字符序列 %s 的对应实例的值。最多可指定 5 个参数: *arg1* 到 *arg5*。*arg1* 将替换 %s 的第一个实例，*arg2* 替换 %s 的第二个实例，以此类推。

授权

对 UTL_FILE 模块的 EXECUTE 特权。

示例

格式化职员数据。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_filehandle    UTL_FILE.FILE_TYPE;
  DECLARE v_dirAlias      VARCHAR(50) DEFAULT 'mydir';
  DECLARE v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
  DECLARE v_format        VARCHAR(200);
  SET v_format = '%s %s, %s\nSalary: $%s Commission: $%s\n\n';
  CALL UTL_DIR.CREATE_OR_REPLACE_DIRECTORY('mydir', '/tmp');
  SET v_filehandle = UTL_FILE.FOPEN(v_dirAlias,v_filename,'w');
  CALL UTL_FILE.PUTF(v_filehandle,v_format,'000030','SALLY','KWAN','40175','3214');
  CALL DBMS_OUTPUT.PUT_LINE('Wrote to file: ' || v_filename);
  CALL UTL_FILE.FCLOSE(v_filehandle);
END@

CALL proc1@
```

此示例生成以下输出:

```
Wrote formatted text to file: myfile.csv
```

格式化文件 myfile.csv 包含以下数据:

```
000030 SALLY, KWAN
Salary: $40175 Commission: $3214
```

UTL_FILE.FILE_TYPE

UTL_FILE.FILE_TYPE 是 UTL_FILE 模块中的例程使用的文件句柄类型。

示例

声明类型为 UTL_FILE.FILE_TYPE 的变量。

```
  DECLARE  v_filehandle    UTL_FILE.FILE_TYPE;
```

第 33 章 UTL_MAIL 模块

UTL_MAIL 模块能够发送电子邮件。

此模块的模式为 SYSIBMADM。

UTL_MAIL 模块包括下列例程。

表 33. UTL_MAIL 模块中可用的内置例程

例程名称	描述
SEND 过程	将电子邮件打包并发送至 SMTP 服务器。
SEND_ATTACH_RAW 过程	与 SEND 过程相同，但带有 BLOB 附件。
SEND_ATTACH_VARCHAR2	与 SEND 过程相同，但带有 VARCHAR 附件。

用法说明

为了使用 UTL_MAIL 模块成功发送电子邮件，数据库配置参数 SMTP_SERVER 必须包含一个或多个有效 SMTP 服务器地址。

示例

示例 1: 设置使用缺省端口 25 的单个 SMTP 服务器:

```
db2 update db cfg using smtp_server 'smtp.ibm.com'
```

示例 2: 设置使用端口 2000（而不是缺省端口 25）的单个 SMTP 服务器:

```
db2 update db cfg using smtp_server 'smtp2.ibm.com:2000'
```

示例 3: 设置 SMTP 服务器列表:

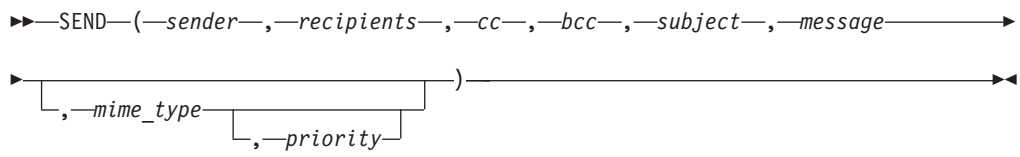
```
db2 update db cfg using smtp_server
   'smtp.example.com,smtp1.example.com:23,smtp2.example.com:2000'
```

注: 电子邮件将按列示顺序发送至每个 SMTP 服务器，直到从其中一个 SMTP 服务器接收到成功应答。

SEND 过程 - 将电子邮件发送至 SMTP 服务器

SEND 过程能够将电子邮件发送至 SMTP 服务器。

语法



参数

sender
类型为 VARCHAR(256) 的输入参数，用于指定发送方的电子邮件地址。

recipients
类型为 VARCHAR(32672) 的输入参数，用于指定用逗号分隔的接收方电子邮件地址。

cc 类型为 VARCHAR(32672) 的输入参数，用于指定用逗号分隔的副本接收方电子邮件地址。

bcc
类型为 VARCHAR(32672) 的输入参数，用于指定用逗号分隔的密送副本接收方电子邮件地址。

subject
类型为 VARCHAR(32672) 的输入参数，用于指定电子邮件的主题行。

message
类型为 VARCHAR(32672) 的输入参数，用于指定电子邮件的正文。

mime_type
类型为 VARCHAR(1024) 的可选输入参数，用于指定消息的 MIME 类型。缺省值为 'text/plain; charset=us-ascii'。

priority
类型为 INTEGER 的可选参数，用于指定电子邮件的优先级。缺省值为 3。

授权

对 UTL_MAIL 模块的 EXECUTE 特权。

示例

示例 1: 以下匿名块发送简单电子邮件消息。

```
BEGIN
  DECLARE v_sender VARCHAR(30);
  DECLARE v_recipients VARCHAR(60);
  DECLARE v_subj VARCHAR(20);
  DECLARE v_msg VARCHAR(200);

  SET v_sender = 'kkent@mycorp.com';
  SET v_recipients = 'bwayne@mycorp.com,pparker@mycorp.com';
  SET v_subj = 'Holiday Party';
  SET v_msg = 'This year''s party is scheduled for Friday, Dec. 21 at ' ||
     '6:00 PM. Please RSVP by Dec. 15th.';
  CALL UTL_MAIL.SEND(v_sender, v_recipients, NULL, NULL, v_subj, v_msg);
END@
```

此示例生成以下输出:

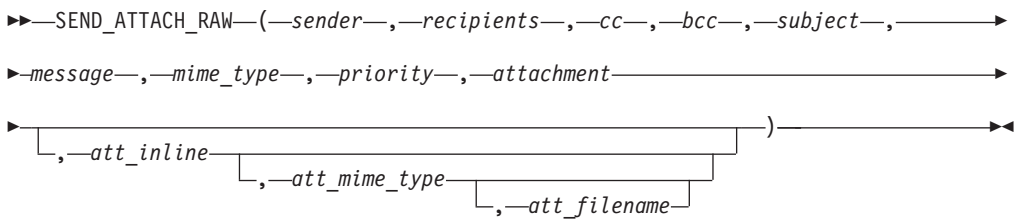
```
BEGIN
  DECLARE v_sender VARCHAR(30);
  DECLARE v_recipients VARCHAR(60);
  DECLARE v_subj VARCHAR(20);
  DECLARE v_msg VARCHAR(200);

  SET v_sender = 'kkent@mycorp.com';
  SET v_recipients = 'bwayne@mycorp.com,pparker@mycorp.com';
  SET v_subj = 'Holiday Party';
```

```
  SET v_msg = 'This year''s party is scheduled for Friday, Dec. 21 at ' ||
     '6:00 PM. Please RSVP by Dec. 15th.';
  CALL UTL_MAIL.SEND(v_sender, v_recipients, NULL, NULL, v_subj, v_msg);
END
DB20000I  The SQL command completed successfully.
```

SEND_ATTACH_RAW 过程 - 将带有 BLOB 附件的电子邮件发送至 SMTP 服务器

SEND_ATTACH_RAW 过程能够将带有二进制附件的电子邮件发送至 SMTP 服务器。

语法



参数

sender
类型为 VARCHAR(256) 的输入参数，用于指定发送方的电子邮件地址。

recipients
类型为 VARCHAR(32672) 的输入参数，用于指定用逗号分隔的接收方电子邮件地址。

cc 类型为 VARCHAR(32672) 的输入参数，用于指定用逗号分隔的副本接收方电子邮件地址。

bcc
类型为 VARCHAR(32672) 的输入参数，用于指定用逗号分隔的密送副本接收方电子邮件地址。

subject
类型为 VARCHAR(32672) 的输入参数，用于指定电子邮件的主题行。

message
类型为 VARCHAR(32672) 的输入参数，用于指定电子邮件的正文。

mime_type
类型为 VARCHAR(1024) 的输入参数，用于指定消息的 MIME 类型。缺省值为 `'text/plain; charset=us-ascii'`。

priority
类型为 INTEGER 的输入参数，用于指定电子邮件的优先级。缺省值为 3。

attachment
类型为 BLOB(10M) 的输入参数，其中包含附件。

att_inline
类型为 BOOLEAN 的可选输入参数，用于指定附件是否为可视直接插入。如果设置为“true”，那么附件为可视直接插入，如果设置为“false”，那么附件为不可视直接插入。缺省值为“true”。

att_mime_type
类型为 VARCHAR(1024) 的可选输入参数，用于指定附件的 MIME 类型。缺省值为 application/octet。

att_filename
类型为 VARCHAR(512) 的可选输入参数，用于指定包含附件的文件名。缺省值为 NULL。

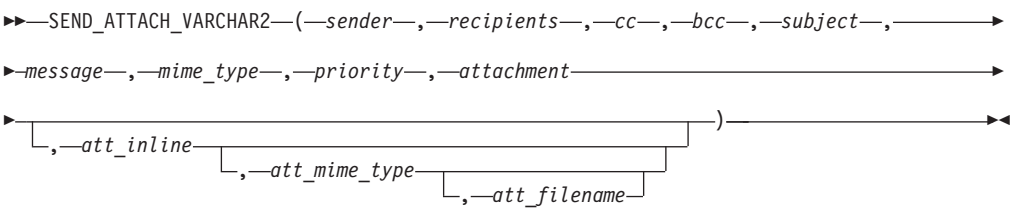
授权

对 UTL_MAIL 模块的 EXECUTE 特权。

SEND_ATTACH_VARCHAR2 过程 - 将带有 VARCHAR 附件的电子邮件发送至 SMTP 服务器

SEND_ATTACH_VARCHAR2 过程能够将带有文本附件的电子邮件发送至 SMTP 服务器。

语法



参数

sender
类型为 VARCHAR(256) 的输入参数，用于指定发送方的电子邮件地址。

recipients
类型为 VARCHAR(32672) 的输入参数，用于指定用逗号分隔的接收方电子邮件地址。

cc 类型为 VARCHAR(32672) 的输入参数，用于指定用逗号分隔的副本接收方电子邮件地址。

bcc
类型为 VARCHAR(32672) 的输入参数，用于指定用逗号分隔的密送副本接收方电子邮件地址。

subject
类型为 VARCHAR(32672) 的输入参数，用于指定电子邮件的主题行。

message
类型为 VARCHAR(32672) 的输入参数，用于指定电子邮件的正文。

mime_type
类型为 VARCHAR(1024) 的输入参数，用于指定消息的 MIME 类型。缺省值为 'text/plain; charset=us-ascii'。

priority
类型为 INTEGER 的输入参数，用于指定电子邮件的优先级。缺省值为 3。

attachment
类型为 VARCHAR(32000) 的输入参数，其中包含附件。

att_inline
类型为 BOOLEAN 的可选输入参数，用于指定附件是否为可视直接插入。如果设置为“true”，那么附件为可视直接插入，如果设置为“false”，那么附件为不可视直接插入。缺省值为“true”。

att_mime_type
类型为 VARCHAR(1024) 的可选输入参数，用于指定附件的 MIME 类型。缺省值为 `'text/plain; charset=us-ascii'`。

att_filename
类型为 VARCHAR(512) 的可选输入参数，用于指定包含附件的文件名。缺省值为 NULL。

授权

对 UTL_MAIL 模块的 EXECUTE 特权。

第 34 章 UTL_SMTP 模块

UTL_SMTP 模块允许通过简单电子邮件传输协议（SMTP）发送电子邮件。

UTL_SMTP 模块包括下列例程。

表 34. UTL_SMTP 模块中可用的内置例程

例程名称	描述
CLOSE_DATA 过程	结束电子邮件消息。
COMMAND 过程	执行 SMTP 命令。
COMMAND_REPLIES 过程	在应该出现多行应答的位置执行 SMTP 命令。
DATA 过程	指定电子邮件消息正文。
EHLO 过程	与 SMTP 服务器进行初始信息交换并返回扩展信息。
HELO 过程	与 SMTP 服务器进行初始信息交换。
HELP 过程	发送 HELP 命令。
MAIL 过程	启动邮件事务。
NOOP 过程	发送空命令。
OPEN_CONNECTION 函数	打开连接。
OPEN_CONNECTION 过程	打开连接。
OPEN_DATA 过程	发送 DATA 命令。
QUIT 过程	终止 SMTP 会话并断开连接。
RCPT 过程	指定电子邮件消息的接收方。
RSET 过程	终止当前邮件事务。
VRFY 过程	验证电子邮件地址。
WRITE_DATA 过程	写入电子邮件消息的部分。
WRITE_RAW_DATA 过程	写入由 RAW 数据组成的电子邮件消息部分。

下表列示模块中可用的公用变量。

表 35. UTL_SMTP 模块中可用的内置类型

公用变量	数据类型	描述
connection	RECORD	SMTP 连接的描述。
reply	RECORD	SMTP 应答行。

CONNECTION 记录类型提供 SMTP 连接的描述。

```
ALTER MODULE SYSIBMADM.UTL_SMTP PUBLISH TYPE connection AS ROW
(
  /* name or IP address of the remote host running SMTP server */
    host VARCHAR(255),
  /* SMTP server port number */
    port INTEGER,
  /* transfer timeout in seconds */
    tx_timeout INTEGER,
);
```

REPLY 记录类型提供 SMTP 应答行的描述。REPLIES 是一组 SMTP 应答行。

```
ALTER MODULE SYSIBMADM.UTL_SMTP PUBLISH TYPE reply AS ROW
(
  /* 3 digit reply code received from the SMTP server */
    code INTEGER,
  /* the text of the message received from the SMTP server */
    text VARCHAR(508)
);
```

示例

示例 1: 以下过程使用 UTL_SMTP 模块构造并发送文本电子邮件消息。

```
CREATE OR REPLACE PROCEDURE send_mail(
IN p_sender VARCHAR(4096),
IN p_recipient VARCHAR(4096),
IN p_subj VARCHAR(4096),
IN p_msg VARCHAR(4096),
IN p_mailhost VARCHAR(4096))
SPECIFIC send_mail
LANGUAGE SQL
BEGIN
  DECLARE v_conn UTL_SMTP.CONNECTION;
  DECLARE v_crlf VARCHAR(2);
  DECLARE v_port INTEGER CONSTANT 25;

  SET v_crlf = CHR(13) || CHR(10);
  SET v_conn = UTL_SMTP.OPEN_CONNECTION(p_mailhost, v_port, 10);
  CALL UTL_SMTP.HELO(v_conn, p_mailhost);
  CALL UTL_SMTP.MAIL(v_conn, p_sender);
  CALL UTL_SMTP.RCPT(v_conn, p_recipient);
  CALL UTL_SMTP.DATA(
    v_conn,
    'Date: ' || TO_CHAR(SYSDATE, 'Dy, DD Mon YYYY HH24:MI:SS') || v_crlf ||
    'From: ' || p_sender || v_crlf ||
    'To: ' || p_recipient || v_crlf ||
    'Subject: ' || p_subj || v_crlf ||
    p_msg);
  CALL UTL_SMTP.QUIT(v_conn);
END@

CALL send_mail('bwayne@mycorp.com','pparker@mycorp.com','Holiday Party',
'Are you planning to attend?','smtp.mycorp.com')@
```

示例 2: 以下示例使用 OPEN_DATA、WRITE_DATA 和 CLOSE_DATA 过程而不是 DATA 过程。

```
CREATE OR REPLACE PROCEDURE send_mail_2(
IN p_sender VARCHAR(4096),
IN p_recipient VARCHAR(4096),
IN p_subj VARCHAR(4096),
IN p_msg VARCHAR(4096),
IN p_mailhost VARCHAR(4096)) SPECIFIC send_mail_2
LANGUAGE SQL
BEGIN
  DECLARE v_conn UTL_SMTP.CONNECTION;
  DECLARE v_crlf VARCHAR(2);
  DECLARE v_port INTEGER CONSTANT 25;

  SET v_crlf = CHR(13) || CHR(10);
  SET v_conn = UTL_SMTP.OPEN_CONNECTION(p_mailhost, v_port, 10);
  CALL UTL_SMTP.HELO(v_conn, p_mailhost);
  CALL UTL_SMTP.MAIL(v_conn, p_sender);
  CALL UTL_SMTP.RCPT(v_conn, p_recipient);
  CALL UTL_SMTP.OPEN_DATA(v_conn);
  CALL UTL_SMTP.WRITE_DATA(v_conn, 'From: ' || p_sender || v_crlf);
```

```
  CALL UTL_SMTP.WRITE_DATA(v_conn, 'To: ' || p_recipient || v_crlf);
  CALL UTL_SMTP.WRITE_DATA(v_conn, 'Subject: ' || p_subj || v_crlf);
  CALL UTL_SMTP.WRITE_DATA(v_conn, v_crlf || p_msg);
  CALL UTL_SMTP.CLOSE_DATA(v_conn);
  CALL UTL_SMTP.QUIT(v_conn);
END@

CALL send_mail_2('bwayne@mycorp.com','pparker@mycorp.com','Holiday Party',
'Are you planning to attend?','smtp.mycorp.com')@
```

CLOSE_DATA 过程 - 结束电子邮件消息

CLOSE_DATA 过程终止电子邮件消息。

该过程通过发送以下序列来终止电子邮件消息:

```
<CR><LF>.<CR><LF>
```

这是一行开头的一个句点。

语法

参数

c 类型为 CONNECTION 的输入或输出参数，用于指定要关闭的 SMTP 连接。

reply
类型为 REPLY 的可选输出参数，用于从 SMTP 服务器返回单行应答。如果 SMTP 服务器返回多行应答，那么这是最后一行应答。

授权

对 UTL_SMTP 模块的 EXECUTE 特权。

用法说明

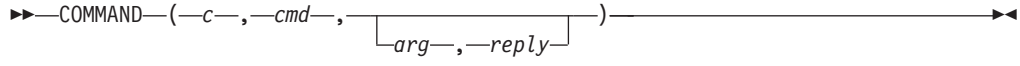
可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

COMMAND 过程 - 运行 SMTP 命令

COMMAND 过程能够执行 SMTP 命令。

注: 如果应该返回多行应答，请使用 COMMAND_REPLIES。

语法



参数

c 类型为 CONNECTION 的输入或输出参数，用于指定命令要发送至的 SMTP 连接。

cmd
类型为 VARCHAR(510) 的输入参数，用于指定要处理的 SMTP 命令。

arg
类型为 VARCHAR(32672) 的可选输入参数，用于指定 SMTP 命令的参数。缺省值为 NULL。

reply
类型为 REPLY 的可选输出参数，用于从 SMTP 服务器返回单行应答。如果 SMTP 服务器返回多行应答，那么这是最后一行应答。

授权

对 UTL_SMTP 模块的 EXECUTE 特权。

用法说明

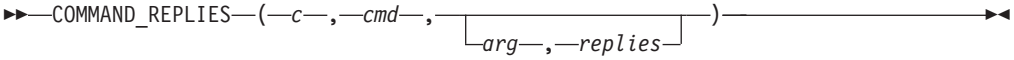
可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

COMMAND_REPLIES 过程 - 在应该出现多行应答的位置运行 SMTP 命令

COMMAND_REPLIES 函数处理返回多行应答的 SMTP 命令。

注: 如果只应出现单行应答，请使用 COMMAND。

语法



参数

c 类型为 CONNECTION 的输入或输出参数，用于指定命令要发送至的 SMTP 连接。

cmd
类型为 VARCHAR(510) 的输入参数，用于指定要处理的 SMTP 命令。

arg
类型为 VARCHAR(32672) 的可选输入参数，用于指定 SMTP 命令的参数。缺省值为 NULL。

replies
类型为 REPLIES 的可选输出参数，用于从 SMTP 服务器返回多行应答。

授权

对 UTL_SMTP 模块的 EXECUTE 特权。

用法说明

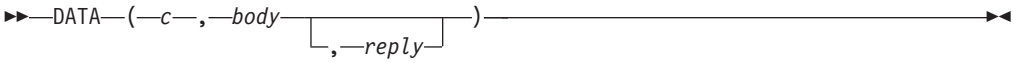
可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

DATA 过程 - 指定电子邮件消息正文

DATA 过程能够指定电子邮件消息正文。

消息以 <CR><LF>.<CR><LF> 序列结尾。

语法



参数

c 类型为 CONNECTION 的输入或输出参数，用于指定命令要发送至的 SMTP 连接。

body
类型为 VARCHAR(32000) 的输入参数，用于指定要发送的电子邮件消息的正文。

reply
类型为 REPLY 的可选输出参数，用于从 SMTP 服务器返回单行应答。如果 SMTP 服务器返回多行应答，那么这是最后一行应答。

授权

对 UTL_SMTP 模块的 EXECUTE 特权。

用法说明

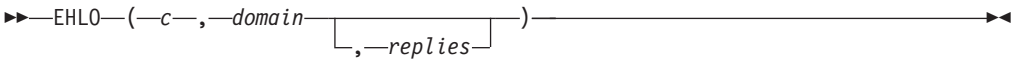
可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

EHLO 过程 - 与 SMTP 服务器进行初始信息交换并返回扩展信息

EHLO 过程在建立连接后与 SMTP 服务器进行初始信息交换

EHLO 过程允许客户机向 SMTP 服务器标识自身。HELO 过程执行等价功能，但返回较少与服务器相关的信息。

语法



参数

c 类型为 CONNECTION 的输入或输出参数，用于指定执行信息交换时使用的与 SMTP 服务器的连接。

domain
类型为 VARCHAR(255) 的输入参数，用于指定发送主机的域名。

replies
类型为 REPLIES 的可选输出参数，用于从 SMTP 服务器返回多行应答。

授权

对 UTL_SMTP 模块的 EXECUTE 特权。

用法说明

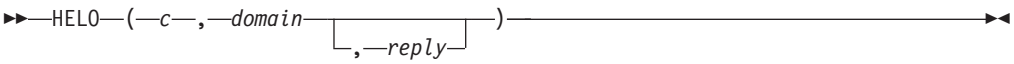
可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

HELO 过程 - 与 SMTP 服务器进行初始信息交换

HELO 过程在建立连接后与 SMTP 服务器进行初始信息交换

HELO 过程允许客户机向 SMTP 服务器标识自身。EHLO 过程执行等价功能，但返回较多与服务器相关的信息。

语法



参数

c 类型为 CONNECTION 的输入或输出参数，用于指定执行信息交换时使用的与 SMTP 服务器的连接。

domain
类型为 VARCHAR(255) 的输入参数，用于指定发送主机的域名。

reply
类型为 REPLY 的可选输出参数，用于从 SMTP 服务器返回单行应答。如果 SMTP 服务器返回多行应答，那么这是最后一行应答。

授权

对 UTL_SMTP 模块的 EXECUTE 特权。

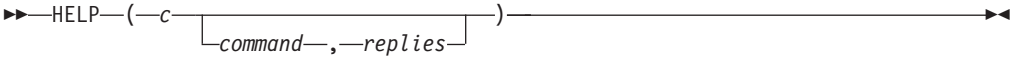
用法说明

可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

HELP 过程 - 发送 HELP 命令

HELP 函数能够将 HELP 命令发送至 SMTP 服务器。

语法



参数

c 类型为 CONNECTION 的输入或输出参数，用于指定命令要发送至的 SMTP 连接。

command
类型为 VARCHAR(510) 的可选输入参数，用于指定请求帮助的命令。

replies
类型为 REPLIES 的可选输出参数，用于从 SMTP 服务器返回多行应答。

授权

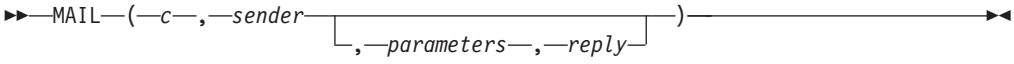
对 UTL_SMTP 模块的 EXECUTE 特权。

用法说明

可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

MAIL 过程 - 启动邮件事务

语法



参数

c 类型为 CONNECTION 的输入或输出参数，用于指定启动邮件事务时使用的与 SMTP 服务器的连接。

sender
类型为 VARCHAR(256) 的输入参数，用于指定发送方的电子邮件地址。

参数
类型为 VARCHAR(32672) 的可选输入参数，用于指定格式为 `key=value` 的可选邮件命令参数。

reply
类型为 REPLY 的可选输出参数，用于从 SMTP 服务器返回单行应答。如果 SMTP 服务器返回多行应答，那么这是最后一行应答。

授权

对 UTL_SMTP 模块的 EXECUTE 特权。

用法说明

可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

NOOP 过程 - 发送空命令

NOOP 过程向 SMTP 服务器发送空命令。除了获取成功响应以外，NOOP 对服务器没有任何影响。

语法

参数

c 类型为 CONNECTION 的输入或输出参数，用于指定发送命令时使用的 SMTP 连接。

reply
类型为 REPLY 的可选输出参数，用于从 SMTP 服务器返回单行应答。如果 SMTP 服务器返回多行应答，那么这是最后一行应答。

授权

对 UTL_SMTP 模块的 EXECUTE 特权。

用法说明

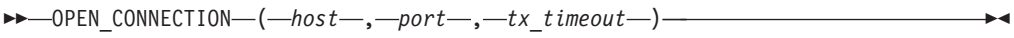
可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

OPEN_CONNECTION 函数 - 返回 SMTP 服务器的连接句柄

OPEN_CONNECTION 函数返回 SMTP 服务器的连接句柄。

该函数将返回 SMTP 服务器的连接句柄。

语法



参数

host
类型为 VARCHAR(255) 的输入参数，用于指定 SMTP 服务器的名称。

port
类型为 INTEGER 的输入参数，用于指定 SMTP 服务器正在侦听的端口号。

tx_timeout
类型为 INTEGER 的输入参数，用于指定超时值（以秒计）。要指示过程不进行等待，请将此值设置为 0。要指示过程无限期等待，请将此值设置为 NULL。

授权

对 UTL_SMTP 模块的 EXECUTE 特权。

OPEN_CONNECTION 过程 - 打开与 SMTP 服务器的连接

OPEN_CONNECTION 过程打开与 SMTP 服务器的连接。

语法

参数

host
类型为 VARCHAR(255) 的输入参数，用于指定 SMTP 服务器的名称。

port
类型为 INTEGER 的输入参数，用于指定 SMTP 服务器正在侦听的端口号。

connection
类型为 CONNECTION 的输出参数，用于返回 SMTP 服务器的连接句柄。

tx_timeout
类型为 INTEGER 的可选输入参数，用于指定超时值（以秒计）。要指示过程不进行等待，请将此值设置为 0。要指示过程无限期等待，请将此值设置为 NULL。

reply
类型为 REPLY 的输出参数，用于从 SMTP 服务器返回单行应答。如果 SMTP 服务器返回多行应答，那么这是最后一行应答。

授权

对 UTL_SMTP 模块的 EXECUTE 特权。

OPEN_DATA 过程 - 将 DATA 命令发送至 SMTP 服务器

OPEN_DATA 过程将 DATA 命令发送至 SMTP 服务器。

语法

参数

c 类型为 CONNECTION 的输入参数，用于指定发送命令时使用的 SMTP 连接

reply
类型为 REPLY 的可选输出参数，用于从 SMTP 服务器返回单行应答。如果 SMTP 服务器返回多行应答，那么这是最后一行应答。

授权

对 UTL_SMTP 模块的 EXECUTE 特权。

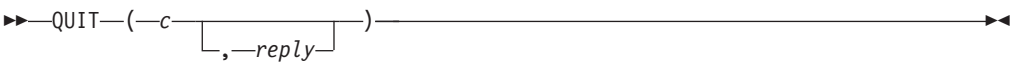
用法说明

可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

QUIT 过程 - 关闭与 SMTP 服务器的会话

QUIT 过程关闭与 SMTP 服务器的会话。

语法



参数

c 类型为 CONNECTION 的输入或输出参数，用于指定要终止的 SMTP 连接。

reply
类型为 REPLY 的可选输出参数，用于从 SMTP 服务器返回单行应答。如果 SMTP 服务器返回多行应答，那么这是最后一行应答。

授权

对 UTL_SMTP 模块的 EXECUTE 特权。

用法说明

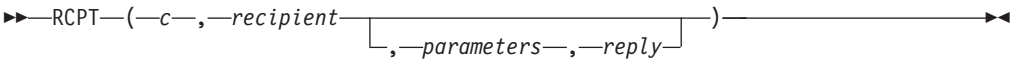
可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

RCPT 过程 - 提供接收方的电子邮件地址

RCPT 过程提供接收方的电子邮件地址。

注: 要调度多个接收方，请多次调用 RCPT 过程。

语法



参数

c 类型为 CONNECTION 的输入或输出参数，用于指定添加接收方时使用的 SMTP 连接。

recipient
类型为 VARCHAR(256) 的输入参数，用于指定接收方的电子邮件地址。

参数
类型为 VARCHAR(32672) 的可选输入参数，用于指定格式为 `key=value` 的邮件命令参数。

reply
类型为 REPLY 的可选输出参数，用于从 SMTP 服务器返回单行应答。如果 SMTP 服务器返回多行应答，那么这是最后一行应答。

授权

对 UTL_SMTP 模块的 EXECUTE 特权。

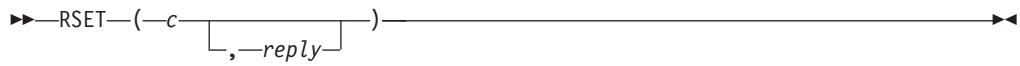
用法说明

可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

RSET 过程 - 结束当前邮件事务

RSET 过程能够终止当前邮件事务。

语法



参数

c 类型为 CONNECTION 的输入或输出参数，用于指定取消邮件事务时使用的 SMTP 连接。

reply
类型为 REPLY 的可选输出参数，用于从 SMTP 服务器返回单行应答。如果 SMTP 服务器返回多行应答，那么这是最后一行应答。

授权

对 UTL_SMTP 模块的 EXECUTE 特权。

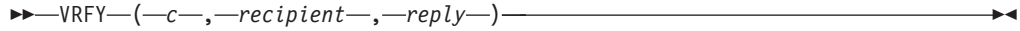
用法说明

可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

VRFY 过程 - 验证接收方的电子邮件地址

VRFY 函数能够验证接收方的电子邮件地址。如果电子邮件地址有效，那么会返回接收方的全名和标准邮箱。

语法



参数

c 类型为 CONNECTION 的输入或输出参数，用于指定验证电子邮件地址时使用的 SMTP 连接。

recipient
类型为 VARCHAR(256) 的输入参数，用于指定要验证的电子邮件地址。

reply
类型为 REPLY 的输出参数，用于从 SMTP 服务器返回单行应答。如果 SMTP 服务器返回多行应答，那么这是最后一行应答。

授权

对 UTL_SMTP 模块的 EXECUTE 特权。

用法说明

可通过在 PL/SQL 赋值语句中使用函数调用语法来调用此过程。

WRITE_DATA 过程 - 写入电子邮件消息的部分

WRITE_DATA 过程能够将数据添加至电子邮件消息。可重复调用 WRITE_DATA 过程以添加数据。

语法

参数

c 类型为 CONNECTION 的输入或输出参数，用于指定添加数据时使用的 SMTP 连接。

data
类型为 VARCHAR(32000) 的输入参数，用于指定要添加至电子邮件消息的数据。

授权

对 UTL_SMTP 模块的 EXECUTE 特权。

WRITE_RAW_DATA 过程 - 将 RAW 数据添加至电子邮件消息

WRITE_RAW_DATA 过程能够将数据添加至电子邮件消息。可重复调用 WRITE_RAW_DATA 过程以添加数据。

语法

参数

c 类型为 CONNECTION 的输入或输出参数，用于指定添加数据时使用的 SMTP 连接。

data
类型为 BLOB(15M) 的输入参数，用于指定要添加至电子邮件消息的数据。

授权

对 UTL_SMTP 模块的 EXECUTE 特权。

第 4 部分 DB2 兼容性功能

第 35 章 DB2 兼容性功能简介

DB2 产品提供了许多功能，它们缩短了允许为关系数据库产品（除 DB2 产品以外）编写的一些应用程序在 DB2 系统上运行的时间，并降低了启用这些应用程序的复杂性。

缺省情况下会启用其中一些功能，包括以下功能。

- 隐式强制类型转换（弱类型），这将减少允许应用程序在 DB2 产品上运行时必须修改的 SQL 语句数。
- 新内置标量函数。有关详细信息，请参阅内置函数（请参阅 *SQL Reference Volume 1*）。
- ITIMESTAMP_FORMAT 和 VARCHAR_FORMAT 标量函数的性能提高。TIMESTAMP_FORMAT 函数使用指定格式返回输入字符串的时间戳记。VARCHAR_FORMAT 函数返回输入表达式的字符串表示，该表示已根据指定的字符模板进行格式化。TO_DATE 和 TO_TIMESTAMP 是 TIMESTAMP_FORMAT 的同义词，而 TO_CHAR 是 VARCHAR_FORMAT 的同义词。
- 撤销了多项 SQL 限制，这使得产品之间的语法更加兼容。例如，在子查询和表函数中使用相关名现在是可选的。
- 其他数据库产品使用的语法的同义词。示例如下所示:
  - 在列函数和查询选择列表中，UNIQUE 是 DISTINCT 的同义词。
  - MINUS 是 EXCEPT 集合运算符的同义词。
  - 可使用 *seqname*.NEXTVAL 来替换 SQL 标准语法 NEXT VALUE FOR *seqname*。还可使用 *seqname*.CURRVAL 来替换 SQL 标准语法 PREVIOUS VALUE FOR *seqname*。
- 全局变量，可用于轻松映射程序包变量、模拟 @@nested、@level 或 @errorlevel 全局变量或将信息从 DB2 应用程序传递至触发器、函数或过程。
- ARRAY 集合数据类型，可用于轻松映射至 SQL 过程中的 VARRAY 构造。
- 增加的标识长度限制。
- 伪列 ROWID，可用于引用 RID。未限定 ROWID 引用等价于 RID_BIT()，限定 ROWID 引用（例如，EMPLOYEE.ROWID）等价于 RID_BIT(EMPLOYEE)。

可选择通过设置 **DB2_COMPATIBILITY_VECTOR** 注册表变量来启用以下其他功能。缺省情况下，这些功能处于禁用状态。

- 使用 CONNECT BY PRIOR 语法的分层查询的实现。
- 对使用外连接运算符（即，加号 (+)）的外连接的支持。
- 将 DATE 数据类型用作 TIMESTAMP(0)，即组合日期和时间值。
- 支持 NUMBER 数据类型的语法和语义。
- 支持 VARCHAR2 数据类型的语法和语义。
- ROWNUM 伪列，它是 ROW_NUMBER() OVER() 的同义词。但是，允许在 SELECT 列表中和 SELECT 语句的 WHERE 子句中使用 ROWNUM 伪列。
- 名为 DUAL 的哑表，用于提供 SYSIBM.SYSDUMMY1 表的类似功能。

- TRUNCATE 语句的备用语义，在此情况下，IMMEDIATE 是可选关键字，如果未指定任何关键字，那么会假定它是缺省关键字。如果 TRUNCATE 语句不是逻辑工作单元中的第一条语句，那么将在执行 TRUNCATE 语句前执行隐式的落实操作。
- 支持将 CHAR 或 GRAPHIC 数据类型（而不是 VARCHAR 或 VARGRAPHIC 数据类型）指定给字符和图形字符串常量（其字节长度小于或等于 254）。
- 使用集合方法对数组执行操作，例如 FIRST、LAST、NEXT 和 PREVIOUS。
- 支持创建 Oracle 数据字典兼容视图。
- 支持编译和执行 PL/SQL 语句及其他语言元素。
- 支持使游标对后续语句不敏感（通过在打开时将游标具体化）。
- 支持在过程中使用 INOUT 参数，您使用缺省值定义这些参数，并且可在不对其指定自变量的情况下调用这些参数。

其他资源

有关兼容性功能的更多信息，请参阅 DB2 Viper 2 兼容性功能。

有关 IBM Migration Toolkit (MTK) 的信息，请参阅立即迁移！。

有关 DB2 Oracle 数据库兼容性功能的信息，请参阅 Oracle to DB2 Conversion Guide: Compatibility Made Easy。

第 36 章 DB2_COMPATIBILITY_VECTOR 注册表变量

DB2_COMPATIBILITY_VECTOR 注册表变量启用一个或多个 DB2 兼容性功能。这些功能使您可轻松完成将为 DB2 产品 产品之外的关系数据库产品编写的应用程序迁移至 DB2 V9.5 或更高版本的任务。

此注册表变量在 Linux、UNIX 和 Windows 操作系统上受支持。

可通过对该注册表变量指定十六进制值来启用个别 DB2 兼容性功能。变量值中的每一位启用不同功能。值如下所示:

- NULL（缺省值）
- 0000 - FFFF
- ORA，对 Oracle 应用程序充分利用 DB2 兼容性功能
- SYB，对 Sybase 应用程序充分利用 DB2 兼容性功能
- MYS，对 MySQL 应用程序充分利用 DB2 兼容性功能部件

建议将值设置为 ORA、SYB 或 MYS。

要点: 仅当需要将这些功能用于特定兼容性用途时，才启用这些功能。如果启用 DB2 兼容性功能，那么 SQL 引用信息中描述的一些 SQL 行为会更改。要确定 SQL 应用程序的兼容性功能的潜在作用，请参阅与该兼容性功能相关联的文档。

注册表变量设置

下表指定启用每个兼容性功能所需的设置。

表 36. **DB2_COMPATIBILITY_VECTOR** *注册表变量值*

位位置（十六进制值）	兼容性功能	描述
1（0x01）	ROWNUM 伪列	允许使用 ROWNUM 作为 ROW_NUMBER() OVER() 的同义词，并允许 ROWNUM 在 SQL 语句的 WHERE 子句中出现。
2（0x02）	DUAL 表	将对 DUAL 表进行的未限定引用解析为 SYSIBM.DUAL。
3 (0x04)	外连接运算符	启用对外连接运算符（即，加号 (+)）的支持。
4 (0x08)	分层查询	启用对使用 CONNECT BY 子句进行的分层查询的支持。
5 (0x10)	NUMBER 数据类型 [1]	启用对 NUMBER 数据类型和关联数字处理的支持。
6 (0x20)	VARCHAR2 数据类型 [1]	启用对 VARCHAR2 和 NVARCHAR2 数据类型及关联字符串处理的支持。

表 36. **DB2_COMPATIBILITY_VECTOR** *注册表变量值 (续)*

位位置（十六进制值）	兼容性功能	描述
7 (0x40)	DATE 数据类型 [1]	允许将 DATE 数据类型解释为 TIMESTAMP(0) 数据类型，即组合日期和时间值。例如，日期兼容性方式的“VALUES CURRENT DATE”返回 2011-02-17-10.43.55 之类的值。
8 (0x80)	TRUNCATE TABLE	启用 TRUNCATE 语句的备用语义，在此情况下，IMMEDIATE 是可选关键字，如果未指定任何关键字，那么会假定它是缺省关键字。如果 TRUNCATE 语句不是逻辑工作单元中的第一条语句，那么将在执行 TRUNCATE 语句前执行隐式的落实操作。
9 (0x100)	字符文字	允许将 CHAR 或 GRAPHIC 数据类型（而不是 VARCHAR 或 VARGRAPHIC 数据类型）指定给字节长度小于或等于 254 的字符和图形字符串常量。
10 (0x200)	集合方法	允许使用方法对数组执行操作，例如 first、last、next 和 previous。并且，允许在对数组中的特定元素的引用中使用圆括号来代替方括号；例如，array1(*i*) 引用 array1 的元素 *i*。
11 (0x400)	Oracle 数据字典兼容视图 [1]	允许创建 Oracle 数据字典兼容视图。
12 (0x800)	PL/SQL 编译 [2]	允许编译和执行 PL/SQL 语句和语言元素。
13 (0x1000)	不敏感游标	在 SELECT 语句未显式指定 FOR UPDATE 的情况下，使定义为 WITH RETURN 的游标不敏感。
14 (0x2000)	第 443 页的第 47 章，『INOUT 参数』	允许对 INOUT 参数声明指定 DEFAULT。
17 (0x10000)	第 421 页的第 40 章，『SQL 数据访问级别强制』	允许例程强制在运行时实施 SQL 数据访问级别。

1. 此功能仅在创建数据库期间适用。在创建数据库后启用或禁用此功能仅影响后续创建的数据库。
2. 请参阅『对 PL/SQL 支持的限制』。

用法

请使用 **db2set** 命令来设置和更新 **DB2_COMPATIBILITY_VECTOR** 注册表变量。

- 要启用所有受支持的 Oracle 兼容性功能，请将该注册表变量设置为值 ORA（相当于十六进制值 10FFF）。
- 要启用所有受支持的 Sybase 兼容性功能，请将此注册表变量设置为值 SYB（相当于十六进制值 3004）。

注册表变量的新设置直到您停止然后重新启动实例后才会生效。而且，必须重新绑定 DB2 程序包，更改才能生效。您未显式重新绑定的程序包将在下一次隐式重新绑定时实现更改。

如果设置 **DB2_COMPATIBILITY_VECTOR** 注册表变量，请创建数据库作为 Unicode 数据库。

示例 1

此示例说明如何设置该注册表变量以启用所有受支持的 Oracle 兼容性功能:

```
db2set DB2_COMPATIBILITY_VECTOR=ORA
db2stop
db2start
```

示例 2

此示例说明如何设置该注册表变量以启用 ROWNUM (0x01) 和 DUAL (0x02):

```
db2set DB2_COMPATIBILITY_VECTOR=03
db2stop
db2start
```

示例 3

此示例说明如何通过重置 **DB2_COMPATIBILITY_VECTOR** 注册表变量来禁用所有兼容性功能:

```
db2set DB2_COMPATIBILITY_VECTOR=
db2stop
db2start
```

如果在启用下列任何功能后创建数据库，那么在执行此 **db2set** 命令后仍会对此数据库启用这些功能:

- NUMBER 数据类型
- VARCHAR2 数据类型
- 将 DATE 数据类型作为 TIMESTAMP(0)
- 创建 Oracle 数据字典兼容视图

第 37 章 设置 DB2 环境以便启用 Oracle 应用程序

如果适当设置 DB2 环境，那么可缩短启用 Oracle 应用程序以与 DB2 数据服务器配合工作的时间并降低启用这些应用程序的复杂性。

开始之前

- 必须安装 DB2 数据服务器产品。
- 您需要 SYSADM 和适当操作系统权限才能发出 **db2set** 命令。
- 您需要 SYSADM 或 SYSCTRL 权限才能发出 **CREATE DATABASE** 命令。

关于此任务

DB2 产品可支持来自其他数据库产品的许多通常引用的功能。要执行从 DB2 接口中引用 Oracle 数据类型的 PL/SQL 语句或 SQL 语句，或者要使用任何其他 SQL 兼容性功能，此任务是先决条件。在数据库级别启用 DB2 兼容性功能；您不能禁用这些功能。

过程

要启用 Oracle 应用程序以与 DB2 数据服务器配合工作，请执行以下操作:

1. 在 DB2 命令窗口中，通过发出以下命令来启动 DB2 数据库管理器:

   ```
   db2start
   ```

2. 将 **DB2_COMPATIBILITY_VECTOR** 注册表变量设置为下列其中一个值:
   - 一个启用您要使用的特定兼容性功能的十六进制值。
   - 要利用所有 DB2 兼容性功能，此值为 ORA，如以下命令中所示：这是建议的设置。

   ```
   db2set DB2_COMPATIBILITY_VECTOR=ORA
   ```

3. 通过将 **DB2_DEFERRED_PREPARE_SEMANTICS** 注册表变量设置为 YES 来启用延迟准备支持，如下所示:

   ```
   db2set DB2_DEFERRED_PREPARE_SEMANTICS=YES
   ```

   如果将 **DB2_COMPATIBILITY_VECTOR** 注册表变量设置为 ORA 并且未设置 **DB2_DEFERRED_PREPARE_SEMANTICS** 注册表变量，那么会使用缺省值 YES。但是，建议将 **DB2_DEFERRED_PREPARE_SEMANTICS** 注册表变量显式设置为 YES。

4. 通过发出 **db2stop** 命令来停止数据库管理器:

   ```
   db2stop
   ```

5. 通过发出 **db2start** 命令来启动数据库管理器:

   ```
   db2start
   ```

6. 通过发出 **CREATE DATABASE** 命令来创建 DB2 数据库。将该数据库创建为 Unicode 数据库。缺省情况下，数据库被创建为 Unicode 数据库。 例如，要创建名为 DB 的数据库，请发出以下命令:

   ```
   db2 CREATE DATABASE DB
   ```

7. 可选: 运行 Command Line Processor Plus (CLPPlus) 或命令行处理器 (CLP) 脚本（例如，script.sql）以验证数据库是否支持 PL/SQL 语句和数据类型。 以下 CLPPlus 脚本创建然后调用简单过程:

```
CONNECT user@hostname:port/dbname;

CREATE TABLE t1 (c1 NUMBER);

CREATE OR REPLACE PROCEDURE testdb(num IN NUMBER, message OUT VARCHAR2)
AS
BEGIN
 INSERT INTO t1 VALUES (num);

 message := 'The number you passed is: ' || TO_CHAR(num);
END;
/

CALL testdb(100, ?);

DISCONNECT;
EXIT;
```

要运行此 CLPPlus 脚本，请发出以下命令:

```
clpplus @script.sql
```

以下示例演示同一个脚本的 CLP 版本。此脚本使用了 **SET SQLCOMPAT PLSQL** 命令，以便将新行中的正斜杠字符（/）识别为 PL/SQL 语句终止符。

```
CONNECT TO DB;

SET SQLCOMPAT PLSQL;

-- Semicolon is used to terminate
-- the CREATE TABLE statement:
CREATE TABLE t1 (c1 NUMBER);

-- Forward slash on a new line is used to terminate
-- the CREATE PROCEDURE statement:
CREATE OR REPLACE PROCEDURE testdb(num IN NUMBER, message OUT VARCHAR2)
AS
BEGIN
 INSERT INTO t1 VALUES (num);

 message := 'The number you passed is: ' || TO_CHAR(num);
END;
/

CALL testdb(100, ?);

SET SQLCOMPAT DB2;

CONNECT RESET;
```

要运行此 CLP 脚本，请发出以下命令:

```
db2 -tvf script.sql
```

结果

此创建的 DB2 数据库支持 Oracle 应用程序。现在可使用您启用的兼容性功能。对于 Oracle 应用程序，仅启用了在设置 **DB2_COMPATIBILITY_VECTOR** 注册表变量之后创建的数据库。

下一步做什么

- 开始使用 CLPPlus 接口。
- 执行 PL/SQL 脚本和语句。

- 传送数据库对象定义。
- 启用数据库应用程序。

第 38 章 数据类型

基于 TIMESTAMP(0) 的 DATE 数据类型

DATE 数据类型支持使用 Oracle DATE 数据类型的应用程序并期望 DATE 值包括时间信息（例如，“2009-04-01-09.43.05”）。

启用

在创建需要“将 DATE 作为 TIMESTAMP(0)”支持的数据库之前，在数据库级别启用该支持。要启用此支持，请将 **DB2_COMPATIBILITY_VECTOR** 注册表变量设置为十六进制值 0x40（位位置 7），停止然后重新启动该实例以使新设置生效。

```
db2set DB2_COMPATIBILITY_VECTOR=40
db2stop
db2start
```

要充分利用 Oracle 应用程序的 DB2 兼容性功能部件，DB2_COMPATIBILITY_VECTOR 的推荐设置是 ORA，这将设置所有兼容性位。

创建启用了“将 DATE 作为 TIMESTAMP(0)”支持的数据库后，**date_compat** 数据库配置参数设置为 ON。

如果创建启用了“将 DATE 作为 TIMESTAMP(0)”支持的数据库，那么不能对该数据库禁用该支持，即使您重置 **DB2_COMPATIBILITY_VECTOR** 注册表变量也是如此。同样，如果创建禁用了“将 DATE 作为 TIMESTAMP(0)”支持的数据库，那么以后不能对该数据库启用该支持，即使通过设置 **DB2_COMPATIBILITY_VECTOR** 注册表变量也是如此。

作用

date_compat 数据库配置参数指示是否对相连数据库应用了与 TIMESTAMP(0) 数据类型相关联的 DATE 兼容性语义。将 **date_compat** 设置为 ON 的作用如下所示。

在 SQL 语句中显式遇到 DATE 数据类型后，在大多数情况下，该数据类型隐式映射至 TIMESTAMP(0)。在 CREATE INDEX 语句的 *XML 索引规范* 子句中指定 SQL DATE 是以上规则的例外。此隐式映射的结果是，消息将引用数据类型 TIMESTAMP 而不是 DATE，并且，任何描述列数据类型或例程数据类型的操作都将返回 TIMESTAMP 而不是 DATE。

日期时间字面值支持更改为如下所示:

- 显式 DATE 字面值的值为时间部分全部为零的 TIMESTAMP(0) 值。例如，DATE '2008-04-28' 表示时间戳记值“2008-04-28-00.00.00”。
- 数据库管理器支持对日期的字符串表示使用其他格式，这些格式对应于“DD-MON-YYYY”和“DD-MON-RR”。仅支持月份的英语缩写词。有关格式元素的描述，请参阅 TIMESTAMP_FORMAT 标量函数（请参阅《*SQL Reference Volume 1*》）。例如，可以使用“28-APR-2008”或“28-APR-08”作为日期的字符串表示，这实际上表示 TIMESTAMP(0) 值“2008-04-28-00.00.00”。

CURRENT_DATE（又称为 CURRENT DATE）专用寄存器返回与 CURRENT_TIMESTAMP(0) 值相同的 TIMESTAMP(0) 值。

对 TIMESTAMP 值加一个数字值或者从 TIMESTAMP 值中减一个数字时，将假定该数字值表示天数。此数字值可以具有任何数字数据类型，任何小数值都将被视为天的小数部分。例如，`TIMESTAMP '2008-03-28 12:00:00' + 1.3` 将对 TIMESTAMP 值加 1 天又 7 小时 12 分，因此结果为“2008-03-29 19:12:00”。如果使用表达式来表示不足一天的时间，例如 1/24（1 小时）或 1/24/60（1 分钟），请确保 **number_compat** 数据库配置参数设置为 `ON`，以便使用 DECFLOAT 算术来进行除法运算。

某些函数的结果会更改:

- 如果将字符串自变量传递至 ADD_MONTHS 标量函数，那么它返回 TIMESTAMP(0) 值。
- DATE 标量函数对所有输入类型返回 TIMESTAMP(0) 值。
- 如果将字符串自变量传递至 LAST_DAY 标量函数，那么它返回 TIMESTAMP(0) 值。
- 如果将 DATE() 自变量传递至 ADD_MONTHS、LAST_DAY、NEXT_DAY、ROUND 或 TRUNCATE 标量函数，那么该函数返回 TIMESTAMP(0) 值。
- 将两个日期值相加返回 TIMESTAMP(0) 值。
- 将两个时间戳记值相减返回 DECFLOAT(34)，表示时差（以天数表示）。同样，将两个日期值相减返回 DECFLOAT(34)，表示天数。
- TIMESTAMPDIFF 标量函数中的第二个参数不表示时间戳记持续时间。而是表示两个时间戳记之差（以天数表示）。返回的估算值可能因天数不同而变化。例如，如果求“2010-03-31-00.00.00.000000”与“2010-03-01-00.00.00.000000”之差的月数（时间间隔 64），那么结果为 1。这是因为两个时间戳记之差为 30 天，并且假定 1 个月为 30 天。下表显示如何针对每个时间间隔确定返回值。

表 37. TIMESTAMPDIFF 计算

结果时间间隔	使用两个时间戳记之差（以天数表示）进行计算
年数	(天数/365) 的整数值
季度数	(天数/90) 的整数值
月数	(天数/30) 的整数值
周数	(天数/7) 的整数值
天数	天数的整数值
小时数	(天数*24) 的整数值
分钟数（天数的绝对值不得超过 1491308.0888888888888882）	(天数*24*60) 的整数值
秒数（天数的绝对值必须小于 24855.1348148148148148）	(天数*24*60*60) 的整数值
微秒数（天数的绝对值必须小于 0.02485513481481481）	(天数*24*60*60*1000000) 的整数值

如果使用导入或装入实用程序将数据输入到 DATE 列中，那么必须使用 timestampformat 文件类型修饰符而不是 dateformat 文件类型修饰符。

NUMBER 数据类型

NUMBER 数据类型支持使用 Oracle NUMBER 数据类型的应用程序。

启用

在创建需要 NUMBER 支持的数据库之前，在数据库级别启用该支持。要启用该支持，请将 **DB2_COMPATIBILITY_VECTOR** 注册表变量设置为十六进制值 0x10（位位置 5），然后停止并重新启动该实例以使新设置生效。

```
db2set DB2_COMPATIBILITY_VECTOR=10
db2stop
db2start
```

要充分利用 Oracle 应用程序的 DB2 兼容性功能部件，DB2_COMPATIBILITY_VECTOR 的推荐设置是 ORA，这将设置所有兼容性位。

创建启用了 NUMBER 支持的数据库之后，将 **number_compat** 数据库配置参数设置为 ON。

如果创建启用了 NUMBER 支持的数据库，那么无法禁用该数据库的 NUMBER 支持，即使重置 **DB2_COMPATIBILITY_VECTOR** 注册表变量也是如此。同样，如果创建启用了 NUMBER 支持的数据库，那么以后无法启用该数据库的 NUMBER 支持，即使通过设置 **DB2_COMPATIBILITY_VECTOR** 注册表变量也是如此。

作用

将 **number_compat** 数据库配置参数设置为 ON 的效果如下所示。

当在 SQL 语句中显式遇到了 NUMBER 数据类型时，将按以下方式对该数据类型进行隐式映射:

- 如果指定 NUMBER 时未指定精度和小数位属性，那么它将映射到 DECFLOAT(16)。
- 如果指定 NUMBER(*p*)，那么它将映射到 DECIMAL(*p*)。
- 如果指定 NUMBER(*p*,*s*)，那么它将映射到 DECIMAL(*p*,*s*)。

支持的最大精度是 31，并且，小数位数必须是不大于精度的正数值。另外，该隐式映射的结果是，消息引用数据类型 DECFLOAT 和 DECIMAL，而不是 NUMBER。另外，任何描述列数据类型或例程数据类型的操作都将返回 DECIMAL 或 DECFLOAT，而不是 NUMBER。

提示: DECFLOAT(16) 数据类型提供的最大精度低于 Oracle NUMBER 数据类型。如果在列中存储数字需要 16 位以上的精度，那么将这些列显式定义为 DECFLOAT(34)。

数字文字支持未更改: 整数、十进制和浮点常量的规则仍适用。这些规则将十进制文字限制为 31 位，并将浮点文字限制为二进制双精度浮点值的范围。如果有必要，对于超出 DECIMAL 或 DOUBLE 的范围并在 DECFLOAT(34) 范围内的值，可以使用字符串到 DECFLOAT(34) 的强制类型转换（使用 CAST 规范或 DECFLOAT 函数）。目前，既不支持以 D 结尾的数字文字（它表示 64 位二进制浮点值），也不支持以 F 结尾的数字文字（它表示 32 位二进制浮点值）。包含 E 的数字文字的数据类型为 DOUBLE，您可以使用 CAST 规范或使用强制类型转换函数 REAL 将其强制转换为 REAL

如果通过使用 CAST 规范或者使用 VARCHAR 或 CHAR 标量函数将 NUMBER 数据值强制转换为字符串，那么将从结果中除去所有前导零。

用于 CREATE SEQUENCE 语句中的序列值的缺省数据类型是 DECIMAL(27) 而不是 INTEGER。

所有涉及 DECIMAL 或 DECFLOAT 数据类型的算术运算以及算术或数学函数实际都使用十进制浮点算术进行运算，并返回数据类型为 DECFLOAT(34) 的值。此类型的性能还适用于两个操作数都是 DECIMAL 或 DECFLOAT(16) 数据类型的算术运算，这与《》表达式部分中的『算术运算符的表达式』（请参阅 *SQL Reference Volume 1*）中有关十进制算术的描述有所不同。另外，所有只涉及整数数据类型（SMALLINT、INTEGER 或 BIGINT）的除法运算实际上都使用十进制浮点算术执行。这些操作返回数据类型为 DECFLOAT(34) 的值，而不是整数数据类型。用整数除以零的运算将返回无穷大和警告，而不是返回错误。

在某些情况下，函数解析也有所更改，例如，数据类型为 DECIMAL 的自变量在解析过程中将被视为 DECFLOAT 值。另外，这实际上是将自变量与 NUMBER(*p*[,*s*]) 数据类型相对应的函数看成自变量数据类型为 NUMBER。但是，函数解析方面的此更改并不适用于那些自变量数目可变并且结果数据类型基于自变量数据类型集的函数集。此集合中包括的函数如下所示:

- COALESCE
- DECODE
- GREATEST
- LEAST
- MAX（标量）
- MIN（标量）
- NVL
- VALUE

结果数据类型的规则（请参阅 *SQL Reference Volume 1*）扩展为: 如果 DECIMAL 结果数据类型的精度将超出 31，那么使 DECFLOAT(34) 成为结果数据类型。这些结果还适用于以下各项:

- 集合运算中的相应列: UNION、EXCEPT(MINUS) 和 INTERSECT
- IN 谓词的 IN 列表中的表达式值
- 多行 VALUES 子句的相应表达式

赋值和强制类型转换所使用的舍入方式取决于所涉及的数据类型。在某些情况下，将进行截断。在目标是二进制浮点（REAL 或 DOUBLE）值的情况下，将按惯例使用 round-half-even。在其他情况下，通常涉及 DECIMAL 或 DECFLOAT 值，舍入基于 **decflt_rounding** 数据库配置参数的值进行。此参数缺省值为 round-half-even，但可以设置为 round-half-up 以便与 Oracle 舍入方式匹配。下表摘要说明用于各种数字赋值和强制类型转换的舍入运算。

表 38. 用于数字赋值和强制类型转换的舍入运算

	目标数据类型			
源数据类型	**整数类型**	**DECIMAL**	**DECFLOAT**	**REAL/DOUBLE**
整数类型	不适用	不适用	decflt_rounding	round_half_even

表 38. 用于数字赋值和强制类型转换的舍入运算 (续)

	目标数据类型			
源数据类型	**整数类型**	**DECIMAL**	**DECFLOAT**	**REAL/DOUBLE**
DECIMAL	decflt_rounding	decflt_rounding	decflt_rounding	round_half_even
DECFLOAT	decflt_rounding	decflt_rounding	decflt_rounding	round_half_even
REAL/DOUBLE	截断	decflt_rounding	decflt_rounding	round_half_even
字符串（仅适用于强制类型转换）	不适用	decflt_rounding	decflt_rounding	round_half_even

DB2 十进制浮点值基于 IEEE 754R 标准。在检索 DECFLOAT 数据以及将 DECFLOAT 数据强制类型转换为字符串时，将除去小数点后的任何尾部零。

客户机/服务器兼容性

如果已对 DB2 数据库服务器启用 NUMBER 数据类型支持，那么与该服务器配合使用的客户机应用程序将永远不会从该服务器接收到 NUMBER 数据类型。在 DB2 服务器上，任何要从 Oracle 服务器上报告 NUMBER 的列或表达式都会报告 DECIMAL 或 DECFLOAT。

由于 Oracle 环境期望舍入方式为 round-half-up，因此客户机舍入方式与服务器舍入方式匹配至关重要。这意味着，`db2cli.ini` 文件设置必须与 **`decflt_rounding`** 数据库配置参数的值匹配。要最接近匹配 Oracle 舍入方式，应该为数据库配置参数指定 `ROUND_HALF_UP`。

限制

NUMBER 数据类型支持具有下列限制:

- 不支持以下各项:
  - 精度属性大于 31
  - 带星号的精度属性 (*)
  - 超过精度属性的小数位属性
  - 负小数位属性

  没有用于 NUMBER 类数据类型规范的相应 DECIMAL 精度和小数位数支持。
- 无法使用数据类型为没有精度的 NUMBER (DECFLOAT) 的自变量调用三角函数或 DIGITS 标量函数。
- 无法创建名为 NUMBER 的单值类型。

VARCHAR2 和 NVARCHAR2 数据类型

VARCHAR2 和 NVARCHAR2 数据类型支持使用 Oracle VARCHAR2 和 NVARCHAR2 数据类型的应用程序。

启用

在创建需要 VARCHAR2 和 NVARCHAR2（后来统称为 VARCHAR2）支持的数据库之前，在数据库级别启用该支持。 要启用此支持，请将 **DB2_COMPATIBILITY_VECTOR** 注册表变量设置为十六进制值 0x20（位位置 6），然后停止并重新启动该实例以使新设置生效。

```
db2set DB2_COMPATIBILITY_VECTOR=20
db2stop
db2start
```

要充分利用 Oracle 应用程序的 DB2 兼容性功能部件，DB2_COMPATIBILITY_VECTOR 的推荐设置是 ORA，这将设置所有兼容性位。

创建启用了 VARCHAR2 支持的数据库之后，将 **varchar2_compat** 数据库配置参数设置为 ON。

如果创建启用了 VARCHAR2 支持的数据库，那么无法禁用该数据库的 VARCHAR2 支持，即使重置 **DB2_COMPATIBILITY_VECTOR** 注册表变量也是如此。同样，如果创建启用了 VARCHAR2 支持的数据库，那么以后无法启用该数据库的 VARCHAR2 支持，即使通过设置 **DB2_COMPATIBILITY_VECTOR** 注册表变量也是如此。

要使用 NVARCHAR2 数据类型，数据库必须是 Unicode 数据库。

作用

将 **varchar2_compat** 数据库配置参数设置为 ON 的效果如下所示。

当 SQL 语句明确遇到了 VARCHAR2 数据类型时，此数据类型将以隐式方式映射到 VARCHAR 数据类型。VARCHAR2 的最大长度与 VARCHAR 的最大长度相同，即: 32, 672。同样，当 SQL 语句显式遇到了 NVARCHAR2 数据类型时，此数据类型将隐式映射至 VARGRAPHIC 数据类型。NVARCHAR2 的最大长度与 VARGRAPHIC 的最大长度相同，即: 16, 336。

长度不超过 254 个字节的字符串文字的数据类型为 CHAR。长度超过 254 个字节的字符串文字的数据类型为 VARCHAR。

涉及可变长度字符串类型的比较操作都使用不填充比较语义，而只涉及固定长度字符串类型的比较操作将继续使用空格填充比较语义，但有两种例外情况:

- 如果比较操作涉及目录视图中的字符串列信息，那么将始终使用具有空格填充比较语义的 IDENTITY 整理规则，而不考虑数据库整理规则。
- 如果字符串比较操作涉及具有 FOR BIT DATA 属性的数据类型，那么将始终使用具有空格填充比较语义的 IDENTITY 整理规则。

如果以下两个条件都为 true，那么 IN 列表表达式被视为具有变长字符串数据类型:

- IN 谓词的 IN 列表的结果类型将解析为固定长度字符串数据类型
- IN 谓词的左操作数是变长字符串数据类型

通常，长度为零的字符串值（LOB 值除外）被视为 null 值。对 CHAR、NCHAR、VARCHAR 或 NVARCHAR 指定空字符串值或者将空字符串值强制类型转换为这些数据类型，都将生成 NULL 值。

对于那些将返回字符串自变量或者基于具有字符串数据类型的参数的函数来说，它们也将空字符串 CHAR、NCHAR、VARCHAR 或 NVARCHAR 值视为 NULL 值。 将 **varchar2_compat** 数据库配置参数设置为 ON 时，注意某些函数适用的特殊注意事项，如下所示:

- CONCAT 函数和并置运算符。在并置结果中，null 值或空字符串值将被忽略。下表列示了并置结果类型。

表 39. 并置的操作数的数据类型和长度

操作数	组合的长度属性	结果
CHAR(A) CHAR(B)	<255	CHAR(A+B)
CHAR(A) CHAR(B)	>254	VARCHAR(A+B)
CHAR(A) VARCHAR(B)	-	VARCHAR(MIN(A+B,32672))
VARCHAR(A) VARCHAR(B)	-	VARCHAR(MIN(A+B,32672))
CLOB(A) CHAR(B)	-	CLOB(MIN(A+B, 2G))
CLOB(A) VARCHAR(B)	-	CLOB(MIN(A+B, 2G))
CLOB(A) CLOB(B)		CLOB(MIN(A+B, 2G))
GRAPHIC(A) GRAPHIC(B)	<128	GRAPHIC(A+B)
GRAPHIC(A) GRAPHIC(B)	>128	VARGRAPHIC(A+B)
GRAPHIC(A) VARGRAPHIC(B)	-	VARGRAPHIC(MIN(A+B,16336))
VARGRAPHIC(A) VARGRAPHIC(B)	-	VARGRAPHIC(MIN(A+B,16336))
DBCLOB(A) CHAR(B)	-	DBCLOB(MIN(A+B, 1G))
DBCLOB(A) VARCHAR(B)	-	DBCLOB(MIN(A+B, 1G))
DBCLOB(A) CLOB(B)		DBCLOB(MIN(A+B, 1G))
BLOB(A) BLOB(B)	-	BLOB(MIN(A+B, 2G))

- INSERT 函数。如果第四个自变量是 null 值或者空字符串，那么将致使从第一个自变量中由第二个自变量指示的字节位置开始删除由第三个自变量指示的字节数。
- LENGTH 函数。LENGTH 函数返回的值是字符串中的字节数。空字符串值将返回 null 值。
- REPLACE 函数。如果所有自变量值的数据类型都是 CHAR、VARCHAR、GRAPHIC 或 VARGRAPHIC，那么:
  - 作为第二个自变量的 null 值或空字符串将被视为空字符串，其结果是，将第一个自变量作为结果返回。
  - 作为第三个自变量的 null 值或空字符串将被视为空字符串，其结果是，没有任何内容替换第二个自变量从源字符串中除去的字符串。

  如果任何自变量值的数据类型为 CLOB 或 BLOB，并且任何自变量是 null 值，那么结果为 null 值。您必须指定 REPLACE 函数的全部三个自变量。
- SUBSTR 函数。对第一个自变量使用字符串输入的 SUBSTR 引用将被替换为 SUBSTRB 调用。对第一个自变量使用国家字符集（图形）字符串输入的 SUBSTR 引用将被替换为 SUBSTR2 调用。
- TRANSLATE 函数。*源字符串表达式* 是第二个自变量，*目标字符串表达式* 是第三个自变量。如果 *目标字符串表达式* 比 *源字符串表达式* 短，那么将除去 *源字符串表*

达式 中的那些能够在*字符串表达式*（第一个自变量）中找到的额外字符；即，除非在第四个自变量中指定另一个填充字符，否则缺省的 *填充字符* 自变量实际上是空字符串。

- TRIM 函数。如果 TRIM 函数调用中的除去字符自变量是 null 值或空字符串，那么此函数将返回 null 值。

在 ALTER TABLE 语句或 CREATE TABLE 语句中，如果指定了 DEFAULT 子句，但没有为数据类型已定义为 VARCHAR 或 VARGRAPHIC 的列指定显式的值，那么缺省值是空白字符。

当数据库配置参数 **varchar2_compat** 将设置为 ON 时，空字符串将转换为空白字符。例如:

- 数据分区可视时，SYSCAT.DATAPARTITIONS.STATUS 包含单一空白字符。
- 未显式设置程序包版本时，SYSCAT.PACKAGES.PKGVERSION 包含单一空白字符。
- 未设置编译选项时，SYSCAT.ROUTINES.COMPILE_OPTIONS 包含 null 值。

如果 SQL 语句使用了参数标记，那么可能会发生将会影响到 VARCHAR2 的使用的数据类型转换。例如，如果输入值是长度为零的 VARCHAR，并且它被转换为 LOB，那么结果将是 null 值。但是，如果输入值是长度为零的 LOB，并且它被转换为 LOB，那么结果将是长度为零的 LOB。输入值的数据类型可能会受延迟准备影响。

限制

VARCHAR2 数据类型以及相关联的字符串处理支持具有下列限制:

- 不接受 VARCHAR2 长度属性限定符 CHAR。
- 当 **varchar2_compat** 数据库配置参数设置为 ON 时，不支持 LONG VARCHAR 和 LONG VARGRAPHIC 数据类型（但不会显式地将其禁用）。

第 39 章 字符和图形常量处理

隐式强制类型转换（或弱类型）是一种备用方法，可解析字符或图形常量以支持期望将这些常量指定给数据类型为 CHAR 和 GRAPHIC 的应用程序。

启用

要启用字符和图形常量处理支持，请将 **DB2_COMPATIBILITY_VECTOR** 注册表变量设置为十六进制值 c0x100（位位置 9），然后停止并重新启动该实例以使新设置生效。

```
db2set DB2_COMPATIBILITY_VECTOR=100
db2stop
db2start
```

要充分利用 Oracle 应用程序的 DB2 兼容性功能部件， **DB2_COMPATIBILITY_VECTOR** 的推荐设置是 ORA，这将设置所有兼容性位。

在支持字符或图形常量之前，在其 SQL 中使用弱类型转换的应用程序将无法针对 DB2 产品进行编译。使用隐式强制类型转换，可以用非常灵活的方式对比、指派和操作字符串和数量。启用字符或图形常量支持后，长度小于或等于 254 字节的字符或图形字符串常量的数据类型将分别是 CHAR 或 GRAPHIC。长度大于 254 字节的字符或图形字符串常量的数据类型将分别是 VARCHAR 或 VARGRAPHIC。由于此数据类型指定可能会更改某些 SQL 语句的结果，因此，对于不进行切换的数据库，强烈建议您使用此注册表变量设置。

作用

启用该支持后，按以下方式处理这些常量:

- 长度小于或等于 254 字节的字符串常量的数据类型为 CHAR。
- 长度小于或等于 254 字节的图形字符串常量的数据类型为 GRAPHIC。
- 长度大于 254 字节的字符串常量的数据类型为 VARCHAR。
- 长度大于 254 字节的图形字符串常量的数据类型为 VARGRAPHIC。

由于这些数据类型指定可能会更改某些 SQL 语句的结果类型，所以不要切换此注册表变量设置。

第 40 章 SQL 数据访问级别强制

例程（存储过程或用户定义的函数）能够执行 SQL 语句的程度由该例程的 SQL访问级别确定。

存在以下四个 SQL 数据访问级别:

- NO SQL
- CONTAINS SQL
- READS SQL DATA
- MODIFIES SQL DATA

缺省情况下，在编译时，SQL PL 和 PL/SQL 例程会强制执行数据访问级别。如果某个例程包含的 SQL 语句要求数据访问级别超过该例程，那么在您创建该例程时会返回错误。同样，如果某个例程调用的另一个例程的数据访问级别超过该调用例程，那么在您创建第一个例程时会返回错误。此外，如果将已编译用户定义的函数定义为 MODIFIES SQL DATA，那么只能将它用作复合 SQL（编译）语句内赋值语句右边的独占元素。编译该语句时，也会执行此检查。

从 V9.7 FP3 开始，通过设置 **DB2_COMPATIBILITY_VECTOR** 注册表变量，可使 SQL PL 和 PL/SQL 例程强制执行运行时而不是编译时的数据访问级别。要启用此支持，请将该注册表变量设置为十六进制值 0x10000（位位置 17），然后停止并重新启动该实例以使新设置生效。

```
db2set DB2_COMPATIBILITY_VECTOR=10000
db2stop
db2start
```

要充分利用 Oracle 应用程序的 DB2 兼容性功能部件，DB2_COMPATIBILITY_VECTOR 的推荐设置是 ORA，这将设置所有兼容性位。

现在，该强制在运行时在该语句级别执行。当执行了超出当前 SQL 数据访问级别的语句时，会返回错误。如果例程调用了使用更严格 SQL 数据访问级别定义的另一例程，那么被调用例程会继承其父级的数据访问级别。此外，如果将已编译用户定义的函数定义为 MODIFIES SQL DATA，并且它不是复合 SQL（编译）语句内赋值语句右边的独占元素，那么仅当该函数发出用于修改 SQL 数据的 SQL 语句时，才会返回错误。

第 41 章 外连接运算符

当您设置 **DB2_COMPATIBILITY_VECTOR** 注册表变量来支持外连接运算符 (+) 时，查询可以在 WHERE 子句的谓词中使用此运算符作为备用语法。

连接是指根据信息的某些公共域对来自两个或更多个表的数据进行组合的过程。当相应行中的信息与基本连接条件相匹配时，一个表中的行与另一个表中的行匹配。外连接会返回所有满足连接条件的行，还会从其行都不满足该连接条件的一个或两个表返回部分或全部行。如果可行，那么还应该使用 RIGHT OUTER JOIN、LEFT OUTER JOIN 或 FULL OUTER JOIN 的外连接语法。应该仅在从数据库产品（而不是 DB2 产品）启用应用程序以在 DB2 数据库系统上运行时使用外连接运算符。

启用

通过将 DB2_COMPATIBILITY_VECTOR 注册表变量设置为十六进制值 0x04（位位置 3），然后停止并重新启动该实例以使新设置生效，从而启用外连接运算符支持。

```
db2set DB2_COMPATIBILITY_VECTOR=04
db2stop
db2start
```

要充分利用 Oracle 应用程序的 DB2 兼容性功能部件，DB2_COMPATIBILITY_VECTOR 的推荐设置是 ORA，这将设置所有兼容性位。

示例

您将括号中应用的外连接运算符 (+) 跟在谓词中的列名称之后，该谓词从两个表的列引用，如以下示例所示:

- 以下查询执行表 T1 和 T2 的左外连接。在 FROM 子句中包括两个表，由逗号分隔。在同时引用了 T1 的谓词中将外连接运算符应用于 T2 中的所有列。

  ```
  SELECT * FROM T1
    LEFT OUTER JOIN T2 ON T1.PK1 = T2.FK1
      AND T1.PK2 = T2.FK2
  ```

  以前的查询等于以下一个列，它使用外连接运算符:

  ```
  SELECT * FROM T1, T2
    WHERE T1.PK1 = T2.FK1(+)
      AND T1.PK2 = T2.FK2(+)
  ```

- 以下查询执行表 T1 和 T2 的右外连接。在 FROM 子句中包括这两个表，使用逗号进行分隔，并在同时引用了 T2 的谓词中将外连接运算符应用于 T1 中的所有列。

  ```
  SELECT * FROM T1
    RIGHT OUTER JOIN T2 ON T1.FK1 = T2.PK1
      AND T1.FK2 = T2.PK2
  ```

  以前的查询等于以下一个列，它使用外连接运算符:

  ```
  SELECT * FROM T1, T2
    WHERE T1.FK1(+) = T2.PK1
      AND T1.FK2(+) = T2.PK2
  ```

有时，对列标记了外连接运算符的表称为 *NULL 生成者*。

一组由 AND 运算符分隔的谓词被称为 *AND 因子*。如果 WHERE 子句未包含任何 AND 运算符，那么该 WHERE 子句中的谓词集被视为唯一的 AND 因子。

规则

以下规则应用于外连接运算符:

- 谓词
  - 考虑 WHERE 谓词时，以进行 AND 运算的布尔因子为粒度。
  - 可以存在局部谓词（例如 `T1.A(+) = 5`），但它们通过连接来执行。不带 (+) 的局部谓词将在连接之后执行。
- 布尔
  - 每个布尔项最多可以引用两个表，例如，不允许 `T1.C11 + T2.C21 = T3.C3(+)`。
  - 不允许对外连接布尔项进行关联。
- 外连接运算符
  - 无法在显式的 JOIN 语法所在的同一子选择中指定外连接运算符
  - 只能在作用于同一个子选择的 FROM 子句所指定表的相关联列的 WHERE 子句中，指定外连接运算符。
  - 无法将外连接运算符应用于整个表达式。在 AND 因子中，每个来自同一个表的列引用都必须后跟外连接运算符，例如 `T1.COL1 (+) - T1.COL2 (+) = T2.COL1`。
  - 只能在作用于同一个子选择的 FROM 子句所指定表的相关联列的 WHERE 子句中，指定外连接运算符。
- NULL 生成者
  - 每个表最多可以是另外一个表的 NULL 生成者。如果一个表连接到第三个表，那么它必须是外表。
  - 仅将某个表作为查询中另一个表的 NULL 生成者使用一次。
  - 无法将同一个表同时用作构成循环的不同外连接中的外表和 NULL 生成者。如果谓词的链返回到先前的表，那么表示跨多个连接构成循环。

    例如，以下查询从 T1 作为第一个谓词中的外表开始，然后在第三个谓词中返回到 T1，从而构成循环。T2 同时用作第一个谓词的 NULL 生成者以及第二个谓词中的外表，但此用法本身不是循环。

    ```
    SELECT ... FROM T1,T2,T3
      WHERE T1.a1 = T2.b2(+)
        AND T2.b2 = T3.c3(+)
        AND T3.c3 = T1.a1(+)   -- 无效的循环
    ```
- AND 因子
  - 一个 AND 因子只能将一个表作为 NULL 生成者。每个后跟外连接运算符的列引用都必须来自同一个表。
  - 包含外连接运算符的 AND 因子最多可以引用两个表。
  - 如果两个表之间的外连接需要多个 AND 因子，那么必须在所有这些 AND 因子中指定外连接运算符。如果 AND 因子未指定外连接运算符，那么将对外连接的结果处理该因子。
  - 谓词只涉及一个表的 AND 因子可以指定外连接运算符的条件是，至少有另一个 AND 因子符合以下条件:
    - AND 因子必须涉及与 NULL 生成者相同的表。

- AND 因子必须涉及作为外表的另一个表。

– 如果 AND 因子包含只涉及一个表的谓词并且不包含外连接运算符，那么它将对连接结果进行处理。
– 包含外连接运算符的 AND 因子必须遵循在所连接表下定义的 ON 子句的连接条件规则。

第 42 章 分层查询

分层查询是一种递归查询形式，它使用 CONNECT BY 子句从关系数据中检索层次结构，例如材料清单。

启用

通过将 DB2_COMPATIBILITY_VECTOR 注册表变量设置为十六进制值 0x08（位位置 4），然后停止并重新启动该实例以使新设置生效，从而启用分层查询支持。

```
db2set DB2_COMPATIBILITY_VECTOR=08
db2stop
db2start
```

要充分利用 Oracle 应用程序的 DB2 兼容性功能部件，DB2_COMPATIBILITY_VECTOR 的推荐设置是 ORA，这将设置所有兼容性位。

然后您可以使用 CONNECT BY 语法，包括伪列、一元运算符和 SYS_CONNECT_BY_PATH 标量函数。

分层查询包含 CONNECT BY 子句，该子句定义父元素和子元素之间的连接条件。“连接”递归将同一个子查询用于种子值（START WITH 子句）和递归步骤（CONNECT BY 子句）。此组合提供了一种简明的方法来表示递归，例如材料清单、报告链或电子邮件线程。

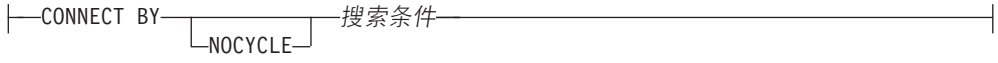
出现循环时，“连接”递归将返回错误。*循环*是指，某一行直接或间接地生成其自身。通过使用可选的 CONNECT BY NOCYCLE 子句，可以指示递归忽略重复的行，从而避免循环和错误。分层查询或“连接”递归不同于 DB2 递归。有关这些差异的更多信息，请参阅将 CONNECT BY 移植到 DB2。

分层查询子句

包含*分层查询子句*的子选择被称为“分层查询”。

START WITH 子句:

CONNECT BY 子句:



START WITH 子句

START WITH 表示递归的种子值。*START WITH 子句*指定分层查询的中间结果表

H_1。表 H_1 包含 R 中*搜索条件*为 true 的那么行。如果未指定 *START WITH 子句*，那么 H_1 将是整个中间结果表 R。*START WITH 子句* 中*搜索条件*的规则与 WHERE 子句中的规则相同。

CONNECT BY 子句

CONNECT BY 描述了该递归步骤。*CONNECT BY 子句*通过将 H_n 与 R 连接，使用搜索条件来根据 H_n 生成中间结果表 H_{n+1}。如果指定 NOCYCLE 关键字，那么重复行不会包括在中间结果表 H_{n+1} 中。不会返回错误。*CONNECT BY 子句*中*搜索条件*的规则与 WHERE 子句的规则相同，除非无法指定 OLAP 规范 (SQLSTATE 42903)。

建立第一个中间结果表 H_1 之后，将生成后续中间结果表 H_2、H_3 等等。通过使用 *CONNECT BY 子句*作为连接条件将 H_n 与表 R 进行连接生成 H_{n+1}，从而生成后续创建的中间结果表。R 是子选择的 FROM 子句以及 WHERE 子句中任何连接谓词的结果。当 H_{n+1} 生成空结果表时，此过程停止。只要 UNION ALL 适用于每个中间结果表，那么*分层查询子句*的结果表 H 是该结果。

您可以使用一元运算符 PRIOR 将对 H_n（上一个递归步骤或父级步骤）的引用列与对 R 的列引用加以区分。考虑以下示例:

```
CONNECT BY MGRID = PRIOR EMPID
```

MGRID 与 R 一起进行解析，而 EMPID 在上一个中间结果表 H_n 的列中进行解析。

规则

- 如果中间结果表 H_{n+1} 对于某分层路径将返回 R 中的某一行，而该行与 R 中已包含在该分层路径中的另一行相同，那么将返回错误 (SQLSTATE 560CO)。
- 如果指定了 NOCYCLE 关键字，那么不会返回错误，但重复的行不会包括在中间结果表 H_{n+1} 中。
- 最多支持 64 级递归 (SQLSTATE 54066)。
- 作为分层查询的子选择按不完整顺序返回中间结果集，除非您通过使用显式的 ORDER BY 子句、GROUP BY 或 HAVING 子句或者选择列表中的 DISTINCT 关键字来破坏该顺序。按不完整顺序返回行时，在 H_{n+1} 中为特定层次结构生成的行将紧跟在生成那些行的 H_n 中的行之后。您可以使用 ORDER SIBLINGS BY 子句在同一父级所生成的一组行中强制实施顺序。
- 具体化查询表不支持分层查询 (SQLSTATE 428EC)。
- 无法将 CONNECT BY 子句与 XML 函数或 XQuery 配合使用 (SQLSTATE 428H4)。
- 无法在以下位置中对序列指定 NEXT VALUE 表达式 (SQLSTATE 428F9):
  - CONNECT_BY_ROOT 运算符或 SYS_CONNECT_BY_PATH 函数的参数列表
  - START WITH 和 CONNECT BY 子句

备注

- 分层查询支持在以下方面影响子选择:
  - 按如下顺序处理子选择的子句:
    1. FROM 子句
    2. *分层查询子句*
    3. WHERE 子句
    4. GROUP BY 子句

5. HAVING 子句
6. SELECT 子句
7. ORDER BY 子句
8. FETCH FIRST 子句

– 特殊规则适用于在 WHERE 子句中处理谓词的顺序。*搜索条件*将与其 AND 条件（合取）一起包括到谓词中。如果某个谓词是隐式连接谓词（即，它引用了 FROM 子句中的多个表），那么将在应用*分层查询子句*之前应用该谓词。任何最多引用了 FROM 子句中的一个表的谓词都将应用于*分层查询子句*的中间结果表。

  如果编写涉及连接的分层查询，那么使用显式的连接表通过 ON 子句来避免产生与 WHERE 子句谓词的应用相关的混淆。

– 可以指定 ORDER SIBLINGS BY 子句。此子句指定顺序仅应用于层次结构中的胞代。

- *伪列*是限定的标识或未限定的标识，它在特定上下文中具有含义，并与某些列和变量共享同一个名称空间。如果未限定的标识未标识列或变量，那么检查该标识以确定它是否标识了伪列。

  LEVEL 是一个供分层查询使用的伪列。LEVEL 伪列返回层次结构中生成行的递归步骤。所有由 START WITH 子句生成的行都将返回值 1。通过应用 CONNECT BY 子句的第一个迭代生成的行将返回 2，依此类推。此列的数据类型是 INTEGER NOT NULL。

  必须在分层查询的上下文中指定 LEVEL。无法在 START WITH 子句中指定 LEVEL 作为 CONNECT_BY_ROOT 运算符的参数或作为 SYS_CONNECT_BY_PATH 函数的参数 (SQLSTATE 428H4)。

- 支持分层查询的一元运算符是 CONNECT_BY_ROOT 和 PRIOR。
- 支持分层查询的函数是 SYS_CONNECT_BY_PATH 标量函数。

示例

- 以下报告链示例演示“连接”递归。此示例基于名为 MY_EMP 的表，该表是使用如下数据来创建和填充的:

```
CREATE TABLE MY_EMP(
  EMPID  INTEGER NOT NULL PRIMARY KEY,
  NAME   VARCHAR(10),
  SALARY DECIMAL(9, 2),
  MGRID  INTEGER);

INSERT INTO MY_EMP VALUES ( 1, 'Jones',    30000, 10);
INSERT INTO MY_EMP VALUES ( 2, 'Hall',     35000, 10);
INSERT INTO MY_EMP VALUES ( 3, 'Kim',      40000, 10);
INSERT INTO MY_EMP VALUES ( 4, 'Lindsay',  38000, 10);
INSERT INTO MY_EMP VALUES ( 5, 'McKeough', 42000, 11);
INSERT INTO MY_EMP VALUES ( 6, 'Barnes',   41000, 11);
INSERT INTO MY_EMP VALUES ( 7, 'O''Neil',  36000, 12);
INSERT INTO MY_EMP VALUES ( 8, 'Smith',    34000, 12);
INSERT INTO MY_EMP VALUES ( 9, 'Shoeman',  33000, 12);
INSERT INTO MY_EMP VALUES (10, 'Monroe',   50000, 15);
INSERT INTO MY_EMP VALUES (11, 'Zander',   52000, 16);
INSERT INTO MY_EMP VALUES (12, 'Henry',    51000, 16);
INSERT INTO MY_EMP VALUES (13, 'Aaron',    54000, 15);
INSERT INTO MY_EMP VALUES (14, 'Scott',    53000, 16);
```

```
INSERT INTO MY_EMP VALUES (15, 'Mills',    70000, 17);
INSERT INTO MY_EMP VALUES (16, 'Goyal',    80000, 17);
INSERT INTO MY_EMP VALUES (17, 'Urbassek', 95000, NULL);
```

以下查询返回所有为 Goyal 工作的职员以及某些其他信息，例如报告链:

```
 1 SELECT NAME,
 2        LEVEL,
 3        SALARY,
 4        CONNECT_BY_ROOT NAME AS ROOT,
 5        SUBSTR(SYS_CONNECT_BY_PATH(NAME, ':'), 1, 25) AS CHAIN
 6   FROM MY_EMP
 7   START WITH NAME = 'Goyal'
 8   CONNECT BY PRIOR EMPID = MGRID
 9   ORDER SIBLINGS BY SALARY;

NAME       LEVEL       SALARY      ROOT  CHAIN
---------- ----------- ----------- ----- ---------------
Goyal                1    80000.00 Goyal :Goyal
Henry                2    51000.00 Goyal :Goyal:Henry
Shoeman              3    33000.00 Goyal :Goyal:Henry:Shoeman
Smith                3    34000.00 Goyal :Goyal:Henry:Smith
O'Neil               3    36000.00 Goyal :Goyal:Henry:O'Neil
Zander               2    52000.00 Goyal :Goyal:Zander
Barnes               3    41000.00 Goyal :Goyal:Zander:Barnes
McKeough             3    42000.00 Goyal :Goyal:Zander:McKeough
Scott                2    53000.00 Goyal :Goyal:Scott
```

第 7 和第 8 行组成递归核心: 可选的 START WITH 子句描述要用于源表以作为递归种子值的 WHERE 子句。在本例中，将只选择职员 Goyal 的行。如果省略 START WITH 子句，那么将使用整个源表作为递归种子值。CONNECT BY 子句描述在给定现有行的情况下如何找到下一组行。一元运算符 PRIOR 用于对上一步骤中的值与当前步骤中的值进行区分。PRIOR 将 EMPID 标识为上一递归步骤的职员标识，并将 MGRID 标识为来自当前递归步骤。

行 2 中的 LEVEL 伪列指示当前的递归级别。

CONNECT_BY_ROOT 是一元运算符，它始终返回其自变量在第一个递归步骤中的值；即，显式或隐式 START WITH 子句所返回的值。

SYS_CONNECT_BY_PATH() 是二进制函数，它将第二个自变量添加到第一个自变量开头，然后将结果追加到它在上一递归步骤中生成的值末尾。自变量必须具有字符类型。

除非被明确覆盖，否则“连接”递归将按不完整顺序来返回结果集；即，某个递归步骤所生成的行始终跟在生成它们的行之后。处于同一递归级别的胞代没有特定的顺序。第 9 行的 ORDER SIBLINGS BY 子句定义这些胞代的顺序，这将进一步优化不完整顺序，并有可能形成完整顺序。

- 返回 DEPARTMENT 表的组织结构。使用部门级别将层次结构可视化。

```
SELECT LEVEL, CAST(SPACE((LEVEL - 1) * 4) || '/' || DEPTNAME
    AS VARCHAR(40)) AS DEPTNAME
  FROM DEPARTMENT
  START WITH DEPTNO = 'A00'
  CONNECT BY NOCYCLE PRIOR DEPTNO = ADMRDEPT
```

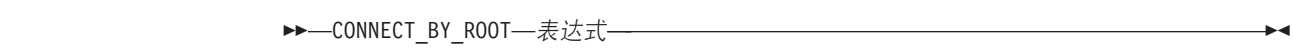
此查询返回:

```
LEVEL       DEPTNAME
----------- ----------------------------------------
          1 /SPIFFY COMPUTER SERVICE DIV.
```

```
2     /PLANNING
2     /INFORMATION CENTER
2     /DEVELOPMENT CENTER
3         /MANUFACTURING SYSTEMS
3         /ADMINISTRATION SYSTEMS
2     /SUPPORT SERVICES
3         /OPERATIONS
3         /SOFTWARE SUPPORT
3         /BRANCH OFFICE F2
3         /BRANCH OFFICE G2
3         /BRANCH OFFICE H2
3         /BRANCH OFFICE I2
3         /BRANCH OFFICE J2
```

CONNECT_BY_ROOT 一元运算符

CONNECT_BY_ROOT 一元运算符仅适于在分层查询中使用。对于层次结构中的每一行，此运算符返回该行的根祖代的表达式。



表达式
未包含 NEXT VALUE 表达式、分层查询构造（例如 LEVEL 伪列）、SYS_CONNECT_BY_PATH 函数或 OLAP 函数的表达式。如果指定任何这些项目，那么都将返回 SQLSTATE 428H4。

用法

此运算符的结果类型是表达式的结果类型。

以下规则适用于 CONNECT_BY_ROOT 运算符:

- CONNECT_BY_ROOT 运算符的优先级高于任何中缀运算符，如加号 (+) 或双竖线 (||)。因此，要传递包含中缀运算符的表达式作为自变量，必须使用括号。例如，以下表达式返回的根祖代行的 FIRSTNME 值与层次结构中实际行的 LASTNAME 值并置。

  ```
  CONNECT_BY_ROOT FIRSTNME || LASTNAME
  ```

  该表达式相当于以下列表中的第一个表达式，但不是第二个表达式:

  ```
  (CONNECT_BY_ROOT FIRSTNME) || LASTNAMECONNECT_BY_ROOT (FIRSTNME || LASTNAME)
  ```

- 不能在分层查询的 START WITH 子句或 CONNECT BY 子句中指定 CONNECT_BY_ROOT 运算符 (SQLSTATE 428H4)。
- 不能指定 CONNECT_BY_ROOT 运算符作为 SYS_CONNECT_BY_PATH 函数的自变量 (SQLSTATE 428H4)。

以下查询返回 DEPARTMENT 表中部门及其根部门的层次结构:

```
SELECT CONNECT_BY_ROOT DEPTNAME AS ROOT, DEPTNAME
  FROM DEPARTMENT START WITH DEPTNO IN ('B01','C01','D01','E01')
  CONNECT BY PRIOR DEPTNO = ADMRDEPT
```

此查询返回以下结果:

```
ROOT                DEPTNAME
------------------- ----------------------
PLANNING            PLANNING
INFORMATION CENTER  INFORMATION CENTER
```

```
DEVELOPMENT CENTER DEVELOPMENT CENTER
DEVELOPMENT CENTER MANUFACTURING SYSTEMS
DEVELOPMENT CENTER ADMINISTRATION SYSTEMS
SUPPORT SERVICES   SUPPORT SERVICES
SUPPORT SERVICES   OPERATIONS
SUPPORT SERVICES   SOFTWARE SUPPORT
SUPPORT SERVICES   BRANCH OFFICE F2
SUPPORT SERVICES   BRANCH OFFICE G2
SUPPORT SERVICES   BRANCH OFFICE H2
SUPPORT SERVICES   BRANCH OFFICE I2
SUPPORT SERVICES   BRANCH OFFICE J2
```

PRIOR 一元运算符

PRIOR 一元运算符仅适于在分层查询的 CONNECT BY 子句中使用。要获得整个级别的所有下级，必须将 PRIOR 运算符添加到分层查询的 CONNECT BY 子句中。

表达式

未包含 NEXT VALUE 表达式、分层查询构造（例如 LEVEL 伪列）、SYS_CONNECT_BY_PATH 函数或 OLAP 函数的任何表达式。如果指定任何这些项目，那么都将返回 SQLSTATE 428H4。

用法

CONNECT BY 子句在分层查询的中间结果表 H_n 与 FROM 子句中指定的源结果表之间执行内连接。如果对 FROM 子句中引用的表进行的列引用是 PRIOR 运算符的自变量，那么所有那些列引用都被视为涉及表 H_n。

此运算符的结果数据类型是表达式的结果数据类型。

如以下示例所示，通常将中间结果表 H_n 的主键连接到源结果表的外键，以便以递归方式遍历层次结构:

```
CONNECT BY PRIOR T.PK = T.FK
```

如果主键是组合键，那么对每个列添加前缀 PRIOR，如以下示例所示:

```
CONNECT BY PRIOR T.PK1 = T.FK1 AND PRIOR T.PK2 = T.FK2
```

PRIOR 运算符的优先级高于任何中缀运算符，如加号 (+) 或双竖线 (||)。因此，要传递包含中缀运算符的表达式作为自变量，必须使用括号。需要使用操作数和运算符的圆括号组指示在其中执行操作的目标顺序。例如，以下表达式返回之前行的 FIRSTNME 值与层次结构中实际行的 LASTNAME 值的并置。

```
PRIOR FIRSTNME || LASTNAME
```

该表达式相当于以下列表中的第一个表达式，但不是第二个表达式:

```
(PRIOR FIRSTNME) || LASTNAMEPRIOR (FIRSTNME || LASTNAME)
```

如果指定分层查询的 CONNECT BY 子句的外部 PRIOR 运算符，那么将返回 SQLSTATE 428H4。

示例

- 以下查询返回 DEPARTMENT 表中部门的层次结构:

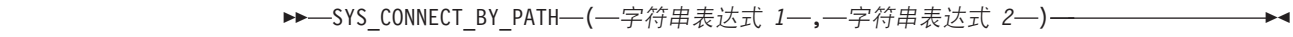
```
SELECT LEVEL, DEPTNAME
  FROM DEPARTMENT START WITH DEPTNO = 'A00'
  CONNECT BY NOCYCLE PRIOR DEPTNO = ADMRDEPT
```

此查询返回以下结果:

```
LEVEL       DEPTNAME
----------- ------------------------------------
          1 SPIFFY COMPUTER SERVICE DIV.
          2 PLANNING
          2 INFORMATION CENTER
          2 DEVELOPMENT CENTER
          3 MANUFACTURING SYSTEMS
          3 ADMINISTRATION SYSTEMS
          2 SUPPORT SERVICES
          3 OPERATIONS
          3 SOFTWARE SUPPORT
          3 BRANCH OFFICE F2
          3 BRANCH OFFICE G2
          3 BRANCH OFFICE H2
          3 BRANCH OFFICE I2
          3 BRANCH OFFICE J2
```

SYS_CONNECT_BY_PATH

SYS_CONNECT_BY_PATH 函数构建代表在分层查询中从根目录到某个节点的路径的字符串。



该模式是 SYSIBM。

字符串表达式 1

这是一个字符串表达式，用于标识行。该表达式不得包括以下列表中的任何项目；除非返回括号中的 SQLSTATE:

- 序列的 NEXT VALUE 表达式 (SQLSTATE 428F9)
- 任何分层查询构造，如 LEVEL 伪列或 CONNECT_BY_ROOT 运算符 (SQLSTATE 428H4)
- OLAP 函数 (SQLSTATE 428H4)
- 聚集函数 (SQLSTATE 428H4)

字符串表达式 2

这是用作分隔符的常量字符串。该表达式不得包括以下列表中的任何项目；除非返回括号中的 SQLSTATE:

- 序列的 NEXT VALUE 表达式 (SQLSTATE 428F9)
- 任何分层查询构造，如 LEVEL 伪列或 CONNECT_BY_ROOT 运算符 (SQLSTATE 428H4)
- OLAP 函数 (SQLSTATE 428H4)
- 聚集函数 (SQLSTATE 428H4)

结果是可变长度字符串。结果数据类型的长度属性是 1000 与*字符串表达式 1* 的长度属性之间的较大者。

对于处于伪列 LEVEL *n* 的特定行，将按如下方式构建字符串:

- 步骤 1（使用第一个中间结果表 H_1 中的根行的值）：

  $path_1$:= *字符串表达式 2* || *字符串表达式 1*
- 步骤 *n*（基于中间结果表 H_n 中的行）：

  $path_n$:= $path_{n-1}$ || *字符串表达式 2* || *字符串表达式 1*

以下规则适用于 SYS_CONTEXT_BY_PATH 函数:

- 如果指定分层查询的上下文的外部函数，那么将返回 SQLSTATE 428H4。
- 如果使用 START WITH 子句或 CONNECT BY 子句中的函数，那么将返回 SQLSTATE 428H4。

以下示例返回 DEPARTMENT 表中部门的层次结构:

```
SELECT CAST(SYS_CONNECT_BY_PATH(DEPTNAME, '/')
    AS VARCHAR(76)) AS ORG
  FROM DEPARTMENT START WITH DEPTNO = 'A00'
  CONNECT BY NOCYCLE PRIOR DEPTNO = ADMRDEPT
```

此查询返回以下结果:

```
ORG
------------------------------------------------------------------------
/SPIFFY COMPUTER SERVICE DIV.
/SPIFFY COMPUTER SERVICE DIV./PLANNING
/SPIFFY COMPUTER SERVICE DIV./INFORMATION CENTER
/SPIFFY COMPUTER SERVICE DIV./DEVELOPMENT CENTER
/SPIFFY COMPUTER SERVICE DIV./DEVELOPMENT CENTER/MANUFACTURING SYSTEMS
/SPIFFY COMPUTER SERVICE DIV./DEVELOPMENT CENTER/ADMINISTRATION SYSTEMS
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/OPERATIONS
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/SOFTWARE SUPPORT
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/BRANCH OFFICE F2
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/BRANCH OFFICE G2
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/BRANCH OFFICE H2
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/BRANCH OFFICE I2
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/BRANCH OFFICE J2
```

第 43 章 兼容性数据库配置参数

可以使用数据库配置参数指示与某些数据类型关联的兼容性语义是否适用于所连接的数据库。

可检查的兼容性参数是:

date_compat
指示与 TIMESTAMP(0) 数据类型关联的 DATE 数据类型兼容性语义是否适用于所连接的数据库。

number_compat
指示与 NUMBER 数据类型关联的兼容性语义是否适用于所连接的数据库。

varchar2_compat
指示与 VARCHAR2 数据类型关联的兼容性语义是否适用于所连接的数据库。

在这些参数中，每个参数的值都在数据库创建时确定并基于 **DB2_COMPATIBILITY_VECTOR** 注册表变量的设置。无法更改该值。

第 44 章 ROWNUM 伪列

将对 ROWNUM 伪列的任何未解析和未限定列引用转换为 OLAP 规范 ROW_NUMBER() OVER()。

启用

通过将 **DB2_COMPATIBILITY_VECTOR** 注册表变量设置为十六进制值 0x01（位位置 1），然后停止并重新启动该实例以使新设置生效，从而启用 ROWNUM 伪列支持。

```
db2set DB2_COMPATIBILITY_VECTOR=01
db2stop
db2start
```

要充分利用 Oracle 应用程序的 DB2 兼容性功能部件，DB2_COMPATIBILITY_VECTOR 的推荐设置是 ORA，这将设置所有兼容性位。

ROWNUM 在结果集中对记录进行编号。符合 SELECT 语句中 WHERE 子句条件的第一个记录所给的行编号为 1，且符合相同标准的每个后续记录都会增加行号。

在子选择的 WHERE 子句中，允许使用 ROWNUM 和 ROW_NUMBER() OVER()，它们对于限制结果集的大小而言非常有用。如果在 WHERE 子句中使用 ROWNUM，并且同一个子选择包含 ORDER BY 子句，那么将在对 ROWNUM 谓词进行求值之前进行排序。同样，如果在 WHERE 子句中使用 ROW_NUMBER() OVER() 函数，并且同一个子选择包含 ORDER BY 子句，那么将在对 ROW_NUMBER() OVER() 函数求值之前进行排序。如果在 WHERE 子句中使用 ROW_NUMBER() OVER() 函数，那么无法指定“窗口顺序子句”或“窗口分区子句”。

在将未限定的“ROWNUM”引用转换为 ROW_NUMBER() OVER() 函数之前，DB2 会尝试将此引用解析为下列其中一项:

- 当前 SQL 查询中的列
- 局部变量
- 例程参数
- 全局变量

在 ROWNUM 伪列支持处于启用状态时，请避免将“ROWNUM”用作列名或变量名。

示例

在假定已对所连接的数据库启用 ROWNUM 伪列支持的情况下，检索临时表中存储的结果集的第 20 到 40 行。

```
SELECT TEXT FROM SESSION.SEARCHRESULTS
  WHERE ROWNUM BETWEEN 20 AND 40
  ORDER BY ID
```

注意，ROWNUM 受 ORDER BY 子句影响。

第 45 章 DUAL 表

DB2 数据服务器将任何对 DUAL 表进行的未限定引用解析为一个内置视图，该视图返回一行以及一个名为 DUMMY 的列（该列的值为“X”）。

启用

通过将 **DB2_COMPATIBILITY_VECTOR** 注册表变量设置为十六进制值 0x02（位位置 2），然后停止并重新启动该实例以使新设置生效，从而启用 DUAL 表支持。

```
db2set DB2_COMPATIBILITY_VECTOR=02
db2stop
db2start
```

要充分利用 Oracle 应用程序的 DB2 兼容性功能部件，DB2_COMPATIBILITY_VECTOR 的推荐设置是 ORA，这将设置所有兼容性位。

将对 DUAL 表进行的未限定表引用解析为 SYSIBM.DUAL。

如果存在用户定义的名为 DUAL 的表，那么仅当显式地限定对这个用户定义的表进行的表引用时，DB2 服务器才会解析此引用。

示例 1

通过从 DUAL 中进行选择来生成随机数。

```
SELECT RAND() AS RANDOM_NUMBER FROM DUAL
```

示例 2

检索 CURRENT SCHEMA 专用寄存器的值。

```
SET SCHEMA = MYSCHEMA;
SELECT CURRENT SCHEMA AS CURRENT_SCHEMA FROM DUAL;
```

第 46 章 不敏感游标

通过在打开时将游标具体化可使游标对后续语句不敏感。在结果表的临时副本上具体化所有行后，打开游标时执行的语句不会影响该结果表。

启用

通过将 **DB2_COMPATIBILITY_VECTOR** 注册表变量设置为十六进制值 0x1000（位位置 13），然后停止并重新启动该实例以使新设置生效，从而启用不敏感游标。

```
db2set DB2_COMPATIBILITY_VECTOR=1000
db2stop
db2start
```

要充分利用 Oracle 应用程序的 DB2 兼容性功能部件，DB2_COMPATIBILITY_VECTOR 的推荐设置是 ORA，这将设置所有兼容性位。

如果在打开时具体化结果集，那么该游标表现为只读游标。只要定义为 WITH RETURN 的所有游标未被显式标记为 FOR UPDATE，那么这些游标便是 INSENSITIVE。如果没有启用不敏感游标支持，那么无法保证将在打开时具体化 DB2 游标。因此，在您针对可立即具体化游标的 DB2 数据库和关系数据库运行相同查询时生成的结果集可能会有所不同。例如，Sybase TSQL 包括从批处理语句或过程（用于为调用者生成结果集）发出查询的功能。立即具体化该查询。块中的其他语句希望它们不会影响结果并针对该查询中引用的同一个表发出各种语句（如 DELETE）。当某个相似方案在没有不敏感游标的情况下运行时，该游标中的结果集将不同于 Sybase 结果。

按如下方式支持不敏感游标:

- 可以在 DECLARE CURSOR 语句中将游标定义为 INSENSITIVE，该语句在复合 SQL（已编译）语句中使用。
- 如果您使用 **BIND** 命令的 **STATICREADONLY INSENSITIVE** 参数绑定程序包，那么所有只读游标和模糊游标都是不敏感游标。
- 如果为 **DB2_SQLROUTINE_PREPOPTS** 注册表变量或 SET_ROUTINE_OPTS 过程指定 STATICREADONLY INSENSITIVE 选项，那么在打开时，SQL 例程会具体化作为静态 SQL 发放的所有只读游标和模糊游标。

限制

任何预编译器均不支持 INSENSITIVE 关键字。CLI 和 JDBC 都不提供对标识不敏感不可滚动游标的支持（游标属性或结果集属性）。

示例

此代码在执行 DELETE 语句之前将 SELECT 语句的整个结果集返回到客户机。

```
BEGIN
 DECLARE res INSENSITIVE CURSOR WITH RETURN TO CLIENT FOR
  SELECT * FROM T;
  OPEN T;
  DELETE FROM T;
END
```

第 47 章 INOUT 参数

可通过使用 DEFAULT 关键字为某个过程定义 INOUT 参数，使其具有缺省值。

启用

通过将 **DB2_COMPATIBILITY_VECTOR** 注册表变量设置为十六进制值 0x2000（位位置 14），然后停止并重新启动该实例以使新设置生效，从而启用 INOUT 参数支持。

```
db2set DB2_COMPATIBILITY_VECTOR=2000
db2stop
db2start
```

要充分利用 Oracle 应用程序的 DB2 兼容性功能部件，DB2_COMPATIBILITY_VECTOR 的推荐设置是 ORA，这将设置所有兼容性位。

INOUT 参数既是输入参数，又是输出参数。可以使用 DEFAULT 关键字将 INOUT 参数的缺省值定义为表达式或 NULL。然后，如果通过指定 DEFAULT 调用该过程或不为该参数调用任何自变量，那么可将为该参数定义的缺省值用于初始化该过程。如果该过程存在，那么不会为此参数返回任何值。

限制

DEFAULT 关键字不支持函数中的 INOUT 参数。

示例

以下代码可创建一个带有可选 INOUT 参数的过程:

```
CREATE OR REPLACE PROCEDURE paybonus
  (IN empid INTEGER,
  IN percentbonus DECIMAL(2, 2),
  INOUT budget DECFLOAT DEFAULT NULL)
  ...
```

此过程通过员工的工资计算奖金金额，发放奖金，然后从部门预算中扣除奖金。如果没有为该过程指定预算，那么将忽略扣除部分。以下是如何调用该过程的示例:

```
CALL paybonus(12, 0.05, 50000);
CALL paybonus(12, 0.05, DEFAULT);
CALL paybonus(12, 0.05);
```

第 48 章 “当前已落实”语义

在*当前已落实*语义下，仅为读程序返回已落实数据。但读程序现在不等待写程序释放锁定。而是，读程序根据当前已落实的数据版本（即，写操作开始前的数据版本）来返回数据。

在进行行级锁定的游标稳定性 (CS) 隔离级别下，可能会发生锁定超时和死锁，对于未设计成能够避免此类问题的应用程序而言尤其如此。某些高吞吐量数据库应用程序无法容忍等待事务处理期间发出的锁定。某些应用程序还无法容忍处理未落实的数据，但仍需要非锁定行为以读取事务。

缺省情况下，对于新数据库，“当前已落实”语义处于打开状态。应用程序可以利用新行为，而不需要更改应用程序本身。要覆盖缺省行为，将 **cur_commit** 数据库配置参数设置为 DISABLED。例如，如果应用程序要求阻塞写程序以便对内部逻辑进行同步，那么覆盖此缺省值很有用。在将数据库从 V9.5 或更低版本进行升级期间，**cur_commit** 配置参数设置为 DISABLED，以保持行为与先前版本的行为一致。如果要在游标稳定性扫描上使用 “当前已落实”语义，需要在升级后将 **cur_commit** 配置参数设置为 ON。

“当前已落实”语义仅适用于不涉及目录表的只读扫描和用于评估或强制实施约束的内部扫描。由于“当前已落实”是在扫描级别确定的，因此写程序的访问方案可能包括当前已落实的扫描。例如，扫描只读子查询可能涉及“当前已落实”语义。

由于“当前已落实”语义遵循隔离级别语义，因此，在“当前已落实”语义下运行的应用程序将继续遵守隔离级别。

“当前已落实”语义要求为写程序增加日志空间。需要附加的空间来记录事务期间对数据行进行的第一次更新。要检索当前已落实的行映像，此数据是必需的。根据工作负载不同，这对使用的日志空间总量可能影响不大，也可能会产生重大影响。当 **cur_commit** 数据库配置参数设置为 DISABLED 时，对附加日志空间的需求不适用。

限制

当前落实的语义存在以下限制:

- 节中用于数据更新或删除操作的目标表对象不会使用当前已落实的语义。要进行修改的行必须处于锁定保护状态，以确保它们在满足属于更新操作的任何查询谓词后，不会发生更改。
- 对某个行进行了未落实修改的事务将强制当前已落实的读程序访问相应日志记录，以确定行的当前已落实版本。尽管能够以物理方式读取不再位于日志缓冲区中的日志记录，但当前已落实的语义不支持从日志归档中检索日志文件。这仅影响配置为使用无限记录的数据库。
- 以下扫描不会使用当前已落实的语义:
  - 目录表扫描
  - 用于实施引用完整性约束的扫描
  - 引用 LONG VARCHAR 或 LONG VARGRAPHIC 列的扫描
  - 范围集群表 (RCT) 扫描
  - 使用空间或扩展索引的扫描

示例

请考虑以下方案，此方案通过使用“当前已落实”语义避免死锁。在此方案中，两个应用程序更新两个不同的表，如步骤 1 中所示，但尚未执行落实。每个应用程序都尝试使用只读游标来读取另一应用程序已更新的表，如步骤 2 中所示。这些应用程序正在 CS 隔离级别运行。

步骤	应用程序 A	应用程序 B
1	update T1 set col1 = ? where col2 = ?	update T2 set col1 = ? where col2 = ?
2	select col1, col3, col4 from T2 where col2 >= ?	select col1, col5, from T1 where col5 = ? and col2 = ?
3	commit	commit

在没有“当前已落实”语义的情况下，在游标稳定性隔离级别下运行的这两个应用程序可能会创建死锁，从而导致其中一个应用程序失败。当两个应用程序都需要读取正在被另一应用程序更新的数据时，就会发生死锁情况。

在“当前已落实”语义下，如果在步骤 2 中运行查询的任何一个应用程序刚好需要正被另一应用程序更新的数据，那么该应用程序并不会等待锁定被释放。结果死锁不可能。第一个应用程序将找到并使用先前落实的数据版本。

第 49 章 兼容 Oracle 数据字典的视图

如果将 **DB2_COMPATIBILITY_VECTOR** 注册表变量设置为支持兼容 Oracle 数据字典的视图，那么创建数据库时会自动创建这些视图。

通过将 **DB2_COMPATIBILITY_VECTOR** 注册表变量设置为十六进制值 0x400（位位置 11），然后停止并重新启动该实例以使新设置生效，从而启用兼容 Oracle 数据字典的视图支持。

```
db2set DB2_COMPATIBILITY_VECTOR=400
db2stop
db2start
```

要充分利用 Oracle 应用程序的 DB2 兼容性功能部件，DB2_COMPATIBILITY_VECTOR 的推荐设置是 ORA，这将设置所有兼容性位。

数据字典是一个数据库元数据存储库。数据字典视图会进行自我描述。DICTIONARY 视图将返回一个列表，该列表中包含所有的数据字典视图以及用于描述各视图内容的注释。DICT_COLUMNS 视图将返回一个列表，该列表包含所有数据字典视图中的所有列。对于这两个视图，可确定可用信息及访问方式。

每个数据字典视图都有三个不同版本，每个版本都通过视图名称前缀加以标识。

- ALL_* 视图返回有关当前用户可访问的对象的信息。
- DBA_* 视图返回有关数据库中的所有对象的信息（不管谁拥有这些对象）。
- USER_* 视图返回有关当前数据库用户拥有的对象的信息。

一部分版本不适用于每个视图。

对于每个与 Oracle 数据字典兼容的视图，数据字典定义包含 CREATE VIEW、CREATE PUBLIC SYNONYM 和 COMMENT 语句。表 40 中列示了这些在 SYSIBMADM 模式中创建的视图。

表 40. 兼容 Oracle 数据字典的视图

类别	定义的视图
常规	DICTIONARY、DICT_COLUMNS、 USER_CATALOG、DBA_CATALOG、ALL_CATALOG、 USER_DEPENDENCIES、DBA_DEPENDENCIES、ALL_DEPENDENCIES、 USER_OBJECTS、DBA_OBJECTS、ALL_OBJECTS、 USER_SEQUENCES、DBA_SEQUENCES、ALL_SEQUENCES、 USER_TABLESPACES 和 DBA_TABLESPACES
表或视图	USER_CONSTRAINTS、DBA_CONSTRAINTS、ALL_CONSTRAINTS、 USER_CONS_COLUMNS、DBA_CONS_COLUMNS、ALL_CONS_COLUMNS、 USER_INDEXES、DBA_INDEXES、ALL_INDEXES、 USER_IND_COLUMNS、DBA_IND_COLUMNS、ALL_IND_COLUMNS、 USER_TAB_PARTITIONS、DBA_TAB_PARTITIONS、ALL_TAB_PARTITIONS、 USER_PART_TABLES、DBA_PART_TABLES、ALL_PART_TABLES、 USER_PART_KEY_COLUMNS、DBA_PART_KEY_COLUMNS、ALL_PART_KEY_COLUMNS、 USER_SYNONYMS、DBA_SYNONYMS、ALL_SYNONYMS、 USER_TABLES、DBA_TABLES、ALL_TABLES、 USER_TAB_COMMENTS、DBA_TAB_COMMENTS、ALL_TAB_COMMENTS、 USER_TAB_COLUMNS、DBA_TAB_COLUMNS、ALL_TAB_COLUMNS、 USER_COL_COMMENTS、DBA_COL_COMMENTS、ALL_COL_COMMENTS、 USER_TAB_COL_STATISTICS、DBA_TAB_COL_STATISTICS、ALL_TAB_COL_STATISTICS、 USER_VIEWS、DBA_VIEWS、ALL_VIEWS、 USER_VIEW_COLUMNS、DBA_VIEW_COLUMNS 和 ALL_VIEW_COLUMNS

表 40. 兼容 Oracle 数据字典的视图 (续)

类别	定义的视图
编程对象	USER_PROCEDURES、DBA_PROCEDURES、ALL_PROCEDURES、 USER_SOURCE、DBA_SOURCE、ALL_SOURCE、 USER_TRIGGERS、DBA_TRIGGERS、ALL_TRIGGERS、 USER_ERRORS、DBA_ERRORS、ALL_ERRORS、 USER_ARGUMENTS、DBA_ARGUMENTS 和 ALL_ARGUMENTS
安全性	USER_ROLE_PRIVS、DBA_ROLE_PRIVS、ROLE_ROLE_PRIVS、 SESSION_ROLES、 USER_SYS_PRIVS、DBA_SYS_PRIVS、ROLE_SYS_PRIVS、 SESSION_PRIVS、 USER_TAB_PRIVS、DBA_TAB_PRIVS、ALL_TAB_PRIVS、ROLE_TAB_PRIVS、 USER_TAB_PRIVS_MADE、ALL_TAB_PRIVS_MADE、 USER_TAB_PRIVS_RECD、ALL_TAB_PRIVS_RECD 和 DBA_ROLES

示例

以下示例显示如何启用和使用名为 MYDB 的数据库的兼容数据字典的视图并获得相关信息:

- 启用兼容数据字典的视图的创建:

```
db2set DB2_COMPATIBILITY_VECTOR=ORA
db2stop
db2start
db2 create db mydb
```

- 确定哪些兼容数据字典的视图可用:

```
connect to mydb
select * from dictionary
```

- 使用 USER_SYS_PRIVS 视图来显示所有已授予当前用户的系统特权:

```
connect to mydb
select * from user_sys_privs
```

- 确定 DBA_TABLES 视图的列定义:

```
connect to mydb
describe select * from dba_tables
```

第 50 章 术语映射: Oracle 到 DB2 产品

由于适当地设置 DB2 环境后可以使 Oracle 应用程序能够与 DB2 数据服务器配合工作，因此，了解某些 Oracle 概念与 DB2 概念之间的映射关系至关重要。

本节提供了由 Oracle 使用的数据管理概念及这些概念和由 DB2 产品使用的概念之间的相似处和差异的概述。表 41 提供常用 Oracle 术语及其等同的 DB2 术语的简明摘要。

表 41. 常用 Oracle 概念到 DB2 概念的映射

Oracle 概念	DB2 概念	备注
活动日志	活动日志	这些概念相同。
实参	自变量	这些概念相同。
警报日志	db2diag 日志文件和管理通知日志	db2diag 日志文件主要供 IBM Software Support 用于进行故障诊断。管理通知日志主要供数据库和系统管理员用于进行故障诊断。管理通知日志消息也以标准化消息格式记录到 db2diag 日志文件中。
归档日志	脱机归档日志	这些概念相同。
归档日志方式	日志归档	这些概念相同。
background_dump_dest	diagpath	这些概念相同。
已创建全局临时表	已创建全局临时表	这些概念相同。
游标共享	语句集中器	这些概念相同。
数据块	数据页	这些概念相同。
数据缓冲区高速缓存	缓冲池	这些概念相同。但是，在 DB2 产品中，可以存在所需任意数目的具有任意页大小的缓冲池。
数据字典	系统目录	DB2 系统目录包含表和视图形式的元数据。数据库管理器创建并维护两组基于基本系统目录表定义的系统目录视图: **SYSCAT 视图** 只读视图。 **SYSSTAT 视图** 可更新视图，包含优化器所使用的统计信息。
数据字典高速缓存	目录高速缓存	这些概念相同。
数据文件	容器	DB2 数据以物理方式存储在容器中，后者包含对象。
数据库链接	昵称	昵称是标识，指的是远程数据源中的对象，即: 联合数据库对象。

表 41. 常用 Oracle 概念到 DB2 概念的映射 (续)

Oracle 概念	DB2 概念	备注
双重表	双重表	这些概念相同。
动态性能视图	快照监视器 SQL 管理视图	使用 SYSIBMADM 模式的快照监视器 SQL 管理视图返回关于数据库系统的特定区域的监视器数据。例如，SYSIBMADM.SNAPBP SQL 管理视图提供缓冲池信息的快照。
扩展数据块	扩展数据块	DB2 扩展数据块由一组连续的数据页组成。
形参	参数	这些概念相同。
全局索引	非分区索引	这些概念相同。
不活动日志	联机归档日志	这些概念相同。
`init.ora` 文件和服务器参数文件 (SPFILE)	数据库管理器配置文件和数据库配置文件	DB2 实例可以包含多个数据库。因此，同时在实例级别（使用数据库管理器配置文件）和数据库级别（使用数据库配置文件）来存储配置参数以及它们的值。可以通过 **GET DBM CFG** 或 **UPDATE DBM CFG** 命令管理数据库管理器配置文件。可以通过 **GET DB CFG** 或 **UPDATE DB CFG** 命令管理数据库配置文件。
实例	实例或数据库管理器	实例是后台进程与共享内存的组合。DB2 实例也被称为数据库管理器。
大池	实用程序堆	实用程序堆由备份、复原和装入实用程序使用。
库高速缓存	程序包高速缓存	从数据库共享内存中分配的程序包高速缓存用于对数据库上执行的静态和动态 SQL 语句以及 XQuery 语句的节进行高速缓存。
局部索引	分区索引	这是同一个概念。
具体化视图	具体化查询表（MQT）	MQT 是一个表，其定义基于某个查询的结果，并有助于提高性能。DB2 SQL 编译器将确定查询对 MQT 运行时的效率是否高于对该 MQT 所基于的基本表运行时的效率。
不归档日志方式	循环日志记录	这些概念相同。

表 41. 常用 Oracle 概念到 DB2 概念的映射 (续)

Oracle 概念	DB2 概念	备注
Oracle 调用接口 (OCI) Oracle 调用接口 (OCI)	DB2CI	DB2CI 是一个 C 和 C++ 应用程序编程接口，此接口使用函数调用来连接至 DB2 数据库、管理游标和执行 SQL 语句。请参阅第 455 页的第 51 章，『IBM 数据服务器 DB2CI 驱动程序』以获取 DB2CI 驱动程序支持的 OCI API 列表。
Oracle 调用接口 (OCI) Oracle 调用接口 (OCI)	调用级接口 (CLI)	CLI 是 C 和 C++ 应用程序编程接口，此接口使用函数调用将动态 SQL 语句作为函数自变量进行传递。在大多数情况下，可以将 OCI 函数替换为 CLI 函数以及对支持程序代码所作的相关更改。
ORACLE_SID 环境变量	**DB2INSTANCE** 环境变量	这些概念相同。
分区表	分区表	这些概念相同。
过程语言/结构化查询语言 (PL/SQL)	SQL 过程语言 (SQL PL)	SQL PL 是对 SQL 的扩展，它由语句和其他语言元素组成。SQL PL 提供了用于声明变量和条件处理程序、用于对变量赋值以及用于实现过程逻辑的语句。SQL PL 是 SQL/持久存储模块 (SQL/PSM) 语言标准的子集。 您可以使用 DB2 接口编译和执行 Oracle PL/SQL 语句。
程序全局区域（PGA）	应用程序共享内存和代理程序专用内存	应用程序共享内存存储在数据库与特定应用程序之间共享的信息：主要是正在传递到数据库以及正在从数据库传递的数据行。代理程序专用内存存储用于为特定应用程序提供服务的信息，例如排序堆、游标信息和会话上下文。
重做日志	事务日志	事务日志可记录数据库事务。您可将其用于恢复。
角色	角色	这些概念相同。
段	存储器对象	这些概念相同。
会话	会话；数据库连接	这些概念相同。
startup nomount 命令	**db2start** 命令	用于启动实例的命令。

表 41. 常用 Oracle 概念到 DB2 概念的映射 (续)

Oracle 概念	DB2 概念	备注
同义词	别名	别名是表、视图、昵称或另一个别名的备用名称。可以指定词汇*同意词*，而不是*别名*。别名并非用于控制由应用程序使用的 DB2 过程或用户定义的函数的版本。要控制版本，请使用 SET PATH 语句将所需的模式添加到 CURRENT PATH 专用寄存器的值。
系统全局区域（SGA）	实例共享内存和数据库共享内存	实例共享内存存储特定实例的所有信息，例如所有活动连接的列表以及安全性信息。数据库共享内存存储特定数据库的信息，例如程序包高速缓存、日志缓冲区和缓冲池。
SYSTEM 表空间	SYSCATSPACE 表空间	SYSCATSPACE 表空间包含系统目录。缺省情况下，此表空间是在您创建数据库时创建的。
表空间	表空间	这些概念相同。
用户全局区域（UGA）	应用程序全局内存	应用程序全局内存由应用程序共享内存和特定于应用程序的内存组成。

第 5 部分 DB2CI 应用程序开发

DB2CI 是 DB2 版本 9.7 数据库服务器的可调用 SQL 接口。这是用于进行 DB2 数据库访问的“C”和“C++”应用程序编程接口，此接口使用函数调用来连接至数据库、管理游标和执行 SQL 语句。

从版本 9.7 修订包 1 开始，可以使用 DB2CI 接口来访问任何受支持操作系统上的 DB2 版本 9.7 服务器上的数据库。

DB2CI 接口支持许多 Oracle 调用接口（OCI） API。此支持能够降低启用现有 OCI 应用程序的复杂性，以使这些应用程序能够与 DB2 数据库一起使用。IBM 数据服务器 DB2CI 驱动程序是 DB2CI 接口的驱动程序。

第 51 章 IBM 数据服务器 DB2CI 驱动程序

IBM 数据服务器 DB2CI 驱动程序 支持进行 DB2CI 应用程序开发。

IBM 数据服务器客户机包括 DB2CI 驱动程序。需要安装此客户机才能安装 DB2CI 驱动程序。

DB2CI 驱动程序支持调用下列 OCI API。

表 42. DB2CI 驱动程序支持

受支持的 OCI API		
OCIAttrGet	OCILobGetLength	OCINumberTan
OCIAttrSet	OCILobIsEqual	OCINumberToInt
OCIBindArrayOfStruct	OCILobIsTemporary	OCINumberToReal
OCIBindByName	OCILobIsOpen	OCINumberToRealArray
OCIBindByPos	OCILobLocatorAssign	OCINumberToText
OCIBindDynamic	OCILobLocatorIsInit	OCINumberTrunc
OCIBreak	OCILobRead	OCIParamGet
OCIClientVersion	OCILobTrim	OCIParamSet
OCIDateAddDays	OCILobWrite	OCIPasswordChange
OCIDateAddMonths	OCILogoff	OCIPing
OCIDateAssign	OCILogon	OCIRawAllocSize
OCIDateCheck	OCILogon2	OCIRawAssignBytes
OCIDateCompare	OCINumberAbs	OCIRawAssignRaw
OCIDateDaysBetween	OCINumberAdd	OCIRawPtr
OCIDateFromText	OCINumberArcCos	OCIRawResize
OCIDateLastDay	OCINumberArcSin	OCIRawSize
OCIDateNextDay	OCINumberArcTan	OCIReset
OCIDateSysDate	OCINumberArcTan2	OCIResultSetToStmt
OCIDateToText	OCINumberAssign	OCIServerAttach
OCIDefineArrayOfStruct	OCINumberCeil	OCIServerDetach
OCIDefineByPos	OCINumberCmp	OCIServerVersion
OCIDefineDynamic	OCINumberCos	OCISessionBegin
OCIDescribeAny	OCINumberDec	OCISessionEnd
OCIDescriptorAlloc	OCINumberDiv	OCISessionGet
OCIDescriptorFree	OCINumberExp	OCISessionRelease
OCIEnvCreate	OCINumberFloor	OCIStmtExecute
OCIEnvInit	OCINumberFromInt	OCIStmtFetch
OCIErrorGet	OCINumberFromReal	OCIStmtFetch2
OCIFileClose	OCINumberFromText	OCIStmtGetBindInfo
OCIFileExists	OCINumberHypCos	OCIStmtGetPieceInfo
OCIFileFlush	OCINumberHypSin	OCIStmtPrepare

表 42. DB2CI 驱动程序支持 (续)

受支持的 OCI API		
OCIFileGetLength	OCINumberHypTan	OCIStmtPrepare2
OCIFileInit	OCINumberInc	OCIStmtRelease
OCIFileOpen	OCINumberIntPower	OCIStmtSetPieceInfo
OCIFileRead	OCINumberIsInt	OCIStringAllocSize
OCIFileSeek	OCINumberIsZero	OCIStringAssign
OCIFileTerm	OCINumberLn	OCIStringAssignText
OCIFileWrite	OCINumberLog	OCIStringPtr
OCIHandleAlloc	OCINumberMod	OCIStringResize
OCIHandleFree	OCINumberMul	OCIStringSize
OCIInitialize	OCINumberNeg	OCITerminate
OCILobAppend	OCINumberPower	OCITransCommit
OCILobAssign	OCINumberPrec	OCITransDetach
OCILobClose	OCINumberRound	OCITransForget
OCILobCopy	OCINumberSetPi	OCITransMultiPrepare
OCILobCreateTemporary	OCINumberSetZero	OCITransPrepare
OCILobDisableBuffering	OCINumberShift	OCITransRollback
OCILobEnableBuffering	OCINumberSign	OCITransStart
OCILobErase	OCINumberSin	xaoEnv
OCILobFreeTemporary	OCINumberSqrt	xaosterr
OCILobFlushBuffer	OCINumberSub	xaoSvcCtx

第 52 章 构建 DB2CI 应用程序

可以使用现有 Oracle 调用接口 (OCI)应用程序和 **bldapp** 脚本文件来构建 DB2CI 应用程序。

开始之前

- 必须具有与现有 OCI 应用程序所使用的 Oracle 数据库有相同结构的 DB2 数据库。
- 必须已经安装了 IBM 数据服务器客户机。

关于此任务

DB2 样本提供了一个称为 bldapp 的脚本，用于编译和链接使用 IBM 数据服务器 DB2CI 驱动程序支持的 OCI 函数的应用程序。它与样本程序一起位于 *DB2DIR*\samples\db2ci 或者 *DB2DIR*/samples/db2ci 目录中。*DB2DIR* 表示安装了 DB2 副本的位置。

bldapp 脚本文件最多可以采用四个参数。第一个参数 $1 指定源文件的名称。仅当构建需要连接至数据库的嵌入式 SQL 程序时才需要其他参数: 第二个参数 $2 指定要连接的数据库的名称; 第三个参数 $3 指定数据库的用户标识，而 $4 则指定密码。如果程序中包含由 .sqc 扩展名指示的嵌入式 SQL，那么将调用 embprep 脚本以预编译此程序，并生成一个具有 .c 扩展名的程序文件。

限制

- 请确保现有 OCI 应用程序仅调用了 DB2CI 驱动程序支持的 OCI 函数。请参阅第 455 页的第 51 章，『IBM 数据服务器 DB2CI 驱动程序』以获取受支持 OCI 函数的完整列表。

过程

1. 如果要使用现有 OCI 应用程序来构建 DB2CI 应用程序，那么请务必指定 db2ci.h 包含文件。
2. 使用 bldapp 脚本文件来构建 DB2CI 应用程序 以下示例说明如何在 Linux 和 UNIX 操作系统上从源文件 tbinfo.c 来构建样本程序 tbinfo:

   ```
   cd $INSTHOME/sqllib/samples/db2ci
   bldapp tbinfo
   ```

   这将生成可执行文件 tbinfo。
3. 通过按如下所示输入可执行文件名称来运行在上一步生成的可执行文件:

   ```
   tbinfo
   ```

DB2CI 应用程序编译和链接选项（AIX）

使用 AIX® IBM C 编译器构建 DB2CI 应用程序时，建议使用本主题中的编译和链接选项。

可以在 *DB2DIR*/samples/cli/bldapp 批处理文件中找到下列选项，其中 *DB2DIR* 是安装了 DB2 副本的位置。

编译选项:

xlc IBM C 编译器。

$EXTRA_CFLAG
对于 64 位环境，包含值“-q64”；否则，不包含值。

-I$DB2PATH/include
指定 DB2 包含文件的位置。例如: `$HOME/sqllib/include`。

-c 只执行编译；不进行链接。此脚本包含独立的编译和链接步骤。

链接选项:

xlc 使用编译器作为链接程序的前端。

$EXTRA_CFLAG
对于 64 位环境，包含值“-q64”；否则，不包含值。

-o $1 指定可执行程序。

$1.o 指定对象文件。

utilci.o
包括用于执行错误检查的实用程序对象文件。

-L$DB2PATH/$LIB
指定 DB2 运行时共享库的位置。例如: `$HOME/sqllib/$LIB`。如果未指定 `-L` 选项，那么编译器将采用以下路径: `/usr/lib:/lib`。

-ldb2ci
与 DB2CI 库链接。

DB2CI 应用程序编译和链接选项 (HP-UX)

使用 HP-UX C 编译器构建 DB2CI 应用程序时，建议使用本主题中的编译和链接选项。

可以在 *DB2DIR*/samples/db2ci/bldapp 批处理文件中找到下列选项，其中 *DB2DIR* 是安装了 DB2 副本的位置。

编译选项:

cc 使用 C 编译器。

$EXTRA_CFLAG
如果 HP-UX 平台为 IA64 并且启用了 64 位支持，那么此标志包含值 **+DD64**；如果启用了 32 位支持，那么此标志包含值 **+DD32**。如果 HP-UX 平台启用了 PA-RISC 和 64 位支持，那么此标志包含值 **+DA2.0W**。对于 PA-RISC 平台上的 32 位支持，此标志包含值 **+DA2.0N**。

+DD64 必须使用此选项才能为 IA64 上的 HP-UX 生成 64 位代码。

+DD32 必须使用此选项才能为 IA64 上的 HP-UX 生成 32 位代码。

+DA2.0W
必须使用此选项才能为 PA-RISC 上的 HP-UX 生成 64 位代码。

+DA2.0N
必须使用此选项才能为 PA-RISC 上的 HP-UX 生成 32 位代码。

-Ae 启用 HP ANSI 扩展方式。

-I$DB2PATH/include
指定 DB2 包含文件的位置。例如: `$HOME/sqllib/include`。

-c 只执行编译; 不进行链接。编译和链接是不同的步骤。

链接选项:

cc 使用编译器作为链接程序的前端。

$EXTRA_CFLAG
如果 HP-UX 平台为 IA64 并且启用了 64 位支持, 那么此标志包含值 **+DD64**; 如果启用了 32 位支持, 那么此标志包含值 **+DD32**。如果 HP-UX 平台启用了 PA-RISC 和 64 位支持, 那么此标志包含值 **+DA2.0W**。对于 PA-RISC 平台上的 32 位支持, 此标志包含值 **+DA2.0N**。

+DD64 必须使用此选项才能为 IA64 上的 HP-UX 生成 64 位代码。

+DD32 必须使用此选项才能为 IA64 上的 HP-UX 生成 32 位代码。

+DA2.0W
必须使用此选项才能为 PA-RISC 上的 HP-UX 生成 64 位代码。

+DA2.0N
必须使用此选项才能为 PA-RISC 上的 HP-UX 生成 32 位代码。

-o $1 指定可执行程序。

$1.o 指定对象文件。

utilci.o
包括用于执行错误检查的实用程序对象文件。

$EXTRA_LFLAG
指定运行时路径。如果设置了此选项, 那么对于 32 位, 它包含值 `-Wl,+b$HOME/sqllib/lib32`, 对于 64 位, 它包含值 `-Wl,+b$HOME/sqllib/lib64`。如果未设置此选项, 那么它不包含任何值。

-L$DB2PATH/$LIB
指定 DB2 运行时共享库的位置。对于 32 位: `$HOME/sqllib/lib32`; 对于 64 位: `$HOME/sqllib/lib64`。

-ldb2ci
与 DB2CI 库链接。

DB2CI 应用程序编译和链接选项 (Linux)

使用 GNU/Linux gcc 编译器构建 DB2CI 应用程序时, 建议使用本主题中的编译和链接选项。

可以在 *DB2DIR*/samples/db2ci/bldapp 批处理文件中找到下列选项, 其中 *DB2DIR* 是安装了 DB2 副本的位置。

编译选项:

gcc C 编译器。

$EXTRA_C_FLAGS
包含下列其中一项:

- -m31（仅限于 Linux for zSeries®），用于构建 32 位库;
- -m32（仅限于 Linux for x86、x64 和 POWER®），用于构建 32 位库;
- -m64（仅限于 Linux for zSeries、POWER 和 x64），用于构建 64 位库；或者
- 不包含任何值（Linux for IA64），用于构建 64 位库。

-I$DB2PATH/include
指定 DB2 包含文件的位置。例如: $HOME/sqllib/include。

-c 只执行编译；不进行链接。编译和链接是不同的步骤。

链接选项:

gcc 使用编译器作为链接程序的前端。

$EXTRA_C_FLAGS
包含下列其中一项:

- -m31（仅限于 Linux for zSeries），用于构建 32 位库;
- -m32（仅限于 Linux for x86、x64 和 POWER），用于构建 32 位库;
- -m64（仅限于 Linux for zSeries、POWER 和 x64），用于构建 64 位库；或者
- 不包含任何值（Linux for IA64），用于构建 64 位库。

-o $1 指定可执行文件。

$1.o 包括程序对象文件。

utilci.o
包括用于执行错误检查的实用程序对象文件。

$EXTRA_LFLAG
对于 32 位，包含值“-Wl,-rpath,$DB2PATH/lib32”；对于 64 位，包含值“-Wl,-rpath,$DB2PATH/lib64”。

-L$DB2PATH/$LIB
指定 DB2 静态库和共享库在链接时的位置。例如，对于 32 位: $HOME/sqllib/lib32；对于 64 位: $HOME/sqllib/lib64。

-ldb2ci
与 DB2CI 库链接。

DB2CI 应用程序编译和链接选项（Solaris）

使用 Solaris C 编译器构建 DB2CI 应用程序时，建议使用本主题中的编译和链接选项。

可以在 *DB2DIR*/samples/db2ci/bldapp 批处理文件中找到下列选项，其中 *DB2DIR* 是安装了 DB2 副本的位置。

编译选项:

cc 使用 C 编译器。

-xarch=$CFLAG_ARCH
此选项确保与 libdb2.so 链接时，编译器生成有效的可执行文件。$CFLAG_ARCH 的值的设置如下所示:

- “v8plusa”: Solaris SPARC 上的 32 位应用程序
- “v9”: Solaris SPARC 上的 64 位应用程序
- “sse2”: Solaris x64 上的 32 位应用程序
- “amd64”: Solaris x64 上的 64 位应用程序

-I$DB2PATH/include
指定 DB2 包含文件的位置。例如: $HOME/sqllib/include。

-c 只执行编译；不进行链接。此脚本包含独立的编译和链接步骤。

链接选项:

cc 使用编译器作为链接程序的前端。

-xarch=$CFLAG_ARCH
此选项确保与 libdb2.so 链接时，编译器生成有效的可执行文件。$CFLAG_ARCH 的值设置为“v8plusa”（表示 32 位）或“v9”（表示 64 位）。

-mt 在支持多线程的情况下进行链接，以防止在调用 fopen 时发生问题。

注: 如果使用 POSIX 线程，那么 DB2 应用程序还必须使用 -lpthread 进行链接，而无论它们是否线程化应用程序。

-o $1 指定可执行程序。

$1.o 包括程序对象文件。

utilci.o
包括用于执行错误检查的实用程序对象文件。

-L$DB2PATH/$LIB
指定 DB2 静态库和共享库在链接时的位置。例如，对于 32 位: $HOME/sqllib/lib32；对于 64 位: $HOME/sqllib/lib64

$EXTRA_LFLAG
指定 DB2 共享库在运行时的位置。对于 32 位，包含值“-R$DB2PATH/lib32”；对于 64 位，包含值“-R$DB2PATH/lib64”。

-ldb2ci
与 DB2CI 库链接。

DB2CI 应用程序编译和链接选项（Windows）

使用 Microsoft Visual C++ 编译器构建 DB2CI 应用程序时，建议使用本主题中的编译和链接选项。

可以在 *DB2DIR*\samples\db2ci\bldapp.bat 批处理文件中找到下列选项，其中 *DB2DIR* 是安装了 DB2 副本的位置。

编译选项:

%BLDCOMP%
编译器的变量。缺省值为 cl，即 Microsoft Visual C++ 编译器。另外，还可以将其设置为 icl（用于 32 位和 64 位应用程序的 Intel C++ 编译器）或 ecl（用于 Itanium 64 位应用程序的 Intel C++ 编译器）。

-Zi 启用调试信息。

-Od 禁止优化。在关闭优化的情况下使用调试器较为容易。

-c 只执行编译；不进行链接。

-W2 设置警告级别。

-DWIN32
Windows 操作系统所必需的编译器选项。

链接选项:

link 使用链接程序。

-debug 包括调试信息。

-out:%1.exe
指定可执行文件。

%1.obj 包括对象文件。

db2ci.lib 或 db2ci64.lib
链接至 DB2CI 库。对于 Windows 32 位操作系统，请使用 db2ci.lib。对于 Windows 64 位操作系统，请使用 db2ci64.lib。

请参阅编译器文档，以了解其他编译器选项。

第 6 部分 附录

附录 A. DB2 技术信息概述

DB2 技术信息以多种可以通过多种方法访问的格式提供。

您可以通过下列工具和方法获得 DB2 技术信息:

- DB2 信息中心
  - 主题（任务、概念和参考主题）
  - 样本程序
  - 教程
- DB2 书籍
  - PDF 文件（可下载）
  - PDF 文件（在 DB2 PDF DVD 中）
  - 印刷版书籍
- 命令行帮助
  - 命令帮助
  - 消息帮助

注: DB2 信息中心主题的更新频率比 PDF 书籍或硬拷贝书籍的更新频率高。要获取最新信息，请安装可用的文档更新或者参阅 ibm.com 上的 DB2 信息中心。

您可以在线访问 ibm.com 上的其他 DB2 技术信息，例如技术说明、白皮书和 IBM Redbooks® 出版物。请访问以下网址处的 DB2 信息管理软件资料库站点: http://www.ibm.com/software/data/sw-library/。

文档反馈

我们非常重视您对 DB2 文档的反馈。如果您想就如何改善 DB2 文档提出建议，请向 db2docs@ca.ibm.com 发送电子邮件。DB2 文档小组将阅读您的所有反馈，但无法直接给您答复。请尽可能提供具体的示例，这样我们才能更好地了解您所关心的问题。如果您要提供有关具体主题或帮助文件的反馈，请加上标题和 URL。

请不要使用以上电子邮件地址与 DB2 客户支持机构联系。如果您遇到文档无法解决的 DB2 技术问题，请与您当地的 IBM 服务中心联系以获得帮助。

硬拷贝或 PDF 格式的 DB2 技术库

下列各表描述 IBM 出版物中心（网址为 www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss）所提供的 DB2 资料库。可从 www.ibm.com/support/docview.wss?rs=71&uid=swg2700947 下载 PDF 格式的 DB2 V10.1 手册的英文版本和翻译版本。

尽管这些表标识书籍有印刷版，但可能未在您所在国家或地区提供。

每次更新手册时，表单号都会递增。确保您正在阅读下面列示的手册的最新版本。

注: *DB2 信息中心*的更新频率比 PDF 或硬拷贝书籍的更新频率高。

表 43. DB2 技术信息

书名	书号	是否提供印刷版	最近一次更新时间
Administrative API Reference	SC27-3864-00	是	2012 年 4 月
Administrative Routines and Views	SC27-3865-00	否	2012 年 4 月
Call Level Interface Guide and Reference Volume 1	SC27-3866-00	是	2012 年 4 月
Call Level Interface Guide and Reference Volume 2	SC27-3867-00	是	2012 年 4 月
Command Reference	SC27-3868-00	是	2012 年 4 月
数据库管理概念和配置参考	S151-1758-00	是	2012 年 4 月
Data Movement Utilities Guide and Reference	S151-1756-00	是	2012 年 4 月
数据库监视指南和参考	S151-1759-00	是	2012 年 4 月
数据恢复及高可用性指南与参考	S151-1755-00	是	2012 年 4 月
数据库安全性指南	S151-1753-01	是	2012 年 4 月
DB2 Workload Management Guide and Reference	SC27-3891-00	是	2012 年 4 月
开发 ADO.NET 和 OLE DB 应用程序	S151-1765-00	是	2012 年 4 月
开发嵌入式 SQL 应用程序	S151-1763-00	是	2012 年 4 月
Developing Java Applications	SC27-3875-00	是	2012 年 4 月
Developing Perl, PHP, Python, and Ruby on Rails Applications	SC27-3876-00	否	2012 年 4 月
开发用户定义的例程（SQL 和外部例程）	S151-1761-00	是	2012 年 4 月
数据库应用程序开发入门	G151-1764-00	是	2012 年 4 月
Linux 和 Windows 上的 DB2 安装和管理入门	G151-1769-00	是	2012 年 4 月
全球化指南	S151-1757-00	是	2012 年 4 月
安装 DB2 服务器	G151-1768-00	是	2012 年 4 月
安装 IBM Data Server Client	G151-1751-00	否	2012 年 4 月
消息参考第 1 卷	S151-1767-00	否	2012 年 4 月
消息参考第 2 卷	S151-1766-00	否	2012 年 4 月
Net Search Extender 管理和用户指南	S151-1078-00	否	2012 年 4 月

表 43. DB2 技术信息 (续)

书名	书号	是否提供印刷版	最近一次更新时间
分区和集群指南	S151-1754-00	是	2012 年 4 月
pureXML 指南	S151-1775-00	是	2012 年 4 月
Spatial Extender User's Guide and Reference	SC27-3894-00	否	2012 年 4 月
SQL 过程语言: 应用程序启用和支持	S151-1762-00	是	2012 年 4 月
SQL Reference Volume 1	SC27-3885-00	是	2012 年 4 月
SQL Reference Volume 2	SC27-3886-00	是	2012 年 4 月
Text Search Guide	SC27-3888-00	是	2012 年 4 月
故障诊断和调整数据库性能	S151-1760-00	是	2012 年 4 月
升级到 DB2 V10.1	S151-1770-00	是	2012 年 4 月
DB2 V10.1 新增内容	S151-1752-00	是	2012 年 4 月
XQuery 参考	S151-1774-00	否	2012 年 4 月

表 44. 特定于 DB2 Connect 的技术信息

书名	书号	是否提供印刷版	最近一次更新时间
DB2 Connect 安装和配置 DB2 Connect Personal Edition	S151-1773-00	是	2012 年 4 月
DB2 Connect 安装和配置 DB2 Connect 服务器	S151-1772-00	是	2012 年 4 月
DB2 Connect 用户指南	S151-1771-00	是	2012 年 4 月

从命令行处理器显示 SQL 状态帮助

DB2 产品针对可能充当 SQL 语句结果的条件返回 SQLSTATE 值。SQLSTATE 帮助说明 SQL 状态和 SQL 状态类代码的含义。

过程

要启动 SQL 状态帮助，请打开命令行处理器并输入:

```
? sqlstate or ? class code
```

其中，*sqlstate* 表示有效的 5 位 SQL 状态，*class code* 表示该 SQL 状态的前 2 位。例如，`? 08003` 显示 08003 SQL 状态的帮助，而 `? 08` 显示 08 类代码的帮助。

访问不同版本的 DB2 信息中心

您可以在 ibm.com® 上的不同信息中心中找到其他版本 DB2 产品的文档。

关于此任务

对于 DB2 V10.1 主题，*DB2 信息中心* URL 是 http://publib.boulder.ibm.com/infocenter/db2luw/v10r1。

对于 DB2 V9.8 主题，*DB2 信息中心* URL 是 http://publib.boulder.ibm.com/infocenter/db2luw/v9r8/。

对于 DB2 V9.7 主题，*DB2 信息中心* URL 是 http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/。

对于 DB2 V9.5 主题，*DB2 信息中心* URL 是 http://publib.boulder.ibm.com/infocenter/db2luw/v9r5。

对于 DB2 V9.1 主题，*DB2 信息中心* URL 是 http://publib.boulder.ibm.com/infocenter/db2luw/v9/。

对于 DB2 V8 主题，请转至 *DB2 信息中心* URL: http://publib.boulder.ibm.com/infocenter/db2luw/v8/。

更新安装在计算机或内部网服务器上的 DB2 信息中心

安装在本地的 DB2 信息中心必须定期进行更新。

开始之前

必须已安装 DB2 V10.1 信息中心。有关详细信息，请参阅安装 DB2 服务器中的“使用 DB2 安装向导来安装 DB2 信息中心”主题。所有适用于安装信息中心的先决条件和限制同样适用于更新信息中心。

关于此任务

可以自动或手动更新现有的 DB2 信息中心:

- 自动更新将更新现有的信息中心功能部件和语言。自动更新的一个优点是，与手动更新相比，信息中心的不可用时间较短。另外，自动更新可设置为作为定期运行的其他批处理作业的一部分运行。
- 可以使用手动更新方法来更新现有的信息中心功能部件和语言。自动更新可以缩短更新过程中的停机时间，但如果您想添加功能部件或语言，那么必须执行手动过程。例如，如果本地信息中心最初安装的是英语和法语版，而现在还要安装德语版；那么手动更新将安装德语版，并更新现有信息中心的功能和语言。但是，手动更新要求您手动停止、更新和重新启动信息中心。在整个更新过程期间信息中心不可用。在自动更新过程中，信息中心仅在更新完成后停止工作以重新启动信息中心。

此主题详细说明了自动更新的过程。有关手动更新的指示信息，请参阅“手动更新安装在您的计算机或内部网服务器上的 DB2 信息中心”主题。

过程

要自动更新安装在计算机或内部网服务器上的 DB2 信息中心:

1. 在 Linux 操作系统上，
   a. 浏览至信息中心的安装位置。缺省情况下，DB2 信息中心安装在 `/opt/ibm/db2ic/V10.1` 目录中。
   b. 从安装目录浏览至 `doc/bin` 目录。
   c. 运行 `update-ic` 脚本:

update-ic

2. 在 Windows 操作系统上，
   a. 打开命令窗口。
   b. 浏览至信息中心的安装位置。缺省情况下，DB2 信息中心安装在 <Program Files>\IBM\DB2 Information Center\V10.1 目录中，其中 <Program Files> 表示 Program Files 目录的位置。
   c. 从安装目录浏览至 doc\bin 目录。
   d. 运行 update-ic.bat 文件:

      update-ic.bat

结果

DB2 信息中心将自动重新启动。如果更新可用，那么信息中心会显示新的以及更新后的主题。如果信息中心更新不可用，那么会在日志中添加消息。日志文件位于 doc\eclipse\configuration 目录中。日志文件名称是随机生成的编号。例如，1239053440785.log。

手动更新安装在计算机或内部网服务器上的 DB2 信息中心

如果您已在本地安装 DB2 信息中心，那么可从 IBM 获取文档更新并进行安装。

关于此任务

手动更新安装在本地的 *DB2 信息中心*要求您:

1. 停止计算机上的 *DB2 信息中心*，然后以独立方式重新启动信息中心。如果以独立方式运行信息中心，那么网络上的其他用户将无法访问信息中心，因而您可以应用更新。DB2 信息中心的工作站版本总是以独立方式运行。
2. 使用“更新”功能部件来查看可用的更新。如果有您必须安装的更新，那么请使用“更新”功能部件来获取并安装这些更新。

   注: 如果您的环境要求在一台未连接至因特网的机器上安装 *DB2 信息中心*更新，请使用一台已连接至因特网并已安装 *DB2 信息中心*的机器将更新站点镜像至本地文件系统。如果网络中有许多用户将安装文档更新，那么可以通过在本地也为更新站点制作镜像并为更新站点创建代理来缩短每个人执行更新所需要的时间。
   如果提供了更新包，请使用“更新”功能部件来获取这些更新包。但是，只有在单机方式下才能使用“更新”功能部件。
3. 停止独立信息中心，然后在计算机上重新启动 *DB2 信息中心*。

注: 在 Windows 2008、Windows Vista 和更高版本上，稍后列示在此部分的命令必须作为管理员运行。要打开具有全面管理员特权的命令提示符或图形工具，请右键单击快捷方式，然后选择**以管理员身份运行**。

过程

要更新安装在您的计算机或内部网服务器上的 *DB2 信息中心*:

1. 停止 *DB2 信息中心*。
   - 在 Windows 上，单击**开始 > 控制面板 > 管理工具 > 服务**。右键单击 **DB2 信息中心**服务，并选择**停止**。

- 在 Linux 上，输入以下命令:

  ```
  /etc/init.d/db2icdv10 stop
  ```

2. 以独立方式启动信息中心。
   - 在 Windows 上:
     a. 打开命令窗口。
     b. 浏览至信息中心的安装位置。缺省情况下，*DB2 信息中心*安装在 *Program_Files*\IBM\DB2 Information Center\V10.1 目录中，其中 *Program Files* 表示 Program Files 目录的位置。
     c. 从安装目录浏览至 doc\bin 目录。
     d. 运行 help_start.bat 文件:

        ```
        help_start.bat
        ```

   - 在 Linux 上:
     a. 浏览至信息中心的安装位置。缺省情况下，*DB2 信息中心*安装在 /opt/ibm/db2ic/V10.1 目录中。
     b. 从安装目录浏览至 doc/bin 目录。
     c. 运行 help_start 脚本:

        ```
        help_start
        ```

   系统缺省 Web 浏览器将打开以显示独立信息中心。
3. 单击**更新**按钮（  ）。（必须在浏览器中启用 JavaScript。）在信息中心的右边面板上，单击**查找更新**。将显示现有文档的更新列表。
4. 要启动安装过程，请检查您要安装的选项，然后单击**安装更新**。
5. 在安装进程完成后，请单击**完成**。
6. 要停止独立信息中心，请执行下列操作:
   - 在 Windows 上，浏览至安装目录中的 doc\bin 目录并运行 help_end.bat 文件:

     ```
     help_end.bat
     ```

     注: help_end 批处理文件包含安全地停止使用 help_start 批处理文件启动的进程所需的命令。不要使用 Ctrl-C 或任何其他方法来停止 help_start.bat。
   - 在 Linux 上，浏览至安装目录中的 doc/bin 目录并运行 help_end 脚本:

     ```
     help_end
     ```

     注: help_end 脚本包含安全地停止使用 help_start 脚本启动的进程所需的命令。不要使用任何其他方法来停止 help_start 脚本。
7. 重新启动 *DB2 信息中心*。
   - 在 Windows 上，单击**开始 > 控制面板 > 管理工具 > 服务**。右键单击 **DB2 信息中心**服务，并选择**启动**。
   - 在 Linux 上，输入以下命令:

     ```
     /etc/init.d/db2icdv10 start
     ```

结果

更新后的 *DB2 信息中心*将显示新的以及更新后的主题。

DB2 教程

DB2 教程帮助您了解 DB2 数据库产品的各个方面。这些课程提供了逐步指示信息。

开始之前

您可以在信息中心中查看 XHTML 版的教程: http://publib.boulder.ibm.com/infocenter/db2luw/v10r1/。

某些课程使用了样本数据或代码。有关其特定任务的任何先决条件的描述，请参阅教程。

DB2 教程

要查看教程，请单击标题。

*pureXML 指南*中的『**pureXML®**』

设置 DB2 数据库以存储 XML 数据以及对本机 XML 数据存储器执行基本操作。

DB2 故障诊断信息

我们提供了各种各样的故障诊断和问题确定信息来帮助您使用 DB2 数据库产品。

DB2 文档

您可以在*故障诊断和调整数据库性能*或者 *DB2 信息中心*的“数据库基础”部分中找到故障诊断信息，这些信息包含以下内容:

- 有关如何使用 DB2 诊断工具和实用程序来隔离和确定问题的信息。
- 一些最常见问题的解决方案。
- 旨在帮助您解决 DB2 数据库产品使用过程中可能会遇到的其他问题的建议。

IBM 支持门户网站

如果您遇到问题并且希望得到帮助以查找可能的原因和解决方案，请访问 IBM 支持门户网站。这个技术支持站点提供了指向最新 DB2 出版物、技术说明、授权程序分析报告（APAR 或错误修订）、修订包和其他资源的链接。可搜索此知识库并查找问题的可能解决方案。

访问 IBM 支持门户网站: http://www.ibm.com/support/entry/portal/Overview/Software/Information_Management/DB2_for_Linux,_UNIX_and_Windows

信息中心条款和条件

如果符合以下条款和条件，那么授予您使用这些出版物的许可权。

适用性: 用户需要遵循 IBM Web 站点的使用条款及以下条款和条件。

个人使用: 只要保留所有的专有权声明，您就可以为个人、非商业使用复制这些出版物。未经 IBM 明确同意，您不可以分发、展示或制作这些出版物或其中任何部分的演绎作品。

商业使用: 只要保留所有的专有权声明，您就可以仅在企业内复制、分发和展示这些出版物。未经 IBM 明确同意，您不可以制作这些出版物的演绎作品，或者在您的企业外部复制、分发或展示这些出版物或其中的任何部分。

权利：除非本许可权中明确授予，否则不得授予对这些出版物或其中包含的任何信息、数据、软件或其他知识产权的任何许可权、许可证或权利，无论是明示的还是暗含的。

IBM 保留根据自身的判断，认为对出版物的使用损害了 IBM 的权益（由 IBM 自身确定）或未正确遵循以上指示信息时，撤回此处所授予权限的权利。

只有您完全遵循所有适用的法律和法规，包括所有的美国出口法律和法规，您才可以下载、出口或再出口该信息。

IBM 对这些出版物的内容不作任何保证。这些出版物“按现状”提供，不附有任何种类的（无论是明示的还是暗含的）保证，包括但不限于暗含的关于适销和适用于某种特定用途的保证。

IBM Trademarks: IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at www.ibm.com/legal/copytrade.shtml

附录 B. 声明

本信息是为在美国提供的产品和服务编写的。有关非 IBM 产品的信息是基于首次出版此文档时的可获信息且会随时更新。

IBM 可能在其他国家或地区不提供本文档中讨论的产品、服务或功能特性。有关您当前所在区域的产品和服务的信息，请向您当地的 IBM 代表咨询。任何对 IBM 产品、程序或服务的引用并非意在明示或暗示只能使用 IBM 的产品、程序或服务。只要不侵犯 IBM 的知识产权，任何同等功能的产品、程序或服务，都可以代替 IBM 产品、程序或服务。但是，评估和验证任何非 IBM 产品、程序或服务，则由用户自行负责。

IBM 公司可能已拥有或正在申请与本文档内容有关的各项专利。提供本文档并未授予用户使用这些专利的任何许可。您可以用书面方式将许可查询寄往:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

有关双字节字符集 (DBCS) 信息的许可查询，请与您所在国家或地区的 IBM 知识产权部门联系，或用书面方式将查询寄往:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

本条款不适用英国或任何这样的条款与当地法律不一致的国家或地区: International Business Machines Corporation“按现状”提供本出版物，不附有任何种类的（无论是明示的还是暗含的）保证，包括但不限于暗含的有关非侵权、适销和适用于某种特定用途的保证。某些国家或地区在某些交易中不允许免除明示或暗含的保证。因此本条款可能不适用于您。

本信息中可能包含技术方面不够准确的地方或印刷错误。此处的信息将定期更改；这些更改将编入本资料的新版本中。IBM 可以随时对本资料中描述的产品和/或程序进行改进和/或更改，而不另行通知。

本信息中对非 IBM Web 站点的任何引用都只是为了方便起见才提供的，不以任何方式充当对那些 Web 站点的保证。那些 Web 站点中的资料不是此 IBM 产品资料的一部分，使用那些 Web 站点带来的风险将由您自行承担。

IBM 可以按它认为适当的任何方式使用或分发您所提供的任何信息而无须对您承担任何责任。

本程序的被许可方如果要了解有关程序的信息以达到如下目的: (i) 允许在独立创建的程序和其他程序（包括本程序）之间进行信息交换，以及 (ii) 允许对已经交换的信息进行相互使用，请与下列地址联系:

IBM Canada Limited
U59/3600
3600 Steeles Avenue East
Markham, Ontario L3R 9Z7
CANADA

只要遵守适当的条款和条件，包括某些情形下的一定数量的付费，都可获得这方面的信息。

本资料中描述的许可程序及其所有可用的许可资料均由 IBM 依据 IBM 客户协议、IBM 国际软件许可协议或任何同等协议中的条款提供。

此处包含的任何性能数据都是在受控环境中测得的。因此，在其他操作环境中获得的数据可能会有明显的不同。有些测量可能是在开发级的系统上进行的，因此不保证与一般可用系统上进行的测量结果相同。此外，有些测量是通过推算而估计的，实际结果可能会有差异。本文档的用户应当验证其特定环境的适用数据。

涉及非 IBM 产品的信息可从这些产品的供应商、其出版说明或其他可公开获得的资料中获取。IBM 没有对这些产品进行测试，也无法确认其性能的精确性、兼容性或任何其他关于非 IBM 产品的声明。有关非 IBM 产品性能的问题应当向这些产品的供应商提出。

所有关于 IBM 未来方向或意向的声明都可随时更改或收回，而不另行通知，它们仅仅表示了目标和意愿而已。

本信息可能包含在日常业务操作中使用的数据和报告的示例。为了尽可能完整地说明这些示例，示例中可能会包括个人、公司、品牌和产品的名称。所有这些名称都是虚构的，与实际商业企业所用的名称和地址的任何雷同纯属巧合。

版权许可:

本信息包括源语言形式的样本应用程序，这些样本说明不同操作平台上的编程方法。如果是为按照在编写样本程序的操作平台上的应用程序编程接口 (API) 进行应用程序的开发、使用、经销或分发，您可以任何形式对这些样本程序进行复制、修改、分发，而无须向 IBM 付费。这些示例并未在所有条件下作全面测试。因此，IBM 不能担保或暗示这些程序的可靠性、可维护性或功能。此样本程序“按现状”提供，且不附有任何种类的保证。对于使用此样本程序所引起的任何损坏，IBM 将不承担责任。

凡这些样本程序的每份拷贝或其任何部分或任何衍生产品，都必须包括如下版权声明:

©（*贵公司的名称*）（*年份*）. 此部分代码是根据 IBM 公司的样本程序衍生出来的。© Copyright IBM Corp. *（输入年份）*. All rights reserved.

商标

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other prod-

uct and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at 『Copyright and trademark information』 at www.ibm.com/legal/copytrade.shtml.

The following terms are trademarks or registered trademarks of other companies

- Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
- Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle, its affiliates, or both.
- UNIX is a registered trademark of The Open Group in the United States and other countries.
- Intel, Intel logo, Intel Inside, Intel Inside logo, Celeron, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
- Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

索引

[B]

[C]

[J]

[K]

[L]

[M]

[N]

[P]

[Q]

[S]

[T]

[W]

[X]

[Y]

[Z]

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Printed in China

S151-1762-00

Spine information:

IBM DB2 10.1 for Linux, UNIX, and Windows

SQL 过程语言：应用程序启用和支持