

DB2存储过程开发最佳实践

级别: 中级

常伟 (changwei@cn.ibm.com), 软件工程师, IBM CSDL

常红平 (changhp@cn.ibm.com), 软件工程师, IBM CSDL

2006 年 4 月 13 日

本文以 DB2 开发人员的角度介绍了在 DB2 存储过程开发中需要注意的事项和技巧。新手如果能够按照本文介绍的最佳实践来开发存储过程, 可以避免一些常见的错误, 从而编写出高效的程序。本文从初始化参数、游标、异常处理、临时表的使用以及如何寻找并 **rebind** 非法存储过程等常见问题进行了着重讨论, 并且给出了示例代码。

DB2 提供的强大功能可以让开发人员创建出非常高效稳定的存储过程。但对于初学者来说, 开发出这样的程序并不容易。本文主要讨论开发高效稳定的 DB2 存储过程的一些常用技巧和方法。

读者定位为具有一定开发经验的 DB2 开发经验的开发人员。

读者可以从本文学习到如何编写稳定、高效的存储过程。并可以直接使用文章中提供的 DB2 代码, 从而节省他们的开发和调试时间, 提高效率。

本文以 DB2 开发人员的角度介绍了在 DB2 存储过程开发中需要注意的事项和技巧。新手如果能够按照本文介绍的最佳实践来开发存储过程, 可以避免一些常见的错误, 从而编写出高效的程序。本文从初始化参数、游标、异常处理、临时表的使用以及如何寻找并 **rebind** 非法存储过程等常见问题进行了着重讨论, 并且给出了示例代码。

在存储过程中, 开发人员能够声明和设置 **SQL** 变量、实现流程控制、处理异常、能够对数据进行插入、更新或者删除。同时, 客户应用 (这里指调用存储过程的应用程序, 它可以是 **JDBC** 的调用, 也可以是 **ODBC** 和 **CLI** 等) 和存储过程之间可以传递参数, 并且从存储过程中返回结果集。其中, 使用 **SQL** 编写的 DB2 存储过程是在开发中常见的一种存储过程。本文主要讨论此类存储过程。

最佳实践 1: 在创建存储过程语句中提供必要的参数

创建存储过程语句 (**CREATE PROCEDURE**) 可以包含很多参数, 虽然从语法角度讲它们不是必须的, 但是在创建存储过程时提供它们可以提高执行效率。下面是一些常用的参数

容许 SQL (**allowed-SQL**)

容许 SQL (**allowed-SQL**) 子句的值指定了存储过程是否会使用 **SQL** 语句, 如果使用, 其类型如何。它的可能值如下所示:

- ❑ **NO SQL**: 表示存储过程不能够执行任何 **SQL** 语句。
- ❑ **CONTAINS SQL**: 表示存储过程可以执行 **SQL** 语句, 但不会读取 **SQL** 数据, 也不会修改 **SQL** 数据。
- ❑ **READS SQL DATA**: 表示在存储过程中包含不会修改 **SQL** 数据的 **SQL** 语句。也就是说该储存过程只从数据库中读取数据。
- ❑ **MODIFIES SQL DATA**: 表示存储过程可以执行任何 **SQL** 语句。即可以对数据库中的数据进行增加、删除和修改。

如果没有明确声明 **allowed-SQL**, 其默认值是 **MODIFIES SQL DATA**。不同类型的存储过程执行的效率是不同的, 其中 **NO SQL** 效率最好, **MODIFIES SQL DATA** 最差。如果存储过程只是读取数据, 但是因为缺少声明 **allowed-SQL** 使其被

当作对数据进行修改的存储过程来执行，这显然会降低程序的执行效率。因此创建存储过程时，应当明确声明其 **allowed-SQL**。

返回结果集个数 (DYNAMIC RESULT SETS n)

存储过程能够返回 0 个或者多个结果集。为了从存储过程中返回结果集，需要执行如下步骤：

- l 在 **CREATE PROCEDURE** 语句的 **DYNAMIC RESULT SETS** 子句中声明存储过程将要返回的结果集的数量 (**number-of-result-sets**)。如果这里声明的返回结果集的数量小于存储过程中实际返回的结果集数量，在执行该存储过程的时候，**DB2** 会返回一个警告。
- l 使用 **WITH RETURN** 子句，在存储过程体中声明游标。
- l 为结果集打开游标。当存储过程返回的时候，保持游标打开。

在创建存储过程时指定返回结果集的个数可以帮助程序员验证存储过程是否返回了所期待数量的结果集，提高了程序的完整性。



最佳实践 2: 对输入参数进行必要的的检查和预处理

无论使用哪种编程语言，对输入参数的判断都是必须的。正确的参数验证是保证程序良好运行的前提。同样的，在 **DB2** 中对输入参数的验证和处理也是很重要的。正确的验证和预处理操作包括：

- l 如果输入参数错误，存储过程应返回一个明确的值告诉客户应用，然后客户应用可以根据返回的值进行处理，或者向存储过程提交新的参数，或者去调用其他的程序。
- l 根据业务逻辑，对输入参数作一定的预处理，如大小写的转换，**NULL** 与空字符串或 0 的转换等。

在 **DB2** 储存过程开发中，如需要遇到对空(**NULL**)进行初始化，我们可以使用 **COALESCE** 函数。**COALESCE**函数返回第一个非空的参数，语法如下：

清单1: COALESCE 函数

```

(1)          .----- .
              v          |
>>-COALESCE-----(--expression----,--expression+--)->><
    
```

COALESCE函数会依次检查输入的参数，返回第一个不是**NULL**的参数，只有当传入**COALESCE**函数的所有的参数都是**NULL**的时候，函数才会返回**NULL**。例如， **COALESCE(piName,")**，如果变量piName为**NULL**，那么函数会返回"，否则就会返回piName本身的值。

下面的例子展示了如何对参数进行检查何初始化。

Person表用来存储个人的基本信息，其定义如下：

表1: Person

列名	参数	定义	说明
Num	piNum	INTEGER NOT NULL	人员编号, 非空, 不允许为''
Name	piName	VARCHAR(20) NOT NULL	人员名字, 非空
Age	piAge	INTEGER NOT NULL	年龄, 非空
Rank	piRank	INTEGER	等级, 缺省为0
Comment	piComment	VARCHAR(100)	备注

下面是用于向表Person插入数据的存储过程的参数预处理部分代码:

```

SET poGenStatus = 0;

SET piName      = RTRIM(COALESCE(piName, ''));
SET piRank      = COALESCE(piRank, 0);

-- make sure all required input parameters are not null
IF ( piNum IS NULL
    OR piName = ''
    OR piAge IS NULL )
THEN
    SET poGenStatus = 34100;
    RETURN poGenStatus;
END IF;

```

表Person中num、name和age都是非空字段。对于name字段, 多个空格我们也认为是空值, 所以在进行判断前我们调用RTRIM和COALESCE对其进行处理, 然后使用 piName = "", 对其进行非空判断; 对于Rank字段, 我们希望如果用户输入的NULL, 我们把它设置成"0", 对其我们也使用COALESCE进行初始化; 对于"Age"和"Num" 我们直接使用 IS NULL进行非空判断就可以了。

如果输入参数没有通过非空判断, 我们就对输出参数poGenStatus设置一个确定的值(例子中为 34100)告知调用者: 输入参数错误。

下面是对参数初始化规则的一个总结, 供大家参考:

1. 输入参数为字符类型, 且允许为空的, 可以使用COALESCE(inputParameter,"")把NULL转换成";
2. 输入类型为整型, 且允许为空的, 可以使用COALESCE(inputParameter,0), 把空转换成0;
3. 输入参数为字符类型, 且是非空非空格的, 可以使用COALESCE(inputParameter,"")把NULL转换成", 然后判断函数返回值是否为";
4. 输入类型为整型, 且是非空的, 不需要使用COALESCE函数, 直接使用IS NULL进行非空判断。

□

[回页首](#)

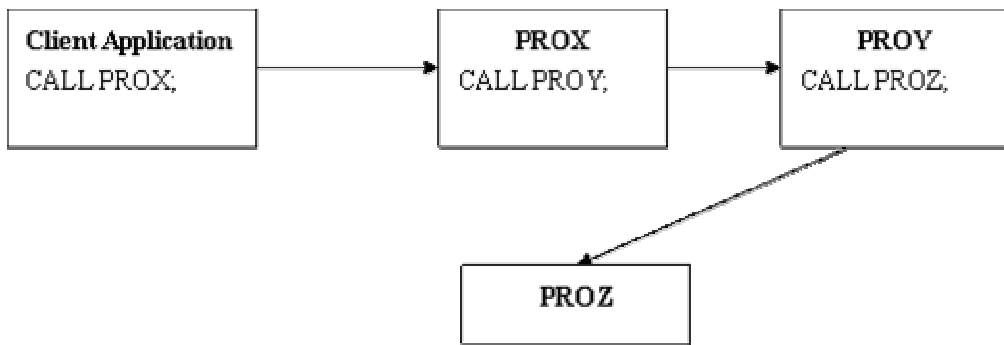
前面我们已经讨论了如何声明存储过程的返回结果集。这里我们讨论一下结果集返回类型的问题。结果集的返回类型有两种：调用者(CALLER) 和客户应用(CLIENT)。首先我们看一下声明这两种游标的例子：

```
CREATE PROCEDURE getPeople(IN piAge INTEGER)
DYNAMIC RESULT SETS 2
READS SQL DATA
LANGUAGE SQL
BEGIN
    DECLARE rs1 CURSOR WITH RETURN TO CLIENT FOR
        SELECT name, age FROM person
            WHERE age<piAge;
    DECLARE rs2 CURSOR WITH RETURN TO CALLER FOR
        SELECT NAME, age FROM person
            WHERE age>piAge;
    OPEN rs1;
    OPEN rs2;
END
```

代码中rs1游标的DECLAER语句中包含WITH RETURN TO CLIENT子句，表示结果集返回给客户应用(CLIENT)。rs2游标的DECLARE语句中包含WITH RETURN TO CALLER子句，表示结果集返回给调用者(CALLER)。

游标返回给调用者(CALLER)表示由存储过程的调用者接收结果集，而不考虑调用者是否是另一个存储过程，还是客户应用。图（1）中存储过程PROZ如果声明为WITH RETURN TO CALLER，那么结果集会返回给存储过程PROY，Client Application是不会得到PROZ返回的结果集的。

图1：存储过程递归调用



游标返回给客户应用（CLIENT）表示由发出最初 CALL 语句的客户应用接收结果集，即使结果集由嵌套层次中的 15 层深的嵌套存储过程发出也是如此。图1中存储过程 PROZ 如果声明为 WITH RETURN TO CLIENT，那么结果集会返回给 Client Application。返回给客户应用（CLIENT）的游标声明是我们经常使用的，也是默认的结果集类型。

在声明返回类型时，我们要认真考虑一下，我们需要把结果集返回给谁，以免丢失返回集，导致程序错误。

在存储过程执行的过程中，经常因为数据或者其他问题产生异常（**condition**）。根据业务逻辑，存储过程应该对异常进行相应处理或直接返回给调用者。此处暂且将**condition**译为异常以方便读者理解。实际上有些异常（**condition**）并非是由于错误引起的，下面将详细讲述。

当存储过程中的语句返回的**SQLSTATE**值超过00000的时候，就表明在存储过程中产生了一个异常（**condition**），它表示出现了错误、数据没有找到或者出现了警告。为了响应和处理存储过程中出现的异常，我们必须在存储过程体中声明异常处理器（**condition handler**），它可以决定存储过程怎样响应一个或者多个已定义的异常或者预定义异常组。声明条件处理器的语法如下，它会位于变量声明和游标声明之后：

清单4：声明异常处理器

```
DECLARE handler-type HANDLER FOR condition handler-action
```

异常处理器类型(handler-type)有以下几种：

- ┆ **CONTINUE** 在处理器操作完成之后，会继续执行产生这个异常语句之后的下一条语句。
- ┆ **EXIT** 在处理器操作完成之后，存储过程会终止，并将控制返回给调用者。
- ┆ **UNDO** 在处理器操作执行之前，DB2会回滚存储过程中执行的SQL操作。在处理器操作完成之后，存储过程会终止，并将控制返回给调用者。

异常处理器可以处理基于特定**SQLSTATE**值的定制异常，或者处理预定义异常的类。预定义的3种异常如下所示：

- ┆ **NOT FOUND** 标识导致**SQLCODE**值为+100或者**SQLSTATE**值为02000的异常。这个异常通常在**SELECT**没有返回行的时候出现。
- ┆ **SQLWARNING** 标识导致警告异常或者导致+100以外的**SQLCODE**正值的异常。
- ┆ **SQLWARNING** 标识导致警告异常或者导致+100以外的**SQLCODE**正值的异常。

如果产生了**NOT FOUND** 或者**SQLWARNING**异常，并且没有为这个异常定义异常处理器，那么就会忽略这个异常，并且将控制流转向下一个语句。如果产生了**SQLWARNING**异常，并且没有为这个异常定义异常处理器，那么存储过程就会失败，并且会将控制流返回调用者。

以下示例声明了两个异常处理器。 **EXIT**处理器会在出现**SQLWARNING** 或者**SQLWARNING**异常的时候被调用。**EXIT**处理器会在终止SQL程序之前，将名为stmt的变量设为"ABORTED",并且将控制流返回给调用者。**UNDO**处理器会将控制流返回给调用者之前，回滚存储过程体中已经完成的SQL操作。

清单5：异常处理器示例

```
DECLARE EXIT HANDLER FOR SQLWARNING, SQLWARNING
    SET stmt = 'ABORTED';
```

```
DECLARE UNDO HANDLER FOR NOT FOUND;
```

如果预定义异常集不能满足需求，就可以为特定的**SQLSTATE**值声明定制异常，然后再为这个定制异常声明处理器。语法如下：

清单6：定制异常处理器

```
DECLARE unique-name CONDITION FOR SQLSTATE 'sqlstate'
```

处理器可以由单独的存储过程语句定义, 也可以使用由BEGIN...END块界定的复合语句定义。注意在执行符合语句的时候, SQLSTATE和SQLCODE的值会被改变, 如果需要保留异常前的SQLSTATE和SQLCODE, 就需要在执行复合语句的第一个语句把SQLSTATE和SQLCODE赋予本地变量或参数。

通常, 我们会为存储过程定义一个执行状态的输出参数 (例如: poGenStatus)。

根据这个输出状态, 可以表明存储过程是否正确执行完毕。我们需要定义一些异常处理器为这个输出参数赋值。下面是一个例子:

清单7: 定义为输出参数赋值的异常处理器

```
-- Generic Handler
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION, SQLWARNING, NOT FOUND
BEGIN NOT ATOMIC
    -- Capture SQLCODE & SQLSTATE
    SELECT  SQLCODE, SQLSTATE
    INTO    hSqlcode, hSqlstate
    FROM    SYSIBM.SYSDUMMY1;

    -- Use the poGenStatus variable to tell the procedure -- what type of
    error occurred
    CASE hSqlstate
        WHEN '02000' THEN
            SET poGenStatus=5000;
        WHEN '42724' THEN
            SET poGenStatus=3;
        ELSE
            IF (hSqlCode < 0) THEN
                SET poGenStatus=hSqlCode;
            END IF;
        END CASE;
    END;
```

上面的异常处理器会在出现SQLEXCEPTION, SQLWARNING, NOT FOUND异常的时候触发。异常处理器会取出当前的SQLCODE, SQLSTATE, 然后根据它们的值来设置输出参数 (poGenStatus) 的值。

我们还可以定制一些异常处理器。例如, 我们可以定义一些对参数进行初始化的异常处理器。这里, 异常处理器可以看作是一个供存储过程自己调用的内部函数。下面是这种情况的一个例子:

清单8: 供存储过程自己调用的内部函数

```
-----
-- CONDITION declaration
-----
-- (80100~80199) SQLCODE & SQLSTATE
DECLARE sqlReset CONDITION for sqlstate '80100';
```

```

-----
-- EXCEPTION HANDLER declaration
-----

-- Handy Handler
DECLARE CONTINUE HANDLER FOR sqlReset
BEGIN NOT ATOMIC
    SET hSqlcode    = 0;
    SET hSqlstate   = '00000';
    SET poGenStatus = 0;
END;

.....

-----
-- Procedure Body
-----

SIGNAL sqlreset;

-- insert the record
.....

```

上面定制的异常处理器负责对参数hSqlcode, hSqlstate和poGenStatus初始化。当我们在程序中需要对它们初始化时, 我们只需要调用SIGNAL sqlreset就可以了。

[回页首](#)

最佳实践 5: 合理使用临时表

我们在储存过程开发中经常使用临时表。合理的使用临时表可以简化程序的编写, 提供执行效率, 然而滥用临时表同样也会使得程序运行效率降低。

临时表一般在如下情况下使用:

1. 临时表用于存储程序运行中的临时数据。例如, 如果在一个程序中第一条查询语句执行的结果会被后续的查询语句用到, 那么我们可以把第一次查询的结果存储在一个临时表中供后续查询语句使用, 而不是在后续查询语句中重新查询一次。如果第一条查询语句非常复杂和耗时, 那么上面的策略是非常有效的。
2. 临时表可以用于存储在一个程序中需要返回多次的结果集。例如, 程序中有一个很耗资源的多表查询, 同时, 该查询在程序中需要执行多次, 那么就可以把第一次查询的结果集存储在临时表中, 后续的查询只需要查临时表就可以了。
3. 临时表也可以用于让SQL访问非关系型数据库。例如, 可以编写程序把非关系型数据库中的数据插入到一个全局临时表中, 那么我们就可以对其数据进行查询。

我们可使用 **DECLARE GLOBAL TEMPORARY TABLE** 语句来定义临时表。DB2的临时表是基于会话的, 且在会话之间是隔离的。当会话结束时, 临时表的数据被删除, 临时表被隐式卸下。对临时表的定义不会在SYSCAT.TABLES中出现下面是定义临时表的一个示例:

清单9: 定义临时表


```

DECLARE GLOBAL TEMPORARY TABLE gbl_temp
LIKE person
ON COMMIT DELETE ROWS
NOT LOGGED
IN usr_tbsp

```

此语句创建一个名为 **gbl_temp** 的用户临时表。定义此用户临时表 所使用的列的名称和说明与 **person** 的列的名称和说明完全相同。

清单10: 创建有两个字段的临时表

```

DECLARE GLOBAL TEMPORARY TABLE SESSION.TEMP2
(
    ID      INTEGER default 3,
    NAME    CHAR(30)
)
--WITH REPLACE
NOT LOGGED;
--IN USER_TEMP_01;

```

此语句创建了一个有两个字段的临时表。

理论上临时表是不需要显示**DROP**的, 因为它是基于会话的, 当临时表基于的连接关闭的时候, 临时表也就不存在了。但是在实际开发中会有一些情况需要我们对临时表加以注意。

一种情况就是被调用的存储过程的返回值是一个基于临时表的结果集。当存储过程执行完毕的时候, 临时表并不会消失, 因为返回的结果集相当于一个指针, 指向临时表所在的内存地址, 此时临时表是不会被**DROP**掉的。这种情况下, 既不能在存储过程中删除这个临时表, 也不应该由客户应用显示的删除临时表, 这就容易出现一些问题。下面我们通过一个例子来说明这个问题。

下面示例代码是返回临时表的存储过程 (**get_temp_table**) :

清单11: 返回临时表的存储过程

```

-----
-- TEMPORARY TABLE & CURSOR declaration
-----
DECLARE GLOBAL TEMPORARY TABLE SESSION.TEMP
(
    ID      INTEGER,
    NAME    CHAR(30)
)
--WITH REPLACE
NOT LOGGED;

P2: BEGIN

```



```
DECLARE R_CRSR CURSOR WITH RETURN TO CLIENT FOR
  SELECT * FROM SESSION.TEMP
FOR READ ONLY;

INSERT INTO SESSION.TEMP VALUES(1,piName);

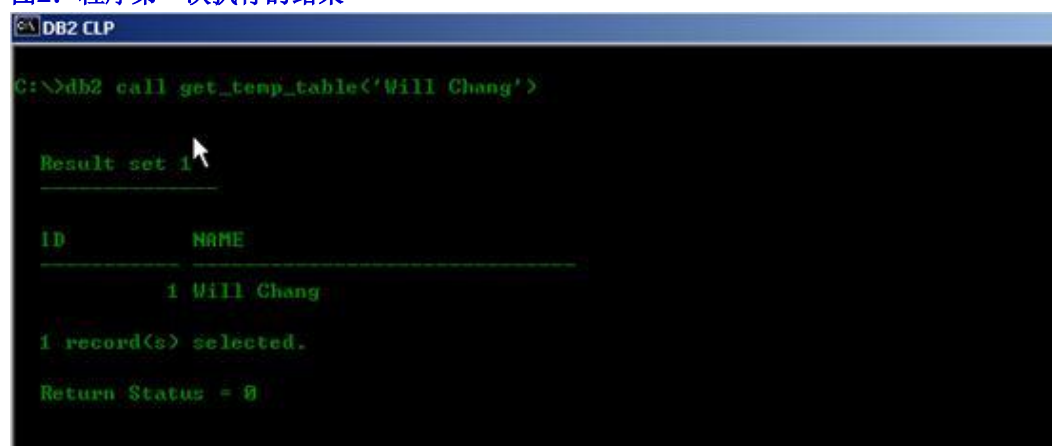
OPEN R_CRSR;

END P2;
```

存储过程中声明了有两个字段的临时表TEMP,声明了一个游标R_CRSR返回临时表中所有记录,最后在临时表中插入两条记录。

程序第一次执行的结果如下:

图2: 程序第一次执行的结果



```
C:\>db2 call get_temp_table('Will Chang')

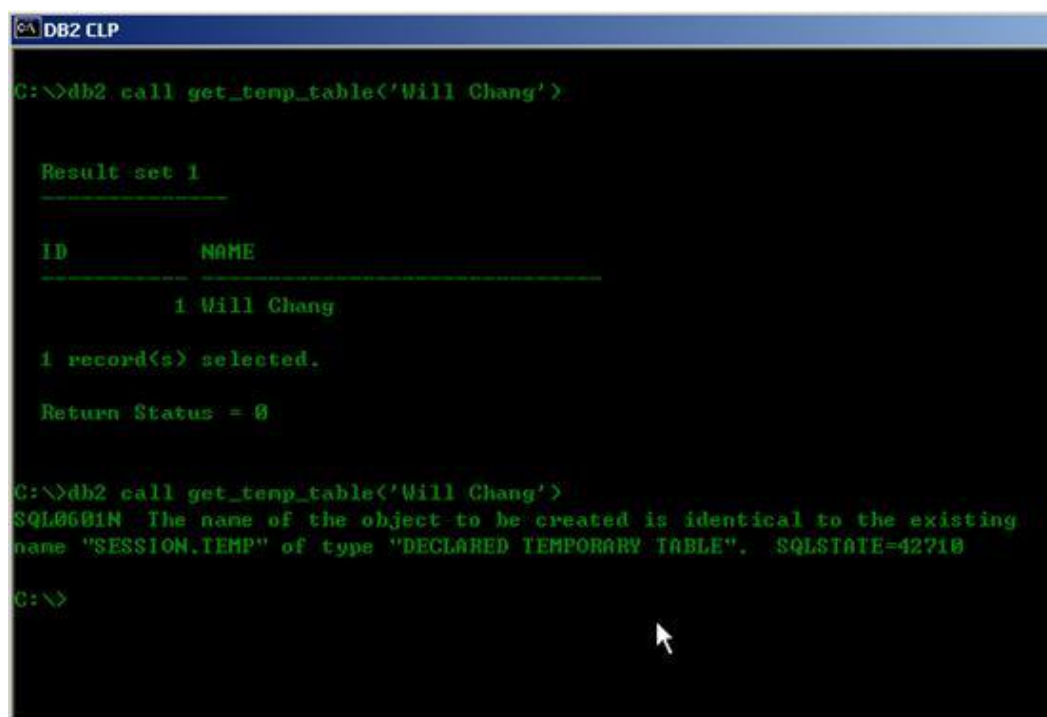
Result set 1
-----
ID      NAME
-----
1 Will Chang

1 record(s) selected.

Return Status = 0
```

可以从图中看出,运行结果是我们期望的。那么如果我们再运行一次,会有什么结果呢?下图是其运行结果:

图3: 程序再次执行的结果



```
DB2 CLP

C:\>db2 call get_temp_table<'Will Chang'>

Result set 1
-----
ID      NAME
-----
1 Will Chang

1 record(s) selected.

Return Status = 0

C:\>db2 call get_temp_table<'Will Chang'>
SQL0601N  The name of the object to be created is identical to the existing
name "SESSION.TEMP" of type "DECLARED TEMPORARY TABLE".  SQLSTATE=42710

C:\>
```

第二次执行的时候程序却出错了，这是因为在同一个连接中，临时表并没有被DROP掉，所以在第二次调用存储过程的时候就会出现临时表已经存在的错误。

另外一种情况，就是很多时候例如在websphere中通过JDBC连接数据库时使用了连接池的技术，这带来了一些效率的提升，同时在某些情况下也容易让人误解。客户应用程序中关闭了数据库连接，但是并不一定真正关闭了数据库连接，如果客户应用程序使用了临时表而数据库连接并没有关闭，那么临时表就不会被DROP。当连接池把这个连接分给另一个客户程序的时候，新的客户程序仍然可以使用旧的临时表，这不是我们希望的。如果想避免上述问题，可以在创建临时表时，加上WITH REPLACE；或者根据业务逻辑在合适的地方显示的DROP临时表。

下面是使用WITH REPLACE创建临时表的执行情况。

图4：使用WITH REPLACE创建临时表

```

C:\>db2 call get_temp_table<'Will Chang'>

Result set 1
-----
ID      NAME
-----
1 Will Chang
1 record(s) selected.
Return Status = 0

C:\>db2 call get_temp_table<'Will'>

Result set 1
-----
ID      NAME
-----
1 Will
1 record(s) selected.
Return Status = 0

C:\>

```

可以看出在一个连接里面，多次调用存储过程`get_temp_table`，也不会出现问题。临时表在某些情况下也是需要避免使用的。大家知道临时表是存放在内存中的，如果一个临时表有上万或者十几万条记录，同时程序的并发数很大，那么在内存中建立的临时表耗费资源就很庞大，此时数据库的性能会急剧下降，甚至会导致数据库崩溃。因此，大家在使用临时表的时候，需要考虑它对资源的耗费，避免盲目使用临时表。

□

[回页首](#)

最佳实践 6: 寻找并rebind 非法的存储过程

存储过程会因为其涉及和引用的对象发生了改变而导致其非法（invalid），例如：修改了表结构，导致引用该表的存储过程非法，或者重新编译一个存储过程，会使调用这个存储过程的父存储过程非法。此时我们需要对非法的存储过程重新编译(rebind)。但是，对非法的存储过程进行rebind的时候，需要确定其引用的对象是合法的，否则非法的存储过程也不能rebind成功。

这里我们介绍一下发现和rebind非法存储过程的方法。我们是通过判断SYSCAT.routines中VALID字段的值来查找非法存储过程的。下面是查找非法存储过程的一段代码：

清单12: 查找非法存储过程

```

SELECT
    RTRIM(r.routineschema) || '.' || RTRIM(r.routinename) AS spname ,
    ' ( ' || RTRIM(r.routineschema) || '.' || 'P' || SUBSTR(CHAR(r.lib_id+10000000),2) || ' ) '
FROM
    SYSCAT.routines r

```

```

WHERE
    r.routinetype = 'P'
AND ((r.origin = 'Q' AND r.valid != 'Y')
    OR EXISTS (
        SELECT 1 FROM syscat.packages
        WHERE pkgschema = r.routineschema
        AND pkgname = 'P' || SUBSTR(CHAR(r.lib_id+10000000),2)
        AND valid != 'Y'
    )
)
ORDER BY
    spname;

```

获得的结果如下:

清单13: 查找非法存储过程的结果

```

SPNAME
-----
TEST.DEMO_INFO_8          (TEST. P3550884)

```

可以使用下面的命令rebind它们

清单14: Rebind 非法存储过程语法

```
rebind package packagename resolve any@
```

Packagename就是查询结果中括号里的值。例如, 如果rebind上面查出来的存储过程。我们只需要执行下面语句

清单15: Rebind 非法存储过程

```
rebind package TEST.P3550884 resolve any@
```

当然, 如果此存储过程程序本身有问题, 需要先修改存储过程代码后再进行编译。

类似的, 通过下面的代码可以获得非法的视图。

清单16: 获得非法的视图

```

SELECT
    RTRIM(viewschema) || '.' || RTRIM(viewname) AS viewname
FROM
    SYSCAT.views
WHERE
    valid = 'X'
ORDER BY

```

```
viewname;
```

 [回页首](#)

结束语

本文介绍了我们在 DB2 存储过程开发中经常用到的一些技巧。同时这些技巧也是编写优秀存储过程的基本要求。本文介绍的一些技巧只是揭开了高效使用 DB2 的冰山一角。DB2 为我们提供了丰富和强大的功能。在使用 DB2 的时候, 我们应当深入理解其原理, 找出更多的最佳实践与大家分享。

参考资料

- 1 Red book: IBM DB2 UDB Command Reference Version 8
- 1 Red Book: IBM DB2 UDB Application Development Guide