Oracle_PLSQL培训_V1.0

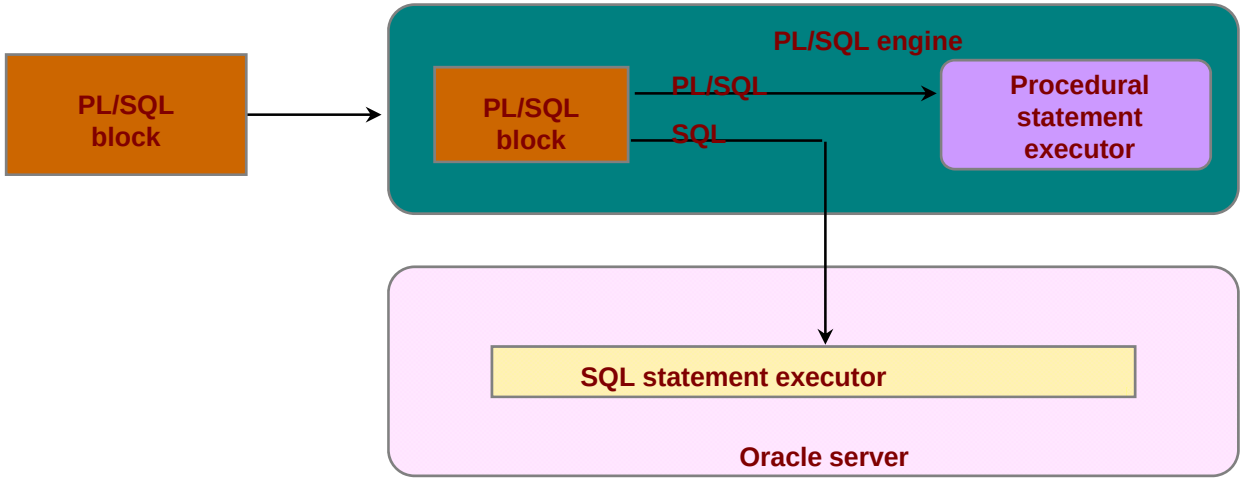
来源：Oracle 标准培训教材
编写人：　商云方
编写日期：20110630
版本：　1.0

汉得信息技术有限公司
HAND Enterprise Solutions Company Ltd.
www.hand-china.com

PLSQL 概览

PLSQL 是Oracle公司在SQL基础上进行扩展而成的一种过程语言。PLSQL提供了典型的高级语言特性，包括封装，例外处理机制，信息隐藏，面向对象等；并把最新的编程思想带到了数据库服务器和工具集中。

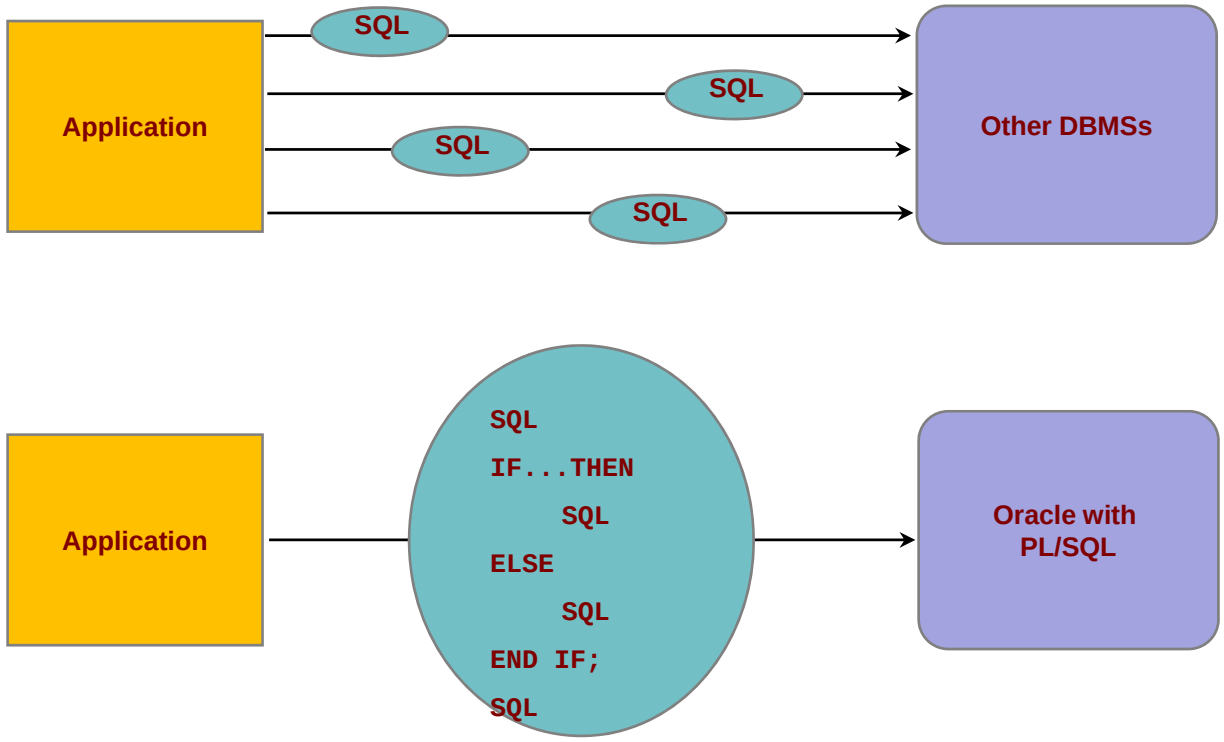
与Java, C#相比，PLSQL的优势是：SQL语言可以直接写到PLSQL的“块”中或者是PLSQL的过程、函数中。没有必要向java那样先创建Statement对象来执行SQL；这使得PLSQL成为很强大的事务处理语言，即：使用SQL来处理数据，使用控制结构来处理业务逻辑。



PLSQL在Oracle 数据库服务器（在存储过程、函数、数据库触发器，Package包中使用）和Oracle开发工具集（在 开发工具组件的触发器中使用）；Form Developer，Report Developer 还可以使用共享库（包含使用PLSQL写的过程和函数，扩展名为PLL的文件）；SQL数据类型也可以在PLSQL中使用，结合SQL提供者的直接访问，这些共享数据类型整合了PLSQL和Oracle的数据库字典。PLSQL消除了存取数据库的便利性与过程语言之间的障碍。

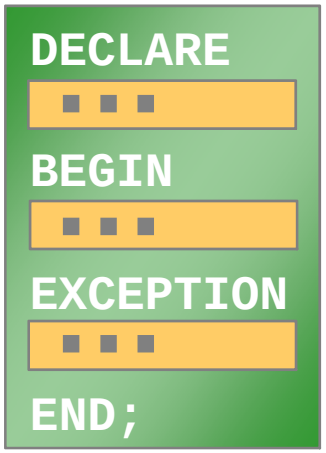
PLSQL 概览

PLSQL的另一个显著好处在于它可以通过减少来回交互减轻网络流量压力、节省时间：



PLSQL的块概念

PLSQL是一种类PASCAL语言，每一段程序都是由Block 组成的：



DECLARE (Optional)

Variables, cursors, user-defined exceptions

BEGIN (Mandatory)

- SQL statements
- PL/SQL statements

EXCEPTION (Optional)

Actions to perform when errors occur

END; (Mandatory)

使用分号作为一句SQL 或者PLSQL语句的结束；块结构关键字（DECLARE, BEGIN,EXCEPTION 后面不跟分号；END后面需带分号；你可以把一句SQL语句写在一行上，但一般不建议这么做，因为代码不够漂亮；

```
DECLARE
  v_variable  VARCHAR2(5);
BEGIN
  SELECT column_name
  INTO    v_variable
  FROM    table_name;
EXCEPTION
  WHEN exception_name THEN
  ...
END;
```

PLSQL的块概念

PLSQL的块包括三种：匿名块、存储过程、函数；

Anonymous

```
[DECLARE]

BEGIN
  --statements

[EXCEPTION]

END;
```

Procedure

```
PROCEDURE name
IS

BEGIN
  --statements

[EXCEPTION]

END;
```

Function

```
FUNCTION name
RETURN datatype
IS
BEGIN
  --statements
  RETURN value;
[EXCEPTION]

END;
```

PLSQL变量

PLSQL的变量类型：

1、系统内置的常规简单变量类型： 比如大多数 数据库表的字段类型都可以作为变量类型；
2、用户自定义复杂变量类型： 比如记录类型；
3、引用类型：保存了一个指针值；
4、大对象类型（ LOB）：保存了一个指向大对象的地址；

SQLPLUS变量

PLSQL本身没有输入输出功能，如果要想 像命令行运行C程序那样可以接收输入值，那你必须依赖执行环境把值传给PLSQL块，比如 iSQL Plus执行环境或者PLSQL Developer的Command Window 执行环境中，有一种 substitution变量 可以用来接收输入值；而另一种Host变量可以把运行时的值传出到执行环境中。有关SQLPLUS变量，我们会在以后详细展开；

PLSQL变量

PLSQL的变量类型举例：

布尔类型 — TRUE

25-JAN-01 — 日期类型

BLOB类型

“Four score and seven years ago our fathers brought forth upon this continent, a new nation, conceived in LIBERTY, and dedicated to the proposition that all men are created equal.”

LONG类型

数值类型 — 256120.08

Atlanta — 字符串类型

BFILE类型

PLSQL变量

PLSQL的变量声明：

语法：

```
identifier [CONSTANT] datatype [NOT NULL]
     [:= | DEFAULT expr];
```

举例：

```
DECLARE
  v_hiredate          DATE;
  v_deptno            NUMBER(2) NOT NULL := 10;
  v_location          VARCHAR2(13) := 'Atlanta';
  c_comm              CONSTANT NUMBER := 1400;
```

说明：

1、变量命名建议遵循通用规则，比如v_name 表示一个变量，c_name表示一个常量；
2、一般建议每一行声明一个变量，这样程序的可读性比较好；
3、如果声明了变量，但未进行初始化，则在没有赋值之前该变量的值为NULL; 一个好的编程习惯是对所有声明的变量进行初始化赋值。

PLSQL变量

说明：

4、在同一个块中，避免命名与数据库表中的字段名相同的变量；

```
DECLARE
  employee_id  NUMBER(6);
BEGIN
  SELECT       employee_id
  INTO         employee_id
  FROM         employees
  WHERE        last_name =
'Kochhar';
END;
```

这是一个反面教材，合理的命名方法是给变量起名为： **v_employee_id**

PLSQL变量

常规类型的变量声明举例：

```
DECLARE
  v_job                 VARCHAR2(9);
  v_count               BINARY_INTEGER := 0;
  v_total_sal           NUMBER(9,2) := 0;
  v_orderdate           DATE := SYSDATE + 7;
  c_tax_rate      CONSTANT NUMBER(3,2) := 8.25;
  v_valid               BOOLEAN NOT NULL := TRUE;
  ...
```

PLSQL特有的%TYPE属性来声明与XX类型一致的变量类型：

语法：

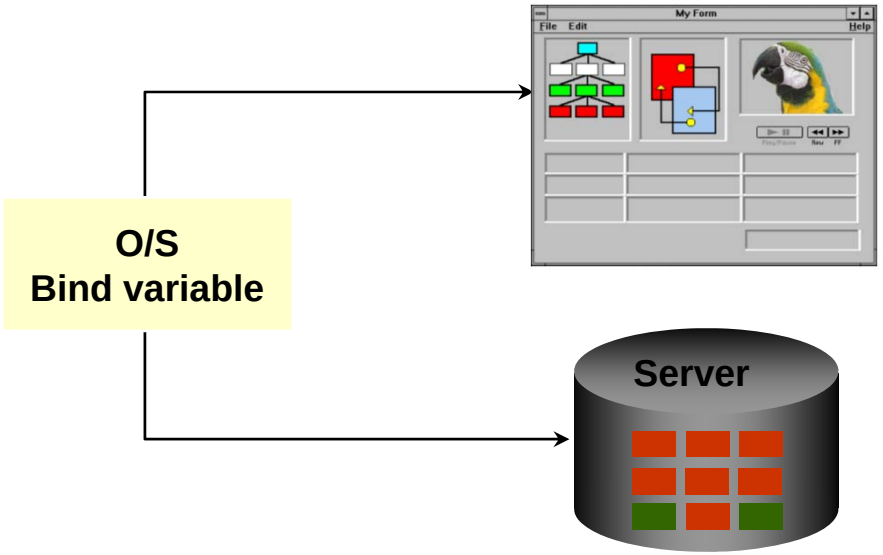
```
identifier              Table.column_name%TYPE;
```

举例：

```
...
  v_name              employees.last_name%TYPE;
  v_min_balance       v_balance%TYPE := 10;
...
```

PLSQL变量

可绑定变量（Bind Variable 也称为Host Variable , 非PLSQL 变量）：




```
VARIABLE      g_salary NUMBER
BEGIN
   SELECT     salary
   INTO       :g_salary
   FROM       employees
   WHERE      employee_id = 178;
END;
/
PRINT g_salary
```

可绑定变量是一种在缩主环境中定义的变量，所谓缩主环境一般指示SQLPLUS执行环境或者是PLSQL Developer 的Command Window执行环境；可绑定变量可用于在运行时把值传递给PLSQL, 创建语法：

```
VARIABLE return_code NUMBER
VARIABLE return_msg  VARCHAR2(30)
```

大家注意，在标准的PLSQL中定义变量是不能用VARIABLE关键字的，此关键字只在SQLPLUS执行环境中有效，可使用PRINT语句输出变量内容。

在PLSQL中使用这种变量时，前面加":", 以示区分。

PLSQL变量

DBMS_OUTPUT.PUT_LINE()介绍：

在接下来的实验中，经常需要在调试程序时输出中间变量的值，我们可使用**Oracle** 内置的**Package**中的函数：

```
DECLARE
  v_sal NUMBER(9,2) := &p_annual_sal;
BEGIN
  v_sal := v_sal/12;
  DBMS_OUTPUT.PUT_LINE ('The monthly salary is ' ||
                        TO_CHAR(v_sal));
END;
```

上述例子中，我们使用**DBMS_OUTPUT.PUT_LINE ()** 输出变量**v_sal**的值；

解释： **&p_annual_sal** 在**Plsql Developer**的**SQL window** 执行环境中，可用于提示用户输入一个具体的值。

注意：在**SQLPLUS**中执行 **DBMS_OUTPUT.PUT_LINE ()** 前，必须先执行：
SET SERVEROUTPUT ON ，而在**PLSQL Developer**的**SQL Window**中则不需要这句话。

PLSQL变量

PLSQL中的注释语句：

1、多行注释类似于java 或者 C， 使用 /* 和 */
2、单行注释是在语句后面使用 –

举例：

```
DECLARE
...
  v_sal NUMBER (9,2);
BEGIN
  /* Compute the annual salary based on the
     monthly salary input from the user */
  v_sal := :g_monthly_sal * 12;
END;              -- This is the end of the block
```

PLSQL变量

SQL函数在PLSQL的过程语句中的使用：

哪些可以用？ 哪些不可以用？

大多数SQL函数都可以在 PLSQL的过程语句中使用，比如：

单行的数值和字符串函数、数据类型转换函数、日期函数、时间函数、求最大、最小值的 GREATEST, LEAST 函数等；

但有些函数在PLSQL的过程语句中是不能使用的，比如：

Decode函数、分组函数（AVG, MIN, MAX, COUNT, SUM, STDDEV, and VARIANCE）等；

PLSQL变量

块嵌套和变量范围：

PLSQL的块是可以嵌套的，变量的作用范围与其他语言类似

```
...
  x  BINARY_INTEGER;
BEGIN
  ...
  DECLARE
    y  NUMBER;
  BEGIN
      y:= x;
  END;
  ...
END;
```

Scope of *x*

Scope of *y*

PLSQL变量

变量限定词： 假设我们在块嵌套的程序中，里层和外层有相同的变量声明，而里层的程序要访问外层的同名变量该怎么办呢？

答案是：使用块限定词，请看例子：

```
<<outer>>
   DECLARE
      birthdate DATE;
   BEGIN
         DECLARE
             birthdate DATE;
         BEGIN
            ...
            outer.birthdate :=
                           TO_DATE('03-AUG-1976',
                                   'DD-MON-YYYY');
         END;
    ....
    END;
```

这个例子中，**birthdate** 是同名变量，限定词 **outer** 表示外层，里层要访问外层的**bithdate**时使用 **outer.birthdate** 这种格式：

PLSQL变量

课堂练习：

```
<<outer>>
DECLARE
  v_sal      NUMBER(7,2) := 60000;
  v_comm     NUMBER(7,2) := v_sal * 0.20;
  v_message  VARCHAR2(255) := ' eligible for commission';
BEGIN
  DECLARE
           v_sal       NUMBER(7,2) := 50000;
           v_comm      NUMBER(7,2) := 0;
           v_total_comp            NUMBER(7,2) := v_sal + v_comm;
  BEGIN
           v_message := 'CLERK not'||v_message;
           outer.v_comm := v_sal * 0.30;
  END;
    v_message := 'SALESMAN'||v_message;
END;
```

1

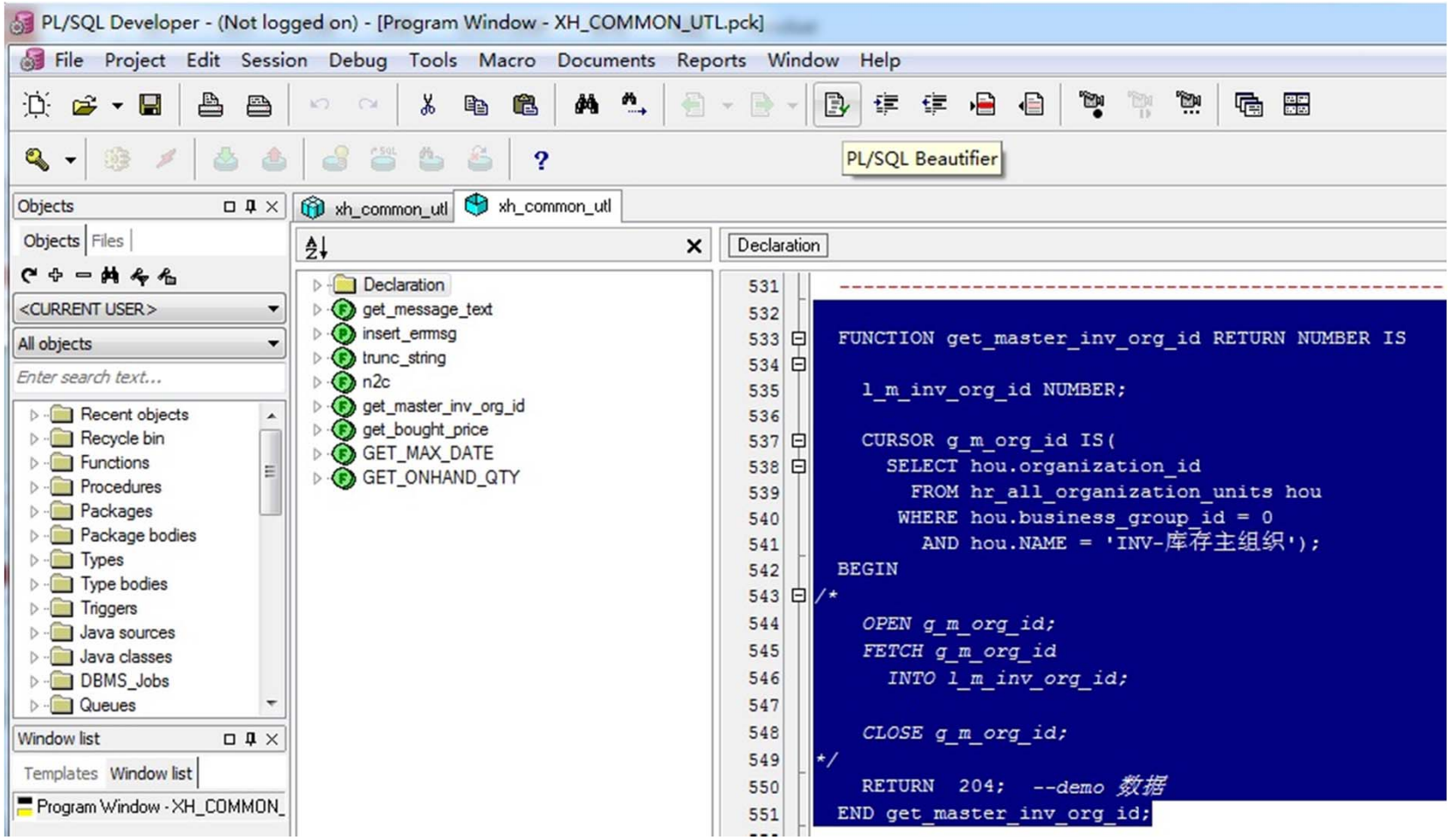
2

告诉我如下答案：

1. The value of V_MESSAGE at position 1.
2. The value of V_TOTAL_COMP at position 2.
3. The value of V_COMM at position 1.
4. The value of outer.V_COMM at position 1.
5. The value of V_COMM at position 2.
6. The value of V_MESSAGE at position 2.

PLSQL变量

PLSQL代码美化： 使用PLSQL Developer中的 Builtifier 功能：



操作：选中欲美化的代码，点上面的 **“PL/SQL Beautifier”**按钮即可，注意有时代码有语法错误的时候该功能无效，所以最好先编译通过再美化；

PLSQL中的SQL语句

SELECT INTO 语句：用于把从数据库查询出内容存入变量

```
DECLARE
  v_hire_date   employees.hire_date%TYPE;
  v_salary      employees.salary%TYPE;
BEGIN
  SELECT   hire_date, salary
  INTO     v_hire_date, v_salary
  FROM     employees
  WHERE    employee_id = 100;
  ...
END;
/
```

```
SET SERVEROUTPUT ON
DECLARE
  v_sum_sal   NUMBER(10,2);
  v_deptno        NUMBER NOT NULL := 60;
BEGIN
  SELECT SUM(salary)  -- group function
  INTO           v_sum_sal
  FROM           employees
  WHERE  department_id = v_deptno;
  DBMS_OUTPUT.PUT_LINE ('The sum salary is ' ||
                        TO_CHAR(v_sum_sal));
END;
/
```

注意点：该语句支持单行的查询结果，如果Where条件控制不好，导致多行查询结果，则会引发Too_many_rows的例外

PLSQL中的SQL语句

INSERT、UPDATE、DELETE 、MERGE语句： 在PLSQL中执行这些SQL语句和直接执行这些语句差不多，只不过可以在SQL语句中使用PLSQL声明的变量；

```
BEGIN
  INSERT INTO employees
  (employee_id, first_name, last_name, email,
   hire_date, job_id, salary)
  VALUES
   (employees_seq.NEXTVAL, 'Ruth', 'Cores', 'RCORES',
    sysdate, 'AD_ASST', 4000);
END;
/
```

```
DECLARE
  v_sal_increase   employees.salary%TYPE := 800;
BEGIN
  UPDATE employees
  SET            salary = salary + v_sal_increase
  WHERE  job_id = 'ST_CLERK';
END;
/
```

PLSQL中的SQL语句

```
DECLARE
  v_deptno   employees.department_id%TYPE := 10;
BEGIN
  DELETE FROM   employees
  WHERE         department_id = v_deptno;
END;
/
```

```
DECLARE
          v_empno employees.employee_id%TYPE := 100;
BEGIN
MERGE INTO copy_emp c
     USING employees e
     ON (e.employee_id = v_empno)
   WHEN MATCHED THEN
     UPDATE SET
       c.first_name     = e.first_name,
       c.last_name      = e.last_name,
       c.email          = e.email,
       . . .
   WHEN NOT MATCHED THEN
     INSERT VALUES(e.employee_id, e.first_name, e.last_name,
          . . .,e.department_id);
END;
```

PLSQL中的控制语句

和其他语言一样，控制主要包括判断和循环；

判断语句的语法与其他语言类似：

```
IF condition THEN
  statements;
[ELSIF condition THEN
  statements;]
[ELSE
  statements;]
END IF;
```

```
CASE selector
   WHEN expression1 THEN result1
   WHEN expression2 THEN result2
   ...
   WHEN expressionN THEN resultN
  [ELSE resultN+1;]
END;
```

需要注意的是对NULL的判断处理：

PLSQL中的控制语句

需要注意的是对NULL的判断处理：一般人容易犯错误或者不容易记住

AND	*TRUE*	*FALSE*	*NULL*
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	*TRUE*	*FALSE*	*NULL*
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

NOT	
TRUE	FALSE
FALSE	TRUE
NULL	NULL

PLSQL中的控制语句

循环语句的语法与其他语言类似：有基本循环、For循环、Wihle循环三种

```
LOOP
  statement1;
  . . .
  EXIT [WHEN condition];
END LOOP;
```

```
WHILE condition LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

Condition is evaluated at the beginning of each iteration.

```
FOR counter IN [REVERSE]
    lower_bound..upper_bound LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

PLSQL中的控制语句

举例：

```
DECLARE
  v_country_id    locations.country_id%TYPE := 'CA';
  v_location_id   locations.location_id%TYPE;
  v_city          locations.city%TYPE := 'Montreal';
BEGIN
  SELECT MAX(location_id) INTO v_location_id
    FROM locations
    WHERE country_id = v_country_id;
  FOR i IN 1..3 LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_location_id + i), v_city, v_country_id );
  END LOOP;
END;
/
```

请把它改写成标准循环、While循环

PLSQL中的控制语句

嵌套循环和Label

```
...
BEGIN
  <<Outer_loop>>
  LOOP
    v_counter := v_counter+1;
  EXIT WHEN v_counter>10;
    <<Inner_loop>>
    LOOP
      ...
      EXIT Outer_loop WHEN total_done = 'YES';
      -- Leave both loops
      EXIT WHEN inner_done = 'YES';
      -- Leave inner loop only
      ...
    END LOOP Inner_loop;
    ...
  END LOOP Outer_loop;
END;
```

Label一般用不着，只有在使用goto语句，或者内部循环需要访问外部的同名变量的时候才需要，而一般这种做法也是不被提倡的。

PLSQL中的复杂自定义数据类型

概述：PLSQL中常用的自定义类型就两种： 记录类型、PLSQL内存表类型（根据表中的数据字段的简单和复杂程度又可分别实现类似于简单数组和记录数组的功能）

记录类型的定义语法：

```
TYPE type_name IS RECORD
     (field_declaration[, field_declaration]…);
identifier       type_name;
```

这里的field_declaration 的具体格式可以是：

```
field_name {field_type | variable%TYPE
           | table.column%TYPE | table%ROWTYPE}
           [[NOT NULL] {:= | DEFAULT} expr]
```

举例：

```
...
  TYPE emp_record_type IS RECORD
    (last_name   VARCHAR2(25),
     job_id      VARCHAR2(10),
     salary      NUMBER(8,2));
  emp_record     emp_record_type;
...
```

PLSQL中的复杂自定义数据类型

%ROWTYPE属性：在PLSQL中 %ROWTYPE 表示某张表的记录类型 或者是用户指定以的记录类型，使用此属性可以很方便的定义一个变量，其类型与某张表的记录或者自定义的记录类型保持一致。极大的方便了Select * into ....的语句使用。

举例：

```
DECLARE
    emp_rec   employees%ROWTYPE;
BEGIN
     SELECT * INTO emp_rec
     FROM employees
     WHERE  employee_id = &employee_number;

     INSERT INTO retired_emps(empno, ename, job, mgr, hiredate,
                              leavedate, sal, comm, deptno)
     VALUES (emp_rec.employee_id, emp_rec.last_name, emp_rec.job_id,
     emp_rec.manager_id, emp_rec.hire_date, SYSDATE, emp_rec.salary,
     emp_rec.commission_pct, emp_rec.department_id);
     COMMIT;
END;
    /
```

PLSQL中的复杂自定义数据类型

PLSQL内存表即 Index By Table，这种结构类似于数组，使用主键提供类似于数组那样的元素访问。 这种类型必须包括两部分：1、使用 BINARY_INTEGER 类型构成的索引主键； 2、另外一个简单类型或者用户自定义类型的字段作为具体的数组元素。 这种类型可以自动增长，所以也类似于可变长数组。

语法：

```
TYPE type_name IS TABLE OF
     {column_type | variable%TYPE
     | table.column%TYPE} [NOT NULL]
     | table.%ROWTYPE
     [INDEX BY BINARY_INTEGER];
identifier       type_name;
```

举例：这是一个简单数组

```
...
TYPE ename_table_type IS TABLE OF
                         employees.last_name%TYPE
  INDEX BY BINARY_INTEGER;
ename_table ename_table_type;
...
```

PLSQL中的复杂自定义数据类型

PLSQL内存表应用举例：
下面定义的两个内存表中的元素都是简单数据类型，所以相当于定义了两个简单数组；

```
DECLARE
  TYPE ename_table_type IS TABLE OF
       employees.last_name%TYPE
       INDEX BY BINARY_INTEGER;
  TYPE hiredate_table_type IS TABLE OF DATE
       INDEX BY BINARY_INTEGER;
  ename_table       ename_table_type;
  hiredate_table    hiredate_table_type;
BEGIN
  ename_table(1)    := 'CAMERON';
  hiredate_table(8) := SYSDATE + 7;
    IF ename_table.EXISTS(1) THEN
      INSERT INTO ...
    ...
END;
/
```

备注：对PLSQL内存表中某个元素的访问类似于数组，可以使用下表，因为BINARY_INTEGER这种数据类型的值在-2147483647 ... 2147483647范围内，所以下表也可以在这个范围内。

PLSQL中的复杂自定义数据类型

PLSQL内存表应用举例：
下面定义的两个内存表中的元素记录类型，所以相当于定义了真正的内存表；

```
DECLARE
   TYPE emp_table_type is table of
      employees%ROWTYPE INDEX BY BINARY_INTEGER;
   my_emp_table  emp_table_type;
   v_count       NUMBER(3):= 104;
BEGIN
  FOR i IN 100..v_count
  LOOP
          SELECT * INTO my_emp_table(i) FROM employees
          WHERE employee_id = i;
  END LOOP;
  FOR i IN my_emp_table.FIRST..my_emp_table.LAST
  LOOP
     DBMS_OUTPUT.PUT_LINE(my_emp_table(i).last_name);
  END LOOP;
END;
```

PLSQL中的游标

游标概论：游标是一个私有的SQL工作区域，Oracle数据库中有两种游标，分别是隐式游标和显式游标，隐式游标不易被用户和程序员察觉和意识到，实际上Oracle服务器使用隐式游标来解析和执行我们提交的SQL语句； 而显式游标是程序员在程序中显式声明的；通常我们说的游标均指显式游标。

隐式游标的几个有用属性：

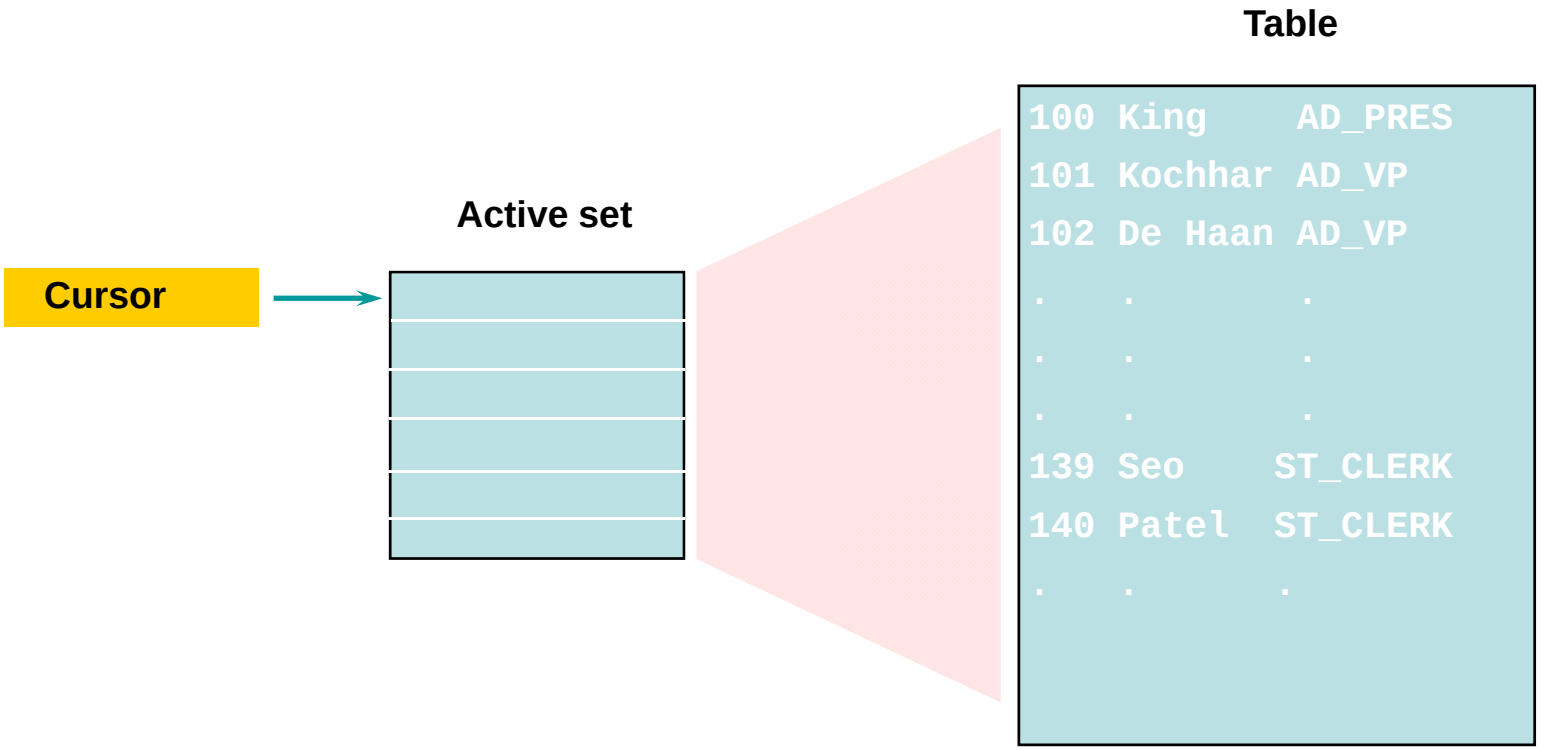
SQL%ROWCOUNT	受最近的SQL语句影响的行数
SQL%FOUND	最近的SQL语句是否影响了一行以上的 数据
SQL%NOTFOUND	最近的SQL语句是否未影响任何数据
SQL%ISOPEN	对于隐式游标而言永远为FALSE

举例：

```
VARIABLE rows_deleted VARCHAR2(30)
DECLARE
  v_employee_id employees.employee_id%TYPE := 176;
BEGIN
  DELETE FROM  employees
  WHERE        employee_id = v_employee_id;
  :rows_deleted := (SQL%ROWCOUNT ||
                       ' row deleted.');
END;
/
PRINT rows_deleted
```

PLSQL中的游标

显式游标：对于返回多行结果的SQL语句的返回结果，可使用显式游标独立的处理器中每一行的数据。



显式游标的相关函数可以做到：

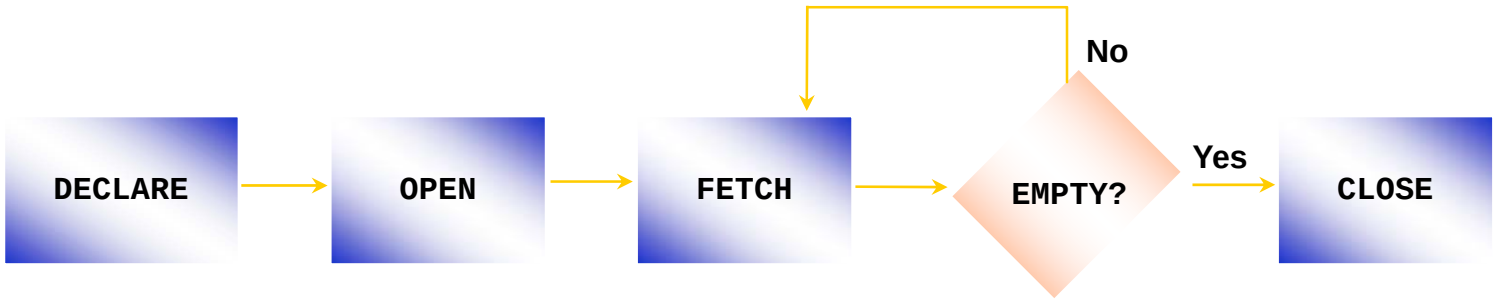
1、一行一行的处理返回的数据。

2、保持当前处理行的一个跟踪，像一个指针一样指示当前的处理的记录。

3、允许程序员在PLSQL块中人为的控制游标的开启、关闭、上下移动；

PLSQL中的游标

在程序中对显式游标控制的一般过程：



- 声明游标
- 打开游标
- 提取当前行到变量
- 测试行的存在
- 如果行存在，则提取
- 关闭游标

PLSQL中的游标

举例：

```
DECLARE
      v_empno  employees.employee_id%TYPE;
      v_ename  employees.last_name%TYPE;
      CURSOR emp_cursor IS
        SELECT employee_id, last_name
        FROM   employees;
    BEGIN
      OPEN emp_cursor;
      LOOP
        FETCH emp_cursor INTO v_empno, v_ename;
        EXIT WHEN emp_cursor%ROWCOUNT > 10 OR
                  emp_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE (TO_CHAR(v_empno)
                              ||' '|| v_ename);
      END LOOP;
      CLOSE emp_cursor;
END ;
```

声明游标

打开游标

提取当前行到变量

测试行存在

关闭游标

PLSQL中的游标

如果你觉得像前面那个例子那样对一个游标的遍历很麻烦的话，可以考虑使用For循环，For循环省去了游标的声明、打开、提取、测试、关闭等语句，对程序员来说很方便，语法如下：

```
FOR record_name IN cursor_name LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

举例1：

```
BEGIN
  FOR emp_record IN (SELECT last_name, department_id
                     FROM   employees) LOOP
         -- implicit open and implicit fetch occur
    IF emp_record.department_id = 80 THEN
       ...
  END LOOP; -- implicit close occurs
END;
```

也可以：

PLSQL中的游标

举例2：

```
DECLARE
  CURSOR emp_cursor IS
    SELECT last_name, department_id
    FROM   employees;
BEGIN
  FOR emp_record IN emp_cursor LOOP
         -- implicit open and implicit fetch occur
    IF emp_record.department_id = 80 THEN
      ...
  END LOOP; -- implicit close occurs
END;
/
```

思考：在什么情况下适用于使用举例1，在哪些情况下适用于举例2？

PLSQL中的游标

游标能否带有参数？答案是肯定的：

```
CURSOR cursor_name
  [(parameter_name datatype, ...)]
IS
  select_statement;
```

举例：

```
DECLARE
  CURSOR emp_cursor
  (p_deptno NUMBER, p_job VARCHAR2) IS
     SELECT employee_id, last_name
     FROM   employees
     WHERE  department_id = p_deptno
     AND    job_id = p_job;
BEGIN
  OPEN emp_cursor (80, 'SA_REP');
  . . .
  CLOSE emp_cursor;
  OPEN emp_cursor (60, 'IT_PROG');
  . . .
END;
```

FOR UPDATE NOWAIT语句：有的时候我们打开一个游标是为了更新或者删除一些记录，这种情况下我们希望在打开游标的时候即锁定相关记录，应该使用**for update nowait**语句，倘若锁定失败我们就停止不再继续，以免出现长时间等待资源的死锁情况。

```
SELECT   ...
FROM              ...
FOR UPDATE [OF column_reference][NOWAIT];
```

举例：

```
DECLARE
  CURSOR emp_cursor IS
    SELECT employee_id, last_name, department_name
    FROM   employees,departments
    WHERE  employees.department_id =
           departments.department_id
    AND employees.department_id = 80
    FOR UPDATE OF salary NOWAIT;
```

PLSQL中的游标

WHERE CURRENT OF cursor：我们经常要逐条处理游标中的每一条记录，在循环体内做Update 或者 Delete 时需要有Where指向游标的当前记录， 有没有简单一点的的Where条件写法呢？ 答案是肯定的，就是。。。

```
DECLARE
CURSOR sal_cursor IS
 SELECT e.department_id, employee_id, last_name, salary
 FROM   employees e, departments d
 WHERE  d.department_id = e.department_id
  and   d.department_id = 60
 FOR UPDATE OF salary NOWAIT;
 BEGIN
 FOR emp_record IN sal_cursor
LOOP
  IF emp_record.salary < 5000 THEN
        UPDATE employees
        SET    salary = emp_record.salary * 1.10
        WHERE CURRENT OF sal_cursor;
  END IF;
END LOOP;
END;
/
```

PLSQL中的例外处理

PLSQL中的例外一般有两种：

1、Oracle 内部错误抛出的例外：这又分为预定义例外（有错误号+常量定义） 和 非预定义例外（仅有错误号，无常量定义）

2、程序员显式的抛出的例外

PLSQL中的例外捕获和传递：与其他语言类似，如果例外在当前块中被处理，则到此为止，否则会被传递到外层（外层BLOCK或者 外层调用者函数）

PLSQL中的例外处理

PLSQL中的例外处理的一般语法：

```
EXCEPTION
  WHEN exception1 [OR exception2 . . .] THEN
    statement1;
    statement2;
    . . .
  [WHEN exception3 [OR exception4 . . .] THEN
    statement1;
    statement2;
    . . .]
  [WHEN OTHERS THEN
    statement1;
    statement2;
    . . .]
```

PLSQL中的例外处理

处理预定义的例外：有些常见例外，Oracle 都已经预定义好了，使用时无需预先声明，比如：

- NO_DATA_FOUND
- TOO_MANY_ROWS
- INVALID_CURSOR
- ZERO_DIVIDE
- DUP_VAL_ON_INDEX

NO_DATA_FOUND 和 TOO_MANY_ROWS 是最常见的例外，大多数Block中都建议对这两种例外有处理； 完整的预定义例外的列表，请参考： *PL/SQL User's Guide and Reference*, "Error Handling."（百度搜索，下载点很多）

```
BEGIN
. . .
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    statement1;
  WHEN TOO_MANY_ROWS THEN
    statement1;
  WHEN OTHERS THEN
    statement1;
    statement2;
END;
```

这两例外总要考虑

OTHERS总在最后

PLSQL中的例外处理

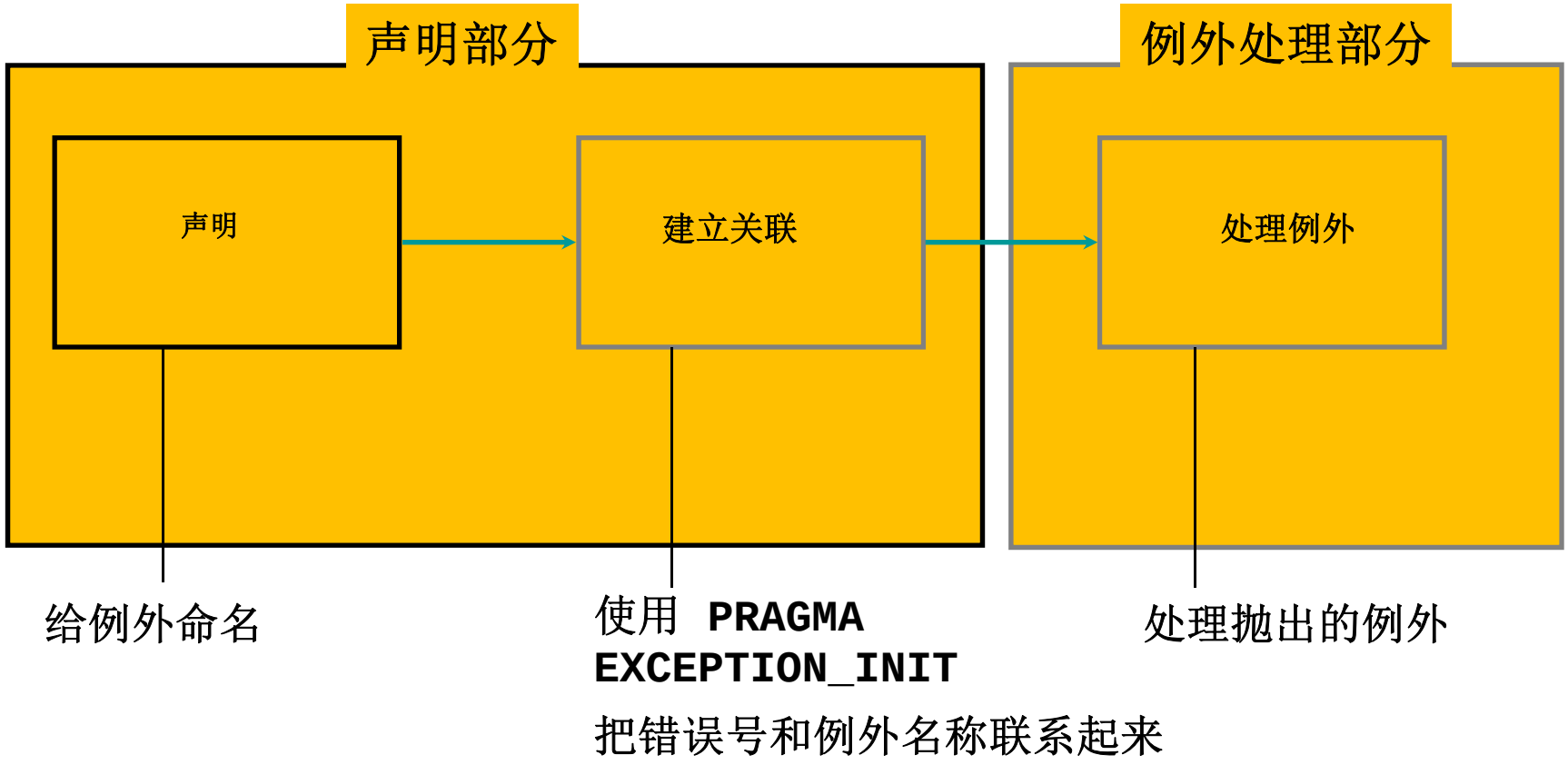
OTHERS的处理： Others表明我们程序员未能预计到这种错误，所以全部归入到others 里面去了，单发生这种情况是，我们还是希望了解当时发生的Oracle错误号和相关描述信息，怎样才能取到呢？

Oracle 提供了两个内置函数 SQLCODE 和 SQLERRM 分别用来返回Oracle 错误号和错误描述

```
DECLARE
  v_error_code      NUMBER;
  v_error_message   VARCHAR2(255);
BEGIN
...
EXCEPTION
...
  WHEN OTHERS THEN
    ROLLBACK;
    v_error_code := SQLCODE ;
    v_error_message := SQLERRM ;
    INSERT INTO errors
    VALUES(v_error_code, v_error_message);
END;
```

PLSQL中的例外处理

处理非预定义的Oracle错误：此类错误属于Oracle错误，有编号，但无错误名称定义，使用时需要先声明，并进行错误初始化：



PLSQL中的例外处理

举例：

```
DEFINE p_deptno = 10
DECLARE
  e_emps_remaining EXCEPTION;
  PRAGMA EXCEPTION_INIT
    (e_emps_remaining, -2292);
BEGIN
  DELETE FROM departments
  WHERE  department_id = &p_deptno;
  COMMIT;
EXCEPTION
  WHEN e_emps_remaining  THEN
   DBMS_OUTPUT.PUT_LINE ('Cannot remove dept ' ||
   TO_CHAR(&p_deptno) || '.  Employees exist. ');
END;
```

1

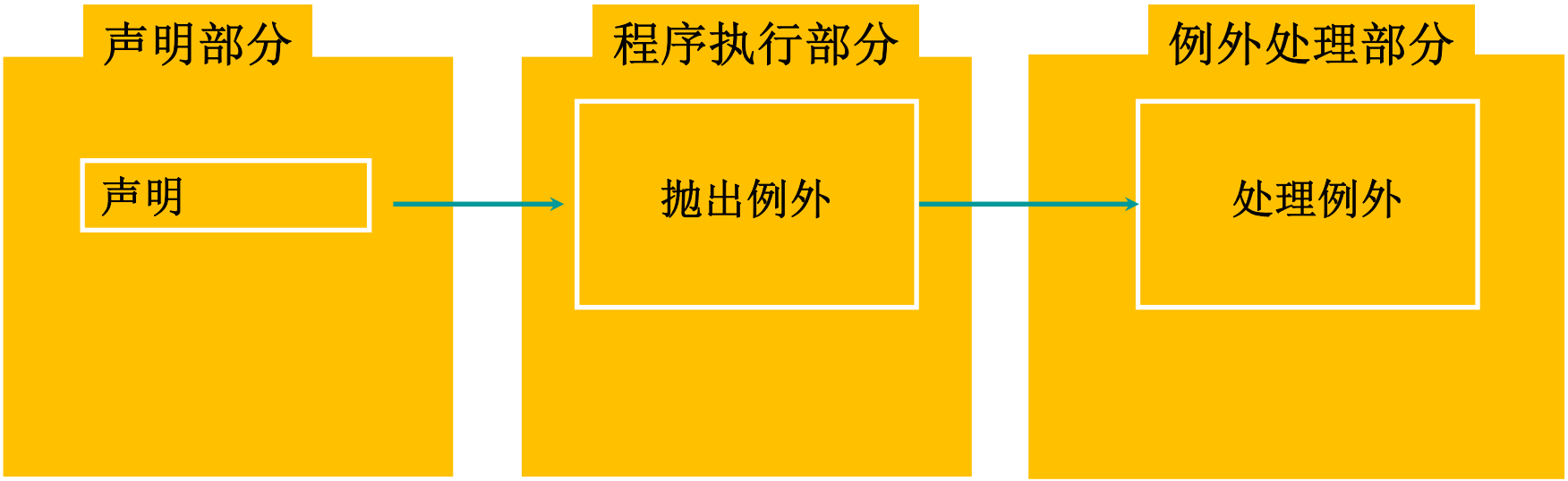
2

3

Oracle内部错误号很多，想了解全部的Ora错误号，请参考：http://www.ora-code.com/

PLSQL中的例外处理

处理用户自定义的错误: 这种错误一般是程序员根据具体的业务逻辑定义的应用类错误，需要先声明后使用：定义和处理过程如下：



给例外命名.

使用**RAISE**语句显式的抛出例外.

处理抛出的例外.

PLSQL中的例外处理

举例

```
DECLARE
  e_invalid_department EXCEPTION;                          1
BEGIN
  UPDATE      departments
  SET         department_name = &p_department_desc
  WHERE       department_id = &p_department_number;
  IF SQL%NOTFOUND THEN
    RAISE e_invalid_department;                            2
  END IF;
  COMMIT;
EXCEPTION
  WHEN e_invalid_department THEN                           3
    DBMS_OUTPUT.PUT_LINE('No such department id.');
END;
```

PLSQL中的例外处理

RAISE_APPLICATION_ERROR() 函数：对于用户自定义的业务错误，如果觉得先定义再使用很麻烦，那么也可以简单的使用raise_application_error() 来简化处理。它可以无需预先定义错误，而在需要抛出错误的地方直接使用此函数抛出例外，例外可以包含用户自定义的错误吗和错误描述；

举例：执行部分

```
BEGIN
...
  DELETE FROM employees
     WHERE  manager_id = v_mgr;
  IF SQL%NOTFOUND THEN
     RAISE_APPLICATION_ERROR(-20202,
       'This is not a valid manager');
  END IF;
   ...
```

例外处理部分：

```
...
EXCEPTION
     WHEN NO_DATA_FOUND THEN
       RAISE_APPLICATION_ERROR (-20201,
         'Manager is not a valid employee.');
END;
```

PLSQL中的例外处理

例外传递：当前块中不处理，传递到外层：

```
DECLARE
  . . .
  e_no_rows      exception;
  e_integrity    exception;
  PRAGMA EXCEPTION_INIT (e_integrity, -2292);
BEGIN
  FOR c_record IN emp_cursor LOOP
    BEGIN
     SELECT ...
     UPDATE ...
     IF SQL%NOTFOUND THEN
       RAISE e_no_rows;
     END IF;
    END;
  END LOOP;
EXCEPTION
  WHEN e_integrity THEN ...
  WHEN e_no_rows THEN ...
END;
```

子块可以直接把例外处理掉，也可以不处理，把它扔到外层去处理.

PLSQL中的存储过程和函数

语法：

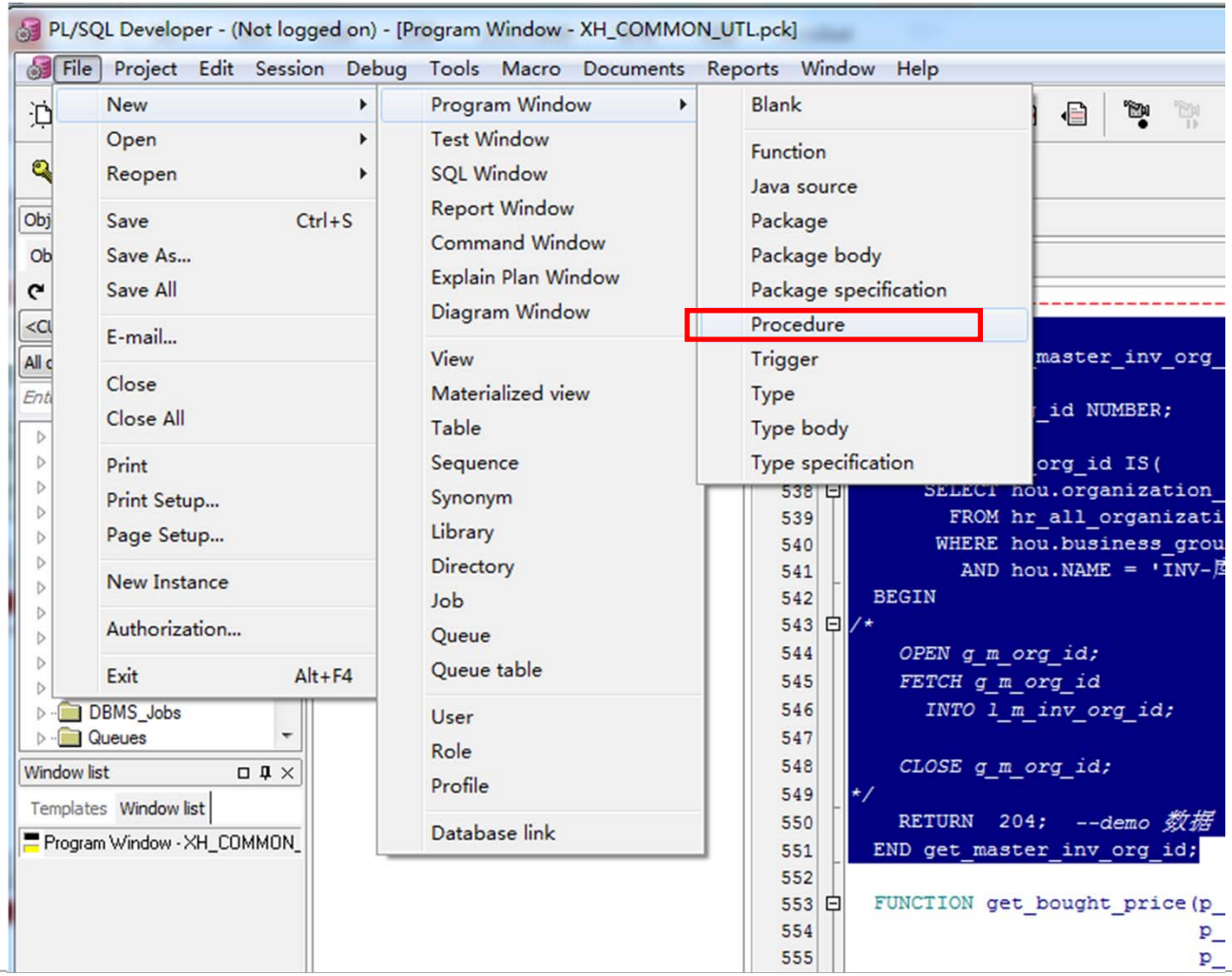
```
CREATE [OR REPLACE] PROCEDURE procedure_name
 [(parameter1 [mode1] datatype1,
  parameter2 [mode2] datatype2,
  . . .)]
IS|AS
PL/SQL Block;
```

举例：

```
CREATE OR REPLACE PROCEDURE raise_salary
  (p_id IN employees.employee_id%TYPE)
IS
BEGIN
  UPDATE employees
  SET    salary = salary * 1.10
  WHERE  employee_id = p_id;
END raise_salary;
/
```

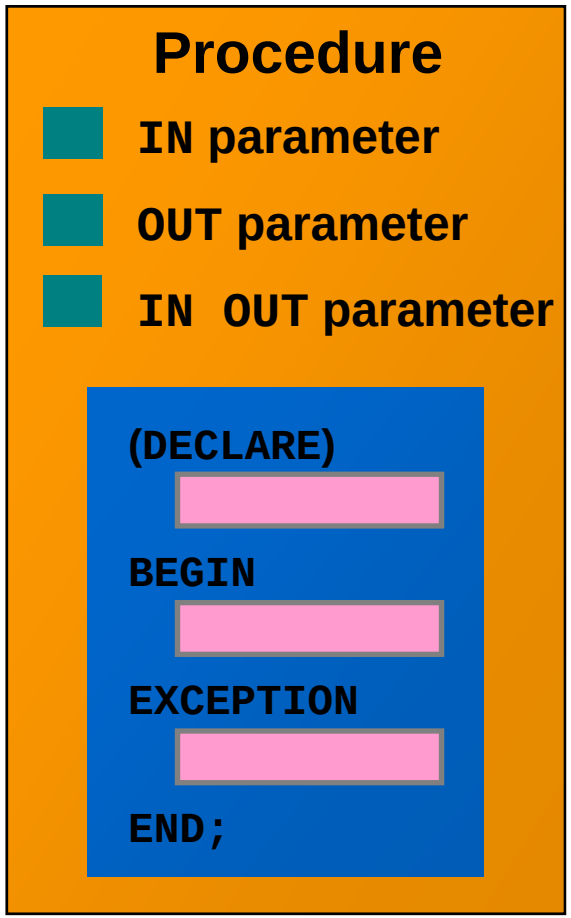
PLSQL中的存储过程和函数

使用**PLSQLDEVELOPER** 开发存储过程：
File->New->Program Window->Procedure



PLSQL中的存储过程和函数

PLSQL存储过程的参数模式：



IN	OUT	IN OUT
默认模式	必须显式指定	必须显式指定
用以把值传给过程	用以把值从过程返回给调用环境	用以把变量传递给过程，并返回给调用环境
参数可以是常数、变量、表达式	必须是个变量个	必须是个变量
可以赋予默认值	不能赋予默认值	不能赋予默认值

PLSQL中的存储过程和函数

举例：

```
CREATE OR REPLACE PROCEDURE query_emp
  (p_id      IN   employees.employee_id%TYPE,
   p_name    OUT  employees.last_name%TYPE,
   p_salary  OUT  employees.salary%TYPE,
   p_comm    OUT  employees.commission_pct%TYPE)
IS
BEGIN
  SELECT   last_name, salary, commission_pct
   INTO    p_name, p_salary, p_comm
   FROM    employees
   WHERE   employee_id = p_id;
END query_emp;
/
```

```
CREATE OR REPLACE PROCEDURE format_phone
  (p_phone_no IN OUT VARCHAR2)
IS
BEGIN
  p_phone_no := '(' || SUBSTR(p_phone_no,1,3) ||
                ')' || SUBSTR(p_phone_no,4,3) ||
                '-' || SUBSTR(p_phone_no,7);
END format_phone;
/
```

PLSQL中的存储过程和函数

参数传递方式（按顺序传递 或者 使用=>符号传递）

```
CREATE OR REPLACE PROCEDURE add_dept
  (p_name  IN departments.department_name%TYPE
                                    DEFAULT 'unknown',
   p_loc   IN departments.location_id%TYPE
                                    DEFAULT 1700)
IS
BEGIN
  INSERT INTO departments(department_id,
              department_name, location_id)
  VALUES (departments_seq.NEXTVAL, p_name, p_loc);
END add_dept;
/
```

```
BEGIN
  add_dept;
  add_dept ('TRAINING', 2500);
  add_dept ( p_loc => 2400, p_name =>'EDUCATION');
  add_dept ( p_loc => 1200) ;
END;
/
SELECT department_id, department_name, location_id
FROM departments;
```

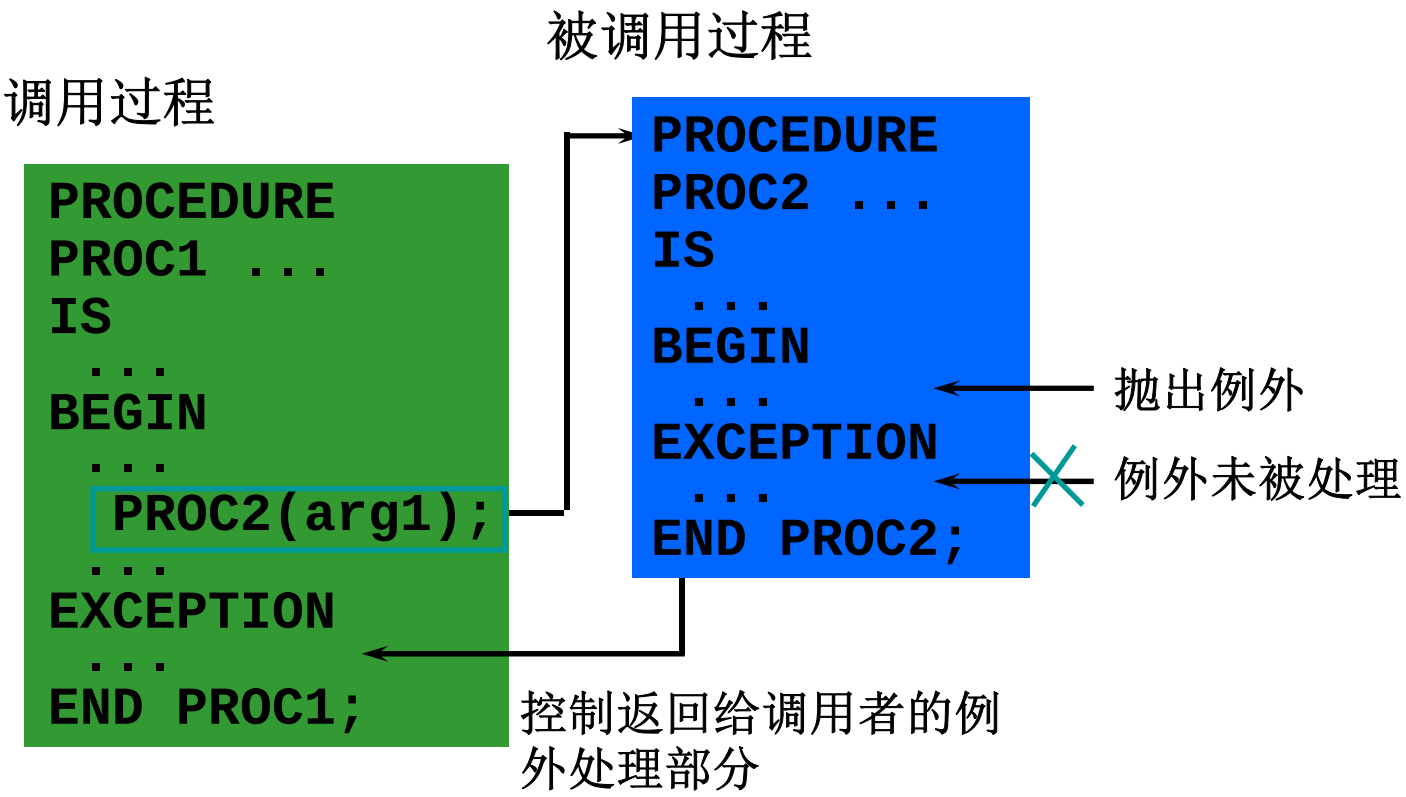
使用默认值

按顺序传递

使用=>符号传递，无顺序要求。

PLSQL中的存储过程和函数

过程调用的例外处理：



PLSQL中的存储过程和函数

删除储存过程：

```
DROP PROCEDURE procedure_name
```

PLSQL中的存储过程和函数

PLSQL存储函数：

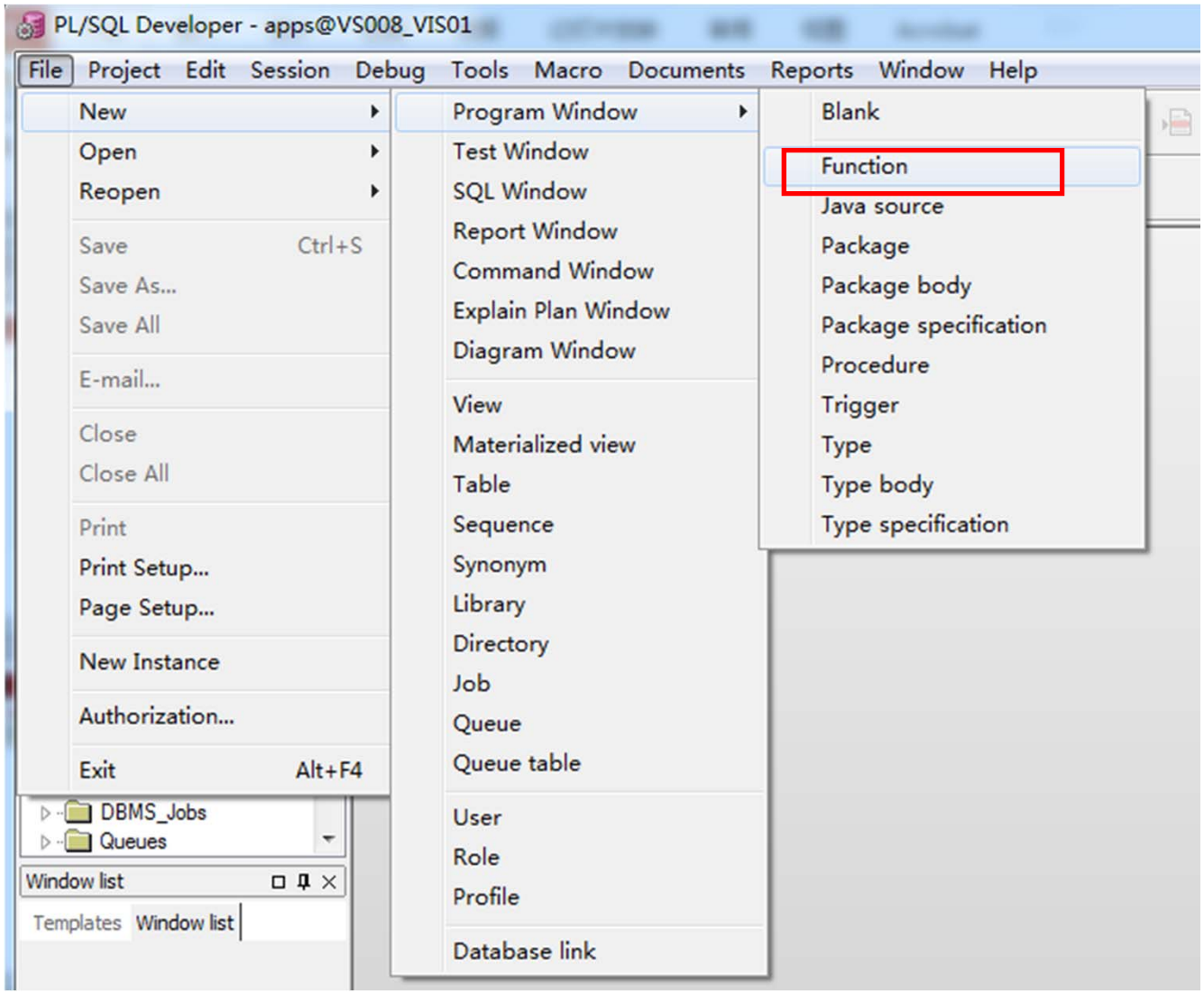
```
CREATE [OR REPLACE] FUNCTION function_name
 [(parameter1 [mode1] datatype1,
  parameter2 [mode2] datatype2,
  . . .)]
RETURN datatype
IS|AS
PL/SQL Block;
```

```
CREATE OR REPLACE FUNCTION get_sal
      (p_id  IN employees.employee_id%TYPE)
      RETURN NUMBER
IS
      v_salary employees.salary%TYPE :=0;
BEGIN
      SELECT salary
      INTO   v_salary
      FROM   employees
      WHERE  employee_id = p_id;
     RETURN v_salary;
END get_sal;
/
```

与存储过程的差异

PLSQL中的存储过程和函数

使用**PLSQLDeveloper** 开发函数：



PLSQL中的存储过程和函数

哪些SQL语句中可以使用用户自定义的函数：

- Select 语句
- Where条件和Having子句
- CONNECT BY, START WITH, ORDER BY, 和GROUP BY 子句
- INSERT的Values子句
- UPDATE的Set子句

举例：

```
SELECT  employee_id, tax(salary)
FROM    employees
WHERE tax(salary)>(SELECT MAX(tax(salary))
FROM employees WHERE department_id = 30)
ORDER BY tax(salary) DESC;
```

PLSQL中的存储过程和函数

想要在SQL语句中可以使用用户自定义的函数，那么这样的用户定义函数有哪些限制？

答: 有如下限制：

- 必须是个函数（不能是过程-Procedure）
- 只能用IN 模式的参数（不能有OUT, IN OUT 模式的参数）
- 只能接收SQL数据类型的参数，不能接收PLSQL 中特有的参数（比如记录、PLSQL内存表）
- 函数返回的数据类型也必须是有效的数据类型，而不能是PLSQL特有的数据类型
- 在SQL中使用的函数，其函数体内部不能有DML语句。
- 在UPDATE/DELETE语句中调用的函数，其函数体内部不能有针对同一张表的查询语句
- 在SQL中调用的函数，其函数体内部不能有事务结束语句（比如Commit，Rollback)

PLSQL中的存储过程和函数

反面教材举例：

```
CREATE OR REPLACE FUNCTION dml_call_sql (p_sal NUMBER)
   RETURN NUMBER IS
BEGIN
   INSERT INTO employees(employee_id, last_name, email,
                         hire_date, job_id, salary)
          VALUES(1, 'employee 1', 'emp1@company.com',
                          SYSDATE, 'SA_MAN', 1000);
   RETURN (p_sal + 100);
END;
/
```

Function created.

```
UPDATE employees SET salary = dml_call_sql(2000)
 WHERE employee_id = 170;
```

UPDATE employees SET salary = dml_call_sql(2000)
*

ERROR at line 1:
ORA-04091: table PLSQL.EMPLOYEES is mutating, trigger/function may not see it
ORA-06512: at "PLSQL.DML_CALL_SQL", line 4

PLSQL中的存储过程和函数

删除储存函数：

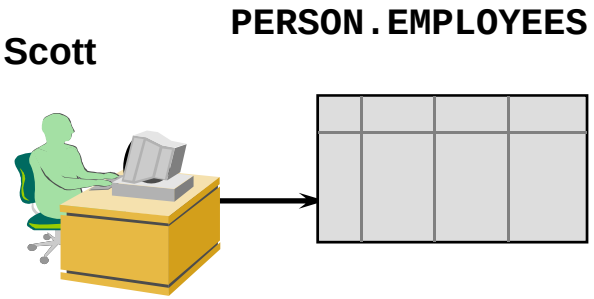
```
DROP FUNCTION function_name
```

PLSQL中的存储过程和函数

函数过程对数据访问的权限概念：定义者权限 和 调用者权限

PERSON给Scott赋予Select权限：

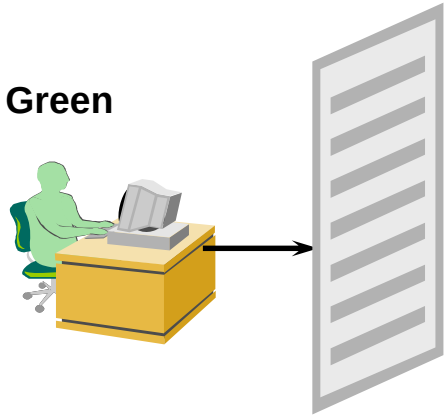
```
GRANT SELECT
 ON    employees
 TO    scott;
Grant Succeeded.
```



创建QUERY_EMP()函数访问Person.employee表

Scott给Green赋予函数执行权限：

```
GRANT EXECUTE
 ON    query_emp
 TO    green;
Grant Succeeded.
```



PLSQL中的存储过程和函数

定义者权限：函数执行时，对表的访问默认使用定义者权限。

那么什么情况会使用调用者权限呢？这需要在写函数的时候有特殊语句标识：**AUTHID CURRENT_USER**

```
CREATE PROCEDURE query_emp
(p_id IN employees.employee_id%TYPE,
 p_name OUT employees.last_name%TYPE,
 p_salary OUT employees.salary%TYPE,
 p_comm OUT  employees.commission_pct%TYPE)
AUTHID CURRENT_USER
IS
BEGIN
  SELECT last_name, salary,
         commission_pct
   INTO  p_name, p_salary, p_comm
   FROM  employees
   WHERE employee_id=p_id;
END query_emp;
/
```

大家想一下在这种情况下：**Green执行函数有没有问题？**

PLSQL中的存储过程和函数

课堂测试：

赋予系统权限：

```
--system/manager

GRANT create session
    , create table
    , create procedure
    , create sequence
    , create trigger
    , create view
    , create synonym
    , alter session
TO ora1,ora2,ora3;
```

Ora1用户创建测试表：

```
--ora1/oracle
 create table testora_1
(t_id number ,
 t_name varchar2(100)
 );

alter table testora_1
  add constraint testora_1_PK
      primary key (t_id);

grant select on  testora_1 to ora2;
```

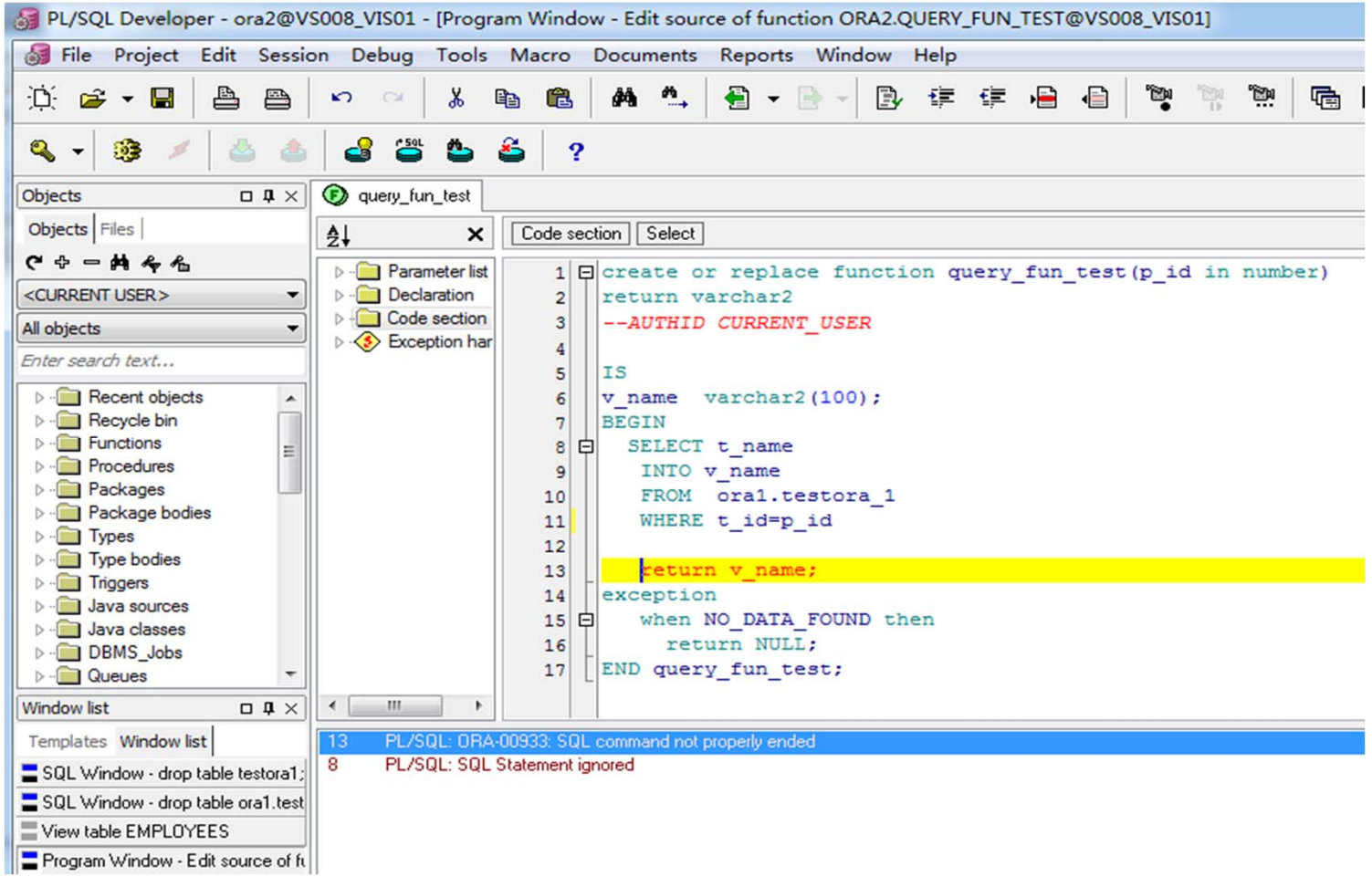
Ora2用户创建函数：

```
--ora2/oracle
CREATE or replace function
query_fun_test(p_id in number)
return varchar2
IS
v_name  varchar2(100);
BEGIN
  SELECT t_name
   INTO v_name
   FROM  ora1.testora_1
   WHERE t_id=p_id;
   return v_name;
exception
   when NO_DATA_FOUND then
     return NULL;
END query_fun_test;

select query_fun_test(1) from dual;

grant execute on query_fun_test to ora3;
```

测试1：在Ora3用户下执行 ora2.query_fun_test()函数；
测试2：更改ora2.query_fun_test()函数为
AUTHID CURRENT_USER，然后再重复“测试1”

PLSQL中的存储过程和函数

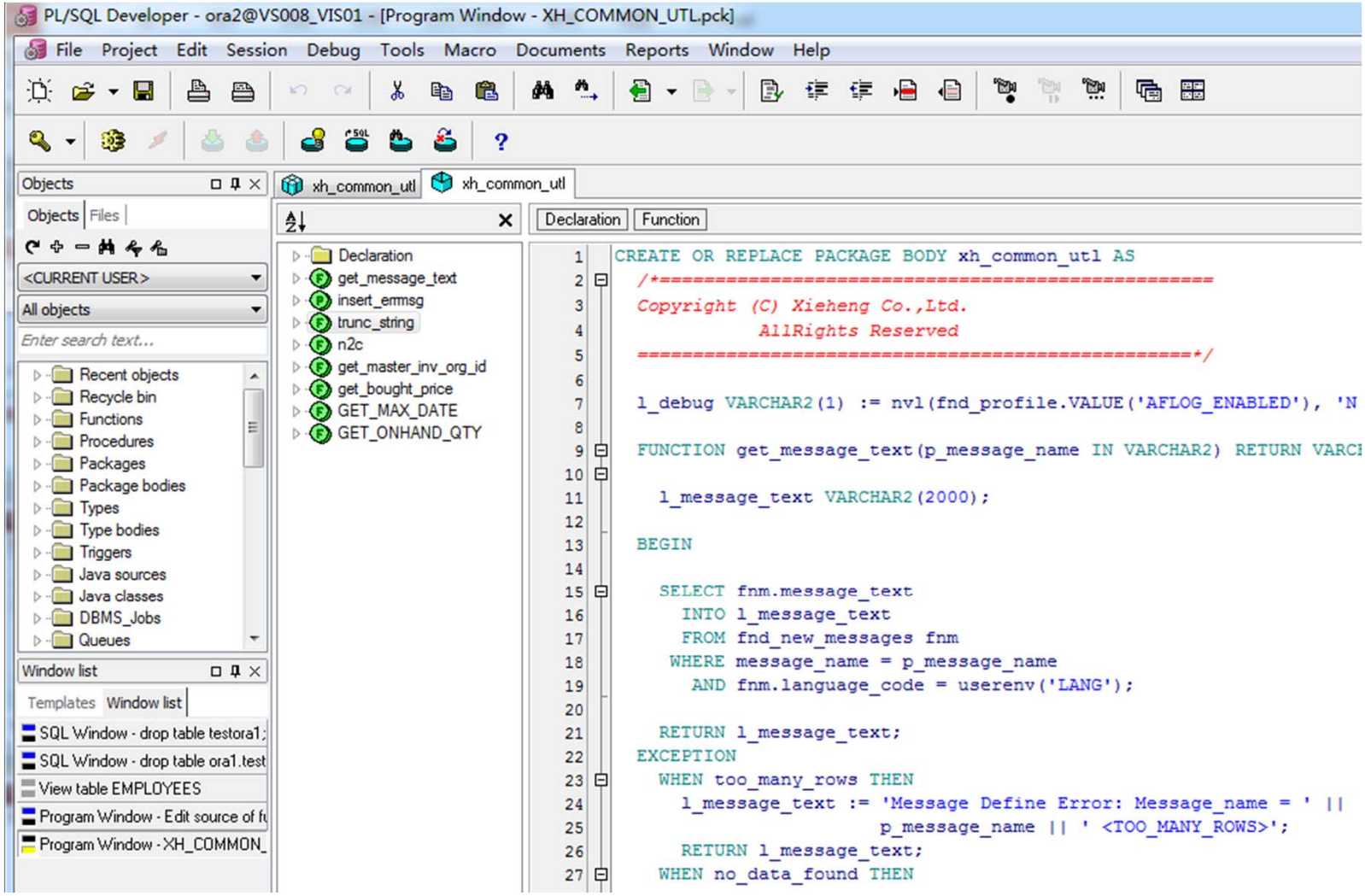
编译PLSQL程序：使用PLSQL Developer集成环境编译：方便；



注意在SQL Window中执行 函数或过程的定义，即使有编译错误，也不会显示；
只有在Program Window中执行 函数或过程的定义的时候才会立即显示编译错误；

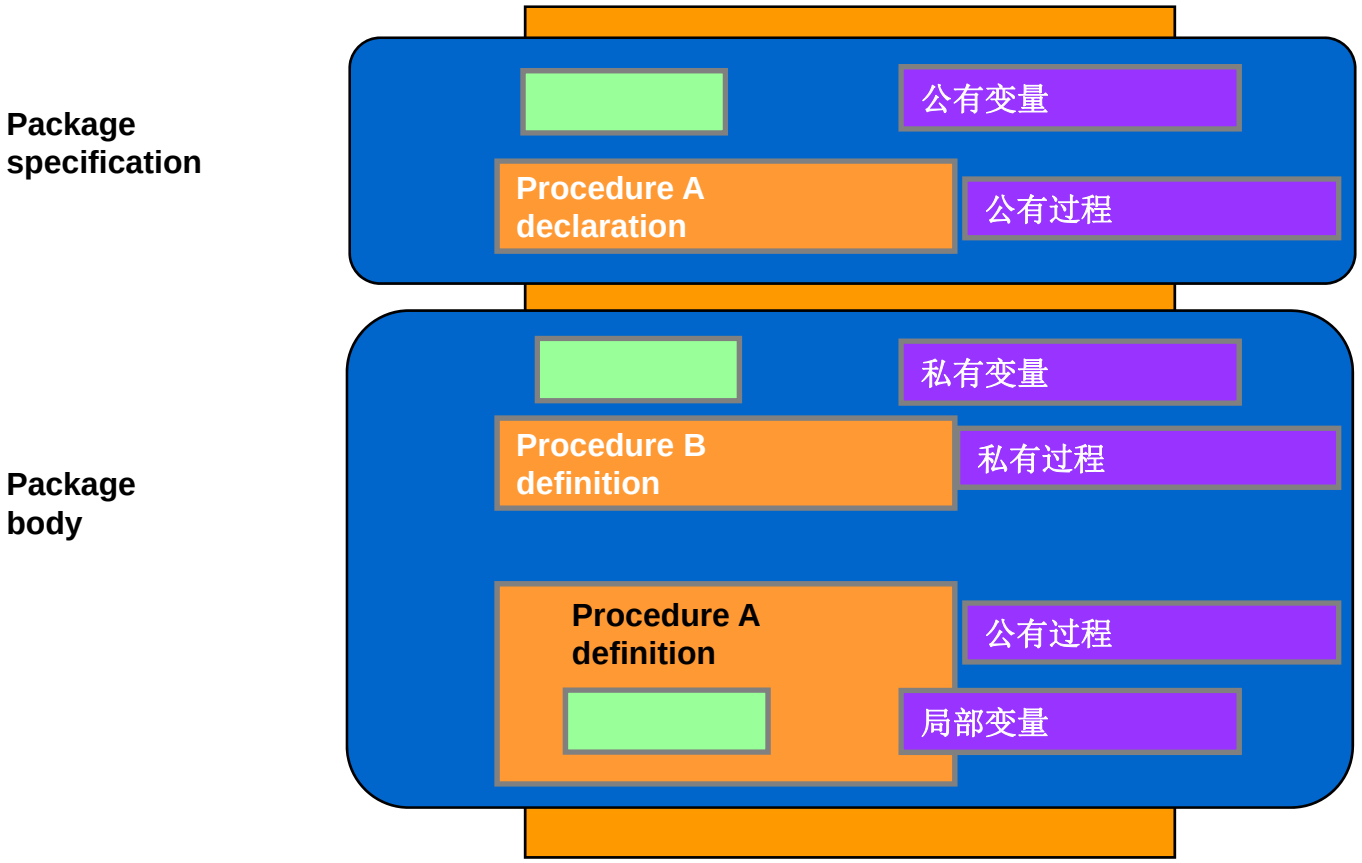
PLSQL中的包PACKAGE

Package概念：按照业务逻辑、把相关的Func , Procedure 组织到一起，形成一个函数或者过程集合，这就是一个Package，这是PLSQL中程序的一种组织形式。也是我们写PLSQL最主要的形式；



PLSQL中的包PACKAGE

Package组成：Package由包说明（package Specification)和包体(package body)两部分构成； 包说明部分相当于C语言里面的.H文件，包体部分相当于 C语言里面针对.H实现的C文件。



PLSQL中的包PACKAGE

Package常用SQL：

CREATE [OR REPLACE] PACKAGE	Create (or modify) an existing package specification
CREATE [OR REPLACE] PACKAGE BODY	Create (or modify) an existing package body
DROP PACKAGE	Remove both the package specification and the package body
DROP PACKAGE BODY	Remove the package body only

Package好处：

1、模块化：一般把有相关性的函数和过程放到一个Package中；
2、易设计：可以把包说明和包体分别编写和编译，先编写和编译包说明部分，在编写和说明包体部分；这有利于分工合作；
3、信息隐藏：包体中函数可以部分出现在包说明中，只有出现在包说明中的函数和过程才是该Package的公有函数和过程，可以被其他包中的函数调用，否则对其他包中的函数是不可见的，未在包说明部分出现的函数和过程相当于私有的。
4、加载性能提高：当Package中有一个函数或过程被调用时，整个Packege就被加载到内存中，这样当该Package中其他函数被调用时，就直接从内存读取了，可以减少磁盘IO,从而提高性能。 这个特性也提醒我们不要去搞巨无霸的Package, 把你用到的任何 函数都写到一个Package中，这会导致严重的内存浪费。
5、重载：一个package 中可以定义同名、不同参数的函数或过程。

PLSQL中的包PACKAGE

Package中的向前声明特性：在Package body中，一个函数中调用另一个函数（也在该Package中），则另一个函数必须在前面先定义；如果你非要调用在程序代码中后定义的函数，可把这个函数设置成公有函数，在包说明部分说明；

```
CREATE OR REPLACE PACKAGE BODY forward_pack
IS
  PROCEDURE award_bonus(. . .)
  IS
  BEGIN
    calc_rating(. . .);        --非法引用

  END;

  PROCEDURE calc_rating(. . .)
  IS
  BEGIN
    ...
  END;

END forward_pack;
/
```

PLSQL中的包PACKAGE

Package中的初始化过程代码：

```
CREATE OR REPLACE PACKAGE taxes
IS
    tax   NUMBER;
  ...  -- declare all public procedures/functions
END  taxes;
/
```

```
CREATE OR REPLACE PACKAGE BODY taxes
IS
  ... -- declare all private variables
  ... -- define public/private procedures/functions
BEGIN
   SELECT   rate_value
   INTO     tax
   FROM     tax_rates
   WHERE    rate_name = 'TAX';
END taxes;
/
```

Package中可以写一段初始化过程代码，这段代码只是一个Session中加载时被执行一次，一般用于一些复杂变量的初始化（比如某个公有变量的初始化值是需要通过一段负责的SQL来获取的）；如果我们没有这样的初始化代码需求，那么一般可以在这部分写个NULL;就可以了。

PLSQL中的包PACKAGE

Package中的变量的持久状态：

```
CREATE OR REPLACE PACKAGE comm_package IS
  g_comm  NUMBER := 10;          --初始化为 10
  PROCEDURE reset_comm (p_comm   IN   NUMBER);
END comm_package;
/
```

```
CREATE OR REPLACE PACKAGE BODY comm_package IS

   FUNCTION  validate_comm (p_comm  IN  NUMBER) RETURN BOOLEAN
   IS v_max_comm    NUMBER;
   BEGIN
    ...      -- 校验 commission是否低于数据库表中的最大ommission，是则返回TRUE
   END validate_comm;

   PROCEDURE  reset_comm (p_comm   IN  NUMBER)
   IS BEGIN
     ...     -- 调用 validate_comm 判断是否超出最大值，若是则
             -- RAISE_APPLICATION_ERROR; 否则设置g_comm := p_comm
END reset_comm;
END comm_package;
/
```

PLSQL中的包PACKAGE

时间	数据库用户：Scott	数据库用户：Jones
9:00	EXECUTE comm_package.reset_comm (0.25) --max_comm=0.4 > 0.25	
9:30	--g_comm = 0.25	INSERT INTO employees (last_name, commission_pct) VALUES ('Madonna', 0.8); --max_comm=0.8
9:35		EXECUTE comm_package.reset_comm(0.5)
10:00		--max_comm=0.8 > 0.5 --g_comm = 0.5
	Select comm_package.g_comm from dual; -- g_comm=?	
11:00	EXECUTE	
11:01	comm_package.reset_comm (0.6) --max_comm=0.4 < 0.6 INVALID	
		ROLLBACK; EXIT

思考：Package中的公共变量，不同的Session是否会相互影响？

内置PLSQL工具包

动态SQL：不是在Designer Time写的SQL, 而是可以在运行时临时拼接起来的SQL语句；

动态SQL可以使用Oracle 内置包 DBMS_SQL 来执行，也可以使用EXECUTE IMMEDIATE 语句来执行：

DBMS_SQL 执行例子：

```
CREATE OR REPLACE PROCEDURE delete_all_rows
   (p_tab_name IN VARCHAR2, p_rows_del OUT NUMBER)
 IS
   cursor_name   INTEGER;
 BEGIN
   cursor_name := DBMS_SQL.OPEN_CURSOR;
   DBMS_SQL.PARSE(cursor_name, 'DELETE FROM '||p_tab_name,
                               DBMS_SQL.NATIVE );
   p_rows_del := DBMS_SQL.EXECUTE (cursor_name);
   DBMS_SQL.CLOSE_CURSOR(cursor_name);
END;
/
```

内置PLSQL工具包

EXECUTE IMMEDIATE 执行例子：

```
CREATE PROCEDURE del_rows
  (p_table_name  IN  VARCHAR2,
   p_rows_deld   OUT NUMBER)
IS
BEGIN
  EXECUTE IMMEDIATE 'delete from '||p_table_name;
  p_rows_deld := SQL%ROWCOUNT;
END;
/
```

动态SQL实际应用例子：POS基础数据同步并发程序 dpos_base_sync.pck

好处： 在某些应用环境中可以做到灵活配置，避免因为添加一种业务而更改代码或者新建代码；

缺点： 调试及阅读理解困难

内置PLSQL工具包

程序中执行DDL：如果想在程序中执行DDL,可使用Oracle 内置包：DBMS_DDL

比如在程序中执行编译命令：

```
DBMS_DDL.ALTER_COMPILE('PROCEDURE','A_USER','QUERY_EMP')
```

比如在程序中执行数据收集命令：

```
DBMS_DDL.ANALYZE_OBJECT('TABLE','A_USER','JOBS','COMPUTE')
```

内置PLSQL工具包

Oracle数据库JOB：定义JOB 可以定期执行某个程序，
应用场景：比如每隔一周对某些表进行数据收集，以确保CBO正确，又比如在消息处理机制中，每隔5分钟对消息队列进行扫描处理等。

Oracle提供内置包 DBMS_JOB，可完成JOB的定义、提交、更改、停止、移除。

例子：提交一个JOB 每隔1天执行一次：

```
DECLARE
    jobno NUMBER;
BEGIN
  DBMS_JOB.SUBMIT (
   job  => jobno ,
   what => 'OVER_PACK.ADD_DEPT(''EDUCATION'',2710);',
   next_date =>  TRUNC(SYSDATE + 1),
   interval  =>  'TRUNC(SYSDATE + 1)'
  );
  dbms_output.put_line('job_no =' || jobno )
  COMMIT;
END;
```

例子：更改JOB的执行频率为：每4小时执行一次

```
    BEGIN
       DBMS_JOB.CHANGE(1, NULL, TRUNC(SYSDATE+1)+6/24, 'SYSDATE+4/24');
    END;
```

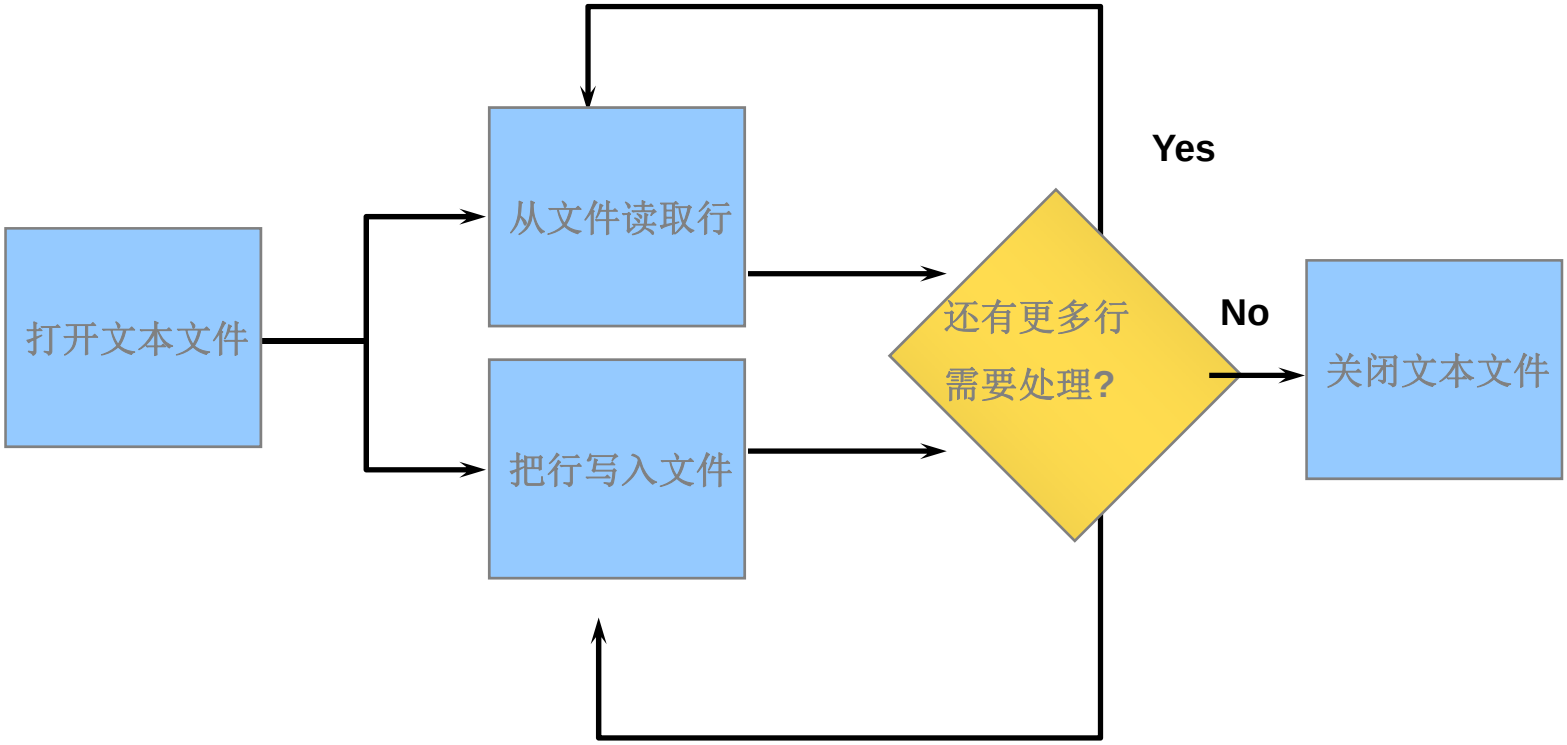
内置PLSQL工具包

如何找到自己提交的JOB号：

```
SELECT   job, log_user, next_date, next_sec, broken, what   FROM DBA_JOBS;
```

内置PLSQL工具包

PLSQL中读写外部文件：Oracle 提供内置包UTL_FILE来读写外部文件，其一般处理过程为：



我们来看个例子：

内置PLSQL工具包

UTL_FILE 应用举例：

```
CREATE OR REPLACE PROCEDURE sal_status
(p_filedir IN VARCHAR2,  p_filename IN VARCHAR2)
IS
 v_filehandle UTL_FILE.FILE_TYPE;
 CURSOR emp_info IS
   SELECT last_name, salary, department_id
    FROM employees
    ORDER BY department_id;
 v_newdeptno employees.department_id%TYPE;
 v_olddeptno employees.department_id%TYPE := 0;
BEGIN
 v_filehandle := UTL_FILE.FOPEN (p_filedir, p_filename,'w');
 UTL_FILE.PUTF (v_filehandle,'SALARY REPORT: GENERATED ON
                              %s\n', SYSDATE);
 UTL_FILE.NEW_LINE (v_filehandle);
 FOR v_emp_rec IN emp_info LOOP
  v_newdeptno := v_emp_rec.department_id;
...
```

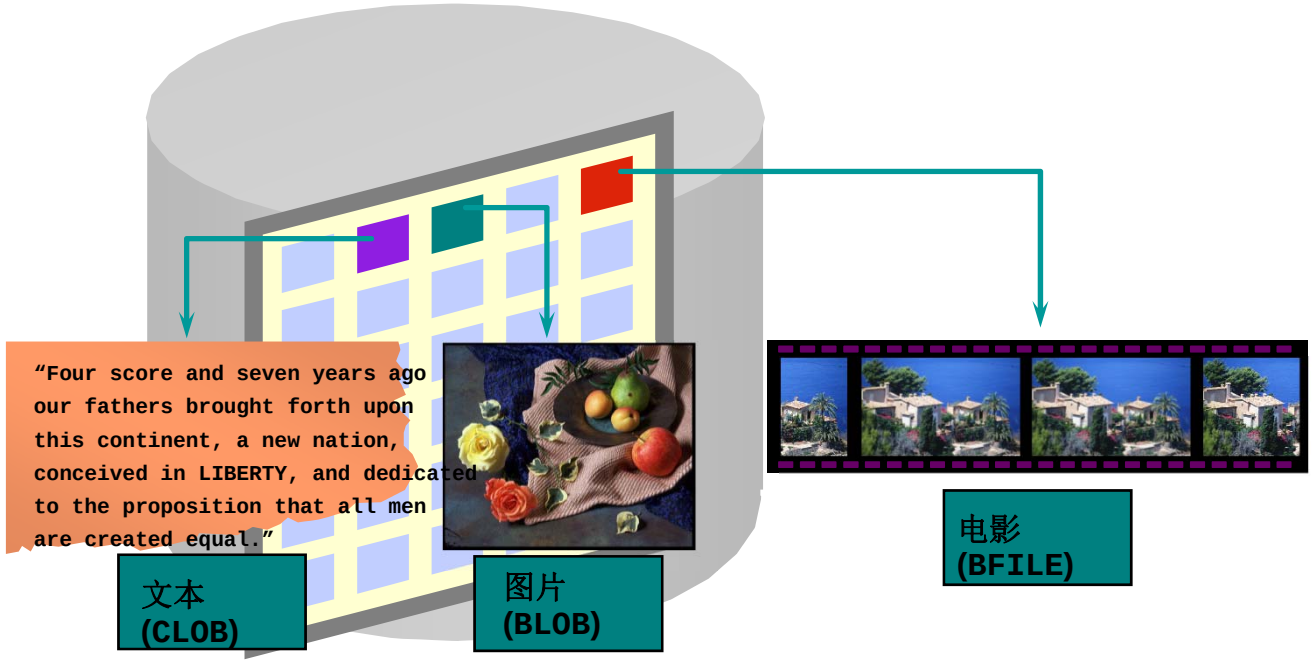
内置PLSQL工具包

UTL_FILE 应用举例（续）：

```
    ...
    IF v_newdeptno <> v_olddeptno THEN
      UTL_FILE.PUTF (v_filehandle, 'DEPARTMENT: %s\n',
                     v_emp_rec.department_id);
    END IF;
    UTL_FILE.PUTF (v_filehandle,'  EMPLOYEE: %s earns: %s\n',
                   v_emp_rec.last_name, v_emp_rec.salary);
    v_olddeptno := v_newdeptno;
  END LOOP;
  UTL_FILE.PUT_LINE (v_filehandle, '*** END OF REPORT ***');
  UTL_FILE.FCLOSE (v_filehandle);
EXCEPTION
  WHEN UTL_FILE.INVALID_FILEHANDLE THEN
    RAISE_APPLICATION_ERROR (-20001, 'Invalid File.');
  WHEN UTL_FILE.WRITE_ERROR THEN
   RAISE_APPLICATION_ERROR (-20002, 'Unable to write to
                                     file');
END sal_status;
/
```

PLSQL中大对象的操作

LOB概念回顾：大对象类型用于存储非结构化的大数据，比如大文本、图片、电影、音乐等



Oracle数据库里面的LOB有四种类型：

1、CLOB ：字符大对象，存储在数据库内部；
2、NCLOB：多字节字符大对象，存储在数据库内部；
3、BLOB：二进制大对象，存储在数据库内部；
4、BFILE：二进制文件，存储在数据库外部；

PLSQL中大对象的操作

LONG 和 LOB的区别

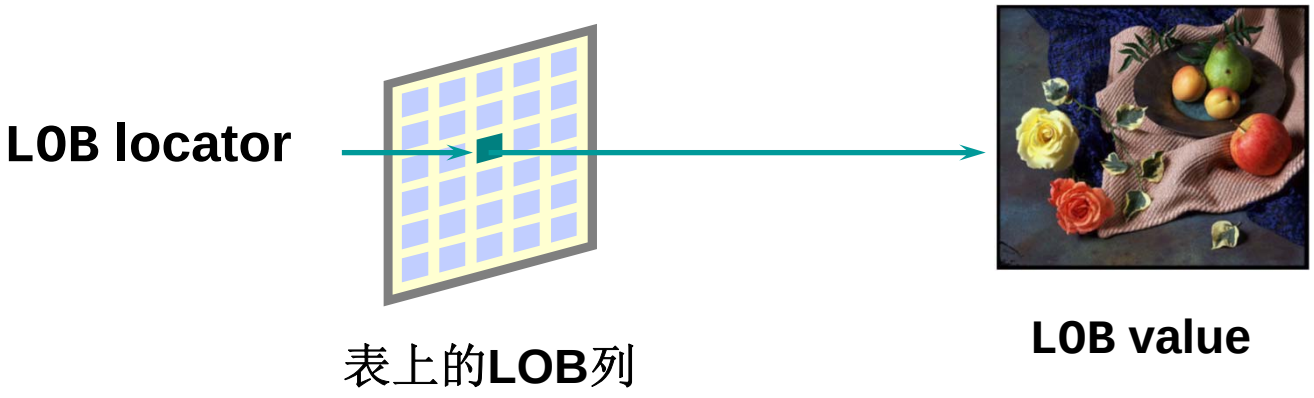
LONG and LONG RAW	LOB
Single LONG column per table	Multiple LOB columns per table
Up to 2 GB	Up to 4 GB
SELECT returns data	SELECT returns locator
Data stored in-line	Data stored in-line or out-of-line
Sequential access to data	Random access to data

Long 和 Long RAW 是Oracle 9i以前的版本使用的大对象类型；在9i以后都建议使用LOB,Oracle 9i 也提供了一系列函数用以从LONG 升级到LOB

OUT-of-LINE 的理解： LOB可以是对象类型的属性，而LONG不行；

PLSQL中大对象的操作

LOB数据的存储方式：LOB 分为 Value 和 Locator 两部分，在我们的数据库表上的LOB字段，肯定会存储LOB的Locator; 至于LOB的value,则要看其内外部类型和大小决定存储位置，默认情况下，内部LOB小于4000字节会被存储在同一行上，超过部分则会存储在数据库内部的其他地方，这都有Oracle数据库自动管理。



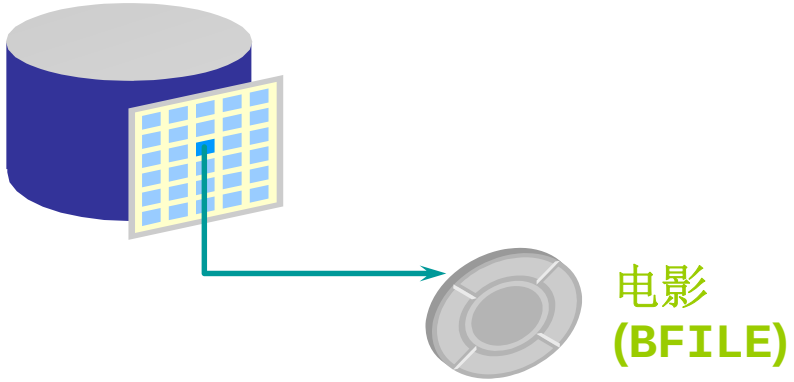
内部LOB的一般操作步骤为：

1、在表中添加LOB类型的列
2、在程序中声明和初始化LOB的Locator
3、使用 SELECT FOR UPDATE 锁定目标行，准备更新行上的LOB列（LOB的Locator）
4、生成LOB对象，可使用DBMS_LOB 这样的PLSQL包，也可以使用OCI,JDBC等；
5、Commit 提交更改；

PLSQL中大对象的操作

外部LOB Bfile 的操作：

Bfile是数据库外部文件，在数据库表上这种类型的字段实际只是存储一个Locator



Bfile的使用限制：Bfile是数据库外部文件，是只读的，所以不参与事务操作；用户必须先创建文件并放到特定的目录下，给予Oracle进程以目录和文件的读取权限；
在Oracle 中删除Bfile这样的LOB数据的时候，它并没有实际的去删除对应的操作系统上的文件，实际文件的删除是DBA,系统管理员的工作，Bfile的大小限制取决于操作系统，不受Oracle限制。

PLSQL中大对象的操作

Oracle Directory：为了便于控制Bfile存储的安全性，Oracle 数据库引入了Oracle Directory的概念；

在Oracle 内部创建的 Directory 默认的所有者是sys, 并有DBA(或者是另一个拥有 CREATE ANY DIRECTORY 权限的用户)创建；Directory 对象可以像表那样给其他用户赋权。

使用Bfile的一般步骤：

1、在操作系统上创建目录，并给Oracle数据库进程赋予阅读权限，把外部文件放入这个目录
2、在Oracle数据库中表添加Bfile类型字段

```
ALTER TABLE employees
  ADD emp_video BFILE;
```

3、在Oracle 数据库中创建Directory 对象

```
CREATE DIRECTORY dir_name
  AS os_path;
```

4、授权读权限给特定的数据库用户
5、往表中插入数据时使用 BFILENAME 函数，它可以关联外部文件和表上的Bfile列
6、在程序中声明和初始化LOB的Locator
7、Select 指定行上 Bfile 列 到Locator
8、使用 DBMS_LOB 或者通过 OCI 读取Bfile (使用Locator作为文件的一个引用）

PLSQL中大对象的操作

使用Bfile的一般步骤：

1、在操作系统上创建目录，并给Oracle数据库进程赋予阅读权限，把外部文件放入这个目录
2、在Oracle数据库中表添加Bfile类型字段

```
ALTER TABLE employees
  ADD emp_video BFILE;
```

3、在Oracle 数据库中创建Directory 对象

```
CREATE DIRECTORY dir_name
  AS os_path;
```

4、授权读权限给特定的数据库用户

```
GRANT READ ON DIRECTORY dir_name TO user|role|PUBLIC;
```

5、往表中插入数据时使用 BFILENAME 函数，它可以关联外部文件和表上的Bfile列

```
FUNCTION BFILENAME (directory_alias IN VARCHAR2,
                    filename IN VARCHAR2)
RETURN BFILE;
```

```
UPDATE employees
     SET emp_video =
BFILENAME('LOG_FILES', 'King.avi'
     WHERE employee_id = 100;
```

6、在程序中声明和初始化LOB的Locator
7、Select 指定行上 Bfile 列 到Locator
8、使用 DBMS_LOB 或者通过 OCI 读取Bfile (使用Locator作为文件的一个引用)

PLSQL中大对象的操作

举例：

```
CREATE OR REPLACE PROCEDURE load_emp_bfile
                  (p_file_loc IN VARCHAR2) IS
  v_file      BFILE;
  v_filename  VARCHAR2(16);
  CURSOR emp_cursor IS
    SELECT first_name FROM employees
    WHERE department_id = 60 FOR UPDATE;
BEGIN
  FOR emp_record IN emp_cursor LOOP
    v_filename := emp_record.first_name || '.bmp';
    v_file := BFILENAME(p_file_loc, v_filename);
    DBMS_LOB.FILEOPEN(v_file);
    UPDATE employees SET emp_video = v_file
      WHERE CURRENT OF emp_cursor;
    DBMS_OUTPUT.PUT_LINE('LOADED FILE: '||v_filename
      || ' SIZE: ' || DBMS_LOB.GETLENGTH(v_file));
    DBMS_LOB.FILECLOSE(v_file);
  END LOOP;
END load_emp_bfile;
/
```

PLSQL中大对象的操作

使用 **DBMS_LOB.FILEEXISTS** 测试文件是否存在：

```
CREATE OR REPLACE PROCEDURE load_emp_bfile
(p_file_loc IN VARCHAR2)
IS
  v_file        BFILE;   v_filename    VARCHAR2(16);
  v_file_exists BOOLEAN;
  CURSOR emp_cursor IS ...
BEGIN
  FOR emp_record IN emp_cursor LOOP
   v_filename := emp_record.first_name || '.bmp';
   v_file := BFILENAME  (p_file_loc, v_filename);
   v_file_exists := (DBMS_LOB.FILEEXISTS(v_file) = 1);
    IF v_file_exists THEN
      DBMS_LOB.FILEOPEN (v_file); ...
```

PLSQL中大对象的操作

DBMS_LOB 主要函数介绍：

1、更改LOB的值： APPEND, COPY, ERASE, TRIM, WRITE, LOADFROMFILE
2、读取、检查LOB的值： GETLENGTH, INSTR, READ, SUBSTR
3、Bfile专用：FILECLOSE, FILECLOSEALL, FILEEXISTS, FILEGETNAME, FILEISOPEN, FILEOPEN

最常用的Read 和Write介绍：

```
PROCEDURE READ (
            lobsrc IN BFILE|BLOB|CLOB ,
            amount IN OUT BINARY_INTEGER,
            offset IN INTEGER,
            buffer OUT RAW|VARCHAR2 )
```

```
PROCEDURE WRITE (
            lobdst IN OUT BLOB|CLOB,
            amount IN OUT BINARY_INTEGER,
            offset IN INTEGER := 1,
            buffer IN RAW|VARCHAR2 )  -- RAW for BLOB
```

PLSQL中大对象的操作

数据库表中LOB列的增删改（下面的例子中resume是CLOB类型，Picture是BLOB类型）：

新增：

```
INSERT INTO employees (employee_id, first_name, last_name, email,
 hire_date, job_id, salary, resume, picture)
VALUES (405, 'Marvin', 'Ellis', 'MELLIS', SYSDATE,  'AD_ASST',
4000, EMPTY_CLOB(),NULL);
```

更新：

```
UPDATE employees
SET resume = 'Date of Birth: 8 February 1951',
    picture = EMPTY_BLOB()
WHERE employee_id = 405;
```

```
UPDATE employees
SET resume = 'Date of Birth: 1 June 1956'
WHERE employee_id = 170;
```

注意： EMPTY_CLOB(), EMPTY_BLOB() 跟NULL是不同的概念，ISNULL 对这两种情况返回FALSE

PLSQL中大对象的操作

```
DECLARE
  lobloc CLOB;            -- serves as the LOB locator
  text   VARCHAR2(32767):='Resigned: 5 August 2000';
  amount NUMBER ;         -- amount to be written
  offset INTEGER;         -- where to start writing
BEGIN
  SELECT resume INTO lobloc
  FROM   employees
  WHERE  employee_id = 405 FOR UPDATE;
  offset := DBMS_LOB.GETLENGTH(lobloc) + 2;
  amount := length(text);
  DBMS_LOB.WRITE (lobloc, amount, offset, text );
  text   := ' Resigned: 30 September 2000';
  SELECT resume INTO lobloc
  FROM   employees
  WHERE  employee_id = 170 FOR UPDATE;
  amount := length(text);
  DBMS_LOB.WRITEAPPEND(lobloc, amount, text);
  COMMIT;
END;
```

PLSQL中大对象的操作

查询：

```
SELECT employee_id, last_name , resume -- CLOB
FROM employees
WHERE employee_id IN (405, 170);
```

EMPLOYEE_ID	LAST_NAME	RESUME
170	Fox	Date of Birth: 1 June 1956 Resigned = 30 September 2000
405	Ellis	Date of Birth: 8 February 1951 Resigned = 5 August 2000

LOB内容截取常用函数：

```
DBMS_LOB.SUBSTR(lob_column, no_of_chars, starting)
DBMS_LOB.INSTR (lob_column, pattern)
```

```
SELECT DBMS_LOB.SUBSTR (resume, 5, 18),
       DBMS_LOB.INSTR (resume,' = ')
FROM   employees
WHERE  employee_id IN (170, 405);
```

DBMS_LOB.SUBSTR(RESUME,5,18)	DBMS_LOB.INSTR(RESUME,'=')
June	36
Febru	40

PLSQL中大对象的操作

删除：

```
DELETE
FROM  employees
WHERE employee_id = 405;
```

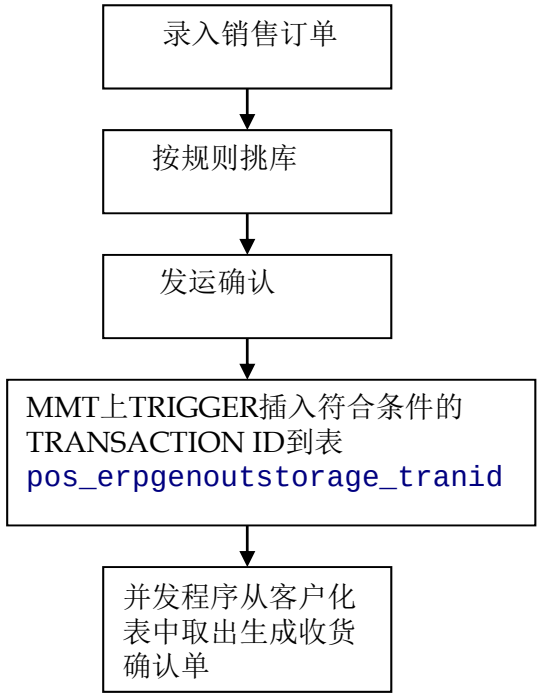
当我们删除一行的时候，该行上的内部LOB对象也被删除。
如果你只是想删除LOB列，而不想删除整行的数据，那么应该使用UPDATE 语句，是LOB列=NULL 或者EMPTY_C/BLOB();

```
UPDATE employees
SET resume = EMPTY_CLOB()
WHERE employee_id = 170;
```

数据库触发器

数据库触发器 **Trigger**的概念：
对数据库对象的操作可引发很多事件，比如**before Insert,before update** 等等，但这些事件产生的时候我们可以写响应代码来完成一些基于事件的操作，通常这些操作被写成一段**Plsql**程序；那么这些更具体的数据库对象上的事件相关的程序呢就称为数据库**Trigger**

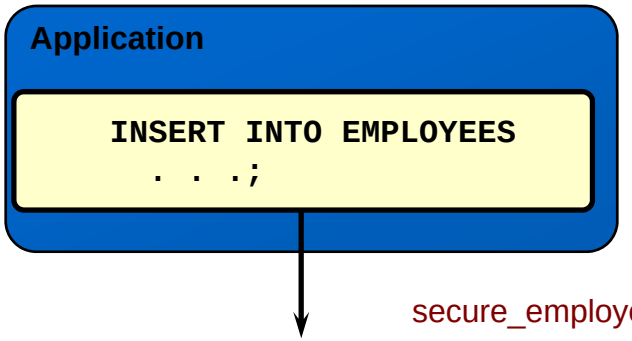
实际应用场景举例：（从**EBS**的发运确认产生的**MMT** 生成 **POS**收货确认单）



注意：除非迫不得已，尽量避免使用**Trigger**,因为这会导致维护困难；

数据库触发器

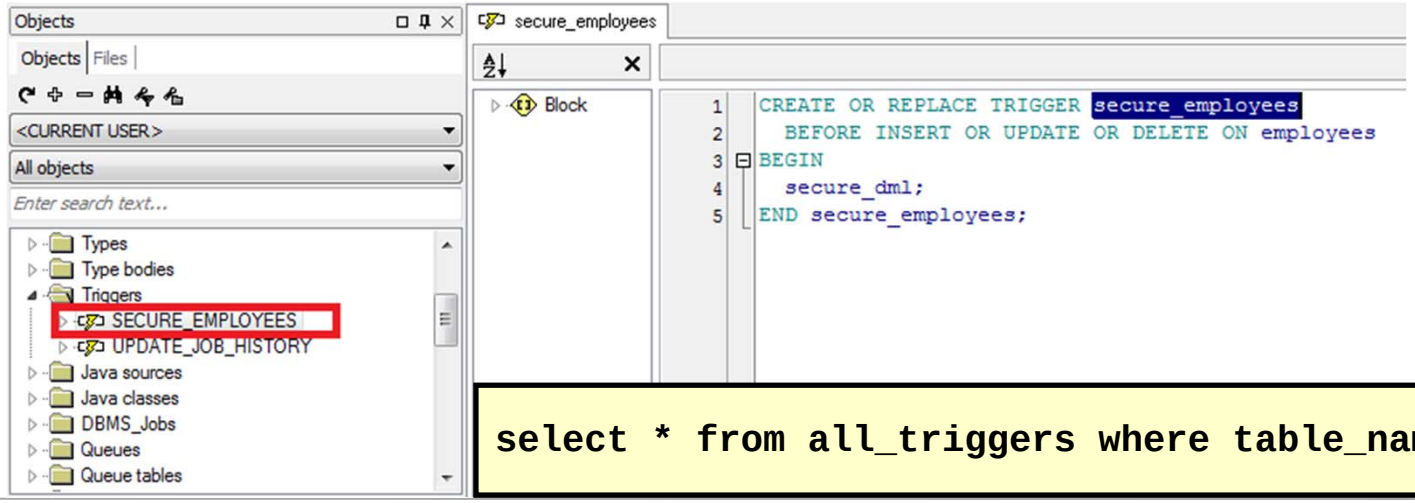
举例：



EMPLOYEES table

secure_employees

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
100	King	AD_PRES	24000
101	Kochhar	AD_VP	17000
102	De Haan	AD_VP	17000
103	Hunold	IT_PROG	9000
...	...	...	...




```
select * from all_triggers where table_name ='EMPLOYEES'
```

数据库触发器

创建Trigger： Trigger的定义语句里面涉及到如下关键因素：

时机：Before 或者 After 或 Instead of
事件：Insert 或 Update 或 Delete
对象：表名（或视图名）
类型：Row 或者 Statement级；
条件：满足特定Where条件才执行；
内容：通常是一段PLSQL块代码；

重点注意：
Instead of ：用Trigger的内容替换 事件本身的动作
Row级：SQL语句影响到的每一行都会引发Trigger
Statement级：一句SQL语句引发一次，不管它影响多少行（甚至0行）

数据库触发器

Trigger的触发顺序：

DML 语句

```
INSERT INTO departments (department_id,
                         department_name, location_id)
VALUES (400, 'CONSULTING', 2400);
```

Triggering 动作

→ BEFORE statement trigger

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
30	Purchasing	1700
...		
400	CONSULTING	2400

→ BEFORE row trigger

→ AFTER row trigger

→ AFTER statement trigger

DML 语句

```
UPDATE employees
    SET salary = salary * 1.1
    WHERE department_id = 30;
```

6 rows updated.

Triggering 动作

→ BEFORE statement trigger

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
114	Raphaely	30
115	Khoo	30
116	Baida	30
117	Tobias	30
118	Himuro	30
119	Colmenares	30

→ BEFORE row trigger

→ AFTER row trigger

...

→ BEFORE row trigger

→ AFTER row trigger

...

→ AFTER statement trigger

数据库触发器

创建Statement 级别Trigger 语法：

```
CREATE [OR REPLACE] TRIGGER trigger_name
   timing
      event1 [OR event2 OR event3]
         ON table_name
trigger_body
```

举例：单一事件Trigger

```
CREATE OR REPLACE TRIGGER secure_emp
 BEFORE INSERT ON employees
 BEGIN
  IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
     (TO_CHAR(SYSDATE,'HH24:MI') NOT BETWEEN '08:00' AND '18:00')
   THEN RAISE_APPLICATION_ERROR (-20500,'You may insert into EMPLOYEES table only
  END IF;
END;
/
```

测试：把数据库服务器日期改成星期天，然后在Insert数据，看有什么效果？

```
INSERT INTO employees (employee_id, last_name, first_name, email, hire_date,
VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE, 'IT_PROG', 4500, 60);
```

数据库触发器

举例：多事件Trigger，在Trigger Body中判断具体事件

```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT OR UPDATE OR DELETE ON employees
BEGIN
 IF (TO_CHAR (SYSDATE,'DY') IN ('SAT','SUN')) OR
    (TO_CHAR (SYSDATE, 'HH24') NOT BETWEEN '08' AND '18')
 THEN
   IF       DELETING THEN
     RAISE_APPLICATION_ERROR (-20502,'You may delete from   EMPLOYEES
                         table only during business hours.');
   ELSIF  INSERTING THEN
     RAISE_APPLICATION_ERROR (-20500,'You may insert into
                EMPLOYEES table only during business hours.');
   ELSIF   UPDATING ('SALARY') THEN
     RAISE_APPLICATION_ERROR (-20503,'You may update
                     SALARY only during business hours.');
   ELSE
     RAISE_APPLICATION_ERROR (-20504,'You may update
              EMPLOYEES table only during normal hours.');
   END IF;
  END IF;
END;
```

数据库触发器

创建Row级别Trigger 语法：

```
CREATE [OR REPLACE] TRIGGER trigger_name
  timing
    event1 [OR event2 OR event3]
      ON table_name
  [REFERENCING OLD AS old | NEW AS new]
FOR EACH ROW
  [WHEN (condition)]
trigger_body
```

举例：

```
CREATE OR REPLACE TRIGGER restrict_salary
  BEFORE  INSERT OR UPDATE OF salary ON employees
    FOR EACH ROW
    BEGIN
      IF  NOT (:NEW.job_id IN ('AD_PRES', 'AD_VP'))
          AND :NEW.salary > 15000
      THEN
        RAISE_APPLICATION_ERROR (-20202,'Employee
     END IF;
END;
/
```

数据库触发器

使用OLD 和 NEW修饰词：

```
CREATE OR REPLACE TRIGGER audit_emp_values
 AFTER DELETE OR INSERT OR UPDATE ON employees
 FOR EACH ROW
BEGIN
  INSERT INTO audit_emp_table (user_name, timestamp,
     id, old_last_name, new_last_name, old_title,
     new_title, old_salary, new_salary)
  VALUES (USER, SYSDATE, :OLD.employee_id,
       :OLD.last_name, :NEW.last_name, :OLD.job_id,
       :NEW.job_id, :OLD.salary, :NEW.salary );
END;
/
```

数据库触发器

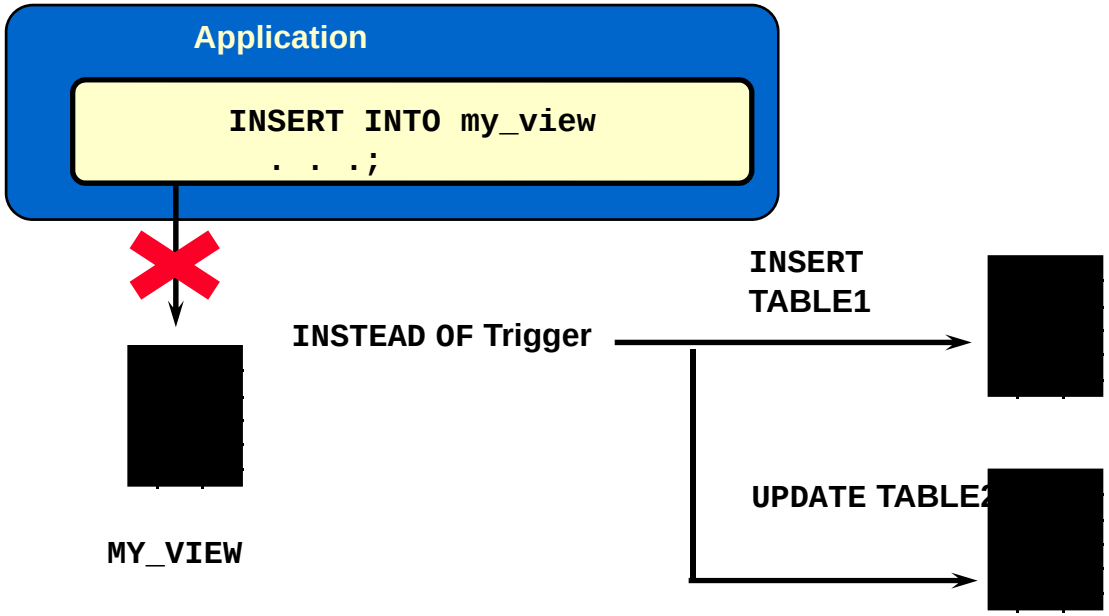
使用WHEN限制 Trigger的触发条件：

```
CREATE OR REPLACE TRIGGER derive_commission_pct
  BEFORE INSERT OR UPDATE OF salary ON employees
  FOR EACH ROW
  WHEN (NEW.job_id = 'SA_REP')
BEGIN
  IF  INSERTING
     THEN :NEW.commission_pct := 0;
  ELSIF :OLD.commission_pct IS NULL
     THEN :NEW.commission_pct := 0;
  ELSE
    :NEW.commission_pct := :OLD.commission_pct + 0.05;
  END IF;
END;
/
```

注意：这个例子中有对：NEW的赋值动作，大家想一想，如果是After类型的Trigger, 还允许对:NEW赋值吗？

数据库触发器

INSTEAD OF Trigger的运行原理图解：



INSTEAD OF Trigger的使用场景：建立在View上的Trigger 通常是INSTEAD OF类型的，因为复杂视图不能被直接更改； 意义不大，很少使用；

数据库触发器

管理Trigger：

失效/生效：

```
ALTER TRIGGER trigger_name  DISABLE | ENABLE
```

批量失效/生效：

```
ALTER TABLE table_name   DISABLE | ENABLE  ALL TRIGGERS
```

重新编译：

```
ALTER TRIGGER trigger_name COMPILE
```

删除：

```
DROP TRIGGER trigger_name;
```

数据库触发器

但约束与Trigger同时存在，那个先执行？

```
UPDATE employees SET department_id = 999
 WHERE employee_id = 170;
-- Integrity constraint violation error
```

```
CREATE OR REPLACE TRIGGER constr_emp_trig
 AFTER UPDATE ON employees
  FOR EACH ROW
BEGIN
  INSERT INTO departments
    VALUES (999, 'dept999', 140, 2400);
END;
/
```

```
UPDATE employees SET department_id = 999
 WHERE employee_id = 170;
-- Successful after trigger is fired
```

数据库触发器

除了针对Table,View的Trigger, 还有针对Database的 Trigger，比如 ： 写Triger记录所有登录/登出数据库的用户

```
CREATE OR REPLACE TRIGGER logon_trig
AFTER LOGON  ON  SCHEMA
BEGIN
 INSERT INTO log_trig_table(user_id, log_date, action)
 VALUES (USER, SYSDATE, 'Logging on');
END;
/
```

```
CREATE OR REPLACE TRIGGER logoff_trig
BEFORE LOGOFF  ON  SCHEMA
BEGIN
 INSERT INTO log_trig_table(user_id, log_date, action)
 VALUES (USER, SYSDATE, 'Logging off');
END;
/
```

数据库触发器

冲突表（Mutating Table) 概念：

当某张表上的针对DML动作的Trigger需要访问到表自身的数据时，对Trigger来说，这就是一张冲突表，看例子：

```
CREATE OR REPLACE TRIGGER check_salary
  BEFORE INSERT OR UPDATE OF salary, job_id
  ON employees
  FOR EACH ROW
  WHEN (NEW.job_id <> 'AD_PRES')
DECLARE
  v_minsalary employees.salary%TYPE;
  v_maxsalary employees.salary%TYPE;
BEGIN
  SELECT MIN(salary), MAX(salary)
   INTO    v_minsalary, v_maxsalary
   FROM    employees
   WHERE job_id = :NEW.job_id;
  IF :NEW.salary < v_minsalary OR
     :NEW.salary > v_maxsalary THEN
     RAISE_APPLICATION_ERROR(-20505,
'Out of range');
  END IF;
END;
/
```

```
UPDATE employees
 SET salary = 3400
 WHERE last_name = 'Stiles';
```

```
UPDATE employees
    *
ERROR at line 1:
ORA-04091: table PLSQL.EMPLOYEES is mutating, trigger/function may not see it
ORA-06512: at "PLSQL.CHECK_SALARY", line 5
ORA-04088: error during execution of trigger 'PLSQL.CHECK_SALARY'
```

对于employee表的新增或者更改Salary动作引发的Trigger，该Trigger又要读取Salary的Min值，这就冲突了。

一般我们要避免写这种Trigger, 若真遇到迫不得已的时候也可参考百度文库中一篇文章的做法：

http://wenku.baidu.com/view/a6d40c2bbd64783e09122b7d.html

数据库触发器

将Trigger用于数据变更审计日志：

Oracle内置数据更改审计的功能： 即更改动作可以被记录到审计记录表 sys.Audit$.
要让审计功能其作用，必须：
1、设置数据库参数：audit_trail = DB 或者 OS （默认是NONE)
2、设置要审计的对象：比如

```
AUDIT INSERT, UPDATE, DELETE
  ON  stu3693.copy_emp
  BY ACCESS
WHENEVER SUCCESSFUL;
```

通常情况下，客户方的DBA从性能角度考虑，不愿意启用数据库的审计功能，但某些客户认为某张表的变更记录非常重要，一定要有审计功能该怎么办呢？ 可考虑使用Trigger作为替代方案，比如：

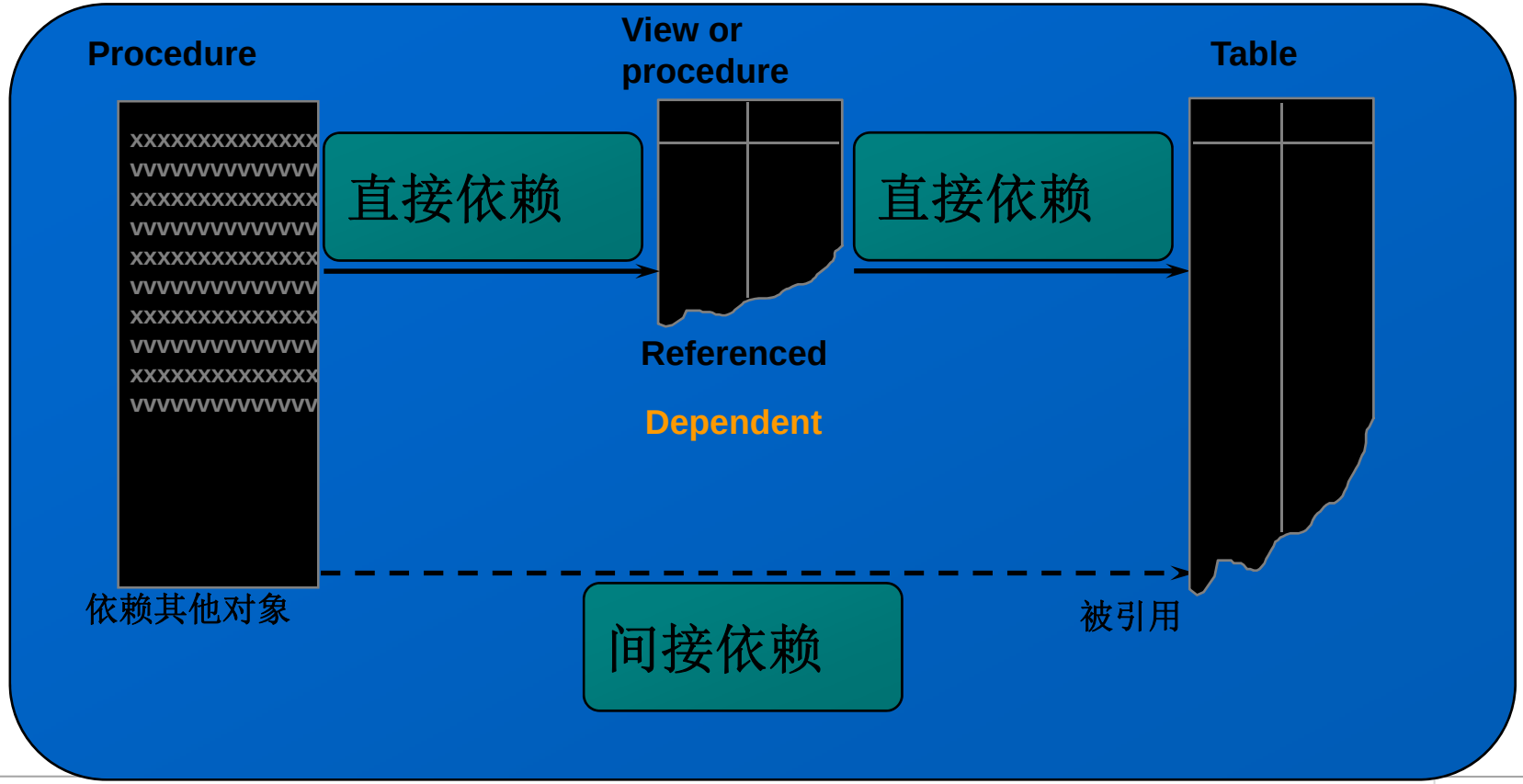
```
CREATE OR REPLACE TRIGGER audit_emp_values
 AFTER DELETE OR INSERT OR UPDATE ON employees
 FOR EACH ROW
BEGIN
 IF (audit_emp_package.g_reason IS NULL) THEN
    RAISE_APPLICATION_ERROR (-20059, 'Specify a reason
     for the data operation through the procedure SET_REASON
     of the AUDIT_EMP_PACKAGE before proceeding.');
 ELSE
    INSERT INTO audit_emp_table (user_name, timestamp, id,
      old_last_name, new_last_name, old_title, new_title,
      old_salary, new_salary, comments)
    VALUES (USER, SYSDATE, :OLD.employee_id, :OLD.last_name,
      :NEW.last_name, :OLD.job_id, :NEW.job_id, :OLD.salary,
      :NEW.salary, audit_emp_package.g_reason);
 END IF;
END;
```

数据库对象的依赖关系

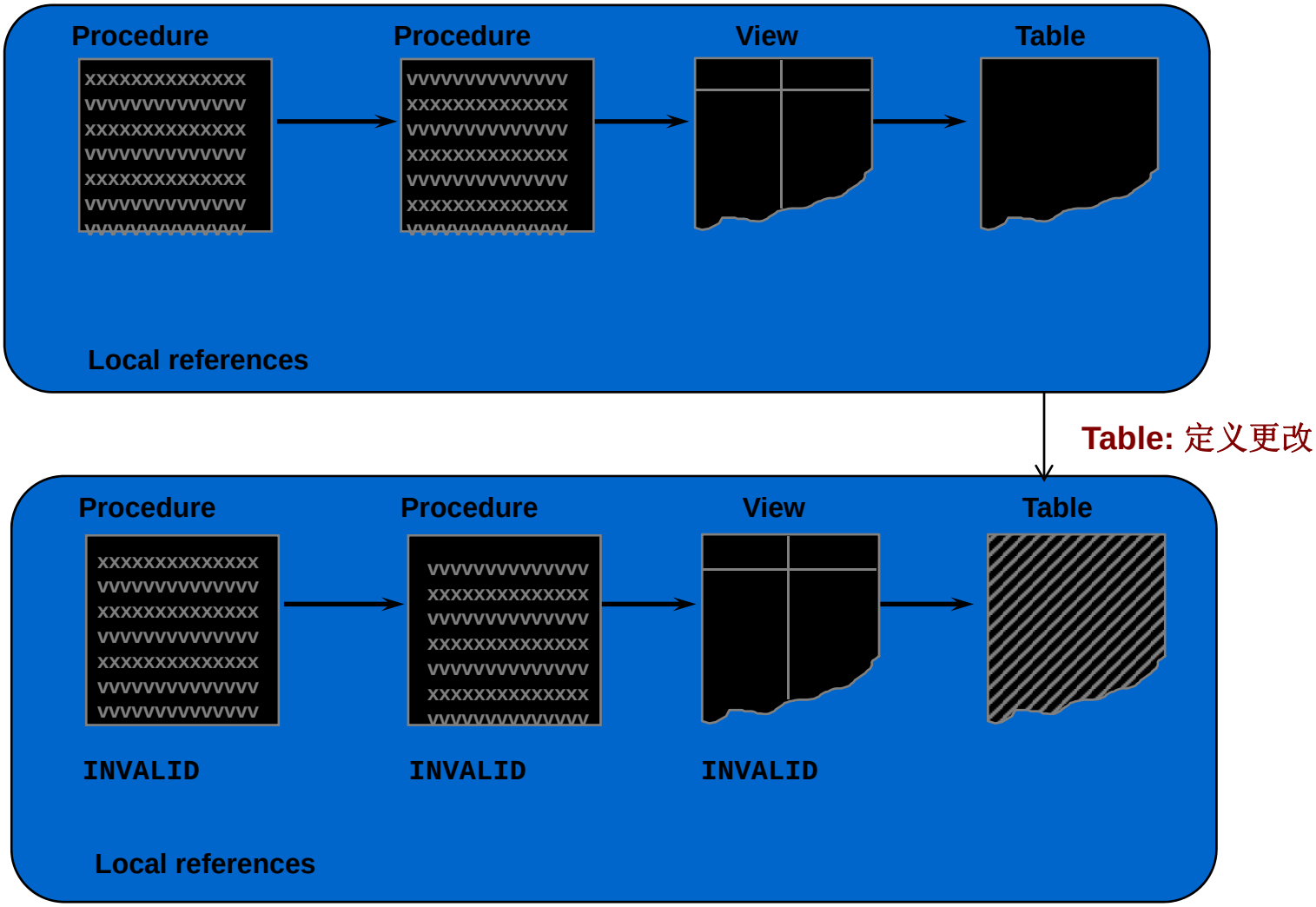
概念：

数据库里面有些对象的定义是依赖于其他对象的，比如View 依赖于Table , Procedure 中有Select 其他Table或者视图的语句，可能依赖于Table 或者视图； 在这里View 或者 Procedure就被称为依赖对象，而table则被称为引用对象。

当引用对象发生定义变更的时候，会导致其依赖对象失效，需要重新编译。



数据库对象的依赖关系



这个例子中，只要被引用对象定义的变更不会导致依赖对象编译出错，那么，当**Procedure** 被调用的时候，会报告对象无效，然后自动进行重新编译，再次**Retry**的时候就**OK**了。

Q & A

Questions & AnswerS