[Oracle 异常处理](http://blog.csdn.net/narutobing/article/details/7994310)

控制PL/SQL错误 EXCEPTION，SQLCODE，SQLERRM --浏览时可通过查找功能跳跃式的寻找自己要查看的内容！

一、错误控制一览

在PL/SQL中，警告或错误被称为异常。异常可以是内部(运行时系统)定义的或是用户定义的。内部定义的案例包括除零操作和内存溢出等。一些常见的内部异常都有一个预定义的名字，如ZERO\_DIVIDE和STORAGE\_ERROR等。对于其它的内部异常，我们可以手动为它们命名。

我们可以在PL/SQL块、子程序或包的声明部分自定义异常。例如，我们可以定义一个名为insufficient\_funds的异常来标示帐户透支的情况。与内部异常不同的是，用户自定义异常必须有一个名字。

错误发生时，异常就会被抛出。也就是说，正常的执行语句会被终止，控制权被转到PL/SQL块的异常控制部分或子程序的异常控制部分。内部异常会由运行时系统隐式地抛出，而用户定义异常必须显式地用RAISE语句抛出，RAISE语句也可以抛出预定义异常。

为了控制被抛出的异常，我们需要单独编写被称为"exception handler"的异常控制程序。异常控制程序运行后，当前块就会停止执行，封闭块继续执行下一条语句。如果没有封闭块，控制权会直接交给主环境。

下例中，我们为一家股票[代码](http://blog.csdn.net/narutobing/article/details/7994310)(Ticker Symbol)为XYZ的公司计算并保存市盈率(price-to-earning)。如果公司的收入为零，预定义异常ZERO\_DIVIDE就会被抛出。这将导致正常的执行被终止，控制权被交给异常控制程序。可选的OTHERS处理器可以捕获所有的未命名异常。

DECLARE  
   pe\_ratio   NUMBER (3, 1);  
BEGIN  
SELECT price / earnings  
    INTO pe\_ratio  
    FROM stocks  
   WHERE symbol = 'XYZ';   -- might cause division-by-zero error  
INSERT INTO stats (symbol, ratio)  
       VALUES ('XYZ', pe\_ratio);  
COMMIT;  
EXCEPTION   -- exception handlers begin  
WHEN ZERO\_DIVIDE THEN   -- handles 'division by zero' error  
    INSERT INTO stats (symbol, ratio)  
         VALUES ('XYZ', NULL);  
    COMMIT;  
     ...  
WHEN OTHERS THEN   -- handles all other errors  
    ROLLBACK;  
END;   -- exception handlers and block end here

上面的例子演示了异常控制，但对于INSERT语句的使用就有些低效了。使用下面的语句就要好一些：

INSERT INTO stats (symbol, ratio)  
SELECT symbol,DECODE(earnings, 0, NULL, price / earnings)  
    FROM stocks  
   WHERE symbol = 'XYZ';

在下面这个例子中，子查询为INSERT语句提供了数据。如果earnings是零的话，函数DECODE就会返回空，否则DECODE就会返回price与earnings的比值。

二、异常的优点

使用异常来控制错误有几个优点。如果没有异常控制的话，每次执行一条语句，我们都必须进行错误检查：

BEGIN  
SELECT ...  
    -- check for ’no data found’ error  
SELECT ...  
    -- check for ’no data found’ error  
SELECT ...  
    -- check for ’no data found’ error

错误处理和正常的处理内容界限不明显，导致代码混乱。如果我们不编写错误检查代码，一个错误就可能引起其它错误，有时还可能是一些无关错误。

但有了异常后，我们就能很方便的控制错误，而且不需要编写多个检查代码：

BEGIN  
SELECT ...  
SELECT ...  
SELECT ...  
   ...  
EXCEPTION  
WHEN NO\_DATA\_FOUND THEN -- catches all 'no data found' errors

异常能把错误控制程序单独分离出来，改善可读性，主要的算法不会受到错误恢复算法影响。异常还可以提高可靠性。我们不需要在每一个可能出现错误的地方编写错误检查代码了，只要在PL/SQL块中添加一个异常控制代码即可。这样，如果有异常被抛出，我们就可以确保它能够被捕获并处理。

三、预定义PL/SQL异常

当我们的PL/SQL程序与Oracle规则相冲突或超过系统相关(system-dependent)的限制时，内部异常就会被抛出。每个Oracle错误都有一个错误编号，但异常只能按名称捕获，然后被处理。所以，PL/SQL把一些常见Oracle错误定义为异常。例如，如果SELECT INTO语句查询不到数据时，PL/SQL就会抛出预定义异常NO\_DATA\_FOUND。

要控制其它Oracle异常，我们可以使用OTHERS处理器。函数SQLCODE和SQLERRM在OTHERS处理器中特别有用，因为它们能返回Oracle错误编号和消息。另外，我们还可以使用编译指示(pragma)EXCEPTION\_INIT把一个异常名称和一个Oracle错误编号关联起来。PL/SQL在STANDARD包中声明了全局预定义异常。所以，我们不需要自己声明它们。我们可以为下面列表中命名的预定义异常编写处理程序：

异常 Oracle错误号SQLCODE值  
ACCESS\_INTO\_NULL ORA-06530 -6530  
CASE\_NOT\_FOUND ORA-06592 -6592  
COLLECTION\_IS\_NULL ORA-06531 -6531  
CURSOR\_ALREADY\_OPEN ORA-06511 -6511  
DUP\_VAL\_ON\_INDEX ORA-00001 -1  
INVALID\_CURSOR ORA-01001 -1001  
INVALID\_NUMBERORA-01722 -1722  
LOGIN\_DENIED ORA-01017 -1017  
NO\_DATA\_FOUND ORA-01403 100  
NOT\_LOGGED\_ON ORA-01012 -1012  
PROGRAM\_ERROR ORA-06501 -6501  
ROWTYPE\_MISMATCH ORA-06504 -6504  
SELF\_IS\_NULL ORA-30625 -30625  
STORAGE\_ERROR ORA-06500 -6500  
SUBSCRIPT\_BEYOND\_COUNT ORA-06533 -6533  
SUBSCRIPT\_OUTSIDE\_LIMIT ORA-06532 -6532  
SYS\_INVALID\_ROWID ORA-01410 -1410  
TIMEOUT\_ON\_RESOURCE ORA-00051 -51  
TOO\_MANY\_ROWS ORA-01422 -1422  
VALUE\_ERROR ORA-06502 -6502  
ZERO\_DIVIDE ORA-01476 -1476

预定义异常的简要描述：

异常 抛出时机  
ACCESS\_INTO\_NULL程序尝试为一个未初始化(自动赋为null)对象的属性赋值。  
CASE\_NOT\_FOUNDCASE语句中没有任何WHEN子句满足条件，并且没有编写ELSE子句。  
COLLECTION\_IS\_NULL程序尝试调用一个未初始化(自动赋为null)嵌套表或变长数组的集合方法(不包括EXISTS)，或者是程序尝试为一个未初始化嵌套表或变长数组的元素赋值。  
CURSOR\_ALREADY\_OPEN程序尝试打开一个已经打开的游标。一个游标在重新打开之前必须关闭。一个游标FOR循环会自动打开它所引用的游标。所以，我们的程序不能在循环内部打开游标。  
DUP\_VAL\_ON\_INDEX程序尝试向一个有着唯一约束条件的[数据库](http://blog.csdn.net/narutobing/article/details/7994310)字段中保存重复值。  
INVALID\_CURSOR程序尝试操作一个不合法的游标，例如关闭一个未打开的游标。  
INVALID\_NUMBER在一个[SQL](http://blog.csdn.net/narutobing/article/details/7994310)语句中，由于字符串并不代表一个有效的数字，导致字符串向数字转换时会发生错误。(在过程化语句中，会抛出异常VALUE\_ERROR。)当FETCH语句的LIMIT子句表达式后面不是一个正数时，这个异常也会被抛出。  
LOGIN\_DENIED程序尝试使用无效的用户名和/或密码来登录Oracle。  
NO\_DATA\_FOUNDSELECT INTO语句没有返回数据，或者是我们的程序引用了一个嵌套表中被删除了的元素或是索引表中未初始化的元素。SQL聚合函数，如AVG和SUM，总是能返回一个值或空。所以，一个调用聚合函数的SELECT INTO语句从来不会抛出NO\_DATA\_FOUND异常。FETCH语句最终会取不到数据，当这种情况发生时，不会有异常抛出的。  
NOT\_LOGGED\_ON 程序没有连接到Oracle就要调用数据库。  
PROGRAM\_ERROR PL/SQL程序发生内部错误。  
ROWTYPE\_MISMATCH赋值语句中使用的主游标变量和PL/SQL游标变量的类型不兼容。例如，当一个打开的主游标变量传递到一个存储子程序时，实参的返回类型和形参的必须一致。  
SELF\_IS\_NULL程序尝试调用一个空实例的MEMBER方法。也就是内置参数SELF(它总是第一个传递到MEMBER方法的参数)是空。  
STORAGE\_ERRORPL/SQL运行时内存溢出或内存不足。  
SUBSCRIPT\_BEYOND\_COUNT程序引用一个嵌套表或变长数组元素，但使用的下标索引超过嵌套表或变长数组元素总个数。  
SUBSCRIPT\_OUTSIDE\_LIMIT程序引用一个嵌套表或变长数组，但使用的下标索引不在合法的范围内(如-1)。  
SYS\_INVALID\_ROWID从字符串向ROWID转换发生错误，因为字符串并不代表一个有效的ROWID。  
TIMEOUT\_ON\_RESOURCE当Oracle等待资源时，发生超时现象。  
TOO\_MANY\_ROWSSELECT INTO语句返回多行数据。  
VALUE\_ERROR发生算术、转换、截位或长度约束错误。例如，当我们的程序把一个字段的值放到一个字符变量中时，如果值的长度大于变量的长度，PL/SQL就会终止赋值操作并抛出异常VALUE\_ERROR。在过程化语句中，如果字符串向数字转换失败，异常VALUE\_ERROR就会被抛出。(在SQL语句中，异常INVALID\_NUMBER会被抛出。)  
ZERO\_DIVIDE 程序尝试除以0。

四、自定义PL/SQL异常

PL/SQL允许我们定义自己的异常。与预定义异常不同的是，用户自定义异常必须声明，并且需要用RAISE语句显式地抛出。

1、声明PL/SQL异常

异常只能在PL/SQL块、子程序或包的声明部分声明。下例中，我们声明一个名为past\_due的异常：

DECLARE  
   past\_due EXCEPTION;

异常和变量的声明是相似的。但是要记住，异常是一种错误情况(error condition)，而不是数据项。与变量不同的是，异常不能出现在赋值语句或是SQL语句中。但是，变量的作用域规则也适用于异常。

2、PL/SQL异常的作用域规则

在同一个块内，异常不能声明两次。但可以在不同的块声明相同的异常。

块中声明的异常对于当前块来说是本地的，但对于当前块的所有子块来说是全局的。因为块只能引用本地或全局的异常，所以封闭块不能引用声明在子块中的异常。

如果我们在子块中重新声明了一个全局的异常，本地声明的异常的优先级是要高于全局的。所以，子块就不能引用全局的异常，除非全局异常在它的所在块中用标签作了标记，这种情况下可以使用下面的语法来引用全局异常：

block\_label.exception\_name

下例中演示了作用范围规则：

DECLARE  
   past\_due   EXCEPTION;  
   acct\_num   NUMBER;  
BEGIN  
DECLARE   -- sub-block begins  
    past\_due   EXCEPTION;   -- this declaration prevails  
     acct\_num   NUMBER;  
BEGIN  
     ...  
    IF ... THEN  
     RAISE past\_due;   -- this is not handled  
    END IF;  
END;   -- sub-block ends  
EXCEPTION  
WHEN past\_due THEN   -- does not handle RAISEd exception  
     ...  
END;

上例中的封闭块并不能捕获抛出来的异常，因为在子块中声明的past\_due优先级要高于封闭块声明的异常。虽然它们的名字相同，但实际上是两个不同的past\_due异常，就像两个acct\_num变量只是共享着相同的名字一样，实际上它们是完全不同的两个变量。因此，RAISE语句和WHEN子句所引用的是不同的异常。如果想让封闭块能捕获到子块中的past\_due异常，我们就必须从子块中删除声明，或是在封闭块中添加OTHERS处理器。

3、把PL/SQL异常与编号关联：编译指示EXCEPTION\_INIT

要想控制没有预定义名称的错误(通常为 ORA- 消息)，我们就必须使用OTHERS处理器或编译指示EXCEPTION\_INIT。编译指示就是能在编译期而非运行时进行处理的编译指令。

在PL/SQL中，编译指示EXCPTION\_INIT能告诉编译器把异常名称和错误编号关联起来。这就能让我们按名称来引用所有的内部异常，并为它编写特定的处理程序。在我们看到的错误栈或是错误消息序列中，最顶层的就是我们能捕获和处理的信息。

我们可以把编译指示EXCEPTION\_INIT写在PL/SQL块、子程序或包的声明部分，语法如下：

PRAGMA EXCEPTION\_INIT(exception\_name, -Oracle\_error\_number);

其中exception\_name是已经声明过的异常名称，Oracle\_error\_number是Oracle错误编号。编译指示必须和异常声明处于同一个声明中，并且只能在异常声明之后出现。如下例所示：

DECLARE  
  deadlock\_detected   EXCEPTION;  
PRAGMA EXCEPTION\_INIT (deadlock\_detected, -60);  
BEGIN  
   ...   -- Some operation that causes an ORA-00060 error  
EXCEPTION  
WHEN deadlock\_detected THEN  
    -- handle the error  
     ...  
END;

4、自定我们自己的错误消息：过程RAISE\_APPLICATION\_ERROR

过程RAISE\_APPLICATION\_ERROR能帮助我们从存储子程序中抛出用户自定义的错误消息。这样，我们就能把错误消息报告给应用程序而避免返回未捕获异常。

调用RAISE\_APPLICATION\_ERROR的语法如下：

raise\_application\_error(error\_number, message[, {TRUE | FALSE}]);

error\_number是一个范围在-20000至-20999之间的负整数，message是最大长度为2048字节的字符串。如果第三个可选参数为TRUE的话，错误就会被放到前面错误的栈顶。如果为FALSE(默认值)，错误就会替代前面所有的错误。

RAISE\_APPLICATION\_ERROR是包DBMS\_STANDARD的一部分，所以，我们对它的引用不需要添加限定修饰词。

应用程序只能从一个正在执行的存储子程序或方法中调用raise\_application\_error。在调用时，raise\_application\_error会结束子程序并把用户定义的错误编号和消息返回给应用程序。错误编号和消息可以像其它的Oracle错误一样被捕获。

在下面的例子中，我们在雇员工资栏的内容为空的情况下调用raise\_application\_error：

CREATE PROCEDURE raise\_salary (emp\_id NUMBER, amount NUMBER) AS  
   curr\_sal   NUMBER;  
BEGIN  
SELECT sal  
    INTO curr\_sal  
    FROM emp  
   WHERE empno = emp\_id;  
IF curr\_sal IS NULL THEN  
     /\* Issue user-defined error message. \*/  
     raise\_application\_error (-20101, 'Salary is missing');  
ELSE  
    UPDATE emp  
       SET sal = curr\_sal + amount  
     WHERE empno = emp\_id;  
END IF;  
END raise\_salary;

调用程序会得到一个PL/SQL异常，它能在OTHERS处理器中使用错误报告函数SQLCODE和SQLERRM来进行处理。同样，我们也可以使用编译指示EXCEPTION\_INIT把raise\_application\_error返回的错误编号映射到异常本身。如下面的Pro\*C例子所示：

EXEC SQL EXECUTE  
   /\* Execute embedded PL/SQL block using host  
   variables my\_emp\_id and my\_amount, which were  
   assigned values in the host environment. \*/

DECLARE  
   null\_salary   EXCEPTION;  
   /\* Map error number returned by raise\_application\_error  
   to user-defined exception. \*/  
PRAGMA EXCEPTION\_INIT (null\_salary, -20101);  
BEGIN  
   raise\_salary (:my\_emp\_id, :my\_amount);  
EXCEPTION  
WHEN null\_salary THEN  
    INSERT INTO emp\_audit  
         VALUES (:my\_emp\_id, ...);  
END;

END-EXEC;

这项[技术](http://blog.csdn.net/narutobing/article/details/7994310)能让调用程序在特定的异常处理程序中控制错误。

5、重新声明预定义异常

请记住，PL/SQL把预定义的异常作为全局内容声明在包STANDARD中，所以，我们没有必要重新声明它们。重新声明预定义异常是错误的做法，因为我们的本地声明会覆盖掉全局声明。例如，如果我们声明了一个invalid\_number，当PL/SQL抛出预定义异常INVALID\_NUMBER时，我们为异常INVALID\_NUMBER编写的异常控制程序就无法正确地捕获到它了。这种情况下，我们必须像下面这样使用点标志来指定预定义异常：

EXCEPTION  
WHENINVALID\_NUMBER OR STANDARD.INVALID\_NUMBER THEN  
    -- handle the error  
END;

五、如何抛出PL/SQL异常

内部异常会由运行时系统隐式地抛出，其中也包括使用编译指示EXCEPTION\_INIT与Oracle错误编号关联起来的用户自定义异常。但是，用户自定义的异常就必须显式地用RAISE语句抛出。

1、使用RAISE语句抛出异常

PL/SQL块和子程序应该只在错误发生或无法完成正常程序处理的时候才抛出异常。下例中，我们用RAISE语句抛出一个用户自定义的out\_of\_stack异常：

DECLARE  
   out\_of\_stock     EXCEPTION;  
   number\_on\_hand   NUMBER (4);  
BEGIN  
   ...  
IF number\_on\_hand < 1 THEN  
    RAISE out\_of\_stock;  
END IF;  
EXCEPTION  
WHEN out\_of\_stock THEN  
    -- handle the error  
END;

我们也可以显式地抛出预定义异常。这样，为预定义异常编写的处理程序也就能够处理其它错误了，示例如下：

DECLARE  
   acct\_type   INTEGER := 7;  
BEGIN  
IF acct\_type NOT IN (1, 2, 3) THEN  
    RAISE INVALID\_NUMBER;   -- raise predefined exception  
END IF;  
EXCEPTION  
WHEN INVALID\_NUMBER THEN  
    ROLLBACK;  
END;  
六、PL/SQL异常的传递

异常被抛出时，如果PL/SQL在当前块或子程序中没有找到对应的异常控制程序，异常就会被继续向上一级传递。也就是说异常会把它自身传递到后继的封闭块直到找到异常处理程序或是再也没有可以搜索到的块为止。在后一种情况下，PL/SQL会向主环境抛出一个未捕获异常。

但是，异常是不能通过远程过程调用(RPC)来传递的。因此，PL/SQL块不能捕获由远程子程序抛出的异常。  
异常可以跨作用域传递，也就是说，它能够超越声明它的块的范围而存在。如下例所示：

BEGIN  
   ...  
DECLARE   -- sub-block begins  
     past\_due   EXCEPTION;  
BEGIN  
     ...  
    IF ... THEN  
      RAISE past\_due;  
    END IF;  
END;   -- sub-block ends  
EXCEPTION  
   ...  
WHEN OTHERS THEN  
    ROLLBACK;  
END;

因为异常past\_due所在的块并没有专门针对它的处理程序，所以异常就被传递到封闭块。但是，按照作用域规则，封闭块是不能引用子块声明的异常。所以，只有OTHERS处理器才能捕获到这个异常。如果没有用户定义异常的处理程序，调用这个程序就会得到下面的错误：

ORA-06510: PL/SQL: unhandled user-defined exception

七、重新抛出PL/SQL异常

有时我们需要重新抛出捕获到异常，也就是说，我们想在本地处理之后再把它传递到封闭块。比如，在异常发生的时候，我们可能需要回滚事务，然后在封闭块中写下错误日志。

要重新抛出异常，只要在本地处理程序中放置一个RAISE语句即可，示例如下：

DECLARE  
   out\_of\_balance   EXCEPTION;  
BEGIN  
   ...  
BEGIN   -- sub-block begins  
     ...  
    IF ... THEN  
      RAISE out\_of\_balance;   -- raise the exception  
    END IF;  
EXCEPTION  
    WHEN out\_of\_balance THEN  
      -- handle the error  
      RAISE;   -- reraise the current exception  
END;   -- sub-block ends  
EXCEPTION  
WHEN out\_of\_balance THEN  
    -- handle the error differently  
     ...  
END;

如果在RAISE语句中省略了异常名称——只允许在异常处理程序中这样做——程序就会把当前的异常重新抛出。

八、处理PL/SQL异常

异常抛出时，PL/SQL块或子程序的正常执行就会停止，控制权转到块或子程序的异常处理部分，语法如下：

EXCEPTION  
WHEN exception\_name1 THEN   -- handler  
     sequence\_of\_statements1  
WHEN exception\_name2 THEN   -- another handler  
     sequence\_of\_statements2  
     ...  
WHEN OTHERS THEN   -- optional handler  
     sequence\_of\_statements3  
END;

为捕获抛出的异常，我们需要编写异常处理程序。每个处理程序都由一个WHEN子句和语句序列组成。这些语句执行完毕后，块或子程序就会结束，控制权不再返回异常被抛起的地方。换句话说，也就是我们不能再次返回异常发生的地方继续执行我们的程序。

可选的OTHERS处理器总是块或子程序的最后一个处理程序，它可以用于捕获所有的未命名异常。因此，块或子程序只能有一个OTHERS处理器。如下例所示，OTHERS处理器能够保证所有的异常都会被控制：

EXCEPTION  
WHEN ... THEN  
    -- handle the error  
WHEN ... THEN  
    -- handle the error  
WHEN OTHERS THEN  
    -- handle all other errors  
END;

如果我们想让两个或更多的异常执行同样的语句序列，只需把异常名称用关键字OR隔开，放在同一个WHEN子句中即可，如下例所示：

EXCEPTION  
WHEN over\_limit OR under\_limit OR VALUE\_ERROR THEN  
-- handle the error

只要在WHEN子句的异常列表中有一项与被抛出异常相匹配，相关的语句序列就会被执行。关键字OTHERS不能出现在异常名称列表中；它只能单独使用。我们可以有任意数量的异常处理程序，而且每个处理程序都与一个异常列表及其对应的语句序列相关联。但是，异常名称只能在块或子程序的异常处理部分出现一次。

变量作用范围的规则在这里也同样适用，所以我们可以在异常处理程序中引用本地或全局变量。但是，当游标FOR循环中有异常抛出时，游标就会在异常处理程序调用之前被隐式地关闭。因此，显式游标的属性值在异常处理程序中就不再可用了。

1、声明中控制异常

如果在声明时使用了错误的初始化表达式也有可能引发异常。例如，下面的声明就是因常量credit\_limit不能存储超过999的数字而抛出了异常：

DECLARE  
   credit\_limit CONSTANT NUMBER(3) := 5000;   -- raises an exception  
BEGIN  
   ...  
EXCEPTION  
WHEN OTHERS THEN   -- cannot catch the exception  
   ...  
END;

当前块中的处理程序并不能捕获到抛出的异常，这是因为声明时抛出的异常会被立即传递到最近的封闭块中去。

2、异常句柄中控制异常

在一个块或子程序中，一次只能有一个异常被激活。所以，一个被异常处理程序抛出的异常会被立即传递到封闭块，在那儿，封闭块会为它查找新的处理程序。从那一刻起，异常传递才开始正常化。参考下面的例子：

EXCEPTION  
WHEN INVALID\_NUMBER THEN  
    INSERT INTO ...   -- might raise DUP\_VAL\_ON\_INDEX  
WHEN DUP\_VAL\_ON\_INDEX THEN ...   -- cannot catch the exception  
END;

3、异常分支

GOTO语句不能跳转到异常控制程序。同样，GOTO语句也不能从异常控制程序跳转到当前块。例如，下面的GOTO语句就是非法的：

DECLARE  
   pe\_ratio   NUMBER (3, 1);  
BEGIN  
DELETE FROM stats  
        WHERE symbol = 'xyz';  
SELECT price / NVL (earnings, 0)  
    INTO pe\_ratio  
    FROM stocks  
   WHERE symbol = 'xyz';

   <<my\_label>>  
INSERT INTO stats (symbol, ratio)  
       VALUES ('xyz', pe\_ratio);  
EXCEPTION  
WHEN ZERO\_DIVIDE THEN  
     pe\_ratio   := 0;  
    GOTO my\_label;   -- illegal branch into current block  
END;

但是，GOTO语句可以从一个异常控制程序中跳转到一个封闭块。  
4、获取错误代号与消息：SQLCODE和SQLERRM

在异常处理程序中，我们可以使用内置函数SQLCODE和SQLERRM来查出到底发生了什么错误，并能够获取相关的错误信息。对于内部异常来说，SQLCODE会返回Oracle错误编号。SQLCODE返回的总是一个负数，除非发生的Oracle错误是没有找到数据，这时返回的是+100。SQLERRM会返回对应的错误消息。消息是以Oracle错误编号开头的。

如果我们没有使用编译指令EXCEPTION\_INIT把异常与编号关联的话，SQLCODE和SQLERRM就会分别返回+1和消息"User-Defined Exception"。Oracle错误消息最大长度是512个字符，其中包括错误编号、嵌套消息和具体表和字段的名称。

如果没有异常抛出，SQLCODE返回0，SQLERRM返回消息"ORA-0000: normal, successful completion"。

我们可以把错误编号传递给SQLERRM，让它返回对应的错误消息。但是，一定要保证我们传递给SQLERRM的错误编号是负数。下例中，我们把一个正数传递给SQLERRM，结果就不是我们想要的那样的了：

DECLARE  
  err\_msg   VARCHAR2(100);  
BEGIN  
/\* Get all Oracle error messages. \*/  
FOR err\_num IN 1 .. 9999 LOOP  
     err\_msg     := SQLERRM(err\_num);   -- wrong; should be -err\_num

    INSERT INTO ERRORS  
         VALUES (err\_msg);  
END LOOP;  
END;

把正数传给SQLERRM时，如果传递的是+100，返回的结果是"no data found"，其他情况总是会返回消息"user-defined exception"。把0传递给SQLERRM，就会返回消息"normal, successful completion"。

我们不能直接在SQL语句中使用SQLCODE或SQLERRM。我们必须先把它们的值赋给本地变量，然后再在SQL中使用变量，如下例所示：

DECLARE  
   err\_num   NUMBER;  
   err\_msg   VARCHAR2(100);  
BEGIN  
   ...  
EXCEPTION  
WHEN OTHERS THEN  
     err\_num     := SQLCODE;  
     err\_msg     := SUBSTR(SQLERRM, 1, 100);

    INSERT INTO ERRORS  
         VALUES (err\_num, err\_msg);  
END;

字符串函数SUBSTR可以保证用SQLERRM为err\_msg赋值时不会引起VALUE\_ERROR异常。函数SQLCODE和SQLERRM在OTHERS异常处理程序中特别有用，因为它们能让我们知道哪个内部异常被抛出。

注意：在使用编译指示RESTRICT\_REFERENCES判断存储函数的纯度时，如果函数调用了SQLCODE和SQLERRM，我们就不能指定约束为WNPS和RNPS了。

5、捕获未控制异常

记住，如果被抛出的异常找不到合适的异常控制程序，PL/SQL会向主环境抛出一个未捕获的异常错误，然后由主环境决定如何处理。例如，在Oracle预编译程序环境中，任何一个执行失败的SQL语句或PL/SQL块所涉及到的改动都会被回滚。

未捕获也能影响到子程序。如果我们成功地从子程序中退出，PL/SQL就会把值赋给OUT参数。但是，如果我们因未捕获异常而退出程序，PL/SQL就不会为OUT参数进行赋值。同样，如果一个存储子程序因异常而执行失败，PL/SQL也不会回滚子程序所做的数据变化。

我们可以在每个PL/SQL程序的顶级使用OTHERS句柄来捕获那些没有被子程序捕捉到的异常。

九、PL/SQL错误控制技巧

这里，我们将[学习](http://blog.csdn.net/narutobing/article/details/7994310)三个提高程序灵活性的技巧。

1、模拟TRY..CATCH..块

异常控制程序能让我们在退出一个块之前做一些恢复操作。但是在异常程序完成后，语句块就会终止。我们不能从异常句柄再重新回到当前块。例如，如果下面的SELECT INTO语句引起了ZERO\_DIVIDE异常，我们就不能执行INSERT语句了：

DECLARE  
   pe\_ratio   NUMBER(3, 1);  
BEGIN  
DELETE FROM stats  
        WHERE symbol = 'XYZ';

SELECT price / NVL(earnings, 0)  
    INTO pe\_ratio  
    FROM stocks  
   WHERE symbol = 'XYZ';

INSERT INTO stats(symbol, ratio)  
       VALUES ('XYZ', pe\_ratio);  
EXCEPTION  
WHEN ZERO\_DIVIDE THEN  
     ...  
END;

其实我们可以控制某一条语句引起的异常，然后继续下一条语句。只要把可能引起异常的语句放到它自己的子块中，并编写对应的异常控制程序。一旦在子块中有错误发生，它的本地异常处理程序就能捕获并处理异常。当子块结束时，封闭块程序会继续执行紧接着的下一条语句。如下例：

DECLARE  
   pe\_ratio   NUMBER(3, 1);  
BEGIN  
DELETE FROM stats  
        WHERE symbol = 'XYZ';

BEGIN   -- sub-block begins  
    SELECT price / NVL(earnings, 0)  
      INTO pe\_ratio  
      FROM stocks  
     WHERE symbol = 'XYZ';  
EXCEPTION  
    WHEN ZERO\_DIVIDE THEN  
       pe\_ratio     := 0;  
END;   -- sub-block ends

INSERT INTO stats(symbol, ratio)  
       VALUES ('XYZ', pe\_ratio);  
EXCEPTION  
WHEN OTHERS THEN  
     ...  
END;

在上面这个例子中，如果SELECT INTO语句抛出了ZERO\_DIVIDE异常，本地异常处理程序就会捕捉到它并把pe\_ratio赋值为0。当处理程序完成时，子块也就终止，INSERT语句就会被执行。

2、反复执行的事务

异常发生后，我们也许还不想放弃我们事务，仍想重新尝试一次。这项技术的实现方法就是：

把事务装入一个子块中。  
把子块放入一个循环，然后反复执行事务  
在开始事务之前标记一个保存点。如果事务执行成功的话，就提交事务并退出循环。如果事务执行失败，控制权就会交给异常处理程序，事务回滚到保存点，然后重新尝试执行事务。  
如下例所示。当异常处理程序完成时，子块终止，控制权被交给外围块的LOOP语句，子块再次重新开始执行。而且，我们还可以用FOR或WHILE语句来限制重做的次数。

DECLARE  
   NAME     VARCHAR2(20);  
   ans1     VARCHAR2(3);  
   ans2     VARCHAR2(3);  
   ans3     VARCHAR2(3);  
   suffix   NUMBER        := 1;  
BEGIN  
   ...  
LOOP   -- could be FOR i IN 1..10 LOOP to allow ten tries  
    BEGIN   -- sub-block begins  
     SAVEPOINT start\_transaction;   -- mark a savepoint

      /\* Remove rows from a table of survey results. \*/  
      DELETE FROM results  
            WHERE answer1 = ’no’;

      /\* Add a survey respondent’s name and answers. \*/  
      INSERT INTO results  
           VALUES (NAME, ans1, ans2, ans3);

      -- raises DUP\_VAL\_ON\_INDEX if two respondents have the same name  
      COMMIT;  
      EXIT;  
    EXCEPTION  
     WHEN DUP\_VAL\_ON\_INDEX THEN  
        ROLLBACK TO start\_transaction;   -- undo changes  
         suffix     := suffix + 1;   -- try to fix problem  
         NAME       := NAME || TO\_CHAR(suffix);  
    END;   -- sub-block ends  
END LOOP;  
END;

3、使用定位变量标记异常发生点

只用一个异常句柄来捕获一系列语句的话，可能无法知道到底是哪一条语句产生了错误：

BEGIN  
SELECT ...  
SELECT ...  
EXCEPTION  
WHEN NO\_DATA\_FOUND THEN ...  
-- Which SELECT statement caused the error?  
END;

要想解决这个问题，我们可以使用一个定位变量来跟踪执行语句，例如：

DECLARE  
   stmt INTEGER := 1;   -- designates 1st SELECT statement  
BEGIN  
SELECT ...  
   stmt := 2;   -- designates 2nd SELECT statement  
SELECT ...  
EXCEPTION  
WHEN NO\_DATA\_FOUND THEN  
    INSERT INTO errors VALUES ('Error in statement ' || stmt);  
END;