PL/SQL，即“Procedural Language extensions to the Structured Query Language”

Oracle公司发明PL/SQL为了克服SQL语言自身的一些局限性，并为那些要在Oralce数据库中运行的核心业务，提供一种更完整的解决方案。这一章将介绍PL/SQL，包括它的起源、版本变迁。

## 1、什么是PL/SQL？

* 它是一种高度结构化、易读、易理解的语言；
* 标准的、可移植的Oracle开发语言；
* 嵌入式语言；
* 高性能的、高度集成的数据库语言。

## 2、PL/SQL的起源

Oracle公司一度领导软件工业为设计数据库和应用程序提供一种声明式的、非过程化的方法。

### 2.1早期的PL/SQL

1988年，Oracle公司发布了Oralce版本6，这个版本出现了一个“程序选项”或者PL/SQL的关键组件。差不多同一时间，Oracle发布了SQL\*Forms 2.3版。SQL\*Forms 3.0版首次在工具端整合了PL/SQL引擎，开发人员可以以一种自然的、直观的方法编码程序逻辑。

第一版的PL/SQL功能非常有限。在服务器一端，我们只能用PL/SQL编写“批处理”形式的过程以及SQL脚本语句。在客户端，SQL\*Forms 3.0版确实能够创建过程和函数，尽管支持函数但是因为没有文档说明，因此多年来大部分开发人员无法使用这个功能。第一版的PL/SQL既不支持数组也不能和操作系统进行交互（输入或输出）。

产品架构中有两个主要缺陷：缺乏可移植能力和执行授权问题。

### 2.2改进的应用程序移植能力

在PL/SQL出现时，基于V语言的数据库可以在许多不同的操作系统和硬件平台上运行，SQL\*Plus和SQL\*Forms也很容易适应不同的终端配置环境。尽管如此，还是有许多的应用程序需要利用主机语言比如COBOL、C、FOTRAN提供的复杂精细的控制能力。一旦开发人员脱离了中立的Oracle工具，最后应用程序就不再是可移植的了。

PL/SQL语言的目的就是为了扩宽应用需求的范围，提供一种完全操作系统独立的编程工具。

### 2.3改进执行授权和事务一致性

比可移植性更基本的问题就是执行授权。数据库和SQL语言让我们可以紧密地控制对任何个别数据库表的访问和修改。例如，通过GRANT命令，我们可以确保只有特定的角色和用户可以对指定表执行更新操作。另一方面，GRANT命令不能确保用户或者应用执行的更新操作能够正确执行，换句话说，数据库不能保证跨越多个表的事务的一致性，在业务事务中涉及多个表是很常见的。

PL/SQL语言为逻辑事务提供了紧密的控制和管理能力。PL/SQL实现这种能力的一个方式就是通过执行授权。我们不再是给一个角色或者用户对某个表的更新权限，我们只是授权执行某个过程，后者控制着对底层数据结构的访问。这个过程是由另外一个Oracle数据库模式所有（程序的“定义者”），是给这个模式授予执行事务需要用到的表的更新权限。现在这个过程成了事务的“门卫”。程序能够运行的唯一方法就是通过这个过程。利用这种方法，整个应用程序的事务一致性都得到了保证。

从Oralce 8i开始，Oracle用通过提供AUTHID子句为PL/SQL的执行授权模型添加了更多的灵活性。使用AUTHID，我们可以继续用之前描述的定义者权限模型运行程序，或者我们也可以选择使用AUTHID CURRENT\_USER（调用者权限），这时这个程序是用调用者（当前）模式的授权运行的。调用者权限模型只是这么多年来PL/SQL逐渐成熟和更加灵活的一个例子。

### 2.4低调起步，稳定地前进

Oracle的PL/SQL语言保证了我们可以完全待在独立于操作系统的Oracle环境中，同时还能编写出满足用户需求的高效应用程序。

今天的PL/SQL基本上可以帮助我们完成任何需要的功能。

数据库的每一个新版本，Oralce都会对PL/SQL语言本身进行一些稳定的基本的改进。提供了大量的（内置的）包在多方面扩展了PL/SQL语言的能力。已经引入了面向对象功能，实现了多种类似数组的数据结构，对编译器进行增强既能够优化我们的代码也可以对可能的质量和性能问题提出警告，总之从广度和深度上对语言进行了提升。

## 3、这就是PL/SQL

下面通过几个例子演示一下PL/SQL结构和功能有关的一些关键元素。

### 3.1与SQL整合

PL/SQL一个最重要的方面就在于它和SQL是紧密整合的。我们在PL/SQL程序中运行SQL语句无需借助任何类似ODBC（开发数据库连接）或者JDBC（Java数据库连接）这样的中间软件“粘合剂”。

DECLARE

l\_book\_count INTEGER;

//声明部分：在此声明PL/SQL用到的变量，类型及游标，以及局部的存储过程和函数

//这里声明了一个整形变量以装载有我编著的图书数量。

BEGIN

SELECT COUNT(\*)

INTO l\_book\_count

FROM books

WHERE author LIKE ‘%FEUERSTEIN, STEVEN%’;

DBMS\_OUTPUT.PUT\_LIKE (

‘STEVEN has written (or co-written)’ || l\_book\_count || ‘books.’);

UPDATE books

SET author = REPLACE(author,’STEVEN’,’STEPHEN’)

WHERE author LIKE ‘%FEUERSTEIN, STEVEN%’;

//执行部分：过程及SQL语句，及程序的主要部分。

END;

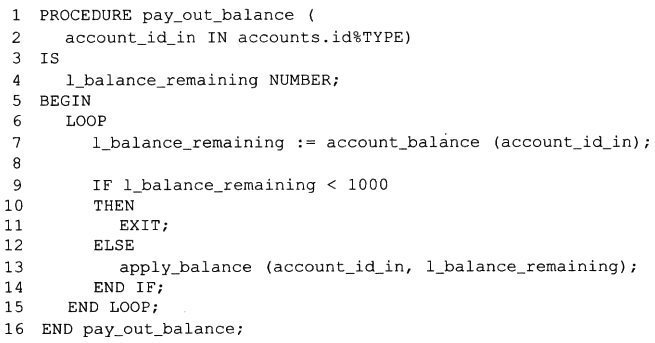
1. 关键字BEGIN代表着执行单元的开始——当我把这个块传给SQL\*Plus时，这部分代码就会执行。
2. 通过一个SELECT查询到我编著的图书数量。这里的INTO子句事实上并不属于SQL语句的一部分，但是充当了数据库和局部PL/SQL变量的“桥梁”作用。
3. 用内置的DBMS\_OUTPUT.PUT\_LINE过程（也就是说，由Oracle提供的DBMS\_OUTPUT包中的一个过程）显示图书的数量。
4. 把姓的拼写改成“Stephen”，因此发出了一个对books表的更新操作。我通过内置的REPLACE函数找到所有的“STEVEN”然后用“STEPHEN”替换它们。

### 3.2控制和条件逻辑

PL/SQL提供了一套完整的语句我们可以紧密的控制流程执行哪一行代码。这些语句包括以下几项。

* IF和CASE语句：这些语句实现了条件逻辑，例如，“如果一本书的页数大于1000，则……”。
* 完整的循环及迭代控制：包括FOR循环、WHILE循环及简单循环。
* GOTO语句：可以无条件地从程序的一部分转到另一部分。不过，这并不意味着我们就应该用它。

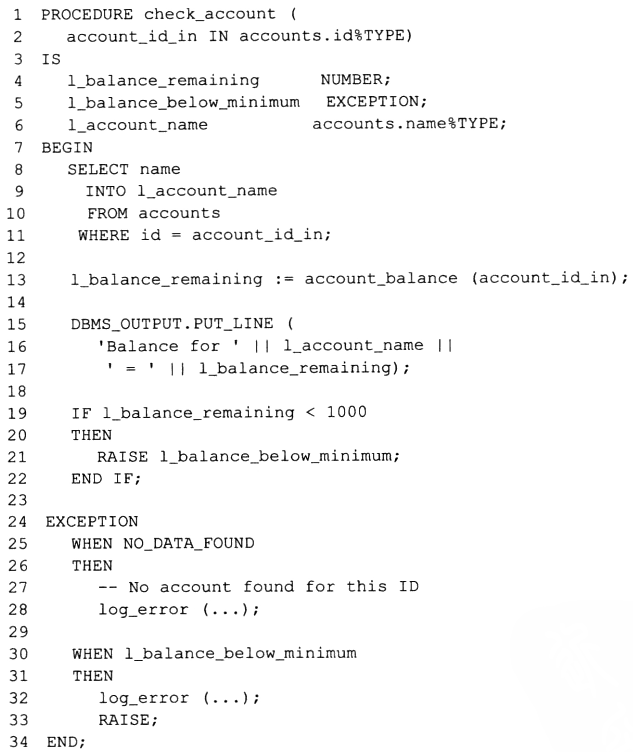
下面是一个过程（一个可以通过名字进行调用的可重用代码块）说明了这些特性。



1. 块头（可选）：块头确定命名块或程序的调用方式。这个工程使用账户余额支付未付账单。account\_id\_in IN accounts.id%TYPE是这个过程的类型参数。这个例子只有一个传入值（账户的标识码）。
2. 3、4行是声明单元。注意不是使用DECLARE关键字，之前的例子用的是DECLARE关键字，而是用关键字IS（或者AS）来把过程头部和声明单元区分开。
3. 6~15行，执行单元：一个简单循环。这个循环用EXIT语句结束了循环，FOR循环和WHILE循环用不同的方式指定结束条件。第7行：调用了account\_balance函数取得账户余额，将其赋值给变量I\_balance\_remaining。（:=赋值符号）。这是在一个可重用程序中调用另外一个可重用程序的例子。13行：在这个过程中调用另外一个过程。9~14行：这个IF语句表示如果账户余额小于$1 000，停止拨款平账。否则，支付余额。

### 3.3出现错误了

PL/SQL语言同时为抛出错误和处理提供了强大的处理机制。在下面这个过程中，我根据ID获得某个账户的名字和余额，然后查看余额是否太少了；如果太少了，就明确地抛出一个异常，终止程序继续操作。



1. 第5行：声明一个自己的异常，叫做i\_balance\_below\_minimum。Oracle提供了一套预定义的异常，比如DUP\_VAL\_ON\_INDEX，不过我的程序还是需要某些专用的异常，因此这个例子中我必须自己定义一个异常。
2. 8~11行：这个查询取出账户的名字。如果没有这个ID的账户，数据库就会抛出一个NO\_DATA\_FOUND异常，结束这个程序。
3. 19~22行：如果余额太少，我明确地抛出自定义的异常，因为这个账户已经出现问题了。
4. 24行：关键字EXCEPTION表明这是执行单元的末尾，以及异常处理单元的开始，这个单元会处理错误。
5. 25~28行：这个错误处理部分针对的是账户没有找到的错误。如果抛出的异常是NO\_DATA\_FOUND，这个异常就在这里被捕获，然后用log\_error过程记录错误日志。
6. 30~33行：这个错误处理部分针对的是账户余额太少的错误（我这个应用程序特有的异常）。如果抛出的异常是I\_balance\_below\_minimum，这个异常在这里被捕获，错误记下来。然后考虑错误的严重性，（RAISE）我又一次把这个错误抛出去，把这个错误传出当前过程传给调用它的PL/SQL模块。

## 4、关于PL/SQL的版本

### 4.1Oracle 11g的新特性

1）基于版本的重定义功能（只有R2版本才有）

Oracle 11g R2版本引入了一个革命性的新功能，可以在不打断应用程序运行的前提下在线对应用程序进行升级。现有的会话可以继续使用升级前的应用程序，直到这些程序的用户确定了结束；然后，同一时刻，新的会话就可以使用升级后的应用程序了。当没有会话再使用升级前的应用程序，它就可以退役了。于是，应用程序作为一个整体可以从升级前版本热过度到升级后的版本。

这个新功能用到了许多的数据库特性，不过最主要的还是基于版本的重定义。

2）CREATE TYPE的FORCE选项（只有R2版本才有）

现在可以通过“CREATE OR REPLACE”指明我们要强调创建一个新类型，就算来由其他类型依赖它。在以前的版本中，这种操作尝试会抛出ORA-02303异常。

3）函数结果缓存

在Oracle 11g R1的版本中，实现了函数结果缓存，并在R2版本加强。这个功能提供的缓存解决方法克服了基于包的缓存的一些不足，而性能差不多同样得快。当我们为某个函数打开函数结果缓存时，我们可以得到下面这些好处。

* Oracle数据库用一个单独的缓存区为每一个函数同时保存输入和返回值。这个缓存区被连接到这个数据库实例的全部会话共享，而不是为每一个会话复制。
* 每当函数被调用时，数据库救护检查是否已经缓存了相同的输入值。如果是，则函数就不用执行了。把缓存中的值简单返回就可以了。
* 每当发现要修改的是缓存所依赖的表时，数据库会自动的把缓存无效。后续的函数调用会用一致数据重新填充缓存。

4）CONTINUE语句

用这个语句可以退出循环的当前迭代，继续循环的下一次迭代。这个语句有两种形式，就像EXIT：无条件的CONTINUE以及有条件的CONTINUE WHEN。

下例在遇到偶数时跳过循环体的执行：

BEGIN

FOR I\_index IN 1 .. 10

LOOP

CONTINUE WHEN MOD(I\_index,2)=0;

DBMS\_OUTPUT.PUTLINE(‘Loop index=’ || TO\_CHAR(I\_index) );

END LOOP;

END;

5）PL/SQL表达式中的序列

可以在PL/SQL中使用sequence\_name.CURRVAL和sequence\_name.NEXTVAL，不再需要SELECT FROM SYS.dual了。

6）动态SQL的增强

PL/SQL的开发团队已经大大地增强了两类动态SQL（DBMS\_SQL和原生态SQL）之间的互操作性，以及功能覆盖的完整性的改进。比如，我们现在可以在一个DBMS\_SQL游标数值和游标变量间相互转化。我们也可以EXECUTE IMMETIATE一个CLOB。

Oracle 11g还加强了DBMS\_SQL的安全性。现在这个包可以防止这种情况：一个程序使用DBMS\_SQL并且抛出异常，攻击者就可以利用未关闭的游标危害数据库的安全。对全权的加强包括生成不可预测的（可能是随机的）游标数值，当传给DBMS\_SQL的是无效的游标数值时限制DBMS\_SQL的使用，如果用户要使用的游标和游标被用户打开时已经不一样了，也会拒绝DBMS\_SQL的操作。

7）新的原生编译和SIMPLE数据类型

PL/SQL原生编译器现在可以直接生成原生机器代码，而不是把PL/SQL代码翻译成C代码，然后再由C编译器生成机器随机码。使用原生编译现在也很简单：每个开发人员都无需DBA的介入就可以对PL/SQL单元进行原生编译。我们可以期望原生编译代码的执行速度会有大幅提升，也许是一个数量级升级。而且如果打开原生编译，我们可以从某些新的、专门的数值类型获得性能提升：SIMPLE\_INTEGER、SIMPLE\_FLOAT和SIMPLE\_DOUBLE。

8）SecureFiles

有关LOB实现方式的术语在Oracle 11g已经变了。Oracle用一个叫做SecureFiles的技术重新实现了LOG。SecureFiles对LOG管理的很多方面都做了改进，包括磁盘格式，缓存，锁，重做日志，以及空间管理算法。这种更新的技术显著的改善了LOG的性能，还可以通过简单的参数设置实现LOG的复制、压缩、以及加密。

9）触发器的增强

我们现在可以创建复合触发器，可以把以前是分开的触发器（事件的BEFORE和AFTER触发器）合并到一个代码体中，这个代码体中有不同的事件单元。这种触发器会让维护复杂的触发器逻辑变得更简单，尤其是对突变表触发器错误。如果同一个事件（以及同一个数据库对象）上定义了多个触发器，现在我们也可以定义这些触发器的执行顺序了。

10）自动的子程序插入

一个新的编译器优化级别（级别3）实现了自动的子程序插入，也就是说编译器把一个局部子程序调用（在同一个PL/SQL单元声明的子程序）用子程序的实现代码拷贝替换。这个优化可以减少运行时的代码执行量因为不再需要“查找动作”发现子程序并执行。

11）PL/Scope

PL/Scope是一个编译器驱动的工具，可以从PL/SQL源代码中收集整理有关用户自定义标识符的数据，并让这些数据可以通过ALL\_IDENTIFIED数据字典视图的使用。PL/Scope使得我们的应用程序构建一个自动的、复杂的质量保证的程序非常容易。我们会非常喜欢在使用的PL/SQL编辑器中使用PL/Scope，不过我们也可以基于ALL\_IDENTIFIERS编写一些查询来“挖掘”我们的代码。

12）PL/SQL层次化分析器

在Oracle 11g中，Oracle用一个新的层次化分析器完善现有的PL/SQL分析器（DBMS\_PROFILER）。通过使用提供的DBMS\_HPROF包，我们可以获得按照子程序调用进行组织的PL/SQL代码动态进行的分析信息。这个分析器把SQL和PL/SQL的执行次数分开统计。每个子程序级别的统计都包括关键数据、子程序的调用次数、子程序花费的时间以及这个子程序的子树（它又调用的子程序）花费的时间，以及详细的父子信息。

13）细粒度依赖关系跟踪

在Oracle 11g之前，对依赖关系的记录粒度是把整个对象作为一个整体看待。如果对象有了任何改动，所有依赖程序单元都会被标志成无效，即使这个修改根本没有影响到这个程序单元。在Oracle 11g中，Oralce把依赖关系的跟踪微调到对象内的元素级别。比如，对于表来说，Oracle数据库现在记录的是某个程序单元依赖表中的某些列。利用细粒度的依赖关系跟踪，数据库可以避免某些之前版本数据库需要的重新编译，从而使得应用程序代码升级更加容易。

## 5、PL/SQL开发人员的资源