

## 第28章 PL/SQL基础

本章要点：

理解PL/SQL

理解PL/SQL引擎

在工具箱里增加PL/SQL

开始学习PL/SQL

语言教程

Oracle 8i特性

### 28.1 理解PL/SQL

PL/SQL是ANSI标准SQL的Oracle版本的过程化语言的扩展。SQL是非过程化语言，程序员只需要说明执行什么工作，至于如何执行则留给 Oracle服务器的 SQL优化器。相反，PL/SQL，像第三代（3GL）过程化语言一样，要求一步步指导来确定下一步做什么。

类似其他的工业标准语言，PL/SQL提供变量声明、赋值、条件测试与分支以及迭代的语言元素。正如C或Pascal一样，它侧重于面向块。PL/SQL服从严格的作用域原则，提供参数化子程序结构，并且类似Ada语言，有一个称为包的类似于容器的特性，由程序员随意决定隐藏或显示数据和函数。它是强类(型)语言，在编译和运行时捕捉数据类型不符的错误。可以进行显式的和隐式的数据类型转换，支持复杂的用户自定义数据结构，子程序可以被重载，以创建一个灵活的、模块化的环境。

另外，因为它是过程化封装的 SQL，该语言能够很好地与 SQL结合使用。某些语言特性让它能够与Oracle RDBMS交互操作，执行集合与单条记录操作。你所了解用于编写 SQL程序的知识越多，你所设计的PL/SQL程序就越好。

PL/SQL提供了一种称为异常事件处理（Exception Handling）的特性，同步地处理事务处理期间可能发生的错误和类似事件。你将学习如何在 PL/SQL代码中嵌入意外事件处理程序以漂亮地处理错误状态，例如NO\_DATA\_FOUND或DUP VAL ON INDEX。

PL/SQL不是面向对象的语言，它具有在其他的语言，例如 Pascal和Ada语言中见到的一些特性。如果熟悉 Pascal语言的语法，学习 PL/SQL不会有任何困难。不像 C和Pascal语言，PL/SQL不支持指针的使用。PL/SQL是主要的后端开发工具，在那里它严格地与数据库表和其他的数据库对象进行交互操作。通过使用所提供的数据库工具包，处理 PL/SQL与操作系统以及外部软件组件的交互操作。

PL/SQL在所有的Oracle平台上是100%可移植的。因为它的数据类型基于数据库服务器上的数据类型，该语言与机器完全无关。不需要学习 UNIX、Windows NT、NetWare等等各种各样的特性。一个PL/SQL程序可以不需要任何修改，在所有的 Oracle服务器上编译并运行。

这种可移植性也扩展到第三代（3GL）编程语言。通过 Oracle提供的预编译器，PL/SQL针对各种语言例如 C和COBOL语言提供了一个标准接口。预编译器支持内嵌 SQL的ANSI标

准。

使用Oracle 8i，Oracle将重点集中到RDBMS与国际互联网（Internet）的结合。除了编写PL/SQL应用，让世界上的所有浏览器能够通过Oracle应用服务器进行访问的能力外，Oracle8i还增加了直接从PL/SQL过程中发出HTTP请求的能力。Oracle还将大量的工作放到存储在数据库中的PL/SQL和Java语言的互用性上，以使它们相互之间可以无缝调用。

## 28.2 理解PL/SQL引擎

在把PL/SQL看作一种语言之前，需要在执行环境中理解它。

### 28.2.1 适合客户/服务器环境

在一个客户/服务器配置中，真正的瓶颈通常是网络。当几百个用户通过编译的C、C++、Delphi或COBOL程序连接到Oracle服务器时，将得到一个非常缓慢的网络系统。解决方案是将复杂的程序段，尤其是那些执行重复的或相关的SQL语句，合并成一个PL/SQL块。这些块可以被嵌入到一个OCI（Oracle调用接口，Oracle Call Interface）程序中，或通过以下方法更为有效地执行：将它们转移到数据库内部作为存储的函数、过程或包。图28-1显示了客户应用与Oracle服务器之间典型的交互作用。

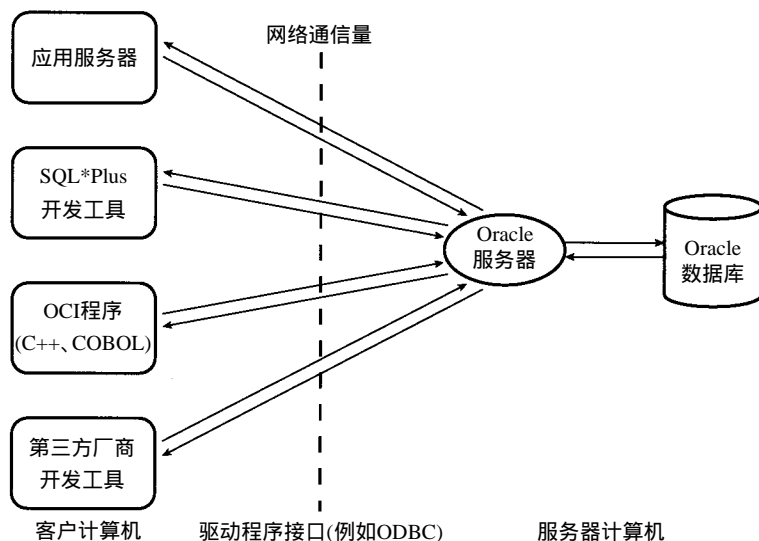


图28-1 一个典型的客户/服务器环境

PL/SQL程序由PL/SQL引擎执行，PL/SQL引擎是数据库服务器的一部分。图28-2表明一个PL/SQL块在内部是如何处理的。

不论使用哪种工具，例如Oracle SQL\*Plus，该工具必须向Oracle服务器提交PL/SQL源代码，PL/SQL引擎扫描、分析并编译代码，然后编译后的代码预备执行。在执行期间，将所有的SQL语句传递给SQL语句执行器（SQL Statement Executor）组件执行。SQL语句执行器执行SQL或DML语句。PL/SQL引擎可以使用由查询提取的数据集进行更进一步的处理。

相对于逐条发送一组SQL语句，使用PL/SQL块执行一组SQL语句的一个好处是降低了网络传输量，图28-3表明了这种思想。

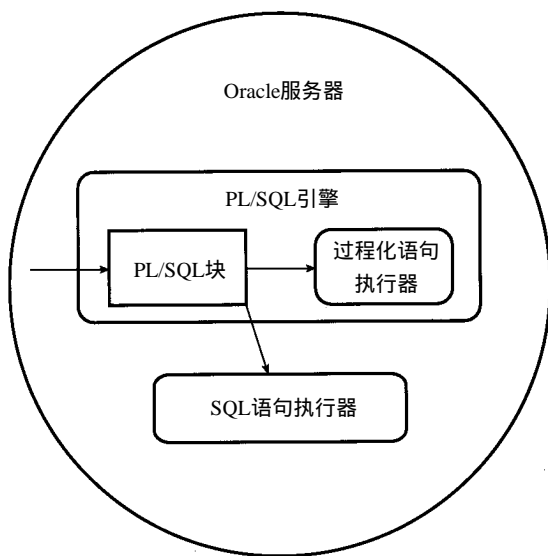


图28-2 PL/SQL引擎是Oracle数据库服务器的一个组件

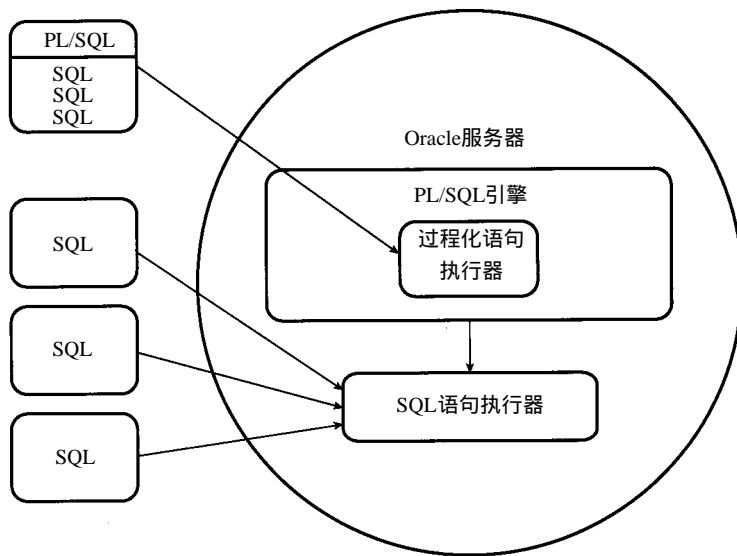


图28-3 将几个SQL语句组合为一个PL/SQL块降低了网络传输量

单独这样做，可以明显地提高应用的性能。另外，SQL/DML语句可以当作一个事务对待。如果整个事务成功，那么对数据库的全部改动会被提交。如果事务的任何部分失败，整个事务会被回滚。因为PL/SQL块中可以包含复杂的逻辑，因此在服务器上执行，客户程序的大小和复杂程度降低了。

#### 1. 执行存储子程序

一个更深入的改进包括在数据库内部存储编译的命名的PL/SQL块。PL/SQL块在本章中统称为存储子程序或称为子程序。“命名”只是意味着子程序包含代码与名称，就像C或Pascal子程序。图28-4显示PL/SQL引擎如何调用存储子程序。

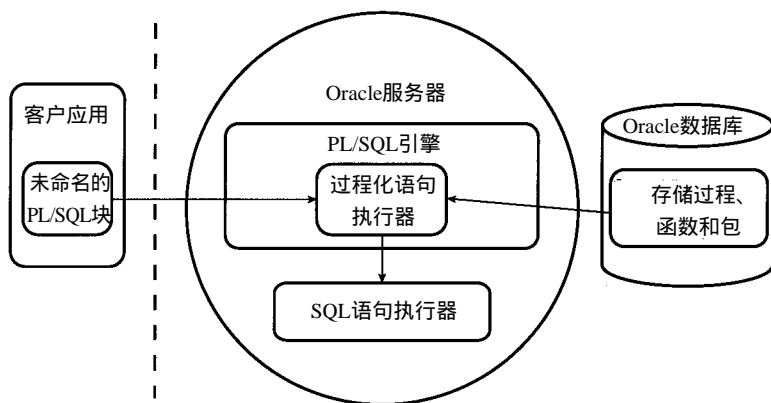


图28-4 PL/SQL引擎运行存储子程序

这些子程序可以执行复杂的逻辑和错误处理。在一个客户应用中嵌入的一个简单的匿名或未命名的块（没有使用名字标记的 PL/SQL 代码块）能够激活这些子程序。这种能力通常称为远程过程调用（Remote Procedure Call, RPC）。子程序还可以调用其他子程序。因为这些子程序已经被编译并被开发者很好地调整过，它们提供了显著的性能改进，并通过为其他应用或模块提供可重复使用的块减少了应用开发。

## 2. 系统全局区内的共享 SQL 区

系统全局区（System Global Area, SGA）是操作系统为 Oracle 服务器分配的一大块内存区域。在这个内存区域中，服务器维护表数据、游标、用户本地变量和其他各式各样项目的本地拷贝。

当编译任意 PL/SQL 程序时，不管是命名块还是未命名块的代码，源代码和对象代码都被高速缓存在共享 SQL 区中。分配给一个 PL/SQL 块的空间叫做一个游标。服务器使用最近最少使用算法，在共享 SQL 区保存缓存的程序直到它过期失效。在 PL/SQL 块中的任何 SQL 语句都被给予各自的共享 SQL 区。当一个命名子程序被编译时，它的源代码还被保存到数据字典中。

包含在一个子程序中的代码是可重入的，也就是说，它在所有连接的用户间是共享的。当提交一个未命名的 PL/SQL 块给服务器执行时，服务器通过比较源代码文本，决定在高速缓存中是否有该程序块。如果代码文本字符之间（包括大小写）完全相同，那么将执行缓存的被编译的代码。对于 SQL 语句这点也适用，如果查询文本完全相同，能够简单执行缓存的分析代码。否则，必须首先分析新的语句。通过共享可执行代码，一个基于服务器的应用能够真正地实现内存节省，这在拥有数百个连接用户时尤其重要。

## 3. 私有 SQL 区

如果几个用户正在执行相同的代码块，服务器如何分开保存它们各自的数据呢？每个用户的会话将得到一个私有 SQL 区。这块内存含有子程序中包含的变量数据的私有拷贝。还要为 PL/SQL 块中的任意 SQL 语句分配一个私有 SQL 区。图 28-5 显示这种模式。

一个存储子程序首次被引用时，必须将它从数据库中加载到 SGA 区。一旦加载后，它对于每个用户都是可用的。只要它继续被任意用户调用（不必是第一个调用它的人），该子程序将继续保留在内存中。包的工作方式与之相同，只是包中可能包含一整套子程序。最初的加载过程可能会花费较长的时间，但是现在所有这些子程序都是可用的，不再需要与磁盘打交

道。当预期激活一套子程序时，最好一次将它们全部加载。包提供了这种机制。

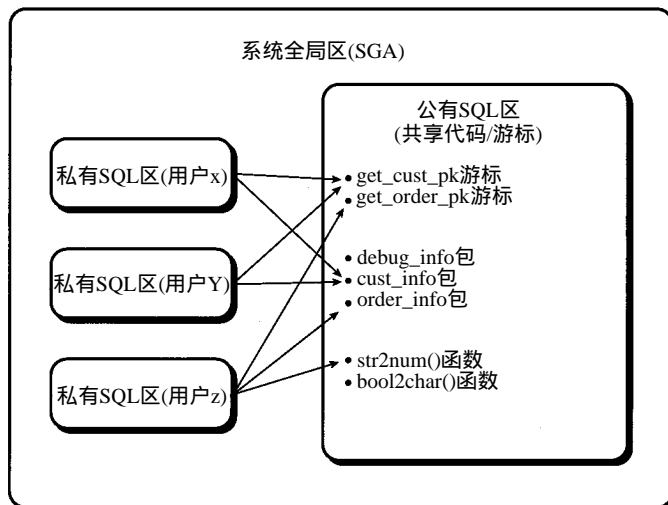


图28-5 在SGA中的共享和私有SQL区

### 28.2.2 适合客户环境

PL/SQL引擎还可以嵌入到某些 Oracle 工具中，例如 SQL\*Forms 4.5 和其他的 Oracle 开发者开发工具，主要的好处是使用熟悉的语言编程。客户程序可以执行在本地 PL/SQL 块中包含的计算，并能够向服务器发送 SQL 命令或激活存储的 PL/SQL 部分。另外，通过直接在数据库服务器上处理复杂的逻辑部分，PL/SQL 引擎支持代码重用并简化了客户程序。

### 28.2.3 对比服务器端与客户端开发

开发者必须有意识地决定在服务器上隐藏多少复杂性，以及在客户程序中保留多少复杂性。经过几年客户/服务器应用的开发，作者建议几条需要遵循的指导原则：

**减少网络传输量。**将一些功能放到服务器上，网络在任何客户/服务器应用中是典型的瓶颈。

**开发标准代码。**标准代码可以在新应用中或当添加到现有应用中时再次使用。这种方法节省了开发努力。

**力争低偶联与高内聚性。**当一个模块依赖于另外一个模块中的特定内容时，例如当使用全局变量而不使用参数传递时，会出现偶联。内聚性意味着放置相似的元素到相同的位置，例如将全部数学函数捆绑为一个数学库。

**隐藏实现的细节。**这样会减少客户程序的复杂性并拆分实现的功能。拆分是指消除或避免模块之间的依赖。例如，当实现自顶向下的设计时，将实现细节推动到一个较低的层次，隐藏实际的“解释作法”。

**以普通方式编写模块。**一个模块越特殊，重用的可能就越小。查找模型和公共特性。但这并不意味着将几个不同的行为塞入一个模块中。

**以一致的、集中的方式处理商务原则。**这使它们在将来的工程项目中更为直观和可重用。如果你的商务原则经常变化并且跨越多个客户程序模块，你必须到处定位和修改

它们——一个既无效率又不便于维护的过程。

**提示** 克莱美格可维护性原则 (Clamage's Rule of Maintainability): 一个高度可维护程序模块是指当对模块的需求改变时, 高度可维护性模块需要少量修改或根本不需要修改。一个不可维护程序是指程序需要做实质的修改以加入新的需求。上述原则会帮助指导朝着建立高度可维护性软件的目标前进。

使用存储子程序实现这些原则。通过提供存储子程序的标准库, 多个和将来的客户应用可以利用较早完成的开发。例如, 商务原则可以作为存储子程序得以实现, 如果一条原则发生改变, 例如计算一个销售佣金的原则发生变化, 仅仅需要修改并重新编译适当的存储程序。倘若存储子程序的接口保持不变, 全部相关的客户程序不需要改动或甚至不需要重新编译。

通过允许子程序的名称被重载, 包是提高子程序的通用性的一种好方法。重载简单地说是给一组代码相同的子程序名称, 通常在参数表中具有不同的数据类型或不同数目的参数。例如, 加运算符在几乎所有的计算机语言中被重载, 以处理整数和浮点数运算。

## 28.3 在工具箱里增加PL/SQL

稍停几分钟, 考虑一下在日常任务中如何使用 PL/SQL。你或许没有被卷入到应用开发工作中。作为一个数据库系统管理员, 必须对数据库进行持续的维护。可以使用 PL/SQL让日常工作变得较为容易。

### 28.3.1 加强SQL脚本

事实上, 一些 SQL\*Plus脚本能够生成其他的脚本, 例如列出表空间中每个表的记录数量。而且, 这些脚本可以用 PL/SQL重写、编译并保存到数据库中, 因此运行速度非常快, 而且不必将中间脚本写入文件。

编写这些类型的脚本也许有些困难; 一个 PL/SQL脚本更为简明易懂并且易于书写和维护。另外, 一个已编译的 PL/SQL程序能够很容易地在数据库系统管理员之间共享 (尤其当你在一个 “24x7” 商店中, 24是指每天24小时, 7是指每周7天, 24x7意即连续不间断交易的商店)。不需要查找硬盘以找出脚本, 每个人总能知道它们在哪里及如何运行它们。

存储子程序可以接收参数, 这使它们非常灵活。你甚至可以为它们编写 SQL\*Plus前端程序, 收集用户输入数值并传入这些数值, 而不必知道参数的顺序和类型。

### 28.3.2 简化数据库管理

或许有一套很好的 SQL脚本, 为你提供运转着的数据库的信息。但在理解 PL/SQL后, 你会希望得到一套全新的程序, 为你提供有关数据库性能、存储、用户负载、锁等等附加信息。PL/SQL消除了普通SQL脚本的约束和限制。

通过使用提供的工具包按照设定的间隔运行它们, 可以自动执行许多任务。可以将系统或应用数据发送给外部的性能监控器 (可能用 C++或Delphi语言编写), 并且使用它们强大的报表和图形功能实时显示这些信息。

有些工具, 例如 Visual Basic很容易使用。加上 PL/SQL提供的附加能力, 可以开发可视化工具以简化任务, 例如运行 EXPLAIN PLAN和以图形方式查看表的索引和约束。几个可扩展

的第三方厂商工具就是采用这种方式开发的。

### 28.3.3 以较少的争论得到较好的信息

可以编写使用数据字典的 PL/SQL 程序去显示任意指定表的详尽索引信息，即使该表在另外一个模式中。如果你不是作为 SYS 或 SYSTEM 用户登录的，你需要一些数据字典的信息，并且不希望进行重连接，这样做非常好。这对于开发者尤为重要，他们经常需要存取这种类型的信息。可以使用 PL/SQL 逆向工程设计任意数据库对象，再建一个语法纠正语句以重建该对象。这样一来，每次改变一个表或授权一个新的权限时，可以解脱手工更新脚本带来的麻烦。

如果编写一条大且复杂的 SQL 语句有困难，可以将它分割成较小的部分，让它们在一个 PL/SQL 脚本中一起运行。这样做胜过尝试连接七个以上的表，连接七个以上的表会让 SQL 执行器陷入困境。通过主表单独查询，可以显著改善多表连结。还有，当知道 PL/SQL 脚本内部的重用代码已工作正常时，验证脚本的输出结果很容易。

### 28.3.4 设计更好的数据库应用

可以使用存储过程和包中的 PL/SQL 建立更好的应用。设计和创建可重用的 PL/SQL 模块的共同努力可以得到长远的利益。通过深谋远虑地设计和计划，可以做到：

- 当创建下一个应用时，可以借鉴一个应用已取得的成果。

- 在几个相关的应用之间分摊开发费用。

- 提供对底层表与其他对象的一致性接口

- 简化并优化数据存取。

- 降低应用维护和展开费用。

- 实施编码标准化。

使用 PL/SQL 是简单而有趣的。与语言相比，它只需要相当短的学习过程。通过试验可判断哪些能够很好地工作或哪些根本就不能工作。如果按照本节详述的步骤去做，只需要几天就能够工作，在短短的几个月内，你就会完全掌握 PL/SQL，取得一些高级的成果。

## 28.4 开始学习 PL/SQL

本节详述在操作 PL/SQL 前，数据库系统管理员必须要做的工作。对工作所在的应用环境的体系结构也将做一探讨。

在准备编写 PL/SQL 前，你或你的数据库系统管理员首先必须做下列工作：

- 授予你（也就是你的 Oracle 帐户名）CREATE PROCEDURE 权限，让你可以在自己的模式中创建子程序。对于新的应用，你或许希望创建一个特殊的用户名和模式（一个特别的表空间）。

- 在可能编写 PL/SQL 程序的任意模式中的数据库对象上（例如，表和序列），直接授予你（而不是通过角色）SELECT、INSERT、UPDATE 或 DELETE 权限。一个存储子程序或包只能引用子程序的拥有者（你）有直接存取权限的对象（不是通过角色）。

- 确认已经编译适当的 Oracle 提供的包，并且对它们拥有 EXECUTE 权限。你可以在目录“<Oracle\_HOME>\rdbms<version>\admin\”下找到它们，其中 Oracle\_HOME 是 Oracle 数据库的安装目录（该名字显示在 UNIX 命令解释程序变量中或在 Windows 的注册表

中), <version>是数据库的版本。包脚本的名字有通用的形式 ‘ dbms\*.sql ’。例如, 在笔者的Windows NT机器上, dbmssql.sql位于 ‘ c:\oraNT\rdbms80\admin ’ 下。

确认有足够的存储空间可以存放存储子程序和包。这些对象在 SYSTEM表空间中产生额外的数据字典项。数据库系统管理员可以为你创建一个缺省的表空间, 用于存储你自己的表和索引的本地副本, 让你不会给产品对象添乱。

确认系统全局区 ( System Global Area, SGA ) 中的共享和私有SQL区足够大, 可以处理运行PL/SQL脚本和子程序产生的预期负载。

确认对SQL\*Plus应用或一个合适的第三方工具拥有使用权限。

一旦这些准备工作各就各位, 就准备开始开发 PL/SQL程序吧!

### 28.4.1 理解事情的模式

按抽象术语说, 模式是相关数据库对象的逻辑集合, 例如表、索引和视图。模式是开发者眼睛看到的感兴趣的数据库对象的视图, 就像实体关系图表 ( Entity-Relationship Diagram, ERD ) 上所显示的。

在一个Oracle数据库中, 模式是存储在一个或多个表空间中对象的逻辑集合。因为模式是逻辑集合, 几个应用可以共享模式, 或跨越模式的界限进入其他模式。模式对象的关系型组织严格按照数据库系统管理员和开发者看到的那样, 这正是为什么像实体关系图这样的“路标”对于任何数据库相关的软件开发如此关键。一个表空间是映射磁盘上一个或多个物理文件的逻辑存储区域。图 28-6显示了这种组织。

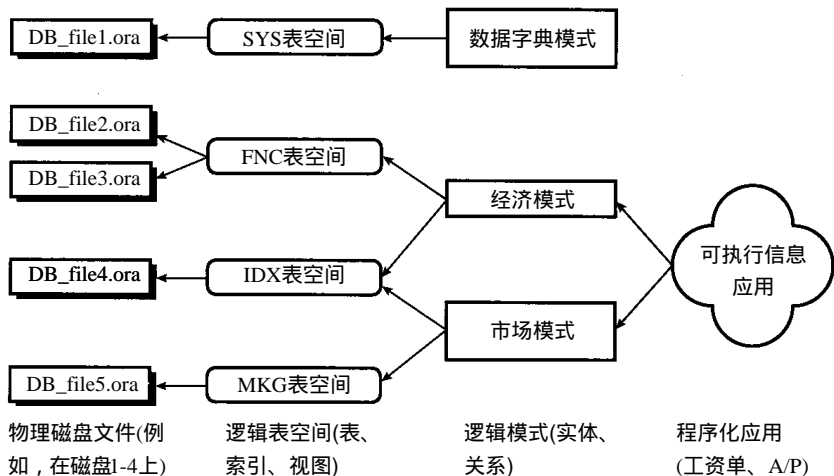


图28-6 应用、模式、表空间和数据库文件被分层组织管理

通常一个模式精确地只有一个拥有者 ( 也就是说, 一个 Oracle用户帐户 ) 负责创建、修改和删除这些对象。这个拥有者可以向其他用户授予存取权限, 既可以直接授权也可以通过使用角色授权。如果你不是这些对象的拥有者, 通过使用圆点符号指明它们在哪个表空间来限定对这些对象的指引, 就像 `SELECT CUST_NAME FROM MKG.LEADS`。

#### 管理PL/SQL代码

当PL/SQL代码在你自己的开发模式中被彻底测试和验证后, 存储 PL/SQL程序极有可能变为由模式拥有者管理的对象。在一个拥有众多开发者的复杂的开发环境中, 决定谁是源代码

的拥有者和在哪里存放源代码的问题变得十分重要。当模块被开发时，模块处于不断变动的状态。开发者可能会对一个私有副本做出重要改动，一些模块可能在后来被取消。强烈推荐使用代码管理系统（Code Management System，CMS）控制源代码的一个正式版本。当彻底测试一个模块并可用于产品时，将新版本存放在版本管理系统中，模式的拥有者可以编译这个新版本。从头开始的最快的方法是放弃追踪哪个拷贝是最好的版本。在长时间的运行中，只为不同的版本保存不同的子路径是一个很坏的编程管理方法。

在模式和表空间之间没有对应性，一个或多个模式可能存储在一个表空间中，或者也可能被分割存储于不同的表空间。实际上，将模式与文档集合存储在它自己的表空间组内是一个好主意。例如，将表存储在一个表空间中，而将索引存储在另外一个表空间中。概念上。在同一个模式中包括两个表空间。

**提示** PL/SQL的开发者应该拥有他们自己的缺省表空间，以便于在上面做试验，最好选择一个非产品数据库作为试验环境。你可以创建你自己的表和索引，并用它来做试验。这样，如果你不小心毁坏一个产品表的一个临时、本地副本，谁会在意呢？否则的话会导致你事业的意外终结（最少你会丢掉现有的工作）。另外，所有编译的存储程序和包被严格地定位在你的模式内部（与属于其他模式的子程序一起存放在数据字典中），因此你可以在一个安全的开发环境中对产品程序代码做试验性的改动。

**注意** 对模式使用的告诫是要注意对象的引用。如果对象在缺省模式中，不需要使用模式的名称来限定引用。否则，必须要限定它，以便于能够正确地找到它。可以对PL/SQL代码中没有限定的对象使用同义词，以提供限定。

出于实际考虑，一些应用可以容易地跨越模式的界限，但是核心应用可能集中在一个模式上。例如，或许已经开发出一个市场应用以使该部门自动化。后来上级管理部门要对整个企业有一个高层的观察，需要一个新应用从几个模式中获得数据。你在一个应用中完成的工作可能在其他工作中被再次使用，所以要注意让PL/SQL程序模块有足够的通用性，使它们可以在多个环境中使用。

#### 28.4.2 基本PL/SQL开发环境

使用你喜欢的文本编辑器和 Oracle提供的SQL\*Plus应用，或几个很好的第三方开发工具包，就可以开发PL/SQL程序代码。如果在UNIX环境中，启动两个会话是一个好主意——一个用来运行文本编辑器，例如vi或emacs，另外一个会话运行SQL\*Plus。如果运行的是Windows环境，可以使用记事本（或一个真正的程序员编辑器）并启动一个SQL\*Plus会话。

**警告** 在Windows 3.x环境下，在SQL\*Plus中运行任何SQL或PL/SQL程序期间，你的整个Windows环境被有效地锁定。在SQL\*Plus中执行程序之前，一定要将改动的文本保存。如果发现需要终止一个长时间执行的查询，知道没有丢失任何工作的危险，会感觉好一些。

Oracle慎重地提供了一些标准的运行库，能够执行某些重要的动作，例如标准输入/输出、计时、活动会话之间的数据交换、创建动态SQL以及其他复杂的底层操作。当程序需要增加复杂性时，可以经常使用这些包。

### 28.4.3 存取数据字典

数据字典通过不同的视图呈现给不同的用户。这些视图拥有带有不同前缀的相同名称。这些视图可以被划分为以下几类（没有特定顺序）：

常规类和对象信息。

对象的存储信息，例如表空间、段和区间。

过程、包、触发器和它们的源代码。

表、列、簇和索引。

视图、同义词和序列。

用户、角色、环境资源文件和权限。

锁和审计信息。

性能信息。

分布式数据库信息。

导出信息。

对于一个授权的数据库系统管理员用户，最适合使用的视图被前缀了 `DBA_`。这些视图提供了所有模式中每个数据库对象的有关信息，通常由模式的拥有者管理。对于所有的其他用户，可以使用 `ALL_` 视图和 `USER_` 视图。另外，性能视图（以 `V$` 开头）是公共可用的。

#### 1. 自己模式中的对象

`USER_` 视图只显示由当前连接用户所拥有的数据库对象。如果创建了它，可以发现它会在此列出。例如，视图 `USER_SOURCE` 包含存储子程序和包的源代码。

#### 2. 其他模式中的对象

`ALL_` 视图显示所拥有的数据库对象或被授权存取的其他模式中的数据库对象。在后面的案例中，可以直接授予或通过角色授予这些权限。但是，在 `PL/SQL` 代码中只能访问那些使用直接方式授权给你的对象。例如，`ALL_TABLES` 含有全部表，加上被授权存取的其他任何用户的表。

例如，视图 `ALL_SOURCE` 包含全部源代码，还有其他用户向你授予 `EXECUTE` 权限的存储包的头。这个细小差别的原因是：只需要知道如何正确地调用其他用户封装的子程序，不能以任何方式修改它们，所以不需要查看实现细节。

当执行他人拥有的子程序时，程序通常以他人的权限被执行，而不是使用你的权限来执行。但是在 `Oracle8i` 中，开发者拥有这样的选择，他可以指定子程序使用“调用者的权限”而不是“定义者的权限”来执行。

#### 3. 数据库系统管理员模式中的对象

数据库系统管理员是享有特权的，他可以存取数据字典中所有模式里的每个数据库对象。存储子程序是将这个信息展示给全体用户的合适方式，而不用给他们任何数据库系统管理员权限——当然，假设数据库系统管理员编译了这些子程序。例如，视图 `DBA_OBJECTS` 包含所有对象类型（例如表、索引、序列等等）的每个数据库对象（包括那些由 `SYSTEM` 和 `SYS` 所拥有的）的有关信息。通过编写并编译少量简单的 `PL/SQL` 存储过程，数据库系统管理员可以采用易于使用的方式，让那些需要该信息的用户使用它。

## 28.5 语言教程

学习掌握一门新语言的最佳方式是投身其中！后面的部分以指导方式介绍 `PL/SQL`，从基

础开始，很快地串讲不同的语言特性。如果熟悉任何计算机语言像 Pacal、FORTRAN、C或 COBOL，学习过程中不会有任何困难。我们强烈建议试验这里所介绍的和附带 CD中所提供的程序代码，并照此编程。

马克吐温曾经说过：“吃一堑，长一智”。在本节，听从他的建议，犯许多错误，然后从中吸引教训。要查看常见的编译和运行错误，以及服务器返回的（通常是简明的）错误信息。有意运行一些这里或那里有问题的代码，目的是让你有兴趣去调试它。这样，当独自处理问题并且遇到一个错误消息时，就不会感到无所适从。

### 28.5.1 编码规定

PL/SQL不区分大小写，所有的文本都被转换为大写形式（除了那些在文字串中的文字）。所以，你不可以使用大小写来区分变量和其他用户定义的元素。出于程序可读性原因，为了区分源代码中的元素，本节中的全部代码都被标准化，使其遵从一些简单的语法规则。

所有保留字、Oracle提供的包、内建函数、用户定义常量、Oracle和用户定义的数据类型以及模式名称都采用大写。

用户定义的游标、变量、包、Oracle和用户定义的子程序、表和列的名称都采用小写。

参数表变量以大写字母开头（例如，Pname中P代表参数）。

可以遵从这些规定或其他你乐于接受的任何规定，只是要一致。本节中的程序代码严格按照以下方式书写：带有大量的内部程序说明；仔细地设定行缩进（通常一次为两个空格，不使用TAB键）；使用空格以更清晰地对齐语句并且使每行的文本在 80个字符以内。

PL/SQL是一种自由形态的文本语言，任意数量的空白，例如空格、新行和跳格，起着分隔符的作用或被忽略。在一行上可以有一条或多条语句，一条语句也可以被拆分为几行。通常，语句以分号（；）结束。对于一行拥有的字符数量和一个模块拥有的语句行数没有限制。但是，源代码模块总的大小依赖于操作系统的限制条件。

**警告** Oracle服务器的早期版本依赖于操作系统，对 PL/SQL模块的大小有限制。在 NetWare 3.x上，模块的大小被限制在32KB以内。对于多数UNIX和Windows NT系统，模块的大小被限制在 64KB以内。违反这个限制，将导致数据库服务器或服务器机器自身崩溃。这个限制在Oracle8i中被取消了。

### 28.5.2 特殊字符

一些字符在PL/SQL程序中有特殊的含义。大多数是显而易见的，例如数学符号和关系运算符。其他的字符，例如联合与主变量指示器（Association and Host Variable Indicator），就不那么明显了。在后面的章节中将对这些字符做一探讨。表 28-1列出了这些字符（有时候是成对的）和一个简要的说明。

注意，有几种方式可以用来在 PL/SQL中表示“不等于”。出于一致性原因，本教程中从头至尾使用一种形式（!=）。以防你感到奇怪，模运算由内建函数处理。

**注意** 注释不能嵌套。单行注释后面的任何语句都被忽略。多行注意标识符之间输入的任何语句行也被忽略。

表28-1 按类型组织的特殊字符

类 型	字 符	说 明
数学运算符	+	加和一元正
	-	减和一元负
	*	乘
	/	除
	**	乘幂
关系运算符（在布尔表达式中使用）	=	相等
	<	小于
	>	大于
	<>	不等于
	!=	不等于（备用）
	~=	不等于（备用）
	^=	不等于（备用）
	<=	小于等于
	>=	大于等于
表达式和列表（在语句、数据类型声明、参数列表声明、变量和表引用中使用）	:=	赋值
	(	开始列表或子表达式
	)	结束列表或子表达式
	,	分隔列表项（在参数列表中时）
	..	范围运算符（在FOR-IN循环中使用）
		字符串连接
	=>	关联（在参数表中使用）
	;	语句结束
	%	游表属性或对象类型
	.	限定引用的成员说明符
	@	远程数据库指示符
	'	开始/结束字符串
	"	开始/结束引用标识符
	:	主变量指示器
	&	捆绑变量指示器
注意和标签	—	单行注意
	/*	开始多行注意
	*/	结束多行注意
	<<	开始标签
	>>	结束标签

### 28.5.3 PL/SQL的块结构

PL/SQL块是基本的编程结构。在块中编程有助于自顶向下、结构化的模块性和简明合理的组织。

一个未命名PL/SQL块含有三个部分：块声明、块体部分以及可选的意外处理部分：

```

DECLARE
  -- declarations
BEGIN
  -- executable code
EXCEPTION

```

```
-- exception handlers
END;
```

事实上，块声明部分也是可选的，但是，如果没有声明变量，几乎什么也做不成！所有用户定义的变量、常量、数据类型、游标、函数和过程都在声明部分定义。如果不需要进行任何定义，可以省略这个部分。

例如，考虑许多编程语言教材中常见的“世界，你好！”示例（见程序清单 28-1）。

注意 如果希望详细了解SET语句的有关信息，参阅第26章。

清单28-1 hello.sql——“世界，你好！”程序

```
SET SERVEROUTPUT ON
BEGIN
  DBMS_OUTPUT.enable;
  DBMS_OUTPUT.put_line('Hello, World!');
END;
/
```

第一行告诉SQL\*Plus将服务器所返回的写出来。

第二行和第五行提供当前块的范围。

第三行打开输出机制（这在7.3版和更高版本中是可选的）。

第四行打印字符串“Hello, World!（世界，你好！）”。

第六行执行这个未命名的基础PL/SQL块。

服务器响应如下：

```
Hello, World!
PL/SQL procedure successfully completed.
SQL>
```

在过程结束前，屏幕上不会有任何返回信息，这是因为服务器正在处理 PL/SQL 块（很有可能通过网络），只有当块操作完全结束时，才会返回信息。

直接量字符串通常使用单引号括起来，在继续学习之前，简单地谈谈直接量。

#### 1. 直接量(Literals)

直接量是固定的字符串、数字和布尔值。在运行时，它们不能够被修改，它们被限制为只读值。单字符和字符串直接量使用单引号括起指定。这里有一些字符串直接量示例：

‘ ’ ——一个空格字符。

‘ Hello, World! ’（世界，你好！）。

‘ This string contains “ embedded ” single quotes ’（这个字符串包含一个内嵌的单引号）。

‘ Double quotes are “ Okay ” too ’（双引号也可以内嵌）。

‘ 12345 ’ ——这是一个字符串，不是一个字面上的整数。

‘ 01/01/1900 ’ ——看起来像个日期。

‘ ~!@#%&\*()\_+=\|{}[];<>.,?/ ’ ——直接量可以包含特殊字符。

‘ ” ” ’ ——指定了一个包含三个字符的字符串‘ ’（单引号、空格、单引号）

字符和字符串直接量被当作 CHAR（定长字符串）类型处理。它可以被赋值给任何 CHAR 或 VARCHAR2 变量。PL/SQL 只有在字符串直接量中才大小写敏感。

数字直接量可以有任意整型值或浮点值，例如：

12 345	整型直接量
-12 345.0	浮点直接量
12345.678 90	浮点直接量可以有任意精度
100.	这是一个浮点直接量，精度为 0
1.2 345E2	可以使用科学计数法
1.2 345E-2	
0.12345或.12345	开头的0是可选的

声明基本常数包含程序中用到的直接量值是一个良好的编程习惯。可以给它们有意义的名字。另外，它们更具有可维护性，如果必须修改一个直接量数值，只需要在一个地方做改动即可。这项技术避免了不可维护的“幻数”综合症，这种症状的表现是在某个地方使用了一个特定数值，但没有任何人记得为什么用它。程序清单 28-2显示了一个例子。

清单28-2 circle.sql——使用常数使代码更易于维护

```

DECLARE
  PI CONSTANT REAL := 3.14159265359;
  circumference REAL;
  area REAL;
  radius REAL := &Radius;
BEGIN
  circumference := PI * radius * 2.0;
  area := PI * radius**2;
  DBMS_OUTPUT.put_line('Radius = ' || TO_CHAR(radius) ||
                        ', Circumference = ' || TO_CHAR(circumference) ||
                        ', Area = ' || TO_CHAR(area));
END;
```

当运行这个程序时，SQL\*Plus首先提示你输入使用“与”符号（&，第五行）指定的捆绑变量的数值。这里是输入数值后所发生的结果：

```

Enter value for radius: 3.5
old 5: radius REAL := &Radius;
new 5: radius REAL := 3.5;
Radius = 3.5, Circumference = 21.99114857513, Area = 38.4845100064775
PL/SQL procedure successfully completed.
```

注意捆绑变量不是一个PL/SQL变量，它只是你必须提供数值的占位符。并且，如果你从第一个例子开始就遵照指示去做的话，不需要再次启动服务器的输出机制或再次激活输出机制。一旦这些事物被启动，它们在会话期间始终存在（除非有意地关闭它们或重新进行连接）。

注意常数变量PI在它的声明期间是如何被指定和接受数值的。你还可以使用关键词DEFAULT代替赋值操作符“:=”，它在声明的环境中有相同的基本功能。

还支持布尔直接量TRUE和FALSE（注意它们没有被括在单引号中），它们可用于任意布尔表达式中和赋值给任意布尔型变量。

表28-2中的真值表显示了关系运算符AND、OR和XOR（异或）的所有二元布尔组合，以及一元否定（NOT或~）的TRUE、FALSE和NULL值。PL/SQL不支持关系运算符XOR，但是，PL/SQL的内建函数XOR（）执行这个操作。

程序清单28-3是一个相关的例子。

表28-2 AND、OR、XOR、NOT三逻辑真值表

p	q	p AND q	p OR q	P XOR q	NOT P
TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE	FALSE
TRUE	NULL	NULL	TRUE	NULL	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	NULL	FALSE	NULL	NULL	TRUE
NULL	TRUE	NULL	TRUE	NULL	NULL
NULL	FALSE	FALSE	NULL	NULL	NULL
NULL	NULL	NULL	NULL	NULL	NULL

清单28-3 booly.sql：使用布尔型数据和运算

```

DECLARE
  T CONSTANT BOOLEAN NOT NULL := TRUE; -- must have a value
  x BOOLEAN;
  y BOOLEAN;
BEGIN -- illustrate Booleans
  x := 1 = 2; -- this expression evaluates to false
  y := XOR(T, x); -- and this evaluates to true
  IF (x) THEN -- test for true fundamental
    DBMS_OUTPUT.put_line('x = TRUE');
  ELSIF (NOT x) THEN -- test for false
    DBMS_OUTPUT.put_line('x = FALSE');
  ELSE -- x must be null
    DBMS_OUTPUT.put_line('x is NULL');
  END IF;
  IF (y) THEN -- test for true
    DBMS_OUTPUT.put_line('y = TRUE');
  ELSIF (NOT y) THEN -- test for false
    DBMS_OUTPUT.put_line('y = FALSE');
  ELSE -- y must be null
    DBMS_OUTPUT.put_line('y is NULL');
    DBMS_OUTPUT.put_line('Booly for you!');
  END IF;
END;
/

```

服务器响应下列信息：

```

x = FALSE
y = TRUE
PL/SQL procedure successfully completed.

```

没有函数可以将一个布尔数值转换为字符串或将字符串转换为布尔值，你可在稍后编写这样的函数。注意程序的第二行，如何指定一个变量不能为空。这限制了程序中的取值，但PL/SQL支持该语法。注意，对于声明不能为空的变量，系统处理时会有一些额外开销。

使用值T和x进行试验，返回该块查看得到了什么（SQL\*Plus 的EDIT命令可以编辑文本并返回该文本）。如果忽略缺省赋值，将得到最基本的结果，如下所示：

```

ERROR at line 1:
ORA-06550: line 2, column 3:
PLS-00322: declaration of a constant 'T' must contain an initialization
assignment
ORA-06550: line 2, column 5:
PL/SQL: Item ignored

```

注意 参阅第26章了解EDIT命令的详细信息。

你遇到了第一个编译错误。在第一行上的错误告诉你整个未命名块有一个错误。这是因为整个块是作为一个事务提交给服务器的。下一个信息非常有用，告诉你忘记了初始化常量值。事实上，所有的常量必须被初始化，即使它为空，只要没有 NOT NULL（不为空）子句限制，它也肯定是合法的。ORA-06550错误代码表明Oracle服务器遇到了某种类型的PL/SQL编译错误（留给PL/SQL编译器解释实际的错误）。

如果删除CONSTANT（常量）和初始化值，但留下NOT NULL（不为空）子句，将得到下列信息：

```
ERROR at line 1:
ORA-06550: line 2, column 5:
PLS-00218: a variable declared NOT NULL must have an initialization assignment
```

在运行时，当块被输入时，一个变量被初始化为指定的值。如果没有规定的赋值，那么缺省情况下该变量被赋予一个NULL（空）值。试一试除去CONSTANT、NOT NULL和初始化，将得到下面的结果：

```
x = FALSE
y is NULL
Booly for you!
```

变量在运行时接受数值，并不是在编译时接受数值，因为像其他语言中的自动（或堆栈）变量一样，除非块在作用域中，否则它们并不真正存在。这意味着当块在作用域外时，它们被释放，并且在块再次进入时被重新分配。要注意多次调用过程和函数所造成的系统额外开销。事实上，要充分利用它。因为变量缺省地被初始化为NULL（空），如果它们确实需要一个初始非空值，这是一个做这件工作的好地方。

当一个变量被声明为CONSTANT（常量）时，它的值在运行时不能改变。试图改变常量的值将会产生一个例外。还有，初始化直接量的类型和长度必须符合变量的类型和长度。例如，如果一个变量x的声明为CHAR(4) := "NO GOOD!"，在运行时（不是编译时）将会产生一个例外。如前所见，常量必须在它们的声明中赋值，即使需要的值是NULL，否则，会发生编译错误。

一个变量可以被声明为NOT NULL（非空），以指明它不能接受一个NULL（空）值。这样的变量在声明时必须被初始化为某个非空值。注意CONSTANT（常量）和NOT NULL（不为空）关键词的正确位置，它们必须被分别地放在数据类型的前后。

这里是一些合法的和非法的初始化语句示例：

```
DECLARE
  price          NUMBER(5,2) := 19.92;    -- valid initialization
  discount       NUMBER(3,3) := .0625;    -- gets rounded to .063
  max_quantity   INTEGER(4) := 50000;     -- runtime error!
  max_discount   CONSTANT REAL := 0.75;   -- valid
  min_discount   CONSTANT REAL;           -- compile error! not initialized
  disc_type      VARCHAR2(1) := NULL;     -- valid and redundant
  disc_name      CONSTANT CHAR(20) := NULL; -- valid but of dubious value
  quantity       INTEGER NOT NULL := 0;   -- correct usage
  item_name      VARCHAR2(30) DEFAULT 'Hammer'; -- alternate assignment
```

## 2. 分支

在考虑程序分支时，首先让我们列举IF/THEN语句支持的全部语法结构类型：

1) 使用条件（IF/THEN/ELSE）逻辑 基本的条件测试与分支逻辑具有以下形式：

IF (某些条件为真) THEN	——测试条件
.....	——条件为真, 做这项工作
ELSE	——条件为假
.....	——做这项工作
END IF ;	——测试结束

就记录来说, IF...THEN后面的第一个语句块称为前项, 跟在 ELSE后面的块称为后项。可以在 THEN、ELSE和ELSIF块中拥有多条语句。如果愿意, 可以在前项或后项内部放置 BEGIN...END语句。当然, 后项是可选的。注意, 块以 END IF结束。在条件语句的开始和结尾处进行注意是一个好习惯, 尤其对于一个长而重要的语句, 可以知道它是用来做什么的。

IF (某些条件为真) THEN	——检验某些条件
BEGIN	
...	
END ;	
ELSE	——条件为假
DECLARE	——定义一些局部变量
X NUMBER ;	
BEGIN	
...	
END ;	
END IF ;	——检验某些条件

还会创建前提中的变量 X, 除非执行了结果。可以使用这种方式来限制局部变量只用于需要它们的块中, 这可以节省小量内存。

如何决定语句的缩进取决于你, 但要注意将一个块缩进到另外一个块内部。

可以嵌套语句到任意深度, 可以使用这种方式执行一些非常复杂的逻辑操作。

使用简化求值计算条件表达式。这意味着, 如果一个复杂 OR表达式的第一部分值为 TRUE (真), 或者一个 AND表达式的第一部分值为 FALSE (假), 求值立即停止, 没有再进一步求值的必要了。大多数其他语言同样支持简化求值。这种表达式的计算步骤总是从左到右, 圆括号内的表达式有较高优先级。

2) 测试 NULL (为空) 条件 特殊的直接量值 NULL可以赋值给任何数据类型的变量。它代表一个未知数值。你必须使用特殊的语法 IS NULL和IS NOT NULL分别检测它的存在或不存在。下面的测试

```
IF (x=NULL) THEN ...和
IF (x= ! NULL) THEN ...

总会出错, 正确的方式是
IF (x IS NULL) THEN ...和
IF (x IS NOT NULL) THEN ...
```

3) 使用连续的 IF逻辑 有时需要一个接着一个地测试一系列的数值。如果前面的一条语句已经完全成功, 后面的语句预期会失败, 那么编写一组 IF/THEN语句效率会很差, 如下所示:

```
IF (val = '0') THEN -- is value a digit?
...
END IF; -- zero?
IF (val = '1') THEN -- maybe a 1?
...
END IF; -- 1?
```

```
...
IF (val = '9') THEN -- maybe a 9?
```

```
...
END IF; -- end of test
```

可以使用嵌套IF/ELSE语句，但这样做会使语句很难看，它使全部语句都缩进。可以使用ELSIF语句代替上面的语句，让代码更有效率。只要其中的一个条件为真，它的前项被执行并且退出整个块。

```
IF (val = '0') THEN -- is value a digit?
```

```
...
ELSIF (val = '1') THEN
```

```
...
ELSIF (val = '2') THEN
```

```
...
ELSIF (val = '9') THEN
```

```
...
ELSE -- not a number
```

```
...
END IF; -- end of test
```

提示 PL/SQL语句没有SWITCH语句（像在C语言中一样），没有CASE语句（像在PASCAL语言一样），也没有可计算的GOTO语句（像FORTRAN语言）。PL/SQL使用连续的IF模拟这些语句结构。

清单28-4显示了一个简单的例子，编写该程序用于估算作者 1996年的基础联邦税款债务（来自美国国税局公告15，通知E，雇主税款指南，修订版本，1997年1月）。

清单28-4 fedtax.sql——用来检测取值范围的连续的IF逻辑

```
SET SERVEROUTPUT ON;
-- assumes annual payroll, married
DECLARE
num_wh NUMBER := &num_withholding;
wh_amount NUMBER;
gross NUMBER := &annual_gross_salary;
liab NUMBER;
adj_gross NUMBER;
BEGIN
  DBMS_OUTPUT.ENABLE;
  wh_amount := num_wh * 2550; -- annual allowance
  adj_gross := gross - wh_amount;
  IF adj_gross <= 6425 THEN
    liab := 0;
  ELSIF adj_gross <= 44250 THEN
    liab := (adj_gross - 6425) * 0.15;
  ELSIF adj_gross <= 89675 THEN
    liab := 5673.75 + (adj_gross - 44250) * 0.28;
  ELSIF adj_gross <= 151850 THEN
    liab := 18392.75 + (adj_gross - 89675) * 0.31;
  ELSIF adj_gross <= 267900 THEN
    liab := 37667 + (adj_gross - 151850) * 0.36;
  ELSE -- over max
    liab := 79445 + (adj_gross - 267900) * 0.396;
  END IF; -- test adjusted gross
  DBMS_OUTPUT.PUT_LINE('FEDERAL TAX LIABILITY: ' || TO_CHAR(liab));
END;
/
```

当运行这个函数时，你被提示输入捆绑变量值，并且服务器返回下列主要信息：

```
Enter value for num_withholding: 4
```

```
old 2: num_wh NUMBER := &num_withholding;
new 2: num_wh NUMBER := 4;
Enter value for annual_gross_salary: 60000
old 4: gross NUMBER := &annual_gross_salary;
new 4: gross NUMBER := 60000;
FEDERAL TAX LIABILITY: 7227.75
PL/SQL procedure successfully completed.
```

被测试表达式的括号是可选的，但是推荐使用它们，对于一个复杂的表达式更要使用括号。

4) 非条件分支 PL/SQL使用GOTO语句支持非条件程序分支。可以使用一个标签指定一个跳转的目标点，就像 BASIC和汇编语言中的一样。可以跳转到当前块的任意地方或跳转到封装块中。通常，不赞成使用 GOTO语句，因为它导致非结构化的程序代码。这样的程序通常很快会变得难以维护。GOTO语句的语法如下面的程序清单 28-5所示。

清单28-5 okgoto.sql——使用GOTO

```
BEGIN
  BEGIN
    GOTO MID;  -- forward reference OK because it's in scope
  END;
  <<MID>>
  NULL;
END;
/
```

标签MID使用双角括号括起来表示。注意标签没有使用分号结束。

这个SQL脚本可以成功地编译。但是，程序清单 28-6显示了一种不同的主要情形。

清单28-6 badgoto.sql——不合理地使用GOTO语句

```
BEGIN
  GOTO MID;  -- invalid!
  BEGIN
    <<MID>>
    NULL;
  END;
END;
/
```

因为标签在范围之外，将得到一个编译错误：

```
ERROR at line 1:
ORA-06550: line 2, column 8:
PLS-00201: identifier 'MID' must be declared
ORA-06550: line 2, column 3:
PL/SQL: Statement ignored
```

外部的块不知道内部块中存在的任何东西。

注意NULL（空）语句。这确实是一个合法的语句，仅仅作为占位符使用。不能够在任意形式的END语句前面（像在END IF中）立即拥有一个标签，否则将得到一个编译错误。可以使用一条NULL语句解决这个问题。

不可以从一个块中跳转到封装块中同一层次上的其他块中，如程序清单 28-7所示。

这些限制还适用于条件语句。原则上，不能从前项跳转到后项（见程序清单 28-8）。

这时Oracle检测出你做了某些超乎常理之外的事情，并且给出下列信息：

```
ERROR at line 3:
ORA-06550: line 3, column 5:
PLS-00375: illegal GOTO statement; this GOTO cannot branch to label 'CONSEQUENT'
ORA-06550: line 4, column 3:
PL/SQL: Statement ignored
```

清单28-7 bad2goto.sql——不能在一个封闭块内部同一层次的块之间使用 GOTO语句

```
BEGIN
  BEGIN
    GOTO OTHER; -- invalid!
  END;
  BEGIN
    <<OTHER>> -- out of scope
    NULL;
  END;
END;
/
```

清单28-8 bad3goto.sql——不可以在前项与后项间跳转

```
BEGIN
  IF (TRUE) THEN -- always do the antecedent
    GOTO CONSEQUENT;
  ELSE -- consequent
    <<CONSEQUENT>>
    NULL;
  END IF; -- of jump example
END;
/
```

还有，Oracle不检查无限循环，如下所示：

```
BEGIN
  <<INFINITE_LOOP>>
  GOTO INFINITE_LOOP;
END;
```

**警告** 少量程序代码就可以使SQL\*Plus会话陷于死循环的混乱状态！如果愚蠢到做出这样的事情，将不得不请求数据库系统管理员终止该会话，尤其在Windows 3.1环境下，系统将会被锁起，直到该会话被终止。如果正在运行Windows 95或Windows NT，可以放弃这个SQL\*Plus会话，丢失机器上用于基本处理的一些资源（例如为该会话分配的内存和它使用的可用句柄）。

5) 标签和作用域规则 变量声明的作用域对于封装块是局部的。变量通常只能在块的范围内被引用，或在定义它的子块中被引用。子程序参数表中的变量声明只能够在那个子程序中被引用。另外，子程序体中的任何变量声明都被限制为子程序的局部变量。一个内部块可以重新命名一个变量，对于外部区域中的变量，它仍然是一个新的、不同的变量。

在运行时，当执行到块作用范围以外时，它的变量、游标以及相似的局部结构都被释放。当该块再次进入作用范围，变量被再次创建并被初始化为在声明中指定的任意值。你将在后面存储子程序部分对此进行仔细观察。

当决定哪里是放置声明的最佳位置时，通常使用的经验规则是缩小变量的声明范围，只作用到那些需要引用它的块。因此，只有当实际需要时，该对象才会存在，节省了内存。为配合节省内存，分配对象所要求的对象保留需要和节约时间的探索成为第一位的。

通过使用标签，一个内部块可以引用包含块中的对象。这是一种坏的编程习惯，应该使用一个不同的变量名或使用向子程序传递堆栈中的变量来替代它。但是，PL/SQL语言确实支持标签，所以如果感觉必须使用这个特性，遵照以下步骤：

- 在外部块的开始处插入一个标签。
- 在内部块中使用点符号引用由外部块定义的变量。
- 标签是一个限定词，需要用来引用与内部变量同名的外部变量。

程序清单 28-9 显示了正确的和错误的限定基本标签的用法。

清单 28-9 badlabel.sql——使用与误用标签来区分块

```
<<OUTER_BLOCK>>    -- this label names the outer block
DECLARE
  x NUMBER; -- available to both inner and next blocks
  y NUMBER;
BEGIN
  <<INNER_BLOCK>>    -- might as well label the inner one too
  DECLARE
    x NUMBER; -- only available to this block
  BEGIN -- inner block
    x := OUTER_BLOCK.x; -- qualified reference to outer block variable
    y := 0; -- an unqualified reference to outer block variable

  END INNER_BLOCK;
  <<NEXT_BLOCK>>
  DECLARE
    x NUMBER;
  BEGIN -- next block
    x := INNER_BLOCK.x; -- THIS IS ILLEGAL!!
    x := OUTER_BLOCK.x; -- this is OK
  END NEXT_BLOCK;
  x := INNER_BLOCK.x; -- THIS IS ALSO ILLEGAL!!
END OUTER_BLOCK;
```

将得到下面的错误信息：

```
ERROR at line 1:
ORA-06550: line 17, column 22:
PLS-00219: label 'INNER_BLOCK' reference is out of scope
ORA-06550: line 17, column 5:
PL/SQL: Statement ignored
ORA-06550: line 20, column 20:
PLS-00219: label 'INNER_BLOCK' reference is out of scope
ORA-06550: line 20, column 3:
PL/SQL: Statement ignored
```

注意，不能够反向操作，即，不能在外部块中引用内部块的变量。为什么不能？因为它不存在。内部块中的变量没有被创建，直到你真正地进入内部块时，它们才被创建，并且在程序退出内部块时，它们立即被释放，所以外部块没有机会引用内部块的变量。

同样，一个子程序中嵌套的子程序使用带有点符号的子程序的名字来引用封装子程序层定义的一个变量。通过参数表传递需要的值，或简单地在内部块中不再使用相同的变量名是一个更好的编程习惯。非正规的或特殊的东西涉及得越少，代码维护起来就越容易。

可以拥有嵌套块而不标注它们。甚至可以在子块中创建新的变量。当执行一个可能会产生例外的操作，并且需要在封装块内部定位例外处理程序时嵌套块非常有用。

你已经看到了一些标识符——变量、数据类型、包和过程。现在注意有关它们命名的一些常用规则。

### 3. 标识符

标识符是代表编程项目的词素，例如变量、常量、用户定义数据类型、游标、例外、数据库表、列、过程、函数和包。一个标识符必须以字母开头，最多可以有 30 个字母、数字、下划线、美元符号（\$）或磅字符（#）。

合法标识符	非法标识符
PI	3D-array
length_of_string	function_header
char\$	var name
RQ\$Get\$Segment	area/height
local##time	\$HOME

标识符不能含有内嵌的空格、连字符或斜线，但是可以使用双引号括起标识符，让它们区别于其他的词素，例如当一个表的列名与 PL/SQL 的保留字相同时。EXCEPTION 是一个 PL/SQL 的保留字，然而它不是 SQL 的保留字，所以可以被用作列名。正确引用这样一个列的基本命令是使用双引号，如下方式表示：

```
SELECT "EXCEPTION" INTO exc FROM prog_errors;
```

因为列以大写形式存储，带引号的标识符必须是大写的。还可以使用双引号来包括空格或其他分隔符，例如：

```
SELECT cr_msg "*** Credit Message ***" FROM messages;
```

在 SQL\*Plus 中这更典型地用作列标题。

### 4. 注释代码

好的程序员要为他们的程序代码添加大量的注意。当编写代码时，他们拥有的时间越多，他们的注意就会越好。当首次设计一个复杂的 PL/SQL 程序时，可以使用注释来指明不同的部分。事实上，可以经常从程序的说明中直接获取注意，然后可以回到每个部分并使用真正的代码充实它。例如，使用含有某些基本信息的模块头注释，然后根据作用范围和类型组织声明，如下所示：

```
/*
  Program Name: c2f
  Module Name : c2f.sql
  Written By  : Daniel J.. Clamage
  Description : .This module converts Celsius to Fahrenheit.
  Modification: V.001 04-OCT-1997 - djc - Initial release.
*/
...
-- public global constant variables (shared between modules)
prog_version CONSTANT VARCHAR2 (25) := 'V.001 04-OCT-1997 - djc';
prog_name CONSTANT VARCHAR2(30) := 'c2f';
...
-- public global record types needed
...
-- public global variables
...
-- public global cursors needed
...
```

当编译一个保存的带有多行注释的基本脚本时，SQL\*Plus 将它们作为一个 DOC（文档）块回显：

```
SQL> @d:\plsql\library\c2f
DOC> Program Name: c2f
...
DOC> Modification: V.001 04-OCT-1997 - djc - Initial release.
DOC>*/
```

当编译一个完整的模块集，并且将结果脱机存储到一个文件时，这非常方便。我们建议在一个模块的注释头块中限制使用多行注释，因为在编译的时候它们会被回显，引起一些混乱。如果第一个多行注释分隔符没有放在它自己的行上，那么第一条注释行就不显示。原则上，可以在任意地方使用单行注释。

#### 28.5.4 声明变量

通常有两种类型的变量：标量的和复合的。标量变量只允许一个值。复合变量可以包含多个相同的或单一的数据类型的值，取决于复合变量的种类。一条语句只能声明一个变量。

表28-3显示了PL/SQL支持的标量数据类型和它们对应的数据库类型（如果有的话）的列表。

表28-3 标量数据类型（+代表7.2版； 代表7.3版；#代表8.0.3版）

类 型	子 类	说 明	取值限制	数据库限制
CHAR(s)	CHARACTER STRING ROWID* NCHAR***	定长字符串，** 接受NLS数据	0~32 767字节， 长度可选，缺省为 1	255字节，*接受数 据库ROWID 字符串，**2 000字节
VARCHAR2(s)	VARCHAR STRING ± NVARCHAR2#**	变长字符串， *接受NLS数据	0~32 767字节， 长度可选，缺省为 1	2 000 字节 #4 000 字节
NUMBER (p, s)	NUMERIC DEC DECIMAL INT* INTEGER * FLOAT * REAL DOUBLE PRECISION SMALLINT-*	压缩十进制值 p = 精度(共#位数字)s=范围 (这里进行四舍五入)， *不能指定范围	取值范围是 1.0E <sup>-129</sup> ~9.99E <sup>125</sup> 。 精度为1~38(缺省长度 是系统支持的最大值)。 比例是-84~127（缺省 为0）	与PL/SQLNUMBER 长度相同
BINARY INTEGER	NATURAL * POSITIVE** NATURALN - POSITIVEN - SIGNTYPE#***	有符号二进制数，*从 0开始，**从1开始，预 定义非空；与上面具有 相同的范围	取值范围是-2 <sup>31</sup> -1到2 <sup>31</sup> -1； 或 ±2 147 483 647，***限 制为(-1, 0, 1)用于实现 三态逻辑	不存在相应的数据库 类型(使用NUMBER)
PLS INTEGER^		快速机器运算的有符号 二进制数字,比NUMBER 占用的存储少	取值范围是-2 <sup>31</sup> -1到2 <sup>31</sup> -1； 或 ±2 147 483 647，	不存在相应的数据 库类型(使用NUMBER)
DATE		一个内部日期值	公元前4712年1月1日 —公元后4712年12月31 日；以秒表示的时间从 午夜开始计时（缺省的 时间是午夜12 00）	与PL/SQL日期相同
BOOLEAN		布尔（逻辑）值	TRUE、FALSE、NULL (未知值)	不存在数据库类型
LONG		变长字符串	0~32 767字节（长度未 规定）	0~2 147 483 647字 节

(续)

类 型	子 类	说 明	取值限制	数据库限制
RAW(s)	LONG RAW	非解释二进制数据 (与字符集无关)	RAW: 0~32 767字节; LONG RAW: 0~32 767 字节(与LONG相同) (要求长度至少为1)	RAW是2000字节, LONG RAW是2 147 483 647字节
REF CURSOR ±		游标变量(游标的指针)		不存在数据库类型
LOB	BFILE *	存储在数据库内部或外	2 <sup>32</sup> -1字节或0~4GB, *只	与PL/SQL LOB相同
(巨型对象)	BLOB**	部操作系统文件中的巨	有外部文件(只读), **内	
	CLOB***	型、无结构的或二进制	部/外部二进制数据(读/写),	
	NCLOB****	数据(图像、声音、视	***内部/外部单字节数据	
		频、文档)	****内部/外部固定宽度多	
			字节数据(读/写)	

子类型通常与它们的对应主体有相同的取值限制。Oracle在PL/SQL中提供它们是为了Oracle与其他数据库中的列数据类型的兼容性。例如,当你希望一些事物更具描述性时,可以使用可选的NUMBER子类。如果一个类型使用了一个可选的长度参数,它被如上所示放在括号内。只能用整型直接量指定长度参数(例如 CHAR(3))。对于字符串,一个长度为0的串是NULL(空)值(未知值)。

字符串被当作标量对待。为了能够存取单个字符,可以使用Oracle提供的字符函数SUBSTR()、INSTR()等等。一个CHAR类型数据的长度使用字节表示,而不是使用字符表示。这意味着多字节字符集将需要更多的存储空间。注意从7.1版到7.2版STRING子类处理上的变化。

为了完成数学运算,Oracle执行从一个NUMBER(压缩十进制)数据类型到一个内部浮点表示的转换。相反,BINARY\_INTEGER和PLS\_INTEGER数据类型已经具有内部格式。使用BINARY\_INTEGER数据类型的索引编写的循环程序,当值限制在一定范围内时,执行速度较快,因为不需要转换过程。PLS\_INTEGER执行速度甚至比BINARY\_INTEGER更快,因为它使用本地机器运算,而BINARY\_INTEGER和NUMBER依赖于数学库程序。但是,BINARY\_INTEGER值可以被赋值给NUMBER变量以避免溢出例外,而使用PLS\_INTEGER则不会消除溢出例外。不需要过多关心这个细小的语义差别。

负数四舍五入到小数点的左边;例如,-3四舍五入到最接近的千位(1000,2000,3000.....),而+3四舍五入到最接近的千分位(0.001,0.002,0.003.....)。

观察这个表,也许会意识到当与数据库中的列交互时,LONG和RAW的PL/SQL类型拥有有限的作用。PL/SQL的LONG RAW数据类型实际上比RAW长度要短!他们应该称它为RAW LONG。但是,有一个Oracle提供的程序可以将一个数据库的LONG值转换为PL/SQL的LONG变量。

这个表中其他要注意的事情是数据库数据类型的限制和对应的PL/SQL数据类型可能会有明显的不同(特别是字符类型)。在编程时请记住这点。可能遇到的取值错误与这个差异有关(当不小心试图将一个较长的字符串赋值给一个较短的变量时,会出现错误)。

#### 1. 使用PL/SQL数据类型

一些标量变量声明的例子如下所示:

item\_name CHAR(32); ——也许尾部使用空格填补。

item\_category VARCHAR2(32); ——这样的字符串不需要填以空格。

price NUMBER ( 5 , 2 ) ; ——可以接受  $\pm 999.99$  之间的任意数字。

quantity INTEGER(3) ; ——可以接受  $\pm 999$  之间的任意数字。

discountable BOOLEAN ; ——可以是 TRUE、FALSE 或 NULL。

discount REAL ( 3 , 3 ) ; ——在  $\pm 0.999$  之间的任意实数。

run\_date DATE ; ——日期域占用 7 个字节。

当然，任意数据类型的变量可以接受一个 NULL ( 空 ) 值 ( 而不仅仅是布尔类型变量才可以 )。

一个真正灵活的特性称作 “ 配置 ” 变量。不需要确实地编写一个数据类型，可以使用其他变量、数据库列或表的数据类型。可以使用 %TYPE 和 %ROWTYPE 属性实现这个目的。%TYPE 属性提供了所需要的数据库列或变量的类型及长度。%ROWTYPE 属性允许人们定义一个记录变量，它的成员变量拥有表或游标中每一列正确的类型及长度。使用点符号引用记录中的每个成员变量。我自己的惯例是使用带有 ‘ \_rec ’ 后缀的 %ROWTYPE 变量以指明该变量是一个记录变量。例如：

quantity orders.qty%TYPE; ——基于表订单中的 qty 列。

discount orders.discount%TYPE; ——基于表订单中的 discount 列。

orders\_rec orders %ROWTYPE; ——基于表订单中的一条记录。

这里是更多的使用类型的示例：

```
DECLARE -- define user variables
MAX_INT CONSTANT INTEGER := +2147483647;
MAX_STR VARCHAR2(32767); -- let's use this for typing only
notes MAX_STR%TYPE; -- based on another variable's type
items_rec items%ROWTYPE; -- based on a table in the database
iname items.item_name%TYPE; -- based on a column in a table
last_id MAX_INT%TYPE; -- this must be a constant too
BEGIN -- executable code
iname := 'HAMMER';
items_rec.item_name := iname; -- compatible data types
last_id := 0; -- Wait! Aagggh! Runtime error!
```

这提供了非常好的方法，控制解决了当数据库列或表改变时所产生的数据类型问题。因为变量的类型是基本类型，不是显式声明的，正确的数据类型和长度已经就绪。如果表发生改变，只需要重新编译代码即可。使用这种方法，基本变量提供了一种数据独立和降低代码维护的手段。想像一下，当一些列的长度或类型改变时，如果代码中使用显式的数据类型，需要做多少工作来修改多个模块中的代码。当然，如果表列的数据类型从 CHAR ( 5 ) 改变为 NUMBER ( 5 )，你还要对你的 PL/SQL 代码重新加工 ( 我已经遇到过这样的事情了 )，例如，要插入 TO\_CHAR ( ) 函数转换。

然而，当使用 %TYPE 时，会卸去所有的负担。在前面的例子中，可以看到程序员错误地试图给一个基于常量的变量赋予一个新值，这样也会使基本变量变为一个常量。

尤其应该在子程序参数表和函数返回值声明中使用基本变量，那里变量的类型被声明，但长度未被声明。

## 2. 定义复合数据类型

PL/SQL 支持两种形式的复合 ( 一个矢量或数值集 ) 数据类型： RECORD ( 记录 ) 和 TABLE ( 表 )。复合变量可以基于一个表或游标的行类型创建，如上所示。这和 RECORD 变量有相似的语义。

要声明一个新的复合数据类型，要使用关键字 TYPE。TYPE声明只声明了一个新的数据类型，它们并没有定义存储空间。为了能够使用它们，必须声明变量为哪种类型。在这里，使用新数据类型名后缀\_TYPE的约定以指明该变量被用作一个数据类型：

```
DECLARE
  -- user-defined data types
  TYPE MY_STRING_TYPE IS RECORD OF (
    str_len INTEGER := 0, -- initialized when variable is declared
    str VARCHAR2(32767)); -- defaults to NULL
  TYPE MY_ARRAY_TYPE IS TABLE OF CHAR(8) INDEX BY BINARY_INTEGER;
  -- variables
  str MY_STRING_TYPE; -- a new composite variable
  arr MY_ARRAY_TYPE; -- a new 1-D array
```

### 3. 创建自己的记录类型

RECORD类型定义了一个结构，该结构可以包含任意数目的任意数据类型的成员变量，包括前面定义的RECORD或TABLE类型。与一些表%ROWTYPE的记录变量一样，使用圆点符号引用每个成员。在运行时对基本变量的成员进行初始化定义。

可以嵌套RECORD类型：

```
DECLARE
  TYPE ZIP_TYPE IS RECORD (
    zip5 VARCHAR(5),
    DASH CONSTANT VARCHAR2(1) := '-', -- for display purposes perhaps
    plus4 VARCHAR2(4) := '0000'); -- initialized at runtime
  TYPE ADDR_TYPE IS RECORD (
    line1 VARCHAR2(30),
    line2 VARCHAR2(30),
    city VARCHAR2(20),
    state VARCHAR2(2),
    zip_code ZIP_TYPE);
  TYPE EMPLOYEE_TYPE IS RECORD (
    ssn VARCHAR2(9),
    dob DATE,
    address ADDR_TYPE);
  employee_rec EMPLOYEE_TYPE; -- actual storage defined
BEGIN
  employee_rec.ssn := '123456789'; -- someone's SSN#
  employee_rec.address.city := 'Pittsburgh';
  employee_rec.address.zip_code.zip5 := '15210'; -- Pittsburgh's zip
  employee_rec.address.zip_code.plus4 := '3702'; -- Mt. Oliver in Pgh
```

注意引用该嵌套记录任一子组件的圆点符号的用法。这种形式的主要优点是可以更为容易地管理大量不同数据类型的相关数据。使用一个参数，就能将一个数据集传递给子程序或将数据集从子程序返回。它比其他方式更加整齐和紧凑。记录使开发者按照紧密相关的数据块来考虑问题。

只要声明为具有相同的数据类型，就可以将一个记录变量赋值给其他的记录变量。基于数据库表的ROWTYPE变量和那些定义为记录类型的变量通常是不能兼容的，即使它们的成员完全匹配：

```
DECLARE
  TYPE ORDERS_TYPE IS RECORD ( -- looks like the database table
    ord_num orders.ord_num%TYPE,
    quantity orders.quantity%TYPE,
    ...
    discount orders.discount%TYPE);
  orders_rec orders%ROWTYPE; -- based on a database table
  new_ord_rec ORDERS_TYPE;
```

```
old_orcs_rec ORDERS_TYPE;
BEGIN
... -- do some work
old_orcs_rec := new_orcs_rec; -- this is correct
new_orcs_rec := orders_rec; -- this is incorrect!
```

当向数据库表中插入记录时，VALUES子句必须分别指明每个成员变量：

```
INSERT INTO orders orders_rec; -- WRONG!
INSERT INTO orders old_orcs_rec; -- WRONG!
INSERT INTO orders
(ord_num, qty, ..., discount) VALUES
(new_orcs_rec.ord_num, new_orcs_rec.quantity, ...,
new_orcs_rec.discount); -- CORRECT!
```

#### 4. 创建自己的数组

PL/SQL表是任何单独的标量类型数据的一维数组。对于 7.3版之前的 Oracle服务器，TABLE可以含有一个 RECORD或另外一个 TABLE。对于 Oracle 7.3 版和更高版本，一个 TABLE可以包含一个用户定义的记录，而不能包含其他的 TABLE。在 Oracle8i中，TABLE就是几种集合的类型。

PL/SQL的TABLE不同于Oracle数据库的表。它是一个无界的数组，在被赋值以前，它的元素是不存在的。它总是使用BINARY\_INTEGER索引，给予它的元素的索引范围是  $\pm 2\,147\,483\,647$  (还好，不是完全无界)。程序员可以选择数组索引的开始偏移量为 0、1或其他任何有意义的位置。因为一个未赋值的元素并不存在，数组可以稀疏地包含定位的数值，不需要额外内存的使用。每个元素使用圆括号中的偏移量来引用。首先声明一个表类型，然后基于该类型创建一个变量，就可以创建一个 PL/SQL表变量了。例如，下面的代码只使用足够的内存来存储这些值：

```
DECLARE
TYPE STR_TYPE IS TABLE OF VARCHAR2(8)
INDEX BY BINARY_INTEGER;
arr STR_TYPE;
i BINARY_INTEGER := 17; -- Universal Constant of Uncertainty
BEGIN
arr(-2,147,483,647) := 'smallest';
arr(0) := 'zero';
arr(+2,147,483,647) := 'biggest';
arr(i) := 'UCU'; -- an index variable makes for good loops
```

**警告** 在7.3以前的版本中，一个TABLE元素被分配一个缓存区，大小依据数据类型和它被声明的长度而定，这增添了系统的一点开销。这意味着，如果表为 VARCHAR2 (32 767)，存储在表中的每条字符串需要 32KB缓存，即使这个字符串只有一个字节。注意不要耗光系统的内存。

当声明许多长字符串变量时，考虑是否 PL/SQL总是为 VARCHAR2变量分配所声明大小的缓存是非常明智的。

通过以如下方式声明表，可以指定一个存储的值不能为 NULL (空)：

```
DECLARE
TYPE STR_TYPE IS TABLE OF VARCHAR2(8) NOT NULL
INDEX BY BINARY_INTEGER;
arr STR_TYPE;
BEGIN
arr(0) := NULL; -- raises an exception!
```

还可以使用基本引用，例如对一个数据库列的引用：

```
DECLARE
  TYPE QTY_TYPE IS TABLE OF items.quantity%TYPE NOT NULL
    INDEX BY BINARY_INTEGER;
  qty QTY_TYPE;
BEGIN
  qty(0) := 0; -- quantity of zero
```

使用PL/SQL表的好处包括以下几个方面：

它们让你能够将大量数据传递进、出一个子程序。

它们能够很快地在一组巨型数据上执行操作。

它们可以和来自数据库表的大量数据集一起加载和处理，而不需要额外的数据库访问开销。

在Oracle 8i中，它们可以在一个带有new Table()运算符的选择语句中使用。

它们简化了执行复杂的操作，这些操作使用SQL实现将非常困难，尤其是那些在数据集中含有多个行的操作。

另外，可以将这些PL/SQL表捆绑到OCI中的数组和预编译程序中，这会允许非常快速的数据传输。但是，当在PL/SQL表和这些程序之间转移大量的数据时，请考虑网络负载的增加。

注意 只可以将基于标量数据类型的一维宿主数组捆绑到PL/SQL表（不支持基于记录的宿主数组）。

如果表变量有相同的基本类型，可以相互赋值。赋值将表的整个内容拷贝给其他的表变量，这项技术还用于清除一个表。

```
DECLARE
  TYPE ORDITEM_TYPE IS TABLE OF orders.item_no%TYPE
    INDEX BY BINARY_INTEGER;
  clr_orditem ORDITEM_TYPE; -- use ONLY to clear array
  new_orditem ORDITEM_TYPE; -- new list of order items
  old_orditem ORDITEM_TYPE; -- old list of order items
BEGIN
  ... -- fill up new order items array
  old_orditem := new_orditem; -- copy for safekeeping
  ... -- modify new order items array and save
  new_orditem := clr_orditem; -- erase contents for next operation
```

如果试图读取一个还没有被赋值的元素，PL/SQL出现NO\_DATA\_FOUND例外。程序清单28-10展示了这种现象。

清单28-10 badref.sql——在给数组元素赋值前引用它会产生一个例外

```
DECLARE
  SCHAR VARCHAR2(1); -- we'll use this for typing only
  TYPE SCHAR_TYPE IS TABLE OF SCHAR%TYPE INDEX BY BINARY_INTEGER;
  schar_arr SCHAR_TYPE;
  local_schar SCHAR%TYPE;
BEGIN
  local_schar := schar_arr(0); -- no value stored!
END;
/
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 7
```

在PL/SQL 2.3版（Oracle 7.3）和以后的版本中，表还可以基于所声明的数据库表和记录类型。每个记录成员都必须是一个标量类型。这样可以让你能够加载并处理一组数据库记录。

## 5. 引用行类型的数组

可以使用点符号引用拥有数组元素的记录的一个特定成员：

```
DECLARE
  TYPE ORDERS_TYPE IS TABLE OF orders%ROWTYPE
    INDEX BY BINARY_INTEGER;
  qty orders.quantity%TYPE; -- local copy of quantity
  orders_tab_rec ORDERS_TYPE; -- array of table rows
BEGIN
  ... -- fill the array with database table rows
  qty := orders_tab_rec(i).quantity;
```

## 6. PL/SQL表属性

在PL/SQL 2.3版中，表拥有可以帮助你更为容易地操作它们的属性。COUNT、FIRST和LAST可以在表自身上操作，EXISTS、PRIOR、NEXT和DELETE可以在一个特定的表元素上操作。表28-4列出了一些这样的属性。

表28-4 使用PL/SQL表属性

属 性	说 明	返 回 值	用 法
EXISTS	测试一个表元素的值	TRUE/FALSE	Tablename.EXISTS ( 偏移量 )
COUNT	返回表中的项目数	BINARY_INTEGER	Tablename.COUNT
FIRST	返回第一个表项的偏移量	BINARY_INTEGER	Tablename.FIRST
LAST	返回最后一个表项的偏移量	BINARY_INTEGER	Tablename.LAST
PRIOR	返回上一个表项的偏移量	BINARY_INTEGER	Tablename.PRIOR ( 偏移量 )
NEXT	返回下一个表项的偏移量	BINARY_INTEGER	Tablename.NEXT ( 偏移量 )
DELETE	删除指定的元素，部分或全部	无	Tablename.DELETE ( [off_m, [off_n]] )

当已经位于FIRST或LAST元素时，PRIOR和NEXT属性分别返回NULL（空）。DELETE可以删除指定的元素。使用下列任意方式，可以删除一定范围内的元素或全部元素：

Tablename.DELETE ( m )；——仅删除一个元素项。

Tablename.DELETE ( m , n )；——删除范围[m..n]内的全部元素。

Tablename.DELETE；——删除表中的全部元素。

如果在一个指定范围内的偏移量不含有元素项，PL/SQL简单地跳过它（不会发生例外）。范围必须是升序的，如果m>n，DELETE属性不做任何操作。

## 7. 变量数组和嵌套表

Oracle 8i定义了两类新类型的类数组结构或集合：变量数组或变长数组，以及嵌套表。它们不同于PL/SQL表，PL/SQL表现在称为“被索引的表”，区别在于如何使用它们以及它们主要的目的不同。被索引的表是PL/SQL结构，而数组和嵌套表则集中在数据库本身。例如，它们可以存储在数据库表中，就像它们是任意其他的数据列，但是它们不能够被捆绑到宿主变量中。

## 8. 创建变量数组

创建一个数组类似于创建一个PL/SQL表，因为首先需要声明类型，然后创建那种类型的一个变量。但是又不同于PL/SQL表，一个数组是“紧凑的”，不是“松散的”——在数组中没有“空洞”，空洞是指没有数据，这点与表完全不同。还有，PL/SQL表事实上是没有界限的，而数组有一个明确的限制。当创建一个数组时，数组为空，并且可以被测试为无效。为了能够使用它，必须首先使用一个构造器初始化它，如下列代码所示：

```
TYPE Performers IS VARRAY(250) OF varchar2(100);
TYPE Contacts is VARRAY(200) OF prospects;
showLineup Performers('Juggler', 'Singer', 'Fire Swallower');
myContacts Contacts := Contacts('John', 'Mary', 'Tito');
```

### 9. 嵌套表

还可以在SQL\*Plus中创建类型，并将类型用做数据库中的嵌套表。这时，由数据库管理存储。

```
CREATE TYPE act AS OBJECT (
    act_no number(5),
    act_name varchar2(35),
    act_desc varchar2(500));

CREATE TYPE performers IS TABLE OF act;

CREATE TABLE schedules (evt_id number(5) PRIMARY KEY,
    evt_name varchar2(100),
    evt_date date,
    evt_location varchar2(100),
    evt_lineup performers)
NESTED TABLE evt_lineup STORE AS evt_lineup_tab;
INSERT INTO schedules VALUES (1138, 'Not Quite the Greatest Show On Earth',
sysdate+180, 'Colorado Springs', performers(act('1', 'John', 'Trick Riding'),
    act('2', 'Mary', 'High Wire'),
    act('3', 'Albert the Dog-faced Boy', 'Contortionist')));
```

处理每个元素涉及 Oracle8i 中一个新运算符的用法——TABLE()。TABLE() 返回一个变量数组，可以允许个别地访问数据，虽然它的语法看起来并不直观。要为上面显示的科罗拉多州春天表演添加一个节目，参阅下列的代码片段：

```
BEGIN
    INSERT INTO
        TABLE(SELECT evt_lineup FROM schedules WHERE evt_id = 1138)
        VALUES('12', 'The Amazing Rando', 'Magician');
END;
```

同样，可以使用这段代码修改信息：

```
BEGIN
    UPDATE TABLE(SELECT evt_lineup FROM schedules WHERE evt_id = 1138)
        SET act_desc = 'Ruler of the Known Universe'
        WHERE act_no = 12;
END;
```

使用这条语句，可以删除 Mary 的走钢丝表演：

```
BEGIN
    DELETE TABLE(SELECT evt_lineup FROM schedules WHERE evt_id = 1138)
        WHERE act_name = 'Mary';
END;
```

当然，使用下面的代码可以取出信息来查看：

```
DECLARE
    MyAct varchar2(35);
    MyDesc varchar2(500);
BEGIN
    SELECT act_name, act_desc INTO MyAct, MyDesc
        FROM TABLE(SELECT evt_lineup FROM schedules WHERE evt_id = 1138)
        WHERE act_no = 3;
    ...
END;
```

### 10. 使用子类型

在 Oracle 7.2 版 PL/SQL 2.2 中引入用户定义子类型。子类型让你能够为一个类型提供另外一种更有意义的名字，不受任何约束。它们不定义存储，仅仅是一个更通用数据类型的另外一个名称。例如，预定义子类型 NUMBER 和 CHAR 就使用了这种方式。不能直接使用一个固定长度约束一个子类型；相反，必须使用包含 %TYPE 和 %ROWTYPE 的两步方法。这里是一些例子（正确的和错误的）：

```

SUBTYPE SSN_TYPE IS VARCHAR2;           -- legal (method #1)
emp_ssn SSN_TYPE(9);                     -- legal
SUBTYPE PHONE_TYPE IS VARCHAR2(10);      -- illegal!
big_string VARCHAR2(32767);              -- will be used below
TYPE MAX_STRING_TYPE IS big_string%TYPE; -- legal (method #2)
dynam_str MAX_STRING_TYPE;               -- legal

```

注意方法#2，这是定义长度约束子类型的一种方法。另外一种方法是基于一个约束的数据库列类型定义变量。使用这种方法能够帮助你基于子类型捆绑检查变量。当合理地使用时，这种方式可以产生良好的自书写程序代码。

**提示** 通过基于数据库列进行变量定义（变量将包含数据库列的信息），可以帮助防止取值错误。如果更改表中某一列的大小或类型，代码将仍然是正确的。

```

small_str VARCHAR2(20) NOT NULL := 'blick'; -- #1
subtype SMALL_TYPE is small_str%TYPE; -- fails NOT NULL constraint!
tiny_str VARCHAR2(1) := 'T';              -- #2
subtype TINY_TYPE is tiny_str%TYPE;
tiny TINY_TYPE; -- does not inherit assignment!

```

只有子类按顺序基于相同的类型时，基于不同的非约束子类的变量出于赋值目的才是兼容的。例如，DECIMAL和NUMERIC作为NUMBER的子类，可以进行内部转换。如果父类不同，必须进行某种类型的转换。

#### 11. 将一种类型转换为其他类型

在使用强类型语言时，有时需要将一种数据类型转换为其他数据类型。这通常发生在给变量赋值时或在计算一个表达式时。有两种方法可以实现类型转换：显式的和隐式的。

#### 12. 隐式的类型转换

隐式的类型转换由PL/SQL编译器执行，你不需要做任何事情。例如，给出下面的例子：

```

DECLARE
  flt_x FLOAT(3,2) := 2.25;
  int_y INTEGER(6) := 100;
  flt_z DOUBLE(10,2);
BEGIN
  flt_z := flt_x + int_y; -- implicit numeric conversion
END;
/

```

为了正确地进行运算，整型值被隐式地转换为一个浮点值。这在编译器中很常见，为了能够执行计算，一个值被转换为最通用类型的值。在PL/SQL中，VARCHAR2数据类型是最通用的数据类型。

表28-5显示了全部合法的隐式类型转换。如果在列表中没有看到一个转换，你将不得不对它进行显式转换，需要使用内嵌的转换函数。

表28-5 隐式类型转换

To FROM	VARCHAR2	CHAR	NUMBER	DATE	RAW	ROWID
BINARY_						
INTEGER						
CHAR						
DATE						
LONG						
NUMBER						
RAW						
ROWID						
VARCHAR2						

PLS\_INTEGER具有与BINARY\_INTEGER相同的隐式转换特性，可以隐式地在它们之间进行类型转换。下面的例子都是合法的声明和赋值：

```
DECLARE
    date_from DATE := '20-AUG-85'; -- char to date (match NLS format)
    cnt INTEGER(3) := '0';          -- char to integer (subtype of number)
    loop_control BINARY_INTEGER;
    bin_val RAW(2) := '1';           -- char to raw
    str VARCHAR2(9) := '26-AUG-60'; -- char to varchar2
    short_num VARCHAR2(3);
BEGIN
    loop_control := cnt + '1';       -- char to number to binary integer
    str := date_from;               -- date to varchar2
    short_num := bin_val;            -- raw to varchar2
    cnt := short_num;                -- varchar2 to integer
```

任何值都可以被隐式地转换为CHAR和VARCHAR2类型，也可以从CHAR和VARCHAR2类型的值隐式地转换为任何值。如果要将一个RAW值转换为NUMBER值，可以首先将RAW值转换为VARCHAR2值，然后再转换为NUMBER值。隐式转换同样支持一个数据类型的子类。

对于DATE（日期）类型，只有当字符串是为数据库定义的缺省NLS格式时，一个从CHAR或VARCHAR2类型的隐式转换才可能正确发生。如果字符串是不同格式，必须执行一个显式转换并要提供一个转换字符串。

13. 显式类型转换 转换表28-6列出了所有能够执行显式数据类型转换的PL/SQL函数。

表28-6 显式数据类型转换

从.....到	VARCHAR2	CHAR	NUMBER	DATE	RAW	ROWID
VARCHAR2			TO_NUMBER (vc, [fmt (,lang]])	TO_DATE (vc, [fmt [, lang]])	HEXTORAW (vc)	CHARTOROWID (vc)
CHAR			TO_NUMBER (vc, [fmt [, (,lang]])	TO_DATE (vc, [fmt [, lang]])	HEXTORAW (c)	CHARTOROWID (c)
NUMBER	TO_CHAR (n, [format [,lang]])	TO_CHAR (n,[format [,lang]])		TO_DATE (n, [fmt [, lang]])		
DATE	TO_CHAR (dt, [format [,lang]])	TO_CHAR (dt, [format [,lang]])				
RAW	RAWTOHEX (raw)	RAWTOHEX (raw)				
ROWID	ROWIDTOCHAR (rowid)	ROWIDTOCHAR (rowid)				

其中：

fmt是一个格式字符串。

lang是NLS语言格式字符串。

实际上，TO\_CHAR()返回VARCHAR2，但是由于CHAR和VARCHAR2两种类型可以隐式转换，所以返回值可以容易地用做CHAR。但是要记住，没有LONG类型的转换程序。如果要使用LONG，必须首先隐式地将它转换为VARCHAR2，然后显式地转换为其他类型的数据。

### 28.5.5 赋值

PL/SQL不像某些语言，例如C或Pascal，赋值必须是一条单独的语句。只能在一条语句里给一个变量赋值。不能在另一条语句内部给变量赋值，例如，在一个 IF条件测试语句中赋值。

只要可以决定表达式的类型，在编译时就可发现大多数数据类型不相符的错误。取值错误在运行时被发现。

在处理 NULL（空）值时要特别注意。任意含有一个 NULL值的数学运算总要返回一个 NULL（空）值；根据定义，结果是不确定的。这点不适用于字符串连接。例如：

```
DECLARE
  x NUMBER;
  y NUMBER;
  z NUMBER := 10;
  a VARCHAR2(20) := 'Hello, ';
BEGIN
  x := 1/NULL;          -- result is NULL
  y := x * (z - 1);     -- result is NULL
  z := (y - 1)**10;     -- result is NULL
  a := a || NULL || 'World!'; -- result is 'Hello, World!'
```

### 28.5.6 循环

PL/SQL支持三种方式的循环：

非条件循环（Do-Forever循环）。

迭代循环（FOR循环）。

条件循环（WHILE循环）。

最简单的循环看起来就像这样：

```
LOOP
  NULL; -- infinite loop!
END LOOP;
```

注意LOOP..END LOOP的语法。Oracle不检查无限循环。必须进行测试以跳出循环。

有几种方式能够退出循环：

EXIT——无条件跳出循环。在一个IF测试中使用它。

EXIT WHEN——当给定的条件存在时，有条件退出循环。

GOTO——跳出循环，到循环外部的语句。

显然，推荐使用头两种涉及EXIT的方法。这里有一些示例：

```
DECLARE
  i NUMBER := 0;
BEGIN
  LOOP -- example #1
    i := i + 1;
    IF (i >= 100) THEN -- enough iterations
      i := 0; -- reset
      EXIT; -- unconditional termination
    END IF; -- enough iterations
  END LOOP; -- example #1 done

  LOOP -- example #2
    i := i + 1;
    EXIT WHEN (i >= 100); -- conditional termination
  END LOOP; -- example #2 done
```

当然，当条件满足，你要做的就是终止循环时，EXIT WHEN语法更加清晰。在退出循环

前，当必须复位数值或做一些其他工作时使用条件逻辑。

### 1. 控制循环

可以使用 WHILE 循环在一个循环的开始处测试一个条件。只要条件为真，循环继续重复。相反，如果需要在循环的尾部测试一个条件，使用 LOOP-EXIT WHEN 结构：

```
WHILE (x < 10) LOOP  -- While loop
...
    x := x + 1;
END LOOP;  -- done
LOOP  -- simulated Repeat-Until (or Do-While) loop
...
    EXIT WHEN ...
END LOOP;  -- done
```

在循环的头部和尾部添加注意是一个良好的编程习惯，尤其当程序非常长时，就会知道为什么要使用注意了。

PL/SQL 不支持类 FORTRAN 的 CONTINUE 语句转到循环的结束，而跳过它们之间的所有代码。一些人会告诉你在这种情况下，使用条件逻辑模块化程序代码，会使程序代码变得令人讨厌，犬牙交错且难于维护。但是，这里有一个解决方案，可以使用 GOTO 语句模拟 CONTINUE 语句（参阅清单 28-11）。CONTINUE 是 SQL 中的一个保留字，所以不可以在一条语句中或标签中使用它。

清单 28-11 contine.sql——模拟

```
DECLARE
j NUMBER := 0;
BEGIN
    DBMS_OUTPUT.enable;
    LOOP  -- print even numbers between 0 and 20
        IF (MOD(j, 2) = 1) THEN  -- skip odd numbers
            GOTO CONTINE;  -- the misspelling is an inside joke
        END IF;  -- skipping odd numbers
        DBMS_OUTPUT.put_line(TO_CHAR(j) || ' is even');
        <<CONTINE>>
        EXIT WHEN j = 20;  -- done
        j := j + 1;  -- don't forget to increment the loop counter!
    END LOOP;  -- print even numbers between 0 and 20
END;
/
```

服务器响应如下信息：

```
0 is even
2 is even
4 is even
6 is even
8 is even
10 is even
12 is even
14 is even
16 is even
18 is even
20 is even
PL/SQL procedure successfully completed.
```

以这种方式使用 GOTO 语句，简化了一个循环中的逻辑关系。

### 2. 使用 FOR 循环进行迭代

另外一种循环执行指定数量的迭代：

```
BEGIN
  FOR i IN 1..100 LOOP -- do nothing for exactly 100 iterations
    NULL;
  END LOOP; -- done doing nothing
END;
```

注意FOR循环的语法，IN子句必须指定一个范围。在这个案例中，不需要声明循环控制变量（在本例中是i），因为它在FOR循环的作用域内自动创建。这意味着不能在循环的外部引用该变量，因为当循环完成时，它不再存在。如果出于某种原因需要循环变量的值，必须将它拷贝到其他的变量中（参见清单 28-12）。

清单28-12 lastodd.sql——获取循环控制变量的值

```
DECLARE
  j NUMBER;
BEGIN
  DBMS_OUTPUT.enable; -- enable output
  FOR i IN 1..100 LOOP -- do nothing for exactly 100 iterations
    IF (MOD(i, 2) = 1) THEN -- must be odd
      j := i;
    END IF; -- capture odd numbers
  END LOOP; -- done doing nothing
  DBMS_OUTPUT.put_line('last odd number was ' || TO_CHAR(j));
END;
/
```

服务器显示下列信息：

```
last odd number was 99
PL/SQL procedure successfully completed.
```

PL/SQL不支持不为1的任意其他步进值。但是可以模拟步进值不为1的情形！有两种方法：一种方法是使用步长乘以循环计数器，以得到需要用于处理的数值。不能真正地修改循环控制变量，因为这样做是非法的（它是严格的只读值）。另外一种方法是创建一个变量，并且使用希望的数值增长它，这样做或许会让一个简单的循环更加清楚，但是它在FOR循环里仍然是合法的。也许必须要调整循环的范围，以得到所希望的行为。当然，可以使用EXIT或EXIT WHEN提早退出一个循环。

可以采用倒数方式，反向执行重复（参见清单 28-13）。

清单28-13 countdwn.sql——在一个循环中使用

```
BEGIN
  FOR j IN REVERSE 1..10 LOOP -- countdown
    DBMS_OUTPUT.put(TO_CHAR(j) || '-');
  END LOOP; -- countdown
  DBMS_OUTPUT.put_line('Blastoff!');
END;
/
```

将得到下列信息：

```
10-9-8-7-6-5-4-3-2-1-Blastoff!
PL/SQL procedure successfully completed.
```

### 28.5.7 使用游标

游标是一个对象，能够提供行级的 SQL 语句控制。游标声明不同于变量，而是用于实现游标的一个内存区域的句柄。游标声明仅仅定义了将提交给 SQL 语句执行器（SQL Statement Executor）哪些查询，在可执行代码中的程序控制下进行查询的管理。游标可以表示任意合法的 SQL SELECT 语句。游标通常是任何 PL/SQL 应用的基本构建块，它们为数据库中存储的数据集合上的操作提供了循环机制。如果还需要进行更新，使用 FOR UPDATE 子句。

**警告** 当心，使用 FOR UPDATE 子句将锁定查询发现的所有记录。所有这些记录在游标被关闭前将保持被锁定状态。

游标可以返回一条或多条记录，或一条也不返回。通常的操作顺序是：

- 1) 声明游标，以及一个检索记录的数据结构。
- 2) 打开游标。
- 3) 从游标中重复提取每条记录到数据结构中，直到数据集合被提空。
- 4) 关闭游标。

#### 1. 定义一个游标

在定义游标时，要注意几个主题上的变化：

```
DECLARE
  -- gets all orders in database
  CURSOR get_orders IS
    SELECT * FROM orders;
  -- gets a few columns for a specified order number
  CURSOR get_orditem(Pord_num orders.ord_num%TYPE) IS
    SELECT seq_num, quantity, unit_price, extended_price
    FROM orders
    WHERE ord_num = Pord_num;
  -- gets the whole row for a particular item#
  CURSOR get_items(Pitem_no items.item_no%TYPE) RETURN items%ROWTYPE IS
    SELECT * FROM items WHERE item = Pitem;

  -- gets the item name for a particular item#
  CURSOR get_item_name(Pitem_no items.item_no%TYPE)
  RETURN items.item_name%TYPE IS
    SELECT item_name FROM items WHERE item_no = Pitem_no;
```

虽然有不少的差别，但仅有两个基本模式。一个游标可以接受参数或不接受参数。当打开游标的时候，为参数提供数值。可以定义返回类型，或不定义返回类型。返回类型可以是用户定义记录、数据库表记录类型或独立的变量。不论哪种方式，在 SELECT 子句中定义的列必须与用于接受返回值的参数一一对应。

一旦游标被声明，就可以打开它、提取记录、检查它的状态并且在不再需要它的时候关闭它。

一个游标可以拥有多个名字，上面例子中的约定是在游标的名字后添加 \_cur 或 \_loop 后缀，或在游标的名字前使用 get\_ 前缀（完全取决于个人的风格）。在游标名字中使用表的名字，或试图让它有意义（不要太罗嗦），例如 get\_addresses 或 employee\_by\_dept\_cur 这样的名字。

要清楚参数表只能输入数值，不能输出数值。这就是为什么既不需要也不允许参数流的原因。而且输入参数必须是标量数值。

当打开一个游标时，执行 SQL 程序并计算相应的数据集合。但是，没有记录被真正地返

回给程序。使用FETCH语句一次提取一条记录。在执行另外一条提取语句之前，被提取的记录保持为当前行。只能正向提取记录，在数据集中没有可以回退的控制。

## 2. 游标属性

游标具有表28-7所描述的属性。

表28-7 游标属性

属 性	返 回 值	说 明
ISOPEN	TRUE/FALSE (真假)	指出一个游标是打开的还是关闭的
FOUND	TRUE/FALSE (真假)	指出是否发现一条记录
NOTFOUND	TRUE/FALSE (真假)	指出是否没有发现一条记录
ROWCOUNT	NUMBER	每条提取记录的序数值 (第一、第二、第三...)

这里是它们使用的一些示例：

```
IF (orders_cur%FOUND) THEN -- got an order
  OPEN items_cur(orders_cur.orderno); -- open a related cursor
  LOOP
    FETCH items_cur into order_item;
    EXIT WHEN items_cur%NOTFOUND; -- break out when done
    -- show how many rows were processed so far
    DBMS_OUTPUT.put_line('On Row #' || TO_CHAR(items_cur%ROWCOUNT));
  END LOOP;
END IF; -- got an order
...
IF (items_cur%ISOPEN) THEN -- close the cursor
  CLOSE items_cur;
END IF; -- close an open cursor
```

## 3. 游标的FOR循环

一个游标的FOR循环是使用游标最简单的方式。在FOR循环范围内自动处理游标的打开、提取和关闭。也隐式定义返回记录变量，不能在循环范围以外引用返回记录变量。游标可以接受参数。既可以在声明部分声明游标，也可以在游标FOR循环本身的程序体中声明游标（参见清单28-14）。

清单28-14 cfor1.sql——简单使用游标循环

```
DECLARE
  CURSOR get_tables IS
    SELECT * FROM user_tables;
BEGIN
  FOR get_tables_cur IN get_tables LOOP
    DBMS_OUTPUT.put_line(get_tables_cur.table_name);
  END LOOP;
END;
/
```

如果作为scott/tiger用户运行这段程序，将得到下列信息：

```
BONUS
DEPT
EMP
SALGRADE
PL/SQL procedure successfully completed.
```

行2~3定义游标。

行5定义了游标记录变量，它与游标一起使用。

行6使用点符号引用游标中指定的列。

如清单28-15所示，可以使用更为简明的方法得到相同的结果：

清单28-15 cfor2.sql——一个更为简单的循环

```
BEGIN
  FOR get_tables_cur IN (SELECT * FROM user_tables) LOOP
    DBMS_OUTPUT.put_line(get_tables_cur.table_name);
  END LOOP;
END;
/
```

区别在于游标也在循环内部定义。对于简单的程序，这样做非常好。

还可以接受一个参数并且查询一个特定的表（参见清单28-16）。

清单28-16 cfor3.sql——接收参数的简单循环游标

```
DECLARE
  CURSOR get_tables(Powner all_tables.owner%TYPE) IS
    SELECT * FROM all_tables
      WHERE owner = Powner;
  local_owner all_tables.owner%TYPE := 'DEMO'; -- search criteria
BEGIN
  FOR get_tables_cur IN get_tables(local_owner) LOOP
    DBMS_OUTPUT.put_line(get_tables_cur.table_name);
  END LOOP;
END;
/
```

注意有效地使用基本变量，尤其在游标定义中，还要注意参数是怎样传递给游标的。这一次你得到下面的信息（因为目前只能查看模式 DEMO中的表，这些表对于用户 scott是可见的）：

```
CUSTOMER
DEPARTMENT
EMPLOYEE
JOB
LOCATION
SALARY_GRADE
SALES_ORDER
PL/SQL procedure successfully completed.
```

对于较大的、复杂的程序，在封装块范围内部定义游标（正如前面进行的），并且自己控制游标。

#### 4. 打开、提取与关闭游标

从上一个例子开始，比较罗嗦的方法，看起来类似清单28-17。

清单28-17 ofc.sql——自己来控制游标

```
DECLARE
  -- cursor definitions
  CURSOR get_tables(Powner all_tables.owner%TYPE) IS
    SELECT * FROM all_tables
      WHERE owner = Powner;
  -- record variable definitions
  get_tables_rec get_tables%ROWTYPE;
```

```

-- local variables
local_owner all_tables.owner%TYPE := 'DEMO'; -- search criteria
BEGIN
  OPEN get_tables(local_owner); -- compute rows to return
  LOOP -- find all DEMO tables available to SCOTT
    FETCH get_tables INTO get_tables_rec; -- try to get a row
    EXIT WHEN get_tables%NOTFOUND; -- no more rows
    DBMS_OUTPUT.put_line(get_tables_rec.table_name);
  END LOOP;
  CLOSE get_tables; -- done with this cursor
END;
/

```

你得到与先前完全相同的数据集。这次定义记录变量来接受基于游标的记录。或许你已经自己定义了一个游标（使用 TYPE RECORD IS... 语句）。事实上，或许你已经定义了一个不同的变量来提取记录，但是基于游标的返回类型定义变量非常简单而且易于维护。现在，如果更改游标的列列表，除了需要修改涉及记录变量的东西之外，不需要修改任何东西。

在打开游标之后，游标将计算游标返回的记录数（虽然游标还没有真正返回任何东西），循环提取全部记录，每次提取一条记录，需要检测数据集的结尾并跳出循环，如第 14 行所示。当循环结束后，第 17 行代码关闭循环。为了能够再次使用这个游标，必须重新打开它，这一次打开游标的时候或许带有几个参数。

假设只关心一条记录，程序清单 28-18 显示了最简单的解决方案。

清单 28-18 onerow.sql——显式提取一条记录

```

DECLARE
-- cursor definitions
CURSOR get_tables(Powner all_tables.owner%TYPE) IS
  SELECT * FROM all_tables
  WHERE owner = Powner;
-- record variable definitions
get_tables_rec get_tables%ROWTYPE;
-- local variables
local_owner all_tables.owner%TYPE := 'DEMO'; -- search criteria
BEGIN
  OPEN get_tables(local_owner); -- compute rows to return
  FETCH get_tables INTO get_tables_rec; -- try to get a row
  IF (get_tables%FOUND) THEN -- got a row
    DBMS_OUTPUT.put_line(get_tables_rec.table_name);
  END IF;
  CLOSE get_tables; -- done with this cursor
END;
/

```

这次得到下面信息：

```

CUSTOMER
PL/SQL procedure successfully completed.

```

注意游标属性 %FOUND 用于检查是否真正地提取到记录。如果没有返回任何记录，记录变量的值将与 FETCH 语句执行前的值相同。实际上，让我们来证明这点（参见清单 28-19）。

清单 28-19 badfetch.sql——仅仅成功地重写一个记录变量

```

DECLARE
-- cursor definitions

```

```

CURSOR get_tables(Powner all_tables.owner%TYPE) IS
    SELECT * FROM all_tables
    WHERE owner = Powner;
-- record variable definitions
get_tables_rec get_tables%ROWTYPE;
-- local variables
local_owner all_tables.owner%TYPE := 'BLICK'; -- unknown owner
BEGIN
    get_tables_rec.table_name := 'GARBAGE'; -- initialize to something
    OPEN get_tables(local_owner);          -- compute rows to return
    FETCH get_tables INTO get_tables_rec; -- try to get a row
    IF (get_tables%NOTFOUND) THEN          -- got a row
        DBMS_OUTPUT.put_line(get_tables_rec.table_name);
    END IF;
    CLOSE get_tables; -- done with this cursor
END;
/

```

你得到下面信息：

```

GARBAGE
PL/SQL procedure successfully completed.

```

这个结果证明了一点：如果FETCH语句失败，它不会重写它提取的任何东西。

### 5. 使用隐式游标

隐式游标相对来说需要较少的维护。只有当你预料将得到一条记录时，才会使用隐式游标。如果找到0条或多于1条的记录，会出现NO\_DATA\_FOUND或TOO\_MANY\_ROWS例外，它并不会给你带来任何麻烦。程序清单 28-20展示了一个活动的隐式游标。

清单28-20 一个简单的隐式游标

```

DECLARE
    -- record variable definitions
    get_tables_rec all_tables%ROWTYPE; -- based on the table
    -- local variables
    local_owner all_tables.owner%TYPE := 'DEMO'; -- search criteria
    local_table all_tables.table_name%TYPE := 'CUSTOMER';
BEGIN
    SELECT * INTO get_tables_rec -- looking for a single row
    FROM ALL_TABLES
    WHERE owner = local_owner AND table_name = local_table;
    DBMS_OUTPUT.put_line(get_tables_rec.tablespace_name);
END;
/

```

你得到如下信息：

```

USER_DATA
PL/SQL procedure successfully completed.

```

这就是你没有构造查询，以便能够准确地返回一条记录所发生的事情（参见程序清单 28-21）。

清单28-21 toomany.sql——当发现多条记录时，隐式游标失效

```

DECLARE
    -- record variable definitions
    get_tables_rec all_tables%ROWTYPE; -- based on table
    -- local variables
    local_owner all_tables.owner%TYPE := 'DEMO';

```

```
BEGIN
  SELECT * INTO get_tables_rec FROM ALL_TABLES
  WHERE owner = local_owner;
  DBMS_OUTPUT.put_line(get_tables_rec.tablespace_name);
END;
/
```

服务器反馈下面一段有意义的错误消息：

```
ERROR at line 1:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 7
```

这是实际发生的事情：

- 1) Oracle打开游标并提取一条记录。
- 2) 没有满足条件，Oracle试图进行另外一次提取操作并得到另外一条记录。
- 3) 因为你不能将10磅的记录放入5磅的变量中，Oracle关闭游标（游标会以任何手段关闭自己）并且产生例外。

提示 Oracle通常在隐式游标上进行两次提取，只是为了查看游标是否返回另外一条记录（并且因此导致请求失败，因为一个隐式游标确切地只返回一条记录）。这是ANSI标准的一个要求。

当全部需求就是一条记录，并且不希望被例外弄乱时，需要使用一个显式游标并且使用一条FETCH语句。如果希望知道是否有多于1条的记录（或根本没有记录），并且不在乎处理例外，那么使用隐式游标。事实上，可以在一个单行查询程序中使用隐式游标，就是出于这个原因。需要知道单行查询是否确实就是一条记录，或者察觉到它有错误。如果只关心得到一条记录，使用显式游标处理起来效率更高。

隐式游标不能使用数组处理额外的记录。根据定义，所有的隐式游标都被假设为只返回一条记录。

### 28.5.8 例外处理

例外是一个非致命事件，它立即中断程序的正常执行并引起一个非条件转移，跳转到当前程序块的例外处理部分。一些例外，像 NO\_DATA\_FOUND 或 TOO\_MANY\_ROWS，可以被认为是正常的处理部分。像 VALUE\_ERROR 这样的例外表明一个程序错误或一些意料之外的事件。还有一些例外表明一个严重的问题，例如耗光内存。

如果没有给一个程序块定义例外处理程序，例外将被返回给下一个更高层的块（如果有的话）。例外从封装的块中“冒出”，直到为该块找到一个例外处理程序，或将控制返回给调用的环境（在我们这里的案例中是 SQL\*Plus）。

你的程序将始终经历例外，你所需要的实际就是一个例外处理程序，用它来处理这些例外。再次参见上次出现问题的隐式游标（参见程序清单 28-22）。

清单 28-22 gracefule.sql——使用例外处理程序

```
DECLARE
  -- record variable definitions
  get_tables_rec all_tables%ROWTYPE; -- based on table
  -- local variables
  local_owner all_tables.owner%TYPE := 'DEMO';
  status NUMERIC := 0; -- capture error code (initialize to OK)
```

```
BEGIN
  SELECT * INTO get_tables_rec FROM ALL_TABLES
  WHERE owner = local_owner;
  DBMS_OUTPUT.put_line(get_tables_rec.tablespace_name);
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
    status := SQLCODE;
    DBMS_OUTPUT.put_line('get_tables: ' || SQLERRM(status));
    DBMS_OUTPUT.put_line('get_tables: Exiting gracefully.');
```

---

这一次，服务器给出下列信息：

```
get_tables: ORA-01422: exact fetch returns more than requested number of rows
get_tables: Exiting gracefully.
PL/SQL procedure successfully completed.
```

在第6行上，定义了一个变量用于存储潜在的错误代码。

在第12行上，在例外部分中使用一个 WHEN 子句指定例外。

在第13行上，捕获到错误代码。

在第14行上，显示与错误代码相关的错误消息。

服务器现在告诉你程序已经成功地完成。如果没有出现错误，SQLCODE将被设置为零，零意味着一切正常。SQLCODE实际上就是一个返回SQL状态的内建函数。

表28-8是一个预定义例外的列表。

表28-8 可以在例外块中被显式地测试的例外

例外名称	Oracle错误代码	值
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
STORAGE_ERROR	ORA-06500	-6500
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
TRANSACTION_BACKED_OUT	ORA-00061	-61
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476

注意大多数例外如何拥有负的 SQLCODE 数值，而 NO\_DATA\_FOUND 是最明显的例外（没有故意使用双关语）。这是因为 ANSI 标准要求当没有记录被发现时，SQLCODE 的数值应该是 +100，即使 Oracle 错误代码都是负的。Oracle 是一个 ANSI 兼容数据库。你或许会说它使得它们 ANSI 化了（不好的双关语）。

如果在上面的列表中不包括预期的例外，或者不能确定将会发生什么例外，或者除了处理它们并不关心它们，可以使用 WHEN OTHERS 子句，这是一个捕捉器，允许例外处理器处理可能会出现的任意错误。

还可以使用任意例外名的组合，并且使用 WHEN OTHERS THEN 子句结束名称组合列表。

它们在一起的作用类似一组 IF ...ELSIF...ELSE 语句。如果它们中的任何一个为真，相关的程序代码将被执行，并且程序块将被退出：

```
EXCEPTION
  WHEN NO_DATA_FOUND THEN -- row not found!
    -- do this and skip to end of block
    ...
  WHEN ZERO_DIVIDE THEN -- divide by zero!
    -- do that and skip to end of block
    ...
  WHEN VALUE_ERROR THEN -- 10 lbs of data in a 51b var
    -- do these and skip to end of block
    ...
  WHEN OTHERS THEN -- dunno, don't care
    -- do the other and skip to end of block
    ...
END;
```

提示 一般要在例外处理器尾部编写一个 WHEN OTHERS 子句。有时候，它还是唯一的处理程序部分。

### 1. 避免在一个例外处理器中的无限循环

有时，执行一个例外处理程序中的动作，动作本身就有可能产生例外。这会导致一个无限循环！避免出现这种现象的窍门是将例外处理程序封装在一个块中，并且给它自己的例外处理器。这种内嵌的例外处理器不需要（也不应该）做任何事情。参考下面代码：

```
EXCEPTION -- main block handler
  WHEN OTHERS THEN
    BEGIN -- embedded block
      IF (get_cursor%ISOPEN) THEN -- cursor left open
        CLOSE get_cursor; -- close it
      END IF; -- cursor left open
    EXCEPTION
      WHEN OTHERS THEN
        NULL; -- don't care
    END; -- embedded block
END; -- main block
```

例外处理程序应该执行全部必要的清除工作，例如关闭可能被打开的游标。

提示 一般将一个例外处理器放置在任何应用 PL/SQL 代码的最顶层块中。例外应该以良好的方式处理。用户需要查看的最后一件事情就是一个简洁的、技术性的错误消息。

### 2. 定义你自己的例外

如果预定义例外列表看起来有些简略，通过为任意特定的 Oracle 例外序号关联一个名称，可以更多地定义它们。然后可以在例外程序块中通过名字引用这些例外。要实现这项工作，定义一个例外和一个注记，传递给编译器，告诉你想使用给定的错误代码关联例外。一个注记就是一个编译指示，在编译时处理而不是在运行时处理。它指示编译器应该如何处理特定语言信息或条件。所有的优秀编译器都具有注记（或者类似的东西）。程序清单 28-23 是使用注记的一个示例。

清单 28-23 excpinit.sql——使用 EXCEPTION\_INIT

```
SET SERVEROUTPUT ON
DECLARE
```

```

-- exceptions and pragmas (compiler directives)
INVALID_NUM_FORMAT EXCEPTION; -- first we define an exception object
PRAGMA EXCEPTION_INIT (INVALID_NUM_FORMAT, -1481); -- then associate it with
--an exception
-- constants
NUM_FMT CONSTANT VARCHAR2(3) := 'aaa'; -- an invalid number format
x NUMBER(10);
BEGIN
  DBMS_OUTPUT.enable;
  SELECT TO_NUMBER('999', NUM_FMT) INTO x FROM DUAL; -- try to convert
EXCEPTION
WHEN INVALID_NUM_FORMAT THEN
  DBMS_OUTPUT.put_line('Trapped an illegal Number Conversion');
WHEN OTHERS THEN
  DBMS_OUTPUT.put_line('Some other error');
END;
/

```

这次，你得到下列信息：

```

Trapped an illegal Number Conversion
PL/SQL procedure successfully completed.

```

这里看到一个两步过程：首先定义了一个例外，然后将它与一个 Oracle 错误代码关联。下一个问题是，为什么 Oracle 不预先定义全部的错误代码？全部 24 000 个代码吗？Richard Bach（理查德·巴克）曾经写过，“如果你正确地提问问题，问题本身就回答了它自己。”Oracle 仅仅定义了带来问题的 80% 的错误。毕竟，这些东西确实占用内存空间。

另外一种方式是定义应用的特定例外。这与 Oracle 错误代码无关，它们是为应用定义的错误。必须检测违犯一个特定商务规则的情形，并且产生合理的用户定义例外。例如，假设非免税雇员，他一小时挣钱少于 \$10，只能允许加薪百分之 1 至 5。你的朋友，Joe（乔），是一个计时雇员，一小时挣 8 美元，他的老板（也是他的叔叔）希望给他大幅度加薪。程序清单 28-24 显示了这个商务规则的一个实现。

清单 28-24 busrule.sql——使用应用特定的例外捕获违反商务规则的操作

```

DECLARE
-- exceptions
INVALID_ANNUAL_RAISE EXCEPTION;
-- constants
HRLY_STATUS CONSTANT VARCHAR2(1) := 'H'; -- hourly status
HRLY_WAGE_LIMIT CONSTANT REAL(4,2) := 10.00; -- upper hourly wage limit
MIN_HRLY_RAISE CONSTANT REAL(5,4) := 1.0100; -- lower hourly raise limit
MAX_HRLY_RAISE CONSTANT REAL(5,4) := 1.0500; -- upper hourly raise limit
-- info entered by user
emp_name VARCHAR2(20) := '&emp_name'; -- employee name
emp_status VARCHAR2(1) := '&emp_status'; -- employee status
hrly_wage REAL := &wage; -- hourly wage
ann_raise REAL := &annual_raise; -- annual raise
BEGIN
  IF (emp_status = HRLY_STATUS AND -- this is business rule #532
      hrly_wage < HRLY_WAGE_LIMIT AND
      ann_raise NOT BETWEEN MIN_HRLY_RAISE AND MAX_HRLY_RAISE) THEN
    RAISE INVALID_ANNUAL_RAISE;
  ELSIF (FALSE) THEN -- code other rules here (pertaining to raises)
    NULL;
  ELSE -- passed the gauntlet
    DBMS_OUTPUT.put_line(emp_name || ' now makes ' ||
                          TO_CHAR(hrly_wage * ann_raise, '$99.99'));

```

```

END IF;
EXCEPTION
WHEN INVALID_ANNUAL_RAISE THEN
  DBMS_OUTPUT.put_line('Don't give ' || emp_name || ' the raise!');
WHEN OTHERS THEN
  DBMS_OUTPUT.put_line('Some other problem computing ' || emp_name || ' 's
increase');
END;
/

```

服务器以下列方式处理乔老板的需求：

```

Enter value for emp_name: Joe
old 10:  emp_name  VARCHAR2(20) := '&emp_name'; -- employee name
new 10:  emp_name  VARCHAR2(20) := 'Joe';      -- employee name
Enter value for emp_status: H
old 11:  emp_status VARCHAR2(1) := '&emp_status'; -- employee status
new 11:  emp_status VARCHAR2(1) := 'H'; -- employment status
Enter value for wage: 8.00
old 12:  hrly_wage REAL(4,2) := &wage;          -- hourly wage
new 12:  hrly_wage REAL(4,2) := 8.00;          -- hourly wage
Enter value for annual_raise: 1.06
old 13:  ann_raise REAL(5,4) := &annual_raise;  -- annual raise
new 13:  ann_raise REAL(5,4) := 1.06;          -- annual raise
Don't give Joe the raise!
PL/SQL procedure successfully completed.

```

使用应用特定的例外并在你的逻辑程序中引发它们是一个基本的商务规则编码技术。通过建立一个每个人都可以遵循的模板，将代码设计得更具有扩展性与易维护性。

如果希望同一个代码执行两个或多个例外，只需要在一个布尔 OR 条件中将它们列出，如下所示：

```

EXCEPTION
WHEN INVALID_NUMBER OR VALUE_ERROR THEN
  ...
WHEN OTHERS THEN
  ...
END;

```

### 3. 处理在线例外

需要养成的好习惯是使用例外处理程序为所有的 SQL 语句编写程序块。如果一个例外发生在 SQL 语句上，通常希望继续处理在线程序（继续处理下一个概念上的程序块）。这点对于循环尤其正确。如果没有为一个循环内部 SQL 语句编写带有例外处理程序的代码块，任何例外将导致循环立即终止，程序跳转到封装的块例外处理程序（“冒泡”带来的问题）。为了表明这点，请看下面的代码：

```

LOOP -- get some rows from a master table
  FETCH master_cursor INTO master_rec;
  EXIT WHEN master_cursor%NOTFOUND;
  BEGIN -- delete some child table rows
    -- master primary key is foreign key in child
    DELETE FROM child_table
    WHERE master_fkey = master_rec.master_pkey;
  EXCEPTION -- something untoward occurred
    WHEN OTHERS THEN -- so output a message, say
      status := SQLCODE; -- always capture error code!
      DBMS_OUTPUT.put_line('during delete: ' || SQLERRM(status));
  END; -- delete some child table rows
END LOOP;

```

如果带有例外处理程序的封装块不存在，一个例外（例如没有发现记录）将立即跳出循环。所希望的事情是程序跳过这一行，继续执行下一条程序。处理在线例外满足了这种需求。

### 28.5.9 使用子程序

现在继续考虑子程序。子程序使你能够模块化程序代码，并让代码更为有效。

#### 1. 什么是子程序？

子程序是可以被调用一次或多次的子例程。我通常使用术语“子程序”，交替使用子例程。实际上，子程序是Oracle使用的术语，我只是更熟悉它而已。

PL/SQL拥有两种类型的子程序（或子例程）：过程和函数。在一个未命名的PL/SQL块中，在声明部分声明（给予名称和可选参数及类型）并定义（给出可执行代码）函数和过程。可以认为它们是可调用静态代码段。为了能够执行子程序包含的代码，过程和函数必须在执行时被引用。这些子程序，像大多数程序块一样，包含声明（除了用子程序的名字替换关键字DECLARE之外），程序体和可选的例外处理部分，并以END语句结束，有时使用子例程的名字作为标签。

#### 2. 函数

函数是一个子程序，它的名字返回一个某一特定数据类型的数值。可以将这个返回值赋值给相同类型的一个变量，或者在一个表达式中使用该值。函数通常接受参数。当没有参数传递时，不需要使用圆括号。一些简单并且非常熟悉的例子如下所示：

```
status := SQLCODE; -- note no parentheses
NL CONSTANT VARCHAR2(1) := CHR(10); -- ASCII character (newline)
IF (employee_exists('Scaboda')) THEN ... -- use them in expressions
```

在一个未命名PL/SQL块中，在块的声明部分，跟随其他的声明之后，同时声明和实现子程序。程序清单28-25显示了一些函数例子。

清单28-25 boolsub.sql——函数使重复性工作变得简单

```
DECLARE
  -- local variables
  x NUMBER(3);
  -- local subprograms (must follow all other declarations!)
  FUNCTION is_even(Pnum IN x%TYPE) -- returns TRUE if number is even
  RETURN BOOLEAN IS
  BEGIN
    RETURN(MOD(Pnum, 2) = 0); -- test for evenness
  EXCEPTION
  WHEN OTHERS THEN
    RETURN (NULL); -- indeterminate
  END is_even;
  FUNCTION bool_to_char(Pbool IN BOOLEAN)
  RETURN VARCHAR2 IS
    str VARCHAR2(5); -- capture string to return
  BEGIN
    IF (Pbool) THEN -- test Boolean value for TRUE
      str := 'TRUE';
    ELSIF (NOT Pbool) THEN -- FALSE
      str := 'FALSE';
    ELSE -- must be NULL
      str := 'NULL';
    END IF; -- test Boolean value
    RETURN (str);
  END bool_to_char;
```

```
BEGIN -- executable code
  x := 0;
  DBMS_OUTPUT.put_line('It is ' || bool_to_char(is_even(x)) ||
    ' that ' || TO_CHAR(x) || ' is even');

  x := 1;
  DBMS_OUTPUT.put_line('It is ' || bool_to_char(is_even(x)) ||
    ' that ' || TO_CHAR(x) || ' is even');

  x := 2;
  DBMS_OUTPUT.put_line('It is ' || bool_to_char(is_even(x)) ||
    ' that ' || TO_CHAR(x) || ' is even');

  x := 3;
  DBMS_OUTPUT.put_line('It is ' || bool_to_char(is_even(x)) ||
    ' that ' || TO_CHAR(x) || ' is even');

  x := 4;
  DBMS_OUTPUT.put_line('It is ' || bool_to_char(is_even(x)) ||
    ' that ' || TO_CHAR(x) || ' is even');

  DBMS_OUTPUT.put_line(bool_to_char(NULL) || ' is neither TRUE nor FALSE');
END;
/
```

服务器输出下列信息：

```
It is TRUE that 0 is even
It is FALSE that 1 is even
It is TRUE that 2 is even
It is FALSE that 3 is even
It is TRUE that 4 is even
NULL is neither TRUE nor FALSE
PL/SQL procedure successfully completed.
```

这个相当长的例子说明了函数的一些属性和限制：

函数（实际上全部子程序）必须在全部类型、常量和变量之后声明。

函数的局部变量在 IS...BEGIN 之间声明。

参数可以是基本量，当表示数据库列值时，这是推荐使用的方法。

RETURN 子句必须用于返回一个值，即使在 EXCEPTION 子句中也不例外。

实际上，函数调用可以嵌套，并且可以在一个任意复杂的表达式中调用函数。

给予函数的 END 语句与函数名称一个相同的标签（虽然它是可选的，我却总是这样做，而且必须一致）。

每个参数拥有一个模式，一个关键字指定参数是否被当作只读（IN）、只写（OUT）或读写（IN OUT）变量。如果不指定模式，函数缺省模式为 IN。

注意函数冗长的声明。参数变量的作用范围被严格地限定为函数或函数内嵌套的任意子程序的局部变量。通过在 IS...BEGIN 部分定义子程序，可以嵌套其他的子程序，只有当程序执行进入封装的子程序，嵌套的子程序才存在。限制作用范围到最小程序单元的技术帮助减少程序例程之间的连接，这是一件好事情。

虽然从技术角度来说，在参数表中返回一个值是合法的（通过指定模式为 OUT 或 IN OUT），但通常不赞成这种不好的编程风格。如果要这样做，必须说明原因并且必须注明函数的准确用法，将它提升到一个标准的层次。这会防止滥用和不标准的编程风格（有人可能争论函数从参数表中返回值就已经是一种滥用风格了）。

提示 大多数 Oracle 开发者避免从函数的参数表中返回值。为了防止令人讨厌的、不可维护的代码增加，你也许希望彻底禁止这种习惯。

在一个函数中封装小块的代码，甚至只有一行，是非常正常且可以接受的。这种技术减小了代码尺寸并且简化了代码的维护。例如，如果以后需要为函数添加一些额外的数据确认，只需要在一个地方添加它即可。

### 3. 过程

过程是一个子程序，可以执行一定量的重复工作，能够严格地通过参数表传入和传出处。一个过程总是一条语句，不能在一个表达式中插入过程。函数作用范围规则也适用于过程。语法几乎和函数完全相同，只是少了返回语句。

如果没有指定模式，过程的模式缺省为 IN。作为一种风格，我总是指定模式。编译器将捕获一个参数的使用是否与它的模式相一致。IN参数必须出现在一个赋值语句的右边，或在任何能被求值的表达式中。OUT参数只能出现在一条赋值语句的左边。IN OUT参数可以出现在任何地方。

通常应使用子程序所需的最具限制性的模式。这会提供最大限度的程序错误保护，例如在一个子程序执行期间，不小心删除了变量。

当使用模式的时候，服务器复制一份变量用于处理，所以如果产生一个例外的话，变量的原始值被保留下来。但是有些时候，这并不是你所希望的，因为无论如何这会造成非常显著的开销。Oracle8i提出NOCOPY修改量，它让服务器按引用直接指出原始值。如果坚持使用OUT变量并且不在乎保留原始值，通过走这条路可以得到显著的性能改善。

尝试将全部的单行查找程序编写为过程。依照相同的编程风格来处理单行查询，传入键值，一个返回记录类型，以及一个状态指示器。这种风格具有以下优点：为了能够处理在线例外，封装了SQL语句或游标，还可以让程序代码段能够被再利用。它还是一个易于维护的风格。程序清单28-26显示了这种风格的一个例子。

清单28-26 table.sql——标准的编程风格使其成为可维护代码

```
DECLARE
  -- constants
  TB CONSTANT VARCHAR2(1) := CHR(9); -- TAB
  -- variables
  status NUMERIC;
  table_rec all_tables%TYPE;
  -- routines
  PROCEDURE get_table(Powner IN      all_tables.owner%TYPE,
                     Ptable IN      all_tables.table_name%TYPE,
                     Prec          OUT all_tables%TYPE,
                     Pstatus IN OUT NUMERIC) IS
    -- local cursors
    CURSOR table_cur(Cowner all_tables.owner%TYPE,
                    Ctable all_tables.table_name%TYPE) IS
      SELECT *
      FROM all_tables
      WHERE owner = Cowner AND table_name = Ctable;
    -- local variables
    Lowner all_tables.owner%TYPE;
    Ltable all_tables.table_name%TYPE;
BEGIN
  Pstatus := 0; -- OK
  Lowner := UPPER(Powner);
  Ltable := UPPER(Ptable);
  OPEN table_cur(Lowner, Ltable);
  FETCH table_cur INTO Prec;
  IF (table_cur%NOTFOUND) THEN
```

```

        RAISE NO_DATA_FOUND;
    END IF;
    CLOSE table_cur;
EXCEPTION
WHEN OTHERS THEN
    BEGIN
        Pstatus := SQLCODE; -- capture error code
        IF (table_cur%ISOPEN) THEN -- close the open cursor
            CLOSE table_cur;
        END IF;
        Prec := NULL; -- clear return values and display input values
        DBMS_OUTPUT.put_line('get_table: ' || SQLERRM(Pstatus));
        DBMS_OUTPUT.put_line('OWNER = ' || '<' || Lower || '>');
        DBMS_OUTPUT.put_line('TABLE = ' || '<' || Ltable || '>');
    EXCEPTION
    WHEN OTHERS THEN
        NULL; -- don't care (avoid infinite loop)
    END;
END get_table;
BEGIN -- display storage parameters for a given table
    DBMS_OUTPUT.enable;
    DBMS_OUTPUT.put_line('TABLE' || TB || 'TABLESPACE' || TB ||
        'INITIAL' || TB || 'NEXT' || TB || 'MAX');
    DBMS_OUTPUT.put_line(RPAD('-', 43, '-')); -- just an underline
    get_table('scott', 'dept', table_rec, status);
    IF (status = 0) THEN
        DBMS_OUTPUT.put_line(
            table_rec.table_name || TB ||
            table_rec.tablespace_name || TB ||
            table_rec.initial_extent || TB ||
            table_rec.next_extent || TB ||
            table_rec.max_extents);
    END IF;
    get_table('scott', 'garbage', table_rec, status);
    IF (status = 0) THEN
        DBMS_OUTPUT.put_line(
            table_rec.table_name || TB ||
            table_rec.tablespace_name || TB ||
            table_rec.initial_extent || TB ||
            table_rec.next_extent || TB ||
            table_rec.max_extents);
    END IF;
END;
/

```

服务器返回下面的信息：

```

TABLE    TABLESPACE    INITIAL NEXT    MAX
-----
DEPT     USER_DATA        10240  10240  121
get_table: ORA-01403: no data found
OWNER = <SCOTT>
TABLE = <GARBAGE>
PL/SQL procedure successfully completed.

```

如果希望使用一个唯一键的精确匹配，需要自己来管理游标并精确地执行依次提取。当发现没有记录时，在例外处理程序中关闭游标，而不是在条件语句块内部关闭游标（无论何都要在例外程序块中关闭游标，为什么要编写三遍程序呢？）。注意，必须引发预先定义的例外NO\_DATA\_FOUND，因为提取动作不会自动地生成一个例外。输入值被转换为大写形式，使用局部变量，因为被转换后的值将在几个地方用到。

还要注意例外处理程序显示的额外信息。为什么不抓住机会显示例外发生时的键值呢？当处理大量的记录时，这样做尤其有价值。这条信息还可以被转储到一个错误表，便于事后深入分析。

你也许认为，“我可以使用一条简单的 SELECT 语句得到相同的信息。这样做能给我带来什么？”。在较大的事物模式中，固定的查询更有效率，因为可以在公共 SQL 区找到它们，并且能够再次使用它们。手工控制游标的单行提取确实更有效率，尤其当游标一次又一次运行了几千遍时。记住，目标是编写效率高的应用。在得到记录后，可以程序化地做任何希望的事来处理它。你拥有全部的灵活性和控制权，还有底层曾经编写且可重用的过程。

清单28-27显示了一个在一个含有数字值的 PL/SQL 表上实现二进制查找的例子。

清单28-27 bintest.sql——过程使一个二进制查找程序易于使用

```

SET SERVEROUTPUT ON
DECLARE
  -- constants
  FIXED_TOP CONSTANT NUMBER := 12; -- fixed # of elements
  -- data types
  TYPE NUMARR_TYPE IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
  -- global variables
  numarr NUMARR_TYPE;
  isfound BOOLEAN;
  rowout NUMBER;

  -- routines
PROCEDURE binary_search( -- binary search on sorted array
  Parr    IN NUMARR_TYPE,
  Pnum    IN NUMBER,
  Pfound  OUT BOOLEAN,
  Prow    OUT NUMBER) IS
  local_found BOOLEAN := NULL;
  top BINARY_INTEGER := FIXED_TOP;
  bottom BINARY_INTEGER := 1;
  middle BINARY_INTEGER := NULL;
BEGIN
  local_found := FALSE;
  LOOP -- binary search
    middle := ROUND((top + bottom) / 2); -- find middle
    IF (Parr(middle) = Pnum) THEN -- exact match
      local_found := TRUE; -- match succeeded
      EXIT; -- break
    ELSIF (Parr(middle) < Pnum) THEN -- GO UP
      bottom := middle + 1;
    ELSE -- GO DOWN
      top := middle - 1;
    END IF; -- test for match
    IF (bottom > top) THEN -- search failed
      IF (Pnum > Parr(middle)) THEN
        middle := middle + 1; -- MAY BE OUTSIDE ARRAY!
      END IF; -- insert after
      EXIT;
    END IF; -- failed
  END LOOP; -- search
  Pfound := local_found;
  Prow := middle;
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(SQLERRM(SQLCODE));
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(middle));
END binary_search;

```

```

FUNCTION bool_to_char(Pbool IN BOOLEAN) -- convert Boolean to char
RETURN VARCHAR2 IS
  str VARCHAR2(5); -- capture string to return
BEGIN
  IF (Pbool) THEN -- test Boolean value for TRUE
    str := 'TRUE';
  ELSIF (NOT Pbool) THEN -- FALSE
    str := 'FALSE';
  ELSE -- must be NULL
    str := 'NULL';
  END IF; -- test Boolean value
  RETURN (str);
END bool_to_char;
BEGIN -- bintest executable code
  DBMS_OUTPUT.enable;
  numarr(1) := 100; -- fill array with numbers in order
  numarr(2) := 103;
  numarr(3) := 104;
  numarr(4) := 108;
  numarr(5) := 110;
  numarr(6) := 120;
  numarr(7) := 121;
  numarr(8) := 122;
  numarr(9) := 130;
  numarr(10) := 140;
  numarr(11) := 145;
  numarr(12) := 149;
  binary_search(numarr, 90, isfound, rowout);
  DBMS_OUTPUT.put_line('FOUND=' || bool_to_char(isfound) ||
    ', ROW=' || TO_CHAR(rowout) || ' SB=1');
  binary_search(numarr, 150, isfound, rowout);
  DBMS_OUTPUT.put_line('FOUND=' || bool_to_char(isfound) ||
    ', ROW=' || TO_CHAR(rowout) || ' SB=13');
  binary_search(numarr, 100, isfound, rowout);
  DBMS_OUTPUT.put_line('FOUND=' || bool_to_char(isfound) ||
    ', ROW=' || TO_CHAR(rowout) || ' SB=1');
  binary_search(numarr, 145, isfound, rowout);
  DBMS_OUTPUT.put_line('FOUND=' || bool_to_char(isfound) ||
    ', ROW=' || TO_CHAR(rowout) || ' SB=11');
  binary_search(numarr, 108, isfound, rowout);
  DBMS_OUTPUT.put_line('FOUND=' || bool_to_char(isfound) ||
    ', ROW=' || TO_CHAR(rowout) || ' SB=4');
  binary_search(numarr, 105, isfound, rowout);
  DBMS_OUTPUT.put_line('FOUND=' || bool_to_char(isfound) ||
    ', ROW=' || TO_CHAR(rowout) || ' SB=4');
END; -- bintest
/

```

服务器的输出如下所示：

```

FOUND=FALSE, ROW=1 SB=1
FOUND=FALSE, ROW=13 SB=13
FOUND=TRUE, ROW=1 SB=1
FOUND=TRUE, ROW=11 SB=11
FOUND=TRUE, ROW=4 SB=4
FOUND=FALSE, ROW=4 SB=4
PL/SQL procedure successfully completed.

```

注意OUT参数模式，它意味着只输出。这表示在过程内部，只能写入这个变量。如果需要对一个参数变量进行读写操作，使用 IN OUT模式声明变量。

二进制到字符转换程序看起来有些面熟是不是？它是不是非常好，可以被复制到每个需要它的PL/SQL程序中？

#### 4. 缺省参数值

参数可以接收缺省值，当在实际调用子程序而没有提供参数值时，就要使用缺省值。这会让子程序看起来好像它具有参数变量表一样：

```
DECLARE
    ... -- types, constants, variables
    FUNCTION get_data (Pkey IN CHAR,
                      Pflag IN BOOLEAN DEFAULT FALSE,
                      Psort IN CHAR DEFAULT ' ')
    RETURN VARCHAR2 IS
    ... -- function implementation
BEGIN -- executable code
    IF get_data(key1) THEN -- valid call (Pflag, Psort defaulted)
    ...
    ELSIF get_data(key2, TRUE) -- valid call (Psort defaulted)
    ...
    ELSIF get_data(key3, , 'ASCENDING') THEN -- invalid!
```

注意关键字DEFAULT的使用。还可以使用赋值运算符（:=）。作为一个编程约定，只有DEFAULT是这样使用的，以使这个语义上与众不同的构造和声明中较为直截了当的赋值相区别。

可以有意地省略两种缺省参数，以便在第一个调用 get\_data 中，标志参数被默认为FALSE，并且排序参数被默认为空间。这样做是一种非常清楚的编程风格，可以在参数表中只定义你感兴趣的参数。但是要注意，不能跳过一个默认参数而直接提供下一个参数，因为定义参数的标记符是有位置的。参数的位置有重要意义，不能尝试使用一个占位符，例如额外的逗号。

#### 5. 位置和命名表示法

可以使用一个称为命名表示法的替换表示法以任意顺序定义参数。与值一起提供参数的名称。

```
ELSIF get_data(key3,
               Psort => 'ASCENDING') THEN -- valid (Pflag defaulted)
```

可以从左到右开始使用位置表示法，然后切换到命名表示法，这被称为混合表示法。一旦使用命名表示法，必须在后面的参数中保持它的名字。命名表示法可以被用于任意参数，而不仅仅是默认的参数。

```
ELSIF get_data(key3, Psort => 'ASCENDING',
               Pflag => TRUE) THEN -- right
    ...
ELSIF get_data(Pkey => key3, 'ASCENDING',
               Pflag => TRUE) THEN -- wrong!
```

虽然这看起来非常方便而且在编程语言中不同寻常，我可是从来不使用它。但是，如果有一组参数，并且几乎每个参数都是缺省参数，而且希望调用子程序具有最大限度的灵活性，这实际上是不可缺少的。一个例子是使用 Oracle 的 Web 应用服务器开发者工具箱，其中的过程具有许多参数，大部分在多数时间都不会用到。

#### 6. 内建函数

几乎全部在 SQL 中使用的内建函数和运算符都可以用于 PL/SQL 的表达式和过程化语句中，只有极个别例外不符合这个规则。

在一个表达式中不能使用 ‘=ANY (...)’，而要以下列形式使用 IN 运算符：

```
IF (key IN ('A', 'B', 'C')) THEN -- acts like an OR conditional
```

还可以使用其他运算符，例如 BETWEEN、IS NULL、IS NOT NULL、LIKE 等等。

不能在过程化语句中使用 DECODE，而且，在过程化语句中不允许使用任何 SQL 组函数，它们在这种环境下没有使用的意义。当然，对于 PL/SQL 中嵌套的 SQL 语句，没有更多的限制。

已经看到如何使用 SQLCODE 捕获数字形式的例外错误值。SQLERRM (sqlcode) 可以将与例外相关的 SQLCODE 值转换为错误消息字符串。所有其他的内建函数都差不多。

## 28.6 Oracle 8i 专有的功能

Oracle 的每个版本都致力于提高系统的性能和可用性，Oracle 8i 也不例外。除了其他方面的改进提高（例如 PL/SQL 表更易于使用）外，Oracle 还使 PL/SQL 的一些功能更加强大的特性易于使用或清楚快捷。

### 28.6.1 本地动态 SQL

有些时候不知道要查找些什么信息，直到实际操作时才知道需要的东西。例如，当工作时，也许不知道需要查看哪些表或需要使用什么查找条件。可以这么说，开发者已经拥有能力生成一个忙碌的查询，但是它是一个神秘的进程，要用到 RDBMS（关系型数据库管理系统）提供的 DBMS\_SQL 包。但是使用 Oracle 8i，处理过程被大大地简化了，如下所示：

```
PROCEDURE INSERT_ITEMS (Ptable varchar2, Pprod varchar2, Pdesc varchar2) is
  sql_statement varchar2(500);
BEGIN
  sql_statement := 'insert into '||p_table||' values (:prod_id, :desc)';
  EXECUTE IMMEDIATE sql_statement USING Pprod, Pdesc;
END;
```

如果使用 DBMS\_SQL 完成同样的任务，则需要几十行的程序代码！本地动态 SQL 将给你一个显著的性能改善。

### 28.6.2 成批捆绑

使用 PL/SQL 的一个好处是可以将发送到服务器的一系列语句包含到一个组中这一事实。虽然如此，当服务器在 PL/SQL 引擎与 SQL 引擎之间移动时，嵌套在 PL/SQL 代码中的 SQL 语句还是需要环境的切换。为了利用这个新特性，语句：

```
FOR i in 1..10000
LOOP
  INSERT INTO orders VALUES (hold_orders(i).cust_id, hold_orders(i).prod_id);
END LOOP;
```

可以被写作：

```
FORALL I in 1..10000
  INSERT INTO orders VALUES (hold_orders(i).cust_id, hold_orders(i).prod_id);
```