第三部分　利用Developer编制程序

第11章　PL/SQL基础

本章介绍如何使用 Oracle Developer在原型和设计的基础上建立应用程序。 PL/SQL是Oracle对SQL的过程化扩展。换句话说， PL/SQL使你不仅能执行SQL，也能将其嵌入到控制结构中，如IF-THEN-ELSE和循环结构。与只使用SQL相比，可以使用户在应用中完成更复杂的工作。PL/SQL的语法来自复杂的Ada程序设计语言，但实际上用起来要简单得多。

本章介绍PL/SQL程序设计语言的基础，包括如何组织数据、如何用控制结构编程和如何将SQL作为语言的一部分使用。第 12章将详细讲述使用更复杂的程序结构，这种结构可将程序组织为可重用性代码。第13章介绍Oracle Developer中的调试工具。读完本书的这一部分后，将掌握如何创建、测试和调试应用程序。

本章着重介绍PL/SQL基本编程结构。在Oracle Developer中，可以在两种结构中放置用户代码：trigger(触发器)和program unit(程序单元)。Developer的事件激活触发器，依次调用程序单元。本章讲述如何编写基本触发器和程序单元代码。第 12章介绍如何编制更复杂的程序单元(procedures、function和package)。也可以使用这部分的知识在 Oracle数据库中开发存储程序单元(数据库触发器、过程和包)。第12章的最后一部分讨论如何将程序单元打包为程序库，使不同的应用可以重用这部分代码。

注意　Oracle Developer使用本地PL/SQL版本编译程序单元。Oracle数据库服务器使用当前Oracle数据库服务器PL/SQL的版本编译程序单元。这两个版本的PL/SQL可能不同，所以某些功能在服务器PL/SQL上可以使用而在Oracle Developer PL/SQL上不能使用。例如，在Oracle数据服务器上使用版本为7.3.4，那么在服务器上的PL/SQL版本与Oracle Developer PL/SQL 8相比缺少许多工具。通常，虽然这只对高级用法有影响，如在代码中使用嵌套表，但是需要了解代码的假定，这些假定是说明建立代码的基础是什么。

PL/SQL基本程序单元结构如下：

```
DECLARE
  --Data declarations
BEGIN
  null;--Program statements
EXCEPTION
  --Exception handlers
  WHEN OTHERS THEN
    null;--default handler
END;
```

带有这种结构的完整程序是一个无名块。整个块是一个 PL/SQL语句，因此，在END后需要分号终结符。如果没有数据说明，可以省略 DECLARE、BEGIN和END关键字。如果没有异常处理，可以省略EXCEPTION关键字和WHEN子句。通常包含BEGIN-EXCEPTION-END

序列结构，而不仅仅只是EXCEPTION子句。在块中至少要包含一个有效的程序语句，因此上面的代码中使用了null(空)语句。在本章的代码例子中，为了使例子能够编译，如果块中没有代码，则其中都包含一个null语句。

注意 PL/SQL块与Oracle Developer Forms的数据块完全不同。PL/SQL块是一段可执行程序(一个无名块、一个过程或一个函数)，而Forms数据块是在一个表单中项定义的集合，这些项可能映射、也可能不映射基准服务器表定义。可以把Forms的数据块看作是记录的列表。在PL/SQL块中可以包含任何级的嵌套PL/SQL块，根据每个块命名的作用域限制所定义嵌套块中的块。PL/SQL是象Ada语言一样的块结构程序设计语言。

下面是应该知道的几个实际问题：

- 大多数PL/SQL表达式使用与SQL表达式一样的语法，可以使用大多数 Oracle SQL内置的函数。如果了解SQL,就了解大部分的PL/SQL。
- PL/SQL不区分大小写字母，所以可以使用大写字母、小写字母，或者为改进可读性而用大小写字母混合编写代码。在引号中的字母 (字符作为其本身使用而不是用作其他东西的符号)区分大小写。
- PL/SQL标识符必须以字母开头，并且只能包含数字、下划线、符号 (#)和美元符号($)。标识符最多为30个字符，并且不能是系统的保留字，如 DECLARE或END。
- 在引号内的字符串中，用两个单引号表示一个单引号，如 'Talbot' s Farm'。
- BOOLEAN型(逻辑型)数据的值为 TRUE(真)、FALSE(假)或NULL(空值)，不用引号。例如IS_WINDOWS_DISPLAYED:=TRUE，不使用'TRUE'。
- 可以使用两个横线注释一行或使用一对 /*-*/之间的多行注释：

```
null;      -- This is an inline comment terminated by the line end
/* This is
a multiple-line
comment terminated by */
```

提示 使用行内注释要优于多行注释。因为很容易忘记注释终结符*/或者可能在编写代码时删除了注释对的某一边。

如果大量使用PL/SQL编程，参见Scott Urman的《Oracle 8 PL/SQL程序设计》一书。这本书的参考资料中包含了各种内部函数的完整列表。

11.1 数据概述

关于程序设计语言需要理解的最重要的问题也许就是这种语言如何组织数据。有的程序设计语言有非常复杂的类型系统，而有的却十分简单。有的语言可以扩展类型系统，而有的却不能。

11.1.1 数据类型

数据类型是对数据的分类，如字符、数值或日期。 PL/SQL的程序数据类型与SQL数据类型一致，并扩充了SQL数据类型。由于PL/SQL最主要的目的是嵌入SQL，所以希望直接映射到SQL语句和其中值的类型。 PL/SQL也增加了几种在块程序设计语言中有用的数据类型：BINARY_INTEGER(二进制整数型)、BOOLEAN(逻辑型)、RECORD(记录型)和TABLE(表型)。

Oracle Developer中使用的PL/SQL增加了几种类型与表单中不同的对象相对应，例如 window (窗口型)和block(块型)。

注意 CURSOR和REF CURSOR是可以与SQL语句联系的特殊数据类型，细节参见11.3节“ 在PL/SQL中使用SQL ”。

1. SQL类型

PL/SQL提供了与SQL一一对应的类型系统，至少是重要 SQL类型是这样。以下这些数据类型的细节参见联机文档： DECIMAL、FLOAT、INTEGER、NUMBER、REAL、SMALLINT、DATE、DATETIME、CHARACTER、LONG、RAW、STRING、VARCHAR和VARCHAR2。在PL/SQL块中可以使用标准Oracle SQL转换函数进行这些类型的显式转换。

2.PL/SQL二进制整数类型和布尔类型

为了处理带符号整型数，PL/SQL增加了三种数据类型。当在PL/SQL中使用带符号整型数进行计算时，不需要将数据像 NUMBER或其他SQL数据类型一样转换为内部格式。在整数运算量很大的块中，使用带符号整数代替 NUMBER数据可以提高性能。

基本的带符号整数类型是 BINARY_INTEGER，可以表示从-2147483647~2147483647范围的整数。两个子类型的范围要小一些。 NATURAL(自然数)类型可以表示从0~2147483647范围的整数，POSITIVE(正数)类型可以表示从1~2147483647范围的整数,可以使用这些数据类型限定非负数值。

BOOLEAN类型可以直接处理TRUE和FALSE。也可以把BOOLEAN数据置为NULL，意思是其值没有定义。这意味着PL/SQL Boolean值有三种可能的状态，TRUE、FALSE和NULL，其逻辑是三值逻辑。

注意 三值逻辑可以是反直观的(counterintuitive)。例如，比较两个Boolean变量，两个中的一个或两个都为NULL，结果为NULL，而不是TRUE。如果在IF语句中表达式值为NULL，则语句执行ELSE子句。第三个例子：NOT(NULL)等于NULL。关于三值逻辑及其在SQL中的使用的讨论参见Chris Date 的《Relational Database: Selected Writings》(Addison—Wesly，1986)第15章中的“ Null Values in Database Management ”(在数据库管理中的空值)部分。数据库管理器不是一贯地采用三值逻辑的一种模型，所以要仔细查阅Oracle文档，理解在特定情况下会发生什么。

3. PL/SQL的记录

用PL/SQL记录可以定义包含几个数据元素的单一变量，很像数据库中的行。可以使用RECORD类型创建任何类型的结构化数据。然后可以使用圆点符号像逻辑单元一样引用数据。

创建记录，必须首先创建一个独立的记录类型。例如，创建一个对应于部分 Ledger表的类型，可以在DECLARE(声明)部分采用如下语句：

```
DECLARE
  TYPE LedgerType IS RECORD (
    LedgerID VARCHAR(25) NOT NULL := 0,
    ActionDate DATE,
    Action VARCHAR(8),
    Amount NUMBER(9,2));
BEGIN
  null;
END;
```

在这里，记录只描述表的一部分数据。 NOT NULL(非空)子句与SQL中NOT NULL一样，阻止用户给域赋值 NULL。尽管RECORD类型常见的应用是描述与表数据相对应的数据，但是在其中可以包含任意类型的域，包括 RECORD类型。可以在程序包中使用记录。例如描述未存储在数据库中的运行时对象，如一天 24小时的特定时间。可以用对记录中数据进行操作的一组函数封装类型，那样，就可以代入时间值和时间间隔的值去调用 Subtract函数求出它们的差值。

```
PACKAGE Time24Package IS
  TYPE Time24Type IS RECORD (
    hour NATURAL NOT NULL := 0,
    minute NATURAL NOT NULL := 0,
    second NUMBER NOT NULL := 0);

FUNCTION Subtract(time Time24Type,
                  interval TimeIntervalPackage.tTimeInterval)
    RETURN Time24Type;
END;
```

可以说明两个Time24Type类型的变量，把它们传递给函数，取回差值，把它赋给第三个Time24Type类型的变量。这种封装和抽象数据类型可以描述比简单 PL/SQL类型更复杂的对象。第12章给出了关于封装和抽象数据类型的完整的讨论。

注意 参见11.1.3节“使用类型属性”，该节介绍了使域类型与数据库列类型一致的方法。

一旦定义了类型，就可以使用类型创建变量，在 DECLARE部分也一样：

```
DECLARE
  TYPE tLedger IS RECORD (
    vLedgerID VARCHAR(25) NOT NULL := 0,
    vActionDate DATE,
    vAction VARCHAR(8),
    vAmount NUMBER(9,2));
  vLedgerItem tLedger;
BEGIN
  null;
END;
```

这个记录包含一个值为 0的LedgerID域和其余值为NULL的域。PL/SQL在执行DECLARE部分时创建变量，在执行定义变量的块的 END时释放变量。如下面代码段中所示，可以给域赋值，使用圆点符号引用域：

```
IF vLedgerItem.vAction = 'Sold'
THEN vLedgerItem.vAmount := 0.0;
END IF;
```

注意，比较操作符为“ = ”，然而赋值操作符是“ := ”。

4. PL/SQL表

尽管RECORD类型功能强大，但是缺乏一个功能：在同一时刻描述多于一行的数据的能力。这就是 TABLE类型的工作。大多数程序设计语言都有这种多值 (multiple-value)数据类型,PL/SQL也不例外。

Array(数组)在程序设计语言中是表示多值的变量，其值标为元素可以通过一个整数下标访问它。数组第一个元素的下标是 1，数组第二个元素的下标是 2，以此类推。有的语言的数

组下标从0开始，而不是从1开始，还有一些是任意值，包括负数下标。 PL/SQL的表像数组一样通过下标访问。与大多数程序设计语言的数组类型不同的是，用户完全控制下标和分配给元素的存储器。在给使用某个下标的元素赋值时，如果元素不存在， PL/SQL则创建它。因此可以建立一个包含五个任意下标值 (如 - 4、33、1275、2和 - 43)并且有五个元素的数组。访问一个未用这种方法创建的元素，将得到 Oracle错误代码 - 1403 —— “ 没找到数据 ”。

Linked list(连接列表)在程序设计语言中是描述通过指针 (first指向第一个、last指向最后一个、next指向下一个或previous指向前一个)访问元素的数据结构。 PL/SQL表在存储上像一些链接列表，只有加入到表中的元素才存在。与链接列表相似，可以使用表属性导航表。与链接列表不同的是，如果知道一个元素的下标，可以直接找到这个元素。

PL/SQL的表将数组和连接列表的最好的特点结合起来。可以使用表来达到许多不同的目的。主要的是在 PL/SQL子程序中描述多值数据和在 PL/SQL代码中描述查询返回的结果集合。例如表使用户在存储过程中建立一个完全的结果集合，并且通过 IN OUT参数返回给调用者。

注意 Oracle Developer Forms有一个类似的结构叫做record group(记录组)。然而这些对象只在一个表单中存在，因为它们不是PL/SQL语言不可或缺的一部分，必须使用内部子程序访问数据元素。在通用PL/SQL编程中，使用表代替记录组。但有些功能需要使用记录组，如建立动态列表(第7章)或传递Oracle Developer参数列表(第10章)。

1) 声明表 和记录一样，用声明表类型来声明表，然后声明该种表类型的变量。这使表对于打包程序设计格外合适 (参见第12章)。可以通过程序包的说明导出一个表类型，然后在引用程序包的 PL/SQL代码中声明满足所需数目的这种类型的表。

可以声明一个简单的数值表：

```
DECLARE
  TYPE tLedgerAction IS TABLE OF Ledger.Action%TYPE INDEX BY BINARY_INTEGER;
  vActionTable tLedgerAction;  --declare an action table
BEGIN
  null;
END;
```

在这种格式中需要 INDEX子句，因为PL/SQL除了BINARY_INTEGER索引以外还不支持其他的索引。也可以声明一个更复杂的使用记录类型的表：

```
DECLARE
  TYPE tLedger IS TABLE OF Ledger%ROWTYPE INDEX BY BINARY_INTEGER;
  vLedgerTable tLedger;  --declare a Ledger table
BEGIN
  null;
END;
```

可以在TYPE声明的OF子句中使用任何记录类型，不仅是 %ROWTYPE派生记录。

2) 用下标和属性使用表 一旦声明了表类型和这种类型的变量，就可以通过给它们赋值建立表中的元素。在使用下标给元素赋值之前， PL/SQL并不真正建立任何元素。

例如，通过直接给 vLedgerTable变量赋4种总帐动作 (action)的方法，建立一个可能值的列表。

```
DECLARE
  TYPE tLedgerAction IS TABLE OF Ledger.Action%TYPE INDEX BY BINARY_INTEGER;
  vActionTable tLedgerAction;  --declare an action table
BEGIN
  vActionTable(1) := 'Bought';
  vActionTable(2) := 'Sold';
  vActionTable(3) := 'Paid';
  vActionTable(4) := 'Received';
```

```
  -- Use the table in further processing
END;
```

在表名后面的括号中指定下标，引用表中特定元素。在这个例子中，通过下标 1、2、3、4在表中建立了四个元素。

注意 可以使用任何下标值，不只是顺序的或从1开始的序列。然而，为了便于记忆和在循环中编程，最好使用这些序列。可以使用SQL来填充表(参见11.3节“在PL/SQL中使用SQL”中关于CURSOR和循环语法的细节)：

```
DECLARE
  TYPE tLedgerAction IS TABLE OF Ledger.Action%TYPE INDEX BY BINARY_INTEGER;
  vActionTable tLedgerAction;  --declare an action table
  CURSOR cLedger IS SELECT DISTINCT Action FROM Ledger;
BEGIN
  FOR vLedgerAction IN cLedger LOOP
    vActionTable(cLedger%ROWCOUNT) := vLedgerAction.Action;
  END LOOP;
END;
```

这个例子从Ledger表中通过一个cursor和特定cursor FOR loop选定Action列的DISTINCT(不同的)值，用特殊的cursor ROWCOUNT属性做为下标把检索值赋给动作(action)表。这和前面的例子的结果相同，但是更灵活。如果表中值的可能集合改变了，第一个例子必须重做，而第二个不用。

也可以通过表11-1所列出的表属性访问表。

表11-1 PL/SQL表属性

属 性	类 型	描 述
Count	NUMBER	返回表中元素的个数
Delete		删除表中所有的元素
Delete(index)		删除表中index指定的元素
Delete(index1,index2)		删除表中所有在index1和index2之间的元素(包括index1和index2)
Exists(index)	BOOLEAN	返回在表中是否存在index指定的元素
First	BINARY_INTEGER	返回表中第一个元素的下标
Last	BINARY_INTEGER	返回表中最后一个元素的下标
Next(index)	BINARY_INTEGER	返回表中下标指定的元素的下一个元素的下标
Prior(index)	BINARY_INTEGER	返回表中下标指定的元素的前一个元素的下标

将表名和圆点符加上表属性来使用表属性。例如，为取得表中元素 4的前一个元素的下标，使用以下语法：

```
vIndex := vActionTable.Prior(4);
```

使用这些属性的例子，参见《PL/SQL User’s Guide and Reference》或Scott Urman的书《Oracle 8 PL/ SQL程序设计》。

5. Oracle Developer的PL/SQL对象类型

在Oracle Developer中使用PL/SQL时,有几种描述不同对象的类型。表 11-2列出了Forms中不同的对象。

表11-2 Form PL/SQL对象类型和查找函数

对 象	类 型	查找函数
Alert	ALERT	FIND_ALERT
Block	BLOCK	FIND_BLOCK

(续)

对 象	类 型	查找函数
Canvas	CANVAS	FIND_CANVAS
Editor	EDITOR	FIND_EDITOR
Form	FORMMODULE	FIND_FORM
Item	ITEM	FIND_ITEM
List of Values	LOV	FIND_LOV
Menu item	MENUITEM	FIND_MENUITEM
Parameter List	PARAMLIST	GET_PARAMETER_LIST
Record Group	RECORDGROUP	FIND_GROUP
Record Group Column	GROUPCOLUMN	FIND_COLUMN
Relation	RELATION	FIND_RELATION
Report	REPORT	FIND_REPORT
Tab Page	TAB_PAGE	FIND_TAB_PAGE
Timer	TIMER	FIND_TIMER
View	VIEWPORT	FIND_VIEW
Window	WINDOW	FIND_WINDOW

在调用带参数内部子程序时，通常要提供对象的唯一名称或对象 ID。对象ID是唯一识别对象的对象句柄。当使用表 11-1中的类型声明一个变量时，实际上是声明了一个对象 ID句柄。通过调用FIND函数，可取得对象 ID，FIND函数在你传入对象名时返回 ID。因此可以用对象名得到对象标识符 ID，然后在一系列子程序中使用 ID。如果多处引用对象时，这可以加快处理的速度，因为每次传递对象名时， Oracle Developer内部都要查找其ID。例如，在循环处理中有一段涉及item(项)值的代码。应用FIND_ITEM()函数在进入循环之前取得 item的ID，然后在循环中使用item ID来查阅item 的值，而不是在每次循环中通过 item名字来查阅 item。

参数列表对象有一个特殊的 ID类型PARAM_LIST，对应GET_PARAMETER_LIST函数而不是用FIND开始的函数。

表11-3列出了Graphics数据类型和其对应的 GET函数，尽管名称约定不同，但与 Forms类型的工作方式基本一样。

表11-3 Graphics对象类型和查找函数

对 象	类 型	查找函数
Axis	OG_AXIS	OG_GET_AXIS
Button Procedure	OG_BUTTONPROC	OG_GET_BUTTONPROC
Chart Template	OG_TEMPLATE	OG_GET_TEMPLATE
Display	OG_DISPLAY	OG_GET_DISPLAY
Field Template	OG_FTEMP	OG_GET_FTEMP
Graphic Object	OG_OBJECT	OG_GET_OBJECT
Layer	OG_LAYER	OG_GET_LAYER
Query	OG_QUERY	OG_GET_QUERY
Reference Line	OG_REFLINE	OG_GET_REFLINE
Sound	OG_SOUND	OG_GET_SOUND
Timer	OG_TIMER	OG_GET_TIMER
Window	OG_WINDOW	OG_GET_WINDOW

注意 报表没有对象类型和对象ID。

11.1.2 声明变量和常量

在本书的前面部分中已经看到了几种变量声明。变量声明按如下形式在 DECLARE部分中声明：

```
variable_name  type_name  [NOT NULL] [:= initial_value];
```

也可以在子程序(过程或函数)的参数列表中声明变量。在子程序 (过程或函数)的参数列表中声明变量，参见第13章。

NOT NULL限定词说明变量的值不能为NULL(空值)。初始值的指定给指定为文字或表达式的值设置变量。如果指定 NOT NULL，则必须指定初始值。初始化所有的变量是一个很好的方法，这样可以确保用户标识符在第一次被引用时有一个有效的值。也可以在赋值时使用保留字DEFAULT，用:DEFAULT initial_value代替:=initial_value。

可以通过宿主变量引用Oracle Developer对象。宿主变量是嵌入应用的一个变量名，通过在变量名前加上冒号 (:)来引用它。可以在使用 PL/SQL变量的任何地方使用宿主变量，包括SQL语句内和语句外。

```
BEGIN
  /* Update the age in the block field and the database. */
  :Person.Age := :Person.Age+1;
  UPDATE Person SET Age = Age+1 WHERE Name = :Person.Name;
END;
```

PL/SQL和所有的Oracle Developer产品的类型一样，所以宿主变量在 PL/SQL过程中不需要单独的声明。PL/SQL把它们看作是已经声明的在item定义中指定的类型。

注意 Oracle Developer对象使用VARCHAR2类型表示文本值。CHAR、VARCHAR、VARCHAR2之间的不同在特定的情况下是十分重要的。CHAR描述定长度数据。如果给一个CHAR变量赋一个比规定值的长度要长的值，PL/SQL就会填入空白字符串，其长度是变量的长度。VARCHAR和VARCHAR2一样都不用这种方式填充。在赋值和比较时可能会有问题。如果给CHAR变量赋一个比其长度要短的值，PL/SQL会填充这个字符串。如果在VARCHAR2变量中有末尾空格，当它与其他任意一种类型的字符串比较时，PL/SQL会把它们看作是比较的一部分，但是并不填充其他字符串。如果比较两个CHAR变量，PL/SQL在比较之前会把比较短的一个填充到与比较长的一样长。这意味着如果在比较长的字符串中有末尾空格，PL/SQL实际上将通过为比较短的字符串填充空格忽略它们。一般来说，在大多数情况下，应该使用 VARCHAR2，避免使用CHAR和VARCHAR。

常数是在声明时赋值并且以后不能再赋值的用户标识符。用 CONSTANT关键字声明常量：

```
constant_name   CONSTANT   type_name   := value;
```

PL/SQL要求初始化常量，常量的初始值是常量整个生存期的值。

11.1.3 使用类型属性

PL/SQL类型属性是一个修饰符，用来取得一个对象在声明其他对象时信息。% TYPE属性可以取得变量、常量或数据库列的类型。% ROWTYPE属性可以取得数据库表或 SELECT Cursor结果表的所有列的类型。尽可能地使用% TYPE属性，这可以大大地提高程序的可

维护性。

可以使用%TYPE说明与另外的变量或常量同一类型的变量。也可以用它来说明与数据库列类型相同的变量，因为通常要在变量中存储与列类型一样的值。

下面的例子代码从数据库中检索一个值，增加这个值，并把结果赋给一个与其类型一样的新变量。

```
DECLARE
  vDBAge      Person.Age%TYPE;    -- database column type
  vNewAge     vDBAge%TYPE;         -- type from dbAge
BEGIN
  SELECT Age INTO vDBAge FROM Person
   WHERE Person.Name = :Person.Name;
  vNewAge := vDBAge + 1;
END;
```

注意 即使利用NOT NULL属性声明一个数据库列，PL/SQL也不会给变量加上NOT NULL限定词。如果要防止给变量赋予NULL，必须明确地说明。

可以使用%ROWTYPE说明能够创建一个包含表所有列的记录变量。例如，在一个触发器中为了进一步的处理，要从数据库中检索 skills(技能)和abilities(能力)。下面的声明创建了一个保存WorkerHasSkill表所有列的记录。

```
DECLARE
  hasSkillRecord   WorkerHasSkill%ROWTYPE;
BEGIN
    -- Retrieve records and process; for example,
hasSkillRecord.Name := 'Gerhardt Kentgen';
END;
```

%ROWTYPE属性也可以应用于对cursor SELECT结果产生的表。使用%ROWTYPE，可以创建一个PL/SQL自动正确格式化的、能够存放 SELECT返回的任何数据的记录。如果使用%ROWTYPE，不需要声明每个变量的类型，使维护更简单。例如，要对 Ledger表中按person分组求和的amount的进行检索，可以按如下方式说明变量：

```
DECLARE
  CURSOR sumCursor IS SELECT Person, SUM(Amount) Total
      FROM Ledger GROUP BY Person;
  sumRecord   sumCursor%ROWTYPE;
BEGIN
   null;  -- Retrieve rows and process
END;
```

当按这种方式使用%ROWTYPE时，必须为SELECT列表中的任一个表达式指定一个别名，比如在sumCursor SELECT中的TOTAL一样。这个别名在PL/SQL创建的记录中成为字段的变量名。关于cursor(游标)的更多细节，参见11.3节“在PL/SQL中使用SQL”。

如果两个%ROWTYPE变量来自同一个table或cursor，可以把一个变量赋给另外一个变量。不能把用% ROWTYPE创建的记录赋给用 RECORD类型创建的变量。除了使用 cursor以外,RECORD类型与%ROWTYPE声明的使用方式基本相同。 RECORD的一个缺点是不能从数据字典自动创建其结构。优点是可以为其指定任意的结构，包括不对应数据库列的附加字段。

11.2 程序控制简介

程序设计语言的控制语句是程序设计语言的主要特点， PL/SQL尤其是这样。控制结构将

面向过程程序设计语言区别于面向说明程序设计语言（如SQL)。在PL/SQL中嵌入SQL，结合了数据说明语言和控制结构及SQL语句的优点。

控制结构有三种：顺序结构、选择结构和循环结构。

顺序结构是程序语句的简单顺序。PL/SQL按顺序执行顺序结构，每次一条。SQL工作在语句接语句(statement-by-statement)的基础上，SQL语言每次只理解一条单个的语句。PL/SQL处理一系列语句，并且有变量连接到语句。这为结构化处理提供了更大的灵活性，用户可以把处理分割为多个步骤，包括复合的SQL语句。

选择结构是根据真或假的逻辑条件决定程序分支的语句。使用选择结构为进一步处理做判断，这在早期的SQL中是无法实现的。尽管DECODE函数提供了在行接行(row-by-row)基础上有限的判断能力，但是数据说明语法的复杂条件很快就超过了其能力。

迭代结构是控制重复执行的循环过程的语句，SQL作为内部机制，为循环提供了几种途径。除了对结果表的多行的循环外，SQL嵌套了子查询，可以在高层结构中实现循环，这是使用谓词演算操作符(例如ANY和ALL，EXISTS和IN)的强大的功能。SQL相关的子查询可以在row-by-row的基础上测试外部查询的行与嵌套子查询的行。即使有这些能力，但也还没有复杂到很容易处理所有可能在数据库应用程序设计中遇到的循环情形。循环控制结构提供了检索数据库中的行的能力，也提供了可以用更多的SQL语句处理它们的能力。这比嵌套查询或连接一个单一的SQL语句要更简单、更快速。循环也使得可以在应用程序级完成许多SQL不适用的标准程序设计任务。

注意 PL/SQL也提供了非条件分支语句(带语句标号的GOTO)，应该尽量避免使用它，除非它能使程序更清晰。例如，在冗长的过程中，有时可能会有非常复杂的选择和循环结构，如果简单地使用GOTO来代替，可能会使程序更容易理解。这种情况很少见。例如，简化错误处理的标准情况，通过分支到ERROR标号比异常(exception)处理要简单，参见第12章。

11.2.1 条件控制

PL/SQL为程序执行的选择结构提供了IF-THEN、IF-THEN-ELSE和IF-THEN-ELSIF-ELSE结构。IF-THEN结构检测条件，如果条件成立，则执行一个代码块。例如，下面的触发器块检测当前工人的年龄并显示一个警告窗(alert)，然后通过引发FORM_TRIGGER_FAILURE异常导致触发器失败。

```
DECLARE
  vAlertButton NUMBER;
BEGIN
  IF :Person.Age < 16 THEN
    vAlertButton := Show_Alert('Under_Age_Alert'); --Display error
    RAISE FORM_TRIGGER_FAILURE;
  END IF;
END;
```

注意 关于RAISE语句的更多信息参见第12章“异常和异常处理”部分。

如果有两个程序块，一个在条件判定为TRUE时执行，另一个在条件判定为FALSE时执行，可以使用IF-THEN-ELSE结构：

```
DECLARE
```

```
  vAlertButton NUMBER;
BEGIN
  IF :Person.Age < 16 THEN
    vAlertButton := Show_Alert('Under_Age_Alert'); --Display error
    RAISE FORM_TRIGGER_FAILURE;
  ELSE
    vAlertButton := Show_Alert('Over_Age_Alert'); --Display error
  END IF;
END;
```

如果判定条件有几个部分，可以在ELSE子句中使用嵌套的IF语句，但更好的方法是使用ELSIF结构：

```
DECLARE
  vAlertButton NUMBER;
BEGIN
  IF :Person.Age < 16 THEN
    vAlertButton := Show_Alert('Under_Age_Alert'); --Display error
    RAISE FORM_TRIGGER_FAILURE;
  ELSIF :Person.Age > 70 THEN
    vAlertButton := Show_Alert('Over_Age_Alert'); --Display error
      RAISE FORM_TRIGGER_FAILURE;
  ELSE  -- Worker is between 16 and 70 years old, inclusive
  vAlertButton := Show_Alert('Age_OK_Alert'); --Display info alert
  END IF;
END;
```

在这个选择结构中，第一个条件识别年龄小于17的值，第二个条件识别17以上(包括17)并且在70以上的值，其ELSE子句识别17到70之间的值，包括17和70。ELSIF子句全部过滤掉了前面的IF条件和ELSIF子句条件计算为假的情况。

注意 检测NULL值的规则与SQL相同。记住NULL不是一个值，而是不存在数据。因此，必须分别地使用IS NULL和IS NOT NULL表达式代替相等比较，这样来检测某些表达式的值是否为NULL。

11.2.2 重复控制

Oracle Developer嵌入了相当多的循环控制结构。Oracle Developer Forms通过程序块自动地检索记录组，在所有的级别上执行触发器。Oracle Developer Reports用一种非常复杂的循环结构在行和组上循环，报表数据模型通过它的组来驱动这种循环结构。Oracle Developer Graphic也采用循环过程取数据到转换到图表的表中。在触发器代码和程序单元中使用循环之前，应该搞清楚那儿已经存在什么。例如，可以把程序语句放到item(项)和record(记录)触发器中，Forms将以循环结构自动地执行它们。这比检索所有的记录然后在PL/SQL块中对它们进行循环要高效，而且更容易维护。可以在报表中把语句放到报表的page(页)触发器中，Reports将在每一页之后执行它们。PL/SQL控制结构在特定的触发器中提供了几种附加的控制，但是可能的话，应该尽量避免复杂性。

在PL/SQL中有三种循环控制结构：LOOP、WHILE和FOR。

LOOP语句，与EXIT WHEN语句一起使用，执行循环直到指定的条件为真。例如，在一个块中对所有的记录循环，每次age增加1，可以使用如下程序块：

```
DECLARE
  vCurrentPosition  VARCHAR2(10) := :SYSTEM.CURSOR_RECORD;
```

```
BEGIN
  FIRST_RECORD;
  LOOP
    :Person.Age := :Person.Age + 1;
    EXIT WHEN :SYSTEM.LAST_RECORD = 'TRUE';
    NEXT_RECORD;
  END LOOP;
  Go_Record(vCurrentPosition);    -- Reset cursor
END;
```

这种结构至少要执行循环语句一次，因为EXIT WHEN的检查在循环语句的后面。如果第一个记录是一个age为NULL的新记录，age的结果将会为NULL，即使经过一次LOOP循环，其值也不会改变。

如果在LOOP语句中嵌套LOOP语句，则应该为LOOP做标号，并在EXIT WHEN中引用标号，以便指定要退出嵌套堆栈中的哪个LOOP。

WHILE语句通过在执行循环语句之前检查条件来控制处理过程：

```
DECLARE
  vCurrentPosition  VARCHAR2 (10) := :SYSTEM.CURSOR_RECORD;
BEGIN
  FIRST_RECORD;
  WHILE :SYSTEM.LAST_RECORD != 'TRUE' LOOP
    :Person.Age := :Person.Age + 1;
    EXIT WHEN :SYSTEM.LAST_RECORD = 'TRUE';
    NEXT_RECORD;
  END LOOP;
  :Person.Age := :Person.Age + 1;
  Go_Record(vCurrentPosition);    -- Reset cursor
END;
```

在条件直接为假时，这种结构根本不执行控制结构中的语句。因此，例子中永远不会更新最后一条记录，所以在循环之后需要对最后记录做更新。和前面一样，如果 age为NULL，语句将会计算并赋值NULL给age,结果没有改变。

最后，FOR语句可以精确地控制循环的次数：

```
DECLARE
  vCurrentPosition  VARCHAR2(10) := :SYSTEM.CURSOR_RECORD;
  vLastRecord VARCHAR2 (10) := 0;
BEGIN
  LAST_RECORD;
  vLastRecord := :SYSTEM.CURSOR_RECORD;
  FIRST_RECORD;
  FOR I IN 1..To_Number(vLastRecord) LOOP
    :Person.Age := :Person.Age + 1;
    NEXT_RECORD;
  END LOOP;
  Go_Record(vCurrentPosition);    -- Reset cursor
END;
```

这种结构在指定的数值范围内执行循环。它也给出了通过在 FOR语句中声明的计数器中使用的值。可以使用该值来查找指定的记录、插入递增值的结果到数据库中，等等。

像例子中一样，可以用表达式指定范围的下限和上限，可以使用这一特性用循环外部的变量来约束循环范围，或者在运行时动态地确定循环范围。例如，通过 SQL语句取得表中的行数，然后按照行数循环读取所对应的行。然而，因为可以对游标使用更有效的% NOTF-

OUND属性(参见11.3节“在PL/SQL中使用SQL”)，所以这不是使用循环最有效的方式。

例子中通过设置cursor到末记录来取得范围的终值，并保存记录数。这使得可以按照正确循环次数对记录执行循环。与LOOP循环的例子一样，如果只有一条New记录，则将被更新为NULL。

这三个例子中，尽管LOOP循环可能是最直接的，但没有哪一种结构比其他结构更好。通常可以使用三种循环结构来作同样的事情。这样将会判断出那一种的代码行更少，并且判断出那一种结构更简单。作出上面判断的一种方法是看在代码正确的前提下，代码的简单程度：要考虑问题有多少？是单一判断还是复合判断？是否存在要考虑的特殊情况？例如，前面的WHILE代码必须在循环后使用一条语句来更新最后的行，看起来这好像将一件事分成为太多的部分。

注意 在“在PL/SQL中使用SQL”部分介绍FOR循环自动处理游标提取记录的特别版本。如果这种结构的功能能够完成想要做的事，那么它可以节省很大的程序量。

11.3 在PL/SQL中使用SQL

有三个原因要在Oracle Developer应用中使用PL/SQL：在事件处理中使用控制结构改变应用程序的行为；增加特别目的代码实现计算 (除了能够完成的计算项目之外)；在Oracle Developer工具提供的自动功能(很少使用)之外对服务器发出SQL语句。Oracle Developer以服务器上的打包过程的数据块为基础，完全不同于基于表或视图上的数据块。

注意 在PL/SQL中的SQL错误处理采用异常方式。第12章详细讨论了产生和处理异常，以及在错误处理中使用SQLCODE和SQLERRM内部函数。

11.3.1 简单的SQL

在PL/SQL中使用SQL最简单的方法是仅仅使用SQL。PL/SQL把SQL语句看作程序设计语言语句，并且接受几乎是任意的标准 SQL。许多触发器中包含简单的 INSERT、UPDATE或DELETE语句，也许能在应用程序中用宿主变量连接语句和 item。

在PL/SQL中有的SQL语句不能使用：

- 隐含事务结束的DDL语句(对于某些例外，参见11.3.4节“数据定义SQL”)，如CREATE TABLE、GRANT或ALTER VIEW
- 会话控制命令，如SET ROLE
- ALTER SYSTEM
- EXPLAIN PLAN

在触发器中应该避免使用在提交数据到数据库之后能够修改应用的 DML语句(INSERT、UPDATE和DELETE)，因为这样会导致不正确的结果。

警告 可以在PL/SQL程序中使用COMMIT(提交)、ROLLBACK(回滚)和SAVEPOINT(保存点)语句，但是对客户端代码来说这并不是一个好的办法。Oracle Developer运行系统对于事务处理有特殊的方法， COMMIT或ROLLBACK有可能中断事务处理。 Oracle Developer Forms不把COMMIT解释为隐式调用COMMIT_FORM过程，但是应该只在非常高级别代码中，在程序中明确的地方执行。因为 Oracle Developer事务处理很谨慎，

所以可以避免自己来完成令人头疼的维护工作。

一个特别的SELECT语法可以检索一个单行数据：隐式游标或单行选择。 ANSI(美国国家标准协会)标准要求SELECT INTO语句只返回一行，如果语句找到的符合标准的数据多于一行，则必定产生错误。

隐式游标在选择列表之后加上了一个INTO子句，其后的宿主变量映射选择列表中的元素。可以用%TYPE说明这些变量，或使用那些正确映射数据的项。

```
DECLARE
  vDBAge NUMBER := 0;
BEGIN
  SELECT Age INTO vDBAge FROM Person WHERE Name = :Person.Name;
  -- Use vDBAge to do something here.
END;
```

注意 INTO子句把PL/SQL变量作为用户标识符，而不是宿主变量，所以不用在变量名前加上冒号(:)。宿主变量:Person.Name指Person数据块中的Name项，而不是一个PL/SQL变量。

警告 使用INTO子句和隐式游标应该小心。每个这样的语句在网络中产生两次往返，其中一次用来确定只返回一行，另一次则返回这一行。可以用一个显式游标和提取回数据到变量中的FOR循环来代替隐式游标。这需要更多的代码，会更复杂，更难以维护，所以不得不做一个选择。通常只有在能够确定对性能的影响很小的时候，才使用隐式游标。如果在Internet上开发应用，最好不要使用隐式游标。

11.3.2 使用显式游标

当需要进行查询并且返回的数据多于一行时， PL/SQL中重要的SQL工作开始了。为了完成这项工作，必须声明并使用显式游标。显式游标是一个描述来自 SELECT语句的查询结果的PL/SQL数据结构。PL/SQL提供了许多工具使游标使用更容易。

注意 记住Oracle Developer Form中的块和块的记录以及报表中的数据模型自动处理这种事情。如果要开发描述重复的块或数据模型，可以使用这些自动的工具。显式游标尽管在一般PL/SQL编程中非常有用，但是在Oracle Developer编程中应用较少，因为Oracle Developer自动完成了大多数 SQL任务。可以使用显式游标建立比 Oracle Developer自动建立的更复杂的SQL。代价是必须重写对你无使用的自动功能。

显式游标的形式如下：

```
DECLARE
  CURSOR cSkill IS SELECT Skill, Description FROM Skill;
  vSkill cSkill%ROWTYPE;
BEGIN
  OPEN cSkill;
  LOOP
    FETCH cSkill INTO vSkill;
     EXIT WHEN cSkill%NOTFOUND;
     -- Process the row
  END LOOP;
  CLOSE cSkill;
END;
```

这两个声明为块建立了基本数据结构。游标包含 SELECT语句从Skill表检索所有数据。记录声明使用%ROWTYPE属性，根据SELECT的结构创建了一个记录。

OPEN(打开)和CLOSE(关闭)语句执行和终止查询。当打开游标时，执行查询，然后必须取回数据行。当关闭游标时，就不能再取回任何数据行了。打开游标后不要忘记关闭。否则将很快达到允许打开的最大游标数。

注意 允许打开的最大游标数有限制，这个限制通过 Oracle数据库服务器 OPEN_CURSORS初始化参数设定。如果从应用程序中作为SQL异常而出现了Forms错误FRM-40515，“ Oracle error:unable to open cursor ”，则需要越过这个参数，或者增大这个参数值，或者在Forms Runtime命令行中设置OPTIMIZETP=NO来共享游标。这会降低处理的速度。共享游标需要重新分析应用程序提交给该服务器的所有 SQL语句。OPEN_CURSORS参数依赖于操作系统，所以在所有的操作系统中有可能无法打开足够多的游标。

LOOP循环显示了提取数据的基本结构。这个游标的% NOTFOUND属性的变异用法测试是否达到了结果表的末尾。这与 SQL消息ROW NOT FOUND一样。

有一个捷径，可以使用 For Loop语句特别的选项实现这个目的：

```
DECLARE
  CURSOR cSkill IS SELECT Skill, Description FROM Skill;
BEGIN
  FOR vSkill IN cSkill LOOP
    null;-- Process the row
  END LOOP;
END;
```

这个块与上面那个较长的块做的事情完全一样。除非在显式游标控制提取、打开游标或关闭游标中需要复杂的逻辑，否则建议使用这种快捷的语法。

注意 记住，如果在SELECT语句中使用表达式，则必须提供一个在记录中能够作为字段名使用的别名。没有别名，无法引用字段。

有几种可以取得游标状态信息的游标属性。表 11-4中的多数属性对显式游标和隐式游标都适用。对显式游标，通过在游标名后加上属性名来使用其属性。对于隐式游标，使用隐式游标名“ SQL ”：例如， SQL%NOTFOUND或SQL%ISOPEN。

表11-4 PL/SQL游标属性

属 性	描 述
%NOTFOUND	当最后一次取数据行失败时为TRUE；使用这个属性检查循环是否已经取得最后一行数据
%FOUND	当最后一次取数据行成功时为TRUE；使用这个属性检查是否取得有效的数据来执行其他进一步的操作，例如插入数据到另一个表
%ROWCOUNT	当打开游标时为0，每取一行数据其值增加1
%ISOPEN	如果游标打开则为TRUE，否则为FALSE；如果有某些控制动作可能已经关闭游标，使用这个属性检测是否游标需要打开

注意 和SELECT语句一样，也可以对数据处理语句和 SELECT语句的隐式游标(INSERT、UPDATE和DELETE)使用这些属性。例如，%ROWCOUNT给出了这些语句的隐式游标的插入、更新或删除的行数。

可重用游标的一个重要功能是参数化游标的能力。可以在 SELECT语句中任何能够使用宿主变量或字符变量(参见11.1.2节“ 声明变量和常量 ”)的地方使用游标参数。当PL/SQL打开游标时，游标接受提供的参数，并把它们插入到 SQL语句。参数的作用域是游标，不能在游标中SELECT语句外的任何地方使用这些参数。下面的代码举例说明了参数化游标的使用：

```
DECLARE
  CURSOR cSkill (pAbbrev IN VARCHAR2 DEFAULT '%') IS
    SELECT Skill, Description FROM Skill
     WHERE UPPER(Skill) LIKE UPPER(pAbbrev);
BEGIN
  FOR vSkill IN cSkill('%Horse%') LOOP
    null; -- Process the row
  END LOOP;
END;
```

这个例子扩展了Skill查询，使其处理基于在 Skill表中模式查找的子集。在例子中使用了LIKE操作符表达式，查询查找出与表达式相匹配的记录。在这个例子中， FOR LOOP在模式中传递，该模式告诉SELECT查找所有包含词语“ horse ”的skill。注意在SELECT中的大写字母转换，这可以进行不区别大小写的比较，使游标的可重用性更好。另外，注意参数中的DEFAULT子句，这个子句提供了在没有参数传递时的显示模式，即“ 匹配任意字符 ”模式。

注意 参数化游标在把其作为程序包说明的一部分时十分有用，可以导出游标及其参数。这可以使外部程序使用游标，提供特定目的程序所需的参数。关于程序包的细节参见第12章。

11.3.3 使用游标变量

游标变量是游标的指针。严格地说，游标变量不是变量。在 PL/SQL代码中不能把游标传递给子程序的参数列表，也不能把不同的游标赋给 PL/SQL代码中的单个变量。游标变量可以完成这些。在Oracle Developer中这个特性的主要用途是把游标作为 IN OUT变量传递给存储过程。当执行查询的过程在服务器端时，这可以实现在客户端检索数据。

不能仅仅声明一个变量来指向一个游标。首先，必须声明一个类型，然后才能声明变量。声明一个引用游标类型，语法如下：

```
TYPE <type name> IS REF CURSOR [RETURN <record type>];
```

type name(类型名)是用来声明游标变量的名字。在 RETURN子句中可选的记录类型与游标SELECT列表相对应。通常使用% ROWTYPE构造记录类型。如果使用 RETURN子句，PL/SQL检查打开的SQL语句，确保返回的数据与已指定的类型相对应。否则，将使你忙于使SQL与处理语句相匹配。

可以使用变异的OPEN语句打开一个游标变量：

```
OPEN <cursor variable> FOR <select>;
```

OPEN语句把变量与特定的 SELECT(选取)联系起来。作为一个正常的游标，必须在声明部分与其SQL SELECT一起声明游标。作为一个游标变量，应该在执行代码中声明 SQL。

注意 不能在SQL语句中使用变量。这意味着，必须在PL/SQL中为SQL语句编写全部的代码，不能动态地构造和把它传递给游标。可以使用 DBMS_SQL包的动态SQL程序设计能力来动态地分析和执行语句，这个技术超出了本书所涉及的范围。参见11.3.4节

“数据定义SQL”中使用DBMS_SQL的例子。

下面是一个通过Skill包(关于程序包语法的细节参见第12章)查询与skill相关的信息的例子：

```
CREATE OR REPLACE PACKAGE SkillsPackage AS
    -- A record type with all the columns in the WorkerHasSkill table
  TYPE tSkill IS RECORD (
    vSkill WorkerHasSkill.Skill%TYPE,
    vAbility WorkerHasSkill.Ability%TYPE);
    -- A record type with the primary key column in the Skill table
  TYPE tSkillCursor IS REF CURSOR RETURN tSkill;
    -- A table type for a table of Skills
  PROCEDURE SelectCursor (pcSkill IN OUT tSkillCursor,
                          pPerson IN VARCHAR2);
END SkillsPackage;
```

下面的程序包体用一个简单的SELECT语句打开游标：

```
CREATE OR REPLACE PACKAGE BODY SkillsPackage AS
  PROCEDURE SelectCursor (pcSkill IN OUT tSkillCursor,
                          pPerson IN VARCHAR2) IS
  BEGIN
    OPEN pcSkill FOR
      SELECT Skill, Ability
        FROM WorkerHasSkill
       WHERE Name = pPerson;
  END SelectCursor;
END SkillsPackage;
```

这个包将记录类型、游标类型和创建游标的过程放在了一起。可以通过声明一个记录和游标变量并调用过程来使用它：

```
DECLARE
  vSkill SkillsPackage.tSkill;
  vSkillCursor SkillsPackage.tSkillCursor;
BEGIN
  SkillsPackage.SelectCursor(vSkillCursor, 'Adah Talbot');
  LOOP
    FETCH vSkillCursor INTO vSkill;
    EXIT WHEN vSkillCursor%NOTFOUND;  -- exit after last row fetched
  -- Process the skills of Adah Talbot
  END LOOP;
  CLOSE vSkillCursor;
END;
```

11.3.4 数据定义SQL

在PL/SQL块中能够执行的SQL语句只有SELECT、INSERT、UPDATE和DELETE语句。PL/SQL不允许使用其他嵌入式SQL语句，如：CREATE TABLE、DROP VIEW、COMMIT、GRANT或ALTER SESSION。这些是数据定义(DDL)语句——改变数据库模式或数据字典的语句。

PL/SQL在编译时绑定碰到的SQL对象到所用到的数据字典。系统的这个特性有两个主要的优点：早一点反馈SQL语句的问题和在编译时将SQL对象与其所用的数据字典绑定而带来的性能提高。然而，其限制是编译器不能处理不存在的对象。这意味着PL/SQL不能处理用于创建或破坏数据字典中对象的。

实际上，这个限制并不很严重。每个 Oracle Developer组件都有一个子程序，可以把任意SQL语句作为其标准程序单元的一部分来执行。另外， PL/SQL提供了支持动态SQL的DBMS_SQL包。

注意 当在应用中使用这些程序单元执行任意SQL语句时，必须清楚它们对事务处理的影响。DDL语句除了影响数据字典之外，会结束当前的事务处理。当执行DDL语句时，Oracle将提交任何未完成的改变给数据库，并移除会话对于数据库特有的任何锁从应用开发角度来看，这意味着必须要小心DDL语句的定时。只有在确保了没有未完成的改变和没有对数据库锁定时，才执行DDL语句。

在Oracle Developer Forms组件中可以使用Forms_DDL过程来执行DDL。例如，如果有一个使用DROP TABLE语句删除一个utility表的菜单项，可以在菜单的 PL/SQL块编写与下面类似的代码：

```
IF :System.Form_Status = 'CHANGED' THEN
  Message('Please save your changes before dropping the utility table.');
ELSE
  Forms_DDL('DROP TABLE UtilityTable');
  IF NOT Form_Success THEN
    MyErrorHandler(DBMS_Error_Code);
  END IF;
END IF;
```

检查表单的状态，确定在执行 DDL语句之前没有未完成的改变。调用 Form_Success内部函数来查看最后语句是否有未完成错误。需要把这段代码传递给用户标准错误处理机构，如例子中的My_Error_Handler过程，这个过程通知用户所发生的事情。

注意 当使用Forms_DDL时，不要用分号终止SQL语句。可以使用Forms_DDL提交一个PL/SQL块，使用标准语法，但不要象在SQL*Plus中那样用斜线终止它。

在Oracle Developer Reports中，使用SRW包中的Do_SQL过程来达到同样的目的。例如，要用下面触发器函数在 After Form触发器中删除utility表：

```
FUNCTION DropUtilityTable RETURN BOOLEAN IS
BEGIN
  SRW.Do_SQL('DROP TABLE UtilityTable');
  RETURN (TRUE);
EXCEPTION
  WHEN SRW.Do_SQL_Failure THEN
    SRW.Message(100, 'Error while dropping the utility table.');
    RAISE SRW.Program_Abort;
END;
```

对于Oracle Developer Reports来说，不存在提交未完成改变的问题。主要的区别是错误处理。它不使用函数和返回代码而是使用异常处理来处理问题。当 Do_SQL失败时，引发SRW包异常Do_SQL_Failure。异常处理程序 (参见第12章关于异常处理的细节)输出一个消息，并用SRW内部过程Program_Abort来终止报表。

在Oracle Developer Graphics显示模块中，可以通过内部 Do_SQL过程执行DDL语句。代码与Reports相似，但是没有对于过程的标准异常。

```
PROCEDURE DropUtilityTable IS
BEGIN
  Do_SQL('DROP TABLE UtilityTable');
END;
```

使用DBMS_SQL程序包代替组件特有的内置子程序来执行 SQL是一个好方法。使用DBMS_SQL提供的一组标准的管理SQL的工具。DBMS_SQL也使用标准SQL错误处理结构，所以可以为数据库的SQL错误编写错误处理代码。

DBMS_SQL的范围太大，本书难以尽述。使用DBMS_SQL的例子如下：

```
CREATE OR REPLACE FUNCTION DropUtilityTable RETURN BOOLEAN IS
  vCursorID INTEGER := DBMS_SQL.Open_Cursor; -- open a cursor for the DDL
  vDummy INTEGER; -- a dummy return value for Execute
BEGIN
  DBMS_SQL.Parse(vCursorID, 'DROP TABLE UtilityTable', DBMS_SQL.NATIVE);
  vDummy := DBMS_SQL.Execute(vCursorID);  -- return value not used for anything
  DBMS_SQL.Close_Cursor(vCursorID);
  RETURN (TRUE);
EXCEPTION
  WHEN OTHERS THEN
    DBMS_SQL.Close_Cursor(vCursorID);
    RAISE;
END;
```

使用DBMS_SQL，首先打开一个显式DDL游标，然后分析与执行语句，最后关闭游标。任何的异常会关闭游标并传递异常(关于异常传播和处理参见第12章)。这个过程非常普遍，可以在客户端或服务器端作为存储函数执行。

第11章介绍了PL/SQL程序设计的基本内容：说明数据结构、编写基本过程代码和把 SQL以相当复杂方式集成到用户代码中。第12章将进入另外一个境界，用PL/SQL组织用户代码。